BYJU'S EXAM PREP

Computer Science & IT

Algorithms

SHORT NOTES

Short Notes — ALGORITHMS

- **ALGORITHM**

  Combination of a sequence of Finite sets of steps to solve a specific problem is called an algorithm.

- **Analysis of algorithm:**

  How to check the available algorithm , which is the best?

  1. **time**: time complexity of the algorithm.

     T (A) = C (A) + R (A)

     C(A): compile time of A, depends on the compiler and software

     R(A): Runtime of A, depends on processor and hardware.

  2. **space:** space complexity of algorithm.

- **Criteria for algorithm-**

  1. Finiteness
  2. Definiteness
  3. Design logic
  4. Validation
  5. Analysis
  6. Implementation
  7. Testing & Debugger

- **Example:**

```
main() {
int x,y,z,i,j,k,n;
for(i=1 to n) { ----------------------> order of magnitude =n
for(j=1 to i){ ------------------------> 1,2,3,...n
for(k=1 to 135){ --------------------> order of magnitude =135
x=y+z;
}}}}
```

i=1	i=2	i=3	. . .	i=n
j=1 to 1	j=1 to 2	j=1,2,3	. . .	j=1,2,3,...n
k=135	135, 135	135,135,135	. . .	135,135,135...135

T= 1*135+2*135+3*135+....+n*135

=135(1+2+3+....+n)

=135*n*(n+1)/2

=O(N^2)

- **Asymptotic Notation :**

  To compare two algorithms' rate of growth with respect to time & space we need asymptotic notation.

- **Big-O Analysis of Algorithm:**
- **Example 1:**

  $f(n) = n^2 \log n$; $g(n) = n (\log n)^{10}$, which of the following is true?

  A. $f(n) = (g(n))$, $g(n) \neq 0$ $(f(n))$

  B. $f(n) \neq 0(g(n))$, $g(n) = 0(f(n))$

  C. $f(n) = 0(g(n))$, $g(n) = 0$ $(f(n))$

  D. $f(n) \neq 0(g(n))$, $g(n) \neq 0$ $f(n)$

  Ans. B

- **Big – Omega (Ω) :**

  Ex:

  If $f(n) = n^2 + n + 1$, then $f(n) = \Omega$ ( )

  → $n^2 \geq n^2$

  $n^2 + n \geq n^2$

  $n^2 + n + 1 \geq 1\ n^2$; $\forall\ n \geq 0$

  $n^2 + n + 1 = \Omega\ (n^2)$

  → $n^2 \geq n$

  $n^2 + n \geq n$

  $n^2 + n + 1 \geq 1.\ n$; $\forall n \geq 0$

  $n^2 + n + 1 = \Omega\ (n)$

- **Little ω asymptotic notation**

  Example-1

  $F(n) = n$

  $G(n) = n^2$

  $F(n) = \Omega(G(n))$

  $n >= c.n^2\ \forall n$, $n >= n_0$

- **Theta (θ) :**

  Example 1:

  If $f(n) = n^2 + n + 1$ then $f(n) = \theta$ ( )

  → $n^2 + n + 1 = 0(n^2)$; $\forall\ n \geq 1$ and for $C_2 = 3$

  &

  $n^2 + n + 1 = \Omega\ (n^2)$; $\forall n \geq 0$ and for $C_1 = 1$

  $1\ n^2 \leq n^2 + n + 1 \leq 3\ .\ n^2$; $\forall\ n \geq 1$

  ↑ ↑ ↑

  C_1 C_2 K_1

  $n^2 + n + 1 = \theta\ (n^2)$

- **Properties of Asymptotic:**
  1. **Reflexivity:**

     If f(n) is given then,**f(n) = O(f(n)**

     Example:

     If $f(n) = n^3 \Rightarrow O(n^3)$

     Similarly,

     $f(n) = \Omega(f(n))$

     $f(n) = \Theta(f(n))$
  2. **Symmetry:**

     $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

     Example:

     If $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$
  3. **Transitivity:**

     $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

     Example:

     If $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$

     $\Rightarrow$ n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$
  4. **Transpose Symmetry:**

     $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

     Example:

     If $f(n) = n$ and $g(n) = n^2$ then n is $O(n^2)$ and n^2 is $\Omega(n)$
  5. **Since these properties hold for asymptotic notations, analogies can be drawn between functions f(n) and g(n) and two real numbers a and b.**
     - $g(n) = O(f(n))$ is similar to $a \leq b$
     - $g(n) = \Omega(f(n))$ is similar to $a \geq b$
     - $g(n) = \Theta(f(n))$ is similar to $a = b$
     - $g(n) = o(f(n))$ is similar to $a < b$
     - $g(n) = \omega(f(n))$ is similar to $a > b$
  6. **$\max(f(n), g(n)) = \Theta(f(n) + g(n))$**
  7. **$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$**

- **Difference Between Big oh, Big Omega and Big Theta :**

S.NO.	BIG OH	BIG OMEGA	BIG THETA
1.	It is like <= rate of growth of an algorithm is less than or equal to a specific value	It is like >= rate of growth is greater than or equal to a specified value	It is like == meaning the rate of growth is equal to a specified value
2.	The upper bound of the algorithm is represented by Big O notation. Only the above function is bounded by Big O. asymptotic upper bound is given by Big O notation.	The algorithm lower bound is represented by Omega notation. The asymptotic lower bond is given by Omega notation	The bonding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
4.	Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω takes a small amount of time as compared to Big-O it could possibly take for the algorithm to complete.	Big- Θ is take very short amount of time as compare to Big-O and Big-? it could possibly take for the algorithm to complete.
5.	Mathematically – Big Oh is 0 <=f(n) <= c g(n) for all n>=n0	Mathematically – Big Omega is O<= C g(n) <= f(n) for all n>=n 0	Mathematically – Big Theta is O<=C 2 g(n)<=f(n)<=C 1 g(n) for n>=n 0

- **Recurrence Relation**

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

  A. Substitution Method
  B. Recurrence Tree Method
  C. Master's theorem

- **Recursive Algorithm :**

  Ex : Find time complexity of recursive tan^n. of Fibonacci sequence.

  int fib (in + n)

  {

  B.C. $\begin{cases} \text{it}(n == 0) \\ \text{return} 0; \\ \text{if}(n == 1) \end{cases}$

  return 1

  else

  I.C. {return tib (n – 1) + fib (n – 2);

  }

  A. 0 (n^2)

  B. 0 (2^n)

  C. 0 (n)

  D. 0 (n log n)

  Ans. B

  Here we take n = 5

  - **Note :**

    For small values of n fib (n) = 0 (n^2) & for large values of n fib (n) = (2^n)

    Since our analysis is only for large values of n. So time complexity of $\boxed{fib(n) = 0(2^n)}$

  - **Note :**

    No. of tun^n. calls on i/p size n is Fibonacci sequence =2.fib (n+1) - 1

    e.g. : n = 5, tun^n. call = 15

    = 16 – 1

    = 2 × 8 – 1

    = 2f (6) – 1

  - **Note :**

    No. of addition perform on input size n in fib (n) = fib (n + 1) – 1

    e.g.

    n = 5 ,addition = 7

    = 8 – 1

    = fib (6) – 1

n	0	1	2	3	4	5	6	7
fib (n)	0	1	1	2	3	5	8	13

    Function call = 15 (Total no. of nodes)

    Total addition = 7

- **Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

  Solve the equation by Substitution Method.

  - **Example- 1**

    Long power (long x, long n)

    { if (n==0) return 1;

    if (n==1) return x;

    if ((n % 2) == 0)

    return power (x*x, n/2);

    else

    return power (x*x, n/2) * x; }

    $T(0) = c_1$

    $T(1) = c_2$

    $T(n) = T(n/2) + c_3$

    (Assume n is a power of 2)

    $T(n) = T(n/2) + c_3$ ☐ $T(n/2) = T(n/4) + c_3$

    $=T(n/4) + c_3 + c_3$

    $= T(n/4)2c_3$ ☐ $T(n/4) = T(n/8) + c_3$

    $=T(n/8) + c_3 + 2c_3$

    $= T(n/8)+2c_3$ ☐$T(n/8) = T(n/16) + c_3$

    $= T(n/16) + c_3 + 3c_3$

    $= T(n/32) + c_3 + 4c_3$

    $= T(n/32) + 5c_3$

    =…..

    $= T (n/2k) + kc_3$

    $T (0) = c_1$

    $T (1) = c_2$

    $T(n) = T (n / 2) + c_3$

    $T(n) = T(n/2^k) + kc_3$

    We want to get rid of $T(n/2^k)$. We get to a relation we can solve directly when we reach T(1)

    lg n = k

    $T(n) = T (n/2^{lgn}) + lgnc_3$

    $= T(1) + c_3lgn$

    $= c_2 + c_3lgn$

    $= \Theta (lg\ n)$

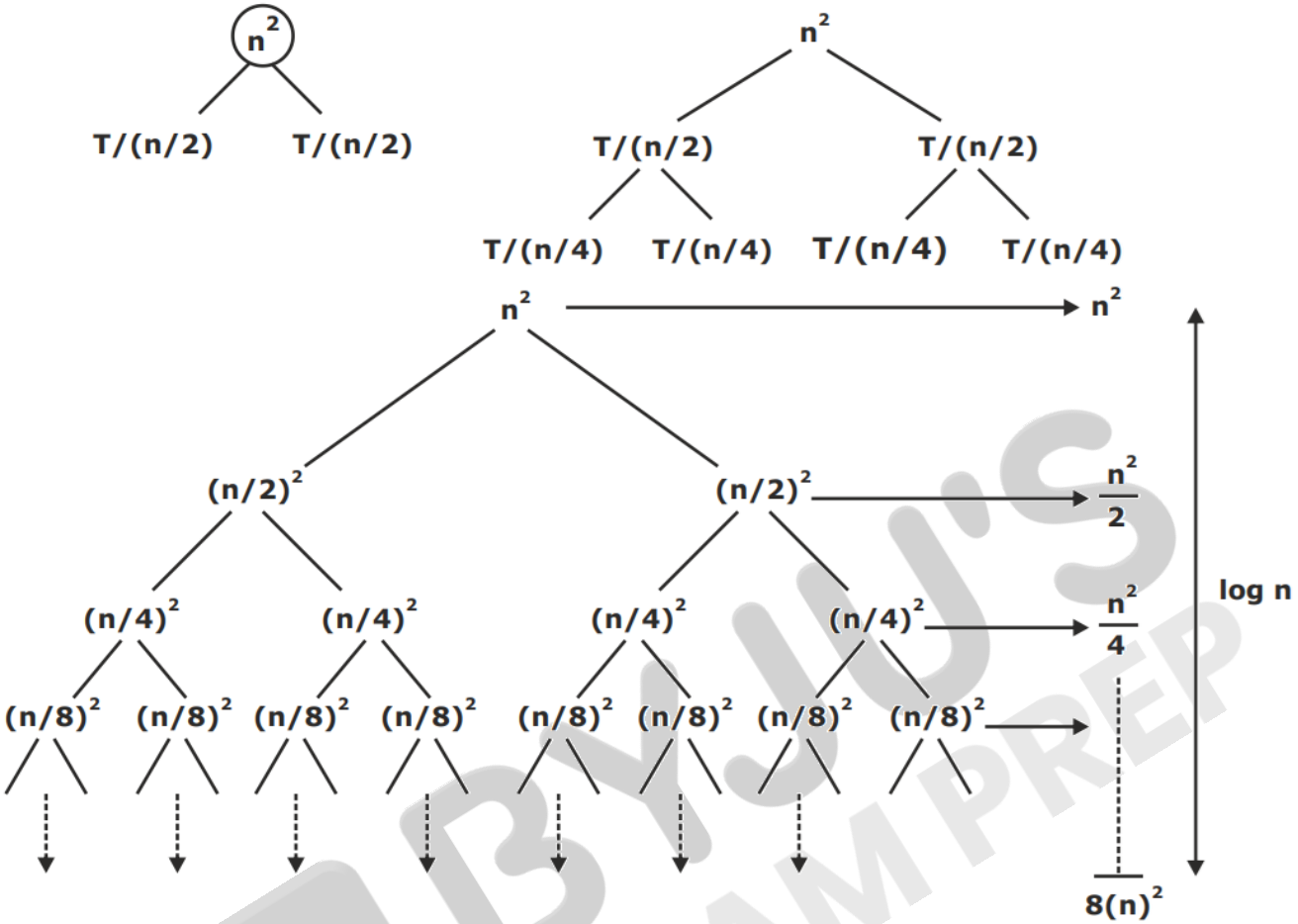
- **Recurrence Tree Method:**

  Example 1-

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

  We have to obtain the asymptotic bound using the recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \text{ logn times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$T(n) = \theta(n^2)$$

- **Master Method**
  - **Case1:** If f (n) = O (n $\log_b$ a — $\mathcal{E}$) for some constant ε >0, then it follows that:

    $$T(n) = \Theta(n^{\log_b a})$$
  - **Case 2:** If it is true, for some constant k ≥ 0 that:

    F (n) = Θ ($n^{\log_b a}\log^k n$) then it follows that : T (n) Θ ($n^{\log_b a}\log^{k+1} n$)
  - **Case 3:** If it is true f(n) = Ω (n $\log_b$ a + $\mathcal{E}$) for some constant ε >0 and it also true that:

    af ($\frac{n}{b}$) ≤ c f (n) for some constant c<1 for large value of n ,

    then : T (n) = Θ((f (n))

- **SPECIAL CASES IN MASTER THEOREM :**

1) $T(n) = 0.5 + \left(\frac{n}{2}\right) + n^2$

Since a = 0.5 (< 1)

So, we can't apply master theorem

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

Here, 'a' can't be a run^n.

So, we can't apply M.T.

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

Negative fun^n. can't allow in M.T. So, we can't apply M.T.

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

exponential fun^n, then put is directly in answer.

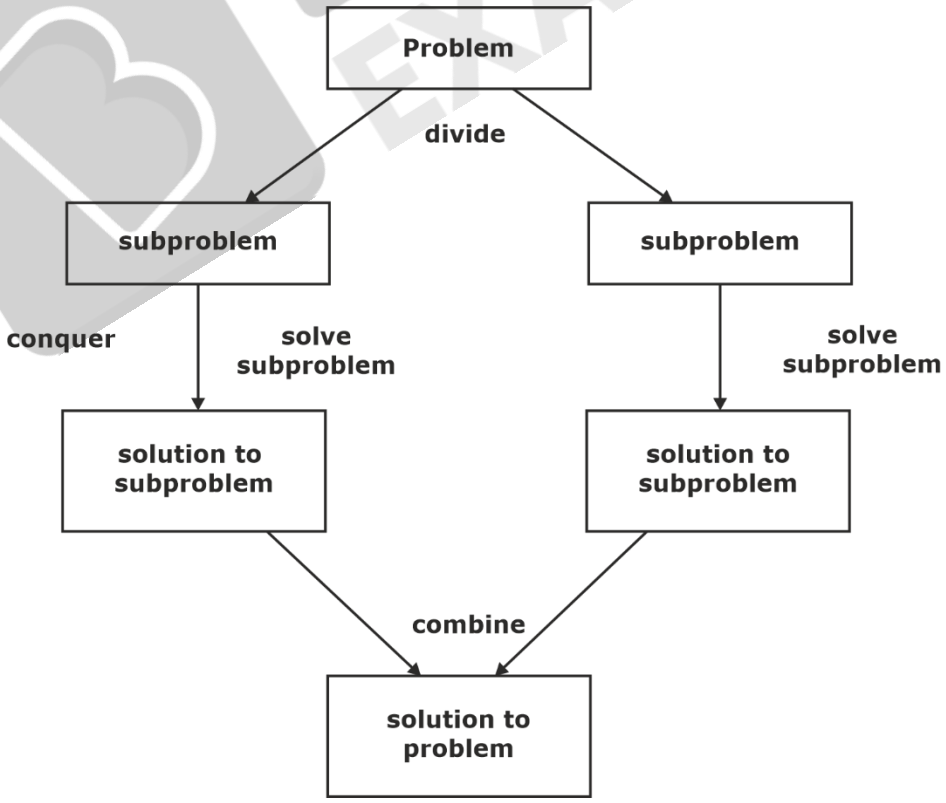
$T(n) = 2\ T\left(\frac{n}{2}\right) + 2^n$

Ans : 0 (2^n)

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

Ans : 0 (n !)

- **Divide and Conquer**



- **Fundamental of Divide & Conquer Strategy:**

  There are two fundamentals of Divide & Conquer Strategy:

  - Relational Formula
  - Stopping Condition

  1. **Relational Formula**: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.
  2. **Stopping Condition**: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called the Stopping Condition.

- **Applications Of Divide and Conquer-**
  - Max Min problem
  - Power of an element
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Selection Procedure
  - Strassen' s matrix multiplication
  - Finding Inversion
- **Max - Min Problem -**

  Analyze the algorithm to find the maximum and minimum element from an array.
- **Maximum Minimum**
  - Find Maximum and minimum element of an array using minimum number of comparisons.
  - In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.
  - In this given problem, the number of elements in an array is $y-x+1$, where **y** is greater than or equal to **x**.
  - $Max-Min(x,y)$ will return the maximum and minimum values of an array $numbers[x...y]$.
- **Algorithm: Max - Min(x, y)**
  - Recurrence Relation: $T(n) = \begin{cases} 0; & \text{if } n = 1 \\ 1; & \text{if } n = 2 \\ T(n/2) + T(n/2) + 1; & \text{if } n > 2 \end{cases}$
  - Time complexity with DAC = O(n)

    (i) Using linear search = O(n)

    (ii) Using Tournament method: Time complexity = O(n)
    - Number of Comparison: When n is power of 2 then

- T(n) = 2 T(n/2) + 2 = (3n/2) -2 else more than (3n/2) – 2.

(iii) Compare in Pairs: Time complexity = O(n)

- Number of comparison

  (a) When n is even = (3n/2) – 2,

  (b) Else n is odd = (3n – 1)/2.

- **Power of an element-**

$$T(n) = \begin{cases} 1 & , \quad \text{if } n = 0 \\ a & , \quad \text{if } n = 1 \\ T(n/2) + c & , \quad \text{otherwise} \end{cases}$$

Time complexity = $O(\log_2 n)$

C Program for calculating Power of an Element

- **Binary Search –**
  - Binary Search is one of the fastest searching algorithms.
  - It is used for finding the location of an element in a linear array.
  - It works on the principle of divide and conquer technique.
  - Binary Search Algorithm can be applied only on Sorted arrays.

    So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
  - To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use a binary search algorithm.
- **Time Complexity Analysis-**

  Binary Search time complexity analysis is done below-
  - In each iteration or in each recursive call, the search gets reduced to half of the array.
  - So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.
- **Binary Search Algorithm Advantages-**

  The advantages of binary search algorithm are-
  - It eliminates half of the list from further searching by using the result of each comparison.
  - It indicates whether the element being searched is before or after the current position in the list.
  - This information is used to narrow the search.
  - For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs a recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.

- The interaction of binary search with memory hierarchy i.e. caching is poor.

(because of its random access nature)

- **Important Note-**

For in-memory searching, if the interval to be searched is small,

  - Linear search may exhibit better performance than binary search.
  - This is because it exhibits better locality of reference.

- **Merge Sort-**
  - Merge sort is a famous sorting algorithm.
  - It uses a divide and conquer paradigm for sorting.
  - It divides the problem into subproblems and solves them individually.
  - It then combines the results of sub problems to get the solution of the original problem.
  - Comparison based sorting.
  - Stable sorting algorithm but outplace.
  - Recurrence Relation: $T(n) = \begin{cases} c \text{ if } n = 1 \\ T(n/2) + T(n/2) + cn \text{ if } n > 1 \end{cases}$
  - Time complexity: (i) Worst case = O(nlogn),
    (ii) Best case = O(nlogn),
    (iii) Average case = O(nlogn).
  - Space complexity: S(n) = 2n + logn + c; Worst case = O(n)
- Merge Sort Algorithm works in the following steps-
  - It divides the given unsorted array into two halves- left and right sub arrays.
  - The sub arrays are divided recursively.
  - This division continues until the size of each sub array becomes 1.
  - After each sub array contains only a single element, each sub array is sorted trivially.
  - Then, the above discussed merge procedure is called.
  - The merge procedure combines these trivially sorted arrays to produce a final sorted array.
- **Properties-**

Some of the important properties of merge sort algorithm are-

  - Merge sort uses a divide and conquer paradigm for sorting.
  - Merge sort is a recursive sorting algorithm.
  - Merge sort is a stable sorting algorithm.
  - Merge sort is not an in-place sorting algorithm.
  - The time complexity of merge sort algorithm is Θ(nlogn).
  - The space complexity of merge sort algorithm is Θ(n).
- **Quick Sort-**
  - Comparison based sorting.
  - 2 to 3 times faster than merge and heap sort.
  - Not a stable sorting algorithm but inplace.

Choosing pivot is the most important factor.

- As Start element T(n) = $O(n^2)$, As End element T(n) = $O(n^2)$
- As Middle element T(n) = $O(n^2)$, As Median element T(n) = $O(n^2)$
- As Median of Median T(n) = O(n logn)

- **Time complexity:**

(i) Best case: each partition splits array into two halves then

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

(ii) Worst case: each partition gives unbalanced splits we get

T(n) T(n – k) + T(k – 1) + cn = $O(n^2)$

(iii) Average case: In the average case, we don't know where the split happens for this reason we take all the possible values of split locations.

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} T(i-1) + (n-i) + n + 1 = O(n\log n)$$

- Space complexity: S(n) = 2n = n + logn = O(n).
- Randomized quicksort choosing pivot from random places to make worst case O(n logn) but for an array with the same element worst case $O(n^2)$.

- **Matrix Multiplication-**

Using DAC: $$T(n) = \begin{cases} O(1), & \text{for } n = 1 \\ 8\,T(n/2) + O(n^2), & \text{for } n > 1 \end{cases}$$

Time complexity = $O(n^3)$

- **Strassen's Matrix Multiplication:**

$$T(n) = \begin{cases} O(1) & \text{, for } n = 1 \\ 7\,T(n/2) + an^2 & \text{, for } n > 1 \end{cases}$$

Time complexity = $O(n^{2.81})$ by Strassen's.

Time complexity = $O(n^{2.37})$ by Coppersmith and Winograd.

- Karatsuba algorithm for fast multiplication of two n-digit numbers required exactly $n^{\log_2 3}$ (when n is a power of 2) using Divide and Conquer.

- **Sorting Techniques**

1. Bubble Sort-

- Bubble sort is the easiest sorting algorithm to implement.
- It is inspired by observing the behaviour of air bubbles over foam.
- It is an in-place sorting algorithm.
- It uses no auxiliary data structures (extra space) while sorting.

Time Complexity Analysis-

Bubble sort uses two loops- inner loop and outer loop.

The inner loop deterministically performs O(n) comparisons.

Worst Case

In worst case, the outer loop runs O(n) times.

Hence, the worst case time complexity of bubble sort is O(n x n) = O(n2).

Best Case-

In best case, the array is already sorted but still to check, bubble sort performs O(n) comparisons.

Hence, the best case time complexity of bubble sort is O(n).

Average Case-

In average case, bubble sort may require (n/2) passes and O(n) comparisons for each pass.

Hence, the average case time complexity of bubble sort is O(n/2 x n) = Θ(n2).

The following table summarizes the time complexities of bubble sort in each case-

Time Complexity

Best Case O(n)

Average Case Θ(n2)

Worst Case O(n2)

From here, it is clear that bubble sort is not at all efficient in terms of time complexity of its algorithm.

Space Complexity Analysis-

Bubble sort uses only a constant amount of extra space for variables like flag, i, n.

Hence, the space complexity of bubble sort is O(1).

It is an in-place sorting algorithm i.e. it modifies elements of the original array to sort the given array.

Properties-

Some of the important properties of bubble sort algorithm are-

Bubble sort is a stable sorting algorithm.

Bubble sort is an in-place sorting algorithm.

The worst case time complexity of the bubble sort algorithm is O(n2).

The space complexity of the bubble sort algorithm is O(1).

Number of swaps in bubble sort = Number of inversion pairs present in the given array.

Bubble sort is beneficial when array elements are less and the array is nearly sorted.

2. Selection Sort-

- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis-

Hence, the space complexity works out to be O(1).

3. **Insertion Sort-**
   - Insertion sort is an in-place sorting algorithm.
   - It uses no auxiliary data structures while sorting.
   - It is inspired from the way in which we sort playing cards.

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

	Time Complexity
Best Case	n
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis-

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).

Important Notes-

- Insertion sort is not a very efficient algorithm when data sets are large.
- This is indicated by the average and worst case complexities.
- Insertion sort is adaptive and the number of comparisons are less if the array is partially sorted.

- **Greedy Method:**
  - "Greedy Method finds out of many options, but you have to choose the best option."
    In this method, we have to find out the best method/option out of many present ways.

In this approach/method we focus on the first stage and decide the output, don't think about the future.

- Greedy Algorithms solve problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

  1. **Greedy Choice Property** 2. **Optimal substructure:**

- **Spanning Tree :**

  Let G (V, E) be an undirected connected graph A subset T (V, E') of G (G, E) is said to be a spanning tree if T is a tree.

  - **Connected graph :**

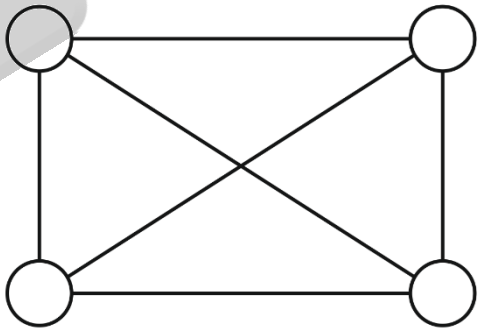
    In a connected graph between every pair of vertices there exists a path.

    e.g.

  - **Complete graph :**

    In a complete graph between every pair of vertices there exists are edge.

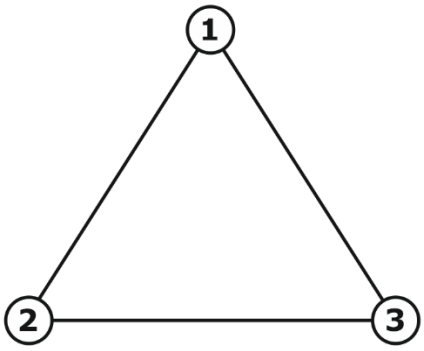
    e.g.



→ Total no of edges in a complete undirected graph with n vertices = (n – 1) + (n – 2) + ... + 0

$= \frac{n(n-1)}{2}$

E.g.



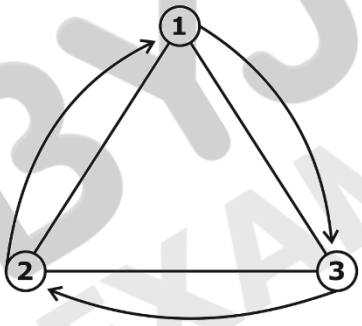
Vertices : 1 2 3

Outbeg : 2 +1 + 0 = 3

→ Total no of edge in a complete directed graph with n vertices $= 2\times\left(\frac{n(n-1)}{2}\right)$

= n (n – 1)

E.g.



⇒ 2 × 3 = 6 edges

- **Theorem :**

  Prove that maximum no. of undirected graph with 'n' vertices = 2 no. of edges

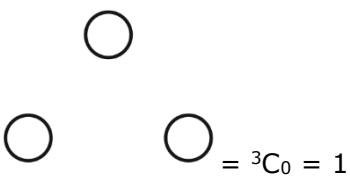
  e.g.

  Take n = 3 vertices.

  ⇒ Maximum no. of edges $\frac{n(n-1)}{2} = \frac{3(3-1)}{2} = 3$ edges

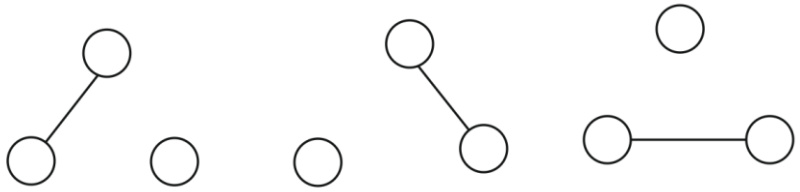
  ↓ undirected graph.

  - **Case (i) :** Graph with 'o' no of edges.



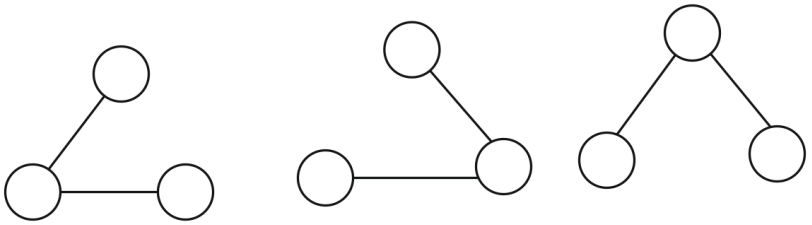
= $^3C_0$ = 1

- **Case (ii) :** Graph with '1' no. of edges.



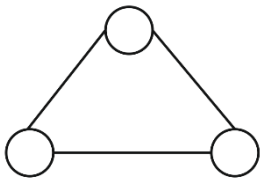
$= {}^3C_1 = 3$

- **Case (iii) :** Graph with '2' no. of edges.



$= {}^3C_2 = 3$

- **Case (iv) :** Graph with '3' no. of edges.



$= {}^3C_3 = 1$

$\therefore$ Total no. of graph = 1 + 3 + 3 + 1 = 8

- **Proof :**

n = vertices

edges $= \frac{n(n-1)}{2}$

(i) Graph with ' 0 ' edge $= \frac{n(n-1)}{2_{C_0}}$

(ii) Graph with ' 1 ' edge $= \frac{n(n-1)}{2_{C_1}}$

Graph with $\frac{n(n-1)}{1}$ edges = $\frac{n(n-1)}{2}$ C $\frac{n(n-1)}{2}$

$= \frac{n(n-1)}{2_{C_0}} + \frac{n(n-1)}{2_{C_1}} + \ldots\cdots + \frac{n(n-1)}{2_C}$.

Total no of graphs

$$= 2^{\frac{n(n-1)}{2}} \left[\because {}^nC_0 + {}^nC_1 + \ldots + {}^nC_n = 2^n\right]$$

$\therefore$ Total no. of graph = 2 No. of edges.

- **Properties of spanning tree :**
- **Difference between Prim's and Kruskal's algorithm-**

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E + log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected components as well as it works only on connected graphs.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

- **SINGLE SOURCE SHORTEST PATHS**

  In a shortest- paths problem, we are given a weighted, directed graph G = (V, E), with weight function w: E → R mapping edges to real-valued weights. The weight of path p = (v0,v1,..... vk) is the total of the weights of its constituent edges:

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

- **Dijkstra Algorithm-**
  - **Single Source Shortest Path (Dijkstra's Algorithm)**
    - Time complexity : O(V + E)log V) using binary min heap.
    - Drawback of Dijikstra'sAlgorithm : It will not give the shortest path for some vertices if the graph contain-negative weight cycle.
    - Time complexity of Dijkstra's and Prim's algorithm using various data structures.

      (i) Using Binary Heap : O(V + E)log V)

      (ii) Fibonacci heap : O((V log V + E))

      (iii) Binomial Heap : O((V + E)log V)

      (iv) Array : $O(V^2 + E)$

- **Bellman Ford Algorithm-**
  - It finds the shortest path from source to every vertex. If the graph doesn't contain a negative weight cycle.
  - If a graph contains a negative weight cycle, it doesn't compute the shortest path from source to all other vertices but it will report saying "negative weight cycle exists".

    Time complexity = O(VE) when dense graph E = V2 and for sparse graph E = V.

- **Difference Between Bellman ford and Dijkstra's Algorithm.**

BELLMAN FORD'S ALGORITHM	DIJKSTRA'S ALGORITHM
Bellman Ford's Algorithm works when there is a negative weight edge, it also detects the negative weight cycle.	Dijkstra's Algorithm doesn't work when there is a negative weight edge.
The result contains the vertices which contain the information about the other vertices they are connected to.	The result contains the vertices containing whole information about the network, not only the vertices they are connected to.
It can easily be implemented in a distributed way.	It can not be implemented easily in a distributed way.
It is more time consuming than Dijkstra's algorithm. Its time complexity is O(VE).	It is less time consuming. The time complexity is O(E logV).
Dynamic Programming approach is taken to implement the algorithm.	Greedy approach is taken to implement the algorithm.

- **Job sequencing problem :**
  - **Shortcut :**

    If there are n jobs, possible subsets are 2^n. Objective of J.S. is to find the subset which generates maximum profit.

    Arrange all jobs in decreasing order of process their profit & then process in that order within its deadline.

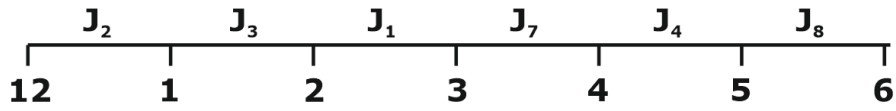
$$J_1 \geq J_4 \geq J_3 \geq J_2$$

$$\langle J_1 . J_4 \rangle \Rightarrow 100 + 27 = 127$$

Ex:

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8
Deadline	6	5	6	6	3	4	4	5
Profit	10	8	9	12	3	6	11	13

$$J_8 \geq J_4 \geq J_7 \geq J_1 \geq J_3 \geq J_2 \geq J_6 \geq J_5$$



Max. Profit = 63

- **Analysis of Job sequencing :**
  To implement J.S. we use a priority queue where priorities are assigned to the profits of a job. So, the time complexity is O (n log n).

- **Knapsack Problem :**
  Shortcut :
  Arrange all unit weight profit into decreasing order & then process objects in that order without exceeding knapsack size.
  → Since we are using priority queue, where priority is assigned to unit weight profit. So, time complexity = 0 (n log n)

- **Optimal Merge Problem :**
  Shortcut : Arrange all files in the increasing order of their record length and in **each iteration** select two files which are having least no. of records and merge them into a single file. Continue the process until all files are merged.

- **DYNAMIC PROGRAMMING**

	Tabulation	**Memorization**
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous	Slow due to lot of recursive calls and return statements

	Tabulation	Memorization
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memorized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memorized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memorized version. The table is filled on demand.

- **APPLICATIONS OF DYNAMIC PROGRAMMING**
  - 0/1 knapsack problem-

$$0/1KS(M,N)=\begin{cases}0; & \text{if } M=0 \text{ or } N=0\\ 0/1KS(M,N-1); & \text{if } w[n]>M\\ \max\begin{Bmatrix}0/1KS(M-W[n],N-1)+P[n]\\ 0/1KS(M,N-1)\end{Bmatrix}; & \text{otherwise}\end{cases}$$

Time complexity = O(MN)

  - Longest common subsequence (LCS)-

A subsequence of a given sequence is just the given sequence with some elements left out. Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

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

Longest Common Subsequence

Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

- **Recurrence Relation:**

$$LCS(i,j)=\begin{cases}0; & \text{if } i=0 \text{ (or) } j=0\\ 1+LCS(i-1,j-1); & \text{if } x[i-1]=y[j-1]\\ \max\left[LCS(i-1,j),LCS(i,j-1)\right]; & \text{if } x[i]\neq y[j]\end{cases}$$

Time complexity : by Brute force = O(2m), by Dynamic programming = O(m × n).

Space complexity = O(mn);

- **Floyd- Warshall's :** All pair Shortest path problem-

For finding shortest paths between all paris of vertices in a weighted graph with positive or negative edge weights (but with no negative cycles).

→ $A^k(I, j)$ = the min cost required to go from i to j by considering all intermediate vertices not greater than k.

$A^0(I, j) = C(I, j)$

$A^k(I, j) = \min\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}$

Time complexity – $O(n^3)$ with help = $O(n^2 \log n)$.

→ Warshall's algorithm will not for negative edge cycle.

- **Floyd Warshall Algorithm Complexity**
  - Time Complexity

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

    The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.
- Floyd Warshall Algorithm Applications
  - To find the shortest path is a directed graph
  - To find the transitive closure of directed graphs
  - To find the Inversion of real matrices
  - For testing whether an undirected graph is bipartite
- Sum of Subset problem

Find if there is a subject of the given set, sum of whose elements is equal to given sum.

- **Recurrence Relation:**

$$SoS(M,N,S)=\begin{cases}\text{return}(S); & M=0\\ \text{return}(-1); & N=0\\ SoS(M,N-1,S); & \text{if } w[N]>M\\ \text{Min}\begin{Bmatrix}SoS(M-w[N],N-1,S\cup W[N])\\ SoS(M,N-1,S)\end{Bmatrix}; & \text{otherwise}\end{cases}$$

Time complexity by Brute force = O(MN)

- **Matrix Chain Multiplication-**

  It is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row [always].

  Matrix chain Multiplication

  Given a sequence of matrices, find the most efficient order to multiply these matirices together in order to minimise the number of multiplications.

$$MCM = \begin{cases} 0 \quad \text{if } i = j \\ \min \begin{cases} mcm(i,K) + mcm(K+1,j) + Pi \times Pj \times Pk \\ \text{where } i \le K < j \text{ or} \\ i \le K \le j-1 \end{cases} \end{cases}$$

  Number of ways to multiply matrix:

$$T(n) = \sum_{i=1}^{n-1} T(i)\,T(n-i)$$

  - Time complexity : Without dynamic programming = $O(n^n)$, with dynamic programming = $O(n^3)$.
  - Space complexity: without dynamic programming = $O(n)$, with dynamic programming = $O(n^2)$.

- **Travelling Salesman Problem-**

$$TSP(A,R) = \begin{cases} C(A,S) \quad \text{if } R = \phi \\ \min\left\{\left(C[A,K] + TSP(K,R-K)\right) \forall K \in R\right\} \end{cases}$$

- **HASHING**
  - **Types of Hash Functions-**

    There are various types of hash functions available such as-

    1. Mid Square Hash Function
    2. Division Hash Function
    3. Folding Hash Function etc

    It depends on the user which hash function he wants to use.
  - **Properties of Hash Function-**

    The properties of a good hash function are-

    - It is efficiently computable.
    - It minimizes the number of collisions.
    - It distributes the keys uniformly over the table.

- **Clustering**
  - Primary clustering:

    The tend is for long sequences of preoccupied positions still become longer, primarily at one place.
  - Secondary clustering:

    The tend is for long sequence of preoccupied position still become longer primarily at different places.

- Collision Resolution Techniques-

  Collision Resolution Techniques are the techniques used for resolving or handling the collision.

  Collision resolution techniques are classified as-

  1. Separate Chaining
  2. Open Addressing

- **Separate Chaining-**

  To handle the collision,

  - This technique creates a linked list to the slot for which collision occurs.
  - The new key is then inserted in the linked list.
  - These linked lists to the slots appear like chains.
  - That is why, this technique is called as **separate chaining**.

  Collision resolution by chaining combines linkes representation with a hash table. When two or more records have the same location, these records are constituted into a singly-linked list called a chain.

- **Time Complexity-**

  For Searching-

  - In the worst case, all the keys might map to the same bucket of the hash table.
  - In such a case, all the keys will be present in a single linked list.
  - Sequential search will have to be performed on the linked list to perform the search.
  - So, time taken for searching in the worst case is O(n).

  For Deletion-

  - In the worst case, the key might have to be searched first and then deleted.
  - In the worst case, time taken for searching is O(n).
  - So, time taken for deletion in the worst case is O(n).

- **Load Factor (α)-**

  Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

- **Open addressing vs closed addressing:**
  - Disadvantages of open addressing:
    1. Collided records required more probes.
    2. Deletion is not possible.
    3. Overflow problem
  - Advantages of chaining:
    1. Collided records required less probes.
    2. Deletion possible
    3. No overflow problem

To avoid this problem we will use double hashing :-

T.C.

insertion	searching	deletion
B.C.→ O(1)	B.C. → O(1)	B.C. → O(1)
W.C. → O(m)	W.C → O(m)	W.C → O(m)

- **Double hashing:-**

  m = 10(0, …a)

  $H.F_1$ (key) = key mod m;

  $H.F_2$ (key) = 1+(key mod m-2);

  $D11(key, 1) = (H_1F_1(key) + i * H.F_2(key)) \bmod m$

- **Secondary clustering:-**

  If the two keys are mapped onto the same starting location in the hash table then they both follow the same path unnecessarily in the quadratic manner. because of this search time complexity will increase.

T.C.

searching	insertion	deletion
B.C.→ O(1)	B.C. → O(1)	B.C. → O(1)
W.C. → O(m)	W.C → O(m)	W.C → O(m)

- **Conclusion :-** if n ≤ m: then by using perfect hashing we can achieve worst call search time complexity as O(1). (99%)

  NOTE:

  1) The Expected no. of probe's in an unsuccessful search of open addressing technique is $\frac{1}{1-\alpha}$

     where α is load factor $\alpha = \frac{n}{m}$

  2) the Expected no. of probe's in an successful search of open addressing technique is

     $\frac{1}{\alpha}\log\frac{1}{1-\alpha}$ where 'α' Load factor $\boxed{\alpha = \frac{n}{m}}$

****