Please follow the instructions provided below to submit your assignment.

Worth: 10% **Due:** By 11:59 pm on Friday Nov 17

- 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 "A3" must be "A3sol.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.

1. **Amoritization**

In this question, we explore a well-known trick that uses two Stacks S_1 and S_2 to simulate the operations of a Queue Q. Consider the following implementations of ENQUEUE and DEQUEUE.

```
ENQUEUE(Q, x):
    if SIZE(S_1) ⩾ 12 and ISEMPTY(S_2):
        while not ISEMPTY(S_1):
            PUSH(S_2, POP(S_1))
    PUSH(S_1, x)

DEQUEUE(Q):
    if ISEMPTY(S_2):
        if ISEMPTY(S_1):
            error "Dequeuing from an empty queue!"
        else:
            while not ISEMPTY(S_1):
                PUSH(S_2, POP(S_1))
    return POP(S_2)
```

For your analysis, assume that each PUSH operation takes 2 units of time, and each POP operation takes 3 units of time. Each ISEMPTY or SIZE operation takes 0 unit of time. The time taken by any other pseudo-code operation is ignored.

(a) (10 Marks) Consider the sequence of operations that consists of 50 ENQUEUE operations followed by 50 DEQUEUE operations. Use **aggregate analysis** to compute the amortized cost per operation for this sequence.

The operations can be categorized into the following types:

- "Short" ENQUEUE: A simple PUSH which costs \$2.
- "Long" ENQUEUE: A PUSH following the migration of 12 elements from S_1 to S_2, whose cost is $5 \times 12 + 2 = \$62$.
- "Short" DEQUEUE: A simple POP which costs \$3.
- "Long" DEQUEUE: A POP following the migration of all elements from S_1 to S_2, whose cost is $5k + 3$ (with k being the number of elements to migrate).

The sequence of 50 ENQUEUE and 50 DEQUEUE operations then looks as follows:

- 12 short ENQUEUES,
- 1 long ENQUEUE,
- 37 short ENQUEUES,
- 12 short DEQUEUES,
- 1 long DEQUEUE with the migration of 38 elements,
- 37 short DEQUEUES.

The total cost is:

$$12 \cdot \$2 + 1 \cdot \$62 + 37 \cdot \$2 + 12 \cdot \$3 + 1 \cdot (\$5 \cdot 38 + 3) + 37 \cdot \$3 = \$500.$$

Therefore the amortized cost per operation is:

$$\$500/100 = \$5.$$

(b) (10 Marks) Now consider **any** sequence of m ENQUEUE and DEQUEUE operations. Use the **accounting method** to derive an upper-bound on the amortized cost per operation.

Using the accounting method, we charge \$10 for each ENQUEUE operation and \$0 for each DEQUEUE operation. Imagine that this \$10 is "attached" to the element that was ENQUEUED: \$2 pays for pushing the element into S_1, \$5 for migrating the element from S_1 to S_2 (1 PUSH + 1 POP), and \$3 for popping the element out of S_2. This \$10 per element is enough to pay for all operations since each element is pushed, migrated and popped at most once during any sequence of operations. Hence, the amortized cost per operation for any sequence is *at most* 10 units of time (maybe less depending on the sequence).

2. **Fuzzy sorting of intervals**
Solve Exercise 7-6 from the textbook (CLRS) page 189.

(a) (10 Marks) The algorithm is similar to the randomized quick-sort. We need to sort the a_i's, but we replace the comparison operators to check for overlapping intervals. The algorithm consider three partitions:

- intervals that are entirely disjoint from the pivot but less than pivot.
- intervals that are entirely disjoint from the pivot but greater than pivot.
- intervals that are not only overlap by the pivot but also overlap with each other.

For such a partitioning, any time we encounter an interval that overlaps with the pivot, we replace the pivot with their intersection. For a given element x in position i, we'll use $x.left$ and $x.right$ to denote a_i and b_i respectively.

```
FUZZY-PARTITION(A, p, r)
    q ← choose a uniformly random value ∈ {p, .., r}
    x ← A[q]
    exchange A[q], A[r]
    i ← p − 1
    k ← p − 1
    for j ← p to r − 1 do
        if b_j < x.left then
            i ← i + 1
            k ← k + 1
            exchange A[i], A[j]
            exchange A[k], A[j]
        else if a_j ≤ x.right then
            x.left ← max(a_j, x.left)
            x.right ← min(b_j, x.right)
            k ← k + 1
            exchange A[k], A[j]
    exchange A[k + 1], A[r]
    return i, k + 1

FUZZY-QUICKSORT(A, p, r)
    if p < r
        small, equal = FUZZY-PARTITION(A, p, r)
        FUZZY-QUICKSORT(A, p, small)
        FUZZY-QUICKSORT(A, equal+1, r)
```

(b) (6 Marks)For non-overlapping intervals the algorithm would run the same as the regular quicksort. Therefore, the expected runtime is $\Theta(n \log n)$ in general. Note that it is the expected worst case running time, so the worst case would be when the intervals are disjoint. We showed that the expected running time of the quicksort is $O(n \log n)$. Similarly we can show that it is $\Omega(n \log n)$. From the quick sort running time in the lec06 page 24:

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \geq \sum_{i=1}^{n-1} 2\ln(n-i+1)$$

$$= 2\ln(\prod_{i=1}^{n-1}(n-i+1)) = 2\ln(n!) = \frac{2}{\log e} \log(n!) \geq cn \log n$$

It shows that $E(X) = \Omega(n \log n)$ and therefore, $E(X) = \Theta(n \log n)$ If all of the intervals overlap, then the algorithm returns p and r.FUZZY-PARTITION will only be called a single time, so, the total expected runtime is $\Theta(n)$.

3. **Randomized Algorithm - Search in an unsorted array**

The following algorithm uses a randomized strategy to search for a value x in an unsorted array A consisting of n elements.
The strategy: Choose a random index i into a given array A. If $A[i] = x$, then we return i; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

(a) (7 Marks) Write a pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

```
RANDOMIZED SEARCH(A, n, x)
    Initialize count = 0 and i = 0
    Initialize a new array Flag with n elements of 0
    while count ≠ n
        Uniformly choose i ∈ {1, ..., n}
        if (A[i] == x)
            return i
        if (Flag[i] == 0)
            Flag[i] = 1
            count = count + 1
    return "A does not contain x"
```

(b) (4 Marks) Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?
Let X be the random variable for the number of searches required. Then

$$E(X) = \sum_{i \geq 1} i.Pr(X = i) = \sum_{i \geq 1} i(\frac{n-1}{n})^{i-1}(\frac{1}{n}) = \frac{1}{n}.\frac{1}{(1-\frac{n-1}{n})^2} = n.$$

Recall that: $\sum_{i\geq 1} ix^{i-1} = \frac{1}{(1-x)^2}$

(c) (4 Marks) Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

Let X be the random variable for the number of searches required. Then

$$E(X) = \sum_{i\geq 1} i.Pr(X = i) = \sum_{i\geq 1} i(\frac{n-k}{n})^{i-1}(\frac{k}{n}) = \frac{k}{n}.\frac{1}{(1-\frac{n-k}{n})^2} = n/k.$$

(d) (10 Marks) Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Let X be the random variable for the number of searches required. For $0 < i \leq n$, state i contains all the random chosen indices which are chosen after $i-1$ indices untill we choose a new index (the i^{th} index). Let x_i be the number of random indices in state i. For the first state, $x_1 = 1$ since the first random index that is chosen is a new index. x_2 would be the number of times that the same index is chosen again plus one. In step i, $i-1$ elements of the array $Flag$ is 1 and $n-i+1$ elements are 0. For each random chosen index in the i^{th} state, the probability of choosing a new index is $\frac{n-i+1}{n}$.

$$E(x_j) = \sum_{i\geq 1} i.Pr(x_j = j) = \sum_{i\geq 1} i(\frac{j-1}{n})^{i-1}(\frac{n-j+1}{n}) = \frac{n}{n-j+1}$$

Since, $X = x_1 + x_2 + ...x_n$.

$$E(X) = \sum_{i=1}^{n} E(x_j) = \sum_{j=1}^{n} \frac{n}{n-j+1} = n\sum_{j=1}^{n} \frac{1}{j} = n(\ln n + c)$$

where c is a constant.

4. **Randomized Algorithm - Search in an almost sorted array**

An array of n distinct elements with starting index zero is *almost sorted* if the elements are sorted but the smallest index is not necessarily the first index of the array. We are given a reference *head* to the index with smallest element. Clearly, for $head > 0$ the largest element is $A[head-1]$ and if $head = 0$ the largest element is $A[n-1]$. An example of an almost sorted array would be:

$head = 3$

0	1	2	3	4	5	6	7
33	40	55	5	12	20	25	30

For a given almost sorted array A and its head, consider the following algorithm for performing SEARCH(x):

SEARCH$(A, head, x)$
  Uniformly choose $i \in \{0, \ldots, n-1\}$.
  Compare $A[i]$ and x: (The number of comparisons in each step is one, not three)

If $A[i] = x$ **return** i
If $A[i] > x$ $i \leftarrow head$
If $A[i] < x$ $i \leftarrow ((i+1) \mod n)$
While $True$ **do**
  Compare $A[i]$ and x
    If $A[i] = x$ **return** i
    If $A[i] > x$ **return** -1
    If $A[i] < x$ $i \leftarrow ((i+1) \mod n)$
    If $i = head$ **return** -1

(a) (4 Marks) What is the worst case number of comparisons with x performed by SEARCH(x)? Briefly justify your answer.

The worst case number of comparisons cannot exceed $n+1$. In the algorithm, there is one comparison before the loop and n comparisons in the loop. The worst case happens when x is larger than all the elements and the random generated variable is $head$.

(b) (10 Marks) Determine the expected worst case number of comparisons with x performed by SEARCH(x):

- Define the sample space.
- Define the random variables for your analysis.
- Compute the expected worst case number of comparisons with x performed by SEARCH(x) and then, write your solution in terms of big-O and Ω.

The expected number of comparisons is computed over the random choices that algorithm makes. So, the sample space is $S = \{0, 1, 2, ..., n-1\}$ and the random variable i is uniformly chosen from S. Let t_i be the worst case number of comparisons for any $i \in S$ chosen in the algorithm. The distribution is uniformly at random, therefore, $P(t_i) = 1/n$.

$$E(t_i) = \sum_{i=0}^{n-1} t_i P(t_i) = \frac{1}{n} \sum_{i=0}^{n-1} t_i$$

Without loss of generality, we can assume that $head = 0$. (Otherwise, we need to shift everything but the number of comparisons are the same). Now, we compute t_i which is the worst case number of compassions for each value of i.
For $0 \le i < \lfloor n/2 \rfloor$, the worst case is when $x > A[n-1]$. In this case, x is compared with elements $A[i]$ to $A[n-1]$. So, $t_i = n - i$.
For $\lfloor n/2 \rfloor \le i < n$, the worst case is when $A[i-1] < x < A[i]$. In this case, x is compared with elements $A[i]$ and then $A[0]$ to $A[i]$. So, $t_i = i + 2$.

$$t_i = \begin{cases} n - i & 0 \le i < \lfloor n/2 \rfloor \\ i + 2 & \lfloor n/2 \rfloor \le i < n \end{cases}$$

- If n is even:

$$\sum_{i=0}^{n-1} t_i = n + (n-1) + ... + (n - n/2 + 1) + (n/2 + 2) + ...n + 1 =$$

$$\frac{n}{2}\left(\frac{n + n - n/2 + 1}{2}\right) + \frac{n}{2}\left(\frac{n/2 + 2 + n + 1}{2}\right) = \frac{3}{4}n^2 + n$$

– If n is odd:

$$\sum_{i=0}^{n-1} t_i = n + (n-1) + ... + (n-(n-1)/2+1) + ((n-1)/2+2) + ...n + 1 =$$

$$\frac{n-1}{2}\left(\frac{n+n-(n-1)/2+1}{2}\right) + \frac{n+1}{2}\left(\frac{(n-1)/2+2+n+1}{2}\right) = \frac{3}{4}n^2 + n + \frac{1}{4}$$

Therefore,

$$E(t_j) = \frac{1}{n}\sum_{i=1}^{n} t_i \begin{cases} \frac{3}{4}n+1 & n \text{ is even} \\ \frac{3}{4}n+1+\frac{1}{4n} & n \text{ is odd} \end{cases}$$

The upperbound is $O(n)$ and the lower bound is $\Omega(n)$.

5. In this question, we want to see how many keystrokes are required to type a text message? You may think that it is equal to the number of characters in the text, but this is correct only if each keystroke generates only one character. With pocketsize devices, the possibilities for typing text are often limited. Some devices provide only a few buttons, significantly fewer than the number of letters in the alphabet. For such devices, several strokes may be needed to type a single character.

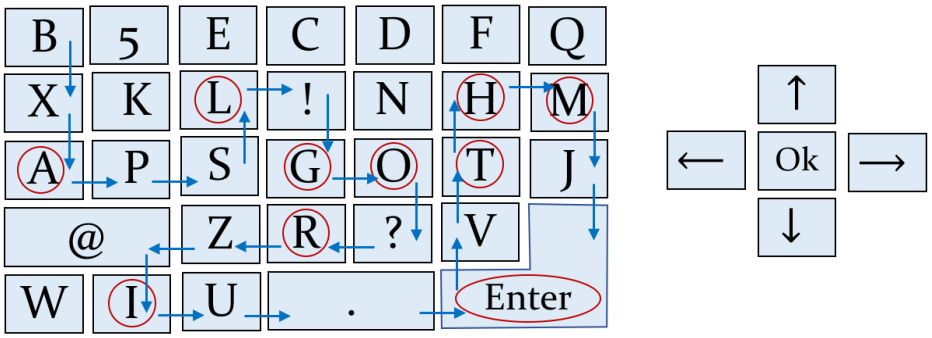
   One mechanism to deal with these limitations is a virtual keyboard displayed on a screen, with a cursor that can be moved from key to key to select characters. Four arrow buttons control the movement of the cursor, and when the cursor is positioned over an appropriate key, pressing the fifth button selects the corresponding character and appends it to the end of the text. To terminate the text, the user must navigate to and select the Enter key. This provides users with an arbitrary set of characters and enables them to type text of any length with only five hardware buttons.

   In this problem, you are given a virtual keyboard layout in a two dimensional array A with r rows and c columns. Your task is to determine the minimal number of strokes needed to type a given text, where pressing any of the five hardware buttons constitutes a stroke. Note that you stroke OK key in the middle to type the letter which **should be counted**. The keys are arranged in a rectangular grid, such that each virtual key occupies one or more connected unit squares of the grid. The cursor starts in the **upper left corner of the keyboard** and moves in the four cardinal directions, in such a way that it always skips to the next unit square in that direction that belongs to a different key. If there is no such unit square, the cursor does not move.

   Figure 1, shows an example for a virtual keyboard. For such an input, we are given an array of size 7×5. It shows a possible way to type ALGORITHM using 32 strokes. Note that there is only one key corresponding to any given character. Each key is made up of one or more grid squares, which will always form a connected region. The character key '\n' represents Enter in the virtual keyboard. The input is A and s where A is an array of size $c \times r$ as a virtual keyboard and s is a string of size n as an input text to be typed.

$$A = \begin{bmatrix} B & 5 & E & C & D & F & Q \\ X & K & L & ! & N & H & M \\ A & P & S & G & O & T & J \\ @ & @ & Z & R & ? & V & \backslash n \\ W & I & U & . & . & \backslash n & \backslash n \end{bmatrix}$$

Figure 1: Sample Input. An example for virtual keyboard and hardware buttons. It shows a possible way to type ALGORITHM using 32 strokes on an example virtual keyboard.



(a) (20 Marks) Write a pseudocode for an algorithm that finds the minimal number of strokes necessary to type the whole text, including the Enter key at the end of typing. It is guaranteed that the text can be typed.

We model the question to a problem of finding a shortest path in a directed graph using a BFS.

1) **Finding the positions of each character in the text in the virtual keyboard:**
For a given text s, we find the position of each letter (pairs of (i, j)) by a simple search and store them in an array. Note that each element of the array can be a list of pairs, since the position of a character can be more than one. Also, we assume that the string ends with $\backslash n$. Since for each character of s we search in the array, the time complexity is $\Theta(ncr)$
We can also use a direct access table using an ASCII codes to make the search faster. (i.e $\Theta(cr)$)

2) **Constructing a graph:**
To construct the graph, first consider **a simpler version of the problem**: Assume that we need to type a single character x instead of a string. The representative graph would be a directed graph $G^0 = (V^0, E^0)$ such that for each element $A[i, j]$ in the given 2D array we consider one node $v^0_{i,j}$. So, the graph has rc nodes. Each node has maximum of 4 edges leaving that node. For each node, we need to compute the right, left, top and bottom neighbours. One approach would be considering four arrays RN, LN, TN, BN to store these neighbours.

RIGHT-NEIGHBOURS(A)
Initialize an $c \times r$ array RN to store the right neighbours
for $i = c - 1$ to 0
  for $j = 0$ to $r - 1$
    if $i + 1 \geq c$ $RN[i, j] = NIL$
    else if $A[i + 1, j] \neq A[i, j]$ $RN[i, j] = (i + 1, j)$
    else $RN[i, j] = RN[i + 1, j]$

Similarly, we can compute the left, top and bottom neighbours, so we can construct the graph G_1. Having this graph, the minimum number of required strokes would be the shortest path from $v^0_{0,0}$ to any desired vertex corresponding to the position of the letter in the virtual keyboard which can be computed by a single $BFS(G, v^0_{0,0})$.
Back to the original problem, since we have a text with n characters, we consider the graph $G' = G^0 \cup G^1 ... \cup G^{n-1}$ where for $0 \leq k < n$, $G^k = (V^k, E^k)$ is constructed the same as G^0. Then, we construct the graph G from G' by adding the following edges:

For $0 \leq k < n$ and the graph G^k, a directed edge from any vertices in G^k that correspond to the position of k^{th} character in s to all vertices in G^{k+1} that correspond to the position of k^{th} character in s.

3) **Finding the shortest path using a BFS($G^0, v^0_{0,0}$)** The problem now is finding the shortest path from $v^0_{0,0}$ to all $v^{n-1}_{i,j}$ where (i, j)s are all pairs of the positions of $\backslash n$ in the virtual keyboard. The result would be the minimum distance computed by BFS to any of those pairs plus the length of s including $\backslash n$.

(b) (5 Marks) Analyze the running time of your algorithm.

The running time of finding the position of the letters is $\Theta(rc)$ or $\Theta(rcn)$ depends on the data structure we use. Constructing the graph is $\theta(rcn)$ since the number of nodes is rcn and the number of edges is $O(rcn)$. The complexity of BFS algorithm on a graph with V nodes and E edges is $\Theta(V + E)$. So, The complexity of Running BFS in our problem is $\Theta(rcn)$. One might still get a full mark if the time complexity is not $\Theta(rcn)$ but the approach is good enough.