School of Computing

SRM IST, Kattankulathur – 603 203

Course Code: 18CSC204J

Course Name: Design And Analysis Of Algorithm

Title of Experiment	**Trapping Rain Water**
Team Members	Abhinav D Trivedi , Allen Jerome, Uppu Manikanta
Register Numbers	RA2011003011181, RA201100301167, RA2011003011182
Date of Submission	15/06/2022

Staff Signature with date

Contribution Table :-

1.Problem Definition :

Finding the maximum amount of water that can be trapped within a given set of bars where each bar with particular width.The idea is to calculate the maximum height bar on the left and right of every bar. The amount of water stored on top of each bar is equal to the minimum among the leading' bar to the left and right minus the current bar's height.

2.Problem Explanation with examples :

Given an integer array A[] consisting of N non-negative integers representing an elevation map, where the width of each bar is 1. The task is to compute the total volume of water that can be trapped after rain.

Example 1:

Explanation :

The rain water trapped is represented by the blue region.

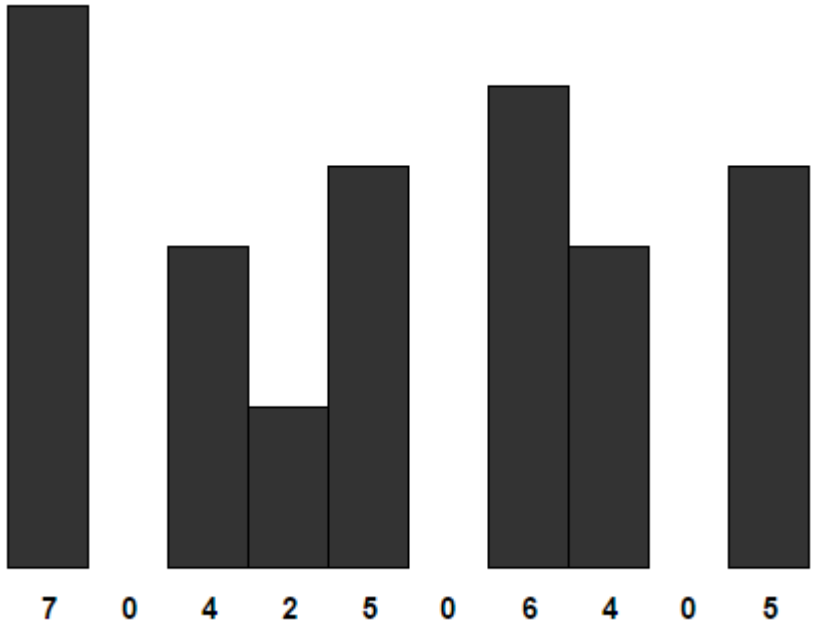
- Trap 1 unit of water between the first and third block
- Trap 4 units of water between the second and third blocks.

Therefore, the total volume of water is – 1 + 4 + 1 = 6 units.

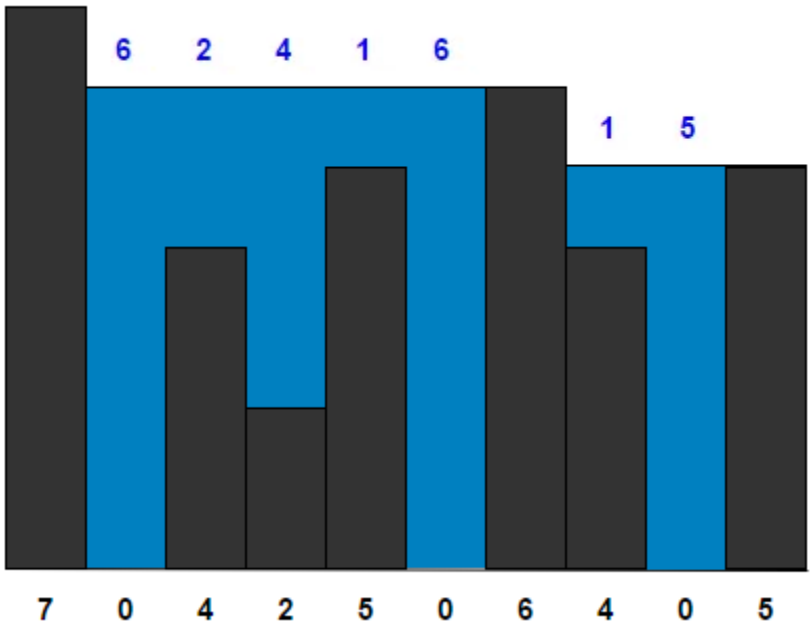
Input An array containing height of bars { 0 , 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 }
Output: The maximum amount of water that can be trapped is 6, as shown below (blue).

Example 2:

Consider the below problem, given an integer array A[] consisting of N non-negative integers representing an elevation map, where the width of each bar is 1. The task is to compute the total volume of water that can be trapped after rain



Input: An array containing height of bars `{7, 0, 4, 2, 5, 0, 6, 4, 0, 5}`



Output : The maximum amount of water that can be trapped is 25, as shown below (blue).

3.Design Technique Used :

A) Dynamic Programming :-

One of the best design technique used for trapping rainwater problem is **Dynamic Programming Approach**

The technique of breaking a problem statement into subproblems and using the optimal result of subproblems as an optimal result of the problem statement is known as dynamic programming. This technique uses an optimized approach and results in optimal solutions. Dynamic programming cannot solve all the problems. There are several algorithms that can be solved using greedy and divide and conquer techniques. The difference between recursion and DP recursion is memoization in DP. If the subproblem does not require memorization, in any case, DP cannot solve that problem.

Dynamic programming cannot solve all the problems. There are several algorithms that can be solved using greedy and divide and conquer techniques. The difference between recursion and DP recursion is memoization in DP. If the subproblem does not require memorization, in any case, DP cannot solve that problem.
Following are the basic approaches of DP -

1. **Top-Down Approach:** In this approach, the problem is broken into subproblems, and the result at each sub problem is remembered. When a result at subproblem is needed again further, it is first checked in the memory stage and then used.
2. **Bottom-Up Approach:** This approach follows the idea of computing first and moves backward. It is done to avoid the process of recursion. The smaller inputs at each step in the bottom-up approach contribute to form larger ones.

B) Using stacks

In **DP**, the arrays were traversed twice. Can we improve the approach to a single scan? The idea is to keep track of the area between the current block **A[i]** and all the previous blocks with smaller heights in the array. We can simply use a **stack** to track the index of the previous smaller blocksA stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

C) Brute force Approach:

Since our task is to find the maximum height on left and right for each element of the array, simply traverse each element of the array **A[]**. For each element, find the maximum height on the left and maximum height on the right. At last, add min(right_max, left_max) – A[i] to answer. A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found. Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

4.Algorithm for the problem :

Algorithm for the trapping rainwater is as follows :

A) Dynamic Programming :-

- Initialise two arrays **left_max[]** and **right_max[]** of size **N.**
- Consider a variable **mx = 0**.
- Traverse from left to right and for each index **i**, update **mx = max(mx, A[i[)** and assign **left_max = mx**.
- Similarly, traverse a loop, **N** to **1** and for each index **i**, update **mx = max(mx, A[i[)** and assign **right_max = mx**.
- Initialise a variable **res = 0** and traverse from **0** to **N – 1**. For each index **i**, add **min(left_max[i], right_max[i]) – A[i]** to **res.**

B) Using stacks:-

- Declare a stack **S.**
- Traverse the array from left to right:
    - If the current block **A[i]** is larger than the top of the stack i.e. **S.top()**, it can be inferred that the block at the top of the stack is confined between the current block and the previous block in the stack.
    - Therefore, perform **S.pop()** and add the water that can be stored.
- The total volume of water can be calculated as follows:
    - Length = Current index **i** – **S.top() – 1**
    - Width = **min(A[i] – A[S.top()] – A[top]**
    - Add the total volume to res = **Length X Width**

C) Brute force Approach:-

- Initialise a variable **res** to 0, to store the final answer.

- Traverse the array **A[]** from **1** to **N** and for each element:
    - Initialise **left_max** = 0 and **right_max** = 0
    - Traverse from **A[i]** till the beginning and update:
        - left_max = max(left_max, A[i])
    - Similarly, traverse from **A[i]** till the end of the array and update:
        - right_max = max(right_max, A[i])

  Add min(left_max, right_max) – A[i] to **res**.

5.Explanation of algorithm with example :

A) Dynamic Programming :-

- We create two arrays **left[n]** and **right[n]**

- Now run a loop from left to right and fill the left[n] array. At every iteration **i**, we store the maximum element that occurred up to that point in left[i]. **(Think!)**

```
left[0] = height[0]
for(int i = 1; i < n; i = i + 1)
{
    left[i] = max(left[i-1], height[i])
}
```

- Similarly, run a loop from right to left, and fill the right[n] array. At every iteration i, store the maximum element occurred up to that point in right[i]. **(Think!)**

```
right[n-1] = height[n-1]
for(int i = n-2; i >= 0; i = i - 1)
{
    right[i] = max(right[i+1], height[i])
}
```

- Now, traverse the height[] array and calculate the total amount of water.

```
int trappeWater = 0
for(int i = 0; i < n; i = i + 1)
{
    trappedWater = trappedWater + min(left[i], right[i]) - height[i]
}
```

- Return the value stored in the variable **trappedWater**.

B) Using stacks:-

- We declare a stack **S** to store the indices of the towers.

- Now we scan the **height[n]** using a loop variable **curr < n**

- If we found a current tower larger than that tower at the top of the stack, we can say that the tower at the top of the stack is confined between the current tower and a previous tower in the stack. Hence, we pop the stack and add water trapped between towers to the total water trapped. We can continue this step in a loop until we find the current tower smaller than the tower at the top of the stack.

- Water trapped between towers = Area of the rectangular region formed by the current tower, popped tower, and tower at the top of the stack

Region length = (current index — index of the top stack element — 1)

Region height = min (height of the current tower, height of tower at top of the stack) — height of the popped tower

Water trapped = Region length x Region height

- If the current tower is smaller than or equal to the tower at the top of the stack, we push the index of the current tower to the stack and move to the next tower. It means current tower is confined with tower at the top of the stack.

C)Brute force Approach:-

- We declare and initialise the variable **trappedWater** to store the total trapped water.

- Now scan the **height[]** from from i = 0 to n-1. Inside the loop, we declare and initialise the variable **left_maxHeight** and **right_maxHeight** to store the maximum height of the tower on both sides of the current tower.

- For each tower **height[i]**, we calculate the maximum height of the tower from current index **i** up to the left end. We store this value in the variable **left_maxHeight**.

```
for(int k = i; k >= 0; k = k - 1)
{
    left_maxHeight = max(height[k], left_maxHeight)
}
```

- For each tower **height[i]**, we calculate the maximum height of tower from current index **i** up to right end. We store this value in the variable **right_maxHeight.**

```
for(int j = i; j < n; j = j + 1)
{
    right_maxHeight = max(height[j], right_maxHeight)
}
```

- Now we update the total amount of water by using the above formula.

```
trappedWater
= trappedWater + min(left_MaxHeight, right_maxHeight) - height[i]
```

- By end of the above loop, we return the value of the variable **trappedWater.**

6.Complexity Analysis:-

A) Dynamic Programming :-

To find **right_maxHeight** and **left_maxHeight**, we used two separate loops. And, to traverse the array, also we used one loop. So, **Time Complexity** = O(n) + O(n) = O(n).

We used two extra arrays of size n. So **Space Complexity =** O(n) + O(n) = O(n)

B) Using stacks:-

We are doing single traversal of the height[] array. In the worst case, we are processing each tower twice using stack i.e. one push() and one pop() operation. **(Think!)** Both push() and pop() operations takes O(1) time in the worst case. So, time Complexity = O(n).

In the worst-case stack can take up to n elements. Space Complexity = O(n)

C)Brute force Approach:-

We are using nested loops where the outer loop is scanning the height[] array, and two inner loops are finding **right_maxHeight** and **left_maxHeight**. So every iteration of the outer

loop, we are traversing each element via inner loops. Time Complexity = O(n * n) = O(n²), Space Complexity = O(1)

7.Conclusion :-

Therefore the Trapping Rain Water Problem using Dynamic Programming, Stacks and Brute Force approaches is successfully done .

8.References

https://medium.com/enjoy-algorithm/trapping-rain-water-a79938abf921

https://www.interviewbit.com/blog/trapping-rain-water/

****