# Divide and Conquer Concepts

In this, a problem is divided into sub-problems and the solutions of these sub-problems are combined into a solution for the large problem.

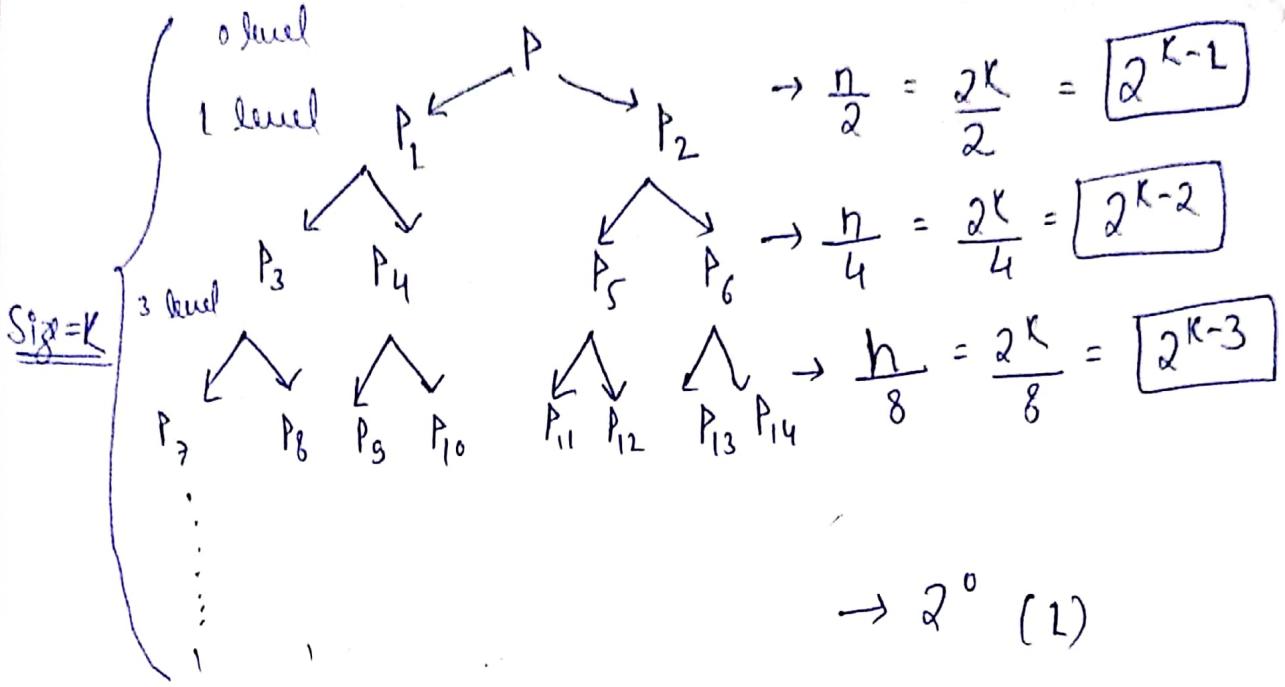
Steps :-

1) - Divide (Actual problem Sub problem ko divide)
2) - Conquer (Subproblem Solution)
3) - Combine (Solution Combine)

Algorithm :- (Given a problem P of size n = $2^k$)

1) - Algorithm DAnd C (P) {
2) - If n is small, solve it
3) - else {
4) - Divide 'P' into sub-problems $P_1$ $P_2$ $P_3$ P.... $P_K$
5) - Apply DAnd C to each of these sub-problem
6) - Combine all the solutions
7) - }
8) - }

P (Problem) Size $n = 2^k$




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

$\rightarrow \frac{n}{4} = \frac{2^k}{4} = 2^{k-2}$

$\rightarrow \frac{n}{8} = \frac{2^k}{8} = 2^{k-3}$

$\rightarrow 2^0$ (1)

## Time Complexity

P Size = n

$$f(n) = f\left(\frac{n}{2}\right) + f\left(\frac{n}{2}\right) + f(n) \quad \text{(Combine) (Const)}$$

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

$$= 3f\left(\frac{n}{3}\right) + f(n)$$

$$f(n) = a \cdot f\left(\frac{n}{a}\right) + fn$$

$f(n) = 2f\left(\frac{n}{2}\right) + n$ (Substitution method)

Divided by :- 2

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

$f\left(\frac{n}{2}\right) = 2 \cdot f\left(\frac{n}{4}\right) + \frac{n}{2}$ — Value put in equation -

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

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

$\vdots$

$8f\left(\frac{n}{8}\right) + 3n$

$= n \cdot f\left(\frac{n}{n}\right) + Kn$

$n = 2^k \Rightarrow k = \log_2 n$ $= nf(1) + K.n$

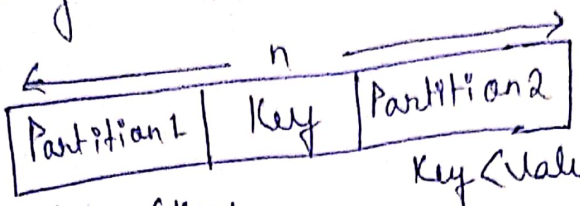
$f(n) = K.n$

$f(n) = \log n . n$

$f(n) = n \log n$

Imp. # Quick sort in Data Structure

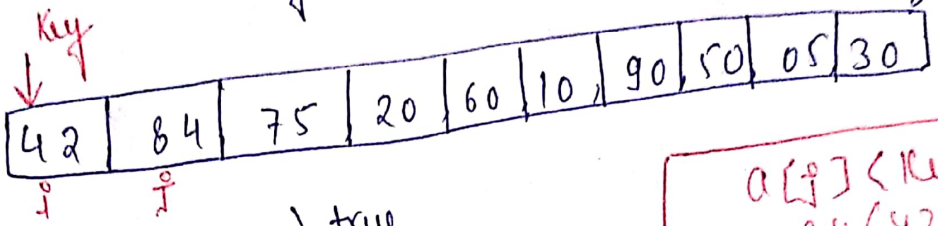
Mainly work on partitioning




Value < Key

Key < Value

Size = 10




if (a[i] < Key) true

i = i++

a[i] and a[j] Swap

j = j++

else j = j++

a[i] < Key false

84 < 42

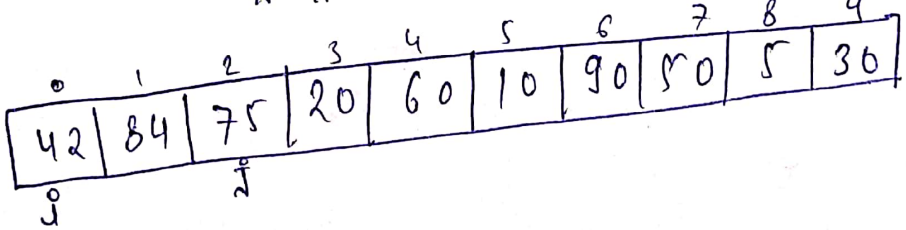
j = j++

Key = 42 a[j] = 84

a[j] < Key false

84 < 42

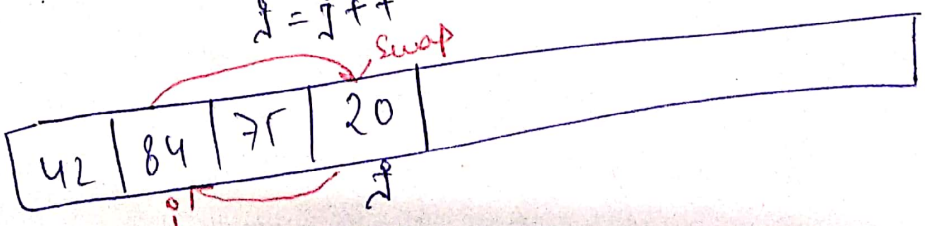
j = j++ (only j increment)




a[j] < Key

(75) < 42 false

j = j++

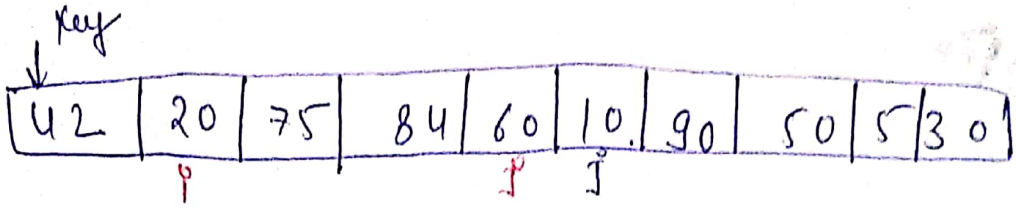



a[i] < Key

20 < 42 true

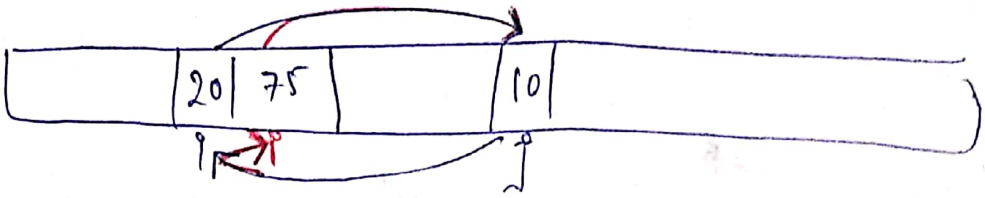
i = i++

a[i] and a[j] Swap i++




a[j] < key

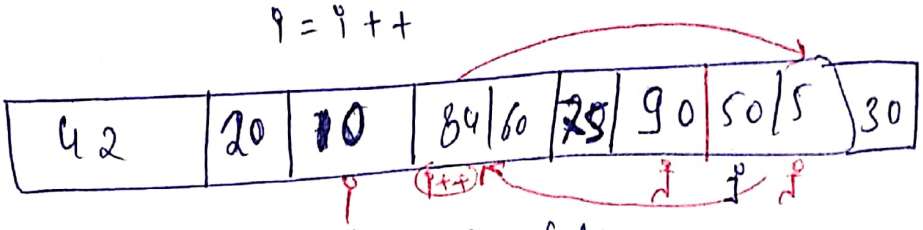
60 < 42 false




a[j] < key

10 < 20 true

i = i++



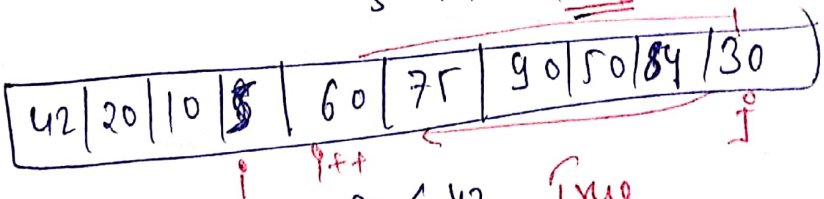

90 < 42 false

j = j++

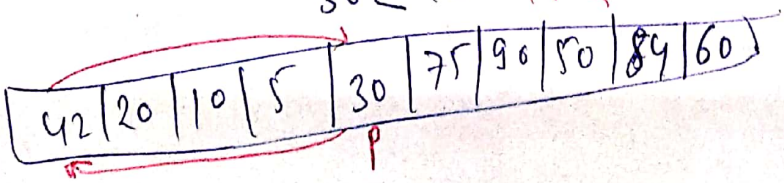
50 < 42 false

j = j++

5 < 42 true

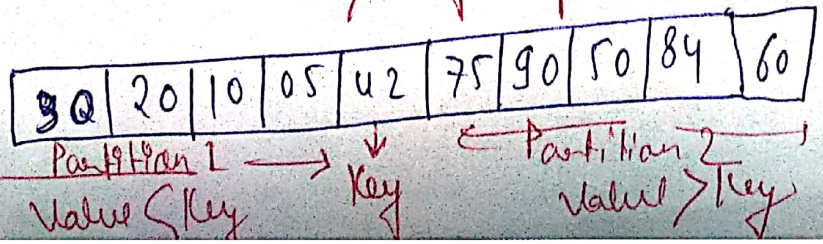



30 < 42 true




if j == max size - 1

a[i] & key swaps



In quick sort algorithm, partitioning the list is performed using following steps:-

1) - Consider the first element of the list as pivot (i.e. element at 1st position in the list)

2) - Define two variables i and j. Set i and j to 1st and last elements of the list respectively.

3) - Increment i until list[i] > pivot then stop.

4) - Decrement j until list[j] < pivot then stop.

5) - if i < j then exchange list[i] and list[j].

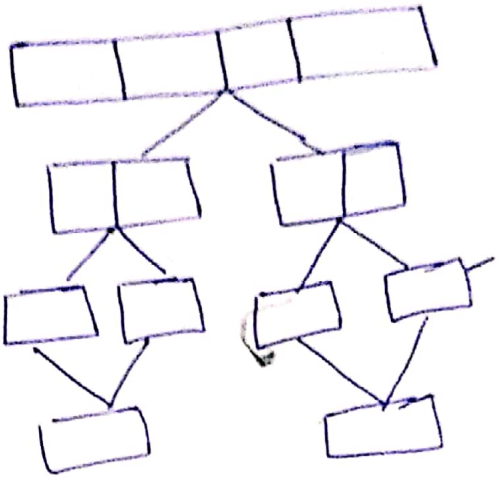
6) - Repeat steps 3, 4, and 5 until i > j.

7) - Exchange the pivot element with list[j] element.

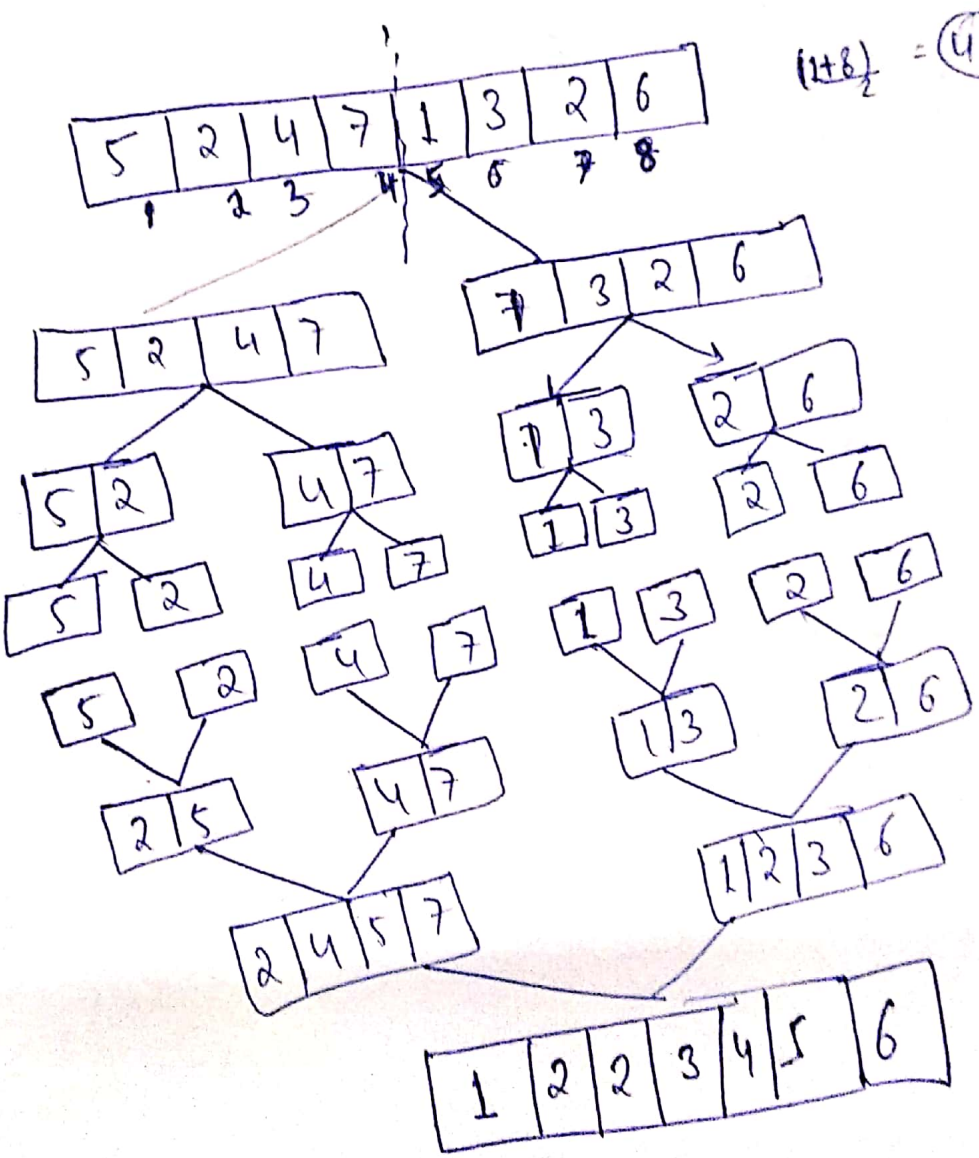
# Merge Sort

(Divide, Conquer and Combines)

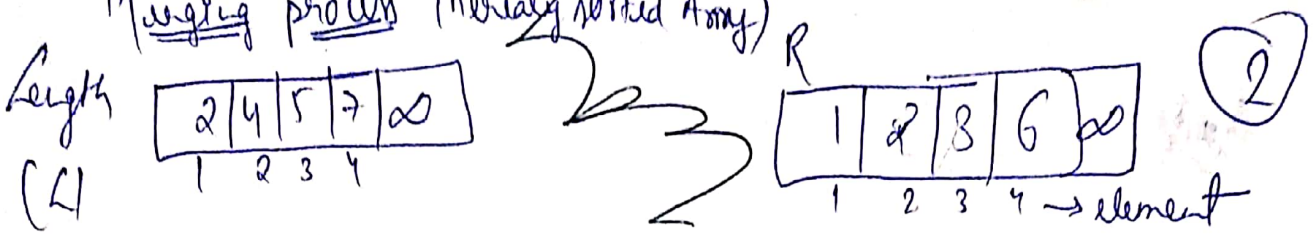




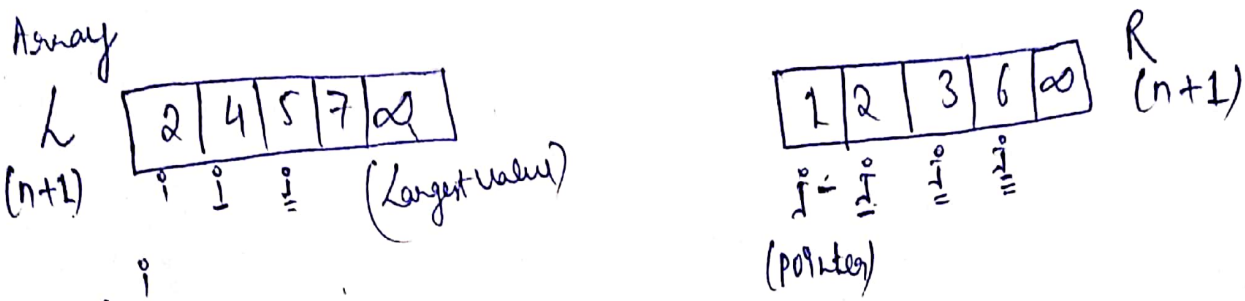
Divide and Conquer

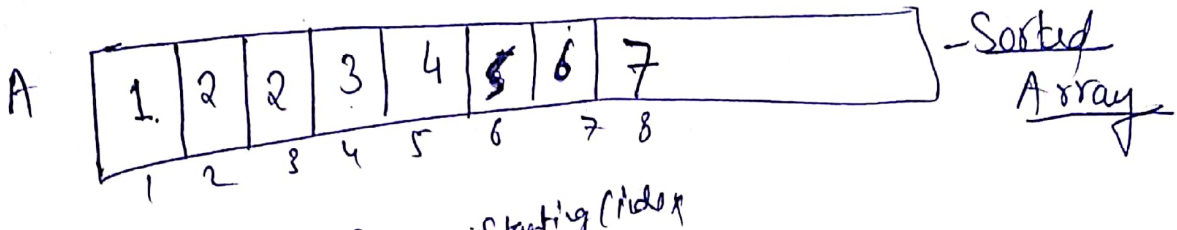


Merging process (Already sorted Array)




(2)




Merg-Sort (A p q r) p - starting index, q - ending index, r

If (p < r) otherwise fail condition

P = 1

~~8~~ = 8

q = 4

15

p = 1  r = 8  q = 4

| 5 | 2 | 4 | 7 | 1 | 3 | 2 | 6 |
|---|---|---|---|---|---|---|---|

Merge (A, p, q, r)

| 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

1)- $n_1 \leftarrow q - p + 1$

$= 4 - 1 + 1$

$= 4$

2)- $n_2 = r - q$

$= 8 - 4$

$= 4$

3)- Create array $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$

Array (L)

| 2 | 4 | 5 | 7 | ∞ |
|---|---|---|---|---|
| 1 | - | - | - | (n+1) |

Array (R)

| 1 | 2 | 3 | 6 | ∞ |
|---|---|---|---|---|
| 1 | - | - | - | (n+1) |

4)- for $i \leftarrow 1$ to $n_1$

5). $L[i] \leftarrow A[p + i - 1]$

6)- for $j \leftarrow A[p + i]$.  Loop

7)-

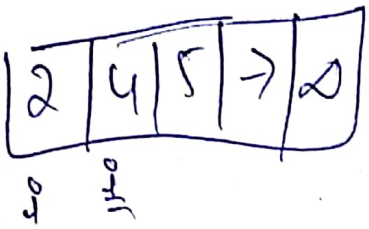
Loop (p to r)

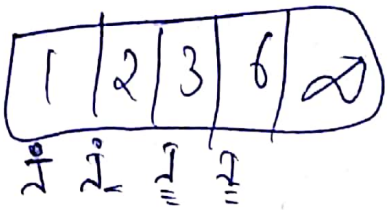
12)- for $k \leftarrow$ p to r

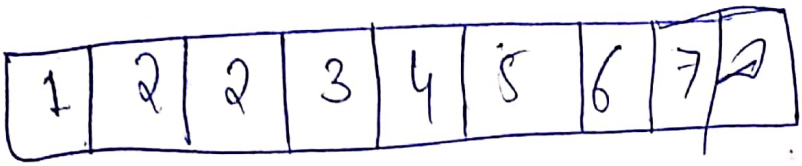
p = 1 , r = 8

Loop (1 ... 8)

Length



## 9ap Dynamic Programming

Dynamic Programming is also used in Optimization problems. Like Divide-and-Conquer method, Dynamic Programming Solves problems by Combining the solution of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just Once and then saves its Answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are Overlapping sub-problems and Optimal Substructure.

OR

The definition of Dynamic Programming says that it is a Technique for solving a Complex problem by first break into a collection of simpler subproblems, solving each sub-problem just once, and then storing their solution to avoid repetitive Computations.

Let's understand this approach through an example:-

Consider an example of the fibonacci series, the following series is the fibonacci series :-

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

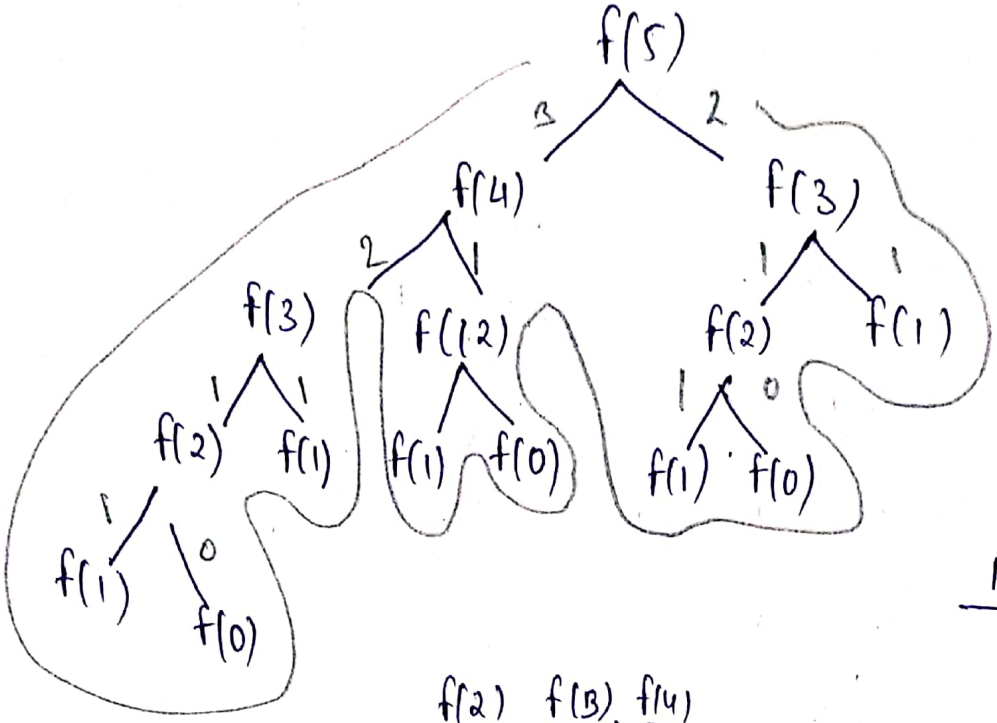
The numbers in the above series are not Randomly Calculated. Mathematically, we could write each of the terms using the below formula:

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

With the base values $f(0)=0$, and $f(1)=1$.

To Calculate the other numbers, we follow the above relationship. for example f(2) is the sum of f(0) and f(1), which is equal to 1.

```
f(n)
{
if (n == 0)
   return 0;
if (n == 1)
   return 1,
if (n > 1)
return (f(n-1) + f(n-2));
}
```




Top

Bottom

Move Left to Right

| f(2) | f(3) | f(4) |
|---|---|---|
| 1 | 2 | 3 |

The following are the steps that the Dynamic Programming follows :-

i) - It breaks Down the Complex problem into simpler subproblems.

ii) - It finds the Optimal solution to these sub-problems.

iii) - it stores the result of Subproblems. The process of storing the results of sub problems is known as memorization.

iv)- It saves them so that same sub-problem is calculated more than once.

v)- finally, Calculate the result of the complex problem.

## Approaches of Dynamic programming

- Top-Down approach
- Bottom-up approach

## Advantages

1)- It is very easy to understand and Implement.

2)- It solves the sub problems only when it is required.

3)- It is easy to debug.

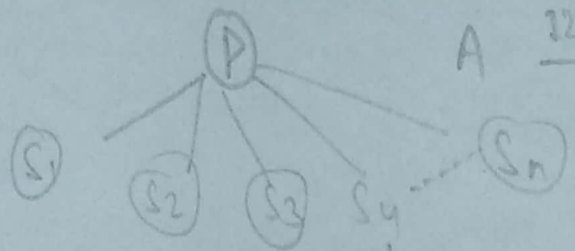
# Greedy Algorithm

A Greedy algo is a strategy that makes the Optimal choice (But choice) at each stage with hope of finding a global optimal.
(1L to other Location.)

Pros:- simple, easy to implement, runfast.

Cons:- do not always yield optimal solution.

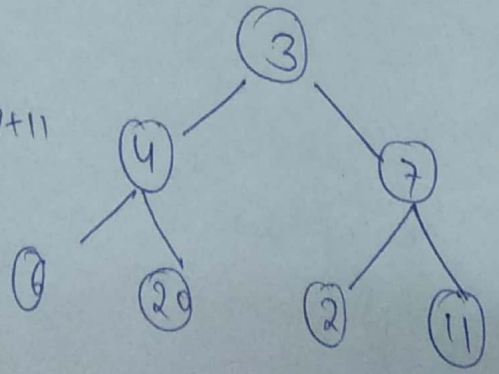
Optimization Problem
- finding Min^m Optimal (1L to OL min cost)
- finding Max^m Optimal (max^m Deadline problem Solve)




A —12hr.→ B — (min^m cost)

12 hrs → feasible solution
↓ optimal problem

Maximization




Greedy approach:- 3+7+11 = 21

Actual:- 3+4+20 = 27

Minimization

Greedy approach:- 3+4+6 = 13

Actual :- 3+7+2 = 12

G2

## Algorithm

a[1 ... n]

```
Algo Greedy(a,n)
{
 Solution = 0;
 for i to n do
  {
    x = select(a)
      if feasibal(solution, x)
        then solution Union (solution x)
  }
  return solution;
}
```

## Application

1)- Activity Selection Problem.
2)- Huffman Coding
3)- Job Sequencing problem
4)- Fraction Knapsack Problem
5)- finding Minimum Spanning trees
6)- Single Source Sortest path.

# Huffman Coding



Huffman Code used to Compress the data upto 90% suppose we have 100 character data having 19 x different character.

7 bit - 128 char
8 bit - 256 char

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 45 | 13 | 12 | 16 | 9 | 5 |

ASCII 01000001, 01000010, 01000011, 01000100, 01000101, 01000110 = 100 x 8 bit = 800 (Actual mtg)

Fixed Length 000 001 010 011 100 101 = 100 x 3 bit = 300 bit

Variable Length (Not fixed size) 0 101 100 111 1101 1100

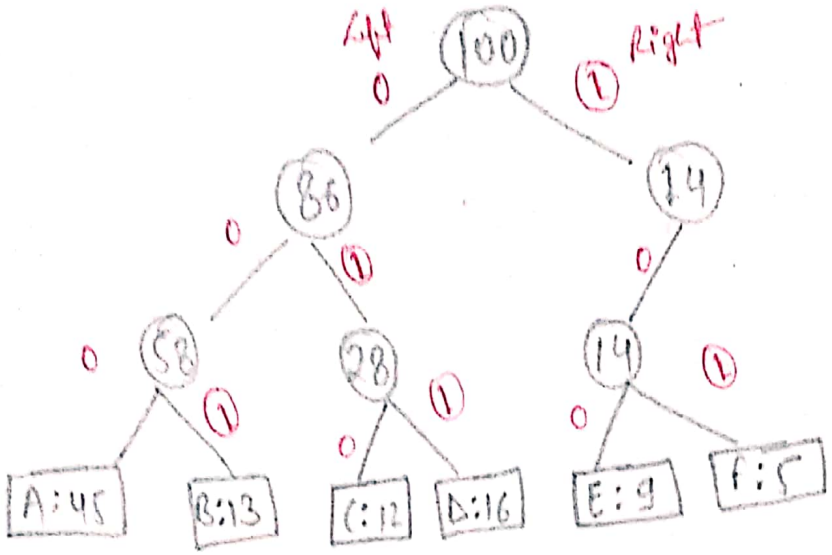
ABCD.... ABBI .... ABCDEF ..

100

000
001
010
011
100
101
110
111

0 101 100 111 1101 1100

45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4 = 224 bit

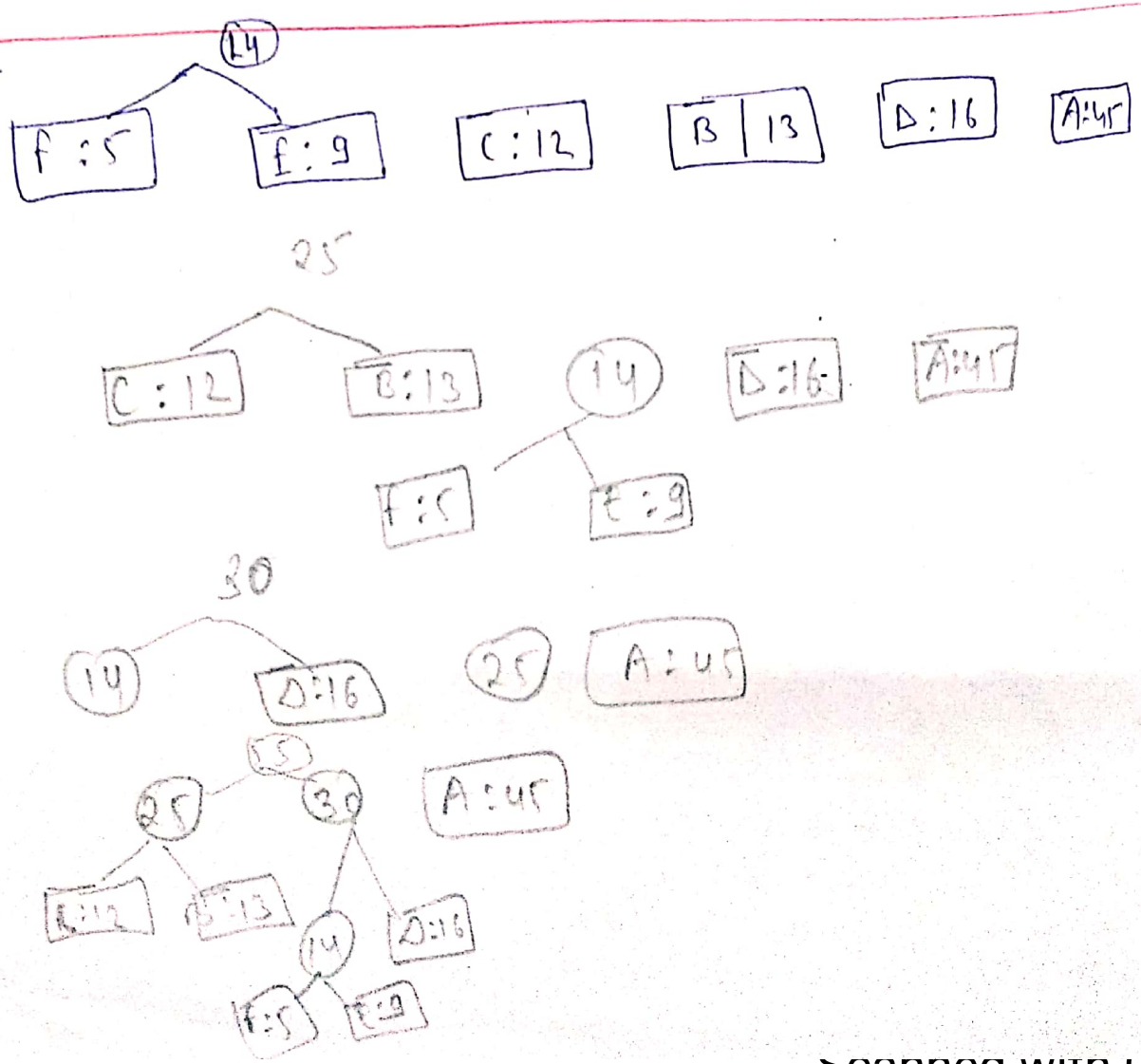


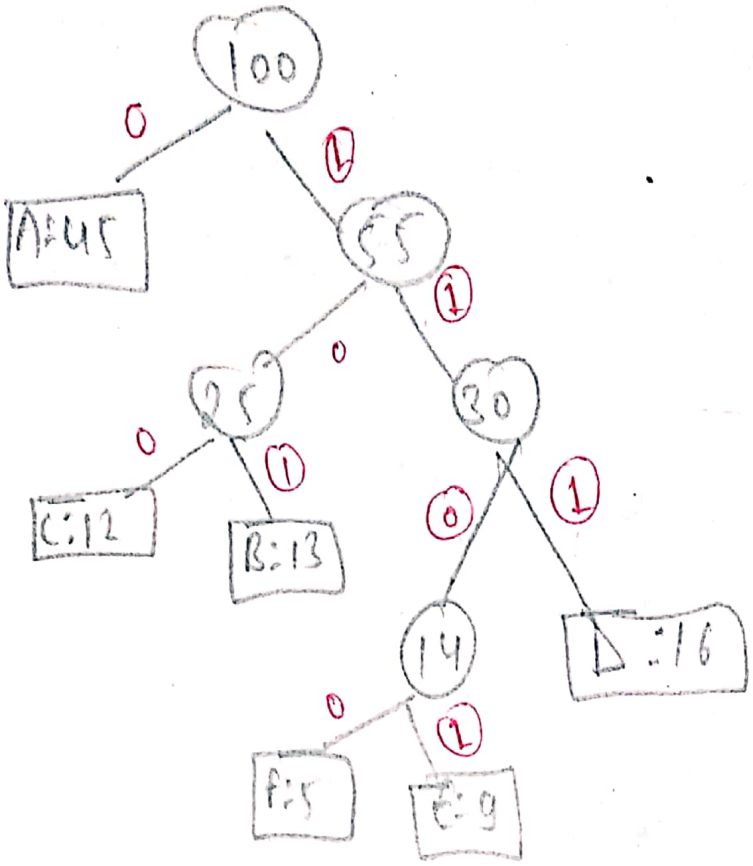

Total size = 300 bit + 6 × 8 + 18

300 bit → Total size
6 → Character
8 → Actual data
18 → Total bit

⇒ 300 + 48 + 18 ⇒ 366 bit

Sorting






$= 224 \text{ bit} + 6 \times 8 + 18 = \boxed{290} \text{ bit}$

↓

Actual msg

Imp KNAPSACK PROBLEM

Let there are 'n' number of objects and each object is having a weight and contribution to profit

Fractional Knapsack

↳ Knapsack Capacity 'M' (Max^m Capacity of BAG)

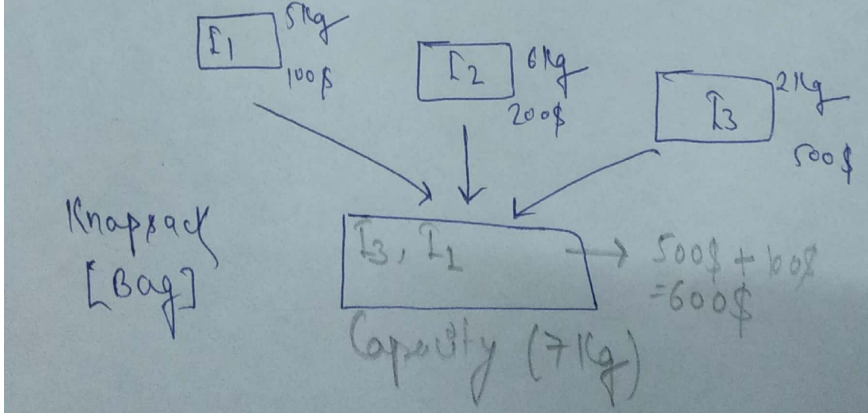
↳ Objective is to fill the knapsack in such a way that profit shall be max^m.

↳ fractional of item can also be added.

maximize $\sum_{i=1}^{n} P_i x_i$ → Profit value

↳ fractional of ith object

Weight $\sum_{i=1}^{n} w_i x_i \leq M$



8kg

$I_3, I_1, \frac{1}{6} I_2$

500 100 $\frac{1}{6} \times 209$

## Fractional Knapsack - Greedy Strategies:-

Strategy 1:- Items are arranged by their profit values. Here an item with max$^m$ profit is selected first (32.8)

Strategy 2:- Items are arranged by weights and an item with minimum weight is selected first. (32)

Strategy 3:- Items are arranged by Profit/weight ratio and item with max$^m$ 'P/W' ratio is selected first. (34.67)

① -

weight Profit

| i | w | P |
|---|---|---|
| 1 | 18 | 30 |
| 2 | 15 | 21 |
| 3 | 10 | 18 |

→ Already sorted

| i | w | P |
|---|---|---|
| 1 | 18 | 30 |
| 2 | 15 | 21 |
| 3 | 10 | 18 |

M=20

Max$^m$ profit

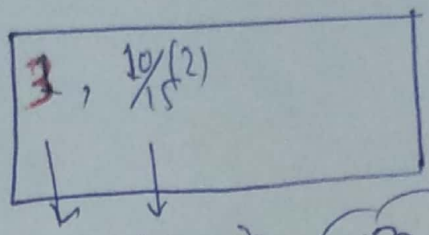
1, $\frac{2}{15}$(2)

20 capacity
2 kg

30 $\frac{2}{15} \times 21 = 32.8$

(2) Minimum weight -

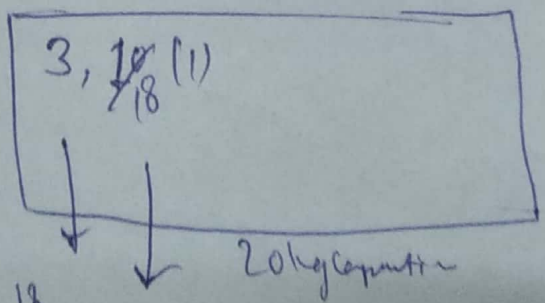
| i | wi | Pi |
|---|---|---|
| 3 | 10 | 18 |
| 2 | 15 | 21 |
| 1 | 18 | 30 |




(3) -

| q | w | P | P/w |
|---|---|---|---|
| 1 | 18 | 30 | 1.67 |
| 2 | 15 | 21 | 1.4 |
| 3 | 10 | 18 | 1.8 |

(Profit max^m weight)




$\frac{10}{18} \times 30 =$

$18 + 16.67 = 34.67$

Q- fractional Knapsack- Solved Question

find the optimal solution for the fractional knapsack by using the Greedy approach

$n = 5$

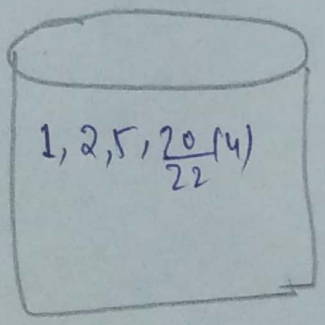
$W = 60kg$

$(w_1, w_2, w_3, w_4, w_5) = (5, 10, 15, 22, 25)$

$(P_1, P_2, P_3, P_4, P_5) = (30, 40, 45, 77, 90)$

| items | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| P | 30 | 40 | 45 | 77 | 90 |
| W | 5 | 10 | 15 | (22) | 25 |
| P/W | (6) | (4) | 3 | (3.5) | (3.6) |

↓ Max^m profit




$60 - 5 = 55$ (W)

$55 - 10 = 45$

$45 - 25 = 20$

$20 - 20 = 0$

Total profit =

$= 30 + 40 + 90 + \frac{20}{22} \times 77$

$= 30 + 40 + 90 + 70$

$= 230$ (optimal solution Max^m profit)

# BACKTRACKING

Backtracking uses brute force approach to solve problem. Brute force approach say that ~~for any problem~~ for any given problem generate all possible solution and pick up desired solution.

Backtracking uses DFS to generate state space tree.

Backtracking is a technique based on algorithm to solve problem. it uses recursive calling to find the solution by building a solution step by step increasing values with time. it removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems:-

i)- Decision Problem used to find a feasible solution of the problems.

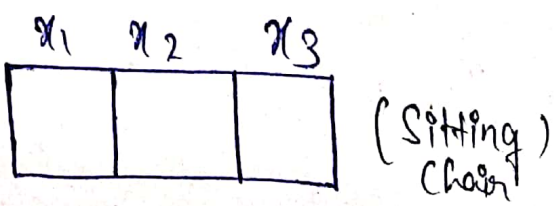
ii)- Optimization problem used to find the best solution that can be applied.

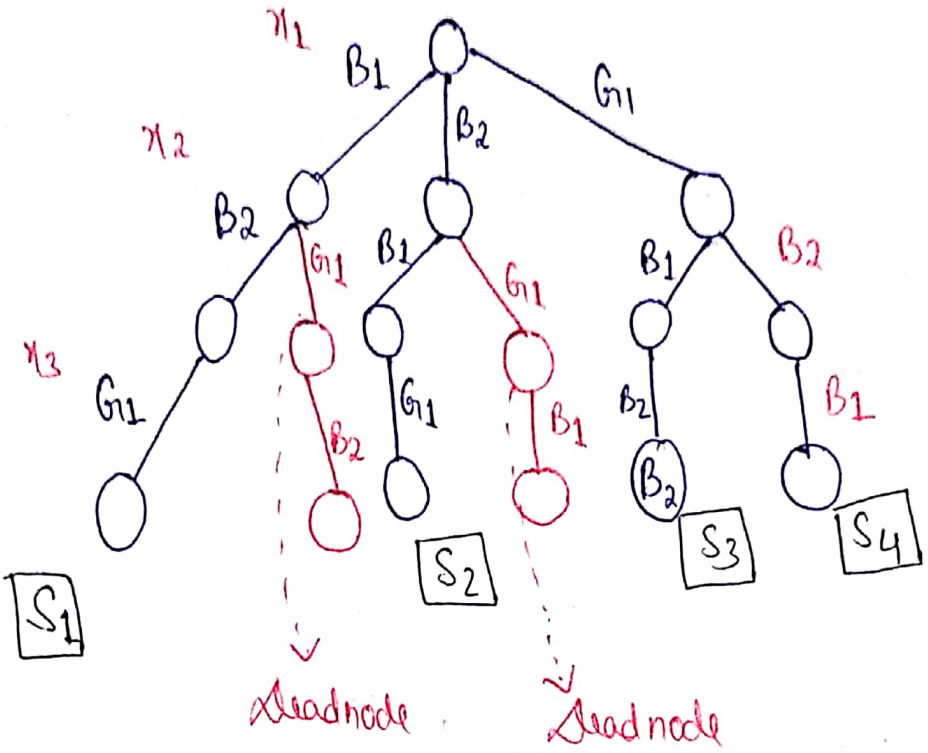
iii)- Enumeration problem used to find the set of all feasible solutions of the problem.

In Backtracking Problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example:- Suppose, there are three students one is Girl and two Boys are present.

$S = \{B_1, B_2, G_1\}$






Constant term → Explicit Constant
→ Implicit Constant

Applications of Backtracking :-

i)- N- queen Problem

ii)- Sum of Subset problem

iii)- Graph Coloring Backtracking

iv)- Hamiltonian cycle

Algorithm :-

Step 1 :- If Current position is goal, return success.

Step 2 :- else,

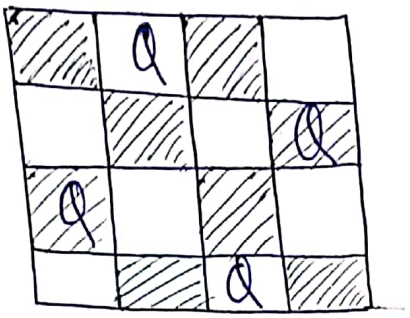
Step 3 :- if Current position is an end point, return failed.

Step 4 :- else, if Current position is not end point, explore and repeat above steps.

# N-Queen Problem

Let's use this Backtracking problem to find the solution to N-Queen Problem.

In N-queen problem, we are given an N×N chessboard and we have to place 'n' queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical, diagonal points in its way. Here, we will do 4-queen Problem.



Here, the Binary output for n-queen problem with 1's as queens to the positions are placed.

{0, 1, 0, 0}
{0, 0, 0, 1}
{1, 0, 0, 0}
{0, 0, 1, 0}

for solving 'n' queen problem, we will try placing queen into different positions of one row. and checks if it clashes with other queens. if Current positioning of queens if there are any two queens attacking each other. if they are attacking, we will backtrack to previous location of the queen and change its positions. And Check clash of queen again.

## Algorithm:-

Step 1:- Start from 1st position in the array.

Step 2:- Place queens in the board and check. Do,

2.1 :- After placing the queen, mark the position as a part of the solution and then recursively check if this will lead to a solution.

2.2 :- Now, if placing the queen doesn't lead to a solution and trackback and go to step (a) and place queens to other rows.

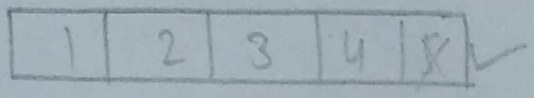
2.3 :- If placing queen returns a lead to solution return. TRUE.

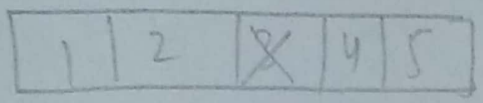
Step 3:- If all queens are placed return True.

Step 4:- If all rows are tried and no solution is found, return FALSE.

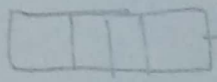
**Linked List:-** A Link List is a Collection of nodes, where each node is made up of a data element and a reference to the next node in the sequence

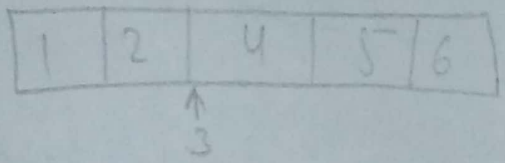
To overcome disadvantage of queue, array, stack.



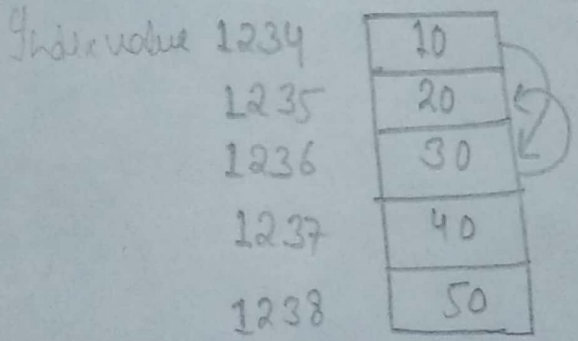



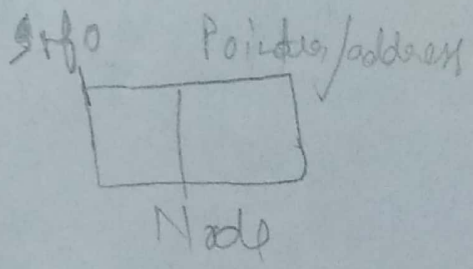

queue is full

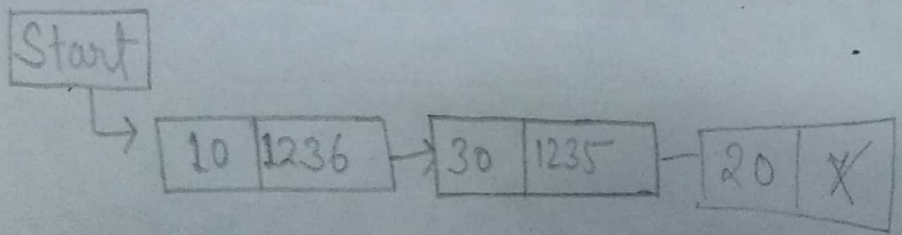


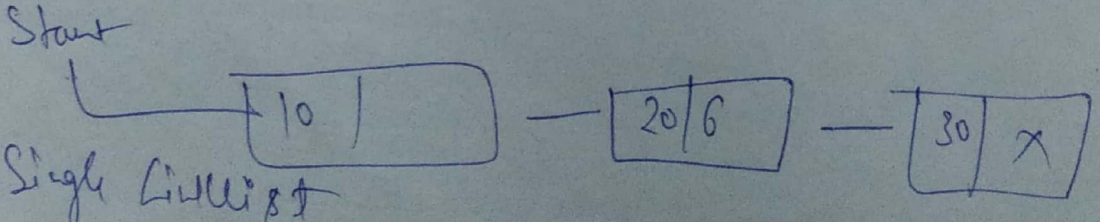




array



# Double Circular List