## CD402 Analysis &Design of Algorithms

**Unit I :** Definitions of algorithms and complexity, Time and Space Complexity; Time space tradeoff, various bounds on complexity, Asymptotic notation, Recurrences and Recurrences solving techniques, Introduction to divide and conquer technique, example: binary search, merge sort, quick sort, heap sort, strassen's matrix multiplication etc, Code tuning techniques: Loop Optimization, Data Transfer Optimization, Logic Optimization, etc.

**Unit II :** Study of Greedy strategy, examples of greedy method like optimal merge patterns, Huffman coding, minimum spanning trees, knapsack problem, job sequencing with deadlines, single source shortest path algorithm etc. Correctness proof of Greedy algorithms.

**Unit III :** Concept of dynamic programming, problems based on this approach such as 0/1 knapsack, multistage graph, reliability design, Floyd-Warshall algorithm etc.

**Unit IV :** Backtracking concept and its examples like 8 queen's problem, Hamiltonian cycle, Graph colouring problem etc. Introduction to branch & bound method, examples of branch and bound method like travelling salesman problem etc. Meaning of lower bound theory and its use in solving algebraic problem, introduction to parallel algorithms.

**Unit V :** Advanced tree and graph algorithms, NP-hard and NP-complete problems, Approximations Algorithms, Data Stream Algorithms, Introduction to design and complexity of Parallel Algorithms

**References:**

1. Coremen Thomas, Leiserson CE, Rivest RL, Introduction to Algorithms, Third edition, PHI.
2. Horowitz &Sahani, Analysis & Design of Algorithm, Fourth Edition Computer Science Press.
3. Dasgupta, algorithms, Fifth Edition, TMH
4. Ullmann; Analysis & Design of Algorithm, Addison-wesley publishing company,
5. Michael T Goodrich, RobartoTamassia, Algorithm Design, Wiely India
6. Rajesh K Shukla: Analysis and Design of Algorithms: A Beginner's Approach; Wiley

**List of Experiments:**
1. Write a program for Iterative and Recursive Binary Search.
2. Write a program for Merge Sort.
3. Write a program for Quick Sort.
4. Write a program for Strassen's Matrix Multiplication.
5. Write a program for optimal merge patterns.
6. Write a program for Huffman coding.
7. Write a program for minimum spanning trees using Kruskal's algorithm.
8. Write a program for minimum spanning trees using Prim's algorithm.
9. Write a program for single sources shortest path algorithm.
10. Write a program for Floye-Warshal algorithm.
11. Write a program for traveling salesman problem.
12. Write a program for Hamiltonian cycle problem.

**RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL**

---

# Unit-1

### ALGORITHM:

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition, every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied; Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

### DESIGNING ALGORITHMS:

In Computer Science, developing an algorithm is an art or a skill. Before actual implementation of the program, designing an algorithm is very important step.

**Steps are:**
1. Understand the problem
2. Decision making on
   a. Capabilities of computational devices
   b. Select exact or approximate methods
   c. Data Structures
   d. Algorithmic strategies
3. Specification of algorithms
4. Algorithmic verification
5. Analysis of algorithm
6. Implementation or coding of algorithm

### ANALYZING ALGORITHMS:

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

**Performance of a program:**

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

**Time Complexity:**

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

**Space Complexity:**

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

**Data space:** Data space is the space needed to store all constant and variable values. Data space has two components:

Space needed by constants and simple variables in program.

Space needed by dynamically allocated objects such as arrays and class instances.

**Environment stack space:** The environment stack is used to save information needed to resume execution of partially completed functions.

**Instruction Space:** The amount of instructions space that is needed depends on factors such as:

The compiler used to complete the program into machine code.

The compiler options in effect at the time of compilation

The target computer.

**Algorithm Design Goals**

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Likewise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

**Basic techniques of designing efficient algorithm**

1. Divide-and-Conquer
2. Greedy method
3. Dynamic Programming
4. Backtracking
5. Branch-and-Bound

In this section we will briefly describe these techniques with appropriate examples.

1. **Divide & conquer technique** is a top-down approach to solve a problem.

    The algorithm which follows divide and conquer technique involves 3 steps:

    - Divide the original problem into a set of sub problems.
    - Conquer (or Solve) every sub-problem individually, recursive.
    - Combine the solutions of these sub problems to get the solution of original problem.

2. **Greedy technique** is used to solve an optimization problem. An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or

minimized (known as objective function) w. r. t. some constraints or conditions. Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function. That is, it makes a locally optimal choice in the hope that this choice will lead to an overall globally optimal solution. The greedy algorithm does not always guarantee the optimal solution but it generally produces solutions that are very close in value to the optimal.

3. **Dynamic programming** technique is similar to divide and conquer approach. Both solve a problem by breaking it down into a several sub problems that can be solved recursively. The difference between the two is that in dynamic programming approach, the results obtained from solving smaller sub problems are *reused* (by maintaining a table of results) in the calculation of larger sub problems. Thus, dynamic programming is a *Bottom-up* approach that begins by solving the smaller sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained. *Reusing* the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the re-computations (computing results twice or more) of the same problem.

   Thus, Dynamic programming approach takes much less time than naïve or straightforward methods, such as divide-and-conquer approach which solves problems in *top-down* method and having lots of re-computations. The dynamic programming approach always gives a guarantee to get an optimal solution.

4. The term **"backtrack"** was coined by American mathematician D.H. Lehmer in the 1950s. Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and relatively quick test of whether it can possibly be completed to a valid solution. Backtrack algorithms try each possibility until they find the right one. It is a depth-first-search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. If there are no more choice points, the search fails.

5. **Branch-and-Bound** (B&B) is a rather general optimization technique that applies where the greedy method and dynamic programming fail.

   B&B design strategy is very similar to backtracking in that a state-space- tree is used to solve a problem. Branch and bound is a systematic method for solving optimization problems. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node  to expand and when, and another criterion tells the algorithm when an optimal solution has been found. Branch and Bound (B&B) is the most widely used tool for solving large scale NP-hard combinatorial optimization problems.

The following table-1.1 summarizes these techniques with some common problems that follow these techniques with their running time.  Each technique has different running time (...time complexity).

| Design strategy | Problems that follows |
|---|---|
| Divide & Conquer | <ul><li>Binary search</li><li>Multiplication of two n-bits numbers</li><li>Quick Sort</li><li>Heap Sort</li><li>Merge Sort</li></ul> |
| Greedy Method | <ul><li>Knapsack (fractional) Problem</li><li>Minimum cost Spanning tree<ul><li>Kruskal"s algorithm</li><li>Prim"s algorithm</li></ul></li><li>Single source shortest path problem<ul><li>Dijkstra"s algorithm</li></ul></li></ul> |
| Dynamic Programming | <ul><li>All pair shortest path-Floyed algorithm</li><li>Chain matrix multiplication</li><li>Longest common subsequence (LCS)</li><li>0/1 Knapsack Problem</li><li>Traveling salesmen problem (TSP)</li></ul> |
| Backtracking | <ul><li>N-queen"s problem</li><li>Sum-of subset</li></ul> |
| Branch & Bound | <ul><li>Assignment problem</li><li>Traveling salesmen problem (TSP)</li></ul> |

**Table 1.1: Various Design Strategies**

## Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

**Log n** When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n2.

**n** When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

**nlog n** This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

**$n^2$** When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four-fold.

**n3** Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight-fold.

**$2^n$** Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

**Complexity of Algorithms**

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

**ASYMPTOTIC NOTATIONS (RATE OF GROWTH):**

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O)
2. Big–OMEGA (Ω)
3. Big–THETA (Θ)

**1. Big–OH O (Upper Bound)**

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (<) that of g(n) figure 1.1.
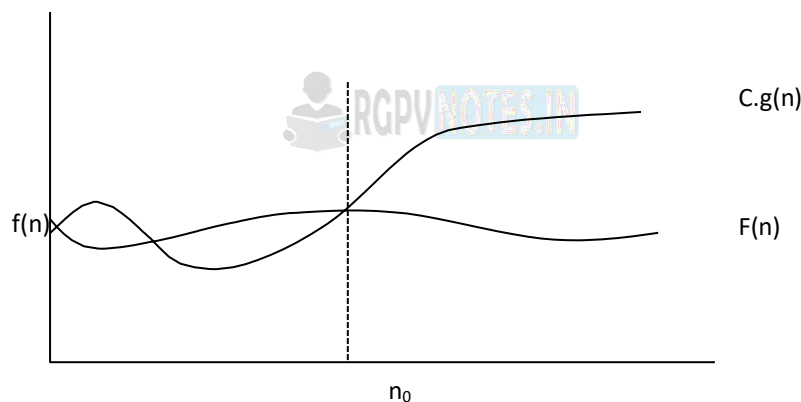


**Figure 1.1: Big O Notation**

**2. Big–OMEGA Ω (Lower Bound)**

f(n) = Ω (g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (>) that of g(n) figure 1.2.
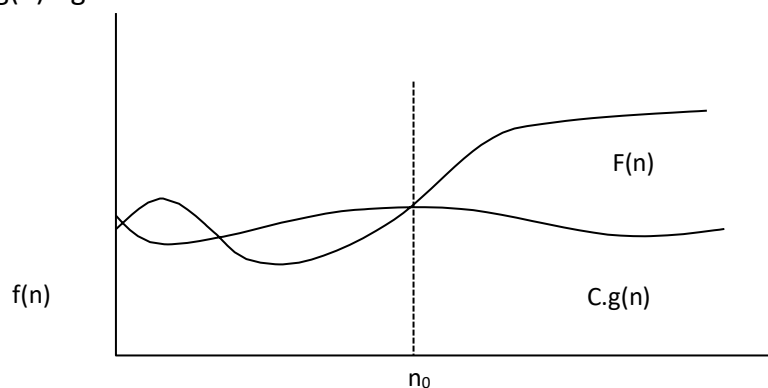


**Figure 1.2: Big Ω Notation**

### 3. Big–THETA Θ (Same order)

f(n) = $\Theta$(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = $\Theta$ (g(n))] figure 1.3.



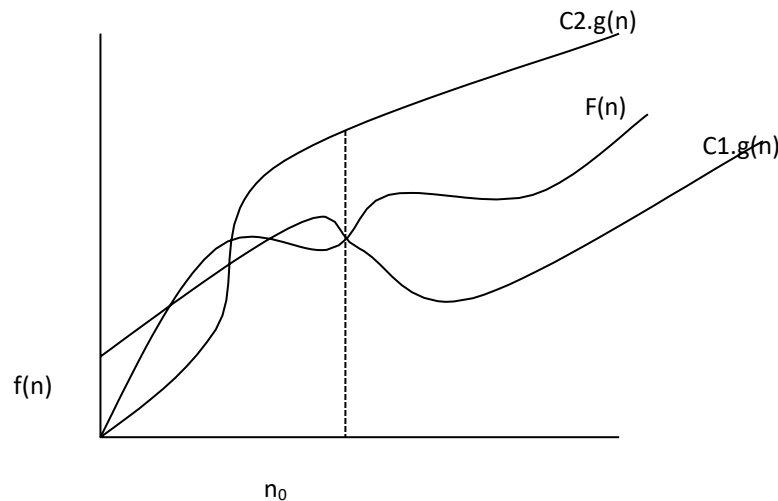**Figure 1.3: Big $\Omega$ Notation**

### Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and $n^n$

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

| n | logn | n*logn | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 2 |
| 2 | 1 | 2 | 4 | 8 | 4 |
| 4 | 2 | 8 | 16 | 64 | 16 |
| 8 | 3 | 24 | 64 | 512 | 256 |
| 16 | 4 | 64 | 256 | 4096 | 65,536 |
| 32 | 5 | 160 | 1024 | 32,768 | 4,294,967,296 |
| 64 | 6 | 384 | 4096 | 2,62,144 | Note 1 |
| 128 | 7 | 896 | 16,384 | 2,097,152 | Note 2 |
| 256 | 8 | 2048 | 65,536 | 1,677,216 | ???????? |

**Table 1.2: Comparison among various complexities**

## HEAP AND HEAP SORT:

A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

- Complete Binary tree
- Heap Order

It is implemented as an array where each node in the tree corresponds to an element of the array.

**Binary Heap:**

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.
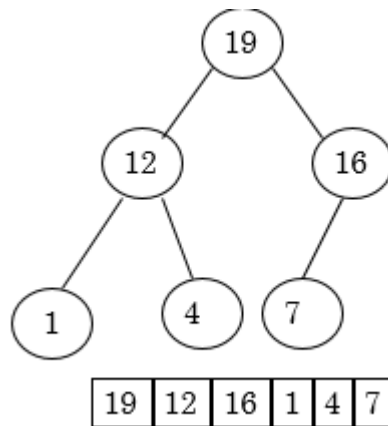


**Figure 1.4: Heap**

- The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed

  PARENT (*i*)
  > return floor(*i*/2)

  LEFT (*i*)
  > return 2*i*
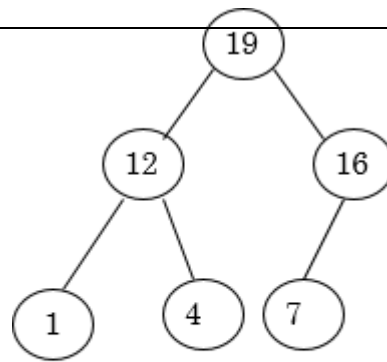
  RIGHT (*i*)
  > return 2*i* + 1

**Types of Heaps**

Heap can be of 2 types:

**Max Heap**

- Store data in ascending order
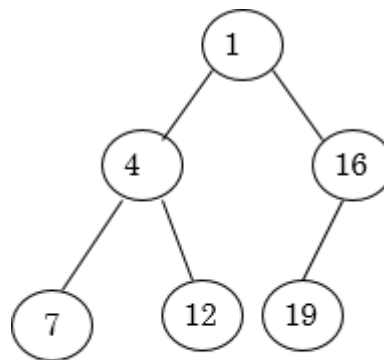- Has property of
  A[Parent(i)] ≥ A[i]

**Min Heap**

- Store data in descending order
- Has property of
  A[Parent(i)] ≤ A[i]

Array A

**Figure 1.5: Max Heap example**



Array A

**Figure 1.6: Min Heap example**

**Heap sort** is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

**Heap Sort Algorithm for sorting in increasing order:**
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

**Procedures on Heap**
- Heapify
- Build Heap
- Heap Sort

## 1. Heapify

Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
    l ← left(i)
    r ← right(i)
    if l <= heapsize[A] and A[l] > A[i]
        then largest ←l
        else largest ← i
    if r <= heapsize[A] and A[r] > A[largest]
        then largest ← r
    if largest != i
        then swap A[i] ←→ A[largest]
            Heapify(A, largest)
}
```

## 2. Build Heap

We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A)
{
    heapsize[A] ←length[A]
    for i ←|length[A]/2 //down to 1
        do Heapify(A, i)
}
```

## 3. Heap Sort

The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
Heapsort(A)
{
    Buildheap(A)
    for i ← length[A] //down to 2
        do swap A[1] ←→ A[i]
        heapsize[A] ← heapsize[A] - 1
        Heapify(A, 1)
}
```

## Complexity

Time complexity of heapify is O(Logn). Time complexity of create and BuildHeap() is O(n) and overall time complexity of Heap Sort is O(n Logn).

## INTRODUCTION TO DIVIDE AND CONQUER TECHNIQUE:

**Divide & conquer technique** is a top-down approach to solve a problem.

The algorithm which follows divide and conquer technique involves 3 steps:

- Divide the original problem into a set of sub problems.
- Conquer (or Solve) every sub-problem individually, recursive.
- Combine the solutions of these sub problems to get the solution of original problem.

## BINARY SEARCH:

The Binary search technique is a search technique which is based on Divide & Conquer strategy. The entered array must be sorted for the searching, then we calculate the location of mid element by using formula mid= (Beg + End)/2, here Beg and End represent the initial and last position of array. In this technique we compare the Key element to mid element. So there May be three cases:-

1. If array[mid] = = Key (Element found and Location is Mid)
2. If array[mid] > Key, then set End = mid-1. (continue the process)
3. If array [mid] < Key, then set Beg=Mid+1. (Continue the process)

## Binary Search Algorithm

1. [Initialize segment variable] set beg=LB,End=UB and Mid=int(beg+end)/2.
2. Repeat step 3 and 4 while beg<=end and Data[mid] != item.
3. If item< data[mid] then set end=mid-1

    Else if Item>data[mid] then set beg=mid+1[end of if structure]
4. Set mid= int(beg+end)/2.[End of step 2 loop]
5. If data[mid]=item then set Loc=

    Mid. Else set loc=null[end of

    if structure]
6. Exit.

### Time complexity

As we dispose off one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of $O(\log_2 n)$.

## MERGE SORT:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

- Divide the unsorted list into NN sublists, each containing 11 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. NN will now convert into N/2N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

**How Merge Sort Works?**

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.
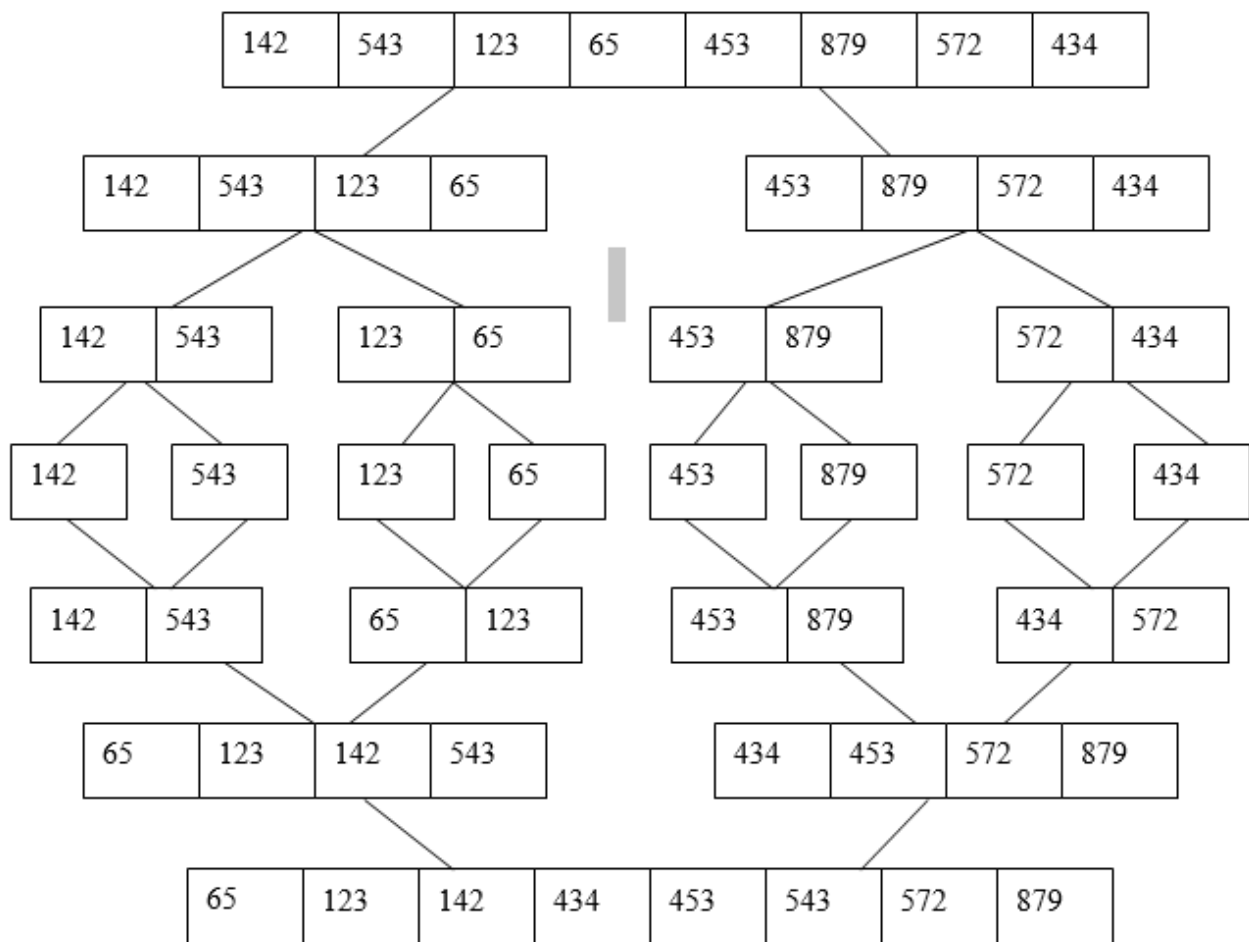


**Figure 1.7: Merge Sort**

**Algorithm:**

```
MergeSort(A, p, r)
{
    if( p < r )
    {
        q = (p+r)/2;
        mergeSort(A, p, q);
        mergeSort(A, q+1, r);
        merge(A, p, q, r);
    }
}

Merge (A, p, q, r )
{
    n₁ = q – p + 1
```

$n_1 = q - p + 1$

```
        n₂ = r − q
        declare L[1…n₁ + 1] and [R1…n₂ + 1] temporary arrays
        for i = 1 to n₁
           L[i] = A[p + i - 1]
        for j = 1 to n₂
      R[j] = numbers[q+ j]
      L[n₁ + 1] = ∞
      R[n₂ + 1] = ∞
      i = 1
      j = 1
     for k = p to r
         If (L[i] ≤ R[j])

            A[k] = L[i]

             i = i + 1
         else
            A[k] = R[j]
            j = j + 1
}
```

## Time Complexity:

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $O(nLogn)$.

Time complexity of Merge Sort is $O(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: $O(n)$

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

## QUICK SORT:

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
/* low  --> Starting index,  high  --> Ending index */

## Quick Sort Algorithm

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;   // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

## Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

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

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

**Worst Case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + 0(n)$

which is equivalent to

$T(n) = T(n-1) + 0(n)$

The solution of above recurrence is $0(n^2)$.

**Best Case:** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + 0(n)$

The solution of above recurrence is $0(nLogn)$. It can be solved using case 2 of Master Theorem.

**Average Case:**

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + 0(n)$

Solution of above recurrence is also $O(nlogn)$

Although the worst-case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**STRASSEN'S MATRIX MULTIPLICATION:**

The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. We've seen so far, some divide and conquer algorithms like merge sort and the Karatsuba's fast multiplication of large numbers. However, let's get again on what's behind the divide and conquer approach.

Unlike the dynamic programming where we "expand" the solutions of sub-problems in order to get the final solution, here we are talking more on joining sub-solutions together. These solutions of some sub-problems of the general problem are equal and their merge is somehow well defined.

**General Algorithm without Strassen's**

MMult(A,B, n)

1. If n = 1 Output A × B
2. Else
3. Compute A11,B11, . . . ,A22,B22 % by computing m = n/2
4. X1  MMult(A11,B11, n/2)
5. X2  MMult(A12,B21, n/2)
6. X3  MMult(A11,B12, n/2)
7. X4  MMult(A12,B22, n/2)
8. X5  MMult(A21,B11, n/2)

9. X6  MMult(A22,B21, n/2)
10. X7  MMult(A21,B12, n/2)
11. X8  MMult(A22,B22, n/2)
12. C11  X1 + X2
13. C12  X3 + X4
14. C21  X5 + X6
15. C22  X7 + X8
16. Output C
17. End If

Complexity of above algorithm is: $T(n) = 0 (n^{\log_2 (8)}) = 0 (n^3)$

**Strassen Multiplication Algorithm**
Strassen(A,B)
1. If n = 1 Output A × B
2. Else
3. Compute A11,B11, . . . ,A22,B22 % by computing m = n/2
4. P1  Strassen(A11,B12 − B22)
5. P2  Strassen(A11 + A12,B22)
6. P3  Strassen(A21 + A22,B11)
7. P4  Strassen(A22,B21 − B11)
8. P5  Strassen(A11 + A22,B11 + B22)
9. P6  Strassen(A12 − A22,B21 + B22)
10. P7 Strassen(A11 − A21,B11 + B12)
11. C11  P5 + P4 − P2 + P6
12. C12  P1 + P2
13. C21  P3 + P4
14. C22  P1 + P5 − P3 − P7
15. Output C
16. End If

Complexity = $T(n) = 0 (n^{\log_2 (7)}) = 0 (n^{2.8})$

---

# Unit-2 Notes

Study of Greedy strategy, examples of greedy method like optimal merge patterns, Huffman coding, minimum spanning trees, knapsack problem, job sequencing with deadlines, single source shortest path algorithm

### Greedy Technique

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called subset paradigm. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on subset paradigm.

### Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
solution := Ø;  // initialize the solution to empty for i:=1 to n do
{
x := select (a);
if feasible (solution, x) then
solution := Union (Solution, x);
}
return solution;
}
```

### OPTIMAL MERGE PATTERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
treenode * lchild;
treenode * rchild;
};

Algorithm TREE (n)
// list is a global of n single node binary trees
{
for i := 1 to n – 1 do
{
pt  = new treenode
(pt.lchild) = least (list);          //        merge two trees  with  smallest lengths
(pt.rchild) = least (list);
(pt.weight) = ((pt.lchild).weight) + ((pt.rchild).weight);
insert (list, pt);
tree
}
}
return least (list);
```

**Analysis:**
T= O (n-1) * max (O (Least), O (Insert)).

- Case 1: L is not sorted.
O (Least)= O (n).
O (Insert)= O (1).
T= O ($n^2$).

- Case 2: L is sorted.
Case 2.1

        O (Least)= O (1)

        O (Insert)= O (n)

    T= O ($n^2$)

Case 2.2

    L is represented as a min-heap.  Value in the root is <= the values of its children.

O (Least)= O (1)

O (Insert)= O (log n)

T= O (n log n).

## Huffman Codes

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab" figure 2.1.

## Steps to build Huffman code

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

**1.** Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

**2.** Extract two nodes with the minimum frequency from the min heap.

**3.** Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

**4.** Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

**Example:**

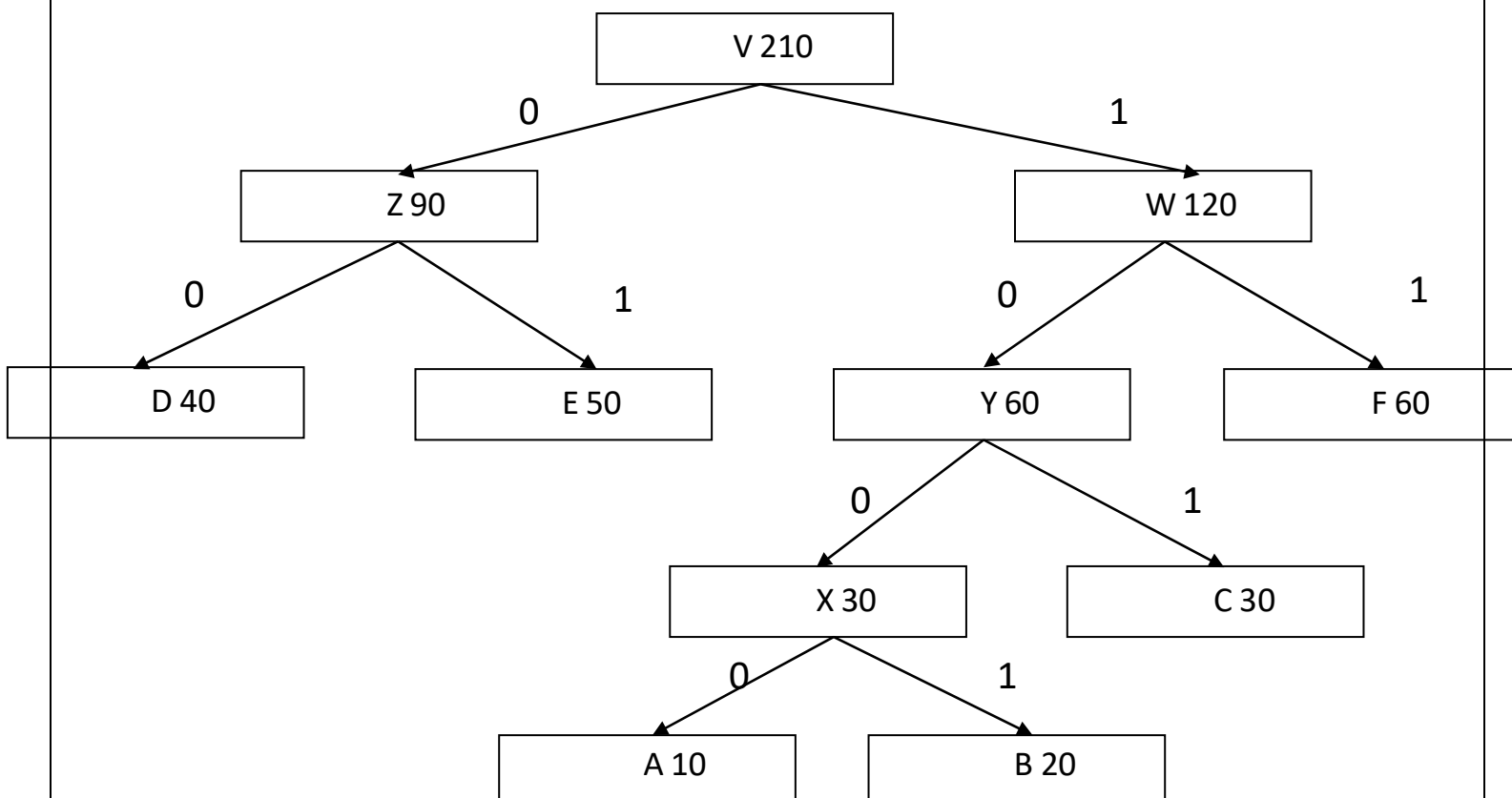| Letter | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Frequency | 10 | 20 | 30 | 40 | 50 | 60 |

Figure 2.1: Example of Huffman code

**Output => A:1000, B: 10001, C: 101, D: 00, E: 01, F:11**

**Algorithm:**
```
   Huffman(A)
   {
       n = |A|;
       Q = A;
                 for i = 1 to n-1
  {
               z = new node;
               left[z] =Extract-Min(Q);
               right[z] =Extract-Min(Q);
               f[z] = f[left[z]] +f[right[z]];
               Insert(Q, z);
       }
   return Extract-Min(Q);
   }
```

**Analysis of algorithm**
Each priority queue operation (e.g. heap): O(log n)
In each iteration: one less subtree.
Initially: n subtrees.
Total: O(n log n) time.

**Kruskal's Algorithm**
This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).
Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

**Algorithm Kruskal (E, cost, n, t)**
```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
Construct a heap out of the edge costs using heapify;
for i := 1 to n do parent [i] := -1;
i := 0; mincost := 0.0;
// Each vertex is in a different set.
while ((i < n -1) and (heap not empty)) do
{
Delete a minimum cost edge (u, v) from the heap and re-heapify using Adjust;
j := Find (u); k := Find (v);
if (j < k) then
{
i := i + 1;
```

```
t [i, 1] := u; t [i, 2] := v; mincost :=mincost + cost [u, v]; Union (j, k);
}
}
if (i >n-1) then write ("no spanning tree");
else return mincost;
}
```

Running time:
- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity.

## MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an
Minimal cost spanning tree.
Prim's algorithm is an example of a greedy algorithm.

## Algorithm Prim (E, cost, n, t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
Let (k, l) be an edge of minimum cost in E;
mincost := cost [k, l];
t [1, 1] := k; t [1, 2] := l;
for    i :=1 to n do      // Initialize near if (cost [i, l] < cost [i, k]) then near [i] := l;
else near [i] := k;
near [k] :=near [l] := 0;
for i:=2 to n -  1 do      // Find n - 2 additional edges for t.
{
Let j be an index such that near [j] 0 and
cost [j, near [j]] is minimum;
t [i, 1] := j; t [i, 2] := near [j];
mincost := mincost + cost [j, near [j]];
near [j] := 0
for    k:= 1 to n do      // Update near[].
if ((near [k] > 0) and (cost [k, near [k]] > cost [k, j]))
then near [k] := j;
}
```

return mincost;
}

Running time:
We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n2) time when we implement dist with array, O (n + E.log n) when we implement it with a heap.

**Comparison of Kruskal's and Prim's MCST Algorithm:**

| Kruskal's Algorithm | Prim's algorithm |
| --- | --- |
| • Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST.<br>• In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps).<br>• At intermediate step of algorithm, there are may be more than one connected components are possible.<br>• Time complexity: O (\|E\| log \|V\|) | • Prim's algorithm always selects a vertex (say, v) to find MCST.<br>• In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps).<br>• At intermediate step of algorithm, there will be only one connected components are possible<br>• Time complexity: O ($V^2$) |

**KNAPSACK PROBLEM**

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight wi and the knapsack has a capacity 'm'. If a fraction xi, 0 < xi < 1 of object i is placed into the knapsack then a profit of pi xi is earned. The objective is to fill the knapsack that maximizes the total profit earned.
Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'.

**Algorithm**
If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

```
            Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
                 /* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered
                      such that
                 X[1..n] is a solution set and M is the capacity of KnapSack*/
            {
1:    For i ← 1 to n do
2:    X[i] ← 0
3:    profit ← 0              //Total profit of item filled in Knapsack
4:    weight ← 0             // Total weight of items packed in KnapSack
5:    i←1
6:    While (Weight < M) // M is the Knapsack Capacity
                {
7:    if (weight + W[i] ≤ M)
8:    X[i] = 1
9:    weight = weight + W[i]
10: else
```

```
11:  X[i] = (M-weight)/w[i]
12:  weight = M
13:  Profit = profit = profit + p [i]*X[i]
14:  i++;
          }//end of while
          }//end of Algorithm
```

Running time:
The objects are to be sorted into non-decreasing order of pi / wi ratio. But if we disregard the time to initially sort the objects, the algorithm requires O(nlogn) time.

**JOB SEQUENCING WITH DEADLINES**
When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i > 0$ and profit $P_i > 0$. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.
Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \le r \le k$ are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i  into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, $1 \le r \le k+1$.

Algorithm GreedyJob (d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
J := {1};
for i := 2 to n do
{
if (all jobs in J U {i} can be completed by their dead lines)
then J := J U {i};
}
}

We still have to discuss the running time of the algorithm. The initial sorting can be done in  time O(n log n), and the rest loop takes time O(n). It is not hard to implement  each body of the second loop in time O(n), so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to O(n log n), but in any case it is a polynomial-time algorithm.

**The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS**
In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.
Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees.
Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.
Dijkstra's algorithm does not work for negative edges at all.

**Algorithm Shortest-Paths (v, cost, dist, n)**
// dist [j], 1 < j < n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{

```
for i :=1 to n do
{
S [i] := false;  // Initialize S. dist [i] :=cost [v, i];
}
S[v] := true; dist[v]  := 0.0;       // Put v in S. for num := 2 to n – 1 do
{
Determine n - 1 paths from v.
Choose u from among those vertices not in S such that dist[u] is minimum; S[u] := true; // Put u is S.
for (each w adjacent to u with S [w] = false)
do
if (dist [w] > (dist [u] + cost [u,  w]) then         // Update distances dist [w] := dist [u] + cost [u, w];
}
}
```

Running time:
For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

Concept of dynamic programming, problems based on this approach such as 0/1 knapsack, multistage graph, reliability design, Floyd Warshall algorithm

# Unit-3

**Concept of Dynamic Programming**

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called dynamic-programming recurrence equations, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

• Verify that the principle of optimality holds

• Set up the dynamic-programming recurrence equations

• Solve the dynamic-programming recurrence equations for the value of the optimal solution.
• Perform a trace back step in which the solution itself is constructed.

**0/1 – KNAPSACK**

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] andwt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Dynamic-0-1-knapsack (v, w, n, W)

for w = 0 to W do

  c[0, w] = 0

for i = 1 to n do

  c[i, 0] = 0

```
    for w = 1 to W do
      if wi ≤ w then
        if vi + c[i-1, w-wi] then
          c[i, w] = vi + c[i-1, w-wi]
        else c[i, w] = c[i-1, w]
      else
        c[i, w] = c[i-1, w]
```

## MULTI STAGE GRAPHS

A multistage graph is a graph
- G=(V,E) with V partitioned into K >= 2 disjoint subsets such that if (a,b) is in E, then a is in $V_i$, and b is in $V_{i+1}$ for some subsets in the partition;
- and $| V_1 | = | V_K | = 1$.

The vertex s in $V_1$ is called the source; the vertex t in $V_K$ is called the sink.

G is usually assumed to be a weighted graph.

The cost of a path from node v to node w is sum of the costs of edges in the path.

The "multistage graph problem" is to find the minimum cost path from s to t.

Dynamic Programming solution:

Let path(i,j) be some specification of the minimal path from vertex j in set i to vertex t; C(i,j) is the cost of this path; c(j,t) is the weight of the edge from j to t.

To write a simple algorithm, assign numbers to the vertices so those in stage Vi have lower number those in stage Vi+1.

```
  int[] MStageForward(Graph G)
  {
    // returns vector of vertices to follow through the graph
    // let c[i][j] be the cost matrix of G

    int n = G.n (number of nodes);
    int k = G.k (number of stages);
    float[] C = new float[n];
    int[] D = new int[n];
    int[]  P = new int[k];
    for (i = 1 to n) C[i] = 0.0;
    for j = n-1 to 1 by -1 {
      r = vertex such that (j,r) in G.E and c(j,r)+C(r) is minimum
      C[j] = c(j,r)+C(r);
      D[j] = r;
    }
    P[1] = 1; P[k] = n;
    for j = 2 to k-1 {
      P[j] = D[P[j-1]];
    }
    return P;
  }
```

## Time complexity:

Complexity is O (|V| + |E|). Where the |V| is the number of vertices and |E| is the number of edges.

**Example**

Consider the following example to understand the concept of multistage graph.
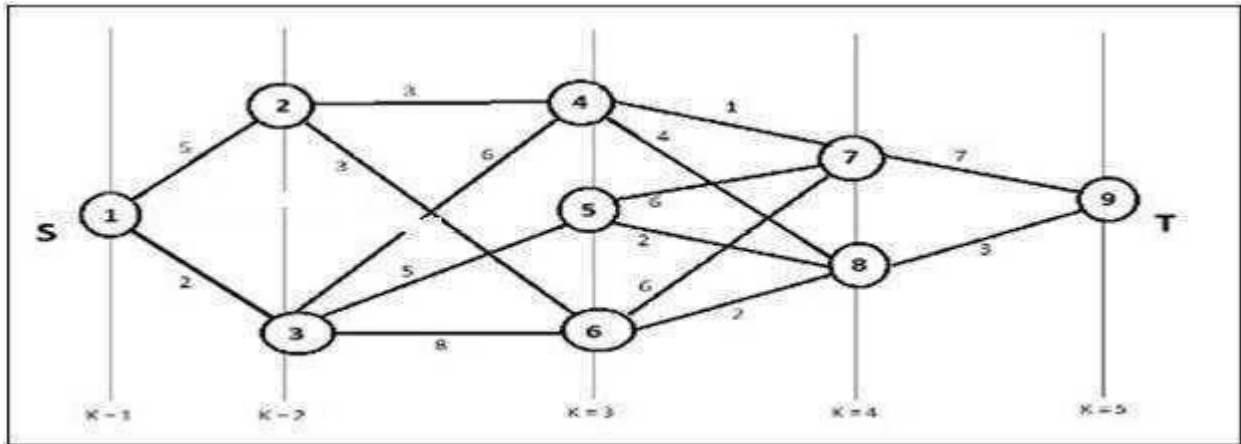


**Figure 3.1 Multistage Graph**

**Step-1: Cost (K-2, j)**

In this step, three nodes (node 4, 5. 6) are selected as j. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

**Step-2: Cost (K-3, j)**

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

**Step-3: Cost (K-4, j)**

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

**Hence, the path having the minimum cost is 1& 3& 5& 8& 9**


**RELIABILITY DESIGN**

In **reliability design**, the problem is to design a system that is composed of several devices connected in series.
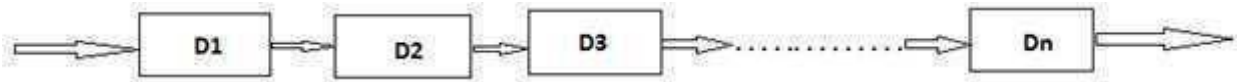
**Figure 3.2 Reliability Design(Serial Connection)**

If we imagine that r1 is the reliability of the device.Then the reliability of the function can be given by $ur1$.If $r1 = 0.99$ and $n = 10$ that **n** devices are set in a series, $1 <= i <= 10$, then reliability of the whole system **uri** can be given as: $\Pi ri = 0.904$

So, if we duplicate the devices at each stage then the reliability of the system can be increased.

It can be said that multiple copies of the same device type are connected in parallel through the use of switching circuits. Here, switching circuit determines which devices in any given group are functioning properly. Then they make use of such devices at each stage, that result is increase in reliability at each stage. If at each stage, there are **mi** similar types of devices **Di**, then the probability that all **mi** have a malfunction is $(1 - ri)^{mi}$, which is very less.

And the reliability of the stage I becomes $(1 - (1 - ri)^{mi})$. Thus, if $ri = 0.99$ and $mi = 2$, then the stage reliability becomes **0.9999** which is almost equal to **1**. Which is much better than that of the previous case or we can say the reliability is little less than $1 - (1 - ri)^{mi}$ because of less reliability of switching circuits.
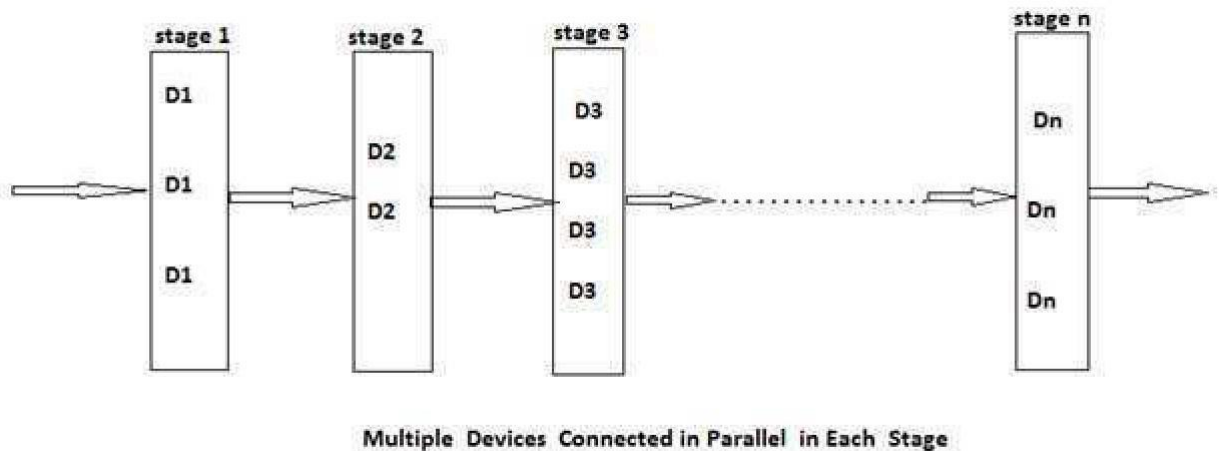


Multiple Devices Connected in Parallel in Each Stage

**Figure 3.3 Reliability Design(Parallel Connection)**

In reliability design, we try to use device duplication to maximize reliability. But this maximization should be considered along with the cost.

### FLOYD WARSHALL ALGORITHM
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

**All pairs shortest paths**

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n²) time, the matrix A can be obtained in O (n³) time.

Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 < i < n.
{
for i := 1 to n do
for j:= 1 to n do
A [i, j] := cost  [i, j]; // copy cost into A. for k := 1 to n do
for i := 1 to n do
for j := 1 to n do
A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}

**Complexity Analysis:**
A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n³).

**Problem-**
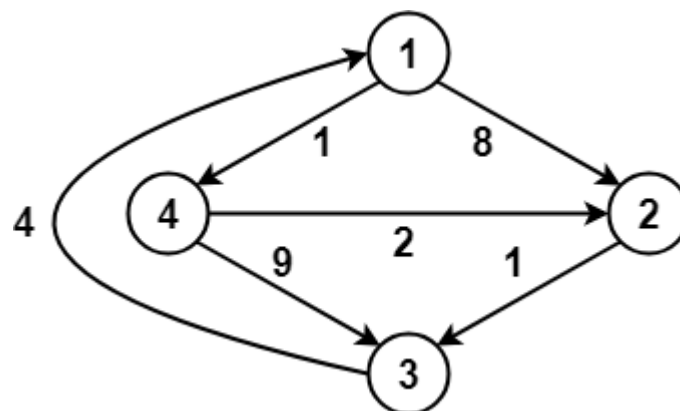Consider the following directed weighted graph-



**Figure 3.4 Floyd Warshall Example**

_Solution-_

**Step-01:**

➢ Remove all the self loops and parallel edges (keeping the edge with lowest weight) from the graph if any.

**Step-02:**

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

➤ For diagonal elements (representing self-loops), value = 0

➤ For vertices having a direct edge between them, value = weight of that edge

➤ For vertices having no direct edges between them, value = ∞

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

**Step-03:**

The four matrices are-

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

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

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

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

The last matrix D4 represents the shortest path distance between every pair of vertices.

# Unit-4

**Syllabus:** Backtracking concept and its examples like 8 queen's problem, Hamiltonian cycle, Graph coloring problem etc. Introduction to branch & bound method, examples of branch and bound method like traveling salesman problem etc. Meaning of lower bound theory and its use in solving algebraic problem, introduction to parallel algorithms.

## BACKTRACKING:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple $(x_1, \quad , x_n)$ where each $x_i \, C \, S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, \quad , x_n)$. Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

## 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.
- **Answer states** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.
- **State space** is the set of paths from root node to other nodes. State space tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.
- **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.
- Depth first node generation with bounding functions is called backtracking. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

## N-QUEENS PROBLEM:

The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x
that no two "attack", that is, no two of them are on the same row, column, or diagonal.
All solutions to the 8-queens problem can be represented as 8-tuples (x1. . . x8), where xi is
the column of the $i_{th}$ row where the $i_{th}$ queen is placed.
The promising function must check whether two queens are in the same column or diagonal:
Suppose two queens are placed at positions (i, j) and (k, l) Then:

- **Column Conflicts:** Two queens conflict if their xi values are identical.
- **Diagonal conflict:** Two queens i and j are on the same diagonal if: $i - j = k - l$.
  This implies, $j - l = i - k$
- **Diagonal conflict:** $i + j = k + l$. This implies, $j - l = k - i$

**Algorithm:**
1) Start in the leftmost column
2) If all queens are placed return true
3) Try all rows in the current column. Do following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row, column] as part of the
      Solution and recursively check if placing queen here leads to a solution.
   b) If placing queen in [row, column] leads to a solution then return true.
   c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go
      to step (a) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

## 8-QUEENS PROBLEM:

The eight queen's problem is the problem of placing eight queens on an 8×8 chessboard such that
none of them attack one another (no two are in the same row, column, or diagonal).

| Algorithm for new queen be placed | All solutions to the n·queens problem |
|---|---|
| Algorithm Place(k,i) | Algorithm NQueens(k, n) |
| //Return true if a queen can be placed | // its prints all possible placements of n- |
|    in kth row & ith column |    queens on an n×n chessboard. |
| //Other wise return false | { |
| { | for i:=1 to n do{ |
| for j:=1 to k-1 do | if Place(k,i) then |
| if(x[j]=i or Abs(x[j]-i)=Abs(j-k))) | { |
| then return false | X[k]:=I; |
| return true | if(k==n) then write (x[1:n]); |
| } | else NQueens(k+1, n); |
| | } |
| | }} |

Figure-4.1: 8-Queens Solution

### HAMILTONIAN CYCLES:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position.

In graph G, Hamiltonian cycle begins at some vertiex $v1 \in G$ and the vertices of G are visited in the order $v_1, v_2, ---v_{n+1}$, then the edges $(v_i, v_{i+1})$ are in E, $1 \le i \le n$.
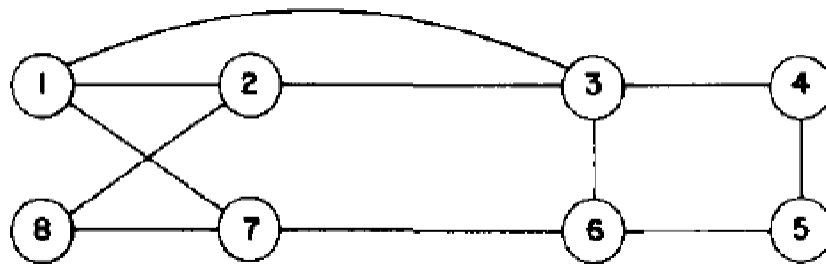

Figure-4.2: Example of Hamiltonian cycle

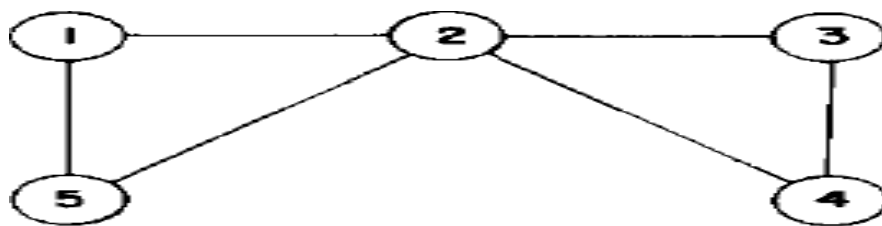The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1


Figure-4.3: Graph

The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
- Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
- The graph may be directed or undirected. Only distinct cycles are output.
- From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}

- The backtracking solution vector $(x_1, x_2, \text{---} x_n)$
  - $x_i$ $i^{th}$ visited vertex of proposed cycle.
- By using backtracking we need to determine how to compute the set of possible vertices for $x_k$ if $x_1, x_2, x_3 \text{---} x_{k-1}$ have already been chosen.
  - If k=1 then x1 can be any of the n-vertices.

By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.

This algorithm is started by 1$^{st}$ initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

| Generating Next Vertex | Finding all Hamiltonian Cycles |
|---|---|
| Algorithm NextValue(k) <br> { <br> // x[1: k-1] is path of k-1 distinct vertices. <br> // if x[k]=0, then no vertex has yet been assigned to x[k] <br> Repeat{ <br> X[k]=(x[k]+1) mod (n+1); //Next vertex <br> If(x[k]=0) then return; <br> If(G[x[k-1], x[k]]≠O) then <br> { <br> For j:=1 to k-1 do if(x[j]=x[k]) then break; <br> //Check for distinctness <br> If(j=k) then //if true , then vertex is distinct <br> If((k<n) or (k=n) and G[x[n], x[1]]≠O)) <br> Then return ; <br> } <br> } <br>    Until (false); <br>    } | Algorithm Hamiltonian(k) <br> { <br> Repeat{ <br> NextValue(k); //assign a legal next value <br>        to x[k] <br> If(x[k]=0) then return; <br> If(k=n) then write(x[1:n]); <br> Else Hamiltonian(k+1); <br> }until(false) <br> } |

**Complexity Analysis**

In Hamiltonian cycle, in each recursive call one of the remaining vertices is selected in the worst case. In each recursive call the branch factor decreases by 1. Recursion in this case can be thought of as n nested loops where in each loop the number of iterations decreases by one. Hence the time complexity is given by:

$$T(N)=N*(T(N-1)+O(1))$$
$$T(N) = N*(N-1)*(N-2).. = O(N!)$$

**GRAPH COLORING :**

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. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Note that, if 'd' is the degree of the given graph then it can be colored with 'd+1' colors.

The m- colorability optimization problem asks for the smallest integer 'm' for which the graph G can be colored. This integer is referred as "Chromatic number" of the graph.
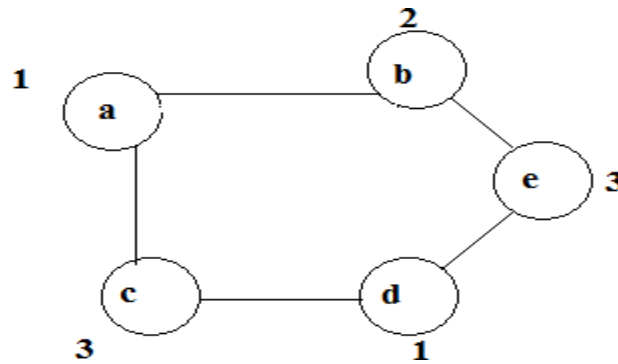
**Example:**



**Figure-4.4: Graph Coloring Example**

**Algorithm:**

| Finding all m-coloring of a graph | Getting next color |
|---|---|
| Algorithm mColoring(k){<br>// g(1:n, 1:n):boolean adjacency matrix.<br>// kindex (node) of the next vertex to color.<br>repeat{<br>nextvalue(k); // assign to x[k] a legal color.<br>if(x[k]=0) then return; // no new color possible<br>if(k=n) then write(x[1: n];<br>else mcoloring(k+1);<br>}<br>until(false)<br>} | Algorithm NextValue(k){<br>//x[1],x[2],---x[k-1]  have  been  assigned integer values in the range [1, m]<br>repeat {<br>x[k]=(x[k]+1)mod  (m+1);  //next  highest color<br>if(x[k]=0) then return; // all colors have been used.<br>for j=1 to n do<br>{<br>if ((g[k,j]≠O) and (x[k]=x[j]))<br>then break;<br>}<br>if(j=n+1) then return; //new color found<br>} until(false)<br>} |

**Complexity Analysis**

**1) 2-colorability**

There is a simple algorithm for determining whether a graph is 2-colorable and assigning colors to its vertices: do a breadth-first search, assigning "red" to the first layer, "blue" to the second layer, "red" to the third layer, etc. Then go over all the edges and check whether the two endpoints of this edge have different colors. This algorithm is $O(|V|+|E|)$ and the last step ensures its correctness.

**2) k-colorability for k>2**

For k > 2 however, the problem is much more difficult. For those interested in complexity theory, it can be shown that deciding whether a given graph is k-colorable for k > 2 is an NP-complete problem. The first algorithm that can  be thought of is brute-force search: consider every possible assignment of k colors to the vertices, and check whether any of them are correct. This of course is very expensive, on the order of $O((n+1)!)$, and impractical. Therefore we have to think of a better algorithm.

## INTRODUCTION TO BRANCH & BOUND METHOD:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

- It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
- It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.
- Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node
- Branch and Bound is the generalization of graph search strategies, BFS and D- search.
  - A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
  - A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

## EXAMPLES OF BRANCH AND BOUND METHOD

There are lots of problem which can be solve using branch and bound methods. Like:

- **Travelling Salesperson Problem**
- 0/1 knapsack
- Quadratic assignment problem
- Nearest neighbor search

## TRAVELLING SALESPERSON PROBLEM

**Definition:** Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.

A tree of nodes is generated where each node has specified constraints regarding edges connecting two cities in a tour that must be present and edges connecting two cities in a tour that cannot be present. Based on the constraints in a given node, a lower bound is formulated for the given node. This lower bound represents the smallest solution that would be possible if a sub-tree of nodes leading eventually to leaf nodes containing legal tours were generated below the given node. If this lower bound is higher than the best known solution to-date, the node may be pruned. This pruning has the effect of sparing result in a significant saving if the pruned node were relatively near the top of the tree.

Let us explore the mechanism for computing lower bounds for a node.

Example:

|   | A | B | C | D |
|---|---|---|---|---|
| A | ∞ | 10 | 5 | 3 |
| B | 8 | ∞ | 9 | 7 |
| C | 1 | 6 | ∞ | 9 |
| D | 2 | 3 | 8 | ∞ |

Now find the reduced matrix by:

• Subtracting the smallest element from row i(for example r1), so one element will become 0 and rest element remain non negative.

• Then after subtracting the smallest element from col j(for example c1), so one element will

become 0 and rest element remain non negative.

• Set element A[I,j] = ∞

So the total reduced cost T= r1 + c1.

Hence the reduced matrix is:

• Subtracted 3 from row 1

• Subtracted 7 from row 2

• Subtracted 1 from row 3

• Subtracted 2 from row 4

• Subtracted 1 from col 2

• Subtracted 2 from col 3

|   | A | B | C | D |
|---|---|---|---|---|
| A | ∞ | 6 | 0 | 0 |
| B | 1 | ∞ | 0 | 0 |
| C | 0 | 4 | ∞ | 8 |
| D | 0 | 0 | 4 | ∞ |

Total reduced cost is = 3+7+1+2+1+2

We examine minimum cost for each node 1,….,4 by using the formula-*l(B)=l(A) + M(i,j) + T*

Here:- l(b) is the cost of new node, l(A) is the cost of previous node, and T is the reduced cost
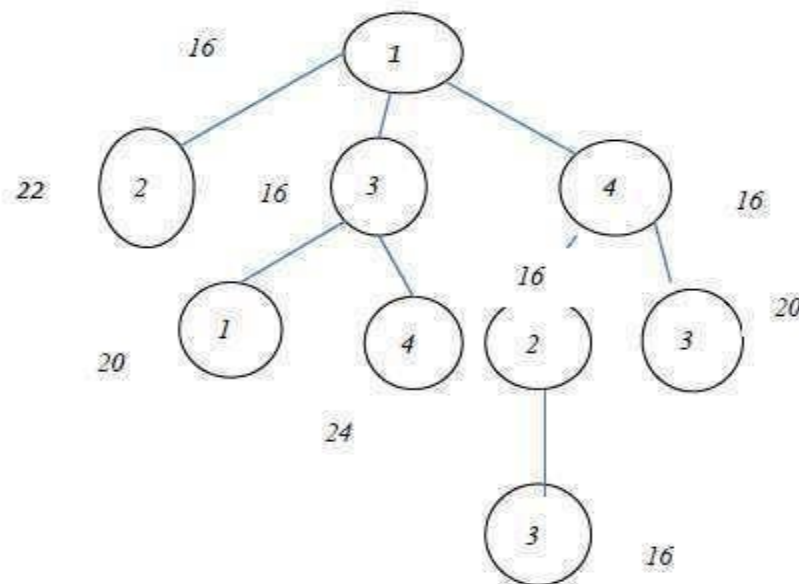
So the state space tree is given as:



Figure-4.5: State Space Tree for TSP

Here the path is using above state space tree is: 1->4->2->3->1

**Algorithm for TSP**

**function CheckBounds(st,des,cost[n][n])**

  Global variable: cost[N ][N ] - the cost assignment.

  pencost[0] = t

  for i C O, n – 1 do

  for j C O, n – 1 do

  reduced[i][j] = cost[i][j]

  end for end for

```
for j C O, n – 1 do

reduced[st][j] = ∞

end for

for i C O, n – 1 do

reduced[i][des] = ∞

end for reduced[des][st] = ∞ RowReduction(reduced) C olumnReduction(reduced)

pencost[des] = pencost[st] + row + col + cost[st][des]

return  pencost[des]

 end function

function RowMin(cost[n][n],i)

min = cost[i][0]

for j C O, n – 1 do

if cost[i][j] < min then

min = cost[i][j]

end if end for return  min

end function


function ColMin(cost[n][n],i)
min = cost[0][j]

for i C O, n – 1 do

if cost[i][j] < min then

min = cost[i][j]

end if end for return  min

end function

function Rowreduction(cost[n][n])

row = 0

for i C O, n – 1 do

rmin = rowmin(cost, i)

if  rmin  /= ∞ then

row = row + rmin

end if

for j C O, n – 1 do

if cost[i][j] /= ∞ then

cost[i][j] = cost[i][j] – rmin
```

end if end for

**end for end function**

**function Columnreduction(cost[n][n])**

col = 0

for j C O, n – 1 do

cmin = columnmin(cost, j)

if cmin /= ∞ then

col = col + cmin

end if

for i C O, n – 1 do

if cost[i][j] /= ∞ then

cost[i][j] = cost[i][j] – cmin

end if end for

**end for end function**

**function Main**

for i C O, n – 1 do

select[i] = 0 end for rowreduction(cost) columnreduction(cost) t = row + col

while all visited(select) != 1 do for i C 1, n – 1 do

if select[i] = 0 then

edgecost[i] = checkbounds(k, i, cost)

end if end for min = ∞

for i C 1, n – 1 do

if select[i] = 0 then

if edgecost[i] < min then

min = edgecost[i]

k = i

end if end if

end for

select[k] = 1

for p C 1, n – 1 do

cost[j][p] = ∞

end for

for p C 1, n – 1 do

cost[p][k] = ∞

end for cost[k][j] = ∞ rowreduction(cost) columnreduction(cost)

**end while end function**

**Complexity Analysis:**
Traveling salesman problem is a NP-hard problem. Until now, researchers have not found a polynomial time algorithm for traveling salesman problem. Among the existing algorithms, dynamic programming algorithm can solve the problem in time $O(n^2*2^n)$ where n is the number of nodes in the graph. The branch-and-cut algorithm has been applied to solve the problem with a large number of nodes. However, branch-and-cut algorithm also has an exponential worst-case running time.

## LOWER AND UPPER BOUND THEORY:

Lower bound is the best case running time. Lower bound is determined by the easiest input. It provides a goal for all input. Whereas upper bound is the worst case and is determined by the most difficult input for a given algorithm. Upper bounds provide guarantees for all inputs i.e. Guarantees on performance of the algorithm when run on different inputs will not perform any worse than over the most difficult input.

There are a number of lower bounds for problems related to sorting, for example element-distinctness: are there two identical elements in a set? These lower bounds are actually interesting because they generalize the comparison-lower bound to more algebraic formulations: for example, you can show that solving element distinctness in a model that allows algebraic operations has a lower bound via analyzing the betti numbers of the space induced by different answers to the problem.

A very interesting example of an unconditional exponential deterministic lower bound (i.e not related to P vs NP) is for estimating the volume of a polytope. There is a construction due to Furedi and Barany that shows that the volume of a convex polytope cannot be approximated to within an even exponential factor in polynomial time unconditionally. This is striking because there are randomized poly-time algorithms that yield arbitrarily good approximations.

• **Lower Bound,** L(n), is a property of the specific problem, i.e. sorting problem, MST, matrix multiplication, not of any particular algorithm solving that problem.

• Lower bound theory says that no algorithm can do the job in fewer than L(n) time units for arbitrary inputs, i.e., that every comparison-based sorting algorithm must take at least L(n) time in the worst case.

• L(n) is the minimum over all possible algorithms, of the maximum complexity.

• **Upper bound** theory says that for any arbitrary inputs, we can always sort in time at most U(n). How long it would take to solve a problem using one of the known Algorithms with worst-case input gives us a upper bound.

• Improving an upper bound means finding an algorithm with better worst-case performance.

• U(n) is the minimum over all known algorithms, of the maximum complexity.

• Both upper and lower bounds are minima over the maximum complexity of inputs of size n.

• The ultimate goal is to make these two functions coincide. When this is done, the optimal algorithm will have L(n) = U(n).

**There are few techniques for finding lower bounds**

**1) Trivial Lower Bounds:**
For many problems it is possible to easily observe that a lower bound identical to n exists, where n is the number of inputs (or possibly outputs) to the problem.
• The method consists of simply counting the number of inputs that must be examined and the number of outputs that must be produced, and note that any algorithm must, at least, read its inputs and write its outputs.

**2) Information Theory:**
The information theory method establishing lower bounds by computing the limitations on information gained by a basic operation and then showing how much information is required before a given problem is solved.
• This is used to show that any possible algorithm for solving a problem must do some minimal amount of work.
• The most useful principle of this kind is that the outcome of a comparison between two items contains one bit of information.

**3) Decision Tree Model**
• This method can model the execution of any comparison based problem. One tree for each input size *n*.
• View the algorithm as splitting whenever it compares two elements.
• The tree contains the comparisons along all possible instruction traces.
• The running time of the algorithm = the length of the path taken.
• Worst-case running time = the height of tree.

**PARALLEL ALGORITHM:**
A parallel algorithm can be executed simultaneously on many different processing devices and then combined together to get the correct result. Parallel algorithms are highly useful in processing huge volumes of data in quick time. This tutorial provides an introduction to the design and analysis of parallel algorithms. In addition, it explains the models followed in parallel algorithms, their structures, and implementation.
An algorithm is a sequence of steps that take inputs from the user and after some computation, produces an output. A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

**Concurrent Processing**
The easy availability of computers along with the growth of Internet has changed the way we store and process data. We are living in a day and age where data is available in abundance. Every day we deal with huge volumes of data that require complex computing and that too, in quick time. Sometimes, we need to fetch data from similar or interrelated events that occur simultaneously. This is where we require concurrent processing that can divide a complex task and process it multiple systems to produce the output in quick time.
Concurrent processing is essential where the task involves processing a huge bulk of complex data. Examples include – accessing large databases, aircraft testing, astronomical calculations, atomic and nuclear physics, biomedical analysis, economic planning, image processing, robotics, weather forecasting, web-based services, etc.

Parallelism is the process of processing several set of instructions simultaneously. It reduces the total computational time. Parallelism can be implemented by using parallel computers, i.e. a computer with many processors. Parallel computers require parallel algorithm, programming languages, compilers and operating system that support multitasking.

# Unit-5

**Syllabus:** Binary search trees, height balanced trees, 2-3 trees, B-trees, basic search and traversal techniques for trees and graphs (In order, preorder, postorder, DFS, BFS), NP-completeness.

## BINARY SEARCH TREES:

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right. To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree.

Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows:-

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.
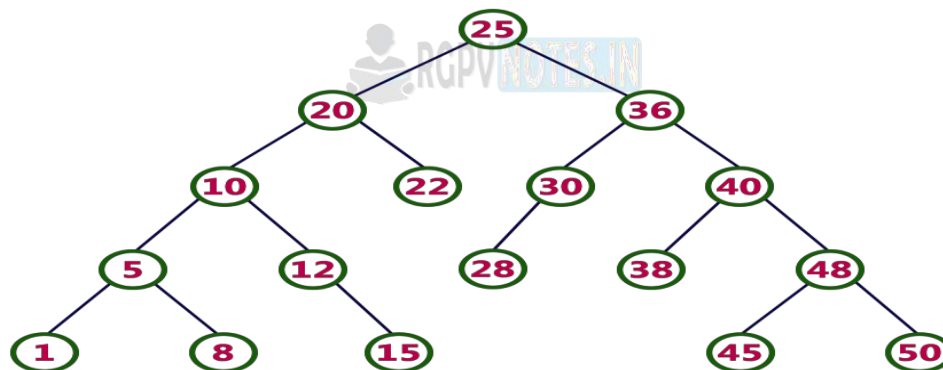


**Figure 5.1 Binary Search Tree**

The following operations are performed on a binary search tree:-1) Search 2) Insertion 3) Deletion

**Search Operation in BST**

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows:-

➢ Step 1: Read the search element from the user
➢ Step 2: Compare, the search element with the value of root node in the tree.
➢ Step 3: If both are matching, then display "Given node found!!!" and terminate the function
➢ Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

- ➢ Step 5: If search element is smaller, then continue the search process in left subtree.
- ➢ Step 6: If search element is larger, then continue the search process in right subtree.
- ➢ Step 7: Repeat the same until we found exact element or we completed with a leaf node
- ➢ Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
- ➢ Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

**Insertion Operation in BST**

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows:-
Step 1: Create a newNode with given value and set its left and right to
NULL. Step 2: Check whether tree is Empty.
Step 3: If the tree is Empty, then set set root to newNode.
Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

**Deletion Operation in BST**
In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases:-

**Case 1: Deleting a leaf node**
**We use the following steps to delete a leaf node from BST:-**
Step 1: Find the node to be deleted using search operation
Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

**Case 2: Deleting a node with one child**
**We use the following steps to delete a node with one child from BST:-**
Step 1: Find the node to be deleted using search operation
Step 2: If it has only one child, then create a link between its parent and child
nodes. Step 3: Delete the node using free function and terminate the
function.

**Case 3: Deleting a node with two children**
**We use the following steps to delete a node with two children from BST:-**
Step 1: Find the node to be deleted using search operation
Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3: Swap both deleting node and node which found in above step.
Step 4: Then, check whether deleting node came to case 1 or case 2 else goto
steps 2 Step 5: If it comes to case 1, then delete using case 1 logic.
Step 6: If it comes to case 2, then delete using case 2 logic.
Step 7: Repeat the same process until node is deleted from the tree.

**Binary Search Tree – Traversal**

There are three types of binary search tree traversals.
1. In - Order Traversal
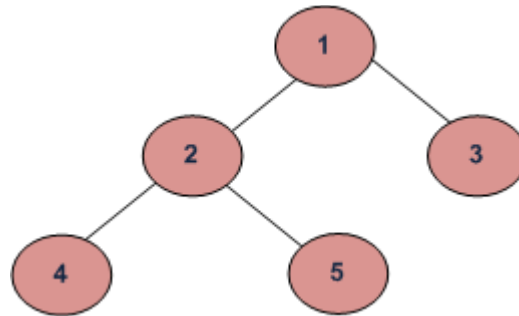2. Pre - Order Traversal
3. Post - Order Traversal



**Figure 5.2 Binary Search Tree**

**1. In - Order Traversal ( leftChild - root - rightChild )**
In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order

traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

**Algorithm Inorder(tree)**

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

**2. Pre - Order Traversal ( root - leftChild - rightChild )**
In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

**Algorithm Preorder(tree)**
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

**3. Post - Order Traversal ( leftChild - rightChild - root )**
In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

**Algorithm Postorder(tree)**
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

## HEIGHT BALANCED TREES:

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

**An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.**

**Balance factor = heightOfLeftSubtree – heightOfRightSubtree**

**AVL Tree Rotations:** Rotation is the process of moving the nodes to either left or right to make tree balanced.

There are **four** rotations and they are classified into **two** types.

**1) Single Rotation**

- Left rotation
- Right rotation

**2) Double Rotation**

- Left-Right rotation
- Right-Left rotation

## INSERTION AND DELETION IN AVL TREE:

So time complexity of AVL insert is O(Logn). The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in O(Log n) time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion.

**Insertion Operation:**
1. Insert the new Node using recursion so while back tracking you will all the parents nodes to check whether they are still balanced or not.
2. Every node has a field called height with default value as 1.
3. When new node is added, its parent's node height get increased by 1.
4. So as mentioned in step 1, every ancestors height will get updated while back tracking to the root.
5. At every node the balance factor will also be checked. **balance factor = (height of left Subtree — height of right Subtree).**
6. If **balance factor =1** means tree is balanced at that node.
7. If **balance factor >1** means tree is not balanced at that node, left height is more that the right height so that means we need rotation. (Either **Left-Left Case or Left-Right Case**).
8. Say the current node which we are checking is X and If new node is less than the X.left then it will be **Left-Left case**, and if new node is greater than the X.left then it will be **Left-Right case**. see the pictures above.

9.  If **balance factor <-1** means tree is not balanced at that node, right height is more that the left height so that means we need rotation. (Either **Right-Right Case or Right– Left Case**)

10. Say the current node which we are checking is X and If new node is less than the X.right then it will be **Right-Right case**, and if new node is greater than the X.right then it will be **Right-Left case**.

**Examples:**

An important example of AVL trees is the behavior on a worst-case add sequence for regular binary trees:

1, 2, 3, 4, 5, 6, 7

All insertions are **right-right** and so rotations are all **single rotate** from the **right**. All but two insertions require re-balancing:
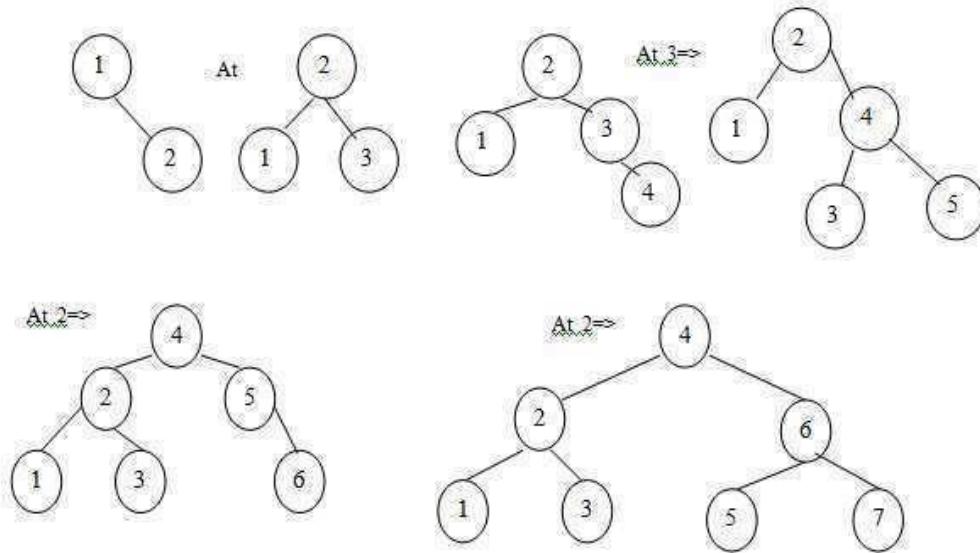


**Figure:5.3 AVL Tree Insertion**

**Deletion in AVL Tree:** If we want to delete any element from the AVL Tree we can delete same as BST deletion. for example Delete 30 in the AVL tree from the figure Figure:5.4 .
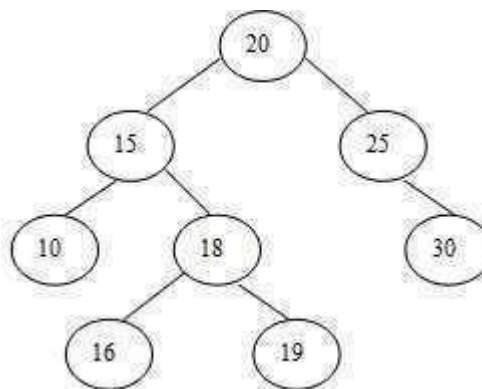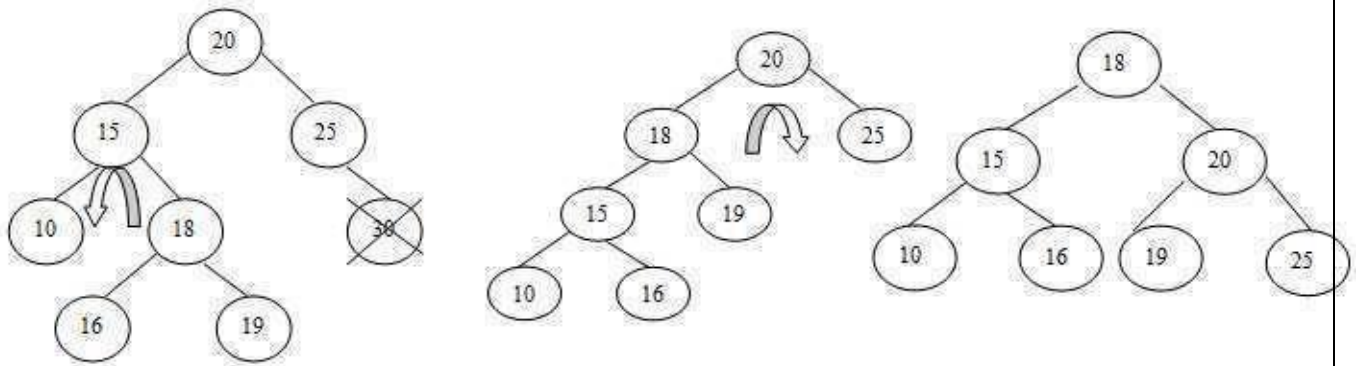


**Figure:5.4 AVL Tree**

**Figure:5.5  AVL Tree Deletion**

A node with value 30 is being deleted in figure 5.5. After deleting 30, we travel up and find the first unbalanced node which is 18. We apply rotation and shift 18 to up for balanced tree .Again we have to move 18 up, so we perform left rotation.

## 2-3 TREES:
A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.
Here are the properties of a 2-3 tree:

- each node has either one value or two value
- a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
- a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
- all leaf nodes are at the same level of the tree

**Insertion algorithm**
Into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:
1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

**The lookup operation**
Recall that the lookup operation needs to determine whether key value k is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:
1.  T is empty: return false
2.  T is a leaf node: return true iff the key value in T is k
And there are 3 recursive cases:
1.  k <= T.leftMax: look up k in T's left subtree
2.  T.leftMax < k <= T.middleMax: look up k in T's middle subtree
3.  T.middleMax < k: look up k in T's right subtree
It should be clear that the time for lookup is proportional to the height of the tree. The height of the tree is O(log N) for N = the number of **nodes** in the tree. You may think this is a problem, since the actual values are only at the leaves. However, the number of leaves is always greater than N/2 (i.e., more than half the nodes in

the tree are leaves). So the time for lookup is also O(log M), where M is the number of key values stored in the tree.

**The delete operation**
Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.
Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:
case 1: n has 3 children
- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.
case 2: n has only 2 children
- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
o     remove the node containing k
o     "steal" one of the sibling's children
o     fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
o     remove the node containing k
o     make n's remaining child a child of n's sibling
o     fix leftMax and middleMax fields of n's sibling as needed
o     remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed
The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).
So the total time is 2 * height-of-tree = O(log N).

**Complexity Analysis**
- keys are stored only at leaves, ordered left-to-right
- non-leaf nodes have 2 or 3 children (never 1)
- non-leaf nodes also have leftMax and middleMax values (as well as pointers to children)
- all leaves are at the same depth
- the height of the tree is O(log N), where N = # nodes in tree
- at least half the nodes are leaves, so the height of the tree is also O(log M) for M = # values stored in tree
- the lookup, insert, and delete methods can all be implemented to run in time O(log N), which is also O(log M)

**B-TREE:**
B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

**Properties of B-Tree**

**1)** All leaves are at same level.

**2)** A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.

**3)** Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

**4)** All nodes (including root) may contain at most 2t – 1 keys.

**5)** Number of children of a node is equal to the number of keys in it plus 1.

**6)** All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.

**7)** B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

**8)** Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(logn).

## BASIC SEARCH AND TRAVERSAL TECHNIQUES FOR TREES AND GRAPHS:

**Search**

Search is similar to search in Binary Search Tree. Let the key to be searched be k. We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

**Traversal**

Tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.
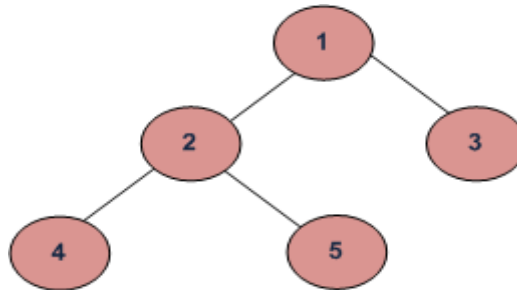


**Figure: 5.6 Binary Tree**

**Depth First Traversals:**
(a) Inorder (Left, Root, Right): 4 2 5 1 3
(b) Preorder (Root, Left, Right): 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

**Breadth First or Level Order Traversal:** 1 2 3 4 5

**Graph Traversal Techniques:**
**Depth First Search (DFS)**
The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search.
Algorithmic Steps
Step 1: Push the root node in the Stack. Step 2: Loop until stack is empty.
Step 3: Peek the node of the stack.
Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.
**Breadth First Search (BFS)**

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search.
Algorithmic Steps
Step 1: Push the root node in the Queue. Step 2: Loop until the queue is empty.
Step 3: Remove the node from the Queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

### NP-COMPLETENESS:
We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.
Can all computational problems be solved by a computer? There are computational problems that cannot be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.
What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in Polynomial time.
NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time) figure 5.1.
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
  1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
  2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).
  A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.
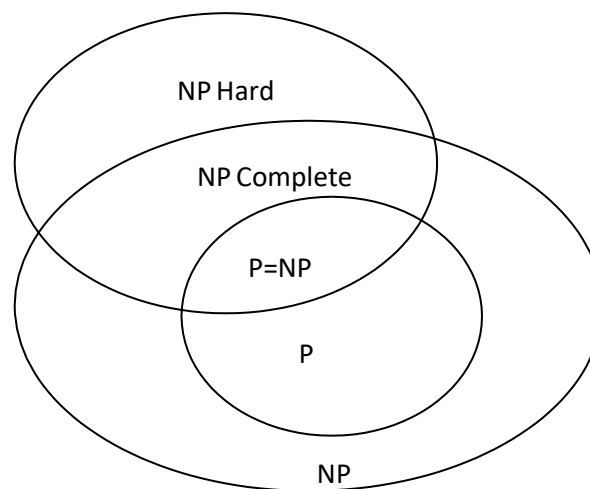


**Figure: 5.7: Relationship between P,NP, NP Complete & NP Hard**