Experiment No. 11

Aim:- Identify and implement at least one problem each based on any two paradigms.

Problem Statement:- Identification and implementation of a problem based on any two below given paradigms.

1. Divide and Conquer
2. Greedy method
3. Dynamic programming
4. Backtracking
5. Branch and Bound
6. String matching algorithms

Theory:-

Divide and Conquer: In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem. Divide-and-conquer design strategy can be applied to solve the problems like Merge sort, Quick sort, Finding maximum and minimum element, Binary Search etc.

Greedy Method: An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions. Most networking algorithms use the greedy approach. For example Knapsack problem, Optimal storage on tapes, job sequencing with deadlines, Prim's minimum cost spanning tree algorithm.

Dynamic Programming: Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems. Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved

sub-problems. The solutions of sub-problems are combined in order to achieve the best solution. Dynamic Programming can be applied to solve problems like Single source

shortest path by Bellman Ford Algorithm, All pair shortest path by Floyd Warshall algorithm, 0/1 knapsack problem, assembly line scheduling etc.

Backtracking: Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem. Backtracking algorithm is applied to decision problem, Optimization problem and Enumeration problem. For example, N-Queen's problem, Sum of subsets and Graph coloring are some of the examples solved using Backtracking.

Branch and Bound: Branch-and-bound is a general technique for improving the searching process by systematically enumerating all candidate solutions and disposing of obviously impossible solutions. Branch-and-bound usually applies to those problems that have finite solutions, in which the solutions can be represented as a sequence of options. The first part of branch-and-bound branching requires several choices to be made so that the choices will branch out into the solution space. In these methods, the solution space is organized as a treelike structure. Branching out to all possible choices guarantees that no potential solutions will be left uncovered. But because the target problem is usually NP-complete or even NP-hard, the solution space is often too vast to traverse. An important advantage of branch-and-bound algorithms is that we can control the quality of the solution to be expected, even if it is not yet found. Branch and bound can be applied to 15 puzzle problem and Travelling salesperson problem.

String Matching Algorithms: Pattern Searching algorithms are used to find a pattern or substring from another bigger string. There are different algorithms. The main goal to design these type of algorithms is to reduce the time complexity. The traditional approach may take lots of time to complete the pattern searching task for a longer text.

Conclusion:-

Note: In this section you have to conclude how appropriate the chosen paradigm is for solving the problem. Also give the justification to your answer. Conclusion such as, "Thus we have successfully implemented or studied the problem will not be accepted."

NAME: HARSHBHAI SOLANKI
ROLL.N0:62
SE-4-D

EXPERIMENT N0-11

AIM: Solve any problems that are based on two different algorithm paradigms.

1:MAX MIN USING GREEDY METHOD:

PROBLEM STATEMENT:

You will be given a list of integers, arr and a single integer k You must create an array of length k from elements of arr such that its *unfairness* is minimized. Call that array arr' Unfairness of an array is calculated as

MAX(arr') – MIN(arr')

Where:
- *max* denotes the largest integer in arr'
- *min* denotes the smallest integer in arr'

THEORY:

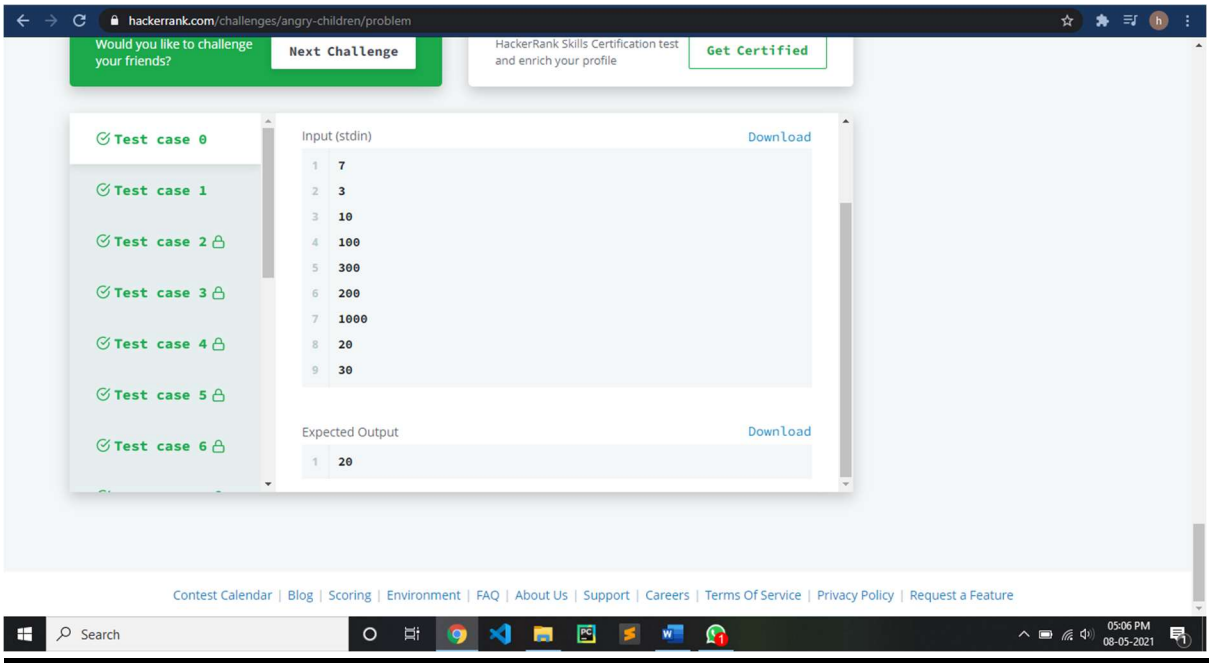
- we have given arr of size n and integer k
- we have to make possible sub array of k size
- from all the sub array we have to find the max(arr) – min(arr)
- and display the smallest among the max(arr) – min(arr)
- EG: arr = [1,4,7]
  Possible sub array :
  [1,4] : max=4 and min=1 : 4-1=3
  [4,7] : max=7 and min=4 : 7-4=3
  [1,7] : max=7 and min=1 : 7-1=6
  Therefore min among above is 3
- Sort the array
- result = arr[k-1] - arr[0]
- for i in range(n-k+1):
- if arr[i+k-1] - arr[i] < result:
- result = arr[i+k-1] - arr[i]
- return result

CODE:

```
import math
import os
import random
import re
import sys
def maxMin(k, arr):
    arr.sort()
    result = arr[k-1] - arr[0]
    for i in range(n-k+1):
        if arr[i+k-1] - arr[i] < result:
            result = arr[i+k-1] - arr[i]
    return result

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')
    n = int(input().strip())
    k = int(input().strip())
    arr = []
    for _ in range(n):
        arr_item = int(input().strip())
        arr.append(arr_item)
    result = maxMin(k, arr)
    fptr.write(str(result) + '\n')
    fptr.close()
```

OUTPUT:



CONCLUSION:

- Time complexity is O(nlogn) Because we used quick sort.
- Space complexity is O(1)

2: THE COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

PROBLEM STATEMENT:-

Given an amount and the denominations of coins available, determine how many ways change can be made for amount. There is a limitless supply of each coin type.

THEORY :-

- The Coin Change Problem is considered by many to be essential to understanding the paradigm of programming known as Dynamic Programming**.**
- Ex. If we have given 4rs to change and available coin types are [1,2,3]. So, there is 4 ways to change 4rs i.e. {1,1,1,1}, {2,2}, {3,1},{2,1,1}.
- First we will enter the amount to change and number of coin types.
- Then we will enter the values of each coin type.
- After that we will check if the entered amount to change is less than 0, if true we return 0 and if the entered amount to change is equal to 0 will return 1 and if number of coins is less than or equal to 0 will return 0.
- counts = [0] * (money+1)
  counts[0] = 1
- for i in range(0, no_of_coins):
  sum = 0
  for j in range(coins[i],money+1):

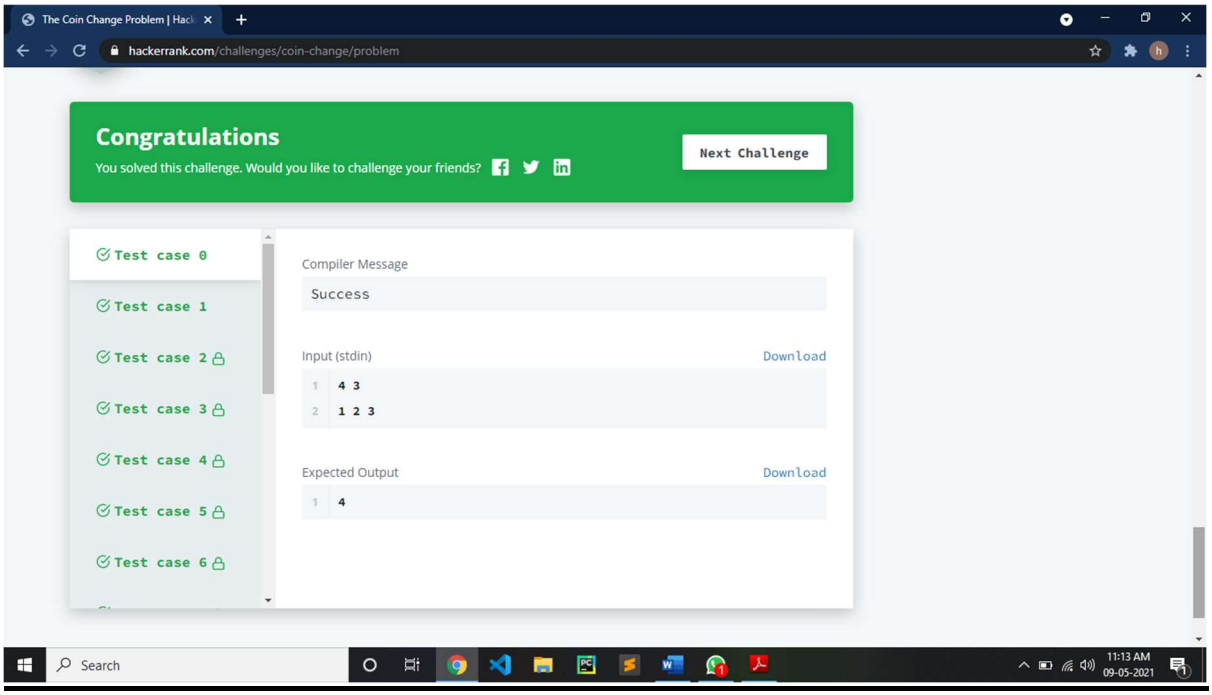
  counts[j] += counts[j-coins[i]]

- return counts[money]

SOURCE CODE:

```
import math
import os
import random
import re
import sys
money,no_of_coins = list(map(int,input().strip().split(' ')))
coins = list(map(int,input().strip().split(' ')))
count = 0
def count_make_change(money, coins, no_of_coins):
    if money < 0:
        return 0
    if money == 0:
        return 1
    if no_of_coins <= 0:
        return 0
    return count_make_change(money-coins[no_of_coins-
    1], coins, no_of_coins) + count_make_change(money,coins,no_of_coins-1)
def count_make_change_bottom_up(money, coins, no_of_coins):
    counts = [0] * (money+1)
    counts[0] = 1
    for i in range(0, no_of_coins):
        sum = 0
        for j in range(coins[i],money+1):
            counts[j] += counts[j-coins[i]]
    return counts[money]
print(count_make_change_bottom_up(money,coins,no_of_coins))
```

OUTPUT:



CONCLUSION :-

- Time Complexity is O(N*M)
- space complexity is O(N).