

ASSIGNMENT

khondoker Md. Sabit Hasan [21-45306-2]



AUGUST 26, 2022

PROBLEM 1 (MATHEMATICIAN) ANSWER

(a)

```
#include<bits/stdc++.h>
using namespace std;

int NewSeries(int n)
{
    if (n <= 3)
        return n;
    return NewSeries(n-1) * NewSeries(n-2) * NewSeries(n-3);
}

int main ()
{
    int n;
    cout << "Which term do you want to extract?\n";
    cin >> n;
    if(n<1){
        cout << "Invalid" << endl;
    }
    else{
        cout << n << "th term of this series is " << NewSeries(n) << endl;
    }
    return 0;
}
```

(b)

```
#include<bits/stdc++.h>
#define size 50
using namespace std;

int result[size];

void init_result()
{
    for(int i = 0; i < size; i++)
        result[i] = -1;
    //-1 indicates that the subproblem result needs to be computed
}

int NewSeries(int n)
{
    //if the subproblem is not computed yet,
    //recursively compute and store the result
    if(result[n] == -1)
    {
        if(n <= 3)
            result[n] = n;
    }
}
```

```

        else
            result[n] = NewSeries(n-1) * NewSeries(n-2) * NewSeries(n-3);
    }
    //Otherwise, just return the result
    return result[n];
}

int main ()
{
    int n;
    init_result();
    cout << "Which term do you want to extract?\n";
    cin >> n;
    if(n<1){
        cout << "Invalid" << endl;
    }
    else{
        cout << n << "th term of this series is " << NewSeries(n) << endl;
    }
    return 0;
}

```

(c)

```

#include<bits/stdc++.h>
using namespace std;

int NewSeries (int n)
{
    int New[n];

    New[0] = 1;
    New[1] = 2;
    New[2] = 3;

    for(int i = 3; i < n; i++)
        New[i] = New[i-1] * New[i-2] * New[i-3];

    //2nd last index will have the result
    return New[n-1];
}

int main ()
{
    int n;
    cout << "Which term do you want to extract?\n";
    cin >> n;
    if(n<1){

```

```

        cout << "Invalid" << endl;
    }
    else{
        cout << n << "th term of this series is " << NewSeries(n) << endl;
    }
    return 0;
}

```

(d)

The Divide and Conquer algorithm from part (a) solves the same subproblems over and over again. The Divide and Conquer algorithm solves a problem by the following two steps-

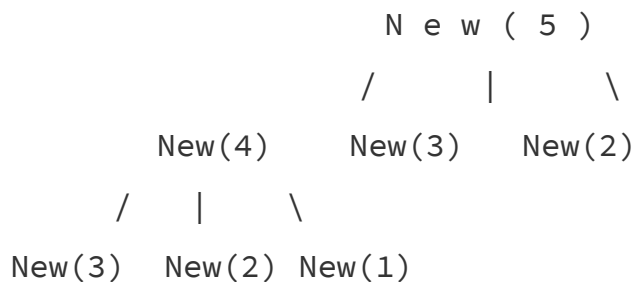
1. Divide the problem into subproblems.
2. Conquer the subproblems by solving them recursively.

Now, if we want to compute New(5), the Divide and Conquer approach will do the following-

New(5) -> Go and compute New(4), New(3) and New(2) and return the results.

New(4) -> Go and compute New(3), New(2) and New(1) and return the results.

Finally, New(3) will return 3, New(2) will return 2, and New(1) will return 1.



Here, we are computing the result of New(3) and New(2) twice. Thus, we can say that the Divide and Conquer algorithm from part (a) solves the same subproblems over and over again.

(e)

The Divide and Conquer algorithm from part (a) does not use computer memory to save time. The Divide and Conquer algorithm solves a problem by the following two steps-

1. Divide the problem into subproblems.
2. Conquer the subproblems by solving them recursively.

There is no use of computer memory in the Divide and Conquer algorithm.

But on the other hand, the Top-Down with Memoization Dynamic Programming algorithm uses computer memory to solve the problem. Top-Down breaks the large problem into multiple subproblems. If the subproblems are solved already, reuse the answer. Otherwise, solve the subproblem and store the result. Top-Down uses memoization to avoid recomputing the same subproblem again.

Also, the Bottom-Up Dynamic Programming algorithm uses computer memory to solve the problem. In the Bottom-Up approach, it starts computing the result for the subproblem. Then using the subproblem result solves another subproblem and finally solves the whole problem.

Thus, we can say that the Divide and Conquer algorithm from part (a) does not use computer memory to save time.

The Divide and Conquer algorithm uses recursion to solve a problem, the Top-Down with Memoization Dynamic Programming algorithm uses recursion and array to solve a problem, and the Bottom-Up Dynamic Programming algorithm uses only an array to solve a problem.

So, the Divide and Conquer algorithm will take the longest amount of time to find the n-th term of the above series, and the Bottom-Up Dynamic Programming algorithm will take the least amount of time.

PROBLEM 2 (CHOWDHURY SAHEB) ANSWER

(a)

At first glance, the problem of finding out the maximum total money that Chowdhury Saheb can earn by selling his lands in a year can be solved either using the Fractional Knapsack algorithm or the 0/1 Knapsack algorithm. But here are two laws that we must consider while solving the problem. The laws are-

1. On one will be able to sell a total of more than 10 Bighas of land in a year.
2. While selling land, a person must sell the full land.

Now, let's talk about the property above two algorithms-

Fractional Knapsack: In Fractional Knapsack, we can break items to maximize the total value of the knapsack.

0/1 Knapsack: In the 0/1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

Thus, for the 2nd law, "While selling land, a person must sell the full land," the appropriate choice of the algorithm should be 0/1 Knapsack because the property of the 0/1 Knapsack algorithm matches the condition.

(b)

0/1 Knapsack algorithm to solve the above problem:

```
#include<bits/stdc++.h>
using namespace std;

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;

    //value table having n+1 rows and W+1 columns
    int V[n+1][W+1];
```

```

//fill the row i=0 with value 0
for(w = 0; w <= W; w++){
    V[0][w] = 0;

//fill the column w=0 with value 0
for(i = 0; i <= n; i++){
    V[i][0] = 0;

//fill the value table
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];
    }
}

    cout << "The Maximum total money that Chowdhury Saheb can earn by selling
lands in a year: " << V[n][W] << " Crores." << endl;
}

int main(){
    //the first element is set to -1 as
    //we are storing items from index 1
    //in val[] and wt[] array
    int val[] = {-1,6,5,9,7};           //price of the lands
    int wt[] = {-1,5,5,6,4};           //size in bighas of the lands

    int W = 10; //maximum bighas can be sold in a year
    int n = 4; //total properties

    knapSack(W, n, val, wt);

    return 0;
}

```

DRY RUN:

Knapsack(int W, int n, int val[], int wt[]) → W = 10, n = 4, val[] = {-1,6,5,9,7} and wt[] = {-1,5,5,6,4}

int V[4+1][10+1]

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

for(w = 0; w <= 10; w++)

V[0][w] = 0;

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

for(i = 0; i <= 4; i++)

V[i][0] = 0;

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

for(i = 1; i <= 4; i++)

for(w = 1; w <= 10; w++)

if(wt[1] <= w) → 5 <= 1 ✗

else

V[1][1] = V[1-1][1] → V[0][1] → 0

for(w = 2; w <= 10; w++)

if(wt[1] <= w) → 5 <= 2 ✗

else

V[1][2] = V[1-1][2] → V[0][2] → 0

for(w = 3; w <= 10; w++)

if(wt[1] <= w) → 5 <= 3 ✗

else

V[1][3] = V[1-1][3] → V[0][3] → 0

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

if(wt[1] <= w) → 5 <= 4 ✗

else

V[1][4] = V[1-1][4] → V[0][4] → 0

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

if(wt[1] <= w) → 5 <= 5 ✓

V[1][5] = getMax(V[1-1][5], val[1]+V[1-1][5-wt[1]]) → getMax(0, 6+0) → 6

```

for(w = 6; w <= 10; w++)
    if(wt[1] <= w) → 5<=6 ✓
        V[1][6] = getMax( V[1-1][6], val[1]+V[1-1][6-wt[1]] ) → getMax( 0, 6+0 ) → 6
for(w = 7; w <= 10; w++)
    if(wt[1] <= w) → 5<=7 ✓
        V[1][7] = getMax( V[1-1][7], val[1]+V[1-1][7-wt[1]] ) → getMax( 0, 6+0 ) → 6
for(w = 8; w <= 10; w++)
    if(wt[1] <= w) → 5<=8 ✓
        V[1][8] = getMax( V[1-1][8], val[1]+V[1-1][8-wt[1]] ) → getMax( 0, 6+0 ) → 6
for(w = 9; w <= 10; w++)
    if(wt[1] <= w) → 5<=9 ✓
        V[1][9] = getMax( V[1-1][9], val[1]+V[1-1][9-wt[1]] ) → getMax( 0, 6+0 ) → 6
for(w = 10; w <= 10; w++)
    if(wt[1] <= w) → 5<=10 ✓
        V[1][10] = getMax( V[1-1][10], val[1]+V[1-1][10-wt[1]] ) → getMax( 0, 6+0 ) → 6
for(w = 11; w <= 10; w++) ✗

```

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

for(i = 2; i <= 4; i++)

```

for(w = 1; w <= 10; w++)
    if(wt[2] <= w) → 5<=1 ✗
    else
        V[2][1] = V[2-1][1] → V[1][1] → 0
for(w = 2; w <= 10; w++)
    if(wt[2] <= w) → 5<=2 ✗
    else
        V[2][2] = V[2-1][2] → V[1][2] → 0
for(w = 3; w <= 10; w++)
    if(wt[2] <= w) → 5<=3 ✗
    else
        V[2][3] = V[2-1][3] → V[1][3] → 0
for(w = 4; w <= 10; w++)
    if(wt[2] <= w) → 5<=4 ✗
    else
        V[2][4] = V[2-1][4] → V[1][4] → 0
for(w = 5; w <= 10; w++)
    if(wt[2] <= w) → 5<=5 ✓
        V[2][5] = getMax( V[2-1][5], val[2]+V[2-1][5-wt[2]] ) → getMax( 6, 5+0 ) → 6
for(w = 6; w <= 10; w++)
    if(wt[2] <= w) → 5<=6 ✓
        V[2][6] = getMax( V[2-1][6], val[2]+V[2-1][6-wt[2]] ) → getMax( 6, 5+0 ) → 6
for(w = 7; w <= 10; w++)
    if(wt[2] <= w) → 5<=7 ✓
        V[2][7] = getMax( V[2-1][7], val[2]+V[2-1][7-wt[2]] ) → getMax( 6, 5+0 ) → 6

```



```

for(w = 8; w <= 10; w++)
    if(wt[2] <= w) → 5<=8 ✓
        V[2][8] = getMax( V[2-1][8], val[2]+V[2-1][8-wt[2]] ) → getMax( 6, 5+0 ) → 6
for(w = 9; w <= 10; w++)
    if(wt[2] <= w) → 5<=9 ✓
        V[2][9] = getMax( V[2-1][9], val[2]+V[2-1][9-wt[2]] ) → getMax( 6, 5+0 ) → 6
for(w = 10; w <= 10; w++)
    if(wt[2] <= w) → 5<=10 ✓
        V[2][10] = getMax( V[2-1][10], val[2]+V[2-1][10-wt[2]] ) → getMax( 6, 5+6 ) → 11
for(w = 11; w <= 10; w++) ✗

```

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

for(i = 3; i <= 4; i++)

```

for(w = 1; w <= 10; w++)
    if(wt[3] <= w) → 6<=1 ✗
    else
        V[3][1] = V[3-1][1] → V[2][1] → 0
for(w = 2; w <= 10; w++)
    if(wt[3] <= w) → 6<=2 ✗
    else
        V[3][2] = V[3-1][2] → V[2][2] → 0
for(w = 3; w <= 10; w++)
    if(wt[3] <= w) → 6<=3 ✗
    else
        V[3][3] = V[3-1][3] → V[2][3] → 0
for(w = 4; w <= 10; w++)
    if(wt[3] <= w) → 6<=4 ✗
    else
        V[3][4] = V[3-1][4] → V[2][4] → 0
for(w = 5; w <= 10; w++)
    if(wt[3] <= w) → 6<=5 ✗
    else
        V[3][5] = V[3-1][5] → V[2][5] → 6
for(w = 6; w <= 10; w++)
    if(wt[3] <= w) → 6<=6 ✓
        V[3][6] = getMax( V[3-1][6], val[3]+V[3-1][6-wt[3]] ) → getMax( 6, 9+0 ) → 9
for(w = 7; w <= 10; w++)
    if(wt[3] <= w) → 6<=7 ✓
        V[3][7] = getMax( V[3-1][7], val[3]+V[3-1][7-wt[3]] ) → getMax( 6, 9+0 ) → 9
for(w = 8; w <= 10; w++)
    if(wt[3] <= w) → 6<=8 ✓
        V[3][8] = getMax( V[3-1][8], val[3]+V[3-1][8-wt[3]] ) → getMax( 6, 9+0 ) → 9
for(w = 9; w <= 10; w++)
    if(wt[3] <= w) → 6<=9 ✓
        V[3][9] = getMax( V[3-1][9], val[3]+V[3-1][9-wt[3]] ) → getMax( 6, 9+0 ) → 9

```

```

for(w = 10; w <= 10; w++)
    if(wt[3] <= w) → 6<=10 ✓
        V[3][10] = getMax( V[3-1][10], val[3]+V[3-1][10-wt[3]] ) → getMax( 11, 9+0 ) → 11
for(w = 11; w <= 10; w++) ✗

```

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

for(i = 4; i <= 4; i++)

```

for(w = 1; w <= 10; w++)
    if(wt[4] <= w) → 4<=1 ✗
    else
        V[4][1] = V[4-1][1] → V[3][1] → 0
for(w = 2; w <= 10; w++)
    if(wt[4] <= w) → 4<=2 ✗
    else
        V[4][2] = V[4-1][2] → V[3][2] → 0
for(w = 3; w <= 10; w++)
    if(wt[4] <= w) → 4<=3 ✗
    else
        V[4][3] = V[4-1][3] → V[3][3] → 0
for(w = 4; w <= 10; w++)
    if(wt[4] <= w) → 4<=4 ✓
        V[4][4] = getMax( V[4-1][4], val[4]+V[4-1][4-wt[4]] ) → getMax( 0, 7+0 ) → 7
for(w = 5; w <= 10; w++)
    if(wt[4] <= w) → 4<=5 ✓
        V[4][5] = getMax( V[4-1][5], val[4]+V[4-1][5-wt[4]] ) → getMax( 6, 7+0 ) → 7
for(w = 6; w <= 10; w++)
    if(wt[4] <= w) → 4<=6 ✓
        V[4][6] = getMax( V[4-1][6], val[4]+V[4-1][6-wt[4]] ) → getMax( 9, 7+0 ) → 9
for(w = 7; w <= 10; w++)
    if(wt[4] <= w) → 4<=7 ✓
        V[4][7] = getMax( V[4-1][7], val[4]+V[4-1][7-wt[4]] ) → getMax( 9, 7+0 ) → 9
for(w = 8; w <= 10; w++)
    if(wt[4] <= w) → 4<=8 ✓
        V[4][8] = getMax( V[4-1][8], val[4]+V[4-1][8-wt[4]] ) → getMax( 9, 7+0 ) → 9
for(w = 9; w <= 10; w++)
    if(wt[4] <= w) → 4<=9 ✓
        V[4][9] = getMax( V[4-1][9], val[4]+V[4-1][9-wt[4]] ) → getMax( 9, 7+6 ) → 13
for(w = 10; w <= 10; w++)
    if(wt[4] <= w) → 4<=10 ✓
        V[4][10] = getMax( V[4-1][10], val[4]+V[4-1][10-wt[4]] ) → getMax( 11, 7+9 ) → 16
for(w = 11; w <= 10; w++) ✗

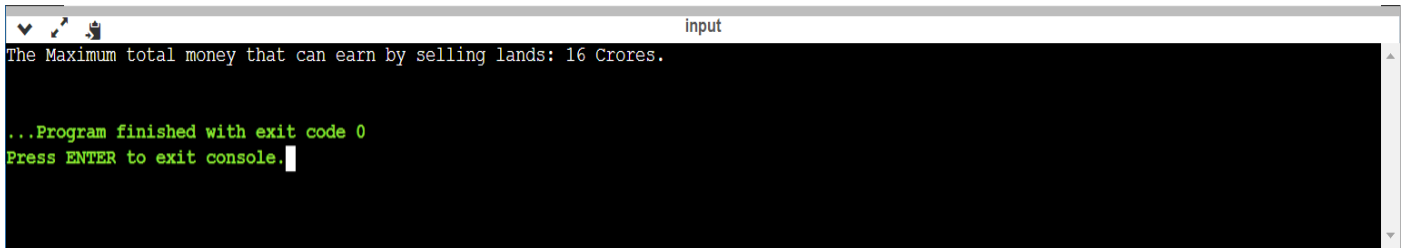
```

`for(i = 5; i <= 4; i++)` ×

Final Array:

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

```
cout << "The Maximum total money that can earn by selling lands: " << V[4][10] << " Crores."
<< endl;
```

A screenshot of a terminal window titled 'input'. The output text is: 'The Maximum total money that can earn by selling lands: 16 Crores.' followed by '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a cursor at the end.

(c)

The algorithm I suggested in response to part (a) was 0/1 Knapsack. 0/1 Knapsack algorithm uses the Bottom-Up Dynamic Programming design technique.

If a problem has any one of the following properties below, then we can use the Dynamic Programming technique-

1. Overlapping Subproblem
2. Optimal Substructure

Here, 0/1 knapsack has the optimal substructure because solutions of smaller problems help to find the solution of the bigger problem. For example- the solution of $V[3][3]$ helps to find the solution of $V[4][3]$.

Thus, the 0/1 knapsack algorithm uses the Bottom-Up Dynamic Programming design technique.

But, we can say partially that it is also an incremental design technique because we get the full solution incrementally. When we get the value of $V[3][10]$, that was a partial solution to the full solution. So, in a way, we can say that 0/1 Knapsack uses the incremental design technique. But the design technique will be more meaningful if we say that 0/1 knapsack uses the Bottom-Up Dynamic Programming design technique.

PROBLEM 3 (ONUSONDHAN) ANSWER

(a)

No, we cannot use a simple string comparison algorithm that takes two strings as input and outputs "MATCHED" only if both the strings are the same to solve the above problem. Because if we use this algorithm, only the web page containing the string "CAT" will appear when the entered search string is "CAT." This algorithm will return true only when the two strings are the same, and it will return false even if one character is different.

So, we cannot use a simple string comparison algorithm in this problem. Because if we entered the search string "CAT," a web page containing "TAB" should appear in the search result as per the problem. But, if we use a simple string comparison algorithm, a web page containing "TAB" will not appear in the search result.

Thus, we cannot use a simple string comparison algorithm in this particular problem.

(b)

I will use the Print-LCS algorithm to print the maximum number of common characters that appear in the same order in a user-entered search string and the content of a web page.

"The Longest Common Subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences."

So, in this problem, I will use the Print-LCS algorithm.

(c)

Yes, the algorithm I suggested in part (b), Print-LCS, will need help from another algorithm for generating the solution. The algorithm is Longest Common Subsequence (LCS). Because in Print-LCS, we need an array called "b" to print the maximum number of common characters. Longest Common Subsequence (LCS) will help us to build that array called "b."

So, I will use the Longest Common Subsequence (LCS) and the Print-LCS to solve this problem.

(d)

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

LCS-LENGTH(X, Y)

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
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              $b[i, j] == "\nwarrow"$ 
13         elseif  $c[i-1, j] \geq c[i, j-1]$ 
14              $c[i, j] = c[i-1, j]$ 
15              $b[i, j] == "\uparrow"$ 
16         else  $c[i, j] = c[i, j-1]$ 
17              $b[i, j] == "\leftarrow"$ 
18 return  $c$  and  $b$ 

```

(e)

DRY RUN:

LCS-Length(X, Y) $\rightarrow X = \text{"CAT"} \ \& \ Y = \text{"TAB"}$

$m = X.length \rightarrow 3$

$n = Y.length \rightarrow 3$

$b[1..3, 1..3] \ \& \ c[0..3, 0..3]$

b

$[i, j]$		1	2	3
	$[x, y]$	T	A	B
1	C	\uparrow	\uparrow	\uparrow
2	A	\uparrow	\nwarrow	\leftarrow
3	T	\nwarrow	\uparrow	\uparrow

c

$[i, j]$		0	1	2	3
	$[x, y]$		T	A	B
0					
1	C				
2	A				
3	T				

for i = 1 to 3
 c[i,0] = 0

c

[i,j]		0	1	2	3
	[x,y]		T	A	B
0					
1	C	0			
2	A	0			
3	T	0			

for j = 0 to 3
 c[0,j] = 0

c

[i,j]		0	1	2	3
	[x,y]		T	A	B
0		0	0	0	0
1	C	0	0	0	0
2	A	0			
3	T	0			

for i = 1 to 3

for j = 1 to 3

if $x_1 == y_1 \rightarrow C == T$ ✗

elseif $c[1-1,1] \geq c[1,1-1] \rightarrow c[0,1] \geq c[1,0] \rightarrow 0 \geq 0$ ✓

$c[1,1] = c[1-1,1] \rightarrow c[0,1] \rightarrow 0$

$b[1,1] = \text{"\u2191"}$

for j = 2 to 3

if $x_1 == y_2 \rightarrow C == A$ ✗

elseif $c[1-1,2] \geq c[1,2-1] \rightarrow c[0,2] \geq c[1,1] \rightarrow 0 \geq 0$ ✓

$c[1,2] = c[1-1,2] \rightarrow c[0,2] \rightarrow 0$

$b[1,2] = \text{"\u2191"}$

for j = 3 to 3

if $x_1 == y_3 \rightarrow C == B$ ✗

elseif $c[1-1,3] \geq c[1,3-1] \rightarrow c[0,3] \geq c[1,2] \rightarrow 0 \geq 0$ ✓

$c[1,3] = c[1-1,3] \rightarrow c[0,3] \rightarrow 0$

$b[1,3] = \text{"\u2191"}$

for j = 4 to 3 ✗

c

[i,j]		0	1	2	3
	[x,y]		T	A	B
0		0	0	0	0
1	C	0	0	0	0
2	A	0	0	1	1
3	T	0			

for i = 2 to 3

for j = 1 to 3

if $x_2 == y_1 \rightarrow A == T$ ✗

elseif $c[2-1,1] \geq c[2,1-1] \rightarrow c[1,1] \geq c[2,0] \rightarrow 0 \geq 0$ ✓

$c[2,1] = c[2-1,1] \rightarrow c[1,1] \rightarrow 0$

$b[2,1] = \text{"\u2191"}$

for j = 2 to 3
 if $x_2 == y_2 \rightarrow A == A$ ✓
 $c[2,2] = c[2-1,2-1] + 1 \rightarrow c[1,1] + 1 \rightarrow 0 + 1 \rightarrow 1$
 $b[2,2] = "\nwarrow"$
 for j = 3 to 3
 if $x_2 == y_3 \rightarrow A == B$ ✗
 elseif $c[2-1,3] \geq c[2,3-1] \rightarrow c[1,3] \geq c[2,2] \rightarrow 0 \geq 1$ ✗
 else $c[2,3] = c[2,3-1] \rightarrow c[2,2] \rightarrow 1$
 $b[2,3] = "\leftarrow"$
 for j = 4 to 3 ✗

c

[i,j]		0	1	2	3
	[x,y]		T	A	B
0		0	0	0	0
1	C	0	0	0	0
2	A	0	0	1	1
3	T	0	1	1	1

for i = 3 to 3

for j = 1 to 3
 if $x_3 == y_1 \rightarrow T == T$ ✓
 $c[3,1] = c[3-1,1-1] + 1 \rightarrow c[2,0] + 1 \rightarrow 0 + 1 \rightarrow 1$
 $b[3,1] = "\nwarrow"$
 for j = 2 to 3
 if $x_3 == y_2 \rightarrow T == A$ ✗
 elseif $c[3-1,2] \geq c[3,2-1] \rightarrow c[2,2] \geq c[3,1] \rightarrow 1 \geq 1$ ✓
 $c[3,2] = c[3-1,2] \rightarrow c[2,2] \rightarrow 1$
 $b[3,2] = "\uparrow"$
 for j = 3 to 3
 if $x_3 == y_3 \rightarrow T == B$ ✗
 elseif $c[3-1,3] \geq c[3,3-1] \rightarrow c[2,3] \geq c[3,2] \rightarrow 1 \geq 1$ ✓
 $c[3,3] = c[3-1,3] \rightarrow c[2,3] \rightarrow 1$
 $b[3,3] = "\uparrow"$
 for j = 4 to 3 ✗

for i = 4 to 3 ✗

Final Array

c

[i,j]		0	1	2	3
	[x,y]		T	A	B
0		0	0	0	0
1	C	0	0	0	0
2	A	0	0	1	1
3	T	0	1	1	1

Final Array

		b		
[i,j]		1	2	3
	[x,y]	T	A	B
1	C	↑	↑	↑
2	A	↑	↖	←
3	T	↖	↑	↑

Print-LCS(b, X, i, j) → b array & X = "CAT" & i = X.length = 3 & j = Y.length

if i == 0 or j == 0 ✗

if b[3,3] == "↖" ✗

elseif b[3,3] == "↑" ✓

Print-LCS(b, X, i-1, j) → b array & X = "CAT" & i = 3-1 = 2 & j = 3

if i == 0 or j == 0 ✗

if b[2,3] == "↖" ✗

elseif b[2,3] == "↑" ✗

else Print-LCS(b, X, i, j-1) → b array & X = "CAT" & i = 2 & j = 3-1 = 2

if i == 0 or j == 0 ✗

if b[2,2] == "↖" ✓

Print-LCS(b, X, i-1, j-1) → b array & X = "CAT" & i = 2-1 = 1 & j = 2-1 = 1

if i == 0 or j == 0 ✗

if b[1,1] == "↖" ✗

elseif b[1,1] == "↑" ✓

Print-LCS(b, X, i-1, j) → b array & X = "CAT" & i = 1-1 = 0 & j = 1

if i == 0 or j == 0 ✓

return

print x2 → x2 = A → Output: A

Final Output : A

(e)

The algorithms that I suggested in response to part (d) are LCS-Length and Print-LCS.

The LCS-Length algorithm uses the Bottom-Up Dynamic Programming design technique. If a problem has any one of the following properties below, then we can use the Dynamic Programming technique-

1. Overlapping Subproblem
2. Optimal Substructure

Here, LCS-Length has the optimal substructure because solutions of smaller problems help to find the solution of the bigger problem. For example- the solution of C[2,0] helps to find out the solution of c[3,1]. Another example is if,

X = {AC}

Y = {AC}

{A}

{A}

LCS = 1

&

{AC}

{AC}

LCS = 1+1 = 2

Here, LCS of {A} & {A} helps to found out the LCS of {AC} & {AC}. This is called optimal Substructure.

Thus, the LCS-Length algorithm uses the Bottom-Up Dynamic Programming design technique.

And the Print-LCS uses the Divide and Conquer design technique. The Divide and Conquer algorithm solves a problem by the following two steps-

1. Divide the problem into subproblems.
2. Conquer the subproblems by solving them recursively.

Here, in Print-LCS, we divide the bigger problem, $i = 3$ & $j = 3$, into subproblem such as $i = 2$ & $j = 3$ and then recursively conquer the subproblem.

Thus, the Print-LCS algorithm uses the Divide and Conquer design technique.

PROBLEM 4 (SPARRSO) ANSWER

(a)

No, in this problem, we cannot apply Dijkstra's Shortest Path Algorithm to find out the tunnels which, if built, will connect all the habitats or locations at the minimum possible cost.

In this problem, we need to find the minimum spanning tree (MST). A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. And, A minimum spanning tree (MST) is a spanning tree in which the sum of the weight of the edges is as minimum as possible. So, the minimum spanning tree of this graph will connect all the habitats or locations at the minimum possible cost.

But on the contrary, Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree (MST) because the shortest distance between two vertices might not include all the graph's vertices.

Thus, Dijkstra's Shortest Path Algorithm will not work here.

(b)

We need to find the minimum spanning tree (MST) in this problem. A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. And, A minimum spanning tree (MST) is a spanning tree in which the sum of the weight of the edges is as minimum as possible. So, the minimum spanning tree of this graph will connect all the habitats or locations at the minimum possible cost.

Kruskal's algorithm is one way to find a minimum spanning tree (MST), and **Prim's** algorithm is another way to find a minimum spanning tree (MST). Both algorithms will find the subgraph (tree) of the original graph with a minimum possible number of edges and keep the sum of the weight of the edges as minimum as possible.

(c)

MST-KRUSKAL(G, w)

```
1  A =  $\emptyset$ 
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in G.E$ , taken in nondecreasing 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
```

MST-PRIM(G, w, r)

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
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  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $u.key = w(u, v)$ 
```

Here, I have written two algorithms. One is MST-Kruskal and another one is MST-Prim. Both algorithms generate a minimum spanning tree, but their approach is different from each other. In Kruskal, the graph **grows as a forest**. On the other hand, in Prim, the graph **grows as a tree**. Also, they have other differences in their property.

(d)

DRY RUN:

MST-Kruskal(G, w) $\rightarrow G \rightarrow G.V = \{0, 1, 2, 3, 4, 5, 6\}$ & $G.E = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,3), \text{edge}(1,5), \text{edge}(1,6), \text{edge}(2,3), \text{edge}(2,4), \text{edge}(2,5), \text{edge}(3,4), \text{edge}(4,6)\}$ and $w \rightarrow$ the edge weights of the graph, all of which are equal to 3.

```
A =  $\emptyset$ 
for each vertex  $v \in G.V$ 
    Make-Set( $v$ )
    {0}
    {1}
    {2}
```

{3}
{4}
{5}
{6}

sort the edges of G.E into nondecreasing order by weight $w = 3$

1. edge(0,1)
2. edge(0,3)
3. edge(1,2)
4. edge(1,3)
5. edge(1,5)
6. edge(1,6)
7. edge(2,3)
8. edge(2,4)
9. edge(2,5)
10. edge(3,4)
11. edge(4,6)

for each edge $(u,v) \in G.E$, taken in nondecreasing order by weight

iteration = 1, $(u,v) \rightarrow (0,1)$

if Find-Set (0) \neq Find-Set (1) $\rightarrow \{0\} \neq \{1\}$ ✓

$A = A \cup \{(0,1)\} \rightarrow A = \{\text{edge}(0,1)\}$

Union (0,1) $\rightarrow \{0,1\}$

iteration = 2, $(u,v) \rightarrow (0,3)$

if Find-Set (0) \neq Find-Set (3) $\rightarrow \{0,1\} \neq \{3\}$ ✓

$A = A \cup \{(0,3)\} \rightarrow A = \{\text{edge}(0,1), \text{edge}(0,3)\}$

Union (0,3) $\rightarrow \{0,1,3\}$

iteration = 3, $(u,v) \rightarrow (1,2)$

if Find-Set (1) \neq Find-Set (2) $\rightarrow \{0,1,3\} \neq \{2\}$ ✓

$A = A \cup \{(1,2)\} \rightarrow A = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2)\}$

Union (1,2) $\rightarrow \{0,1,2,3\}$

iteration = 4, $(u,v) \rightarrow (1,3)$

if Find-Set (1) \neq Find-Set (3) $\rightarrow \{0,1,2,3\} \neq \{0,1,2,3\}$ ✗

iteration = 5, $(u,v) \rightarrow (1,5)$

if Find-Set (1) \neq Find-Set (5) $\rightarrow \{0,1,2,3\} \neq \{5\}$ ✓

$A = A \cup \{(1,5)\} \rightarrow A = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,5)\}$

Union (1,2) $\rightarrow \{0,1,2,3,5\}$

iteration = 6, $(u,v) \rightarrow (1,6)$

if Find-Set (1) \neq Find-Set (6) $\rightarrow \{0,1,2,3,5\} \neq \{6\}$ ✓

$A = A \cup \{(1,6)\} \rightarrow A = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,5), \text{edge}(1,6)\}$

Union (1,2) $\rightarrow \{0,1,2,3,5,6\}$

iteration = 7, $(u,v) \rightarrow (2,3)$

if Find-Set (2) \neq Find-Set (3) $\rightarrow \{0,1,2,3,5,6\} \neq \{0,1,2,3,5,6\}$ ✗

iteration = 8, $(u,v) \rightarrow (2,4)$

if Find-Set (2) \neq Find-Set (4) $\rightarrow \{0,1,2,3,5,6\} \neq \{4\}$ ✓

$A = A \cup \{(2,4)\} \rightarrow A = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,5), \text{edge}(1,6), \text{edge}(2,4)\}$

Union (1,2) $\rightarrow \{0,1,2,3,4,5,6\}$

iteration = 9, $(u,v) \rightarrow (2,5)$

if Find-Set (2) \neq Find-Set (5) $\rightarrow \{0,1,2,3,4,5,6\} \neq \{0,1,2,3,4,5,6\}$ ✗

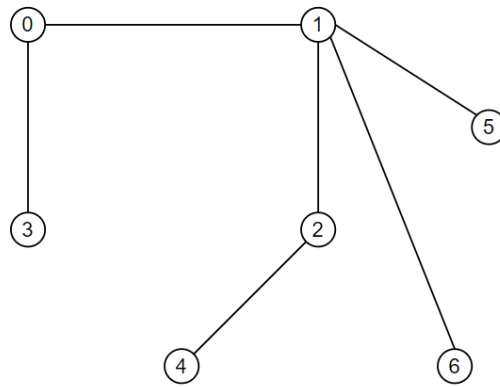
iteration = 10, $(u,v) \rightarrow (3,4)$

if Find-Set (3) \neq Find-Set (4) $\rightarrow \{0,1,2,3,4,5,6\} \neq \{0,1,2,3,4,5,6\}$ ✗

iteration = 11, $(u,v) \rightarrow (4,6)$

if Find-Set (4) \neq Find-Set (6) $\rightarrow \{0,1,2,3,4,5,6\} \neq \{0,1,2,3,4,5,6\}$ ✗

return $A \rightarrow \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,5), \text{edge}(1,6), \text{edge}(2,4)\}$



DRY RUN:

MST-Prim(G, w, r) $\rightarrow G \rightarrow G.V = \{0, 1, 2, 3, 4, 5, 6\}$ & $G.E = \{\text{edge}(0,1), \text{edge}(0,3), \text{edge}(1,2), \text{edge}(1,3), \text{edge}(1,5), \text{edge}(1,6), \text{edge}(2,3), \text{edge}(2,4), \text{edge}(2,5), \text{edge}(3,4), \text{edge}(4,6)\}$, $w \rightarrow$ the edge weights of the graph, all of which are equal to 3 and $r \rightarrow$ root node or root vertex, let's take **vertex 0**.

for each $u \in G.V$

$u.\text{key} = \infty$

$u.\pi = \text{NIL}$

key

0	1	2	3	4	5	6
∞	∞	∞	∞	∞	∞	∞

π

0	1	2	3	4	5	6
NIL	NIL	NIL	NIL	NIL	NIL	NIL

$r.\text{key} = 0 \rightarrow 0.\text{key} = 0$

key

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

$Q = G.V \rightarrow Q = \{0,1,2,3,4,5,6\}$

while $Q \neq \emptyset \rightarrow Q = \{0,1,2,3,4,5,6\}$ ✓

$u = \text{Extract-Min}(Q) \rightarrow u = 0 \rightarrow Q = \{1,2,3,4,5,6\}$

for each $v \in G.\text{adj}[u] \rightarrow G.\text{adj}[0] \rightarrow 1, 3$

iteration = 1, v = 1

if $1 \in Q$ and $w(0,1) < 1.\text{key} \rightarrow 3 < \infty$ ✓

1. $\pi = 0$

1. $\text{key} = w(0,1) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	∞	∞	∞	∞	∞

π

0	1	2	3	4	5	6
NIL	0	NIL	NIL	NIL	NIL	NIL

iteration = 2, v = 3

if $3 \in Q$ and $w(0,3) < 3.key \rightarrow 3 < \infty$ ✓

3. $\pi = 0$

3.key = $w(0,3) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	∞	3	∞	∞	∞

π

0	1	2	3	4	5	6
NIL	0	NIL	0	NIL	NIL	NIL

while $Q \neq \emptyset \rightarrow Q = \{1,2,3,4,5,6\}$ ✓

u = Extract-Min(Q) $\rightarrow u = 1 \rightarrow Q = \{2,3,4,5,6\}$

for each $v \in G.adj[u] \rightarrow G.adj[1] \rightarrow 2, 3, 5, 6$

iteration = 1, v = 2

if $2 \in Q$ and $w(1,2) < 2.key \rightarrow 3 < \infty$ ✓

2. $\pi = 1$

2.key = $w(1,2) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	3	3	∞	∞	∞

π

0	1	2	3	4	5	6
NIL	0	1	0	NIL	NIL	NIL

iteration = 2, v = 3

if $3 \in Q$ and $w(1,3) < 3.key \rightarrow 3 < 3$ ✗

iteration = 3, v = 5

if $5 \in Q$ and $w(1,5) < 5.key \rightarrow 3 < \infty$ ✓

5. $\pi = 1$

5.key = $w(1,5) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	3	3	∞	3	∞

π

0	1	2	3	4	5	6
NIL	0	1	0	NIL	1	NIL

iteration = 4, v = 6

if $6 \in Q$ and $w(1,6) < 6.key \rightarrow 3 < \infty$ ✓

6. $\pi = 1$

6.key = $w(1,6) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	3	3	∞	3	3

π

0	1	2	3	4	5	6
NIL	0	1	0	NIL	1	1

while $Q \neq \emptyset \rightarrow Q = \{2,3,4,5,6\}$ ✓
 $u = \text{Extract-Min}(Q) \rightarrow u = 2 \rightarrow Q = \{3,4,5,6\}$
 for each $v \in G.\text{adj}[u] \rightarrow G.\text{adj}[2] \rightarrow 1, 3, 4, 5$
 iteration = 1, $v = 1$
 if $1 \in Q$ ✗
 iteration = 2, $v = 3$
 if $3 \in Q$ and $w(2,3) < 3.\text{key} \rightarrow 3 < 3$ ✗
 iteration = 3, $v = 4$
 if $4 \in Q$ and $w(2,4) < 4.\text{key} \rightarrow 3 < \infty$ ✓
 4. $\pi = 2$
 4. $\text{key} = w(2,4) \rightarrow 3$

key

0	1	2	3	4	5	6
0	3	3	3	3	3	3

π

0	1	2	3	4	5	6
NIL	0	1	0	2	1	1

iteration = 4, $v = 5$
 if $5 \in Q$ and $w(2,5) < 5.\text{key} \rightarrow 3 < 3$ ✗

while $Q \neq \emptyset \rightarrow Q = \{3,4,5,6\}$ ✓
 $u = \text{Extract-Min}(Q) \rightarrow u = 3 \rightarrow Q = \{4,5,6\}$
 for each $v \in G.\text{adj}[u] \rightarrow G.\text{adj}[3] \rightarrow 0, 1, 2, 4$
 iteration = 1, $v = 0$
 if $0 \in Q$ ✗
 iteration = 2, $v = 1$
 if $1 \in Q$ ✗
 iteration = 3, $v = 2$
 if $2 \in Q$ ✗
 iteration = 4, $v = 4$
 if $4 \in Q$ and $w(3,4) < 4.\text{key} \rightarrow 3 < 3$ ✗

while $Q \neq \emptyset \rightarrow Q = \{4,5,6\}$ ✓
 $u = \text{Extract-Min}(Q) \rightarrow u = 4 \rightarrow Q = \{5,6\}$
 for each $v \in G.\text{adj}[u] \rightarrow G.\text{adj}[4] \rightarrow 2, 3, 6$
 iteration = 1, $v = 2$
 if $2 \in Q$ ✗
 iteration = 2, $v = 3$
 if $3 \in Q$ ✗
 iteration = 3, $v = 6$
 if $6 \in Q$ and $w(4,6) < 6.\text{key} \rightarrow 3 < 3$ ✗

while $Q \neq \emptyset \rightarrow Q = \{5,6\}$ ✓
 $u = \text{Extract-Min}(Q) \rightarrow u = 5 \rightarrow Q = \{6\}$
 for each $v \in G.\text{adj}[u] \rightarrow G.\text{adj}[5] \rightarrow 1, 2$
 iteration = 1, $v = 1$
 if $1 \in Q$ ✗
 iteration = 2, $v = 2$
 if $2 \in Q$ ✗

while $Q \neq \emptyset \rightarrow Q = \{6\}$ ✓
 u = Extract-Min(Q) $\rightarrow u = 6 \rightarrow Q = \{\}$
 for each $v \in G.adj[u] \rightarrow G.adj[6] \rightarrow 1, 4$
 iteration = 1, v = 1
 if $1 \in Q$ ✗
 iteration = 2, v = 4
 if $4 \in Q$ ✗

while $Q \neq \emptyset \rightarrow Q = \{\}$ ✗

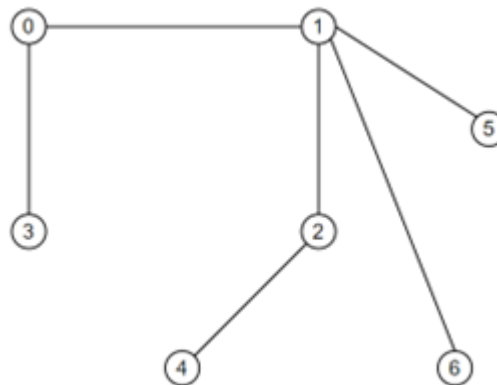
Final key and π array:

key

0	1	2	3	4	5	6
0	3	3	3	3	3	3

π

0	1	2	3	4	5	6
NIL	0	1	0	2	1	1



(e)

No, I'm afraid I have to disagree with him. If a problem has any one of the following properties below, then we can use the Dynamic Programming technique-

1. Overlapping Subproblem
2. Optimal Substructure

But, here, none of the algorithms have this property. So, they do not use the Dynamic Programming design technique.

Kruskal and Prim use the greedy algorithm technique that finds the local optimum in the hopes of finding a global optimum. A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. That's how Kruskal and Prim generate the minimum spanning tree (MST). Thus, we can say that Kruskal and Prim use the greedy algorithm technique.

PROBLEM 5 (FACEBOOK) ANSWER

(a)

The two appropriate algorithms for this problem are-

1. Breadth First Search (BFS)
2. Print Path

BFS (G, s)

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

PRINT-PATH(G, s, v)

```
1  if  $v == s$ 
2      print  $s$ 
3  elseif  $v.\pi == NIL$ 
4      print "no connection from"  $s$  "to"  $v$  "exists"
5  else PRINT-PATH( $G, s, v.\pi$ )
6      print  $v$ 
```


(b)

DRY RUN:

BFS(G, s) → G → G.V = {0, 1, 2, 3, 4, 5, 6} & **G.E** = {edge(0,1), edge(0,3), edge(1,2), edge(1,3), edge(1,5), edge(1,6), edge(2,3), edge(2,4), edge(2,5), edge(3,4), edge(4,6)} and **s** → vertex 3.

for each vertex $u \in G.V - \{s\}$ or, {3}

u.color = WHITE

u.d = ∞

u. π = NIL

color

0	1	2	3	4	5	6
WHITE	WHITE	WHITE		WHITE	WHITE	WHITE

d

0	1	2	3	4	5	6
∞	∞	∞		∞	∞	∞

π

0	1	2	3	4	5	6
NIL	NIL	NIL		NIL	NIL	NIL

3.color = GRAY

3.d = 0

3. π = NIL

color

0	1	2	3	4	5	6
WHITE	WHITE	WHITE	GRAY	WHITE	WHITE	WHITE

d

0	1	2	3	4	5	6
∞	∞	∞	0	∞	∞	∞

π

0	1	2	3	4	5	6
NIL	NIL	NIL	NIL	NIL	NIL	NIL

Q = \emptyset

Enqueue(Q,s) → (Q,3) → Q = {3}

while Q $\neq \emptyset$ → Q = {3} ✓

u = Dequeue(Q) → u = 3 → Q = {}

for each v \in G.adj[u] → G.adj[3] → 0, 1, 2, 4

iteration = 1, v = 0

if 0.color == WHITE ✓

0.color = GRAY

0.d = 3.d + 1 → 0+1 → 1

0. π = 3

Enqueue(Q,v) → (Q,0) → Q = {0}

color

0	1	2	3	4	5	6
GRAY	WHITE	WHITE	GRAY	WHITE	WHITE	WHITE

d

0	1	2	3	4	5	6
1	∞	∞	0	∞	∞	∞

π

0	1	2	3	4	5	6
3	NIL	NIL	NIL	NIL	NIL	NIL

iteration = 2, v = 1

if 1.color == WHITE ✓

1.color = GRAY

1.d = 3.d + 1 \rightarrow 0+1 \rightarrow 1

1. π = 3

Enqueue(Q,v) \rightarrow (Q,1) \rightarrow Q = {0,1}

color

0	1	2	3	4	5	6
GRAY	GRAY	WHITE	GRAY	WHITE	WHITE	WHITE

d

0	1	2	3	4	5	6
1	1	∞	0	∞	∞	∞

π

0	1	2	3	4	5	6
3	3	NIL	NIL	NIL	NIL	NIL

iteration = 3, v = 2

if 2.color == WHITE ✓

2.color = GRAY

2.d = 3.d + 1 \rightarrow 0+1 \rightarrow 1

2. π = 3

Enqueue(Q,v) \rightarrow (Q,2) \rightarrow Q = {0,1,2}

color

0	1	2	3	4	5	6
GRAY	GRAY	GRAY	GRAY	WHITE	WHITE	WHITE

d

0	1	2	3	4	5	6
1	1	1	0	∞	∞	∞

π

0	1	2	3	4	5	6
3	3	3	NIL	NIL	NIL	NIL

iteration = 4, v = 4

if 4.color == WHITE ✓

4.color = GRAY

4.d = 3.d + 1 \rightarrow 0+1 \rightarrow 1

4. π = 3

Enqueue(Q,v) \rightarrow (Q,4) \rightarrow Q = {0,1,2,4}

color

0	1	2	3	4	5	6
GRAY	GRAY	GRAY	GRAY	GRAY	WHITE	WHITE

d

0	1	2	3	4	5	6
1	1	1	0	1	∞	∞

π

0	1	2	3	4	5	6
3	3	3	NIL	3	NIL	NIL

3.color = BLACK

color

0	1	2	3	4	5	6
GRAY	GRAY	GRAY	BLACK	GRAY	WHITE	WHITE

while $Q \neq \emptyset \rightarrow Q = \{0,1,2,4\}$ ✓
 u = Dequeue(Q) $\rightarrow u = 0 \rightarrow Q = \{1,2,4\}$
 for each $v \in G.adj[u] \rightarrow G.adj[0] \rightarrow 1, 3$
 iteration = 1, v = 1
 if 1.color == WHITE ✗
 iteration = 2, v = 3
 if 3.color == WHITE ✗

0.color = BLACK

color

0	1	2	3	4	5	6
BLACK	GRAY	GRAY	BLACK	GRAY	WHITE	WHITE

while $Q \neq \emptyset \rightarrow Q = \{1,2,4\}$ ✓
 u = Dequeue(Q) $\rightarrow u = 1 \rightarrow Q = \{2,4\}$
 for each $v \in G.adj[u] \rightarrow G.adj[1] \rightarrow 0, 2, 3, 5, 6$
 iteration = 1, v = 0
 if 0.color == WHITE ✗
 iteration = 2, v = 2
 if 2.color == WHITE ✗
 iteration = 3, v = 3
 if 3.color == WHITE ✗
 iteration = 4, v = 5
 if 5.color == WHITE ✓
 5.color = GRAY
 5.d = 1.d + 1 $\rightarrow 1+1 \rightarrow 2$
 5. π = 1
 Enqueue(Q,v) $\rightarrow (Q,5) \rightarrow Q = \{2,4,5\}$

color

0	1	2	3	4	5	6
BLACK	GRAY	GRAY	BLACK	GRAY	GRAY	WHITE

d

0	1	2	3	4	5	6
1	1	1	0	1	2	∞

π

0	1	2	3	4	5	6
3	3	3	NIL	3	1	NIL

iteration = 5, v = 6

if 6.color == WHITE

✓

6.color = GRAY

6.d = 1.d + 1 \rightarrow 1+1 \rightarrow 2

6. π = 1

Enqueue(Q,v) \rightarrow (Q,6) \rightarrow Q = {2,4,5,6}

color

0	1	2	3	4	5	6
BLACK	GRAY	GRAY	BLACK	GRAY	GRAY	GRAY

d

0	1	2	3	4	5	6
1	1	1	0	1	2	2

π

0	1	2	3	4	5	6
3	3	3	NIL	3	1	1

1.color = BLACK

color

0	1	2	3	4	5	6
BLACK	BLACK	GRAY	BLACK	GRAY	GRAY	GRAY

while Q $\neq \emptyset$ \rightarrow Q = {2,4,5,6}

✓

u = Dequeue(Q) \rightarrow u = 2 \rightarrow Q = {4,5,6}

for each v \in G.adj[u] \rightarrow G.adj[2] \rightarrow 1, 3, 4, 5

iteration = 1, v = 1

if 1.color == WHITE

✗

iteration = 2, v = 3

if 3.color == WHITE

✗

iteration = 3, v = 4

if 4.color == WHITE

✗

iteration = 4, v = 5

if 5.color == WHITE

✗

2.color = BLACK

color

0	1	2	3	4	5	6
BLACK	BLACK	BLACK	BLACK	GRAY	GRAY	GRAY

while $Q \neq \emptyset \rightarrow Q = \{4,5,6\}$ ✓
 u = Dequeue(Q) $\rightarrow u = 4 \rightarrow Q = \{5,6\}$
 for each $v \in G.adj[u] \rightarrow G.adj[4] \rightarrow 2, 3, 6$
 iteration = 1, v = 2
 if 2.color == WHITE ✗

 iteration = 2, v = 3
 if 3.color == WHITE ✗

 iteration = 3, v = 6
 if 6.color == WHITE ✗

 4.color = BLACK

color

0	1	2	3	4	5	6
BLACK	BLACK	BLACK	BLACK	BLACK	GRAY	GRAY

while $Q \neq \emptyset \rightarrow Q = \{5,6\}$ ✓
 u = Dequeue(Q) $\rightarrow u = 5 \rightarrow Q = \{6\}$
 for each $v \in G.adj[u] \rightarrow G.adj[5] \rightarrow 1, 2$
 iteration = 1, v = 1
 if 1.color == WHITE ✗

 iteration = 2, v = 2
 if 2.color == WHITE ✗

 5.color = BLACK

color

0	1	2	3	4	5	6
BLACK	BLACK	BLACK	BLACK	BLACK	BLACK	GRAY

while $Q \neq \emptyset \rightarrow Q = \{6\}$ ✓
 u = Dequeue(Q) $\rightarrow u = 6 \rightarrow Q = \{\}$
 for each $v \in G.adj[u] \rightarrow G.adj[6] \rightarrow 1, 4$
 iteration = 1, v = 1
 if 1.color == WHITE ✗

 iteration = 2, v = 4
 if 4.color == WHITE ✗

 6.color = BLACK

color

0	1	2	3	4	5	6
BLACK	BLACK	BLACK	BLACK	BLACK	BLACK	BLACK

while $Q \neq \emptyset \rightarrow Q = \{\}$ ✗

Final color, d and π array:

color

0	1	2	3	4	5	6
BLACK	BLACK	BLACK	BLACK	BLACK	BLACK	BLACK

d

0	1	2	3	4	5	6
1	1	1	0	1	2	2

π

0	1	2	3	4	5	6
3	3	3	NIL	3	1	1

DRY RUN:

Print-Path(G, s, v) \rightarrow G \rightarrow G.V = {0, 1, 2, 3, 4, 5, 6} & G.E = {edge(0,1), edge(0,3), edge(1,2), edge(1,3), edge(1,5), edge(1,6), edge(2,3), edge(2,4), edge(2,5), edge(3,4), edge(4,6)}, s \rightarrow vertex 3 and v \rightarrow vertex 5.

```

if v == s  $\rightarrow$  5 == 3 ✗
elseif v. $\pi$  == NIL  $\rightarrow$  5.  $\pi$  == NIL ✗
else Print-Path(G, s, v. $\pi$ )  $\rightarrow$  G.V, G.E & s  $\rightarrow$  3 & v. $\pi$   $\rightarrow$  5.  $\pi$   $\rightarrow$  1
    if v == s  $\rightarrow$  1 == 3 ✗
    elseif v. $\pi$  == NIL  $\rightarrow$  1.  $\pi$  == NIL ✗
    else Print-Path(G, s, v. $\pi$ )  $\rightarrow$  G.V, G.E & s  $\rightarrow$  3 & v. $\pi$   $\rightarrow$  1.  $\pi$   $\rightarrow$  3
        if v == s  $\rightarrow$  3 == 3 ✓
            print s  $\rightarrow$  s = 3  $\rightarrow$  Output : 3
        print v  $\rightarrow$  v = 1  $\rightarrow$  Output : 3, 1
    print v  $\rightarrow$  v = 5  $\rightarrow$  Output : 3, 1, 5

```

Final Output: 3, 1, 5

(c)

Yes, another algorithm could be used instead of the first one I wrote in response to part (a). This algorithm's name is Depth First Search (DFS).

In many ways, DFS is different from BFS. Such as-

1. DFS algorithm is a recursive algorithm that uses the idea of the Backtracking technique, but in BFS, there is no concept of Backtracking.
2. DFS creates the tree sub-tree by sub-tree, but BFS creates the tree level by level.
3. DFS uses the concept of Stack (Last In First Out), but BFS uses the Queue (First In First Out).
4. DFS requires less memory, but BFS requires more memory.

These are the many ways in which DFS differs from BFS.

(d)

A graph can be represented using the following two ways-

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix: An adjacency matrix can be considered a table with rows and columns. Both the row and column labels represent the vertices or nodes of a graph.

Adjacency matrix represents a graph as $n \times n$ matrix **A** (here, n is the number of vertices/nodes):

$$A[i,j] = 1 \text{ or, weight of edge, if edge } (i, j) \in E$$

$$= 0 \text{ if edge } (i, j) \notin E$$

	0	1	2	3	4	5	6
0	0	1	0	1	0	0	0
1	1	0	1	1	0	1	1
2	0	1	0	1	1	1	0
3	1	1	1	0	1	0	0
4	0	0	1	1	0	0	1
5	0	1	1	0	0	0	0
6	0	1	0	0	1	0	0

Adjacency List: Every vertex is represented as a node object in the adjacency list representation of a graph. The node may either contain data or a reference to a linked list. This linked list provides a list of all nodes adjacent to the current node.

The adjacency list is a list of adjacent vertices. For each vertex $v \in V$, store a list of vertices adjacent to v.

0 → 1 × → 3 ×

1 → 0 × → 2 × → 3 × → 5 × → 6 ×

2 → 1 × → 3 × → 4 × → 5 ×

3 → 0 × → 1 × → 2 × → 4 ×

4 → 2 × → 3 × → 6 ×

5 → 1 × → 2 ×

6 → 1 × → 4 ×

PROBLEM 6 (COMPUTER NETWORK) ANSWER

(a)

No, we don't need to use the Bellman Ford Algorithm or the Floyd Warshall algorithm if we want to determine the maximum speed from node E to node A.

We can use Dijkstra's Shortest Path algorithm to solve this problem. Dijkstra's algorithm solves the shortest path problem for any weighted graph with non-negative weights. It can handle graphs consisting of cycles, but negative weights will cause this algorithm to produce incorrect results.

On the other hand, the Bellman-Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm, but it can work with graphs in which edges can have negative weights.

And, the Floyd-Warshall algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. It also works with negative weights but fails when there are negative cycles.

Since this graph has no negative weight, and we just need to find the shortest path from node E to node A, then Dijkstra's Shortest Path algorithm will be sufficient to solve this problem.

(b)

Dijkstra's algorithm to solve the above problem:

```
#include <iostream>
#define INFINITY 9999
#define MAX 5
using namespace std;

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;

    // Creating cost matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (Graph[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = Graph[i][j];

    for (j = 0; j < n; j++) {
        distance[j] = cost[start][j];
        pred[j] = start;
        visited[j] = 0;
    }

    distance[start] = 0;
    visited[start] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;

        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }

        visited[nextnode] = 1;
```



```

    for (i = 0; i < n; i++)
        if (!visited[i])
            if (mindistance + cost[nextnode][i] < distance[i]) {
                distance[i] = mindistance + cost[nextnode][i];
                pred[i] = nextnode;
            }
        count++;
    }

    // Printing the distance
    cout << "\nThe maximum speed path from node E to node A is " << distance[0] <<
endl;

}

int main() {
    int Graph[MAX][MAX], i, j, n, u;
    n = 5;

    //let's assume {A,B,C,D,E} = {0,1,2,3,4}

    Graph[0][0] = 0;
    Graph[0][1] = 3;
    Graph[0][2] = 1;
    Graph[0][3] = 0;
    Graph[0][4] = 0;

    Graph[1][0] = 3;
    Graph[1][1] = 0;
    Graph[1][2] = 7;
    Graph[1][3] = 5;
    Graph[1][4] = 1;

    Graph[2][0] = 1;
    Graph[2][1] = 7;
    Graph[2][2] = 0;
    Graph[2][3] = 2;
    Graph[2][4] = 0;

    Graph[3][0] = 0;
    Graph[3][1] = 5;
    Graph[3][2] = 2;
    Graph[3][3] = 0;
    Graph[3][4] = 7;

    Graph[4][0] = 0;
    Graph[4][1] = 1;
    Graph[4][2] = 0;
    Graph[4][3] = 7;
    Graph[4][4] = 0;

    u = 4;
    Dijkstra(Graph, n, u);

```

```

return 0;
}

```

(c)

DRY RUN:

Dijkstra(int Graph[MAX][MAX], int n, int start) → Graph, n = 5, start = 4 (node E)

```

cost[5][5], distance[5], pred[5], visited[5]
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (Graph[i][j] == 0)
            cost[i][j] = INFINITY
        else
            cost[i][j] = Graph[i][j]

```

Cost[i,j]	A / 0	B / 1	C / 2	D / 3	E / 4
A / 0	INF	3	1	INF	INF
B / 1	3	INF	7	5	1
C / 2	1	7	INF	2	INF
D / 3	INF	5	2	INF	7
E / 4	INF	1	INF	7	INF

```

for (j = 0; j < n; j++)
    distance[j] = cost[4][j]
    pred[j] = 4
    visited[j] = 0

```

distance

A / 0	B / 1	C / 2	D / 3	E / 4
INF	1	INF	7	INF

pred

A / 0	B / 1	C / 2	D / 3	E / 4
4	4	4	4	4

visited

A / 0	B / 1	C / 2	D / 3	E / 4
0	0	0	0	0

```

distance[4] = 0
visited[4] = 1
count = 1

```

distance

A / 0	B / 1	C / 2	D / 3	E / 4
INF	1	INF	7	0

visited

A / 0	B / 1	C / 2	D / 3	E / 4
0	0	0	0	1

while (count < n - 1) $\rightarrow 1 < 5-1 \rightarrow 1 < 4$

mindistance = INFINITY;

for (i = 0; i < 5; i++)

if (distance[0] < mindistance && !visited[0]) $\rightarrow \text{INF} < \text{INF}$ ✗

for (i = 1; i < 5; i++)

if (distance[1] < mindistance && !visited[1]) $\rightarrow 1 < \text{INF} \ \&\& \ !\text{visited}[1]$ ✓

mindistance = distance[1] $\rightarrow 1$

nextnode = 1

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

if (distance[2] < mindistance && !visited[2]) $\rightarrow \text{INF} < 1$ ✗

for (i = 3; i < 5; i++)

if (distance[3] < mindistance && !visited[3]) $\rightarrow 7 < 1$ ✗

for (i = 4; i < 5; i++)

if (distance[4] < mindistance && !visited[4]) $\rightarrow !\text{visited}[4]$ ✗

for (i = 5; i < 5; i++) ✗

visited[nextnode] = 1 $\rightarrow \text{visited}[1] = 1$

visited

A / 0	B / 1	C / 2	D / 3	E / 4
0	1	0	0	1

for (i = 0; i < 5; i++)

if (!visited[0]) ✓

if (mindistance + cost[1][0] < distance[0]) $\rightarrow 1 + 3 < \text{INF}$ ✓

distance[0] = mindistance + cost[1][0] $\rightarrow 1+3 = 4$

pred[0] = 1

distance

A / 0	B / 1	C / 2	D / 3	E / 4
4	1	INF	7	0

pred

A / 0	B / 1	C / 2	D / 3	E / 4
1	4	4	4	4

for (i = 1; i < 5; i++)

if (!visited[1]) ✗

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

if (!visited[2]) ✓

if (mindistance + cost[1][2] < distance[2]) $\rightarrow 1 + 7 < \text{INF}$ ✓

distance[2] = mindistance + cost[1][2] $\rightarrow 1+7 = 8$

pred[2] = 1

distance

A / 0	B / 1	C / 2	D / 3	E / 4
4	1	8	7	0

pred

A / 0	B / 1	C / 2	D / 3	E / 4
1	4	1	4	4

for (i = 3; i < 5; i++)

if (!visited[3]) ✓

if (mindistance + cost[1][3] < distance[3]) → 1 + 5 < 7 ✓

distance[3] = mindistance + cost[1][3] → 1 + 5 = 6

pred[3] = 1

distance

A / 0	B / 1	C / 2	D / 3	E / 4
4	1	8	6	0

pred

A / 0	B / 1	C / 2	D / 3	E / 4
1	4	1	1	4

for (i = 4; i < 5; i++)

if (!visited[4]) ✗

for (i = 5; i < 5; i++) ✗

count++; → count = 2

while (count < n - 1) → 2 < 5 - 1 → 2 < 4

mindistance = INFINITY;

for (i = 0; i < 5; i++)

if (distance[0] < mindistance && !visited[0]) → 4 < INF && !visited[0] ✓

mindistance = distance[0] → 4

nextnode = 0

for (i = 1; i < 5; i++)

if (distance[1] < mindistance && !visited[1]) → !visited[1] ✗

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

if (distance[2] < mindistance && !visited[2]) → 8 < 4 ✗

for (i = 3; i < 5; i++)

if (distance[3] < mindistance && !visited[3]) → 6 < 4 ✗

for (i = 4; i < 5; i++)

if (distance[4] < mindistance && !visited[4]) → !visited[4] ✗

for (i = 5; i < 5; i++) ✗

visited[nextnode] = 1 → visited[0] = 1

visited

A / 0	B / 1	C / 2	D / 3	E / 4
1	1	0	0	1

for (i = 0; i < 5; i++)

if (!visited[0]) ✗

for (i = 1; i < 5; i++)

if (!visited[1]) ✗

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

if (!visited[2]) ✓

if (mindistance + cost[0][2] < distance[2]) → 4 + 1 < 8 ✓

distance[2] = mindistance + cost[0][2] → 4 + 1 = 5

pred[2] = 0

distance

A / 0	B / 1	C / 2	D / 3	E / 4
4	1	5	6	0

pred

A / 0	B / 1	C / 2	D / 3	E / 4
1	4	0	1	4

```

for (i = 3; i < 5; i++)
    if (!visited[3]) ✓
        if (mindistance + cost[0][3] < distance[3]) → 4 + INF < 6 ✗
for (i = 4; i < 5; i++)
    if (!visited[4]) ✗
for (i = 5; i < 5; i++) ✗

```

count++; → count = 3

while (count < n - 1) → 3 < 5-1 → 3 < 4

mindistance = INFINITY;

```

for (i = 0; i < 5; i++)
    if (distance[0] < mindistance && !visited[0]) → !visited[0] ✗
for (i = 1; i < 5; i++)
    if (distance[1] < mindistance && !visited[1]) → !visited[1] ✗
for (i = 2; i < 5; i++)
    if (distance[2] < mindistance && !visited[2]) → 5 < INF && !visited[2] ✗
        mindistance = distance[2] → 5
        nextnode = 2
for (i = 3; i < 5; i++)
    if (distance[3] < mindistance && !visited[3]) → 6 < 5 ✗
for (i = 4; i < 5; i++)
    if (distance[4] < mindistance && !visited[4]) → !visited[4] ✗
for (i = 5; i < 5; i++) ✗

```

visited[nextnode] = 1 → visited[2] = 1

visited

A / 0	B / 1	C / 2	D / 3	E / 4
1	1	1	0	1

```

for (i = 0; i < 5; i++)
    if (!visited[0]) ✗
for (i = 1; i < 5; i++)
    if (!visited[1]) ✗
for (i = 2; i < 5; i++)
    if (!visited[2]) ✗
for (i = 3; i < 5; i++)
    if (!visited[3]) ✓
        if (mindistance + cost[2][3] < distance[3]) → 5 + 2 < 6 ✗
for (i = 4; i < 5; i++)
    if (!visited[4]) ✗
for (i = 5; i < 5; i++) ✗

```

count++; \rightarrow count = 4

while (count < n - 1) \rightarrow 4 < 5-1 \rightarrow 4 < 4 ✗

Final distance, pred and visited array:

distance

A / 0	B / 1	C / 2	D / 3	E / 4
4	1	5	6	0

pred

A / 0	B / 1	C / 2	D / 3	E / 4
1	4	0	1	4

visited

A / 0	B / 1	C / 2	D / 3	E / 4
1	1	1	0	1

cout << "\n\nThe maximum speed path from node E to node A is " << distance[0] << endl;

```
input
The maximum speed path from node E to node A is 4

...Program finished with exit code 0
Press ENTER to exit console.
```

(d)

Yes, I do agree with his. Because the algorithm I have written in response to part (b) is Dijkstra's Algorithm, it uses the Greedy algorithm design technique.

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

Dijkstra's algorithm uses weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes. This algorithm is known as the single source shortest path algorithm. That's the principle of the Greedy algorithm. Here, Dijkstra greed to minimize the path.

Thus, we can say that Dijkstra's algorithm uses the Greedy algorithm design technique.

THE END