Please follow the instructions provided below to submit your assignment.

Worth: 10% **Due:** By 11:59 pm on Friday Sep 29

- You must submit your assignment as a single PDF file through the MarkUs system.

  https://markus.teach.cs.toronto.edu/csc263-2017-09/

  The filename must be the assignment's name followed by "sol", e.g. your submission for the assignment "A1" must be "A1sol.pdf". Any group of two students would submit a single solution through Markus. Your PDF file must contain both team member's full names.
- Make sure you read and understand "POLICY REGARDING PLAGIARISM AND ACADEMIC OFFENSE" provided in the course information sheet.
- The PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTex typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can find a latex template in Piazza under resources. You can use other typesetting systems if you prefer. Handwritten documents are acceptable but not recommended. The submitted documents with low quality that are not clearly legible, will not be marked.
- You may not include extra descriptions in your provided solution. The maximum space limit for each question is 2 pages. (using a reasonable font size and page margin)
- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.
- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.
- For describing algorithms, you may not use a specific programming language. Describe your algorithms clearly and precisely in plain English or write pseudo-code.

a

1. (20 MARKS) In the following procedure, the input is an array A of size $n \geq 2$ which contains arbitrary integers.

```
1:  procedure ODD-OR-EVEN(A, n)
2:      for i = 2 to n do
3:          x = (i mod 2)                    ▷ (i mod 2) is one if i is odd; it is zero, otherwise.
4:          if (A[i] + A[i − 1]) mod 2 ≠ x then
5:              for j = i to n do
6:                  A[j] + +
7:              end for
8:          else
9:              return                       ▷ Returns from the procedure call.
10:         end if
11:     end for
12: end procedure
```

Let $T(n)$ be the worst case time complexity of executing procedure ODD-OR-EVEN on an array of size $n \geq 2$. Assume that assignments, comparisons and arithmetic operations, like additions take a constant time each.

(a) (5 MARKS) State whether $T(n)$ is $O(n^2)$ and justify your answer.

For every input array A of size n, the outer for loop of Line 2 consists of doing at most $(n-1)$ iterations and each such iteration causes at most $(n-1)$ inner iterations of the nested for loop of line 5 ; so a total of at most $(n-1)(n-1) < n^2$ inner loop iterations are executed. Each inner loop iteration, and each one of the statements in the inner loop takes constant time.(because each consists of a constant number of comparisons and addition)

So, it is clear that there is a constant $c > 0$ such that for all $n \geq 2$: for every input A of size n, executing the procedure takes at most $c.n^2$ time.

Note that finding the upper bound in this example is almost as simple as a first look, regardless of the condition and the input, you can see that the algorithm cannot run more than $c.n^2$ steps for some constant c.Because you have two nested loops of size at most n.

(b) (15 MARKS) State whether $T(n)$ is $\Omega(n^2)$ and justify your answer. Any answer without a sound and clear justification will receive no credit.

This part is not obvious because the for loop of Line 3 may end "early" because of the return statement in Line 9. If the condition in Line 4 is not satisfied then the procedure call immediately ends. Thus, to show that $T(n)$ is $\Omega(n^2)$, we must show that there is at least one input array A such that the procedure takes time proportional to n^2 on this input, despite the loop exit condition of Line 4.

For an input with the first element being odd and the other elements being pairs of alternatively even and odd, the condition in Line 4 would be always true. For the input family A such that $A[0]$ is odd, $A[1], A[2]$ are even, $A[3], A[4]$, ... the condition of Line 4 us always true. An example of this input family is:

$A = \{1, 0, 0, 1, 1, 0, 0, ..., 0, 0, 1\}$ if $n = 4k$
$A = \{1, 0, 0, 1, 1, 0, 0, ..., 0, 0, 1, 1\}$ if $n = 4k + 1$
$A = \{1, 0, 0, 1, 1, 0, 0, \quad ... \quad , \quad 1, 1, 0\}$ if $n = 4k + 2$
$A = \{1, 0, 0, 1, 1, 0, 0, \quad ... \quad , \quad 1, 1, 0, 0\}$ if $n = 4k + 3$

the condition would always be true.
Therefore, there are cases in which the algorithm always executes the inner loop. The inner loop runs from $j = i$ to n. So, line 6 for the given input family executes $\sum_{i=2}^{n}(n-i+1) = n(n-1)/2$ times.

2. (35 MARKS) Considering the following algorithm which searches for the last appearance of value k in an array A of length n. The index of the array starts from 0.

```
FindLast(A, k):
1   for i ← n − 1, n − 2, ..., 0:
2       if A[i] = k:
3           return i
4   return −1
```

The input array A is generated in the following specific way: for $A[0]$ we pick an integer from $\{0,1\}$ uniformly at random; for $A[1]$ we pick an integer from $\{0,1,2\}$ uniformly at random; for $A[2]$ we pick an integer from $\{0,1,2,3\}$ uniformly at random, etc. That is, for $A[i]$ we pick an integer from $\{0,\ldots,i+1\}$ uniformly at random. All choices are independent from each other. Now, let's analyse the complexity of FindLast by answering the following questions. All your answers should be in **exact form**, i.e., **not** in asymptotic notations.

NOTE: For simplicity, we assume that k is an integer whose values satisfies $1 \leqslant k \leqslant n$.

(a) (5 MARKS) In the **best case**, for input (A,k), how many times is Line #2 ("**if** $A[i] = k$") executed? Justify your answer.

In the best case, $A[n-1]$, the first element we check, is k, so we find k after one comparison, i.e., Line #2 is executed only once.

(b) (5 MARKS) What is the probability that the best case occurs? Justify carefully: show your work and explain your calculation.

The probability of the best case is the probability that we choose k for $A[n-1]$, out of $n+1$ candidates ($\{0,1,\ldots,n\}$). Since it's chosen uniformly at random, this probability is $1/(n+1)$.

(c) (5 MARKS) In the **worst case**, for input (A,k), how many times is Line #2 executed? Justify your answer.

In the worst case, k does not exist in A, and the algorithm will go through the whole array, i.e., executing Line #2 for n times.

(d) (10 MARKS) What is the probability that the worst case occurs? Justify carefully: show your work and explain your calculation.

The probability of the worst case is the probability that k is not chosen by any of the entries $A[0], A[1], \ldots, A[n-1]$. For each one entry, the probability of not choosing k is different.

For entries before $A[k-2]$ (including $A[k-2]$), the probability of not choosing k is simply 1, because k is not even a candidate.

For entries after $A[k-1]$ (including $A[k-1]$), k becomes a candidate. For $A[k-1]$, the probability of not choosing k is $k/(k+1)$ (choose anything but k out of $k+1$ candidates). Similarly, for $A[k]$, the probability of not choosing k is $(k+1)/(k+2)$. In general, for $A[i]$ $(k-1 \leqslant i \leqslant n-1)$, the probability of not choosing k is $(i+1)/(i+2)$.

Since all choices are made independently, we just take the product of probabilities of k not chosen by each entry to get the probability that k is not chosen by any entry, i.e.,

$$\Pr(k \text{ not in } A) = \prod_{i=0}^{n-1} \Pr(k \text{ not chosen by } A[i]) = \prod_{i=k-1}^{n-1} \frac{i+1}{i+2} = \frac{k}{n+1}$$

However, there is a special case when $k = 1$, because the worst-case (n comparisons) happens in two possible ways. One is that k does not exist in A; the other is that k is picked by $A[0]$ but we do n comparisons anyway. The probability of the former way is $1/(n+1)$ by plugging in $k = 1$ to the above equation; the probability of the latter way is also $1/(n+1)$ because $A[0]$ chooses 0 and 1 with equal probabilities. So overall the probability of worst-case when $k = 1$ is $2/(n+1)$.

The final answer to the worst-case probability is

$$\Pr(\text{worst-case}) = \begin{cases} k/(n+1) & \text{if } 2 \leqslant k \leqslant n \\ 2/(n+1) & \text{if } k = 1 \end{cases}$$

(e) (10 MARKS) In the **average case**, for input (A, k), how many times is Line #2 expected to be executed? Justify your answer carefully: show your work and explain your calculation.

For the average case, we need to determine the probability that we need to perform t ($1 \leqslant t \leqslant n$) comparisons until finding (or not finding) k. Let t_n be a random variable that denotes the number of comparisons (Line #2) executed. The first thing to note is that only certain values are possible for t_n, which are $1, 2, \ldots, n-k+1$ and n. That is, either you find k within $n-k+1$ steps, or you never find it (after $n-k+1$ steps, k would not be a candidate any more).

Now let's figure out the probability of $t_n = 1, 2, \ldots, n-k+1$, when k is found. The probability that $t_n = t$ is the probability that k is not chosen by $A[n-1]$, $A[n-2]$, …, $A[n-t+1]$ and is chosen by $A[n-t]$, i.e.,

$$\Pr(t_n = t) = \frac{1}{n-t+2} \cdot \prod_{i=n-t+1}^{n-1} \frac{i+1}{i+2} = \frac{1}{n+1}$$

Note that, this probability is independent of t as long as we are given that k is found. Then the probability that k is not found is simply 1 minus the sum of probabilities of all k-found cases (also the result of Part (d)), which is

$$\Pr(k \text{ not found}) = 1 - (n-k+1) \cdot \frac{1}{n+1} = \frac{k}{n+1}$$

Thus the complete description of t_n's probability distribution is

$$\Pr(t_n = t) = \begin{cases} 1/(n+1) & 1 \leqslant t \leqslant n-k+1 \ \ (k \text{ found}) \\ k/(n+1) & t = n \ \ (k \text{ not found}) \\ 0 & \text{otherwise} \end{cases}$$

Note that when $k = 1$, t_n could be n for two reasons: found at the last comparison or not found.

Finally, the average running time of FINDLAST, with the given input distributions, is

$$E[t_n] = n \cdot \frac{k}{n+1} + \sum_{i=1}^{n-k+1} i \cdot \frac{1}{n+1} = \frac{2nk + (n-k+2)(n-k+1)}{2(n+1)}$$

HINT: Your answers should be in terms of both n and k. Thinking about the number of comparisons needed to find a given k, which values are possible, which values are not?

3. (20 MARKS) Given an integer array of n elements, write an algorithm that finds the k smallest elements of the array where k and n are two integers such that $k < n$. These elements should be in a sorted order. You need to give an algorithm with time complexity better than $O(n \log n)$. Analyze the time complexity of your algorithm.

   A simple approach would be to sort the elements using a sorting algorithm such as merge sort. Then return the first k elements in the sorted order. it would take $O(nlogn)$ time. For a slightly better algorithm we use a Min-heap data structure.

   We use a Min-heap data and extract the minimum from the Min-heap k times.

```
1: procedure GET-SMALLEST(A, n, k)
2:     BUILT-MAX-HEAP(A)
3:     for i = 1 to k do
4:         x = EXTRACT-MAX(A)
5:         print x
6:     end for
7: end procedure
```

   Worst-case runtime:

   - $\Theta(n)$ for creating max-heap H. $\Theta(\log n)$ for each iteration of the main loop (for ExtractMax).
   - Total is $\Theta(n + k \log n)$.

   Another approach is that we keep track of the first k elements seen so far. Let $H = A[0, \ldots, k-1]$. Initially, this is just the first k elements in A. Then, as each element of A is examined, it is compared with the largest element in H: if it is smaller, then it is one of the k smallest seen so far, so H is modified by removing its largest element and replacing it with $A[i]$ (we could make this slightly more efficient by copying $A[i]$ directly into the "root" of H and then percolating down the value). At the end, H will contain the k smallest elements in A.

```
1: procedure GET-SMALLEST(A, n, k)
2:     H ← (A[0...k − 1])
3:     BUILT-MAX-HEAP (H)
4:     for i = k to n − 1 do
5:         if A[i] < Max(H) then
6:             EXTRACT-MAX(A)
7:             INSERT(H, A[i])
8:         end if
9:     end for
10:    H ← HEAP-SORT (H)
11:    Print H
12: end procedure
```

   Worst-case runtime:

   - $\Theta(k)$ for creating max-heap H. $\Theta(\log k)$ for each iteration of the main loop (for ExtractMax and Insert. Heap sort is $O(k \log k)$

- Total is $\Theta(k + (n-k)\log k + k\log k) = \Theta(n + n\log k)$.

4. (25 MARKS) Given an input stream of n integers, we want an efficient algorithm that finds the median of elements that have been read so far. The algorithm must perform the following task for each i^{th} integer:

   - Add the integer to a running list of i^{th} integers and find the median of the updated list.
   - Print the list's updated median after reading each element.

   For example, let us consider the stream $5, 15, 1, 3$

   - After reading the first element of the stream $5 \rightarrow$ median: 5
   - After reading the 2nd element of the stream $5, 15 \rightarrow$ median: 10
   - After reading the 3rd element of the stream $5, 15, 1 \rightarrow$ median: 5
   - After reading the 4th element of the stream $5, 15, 1, 3 \rightarrow$ median: 4

   So, the algorithm output will be $5, 10, 5, 4$.

   The worst case running time of your algorithm must be less than $\Theta(n^2)$. Algorithms with time complexity of $\Theta(n^2)$ and higher will receive no credit.
   HINT: You need to use the heap data structure for an efficient algorithm.

   (a) (20 MARKS) Describe your algorithm clearly and precisely in plain English.

   The idea is to use two heap data structures to store the elements, one Max-heap and one Min-heap. Min-heap will contain the maximum half of the numbers from the array. Max-heap will contain the minimum half of the numbers from the array. So, the top value from the min-heap will be the minimum number from the maximum half of the array. The top value from the max-heap will be the maximum number from the min half of the array. Therefore, you would have the middle values of the array on top of the two heaps. We insert the element into the two heaps such that the size difference of the two heaps is always less than two and the elements in the Min Heap should be larger than elements in the Max Heap.

   1 Initialize two empty heaps. A Max-heap and a Min-heap. ($size_{max} = 0, size_{min} = 0$).
   2 For the first input element add it to the Max-heap and return it as the median. ($size_{max} = 1, size_{min} = 0$).
   3 For all the next elements ($2 \leq i \leq n$):
   3.a If the element is smaller or equal to the Max-heap root value insert the element to the Max-heap ($size_{max}$++), otherwise insert the element to the Min-heap ($size_{min}$++).
   3.b Balance the heaps (we want to make sure that the difference size of the two heaps is not more than one.) If the number of elements in one of the heaps is greater than the other by more than 1, extract the root element from the one containing more elements and insert it to the other one.
   3.c If i is even, the median would be the average of the Max-heap root and the Min-heap root, else the median is the root of the larger heap.

(b) (5 MARKS) Analyze the worst case running time of your algorithm.

Step 1 and 2 execute in a constant time. Step 3.a executes a Max-heap or a Min-heap insert and step 3.b might executes an Extract-max or an Extract-min. The time complexity of both insert and extract is $\Theta(\log n)$. Finally, 3.c is in a constant time. Since 3.a, 3.b, 3.c execute $n-1$ times, the time complexity of the algorithm will be $\Theta(n \log n)$.

REMARK: The *median* is the value separating the higher half of a data set from the lower half. For a sorted data set,

- if a dataset contains an odd number of elements, the median is the middle element of the sorted sample.
- if the dataset contains an even number of elements, the median is the average of the two middle elements of the sorted sample.