C++ Programming: Exam Variant 2 (Exam-2017-05-28)

Solutions for each task will be submitted in the form of compressed archive (.zip) files, containing .h and .cpp files.

Please be mindful of the strict input and output requirements for each task, as well as any additional requirements on running time, used memory, etc., as the tasks are evaluated automatically and not following the requirements strictly may result in your program's output being evaluated as incorrect, even if the program's logic is mostly correct.

For some of the tasks in this exam you are provided with files, which the Judge system places in your submitted solution. These files are the so-called "Solution Skeleton" and, depending on the task, may require you to write specific code for your solution to work (e.g. a Solution Skeleton may contain a file with the **`main()`** function defined, in which case your task will usually be to implement a class or function in another file, for the program to work correctly). DO NOT attempt to edit the Solution Skeleton files – the Judge system overwrites any files from the skeleton you submit, so it won't see your changes to them. Some tasks may contain additional files you can use (and edit) if you want – if so, this will be described explicitly in the task.

You can use C++03 and C++11 features in your code.

Unless explicitly stated, any integer **input** fits into **`int`** and any floating-point **input** can be stored in **`double`**.

Task 2 – IndexedSet (E2-Task-2-IndexedSet)

You are tasked with implementing methods for an **IndexedSet** class. An indexed set works just like a normal **std::set**, but also keeps an array of the sorted elements and can access the **i**th element immediately. For example, to get the **5**th element of a normal **std::set** you need to run a **for** loop **5** iterations from the **begin()** of the set – the **IndexedSet** supports the **operator[]** and you can directly ask for the **5**th element, retrieving it in constant time i.e. **O(1)**, just like with an array/vector.

To work fast, the **IndexedSet** doesn't constantly update the sorted array – it does a so-called "lazy-initialization" of the sorted array, creating it only when it is needed and clearing it when the **IndexedSet** is modified. More formally:

- The first time **operator[]** is called, the sorted array is built
- If **operator[]** is called and the sorted array exists, it is used without being rebuilt. If it doesn't exist (or is empty – implementations can vary), it is rebuilt and then used.
- If the **IndexedSet** is modified, the sorted array is cleared, so that the next **operator[]** call will rebuild it to match the current contents of the **IndexedSet**

You are given a skeleton containing the declaration of an **IndexedSet** class and its members. You need to implement the **IndexedSet** class in a new **IndexedSet.cpp** file:

- Fields **std::set<Value> valueSet** and **Value * valuesArray** represent the actual set and the sorted array of the set's elements, correspondingly
- **IndexedSet()** – constructs an empty **IndexedSet**
- **IndexedSet(const IndexedSet& other)** – copy-constructs **IndexedSet**
- **void add(const Value& v)** – adds an element to the **IndexedSet** (and clears the sorted array, so that the next call to **operator[]** will rebuild it)
- **size_t size() const** – returns the size of the set (the number of elements)
- **const Value& operator[](size_t index)** – returns a **const** reference to the element at the given index (and constructs the sorted array, if necessary, as described above). *Hint: don't worry about* **const***, just return the value at that position in the array*
- **IndexedSet& operator=(const IndexedSet& other)** – copy-assigns **IndexedSet**
- **~IndexedSet()** – destructs **IndexedSet** (only necessary to clear the sorted array)
- **buildIndex()** and **clearIndex()** are intended as helper methods you can define to create and clear the sorted array – they aren't used in external code, so you can chose whether you want to implement them

The skeleton also contains the **main.cpp** file, which defines the **main()** function. The program in **main.cpp** reads a series of arrays from the console, then reads a line of indices from the console. It then finds the array, for which the sum of the indices (read last from the console) of its elements – when the elements are sorted and the duplicates removed – is the maximum of the given arrays, and prints it in that form (sorted with duplicate values removed). To do that, it uses the **IndexedSet**.

For example, given the following arrays:

```
1 1 7 2 7 3 1
4 12 2 8 8
5 6 5 1 1 1
```

and the indices **0** and **2**, the program will sort and remove duplicates from the arrays:

```
1 2 3 7
2 4 8 12
1 5 6
```

and will print the result **2 4 8 12**, because that array has the maximum sum of the indices **0** and **2**

(the sum is **2 + 8 = 10**, which is bigger than both **1 + 3** and **1 + 6** for the other two arrays). The program uses **IndexedSet** to get a representation of the sorted arrays with removed duplicates.

Input

The program defined in **main.cpp** reads the following input:

One or more lines from the standard input, containing the input arrays, ending with a line containing the string **end**. Then one more line containing the indices, which will be used for the sums for determining the output array.

Output

The program defined in **main.cpp** writes the following output:

The sorted array with duplicates removed, which has the highest sum of the indices defined in the input (the sum is calculated after the array is sorted and its duplicates are removed).

Restrictions

There will be between **1** and **50** (inclusive) arrays in the input. Each array will have no more than **2000** elements, and each of its elements will be a value between **0** and **2000** (inclusive).

There will be between **1** and **5000** indices in the input. No index will be larger than the number of elements in the smallest array after all its duplicate elements have been removed. That is, there is no need to add range checks and validation logic to the **IndexedSet** class.

The total running time of your program should be no more than **0.3s**

The total memory allowed for use by your program is **8MB**

Submission Instructions

Submit only a single file – **IndexedSet.cpp**, containing the implementation of the **IndexedSet** class, as declared in **IndexedSet.h**.

Example I/O

Example Input	Expected Output
5 1 3 7 9 3 2 1 4 12 10 9 8 10 100 15 2 3 4 end 0 1 2	8 9 10 12
1 1 1 2 3 2 1 1 3 3 3 end 1 1 1 1	1 3