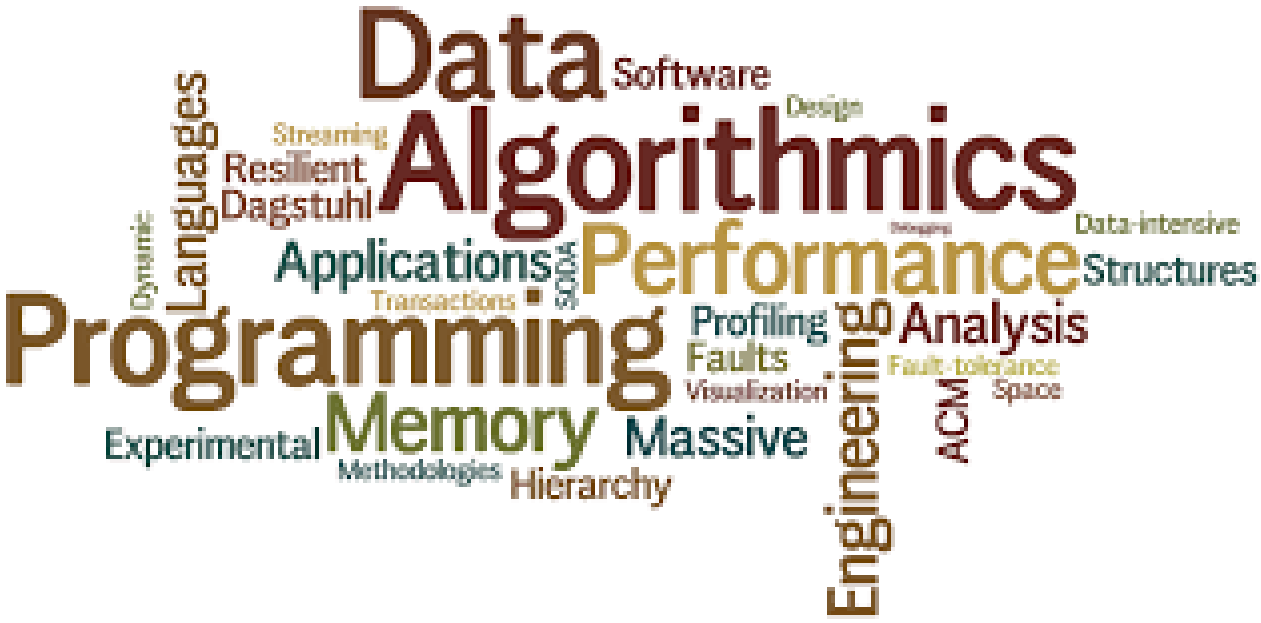
Data Structures and Algorithms

Akash Cherukuri - 190050009

Computer Science and Engineering

Mentor:- Prasanna Telawane



Contents

1 Introduction **4**
1.1 Prerequisites 4
1.1.1 Debugging your Algorithm - Stress Testing 4
1.1.2 Pseudo-Code 5

2 Algorithms **5**
2.1 What makes an Algorithm good? 6
2.2 Computing Runtimes 8
2.2.1 Asymptomatic Notation 9
2.2.2 Big-O Notation 10
2.2.3 Ω-Notation and Θ-Notation 11
2.2.4 Master Theorem 11
2.3 Types of Algorithms 12
2.3.1 Greedy Algorithms 12
2.3.2 Divide and Conquer Algorithms 14
2.3.3 Dynamic Programming 17

3 Data Structures **20**
3.1 Arrays 20
3.2 Singly-Linked List 21
3.3 Doubly-Linked List 22
3.4 Stacks 23
3.4.1 Implementing Stacks using Arrays 23
3.4.2 Implementing using Linked Lists 24
3.5 Queue 25
3.5.1 Circular Queue in Arrays 25
3.6 Tree 26
3.6.1 Traversal through a Binary Tree 27
3.7 Dynamic Arrays 28
3.7.1 Amortized Time Analysis 29
3.8 Priority Queues 30
3.8.1 Naive Implementation - Arrays 30
3.8.2 Implementation using Heaps 31
3.9 Hash Tables 33
3.9.1 Implementing using Arrays 33

3.9.2 Implementing using Linked Lists 34
3.9.3 Best of both - Hash Tables 34
3.9.4 Collision Handling 35
3.9.5 Time Calculation and Increasing Speed 35
3.10 Binary Search Trees 36
3.10.1 Naive Implementation using Hash Tables 36
3.10.2 Implementation using Lists and Sorted Arrays . . 36
3.10.3 Implementation of the Binary Search Tree 36
3.10.4 Balancing Trees - AVL Property 38

4 Algorithms on Graphs **39**
4.1 Representing Graphs 40
4.2 Computing Runtime 42
4.3 Graph Exploration 43
4.3.1 Connected Components 44
4.4 Directed Graphs 45
4.4.1 Linear Ordering 46
4.5 Dijkstra's Algorithm 47
4.6 $A^* - Algorithm$ 48

5 String Algorithms **50**
5.1 Pattern Matching Algorithms 50
5.1.1 Trie-Matching the Smaller Strings 51
5.1.2 More Efficient Trie-Matching 53

6 References **55**

1 Introduction

Data Structures are structures which are used to handle and store data. Data Structures may be present by default in a programming language, and if they aren't present in there, we can code them in. Having a good knowledge of data structures is essential as it allows the programmer to be able to manipulate the data as needed seamlessly and in an effortless manner.

An **algorithm** is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description using a flowchart.

1.1 Prerequisites

1.1.1 Debugging your Algorithm - Stress Testing

Creating and Coding of a better Algorithm than the existing one is challenging, and usually ends up having some mistakes in it. Usually, the programmer might forget some rare cases in which the algorithm fails, or worse, produces wrong results. It can also be seen that debugging of such a case is difficult, as the problem doesn't occur frequently. **Stress-Testing** can be employed to identify some of these bugs.

Stress-Testing is an exhaustive method of testing your bugs, in which the inputs are generated randomly. It is carried out by repeatedly giving the same inputs to both the algorithms (the older one and the faster, more efficient newer one) and checking whether the output produced by both the algorithms is the same or not. By the use of **cout** (in C) or **print** (in Python) statements, we can identify the cases in which our algorithm(s) fail, and debug them appropriately.

An Example of Stress Testing is shown Below:-

```
//f1 is the first algorithm, f2 is the second algorithm.
while(TRUE){
    x=rand()%100;
    cout << "Input:" << x << endl;
    res1 = f1(x);
    res2 = f2(x);
    if(res1!=res2){
        cout << "Error" << endl;
        cout << res1 << " " << res2 << endl;
        break;
    }
    else cout << "OK" << endl;
}
```

1.1.2 Pseudo-Code

Because data structures and Algorithms are universally applicable to all languages, it doesn't make sense to talk about the logic in terms of a single programming language. Neither can we just simply state the logic as it may not be understood by everyone.

Therefore, in order to express the logic, we use an informal high-level description of the logic. Further more, we can make it so that our descriptions follow the general coding syntax, without actually using a programming language.

This informal descrition of the logic in an alforithm is reffered to as **Pseudo-Code**. Pseudo-Code means "False Code", and it is called so because the description *looks* like a code, but is not.

2 Algorithms

As previously discussed, and Algorithm is just a set of instructions which can be followed to solve a specific set of problems. Being able to create, and code, efficient algorithms is a very important skill that every programmer must try to include in his/her skillset.

2.1 What makes an Algorithm good?

An Algorithm is said to be **"good"** or **"better"** than a different algorithm when both the algorithms are designed to perform the same tasks, but the first algorithm excels the second Algorithm in any of the following criteria :-

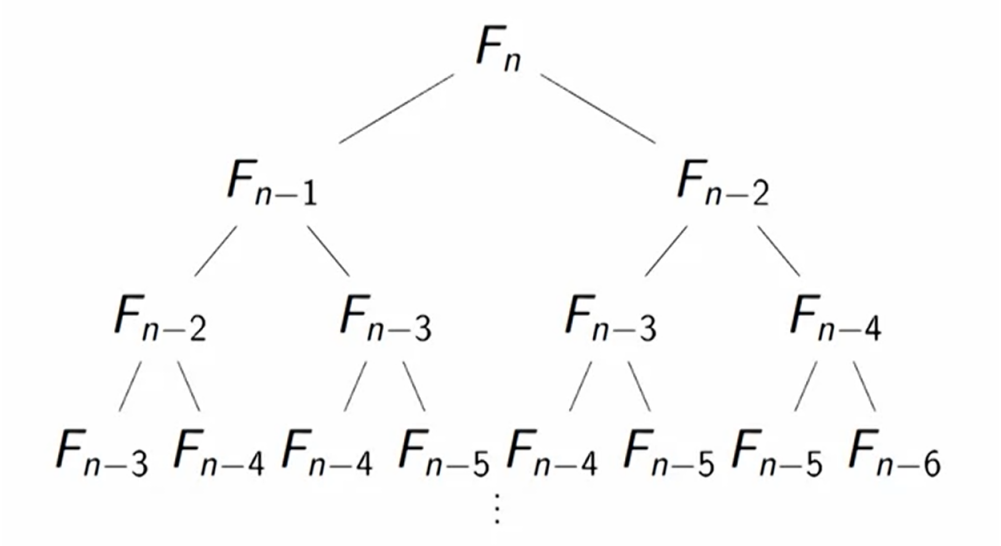
- It takes lesser time to arrive at the solution.
- It is computationally less heavy.
- It uses up lesser computational resources.

Example of a good Algorithm

Suppose we would like to create a program to compute the value of the n^{th} number in the Fibonacci Series. A trivial way of coding this in would be to have and iterated function as follows:

```
//Recursive version of the code
int Fib_Calc(int n){
    if(n<=1) return n;
    else return (Fib_Calc(n-1)+Fib_Calc(n-2));
}
```

This Algorithm for calculating the n^{th} number seems to be very simple and straight-forward, but is highly inefficient. For example, if we needed the 100^{th} number in the series, the program running on this code wouldn't be able to calculate it simple because of how inefficient this method is. To understand why this inefficient, lets try to create a Binary Tree model for the way in which the n^{th} number in the sequence, say F_n is calculated:



Notice how F_{n-3} is being calculated thrice unnecessarily? This is what causes our algorithm to be very tedious computational-wise and what causes it to be very time consuming. In order to overcome this, we could devise another algorithm which actually stores the values of Fn as it goes upto the required value of n as follows:

```
//New Algorithm for the task
int Fib_Calc(int n){
    int i=2;
    int val[n+1];
    val[0] = 0;
    while(TRUE){
        val[i] = val[i-1]+val[i-2];
        if(i==) return val[n];
        i++;
    };
}
```

This is a better algorithm when compared with our previously written recursive algorithm, as it calculates the values **QUICKER** then the first algorithm, and it also is **COMPUTATIONALLY LESS HEAVY**.

From this example, we can see understand the importance of studying and understanding and the power of being able to write good Algorithms. A good algorithm is what makes the difference between a

solution being given out in a million years, or a solution being given out as soon as we hit run. It is what makes the searching through trillions of webpages be done within seconds, and it is what is the most versatile, and arguable, on of the most important topics to be understood properly.

2.2 Computing Runtimes

We've been talking about an algorithm being faster than another and about an algorithm being more efficient than the other. So, the next logical question to be posed would be, "How long in general would the algorithm take to finish a computation?" This question opens up a whole new can of worms, as it cannot be answered very easily.

We can see why this would be a difficult question to answer as there are many factors which can cause the runtime of an algorithm to vary, like the difference in the computational power of different clients and the different number of computations to be performed when the inputs are different, to name a few. One might think of a workaround to count the number of lines of code being run.

However, this would be very vague as different lines of code would require different degrees of computational power. For example, a line involving addition of two numbers and another initialising an array of numbers would obviously different time.

IDEA: Assume that all the above issues affect the runtime by a constant factors, which we can ignore.

That is, a client who has a slow computer might cause the speed of computation of ALL processes to be doubled. This idea, however, is a little problematic as then runtimes of a second and an year would be considered the same as they differ by a constant multiple. This problem is solved by talking about runtimes in an **Asymptomatic Notation**.

2.2.1 Asymptomatic Notation

Asymptomatic Notation or Asymptomatic Method of talking of runtimes invloves thinking about how the runtimes scale with respect to the size of the input data. This makes sense, as when the input data is Large enough, an algorithm whose runtime scales as n^2 would be significantly worse than one which scales as $1000n$.

The maximum time that an algorithm must take can be defined by us, and from that we can calculate a good estimate of the maximum input size that the algorithm can compute, given that we know the **scaling** of the algorithm. **"Scaling"** refers to how the runtime depends on the input size.

General Scaling Comparisions

$$log_a n < \sqrt[n]{x} < n < nlog_a n < n^a < a^n$$

$$a \in \mathbb{N}$$

2.2.2 Big-O Notation

Big-O is a notation which takes the ideology of Asymptomatic Notation and gives it a mathematical definition. This notation allows us to be able to quantitatively descern *How Fast* our algorithm is.

Definition. Let f and g be two functions. "f is Big-O of g", or, $f(n) = O(g(n))$ when $\exists\ N, c \in \mathbb{Q}^+$ such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq N$.

We can see from the above definition that the Big-O notation tells us about the **Upper Bound** for the number of operations that would need to to be performed by the computer when following the algorithm. This notion has many advantages, some of them being:

- Clarifies growth rates for large sized inputs
- Clean notations for better representation and understanding
- Mathematically quantified $\implies$ Algebra is possible
- For large enough inputs, choosing a faster algorithm is easy.

Algebra of Big-O notation

1. Multiplicative Constants can be ignored
   $7n^2 = O(n^2)$
2. Smaller terms can be ignored
   $n^2 + 3n + log_a^n = O(n^2)$
3. $a, b \in \mathbb{Q}^+,\ a < b \implies n^a = O(n^b)$
4. $a, b \in \mathbb{Q}^+$ and $a > 0, b > 1 \implies n^a = O(b^n)$
5. $a, b \in \mathbb{Q}^+ \implies (logn)^a = O(n^b)$

By using the above algebra and our intuition, we can accurately talk about the upper bound of the runtime of the algorithm. Consecutive steps in the algorithm require us to add the Big-O functions, while those in a loop require us to multiply the Bog-O function of the steps **IN** the loop with the Big-O function of the looping mechanism itself. We then further simplify the Big-O notation and arrive at our Final Answer.

However, this noation of representing runtimes is not without faults. The faults of this notation are given below:

- Runtimes which differ by a constant factor cannot be distinguished by this notation.
- By definition, Big-O is asymptomatic in nature.
- It works well only for Large sizes of data. An Algorithm which follows n^3 (for example) is faster than 3^n for small sizes of data.

2.2.3 Ω-Notation and Θ-Notation

Ω-Notation is just like the Big-O notation, but it tells us about the lower bounds of the runtime.

Definition. Let f and g be two functions. "f is Ω of g", written as, $f(n) = \Omega(g(n))$ when $\exists\ N, c \in \mathbb{Q}^+$ such that $f(n) \geq c \cdot g(n)\ \forall\ n \geq N$.

All the algebra of Big-O is still valid for Ω with logic reversed. That is, instead of the smaller terms being ignored, we ignore the larger terms and the such.

Θ-Notation is similar to an equality in Algebra. It tells us that the functions increase at a similar rate.

Definition. Let f and g be two functions. "f is Θ of g", written as, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

For example, $3n^2 + 5n + 3 = \Theta(n^2)$

2.2.4 Master Theorem

This theorem is used to quickly find the run-times of an algorithm in terms of the Big-O notation, when the relations are in recursive format, $T(n) = aT(\frac{n}{b}) + O(n^d)$. The formula is as follows:

$$T(n) \begin{cases} O(n^d) & d > log_b^a \\ O(n^d logn) & d = log_b^a \\ O(log_b^a) & d < log_b^a \end{cases}$$

2.3 Types of Algorithms

An algorithm in development has different stages to it, and its development cycle mostly goes along the steps:

1. **Naive Algorithm:** This algorithm is derived from the definition of the problem statement. It's usually slow and requires optimization.
2. **Optimized Algorithm:** The naive algorithm is optimized by using some standard tools(discussed below) or by the usage of data structures and reorganizing the way in which steps are done in the algorithm.
3. **"Magic" Algorithm:** This is completely different from the above two, in the sense that it uses a unique insight into the problem, and requires a good understanding of the problem statement.

There are different ways of optimizing the Naive Algorithm, which are given below. However keep in mind that there is no Silver Bullet for this problem, as no method, that we know of, will work for all the infinitely many types of problem statements. With that being said, it is up to the programmer to decide which method would work, and this requires them to have a good grasp of the methods.

2.3.1 Greedy Algorithms

These classes of algorithms work out an optimised solution for a given problem statement by taking the most "greedy" step at a localised level, and then they repeat this till they reach the end. For example, a shortest path finding algorithm working in this way would choose the option which takes it closest to the destination, and then it repeats this till it reaches the solution.

Before we move further, we must define some terms so that it is easier to understand some core concepts of this class of Algorithms.

Sub Problem:

A Sub-Problem is a problem which belongs to the same problem-class as the original problem statement, and is smaller than the original problem statement.

In the previous path-finding example problem, finding a path from the end of the greedy step to the end would constitute as a sub problem.

Safe Step:

An option among the many present at any given point is considered to be "Safe", if there exists an optimized path from the end of this step to the end of the original problem.

A safe step may, or may not be intuitive in nature as that depends on the complexity of the problem statement. Therefore, it generally is a good practice to prove mathematically that the steps being taken by your algorithm are all safe.

Now that the definitions of a safe step and a Sub-Problem are clear, lets look at how a Greedy Algorithm can be coded to work:

1. **Reduce the given problem statement to a mathematical model.** This makes the following steps easier, and proving that the steps taken are safe into a mathematical problem, instead of one relying on intuition.
2. **Determine the Greedy Step.** This is a step which takes the algorithm closest to the end point of the problem, where the computation becomes so easy that it can be computed manually.
3. **Prove that the Greedy Step taken is Safe.**This ensures that there are no exceptions left out. This is critical, because if we finish coding an algorithm whose greedy steps aren't safe, we would need to scrap the entire thing.
4. **Ensure that once the Greedy Step is taken, the problem becomes a Sub-Problem.** If it doesn't either the logic of our greedy step is wrong or the algorithm can't be coded in this fashion.

2.3.2 Divide and Conquer Algorithms

As the name suggests, this class of Algorithms aims to solve a problem by dividing it into smaller **Sub-Problems**, solving each one of them, and then using the answers from all the sub parts to produce the final answer. Although it looks very similar to the Greedy Algorithm, ONE key difference they both have is:

- In Divide and Conquer algorithms, division occurs first and then the computation follows.
- In Greedy Algorithms; computation occurs, then forming a sub-problem follows automatically, and then this loop is repeated till we reach our destination.

Example of some problem-statements where this methodolgy is used would be:-

- Binary Search
- Polynomial Multiplication
- Sorting Algorithms

We shall look at Polynomial Multiplication in detail now.

Polynomial Multiplication

The problem statement is straightforward; Given two polynomials of degrees n and m, what is their product?

Naive Algorithm

Let us first define that the input of our problem to be n (the max degree between the two polynomials), followed by the co-efficients of the polynomials from x^n.

Let the polynomials be:

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_o$$

$$B(x) = b_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + b_1x + b_o$$

And let the product of these two polynomials be:

$$P(x) = p_{2n-2}x^{2n-2} + p_{2n-3}x^{2n-3} + ... + p_1x + p_o$$

We can quite easily calculate the value of each p_i by the following formula:

$$p_i = \sum_{j=0}^{i} a_j b_{i-j}$$

A naive Algorithm can be set up from the formula. But let's look at the time it would take this algorithm to actually compute the product. For every p_i, we would need to perform addition at least n times; and this would need to be repeated for $2n$ tries to calculate all the co-efficients.

Therefore, the run-time of out naive algorithm is $O(n^2)$.

Divide and Conquer Algorithm

To reduce the run-time, let's try to implement a divide and conquer strategy. Let's split the given polynomials $A(x)$ and $B(x)$ as follows:

$$A(x) = C(x) \cdot x^{n/2} + D(x)$$

$$B(x) = E(x) \cdot x^{n/2} + F(x)$$

(if n is not even, just pad the polynomial by adding zero terms)

$$A(x) \cdot B(x) = \{C(x) \cdot E(x)\}x^n + \{C(x) \cdot F(x) + D(x) \cdot E(x)\}x^{n/2} + D(x) \cdot F(x)$$

Notice how we've broken the problem down in **FOUR** sub problems, half as big as the original. The four sub-Problems are the product of the polynomials in the RHS of the above equation. Keep in mind that the degree of each of the sub-problem is $(n/2) - 1$

Let's look at the Pseudo-Code for this Algorithm:

```
Function Multiply(A, B, n, ai, bi):
    res = array[2n-1];
    if n==1 :
        res[0] = A[i] * B[j];
        return res;
    //Calculating D(x)*F(x)
    res[0, ...  n-2] = Multiply(A, B, n/2, ai, bi);
    //Calculating C(x)*E(x)
    res[n, ...  2n-2] = Multiply(A, B, n/2, ai+n/2, bi+n/2);
    //Adding the value of middle term
    CF = Multiply(A, B, n/2, ai+n/2, bi);
    DE = Multiply(A, B, n/2, ai, bi+n/2);
    res[(3n/2)-2 ...  , n/2] += CF + DE;
    return res;
```

Let's carry out an analysis for the time taken. Assuming that n is a power of 2, (even if it isn't we can the pad the polynomials), the number of sub-problems that we will have at depth of division i is equal to 4^i, because each division further divides each product into four smaller products.

The work done for size n in this algorithm is kn, because the calculated values of the co-efficients need to be assigned to the the array res, as seen in the above pseudo-code.

Therefore, work done at a given depth i is given by:

$$4^i \cdot k(\frac{n}{2^i}) = k \cdot 2^i n$$

But the value of i is log_2^n, and this would mean work done at last step is $k \cdot n^2$. So this would mean that this algorithm is also not better than the naive algorithm. However, as we shall see next, there is scope for improving this algorithm.

Optimized Divide and Conquer Algorithm or Karatsuba's Approach

The reason why our algorithm is still slow is because 4 sub-problems are created in each step. This approach aims to reduce the number of sub-problems created by some clever mathematics.

The second term in the above algorithm's method is $C(x) \cdot F(x) + D(x) \cdot E(x)$. However, we can write it more efficiently as:

$$C(x)\cdot F(x)+D(x)\cdot E(x) = [C(x)+D(x)]\cdot[E(x)+F(x)]-C(x)\cdot E(x)-D(x)\cdot F(x)$$

This might seem cumbersome, but notice that the problem is divided into only 3 new sub-problems, namely, $[C(x) + D(x)] \cdot [E(x) + F(x)]$, $C(x) \cdot E(x)$ and $D(x) \cdot F(x)$. This is a game changer, as the work to be done now becomes $3^{log_2^n}$ or approximately $n^{1.58}$. We can truly appreciate how much better this is for very large values of n.

2.3.3 Dynamic Programming

This is yet another method of solving problems and optimizing naive algorithms. This is not a class of Algorithms per se, but rather is a clever application of logic and the constraints which are presented by the problem statement.

As Dynamic Programming is not a concept, but rather an idea, we can understand it via the following example:

Problem Statement. *You are a cashier, and as part of your job is returning change. You would like to know the least number of coins needed for the transaction.*

One of the first ideas that we would have to solve this problem would we to use a Greedy Algorithm. Like, find the largest coin which is smaller than the change. Use it, and repeat. This intuition isn't wrong, and it does in fact yield correct answers as shown:

- Values of coins that we have: 1, 5, 10, 25
- Change to be returned: 42
- Using Greedy Algorithm, we get that the coins are 25, 10, 5, 1, 1.

However, It doesn't work for all cases as we can see below:

- Values of coins that we have: 1, 5, 10, 20, 25
- Change to be returned: 40
- Using Greedy Algorithm, we get that the coins are 25, 10, 5.

However, notice that using two 20 coins is more efficient! Greedy Algorithm doesn't work because we haven't proved that using the highest possible denomination is a Safe Move.

Solving this problem using Dynamic Programming
Let us first define the terms Mathematically;

- Optimal number of coins for Change m is given by $C(m)$
- There are n coins.
- Value of i^{th} coin is given by v_i
- The coins are sorted in increasing order of value

By observing the problem statement, we can easily develop this recursive statement:

$$C(m) = min\{C(m - v_i) + 1, C(m - 1)\} \forall v_i \leq m$$

That is, the optimal solution can be obtained by backtracking. This is why this works: Suppose that the optimal number of coins to return a change of x is n. Obviously, the optimal number of coins to return a change of $x + v_i$ is $n + 1$! The above relation is just this logic in reverse.

So the idea that we might get would be to write a recursive algorithm which branches out from $C(m)$; which can be shown by the following Pseudo-Code:

```
//Pseudo-Code for number of Coins
Function Coins(m):
    for v_i ≤ m:
        rec = min(Coins(m − v_i) + 1, Coins(m − 1));
    return rec;
```

However, notice that a certain value of Coins(x) will be calculated many times over. Re-calculating this value many times will slow down the algorithm. Also, recall that this situation is highly similar to the Fibonacci Numbers problem that we had encountered earlier.

So we can use the same solution that we had used for the Fibonacci Series here as well. That is, in order to prevent recalculations, we create an array which stores the values of the optimal number of coins for a given number of change, and then values can be pulled directly from this array to save computation time.

The Pseudo-Code for this idea is:

```
//Pseudo-Code for Better Algorithm
Function Fast_Coins(m):
    val = arr[m+1]; i=1;
    val[0] = 0;
    while i!=m:
        for v_i ≤ i:
            rec = min(val[i − v_i] + 1, val[i − 1]);
            i++;
    return val[m-1];
```

This is the philosophy of Dynamic Programming. Instead of calculating the values over and over again in a recursion; we store the required values and look them up whenever needed.

However, this might rise a funny question in our heads, "Why is this called *Dynamic Programming* in the first place?" The answer has a historic significance regarding its inception into Computer Science. The concept of Dynamic Programming was introduced by Richard Bellman in the 1950s while he was working on an Air Force Project.

However, when he introduced this, everyone dismissed this idea as it seemed non-sensical. We can see why they thought so as well, as it seems illogical to compute all the values till the required answer. However, we know mathematically that this wasn't the case! Richard wanted to hide the fact that he was really doing mathematics from the Secretary of Defense. The following quote from him drives the fact home:

"..what name do I chose? I was interested in planning, but planning is not a good word for various reasons. I decided to therefore use the word 'programming' and I wanted to get across the idea that this was dynamic.."

3 Data Structures

As we've discussed earlier, Data Structures are structures which are used to handle and store data. Data Structures may be present by default in a programming language, and if they aren't present in there, we can code them in. Having a good knowledge of data structures is essential as it allows the programmer to be able to manipulate the data as needed seamlessly and in an effortless manner.

3.1 Arrays

- Arrays are a type of Data Structures in which data is stored in partitions of **Equal Size**.
- The partitions must have **continuous indexing**. The starting index can be 0, 1, or it can be defined by the user.
- It has **Constant Time access**, meaning that the time taken for accessing the array elements takes $O(1)$ time.

Also, the address of any element in the array can be deduced from the address of the array as follows:

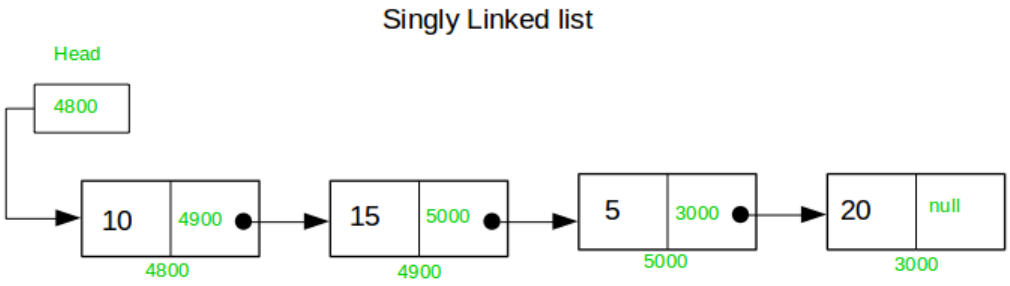
$$Elem\_Addr = Array\_Addr + (SizeofPartitions) \cdot (i - index\_of\_first)$$

Arrays also have the functionality to be **Multi-Dimensional**. There are two ways in which this may be done:

- **Row-Major:** The elements are filled in Row-after-row, and we go to the next column after the row has been filled.
- **Column-Major:** The elements are filled in Column-wise, and we move to the next row once all the columns have been filled.

3.2 Singly-Linked List

A singly linked list is generally defined by the programmer, and it can be easily done recursively. A singly-linked list has **head pointer**, which points to a **node**. A node consists of variables to store data and a pointer which either points to the next node or points to None, if it is the end.



A variation of the singly-linked list has a **Tail Pointer**, which points to the last node in the entire structure. The advantage of having a tail-pointer is to quickly get to the element at the end of the linked list. Also keep in mind that moving from one element to another is possible **ONLY** in the forward direction.

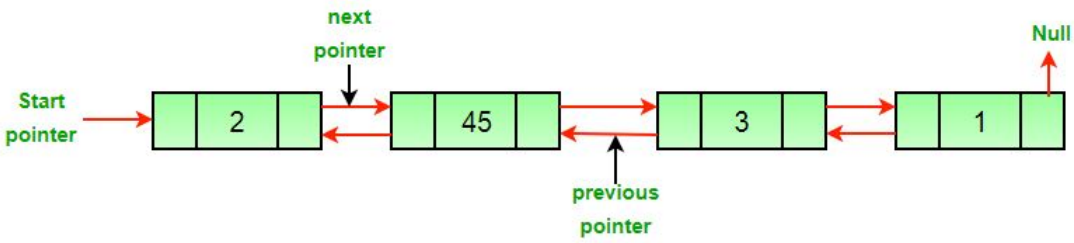
Available functions and their run-times:

Function	Functionality	Runtime
pushfront(node)	Adds the node to the front	$O(1)$
top()	Returns the top node	$O(1)$
pop(i)	Removes the i^{th} node	$O(n)$
append(node)	Adds the given node to end	$O(n)$
find(node)	Searches for the given key, returns True if found	$O(n)$
add(node, key)	Adds the node after the given key	$O(n)$

Notice that the runtime for the *append*(*node*) function can be reduced from $O(n)$ to $O(1)$ by using the tail pointer.

3.3 Doubly-Linked List

This is similar to the Singly-linked list, but a doubly-linked list also has a pointer which points towards the previous element in the sequence of nodes. This can be seen in the following schematic:



The advantages of having two pointers is evident: We can travel backwards from a node, instead of having to get to the node from all the way from the front. This makes some operations easy, and adding in a tail pointer makes this data structure very powerful.

3.4 Stacks

A Stack is an **abstract** data type. It is similar to a "stack" of books, and the functions in this data type are similar to how a stack of books can be manipulated in real life.

Function	**Functionality**
push(node)	Adds the node to the front
top()	Returns the top node
pop()	Removes the last node
empty()	Return True if the stack is empty

Stacks are also reffered to as **Last-in-First-Out** or **LIFO Queues**. As they are abstract, they may be implemented as follows:

3.4.1 Implementing Stacks using Arrays

It can be seen quite easily how a stack can be implemented using an array. However, one major constraint in using arrays is that **the maximum size of the stack should be known.** This is because the size of an array cannot be modified once it has been allocated. (A work-around would be to use Dynamic Arrays, but more on that later.)

The pseudo-code for the above functions would be as follows:

```
1     def Push(node new):
2         if free = capacity :
3             return False;
4         val[free] ← new;
5         free ← free + 1;
6         return True;
7
8     def top():
9         return val[free - 1];
10
11    #Here, Val is the array storing the nodes
12    #free is an integer showing where the next free
       space is
13
```

```
14      def pop():
15          if empty():
16              return False;
17          val[free - 1] ← None;
18          free ← free - 1;
19          return True;
20
21      def empty():
22          if val[0] = None:
23              return True;
24          else:
25              return False;
```

We can clearly see from the above code that stacking fails if the capacity is exceeded. Also, notice that all the functions take $O(1)$ time to complete.

3.4.2 Implementing using Linked Lists

Linked lists have an advantage over arrays because the size is indefinite. However, keep in mind that we are using a singly-linked list with a tail pointer. If the tail pointer had not been present, the operations would've become $O(n)$.

Pseudo-code is as follows:

```
1       def push(node new):
2           #Changing pointer of last node
3           ((tail → last_node).next) ← new;
4           #changing the tail pointer
5           tail ← new;
6
7       def top():
8           return (head → first_node);
9
10      def pop():
11          if empty():
12              return False;
```

```
13          else:
14              head ← head.next;
15              return True;
16
17      def empty():
18          if head = Null;
19              return True;
20          else:
21              return False;
```

3.5 Queue

This is similar to a real-life queue, in the fact that the people are put into a line and the first one in is the first one out; **FIFO Queue**. This is also an abstract data type. The functions in this are similar to that of a Stack, so we will not delve into much detail about the pseudo code here.

Function	**Functionality**
push(node)	Adds the node to the front
top()	Returns the top node
pop()	Removes the **FIRST** node
empty()	Return True if the stack is empty

3.5.1 Circular Queue in Arrays

The general pseudo code for the `pop()` in array would be:

```
1       def Pop():
2           result ← val[0];
3           i ← 1
4           while i < capacity:
5               val[i-1] ← val[i];
6               i ← i+1;
7           return result;
```

However, this would take $O(n)$ order of time to be done. A faster way is to use a **HEAD** and a **TAIL** pointer to point where the queue starts and where the queue ends.

```
1     def Pop():
2         result ← val[tail];
3         Delete ← val[tail];
4         cycle(tail);
5         return result;
6         #Cycle() increases the value of tail by one
       and if tail = capacity, it makes it's value 0
```

This method makes the function to follow $O(1)$ time, and is thus faster.

3.6 Tree

As we all know, trees are used extensively in the representation of data. They consist of a node, and contain subsequent smaller branches which lead downwards. Formally in computer science, trees are recursively defined as follows:

`Tree is either empty OR points to nodes with smaller trees.`

Some terms are to be defined before we proceed further:

- **Root:** The topmost node of the tree
- **Child:** Immediate node(s) below the current node
- **Ancestor:** All the modes above the current node
- **Descendants:** All the nodes below the current node
- **Sibling:** Nodes with the same parent
- **Leaf:** Nodes with no children
- **Level:** (Depth from the root) + 1
- **Height:** (Max distance from the leaves) + 1
- **Forest:** Collection of trees

A **Binary Tree** is a tree in which each node has a maximum of two sub-nodes. A binary tree is very simple, yet is VERY powerful.

3.6.1 Traversal through a Binary Tree

As the name suggests, traversal means travelling. This topic refers to how we may travel through the tree and obtain the data in the nodes for further usage. The two main types of traversal are:

1. Depth First
2. Breadth First

Depth First Traversal

This type of traversal has the program running through the binary tree through each of the consecutive branches. The pseudo-code for a function which prints elements traversing in this way would be:

```
1   #Node is a class with the stored integer,
    pointers to the left and right side.
2
3   def PrintTreeElem(node elem):
4       if left is not NULL:
5           PrintTreeElem(left);
6       print stored;
7       if right is not NULL:
8           PrintTreeElem(right);
```

Notice that we are printing the elements of Left node first, then the node itself, then the right node. This manner is called **InOrder**. If we print the node itself first, it is called **PreOrder**, and if we print it last, it is called **PostOrder**.

Breadth First Traversal

This type of traversal implies that we are going to be printing out all the elements at the same level from the root node first, and we then start the process again with the nodes at the next level.

The Pseudo-code for Depth First Traversal is as follows:

```
1     def Breadth(node root):
2         queue q;
3         q.add(root)
4         while q not empty:
5             write q[0], q.pop()
6             q.add(left), q.add(right)
```

Also, understand that this works not only for a binary tree, but also for a general tree. The pseudo code for a general tree would be:

```
1     def Breadth(node root):
2         queue q;
3         q.add(root)
4         while q not empty:
5             write q[0], q.pop()
6             for subnode of q[0]:
7                 q.add(subnode)
```

This is how breadth first traversal works. We can use other forms of storage such as an array, or a linked list in place of a queue.

3.7 Dynamic Arrays

The arrays which we have discussed earlier are "static" in nature. That is, they cannot be modified with regards to their size. Once their size has been set, they become fixed. A logical way around this to make a "Dynamic" array.

These aren't necessarily abstract, and they are implemented in most languages. For example, Dynammic arrays in C++ are included in the `#include <vector>` standard library, and in python we only have a Dynamic array.

The append function, in particular needs modification as we can see in the pseudo code below. The size of the array is doubled, and the elements are copied.

```
#Dynamic array for storing integers
    def append(int x):
        if head < capacity:
            val[head] ← x
        else:
            n_val[2*capacity]
            n_val[0, 1.. (capacity - 1)] ← val
            n_val[capacity] ← x
            capacity ← 2*capacity
```

3.7.1 Amortized Time Analysis

We can easily look at the code and say that the time taken for appending the new element is of $O(n)$ order. This is because all the elements have to be copied over to the new array. This seems wrong, as an array, by definition, must have **Constant Time access**.

Are we really right in thinking about the time to be $O(n)$ order? We can look at an **Amortized** analysis, or in more common terms, average analysis of the time.

Amortized Analysis tells us to take the average of the time taken for n consecutive operations. For the appending function in the dynamic array, it shall be:

$$\frac{\sum_{i=0}^{i=(n-1)} O(1) + O(n)}{n} = \frac{O(n)}{n} = O(1)$$

Therefore, we can see that although the max time taken for appending a new integer is $O(n)$, on average, dynamic arrays still have Constant Time Access.(They wouldn't be called arrays otherwise)

3.8 Priority Queues

A priority queue is an abstract data type in which every element is assigned a **priority**, and the ones with the highest priority are extracted first. A real-life analogy to this would be how some airplane tickets have "classes", and the classes with the higher priority are allowed to board first.

The elements in a priority queue must be akin to elements in a bag. The elements need not necessarily be ordered inside the bag, but the following functions must be performable.

Function	**Functionality**
$insert(node, priority)$	Inserts a node into the bag with the given priority
$extract()$	Returns the node with highest priority
$pop()$	Deletes the highest priority element

The time taken for the functions is our indication of how good the data type has been implemented. We shall see some Naive Implementations and then a full fledged implementation of the Priority queues.

3.8.1 Naive Implementation - Arrays

We could easily use a dynamic array to store a priority queue. The first element of the dynamic array stores the element with the highest priority.

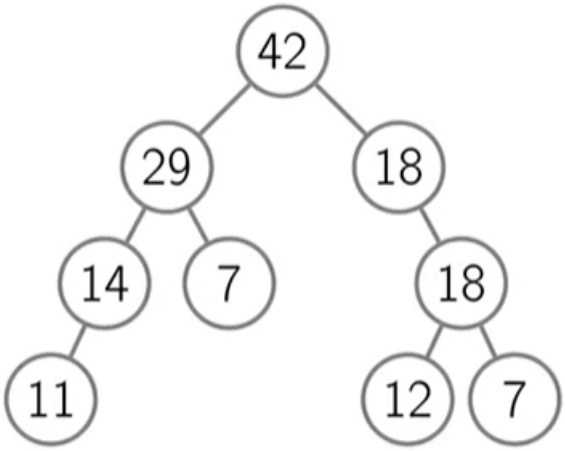
The pseudo-code for the above two functions would be as follows:

```
1    def extract():
2        return val[0]
3
4    def insert(node, priority):
5        for i from 0 to n:
6            if (val[i].pri < priority):
7                val[i, (i+1)..] ← val[(i-1), i..]
8                val[i-1] ← node
9                return
```

However, notice that the runtime for $insert(node, priority)$ is of the order of $O(n)$, because in the worst case scenario, all the elements would need to be moved forward. The next type of implementation would take care of this issue.

3.8.2 Implementation using Heaps

A heap is a binary tree in which the left and right sub-nodes of a node are smaller than or equal to the parent node. We can easily see that a heap could be used to represent a priority queue, with the root being the element with the highest priority.



Before we talk about the implementation of the above functions, let us first discuss the ways in which a wrong heap can be corrected. For example, in the above tree, if 11 and 42 were replaced with each other, we would need to have a method to correct the heap. $ShiftUp(node)$ and $ShiftDown(node)$ are the functions implemented for this purpose.

```
    def ShiftUp(node):
        if node.parent.pri < node.pri:
            swap node and node.parent
            ShiftUp(node.parent)
        else:
            return
    #ShiftDown will be implemented in the similar
  way
```

Now that these both functions are defined, let's discuss how *pop*() function would work. Well, instead of directly deleting the root and being left with two sub-trees, we swap the place of the root with any random leaf of the tree, and then employ $ShiftDown()$ for the new root. The pseudo-code for this would look like:

```
    def pop():
        temp ← root
        while temp.left is not null:
            temp ← temp.left
        #temp is now the left most leaf
        root ← temp
        #replace root with temp
        ShiftDown(root)
        return
```

This is much better than the dynamic array implementation because this takes $O(height)$ time to compute, compared to $O(n)$. However, this requires our tree to be somewhat branched out, because that would enable the tree to store the maximum number of nodes with the minimum height.

Definition. A binary tree is said to be complete if all the levels of the tree are completely filled; except possibly the last one. The last level of the tree may be incomplete, but the present children must be filled from the left to the right.

Therefore, by modifying our codes appropriately to keep the binary tree complete, we can have the above functions run with the runtimes of $O(log_2^n)$, which is much better than the linear implementation that we had for the dynamic Arrays.

3.9 Hash Tables

Problem Statement. *You are the manager of a server, and you want to know the number of different IPs connecting to your server in the previous 60 minute interval, and also maintain a count of how many times the IP has connected.*

This is a fairly common problem faced by servers, which we shall attempt to tackle now. IPs are generally given as $a.b.c.d$ where $0 \leq a, b, c, d \leq 255$. For example, 101.23.40.120 is a valid IP.

Each number is less than 256, meaning each number can be stored in 8 Bits. This implies that the IP as a whole can be stored in 32 Bits. Let's try to tackle this problem using different data structures and see why the need for a new data structure has risen.

3.9.1 Implementing using Arrays

Because each IP takes up 32 Bits, all IPs can be stored in an array whose size is 2^{32}. The mapping from the IP to the array address (n) can be done as follows:

$$n = d + c \cdot 2^8 + b \cdot 2^{16} + a \cdot 2^{24}$$

All the values in the array will be initialized to 0. Everytime an IP accesses the server, we would increase the value of the corresponding number in the array by one. This is good, as all the operations would take a $O(1)$, constant amount of time, which is what we need.

However, one BIG disadvantage of using this would be the memory it requires. We need to store a LARGE number of values without actually using most of them, because not all the IPs would connect to our server(obviously). This problem is worsened if we use IPv6, increasing the memory needed astronomically.

3.9.2 Implementing using Linked Lists

Because of the draw backs of arrays as show previously, we might be tempted to use Linked Lists. The nodes of the linked list would be storing the IP and the number of times that IP has connected to the server.

Pseudo-code which would be needed to run every time a new connection is detected would be:

```
1    #Head refers to the head pointer
2    def update(IP):
3        temp ← head
4        while temp.next is not NULL:
5            if temp.IP = IP:
6                temp.count++
7                return
8            else:
9                temp ← temp.next
```

That is, we need to check through the ENTIRE linked list to see if the IP has been repeated already. This takes $O(n)$ time, and needs to be dome everytime. So although Linked lists saves us space, we lose the speed at which arrays were able to compute the values.

3.9.3 Best of both - Hash Tables

Simply speaking, Hash-Tables are arrays containing lists. We take all the IPs and we mathematically divide them into groups of approximately equal size. This is usually done by a function called the $Hashing - Function$.

The Hashing function takes the value of the IP and it returns the index of the array where it would be located in. Because the array index is obviously smaller than the total number of IPs, some IPs would return the same index when the Hashing-Function is passed. This is called as **Collision**.

3.9.4 Collision Handling

This refers to how Collisions are handled. In this case, if there is already a non-empty list at the array index, we scan this list. If the IP is already present in the list, it means that the IP has reconnected, so we can increase the count by one. If it isn't present we can append this IP with it's counter as one.

However, if the array element is empty, simply create a new empty list at that index and append this IP with it's counter as one. We can clearly see that this would be faster than only using arrays or lists.

3.9.5 Time Calculation and Increasing Speed

The time taken to look up whether an IP has connected would involve the following steps:

1. Calculate the Array index from the Hashing Function - $O(1)$
2. Look-up the array index - $O(1)$
3. Scan through the list and check whether the IP is present - $O(length-of-list)$

Therefore, the total time taken for looking up the IP would depend on the number of elements in the list. This is faster than arrays and list which would take $O(n)$ time. However, if our hashing function is bad, then all the IPs would be grouped into a sinle list which makes all the developments that we've done so far worthless.

Therefore, our Hashing function should distribute all the IPs uniformly so that the possibility for the is minimized. This involves multiple factors, and the definition of our Hashing Function determines how efficient our Data structure is, overall.

3.10 Binary Search Trees

Suppose that we would like to be able to do a "Local" search for an element. That is, we would like to return all the elements withing a corresponding range; or the nearest neighbours of the given element. We also need to be able to insert new elements and remove already existing elements in the data structure.

3.10.1 Naive Implementation using Hash Tables

As we've discussed previously, Hash tables make inserting and deleting new elements fairly simple. However, the range search and finding the nearest elements would prove to be difficult as the elements which we are looking for need not be in the same part of the main array.

This would force us to search through the entire data structure and thus, would take a time of the order of $O(n)$. This is very much not ideal, and we thus look upon other data structures for this reason.

3.10.2 Implementation using Lists and Sorted Arrays

Sorted arrays work better in this case, as we can use **Binary Searching** for employing the Ranged Search and the Nearest Neighbours functions, as it takes only $O(log_2^n)$ order of time. However, it fails for inserting new elements because all the elements need to be shifted inorder for the array to remain sorted.

Linked Lists are immediately out of the picture, because we cannot aply a Binary search on them. This causes us to manually search the entire data structure, again taking $O(n)$ time.

However, these gripes feel similar to the ones we had when the need for trees had arisen. We shall therefore look at how a Binary tree is more efficient than all the data structures that we have studied so far.

3.10.3 Implementation of the Binary Search Tree

As the name suggests, a Binary Search Tree is a Binary tree with a special property that the left child of any node is smaller than the

parent node and the right child of any node is greater than or equal to the parent node.

The pseudo-code for the above mentioned function, namely, *RangedSearch*(), along with some other important functions is shown below.

```
    #next() gives the next biggest element stored
    def next(node N):
        if N.right is not NULL:
            temp ← N.right
            while temp.left is not NULL:
                temp ← temp.left
            return temp
        else:
            #return nothing if N is the largest
            if N.parent is NULL:
                return
            temp ← N.parent
            next(temp)

    #Using next(), we can write RangeSearch()
    #val is a list storing the results
    def RangeSearch(node low, node high):
        if low is in tree:
            val.append(low)
        low ← next(low)
        while low less than high:
            val.append(low)
            low ← next(low)
        return val
```

The best part of doing stuff in this way that the run time of our algorithm here is dependent only on the height of the tree, that is, if our tree is balanced, the runtime is $O(log_2^n)$. We can also quite easily see that the time taken for adding and deleting a node from this tree

would also be dependent on the height of the tree, thus making it's runtime logarithmic as well.

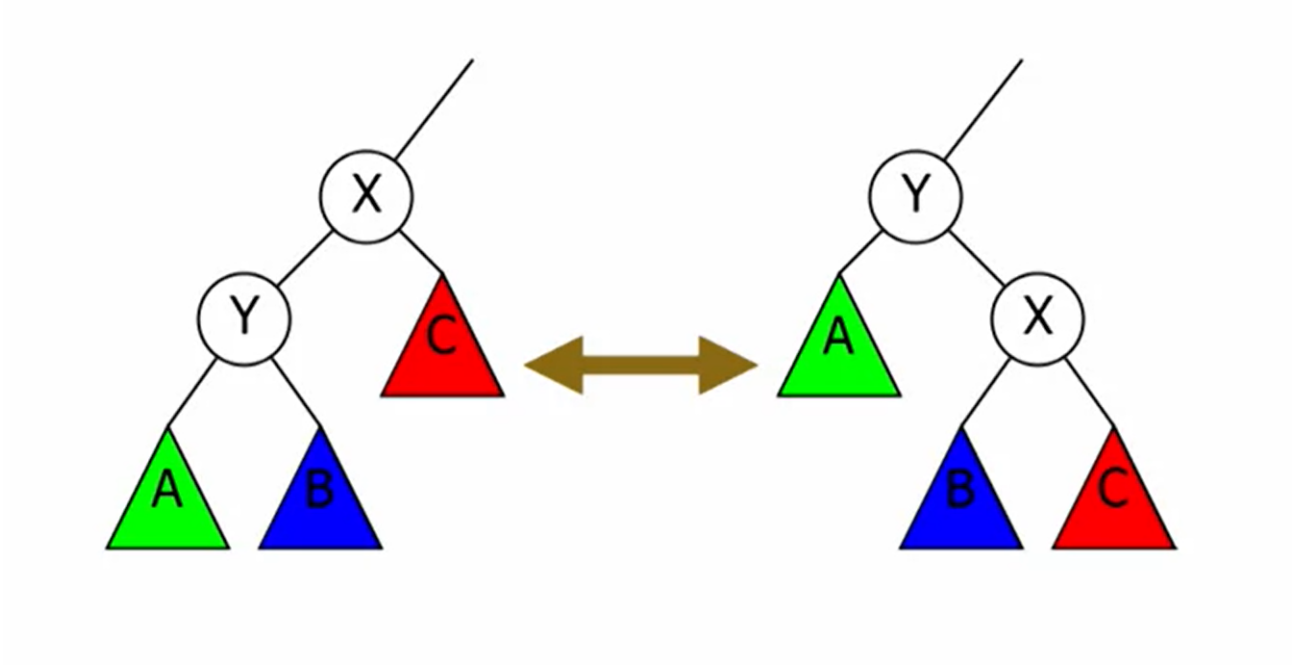
Therefore, in order to have a good run-time, our trees need to be shallow. Do not confuse the properties mentioned after this with the properties discussed with heaps, as they are two completely different data types.

3.10.4 Balancing Trees - AVL Property

Definition. A Binary Tree is said to be AVL if the heights of both the children of any node(if the node has two children) differ atmost by 1.

We can very clearly see that Binary trees which satisfy the AVL property are going to be shallow. Inorder for AVL property to be satisfied, we need to define some functions for us to be able to modify the Binary tree.

The functions which we would need to define would be $Rot\_L(node)$ and $Rot\_R(node)$, and their functionality would be as shown in the following figure:



Implementing this feature is pretty simple, we need to check the tree

after every insertion or deletion for the AVL Property. If the AVL property is not satisfied, we can use the rotate functions accordingle.

The heights need not be calculated on the go, but rather, they can be stored in the node itself. This would further speed up the process and make the entire tree faster. However, note that whenever any modification to the tree is done i terms of its structure, like, rotations, insertions and Deletions, all the values of height need to be recomputed. The pseudo-code for this would be as follows:

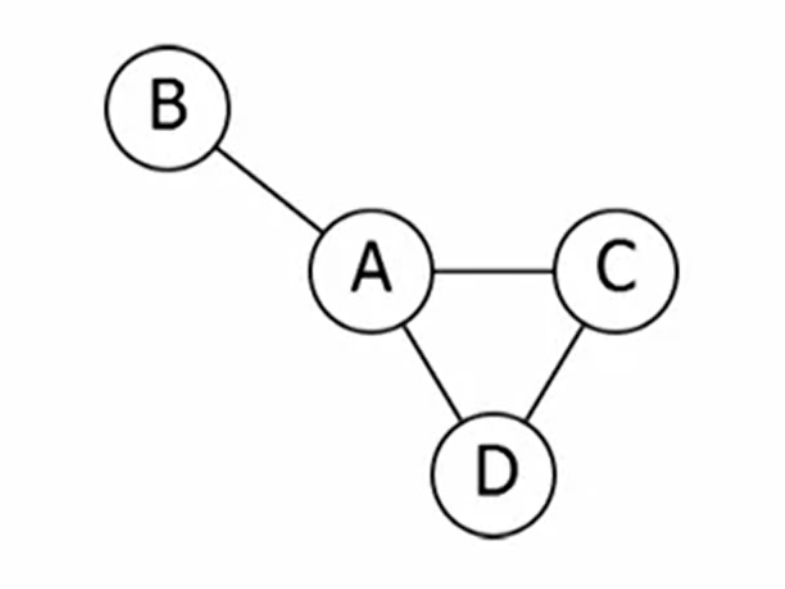
```
1    #Note that Dynamic Programming can't be used
2    #As nothing is repeated necessarily
3    def Calc_Ht(node root):
4        height ← 1 + max(Calc_Ht(root.left),
   Calc_Ht(root.right))
5        return height
```

4 Algorithms on Graphs

Definition. An Undirected Graph is a collection of vertices (V), and Edges (E) which connect pairs of Vertices.

The above given definition is how graphs are defined formally. The vertices may be represented by points, and the edges may be represented by lines connecting the points. Some examples of Graphs in practical usage would be how social networks are represented and the such.

Edges generally connect two different Vertices but this need not always be the case. An edge which connects a vertex to itself is called as a **"Loop"**. Similarly, We can have multiple vertices between two vertices, but they are not considered generally.



The above image is an example of a graph in which the vertices are A, B, C and D and the edges are the lines connecting the vertices. Notice that all the vertices need not be connected with each other; and there may be some vertices which are not connected to any vertices at all!

4.1 Representing Graphs

We have seen in the above image how graphs may be represented VISUALLY. However, they also need to be represented in such a manner that operating with them is easy, fast and efficient. Some of the ways in which Graphs can be represented is as follows:

1. Edge List
2. Adjacent Matrix
3. Adjacent List

Edge List is pretty self-explanatory. All the edges are stored in a list, and when required, they can be looked up from the list.

Adjacent Matrix is an $|V| \times |V|$ matrix in which all elements are initialised to Zero. The cell $(1, 2)$ is marked as 1 if there id an edge between Vertex 1 and Vertex 2. Notice that this array must be Symmetric.

Adjacent List is also pretty self explanatory. For every Vertex, we store a corresponding list as to which vertices are adjacent to it. Therefore, if we have $|V|$ vertices, we would have $|V|$ corresponding lists as well.

Storage Method	**Edge?**	**List Edges**	**List Neighbours**
Edge List	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
Adjacent Matrix	$\Theta(1)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|)$
Adjacent List	$\Theta(deg)$	$\Theta(E)$	$\Theta(deg)$

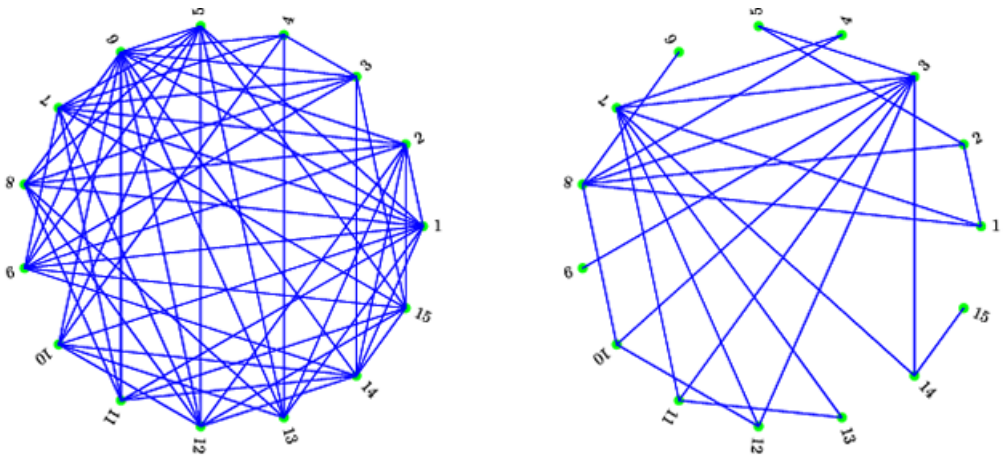
We will be looking at an Adjacent list From here on, as it's easy enough to implement and it is more efficient in the broader sense.

4.2 Computing Runtime

We've previously talked about how runtimes of an algorithm may be computed. We've been using the $Big - O$ Notation ever since, and it has worked quite well till now. However, ALL the runtimes that we've discussed till now only rely on ONE variable, the size of the input, which makes comparision easy. That's no longer the case here.

In graphs, the run time of the algorithm is going to depend on both $|V|$ and $|E|$. This makes it tough because we cannot really compare two algorithms whose runtimes are $O(|V|^{\frac{3}{2}})$ and $O(|E|)$. Thus, we define another classification of the graphs:

1. **DENSE:** A graph is said to be Dense, if mostly all the vertices are interconnected with each other by a vertex. In a dense graph, $|E| \approx |V|^2$.
2. **SPARSE:** A Graph is called Sparse, if the weaving isn't very complex. Here $|E| \approx |V|$.



(a) dense graph

(b) sparse graph

4.3 Graph Exploration

This is pretty self-explanatory. We shall aim to write an algorithm which can "explore" our graph and see how different vertices are connected to each other. It is intuitively the first step do be done as any other complex algorithm tat we'd write would surely use this as it's subpart.

Before we write an algorithm for this, let us formally define what a "path" is in an Algorithm.

Definition. A Path is a collection of n vertices like $\{v_1, v_2, ..v_n\}$ such that all (v_i, v_{i+1}) are connected by an edge.

We shall first present the Pseudo-code for an algorithm which, when presented with a node, returns a list containing all the reachable nodes from the input node.

```
1    #discover is the list storing the nodes
2    #node has bool visit, which is made True once
     visited
3    #neighbours of a node stored in list adjacent
4
5    def reach(node s):
6        s.visit ← True
7        discover.append(s)
8        for temp in s.adjacent:
9            if temp.visit is False:
10               reach(temp)
11       return discover
```

Now that *reach*() has been properly defined, let us write the pseudo-code for a Depth-First Search Algorithm. This need not do much as of now, as it can be modified later to suit our needs. As of now though, this algorithm should be able to traverse and return all the vertices present in the given graph BY TRAVERSING THE GRAPH.

```
    #nodes is the list which stores all nodes
    def DFS():
        create empty list called nodes
        for node in tree:
            if node.visit is False:
                nodes.append(reach(node))
```

Understand that the algorithm's run-time would be of the order $O(|V|+|E|)$. It's simple enough to be seen directly without any outright explanation. This is **Linear Time** in case of Graphs.

4.3.1 Connected Components

Connected Components are like "islands" when we look at a pictorial representation of graphs.

Definition. A set of vertices (S) is said to be a Connected Component if for every $u, v \in S$, There exists a path connecting u and v.

Notice that this relation is Equivalence in nature. We can modify our $DFS()$ algorithm to easily count the number of Connected Components in the given graph, and also find out which nodes belong to the same connected component. We first add a new variable called `cc` in the node to store the label of the connected component. The Pseudo-Code is:

```

    def DFS():
        #create label here
        num = 1
        create empty list called nodes
        for node in tree:
            if node.visit is False:
                nodes.append(reach(node, num))
                #increase label
                num ← num + 1


```

```
13
14      def reach(node s, int num):
15          s.visit ← True
16          s.cc ← num #storing the label here
17          discover.append(s)
18          for temp in s.adjacent:
19              if temp.visit is False:
20                  reach(temp)
21          return discover
```

Now, once the labels have been set, we could simply scan through the vertices and pick out the ones we need. This algorithm also takes $O(|V| + |E|)$ time as nothing has changed significantly, and only constant time operations have been added.

4.4 Directed Graphs

The graphs that we have discussed so for have no directional component for their edges. This means that the edge can be traversed from either end to the other. However, if an edge only allowed for travel in one direction, then the graph would be called a "Directed Graph".

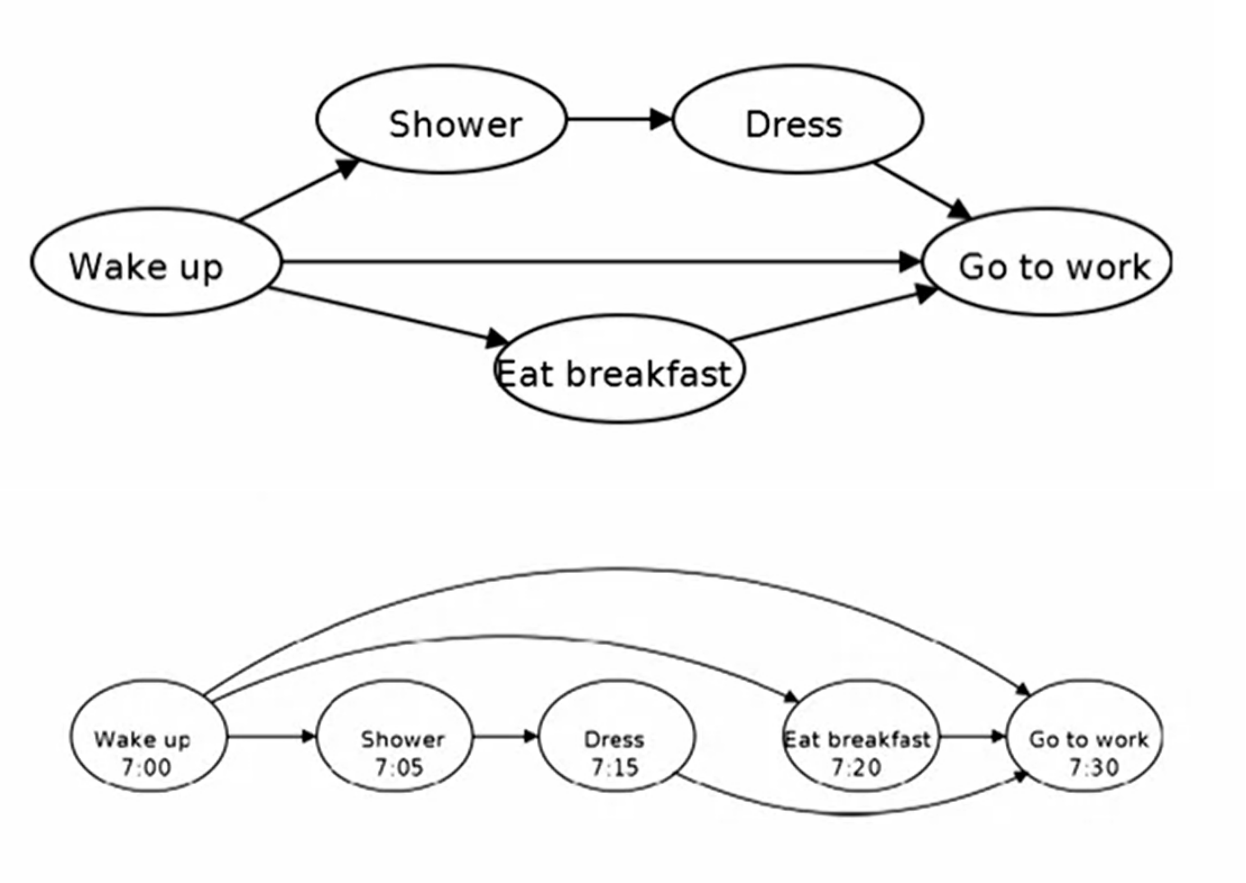
If we think about it, Directed graphs are more useful in applications of road mapping and the such because not all roads would allow passenger to move in both the directions freely. Thus, defining these graphs is important and necessary.

We can simply modify the definition of Adjacent list to account for this. The vertices which are present in the adjacent list of a vertex v all have a directed edge from v to the corresponding vertex.

1. **SINKS:** The vertices which only have edges pointing towards them.
2. **SOURCES:** The vertices which only have edges pointing away from them.

4.4.1 Linear Ordering

A Linear arrangement of the vertices in the given graph which satisfies the directional orders in the graphs is called as a Linear Ordering. The following image makes it more clear:



It can be clearly seen that Linear Ordering of a Directed Graph is possible **IFF** there are no cycles present in the graph. A directed Graph which has no cycles present in it is called as **Directed Acyclic Graph**.

Algorithm for Linear Ordering a graph

The logic behind this is pretty simple. We first find the a sink in the graph. (A DAG must have atleast one sink) We then append this graph to the last of the LO. We then remove this vertex from the graph and iterate till there are no more vertices left in the Graph.

Strongly Connected Components

Simply put, this is similar to the Connected-Components that we described earlier for undirected graphs, but for Directed Graphs.

4.5 Dijkstra's Algorithm

As we can realise, graphs can be used pretty easily to represent a map of an area with the edges representing nodes. The next logical step would be to develop an algorithm for finding the shortest path between two vertices. The problem statement has been stated properly below.

Problem Statement. *Given a Graph and a starting vertex S. Each edge has a corresponding "weight", i.e., the time taken to traverse the edge. The weight is Non-Zero. Find the minimum time it would take to reach ANY vertex from the vertex S.*

Dijkstra's Algorithm is a simple way of achieving this. We have a variable in each of the vertex which stores the minimum distance from S, let it be called *dist*.

1. We first instantiate *dist* in S to be 0, and then all the others are instantiated to ∞.
2. The vertices which have their *dist* figured out are stored in a set called R. Calculate the possible values of *dist* for it's neighbours(which are not in R) by using *dist* of the current vertex.
3. If the calculated value is smaller than what is stored, we replace the stored with the calculated value.
4. Choose the vertex which is not in R, which has the lowest value of *dist* to be the current vertex and iterate from step 2.
5. When all belong to R, all the distances are known.

We can go further and add another variable called *prev* which points to the node from where we arrived at the current node. This can be useful if we are trying to obtain the path in which our algorithm travels. The pseudo-code would be:

```
# G is the Graph, S is the Source
def Dijkstra(G, S):
    S.dist ← 0
    v ← S
    #infinity refers to very large number
    rest all Vertices set dist ← infinity
    create PriorityQueue(Q) and store all vertices
    while Q is not empty:
        for all w neighbours of v:
            if v.dist + E(v,w) < w.dist:
                w.dist ← v.dist + E(v,w)
 #ExtractMax() returns and removes least priority
        v ← ExtractMax(Q)
```

From this we can calculate the minimum distance from the mentioned source to any vertex in the entire graph. This algorithm forms the basis of another efficient algorithm called the $A^* - Algorithm$ which can be used to easily find the shortest distance between any two given vertices.

4.6 $A^* - Algorithm$

As discussed earlier, this algorithm is just a small variation of the Dijkstra's Algorithm that we've discussed earlier. This algorithm involves a **Directed Search**, meaning that the algorithm tends to prioritize the choices which take it closer to the target amongst all the choices.

Definition. Potential Function is a function which maps all the vertices in the graph to a numerical value. It is denoted by $\pi(v)$.

Now that a Potential function has been defined, then next logical step is to use this potential function to re-define the edge weights of our graph again. Keep in mind that we are talking about a directed graph here.

The redefined edge weight for an edge from a vertex u to another vertex v is given by:

$$l_\pi(u, v) = l(u, v) - \pi(u) + \pi(v)$$

This poses another constraint; by definition, all the new edge weights must also be non-negative. The Potential Functions which satisfy this condition are said to be **Feasible**.

Let s be the Starting Vertex and t be the target vertex. There exist many paths, and the i^{th} path is denoted by P_i. Notice that after redefining all the edges, the Path Length shall be:

$$l_\pi(P_i) = l(P_i) - \pi(s) + \pi(t)$$

That is, the path lengths only change by a constant. We can therefore also conclude from this that the **Shortest path is the same after conversion**, because all the paths differ from the original by a constant. Sure, the length has changed, but the path is still the same.

Therefore, we can apply Dijkstra's Algorithm to this modified graph and still end up with the shortest path for the original graph. If we were required to calculate the minimum length between the two vertices, we can get that from the new graph as well, as it differs by a constant.

5 String Algorithms

Algorithms on Strings have an immense importance in our daily lives. Search Engines use these algorithms to provide us with the results which are related to our search, after scanning through Terabytes of Webpages in an instant.

These algorithms can also be used in Bio-informatics, to check how similar the DNA of two species is, or where the DNA of a person with a genetic disease and a healthy person differ. This shall involve scanning through a large number of strings, and the methods to do o shall be discussed below.

The Human Genome project, consisted of DNA from humans to be extracted and broken up randomly by enzymes. The fragments are then analyzed to get the sequence of the Amino Acids making up that part. Then an algorithm is tasked with scanning all these pieces and figuring out where each piece should be located, much like a jigsaw puzzle, in which the pieces can be overlapping, are randomly generated, and the final solution is Millions of Amino Acids long. This is where these algorithms come into play.

We shall look at some ways of checking whether given strings are sub-strings of a larger string, in an efficient manner.

5.1 Pattern Matching Algorithms

Problem Statement. *Given a main string S, and n other strings p_n, check which of the p_n are substrings of S, and where they are located in the main string as well.*

The easiest solution would be to simply check for the first letter of the main with the first letter of p_1. This is the simplest solution, and it DOES work, but its VERY slow. The pseudo-code for the naive algorithm would be as follows:

```
#P[n] is the array with all the strings to be checked
#S[0] refers to first letter of string
#result is the array with the substrings
def Naive_Matching(S, P[n]):
    result ← []
    for i in range(n):
        j ← 0
        while True:
            if S[j] != P[i][j]:
                break
            if j = len(P[i]):
                result.append(P[i])
            j ← j + 1
    return result
```

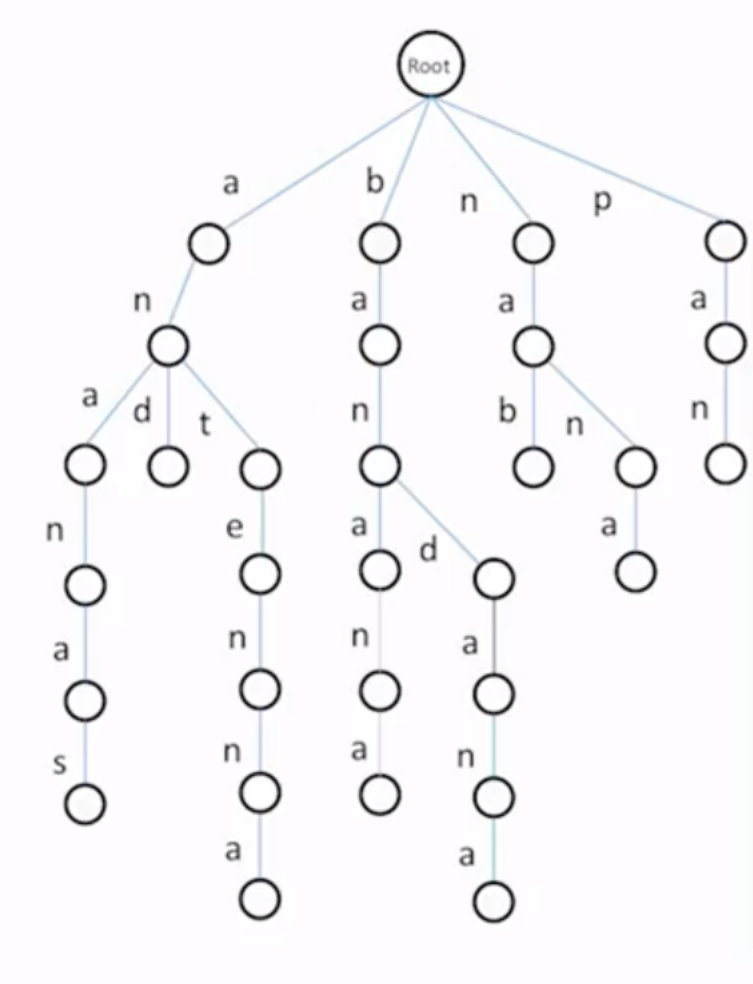
Let's look at how this algorithm performs time-wise. It can be clearly seen that the time would be of the order $O(len(S) \cdot n)$, and in the case of the human genome, this would never get done for obvious reasons.

One reason why this algorithm is so slow is that we are needed to scan through the same string multiple times. We can modify the approach to be so that we are required to scan the string only once. The following method talks about this approach.

5.1.1 Trie-Matching the Smaller Strings

A "Trie" is a way of grouping together all the smaller strings efficiently and simply. A Trie is basically a Tree which stores the strings with the common alphabets acting as nodes. For example, the below structure is how the following sub-strings are stored:

ananas, and, antenna, banana, bandana, nab, nana, pan



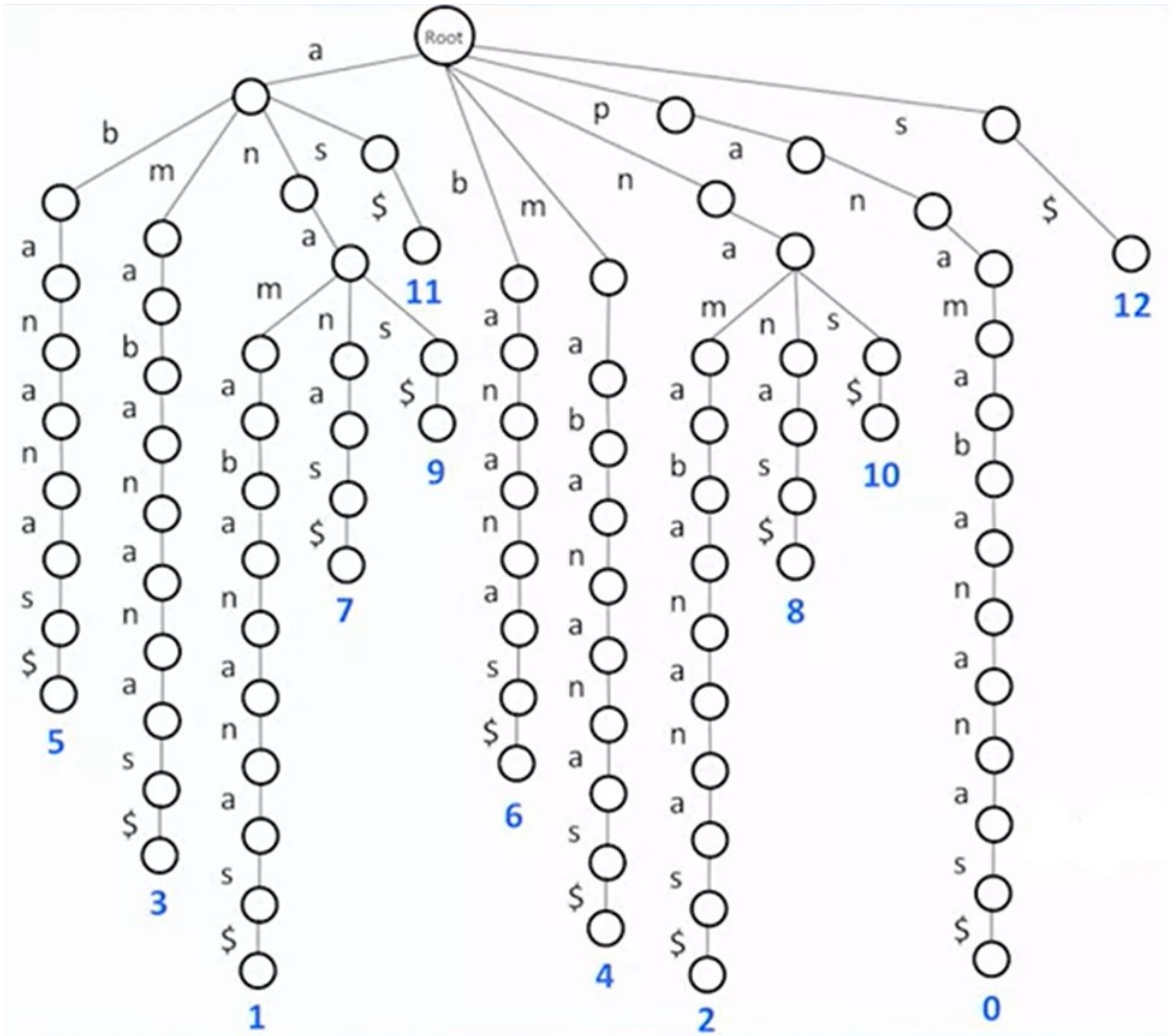
It can be seen clearly that this method would save a lot of time as we would be checking all the sub-strings at once, and this would obviously more faster. But would this be enough? Actually no, because we would still have the run-time would be $O(len(S) \cdot max(P[i]))$.

The time for this still be very high, and thus we are required to think of more efficient methods.

5.1.2 More Efficient Trie-Matching

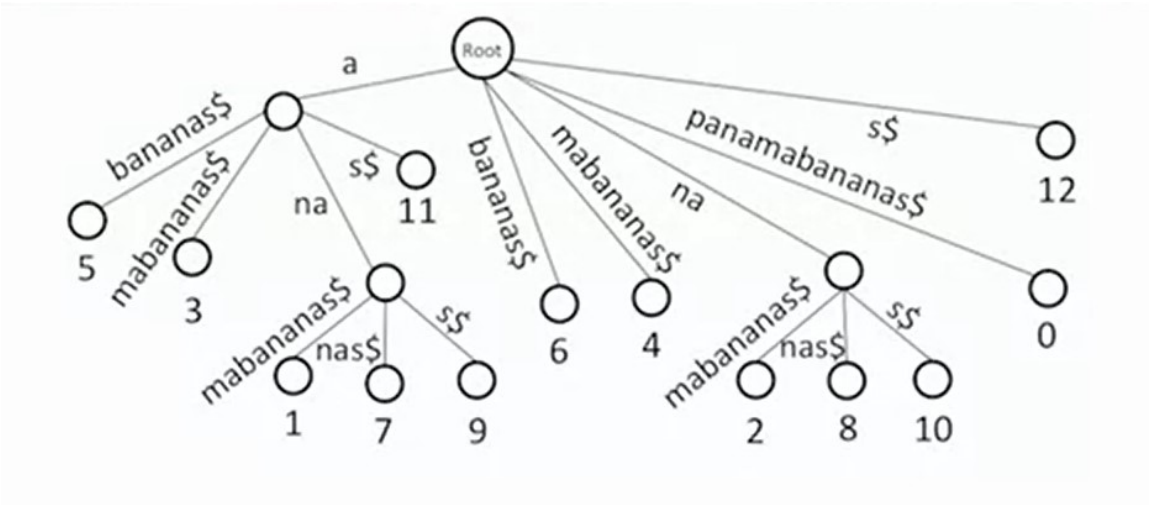
Instead of putting all the strings to be searched in a Trie, lets instead put the ORIGINAL STRING, i.e., S, in a Trie. In the above example, suppose $S =$ "PANAMABANANAS".

We then put all the possible sub-strings of S into a Trie, as shown below. The sub-strings of S would be "PANAMABANANAS", "ANAMABANANAS", and so on. The Leaves of the Trie are denoted by $ sign, and it also stores the number where the Substring is located. This allows us to pin-point where the particular substring is present in the original string.

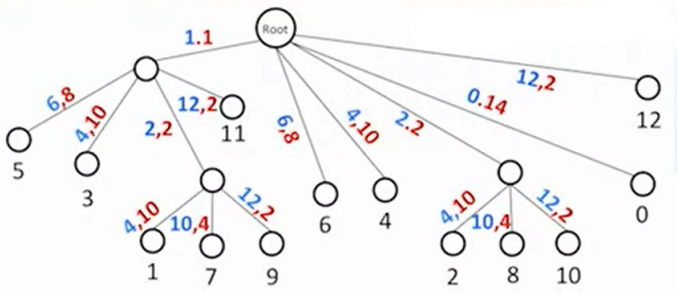


Although this would make matching VERY fast, the space needed to store the sub-string is HUGE. By some quick math, we can see that the space needed to store a string of size n would be $O(n^2)$; which is completely impractical.

We overcome this problem pretty simply. First, storing each letter of the word "BANANAS" in the left-most branch in a different node is a huge waste of space. We can simply store the entire word "BANANAS$" in one leaf itself. Doing this to the above tree gives us this:



This STILL doesn't solve our problem, as we still need to store the entire sub-strings. However, we can work around this, pretty easily. Because a string can be accessed in constant-time, we can simply store the starting index and the length of the string. This DRASTICALLY reduces the space needed to store the string. The modified Trie is as follows:



6 References

- Coursera's Specialization Course
- Coursera's Data Structures and Algorithms Course
- Coursera's Courses on Graphs and Strings
- Tutorials by TutorialsPoint