CSC263 Fall 2017 Assignment 1

Name: Guanchun Zhao, Ruijie Sun
SN: 1002601847, 1003326046

Q1. SOLUTION

Let i $\in N$ and $2 \leq i \leq n$

In the line-5 loop (the inner loop), there are (n-i+1) times of basic operations.

In the line-2 loop (the outer loop), there are (n-1) times of the inner loops with different i from 2 to n.

Therefore, the upper bound running time T(n) = $\sum_{i=2}^{n}(n-i+1) = \sum_{j=1}^{n-1} j = n(n-1)/2$.

(a) Prove that $T(n) \in O(n^2)$

Want to show that $\exists\ b \in N$ and $c \in R^+$ such that $n > b \implies T(n) \leq c * n^2$.

Let b = 0, c = 1

By assumption, Let n $\in N$ and n> 0

$n^2 - n(n-1)/2$

$= n^2/2 + n$ (Because n > 0)

≥ 0

Thus, $n^2 \geq n(n-1)/2$.

Therefore, $T(n) \in O(n^2)$.

(b) Want to find an input family such that its running time is $\in \Theta(n^2)$.

Make list A be any repetition of [1,0,0,1], and the last repetition could be broken, like [1, 0, 0, 1, 1, 0].

Because i is from 2 to n for each iteration, x is the repetition of [0, 1].

In this input family, (A[i] + A[i-1]) mod 2 is the repetition of [1, 0].

That means (A[i] + A[i-1]) mod 2 is not equal to x all the time. Thus, the if statement always passes, which means the inner loop would run in each iteration of the outer loop.

As aforementioned, its running time is n(n-1)/2 because it passes all the iterations.

Thus, we conclude that $T(n) \in \Omega(n^2)$.

Q2. SOLUTION

a)

In the best case, the Line2 (if A[i] = k) will be executed for 1 time.

The reason behind it is that if value k exists in the (n-1) index of A, the Line 3 will be triggered in the first time iteration of for-loop. Therefore, the Line 2 is executed for only 1 time.

b)

The number of all possibilities is $2 \times 3 \times 4 \times \cdots \times n \times (n+1)$. But for the best case , as we have already known, the last number of A is k, so the number of all possibilities for best case is $2 \times 3 \times 4 \times \cdots \times n \times 1$.

So, $P = \frac{2\times3\times4\times\cdots\times n\times1}{2\times3\times4\times\cdots\times n\times(n+1)} = \frac{1}{n+1}$

c)

For the worst case, the Line 2 is executed for n times.

If $A[0] = k$ or value k doesn't exist in A, the for-loop will never stop until it iterates n times. Therefore, the Line 2 is executed for n times

d)

The number of all possibilities is $2 \times 3 \times 4 \times \cdots \times k \times (k+1) \times \cdots \times n \times (n+1)$.

Case 1, k is from $\{1\}$

Since there are two cases for the worst case: we find k in index 0 or we don't find k at all.

Therefore, $P = \frac{n!+n!}{2\times3\times4\times\cdots\times n\times(n+1)} = \frac{2}{n+1}$

Case 2, k is not from $\{0, 1\}$

Since this is the worst case, the value k can't exist from index $(k-1)$ to index $(n-1)$.

Meanwhile, all possible number for index 0 to index $(k-2)$ can be taken since all possible number in this index range are all less than k.

Therefore, $P = \frac{2\times3\times4\times\cdots\times k\times k\times(k+1)\times\cdots\times n}{2\times3\times4\times\cdots\times k\times(k+1)\times\cdots\times n\times(n+1)} = \frac{k}{n+1}$

Thus, P = $\begin{cases} \frac{2}{n+1} & \text{if } k = 1 \\ \frac{k}{n+1} & \text{Otherwise} \end{cases}$

e)

Let t_n be a random variable that denotes the number of comparisons executed (Line 2)

Let v denotes the index where value k exists in A

Let P(A,v) denotes the probability for input A,k, where A[v] = k

Since biggest possible number from index0 to index$(k-2)$ in A is $(k-1)$, then we can know $(k-1) \leq v \leq (n-1)$ if k exists in A

then, we have

$$t_n(A,v) = \begin{cases} n-v & \text{if } (k-1) \leq v \leq (n-1) \\ n & \text{if } v = n \end{cases}$$

$$P(A,v) = \begin{cases} \frac{2\times3\times4\times\cdots\times(v+1)\times1\times(v+2)\times\cdots\times n}{2\times3\times4\times\cdots\times(v+1)\times(v+2)\times\cdots\times n\times(n+1)} = \frac{1}{n+1} & \text{if } (k-1) \leq v \leq (n-1) \\ \frac{2\times3\times4\times\cdots\times k\times k\times(k+1)\times\cdots\times n}{2\times3\times4\times\cdots\times k\times(k+1)\times\cdots\times n\times(n+1)} = \frac{k}{n+1} & \text{if } v = n \end{cases}$$

Therefore, $E(t_n) = \sum\limits_{v=0}^{n} t_n(A,v) \times P(A,v) = \sum\limits_{v=0}^{n-1} \frac{1}{n+1} \times (n-v) + n \times \frac{k}{n+1} = \frac{n^2+3n+k^2-3k+2}{2(n+1)}$

Q3. SOLUTION

Pseudo code :

```
def heapFind(A, k):
    buildMaxHeap(A)
    for _ in range(k):
        max = extractMax(A)
    return max
```

Description :

At first, A is reconstructed into a heap list. After that, we extract the max number and reconstruct A for k times, and max in the k^{th} time is the result.

Analyze the time complexity :

Let n be the size of A.

By the lecture, we know that buildMaxHeap(A) $\in \Theta(n)$.

In addition, there are k times of iterations of extractMax(A), and extractMax(A) $\in \Theta(\log n)$.

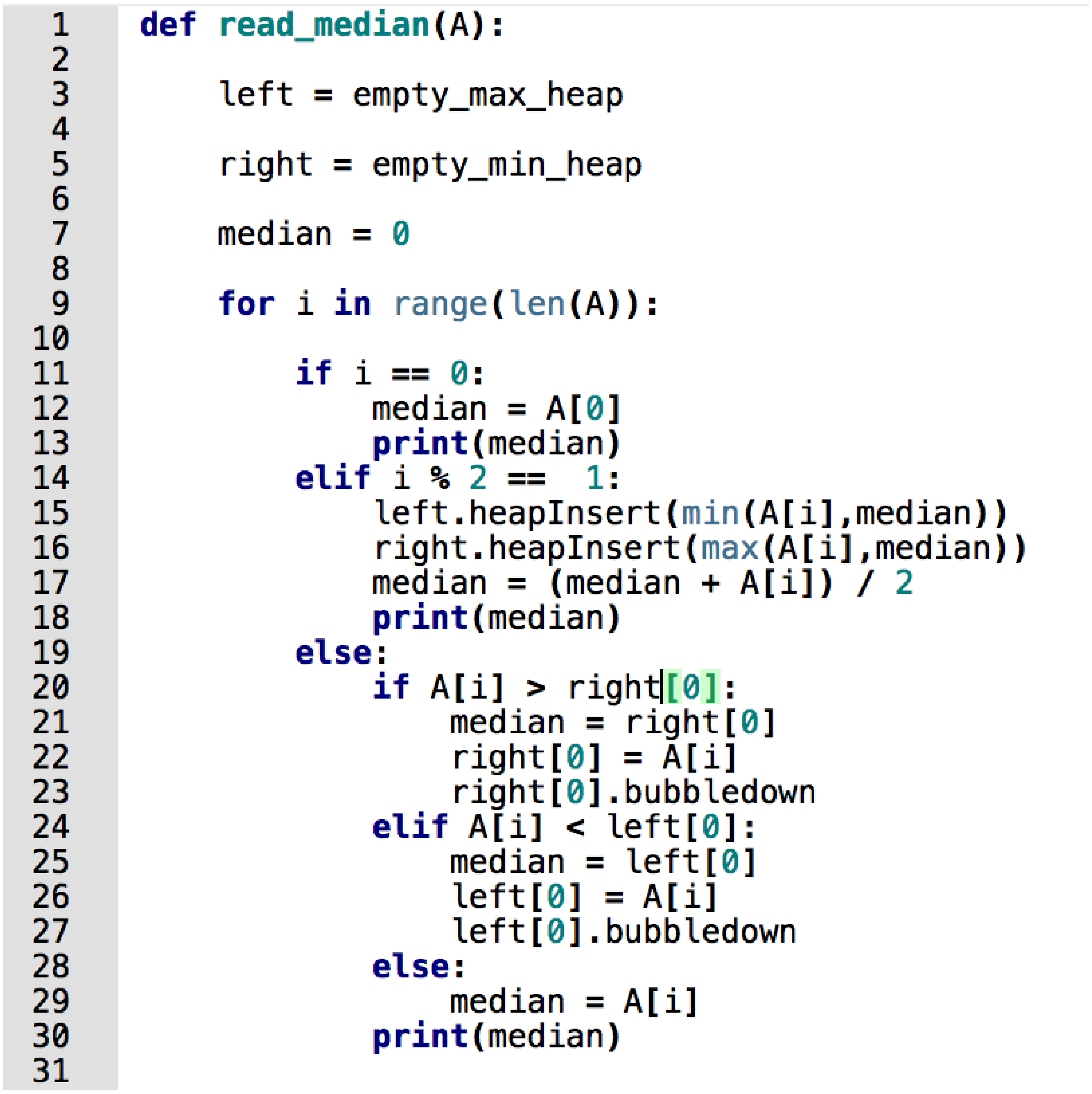
Thus, the loop costs $\Theta(k \log n)$.

Therefore, on the whole, the time $\in \Theta(n + k \log n)$.

Q4. SOLUTION

a)

Pseudo code:



```
def read_median(A):

    left = empty_max_heap

    right = empty_min_heap

    median = 0

    for i in range(len(A)):

        if i == 0:
            median = A[0]
            print(median)
        elif i % 2 ==  1:
            left.heapInsert(min(A[i],median))
            right.heapInsert(max(A[i],median))
            median = (median + A[i]) / 2
            print(median)
        else:
            if A[i] > right[0]:
                median = right[0]
                right[0] = A[i]
                right[0].bubbledown
            elif A[i] < left[0]:
                median = left[0]
                left[0] = A[i]
                left[0].bubbledown
            else:
                median = A[i]
            print(median)
```

Description:
Generally, we use three containers(a max-heap, a min-heap,and a median variable) to store the elements we have read and the elements we are going to read. Further, the max-heap is designed to store the elements which are smaller than the corresponding median and the min-heap is designed to store the elements which are larger than the corresponding median and median is for the corresponding median. With the help of the three containers, when we read a new element, what we need to do is to adjust the structure of max-heap, min-heap, and median value, basing on the values of previous max-heap root, previous min-heap root, previous median and new element.

For more detail, firstly we initialize our three data containers (Line 3 to Line 13). After, there are 2 cases to consider: the number of elements we have read is odd and the number of elements we have read is even.

For the first case, we need to insert the previous median and new element to right heap by their value relationship and corresponding heap-insert method and calculate new median value (Line14 to Line18).

For the second case, according to the value of new element, root value of max-heap
and root value of min-heap, we set the new median and adjust corresponding heap structure by corresponding bubble-down operation (Line 19 to Line 30.)

b)
For the worst case, the running time of both insert and bubble-down operation are log(n). So the total cost time is approximate to $log(n!) < log(n^n) = nlogn < n^2$. Therefore, the running time is $\theta(log(n!))$.