Section 02: Asymptotic Analysis

Section Problems

1. Comparing growth rates

(a) Simplify each of the following functions to a tight big-$\mathcal{O}$ bound in terms of n. Then order them from fastest to slowest in terms of asymptotic growth. (By "fastest", we mean which function increases the most rapidly as n increases.)

- $\log_4(n) + \log_2(n)$
- $\frac{n}{2} + 4$
- $2^n + 3$
- $750,000,000$
- $8n + 4n^2$

(b) Order each of these more esoteric functions from fastest to slowest in terms of asymptotic growth. (By "fastest", we mean which function increases the most rapidly as n increases.) Also state a simplified tight $\mathcal{O}$ bound for each.

- $2^{n/2}$
- 3^n
- 2^n

2. True or false?

(a) In the worst case, finding an element in a sorted array using binary search is $\mathcal{O}(n)$.

(b) In the worst case, finding an element in a sorted array using binary search is $\Omega(n)$.

(c) If a function is in $\Omega(n)$, then it could also be in $\mathcal{O}\left(n^2\right)$.

(d) If a function is in $\Theta(n)$, then it could also be in $\mathcal{O}\left(n^2\right)$.

(e) If a function is in $\Omega(n)$, then it is always in $\mathcal{O}(n)$.

3. Modeling code

For each of the following code blocks, give a summation that represents the worst-case runtime in terms of n.

(a)

```
int x = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        x++;
    }
}
```

(b)

```
int x = 0;
for (int i = n; i >= 1; i /= 2) {
    x += i;
}
```

4. Finding bounds

For each of the following code blocks, construct a mathematical function modeling the worst-case runtime of the code in terms of n. Then, give a tight big-$\mathcal{O}$ bound of your model.

(a)

```
int x = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n * n / 3; j++) {
        x += j;
    }
}
```

(b)

```
int x = 0;
for (int i = n; i >= 0; i -= 1) {
    if (i % 3 == 0) {
        break;
    } else {
        x += n;
    }
}
```

(c)

```
int x = 0;
for (int i = 0; i < n; i++) {
    if (i % 5 == 0) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                x += i * j;
            }
        }
    }
}
```

(d)

```
int x = 0;
for (int i = 0; i < n; i++) {
    if (n < 100000) {
        for (int j = 0; j < n; j++) {
            x += 1;
        }
    } else {
        x += 1;
    }
}
```

(e)

```
int x = 0;
if (n % 2 == 0) {
    for (int i = 0; i < n * n * n * n; i++) {
        x++;
    }
} else {
    for (int i = 0; i < n * n * n; i++) {
        x++;
    }
}
```

5. Applying definitions

For each of the following, choose a c and n_0 which show $f(n) \in \mathcal{O}(g(n))$. Explain why your values of c and n_0 work.

(a) $f(n) = 3n + 4, g(n) = 5n^2$

(b) $f(n) = 33n^3 + \sqrt{n} - 6, g(n) = 17n^4$

(c) $f(n) = 17\log(n), g(n) = 32n + 2n\log(n)$

6. Using our definitions

Most of the time in the real world, we don't write formal big-$\mathcal{O}$ proofs. The point of having these definitions is not to use them every single time we think about big-$\mathcal{O}$. Instead, we use the formal definitions when a question is particularly tricky, or we want to make a very general statement.

Here are some particularly tricky or general statements that are easier to justify with the formal definitions than with just your intuition.

(a) We almost never say a function is $\mathcal{O}(5n)$, we always say it is $\mathcal{O}(n)$ instead. Show that this transformation is ok, i.e. that if $f(n)$ is $\mathcal{O}(5n)$ then it is $\mathcal{O}(n)$ as well.

(b) When we decide on the big-$\mathcal{O}$ running time of a function, we like to say that whatever happens on small n doesn't matter. Let's see why with an actual proof. You write two functions to solve the same problem: `method1` and `method2`. `method1` takes $\mathcal{O}(n^2)$ time and `method2` takes $\mathcal{O}(n)$ time. What is the big-$\mathcal{O}$ running time of the following function:

```
public void combined(n){
  if(n < 10000)
  method1(n);
  else
  method2(n);
}
```

(c) Consider this code for telling whether an integer n is prime:

```
public boolean isPrime(int n){
        for(int i = 2; i < n; i++){
                if(n % i == 0)
                        return false;
        }
        return true;
}
```

The running time of `isPrime` is $\mathcal{O}(n)$, but is it also $\Omega(n)$? Hint: these definitions will be useful: $f(n)$ is $\Omega(g(n))$ if there exist positive c, n_0 such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

$f(n)$ is not $\Omega(g(n))$ if for all positive c, n_0 there exists and $n \geq n_0$ such that $f(n) < c \cdot g(n)$.

7. Memory analysis

For each of the following functions, construct a mathematical function modeling the amount of memory used by the algorithm in terms of n. Then, give a Θ bound of your model.

(a)

```
List<Integer> list = new LinkedList<Integer>();
for (int i = 0; i < n * n; i++) {
    list.insert(i);
}
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

(b)

```
int[] arr = {0, 0, 0};
for (int i = 0; i < n; i++) {
    arr[0]++;
}
```

(c)

```
ArrayDictionary<Integer, String> dict = new ArrayDictionary<>();
for (int i = 0; i < n; i++) {
    String curr = ””;
    for (int j = 0; j < i; j++) {
        for (int k = 0; k < j; k++) {
            curr += ”?”;
        }
    }
    dict.put(i, curr);
}
```

Note: for simplicity, assume the dictionary has an internal capacity of exactly n.

Food for thought

8. LRU Caching

When writing programs, it turns out to be the case that opening and loading data in files can be a very slow process. If we plan on reading information from those files very frequently (for example, if we want to implement a database), what we might want to do is *cache* the data we loaded from the files – that is, keep that information in-memory.

That way, if the user requests information already present in our cache, we can return it directly without needing to open and read the file again.

However, computers have a much smaller amount of RAM then they have hard drive space. This means that our cache can realistically contain only a certain amount of data. Often, once we run out of space in our cache, we get rid of the items we used the *least recent*. We call these caches **Least-Recently-Used (LRU)** caches.

Discuss how you might apply or adapt the ADTs and data structures you know so far to develop an LRU cache. Your data type should store the most recently used data, and handle the logic of whether it can find the data in the cache, or if it needs to read it from the disk. Assume you have a helper function that handles fetching the data from disk.

Your cache should implement our `IDictionary` interface and optimize its operations with the LRU caching strategy. After you've decided on a solution, describe the tradeoffs of your structure, possibly including a worst-case and average-case analysis.