



PIE Tech

POLLACHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by **AICTE** and Affiliated to **Anna University**)

sky is the limit

Department of Computer Science and Engineering

Regulation 2021

II Year – IV Semester

CS3401- Algorithm

COURSE OBJECTIVES:

- ☐ To understand and apply the algorithm analysis techniques on searching and sorting algorithms
- ☐ To critically analyze the efficiency of graph algorithms ☐ To understand different algorithm design techniques
- ☐ To solve programming problems using state space tree
- ☐ To understand the concepts behind NP Completeness, Approximation algorithms and randomized algorithms.

UNIT I INTRODUCTION 9

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – searching: linear search, binary search and Interpolation Search, Pattern search: The naïve string matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. Sorting: Insertion sort – heap sort

UNIT II GRAPH ALGORITHMS 9

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method – Matching: Maximum bipartite matching

UNIT III ALGORITHM DESIGN TECHNIQUES 9

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. Greedy Technique: Elements of the greedy strategy - Activity-selection problem — Optimal Merge pattern — Huffman Trees.

UNIT IV STATE SPACE SEARCH ALGORITHMS 9

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem – Graph colouring problem Branch and Bound: Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

UNIT V NP-COMPLETE AND APPROXIMATION ALGORITHM 9

Tractable and intractable problems: Polynomial time algorithms – Venn diagram representation - NP algorithms - NP-hardness and NP-completeness – Bin Packing problem - Problem reduction: TSP – 3CNF problem. Approximation Algorithms: TSP - Randomized Algorithms: concept and application - primality testing - randomized quick sort - Finding kth smallest number

UNIT I INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – searching: linear search, binary search and Interpolation Search, Pattern search: The naïve string matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. Sorting: Insertion sort – heap sort

Introduction

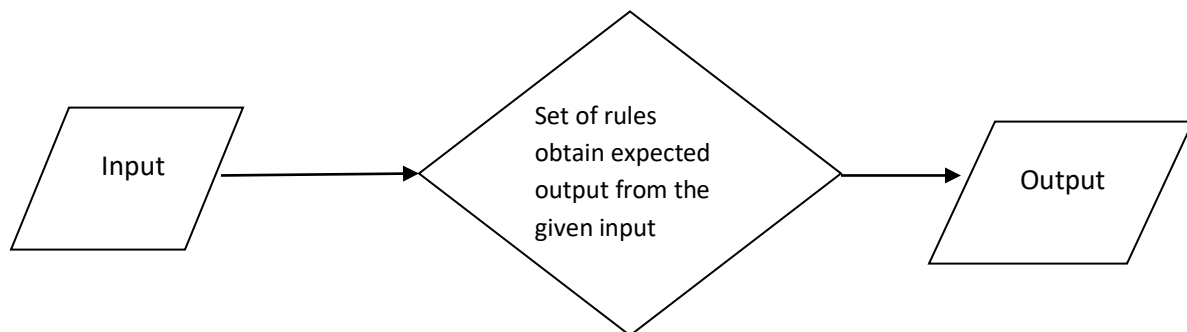
Definition of Algorithm

The word Algorithm means ” **A set of finite rules or instructions to be followed in calculations or other problem-solving operations** ”

Or

” **A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations**”.

What is Algorithm?



Algorithm Analysis

Algorithm analysis is the process of determining the time and space complexity of an algorithm, which are measures of the algorithm's efficiency.

Time complexity refers to the amount of time it takes for an algorithm to run as a function of the size of the input, and is typically expressed using big O notation. Space complexity refers to the amount of memory required by an algorithm as a function of the size of the input, and is also typically expressed using big O notation.

Time and Space Complexity

Time complexity is a measure of how long an algorithm takes to run as a function the size of the input. It is typically expressed using big O notation, which describes the upper bound on the growth of the time required by the algorithm. For example, an algorithm with a time complexity of $O(n)$ takes longer to run as the input size (n) increases.

There are different types of time complexities:

- $O(1)$ or constant time: the algorithm takes the same amount of time to run regardless of the size of the input.
- $O(\log n)$ or logarithmic time: the algorithm's running time increases logarithmically with the size of the input.
- $O(n)$ or linear time: the algorithm's running time increases linearly with the size of the input.
- $O(n \log n)$ or linear logarithmic time: the algorithm's running time increases linearly with the size of the input and logarithmically with the size of the input.
- $O(n^2)$ or quadratic time: the algorithm's running time increases quadratically with the size of the input.

Space complexity, on the other hand, is a measure of how much memory an algorithm uses as a function of the size of the input. Like time complexity, it is typically expressed using big O notation. For example, an algorithm with a space complexity of $O(n)$ uses more memory as the input size (n) increases. Space complexities are generally categorized as:

- $O(1)$ or constant space: the algorithm uses the same amount of memory regardless of the size of the input.
- $O(n)$ or linear space: the algorithm's memory usage increases linearly with the size of the input.
- $O(n^2)$ or quadratic space: the algorithm's memory usage increases quadratically with the size of the input.
- $O(2^n)$ or exponential space: the algorithm's memory usage increases exponentially with the size of the input.

Asymptotic notation and its properties

Asymptotic notation is a mathematical notation used to describe the behavior of an algorithm as the size of the input (usually denoted by n) becomes arbitrarily large.

There are mainly three asymptotic notations:

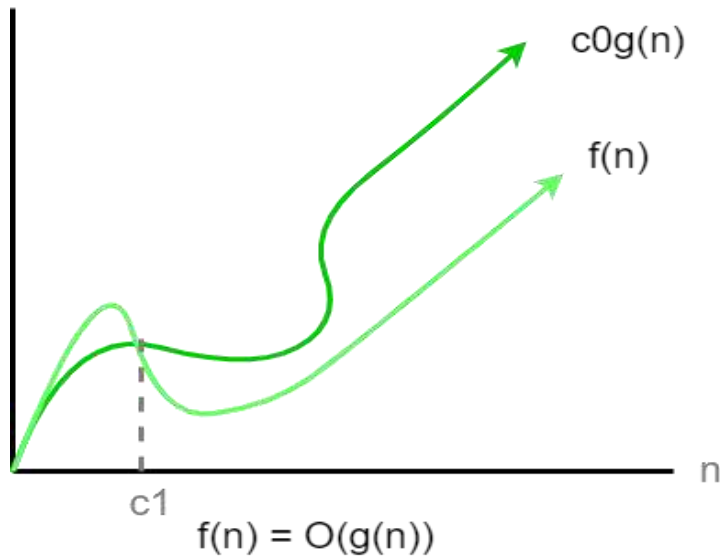
1. **Big-O Notation (O -notation)**
2. **Omega Notation (Ω -notation)**
3. **Theta Notation (Θ -notation)**

1. Big-O Notation

Big O notation is a mathematical concept used in computer science to describe the upper bound of an algorithm's time or space complexity. It provides a way to express the worst-case scenario of how an algorithm performs as the size of the input increases.

Mathematical Representation of Big O Notation

A function $f(n)$ is said to be $O(g(n))$ if there exist positive constants c_0 and c_1 such that $0 \leq f(n) \leq c_0 \cdot g(n)$ for all $n \geq c_1$. This means that for sufficiently large values of n , the function $f(n)$ does not grow faster than $g(n)$ up to a constant factor.



$O(g(n)) = \{f(n): \text{there exist positive constants } c_0 \text{ and } c_1 \text{ such that } 0 \leq f(n) \leq c_0g(n) \text{ for all } n \geq c_1\}.$

For Example:

Let, $f(n) = n^2 + n + 1$

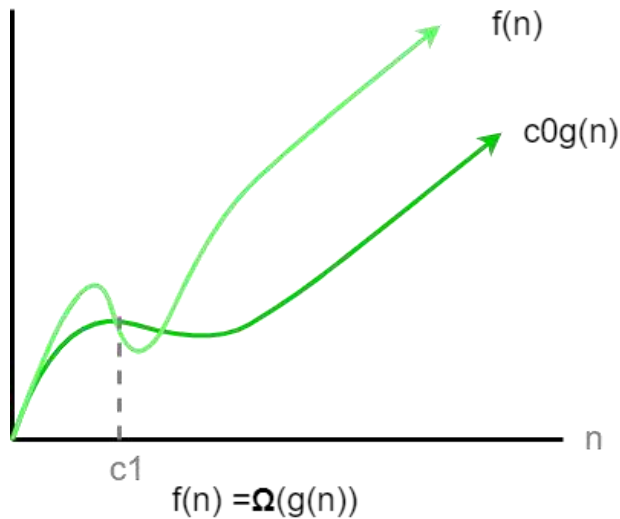
$g(n) = n^2$

$n^2 + n + 1 \leq c(n^2)$

The time complexity of the above function is $O(n^2)$, because the above function has to run for n^2 time at least.

Omega Notation(Ω)

Omega notation is used to denote the lower bound of the algorithm; it represents the minimum running time of an algorithm. Therefore, it provides the best-case complexity of any algorithm.



$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c_0 \text{ and } c_1, \text{ such that } 0 \leq c_0g(n) \leq f(n) \text{ for all } n \geq c_1\}.$

For Example:

Let,

1. $f(n) = n^2 + n$

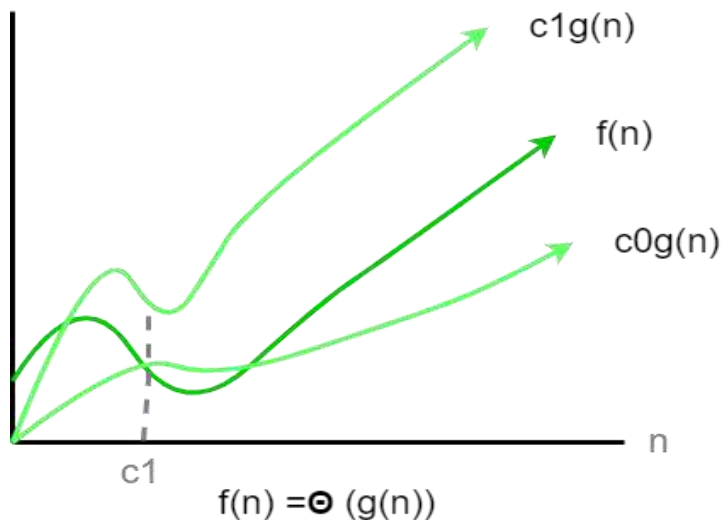
Then, the best-case time complexity will be $\Omega(n^2)$

2. $f(n) = 100n + \log(n)$

Then, the best-case time complexity will be $\Omega(n)$.

Theta Notation(Θ)

Theta notation is used to denote the average bound of the algorithm; it bounds a function from above and below, that's why it is used to represent exact asymptotic behaviour.



$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_0, c_1 \text{ and } c_2, \text{ such that } 0 \leq c_0g(n) \leq f(n) \leq c_1g(n) \text{ for all } n \geq c_2\}$

Difference between Big O Notation, Omega Notation, and Theta Notation

Parameter	Big O Notation (O)	Omega Notation (Ω)	Theta Notation (Θ)
Definition	Describes an upper bound on the time or space complexity of an algorithm.	Describes a lower bound on the time or space complexity of an algorithm.	Describes both an upper and a lower bound on the time or space complexity.
Purpose	Used to characterize the worst-case scenario of an algorithm.	Used to characterize the best-case scenario of an algorithm.	Used to characterize an algorithm's precise bound (both worst and best cases).
Interpretation	Indicates the maximum rate of growth of the algorithm's complexity.	Indicates the minimum rate of growth of the algorithm's complexity.	Indicates the exact rate of growth of the algorithm's complexity.
Mathematical Expression	$f(n) = O(g(n))$ if \exists constants $c > 0$, n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.	$f(n) = \Omega(g(n))$ if \exists constants $c > 0$, n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.	$f(n) = \Theta(g(n))$ if \exists constants $c_1, c_2 > 0$, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
Focus	Focuses on the upper limit of performance (less efficient aspects).	Focuses on the lower limit of performance (more efficient aspects).	Focuses on both the upper and lower limits, providing a balanced view of performance.
Usage in Algorithm Analysis	It is commonly used to analyze efficiency, especially concerning worst-case performance.	Used to demonstrate the effectiveness under optimal conditions.	Used to provide a precise analysis of algorithm efficiency in typical scenarios.
Common Usage	Predominant in theoretical and practical applications for worst-case analysis.	It is less common than Big O but important for understanding best-case efficiency.	Used when an algorithm exhibits a consistent performance across different inputs.
Examples	Searching in an unsorted list: $O(n)$.	Inserting an element in a sorted array: $\Omega(1)$.	Linear search in a sorted array, where the element is always in the middle: $\Theta(n)$.

Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time $T(n)$ of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \theta(1) \text{ if } n=1$$

$$2T\left(\frac{n}{2}\right) + \theta(n) \text{ if } n>1$$

There are three methods for solving Recurrence:

1. Substitution Method
2. Recursion Tree Method
3. Master Method

1. Substitution Method:

The Substitution Method consists of two types

1. Forward substitution
2. Backward substitution

1. Forward substitution

This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed.

For example:

Consider a recurrence relation

$$T(n)=T(n-1)+n$$

With initial condition $T(0)=0$

$$\text{Let } T(n)=T(n-1)+n$$

$$\text{If } n=1 \text{ then } T(1)=1$$

$$\text{If } n=2 \text{ then } T(2)=3$$

$$\text{If } n=3 \text{ then } T(3)=6$$

$$T(n) = n(n+1)/2 = n^2/2 + n/2$$

We can denote $T(n)$ in terms of Big oh notation as **$T(n)=O(n^2)$**

2. Backward substitution

In this method backward values are substituted recursively in order to derive some formula.

For example,

Consider a recurrence relation

$$T(n) = T(n-1) + n$$

With initial condition $T(0) = 0$

$$T(n-1) = T(n-1-1) + (n-1)$$

$$T(n) = T(n-2) + (n-1) + n$$

$$\text{Let } T(n-2) = T(n-2-1) + (n-2)$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

::

$$T(n) = T(n-k) + (n-k-1) + T(n-k-2) + \dots + n$$

If $k=n$ then

$$T(n) = T(0) + 1 + 2 + \dots + n$$

$$T(n) = 0 + 1 + 2 + \dots + n$$

$$T(n) = n(n+1)/2 = n^2/2 + n/2$$

We can denote $T(n)$ in terms of Big oh notation as $T(n) = O(n^2)$

2. Tree Method

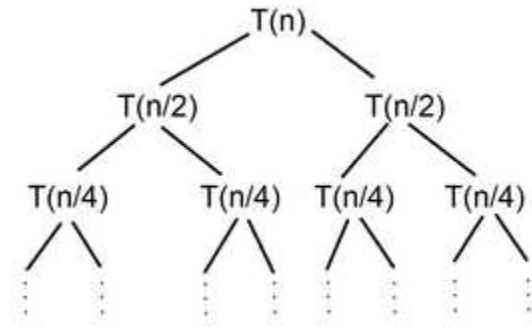
Steps to solve recurrence relation using recursion tree method:

1. Draw a recursive tree for given recurrence relation
2. Calculate the cost at each level and count the total no of levels in the recursion tree.
3. Count the total number of nodes in the last level and calculate the cost of the last level
4. Sum up the cost of all the levels in the recursive tree

For example

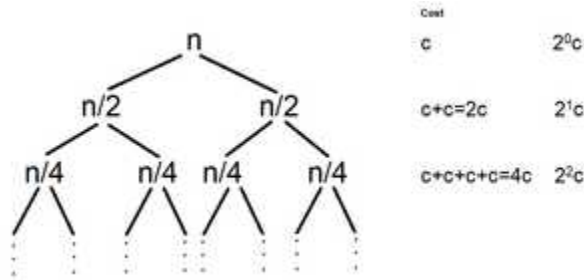
$$T(n) = 2T(n/2) + c$$

Step 1: Draw a recursive tree



Recursion Tree

Step 2: Calculate the work done or cost at each level and count total no of levels in recursion tree



Recursive Tree with each level cost

Count the total number of levels –

Choose the longest path from root node to leaf node

$$n/2^0 \rightarrow n/2^1 \rightarrow n/2^2 \rightarrow \dots \rightarrow n/2^k$$

Size of problem at last level = $n/2^k$

At last level size of problem becomes 1

$$n/2^k = 1$$

$$2^k = n$$

$$k = \log_2(n)$$

Total no of levels in recursive tree = $k + 1 = \log_2(n) + 1$

Step 3: Count total number of nodes in the last level and calculate cost of last level

No. of nodes at level 0 = $2^0 = 1$

No. of nodes at level 1 = $2^1 = 2$

.....

No. of nodes at level $\log_2(n) = 2^{\log_2(n)} = n^{\log_2(2)} = n$

Cost of sub problems at level $\log_2(n)$ (last level) = $n \times T(1) = n \times 1 = n$

Step 4: Sum up the cost all the levels in recursive tree

$T(n) = c + 2c + 4c + \dots + (\text{no. of levels}-1) \text{ times} + \text{last level cost}$

$= c + 2c + 4c + \dots + \log_2(n) \text{ times} + \Theta(n)$

$= c(1 + 2 + 4 + \dots + \log_2(n) \text{ times}) + \Theta(n)$

$1 + 2 + 4 + \dots + \log_2(n) \text{ times} \rightarrow 2^0 + 2^1 + 2^2 + \dots + \log_2(n) \text{ times} \rightarrow \text{Geometric Progression(G.P.)}$

$$= c(n) + \Theta(n)$$

3. Master Method

The Master Method is used for solving the following types of recurrence

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \text{ with } a \geq 1 \text{ and } b \geq 1 \text{ be constant \& } f(n) \text{ be a function and } \frac{n}{b} \text{ can be interpreted as}$$

Let $T(n)$ is defined on non-negative integers by the recurrence.

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND } af(n/b) < cf(n) \text{ for large } n \end{cases} \left\{ \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array} \right.$$

Lower Bound

Steps to find the recurrence using lower bound:

1. Guess the solutions
2. Use the mathematic induction to find the boundary condition and shows that the guess is correct.

Searching

It is a technique in which the location of desired element is obtained. The searching technique is based on search key. This key is compared with array elements. If the key and the current element match then position of that element is returned.

Commonly used searching algorithms are,

1. Linear search
2. Binary search
3. Interpolation search

1. Linear Search

Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

Linear Search Algorithm

The algorithm for linear search is relatively simple. The procedure starts at the very first index of the input array to be searched.

Step 1 – Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

Step 2 – If the value matches with the key, return the position at which the value was found.

Step 3 – If the value does not match with the key, compare the next element in the array.

Step 4 – Repeat Step 3 until there is a match found. Return the position at which the match was found.

Step 5 – If it is an unsuccessful search, print that the element is not present in the array and exit the program.

Pseudo code

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

Analysis

Linear search traverses through every element sequentially therefore, the best case is when the element is found in the very first iteration. The best-case time complexity would be **O(1)**.

However, the worst case of the linear search method would be an unsuccessful search that does not find the key value in the array, it performs n iterations. Therefore, the worst-case time complexity of the linear search algorithm would be **O(n)**.

Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

Step 1

The linear search starts from the 0th index. Compare the key element with the value in the 0th index, 34.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

=
47

However, $47 \neq 34$. So it moves to the next element.

Step 2

Now, the key is compared with value in the 1st index of the array.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

=
47

Still, $47 \neq 10$, making the algorithm move for another iteration.

Step 3

The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

=
47

Step 4

Now the element in 3rd index, 27, is compared with the key value, 47. They are not equal so the algorithm is pushed forward to check the next element.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

= 47

Step 5

Comparing the element in the 4th index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

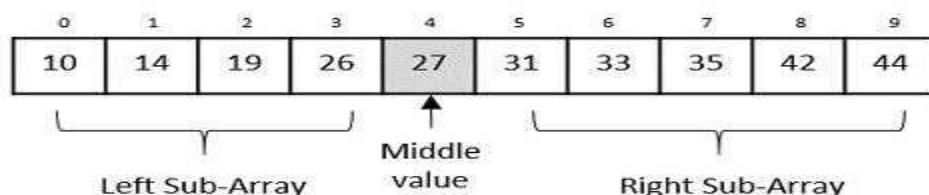
= 47

The output achieved is “Element found at 4th index”.

2. Binary Search

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a sub array reduces to zero.



Binary Search Algorithm

Step 1 – Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 – If it does not match the key value, check if the key value is either greater than or less than the median value.

Step 3 – If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 – Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 – If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

Pseudocode

Procedure binary_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + (upperBound - lowerBound) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

Analysis

To achieve a successful search, after the last iteration the length of array must be 1. Hence,

$$n/2^i = 1$$

That gives us – $n = 2^i$

Applying log on both sides,

$$\log n = \log 2^i$$

$$\log n = i \cdot \log 2$$

$$i = \log n$$

The time complexity of the binary search algorithm is **$O(\log n)$**

Example

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.


0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44

First, we shall determine half of the array by using this formula –

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Here it is, $0 + (9 - 0) / 2 = 4$ (integer value of 4.5). So, 4 is the mid of the array.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44


We change our low to $\text{mid} + 1$ and find the new mid value again.

$$\text{low} = \text{mid} + 1$$

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44




The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44

Hence, we calculate the mid again. This time it is 5.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44



We compare the value stored at location 5 with our target value. We find that it is a match.

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44

We conclude that the target value 31 is stored at location 5.

3. Interpolation Search

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

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





If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method –

Interpolation Search Algorithm

1. Start searching data from middle of the list.
2. If it is a match, return the index of the item, and exit.
3. If it is not a match, probe position.
4. Divide the list using probing formula and find the new middle.
5. If data is greater than middle, search in higher sub-list.
6. If data is smaller than middle, search in lower sub-list.
7. Repeat until match.

Example

To understand the step-by-step process involved in the interpolation search, let us look at an example and work around it.

Consider an array of sorted elements given below –

0	1	2	3	4	5	6	7	8	9
10	14	19	26	27	31	33	35	42	44

Let us search for the element 19.

Solution

Unlike binary search, the middle point in this approach is chosen using the formula

$$\text{Mid} = \text{low} + \left[\frac{(\text{high} - \text{low}) * (\text{key} - \text{A}[\text{low}])}{\text{A}[\text{high}] - \text{A}[\text{Low}]} \right]$$

So in this given array input,

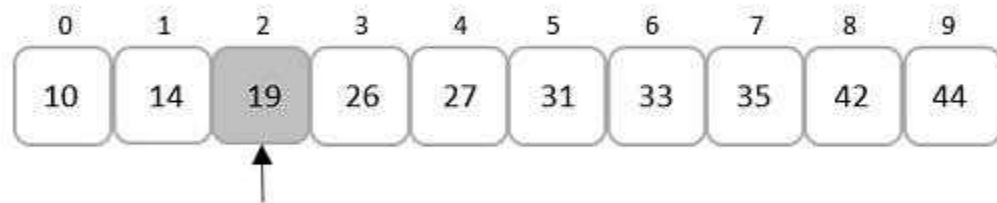
$Lo = 0, A[Lo] = 10$

$Hi = 9, A[Hi] = 44$

$X = 19$

Applying the formula to find the middle point in the list,

Since, mid is an index value, we only consider the integer part of the decimal. That is, $mid = 2$.



Comparing the key element given, that is 19, to the element present in the mid index, it is found that both the elements match.

Therefore, the element is found at index 2.

String Matching Algorithm

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 or 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 problem is to find an integer s , called **valid shift** where $0 \leq s < n-m$ 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 items of P and T are characters drawn from some finite alphabet such as $\{0, 1\}$ or $\{A, B, ..., Z, a, b, ..., z\}$.

There are different types of methods used to find the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Knuth-Morris-Pratt Algorithm

1. Naive String Matching Algorithm

The naïve approach tests all the possible placements 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. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. for $s \leftarrow 0$ 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 $O(n - m + 1)$.

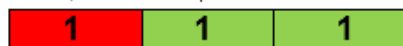
Example:

Suppose $T = 1011101110$ $P = 111$ Find all the Valid Shift

T = Text



$S=0$



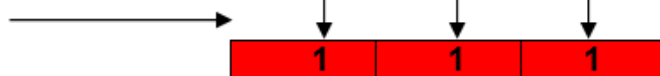
P = Pattern



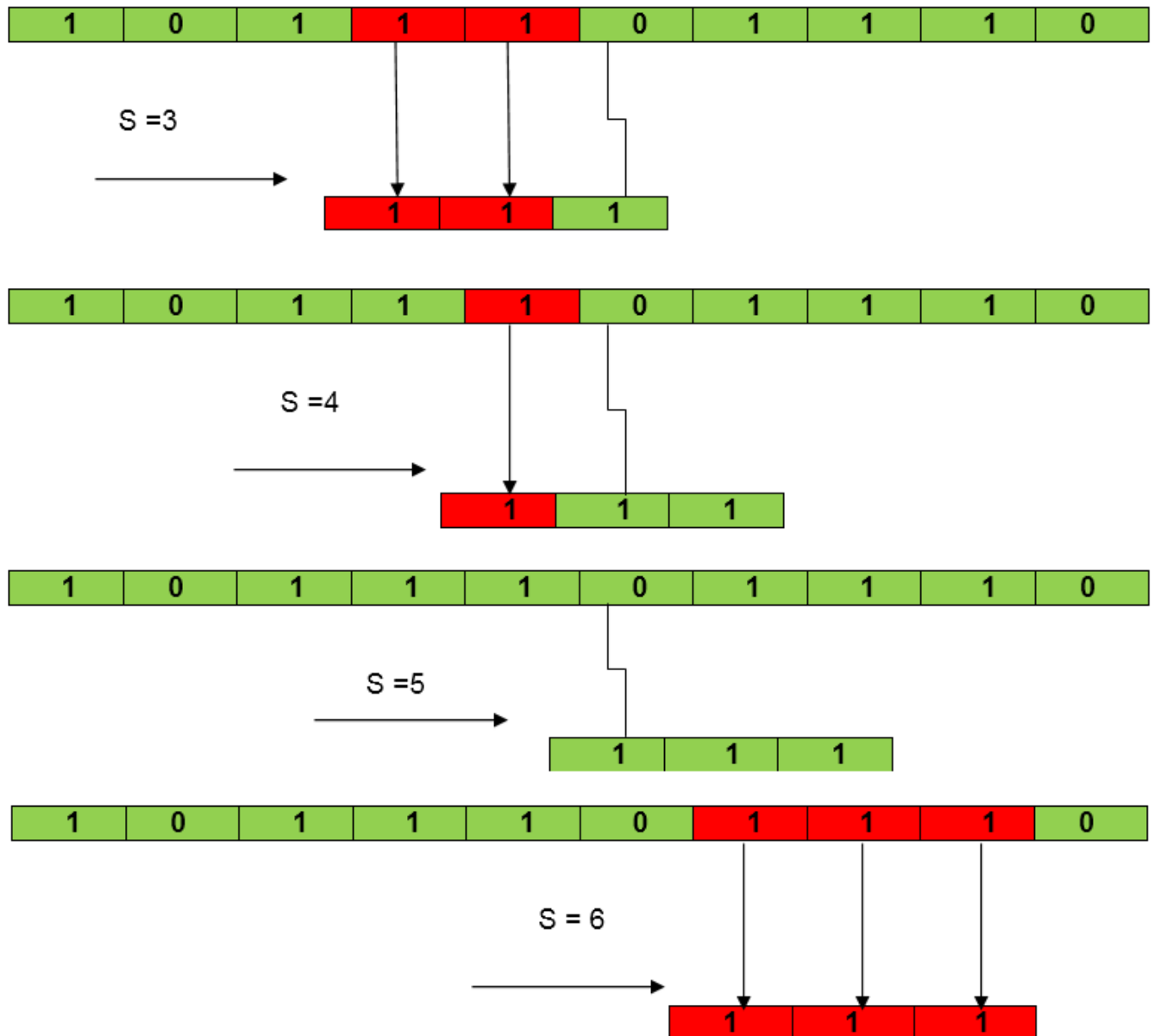
$S=1$



$S=2$



So, S=2 is a Valid Shift



2. 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. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $h \leftarrow d^{m-1} \bmod q$
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$

6. for $i \leftarrow 1$ to m
7. do $p \leftarrow (dp + P[i]) \bmod q$
8. $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9. for $s \leftarrow 0$ 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 $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

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 $P \bmod Q = 26 \bmod 11 = 4$
5. Now find the exact match of $P \bmod Q...$

Solution:

T =

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

P =

2	6
---	---

S = 0
→

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$31 \bmod 11 = 9$ not equal to 4

S = 1
→

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$14 \bmod 11 = 3$ not equal to 4

S = 2
→

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$41 \bmod 11 = 8$ not equal to 4

S = 3
→

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$15 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



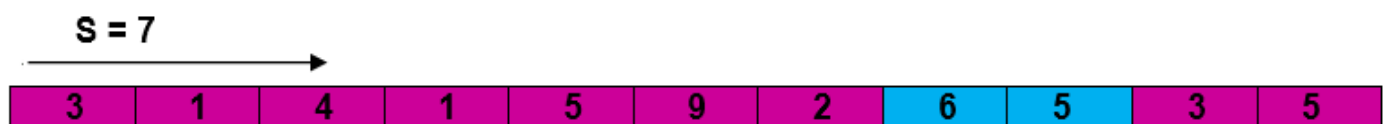
$59 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



$92 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



$26 \bmod 11 = 4$ EXACT MATCH



$65 \bmod 11 = 10$ not equal to 4



$53 \bmod 11 = 9$ not equal to 4



$35 \bmod 11 = 2$ 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.

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.

Following pseudo code compute the prefix function, Π :

COMPUTE- PREFIX- FUNCTION (P)

1. $m \leftarrow \text{length}[P]$ // 'p' pattern to be matched
2. $\Pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. for $q \leftarrow 2$ to m
5. do while $k > 0$ and $P[k+1] \neq P[q]$
6. do $k \leftarrow \Pi[k]$
7. If $P[k+1] = P[q]$
8. then $k \leftarrow k+1$
9. $\Pi[q] \leftarrow k$
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: $m = \text{length}[p] = 7$

$$\Pi[1] = 0$$

k

=

0

Step 1: $q = 2, k = 0$

$$\Pi[2] = 0$$

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

Step 2: $q = 3, k = 0$

$$\Pi[3] = 1$$

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

Step3: $q = 4, k = 1$

$$\Pi[4] = 2$$

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

Step4: $q = 5, k = 2$

$$\Pi[5] = 3$$

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

Step5: $q = 6, k = 3$

$$\Pi[6] = 0$$

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

Step6: $q = 7, k = 1$

$$\Pi[7] = 1$$

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	
p	a	b	A	b	a	c	
π	0	0	1	2	3	0	

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. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\Pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$ // numbers of characters matched
5. for $i \leftarrow 1$ to n // scan S from left to right
6. do while $q > 0$ and $P[q + 1] \neq T[i]$
7. do $q \leftarrow \Pi[q]$ // next character does not match
8. If $P[q + 1] = T[i]$

```

9. then  $q \leftarrow q + 1$  // next character matches
10. If  $q = m$  // is all of p matched?
11. then print "Pattern occurs with shift"  $i - m$ 
12.  $q \leftarrow \Pi[q]$  // look for the next match

```

ADVERTISEMENT

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 $O(n)$.

Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Insertion Sort Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

Algorithm: Insertion-Sort(A)

for $j = 2$ to $A.length$

$key = A[j]$

$i = j - 1$

 while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Analysis

Run time of this algorithm is very much dependent on the given input.

If the given numbers are sorted, this algorithm runs in $O(n)$ time. If the given numbers are in reverse order, the algorithm runs in $O(n^2)$ time.

Example

We take an unsorted array for our example.

0	1	2	3	4	5	6	7
14	33	27	10	35	19	44	42

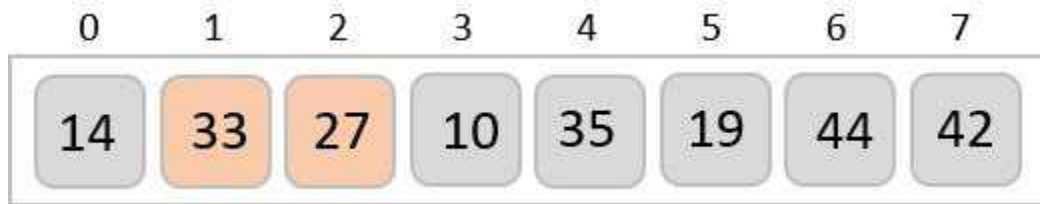
Insertion sort compares the first two elements.

0	1	2	3	4	5	6	7
14	33	27	10	35	19	44	42

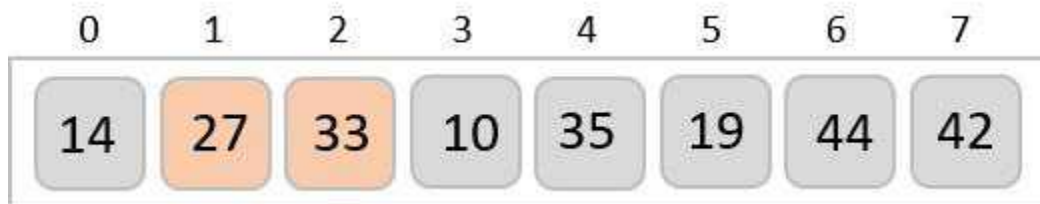
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

0	1	2	3	4	5	6	7
14	33	27	10	35	19	44	42

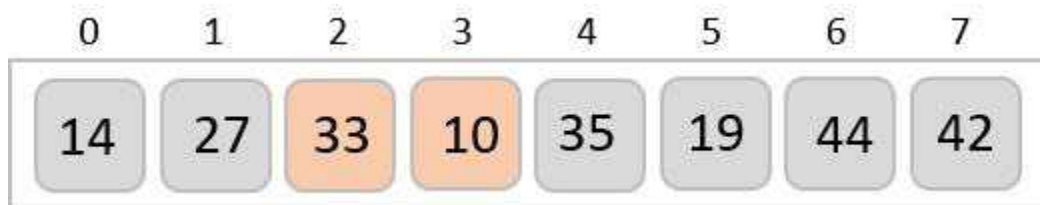
Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



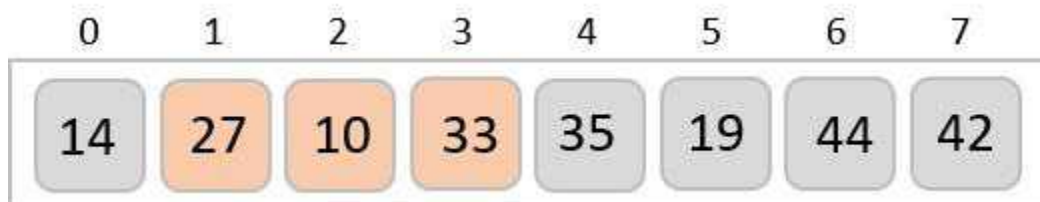
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10. These values are not in a sorted order.



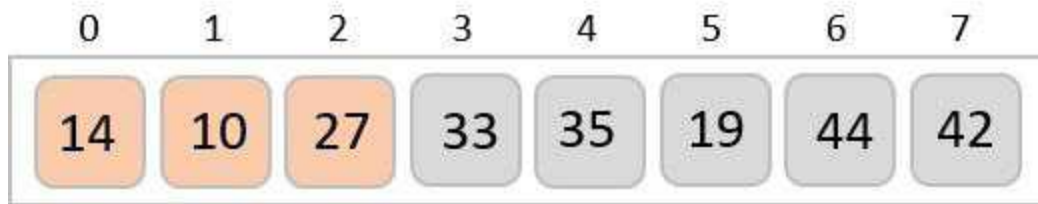
So they are swapped.



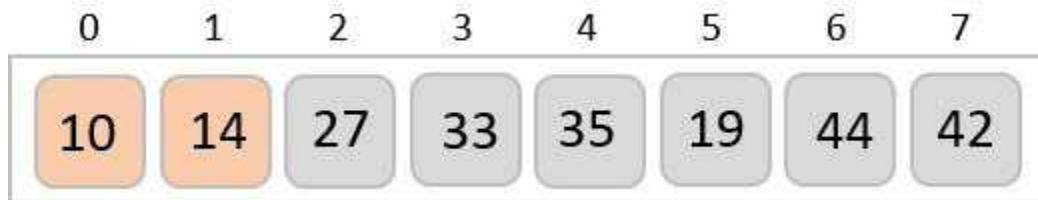
However, swapping makes 27 and 10 unsorted.



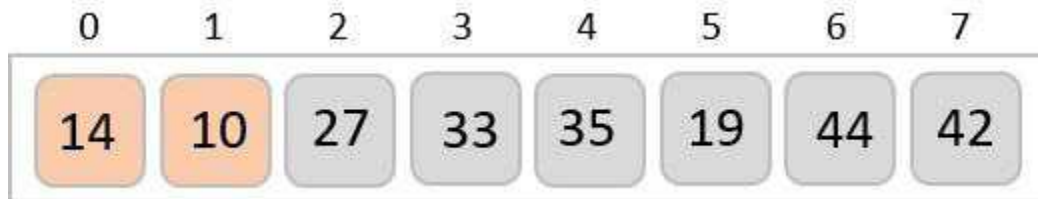
Hence, we swap them too.



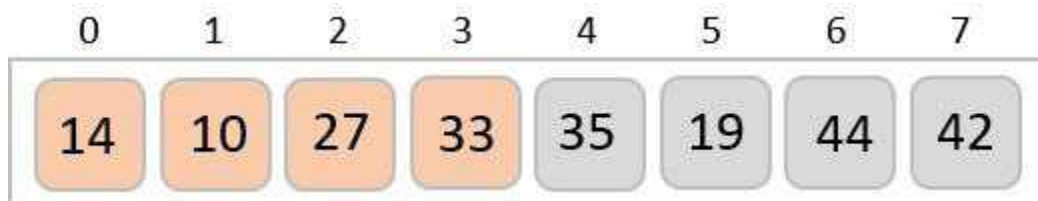
Again we find 14 and 10 in an unsorted order.



We swap them again.



By the end of third iteration, we have a sorted sub-list of 4 items.



Heap Sort

Heap Sort is an efficient sorting technique based on the heap data structure.

The heap is a nearly-complete binary tree where the parent node could either be minimum or maximum. The heap with minimum root node is called **min-heap** and the root node with maximum root node is called **max-heap**. The elements in the input data of the heap sort algorithm are processed using these two methods.

The heap sort algorithm follows two main operations in this procedure –

- Builds a heap H from the input data using the **heapify** (explained further into the chapter) method, based on the way of sorting – ascending order or descending order.
- Deletes the root element of the root element and repeats until all the input elements are processed.

The heap sort algorithm heavily depends upon the heapify method of the binary tree. So what is this heapify method?

Heapify Method

The *heapify* method of a binary tree is to convert the tree into a heap data structure. This method uses recursion approach to heapify all the nodes of the binary tree.

Note – The binary tree must always be a complete binary tree as it must have two children nodes always.

The complete binary tree will be converted into either a max-heap or a min-heap by applying the *heapify* method.

To know more about the heapify algorithm, please [click here](#).

Heap Sort Algorithm

As described in the algorithm below, the sorting algorithm first constructs the heap ADT by calling the Build-Max-Heap algorithm and removes the root element to swap it with the minimum valued node at the leaf. Then the heapify method is applied to rearrange the elements accordingly.

```
Algorithm: Heapsort(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
exchange A[1] with A[i]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
```

Analysis

The heap sort algorithm is the combination of two other sorting algorithms: insertion sort and merge sort.

The similarities with insertion sort include that only a constant number of array elements are stored outside the input array at any time.

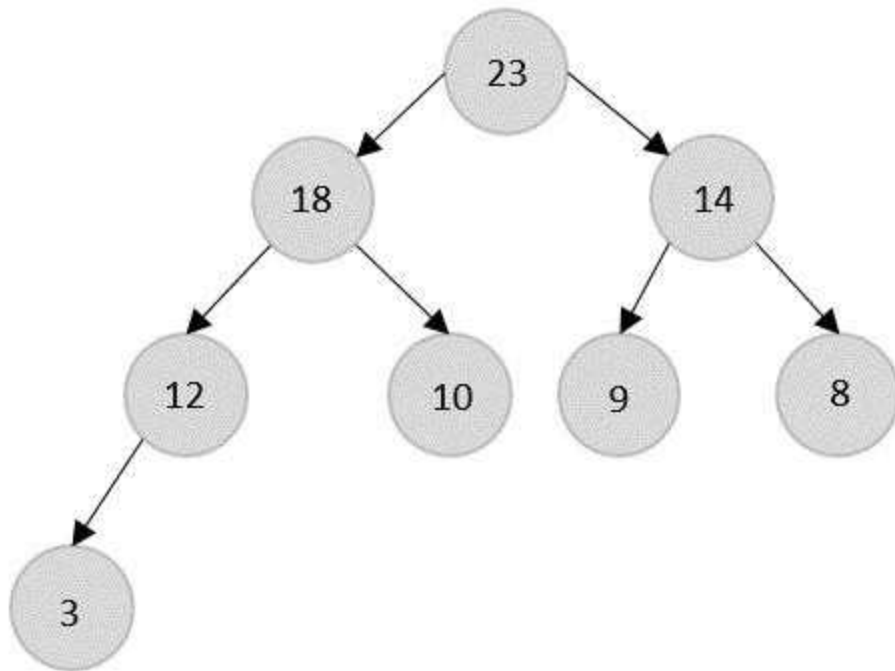
The time complexity of the heap sort algorithm is **$O(n \log n)$** , similar to merge sort.

Example

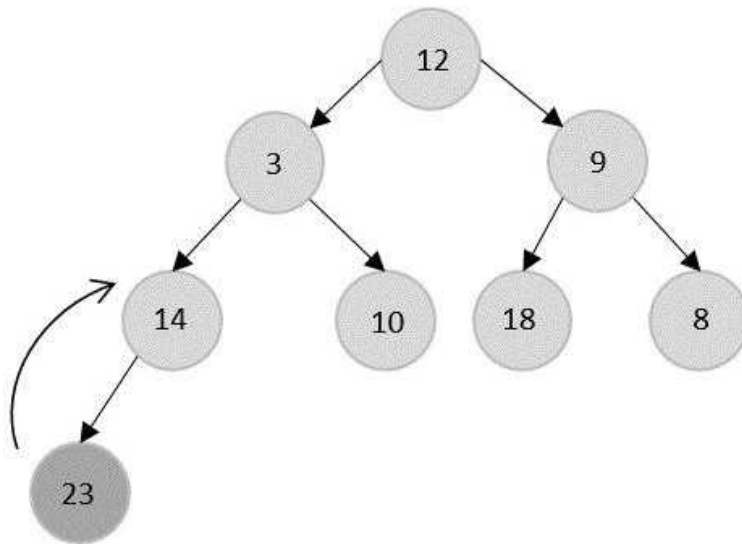
Let us look at an example array to understand the sort algorithm better –

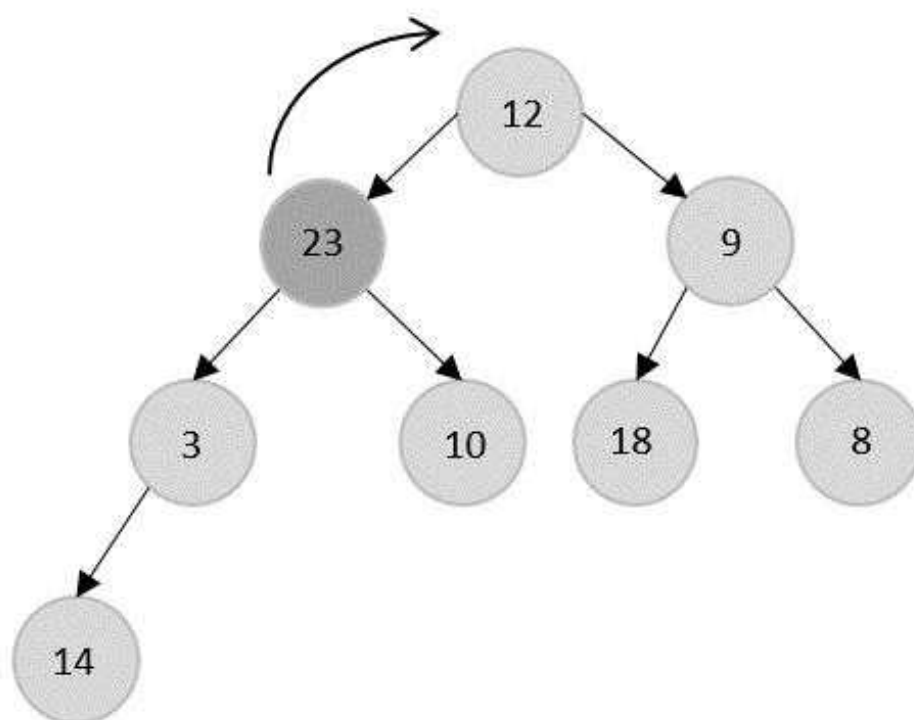
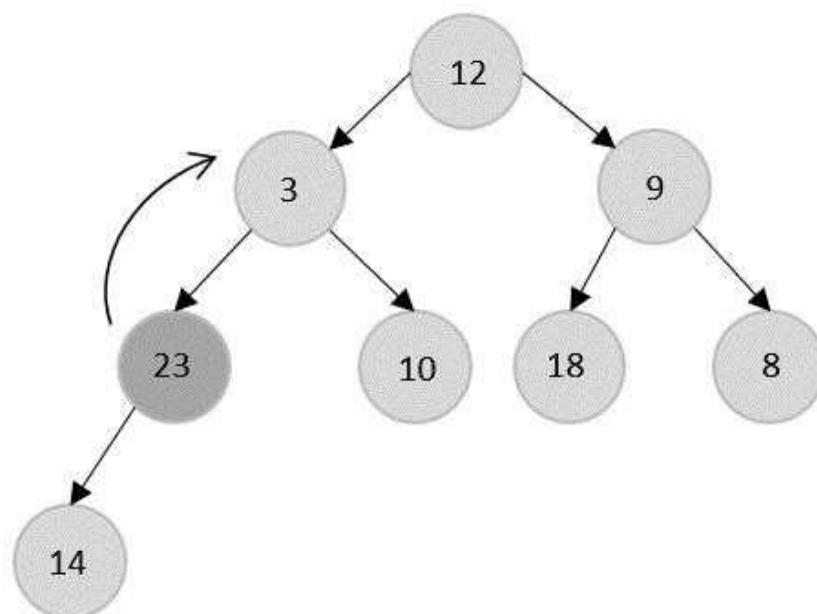
12 3 9 14 10 18 8 23

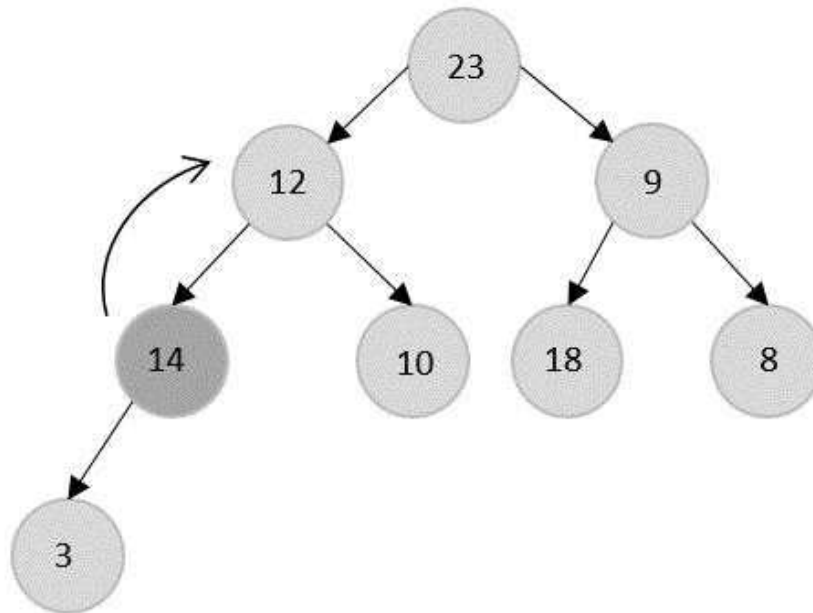
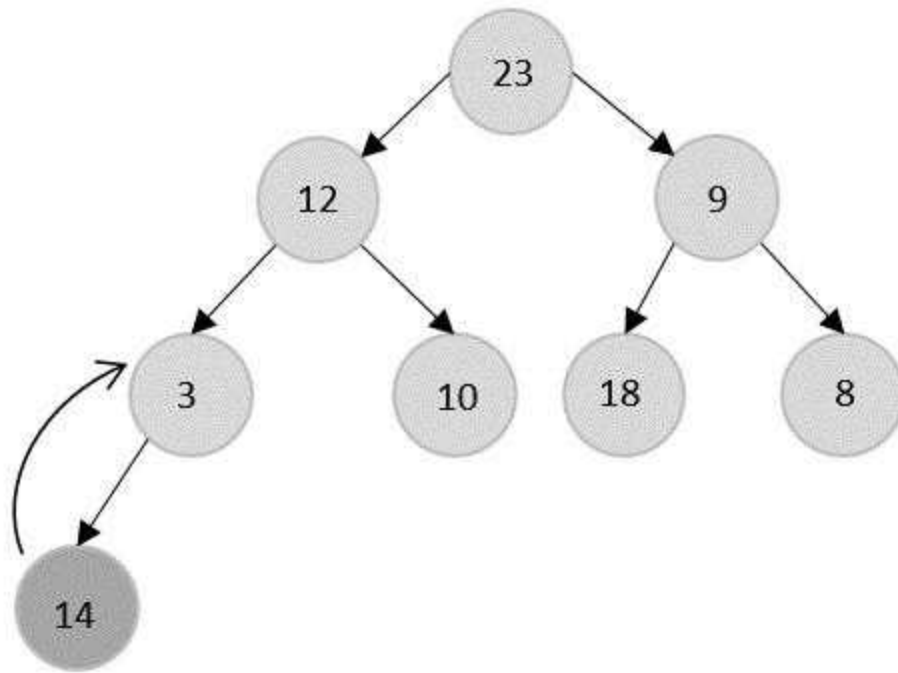
Building a heap using the BUILD-MAX-HEAP algorithm from the input array –

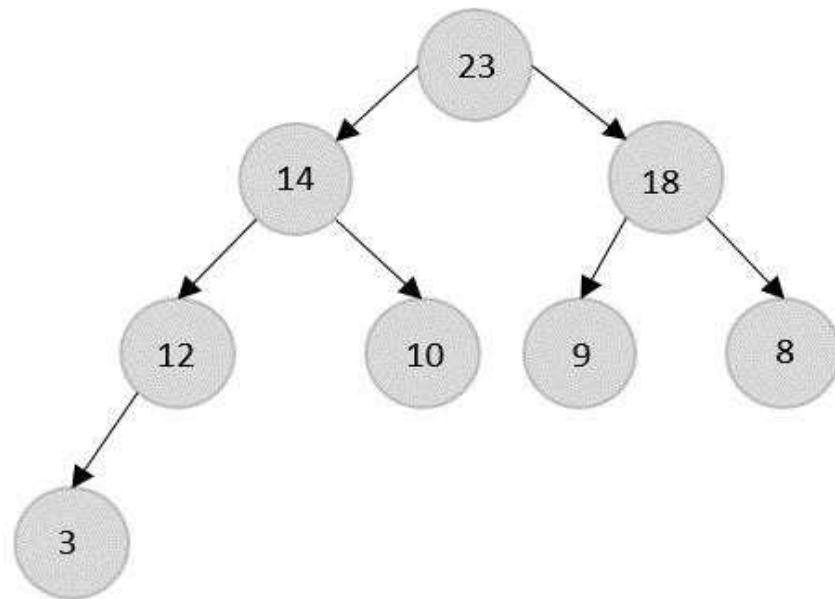
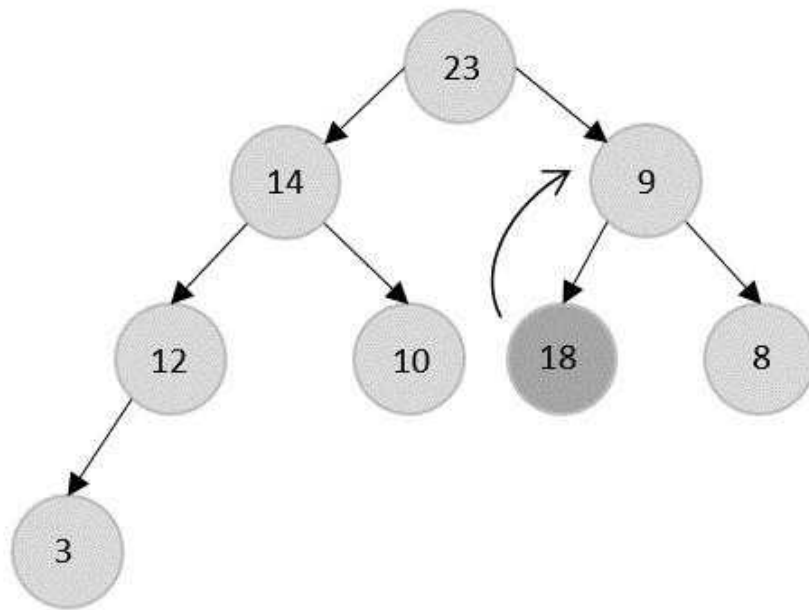


Rearrange the obtained binary tree by exchanging the nodes such that a heap data structure is formed.





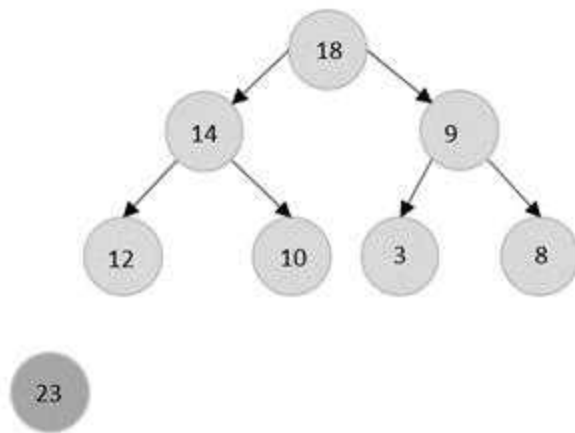




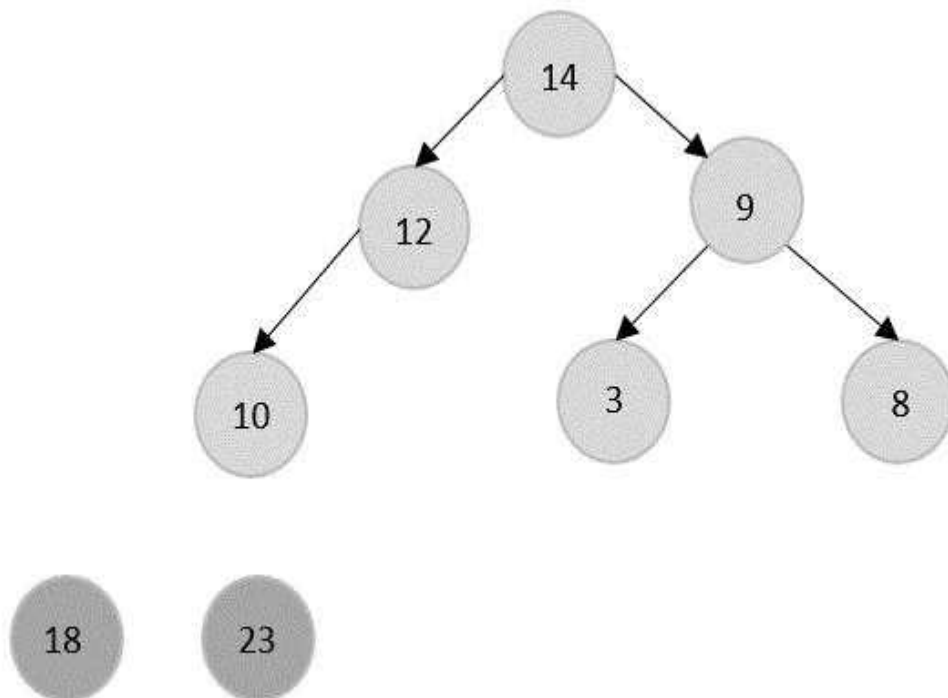
The Heapify Algorithm

Applying the heapify method, remove the root node from the heap and replace it with the next immediate maximum valued child of the root.

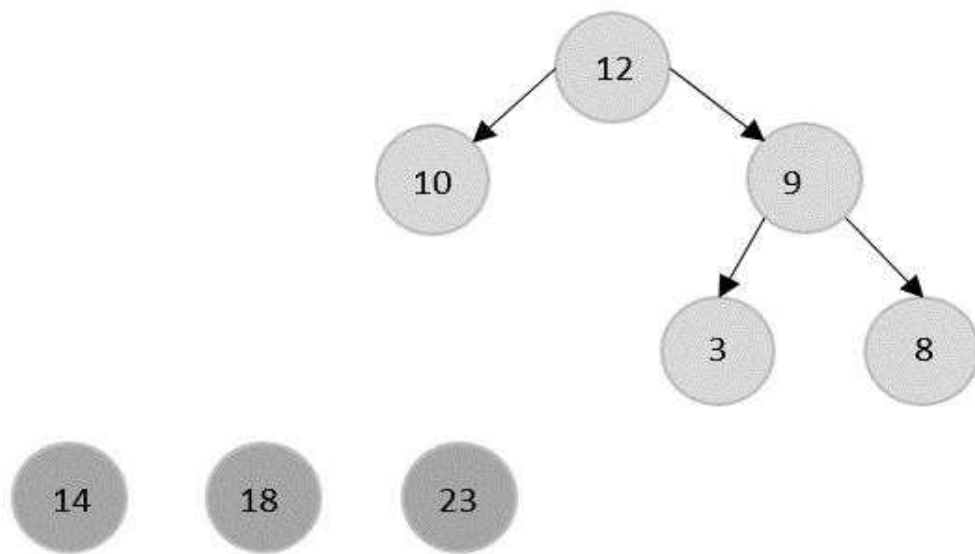
The root node is 23, so 23 is popped and 18 is made the next root because it is the next maximum node in the heap.



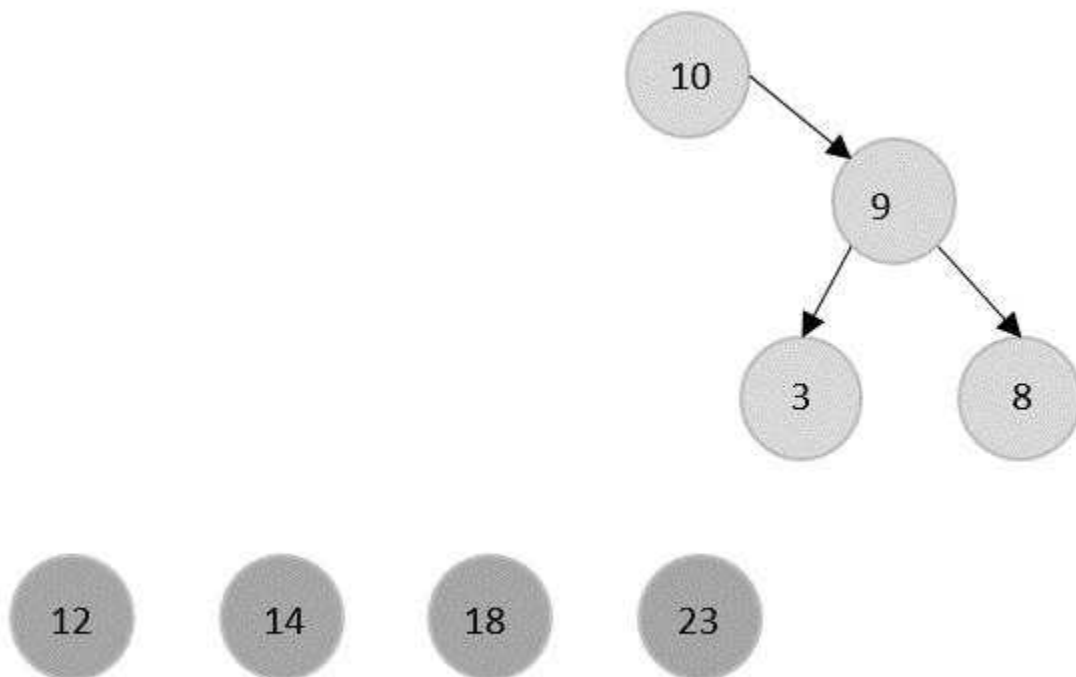
Now, 18 is popped after 23 which is replaced by 14.



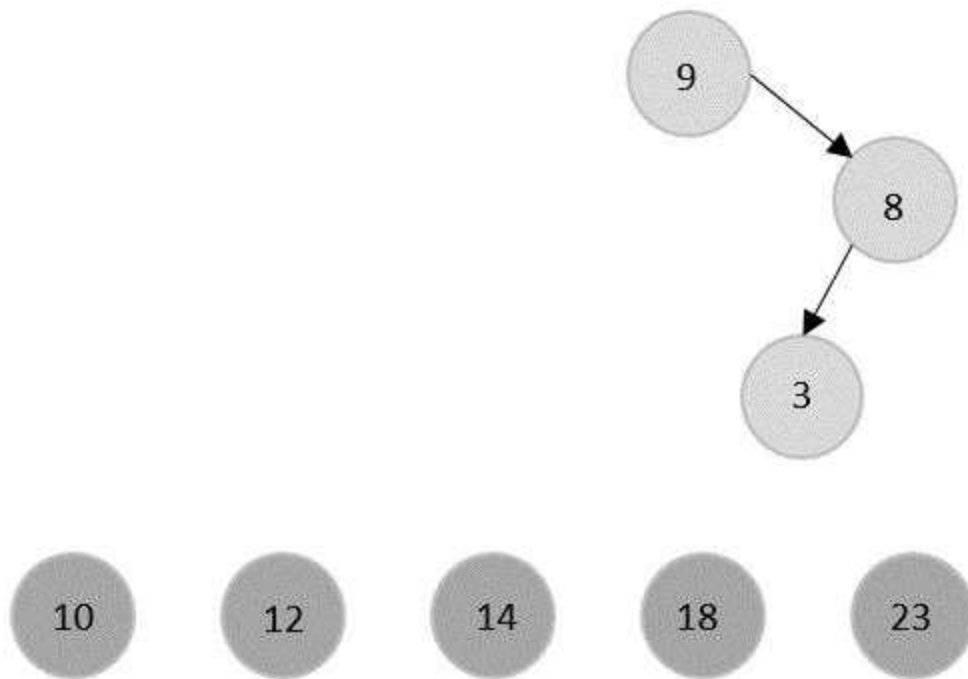
The current root 14 is popped from the heap and is replaced by 12.



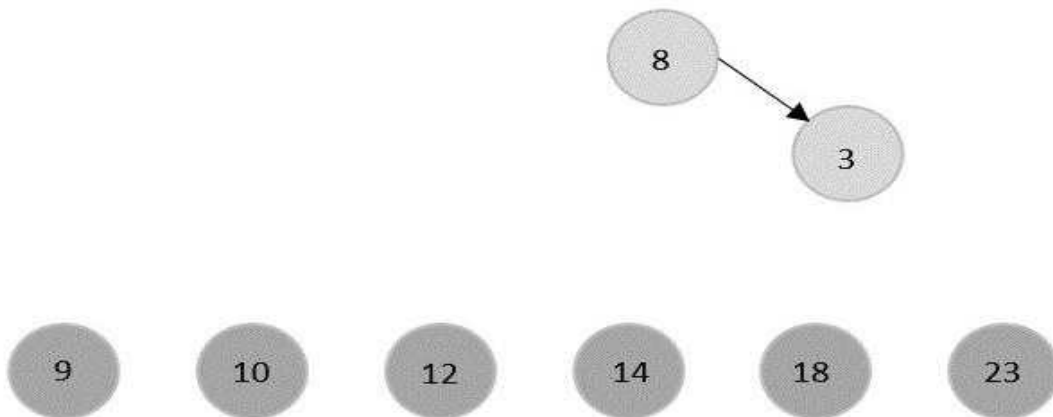
12 is popped and replaced with 10.



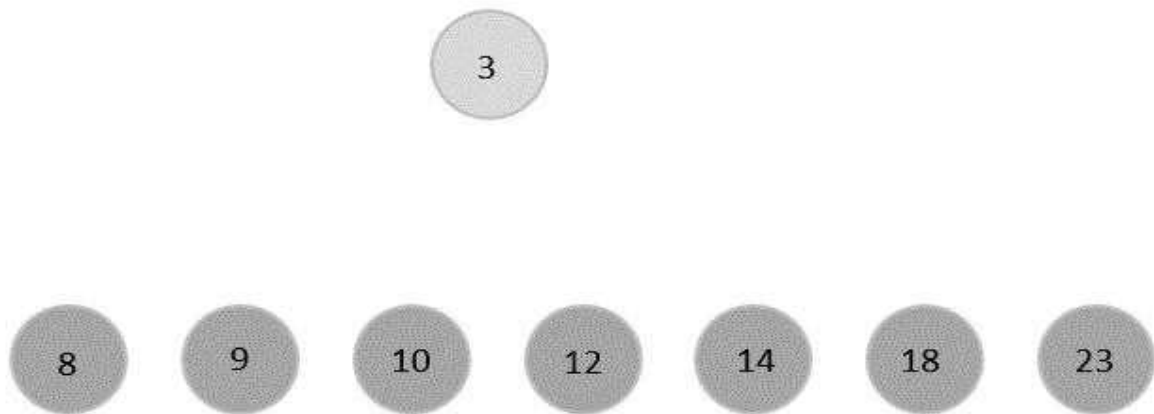
Similarly all the other elements are popped using the same process.



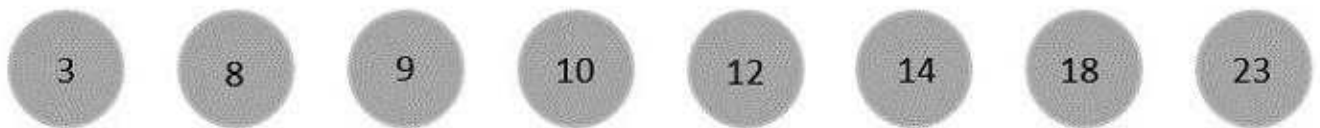
Here the current root element 9 is popped and the elements 8 and 3 are remained in the tree.



Then, 8 will be popped leaving 3 in the tree.



After completing the heap sort operation on the given heap, the sorted elements are displayed as shown below –



Every time an element is popped, it is added at the beginning of the output array since the heap data structure formed is a max-heap. But if the heapify method converts the binary tree to the min-heap, add the popped elements are on the end of the output array.

The final sorted list is,

3 8 9 10 12 14 18 23

UNIT II GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method – Matching: Maximum bipartite matching

Definition

A graph $G(V, E)$ is a non-linear data structure that consists of node and edge pairs of objects connected by links.

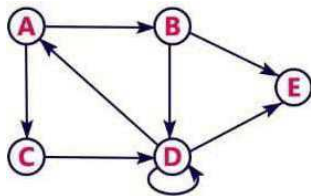
There are 2 types of graphs:

1. Directed
2. Undirected

1. Directed graph

A graph with only directed edges is said to be a directed graph. Example

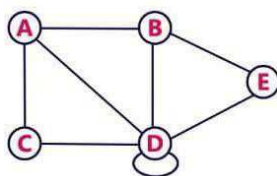
The following directed graph has 5 vertices and 8 edges. This graph G can be defined as $G = (V, E)$, where $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (A, C), (B, E), (B, D), (D, A), (D, E), (C, D), (D, D)\}$.



2. Undirected graph

A graph with only undirected edges is said to be an undirected graph. Example

The following is an undirected graph.



Representation of Graphs

Graph data structure is represented using the following representations.

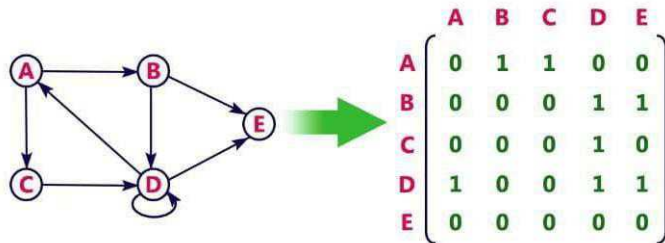
1. Adjacency Matrix
2. Adjacency List

1. Adjacency Matrix

In this representation, the graph can be represented using a matrix of size $n \times n$, where n is the number of vertices.

This matrix is filled with either 1's or 0's.

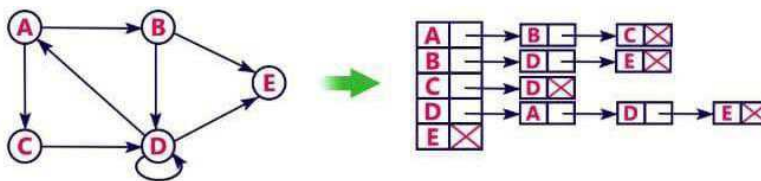
Here, 1 represents that there is an edge from row vertex to column vertex, and 0 represents that there is no edge from row vertex to column vertex.



2. Adjacency list

In this representation, every vertex of the graph contains a list of its adjacent vertices.

If the graph is not dense, i.e., the number of edges is less, then it is efficient to represent the graph through the adjacency list.



Graph traversals

Graph traversal is a technique used to search for a vertex in a graph. It is also used to decide the order of vertices to be visited in the search process.

A graph traversal finds the edges to be used in the search process without creating loops. This means that, with graph traversal, we can visit all the vertices of the graph without getting into a looping path. There are two graph traversal techniques:

1. DFS (Depth First Search)
2. BFS (Breadth-First Search)

Applications of graphs

Social network graphs : To tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom, or other relationships in social structures. An example is the twitter graph of who follows whom.

Graphs in epidemiology: Vertices represent individuals and directed edges to view the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

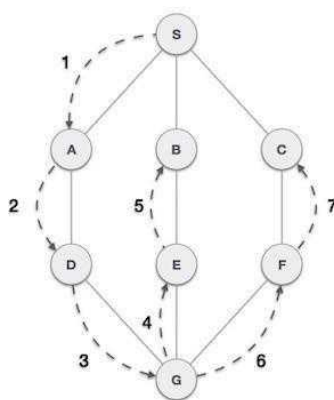
Protein-protein interactions graphs: Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used to, for example, study molecular pathway—chains of molecular interactions in a cellular process.

Network packet traffic graphs: Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

Neural networks: Vertices represent neurons and edges are the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

DFS – Depth First Search

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



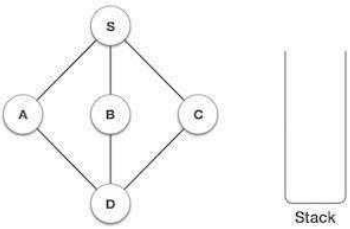
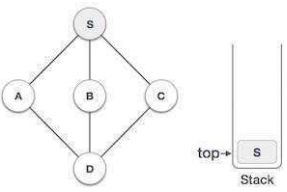
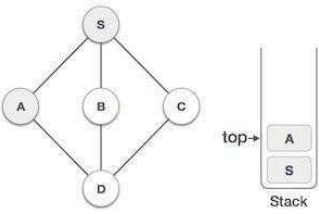
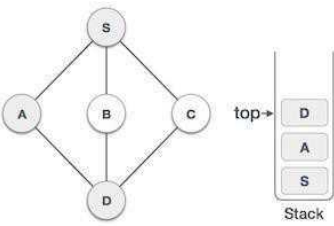
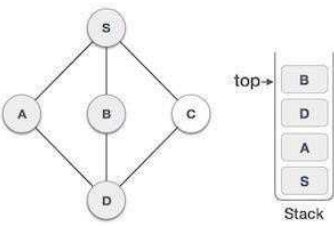
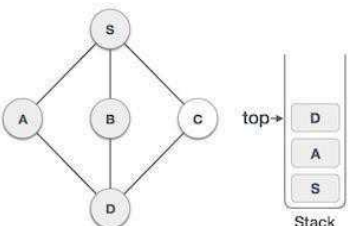
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

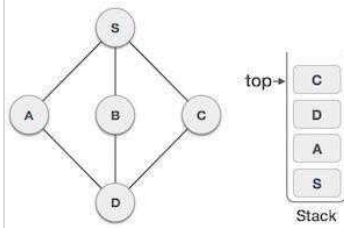
Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
------	-----------	-------------

1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.

7



Only unvisited adjacent node is
from D is C now. So we visit C, mark
it as visited and put it onto the stack.

DFS(G, u)

u.visited = true

for each $v \in G.\text{Adj}[u]$ if $v.\text{visited} == \text{false}$

DFS(G,v)

init() {

For each $u \in G$ u.visited = false

For each $u \in G$ DFS(G, u)

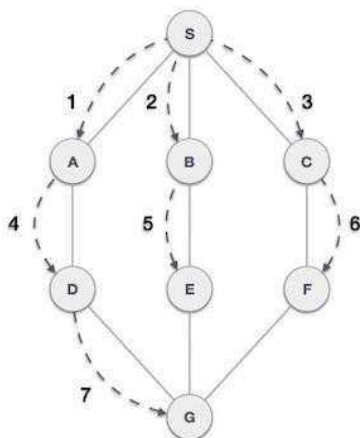
}

Application of DFS Algorithm

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

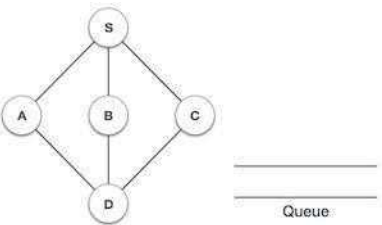
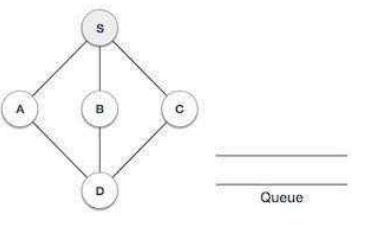
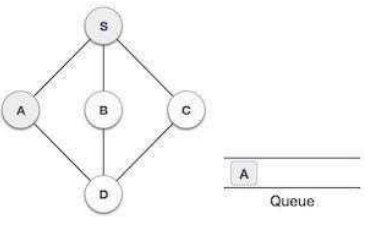
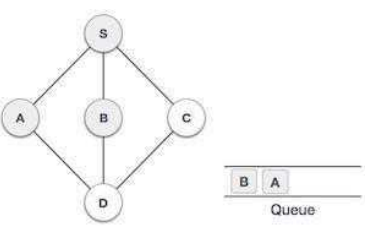
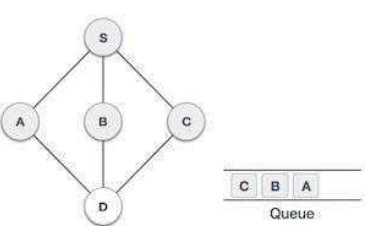


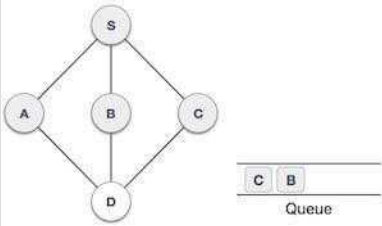
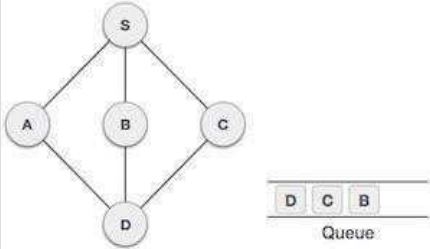
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.

6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

BFS pseudocode

create a queue Q

mark v as visited and put v into Q while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

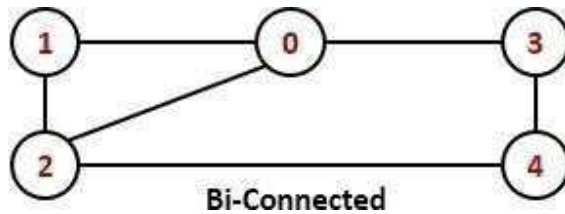
The space complexity of the algorithm is $O(V)$.

BFS Algorithm Applications

- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

Bi Connectivity Graph

An undirected graph is said to be a bi connected graph, if there are two vertex-disjoint paths between any two vertices are present. In other words, we can say that there is a cycle between any two vertices.



We can say that a graph G is a bi-connected graph if it is connected, and there are no articulation points or cut vertex are present in the graph.

To solve this problem, we will use the DFS traversal. Using DFS, we will try to find if there is any articulation point is present or not. We also check whether all vertices are visited by the DFS or not, if not we can say that the graph is not connected.

Pseudocode for Bi connectivity

isArticulation(start, visited, disc, low, parent)

Begin

time := 0 //the value of time will not be initialized for next function calls dfsChild := 0

mark start as visited

set disc[start] := time+1 and low[start] := time + 1 time := time + 1

for all vertex v in the graph G , do

if there is an edge between (start, v), then if v is visited, then

increase dfsChild parent[v] := start

if isArticulation(v , visited, disc, low, parent) is true, then return true

low[start] := minimum of low[start] and low[v] if parent[start] is ϕ AND dfsChild > 1, then

return true

if parent[start] is ϕ AND low[v] >= disc[start], then return true

else if v is not the parent of start, then low[start] := minimum of low[start] and disc[v]

done return false

End

isBiconnected(graph)

Begin

initially set all vertices are unvisited and parent of each vertices are ϕ if isArticulation(0, visited, disc, low, parent) = true, then

return false

for each node i of the graph, do if i is not visited, then

return false done

return true End

Minimum Spanning Tree

A Spanning Tree is a tree which have V vertices and $V-1$ edges. All nodes in a spanning tree are reachable from each other.

A Minimum Spanning Tree(MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. In short out of all spanning trees of a given graph, the spanning tree having minimum weight is MST.

Algorithms for finding Minimum Spanning Tree(MST):-

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

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

form a tree that includes every vertex

has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

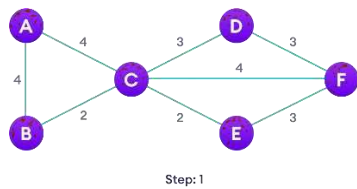
We start from one vertex and keep adding edges with the lowest weight until we reach our goal. The steps for implementing Prim's algorithm are as follows:

Initialize the minimum spanning tree with a vertex chosen at random.

Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree

Keep repeating step 2 until we get a minimum spanning tree

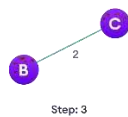
Example of Prim's algorithm



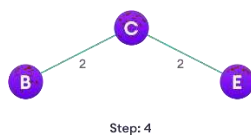
Start with a weighted graph



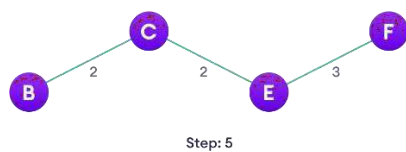
Choose a vertex



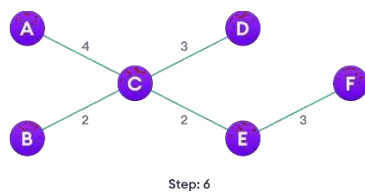
Choose the shortest edge from this vertex and add it



Choose the nearest vertex not yet in the solution



Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and $V-U$. U contains the list of vertices that have been visited and $V-U$ the list of vertices that haven't. One by one, we move vertices from set $V-U$ to set U by connecting the least weight edge.

$T = \emptyset$;

$U = \{ 1 \}$;

while ($U \neq V$)

let (u, v) be the lowest cost edge such that $u \in U$ and $v \in V - U$; $T = T \cup \{(u, v)\}$

$U = U \cup \{v\}$

Prim's Algorithm Complexity

The time complexity of Prim's algorithm is $O(E \log V)$.

Kruskal Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

form a tree that includes every vertex

has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

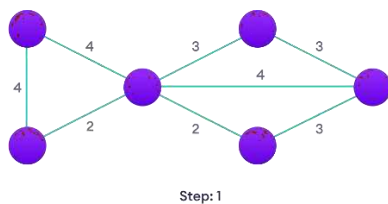
We start from the edges with the lowest weight and keep adding edges until we reach our goal. The steps for implementing Kruskal's algorithm are as follows:

Sort all the edges from low weight to high

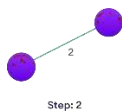
Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

Keep adding edges until we reach all vertices.

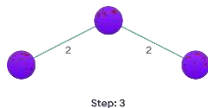
Example of Kruskal's algorithm



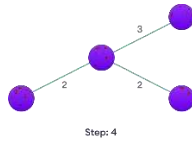
Start with a weighted graph



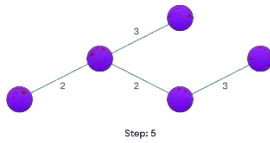
Choose the edge with the least weight, if there are more than 1, choose anyone



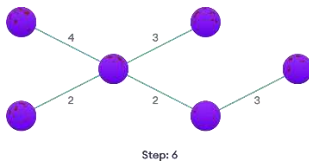
Choose the next shortest edge and add it



Choose the next shortest edge that doesn't create a cycle and add it



Choose the next shortest edge that doesn't create a cycle and add it



Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

KRUSKAL(G):

$A = \emptyset$

For each vertex $v \in G.V$:

MAKE-SET(v)

For each edge $(u, v) \in G.E$ ordered by increasing order by weight(u, v):

if FIND-SET(u) \neq FIND-SET(v):

$A = A \cup \{(u, v)\}$ UNION(u, v)

return A

Shortest Path Algorithm

The shortest path problem is about finding a path between vertices in a graph such that the total sum of the edges weights is minimum.

Algorithm for Shortest Path

1. Bellman Algorithm
2. Dijkstra Algorithm
3. Floyd Warshall Algorithm

Bellman Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

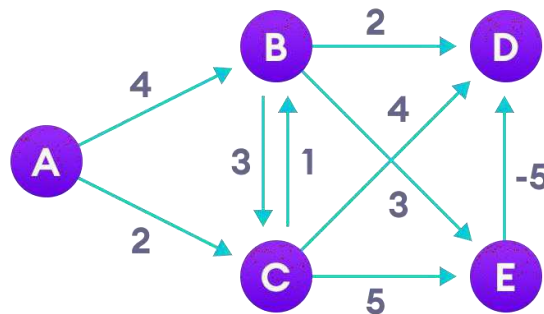
It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

How Bellman Ford's algorithm works

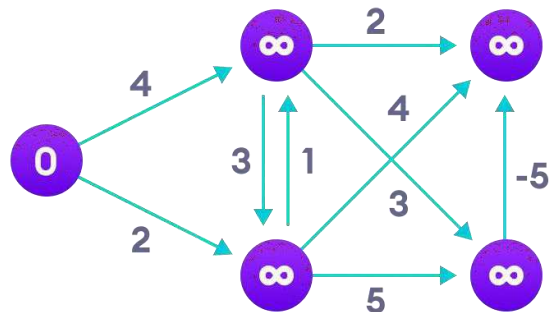
Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

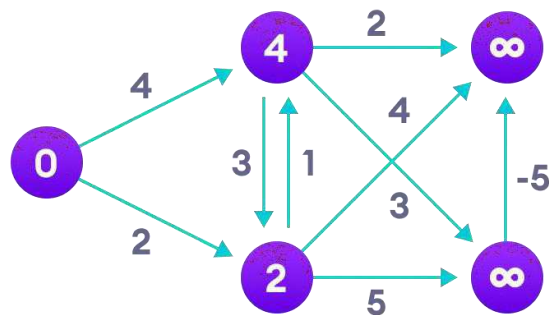
Step 1: Start with the weighted graph



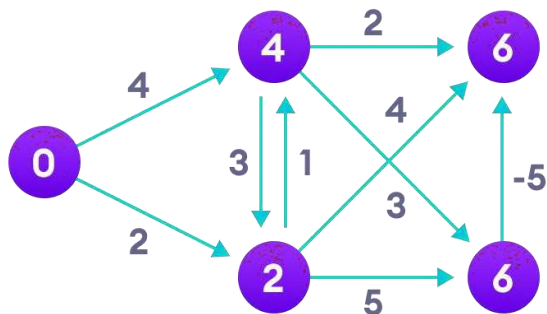
Step 2: Choose a starting vertex and assign infinity path values to all other vertices



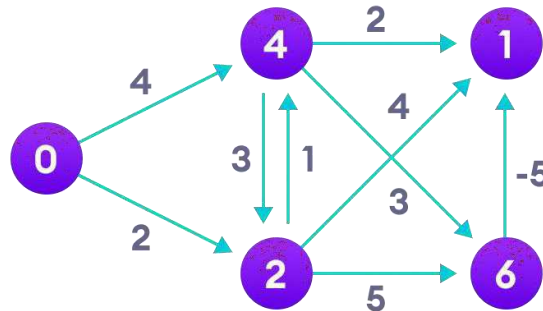
Step 3: Visit each edge and relax the path distances if they are inaccurate



Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step 5: Notice how the vertex at the top right corner had its path length adjusted



Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Bellman Ford Pseudo code

We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.


```

function bellmanFord(G, S) for each vertex V in G distance[V] <- infinite
previous[V] <- NULL distance[S] <- 0
for each vertex V in G for each edge (U,V) in G
tempDistance <- distance[U] + edge_weight(U, V) if tempDistance < distance[V]
distance[V] <- tempDistance previous[V] <- U

```

for each edge (U,V) in G

If $\text{distance}[U] + \text{edge_weight}(U, V) < \text{distance}[V]$

Error: Negative Cycle Exists return distance[], previous[] Bellman Ford's Complexity Time Complexity

Best Case Complexity	$O(E)$
Average Case Complexity	$O(VE)$
Worst Case Complexity	$O(VE)$

Dijkstra Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath B → D of the shortest path A → D between vertices A and D is also the shortest path between vertices B and D.



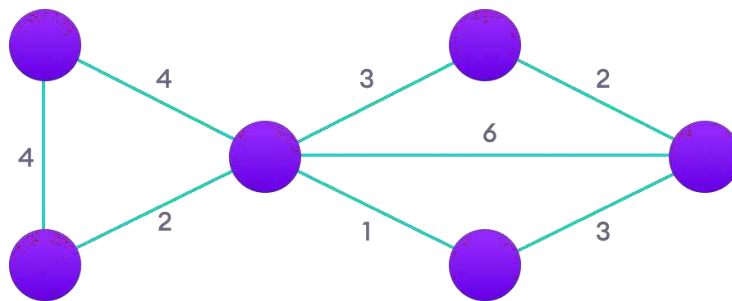
Each subpath is the shortest path

Dijkstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

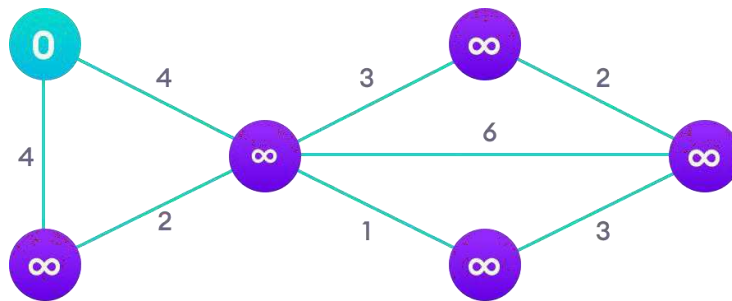
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



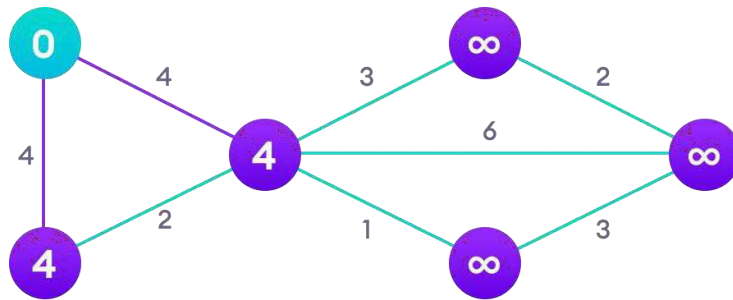
Step: 1

Start with a weighted graph



Step: 2

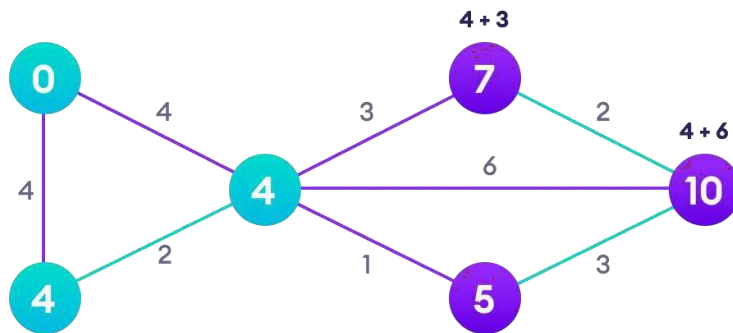
Choose a starting vertex and assign infinity path values to all other devices



Step: 3

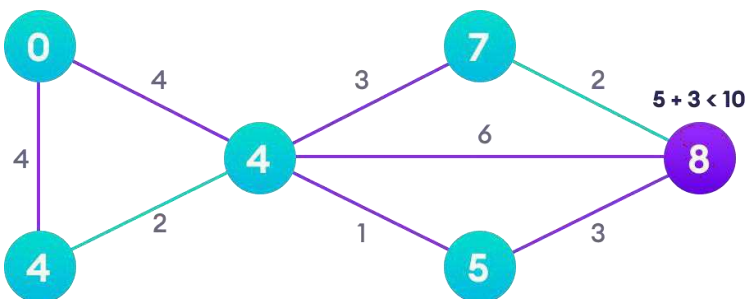
Go to each vertex and update its path length

If the path length of the adjacent vertex is lesser than new path length, don't update it



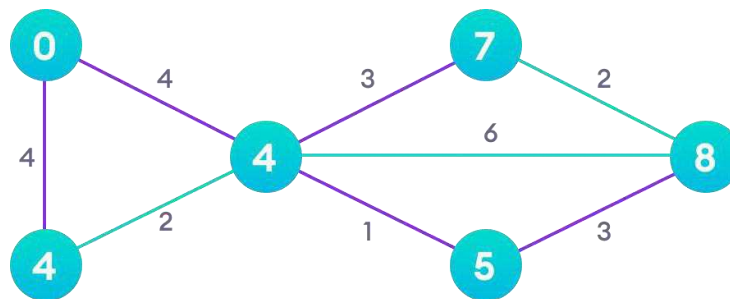
Step: 5

Avoid updating path lengths of already visited vertices



Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 8

Notice how the rightmost vertex has its path length updated twice

Repeat until all the vertices have been visited

Dijkstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
  for each vertex V in G distance[V] <- infinite previous[V] <- NULL
```

```
  If V != S, add V to Priority Queue Q distance[S] <- 0
```

```
  while Q IS NOT EMPTY
```

```
    U <- Extract MIN from Q
```

```
    for each unvisited neighbour V of U
```

```
      tempDistance <- distance[U] + edge_weight(U, V) if tempDistance < distance[V]
```

```
distance[V] <- tempDistance
previous[V] <- U
return distance[], previous[]
```

Dijkstra's Algorithm Complexity

Time Complexity: $O(E \log V)$

where, E is the number of edges and V is the number of vertices. Space Complexity: $O(V)$

Floyd Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

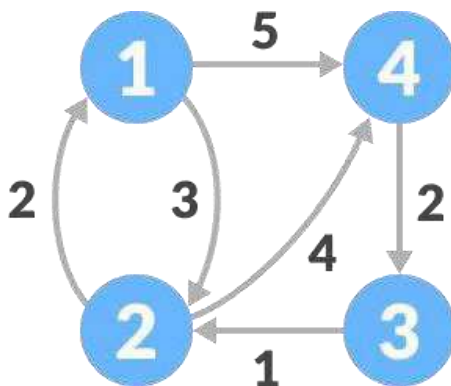
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Initial graph

Follow the steps below to find the shortest path between all the pairs of vertices.

Create a matrix A^0 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i th vertex to the j th vertex. If there is no path from i th vertex to j th vertex, the cell is left as infinity.

$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

Fill each cell with the distance between i th and j th vertex

Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

through this vertex k . Calculate the distance from the source vertex to destination vertex through this vertex k

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7.

Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step

$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 \\ 4 & \infty & \infty & 2 \end{bmatrix} \end{matrix}$$

2. Calculate the distance from the source vertex to destination vertex through this vertex 2

Similarly, A3 and A4 is also created.

$$A^3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 \\ 4 & 5 & 3 & 2 \end{bmatrix} \end{matrix}$$

Calculate the distance from the source vertex to destination vertex through this

$$A^4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 \\ 4 & 5 & 3 & 2 \end{bmatrix} \end{matrix}$$

vertex Calculate the distance from the source vertex to destination vertex through this vertex 4

A4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

n = no of vertices

A = matrix of dimension $n \times n$ for $k = 1$ to n

for $i = 1$ to n for $j = 1$ to n

$A_k[i, j] = \min (A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j])$ return A

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Network Flow

Flow Network is a directed graph that is used for modeling material Flow. There are two different vertices; one is a source which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modelled using flow networks.

Definition: A Flow Network is a directed graph $G = (V, E)$ such that

For each edge $(u, v) \in E$, we associate a nonnegative weight capacity $c(u, v) \geq 0$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$.

There are two distinguishing points, the source s , and the sink t ;

For every vertex $v \in V$, there is a path from s to t containing v .

Let $G = (V, E)$ be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function $f: V \times V \rightarrow \mathbb{R}$ such that the following properties hold:

Play Video

Capacity Constraint: For all $u, v \in V$, we need $f(u, v) \leq c(u, v)$.

Skew Symmetry: For all $u, v \in V$, we need $f(u, v) = -f(v, u)$.

Flow Conservation: For all $u \in V - \{s, t\}$, we need

$$\sum_{v \in V} f(u, v) = \sum_{u \in V} f(u, v) = 0$$

The quantity $f(u, v)$, which can be positive or negative, is known as the net flow from vertex u to vertex v . In the maximum-flow problem, we are given a flow network G with source s and sink t , and

a flow of maximum value from s to t . Ford-Fulkerson Algorithm

Initially, the flow of value is 0. Find some augmenting Path p and increase flow f on each edge of p by residual Capacity $cf(p)$. When no augmenting path exists, flow f is a maximum flow.

FORD-FULKERSON METHOD (G, s, t)

Initialize flow f to 0

while there exists an augmenting path p

do argument flow f along p

Return f

FORD-FULKERSON (G, s, t)

for each edge $(u, v) \in E[G]$

do $f[u, v] \leftarrow 0$

3. $f[u, v] \leftarrow 0$

while there exists a path p from s to t in the residual network G_f .

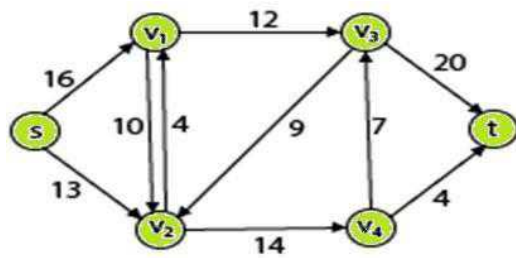
do $cf(p) \leftarrow \min\{C_f(u, v) : (u, v) \text{ is on } p\}$

for each edge (u, v) in p

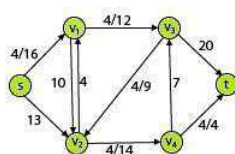
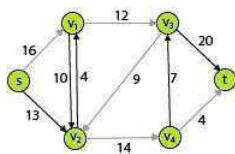
do $f[u, v] \leftarrow f[u, v] + cf(p)$

8. $f[u, v] \leftarrow -f[u, v]$

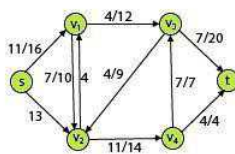
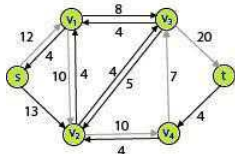
Example: Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.



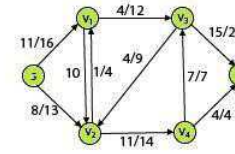
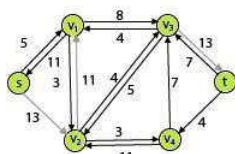
Solution: The left side of each part shows the residual network G_f with a shaded augmenting path p , and the right side of each part shows the net flow f .



(a)

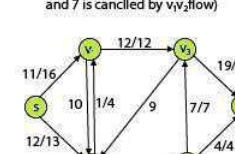
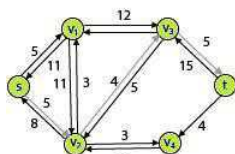


(b)

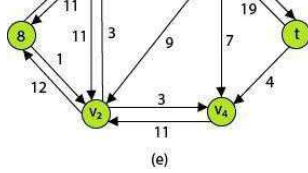


(c)

(In this, 8 is break into 7 and 1 and 7 is cancelled by v_1v_2 flow)



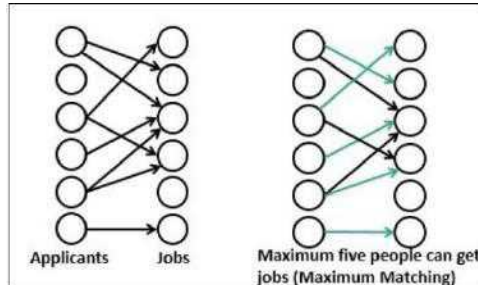
(d)



Now, it has no augmenting paths. So, the maximum flow shown in (d) is 23 is a maximum flow.

Maximum Bipartite Matching

The bipartite matching is a set of edges in a graph is chosen in such a way, that no two edges in that set will share an endpoint. The maximum matching is matching the maximum number of edges.



When the maximum match is found, we cannot add another edge. If one edge is added to the maximum matched graph, it is no longer a matching. For a bipartite graph, there can be more than one maximum matching is possible.

Algorithm

bipartiteMatch(u, visited, assign)

Input: Starting node, visited list to keep track, assign the list to assign node with another node.

Output – Returns true when a matching for vertex u is possible.

Begin

for all vertex v, which are adjacent with u, do if v is not visited, then

mark v as visited

if v is not assigned, or bipartiteMatch(assign[v], visited, assign) is true, then assign[v] := u

return true done

return false End

maxMatch(graph) Input – The given graph.

Output – The maximum number of the match.

Begin

initially no vertex is assigned count := 0

for all applicant u in M , do make all node as unvisited

if bipartiteMatch(u , visited, assign), then increase count by 1

done End

UNIT III ALGORITHM DESIGN TECHNIQUES

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. Greedy Technique: Elements of the greedy strategy - Activity-selection problem — Optimal Merge pattern — Huffman Trees.

Divide and Conquer Algorithm

A divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems solving the sub-problems, and combining them to get the desired output.

To use the divide and conquer algorithm, recursion is used.

How Divide and Conquer Algorithms Work?

Here are the steps involved:

Divide: Divide the given problem into sub-problems using recursion.

Conquer: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.

Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Finding Maximum and Minimum

To find the maximum and minimum numbers in a given array `numbers[]` of size `n`, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

Algorithm: Max-Min-Element (`numbers[]`) `max` := `numbers[1]`

`min` := `numbers[1]` for `i = 2` to `n` do

if `numbers[i] > max` then `max` := `numbers[i]`

if `numbers[i] < min` then `min` := `numbers[i]`

return (`max`, `min`)

Analysis

The number of comparison in Naive method is $2n - 2$.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y-x+1$, where y is greater than or equal to x .

Max-Min(x, y) will return the maximum and minimum values of an array numbers[$x \dots y$].

Algorithm: Max - Min(x, y)

if $y - x \leq 1$ then

return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y])) else

(max1, min1):= maxmin($x, \lfloor ((x + y)/2) \rfloor$)

(max2, min2):= maxmin($\lceil ((x + y)/2) \rceil, y$) return (max(max1, max2), min(min1, min2))

Analysis
Let $T(n)$ be the number of comparisons made by Max-Min(x, y), where the number of elements $n = y - x + 1$.

If $T(n)$ represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2 & \text{for } n > 2 \\ 1 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases}$$

Let us assume that n is in the form of power of 2. Hence, $n = 2^k$ where k is height of the recursion tree.

So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2. (2.T(\frac{n}{4}) + 2) + 2 \dots = \frac{3n}{2} - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented

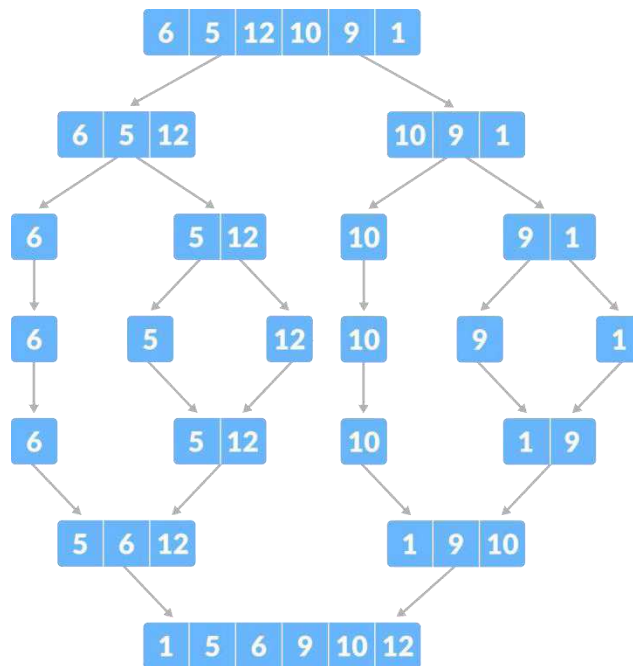
by $O(n)$.

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Merge Sort example



Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each sub problem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. $p == r$.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

MergeSort(A, p, r): if $p > r$

return

$q = (p+r)/2$ mergeSort(A, p, q) mergeSort(A, q+1, r) merge(A, p, q, r)

void merge(int arr[], int p, int q, int r)

{

// Create $L \leftarrow A[p..q]$ and $M \leftarrow A[q+1..r]$ int $n1 = q - p + 1$;

int $n2 = r - q$;

int L[n1], M[n2];

for (int $i = 0$; $i < n1$; $i++$) $L[i] = arr[p + i]$;

for (int $j = 0$; $j < n2$; $j++$) $M[j] = arr[q + 1 + j]$;

// Maintain current index of sub-arrays and main array int i, j, k ;

$i = 0$;

$j = 0$;

$k = p$;

// Until we reach either end of either L or M, pick larger among

// elements L and M and place them in the correct position at $A[p..r]$ while ($i < n1 \ \&\& \ j < n2$)

{

if ($L[i] \leq M[j]$)

{

$arr[k] = L[i]$;

}

else

{ $k++$;

}

// When we run out of elements in either L or M,

// pick up the remaining elements and put in $A[p..r]$ while ($i < n1$)


```

{
arr[k] = L[i]; i++;
k++;
}
while (j < n2)
{
arr[k]=M[j];
j++;
k++;
}
}

```

Time Complexity

Best Case Complexity: $O(n \cdot \log n)$

Worst Case Complexity: $O(n \cdot \log n)$

Average Case Complexity: $O(n \cdot \log n)$

Dynamic Programming -Matrix Chain Multiplication

Dynamic programming is a method for solving optimization problems.

It is algorithm technique to solve a complex and overlapping sub-problems. Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.

Dynamic programming is more efficient than other algorithm methods like as Greedy method, Divide and Conquer method, Recursion method, etc....

The real time many of problems are not solve using simple and traditional approach methods. like as coin change problem , knapsack problem, Fibonacci sequence generating , complex matrix multiplication....To solve using Iterative formula, tedious method , repetition again and again it become a more time consuming and foolish. some of the problem it should be necessary to divide a sub problems and compute its again and again to solve a

such kind of problems and give the optimal solution , effective solution the Dynamic programming is needed...

Basic Features of Dynamic programming:-

- Get all the possible solution and pick up best and optimal solution.
- Work on principal of optimality.
- Define sub-parts and solve them using recursively.
- Less space complexity But more Time complexity.

- Dynamic programming saves us from having to re compute previously calculated sub- solutions.
- Difficult to understanding.

We are covered a many of the real world problems. In our day to day life when we do making coin change, robotics world, aircraft, mathematical problems like Fibonacci sequence, simple matrix multiplication of more than two matrices and its multiplication possibility is many more so in that get the best and optimal solution. NOW we can look about one problem that is MATRIX CHAIN MULTIPLICATION PROBLEM.

Suppose, We are given a sequence (chain) (A_1, A_2, \dots, A_n) of n matrices to be multiplied, and we wish to compute the product $(A_1 A_2 \dots A_n)$. We can evaluate the above expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.

Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is (A_1, A_2, A_3, A_4) then we can fully parenthesize the product $(A_1 A_2 A_3 A_4)$ in five distinct ways:

1:- $(A_1(A_2(A_3 A_4)))$,

2:- $(A_1((A_2 A_3) A_4))$,

3:- $((A_1 A_2)(A_3 A_4))$,

4:- $((A_1(A_2 A_3)) A_4)$,

5:- $((A_1 A_2) A_3) A_4$.

We can multiply two matrices A and B only if they are compatible. the number of columns of A must equal the number of rows of B . If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications is pqr . we shall express costs in terms of the number of scalar multiplications. For example, if we have three matrices (A_1, A_2, A_3) and its cost is $(10 \times 100), (100 \times 5), (5 \times 500)$ respectively. so we can calculate the cost of scalar multiplication is $10 * 100 * 5 = 5000$ if $((A_1 A_2) A_3)$, $10 * 5 * 500 = 25000$ if $(A_1 (A_2 A_3))$, and so on cost

calculation. Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. that is here is minimum cost is 5000 for above example .So problem is we can perform a many time of cost multiplication and repeatedly the calculation is performing. So this general method is very time consuming and tedious. So we can apply dynamic programming for solve this kind of problem.

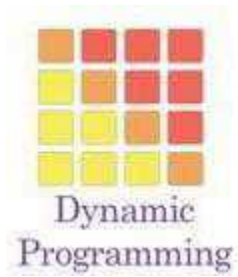
when we used the Dynamic programming technique we shall follow some steps.

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution.

Construct an optimal solution from computed information.



we have matrices of any of order. our goal is find optimal cost multiplication of matrices. when we solve this kind of problem using DP step 2 we can get

$m[i, j] = \min \{ m[i, k] + m[k+1, j] + p_{i-1} * p_k * p_j \}$ if $i < j$ where p is dimension of matrix, $i \leq k < j$

The basic algorithm of matrix chain multiplication:-

// Matrix $A[i]$ has dimension $\text{dims}[i-1] \times \text{dims}[i]$ for $i = 1..n$

MatrixChainMultiplication(int dims[])

{

// length[dims] = n + 1

n = dims.length - 1;

// $m[i, j]$ = Minimum number of scalar multiplications (i.e., cost)

// needed to compute the matrix $A[i]A[i+1]...A[j] = A[i..j]$

// The cost is zero when multiplying one matrix

for (i = 1; i <= n; i++) m[i, i] = 0;

for (len = 2; len <= n; len++){

// Subsequence lengths

for (i = 1; i <= n - len + 1; i++) { j = i + len - 1;

m[i, j] = MAXINT;

for (k = i; k <= j - 1; k++) {

cost = m[i, k] + m[k+1, j] + dims[i-1]*dims[k]*dims[j];

if (cost < m[i, j]) { m[i, j] = cost;

s[i, j] = k;

// Index of the subsequence split that achieved minimal cost

}

}

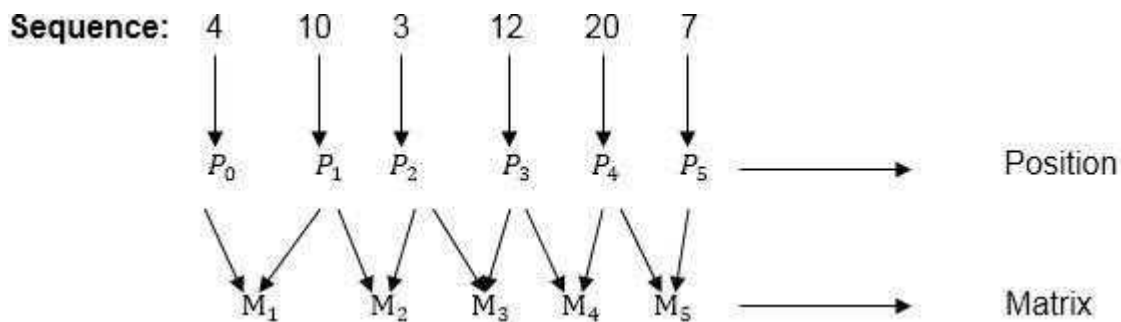
}
}
}

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute $M[i, j]$, $0 \leq i, j \leq 5$. We know $M[i, i] = 0$ for all i .

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



In Dynamic Programming, initialization of every method done by '0'. So we initialize it by '0'. It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

$$1. m(1, 2) = m_1 \times m_2$$

$$= 4 \times 10 \times 10 \times 3$$

$$= 4 \times 10 \times 3 = 120$$

$$2. m(2, 3) = m_2 \times m_3$$

$$= 10 \times 3 \times 3 \times 12$$

$$= 10 \times 3 \times 12 = 360$$

$$3. m(3, 4) = m_3 \times m_4$$

$$= 3 \times 12 \times 12 \times 20$$

$$= 3 \times 12 \times 20 = 720$$

$$4. m(4,5) = m_4 \times m_5$$

$$= 12 \times 20 \times 20 \times 7$$

$$= 12 \times 20 \times 7 = 1680$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

We initialize the diagonal element with equal i,j value with '0'.

After that second diagonal is sorted out and we get all the values corresponded to it Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

$$M[1, 3] = M_1 M_2 M_3$$

There are two cases by which we can solve this multiplication: $(M_1 \times M_2) + M_3$, $M_1 + (M_2 \times M_3)$

After solving both cases we choose the case in which minimum output is there.

$$M[1, 3] = \min \left\{ \begin{array}{l} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{array} \right\}$$

$$M[1, 3] = 264$$

As Comparing both output 264 is minimum in both cases so we insert 264 in table and $(M_1 \times M_2) + M_3$ this combination is chosen for the output making.

$$M[2, 4] = M_2 M_3 M_4$$

There are two cases by which we can solve this multiplication: $(M_2 \times M_3) + M_4$, $M_2 + (M_3 \times M_4)$

After solving both cases we choose the case in which minimum output is there.

$$M[2, 4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

$$M[2, 4] = 1320$$

As Comparing both output 1320 is minimum in both cases so we insert 1320 in table and $M_2+(M_3 \times M_4)$ this combination is chosen for the output making.

$$M[3, 5] = M_3 M_4 M_5$$

There are two cases by which we can solve this multiplication: $(M_3 \times M_4) + M_5$, $M_3+(M_4 \times M_5)$

After solving both cases we choose the case in which minimum output is there.

$$M[3, 5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

$$M[3, 5] = 1140$$

As Comparing both output 1140 is minimum in both cases so we insert 1140 in table and $(M_3 \times M_4) + M_5$ this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:

$$M[1, 4] = M_1 M_2 M_3 M_4$$

There are three cases by which we can solve this multiplication:

$$(M_1 \times M_2 \times M_3) M_4$$

$$M_1 \times (M_2 \times M_3 \times M_4) \quad 3. \quad (M_1 \times M_2) \times (M_3 \times M_4)$$

After solving these cases we choose the case in which minimum output is there

$$M[1, 4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

$$M[1, 4] = 1080$$

As comparing the output of different cases then '1080' is minimum output, so we insert 1080 in the table and (M1 x M2) x (M3 x M4) combination is taken out in output making,

$$M[2, 5] = M2 \ M3 \ M4 \ M5$$

There are three cases by which we can solve this multiplication:

$$(M2 \times M3 \times M4) \times M5$$

$$M2 \times (M3 \times M4 \times M5)$$

$$3. (M2 \times M3) \times (M4 \times M5)$$

After solving these cases we choose the case in which minimum output is there

$$M[2, 5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

$$M[2, 5] = 1350$$

As comparing the output of different cases then '1350' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4xM5)combination is taken out in output making.

1	2	3	4	5		1	2	3	4	5	
0	120	264			1	0	120	264	1080		1
	0	360	1320		2		0	360	1320	1350	2
		0	720	1140	3			0	720	1140	3
			0	1680	4				0	1680	4
				0	5					0	5

Now Product of 5 matrices:

$$M[1, 5] = M1 \ M2 \ M3 \ M4 \ M5$$

There are five cases by which we can solve this multiplication:

$$(M1 \times M2 \times M3 \times M4) \times M5$$

$$M1 \times (M2 \times M3 \times M4 \times M5)$$

$$(M1 \times M2 \times M3) \times M4 \times M5$$

$$M1 \times M2 \times (M3 \times M4 \times M5)$$

After solving these cases we choose the case in which minimum output is there

$$M[1, 5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

$M[1, 5] = 1344$

As comparing the output of different cases then '1344' is minimum output, so we insert 1344 in the table and $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$ combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

So we can get the optimal solution of matrices multiplication....

Multi Stage Graph

Multistage Graph problem is defined as follow:

Multistage graph $G = (V, E, W)$ is a weighted directed graph in which vertices are partitioned into $k \geq 2$ disjoint sub sets $V = \{V_1, V_2, \dots, V_k\}$ such that if edge (u, v) is present in E then $u \in V_i$ and $v \in V_{i+1}$, $1 \leq i \leq k$. The goal of multistage graph problem is to find minimum cost path from source to destination vertex.

The input to the algorithm is a k-stage graph, n vertices are indexed in increasing order of stages.

The algorithm operates in the backward direction, i.e. it starts from the last vertex of the graph and proceeds in a backward direction to find minimum cost path.

Minimum cost of vertex $j \in V_i$ from vertex $r \in V_{i+1}$ is defined as, $\text{Cost}[j] = \min\{ c[j, r] + \text{cost}[r] \}$

where, $c[j, r]$ is the weight of edge $\langle j, r \rangle$ and $\text{cost}[r]$ is the cost of moving from end vertex to vertex r .

Algorithm for the multistage graph is described below :

Algorithm for Multistage Graph Algorithm MULTI_STAGE(G, k, n, p)

// Description: Solve multi-stage problem using dynamic programming

// Input:

k: Number of stages in graph $G = (V, E)$ $c[i, j]$: Cost of edge (i, j)

// Output: $p[1:k]$: Minimum cost path $\text{cost}[n] \leftarrow 0$

for $j \leftarrow n - 1$ to 1 do

// Let r be a vertex such that $(j, r) \in E$ and $c[j, r] + \text{cost}[r]$ is minimum $\text{cost}[j] \leftarrow c[j, r] + \text{cost}[r]$

$\pi[j] \leftarrow r$

end

// Find minimum cost path $p[1] \leftarrow 1$

$p[k] \leftarrow n$

for $j \leftarrow 2$ to $k - 1$ do

$p[j] \leftarrow \pi[p[j - 1]]$

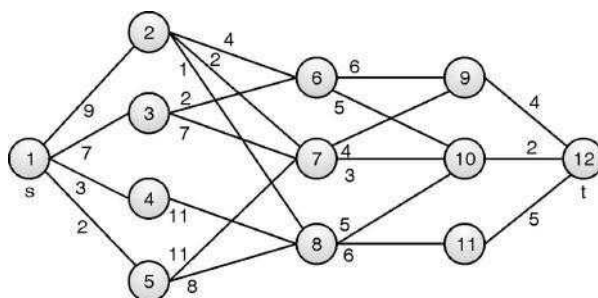
end

Complexity Analysis of Multistage Graph

If graph G has $|E|$ edges, then cost computation time would be $O(n + |E|)$. The complexity of tracing the minimum cost path would be $O(k)$, $k < n$. Thus total time complexity of multistage graph using dynamic programming would be $O(n + |E|)$.

Example

Example: Find minimum path cost between vertex s and t for following multistage graph using dynamic programming.



Solution:

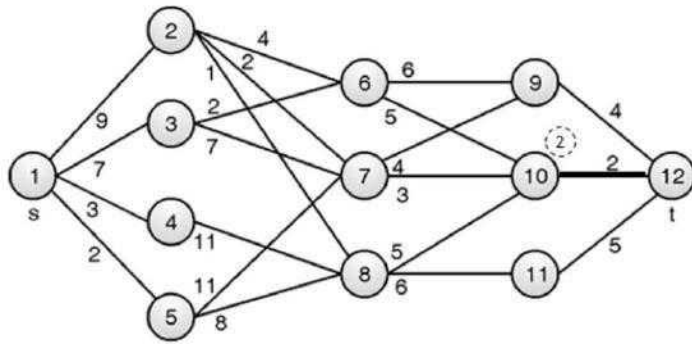
Solution to multistage graph using dynamic programming is constructed as, $\text{Cost}[j] = \min\{c[j, r] + \text{cost}[r]\}$

Here, number of stages $k = 5$, number of vertices $n = 12$, source $s = 1$ and target $t = 12$ Initialization:

$\text{Cost}[n] = 0 \Rightarrow \text{Cost}[12] = 0$. $p[1] = s \Rightarrow p[1] = 1$

$p[k] = t \Rightarrow p[5] = 12$. $r = t = 12$.

Stage 4:



Stage 3:

Vertex 6 is connected to vertices 9 and 10:

$$\text{Cost}[6] = \min\{ c[6, 10] + \text{Cost}[10], c[6, 9] + \text{Cost}[9] \}$$

$$= \min\{5 + 2, 6 + 4\} = \min\{7, 10\} = 7$$

$$p[6] = 10$$

Vertex 7 is connected to vertices 9 and 10:

$$\text{Cost}[7] = \min\{ c[7, 10] + \text{Cost}[10], c[7, 9] + \text{Cost}[9] \}$$

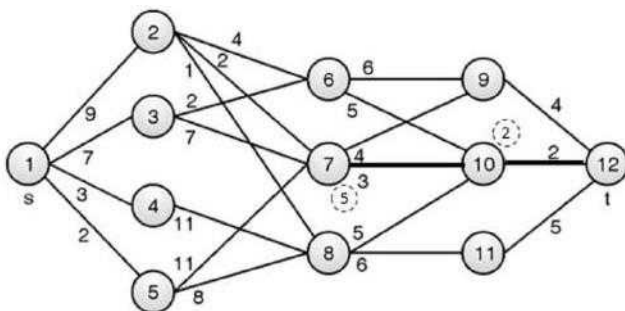
$$= \min\{3 + 2, 4 + 4\} = \min\{5, 8\} = 5$$

$$p[7] = 10$$

Vertex 8 is connected to vertex 10 and 11:

$$\text{Cost}[8] = \min\{ c[8, 11] + \text{Cost}[11], c[8, 10] + \text{Cost}[10] \}$$

$$= \min\{6 + 5, 5 + 2\} = \min\{11, 7\} = 7 \quad p[8] = 10$$



Stage 2:

Vertex 2 is connected to vertices 6, 7 and 8:

$$\text{Cost}[2] = \min\{ c[2, 6] + \text{Cost}[6], c[2, 7] + \text{Cost}[7], c[2, 8] + \text{Cost}[8] \}$$

$$= \min\{4 + 7, 2 + 5, 1 + 7\} = \min\{11, 7, 8\} = 7$$

$$p[2] = 7$$

Vertex 3 is connected to vertices 6 and 7:

$$\text{Cost}[3] = \min\{ c[3, 6] + \text{Cost}[6], c[3, 7] + \text{Cost}[7] \}$$

$$= \min\{ 2 + 7, 7 + 5 \} = \min\{ 9, 12 \} = 9$$

$$p[3] = 6$$

Vertex 4 is connected to vertex 8:

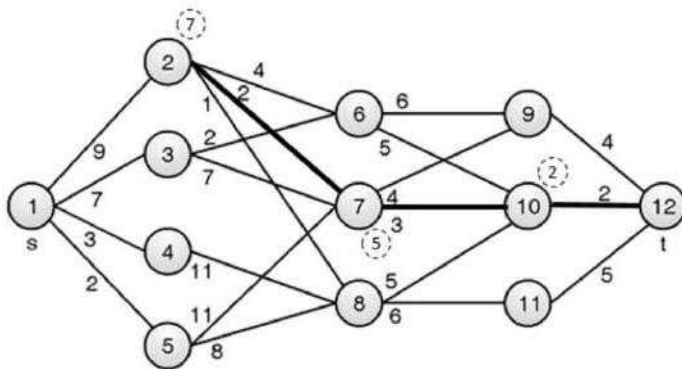
$$\text{Cost}[4] = c[4, 8] + \text{Cost}[8] = 11 + 7 = 18$$

$$p[4] = 8$$

Vertex 5 is connected to vertices 7 and 8:

$$\text{Cost}[5] = \min\{ c[5, 7] + \text{Cost}[7], c[5, 8] + \text{Cost}[8] \}$$

$$= \min\{ 11 + 5, 8 + 7 \} = \min\{ 16, 15 \} = 15 \quad p[5] = 8$$



Stage 1:

Vertex 1 is connected to vertices 2, 3, 4 and 5:

$$\text{Cost}[1] = \min\{ c[1, 2] + \text{Cost}[2], c[1, 3] + \text{Cost}[3], c[1, 4] + \text{Cost}[4], c[1, 5] + \text{Cost}[5] \}$$

$$= \min\{ 9 + 7, 7 + 9, 3 + 18, 2 + 15 \}$$

$$= \min\{ 16, 16, 21, 17 \} = 16 \quad p[1] = 2$$

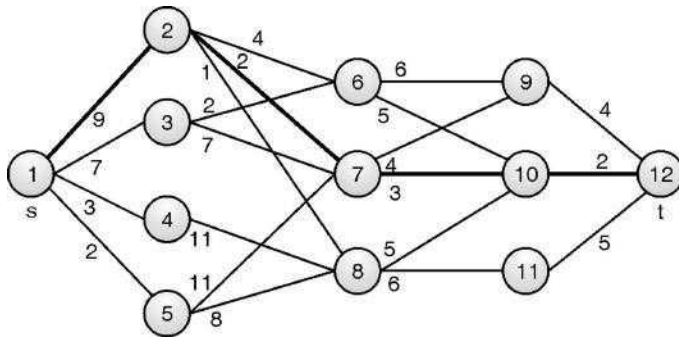
Trace the solution:

$$p[1] = 2$$

$$p[2] = 7$$

$$p[7] = 10$$

d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]
2	7	6	8	8	10	10	10	12	12	12	12



$p[10] = 12$

Minimum cost path is : 1 – 2 – 7 – 10 – 12

Cost of the path is : $9 + 2 + 3 + 2 = 16$

Optimal Binary Search Tree

Optimal Binary Search Tree extends the concept of Binary search tree. Binary Search Tree (BST) is a nonlinear data structure which is used in many scientific applications for reducing the search time. In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.

Optimal Binary Search Tree (OBST) is very useful in dictionary search. The probability of searching is different for different words. OBST has great application in translation.

If we translate the book from English to German, equivalent words are searched from English to German dictionary and replaced in translation. Words are searched same as in binary search tree order.

Binary search tree simply arranges the words in lexicographical order. Words like ‘the’, ‘is’, ‘there’ are very frequent words, whereas words like ‘xylophone’, ‘anthropology’ etc. appears rarely.

It is not a wise idea to keep less frequent words near root in binary search tree. Instead of storing words in binary search tree in lexicographical order, we shall arrange them according to their probabilities. This arrangement facilitates few searches for frequent words as they would be near the root. Such tree is called Optimal Binary Search Tree.

Consider the sequence of n keys $K = \langle k_1, k_2, k_3, \dots, k_n \rangle$ of distinct probability in sorted order such that

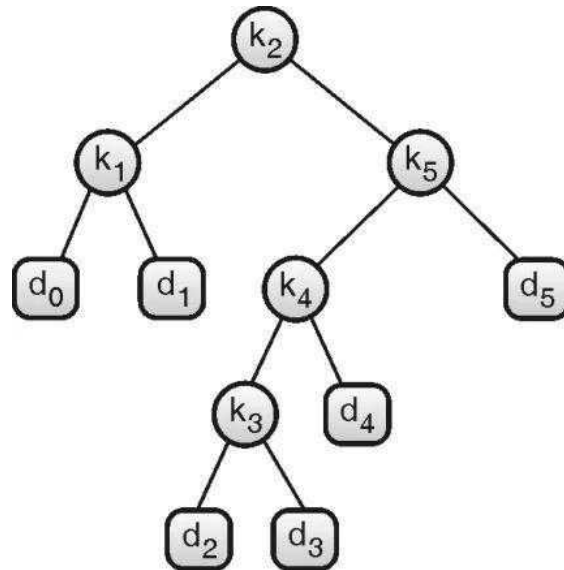
$k_1 < k_2 < \dots < k_n$. Words between each pair of key lead to unsuccessful search, so for n keys, binary search tree contains $n + 1$ dummy keys d_i , representing unsuccessful searches.

Two different representation of BST with same five keys $\{k_1, k_2, k_3, k_4, k_5\}$ probability is shown in following figure

With n nodes, there exist $(2n)! / ((n + 1)! * n!)$ different binary search trees. An exhaustive search for optimal binary search tree leads to huge amount of time.

The goal is to construct a tree which minimizes the total search cost. Such tree is called optimal binary search tree. OBST does not claim minimum height. It is also not necessary that parent of sub tree has higher priority than its child.

Dynamic programming can help us to find such optima tree.



Binary search trees with 5 keys

Mathematical formulation

We formulate the OBST with following observations

Any sub tree in OBST contains keys in sorted order $k_i \dots k_j$, where $1 \leq i \leq j \leq n$.

Sub tree containing keys $k_i \dots k_j$ has leaves with dummy keys $d_{i-1} \dots d_j$.

Suppose k_r is the root of sub tree containing keys $k_i \dots k_j$. So, left sub tree of root k_r contains keys

$k_i \dots k_{r-1}$ and right sub tree contain keys k_{r+1} to k_j . Recursively, optimal sub trees are constructed from the left and right sub trees of k_r .

Let $e[i, j]$ represents the expected cost of searching OBST. With n keys, our aim is to find and minimize $e[1, n]$.

Base case occurs when $j = i - 1$, because we just have the dummy key d_{i-1} for this case. Expected search cost for this case would be $e[i, j] = e[i, i - 1] = q_{i-1}$.

For the case $j \geq i$, we have to select any key k_r from $k_i \dots k_j$ as a root of the tree.

With k_r as a root key and sub tree $k_i \dots k_j$, sum of probability is defined as

$$w(i, j) = \sum_{m=i}^j p_m + \sum_{m=i-1}^j q_m$$

(Actual key starts at index 1 and dummy key starts at index 0)

Thus, a recursive formula for forming the OBST is stated below :

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \end{cases}$$

$e[i, j]$ gives the expected cost in the optimal binary search tree.

Algorithm for Optimal Binary Search Tree

The algorithm for optimal binary search tree is specified below :

Algorithm OBST(p, q, n)

```
// e[1...n+1, 0...n] : Optimal sub tree
// w[1...n+1, 0...n] : Sum of probability
// root[1...n, 1...n] : Used to construct OBST

for i ← 1 to n + 1 do
    e[i, i - 1] ← qi - 1
    w[i, i - 1] ← qi - 1
end

for m ← 1 to n do
    for i ← 1 to n - m + 1 do
        j ← i + m - 1
        e[i, j] ← ∞
        w[i, j] ← w[i, j - 1] + pj + qj
        for r ← i to j do
            t ← e[i, r - 1] + e[r + 1, j] + w[i, j]
            if t < e[i, j] then e[i, j] ← t
            root[i, j] ← r
        end
    end
end

return (e, root)
```

Complexity Analysis of Optimal Binary Search Tree

It is very simple to derive the complexity of this approach from the above algorithm. It uses three nested loops. Statements in the innermost loop run in $O(1)$ time. The running time of the algorithm is computed as

$$\begin{aligned}
 T(n) &= \sum_{m=1}^n \sum_{i=1}^{n-m+1} \sum_{j=i}^{n-l+1} \Theta(1) \\
 &= \sum_{m=1}^n \sum_{i=1}^{n-m+1} n = \sum_{m=1}^n n^2 \\
 &= \Theta(n^3)
 \end{aligned}$$

Thus, the OBST algorithm runs in cubic time

Example

Problem: Let $p(1 : 3) = (0.5, 0.1, 0.05)$ $q(0 : 3) = (0.15, 0.1, 0.05, 0.05)$ Compute and construct OBST for above values using Dynamic approach. Solution:

Here, given that

i	0	1	2	3
pi		0.5	0.1	0.05
qi	0.15	0.1	0.05	0.05

Recursive formula to solve OBST problem is

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \end{cases}$$

Where,

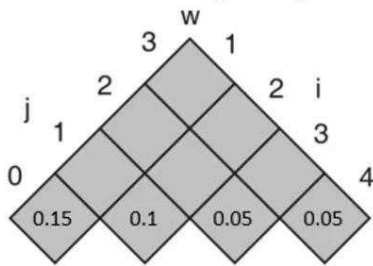
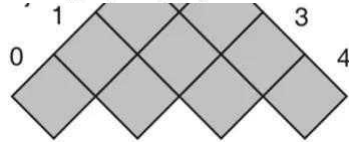
$$w(i, j) = \sum_{m=i}^j p_m + \sum_{m=i-1}^j q_m$$

Initially,

$$v \quad w[2, 1] = \sum_{m=2}^1 p_m + \sum_{m=1}^1 q_m = q_1 = 0.1 = 0.15$$

$$w[3, 2] = \sum_{m=3}^2 p_m + \sum_{m=2}^2 q_m = q_2 = 0.05$$

$$w[4, 3] = \sum_{m=4}^3 p_m + \sum_{m=3}^3 q_m = q_3 = 0.05$$



$$w[1, 1] = \sum_{m=1}^1 p_m + \sum_{m=0}^1 q_m = p_1 + q_0 + q_1$$

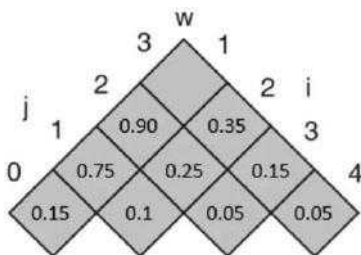
$$= 0.5 + 0.25 = 0.75$$

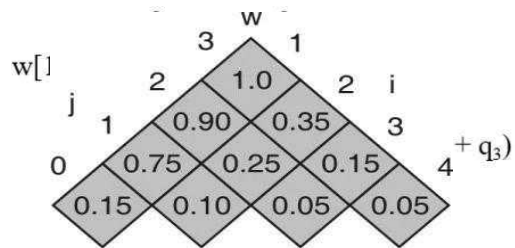
$$w[2, 2] = \sum_{m=2}^2 p_m + \sum_{m=1}^2 q_m = p_2 + q_1 + q_2$$

$$= 0.1 + 0.15 = 0.25$$

$$w[3, 3] = \sum_{m=3}^3 p_m + \sum_{m=2}^3 q_m = p_3 + q_2 + q_3$$

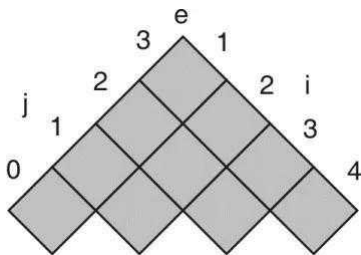
$$= \underbrace{0.05} + \underbrace{0.10} = \underbrace{0.15}$$





Now, we will compute $e[i, j]$

Initially,

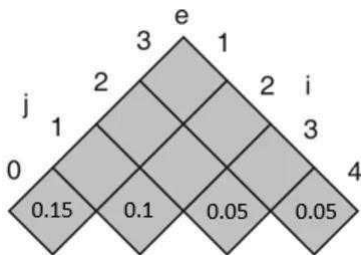


$$e[1, 0] = q_0 = 0.15 \quad (\because j = i - 1)$$

$$e[2, 1] = q_1 = 0.1 \quad (\because j = i - 1)$$

$$e[3, 2] = q_2 = 0.05 \quad (\because j = i - 1)$$

$$e[4, 3] = q_3 = 0.05 \quad (\because j = i - 1)$$



$$e[1, 1] = \min \{ e[1, 0] + e[2, 1] + w(1, 1) \}$$

$$= \min \{ 0.15 + 0.1 + 0.75 \} = 1.0$$

$$e[2, 2] = \min \{ e[2, 1] + e[3, 2] + w(2, 2) \}$$

$$= \min \{ 0.1 + 0.05 + 0.25 \} = 0.4$$

$$e[3, 3] = \min \{ e[3, 2] + e[4, 3] + w(3, 3) \}$$

$$= \min \{ 0.05 + 0.05 + 0.15 \} = 0.25$$

$$e[1, 2] = \min \left\{ \begin{array}{l} e[1, 0] + e[2, 2] + w(1, 2) \\ e[1, 1] + e[3, 2] + w(1, 2) \end{array} \right\}$$

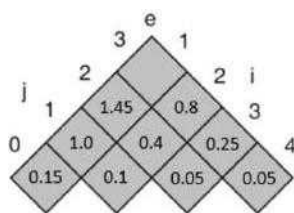
$$= \min \left\{ \begin{array}{l} 0.15 + 0.4 + 0.90 \\ 1.0 + 0.05 + 0.90 \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} 1.45 \\ 1.95 \end{array} \right\} = 1.45$$

$$e[2, 3] = \min \left\{ \begin{array}{l} e[2, 1] + e[3, 3] + w(2, 3) \\ e[2, 2] + e[4, 3] + w(2, 3) \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} 0.1 + 0.25 + 0.35 \\ 0.4 + 0.05 + 0.35 \end{array} \right\}$$

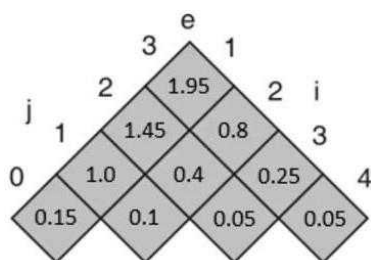
$$= \min \left\{ \begin{array}{l} 0.90 \\ 0.80 \end{array} \right\} = 0.8$$



$$e[1, 3] = \min \left\{ \begin{array}{l} e[1, 0] + e[2, 3] + w(1, 3) \\ e[1, 1] + e[3, 3] + w(1, 3) \\ e[1, 2] + e[4, 3] + w(1, 3) \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} 0.15 + 0.80 + 1.0 \\ 1.0 + 0.25 + 1.0 \\ 1.45 + 0.05 + 1.0 \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} 1.95 \\ 2.25 \\ 2.5 \end{array} \right\} = 1.95$$



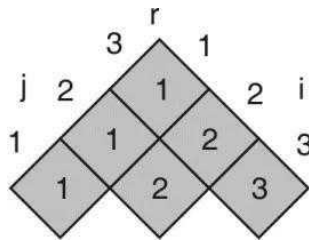
$e[1, 3]$ is minimum for $r = 1$, so $r[1, 3] = 1$

$e[2, 3]$ is minimum for $r = 2$, so $r[2, 3] = 2$

$e[1, 2]$ is minimum for $r = 1$, so $r[1, 2] = 1$

$e[3, 3]$ is minimum for $r = 3$, so $r[3, 3] = 3$

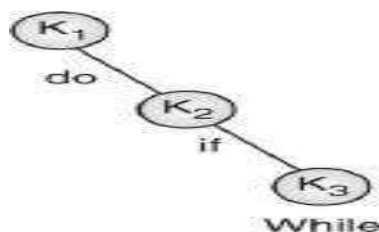
$e[2, 2]$ is minimum for $r = 2$, so $r[2, 2] = 2$



$e[1, 1]$ is minimum for $r = 1$, so $r[1, 1] = 1$. Let us now construct OBST for given data. $r[1, 3] = 1$, so k_1 will be at the root. k_2, \dots, k_3 are on right side of k_1 .

$r[2, 3] = 2$, So k_2 will be the root of this sub tree. k_3 will be on the right of k_2 .

Thus, finally, we get.



Greedy Technique

Activity Selection problem is an approach of selecting non-conflicting tasks based on start and end time and can be solved in $O(N \log N)$ time using a simple greedy approach. Modifications of this problem are complex and interesting which we will explore as well. Surprisingly, if we use a Dynamic Programming approach, the time complexity will be $O(N^3)$ that is lower performance.

The problem statement for Activity Selection is that "Given a set of n activities with their start and finish times, we need to select maximum number of non-conflicting activities that can be performed by a single person, given that the person can handle only one activity at a time." The Activity Selection problem follows Greedy approach i.e. at every step, we can make a choice that looks best at the moment to get the optimal solution of the complete problem.

Our objective is to complete maximum number of activities. So, choosing the activity which is going to finish first will leave us maximum time to adjust the later activities. This is the intuition that greedily choosing the activity with earliest finish time will give us an optimal solution. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution, so we choose the activity which finishes first. If we sort elements based on their starting time, the activity with least starting time could take the maximum duration for completion, therefore we won't be able to maximise number of activities.

Algorithm

The algorithm of Activity Selection is as follows: Activity-Selection(Activity, start, finish)

Sort Activity by finish times stored in

finishSelected = {Activity[1]}

n =

Activity.length j =

1

for i = 2 to n:

if start[i] \geq finish[j]:

Selected = Selected \cup

{Activity[i]} j = i

return Selected

Complexity

Time Complexity:

When activities are sorted by their finish time: **$O(N)$**

When activities are not sorted by their finish time, the time complexity is **$O(N \log N)$** due to complexity of sorting

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[4] >= END[2], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[5] < END[4], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[6] >= END[4], SELECTED

0 1 2 3 4 5 6

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[1] >= END[0], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[2] < END[1], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[3] < END[1], REJECTED

In this example, we take the start and finish time of activities as follows:

start = [1, 3, 2, 0, 5, 8, 11]

finish = [3, 4, 5, 7, 9, 10, 12]

Sorted by their finish time, the activity 0 gets selected. As the activity 1 has starting time which is equal to the finish time of activity 0, it gets selected. Activities 2 and 3 have smaller starting time than finish time of activity 1, so they get rejected. Based on similar comparisons, activities 4 and 6 also get selected, whereas activity 5 gets rejected. In this example, in all the activities 0, 1, 4 and 6 get selected, while others get rejected.

Optimal Merge Pattern

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a p-record file and a q-record file requires possibly $p + q$ record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files $\{f_1, f_2, f_3, \dots, f_n\}$. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE

(n) for $i := 1$ to $n - 1$

 dodeclare new node

 node.leftchild := least (list)

 node.rightchild := least (list)

 node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)

 insert (list, node);

return least (list);

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

$$M_1 = \text{merge } f_1 \text{ and } f_2 \Rightarrow 20 + 30 = 50$$

$$M_2 = \text{merge } M_1 \text{ and } f_3 \Rightarrow 50 + 10 = 60$$

$$M_3 = \text{merge } M_2 \text{ and } f_4 \Rightarrow 60 + 5 = 65$$

$$M_4 = \text{merge } M_3 \text{ and } f_5 \Rightarrow 65 + 30 = 95$$

Hence, the total number of operations is

$$50 + 60 + 65 + 95 = 270$$

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence –

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

$M_1 = \text{merge } f_4 \text{ and } f_3 \Rightarrow 5 + 10 = 15$

$M_2 = \text{merge } M_1 \text{ and } f_1 \Rightarrow 15 + 20 =$

35 $M_3 = \text{merge } M_2 \text{ and } f_2 \Rightarrow 35 + 30$

$= 65$ $M_4 = \text{merge } M_3 \text{ and } f_5 \Rightarrow 65 +$

$30 = 95$

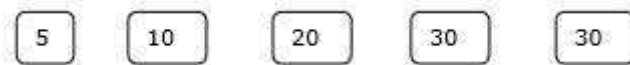
Therefore, the total number of operations is

$15 + 35 + 65 + 95 = 210$

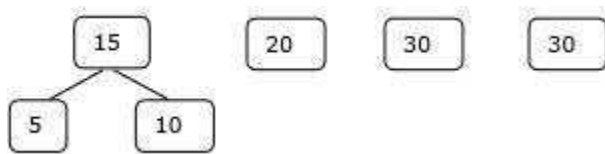
Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

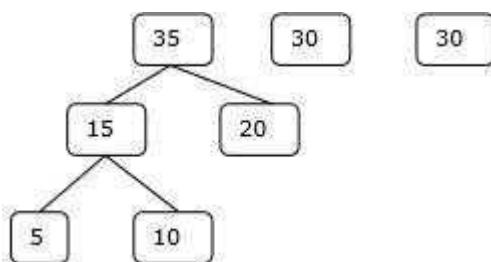
Initial Set



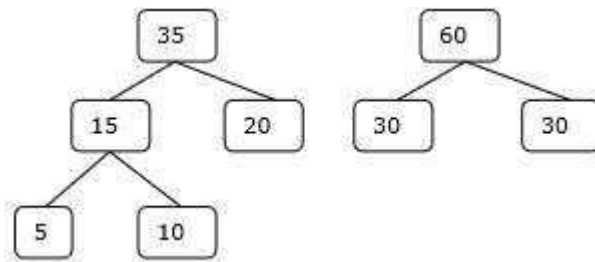
Step 1



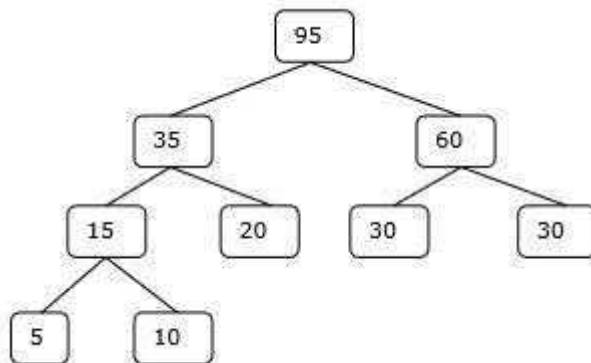
Step 2



Step 3



Step 4



Hence, the solution takes $15 + 35 + 60 + 95 = 205$ number of comparisons.

Huffman Tree

Huffman coding provides codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character. Huffman codes are of variable-length, and without any prefix (that means no code is a prefix of any other). Any prefix-free binary code can be displayed or visualized as a binary tree with the encoded characters stored at the leaves.

Huffman tree or Huffman coding tree defines as a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet.

The Huffman tree is treated as the binary tree associated with minimum external path weight that means, the one associated with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to construct a tree with the minimum external path weight.

An example is given

below- Letter frequency

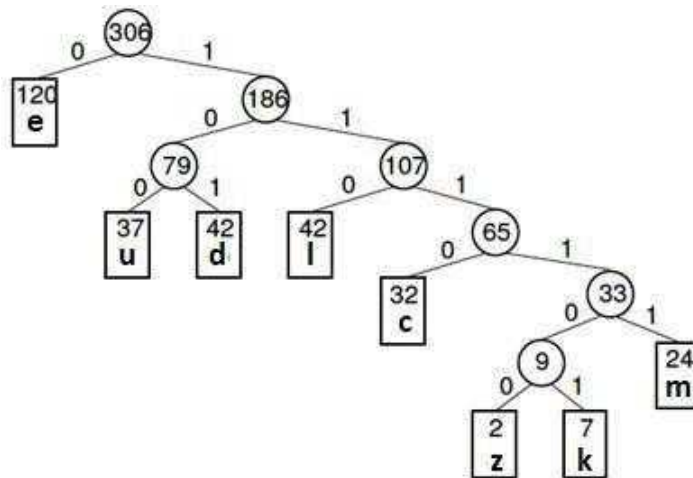
table

Letter	z	k	m	c	u	d	l	e
--------	---	---	---	---	---	---	---	---

Frequency	2	7	24	32	37	42	42	120
------------------	---	---	----	----	----	----	----	-----

Huffman code

Letter	Freq	Code	Bits
e	120	0	1
d	42	101	3
l	42	110	3
u	37	100	3
c	32	1110	4
m	24	11111	5
k	7	111101	6
z	2	111100	6



The Huffman tree (for the above example) is

given below -Algorithm Huffman (c)

```
{
  n= |c|
  Q = c
  for i<-1 to n-1
  do
  {
    temp <- get node ()
    left (temp) Get_min (Q) right [temp]

    Get Min (Q) a = left [temp] b = right
    [temp]

    F

    [temp]

    <- f[a]

    + [b]
```

```
    insert  
  
    (Q,  
  
    temp)  
  
    }  
    return Get_min (0)  
    }
```

UNIT IV STATE SPACE SEARCH ALGORITHMS

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem – Graph colouring problem
Branch and Bound: Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

Backtracking

N queen Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for $n = 1$, the problem has a trivial solution, and no solution exists for $n = 2$ and $n = 3$. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen " i " on row " i ."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen ' q_3 ' safely. So we backtrack one step and place the queen ' q_2 ' in (2, 4), the next best possible solution. Then we obtain the position for placing ' q_3 ' which is (3, 2). But later this position also leads to a dead end, and no place is found where ' q_4 ' can be placed safely. Then we have to backtrack till ' q_1 ' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

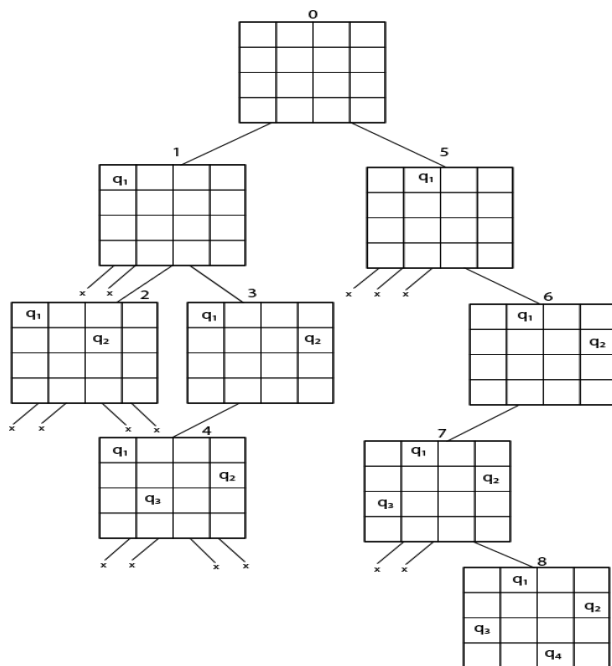
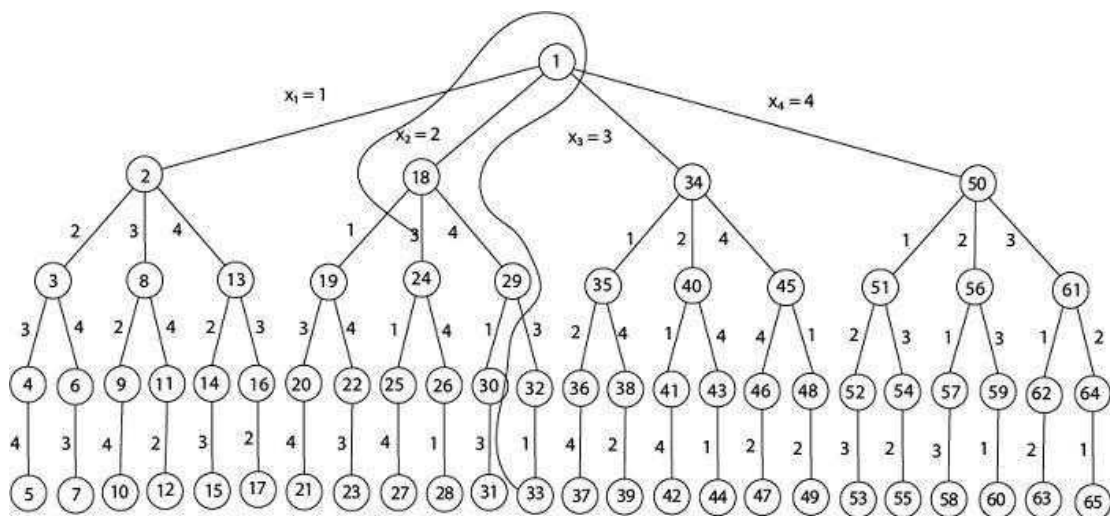


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen " q_i " is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q ₁				
2						q ₂		
3							q ₃	
4		q ₄						
5						q ₅		
6	q ₆							
7			q ₇					
8					q ₈			

Thus, the solution for 8 -queen problem for (4, 6, 8, 2, 7, 1, 3, 5).

If two queens are placed at position (i, j) and (k, l).

Then they are on same diagonal only if $(i - j) = k - l$ or $i + j = k + l$.

The first equation implies that $j - l = i - k$.

The second equation implies that $j - l = k - i$.

Therefore, two queens lie on the duplicate diagonal if and only if $|j-l|=|i-k|$

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2, \dots, x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to the n- queens problem.

Place (k, i)

{

For j ← 1 to k - 1

do if $(x[j] = i)$

or $(\text{Abs } x[j] - i) = (\text{Abs } (j - k))$

then return false;

return true;

}

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false. $x[]$ is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

N - Queens (k, n).

{

For i ← 1 to n

do if Place (k, i) then5. {

$x[k] \leftarrow i;$

if $(k == n)$ then

write $(x[1] \dots \dots \dots n);$

else

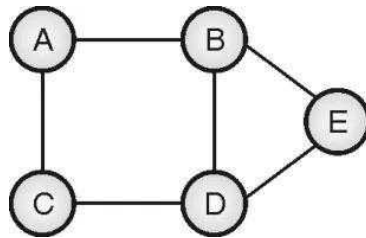
N - Queens (k + 1, n); } }

Hamiltonian Circuit

The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges

- **The Hamiltonian cycle problem** is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.

- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.



The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, “Given a path, is it a Hamiltonian cycle of the graph?”.

1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).

Vertex i must not appear in the first (i – 1) vertices of any path.

With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Algorithm

HAMILTONIAN (i)

// Description : Solve Hamiltonian cycle problem using backtracking.

// Input : Undirected, connected graph $G = \langle V, E \rangle$ and initial vertex i

// Output : Hamiltonian cycle

if

FEASIBLE(i)

then

if

(i == n - 1)

then

Print V[0... n - 1]

else

j ← 2

while(j

≤ n) **do**

```

V[i] ← j HAMILTONIAN(i + 1)
j ← j + 1 end
end

```

```

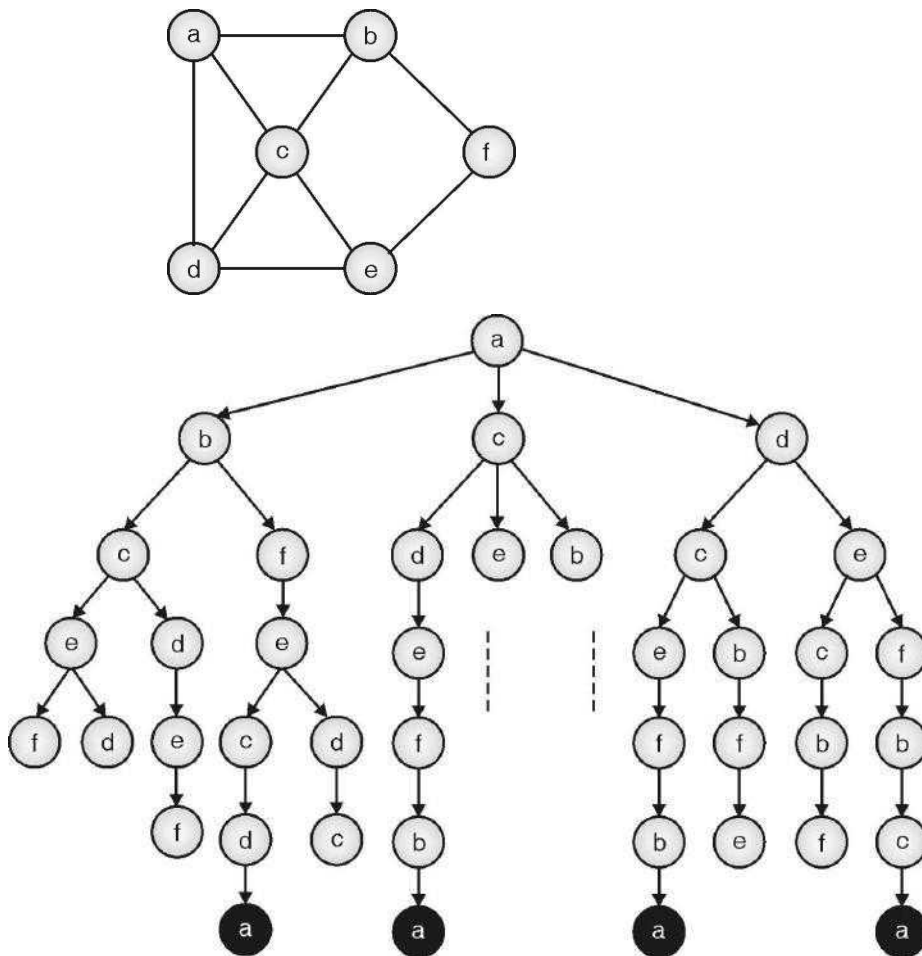
end
function
FEASIBLE(i)
flag ← 1
for
j ← 1 to i - 1
doif
Adjacent(Vi, Vj)
then
flag ← 0
endend if
Adjacent (Vi, Vi-1)
then
flag ← 1
else
flag ← 0
end return flag

```

Complexity Analysis

Looking at the state space graph, in worst case, total number of nodes in tree would be, $T(n) = 1 + (n - 1) + (n - 1)^2 + (n - 1)^3 + \dots + (n - 1)^{n-1}$
 $= \frac{(n-1)^n - 1}{n-2}$
 $T(n) = O(n^n)$. Thus, the Hamiltonian cycle algorithm runs in exponential time.

Example: Find the Hamiltonian cycle by using the backtracking approach for a given graph.



The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure (g) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all possible paths, the concept is self-explanatory. It is not possible to include all the paths in the graph, so few of the successful and unsuccessful paths are traced in the graph. Black nodes indicate the Hamiltonian cycle.

Subset Sum Problem

Sum of Subsets Problem: Given a set of positive integers, find the combination of numbers that sum to given value M.

Sum of subsets problem is analogous to the **knapsack problem**. The Knapsack Problem tries to fill the knapsack using a given set of items to maximize the profit. Items are selected in such a way that the total weight in the knapsack does not exceed the capacity of the knapsack. The inequality condition in the knapsack problem is replaced by equality in the sum of subsets problem.

Given the set of n positive integers, $W = \{w_1, w_2, \dots, w_n\}$, and given a positive integer M, the sum of the subset problem can be formulated as follows (where w_i and M correspond to item weights and knapsack capacity in the knapsack problem):

$$\sum_{i=1}^n w_i x_i = M$$

Where,

$$x_i \in \{0, 1\}$$

Numbers are sorted in ascending order, such that $w_1 < w_2 < w_3 < \dots < w_n$. The solution is often represented using the solution vector X. If the i th item is included, set x_i to 1 else set it to 0. In each iteration, one item is tested. If the inclusion of an item does not violate the constraint of the problem, add it. Otherwise, backtrack, remove the previously added item, and continue the same procedure for all remaining items. The solution is easily described by the state space tree. Each left edge denotes the inclusion of w_i and the right edge denotes the exclusion of w_i . Any path from the root to the leaf forms a subset. A state-space tree for $n = 3$ is demonstrated in Fig. (a).

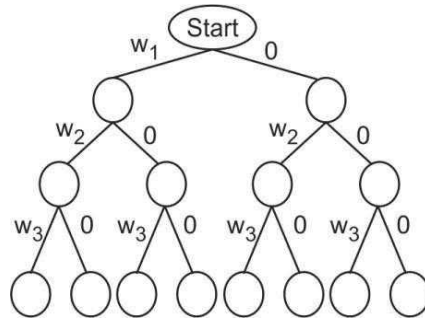


Fig. (a): State space tree for $n = 3$

Algorithm for Sum of subsets

The algorithm for solving the sum of subsets problem using recursion is stated below:

```

Algorithm sumofsubsets(s,k,r)
{
  X[k]:=1;
  if (s+w[k]=m) then write (x[1:k]); // subset found
  else
    if (s+w[k]+w[k+1]<=m) then
      sumofsubsets(s+w[k],k+1,r-w[k]);

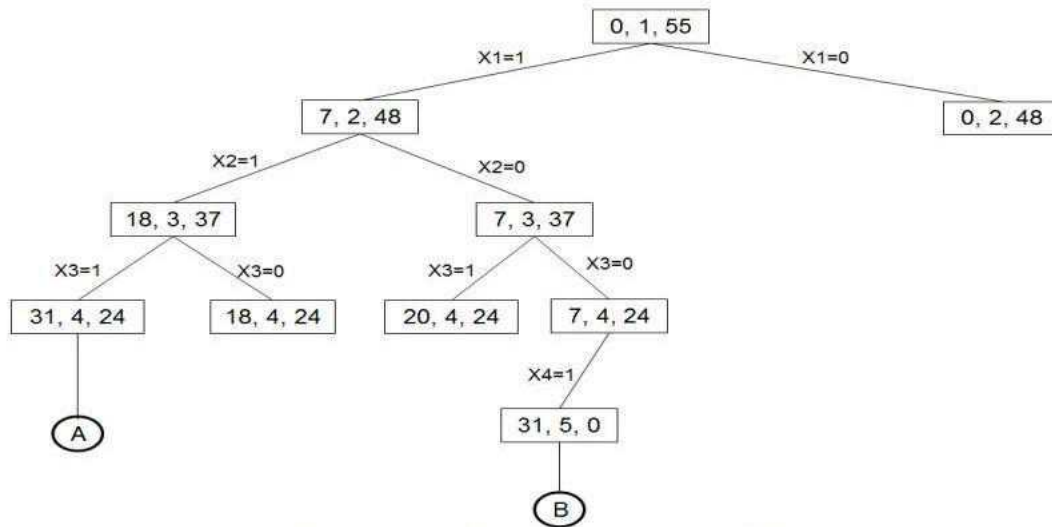
  //generate right child and evaluate Bk.

  if ((s+r-w[k]>=m) and (s+w[k+1]<=m)) then
  {
    X[k]:=0;
    sumofsubsets(s,k+1,r-w[k]);
  }
}

```

Examples

Ex) $n=4$, $(w_1, w_2, w_3, w_4) = (7, 11, 13, 24)$ and $m=31$
 Solution Vector = $(x[1], x[2], x[3], x[4])$



Portion of state space Tree

Solution A = $\{1, 1, 1, 0\}$
 Solution B = $\{1, 0, 0, 1\}$

Graph Colouring

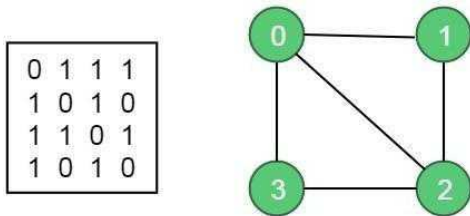
In this problem, an undirected graph is given. There is also provided m colors. The problem is to find if it is possible to assign nodes with m different colors, such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing the same color.

Input and

Output

The adjacency matrix of a graph $G(V, E)$ and an integer m , which indicates the maximum number of colors that can be used.



Let the maximum color $m =$

3. Output:

This algorithm will return which node will be assigned with which color. If the solution is not possible, it will return false.

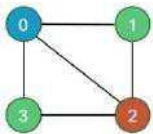
For this input the assigned colors are:

Node 0 -> color 1

Node 1 -> color 2

Node 2 -> color 3

Node 3 -> color 2



Algorithm

isValid(vertex, colorList, col)

Input – Vertex, colorList to check, and color, which is trying to assign.

Output – True if the color assigning is valid, otherwise false.

Begin

for all vertices v of the graph, do

if there is an edge between v and i , and $col = colorList[i]$,

then return false

done return

true

End

graphColoring(colors, colorList, vertex)

Input – Most possible colors, the list for which vertices are colored with which color, and the starting vertex.

Output – True, when colors are assigned, otherwise false.

Begin

if all vertices are checked, then

return true

for all colors col from available colors,

do if isValid(vertex, color, col), then

add col to the colorList for vertex

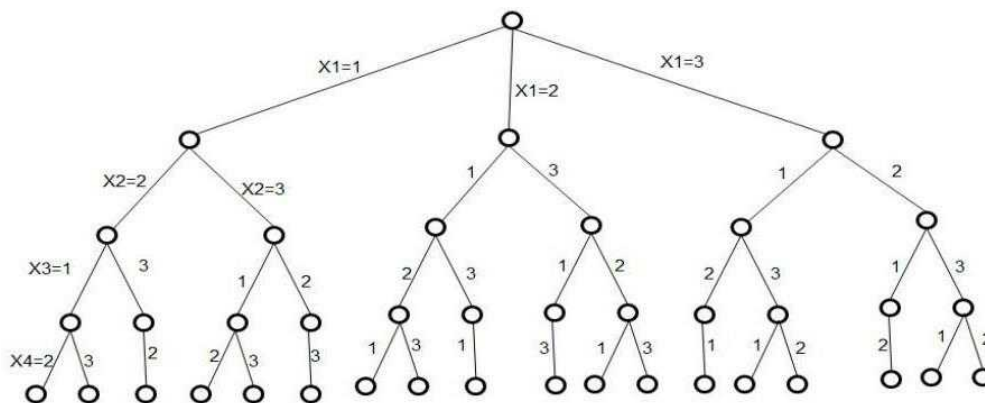
if graphColoring(colors, colorList, vertex+1) = true, then

return true

remove color for vertex

done

return false



A 4-node graph and all possible 3-colorings

End

Branch and Bound

Solving 15 puzzle Problem (LCBB)

The problem consists of 15 numbered (0-15) tiles on a square box with 16 tiles (one tile is blank or empty). The objective of this problem is to change the arrangement of initial node to goal node by using series of legal moves.

The Initial and Goal node arrangement is shown by following figure.

1	2	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial Arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Final Arrangement

In initial node four moves are possible. User can move any one of the tile like 2, or 3, or 5, or 6 to the empty tile. From this we have four possibilities to move from initial node.

The legal moves are for adjacent tile number is left, right, up, down, ones at a time.

Each and every move creates a new arrangement, and this arrangement is called state of puzzle problem. By using different states, a state space tree diagram is created, in which edges are labeled according to the direction in which the empty space moves.

The state space tree is very large because it can be 16! Different arrangements.

In state space tree, nodes are numbered as per the level. In each level we must calculate the value or cost of each node by using given formula:

$$C(x) = f(x) + g(x),$$

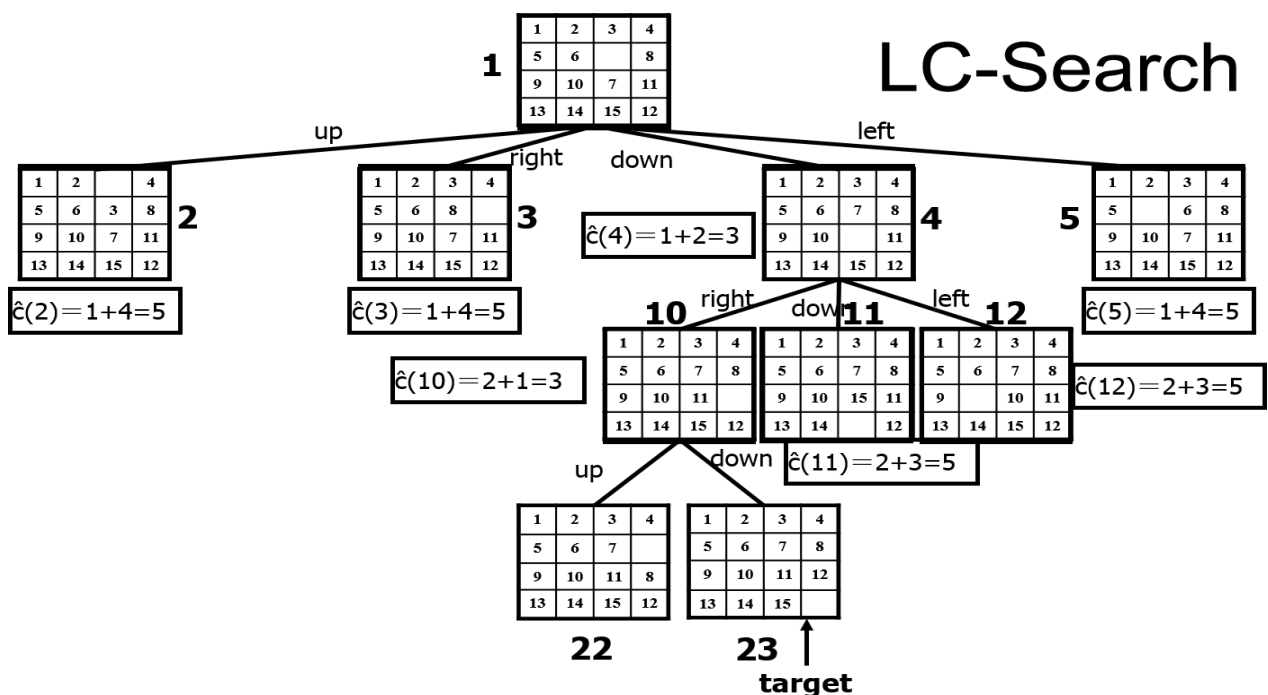
$f(x)$ is length of path from root or initial node to node x ,

$g(x)$ is estimated length of path from x downward to the goal node. Number of non blank tile not in their correct position.

$$C(x) < \text{Infinity. (initially set bound).}$$

Each time node with smallest cost is selected for further expansion towards goal node. This node becomes the e-node.

State Space tree with node cost is shown in diagram.



Assignment Problem

Problem Statement

Let's first define a job assignment problem. In a standard version of a job assignment problem, there can be jobs and workers. To keep it simple, we're taking jobs and workers in our example:

	Job 1	Job 2	Job 3
A	9	3	4
B	7	8	4
C	10	5	2

We can assign any of the available jobs to any worker with the condition that if a job is assigned to a worker, the other workers can't take that particular job. We should also notice that each job has some cost associated with it, and it differs from one worker to another.

Here the main aim is to complete all the jobs by assigning one job to each worker in such a way that the sum of the cost of all the jobs should be minimized.

Branch and Bound Algorithm Pseudocode

Now let's discuss how to solve the job assignment problem using a branch and bound algorithm. Let's see the pseudocode first:

Algorithm 1: Job Assignment Problem Using Branch And Bound

Data: Input cost matrix $M[][][]$

Result: Assignment of jobs to each worker according to optimal cost

Function $MinCost(M[][][])$

while *True* **do**

$E = LeastCost();$

if *E is a leaf node* **then**

$print();$

return;

end

for *each child S of E* **do**

$Add(S);$

$S \rightarrow parent = E;$

end

end

Here, is the input cost matrix that contains information like the number of available jobs, a list of available workers, and the associated cost for each job. The function $MinCost()$ maintains a list of active nodes. The function $Leastcost()$ calculates the minimum cost of the active node at each level of the tree. After finding the node with minimum cost, we remove the node from the list of active nodes and return it.

We're using the $add()$ function in the pseudocode, which calculates the cost of a particular node and adds it to the list of active nodes.

In the search space tree, each node contains some information, such as cost, a total number of jobs, as well as a total number of workers.

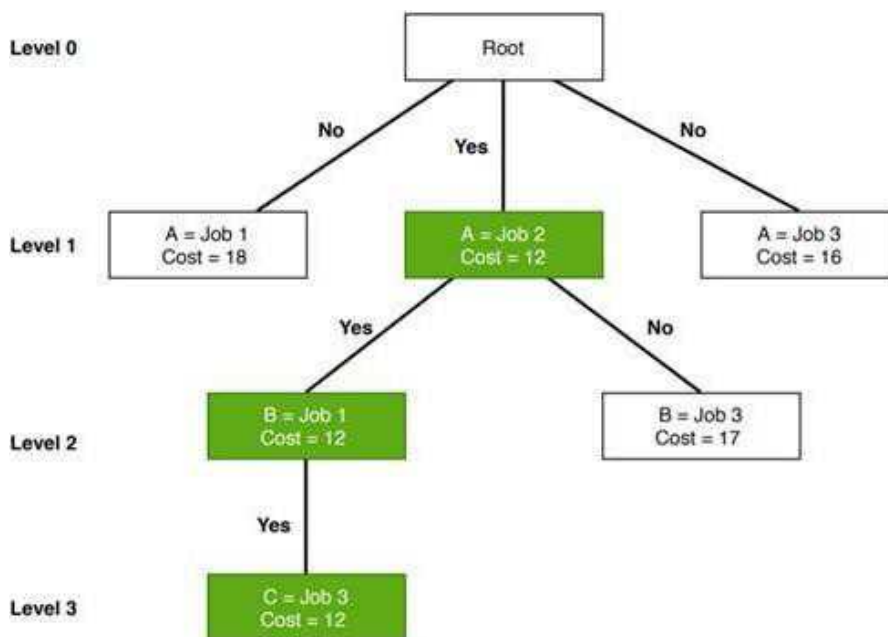
Now let's run the algorithm on the sample example we've created:

Initially, we've 3 jobs available. The worker *A* has the option to take any of the available jobs. So at level 1, we assigned all the available jobs to the worker *A* and calculated the cost. We can see that when we assigned jobs 2 to the worker *A*, it gives the lowest cost in level 1 of the search space tree. **So we assign the job 2 to worker *A* and continue the algorithm. "Yes" indicates that this is currently optimal cost.**

After assigning the job 2 to worker *A*, we still have two open jobs. Let's consider worker *B* now. We're trying to assign either job 1 or 3 to worker *B* to obtain optimal cost.

Either we can assign the job 1 or 3 to worker *B*. **Again we check the cost and assign job 1 to worker *B* as it is the lowest in level 2.**

Finally, we assign the job 3 to worker *C*, and the optimal cost is 12.



Advantages

In a branch and bound algorithm, we don't explore all the nodes in the tree. That's why **the time complexity of the branch and bound algorithm is less when compared with other algorithms.**

If the problem is not large and if we can do the branching in a reasonable amount of time, **it finds an optimal solution for a given problem.**

The branch and bound algorithm find a minimal path to reach the optimal solution for a given problem. **It doesn't repeat nodes while exploring the tree.**

Disadvantages

The branch and bound algorithm are time-consuming. Depending on the size of the given problem, the number of nodes in the tree can be too large in the worst case.

Knapsack Problem using branch and bound

Problem Statement

We are given a set of n objects which have each have a value v_i and a weight w_i . The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is

maximized, and

the sum of weights of the objects does not exceed a given threshold W . An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object.

Consider an example where $n = 4$, and the values are given by $\{10, 12, 12, 18\}$ and the weights given by $\{2, 4, 6, 9\}$. The maximum weight is given by $W = 15$. Here, the solution to the problem will be including the first, third and the fourth objects.

Here, the procedure to solve the problem is as follows are:

- Calculate the cost function and the Upper bound for the two children of each node. Here, the $(i + 1)^{\text{th}}$ level indicates whether the i^{th} object is to be included or not.
- If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than U , then replace the value of U with this value. Then, kill all unexplored nodes which have cost function greater than this value.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
- While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

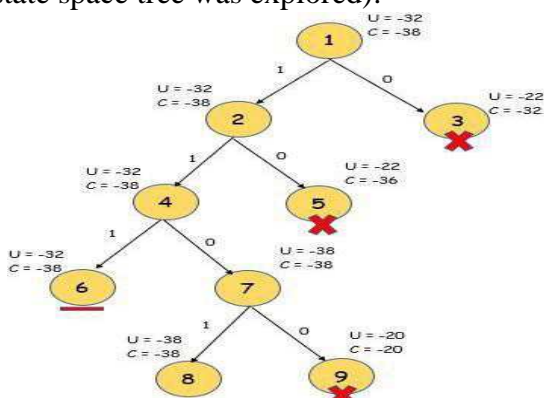
Time and Space Complexity

Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by $O(2^n)$, in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to be explored, and hence its best case time complexity is given by $O(n)$. Since this method requires the creation of the state space tree, its space complexity will also be exponential.

Solving an Example

Consider the problem with $n = 4$, $V = \{10, 10, 12, 18\}$, $w = \{2, 4, 6, 9\}$ and $W = 15$. Here, we calculate the initial upper bound to be $U = 10 + 10 + 12 = 32$. Note that the 4th object cannot be included here, since that would exceed W . For the cost, we add $3/9^{\text{th}}$ of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):



Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

Travelling salesman problem

- Travelling Salesman Problem (TSP) is an interesting problem. Problem is defined as “given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with the constraint of minimizing the travelling distance.”

- TSP has many practical applications. It is used in network design, and transportation route design. The objective is to minimize the distance. We can start tour from any random city and visit other cities in any order. With n cities, n! different permutations are possible. Exploring all paths using brute force attacks may not be useful in real life applications.

LCBB using Static State Space Tree for Travelling Salesman Problem

- Branch and bound is an effective way to find better, if not best, solution in quick time by pruning some of the unnecessary branches of search tree.

- It works as follow:

Consider directed weighted graph $G = (V, E, W)$, where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C, where

C_{ij} = cost of edge, if there is a direct path from city i to city j
 $C_{ij} = \infty$, if there is no direct path from city i to city j.

2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.

3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.

4. Prepare state space tree for the reduce matrix

5. Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.

6. If $\langle i, j \rangle$ edge is to be included, then do following :

- (a) Set all values in row i and all values in column j of A to ∞

- (b) Set $A[j, 1] = \infty$

- (c) Reduce A again, except rows and columns having all ∞ entries.

7. Compute the cost of newly created reduced matrix

as, $\text{Cost} = L + \text{Cost}(i, j) + r$

Where, L is cost of original reduced cost matrix and r is $A[i, j]$.

8. If all nodes are not visited then go to step 4. Reduction procedure is described below

Raw Reduction:

Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow:

$M_{\text{RowRed}} = \{M_{ij} - \min \{M_{ij} \mid 1 \leq j \leq n, \text{ and } M_{ij} < \infty\}\}$ Consider the following distance matrix:

$$M = \begin{array}{|c|c|c|c|c|} \hline \infty & 20 & 30 & 10 & 11 \\ \hline 15 & \infty & 16 & 4 & 2 \\ \hline 3 & 5 & \infty & 2 & 4 \\ \hline 19 & 6 & 18 & \infty & 3 \\ \hline 16 & 4 & 7 & 16 & \infty \\ \hline \end{array}$$

Find the minimum element from each row and subtract it from each cell of matrix.

$$M = \begin{array}{|c|c|c|c|c|} \hline \infty & 20 & 30 & 10 & 11 \\ \hline 15 & \infty & 16 & 4 & 2 \\ \hline 3 & 5 & \infty & 2 & 4 \\ \hline 19 & 6 & 18 & \infty & 3 \\ \hline 16 & 4 & 7 & 16 & \infty \\ \hline \end{array} \begin{array}{l} \rightarrow 10 \\ \rightarrow 2 \\ \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 4 \end{array}$$

Reduced matrix would be:

$$M_{\text{RowRed}} = \begin{array}{|c|c|c|c|c|} \hline \infty & 10 & 20 & 0 & 1 \\ \hline 13 & \infty & 14 & 2 & 0 \\ \hline 1 & 3 & \infty & 0 & 2 \\ \hline 16 & 3 & 15 & \infty & 0 \\ \hline 12 & 0 & 3 & 12 & \infty \\ \hline \end{array}$$

Row reduction cost is the summation of all the values subtracted from each rows: Row reduction cost (M) = 10 + 2 + 2 + 3 + 4 = 21

Column reduction:

Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$$M_{\text{ColRed}} = \{M_{ji} - \min \{M_{ji} \mid 1 \leq j \leq n, \text{ and } M_{ji} < \infty\}\}$$

To reduced above matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

$$M_{\text{RowRed}} = \begin{array}{|c|c|c|c|c|} \hline \infty & 10 & 20 & 0 & 1 \\ \hline 13 & \infty & 14 & 2 & 0 \\ \hline 1 & 3 & \infty & 0 & 2 \\ \hline 16 & 3 & 15 & \infty & 0 \\ \hline 12 & 0 & 3 & 12 & \infty \\ \hline \end{array}$$

$$\begin{array}{c} \downarrow \\ \downarrow \\ \downarrow \\ \downarrow \\ \downarrow \end{array}$$

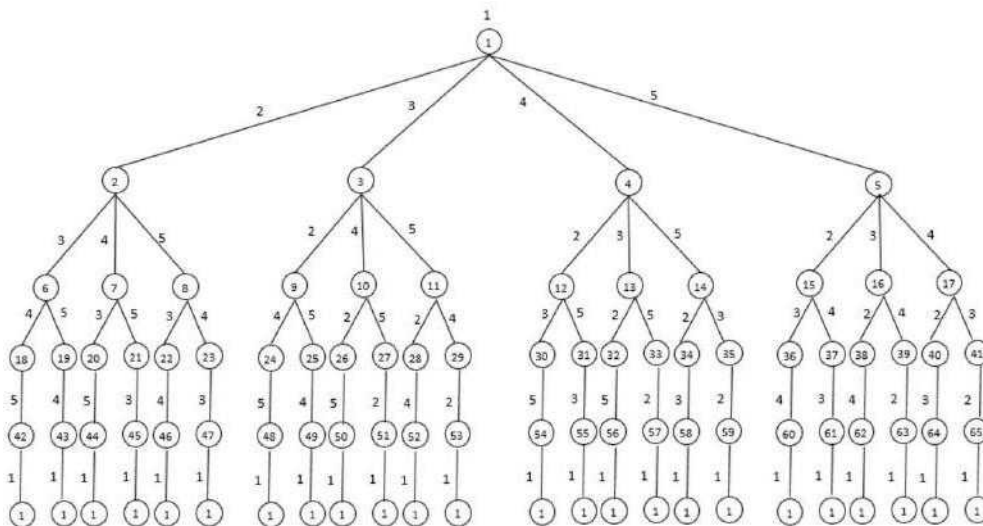
$$\begin{array}{|c|c|c|c|c|} \hline 1 & 0 & 3 & 0 & 0 \\ \hline \end{array}$$

Column reduced matrix M_{ColRed} would be:

$$M_{\text{ColRed}} = \begin{array}{|c|c|c|c|c|} \hline \infty & 10 & 17 & 0 & 1 \\ \hline 12 & \infty & 11 & 2 & 0 \\ \hline 0 & 3 & \infty & 0 & 2 \\ \hline 15 & 3 & 12 & \infty & 0 \\ \hline 11 & 0 & 0 & 12 & \infty \\ \hline \end{array}$$

Each row and column of M_{ColRed} has at least one zero entry, so this matrix is reduced matrix. Column reduction cost (M) = 1 + 0 + 3 + 0 + 0 = 4

State space tree for 5 city problem is depicted in Fig. 6.6.1. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.



Example

Example: Find the solution of following travelling salesman problem using branch and bound method.

$$\text{Cost Matrix} = \begin{array}{|c|c|c|c|c|} \hline \infty & 20 & 30 & 10 & 11 \\ \hline 15 & \infty & 16 & 4 & 2 \\ \hline 3 & 5 & \infty & 2 & 4 \\ \hline 19 & 6 & 18 & \infty & 3 \\ \hline 16 & 4 & 7 & 16 & \infty \\ \hline \end{array}$$

Solution:

- The procedure for dynamic reduction is as follow:
- Draw state space tree with optimal reduction cost at root node.
- Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞ . Set $M[j][i] = \infty$
- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + $M[j][i]$

- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
 - Reduce above cost matrix by subtracting minimum value from each row and column.

∞	20	30	10	11	$\rightarrow 10$	∞	10	20	0	1	
15	∞	16	4	2	$\rightarrow 2$	13	∞	14	2	0	
3	5	∞	2	4	$\rightarrow 2$	1	3	∞	0	2	$= M'_1$
19	6	18	∞	3	$\rightarrow 3$	16	3	15	∞	0	
16	4	7	16	∞	$\rightarrow 4$	12	0	3	12	∞	
						\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
						1	0	3	0	0	

M'_1

is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

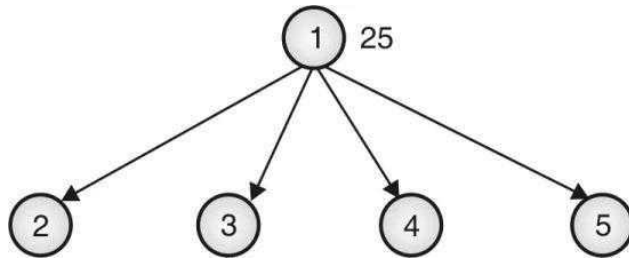
∞	10	17	0	1	
12	∞	11	2	0	
0	3	∞	0	2	$= M_1$
15	3	12	∞	0	
11	0	0	12	∞	

Cost of $M_1 = C(1)$

= Row reduction cost + Column reduction cost
 = $(10 + 2 + 2 + 3 + 4) + (1 + 3) = 25$

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree



Let us find cost of edge from node 1 to 2, 3, 4, 5.

Select edge 1-2:

Set $M_1[1][] = M_1[][2] =$

∞ Set $M_1[2][1] = \infty$

Reduce the resultant matrix if required.

∞	∞	∞	∞	∞	$\rightarrow x$
∞	∞	11	2	0	$\rightarrow 0$
0	∞	∞	0	2	$\rightarrow 0 = M_2$
15	∞	12	∞	0	$\rightarrow 0$
11	∞	0	12	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
0	x	0	0	0	

M_2 is already reduced.

Cost of node 2 :

$C(2) = C(1) + \text{Reduction cost} + M_1[1][2]$

$= 25 + 0 + 10 = 35$

Select edge 1-3

Set $M_1[1][] = M_1[][3] =$

∞ Set $M_1[3][1] = \infty$

Reduce the resultant matrix if required.

∞	∞	∞	∞	∞	$\rightarrow x$
12	∞	11	2	0	$\rightarrow 0$
0	∞	∞	0	2	$\rightarrow 0 \Rightarrow$
15	∞	12	∞	0	$\rightarrow 0$
11	∞	0	12	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
11	0	x	0	0	

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$= M_3$

Cost of node 3:

$C(3) = C(1) + \text{Reduction cost} + M_1[1][3]$

$= 25 + 11 + 17 = 53$

Select edge 1-4:

Set $M_1[1][] = M_1[][4]$

$= \infty$ Set $M_1[4][1] = \infty$

Reduce resultant matrix if required.

∞	∞	∞	∞	∞	$\rightarrow x$
12	∞	11	∞	0	$\rightarrow 0$
0	3	∞	∞	2	$\rightarrow 0$
∞	3	12	∞	0	$\rightarrow 0$
11	0	0	∞	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
0	0	0	x	0	

$M_1 \Rightarrow$ M_4

Matrix M_4 is already reduced. Cost of node 4:

$$C(4) = C(1) + \text{Reduction cost} + M_1[1][4] \\ = 25 + 0 + 0 = 25$$

Select edge 1-5:

Set $M_1[1][] = M_1[][5]$

$= \infty$ Set $M_1[5][1] = \infty$

Reduce the resultant matrix if required.

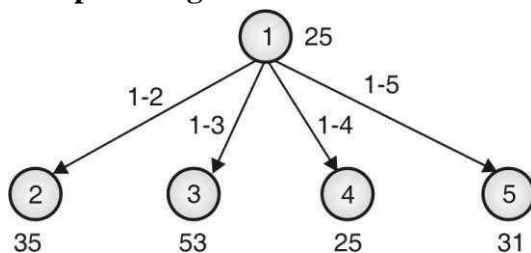
∞	∞	∞	∞	∞	$\rightarrow x$
12	∞	11	2	∞	$\rightarrow 2$
0	3	∞	0	∞	$\rightarrow 0$
15	3	12	∞	∞	$\rightarrow 3$
∞	0	0	12	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
0	0	0	0	x	

$M_1 \Rightarrow$ M_5

Cost of node 5:

$$C(5) = C(1) + \text{reduction cost} + M_1[1][5] \\ = 25 + 5 + 1 = 31$$

State space diagram:



Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5. Let's explore all three nodes.

Select path 1-4-2 : (Add edge 4-2)

Set $M_4[1][] = M_4[4][] = M_4[][]$

$[2] = \infty$ Set $M_4[2][1] = \infty$

Reduce resultant matrix if required.

∞	∞	∞	∞	∞	$\rightarrow x$
∞	∞	11	∞	0	$\rightarrow 0$
0	∞	∞	∞	2	$\rightarrow 0$
∞	∞	∞	∞	∞	$\rightarrow x$
11	∞	0	∞	∞	$\rightarrow 0$
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	
0	0	0	x	0	

$M_4 \Rightarrow$ M_6

Matrix M_6 is already reduced. Cost of node 6:

$$C(6) = C(4) + \text{Reduction cost} + M_4[4][2]$$

$$= 25 + 0 + 3 = 28$$

Select edge 4-3 (Path 1-4-3):

$$\text{Set } M_4[1][] = M_4[4][] = M_4[][3] =$$

$$\infty \text{ Set } M[3][1] = \infty$$

Reduce the resultant matrix if required.

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 12 & \infty & \infty & \infty & 0 \\ \hline \infty & 3 & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 11 & 0 & \infty & \infty & \infty \\ \hline \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow 0 \\ \rightarrow 2 \\ \rightarrow \infty \\ \rightarrow 0 \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 12 & \infty & \infty & \infty & 0 \\ \hline \infty & 1 & \infty & \infty & 0 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 11 & 0 & \infty & \infty & \infty \\ \hline \end{array} = M'_7$$

$$\downarrow \downarrow \downarrow \downarrow \downarrow$$

$$11 \ 0 \ x \ x \ 0$$

M'_7 is not reduced. Reduce it by subtracting 11 from column 1.

$$\therefore M'_7 \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 1 & \infty & \infty & \infty & 0 \\ \hline \infty & 1 & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & 0 & \infty & \infty & \infty \\ \hline \end{array} = M_7$$

Cost of node 7:

$$C(7) = C(4) + \text{Reduction cost} + M_4[4][3]$$

$$= 25 + 2 + 11 + 12 = 50$$

Select edge 4-5 (Path 1-4-5):

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 12 & \infty & 11 & \infty & \infty \\ \hline 0 & 3 & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & 0 & 0 & \infty & \infty \\ \hline \end{array} \begin{array}{l} \rightarrow x \\ \rightarrow 11 \\ \rightarrow 0 \\ \rightarrow x \\ \rightarrow 0 \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 1 & \infty & 0 & \infty & \infty \\ \hline 0 & 3 & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & 0 & 0 & \infty & \infty \\ \hline \end{array} = M_8$$

$$\downarrow \downarrow \downarrow \downarrow \downarrow$$

$$0 \ 0 \ 0 \ x \ x$$

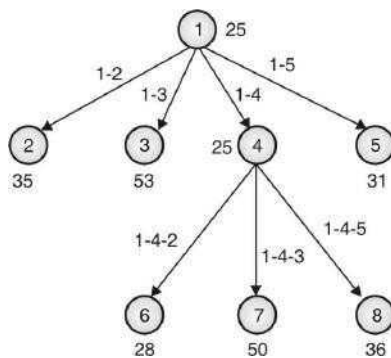
Matrix M_8 is reduced.

Cost of node 8:

$$C(8) = C(4) + \text{Reduction cost} + M_4[4][5]$$

$$= 25 + 11 + 0 = 36$$

State space tree



Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.

Add edge 2-3 (Path 1-4-2-3):

Set $M_6[1][] = M_6[4][] = M_6[2][]$

$= M_6[][3] = \infty$

Set $M_6[3][1] = \infty$

Reduce resultant matrix if required.

$M_6 \Rightarrow$	∞	∞	∞	∞	∞	$\rightarrow x$	\Rightarrow	∞	∞	∞	∞	∞	$= M'_9$
	∞	∞	∞	∞	∞	$\rightarrow x$		∞	∞	∞	∞	∞	
	0	∞	∞	∞	2	$\rightarrow 0$		0	∞	∞	∞	2	
	∞	∞	∞	∞	∞	$\rightarrow x$		∞	∞	∞	∞	∞	
	11	∞	∞	∞	∞	$\rightarrow 11$		0	∞	∞	∞	∞	

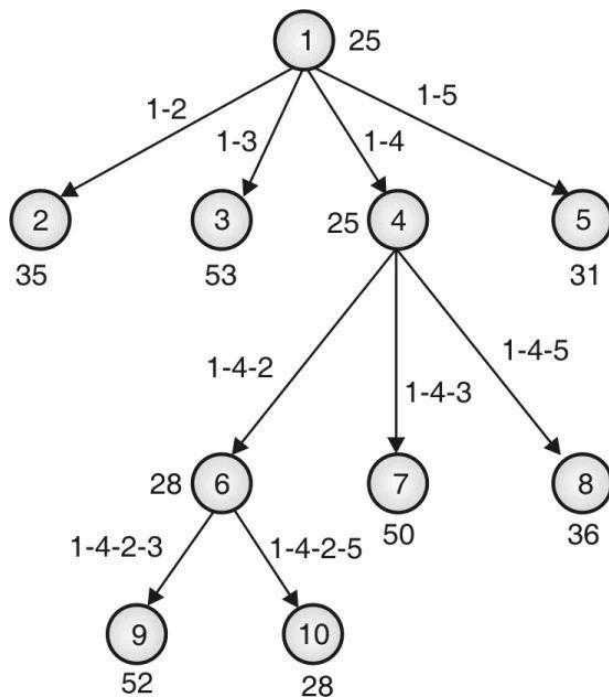
$$\therefore M_6 \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & 0 & \infty & \infty \\ \hline \end{array} = M_{10}$$

Cost of node 10:

$$C(10) = C(6) + \text{Reduction cost} + M_6 [2][5]$$

$$= 28 + 0 + 0 = 28$$

State space tree



Add edge 5-3 (Path 1-4-2-5-3):

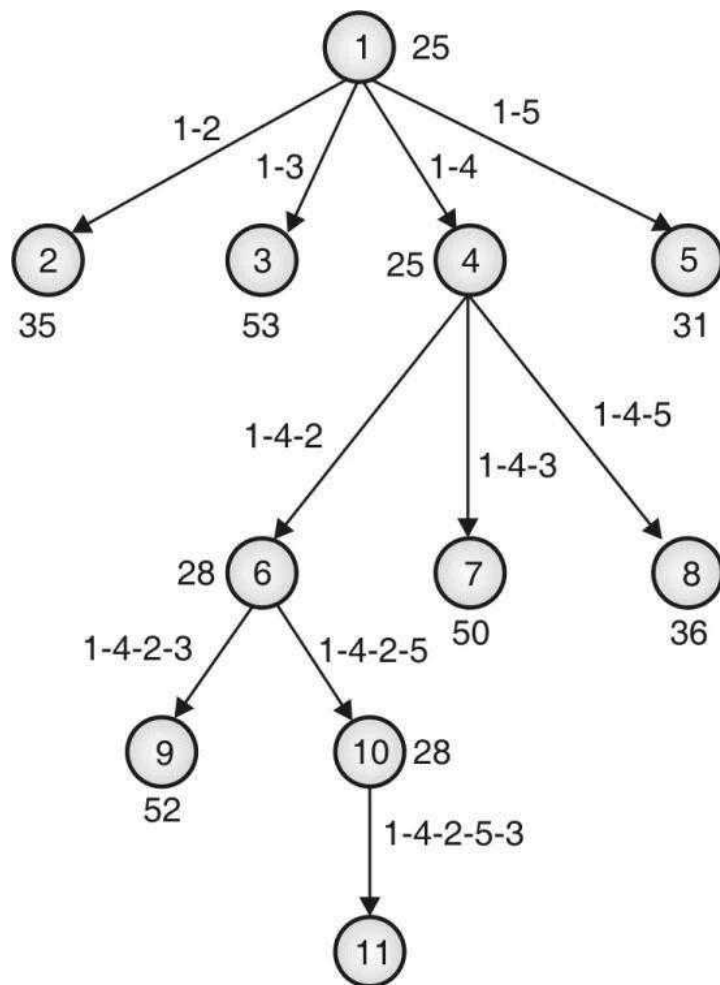
$$\therefore M_{10} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \end{array} = M_{11}$$

Cost of node 11:

$$C(11) = C(10) + \text{Reduction cost} + M_{10} [5][3]$$

$$= 28 + 0 + 0 = 28$$

State space tree:



So we can select any of the edge. Thus the final path includes the edges $\langle 3, 1 \rangle$, $\langle 5, 3 \rangle$, $\langle 1, 4 \rangle$, $\langle 4, 2 \rangle$, $\langle 2, 5 \rangle$, that forms the path $1 - 4 - 2 - 5 - 3 - 1$. This path has cost of 28.

UNIT 5

Tractable and Intractable Problems

Tractable problems refer to computational problems that can be solved efficiently using algorithms that can scale with the input size of the problem. In other words, the time required to solve a tractable problem increases at most polynomial with the input size.

On the other hand, intractable problems are computational problems for which no known algorithm can solve them efficiently in the worst-case scenario. This means that the time required to solve an intractable problem grows exponentially or even faster with the input size.

One example of a tractable problem is computing the sum of a list of n numbers. The time required to solve this problem scales linearly with the input size, as each number can be added to a running total in constant time. Another example is computing the shortest path between two nodes in a graph, which can be solved efficiently using algorithms like Dijkstra's algorithm or the A* algorithm.

In contrast, some well-known intractable problems include the traveling salesman problem, the knapsack problem, and the Boolean satisfiability problem. These problems are NP-hard, meaning that any problem in NP (the set of problems that can be solved in polynomial time using a non-deterministic Turing machine) can be reduced to them in polynomial time. While it is possible to find approximate solutions to these problems, there is no known algorithm that can solve them exactly in polynomial time.

In summary, tractable problems are those that can be solved efficiently with algorithms that scale well with the input size, while intractable problems are those that cannot be solved efficiently in the worst-case scenario.

Examples of Tractable problems

1. **Sorting:** Given a list of n items, the task is to sort them in ascending or descending order. Algorithms like Quick Sort and Merge Sort can solve this problem in $O(n \log n)$ time complexity.
2. **Matrix multiplication:** Given two matrices A and B , the task is to find their product $C = AB$. The best-known algorithm for matrix multiplication runs in $O(n^{2.37})$ time complexity, which is considered tractable for practical applications.
3. **Shortest path in a graph:** Given a graph G and two nodes s and t , the task is to find the shortest path between s and t . Algorithms like Dijkstra's algorithm and the A* algorithm can solve this problem in $O(m + n \log n)$ time complexity, where m is the number of edges and n is the number of nodes in the graph.
4. **Linear programming:** Given a system of linear constraints and a linear objective function, the task is to find the values of the variables that optimize the objective function subject to the constraints. Algorithms like the simplex method can solve this problem in polynomial time.
5. **Graph coloring:** Given an undirected graph G , the task is to assign a color to each node such that no two adjacent nodes have the same color, using as few colors as possible. The greedy algorithm can solve this problem in $O(n^2)$ time complexity, where n is the number of nodes in the graph.

These problems are considered tractable because algorithms exist that can solve them in polynomial time complexity, which means that the time required to solve them grows no faster than a polynomial function of the input size.

Examples of intractable problems

1. Traveling salesman problem (TSP): Given a set of cities and the distances between them, the task is to find the shortest possible route that visits each city exactly once and returns to the starting city. The best-known algorithms for solving the TSP have an exponential worst-case time complexity, which makes it intractable for large instances of the problem.
2. Knapsack problem: Given a set of items with weights and values, and a knapsack that can carry a maximum weight, the task is to find the most valuable subset of items that can be carried by the knapsack. The knapsack problem is also NP-hard and is intractable for large instances of the problem.
3. Boolean satisfiability problem (SAT): Given a boolean formula in conjunctive normal form (CNF), the task is to determine if there exists an assignment of truth values to the variables that makes the formula true. The SAT problem is one of the most well-known NP-complete problems, which means that any NP problem can be reduced to SAT in polynomial time.
4. Subset sum problem: Given a set of integers and a target sum, the task is to find a subset of the integers that sums up to the target sum. Like the knapsack problem, the subset sum problem is also intractable for large instances of the problem.

P (Polynomial) problems

P problems refer to problems where an algorithm would take a polynomial amount of time to solve, or where Big-O is a polynomial (i.e. $O(1)$, $O(n)$, $O(n^2)$, etc). These are problems that would be considered 'easy' to solve, and thus do not generally have immense run times.

NP (Non-deterministic Polynomial) Problems

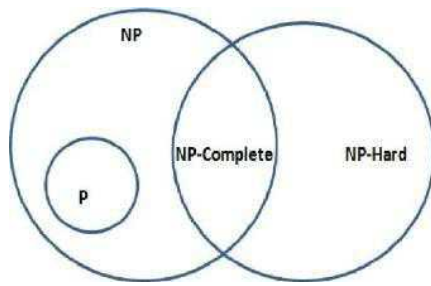
NP problems were a little harder for me to understand, but I think this is what they are. In terms of solving a NP problem, the run-time would not be polynomial. It would be something like $O(n!)$ or something much larger.

NP-Hard Problems

A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any* NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.

NP-Complete Problems

NP-Complete problems are problems that live in both the NP and NP-Hard classes. This means that NP-Complete problems can be verified in polynomial time and that any NP problem can be reduced to this problem in polynomial time.



Bin Packing problem

Bin Packing problem involves assigning n items of different weights and bins each of capacity c to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

The following 4 algorithms depend on the order of their inputs. They pack the item given first and then move on to the next input or next item

1) Next Fit algorithm

The simplest approximate approach to the bin packing problem is the Next-Fit (NF) algorithm which is explained later in this article. The first item is assigned to bin 1. Items 2,..., n are then considered by increasing indices : each item is assigned to the current bin, if it fits; otherwise, it is assigned to a new bin, which becomes the current one.

Visual Representation

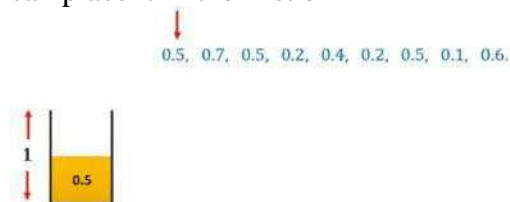
Let us consider the same example as used above and bins of size 1



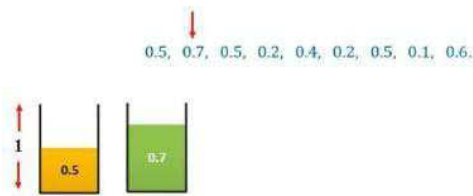
Assuming the sizes of the items be $\{0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6\}$.

The minimum number of bins required would be $\text{Ceil}((\text{Total Weight}) / (\text{Bin Capacity})) = \text{Ceil}(3.7/1) = 4$ bins.

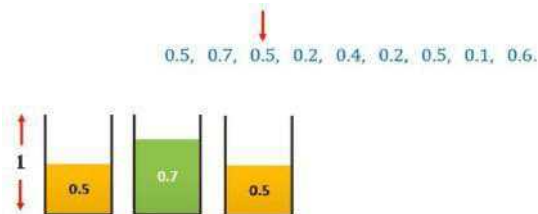
The Next fit solution (NF(I))for this instance I would be- Considering 0.5 sized item first, we can place it in the first bin



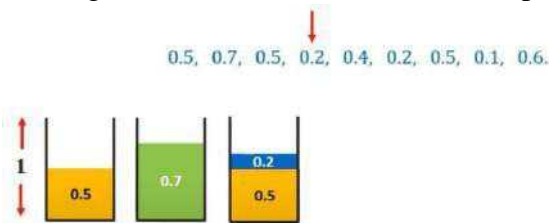
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



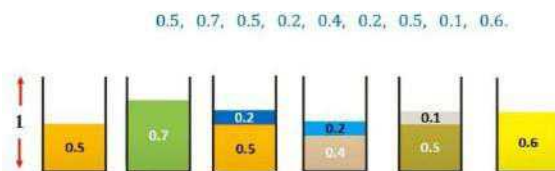
Moving on to the 0.5 sized item, we cannot place it in the current bin. Hence we place it in a new bin.



Moving on to the 0.2 sized item, we can place it in the current (third bin)



Similarly, placing all the other items following the Next-Fit algorithm we get-



Thus we need 6 bins as opposed to the 4 bins of the optimal solution. Thus we can see that this algorithm is not very efficient.

Analyzing the approximation ratio of Next-Fit algorithm

The time complexity of the algorithm is clearly $O(n)$. It is easy to prove that, for any instance I of BPP, the solution value $NF(I)$ provided by the algorithm satisfies the bound

$$NF(I) < 2z(I)$$

where $z(I)$ denotes the optimal solution value. Furthermore, there exist instances for which the ratio $NF(I)/z(I)$ is arbitrarily close to 2, i.e. the worst-case approximation ratio of NF is $r(NF) = 2$.

Pseudocode

NEXT FIT (size[], n, c)

size[] is the array containing the sizes of the items, n is the number of items and c is the capacity of the bin

{

 Initialize result (Count of bins) and remaining capacity in current bin. res = 0

 bin_rem = c

 Place items one by one for (int i = 0; i < n; i++) {

```

// If this item can't fit in current bin if (size[i] > bin_rem)
{
    Use a new binres++
    bin_rem = c - size[i]
}
else
    bin_rem -= size[i];
}
return res;
}

```

2) First Fit algorithm

A better algorithm, First-Fit (FF), considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits; only when the current item cannot fit into any initialized bin, is a new bin introduced

Visual Representation

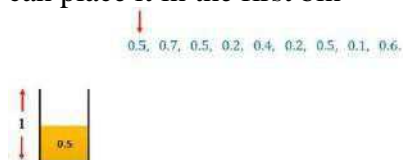
Let us consider the same example as used above and bins of size 1



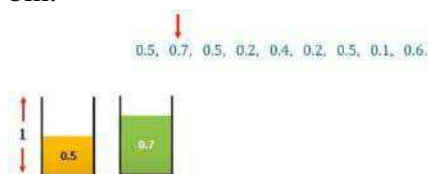
Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

The minimum number of bins required would be $\text{Ceil}((\text{Total Weight}) / (\text{Bin Capacity})) = \text{Ceil}(3.7/1) = 4$ bins.

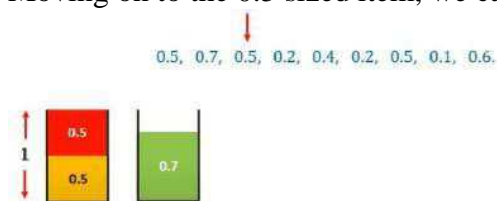
The First fit solution (FF(I)) for this instance I would be- Considering 0.5 sized item first, we can place it in the first bin



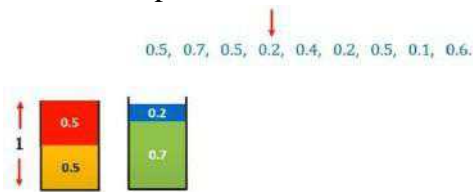
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



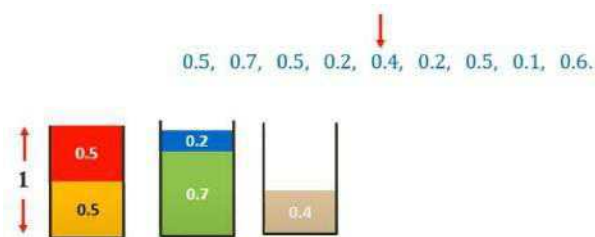
Moving on to the 0.5 sized item, we can place it in the first bin.



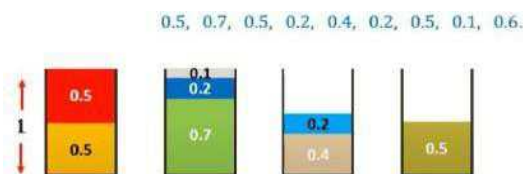
Moving on to the 0.2 sized item, we can place it in the first bin, we check with the second bin and we can place it there.



Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the First-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

Analyzing the approximation ratio of Next-Fit algorithm

If $FF(I)$ is the First-fit implementation for I instance and $z(I)$ is the most optimal solution, then:

$$FF(I) \leq \frac{17}{10} z(I) + 2$$

for all instances I of BPP, and that there exist instances I , with $z(I)$ arbitrarily large, for which

$$FF(I) > \frac{17}{10} z(I) - 8.$$

It can be seen that the First Fit never uses more than $1.7 * z(I)$ bins. So First-Fit is better than Next Fit in terms of upper bound on number of bins.

Pseudocode

FIRSTFIT(size[], n, c)

{

size[] is the array containing the sizes of the items, n is the number of items and c is the capacity of the bin

/Initialize result (Count of bins)

```

res = 0;
Create an array to store remaining space in bins there can be at most n bins bin_rem[n];

Place items one by one for (int i = 0; i < n; i++) {
    Find the first bin that can accommodate weight[i] int j;
    for (j = 0; j < res; j++) {
        if (bin_rem[j] >= size[i]) { bin_rem[j] = bin_rem[j] - size[i]; break;
        }
    }

    If no bin could accommodate size[i] if (j == res) {
        bin_rem[res] = c - size[i]; res++;
    }

}
return res;
}

```

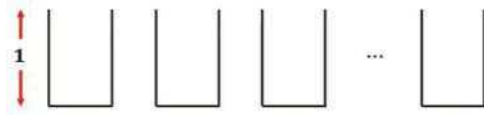
3) Best Fit Algorithm

The next algorithm, Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity (breaking ties in favor of the lowest indexed bin).

Simply put, the idea is to place the next item in the *tightest* spot. That is, put it in the bin so that the smallest empty space is left.

Visual Representation

Let us consider the same example as used above and bins of size 1

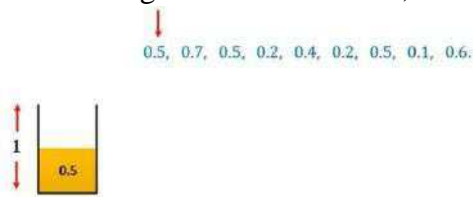


Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

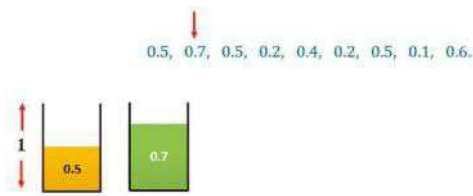
The minimum number of bins required would be $\text{Ceil}((\text{Total Weight}) / (\text{Bin Capacity})) = \text{Ceil}(3.7/1) = 4$ bins.

The First fit solution (FF(I)) for this instance I would be-

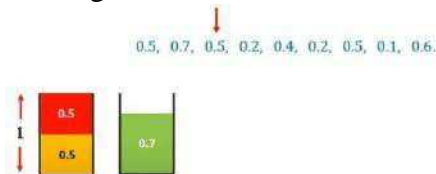
Considering 0.5 sized item first, we can place it in the first bin



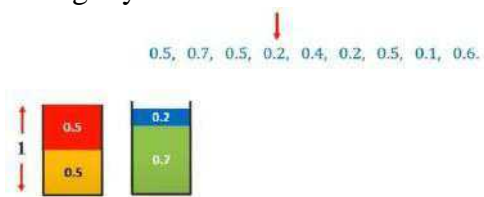
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



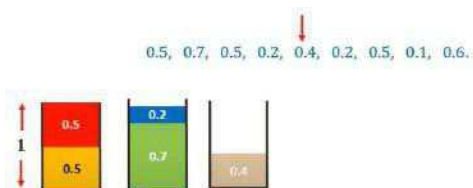
Moving on to the 0.5 sized item, we can place it in the first bin tightly.



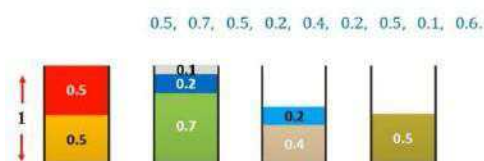
Moving on to the 0.2 sized item, we cannot place it in the first bin but we can place it in second bin tightly.



Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the First-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

Analyzing the approximation ratio of Best-Fit algorithm

It can be noted that Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity (breaking ties in favour of the lowest indexed bin). BF satisfies the same worst-case bounds as FF

Analysis Of upper-bound of Best-Fit algorithm

If $z(I)$ is the optimal number of bins, then Best Fit never uses more than $2 * z(I) - 2$ bins. So Best Fit is same as Next Fit in terms of upper bound on number of bins.

Pseudocode

BESTFIT(size[], n, c)

{
size[] is the array containing the sizes of the items, n is the number of items and c is the capacity of the bin

Initialize result (Count of bins) res = 0;

Create an array to store remaining space in bins there can be at most n bins bin_rem[n];

Place items one by one for (int i = 0; i < n; i++) {

Find the best bin that can accommodate weight[i] int j;

Initialize minimum space left and index of best bin int min = c + 1, bi = 0;

for (j = 0; j < res; j++) {

if (bin_rem[j] >= size[i] && bin_rem[j] - size[i] < min) { bi = j;

min = bin_rem[j] - size[i];

}

}

If no bin could accommodate weight[i], create a new bin if (min == c + 1) {

bin_rem[res] = c - size[i]; res++;

}

else

Assign the item to best bin bin_rem[bi] -= size[i];

}

return res;

}

Approximation Algorithms for the Traveling Salesman Problem

We solved the traveling salesman problem by exhaustive search in Section 3.4, mentioned its decision version as one of the most well-known *NP*-complete problems in Section 11.3, and saw how its instances can be solved by a branch-and-bound algorithm in Section 12.2. Here, we consider several approximation algorithms, a small sample of dozens of such algorithms suggested over the years for this famous problem.

But first let us answer the question of whether we should hope to find a polynomial-time approximation algorithm with a finite performance ratio on all instances of the traveling salesman problem. As the following theorem [Sah76] shows, the answer turns out to be no, unless $P = NP$.

THEOREM 1 If $P \neq NP$, there exists no c -approximation algorithm for the traveling salesman problem, i.e., there exists no polynomial-time approximation algorithm for this problem so that for all instances

$$f(s_a) \leq cf(s^*)$$

for some constant c .

Nearest-neighbour algorithm

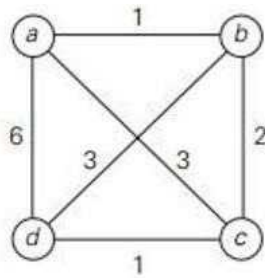
The following well-known greedy algorithm is based on the *nearest-neighbor* heuristic: always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

EXAMPLE 1 For the instance represented by the graph in Figure 12.10, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a : $a - b - c - d - a$ of length 10.



The optimal solution, as can be easily checked by exhaustive search, is the tour s^* : $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour s^*).

Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbor algorithm. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour. Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in Example 1, the algorithm will still yield the tour $a - b - c - d - a$ of length $4 + w$, and the optimal solution will still be $a - b - d - c - a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8},$$

which can be made as large as we wish by choosing an appropriately large value of w . Hence,

$R_A = \infty$ for this algorithm (as it should be according to Theorem 1).

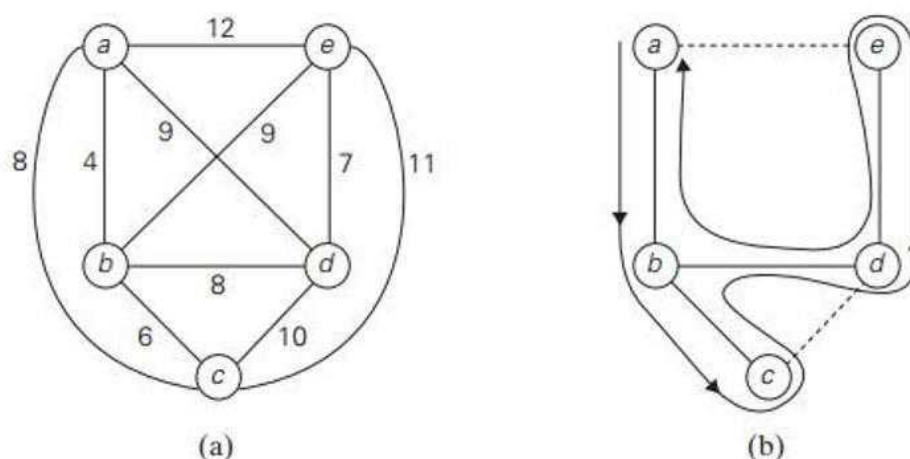
Twice-around-the-tree algorithm

Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)

Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

EXAMPLE 2 Let us apply this algorithm to the graph in Figure 12.11a. The minimum spanning tree of this graph is made up of edges (a, b) , (b, c) , (b, d) , and (d, e) . A twice-



around-the-tree walk that starts and ends at a is

$a, b, c, b, d, e, d, b, a$.

Eliminating the second b (a shortcut from c to d), the second d , and the third b (a shortcut from e to a) yields the Hamiltonian circuit

a, b, c, d, e, a

of length 39.

The tour obtained in Example 2 is not optimal. Although that instance is small enough to find an optimal solution by either exhaustive search or branch-and-bound, we refrained from doing so to reiterate a general point. As a rule, we do not know what the length of an optimal tour actually is, and therefore we cannot compute the accuracy ratio $f(s_a)/f(s^*)$. For the twice-around-the-tree algorithm, we can at least estimate it above, provided the graph is Euclidean.

Fermat's Little Theorem:

If n is a prime number, then for every a , $1 < a < n-1$,

$$a^{n-1} \equiv 1 \pmod{n} \text{ OR}$$

$$a^{n-1} \% n = 1$$

Example: Since 5 is prime, $2^4 \equiv 1 \pmod{5}$ [or $2^4 \% 5 = 1$],

$$3^4 \equiv 1 \pmod{5} \text{ and } 4^4 \equiv 1 \pmod{5}$$

Since 7 is prime, $2^6 \equiv 1 \pmod{7}$,

$$3^6 \equiv 1 \pmod{7}, 4^6 \equiv 1 \pmod{7}$$

$$5^6 \equiv 1 \pmod{7} \text{ and } 6^6 \equiv 1 \pmod{7}$$

1) Repeat following k times:

- a) Pick a randomly in the range $[2, n - 2]$
- b) If $\gcd(a, n) \neq 1$, then return false
- c) If $a^{n-1} \not\equiv 1 \pmod{n}$, then return false

2) Return true [probably prime].

Unlike merge sort, we don't need to merge the two sorted arrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort.

Using a randomly generated pivot we can further improve the time complexity of Quicksort.

Algorithm for random pivoting

partition(arr[], lo, hi)

 pivot = arr[hi]

 i = lo // place for swapping

 for j := lo to hi - 1 do

 if arr[j] <= pivot then swap arr[i] with arr[j] i = i + 1

 swap arr[i] with arr[hi] return i

partition_r(arr[], lo, hi)

 r = Random Number from lo to hi Swap arr[r] and arr[hi]

 return partition(arr, lo, hi) quicksort(arr[], lo, hi)

 if lo < hi

 p = partition_r(arr, lo, hi) quicksort(arr, lo, p-1) quicksort(arr, p+1, hi)

Finding kth smallest element

Problem Description: Given an array $A[]$ of n elements and a positive integer K , find the K th smallest element in the array. It is given that all array elements are distinct.

For Example :

Input : $A[] = \{10, 3, 6, 9, 2, 4, 15, 23\}$, $K = 4$

Output: 6

Input : A[] = {5, -8, 10, 37, 101, 2, 9}, K = 6

Output: 37

Quick-Select : Approach similar to quick sort

This approach is similar to the quick sort algorithm where we use the partition on the input array recursively. But unlike quicksort, which processes both sides of the array recursively, this algorithm works on only one side of the partition. We recur for either the left or right side according to the position of pivot.

Solution Steps

1. Partition the array A[left .. right] into two subarrays A[left .. pos] and A[pos + 1 .. right] such that each element of A[left .. pos] is less than each element of A[pos + 1 .. right].
2. Computes the number of elements in the subarray A[left .. pos] i.e. count = pos - left + 1
3. if (count == K), then A[pos] is the Kth smallest element.
4. Otherwise determines in which of the two subarrays A[left .. pos-1] and A[pos + 1 .. right] the Kth smallest element lies.
 - If (count > K) then the desired element lies on the left side of the partition
 - If (count < K), then the desired element lies on the right side of the partition. Since we already know i values that are smaller than the kth smallest element of A[left .. right], the desired element is the (K - count)th smallest element of A[pos + 1 .. right].
- Base case is the scenario of single element array i.e left ==right. return A[left] or A[right].

Pseudo-Code

// Original value for left = 0 and right = n-1

int kthSmallest(int A[], int left, int right, int K)

```
{
if (left == right)
return A[left]

    int pos = partition(A, left, right)
    count = pos - left + 1
    if ( count == K )
        return A[pos]

    else if ( count > K )

        return kthSmallest(A, left, pos-1, K)
    else
        return kthSmallest(A, pos+1, right, K-i)
}
```

int partition(int A[], int l, int r)

```
{
    int x = A[r]

    int i = l-1
```



```

for ( j = l to r-1 )
{
    if (A[j] <= x)

        {
            i = i + 1 swap(A[i], A[j])
        }
}
swap(A[i+1], A[r])
return i+1
}

```

Complexity Analysis

Time Complexity: The worst-case time complexity for this algorithm is $O(n^2)$, but it can be improved if we choose the pivot element randomly. If we randomly select the pivot, the expected time complexity would be linear, **$O(n)$** .