Section 02: Solutions

Section Problems

1. Comparing growth rates

(a) Order each of the following functions from fastest to slowest in terms of asymptotic growth. (By "fastest", we mean which function increases the most rapidly as n increases.)

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

Solution:

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

(b) For each of the above expressions, state the simplified tight $\mathcal{O}$ bound in terms of n.

Solution:

- $\mathcal{O}(\log(n))$
- $\mathcal{O}(n)$
- $\mathcal{O}(2^n)$
- $\mathcal{O}(1)$
- $\mathcal{O}(n^2)$

(c) 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

Solution:

- 3^n, which is in $\mathcal{O}(3^n)$
- 2^n, which is in $\mathcal{O}(2^n)$
- $2^{n/2}$, which is in $\mathcal{O}\left(\sqrt{2}^n\right)$ (or $\mathcal{O}(2^{n/2})$).

Constant multipliers don't matter in big-O notation, but a constant factor in the exponent **does** matter, since it corresponds to multiplying by some constant to the n^{th} power. Saying $2^{n/2}$ is in $\mathcal{O}(2^n)$ would be true, but it would not be a tight bound.

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 $\mathcal{O}(n^2)$.

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

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

Solution:

(a) True

(b) False

(c) True

(d) True

(e) False

As a reminder, we can think about $\mathcal{O}$ informally as an upper bound. If a function $f(n)$ is in $\mathcal{O}(g(n))$, then $g(n)$ is a function that *dominates* $f(n)$, and this domination can be really overshooting the mark. Every (correct) piece of code we write in this class will have a running time that is $\mathcal{O}\left(n!^{n!^{n!}}\right)$. Conversely, we can think about Ω informally as a lower bound. If a function $f(n)$ is in $\Omega(g(n))$, then $f(n)$ is a function that *dominates* $g(n)$, and this domination can be really overshooting the mark also. The running time of any piece of code is always in $\Omega(1)$. And finally, Θ is a much stricter definition. $f(n)$ is in $\Theta(g(n))$ (if and only if) $f(n)$ is in $\mathcal{O}(g(n))$ and in $\Omega(g(n))$. Usually when people say $\mathcal{O}$, they mean Θ.

For questions a and b: note that binary search takes $\log(n)$ time to complete. $\log(n)$ is upper-bounded by n, so $\log(n) \in \mathcal{O}(n)$. However, $\log(n)$ is not lower-bounded by n, which means $\log(n) \in \Omega(n)$ is false.

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++;
    }
}
```

Solution:

One possible solution is

$$T(n) = 1 + \sum_{i=0}^{n-1} \sum_{k=0}^{i-1} 1$$

(b)

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

Solution:

One possible solution is

$$T(n) = 1 + \sum_{i=1}^{\log(n)} 1$$

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;
    }
}
```

Solution:

One possible answer is $T(n) = \frac{n^3}{3}$. The inner loop performs approximately $\frac{n^2}{3}$ iterations; the outer loop repeats that n times, and each inner iteration does a constant amount of work.

So the tight worst-case runtime is $\mathcal{O}\left(n^3\right)$.

The exact constant you get doesn't matter here, since we'll ignore the constant when we put it into $\mathcal{O}$ notation anyway. For example, saying we do 3 operations per inner-loop iteration (checking the loop condition, updating `j`, and updating `x`) and getting n^3 instead of $n^3/3$ is also completely reasonable.

(b)

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

Solution:

The tightest possible big-$\mathcal{O}$ bound is $\mathcal{O}(1)$ because exactly one of n, $n-1$, or $n-2$ will be divisible by three for all possible values of n. So, the loop runs at most 3 times.

(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;
            }
        }
    }
}
```

Solution:

While the inner-most if statement executes only once per loop, we must check if `i == j` is true once per each iteration. This will take some non-zero constant amount of time, so the inner-most loop will perform approximately n work (setting the constant factors equal to 1, is conventional, since constant factors can depend on things like system architecture, what else the computer is doing, the temperature of the room, etc.).

The outer-most loop and if statement will perform n work during only 1/5th of the iterations and will perform a constant amount of work the remaining 4/5ths of the time. So, the total amount work done is approximately $\frac{n}{5} \cdot n + \frac{4n}{5} \cdot 1$. If we simplify, this means we can ultimately model the runtime as approximately $T(n) = \frac{n^2}{5} + \frac{4n}{5}$.

Therefore, the tightest worst-case asymptotic runtime will be $\mathcal{O}\left(n^2\right)$.

(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;
    }
}
```

Solution:

Recall that when computing the asymptotic complexity, we only care about the behavior for large inputs. Once n is large enough, we will only execute the second branch of the if statement, which means the runtime of the code can be modeled as just $T(n) = n$. So, the tightest worst-case runtime is $\mathcal{O}(n)$.

(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++;
    }
}
```

Solution:

We can model the runtime of this function in the **general** case as:

$$T_g(n) = \begin{cases} n^4 & \text{when } n \text{ is even} \\ n^3 & \text{when } n \text{ is odd} \end{cases}$$

However, the prompt was asking you to prove a model for the **worst** possible case – that is, when n is even. If we assume n is even, we can produce the following model:

$$T_w(n) = n^4$$

The tightest worst-case asymptotic runtime is then $\mathcal{O}\left(n^4\right)$ in this case.

Something interesting to note is that the **general** model has differing tight big-$\mathcal{O}$ and big-Ω bounds and so therefore has no big-Θ bound.

That is, the best big-$\mathcal{O}$ bound we can give for $T_g(n)$ is $T_g(n) \in \mathcal{O}\left(n^4\right)$; the best big-$\Omega$ bound we can give is $T_g(n) \in \Omega\left(n^3\right)$. These two bounds ($n^4$ and n^3) are different so there is no big-Θ for T_g. Importantly however, there is a big-Θ for our simpler model, T_w. That is, $T_w(n) \in \Theta\left(n^4\right)$.

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.

Solution:

These solutions are divided into "scratch work" which is algebra you have to do before you start writing the proof and the "proof" itself. The scratch work technically doesn't belong in a final answer, but the proofs are difficult to understand without them.

For these, the proof will just be the scratch work algebra, possibly done in a different order, with some connecting words.

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

Solution:

scratch work: Our goal is to bound f by a function with n^2 terms so comparing to g is easier.

$$3n \leq 3n^2 = \frac{3}{5} \cdot 5n^2 \qquad \text{if } n \geq 1$$
$$4 \leq 4n^2 = \frac{4}{5} \cdot 5n^2 \qquad \text{if } n \geq 1$$

We add together the inequalities to get:

$$f(n) = 3n + 4 \leq \left(\frac{3}{5} + \frac{4}{5}\right) 5n^2 = \frac{7}{5} g(n)$$

proof: One possible solution is $c = \frac{7}{5}$ and $n_0 = 1$.

We note that $3n \leq 3n^2$ and $4 \leq 4n^2$ as long as $n \geq n_0$. Adding these two inequalities, we have $f(n) = 3n + 4 \leq 7n^2 = \frac{7}{5} g(n)$ is true for all $n \geq 1$.

Therefore, we know that $3n + 4 \leq c \cdot 5n^2$ is true for our chosen values of c and for all $n \geq n_0$.

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

Solution:

scratch work: Since g's dominating term is n^4, we will try to bound f by a function with only n^4 terms. Going term by term of f:

$33n^3 \leq 33n^4$ as long as muliplying by n increases the function (i.e. as long as $n \geq 1$).
$\sqrt{n} \leq n^4$ as long as $n \geq 1$.
$-6 \leq 0n^4$ (always).

Combining these we want to get: $33n^3 + \sqrt{n} - 6 \leq 33n^4 + n^4 \leq 34n^4 \leq c \cdot 17n^4$ c being 2 is enough.

proof: One possible solution is $c = 2$ and $n_0 = 1$.

We note that $33n^3 \leq 33n^4$, $\sqrt{n} \leq n^4$, and -6 $leq 0n^4$ all hold for $n \geq n_0 = 1$.

Next, note that $34n^4 \leq c \cdot 17n^4$ is true for all values of n and when $c = 2$.

Therefore, we know that $33n^3 + \sqrt{n} - 6 \leq c \cdot 17n^4$ is true for our chosen value of c and for all $n \geq n_0$.

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

Solution:

scratch work: There are a lot of ways to do this one. Normally we would compare to the highest order term in g, but because the constant is larger on the n term, it will be easier to compare to that.

$17\log(n) \leq 17n$ as long as $\log(n) < n$. $\log(n) < n$ whenever $n > 2$. Then we can compare immediately to g: $17\log(n) \leq 17n \leq 32n \leq 32n + 2n\log(n) \leq c(32n + 2n\log(n))$ where it's good enough to set c to 1

proof: One possible solution is $c = 1$ and $n_0 = 2$.

We can convince ourselves this is true by examining our inequalities: $17\log(n) \leq 17n \leq 1 \cdot 32n$ for $n \geq n_0$. Since $2n\log(n)$ is always positve, we have So, since $17\log(n) \leq c \cdot (32 + 2n\log(n))$ is true for our chosen values of c and n_0, we know that $f(n) \in \mathcal{O}(2n\log(n))$.

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.

Solution:

Let $f(n)$ be the running time of the function. Since $f(n)$ is $\mathcal{O}(5n)$, there exist positive constants c, n_0 such that $f(n) \leq c \cdot 5n$ for all $n \geq n_0$. We need to find positive constants c', n_0' such that $f(n) \leq c' \cdot n$ for all $n \geq n_0'$. If we look at the inequality we have, it seems like a good idea to take $c' = 5c$ and $n_0' = n_0$. Then plugging in we have: $f(n) \leq c \cdot 5n = c' \cdot n$ for all $n \geq n_0 = n_0'$, which is what we needed to show $f(n)$ is $\mathcal{O}(n)$.

(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 $\mathcal{O}\left(n^2\right)$ time and `method2` takes $\mathcal{O}\left(n\right)$ 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);
}
```

Solution:

Let's denote the number of operations needed to run `method2` by $g(n)$. What does it mean that `method2` runs in $\mathcal{O}\left(n\right)$ time? It means that there exist numbers c, n_0 such that for all $n \geq n_0$, $g(n) \leq cn$. Let's try to argue about `combined`. When $n \geq 10000$ how many operations does it do? Something like $2 + g(n)$. But we already know that $g(n) \leq cn$ whenever $n \geq n_0$. Now let's try to find a c', n_0' for `combined`.

If $n < 10000$ we won't be using `method2`, so we want to take $n \geq 10,000$. We also want to take $n \geq n_0$, so that we will be able to bound $g(n)$. So set $n_0' = \max\{10000, n_0\}$.

For $n \geq n_0'$, the number of operations is $2 + g(n)$. We have $2 + g(n) \leq 2g(n) \leq 2cn$ for all $n \geq n_0'$. So if we take $c' = 2c$, we have exactly the definition of $\mathcal{O}\left(n\right)$, so the running time of `combined` is $\mathcal{O}\left(n\right)$.

(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 $f(n) < c \cdot g(n)$.

Solution:

Our intution for big-Ω is that it is a lower bound on the running time of the function. `isPrime` is a strange function though, on every even input, it does only about 4 operations (it only goes through the loop once), but on every prime number, it goes through the loop n times. So is the lower bound $\Omega(1)$ or $\Omega(n)$?

Let's see if the running time of `isPrime` is $\Omega(1)$. Regardless of input, the function always goes through the loop at least once, which is 4 operations. So we set $n_0 = 1$ and $c = 4$. For every $n \geq n_0$, the number of operations is at least $4 \cdot 1$, which means the function is $\Omega(1)$.

What about $\Omega(n)$? If we try to do the proof, we're going to get stuck. We want to say for every $n \geq n_0$ we'll take at least cn operations, but no matter what n_0 is, if we take an even number just a bit larger, we'll only do 4 steps and be done. Let's try the other definition (that it's not $\Omega(n)$).

The definition of "not big-Ω" says we need to show something for **every** c, n_0. So let's think about some arbitrary c, n_0. We need to show there exists an $n \geq n_0$ such that $f(n) < c \cdot g(n)$, where $f(n)$ is the number of operations.

Well, we already know that on every even input, we'll always do only 4 operations, which is another way of saying that $f(n) = 4$ when n is even. Can we always choose a big enough even number? Yes! Pick one bigger than n_0 and bigger than $4/c$. For that n we have $c \cdot n > c \cdot 4/c = 4 = f(n)$, so $c \cdot n > f(n)$, which is what we needed to find.

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());
}
```

Solution:

We insert n^2 items into our linked list. Each inserted item will create a new node, which uses up a constant amount of memory. The iterator itself will only *view* the underlying data, without making a copy.

So, the overall memory usage can be modeled as:

$$M(n) = \sum_{i=0}^{n^2-1} c$$

...where c is the amount of memory used per each node.

This is in $\Theta\left(n^2\right)$.

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

Solution:

While we iterate n times, this algorithm only uses up a constant amount of memory. So, the overall memory usage can be modeled as roughly $M(n) = 3c$, where c is the amount of memory used by each int in the array.

This is in $\Theta\left(1\right)$.

(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.

Solution:

This problem is best solved intuitively first, then rigorously after we know what to look for.

We know that the loops run (from outer-most to inner-most) from $i = 0, n$, $j = 0, i$, and $k = 0, j$. If we consider just i and j, we can imagine a sort of "triangle" of values, where i always iterates to n overall but j iterates to 0, then 1, then 2, on and on until j iterates to n. This approximates to $M(n) = \frac{1}{2}n^2$, but because we don't care about constants so much, this really approximates $\Theta\left(n^2\right)$. The same logic follows if we include k, such that overall, this section of code approximates $\Theta\left(n^3\right)$. Now that we know what to expect, the rigorous derivation comes next.

Note that the two nested loops ultimately construct a string of length $\sum_{j=0}^{i-1}\sum_{k=0}^{j-1} c$, where c is the amount of memory used per character.

The code will then insert each string along with the int into the internal array. If we let x represent the amount of memory used per int, we can model the overall memory usage as:

$$M(n) = \sum_{i=0}^{n-1}\left(x + \sum_{j=0}^{i-1}\sum_{k=0}^{j-1} c\right)$$

If we apply summation rules, we can simplify this into:

$$\begin{aligned}\sum i = 0^{n-1}\left(x + \sum_{j=0}^{i-1}\sum_{k=0}^{j-1} c\right) &= \sum_{i=0}^{n-1}\left(x + c\sum_{j=0}^{i-1} j\right) \\ &= \sum_{i=0}^{n-1}\left(x + c\frac{i(i-1)}{2}\right) \\ &= xn + c\frac{1}{6}(n-2)(n-1)n\end{aligned}$$

This is in $\Theta\left(n^3\right)$.

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.

Solution:

We can use two ADTs: a dictionary, which stores each request and the corresponding file data, and a list, which keeps track of the last n requests made.

Here, we will probably want to use a hashmap (which has $\Theta(1)$ lookup) rather then our `ArrayDictionary` which has $\Theta(n)$ lookup.

Every time somebody makes a request, we search to see if it's located inside the list. If it is, we remove it and re-add that request to the very front. If the request is new, we just add it directly to the front.

This will cause the least-frequently made requests to naturally end up near the end of the list. If the list increases beyond a certain length (beyond our cache size), we'll remove them from both list and our dictionary.

This makes any lookup in the worst case $\Theta(n)$, where n is the size of our cache. However, the most frequently made requests will be located near the front of the list, which makes their lookup time roughly constant.