Data Analysis and Algorithm

TA – I

Name: Madhura Mahatme

Section: A1-B3

Roll no.: 36

Git Link

1. Trapping Rain Water:

Explanation:

We are given heights of vertical bars.
When it rains, water will be trapped between the bars.
We need to calculate how much water is trapped.

Two-Pointer Approach:

- Water above a bar depends on the minimum of the tallest bar on the left and the tallest bar on the right.
- At any position i in the array height[]:

$$\text{Water at index } i = \min(\text{leftMax}, \text{rightMax}) - \text{height}[i]$$

- leftMax = the tallest bar from the start up to index i.
- rightMax = the tallest bar from index i to the end.
- The amount of water that can stay at index i depends on the shorter wall between left and right.
- From that shorter wall's height, we subtract the current bar height (height[i]) to see how much space is left for water.
- Instead of precomputing arrays for leftMax and rightMax, we can use two pointers (left and right) moving inward.

Logic:

1. Start with:
    - left pointer at index 0
    - right pointer at index n-1
    - leftMax = height[left]
    - rightMax = height[right]
2. While left < right:
    - If leftMax < rightMax, we know the water trapped depends on leftMax:

i. Move left one step right.
ii. Update leftMax.
iii. Add trapped water: leftMax - height[left].
- Else:
  i. Move right one step left.
  ii. Update rightMax.
  iii. Add trapped water: rightMax - height[right].

3. Continue until left and right meet.
4. The sum is the total trapped water.

Example:

Heights:

- height = [0,1,0,2,1,0,1,3,2,1,2,1]

Step by step:

- Start: left=0 (0), right=11 (1), leftMax=0, rightMax=1.
- Since leftMax < rightMax, move left:
- left=1, leftMax=max(0,1)=1.
  - water += 1-1=0.
- left=2, leftMax=1.
  - water += 1-0=1.
- left=3, leftMax=max(1,2)=2.
  - water += 2-2=0.
- left=4, leftMax=2.
  - water += 2-1=1.
- left=5, leftMax=2.
  - water += 2-0=2.
- left=6, leftMax=2.
  - water += 2-1=1.
- left=7, leftMax=max(2,3)=3.
  - water += 3-3=0.
- left=8, leftMax=3.
  - water += 3-2=1.
- left=9, leftMax=3.
  - water += 3-1=2.
- left=10, leftMax=3.
  - water += 3-2=1.
- left=11 → loop ends.

Total water = 1+1+2+1+2+1 = 6

Final Answer :

For input:

12

0 1 0 2 1 0 1 3 2 1 2 1

Output:

Total trapped Water: 6

We keep track of the tallest bars from both ends, move the pointer with the smaller height inward, and calculate water step by step. This avoids using extra space.

Code:

```
#include <iostream>
#include <vector>
using namespace std;
int trap(vector<int>& height) {
    int left =0;
    int right = height.size() - 1;
    int leftMax =height[left];
    int rightMax=height[right];
    int water = 0;
    while(left< right){
        if (leftMax< rightMax) {
            left++;
            leftMax =max(leftMax, height[left]);
            water += leftMax - height[left];
        }
        else{
            right--;
            rightMax = max(rightMax, height[right]);
            water += rightMax - height[right];
        }
    }
return water;
}
int main() {
    int n;
    cout<< "Enter no. of bars: ";
    cin >> n;
    vector<int> height(n);
    cout << "Enter heights: ";
    for (int i = 0; i < n; i++) {
        cin >> height[i];
    }
```

```
    int result = trap(height);
    cout <<"Total trapped Water: "<< result << endl;

    return 0;
}
```

Output:

```
Enter no. of bars: 5
Enter heights: 1 6 3 8 9
Total trapped Water: 3
PS C:\Users\Madhura\OneDrive\Desktop\C>
```

Test Cases:

Test Case 1:

Input:

Enter no. of bars: 6

Enter heights: 0 1 0 2 1 0

Output:

Total trapped Water: 2

Expected Output:

Water trapped between bars = 2 units.

Test Case 2

Input:

Enter no. of bars: 6

Enter heights: 4 2 0 3 2 5

Output:

Total trapped Water: 9

Expected Output:

Traps: (4,2,0,3,2,5) → total = 9.

Test Case 3

Input

Enter no. of bars: 3

Enter heights: 3 2 1

Output

Total trapped Water: 0

Expected Output

Decreasing slope, so no water trapped.

Test Case 4

Input:

Enter no. of bars: 5

Enter heights: 2 0 2 0 2

Output:

Total trapped Water: 4

Expected Output:

Water between first and last peaks = 4 units.

Test Case 5

Input:

Enter no. of bars: 12

Enter heights: 0 1 0 2 1 0 1 3 2 1 2 1

Output:

Total trapped Water: 6

Expected Output:

Classic example, total trapped water = 6.

2. Merge K Sorted Lists

Explanation:

Problem:

We are given **k sorted linked lists**.
We need to **merge them into one sorted linked list**.

Working:

The code does this in steps:

1. mergeTwo(a, b)

- This merges two sorted linked lists into one sorted list.
- We keep comparing the first nodes of a and b.
- Pick the smaller one and attach it to the new list.
- Continue until one list finishes, then attach the remaining nodes.
- Return the merged list.

2. mergeAll(lists)

- We don't merge all k lists at once.
- Instead, we merge them two by two:
    - Take list1 & list2 → merge them.
    - Take list3 & list4 → merge them.
    - Store results in a new vector temp.

- After one round, the number of lists becomes smaller.
- Keep repeating until only one list is left.
- That final list is the fully merged sorted list.

3. Helper Functions

- buildList(arr) → builds a linked list from an array of numbers.
- printList(head) → prints the linked list in a -> b -> c format.

Example

Suppose the input is:

3 (k = 3 lists)

List 1: 1 -> 4 -> 7

List 2: 2 -> 5 -> 8

List 3: 3 -> 6 -> 9

Step 1: Merge Pairwise

Merge list1 and list2:

[1 -> 4 -> 7] + [2 -> 5 -> 8]

= 1 -> 2 -> 4 -> 5 -> 7 -> 8

- List3 is alone, so it stays as is.

Now lists =

[1 -> 2 -> 4 -> 5 -> 7 -> 8], [3 -> 6 -> 9]

Step 2: Merge Again

Merge these two:

[1 -> 2 -> 4 -> 5 -> 7 -> 8] + [3 -> 6 -> 9]

= 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

Final Answer

Merged list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

MergeTwo merges 2 sorted lists. MergeAll keeps merging pairs until only 1 list is left.
The final list is sorted and contains all elements.

Code:

```
#include <iostream>
#include <vector>
using namespace std;
struct Node {
    int data;
    Node* next;
    Node(int x) : data(x), next(nullptr) {}
};

Node* mergeTwo(Node* a, Node* b) {
    Node dummy(0);
    Node* tail = &dummy;
    while(a && b) {
        if(a->data <= b->data) {
            tail->next = a;
            a = a->next;
        }
        else{
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next=(a ? a : b);
    return dummy.next;
}

Node* mergeAll(vector<Node*>& lists) {
    if (lists.empty()) return nullptr;
    while (lists.size() > 1){
        vector<Node*> temp;
        for (size_t i = 0; i < lists.size(); i += 2){
            Node* first = lists[i];
            Node* second = (i + 1 < lists.size()) ? lists[i + 1] :
nullptr;
            temp.push_back(mergeTwo(first, second));
        }
        lists = move(temp);
    }
    return lists[0];
}
Node* buildList(const vector<int>& arr){
```

```cpp
    Node dummy(0);
    Node* curr = &dummy;
    for (int val : arr) {
        curr->next = new Node(val);
        curr = curr->next;
    }
    return dummy.next;
}
void printList(Node* head){
    while (head) {
        cout << head->data;
        if (head->next) cout << " -> ";
        head = head->next;
    }
    cout << "\n";
}
int main() {
    int k;
    cout << "Enter how many lists: ";
    cin >> k;
    vector<Node*> lists;
    for (int i = 0; i < k; i++) {
        int n;
        cout << "Enter size of list " << i + 1 << ": ";
        cin >>n;

        vector<int> arr(n);
        cout << "Enter " << n << " sorted elements: ";
        for (int j= 0; j< n; j++) {
            cin >> arr[j];
        }
        lists.push_back(buildList(arr));
    }
    Node* result = mergeAll(lists);
    cout<<"Merged list: ";
    printList(result);
    return 0;
}
```

Output:

```
Enter how many lists: 4
Enter size of list 1: 5
Enter 5 sorted elements: 1 5 9 7 5
Enter size of list 2: 3
Enter 3 sorted elements: 1 2 9
Enter size of list 3: 4
Enter 4 sorted elements: 2 3 5 8
Enter size of list 4: 3
Enter 3 sorted elements: 2 8 9
Merged list: 1 -> 1 -> 2 -> 2 -> 2 -> 3 -> 5 -> 5 -> 8 -> 8 -> 9 -> 7 -> 5 -> 9 -> 9
PS C:\Users\Madhura\OneDrive\Desktop\C>
```

Test Cases:

Test Case 1

Input:

Enter how many lists: 3

Enter size of list 1: 3

Enter 3 sorted elements: 1 4 5

Enter size of list 2: 3

Enter 3 sorted elements: 1 3 4

Enter size of list 3: 2

Enter 2 sorted elements: 2 6

Output:

Merged list: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6

Expected Output:

- All three lists are merged in order.
- Result: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6

Test Case 2

Input:

Enter how many lists: 2

Enter size of list 1: 4

Enter 4 sorted elements: 2 5 8 10

Enter size of list 2: 3

Enter 3 sorted elements: 1 3 7

Output:

Merged list: 1 -> 2 -> 3 -> 5 -> 7 -> 8 -> 10

Expected Output:

- Merging keeps order intact.
- Result: 1 -> 2 -> 3 -> 5 -> 7 -> 8 -> 10
-

Test Case 3

Input:

Enter how many lists: 3

Enter size of list 1: 0

Enter 0 sorted elements:

Enter size of list 2: 0

Enter 0 sorted elements:

Enter size of list 3: 0

Enter 0 sorted elements:

Output:

Merged list:

Expected Output:

- All lists are empty, so merged result is empty.

Test Case 4

Input:

Enter how many lists: 2

Enter size of list 1: 5

Enter 5 sorted elements: 1 2 3 4 5

Enter size of list 2: 5

Enter 5 sorted elements: 6 7 8 9 10

Output:

Merged list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

Expected Output:

- First list entirely before second list.
- Result: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

Test Case 5

Input:

Enter how many lists: 4

Enter size of list 1: 2

Enter 2 sorted elements: 1 10

Enter size of list 2: 2

Enter 2 sorted elements: 2 9

Enter size of list 3: 2

Enter 2 sorted elements: 3 8

Enter size of list 4: 2

Enter 2 sorted elements: 4 7

Output:

Merged list: 1 -> 2 -> 3 -> 4 -> 7 -> 8 -> 9 -> 10

Expected Output:

- All four lists merged, sorted ascending.
- Result: 1 -> 2 -> 3 -> 4 -> 7 -> 8 -> 9 -> 10

3. Best Time to Buy and Sell Stock

Explanation:

This problem is about finding the best day to buy and the best day to sell a stock to get the highest profit.

Concept:

- To earn maximum profit, we need to buy at the lowest price and sell at a higher price later.
- We can solve it efficiently with just one pass through the price list.

Approach:

1. Keep track of:
    - minPrice → the smallest price seen so far
    - maxProfit → the largest profit calculated so far
2. Go through each day's price:
    - If today's price < minPrice, update minPrice
    - Otherwise, calculate potential profit: profit = price - minPrice
    - If profit > maxProfit, update maxProfit
3. After checking all days, maxProfit gives the answer.

Efficiency:

- Time Complexity: O(n)
- Space Complexity: O(1)

Working:

- Tracking the lowest price ensures we always consider the best day to buy.
- Calculating profit at each step ensures we capture the largest possible profit when selling later.

Code:

```
#include <iostream>
#include <vector>
using namespace std;
int maxProfit(vector<int>& prices){
    if(prices.empty())
    return 0;
    int buy= prices[0];
    int profit= 0;
    for(int i= 1; i < prices.size(); i++){
        if (prices[i] < buy){
            buy= prices[i];
        }
        else if(prices[i] - buy > profit){
            profit =prices[i]-buy;
        }
    }
    return profit;
}
int main(){
    int n;
    cout <<"Enter number of days: ";
    cin >>n;
    vector<int>prices(n);
    cout << "Enter stock prices: ";
    for (int i = 0; i < n; i++){
        cin>>prices[i];
    }
    int result =maxProfit(prices);
    cout<<"Maximum Profit: "<<result<< endl;
    return 0;
}
```

Output:

```
Enter number of days: 5
Enter stock prices: 900
500
423
1000
965
Maximum Profit: 577
```

Test Cases:

Test Case 1

Input:

Prices = [7, 1, 5, 3, 6, 4]

Output:

Maximum Profit = 5

Test Case 2

Input:

Prices = [7, 6, 4, 3, 1]

Output:

Maximum Profit = 0

Test Case 3

Input:

Prices = [1, 2, 3, 4, 5]

Output:

Maximum Profit = 4

Test Case 4

Input:

Prices = [3, 3, 5, 0, 0, 3, 1, 4]

Output:

Maximum Profit = 4

Test Case 5

Input:

Prices = [2, 4, 1]

Output:

Maximum Profit = 2

4. Find Median from Data Stream

Explanation:

We are getting numbers one by one. We need to do two things:

1. Add the new number to our collection.
2. Find the median of all numbers we have so far.

Problem:

- We cannot sort the whole list every time a new number comes.
- We need a way to find the median quickly after each new number.

Solution:

- We use two heaps (special lists that give the largest or smallest number quickly):
  1. Max-Heap: Stores the smaller half of numbers (biggest number is on top).
  2. Min-Heap: Stores the larger half of numbers (smallest number is on top).

How to get median:

- If both heap have the same size → median = average of the tops of both heaps.
- If one heap has more numbers → median = top of that heap.

Example

Input numbers: 5, 3, 8, 9

Step by step:

1. We add 5 → median = 5
2. We add 3 → numbers = [3,5] → median = (3+5)/2 = 4
3. We add 8 → numbers = [3,5,8] → median = 5
4. We add 9 → numbers = [3,5,8,9] → median = (5+8)/2 = 6.5

Medians after each number: 5 → 4 → 5 → 6.5

Code:

```cpp
#include <iostream>
#include <queue>
using namespace std;

priority_queue<int>leftHeap;
priority_queue<int, vector<int>, greater<int>> rightHeap;

void insertNumber(int value) {
    leftHeap.push(value);
    rightHeap.push(leftHeap.top());
    leftHeap.pop();

    if (rightHeap.size() > leftHeap.size()) {
        leftHeap.push(rightHeap.top());
        rightHeap.pop();
    }
}
double findMedian(){
    if (leftHeap.size() == rightHeap.size())
        return (leftHeap.top()+ rightHeap.top()) / 2.0;
    return leftHeap.top();
}
int main(){
    int count;
    cout<<"Enter the number of numbers you want to put:";
    cin >>count;
    cout <<"Enter "<< count<<" values:\n";
    for(int i=0; i<count; i++){
        int x;
        cin>>x;
        insertNumber(x);
        cout << "Median so far: "<<findMedian()<<endl;
    }
    cout<<"Median of all numbers: "<< findMedian()<<endl;
    return 0;
}
```

Output:

```
Enter the number of numbers you want to put:5
Enter 5 values:
45 8 23 63 89
Median so far: 45
Median so far: 26.5
Median so far: 23
Median so far: 34
Median so far: 45
Median of all numbers: 45
PS C:\Users\Madhura\OneDrive\Desktop\C>
```

Test Cases:

Test Case 1

Input:

5

1 2 3 4 5

Output after each insertion:

Current median: 1

Current median: 1.5

Current median: 2

Current median: 2.5

Current median: 3

Final median:

Final median: 3

Test Case 2

Input:

10 20 30 40

Output after each insertion:

Current median: 10

Current median: 15

Current median: 20

Current median: 25

Final median:

Final median: 25

Test Case 3

Input:

3

5 15 10

Output after each insertion:

Current median: 5

Current median: 10

Current median: 10

Final median:

Final median: 10

Test Case 4

Input:

6

2 4 6 8 10 12

Output after each insertion:

Current median: 2

Current median: 3

Current median: 4

Current median: 5

Current median: 6

Current median: 7

Final median:

Final median: 7

Test Case 5

Input:

5

7 3 5 1 9

Output after each insertion:

Current median: 7

Current median: 5

Current median: 5

Current median: 4

Current median: 5

Final median:

Final median: 5