FIT2004 S2/2016: Assessment questions for week 4

THIS PRAC IS ASSESSED! (6 Marks)

DEADLINE: Monday, 15-Aug-2016 23:59:00
CLASS: This programming exercise has to be completed before the deadline. Your demonstrator will check your submission during the lab. He will ask you a series of questions to assess your understanding of this exercise, and gauge how you implemented it. It is required that you implement this exercise strictly using **Python programming language**. Practical work is marked on the performance of your program **and also on your understanding of the program**. A *perfect* program with zero understanding implies you will get **zero** marks! "Forgetting" is not an acceptable explanation for lack of understanding. Demonstrators are not obliged to mark programs that do not run or that crash.
After/befor your demonstrators have interviewed you, you are expected to work towards the programming competition the details of which will be released before the lab.
SUBMISSION REQUIREMENT: You will need to submit a zipped file containing your Python programs (named 1A.py, 1B.py, celebrity.py and median.py) as well as a PDF file briefly describing your solution and its space and time complexity. The PDF file must give an outline of your solution (e.g., a high level idea of how did you solve it) and the **worst-case** space and time complexity of your solution. The zipped file is to be submitted on Moodle before the deadline.
Important: You should carefully consider the border cases and make sure that you return correct results for all cases. Although you are not required to include the proof of correctness in your submissions, you are encouraged to formally prove the correctness of your programs. This will help you in identifying and fixing the bugs in your program if present. Data generators are provided to generate test cases. However, be mindful that tests do not guarantee the correctness.

1 Task 1

A company keeps hard copies of all the invoices it ever issued to its customer. The invoices are sequentially numbered from 1 to N where N is the total number of invoices.

During the annual audit, it was found that k invoices have gone missing. Luckily, the company's head office also keeps copies of all the invoices. Hence, if the invoice numbers of the missing invoices are known, the invoices can be retrieved from the company's head office. The hard copies of the invoices are sorted on the alphabetical order of customer names which makes it hard to find the invoice numbers of missing invoices.

Since you are a new hire, you are given the cumbersome task of finding the invoice numbers of the missing invoices. Adding to your misery, the manager tells you not to change the sorted order of the invoices (i.e., the invoices must remain sorted in alphabetical order). Your goal is to find the invoice numbers of all k missing invoices as efficiently as possible under each of the following circumstances:

- **Task 1-A.** You are given a pen and a paper. The paper is only big enough that you can only write the invoice numbers of all the invoices, i.e., you can only use $O(N)$ space. To demonstrate your solution, write a Python program that uses $O(N)$ space and finds the invoice numbers of all k missing invoices in $O(N)$ worst-case time complexity. Your program should be named 1A.py. ↳ O(N) complexity: can store cN+d elements

- **Task 1-B.** You are given a calculator but neither a pen nor a paper. This means that you are not able to record the invoice numbers of the non-missing invoices, i.e., you can only use $O(1)$ space. Assume that only one invoice has gone missing (i.e., $k = 1$). In this case, it is possible to find the missing invoice in $O(N)$ time complexity using only $O(1)$ space. Write a Python program with worst-case time complexity of $O(N)$ that uses $O(1)$ space and prints the invoice number of the missing invoice. Your program should be named 1B.py. O(n) complexity; can store c elements where c does not depend on input size.

1.1 Input

The input file consists of $N - k + 1$ lines. The first line of the input will consist of two values N and k where $0 < k \leq N < 1000000$. The remaining $N - k$ lines correspond to the unique invoice numbers of the $N - k$ non-missing invoices (each line contains one invoice number). Below is a sample input (for $N = 5$ and $k = 2$). Note that the non-missing invoice numbers are not sorted.

```
5 2
3
2
5
```

1.2 Output

The output should display the invoice numbers of missing invoices (one at each line) in ascending order of the invoice numbers. Below is the output for the above sample input.

```
1
4
```

1.3 Provided files

In the zipped file provided to you on Moodle, you will find a data generator (datagen.py) under the folder /missingInvoice. The generator takes N and k as input on two separate lines and generates an input file named "testdata.txt". You should test your program on several sample input files. However, note that the tests cannot guarantee the correctness of the algorithm. Therefore, you need to make sure that your algorithm is correct.

2 Task 2

Barack Obama is followed by 76, 454, 383 users on Twitter (at the time of writing this). Alice is wondering whether there are other celebrities with more followers. Somehow she gets access to the complete Twitter data set that contains the following information for each user: a unique

identifier (ID), the number of his/her followers, and the number of people he/she is following, and the total number of tweets by the user.

She decides to use this data set to find top-k celebrities on Twitter based on the number of their followers. She writes a Python program to sort the Twitter data set in descending order of the number of followers. However, the program does not work because the Twitter data set is *HUGE* and sorting it would be quite time consuming. More importantly, her computer does not have enough main memory to load all the records (e.g., she cannot create an array containing all the records). She knows that you are taking a course on Algorithms and Data Structures and calls you for help with high hopes.

Your goal is to write a Python program that prints top-k celebrities based on the number of followers. If two users have same number of followers, tie is broken by preferring the user with smaller ID. The program must use $O(k)$ space and run in $O(N \log k)$ where N is the total number of users in the Twitter data set. Since $k << N$, the program would run without any problem in contrast to the sorting algorithm that takes $O(N \log N)$ time and $O(N)$ space. Your program must be named celebrity.py.

2.1 Input

The input file will consist of $N+1$ lines. The first line contains two positive integers representing the value of N and k, respectively, where $1 < N < 10^9$ and $1 \leq k \leq 500$. Each of the remaining N lines contains a positive integer denoting the number of followers for the user, e.g., the first of the remaining N lines correspond to the number of followers for the user with ID 1, and i-th line corresponds to the user with ID i. Below is a sample input for $N = 5$ and $k = 2$.

```
5 2
7
5
1
7
10
```

2.2 Output

The output should print top-k celebrities in the sorted order of their ranks described in the criteria earlier. Each output line contains three integers separated by whitespace. The first integer represents the rank of the celebrity, the second is the user ID and the third is the number of his/her followers. Below is an output for the above sample input data.

rank ID #followers

```
1 5 10
2 1 7
```

The user 5 is ranked first and has 10 followers and the user 1 is ranked second and has 7 followers. Note that the users with IDs 1 and 4 both have 7 followers but the user 1 is given priority due to having a smaller ID.

2.3 Provided files

In the zipped file provided to you on Moodle, you will find a data generator (datagen.py) under the folder /celebrity. The generator takes N and k as input on two separate lines and generates an input file named "testdata.txt" which follows the format as described above. The folder also contains a file named naive.py which contains a straightforward algorithm taking $O(N \log N)$. Use the datagen.py to generate a very large data set (e.g., $N = 300,000,000$ and $k = 500$) and see if naive.py can handle it. Your program will be able to handle such data if it uses $O(k)$ space and takes $O(N \log k)$ time.

3 Task 3

The median of a list of N numbers can be found by sorting the numbers in ascending order and returning the $(N + 1)/2$-th number if N is odd or returning the average of $N/2$-th and $(N/2 + 1)$-th number if N is even. E.g., the median of the list $\{1, 2, 10, 15, 16\}$ is 10 whereas the median of $\{1, 2, 10, 15\}$ is $(2 + 10)/2 = 6$.

Consider an online scenario where you receive a stream of *positive* integers in unsorted order. Your goal is to *maintain* the median of integers, i.e., whenever you receive a new integer, your program must output the median of all the numbers seen so far.

One straightforward approach is to store the incoming integers in an array in a sorted order and returning the middle element (or the average of the two "middle" elements). However, this may take $O(N)$ even when you are using a Binary Search to find the correct location of the new integer in the array. This is because, to insert the integer in its correct location, you will need to shift all the elements greater than it to the right (just like we do in Insertion Sort). Below is an example. Consider that you have already received the integers 10, 20, 30, 40, 50 and 60 and you have somehow sorted them in the array as shown below.

Value	10	20	30	40	50	60
Index	0	1	2	3	4	5

Suppose the new integer that you receive is 5. You can use a Binary Search to determine that it is to be inserted at the index 0. However, to do this, you have to shift all existing integers towards right by 1 (as shown in the array below). This takes $O(N)$ in the worst-case where N is the number of integers seen so far.

Value	5	10	20	30	40	50	60
Index	0	1	2	3	4	5	6

Your goal is to write a Python program that reads the integers from an input file and, after reading each integer, prints the median of the integers seen so far. The worst-case time complexity of your program must be $O(\log N)$ (per incoming integer) where N is the number of integers seen so far. The program must be named median.py.

3.1 Input

Each line in the input file represents an integer. Below is a sample input file.

```
10
5
8
11
7
3
```

3.2 Output

Your program must print, for each integer read from the input file, the median of the integers seen so far. If the median is a floating point value, your program should print its floor value (e.g., the median of $\{10, 5\}$ is 7.5 and your program should display 7). Below is the output for the above input file.

```
10
7
8
9
8
7
```

3.3 Provided files

In the zipped file provided to you on Moodle, you will find a data generator (datagen.py) under the folder /median. The generator takes N as input and generates an input file named "testdata.txt" which follows the format as described above. The folder also contains a file named naive.py which contains a straightforward algorithm that reports median taking $O(N)$ per integer read from the input file. Use the datagen.py to generate a very large data set (e.g., $N = 1,000,000$) and observe that naive.py becomes increasingly slow when N, the number of integers read so far, increases. If you find an $O(\log N)$ algorithm, you will clearly see the performance difference for such large input files.

3.4 Hint

One possible solution is to use two heaps.

```
-=o0o=-
  END
-=o0o=-
```