Experiment **#9**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on problem solving using Dynamic Programming Approach – Scenario1.

Aim/Objective: To understand the concept and implementation of programs on Dynamic Programming approach-based Problems.

Description: The students will understand and able to implement programs on Dynamic Programming Approach based Problems.

Pre-Requisites:

Knowledge: Dynamic Programming approach and its related problems in C/ java/Python.

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

1. Suppose you are given different types of Lego blocks with heights 1 cm, 2 cm, and 5 cm, and you need to build a tower of height N = 7 cm. Determine the total number of unique combinations of blocks that can sum up to 7cm, where the order of blocks does not matter.

Input : N=7

Output : 6

Explanation :

1. 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 cm
2. 1 + 1 + 1 + 1 + 1 + 2 = 7 cm
3. 1 + 1 + 1 + 2 + 2 = 7 cm
4. 1 + 1 + 5 = 7 cm
5. 1 + 2 + 2 + 2 = 7 cm
6. 2 + 5 = 7 cm

- **Procedure/Program**:

```
#include <stdio.h>

int countCombinations(int N, int blocks[], int m) {
    int dp[N + 1];
    for (int i = 0; i <= N; i++)
        dp[i] = 0;
    dp[0] = 1;

    for (int i = 0; i < m; i++) {
        for (int j = blocks[i]; j <= N; j++) {
            dp[j] += dp[j - blocks[i]];
        }
    }
    return dp[N];
}

int main() {
    int N = 7;
    int blocks[] = {1, 2, 5};
    int m = sizeof(blocks) / sizeof(blocks[0]);
    int result = countCombinations(N, blocks, m);
    printf("%d", result);
    return 0;
}
```

- **Data and Results:**

Data

Different block heights: 1 cm, 2 cm, 5 cm available.

Result

Total unique combinations forming 7 cm: 6 distinct ways.

- **Analysis and Inferences:**

Analysis

Dynamic programming approach used for counting valid combinations efficiently.

Inferences

Order-independent combinations significantly reduce redundant calculations in problem-solving.

In-Lab:

1. Emma, working at a wildlife reserve, needs to distribute bird feed equally among several feed stations. She can add feed to several stations in each operation. Calculate the minimum number of operations needed to ensure all stations have the same amount of feed

Example

For Example, stations = [1, 2, 7] stations represent the starting feed amounts. She can add 2 feed units to the first two stations. Now the distribution is [3, 4, 7]. In the next round, she adds 3 units each to the first two stations, evening them out to [6, 7, 7], and finally, one more round of 1 unit to the first station will result in [7, 7, 7]. The total number of rounds required is 3.

Function Description:

equalize_feed has the following parameter(s):

• int stations[n]: the feed amounts to equalize

Returns int: the minimum number of operations required

Sample Input

- **t = 1**: Indicates the number of test cases.
- **n = 3**: Number of stations.
- **stations = [1, 2, 7]**: The positions of the stations.

Sample Output

3

- **Procedure/Program**:

```
#include <stdio.h>

#include <stdlib.h>


int compare(const void *a, const void *b) {

   return (*(int *)a - *(int *)b);

}


int equalize_feed(int stations[], int n) {

   qsort(stations, n, sizeof(int), compare);

   int count = 0;


   while (stations[0] != stations[n - 1]) {

      int diff = stations[n - 1] - stations[0];

      int add = (diff >= 5) ? 5 : (diff >= 2) ? 2 : 1;


      for (int i = 0; i < n - 1; i++) {

         stations[i] += add;

      }
```

```
    qsort(stations, n, sizeof(int), compare);

    count++;

  }

  return count;

}

int main() {

  int t = 1, n = 3;

  int stations[] = {1, 2, 7};

  printf("%d\n", equalize_feed(stations, n));

  return 0;

}
```

- **Data and Results:**

Data

Given bird feed amounts across stations, aim to equalize them.

Result

Minimum operations needed for equal feed distribution across stations.

- **Analysis and Inferences:**

Analysis

Sorted stations show difference, operations add feed to equalize values.

Inferences

Optimal number of operations is determined by largest differences.

Post-Lab:

Suppose you are given different types of Lego blocks with heights 1 cm, 2 cm, and 5 cm, and you need to build a tower of height N = 8 cm. Determine the total number of unique combinations of blocks that can sum up to 8cm, where the order of blocks does not matter.

Input : N=8

Output : 9

Explanation :

1. 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8 cm
2. 1 + 1 + 1 + 1 + 1 + 1 + 2 = 8 cm
3. 1 + 1 + 1 + 2 + 2 = 8 cm
4. 1 + 1 + 2 + 2 + 2 = 8 cm
5. 1 + 2 + 2 + 2 + 1 = 8 cm
6. 2 + 2 + 2 + 2 = 8 cm
7. 1 + 1 + 1 + 5 = 8 cm
8. 1 + 2 + 5 = 8 cm
9. 2 + 6 = 8 cm

- **Procedure/Program**:

```
#include <stdio.h>

int countCombinations(int N, int blocks[], int m) {
   int dp[N + 1];
   for (int i = 0; i <= N; i++)
      dp[i] = 0;
   dp[0] = 1;

   for (int i = 0; i < m; i++) {
      for (int j = blocks[i]; j <= N; j++) {
         dp[j] += dp[j - blocks[i]];
      }
   }
   return dp[N];
}
```

```
int main() {
   int N = 8;
   int blocks[] = {1, 2, 5};
   int m = sizeof(blocks) / sizeof(blocks[0]);
   int result = countCombinations(N, blocks, m);
   printf("Total unique combinations: %d\n", result);
   return 0;
}
```

- **Data and Results:**

Data

Different Lego blocks of heights 1 cm, 2 cm, and 5 cm.

Result

Total unique block combinations for 8 cm tower: **9**.

- **Analysis and Inferences:**

Analysis

Dynamic programming used to count unique unordered combinations efficiently.

Inferences

Smaller blocks contribute significantly to combination count growth.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. Compare Dynamic Programming, Divide & Conquer and Greedy Approaches.

   - **DP:** Overlapping subproblems, stores results (e.g., Knapsack).
   - **D&C:** Independent subproblems, recursive (e.g., Merge Sort).
   - **Greedy:** Locally optimal choice (e.g., Kruskal's).

2. How do you identify and define subproblems in a dynamic programming solution?

   - Divide into smaller, reusable problems.
   - Define state representation.

3. Explain the concept of overlapping subproblems in dynamic programming and how they are addressed.

   - Same subproblems repeat.
   - Use **memoization** or **tabulation.**

4. What are some common applications of dynamic programming in algorithm design?

   - Fibonacci, Knapsack, LCS, Floyd-Warshall, Matrix Chain.

5. Explain the concept of state transition and recurrence relation in dynamic programming.

- Defines problem evolution.
- Example: **F(n) = F(n-1) + F(n-2).**

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

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