# Computer Algorithms and Complexity: Handnote Guide

Md. Jalal Uddin Chowdhury

# Computer Algorithms & Complexity

Date : .....

## Algorithm

An algorithm is a procedure that describes a set of instructions that must be carried out in a specific order to get the desired result.

OR

→ Step by Step procedure of Solving any Computational problem.

## Program

A program is an organized sequence of instructions a computer must follow to complete a task.

→ It refers to the code (written by programmers) for any program that follows the basic rules of the concerned programming language.

# What is the difference between Algorithm & Program?

⇒ For Software development life cycle (SDLC)

- Design → Algorithm
- Implementation → Program

⇒ Person

- Domain Knowledge → Algorithm
- Programmer → Program

⇒ Language

- Any language → Algorithm
- Programming language → Program

⇒ Dependency

- H/W & OS independent → Algorithm
- H/W & OS dependent → Program

⇒ Analysis

- Analyzing → Algorithm
- Testing → Program

## Priori Analysis & Posteriori Testing

⇒ Priori Analysis

- Algorithm
- Independent of language
- H/W Independent
- Time & Space function

⇒ Posteriori Testing

- Program
- Language Dependent
- H/W Dependent
- Watch time & Bytes.

Date : .....

## ☆ Characteristics of Algorithm

⇒ Input → 0 or more input
⇒ Output → at least one output
⇒ Definiteness → no ambiguity {like sqrt(-1) is not defined}
⇒ Finiteness → must has termination or stop point.
⇒ Effectiveness → do not use unnecessary statements

## ☆ Write an Algorithm

1. Algorithm swap (a, b)

```
{
    temp := a;
    a = b;
    b = temp;
}
```

2. Algorithm swap (a, b)

```
Begin
    temp = a ;
    a = b ;
    b = temp;
End
```

NB → We can use ":", "←" sign for variable declare.

## Analyze an Algorithm

* ⇒ Time function ---------- $f(n)$
* ⇒ Space function ---------- $s(n)$

⇒ Network based data tranfers

⇒ Power Consumption

⇒ CPU registers.

### # Time

→ Every Symbol Statement will take one unit of time.

```
Algorithm swap (a, b)
{
    temp = a ;  ----- 1
    a = b ;  ------ 1
    b = temp ;  ----- 1
}
```

* $f(n) = 1 + 1 + 1$

$= 3$

$= 3 \times n^0$

$= 3 \times 1$

$= 3$

$\Big\rangle$ $O(1)$ or $O(n^0)$

Date : .....

We know,

→ $x = 9*b + 6*5$ ---- Basically 4 unit

→ $x = 9*b + 6*5$ ---- We will assume 1.

→ We will go with the Basic now.

# Space

→ Each variable has taken one word. min 1

Algorithm Swap (a, b)

{

| | variable |
|---|---|
| temp = a; | a ---- 1 |
| a = b; | b ---- 1 |
| b = temp; | temp ---- 1 |

}

* $S(n) = 1+1+1$

$= 3$

$= 3 \times n^0$

$= 3 \times 1$

} $O(n^0)$ or $O(1)$

## ⇒ Frequency Count method

* For Time,

```
• Algorithm sum(A, n)
{
    S = 0          ---------- 1
    for (i=0; i<n; i++) --- 1 + (n+1) + n --- (n+1)
    {
        S = S + A[i]; --- n
    }
    return S; ---------- 1
}
```

$$f(n) = 1 + (n+1) + n + 1$$
$$= 2n + 3 \text{ ----- } O(n)$$

* For Space

```
• Algorithm Sum(A, n)
{
    S = 0;
    for(i=0; i<n; i++)
    {
        S = S + A[i];
    }
    return S;
}
```

| Space | |
|---|---|
| A | n |
| S | 1 |
| i | 1 |
| n | 1 |

$$* S(n) = n + 1 + 1 + 1$$
$$= n + 3 \text{ ----- } O(n)$$

# Add Two Matri

→ We

Step-1: n×

```
Algorithm
{
    for(i=0
    {
        for(j=
        {
            C[
        }
    }
}
```

Step-2:

```
Algorithm
{
    for(i=0
    {
        for(
        {
            C
        }
    }
```

Date : .....

# Add Two Matrix (For Time)

→ We assume that A and B matrix are n×n 2D array.

Step-1 :

```
Algorithm Sum (A, B, n)
{
    for(i=0; i<n; i++) ---------- n+1
    {
        for(j=0; j<n; j++) ------- n
        {
            C[i,j] = A[i,j] + B[i,j]; --- n
        }
    }
}
```

Step-2 :

```
Algorithm Sum (A, B, n)
{
    for(i=0; i<n; i++) ---------- (n+1)
    {
        for(j=0; j<n; j++) ------- n * (n+1)
        {
            C[i,j] = A[i,j] + B[i,j]; --- n * n
        }
    }
```

* $f(n) = 2n^2 + 2n + 1$ --- $O(n^2)$.

# Add Two Matrices (For Space)

Step-1:

```
Algorithm Sum (A, B, n)
{
    for(i=0; i<n ; i++)
    {
        for(j=0; j<n; j++)
        {
            c[i,j] = A[i,j] + B[i,j];
        }
    }
}
```

| Space | |
|---|---|
| A ----- | n×n |
| B ----- | n×n |
| C ----- | n×n |
| n ----- | 1 |
| i ----- | 1 |
| j ----- | 1 |

* $S(n) = 3n^2 + 3$

$---------- O(n^2)$

Date : .....

# Multiply Two Matrices (For time)

```
Algorithm Multiply (A, B, n) {
    for (i=0 ; i<n ; i++){ ------ (n+1)
        for (j=0 ; j<n ; j++){ ------ n*(n+1)
            C[i,j] = 0; ------ n*n
            for (k=0 ; k<n ; k++){ ---- n*n*(n+1)
                C[i,j] = C[i,j] + A[i,k] * B[k,j]; ---- n*n*n
            }
        }
    }
```

$\therefore f(n) = 2n^3 + 3n^2 + 2n + 1$ ----- $O(n^3)$

## For Space

A --- n×n
B --- n×n
C --- n×n
n --- 1
i --- 1
j --- 1
k --- 1

$S(n) = 3n^2 + 4$ ----- $O(n^2)$

Date :

# Examples for Time Complexity

1. 
```
for (i=0; i<n ; i+=2)
{
    Statement; ---- n/2
}
```

$\Rightarrow O(n)$

2. 
```
for (i=1 ; i<n ; i+=70)
{
    Statement; ------ n/70.
}
```

$\Rightarrow O(n)$

3. 
```
for (i=0; i<n ; i++){
    for (j=0; j<i; j++){
        Statement;
    }
}
```

| i | j | no. of times executed |
|---|---|---|
| 0 | 0 × | 0 |
| 1 | 0<br>1 × | 1 |
| 2 | 0<br>1<br>2 × | 2 |
| 3 | 0<br>1<br>2<br>3 × | 3 |

n ---- n

$= 0+1+2+3+\cdots+n$

$= n*(n+1)/2$ ------ $O(n^2)$

4.

```
p=0;
for (i=1; p<=n; i++)
{
    p= p+i;
}
```

| i | p |
|---|---|
| 1 | 0+1 = 1 |
| 2 | 1+2 = 3 |
| 3 | 1+2+3 = 6 |
| 4 | 1+2+3+4 = |
| ⋮ | |
| K | $1+2+3+4\ldots+K = K(K+1)/2$ |

Assume,

$p > n$

$\frac{K(K+1)}{2} > n$ $\quad \because p = K(K+1)/2$

$K^2 > n$

$K > \sqrt{n}$ --- $O(\sqrt{n})$

Date :

5. For (i=1; i<n ; i= i * 2)
{ Statement;
}

| i |
|---|
| 1 |
| $1 \times 2 = 2^1$ |
| $2^1 \times 2 = 4 = 2^2$ |
| $2^2 \times 2 = 8 = 2^3$ |
| ⋮ |
| $2^K$ |

Assume,

$i >= n$

$\therefore i = 2^K$

$2^K = n$

$K = \log_2 n$ --- $O(\log_2 n)$

6. For (i=0; i<n ; i++){
{ Statement; ⟶ n
}
For (j=0; j<n ; j++)
{ Statement; ⟶ n
}

$f(n) = 2n$

$O(n)$

7.
```
p = 0
for (i=1 ; i<n ; i = i*2)
{
        p++ ;          ----------> log n    p = log n
}
for (j=1 ; j<p ; j = j*2)
{       statement;  ----------> log p = log log n
}
```

$O(\log \log n)$

8.
```
i = 0 ;
while (i<n)  ----------> n+1
{   statement ; ----> n
    i++       ; ----> n
}
```

$f(n) = 3n+1$ --- $O(n)$

9.
```
a = 1
while (a<c){
        statement ;
        a = a*2 ;
}
```

$f(n) = ?$

Date : .....

10.
```
i = n;
while (i > 1)
{
    Statement;
    i = i/2;
}
```

| i |
|---|
| n |
| n/2 |
| $n/2^2$ |
| $n/2^3$ |
| ⋮ |
| $n/2^K$ |

Terminate,
$i <= 1$
$\therefore i = n/2^K$

$\frac{n}{2^K} = 1$

$\Rightarrow 2^K = n$

$\Rightarrow K = \log_2 n$

$\therefore O(\log n)$

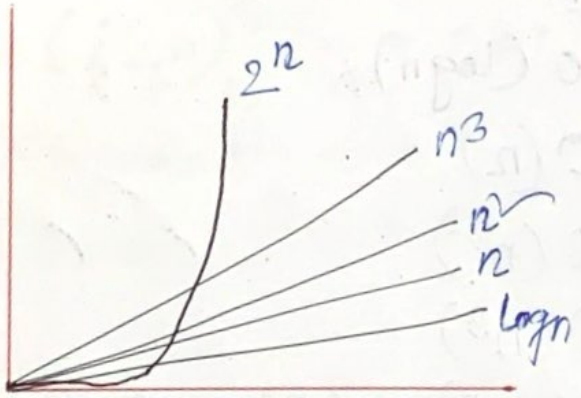
## Types of Time Function

- Constant - $O(1)$
- Logarithmic - $O(\log n)$
- Linear - $O(n)$
- Quadratic - $O(n^2)$
- Cubic - $O(n^3)$
- Exponential - $O(2^n)$, $O(3^n)$, $O(n^n)$.

## ☆ Comparison of time function

- $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \dots < 2^n < 3^n$

| $\log n$ | $n$ | $n^2$ | $2^n$ |
|---|---|---|---|
| 0 | 1 | 1 | 2 |
| 1 | 2 | 4 | 4 |
| 2 | 4 | 16 | 16 |
| 3 | 8 | 65 | 256 |
| 3.1 | 9 | 81 | 512 |




* $n^{1200} < 2^n$ ?

Date : .....

# Asymptotic Notation

- O – Big O ⇒ Upper bound
  Ω – Big Omega ⇒ Lower Bound
  Θ – Big Theta ⇒ Average Bound.

Date : .....

# Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

## Algorithm

To sort an array of size n in ascending order:

Step-1 : Iterate from array[1] to array[n] over the array. If it is the first element, assume that it is already sorted. return 1.

Step-2 : pick the next element, and store it separately in a key.

Step-3 : Compare the current element (key) to its predecessor.

Step-4 : If the key element is smaller than its predecessor, compare it to the elements before.

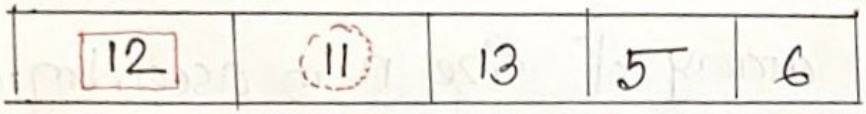
Step-5 : Move the greater elements one position up to make space for the swapped element.

Step-6 : Repeat until the array is sorted.

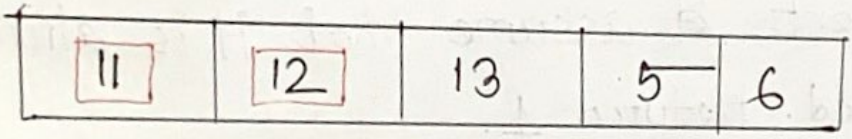
Example : Suppose an array A contains 5 elements as follows : {12, 11, 13, 5, 6}

## First pass

Initially, the first two elements of the array are compared in insertion sort.

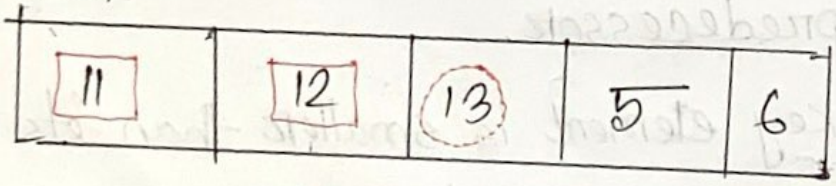



Here, 12 is greater than 11 hence they are not not in ascending order.

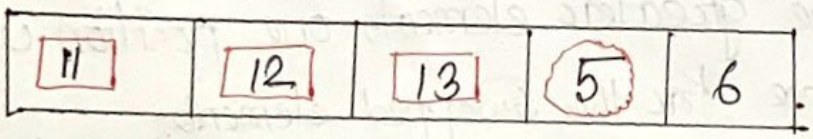



## Second pass:

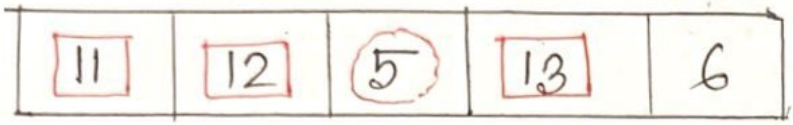
Now move to the next two elements and compare them.

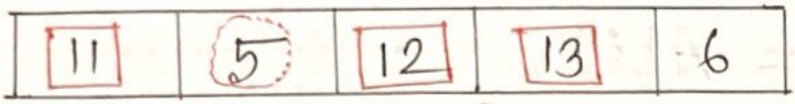


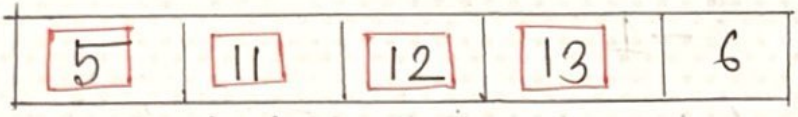

## Third Pass



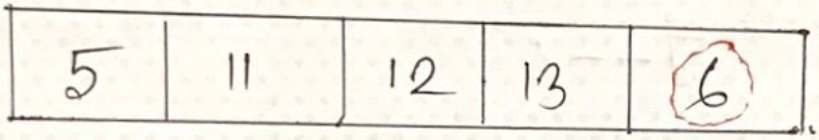
Date : .....

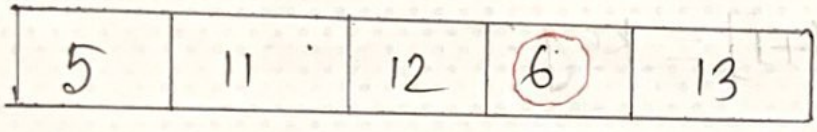


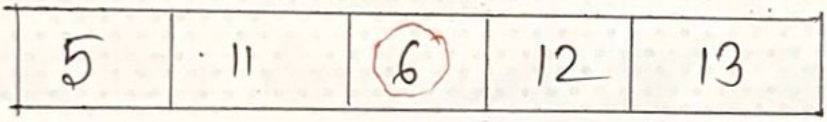



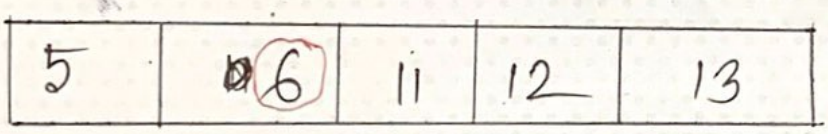




## Fourth pass




Finally, the array is completly sorted.

## Practice Problem:

Array : {12, 30, 24, 9, 33, 16}

pseudocode

Insertion-Sort (A)

```
for i = 1 to A.length
    key = A[i]
    j = i-1
    while (j >= 1 && A[j] > key) {
        A[j+1] = A[j]
        j--;
    }
    A[j+1] = key;
```

Date : .....

# Selection Sort

Selection Sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

## # Algorithm

The Selection sort algorithm is as follows:

Step-1 : Set MIN to location 0.

Step-2 : Search the minimum element in the list.

Step-3 : Swap with value at location MIN.

Step-4 : Increment MIN to point to next element.

Step-5 : Repeat until the list is sorted.

Example: arr[] = {4, 12, 7, 3, 9}

For the first position in the ~~Store~~ sorted list, the whole list is scan sequentially,

First pass:

(4), 12, 7, 3, 9

(4), 12, 7, 3, 9

(4), 12, 7, 3, 9

(3), 12, 7, 4, 9

Second pass:

[3], (12), 7, 4, 9

[3], (7), 12, 4, 9

[3], (4), 12, 7, 9

Third Pass:

[3], [4], (12), 7, 9

[3], [4], (7), 12, 9

Fourth pass:

[3], [4], [7], (12), 9

[3], [4], [7], [9], (12)

[3], [4], [7], [9], [12]

Date : .....

# Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into small subarrays, sorting each subarray and then merging the sorted subarrays back together to form the final sorted array.

Algorithm

Step-1 : If it is only one element in the list, consider it already sorted, so return.

Step-2 : Divide the list recursively into two halves until it can no more be devided.

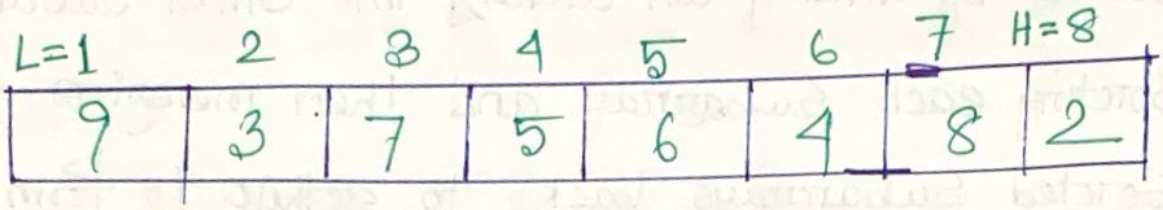
Step-3 : Merge the smaller lists into new list in sorted order.

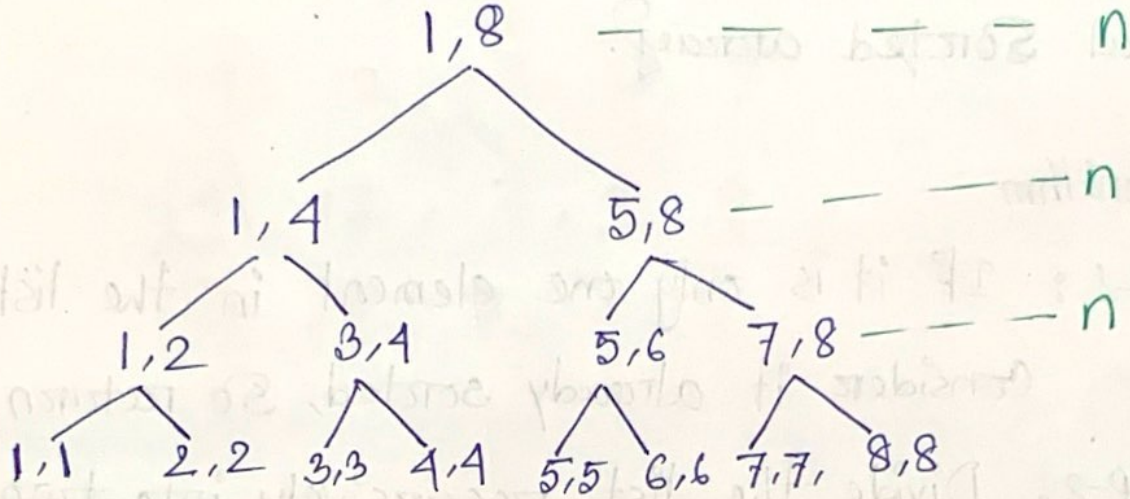
# There is two process of merge sort method

- Merge Sort → recursive method
- 2-way merge sort → iterative method

# Merge Sort - Recursive

Sort a single array using merge sort. Showing tracing -




⇒ Height of the recursion tree is $\log n$ | $\log 8 = 3$

⇒ In each level there are always n elements to merge

⇒ So time complexity is $O(n \times \log n)$

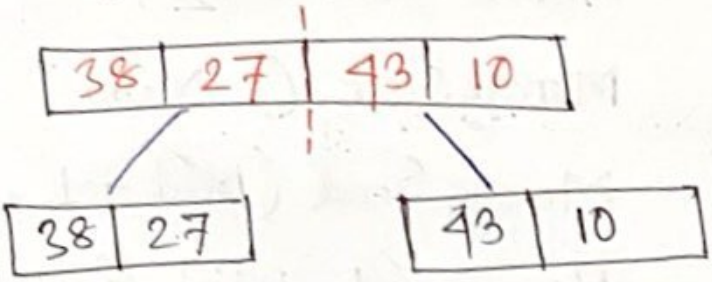
Date : .....

```
Algorithm mergeSort (L, H){
  if (L < H){
      Mid = (L+H)/2;
      MergeSort (L, Mid);
      MergeSort (Mid+1, H);
      Merge (L, Mid, H);
  }
```

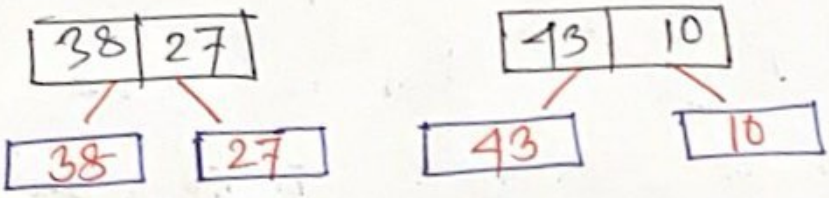
Example: arr[] = {38, 27, 43, 10}

Initially divide the array into two equal halves:

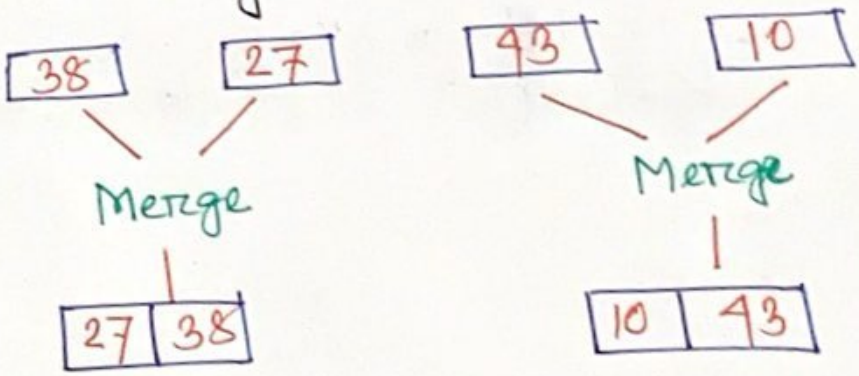
Step-1: Splitting the array into two equal halves



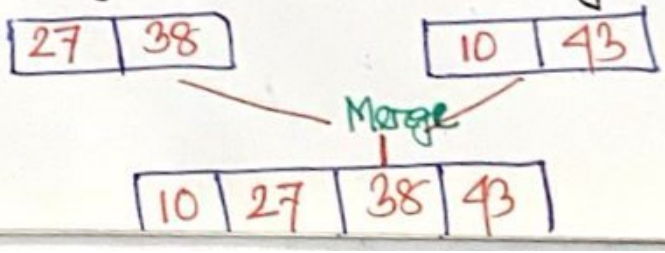

Step-2: Splitting the subarrays into two halves




Step-3: Merging unit length cells into sorted subarrays

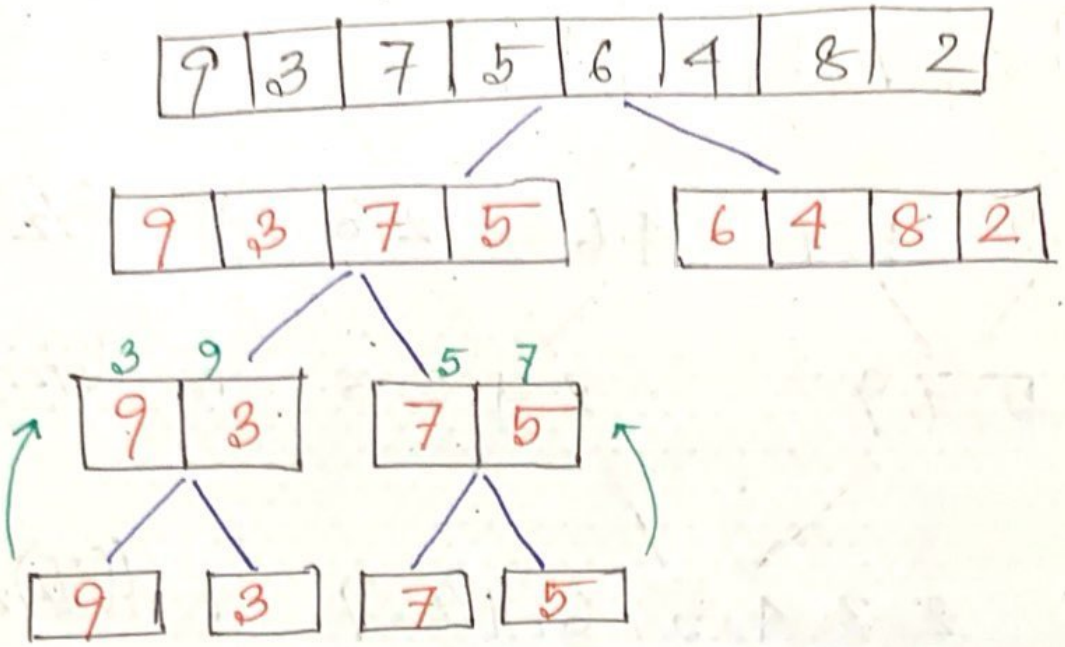



Step-4: Merging sorted subarrays into the sorted a



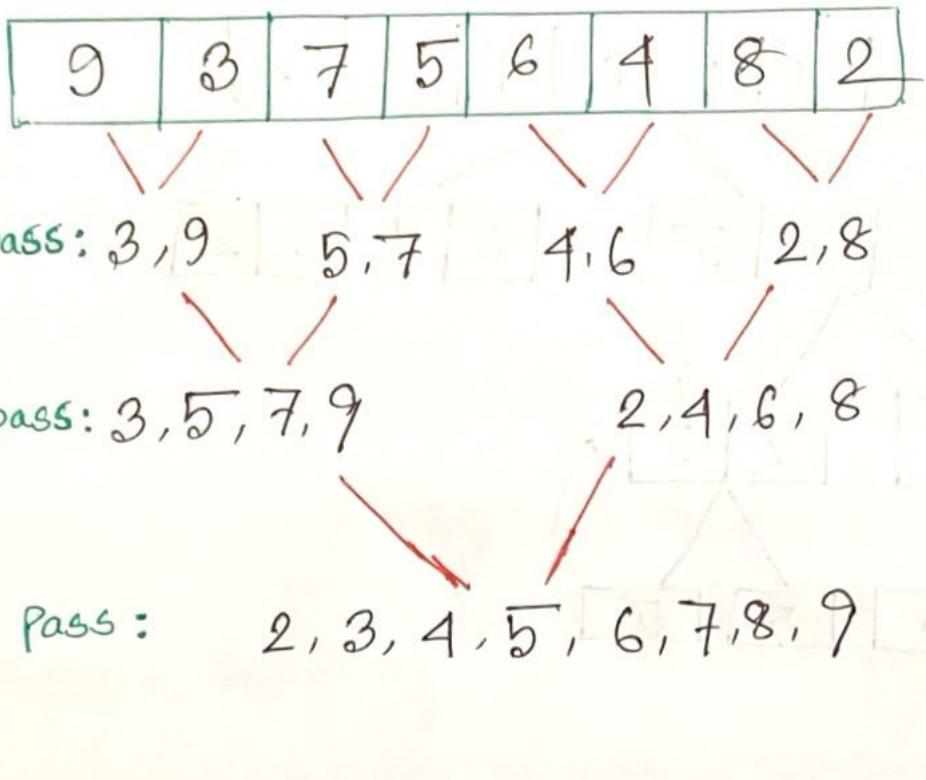
Date :

Example - 2 :



# * 2-Way Merge Sort – Iterative method

⇒ sort a single array using merge sort.




$\therefore \frac{8}{2^3} = 1$

$\Rightarrow 8 = 2^3$

$\Rightarrow \log_2 8 = 3$

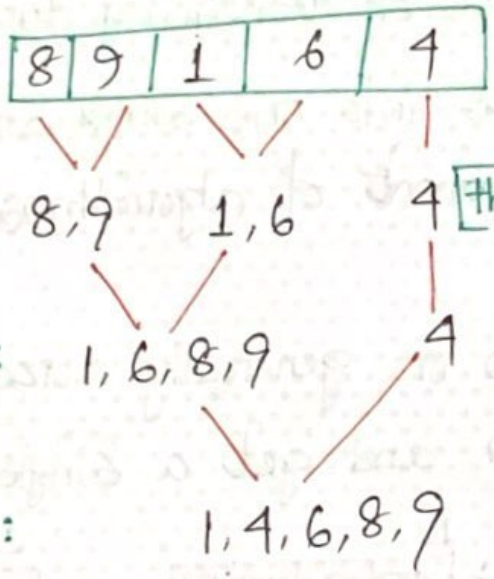
No. of passes $\log n$

- No of passes $= \log_2 n$
- In each pass we always merge n elements.
- So time complexity is $O(n \log n)$

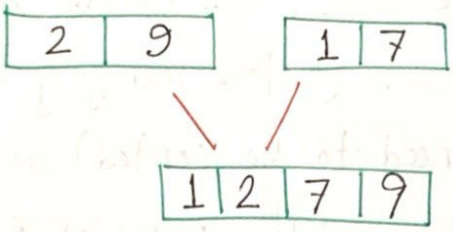
Date : .....

In 2 way merge sort, we devide the array in two elements each (before merging every two elements parts need to be sorted) so, it also start from single element and merge them.

Example-2: arr = {8, 9, 1, 6, 4}



## Merge Method

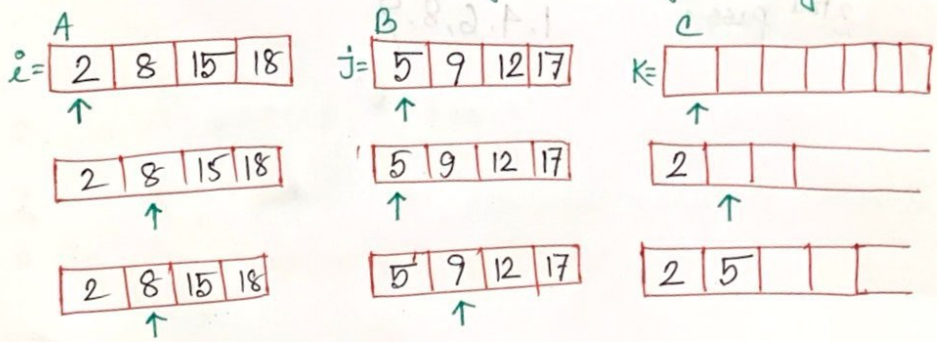



?? How it works.

Let's start,

Merging is a process of combining two or more types of structures into one single structure, which is important component of algorithms such as merge sort.

→ If we have two or generally more arrays, we can merge them and get a single array.



# Pseudocode

```
Algorithm Merge(A, B, m, n){
    i=1, j=1, K=1;
    While (i<=m && j<=n){
        if (A[i] < B[j]){
            C[K] = A[i];
            K++;
            i++;
        }
        else{
            C[K] = B[j];
            K++;
            J++;
        }
    }
    for ( ; i<=m ; i++){
        C[K] = A[i];
        K++;
    }
    For ( ; j<=n ; j++){
        C[K] = B[j];
        K++;
    }
}
```

Date : .....

## Greedy Algorithm

Greedy is an algorithmic paradigm that builds up a solutions piece by piece, always choosing the next piece that offers ~~that~~ the most obvious and immediate benefit.

⇒ The algorithm never reverses the earliest decision even if the choice is wrong. It works in a top-down approach.

⇒ Used for solving optimization problem.

⇒ Optimization Problem –
- Minimum result and maximum result.

⇒ A problem P which States we have to go Sylhet to Dhaka.
- Bicycle – Solution
- CNG – Solution
- Train – Feasible Solution
- Flight – Feasible Solution

" Constraints – Journey must has to end within 6 hours

⇒ Solutions which satisfying the condition given in the problem is called feasible solution.

⇒ If I want to do the journey with minimum cost, now it is called a ~~minimum~~ minimization problem.

⇒ By train the cost is minimum, then will be the optimal solution. So, there can be many solutions.

⇒ There can be many feasible solutions also. But there can be only one optimal solution for any problem.

⇒ Strategies for solving optimization problem are:

→ Greedy Method
→ Dynamic Programming
→ Branch and Bound Method

∴ Approach is different for all three.

## * Greed Algorithm Pseudocode

```
Algorithm Greedy (a, n)
{
    for i = 1 to n do
    {
        x = select(a);
        IF (feasible(x)) then
        {
            Solution = Solution + x;
        }
    }
}
```

Date : .....

# Knapsack Problem

$n = 7$, $m = 15$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Objects, O = | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Profit, P = | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| Weights, W = | 2 | 3 | 5 | 7 | 1 | 4 | 1 |

→ There are 7 objects. Each object has some profit associated with it.

→ A Bag/Knapsack is given which can carry maximum $m = 15$ kg weights. Load this bag and transfer it to market and make highest profit.

Our target is to fill in such a way that the profit is maximized.

## # Solution Process:

→ Fraction can be taken

- Take the highest profit element and fill it.
- OR fill with smaller things so that we can have more and more things on the bags and more profit.

But we should select with

- highest (profit/weight)

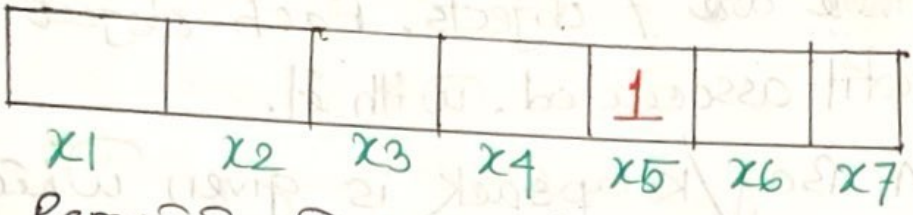
That means we have to calculate per kg profit for all objects.

⇒ n = 7, m = 15

| Objects (O) | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Profit (P) | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| Weights (W) | 2 | 3 | 5 | 7 | 1 | 4 | 1 |
| P/W | 5 | 1.6 | 3 | 1 | 6 | 4.5 | 3 |

Step-1:

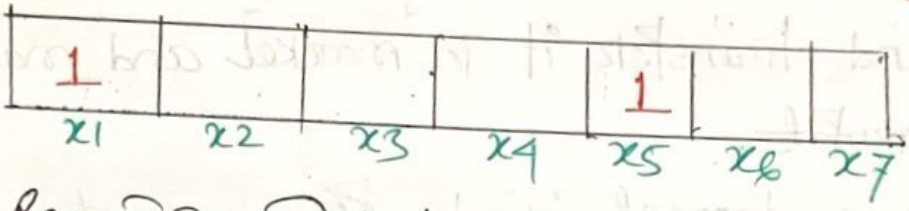
$x$ =




Remaining Weights = 15 − 1 = 14 kg

Step-2:

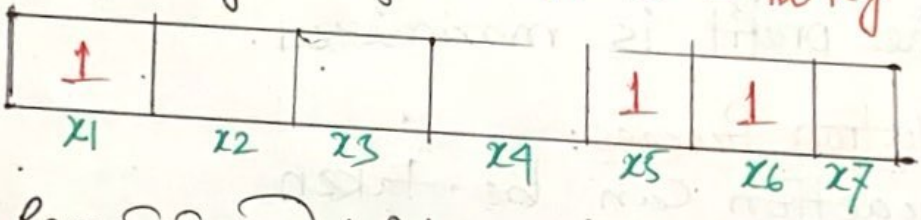
$x$ =




Remaining Weights = 14 − 2 = 12 kg

Step-3:

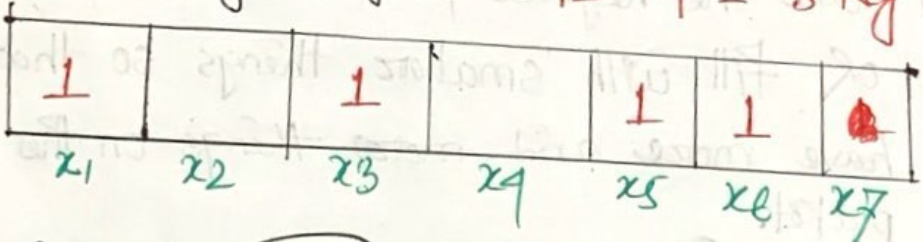
$x$ =




Remaining Weights = 12 − 4 = 8 kg

Step-4:

$x$ =

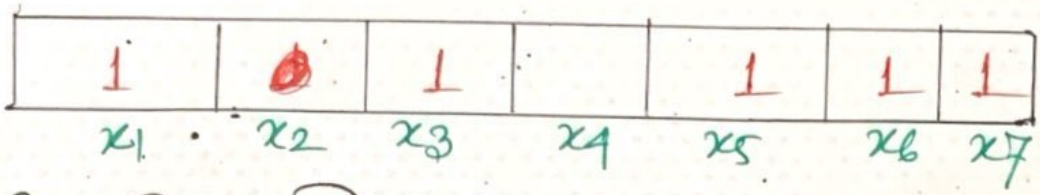



Remaining Weights = 8 − 5

= 3 kg

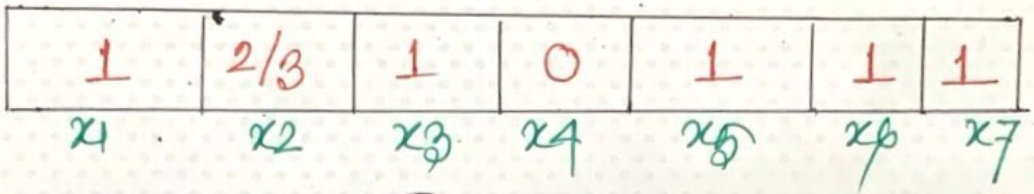
Date : .....

Step-5 :




Remaining Weights = 3 - 1 = 2 kg

Step-6 :




Remaining Weights = 2 - 2 = 0 kg

$\therefore \sum x_i w_i = 1\times2 + 2/3 \times 3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$

$= 15$

So, ~~$\sum x_i w_i$~~ $\sum x_i W_i <= m = 15$.

For Profit Calculation

$\sum x_i P_i = 1\times10 + 2/3 \times 5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$

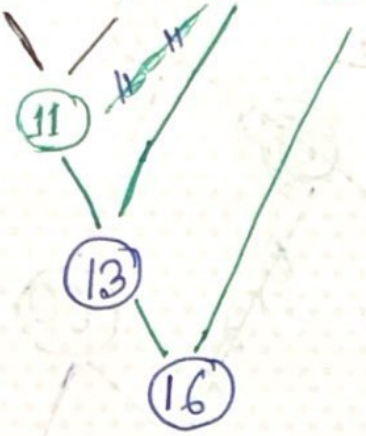
$= 55.33$

Date : .....

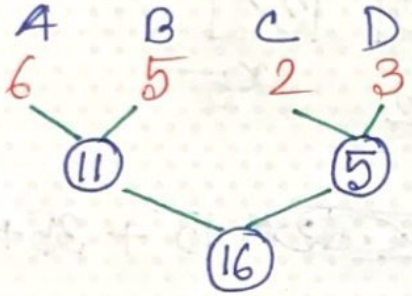
# Optimal Merge Pattern

List → A B C D

Sizes → 6 5 2 3



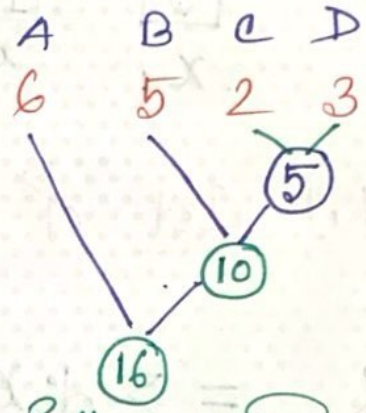

First Pattern Solution: Total Steps = 11 + 3 + 16 = 40




Second Pattern Solution:

Total Steps = 11 + 5 + 16

= 32




Third Pattern Solution:

Total Steps = 5 + 10 + 16

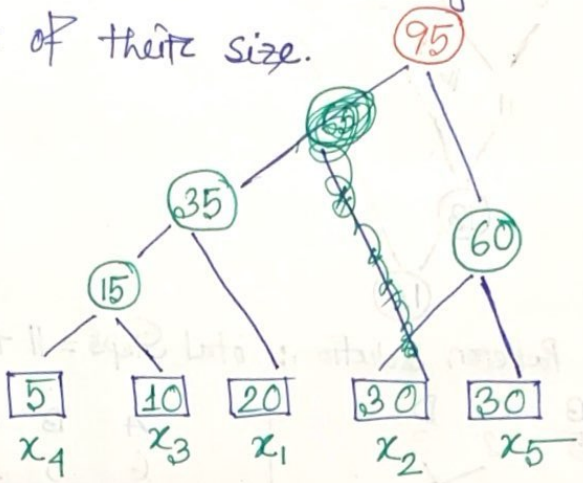
= 31.

Out of all patterns, last one is the optimal solution. So we have found that always select two smaller list for merging.

Lists → $x_1$ $x_2$ $x_3$ $x_4$ $x_5$

Sizes → 20 30 10 5 30

Now, sort the list according to the accending order of their size.




Total Cost of merging = 15 + 35 + 60 + 95

= 205.

OR $= \sum d_i x_i$

where,

$d_i$ = Distance from root to i-th leaf

$x_i$ = Size of i-th list.

$= 3 \times 5 + 3 \times 10 + 2 \times 20 + 2 \times 30 + 2 \times 30$

$= 205.$

## Huffman Coding

Huffman Coding is
algorithm. It is
encoding. It is
→ Compression
transmission.
→ Cost is repres

# Cost Measurement

→ Normal Messa
→ Fixed Size E
→ Variable Size

Date : .....

# Huffman Coding

Huffman Coding is a lossless data Compression algorithm. It is also known as data compression encoding. It is widely used in image compression.

→ Compression need for reducing the cost of transmission.

→ Cost is represented by bits.

## # Cost Measurement methods :

→ Normal Message Sending Cost.

→ Fixed Size Encoding

→ Variable Size Encoding – Huffman Coding

# Normal Message Sending Cost

Message: BCCABBDDAECCBBAEDDCC

∴ Message Length = 20

Normal message has to be sent using ASCII Code. So, ASCII Codes are 8-bits.

* Total Unique representations = $2^8$ = 256 chars

Here, in this message there exists only 5 characters.

→ A, B, C, D, E.

| Characters | ASCII Value | ASCII Code |
|---|---|---|
| A | 65 | 01000001 |
| B | 66 | 01000010 |
| C | 67 | 01000011 |
| D | 68 | 01000100 |
| E | 69 | 01000101 |

∴ Total bits = 8 × 20 = 160 bits

Message Sending Without Encoding.

Date : .....

# Fixed Size Encoding

Message :

→ If we have 1 position to fill with 0/1, we can represent 2 characters with 0 and 1.

→ If we have 3 position to fill with 0/1, we can represent 8 characters with 000, 001, 010, 011, 100, 101, 110, 111.

| Characters | Count | Code |
|---|---|---|
| A | 3 | 000 |
| B | 5 | 001 |
| C | 6 | 010 |
| D | 4 | 011 |
| E | 2 | 100 |

Message Length = 20.

Total message bits = 20 × 3 = 60 bits

For decoding, with the message → also send the mapping table.

# 8bit-3bit mapping

character ASCII value = 8 bits, Code = 3 bits

So, mapping table size = 5×8 + 5×3 = 55 bits

Total bits passed = 60+55 = 115.

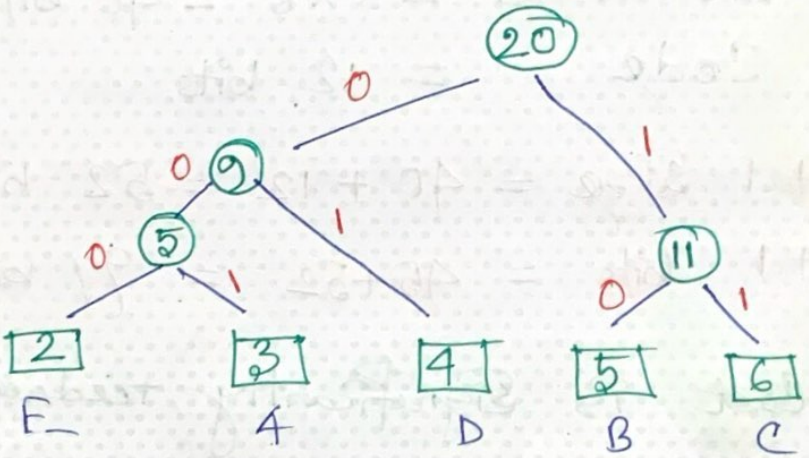
→ Sending Cost

So, cost/size is reduced from the normal Message

Date : .....

Message :

| Characters | Count | Code |
|---|---|---|
| A | 3 | 001 |
| B | 5 | 10 |
| C | 6 | 11 |
| D | 4 | 01 |
| E | 2 | 000 |
| | 20 | = 12 bits |



| Characters | Count | Code | |
|---|---|---|---|
| | | | $3 \times 3 = 9$ bits |
| | | | $5 \times 2 = 10$ |
| | | | $6 \times 2 = 12$ |
| | | | $4 \times 2 = 8$ |
| | | | $2 \times 3 = 6$ |
| | | | Total = 45 bits |

Tota message bits = 45 bits.

Now we also send mapping Table,

Characters $= 5 \times 8 = 40$ bits

Code $= 12$ bits

Total Size $= 40 + 12 = 52$ bits.

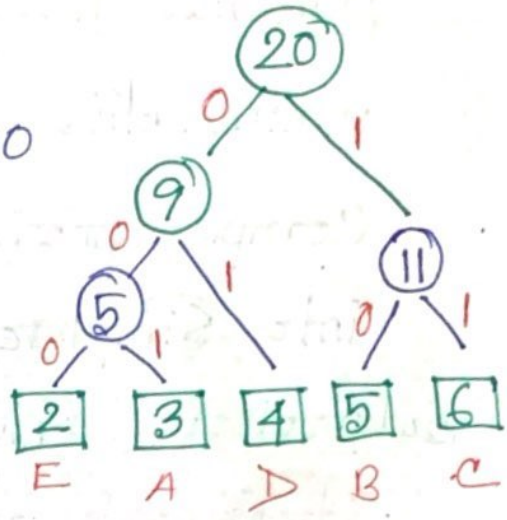
Total bits $= 45 + 52 = 97$ bits.

∴ Cost is significantly reduced.

Date : .....

For Decoding

∴ If the Code = 01101111000




Soln:

We will decode from left to right. and

Start from root-end to leaf.

01 = D
10 = B
11 = C
11 = C
000 = E

—— × ——

∴ If the Code = 0011101

Soln:

001 = A
11 = C
101 = D

## Dynamic Programming (DP)

Dynamic Programming is a method used in mathematics and Computer Science to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for wide range of problems.

### # Characteristics of Dynamic Programming

A problem that can be solved using Dynamic Programming must follow the below mentioned properties:

- → Optimal Structure Property
- → Overlapping Subproblems.

Date : .....

# Major Differences between Greedy Method and DP

| Feature | Greedy Method (GM) | Dynamic Programming |
|---|---|---|
| Feasibility | In GM, We follow the predefined procedure for getting the result. | In DP, We will try to find all possible feasible solutions then find the optimal one |
| optimality | In GM, sometimes there is no such guarantee of getting optimal solution. | It is guaranteed that DP will generate an optimal solution. |
| Recursion | A GM follows the problem solving heuristic of making the locally optimal choice at each stage. | A DP is an algorithmic technique which is based on a recurrent that uses some previously calculated states. |
| Memoization | It is more efficient in terms of memory as it never look back or revise previous choices. | It requires DP table for Memoization and it increases it's memory complexity. |
| Example | Fractional Knapsack | 0/1 Knapsack Problem |

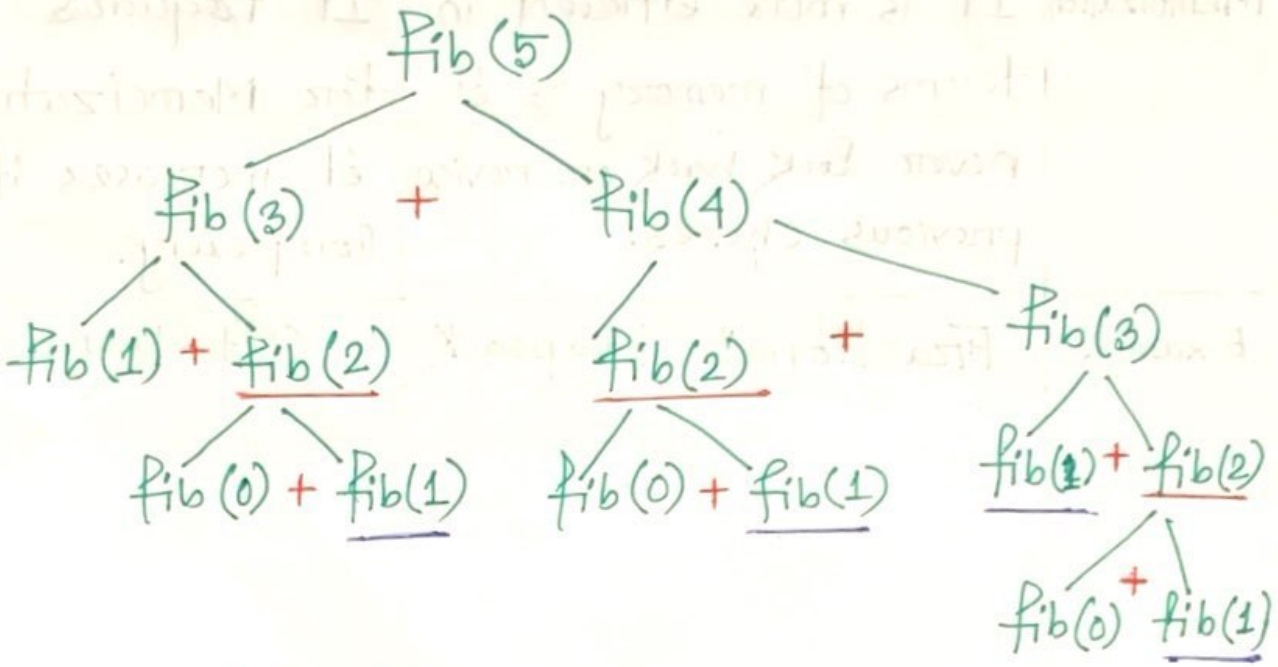
# Nth Fibonacci Number

The Fibonacci numbers are the numbers in the following integer sequence : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

```
int fib(int n)
{
    if(n<=1)
        return n;
    return fib(n-2)+fib(n-1);
}
```

$$fib(n) = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ fib(n-2)+fib(n-1), & \text{if } n>1 \end{cases}$$

# Recursion Tracing Tree → for n = 5




∴ Total Call = 15

# Complexity: $O(2^n)$ [Recurrance Relation → Check Slide]

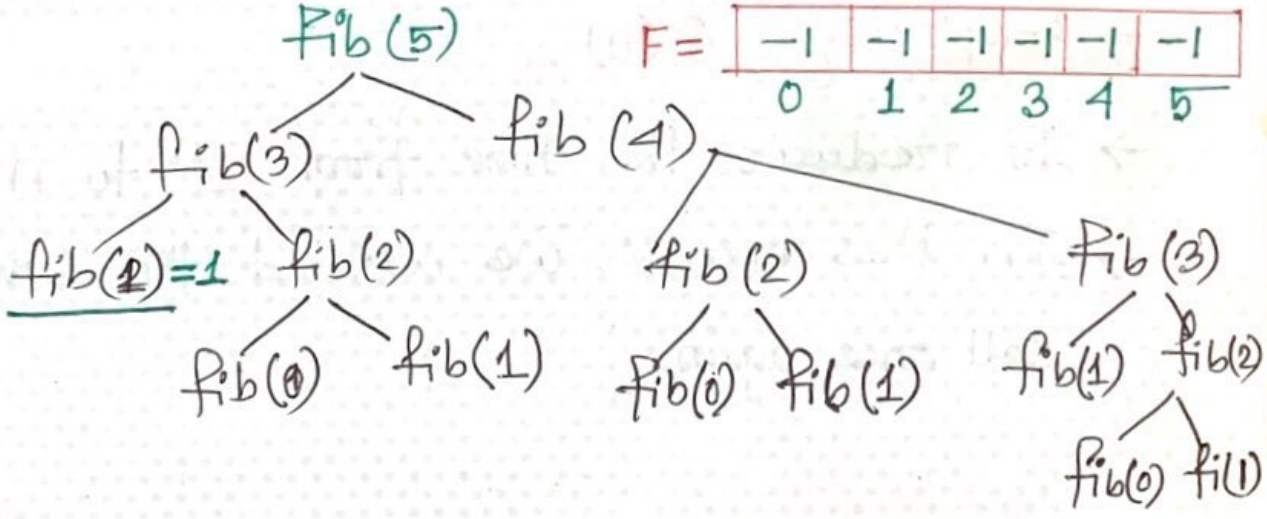
Date : .....

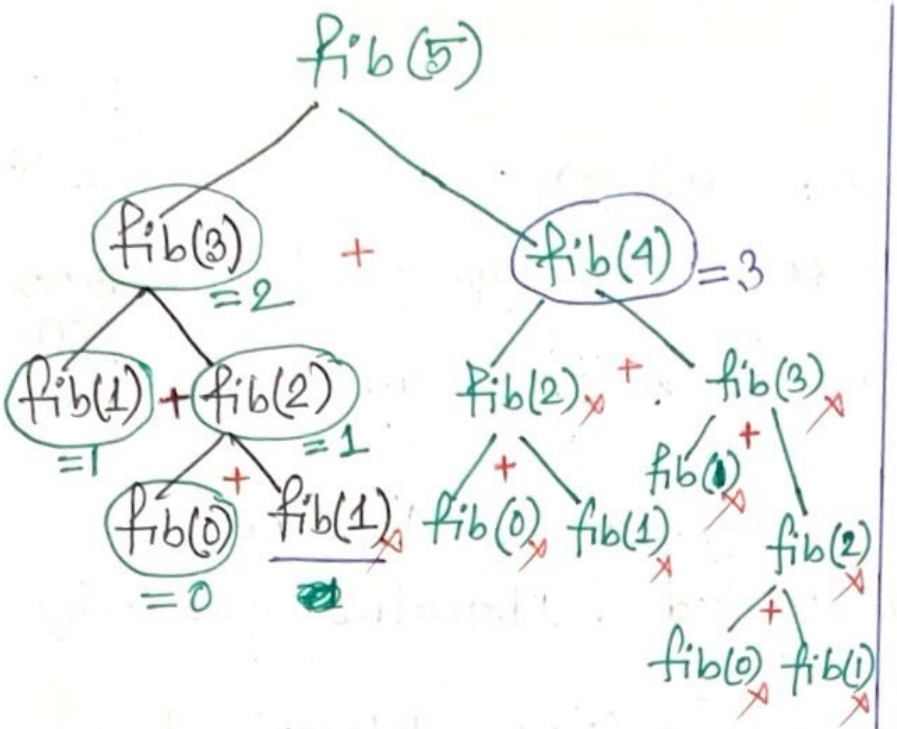
→ Lots of recurrence call for of same fib function
→ It takes more and more time for this reason.
⇒ In DP, we can store the result using two methods:
- Memoization (Recursive Approach)
- Tabulation Method (Iterative approach)

## # Dynamic Programming (DP) – Memoization.

→ Top-down approach
→ Caches the results of function calls
→ Well-suitable Suited for problems with a relatively small sets of inputs.
→ Initially take a global array of size n+1
→ Initialize all the index with -1

Now let's start the recursion tracing tree.






F= | -1 | 1 | -1 | -1 | -1 | -1 |

F= | 0 | 1 | -1 | -1 | -1 | -1 |

F= | 0 | 1 | -1 | -1 | -1 | -1 |

F= | 0 | 1 | 1 | -1 | -1 | -1 |

F= | 0 | 1 | 1 | 2 | -1 | -1 |

F= | 0 | 1 | 1 | 2 | -1 | -1 |

F= | 0 | 1 | 1 | 2 | 3 | -1 |

F= | 0 | 1 | 1 | 2 | 3 | 5 |

→ For fib(5), Now there are 6 calls.

→ So for fib(n) there are n+1 calles

→ Complexity : O(n)

→ It reduces the time from $2^n$ to n

→ In this process, We avoided the same recursion call once again.

Date : .....

# DP – Tabulation

```
int fib (int n) {
    if (n<=1)
        return n;
    F[0] = 0, F[1] = 1;
    for(int i=2; i<=n ; i++)
    {
        F[i] = F[i-2] + F[i-1];
    }
    return F[n];
}
```

→ Bottom up Approach

→ Take an array of size n+1

→ Let us iterate for n = 5

Firstly, F[0] = 0, F[1] = 1 will be executed.

F =

| 0 | 1 | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

F =

| 0 | 1 | 1 | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

F =

| 0 | 1 | 1 | 2 | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

F =

| 0 | 1 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

F =

| 0 | 1 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

Date : .....

## 0/1 Knapsack Problem

The 0/1 Knapsack problem means that the items are either completly or no items are filled in a Knapsack.

For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not visible); we have to pick the 2kg item completly. This is a 0/1 knapsack problem in which either we pick the item completly or we will pick that item.

\# $n = 4$ (objects)

$m = 8$ (The knapsack can carry highest 8kg)

$P = \{1, 2, 5, 6\}$

$W = \{2, 3, 4, 5\}$

$x_i = 0/1 \quad x = \{1, 0, 0 --\}$

0 0 0 0 = 1 = $2^0$
1 1 1 1
0 0 0 1 = $O(2^n)$
1 0 0 0
---
0 1 1 0

$\therefore \max \sum p_i x_i \leftarrow$ Profit

$\sum w_i x_i \leq m \leftarrow$ Weight

V → Weights

| P | W |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 5 | 4 |
| 6 | 5 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 |
| 3 | 0 | 0 | 1 | 2 | 5 | 5 | 6 | 7 | 7 |
| 4 | 0 | 0 | 1 | 2 | 5 | 6 | 6 | 7 | 8 |

Formula,

$$V[i,w] = \max\{V[i-1,w], V[i-1, w-w[i]]+P[i]\}$$

For $V[4,1] = \max\{V[3,1], V[3,1-5]+6\}$

$= \max\{$ undefined, undefined $\}$

$= 0.$

$V[4,7] = \max\{V[3,7], V[3,7-5]+6\}$

$= \max\{7, 1+6\}$

$= 7$

$x = \{x_1, x_2, x_3, x_4\}$

$\{\quad, 1\}$

Remaining Profit $= 8-6 = 2$

Date : .....

$x = \{x_1, x_2, x_3, x_4\}$

$= \{ \quad 0, 1\}$

$\{ \quad 1, 0, 1\}$

Remaining Profit, $= 2 - 2 = 0$

$x = \{x_1, x_2, x_3, x_4\}$

$= \{0, 1, 0, 1\}$

So, we will select 2nd & 4th object for maximum Profit.

## Longest Common Subsequence (LCS)

The LCS problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequence).

# Example-1 :

String 1 : abcdefghij

String 2 : cdgi

∴ Common Subsequences are :

c, d, g, i, cd, cdg, dgi, gi, cdgi.

∴ Longest Common Subsequence : cdgi.

# Example-2 :

String 1: abcdefghijK

String 2: ecdgi

Interselection is not allowed.

String 1: abcdefghijK

String 2: ecdgi

Interselection is not allowed.

Date : .....

∴ Common Subsequents (CS) are : egi, cdgi.

∴ LCS - cdgi.

# Example-3 :

String 1 : abdace

String 2 : babce

∴ Common Subsequents (CS) are : bace, abce.

∴ (bace) and (abce) are their longest Common Subsequences.

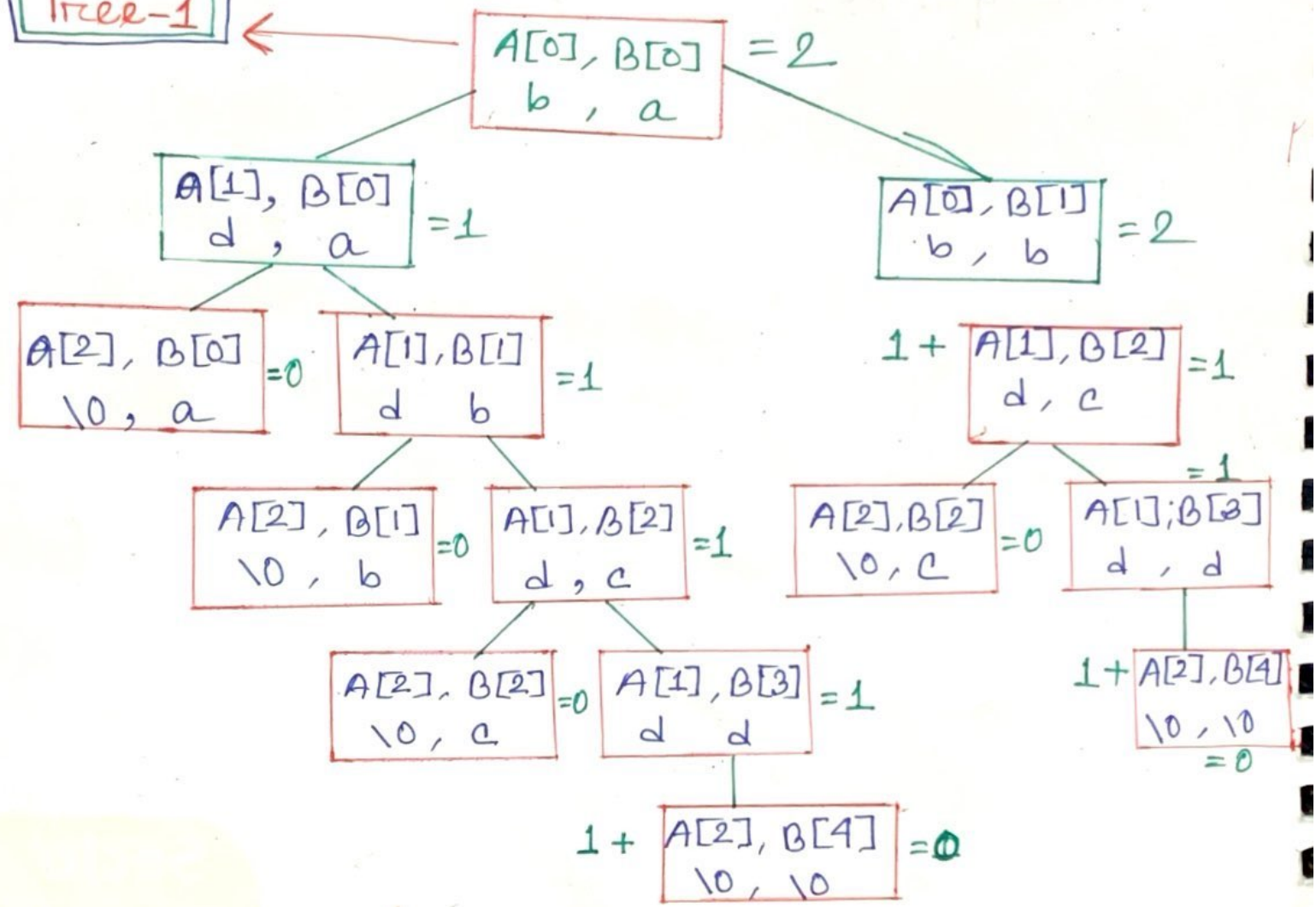
# Longest Common Subsequence using Recursion.

A = | b | d | \0 |
0 1 2

B = | a | b | c | d | \0 |
0 1 2 3 4

```
int LCS (int i, int j){
    if (A[i] == '\0' || B[j] == '\0') return 0;
    else if (A[i] == B[j]){
        return 1 + LCS (i+1, j+1);
    }
    else {
        return max (LCS (i+1, j), LCS (i, j+1));
    }
}
```

Tree-1

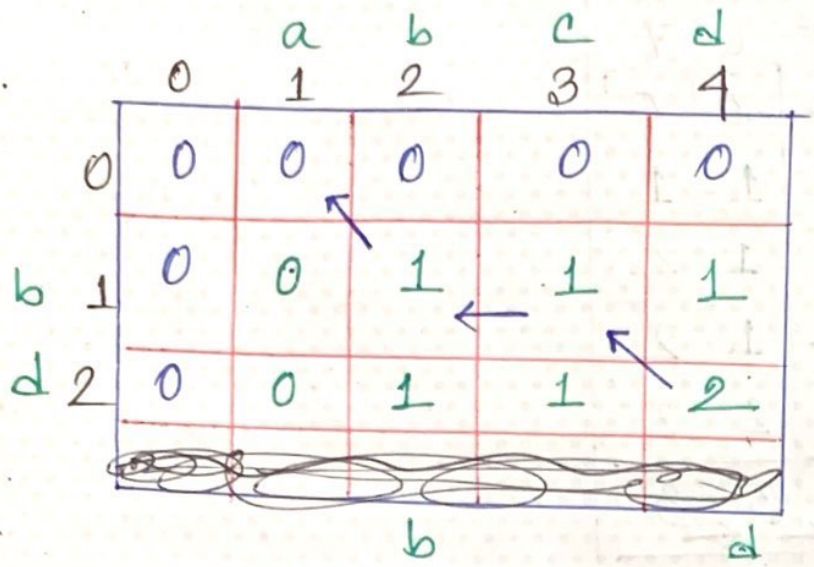


Date : .....

# #LCS using Dynamic Programming

String 1, A = | b | d | → m
1 2

String 2, B = | a | b | c | d | → n
1 2 3 4




# pseudocode

```
if (A[i] = B[j]){
  LCS[i,j] = 1 + LCS[i-1, j-1];
}
else
{
  LCS[i,j] = max(LCS[i-1,j]
              , LCS[i, j-1]);
}
```

∴ Longest Common Sequence is : bd.

∴ Complexity : O(m×n)

Example-2:

Str1 : Stone

Str2 : Longest

| | | L | o | n | g | e | s | t |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| t 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| o 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 |
| n 4 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| e 5 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 |

o n e

∴ Longest Common Subsequence is : one.

Date : .....

# Coin Change Problem

The Coin Change Problem is a classic dynamic Programming (DP) Problem in Computer Science. It asks for the number of ways to make change for a given amount of money using a specific set of coin denominations.

→ Suppose you have 3 types of coin
- 1 taka coin, 2 taka coin, 5 taka coin.
- These coins are infinite in numbers.

→ Now, you have been given an amount -
- In this case, the amount is = 6 taka

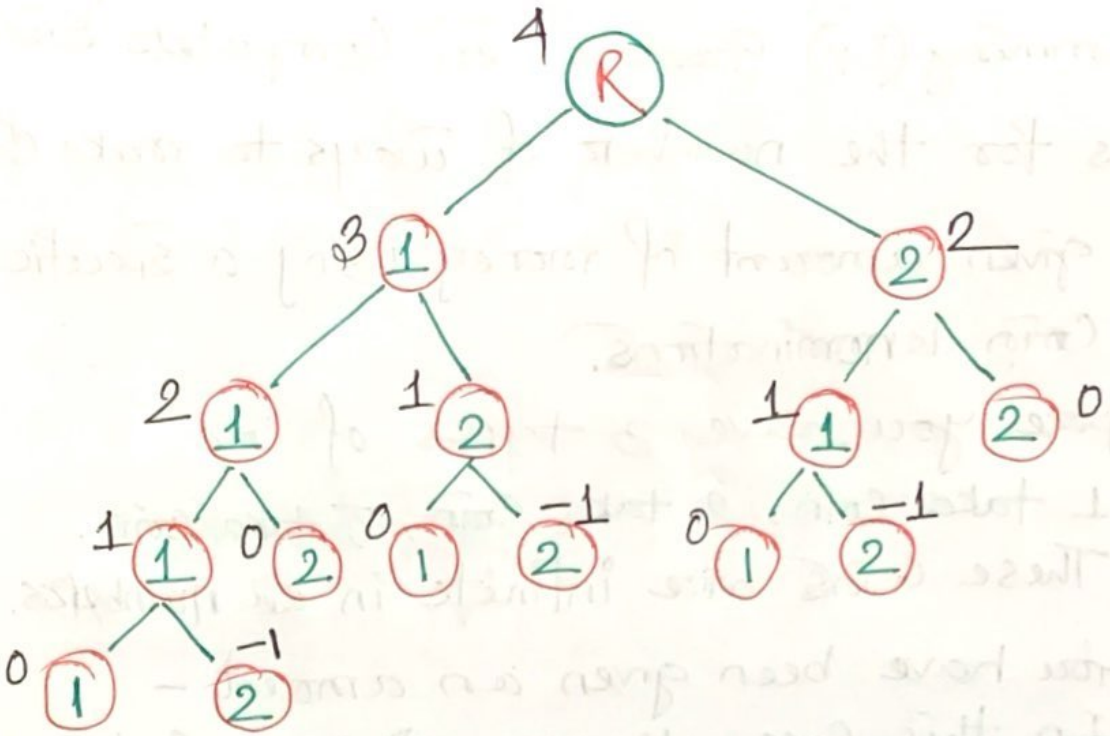
You have to find the minimum number of coins to form the Amount of taka 6.

⇒ Let us explore some possibilities.

| 1 taka | 2 taka | 5 taka | Total no. of Coins. |
|---|---|---|---|
| 6 | 0 | 0 | 6 |
| 4 | 1 | 0 | 5 |
| 2 | 2 | 0 | 4 |
| 0 | 3 | 0 | 3 |
| 1 | 0 | 1 | 2 |

# Backtrack Concept.

Suppose, you have coins [1, 2] and amount = 4 take




All possible pattern — {1,1,1,1}, {1,1,2} {1,2,1} {2,1,1} {2,2}.

⇒ For 2 types of coin, Complexity = $O(2^n)$

⇒ For n types of coin, Complexity = $O(n^n)$

⇒ Dynamic Programming can optimize the solutions.

⇒ This is called unbounded knapsack problem.

⇒ 0/1 knapsack is called bounded knapsack — Because there are specific no. of objects to load.

⇒ But in coin change, no. of coins/objects are infinite.

# Coin Change Problem

Date : .....

# Let's us Solve the Problem using bottom-up DP

* Suppose,

~~Coins Denomiz~~

- Coins Denomination = [1, 2, 5]
- Size, n = 3
- Amount, m = 6
- The table Size will be
  - int coins[n+1][m+1];

* Two Cases.
  - When no. of coins = 0, this case is equivalent to infinity
  - When amount = 0, no. of needed Coins = 0.

| Coin[i] | → j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | inf | inf | inf | inf | inf | inf |
| 1 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 2 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 5 | 3 | 0 | 1 | 1 | 2 | 2 | 1 | 3 |

* When j >= coin[i]

Coins[i][j] = min (Coins[i-1][j], 1 + Coins[i][j-coin[i]]

* When j < coin[i]

Coins[i][j] = Coins[i-1][j].

* Values of coins[i][j] are filled

Either from the upper cell of previous row OR from the Left part of the same row

| Coin[i] |
|---|
| 0 |
| 1 |
| 2 |
| 5 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | inf | inf | inf | inf | inf | inf |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 3 | 0 | 1 | 1 | 2 | 2 | 1 | 2 |

* Last/3rd row,
  - If value has been come from Left Side of same row, include that coin.
  - So ans = {5,
  - go 5 step back of the same row

    Select upper cell value of the 2nd row
* 1st row
  - This value has come from left so Select 1 taka coin.
  - So ans = {5, 1}
  - Go 1 step back of the same row and Select the value

Date : .....

* So, answer is
  - We need to maximum 2 coins to form the amount 6 taka
  - The coins are = {5, 1}

* Coins can be kept randomly in any order in the coins array

| coin[i] |
|---|
| 0 |
| 1 |
| 2 |
| 5 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | inf | inf | inf | inf | inf | inf |
| 1 | 0 | inf | 1 | inf | 2 | inf | 3 |
| 2 | 0 | inf | 1 | inf | 2 | 1 | inf |
| 3 | 0 | 1 | 1 | 2 | 2 | 1 | 2 |

Here, the answer is also = {1, 5}