7

3 2 5 5 4 2 4

Sample Output:

5 (1+1+2+1)

5(2+2+1)

6(2+4+2)

- **Procedure/Program:**

To Solve the Problem of minimizing the total hours spent on NCC activites while adhering to the rule that no student can skip NCC activity for 3 consecutive days, we can use a dynamic Programming approach

n=10

hours = [3, 4, 1, 1, 2, 3, 2, 3, 2, 1]

dynamic programming Array calculation

1) Initialize dP:

d[0] = hours [0] = 3

d[1] = hours[0] + hours [1] = 3+4 = 7

d[2] = hours [0] + hours [2] + hours[1]

= 3+4+1 = 8

Fill the dP Array

for day 3

dP [3]

```
#include <stdio.h>
#include <limits.h>
int minHours(int n, int hours[]){
    if(n==0) return 0;
    if(n==1) return hours[0];
    if(n==2) return hours[1];
    int dP[n];
    dP[0] = hours[0];
    dP[1] = hours[1];
    dP[2] = hours[2];
for(int i=3; i<n; i++){
    dP[i]= hours[i]+(dP[i-1] < dP[i-2] ? dP[i-1]:
                dP[i-2]);
if(i>2) dP[i] = dP[i] < dP[i-3] + hours[i] ?
                dP[i] : dP[i-3];
    }
return dP[n-1];
}
int main(){
    int n= 10;
    int hours[]= {3, 4, 1, 1, 2, 3, 2, 3, 2, 1};
    Printf("%d\n", minHours(n, hours));
    return 0; }
```

- **Data and Results:**

Input: 10, [3, 4, 1, 1, 2, 3, 2, 3, 2, 1]

output: 5 (Total hours spent)

- **Analysis and Inferences:**

Dynamic Programming minimizes hours, ensuring NCC participation without consecutive skips.

In-Lab:

1. You are given a set of keys and their frequencies, representing the number of times each key is accessed. The goal is to construct a binary search tree with these keys in such a way that the average search cost is minimized. This involves finding an optimal way to arrange the keys in the tree using dynamic programming techniques.

Input:

Keys: [10, 20, 30, 40, 50]

Frequencies: [4, 2, 6, 3, 5]

Output:

Minimum Average Search Cost: 34

- **Procedure/Program:**

```
#include<stdio.h>
#define INF 999999
int mincost(int freq[], int n){
int cost[n][n], sum[n][n];
```

```
for (int i=0; i<n; i++)
    cost[i][i] = sum[i][i] = freq[i];

for(int l=2; l<=n; l++)
for(int i=0; i<=n-l; i++){
    int j = i + l - 1;
    cost[i][j] = INF;
sum[i][j] = sum[i][j-1] + freq[j];
  for(int r=i; r<=j; r++)
cost[i][j] = (r > i ? cost[i][r-1] : 0) +
            (r < j ? cost[r+1][j] : 0) + sum[i][j]
            < cost[i][j] ? (r > i ?
cost[i][r-1] : 0) + (r < j ? cost[r+1][j] : 0)
  + sum[i][j] : cost[i][j];
 }
return cost[0][n-1];
}
int main(){
 int freq[] = {4, 2, 6, 3, 5}, n = 5;
     printf("%d\n", minCost(freq, n));
     }
```

- **Data and Results:**

keys: [10, 20, 30, 40, 50]; frequencies: [4, 2, 6, 3, 5];

minimum average search cost: 34

- **Analysis and Inferences:**

Analysis

Dynamic Programming minimizes the

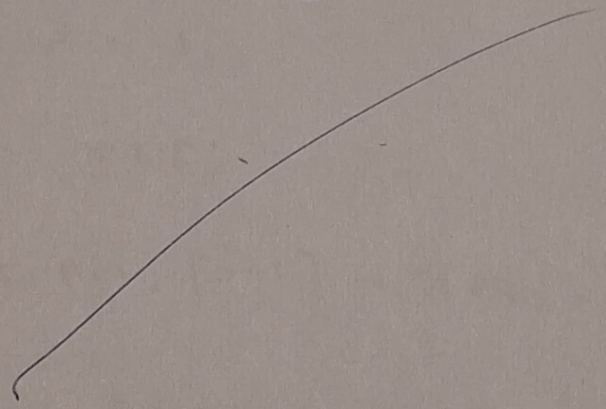
search cost by optimizing tree structure

inferences

2

optimal Binary search tree reduces search

operations for frequently accessed keys.



2. The 0/1 Knapsack problem involves selecting items with given weights and values to maximize the total value without exceeding a specified weight limit. Each item can either be included in the knapsack (1) or excluded (0). Dynamic programming is used to solve this problem by breaking it down into simpler subproblems and solving each subproblem only once.

Input:

Weights of Items: [2, 3, 4, 5]

Values of Items: [3, 4, 5, 6]

Knapsack Capacity: 8

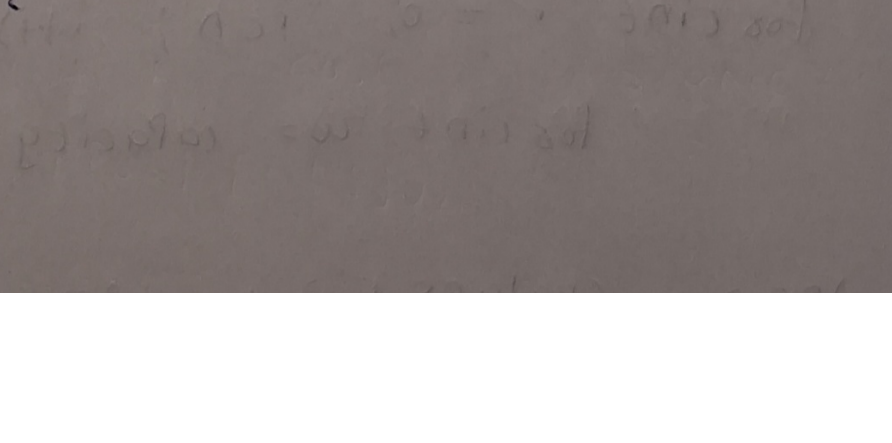
Output:

Maximum Value: 10

- **Procedure/Program:**

```
#include <stdio.h>
int knapSack(int weights[], int values[], int n, int capacity) {
    int dP[capacity+1];
for(int w = 0; w <= capacity; w++)
    dP[w] = 0;
for(int i = 0; i<n; i++){
    for(int w= capacity; w>=weights[i]; w--){
dP[w] = (values[i] + dP[w- weights[i]] > dP[w]) ?
```

```
values[i] + dp[w - weights[i]] : dp[w];
    }
  }
  return dp[capacity];
}
int main(){
int weights[] = {2, 3, 4, 5};
int values[] = {3, 4, 5, 6};
int capacity = 8;
int maxValue = knapsack(weights, values,
        4, capacity);
Printf("Maximum value: %d\n", maxvalue);
    return 0;
}
```



• **Data and Results:**

weights [2, 3, 4, 5]

values [3, 4, 5, 6]

maximum value : 10

• **Analysis and Inferences:**

Dynamic programming optimally selects items to maximize knapsack value efficiently.

Post-Lab:

A delivery person needs to visit 4 locations (Home, Office, Grocery Store, Park) with known distances between each pair of locations. The goal is to determine the shortest route that visits each location exactly once and returns to the starting location.

Input:

Locations: [Home, Office, Grocery Store, Park]

Distance Matrix:

0	5	10	15
5	0	8	20
10	8	0	12
15	20	12	0

Output:

Shortest Route: [Home, Office, Grocery Store, Park, Home]

Total Distance: 37

• **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#define locations 4
int distancematrix[locations][locations] = {
```

```
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30}
    {20, 25, 30, 0}
};
int mincost = INT_MAX;
void tsp(int Pos, int visited, int count,
         int cost){
 if(count == locations && Pos == 0){
 if(cost < mincost) mincost = cost;
    return;
}
for(int city = 0; city < locations; city++){
   if(!(visited & (1 << city))){
   tsp(city, visited | (1 << city), count+1,
          cost + 1, cost + distancematrix
              [Pos][city]);
    }
  }
}
   int main(){
```

```
tsp (0, 1, 1, 0);
printf("Total Distance: %d\n", mincost);
return 0;
}
```

- **Data and Results:**

Locations: Home, office, grocery store, park; Total Distance 80

- **Analysis and Inferences:**

Brute-force method computes shortest route effectively for small datasets.

Experiment #8		Student ID	
Date		Student Name	

Sample VIVA-VOCE Questions (In-Lab):

1. Explain the concept of memorization in dynamic programming.
2. How does the 0/1 Knapsack problem differ from the fractional knapsack problem?
3. Describe the Optimal Binary Search Tree (OBST) problem and its significance.
4. What is the main challenge in solving the Travelling Salesman Problem (TSP)?
5. What are the advantages of using dynamic programming over brute-force methods?

1) Memorization stores intermediate results to avoid redundant calculations

2) 0/1 knapsack involves discrete items, fractional allows partial items.

3) OBST minimizes search cost, optimizing node arrangement for queries.

Evaluator Remark (if Any):	Marks Secured: 48 out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

4) TSP challenges is finding shortest route efficiently among permutations.

5) dynamic Programming reduces time complexity by avoiding repeated work.