Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer

5. November 2018

# Datenstrukturen & Algorithmen — Blatt P7 — HS 18

Please remember the rules of honest conduct:

- Programming exercises are to be solved alone
- Do not copy code from any source
- Do not show your code to others

**Hand-in:** Sunday, 18. November 2018, 23:59 clock via Online Judge (source code only).
Questions concerning the assignment will be discussed as usual in the forum.

## Exercise P7.1 *Bitcoins.*

As an employee of the Federal Tax Administrations office in Switzerland you have been tasked to monitor bitcoin transactions. In particular, you are interested to discover the range of the 1/3 most valuable transactions ever made using bitcoin. The bitcoin network is a distributed database that constantly gets updated, and continuously grows in size. As a result, you need to create a live system that can efficiently consume new transactions, and report the range from the $\lfloor n/3 \rfloor$-th most valuable transaction to the most valuable transaction. Therefore, your system supports two operations:

1. **Insert**. The insertion is done by entering the number 1 on the standard input, followed by a number $V$ ($1 \leq V \leq 10^9$) that indicates the (integer) value of the new transaction. Every time a transaction is added, the number of transactions $n$ in the system is increased by 1.
2. **Report**. The reporting is done by entering the number 2 on the standard input. Then the monitoring system will report the $\lfloor n/3 \rfloor$-th transaction and the first transaction, assuming that all $n$ transactions have been previously sorted in a decreasing order. If $n < 3$ at the time the reporting routine is being invoked, the system will print out the message *Not enough transactions*.

Note that as the monitoring system is live, it is capable of executing both operations in any order (i.e., *insert* and *report* can come one after the other) and the *report* routine can be as frequent as the *insert* routine. Also note that every time the *report* routine is invoked, it will perform an analysis on the transactions already available by the system.

**Input** The first line of the input consists of the number $Q$ ($1 \leq Q \leq 5 \cdot 10^5$) that indicates the number of routines that will be invoked. Each of the next $Q$ lines contain either an *insert* routine in the form of "`1 V`" or a *report* routine in the form of "2" as described above.

**Output** The output consists of $R$ ($R \leq Q$) lines such that $R$ corresponds to the number of *report* routines present in the input. Each line is either the message *Not enough transactions* or two numbers $L$ and $H$ in the form of "`L - H`" such that $L$ is the $\lfloor n/3 \rfloor$-th most valuable transaction and $H$ is the most valuable transaction present in the system when the *report* routine was invoked. The output is terminated with an end-line character.

**Grading** You get 3 bonus points if your program works for all inputs. Ideally, your algorithm should require $O(1)$ time for the *report* routine and $O(\log(n))$ for the *insert* routine (with reasonable hidden constants). Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=25012&problem=AD18H7P1`. The enrollment password is "`asymptotic`".

**Example**

*Input:*

```
12
1 1
1 7
2
1 9
1 8
1 5
1 6
2
1 21
2
1 9
2
```

*Output:*

```
Not enough transactions
8 - 9
9 - 21
9 - 21
```

A detailed explanation for the 12 routines above:

1. Insert 1 to the array. Current array is `[1]`.
2. Insert 7 to the array. Current array is `[7,1]`.
3. Report. Array size is less than 3. Output is `Not enough transactions`.
4. Insert 9 to the array. Current array is `[9,7,1]`.
5. Insert 8 to the array. Current array is `[9,8,7,1]`.
6. Insert 5 to the array. Current array is `[9,8,7,5,1]`.
7. Insert 6 to the array. Current array is `[9,8,7,6,5,1]`.
8. Report. Array size is 6. $\lfloor n/3 \rfloor = 2$, and the 2-nd element in the sorted array is 8 and highest is 9, therefore $L = 8$ and $H = 9$. Output is `8 - 9`.
9. Insert 21 to the array. Current array is `[21,9,8,7,6,5,1]`.
10. Report. Array size is 7. $\lfloor n/3 \rfloor = 2$, therefore $L = 9$ and $H = 21$. Output is `9 - 21`.
11. Insert 9 to the array. Current array is `[21,9,9,8,7,6,5,1]`.
12. Report. Array size is 8. $\lfloor n/3 \rfloor = 2$, therefore $L = 9$ and $H = 21$. Output is `9 - 21`.

**Notes** For this exercise we provide an archive containing a program template available at `https://www.cadmo.ethz.ch/education/lectures/HS18/DA/uebungen/AD18H7P1.Bitcoins.zip` The archive also contains additional test cases (which differ from the ones used for grading). Importing any additional Java class is **not allowed** (with the exception of the already imported ones `java.io.`{`InputStream`, `OutputStream`} and `java.util.Scanner` class).

**Solution.**

The solution is based on using two heaps, a min-heap, and a max-heap, and is given below:

```java
public static void read_and_solve(InputStream in, PrintStream out)
{
    Scanner scanner = new Scanner(in);
    int Q = scanner.nextInt();

    MaxHeap maxHeap  = new MaxHeap (2 * Q / 3 + 2);
    MinHeap minHeap  = new MinHeap (Q / 3 + 2);
    int     maxValue = Integer.MIN_VALUE;

    while (Q-- > 0) {
        int command = scanner.nextInt();
        if (command == 1) {
            maxHeap.insert(scanner.nextInt());
            if (maxHeap.size() == 3) {
                maxValue = maxHeap.pop();
                minHeap.insert(maxValue);
                break;
            }
        } else if (command == 2) {
            out.println("Not enough transactions");
        }
    }
    while (Q-- > 0) {
        int command = scanner.nextInt();
        if (command == 1) {
            // Get the value, and update the max if necessary,
            // then insert the value into one of the heaps
            int value = scanner.nextInt();
            if (value > maxValue) maxValue = value;
            if ((maxHeap.size() + minHeap.size()) % 3 == 2) {
                minHeap.insert(value);
            } else {
                maxHeap.insert(value);
            }
            // Now check whether we need to transfer an element
            // from one heap to the other.
            if (minHeap.top() < maxHeap.top()) {
                maxHeap.insert(minHeap.pop());
                minHeap.insert(maxHeap.pop());
            }

        } else if (command == 2) {
            out.println(minHeap.top() + " - " + maxValue);
        }
    }

    scanner.close();
}
```

## Explanation

The first observation is, that as we are given $Q$ routines, we can insert at most $Q$ elements. As a result, we need memory space of $O(Q)$ in the worst case scenario, or in other words, possibly an array of size $n$ such that $n = Q$. As we need to output two values, namely the 1-st and the $\lfloor n/3 \rfloor$-th value once the array is sorted in descending order, we can maintain a single variable that represents the max for the former one and update this value upon each insertion.

For the rest of the values, we could maintain an array of size $n$. However, as new transactions are inserted into the system, $n$ as well as the array would grow, and thus to obtain the $\lfloor n/3 \rfloor$-th instruction, we would have to sort the array. Depending on the choice of the algorithm for sorting, this could take us at least $O(n)$ complexity, assuming that insertions sort is used every time we insert a new element. As such we would not be able to insert in $O(log(n))$ time, as suggested in the problem statement.

We could maintain a heap for the highest $\lfloor n/3 \rfloor$ transaction, in particular a min-heap. This would allow us insertion time in $O(log(n))$ time, and the $\lfloor n/3 \rfloor$-th highest transaction will always stay at the top of the heap, allowing us to access it in $O(1)$ time. However, as $n$ grows, we should be able to fetch the next highest transaction (smaller than the one on the top of the min-heap) and add it to the min-heap in order to be able to return the $\lfloor n/3 \rfloor$-th highest transaction. This means that the rest of $n - \lfloor n/3 \rfloor$ transactions must be somehow sorted.

To achieve both, we can maintain **two heaps**, a min-heap for the highest $\lfloor n/3 \rfloor$ transactions and a max-heap for the rest of $n - \lfloor n/3 \rfloor$ transactions. In order to make sure that the $\lfloor n/3 \rfloor$-th transaction is always at the top of the min-heap, the following property must hold:

$$\text{min-heap.top()} \geq \text{max-heap.top().} \tag{1}$$

Every time, a new transaction is being inserted, we can insert it in one of the heaps. Namely, every third insertion, we can add it to the min-heap, to make sure that it holds $\lfloor n/3 \rfloor$ elements as $n$ grows, and every other transaction we insert it in the max-heap. Then, after a transaction is inserted, we check whether property (1) holds. If it doesn't, we pop the top elements from both heaps, and we transfer them in the opposite heap. In such scenario, insertion will take $O(log(n))$, as inserting and removing elements from heap takes $O(log(n))$, and the $\lfloor n/3 \rfloor$-th highest transaction will always be at the top of the min-heap, thus accessible in $O(1)$.

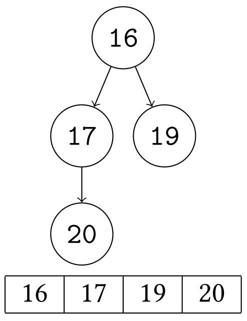
The algorithm above works as follows:

1. (lines 6-8) It maintains two structures, a min-heap represented with variable `minHeap` and a max-heap represented with variable `maxHeap`. It also maintains a single variable `maxValue`, that will hold the transaction with the maximal value. The min-heap is intended to store the first $\lfloor n/3 \rfloor$ values, while the max-heap the rest of the $n - \lfloor n/3 \rfloor$ values, and as a result both are initialized with sufficient memory space.
2. (lines 10-22) It takes care of inserting the first 3 transactions and prints *Not enough transactions* in case the reporting routine is invoked.
3. (lines 23-45) Either prints the $\lfloor n/3 \rfloor$-th and the 1-st highest transactions, or inserts a transaction in one of the heaps. At each insertions, the algorithm will make sure that property (1) holds.

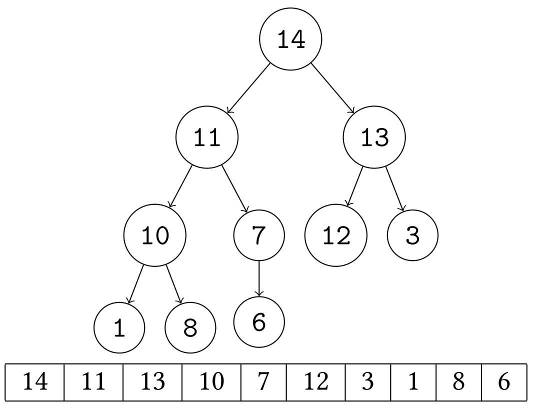
To illustrate the algorithm, consider the following list of 15 transactions that have to be inserted into the system (the order is defined from left to right):

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 12 | 14 | 17 | 16 | 1 | 7 | 13 | 19 | 3 | 10 | 20 | 8 | 6 | 4 |

Now let's assume that we have already inserted 14 transactions into the system. The min-heap structure will contain the highest $\lfloor n/3 \rfloor = \lfloor 14/3 \rfloor = 4$ transactions, while the max-heap will contain the rest of the 10 transaction. In the min-heap, the $\lfloor n/3 \rfloor$ transaction (assuming transactions are sorted in decreasing order) is at the top of the heap. And thus is accessible in $O(1)$. In the max-heap, the next highest transaction is accessible at the top of the heap as well, and thus accessible in $O(1)$ too.

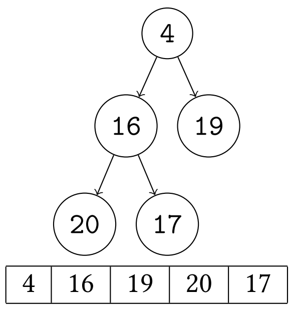


Min-Heap

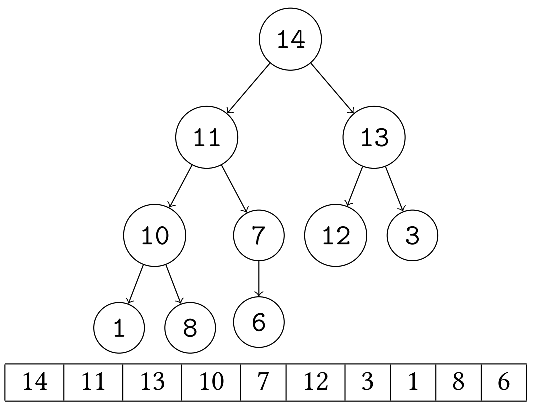


Max-Heap

Now, lets insert transaction with value of $4$ into the structure. As it is the 15-th transaction in a row, we insert it into the min-heap structure.

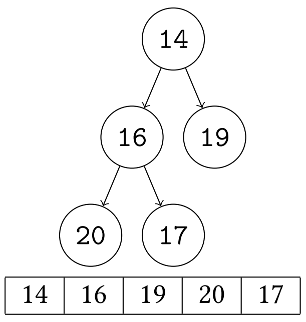


Min-Heap

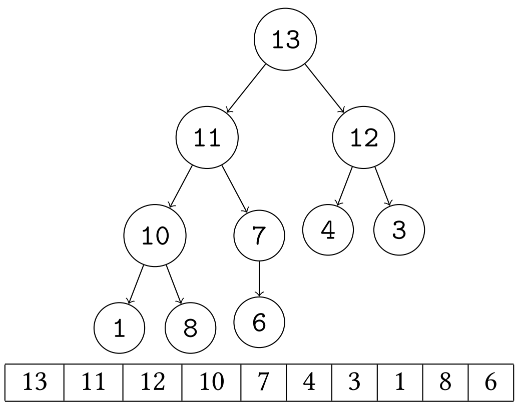


Max-Heap

Property (1) no longer holds. Therefore, we pop the top element from the min-heap (with value 4), and we pop the top element from the max-heap (with value 14). Then we insert element with value 4 into the max-heap, and we insert element with value 14 into the min-heap. Finally, property (1) holds again, and we obtain:



Min-Heap



Max-Heap

**Exercise P7.2** *Mountain Trip.*

A road is $n$ kilometers long and passes through several cities. Each city can be either a mountain city or a sea city. There are $M$ mountain cities, the $i$-th of which is located $m_i$ kilometers after the beginning of the road. Similarly, there are $S$ sea cities and the $i$-th sea city is located $s_i$ kilometers after the beginning of the road ($m_i$ and $s_i$ are integers between 0 and $n$, endpoints included, and each kilometer of the road can traverse at most one city).

A travel agency offers $T$ possible trips. The $i$-th trip starts from kilometer $b_i$ and ends at kilometer $e_i$ of the road, visiting all the cities in-between (endpoints included). Alice wants to buy a trip that visits the largest number of mountain cities and that does not visit any sea city.

Your task is to design an algorithm that finds the best trip for Alice.

**Input** The input consists of a set of instances, or *test-cases*, of the previous problem. The first line of the input contains the number $C$ of test-cases, and each test-case consists of 5 lines. The first line of each test-case contains the four integers $n$, $M$, $S$, and $T$. The second line contains $M$ integers, where the $i$-th integer is the position $m_i$ of the $i$-th mountain city. The third line contains $S$ integers, where the $i$-th integer is the position $s_i$ of the $i$-th sea city. The fourth line contains $T$ integers, where the $i$-th integer is the number $b_i$. Finally, the fifth line also contains $T$ integers, where the $i$-th integer is the number $e_i$.

**Output** The output consists of $C$ lines, where the $i$-th line is the answer to the $i$-th test-case and contains the index of the best trip, i.e., an integer $t$ such that $1 \leq t \leq T$ and:

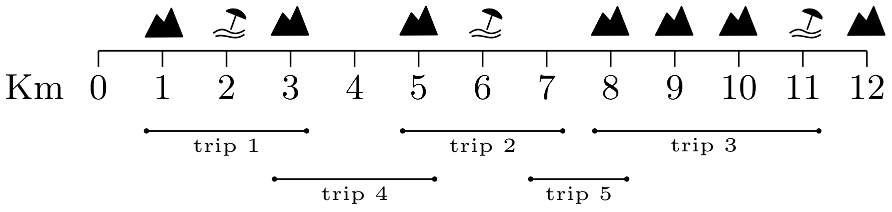
(1) there exists no $j$ such that $b_t \leq s_j \leq e_t$;

(2) for every index $r \neq t$ that satisfies condition (1), $|\{j : b_r \leq m_j \leq e_r\}| < |\{j : b_t \leq m_j \leq e_t\}|$.

You can assume that such an index $t$ always exists.

**Grading** This exercise awards no bonus points. Your algorithm should require $O((M + S + T)\log(M + S))$ time (with reasonable hidden constants). Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=25012&problem=AD18H7P2`. The enrollment password is "`asymptotic`".

**Example**



*Input (corresponding to the instance in the previous picture):*

```
1
12 7 3 5
10 8 5 3 9 1 12
6 2 11
1 5 8 3 7
3 7 11 5 8
```

*Output:*

```
4
```

**Notes** For this exercise we provide an archive containing a program template available at `https://www.cadmo.ethz.ch/education/lectures/HS18/DA/uebungen/AD18H7P2.MountainTrip.zip` The archive also contains additional test cases (which differ from the ones used for grading). Importing any additional Java class is **not allowed** (with the exception of the already imported ones `java.io.`{`InputStream`, `OutputStream`} and `java.util.Scanner` class).

**Solution**.

We first consider the following auxiliary problem: given a sorted vector $A = \langle a_1, a_2, \dots, a_\eta \rangle$ of $m$ distinct integers and two additional numbers $x, y$, compute the number $N(A, x, y)$ of elements $a$ in $A$ such that $x \le a \le y$. This problem can be solved in $O(\log \eta)$ time by performing two binary searches on $A$: the first binary search looks for the largest index $i \le m$ such that $a_i < x$, while the second binary search looks for the smallest index $j > 0$ such that $a_j > y$.[1]. The elements of $A$ between $x$ and $y$ are exactly the ones in the sub-array $\langle a_{i+1}, a_{i+2} \dots, a_{j-2} \rangle$ and hence $N(A, x, y) = j - i + 1$.

To solve the original problem we first sort the arrays $\langle s_1, \dots, s_S \rangle$ and $\langle m_1, \dots, m_M \rangle$ containing the positions of the sea and mountain cities, respectively. Let $S'$ and $M'$ be the sorted vectors, respectively, and notice that this step requires $O(S \log S + M \log M)$ time (e.g., using Mergesort). Then, we examine one trip ad a time: when the $i$-th trip is considered we check whether $N(S', b_i, e_i) > 0$ and, if this is the case, the trip is ignored. Otherwise, if $N(S', b_i, e_i) = 0$, we compute the number $N(M', b_i, e_i)$ of mountain cities visited by the trip and we keep track of the best trip examined so far. The time required by this step is $O(\log S + \log M)$ per trip, therefore the total time spent by the algorithm is $O((T + S) \log S + (T + M) \log M) = O((M + S + T) \log(M + S))$.

[1]If $a_0 \ge x$ then let $i = 0$. If $a_m \le y$ then let $j = m + 1$.