1 What is an Algorithm? Explain its characteristics in detail.

An algorithm is a step-by-step procedure or set of rules designed to perform a specific task or solve a particular problem. It is a finite sequence of well-defined, unambiguous instructions that, when executed, produces a desired output from a given set of inputs. Algorithms serve as the foundation for computer programs and are essential in various fields such as computer science, mathematics, and engineering. Here are the key characteristics of algorithms:

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each instruction must be unambiguous.
4. Finiteness: The algorithm will terminate after a finite number of steps.
5. Effectiveness: Every instruction must be sufficiently basic.

2 Explain the general method of Divide and Conquer algorithm and specify its Applications?

The divide and conquer algorithm is a general problem-solving strategy that involves breaking down a problem into smaller sub-problems, solving each sub-problem independently, and then combining the solutions of the sub-problems to solve the original problem. This approach typically follows three steps: divide, conquer, and combine.

1. Divide: The original problem is divided into smaller, more manageable sub-problems. This step continues recursively until the sub-problems become simple enough to be solved directly.

2. Conquer: Each sub-problem is solved independently. This is often the recursive application of the divide and conquer strategy. The base case of the recursion involves solving the simplest form of the problem without further subdivision.

3. Combine: The solutions to the sub-problems are combined to obtain the solution to the original problem. This step is crucial in ensuring that the solutions to the sub-problems collectively contribute to solving the overarching problem.

The divide and conquer approach is often used to solve problems efficiently, especially when the problem exhibits overlapping sub-problems and optimal substructure. This strategy is employed in various algorithms, and some of its applications include:

1. Sorting Algorithms:

- Merge Sort: The divide and conquer strategy is used to recursively divide an array into two halves, sort each half, and then merge the sorted halves to obtain the final sorted array.
- QuickSort: The array is partitioned into two sub-arrays, and the sorting process is applied recursively to each sub-array. The combined results give the sorted array.

2. Searching Algorithms:

- Binary Search:The divide and conquer approach is employed to search for a target element in a sorted array by repeatedly dividing the array into halves.

3. Matrix Multiplication:

- Strassen's Algorithm: A more efficient way to multiply matrices by breaking down the matrix multiplication into sub-multiplications and combining the results.

4. Closest Pair Problem:

- Closest Pair of Points: Given a set of points in a plane, the divide and conquer algorithm can be used to find the pair of points with the smallest distance between them.

5. Fast Fourier Transform (FFT): Divide and conquer is utilized in algorithms for efficient computation of the Fourier transform, such as the Cooley-Tukey FFT algorithm.

6. Multiplication of Large Integers: Karatsuba algorithm is an example that applies the divide and conquer strategy to efficiently multiply large integers.

The divide and conquer paradigm is a powerful tool for designing algorithms that can efficiently solve a wide range of problems by breaking them down into smaller, more manageable parts.

3 Discuss about Pseudo Code conventions for expressing algorithms

Pseudo code is an artificial and informal language that helps programmers to develop algorithms.

Pseudo code is a "text-based" detail design tool.

Example:

```
If student's grade is greater than or equal to 60
Print "pass"
else
Print "fail"
```

Pseudocode consists of ,

1. Single line comments start with **//**

2. Multi-line comments occur between /* **and** */

3. Blocks are represented using brackets. Blocks can be used to represent compound statements or the procedures.

{
statements
}

4. Statements are delimited by **semicolon**.

5. Assignment statements indicates that the result of evaluation of the expression will be stored in the variable.

< variable > := < expression >

6. The Boolean expression 'x > y' returns **true** if x is **greater** than y, **else** returns **false**.

7. **if< condition >then< statement >**

8. This condition is an enhancement of the above 'if' statement. It can also handle the case where the condition isn't satisfied.

if< condition >then< statement1 >else< statement2 >

9. switch case (C or C++)

```
case {
:< condition 1 >: < statement 1 >
.....
.....
.....
:< condition n >: < statement n >
:default: < statement n+1 >
}
```

10. while loop

```
while< condition > {
statements;
}
```

11. do-while loop

```
repeat
statements;
until< condition >
```

12. for loop

```
for variable: = value1 to value2 {
statements;
}
```

13. input instruction

Read

14. Output instruction

Print

15. The name of the algorithm is < name > and the arguments are stored in the < parameter list >

Algorithm< name > (< parameter list >)

4 What do you mean by Performance analysis? Explain about various Asymptotic notations for time complexity analysis with neat diagrams.

Performance Analysis:

- Performance of an algorithm is a process of making evaluative judgment about algorithms.
- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

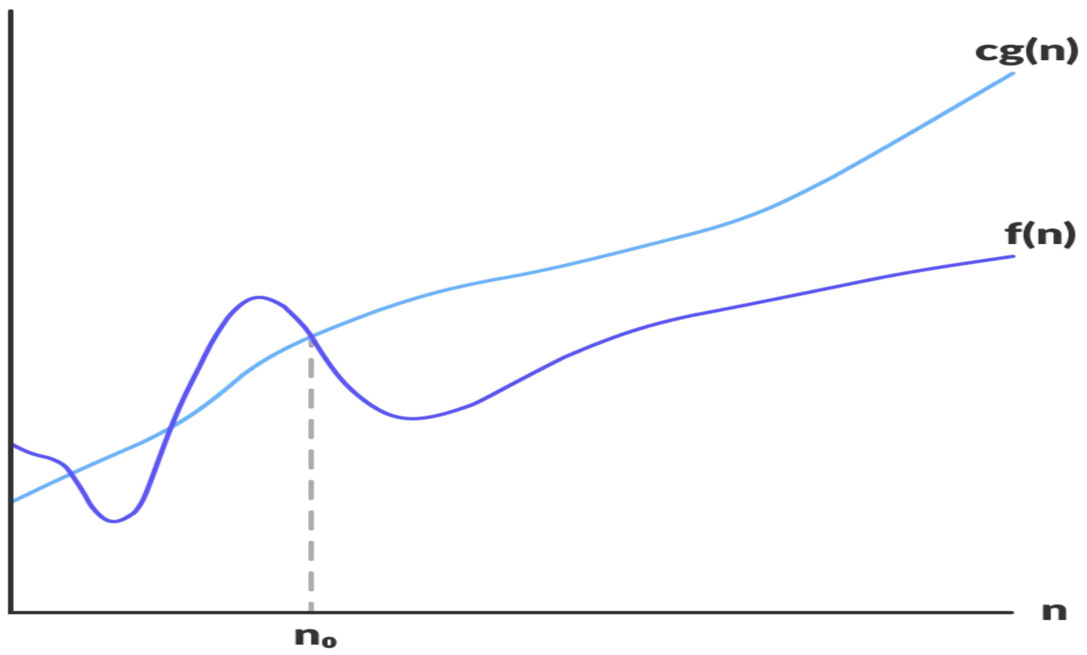
Asymptotic Notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- There are mainly three asymptotic notations
- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

- Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.
- For any value of n, the maximum time required by the algorithm is given by Big-O O(g(n)).
- A function f(n) belongs to the set O(g(n)) if there exists a positive constant c and n_0 such that ,

$$\mathbf{f(n) \leq c*g(n) \text{ for all } n \geq n_0}$$

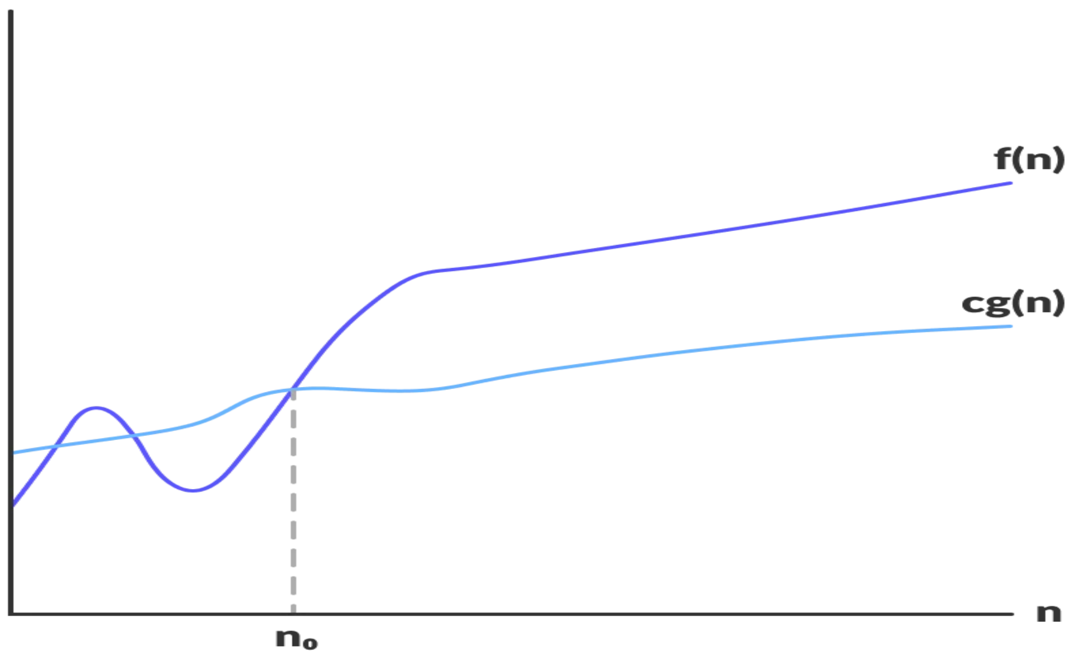


f(n) = O(g(n))

Omega Notation (Ω-notation)

- Omega notation represents the lower bound of the running time of an algorithm.
- For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).
- A function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c and n_0 such that,

$$\mathbf{f(n) \geq c * g(n) \text{ for all } n, n \geq n_0}$$



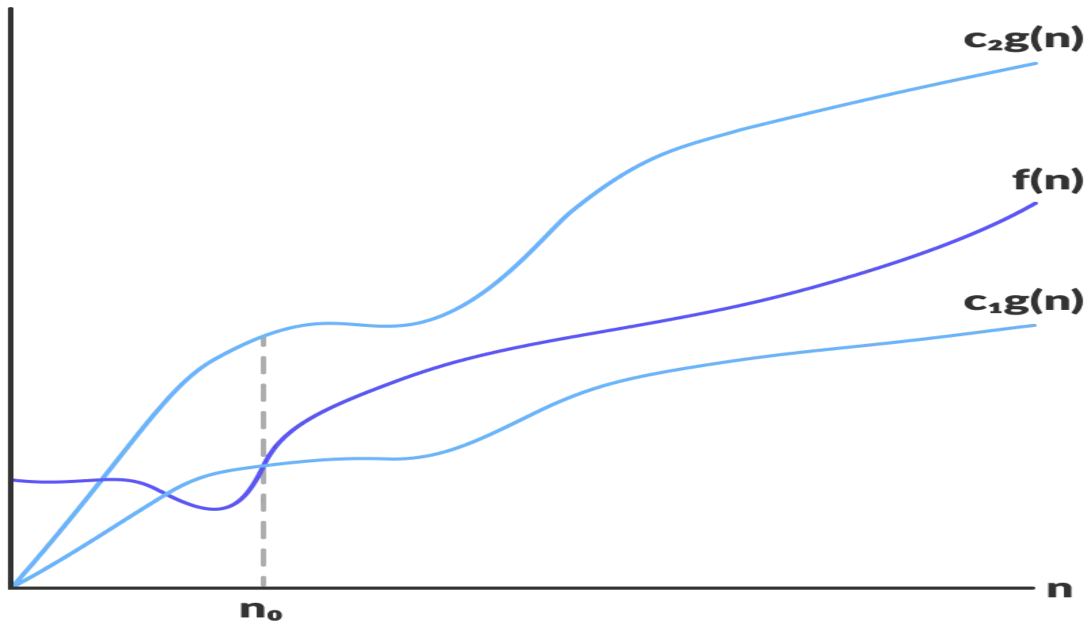
f(n) = Ω(g(n))

Theta Notation (Θ-notation)

- Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm.
- For any value of n, the average time required by the algorithm is given by Theta **Θ(g(n))**.
- A function f(n) belongs to the set Θ(g(n)) if there exists a positive constant c_1, c_2 and n_0 such that,

$$\mathbf{c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0}$$

- If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n0$, then f(n) is said to be tight bound.



$f(n) = \Theta(g(n))$

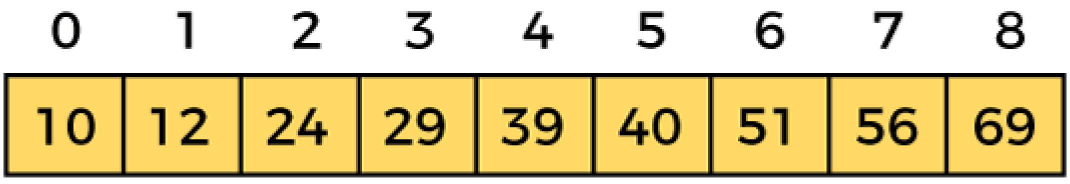
5 Write Binary Search algorithm with suitable example and analyse its time complexity?

- searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

The basic steps to perform Binary Search are:

- Sort the array in ascending order.
- Set the **low index** to the first element of the array and the **high index** to the last element.
- Set the **middle index** to the **average of the low and high** indices.
- If the element at the middle index is the target element, return the middle index.
- If the **target element is less** than the element at the middle index, set the **high index** to the **middle index – 1**.
- If the **target element is greater** than the element at the middle index, set the **low index** to the **middle index + 1**.
- Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

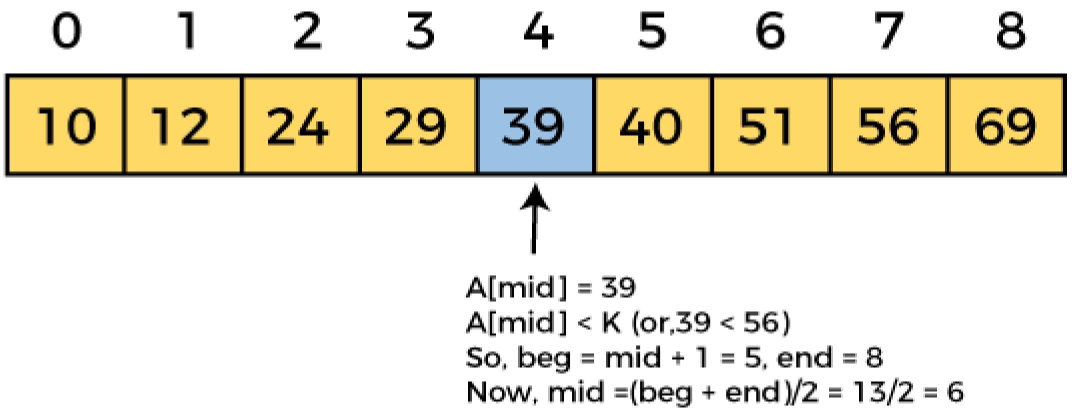
1. mid = (beg + end)/2

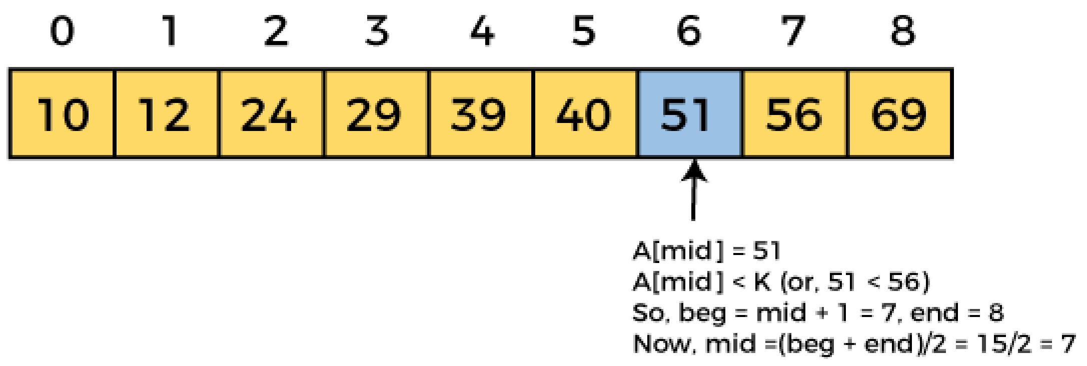
So, in the given array -

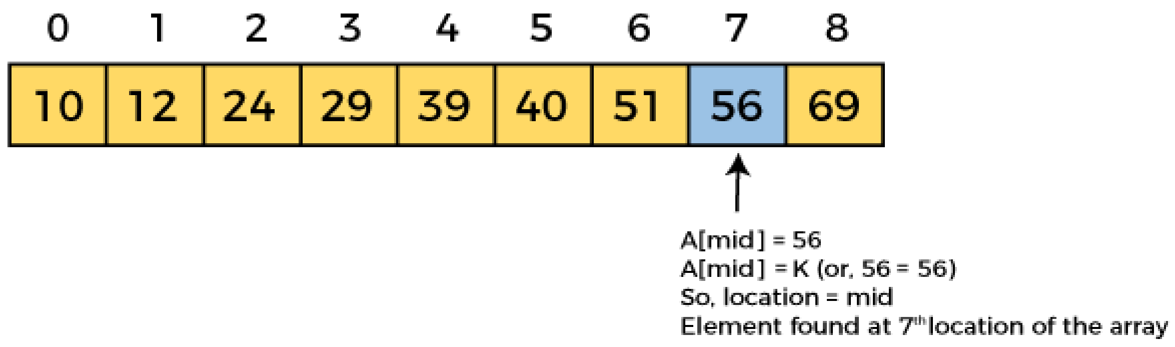
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.


0
1
2
3
4
5
6
7
8
10
12
24
29
39
40
51
56
69
A[mid] = 39
A[mid] < K (or,39 < 56)
So, beg = mid + 1 = 5, end = 8
Now, mid =(beg + end)/2 = 13/2 = 6


0
1
2
3
4
5
6
7
8
10
12
24
29
39
40
51
56
69
A[mid] = 51
A[mid] < K (or, 51 < 56)
So, beg = mid + 1 = 7, end = 8
Now, mid =(beg + end)/2 = 15/2 = 7



Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search Algorithm can be implemented in the following two ways

- Iterative Method
- Recursive Method

Iterative Method:

```
binarySearch(arr, key, low, high)
    repeat till low <= high
        mid = (low + high)/2
            if (key == arr[mid])
            return mid
            else if (key > arr[mid])        // key is on the right side
              low = mid + 1
            else                            // key is on the left side
              high = mid - 1
```

Recursive Method:

```
binarySearch(arr, key, low, high)
      if low > high
         return False
      else
         mid = (low + high) / 2
            if key== arr[mid]
            return mid
      else if key > arr[mid]                          // key is on the right side
            return binarySearch(arr, key, mid + 1, high)
      else                                       // key is on the left side
            return binarySearch(arr, key, low, mid - 1)
```

Time Complexity:

- Minimum time is 1 and Maximum time is logn

Best case = **O(1) ,** Element to be searched is at Root

Average case= **O(logn),** All possible cases of search

Worst case= **O(logn),** Element to be searched is not in the list

6 Illustrate the tracing of Quick sort algorithm for the following set of numbers: 25, 10, 72, 18, 40, 11, 64, 58, 32, 9. Specify best case and worst case time complexity ofthe Quick sort algorithm.

$\overset{i}{25}$ 10 72 18 40 11 64 58 32 $\overset{j}{9}$

Pivot=25

i>P→stop
j<P→stop

$\overset{i}{25}$ 10 72 18 40 11 64 58 32 $\overset{j}{9}$

i stop

i>P stop
25>25 →Yes
10>25 →NO
72>25 →Stop

j<P→stop
9<25→stop

Swap 72 and 9

25 10 9 18 40 11 64 58 32 72

start start end end

9>25→NO
18>25→NO
40>25→stop

72<25→NO
32<25→NO
58<25→NO
64<25→NO
11<25→Yes
↓
Stop

25 10 9 18 11 40 64 58 32 72

Start end Start
end

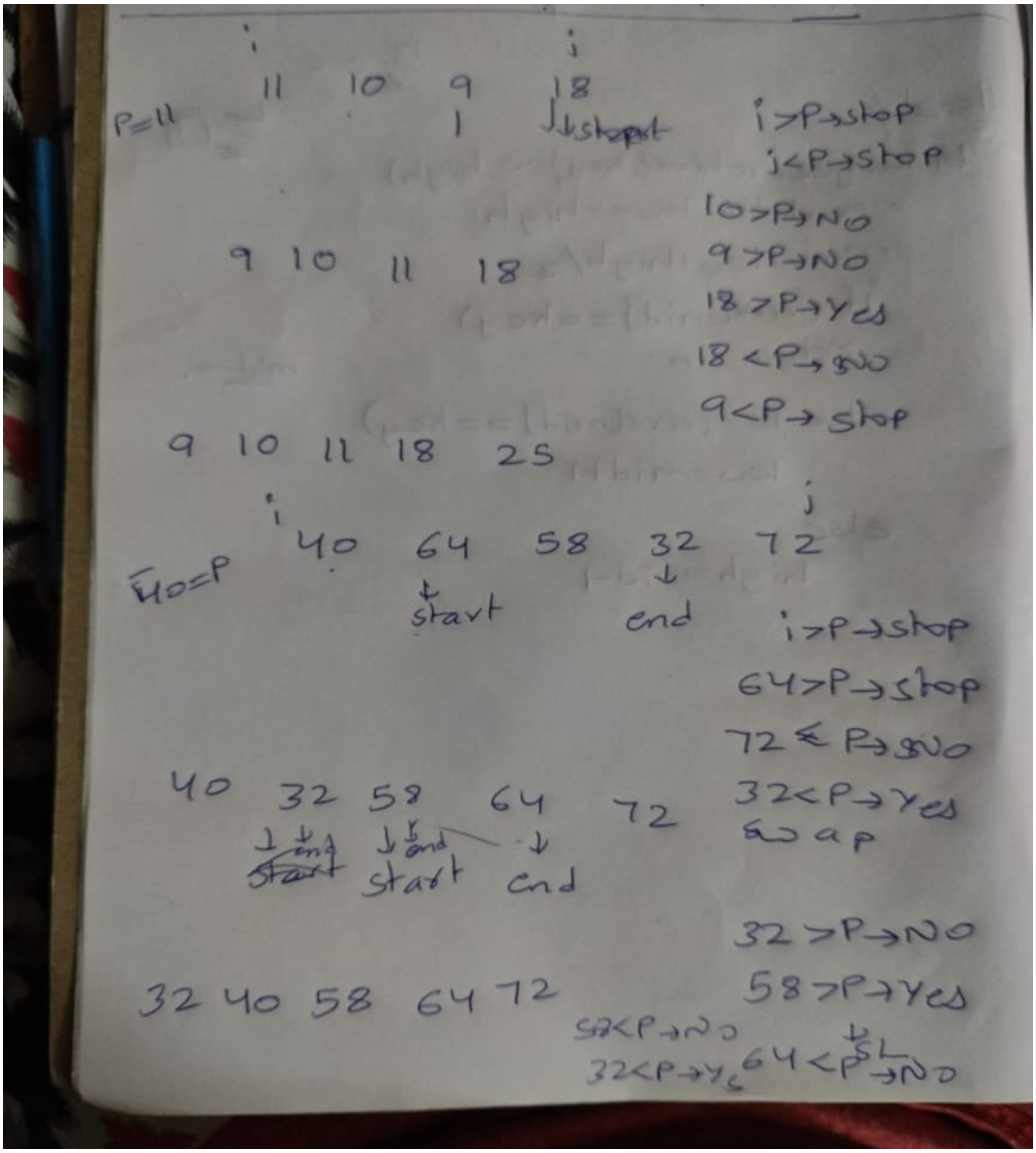
11>25 →No

40>25→Yes→stop

40<25

11<25→Stop

11 10 9 18 $\boxed{25}$ 40 64 58 32 72



i j

11 10 9 18

P=11 | j↓ ~~stop~~ start

i>P→stop

j<P→stop

10>P→NO

9 10 11 18

9>P→NO

18>P→Yes

18<P→NO

9<P→stop

9 10 11 18 25

i j

40 64 58 32 72

40=P

↓ start ↓ end

i>P→stop

64>P→stop

72 ≮ P→NO

40 32 58 64 72

↓ ↓ end ~~start~~ ↓ ↓ end start ↓ end

32<P→Yes swap

32>P→NO

32 40 58 64 72

58>P→Yes

58<P→NO

32<P→Yes

64<P→NO

- Best Case : O(n log n)
  Average Case : O (n log n)
- Worst Case : O(n^2)
- Space Complexity of quick sort is O(n)

```
QUICKSORT (alist, start, end)
{
if end - start > 1:
        p = partition(alist, start, end)
        quicksort(alist, start, p)
        quicksort(alist, p + 1, end)
}
PARTITION (alist, start, end)
{
    pivot = alist[start];
    i = start + 1;
    j = end;
while (i<j):
{
        while (alist[i] <= pivot):
            i = i++;
        while (alist[j] >= pivot):
            j = j--;
        if i <= j:
            alist[i], alist[j] = alist[j], alist[i];
}
alist[start], alist[j] = alist[j], alist[start];
return j;
}
```

7 Give an algorithm for Merge sort. Derive it’s time complexity.

```
1.  MERGE_SORT(arr, beg, end)
2.
3.  if beg < end
```

```
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0, /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

```

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most 2T $\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So T (n) = 2T $\left(\frac{n}{2}\right)$ + n...equation 1

Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that need to be copied, and there will be no comparison.

Putting n=$\frac{n}{2}$ in place of n in ....equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$..................equation2

Put 2 equation in 1 equation

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$

= $2^2 T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$

T (n) = $2^2 T\left(\frac{n}{2^2}\right)$+2n.....................equation 3

Putting n= $\frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$.....................equation4

Putting 4 equation in 3 equation

T (n) = $2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

T (n) =$2^3\ T\left(\frac{n}{2^3}\right) + n + 2n$

T (n) = $2^3\ T\left(\frac{n}{2^3}\right) + 3n$..........................equation5

From eq 1, eq3, eq 5.......we get

T (n) = $2^i\ T\left(\frac{n}{2^i}\right) + in$............................equation6

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right)$=0

n = 2^i

Apply log both sides:

log n=$\log_2 i$
logn= i log2
$\frac{\log n}{\log 2}$ =i

$\log_2 n = i$

From 6 equation

$$T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \,.\, n$$

$$\mathbf{= T(n) = n.\log n}$$

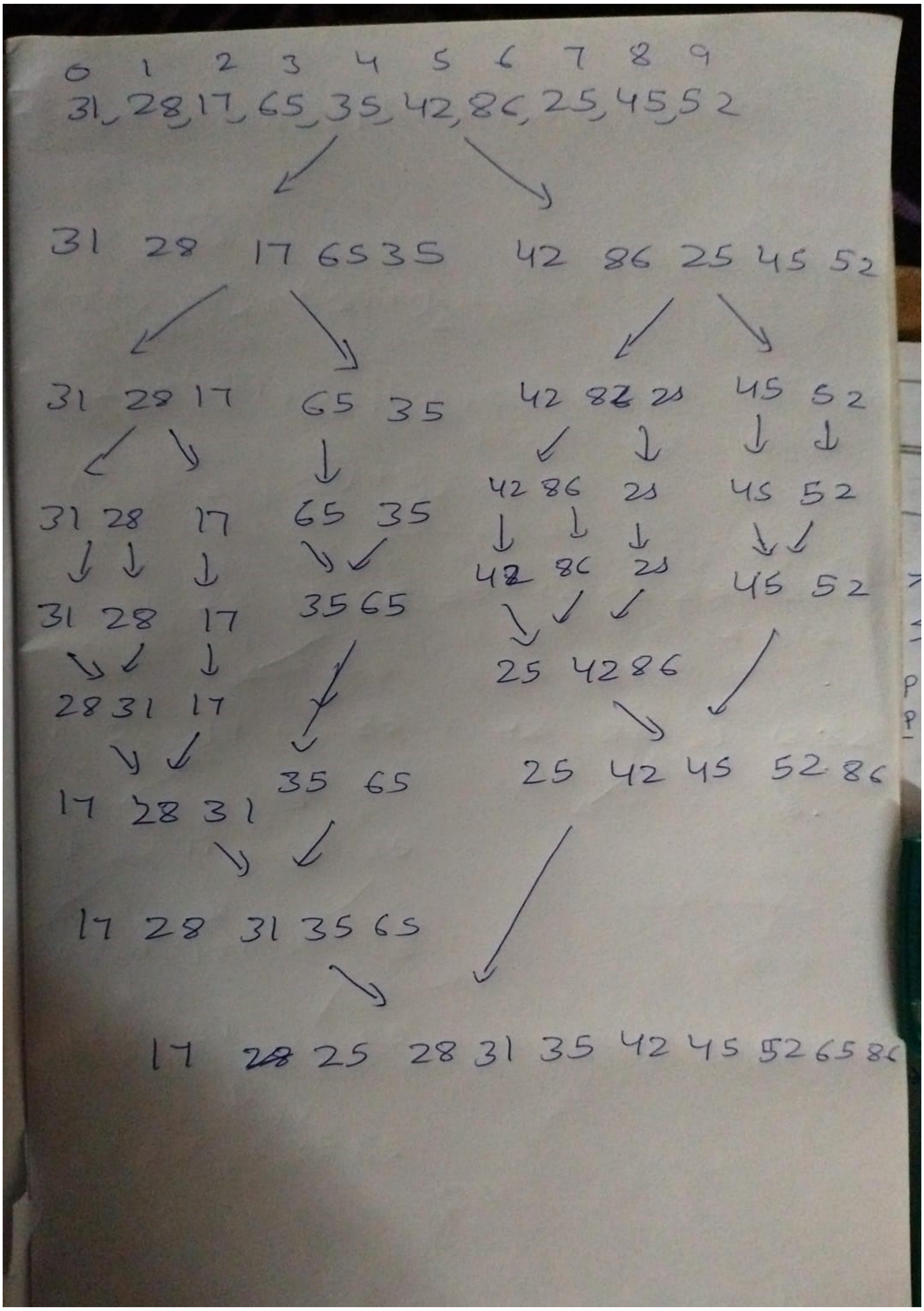
Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

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 nor 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)**.

8 Perform Merge sort on the array of elements a[1:10] ={31,28,17,65,35,42,86,25,45,52}.Represent tree of calls for Merge sort.



9 Define time and space complexity. Explain with examples

Time complexity

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

- To calculate the time complexity of an algorithm,
    - ➢ It requires 1 unit of time for Arithmetic and Logical operations
    - ➢ It requires 1 unit of time for Assignment and Return value
    - ➢ It requires 1 unit of time for Read and Write operations

Example

```
int sum(int a, int b)
{
a=5;
b=6;
c=a+b;
}
```

- It requires **1 unit of time for a=5, 1 unit of time for b=6 and 1 unit of time** to calculate **c=a+b**.
- Totally it takes **3 units** of time to complete its execution.

  so, T(n)=3

Space complexity

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

- ➢ **Instruction Space:** It is the amount of memory used to store compiled version of instructions.

- **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
- **Data Space:** It is the amount of memory used to store all the variables and constants.

Example

```
int sum(int a, int b)
{
a=5;
b=6;
c=a+b;
}
```

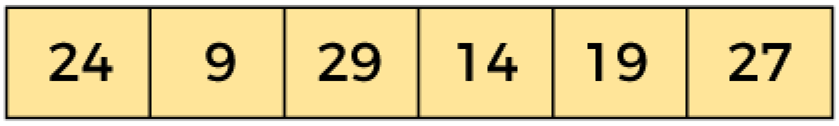
- It requires **1 word space** for a variable **a, 1 word space** for a variable **b** and **1 word space** a variable **c**.
- Totally it takes **3 word spaces** to complete its execution.

  so, S(n)=3

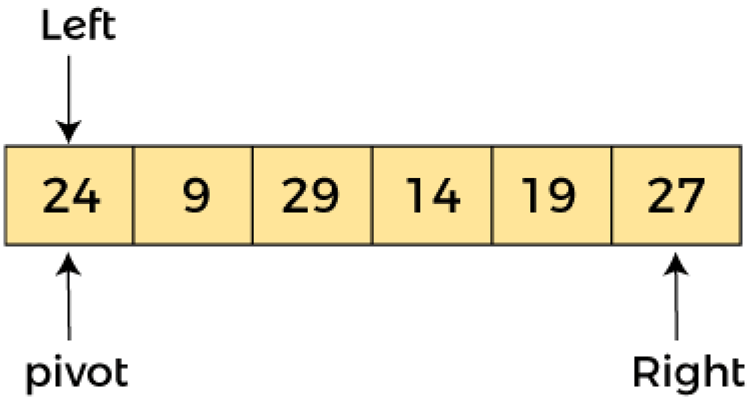
10 Explain Quick Sort Algorithm and analyse with suitable example

- In Quick sort algorithm, partitioning of the list is performed using following steps...
- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.

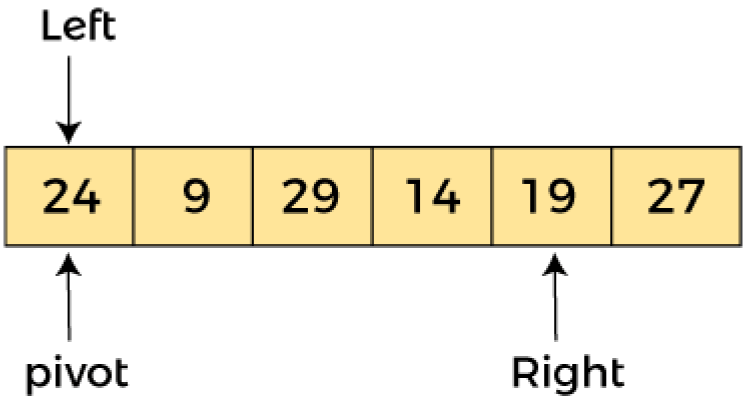
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.
- Let the elements of array are -



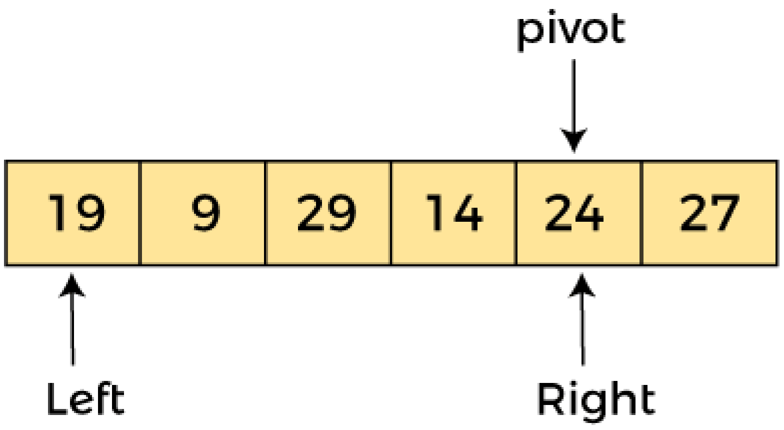
-
- In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.
- Since, pivot is at left, so algorithm starts from right and move towards left.



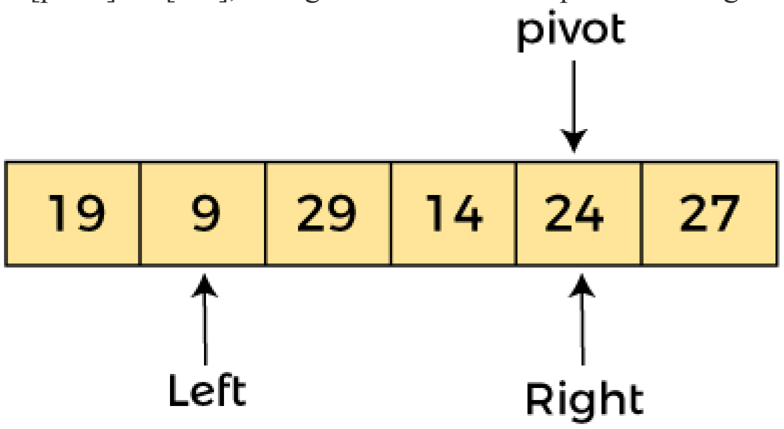
-
- Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



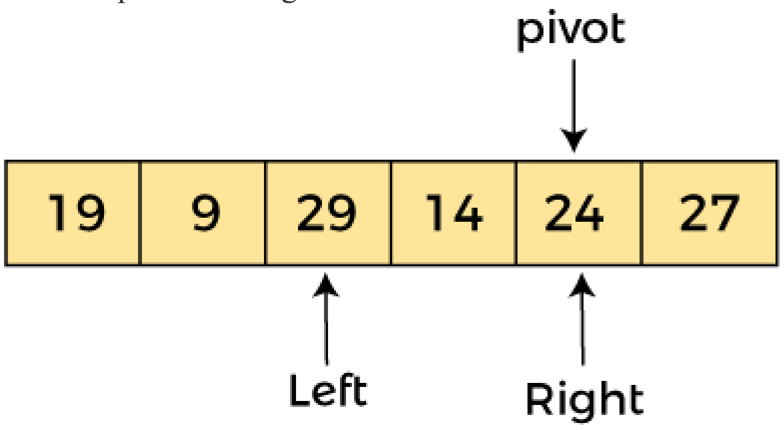
-
- Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
- Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



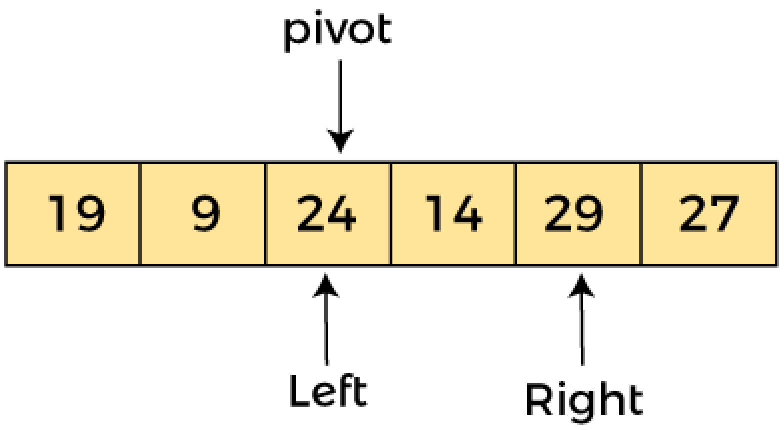
-
- Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
- As a[pivot] > a[left], so algorithm moves one position to right as -



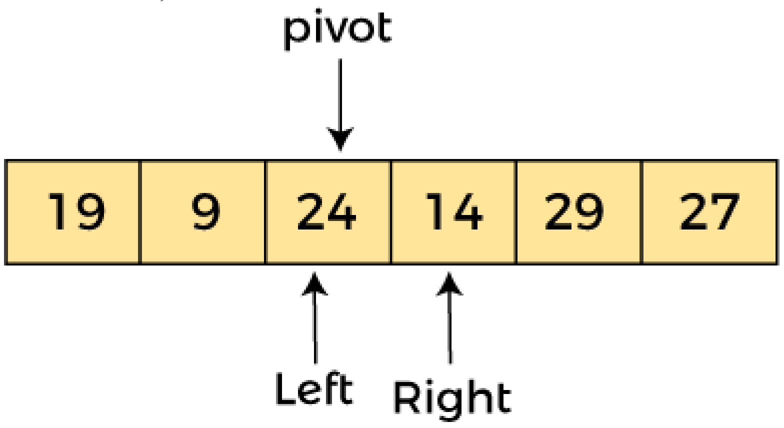
-
- Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



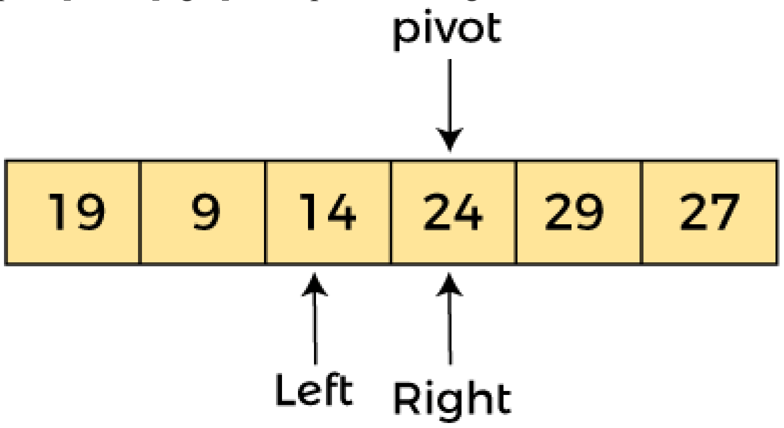
-
- Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



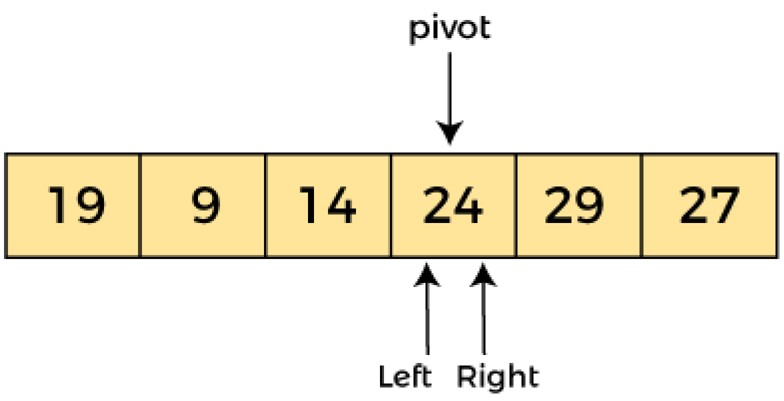
-
- Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



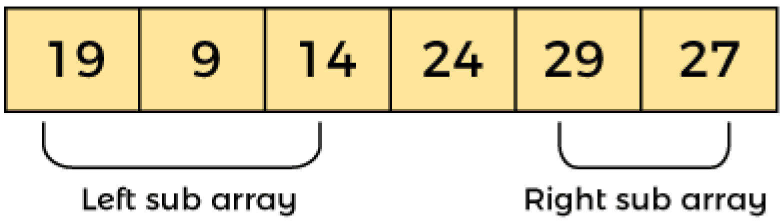
-
- Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -

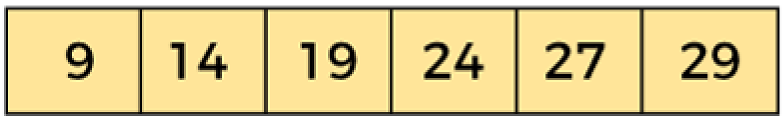


-
- Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



-
- Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
- Element 24, which is the pivot element is placed at its exact position.
- Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



-
- Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -
- 

```
QUICKSORT (alist, start, end)
{
if end - start > 1:
        p = partition(alist, start, end)
        quicksort(alist, start, p)
        quicksort(alist, p + 1, end)
}
PARTITION (alist, start, end)
{
    pivot = alist[start];
    i = start + 1;
```

```
   j = end;
while (i<j):
{
       while (alist[i] <= pivot):
          i = i++;
       while (alist[j] >= pivot):
          j = j--;
       if i <= j:
          alist[i], alist[j] = alist[j], alist[i];
}
alist[start], alist[j] = alist[j], alist[start];
return j;
}
```

Time Complexity:

- **Best Case : O (n log n)**

  Average Case : O (n log n)
- **Worst Case : O(n^2)**

- **Space Complexity of quick sort is O(n)**

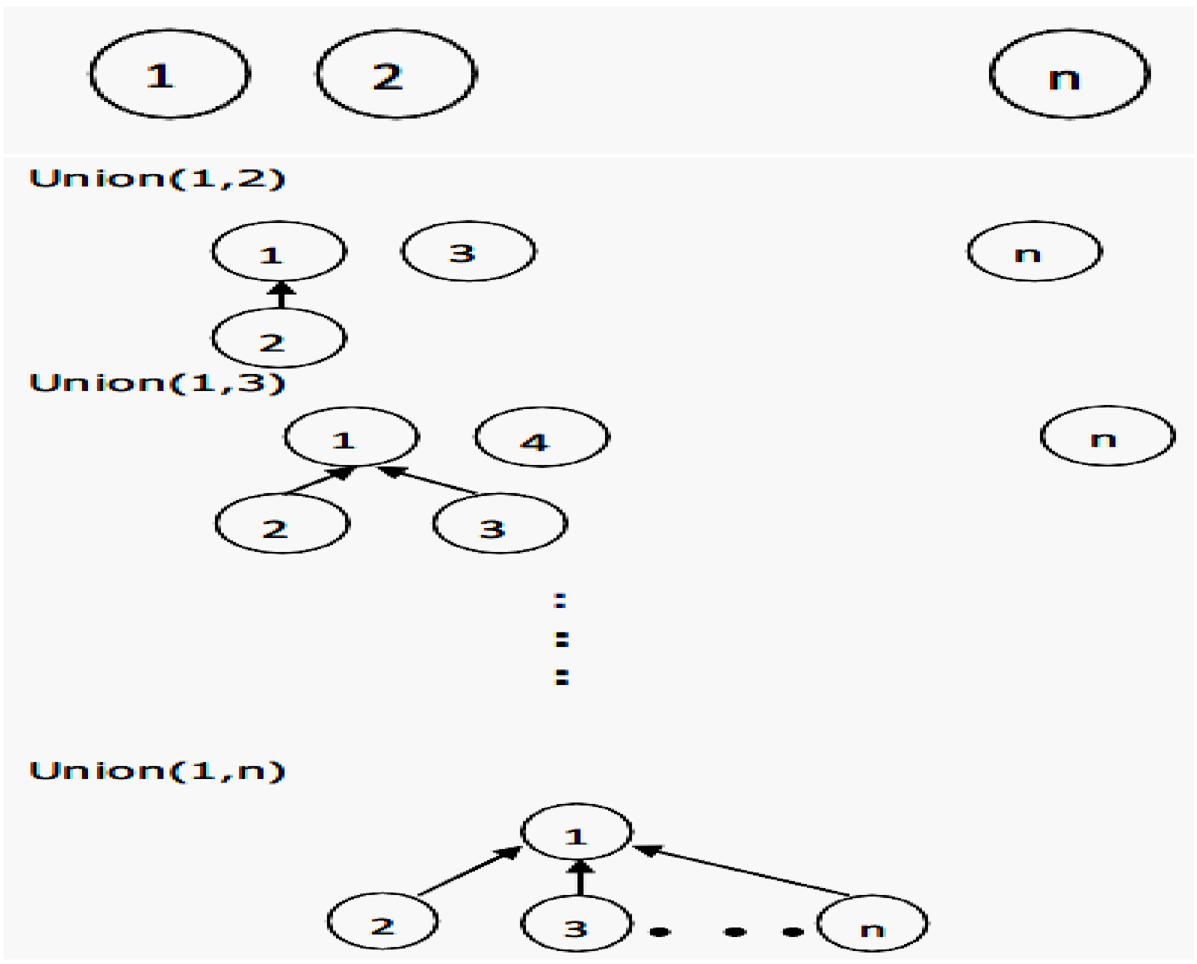
11 Explain the importance of weighted rule for Union operation with a suitable example.

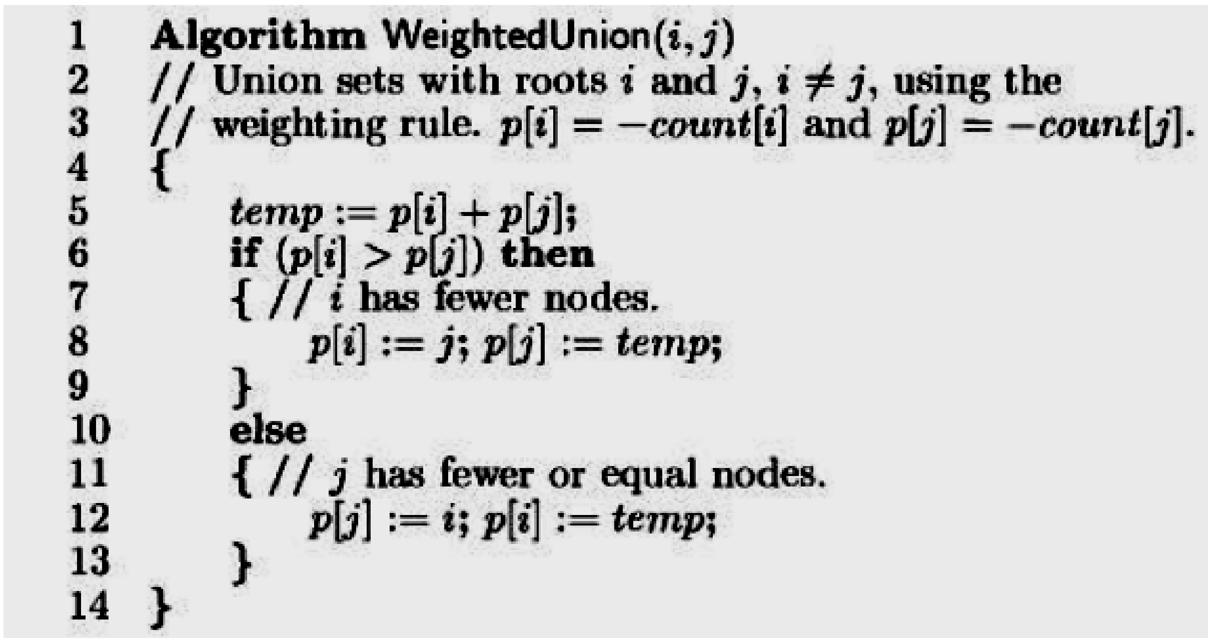
The weighted rule is an essential concept in the context of union operations in the context of Disjoint-Set Data Structures, particularly when implementing the Union-Find (or Disjoint-Set Union) data structure. The weighted rule is aimed at optimizing the union operation to ensure that the tree height (or rank) is minimized, leading to efficient and balanced structures. This is crucial for preventing the degradation of performance, especially when repeatedly performing union operations.
In a disjoint-set data structure, each element belongs to a set, and the structure maintains a collection of disjoint sets. The two fundamental operations are:
1. Find operation: Determines the representative (root) of the set to which an element belongs.
2. Union operation: Merges two sets into a single set.

Weighted Rule:
The weighted rule involves augmenting the data structure to keep track of the size or rank of each set. The key idea is to always attach the smaller tree to the root of the larger tree during the union operation. This helps in maintaining a balanced and efficient structure, preventing the tree from becoming excessively tall.
Importance of the Weighted Rule:
1. Preventing Unbalanced Trees:
   - Without the weighted rule, the union operation might lead to unbalanced trees, where one subtree becomes much larger than the other.
   - Unbalanced trees can result in longer find operation times, making the overall structure less efficient.
2. Optimizing Tree Height:

- By always attaching the smaller tree to the root of the larger tree, the weighted rule ensures that the height of the resulting tree is minimized.
- A balanced tree structure helps in achieving faster find operations, as the depth of the tree is limited.

3. Improving Overall Efficiency:
- The weighted rule contributes to the overall efficiency of the Union-Find data structure by preventing the worst-case scenario of having a linear chain of elements.
- A well-balanced structure ensures that find and union operations have an amortized time complexity of nearly constant time.





```
Algorithm WeightedUnion($i, j$)
// Union sets with roots $i$ and $j$, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if ($p[i] > p[j]$) then
    { // $i$ has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // $j$ has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}
```

12 Explain the General method of Backtracking and write the various applications of Backtracking.

General Method of Backtracking:

Backtracking is a general problem-solving algorithmic paradigm that explores all possible solutions to a problem and abandons a solution ("backtracks") as soon as it determines that the current solution cannot be extended to a valid one. It is particularly useful for solving problems where multiple decisions need to be made to reach a solution, and it systematically explores different options.

Terminology:

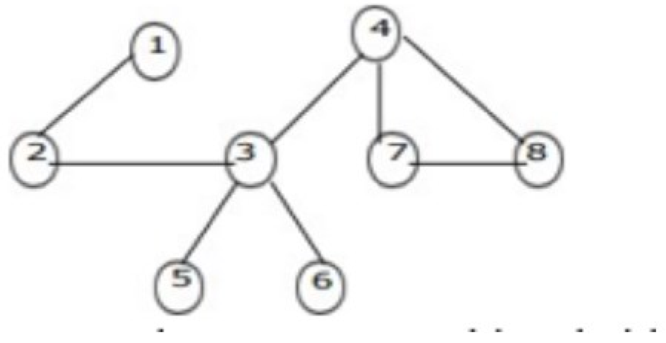
- **Problem state** is each node in the depth first search tree.
- **Solution states** are the problem states ‘S’ for which the path from the root node to ‘S’ defines a tuple in the solution space.
- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E node is a node currently being expanded.

- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Applications of Backtracking:
Backtracking is applied to various problems across different domains. Here are some common applications:
1. N-Queens Problem:
 - Place N queens on an N×N chessboard in such a way that no two queens threaten each other. Backtracking is used to explore different configurations.
2. Subset Sum Problem:
 - Given a set of positive integers and a target sum, find all possible subsets that sum up to the target. Backtracking helps explore different subsets.
3.Graph Coloring:
 - Color the vertices of a graph in such a way that no two adjacent vertices have the same color. Backtracking is used to explore different color assignments.
4.Hamiltonian Cycle:
 - Find a Hamiltonian cycle (a cycle that visits every vertex exactly once) in a given graph. Backtracking explores different paths in the graph.

Backtracking is a versatile approach that can be adapted to solve a wide range of problems where systematic exploration of possible solutions is required. It's particularly powerful in situations where the problem exhibits a recursive structure and involves making decisions at each step.

13 Define Articulation Point or Cut Vertex? Find Articulation points for the following graph?



Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Let us consider the typical case of vertex v,

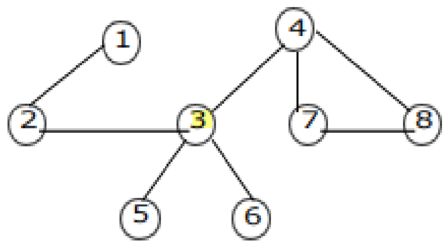
- v is not a leaf
- v is not the root.
- Let w1, w2, ... w_k be the children of v.
- For each child there is a subtree of the DFS tree rooted at this child.
- If for some child, there is no back edge going to a proper ancestor of v.
- if we remove v, this subtree becomes disconnected from the rest of the graph,
- Hence v is an articulation point.

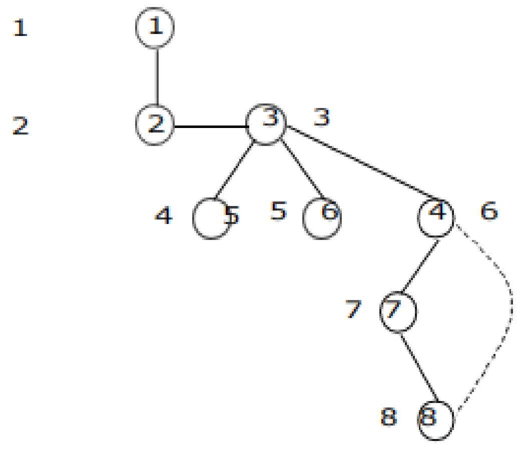
L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN(w) | (u, w) is a back edge}}.

L (u) is the lowest depth first number

It can be reached from 'u' using a path of descendents followed by at most one back edge.

If 'u' is not the root then 'u' is an articulation point if 'u' has a child 'w' such that:


Graph


DFS spanningTree

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}} = min {1, L (2)} = min {1, 2} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min {2, 3} = 2

L (3) = min {DFN (3), min {L (4), L (5), L (6)}} = min {3, min {6, 4, 5}} = 3

L (4) = min {DFN (4), min {L (7)} = min {6, L (7)} = min {6, 6} = 6

L (5) = min {DFN (5)} = 4

L (6) = min {DFN (6)} = 5

L (7) = min {DFN (7), min {L (8)}} = min {7, 6} = 6

L (8) = min {DFN (8), min {DFN (4)}} = min {8, 6} = 6

Therefore, L (1: 8) = {1, 2, 3, 6, 4, 5, 6, 6}

14 Briefly discuss Bi connected components with suitable graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components.

A biconnected component of an undirected graph is a maximal subgraph in which any two vertices are connected by at least two disjoint paths. In other words, removing any single vertex and its incident edges does not disconnect the biconnected component. Biconnected components are particularly useful in analyzing the connectivity and robustness of a graph.

Let's consider an example graph to illustrate biconnected components:

```
 1---2---3
  |  |  |
  4---5---6
```

In this example:

- The biconnected component {1, 2} consists of vertices 1 and 2 along with the edges (1, 2) and (2, 5). Removing any single vertex and its incident edges won't disconnect this subgraph.
- The biconnected component {2, 3} consists of vertices 2 and 3 along with the edges (2, 3) and (2, 5). Removing any single vertex and its incident edges won't disconnect this subgraph.
- The biconnected component {3, 6, 5} consists of vertices 3, 5, and 6 along with the edges (3, 6) and (5, 6). Removing any single vertex and its incident edges won't disconnect this subgraph.
- The biconnected component {5, 4} consists of vertices 4 and 5 along with the edges (4, 5) and (2, 5). Removing any single vertex and its incident edges won't disconnect this subgraph.

Each of these components is biconnected, meaning that there are at least two disjoint paths between any pair of vertices within the component.

In the context of graph algorithms, the identification of biconnected components is often achieved using algorithms such as Tarjan's algorithm or Hopcroft and Tarjan's algorithm. These algorithms efficiently identify the biconnected components of a graph and help analyze the structural properties of the graph in terms of its connectivity and resilience to node or edge failures. Biconnected components are particularly relevant in network design, fault tolerance analysis, and the study of robustness in various applications, including communication networks and transportation systems.

15 State and explain the Sum of Subset problem, Consider the following Sum of Subsets problem instance:n = 6, m = 30 and w [1:6] = {5, 10, 12, 13, 15, 18}. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

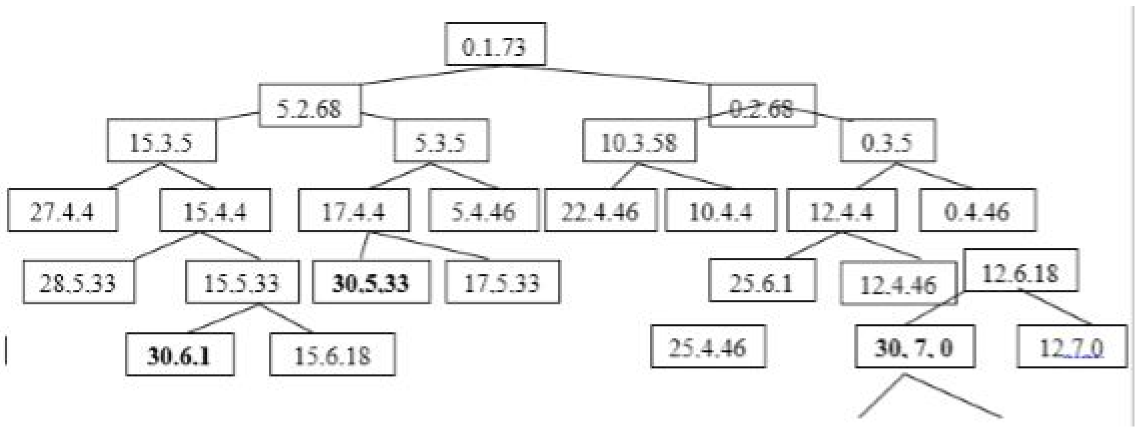
Let, S = {S1 …. Sn} be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.It is always convenient to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤…. ≤ Sn

Algorithm:

Let, S is a set of elements and m is the expected sum of subsets. Then:

1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

n =6, m = 30 and w[6] = {5, 10, 12, 13, 15, 18)



Hence, there are three subset with given sum m=30

A(1011)

B(11001)

C(001001)

```
Algorithm SumOfSub($s, k, r$)
// Find all subsets of $w[1:n]$ that sum to $m$. The values of $x[j]$,
// $1 \le j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] \ge m$.
{
    // Generate left child. Note: $s + w[k] \le m$ since $B_{k-1}$ is true.
    $x[k] := 1$;
    if ($s + w[k] = m$) then write ($x[1:k]$); // Subset found
        // There is no recursive call here as $w[j] > 0$, $1 \le j \le n$.
    else  if ($s + w[k] + w[k+1] \le m$)
            then SumOfSub($s + w[k], k+1, r - w[k]$);
    // Generate right child and evaluate $B_k$.
    if (($s + r - w[k] \ge m$) and ($s + w[k+1] \le m$)) then
    {
        $x[k] := 0$;
        SumOfSub($s, k+1, r - w[k]$);
    }
}
```

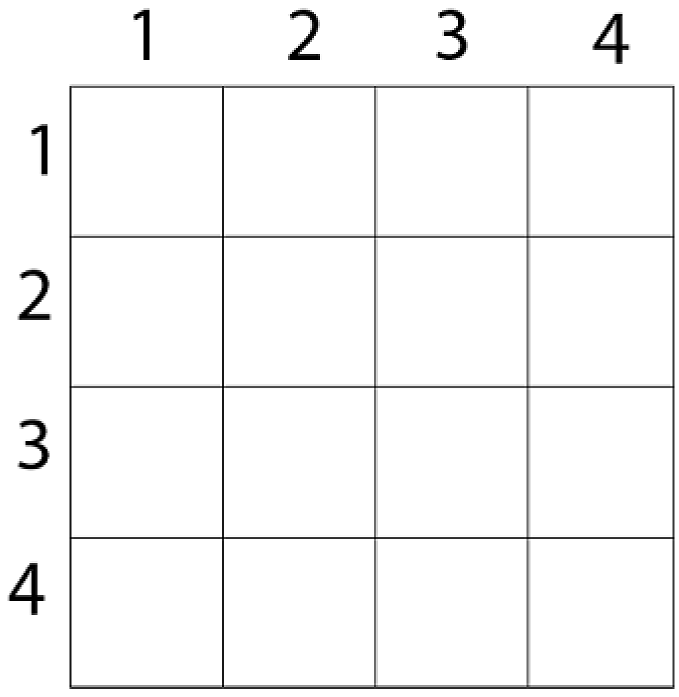
Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

16 Discuss the 4 –Queen’s problem. Draw the portion of the state space tree for n = 4 queens using Backtracking algorithm

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.



4x4 chessboard

- Consider a 4*4 chessboard. Let there are 4 queens.
- The objective is place the 4 queens on 4*4 chessboard in such a way that no two queens should be placed in the same row, same column or diagonal position.
- The explicit constraints are 4 queens are to be placed on 4*4 chessboards in 44 ways.
- The implicit constraints are no two queens are in the same row column or diagonal.
- Let{x1, x2, x3, x4} be the solution vector where x1 column on which the queen i is placed.
- First queen is placed in first row and first column.

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

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

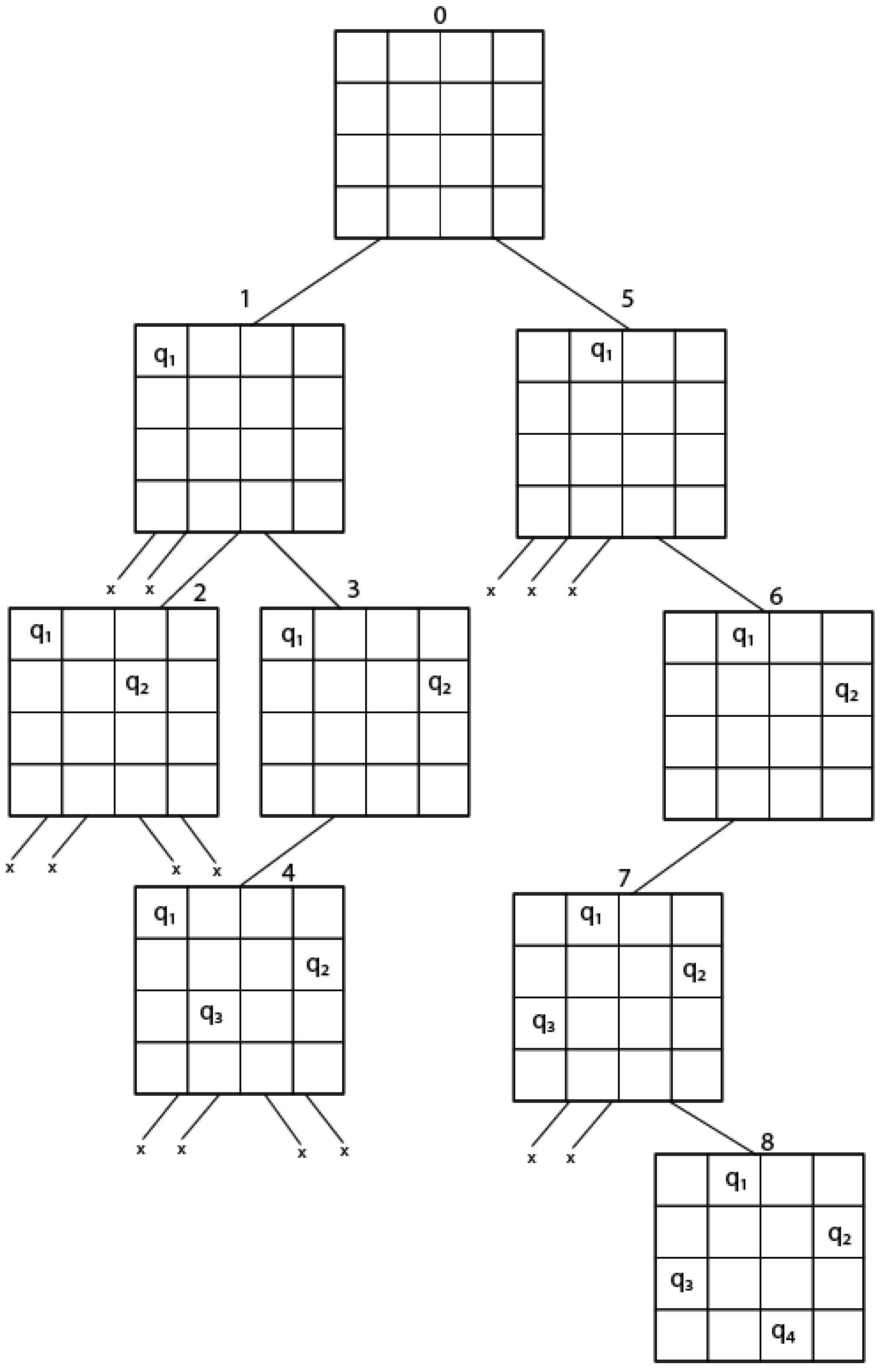
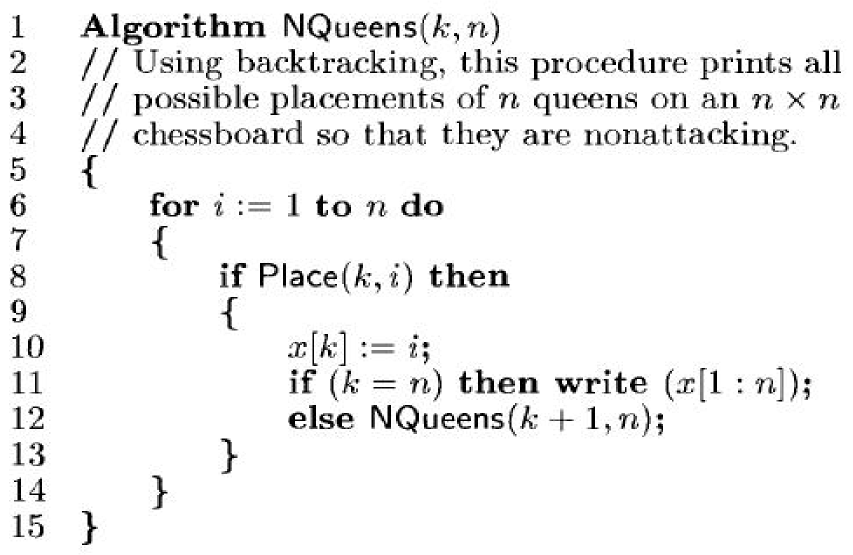
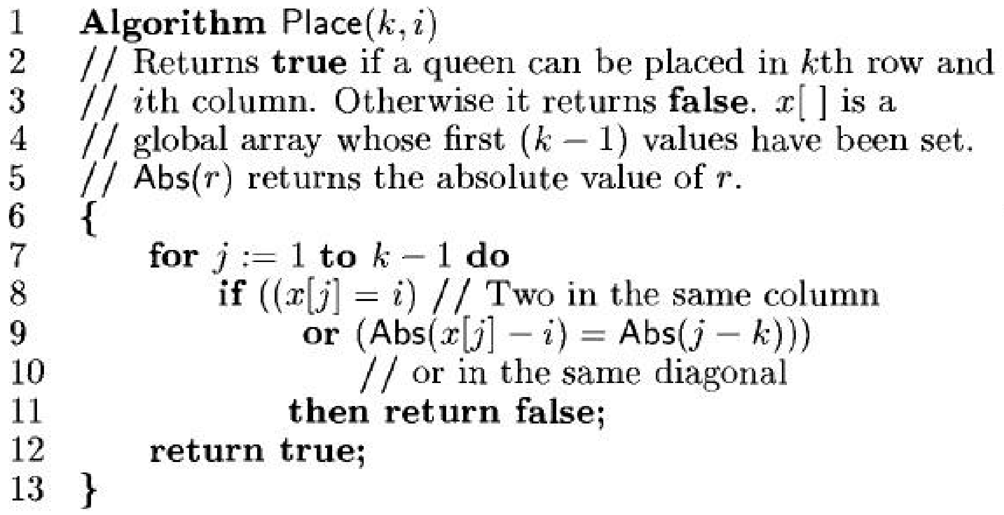


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



```
Algorithm NQueens($k, n$)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place($k, i$) then
        {
            $x[k] := i$;
            if ($k = n$) then write ($x[1 : n]$);
            else NQueens($k + 1, n$);
        }
    }
}
```



```
Algorithm Place($k, i$)
// Returns true if a queen can be placed in $k$th row and
// $i$th column. Otherwise it returns false. $x[\,]$ is a
// global array whose first $(k - 1)$ values have been set.
// Abs($r$) returns the absolute value of $r$.
{
    for $j := 1$ to $k - 1$ do
        if (($x[j] = i$) // Two in the same column
            or (Abs($x[j] - i$) = Abs($j - k$)))
                // or in the same diagonal
            then return false;
    return true;
}
```

17 What is a Hamiltonian Cycle? Explain how to find Hamiltonian path and cycle using backtracking algorithm?

A Hamiltonian Cycle is a cycle that visits every vertex in a graph exactly once, except for the starting and ending vertices which are the same. In simpler terms, it's a closed loop that traverses all the vertices of a graph exactly once.

A Hamiltonian Path is a path in a graph that visits every vertex exactly once, but it may or may not be a cycle.

The problem of finding a Hamiltonian Cycle (or Hamiltonian Path) is a classic NP-complete problem, which means there is no known polynomial-time algorithm to solve it for all cases.

One approach to solving the Hamiltonian Cycle problem is by using the backtracking algorithm. Here's a general outline of how the backtracking algorithm works:

1. Initialization: Start with an empty solution path.
2. Selection: Choose the next vertex to add to the path. It should be adjacent to the last added vertex and not already included in the path.
3. Feasibility:Check if adding the chosen vertex violates any constraints. In the case of a Hamiltonian Cycle, ensure that the vertex is not already in the path and that adding it creates a valid edge.
4. Recursive call: If the chosen vertex is feasible, add it to the path and recursively call the algorithm for the remaining vertices.
5. Backtrack: If the recursive call doesn't lead to a solution, remove the last added vertex from the path and try another vertex.
6. Termination: If all vertices are added to the path and a valid Hamiltonian Cycle is formed, the algorithm terminates. Otherwise, it backtracks until a solution is found or all possibilities are explored.

Here's a pseudocode representation:

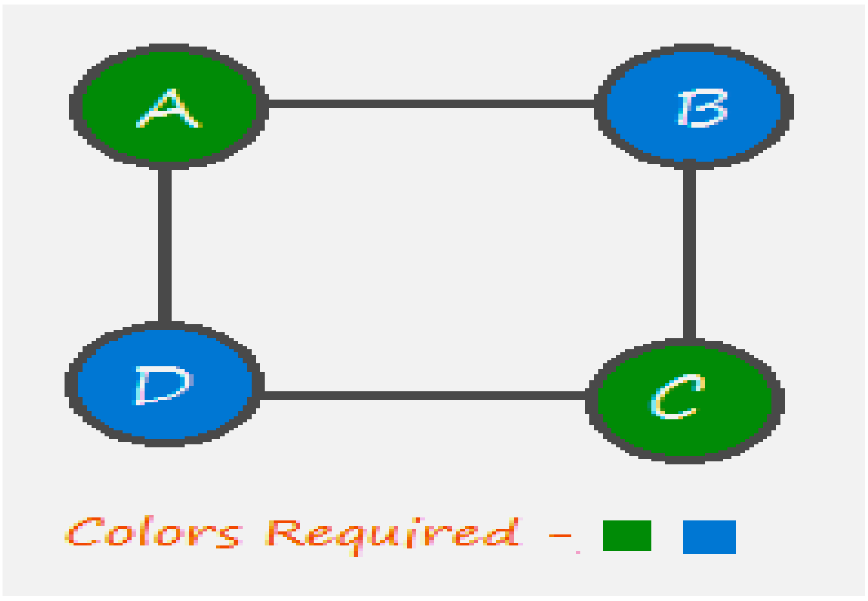
```
function HamiltonianCycle(graph, path, pos):
    if pos == num_vertices:
        if graph[path[pos - 1]][path[0]] == 1:
            return true  // Hamiltonian Cycle found
        else:
            return false  // No Hamiltonian Cycle

    for v in range(num_vertices):
        if isFeasible(v, graph, path, pos):
            path[pos] = v

            if HamiltonianCycle(graph, path, pos + 1):
                return true

            path[pos] = -1  // Backtrack

    return false

function isFeasible(v, graph, path, pos):
    if graph[path[pos - 1]][v] == 0:
        return False

    for i in range(pos):
        if path[i] == v:
            return False

    return True
```

In this pseudocode, `graph` represents the adjacency matrix of the graph, `path` is the current solution path, and `pos` is the position in the path. The algorithm starts with the first vertex and explores all possibilities until a Hamiltonian Cycle is found or all possibilities are exhausted.

It's worth noting that the Hamiltonian Cycle problem is computationally expensive, and the backtracking algorithm explores many possibilities. There are more efficient heuristics and approximation algorithms for practical scenarios, but they don't guarantee an optimal solution in all cases.

18 Explain the Graph–Coloring problem and draw the state space tree for m= 3 colors and n=4 vertices graph. Discuss the time and space complexity

- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used.
- This is termed the m-colorabiltiy decision problem.



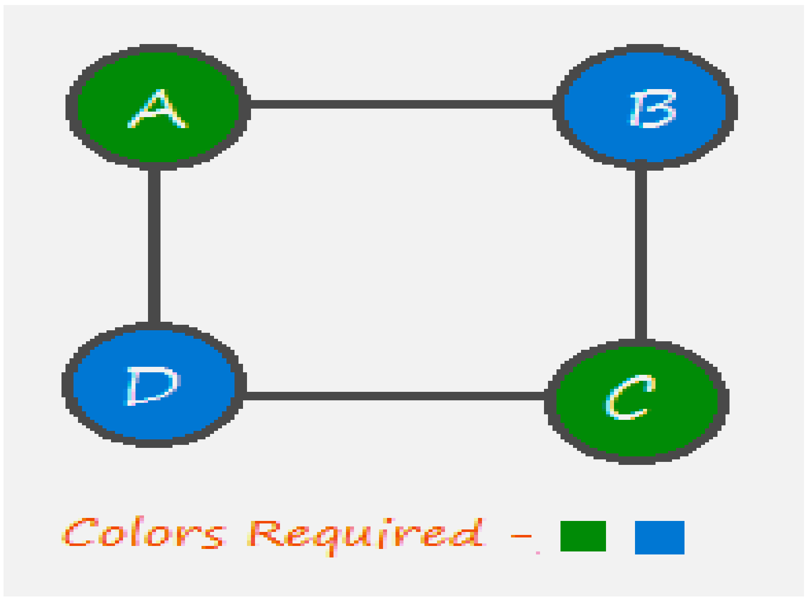
- In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color.
- After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored.
- In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last colored vertices and again proceed further.

Steps To color graph using the Backtracking Algorithm:

- **Different colors**:
- Confirm whether it is valid to color the current vertex with the current color (by checking whether any of its adjacent vertices are colored with the same color).
- If yes then color it and otherwise try a different color.
- Check if all vertices are colored or not.
- If not then move to the next adjacent uncolored vertex.
- If no other color is available then backtrack (i.e. un-color last colored vertex).
- Here backtracking means to stop further recursive calls on adjacent vertices by returning false. In this algorithm Step-1.(**Continue**) and Step-2

(**backtracking**) is causing the program to try different color option.

Continue – try a different color for current vertex.
Backtrack – try a different color for last colored vertex.



21 Explain the control abstraction for Greedy method with suitable example

22 Briefly discuss components of Greedy algorithm. What are the applications of Greedy algorithm?

Components of a Greedy Algorithm:

A Greedy algorithm is characterized by its strategy of making locally optimal choices at each stage with the hope of finding a global optimum. The key components of a Greedy algorithm include:

1. Greedy Choice Property:At each step, make the choice that seems best at the moment without considering the global context. This involves selecting the most promising option based on some criteria.
2. Feasibility:Ensure that the chosen solution is feasible and satisfies the problem constraints.
3. Optimal Substructure: The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
4. Algorithmic Paradigm:Greedy algorithms follow a specific paradigm that involves making a series of locally optimal choices, resulting in a solution that is close to the optimum.

Applications of Greedy Algorithm:

Greedy algorithms are used in various real-world scenarios where making locally optimal choices leads to overall optimal solutions or close-to-optimal solutions. Some common applications include:

- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in finding the shortest path.

25 What is a Minimum Cost Spanning tree list some applications of it?

A Minimum Cost Spanning Tree (MCST) is a tree that spans all the vertices of a connected, undirected graph with the minimum possible total edge weights. In other words, it is a subset of the edges of the graph that forms a tree and connects all the vertices with the minimum total edge weight.

Some well-known algorithms for finding the Minimum Cost Spanning Tree include Prim's algorithm and Kruskal's algorithm.

Applications of Minimum Cost Spanning Tree:

1. Network Design:

- In network design, the Minimum Cost Spanning Tree is used to connect a set of locations with the minimum possible total cost, such as laying cables or pipelines.

2. Circuit Design:

- In electronic circuit design, the Minimum Cost Spanning Tree can be used to connect a set of components with the minimum total wire length.

3. Telecommunication Networks:

   - In the design of telecommunication networks, the MCST can be employed to minimize the cost of laying cables or connecting cell towers.

4. Transportation Networks:

   - In transportation planning, the Minimum Cost Spanning Tree can help in designing cost-effective routes for connecting cities or locations.

5. Water Supply Networks:

   - The MCST can be applied to design the most cost-efficient pipeline network for supplying water to different regions.

6. Power Distribution Networks:

   - In power distribution systems, the MCST can be used to minimize the cost of connecting power stations to various cities or industrial areas.

7. Computer Network Design:

   - In computer networks, the MCST is useful for designing efficient communication networks with minimum data transmission costs.

8. Robotics:

   - In robotics, when robots need to be connected in a network for communication or coordination, the Minimum Cost Spanning Tree can be applied to minimize communication costs.

9. Resource Management:

   - In resource management scenarios, where resources need to be connected or distributed efficiently, the MCST can help optimize resource utilization.

10. Traffic Engineering:

- In traffic engineering, the MCST can be applied to design optimal road networks that minimize the overall travel distance or time.

11. Railway Planning:

- In railway systems, the MCST can assist in planning optimal routes for connecting different railway stations with minimum track length.

These applications highlight the versatility of the Minimum Cost Spanning Tree concept in solving real-world problems related to network connectivity and resource optimization. The choice of the appropriate algorithm depends on factors such as the characteristics of the graph and the specific requirements of the application.

27 Distinguish between the Dynamic Programming and Greedy Method

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

solution using Principle of Optimality.	Optimal Solution.

28 Contrast between the Prim’s and Kruskal’s algorithm.

Prim's algorithm and Kruskal's algorithm are both algorithms for finding the Minimum Spanning Tree (MST) in a connected, undirected graph. However, they differ in terms of their approaches, implementations, and the way they select edges to build the spanning tree. Here's a contrast between Prim's and Kruskal's algorithms:

Prim's Algorithm:

1. Nature:

- Prim's algorithm is a greedy algorithm that grows the Minimum Spanning Tree from an arbitrary starting vertex.

2. Edge Selection:

- It selects the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST at each step.

3. Implementation:

- Prim's algorithm is often implemented using a priority queue or a min-heap to efficiently find the minimum-weight edge.

4. Data Structures:

- Typically uses an array or heap-based data structure to maintain the set of vertices not yet included in the MST.5. Time Complexity:

- The time complexity of Prim's algorithm is O(V^2) with an adjacency matrix and O(E + V log V) with an adjacency list, where V is the number of vertices and E is the number of edges.

6. Parallel Implementation:

- It's more challenging to parallelize Prim's algorithm efficiently compared to Kruskal's algorithm.

Kruskal's Algorithm:

1. Nature:

- Kruskal's algorithm is a greedy algorithm that selects edges based on their weights, without regard to the vertices they connect.

2. Edge Selection:

- It sorts all the edges in non-decreasing order of weight and adds the smallest edge that does not form a cycle with the edges already included.

3. Implementation:

- Kruskal's algorithm is often implemented using a disjoint-set data structure (union-find) to efficiently check for cycles and maintain connected components.

4. Data Structures:

- Uses disjoint-set data structure to keep track of connected components.

5. Time Complexity:

- The time complexity of Kruskal's algorithm is O(E log E) with an efficient sorting algorithm, where E is the number of edges.

6. Parallel Implementation:

- Kruskal's algorithm can be parallelized more easily than Prim's algorithm, making it more suitable for parallel computing environments.

Choice of Algorithm:

1. Sparse vs. Dense Graphs:

- Prim's algorithm can be more efficient on dense graphs with a large number of edges, while Kruskal's algorithm is often preferred for sparse graphs.

2. Data Structure Consideration:

- The choice of data structures (priority queue or disjoint-set) can influence the efficiency of both algorithms.

3. Parallelization:

- Kruskal's algorithm is generally more amenable to parallelization, which can be advantageous in certain computing environments.

4. Memory Requirements:

- Kruskal's algorithm may have lower memory requirements in certain scenarios due to the use of a disjoint-set data structure.

In summary, the choice between Prim's and Kruskal's algorithms depends on factors such as the characteristics of the graph, available data structures, and parallelization considerations. Both algorithms guarantee the construction of a Minimum Spanning Tree with optimal total edge weights.

31 State the principle of optimality. List the characteristics of dynamic programming.

- A technique that breaks the problems into sub-problems, and saves the result for future purposes to avoid repetitive computations.
- dynamic programming is to solve optimization problems.
- optimization problems means that to find out the minimum or the maximum solution of a problem.

Example: Fibonacci series,

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,...

With the base values F(0) = 0, and F(1) = 1,

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

Characteristics commonly associated with dynamic programming:

1. Optimal Substructure:
   - The optimal solution to a problem can be constructed from optimal solutions to its subproblems. This property allows breaking down a complex problem into simpler, more manageable subproblems.
2. Overlapping Subproblems:
   - Dynamic programming problems exhibit overlapping subproblems, where the same subproblems are solved multiple times. To avoid redundant calculations, solutions to subproblems are stored and reused.
3. Memoization:
   - Memoization is a technique used in dynamic programming to store the results of expensive function calls and return the cached result when the same inputs occur again. This helps in avoiding redundant computations.
4. Recursive Relation (Recurrence Relation):
   - Dynamic programming problems often involve defining recursive relations that express the optimal solution to a problem in terms of solutions to its smaller subproblems. These relations guide the construction of the dynamic programming algorithm.
5. Bottom-Up and Top-Down Approaches:
   - Dynamic programming solutions can be constructed using a bottom-up approach, where solutions to smaller subproblems are computed first and combined to solve larger subproblems. Alternatively, a top-down approach involves recursively breaking down the problem until base cases are reached.
6. State Transition:

- Dynamic programming algorithms involve defining states for a problem and transitioning between states to build the optimal solution. The state transition is often governed by a recurrence relation.

7. Tabulation:

   - In a bottom-up dynamic programming approach, tabulation is used to build a table that stores solutions to subproblems in a systematic manner. The table is filled iteratively, starting from smaller subproblems and progressing to larger ones.

8. Polynomial Time Complexity:

   - Dynamic programming algorithms are designed to have polynomial time complexity, making them efficient for solving optimization problems that exhibit the optimal substructure and overlapping subproblems.

9. Optimization Problems:

   - Dynamic programming is often applied to solve optimization problems, where the goal is to find the best solution among a set of feasible solutions.

10. Deterministic Solutions:

    - Dynamic programming algorithms provide deterministic solutions, meaning that the same input will always produce the same output. This predictability is crucial for their widespread application.

11. Applications in Various Domains:

    - Dynamic programming finds applications in diverse domains, including computer science, operations research, economics, artificial intelligence, bioinformatics, and more.

38 Compare the 0/1 knapsack problem and fractional knapsack problem. List out the applications of Dynamic programming.

1. Nature of Items:

-0/1 Knapsack Problem: In the 0/1 Knapsack Problem, items cannot be divided. You either select an entire item or do not include it in the knapsack at all.

-Fractional Knapsack Problem: In the Fractional Knapsack Problem, items can be divided, and you can take fractions of items.

2. Decision Variables:

- 0/1 Knapsack Problem: The decision variable is binary, representing whether an item is selected or not (0 or 1).

- Fractional Knapsack Problem: The decision variable is continuous, representing the fraction of an item included in the knapsack.

3. Greedy Approach:

- 0/1 Knapsack Problem: Typically solved using dynamic programming or backtracking.

- Fractional Knapsack Problem: Solved using a greedy approach, where items are selected based on their value-to-weight ratio.

4. Optimality:

- 0/1 Knapsack Problem:Finding an optimal solution requires considering all possible combinations of items, making it more computationally expensive.

- Fractional Knapsack Problem:A greedy approach, such as selecting items with the highest value-to-weight ratio first, often leads to an optimal solution.

5. Complexity:

- 0/1 Knapsack Problem:Solving the 0/1 Knapsack Problem may involve exponential time complexity due to the need to explore all possible combinations.
- Fractional Knapsack Problem: Solving the Fractional Knapsack Problem is generally more efficient, often achievable in linear time.

Applications of Dynamic Programming:

Dynamic programming is a versatile technique with applications in various domains. Some common applications include:

1. Optimization Problems:

- Dynamic programming is widely used to solve optimization problems, such as the knapsack problem, where the goal is to find the best solution among a set of feasible solutions.

2. Resource Allocation:

- It is applied in resource allocation problems, such as project scheduling and task assignment, to optimize the use of limited resources.

3. Shortest Path Problems:

- Dynamic programming algorithms, like Dijkstra's algorithm and Floyd-Warshall algorithm, are used for finding shortest paths in graphs.

4. Sequence Alignment (Bioinformatics):
   - Dynamic programming is used in bioinformatics for sequence alignment, where it helps find the optimal alignment between biological sequences.

5. Text Justification:
   - In text justification problems, dynamic programming can be employed to determine the optimal way to format text within a specified width.

42 Define: i) State-Space tree ii) E – Node iii) Dead Node. iv) LC – Search v) Branch and Bound.

1. State-Space Tree:
   - A state-space tree is a graphical representation of all possible states that a system can be in, along with the transitions between those states. It is commonly used in computer science and artificial intelligence to model problems involving decision-making or searching for a solution. Each node in the tree represents a specific state, and edges between nodes represent possible transitions or actions.

2. E-Node:
   - An E-node, or expansion node, is a node in a state-space tree where the expansion or exploration of possible actions or transitions takes place. It is a point in the tree where the algorithm generates child nodes by applying actions to the

current state. The term "E-node" is often used in the context of search algorithms like depth-first search or breadth-first search.

3. Dead Node:

- A dead node, also known as a terminal node or a goal node, is a node in a state-space tree that represents a final or goal state. In the context of search algorithms, reaching a dead node indicates that a solution or goal has been found. Dead nodes are typically the leaves of the tree where the desired outcome has been achieved.

4. LC-Search (Least-Cost Search):

- LC-Search is a search algorithm used in artificial intelligence for finding the path from a start state to a goal state with the minimum cost. It considers the cost associated with each state and aims to find the path with the least overall cost. Dijkstra's algorithm and A* search algorithm are examples of LC-Search algorithms.

5. Branch and Bound:

- Branch and Bound is an algorithmic paradigm used for solving optimization problems. It systematically searches through the solution space of a problem, dividing it into subproblems and using bounds to eliminate subproblems that cannot provide better solutions than the current best-known solution. It is often used for combinatorial optimization problems and is applicable to problems like the Traveling Salesman Problem.

These definitions provide an overview of the terms related to state-space trees, search algorithms, and optimization techniques. Each term is commonly used in the context of problem-solving, especially in areas like artificial intelligence and algorithms.

43 Discuss about general method of branch and bound technique

Branch and Bound is a general algorithmic technique used for solving optimization problems, particularly combinatorial optimization problems. The goal of Branch and Bound is to systematically search through the solution space of a problem, dividing it into subproblems and using bounds to eliminate subproblems that cannot provide better solutions than the current best-known solution. The technique is widely used in areas such as operations research, scheduling, and network design. Here is a general method for the Branch and Bound technique:

General Steps for Branch and Bound:

1. Initialization:

- Initialize the algorithm with an initial solution, often an empty or trivial one.
- Set an initial upper bound to represent the best-known solution.
- Set up a priority queue or a data structure to manage the subproblems.

2. Bounding:

- For each subproblem, compute an upper bound on the optimal solution and a lower bound on the subproblem's solution.
- The bounds are used to determine whether a subproblem should be further subdivided or can be pruned (eliminated) without affecting the optimality of the solution.

3. Branching:
   - Divide the current subproblem into smaller subproblems by generating new partial solutions.
   - The branching process creates a tree structure, where each node represents a subproblem and each branch represents a decision or choice.
4. Exploration:
   - Use a systematic strategy to explore the tree of subproblems. Common strategies include depth-first search, breadth-first search, or a combination of both (iterative deepening).
   - Prioritize the exploration based on the bounds to focus on the most promising subproblems first.
5. Pruning:
   - Prune (discard) subproblems that can be proven to be suboptimal or cannot lead to a better solution than the current best-known solution.
   - Pruning helps reduce the search space, improving the efficiency of the algorithm.

6. Updating Best Solution:
   - Whenever a new feasible solution is found, update the best-known solution if the new solution is better than the current best-known one.
7. Termination:

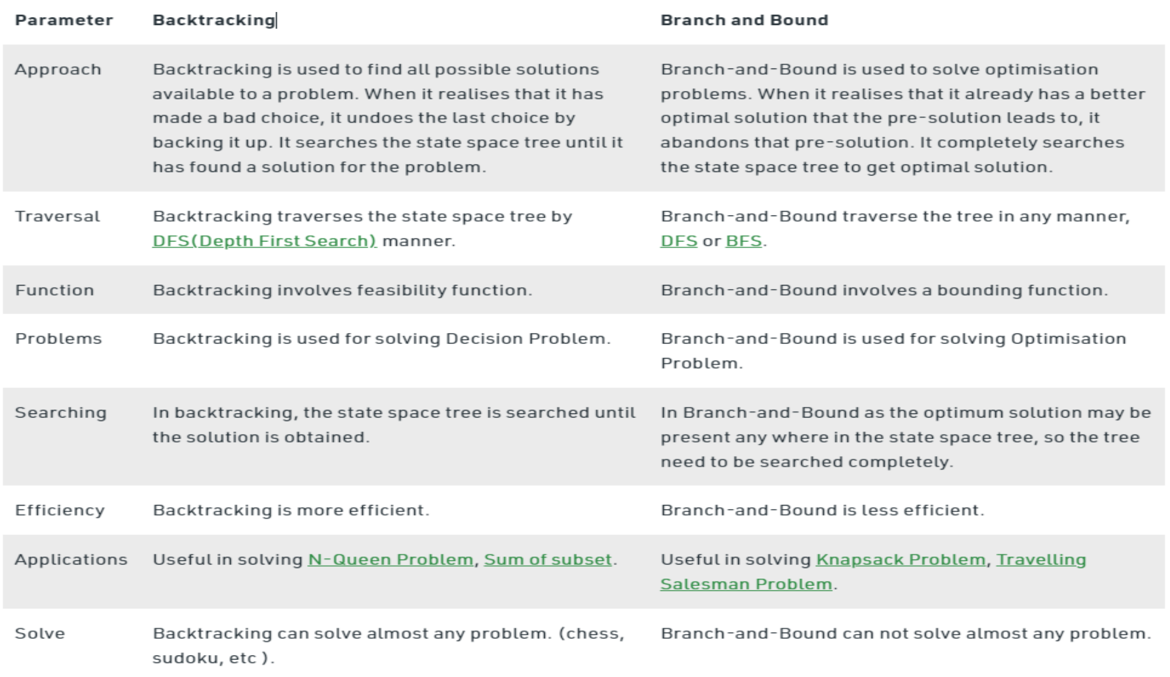
- The algorithm terminates when all subproblems have been explored, and no further improvement can be made to the best-known solution.

Example: Traveling Salesman Problem (TSP)

In the context of the Traveling Salesman Problem, the Branch and Bound technique would involve systematically exploring different tour possibilities while pruning branches that are guaranteed to be suboptimal. The bounding step may involve computing lower bounds on subproblems based on partial tours, and the branching step may involve extending the partial tours to consider additional cities.

The efficiency of the Branch and Bound technique depends on the problem structure, the quality of bounding functions, and the choice of exploration and pruning strategies. It is a powerful approach for solving NP-hard optimization problems where an exhaustive search is impractical.

46 Distinguish branch and bound method and backtracking.



Parameter	Backtracking	Branch and Bound
Approach	Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution that the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.
Traversal	Backtracking traverses the state space tree by DFS(Depth First Search) manner.	Branch-and-Bound traverse the tree in any manner, DFS or BFS.
Function	Backtracking involves feasibility function.	Branch-and-Bound involves a bounding function.
Problems	Backtracking is used for solving Decision Problem.	Branch-and-Bound is used for solving Optimisation Problem.
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Branch-and-Bound as the optimum solution may be present any where in the state space tree, so the tree need to be searched completely.
Efficiency	Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications	Useful in solving N-Queen Problem, Sum of subset.	Useful in solving Knapsack Problem, Travelling Salesman Problem.
Solve	Backtracking can solve almost any problem. (chess, sudoku, etc).	Branch-and-Bound can not solve almost any problem.

Backtracking	Branch and bound
Backtracking is a problem-solving technique so it solves the decision problem.	Branch n bound is a problem-solving technique so it solves the optimization problem.
When we find the solution using backtracking then some bad choices can be made.	When we find the solution using Branch n bound then it provides a better solution so there are no chances of making a bad choice.
Backtracking uses a Depth first search.	It is not necessary that branch n bound uses Depth first search. It can even use a Breadth-first search and best-first search.
The state space tree is searched until the	The state space tree needs to be searched

solution of the problem is obtained.	completely as the optimum solution can be present anywhere in the state space tree.
In backtracking, all the possible solutions are tried. If the solution does not satisfy the constraint, then we backtrack and look for another solution.	In branch and bound, based on search; bounding values are calculated. According to the bounding values, we either stop there or extend.
Applications of backtracking are n-Queens problem, Sum of subset.	Applications of branch and bound are knapsack problem, travelling salesman problem, etc.
Backtracking is more efficient than the Branch and bound.	Branch n bound is less efficient.
It contains the feasibility function.	It contains the bounding function.
Backtracking solves the given problem by first finding the solution of the subproblem and then recursively solves the other problems based on the solution of the first subproblem.	Branch and bound solves the given problem by dividing the problem into two atleast subproblems.

47 What is state space tree? What are the different ways of searching an answer node in an state space tree explain with example

A state space tree is a graphical representation of the possible states that a system can be in, along with the transitions between those states. It is commonly used in computer science and artificial intelligence to model problems involving decision-making or searching for a solution. Each node in the tree represents a specific state, and edges between nodes represent possible transitions or actions.

Components of a State Space Tree:

1. Nodes:Represent individual states.
2. Edges:Represent transitions or actions that lead from one state to another.
3. Root Node:Represents the initial state.
4. Leaf Nodes: Represent goal states or terminal states.

Example: N-Queens Problem

Consider the N-Queens problem, where you need to place N queens on an N×N chessboard such that no two queens threaten each other

Searching an Answer Node in a State Space Tree:

1. Breadth-First Search (BFS):
   - Explore all nodes at the current depth before moving on to the next depth.
   - Example: In the N-Queens problem, BFS would consider all possible board configurations with one queen placed before considering configurations with two queens placed.
2. Depth-First Search (DFS):
   - Explore as far as possible along one branch before backtracking.

- Example: DFS might explore one branch of the state space tree, placing queens in a sequence, before exploring alternative sequences.

3. Iterative Deepening Depth-First Search (IDDFS):

- A combination of BFS and DFS where depth-first search is applied with increasing depth limits until the goal is found.

- Example: IDDFS might start with a depth limit of 1, then 2, and so on until a solution is found.

4. Search Algorithm:

- Uses a heuristic function to estimate the cost of reaching the goal from a given state.

- Example: In the N-Queens problem, A* might prioritize paths that place queens in positions that reduce the number of threatening pairs.

These are just a few examples of search algorithms applied to state space trees. The choice of algorithm depends on the specific problem and its characteristics, such as the size of the state space and the complexity of the transitions between states.

48 Explain in detail about Deterministic and non-deterministic algorithms

Deterministic and non-deterministic algorithms are two fundamental concepts in the field of computer science and algorithms. These terms describe the nature of the computation or decision-making process within an algorithm.

Deterministic Algorithm:

1. Definition:

   - A deterministic algorithm is one that, given the same input and initial state, will always produce the same output and final state, and will follow the same sequence of steps every time it is executed.

   - Deterministic algorithms are entirely predictable and can be precisely described.

2. Characteristics:

   - The behavior of a deterministic algorithm is entirely determined by its input and its logic.

   - Examples include sorting algorithms like Quicksort and searching algorithms like Binary Search.

   - These algorithms are easy to analyze, and their correctness is straightforward to prove.

3. Pros and Cons:

   - Pros: Predictable, easy to understand, and deterministic behavior simplifies analysis.

   - Cons:Some problems might require non-deterministic approaches, and determinism can make certain algorithms less efficient.

4. Example:

   - Consider a simple deterministic algorithm that checks whether a given number is prime or not. The steps of the algorithm, given the same input, will always yield the same result.

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Non-deterministic Algorithm:

1. Definition:
   - A non-deterministic algorithm is one in which the sequence of steps or the outcome is not uniquely determined by the input and the initial state.
   - Non-deterministic algorithms may introduce randomness or make non-unique choices during their execution.
2. Characteristics:
   - Non-deterministic algorithms explore multiple possibilities simultaneously or make choices without committing to a single path until forced to do so.
   - Examples include certain algorithms in the realm of artificial intelligence, cryptography, and some optimization problems.
3. Pros and Cons:

- Pros: Can be more efficient for certain problems, may provide better solutions in some cases.
- Cons: Harder to analyze and prove correctness, may require more resources.

4. Example:

- The Traveling Salesman Problem (TSP) is an example where a non-deterministic algorithm (like simulated annealing) might be used to explore various routes before settling on an optimal or near-optimal solution.# Simulated annealing approach for the Traveling Salesman Problem

```
# (Note: This is a simplified example and not a complete implementation)
def simulated_annealing_tsp(graph):
    # Algorithm explores multiple possibilities, introducing randomness
    # and making non-unique choices during the optimization process
    # ...
```

In summary, the key difference lies in the predictability of the algorithm's behavior. Deterministic algorithms are entirely predictable and reproduce the same output given the same input and state, while non-deterministic algorithms may introduce randomness or make non-unique choices, leading to different outcomes on different runs. Each type has its advantages and is suitable for different types of problems.

Deterministic Algorithm	Non-deterministic Algorithm
A deterministic algorithm is one whose behavior is completely determined by its inputs and the sequence of its instructions.	A non-deterministic algorithm is one in which the outcome cannot be predicted with certainty, even if the inputs are known.
For a particular input, the computer will give always the same output.	For a particular input the computer will give different outputs on different execution.
Can solve the problem in polynomial time.	Can’t solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution due to more than one path the algorithm can take.
Operation are uniquely defined.	Operation are not uniquely defined.
Like linear search and binary search	like 0/1 knapsack problem.
Deterministic algorithms usually have a well-defined worst-case time complexity.	Time complexity of non-deterministic algorithms is often described in terms of expected running time.
Deterministic algorithms are entirely predictable and always produce the same output for the same input.	Non-deterministic algorithms may produce different outputs for the same input due to random events or other factors.
Deterministic algorithms usually provide precise solutions to problems.	non-deterministic algorithms often provide approximate solutions to the problems.
Deterministic algorithms are commonly used in applications where precision is critical, such as in cryptography, numerical analysis, and computer graphics.	Non-deterministic algorithms are often used in applications where finding an exact solution is difficult or impractical, such as in artificial intelligence, machine learning, and optimization problems.

Deterministic Algorithm	Non-deterministic Algorithm
Examples of deterministic algorithms include sorting algorithms like bubble sort, insertion sort, and selection sort, as well as many numerical algorithms.	Examples of non-deterministic algorithms include probabilistic algorithms like Monte Carlo methods, genetic algorithms, and simulated annealing.

49 What are differences between NP-Hard and NP-Complete classes? Explain with examples.

Satisfiability:

The satisfiability is a boolean formula that can be constructed using the following literals and operations.

- A literal is either a variable or its negation of the variable.
- The literals are connected with operators **˅, ˄, ⇒ , ⇔**
- Parenthesis

- It is to determine whether a Boolean formula is true for some assignment of truth values to the variables.

Reducability:

- A problem Q1 can be reduced to Q2 if any instance of Q1 can be easily rephrased as an instance of Q2.
- If the solution to the problem Q2 provides a solution to the problem Q1, then these are said to be reducable problems.

NP-Hard and NP-Complete Problem:

- The nondeterministic polynomial time problems can be classified into two classes. They are

   1. NP Hard and
   2. NP Complete

NP-Hard:

A problem L is NP-Hard iff satisfiability reduces to L i.e., any nondeterministic polynomial time problem is satisfiable and reducable then the problem is said to be NP-Hard.

Example: Halting Problem, Flow shop scheduling problem

NP-Complete:

A problem L is NP-Complete iff L is NP-Hard and L belongs to NP (nondeterministic polynomial).

If an NP-hard problem can be solved in polynomial time, then all NP- complete problems can be solved in polynomial time.

All NP-Complete problems are NP-hard, but some NP- hard problems are not known to be NP- Complete.

Example:

Knapsack decision problem can be reduced to knapsack optimization problem

50 How are P and NP problems related? Give the relation between NP-hard and NP problems with a neat diagram.

P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

Features:

- The solution to **P problem**s is easy to find.
- **P** is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.
- **Examples:**
- Searching – Linear search & Binary search
- Sorting – Insertion sort, Merge sort, Quick sort
- Matrix chain multiplication
- Element uniqueness
- Graph connectivity

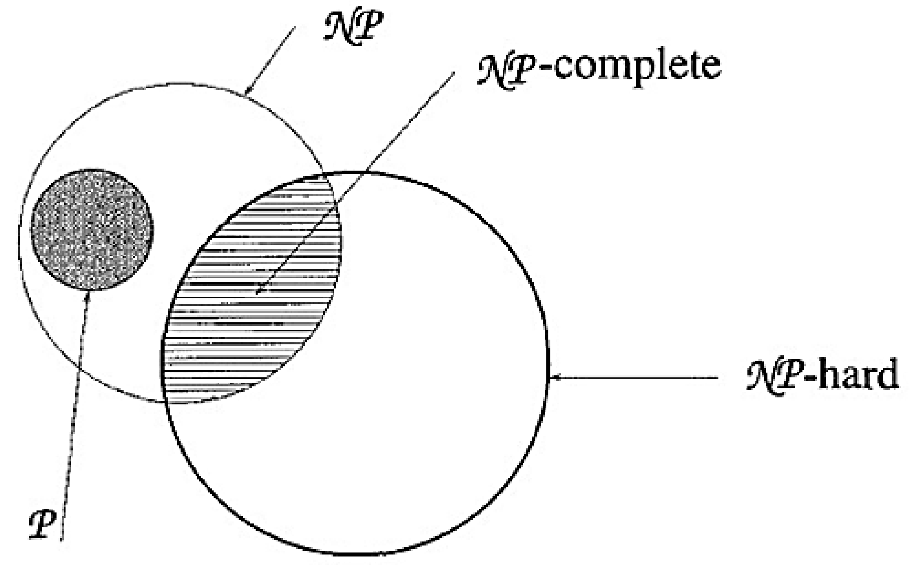
NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.
- **Examples:**
- Travelling Salesperson
- 0/1 Knapsack
- Sum of subsets
- Hamiltonian cycle
- Graph coloring

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-complete respectively.



Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems