CS61B Spring 2015 Guerrilla Section 2 Worksheet SOLUTIONS

Akhil Batra, Leo Colobong, Nick Fong, Jasmine Giang, Laura Harker, Anusha Ramakuri, Charles Zhang, Jason Zhang, Giulio Zhou

14 March 2015

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is*. Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 Asymptotic Analysis

Given the following code snippet, give a bound in Big-O for the runtime with respect to the length of the `String` input.

```
1 public void mystery(String input) {
2     int n = input.length();
3     for (int i = 0; i < n; i++) {
4         for (int j = 0; j < n; j++) {
5             i = i*2;
6             j = j*2;
7         }
8     }
9 }
```

$O(\log(n))$: both i and j are doubled in the inner loop, so it takes $\log(n)$ time for both i and j to be incremented to n

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

2 Asymptotic Proofs

Given the following block of code, answer the following questions. Assume that $n \geq 1$.

```
for (int i = 1; i <= n; i++){
    for (int j = 1; j <= i; j = j*2){
        System.out.println(i+j);
    }
}
```

(a) Prove that the code runs in approximately $O(\log(n!))$ time.
Hint: $\log(a \cdot b) = \log(a) + \log(b)$ and $n! = n(n-1)(n-2)\dots(2)(1)$
The inner loop runs $\log(i)$ times for each i since we condition on $2^j < i$. Thus our total run time is $\log(n) + \log(n-1) + \dots + 2 + 1 = \log((n)(n-1)\dots(2)(1)) = \log(n!)$.

(b) We will now prove that $log(n!) = \Theta(n\log(n))$ in two steps:

(i) Prove that $\log(n!) = O(n\log(n))$
Hint: $\log(a^b) = b\log(a)$
Upper bound each term of the factorial by n:

$$\begin{aligned} \log(n!) &= \log((n)(n-1)(n-2)\dots(2)(1)) \\ &<= \log(n \cdot n \cdot n \cdot \dots \cdot n) \text{ where we have } n \text{ terms of } n \\ &= \log(n^n) \\ &= n\log(n) \end{aligned} \tag{1}$$

(ii) Prove that $\log(n!) = \Omega(n\log(n))$
We throw out the smaller half of the factorial, and lower bound the remaining terms by $\frac{n}{2}$. We know that the terms we threw out are ≥ 1, so throwing them out will only make our product smaller.

$$\begin{aligned} \log(n!) &>= \log((n)(n-1)(n-2)\dots(\frac{n}{2})) \\ &>= \log((\frac{n}{2})(\frac{n}{2})\dots(\frac{n}{2})) \\ &= \log((\frac{n}{2})^{(\frac{n}{2})}) \\ &= (\frac{n}{2})\log(\frac{n}{2}) \end{aligned} \tag{2}$$

So $\log(n!) >= (\frac{n}{2})\log(\frac{n}{2}) = \Theta(n\log(n))$, which gives us that $\log(n!) = \Theta(n\log(n))$.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

3 Count Down

(a) Give bounds in Big-Theta (Θ) notation for the runtimes of the following methods with respect to the arguments passed in to the functions.

```
1  public int countDown() {
2      for(int i = 100; i >= 0; i--) {
3          System.out.println(i);
4      }
5  }
6
7  public int countDown(int length) {
8      for(int i = length; i>= 0; i--) {
9          System.out.println(i);
10     }
11 }
12
13 public int launchRockets(int numRockets) {
14     for(int i = 0; i < numRockets; i++) {
15         countDown();
16     }
17     for(int i = 0; i < 100; i++) {
18         countDown(i);
19     }
20 }
```

`countDown()` is in $\Theta(1)$, since it does not depend on the size of any input.
`countDown(int length)` is in $\Theta(n)$
`launchRockets` is in $\Theta(n)$ since it does `numRockets` constant-time calls. The second for loop does not affect running time because it is constant

(b) What if we changed the second for loop (on line 17) in `launchRockets(int numrockets)` to:

```
17     for(int i = 0; i < numRockets; i++) {
```

Would the running time of `launchRocket(int numrockets)` change, and if so, how?
Yes, it would change since the second for loop is not constant. The running time would become $\Theta(n^2)$. The first loop has n iterations with a constant number of operations in each iteration, which takes n time. The second loop also has n iterations, and calls `countDown(int length)` once per iteration. This gives us $n + (n-1) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$ which gives $\Theta(n^2)$.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

4 Data Structures (Stacks, Queues, Maps)

Using inheritance, define a class `TrackedQueue` that behaves like Queue except for an extra method, `maxSizeSoFar()` which returns an int corresponding to the maximum number of elements in this queue since it was constructed. Assume that the `Queue` class has the following methods:
`void enqueue(Object obj)`
`int size()`

```
1  public class TrackedQueue extends Queue {
2      private int maxSize;
3      public TrackedQueue(){
4          super();
5          maxSize = 0;
6      }
7      public int maxSizeSoFar(){
8          return maxSize;
9      }
10     public int enqueue(Object obj){
11         super.enqueue(obj);
12         if (super.size() > maxSize){
13             maxSize = super.size();
14         }
15     }
16 }
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 Queue Implementation

Consider the following implementations for a queue:

(a) A `java.util.ArrayList` with the front of the queue at the end of the list (the $(n-1)$th element in a queue of n elements) and the back of the queue at the start of the array (element 0). Assume that there is room in the array to enqueue an element.

(b) A singly linked list with an additional reference to the last node in the list (the tail), with the front of the queue last in the list and the back of the queue at the head of the list.

For each implementation, give estimates for the number of operations necessary to enqueue an element and to dequeue an element given that the queue has n elements. You can answer either with Big-O notation or by saying that the enqueue/dequeue takes "time proportional to X" for some X. Provide a brief explanation for your answers.

Implementation (a)

1. `enqueue` $O(n)$: We have to shift every element over by one in order to make room at index 0 for the new element to enqueue.

2. `dequeue` $O(1)$: We return the element at index $n-1$ and set that location to `null`.

Implementation (b)

1. `enqueue` $O(1)$: We make a new node to add to the beginning of our list. We make this new node point to the head, and change the head pointer to point to this new node.

2. `dequeue` $O(n)$: We return the element in the tail node. We also have to iterate through the entire list starting at the head until we read the node right before tail, set its tail to `null`, and update the tail pointer to point to this node.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 ExpandableSet

Using inheritance, define a class `ExpandableSet` that behaves like `Set` (see below) except that when `insert` is called with a value to be inserted larger than the `Set` can currently hold, the `Set` doubles in size until the value can be added. `ExpandableSet` should have a no argument constructor that makes the initialize size of the `Set` to be 1.

```
/**
 * Represent a set of nonnegative ints from 0 to maxElement-1
 * for some initially specified maxElement.
 * contains[k] is true if k is in this set, false otherwise.
 */
public class Set {
    protected boolean[] contains;
    public Set(int maxElement) { //Initialize a set of ints from 0 to maxElement-1
        contains = new boolean[maxElement];
    }
    public void insert(int k) {
        contains[k] = true;
    }
    public void remove(int k) {
        contains[k] = false;
    }
    public boolean member(int k) {
        return contains[k];
    }
}
```

Solution:

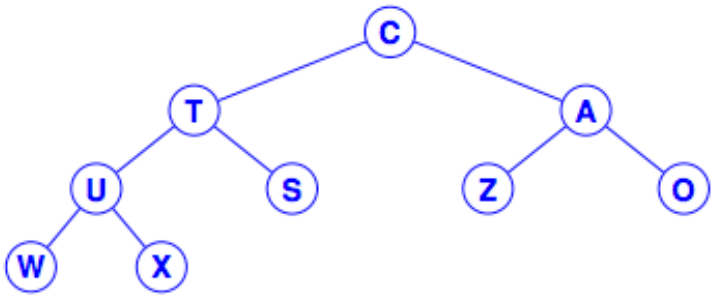
```
public class ExpandableSet extends Set {
    public ExpandableSet() {
        super(1);
    }

    public void insert(int k) {
        if (k > contains.length) {
            int origSize = contains.length;
            int newSize = origSize * 2;
            while (newSize <= k) {
                newSize = newSize * 2;
            }

            boolean[] biggerVersion = new boolean[newSize];
            for (int i = 0; i < origSize; i++) {
                biggerVersion[i] = contains[i];
            }
            contains = biggerVersion;
        }
        super.insert(k);
    }
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

7 Trees (Extra for Experts)

(a) Draw a full binary tree that has a preorder of $C, T, U, W, X, S, A, Z, O$ and a postorder of $W, X, U, S, T, Z, O, A, C$. Each node should contain exactly one letter.



(b) What is the inorder of this tree?
$W, U, X, T, S, C, Z, A, O$