Infosys Prepration Questions

Question No 1

You are given two arrays arr1 of size n and arr2 of size 4*n.
Your task is to divide arr2 into n groups, with 4 elements in each group, and assign each group to one element in arr1.

Each element in arr1 represents a base time.
When you assign a group to a base time, each of the 4 elements in that group is added to the base time to produce 4 finish times.
The finish time for that group is the maximum of these 4 times.

Your goal is to find an assignment that minimizes the finish time across all groups.

Function Description

The function **minTime** takes the following inputs:

- `int arr1[n]`: an array of integers
- `int arr2[4*n]`: an array of integers

The function should return an integer:
the minimum possible finish time for all groups.

```
n = 2
arr1 = [8, 10]
arr2 = [2, 2, 3, 1, 8, 7, 4, 5]
```

One optimal solution is:

Assign elements **[2, 3, 7, 8]** from `arr2` to the first base time **8** in `arr1`.

1. Finish times: 8+2, 8+3, 8+7, 8+8 = 10, 11, 15, 16
2. Maximum finish time = **16**

Assign elements **[2, 1, 4, 5]** from `arr2` to the second base time **10** in `arr1`.

- Finish times: 10+2, 10+1, 10+4, 10+5 = 12, 11, 14, 15
- Maximum finish time = **15**

The **maximum finish time among all groups is 16**, which is the **minimum possible finish time** for any assignment.

Solution

```cpp
#include <bits/stdc++.h>
using namespace std;

bool canAssign(vector<int>& arr1, vector<int>& arr2, int T) {
    int j = 0; // pointer in arr2
    int n = arr1.size();

    for (int i = 0; i < n; i++) {
        int limit = T - arr1[i];
        int count = 0;

        while (j < arr2.size() && arr2[j] <= limit && count < 4) {
            j++;
            count++;
        }

        if (count < 4) return false;
    }
    return true;
}

int minTime(vector<int>& arr1, vector<int>& arr2) {
    sort(arr1.begin(), arr1.end());
    sort(arr2.begin(), arr2.end());

    int low = arr1[0] + arr2[0];
    int high = arr1.back() + arr2.back();
    int ans = high;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (canAssign(arr1, arr2, mid)) {
            ans = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return ans;
}

int main() {
    vector<int> arr1 = {8, 10};
    vector<int> arr2 = {2, 2, 3, 1, 8, 7, 4, 5};

    cout << minTime(arr1, arr2);
    return 0;
}
```

Question No 2

You are working with a **Git-based version control system**.

Each commit in a repository has a **unique integer ID** representing its timestamp, and the list of commit IDs is given in **ascending order**. The commit history is **complete and consistent**.

You are auditing this history and want to test **how many consecutive commits can be temporarily hidden** (for example, by a UI filter) such that the remaining visible commits still allow someone to **deduce exactly which commits were hidden**.

You are allowed to hide **a contiguous block of consecutive commits**.

Implement a function to **maximize the number of commits hidden**, while ensuring that the surrounding visible commits **clearly define the hidden range**.

The function **getMaxConsecutiveHidden** takes the following input:

- int commits[n]: an array of **unique integers in ascending order**

The function should return an **integer**, representing the **maximum number of consecutive commits** that can be hidden.

commits = [1, 3, 4, 5, 6, 9]

After hiding the maximum number of consecutive commits, the array becomes: [1, 3, ..., 6, 9] It is clear that the missing commits are 4 and 5. Hence, the optimal answer is: 2

```cpp
#include <bits/stdc++.h>
using namespace std;

int getMaxConsecutiveHidden(vector<int>& commits) {
    int n = commits.size();
    int ans = 0;

    for (int i = 1; i < n; i++) {
        int gap = commits[i] - commits[i - 1] - 1;
        ans = max(ans, gap);
    }

    return ans;
}

int main() {
    vector<int> commits = {1, 3, 4, 5, 6, 9};
    cout << getMaxConsecutiveHidden(commits);
    return 0;
}
```

Question 3

Consider a string of length **N** which consists of only **'1'** and **'0'**. You are also given a **2D array A** consisting of **Q queries**.
Each query is of the form **[L, R, X]**, and you have to perform these queries on the string.

The **score of the string** is calculated as follows for each query:

- The value **X** is added to the score **if the substring [L, R] of the string contains at least one '1'**.

Find the **maximum possible score** of the string.

- A substring is a **contiguous part** of the string.
- Initially, the **score of the string is 0**.

Input Format

- The first line contains an integer **N**, denoting the length of the string.
- The next line contains an integer **Q**, denoting the number of rows in **A**.
- The next **Q** lines each contain three integers **L, R, X**, representing a query

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Query {
    int L, R, X;
};

struct Fenwick {
    int n;
    vector<int> bit;
    Fenwick(int n) : n(n), bit(n + 1, 0) {}
    void update(int i, int v) {
        for (; i <= n; i += i & -i)
            bit[i] += v;
    }
    int query(int i) {
        int s = 0;
        for (; i > 0; i -= i & -i)
            s += bit[i];
        return s;
    }
    int rangeQuery(int l, int r) {
        return query(r) - query(l - 1);
    }
};

int main() {
    int N;
    cin >> N;
    int Q;
    cin >> Q;
    vector<Query> queries(Q);
    for (int i = 0; i < Q; i++) {
        cin >> queries[i].L >> queries[i].R >> queries[i].X;
    }
    sort(queries.begin(), queries.end(),
         [](const Query& a, const Query& b) {
             return a.X > b.X;
         });
    Fenwick fenwick(N);
    long long score = 0;
    for (auto &q : queries) {
        if (fenwick.rangeQuery(q.L, q.R) > 0) {
            score += q.X;
        } else {
            fenwick.update(q.R, 1);
            score += q.X;
        }
    }

    cout << score << endl;
    return 0;
}
```

Question 4

Problem Statement: Maximum of Subarray Minimums

Description

Implement a function that takes an array of integers and a subarray size k . Your goal is to find the **minimum** value for every contiguous subarray of size k , and then return the **maximum** among all those minimums.

Input Format

- int n: The number of elements in the array.
- int arr[n]: An array of n integers.
- int k: The size of the sliding window (subarray length).

Output Format

Return a single integer representing the maximum value among all subarray minimums.

Example

Input:

n = 5 arr = [1, 2, 3, 4, 5] k = 2

Step-by-Step Execution:

Subarrays of size 2: [1, 2], [2, 3], [3, 4], [4, 5]

Minimums of each: min(1,2)=1, min(2,3)=2, min(3,4)=3, min(4,5)=4

Minimums set: {1, 2, 3, 4}

Maximum of these minimums: **4**

Constraints

$1 \leq n \leq 10^6$

$1 \leq k \leq n$

$1 \leq arr[i] \leq 10^9$

```cpp
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>

using namespace std;
int findMaxOfMins(int n, const vector<int>& arr, int k) {
    deque<int> dq;
    int maxOfMins = -2147483648;

    for (int i = 0; i < n; i++) {
        if (!dq.empty() && dq.front() == i - k) {
            dq.pop_front();
        }
        while (!dq.empty() && arr[dq.back()] >= arr[i]) {
            dq.pop_back();
        }

        dq.push_back(i);
        if (i >= k - 1) {
            maxOfMins = max(maxOfMins, arr[dq.front()]);
        }
    }
    return maxOfMins;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, k;
    if (!(cin >> n >> k)) return 0;

    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << findMaxOfMins(n, arr, k) << endl;

    return 0;
}
```

Question 5

Problem Description
You are given an integer array arr of size n. You need to sort the array using a specific algorithm that follows these rules:

Find the lexicographically smallest pair of indices (i, j) such that
$0 \leq i < j \leq n-1, arr[i] > arr[j]$

A pair of indices (i_1, j_1) is lexicographically smaller than
(i_2, j_2) if $\text{i\_1} < \text{i\_2}$, or if $\text{i\_1} = \text{i\_2}$ and $\text{j\_1} < \text{j\_2}$.
If no such pair exists (the array is sorted), the algorithm stops.
Otherwise, swap arr[i] and arr[j] and repeat from step 1.
Return the total number of swaps performed by the algorithm until it terminates.
Example 1:

Input: n = 4, arr = [5, 1, 4, 2]

Output: 4

Explanation:
1. The smallest pair (i, j) is (0, 1) because arr[0]=5 > arr[1]=1 . After swap: [1, 5, 4, 2]
2. The smallest pair (i, j) is (1, 2) because arr[1]=5 > arr[2]=4 . After swap: [1, 4, 5, 2]
3. The smallest pair (i, j) is (1, 3) because arr[1]=4 > arr[3]=2 . After swap: [1, 2, 5, 4]
4. The smallest pair (i, j) is (2, 3) because arr[2]=5 > arr[3]=4 . After swap: [1, 2, 4, 5]

Sorted. Total swaps = 4.
Example 2:
Input: n = 3, arr = [1, 2, 3]
Output: 0
Explanation: The array is already sorted, so no swaps are neede
Constraints:

$1 \leq n \leq 10^5, 1 \leq arr[i] \leq 10^9$

```cpp
#include <bits/stdc++.h>
using namespace std;

long long mergeAndCount(vector<int>& arr, vector<int>& temp, int left, int right);
long long merge(vector<int>& arr, vector<int>& temp, int left, int mid, int right);

long long howManySwaps(vector<int>& arr) {
    int n = arr.size();
    vector<int> temp(n);
    return mergeAndCount(arr, temp, 0, n - 1);
}

long long mergeAndCount(vector<int>& arr, vector<int>& temp, int left, int right) {
    long long count = 0;
    if (left < right) {
        int mid = left + (right - left) / 2;
        count += mergeAndCount(arr, temp, left, mid);
        count += mergeAndCount(arr, temp, mid + 1, right);
        count += merge(arr, temp, left, mid, right);
    }
    return count;
}

long long merge(vector<int>& arr, vector<int>& temp, int left, int mid, int right) {
    int i = left;      // Left subarray index
    int j = mid + 1;   // Right subarray index
    int k = left;      // Temp array index
    long long invCount = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            invCount += (mid - i + 1);  // Count inversions
        }
    }

    while (i <= mid)
        temp[k++] = arr[i++];

    while (j <= right)
        temp[k++] = arr[j++];

    for (int idx = left; idx <= right; idx++)
        arr[idx] = temp[idx];

    return invCount;
}

int main() {
    vector<int> arr = {5, 1, 4, 2};
    cout << howManySwaps(arr);  // Output: 4
    return 0;
}
```

Question 6

Minimum Cost to Reach at Least Target Weight

You are given an integer **n**, an integer array **cost** of length **n**, and an integer **minWeight**.

The **i-th** item has:

- **weight = 2^i**
- **cost = cost[i]**

You may purchase **any number of each item**. Return the **minimum total cost** such that the **sum of the weights of the purchased items is at least minWeight**.

Example 1

Input

n = 5 cost = [2,5,7,11,25] minWeight = 26

Output

37

Explanation

Buy 3 items of weight 8 and 2 items of weight 1

Total weight = 24 + 2 = 26

Total cost = 33 + 4 = 37

Constraints

$1 \leq n \leq 60$

$1 \leq cost[i] \leq 10^9$

$1 \leq minWeight \leq 10^{18}$

```cpp
class Solution {
public:
    long long minimumCost(int n, vector<long long>& cost, long long minWeight) {
        // Normalize costs
        for (int i = 1; i < n; i++) {
            cost[i] = min(cost[i], 2 * cost[i - 1]);
        }

        long long ans = INT_MAX;
        long long currentCost = 0;
        long long remaining = minWeight;

        // Greedy from largest to smallest
        for (int i = n - 1; i >= 0; i--) {
            long long weight = 1LL << i;
            long long take = remaining / weight;

            currentCost += take * cost[i];
            remaining -= take * weight;

            // Option to overshoot
            ans = min(ans, currentCost + (remaining > 0 ? cost[i] : 0));
        }

        return ans;
    }
};
```