MODULE-III:
Asymptotic Notations,
Dynamic Programming (LCS,
Floyd-Warshall Algorithm,
Matrix Chain Multiplication),

Greedy Algorithm (Single Source Shortest Path,
Knapsack problem,
Minimum Cost Spanning Trees),

Geometric Algorithm (Convex hulls,
Segment Intersections,
Closest Pair),

Internet Algorithm (Tries,
Ukonnen's Algorithm,
Text pattern matching),

Numerical Algorithm (Integer,
Matrix and Polynomial multiplication,
Extended Euclid's algorithm)

MODULE-IV:
Polynomial Time,
Polynomial-Time Verification,
NP Completeness & reducibility,
NP Completeness proofs,
Cook's theorem

Asymptotic Notations

Why performance analysis?

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun! To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.

2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Why performance analysis?

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun! To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

we will take an example of Linear Search and analyze it using Asymptotic analysis. Let us consider the following implementation of Linear Search.

```
# Python 3 implementation of the approach
# Linearly search x in arr[]. If x is present
# then return the index, otherwise return -1
def search(arr, n, x):
        i = 0
        for i in range(i, n):
                if (arr[i] == x):
                        return i
        return -1

# Driver Code
arr = [1, 10, 30, 15]
x = 30
n = len(arr)
print(x, "is present at index", search(arr, n, x))
```

Output:
30 is present at index 2

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.

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. 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.

Asymptotic Notations and Apriori Analysis

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.
Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

Big O Notation:

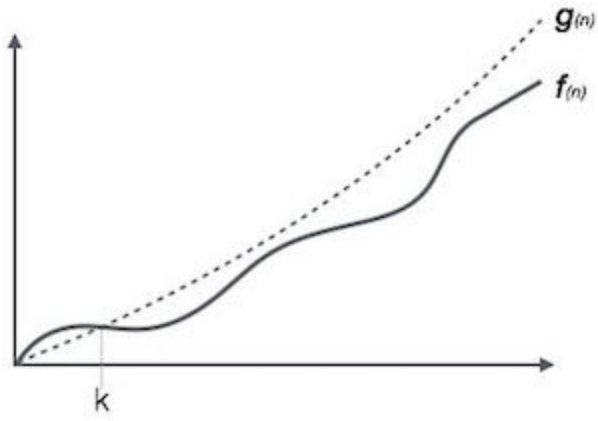
The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

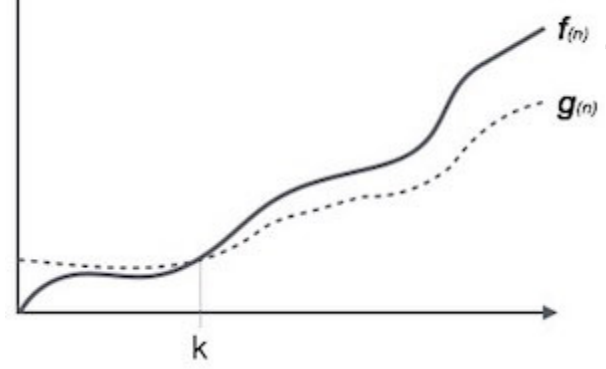
1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The **Big O** notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(f(n)) = { g(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ c*g(n) for all n >= n0}



Big Oh Notation, O



Omega Notation, Ω

Ω Notation

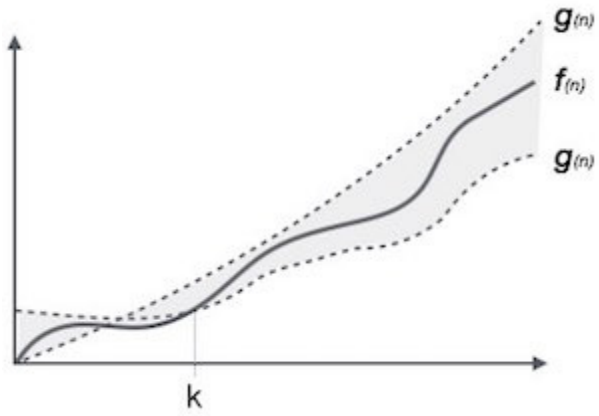
The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
For example, for a function f(n)

Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that 0 ≤ g(n) ≤ c.f(n) for all n > n0. }

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −

θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }



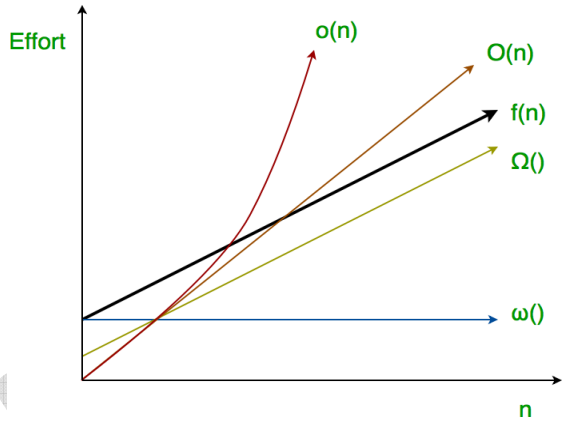
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$
Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than Θn^2) irrespective of the constants involved.

Little o

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. **"Little-o" (o())** notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).



Its means little o() means **loose upper-bound** of f(n).

In mathematical relation,
f(n) = o(g(n)) means
lim f(n)/g(n) = 0
n→∞

Examples:

Is $7n + 8 \in o(n^2)$?
In order for that to be true, for any c, we have to be able to find an n0 that makes
f(n) < c * g(n) asymptotically true.

lets took some example, If c = 100,we check the inequality is clearly true. If c = 1/100 , we'll have to use a little more imagination, but we'll be able to find an n0. (Try n0 = 1000.) From these examples, the conjecture appears to be correct. then check limits,
lim f(n)/g(n) = lim (7n + 8)/(n^2) = lim 7/2n = 0 (l'hospital)
n→∞ n→∞ n→∞
hence $7n + 8 \in o(n^2)$

Little ω

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n0.

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions.In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n).

we use ω notation to denote a lower bound that is not asymptotically tight. and, f(n) ∈ ω(g(n)) if and only if g(n) ∈ o((f(n)).

In mathematical relation, if f(n) ∈ ω(g(n)) then, **lim f(n)/g(n) = ∞ n→∞**
Example:
Prove that $4n + 6 \in \omega(1)$;
the little omega(o) running time can be proven by applying limit formula given below.
if lim f(n)/g(n) = ∞ then functions f(n) is ω(g(n))
n→∞
here,we have functions f(n)=4n+6 and g(n)=1 lim (4n+6)/(1) = ∞ n→∞
and,also for any c we can get n0 for this inequality 0 <= c*g(n) < f(n), 0 <= c*1 < 4n+6
Hence proved.

Analysis of iterative programs

O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

// set of non-recursive and non-loop statements

For example swap() function has O(1) time complexity.

A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions
}
```

O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

$O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
    }
}
for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
}
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.

O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

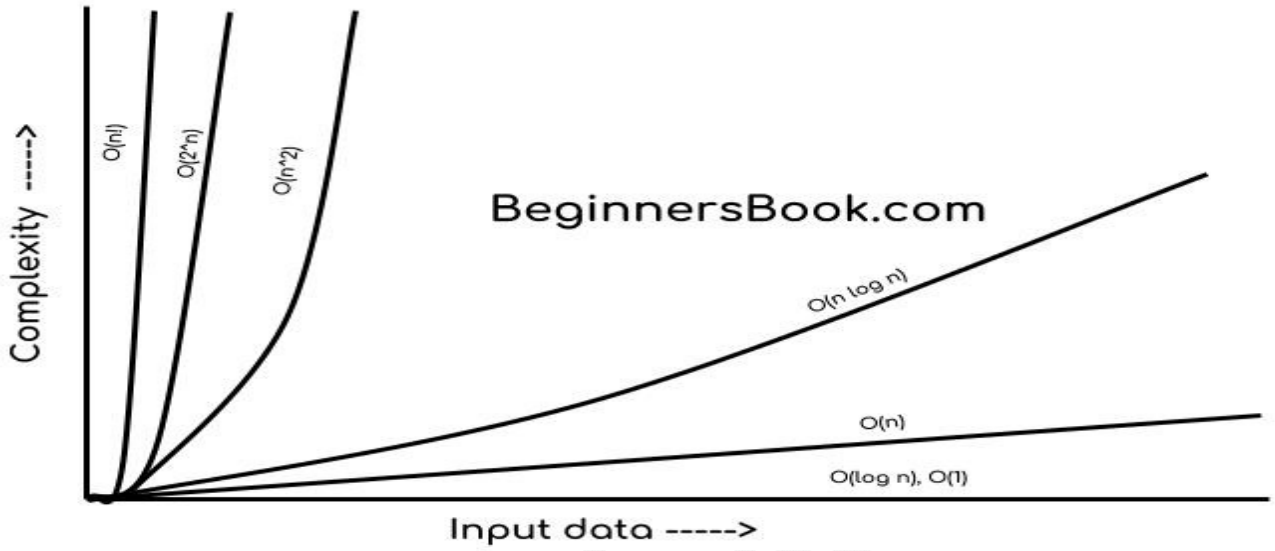
For example Binary Search(refer iterative implementation) has O(Logn) time complexity. Let us see mathematically how it is O(Log n). The series that we get in first loop is 1, c, c^2, c^3, … c^k. If we put k equals to $Log_c n$, we get $c^{Log}{}_c n$ which is n.

O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
```

```
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 1; i = fun(i)) {
   // some O(1) expressions
}
```

O(1) < O(log n) < O (n) < O(n log n) < O(n^2) < O (n^3)< O(2^n) < O(n!)



Apriori and Apostiari Analysis

Apriori analysis means, analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.

Apostiari analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.
In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.
In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same.

Amortized Analysis

Amortized analysis is generally used for certain algorithms where a sequence of similar operations are performed.

- Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.
- Amortized analysis differs from average-case analysis; probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

It is not just a tool for analysis, it's a way of thinking about the design, since designing and analysis are closely related.

Aggregate Method

The aggregate method gives a global view of a problem. In this method, if **n** operations takes worst-case time ***T(n)*** in total. Then the amortized cost of each operation is ***T(n)/n***. Though different operations may take different time, in this method varying cost is neglected.

Accounting Method

In this method, different charges are assigned to different operations according to their actual cost. If the amortized cost of an operation exceeds its actual cost, the difference is assigned to the object as credit. This credit helps to pay for later operations for which the amortized cost less than actual cost.

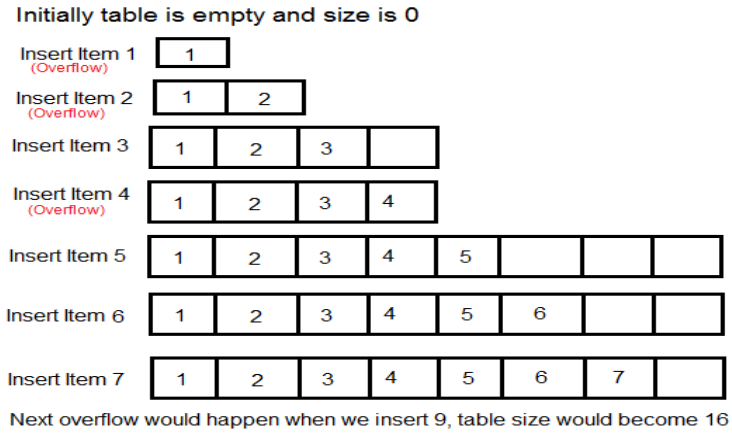
Potential Method

This method represents the prepaid work as potential energy, instead of considering prepaid work as credit. This energy can be released to pay for future operations.

If we perform ***n*** operations starting with an initial data structure D_0. Let us consider, c_i as the actual cost and D_i as data structure of i^{th} operation. The potential function Φ maps to a real number $\Phi(D_i)$, the associated potential of D_i.

Dynamic Table

If the allocated space for the table is not enough, we must copy the table into larger size table. Similarly, if large number of members are erased from the table, it is a good idea to reallocate the table with a smaller size. Using amortized analysis, we can show that the amortized cost of insertion and deletion is constant and unused space in a dynamic table never exceeds a constant fraction of the total space.



What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing.

Following are few important notes.
1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.
2) The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called ***Accounting Method*** and ***Potential Method***. We will be discussing the other two methods in separate posts.
3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing.

What is Space Complexity?

- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- We often speak of **extra memory** needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
- We can use bytes, but it's easier to use, say, the number of integers used, the number of fixed-sized structures, etc.
- In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.
- Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important issue as time complexity

Definition
Let **M** be a deterministic **Turing machine (TM)** that halts on all inputs. The space complexity of **M** is the function $f \colon N \rightarrow N$, where ***f(n)*** is the maximum number of cells of tape and **M** scans any input of length **M**. If the space complexity of **M** is ***f(n)***, we can say that **M** runs in space ***f(n)***.
We estimate the space complexity of Turing machine by using asymptotic notation.
Let $f \colon N \rightarrow R^+$ be a function. The space complexity classes can be defined as follows –
SPACE = {L | L is a language decided by an O(f(n)) space deterministic TM}
SPACE = {L | L is a language decided by an O(f(n)) space non-deterministic TM}
PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine.
In other words, $PSPACE = U_k SPACE (n^k)$

Dynamic Programming

Dynamic Programming is the most powerful design technique for solving optimization problems.
Divide & Conquer algorithm partition the problem into disjoint sub problems solve the sub problems recursively and then combine their solution to solve the original problems.

Dynamic Programming is used when the sub problems are not independent, e.g. when they share the same sub problems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming solves each sub problems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
Dynamic Programming is a Bottom-up approach- we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:
Dynamic Programming works when a problem has the following features:-

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping sub problems:** When a recursive algorithm would visit the same sub problems repeatedly, then a problem has overlapping sub problems.

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub problems, then we can improve on a recursive implementation by computing each sub problem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

If the space of sub problems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming
There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

i. Start with smallest subproblems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.
iii. Until solving at the solution of the original problem.

Components of Dynamic programming

1. **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Longest common subsequence (LCS)
3. Floyd-Warshall Algorithm
4. Matrix Chain Multiplication

Longest common subsequence (LCS)

What is Longest Common Subsequence:

A longest subsequence is a sequence that appears in the same relative order, but not necessarily contiguous(not substring) in both the string.

A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2\ y_n)$ and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Characteristics of Longest Common Sequence

A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given a sequence $X = (x_1\ x_2.....x_m)$ we define the ith prefix of X for i=0, 1, and 2...m as $X_i = (x_1\ x_2.....x_i)$. For example: if X = (A, B, C, B, C, A, B, C) then X_4= (A, B, C, B)

Optimal Substructure of an LCS:

Let $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2.....)\ y_n)$ be the sequences and let $Z = (z_1\ z_2.....z_k)$ be any LCS of X and Y.

- If $x_m = y_n$, then z_k=x_m=y_n and Z_{k-1} is an LCS of $X_{m-1}$and Y_{n-1}
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of $X_{m-1}$and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: Recursive Solution:

LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding an LCS of X and Y_{n-1}.Whenever of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of X_{m-1} and Y_{n-1}.

Let c [i,j] be the length of LCS of the sequence $X_i$and Y_j.If either i=0 and j =0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem given the recurrence formula

$$c\ [i, j] = \begin{cases} 0 & \text{if } i = 0 \quad \text{or} \quad j = 0 \\ c\ [i-1, j-1]1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Step3: Computing the length of an LCS: let two sequences $X = (x_1\ x_2.....x_m)$ and $Y = (y_1\ y_2..... y_n)$ as inputs. It stores the c [i,j] values in the table c [0.....m,0.....n].Table b [1.....m, 1.....n] is maintained which help us to construct an optimal solution. c [m, n] contains the length of an LCS of X,Y.

LCS Problem Statement*:* Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall from theory of permutation and combination that number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on. We know that ${}^nC_0 + {}^nC_1 + {}^nC_2 + \ldots {}^nC_n = 2^n$. So a string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

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

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

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:
Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).
If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])
If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))
Examples:
1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as: L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:
L(“ABCDGH”, “AEDFHR”) = MAX (L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”))
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

2) Overlapping Subproblems:
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

```
# A Naive recursive Python implementation of LCS problem
def lcs(X, Y, m, n):

    if m == 0 or n == 0:
    return 0;
    elif X[m-1] == Y[n-1]:
    return 1 + lcs(X, Y, m-1, n-1);
    else:
    return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X , Y, len(X), len(Y))
```

Output:
Length of LCS is 4

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

```
                         lcs("AXYT", "AYZX")
                        /
      lcs("AXY", "AYZX")                    lcs("AXYT", "AYZ")
          /                                          /
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

```
# Dynamic Programming implementation of LCS problem
def lcs(X , Y):
        # find the length of the strings
        m = len(X)
        n = len(Y)

        # declaring the array for storing the dp values
        L = [[None]*(n+1) for i in xrange(m+1)]

        """Following steps build L[m+1][n+1] in bottom up fashion
        Note: L[i][j] contains length of LCS of X[0..i-1]
        and Y[0..j-1]"""
        for i in range(m+1):
                for j in range(n+1):
                        if i == 0 or j == 0 :
                                L[i][j] = 0
                        elif X[i-1] == Y[j-1]:
                                L[i][j] = L[i-1][j-1]+1
                        else:
                                L[i][j] = max(L[i-1][j] , L[i][j-1])

        # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
        return L[m][n]
#end of function lcs

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X, Y)
```

Output:
Length of LCS is 4

Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation.

Following is detailed algorithm to print the LCS. It uses the same 2D table L[][].
1) Construct L[m+1][n+1] using the steps discussed in above.
2) The value L[m][n] contains length of LCS. Create a character array lcs[] of length equal to the length of lcs plus 1 (one extra to store \0).

2) Traverse the 2D array starting from L[m][n]. Do following for every cell L[i][j]
…..**a)** If characters (in X and Y) corresponding to L[i][j] are same (Or X[i-1] == Y[j-1]), then include this character as part of LCS.
…..**b)** Else compare values of L[i-1][j] and L[i][j-1] and go in direction of greater value.

The following table (taken from Wiki) shows steps (highlighted) followed by the above algorithm.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	**0**	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	**1**	1	1	1	1	1	1
3	J	0	1	1	**2**	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	**3**	3	3	3
6	U	0	1	1	2	3	3	3	**4**
7	Z	0	1	2	2	3	3	3	4

```
# Dynamic programming implementation of LCS problem
# Returns length of LCS for X[0..m-1], Y[0..n-1]
def lcs(X, Y, m, n):
        L = [[0 for x in xrange(n+1)] for x in xrange(m+1)]

        # Following steps build L[m+1][n+1] in bottom up fashion. Note
        # that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
        for i in xrange(m+1):
                for j in xrange(n+1):
                        if i == 0 or j == 0:
                                L[i][j] = 0
                        elif X[i-1] == Y[j-1]:
                                L[i][j] = L[i-1][j-1] + 1
                        else:
                                L[i][j] = max(L[i-1][j], L[i][j-1])

        # Following code is used to print LCS
        index = L[m][n]

        # Create a character array to store the lcs string
        lcs = [""] * (index+1)
        lcs[index] = ""

        # Start from the right-most-bottom-most corner and
        # one by one store characters in lcs[]
```

```
        i = m
        j = n
        while i > 0 and j > 0:

                # If current character in X[] and Y are same, then
                # current character is part of LCS
                if X[i-1] == Y[j-1]:
                        lcs[index-1] = X[i-1]
                        i-=1
                        j-=1
                        index-=1

                # If not same, then find the larger of two and
                # go in the direction of larger value
                elif L[i-1][j] > L[i][j-1]:
                        i-=1
                else:
                        j-=1

        print "LCS of " + X + " and " + Y + " is " + "".join(lcs)

# Driver program
X = "AGGTAB"
Y = "GXTXAYB"
m = len(X)
n = len(Y)
lcs(X, Y, m, n)
```

Floyd-Warshall Algorithm

- Floyd-Warshall Algorithm is an algorithm for solving All Pairs Shortest path problem which gives the shortest path between every pair of vertices of the given graph.
- Floyd-Warshall Algorithm is an example of dynamic programming.
- The main advantage of Floyd-Warshall Algorithm is that it is extremely simple and easy to implement.

Algorithm-

```
Create a |V| x |V| matrix
For each cell (i,j) in M do-
if i = = j
        M[ i ][ j ] = 0            // For all diagonal elements, value = 0
if (i , j) is an edge in E
        M[ i ][ j ] = weight(i,j)   // If there exists a direct edge between the vertices, value = weight of edge
else
        M[ i ][ j ] = infinity      // If there is no direct edge between the vertices, value = ∞
for k from 1 to |V|
        for i from 1 to |V|
                for j from 1 to |V|
                        if M[ i ][ j ] > M[ i ][ k ] + M[ k ][ j ]
                                M[ i ][ j ] = M[ i ][ k ] + M[ k ][ j ]
```

Time Complexity-

- Floyd Warshall Algorithm consists of three loops over all nodes.
- The inner most loop consists of only operations of a constant complexity.

- Hence, the asymptotic complexity of Floyd-Warshall algorithm is $O(n^3)$, where n is the number of nodes in the given graph.

When Floyd- Warshall Algorithm is used?

- Floyd-Warshall Algorithm is best suited for dense graphs since its complexity depends only on the number of vertices in the graph.
- For sparse graphs, Johnson's Algorithm is more suitable.

Example: Input:

```
graph[][] = { {0,   5,  INF, 10},
              {INF,  0,  3,  INF},
              {INF, INF, 0,   1},
              {INF, INF, INF, 0} }
```

which represents the following graph

```
          10
     (0)------->(3)
      |         /|\
   5  |          |
      |          | 1
      \|/        |
     (1)------->(2)
          3
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

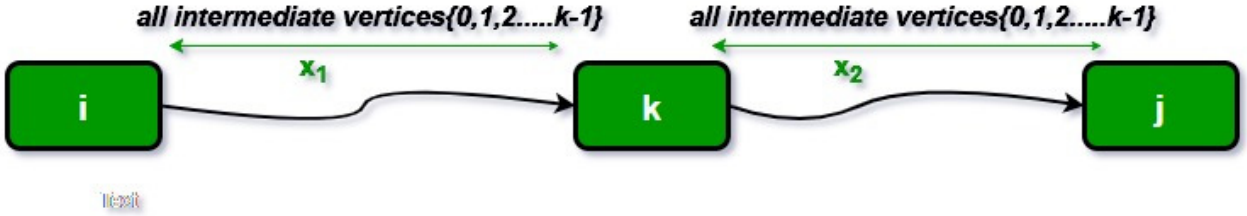
Output:
Shortest distance matrix

```
   0     5     8     9
 INF     0     3     4
 INF   INF     0     1
 INF   INF   INF     0
```

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.




```
# Python Program for Floyd Warshall Algorithm
# Number of vertices in the graph
V = 4
```

```
# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

        """ dist[][] will be the output matrix that will finally
                have the shortest distances between every pair of vertices """
        """ initializing the solution matrix same as input graph matrix
        OR we can say that the initial values of shortest distances
        are based on shortest paths considering no
        intermediate vertices """
        dist = map(lambda i : map(lambda j : j , i) , graph)

        """ Add all vertices one by one to the set of intermediate
        vertices.
        ---> Before start of an iteration, we have shortest distances
        between all pairs of vertices such that the shortest
        distances consider only the vertices in the set
        {0, 1, 2, .. k-1} as intermediate vertices.
        ----> After the end of a iteration, vertex no. k is
        added to the set of intermediate vertices and the
        set becomes {0, 1, 2, .. k}
        """
        for k in range(V):

                # pick all vertices as source one by one
                for i in range(V):

                        # Pick all vertices as destination for the
                        # above picked source
                        for j in range(V):

                                # If vertex k is on the shortest path from
                                # i to j, then update the value of dist[i][j]
                                dist[i][j] = min(dist[i][j] ,
                                                        dist[i][k]+ dist[k][j]
                                                        )
        printSolution(dist)


# A utility function to print the solution
def printSolution(dist):
        print "Following matrix shows the shortest distances\
between every pair of vertices"
        for i in range(V):
                for j in range(V):
                        if(dist[i][j] == INF):
                                print "%7s" %("INF"),
                        else:
                                print "%7d\t" %(dist[i][j]),
                        if j == V-1:
                                print ""
```

```
# Driver program to test the above program
# Let us create the following weighted graph
"""
                        10
                  (0)------->(3)
                   |         /|\
                 5 |          |
                   |          | 1
                  \|/         |
                  (1)------->(2)
                        3               """
graph = [[0,5,INF,10],
                        [INF,0,3,INF],
                        [INF, INF, 0, 1],
                        [INF, INF, INF, 0]
                    ]
# Print the solution
floydWarshall(graph)
```

Output:

Following matrix shows the shortest distances between every pair of vertices

```
  0     5     8     9
INF     0     3     4
INF   INF     0     1
INF   INF   INF     0
```

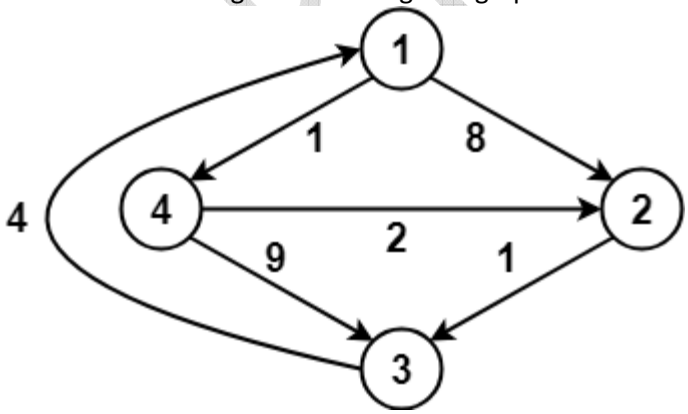
Time Complexity: O(V^3)

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

Problem-

Consider the following directed weighted graph-



Using Floyd-Warshall Algorithm, find the shortest path distance between every pair of vertices.

Solution-

Step-01:

- Remove all the self loops and parallel edges (keeping the edge with lowest weight) from the graph if any.
- In our case, we don't have any self edge and parallel edge.

Step-02:

Now, write the initial distance matrix representing the distance between every pair of vertices as mentioned in the given graph in the form of weights.

- For diagonal elements (representing self-loops), value = 0
- For vertices having a direct edge between them, value = weight of that edge
- For vertices having no direct edges between them, value = ∞

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

Step-03:

From step-03, we will start our actual solution.

NOTE

- Since, we have total 4 vertices in our given graph, so we will have total 4 matrices of order 4 x 4 in our solution. (excluding initial distance matrix)
- Diagonal elements of each matrix will always be 0.

The four matrices are-

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

The last matrix D_4 represents the shortest path distance between every pair of vertices.

Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next.
Here, Chain means one matrix's column is equal to the second matrix's row [always].
In general:
If A = $\lfloor a_{ij} \rfloor$ is a p x q matrix
B = $\lfloor b_{ij} \rfloor$ is a q x r matrix
C = $\lfloor c_{ij} \rfloor$ is a p x r matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\ b_{kj}$$

Given following matrices $\{A_1, A_2, A_3, \ldots A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications
$A_1 x A_2 x, A_3 x \ldots\ldots x A_n$

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for $1 \leq i \leq p$ and $1 \leq j \leq r$

$$C[i, j] = \sum_{k=1}^{q} A[i, k] B[k, j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension p x r Also each entry takes O (q) times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension p q r.
It is also noticed that we can save the number of operations by reordering the parenthesis.

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have: **(ABC)D = (AB)(CD) = A(BCD) = ...**

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way **(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30**

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way **((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30**

Input: p[] = {10, 20, 30}
Output: 6000
There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is 10*20*30

Optimal Substructure:
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

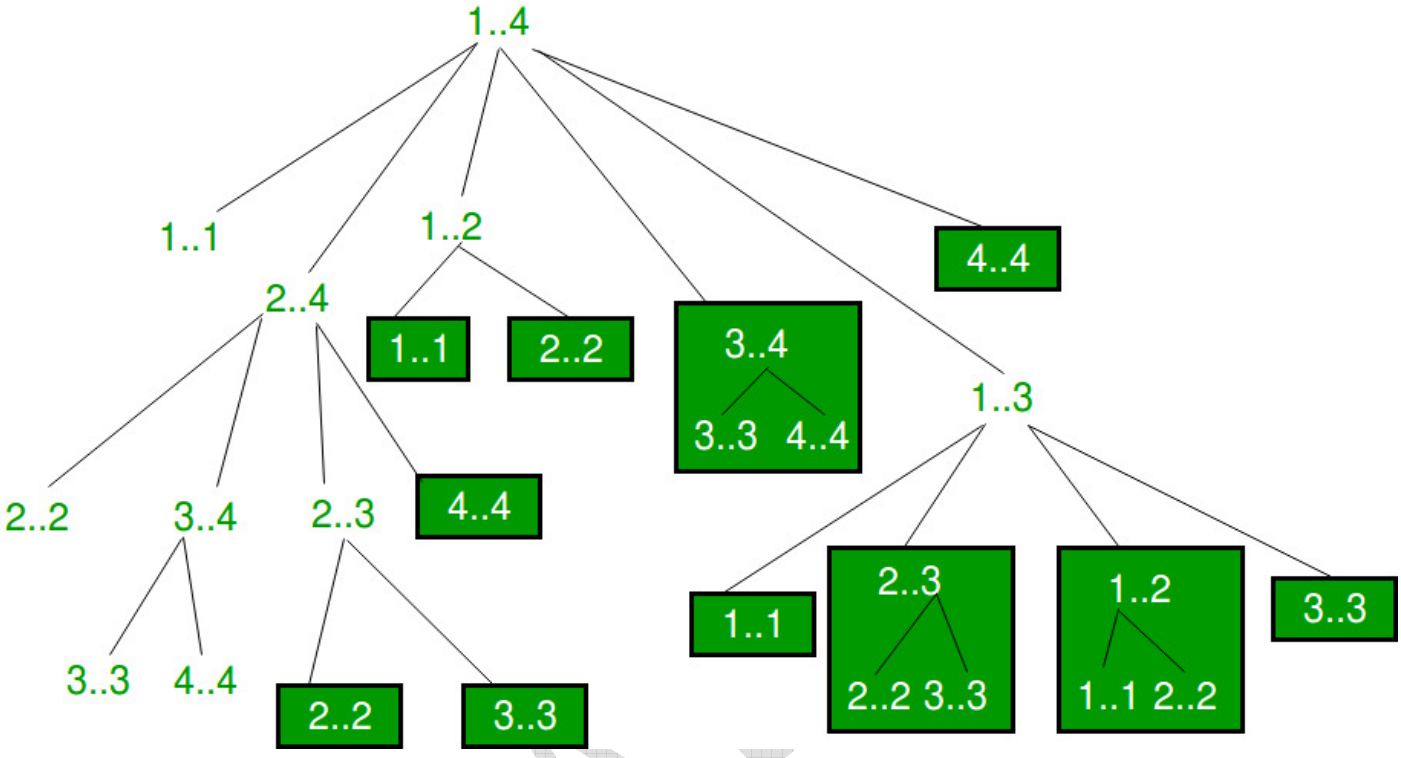
Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

2) Overlapping Subproblems
Following is a recursive implementation that simply follows the **above optimal substructure property.**

```
# A naive recursive implementation that
# simply follows the above optimal
# substructure property
import sys

# Matrix A[i] has dimension p[i-1] x p[i]
# for i = 1..n
def MatrixChainOrder(p, i, j):

        if i == j:
                return 0
        _min = sys.maxsize

        # place parenthesis at different places
        # between first and last matrix,
        # recursively calculate count of
        # multiplications for each parenthesis
        # placement and return the minimum count
        for k in range(i, j):
                count = (MatrixChainOrder(p, i, k)
                        + MatrixChainOrder(p, k+1, j)
                                + p[i-1] * p[k] * p[j])

                if count < _min:
                        _min = count;

        # Return minimum count
        return _min;

# Driver program to test above function
arr = [1, 2, 3, 4, 3];
n = len(arr);
print("Minimum number of multiplications is ", MatrixChainOrder(arr, 1, n-1));
```

Output:
Minimum number of multiplications is 30

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So Matrix Chain Multiplication problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.
Dynamic Programming Solution

Following is the implementation for Matrix Chain Multiplication problem using Dynamic Programming.

```
# Dynamic Programming Python implementation of Matrix
# Chain Multiplication. See the Cormen book for details
# of the following algorithm
import sys

# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
        # For simplicity of the program, one extra row and one
        # extra column are allocated in m[][]. 0th row and 0th
        # column of m[][] are not used
        m = [[0 for x in range(n)] for x in range(n)]

        # m[i,j] = Minimum number of scalar multiplications needed
        # to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
        # dimension of A[i] is p[i-1] x p[i]

        # cost is zero when multiplying one matrix.
```

```
        for i in range(1, n):
                m[i][i] = 0

        # L is chain length.
        for L in range(2, n):
                for i in range(1, n-L+1):
                        j = i+L-1
                        m[i][j] = sys.maxint
                        for k in range(i, j):

                                # q = cost/scalar multiplications
                                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                                if q < m[i][j]:
                                        m[i][j] = q
        return m[1][n-1]

# Driver program to test above function
arr = [1, 2, 3 ,4]
size = len(arr)

print("Minimum number of multiplications is " +str(MatrixChainOrder(arr, size)))
```

Output:
Minimum number of multiplications is 18
Time Complexity: O(n^3)
Auxiliary Space: O(n^2)

Greedy Algorithm

"Greedy Method finds out of many options, but you have to choose the best option."
In this method, we have to find out the best method/option out of many present ways.
In this approach/method we focus on the first stage and decide the output, don't think about the future.

This method may or may not give the best output. Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
2. **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

Components of Greedy Algorithm
Greedy algorithms have the following five components −

- A candidate set − A solution is created from this set.
- A selection function − Used to choose the best candidate to be added to the solution.
- A feasibility function − Used to determine whether a candidate can be used to contribute to the solution.
- An objective function − Used to assign a value to a solution or a partial solution.
- A solution function − Used to indicate whether a complete solution has been reached.

Areas of Application
Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Where Greedy Approach Fails
In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

Single Source Shortest Path

Introduction:
In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v_0,v_1,..... v_k) is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

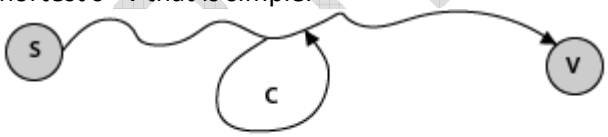
We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.
The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.
In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.
Variants:

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
- **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path: Existence:
If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



Cost of C < 0

Negative Weight Edges
It is a weighted graph in which the total weight of an edge is negative. If a graph has a negative edge, then it produces a chain. After executing the chain if the output is negative then it will give - ∞ weight and condition get discarded. If weight is less than negative and - ∞ then we can't have the shortest path in it.
Briefly, if the output is -ve, then both condition get discarded.

1. - ∞
2. Not less than 0.

And we cannot have the shortest Path.

Representing: Shortest Path

Given a graph G = (V, E), we maintain for each vertex v ∈ V a **predecessor** π [v] that is either another vertex or NIL. During the execution of shortest paths algorithms, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the **predecessor subgraph** $G_n = (V_n, E_n)$ induced by the value π. Here again, we define the vertex set V_π, to be the set of vertices of G with non - NIL predecessors, plus the source s:

$V_\pi = \{v \in V: \pi [v] \neq NIL\} \cup \{s\}$ }

The directed edge set E_Π is the set of edges induced by the Π values for vertices in V_Π:

$E_\Pi = \{(\Pi[v], v) \in E: v \in V_\Pi - \{s\}\}$

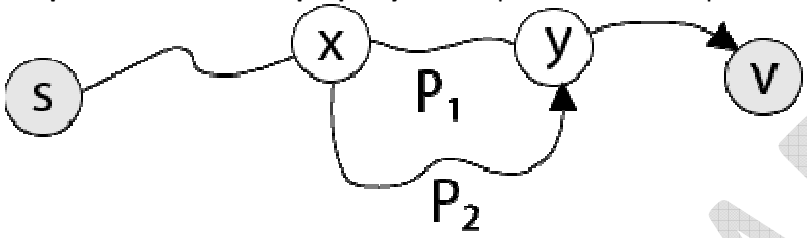
A **shortest - paths tree** rooted at s is a directed subgraph G = (V' E'), where V'∈ V andE'∈E, such that

1. V' is the set of vertices reachable from s in G
2. G' forms a rooted tree with root s, and
3. For all v ∈ V', the unique, simple path from s to v in G' is the shortest path from s to v in G.

Shortest paths are not naturally unique, and neither is shortest - paths trees.

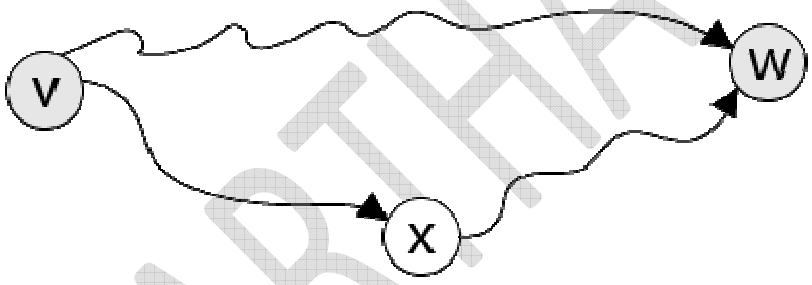
Properties of Shortest Path:

1. Optimal substructure property: All subpaths of shortest paths are shortest paths.



Let P_1 be x - y sub path of shortest s - v path. Let P_2 be any x - y path. Then cost of $P_1 \leq$ cost of P_2,otherwise P not shortest s - v path.

2. Triangle inequality: Let d (v, w) be the length of shortest path from v to w. Then,
d (v, w) ≤ d (v, x) + d (x, w)



3. Upper-bound property: We always have d[v] ≥ δ(s, v) for all vertices v ∈ V, and once d[v] conclude the value δ(s, v), it never changes.

4. No-path property: If there is no path from s to v, then we regularly have d[v] = δ(s, v) = ∞.

5. Convergence property: If s->u->v is a shortest path in G for some u, v ∈ V, and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times thereafter.

Relaxation

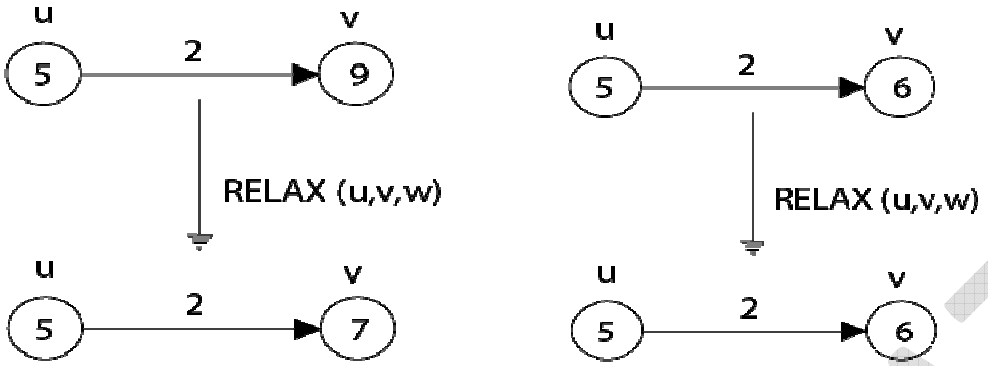
The single - source shortest paths are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight. For each vertex v ∈ V, we maintain an attribute d [v], which is an upper bound on the weight of the shortest path from source s to v. We call d [v] the **shortest path estimate**.

INITIALIZE - SINGLE - SOURCE (G, s)

1. for each vertex v ∈ V [G]
2. do d [v] ← ∞
3. π [v] ← NIL
4. d [s] ← 0

After initialization, π [v] = NIL for all v ∈ V, d [v] = 0 for v = s, and d [v] = ∞ for v ∈ V - {s}.
The development of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating d [v] and π [v]. A relaxation step may decrease the value of the shortest - path estimate d [v] and updated v's predecessor field π [v].

Fig: Relaxing an edge (u, v) with weight w (u, v) = 2. The shortest-path estimate of each vertex appears within the vertex.



(a) Because v. d > u. d + w (u, v) prior to relaxation, the value of v. d decreases
(b) Here, v. d < u. d + w (u, v) before relaxing the edge, and so the relaxation step leaves v. d unchanged.
The subsequent code performs a relaxation step on edge (u, v)
RELAX (u, v, w)
1. If d [v] > d [u] + w (u, v)
2. then d [v] ← d [u] + w (u, v)
3. π [v] ← u

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.
Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.
Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Dijkstra's Algorithm (G, w, s)
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

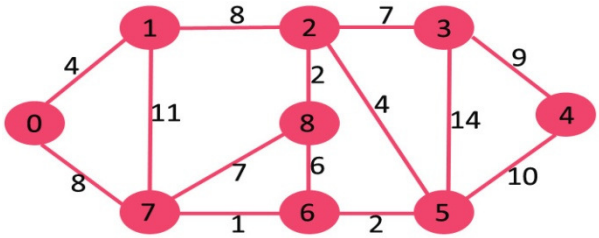
3) While *sptSet* doesn't include all vertices

….**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.

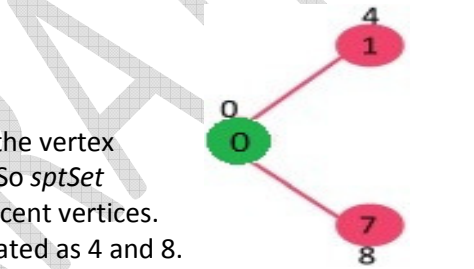
….**b)** Include u to *sptSet*.

….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

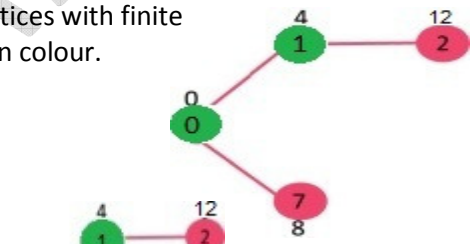
Let us understand with the following example:



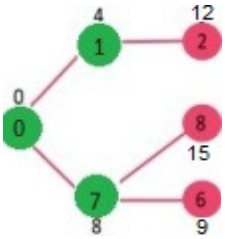
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



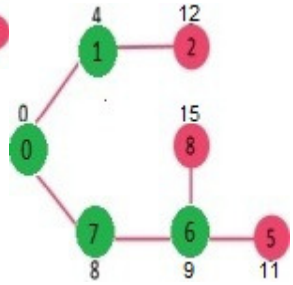
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



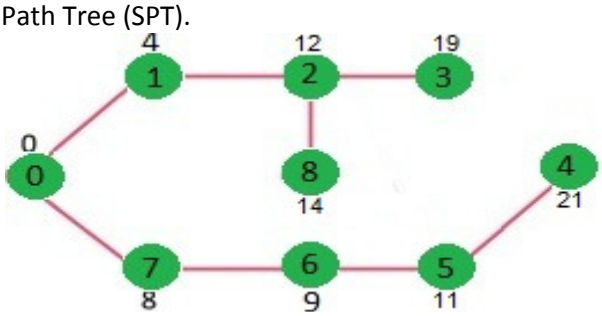
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph
# Library for INT_MAX
import sys

class Graph():
        def __init__(self, vertices):
                self.V = vertices
                self.graph = [[0 for column in range(vertices)]
                                        for row in range(vertices)]

        def printSolution(self, dist):
                print "Vertex \tDistance from Source"
                for node in range(self.V):
                        print node, "\t", dist[node]

        # A utility function to find the vertex with
        # minimum distance value, from the set of vertices
        # not yet included in shortest path tree
        def minDistance(self, dist, sptSet):
                # Initilaize minimum distance for next node
                min = sys.maxint

                # Search not nearest vertex not in the
                # shortest path tree
                for v in range(self.V):
                        if dist[v] < min and sptSet[v] == False:
                                min = dist[v]
                                min_index = v
                return min_index

        # Funtion that implements Dijkstra's single source
        # shortest path algorithm for a graph represented
        # using adjacency matrix representation
        def dijkstra(self, src):
                dist = [sys.maxint] * self.V
                dist[src] = 0
                sptSet = [False] * self.V
                for cout in range(self.V):
                        # Pick the minimum distance vertex from
                        # the set of vertices not yet processed.
                        # u is always equal to src in first iteration
                        u = self.minDistance(dist, sptSet)

                        # Put the minimum distance vertex in the
                        # shotest path tree
                        sptSet[u] = True

                        # Update dist value of the adjacent vertices
                        # of the picked vertex only if the current
                        # distance is greater than new distance and
                        # the vertex in not in the shotest path tree
```

```
                        for v in range(self.V):
                                if self.graph[u][v] > 0 and sptSet[v] == False and \
                                dist[v] > dist[u] + self.graph[u][v]:
                                        dist[v] = dist[u] + self.graph[u][v]
                self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[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]
                ];
g.dijkstra(0);
```

Output:

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

Notes:

1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it show the shortest path from source to different vertices.

2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.

3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).

4) Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap. Please see

Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Single Source Shortest Path in a directed Acyclic Graphs

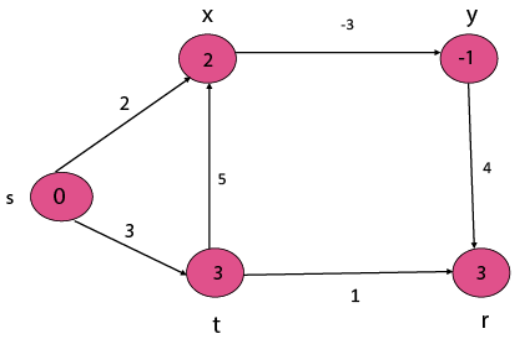
By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in Ø(V+E) time. Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

DAG - SHORTEST - PATHS (G, w, s)

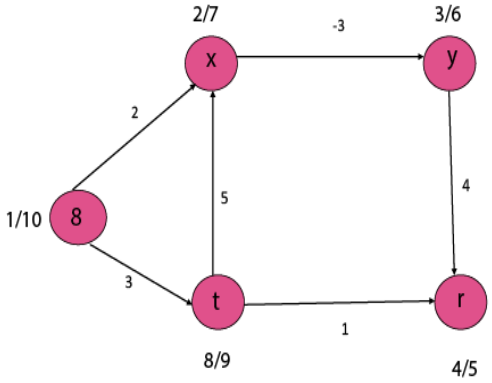
1. Topologically sort the vertices of G.
2. INITIALIZE - SINGLE- SOURCE (G, s)
3. for each vertex u taken in topologically sorted order
4. do for each vertex v ∈ Adj [u]
5. do RELAX (u, v, w)

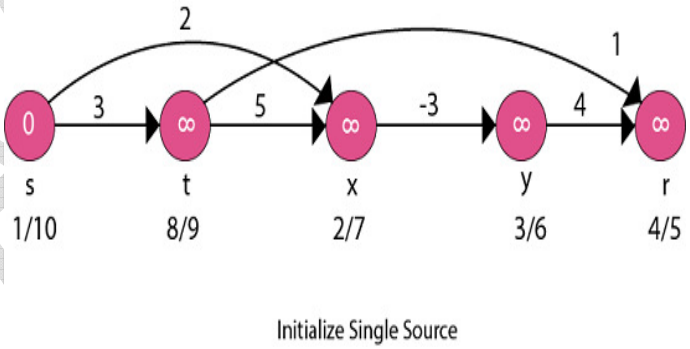
The running time of this data is determined by line 1 and by the for loop of lines 3 - 5. The topological sort can be implemented in Ø (V + E) time. In the for loop of lines 3 - 5, as in Dijkstra's algorithm, there is one repetition per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, we use only O (1) time per edge. The running time is thus Ø (V + E), which is linear in the size of an adjacency list depiction of the graph.

Example:



Step1: To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.

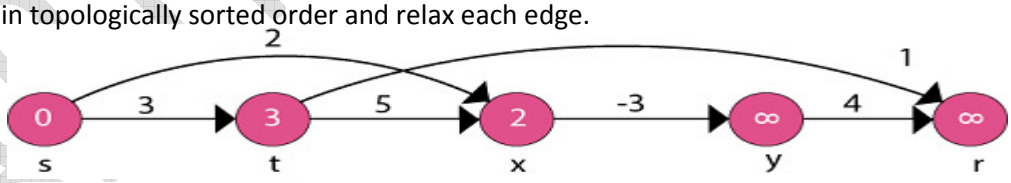




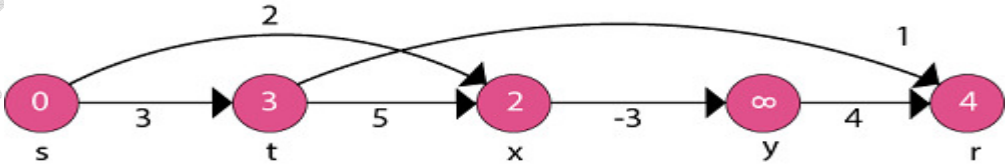
Initialize Single Source

Now, take each vertex in topologically sorted order and relax each edge.

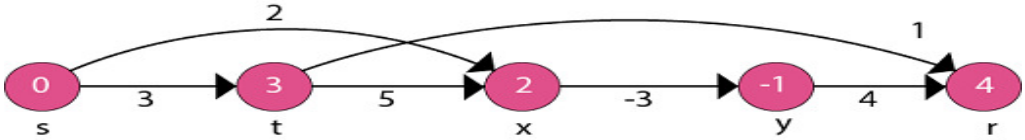
1. adj [s] →t, x
2. 0 + 3 < ∞
3. d [t] ← 3
4. 0 + 2 < ∞
5. d [x] ← 2



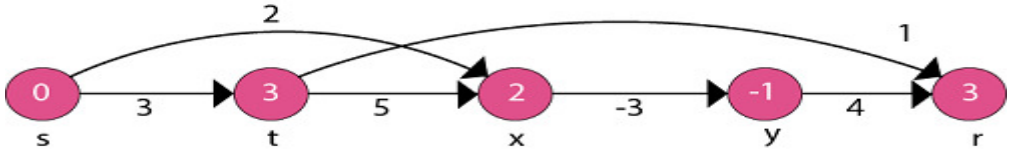
1. adj [t] → r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] → y
2. 2 - 3 < ∞
3. d [y] ← -1

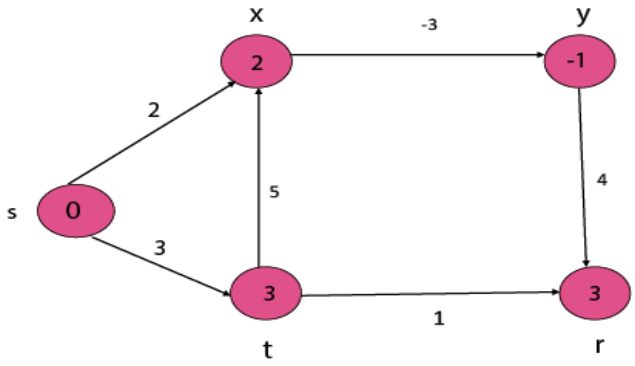


1. adj [y] → r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



Thus the Shortest Path is:

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3



Bellman-Ford Algorithm

Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.
This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.
It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

This algorithm detects the negative cycle in a graph and reports their existence.
Based on the "**Principle of Relaxation**" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.

Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

Recurrence Relation

$dist^{k}[u] = [min[dist^{k-1}[u], min[dist^{k-1}[i] + cost[i,u]]]]$ as i except u.

k → k is the source vertex
u → u is the destination vertex
i → no of edges to be scanned concerning a vertex.

BELLMAN -FORD (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)
2. for i ← 1 to |V[G]| - 1
3. do for each edge (u, v) ∈ E [G]
4. do RELAX (u, v, w)
5. for each edge (u, v) ∈ E [G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE.
8. return TRUE.

Algorithm: Bellman-Ford-Algorithm (G, w, s)

```
for each vertex v Є G.V
  v.d := ∞
  v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
  for each edge (u, v) Є G.E
    if v.d > u.d + w(u, v)
      v.d := u.d +w(u, v)
      v.∏ := u
for each edge (u, v) Є G.E
  if v.d > u.d + w(u, v)
    return FALSE
return TRUE
```

Analysis

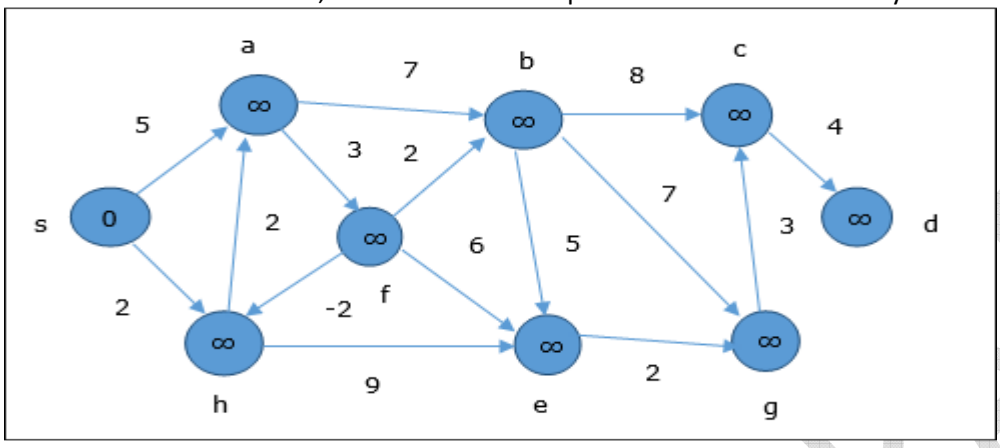
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

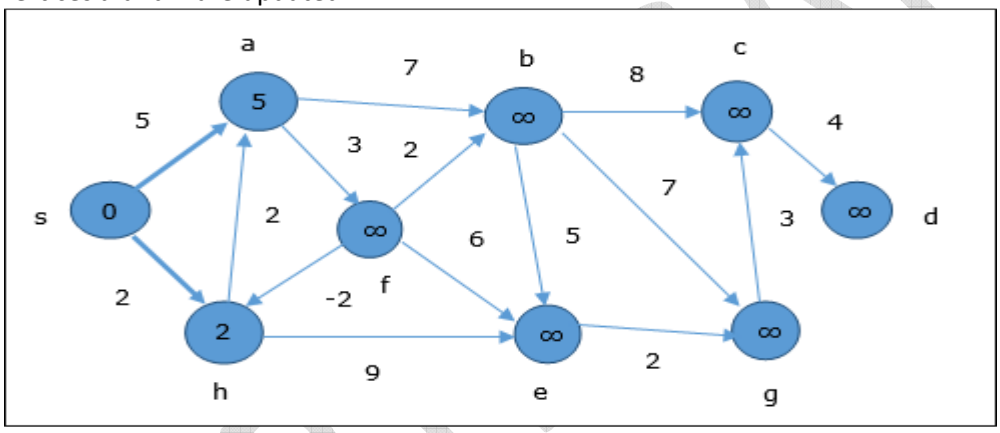
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

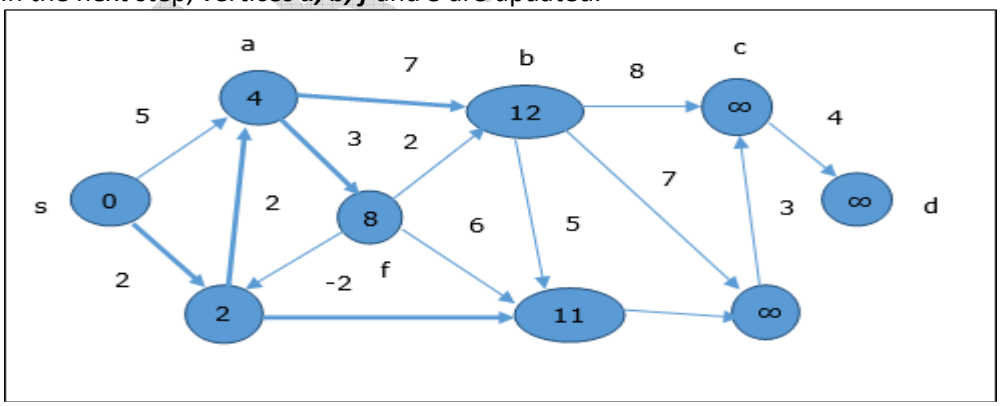
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



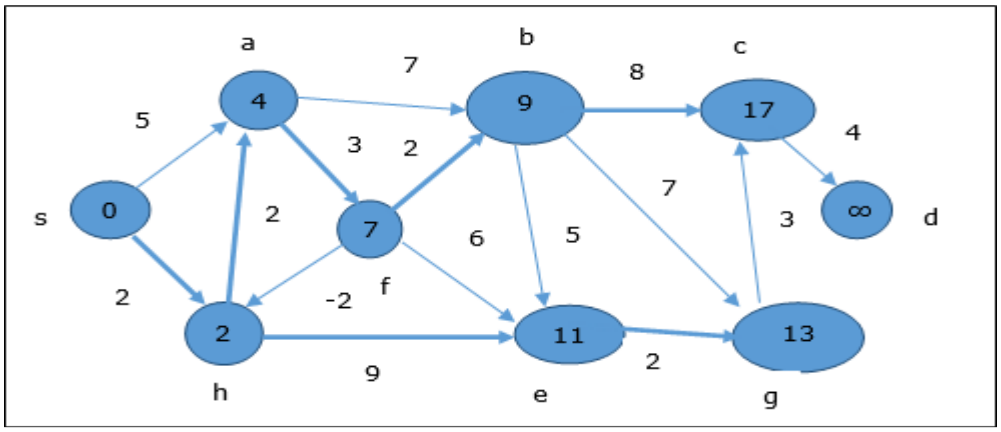
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



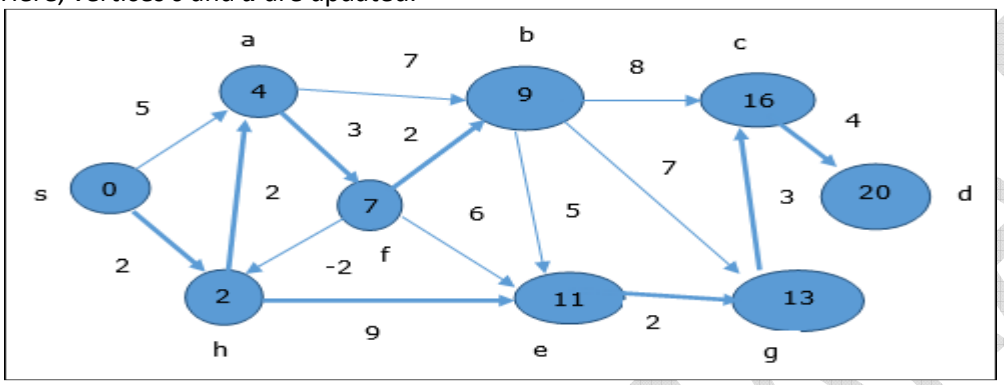
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20**.
Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

```
# Python program for Bellman-Ford's single source
# shortest path algorithm.
from collections import defaultdict

# Class to represent a graph
class Graph:
        def __init__(self, vertices):
                self.V = vertices # No. of vertices
                self.graph = [] # default dictionary to store graph

        # function to add an edge to graph
        def addEdge(self, u, v, w):
                self.graph.append([u, v, w])

        # utility function used to print the solution
        def printArr(self, dist):
                print("Vertex Distance from Source")
                for i in range(self.V):
                        print("% d \t\t % d" % (i, dist[i]))

        # The main function that finds shortest distances from src to
        # all other vertices using Bellman-Ford algorithm. The function
        # also detects negative weight cycle
        def BellmanFord(self, src):
                # Step 1: Initialize distances from src to all other vertices
                # as INFINITE
                dist = [float("Inf")] * self.V
                dist[src] = 0
```

```
        # Step 2: Relax all edges |V| - 1 times. A simple shortest
        # path from src to any other vertex can have at-most |V| - 1
        # edges
        for i in range(self.V - 1):
                # Update dist value and parent index of the adjacent vertices of
                # the picked vertex. Consider only those vertices which are still in
                # queue
                for u, v, w in self.graph:
                        if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                                dist[v] = dist[u] + w
```

```
        # Step 3: check for negative-weight cycles. The above step
        # guarantees shortest distances if graph doesn't contain
        # negative weight cycle. If we get a shorter path, then there
        # is a cycle.
        for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                        print "Graph contains negative weight cycle"
                        return
        # print all distance
        self.printArr(dist)
```

```
g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

# Print the solution
g.BellmanFord(0)
```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

Notes

1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.

2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Knapsack problem

Given a set of items, each with a weight and a value, determine a subset of items 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.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most

restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications
In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario
A thief is robbing a store and can carry a maximal weight of *W* into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?
In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack
Fractions of items can be taken rather than having to make binary (0-1) choices for each item. Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Steps to solve the Fractional Problem:

1. Compute the value per pound v_i/w_i for each item.
2. Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
4. Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.
5. **Fractional Knapsack**

Fractional Knapsack (Array v, Array w, int W)
1. for i= 1 to size (v)
2. do p [i] = v [i] / w [i]
3. Sort-Descending (p)
4. i ← 1
5. while (W>0)
6. do amount = min (W, w [i])
7. solution [i] = amount
8. W= W-amount
9. i ← i+1
10. return solution
Example: Consider 5 items along their respective weights and values: -
I = (I_1,I_2,I_3,I_4,I_5)
w = (5, 10, 20, 30, 40)
v = (30, 20, 100, 90,160)
The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of p_i.
First, we choose the item I_i whose weight is 5.
Then choose item I_3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25
Now the next item is I_5, and its weight is 40, but we want only 35, so we chose the fractional part of it,

i.e., $5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$

Weight = 5 + 20 + 35 = 60

Maximum Value:-

$$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$$

= 30 + 100 + 140 = 270 (Minimum Cost)

Solution:

ITEM	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	w_i	v_i	$P_i = \frac{v_i}{w_i}$
I_1	5	30	6
I_2	10	20	2
I_3	20	100	5
I_4	30	90	3
I_5	40	160	4

Now, arrange the value of pi in decreasing order.

ITEM	w_i	v_i	$p_i = \frac{v_i}{w_i}$
I_1	5	30	6
I_3	20	100	5
I_5	40	160	4
I_4	30	90	3
I_2	10	20	2

```
# Python3 program to solve fractional
# Knapsack Problem
class ItemValue:
        """Item Value DataClass"""
        def __init__(self, wt, val, ind):
                self.wt = wt
                self.val = val
                self.ind = ind
                self.cost = val // wt
```

```
        def __lt__(self, other):
                return self.cost < other.cost

# Greedy Approach
class FractionalKnapSack:
        """Time Complexity O(n log n)"""
        @staticmethod
        def getMaxValue(wt, val, capacity):
                """function to get maximum value """
                iVal = []
                for i in range(len(wt)):
                        iVal.append(ItemValue(wt[i], val[i], i))

                # sorting items by value
                iVal.sort(reverse = True)
                totalValue = 0
                for i in iVal:
                        curWt = int(i.wt)
                        curVal = int(i.val)
                        if capacity - curWt >= 0:
                                capacity -= curWt
                                totalValue += curVal
                        else:
                                fraction = capacity / curWt
                                totalValue += curVal * fraction
                                capacity = int(capacity - (curWt * fraction))
                                break
                return totalValue

# Driver Code
if __name__ == "__main__":
        wt = [10, 40, 20, 30]
        val = [60, 40, 100, 120]
        capacity = 50

        maxValue = FractionalKnapSack.getMaxValue(wt, val, capacity)
        print("Maximum value in Knapsack =", maxValue)
```

Output :
Maximum value in Knapsack = 240

0/1 Knapsack Problem: Dynamic Programming Approach:
Knapsack is basically means bag. A bag of given capacity.
We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

1. W ≤ capacity
2. Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.
The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:
1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not. It can be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:
In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

KNAPSACK (n, W)
1. for w = 0, W
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if ($w_i \leq w$ & v_i + V [i-1, w - w_i]> V [i -1,W])
7. then V [i, W] ← v_i + V [i - 1, w - w_i]
8. else V [i, W] ← V [i - 1, w]

Example of 0/1 Knapsack Problem:
Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$												
$w_2 = 2\ v_2 = 6$												
$w_3 = 5\ v_3 = 18$												
$w_4 = 6\ v_4 = 22$												
$w_5 = 7\ v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0											
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

V [i, j] = max {V [i - 1, j], v_i + V [i - 1, j -w_i]

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

V [3, 7] = max {V [3 - 1, 7], v_3 + V [3 - 1, 7 - w_3]

= max {V [2, 7], 18 + V [2, 7 - 5]}

= max {7, 18 + 6}

= 24

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry

```
#A naive recursive implementation of 0-1 Knapsack Problem
# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):
        # Base Case
        if n == 0 or W == 0 :
                return 0

        # If weight of the nth item is more than Knapsack of capacity
        # W, then this item cannot be included in the optimal solution
        if (wt[n-1] > W):
                return knapSack(W , wt , val , n-1)

        # return the maximum of two cases:
        # (1) nth item included
        # (2) not included
        else:
                return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                                knapSack(W , wt , val , n-1))

# end of function knapSack
```

```
# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)
```

Output:
220

Minimum Cost Spanning Trees

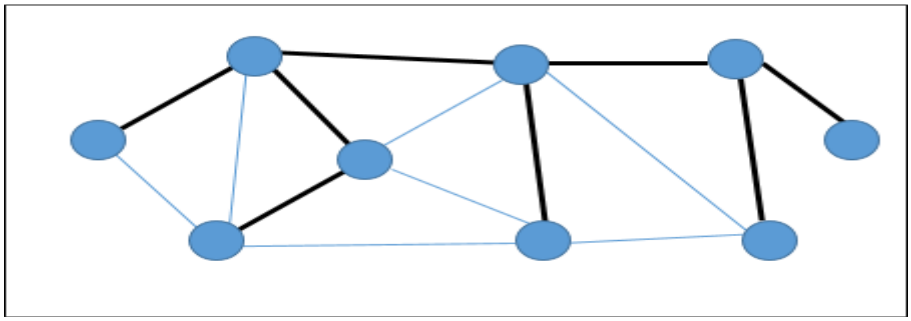
A spanning tree is a subset of an undirected Graph that has all the vertices connected by minimum number of edges. If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties of Spanning Tree:

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

Example

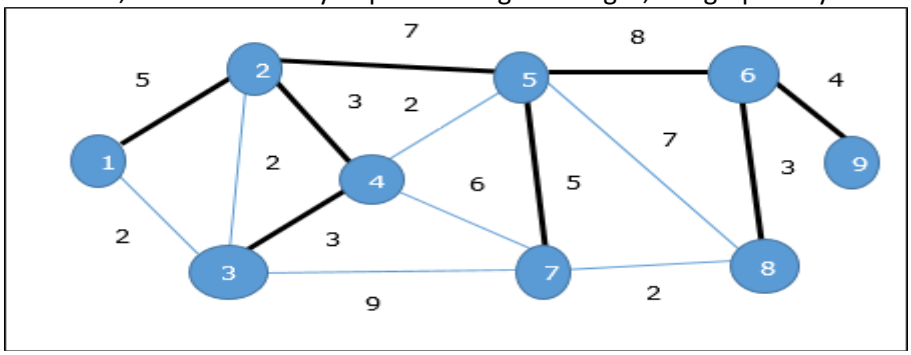
In the following graph, the highlighted edges form a spanning tree.



Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are n number of vertices, the spanning tree should have *n* - 1 number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph. Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.
If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

1. A ← Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A

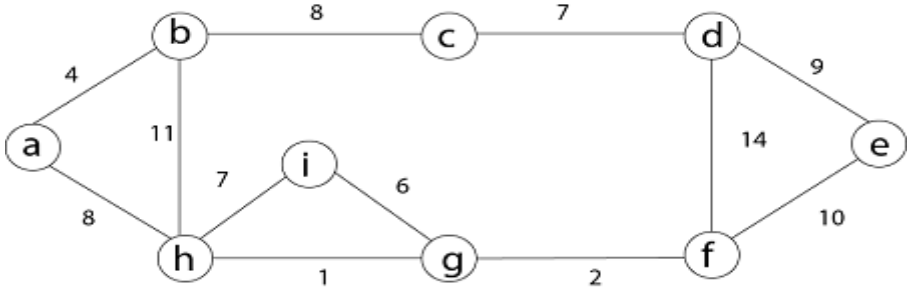
Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

1. O (E log E) = O (E log V).

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.

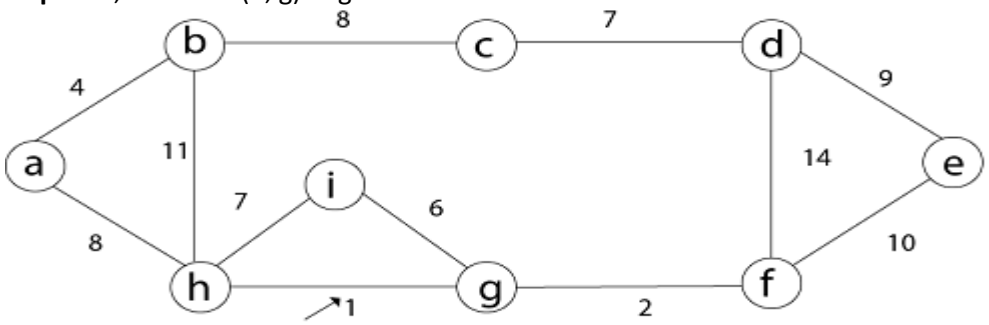


Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

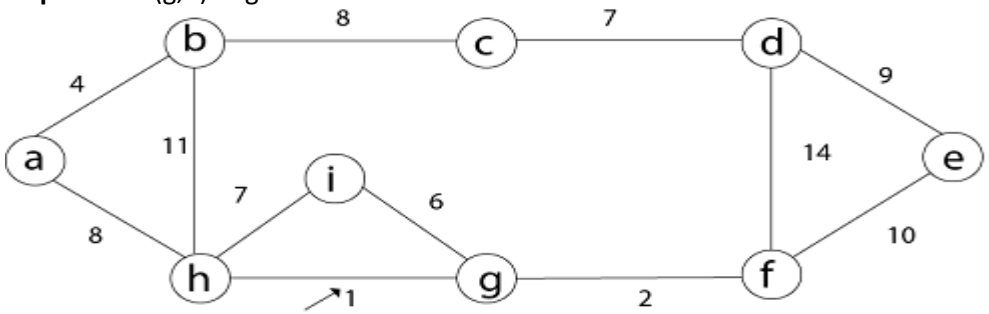
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

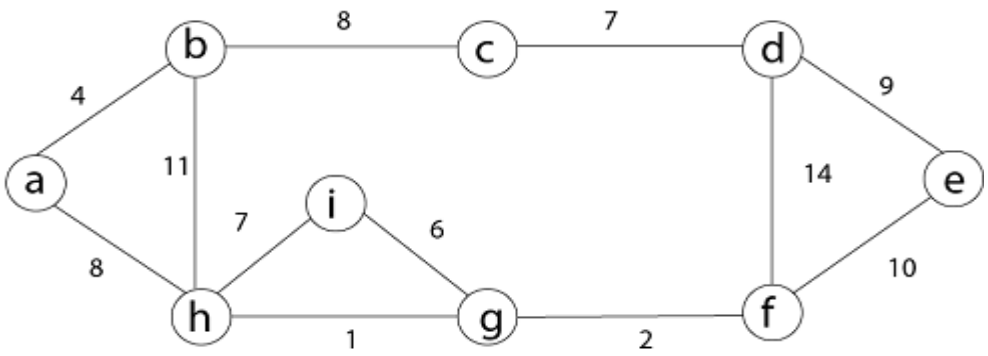
Step1: So, first take (h, g) edge



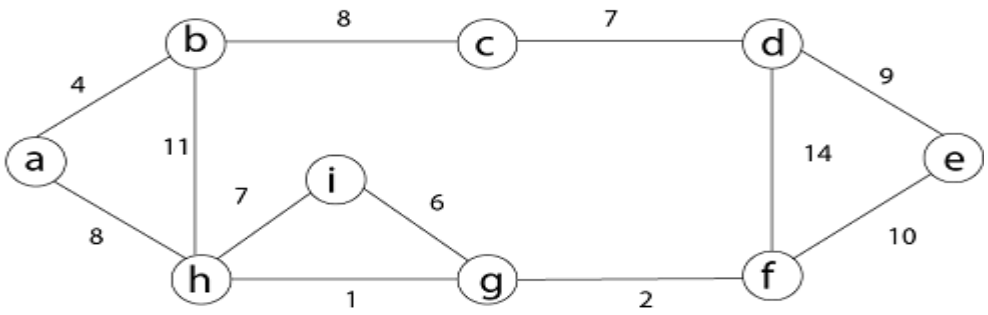
Step 2: then (g, f) edge.



Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes

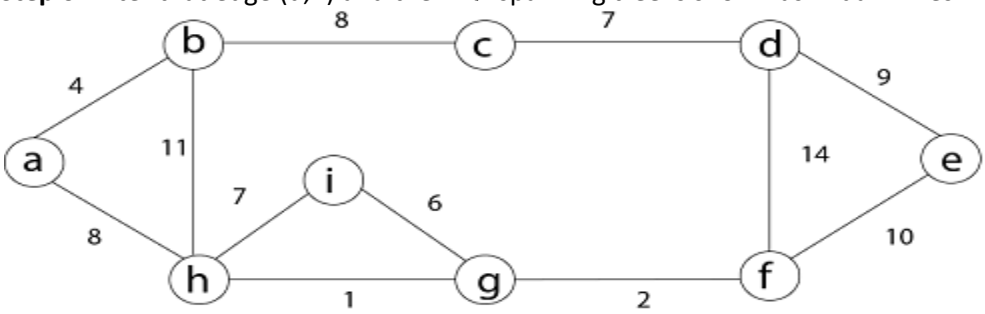


Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded. Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



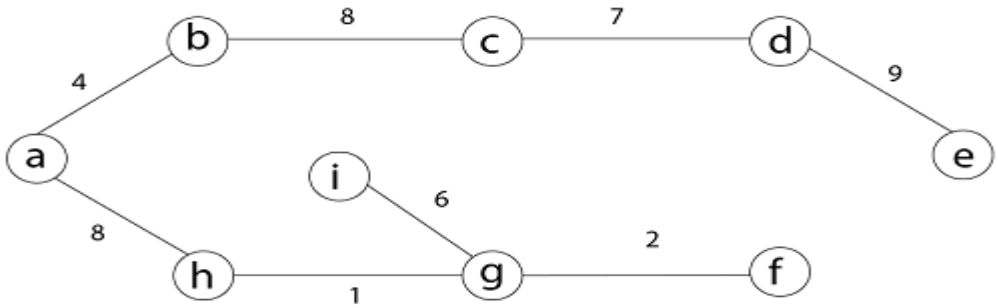
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph
from collections import defaultdict

#Class to represent a graph
class Graph:
        def __init__(self,vertices):
                self.V= vertices #No. of vertices
                self.graph = [] # default dictionary
                # to store graph

        # function to add an edge to graph
        def addEdge(self,u,v,w):
                self.graph.append([u,v,w])

        # A utility function to find set of an element i
        # (uses path compression technique)
```

```
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else :
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's
        # algorithm
    def KruskalMST(self):
        result =[] #This will store the resultant MST
        i = 0 # An index variable, used for sorted edges
        e = 0 # An index variable, used for result[]

        # Step 1: Sort all the edges in non-decreasing
        # order of their
        # weight. If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,key=lambda item: item[2])
        parent = [] ; rank = []
        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V -1 :
            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u,v,w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent ,v)

            # If including this edge does't cause cycle,
            # include it in result and increment the index
            # of result for next edge
            if x != y:
```

```
                    e = e + 1
                    result.append([u,v,w])
                    self.union(parent, rank, x, y)
            # Else discard the edge

    # print the contents of result[] to display the built MST
    print "Following are the edges in the constructed MST"
    for u,v,weight in result:
        #print str(u) + " -- " + str(v) + " == " + str(weight)
        print ("%d -- %d == %d" % (u,v,weight))

# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)
g.KruskalMST()
```

Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Time Complexity: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time.

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

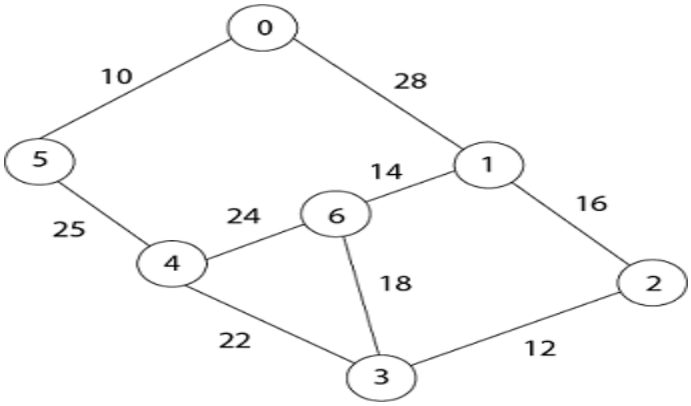
1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    - Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
    - Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

1. for each u □ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? □
7. do u ← EXTRACT - MIN (Q)
8. for each v □ Adj [u]
9. do if v □ Q and w (u, v) < key [v]
10. then π [v] ← u

11. key [v] ← w (u, v)

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



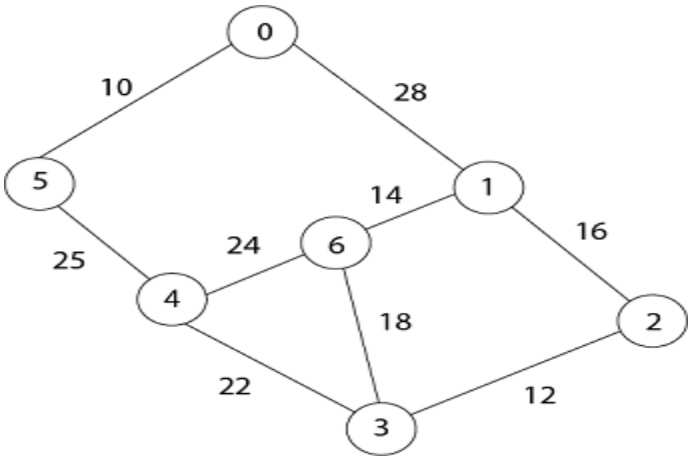
Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

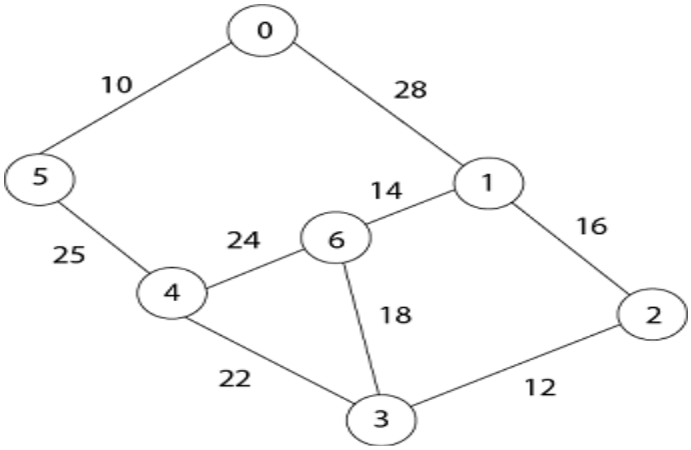
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞ π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞ key [6] = ∞
3. w (4,3) = 22 w (4,6) = 24
4. w (u, v) < key [v] w (u, v) < key [v]
5. w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.
And the parent of 3, 6 as 4.

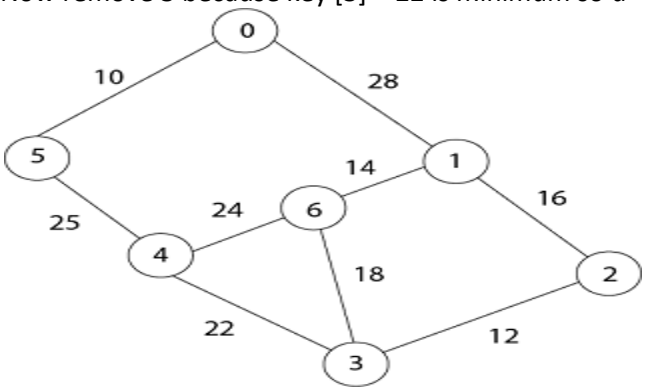
π[3]= 4 π[6]= 4

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

u = EXTRACT_MIN (3, 6) [key [3] < key [6]]

u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}

2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞ key [6] = 24
5. w (3, 2) = 12 w (3, 6) = 18
6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

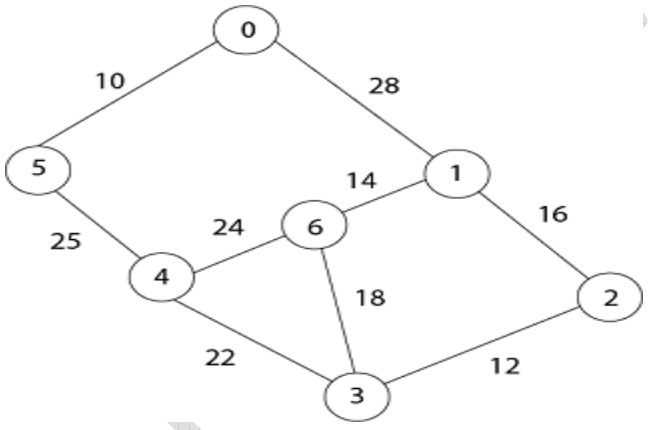
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

1. u = EXTRACT_MIN (2, 6)
2. u = 2 [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}
2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

Π [6] = 1

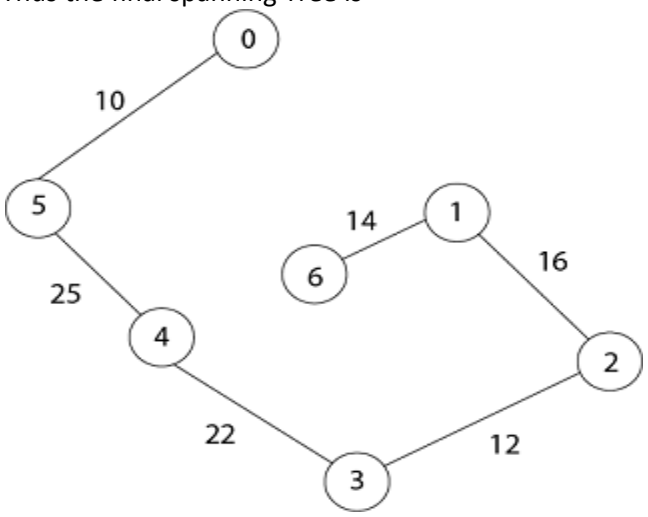
Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

0 → 5 → 4 → 3 → 2 → 1 → 6

[Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph
import sys # Library for INT_MAX

class Graph():
        def __init__(self, vertices):
                self.V = vertices
                self.graph = [[0 for column in range(vertices)]
                                        for row in range(vertices)]

        # A utility function to print the constructed MST stored in parent[]
        def printMST(self, parent):
                print "Edge \tWeight"
                for i in range(1, self.V):
                        print parent[i], "-", i, "\t", self.graph[i][ parent[i] ]

        # A utility function to find the vertex with
        # minimum distance value, from the set of vertices
        # not yet included in shortest path tree
        def minKey(self, key, mstSet):
                # Initilaize min value
                min = sys.maxint

                for v in range(self.V):
                        if key[v] < min and mstSet[v] == False:
                                min = key[v]
                                min_index = v
                return min_index
```

```
        # Function to construct and print MST for a graph
        # represented using adjacency matrix representation
        def primMST(self):
                # Key values used to pick minimum weight edge in cut
                key = [sys.maxint] * self.V
                parent = [None] * self.V # Array to store constructed MST
                # Make key 0 so that this vertex is picked as first vertex
                key[0] = 0
                mstSet = [False] * self.V
                parent[0] = -1 # First node is always the root of

                for cout in range(self.V):
                        # Pick the minimum distance vertex from
                        # the set of vertices not yet processed.
                        # u is always equal to src in first iteration
                        u = self.minKey(key, mstSet)

                        # Put the minimum distance vertex in
                        # the shortest path tree
                        mstSet[u] = True

                        # Update dist value of the adjacent vertices
                        # of the picked vertex only if the current
                        # distance is greater than new distance and
                        # the vertex in not in the shotest path tree
                        for v in range(self.V):
                                # graph[u][v] is non zero only for adjacent vertices of m
                                # mstSet[v] is false for vertices not yet included in MST
                                # Update the key only if graph[u][v] is smaller than key[v]
                                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                                        key[v] = self.graph[u][v]
                                        parent[v] = u
                self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],
                        [2, 0, 3, 8, 5],
                        [0, 3, 0, 0, 7],
                        [6, 8, 0, 0, 9],
                        [0, 5, 7, 9, 0]]
g.primMST();
```

Output:

```
Edge   Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Time Complexity of the above program is O(V^2).

Differentiate between Dynamic Programming and Greedy Method

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	3. More efficient as compared to a greedy approach
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

Geometric Algorithm

How to check if two given line segments intersect?

Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other.

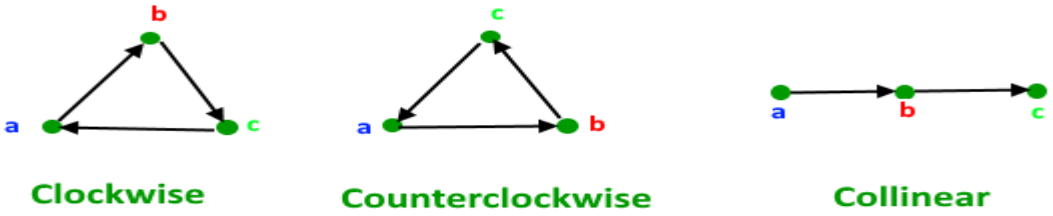
Before we discuss solution, let us define notion of orientation. Orientation of an ordered triplet of points in the plane can be

–counterclockwise

–clockwise

–colinear

The following diagram shows different possible orientations of (a, b, c)



How is Orientation useful here?

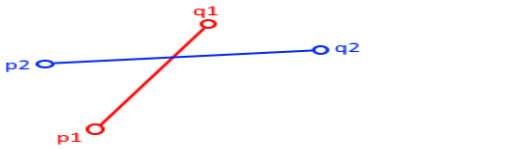
Two segments (p1,q1) and (p2,q2) intersect if and only if one of the following two conditions is verified

1. *General Case:*

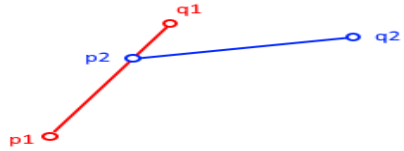
– (p1, q1, p2) and (p1, q1, q2) have different orientations and

– (p2, q2, p1) and (p2, q2, q1) have different orientations.

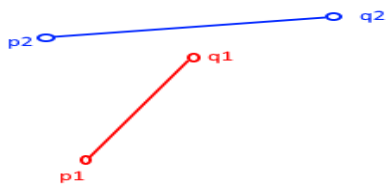
Examples:



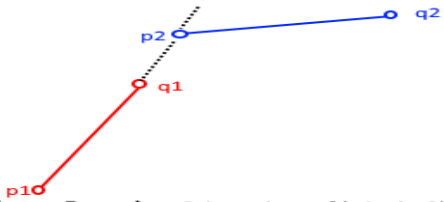
Example : Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also differnet.



Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) also different

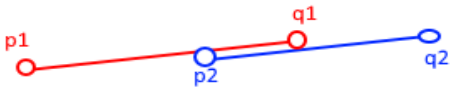


Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same

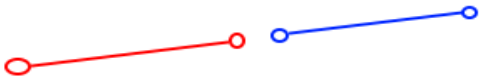


Example: Orientations of (p1,q1,p2) and (p1,q1,q2) are different. Orientations of (p2,q2,p1) and (p2,q2,q1) are same.

2. *Special Case*
– (p1, q1, p2), (p1, q1, q2), (p2, q2, p1), and (p2, q2, q1) are all collinear and
– the x-projections of (p1, q1) and (p2, q2) intersect
– the y-projections of (p1, q1) and (p2, q2) intersect
Examples:



Example: All points are collinear. The x-projections pf (p1,q1) and (p2,q2) intersect. The y-projection of (p1,q1) and (p2,q2) also intersect



Example: All points are collinear .The x-projections of (p1,q1) and (p2,q2) not intersect. The y-projection of (p1,q1) and (p2,q2) do not intersect

Given n line segments, find if any two segments intersect
Naive Algorithm A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. We can check two line segments in O(1) time. Therefore, this approach takes $O(n^2)$.

Sweep Line Algorithm: We can solve this problem in O(nLogn) time using Sweep Line Algorithm. The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.

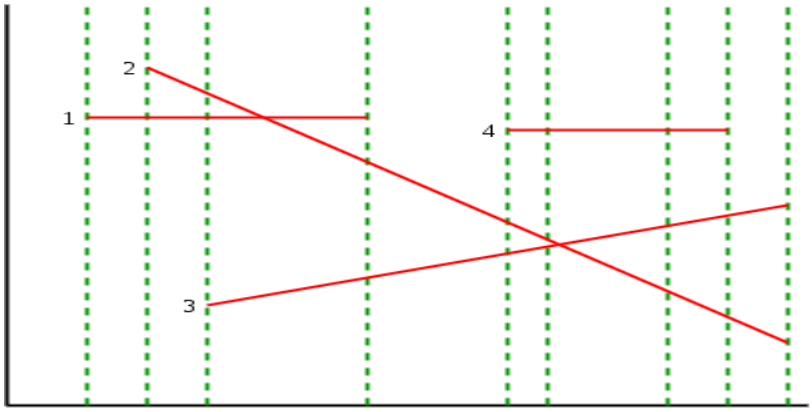
1. 1)Let there be n given lines. There must be 2n end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.
2. 2) Start from the leftmost point. Do following for every point
    a. If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to *active* line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.
    b. If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm. The Sweep Line technique is useful in many other geometric algorithms like calculating the 2D Voronoi diagram

What data structures should be used for efficient implementation?
In step 2, we need to store all active line segments. We need to do following operations efficiently:
a) Insert a new line segment
b) Delete a line segment
c) Find predecessor and successor according to y coordinate values
The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in O(Logn) time.

Also, in step 1, instead of sorting, we can use min heap data structure. Building a min heap takes O(n) time and every extract min operation takes O(Logn) time

Example:
Let us consider the following example taken from here. There are 5 line segments 1, 2, 3, 4 and 5. The dotted green lines show sweep lines.



Sweep Lines

Following are steps followed by the algorithm. All points from left to right are processed one by one. We maintain a self-balancing binary search tree.

Left end point of line segment 1 is processed: 1 is inserted into the Tree. The tree contains 1. No intersection.

Left end point of line segment 2 is processed: Intersection of 1 and 2 is checked. 2 is inserted into the Tree. No intersection. The tree contains 1, 2.

Left end point of line segment 3 is processed: Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.

Right end point of line segment 1 is processed: 1 is deleted from the Tree. Intersection of 2 and 3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3.

Left end point of line segment 4 is processed: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.

Left end point of line segment 5 is processed: Intersection of 5 with 3 is checked. No intersection. 5 is inserted into the Tree. The tree contains 2, 4, 3, 5.

Right end point of line segment 5 is processed: 5 is deleted from the Tree. The tree contains 2, 4, 3.

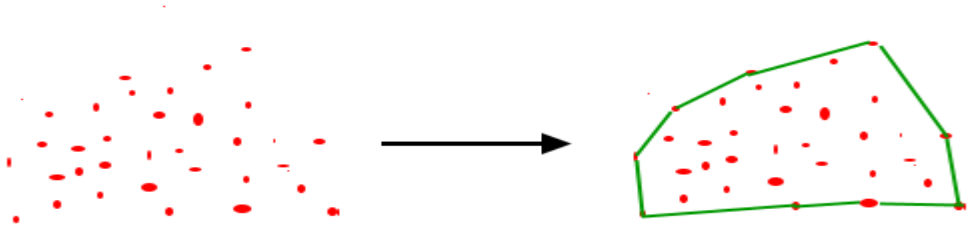
Right end point of line segment 4 is processed: 4 is deleted from the Tree. The tree contains 2, 4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported. The tree contains 2, 3. Note that the intersection of 2 and 3 is reported again. We can add some logic to check for duplicates.

Right end point of line segment 2 and 3 are processed: Both are deleted from tree and tree becomes empty.

Time Complexity: The first step is sorting which takes O(nLogn) time. The second step process 2n points and for processing every point, it takes O(Logn) time. Therefore, overall time complexity is O(nLogn)

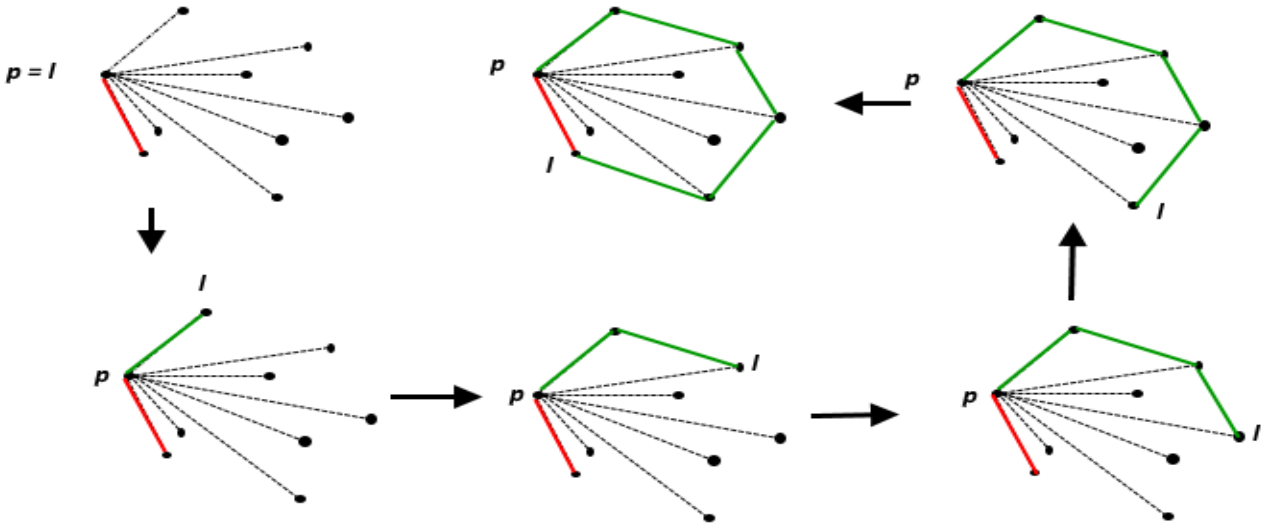
Convex Hull (Jarvis's Algorithm or Wrapping)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output? The idea is to use orientation () here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, q, r) = counterclockwise". Following is the detailed algorithm.

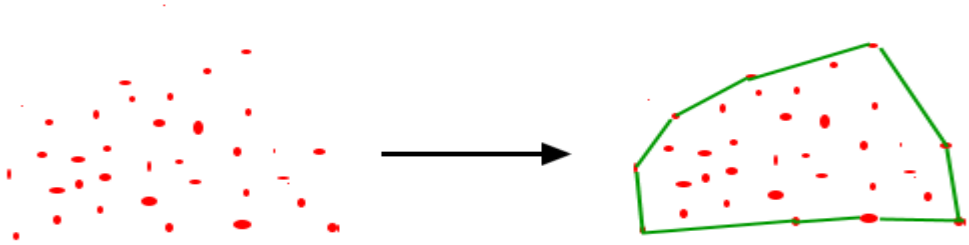
1) Initialize p as leftmost point.
2) Do following while we don't come back to the first (or leftmost) point.
…..a) The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r.
…..b) next[p] = q (Store q as next of p in the output convex hull).
…..c) p = q (Set p as q for next iteration).



The execution of jarvis's March

Time Complexity: For every point on the hull we examine all the other points to determine the next point. Time complexity is ?(m * n) where n is number of input points and m is number of output or hull points (m <= n). In worst case, time complexity is O(n^2). The worst case occurs when all the points are on the hull (m = n)

Convex Hull (Graham Scan)
Given a set of points in the plane. The convex hull of the set is the smallest convex polygon that contains all the points of it.



The worst case time complexity of Jarvis's Algorithm is O(n^2). Using Graham's scan algorithm, we can find Convex Hull in O(nLogn) time. Following is Graham's algorithm
Let points[0..n-1] be the input array.

1) Find the bottom-most point by comparing y coordinate of all points. If there are two points with the same y value, then the point with smaller x coordinate value is considered. Let the bottom-most point be P0. Put P0 at first position in output hull.
2) Consider the remaining n-1 points and sort them by polar angle in counterclockwise order around points[0]. If the polar angle of two points is the same, then put the nearest point first.
3 After sorting, check if two or more points have the same angle. If two more points have the same angle, then remove all same angle points except the point farthest from P0. Let the size of the new array be m.
4) If m is less than 3, return (Convex Hull not possible)
5) Create an empty stack 'S' and push points[0], points[1] and points[2] to S.
6) Process remaining m-3 points one by one. Do following for every point 'points[i]'
4.1) Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they

don’t make a left turn).
    a) Point next to top in stack
    b) Point at the top of stack
    c) points[i]
4.2) Push points[i] to S
5) Print contents of S
The above algorithm can be divided into two phases.

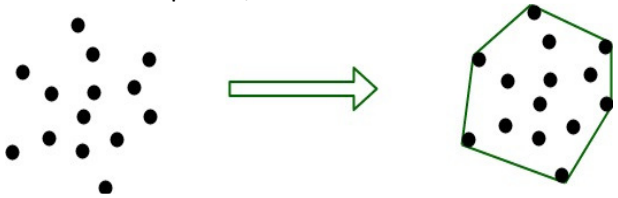
Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points be sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See the following diagram).
What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the compare() function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev(p), curr(c) and next(n). If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase

Time Complexity: Let n be the number of input points. The algorithm takes O(nLogn) time if we use a O(nLogn) sorting algorithm.
The first step (finding the bottom-most point) takes O(n) time. The second step (sorting points) takes O(nLogn) time. The third step takes O(n) time. In the third step, every element is pushed and popped at most one time. So the sixth step to process points one by one takes O(n) time, assuming that the stack operations take O(1) time. Overall complexity is O(n) + O(nLogn) + O(n) + O(n) which is O(nLogn)

Quickhull Algorithm for Convex Hull

Given a set of points, a Convex hull is the smallest convex polygon containing all the given points.



Input is an array of points specified by their x and y coordinates. Output is a convex hull of this set of points in ascending order of x coordinates.

Example :
Input : points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},{0, 0}, {1, 2}, {3, 1}, {3, 3}};
Output : The points in convex hull are: (0, 0) (0, 3) (3, 1) (4, 4)

Input : points[] = {{0, 3}, {1, 1}
Output : Not Possible
There must be at least three points to form a hull.

Input : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0), (0, -6), (1, 0)};
Output : (-4, 0), (5, 0), (0, -6), (0, 4)

The QuickHull algorithm is a Divide and Conquer algorithm similar to Quick Sort. Let a[0…n-1] be the input array of points. Following are the steps for finding the convex hull of these points.

1. Find the point with minimum x-coordinate lets say, min_x and similarly the point with maximum x-coordinate, max_x.

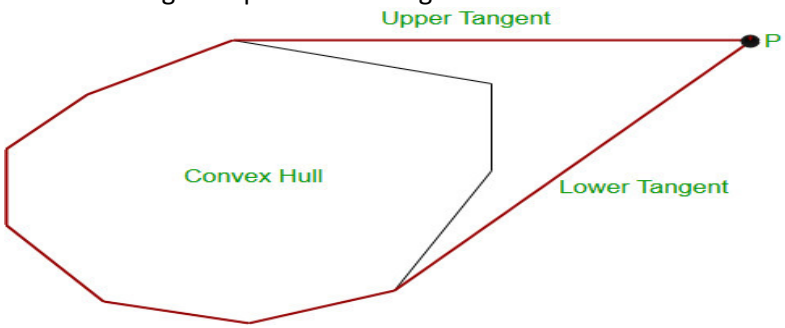
2. Make a line joining these two points, say **L**. This line will divide the whole set into two parts. Take both the parts one by one and proceed further.
3. For a part, find the point P with maximum distance from the line L. P forms a triangle with the points min_x, max_x. It is clear that the points residing inside this triangle can never be the part of convex hull.
4. The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points P and min_x and the line joining the points P and max_x are new lines and the points residing outside the triangle is the set of points. Repeat point no. 3 till there no point left with the line. Add the end points of this point to the convex hull.

Time Complexity: The analysis is similar to Quick Sort. On average, we get time complexity as O(n Log n), but in worst case, it can become $O(n^2)$

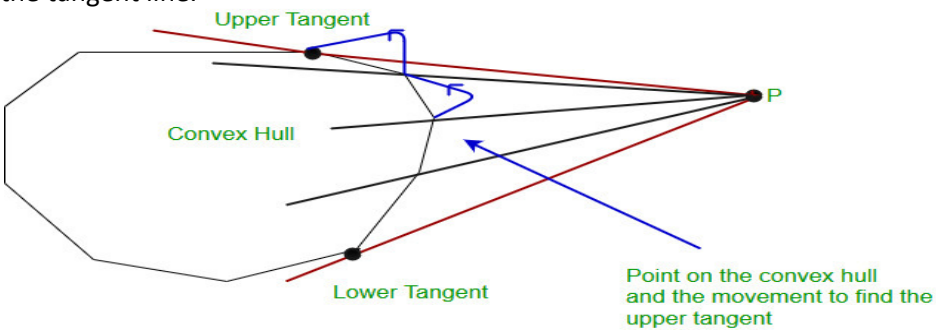
Dynamic Convex hull | Adding Points to an Existing Convex Hull

Given a convex hull, we need to add a given number of points to the convex hull and print the convex hull after every point addition. The points should be in anti-clockwise order after addition of every point.
Examples:
Input :
Convex Hull : (0, 0), (3, -1), (4, 5), (-1, 4)
Point to add : (100, 100)

Output :
New convex hull : (-1, 4) (0, 0) (3, -1) (100, 100)
We first check whether the point is inside the given convex hull or not. If it is, then nothing has to be done we directly return the given convex hull. If the point is outside the convex hull, we find the lower and upper tangents, and then merge the point with the given convex hull to find the new convex hull, as shown in the figure.



The red outline shows the new convex hull after merging the point and the given convex hull.
To find the upper tangent, we first choose a point on the hull that is nearest to the given point. Then while the line joining the point on the convex hull and the given point crosses the convex hull, we move anti-clockwise till we get the tangent line.



The figure shows the moving of the point on the convex hull for finding the upper tangent.
Note: It is assumed here that the input of the initial convex hull is in the anti-clockwise order, otherwise we have to first sort them in anti-clockwise order then apply the following code.

Deleting points from Convex Hull

Given a fixed set of points. We need to find convex hull of given set. We also need to find convex hull when a point is removed from the set.

Example:
Initial Set of Points: (-2, 8) (-1, 2) (0, 1) (1, 0) (-3, 0) (-1, -9) (2, -6) (3, 0) (5, 3) (2, 5)
Initial convex hull:- (-2, 8) (-3, 0) (-1, -9) (2, -6)(5, 3)
Point to remove from the set : (-2, 8)
Final convex hull: (2, 5) (-3, 0) (-1, -9) (2, -6) (5, 3)

Closest Pair of Points using Divide and Conquer algorithm

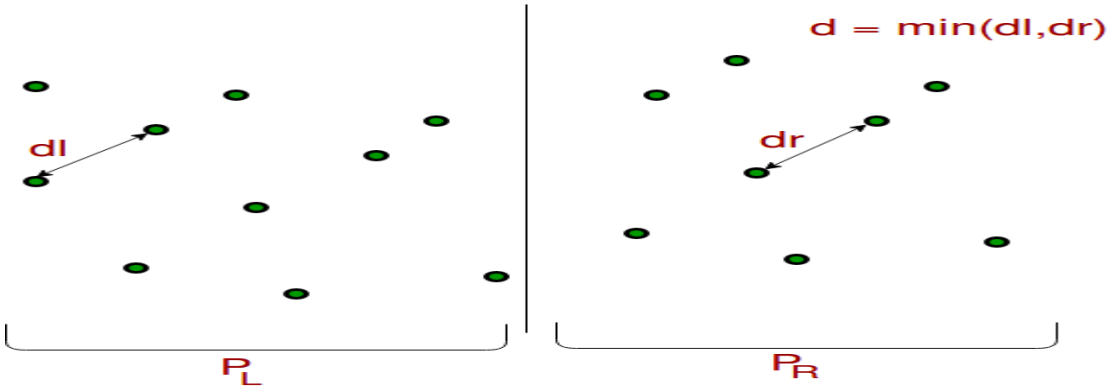
We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

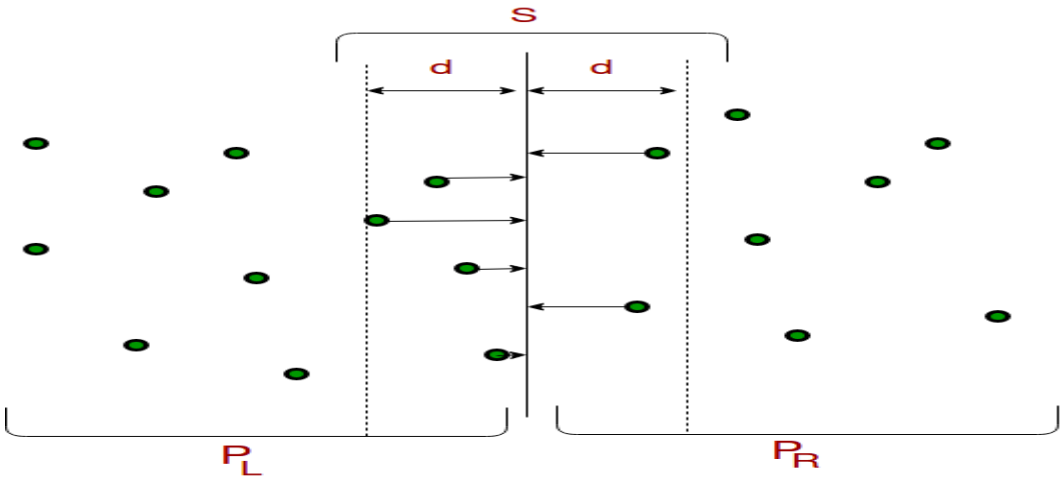
The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy. In this post, a O(n x (Logn)^2) approach is discussed.

Algorithm

Following are the detailed steps of a O(n (Logn)^2) algortihm.
Input: An array of n points *P[]*
Output: The smallest distance between two points in the given array.
As a pre-processing step, the input array is sorted according to x coordinates.
1) Find the middle point in the sorted array, we can take *P[n/2]* as middle point.
2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].
3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.



4) From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.



5)Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.
6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).
7) Finally return the minimum of d and distance calculated in the above step (step 6)

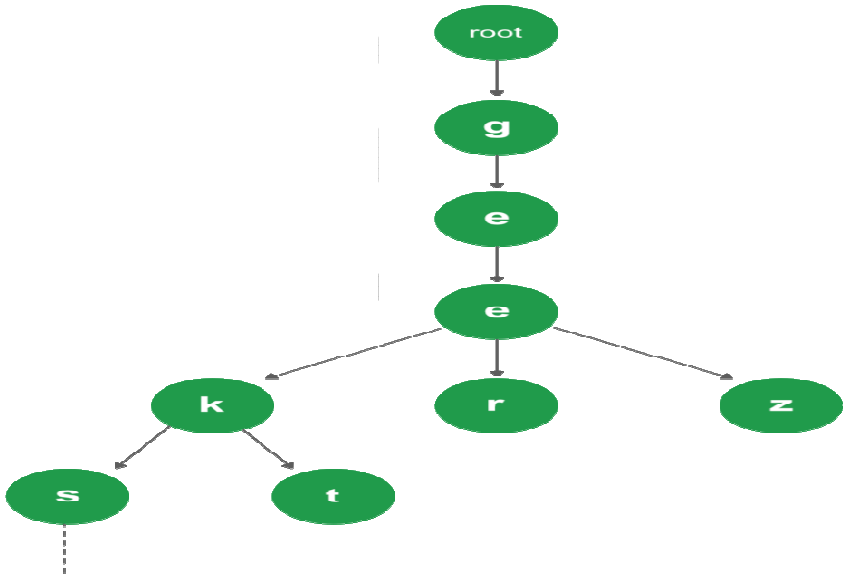
Time Complexity Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(nLogn) + O(n)
T(n) = 2T(n/2) + O(nLogn)
T(n) = T(n x Logn x Logn)

Notes
1) Time complexity can be improved to O(nLogn) by optimizing step 5 of the above algorithm. We will soon be discussing the optimized solution in a separate post.
2) The code finds smallest distance. It can be easily modified to find the points with the smallest distance.
3) The code uses quick sort which can be O(n^2) in the worst case. To have the upper bound as O(n (Logn)^2), a O(nLogn) sorting algorithm like merge sort or heap sort can be used

Internet Algorithm

Trie

Trie is an efficient information re***Trie***val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements



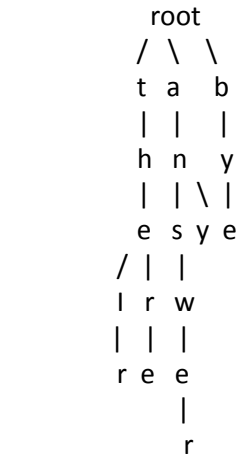
Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to distinguish the node as end of word node. A simple structure to represent nodes of the English alphabet can be as following,

```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a word
```

```
    bool isEndOfWord;
};
```

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.
The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type trie_node_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

nsert and search costs **O(key_length)**, however the memory requirements of Trie is **O(ALPHABET_SIZE * key_length * N)** where N is number of keys in Trie. There are efficient representation of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements of trie.

Trie | (Delete)

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

```
# Python program for insert and search
# operation in a Trie
class TrieNode:

        # Trie node class
        def __init__(self):
                self.children = [None]*26
                # isEndOfWord is True if node represent the end of the word
                self.isEndOfWord = False

class Trie:
```

```
	# Trie data structure class
	def __init__(self):
		self.root = self.getNode()

	def getNode(self):
		# Returns new trie node (initialized to NULLs)
		return TrieNode()

	def _charToIndex(self,ch):
		# private helper function
		# Converts key current character into index
		# use only 'a' through 'z' and lower case
		return ord(ch)-ord('a')


	def insert(self,key):
		# If not present, inserts key into trie
		# If the key is prefix of trie node,
		# just marks leaf node
		pCrawl = self.root
		length = len(key)
		for level in range(length):
			index = self._charToIndex(key[level])

			# if current character is not present
			if not pCrawl.children[index]:
				pCrawl.children[index] = self.getNode()
			pCrawl = pCrawl.children[index]
		# mark last node as leaf
		pCrawl.isEndOfWord = True

	def search(self, key):
		# Search key in the trie
		# Returns true if key presents
		# in trie, else false
		pCrawl = self.root
		length = len(key)
		for level in range(length):
			index = self._charToIndex(key[level])
			if not pCrawl.children[index]:
				return False
			pCrawl = pCrawl.children[index]
		return pCrawl != None and pCrawl.isEndOfWord

# driver function
def main():
	# Input keys (use only 'a' through 'z' and lower case)
	keys = ["the","a","there","anaswe","any",
			"by","their"]
	output = ["Not present in trie",
			"Present in trie"]

	# Trie object
	t = Trie()
	# Construct trie
```

```
        for key in keys:
                t.insert(key)
        # Search for different keys
        print("{} ---- {}".format("the",output[t.search("the")]))
        print("{} ---- {}".format("these",output[t.search("these")]))
        print("{} ---- {}".format("their",output[t.search("their")]))
        print("{} ---- {}".format("thaw",output[t.search("thaw")]))

if __name__ == '__main__':
        main()
```

Output :
the --- Present in trie
these --- Not present in trie
their --- Present in trie
thaw --- Not present in trie

Why Trie? :-

1. With Trie, we can insert and find strings in *O(L)* time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)
2. Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.
3. We can efficiently do prefix search (or auto-complete) with Trie.

Issues with Trie :-
The main disadvantage of tries is that they need a lot of memory for storing the strings. For each node we have too many node pointers(equal to number of characters of the alphabet), if space is concerned, then Ternary Search Tree can be preferred for dictionary implementations. In Ternary Search Tree, the time complexity of search operation is O(h) where h is the height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing, and nearest neighbor search.
The final conclusion is regarding *tries data structure* is that they are faster but require *huge memory* for storing the strings.

Ukkonen's Suffix Tree

Suffix Tree is very useful in numerous string processing and computational biology problems.
A suffix tree **T** for a m-character string S is a rooted directed tree with exactly m leaves numbered 1 to **m.** (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i…m].
Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will used zero indexed position)
For string S = xabxac with m = 6, suffix tree will look like following:

It has one root node and two internal nodes and 6 leaf nodes.
String Depth of red path is 1 and it represents suffix c starting at position 6
String Depth of blue path is 4 and it represents suffix bxca starting at position 3
String Depth of green path is 2 and it represents suffix ac starting at position 5
String Depth of orange path is 6 and it represents suffix xabxac starting at position 1

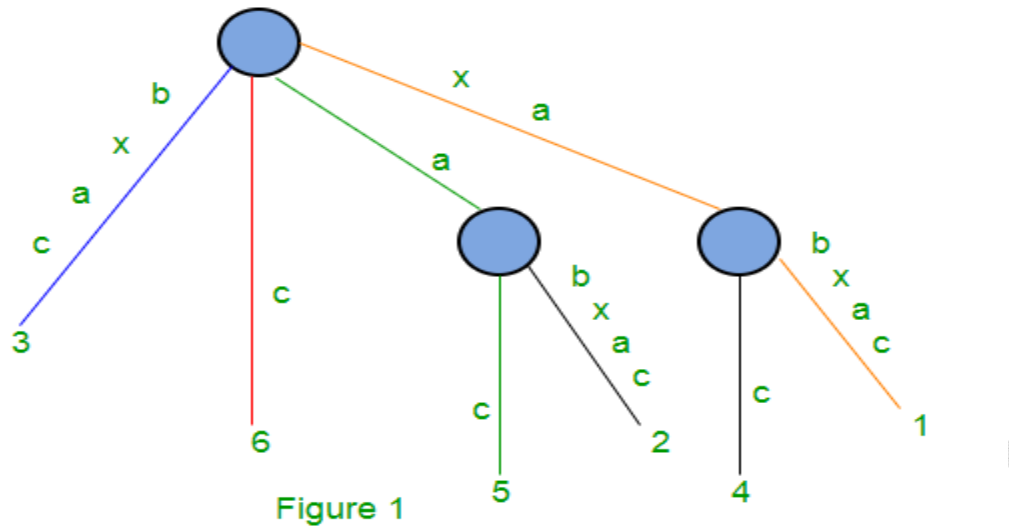


Figure 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)
If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.
For String S = xabxa, with m = 5, following is the suffix tree:

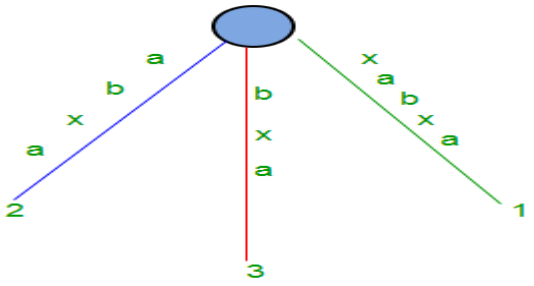


Figure 2

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.
Path for suffixes ‘xa’ and ‘a’ do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes (‘xa’ and ‘a’) are not seen explicitly in tree.
To avoid this problem, we add a character which is not present in string already. We normally use $, # etc as termination characters.
Following is the suffix tree for string S = xabxa$ with m = 6 and now all 6 suffixes end at leaf.

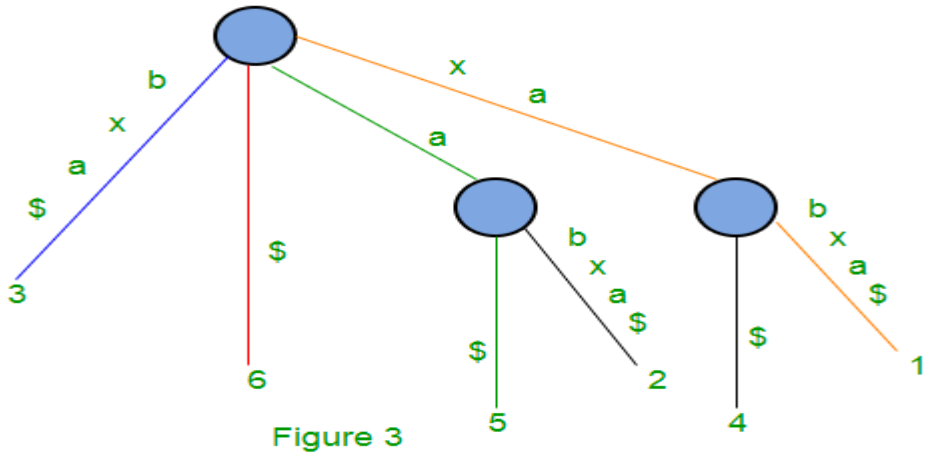


Figure 3

A naive algorithm to build a suffix tree

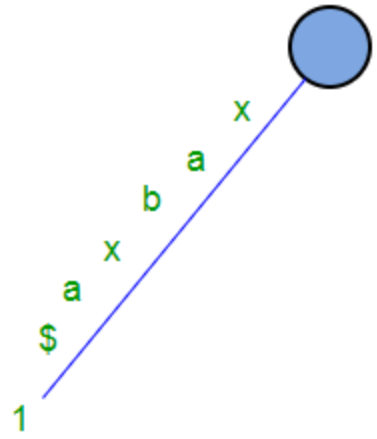
Given a string S of length m, enter a single edge for suffix S[I ..m]$ (the entire string) into the tree, then successively enter suffix S[i..m]$ into the growing tree, for i increasing from 2 to m. Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i.

So N_i+1 is constructed from N_i as follows:

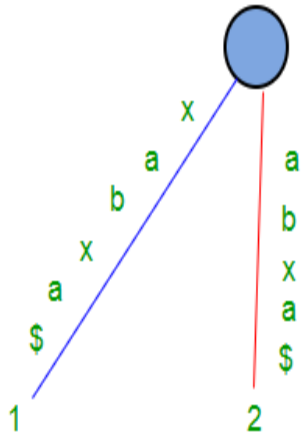
- Start at the root of N_i
- Find the longest path from the root which matches a prefix of S[i+1..m]$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v), break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in S[i+l..m] and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with S[i+1..m], and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge (w, i+1) from w to a new leaf labelled i+1 and it labels the new edge with the unmatched part of suffix S[i+1..m]

This takes $O(m^2)$ to build the suffix tree for the string S of length m.

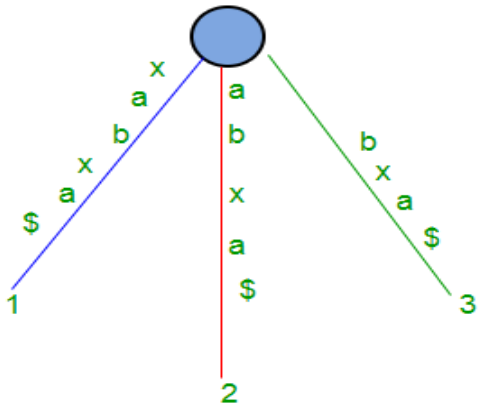
Following are few steps to build suffix tree based for string “xabxa$” based on above algorithm:



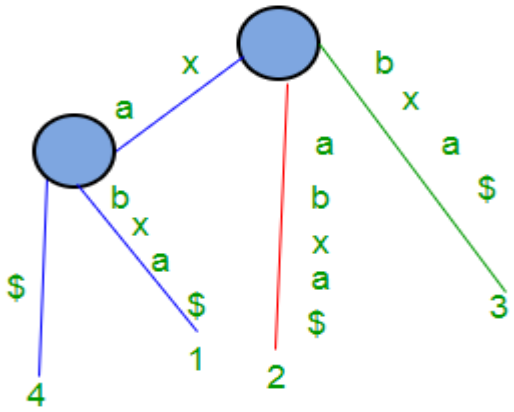
Tree with suffix N1, S[1....6]
Figure 4



Tree with suffix N1, S[1....6] and N2, S[2...6]
Figure 5



Tree with suffixes N1,N2 and N3
Figure 6



Tree with suffix N1, N2, N3 and N4
Figure 7

Implicit suffix tree

While generating suffix tree using Ukkonen’s algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with $ (or # or any other

termination character) label and no internal node with only one edge going out of it.
To get implicit suffix tree from a suffix tree S$,

- Remove all terminal symbol $ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

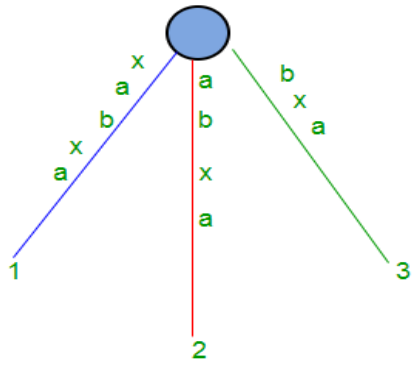


Figure 8 : Implicit suffix tree for storing xabxa
Suffix tree shown in Figure 3

High Level Description of Ukkonen's algorithm
Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix S[I ..i] of S (of length m).
It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, …, T_m using mth character.
Implicit suffix tree T_i+1 is built on top of implicit suffix tree T_i.
The true suffix tree for S is built from T_m by adding $.
At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.
Time taken is O(m).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)
In phase i+1, tree T_i+1 is built from tree T_i.

Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]
In extension j of phase i+1, the algorithm first finds the end of the path from the root labelled with substring S[j..i].
It then extends the substring by adding the character S(i+1) to its end (if it is not there already).
In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already).
In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
In extension 3 of phase i+1, we put string S[3..i+1] in the tree. Here S[3..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
.
.
In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label S[i+1].

High Level Ukkonen's algorithm

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
    For j from 1 to i+1
        begin {extension j}
        Find the end of the path from the root labelled S[j..i] in the current tree.
        Extend that path by adding character S[i+l] if it is not there already
```

```
    end;
end;
```

Suffix extension is all about adding the next character into the suffix tree built so far.
In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 extension rules:
Rule 1: If the path from the root labelled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.
Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

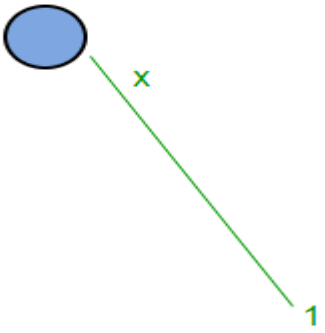


Figure 9 : T1 for S[1...1]
Adding suffixes of x(x)
Rule2-A new leaf node

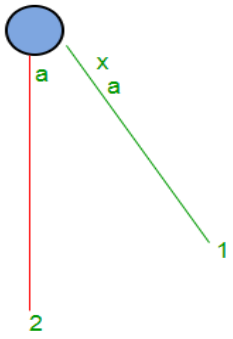


Figure 10 : T2 for S[1...2]
Adding suffixes of xa(xa and a)
Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

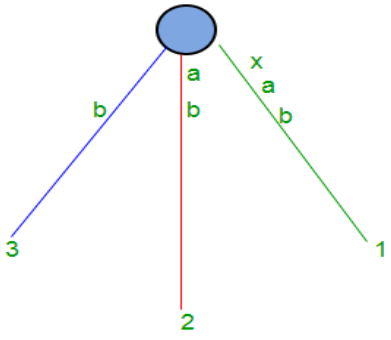


Figure 11 : T3 for S[1...3]
Adding suffixes of xab(xab,ab and b)
Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

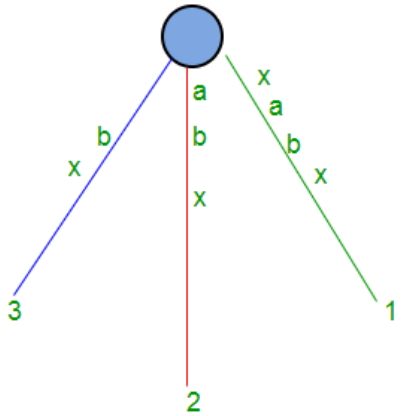


Figure 12 : T4 for S[1...4]
Adding suffixes of xabx(xabx,abx,bx and x)
Rule1-Extending path label in existing leaf edge
Rule3-Do nothing(path with label x already present)

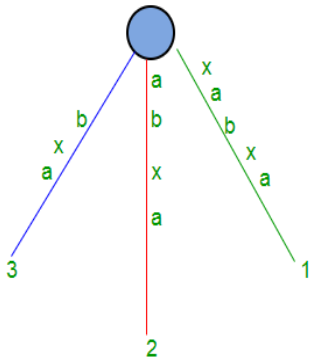


Figure 13 : T5 for S[1...5]
Adding suffixes of xabxa(xabxa,abxa,bxa,xa and x)
Rule1-Extending path label in existing leaf edge
Rule3-Do nothing(path with label xa and a already present)

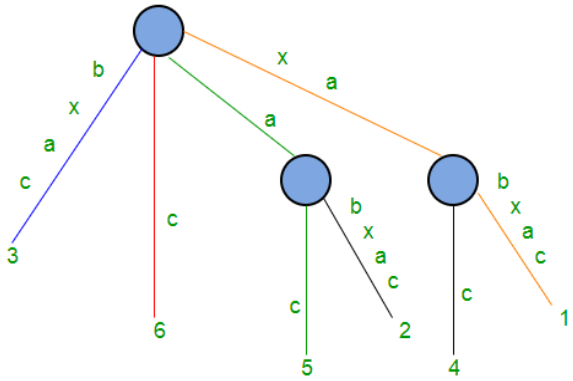


Figure 14 : T6 for S[1...6]
Adding suffixes of xabxac(xabxac, abxac, bxac, xac, ac, c)
Rule1-Extending path label in Existing leaf edge.
Rule2-Three new leaf edges and two new internal nodes

In Suffix Tree Construction of string S of length m, there are m phases and for a phase j (1 <= j <= m), we add jth character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules

To do jth extension of phase i+1 (adding character S[i+1]), we first need to find end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. This will take O(m3) time to build the suffix tree. Using few observations and implementation tricks, it can be done in O(m) which we will see now.

Suffix links

For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.

If A is empty string, suffix link from internal node will go to root node.

There will not be any suffix link from root node (As it's not considered as internal node).

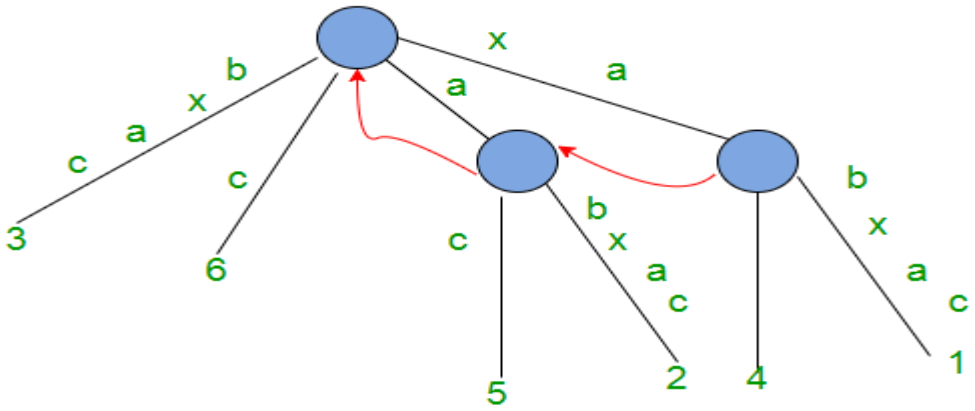


Figure 15 : Suffix links in red arrows

In extension j of some phase i, if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension j+1 of same phase i, we will create a suffix link from the internal node created in jth extension to the node with path labelled A.

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.
In any implicit suffix tree Ti after phase i, if internal node v has path-label xA, then there is a node s(v) in Ti with path-label A and node v will point to node s(v) using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.
How suffix links are used to speed up the implementation?
In extension j of phase i+1, we need to find the end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. Suffix links provide a short cut to find end of the path.

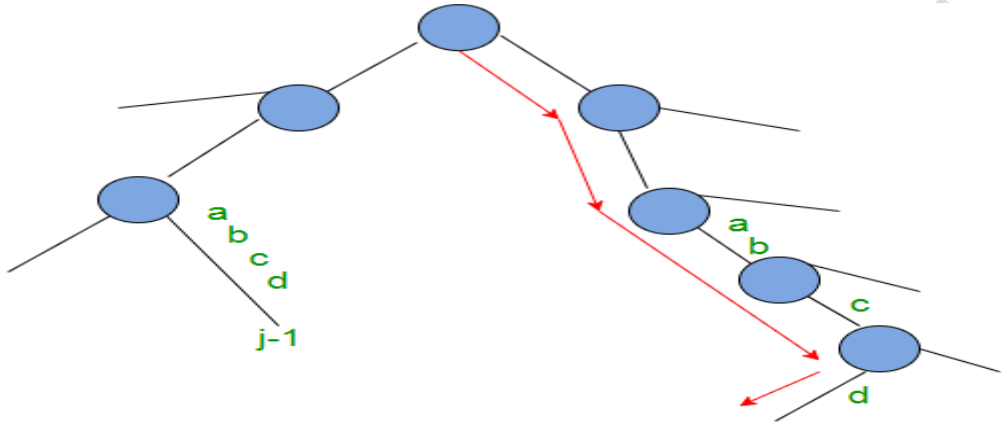


Figure 16 : Traversal from root to leaf in extension j of phase i+1, to find end of S[j....i], when suffix link is not used.

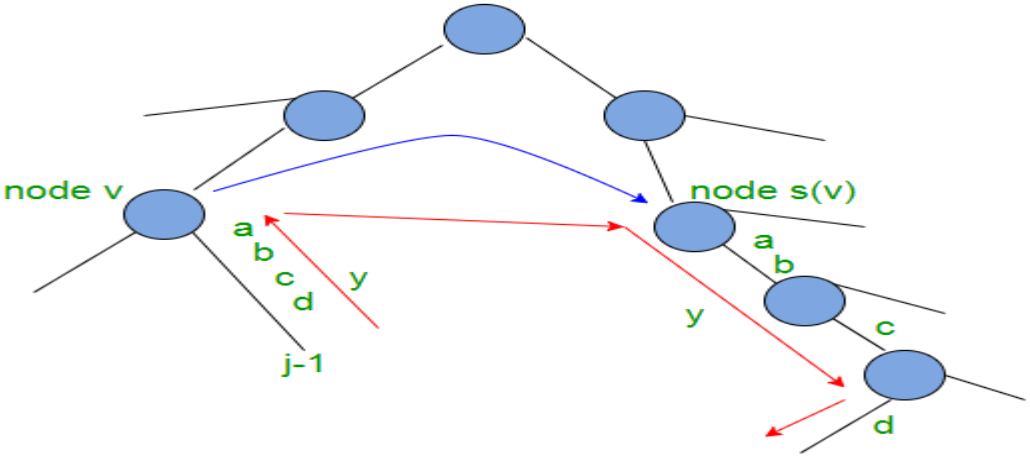


Figure 17 : Traversal from root to leaf in extension j of phase i+1, to find end of S[j....i], when suffix link(blue arrow) is used.

So we can see that, to find end of path S[j..i], we need not traverse from root. We can start from the end of path S[j-1..i], walk up one edge to node v (i.e. go to parent node), follow the suffix link to s(v), then walk down the path y (which is abcd here in Figure 17).
This shows the use of suffix link is an improvement over the process.
Note: In the next part 3, we will introduce activePoint which will help to avoid “walk up”. We can directly go to node s(v) from node v.

When there is a suffix link from node v to node s(v), then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node s(v) to a leaf. In Figure 17, there is a path label “abcd” from node v to a leaf, then there is a path will same label “abcd” from node s(v) to a leaf.
This fact can be used to improve the walk from s(v) to leaf along the path y. This is called “skip/count” trick.

Skip/Count Trick

When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.

If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

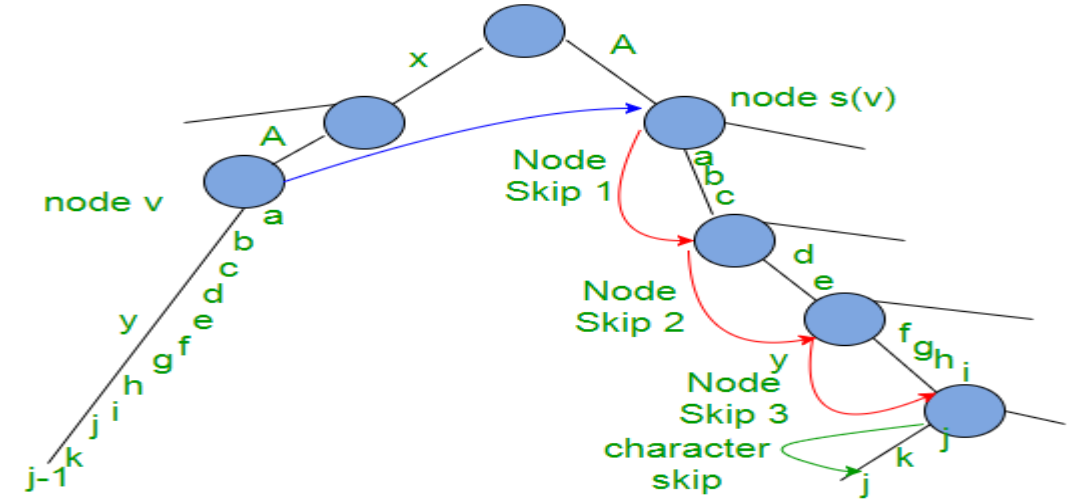


Figure 18 : skip/count trick : substring y from node v has length 11. Substring y from node s(v) is two characters down the last node, after 3 node skips

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes O(m).

Edge-label compression

So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. With this, suffix tree needs O(m) space.

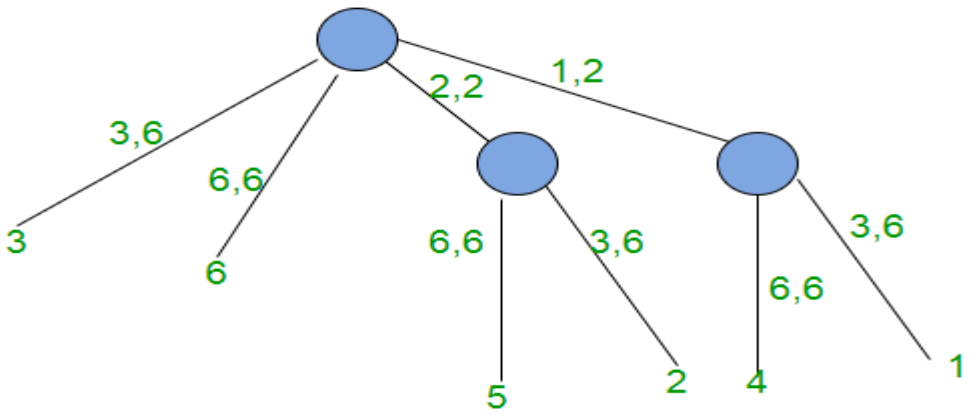


Figure 19 : Suffix tree for string xabxac with edge-label compression
Figure 14 shows same suffix tree without edge-label compression

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick "skip/count" is seen already while walk down).

Observation 1: Rule 3 is show stopper

In a phase i, there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase i+1 (i.e. path labelled S[j..i] continues with character S[i+1]), then it will also apply in all further extensions of same phase (i.e. extensions j+1 to i+1 in phase i+1). That's because if path labelled S[j..i] continues with character S[i+1], then path labelled S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also continue

with character S[i+1].
Consider Figure 11, Figure12 and Figure 13 in Part 1 where Rule 3 is applied.

In Figure 11, “xab” is added in tree and in Figure 12 (Phase 4), we add next character “x”. In this, 3 extensions are done (which adds 3 suffixes). Last suffix “x” is already present in tree.

In Figure 13, we add character “a” in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes “xa” and “a” are already present in tree. This shows that if suffix S[j..i] present in tree, then ALL the remaining suffixes S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also be there in tree and no work needed to add those remaining suffixes.
So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension j+1, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2
Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf
Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j, extension rule 1 will always apply to extension j in all successive phases.
Consider Figure 9 to Figure 14 in Part 1.
In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.
In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.
In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.
In any phase i, there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.
Also rule 2 creates a new leaf always (and internal node sometimes).
If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, …, J_i) , then $J_i <= J_{i+1}$
J_i will be equal to J_{i+1} when there are no new leaf created in phase i+1 (i.e rule 3 is applied in J_{i+1} extension)
In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$
In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$
In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$
J_i will be less than J_{i+1} when few new leaves are created in phase i+1.
In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$
So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i (which really doesn’t need much work, can be done in constant time and that’s the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.
Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i, end = i for leaf edges, for phase i+1, end = i+1 for leaf edges.

Trick 3
In any phase i, leaf edges may look like (p, i), (q, i), (r, i), …. where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r, i+1),…. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to

phase number. So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.
So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.
Tree Tm could be implicit tree if a suffix is prefix of another. So we can add a $ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next Part 3, we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints.

Here we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree.
While building suffix tree for string S of length m:

- There will be m phases 1 to m (one phase for each character)
  In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, ……, m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.
  (In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)
  Extension 1 will add suffix "a" in tree. We start from root and traverse path with label 'a'. There is no path from root, going out with label 'a', so create a leaf edge (Rule 2).

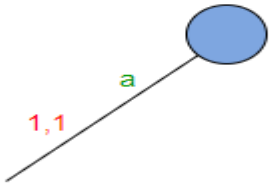


Figure 20 : Phase 1, extension 1-Rule2 applied. Created a leaf edge(1,1) Phase 1 completes here

- Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)
  For any string, Phase 1 will have only one extension and it will always follow Rule 2.
- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
  In our example, phase 2 will read second character 'b'. Suffixes to be added are "ab" and "b".
  Extension 1 adds suffix "ab" in tree.
  Path for label 'a' ends at leaf edge, so add 'b' at the end of this edge.

Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

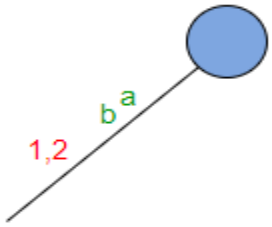


Figure 21 : Phase 2, Extension1-Rule1 applied extended the leaf edge from (1,1) to (1,2)

- Extension 2 adds suffix “b” in tree. There is no path from root, going out with label ‘b’, so creates a leaf edge (Rule 2).

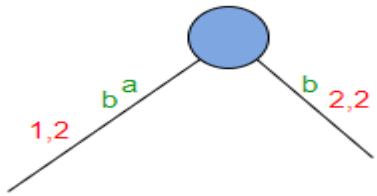


Figure 22 : Phase 2, Extension2-Rule2 applied
Created a leaf edge(2,2)
Phase 2 completes here

Phase 2 completes with the completion of extension 2.
Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
  In our example, phase 3 will read third character ‘c’. Suffixes to be added are “abc”, “bc” and “c”.
  Extension 1 adds suffix “abc” in tree.
  Path for label ‘ab’ ends at leaf edge, so add ‘c’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

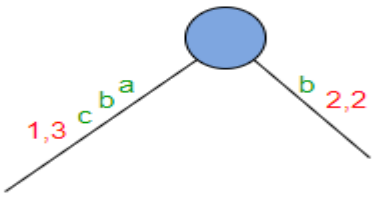


Figure 23 : Phase 3, Extension1-Rule1 applied
Extended the leaf edge from (1,2) to (1,3)

- Extension 2 adds suffix “bc” in tree.
  Path for label ‘b’ ends at leaf edge, so add ‘c’ at the end of this edge.
  Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

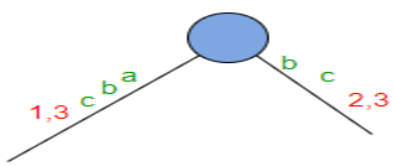


Figure 24 : Phase 3, Extension2-Rule1 applied
Extended the leaf edge from (2,2) to (2,3)

Extension 3 adds suffix “c” in tree. There is no path from root, going out with label ‘c’, so creates a leaf edge (Rule 2).

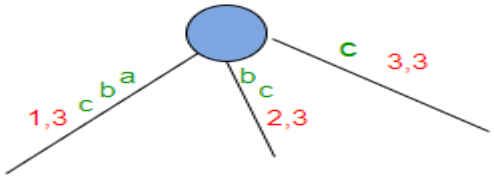


Figure 25 : Phase 3, Extension3-Rule2 applied
Created a leaf edge(3,3)
Phase 3 completes here

Phase 3 completes with the completion of extension 3.
Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
  In our example, phase 4 will read fourth character ‘a’. Suffixes to be added are “abca”, “bca”, “ca” and “a”.
  Extension 1 adds suffix “abca” in tree.
  Path for label ‘abc’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

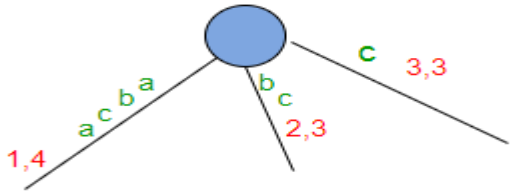


Figure 26 : Phase 4, Extension 1
Rule 1 applied

Extension 2 adds suffix “bca” in tree.
Path for label ‘bc’ ends at leaf edge, so add ‘a’ at the end of this edge.
Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

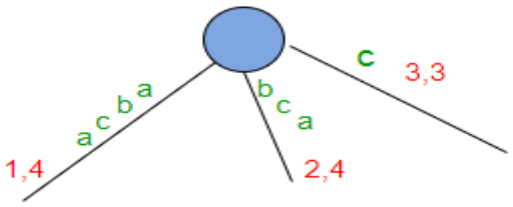


Figure 27 : Phase 4, Extension 2
Rule 1 applied

- Extension 3 adds suffix “ca” in tree.
  Path for label ‘c’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

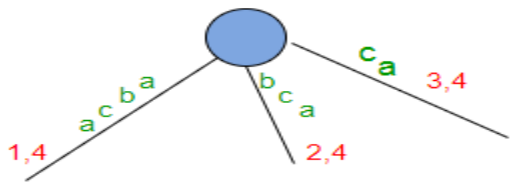


Figure 28 : Phase 4, Extension 3
Rule 1 applied

Extension 4 adds suffix “a” in tree.
Path for label ‘a’ exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix “a” is not seen explicitly (because it doesn’t end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.
Phase 4 completes as soon as Rule 3 is applied while Extension 4.
Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase i, there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
   e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).
2. After completing phase i, “end” indices of all leaf edges are i. How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the “end” index? Answer is “NO”.
   For this, we will maintain a global variable (say “END”) and we will just increment this global variable “END” and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i, then in phase i+1, first j extensions (1 to j) will be done by just incrementing variable “END” by 1 (END will be i+1 at the point).
   Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.
3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i, then in phase i+1, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are i+1-j extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.
   If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase i+1, trick 3 will take care of first i suffixes (the i leaf edges) and then extension i+1 will start from root node and it will insert just one character [$(i+1)^{th}$] suffix in tree by creating a leaf edge using Rule 2.
   If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are j-1 leaf edges so far.
   We will state few more facts (which may be a repeat, but we want to make sure it’s clear to you at this point) here based on discussion so far:
   - *Phase 1 starts with Rule 2, all other phases start with Rule 1*
   - *Any phase ends with either Rule 2 or Rule 3*
   - *Any phase i may go through a series of j extensions (1 <= j <= i). In these j extensions, first p (0 <= p < i) extensions will follow Rule 1, next q (0 <= q <= i-p) extensions will follow Rule 2 and next r (0<= r <= 1) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
   - *In a phase i, p + q + r <= i*
   - *At the end of any phase i, there will be p+q leaf edges and next phase i+1 will go through Rule 1 for first p+q extensions*

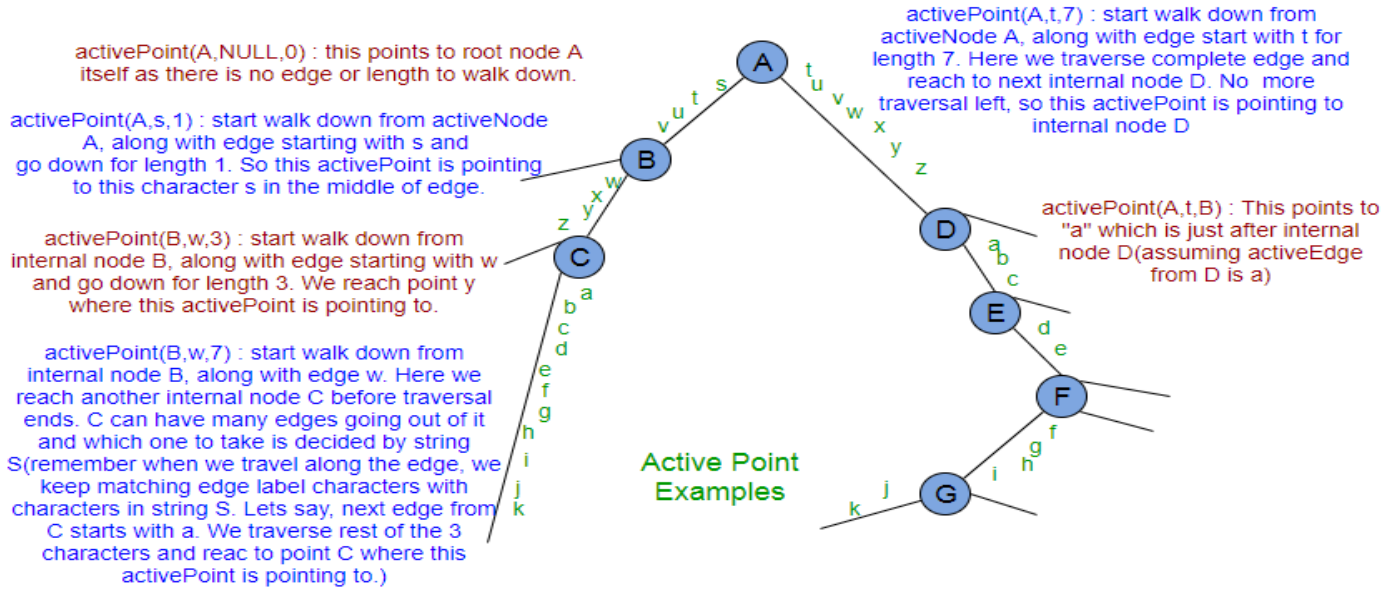
In the next phase i+1, trick 3 (Rule 1) will take care of first j-1 suffixes (the j-1 leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall

algorithm will not be linear. activePoint comes to the rescue here.

In previous phase i, while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase i+1 will start exactly from the same point and we start matching path against $(i+1)^{th}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1^{st} p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as apprppriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).
To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.

activeNode: This could be root node or an internal node.
activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.
activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.



After phase i, if there are j leaf edges then in phase i+1, first j extensions will be done by trick 3. activePoint will be needed for the extensions from j+1 to i+1 and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i, then before we move on to next phase i+1, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLenth is increased by 1 (because of

matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase i+1.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is (A, s, 11) in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLenght as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:
(A, s, 11) — >>> (B, w, 7) —- >>> (C, a, 3)
All above three activePoints refer to same point 'c'
Let's take another example.
If activePoint is (D, a, 11) at the start of an extension, then while walk down, below is the sequence of changes in activePoint:
(D, a, 10) — >>> (E, d, 7) — >>> (F, f, 5) — >> (G, j, 1)
All above activePoints refer to same point 'k'.
If activePoints are (A, s, 3), (A, t, 5), (B, w, 1), (D, a, 2) etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.
The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase i+1, we don't really have to perform all i+1 extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

String Matching Introduction

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."
Given a text array, T [1.....n], of n character and a pattern array, P [1.....m], of m characters. The problems are to find an integer s, called **valid shift** where $0 \le s < n-m$ and T [s+1.....s+m] = P [1.....m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.Given a string T [1.....n], the **substrings** are represented as T [i.....j] for some $0 \le i \le j \le n-1$, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).
The **proper substring** of string T [1.....n] is T [1.....j] for some $0 < i \le j \le n-1$. That is, we must have either i>0 or j < m-1.

Using these descriptions, we can say given any string T [1.....n], the substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some $0 \le i \le j \le n-1$.

And proper substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some $0 \le i \le j \le n-1$.

Note: If i>j, then T [i.....j] is equal to the empty string or null, which has length zero.

Algorithms used for String Matching:
There are different types of method is used to finding the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

Naive algorithm for Pattern Searching
The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].
The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.....m] = T [s+1.....s+m] for each of the n - m +1 possible value of s.

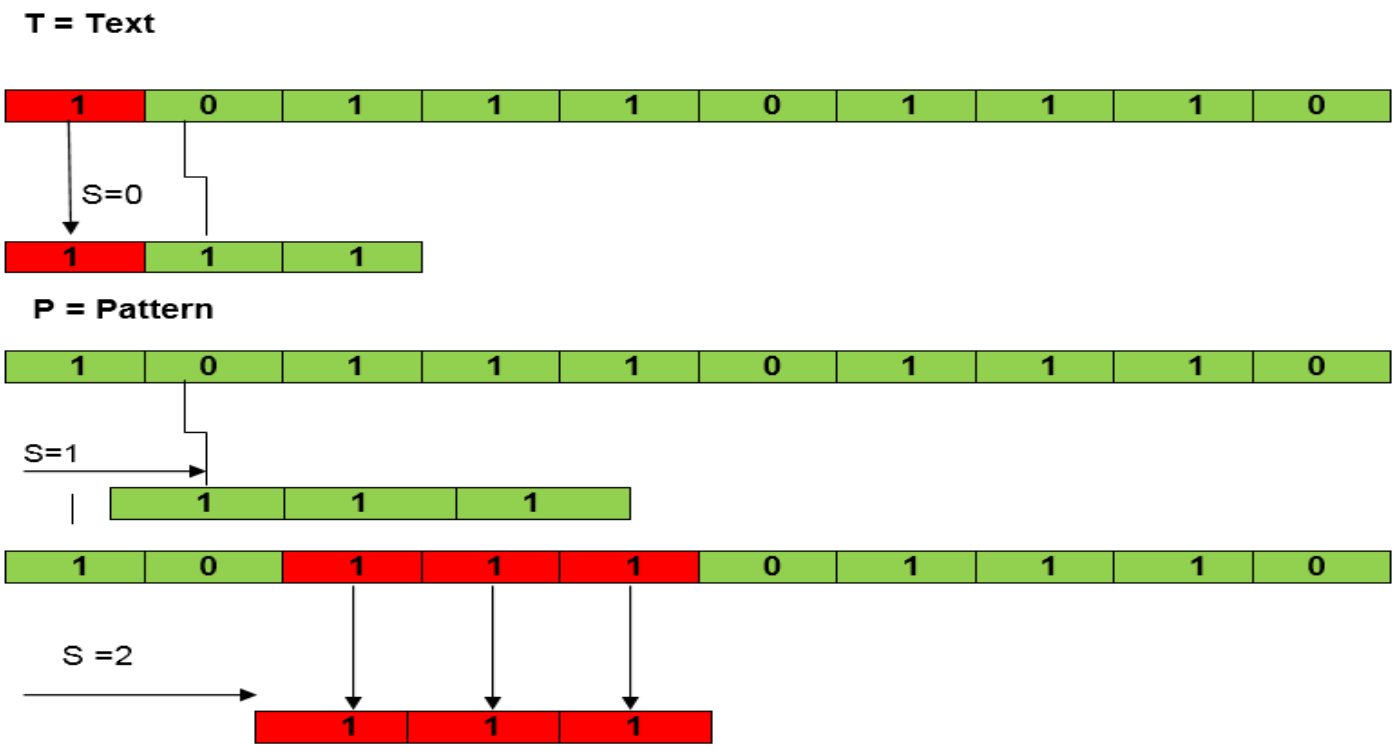
NAIVE-STRING-MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

Analysis: This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

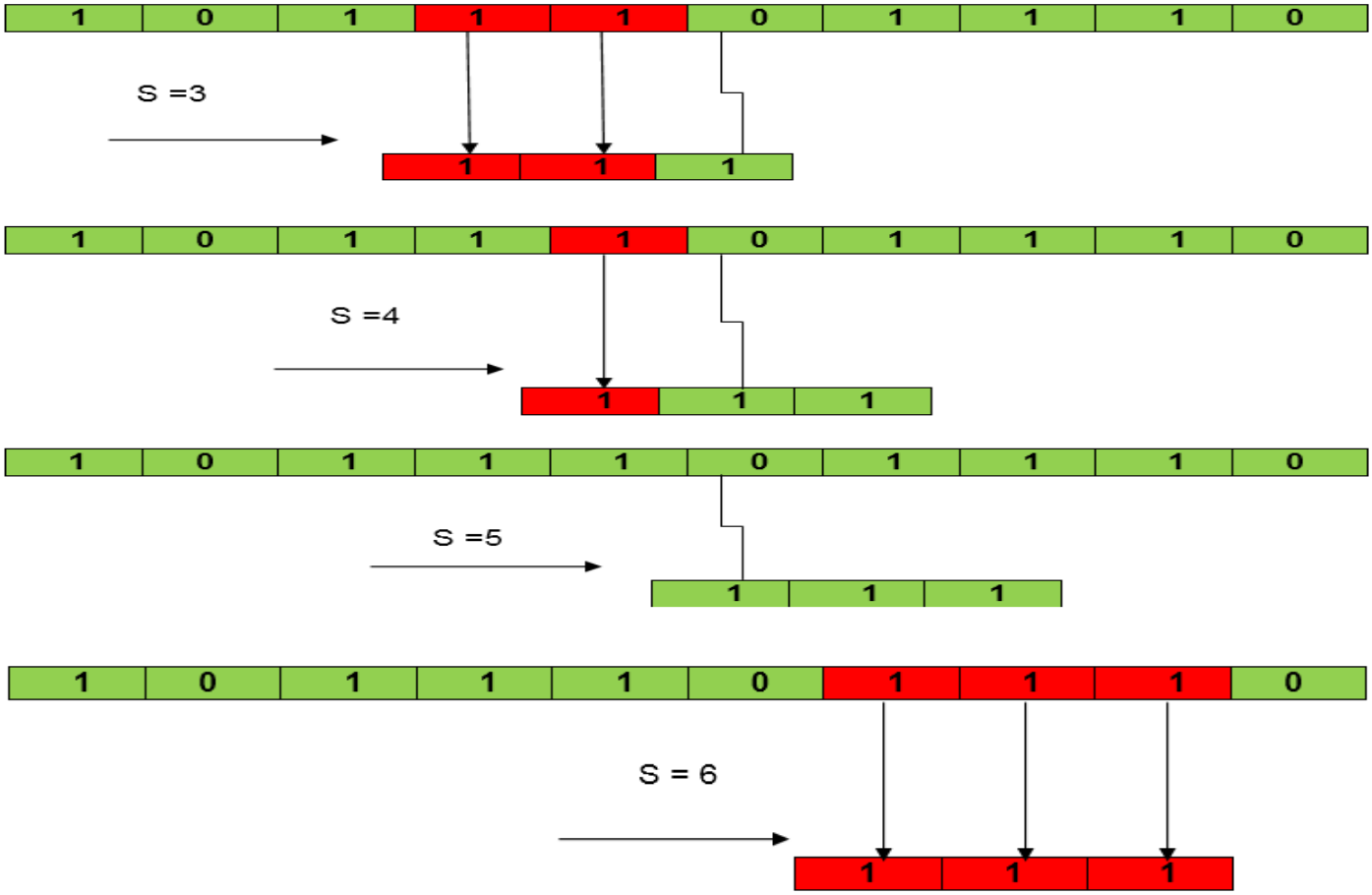
Example:

1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift

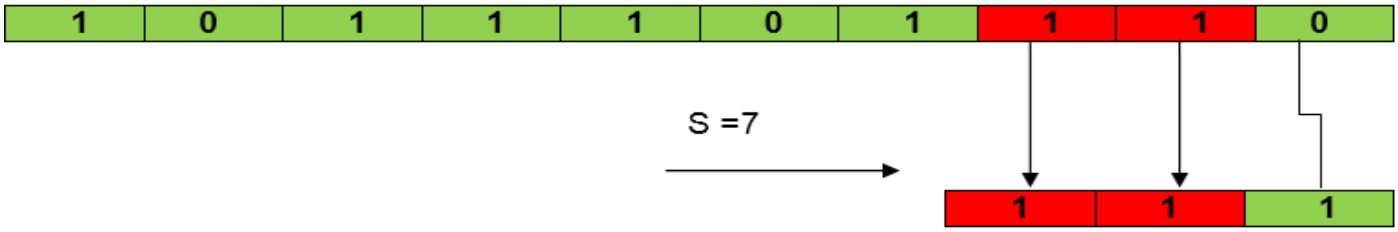
Solution:



So, S=2 is a Valid Shift



So, S=6 is a Valid Shift




```
# Python3 program for Naive Pattern
# Searching algorithm
def search(pat, txt):
        M = len(pat)
        N = len(txt)
        # A loop to slide pat[] one by one */
        for i in range(N - M + 1):
                j = 0

                # For current index i, check
                # for pattern match */
                while(j < M):
                        if (txt[i + j] != pat[j]):
                                break
                        j += 1

                if (j == M):
```

```
                print("Pattern found at index ", i)
# Driver Code
if __name__ == '__main__':
        txt = "AABAACAADAABAAABAA"
        pat = "AABA"
        search(pat, txt)
```

Output:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

The Rabin-Karp-Algorithm
The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)
1. n ← length [T]
2. m ← length [P]
3. h ← d^{m-1} mod q
4. p ← 0
5. t_0 ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. t_0 ← (dt_0+T [i]) mod q
9. for s ← 0 to n-m
10. do if p = t_s
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then t_{s+1} ← (d (t_s-T [s+1]h)+T [s+m+1])mod q

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....

1. T = 31415926535.....
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

Solution:

31 mod 11 = 9 not equal to 4

14 mod 11 = 3 not equal to 4

41 mod 11 = 8 not equal to 4

15 mod 11 = 4 equal to 4 SPURIOUS HIT

59 mod 11 = 4 equal to 4 SPURIOUS HIT

92 mod 11 = 4 equal to 4 SPURIOUS HIT

26 mod 11 = 4 EXACT MATCH

65 mod 11 = 10 not equal to 4

53 mod 11 = 9 not equal to 4

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

Complexity:
The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a

Solution:

Initially: m = length [p] = 7

Π [1] = 0

k = 0

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0                    // numbers of characters matched
5. for i ← 1 to n           // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]             // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1                   // next character matches
10. If q = m                        // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                       // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Example: Given a string 'T' and pattern 'P' as follows:

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

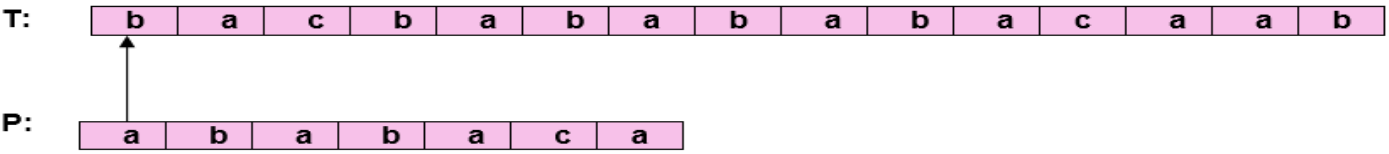
Solution:

Initially: n = size of T = 15

m = size of P = 7

Step1: i=1, q=0

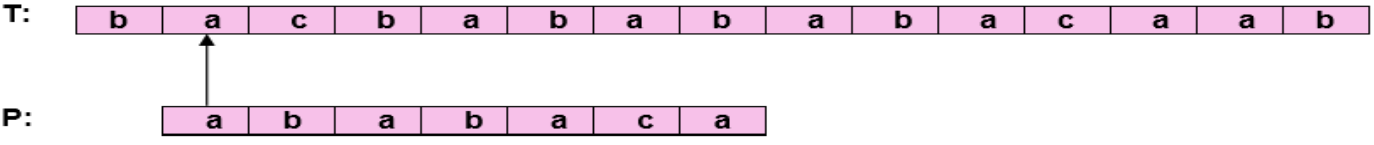
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

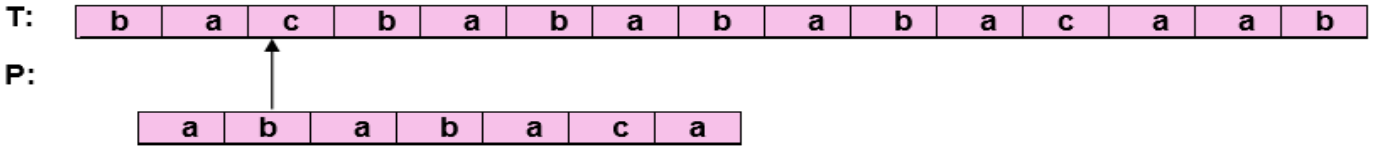
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

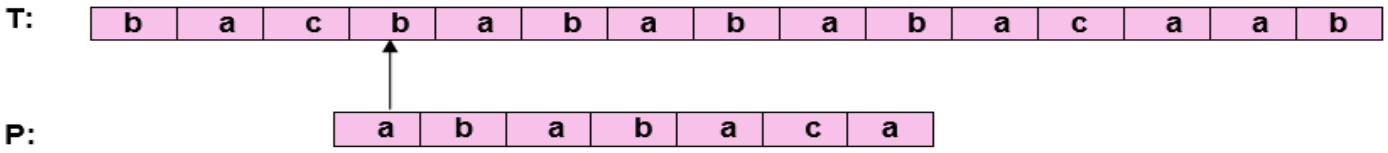
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

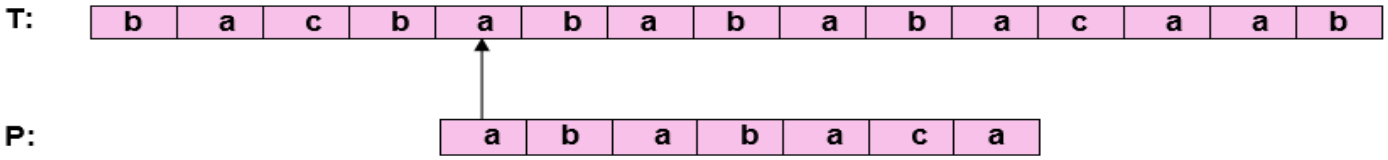
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



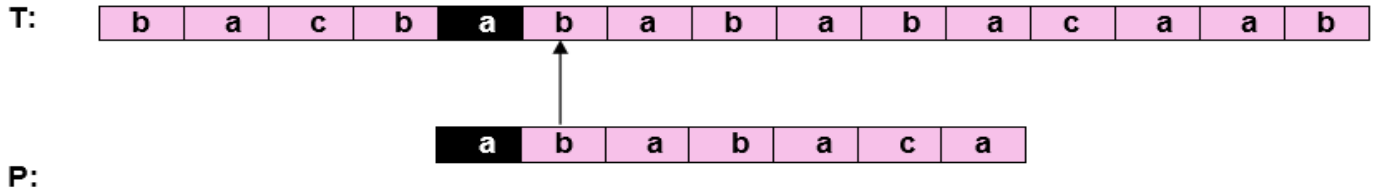
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



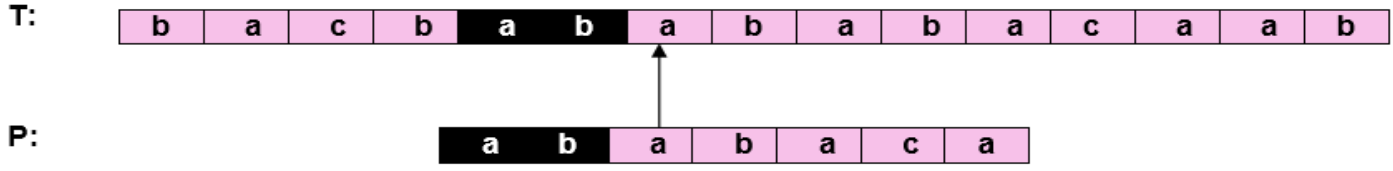
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



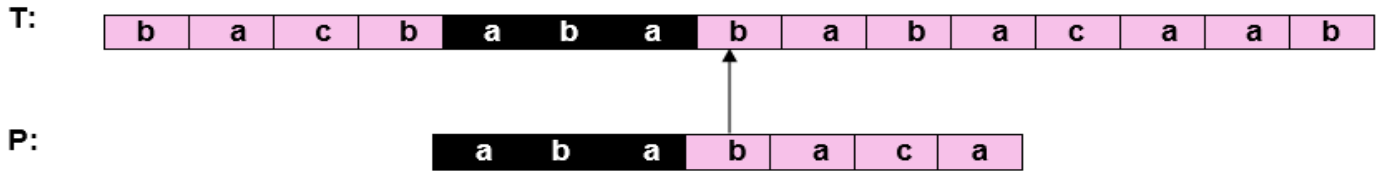
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



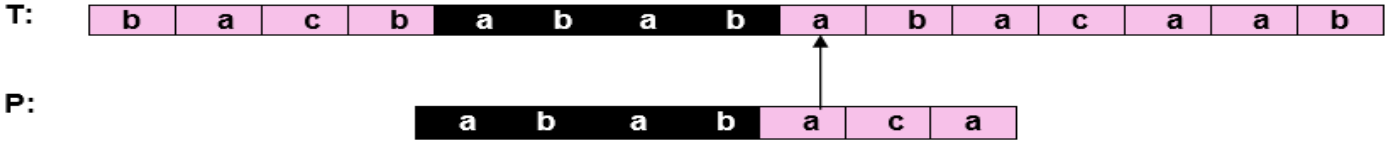
Step8: i = 8, q =3

Comparing P [4] with T [8] P [4] matches with T [8]



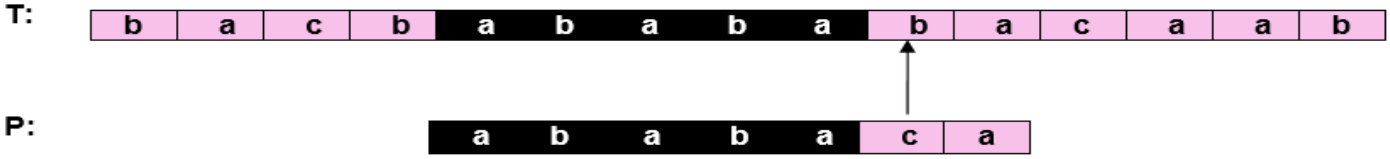
Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]



Step10: i = 10, q = 5

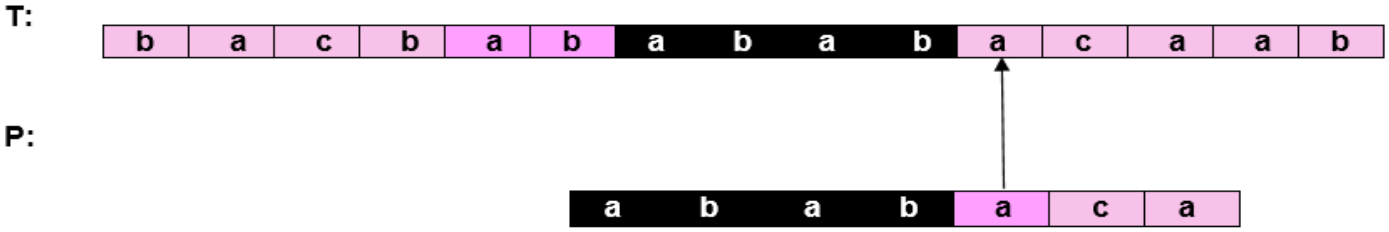
Comparing P [6] with T [10] P [6] doesn't match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

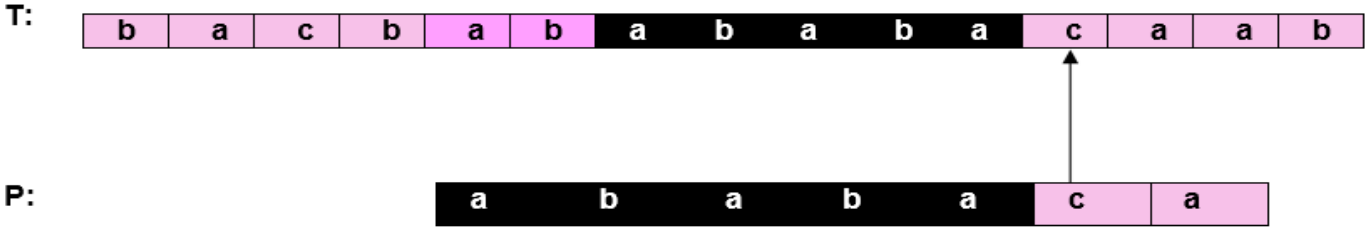
Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]



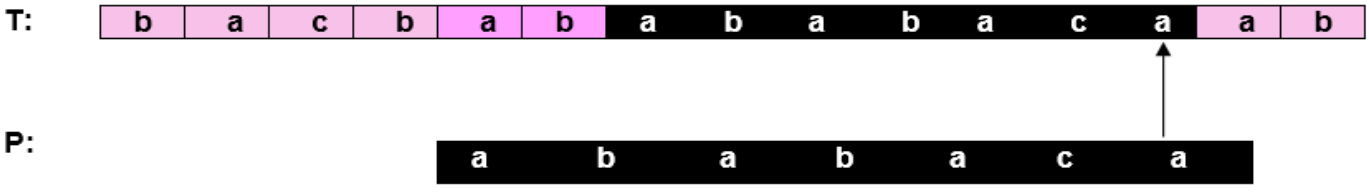
Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Mod-4

Definition of NP class Problem: - The set of all decision-based problems came into the division of NP Problems who can't be solved or produced an output within polynomial time but verified in the **polynomial time**. NP class contains P class as a subset. NP problems being hard to solve.

Definition of P class Problem: - The set of decision-based problems come into the division of P Problems who can be solved or produced an output within polynomial time. P problems being easy to solve

Definition of Polynomial time: - If we produce an output according to the given input within a specific amount of time such as within a minute, hours. This is known as Polynomial time.

Definition of Non-Polynomial time: - If we produce an output according to the given input but there are no time constraints is known as Non-Polynomial time. But yes output will produce but time is not fixed yet.

Definition of Decision Based Problem: - A problem is called a decision problem if its output is a simple "yes" or "no" (or you may need this of this as true/false, 0/1, accept/reject.) We will phrase many optimization problems as decision problems. For example, Greedy method, D.P., given a graph G= (V, E) if there exists any Hamiltonian cycle.

Definition of NP-hard class: - Here you to satisfy the following points to come into the division of NP-hard

1. If we can solve this problem in polynomial time, then we can solve all NP problems in polynomial time
2. If you convert the issue into one form to another form within the polynomial time

Definition of NP-complete class: - A problem is in NP-complete, if

1. It is in NP
2. It is NP-hard

Pictorial representation of all NP classes which includes NP, NP-hard, and NP-complete

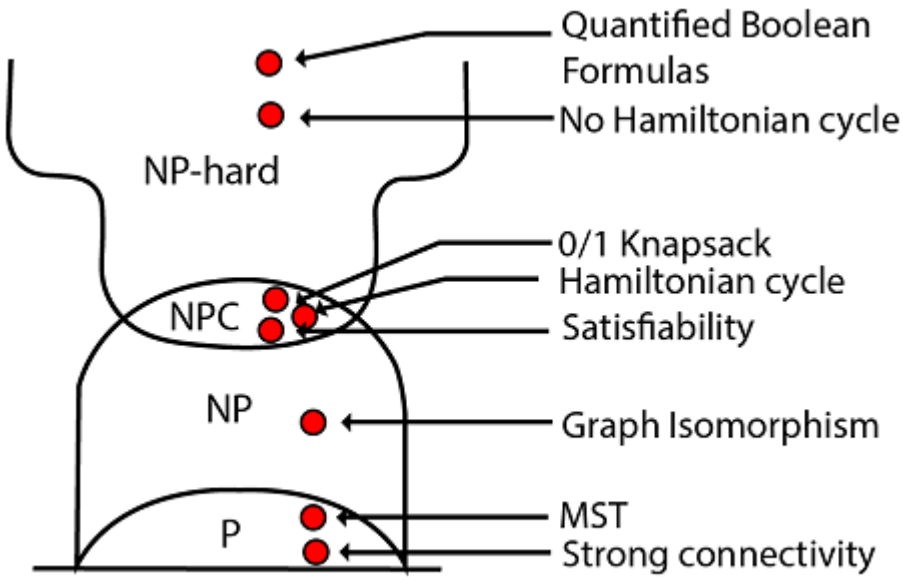


Fig: Complexity Classes

Polynomial Time Verification

Before talking about the class of NP-complete problems, it is essential to introduce the notion of a verification algorithm.

Many problems are hard to solve, but they have the property that it easy to authenticate the solution if one is provided.

Hamiltonian cycle problem:-

Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.

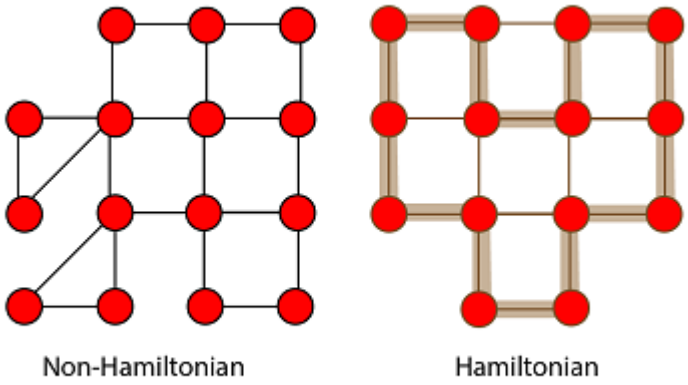


Fig: Hamiltonian Cycle

Let us understand that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would similarly say: "the period is hv3, v7, v1....v13i.

We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle.

Definition of Certificate: - A piece of information which contains in the given path of a vertex is known as certificate
Relation of P and NP classes

1. P contains in NP
2. P=NP

1. Observe that P contains in NP. In other words, if we can solve a problem in polynomial time, we can indeed verify the solution in polynomial time. More formally, we do not need to see a certificate (there is no need to specify the vertex/intermediate of the specific path) to solve the problem; we can explain it in polynomial time anyway.
2. However, it is not known whether P = NP. It seems you can verify and produce an output of the set of decision-based problems in NP classes in a polynomial time which is impossible because according to the definition of NP classes you can verify the solution within the polynomial time. So this relation can never be held.

Reductions:

The class NP-complete (NPC) problems consist of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently. But if there were a polynomial solution for even a single NP-complete problem, then every problem in NPC will be solvable in polynomial time. For this, we need the concept of reductions.
Suppose there are two problems, A and B. You know that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be explained in polynomial time. We want to show that ($A \notin P$) => ($B \notin P$)
Consider an example to illustrate reduction: The following problem is well-known to be NPC:

3-color: Given a graph G, can each of its vertices be labeled with one of 3 different colors such that two adjacent vertices do not have the same label (color).
Coloring arises in various partitioning issues where there is a constraint that two objects cannot be assigned to the same set of partitions. The phrase "coloring" comes from the original application which was in map drawing. Two countries that contribute a common border should be colored with different colors.
It is well known that planar graphs can be colored (maps) with four colors. There exists a polynomial time algorithm for this. But deciding whether this can be done with 3 colors is hard, and there is no polynomial time algorithm for it.

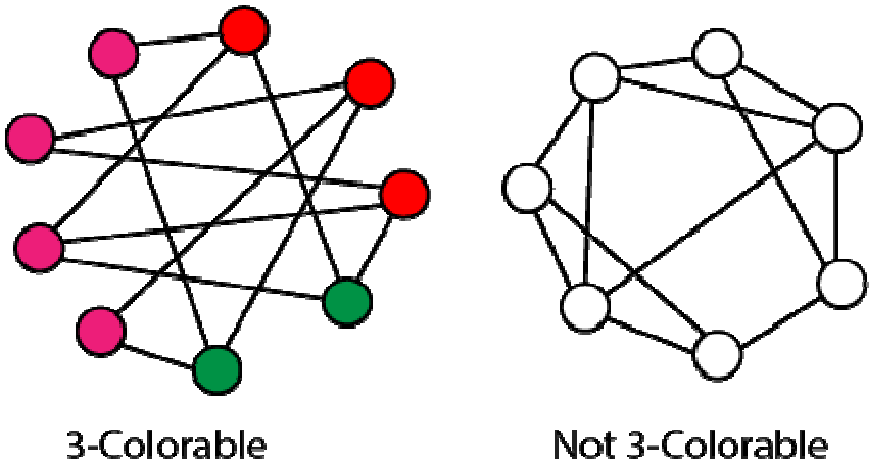


Fig: Example of 3-colorable and non-3-colorable graphs.

Polynomial Time Reduction:

We say that Decision Problem L_1 is Polynomial time Reducible to decision Problem L_2 ($L_{1 \leq p} L_2$) if there is a polynomial time computation function f such that of all x, $x \epsilon L_1$ if and only if $x \epsilon L_2$.

NP-Completeness

A decision problem L is NP-Hard if

L' ≤p L for all L' ϵ NP.

Definition: L is NP-complete if

1. L ϵ NP and
2. L' ≤ p L for some known NP-complete problem L.' Given this formal definition, the complexity classes are:

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all L' ϵ NP, L' ≤p L. Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

1. L ϵ NP and
2. L is NP-hard

If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time.

Reductions

Concept: - If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time. For this, you need the concept of reduction. If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe). For this, you need the concept of reduction.

Example: - Suppose there are two problems, **A** and **B**. You know that it is impossible to solve problem **A** in polynomial time. You want to prove that B cannot be solved in polynomial time. So you can convert the problem **A** into problem **B** in polynomial time.

Example of NP-Complete problem

NP problem: - Suppose a DECISION-BASED problem is provided in which a set of inputs/high inputs you can get high output.

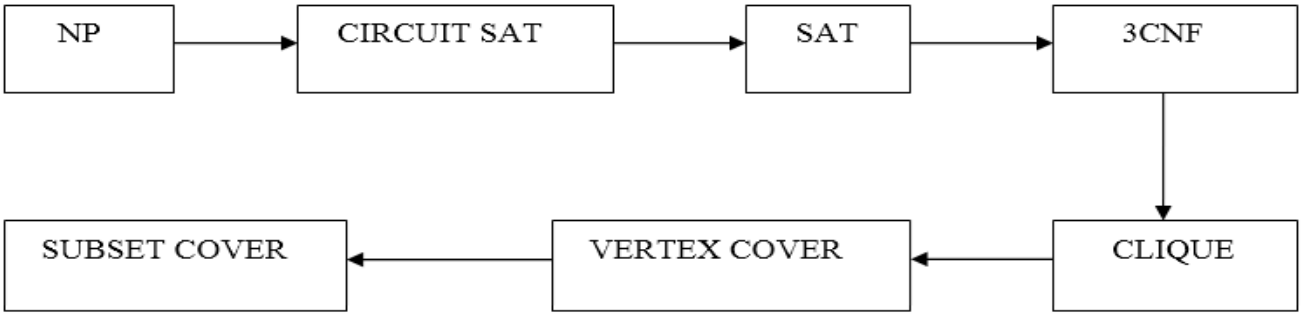
Criteria to come either in NP-hard or NP-complete.

1. The point to be noted here, the output is already given, and you can verify the output/solution within the polynomial time but can't produce an output/solution in polynomial time.

2. Here we need the concept of reduction because when you can't produce an output of the problem according to the given input then in case you have to use an emphasis on the concept of reduction in which you can convert one problem into another problem.

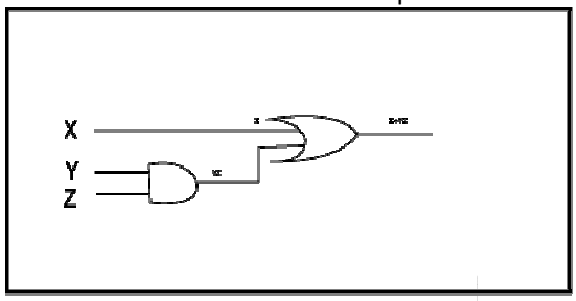
So according to the given decision-based NP problem, you can decide in the form of yes or no. If, yes then you have to do verify and convert into another problem via reduction concept. If you are being performed, both then decision-based NP problems are in NP compete.

Here we will emphasize NPC.



CIRCUIT SAT

According to given decision-based NP problem, you can design the CIRCUIT and verify a given mentioned output also within the P time. The CIRCUIT is provided below:-



SAT (Satisfiability):-

A Boolean function is said to be SAT if the output for the given value of the input is true/high/1

F=X+YZ (Created a Boolean function by CIRCUIT SAT)

These points you have to be performed for NPC

1. CONCEPTS OF SAT
2. CIRCUIT SAT≤ρ SAT
3. SAT≤ρ CIRCUIT SAT
4. SAT ϵ NPC

1. **CONCEPT:** - A Boolean function is said to be SAT if the output for the given value of the input is true/high/1.
2. **CIRCUIT SAT≤ρ SAT:** - In this conversion, you have to convert CIRCUIT SAT into SAT within the polynomial time as we did it
3. **SAT≤ρ CIRCUIT SAT:** - For the sake of verification of an output you have to convert SAT into CIRCUIT SAT within the polynomial time, and through the CIRCUIT SAT you can get the verification of an output successfully
4. **SAT ϵ NPC:** - As you know very well, you can get the SAT through CIRCUIT SAT that comes from NP.

Proof of NPC: - Reduction has been successfully made within the polynomial time from CIRCUIT SAT TO SAT. Output has also been verified within the polynomial time as you did in the above conversation.
So concluded that SAT ϵ NPC.

3CNF SAT

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)
You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)
V=OR operator
^ =AND operator
These all the following points need to be considered in 3CNF SAT.

To prove: -

1. Concept of 3CNF SAT
2. SAT≤ρ 3CNF SAT
3. 3CNF≤ρ SAT
4. 3CNF ϵ NPC

1. **CONCEPT:** - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
2. **SAT ≤ρ 3CNF SAT:**- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

   F=X+YZ
   = (X+Y) (X+Z)
   = (X+Y+ZZ') (X+YY'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
3. **3CNF ≤ρ SAT:** - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

   F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
   = (X+Y+ZZ') (X+YY'+Z)
   = (X+Y) (X+Z)
   = X+YZ
4. **3CNF ϵ NPC:** - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

Proof of NPC:-

1. It shows that you can easily convert a Boolean function of SAT into 3CNF SAT and satisfied the concept of 3CNF SAT also within polynomial time through Reduction concept.
2. If you want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

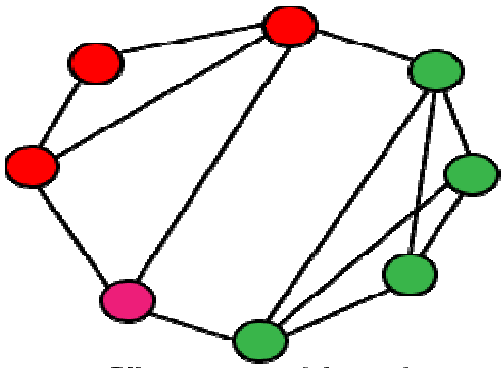
If you can achieve these two points that means 3CNF SAT also in NPC

Clique
To Prove: - Clique is an NPC or not?
For this you have to satisfy the following below-mentioned points: -

1. Clique
2. 3CNF ≤ρ Clique
3. Clique ≤ρ 3CNF≤SAT
4. Clique ϵ NP

1) Clique
Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

CLIQUE COVER: - Given a graph G and an integer k, can we find k subsets of verticesV_1, V_2...V_K, such that UiVi = V, and that each Vi is a clique of G.
The following figure shows a graph that has a clique cover of size 3.



Clique cover (size=3)

2)3CNF ≤ρ Clique

Proof:-For the successful conversion from 3CNF to Clique, you have to follow the two steps:-

Draw the clause in the form of vertices, and each vertex represents the literals of the clauses.

1. They do not complement each other
2. They don't belong to the same clause
   In the conversion, the size of the Clique and size of 3CNF must be the same, and you successfully converted 3CNF into Clique within the polynomial time

Clique ≤ρ 3CNF

Proof: - As you know that a function of K clause, there must exist a Clique of size k. It means that P variables which are from the different clauses can assign the same value (say it is 1). By using these values of all the variables of the CLIQUES, you can make the value of each clause in the function is equal to 1

Example: - You have a Boolean function in 3CNF:-

(X+Y+Z) (X+Y+Z') (X+Y'+Z)

After Reduction/Conversion from 3CNF to CLIQUE, you will get P variables such as: - x +y=1, x +z=1 and x=1

Put the value of P variables in equation (i)

(1+1+0)(1+0+0)(1+0+1)

(1)(1)(1)=1 output verified

4) Clique ϵ NP:-

Proof: - As you know very well, you can get the Clique through 3CNF and to convert the decision-based NP problem into 3CNF you have to first convert into SAT and SAT comes from NP.

So, concluded that CLIQUE belongs to NP.

Proof of NPC:-

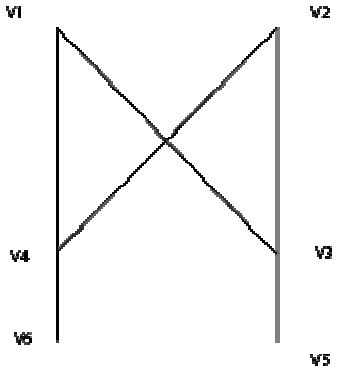
1. Reduction achieved within the polynomial time from 3CNF to Clique
2. And verified the output after Reduction from Clique To 3CNF above
   So, concluded that, if both Reduction and verification can be done within the polynomial time that means **Clique also in NPC**.

Vertex Cover

1. Vertex Cover Definition
2. Vertex Cover ≤ρ Clique
3. Clique ≤ρ Vertex Cover
4. Vertex Cover ϵ NP

1) Vertex Cover:

Definition: - It represents a set of vertex or node in a graph G (V, E), which gives the connectivity of a complete graph According to the graph G of vertex cover which you have created, **the size of Vertex Cover =2**



2) Vertex Cover ≤ρ Clique

In a graph G of Vertex Cover, you have N vertices which contain a Vertex Cover K. There must exist of Clique Size of size N-K in its complement.

According to the graph G, you have

Number of vertices=6

Size of Clique=N-K=4

You can also create the Clique by complimenting the graph G of Vertex Cover means in simpler form connect the vertices in Vertex Cover graph G through edges where edges don?t exist and remove all the existed edges

You will get the graph G with Clique Size=4

3) Clique ≤ρ Vertex Cover

Here through the Reduction process, you can get the Vertex Cover form Clique by just complimenting the Clique graph G within the polynomial time.

4) Vertex Cover ϵ NP

As you know very well, you can get the Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into 3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.

Proof of NPC:-

1. Reduction from Clique to Vertex Cover has been made within the polynomial time. In the simpler form, you can convert into Vertex Cover from Clique within the polynomial time
2. And verification has also been done when you convert Vertex Cover to Clique and Clique to 3CNF and satisfy/verified the output within a polynomial time also, so it concluded that Reduction and Verification had been done in the polynomial time that means **Vertex Cover also comes in NPC**

Subset Cover

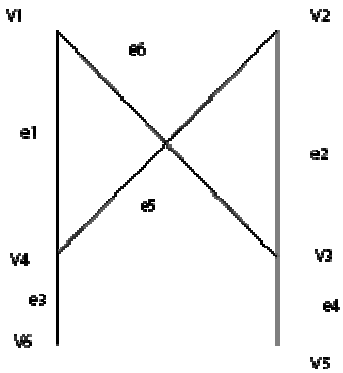
To Prove:-

1. Subset Cover
2. Vertex Cover ≤ρ Subset Cover
3. Subset Cover≤ρ Vertex Cover
4. Subset Cover ϵ NP

1) Subset Cover

Definition: - Number of a subset of edges after making the union for a get all the edges of the complete graph G, and that is called Subset Cover.

According to the graph G, which you have created the size of Subset Cover=2



1. v1{e1,e6} v2{e5,e2} v3{e2,e4,e6} v4{e1,e3,e5} v5{e4} v6{e3}
2. v3Uv4= {e1, e2, e3, e4, e5, e6} complete set of edges after the union of vertices.

2) Vertex Cover ≤ρ Subset Cover
In a graph G of vertices N, if there exists a Vertex Cover of size k, then there must also exist a Subset Cover of size k even. If you can achieve after the Reduction from Vertex Cover to Subset Cover within a polynomial time, which means you did right.

3) Subset Cover ≤ρ Vertex Cover
Just for verification of the output perform the Reduction and create Clique and via an equation, N-K verifies the Clique also and through Clique you can quickly generate 3CNF and after solving the Boolean function of 3CNF in the polynomial time. You will get output. It means the output has been verified.

4) Subset Cover ϵ NP:-
Proof: - As you know very well, you can get the Subset-Cover through Vertex Cover and Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.
Proof of NPC:-
The Reduction has been successfully made within the polynomial time form Vertex Cover to Subset Cover
Output has also been verified within the polynomial time as you did in the above conversation so, concluded that **SUBSET COVER also comes in NPC**.

Independent Set:
An independent set of a graph G = (V, E) is a subset V'⊆V of vertices such that every edge in E is incident on at most one vertex in V.' The independent-set problem is to find a largest-size independent set in G. It is not hard to find small independent sets, e.g., a small independent set is an individual node, but it is hard to find large independent sets.

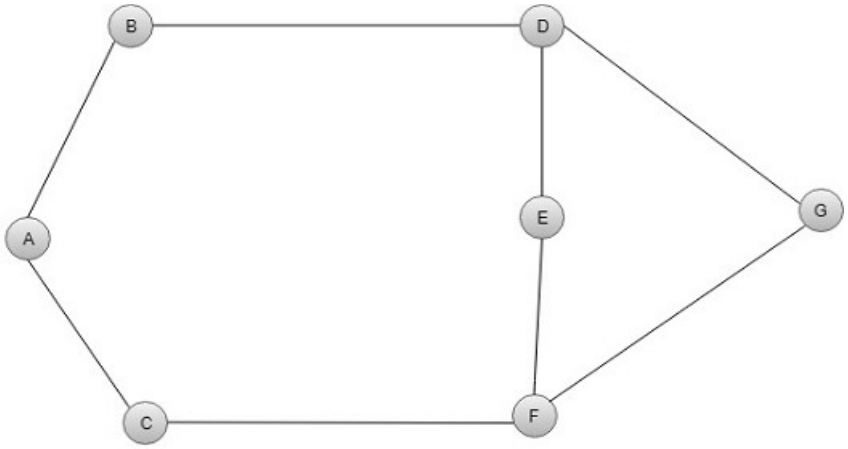


Fig:A graph with large independent sets of size 4 and smallest vertex cover of size 3.