FIT2004 Prac Week4 Documentation By Pavin Wu (26916592)

1A

Solution Outline:

Read the first line of the file to get N (number of invoices) and k (number of missing invoices).

Use N to construct an array of 0's with length n. This will allow us O(1) access time. Then, as we read each line, we found the invoice number. Name this *inv*. We use this number to mark the array index *inv*-1 as 1 (since index starts at 0, not 1). So when we write the missing invoices onto a file, we simply check for the entry in the array with value 0 and write it down sequentially.

Note: For simplicity, we created the array as a list in python, which will result in O(N) access time. However, we are focusing on the algorithm, not an implementation in a particular language.

Worst case space:

The invoices array takes O(N) space. Other variables such as N and k take O(1) space. So overall O(N) space.

Worse case time:

There are N+1 lines in the text file. So reading all the lines takes O(N). Marking array with 1 is O(1) each time. So O(N) worst case. Writing on text file in the end takes O(N). So overall O(N) time worst case.

1B

Solution Outline:

Read the first line of the file to get N (number of invoices) and k (number of missing invoices).

We need to check if k is 1. Since we assumed k to be only 1, return False otherwise.

Then, based on the sum of 1+2+...+N series we leant previously, we know we can write it as N*(N+1)/2. Call this an ideal sum. We can then sum all of the invoices number in a similar manner using for loop. Now, to find a missing number, we can subtract the ideal sum with the actual sum we get from for loop. Since we assumed only one invoice is missing, the result of subtraction is the invoice number.

Worst case space:

The variable used to hold the actual sum *sum_inv* does not change its size in the iteration. Its size is also not dependent on N. So this is O(1). The other variables such as N and k also take O(1). So overall, this takes O(1) space.

Worse case time:

There are N+1 lines in the text file. So reading all the lines takes O(N). Summing in each step takes O(1). All other operations such as subtraction and assignments also take O(1). So overall O(N) time worst case.

2 celebrity

Solution Outline:

First create an empty list to store celebrities' data. We will treat this as a min-heap. Then for each user from text file, heap-push the data into the list using the function *heappush()* from the library *heapq.py* (https://docs.python.org/3.0/library/heapq.html). Repeat this until length of the heap is k

Note that in case the users have the same number of follower, we need to multiply each ID by -1 so the smaller ID appear first in the list when sorting it in reverse.

Then, when length of the list is equal to k, we want to keep it in this size. So after this point, after we have heap-pushed a new user data, we also have to pop the user with minimum follower out of the heap. We can implement this using *heappop()* function from *heapq.py*. If we keep repeating this process for all remaining users, all the users with less followers will be popped out and discarded. The users remaining in the heap after the operation is finished will be the top k celebrities.

We then sort this heap list in reverse using built-in *sort()* method, and print the data (with ID multiplied -1 back) as required.

Worst case space:

The list of celebrity *celeb* take O(k) worst space complexity since it only stores up to k celebrities. As all the other variables such as *ID* and *line* take O(1) space, the overall space complexity is O(k).

Worse case time:

Heapq.py implements a priority queue data structure. This means insertion (push) and deletion (pop) will take O(logk) time where k is the size of the heap. Since having algorithm with the same time complexity running after the other will result in same overall time complexity, so creating the heap of top k celebrities will take O(logk) worst time in each iteration. Coupled this with the fact that the algorithm goes through all users data at least once. The overall worst time complexity is O(Nlogk).

Since python uses Timsort (https://en.wikipedia.org/wiki/Timsort) for its sorting algorithm, the worst time complexity is O(klogk) where k is the size of the list to be sorted. Also, Nlogk accelerates faster than klogk as a function since k << N. So Nlogk will be an upper-bound for klogk. This implies O(Nlogk) overall time complexity.

Finally, the for loop being used to print the list of celebrities at the end runs k times. This means O(k) time complexity. The same argument in the previous paragraph apply where Nlogk will dominate k as a function. Therefore, the overall worst time complexity for this algorithm is O(Nlogk).

3: median

Solution Outline:

We have a list of numbers from index 0, 1, ..., n. We can find the median of this list by using a min heap and a max heap. The min heap is used to store the numbers with index {(n+1)//2+1, (n+1)//2} ,...,n. The max heap is used to store the other half. Notice that the middle element will change depending on the state of the input number and the current median. This will be discussed below.

Due to the property of min-heap, the smallest number in a heap will be the root. Similarly, max-heap will have the largest number as a root. This means if the number of elements in the heaps are equal, in our case, the median will be (root of min-heap + root of max-heap)/2. If there is one more element in one heap than the other, the median will be the root of the heap with more elements.

Note that we have to keep the difference of the number of elements between the two heaps to no more than 1, otherwise the root will no longer be the median, but the number next to the median by the length k, where k=|(length of min-heap)-(length of max-heap)|-1. Call the state the difference in number of elements not greater than 1 be “heaps in balance”.

To implement this in python, we use the function *heappush()* and *heappop()* from the library heapq.py (https://docs.python.org/3.0/library/heapq.html). *heappush()* and *heappop()* are used to push in and pop out the root of the heap respectively. Therefore, we generally use *heappush()* to push item into min heap if it is >= median, and max heap otherwise. Only in the case when we know the heaps are going to be out of balance if we put an item in (as described in paragraph 3), only then we do *heappop()* to the heap with more elements and then put the popped item into the other heap by *heappush()*. Since heaps are always made balanced every time a new item is pushed in, we know heaps will always be in balance after each iteration.

Note also that heapq.py only deals with min-heap by default, so we have to multiply the items to be pushed into max-heap by -1, so heapq will complete the heap backwards. It follows that we multiply back -1 to the item we popped from max-heap if we want to use it.

Worst case space:

Each of the list representing a heap each have the size of ~N//2 where N is the Number of numbers. The total size of two lists will be the be N. Since the other variables such as median and num take O(1) space. The overall worst case space is O(N)

Worse case time:

From the lecture, pushing and popping item to and from a heap (i.e. *heappush()* and *heappop())* only require a comparison (between parent/children and siblings) only as many as the height of the tree, which is O(logN). Since pushing and popping in our algorithms are sequential, meaning that they are done after the previous O(logN) operations are finished, The pushing and popping have overall worst case time O(logN). Since all the other operations such as print and references take O(1) worst time. The overall worst time complexity is O(logN) per incoming integer.