EXPERIMENT - 3

Student Name: Sandeep Kumar **UID: 20BCS4885**

Branch: CSE **Section/Group: 603/A**

Semester: 6th semester **Subject: Competitive Coding**

Aim: To demonstrate the concept of Heap Model

Objective:

a) Last stone weight

b) Cheapest flight with K stops

Problem 1: Last stone weight

Solution code:

```
class Solution {
public:
   int lastStoneWeight(vector<int>& stones) {
      priority_queue<int> pq;
      for(int i=0;i<stones.size();i++){
         pq.push(stones[i]);
      }
      while(pq.size()!=1){
         int a=pq.top();
         pq.pop();
         if(!pq.empty()){
         int b=pq.top();
         pq.pop();
         pq.push(a-b);}
         else
         return pq.top();
      }
      return pq.top();
   }

};
```

Explanation of code:

- This code implements a solution to the "Last Stone Weight" problem, which is as follows:

- You are given an array of integers stones, where each element represents the weight of a stone. We are to repeatedly perform the following operation until there is at most one stone remaining:

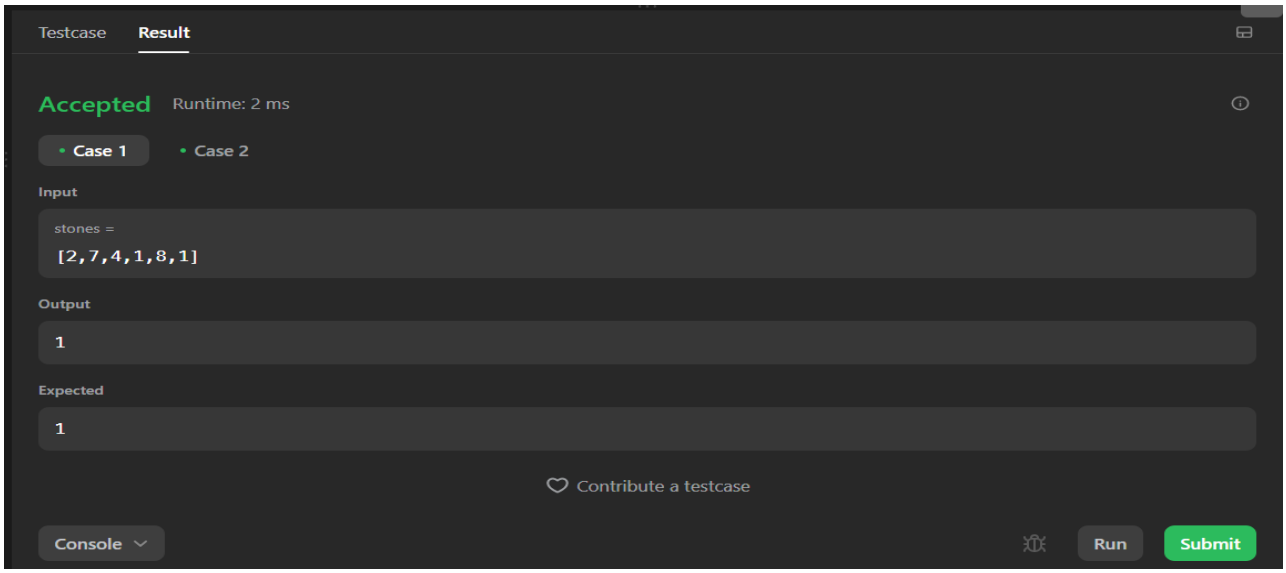
- Select the two heaviest stones and smash them together.
  1. If the two stones are of equal weight, both stones are destroyed.
  2. If the two stones are different weights, the heavier stone is destroyed, and the remaining stone's weight is updated to the difference between the weights.
  3. Return the weight of the last stone that remains. If there are no stones left, return 0.

- The solution creates a max heap (priority queue) of the given stones array. The top element of the heap represents the heaviest stone. The solution then repeatedly extracts the two heaviest stones, and if they are different, it computes their difference and adds the result back to the heap. This process continues until there is only one or zero stones left.

- The solution returns the weight of the remaining stone (if any) or 0 if the heap is empty.

  Overall, the solution has a time complexity of O(NlogN), where N is the size of the stones array, due to the use of a priority queue.

Output:



Problem 2: Cheapest Flights Within K Stops

Input Code:

```
class Solution {
public:
   int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {
      unordered_map<int, vector<pair<int, int>>> graph;
      for(auto e: flights) graph[e[0]].push_back({e[1], e[2]});
      vector<int> prices(n, -1);
      queue<pair<int, int>> q; q.push({src, 0});
      ++K;
      while(!q.empty()) {
         if(!K) break;
         int len = q.size();
         for(int i = 0; i < len; i++) {
            auto cur = q.front(); q.pop();
            for(auto e: graph[cur.first]) {
               int price = cur.second + e.second;
               if(prices[e.first] == -1 || price < prices[e.first]) {
                  prices[e.first] = price;
                  q.push({e.first, price});
               }
            }
         }
         K--;
      }
      return prices[dst];
   }
};
```

Explanation of Code:

- The given code defines a class Solution with a public method findCheapestPrice which takes five arguments: an integer n representing the number of cities, a vector of vectors flights representing the flights between cities and their costs, two integers src and dst representing the source and destination cities, and an integer K representing the maximum number of stops allowed. The method returns the minimum cost to travel from src to dst with at most K stops.

- The method first creates an unordered map graph to store the flights as an adjacency list, where each vertex is mapped to a vector of pairs representing its neighbors and their costs.

It then initializes a vector prices of size n with -1 values, and a queue q containing a pair representing the source city and a price of 0. It also increments K by 1 to account for the source city itself.

- The method then enters a loop that continues until either the queue is empty or K reaches 0. In each iteration, it removes all the elements from the queue and processes their neighbors. For each neighbor, it computes the total cost from the source city to the neighbor by adding the cost of the flight to the current price. If the total cost is lower than the current price stored in prices, or if the neighbor has not been visited yet, it updates the price and adds the neighbor to the queue with its updated price.

- The loop continues until either the queue is empty or K reaches 0. Finally, the method returns the price of the destination city stored in prices.

- Overall, this code appears to be a correct and efficient solution to the "Cheapest Flights Within K Stops" problem using BFS and an adjacency list.

Output: