Standard template library

Upper Bound vs Lower Bound

Upper-Bound

It returns an iterator pointing to the **first** element in the range [first, last) that is **greater** than value, or last if no such element is found.

Lower-bound

lower_bound returns an iterator pointing to the first element in the range [first,last) which has a value **not less** than ‘val’.

Priority_Queue

- However in C++ STL, by default, the top element is always the greatest element.
- We can also change it to the smallest element at the top.
- Priority queues are built on the top to the max heap and uses array or vector as an internal structure.

Min Heap From Priority Queue

How to create a min-heap for the priority queue?
We can pass another parameter while creating the priority queue to make it a min heap. C++ provides the below syntax for the same.

```
Syntax:
priority_queue <int, vector<int>, greater<int>> g = gq;

Element
Int
Vector<int>
greater<int>
```

Method	Definition
priority_queue::empty()	Returns whether the queue is empty.
priority_queue::size()	Returns the size of the queue.
priority_queue::top()	Returns a reference to the topmost element of the queue.
priority_queue::push()	Adds the element ‘g’ at the end of the queue.
priority_queue::pop()	Deletes the first element of the queue.
priority_queue::swap()	Used to swap the contents of two queues provided the queues must be of the same type, although sizes may differ.
priority_queue::emplace()	Used to insert a new element into the priority queue container.
priority_queue value_type	Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter.

Multiset

Multisets are a type of associative containers similar to the set, with the exception that multiple elements can have the same values.

- begin() – Returns an iterator to the first element in the multiset -> O(1)
- end() – Returns an iterator to the theoretical element that follows the last element in the multiset -> O(1)
- size() – Returns the number of elements in the multiset -> O(1)
- max_size() – Returns the maximum number of elements that the multiset can hold -> O(1)
- empty() – Returns whether the multiset is empty -> O(1)
- insert (x) – Inserts the element x in the multiset -> O(log n)
- clear () – Removes all the elements from the multiset -> O(n)
- erase(x) – Removes all the occurrences of x -> O(log n)

Removing Element From Multiset Which Have Same Value:

- a.erase() – Remove all instances of element from multiset having the same value
- a.erase(a.find()) – Remove only one instance of element from multiset having same value
- Insertion of Elements- O(log N)
- Accessing Elements – O(log N)
- Deleting Elements- O(log N)

Multiset Question

Kiyora has n whiteboards numbered from 1 to n. Initially, the i-th whiteboard has the integer a_i written on it.

Koxia performs m operations. The j-th operation is to choose one of the whiteboards and change the integer written on it to b_j.

Find the maximum possible sum of integers written on the whiteboards after performing all m operations.

Input

Each test consists of multiple test cases. The first line contains a single integer t $(1 \leq t \leq 1000)$ — the number of test cases. The description of test cases follows.

The first line of each test case contains two integers n and m $(1 \leq n, m \leq 100)$.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ $(1 \leq a_i \leq 10^9)$.

The third line of each test case contains m integers $b_1, b_2, \ldots, b_m$ $(1 \leq b_i \leq 10^9)$.

Output

For each test case, output a single integer — the maximum possible sum of integers written on whiteboards after performing all m operations.

```
int t;
    cin>>t;
    while(t--){
        int n;
        cin>>n;
        int m;
        cin>>m;
        multiset<int>st;
        vector<int>v(n),vm(m);
        mac(i,0,n){
            cin>>v[i];
            st.insert(v[i]);
        }
        mac(i,0,m){
            cin>>vm[i];
            auto it=st.begin();
            st.erase(it);
            st.insert(vm[i]);
        }
        int sum=0;
        for(auto x: st){
            sum+=x;
        }
        cout<<sum<<nline;
    }
```

Sparse Table

- Sparse Table is a data structure, that allows answering range queries. It can answer most range queries in $O(\log n)$
- but its true power is answering range minimum queries (or equivalent range maximum queries).
- For those queries it can compute the answer in $O(1)$ time.
- The only drawback of this data structure is, that it can only be used on *immutable* arrays. This means, that the array cannot be changed between two queries.
- If any element in the array changes, the complete data structure has to be recomputed.

Set precision

- To set the precision of the double no then:
- cout<<setprecision(10)<<nline;