Midterm 01

Instructions. This exam is open book and open note—you may freely use your notes, lecture notes, or textbook while working on it. You may *not* consult any living resources such as other students or web forums. The exam must be submitted by the beginning of class on Thursday, *September 30th, 2021*. If you do not attend class in person, you may email your scanned or typeset solution **in PDF format** to the professor using the subject line `[COSC 211] Midterm 01`.

Affirmation. I attest that that work presented here is mine and mine alone. I have not consulted any disallowed resources while taking this exam.

Name: Dhyey Dharmendrakumar Mavani

Signature: ____________

Problem 1. (Big O Notation)

(a) Complete the following table by placing an "X" in a cell if the function in the cell's row is O of the function in the cell's column. You may assume that all primitive computer operations are performed in time $O(1)$.

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
$1,000,000,000$	X				
$0.001n^2 + 400n$					X
$4\sqrt{n} + 30 \log n$			X	X	X
$4n^2 + 3n^{5/2}$					X
time to search a linked list of length n for a given element			X	X	X
time to perform binary search on a sorted array of length n		X	X	X	X

(b) Suppose the running time of some method `foo` has worst-case running time $T_1(n) = O(\log n)$ on inputs of size n, while another method `bar` has worst-case running time $T_2(n) = O(n)$ and $T_2(n) \neq O(\log n)$ (i.e., T_2 is *not* $O(\log n)$). What can we say about the relative *empirical* running times of `foo` and `bar`? Is `foo` guaranteed to run faster than `bar` on all inputs?

foo $\rightarrow$ empirical running time for $T_1(n)$ should be $\leq O(\log n)$

bar $\rightarrow$ empirical running time for $T_2(n)$ should be $\leq O(n)$

Moreover, as per question, $T_2(n) \neq O(\log n) \Rightarrow T_2(n)$ will be definitely greater than $O(\log n)$. because $T_2(n)$ is also $\leq O(n)$.

$\Rightarrow$ Relative empirical runtime of bar is greater than foo.

Still, we can't say that foo is guaranteed to run faster than bar for all inputs, because the actual (empirical) time is random and depends on a large number of factors which we can't estimate. There is an error and randomness associated with it, which messes things sometimes as we have already encountered in lectures'/assignments' run-time analysis parts especially for low n.

Problem 2. Recall that the SimpleList<E> interface specifies the following methods (among others):

- E get(i) return the element at position i in the list
- void add(i, y) insert the element y to position i in the list
- void remove(i) remove and return the element at position i in the list

We would like to implement a SimpleSet<E> using a SimpleList<E> to store the contents of the set. For example, we might have

```
class MySet<E> implements SimpleSet<E> {
    SimpleList<E> contents = new SomeList<E>();
    ...
    boolean add(E y) { ... }
    ...
}
```

where SomeList<E> implements SimpleList<E>.

(a) How could you implement the add(E y) method for MySet<E>, which should add the element y to the set if y is not already present?

Relevant Pseudocode chunk for add(E y)

```
for (i=0; i < size; i++){
    if (y.equals(get(i))){
        return false;          ] -> primary operation
    }
}
add(size, y);
size++;                        ] -> primary operations
return true;
```

We first check if the element y in "boolean add(E y)" is not already in the MySet<E>. For that we use the if statement "y.equals(get(i))", and return false if true. If the element to be added (y) is unique to the MySet<E>, then add the element at end of set using the given method "add(size, y)", increment size, and return true.

(b) Suppose that for SomeList, the methods get(i), add(i, y), and remove(i) have worst-case running times $O(1)$, $O(n)$, and $O(n)$, respectively, where n is the size of the SomeList. What is the worst-case running time of the add(E y) implementation you described in part (a)?

GIVEN: $\left\{\begin{array}{l} get(i) \to O(1) \\ add(i,y) \to O(n) \\ remove(i) \to O(n) \end{array}\right\}$

Worst case add(E y)'s runtime is bigO.

primary operations

bigO for add(E y) = bigO for for-loop + bigO for add(i,y) + O(1)

= bigO for for-loop + O(n) + O(1) = bigO for for-loop + O(n)

= n(bigO for if-statement) + O(n) [because looping n times]

= n(bigO for get(i) + O(1)) + O(n)

= n(O(1) + O(1)) + O(n) = n(O(1)) + O(n)

= O(n) + O(n) = $\boxed{O(n)}$ Ans

worst case running time for add(E y) as in part (a) & (b)

(c) Suppose you use a different SimpleList implementation where the worst-case running times of get(i), add(i, y), and remove(i) are all $O(\log n)$. What would be the new running time of the add(E y) implementation you described in part (a)?

GIVEN: $\left\{\begin{array}{l} get(i) \\ add(i,y) \\ remove(i) \end{array}\right\} \to O(\log n)$

Worst case add(E, y)'s runtime is bigO.

primary operation

bigO for add(E y) = bigO for for-loop + bigO for add(i,y) + O(1)

= bigO for for-loop + O(log n) + O(1) = (bigO for for-loop + O(log n))

= n(bigO for if-statement) + O(log n)

= n(bigO for get(i) + O(1)) + O(log n)

= n(O(log n) + O(1)) + O(log n)

= n(O(log n)) + O(log n)

= O(n log n) + O(log n) = $\boxed{O(n \log n)}$ Ans

worst case running time for add(E, y) as in part (a) & (c)

Problem 3. In class, we discussed an array-based `SimpleStack<E>` implementation with the following `push(E x)` method:

```
public class ArraySimpleStack<E> implements SimpleStack<E> {
  private int size = 0;
  private Object[] contents;

  public void push(E x) {
    if (size == capacity) {
      increaseCapacity();
    }
    contents[size] = x;
    ++size;
  }

  private void increaseCapacity() {
    Object[] bigContents = new Object[2 * capacity];
    for (int i = 0; i < capacity; ++i) {
      bigContents[i] = contents[i];
    }
    contents = bigContents;
    capacity = 2 * capacity;
  }
}
```

We showed that when defined as above, the `push(E x)` method has *amortized* running time $O(1)$. Consider the following variant of the `pop()` method, which ensures that the capacity of `contents` is never more than twice the size of the stack:

```
public E pop() {
  if (size == 0) {
    throw new EmptyStackException();
  }
  if (size <= capacity / 2) {
    decreaseCapacity()
  }
  --size;
  return (E) contents[size];
}

private void decreaseCapacity() {
  Object[] smallContents = new Object[capacity / 2];
  for (int i = 0; i < size; ++i) {
    smallContents[i] = contents[i];
  }
  contents = smallContents;
  capacity = capacity / 2;
}
```

(a) What is the amortized running time of the pop() method defined on the previous page?

→ suppose last resize occurs at capacity N.

→ next resize will happen when size hits N/2.

→ Cost of my next resize $C_{N/2}$ is $O(N/2)$

⇒ each pop() operation (before resize)

→ pay cost of pop (without resize) $= O(1)$

→ deposit $= \frac{1}{(N/2)} C_{N/2} = \frac{O(N/2)}{(N/2)} = O(1)$

$\boxed{\text{Amortized Run time: } O(1)}$

⇒ before next resize (size = N/2) ⇒ must do $\frac{N}{2}$ pop(s)s

→ There will be $\frac{N}{2} O(1) = O(\frac{N}{2}) = \boxed{O(N) = C_{N/2}}$

⇒ for resize

→ Pay $C_{N/2}$ out of my bank account

AMORTIZED COST = After − before + $C_{N/2}$ = $\boxed{0}$ ✓

(b) In the original ArraySimpleStack implementation (in which pop() does not resize the array), we showed that push(E x) as amortized running time $O(1)$. With the variant of pop() defined on the previous page, is the amortized running time of push(E x) still $O(1)$?

→ I think it will be same because we are not calling them when we are defining the methods.

→ But, when we call push and pop back to back when capacity = size * 2, we can see that the effects of increase Capacity() and decrease Capacity() cancels each other out leading to no amortization. This leads to a runtime of O(n) rather than O(1).

Problem 4. Consider a variant of a `SimpleSSet`, called `MultiSSet`, which can store multiple copies of the same element. Thus, the state of a `MultiSSet` could be, for example,

$$S = \{\{1, 2, 2, 3, 3, 3, 4, 5, 5\}\}.$$

Suppose we wish to implement a `MultiSSet` in which the elements of the set are stored in an array `Object[] contents` in ascending order. That is, if element x_i is stored in `contents[i]`, then we have

$$x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_{n-1}.$$

In addition to the `SimpleSSet` operations, a `MultiSSet` has a method `int findMultiplicity(E x)` that returns the number of occurrences of `x` in the `MultiSSet`. For example, with S as above, `findMultiplicity(5)` should return 2 since 5 occurs twice in S.
In the space below, describe how you could implement `int findMultiplicity(E x)` with a worst-case running time of $O(\log n)$, where n is the number of (not necessarily distinct) elements store in the `MultiSSet`.

This approach implements two Binary Searches Concurrently using the Divide and Conquer Paradigm, and eventually leads to a run time of O(logn)

```
int findMultiplicity (E x){
    int i=headBinarySearch (0, n-1, x)
    if (i==-1){
        return 0;
    }
    int j= tailBinarySearch (i, n-1, x)
    return (j-i+1);
}
int headBinarySearch(int lo, int hi, E x) {
    if (hi >= lo)
    {
        int mid = lo + (hi-lo)/2;
        if ((mid==0||(int)x> Object[mid-1]) && (Object[mid]==(int)x))
            return mid;
        elseif((int)x > Object[mid])
            return headBinarySearch ((mid+1), hi, x);
        else
            return headBinarySearch (lo, (mid-1), x);
    }
    return -1;
}
int tailBinarySearch (int lo, int hi, E x){
    if (hi >= lo)
    {
        int mid = lo + (hi-lo)/2;
        if ((mid == n-1||(int)x < Object[mid+1]) && (Object[mid] == (int) x))
            return mid;
        elseif((int)x < Object[mid])
            return tailBinarySearch (lo, (mid-1), x);
        else return tailBinarySearch ((mid+1), hi, x);
    }
    return -1;
}
```