CS2710 - Programming and Data Structures Lab

Lab 13 (graded)

Oct 30, 2024

Instructions

- For each question, first think and write up the pseudocode of your solution in the answer sheet provided, then code it up, test it and upload tested code to moodle. **Remember to submit the answer sheet containing the pseudocode** to your TA, as it will also be graded, besides the uploaded code.
- You are expected to solve ALL the problems in the lab, using the local computer, C++ language, and g++ compiler. (Bonus problems can fetch 5% capped bonus, and Optional problems are just for fun and won't be graded.)
- You should submit your code to the course **moodle** on time, i.e., on/before 4.45pm (so that TAs can subsequently grade your submissions for graded lab assignments using the private test cases).
- You must strictly adhere to the following naming convention for your .cpp files and the single .zip file submission. For example, for a Lab Session 13 consisting of 2 questions, a student with roll number CS23B000 should
  - Name their .cpp files as **CS23B000_LAB13_Q1.cpp** and **CS23B000_LAB13_Q2.cpp**
  - Put both these .cpp files in a directory named **CS23B000_LAB13**
  - Zip this directory into a file named **CS23B000_LAB13.zip**
  - Submit only this single .zip file to moodle.
- If you need assistance, ask your TA, not your classmate.
- Internet access and mobile devices are prohibited in the lab.
- Check course Moodle for the public test cases and the evaluation script, which you can use to test your programs.
- Priority Queues are implemented as heaps in **std::priority_queue**. You can use it for Q2, Q3, Q4, but **for Q1, you must not use it**. Please clarify with the TAs if you have any questions about which data structures to use or avoid.

---

1. [4-ARY HEAPS] There is evidence suggesting that 4-ary heaps, i.e., heaps which uses its internal array to represent a complete 4-ary tree, may perform better than binary heaps in practice. In this task, implement a Priority Queue based on a 4-ary Heap. The priority queue supports 3 operations:

   - Insert - *Insert new elements into the Priority Queue (Keys will be distinct)*
   - DeleteMax - *Remove the largest key*
   - Maximum - *Print the largest Key, or "Empty" if PQ is empty*

   You may start from the binary MaxHeap implementation in class and change it accordingly. In your answer sheet, write the indices of the children of the node stored at index i of the internal array (0-based indexing may make calculations easier), and list down the pros/cons of a 4-ary heap compared to a binary heap (by also writing down how the running time complexity of the above three operations change between binary and 4-ary heap).

   A list of operations is given as input. You must perform those operations sequentially, and output the result of the Maximum operations.

   Input Format
   The first line of the input contains a single integer n, the number of operations. The following n lines, each of which is one of the following:

   - Insert *xyz* , where *xyz* is an integer.
   - DeleteMax
   - Maximum

   Output Format
   For every Maximum operation, print the maximum key in a new line. Each line contains a single integer. If the heap is empty when *Maximum* is performed, print *Empty*. If the heap is empty when *DeleteMax* is performed, do not print anything.

   Constraints:

   - Number of operations, $n \leq 1000$.
   - $-10^4 \leq$ Keys $\leq 10^4$.

   Example:

   Input:
   7
   Maximum
   Insert 45
   Insert 85
   Maximum
   DeleteMax
   Insert 55
   Maximum

   Output:
   Empty
   85
   55

2. [GREEDY DURING FESTIVAL TIME!] During your favorite festival, you receive a box of N laddoos. The laddoos are of different sizes, $A[1], A[2], \ldots, A[N]$. You have K moves, and in each move you can pick any one laddoo and eat half of it. Specifically, you eat $\lfloor A[i]/2 \rfloor$ of the chosen laddoo i, with $\lceil A[i]/2 \rceil$ laddoo remaining. Your goal is code up a strategy to eat as much total amount of laddoo as possible (i.e., total amount across the K moves).

   You can use `std::priority_queue`. Expected time complexity is $O(N + K \log N)$.

   In the answer sheet provided, before coding up your solution, (i) write the psuedocode that uses a priority queue to answer this question, and argue briefly why it is a correct solution, and why it achieves the above expected running time. (ii) What is the running time complexity you could achieve using a naive array/vector based algorithm that uses only sorting and/or binary search algorithms?, and finally (iii) if you are health-conscious and want to eat as less amount of laddoo as possible in K moves, then what will your best strategy be (write down this health-conscious strategy in the answer sheet; no need to code it up).

   Input Format
   The first line of the input contains (N, K) two space separated integers N (number of laddoos) and K (number of moves). The second line contains N space separated integers, which are the sizes of the N laddoos.

   Output Format
   Print a single integer value, that is the maximum amount of laddoo you can eat (summed across all moves).

   Constraints:

   - Number of laddoos, $N \leq 1000$.
   - $0 \leq$ A[i] $\leq 10^4$.

   Example 1:
   Input:
   5 4
   8 10 8 8 8
   Output:
   17

   Explanation: We eat half of the largest 4 laddoos.

   Example 2:
   Input:
   4 4
   21 8 5 3
   Output:
   22

   Explanation: $(21, 8, 5, 3) \rightarrow (11, 8, 5, 3) \rightarrow (6, 8, 5, 3) \rightarrow (6, 4, 5, 3) \rightarrow (3, 4, 5, 3)$

3. [(BONUS) MERGE K ARRAYS] You have been given K sorted arrays. Your task is to merge these sorted arrays into a single sorted array. Expected time complexity is $O(n \log k)$. *Here, n is the total length of all the arrays.*

   You can use `std::priority_queue` (which supports storage of duplicate elements, i.e., more than one element with the same priority).

   Input Format
   The first line of the input contains two space separated integers: K and M. Here, K is the number of sorted arrays, and M is the number of elements of each array. From line 2 onwards: M space separated integers in non-decreasing order, corresponding to the M elements of that sorted array.

   Output Format
   A single line containing all $K \times M$ elements in sorted order.

   Constraints:

   - $K, M \leq 1000$.
   - $0 \leq$ A[i] $\leq 10^4$.

   Example:
   Input:
   3 4
   1 5 10 15
   2 3 4 5
   6 8 9 10
   Output:
   1 2 3 4 5 5 6 8 9 10 10 15

4. [(OPTIONAL) SMALLEST HITTING RANGE] You are given K sorted lists of integers. Find the smallest range (that is, a min_value and a max_value) such that each list contains at least one element in that range. In case of multiple smallest ranges, print the range with smaller min and max values.

   Input Format
   The first line of the input contains two space separated integers: K and M. Here, K is the number of lists, and M is the number of integers in each list. From line 2 onwards: M space separated integers in non-decreasing order, corresponding to the M elements of a sorted array.

   Output Format
   2 space separated integers that represent the start and end of the range.

   Example1: If
   arr1[] = 4, 7, 9, 12, 15
   arr2[] = 0, 8, 10, 14, 20, and
   arr3[] = 6, 12, 16, 30, 50, then
   the answer is [6 8] (because smallest range is formed by number 7 from the first list, 8 from second list and 6 from the third list).

   Example2: If
   arr1[] = 4, 7,
   arr2[] = 1, 2, and
   arr3[] = 20, 40, then
   the answer is [2 20] (because [2, 20] is the smallest range containing at least one element (4 or 7, 2, and 20) from all the three arrays).