Algorithm Short Notes

1 ASYMPTOTIC NOTATION

1.1 Introduction of Course

Algorithms

Analysis of Algorithms

- Algorithm concept and life cycle
- Needs for analysis
- Methodology of analysis
- Types of analysis
- Asymptotic notation
- Framework for analysis of recursive and non recursive Algorithms
- Analysing loop
- Space complexities

Design Strategies

- Divide and Conquer
- Greedy method
- Dynamic programming

Graph Algorithms

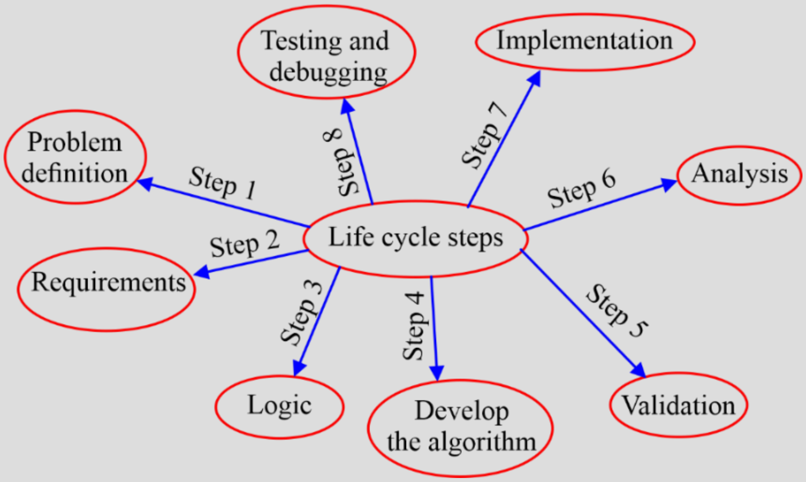
- Graph Algorithms
- Heap Algorithms
- Hashing
- Sets
- Sorting Algorithms

1.2 Algorithm Concept and Life Cycle Steps

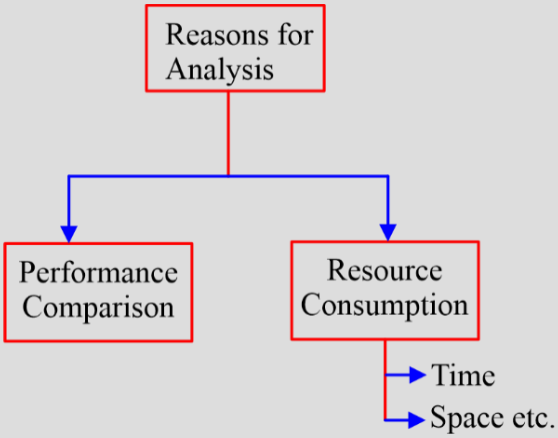
1.2.1 Algorithm

- An Algorithm consists finite number of steps to solve any problem.
- Every step involves some operations and each operation must be definite and effective.

1.2.2 Life Cycle Steps



1.3 Needs of Analysis



In performance comparison comparing different algorithms for optimal solution.

1.3.1 Time Complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the input size.

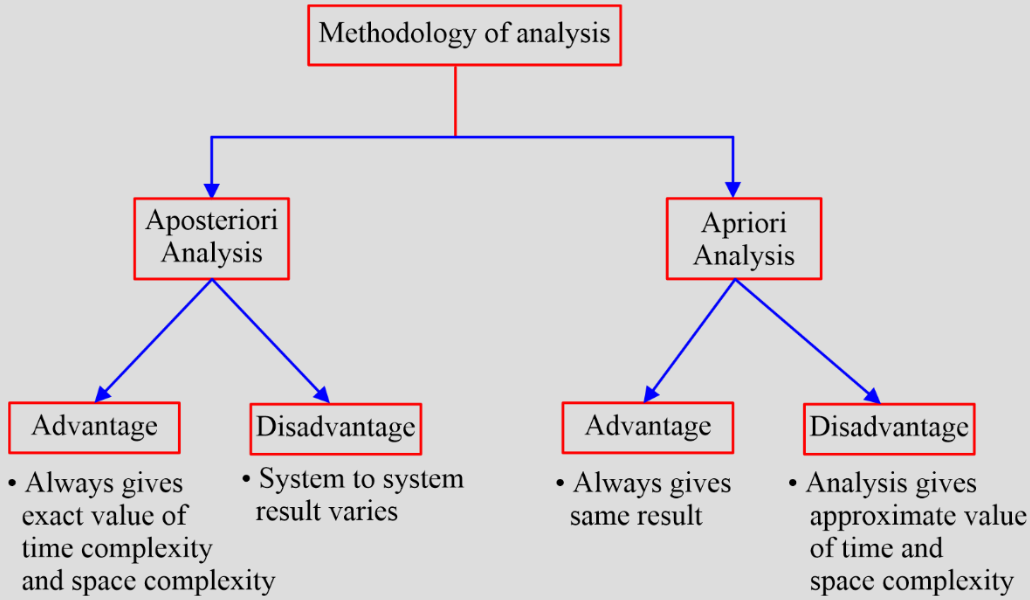
1.3.2 Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of input size.

Note:

To find the time complexity of an algorithm, find the loops and also consider larger loops.
Space complexity is dependent on two things input size and some extra space (stack space link, space list etc).

1.4 Methodology of Analysis



1.5 Types of Analysis

Worst Case

The input class for which the algorithm does maximum work and hence, take maximum time.

Best Case

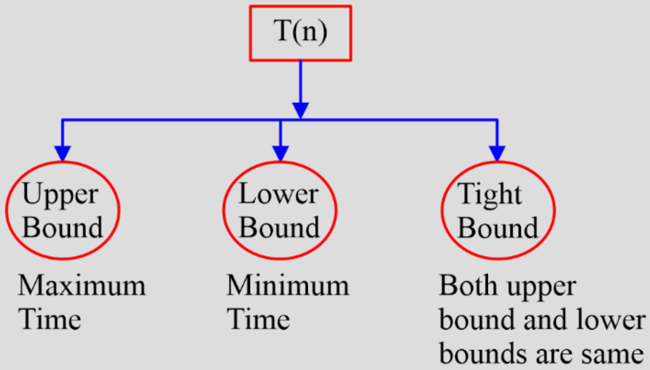
The input class for which the algorithm does minimum work hence, take minimum time.

Average Case

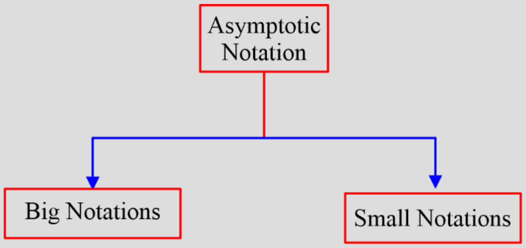
Average case can be calculated form best case to worst case.

1.6 Asymptotic Notations

Suppose, T(n) be a function of time for any algorithm.



1.7 Types of Asymptotic Notations



- Big oh (O): Upper bound
- Big omega (Ω): Lower bound
- Theta (θ): Tight bound

- Small oh (o): Proper upper bound
- Small omega (ω): Proper lower bound

1.7.1 Big O – Notation

Two Functions $f(n), g(n)$

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

When the growth of $g(n)$ is same or higher than $f(n)$ like $a \leq b$

Example:

$$f(n) = 3n + 10, g(n) = n^2 + 2n + 5$$

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

1.7.2 Ω - Notation

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

$\therefore f(n) \geq C \cdot g(n) \qquad (a \geq b)$

Example: $3^n = \Omega(2^n)$

1.7.3 θ - Notation

If $f(n) \leq g(n)$

And

$f(n) \geq g(n)$

$f(n) = g(n)$

$\therefore \quad f(n) = \theta(g(n))$

Example:

$f(n) = 2n^2, g(n) = n+10$

$f(n) > g(n)$ here

so, $f(n) = \Omega(g(n))$ or $g(n)=O(f(n))$

1.7.4. Properties with respect to asymptotic notations

	Reflexive	Symmetric	Transitive	Transpose symmetric
Big oh (O)	✓	×	✓	✓
Big omega (Ω)	✓	×	✓	✓
Theta (θ)	✓	✓	✓	×
Small oh (o)	×	×	✓	✓
Small omega (ω)	×	×	✓	✓

Example 1. Consider the following function

$$f(n) = \sum_{p=1}^{n} p^3 = q$$

Which of the following is/are true for 'q'

(a) $\theta(n^4)$ (b) $\theta(n^5)$ (c) $O(n^5)$ (d) $\Omega(n^3)$

Solution: (a, c, d)

$$f(n) = \sum_{P=1}^{n} P^3$$

$$= 1^3 + 2^3 + 3^3 + 4^3 \ldots\ldots\ldots\ldots\ldots\ldots + n^3$$

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

$$= O(n^4) \text{ or } \Omega\,(n^4)$$

$$= \theta\,(n^4)$$

Example 2. Consider the following functions:

$$f(n) = \sum_{P=1}^{n} P^{1/2} = q$$

Find the value of q in terms of asymptotic notation.

Solution: $f(n) = \sum_{P=1}^{n} P^{1/2}$

$$= 1 + (2)^{1/2} + (3)^{1/2} + \ldots\ldots.$$

$$= \frac{2}{3}\left[n^{3/2} - 1\right]$$

$$= \frac{2}{3}n^{3/2} - \frac{2}{3}$$

$$= O(n^{1.5})$$

$$= O(n\sqrt{n})$$

Example 3. Arrange the following functions in increasing order.

$$f_1 = n\log n,\ f_2 = \sqrt{n},\ f_3 = 2^n,\ f_4 = 3^n,\ f_5 = n!,\ f_6 = n^n,\ f_7 = \sqrt{\log n},\ f_8 = 100n\log n$$

$$\rightarrow f_7 < f_2 < fl = f_8 < f_3 < f_4 < f_5 < f_6$$

Example 4. Arrange the following functions in increasing order.

$$f_1 = 10,\ f_2 = \sqrt{n},\ f_3 = \log\log n,\ f_4 = (\log n)^2,\ f_5 = n^2$$

$$f_6 = n\log n,\ f_7 = n!,\ f_8 = 2^n,\ f_9 = n^n,\ f_{10} = n^2 \log n$$

$$\rightarrow f_1 < f_3 < f_4 < f_2 < f_6 < f_5 < f_{10} < f_8 < f_7 < f_9$$

Example 5. Arrange the following functions in increasing order.

$f_1 = \log\log n$ $f_9 = n\log\log n$

$f_2 = \log n$ $f_{10} = n^2 \log n$

$f_3 = (\log n)^2$ $f_{11} = n^3$

$f_4 = \sqrt{\log n}$ $f_{12} = 2^n$

$f_5 = n^{1/10}$ $f_{13} = e^n$

$f_6 = n$ $f_{14} = n!$

$f_7 = n^2$ $f_{15} = n^n$

$f_8 = n\log n$ $f_{16} = n^{3/2}$

$$f_1 < f_4 < f_2 < f_3 < f_5 < f_6 < f_9 < f_8 < f_{16} < f_7 < f_{10} < f_{11} < f_{12} < f_{13} < f_{14} < f_{15}$$

$$a^{\log_b c} \Leftrightarrow c^{\log_b a}$$

$$\therefore 2^{\log_2 n} \Leftrightarrow n^{\log_2 2} = n$$

Example 6. Arrange the following functions in increasing order.

$$f_1 = n!,\ f_2 = n^n$$

$$f_1 = n \times (n-1)(n-2) \times \ldots \times 3 \times 2 \times 1$$

$$f_2 = n \times n \times n \times n \times \ldots \times n \times n \times n$$

$$f_2 > f_1$$

$$\therefore f_1 = O(f_2)$$

$$\boxed{2^n < 3^n < 4^n < n! < n^n}$$

Question.

Which of following is TRUE?

(1) $2^{\log_2 n} = O(n^2)$ TRUE

(2) $n^2 \cdot 2^{3\log_2 n} = O(n^5)$ TRUE

(3) $2^n = O(2^{2n})$ TRUE

(4) $\log n = O(\log\log n)$ FALSE

(5) $\log\log n = O(n\log n)$ TRUE

Solution:

(1) $2^{\log_2 n} = O(n^2)$

$$= n^{\log_2 2}$$

$$= n$$

$$= n = O(n^2)$$

(2) $n^2 \cdot 2^{3\log_2 n} = O(n^5)$

$$= n^2 \cdot n^{3\log_2 2}$$

$$= n^2 \cdot n^3$$

$$= n^5$$

$$= n^5 = O(n^5)$$

(3) $2^n = 2^{2n}$

$$2^n = 2^n . 2^n$$

$$2^n \leq 2^{2n}$$

$2^n = O(2^{2n})$ True

(4) $\log n > \log\log n$

$\log n \neq O(\log n)$

False

(5) $\log\,\log n \leq n\log n$

$\log\log n = O(n\log n)$

True

1.8. Analysis of an Algorithm

Algorithms

Without Loop | **Interactive Algorithm** | **Recursive Algorithm**

1.8.1 Without loop

Example:

```
int fun (in + n)
{
    return n*(n+1)/2;
}
```

Solution.

Here 1 multiply, 1 division, 1 addition

$\therefore$ O (1) [no loops, no recursion]

1.8.2. Iterative Algorithm Analysis

Example 1:

for (i =1; i $\leq n$; i=i*2)

printf("Sushil")

Solution.

i=1, 2, 2^2, 2^3 ..., 2^k

$\rightarrow$

$2k \leq n$

$k \log 2 \leq \log n$

$k \leq \dfrac{\log n}{\log 2}$

$\therefore k \leq \log_2 n$

$k = \lfloor \log_2 n \rfloor$

So, this will execute $|\lfloor \log_2 n \rfloor| + 1$ time and Complexity O ($\log_2 n$)

Example 2:

For (i=1; i ≤ *n*; i=i*3)

printf("Aaveg");

Solution.

So, this will execute $|\lfloor \log_3 n \rfloor| + 1$ time and complexity O ($\log_3 n$)

- i = 1→2→4→8→16→...→n

  i = n→n/2→n/4→n/8→...1

Example 3:

```
for (i = 1; i ≤ n; i++)
    {
        for (j=1; j ≤ 10; j++)
            {
                printf("Dhananjay");
            }
        }
```

Solution.

This will execute 10·*n* times and complexity O(*n*)

Example 4:

```
for (i = 1; i <= n; i = i*3)
    for (j = 1; j ≤ n; j++)
        printf("Prapti");
```

Solution.

Total $n\left(|\lfloor \log_3 n \rfloor| + 1\right)$ time execute and Complexity = O ($n \log_3 n$)

1.8.3. Recursive Algorithm Analysis

Example 1:

```
void fun (in + n)        T(n)
{
      if (n > 0)          1 compare; C1 time
      {
            if ("% d", n);  <----   C2 time
            fun (n - 1);    <----   T(n-1)
      }
}
```

Let T(*n*) be the Complexity time taken by algo for n size i/p

Solution.

$$\left.\begin{array}{l} T(n) = C_1+C_2+T(n-1) \\ T(n) = T(n-1) + C \end{array}\right\} \quad n > 0$$

$T(0) = C$ ← Constant

$$\boxed{\begin{array}{ll} T(n) = C; & n = 0 \\ T(n) = T(n-1) + C; & n > 0 \end{array}}$$

Example 2:

```
      void fun (in + n)        T (n)
{
       if (n > 0)  <---- C1 time
      {
            for (i = 1; i <= n; i + 1)  <----  n time
             printf("Hello");
            fun (n - 1);  <----  T (n - 1)
       }
}
```

Solution.

$$\left.\begin{array}{l} T(n) = C_1 + n - 1 + T(n-1) \\ \quad\;\; = T(n-1) + n \end{array}\right\} \quad n > 0$$

$$T(0) = C \qquad n = 0$$

Example 3:

```
void fun (in + n)              T(n)
{
 if (n > 0)  <----  C1
  {
      for (i = 1; i < = n; i = i*2)  <----  ⌊log2 n⌋
```

```
            printf("Divyajyoti");
        fun (n - 1);   ←   T (n - 1)
      }
  }
```

Solution. $T(n) = T(n-1) + O(\log_2 n)$; $n > 0$

or

$T(n) = T(n-1) + \log_2 n$

$T(0) = C$

$T(0) = O(1)$ — or $n = 0$

1.9 Solving Recurrence Relation

1.9.1 Substitution Method

Example: (1)

$$\mathbf{T(n) = T(n-1) + C}$$
$$\mathbf{T(1) = C}$$

n size on problem $n - 1$ size x_1 convert them

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

$[T(n-2) + C] + C$

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

$= T(n-3) + 3C$

$T(n) = T(n-k) + kC$

$\therefore n - k = 1$

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

$= C + (n-1)C$

$T(n) = O(n)$

Example (2)

$$\mathbf{T(n) = T(n-1) + C \cdot n}$$
$$\mathbf{T(1) = C}$$

Solution.

$\therefore T(n) = T(n-1) + C \cdot n$

$= [T(n-2) + C \cdot (n-1)] + C \cdot n$

$= [T(n-3) + C \cdot (n-2)] + C(n-1) + C \cdot n$

$= T(n-3) + (n-2) \cdot C + (n-1) \cdot C + n \cdot C$

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

$\therefore n - k = 1$

$T(n) = T(1) + T(2) + C(3) + C(4) + \ldots + C(n-1) + C(n)$

$= C + C(2) + (3)C + 4(C) + \ldots + (n-1)C + (n) \cdot C$

$= C[1 + 2 + 3 + \ldots + n]$

$$= C \cdot n \frac{(n+1)}{2}$$

$$= O\ (n^2)$$

Example (3)

T (*n*) = T (*n*/2) + C

T (1) = 1

Solution.

$$T\ (n) = T\ (n/2) + C$$

$$= [T\ (n/2^2) + C] + C$$

$$= T\ (n/4) + 2C$$

$$= T\ (n/2^3) + 3C$$

$$T\ (n) = T\ (n/2^k) + kC$$

$$= (n/2^k) = 2$$

$$T\ (n) = T\ (2) + (\log_2 n - 1)\ C$$

$$= 1 + (\log_2 n - 1)\ C$$

$$= O\ (\log n)$$

Example (4)

T (1) = 1

T (n)= 2T (n/2) + C

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

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

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

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

$$= 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} C + 2^{k-2} C + \ldots + 2^1 \cdot C + C$$

$$\frac{n}{2^k} = 1 \quad \therefore n = 2^k$$

$\rightarrow$ T (n) $= n\text{T}(1) + 2^{k-1} \cdot C + 2^{k-2} \cdot C + \ldots + 2C + C$

$= 2^k + 2^{k-1} \cdot C + 2^{k-2} + \ldots + 2C + C$

$= 2^k + C\,(2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0)$

$= 2^k + C\,\dfrac{(2^k - 1)}{2 - 1}$

$= 2^k + C\,(2^k - 1)$

$= 2^k + 2^k \cdot C - C$

$= n \cdot C$

$= O(n)$

1.9.2 Master's Method

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^k (\log n)^p\right)$$

$a \geq 1$, $b > 1$, $k \geq 0$, p = real number

If $a > b^k$ or $\log_b a > k$

$T(n) = \Theta\left(n^{\log_b a}\right)$

Question 1. $\mathbf{T(n) = 2T\left(\frac{n}{2}\right) + (n)^0 \log n}$

Solution. $a = 2$, $b = 2$, $k = 0$

$a > b^k$; $2 > 2^0$; $2 > 1$

$T(n) = \Theta(n)$

Question 2. $\mathbf{T(n) = 2T\left(\frac{n}{2}\right)} + n$

Solution. $a = 2$, $b = 2$, $k = 1$, $p = 0$

$T(n) = \Theta(n \cdot \log n)$

Question 3. $\mathbf{T(n) = 2T\left(\frac{n}{2}\right) + n \log n}$

Solution. $a = 2$, $b = 2$, $k = 1$, $p = 1$

$\therefore T(n) = \Theta(n (\log n)^2)$

(b) If $p < 0$ then $T(n)$

$T(n) = O(n^k)$

Question 4. $\mathbf{T}(\boldsymbol{n}) = \mathbf{T}\left(\frac{\boldsymbol{n}}{\mathbf{2}}\right) + \boldsymbol{C}$

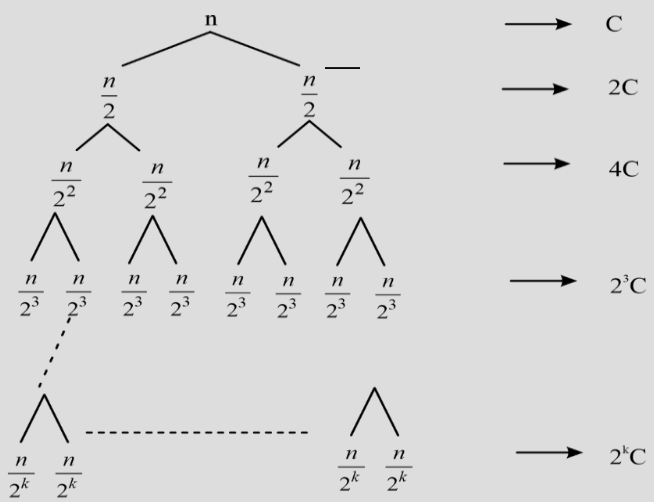
Solution.

$$T(n) = \Theta\,(n^2 \log n)$$

1.9.3. Recursive Tree

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

$$T(1) = C$$



$$\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow k = \log_2 n$$

Total Work done $= C + 2C + 2^2C + 2^3C + \ldots + 2^kC$

$= C\,(1 + 2 + 2^2 + \ldots + 2^k)$

$= c\left(\frac{2^{k+1} - 1}{2 - 1}\right)$

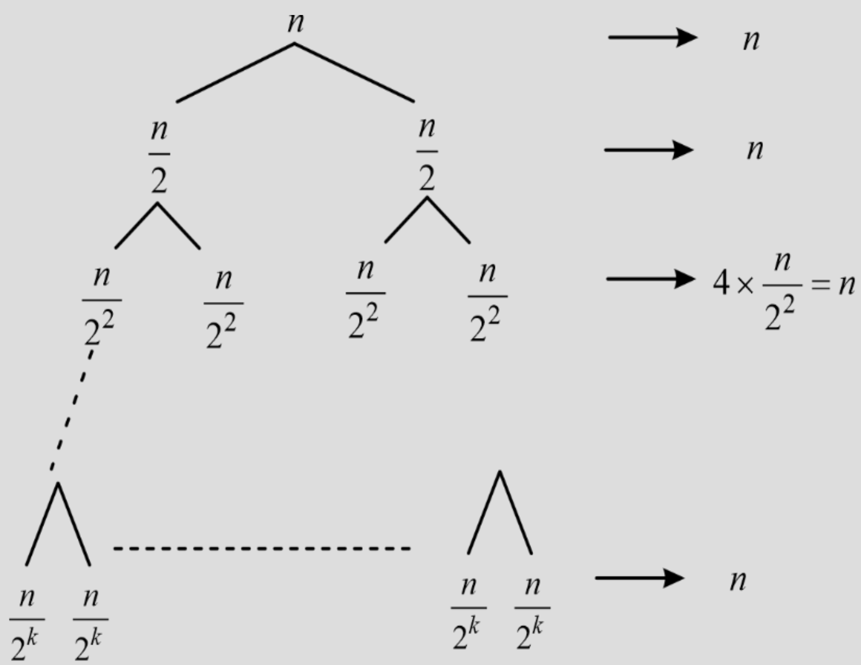
$= C\,(2^{k+1} - 1$

$= C\,(2.2^k - 1)$

$= C\,(2n - 1)$

$= O\,(n)$

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



$\frac{n}{2^k} - 1; n - 2^k; k - \log_2 n$

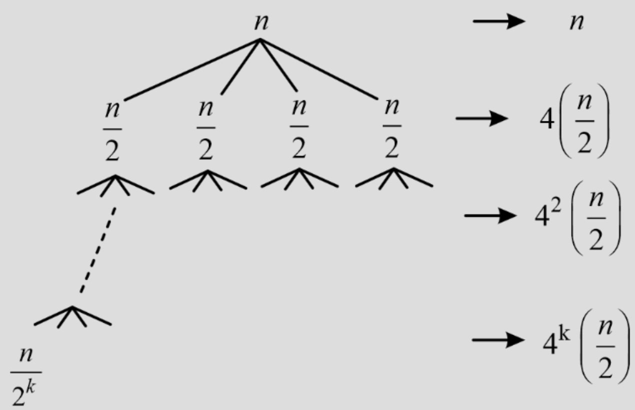
$\therefore n + n + n + \ldots + n$

$= \text{k}\, n$

$= \text{n} \;\; nlog_2\, n$

$= \text{O}(nlog_2\, n)$

(3) $T(n) = 4T\left(\frac{n}{2}\right) + n$



$n = 2^k$, $k = \log_2 n$

$= n + 4\left(\frac{n}{2}\right) + 4^2\left(\frac{n}{2}\right) + \ldots + 4^k\left(\frac{n}{2}\right)$

$$= n\left[1+2+2^2+2^3+\ldots+2^k\right]$$

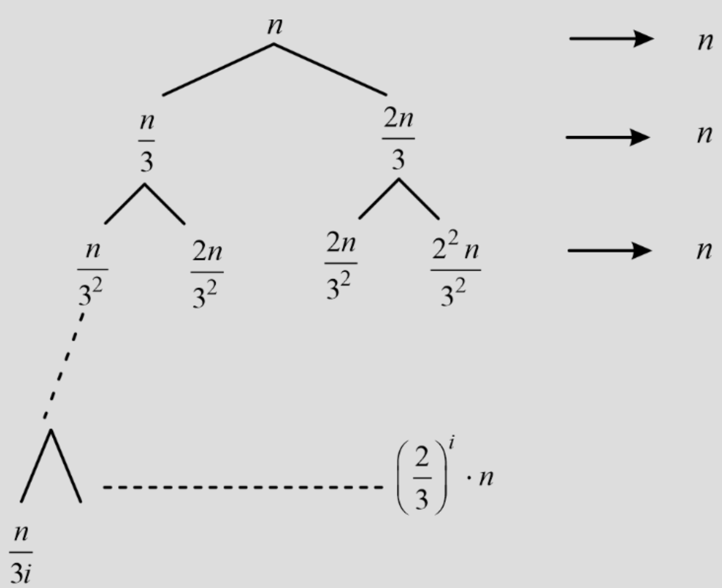
$$= n\left(\frac{2^{k+1}-1}{2-1}\right)$$

$$= n\left(2\cdot 2^{k-1}\right)$$

$$= n\,(2n)-1$$

$$= O\left(n^2\right)$$

(4) $T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+n$ T(1) = 1



$\frac{n}{3i}-1;\ n-3i;\ i-\log_3 n$ 3

$= n + n + \ldots + \log_3\ n\ \mathrm{T}\,(n)$

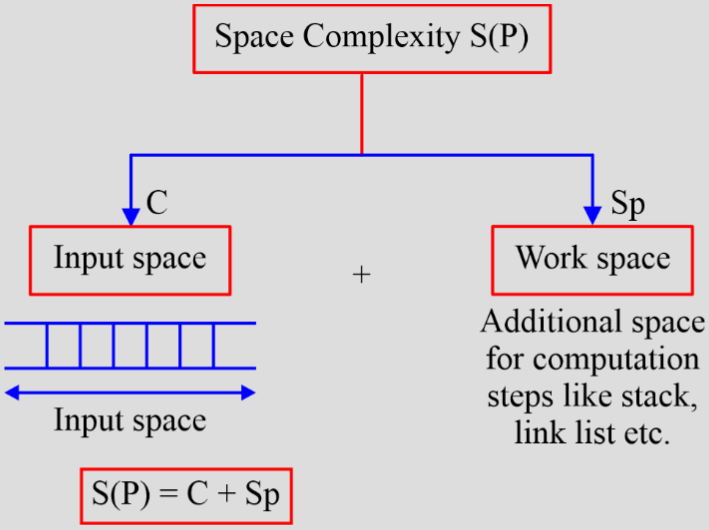
$= (n + n + n + \ldots + \log_3 n) \geq n + \ldots + \log_3 n$

$\Omega(n{\cdot}\log_{3/2}\ n\ )$

1.10 Recurrence Relations and their Time Complexity

$T(n) = C;\ n = 2$ $T(n) = 2\,T(\sqrt{n}) + C;\ n > 2$	O (logn)
$T(n) = C;\ n = 2$ $T(n) = T(n-1) + C;\ n > 2$	$O(n)$
$T(n) = C;\ n = 1$ $T(n) = T(n-1) + n + C;\ n > 2$	$O(n^2)$
$T(n) = C;\ n = 1$ $T(n) = 2T(n-1) + C;\ n > 1$	$O(2^n)$
$T(n) = C;\ n = 1$ $T(n) = 2T\left(\frac{n}{2}\right) + C;\ n > 1$	$\theta(n)$
$T(n) = C;\ n = 1$ $T(n) = 2T\left(\frac{n}{2}\right) + n;\ n > 1$	$\theta(n\log n)$
$T(n) = C;\ n = 1$ $T(n) = T\left(\frac{n}{2}\right) + C;\ n > 1$	$\theta(\log n)$
$T(n) = 1;\ n = 2$ $T(n) = T\left(\sqrt{n}\right) + C;\ n > 2$	$\theta(\log\log n)$

1.11 Space Complexities



Algorithm

```
Int n, A[n];
Algorithm Rsum(A, n)
{
  if (n = 1) return (A(1));
  else;
  return (A[n] + RSum(A, (n–1));
 }
```

- **Time Complexity =** O(n)
- **Space Complexity**
- We need stack space
- Stack is used to store activation records of function calls
- Size of activation records is trivial
- Stack size that we need = O(n)
- Space complexity = O(n)

```
Algorithm A(n)
{
   if (n = 1) return;
   else;
 {
   A(n/2);
  }
}
```

$A\left(\frac{n}{2}\right);$

Recurrence relation

$$T(n) = C;\ n = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + C;\ n > 1$$

Time Complexity = O(log n)

Space Complexity

- Space complexity will depend on number of activation record pushed into the stack
  Suppose, n = 16

A (1)
A (2)
A (4)
A (8)
A (16)

For n = 2^K we are pushing the ‘K’ activation record

$\therefore$ **Space Complexity**

$$n = 2^K$$
$$\log n = K\log_2 2$$
$$K = \log_2 n$$
$$\boxed{\text{Space Complexity} = O(\log n)}$$

Example 3

```
Algorithm A(n)
{
  if (n = 2) return;
  else;
  return (A √n );
}
```

Solution:

$$T(n) = 1;\ n = 2$$
$$T(n) = T\left(\sqrt{n}\right) + C;\ n > 2$$

Time Complexity = O (loglogn)

Space Complexity

Suppose n = 16

A(1)
A(2)
A(4)
A(16)

$\therefore$ For $2^{n/2^k}$ manner we are pushing in stack

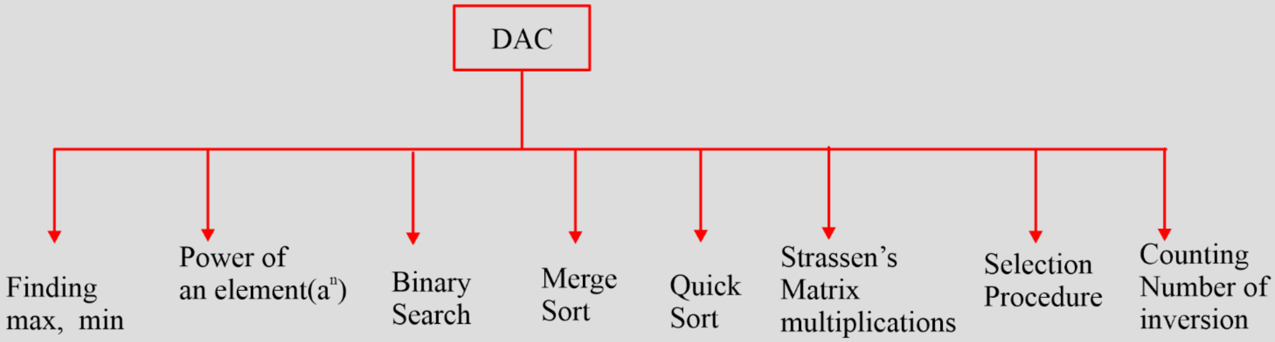
$$2^{n/2^k} \geq 2$$
$$\frac{n}{2^k}\log_2 2 \geq \log_2 2$$
$$n \geq 2^K$$
$$K \leq \log_2 n$$

$$\boxed{\text{Space complexity} = O\left(\log_2 n\right)}$$

❑❑❑

2 Divide and Conquer

2.1 DAC Application



2.2 Finding Maximum Minimum element

Recurrence Relation:

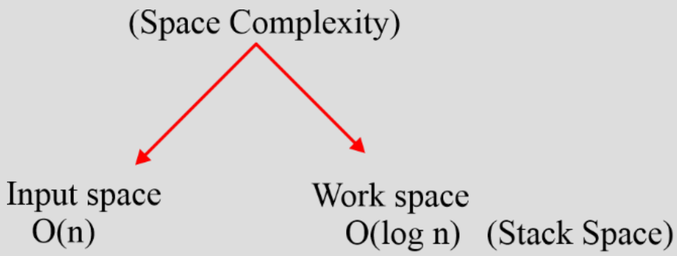
$$T(n) = \begin{cases} 1 \text{ if } n = 1 \text{ or } n = 2 \\ 2T\left(\frac{n}{2}\right) + 1; \ n > 2 \end{cases}$$

Time Complexity:

$$T(n) = O(n)$$

- Time complexity is same for every case (Best case/Worst case).

Space Complexity:



$$\text{Space Complexity} = O(n) + O(\log n)$$
$$= O(n)$$

Number of comparisons to find maximum / minimum element on an given array of n elements:

$$\text{Comparison} = \frac{3n}{2} - 2$$

2.3 Power of an Element

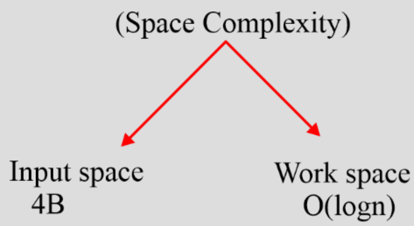
Recurrence relation:

$$T(n) = \begin{cases} 1 \text{ if } n = 1 \\ T\left(\frac{n}{2}\right) + 1;\ n > 1 \end{cases}$$

Time Complexity:

$$T(n) = T\left(\frac{n}{2}\right) + C$$
$$T(n) = O(\log n)$$

Space Complexity:



Space Complexity = 4B + O(logn)

= O(logn)

Number of multiplications to find a^n

Multiplication = O(logn)

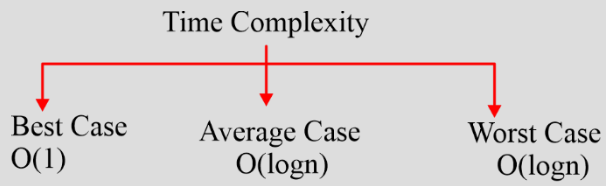
2.4 Binary Search

Given a sorted array and an element x, need to return the index of element x if it is present then 1, otherwise – 1.

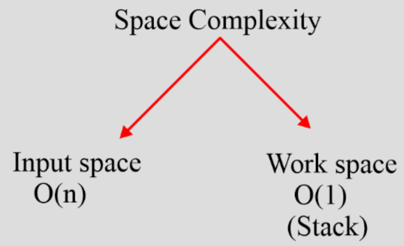
Recurrence relation:

$$T(n) = \begin{cases} 1; & n = 1 \\ T\left(\frac{n}{2}\right) + C; & n > 1 \end{cases}$$

Time Complexity:



Space Complexity:



Space complexity = O(n) + O(1)

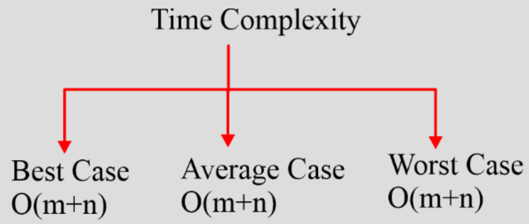
= O (n)

2.5 Merge Algorithm

- Merging two sorted sub arrays of input size m,n.
- Number of comparisons to merge two sorted sub arrays of size m,n.

Comparisons = m + n – 1 (worst case)

Number of moves =m + n (Outplace Algorithm)



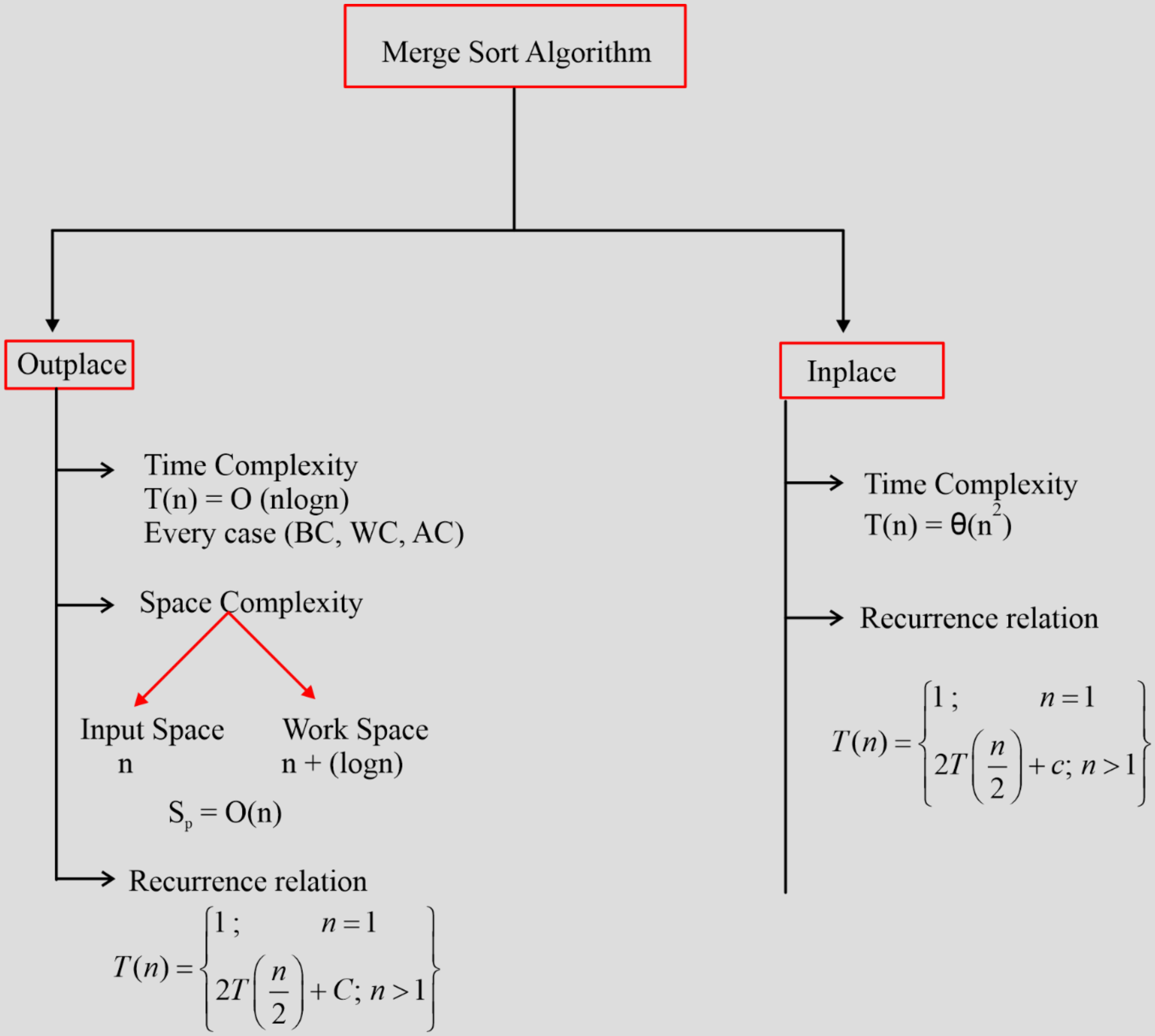
Number of comparisons in best case of merging two sorted subarrays of size m, n.

comparisons = min (m, n)

Moves = m + n (Always)

Note:
Best Case comes in comparisons no effect on moves.

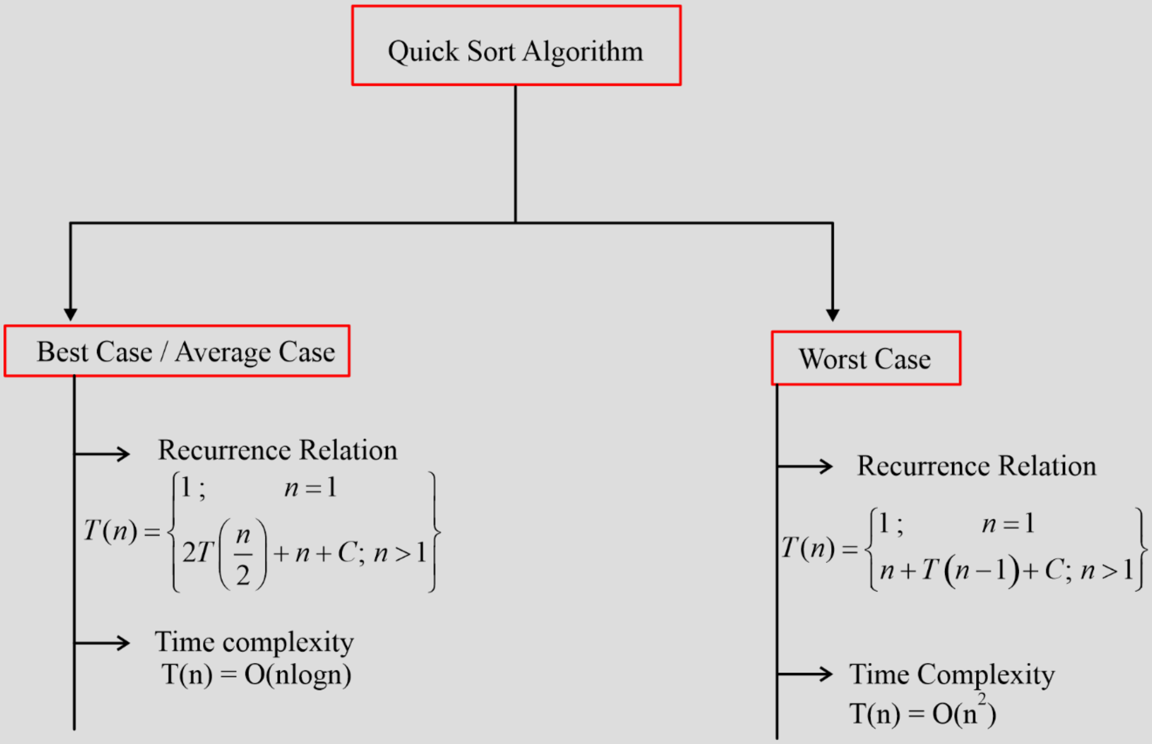
2.5.1 Merge Sort Algorithm:



Note:

- In GATE exam if merge sort given then always consider outplace.
- If array size very large, merge sort preferable.
- If array size very small, then prefer insertion sort.
- Merge sort is stable sorting technique.

2.6 Quick Sort Algorithm



Example 1: In Quick for sorting n elements, the $\left(\frac{n}{16}\right)^{th}$ smallest element is selected as pivot. what is the worst-case time Complexity?

Solution.

$$T(n) = T\left(\frac{n}{16}\right) + T\left(\frac{15n}{16}\right) + O(n)$$

= (solve by recursive tree method)

Example 2: The median of *n* elements can be found in O (n) time then, what is the time complexity of quick sort algo in which median selected as pivot?

Solution.

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

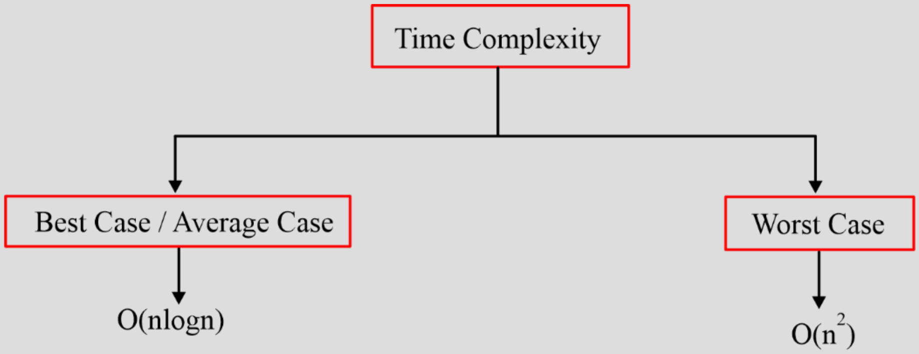
Find median swap median with last Partition algo

$= 2T(n/2) + C \cdot n$

$= O(n \log n)$

2.6.1 Randomized Quick Sort

- In Randomized quick short algorithm selection of pivot element can be taken randomly.



2.7 Counting Number of Inversion

- Counting number of inversion on given an array of an element.

  Time complexity $T(n) = O(n\log n)$

2.8 Selection Procedure

Find K^{th} smallest on given an array of an element and integer K.

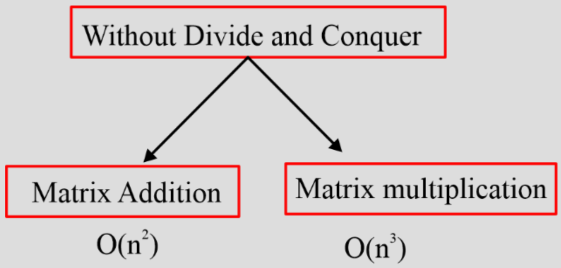
Time Complexity:

$T(n) = O(n^2)$

Space complexity:

Space Complexity = $O(n)$

2.9 Strassen's matrix Multiplication



Algorithm

According to Strassen's

Time Complexity

$O(n^{2.81})$

Recurrence relation

$$7T\left(\frac{n}{2}\right)+n^2$$

2.10 Comparison Based Sorting Algorithms

Sorting Algorithm	Basic logic of sorting Algo	BC	AC	WC	Stable sorting	Inplace sorting
Quick sort	Choose pivot element place in correct position	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n^2)$	No	Yes
Merge sort	Divide to equal parts recursively sort each sub part & marge them	$\theta(n\log n)$	$\theta(n\log n) = n \log n$	$\theta(n\log n) = n \log n$	Yes	No
Heap sort	Build heap(max) delete max place	$\theta(n\log n)$	$\theta(n\log n)$	$\theta(n\log n)$	No	Yes
Bubble sort	Compare exchange	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	Yes	Yes
Selection sort	Find position of min element from [1 to n]	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	No	Yes
Insertion sort	Insert a [i + 1] into correct position	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	Yes	Yes

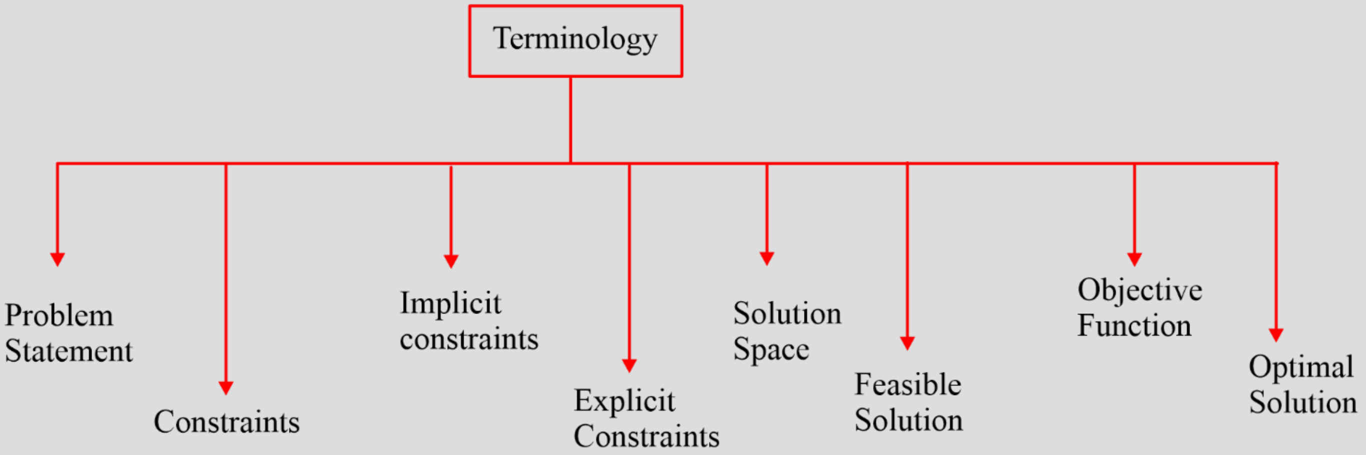
❑❑❑

3 Greedy Technique

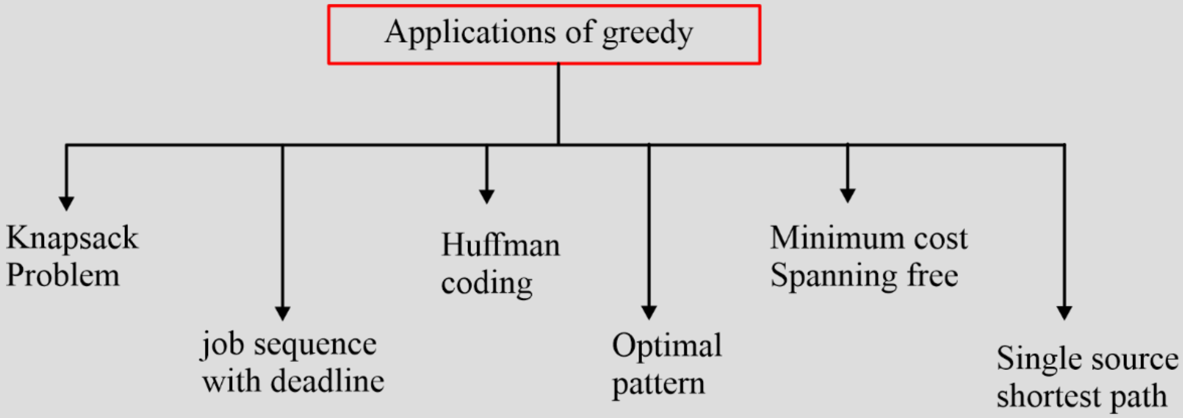
3.1 Greedy Technique

- Greedy method is an algorithm design strategy used for solving problems where solution are seen as result of making a sequence of decisions.
- A problem may contain more than one solution.

3.2 Terminology



3.3 Applications of greedy

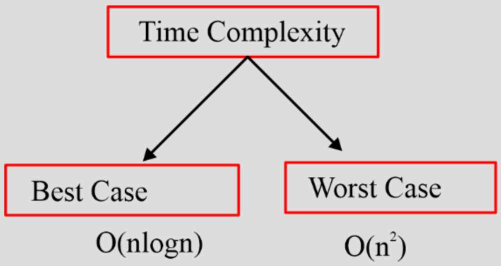


3.4 Knapsack Problem

Time complexity T(n) = O(nlogn)

3.5 Job Sequence with Deadline

- Single CPU only.
- Arrival time of each job is same.
- No pre-emption.



3.6 Optimal Merge Pattern

- This is a problem related to merging of files. Given a set of n-files in sorted order. It is required to merge them into a single sorted file with 2-way merging.
- This problem is like merging process in merge sort. In merge sort we were interested in number of comparisons but in optimal merge pattern we are interested in record movement (i.e moving a record from one file to another file).
    - If file F1 has 'n' records and file 'F2 ' has 'm' records then number of record movement will be 'm+n'. 1 2

  The problem of optimal merge pattern involves merging of n-files (n≥2).
- At any point choose two records with least weight merge them and put them in list and continue it until all records are merged.
- Time complexity T(n) = O(nlogn)
- Space complexity = O(n)

3.7 Huffman Coding

- Huffman coding is essentially a non-uniform encoding with convention that the character with higher frequency (probability) of occurrence will be enclosed with less number of bits.
- It comes under data compression technique.
- Time complexity T(n) = O(nlogn)

3.8 Minimum Cost Spanning Tree

3.8.1 Graph

Graph (V., E)

Set of vertices   set of edges

- Let G(V, E) be a simple graph then

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

$E \leq \frac{V(V-1)}{2}$

$E \leq C.V^2$ C is constant

Note:

$E = O(V^2)$

$\text{Log } E = O(\log V)$

3.8.2 Graph Representation

Graph Representation

Adjacency matrix   Adjacency list

- For more edges (Dense Graph) Adj. matrix is better (density more).
- For less edge (sparse graph) Adj list is better.

	Matrix	**List**
(1) Finding degree of vertex $\Rightarrow$ Time Complexity	O(V) Every Case	O(1) Best Case O(V1) Worst Case
(2)Finding total edges $\Rightarrow$ Time Complexity	$O(V^2)$ Every Case	O(V+2E) Worst Case O(V) Best Case
(3) Finding 2-vertices adjacent (or)not $\Rightarrow$ Time Complexity	O(1)	O(V-1) Worst Case O(1) Best Case
(4) G(V,E) $\Rightarrow$ space	$O(V^2)$ Every Case	O(V+E) Every Case

3.8.3 What is Spanning Tree

A subgraph T(V, E') of G(V, E) where E' is the subset of ($E' \subseteq E$) is a spanning tree iff 'T' is a tree.

A sub graph G('V, E') of G(V, E) is said to be spanning tree.

(1) T' should contain all vertices of G

(2) T' should contain (V-1) edged where V is number of vertices without cycle.

(3) T’ should connected.

3.8.4 Minimum Cost Spanning Tree

Minimum cost spanning tree is the one in which cost of the spanning tree formed should be minimum.

3.8.5 Prims Algorithm

- Select Any vertex

  Time complexity = V + VlogV + 2E + ElogV

  = O(E + V)logV

Using Sorted Array & Adjacency List

V + 2E + E × V = O(EV)

Using Sorted Array & Adjacency List

V × O(1) + V2 + E × V= O(EV)

3.8.6 Kruskal algorithm

- Take first minimum edge

  Time complexity = E log E + (V + E)

  = O(E log E) = O(ElogV)

  If edges are already sorted

  TC = O(E + V)

3.9 Single Source Shortest Path

3.9.1 Dijkstra Algorithm

- Using min heap & adjacency list = O(E + V)logV
- Using adjacency Matrix & Min heap = $O(V^2 \text{ ElogV})$
- Using adjacency list & Unsorted Array = $O(V^2)$
- Using adjacency list & Sorted Doubly Linked List = O(EV)

3.9.2 Bellman-Ford

- Time Complexity = O(EV)
- If negative edge weight cycle then for some vertices Incorrect answer.

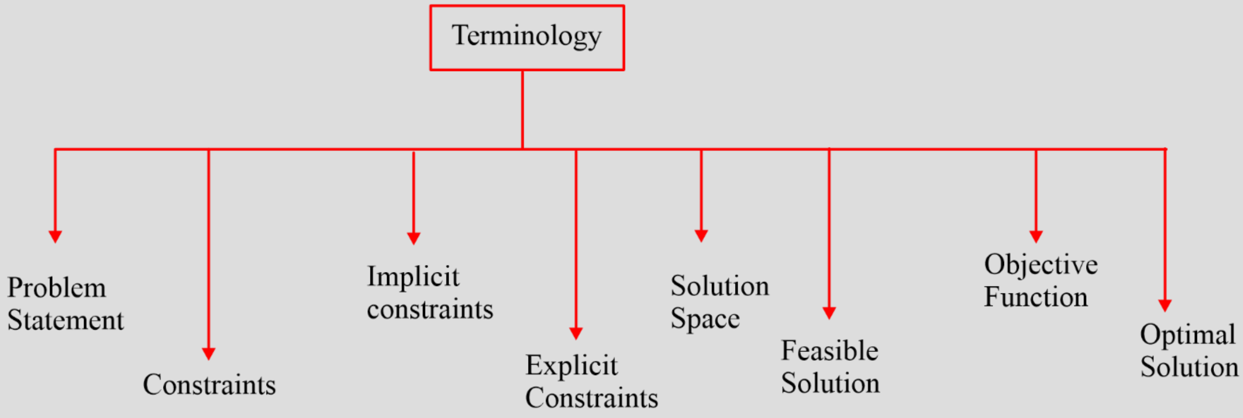
❑❑❑

4 Dynamic Programming

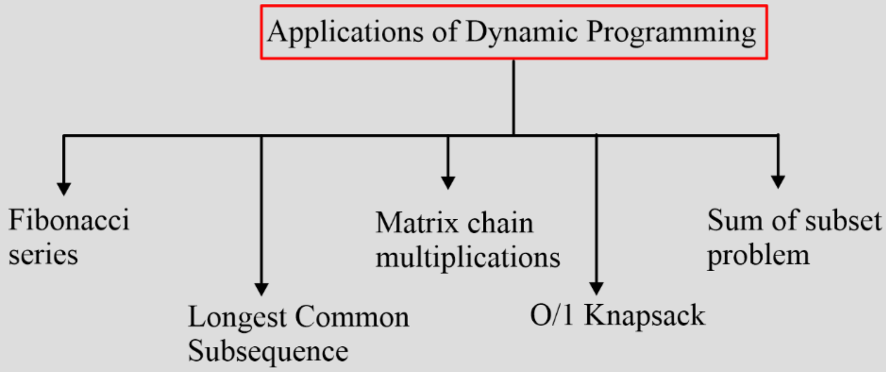
4.1 Dynamic Programming

In dynamic programming for optimal solution always computes distinct function calls.

4.2 Terminology



4.3 Application of Dynamic Programming

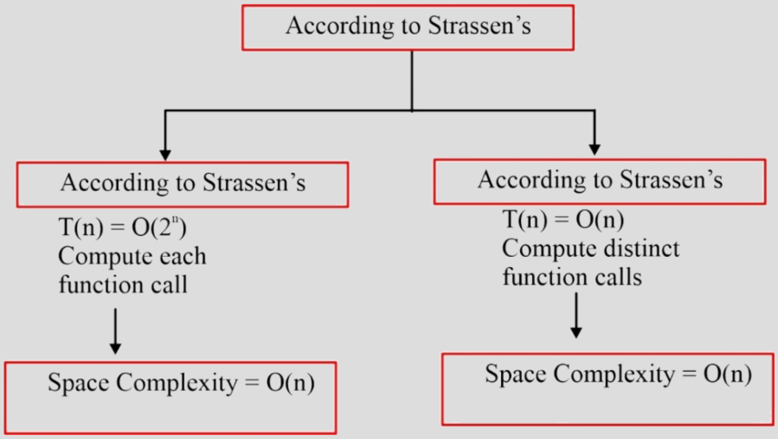


4.4 Fibonacci Series

- Time complexity T(n) = O(nlogn)
- Computes distinct function calls.

4.5 Job Sequence with Deadline

- Single CPU only
- Arrival time of each job is same
- No pre-emption



4.6 Longest Common Sub sequence (LCS)

- For common subsequence always consider two strings:
- P = <ABCDB> – Q = <BDCABA>
- Common subsequences for both 'p' are
- S = <A>
- S = <AB>
- S = <CAB>
- S = <BDAB>
- A common subsequence of longest length is known as longest common subsequence.
- For above problem longest common subsequence will be of length u.

4.6.1 Applications of LCS

1. Genomics
2. Software engineering applications
3. Plagiarism
4. Data gathering system of search engines

4.6.2 Algorithm for LCS

LCS (p,q)
{

1. *For* $i \leftarrow 0 \ to \ n-1$

$L[i-1]=0$

2. *For* $j \leftarrow 0 \ to \ m-1$

$L[-1, j]=0$

3. *For* $i \leftarrow 0$ *to* $n-1$

   For $j \leftarrow 0$ *to* $m-1$

   $If\ (p[i]=q[j])\ then$

   $L[i,j]=1+L\ (i-1,j-1];$

   else

   $L[i,j]=max\{L[i,j-1],L[i-1,j]\}$

}

- Time complexity of step 1 = O(n)
- Time complexity of step 2 = O(m)
- Time complexity of step 3 = O(mn)
- Total Time complexity = O(n) + O(m) + O(mn)
  = O(mn)
- Space complexity = O[(M+1).(n+1)]
  = O(mn)

4.7 Matrix Chain Multiplications

Two matrices ‘A’ and ‘B’ are compatible if and only number of column of first matrix must be equal to number of rows of second matrix.

4.7.1 Brute force method

Number of parenthesizing for a given chain is given by Catalan number: $\left[\frac{1}{n+1}\ {}^{2n}C_n\right]$

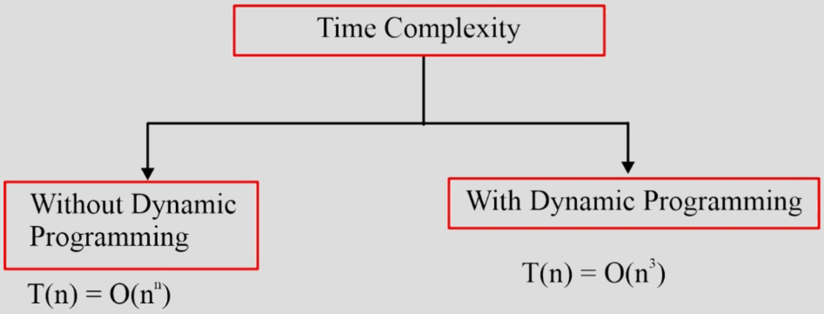
Time complexity = $O(n^n)$

Space complexity = O(n)

4.7.2 Algorithm For Matrix Chain Multiplication

The time complexity of multiply the given chain of n matrices <A_1, A2 A3…. An> using dynamic programming (district function call) is $O(n^3)$

Space complexity = $O(n^2)$



4.8 O/1 Knapsack Problem

The maximum profit can be achieved by O/1 knapsack problem where capacity of problem is ‘p’ and number of objects are ‘q’.

Time Complexity

Without Dynamic Programming

$T(n) = O(2^p)$
Space complexity = O(p)

With Dynamic Programming

$T(n) = O(pq)$
Space complexity = O(p)

4.9 Sum of Subset Problem

- Given n-elements and an integer 'm', it is required to determine whether there exists a subset of given n elements, whose sum equal M.
- This is a decision problem (True/False).

4.9.1 Algorithm for Sum of Subset Problem

```
SoS(n, M, A)
// A [1 . . . n] is an array of elements
    {
    1.  for i = 0 to n
            for j = 0 to M
            if (i > =0 and j = 0)
            SoS [i, j] = T
    else
            if (i = 0 and j > 0)
            SoS [i, j] = F;
    else
            if (A[i] > j)
            SoS [i, j] = SoS [i – 1, j]
    else
            SoS [i, j] = SoS [i – 1, j] or
            SoS [i – 1, j – A[i]]
    }
```

4.9.2 Time Complexity of SoS

Two for loops are there thus repeating for (n * m) times. Thus, time complexity = O(n * m)

Time complexity of SoS becomes exponential if $M = 2^n$

$$T.C = O(n \times 2^n)$$