❖**Characteristics of algorithm: All algorithm must satisfy the following criteria**.

1. Input: Zero or more quantities are externally supplied. If there are 0 external inputs then the input will be taken internally.

2. Output: At least one quantity is produced.

3. Definiteness: Each instruction is clear and unambiguous e.g. add 5 to x or y is not clear statement and “Divide 5 by 0” is indefinite

4.Finiteness: If we trace out the instructions of an algorithm then for all cases the algorithm terminates after a finite number of steps. e.g. operating system is exceptional. It is made to run infinitely till power is not off.

5. Effectiveness: Every instruction must be very basic so that it can be carried out in principle by a person using only paper and pencil

**7. Explain the process of designing an algorithm.**

The first step in designing an algorithm is to thoroughly understand the problem statement, including its requirements, constraints, and objectives.

**Analyze the Problem**:

Analyze the problem to determine its complexity, possible approaches, and potential challenges.

Break down the problem into smaller subproblems or components, if applicable.

**Choose a Solution Strategy**:

Select an appropriate solution strategy based on the nature of the problem and its requirements.

Consider different algorithmic techniques such as brute force, greedy algorithms, divide and conquer, dynamic programming, or backtracking.

**Design the Algorithm**:

Develop a step-by-step plan or a set of rules to solve the problem using the chosen solution strategy.

**Consider Efficiency and Optimization**:

Evaluate the time and space complexity of the algorithm to ensure it meets the performance requirements.

**Test and Validate**:

Test the algorithm with various inputs, including normal cases, boundary cases, and edge cases, to verify its correctness and robustness.

**Document the Algorithm**:

Document the algorithm's design, including its description, inputs, outputs, and implementation details.

**Implement and Deploy**:

Implement the algorithm in a programming language of your choice, following the design specifications and documentation.

**❖Different design strategies are:**

a. Divide and Conquer

b. Greedy Method

c. Dynamic Programming

d. Backtracking e. Branch Bound

**Time Complexity**:

* Time complexity refers to the amount of time taken by an algorithm to run as a function of the length of the input.

**Space Complexity**:

* Space complexity refers to the amount of memory (space) required by an algorithm to execute as a function of the length of the input.

**what are three different asyptoticnotations**

1. **Big O Notation (𝑂*O*)**:

* Big O notation represents the upper bound or worst-case scenario of the growth rate of a function.
* It provides an upper limit on the rate of growth of a function, indicating the maximum amount of resources (time, space) required by an algorithm.
* Formally, 𝑓(𝑛)=𝑂(𝑔(𝑛))*f*(*n*)=*O*(*g*(*n*)) if there exist positive constants 𝑐*c* and 𝑛0*n*0​ such that 𝑓(𝑛)≤𝑐⋅𝑔(𝑛)*f*(*n*)≤*c*⋅*g*(*n*) for all 𝑛≥𝑛0*n*≥*n*0​.
* For example, if 𝑓(𝑛)=2𝑛2+3𝑛+1*f*(*n*)=2*n*2+3*n*+1 and 𝑔(𝑛)=𝑛2*g*(*n*)=*n*2, we write 𝑓(𝑛)=𝑂(𝑛2)*f*(*n*)=*O*(*n*2), indicating that 𝑓(𝑛)*f*(*n*) grows no faster than 𝑛2*n*2 for large 𝑛*n*.

1. **Big Omega Notation (ΩΩ)**:

* Big Omega notation represents the lower bound or best-case scenario of the growth rate of a function.
* It provides a lower limit on the rate of growth of a function, indicating the minimum amount of resources (time, space) required by an algorithm.
* Formally, 𝑓(𝑛)=Ω(𝑔(𝑛))*f*(*n*)=Ω(*g*(*n*)) if there exist positive constants 𝑐*c* and 𝑛0*n*0​ such that 𝑓(𝑛)≥𝑐⋅𝑔(𝑛)*f*(*n*)≥*c*⋅*g*(*n*) for all 𝑛≥𝑛0*n*≥*n*0​.
* For example, if 𝑓(𝑛)=2𝑛2+3𝑛+1*f*(*n*)=2*n*2+3*n*+1 and 𝑔(𝑛)=𝑛2*g*(*n*)=*n*2, we write 𝑓(𝑛)=Ω(𝑛2)*f*(*n*)=Ω(*n*2), indicating that 𝑓(𝑛)*f*(*n*) grows at least as fast as 𝑛2*n*2 for large 𝑛*n*.

1. **Big Theta Notation (ΘΘ)**:

* Big Theta notation represents both the upper and lower bounds or tight bound of the growth rate of a function.
* It provides a precise characterization of the rate of growth of a function, indicating the exact amount of resources (time, space) required by an algorithm.
* Formally, 𝑓(𝑛)=Θ(𝑔(𝑛))*f*(*n*)=Θ(*g*(*n*)) if there exist positive constants 𝑐1*c*1​, 𝑐2*c*2​, and 𝑛0*n*0​ such that 𝑐1⋅𝑔(𝑛)≤𝑓(𝑛)≤𝑐2⋅𝑔(𝑛)*c*1​⋅*g*(*n*)≤*f*(*n*)≤*c*2​⋅*g*(*n*) for all 𝑛≥𝑛0*n*≥*n*0​.
* For example, if 𝑓(𝑛)=2𝑛2+3𝑛+1*f*(*n*)=2*n*2+3*n*+1 and 𝑔(𝑛)=𝑛2*g*(*n*)=*n*2, we write 𝑓(𝑛)=Θ(𝑛2)*f*(*n*)=Θ(*n*2), indicating that 𝑓(𝑛)*f*(*n*) grows at the same rate as 𝑛2*n*2 for large 𝑛*n*.

**Heap Sort**

A heap is a complete binary tree and it is implemented in an array as an sequential representation.

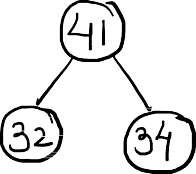
▪ Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if it is a complete binary tree (every level, except possibly the last , is completely filled and all the nodes are as far left as possible.

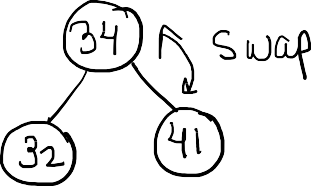
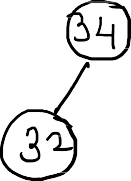
▪ It is comparison based sorting.

▪ All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children smaller than the root and so on. Such a heap is called a max-heap

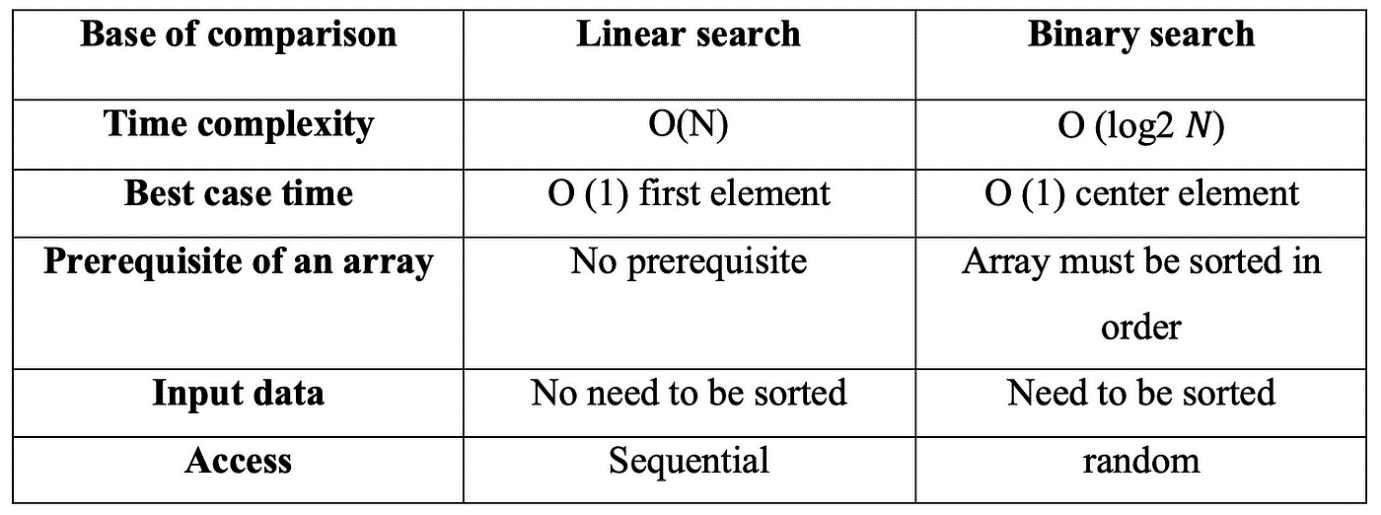
. ▪ A heap is called max heap or descending heap if every node of heap has value greater than or equal to the value of every child of that node.

▪ So in max heap root will keep the highest value. ▪ Similarly a heap is called min heap or ascending heap if every node of heap has a value less than or equal to the value of every child of that node



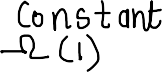
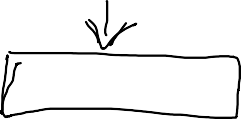


**Diiference Linear And Binary**



|  |  |
| --- | --- |
|  | * **Linear Search** |
|  | In Computer Science , a linear search or sequential search is a method for |
|  | finding an element within a list. It sequentially checks each element of the list |

until the match is found or the whole list has been searched.



Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is matched successfully,

Best Case Complexity = O(1)

Worst Case Complexity and Average Case Complexity = O(n) It is used for unsorted and unordered small list of elements.

It has a time complexity of O(n), which means the time is linearly dependent on the number of elements.

for i=0; i<n;i++)

{ if (arr[i]==x)

{printf("element found" flag=1;

break;

}}

if( flag==0) { printf("element not found");}

#### Binary Search:

* The sequential search situation will be in worst case if the element is at the end of the list.
* For eliminating this problem, we have one efficient searching technique called binary search.
* The condition for binary search is that all data should be in a Sorted array.
* If it is less than the middle element then we search it in the left portion of the

array and if it is greater than the middle element then search will be in the right portion of the array. Now we will take that portion only for search and compare with middle element of that portion. This process will be in iteration

until we find the element or middle element has no left or right portion to

compare with middle element of that portion. This process will be in iteration

until we find the element or middle element has no left or right portion to search.

* **Non-Recursive Binary Search Algorithm: Algorithm :** BinSearch(a, n, x)

//Given an array a[1..n] of elements in non-decreasing order, n>=0, Determine whether x is present, and if so return j such that x=a[j]; else return 0

{

low =1; high=n;

While low<=high do

{

mid=(low + high)/2 if a[mid]=x then return mid;

else if

a[mid] > x then

else

}

return 0;

}

high=mid-1; low=mid+1;

**Insertion Sort is** a simple sorting algorithm that builds the final sorted array one item at a time by repeatedly inserting the next element into the sorted part of the array.

Here's how Insertion Sort works:

1. **Initialization**: Start with the second element (index 1) and consider it as the current element.
2. **Insertion**: Iterate through the unsorted part of the array, comparing the current element with the elements in the sorted part of the array.
3. **Shift**: If the current element is smaller (or larger, depending on the sorting order) than the element to its left, shift the larger element one position to the right to make space for the current element.
4. **Insert**: Place the current element in the correct position in the sorted part of the array.
5. **Repeat**: Repeat steps 2-4 for all remaining unsorted elements.

Here's a note on the best, average, and worst-case scenarios for Insertion Sort:



best-case scenario occurs when the input array is already sorted. In this case, Insertion Sort performs efficiently, requiring only 𝑂(𝑛)*O*(*n*) comparisons and no swaps. The time complexity is 𝑂(𝑛)*O*(*n*).

* **Average Case**: In the average case, Insertion Sort requires 𝑂(𝑛2)*O*(*n*2) comparisons and swaps to sort an array of 𝑛*n* elements. This occurs when the elements are randomly distributed and not in any particular order.
* **Worst Case**: The worst-case scenario occurs when the input array is sorted in reverse order. In this case, each element must be compared and shifted to its correct position in the sorted part of the array. The time complexity is 𝑂(𝑛2)*O*(*n*2).

Merge sort is yet another sorting algorithm that falls under the category of [Divide and Conquer](https://www.javatpoint.com/divide-and-conquer-introduction) technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

1. FUNCTIONS: MERGE (A, p, q, r)
3. 1. n 1 = q-p+1
4. 2. n 2= r-q
5. 3. create arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
6. 4. for i ← 1 to n 1
7. 5. do [i] ← A [ p+ i-1]
8. 6. for j ← 1 to n2
9. 7. do R[j] ← A[ q + j]
10. 8. L [n 1+ 1] ← ∞
11. 9. R[n 2+ 1] ← ∞
12. 10. I ← 1
13. 11. J ← 1
14. 12. For k ← p to r
15. 13. Do if L [i] ≤ R[j]
16. 14. then A[k] ← L[ i]
17. 15. i ← i +1
18. 16. else A[k] ← R[j]
19. 17. j ← j+1

**Best Case Complexity:** The merge sort algorithm has a best-case time complexity of **O(n\*log n)**

 for the already sorted array.

**Average Case Complexity:** The average-case time complexity for the merge sort algorithm is

**O(n\*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending

order nr in the descending order.

**Worst Case Complexity:** The worst-case time complexity is also **O(n\*log n)**, which occurs when we sort the descending order of an array into the ascending order.

**Space Complexity:** The space complexity of merge sort is **O(n)**.

Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.

Example of Longest Common Sequence

here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)

m = length [X] and n = length [Y]

m = 7 and n = 6

Here x1= x [1] = A y1= y [1] = B

x2= B y2= D

x3= C y3= C

x4= B y4= A

x5= D y5= B

x6= A y6= A

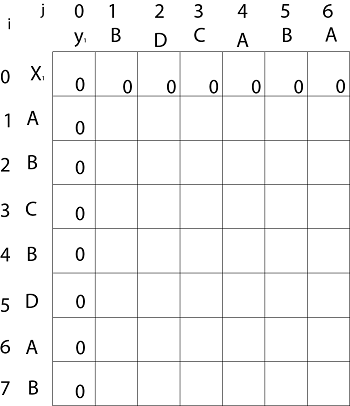
x7= B

Now fill the values of c [i, j] in m x n table

Initially, for i=1 to 7 c [i, 0] = 0

For j = 0 to 6 c [0, j] = 0

That is:



**Now for i=1 and j = 1**

x1 and y1 we get x1 ≠ y1 i.e. A ≠ B

And c [i-1,j] = c [0, 1] = 0

c [i, j-1] = c [1,0 ] = 0

That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '

**Now for i=1 and j = 2**

x1 and y2 we get x1 ≠ y2 i.e. A ≠ D

c [i-1,j] = c [0, 2] = 0

c [i, j-1] = c [1,1 ] = 0

That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '

**Now for i=1 and j = 3**

x1 and y3 we get x1 ≠ y3 i.e. A ≠ C

c [i-1,j] = c [0, 3] = 0

c [i, j-1] = c [1,2 ] = 0

so c [1,3] = 0 b [1,3] = ' ↑ '

**Now for i=1 and j = 4**

x1 and y4 we get. x1=y4 i.e A = A

c [1,4] = c [1-1,4-1] + 1

= c [0, 3] + 1

= 0 + 1 = 1

c [1,4] = 1

b [1,4] = ' ↖ '

**Now for i=1 and j = 5**

x1 and y5 we get x1 ≠ y5

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4 ] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '←'

**Now for i=1 and j = 6**

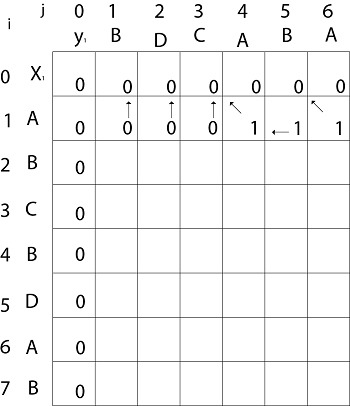
x1 and y6 we get x1=y6

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '



**Now for i=2 and j = 1**

We get x2 and y1 B = B i.e. x2= y1

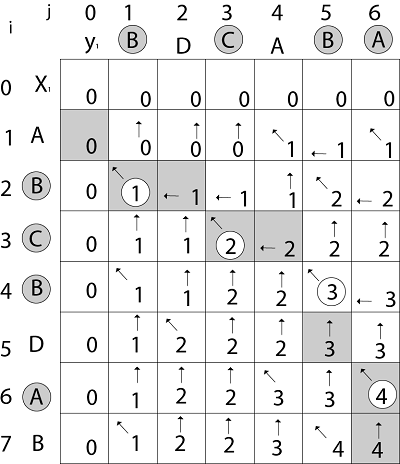
c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get



**Quick Sort:**

* The divide and Conquer approach can be used to arrive at an efficient sorting
* Method different from merge sort.
* In merge sort the file a[1..n] was divided at its midpoint into subarrays which are independently sorted and later merged.
* In quick sort, the division into two subarrays is made. So that the sorted

subarrays do not need to be merged later.

**3.Analyze the best, average and worst-case complexity of quick sort**.

1. **Best-case complexity**:
   * The best-case scenario occurs when the pivot element divides the array into two equal parts in each recursion level.
   * In this case, each partitioning step results in two subproblems of approximately equal size.
   * The best-case time complexity is 𝑂(𝑛log⁡𝑛)*O*(*n*log*n*), where 𝑛*n* is the number of elements in the array.
   * This happens when the pivot is the median of the array or when the array is already sorted.
2. **Average-case complexity**:
   * In the average case, QuickSort divides the array into approximately two equal parts in each recursion level, resulting in 𝑂(𝑛log⁡𝑛)*O*(*n*log*n*) time complexity.
   * QuickSort's average-case time complexity is 𝑂(𝑛log⁡𝑛)*O*(*n*log*n*) because it splits the array into roughly equal parts at each step.
   * The average-case time complexity analysis assumes that the pivot selection is random or uniformly distributed across the array.
3. **Worst-case complexity**:
   * The worst-case scenario occurs when the pivot element is either the smallest or the largest element in the array, causing partitioning to be highly unbalanced.
   * In the worst case, one partition will have 𝑛−1*n*−1 elements, while the other partition will have 00 elements.
   * This leads to 𝑂(𝑛2)*O*(*n*2) time complexity.
   * The worst-case time complexity can be mitigated by using different strategies for pivot selection, such as choosing the median of the first, middle, and last elements of the array
   * .

**5.Solve using Insertion sort method 67 44 82 17 20 and discuss the time**

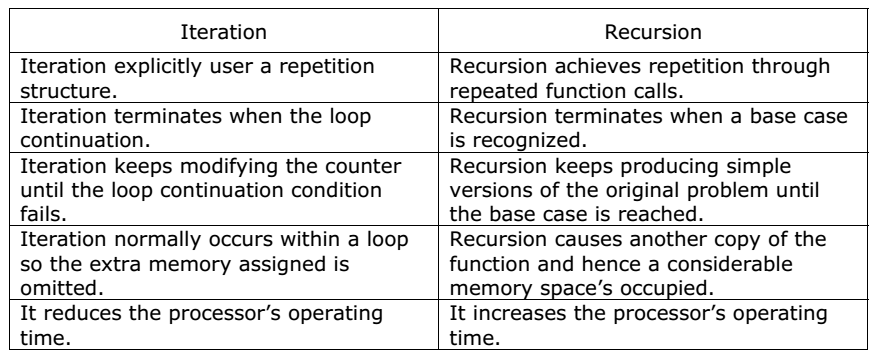
1. **Initial List**: 67,44,82,17,2067,44,82,17,20
2. **Pass 1**:
   * Start with the second element (44).
   * Compare it with the first element (67) and swap if necessary.
   * Result: 44,67,82,17,2044,67,82,17,20
3. **Pass 2**:
   * Consider the third element (82).
   * Compare it with the elements before it and insert it in the correct position.
   * Result: 44,67,82,17,2044,67,82,17,20 (no change)
4. **Pass 3**:
   * Consider the fourth element (17).
   * Compare it with the elements before it and insert it in the correct position.
   * Result: 17,44,67,82,2017,44,67,82,20
5. **Pass 4**:
   * Consider the fifth element (20).
   * Compare it with the elements before it and insert it in the correct position.
   * Result: 17,20,44,67,8217,20,44,67,82
6. **Sorted List**: 17,20,44,67,8217,20,44,67,82

* **Time Complexity**:
  + In the worst-case scenario, when the list is in reverse order, Insertion Sort will require 𝑂(𝑛2)*O*(*n*2) comparisons and swaps, where 𝑛*n* is the number of elements in the list.
  + In the best-case scenario, when the list is already sorted, Insertion Sort will require 𝑂(𝑛)*O*(*n*) comparisons and no swaps.
  + In the average-case scenario, Insertion Sort has a time complexity of 𝑂(𝑛2)*O*(*n*2).

**9.State the best, average and worst-case complexities of binary search for successful and unsuccessful search.**

:

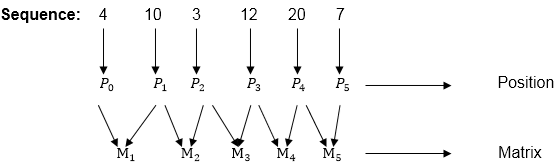
1. **For Successful Search**:
   * **Best case**: 𝑂(1)*O*(1)
     + The best case occurs when the target element is found at the middle of the array in the first comparison.
     + In this case, the algorithm terminates immediately after one comparison, resulting in constant time complexity.
   * **Average case**: 𝑂(log⁡𝑛)*O*(log*n*)
     + In the average case, the binary search algorithm splits the search space in half with each comparison.
     + The number of comparisons required to find the target element is proportional to the logarithm of the number of elements in the array.
   * **Worst case**: 𝑂(log⁡𝑛)*O*(log*n*)
     + Similarly to the average case, the worst case also has a time complexity of 𝑂(log⁡𝑛)*O*(log*n*).
     + Even if the target element is located at the end of the array, the binary search algorithm efficiently reduces the search space by half in each step.
2. **For Unsuccessful Search**:
   * **Best case**: 𝑂(1)*O*(1)
     + The best case occurs when the target element is smaller or larger than all elements in the array.
     + In this scenario, the algorithm immediately determines that the target element is not present, resulting in constant time complexity.
   * **Average case**: 𝑂(log⁡𝑛)*O*(log*n*)
     + The average case time complexity for an unsuccessful search is the same as the successful search, 𝑂(log⁡𝑛)*O*(log*n*).
     + Binary search efficiently reduces the search space in each step, even when the target element is not present.
   * **Worst case**: 𝑂(log⁡𝑛)*O*(log*n*)
     + The worst-case time complexity for an unsuccessful search is also 𝑂(log⁡𝑛)*O*(log*n*).
     + Even if the target element is not in the array, the binary search algorithm efficiently narrows down the search space in each iteration.
3. **Write difference between recursion and iteration.**



1. **Divide and Conquer**:
   * **Example**: Merge Sort
   * **Explanation**: Merge Sort is a classic example of the divide and conquer technique. It divides the input array into two halves, recursively sorts each half, and then merges the sorted halves to produce a single sorted array. Here's how it works:
     + **Divide**: Divide the unsorted array into two halves.
     + **Conquer**: Recursively sort each half.
     + **Merge**: Merge the sorted halves into a single sorted array.
   * **Application**: Merge Sort is commonly used for sorting large datasets efficiently. It has a time complexity of 𝑂(𝑛log⁡𝑛)*O*(*n*log*n*) and is considered one of the most efficient sorting algorithms.
2. **Greedy Method**:
   * **Example**: Fractional Knapsack Problem
   * **Explanation**: The Fractional Knapsack Problem involves selecting items with maximum value while staying within a weight constraint. The greedy approach involves selecting items with the maximum value-to-weight ratio first, filling the knapsack as much as possible at each step. Here's how it works:
     + **Step 1**: Calculate the value-to-weight ratio for each item.
     + **Step 2**: Sort the items based on their value-to-weight ratio in non-increasing order.
     + **Step 3**: Start with an empty knapsack. Add items to the knapsack one by one until the weight constraint is reached.
   * **Application**: The greedy approach is often used for optimization problems where making locally optimal choices leads to a globally optimal solution. The Fractional Knapsack Problem is one such example.
3. **Dynamic Programming**:
   * **Example**: Fibonacci Sequence
   * **Explanation**: The Fibonacci Sequence is a classic example of dynamic programming. The sequence is defined recursively as 𝐹(𝑛)=𝐹(𝑛−1)+𝐹(𝑛−2)*F*(*n*)=*F*(*n*−1)+*F*(*n*−2) with base cases 𝐹(0)=0*F*(0)=0 and 𝐹(1)=1*F*(1)=1. Dynamic programming optimizes this recursive approach by storing the results of subproblems in a table and reusing them when needed. Here's how it works:
     + **Step 1**: Define the recursive relation 𝐹(𝑛)=𝐹(𝑛−1)+𝐹(𝑛−2)*F*(*n*)=*F*(*n*−1)+*F*(*n*−2) and base cases 𝐹(0)=0*F*(0)=0 and 𝐹(1)=1*F*(1)=1.
     + **Step 2**: Use memoization or tabulation to store and reuse the results of subproblems.
     + **Step 3**: Compute the Fibonacci numbers iteratively or recursively using the stored results.
   * **Application**: Dynamic programming is used to solve optimization problems where the solution can be efficiently computed by breaking it down into smaller overlapping subproblems. The Fibonacci Sequence is a classic example of such a problem.
4. **Decrease and Conquer**:
   * **Example**: Binary Search
   * **Explanation**: Binary Search is a classic example of the decrease and conquer technique. It reduces the search space by half at each step until the target element is found or the search space is empty. Here's how it works:
     + **Step 1**: Compare the target element with the middle element of the sorted array.
     + **Step 2**: If the target element is equal to the middle element, return its index.
     + **Step 3**: If the target element is less than the middle element, search the left half of the array recursively.
     + **Step 4**: If the target element is greater than the middle element, search the right half of the array recursively.
   * **Application**: Binary Search is commonly used to efficiently search for an element in a sorted array. It has a time complexity of 𝑂(log⁡𝑛)*O*(log*n*) and is significantly faster than linear search for large datasets
5. **Strassen's Matrix** Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is **O(n3)**, since it takes two loops to multiply. Strassen’s method was introduced to reduce the time complexity from **O(n3)** to **O(nlog 7)**.

**Multiple Chain matrix**

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 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.



**Calculation of Product of 2 matrices:**

1. m (1,2) = m1 x m2

= 4 x 10 x 10 x 3

= 4 x 10 x 3 = 120

2. m (2, 3) = m2 x m3

= 10 x 3 x 3 x 12

= 10 x 3 x 12 = 360

3. m (3, 4) = m3 x m4

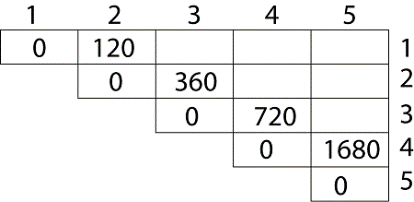
= 3 x 12 x 12 x 20

= 3 x 12 x 20 = 720

4. m (4,5) = m4 x m5

= 12 x 20 x 20 x 7

= 12 x 20 x 7 = 1680



**Now product of 3 matrices:**

M [1, 3] = M1 M2 M3

1. There are two cases by which we can solve this multiplication: ( M1 x M2) + M3, M1+ (M2x M3)

Example of Matrix Chain Multiplication

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

M [2, 4] = M2 M3 M4

DAA Example of Matrix Chain Multiplication

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

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M2+(M3 x M4) this combination is chosen for the output making.

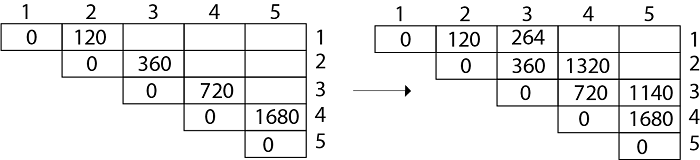
M [3, 5] = M3 M4 M5

1. There are two cases by which we can solve this multiplication: ( M3 x M4) + M5, M3+ ( M4xM5)

Example of Matrix Chain Multiplication

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and ( M3 x M4) + M5this combination is chosen for the output making.

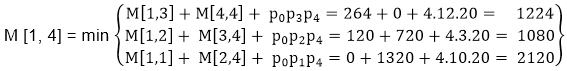


Now Product of 4 matrices:

M [1, 4] = M1 M2 M3 M4

1. ( M1 x M2 x M3) M4
2. M1 x(M2 x M3 x M4)
3. (M1 xM2) x ( M3 x M4)

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



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

### Knapsack Problem:

* Suppose a student is going on a trip. He has a single knapsack that can carry items of weights at most 'm'.
* In addition, he is allowed to break items into fractions arbitrarily. Each item i has some utility value say pi. He wants to fill his knapsack with items most useful to him with total weight at most m.
* Let us try to apply the greedy method to solve knapsack problem.

We are given n objects and a knapsack or bag.

Object i has a weight wi and the knapsack has capacity m. If a fraction xi, of object i is placed into the knapsack, then profit of pixi is earned.

* The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require a total weight of all chosen objects to be at most m.

Formally the problem can be stated as

maximize ∑ pixi Subject to

∑ wixi <= m

The profit and weights are positive numbers.

An optimal solution is a feasible solution for which pixi is maximized.

There are following three approaches to generate feasible solutions.

1. Greedy by profit
2. Greedy by weight
3. Greedy by density

### Greedy by Profit:

At each step select from the remaining items the one with the highest

### Greedy by Profit:

At each step select from the remaining items the one with the highest profit.(provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing the most profitable items first.

### Greedy by Weight:

At each step select from the remaining items the one with the least weight. (the capacity of knapsack not exceeded). This approach tries to maximize the profit by putting as many items into the knapsack as possible.

### Greedy by profit density:

At each step select from the remaining items the one with the highest or largest profit density, pi/wi the capacity of knapsack is not exceeded. This approach tries to maximize the profit by choosing items with the largest profit per unit of weight.

### Algorithm: Greedy\_knapsack(m, n)

//p[1..n] and w[1..n] contain the profits and weights respectively of the n objects ordered such that p[i]/w[i] >= p[i+1]/ w[i+1]

m is knapsack size and x[1..n] is the solution vector.

{

for i=1 to n do

x[i]=0.0; // Initialize x

U=m;

for i=1 to n do

{

If (w[i] > U) then break; x[i]=1.0;

U=U-w[i]; // decreasing the available capacity

After putting the object into knapsack.

}

If (i<=n) then x[i]=U/w[i];

}

**Knapsacks algorihm**

Objects:         1     2     3     4     5     6     7

Profit (P):         10     15     7     8     9     4

Weight(w):       1     3     5     4     1     3     2

W (Weight of the knapsack): 15

n (no of items): 7

### **First approach:**

**First approach:**

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 3 | 15 | 5 | 15 - 5 = 10 |
| 2 | 10 | 3 | 10 - 3 = 7 |
| 6 | 9 | 3 | 7 - 3 = 4 |
| 5 | 8 | 1 | 4 - 1 = 3 |
| 7 | 7 \* ¾ = 5.25 | 3 | 3 - 3 = 0 |

The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25

### Minimum cost Spanning Tree:

A spanning tree of connected graph G contains all the nodes and has the edges, which connects all the nodes. So number of edges will be 1 less than the number of nodes. Let us take a connected graph G.

* **Kruskal Algorithm:**
* Kruskal's Algorithm uses **greedy** method.
* Kruskal's Algorithm is used to find the minimum spanning tree for a

connected weighted graph.

* The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.
* Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.
* **Kruskal Algorithm Steps:**

1. Arrange all the edges in increasing order of weight or cost.

Construct **heap** out of edge costs using Heapify;

**Heapify** procedure which creates the heap of edges such that edge with

minimum cost is at the root.

1. Choose edge with lowest cost and delete from edges group and add it into Spanning tree. If the edge creates cycle then that edge will be rejected.
2. **Adjust:** Adjust is procedure which again built the heap so that edge

with minimum cost from the remaining edge will be selected as root.

1. This procedure should be applied until the heap becomes empty.

#### Prims Algorithm:

* Prims algorithm is **Greedy Method.**
* Prims algorithm the minimum weighted edges associated with particular vertices are considered one by one.
* The algorithm will start with a tree that includes only a minimum cost edge of G.
* Then edges are added to this tree one by one .
* The next edge (i, j) to be added in such way that i is a vertex already included in the tree, j is a vertex not yet included and the cost of (i,j) is minimum among all edges.
* To determine this edge (i, j) efficiently we associate with each vertex j

not yet included in the tree and find j in such way that the new edge should contains minimum weight.

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:**

Step 1: Select any connected vertices with min weight.

Step 2: Select unvisited vertex which is adjacent of visited vertices with min weight.

Step 3: Repeat step 2 until all vertices are visited and we will get the spanning tree.

* **Optimal Storage on Tapes:**

There are n programs that are to be stored on a computer **tape of length l.** Associated with each program i is a length li, 1<=i<=n

All the programs can be stored on a tape if and only if the sum of lengths of the programs is at most l. such that

l >= ∑ li

The tape is a **sequential device** so every program on the tape is stored one after the other.

If a program pi is to be retrieved from the tape, we first have to retrieve the programs p1,p2……pi-1

Therefore the total time needed to retrieve pi = time needed to retrieve p1+ time needed to retrieve p2 + ………+ time needed to retrieve pi-1.

The arrangement of programs on the tape is expected to be done in such a way that the **total retrieval time of each program from the tape is minimum.**

If we assume that all the programs are accessed only once, then the **mean retrieval time (MRT) is**

n

MRT= ∑ d(pi) / n i=1

Where d(pi) is the retrieval time of program pi

* **Dijkstra’s shortest path algorithm:**
* Dijkstra's algorithm is an algorithm for finding the shortest paths between

nodes in a graph, which may represent, for example, road networks.

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can be then be assigned weights which may be either distance between the two cities connected by the edge or the average time to drive along that section of highway.

**Huffman Code:**

Another application of binary tree with minimal weighted external path length is to obtain an optimal set of codes for messages M1…..Mn+1 . Each code is binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using decode tree.

* Huffman coding is a lossless data compression algorithm.
* Compression technique
* Reducing the size of the data
* Used to store data in compresses form
* When data is sent over the network then data is compresses and then transmitted to reduce the cost of transmission.

#### All pairs Shortest Path (Floyd- Warshall Algorithm)

* Dynamic programming approach is being followed in **Floyd Warshall Algorithm** to solve **all pair shortest path problem**.
* Floyd-Warshall Algorithm is used to find **all pair shortest distances** using Dynamic Programming method.
* **Floyd Warshall’s** Algorithm is used for solving all pair shortest path problems. It means the algorithm is used for finding the shortest paths between all pairs of vertices in a graph.
* Floyd Warshall’s Algorithm can be applied on **directed graph.**

##### Bellman Ford Algorithm:

* Bellman ford algorithm provides solution for **Single source shortest path**

problems.

* Which algorithm is used to find single source shortest path when the graph contains negative weighted edges?
* Bellman ford algorithm is used to **indicate whether the graph has negative weight cycles or not.**

Bellman Ford Algorithm can be applied for **directed and weighted graphs**

##### Longest Common Subsequence:

* Longest Common Subsequence (LCS) is a good example of the technique of

dynamic programming.

* It is one of the **string matching problem**.
* This technique is also used in file comparison. We can compare two different versions of the same file to determine what changes have been made to the file. This comparison is made by line by line.

**Application :** For **DNA matching** LCS with dynamic programming is used.

##### Travelling Salesman problem:

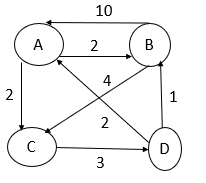
* Travelling Salesman Problem is solved by using dynamic programming method.
* It consists of Salesman and set of cities. The salesman travels along all the cities and cost of travelling should be minimum.
* The salesman has to visit each city starting from one city and returning to same

city. The main challenge to this is minimize the total distance in this trip.

We have find out we should have minimum cost/distance in this trip.

Let G=(V,E) be a directed graph on n vertices. The cost of tour is sum of cost of

the edges on the tour. Look at the figure:



A,B,C,D are four cities. These are connected by different roads. The distance of the road are written (in kms) in the figure. A salesman starts his tour at A and travels along the roads in such way that he visits every other **city only once** and comes back to A. The distance travelled during his tour is minimum.

We observe different tours like

A->C->D->B->A , A->B->C->D->A

out of which in the first tour distance travelled is 16 km and in that of second is 11 km minimum.

Naturally the salesperson prefers the tour with minimum distance travelled.

* This is **minimization problem.** It is the problem of finding minimum distance. g(i, s )=min {cij + g(j, s-{j})
* It gives shortest path traversed by salesperson and path includes every vertex.

Shortest path is always **cycle.**

* The traveling salesman problem involves visiting each city only **once.**

##### **Depth First Search:**

This is traversal technique. 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.

It employs the following rules.

**Rule 1** − **Visit** the adjacent unvisited vertex. Mark it as visited. Display it. **Push** it

**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.**

* DFS uses **LIFO** technique. In DFS, how many times a node is visited? **Once**
* The Data structure used in standard implementation of Depth First Search is

**Stack**. That is use of stack when we traverse the graph.

##### A vertex is pushed on the stack when it is visited first time and it is poped from the stack when explored.

* A vertex which itself is visited and all its adjacent vertices are also visited is said to be **explored**.

##### Note:

* The Depth First Search traversal of a graph will result into **tree**.
* Depth first Search is equivalent to **Pre-order Traversal** in the binary Trees.
* The maximum number of edges in the tree generated by DFS from an undirected graph with n vertices is **n-1.**
* DFS uses **backtracking** and visits all the vertices in the graph.
* Time Complexity of DFS is? (V – number of vertices, E – number of edges)

##### O(V+E)

* The output generated by BFT or DFT of a directed graph can be either **tree or forest. (Forest is a collection of disjoint trees.)**

##### **Breadth First Search:**

* In breadth first search we start at vertex V and mark it as having been reached (visited). The vertex V is at this time said to be unexplored. **A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices** adjacent from it. All unvisited vertices adjacent from V are visited next. Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as **queue**.
* BFS is a graph search algorithm that begins at **root node** and **explores all the neighboring vertices**. Then for each of those nearest nodes it explores their unexplored neighbor nodes.
* The Data structure **QUEUE** used in standard implementation of Breadth First Search.
* In BFS, how many times a node is visited? **Once**
* Breadth first Search is equivalent to **Level-order Traversal** in the binary Trees.

##### Application of Breadth First Search

* 1. Finding shortest path between two nodes or shortest path from Source to every vertex in the graph.
  2. GPS Navigation systems: Breadth First Search is used to find all neighboring locations.
  3. Peer to Peer Networks. In Peer to Peer Networks, Breadth First Search is used to find all neighbor nodes. (DFS is not used in peer to peer network).

##### Finding shortest distance between every pair vertices is not an application of BFS.

* The tree generated by BFS traversal from a source node A in an unweighted, connected, undirected graph is used to find **the shortest path from S to every vertex in the graph.**
* Given two vertices in graph P, Q. Which of the following traversal is used to find path from P to Q?
* We can use both BFS and DFS traversal method.
* The output generated by Breadth First Traversal or Depth First Traversal of a directed graph can be either **tree or forest. (Forest is a collection of disjoint trees.)**
* **Topological Sorting:**
* A topological sort or topological ordering of a directed acyclic graph G=(V,E) is a linear ordering of its vertices such that for every edge <u, v>, **u comes before v in the linear ordering.**
* **Graph should be directed acyclic graph**.(DAG)
* For instance the vertices of the graph may represent tasks to be performed and the edges may represents constraints that one task must be performed before another.
* The topological sorting is a valid sequence for the tasks.
* A topological ordering is possible if and if the graph has no directed cycles, that is, if it is **directed acyclic graph**.(DA
  + **Algorithm for topological sort:**

1. Call DFS(V,E) to compute finishing time for each vertex V.
2. When each vertex is finished, insert into front of a linked list.
3. Return linked list of vertices.
4. Topological sort can be applied to which of the following graphs?

**Directed Acyclic Graphs (DAG)**

1. Topological sort can be implemented by Using **both Depth and Breadth First Search.**

* **Articulation Point and Bridge edge:**
* **Articulation Point:**

A **vertex** 'V' in an undirected connected graph G is an articulation point iff deletion of vertex V along with the edges incident to it disconnects the graph into 2 or more non-empty components.

**Backtracking**

* Backtracking is **design strategy/ approach** used to solve puzzles or problems include such as eight queens puzzle, four queens puzzle, Sudoku.
* Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
* It removes the solutions that doesn't give rise to the solution of the problem based on the **constraints** given to solve the problem.
* **n-queen Problem:**

In n queens problem there is n × n chessboard and we have to place n queen on

n × n chessboard so that no two queens should attack. That is no two queens should be on the same row, same column and same diagonal. This is generalized problem. This is generalized problem to understand this we consider 4 queen problem. In **three** directions the queens can attack each other.

* **4 - queen Problem:**

Given a 4×4 chessboard and we have to arrange 4 queens in such way that

No two queens attack each other. No two queens should be on the same row and column and on the same diagonal.

* **Explicit Constraint:**

These are the rules which restrict each xi to take value from a given set.

* + Si = {1, 2, 3, 4} or 1 ≤ i ≤ 4
* **Implicit Constraint:**

1. No two queens should be on the same row.
2. No two queens should be on the same column.
3. No two queens should be on the same diagonal.

* This backtracking algorithm is implemented by constructing a tree of choices called as **State-space tree.**
* The problem of placing n queens in a chessboard such that no two queens attack each other is called as **n queen problem.**

**Find correct solutions for 4-queens problem**

* + **Graph Coloring Problem:**
* Let G=(V,E) be a graph and 'm' be the given positive integer. We want to discover whether nodes of G can be colored in a such way that **no two adjacent nodes have the same color** yet only **m** colors are used. This is termed the **m-colorability decision problem.**
* If d is degree of given graph, then it can be colored with d+1 colors.
* The m- colorablity **optimization problem** asks for the **smallest integer m** for which the graph G can be colored. This integer is referred as **chromatic number of graph.**
* The below graph can be colored with three colors 1, 2 and 3.

#### Hamiltonian Cycles:

* **Hamiltonian circuit problem** is solved by using backtracking.
* Let G =(V,E) be a connected graph with n vertices. A Hamiltonian cycle (Suggested by Sir William Hamilton) is a round-trip path along n edges of G That visits every vertex once and returns to its starting position.
* A Hamiltonian cycle is a closed loop on a graph where every node (vertex) is **visited exactly once**. A loop is just an edge that joins a node to itself; so a Hamiltonian cycle is a path traveling from a point back to itself, visiting every node on route.
* Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. **A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not.**
* It is used in various fields such as Computer Graphics, electronic circuit design.

Live node is a generated node for which all of the children have not been generated yet

* **Sum of subset Problem:**

Suppose we are given n distinct positive numbers w1,w2, ….. wn usually called as weights and an **integer number m**.

We have to find out all subsets of W ={w1,w2,….wn} such that the sum of subset of elements is **equal to m** or desire to find all the combinations of these numbers whose sums are m.

This is called **sum of subset problem**.

The sum of subset problem is used for checking for the presence of a subset that has **sum of elements equal to a given number m** and printing true or false based on the result.

X Y Z

Where x=sum of weights included y=index of next object considered.

z=total remaining weight to be considered.

**Procedure for generating state space tree:**

Initially for root node of the state space tree. x=0 No weight included yet

y=1 1st object weight to be considered.

z= ∑ wi Remaining weight is equal to total weight since none of the weight included

The node indicating inclusion of next object will have

x=x + wi, y=i +1 and z= z - wi

A solution node is said to obtained if x=m ( whatever may be the value of y and z) The branch is **discarded** if any one of the following is true.

x+wi+1 > m

y+1 > n

x + z <m

**LCBB (Least Cost branch and Bound)**

* + **Types of Searching:**

1. **LIFOBB:**

**I**n data structure such as **stack** we use the concept of last in first out(LIFO). LIFOBB uses depth first search(**DFS**).

1. **FIFOBB:**

**I**n data structure such as **queue** we use the concept of first in first out(FIFO). FIFOBB uses breadth first search **(BFS)**.

* + **Basic Terminologies used in Branch and Bound:**
    1. **Live node:** The node which has been generated **but none of its children nodes have been generated** in state space tree is called Live node.
    2. **Dead node:** The node that is not to be expanded or explored any further. All

**children of a dead node have already been expanded** is called dead node.

* + 1. **E-node:** The node whose children are currently being **explored** is called E- node.
  + **Least Cost search (LC search):**
* This search is preferable over FIFOBB and LIFOBB.
* The method for solving a problem by LCBB is same as FIFOBB. The only difference is the way for poping the node from the queue.
* In FIFOBB we **sequentially remove the nodes from the queue** whereas in LCBB the **node which has minimum value of c^ is considered for been exploring further.** This makes mechanism faster.
* It is faster than FIFOBB and LIFOBB since it concentrates on that path which has maximum probability of reaching the optimal solution.
* **We can use fixed tuple size formulation or variable tuple size formulation to generate SST.**

**Live node** is a generated node for which all of the children have not been generated yet.

**E-node** is a live node whose children are currently being generated or explored.

**Bounding functions** are used to kill live nodes to make them dead nodes.

* **Least Cost search (LC search):**
* This search is preferable over FIFOBB and LIFOBB.
* The method for solving a problem by LCBB is same as FIFOBB. The only difference is the way for poping the node from the queue.
* In FIFOBB we **sequentially remove the nodes from the queue** whereas in LCBB

the **node which has minimum value of c^ is considered for been exploring further.**

This makes mechanism faster.

* It is faster than FIFOBB and LIFOBB since it concentrates on that path which has maximum probability of reaching the optimal solution.
* **We can use fixed tuple size formulation or variable tuple size formulation to generate SST.**

**Solve the**

* **Polynomial and Non Polynomial Algorithm:**
* The decision problem that can be solved in short time **(polynomial)** are called

**Polynomial algorithm.**

e.g. Search O(log n), Sort, String editing

* The decision problem that require **vast amounts of time (non-polynomial)** because of the numerous possibilities are called as **non-polynomial algorithm**.

e.g. TSP and Knapsack Problem.

**Classification of Polynomial time algorithm**

**Nondeterministic Polynomial:**

* Non deterministic polynomial algorithm **(NP)** is divided into **NP hard and NP complete.**
* **NP class** is the class of decision problems that can be solved by non-deterministic polynomial algorithms.

**Explanation:** NP problems are called as non-deterministic polynomial problems. They are a class of decision problems that can be solved using NP algorithms.

Problems which can be solved using polynomial time algorithms are called as **tractable (easy).**

Problems that can be solved using super polynomial time algorithms are called **intractable (hard)**

1. **NP Hard:**

Non deterministic polynomial time hard in computational theory is a class of problems that are informally at least as hard as the hardest problems in NP.

e.g. of NP-hard problem is the decision sum of subset problem.

1. **NP Complete:** NP-complete problem, any of a class of computational problems for which **no efficient solution algorithm** has been found. Many significant computer-science problems belong to this class—e.g. graph-covering problems.

This problem require **vast amount of time** to execute.

NP complete problems are defined as the hardest problems in NP.

Let S be a set of algorithms. An algorithm A is said to be **S-hard** if A is harder than all members of S .