Please follow the instructions provided on the course website to submit your assignment.
Worth: 8% **Due:** By 11:59 pm on Tuesday July 18

Randomized/Probabilistic Algorithms

1. Suppose a sorted list of length n is represented using an integer $head \in \{1,\dots,n\}$ and an $n \times 2$ matrix A. This data structure is equivalent to a linked list, where each row of A is a node, containing a value and a pointer. Specifically,

   - $A[i,1]$ stores the value of a node in the list,
   - $A[i,2]$ stores the index of the row representing the next node in the list or 0, if it is the last node in the list, and
   - $head$ is the index of the row containing the first node in the list.

   For example, the list 11, 22, 66, 88, 99 could be represented as follows:

   $head$ $\boxed{3}$

A	1	2
1	66	2
2	88	5
3	11	4
4	22	1
5	99	0

   Consider the following algorithm for performing $Search(x)$:

```
Uniformly choose $i \in \{1,\dots,n\}$.
Compare $A[i,1]$ and $x$:
    If $A[i,1] = x$ then return $i$
    If $A[i,1] > x$ then $i \leftarrow head$
    If $A[i,1] < x$ then $i \leftarrow A[i,2]$
While $i \neq 0$ do
    Compare $A[i,1]$ and $x$
        If $A[i,1] = x$ then return $i$
        If $A[i,1] > x$ then return 0
        If $A[i,1] < x$ then $i \leftarrow A[i,2]$
return 0
```

   (a) 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 i is the index of the largest element (in the example, the index of the largest element is 5), and x is between the second largest and largest value (In the example $88 < x < 99$). In this case, since $A[i,1] > x$, $head$ will be assigned to i and x will be compared to all the n elements of A. So, the number of comparisons in the worst case is $n+1$.

   (b) Determine the worst case expected number of comparisons with x performed by $Search(x)$:

   - Define the probability space.

- Define the random variables for your analysis.
- Derive an upper bound on the worst case expected number of comparisons with x performed by $Search(x)$.
- Derive a lower bound on the worst case expected number of comparisons with x performed by $Search(x)$.

The expected number of comparisons is computed over the random choices that algorithm makes. So, the sample space is $S = \{1, 2, ..., n\}$ 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, $P(t_i) = 1/n$.

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

Now, we compute t_i which is the worst case number of compassions for each value of i. Let $a_1 = head$ and $a_j = A[a_{j-1}, 2]$ for $2 \le j \le n$. Therefore,

$$A[a_1, 1] \le A[a_2, 1] \le ... \le A[a_n, 1]$$

It is easy to see that $i \in \{a_1, a_2, ...a_n\}$. Let $i = a_j$

For $1 \le j \le \lfloor n/2 \rfloor$, the worst case is when $x > A[a_{n-1}, 1]$. In this case, x is compared with elements $A[a_j, 1]$ to $A[a_n, 1]$. So, $t_j = n - j + 1$.

For $\lfloor n/2 \rfloor < j \le n$, the worst case is when $A[a_{j-1}, 1] < x < A[a_j, 1]$. In this case, x is compared with elements $A[a_j, 1]$ and then $A[a_1, 1]$ to $A[a_j, 1]$. So, $t_j = j + 1$.

$$t_j = \begin{cases} n - j + 1 & 1 \le j \le \lfloor n/2 \rfloor \\ j + 1 & \lfloor n/2 \rfloor < j \le n \end{cases}$$

– If n is even:

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

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

– If n is odd:

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

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

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 $\Omega(n)$.

Amortization, Dynamic Arrays

2. Show that if a DECREMENT operation were included in the k-bit counter example, (The example given in Amortization lecture), n operations could cost as much as $\Theta(nk)$ time.

Suppose the input is a 1 followed by $k-1$ zeros. If we call DECREMENT we must change k entries. If we then call INCREMENT on this it reverses these k changes. Thus, by calling them alternately n times, the total time is $\Theta(nk)$.

3. Consider the following idea to implement the Dynamic Array ADT: initially, the array has size 10 and is empty; when the array gets full and an Append operation is performed, allocate a new array with room for 10 more elements, copy all the existing elements over, then store the new element. Show that the amortized complexity over any sequence of m Append and Delete operations is not constant for this data structure. (Hint: If the amortized complexity were constant, what would that mean for the total charge compared to the total cost, for any sequence of m operations and for any $m > 0$? It's fine to use some specific constant(s) to figure out the answer, but your final solution must show that no constant will work.)

We compute the worst-case sequence complexity directly. Because the ADT supports only growth when the array becomes over-full, but not shrinking when the array becomes over-empty, we consider a sequence with only APPEND operations. This is likely to be one of the worst sequences possible—though we don't need to prove this formally: as long as the sequence we choose is bad enough (more than amortized constant time), we can conclude that the worst-case sequence complexity is at least as time-consuming as our sequence.

So, consider a sequence of m APPEND operations, starting from an array that is initially empty (but with room for 10 elements). The total running time for the sequence consists of the time taken to store each element in the array plus the time taken for each array growth. By the end of the m operations, the array will store exactly m elements so it will have grown exactly $\lfloor m/10 \rfloor$ times. The first growth requires copying 10 elements, the second growth requires copying 20 elements, and so on. So the total time taken to copy elements during the entire sequence is equal to:

$$\sum_{i=1}^{\lfloor m/10 \rfloor} 10i = 10\frac{\lfloor m/10 \rfloor(\lfloor m/10 \rfloor + 1)}{2} = 5\lfloor m/10 \rfloor^2 + 5\lfloor m/10 \rfloor.$$

Adding the time to store each element (exactly m array accesses), this shows that the worst-case sequence complexity is at least:

$$\begin{aligned} m + 5\lfloor m/10 \rfloor^2 + 5\lfloor m/10 \rfloor &\geqslant m + 5\left(\frac{m}{10} - 1\right)^2 + 5\left(\frac{m}{10} - 1\right) \\ &= m + 5\left(\frac{m^2}{100} - \frac{2m}{10} + 1\right) + \frac{m}{2} - 5 \\ &= m + \frac{m^2}{20} - m + 5 + \frac{m}{2} - 5 = \frac{m^2 + 10m}{20} = \frac{m(m+10)}{20}. \end{aligned}$$

Finally, for every constant $C \in \mathbb{R}^+$, if we perform $m > 20C - 10$ operations, the total running time will be:

$$\frac{m(m+10)}{20} > \frac{m(20C - 10 + 10)}{20} = Cm.$$

Hence, the worst-case sequence complexity is *not* constant.

Breadth-First Search

4. Modify the Breadth-First Search algorithm given in the textbook so that it takes as input the adjacency matrix for an **undirected** graph $G = (V, E)$, and it returns TRUE iff G contains a simple **undirected** cycle, FALSE otherwise.

   Write the full modified algorithm, in pseudocode, and make it as simple as possible—in particular, your modified algorithm does **not** need to compute distances $v.d$ or BFS edges $v.\pi$ for every vertex $v \in V$. However, be careful to write your algorithm explicitly in terms of the *adjacency matrix* for the input graph G. In other words, your algorithm must explicitly access the appropriate entries in the adjacency matrix whenever it carries out edge queries.

   Finally, briefly argue that your algorithm is correct and analyse its worst-case running time.

Algorithm:

```
CYCLIC($G$):   # let $G.A$ denote the adjacency matrix, with dimensions $[n \times n]$
    for $i \leftarrow 1, \dots, n$:
        $\pi[i] \leftarrow 0$   # use in place of colour
    initialize an empty queue $Q$
    for $i \leftarrow 1, \dots, n$:   # ensure every vertex is examined
        if $\pi[i] = 0$:   # vertex $i$ has not been "discovered" yet
            $\pi[i] \leftarrow i$   # start BFS at vertex $i$
            ENQUEUE($Q, i$)
            while $Q$ is not empty:
                $j \leftarrow$ DEQUEUE($Q$)
                for $k \leftarrow 1, \dots, n$:
                    if $G.A[j,k] = 1$:   # $G$ contains an edge $\{j,k\}$
                        if $\pi[k] = 0$:   # $k$ is "undiscovered"
                            $\pi[k] \leftarrow j$
                            ENQUEUE($Q, k$)
                        else if $k \neq \pi[j]$:   # $k$ is not the parent of $j$
                            return TRUE
    return FALSE   # entire graph explored and no cycle found
```

[Correctness:] If the algorithm returns TRUE, then it has found an edge $\{j,k\}$ from a vertex j to a vertex k such that k is not the parent of j in the BFS tree. Let i be the first common ancestor of j and k in the BFS tree—it may be that $i = k$. Then G contains a cycle i—j–k—i.

Conversely, if G contains a simple cycle, then let i be the first vertex in a simple cycle to be dequeued by the algorithm and let m be the length of this cycle. Other vertices in the cycle will be examined one by one (possibly along with other vertices not in the cycle), in order of their distance from i. When the algorithm dequeues the first (when i is odd) or last (when i is even) vertex in the cycle at distance $\lfloor (m-1)/2 \rfloor$ from i, it will find an edge from this vertex to its neighbour (not its parent) in the cycle and return TRUE.

Runtime: Each vertex i is enqueued at most once (because it is only enqueued when $\pi[i] = 0$ and in that case, $\pi[i]$ is set to a positive value just before enqueuing i). But the outer **for** loop ensures that each vertex is enqueued at least once—so each vertex is enqueued exactly once. This means each row of the adjacency matrix is examined exactly once. So the total running time is $\Theta(n^2)$ in the worst-case. (Note that, as a function of the input size, this is *linear* time—since the input size is exactly n^2.)