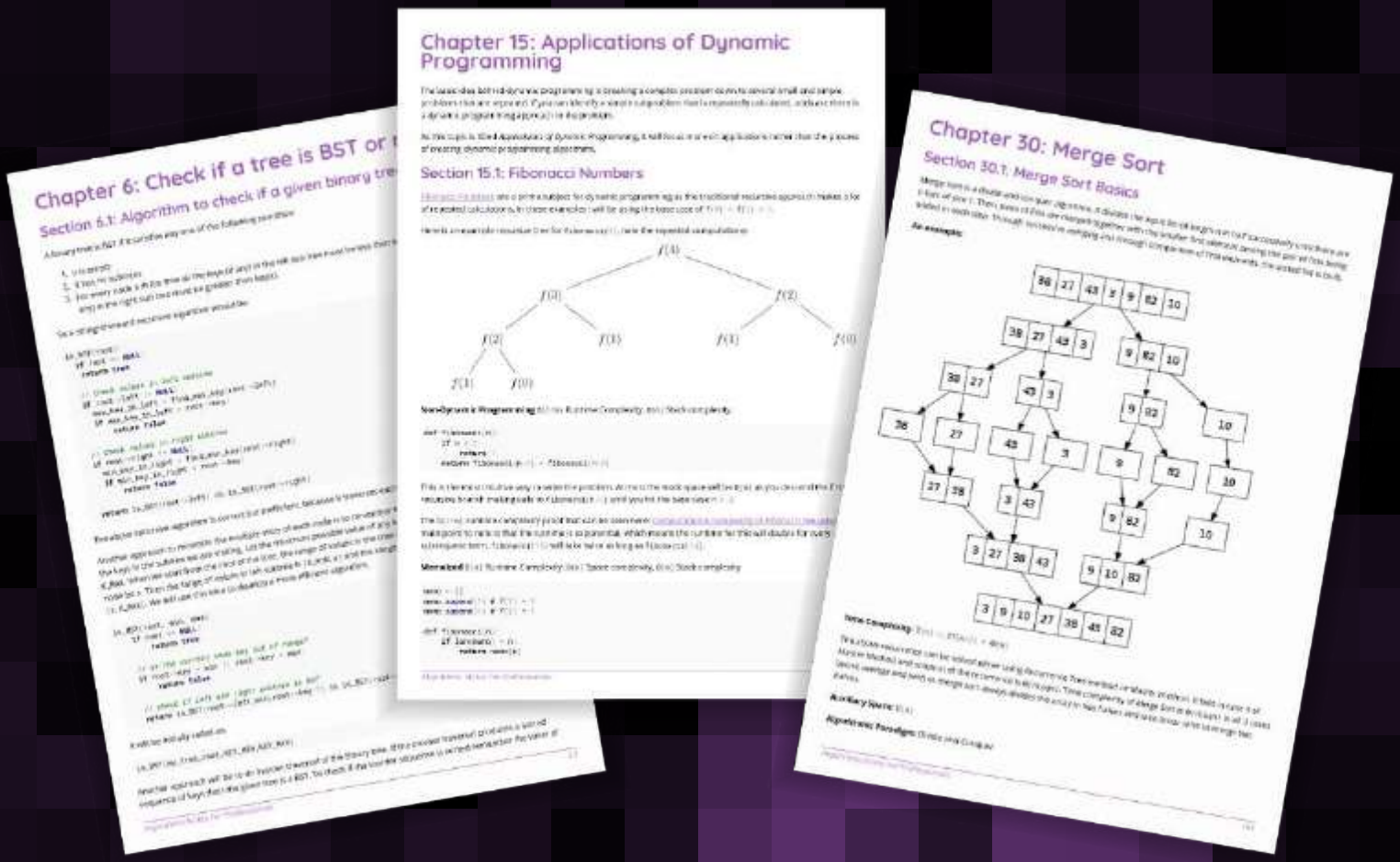
Algorithms

Notes for Professionals



200+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/AlgorithmsBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with algorithms

Section 1.1: A sample algorithmic problem

An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. The algorithmic *problem* known as *sorting* is defined as follows: [Skiena:2008:ADM:1410219]

- Problem: Sorting
- Input: A sequence of *n* keys, `a_1, a_2, ..., a_n`.
- Output: The reordering of the input sequence such that `a'_1 <= a'_2 <= ... <= a'_{n-1} <= a'_n`

An *instance* of sorting might be an array of strings, such as `{ Haskell, Emacs }` or a sequence of numbers such as `{ 154, 245, 1337 }`.

Section 1.2: Getting Started with Simple Fizz Buzz Algorithm in Swift

For those of you that are new to programming in Swift and those of you coming from different programming bases, such as Python or Java, this article should be quite helpful. In this post, we will discuss a simple solution for implementing swift algorithms.

Fizz Buzz

You may have seen Fizz Buzz written as Fizz Buzz, FizzBuzz, or Fizz-Buzz; they're all referring to the same thing. That "thing" is the main topic of discussion today. First, what is FizzBuzz?

This is a common question that comes up in job interviews.

Imagine a series of a number from 1 to 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz and Buzz refer to any number that's a multiple of 3 and 5 respectively. In other words, if a number is divisible by 3, it is substituted with fizz; if a number is divisible by 5, it is substituted with buzz. If a number is simultaneously a multiple of 3 AND 5, the number is replaced with "fizz buzz." In essence, it emulates the famous children game "fizz buzz".

To work on this problem, open up Xcode to create a new playground and initialize an array like below:

```
// for example
let number  = [1,2,3,4,5]
// here 3 is fizz and 5 is buzz
```

To find all the fizz and buzz, we must iterate through the array and check which numbers are fizz and which are buzz. To do this, create a for loop to iterate through the array we have initialised:

```
for num in number {
  // Body and calculation goes here
}
```

After this, we can simply use the "if else" condition and module operator in swift ie - % to locate the fizz and buzz

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else {
    print(num)
  }
}
```

Great! You can go to the debug console in Xcode playground to see the output. You will find that the "fizzes" have been sorted out in your array.

For the Buzz part, we will use the same technique. Let's give it a try before scrolling through the article — you can check your results against this article once you've finished doing this.

```
for num in number {
  if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Check the output!

It's rather straight forward — you divided the number by 3, fizz and divided the number by 5, buzz. Now, increase the numbers in the array

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

We increased the range of numbers from 1-10 to 1-15 in order to demonstrate the concept of a "fizz buzz." Since 15 is a multiple of both 3 and 5, the number should be replaced with "fizz buzz." Try for yourself and check the answer!

Here is the solution:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Wait...it's not over though! The whole purpose of the algorithm is to customize the runtime correctly. Imagine if the range increases from 1-15 to 1-100. The compiler will check each number to determine whether it is divisible by 3 or 5. It would then run through the numbers again to check if the numbers are divisible by 3 and 5. The code would essentially have to run through each number in the array twice — it would have to runs the numbers by 3 first and then run it by 5. To speed up the process, we can simply tell our code to divide the numbers by 15 directly.

Here is the final code:

```
for num in number {
```

```
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

As Simple as that, you can use any language of your choice and get started

Enjoy Coding

Chapter 2: Algorithm Complexity

Section 2.1: Big-Theta notation

Unlike Big-O notation, which represents only upper bound of the running time for some algorithm, Big-Theta is a tight bound; both upper and lower bound. Tight bound is more precise, but also more difficult to compute.

The Big-Theta notation is symmetric: `f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))`

An intuitive way to grasp it is that `f(x) = Θ(g(x))` means that the graphs of f(x) and g(x) grow in the same rate, or that the graphs 'behave' similarly for big enough values of x.

The full mathematical expression of the Big-Theta notation is as follows:
Θ(f(x)) = {g: N0 -> R and c1, c2, n0 > 0, where c1 < abs(g(n) / f(n)), for every n > n0 and abs is the absolute value }

An example

If the algorithm for the input `n` takes `42n^2 + 25n + 4` operations to finish, we say that is `O(n^2)`, but is also `O(n^3)` and `O(n^100)`. However, it is `Θ(n^2)` and it is not `Θ(n^3)`, `Θ(n^4)` etc. Algorithm that is `Θ(f(n))` is also `O(f(n))`, but not vice versa!

Formal mathematical definition

`Θ(g(x))` is a set of functions.

```
Θ(g(x)) = {f(x) such that there exist positive constants c1, c2, N such that 0 <= c1*g(x) <= f(x) <= c2*g(x) for all x > N}
```

Because `Θ(g(x))` is a set, we could write `f(x) ∈ Θ(g(x))` to indicate that `f(x)` is a member of `Θ(g(x))`. Instead, we will usually write `f(x) = Θ(g(x))` to express the same notion - that's the common way.

Whenever `Θ(g(x))` appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example the equation `T(n) = T(n/2) + Θ(n)`, means `T(n) = T(n/2) + f(n)` where `f(n)` is a function in the set `Θ(n)`.

Let `f` and `g` be two functions defined on some subset of the real numbers. We write `f(x) = Θ(g(x))` as `x->infinity` if and only if there are positive constants `K` and `L` and a real number `x0` such that holds:

`K|g(x)| <= f(x) <= L|g(x)|` for all `x >= x0`.

The definition is equal to:

`f(x) = O(g(x)) and f(x) = Ω(g(x))`

A method that uses limits

if `limit(x->infinity) f(x)/g(x) = c ∈ (0,∞)` i.e. the limit exists and it's positive, then `f(x) = Θ(g(x))`

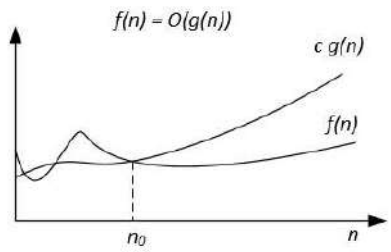
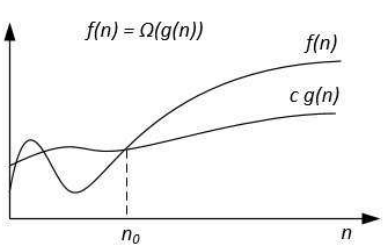
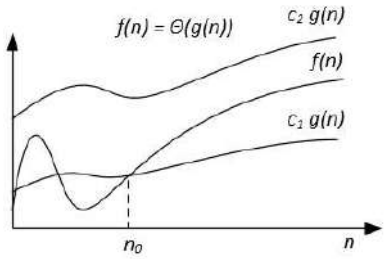
Common Complexity Classes

Name	Notation	n = 10	n = 100
Constant	Θ(1)	1	1
Logarithmic	Θ(log(n))	3	7
Linear	Θ(n)	10	100

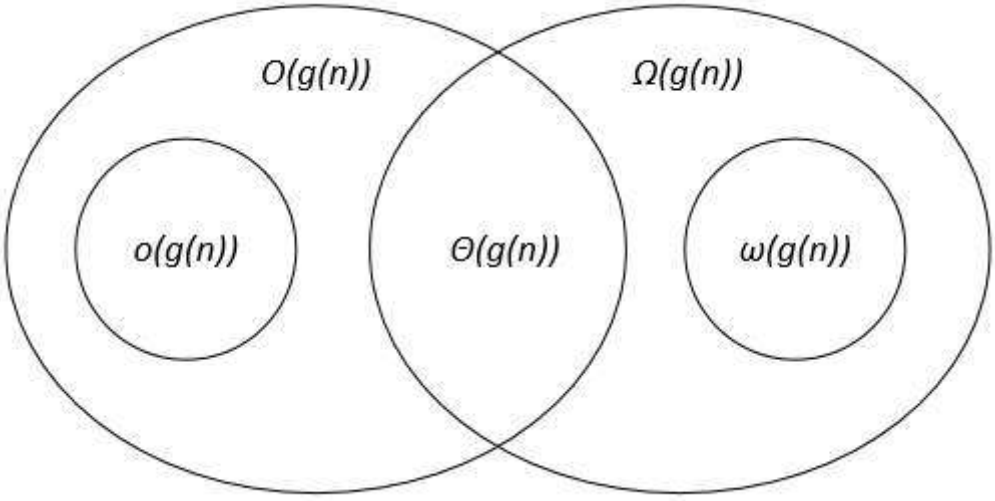
Linearithmic	Θ(n*log(n))	30	700
Quadratic	Θ(n^2)	100	10 000
Exponential	Θ(2^n)	1 024	1.267650e+ 30
Factorial	Θ(n!)	3 628 800	9.332622e+157

Section 2.2: Comparison of the asymptotic notations

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers, `c`, `c1`, `c2`, `n0` are positive real constants.

Notation	f(n) = O(g(n))	f(n) = Ω(g(n))	f(n) = Θ(g(n))	f(n) = o(g(n))	f(n) = ω(g(n))
Formal definition	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq f(n) \leq c\ g(n)$	$\exists\ c > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c\ g(n) \leq f(n)$	$\exists\ c1, c2 > 0, \exists\ n0 > 0 : \forall\ n \geq n0, 0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0\ 0 : \forall\ n \geq n0, 0 \leq f(n) < c\ g(n)$	$\forall\ c > 0, \exists\ n0 > 0\ 0 : \forall\ n \geq n0, 0 \leq c\ g(n) < f(n)$
Analogy between the asymptotic comparison of `f`, `g` and real numbers `a`, `b`	$a \leq b$	$a \geq b$	$a = b$	$a < b$	$a > b$
Example	$7n + 10 = O(n^2 + n - 9)$	$n^3 - 34 = \Omega(10n^2 - 7n + 1)$	$1/2\ n^2 - 7n = \Theta(n^2)$	$5n^2 = o(n^3)$	$7n^2 = \omega(n)$
Graphic interpretation					

The asymptotic notations can be represented on a Venn diagram as follows:



Links

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

Section 2.3: Big-Omega Notation

Ω-notation is used for asymptotic lower bound.

Formal definition

Let `f(n)` and `g(n)` be two functions defined on the set of the positive real numbers. We write `f(n) = Ω(g(n))` if there are positive constants `c` and `n0` such that:

`0 ≤ c g(n) ≤ f(n) for all n ≥ n0`.

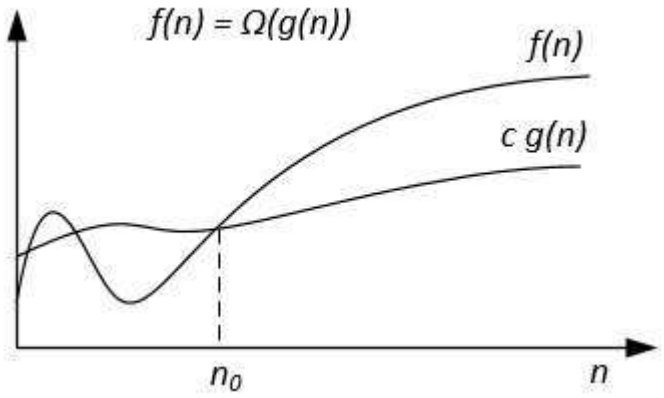
Notes

`f(n) = Ω(g(n))` means that `f(n)` grows asymptotically no slower than `g(n)`. Also we can say about `Ω(g(n))` when algorithm analysis is not enough for statement about `Θ(g(n))` or / and `O(g(n))`.

From the definitions of notations follows the theorem:

For two any functions `f(n)` and `g(n)` we have `f(n) = Θ(g(n))` if and only if `f(n) = O(g(n)) and f(n) = Ω(g(n))`.

Graphically Ω-notation may be represented as follows:



For example lets we have `f(n) = 3n^2 + 5n - 4`. Then `f(n) = Ω(n^2)`. It is also correct `f(n) = Ω(n)`, or even `f(n) = Ω(1)`.

Another example to solve perfect matching algorithm : If the number of vertices is odd then output "No Perfect Matching" otherwise try all possible matchings.

We would like to say the algorithm requires exponential time but in fact you cannot prove a `Ω(n^2)` lower bound using the usual definition of Ω since the algorithm runs in linear time for n odd. We should instead define `f(n)=Ω(g(n))` by saying for some constant `c>0`, `f(n)≥ c g(n)` for infinitely many `n`. This gives a nice correspondence between upper and lower bounds: `f(n)=Ω(g(n))` iff `f(n) != o(g(n))`.

References

Formal definition and theorem are taken from the book "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms".

Chapter 3: Big-O Notation

Definition

The Big-O notation is at its heart a mathematical notation, used to compare the rate of convergence of functions. Let `n -> f(n)` and `n -> g(n)` be functions defined over the natural numbers. Then we say that `f = O(g)` if and only if `f(n)/g(n)` is bounded when n approaches infinity. In other words, `f = O(g)` if and only if there exists a constant A, such that for all n, `f(n)/g(n) <= A`.

Actually the scope of the Big-O notation is a bit wider in mathematics but for simplicity I have narrowed it to what is used in algorithm complexity analysis : functions defined on the naturals, that have non-zero values, and the case of n growing to infinity.

What does it mean ?

Let's take the case of `f(n) = 100n^2 + 10n + 1` and `g(n) = n^2`. It is quite clear that both of these functions tend to infinity as n tends to infinity. But sometimes knowing the limit is not enough, and we also want to know the *speed* at which the functions approach their limit. Notions like Big-O help compare and classify functions by their speed of convergence.

Let's find out if `f = O(g)` by applying the definition. We have `f(n)/g(n) = 100 + 10/n + 1/n^2`. Since `10/n` is 10 when n is 1 and is decreasing, and since `1/n^2` is 1 when n is 1 and is also decreasing, we have `f(n)/g(n) <= 100 + 10 + 1 = 111`. The definition is satisfied because we have found a bound of `f(n)/g(n)` (111) and so `f = O(g)` (we say that f is a Big-O of `n^2`).

This means that f tends to infinity at approximately the same speed as g. Now this may seem like a strange thing to say, because what we have found is that f is at most 111 times bigger than g, or in other words when g grows by 1, f grows by at most 111. It may seem that growing 111 times faster is not "approximately the same speed". And indeed the Big-O notation is not a very precise way to classify function convergence speed, which is why in mathematics we use the equivalence relationship when we want a precise estimation of speed. But for the purposes of separating algorithms in large speed classes, Big-O is enough. We don't need to separate functions that grow a fixed number of times faster than each other, but only functions that grow *infinitely* faster than each other. For instance if we take `h(n) = n^2*log(n)`, we see that `h(n)/g(n) = log(n)` which tends to infinity with n so h is *not* O(n^2), because h grows *infinitely* faster than n^2.

Now I need to make a side note : you might have noticed that if `f = O(g)` and `g = O(h)`, then `f = O(h)`. For instance in our case, we have `f = O(n^3)`, and `f = O(n^4)`... In algorithm complexity analysis, we frequently say `f = O(g)` to mean that `f = O(g)` *and* `g = O(f)`, which can be understood as "g is the smallest Big-O for f". In mathematics we say that such functions are Big-Thetas of each other.

How is it used ?

When comparing algorithm performance, we are interested in the number of operations that an algorithm performs. This is called *time complexity*. In this model, we consider that each basic operation (addition, multiplication, comparison, assignment, etc.) takes a fixed amount of time, and we count the number of such operations. We can usually express this number as a function of the size of the input, which we call n. And sadly, this number usually grows to infinity with n (if it doesn't, we say that the algorithm is O(1)). We separate our algorithms in big speed classes defined by Big-O : when we speak about a "O(n^2) algorithm", we mean that the number of operations it performs, expressed as a function of n, is a O(n^2). This says that our algorithm is approximately as fast as an algorithm that would do a number of operations equal to the square of the size of its input, *or faster*. The "or faster" part is there because I used Big-O instead of Big-Theta, but usually people will say Big-O to mean Big-Theta.

When counting operations, we usually consider the worst case: for instance if we have a loop that can run at most n times and that contains 5 operations, the number of operations we count is 5n. It is also possible to consider the average case complexity.

Quick note : a fast algorithm is one that performs few operations, so if the number of operations grows to infinity *faster*, then the algorithm is *slower*: O(n) is better than O(n^2).

We are also sometimes interested in the *space complexity* of our algorithm. For this we consider the number of bytes in memory occupied by the algorithm as a function of the size of the input, and use Big-O the same way.

Section 3.1: A Simple Loop

The following function finds the maximal element in an array:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

The input size is the size of the array, which I called `len` in the code.

Let's count the operations.

```
int max = INT_MIN;
size_t i = 0;
```

These two assignments are done only once, so that's 2 operations. The operations that are looped are:

```
if (max < array[i])
i++;
max = array[i]
```

Since there are 3 operations in the loop, and the loop is done n times, we add `3n` to our already existing 2 operations to get `3n + 2`. So our function takes `3n + 2` operations to find the max (its complexity is `3n + 2`). This is a polynomial where the fastest growing term is a factor of n, so it is O(n).

You probably have noticed that "operation" is not very well defined. For instance I said that `if (max < array[i])` was one operation, but depending on the architecture this statement can compile to for instance three instructions : one memory read, one comparison and one branch. I have also considered all operations as the same, even though for instance the memory operations will be slower than the others, and their performance will vary wildly due for instance to cache effects. I also have completely ignored the return statement, the fact that a frame will be created for the function, etc. In the end it doesn't matter to complexity analysis, because whatever way I choose to count operations, it will only change the coefficient of the n factor and the constant, so the result will still be O(n). Complexity shows how the algorithm scales with the size of the input, but it isn't the only aspect of performance!

Section 3.2: A Nested Loop

The following function checks if an array has any duplicates by taking each element, then iterating over the whole array to see if the element is there

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

The inner loop performs at each iteration a number of operations that is constant with `n`. The outer loop also does a few constant operations, and runs the inner loop `n` times. The outer loop itself is run `n` times. So the operations inside the inner loop are run `n^2` times, the operations in the outer loop are run `n` times, and the assignment to `i` is done one time. Thus, the complexity will be something like `an^2 + bn + c`, and since the highest term is `n^2`, the O notation is `O(n^2)`.

As you may have noticed, we can improve the algorithm by avoiding doing the same comparisons multiple times. We can start from `i + 1` in the inner loop, because all elements before it will already have been checked against all array elements, including the one at index `i + 1`. This allows us to drop the `i == j` check.

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Obviously, this second version does less operations and so is more efficient. How does that translate to Big-O notation? Well, now the inner loop body is run `1 + 2 + ... + n - 1 = n(n-1)/2` times. This is *still* a polynomial of the second degree, and so is still only `O(n^2)`. We have clearly lowered the complexity, since we roughly divided by 2 the number of operations that we are doing, but we are still in the same complexity *class* as defined by Big-O. In order to lower the complexity to a lower class we would need to divide the number of operations by something that *tends to infinity* with `n`.

Section 3.3: O(log n) types of Algorithms

Let's say we have a problem of size n. Now for each step of our algorithm(which we need write), our original problem becomes half of its previous size(n/2).

So at each step, our problem becomes half.

Step	Problem
1	n/2
2	n/4
3	n/8
4	n/16

When the problem space is reduced(i.e solved completely), it cannot be reduced any further(n becomes equal to 1) after exiting check condition.

1. Let's say at kth step or number of operations:

   problem-size = 1

2. But we know at kth step, our problem-size should be:

   problem-size = n/2k

3. From 1 and 2:

   n/2k = 1 or

   n = 2k

4. Take log on both sides

   loge n = k loge2

   or

   k = loge n / loge 2

5. Using formula **logx m / logx n = logn m**

   k = log2 n

   or simply k = log n

> Now we know that our algorithm can run maximum up to log n, hence time complexity comes as O(log n)

A very simple example in code to support above text is :

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

So now if some one asks you if n is 256 how many steps that loop(or any other algorithm that cuts down it's problem size into half) will run you can very easily calculate.

k = log2 256

k = log2 2 8 (=> logaa = 1)

k = 8

Another very good example for similar case is **Binary Search Algorithm**.

```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return –1;// Unsuccessful result
}
```

Section 3.4: An O(log n) example

Introduction

Consider the following problem:

`L` is a sorted list containing `n` signed integers (`n` being big enough), for example `[-5, -2, -1, 0, 1, 2, 4]` (here, `n` has a value of 7). If `L` is known to contain the integer 0, how can you find the index of 0 ?

Naïve approach

The first thing that comes to mind is to just read every index until 0 is found. In the worst case, the number of operations is `n`, so the complexity is O(n).

This works fine for small values of `n`, but is there a more efficient way ?

Dichotomy

Consider the following algorithm (Python3):

```
a = 0
b = n-1
while True:
  h = (a+b)//2 ## // is the integer division, so h is an integer
  if L[h] == 0:
    return h
  elif L[h] > 0:
    b = h
  elif L[h] < 0:
    a = h
```

`a` and `b` are the indexes between which 0 is to be found. Each time we enter the loop, we use an index between `a` and `b` and use it to narrow the area to be searched.

In the worst case, we have to wait until `a` and `b` are equal. But how many operations does that take? Not n, because each time we enter the loop, we divide the distance between `a` and `b` by about two. Rather, the complexity is O(log n).

Explanation

Note: When we write "log", we mean the binary logarithm, or log base 2 (which we will write "log_2"). As O(log_2 n) = O(log n) (you can do the math) we will use "log" instead of "log_2".

Let's call x the number of operations: we know that 1 = n / (2^x).

So 2^x = n, then x = log n

Conclusion

When faced with successive divisions (be it by two or by any number), remember that the complexity is logarithmic.

Chapter 4: Trees

Section 4.1: Typical anary tree representation

Typically we represent an anary tree (one with potentially unlimited children per node) as a binary tree, (one with exactly two children per node). The "next" child is regarded as a sibling. Note that if a tree is binary, this representation creates extra nodes.

We then iterate over the siblings and recurse down the children. As most trees are relatively shallow - lots of children but only a few levels of hierarchy, this gives rise to efficient code. Note human genealogies are an exception (lots of levels of ancestors, only a few children per level).

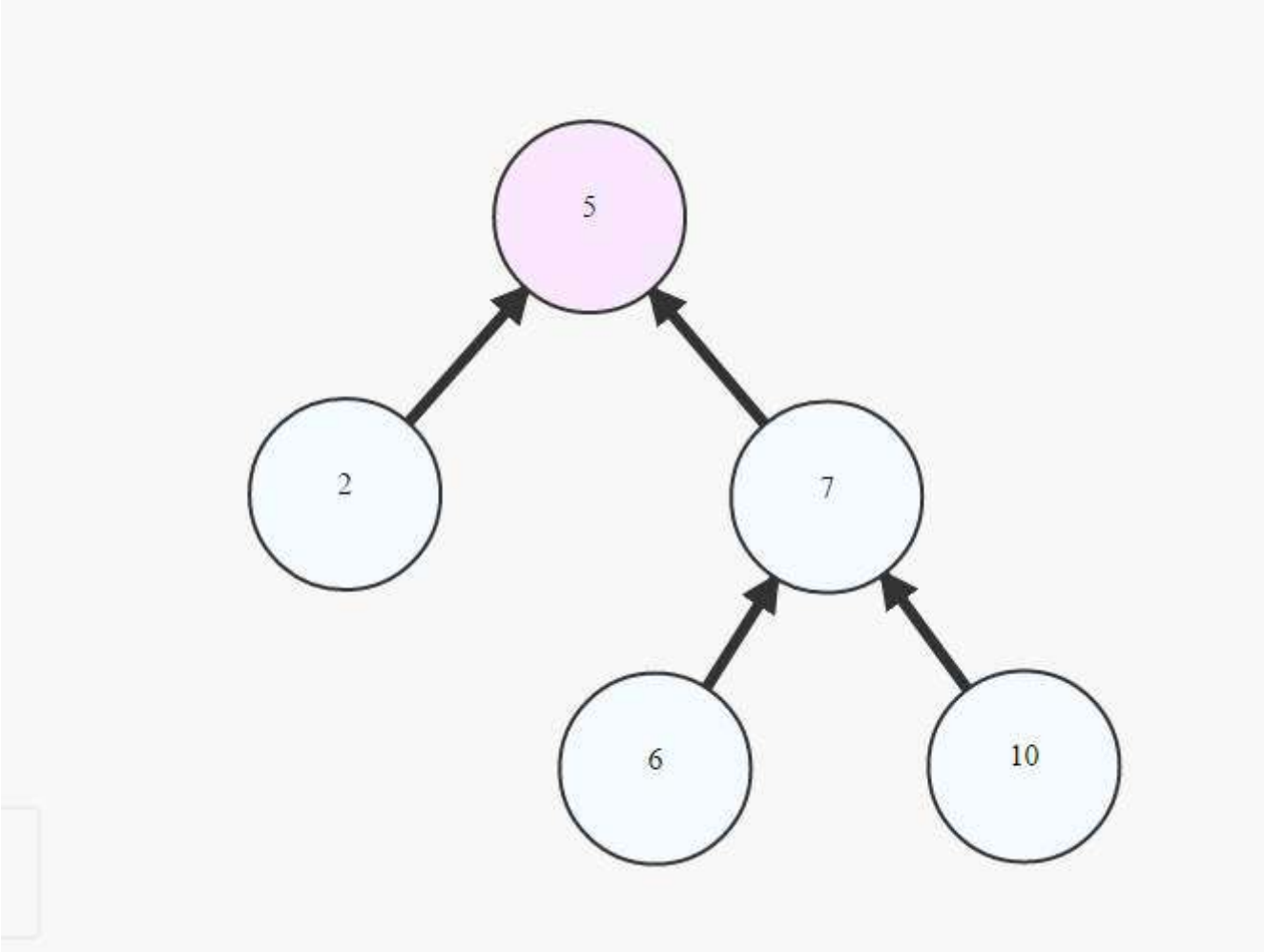
If necessary back pointers can be kept to allow the tree to be ascended. These are more difficult to maintain.

Note that it is typical to have one function to call on the root and a recursive function with extra parameters, in this case tree depth.

```
struct node
{
   struct node *next;
   struct node *child;
   std::string data;
}

void printtree_r(struct node *node, int depth)
{
   int i;

   while(node)
   {
       if(node->child)
       {
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):
          printtree_r(node->child, depth +1);
          for(i=0;i<depth*3;i++)
              printf(" ");
          printf("{\n"):

          for(i=0;i<depth*3;i++)
             printf(" ");
           printf("%s\n", node->data.c_str());

           node = node->next;
        }
    }
}

void printtree(node *root)
{
   printree_r(root, 0);
}
```

Section 4.2: Introduction

Trees are a sub-type of the more general node-edge graph data structure.



To be a tree, a graph must satisfy two requirements:

- **It is acyclic.** It contains no cycles (or "loops").
- **It is connected.** For any given node in the graph, every node is reachable. All nodes are reachable through one path in the graph.

The tree data structure is quite common within computer science. Trees are used to model many different algorithmic data structures, such as ordinary binary trees, red-black trees, B-trees, AB-trees, 23-trees, Heap, and tries.

it is common to refer to a Tree as a `Rooted Tree` by:

```
choosing 1 cell to be called `Root`
painting the `Root` at the top
creating lower layer for each cell in the graph depending on their distance from the root -the
bigger the distance, the lower the cells (example above)
```

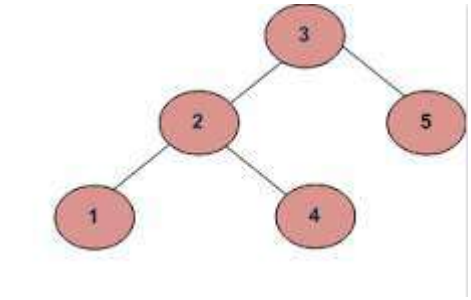
common symbol for trees: `T`

Section 4.3: To check if two Binary trees are same or not

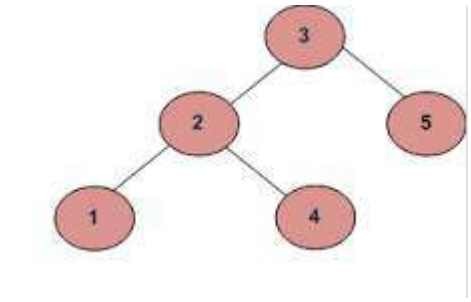
1. For example if the inputs are:

Example:1

a)



b)

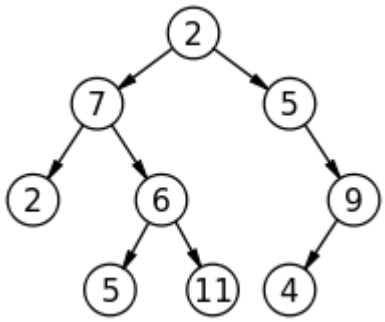


Output should be true.

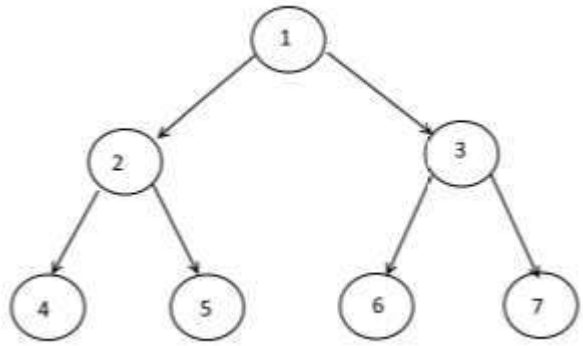
Example:2

If the inputs are:

a)



b)



Output should be false.

Pseudo code for the same:

```
boolean sameTree(node root1, node root2){
```

```
if(root1 == NULL && root2 == NULL)
return true;

if(root1 == NULL || root2 == NULL)
return false;

if(root1->data == root2->data
     && sameTree(root1->left,root2->left)
        && sameTree(root1->right, root2->right))
return true;

}
```

Chapter 5: Binary Search Trees

Binary tree is a tree that each node in it has maximum of two children. Binary search tree (BST) is a binary tree which its elements positioned in special order. In each BST all values(i.e key) in left sub tree are less than values in right sub tree.

Section 5.1: Binary Search Tree - Insertion (Python)

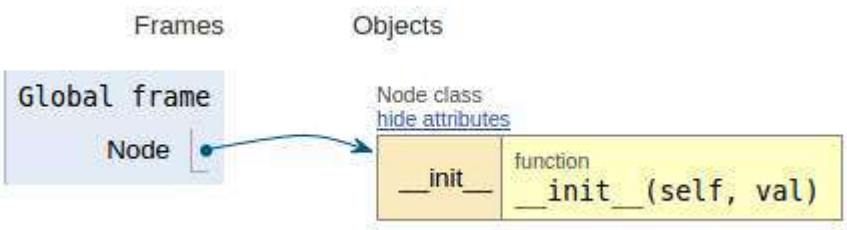
This is a simple implementation of Binary Search Tree Insertion using Python.

An example is shown below:

www.penjee.com

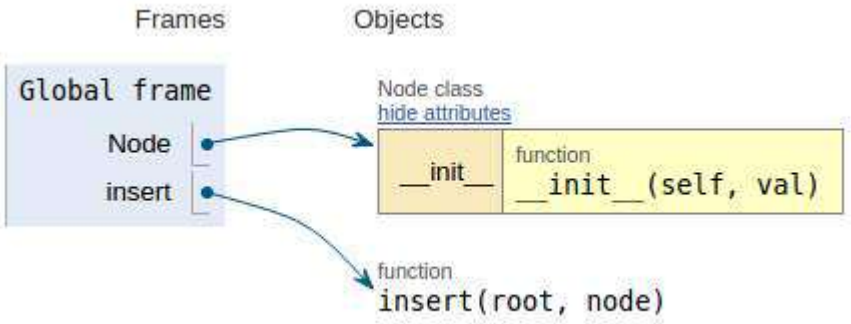
Following the code snippet each image shows the execution visualization which makes it easier to visualize how this code works.

```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```

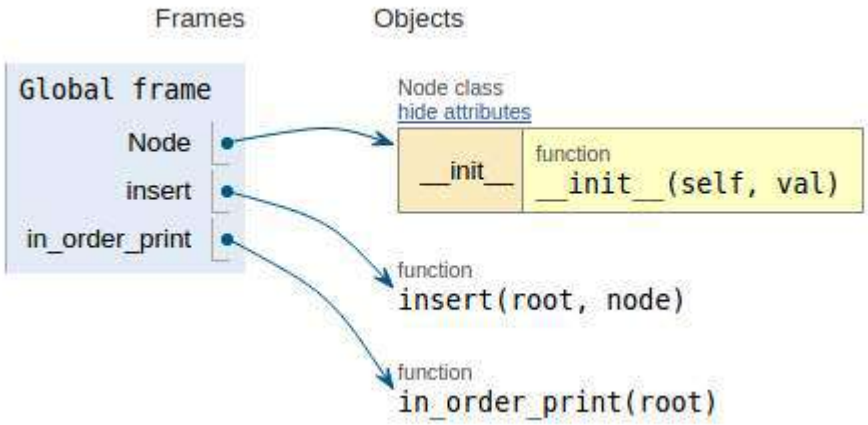



```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
```

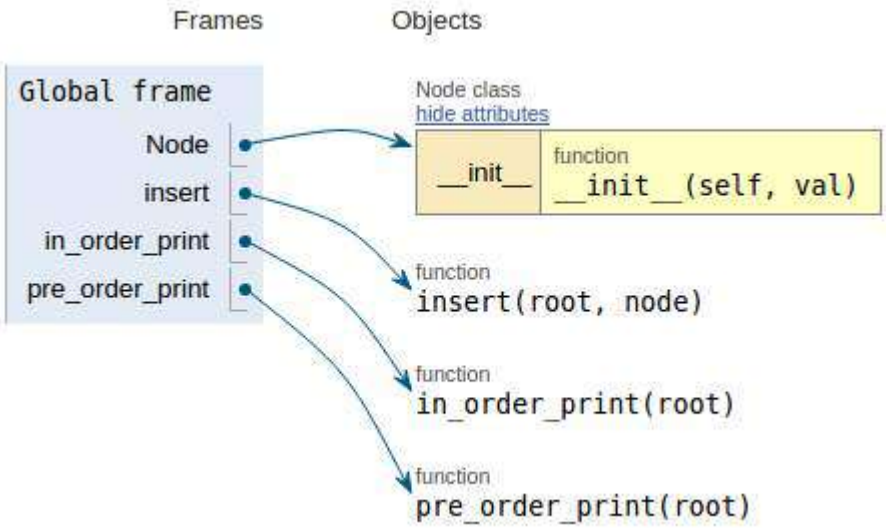
```
            root.r_child = node
        else:
            insert(root.r_child, node)
```




```
def in_order_print(root):
    if not root:
        return
    in_order_print(root.l_child)
    print root.data
    in_order_print(root.r_child)
```



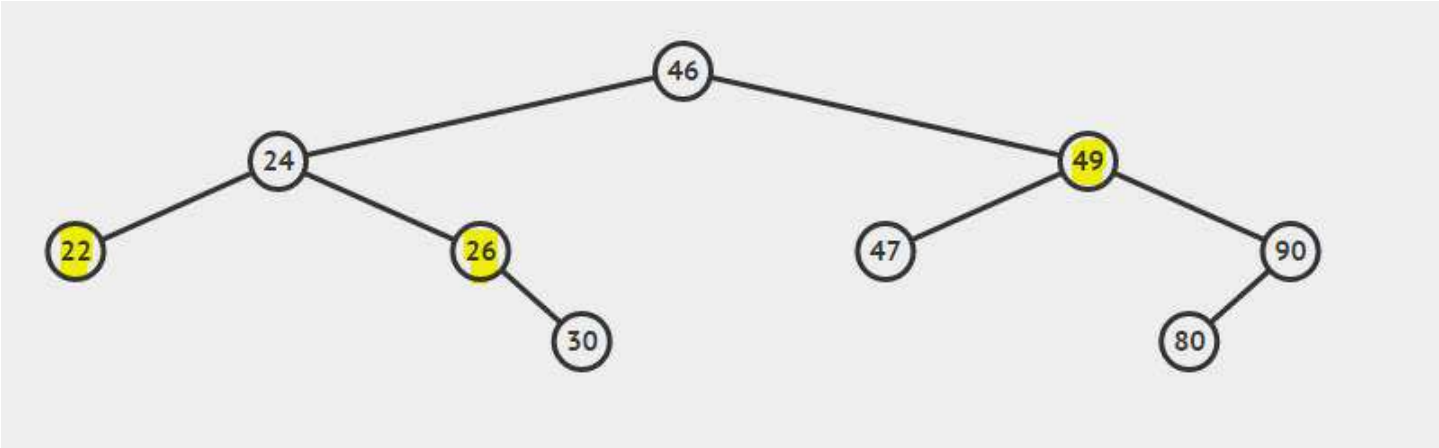

```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



Section 5.2: Binary Search Tree - Deletion(C++)

Before starting with deletion I just want to put some lights on what is a Binary search tree(BST), Each node in a BST can have maximum of two nodes(left and right child).The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key.

Deleting a node in a tree while maintaining its **Binary search tree property.**



There are three cases to be considered while deleting a node.

- Case 1: Node to be deleted is the leaf node.(Node with value 22).
- Case 2: Node to be deleted has one child.(Node with value 26).
- Case 3: Node to be deleted has both children.(Node with value 49).

Explanation of cases:

1. When the node to be deleted is a leaf node then simply delete the node and pass `nullptr` to its parent node.
2. When a node to be deleted is having only one child then copy the child value to the node value and delete the child **(Converted to case 1)**
3. When a node to be delete is having two childs then the minimum from its right sub tree can be copied to the node and then the minimum value can be deleted from the node's right subtree **(Converted to Case 2)**

Note: The minimum in the right sub tree can have a maximum of one child and that too right child if it's having the left child that means it's not the minimum value or it's not following BST property.

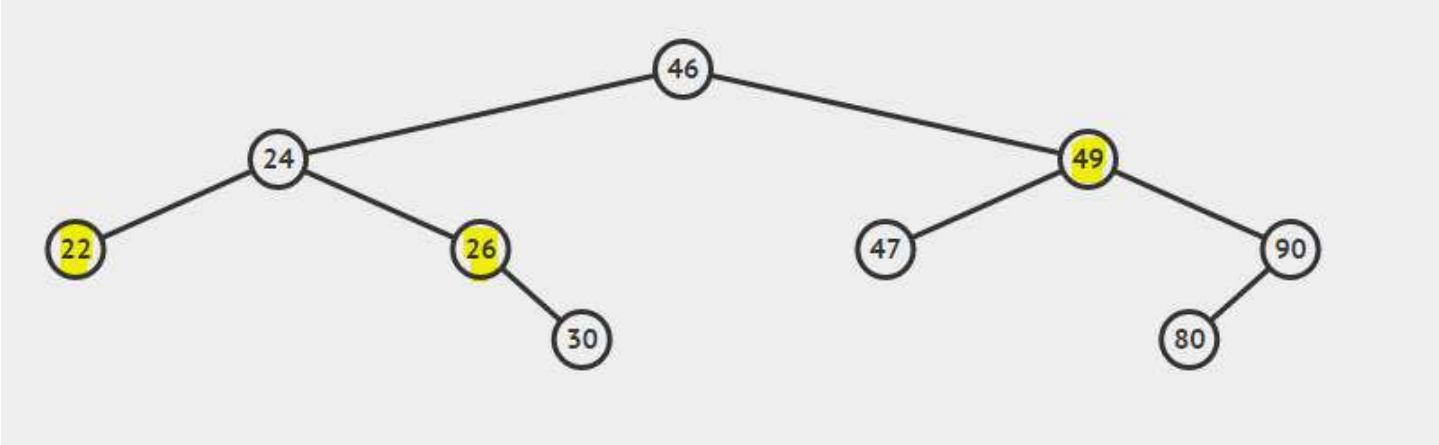
The structure of a node in a tree and the code for Deletion:

```
struct node
{
    int data;
    node *left, *right;
};

node* delete_node(node *root, int data)
{
  if(root == nullptr) return root;
  else if(data < root->data) root->left  = delete_node(root->left, data);
  else if(data > root->data) root->right = delete_node(root->right, data);

  else
  {
    if(root->left == nullptr && root->right == nullptr) // Case 1
    {
      free(root);
      root = nullptr;
    }
    else if(root->left == nullptr)       // Case 2
    {
       node* temp = root;
       root= root->right;
       free(temp);
    }
    else if(root->right == nullptr)      // Case 2
    {
       node* temp = root;
       root = root->left;
       free(temp);
    }
    else                                 // Case 3
    {
       node* temp = root->right;

       while(temp->left != nullptr) temp = temp->left;

       root->data = temp->data;
       root->right = delete_node(root->right, temp->data);
    }
  }
  return root;
}
```

Time complexity of above code is O(*h*), where *h* is the height of the tree.

Section 5.3: Lowest common ancestor in a BST

Consider the BST:



Lowest common ancestor of 22 and 26 is 24

Lowest common ancestor of 26 and 49 is 46

Lowest common ancestor of 22 and 24 is 24

Binary search tree property can be used for finding nodes lowest ancestor

Psuedo code:

```
lowestCommonAncestor(root,node1, node2){

if(root == NULL)
return NULL;



 else if(node1->data == root->data || node2->data== root->data)
    return root;

    else if((node1->data <= root->data && node2->data > root->data)
              || (node2->data <= root->data && node1->data > root->data)){

             return root;
    }

    else if(root->data > max(node1->data,node2->data)){
            return lowestCommonAncestor(root->left, node1, node2);
    }

        else {
            return lowestCommonAncestor(root->right, node1, node2);
      }
        }
```

Section 5.4: Binary Search Tree - Python

```
class Node(object):
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.val = val


class BinarySearchTree(object):
    def insert(self, root, node):
```

```
        if root is None:
            return node

        if root.val < node.val:
            root.r_child = self.insert(root.r_child, node)
        else:
            root.l_child = self.insert(root.l_child, node)

        return root


    def in_order_place(self, root):
        if not root:
            return None
        else:
            self.in_order_place(root.l_child)
            print root.val
            self.in_order_place(root.r_child)

    def pre_order_place(self, root):
        if not root:
            return None
        else:
            print root.val
            self.pre_order_place(root.l_child)
            self.pre_order_place(root.r_child)

    def post_order_place(self, root):
        if not root:
            return None
        else:
            self.post_order_place(root.l_child)
            self.post_order_place(root.r_child)
            print root.val
```

""" Create different node and insert data into it"""

```
r = Node(3)
node = BinarySearchTree()
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8]

for nd in nodeList:
    node.insert(r, Node(nd))

print "------In order ---------"
print (node.in_order_place(r))
print "------Pre order ---------"
print (node.pre_order_place(r))
print "------Post order ---------"
print (node.post_order_place(r))
```

Chapter 6: Check if a tree is BST or not

Section 6.1: Algorithm to check if a given binary tree is BST

A binary tree is BST if it satisfies any one of the following condition:

1. It is empty
2. It has no subtrees
3. For every node x in the tree all the keys (if any) in the left sub tree must be less than key(x) and all the keys (if any) in the right sub tree must be greater than key(x).

So a straightforward recursive algorithm would be:

```
is_BST(root):
  if root == NULL:
   return true

  // Check values in left subtree
  if root->left != NULL:
    max_key_in_left = find_max_key(root->left)
    if max_key_in_left > root->key:
        return false

  // Check values in right subtree
  if root->right != NULL:
    min_key_in_right = find_min_key(root->right)
    if min_key_in_right < root->key:
        return false

  return is_BST(root->left) && is_BST(root->right)
```

The above recursive algorithm is correct but inefficient, because it traverses each node mutiple times.

Another approach to minimize the multiple visits of each node is to remember the min and max possible values of the keys in the subtree we are visiting. Let the minimum possible value of any key be `K_MIN` and maximum value be `K_MAX`. When we start from the root of the tree, the range of values in the tree is `[K_MIN,K_MAX]`. Let the key of root node be `x`. Then the range of values in left subtree is `[K_MIN,x)` and the range of values in right subtree is `(x,K_MAX]`. We will use this idea to develop a more efficient algorithm.

```
is_BST(root, min, max):
    if root == NULL:
        return true

    // is the current node key out of range?
    if root->key < min || root->key > max:
        return false

    // check if left and right subtree is BST
    return is_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)
```

It will be initially called as:

```
is_BST(my_tree_root,KEY_MIN,KEY_MAX)
```

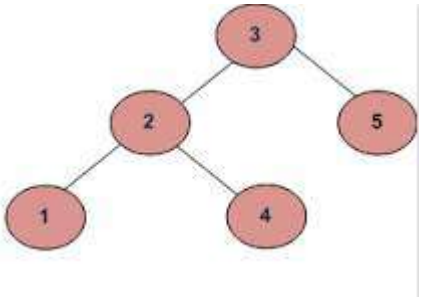
Another approach will be to do inorder traversal of the Binary tree. If the inorder traversal produces a sorted sequence of keys then the given tree is a BST. To check if the inorder sequence is sorted remember the value of

previously visited node and compare it against the current node.

Section 6.2: If a given input tree follows Binary search tree property or not

For example

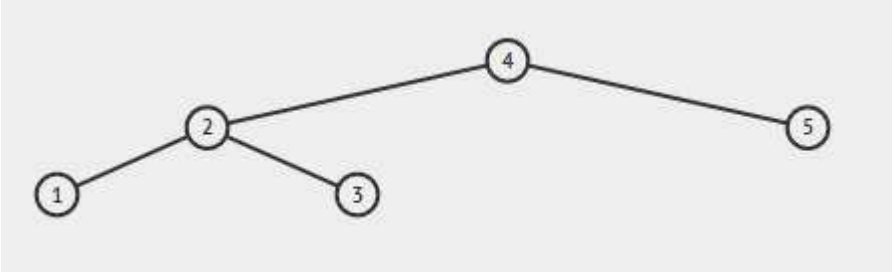
if the input is:



Output should be false:

As 4 in the left sub-tree is greater than the root value(3)

If the input is:



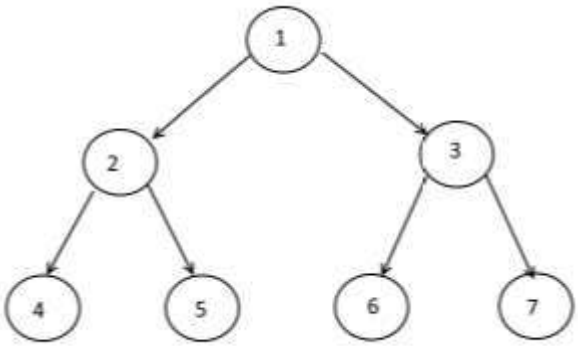
Output should be true

Chapter 7: Binary Tree traversals

Visiting a node of a binary tree in some particular order is called traversals.

Section 7.1: Level Order traversal - Implementation

For example if the given tree is:



Level order traversal will be

1 2 3 4 5 6 7

Printing node data level by level.

Code:

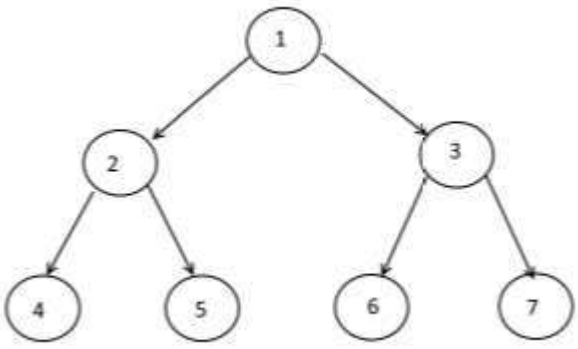
```
#include<iostream>
#include<queue>
#include<malloc.h>

using namespace std;

struct node{

    int data;
    node *left;
    node *right;
};

void levelOrder(struct node *root){

        if(root == NULL)    return;

        queue<node *> Q;
        Q.push(root);

        while(!Q.empty()){
        struct    node* curr = Q.front();
            cout<< curr->data <<" ";
            if(curr->left != NULL) Q.push(curr-> left);
                if(curr->right != NULL) Q.push(curr-> right);

                Q.pop();


        }
```

```
}
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main(){

    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left  = newNode(6);
    root->right->right = newNode(7);


    printf("Level Order traversal of binary tree is \n");
    levelOrder(root);

    return 0;


}
```

Queue data structure is used to achieve the above objective.

Section 7.2: Pre-order, Inorder and Post Order traversal of a Binary Tree

Consider the Binary Tree:



Pre-order traversal(root) is traversing the node then left sub-tree of the node and then the right sub-tree of the node.

So the pre-order traversal of above tree will be:

1 2 4 5 3 6 7

In-order traversal(root) is traversing the left sub-tree of the node then the node and then right sub-tree of the

node.

So the in-order traversal of above tree will be:

4 2 5 1 6 3 7

Post-order traversal(root) is traversing the left sub-tree of the node then the right sub-tree and then the node.

So the post-order traversal of above tree will be:

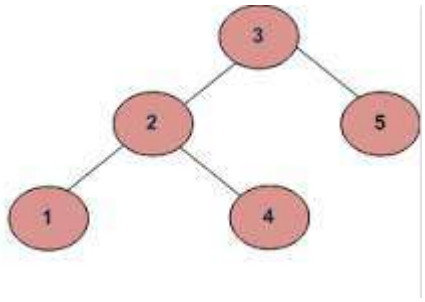
4 5 2 6 7 3 1

Chapter 8: Lowest common ancestor of a Binary Tree

Lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in the tree that has both n1 and n2 as descendants.

Section 8.1: Finding lowest common ancestor

Consider the tree:



Lowest common ancestor of nodes with value 1 and 4 is 2

Lowest common ancestor of nodes with value 1 and 5 is 3

Lowest common ancestor of nodes with value 2 and 4 is 4

Lowest common ancestor of nodes with value 1 and 2 is 2

Chapter 9: Graph

A graph is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are called graph vertices, "nodes" or simply "points." Similarly, the lines connecting the vertices of a graph are called graph edges, "arcs" or "lines."

A graph G can be defined as a pair (V,E), where V is a set of vertices, and E is a set of edges between the vertices $E \subseteq \{(u,v) \mid u, v \in V\}$.

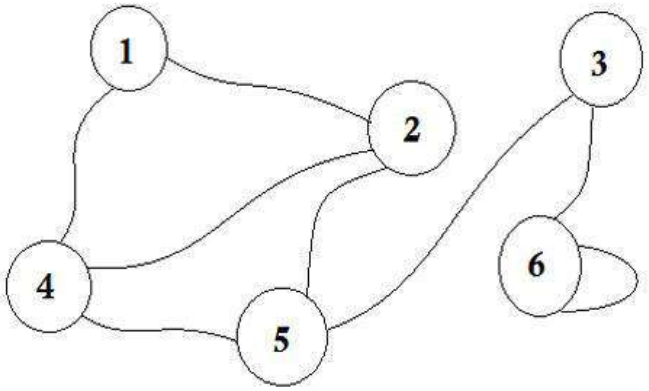
Section 9.1: Storing Graphs (Adjacency Matrix)

To store a graph, two methods are common:

- Adjacency Matrix
- Adjacency List

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

Adjacent means 'next to or adjoining something else' or to be beside something. For example, your neighbors are adjacent to you. In graph theory, if we can go to **node B** from **node A**, we can say that **node B** is adjacent to **node A**. Now we will learn about how to store which nodes are adjacent to which one via Adjacency Matrix. This means, we will represent which nodes share edge between them. Here matrix means 2D array.

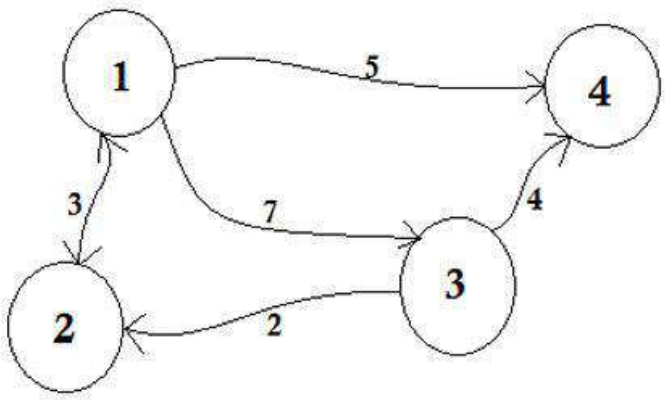


Node	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

Here you can see a table beside the graph, this is our adjacency matrix. Here **Matrix[i][j] = 1** represents there is an edge between **i** and **j**. If there's no edge, we simply put **Matrix[i][j] = 0**.

These edges can be weighted, like it can represent the distance between two cities. Then we'll put the value in **Matrix[i][j]** instead of putting 1.

The graph described above is *Bidirectional* or *Undirected*, that means, if we can go to **node 1** from **node 2**, we can also go to **node 2** from **node 1**. If the graph was *Directed*, then there would've been arrow sign on one side of the graph. Even then, we could represent it using adjacency matrix.



Node	1	2	3	4
1	Inf	3	7	5
2	3	Inf	Inf	Inf
3	Inf	2	Inf	4
4	Inf	Inf	Inf	Inf

We represent the nodes that don't share edge by *infinity*. One thing to be noticed is that, if the graph is undirected, the matrix becomes *symmetric*.

The pseudo-code to create the matrix:

```
Procedure AdjacencyMatrix(N):    //N represents the number of nodes
Matrix[N][N]
for i from 1 to N
    for j from 1 to N
        Take input -> Matrix[i][j]
    endfor
endfor
```

We can also populate the Matrix using this common way:

```
Procedure AdjacencyMatrix(N, E):    // N -> number of nodes
Matrix[N][E]                        // E -> number of edges
for i from 1 to E
    input -> n1, n2, cost
    Matrix[n1][n2] = cost
    Matrix[n2][n1] = cost
endfor
```

For directed graphs, we can remove **Matrix[n2][n1] = cost** line.

The drawbacks of using Adjacency Matrix:

Memory is a huge problem. No matter how many edges are there, we will always need N * N sized matrix where N is the number of nodes. If there are 10000 nodes, the matrix size will be 4 * 10000 * 10000 around 381 megabytes. This is a huge waste of memory if we consider graphs that have a few edges.

Suppose we want to find out to which node we can go from a node **u**. We'll need to check the whole row of **u**, which costs a lot of time.

The only benefit is that, we can easily find the connection between **u-v** nodes, and their cost using Adjacency Matrix.

Java code implemented using above pseudo-code:

```
import java.util.Scanner;

public class Represent_Graph_Adjacency_Matrix
{
    private final int vertices;
```

```
    private int[][] adjacency_matrix;

    public Represent_Graph_Adjacency_Matrix(int v)
    {
        vertices = v;
        adjacency_matrix = new int[vertices + 1][vertices + 1];
    }

    public void makeEdge(int to, int from, int edge)
    {
        try
        {
            adjacency_matrix[to][from] = edge;
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
    }

    public int getEdge(int to, int from)
    {
        try
        {
            return adjacency_matrix[to][from];
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
        return -1;
    }

    public static void main(String args[])
    {
        int v, e, count = 1, to = 0, from = 0;
        Scanner sc = new Scanner(System.in);
        Represent_Graph_Adjacency_Matrix graph;
        try
        {
            System.out.println("Enter the number of vertices: ");
            v = sc.nextInt();
            System.out.println("Enter the number of edges: ");
            e = sc.nextInt();

            graph = new Represent_Graph_Adjacency_Matrix(v);

            System.out.println("Enter the edges: <to> <from>");
            while (count <= e)
            {
                to = sc.nextInt();
                from = sc.nextInt();

                graph.makeEdge(to, from, 1);
                count++;
            }

            System.out.println("The adjacency matrix for the given graph is: ");
            System.out.print("  ");
            for (int i = 1; i <= v; i++)
                System.out.print(i + " ");
            System.out.println();
```

```
            for (int i = 1; i <= v; i++)
            {
                System.out.print(i + " ");
                for (int j = 1; j <= v; j++)
                    System.out.print(graph.getEdge(i, j) + " ");
                System.out.println();
            }

        }
        catch (Exception E)
        {
            System.out.println("Somthing went wrong");
        }

        sc.close();
    }
}
```

Running the code: Save the file and compile using `javac Represent_Graph_Adjacency_Matrix.java`

Example:

```
$ java Represent_Graph_Adjacency_Matrix
Enter the number of vertices:
4
Enter the number of edges:
6
Enter the edges:
1 1
3 4
2 3
1 4
2 4
1 2
The adjacency matrix for the given graph is:
1 2 3 4
1 1 1 0 1
2 0 0 1 1
3 0 0 0 1
4 0 0 0 0
```

Section 9.2: Introduction To Graph Theory

Graph Theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.

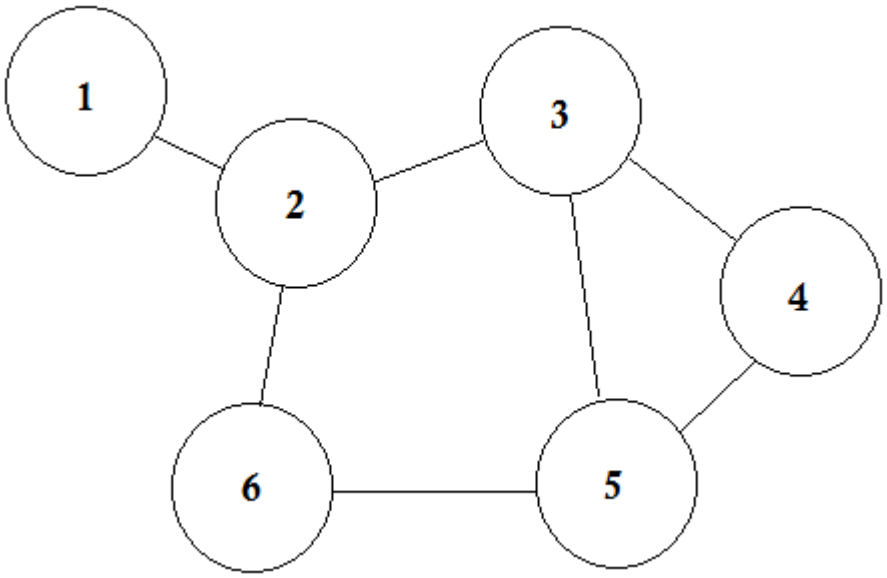
Did you know, almost all the problems of planet Earth can be converted into problems of Roads and Cities, and solved? Graph Theory was invented many years ago, even before the invention of computer. Leonhard Euler wrote a paper on the Seven Bridges of Königsberg which is regarded as the first paper of Graph Theory. Since then, people have come to realize that if we can convert any problem to this City-Road problem, we can solve it easily by Graph Theory.

Graph Theory has many applications.One of the most common application is to find the shortest distance between one city to another. We all know that to reach your PC, this web-page had to travel many routers from the server. Graph Theory helps it to find out the routers that needed to be crossed. During war, which street needs to be bombarded to disconnect the capital city from others, that too can be found out using Graph Theory.

Let us first learn some basic definitions on Graph Theory.

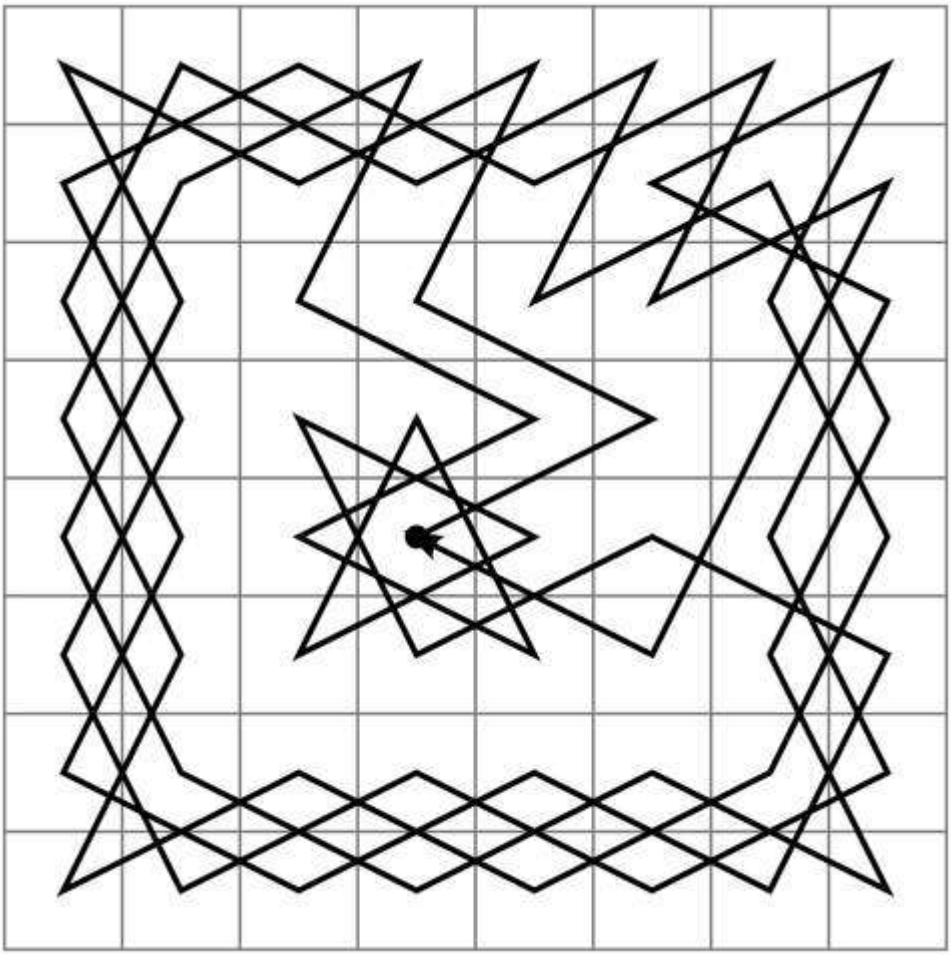
Graph:

Let's say, we have 6 cities. We mark them as 1, 2, 3, 4, 5, 6. Now we connect the cities that have roads between each other.



This is a simple graph where some cities are shown with the roads that are connecting them. In Graph Theory, we call each of these cities **Node** or **Vertex** and the roads are called **Edge.** Graph is simply a connection of these nodes and edges.

A **node** can represent a lot of things. In some graphs, nodes represent cities, some represent airports, some represent a square in a chessboard. **Edge** represents the relation between each nodes. That relation can be the time to go from one airport to another, the moves of a knight from one square to all the other squares etc.



Path of Knight in a Chessboard

In simple words, a **Node** represents any object and **Edge** represents the relation between two objects.

Adjacent Node:

If a node **A** shares an edge with node **B**, then **B** is considered to be adjacent to **A**. In other words, if two nodes are directly connected, they are called adjacent nodes. One node can have multiple adjacent nodes.

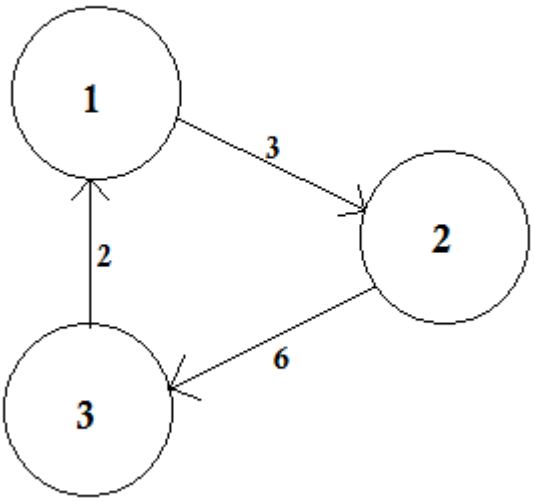
Directed and Undirected Graph:

In directed graphs, the edges have direction signs on one side, that means the edges are *Unidirectional*. On the other hand, the edges of undirected graphs have direction signs on both sides, that means they are *Bidirectional*. Usually undirected graphs are represented with no signs on the either sides of the edges.

Let's assume there is a party going on. The people in the party are represented by nodes and there is an edge between two people if they shake hands. Then this graph is undirected because any person **A** shake hands with person **B** if and only if **B** also shakes hands with **A**. In contrast, if the edges from a person **A** to another person **B** corresponds to **A**'s admiring **B**, then this graph is directed, because admiration is not necessarily reciprocated. The former type of graph is called an *undirected graph* and the edges are called *undirected edges* while the latter type of graph is called a *directed graph* and the edges are called *directed edges.*

Weighted and Unweighted Graph:

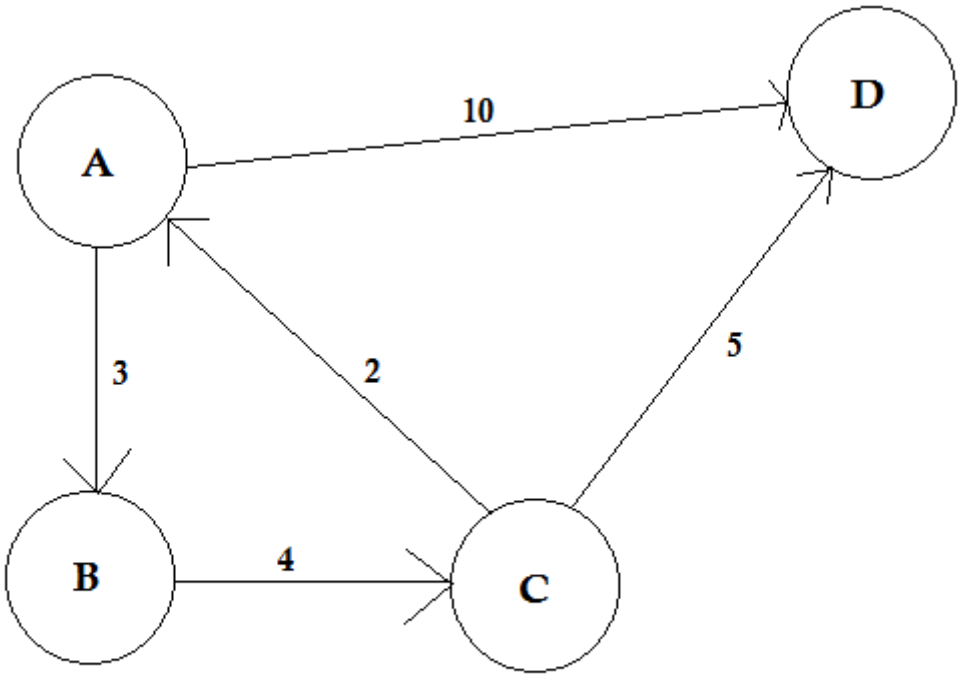
A weighted graph is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand.



An unweighted graph is simply the opposite. We assume that, the weight of all the edges are same (presumably 1).

Path:

A path represents a way of going from one node to another. It consists of sequence of edges. There can be multiple



paths between two nodes.

In the example above, there are two paths from **A** to **D**. **A->B, B->C, C->D** is one path. The cost of this path is **3 + 4 + 2 = 9**. Again, there's another path **A->D**. The cost of this path is **10**. The path that costs the lowest is called *shortest path*.

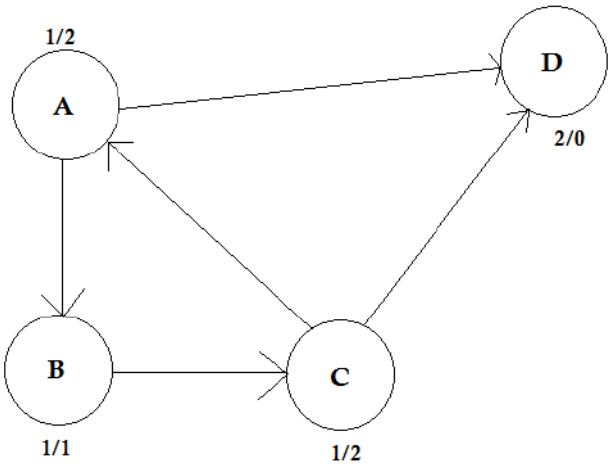
Degree:

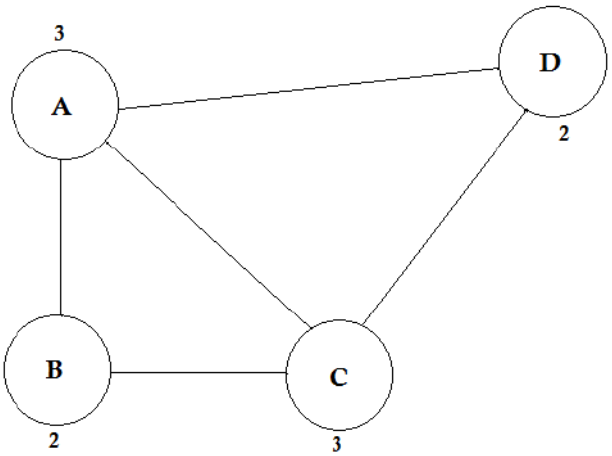
The degree of a vertex is the number of edges that are connected to it. If there's any edge that connects to the vertex at both ends (a loop) is counted twice.

In directed graphs, the nodes have two types of degrees:

- In-degree: The number of edges that point to the node.
- Out-degree: The number of edges that point from the node to other nodes.

For undirected graphs, they are simply called degree.





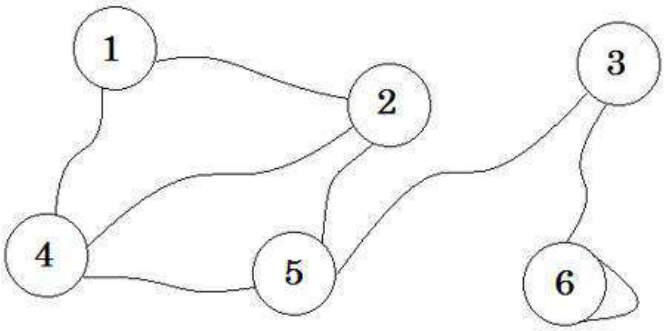
Some Algorithms Related to Graph Theory

- Bellman–Ford algorithm
- Dijkstra's algorithm
- Ford–Fulkerson algorithm
- Kruskal's algorithm
- Nearest neighbour algorithm
- Prim's algorithm
- Depth-first search
- Breadth-first search

Section 9.3: Storing Graphs (Adjacency List)

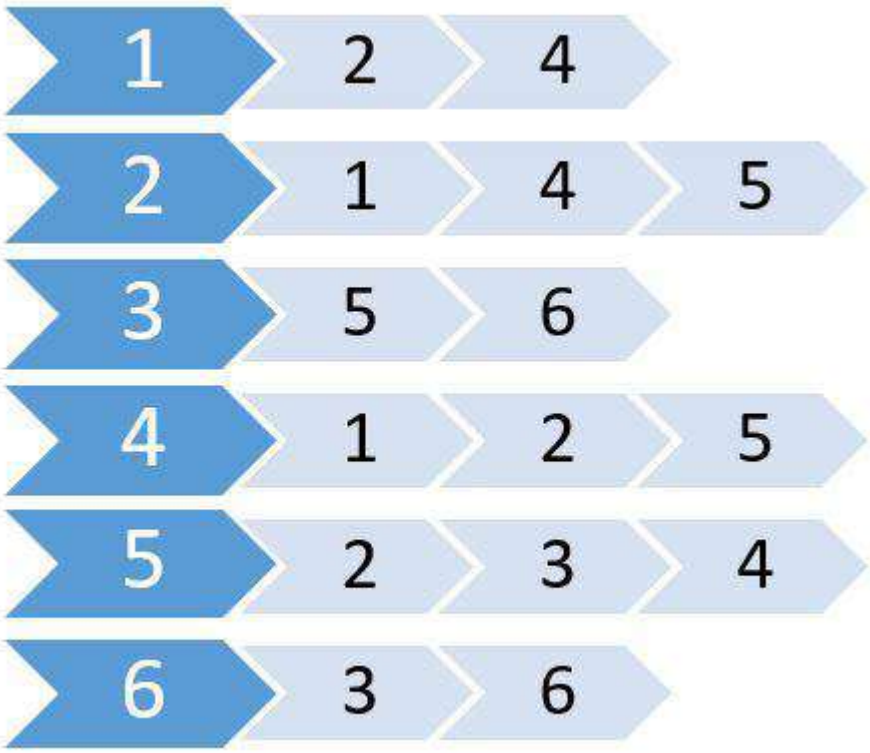
Adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in a graph. It takes less memory to store graphs.

Let's see a graph, and its adjacency matrix:



Node	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

Now we create a list using these values.



This is called adjacency list. It shows which nodes are connected to which nodes. We can store this information using a 2D array. But will cost us the same memory as Adjacency Matrix. Instead we are going to use dynamically allocated memory to store this one.

Many languages support **Vector** or **List** which we can use to store adjacency list. For these, we don't need to specify the size of the **List**. We only need to specify the maximum number of nodes.

The pseudo-code will be:

```
Procedure Adjacency-List(maxN, E):        // maxN denotes the maximum number of nodes
edge[maxN] = Vector()                     // E denotes the number of edges
for i from 1 to E
    input -> x, y                         // Here x, y denotes there is an edge between x, y
    edge[x].push(y)
    edge[y].push(x)
end for
Return edge
```

Since this one is an undirected graph, it there is an edge from **x** to **y**, there is also an edge from **y** to **x**. If it was a directed graph, we'd omit the second one. For weighted graphs, we need to store the cost too. We'll create another **vector** or **list** named **cost[]** to store these. The pseudo-code:

```
Procedure Adjacency-List(maxN, E):
edge[maxN] = Vector()
cost[maxN] = Vector()
for i from 1 to E
    input -> x, y, w
    edge[x].push(y)
    cost[x].push(w)
end for
Return edge, cost
```

From this one, we can easily find out the total number of nodes connected to any node, and what these nodes are.

It takes less time than Adjacency Matrix. But if we needed to find out if there's an edge between **u** and **v**, it'd have been easier if we kept an adjacency matrix.

Section 9.4: Topological Sort

A topological ordering, or a topological sort, orders the vertices in a directed acyclic graph on a line, i.e. in a list, such that all directed edges go from left to right. Such an ordering cannot exist if the graph contains a directed cycle because there is no way that you can keep going right on a line and still return back to where you started from.

Formally, in a graph `G = (V, E)`, then a linear ordering of all its vertices is such that if `G` contains an edge `(u, v) ∈ E`from vertex `u` to vertex `v` then `u` precedes `v` in the ordering.

It is important to note that each DAG has *at least one* topological sort.

There are known algorithms for constructing a topological ordering of any DAG in linear time, one example is:

1. Call `depth_first_search(G)` to compute finishing times `v.f` for each vertex `v`
2. As each vertex is finished, insert it into the front of a linked list
3. the linked list of vertices, as it is now sorted.

A topological sort can be performed in `□(V + E)` time, since the depth-first search algorithm takes `□(V + E)` time and it takes `Ω(1)` (constant time) to insert each of `|V|` vertices into the front of a linked list.

Many applications use directed acyclic graphs to indicate precedences among events. We use topological sorting so that we get an ordering to process each vertex before any of its successors.

Vertices in a graph may represent tasks to be performed and the edges may represent constraints that one task must be performed before another; a topological ordering is a valid sequence to perform the tasks set of tasks described in `V`.

Problem instance and its solution

Let a vertice `v` describe a `Task(hours_to_complete: int)`, i. e. `Task(4)` describes a `Task` that takes `4` hours to complete, and an edge `e` describe a `Cooldown(hours: int)` such that `Cooldown(3)` describes a duration of time to cool down after a completed task.

Let our graph be called `dag` (since it is a directed acyclic graph), and let it contain 5 vertices:

```
A <- dag.add_vertex(Task(4));
B <- dag.add_vertex(Task(5));
C <- dag.add_vertex(Task(3));
D <- dag.add_vertex(Task(2));
E <- dag.add_vertex(Task(7));
```

where we connect the vertices with directed edges such that the graph is acyclic,

```
// A ---> C ----+
// |      |     |
// v      v     v
// B ---> D --> E
dag.add_edge(A, B, Cooldown(2));
dag.add_edge(A, C, Cooldown(2));
dag.add_edge(B, D, Cooldown(1));
dag.add_edge(C, D, Cooldown(1));
dag.add_edge(C, E, Cooldown(1));
dag.add_edge(D, E, Cooldown(3));
```

then there are three possible topological orderings between `A` and `E`,

1. `A -> B -> D -> E`
2. `A -> C -> D -> E`
3. `A -> C -> E`

Section 9.5: Detecting a cycle in a directed graph using Depth First Traversal

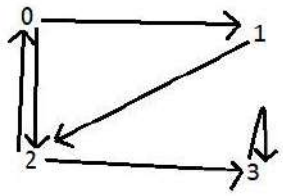
A cycle in a directed graph exists if there's a back edge discovered during a DFS. A back edge is an edge from a node to itself or one of the ancestors in a DFS tree. For a disconnected graph, we get a DFS forest, so you have to iterate through all vertices in the graph to find disjoint DFS trees.

C++ implementation:

```
    #include <iostream>
     #include <list>

    using namespace std;

    #define NUM_V   4

    bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
    {
        visited[u]=true;
        recStack[u]=true;
        list<int>::iterator i;
        for(i = graph[u].begin();i!=graph[u].end();++i)
        {
            if(recStack[*i]) //if vertice v is found in recursion stack of this DFS traversal
                return true;
            else if(*i==u) //if there's an edge from the vertex to itself
                return true;
            else if(!visited[*i])
            {   if(helper(graph, *i, visited, recStack))
                   return true;
            }
        }
        recStack[u]=false;
        return false;
    }
    /*
    /The wrapper function calls helper function on each vertices which have not been visited. Helper
function returns true if it detects a back edge in the subgraph(tree) or false.
*/
    bool isCyclic(list<int> *graph, int V)
    {
      bool visited[V];  //array to track vertices already visited
      bool recStack[V]; //array to track vertices in recursion stack of the traversal.

      for(int i = 0;i<V;i++)
       visited[i]=false, recStack[i]=false;  //initialize all vertices as not visited and not
recursed

      for(int u = 0; u < V; u++) //Iteratively checks if every vertices have been visited
      {   if(visited[u]==false)
          {  if(helper(graph, u, visited, recStack)) //checks if the DFS tree from the vertex
contains a cycle
                return true;
```

```
        }
      }
       return false;
    }
    /*
    Driver function
    */
    int main()
    {
        list<int>* graph = new list<int>[NUM_V];
        graph[0].push_back(1);
        graph[0].push_back(2);
        graph[1].push_back(2);
        graph[2].push_back(0);
        graph[2].push_back(3);
        graph[3].push_back(3);
        bool res = isCyclic(graph, NUM_V);
        cout<<res<<endl;
    }
```

Result: As shown below, there are three back edges in the graph. One between vertex 0 and 2; between vertice 0, 1, and 2; and vertex 3. Time complexity of search is O(V+E) where V is the number of vertices and E is the number of edges.



Section 9.6: Thorup's algorithm

Thorup's algorithm for single source shortest path for undirected graph has the time complexity O(m), lower than Dijkstra.

Basic ideas are the following. (Sorry, I didn't try implementing it yet, so I might miss some minor details. And the original paper is paywalled so I tried to reconstruct it from other sources referencing it. Please remove this comment if you could verify.)

- There are ways to find the spanning tree in O(m) (not described here). You need to "grow" the spanning tree from the shortest edge to the longest, and it would be a forest with several connected components before

fully grown.

- Select an integer b (b>=2) and only consider the spanning forests with length limit b^k. Merge the components which are exactly the same but with different k, and call the minimum k the level of the component. Then logically make components into a tree. u is the parent of v iff u is the smallest component distinct from v that fully contains v. The root is the whole graph and the leaves are single vertices in the original graph (with the level of negative infinity). The tree still has only O(n) nodes.
- Maintain the distance of each component to the source (like in Dijkstra's algorithm). The distance of a component with more than one vertices is the minimum distance of its unexpanded children. Set the distance of the source vertex to 0 and update the ancestors accordingly.
- Consider the distances in base b. When visiting a node in level k the first time, put its children into buckets shared by all nodes of level k (as in bucket sort, replacing the heap in Dijkstra's algorithm) by the digit k and higher of its distance. Each time visiting a node, consider only its first b buckets, visit and remove each of them, update the distance of the current node, and relink the current node to its own parent using the new distance and wait for the next visit for the following buckets.
- When a leaf is visited, the current distance is the final distance of the vertex. Expand all edges from it in the original graph and update the distances accordingly.
- Visit the root node (whole graph) repeatedly until the destination is reached.

It is based on the fact that, there isn't an edge with length less than l between two connected components of the spanning forest with length limitation l, so, starting at distance x, you could focus only on one connected component until you reach the distance x + l. You'll visit some vertices before vertices with shorter distance are all visited, but that doesn't matter because it is known there won't be a shorter path to here from those vertices. Other parts work like the bucket sort / MSD radix sort, and of course, it requires the O(m) spanning tree.

Chapter 10: Graph Traversals

Section 10.1: Depth First Search traversal function

The function takes the argument of the current node index, adjacency list (stored in vector of vectors in this example), and vector of boolean to keep track of which node has been visited.

```
void dfs(int node, vector<vector<int>>* graph, vector<bool>* visited) {
    // check whether node has been visited before
    if((*visited)[node])
        return;

    // set as visited to avoid visiting the same node twice
    (*visited)[node] = true;

    // perform some action here
    cout << node;

    // traverse to the adjacent nodes in depth-first manner
    for(int i = 0; i < (*graph)[node].size(); ++i)
        dfs((*graph)[node][i], graph, visited);
}
```

Chapter 11: Dijkstra's Algorithm

Section 11.1: Dijkstra's Shortest Path Algorithm

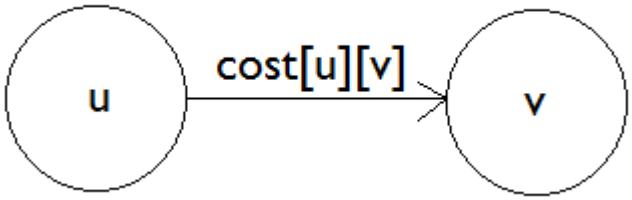
Before proceeding, it is recommended to have a brief idea about Adjacency Matrix and BFS

Dijkstra's algorithm is known as single-source shortest path algorithm. It is used for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by Edsger W. Dijkstra in 1956 and published three years later.

We can find shortest path using Breadth First Search (BFS) searching algorithm. This algorithm works fine, but the problem is, it assumes the cost of traversing each path is same, that means the cost of each edge is same. Dijkstra's algorithm helps us to find the shortest path where the cost of each path is not the same.

At first we will see, how to modify BFS to write Dijkstra's algorithm, then we will add priority queue to make it a complete Dijkstra's algorithm.

Let's say, the distance of each node from the source is kept in **d[]** array. As in, **d[3]** represents that **d[3]** time is taken to reach **node 3** from **source**. If we don't know the distance, we will store *infinity* in **d[3]**. Also, let **cost[u][v]** represent the cost of **u-v**. That means it takes **cost[u][v]** to go from **u** node to **v** node.



We need to understand Edge Relaxation. Let's say, from your house, that is **source**, it takes *10* minutes to go to place **A**. And it takes *25* minutes to go to place **B**. We have,

```
d[A] = 10
d[B] = 25
```

Now let's say it takes *7* minutes to go from place **A** to place **B**, that means:

```
cost[A][B] = 7
```

Then we can go to place **B** from **source** by going to place **A** from **source** and then from place **A**, going to place **B**, which will take *10 + 7 = 17* minutes, instead of *25* minutes. So,

```
d[A] + cost[A][B] < d[B]
```

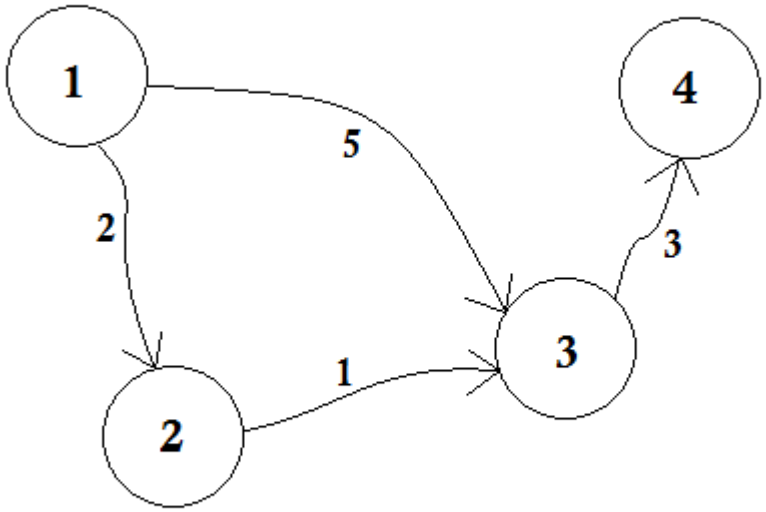
Then we update,

```
d[B] = d[A] + cost[A][B]
```

This is called relaxation. We will go from node **u** to node **v** and if **d[u] + cost[u][v] < d[v]** then we will update **d[v] = d[u] + cost[u][v]**.

In BFS, we didn't need to visit any node twice. We only checked if a node is visited or not. If it was not visited, we pushed the node in queue, marked it as visited and incremented the distance by 1. In Dijkstra, we can push a node

in queue and instead of updating it with visited, we *relax* or update the new edge. Let's look at one example:



Let's assume, **Node 1** is the **Source**. Then,

```
d[1] = 0
d[2] = d[3] = d[4] = infinity (or a large value)
```

We set, **d[2], d[3]** and **d[4]** to *infinity* because we don't know the distance yet. And the distance of **source** is of course *0*. Now, we go to other nodes from **source** and if we can update them, then we'll push them in the queue. Say for example, we'll traverse **edge 1-2**. As **d[1] + 2 < d[2]** which will make **d[2] = 2**. Similarly, we'll traverse **edge 1-3** which makes **d[3] = 5**.

We can clearly see that *5* is not the shortest distance we can cross to go to **node 3**. So traversing a node only once, like BFS, doesn't work here. If we go from **node 2** to **node 3** using **edge 2-3**, we can update **d[3] = d[2] + 1 = 3**. So we can see that one node can be updated many times. How many times you ask? The maximum number of times a node can be updated is the number of in-degree of a node.

Let's see the pseudo-code for visiting any node multiple times. We will simply modify BFS:

```
procedure BFSmodified(G, source):
Q = queue()
distance[] = infinity
Q.enqueue(source)
distance[source]=0
while Q is not empty
    u <- Q.pop()
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
        end if
    end for
end while
Return distance
```

This can be used to find the shortest path of all node from the source. The complexity of this code is not so good. Here's why,

In BFS, when we go from **node 1** to all other nodes, we follow *first come, first serve* method. For example, we went to **node 3** from **source** before processing **node 2**. If we go to **node 3** from **source**, we update **node 4** as **5 + 3 = 8**. When we again update **node 3** from **node 2**, we need to update **node 4** as **3 + 3 = 6** again! So **node 4** is updated twice.

Dijkstra proposed, instead of going for *First come, first serve* method, if we update the nearest nodes first, then it'll take less updates. If we processed **node 2** before, then **node 3** would have been updated before, and after updating **node 4** accordingly, we'd easily get the shortest distance! The idea is to choose from the queue, the node, that is closest to the **source**. So we will use *Priority Queue* here so that when we pop the queue, it will bring us the closest node **u** from **source**. How will it do that? It'll check the value of **d[u]** with it.

Let's see the pseudo-code:

```
procedure dijkstra(G, source):
Q = priority_queue()
distance[] = infinity
Q.enqueue(source)
distance[source] = 0
while Q is not empty
    u <- nodes in Q with minimum distance[]
    remove u from the Q
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
            Q.enqueue(v)
        end if
    end for
end while
Return distance
```

The pseudo-code returns distance of all other nodes from the **source**. If we want to know distance of a single node **v**, we can simply return the value when **v** is popped from the queue.

Now, does Dijkstra's Algorithm work when there's a negative edge? If there's a negative cycle, then infinity loop will occur, as it will keep reducing the cost every time. Even if there is a negative edge, Dijkstra won't work, unless we return right after the target is popped. But then, it won't be a Dijkstra algorithm. We'll need Bellman–Ford algorithm for processing negative edge/cycle.

Complexity:

The complexity of BFS is **O(log(V+E))** where **V** is the number of nodes and **E** is the number of edges. For Dijkstra, the complexity is similar, but sorting of *Priority Queue* takes **O(logV)**. So the total complexity is: **O(Vlog(V)+E)**

Below is a Java example to solve Dijkstra's Shortest Path Algorithm using Adjacency Matrix

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
```

```
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    void dijkstra(int graph[][], int src)
    {

        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        dist[src] = 0;

        for (int count = 0; count < V-1; count++)
        {
            int u = minDistance(dist, sptSet);

            sptSet[u] = true;

            for (int v = 0; v < V; v++)

                if (!sptSet[v] && graph[u][v]!=0 &&
                        dist[u] != Integer.MAX_VALUE &&
                        dist[u]+graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        printSolution(dist, V);
    }

    public static void main (String[] args)
    {
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                };
        ShortestPath t = new ShortestPath();
```

```
        t.dijkstra(graph, 0);
    }
}
```

Expected output of the program is

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

Chapter 12: A* Pathfinding

Section 12.1: Introduction to A*

A* (A star) is a search algorithm that is used for finding path from one node to another. So it can be compared with Breadth First Search, or Dijkstra's algorithm, or Depth First Search, or Best First Search. A* algorithm is widely used in graph search for being better in efficiency and accuracy, where graph pre-processing is not an option.

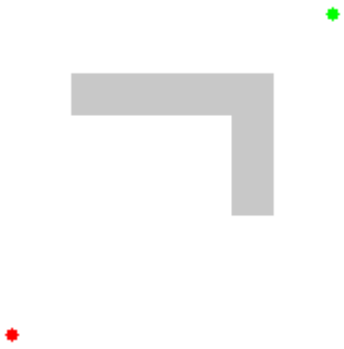
A* is a an specialization of Best First Search , in which the function of evaluation *f* is define in a particular way.

f(n) = g(n) + h(n) is the minimum cost since the initial node to the objectives conditioned to go thought node *n*.

g(n) is the minimum cost from the initial node to *n*.

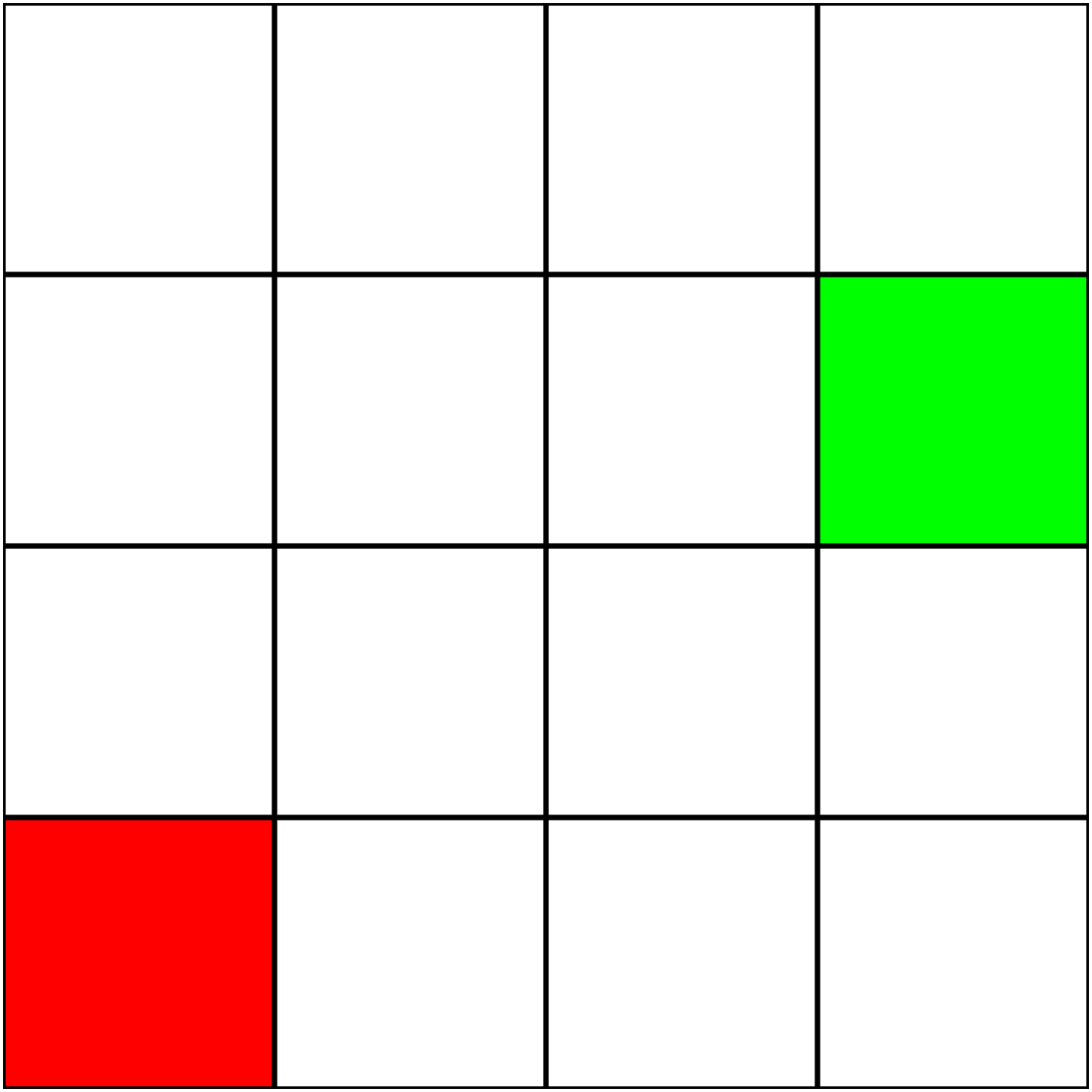
h(n) is the minimum cost from *n* to the closest objective to *n*

A* is an informed search algorithm and it always guarantees to find the smallest path (path with minimum cost) in the least possible time (if uses admissible heuristic). So it is both *complete* and *optimal*. The following animation demonstrates A* search-

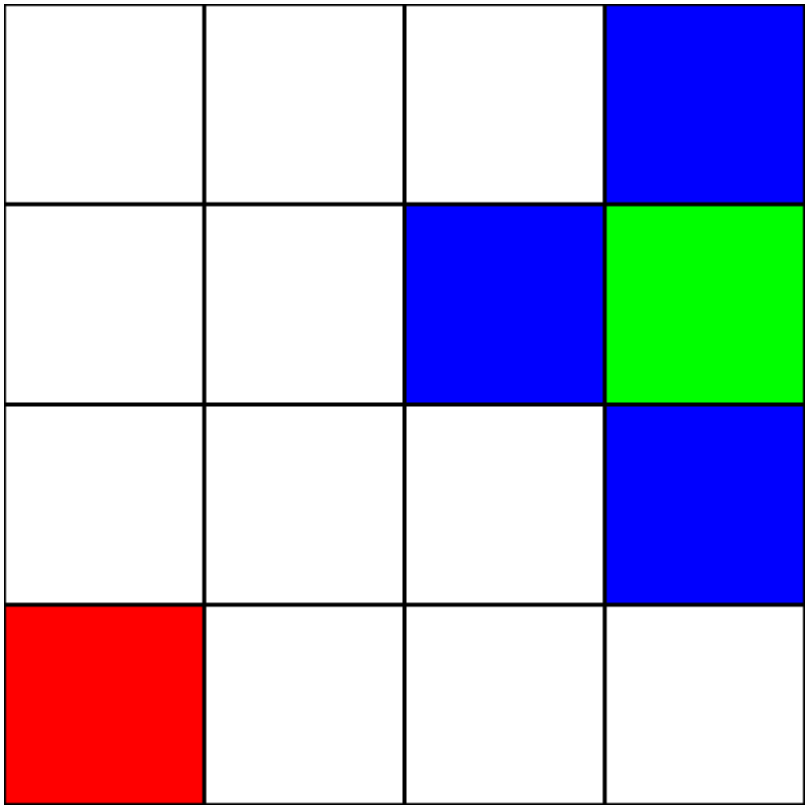


Section 12.2: A* Pathfinding through a maze with no obstacles

Let's say we have the following 4 by 4 grid:



Let's assume that this is a *maze*. There are no walls/obstacles, though. We only have a starting point (the green square), and an ending point (the red square). Let's also assume that in order to get from green to red, we cannot move diagonally. So, starting from the green square, let's see which squares we can move to, and highlight them in blue:



In order to choose which square to move to next, we need to take into account 2 heuristics:

1. The "g" value - This is how far away this node is from the green square.
2. The "h" value - This is how far away this node is from the red square.
3. The "f" value - This is the sum of the "g" value and the "h" value. This is the final number which tells us which node to move to.

In order to calculate these heuristics, this is the formula we will use: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

This is known as the "Manhattan Distance" formula.

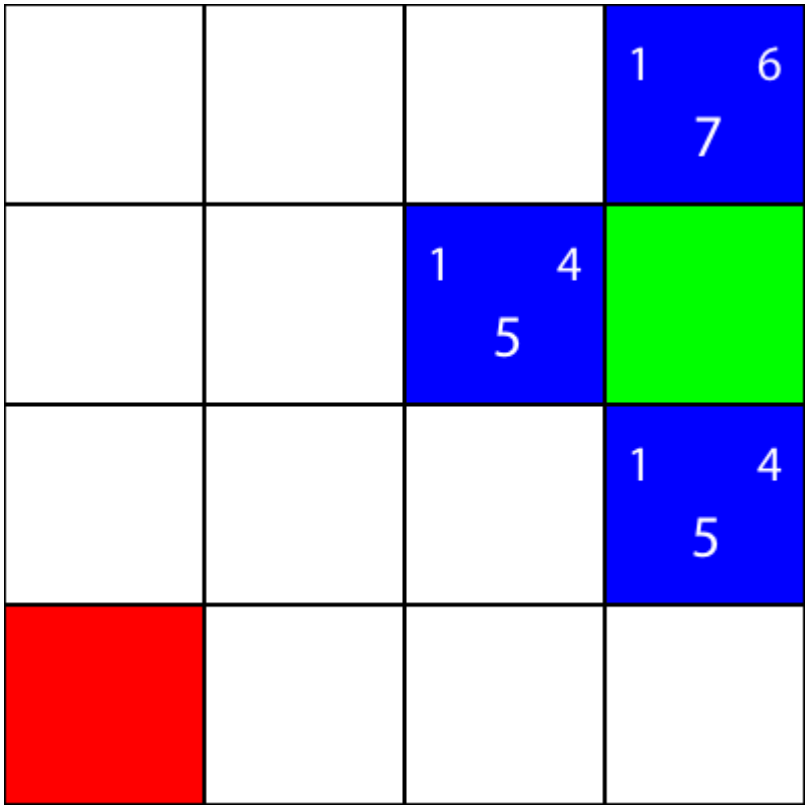
Let's calculate the "g" value for the blue square immediately to the left of the green square: `abs(3 - 2) + abs(2 - 2) = 1`

Great! We've got the value: 1. Now, let's try calculating the "h" value: `abs(2 - 0) + abs(2 - 0) = 4`

Perfect. Now, let's get the "f" value: `1 + 4 = 5`

So, the final value for this node is "5".

Let's do the same for all the other blue squares. The big number in the center of each square is the "f" value, while the number on the top left is the "g" value, and the number on the top right is the "h" value:



We've calculated the g, h, and f values for all of the blue nodes. Now, which do we pick?

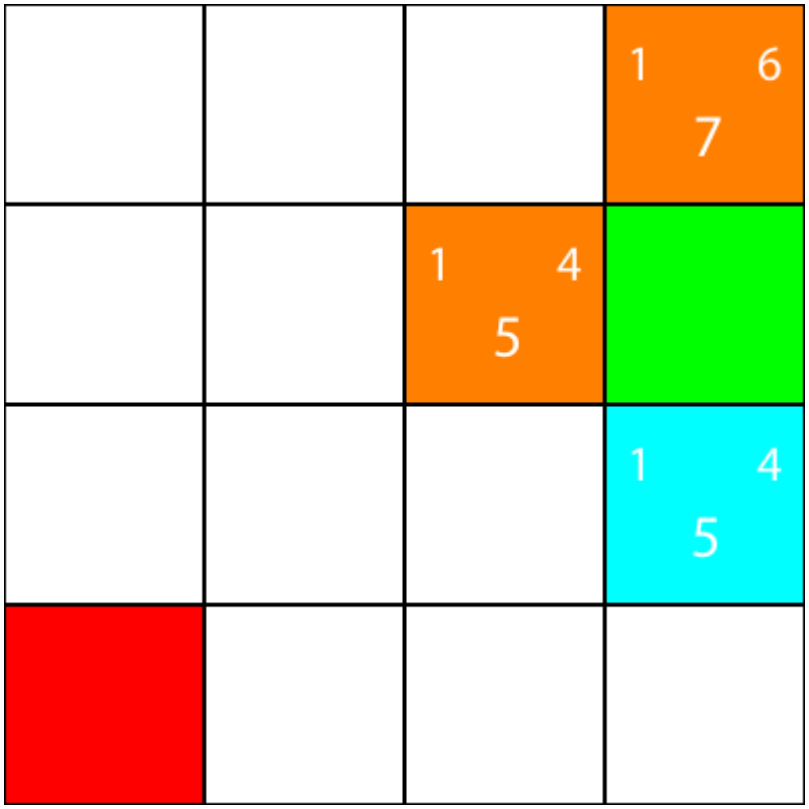
Whichever one has the lowest f value.

However, in this case, we have 2 nodes with the same f value, 5. How do we pick between them?

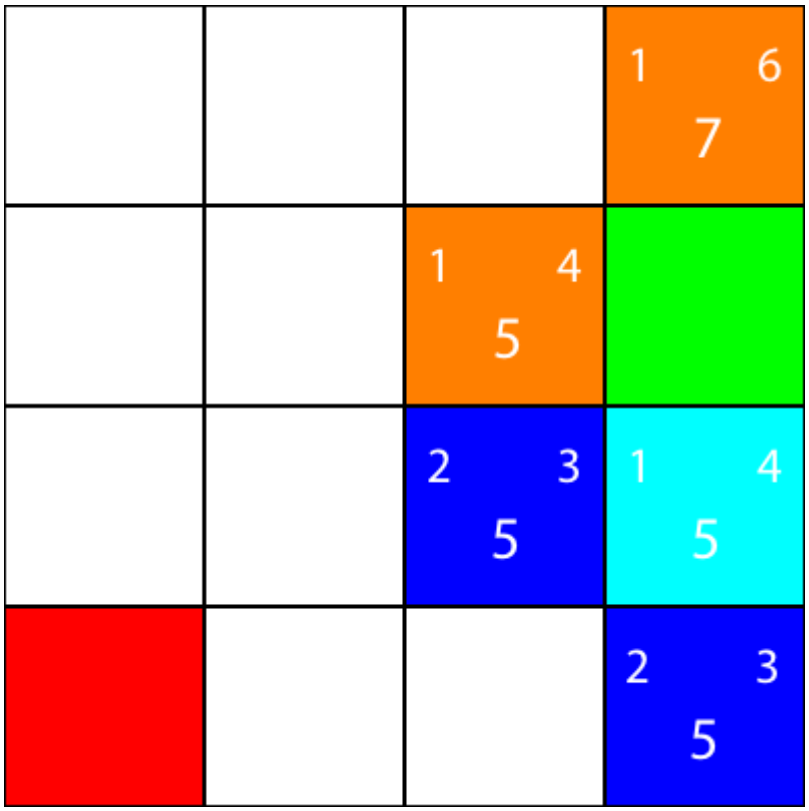
Simply, either choose one at random, or have a priority set. I usually prefer to have a priority like so: "Right > Up > Down > Left"

One of the nodes with the f value of 5 takes us in the "Down" direction, and the other takes us "Left". Since Down is at a higher priority than Left, we choose the square which takes us "Down".

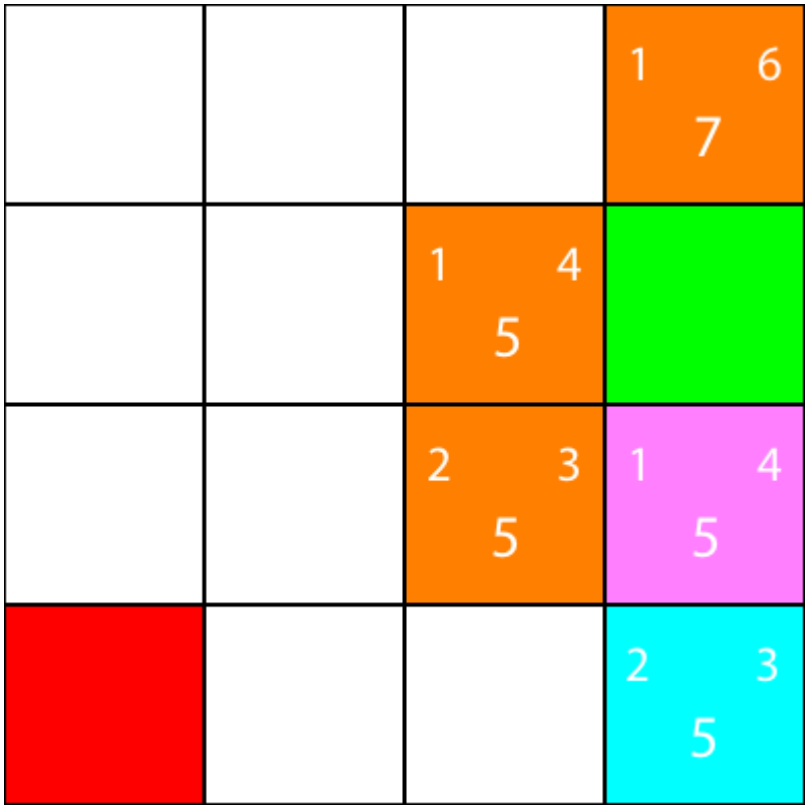
I now mark the nodes which we calculated the heuristics for, but did not move to, as orange, and the node which we chose as cyan:



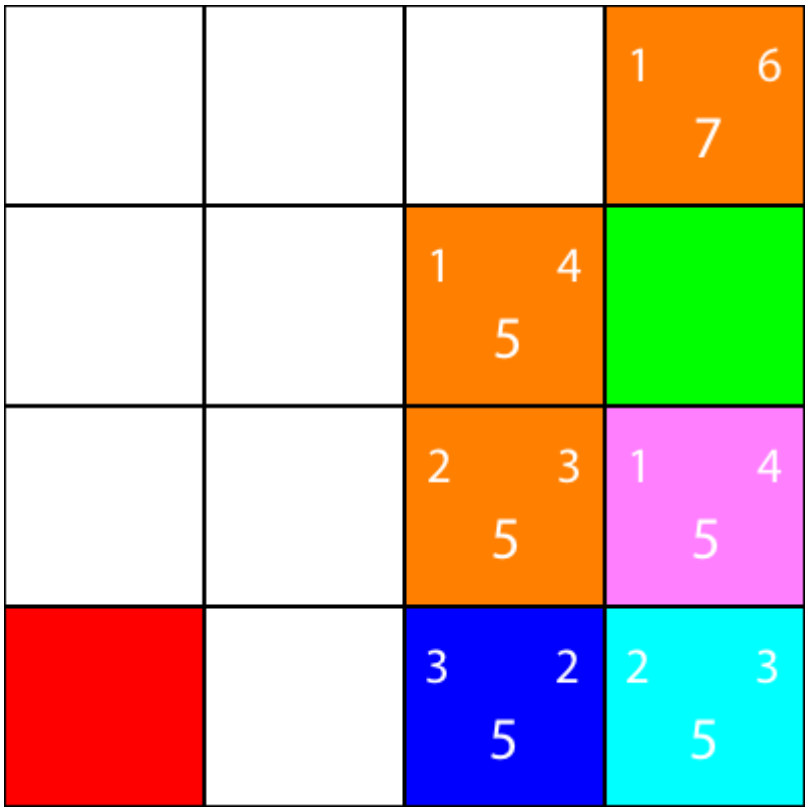
Alright, now let's calculate the same heuristics for the nodes around the cyan node:



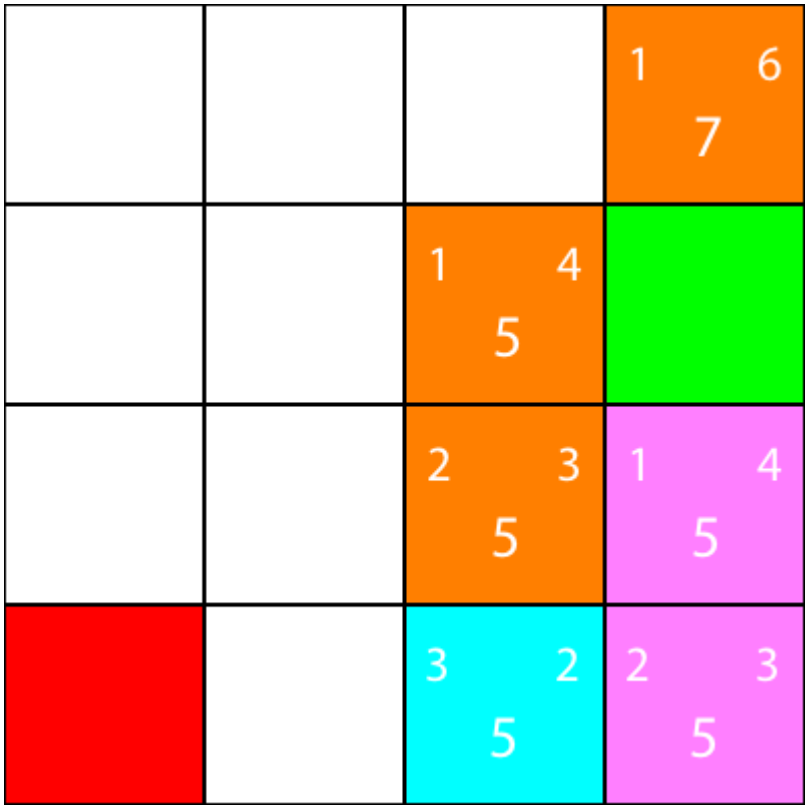
Again, we choose the node going down from the cyan node, as all the options have the same f value:



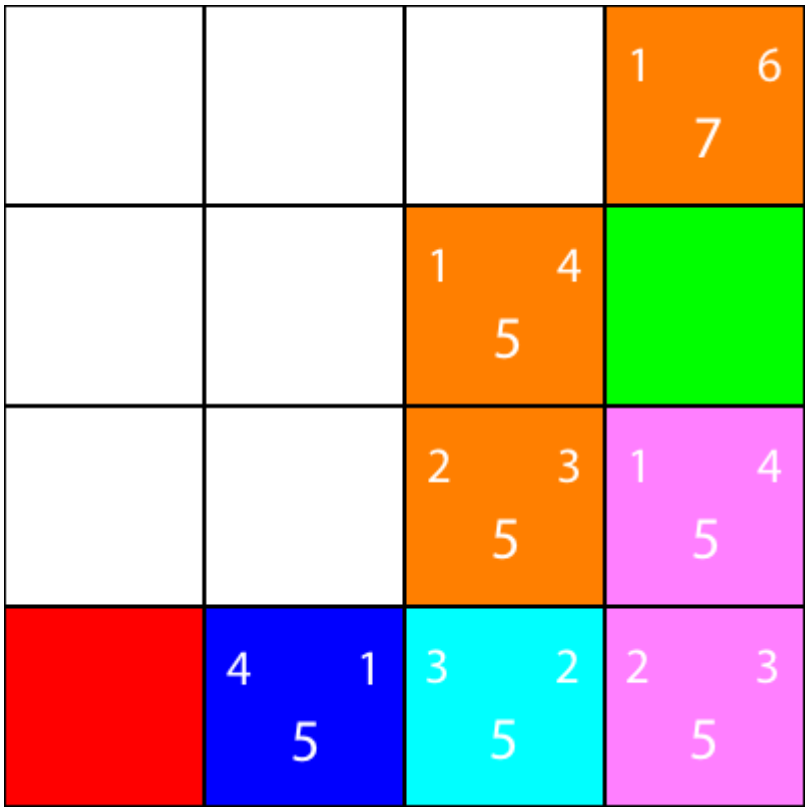
Let's calculate the heuristics for the only neighbour that the cyan node has:



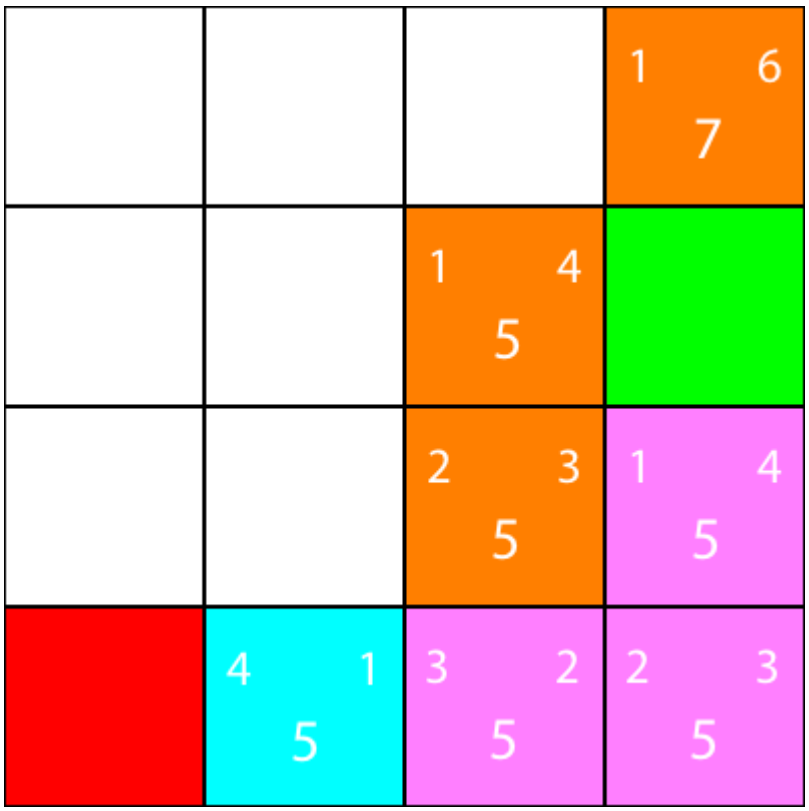
Alright, since we will follow the same pattern we have been following:



Once more, let's calculate the heuristics for the node's neighbour:



Let's move there:

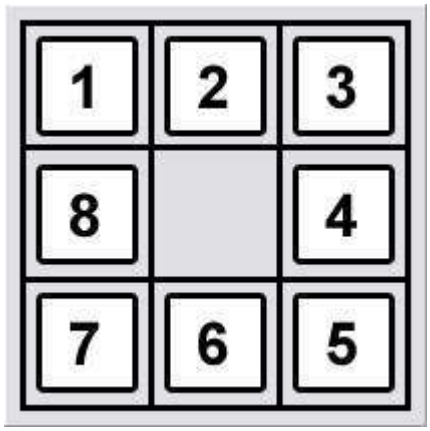


Finally, we can see that we have a *winning square* beside us, so we move there, and we are done.

Section 12.3: Solving 8-puzzle problem using A* algorithm

> **Problem definition**:

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty. The object is to move to squares around into different positions and having the numbers displayed in the "goal state".



Given an initial state of 8-puzzle game and a final state of to be reached, find the most cost-effective path to reach the final state from initial state.

> **Initial state**:

```
_ 1 3
4 2 5
7 8 6
```

> **Final state**:

```
1 2 3
4 5 6
7 8 _
```

> **Heuristic to be assumed**:

Let us consider the Manhattan distance between the current and final state as the heuristic for this problem statement.

```
h(n) = | x - p | + | y - q |
where x and y are cell co-ordinates in the current state
      p and q are cell co-ordinates in the final state
```

> **Total cost function**:

So the total cost function `f(n)` is given by,

```
f(n) = g(n) + h(n), where g(n) is the cost required to reach the current state from given initial state
```

> **Solution to example problem**:

First we find the heuristic value required to reach the final state from initial state. The cost function, g(n) = 0, as we are in the initial state

```
h(n) = 8
```

The above value is obtained, as `1` in the current state is 1 horizontal distance away than the `1` in final state. Same goes for `2`, `5`, `6`. `_` is 2 horizontal distance away and 2 vertical distance away. So total value for `h(n)` is 1 + 1 + 1 + 1 + 2 + 2 = 8. Total cost function `f(n)` is equal to 8 + 0 = 8.

Now, the possible states that can be reached from initial state are found and it happens that we can either move `_` to right or downwards.

So states obtained after moving those moves are:

```
1 _ 3    4 1 3
4 2 5    _ 2 5
7 8 6    7 8 6
 (1)      (2)
```

Again the total cost function is computed for these states using the method described above and it turns out to be 6 and 7 respectively. We chose the state with minimum cost which is state (1). The next possible moves can be Left, Right or Down. We won't move Left as we were previously in that state. So, we can move Right or Down.

Again we find the states obtained from (1).

```
1 3 _    1 2 3
```

```
4 2 5     4 _ 5
7 8 6     7 8 6
 (3)       (4)
```

(3) leads to cost function equal to 6 and (4) leads to 4. Also, we will consider (2) obtained before which has cost function equal to 7. Choosing minimum from them leads to (4). Next possible moves can be Left or Right or Down. We get states:

```
1 2 3     1 2 3     1 2 3
_ 4 5     4 5 _     4 8 5
7 8 6     7 8 6     7 _ 6
 (5)       (6)       (7)
```

We get costs equal to 5, 2 and 4 for (5), (6) and (7) respectively. Also, we have previous states (3) and (2) with 6 and 7 respectively. We chose minimum cost state which is (6). Next possible moves are Up, and Down and clearly Down will lead us to final state leading to heuristic function value equal to 0.

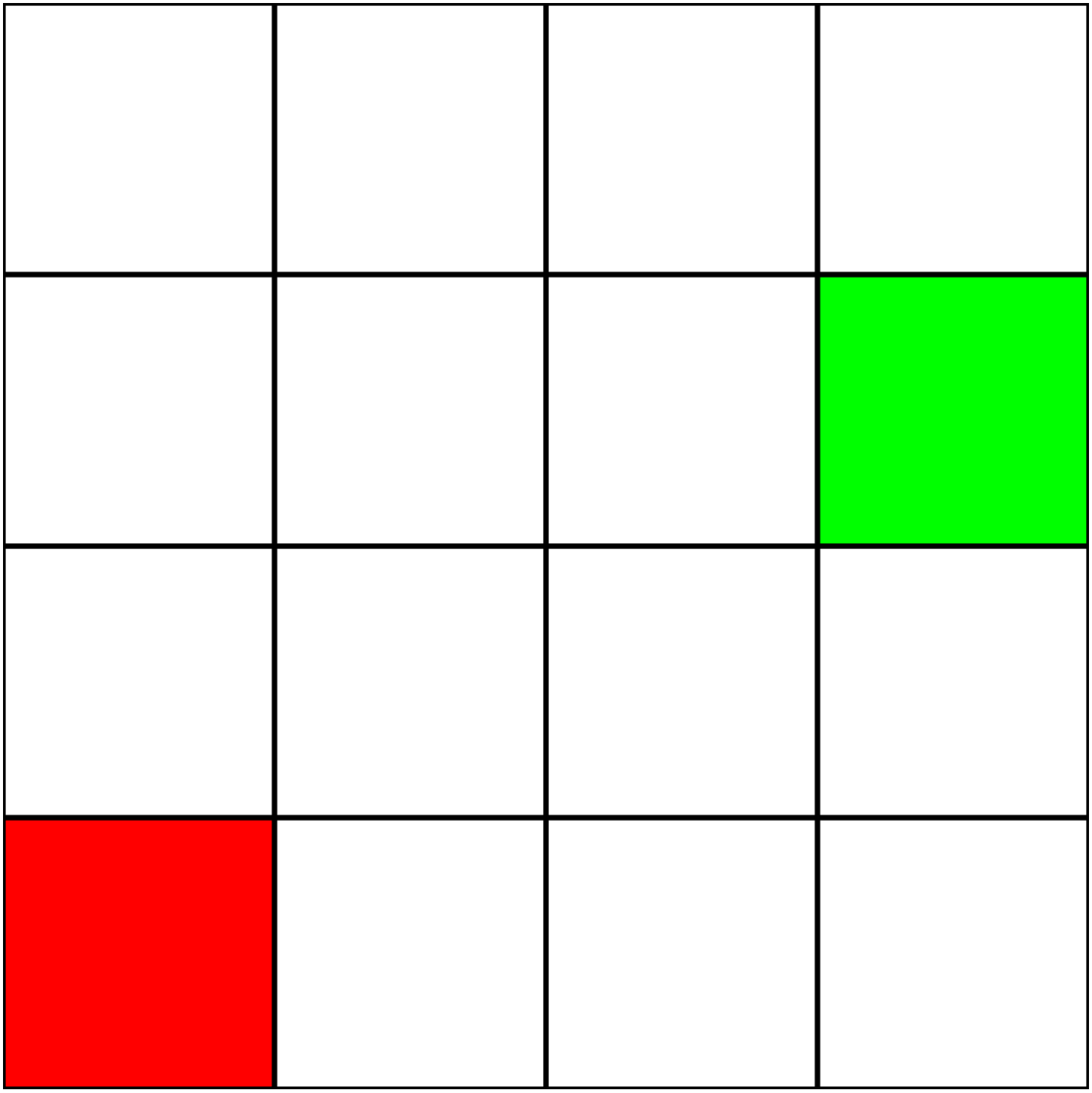
Chapter 13: A* Pathfinding Algorithm

This topic is going to focus on the A* Pathfinding algorithm, how it's used, and why it works.

Note to future contributors: I have added an example for A* Pathfinding without any obstacles, on a 4x4 grid. An example with obstacles is still needed.

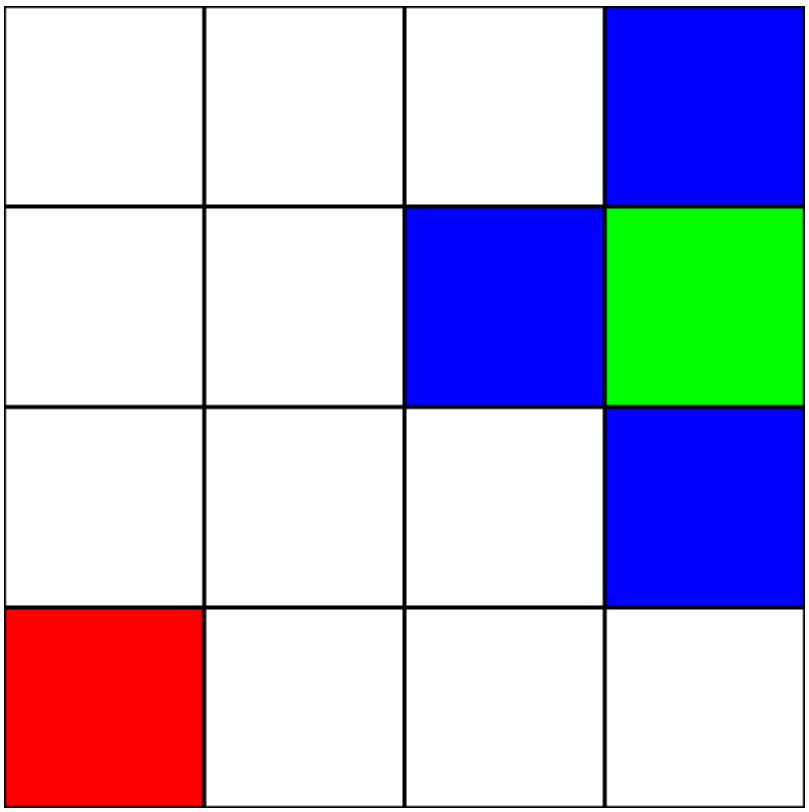
Section 13.1: Simple Example of A* Pathfinding: A maze with no obstacles

Let's say we have the following 4 by 4 grid:



Let's assume that this is a *maze*. There are no walls/obstacles, though. We only have a starting point (the green square), and an ending point (the red square). Let's also assume that in order to get from green to red, we cannot

move diagonally. So, starting from the green square, let's see which squares we can move to, and highlight them in blue:



In order to choose which square to move to next, we need to take into account 2 heuristics:

1. The "g" value - This is how far away this node is from the green square.
2. The "h" value - This is how far away this node is from the red square.
3. The "f" value - This is the sum of the "g" value and the "h" value. This is the final number which tells us which node to move to.

In order to calculate these heuristics, this is the formula we will use: `distance = abs(from.x - to.x) + abs(from.y - to.y)`

This is known as the "Manhattan Distance" formula.

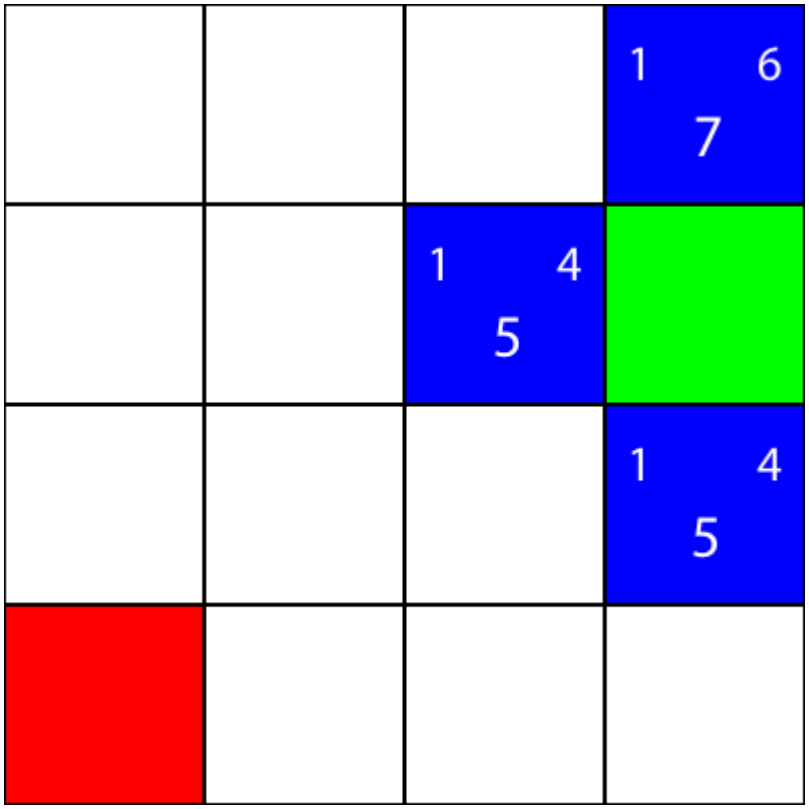
Let's calculate the "g" value for the blue square immediately to the left of the green square: `abs(3 - 2) + abs(2 - 2) = 1`

Great! We've got the value: 1. Now, let's try calculating the "h" value: `abs(2 - 0) + abs(2 - 0) = 4`

Perfect. Now, let's get the "f" value: `1 + 4 = 5`

So, the final value for this node is "5".

Let's do the same for all the other blue squares. The big number in the center of each square is the "f" value, while the number on the top left is the "g" value, and the number on the top right is the "h" value:



We've calculated the g, h, and f values for all of the blue nodes. Now, which do we pick?

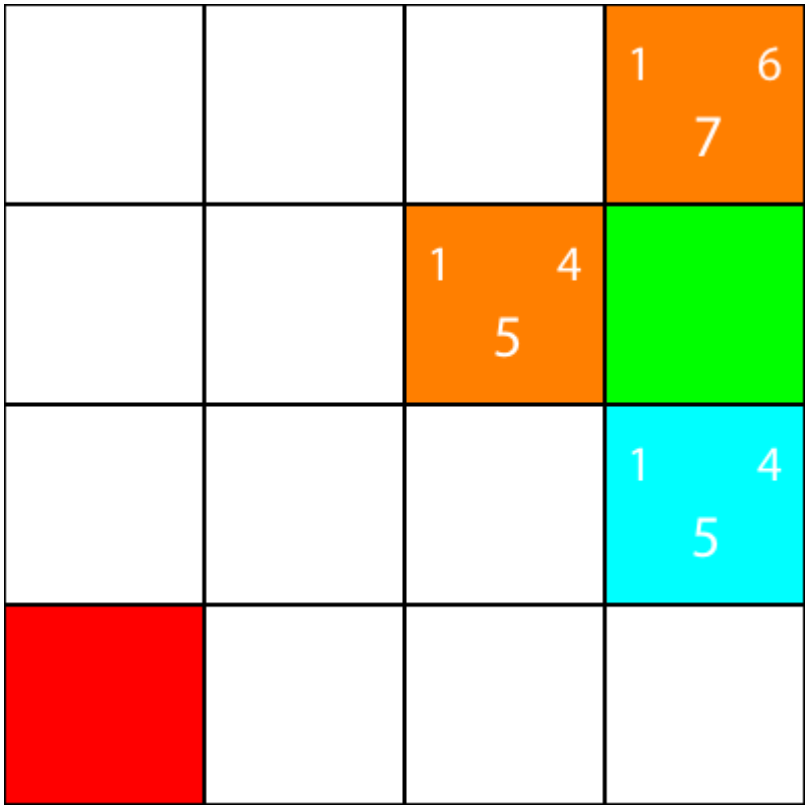
Whichever one has the lowest f value.

However, in this case, we have 2 nodes with the same f value, 5. How do we pick between them?

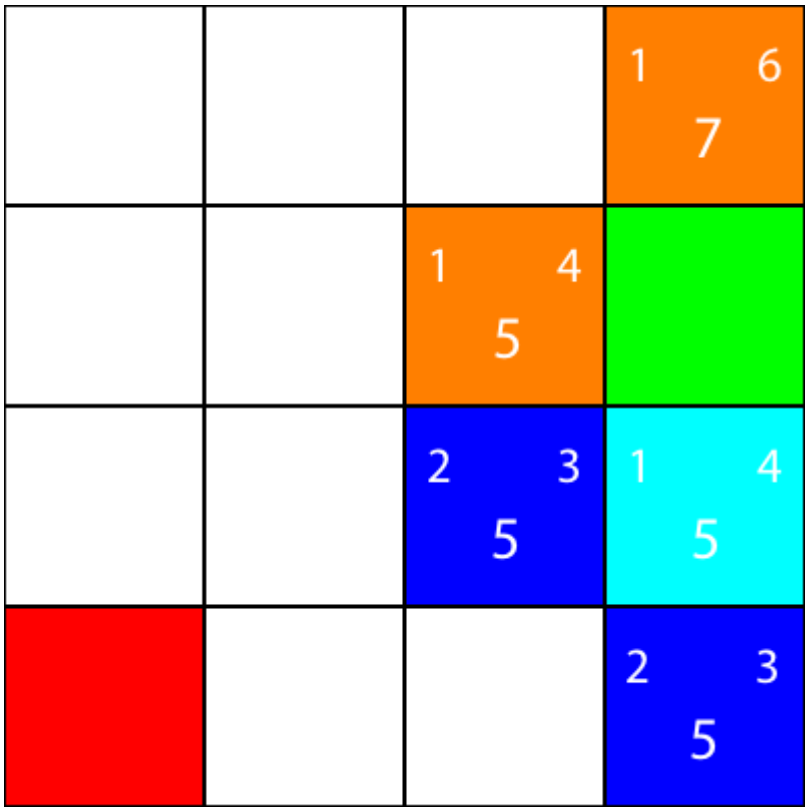
Simply, either choose one at random, or have a priority set. I usually prefer to have a priority like so: "Right > Up > Down > Left"

One of the nodes with the f value of 5 takes us in the "Down" direction, and the other takes us "Left". Since Down is at a higher priority than Left, we choose the square which takes us "Down".

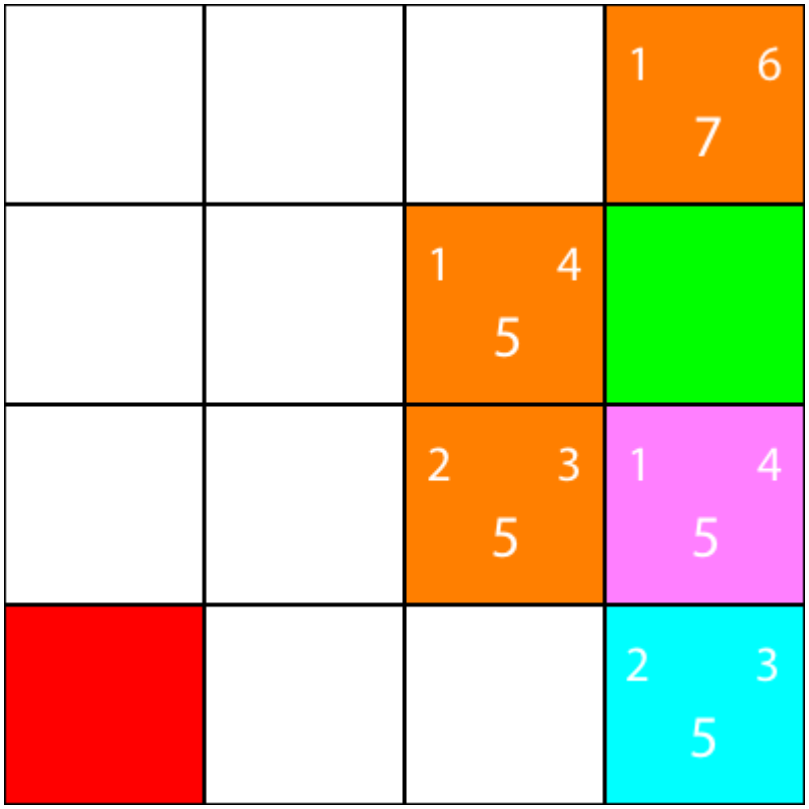
I now mark the nodes which we calculated the heuristics for, but did not move to, as orange, and the node which we chose as cyan:



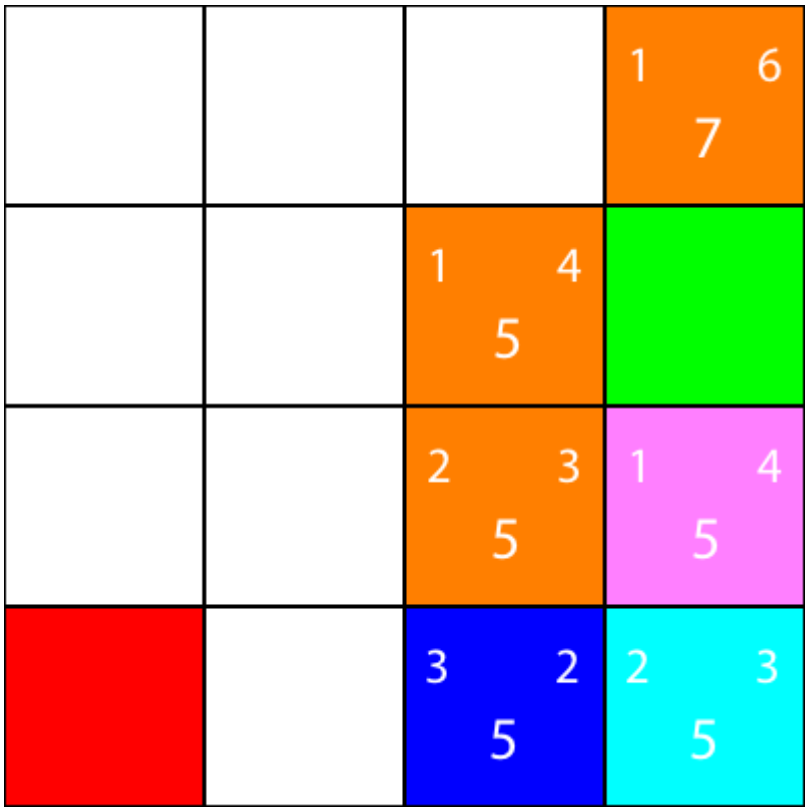
Alright, now let's calculate the same heuristics for the nodes around the cyan node:



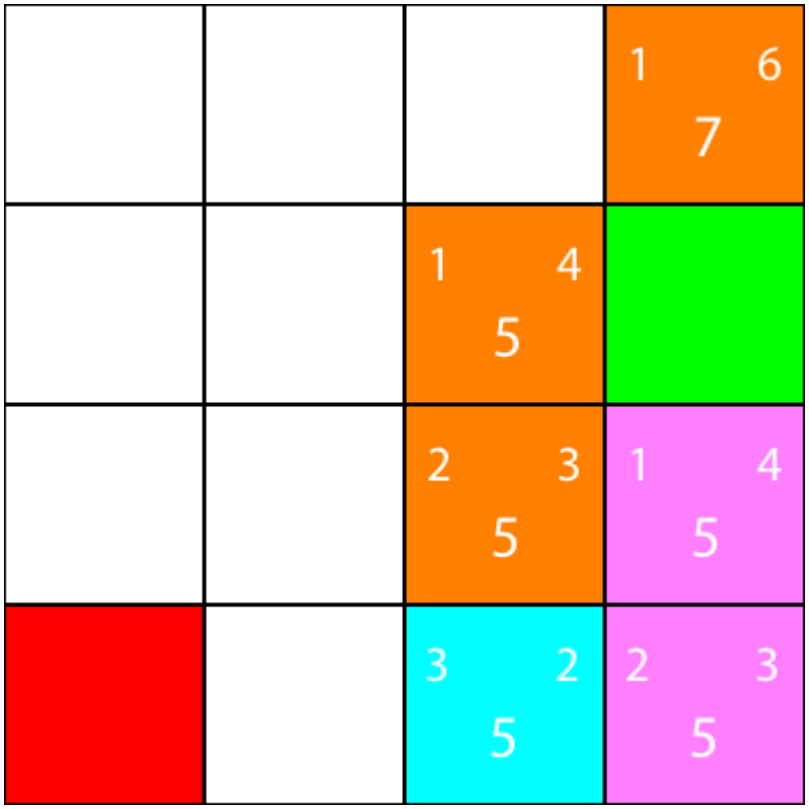
Again, we choose the node going down from the cyan node, as all the options have the same f value:



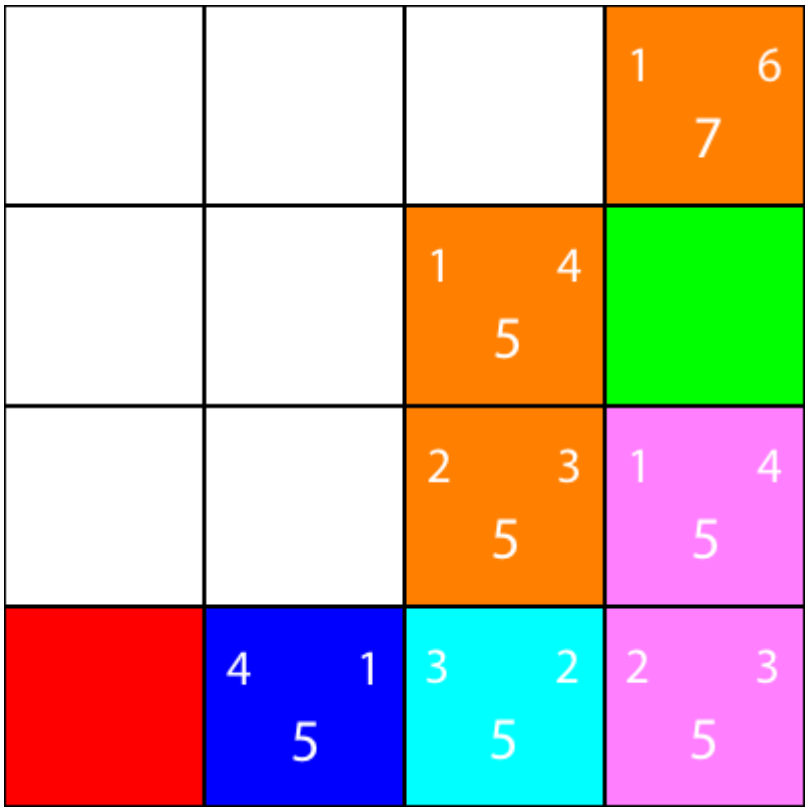
Let's calculate the heuristics for the only neighbour that the cyan node has:



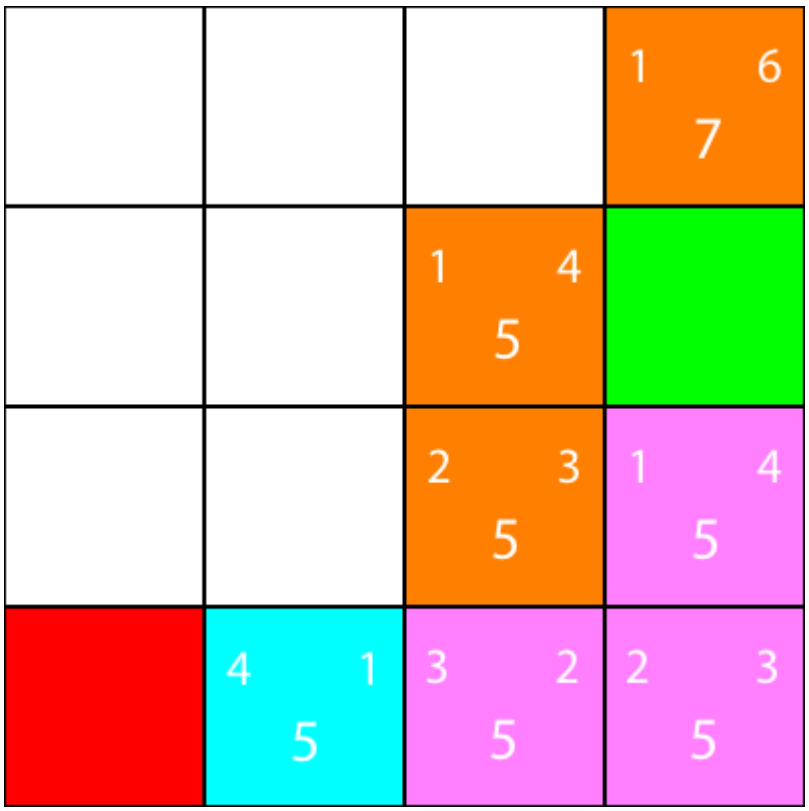
Alright, since we will follow the same pattern we have been following:



Once more, let's calculate the heuristics for the node's neighbour:



Let's move there:



Finally, we can see that we have a *winning square* beside us, so we move there, and we are done.

Chapter 14: Dynamic Programming

Dynamic programming is a widely used concept and its often used for optimization. It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner usually a bottom-up approach. There are two key attributes that a problem must have in order for dynamic programming to be applicable "Optimal substructure" and "Overlapping sub-problems". To achieve its optimization, dynamic programming uses a concept called memoization

Section 14.1: Edit Distance

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2.

Implementation in Java

```
public class EditDistance {

public static void main(String[] args) {
    // TODO Auto-generated method stub
    String str1 = "march";
    String str2 = "cart";

    EditDistance ed = new EditDistance();
    System.out.println(ed.getMinConversions(str1, str2));
}

public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

}

Output

3

Section 14.2: Weighted Job Scheduling Algorithm

Weighted Job Scheduling Algorithm can also be denoted as Weighted Activity Selection Algorithm.

The problem is, given certain jobs with their start time and end time, and a profit you make when you finish the job, what is the maximum profit you can make given no two jobs can be executed in parallel?

This one looks like Activity Selection using Greedy Algorithm, but there's an added twist. That is, instead of maximizing the number of jobs finished, we focus on making the maximum profit. The number of jobs performed doesn't matter here.

Let's look at an example:

Name	A	B	C	D	E	F
(Start Time, Finish Time)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
Profit	6	4	2	5	11	5

The jobs are denoted with a name, their start and finishing time and profit. After a few iterations, we can find out if we perform **Job-A** and **Job-E**, we can get the maximum profit of 17. Now how to find this out using an algorithm?

The first thing we do is sort the jobs by their finishing time in non-decreasing order. Why do we do this? It's because if we select a job that takes less time to finish, then we leave the most amount of time for choosing other jobs. We have:

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2

We'll have an additional temporary array **Acc_Prof** of size **n** (Here, **n** denotes the total number of jobs). This will contain the maximum accumulated profit of performing the jobs. Don't get it? Wait and watch. We'll initialize the values of the array with the profit of each jobs. That means, **Acc_Prof[i]** will at first hold the profit of performing **i-th** job.

Acc_Prof	5	6	5	4	11	2

Now let's denote **position 2** with **i**, and **position 1** will be denoted with **j**. Our strategy will be to iterate **j** from **1** to **i-1** and after each iteration, we will increment **i** by 1, until **i** becomes **n+1**.

```
                          j       i
```

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

We check if **Job[i]** and **Job[j]** overlap, that is, if the **finish time** of **Job[j]** is greater than **Job[i]**'s start time, then these two jobs can't be done together. However, if they don't overlap, we'll check if **Acc_Prof[j] + Profit[i] > Acc_Prof[i]**. If this is the case, we will update **Acc_Prof[i] = Acc_Prof[j] + Profit[i]**. That is:

```
if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
    endif
endif
```

Here **Acc_Prof[j] + Profit[i]** represents the accumulated profit of doing these two jobs toegther. Let's check it for our example:

Here **Job[j]** overlaps with **Job[i]**. So these to can't be done together. Since our **j** is equal to **i-1**, we increment the value of **i** to **i+1** that is **3**. And we make **j = 1**.

```
                            j                   i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Now **Job[j]** and **Job[i]** don't overlap. The total amount of profit we can make by picking these two jobs is: **Acc_Prof[j] + Profit[i] = 5 + 5 = 10** which is greater than **Acc_Prof[i]**. So we update **Acc_Prof[i] = 10**. We also increment **j** by 1. We get,

```
                                    j         i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Here, **Job[j]** overlaps with **Job[i]** and **j** is also equal to **i-1**. So we increment **i** by 1, and make **j = 1**. We get,

```
                            j                             i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    10   |    4    |    11   |    2    |
```

```
+------------------------+---------+---------+---------+---------+---------+---------+
```

Now, **Job[j]** and **Job[i]** don't overlap, we get the accumulated profit **5 + 4 = 9**, which is greater than **Acc_Prof[i]**. We update **Acc_Prof[i] = 9** and increment **j** by 1.

```
                                    j                   i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    9    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

Again **Job[j]** and **Job[i]** don't overlap. The accumulated profit is: **6 + 4 = 10**, which is greater than **Acc_Prof[i]**. We again update **Acc_Prof[i] = 10**. We increment **j** by 1. We get:

```
                                          j         i

+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    10   |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

If we continue this process, after iterating through the whole table using **i**, our table will finally look like:

```
+------------------------+---------+---------+---------+---------+---------+---------+
|          Name          |    D    |    A    |    F    |    B    |    E    |    C    |
+------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Profit         |    5    |    6    |    5    |    4    |    11   |    2    |
+------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof       |    5    |    6    |    10   |    14   |    17   |    8    |
+------------------------+---------+---------+---------+---------+---------+---------+
```

* A few steps have been skipped to make the document shorter.

If we iterate through the array **Acc_Prof**, we can find out the maximum profit to be **17**! The pseudo-code:

```
Procedure WeightedJobScheduling(Job)
sort Job according to finish time in non-decreasing order
for i -> 2 to n
    for j -> 1 to i-1
        if Job[j].finish_time <= Job[i].start_time
            if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
                Acc_Prof[i] = Acc_Prof[j] + Profit[i]
```

```
            endif
        endif
    endfor
endfor

maxProfit = 0
for i -> 1 to n
    if maxProfit < Acc_Prof[i]
        maxProfit = Acc_Prof[i]
return maxProfit
```

The complexity of populating the **Acc_Prof** array is **O(n2).** The array traversal takes **O(n)**. So the total complexity of this algorithm is **O(n2).**

Now, If we want to find out which jobs were performed to get the maximum profit, we need to traverse the array in reverse order and if the **Acc_Prof** matches the **maxProfit**, we will push the **name** of the job in a **stack** and subtract **Profit** of that job from **maxProfit**. We will do this until our **maxProfit > 0** or we reach the beginning point of the **Acc_Prof** array. The pseudo-code will look like:

```
Procedure FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
for i -> n down to 0 and maxProfit > 0
    if maxProfit is equal to Acc_Prof[i]
        S.push(Job[i].name
        maxProfit = maxProfit - Job[i].profit
    endif
endfor
```

The complexity of this procedure is: **O(n)**.

One thing to remember, if there are multiple job schedules that can give us maximum profit, we can only find one job schedule via this procedure.

Section 14.3: Longest Common Subsequence

If we are given with the two strings we have to find the longest common sub-sequence present in both of them.

Example

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

Implementation in Java

```
public class LCS {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str1 = "AGGTAB";
        String str2 = "GXTXAYB";
        LCS obj = new LCS();
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        System.out.println(obj.lcs2(str1, str2));
    }

    //Recursive function
    public int lcs(String str1, String str2, int m, int n){
```

```
        if(m==0 || n==0)
            return 0;
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }

    //Iterative function
    public int lcs2(String str1, String str2){
        int lcs[][] = new int[str1.length()+1][str2.length()+1];

        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                if(i==0 || j== 0){
                    lcs[i][j] = 0;
                }
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{
                    lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
                }
            }
        }

        return lcs[str1.length()][str2.length()];
    }

}
```

Output

```
4
```

Section 14.4: Fibonacci Number

Bottom up approach for printing the nth Fibonacci number using Dynamic Programming.

Recursive Tree

```
                        fib(5)
                    /             \
               fib(4)               fib(3)
             /      \               /     \
         fib(3)      fib(2)        fib(2)    fib(1)
        /     \        /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Overlapping Sub-problems

Here fib(0),fib(1) and fib(3) are the overlapping sub-problems.fib(0) is getting repeated 3 times, fib(1) is getting repeated 5 times and fib(3) is getting repeated 2 times.

Implementation

```
public int fib(int n){
        int f[] = new int[n+1];
```

```
        f[0]=0;f[1]=1;
        for(int i=2;i<=n;i++){
            f[i]=f[i-1]+f[i-2];
        }
        return f[n];
    }
```

Time Complexity

O(n)

Section 14.5: Longest Common Substring

Given 2 string str1 and str2 we have to find the length of the longest common substring between them.

Examples

Input : X = "abcdxyz", y = "xyzabcd" Output : 4

The longest common substring is "abcd" and is of length 4.

Input : X = "zxabcdezy", y = "yzabcdezx" Output : 6

The longest common substring is "abcdez" and is of length 6.

Implementation in Java

```
public int getLongestCommonSubstring(String str1,String str2){
        int arr[][] = new int[str2.length()+1][str1.length()+1];
        int max = Integer.MIN_VALUE;
        for(int i=1;i<=str2.length();i++){
            for(int j=1;j<=str1.length();j++){
                if(str1.charAt(j-1) == str2.charAt(i-1)){
                    arr[i][j] = arr[i-1][j-1]+1;
                    if(arr[i][j]>max)
                        max = arr[i][j];
                }
                else
                    arr[i][j] = 0;
            }
        }
        return max;
    }
```

Time Complexity

O(m*n)

Chapter 15: Applications of Dynamic Programming

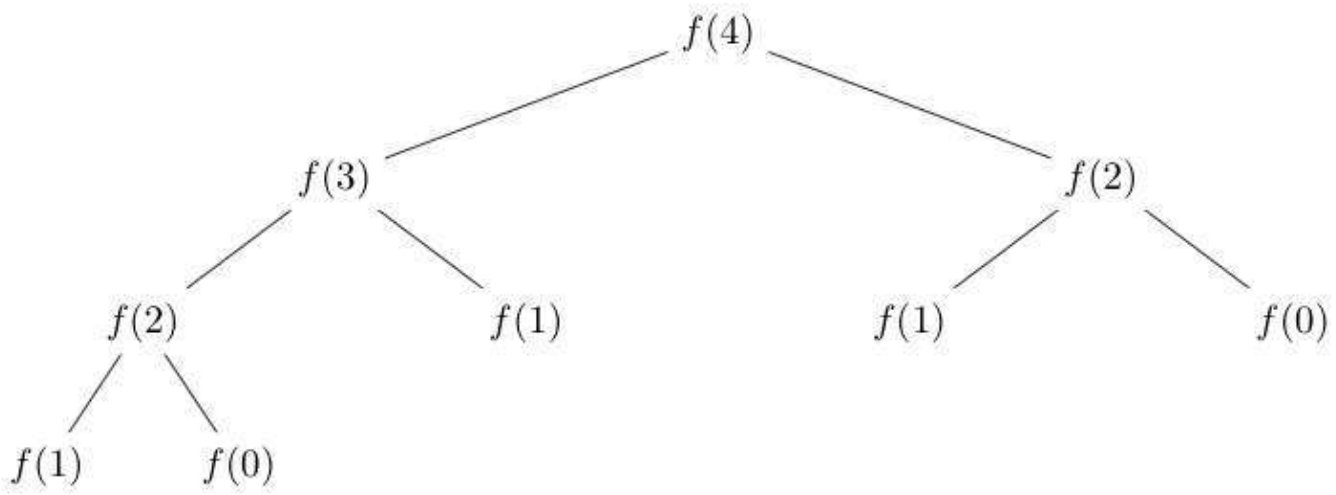
The basic idea behind dynamic programming is breaking a complex problem down to several small and simple problems that are repeated. If you can identify a simple subproblem that is repeatedly calculated, odds are there is a dynamic programming approach to the problem.

As this topic is titled *Applications of Dynamic Programming*, it will focus more on applications rather than the process of creating dynamic programming algorithms.

Section 15.1: Fibonacci Numbers

Fibonacci Numbers are a prime subject for dynamic programming as the traditional recursive approach makes a lot of repeated calculations. In these examples I will be using the base case of `f(0) = f(1) = 1`.

Here is an example recursive tree for `fibonacci(4)`, note the repeated computations:



Non-Dynamic Programming `O(2^n)` Runtime Complexity, `O(n)` Stack complexity

```
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

This is the most intuitive way to write the problem. At most the stack space will be `O(n)` as you descend the first recursive branch making calls to `fibonacci(n-1)` until you hit the base case `n < 2`.

The `O(2^n)` runtime complexity proof that can be seen here: Computational complexity of Fibonacci Sequence. The main point to note is that the runtime is exponential, which means the runtime for this will double for every subsequent term, `fibonacci(15)` will take twice as long as `fibonacci(14)`.

Memoized `O(n)` Runtime Complexity, `O(n)` Space complexity, `O(n)` Stack complexity

```
memo = []
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonacci(n):
    if len(memo) > n:
        return memo[n]
```

```
    result = fibonacci(n-1) + fibonacci(n-2)
    memo.append(result) # f(n) = f(n-1) + f(n-2)
    return result
```

With the memoized approach we introduce an array that can be thought of as all the previous function calls. The location `memo[n]` is the result of the function call `fibonacci(n)`. This allows us to trade space complexity of `O(n)` for a `O(n)` runtime as we no longer need to compute duplicate function calls.

Iterative Dynamic Programming `O(n)` Runtime complexity, `O(n)` Space complexity, No recursive stack

```
def fibonacci(n):
    memo = [1,1] # f(0) = 1, f(1) = 1

    for i in range(2, n+1):
         memo.append(memo[i-1] + memo[i-2])

    return memo[n]
```

If we break the problem down into it's core elements you will notice that in order to compute `fibonacci(n)` we need `fibonacci(n-1)` and `fibonacci(n-2)`. Also we can notice that our base case will appear at the end of that recursive tree as seen above.

With this information, it now makes sense to compute the solution backwards, starting at the base cases and working upwards. Now in order to calculate `fibonacci(n)` we first calculate **all** the fibonacci numbers up to and through `n`.

This main benefit here is that we now have eliminated the recursive stack while keeping the `O(n)` runtime. Unfortunately, we still have an `O(n)` space complexity but that can be changed as well.

Advanced Iterative Dynamic Programming `O(n)` Runtime complexity, `O(1)` Space complexity, No recursive stack

```
def fibonacci(n):
    memo = [1,1] # f(1) = 1, f(2) = 1

    for i in range (2, n):
        memo[i%2] = memo[0] + memo[1]

    return memo[n%2]
```

As noted above, the iterative dynamic programming approach starts from the base cases and works to the end result. The key observation to make in order to get to the space complexity to `O(1)` (constant) is the same observation we made for the recursive stack - we only need `fibonacci(n-1)` and `fibonacci(n-2)` to build `fibonacci(n)`. This means that we only need to save the results for `fibonacci(n-1)` and `fibonacci(n-2)` at any point in our iteration.

To store these last 2 results I use an array of size 2 and simply flip which index I am assigning to by using `i % 2` which will alternate like so: `0, 1, 0, 1, 0, 1, ..., i % 2`.

I add both indexes of the array together because we know that addition is commutative (`5 + 6 = 11` and `6 + 5 == 11`). The result is then assigned to the older of the two spots (denoted by `i % 2`). The final result is then stored at the position `n%2`

Notes

- It is important to note that sometimes it may be best to come up with a iterative memoized solution for

functions that perform large calculations repeatedly as you will build up a cache of the answer to the function calls and subsequent calls may be `O(1)` if it has already been computed.

Chapter 16: Kruskal's Algorithm

Section 16.1: Optimal, disjoint-set based implementation

We can do two things to improve the simple and sub-optimal disjoint-set subalgorithms:

1. **Path compression heuristic**: `findSet` does not need to ever handle a tree with height bigger than 2. If it ends up iterating such a tree, it can link the lower nodes directly to the root, optimizing future traversals;

```
subalgo findSet(v: a node):
    if v.parent != v
        v.parent = findSet(v.parent)
    return v.parent
```

2. **Height-based merging heuristic**: for each node, store the height of its subtree. When merging, make the taller tree the parent of the smaller one, thus not increasing anyone's height.

```
subalgo unionSet(u, v: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    if vRoot == uRoot:
        return

    if vRoot.height < uRoot.height:
        vRoot.parent = uRoot
    else if vRoot.height > uRoot.height:
        uRoot.parent = vRoot
    else:
        uRoot.parent = vRoot
        uRoot.height =  uRoot.height + 1
```

This leads to `O(alpha(n))` time for each operation, where `alpha` is the inverse of the fast-growing Ackermann function, thus it is very slow growing, and can be considered `O(1)` for practical purposes.

This makes the entire Kruskal's algorithm `O(m log m + m) = O(m log m)`, because of the initial sorting.

Note

Path compression may reduce the height of the tree, hence comparing heights of the trees during union operation might not be a trivial task. Hence to avoid the complexity of storing and calculating the height of the trees the resulting parent can be picked randomly:

```
subalgo unionSet(u, v: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    if vRoot == uRoot:
        return
    if random() % 2 == 0:
        vRoot.parent = uRoot
    else:
        uRoot.parent = vRoot
```

In practice this randomised algorithm together with path compression for `findSet` operation will result in

comparable performance, yet much simpler to implement.

Section 16.2: Simple, more detailed implementation

In order to efficiently handle cycle detection, we consider each node as part of a tree. When adding an edge, we check if its two component nodes are part of distinct trees. Initially, each node makes up a one-node tree.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = a forest of trees, initially each tree is a node in the graph
    for each edge e in G:
        if the root of the tree that e.first belongs to is not the same
        as the root of the tree that e.second belongs to:
            connect one of the roots to the other, thus merging two trees

    return MST, which now a single-tree forest
```

Section 16.3: Simple, disjoint-set based implementation

The above forest methodology is actually a disjoint-set data structure, which involves three main operations:

```
subalgo makeSet(v: a node):
    v.parent = v    <- make a new tree rooted at v


subalgo findSet(v: a node):
    if v.parent == v:
        return v
    return findSet(v.parent)

subalgo unionSet(v, u: nodes):
    vRoot = findSet(v)
    uRoot = findSet(u)

    uRoot.parent = vRoot

algorithm kruskalMST(G: a graph):
    sort Gs edges by their value
    for each node n in G:
        makeSet(n)
    for each edge e in G:
        if findSet(e.first) != findSet(e.second):
            unionSet(e.first, e.second)
```

This naive implementation leads to `O(n log n)` time for managing the disjoint-set data structure, leading to `O(m*n log n)` time for the entire Kruskal's algorithm.

Section 16.4: Simple, high level implementation

Sort the edges by value and add each one to the MST in sorted order, if it doesn't create a cycle.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    MST = an empty graph
    for each edge e in G:
        if adding e to MST does not create a cycle:
            add e to MST
```

```
    return MST
```

Chapter 17: Greedy Algorithms

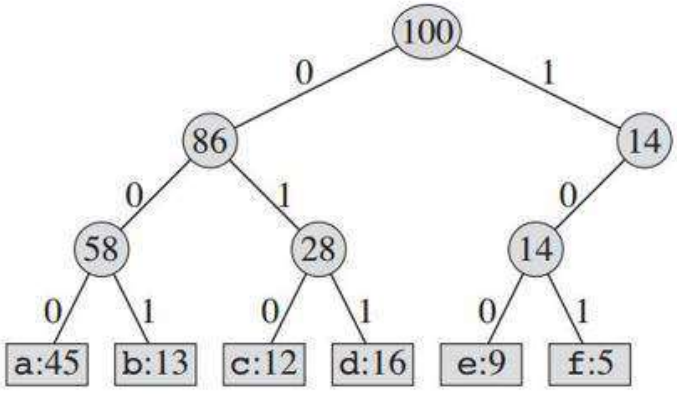
Section 17.1: Huffman Coding

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Huffman code was proposed by David A. Huffman in 1951.

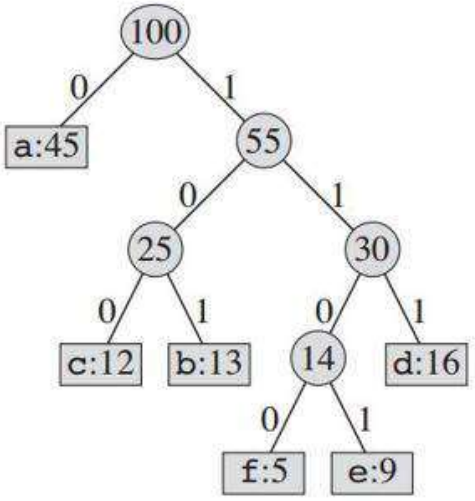
Suppose we have a 100,000-character data file that we wish to store compactly. We assume that there are only 6 different characters in that file. The frequency of the characters are given by:

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *Binary Character Code* in which each character is represented by a unique binary string, which we call a **codeword**.



Fixed-length Codeword



Variable-length Codeword

The constructed tree will provide us with:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

If we use a **fixed-length code**, we need three bits to represent 6 characters. This method requires 300,000 bits to code the entire file. Now the question is, can we do better?

A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. This code requires: (45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 + 5 X 4) X 1000 = 224000 bits to represent the file, which saves approximately 25% of memory.

One thing to remember, we consider here only codes in which no codeword is also a prefix of some other codeword. These are called *prefix codes.* For variable-length coding, we code the 3-character file *abc* as 0.101.100 = 0101100, where "." denotes the concatenation.

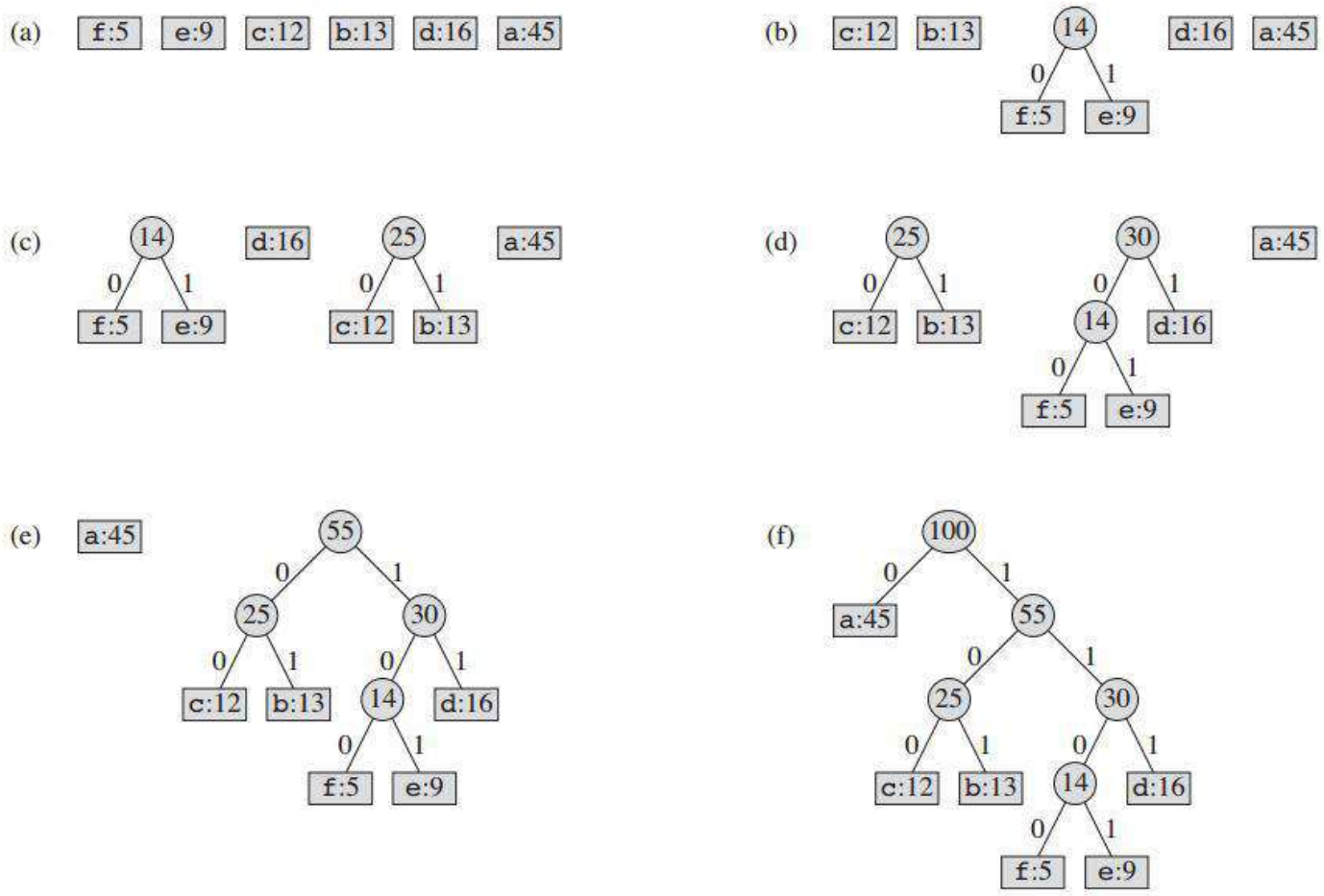
Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. For example, 001011101 parses uniquely as 0.0.101.1101, which decodes to *aabe*. In short, all the combinations of binary representations are unique. Say for example, if one letter is denoted by 110, no other letter will be denoted by 1101 or 1100. This is because you might face confusion on whether to select 110 or to continue on concatenating the next bit and select that one.

Compression Technique:

The technique works by creating a *binary tree* of nodes. These can stored in a regular array, the size of which depends on the number of symbols, **n**. A node can either be a *leaf node* or an *internal node*. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. *Priority queue* is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the two nodes of highest priority from the queue.
    2. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.
    3. Add the new node to the queue.
3. The remaining node is the root node and the Huffman tree is complete.

For our example:



The pseudo-code looks like:

```
Procedure Huffman(C):     // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
end while
Return Q
```

Although linear-time given sorted input, in general cases of arbitrary input, using this algorithm requires pre-sorting. Thus, since sorting takes **O(nlogn)** time in general cases, both methods have same complexity.

Since **n** here is the number of symbols in the alphabet, which is typically very small number (compared to the length of the message to be encoded), time complexity is not very important in the choice of this algorithm.

Decompression Technique:

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired

character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately. The pseudo-code:

```
Procedure HuffmanDecompression(root, S):   // root represents the root of Huffman Tree
n := S.length                              // S refers to bit-stream to be decompressed
for i := 1 to n
    current = root
    while current.left != NULL and current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
    print current.symbol
endfor
```

Greedy Explanation:
Huffman coding looks at the occurrence of each character and stores it as a binary string in an optimal way. The idea is to assign variable-length codes to input input characters, length of the assigned codes are based on the frequencies of corresponding characters. We create a binary tree and operate on it in bottom-up manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

References:

- Introduction to Algorithms - Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen
- Huffman Coding - Wikipedia
- Discrete Mathematics and Its Applications - Kenneth H. Rosen

Section 17.2: Activity Selection Problem

The Problem

You have a set of things to do (activities). Each activity has a start time and a end time. You aren't allowed to perform more than one activity at a time. Your task is to find a way to perform the maximum number of activities.

For example, suppose you have a selection of classes to choose from.

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

Remember, you can't take two classes at the same time. That means you can't take class 1 and 2 because they share a common time 10.30 A.M to 11.00 A.M. However, you can take class 1 and 3 because they don't share a common time. So your task is to take maximum number of classes as possible without any overlap. How can you do that?

Analysis

Lets think for the solution by greedy approach.First of all we randomly chose some approach and check that will work or not.

- **sort the activity by start time** that means which activity start first we will take them first. then take first to last from sorted list and check it will intersect from previous taken activity or not. If the current activity is not intersect with the previously taken activity, we will perform the activity otherwise we will not perform. this approach will work for some cases like

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	11.00AM

the sorting order will be 4-->1-->2-->3 .The activity 4--> 1--> 3 will be performed and the activity 2 will be skipped. the maximum 3 activity will be performed. It works for this type of cases. but it will fail for some cases. Lets apply this approach for the case

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	3.00P.M

The sort order will be 4-->1-->2-->3 and only activity 4 will be performed but the answer can be activity 1-->3 or 2-->3 will be performed. So our approach will not work for the above case. Let's try another approach

- **Sort the activity by time duration** that means perform the shortest activity first. that can solve the previous problem . Although the problem is not completely solved. There still some cases that can fail the solution. apply this approach on the case bellow.

Activity No.	start time	end time
1	6.00 A.M	11.40A.M
2	11.30 A.M	12.00P.M
3	11.40 P.M	2.00P.M

if we sort the activity by time duration the sort order will be 2--> 3 --->1 . and if we perform activity No. 2 first then no other activity can be performed. But the answer will be perform activity 1 then perform 3 . So we can perform maximum 2 activity.So this can not be a solution of this problem. We should try a different approach.

The solution

- **Sort the Activity by ending time** that means the activity finishes first that come first. the algorithm is given below

> 1. Sort the activities by its ending times.
> 2. If the activity to be performed do not share a common time with the activities that previously performed, perform the activity.

Lets analyse the first example

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

sort the activity by its ending times , So sort order will be 1-->5-->2-->4-->3.. the answer is 1-->3 these two activities will be performed. ans that's the answer. here is the sudo code.

> 1. sort: activities
> 2. perform first activity from the sorted list of activities.
> 3. Set : Current_activity := first activity
> 4. set: end_time := end_time of Current activity
> 5. go to next activity if exist, if not exist terminate .
> 6. if start_time of current activity <= end_time : perform the activity and go to 4
> 7. else: got to 5.

see here for coding help http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

Section 17.3: Change-making problem

Given a money system, is it possible to give an amount of coins and how to find a minimal set of coins corresponding to this amount.

Canonical money systems. For some money system, like the ones we use in the real life, the "intuitive" solution works perfectly. For example, if the different euro coins and bills (excluding cents) are 1€, 2€, 5€, 10€, giving the highest coin or bill until we reach the amount and repeating this procedure will lead to the minimal set of coins.

We can do that recursively with OCaml :

```
(* assuming the money system is sorted in decreasing order *)
let change_make money_system amount =
  let rec loop given amount =
    if amount = 0 then given
    else
      (* we find the first value smaller or equal to the remaining amount *)
      let coin = List.find ((>=) amount) money_system in
      loop (coin::given) (amount - coin)
  in loop [] amount
```

These systems are made so that change-making is easy. The problem gets harder when it comes to arbitrary money system.

General case. How to give 99€ with coins of 10€, 7€ and 5€? Here, giving coins of 10€ until we are left with 9€ leads obviously to no solution. Worse than that a solution may not exist. This problem is in fact np-hard, but acceptable solutions mixing **greediness** and **memoization** exist. The idea is to explore all the possibilies and pick the one with the minimal number of coins.

To give an amount X > 0, we choose a piece P in the money system, and then solve the sub-problem corresponding to X-P. We try this for all the pieces of the system. The solution, if it exists, is then the smallest path that led to 0.

Here an OCaml recursive function corresponding to this method. It returns None, if no solution exists.

```
(* option utilities *)
let optmin x y =
  match x,y with
  | None,a | a,None -> a
  | Some x, Some y-> Some (min x y)

let optsucc = function
  | Some x -> Some (x+1)
  | None -> None

(* Change-making problem*)
let change_make money_system amount =
  let rec loop n =
    let onepiece acc piece =
      match n - piece with
      | 0 -> (*problem solved with one coin*)
             Some 1
      | x -> if x < 0 then
               (*we don't reach 0, we discard this solution*)
               None
             else
               (*we search the smallest path different to None with the remaining pieces*)
               optmin (optsucc (loop x)) acc
    in
    (*we call onepiece forall the pieces*)
    List.fold_left onepiece None money_system
  in loop amount
```

Note: We can remark that this procedure may compute several times the change set for the same value. In practice, using memoization to avoid these repetitions leads to faster (way faster) results.

Chapter 18: Applications of Greedy technique

Section 18.1: Offline Caching

The caching problem arises from the limitation of finite space. Lets assume our cache `C` has `k` pages. Now we want to process a sequence of `m` item requests which must have been placed in the cache before they are processed.Of course if `m<=k` then we just put all elements in the cache and it will work, but usually is `m>>k`.

We say a request is a **cache hit**, when the item is already in cache, otherwise its called a **cache miss**. In that case we must bring the requested item into cache and evict another, assuming the cache is full. The Goal is a eviction schedule that **minimizes the number of evictions**.

There are numerous greedy strategies for this problem, lets look at some:

1. **First in, first out (FIFO)**: The oldest page gets evicted
2. **Last in, first out (LIFO)**: The newest page gets evicted
3. **Last recent out (LRU)**: Evict page whose most recent access was earliest
4. **Least frequently requested(LFU)**: Evict page that was least frequently requested
5. **Longest forward distance (LFD)**: Evict page in the cache that is not requested until farthest in the future.

Attention: For the following examples we evict the page with the smallest index, if more than one page could be evicted.

Example (FIFO)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	d	d	d	a	a	a	d	d	d	f	f	f	c
cache 2	b	b	b	e	e	e	e	c	c	c	e	e	e	b	b	b
cache 3	c	c	c	c	b	b	b	b	f	f	f	a	a	a	e	e
cache miss			x	x	x		x	x	x	x	x	x	x	x	x	x

Thirteen cache misses by sixteen requests does not sound very optimal, lets try the same example with another strategy:

Example (LFD)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	e	e	e	e	e	e	e	e	e	e	e	e	c
cache 2	b	b	b	b	b	b	a	a	a	a	a	a	f	f	f	f
cache 3	c	c	c	c	c	c	c	c	f	d	d	d	d	b	b	b
cache miss			x	x		x		x	x			x	x			x

Eight cache misses is a lot better.

Selftest: Do the example for LIFO, LFU, RFU and look what happend.

The following example programm (written in C++) consists of two parts:

The skeleton is a application, which solves the problem dependent on the chosen greedy strategy:

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize     = 3;
const int requestLength = 16;

const char request[]    = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c'};
char cache[]            = {'a','b','c'};

// for reset
char originalCache[]    = {'a','b','c'};


class Strategy {

public:
    Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    // calculate which cache place should be used
    virtual int apply(int requestIndex)                                     = 0;

    // updates information the strategy needs
    virtual void update(int cachePlace, int requestIndex, bool cacheMiss)    = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    // calculate where to put request
    int cachePlace = strategy->apply(requestIndex);

    // proof whether its a cache hit or a cache miss
    bool isMiss = request[requestIndex] != cache[cachePlace];

    // update strategy (for example recount distances)
    strategy->update(cachePlace, requestIndex, isMiss);

    // write to cache
    cache[cachePlace] = request[requestIndex];

    return isMiss;
}

int main()
{
    Strategy* selectedStrategy[] = { new FIFO, new LIFO, new LRU, new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {
        // reset cache
        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];

        cout <<"\nStrategy: " << selectedStrategy[strat]->strategyName << endl;

        cout << "\nCache initial: (";
        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] << ",";
```

```
        cout << cache[cacheSize-1] << ")\n\n";

        cout << "Request\t";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "\t";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {
            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;

            cout << "   " << request[i] << "\t";
            for (int l=0; l < cacheSize; ++l) cout << "   " << cache[l] << "\t";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "\nTotal cache misses: " << cntMisses << endl;
    }

    for(int i=0; i<5; ++i) delete selectedStrategy[i];
}
```

The basic idea is simple: for every request I have two calls two my strategy:

1. **apply**: The strategy has to tell the caller which page to use
2. **update**: After the caller uses the place, it tells the strategy whether it was a miss or not. Then the strategy may update its internal data. The strategy **LFU** for example has to update the hit frequency for the cache pages, while the **LFD** strategy has to recalculate the distances for the cache pages.

Now lets look of example implementations for our five strategies:

FIFO

```
class FIFO : public Strategy {
public:
    FIFO() : Strategy("FIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
```

```
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO just needs the information how long a page is in the cache (and of course only relative to the other pages). So the only thing to do is wait for a miss and then make the pages, which where not evicted older. For our example above the program solution is:

```
Strategy: FIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          d          e          c          x
  b          d          e          b          x
  b          d          e          b
  a          a          e          b          x
  c          a          c          b          x
  f          a          c          f          x
  d          d          c          f          x
  e          d          e          f          x
  a          d          e          a          x
  f          f          e          a          x
  b          f          b          a          x
  e          f          b          e          x
  c          c          b          e          x

Total cache misses: 13
```

Thats exact the solution from above.

LIFO

```
class LIFO : public Strategy {
public:
    LIFO() : Strategy("LIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int newest = 0;
```

```
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] < age[newest])
                newest = i;
        }

        return newest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

The implementation of **LIFO** is more or less the same as by **FIFO** but we evict the youngest not the oldest page. The program results are:

```
Strategy: LIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          e          b          c          x
  b          e          b          c
  b          e          b          c
  a          a          b          c          x
  c          a          b          c
  f          f          b          c          x
  d          d          b          c          x
  e          e          b          c          x
  a          a          b          c          x
  f          f          b          c          x
  b          f          b          c
  e          e          b          c          x
  c          e          b          c

Total cache misses: 9
```

LRU

```
class LRU : public Strategy {
public:
    LRU() : Strategy("LRU")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    // here oldest mean not used the longest
    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // all old pages get older, the used one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

In case of **LRU** the strategy is independent from what is at the cache page, its only interest is the last usage. The programm results are:

```
Strategy: LRU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          b          d          e          x
  b          b          d          e
  a          b          a          e          x
  c          b          a          c          x
  f          f          a          c          x
  d          f          d          c          x
  e          f          d          e          x
  a          a          d          e          x
```

```
  f          a          f          e          x
  b          a          f          b          x
  e          e          f          b          x
  c          e          c          b          x

Total cache misses: 13
```

LFU

```
class LFU : public Strategy {
public:
    LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int least = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(cacheMiss)
            requestFrequency[cachePos] = 1;

        else
            ++requestFrequency[cachePos];
    }

private:

    // how frequently was the page used
    int requestFrequency[cacheSize];
};
```

LFU evicts the page uses least often. So the update strategy is just to count every access. Of course after a miss the count resets. The program results are:

```
Strategy: LFU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          a          b          e          x
  b          a          b          e
  a          a          b          e
```

```
  c          a          b          c          x
  f          a          b          f          x
  d          a          b          d          x
  e          a          b          e          x
  a          a          b          e
  f          a          b          f          x
  b          a          b          f
  e          a          b          e          x
  c          a          b          c          x

Total cache misses: 10
```

LFD

```
class LFD : public Strategy {
public:
    LFD() : Strategy("LFD")
    {
        // precalc next usage before starting to fullfill requests
        for (int i=0; i<cacheSize; ++i) nextUse[i] = calcNextUse(-1, cache[i]);
    }

    int apply(int requestIndex) override
    {
        int latest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(nextUse[i] > nextUse[latest])
                latest = i;
        }

        return latest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
            nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
    }

private:

    int calcNextUse(int requestPosition, char pageItem)
    {
        for(int i = requestPosition+1; i < requestLength; ++i)
        {
            if (request[i] == pageItem)
                return i;
        }

        return requestLength + 1;
    }

    // next usage of page
    int nextUse[cacheSize];
};
```

The **LFD** strategy is different from everyone before. Its the only strategy that uses the future requests for its decission who to evict. The implementation uses the function `calcNextUse` to get the page which next use is

farthest away in the future. The program solution is equal to the solution by hand from above:

```
Strategy: LFD

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          b          d          x
  e          a          b          e          x
  b          a          b          e
  b          a          b          e
  a          a          b          e
  c          a          c          e          x
  f          a          f          e          x
  d          a          d          e          x
  e          a          d          e
  a          a          d          e
  f          f          d          e          x
  b          b          d          e          x
  e          b          d          e
  c          c          d          e          x

Total cache misses: 8
```

The greedy strategy **LFD** is indeed the only optimal strategy of the five presented. The proof is rather long and can be found here or in the book by Jon Kleinberg and Eva Tardos (see sources in remarks down below).

Algorithm vs Reality

The **LFD** strategy is optimal, but there is a big problem. Its an optimal **offline** solution. In praxis caching is usually an **online** problem, that means the strategy is useless because we cannot now the next time we need a particular item. The other four strategies are also **online** strategies. For online problems we need a general different approach.

Section 18.2: Ticket automat

First simple Example:

You have a ticket automat which gives exchange in coins with values 1, 2, 5, 10 and 20. The dispension of the exchange can be seen as a series of coin drops until the right value is dispensed. We say a dispension is **optimal** when its **coin count is minimal** for its value.

Let `M` in `[1,50]` be the price for the ticket `T` and `P` in `[1,50]` the money somebody paid for `T`, with `P >= M`. Let `D=P-M`. We define the **benefit** of a step as the difference between `D` and `D-c` with `c` the coin the automat dispense in this step.

The **Greedy Technique** for the exchange is the following pseudo algorithmic approach:

Step 1: while `D > 20` dispense a 20 coin and set `D = D - 20`
Step 2: while `D > 10` dispense a 10 coin and set `D = D - 10`
Step 3: while `D > 5` dispense a 5 coin and set `D = D - 5`
Step 4: while `D > 2` dispense a 2 coin and set `D = D - 2`
Step 5: while `D > 1` dispense a 1 coin and set `D = D - 1`

Afterwards the sum of all coins clearly equals `D`. Its a **greedy algorithm** because after each step and after each repitition of a step the benefit is maximized. We cannot dispense another coin with a higher benefit.

Now the ticket automat as program (in C++):

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// read some coin values, sort them descending,
// purge copies and guaratee the 1 coin is in it
std::vector<unsigned int> readInCoinValues();

int main()
{
    std::vector<unsigned int> coinValues;   // Array of coin values ascending
    int ticketPrice;                        // M in example
    int paidMoney;                          // P in example

    // generate coin values
    coinValues = readInCoinValues();

    cout << "ticket price: ";
    cin >> ticketPrice;

    cout << "money paid: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "No exchange money" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    // Here starts greedy

    // we save how many coins we have to give out
    std::vector<unsigned int> coinCount;

    for(auto coinValue  = coinValues.begin();
             coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;

        while (diffValue >= *coinValue)
        {
            diffValue -= *coinValue;
            countCoins++;
        }

        coinCount.push_back(countCoins);
    }

    // print out result
    cout << "the difference " << paidMoney - ticketPrice
         << " is paid with: " << endl;
```

```
    for(unsigned int i=0; i < coinValues.size(); ++i)
    {
        if(coinCount[i] > 0)
            cout << coinCount[i] << " coins with value "
                 << coinValues[i] << endl;
    }

    return 0;
}

std::vector<unsigned int> readInCoinValues()
{
    // coin values
    std::vector<unsigned int> coinValues;

    // make sure 1 is in vectore
    coinValues.push_back(1);

    // read in coin values (attention: error handling is omitted)
    while(true)
    {
        int coinValue;

        cout << "Coin value (<1 to stop): ";
        cin >> coinValue;

        if(coinValue > 0)
            coinValues.push_back(coinValue);

        else
            break;
    }

    // sort values
    sort(coinValues.begin(), coinValues.end(), std::greater<int>());

    // erase copies of same value
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    // print array
    cout << "Coin values: ";

    for(auto i : coinValues)
        cout << i << " ";

    cout << endl;

    return coinValues;
}
```

Be aware there is now input checking to keep the example simple. One example output:

```
Coin value (<1 to stop): 2
Coin value (<1 to stop): 4
Coin value (<1 to stop): 7
Coin value (<1 to stop): 9
Coin value (<1 to stop): 14
Coin value (<1 to stop): 4
Coin value (<1 to stop): 0
```

```
Coin values: 14 9 7 4 2 1
ticket price: 34
money paid: 67
the difference 33 is paid with:
2 coins with value 14
1 coins with value 4
1 coins with value 1
```

As long as `1` is in the coin values we now, that the algorithm will terminate, because:

- `D` strictly decreases with every step
- `D` is never `>0` and smaller than than the smallest coin `1` at the same time

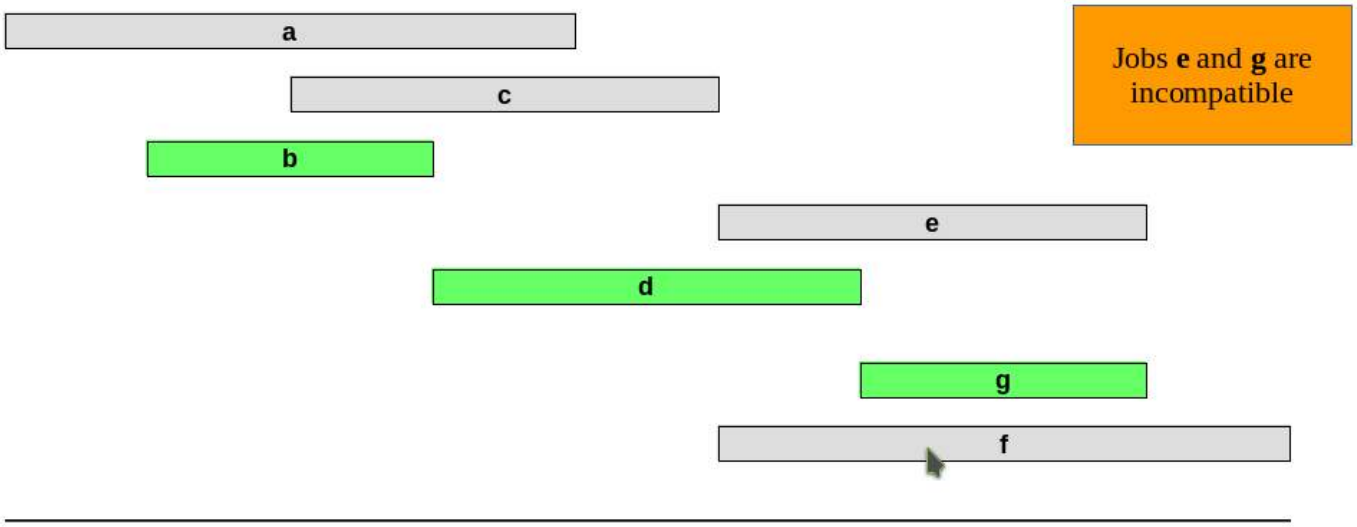
But the algorithm has two pitfalls:

1. Let `C` be the biggest coin value. The runtime is only polynomial as long as `D/C` is polynomial, because the representation of `D` uses only `log D` bits and the runtime is at least linear in `D/C`.
2. In every step our algorithm chooses the local optimum. But this is not sufficient to say that the algorithm finds the global optimal solution (see more information here or in the Book of Korte and Vygen).

A simple counter example: the coins are `1,3,4` and `D=6`. The optimal solution is clearly two coins of value `3` but greedy chooses `4` in the first step so it has to choose `1` in step two and three. So it gives no optimal soution. A possible optimal Algorithm for this example is based on **dynamic programming**.

Section 18.3: Interval Scheduling

We have a set of jobs `J={a,b,c,d,e,f,g}`. Let `j in J` be a job than its start at `sj` and ends at `fj`. Two jobs are compatible if they don't overlap. A picture as example:



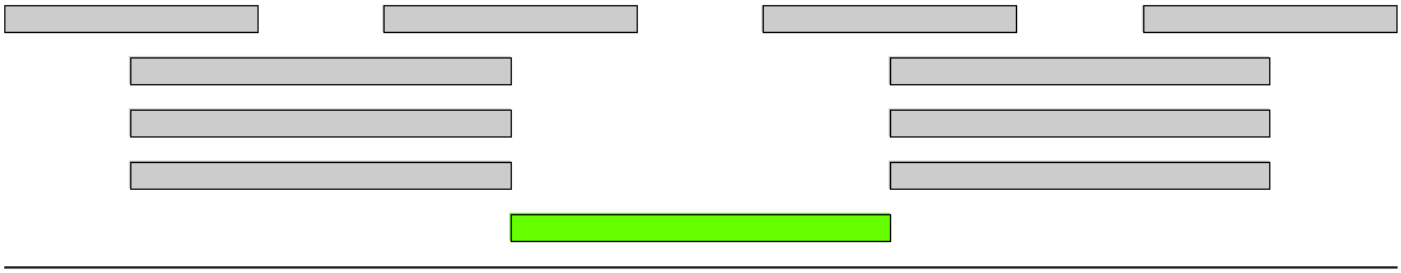
The goal is to find the **maximum subset of mutually compatible jobs**. There are several greedy approaches for this problem:

1. **Earliest start time**: Consider jobs in ascending order of `sj`
2. **Earliest finish time**: Consider jobs in ascending order of `fj`
3. **Shortest interval**: Consider jobs in ascending order of `fj-sj`
4. **Fewest conflicts**: For each job `j`, count the number of conflicting jobs `cj`

The question now is, which approach is really successfull. **Early start time** definetly not, here is a counter example

Shortest interval is not optimal either

and **fewest conflicts** may indeed sound optimal, but here is a problem case for this approach:



Which leaves us with **earliest finish time**. The pseudo code is quiet simple:

1. Sort jobs by finish time so that `f1<=f2<=...<=fn`
2. Let `A` be an empty set
3. for `j=1` to `n` if `j` is compatible to **all** jobs in `A` set `A=A+{j}`
4. `A` is a **maximum subset of mutually compatible jobs**

Or as C++ program:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};

// Job end times
const int endTimes[]   = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(startTimes[i], endTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                    { return p1.second < p2.second; });
```

```
    // step 2: empty set A
    vector<int> A;

    // step 3:
    for(int i=0; i<jobCnt; ++i)
    {
        auto job = jobs[i];
        bool isCompatible = true;

        for(auto jobIndex : A)
        {
            // test whether the actual job and the job from A are incompatible
            if(job.second >= jobs[jobIndex].first &&
               job.first  <= jobs[jobIndex].second)
            {
                isCompatible = false;
                break;
            }
        }

        if(isCompatible)
            A.push_back(i);
    }

    //step 4: print A
    cout << "Compatible: ";

    for(auto i : A)
        cout << "(" << jobs[i].first << "," << jobs[i].second << ") ";
    cout << endl;

    return 0;
}
```

The output for this example is: `Compatible: (1,3) (4,5) (6,8) (9,10)`

The implementation of the algorithm is clearly in Θ(n^2). There is a Θ(n log n) implementation and the interested reader may continue reading below (Java Example).

Now we have a greedy algorithm for the interval scheduling problem, but is it optimal?

Proposition: The greedy algorithm **earliest finish time** is optimal.

Proof:*(by contradiction)*

Assume greedy is not optimal and `i1,i2,...,ik` denote the set of jobs selected by greedy. Let `j1,j2,...,jm` denote the set of jobs in an **optimal** solution with `i1=j1,i2=j2,...,ir=jr` for the **largest possible** value of `r`.

The job `i(r+1)` exists and finishes before `j(r+1)` (earliest finish). But than is `j1,j2,...,jr,i(r+1),j(r+2),...,jm` also a **optimal** solution and for all `k` in `[1,(r+1)]` is `jk=ik`. thats a **contradiction** to the maximality of `r`. This concludes the proof.

This second example demonstrates that there are usually many possible greedy strategies but only some or even none might find the optimal solution in every instance.

Below is a Java program that runs in Θ(n log n)

```
import java.util.Arrays;
```

```
import java.util.Comparator;

class Job
{
    int start, finish, profit;

    Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}


class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}

public class WeightedIntervalScheduling
{
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;

        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                else
                    return mid;
            }
            else
                hi = mid - 1;
        }

        return -1;
    }

    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());

        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;

        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
                inclProf += table[l];

            table[i] = Math.max(inclProf, table[i-1]);
```

```
        }

        return table[n-1];
    }

    public static void main(String[] args)
    {
        Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                      new Job(6, 19, 100), new Job(2, 100, 200)};

        System.out.println("Optimal profit is " + schedule(jobs));
    }
}
```

And the expected output is:

```
Optimal profit is 250
```

Section 18.4: Minimizing Lateness

There are numerous problems minimizing lateness, here we have a single resource which can only process one job at a time. Job `j` requires `tj` units of processing time and is due at time `dj`. if `j` starts at time `sj` it will finish at time `fj=sj+tj`. We define lateness `L=max{0,fj-dh}` for all `j`. The goal is to minimize the **maximum lateness** L.

	1	**2**	**3**	**4**	**5**	**6**
tj	3	2	1	4	3	2
dj	6	8	9	9	10	11

Job	**3**	**2**	**2**	**5**	**5**	**5**	**4**	**4**	**4**	**4**	**1**	**1**	**1**	**6**	**6**
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lj	-8		-5			-4				1			**7**		4

The solution `L=7` is obviously not optimal. Lets look at some greedy strategies:

1. **Shortest processing time first**: schedule jobs in ascending order og processing time j`
2. **Earliest deadline first**: Schedule jobs in ascending order of deadline `dj`
3. **Smallest slack**: schedule jobs in ascending order of slack `dj-tj`

Its easy to see that **shortest processing time first** is not optimal a good counter example is

	1	**2**
tj	1	5
dj	10	5

the **smallest stack** solution has simillar problems

	1	**2**
tj	1	5
dj	3	5

the last strategy looks valid so we start with some pseudo code:

1. Sort `n` jobs by due time so that `d1<=d2<=...<=dn`
2. Set `t=0`
3. for `j=1` to `n`
    - Assign job `j` to interval `[t,t+tj]`

- set `sj=t` and `fj=t+tj`
- set `t=t+tj`

4. return intervals `[s1,f1],[s2,f2],...,[sn,fn]`

And as implementation in C++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};

// Job end times
const int dueTimes[]     = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(processTimes[i], dueTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                  { return p1.second < p2.second; });

    // step 2: set t=0
    int t = 0;

    // step 3:
    vector<pair<int,int>> jobIntervals;

    for(int i=0; i<jobCnt; ++i)
    {
        jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }

    //step 4: print intervals
    cout << "Intervals:\n" << endl;

    int lateness = 0;

    for(int i=0; i<jobCnt; ++i)
    {
        auto pair = jobIntervals[i];

        lateness = max(lateness, pair.second-jobs[i].second);

        cout << "(" << pair.first << "," << pair.second << ") "
             << "Lateness: " << pair.second-jobs[i].second << std::endl;
    }

    cout << "\nmaximal lateness is " << lateness << endl;
```

```
    return 0;
}
```

And the output for this program is:

```
Intervals:

(0,2)   Lateness:-2
(2,5)   Lateness:-2
(5,8)   Lateness: 0
(8,9)   Lateness: 0
(9,12)  Lateness: 3
(12,17) Lateness: 6
(17,21) Lateness: 8
(21,23) Lateness: 6
(23,25) Lateness: 3
(25,26) Lateness: 1

maximal lateness is 8
```

The runtime of the algorithm is obviously Θ(n log n) because sorting is the dominating operation of this algorithm. Now we need to show that it is optimal. Clearly an optimal schedule has no **idle time**. the **earliest deadline first** schedule has also no idle time.

Lets assume the jobs are numbered so that `d1<=d2<=...<=dn`. We say a **inversion** of a schedule is a pair of jobs `i` and `j` so that `i<j` but j is scheduled before `i`. Due to its definition the **earliest deadline first** schedule has no inversions. Of course if a schedule has an inversion it has one with a pair of inverted jobs scheduled consecutively.

Proposition: Swapping two adjacent, inverted jobs reduces the number of inversions by **one** and **does not increase** the maximal lateness.

Proof: Let `L` be the lateness before the swap and `M` the lateness afterwards. Because exchanging two adjacent jobs does not move the other jobs from their position it is `Lk=Mk` for all `k != i,j`.

Clearly it is `Mi<=Li` since job `i` got scheduled earlier. if job `j` is late, so follows from the definition:

```
Mj = fi-dj    (definition)
  <= fi-di    (since i and j are exchanged)
  <= Li
```

That means the lateness after swap is less or equal than before. This concludes the proof.

Proposition: The **earliest deadline first** schedule `S` is optimal.

Proof:*(by contradiction)*

Lets assume `S*` is optimal schedule with the **fewest possible** number of inversions. we can assume that `S*` has no idle time. If `S*` has no inversions, then `S=S*` and we are done. If `S*` has an inversion, than it has an adjacent inversion. The last Proposition states that we can swap the adjacent inversion without increasing lateness but with decreasing the number of inversions. This contradicts the definition of `S*`.

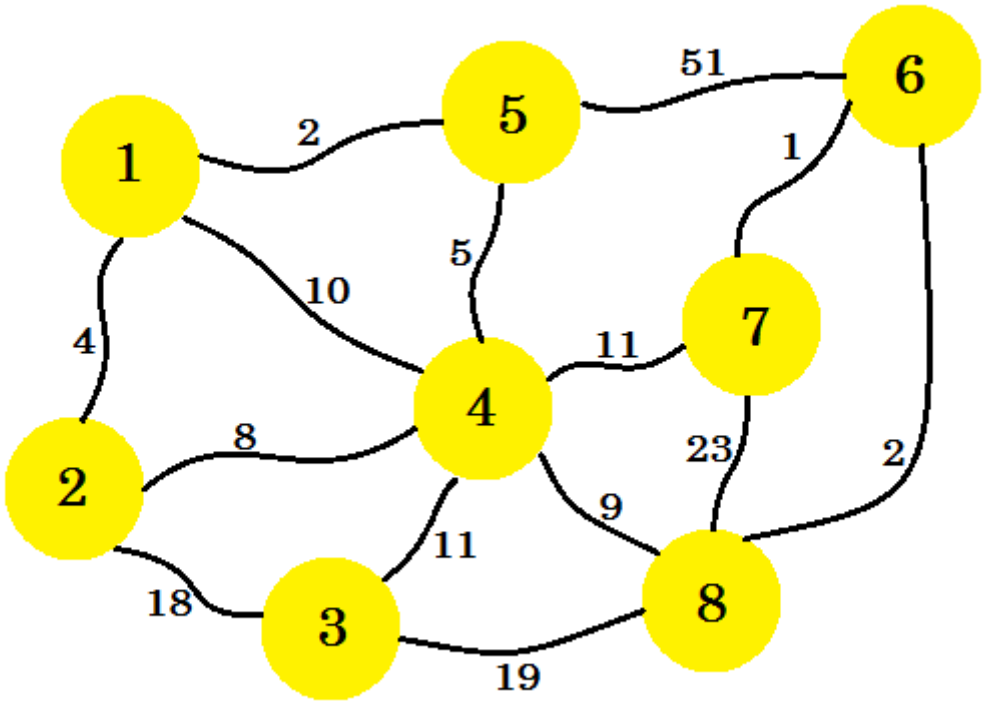
The minimizing lateness problem and its near related **minimum makespan** problem, where the question for a minimal schedule is asked have lots of applications in the real world. But usually you don't have only one machine but many and they handle the same task at different rates. These problems get NP-complete really fast.

Another interesting question arises if we don't look at the **offline** problem, where we have all tasks and data at hand but at the **online** variant, where tasks appear during execution.

Chapter 19: Prim's Algorithm

Section 19.1: Introduction To Prim's Algorithm

Let's say we have **8** houses. We want to setup telephone lines between these houses. The edge between the houses represent the cost of setting line between two houses.



Our task is to set up lines in such a way that all the houses are connected and the cost of setting up the whole connection is minimum. Now how do we find that out? We can use **Prim's Algorithm**.

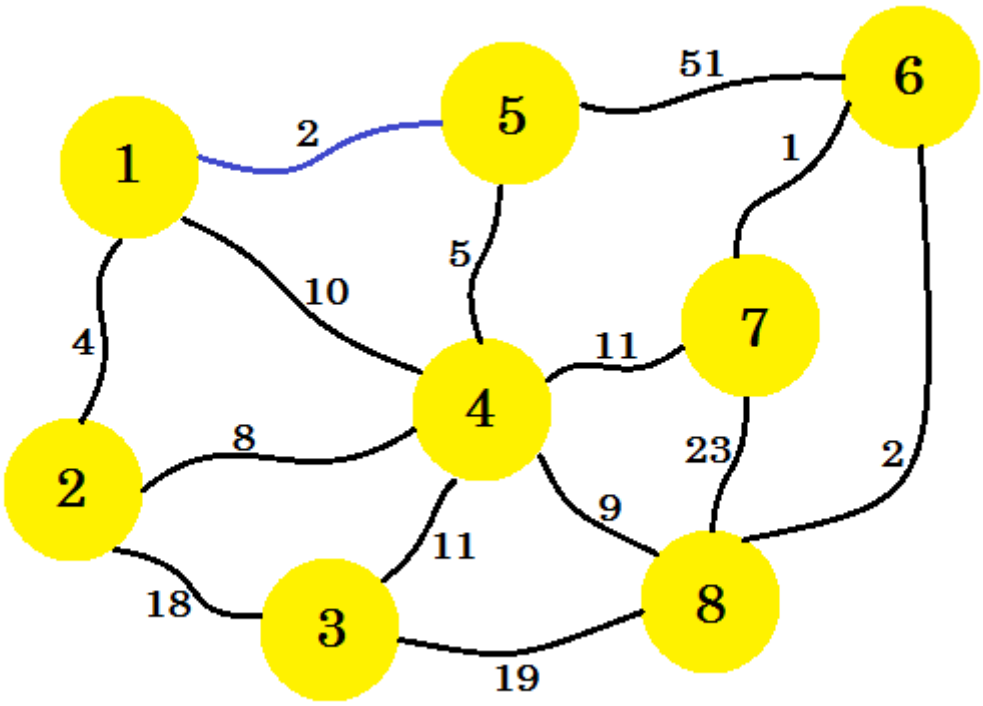
Prim's Algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every node, where the total weight of all the edges in the tree are minimized. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientist Robert Clay Prim in 1957 and Edsger Wybe Dijkstra in 1959. It is also known as **DJP algorithm**, **Jarnik's algorithm**, **Prim-Jarnik algorithm** or **Prim-Dijsktra algorithm**.

Now let's look at the technical terms first. If we create a graph, **S** using some nodes and edges of an undirected graph **G**, then **S** is called a **subgraph** of the graph **G**. Now **S** will be called a **Spanning Tree** if and only if:

- It contains all the nodes of **G**.
- It is a tree, that means there is no cycle and all the nodes are connected.
- There are **(n-1)** edges in the tree, where **n** is the number of nodes in **G**.

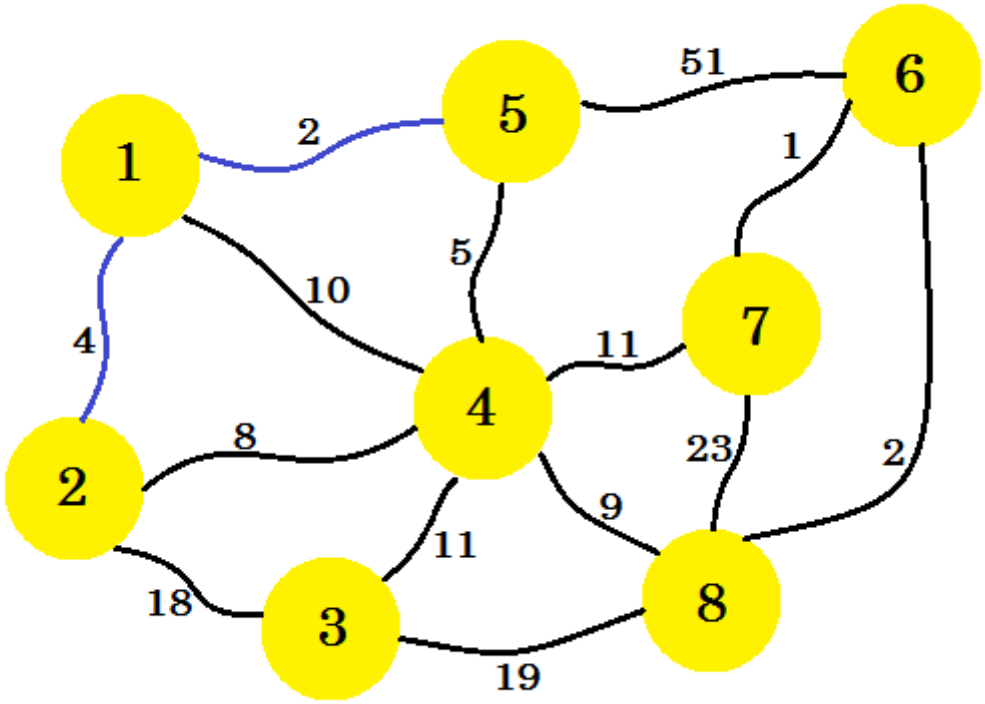
There can be many **Spanning Tree**'s of a graph. The **Minimum Spanning Tree** of a weighted undirected graph is a tree, such that sum of the weight of the edges is minimum. Now we'll use **Prim's algorithm** to find out the minimum spanning tree, that is how to set up the telephone lines in our example graph in such way that the cost of set up is minimum.

At first we'll select a **source** node. Let's say, **node-1** is our **source**. Now we'll add the edge from **node-1** that has the minimum cost to our subgraph. Here we mark the edges that are in the subgraph using the color **blue**. Here **1-5** is



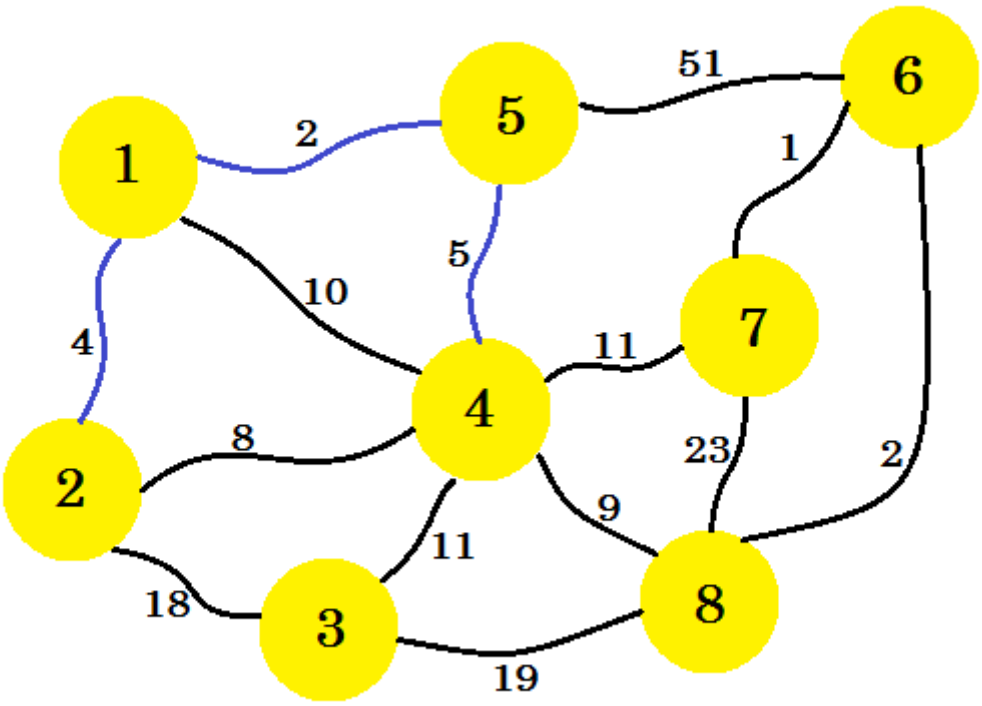
our desired edge.

Now we consider all the edges from **node-1** and **node-5** and take the minimum. Since **1-5** is already marked, we

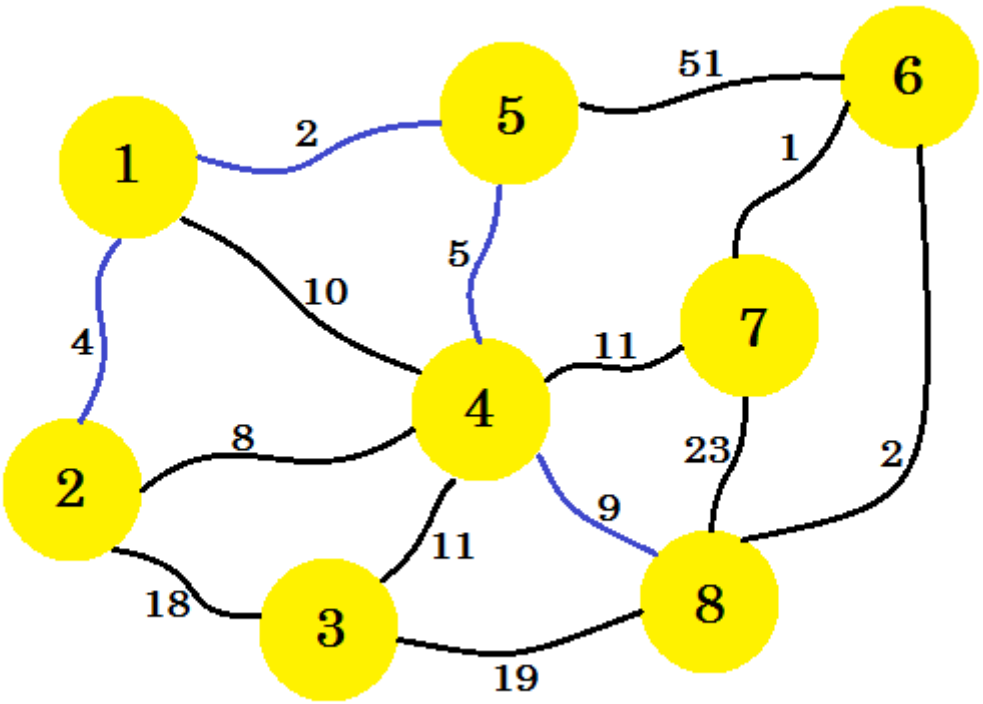


take **1-2**.

This time, we consider **node-1**, **node-2** and **node-5** and take the minimum edge which is **5-4**.

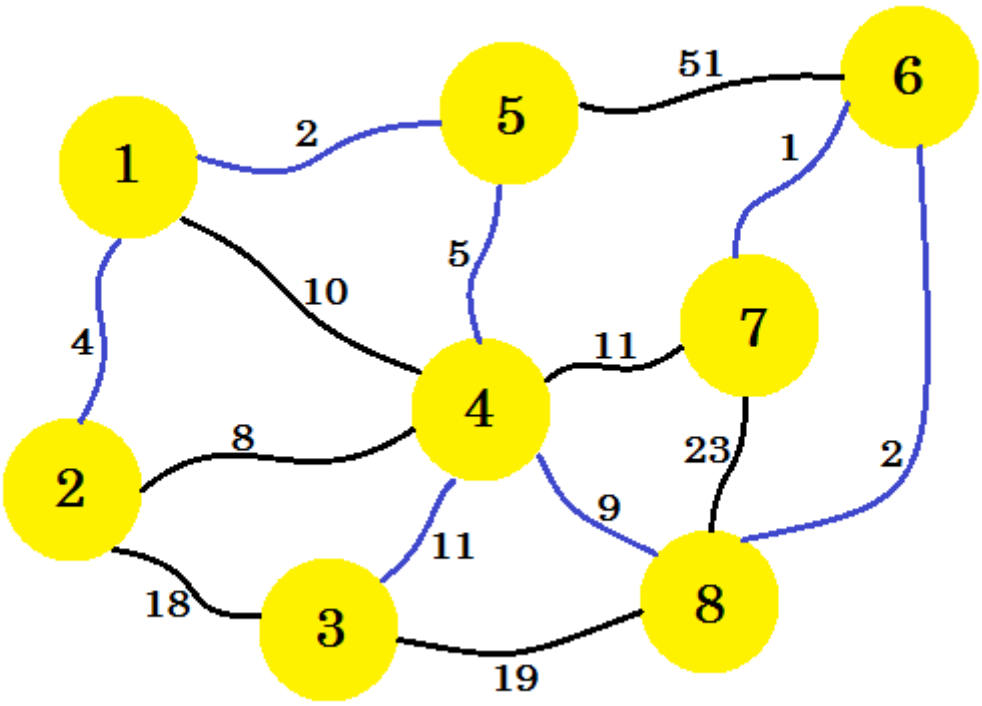


The next step is important. From **node-1**, **node-2**, **node-5** and **node-4**, the minimum edge is **2-4**. But if we select that one, it'll create a cycle in our subgraph. This is because **node-2** and **node-4** are already in our subgraph. So taking edge **2-4** doesn't benefit us. *We'll select the edges in such way that it adds a new node in our subgraph*. So we

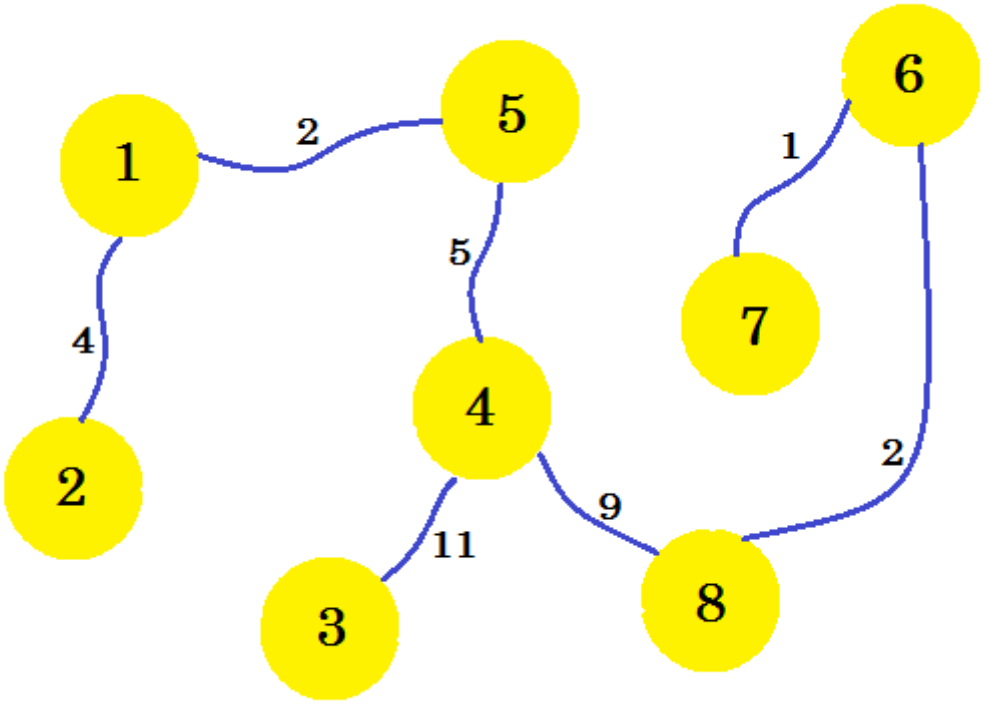


select edge **4-8**.

If we continue this way, we'll select edge **8-6**, **6-7** and **4-3**. Our subgraph will look like:



This is our desired subgraph, that'll give us the minimum spanning tree. If we remove the edges that we didn't



select, we'll get:

This is our **minimum spanning tree** (MST). So the cost of setting up the telephone connections is: **4** + **2** + **5** + **11** + **9** + **2** + **1** = **34**. And the set of houses and their connections are shown in the graph. There can be multiple **MST** of a graph. It depends on the **source** node we choose.

The pseudo-code of the algorithm is given below:

```
Procedure PrimsMST(Graph):      // here Graph is a non-empty connected weighted graph
Vnew[] = {x}                    // New subgraph Vnew with source node x
```

```
Enew[] = {}
while Vnew is not equal to V
    u -> a node from Vnew
    v -> a node that is not in Vnew such that edge u-v has the minimum cost
                                // if two nodes have same weight, pick any of them
    add v to Vnew
    add edge (u, v) to Enew
end while
Return Vnew and Enew
```

Complexity:

Time complexity of the above naive approach is **O(V²)**. It uses adjacency matrix. We can reduce the complexity using priority queue. When we add a new node to **Vnew**, we can add its adjacent edges in the priority queue. Then pop the minimum weighted edge from it. Then the complexity will be: **O(ElogE)**, where **E** is the number of edges. Again a Binary Heap can be constructed to reduce the complexity to **O(ElogV)**.

The pseudo-code using Priority Queue is given below:

```
Procedure MSTPrim(Graph, source):
for each u in V
    key[u] := inf
    parent[u] := NULL
end for
key[source] := 0
Q = Priority_Queue()
Q = V
while Q is not empty
    u -> Q.pop
    for each v adjacent to i
        if v belongs to Q and Edge(u,v) < key[v]    // here Edge(u, v) represents
                                                    // cost of edge(u, v)
            parent[v] := u
            key[v] := Edge(u, v)
        end if
    end for
end while
```

Here **key[]** stores the minimum cost of traversing **node-v**. **parent[]** is used to store the parent node. It is useful for traversing and printing the tree.

Below is a simple program in Java:

```
import java.util.*;

public class Graph
{
   private static int infinite = 9999999;
   int[][]  LinkCost;
   int      NNodes;
   Graph(int[][] mat)
   {
      int i, j;
      NNodes = mat.length;
      LinkCost = new int[NNodes][NNodes];
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
         {
```

```
            LinkCost[i][j] = mat[i][j];
            if ( LinkCost[i][j] == 0 )
               LinkCost[i][j] = infinite;
         }
      }
      for ( i=0; i < NNodes; i++)
      {
         for ( j=0; j < NNodes; j++)
            if ( LinkCost[i][j] < infinite )
               System.out.print( " " + LinkCost[i][j] + " " );
            else
               System.out.print(" * " );
         System.out.println();
      }
   }
   public int unReached(boolean[] r)
   {
      boolean done = true;
      for ( int i = 0; i < r.length; i++ )
         if ( r[i] == false )
            return i;
      return -1;
   }
   public void Prim( )
   {
      int i, j, k, x, y;
      boolean[] Reached = new boolean[NNodes];
      int[] predNode = new int[NNodes];
      Reached[0] = true;
      for ( k = 1; k < NNodes; k++ )
      {
         Reached[k] = false;
      }
      predNode[0] = 0;
      printReachSet( Reached );
      for (k = 1; k < NNodes; k++)
      {
         x = y = 0;
         for ( i = 0; i < NNodes; i++ )
            for ( j = 0; j < NNodes; j++ )
            {
                if ( Reached[i] && !Reached[j] &&
                     LinkCost[i][j] < LinkCost[x][y] )
                {
                   x = i;
                   y = j;
                }
            }
         System.out.println("Min cost edge: (" +
                               + x + "," +
                               + y + ")" +
                               "cost = " + LinkCost[x][y]);
         predNode[y] = x;
         Reached[y] = true;
         printReachSet( Reached );
         System.out.println();
      }
      int[] a= predNode;
   for ( i = 0; i < NNodes; i++ )
          System.out.println( a[i] + " --> " + i );
   }
   void printReachSet(boolean[] Reached )
```

```
   {
      System.out.print("ReachSet = ");
      for (int i = 0; i < Reached.length; i++ )
         if ( Reached[i] )
           System.out.print( i + " ");
      //System.out.println();
   }
 public static void main(String[] args)
   {
      int[][] conn = {{0,3,0,2,0,0,0,0,4},  // 0
                      {3,0,0,0,0,0,0,4,0},  // 1
                      {0,0,0,6,0,1,0,2,0},  // 2
                      {2,0,6,0,1,0,0,0,0},  // 3
                      {0,0,0,1,0,0,0,0,8},  // 4
                      {0,0,1,0,0,0,8,0,0},  // 5
                      {0,0,0,0,0,8,0,0,0},  // 6
                      {0,4,2,0,0,0,0,0,0},  // 7
                      {4,0,0,0,8,0,0,0,0}   // 8
                     };
      Graph G = new Graph(conn);
      G.Prim();
   }
}
```

Compile the above code using `javac Graph.java`

Output:

```
$ java Graph
 *  3  *  2  *  *  *  *  4
 3  *  *  *  *  *  *  4  *
 *  *  *  6  *  1  *  2  *
 2  *  6  *  1  *  *  *  *
 *  *  *  1  *  *  *  *  8
 *  *  1  *  *  *  8  *  *
 *  *  *  *  *  8  *  *  *
 *  4  2  *  *  *  *  *  *
 4  *  *  *  8  *  *  *  *
ReachSet = 0 Min cost edge: (0,3)cost = 2
ReachSet = 0 3
Min cost edge: (3,4)cost = 1
ReachSet = 0 3 4
Min cost edge: (0,1)cost = 3
ReachSet = 0 1 3 4
Min cost edge: (0,8)cost = 4
ReachSet = 0 1 3 4 8
Min cost edge: (1,7)cost = 4
ReachSet = 0 1 3 4 7 8
Min cost edge: (7,2)cost = 2
ReachSet = 0 1 2 3 4 7 8
Min cost edge: (2,5)cost = 1
ReachSet = 0 1 2 3 4 5 7 8
Min cost edge: (5,6)cost = 8
ReachSet = 0 1 2 3 4 5 6 7 8
0 --> 0
0 --> 1
7 --> 2
0 --> 3
3 --> 4
2 --> 5
5 --> 6
```

```
1 --> 7
0 --> 8
```

Chapter 20: Bellman–Ford Algorithm

Section 20.1: Single Source Shortest Path Algorithm (Given there is a negative cycle in a graph)

Before reading this example, it is required to have a brief idea on edge-relaxation. You can learn it from here

Bellman-Ford Algorithm is computes the shortest paths from a single source vertex to all of the other vertices in a weighted digraph. Even though it is slower than Dijkstra's Algorithm, it works in the cases when the weight of the edge is negative and it also finds negative weight cycle in the graph. The problem with Dijkstra's Algorithm is, if there's a negative cycle, you keep going through the cycle again and again and keep reducing the distance between two vertices.

The idea of this algorithm is to go through all the edges of this graph one-by-one in some random order. It can be any random order. But you must ensure, if **u-v** (where **u** and **v** are two vertices in a graph) is one of your orders, then there must be an edge from **u** to **v**. Usually it is taken directly from the order of the input given. Again, any random order will work.

After selecting the order, we will *relax* the edges according to the relaxation formula. For a given edge **u-v** going from **u** to **v** the relaxation formula is:

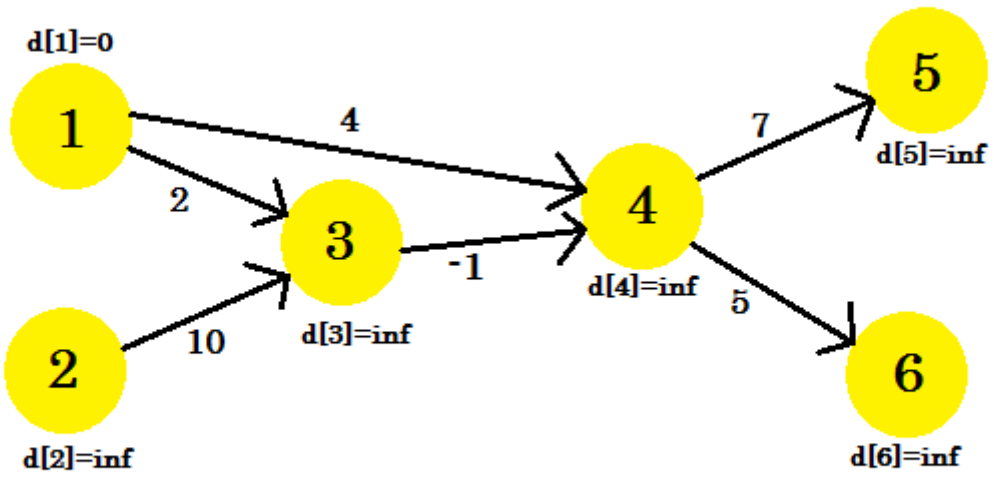
```
if distance[u] + cost[u][v] < d[v]
    d[v] = d[u] + cost[u][v]
```

That is, if the distance from **source** to any vertex **u** + the weight of the **edge u-v** is less than the distance from **source** to another vertex **v**, we update the distance from **source** to **v**. We need to *relax* the edges at most **(V-1)** times where **V** is the number of edges in the graph. Why **(V-1)** you ask? We'll explain it in another example. Also we are going to keep track of the parent vertex of any vertex, that is when we relax an edge, we will set:

```
parent[v] = u
```

It means we've found another shorter path to reach **v** via **u**. We will need this later to print the shortest path from **source** to the destined vertex.

Let's look at an example. We have a graph:



We have selected **1** as the **source** vertex. We want to find out the shortest path from the **source** to all other vertices.

At first, **d[1] = 0** because it is the source. And rest are *infinity*, because we don't know their distance yet.

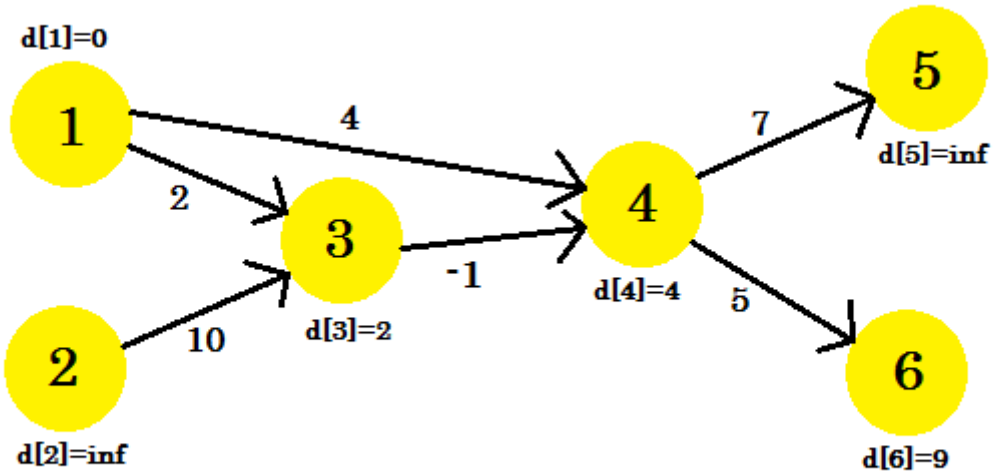
We will relax the edges in this sequence:

Serial	1	2	3	4	5	6
Edge	4->5	3->4	1->3	1->4	4->6	2->3

You can take any sequence you want. If we *relax* the edges once, what do we get? We get the distance from **source** to all other vertices of the path that uses at most 1 edge. Now let's relax the edges and update the values of **d[]**. We get:

1. **d[4]** + **cost[4][5]** = **infinity** + **7** = **infinity**. We can't update this one.
2. **d[2]** + **cost[3][4]** = **infinity**. We can't update this one.
3. **d[1]** + **cost[1][3]** = **0** + **2** = **2** < **d[2]**. So **d[3]** = **2**. Also **parent[1] = 1**.
4. **d[1]** + **cost[1][4]** = **4**. So **d[4] = 4** < **d[4]**. **parent[4]** = **1**.
5. **d[4]** + **cost[4][6]** = **9**. **d[6]** = **9** < **d[6]**. **parent[6]** = **4**.
6. **d[2]** + **cost[2][3]** = **infinity**. We can't update this one.

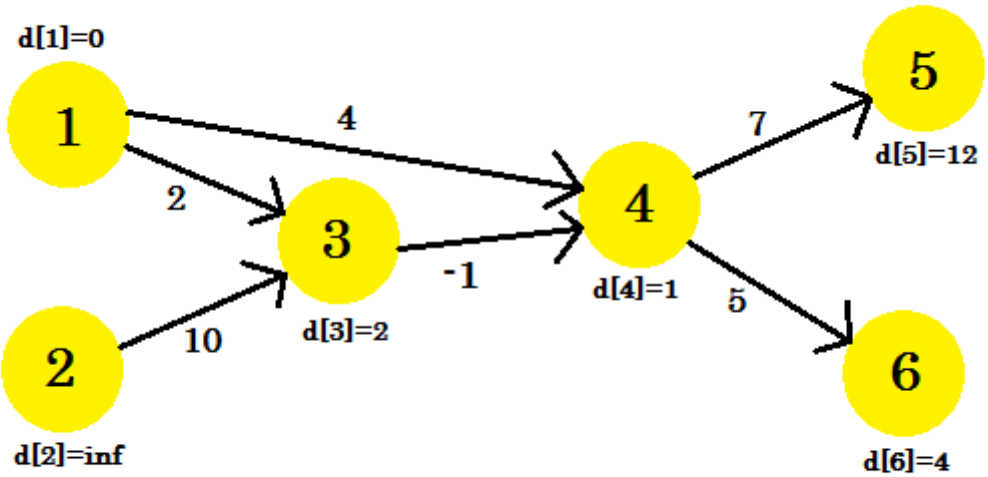
We couldn't update some vertices, because the `d[u] + cost[u][v] < d[v]` condition didn't match. As we have said before, we found the paths from **source** to other nodes using maximum 1 edge.



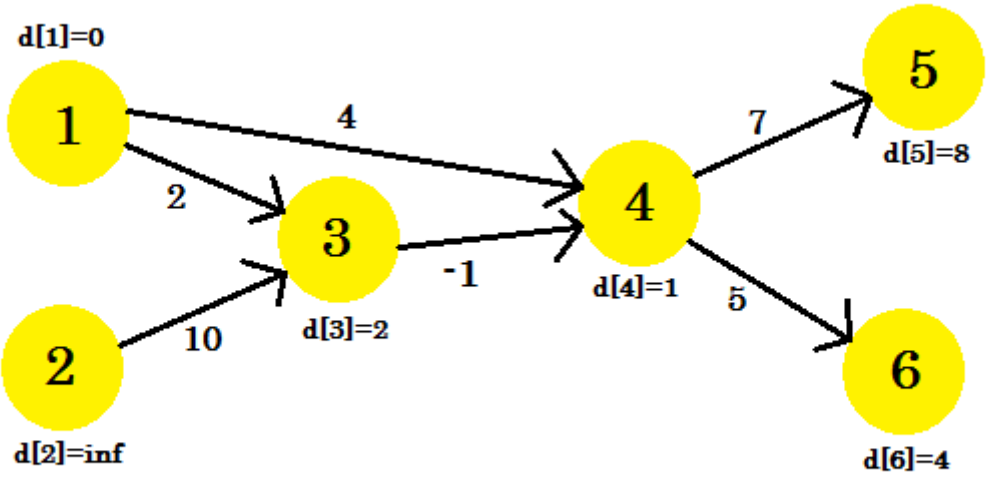
Our second iteration will provide us with the path using 2 nodes. We get:

1. **d[4]** + **cost[4][5]** = **12** < **d[5]**. **d[5]** = **12**. **parent[5]** = **4**.
2. **d[3]** + **cost[3][4]** = **1** < **d[4]**. **d[4]** = **1**. **parent[4]** = **3**.
3. **d[3]** remains unchanged.
4. **d[4]** remains unchanged.
5. **d[4]** + **cost[4][6]** = **6** < **d[6]**. **d[6]** = **6**. **parent[6]** = **4**.
6. **d[3]** remains unchanged.

Our graph will look like:



Our 3rd iteration will only update **vertex 5**, where **d[5]** will be **8**. Our graph will look like:



After this no matter how many iterations we do, we'll have the same distances. So we will keep a flag that checks if any update takes place or not. If it doesn't, we'll simply break the loop. Our pseudo-code will be:

```
Procedure Bellman-Ford(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
```

```
        end if
    end for
    if flag == false
        break
end for
Return d
```

To keep track of negative cycle, we can modify our code using the procedure described here. Our completed pseudo-code will be:

```
Procedure Bellman-Ford-With-Negative-Cycle-Detection(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return d
```

Printing Path:

To print the shortest path to a vertex, we'll iterate back to its parent until we find **NULL** and then print the vertices. The pseudo-code will be:

```
Procedure PathPrinting(u)
v := parent[u]
if v == NULL
    return
PathPrinting(v)
print -> u
```

Complexity:

Since we need to relax the edges maximum **(V-1)** times, the time complexity of this algorithm will be equal to **O(V * E)** where **E** denotes the number of edges, if we use `adjacency list` to represent the graph. However, if `adjacency matrix` is used to represent the graph, time complexity will be **O(V^3)**. Reason is we can iterate through all edges in **O(E)** time when `adjacency list` is used, but it takes **O(V^2)** time when `adjacency matrix` is used.

Section 20.2: Detecting Negative Cycle in a Graph

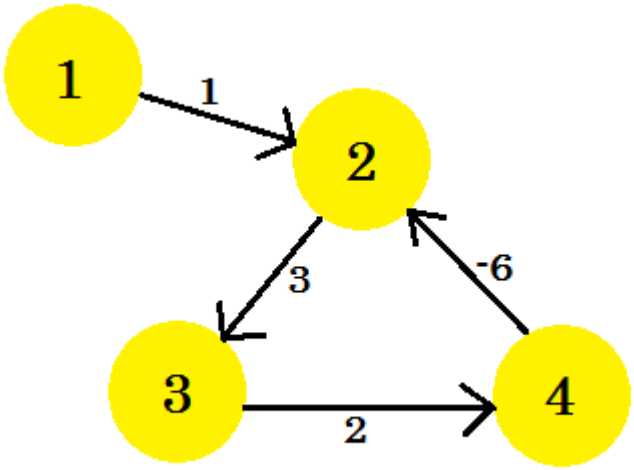
To understand this example, it is recommended to have a brief idea about Bellman-Ford algorithm which can be found

here

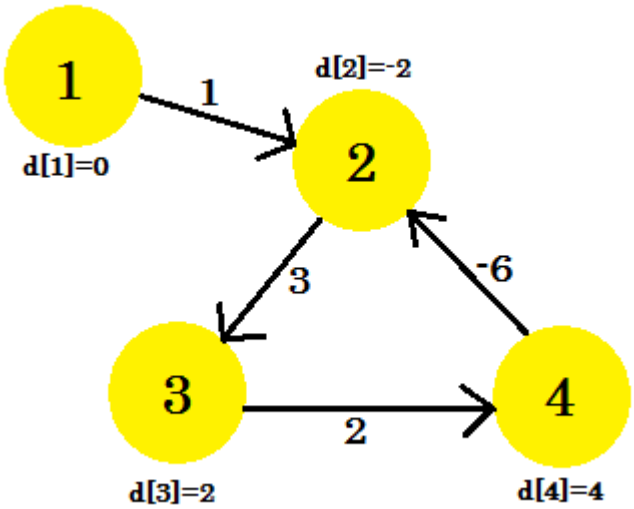
Using Bellman-Ford algorithm, we can detect if there is a negative cycle in our graph. We know that, to find out the shortest path, we need to *relax* all the edges of the graph **(V-1)** times, where **V** is the number of vertices in a graph. We have already seen that in this example, after **(V-1)** iterations, we can't update **d[]**, no matter how many iterations we do. Or can we?

If there is a negative cycle in a graph, even after **(V-1)** iterations, we can update **d[]**. This happens because for every iteration, traversing through the negative cycle always decreases the cost of the shortest path. This is why Bellman-Ford algorithm limits the number of iterations to **(V-1)**. If we used Dijkstra's Algorithm here, we'd be stuck in an endless loop. However, let's concentrate on finding negative cycle.

Let's assume, we have a graph:



Let's pick **vertex 1** as the **source**. After applying Bellman-Ford's single source shortest path algorithm to the graph, we'll find out the distances from the **source** to all the other vertices.



This is how the graph looks like after **(V-1)** = **3** iterations. It should be the result since there are **4** edges, we need at most **3** iterations to find out the shortest path. So either this is the answer, or there is a negative weight cycle in the graph. To find that, after **(V-1)** iterations, we do one more final iteration and if the distance continues to decrease, it means that there is definitely a negative weight cycle in the graph.

For this example: if we check **2-3**, **d[2]** + **cost[2][3]** will give us **1** which is less than **d[3]**. So we can conclude that there is a negative cycle in our graph.

So how do we find out the negative cycle? We do a bit modification to Bellman-Ford procedure:

```
Procedure NegativeCycleDetector(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return "No Negative Cycle"
```

This is how we find out if there is a negative cycle in a graph. We can also modify Bellman-Ford Algorithm to keep track of negative cycles.

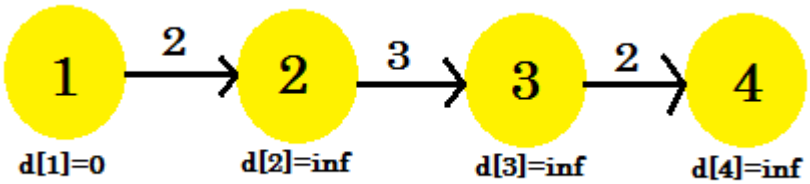
Section 20.3: Why do we need to relax all the edges at most (V-1) times

To understand this example, it is recommended to have a brief idea on Bellman-Ford single source shortest path algorithm which can be found here

In Bellman-Ford algorithm, to find out the shortest path, we need to *relax* all the edges of the graph. This process is repeated at most **(V-1)** times, where **V** is the number of vertices in the graph.

The number of iterations needed to find out the shortest path from **source** to all other vertices depends on the order that we select to *relax* the edges.

Let's take a look at an example:



Here, the **source** vertex is 1. We will find out the shortest distance between the **source** and all the other vertices. We can clearly see that, to reach **vertex 4**, in the worst case, it'll take **(V-1)** edges. Now depending on the order in which the edges are discovered, it might take **(V-1)** times to discover **vertex 4**. Didn't get it? Let's use Bellman-Ford

algorithm to find out the shortest path here:

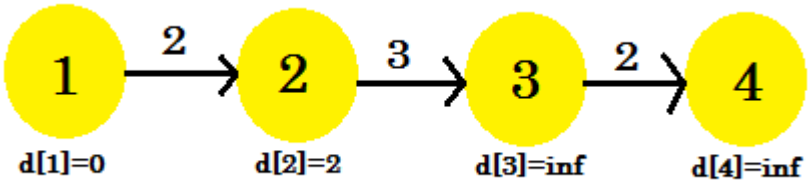
We're going to use this sequence:

```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  3->4  |  2->3  |  1->2  |
+--------+--------+--------+--------+
```

For our first iteration:

1. **d[3]** + **cost[3][4]** = **infinity**. It won't change anything.
2. **d[2]** + **cost[2][3]** = **infinity**. It won't change anything.
3. **d[1]** + **cost[1][2]** = **2** < **d[2]**. **d[2]** = **2**. **parent[2]** = **1**.

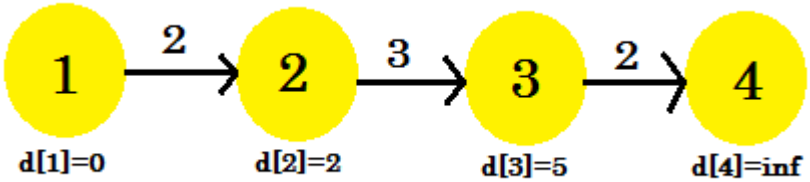
We can see that our *relaxation* process only changed **d[2]**. Our graph will look like:



Second iteration:

1. **d[3]** + **cost[3][4]** = **infinity**. It won't change anything.
2. **d[2]** + **cost[2][3]** = **5** < **d[3]**. **d[3]** = **5**. **parent[3]** = **2.**
3. It won't be changed.

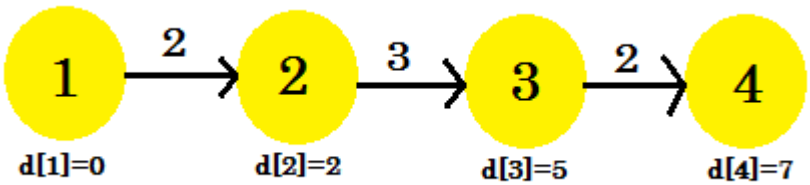
This time the *relaxation* process changed **d[3]**. Our graph will look like:



Third iteration:

1. **d[3]** + **cost[3][4]** = **7** < **d[4]**. **d[4]** = **7**. **parent[4]** = **3**.
2. It won't be changed.
3. It won't be changed.

Our third iteration finally found out the shortest path to **4** from **1**. Our graph will look like:



So, it took **3** iterations to find out the shortest path. After this one, no matter how many times we *relax* the edges,

the values in **d[]** will remain the same. Now, if we considered another sequence:

```
+--------+--------+--------+--------+
| Serial |    1   |    2   |    3   |
+--------+--------+--------+--------+
|  Edge  |  1->2  |  2->3  |  3->4  |
+--------+--------+--------+--------+
```

We'd get:

1. **d[1]** + **cost[1][2]** = **2** < **d[2]**. **d[2]** = **2**.
2. **d[2]** + **cost[2][3]** = **5** < **d[3]**. **d[3]** = **5**.
3. **d[3]** + **cost[3][4]** = **7** < **d[4]**. **d[4]** = **5**.

Our very first iteration has found the shortest path from **source** to all the other nodes. Another sequence **1->2**, **3->4**, **2->3** is possible, which will give us shortest path after **2** iterations. We can come to the decision that, no matter how we arrange the sequence, it won't take more than **3** iterations to find out shortest path from the **source** in this example.

We can conclude that, for the best case, it'll take **1** iteration to find out the shortest path from **source**. For the worst case, it'll take **(V-1)** iterations, which is why we repeat the process of *relaxation* **(V-1)** times.

Chapter 21: Line Algorithm

Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints.

Section 21.1: Bresenham Line Drawing Algorithm

Background Theory: Bresenham's Line Drawing Algorithm is an efficient and accurate raster line generating algorithm developed by Bresenham. It involves only integer calculation so it is accurate and fast. It can also be extended to display circles another curves.

In Bresenham line drawing algorithm:

For Slope |m|<1:
Either value of x is increased
OR both x and y is increased using decision parameter.

For Slope |m|>1:
Either value of y is increased
OR both x and y is increased using decision parameter.

Algorithm for slope |m|<1:

1. Input two end points (x1,y1) and (x2,y2) of the line.
2. Plot the first point (x1,y1).
3. Calculate
   Delx =| x2 – x1 |
   Dely = | y2 – y1 |
4. Obtain the initial decision parameter as
   P = 2 * dely – delx
5. For I = 0 to delx in step of 1

   If p < 0 then
   X1 = x1 + 1
   Pot(x1,y1)
   P = p+ 2*dely*

   Else
   X1 = x1 + 1
   Y1 = y1 + 1
   Plot(x1,y1)
   P = p + *2*dely – 2 * delx

   End if

   End for
6. END

Source Code:

```c
/* A C program to implement Bresenham line drawing algorithm for |m|<1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>

int main()
{
 int gdriver=DETECT,gmode;
 int x1,y1,x2,y2,delx,dely,p,i;
 initgraph(&gdriver,&gmode,"c:\\TC\\BGI");

printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);

putpixel(x1,y1,RED);

delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*dely)-delx;
for(i=0;i<delx;i++){
if(p<0)
{
 x1=x1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely);
}
else
{
 x1=x1+1;
 y1=y1+1;
 putpixel(x1,y1,RED);
 p=p+(2*dely)-(2*delx);
}
}
 getch();
 closegraph();
 return 0;
}
```

Algorithm for slope |m|>1:

1. Input two end points (x1,y1) and (x2,y2) of the line.
2. Plot the first point (x1,y1).
3. Calculate
   Delx =| x2 – x1 |
   Dely = | y2 – y1 |
4. Obtain the initial decision parameter as
   P = 2 * delx – dely
5. For I = 0 to dely in step of 1

   If p < 0 then
   y1 = y1 + 1
   Pot(x1,y1)

P = p+ 2*delx*

Else
X1 = x1 + 1
Y1 = y1 + 1
Plot(x1,y1)
*P = p + 2*delx – 2 * dely

End if

End for

6. END

Source Code:

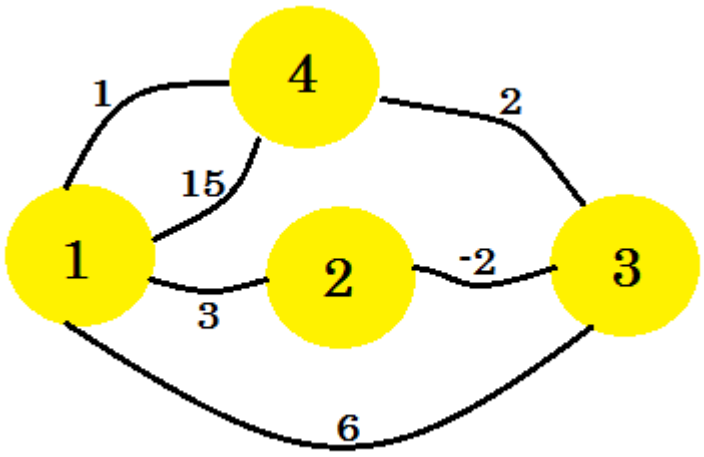
```c
/* A C program to implement Bresenham line drawing algorithm for |m|>1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
int main()
{
int gdriver=DETECT,gmode;
int x1,y1,x2,y2,delx,dely,p,i;
initgraph(&gdriver,&gmode,"c:\\TC\\BGI");
printf("Enter the intial points: ");
scanf("%d",&x1);
scanf("%d",&y1);
printf("Enter the end points: ");
scanf("%d",&x2);
scanf("%d",&y2);
putpixel(x1,y1,RED);
delx=fabs(x2-x1);
dely=fabs(y2-y1);
p=(2*delx)-dely;
for(i=0;i<delx;i++){
if(p<0)
{
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx);
}
else
{
x1=x1+1;
y1=y1+1;
putpixel(x1,y1,RED);
p=p+(2*delx)-(2*dely);
}
}
getch();
closegraph();
 return 0;
}
```

Chapter 22: Floyd-Warshall Algorithm

Section 22.1: All Pair Shortest Path Algorithm

Floyd-Warshall's algorithm is for finding shortest paths in a weighted graph with positive or negative edge weights. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pair of vertices. With a little variation, it can print the shortest path and can detect negative cycles in a graph. Floyd-Warshall is a Dynamic-Programming algorithm.

Let's look at an example. We're going to apply Floyd-Warshall's algorithm on this graph:



First thing we do is, we take two 2D matrices. These are adjacency matrices. The size of the matrices is going to be the total number of vertices. For our graph, we will take **4 * 4** matrices. The **Distance Matrix** is going to store the minimum distance found so far between two vertices. At first, for the edges, if there is an edge between **u-v** and the distance/weight is **w**, we'll store: `distance[u][v] = w`. For all the edges that doesn't exist, we're gonna put *infinity*. The **Path Matrix** is for regenerating minimum distance path between two vertices. So initially, if there is a path between **u** and **v**, we're going to put `path[u][v] = u`. This means the best way to come to **vertex-v** from **vertex-u** is to use the edge that connects **v** with **u**. If there is no path between two vertices, we're going to put **N** there indicating there is no path available now. The two tables for our graph will look like:

```
+-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  6  |  15 |
+-----+-----+-----+-----+-----+
|  2  | inf |  0  | -2  | inf |
+-----+-----+-----+-----+-----+
|  3  | inf | inf |  0  |  2  |
+-----+-----+-----+-----+-----+
|  4  |  1  | inf | inf |  0  |
+-----+-----+-----+-----+-----+
distance
```

```
+-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+
|  1  |  N  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+
|  2  |  N  |  N  |  2  |  N  |
+-----+-----+-----+-----+-----+
|  3  |  N  |  N  |  N  |  3  |
+-----+-----+-----+-----+-----+
|  4  |  4  |  N  |  N  |  N  |
+-----+-----+-----+-----+-----+
  path
```

Since there is no loop, the diagonals are set **N**. And the distance from the vertex itself is **0**.

To apply Floyd-Warshall algorithm, we're going to select a middle vertex **k**. Then for each vertex **i**, we're going to check if we can go from **i** to **k** and then **k** to **j**, where **j** is another vertex and minimize the cost of going from **i** to **j**. If the current **distance[i][j]** is greater than **distance[i][k]** + **distance[k][j]**, we're going to put **distance[i][j]** equals to the summation of those two distances. And the **path[i][j]** will be set to **path[k][j]**, as it is better to go from **i** to **k**,

and then **k** to **j**. All the vertices will be selected as **k**. We'll have 3 nested loops: for **k** going from 1 to 4, **i** going from 1 to 4 and **j** going from 1 to 4. We're going check:

```
if distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if
```

So what we're basically checking is, *for every pair of vertices, do we get a shorter distance by going through another vertex?* The total number of operations for our graph will be **4 * 4 * 4** = **64**. That means we're going to do this check **64** times. Let's look at a few of them:

When **k** = **1**, **i** = **2** and **j** = **3**, **distance[i][j]** is **-2**, which is not greater than **distance[i][k]** + **distance[k][j]** = **-2** + **0** = **-2**. So it will remain unchanged. Again, when **k** = **1**, **i** = **4** and **j** = **2**, **distance[i][j]** = **infinity**, which is greater than **distance[i][k]** + **distance[k][j]** = **1** + **3** = **4**. So we put **distance[i][j]** = **4**, and we put **path[i][j]** = **path[k][j]** = **1**. What this means is, to go from **vertex-4** to **vertex-2**, the path **4->1->2** is shorter than the existing path. This is how we populate both matrices. The calculation for each step is shown here. After making necessary changes, our matrices will look like:

```
+-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  1  |  3  |
+-----+-----+-----+-----+-----+
|  2  |  1  |  0  | -2  |  0  |
+-----+-----+-----+-----+-----+
|  3  |  3  |  6  |  0  |  2  |
+-----+-----+-----+-----+-----+
|  4  |  1  |  4  |  2  |  0  |
+-----+-----+-----+-----+-----+
distance
```

```
+-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+
|  1  |  N  |  1  |  2  |  3  |
+-----+-----+-----+-----+-----+
|  2  |  4  |  N  |  2  |  3  |
+-----+-----+-----+-----+-----+
|  3  |  4  |  1  |  N  |  3  |
+-----+-----+-----+-----+-----+
|  4  |  4  |  1  |  2  |  N  |
+-----+-----+-----+-----+-----+
  path
```

This is our shortest distance matrix. For example, the shortest distance from **1** to **4** is **3** and the shortest distance between **4** to **3** is **2**. Our pseudo-code will be:

```
Procedure Floyd-Warshall(Graph):
for k from 1 to V     // V denotes the number of vertex
    for i from 1 to V
       for j from 1 to V
           if distance[i][j] > distance[i][k] + distance[k][j]
               distance[i][j] := distance[i][k] + distance[k][j]
               path[i][j] := path[k][j]
           end if
       end for
    end for
end for
```

Printing the path:

To print the path, we'll check the **Path** matrix. To print the path from **u** to **v**, we'll start from **path[u][v]**. We'll set keep changing **v** = **path[u][v]** until we find **path[u][v]** = **u** and push every values of **path[u][v]** in a stack. After finding **u**, we'll print **u** and start popping items from the stack and print them. This works because the **path** matrix stores the value of the vertex which shares the shortest path to **v** from any other node. The pseudo-code will be:

```
Procedure PrintPath(source, destination):
```

```
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
    S.push(Path[source][destination])
    destination := Path[source][destination]
end while
print -> source
while S is not empty
    print -> S.pop
end while
```

Finding Negative Edge Cycle:

To find out if there is a negative edge cycle, we'll need to check the main diagonal of **distance** matrix. If any value on the diagonal is negative, that means there is a negative cycle in the graph.

Complexity:

The complexity of Floyd-Warshall algorithm is $\mathbf{O(V^3)}$ and the space complexity is: $\mathbf{O(V^2)}$.

Chapter 23: Catalan Number Algorithm

Section 23.1: Catalan Number Algorithm Basic Information

Catalan numbers algorithm is Dynamic Programming algorithm.

In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. The Catalan numbers on nonnegative integers n are a set of numbers that arise in tree enumeration problems of the type, 'In how many ways can a regular n-gon be divided into n-2 triangles if different orientations are counted separately?'

Application of Catalan Number Algorithm:

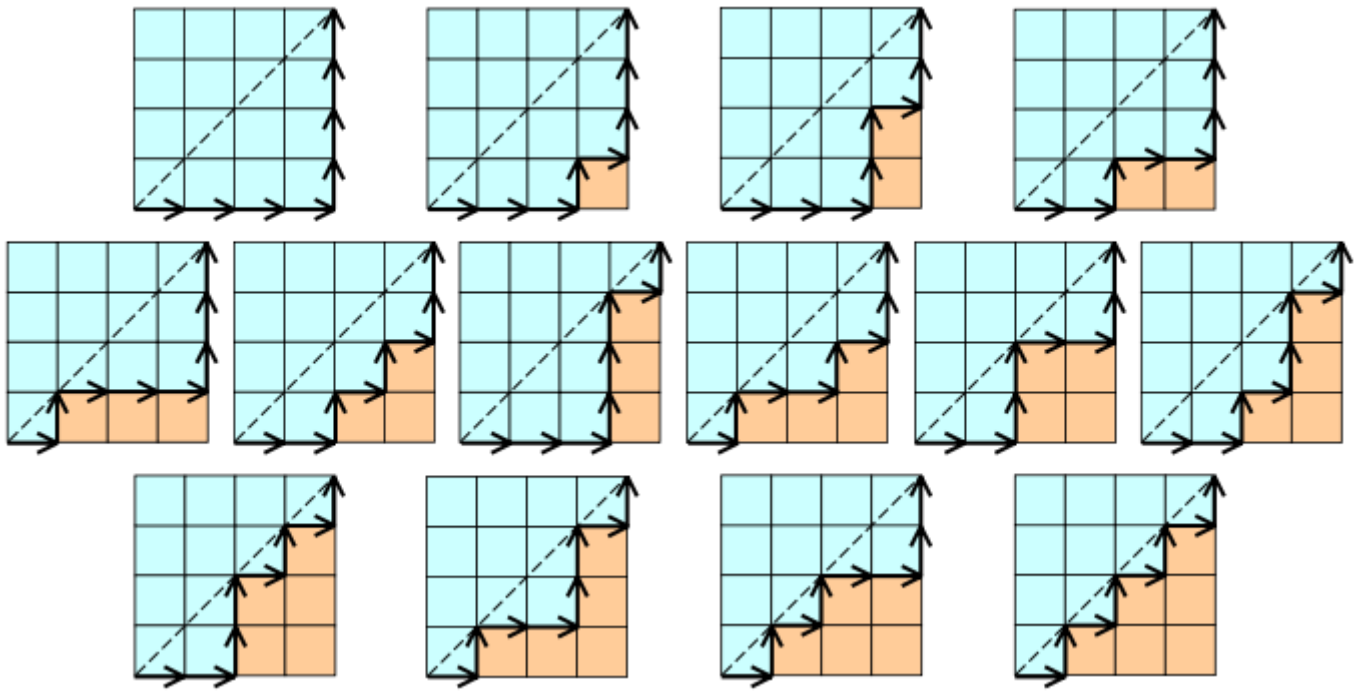
1. The number of ways to stack coins on a bottom row that consists of n consecutive coins in a plane, such that no coins are allowed to be put on the two sides of the bottom coins and every additional coin must be above two other coins, is the nth Catalan number.
2. The number of ways to group a string of n pairs of parentheses, such that each open parenthesis has a matching closed parenthesis, is the nth Catalan number.
3. The number of ways to cut an n+2-sided convex polygon in a plane into triangles by connecting vertices with straight, non-intersecting lines is the nth Catalan number. This is the application in which Euler was interested.

Using zero-based numbering, the *n*th Catalan number is given directly in terms of binomial coefficients by the following equation.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

Example of Catalan Number:

Here value of n = 4.(Best Example - From Wikipedia)



Auxiliary Space: O(n)

Time Complexity: O(n^2)

Chapter 24: Multithreaded Algorithms

Examples for some multithreaded algorithms.

Section 24.1: Square matrix multiplication multithread

```
multiply-square-matrix-parallel(A, B)
    n = A.lines
    C = Matrix(n,n) //create a new matrix n*n
    parallel for i = 1 to n
        parallel for j = 1 to n
            C[i][j] = 0
            pour k = 1 to n
                C[i][j] = C[i][j] + A[i][k]*B[k][j]
    return C
```

Section 24.2: Multiplication matrix vector multithread

```
matrix-vector(A,x)
    n = A.lines
    y = Vector(n) //create a new vector of length n
    parallel for i = 1 to n
        y[i] = 0
    parallel for i = 1 to n
        for j = 1 to n
            y[i] = y[i] + A[i][j]*x[j]
    return y
```

Section 24.3: merge-sort multithread

A is an array and *p* and *q* indexes of the array such as you gonna sort the sub-array *A[p..r]*. *B* is a sub-array which will be populated by the sort.

A call to *p-merge-sort(A,p,r,B,s)* sorts elements from *A[p..r]* and put them in *B[s..s+r-p]*.

```
p-merge-sort(A,p,r,B,s)
    n = r-p+1
    if n==1
        B[s] = A[p]
    else
        T = new Array(n) //create a new array T of size n
        q = floor((p+r)/2))
        q_prime = q-p+1
        spawn p-merge-sort(A,p,q,T,1)
        p-merge-sort(A,q+1,r,T,q_prime+1)
        sync
        p-merge(T,1,q_prime,q_prime+1,n,B,s)
```

Here is the auxiliary function that performs the merge in parallel.
p-merge assumes that the two sub-arrays to merge are in the same array but doesn't assume they are adjacent in the array. That's why we need *p1,r1,p2,r2*.

```
p-merge(T,p1,r1,p2,r2,A,p3)
    n1 = r1-p1+1
    n2 = r2-p2+1
    if n1<n2     //check if n1>=n2
```

```
        permute p1 and p2
        permute r1 and r2
        permute n1 and n2
    if n1==0     //both empty?
        return
    else
        q1 = floor((p1+r1)/2)
        q2 = dichotomic-search(T[q1],T,p2,r2)
        q3 = p3 + (q1-p1) + (q2-p2)
        A[q3] = T[q1]
        spawn p-merge(T,p1,q1-1,p2,q2-1,A,p3)
        p-merge(T,q1+1,r1,q2,r2,A,q3+1)
        sync
```

And here is the auxiliary function dichotomic-search.

x is the key to look for in the sub-array T[p..r].

```
dichotomic-search(x,T,p,r)
    inf = p
    sup = max(p,r+1)
    while inf<sup
        half = floor((inf+sup)/2)
        if x<=T[half]
            sup = half
        else
            inf = half+1
    return sup
```

Chapter 25: Knuth Morris Pratt (KMP) Algorithm

The KMP is a pattern matching algorithm which searches for occurrences of a "word" **W** within a main "text string" **S** by employing the observation that when a mismatch occurs, we have the sufficient information to determine where the next match could begin.We take advantage of this information to avoid matching the characters that we know will anyway match.The worst case complexity for searching a pattern reduces to **O(n)**.

Section 25.1: KMP-Example

Algorithm

This algorithm is a two step process.First we create a auxiliary array lps[] and then use this array for searching the pattern.

Preprocessing :

1. We pre-process the pattern and create an auxiliary array lps[] which is used to skip characters while matching.
2. Here lps[] indicates longest proper prefix which is also suffix.A proper prefix is prefix in which whole string is not included.For example, prefixes of string **ABC** are “ ”, “**A**”, “**AB**” and “**ABC**”. Proper prefixes are “ ”, “**A**” and “**AB**”. Suffixes of the string are “ ”, “**C**”, “**BC**” and “**ABC**”.

Searching

1. We keep matching characters **txt[i]** and **pat[j]** and keep incrementing i and j while **pat[j]** and **txt[i]** keep matching.
2. When we see a mismatch,we know that characters **pat[0..j-1]** match with **txt[i-j+1**…**i-1]**.We also know that lps[j-1] is count of characters of **pat[0**…**j-1]** that are both proper prefix and suffix.From this we can conclude that we do not need to match these **lps[j-1]** characters with **txt[i-j**…**i-1]** because we know that these characters will match anyway.

Implementaion in Java

```
public class KMP {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str = "abcabdabc";
        String pattern = "abc";
        KMP obj = new KMP();
        System.out.println(obj.patternExistKMP(str.toCharArray(), pattern.toCharArray()));
    }

    public int[] computeLPS(char[] str){
        int lps[] = new int[str.length];

        lps[0] = 0;
        int j = 0;
        for(int i =1;i<str.length;i++){
            if(str[j] == str[i]){
                lps[i] = j+1;
                j++;
```

```
                i++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    lps[i] = j+1;
                    i++;
                }
            }

        }

        return lps;
    }

    public boolean patternExistKMP(char[] text,char[] pat){
        int[] lps = computeLPS(pat);
        int i=0,j=0;
        while(i<text.length && j<pat.length){
            if(text[i] == pat[j]){
                i++;
                j++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    i++;
                }
            }
        }

        if(j==pat.length)
            return true;
        return false;
    }

}
```

Chapter 26: Edit Distance Dynamic Algorithm

Section 26.1: Minimum Edits required to convert string 1 to string 2

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2.The Operations can be:

1. **Insert**
2. **Remove**
3. **Replace**

For Example

```
Input: str1 = "geek", str2 = "gesek"
Output: 1
We only need to insert s in first string

Input: str1 = "march", str2 = "cart"
Output: 3
We need to replace m with c and remove character c and then replace h with t
```

To solve this problem we will use a 2D array dp[n+1][m+1] where n is the length of the first string and m is the length of the second string. For our example, if str1 is **azcef** and str2 is **abcdef** then our array will be dp[6][7]and our final answer will be stored at dp[5][6].

```
          (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+---+
(a)| 1 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

For **dp[1][1]** we have to check what can we do to convert **a** into **a**.It will be **0**.For **dp[1][2]** we have to check what can we do to convert **a** into **ab**.It will be **1** because we have to **insert b**.So after 1st iteration our array will look like

```
          (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+---+
(z)| 2 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
```

```
+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

For iteration 2

For **dp[2][1]** we have to check that to convert **az** to **a** we need to remove **z**, hence **dp[2][1]** will be **1**.Similary for **dp[2][2]** we need to replace **z** with **b**, hence **dp[2][2]** will be **1**.So after 2nd iteration our **dp[]** array will look like.

```
         (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+---+
(c)| 3 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(e)| 4 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
(f)| 5 |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

So our **formula** will look like

```
if characters are same
    dp[i][j] = dp[i-1][j-1];
else
    dp[i][j] = 1 + Min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

After last iteration our dp[] array will look like

```
          (a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+---+
(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+---+
(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+---+
(c)| 3 | 2 | 2 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+---+---+---+
(e)| 4 | 3 | 3 | 2 | 2 | 2 | 3 |
+---+---+---+---+---+---+---+---+
(f)| 5 | 4 | 4 | 2 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+---+
```

Implementation in Java

```
public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
```

```
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

Time Complexity

```
O(n^2)
```

Chapter 27: Online algorithms

Theory

Definition 1: An **optimization problem** Π consists of a set of **instances** ΣΠ. For every instance σ∈ΣΠ there is a set Zσ of **solutions** and a **objective function** fσ : Zσ → ℜ≥0 which assigns apositive real value to every solution.
We say OPT(σ) is the value of an optimal solution, A(σ) is the solution of an Algorithm A for the problem Π and wA(σ)=fσ(A(σ)) its value.

Definition 2: An online algorithm A for a minimization problem Π has a **competetive ratio** of r ≥ 1 if there is a constant τ∈ℜ with

> wA(σ) = fσ(A(σ)) ≤ r · OPT(&sigma) + τ

for all instances σ∈ΣΠ. A is called a **r-competitive** online algorithm. Is even

> wA(σ) ≤ r · OPT(&sigma)

for all instances σ∈ΣΠ then A is called a **strictly r-competitive** online algorithm.

Proposition 1.3: LRU and **FWF** are marking algorithm.

Proof: At the beginning of each phase (except for the first one) **FWF** has a cache miss and cleared the cache. that means we have k empty pages. In every phase are maximal k different pages requested, so there will be now eviction during the phase. So **FWF** is a marking algorithm.
Lets assume **LRU** is not a marking algorithm. Then there is an instance σ where **LRU** a marked page x in phase i evicted. Let σt the request in phase i where x is evicted. Since x is marked there has to be a earlier request σt* for x in the same phase, so t* < t. After t* x is the caches newest page, so to got evicted at t the sequence σt*+1,...,σt has to request at least k from x different pages. That implies the phase i has requested at least k+1 different pages which is a contradictory to the phase definition. So **LRU** has to be a marking algorithm.

Proposition 1.4: Every marking algorithm **is strictly k-competitive**.

Proof: Let σ be an instance for the paging problem and l the number of phases for σ. Is l = 1 then is every marking algorithm optimal and the optimal offline algorithm cannot be better.
We assume l ≥ 2. the cost of every marking algorithm for instance σ is bounded from above with l · k because in every phase a marking algorithm cannot evict more than k pages without evicting one marked page.
Now we try to show that the optimal offline algorithm evicts at least k+l-2 pages for σ, k in the first phase and at least one for every following phase except for the last one. For proof lets define l-2 disjunct subsequences of σ. Subsequence i ∈ {1,...,l-2} starts at the second position of phase i+1 and end with the first position of phase i+2. Let x be the first page of phase i+1. At the beginning of subsequence i there is page x and at most k-1 different pages in the optimal offline algorithms cache. In subsequence i are k page request different from x, so the optimal offline algorithm has to evict at least one page for every subsequence. Since at phase 1 beginning the cache is still empty, the optimal offline algorithm causes k evictions during the first phase. That shows that

> wA(σ) ≤ l·k ≤ (k+l-2)k ≤ OPT(σ) · k

Corollary 1.5: LRU and **FWF** are **strictly k-competitive**.

Is there no constant r for which an online algorithm A is r-competitive, we call A **not competitive**.

Proposition 1.6: LFU and **LIFO** are **not competitive**.

Proof: Let l ≥ 2 a constant, k ≥ 2 the cache size. The different cache pages are nubered 1,...,k+1. We look at the following sequence:

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

First page 1 is requested l times than page 2 and so one. At the end there are (l-1) alternating requests for page k and k+1.
LFU and **LIFO** fill their cache with pages 1-k. When page k+1 is requested page k is evicted and vice versa. That means every request of subsequence (k,k+1)l-1 evicts one page. In addition their are k-1 cache misses for the first time use of pages 1-(k-1). So **LFU** and **LIFO** evict exact k-1+2(l-1) pages.
Now we must show that for every constant $\tau \in \Re$ and every constan r ≤ 1 there exists an l so that

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

which is equal to

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

To satisfy this inequality you just have to choose l sufficient big. So **LFU** and **LIFO** are not competetive.

Proposition 1.7: There is **no r-competetive** deterministic online algorithm for paging with **r < k**.

Sources
Basic Material

1. Script Online Algorithms (german), Heiko Roeglin, University Bonn
2. Page replacement algorithm

Further Reading

1. Online Computation and Competetive Analysis by Allan Borodin and Ran El-Yaniv

Source Code

1. Source code for offline caching
2. Source code for adversary game

Section 27.1: Paging (Online Caching)

Preface

Instead of starting with a formal definition, the goal is to approach these topic via a row of examples, introducing definitions along the way. The remark section **Theory** will consist of all definitions, theorems and propositions to give you all information to faster look up specific aspects.

The remark section sources consists of the basis material used for this topic and additional information for further reading. In addition you will find the full source codes for the examples there. Please pay attention that to make the source code for the examples more readable and shorter it refrains from things like error handling etc. It also passes on some specific language features which would obscure the clarity of the example like extensive use of advanced libraries etc.

Paging

The paging problem arises from the limitation of finite space. Let's assume our cache `C` has `k` pages. Now we want to process a sequence of `m` page requests which must have been placed in the cache before they are processed. Of course if `m<=k` then we just put all elements in the cache and it will work, but usually is `m>>k`.

We say a request is a **cache hit**, when the page is already in cache, otherwise, its called a **cache miss**. In that case, we must bring the requested page into the cache and evict another, assuming the cache is full. The Goal is an eviction schedule that **minimizes the number of evictions**.

There are numerous strategies for this problem, let's look at some:

1. **First in, first out (FIFO)**: The oldest page gets evicted
2. **Last in, first out (LIFO)**: The newest page gets evicted
3. **Least recently used (LRU)**: Evict page whose most recent access was earliest
4. **Least frequently used (LFU)**: Evict page that was least frequently requested
5. **Longest forward distance (LFD)**: Evict page in the cache that is not requested until farthest in the future.
6. **Flush when full (FWF)**: clear the cache complete as soon as a cache miss happened

There are two ways to approach this problem:

1. **offline**: the sequence of page requests is known ahead of time
2. **online**: the sequence of page requests is not known ahead of time

Offline Approach

For the first approach look at the topic Applications of Greedy technique. It's third Example **Offline Caching** considers the first five strategies from above and gives you a good entry point for the following.

The example program was extended with the **FWF** strategy:

```
class FWF : public Strategy {
public:
    FWF() : Strategy("FWF")
    {
    }

    int apply(int requestIndex) override
    {
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            // after first empty page all others have to be empty
            else if(cache[i] == emptyPage)
                return i;
        }

        // no free pages
```

```
        return 0;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {

        // no pages free -> miss -> clear cache
        if(cacheMiss && cachePos == 0)
        {
            for(int i = 1; i < cacheSize; ++i)
                cache[i] = emptyPage;
        }
    }
};
```

The full sourcecode is available here. If we reuse the example from the topic, we get the following output:

```
Strategy: FWF

Cache initial: (a,b,c)

Request cache 0 cache 1 cache 2 cache miss
  a       a       b       c
  a       a       b       c
  d       d       X       X       x
  e       d       e       X
  b       d       e       b
  b       d       e       b
  a       a       X       X       x
  c       a       c       X
  f       a       c       f
  d       d       X       X       x
  e       d       e       X
  a       d       e       a
  f       f       X       X       x
  b       f       b       X
  e       f       b       e
  c       c       X       X       x

Total cache misses: 5
```

Even though **LFD** is optimal, **FWF** has fewer cache misses. But the main goal was to minimize the number of evictions and for **FWF** five misses mean 15 evictions, which makes it the poorest choice for this example.

Online Approach

Now we want to approach the online problem of paging. But first we need an understanding how to do it. Obviously an online algorithm cannot be better than the optimal offline algorithm. But how much worse it is? We need formal definitions to answer that question:

Definition 1.1: An **optimization problem** Π consists of a set of **instances** $\Sigma\Pi$. For every instance $\sigma\in\Sigma\Pi$ there is a set $Z\sigma$ of **solutions** and a **objective function** $f\sigma : Z\sigma \rightarrow \Re\geq 0$ which assigns apositive real value to every solution. We say $OPT(\sigma)$ is the value of an optimal solution, $A(\sigma)$ is the solution of an Algorithm A for the problem Π and $wA(\sigma)=f\sigma(A(\sigma))$ its value.

Definition 1.2: An online algorithm A for a minimization problem Π has a **competetive ratio** of $r \geq 1$ if there is a constant $\tau\in\Re$ with

> $wA(\sigma) = f\sigma(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$

for all instances $\sigma \in \Sigma\Pi$. A is called a **r-competitive** online algorithm. Is even

> $wA(\sigma) \leq r \cdot OPT(\sigma)$

for all instances $\sigma \in \Sigma\Pi$ then A is called a **strictly r-competitive** online algorithm.

So the question is how **competitive** is our online algorithm compared to an optimal offline algorithm. In their famous book Allan Borodin and Ran El-Yaniv used another scenario to describe the online paging situation:

There is an **evil adversary** who knows your algorithm and the optimal offline algorithm. In every step, he tries to request a page which is worst for you and simultaneously best for the offline algorithm. the **competitive factor** of your algorithm is the factor on how badly your algorithm did against the adversary's optimal offline algorithm. If you want to try to be the adversary, you can try the Adversary Game (try to beat the paging strategies).

Marking Algorithms

Instead of analysing every algorithm separately, let's look at a special online algorithm family for the paging problem called **marking algorithms**.

Let $\sigma=(\sigma 1,\ldots,\sigma p)$ an instance for our problem and k our cache size, than σ can be divided into phases:

- Phase 1 is the maximal subsequence of σ from the start till maximal k different pages are requested
- Phase $i \geq 2$ is the maximal subsequence of σ from the end of pase i-1 till maximal k different pages are requested

For example with k = 3:

> $$\sigma = \left(\overbrace{a, b, d, a}^{phase\ 1}, \overbrace{e, a, f, a, f}^{phase\ 2}, \overbrace{b, d, a}^{phase\ 3}, \overbrace{c, c, d}^{phase\ 4} \right)$$

A marking algorithm (implicitly or explicitly) maintains whether a page is marked or not. At the beginning of each phase are all pages unmarked. Is a page requested during a phase it gets marked. An algorithm is a marking algorithm **iff** it never evicts a marked page from cache. That means pages which are used during a phase will not be evicted.

Proposition 1.3: LRU and **FWF** are marking algorithm.

Proof: At the beginning of each phase (except for the first one) **FWF** has a cache miss and cleared the cache. that means we have k empty pages. In every phase are maximal k different pages requested, so there will be now eviction during the phase. So **FWF** is a marking algorithm.
Let's assume **LRU** is not a marking algorithm. Then there is an instance σ where **LRU** a marked page x in phase i evicted. Let σt the request in phase i where x is evicted. Since x is marked there has to be a earlier request σt^* for x in the same phase, so $t^* < t$. After t^* x is the caches newest page, so to got evicted at t the sequence $\sigma t^*+1,\ldots,\sigma t$ has to request at least k from x different pages. That implies the phase i has requested at least k+1 different pages which is a contradictory to the phase definition. So **LRU** has to be a marking algorithm.

Proposition 1.4: Every marking algorithm **is strictly k-competitive**.

Proof: Let σ be an instance for the paging problem and l the number of phases for σ. Is l = 1 then is every marking algorithm optimal and the optimal offline algorithm cannot be better.
We assume l ≥ 2. the cost of every marking algorithm, for instance, σ is bounded from above with l · k because in every phase a marking algorithm cannot evict more than k pages without evicting one marked page.
Now we try to show that the optimal offline algorithm evicts at least k+l-2 pages for σ, k in the first phase and at least one for every following phase except for the last one. For proof lets define l-2 disjunct subsequences of σ. Subsequence i ∈ {1,...,l-2} starts at the second position of phase i+1 and end with the first position of phase i+2. Let x be the first page of phase i+1. At the beginning of subsequence i there is page x and at most k-1 different pages in the optimal offline algorithms cache. In subsequence i are k page request different from x, so the optimal offline algorithm has to evict at least one page for every subsequence. Since at phase 1 beginning the cache is still empty, the optimal offline algorithm causes k evictions during the first phase. That shows that

> $wA(\sigma) \leq l \cdot k \leq (k+l-2)k \leq OPT(\sigma) \cdot k$

Corollary 1.5: LRU and **FWF** are **strictly k-competitive**.

Excercise: Show that **FIFO** is no marking algorithm, but **strictly k-competitive**.

Is there no constant r for which an online algorithm A is r-competitive, we call A **not competitive**

Proposition 1.6: LFU and **LIFO** are **not competitive**.

Proof: Let l ≥ 2 a constant, k ≥ 2 the cache size. The different cache pages are nubered 1,...,k+1. We look at the following sequence:

$$\sigma = \left(1^l, 2^l, \ldots, (k-1)^l, (k, k+1)^{l-1}\right)$$

The first page 1 is requested l times than page 2 and so one. At the end, there are (l-1) alternating requests for page k and k+1.
LFU and **LIFO** fill their cache with pages 1-k. When page k+1 is requested page k is evicted and vice versa. That means every request of subsequence (k,k+1)l-1 evicts one page. In addition, their are k-1 cache misses for the first time use of pages 1-(k-1). So **LFU** and **LIFO** evict exact k-1+2(l-1) pages.
Now we must show that for every constant $\tau \in \Re$ and every constant r ≤ 1 there exists an l so that

$$w_{\mathrm{LFU}}(\sigma) = w_{\mathrm{LIFO}}(\sigma) > r \cdot OPT(\sigma) + \tau$$

which is equal to

$$k - 1 + 2(l-1) > r(k+1) + \tau \iff l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

To satisfy this inequality you just have to choose l sufficient big. So **LFU** and **LIFO** are not competitive.

Proposition 1.7: There is **no r-competetive** deterministic online algorithm for paging with **r < k**.

The proof for this last proposition is rather long and based of the statement that **LFD** is an optimal offline algorithm. The interested reader can look it up in the book of Borodin and El-Yaniv (see sources below).

The Question is whether we could do better. For that, we have to leave the deterministic approach behind us and start to randomize our algorithm. Clearly, its much harder for the adversary to punish your algorithm if it's randomized.

Randomized paging will be discussed in one of next examples...

Chapter 28: Sorting

Parameter	Description
Stability	A sorting algorithm is **stable** if it preserves the relative order of equal elements after sorting.
In place	A sorting algorithm is **in-place** if it sorts using only `O(1)` auxiliary memory (not counting the array that needs to be sorted).
Best case complexity	A sorting algorithm has a best case time complexity of `O(T(n))` if its running time is **at least** `T(n)` for all possible inputs.
Average case complexity	A sorting algorithm has an average case time complexity of `O(T(n))` if its running time, **averaged over all possible inputs**, is `T(n)`.
Worst case complexity	A sorting algorithm has a worst case time complexity of `O(T(n))` if its running time is **at most** `T(n)`.

Section 28.1: Stability in Sorting

Stability in sorting means whether a sort algorithm maintains the relative order of the equals keys of the original input in the result output.

So a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Consider a list of pairs:

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

Now we will sort the list using the first element of each pair.

A **stable sorting** of this list will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

Because `(9, 3)` appears after `(9, 7)` in the original list as well.

An **unstable sorting** will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

Unstable sort may generate the same output as the stable sort but not always.

Well-known stable sorts:

- Merge sort
- Insertion sort
- Radix sort
- Tim sort
- Bubble Sort

Well-known unstable sorts:

- Heap sort
- Quick sort

Chapter 29: Bubble Sort

Parameter	Description
Stable	Yes
In place	Yes
Best case complexity	O(n)
Average case complexity	O(n^2)
Worst case complexity	O(n^2)
Space complexity	O(1)

Section 29.1: Bubble Sort

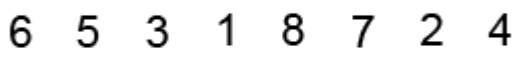
The `BubbleSort` compares each successive pair of elements in an unordered list and inverts the elements if they are not in order.

The following example illustrates the bubble sort on the list `{6,5,3,1,8,7,2,4}` (pairs that were compared in each step are encapsulated in '**'):

```
{6,5,3,1,8,7,2,4}
{**5,6**,3,1,8,7,2,4} -- 5 < 6 -> swap
{5,**3,6**,1,8,7,2,4} -- 3 < 6 -> swap
{5,3,**1,6**,8,7,2,4} -- 1 < 6 -> swap
{5,3,1,**6,8**,7,2,4} -- 8 > 6 -> no swap
{5,3,1,6,**7,8**,2,4} -- 7 < 8 -> swap
{5,3,1,6,7,**2,8**,4} -- 2 < 8 -> swap
{5,3,1,6,7,2,**4,8**} -- 4 < 8 -> swap
```

After one iteration through the list, we have `{5,3,1,6,7,2,4,8}`. Note that the greatest unsorted value in the array (8 in this case) will always reach its final position. Thus, to be sure the list is sorted we must iterate n-1 times for lists of length n.

Graphic:



Section 29.2: Implementation in C & C++

An example implementation of `BubbleSort` in `C++`:

```
void bubbleSort(vector<int>numbers)
{
    for(int i = numbers.size() - 1; i >= 0; i--) {
        for(int j = 1; j <= i; j++) {
            if(numbers[j-1] > numbers[j]) {
                swap(numbers[j-1],numbers(j));
```

```
            }
        }
    }
}
```

C Implementation

```
void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */

        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

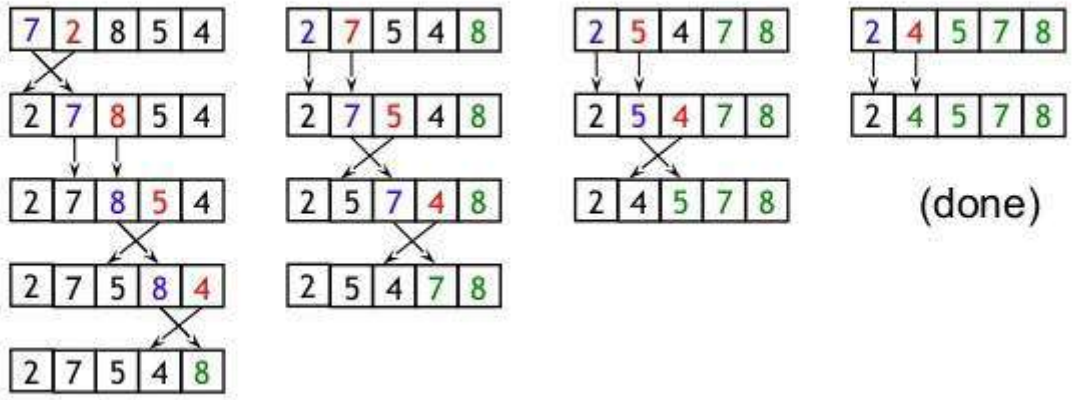
Bubble Sort with pointer

```
void pointer_bubble_sort(long * list, long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if ( * (list + d ) > *(list+d+1))
      {
        /* Swapping */

        t         = * (list + d );
        * (list + d )   = * (list + d + 1 );
        * (list + d + 1) = t;
      }
    }
  }
}
```

Section 29.3: Implementation in C#

Bubble sort is also known as **Sinking Sort**. It is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Bubble sort example



Implementation of Bubble Sort

I used C# language to implement bubble sort algorithm

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--)
            {
                if (input[j] <= input[j + 1]) continue;
                var temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBubble(input);
        return input;
    }
}
```

Section 29.4: Python Implementation

```
#!/usr/bin/python

input_list = [10,1,2,11]

for i in range(len(input_list)):
  for j in range(i):
    if int(input_list[j]) > int(input_list[j+1]):
      input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list
```

Section 29.5: Implementation in Java

```
public class MyBubbleSort {

    public static void bubble_srt(int array[]) {//main logic
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
                if (array[i] > array[k]) {
                    swapNumbers(i, k, array);
                }
            }
            printNumbers(array);
        }
    }

    private static void swapNumbers(int i, int j, int[] array) {

        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void printNumbers(int[] input) {

        for (int i = 0; i < input.length; i++) {
            System.out.print(input[i] + ", ");
        }
        System.out.println("\n");
    }

    public static void main(String[] args) {
        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
        bubble_srt(input);

    }
}
```

Section 29.6: Implementation in Javascript

```
        function bubbleSort(a)
          {
                var swapped;
                do {
                    swapped = false;
                    for (var i=0; i < a.length-1; i++) {
                        if (a[i] > a[i+1]) {
                            var temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                            swapped = true;
                        }
                    }
                } while (swapped);
            }

   var a = [3, 203, 34, 746, 200, 984, 198, 764, 9];
```

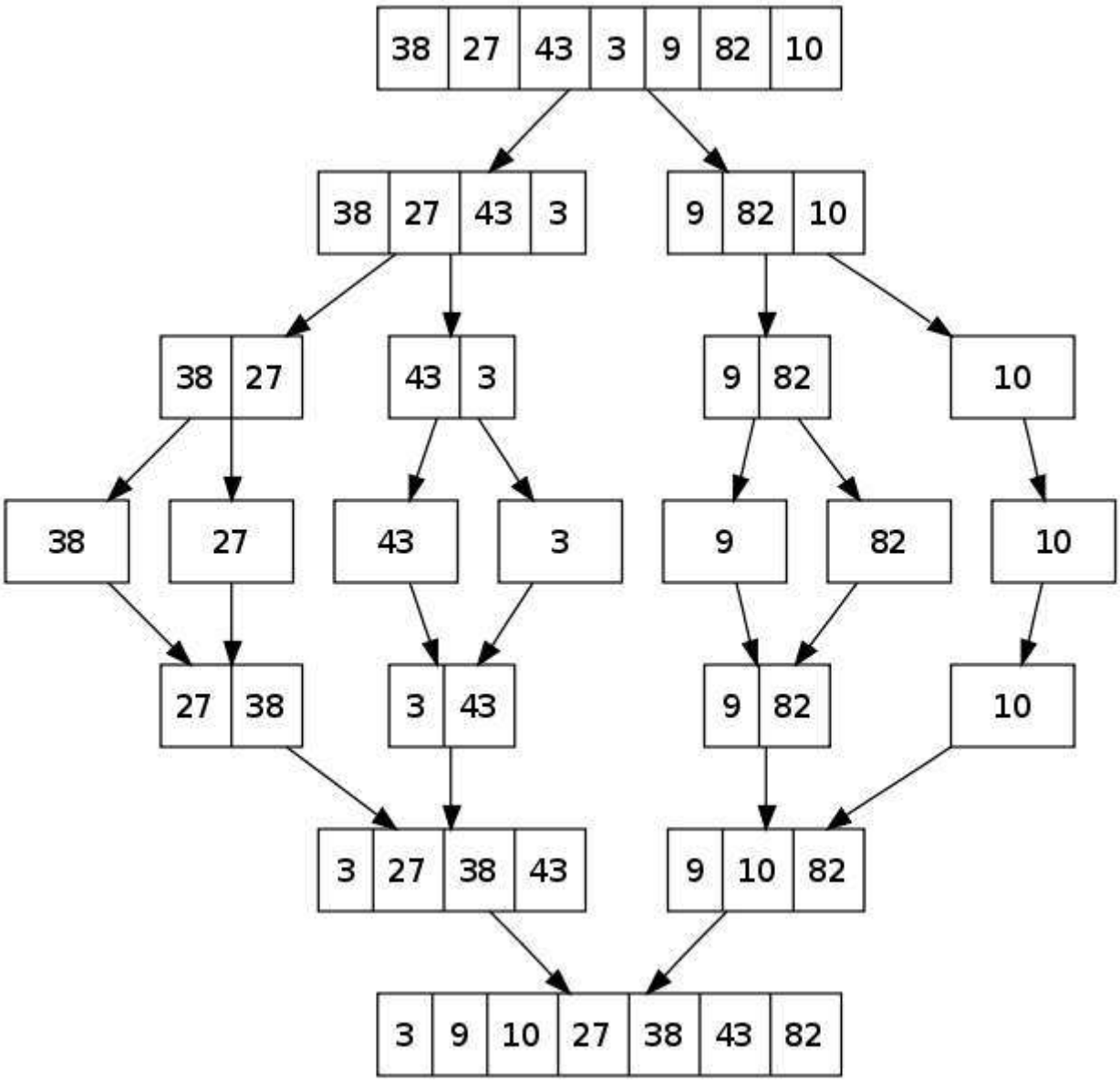
```
bubbleSort(a);
console.log(a); //logs [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]
```

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

Merge Sort is a divide-and-conquer algorithm. It divides the input list of length n in half successively until there are n lists of size 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built.

An example:



Time Complexity: `T(n) = 2T(n/2) + Θ(n)`

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is `Θ(nLogn)`. Time complexity of *Merge Sort* is `Θ(nLogn)` in all 3 cases (*worst, average and best*) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: `O(n)`

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: Not in a typical implementation

Stable: Yes

Section 30.2: Merge Sort Implementation in Go

```
package main

import "fmt"

func mergeSort(a []int) []int {
    if len(a) < 2 {
        return a
    }
    m := (len(a)) / 2

    f := mergeSort(a[:m])
    s := mergeSort(a[m:])

    return merge(f, s)
}

func merge(f []int, s []int) []int {
    var i, j int
    size := len(f) + len(s)

    a := make([]int, size, size)

    for z := 0; z < size; z++ {
        lenF := len(f)
        lenS := len(s)

        if i > lenF-1 && j <= lenS-1 {
            a[z] = s[j]
            j++
        } else if j > lenS-1 && i <= lenF-1 {
            a[z] = f[i]
            i++
        } else if f[i] < s[j] {
            a[z] = f[i]
            i++
        } else {
            a[z] = s[j]
            j++
        }
    }

    return a
}

func main() {
    a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
    fmt.Println(a)
    fmt.Println(mergeSort(a))
}
```

Section 30.3: Merge Sort Implementation in C & C#

C Merge Sort

```
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // Two temporary arrays to
  hold the two arrays to be merged
  int n1,n2,i,j,k;
  n1=m-l+1;
  n2=h-m;

  for(i=0; i<n1; i++)
    arr1[i]=arr[l+i];
  for(j=0; j<n2; j++)
    arr2[j]=arr[m+j+1];

  arr1[i]=9999;  // To mark the end of each temporary array
  arr2[j]=9999;

  i=0;
  j=0;
  for(k=l; k<=h; k++) { //process of combining two sorted arrays
    if(arr1[i]<=arr2[j])
      arr[k]=arr1[i++];
    else
      arr[k]=arr2[j++];
  }

  return 0;
}

int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high) {
    mid=(low+high)/2;
    // Divide and Conquer
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
    // Combine
    merge(arr,low,mid,high);
  }

  return 0;
}
```

C# Merge Sort

```
public class MergeSort
    {
        static void Merge(int[] input, int l, int m, int r)
        {
            int i, j;
            var n1 = m - l + 1;
            var n2 = r - m;

            var left = new int[n1];
            var right = new int[n2];

            for (i = 0; i < n1; i++)
            {
                left[i] = input[l + i];
            }
```

```
            for (j = 0; j < n2; j++)
            {
                right[j] = input[m + j + 1];
            }

            i = 0;
            j = 0;
            var k = l;

            while (i < n1 && j < n2)
            {
                if (left[i] <= right[j])
                {
                    input[k] = left[i];
                    i++;
                }
                else
                {
                    input[k] = right[j];
                    j++;
                }
                k++;
            }

            while (i < n1)
            {
                input[k] = left[i];
                i++;
                k++;
            }

            while (j < n2)
            {
                input[k] = right[j];
                j++;
                k++;
            }
        }

        static void SortMerge(int[] input, int l, int r)
        {
            if (l < r)
            {
                int m = l + (r - l) / 2;
                SortMerge(input, l, m);
                SortMerge(input, m + 1, r);
                Merge(input, l, m, r);
            }
        }

        public static int[] Main(int[] input)
        {
            SortMerge(input, 0, input.Length - 1);
            return input;
        }
    }
```

Section 30.4: Merge Sort Implementation in Java

Below there is the implementation in Java using a generics approach. It is the same algorithm, which is presented above.

```
public interface InPlaceSort<T extends Comparable<T>> {
void sort(final T[] elements); }


public class MergeSort < T extends Comparable < T >> implements InPlaceSort < T > {


@Override
public void sort(T[] elements) {
    T[] arr = (T[]) new Comparable[elements.length];
    sort(elements, arr, 0, elements.length - 1);
}

// We check both our sides and then merge them
private void sort(T[] elements, T[] arr, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    sort(elements, arr, low, mid);
    sort(elements, arr, mid + 1, high);
    merge(elements, arr, low, high, mid);
}


private void merge(T[] a, T[] b, int low, int high, int mid) {
    int i = low;
    int j = mid + 1;

    // We select the smallest element of the two. And then we put it into b
    for (int k = low; k <= high; k++) {

        if (i <= mid && j <= high) {
            if (a[i].compareTo(a[j]) >= 0) {
                b[k] = a[j++];
            } else {
                b[k] = a[i++];
            }
        } else if (j > high && i <= mid) {
            b[k] = a[i++];
        } else if (i > mid && j <= high) {
            b[k] = a[j++];
        }
    }

    for (int n = low; n <= high; n++) {
        a[n] = b[n];
    }}}
```

Section 30.5: Merge Sort Implementation in Python

```
def merge(X, Y):
    " merge two sorted lists "
    p1 = p2 = 0
    out = []
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
            out.append(X[p1])
            p1 += 1
        else:
            out.append(Y[p2])
            p2 += 1
    out += X[p1:] + Y[p2:]
```

```
    return out

def mergeSort(A):
    if len(A) <= 1:
        return A
    if len(A) == 2:
        return sorted(A)

    mid = len(A) / 2
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))

if __name__ == "__main__":
    # Generate 20 random numbers and sort them
    A = [randint(1, 100) for i in xrange(20)]
    print mergeSort(A)
```

Section 30.6: Bottoms-up Java Implementation

```
public class MergeSortBU {
    private static Integer[] array = { 4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70,
60,80,0,9,67,54,51,52,24,54,7 };

    public MergeSortBU() {
    }

    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid, int hi) {

        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }

        int i = lo;
        int j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
                arrayToSort[k] = aux[j++];
            else if (j > hi)
                arrayToSort[k] = aux[i++];
            else if (isLess(aux[i], aux[j])) {
                arrayToSort[k] = aux[i++];
            } else {
                arrayToSort[k] = aux[j++];
            }

        }
    }

    public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
        int N = arrayToSort.length;
        for (int sz = 1; sz < N; sz = sz + sz) {
            for (int low = 0; low < N; low = low + sz + sz) {
                System.out.println("Size:"+ sz);
                merge(arrayToSort, aux, low, low + sz -1 ,Math.min(low + sz + sz - 1, N - 1));
                print(arrayToSort);
            }
        }

    }

    public static boolean isLess(Comparable a, Comparable b) {
        return a.compareTo(b) <= 0;
```

```
    }

    private static void print(Comparable[] array)
{http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        StringBuffer buffer = new
StringBuffer();http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        for (Comparable value : array) {
            buffer.append(value);
            buffer.append(' ');
        }
        System.out.println(buffer);
    }

    public static void main(String[] args) {
        Comparable[] aux = new Comparable[array.length];
        print(array);
        MergeSortBU.sort(array, aux, 0, array.length - 1);
    }
}
```

Chapter 31: Insertion Sort

Section 31.1: Haskell Implementation

```
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: Ord a => a-> [a] -> [a]
insert n [] = [n]
insert n (x:xs) | n <= x    = (n:x:xs)
                | otherwise = x:insert n xs
```

Chapter 32: Bucket Sort

Section 32.1: C# Implementation

```
public class BucketSort
{
    public static void SortBucket(ref int[] input)
    {
        int minValue = input[0];
        int maxValue = input[0];
        int k = 0;

        for (int i = input.Length - 1; i >= 1; i--)
        {
            if (input[i] > maxValue) maxValue = input[i];
            if (input[i] < minValue) minValue = input[i];
        }

        List<int>[] bucket = new List<int>[maxValue - minValue + 1];

        for (int i = bucket.Length - 1; i >= 0; i--)
        {
            bucket[i] = new List<int>();
        }

        foreach (int i in input)
        {
            bucket[i - minValue].Add(i);
        }

        foreach (List<int> b in bucket)
        {
            if (b.Count > 0)
            {
                foreach (int t in b)
                {
                    input[k] = t;
                    k++;
                }
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBucket(ref input);
        return input;
    }
}
```

Chapter 33: Quicksort

Section 33.1: Quicksort Basics

Quicksort is a sorting algorithm that picks an element ("the pivot") and reorders the array forming two partitions such that all elements less than the pivot come before it and all elements greater come after. The algorithm is then applied recursively to the partitions until the list is sorted.

1. Lomuto partition scheme mechanism :

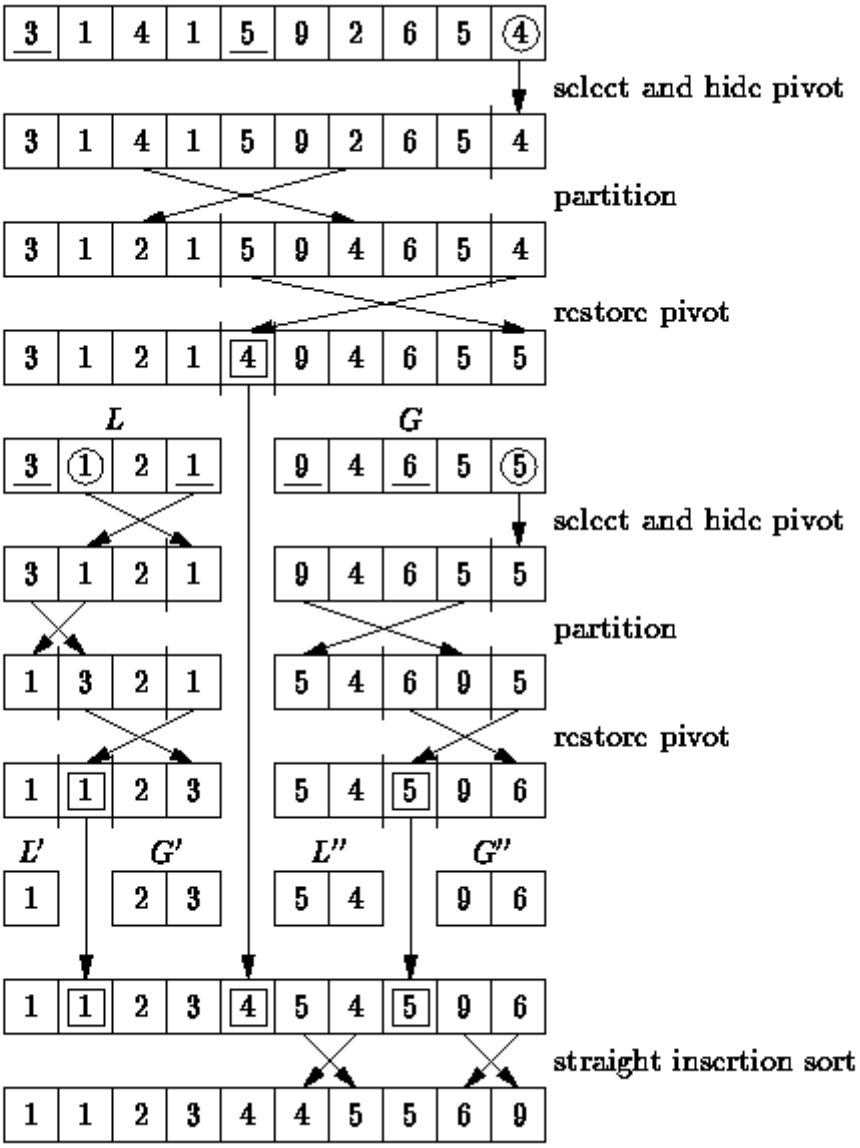
This scheme chooses a pivot which is typically the last element in the array. The algorithm maintains the index to put the pivot in variable i and each time it finds an element less than or equal to pivot, this index is incremented and that element would be placed before the pivot.

```
partition(A, low, high) is
pivot := A[high]
i := low
for j := low to high – 1 do
    if A[j] ≤ pivot then
        swap A[i] with A[j]
        i := i + 1
swap A[i] with A[high]
return i
```

Quick Sort mechanism :

```
quicksort(A, low, high) is
if low < high then
    p := partition(A, low, high)
    quicksort(A, low, p – 1)
    quicksort(A, p + 1, high)
```

Example of quick sort:



2. Hoare partition scheme:

It uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater or equal than the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index. Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.

```
quicksort(A, lo, hi) is
if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Partition :

```
partition(A, lo, hi) is
pivot := A[lo]
i := lo - 1
j := hi + 1
loop forever
    do:
        i := i + 1
```

```
    while A[i] < pivot do

    do:
        j := j - 1
    while A[j] > pivot do

    if i >= j then
        return j

    swap A[i] with A[j]
```

Section 33.2: Quicksort in Python

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) / 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])
```

Prints "[1, 1, 2, 3, 6, 8, 10]"

Section 33.3: Lomuto partition java implementation

```
public class Solution {

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] ar = new int[n];
    for(int i=0; i<n; i++)
      ar[i] = sc.nextInt();
     quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high)
 {
    if(low<high)
    {
        int p = partition(ar, low, high);
        quickSort(ar, 0 , p-1);
        quickSort(ar, p+1, high);
    }
 }
public static int partition(int[] ar, int l, int r)
 {
    int pivot = ar[r];
    int i =l;
    for(int j=l; j<r; j++)
     {
        if(ar[j] <= pivot)
         {
            int t = ar[j];
            ar[j] = ar[i];
            ar[i] = t;
            i++;
        }
```

```
    }
    int t = ar[i];
    ar[i] = ar[r];
    ar[r] = t;

    return i;
}
```

Chapter 34: Counting Sort

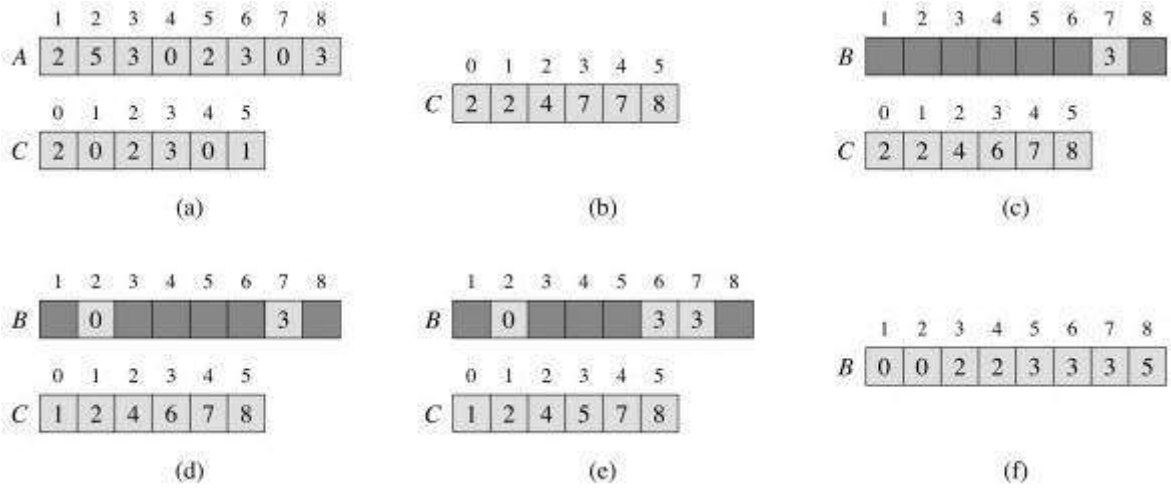
Section 34.1: Counting Sort Basic Information

Counting sort is an integer sorting algorithm for a collection of objects that sorts according to the keys of the objects.

Steps

1. Construct a working array *C* that has size equal to the range of the input array *A*.
2. Iterate through *A*, assigning *C*[x] based on the number of times x appeared in *A*.
3. Transform *C* into an array where *C*[x] refers to the number of values ≤ x by iterating through the array, assigning to each *C*[x] the sum of its prior value and all values in *C* that come before it.
4. Iterate backwards through *A*, placing each value in to a new sorted array *B* at the index recorded in *C*. This is done for a given *A*[x] by assigning *B*[*C*[*A*[x]]] to *A*[x], and decrementing *C*[*A*[x]] in case there were duplicate values in the original unsorted array.

Example of Counting Sort



Auxiliary Space: O(n+k)
Time Complexity: Worst-case: O(n+k), Best-case: O(n), Average-case O(n+k)

Section 34.2: Psuedocode Implementation

Constraints:

1. Input (an array to be sorted)
2. Number of element in input (n)
3. Keys in the range of *0..k-1* (k)
4. Count (an array of number)

Pseudocode:

```
for x in input:
    count[key(x)] += 1
total = 0
for i in range(k):
    oldCount = count[i]
    count[i] = total
    total += oldCount
```

```
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1
return output
```

Chapter 35: Heap Sort

Section 35.1: C# Implementation

```
public class HeapSort
{
    public static void Heapify(int[] input, int n, int i)
    {
        int largest = i;
        int l = i + 1;
        int r = i + 2;

        if (l < n && input[l] > input[largest])
            largest = l;

        if (r < n && input[r] > input[largest])
            largest = r;

        if (largest != i)
        {
            var temp = input[i];
            input[i] = input[largest];
            input[largest] = temp;
            Heapify(input, n, largest);
        }
    }

    public static void SortHeap(int[] input, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
            Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
            var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
        SortHeap(input, input.Length);
        return input;
    }
}
```

Section 35.2: Heap Sort Basic Information

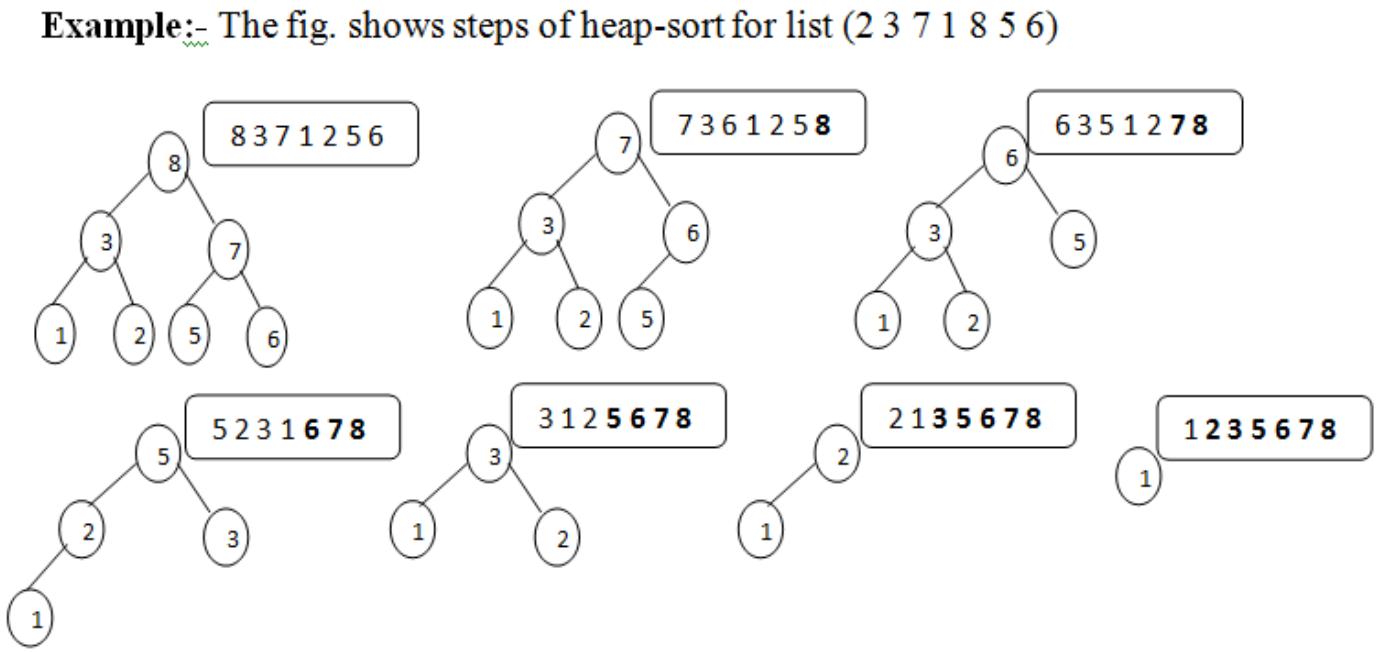
Heap sort is a comparison based sorting technique on binary heap data structure. It is similar to selection sort in which we first find the maximum element and put it at the end of the data structure. Then repeat the same process for the remaining items.

Pseudo code for Heap Sort:

```
function heapsort(input, count)
```

```
    heapify(a,count)
    end <- count - 1
    while end -> 0 do
    swap(a[end],a[0])
    end<-end-1
    restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

Example of Heap Sort:



Auxiliary Space: O(1)
Time Complexity: O(nlogn)

Chapter 36: Cycle Sort

Section 36.1: Pseudocode Implementation

```
(input)
output = 0
for cycleStart from 0 to length(array) - 2
    item = array[cycleStart]
    pos = cycleStart
    for i from cycleStart + 1 to length(array) - 1
        if array[i] < item:
            pos += 1
    if pos == cycleStart:
        continue
    while item == array[pos]:
        pos += 1
    array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
        pos = cycleStart
        for i from cycleStart + 1 to length(array) - 1
            if array[i] < item:
                pos += 1
        while item == array[pos]:
            pos += 1
        array[pos], item = item, array[pos]
        writes += 1
return outout
```

Chapter 37: Odd-Even Sort

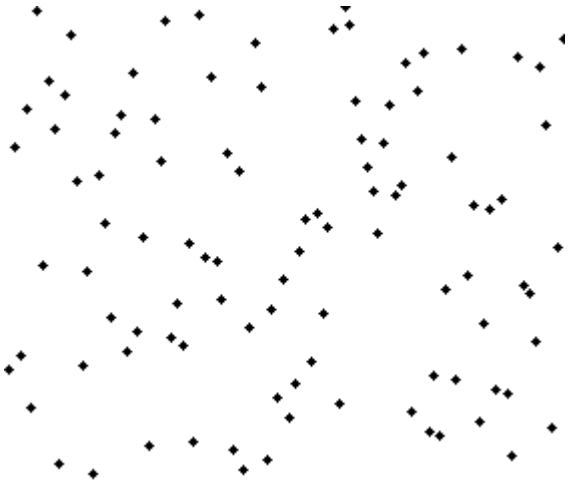
Section 37.1: Odd-Even Sort Basic Information

An Odd-Even Sort or brick sort is a simple sorting algorithm, which is developed for use on parallel processors with local interconnection. It works by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order the elements are switched. The next step repeats this for even/odd indexed pairs. Then it alternates between odd/even and even/odd steps until the list is sorted.

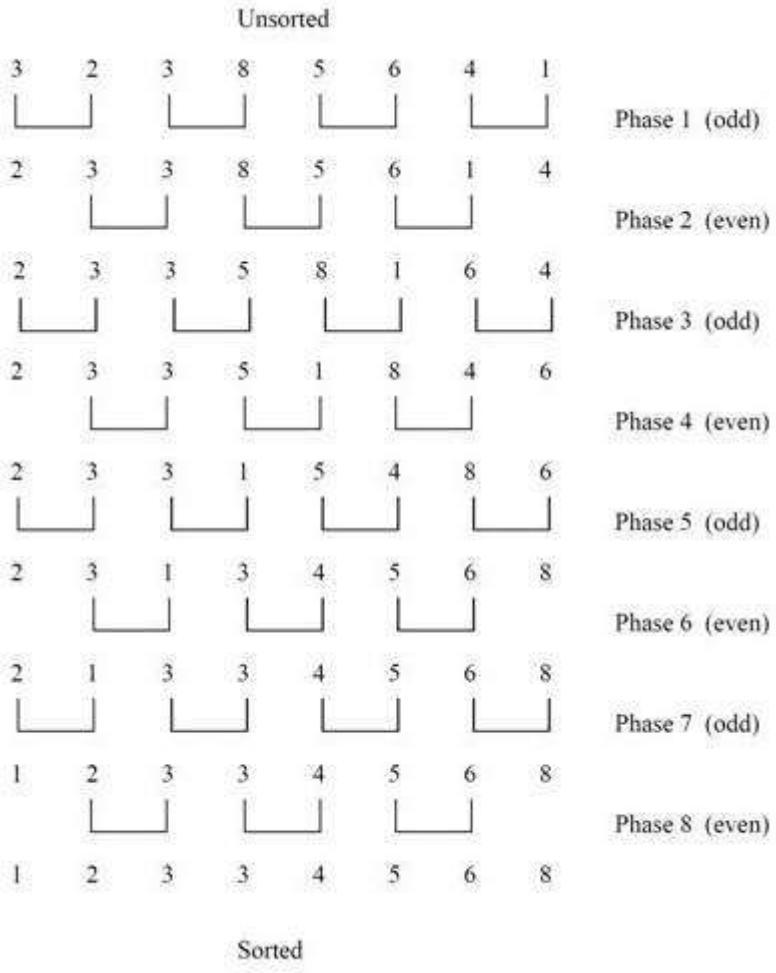
Pseudo code for Odd-Even Sort:

```
if n>2 then
    1. apply odd-even merge(n/2) recursively to the even subsequence a0, a2, ..., an-2 and to the
odd subsequence a1, a3, , ..., an-1
    2. comparison [i : i+1] for all i element {1, 3, 5, 7, ..., n-3}
else
    comparison [0 : 1]
```

Wikipedia has best illustration of Odd-Even sort:



Example of Odd-Even Sort:



Implementation:

I used C# language to implement Odd-Even Sort Algorithm.

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
        var sort = false;

        while (!sort)
        {
            sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
            for (var i = 0; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
```

```
        }
    }

    public static int[] Main(int[] input)
    {
        SortOddEven(input, input.Length);
        return input;
    }
}
```

Auxiliary Space: O(n)
Time Complexity: O(n)

Chapter 38: Selection Sort

Section 38.1: Elixir Implementation

```
defmodule Selection do

  def sort(list) when is_list(list) do
    do_selection(list, [])
  end

  def do_selection([head|[]], acc) do
    acc ++ [head]
  end

  def do_selection(list, acc) do
    min = min(list)
    do_selection(:lists.delete(min, list), acc ++ [min])
  end

  defp min([first|[second|[]]]) do
    smaller(first, second)
  end

  defp min([first|[second|tail]]) do
    min([smaller(first, second)|tail])
  end

  defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    else
      e2
    end
  end
end

Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

Section 38.2: Selection Sort Basic Information

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

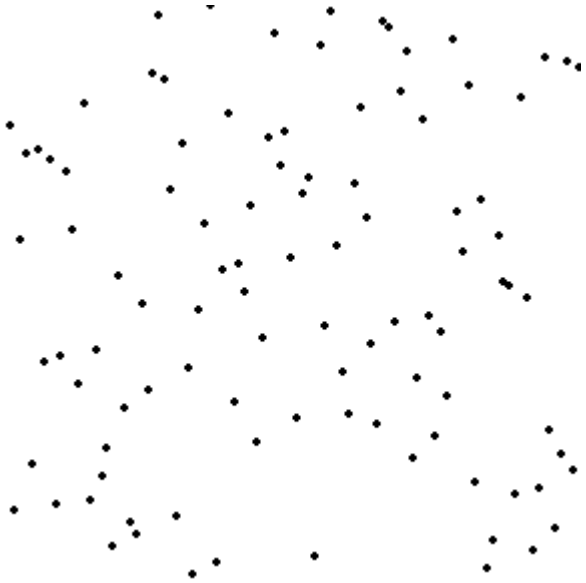
The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Pseudo code for Selection sort:

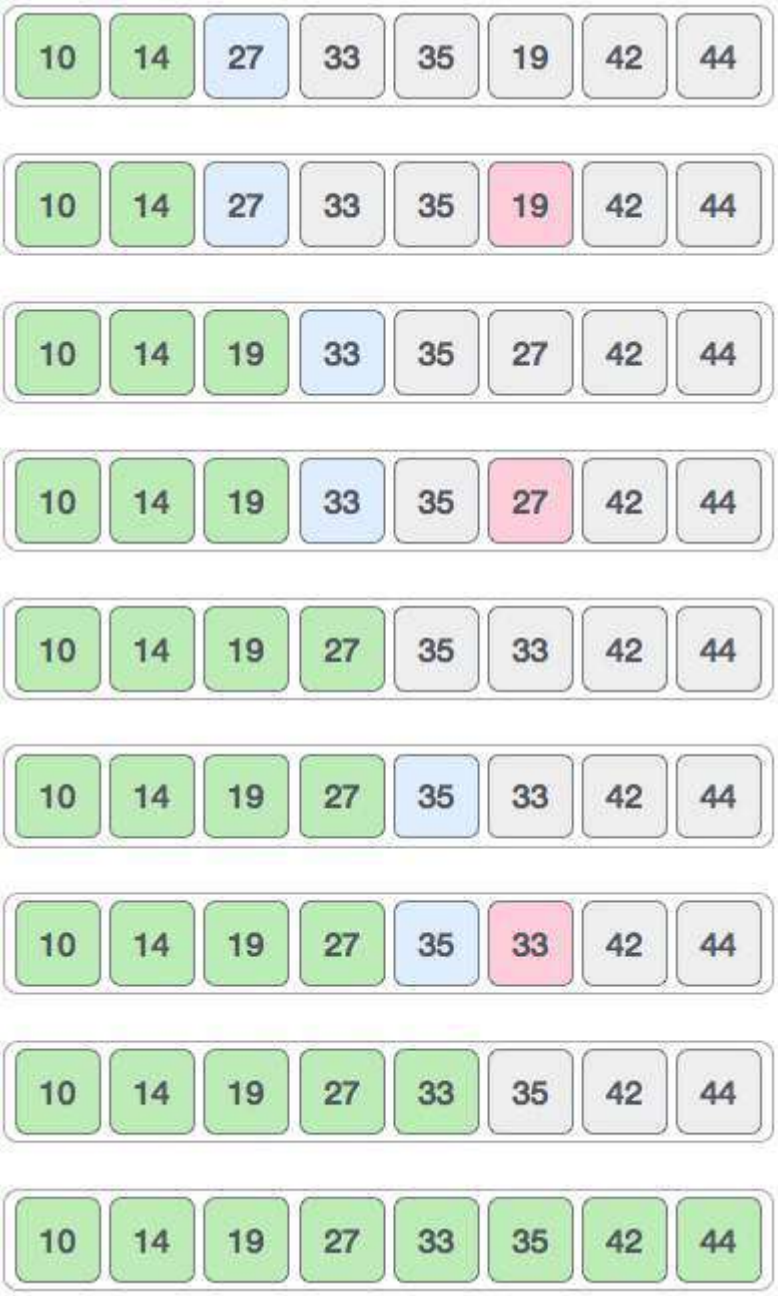
```
function select(list[1..n], k)
 for i from 1 to k
```

```
    minIndex = i
    minValue = list[i]
    for j from i+1 to n
        if list[j] < minValue
            minIndex = j
            minValue = list[j]
    swap list[i] and list[minIndex]
return list[k]
```

Visualization of selection sort:



Example of Selection sort:



Auxiliary Space: O(n)
Time Complexity: O(n^2)

Section 38.3: Implementation of Selection sort in C#

I used C# language to implement Selection sort algorithm.

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
            var temp = input[minId];
```

```
            input[minId] = input[i];
            input[i] = temp;
        }
    }

    public static int[] Main(int[] input)
    {
        SortSelection(input, input.Length);
        return input;
    }
}
```

Chapter 39: Searching

Section 39.1: Binary Search

Introduction

Binary Search is a Divide and Conquer search algorithm. It uses `O(log n)` time to find the location of an element in a search space where `n` is the size of the search space.

Binary Search works by halving the search space at each iteration after comparing the target value to the middle value of the search space.

To use Binary Search, the search space must be ordered (sorted) in some way. Duplicate entries (ones that compare as equal according to the comparison function) cannot be distinguished, though they don't violate the Binary Search property.

Conventionally, we use less than (<) as the comparison function. If a < b, it will return true. if a is not less than b and b is not less than a, a and b are equal.

Example Question

You are an economist, a pretty bad one though. You are given the task of finding the equilibrium price (that is, the price where supply = demand) for rice.

Remember the higher a price is set, the larger the supply and the lesser the demand

As your company is very efficient at calculating market forces, you can instantly get the supply and demand in units of rice when the price of rice is set at a certain price `p`.

Your boss wants the equilibrium price ASAP, but tells you that the equilibrium price can be a positive integer that is at most `10^17` and there is guaranteed to be exactly 1 positive integer solution in the range. So get going with your job before you lose it!

You are allowed to call functions `getSupply(k)` and `getDemand(k)`, which will do exactly what is stated in the problem.

Example Explanation

Here our search space is from `1` to `10^17`. Thus a linear search is infeasible.

However, notice that as the `k` goes up, `getSupply(k)` increases and `getDemand(k)` decreases. Thus, for any `x > y`, `getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)`. Therefore, this search space is monotonic and we can use Binary Search.

The following psuedocode demonstrates the usage of Binary Search:

```
high = 100000000000000000      <- Upper bound of search space
low = 1                        <- Lower bound of search space
while high - low > 1
    mid = (high + low) / 2     <- Take the middle value
    supply = getSupply(mid)
    demand = getDemand(mid)
    if supply > demand
        high = mid              <- Solution is in lower half of search space
```

```
    else if demand > supply
        low = mid              <- Solution is in upper half of search space
    else                       <- supply==demand condition
        return mid             <- Found solution
```

This algorithm runs in `~O(log 10^17)` time. This can be generalized to `~O(log S)` time where S is the size of the search space since at every iteration of the **`while`** loop, we halved the search space (*from [low:high] to either [low:mid] or [mid:high]*).

C Implementation of Binary Search with Recursion

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Section 39.2: Rabin Karp

The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text.Its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm) where n is the length of the text and m is the length of the pattern.

Algorithm implementation in java for string matching

```
void RabinfindPattern(String text,String pattern){
    /*
    q a prime number
    p hash value for pattern
    t hash value for text
    d is the number of unique characters in input alphabet
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash value calculating function
    for (i=0;i<m-1;i++)
            h = (h*d)%q;
        for (i=0;i<m;i++){
        p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//search for the pattern
```

```
    for(i=0;i<end-m;i++){
        if(p==t){
//if the hash value matches match them character by character
            for(j=0;j<m;j++)
                if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
        }
        if(i<end-m){
            t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
            if(t<0)
                t=t+q;
        }
    }
}
```

While calculating hash value we are dividing it by a prime number in order to avoid collision.After dividing by prime number the chances of collision will be less, but still ther is a chance that the hash value can be same for two strings,so when we get a match we have to check it character by character to make sure that we got a proper match.

t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;

This is to recalculate the hash value for pattern,first by removing the left most character and then adding the new character from the text.

Section 39.3: Analysis of Linear search (Worst, Average and Best Cases)

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/* Driver program to test above functions*/

```
int main()
```

```
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n)

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \theta(n)$$

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1) Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case

occurs when the array is sorted in the same order as output.

Section 39.4: Binary Search: On Sorted Numbers

It's easiest to show a binary search on numbers using pseudo-code

```
int array[1000] = { sorted list of numbers };
int N = 100;  // number of entries in search space;
int high, low, mid; // our temporaries
int x; // value to search for

low = 0;
high = N -1;
while(low < high)
{
    mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    else
        high = mid;
}
if(array[low] == x)
    // found, index is low
else
    // not found
```

Do not attempt to return early by comparing array[mid] to x for equality. The extra comparison can only slow the code down. Note you need to add one to low to avoid becoming trapped by integer division always rounding down.

Interestingly, the above version of binary search allows you to find the smallest occurrence of x in the array. If the array contains duplicates of x, the algorithm can be modified slightly in order for it to return the largest occurrence of x by simply adding to the if conditional:

```
while(low < high)
    {
        mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        else
            high = mid;
    }
```

Note that instead of doing `mid = (low + high) / 2`, it may also be a good idea to try `mid = low + ((high - low) / 2)` for implementations such as Java implementations to lower the risk of getting an overflow for really large inputs.

Section 39.5: Linear search

Linear search is a simple algorithm. It loops through items until the query has been found, which makes it a linear algorithm - the complexity is O(n), where n is the number of items to go through.

Why O(n)? In worst-case scenario, you have to go through all of the n items.

It can be compared to looking for a book in a stack of books - you go through them all until you find the one that you want.

Below is a Python implementation:

```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #returns True
```

Chapter 40: Substring Search

Section 40.1: Introduction To Knuth-Morris-Pratt (KMP) Algorithm

Suppose that we have a *text* and a *pattern*. We need to determine if the pattern exists in the text or not. For example:

```
+-------+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+
|  Text | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| Index   | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| Pattern | b | c | g | l |
+---------+---+---+---+---+
```

This *pattern* does exist in the *text*. So our substring search should return **3**, the index of the position from which this *pattern* starts. So how does our brute force substring search procedure work?

What we usually do is: we start from the **0th** index of the *text* and the **0th** index of our *pattern and we compare **Text[0]** with **Pattern[0]**. Since they are not a match, we go to the next index of our *text* and we compare **Text[1]** with **Pattern[0]**. Since this is a match, we increment the index of our *pattern* and the index of the *Text* also. We compare **Text[2]** with **Pattern[1]**. They are also a match. Following the same procedure stated before, we now compare **Text[3]** with **Pattern[2]**. As they do not match, we start from the next position where we started finding the match. That is index **2** of the *Text*. We compare **Text[2]** with **Pattern[0]**. They don't match. Then incrementing index of the *Text*, we compare **Text[3]** with **Pattern[0]**. They match. Again **Text[4]** and **Pattern[1]** match, **Text[5]** and **Pattern[2]** match and **Text[6]** and **Pattern[3]** match. Since we've reached the end of our *Pattern*, we now return the index from which our match started, that is **3**. If our *pattern* was: `bcgll`, that means if the *pattern* didn't exist in our *text*, our search should return exception or **-1** or any other predefined value. We can clearly see that, in the worst case, this algorithm would take $O(mn)$ time where **m** is the length of the *Text* and **n** is the length of the *Pattern*. How do we reduce this time complexity? This is where KMP Substring Search Algorithm comes into the picture.

The Knuth-Morris-Pratt String Searching Algorithm or KMP Algorithm searches for occurrences of a "Pattern" within a main "Text" by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The algorithm was conceived in 1970 by Donuld Knuth and Vaughan Pratt and independently by James H. Morris. The trio published it jointly in 1977.

Let's extend our example *Text* and *Pattern* for better understanding:

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Index |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Text |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
```

```
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

At first, our *Text* and *Pattern* matches till index **2**. **Text[3]** and **Pattern[3]** doesn't match. So our aim is to not go backwards in this *Text*, that is, in case of a mismatch, we don't want our matching to begin again from the position that we started matching with. To achieve that, we'll look for a **suffix** in our *Pattern* right before our mismatch occurred (substring **abc**), which is also a **prefix** of the substring of our *Pattern*. For our example, since all the characters are unique, there is no suffix, that is the prefix of our matched substring. So what that means is, our next comparison will start from index **0**. Hold on for a bit, you'll understand why we did this. Next, we compare **Text[3]** with **Pattern[0]** and it doesn't match. After that, for *Text* from index **4** to index **9** and for *Pattern* from index **0** to index **5**, we find a match. We find a mismatch in **Text[10]** and **Pattern[6]**. So we take the substring from *Pattern* right before the point where mismatch occurs (substring **abcdabc**), we check for a suffix, that is also a prefix of this substring. We can see here **ab** is both the suffix and prefix of this substring. What that means is, since we've matched until **Text[10]**, the characters right before the mismatch is **ab**. What we can infer from it is that since **ab** is also a prefix of the substring we took, we don't have to check **ab** again and the next check can start from **Text[10]** and **Pattern[2]**. We didn't have to look back to the whole *Text*, we can start directly from where our mismatch occurred. Now we check **Text[10]** and **Pattern[2]**, since it's a mismatch, and the substring before mismatch (**abc**) doesn't contain a suffix which is also a prefix, we check **Text[10]** and **Pattern[0]**, they don't match. After that for *Text* from index **11** to index **17** and for *Pattern* from index **0** to index **6**. We find a mismatch in **Text[18]** and **Pattern[7]**. So again we check the substring before mismatch (substring **abcdabc**) and find **abc** is both the suffix and the prefix. So since we matched till **Pattern[7]**, **abc** must be before **Text[18]**. That means, we don't need to compare until **Text[17]** and our comparison will start from **Text[18]** and **Pattern[3]**. Thus we will find a match and we'll return **15** which is our starting index of the match. This is how our KMP Substring Search works using suffix and prefix information.

Now, how do we efficiently compute if suffix is same as prefix and at what point to start the check if there is a mismatch of character between *Text* and *Pattern*. Let's take a look at an example:

```
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

We'll generate an array containing the required information. Let's call the array **S**. The size of the array will be same as the length of the pattern. Since the first letter of the *Pattern* can't be the suffix of any prefix, we'll put **S[0]** = **0**. We take **i** = **1** and **j** = **0** at first. At each step we compare **Pattern[i]** and **Pattern[j]** and increment **i**. If there is a match we put **S[i]** = **j** + **1** and increment **j**, if there is a mismatch, we check the previous value position of **j** (if available) and set **j** = **S[j-1]** (if **j** is not equal to **0**), we keep doing this until **S[j]** doesn't match with **S[i]** or **j** doesn't become **0**. For the later one, we put **S[i]** = **0**. For our example:

```
            j   i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Pattern[j] and **Pattern[i]** don't match, so we increment **i** and since **j** is **0**, we don't check the previous value and put **Pattern[i]** = **0**. If we keep incrementing **i**, for **i** = **4**, we'll get a match, so we put **S[i]** = **S[4]** = **j** + **1** = **0** + **1** = **1** and

increment **j** and **i**. Our array will look like:

```
               j               i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

Since **Pattern[1]** and **Pattern[5]** is a match, we put **S[i]** = **S[5]** = **j** + **1** = **1** + **1** = **2**. If we continue, we'll find a mismatch for **j** = **3** and **i** = **7**. Since **j** is not equal to **0**, we put **j** = **S[j-1]**. And we'll compare the characters at **i** and **j** are same or not, since they are same, we'll put **S[i]** = **j** + 1. Our completed array will look like:

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

This is our required array. Here a nonzero-value of **S[i]** means there is a **S[i]** length suffix same as the prefix in that substring (substring from **0** to **i**) and the next comparison will start from **S[i]** + **1** position of the *Pattern*. Our algorithm to generate the array would look like:

```
Procedure GenerateSuffixArray(Pattern):
i := 1
j := 0
n := Pattern.length
while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i] := j + 1
        j := j + 1
        i := i + 1
    else
        if j is not equal to 0
            j := S[j-1]
        else
            S[i] := 0
            i := i + 1
        end if
    end if
end while
```

The time complexity to build this array is `O(n)` and the space complexity is also `O(n)`. To make sure if you have completely understood the algorithm, try to generate an array for pattern `aabaabaa` and check if the result matches with this one.

Now let's do a substring search using the following example:

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   Text  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 |
```

```
+---------+---+---+---+---+---+---+
| Pattern | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

We have a *Text*, a *Pattern* and a pre-calculated array *S* using our logic defined before. We compare **Text[0]** and **Pattern[0]** and they are same. **Text[1]** and **Pattern[1]** are same. **Text[2]** and **Pattern[2]** are not same. We check the value at the position right before the mismatch. Since **S[1]** is **0**, there is no suffix that is same as the prefix in our substring and our comparison starts at position **S[1]**, which is **0**. So **Pattern[0]** is not same as **Text[2]**, so we move on. **Text[3]** is same as **Pattern[0]** and there is a match till **Text[8]** and **Pattern[5]**. We go one step back in the **S** array and find **2**. So this means there is a prefix of length **2** which is also the suffix of this substring (**abcab)** which is **ab**. That also means that there is an **ab** before **Text[8]**. So we can safely ignore **Pattern[0]** and **Pattern[1]** and start our next comparison from **Pattern[2]** and **Text[8]**. If we continue, we'll find the *Pattern* in the *Text*. Our procedure will look like:

```
Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)
m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
    if j is equal to n
        Return (j-i)
    else if i < m and Pattern[j] is not equal t Text[i]
        if j is not equal to 0
            j = S[j-1]
        else
            i := i + 1
        end if
    end if
end while
Return -1
```

The time complexity of this algorithm apart from the Suffix Array Calculation is `O(m)`. Since *GenerateSuffixArray* takes `O(n)`, the total time complexity of KMP Algorithm is: `O(m+n)`.

PS: If you want to find multiple occurrences of *Pattern* in the *Text*, instead of returning the value, print it/store it and set `j := S[j-1]`. Also keep a `flag` to track whether you have found any occurrence or not and handle it accordingly.

Section 40.2: Introduction to Rabin-Karp Algorithm

Rabin-Karp Algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any one of a set of pattern strings in a text.

A substring of a string is another string that occurs in. For example, *ver* is a substring of *stackoverflow*. Not to be confused with subsequence because *cover* is a subsequence of the same string. In other words, any subset of consecutive letters in a string is a substring of the given string.

In Rabin-Karp algorithm, we'll generate a hash of our *pattern* that we are looking for & check if the rolling hash of our *text* matches the *pattern* or not. If it doesn't match, we can guarantee that the *pattern* **doesn't exist** in the *text*.

However, if it does match, the *pattern* **can** be present in the *text*. Let's look at an example:

Let's say we have a text: **yeminsajid** and we want to find out if the pattern **nsa** exists in the text. To calculate the hash and rolling hash, we'll need to use a prime number. This can be any prime number. Let's take **prime** = **11** for this example. We'll determine hash value using this formula:

```
(1st letter) X (prime) + (2nd letter) X (prime)¹ + (3rd letter) X (prime)² X + ......
```

We'll denote:

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

The hash value of **nsa** will be:

```
14 X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

Now we find the rolling-hash of our text. If the rolling hash matches with the hash value of our pattern, we'll check if the strings match or not. Since our pattern has **3** letters, we'll take 1st **3** letters **yem** from our text and calculate hash value. We get:

```
25 X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

This value doesn't match with our pattern's hash value. So the string doesn't exists here. Now we need to consider the next step. To calculate the hash value of our next string **emi**. We can calculate this using our formula. But that would be rather trivial and cost us more. Instead, we use another technique.

- We subtract the value of the **First Letter of Previous String** from our current hash value. In this case, **y**. We get, `1653 - 25 = 1628`.
- We divide the difference with our **prime**, which is **11** for this example. We get, `1628 / 11 = 148`.
- We add **new letter X (prime)□⁻¹**, where **m** is the length of the pattern, with the quotient, which is **i** = **9**. We get, `148 + 9 X 11² = 1237`.

The new hash value is not equal to our patterns hash value. Moving on, for **n** we get:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
New String: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

It doesn't match. After that, for **s**, we get:

```
Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
New String: "ins"
1806 - 13 = 1793
1793 / 11 = 163
```

```
163 + 19 X 11² = 2462
```

It doesn't match. Next, for **a**, we get:

```
Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
New String: "nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

It's a match! Now we compare our pattern with the current string. Since both the strings match, the substring exists in this string. And we return the starting position of our substring.

The pseudo-code will be:

Hash Calculation:

```
Procedure Calculate-Hash(String, Prime, x):
hash := 0                                   // Here x denotes the length to be considered
for m from 1 to x                           // to find the hash value
    hash := hash + (Value of String[m])□⁻¹
end for
Return hash
```

Hash Recalculation:

```
Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]  //here Curr denotes First Letter of Previous String
Hash := Hash / Prime
m := String.length
New := Curr + m - 1
Hash := Hash + (Value of String[New])□⁻¹
Return Hash
```

String Match:

```
Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m - 1
    if Text[i] is not equal to Pattern[i]
        Return false
    end if
end for
Return true
```

Rabin-Karp:

```
Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length - m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for
```

```
Return -1
```

If the algorithm doesn't find any match, it simply returns **-1**.

This algorithm is used in detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical here. Again, **Knuth-Morris-Pratt algorithm** or **Boyer-Moore String Search algorithm** is faster single pattern string searching algorithm, than **Rabin-Karp**. However, it is an algorithm of choice for multiple pattern search. If we want to find any of the large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin-Karp algorithm.

For text of length **n** and **p** patterns of combined length **m**, its average and best case running time is **O(n+m)** in space **O(p)**, but its worst-case time is **O(nm)**.

Section 40.3: Python Implementation of KMP algorithm

Haystack: The string in which given pattern needs to be searched.
Needle: The pattern to be searched.

Time complexity: Search portion (strstr method) has the complexity O(n) where `n` is the length of haystack but as needle is also pre parsed for building prefix table O(m) is required for building prefix table where `m` is the length of the needle.
Therefore, overall time complexity for KMP is **O(n+m)**
Space complexity: **O(m)** because of prefix table on needle.

Note: Following implementation returns the start position of match in haystack (if there is a match) else returns -1, for edge cases like if needle/haystack is an empty string or needle is not found in haystack.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
        prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
                prefix_table[delimeter] = delimeter - j + 1
                break
            j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
    # m: denoting the position within S where the prospective match for W begins
    # i: denoting the index of the currently considered character in W.
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
```

```
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

Section 40.4: KMP Algorithm in C

Given a text *txt* and a pattern *pat*, the objective of this program will be to print all the occurance of *pat* in *txt*.

Examples:

Input:

```
  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"
```

output:

```
Pattern found at index 10
```

Input:

```
  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"
```

output:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

C Language Implementation:

```
// C program for implementation of KMP pattern searching
// algorithm
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
```

```
    // values for pattern
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // length of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // This is tricky. Consider the example
           // AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
```

```
            }
            else // if (len == 0)
            {
              lps[i] = 0;
              i++;
            }
          }
      }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Output:

```
Found pattern at index 10
```

Reference:

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

Chapter 41: Breadth-First Search

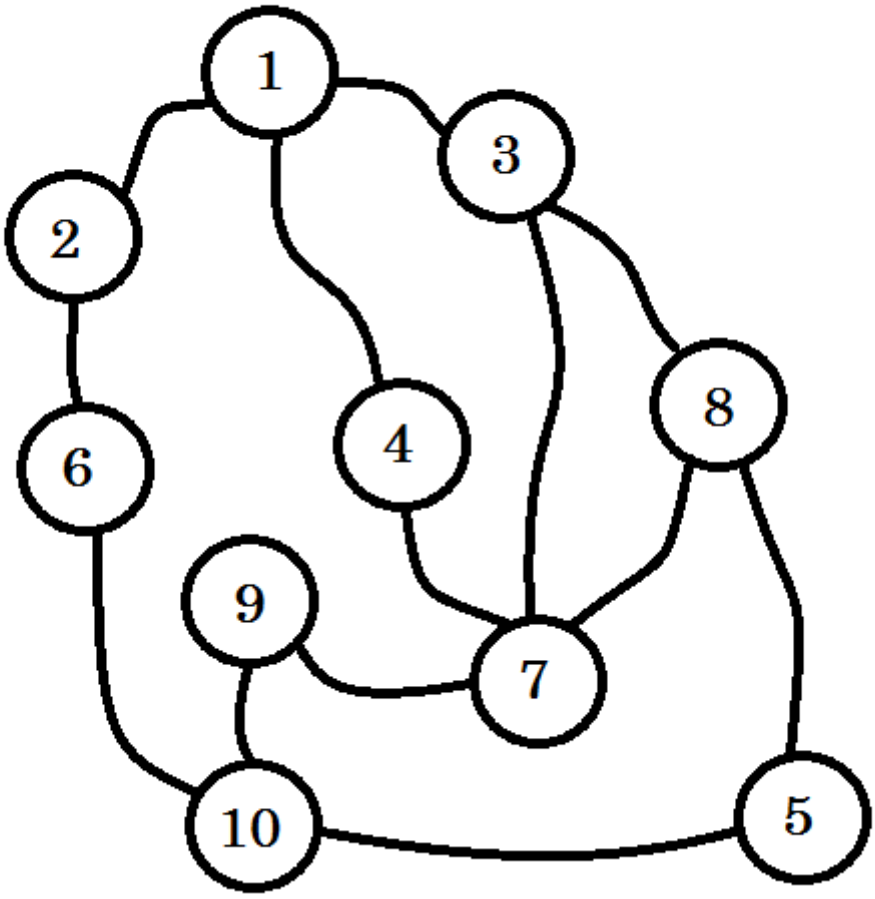
Section 41.1: Finding the Shortest Path from Source to other Nodes

Breadth-first-search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. BFS was invented in the late 1950s by Edward Forrest Moore, who used it to find the shortest path out of a maze and discovered independently by C. Y. Lee as a wire routing algorithm in 1961.

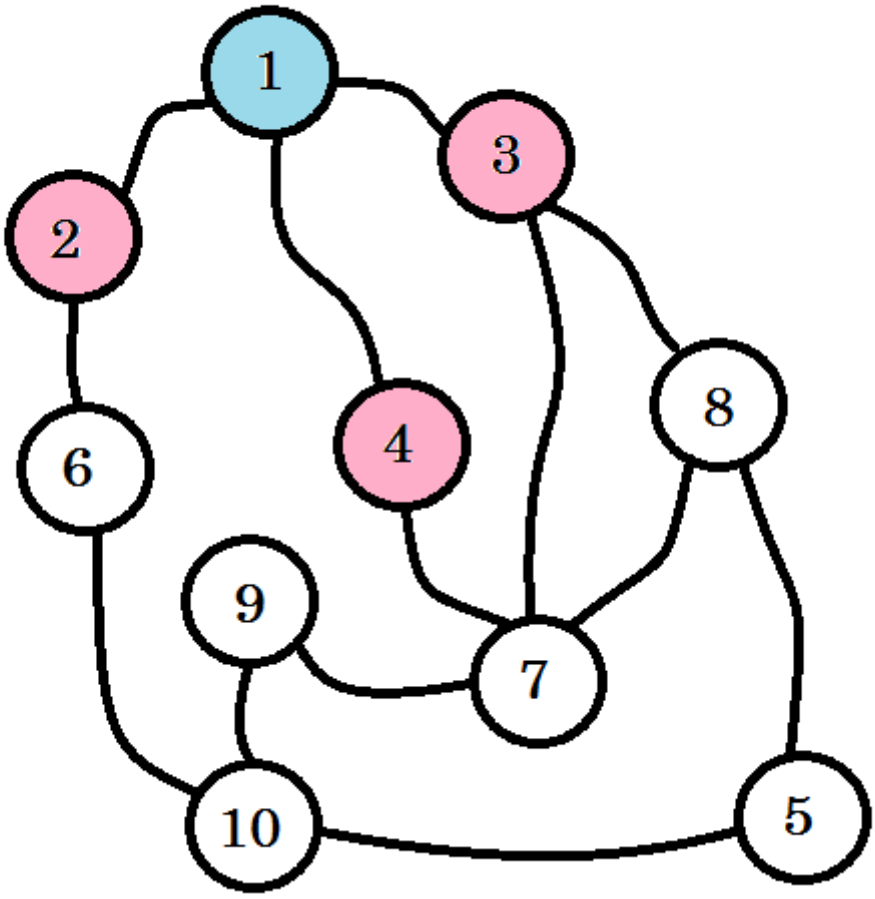
The processes of BFS algorithm works under these assumptions:

1. We won't traverse any node more than once.
2. Source node or the node that we're starting from is situated in level 0.
3. The nodes we can directly reach from source node are level 1 nodes, the nodes we can directly reach from level 1 nodes are level 2 nodes and so on.
4. The level denotes the distance of the shortest path from the source.

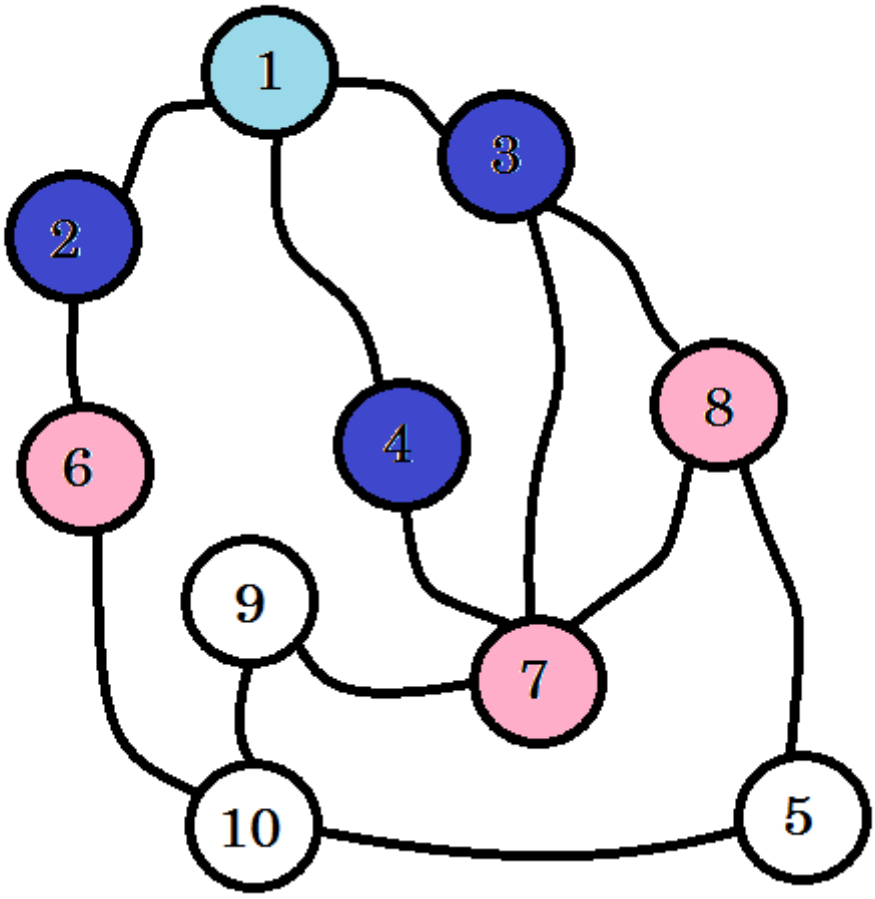
Let's see an example:



Let's assume this graph represents connection between multiple cities, where each node denotes a city and an edge between two nodes denote there is a road linking them. We want to go from **node 1** to **node 10**. So **node 1** is our **source**, which is **level 0**. We mark **node 1** as visited. We can go to **node 2**, **node 3** and **node 4** from here. So they'll be **level (0+1)** = **level 1** nodes. Now we'll mark them as visited and work with them.

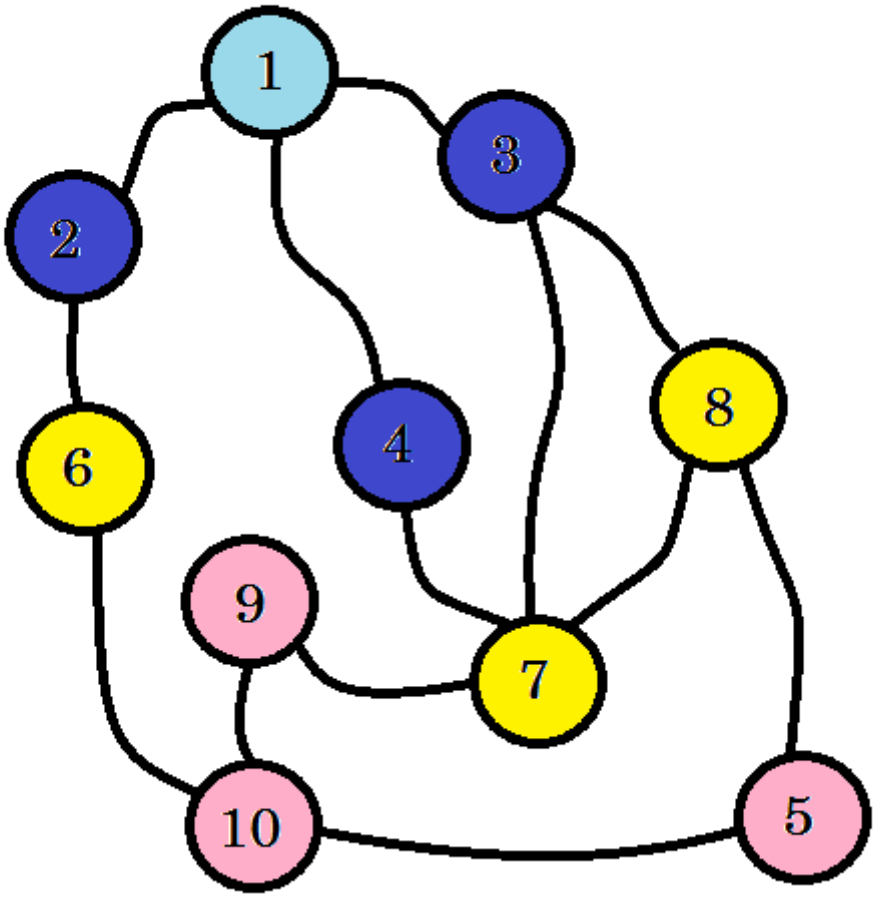


The colored nodes are visited. The nodes that we're currently working with will be marked with pink. We won't visit the same node twice. From **node 2**, **node 3** and **node 4**, we can go to **node 6, node 7** and **node 8**. Let's mark them as visited. The level of these nodes will be **level (1+1)** = **level 2**.

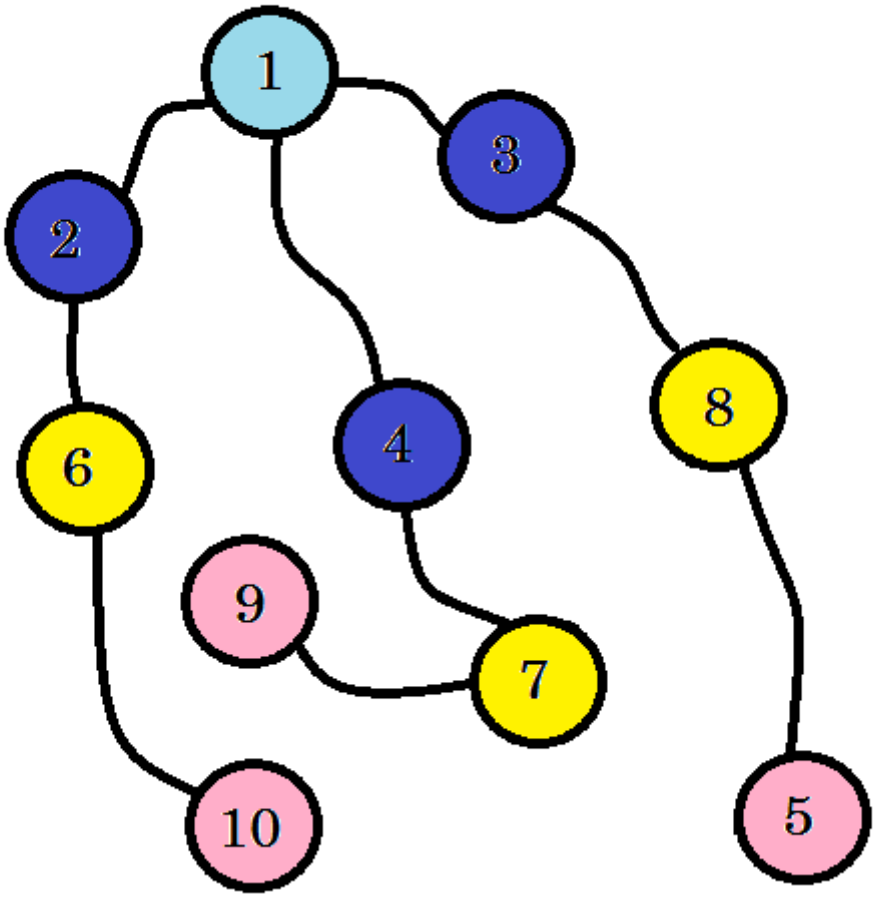


If you haven't noticed, the level of nodes simply denote the shortest path distance from the **source**. For example: we've found **node 8** on **level 2**. So the distance from **source** to **node 8** is **2**.

We didn't yet reach our target node, that is **node 10**. So let's visit the next nodes. we can directly go to from **node 6**, **node 7** and **node 8**.



We can see that, we found **node 10** at **level 3**. So the shortest path from **source** to **node 10** is **3.** We searched the graph level by level and found the shortest path. Now let's erase the edges that we didn't use:



After removing the edges that we didn't use, we get a tree called BFS tree. This tree shows the shortest path from **source** to all other nodes.

So our task will be, to go from **source** to **level 1** nodes. Then from **level 1** to **level 2** nodes and so on until we reach our destination. We can use *queue* to store the nodes that we are going to process. That is, for each node we're going to work with, we'll push all other nodes that can be directly traversed and not yet traversed in the queue.

The simulation of our example:

First we push the source in the queue. Our queue will look like:

```
 front
+-----+
|  1  |
+-----+
```

The level of **node 1** will be 0. **level[1]** = **0**. Now we start our BFS. At first, we pop a node from our queue. We get **node 1**. We can go to **node 4**, **node 3** and **node 2** from this one. We've reached these nodes from **node 1**. So **level[4]** = **level[3]** = **level[2]** = **level[1]** + **1** = **1**. Now we mark them as visited and push them in the queue.

```
                  front
+-----+  +-----+  +-----+
|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

Now we pop **node 4** and work with it. We can go to **node 7** from **node 4**. **level[7]** = **level[4]** + **1** = **2**. We mark **node 7** as visited and push it in the queue.

```
                  front
+-----+  +-----+  +-----+
|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

From **node 3**, we can go to **node 7** and **node 8**. Since we've already marked **node 7** as visited, we mark **node 8** as visited, we change **level[8]** = **level[3]** + **1** = **2**. We push **node 8** in the queue.

```
                  front
+-----+  +-----+  +-----+
|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

This process will continue till we reach our destination or the queue becomes empty. The **level** array will provide us with the distance of the shortest path from **source**. We can initialize **level** array with *infinity* value, which will mark that the nodes are not yet visited. Our pseudo-code will be:

```
Procedure BFS(Graph, source):
Q = queue();
level[] = infinity
level[source] := 0
Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
        end if
    end for
end while
Return level
```

By iterating through the **level** array, we can find out the distance of each node from source. For example: the distance of **node 10** from **source** will be stored in **level[10]**.

Sometimes we might need to print not only the shortest distance, but also the path via which we can go to our destined node from the **source**. For this we need to keep a **parent** array. **parent[source]** will be NULL. For each update in **level** array, we'll simply add `parent[v] := u` in our pseudo code inside the for loop. After finishing BFS, to find the path, we'll traverse back the **parent** array until we reach **source** which will be denoted by NULL value. The pseudo-code will be:

```
Procedure PrintPath(u):  //recursive
if parent[u] is not equal to null
    PrintPath(parent[u])
end if
print -> u
```

```
Procedure PrintPath(u):   //iterative
S =  Stack()
while parent[u] is not equal to null
    S.push(u)
    u := parent[u]
end while
while S is not empty
    print -> S.pop
end while
```

Complexity:

We've visited every node once and every edges once. So the complexity will be **O(V + E)** where **V** is the number of nodes and **E** is the number of edges.

Section 41.2: Finding Shortest Path from Source in a 2D graph

Most of the time, we'll need to find out the shortest path from single source to all other nodes or a specific node in a 2D graph. Say for example: we want to find out how many moves are required for a knight to reach a certain square in a chessboard, or we have an array where some cells are blocked, we have to find out the shortest path from one cell to another. We can move only horizontally and vertically. Even diagonal moves can be possible too. For these cases, we can convert the squares or cells in nodes and solve these problems easily using BFS. Now our **visited**, **parent** and **level** will be 2D arrays. For each node, we'll consider all possible moves. To find the distance to a specific node, we'll also check whether we have reached our destination.

There will be one additional thing called direction array. This will simply store the all possible combinations of directions we can go to. Let's say, for horizontal and vertical moves, our direction arrays will be:

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

Here *dx* represents move in x-axis and *dy* represents move in y-axis. Again this part is optional. You can also write all the possible combinations separately. But it's easier to handle it using direction array. There can be more and even different combinations for diagonal moves or knight moves.

The additional part we need to keep in mind is:

- If any of the cell is blocked, for every possible moves, we'll check if the cell is blocked or not.
- We'll also check if we have gone out of bounds, that is we've crossed the array boundaries.
- The number of rows and columns will be given.

Our pseudo-code will be:

```
Procedure BFS2D(Graph, blocksign, row, column):
for i from 1 to row
    for j from 1 to column
        visited[i][j] := false
    end for
end for
visited[source.x][source.y] := true
level[source.x][source.y] := 0
Q = queue()
Q.push(source)
m := dx.size
while Q is not empty
    top := Q.pop
    for i from 1 to m
        temp.x := top.x + dx[i]
        temp.y := top.y + dy[i]
        if temp is inside the row and column and top doesn't equal to blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
```

```
        end if
    end for
end while
Return level
```

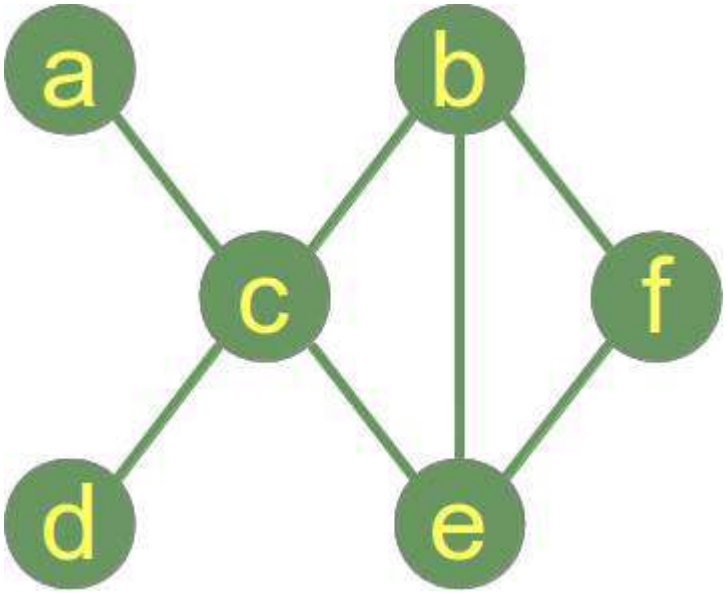
As we have discussed earlier, BFS only works for unweighted graphs. For weighted graphs, we'll need Dijkstra's algorithm. For negative edge cycles, we need Bellman-Ford's algorithm. Again this algorithm is single source shortest path algorithm. If we need to find out distance from each nodes to all other nodes, we'll need Floyd-Warshall's algorithm.

Section 41.3: Connected Components Of Undirected Graph Using BFS

BFS can be used to find the connected components of an undirected graph. We can also find if the given graph is connected or not. Our subsequent discussion assumes we are dealing with undirected graphs.The definition of a connected graph is:

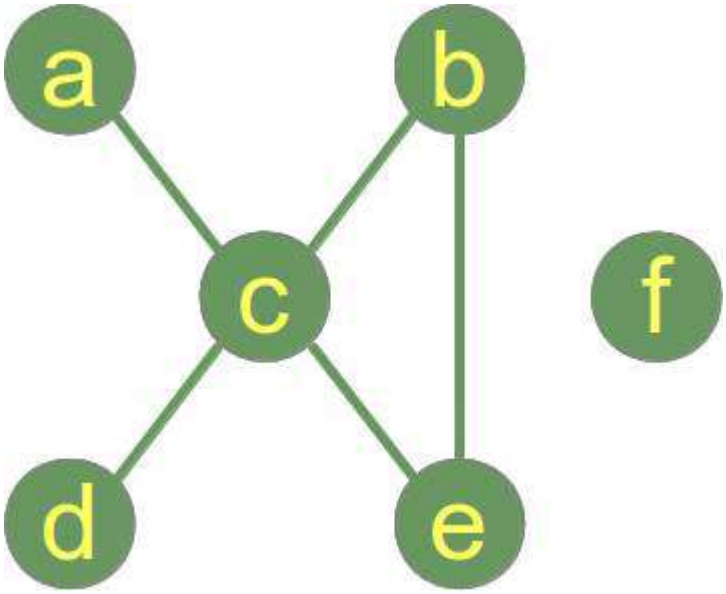
> A graph is connected if there is a path between every pair of vertices.

Following is a **connected graph**.



Following graph is **not connected** and has 2 connected components:

1. Connected Component 1: {a,b,c,d,e}
2. Connected Component 2: {f}



BFS is a graph traversal algorithm. So starting from a random source node, if on termination of algorithm, all nodes are visited, then the graph is connected,otherwise it is not connected.

PseudoCode for the algorithm.

```
boolean isConnected(Graph g)
{
 BFS(v)//v is a random source node.
 if(allVisited(g))
 {
  return true;
 }
 else return false;
}
```

C implementation for finding the whether an undirected graph is connected or not:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//Queue node depicts a single Queue element
//It is NOT a graph node.
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//array that keeps track of visited vertices.

int main()
```

```
{
    int n,e;//n is number of vertices, e is number of edges.
    int i,j;
    char **graph;//adjacency matrix

    printf("Enter number of vertices:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "Please enter a valid positive integer from 1 to %d",MAXVERTICES);
     return -1;
    }

    graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
        graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//initially all vertices are not visited.
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("enter number of edges and then enter them in pairs:");
    scanf("%d",&e);

    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("The graph is connected");
    else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if(Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int deque()
{
```

```
    if(Qfront == NULL)
    {
        printf("Q is empty , returning -1\n");
        return -1;
    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
            Qfront = Qfront->next;
            Qrear = NULL;
        }
        else
            Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //let random source vertex be vertex 0;
    BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 implies false;

    return 1;//1 implies true;
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
    visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

For Finding all the Connected components of an undirected graph, we only need to add 2 lines of code to the BFS function. The idea is to call BFS function until all vertices are visited.

The lines to be added are:

```
printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function
```

AND

```
printf("%d ",vertex+1);
add this as first line of while loop in BFS
```

and we define the following function:

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```

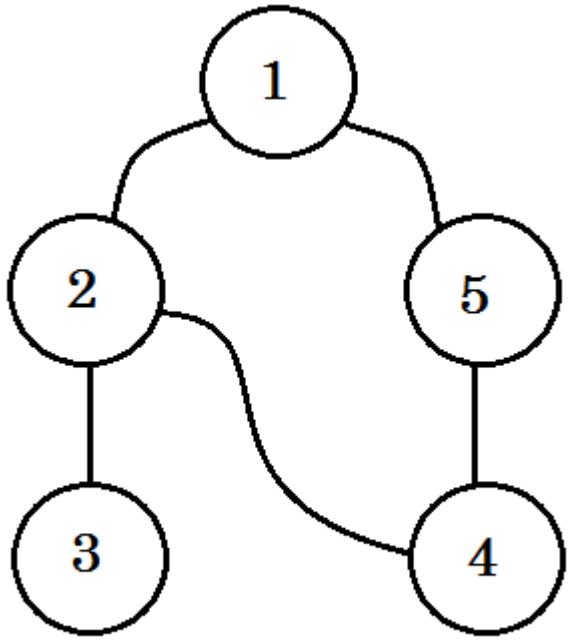
Chapter 42: Depth First Search

Section 42.1: Introduction To Depth-First Search

Depth-first search is an algorithm for traversing or searching tree or graph data structures. One starts at the root and explores as far as possible along each branch before backtracking. A version of depth-first search was investigated in the 19th century French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

Depth-first search is a systematic way to find all the vertices reachable from a source vertex. Like breadth-first search, DFS traverse a connected component of a given graph and defines a spanning tree. The basic idea of depth-first search is methodically exploring every edge. We start over from a different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it (unlike BFS, which puts a vertex on a queue so that it explores from it later).

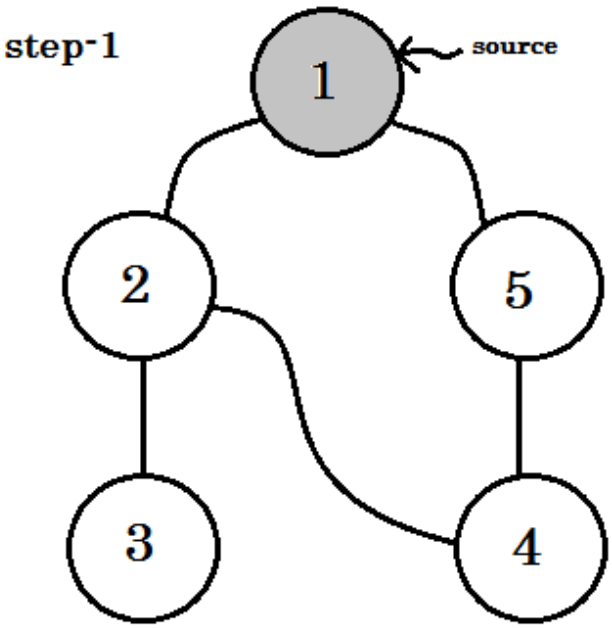
Let's look at an example. We'll traverse this graph:

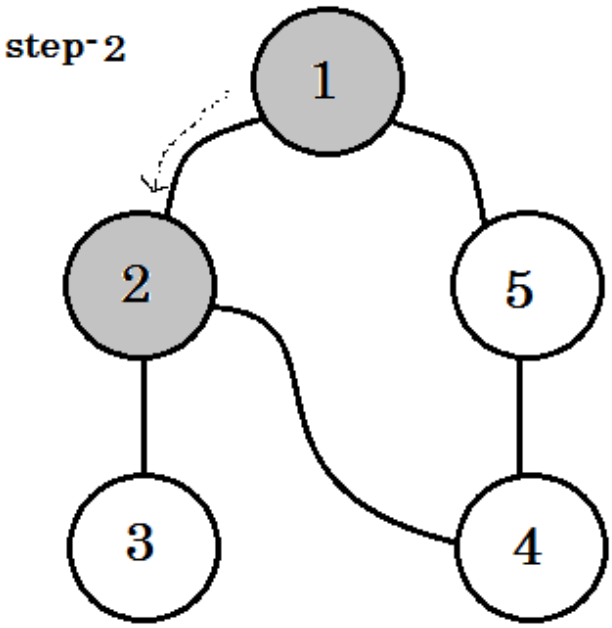


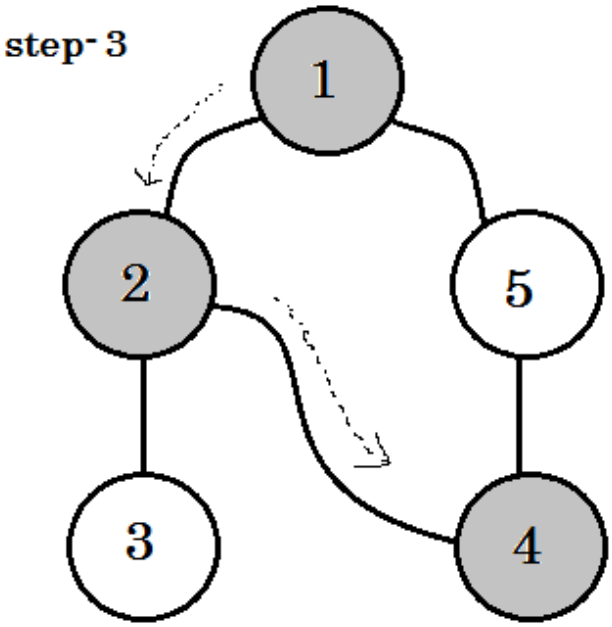
We'll traverse the graph following these rules:

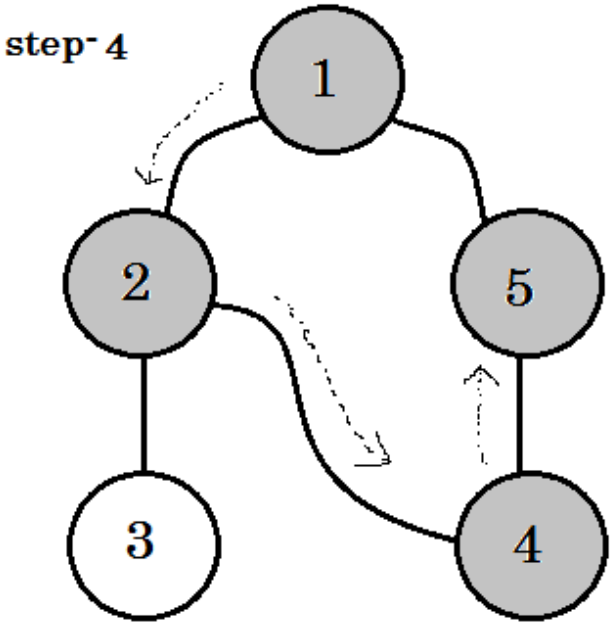
- We'll start from the source.
- No node will be visited twice.
- The nodes we didn't visit yet, will be colored white.
- The node we visited, but didn't visit all of its child nodes, will be colored grey.
- Completely traversed nodes will be colored black.

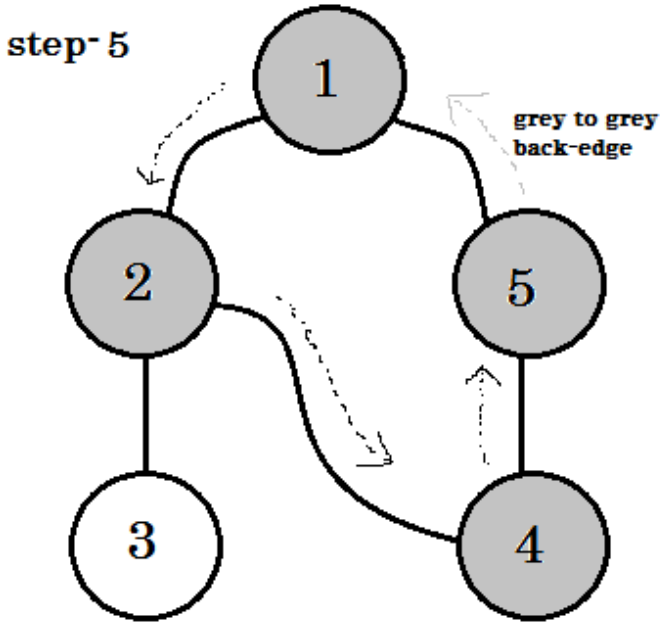
Let's look at it step by step:

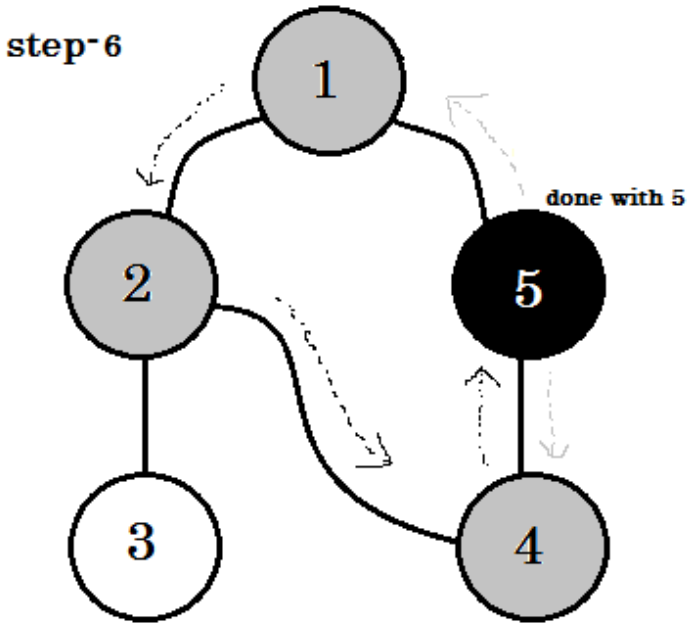

step-1
source
1
2
5
3
4

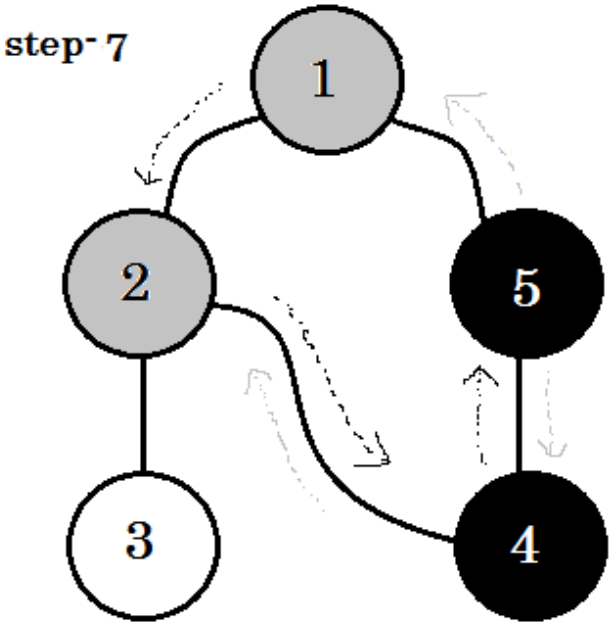

step-2
1
2
5
3
4

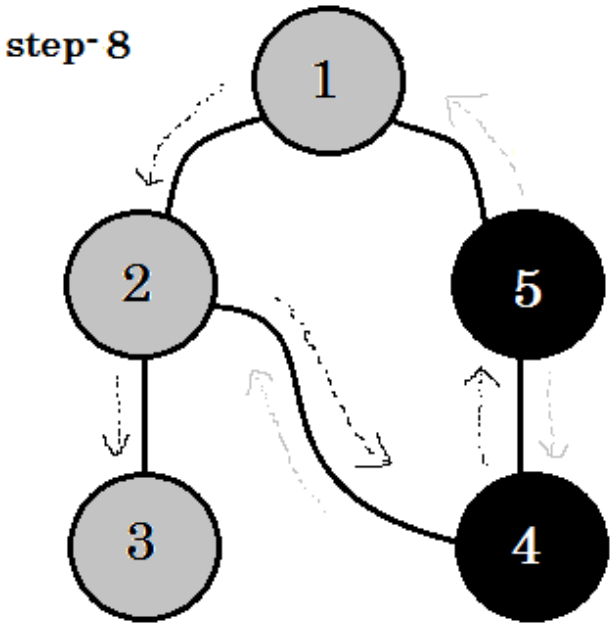

step-3
1
2
5
3
4

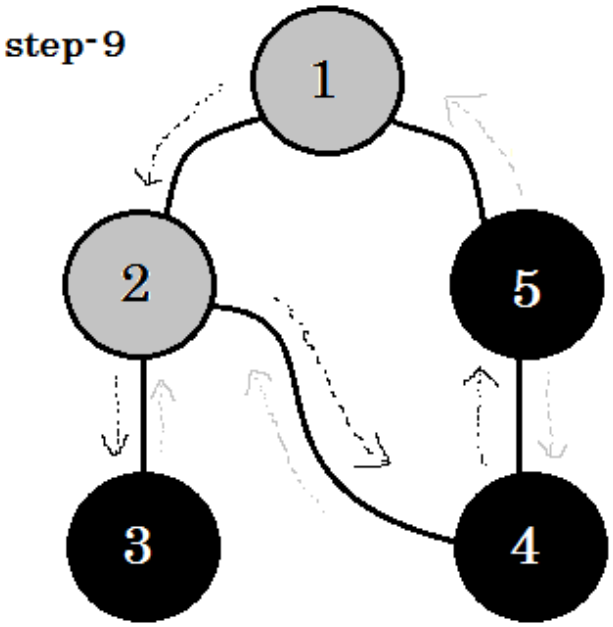

step-4
1
2
5
3
4

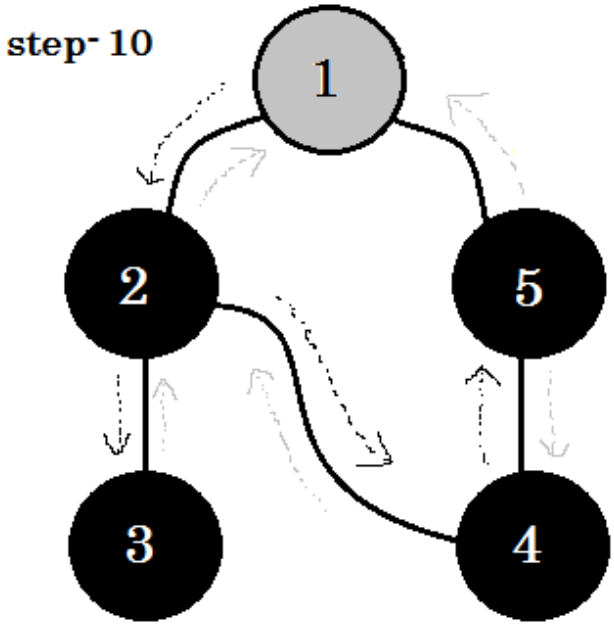

step- 5
1
grey to grey
back-edge
2
5
3
4

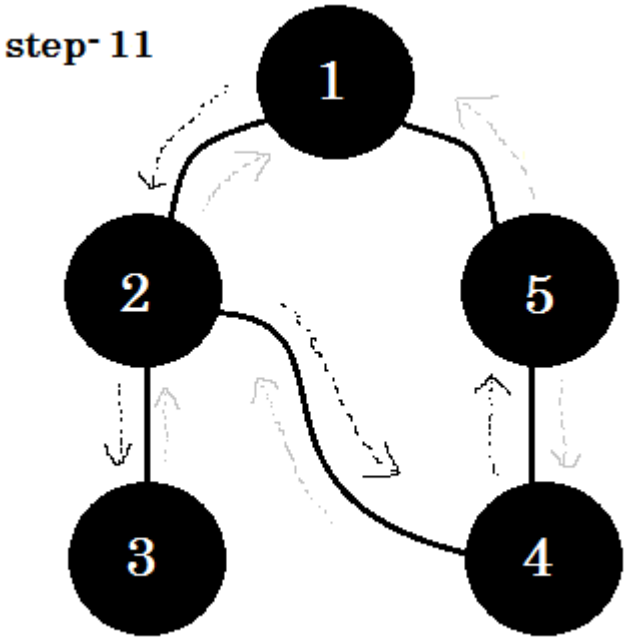

step- 6
1
done with 5
2
5
3
4


step- 7
1
2
5
3
4


step- 8
1
2
5
3
4







We can see one important keyword. That is **backedge**. You can see. **5-1** is called backedge. This is because, we're not yet done with **node-1**, so going from another node to **node-1** means there's a cycle in the graph. In DFS, if we can go from one gray node to another, we can be certain that the graph has a cycle. This is one of the ways of detecting cycle in a graph. Depending on **source** node and the order of the nodes we visit, we can find out any edge in a cycle as **backedge**. For example: if we went to **5** from **1** first, we'd have found out **2-1** as backedge.

The edge that we take to go from gray node to white node are called **tree edge**. If we only keep the **tree edge**'s and remove others, we'll get **DFS tree**.

In undirected graph, if we can visit a already visited node, that must be a **backedge**. But for directed graphs, we must check the colors. *If and only if we can go from one gray node to another gray node, that is called a backedge*.

In DFS, we can also keep timestamps for each node, which can be used in many ways (e.g.: Topological Sort).

1. When a node **v** is changed from white to gray the time is recorded in **d[v]**.

2. When a node **v** is changed from gray to black the time is recorded in **f[v]**.

Here **d[]** means *discovery time* and **f[]** means *finishing time*. Our pesudo-code will look like:

```
Procedure DFS(G):
for each node u in V[G]
    color[u] := white
    parent[u] := NULL
end for
time := 0
for each node u in V[G]
    if color[u] == white
        DFS-Visit(u)
    end if
end for

Procedure DFS-Visit(u):
color[u] := gray
time := time + 1
d[u] := time
for each node v adjacent to u
    if color[v] == white
        parent[v] := u
        DFS-Visit(v)
    end if
end for
color[u] := black
time := time + 1
f[u] := time
```

Complexity:

Each nodes and edges are visited once. So the complexity of DFS is **O(V+E)**, where **V** denotes the number of nodes and **E** denotes the number of edges.

Applications of Depth First Search:

- Finding all pair shortest path in an undirected graph.
- Detecting cycle in a graph.
- Path finding.
- Topological Sort.
- Testing if a graph is bipartite.
- Finding Strongly Connected Component.
- Solving puzzles with one solution.

Chapter 43: Hash Functions

Section 43.1: Hash codes for common types in C#

The hash codes produced by `GetHashCode()` method for built-in and common C# types from the `System` namespace are shown below.

Boolean

1 if value is true, 0 otherwise.

Byte, UInt16, Int32, UInt32, Single

Value (if necessary casted to Int32).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

Char

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64, Double

Xor between lower and upper 32 bits of 64 bit number

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64, DateTime, TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Decimal

```
((((int *)&dbl)[0]) & 0xFFFFFFF0) ^ ((int *)&dbl)[1];
```

Object

```
RuntimeHelpers.GetHashCode(this);
```

The default implementation is used sync block index.

String

Hash code computation depends on the platform type (Win32 or Win64), feature of using randomized string hashing, Debug / Release mode. In case of Win64 platform:

```
int hash1 = 5381;
int hash2 = hash1;
int c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```

```
return hash1 + (hash2 * 1566083941);
```

ValueType

The first non-static field is look for and get it's hashcode. If the type has no non-static fields, the hashcode of the type returns. The hashcode of a static member can't be taken because if that member is of the same type as the original type, the calculating ends up in an infinite loop.

Nullable<T>

```
return hasValue ? value.GetHashCode() : 0;
```

Array

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

References

- GitHub .Net Core CLR

Section 43.2: Introduction to hash functions

Hash function `h()` is an arbitrary function which mapped data `x ∈ X` of arbitrary size to value `y ∈ Y` of fixed size: `y = h(x)`. Good hash functions have follows restrictions:

- hash functions behave likes uniform distribution
- hash functions is deterministic. `h(x)` should always return the same value for a given `x`
- fast calculating (has runtime O(1))

In general case size of hash function less then size of input data: `|y| < |x|`. Hash functions are not reversible or in other words it may be collision: `∃ x1, x2 ∈ X, x1 ≠ x2: h(x1) = h(x2)`. `X` may be finite or infinite set and `Y` is finite set.

Hash functions are used in a lot of parts of computer science, for example in software engineering, cryptography, databases, networks, machine learning and so on. There are many different types of hash functions, with differing domain specific properties.

Often hash is an integer value. There are special methods in programmning languages for hash calculating. For example, in `C#` `GetHashCode()` method for all types returns `Int32` value (32 bit integer number). In `Java` every class provides `hashCode()` method which return **`int`**. Each data type has own or user defined implementations.

Hash methods

There are several approaches for determinig hash function. Without loss of generality, lets `x ∈ X = {z ∈ ℤ: z ≥ 0}` are positive integer numbers. Often `m` is prime (not too close to an exact power of 2).

Method	Hash function
Division method	`h(x) = x mod m`
Multiplication method	`h(x) = ⌊m (xA mod 1)⌋, A ∈ {z ∈ ℝ: 0 < z < 1}`

Hash table

Hash functions used in hash tables for computing index into an array of slots. Hash table is data structure for

implementing dictionaries (key-value structure). Good implemented hash tables have O(1) time for the next operations: insert, search and delete data by key. More than one keys may hash to the same slot. There are two ways for resolving collision:

1. Chaining: linked list is used for storing elements with the same hash value in slot
2. Open addressing: zero or one element is stored in each slot

The next methods are used to compute the probe sequences required for open addressing

Method	Formula
Linear probing	`h(x, i) = (h'(x) + i) mod m`
Quadratic probing	`h(x, i) = (h'(x) + c1*i + c2*i^2) mod m`
Double hashing	`h(x, i) = (h1(x) + i*h2(x)) mod m`

Where `i ∈ {0, 1, ..., m-1}`, `h'(x), h1(x), h2(x)` are auxiliary hash functions, `c1, c2` are positive auxiliary constants.

Examples

Lets `x ∈ U{1, 1000}, h = x mod m`. The next table shows the hash values in case of not prime and prime. Bolded text indicates the same hash values.

x	m = 100 (not prime)	m = 101 (prime)
723	23	16
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86
549	49	44
991	91	82
757	**57**	50
920	20	11
626	26	20
557	**57**	52
831	31	23
619	19	13

Links

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.
- Overview of Hash Tables
- Wolfram MathWorld - Hash Function

Chapter 44: Travelling Salesman

Section 44.1: Brute Force Algorithm

A path through every vertex exactly once is the same as ordering the vertex in some way. Thus, to calculate the minimum cost of travelling through every vertex exactly once, we can brute force every single one of the `N!` permutations of the numbers from `1` to `N`.

Psuedocode

```
minimum = INF
for all permutations P

    current = 0

    for i from 0 to N-2
        current = current + cost[P[i]][P[i+1]]  <- Add the cost of going from 1 vertex to the next

    current = current + cost[P[N-1]][P[0]]      <- Add the cost of going from last vertex to the
first

    if current < minimum                        <- Update minimum if necessary
        minimum = current

output minimum
```

Time Complexity

There are `N!` permutations to go through and the cost of each path is calculated in `O(N)`, thus this algorithm takes `O(N * N!)` time to output the exact answer.

Section 44.2: Dynamic Programming Algorithm

Notice that if we consider the path (in order):

```
(1,2,3,4,6,0,5,7)
```

and the path

```
(1,2,3,5,0,6,7,4)
```

The cost of going from vertex `1` to vertex `2` to vertex `3` remains the same, so why must it be recalculated? This result can be saved for later use.

Let `dp[bitmask][vertex]` represent the minimum cost of travelling through all the vertices whose corresponding bit in `bitmask` is set to `1` ending at `vertex`. For example:

```
dp[12][2]

   12   =   1 1 0 0
            ^ ^
vertices:   3 2 1 0
```

Since `12` represents `1100` in binary, `dp[12][2]` represents going through vertices `2` and `3` in the graph with the path ending at vertex 2.

Thus we can have the following algorithm (C++ implementation):

```
int cost[N][N];                         //Adjust the value of N if needed
int memo[1 << N][N];                    //Set everything here to -1
int TSP(int bitmask, int pos){
    int cost = INF;
    if (bitmask == ((1 << N) - 1)){      //All vertices have been explored
        return cost[pos][0];             //Cost to go back
    }
    if (memo[bitmask][pos] != -1){       //If this has already been computed
        return memo[bitmask][pos];       //Just return the value, no need to recompute
    }
    for (int i = 0; i < N; ++i){         //For every vertex
        if ((bitmask & (1 << i)) == 0){  //If the vertex has not been visited
            cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);   //Visit the vertex
        }
    }
    memo[bitmask][pos] = cost;           //Save the result
    return cost;
}
//Call TSP(1,0)
```

This line may be a little confusing, so lets go through it slowly:

```
cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);
```

Here, `bitmask | (1 << i)` sets the ith bit of `bitmask` to 1, which represents that the ith vertex has been visited. The `i` after the comma represents the new `pos` in that function call, which represents the new "last" vertex. `cost[pos][i]` is to add the cost of travelling from vertex `pos` to vertex `i`.

Thus, this line is to update the value of `cost` to the minimum possible value of travelling to every other vertex that has not been visited yet.

Time Complexity

The function `TSP(bitmask,pos)` has `2^N` values for `bitmask` and `N` values for `pos`. Each function takes `O(N)` time to run (the **`for`** loop). Thus this implementation takes `O(N^2 * 2^N)` time to output the exact answer.

Chapter 45: Knapsack Problem

Section 45.1: Knapsack Problem Basics

The Problem: Given a set of items where each item contains a weight and value, determine the number of each to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Pseudo code for Knapsack Problem

Given:

1. Values(array v)
2. Weights(array w)
3. Number of distinct items(n)
4. Capacity(W)

```
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

A simple implementation of the above pseudo code using Python:

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

Running the code: Save this in a file named knapSack.py

```
$ python knapSack.py
220
```

Time Complexity of the above code: `O(nW)` where n is the number of items and W is the capacity of knapsack.

Section 45.2: Solution Implemented in C#

```
public class KnapsackProblem
{
```

```
    private static int Knapsack(int w, int[] weight, int[] value, int n)
    {
        int i;
        int[,] k = new int[n + 1, w + 1];
        for (i = 0; i <= n; i++)
        {
            int b;
            for (b = 0; b <= w; b++)
            {
                if (i==0 || b==0)
                {
                    k[i, b] = 0;
                }
                else if (weight[i - 1] <= b)
                {
                    k[i, b] = Math.Max(value[i - 1] + k[i - 1, b - weight[i - 1]], k[i - 1, b]);
                }
                else
                {
                    k[i, b] = k[i - 1, b];
                }
            }
        }
        return k[n, w];
    }

    public static int Main(int nItems, int[] weights, int[] values)
    {
        int n = values.Length;
        return Knapsack(nItems, weights, values, n);
    }
}
```

Chapter 46: Equation Solving

Section 46.1: Linear Equation

There are two classes of methods for solving Linear Equations:

1. **Direct Methods**: Common characteristics of direct methods are that they transform the original equation into equivalent equations that can be solved more easily, means we get solve directly from an equation.
2. **Iterative Method**: Iterative or Indirect Methods, start with a guess of the solution and then repeatedly refine the solution until a certain convergence criterion is reached. Iterative methods are generally less efficient than direct methods because large number of operations required. Example- Jacobi's Iteration Method, Gauss-Seidal Iteration Method.

Implementation in C-

```
//Implementation of Jacobi's Method
void JacobisMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    while(!rootFound){
        for(i=0; i<n; i++){              //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*x[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                     //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){              //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
}

//Implementation of Gauss-Seidal Method
void GaussSeidalMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; //modified form of variables
    int rootFound=0; //flag

    int i, j;
    for(i=0; i<n; i++){                  //initialization
        Nx[i]=x[i];
    }
```

```
    while(!rootFound){
        for(i=0; i<n; i++){              //calculation
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*Nx[j];
            }
            Nx[i] = Nx[i] / a[i][i];
        }

        rootFound=1;                     //verification
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){              //evaluation
            x[i]=Nx[i];
        }
    }

    return ;
}

//Print array with comma separation
void print(int n, double x[n]){
    int i;
    for(i=0; i<n; i++){
        printf("%lf, ", x[i]);
    }
    printf("\n\n");

    return ;
}

int main(){
    //equation initialization
    int n=3;    //number of variables

    double x[n];    //variables

    double b[n],    //constants
        a[n][n];    //arguments

    //assign values
    a[0][0]=8; a[0][1]=2; a[0][2]=-2; b[0]=8;    //8x₁+2x₂-2x₃+8=0
    a[1][0]=1; a[1][1]=-8; a[1][2]=3; b[1]=-4;   //x₁-8x₂+3x₃-4=0
    a[2][0]=2; a[2][1]=1; a[2][2]=9; b[2]=12;    //2x₁+x₂+9x₃+12=0


    int i;

    for(i=0; i<n; i++){                      //initialization
        x[i]=0;
    }
    JacobisMethod(n, x, b, a);
    print(n, x);


    for(i=0; i<n; i++){                      //initialization
```

```
        x[i]=0;
    }
    GaussSeidalMethod(n, x, b, a);
    print(n, x);

    return 0;
}
```

Section 46.2: Non-Linear Equation

An equation of the type `f(x)=0` is either algebraic or transcendental. These types of equations can be solved by using two types of methods-

1. **Direct Method**: This method gives the exact value of all the roots directly in a finite number of steps.
2. **Indirect or Iterative Method**: Iterative methods are best suited for computer programs to solve an equation. It is based on the concept of successive approximation. In Iterative Method there are two ways to solve an equation-
    - **Bracketing Method**: We take two initial points where the root lies in between them. Example- Bisection Method, False Position Method.
    - **Open End Method**: We take one or two initial values where the root may be any-where. Example- Newton-Raphson Method, Successive Approximation Method, Secant Method.

Implementation in C:

```
/// Here define different functions to work with
#define f(x) ( ((x)*(x)*(x)) - (x) - 2 )
#define f2(x) ( (3*(x)*(x)) - 1 )
#define g(x) ( cbrt( (x) + 2 ) )


/**
* Takes two initial values and shortens the distance by both side.
**/
double BisectionMethod(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;
            c=(a+b)/2;

            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }
```

```
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Takes two initial values and shortens the distance by single side.
**/
double FalsePosition(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b) < 0){
        while(1){
            loopCounter++;

            c=(a*f(b) - b*f(a)) / (f(b) - f(a));

            /*/printf("%lf\t %lf \n", c, f(c));/**////test
            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c)) < 0){
                b=c;
            }else{
                a=c;
            }
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double NewtonRaphson(){
    double root=0;

    double x1=1;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        x2 = x1 - (f(x1)/f2(x1));
        /*/printf("%lf \t %lf \n", x2, f(x2));/**////test

        if(f(x2)<0.00001 && f(x2)>-0.00001){
            root=x2;
            break;
        }
```

```
        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* Uses one initial value and gradually takes that value near to the real one.
**/
double FixedPoint(){
    double root=0;
    double x=1;

    int loopCounter=0;
    while(1){
        loopCounter++;

        if( (x-g(x)) <0.00001 && (x-g(x)) >-0.00001){
            root = x;
            break;
        }

        /*/printf("%lf \t %lf \n", g(x), x-(g(x)));/**////test

        x=g(x);
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

/**
* uses two initial values & both value approaches to the root.
**/
double Secant(){
    double root=0;

    double x0=1;
    double x1=2;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        /*/printf("%lf \t %lf \t %lf \n", x0, x1, f(x1));/**////test

        if(f(x1)<0.00001 && f(x1)>-0.00001){
            root=x1;
            break;
        }

        x2 = ((x0*f(x1))-(x1*f(x0))) / (f(x1)-f(x0));

        x0=x1;
        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}
```

```
int main(){
    double root;

    root = BisectionMethod();
    printf("Using Bisection Method the root is: %lf \n\n", root);

    root = FalsePosition();
    printf("Using False Position Method the root is: %lf \n\n", root);

    root = NewtonRaphson();
    printf("Using Newton-Raphson Method the root is: %lf \n\n", root);

    root = FixedPoint();
    printf("Using Fixed Point Method the root is: %lf \n\n", root);

    root = Secant();
    printf("Using Secant Method the root is: %lf \n\n", root);

    return 0;
}
```

Chapter 47: Longest Common Subsequence

Section 47.1: Longest Common Subsequence Explanation

One of the most important implementations of Dynamic Programming is finding out the Longest Common Subsequence. Let's define some of the basic terminologies first.

Subsequence:

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Let's say we have a string **ABC**. If we erase zero or one or more than one character from this string we get the subsequence of this string. So the subsequences of string **ABC** will be {**"A"**, **"B"**, **"C"**, **"AB"**, **"AC"**, **"BC"**, **"ABC"**, **" "**}. Even if we remove all the characters, the empty string will also be a subsequence. To find out the subsequence, for each characters in a string, we have two options - either we take the character, or we don't. So if the length of the string is **n**, there are **2n** subsequences of that string.

Longest Common Subsequence:

As the name suggest, of all the common subsequencesbetween two strings, the longest common subsequence(LCS) is the one with the maximum length. For example: The common subsequences between **"HELLOM"** and **"HMLD"** are **"H"**, **"HL"**, **"HM"** etc. Here **"HLL"** is the longest common subsequence which has length 3.

Brute-Force Method:

We can generate all the subsequences of two strings using *backtracking*. Then we can compare them to find out the common subsequences. After we'll need to find out the one with the maximum length. We have already seen that, there are **2n** subsequences of a string of length **n**. It would take years to solve the problem if our **n** crosses **20-25**.

Dynamic Programming Method:

Let's approach our method with an example. Assume that, we have two strings **abcdaf** and **acbcf**. Let's denote these with **s1** and **s2**. So the longest common subsequence of these two strings will be **"abcf"**, which has length 4. Again I remind you, subsequences need not be continuous in the string. To construct **"abcf"**, we ignored **"da"** in **s1** and **"c"** in **s2**. How do we find this out using Dynamic Programming?

We'll start with a table (a 2D array) having all the characters of **s1** in a row and all the characters of **s2** in column. Here the table is 0-indexed and we put the characters from 1 to onwards. We'll traverse the table from left to right for each row. Our table will look like:

	0	1	2	3	4	5	6
chr		a	b	c	d	a	f
0							
1	a						
2	c						
3	b						
4	c						

```
5  |  f  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Here each row and column represent the length of the longest common subsequence between two strings if we take the characters of that row and column and add to the prefix before it. For example: **Table[2][3]** represents the length of the longest common subsequence between **"ac"** and **"abc"**.

The 0-th column represents the empty subsequence of **s1**. Similarly the 0-th row represents the empty subsequence of **s2**. If we take an empty subsequence of a string and try to match it with another string, no matter how long the length of the second substring is, the common subsequence will have 0 length. So we can fill-up the 0-th rows and 0-th columns with 0's. We get:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Let's begin. When we're filling **Table[1][1]**, we're asking ourselves, if we had a string **a** and another string **a** and nothing else, what will be the longest common subsequence here? The length of the LCS here will be 1. Now let's look at **Table[1][2]**. We have string **ab** and string **a**. The length of the LCS will be 1. As you can see, the rest of the values will be also 1 for the first row as it considers only string **a** with **abcd**, **abcda**, **abcdaf**. So our table will look like:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

For row 2, which will now include **c**. For **Table[2][1]** we have **ac** on one side and **a** on the other side. So the length of the LCS is 1. Where did we get this 1 from? From the top, which denotes the LCS **a** between two substrings. So what we are saying is, if **s1[2]** and **s2[1]** are not same, then the length of the LCS will be the maximum of the length of

LCS at the **top**, or at the **left**. Taking the length of the LCS at the top denotes that, we don't take the current character from **s2**. Similarly, Taking the length of the LCS at the left denotes that, we don't take the current character from **s1** to create the LCS. We get:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |  1  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

So our first formula will be:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1]
endif
```

Moving on, for **Table[2][2]** we have string **ab** and **ac**. Since **c** and **b** are not same, we put the maximum of the top or left here. In this case, it's again 1. After that, for **Table[2][3]** we have string **abc** and **ac**. This time current values of both row and column are same. Now the length of the LCS will be equal to the maximum length of LCS so far + 1. How do we get the maximum length of LCS so far? We check the diagonal value, which represents the best match between **ab** and **a**. From this state, for the current values, we added one more character to **s1** and **s2** which happened to be the same. So the length of LCS will of course increase. We'll put **1 + 1 = 2** in **Table[2][3]**. We get,

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0   |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1   |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2   |  c  |  0  |  1  |  1  |  2  |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3   |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4   |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5   |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

So our second formula will be:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

We have defined both the cases. Using these two formulas, we can populate the whole table. After filling up the table, it will look like this:

```
             0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The length of the LCS between **s1** and **s2** will be **Table[5][6] = 4**. Here, 5 and 6 are the length of **s2** and **s1** respectively. Our pseudo-code will be:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

The time complexity for this algorithm is: **O(mn)** where **m** and **n** denotes the length of each strings.

How do we find out the longest common subsequence? We'll start from the bottom-right corner. We will check from where the value is coming. If the value is coming from the diagonal, that is if **Table[i-1][j-1]** is equal to **Table[i][j] - 1**, we push either **s2[i]** or **s1[j]** (both are the same) and move diagonally. If the value is coming from top, that means, if **Table[i-1][j]** is equal to **Table[i][j]**, we move to the top. If the value is coming from left, that means, if **Table[i][j-1]** is equal to **Table[i][j]**, we move to the left. When we reach the leftmost or topmost column, our search ends. Then we pop the values from the stack and print them. The pseudo-code:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
     if Table[i-1][j-1] == Table[i][j] - 1 and s1[j]==s2[i]
```

```
        S.push(s1[j])   //or S.push(s2[i])
        i := i - 1
        j := j - 1
    else if Table[i-1][j] == Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
while S is not empty
    print(S.pop)
endwhile
```

Point to be noted: if both **Table[i-1][j]** and **Table[i][j-1]** is equal to **Table[i][j]** and **Table[i-1][j-1]** is not equal to **Table[i][j] - 1**, there can be two LCS for that moment. This pseudo-code doesn't consider this situation. You'll have to solve this recursively to find multiple LCSs.

The time complexity for this algorithm is: **O(max(m, n))**.

Chapter 48: Longest Increasing Subsequence

Section 48.1: Longest Increasing Subsequence Basic Information

The Longest Increasing Subsequence problem is to find subsequence from the give input sequence in which subsequence's elements are sorted in lowest to highest order. All subsequence are not contiguous or unique.

Application of Longest Increasing Subsequence:

Algorithms like Longest Increasing Subsequence, Longest Common Subsequence are used in version control systems like Git and etc.

Simple form of Algorithm:

1. Find unique lines which are common to both documents.
2. Take all such lines from the first document and order them according to their appearance in the second document.
3. Compute the LIS of the resulting sequence (by doing a Patience Sort), getting the longest matching sequence of lines, a correspondence between the lines of two documents.
4. Recurse the algorithm on each range of lines between already matched ones.

Now let us consider a simpler example of the LCS problem. Here, input is only one sequence of distinct integers `a1, a2, ..., an.`, and we want to find the longest increasing subsequence in it. For example, if input is **7,3,8,4,2,6** then the longest increasing subsequence is **3,4,6**.

The easiest approach is to sort input elements in increasing order, and apply the LCS algorithm to the original and sorted sequences. However, if you look at the resulting array you would notice that many values are the same, and the array looks very repetitive. This suggest that the LIS (longest increasing subsequence) problem can be done with dynamic programming algorithm using only one-dimensional array.

Pseudo Code:

1. Describe an array of values we want to compute.
   For `1 <= i <= n`, let **A(i)** be the length of a longest increasing sequence of input. Note that the length we are ultimately interested in is `max{A(i)|1 ≤ i ≤ n}`.
2. Give a recurrence.
   For `1 <= i <= n`, `A(i) = 1 + max{A(j)|1 ≤ j < i` and `input(j) < input(i)}.`
3. Compute the values of A.
4. Find the optimal solution.

The following program uses A to compute an optimal solution. The first part computes a value m such that **A(m)** is the length of an optimal increasing subsequence of input. The second part computes an optimal increasing subsequence, but for convenience we print it out in reverse order. This program runs in time O(n), so the entire algorithm runs in time O(n^2).

Part 1:

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
```

```
        m ← i
    end if
end for
```

Part 2:

```
put a
while A(m) > 1 do
    i ← m-1
    while not(ai < am and A(i) = A(m)-1) do
        i ← i-1
    end while
    m ← i
    put a
 end while
```

Recursive Solution:

Approach 1:

```
LIS(A[1..n]):
    if (n = 0) then return 0
    m = LIS(A[1..(n - 1)])
    B is subsequence of A[1..(n - 1)] with only elements less than a[n]
    (* let h be size of B, h ≤ n-1 *)
    m = max(m, 1 + LIS(B[1..h]))
    Output m
```

Time complexity in Approach 1 : `O(n*2^n)`

Approach 2:

```
LIS(A[1..n], x):
    if (n = 0) then return 0
    m = LIS(A[1..(n - 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS(A[1..(n - 1)], A[n]))
    Output m

MAIN(A[1..n]):
    return LIS(A[1..n], ∞)
```

Time Complexity in Approach 2: `O(n^2)`

Approach 3:

```
LIS(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n - 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS(A[1..i]))
    return m

MAIN(A[1..n]):
    return LIS(A[1..i])
```

Time Complexity in Approach 3: `O(n^2)`

Iterative Algorithm:

Computes the values iteratively in bottom up fashion.

```
LIS(A[1..n]):
    Array L[1..n]
    (* L[i] = value of LIS ending(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i - 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L

MAIN(A[1..n]):
    L = LIS(A[1..n])
        return the maximum value in L
```

Time complexity in Iterative approach: `O(n^2)`

Auxiliary Space: `O(n)`

Lets take **{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}** as input. So, Longest Increasing Subsequence for the given input is **{0, 2, 6, 9, 11, 15}**.

Chapter 49: Check two strings are anagrams

Two string with same set of character is called anagram. I have used javascript here.

We will create an hash of str1 and increase count +1. We will loop on 2nd string and check all characters are there in hash and decrease value of hash key. Check all value of hash key are zero will be anagram.

Section 49.1: Sample input and output

Ex1:

```
let str1 = 'stackoverflow';
let str2 = 'flowerovstack';
```

These strings are anagrams.

// Create Hash from str1 and increase one count.

```
hashMap = {
    s : 1,
    t : 1,
    a : 1,
    c : 1,
    k : 1,
    o : 2,
    v : 1,
    e : 1,
    r : 1,
    f : 1,
    l : 1,
    w : 1
}
```

You can see hashKey 'o' is containing value 2 because o is 2 times in string.

Now loop over str2 and check for each character are present in hashMap, if yes, decrease value of hashMap Key, else return false (which indicate it's not anagram).

```
hashMap = {
    s : 0,
    t : 0,
    a : 0,
    c : 0,
    k : 0,
    o : 0,
    v : 0,
    e : 0,
    r : 0,
    f : 0,
    l : 0,
    w : 0
}
```

Now, loop over hashMap object and check all values are zero in the key of hashMap.

In our case all values are zero so its a anagram.

Section 49.2: Generic Code for Anagrams

```
(function(){

    var hashMap = {};

    function isAnagram (str1, str2) {

        if(str1.length !== str2.length){
            return false;
        }

        // Create hash map of str1 character and increase value one (+1).
        createStr1HashMap(str1);

        // Check str2 character are key in hash map and decrease value by one(-1);
        var valueExist = createStr2HashMap(str2);

        // Check all value of hashMap keys are zero, so it will be anagram.
        return isStringsAnagram(valueExist);
    }

    function createStr1HashMap (str1) {
        [].map.call(str1, function(value, index, array){
            hashMap[value] = value in hashMap ?  (hashMap[value] + 1) : 1;
            return value;
        });
    }

    function createStr2HashMap (str2) {
        var valueExist = [].every.call(str2, function(value, index, array){
            if(value in hashMap) {
                hashMap[value] = hashMap[value] - 1;
            }
            return value in hashMap;
        });
        return valueExist;
    }

    function isStringsAnagram (valueExist) {
        if(!valueExist) {
            return valueExist;
        } else {
            var isAnagram;
            for(var i in hashMap) {
                if(hashMap[i] !== 0) {
                    isAnagram = false;
                    break;
                } else {
                    isAnagram = true;
                }
            }

            return isAnagram;
        }
    }

    isAnagram('stackoverflow', 'flowerovstack'); // true
    isAnagram('stackoverflow', 'flowervvstack'); // false
```

```
})();
```

Time complexity: 3n i.e O(n).

Chapter 50: Pascal's Triangle

Section 50.1: Pascal triangle in C

```
int i, space, rows, k=0, count = 0, count1 = 0;
row=5;
for(i=1; i<=rows; ++i)
{
    for(space=1; space <= rows-i; ++space)
    {
        printf("  ");
        ++count;
    }

    while(k != 2*i-1)
    {
        if (count <= rows-1)
        {
            printf("%d ", i+k);
            ++count;
        }
        else
        {
            ++count1;
            printf("%d ", (i+k-2*count1));
        }
        ++k;
    }
    count1 = count = k = 0;

    printf("\n");
}
```

Output

```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

Chapter 51: Algo:- Print a m*n matrix in square wise

Check sample input and output below.

Section 51.1: Sample Example

```
Input:

14 15 16 17 18 21
19 10 20 11 54 36
64 55 44 23 80 39
91 92 93 94 95 42

Output:
print value in index
14 15 16 17 18 21 36 39 42 95 94 93 92 91 64 19 10 20 11 54 80 23 44 55

or print index
00 01 02 03 04 05 15 25 35 34 33 32 31 30 20 10 11 12 13 14 24 23 22 21
```

Section 51.2: Write the generic code

```
function noOfLooping(m,n) {
    if(m > n) {
        smallestValue = n;
    } else {
        smallestValue = m;
    }

    if(smallestValue % 2 == 0) {
            return smallestValue/2;
    } else {
            return (smallestValue+1)/2;
    }
}

function squarePrint(m,n) {
    var looping = noOfLooping(m,n);
    for(var i = 0; i < looping; i++) {
        for(var j = i; j < m - 1 - i; j++) {
                console.log(i+''+j);
        }
        for(var k = i; k < n - 1 - i; k++) {
                console.log(k+''+j);
        }
        for(var l = j; l > i; l--) {
                console.log(k+''+l);
        }
        for(var x = k; x > i; x--) {
                console.log(x+''+l);
        }
    }
}

squarePrint(6,4);
```

Chapter 52: Matrix Exponentiation

Section 52.1: Matrix Exponentiation to Solve Example Problems

Find f(n): nth Fibonacci number. The problem is quite easy when **n** is relatively small. We can use simple recursion, `f(n) = f(n-1) + f(n-2)`, or we can use dynamic programming approach to avoid the calculation of same function over and over again. But what will you do if the problem says, **Given 0 < n < 10^9, find f(n) mod 999983?** Dynamic programming will fail, so how do we tackle this problem?

First let's see how matrix exponentiation can help to represent recursive relation.

Prerequisites:

- Given two matrices, know how to find their product. Further, given the product matrix of two matrices, and one of them, know how to find the other matrix.
- Given a matrix of size **d X d**, know how to find its nth power in **O(d3log(n))**.

Patterns:

At first we need a recursive relation and we want to find a matrix **M** which can lead us to the desired state from a set of already known states. Let's assume that, we know the **k** states of a given recurrence relation and we want to find the **(k+1)th** state. Let **M** be a **k X k** matrix, and we build a matrix **A:[k X 1]** from the known states of the recurrence relation, now we want to get a matrix **B:[k X 1]** which will represent the set of next states, i. e. **M X A = B** as shown below:

```
      |  f(n)  |      | f(n+1) |
      | f(n-1) |      |  f(n)  |
  M X | f(n-2) |  =   | f(n-1) |
      | ...... |      | ...... |
      | f(n-k) |      |f(n-k+1)|
```

So, if we can design **M** accordingly, our job will be done! The matrix will then be used to represent the recurrence relation.

Type 1:
Let's start with the simplest one, `f(n) = f(n-1) + f(n-2)`
We get, `f(n+1) = f(n) + f(n-1)`.
Let's assume, we know `f(n)` and `f(n-1)`; We want to find out `f(n+1)`.
From the situation stated above, matrix **A** and matrix **B** can be formed as shown below:

```
 Matrix A          Matrix B

|  f(n)  |        | f(n+1) |
| f(n-1) |        |  f(n)  |
```

[Note: Matrix **A** will be always designed in such a way that, every state on which `f(n+1)` depends, will be present]
Now, we need to design a **2X2** matrix **M** such that, it satisfies **M X A = B** as stated above.
The first element of **B** is `f(n+1)` which is actually `f(n) + f(n-1)`. To get this, from matrix **A**, we need, **1 X f(n)** and **1 X f(n-1)**. So the first row of **M** will be **[1 1]**.

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| ----- |     | f(n-1) |     | ------ |
```

[Note: ----- means we are not concerned about this value.]
Similarly, 2nd item of **B** is `f(n)` which can be got by simply taking **1 X f(n)** from **A**, so the 2nd row of **M** is [1 0].

```
| ----- |  X  |  f(n)  |  =  | ------ |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Then we get our desired **2 X 2** matrix **M**.

```
| 1   1 |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

These matrices are simply derived using matrix multiplication.

Type 2:

Let's make it a little complex: find `f(n) = a X f(n-1) + b X f(n-2)`, where **a** and **b** are constants.
This tells us, `f(n+1) = a X f(n) + b X f(n-1)`.
By this far, this should be clear that the dimension of the matrices will be equal to the number of dependencies, i.e. in this particular example, again 2. So for **A** and **B**, we can build two matrices of size **2 X 1**:

```
Matrix A            Matrix B
|  f(n)  |         | f(n+1) |
| f(n-1) |         |  f(n)  |
```

Now for `f(n+1) = a X f(n) + b X f(n-1)`, we need [a, b] in the first row of objective matrix **M**. And for the 2nd item in **B**, i.e. `f(n)` we already have that in matrix **A**, so we just take that, which leads, the 2nd row of the matrix M to [1 0]. This time we get:

```
| a   b |  X  |  f(n)  |  =  | f(n+1) |
| 1   0 |     | f(n-1) |     |  f(n)  |
```

Pretty simple, eh?

Type 3:

If you've survived through to this stage, you've grown much older, now let's face a bit complex relation: find `f(n) = a X f(n-1) + c X f(n-3)`?
Ooops! A few minutes ago, all we saw were contiguous states, but here, the state **f(n-2)** is missing. Now?

Actually this is not a problem anymore, we can convert the relation as follows: `f(n) = a X f(n-1) + 0 X f(n-2) + c X f(n-3)`, deducing `f(n+1) = a X f(n) + 0 X f(n-1) + c X f(n-2)`. Now, we see that, this is actually a form described in Type 2. So here the objective matrix **M** will be **3 X 3**, and the elements are:

```
| a 0 c |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 1 0 |     | f(n-2) |     | f(n-1) |
```

These are calculated in the same way as type 2, if you find it difficult, try it on pen and paper.

Type 4:

Life is getting complex as hell, and Mr, Problem now asks you to find `f(n) = f(n-1) + f(n-2) + c` where **c** is any constant.
Now this is a new one and all we have seen in past, after the multiplication, each state in **A** transforms to its next

state in **B**.

```
f(n) = f(n-1) + f(n-2) + c
f(n+1) = f(n) + f(n-1) + c
f(n+2) = f(n+1) + f(n) + c
..................... so on
```

So , normally we can't get it through previous fashion, but how about we add **c** as a state:

```
     |  f(n)  |   | f(n+1) |
  M X | f(n-1) | = |  f(n)  |
     |    c   |   |    c   |
```

Now, its not much hard to design **M**. Here's how its done, but don't forget to verify:

```
| 1 1 1 |     |  f(n)  |     | f(n+1) |
| 1 0 0 |  X  | f(n-1) |  =  |  f(n)  |
| 0 0 1 |     |    c   |     |    c   |
```

Type 5:

Let's put it altogether: find `f(n) = a X f(n-1) + c X f(n-3) + d X f(n-4) + e`. Let's leave it as an exercise for you. First try to find out the states and matrix **M**. And check if it matches with your solution. Also find matrix **A** and **B**.

```
| a 0 c d 1 |
| 1 0 0 0 0 |
| 0 1 0 0 0 |
| 0 0 1 0 0 |
| 0 0 0 0 1 |
```

Type 6:

Sometimes the recurrence is given like this:

```
f(n) = f(n-1)   -> if n is odd
f(n) = f(n-2)   -> if n is even
```

In short:

```
f(n) = (n&1) X f(n-1) + (!(n&1)) X f(n-2)
```

Here, we can split the functions in the basis of odd even and keep 2 different matrix for both of them and calculate them separately.

Type 7:

Feeling little too confident? Good for you. Sometimes we may need to maintain more than one recurrence, where they are interested. For example, let a recurrence re;atopm be:

```
g(n) = 2g(n-1) + 2g(n-2) + f(n)
```

Here, recurrence *g(n)* is dependent upon `f(n)` and this can be calculated in the same matrix but of increased dimensions. From these let's at first design the matrices **A** and **B**.

```
 Matrix A            Matrix B
|  g(n)  |          | g(n+1) |
| g(n-1) |          |  g(n)  |
| f(n+1) |          | f(n+2) |
|  f(n)  |          | f(n+1) |
```

Here, `g(n+1) = 2g(n-1) + f(n+1) and f(n+2) = 2f(n+1) + 2f(n)`. Now, using the processes stated above, we can find the objective matrix **M** to be:

```
| 2 2 1 0 |
| 1 0 0 0 |
| 0 0 2 2 |
| 0 0 1 0 |
```

So, these are the basic categories of recurrence relations which are used to solveby this simple technique.

Chapter 53: polynomial-time bounded algorithm for Minimum Vertex Cover

Variable	Meaning
G	Input connected un-directed graph
X	Set of vertices
C	Final set of vertices

This is a polynomial algorithm for getting the minimum vertex cover of connected undirected graph. The time complexity of this algorithm is O(n2)

Section 53.1: Algorithm Pseudo Code

Algorithm PMinVertexCover (graph G)
Input connected graph G
Output Minimum Vertex Cover Set C

```
Set C <- new Set<Vertex>()

Set X <- new Set<Vertex>()

X <- G.getAllVerticiesArrangedDescendinglyByDegree()

for v in X do
    List<Vertex> adjacentVertices1 <- G.getAdjacent(v)

    if !C contains any of adjacentVertices1 then

        C.add(v)

for vertex in C do

    List<vertex> adjacentVertices2 <- G.adjacentVertecies(vertex)

    if C contains any of adjacentVertices2 then

        C.remove(vertex)


return C
```

> C is the minimum vertex cover of graph G

> we can use bucket sort for sorting the vertices according to its degree because the maximum value of degrees is (n-1) where n is the number of vertices then the time complexity of the sorting will be O(n)

Chapter 54: Dynamic Time Warping

Section 54.1: Introduction To Dynamic Time Warping

Dynamic Time Warping (DTW) is an algorithm for measuring similarity between two temporal sequences which may vary in speed. For instance, similarities in walking could be detected using DTW, even if one person was walking faster than the other, or if there were accelerations and decelerations during the course of an observation. It can be used to match a sample voice command with others command, even if the person talks faster or slower than the prerecorded sample voice. DTW can be applied to temporal sequences of video, audio and graphics data-indeed, any data which can be turned into a linear sequence can be analyzed with DTW.

In general, DTW is a method that calculates an optimal match between two given sequences with certain restrictions. But let's stick to the simpler points here. Let's say, we have two voice sequences **Sample** and **Test**, and we want to check if these two sequences match or not. Here voice sequence refers to the converted digital signal of your voice. It might be the amplitude or frequency of your voice that denotes the words you say. Let's assume:

```
Sample = {1, 2, 3, 5, 5, 5, 6}
Test   = {1, 1, 2, 2, 3, 5}
```

We want to find out the optimal match between these two sequences.

At first, we define the distance between two points, *d(x, y)* where **x** and **y** represent the two points. Let,

```
d(x, y) = |x - y|     //absolute difference
```

Let's create a 2D matrix **Table** using these two sequences. We'll calculate the distances between each point of **Sample** with every points of **Test** and find the optimal match between them.

	0	1	1	2	2	3	5
0							
1							
2							
3							
5							
5							
5							
6							

Here, **Table[i][j]** represents the optimal distance between two sequences if we consider the sequence up to **Sample[i]** and **Test[j]**, considering all the optimal distances we observed before.

For the first row, if we take no values from **Sample**, the distance between this and **Test** will be *infinity*. So we put *infinity* on the first row. Same goes for the first column. If we take no values from **Test**, the distance between this one and **Sample** will also be infinity. And the distance between **0** and **0** will simply be **0**. We get,

	0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf						
2	inf						
3	inf						
5	inf						
5	inf						
5	inf						
6	inf						

Now for each step, we'll consider the distance between each points in concern and add it with the minimum distance we found so far. This will give us the optimal distance of two sequences up to that position. Our formula will be,

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

For the first one, **d(1, 1)** = **0**, **Table[0][0]** represents the minimum. So the value of **Table[1][1]** will be **0** + **0** = **0**. For the second one, **d(1, 2)** = **0**. **Table[1][1]** represents the minimum. The value will be: **Table[1][2]** = **0** + **0** = **0**. If we continue this way, after finishing, the table will look like:

	0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	1

The value at **Table[7][6]** represents the maximum distance between these two given sequences. Here **1** represents the maximum distance between **Sample** and **Test** is **1**.

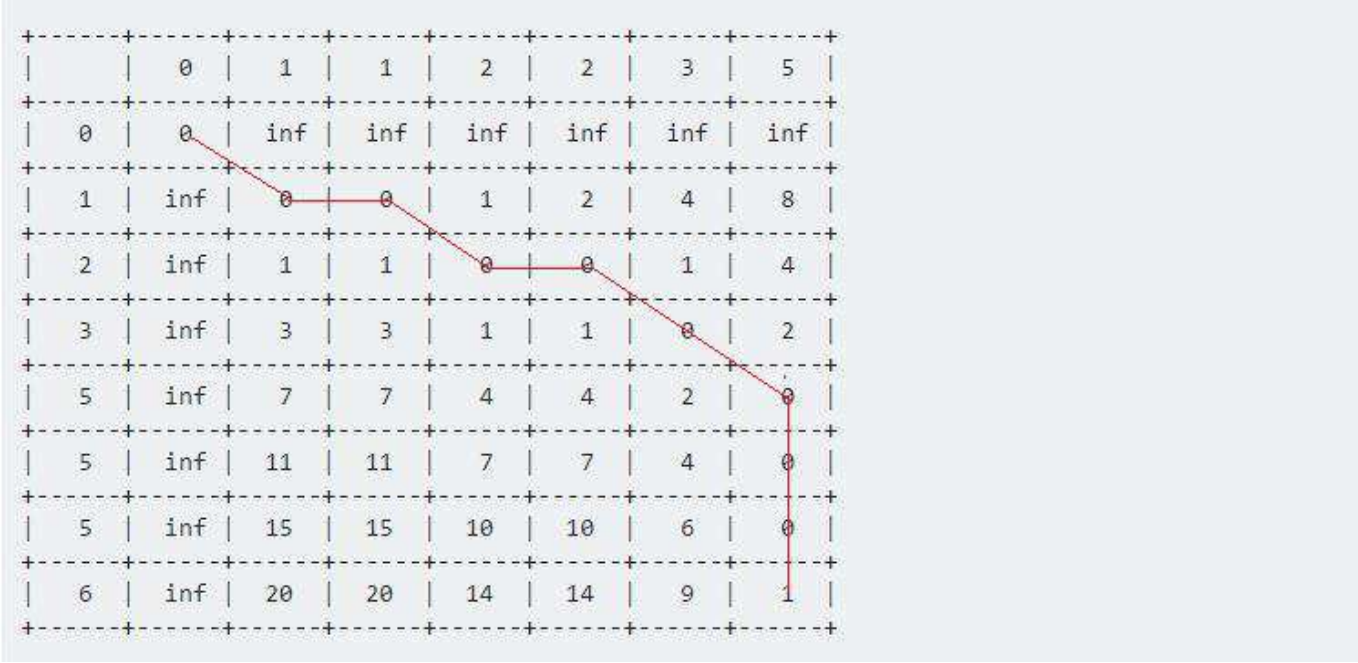
Now if we backtrack from the last point, all the way back towards the starting **(0, 0)** point, we get a long line that moves horizontally, vertically and diagonally. Our backtracking procedure will be:

```
if Table[i-1][j-1] <= Table[i-1][j] and Table[i-1][j-1] <= Table[i][j-1]
```

```
    i := i - 1
    j := j - 1
else if Table[i-1][j] <= Table[i-1][j-1] and Table[i-1][j] <= Table[i][j-1]
    i := i - 1
else
    j := j - 1
end if
```

We'll continue this till we reach **(0, 0)**. Each move has its own meaning:

- A horizontal move represents deletion. That means our **Test** sequence accelerated during this interval.
- A vertical move represents insertion. That means out **Test** sequence decelerated during this interval.
- A diagonal move represents match. During this period **Test** and **Sample** were same.



Our pseudo-code will be:

```
Procedure DTW(Sample, Test):
n := Sample.length
m := Test.length
Create Table[n + 1][m + 1]
for i from 1 to n
    Table[i][0] := infinity
end for
for i from 1 to m
    Table[0][i] := infinity
end for
Table[0][0] := 0
for i from 1 to n
    for j from 1 to m
        Table[i][j] := d(Sample[i], Test[j])
                       + minimum(Table[i-1][j-1],      //match
                                 Table[i][j-1],        //insertion
                                 Table[i-1][j])        //deletion
    end for
end for
Return Table[n + 1][m + 1]
```

We can also add a locality constraint. That is, we require that if `Sample[i]` is matched with `Test[j]`, then `|i - j|` is no larger than **w**, a window parameter.

Complexity:

The complexity of computing DTW is **O(m * n)** where **m** and **n** represent the length of each sequence. Faster techniques for computing DTW include PrunedDTW, SparseDTW and FastDTW.

Applications:

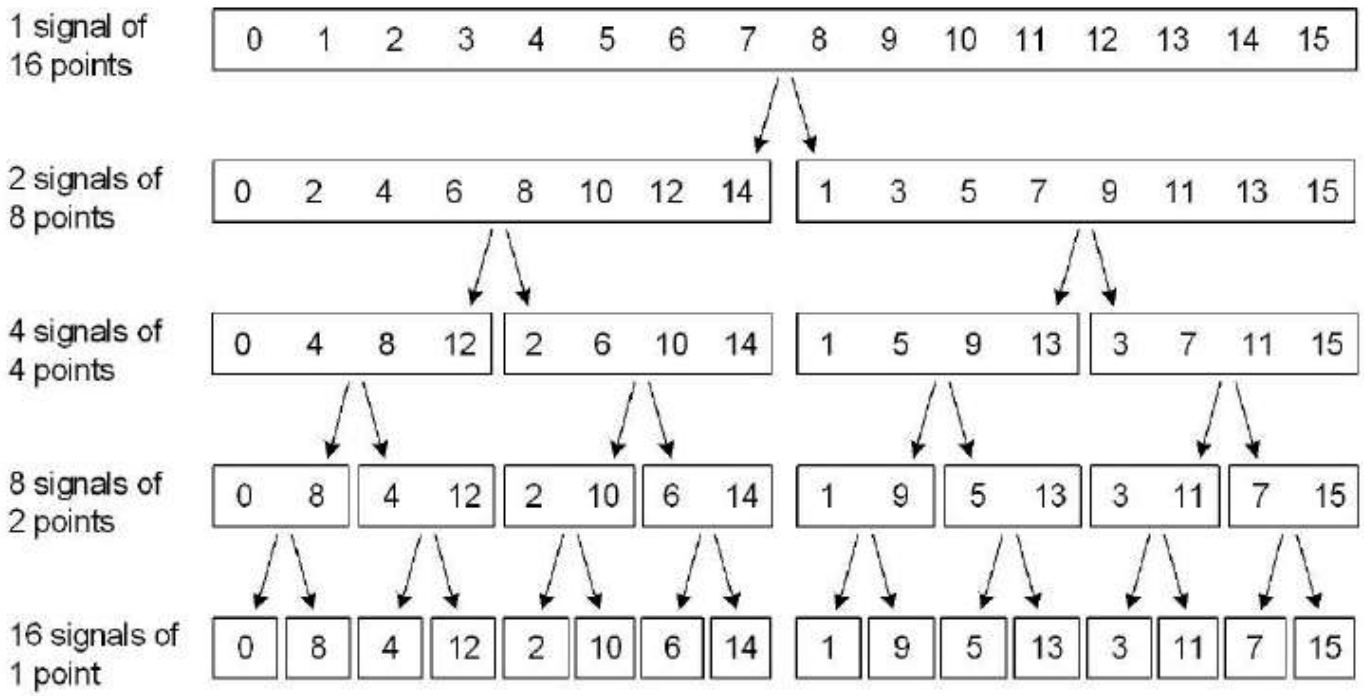
- Spoken word recognition
- Correlation Power Analysis

Chapter 55: Fast Fourier Transform

The Real and Complex form of DFT (**D**iscrete **F**ourier **T**ransforms) can be used to perform frequency analysis or synthesis for any discrete and periodic signals. The FFT (**F**ast **F**ourier **T**ransform) is an implementation of the DFT which may be performed quickly on modern CPUs.

Section 55.1: Radix 2 FFT

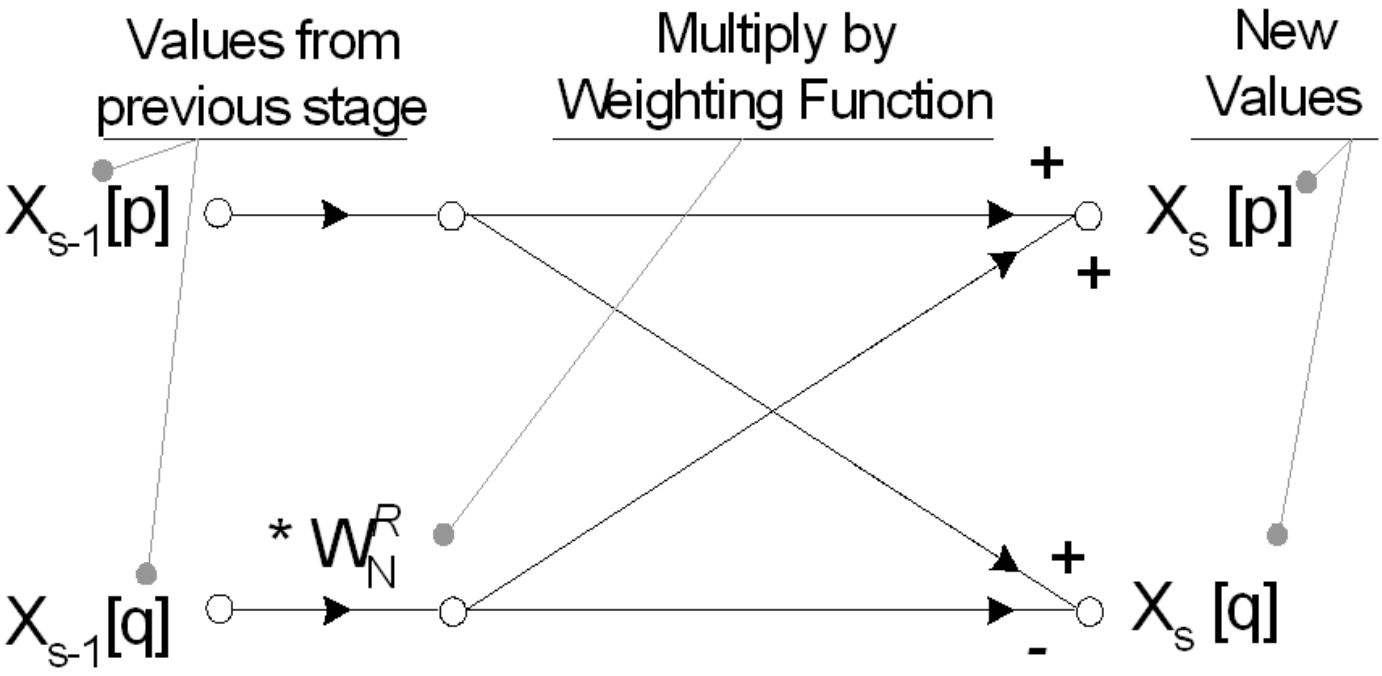
The simplest and perhaps best-known method for computing the FFT is the Radix-2 Decimation in Time algorithm. The Radix-2 FFT works by decomposing an N point time domain signal into N time domain signals each composed of a single point



.

Signal decomposition, or 'decimation in time' is achieved by bit reversing the indices for the array of time domain data. Thus, for a sixteen-point signal, sample 1 (Binary 0001) is swapped with sample 8 (1000), sample 2 (0010) is swapped with 4 (0100) and so on. Sample swapping using the bit reverse technique can be achieved simply in software, but limits the use of the Radix 2 FFT to signals of length N = 2^M.

The value of a 1-point signal in the time domain is equal to its value in the frequency domain, thus this array of decomposed single time-domain points requires no transformation to become an array of frequency domain points. The N single points; however, need to be reconstructed into one N-point frequency spectra. Optimal reconstruction of the complete frequency spectrum is performed using butterfly calculations. Each reconstruction stage in the Radix-2 FFT performs a number of two point butterflies, using a similar set of exponential weighting functions, Wn^R.

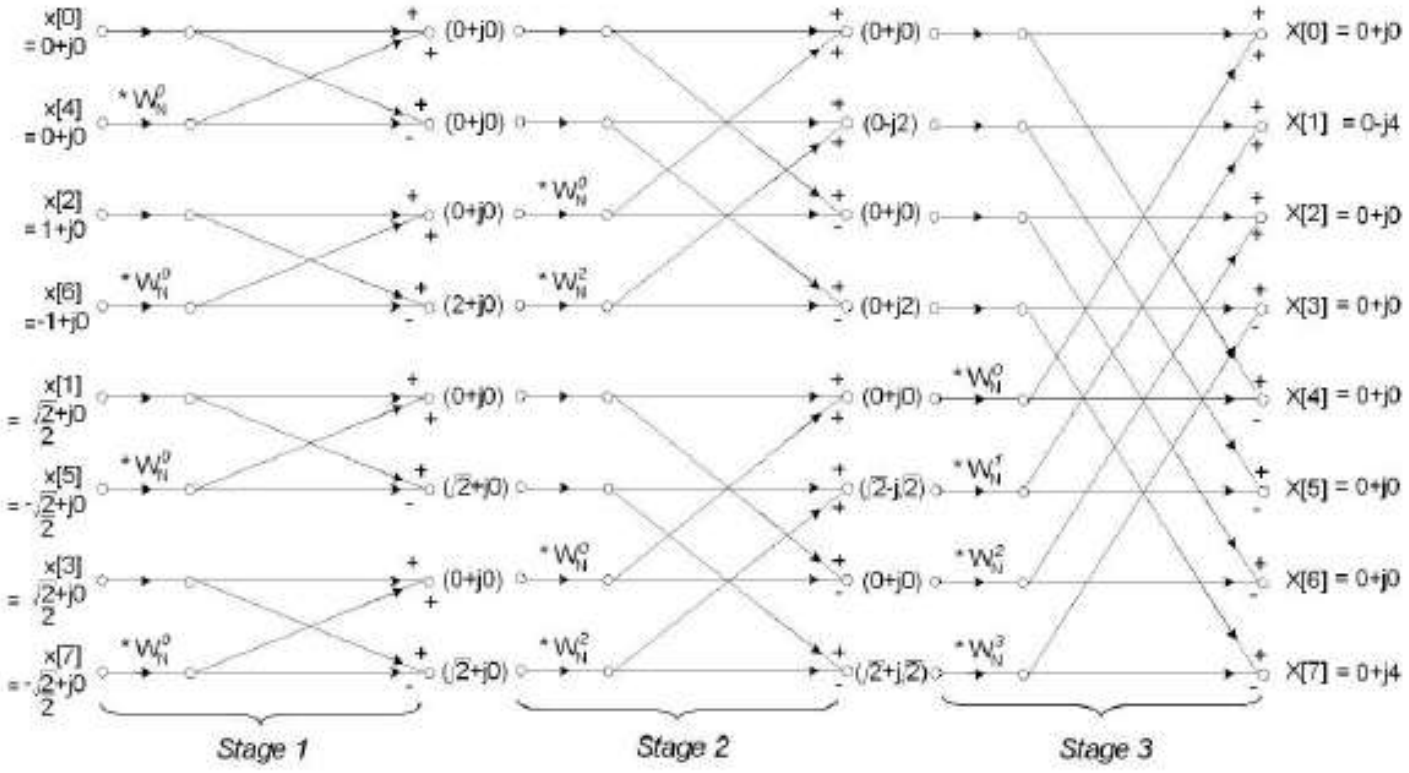


The FFT removes redundant calculations in the Discrete Fourier Transform by exploiting the periodicity of Wn^R. Spectral reconstruction is completed in log2(N) stages of butterfly calculations giving X[K]; the real and imaginary frequency domain data in rectangular form. To convert to magnitude and phase (polar coordinates) requires finding the absolute value, √(Re2 + Im2), and argument, tan-1(Im/Re).

Exponential Weighting Factor: $W_N^R = e^{-j(2\pi R/N)} = \cos(2\pi R/N) - j\sin(2\pi R/N)$

N:	*Number of points in the FFT*
R:	*Current WN Factor: depends on N, current FFT stage and separation of butterflies in that stage*

The complete butterfly flow diagram for an eight point Radix 2 FFT is shown below. Note the input signals have previously been reordered according to the decimation in time procedure outlined previously.



The FFT typically operates on complex inputs and produces a complex output. For real signals, the imaginary part may be set to zero and real part set to the input signal, x[n], however many optimisations are possible involving the transformation of real-only data. Values of Wn^R used throughout the reconstruction can be determined using the exponential weighting equation.

The value of R (the exponential weighting power) is determined the current stage in the spectral reconstruction and the current calculation within a particular butterfly.

Code Example (C/C++)

A C/C++ code sample for computing the Radix 2 FFT can be found below. This is a simple implementation which works for any size N where N is a power of 2. It is approx 3x slower than the fastest FFTw implementation, but still a very good basis for future optimisation or for learning about how this algorithm works.

```
#include <math.h>

#define PI        3.14159265358979323846264338327950    // PI for sine/cos calculations
#define TWOPI     6.28318530717958647692528676655900    // 2*PI for sine/cos calculations
#define Deg2Rad   0.01745329251994329576923690768488612  // Degrees to Radians factor
#define Rad2Deg   57.29577951308232087679815481410517    // Radians to Degrees factor
#define log10_2   0.30102999566398119521373889472449    // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942949483 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;         // Not so complicated after all
};

// Returns true if N is a power of 2
bool isPwrTwo(int N, int *M)
{
    *M = (int)ceil(log10((double)N) * log10_2_INV);// M is number of stages to perform. 2^M = N
    int NN = (int)pow(2.0, *M);
```

```
    if ((NN != N) || (NN == 0)) // Check N is a power of 2.
        return false;

    return true;
}

void rad2FFT(int N, complex *x, complex *DFT)
{
    int M = 0;

    // Check if power of two. If not, exit
    if (!isPwrTwo(N, &M))
        throw "Rad2FFT(): N must be a power of 2 for Radix FFT";

    // Integer Variables

    int BSep;                   // BSep is memory spacing between butterflies
    int BWidth;                 // BWidth is memory spacing of opposite ends of the butterfly
    int P;                      // P is number of similar Wn's to be used in that stage
    int j;                      // j is used in a loop to perform all calculations in each stage
    int stage = 1;              // stage is the stage number of the FFT. There are M stages in total
(1 to M).
    int HiIndex;                // HiIndex is the index of the DFT array for the top value of each
butterfly calc
    unsigned int iaddr;         // bitmask for bit reversal
    int ii;                     // Integer bitfield for bit reversal (Decimation in Time)
    int MM1 = M - 1;

    unsigned int i;
    int l;
    unsigned int nMax = (unsigned int)N;

    // Double Precision Variables
    double TwoPi_N = TWOPI / (double)N;    // constant to save computational time.  = 2*PI / N
    double TwoPi_NP;

    // complex Variables (See 'struct complex')
    complex WN;                 // Wn is the exponential weighting function in the form a + jb
    complex TEMP;               // TEMP is used to save computation in the butterfly calc
    complex *pDFT = DFT;        // Pointer to first elements in DFT array
    complex *pLo;               // Pointer for lo / hi value of butterfly calcs
    complex *pHi;
    complex *pX;                // Pointer to x[n]


    // Decimation In Time - x[n] sample sorting
    for (i = 0; i < nMax; i++, DFT++)
    {
        pX = x + i;             // Calculate current x[n] from base address *x and index i.
        ii = 0;                 // Reset new address for DFT[n]
        iaddr = i;              // Copy i for manipulations
        for (l = 0; l < M; l++) // Bit reverse i and store in ii...
        {
            if (iaddr & 0x01)     // Detemine least significant bit
                ii += (1 << (MM1 - l));    // Increment ii by 2^(M-1-l) if lsb was 1
            iaddr >>= 1;                // right shift iaddr to test next bit. Use logical
operations for speed increase
            if (!iaddr)
                break;
        }
        DFT = pDFT + ii;        // Calculate current DFT[n] from base address *pDFT and bit
reversed index ii
```

```
        DFT->Re = pX->Re;       // Update the complex array with address sorted time domain signal
x[n]
        DFT->Im = pX->Im;       // NB: Imaginary is always zero
    }

    // FFT Computation by butterfly calculation
    for (stage = 1; stage <= M; stage++) // Loop for M stages, where 2^M = N
    {
        BSep = (int)(pow(2, stage)); // Separation between butterflies = 2^stage
        P = N / BSep;              // Similar Wn's in this stage = N/Bsep
        BWidth = BSep / 2;     // Butterfly width (spacing between opposite points) = Separation /
2.

        TwoPi_NP = TwoPi_N*P;

        for (j = 0; j < BWidth; j++) // Loop for j calculations per butterfly
        {
            if (j != 0)             // Save on calculation if R = 0, as WN^0 = (1 + j0)
            {
                //WN.Re = cos(TwoPi_NP*j)
                WN.Re = cos(TwoPi_N*P*j);     // Calculate Wn (Real and Imaginary)
                WN.Im = -sin(TwoPi_N*P*j);
            }

            for (HiIndex = j; HiIndex < N; HiIndex += BSep) // Loop for HiIndex Step BSep
butterflies per stage
            {
                pHi = pDFT + HiIndex;                   // Point to higher value
                pLo = pHi + BWidth;                     // Point to lower value (Note VC++ adjusts
for spacing between elements)

                if (j != 0)                             // If exponential power is not zero...
                {
                    //CMult(pLo, &WN, &TEMP);           // Perform complex multiplication of Lovalue
with Wn
                    TEMP.Re = (pLo->Re * WN.Re) - (pLo->Im * WN.Im);
                    TEMP.Im = (pLo->Re * WN.Im) + (pLo->Im * WN.Re);

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;        // Find new Lovalue (complex subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);           // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
                else
                {
                    TEMP.Re = pLo->Re;
                    TEMP.Im = pLo->Im;

                    //CSub (pHi, &TEMP, pLo);
                    pLo->Re = pHi->Re - TEMP.Re;        // Find new Lovalue (complex subtraction)
                    pLo->Im = pHi->Im - TEMP.Im;

                    //CAdd (pHi, &TEMP, pHi);           // Find new Hivalue (complex addition)
                    pHi->Re = (pHi->Re + TEMP.Re);
                    pHi->Im = (pHi->Im + TEMP.Im);
                }
            }
        }
    }
```

```
    pLo = 0;    // Null all pointers
    pHi = 0;
    pDFT = 0;
    DFT = 0;
    pX = 0;
}
```

Section 55.2: Radix 2 Inverse FFT

Due to the strong duality of the Fourier Transform, adjusting the output of a forward transform can produce the inverse FFT. Data in the frequency domain can be converted to the time domain by the following method:

1. Find the complex conjugate of the frequency domain data by inverting the imaginary component for all instances of K.
2. Perform the forward FFT on the conjugated frequency domain data.
3. Divide each output of the result of this FFT by N to give the true time domain value.
4. Find the complex conjugate of the output by inverting the imaginary component of the time domain data for all instances of n.

***Note**: both frequency and time domain data are complex variables. Typically the imaginary component of the time domain signal following an inverse FFT is either zero, or ignored as rounding error. Increasing the precision of variables from 32-bit float to 64-bit double, or 128-bit long double significantly reduces rounding errors produced by several consecutive FFT operations.*

Code Example (C/C++)

```
#include <math.h>

#define PI       3.14159265358979323846264338327795    // PI for sine/cos calculations
#define TWOPI    6.28318530717958647692528676655900     // 2*PI for sine/cos calculations
#define Deg2Rad  0.01745329251994329576923690768488612  // Degrees to Radians factor
#define Rad2Deg  57.29577951308232087679815481410517    // Radians to Degrees factor
#define log10_2  0.30102999566398119521373889472449    // Log10 of 2
#define log10_2_INV 3.32192809488736234787031942949483 // 1/Log10(2)

// complex variable structure (double precision)
struct complex
{
public:
    double  Re, Im;         // Not so complicated after all
};

void rad2InverseFFT(int N, complex *x, complex *DFT)
{
    // M is number of stages to perform. 2^M = N
    double Mx = (log10((double)N) / log10((double)2));
    int a = (int)(ceil(pow(2.0, Mx)));
    int status = 0;
    if (a != N) // Check N is a power of 2
    {
        x = 0;
        DFT = 0;
        throw "rad2InverseFFT(): N must be a power of 2 for Radix 2 Inverse FFT";
    }

    complex *pDFT = DFT;         // Reset vector for DFT pointers
    complex *pX = x;             // Reset vector for x[n] pointer
    double NN = 1 / (double)N;   // Scaling factor for the inverse FFT
```

```
    for (int i = 0; i < N; i++, DFT++)
        DFT->Im *= -1;          // Find the complex conjugate of the Frequency Spectrum

    DFT = pDFT;                 // Reset Freq Domain Pointer
    rad2FFT(N, DFT, x); // Calculate the forward FFT with variables switched (time & freq)

    int i;
    complex* x;
    for ( i = 0, x = pX; i < N; i++, x++){
        x->Re *= NN;    // Divide time domain by N for correct amplitude scaling
        x->Im *= -1;    // Change the sign of ImX
    }
}
```

Appendix A: Pseudocode

Section A.1: Variable affectations

You could describe variable affectation in different ways.

Typed

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

No type

```
a = 1
a := 1
let a = 1
a <- 1
```

Section A.2: Functions

As long as the function name, return statement and parameters are clear, you're fine.

```
def incr n
    return n + 1
```

or

```
let incr(n) = n + 1
```

or

```
function incr (n)
    return n + 1
```

are all quite clear, so you may use them. Try not to be ambiguous with a variable affectation

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Abdul Karim	Chapter 1
afeldspar	Chapter 43
Ahmad Faiyaz	Chapter 28
Alber Tadrous	Chapter 53
Anagh Hegde	Chapters 29 and 39
Andrii Artamonov	Chapter 27
AnukuL	Chapter 40
Bakhtiar Hasan	Chapters 9, 11, 14, 17, 19, 20, 22, 40, 41, 42, 47, 52 and 54
Benson Lin	Chapters 14, 39 and 44
brijs	Chapter 39
Chris	Chapter 15
Creative John	Chapters 49 and 51
Dian Bakti	Chapter 10
Didgeridoo	Chapters 2 and 43
Dipesh Poudel	Chapter 21
Dr. ABT	Chapter 55
EsmaeelE	Chapters 2, 29, 30, 39 and 50
Filip Allberg	Chapters 1 and 9
ghilesZ	Chapter 17
goeddek	Chapters 18 and 27
greatwolf	Chapter 5
Ijaz Khan	Chapter 29
invisal	Chapter 31
Isha Agarwal	Chapters 4, 5, 6, 7 and 8
Ishit Mehta	Chapter 5
IVlad	Chapters 16 and 28
Iwan	Chapter 30
Janaky Murthy	Chapter 6
JJTO	Chapter 9
Julien Rousé	Chapter 24
Juxhin Metaj	Chapters 2 and 30
Keyur Ramoliya	Chapters 23, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 45, 47, 48 and 50
Khaled.K	Chapter 39
kiner_shah	Chapter 12
lambda	Chapter 38
Luv Agarwal	Chapter 30
Lymphatus	Chapter 31
M S Hossain	Chapter 17
Malav	Chapter 33
Malcolm McLean	Chapters 4 and 39
Martin Frank	Chapter 21
Mehedi Hasan	Chapter 5
Miljen Mikic	Chapters 2, 28 and 39
Minhas Kamal	Chapters 12 and 46
mnoronha	Chapters 23, 29, 31, 32, 33, 34, 35, 36 and 45
msohng	Chapter 39
Nick Larsen	Chapter 2

Nick the coder	Chapter 3
optimistanoop	Chapters 29 and 33
Peter K	Chapter 2
Rashik Hasnat	Chapter 40
Roberto Fernandez	Chapter 12
samgak	Chapter 29
Samuel Peter	Chapter 3
Santiago Gil	Chapter 30
Sayakiss	Chapters 9 and 14
SHARMA	Chapter 30
ShreePool	Chapter 39
Shubham	Chapter 16
Sumeet Singh	Chapters 20 and 41
TajyMany	Chapters 12 and 13
Tejus Prasad	Chapters 2, 5, 9, 11, 18, 19 and 45
theJollySin	Chapter 17
umop apisdn	Chapter 39
User0911	Chapter 29
user23013	Chapter 9
VermillionAzure	Chapters 4 and 9
Vishwas	Chapters 14, 25 and 26
WitVault	Chapter 3
xenteros	Chapters 17, 29 and 39
Yair Twito	Chapter 2
yd1	Chapter 4
Yerken	Chapters 16 and 20
YoungHobbit	Chapter 29

You may also like