CSE 373 Summer 2017 Homework 2: Asymptotic Analysis

Due Thursday, July 6th at 5:00pm

Name: ______Ran Ju________________________

Students you worked with (but did not share solutions with!):

Student ID number: _________1621899______________________

1) Big-O (3 points)

For each of the following, show that $f \in O(g)$ by finding values for *c* and n_0 such that the definition of big-O holds true, as we did with examples in lecture. Listing valid values for *c* and n_0 is sufficient (no further proof necessary).

a) $f(n) = 47n^3 + 2050n$ $\qquad$ $g(n) = n^5$

choose c = 2050, n_0= 2, for all n>=n_0, $f(n) \le cg(n)$.

b) $f(n) = 12(4 + log(n))$ $\qquad$ $g(n) = 3n$

choose c = 1, n_0= 22, for all n>=n_0, $f(n) \le cg(n)$.

c) $f(n) = 18n$ $\qquad$ $g(n) = \frac{n}{42}$

choose c = 756, n_0= 1, for all n>=n_0, $f(n) \le cg(n)$.

2) Runtime Analysis (6 points)

For each of the following program fragments, determine the asymptotic runtime in terms of n. Explain or show your work (an explanation of your intuition can be sufficient).

a)

```
public void mysteryOne(int n) {
    int y = 1;
    for (int j = 0; j < n*(n-4); j++) {
        for (int i = 1; i < n; i *= 2) {
            y *= y;
        }
    }
}
```

Runtime is $O(n^3)$

Because in the first for loop, the runtime is $O(n^2)$, in the inner loop, the runtime is $O(\log(n))$.
Therefore, the runtime of this programming is $O(n^2 \log(n))$.

b)

```
public void mysteryTwo(int n) {
    int x = 0;
    for (int i = n/2; i < n; i++) {
        if (i % 5 == 0) {
            break;
        } else {
            for (int j = 1; j < n; j += 2) {
                x++;
                x *= 2;
            }
        }
    }
}
```

The outer loop runtime is O(n), the inner loop runtime is O(n)
Therefore, the runtime of this programming is O(n^2)

c)

```
public void mysteryThree(int n) {
    for (int i = n; i >= 0; i--) {
        helper(i);
    }
}

private void helper(int x) {
    if (x > 0) {
        helper(x – 3);
    }
}
```

The for loop runtime is O(n) and the helper function runtime is O(n). therefore the runtime of programming is O(n^2)

3) More Asymptotic Analysis (3 points)

For each of the following, determine if $f \in O(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$, $f \in o(g)$, $f \in \omega(g)$, **several of these (and which), or none of these. No proofs necessary, listing your answer is sufficient.**

a) $f(n) = 24n^4 - n$ $\qquad g(n) = 3n^2 + 8 + n^4$

$f \in O(g)$

$f \in o(g)$

b) $f(n) = 10^n$ $\qquad g(n) = n^{10}$

$f \in \Omega(g)$

$f \in \omega(g)$

c) $f(n) = log(log(n))$ $\qquad g(n) = log(n)$

$f \in O(g)$

$f \in o(g)$

4) Pseudocode and Recurrence Relations (9 points)

a) Write pseudocode for a function that calculates the largest difference between any two numbers in an array of positive integers with a runtime in Θ(n^2).

For example, the largest difference between any two numbers in the following array would be 90.

a = [37, 8, 92, 4, 6, 20, 11, 2, 40]

```
int max = Integer.MIN_VALUE;
for ( int i = 0; i < a.length-1; i++ ){
    for ( int j = i+1; j < a.length; j++){
        update max value if the difference between i and j is the larger one.
    }
}
return max;
```

b) Can this function be written with a runtime in Θ(n)? If yes, write pseudocode below. If no, why? What would have to be different about the input in order to do so?

Yes

```
int max = a[0];
int min = a[0];

for( int i = 0; i < a.length; i++){
    update max value if a[i] is the larger one;
    update min value if a[i] is the smaller one;
}
return max – min;
```

don’t need any difference about the input;

c) Can this function be written with a runtime in Θ(1)?. If yes, write pseudocode below. If no, why? What would have to be different about the input in order to do so?

Yes

```
int max = a[a.length – 1] – a[0];
return max;
```

the array must be a sorted array, like the largest one is in the end and the smallest one is in the first.(the largest one is in the first and the smallest one is in the first is ok but the code is int max = a[0] -a[a.length-1]);

5) Recursion and Recurrence Relations (5 points)

Note: For this set of problems, let *c* and *d* be constants and let the base case be *T(c) = d.*

a) What is the recurrence relation for the following method? (The base case is given)

```
public void recursiveMethod(int n) {
	if (n == 1) {
		return;
	}
	int sum = 0;
	for (int i = 0; i < n; i++) {
		sum += i;
	}
	if (sum % 2 = 0) {
		System.out.println("The sum is even");
		return recursiveMethod(n/2);
	} else {
		System.out.println("The sum is odd");
		return recursiveMethod(n/2);
	}
}
```

Base case: T(1) = 1
Recurrence Relation: T(n) = n + 2 + T(n/2)

b) Find the *tightest* Big-Oh bound for your recurrence relation from part (a) using the base case *T(c) = b.* Justify your answer.
(An example of the *tightest* Big-O bound: for f(n) = 5n, the *tightest* bound is O(n), not O(n^2))

"Expand" the original relation to find an equivalent general expression in terms of the number of expansions.
T(n) = n + 2 + T(n/2)
T(n) = n + 2 + (n/2 + 2 + T(n/4))
T(n) = 2k + k*(1+1/2+1/4…+1/2^k) + T(n/(2^k))

Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case.
n/(2^k) = 1 which means k = $\log_2 n$.
k*(1+1/2+1/4…+1/2^k) is a sum of geometric series, which is equal to 2. Therefore, the sum is 2k.

So T(n) = O(n)

6) Growth Rates (3 points)

Order the following functions from slowest to fastest in terms of asymptotic runtime. Be sure to show whether two functions have the same asymptotic runtime.

Ex: $5n < 3n^2 = n^2$

- n^{72}
- $n^2 \log(n)$
- $2^{(n/2)}$
- $\log(n)$
- $n \log(n^2)$
- n^6
- $n \log(\log(n))$
- $n \log^2(n)$
- n
- n^2
- $n \log(n)$
- 2^n
- $\log^{2(}n)$
- $2/n$
- $\log(\log(n)$
- $2^{(1/2)}$

$2^{(n/2)} = 2^n > n^{72} > n^6 > n^2 \log(n) > n^2 > n \log(n^2) > n \log^2(n) > n \log(n) > n \log(\log(n)) > n > \log^{2(}n) > \log(n) > \log(\log(n)) > 2^{(1/2)} > 2/n$

7) Proof by Induction (8 points)

Our friend the 3-Eyed Alien doesn't believe that $2^n > n^2$ as n approaches infinity. "See," they say, "for $n = 3$ the opposite is true!" Use induction to prove to our friend that after a certain point, $2^n > n^2$ is indeed true.

Hint: It may be useful to use the fact that $(n-1)^2 > 2$ for $n \geq$ the base case, along with some clever addition and subtraction after multiplying out the polynomial.

Be sure to clearly mark each step, and where you use the inductive hypothesis.

Base case: n = 5

$2^5 > 5^2$,right

Inductive Hypothesis:

Assume true for n = k where k is an arbitrary integer>5.

Inductive Step:

It is true that $(k-1)^2 > 2, when\ k\ is\ larger\ than\ 5. therefore\ k^2 > 2k+1$

$$2^{k+1} = 2^k * 2 > 2 * k^2 = k^2 + k^2 > k^2 + 2k + 1 > (k+1)^2$$

Now we proved for k+1 case.

QED

8) Amortized Analysis (5 points)

Note: For all parts of this problem, when we ask for Big-O we're asking for the *tightest* Big-O bound. (For example, the *tightest* Big-O bound for f(n) = 5n is O(n), not O(n^2))

a) Imagine an array-based implementation of a stack that, instead of doubling its array size, it increases its array size by the fixed amount of 1000 every time it's full. For this stack, what is the Big-O amortized cost (i.e. average asymptotic running time) to `push()` n times? Justify your answer.

When push() 1000i times, it should copy array 1000i(i-1)/2 times, so the n times' cost is (n-1)*n/2, the worst case is O(n^2).
The average time is O(n).

b) Based on their amortized costs for `push()`, which version of an array-stack would you use: one that doubles its array size or one that increases its array size by a fixed amount (i.e. 1000) whenever it's full, and why?

In the method of double its array size, the amortized cost is O(1)
Cost = n + 2/n + 4/n+ 8/n…..=2n=O(n)
So the amortized cost is O(1);
The amortized cost of other version is O(n), so based on this, I would like to use doubles its array size because it is faster than other version.

c) Imagine an array-based implementation of a stack that doubles its array size when the array is full, and halves the array size when it's ¾ empty. What is the Big-O amortized cost of `pop()` if it's the n^{th} operation done on an initially empty stack? Justify your answer.

The worst-case running time of pop(): O(n)
Cost = 1/4 n+1/16 n+…+$1/4^i$ n = 1/2 n = O(n);
The average running time of pop(): O(1)