

Problem-1

a)

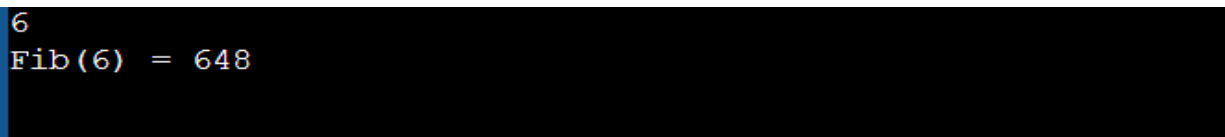
```
#include<stdio.h>

int Fibonacci(int N)
{
    if(N <= 3)
        return N;
    return Fibonacci(N-1)*Fibonacci(N-2)*Fibonacci(N-3);
}

int main()
{
    int n;
    scanf("%d",&n);
    printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;
}
```

Output



```
6
Fib(6) = 648
```

b)

```
#include<stdio.h>
#define size 50

int result[size];

void init_result()
{
    int i;

    for(i = 0; i < size; i++)
```

```

        result[i] = -1;
    }

    int Fibonacci(int N)
    {

        if(result[N] == -1)
        {
            if(N <= 3)
                result[N] = N;
            else
                result[N] = Fibonacci(N-1)*Fibonacci(N-2)*Fibonacci(N-3);
        }
        return result[N];
    }

    int main()
    {
        int n;

        scanf("%d",&n);

        init_result();
        printf("Fib(%d) = %d\n",n,Fibonacci(n));

        return 0;
    }

```

Output

```

6
Fib(6) = 648

```

c)

```

#include<stdio.h>

int Fibonacci(int N)
{

    int Fib[N+1],i;
    Fib[1] = 1;

```

```

Fib[2] = 2;
Fib[3] = 3;

for(i = 4; i <= N; i++)
    Fib[i] = Fibonacci(N-1)*Fibonacci(N-2)*Fibonacci(N-3);
return Fib[N];
}

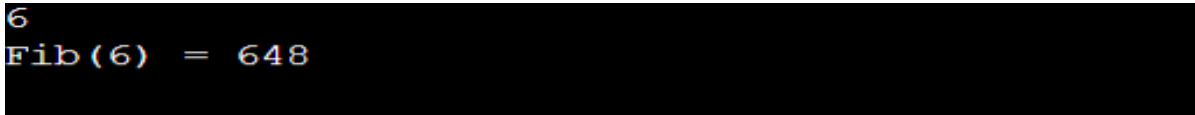
int main()
{
    int n;
    scanf("%d",&n);

    if(n <= 1)
        printf("Fib(%d) = %d\n",n,n);
    else
        printf("Fib(%d) = %d\n",n,Fibonacci(n));

    return 0;
}

```

Output



```

6
Fib(6) = 648

```

d)

(a) solves same subproblem again and again.

Time Complexity: $O(3^n)$

e)

part(c) do not use computer memory in order to save time.

part(a) will take longest amount of time to solve the problem because the time complexity of part(a) is $O(3^n)$.

part(a) will take least amount of time to solve the problem because the time complexity of part(c) is $O(n)$.

Problem-2

- a) 0/1 knapsack algorithm is used to get maximum total money that Chowdhury Shaheb can earn by selling his land.

b) Algorithm

```
void knapSack(int W, int n, int val[], int wt[]);  
int getMax(int x, int y);
```

```
int main(void) {
```

```
    int val[] = {6,5,9,7};  
    int wt[] = { 5, 5, 6, 4};
```

```
    int n = 4;  
    int W = 5;
```

```
    knapSack(W, n, val, wt);
```

```
    return 0;  
}
```

```
int getMax(int x, int y) {  
    if(x > y) {  
        return x;  
    } else {  
        return y;  
    }  
}
```

```
void knapSack(int W, int n, int val[], int wt[]) {  
    int i, w;  
    int V[n+1][W+1];  
  
    for(w = 0; w <= W; w++) {  
        V[0][w] = 0;  
    }  
}
```

```

for(i = 0; i <= n; i++) {
    V[i][0] = 0;
}

for(i = 1; i <= n; i++) {
    for(w = 1; w <= W; w++) {
        if(wt[i] <= w) {
            V[i][w] = getMax(V[i-1][w], val[i] + V[i-1][w - wt[i]]);
        } else {
            V[i][w] = V[i-1][w];
        }
    }
}

printf("Max Value: %d\n", V[n][W]);
}

```

Dry run

1st phase

```

for(i = 1; i <= 5; i++)
    for(w = 5; w <= 10; w++)
        if(wt[1] <= 5)
            V[1][5] = getMax(V[1-1][5], val[1] + V[1-1][5 - wt[1]]);

```

$$\begin{aligned}
 &= V[0][5] = 0, 6 + V[0][5-5] \\
 &= V[0][5] = 0, 6 + V[0][0] \\
 &= \text{getMax}(6)
 \end{aligned}$$

N[1][5] = 6

```

for(i = 1; i <= 5; i++)
    for(w = 5; w <= 10; w++)
        if(wt[1] <= 5)
            V[1][5] = getMax(V[1-1][5], val[1] + V[1-1][5 - wt[1]]);
                    = V[0][5] = 0, 6 + V[0][5-5]
                    = V[0][5] = 0, 6 + V[0][0]
                    = getMax(6)

```

N[1][5] = 6

```

for(i = 1; i <= 5; i++)
    for(w = 6; w <= 10; w++)
        if(wt[1] <= 6)
            V[1][6] = getMax(V[1-1][6], val[1] + V[1-1][6 - wt[1]]);
                    = V[0][6] = 0, 6 + V[0][6-5]
                    = V[0][6] = 0, 6 + V[0][1]
                    = getMax(6)

```

N[1][6] = 6

```

for(i = 1; i <= 5; i++)
    for(w = 4; w <= 10; w++)
        if(wt[1] <= 4)
            V[1][4] = getMax(V[1-1][4], val[1] + V[1-1][4 - wt[1]]);
                    = V[0][4] = 0, 6 + V[0][4-5]
                    = V[0][4] = 0, 6 + V[0][-1]
                    = getMax(6)

```

N[1][4] = 6

2nd phase

```
for(i = 2; i <= 5; i++)
```

```
    for(w = 5; w <= 10; w++)
```

```
        if(wt[2] <= 5)
```

```
            V[2][5] = getMax(V[2-1][5], val[2] + V[2-1][5 - wt[2]]);
```

```
                = V[1][5] = 6, 5 + V[1][5-5]
```

```
                = V[1][5] = 6, 5 + V[1][0]
```

```
                = getMax(5)
```

N[2][5] = 5

```
for(i = 2; i <= 5; i++)
```

```
    for(w = 5; w <= 10; w++)
```

```
        if(wt[2] <= 5)
```

```
            V[2][5] = getMax(V[2-1][5], val[2] + V[2-1][5 - wt[2]]);
```

```
                = V[1][5] = 6, 5 + V[1][5-5]
```

```
                = V[1][5] = 6, 5 + V[1][0]
```

```
                = getMax(5)
```

N[2][5] = 5

```
for(i = 2; i <= 5; i++)
```

```
    for(w = 6; w <= 10; w++)
```

```
        if(wt[2] <= 6)
```

```
            V[2][6] = getMax(V[2-1][6], val[2] + V[2-1][6 - wt[2]]);
```

```
                = V[1][6] = 6, 5 + V[1][6-5]
```

```
                = V[0][6] = 6, 5 + V[0][1]
```

```
                = getMax(5)
```

$$N[2][6] = 5$$

```
for(i = 2; i <= 5; i++)
```

```
    for(w = 4; w <= 10; w++)
```

```
        if(wt[2] <= 4)
```

```
            V[2][4] = getMax(V[2-1][4], val[2] + V[2-1][4 - wt[2]]);
```

```
                = V[1][4] = 0, 5 + V[1][4-5]
```

```
                = V[1][4] = 0, 5 + V[1][-1]
```

```
                = getMax(0)
```

$$N[2][4] = 0$$

Size in Bighas: { 5, 5, 6, 4 }

Price in Crores of taka: {6,5,9,7}

V[i,w]	W = 0	1	2	3	4	5	6	7	8	9	10
i = 0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	6	6	6	6	6	6
2	0	0	0	0	0	5	5	5	5	5	11
3	0	0	0	0	0	5	9	9	9	9	11
4	0	0	0	0	7	7	7	7	7	13	16

- c) I agree with my friend. He uses in part (a) incremental design technique. Part (a) is dynamic programming bottom-up method. It divides the problem into sub problem and stores data in the computer memory. bottom-up method also known as incremental design technique.

Problem-3

a) NO, simple string comparison algorithm cannot be used to solve this problem. In this question simple string comparison algorithm returns MATCHED when input is the same. But in this problem, we have 2 different inputs for this reason this algorithm is not working

b)

LCS-LENGTH(X, Y)

```
1  m=X.length
2  n=Y.length
3  let b[1 to m, 1 to n] and c[0 to m, 0 to n] be new tables
4  for i=1 to m
5      c[i,0]=0
6  for i=1 to n
7      c[0,i]=0
8  for i=1 to m
9      for j=1 to n
10         if  $x_i == y_j$ 
11              $c[i,j] = c[i-1, j-1] + 1$ 
12              $b[i,j] = \text{"↖"}$ 
13         elseif  $c[i-1,j] > c[i,j-1]$ 
14              $c[i,j] = c[i-1,j]$ 
15              $b[i,j] = \text{"↑"}$ 
16         else  $c[i,j] = c[i, j-1]$ 
17              $b[i,j] = \text{"↘"}$ 
18  return c and b
```

C)

PRINT-LCS(b, X, I, j)

```
1  if  $i==0$  or  $j==0$ 
2    return
3  if  $b[i,j]==$  " ↖ "
4    PRINT-LCS( $b, X, i-1, j-1$ )
5    print  $x_i$ 
6  elseif  $b[i,j]==$  " ↑ "
7    PRINT-LCS( $b, X, i-1, j$ )
8  else PRINT-LCS( $b, X, I, j-1$ )
```

f) Dynamic Programming can be used to find the longest common substring.

Problem-4

a)

b) We Can use 2 algorithms used to solve the problem faced by SPARRSO. They are Kruskal & prim's algorithms.

c)

Kruskal Algorithm:

MST-KRUSKAL(G, w)

```
1   $A = \emptyset$ 
2  for each vertex  $v$  belongs to  $G.V$ 
3      MAKE-SET( $v$ )
4  sort the edges  $G.E$  into non decreasing order by weight  $w$ 
5  for each edge  $(u, v)$  belongs to  $G.E$  in non decreasing order by weight
6.      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{(u, v)\}$ 
8          UNION( $u, v$ )
9  return  $A$ 
```

Prim's Algorithms:

MST-PRIM(G, w, r)

```
1  for each  $u$  belongs to  $G.V$ 
2       $u.key = \text{infinity}$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v$  belongs to  $G.Adj[u]$ 
9          if  $v$  belongs to  $Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```

d)

e)

Problem-5

a)

BFS(G, s)

```
1  for each  $u$  belongs to  $G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \text{infinity}$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE ( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v$  belongs to  $G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE ( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

b) Depth first search (DFS) algorithm also is used for this problem.

BFS(G, s)

```
1  for each  $u$  belongs to  $G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each  $u$  belongs to  $G.V$ 
6      if  $v.color == \text{WHITE}$ 
7          DFS – VISIT ( $G, u$ )
```

c)

There are two methods for representing graph in computer:

1. Adjacency matrix
2. Adjacency list

Problem-6

BELLMAN-FORD(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE ( $G, s$ )
2 for  $i = 1$  to  $|G.V| - 1$ 
3   for each edge  $(u, v)$  belongs to  $G.E$ 
4     RELAX ( $u, v, w$ )
5 for each edge  $(u, v)$  belongs to  $G.E$ 
6   if  $v.d > u.d + w(u, v)$ 
7     return FALSE
8 return TRUE
```