Section 02: Solutions

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.)

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

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:

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

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:

The answers are true, false, true, true, and false respectively.

Clarification regarding 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 certainly 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 k = 0; k < i; k++) {
        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 $\frac{n^2}{3}$ work; the outer loop repeats that n times.

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

(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, because why not).

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 clearly $\mathcal{O}\left(n^4\right)$ in this case.

Although we have not yet fully talked about big-Ω and big-Θ yet, 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.

(Important: there is, however, 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.

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

Solution:

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

First, note that $3n + 4 \leq 3n + 4n = 7n < 7n^2$ is true for all values of $n \geq 1$.

Next, note that $7n^2 \leq c \cdot 5n^2$ is true for all values of n and when $c = \frac{7}{5}$.

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

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

Solution:

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

First, note that $33n^3 + \sqrt{n} - 6 \leq 33n^3 + \sqrt{n} \leq 33n^3 + n^3 = 34n^3 \leq 34n^4$ is true for all values of $n \geq 1$.

Next, note that $34n^4 \leq c \cdot 17n^2$ 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 values of c and n_0.

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

Solution:

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

We can convince ourselves this is true by examining our inequalities:

$$\begin{aligned} 17\log(n) &\leq 17n && \text{for all } n \geq 1 \\ &\leq 34n \\ &= c \cdot 34n && \text{since } c = 1 \\ &= c \cdot (32n + 2n) \\ &\leq c \cdot (32n + 2n\log(n)) && \text{for all } n \geq 2 \end{aligned}$$

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. 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 $\Theta(1)$.

(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:

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 our 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 $\Theta\left(n^3\right)$.

7. Analyzing recursive code

For each of the following recurrences, use any of the methods discussed in class to find a Θ bound on the function in terms of n.

(a) $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + 3 & \text{otherwise} \end{cases}$

Solution:

We first must simplify the function to eliminate the recursion. We can do this by unrolling the recurrence:

$$T(n) = 3 + T(n-1) = 3 + (3 + T(n-2)) = 3 + (3 + \ldots (3 + 1))$$

We repeat the 3 exactly n times. Therefore, we know that $T(n) = 3n + 1$ (the $+1$ is from the base case).

We conclude $T(n) \in \Theta(n)$.

(b) $T(n) = \begin{cases} 10 & \text{if } n = 1 \\ T(n/2) + 4 & \text{otherwise} \end{cases}$

Solution:

We can again unroll the recurrences:

$$T(n) = 4 + T(n/2) = 4 + (4 + T(n/4)) = 4 + (4 + \ldots 10)$$

Because we constantly divide by two, we'll end up unrolling approximately $\log_2(n)$ times. So, we know $T(n) \approx 4\log(n) + 10$.

This is in $\Theta(\log(n))$.

(c) $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n/3) + n & \text{otherwise} \end{cases}$

Solution:

We can again unroll the recurrences:

$$T(n) = n + T(n/3) = n + \left(\frac{n}{3} + T(n/9)\right) = n + \left(\frac{n}{3} + \ldots 1\right)$$

Because we constantly divide by three, we'll end up unrolling approximately $\log_3(n)$ times. So, we know $T(n) \approx \log_3(n) + 1$.

This is in $\Theta(\log(n))$.

8. Application: Binary Search Trees

Consider the following method, which is a part of a Binary Search Tree class.

```
public V find(K key) {
    return find(this.root, key);
```

```
}

private V find(Node<K, V> current, K key) {
    if (current == null) {
        return null;
    }
    if (current.key.equals(key)) {
        return current.value;
    }
    if (current.key.compareTo(key) > 0) {
        return find(current.left, key);
    } else {
        return find(current.right, key);
    }
}
```

(a) We want to analyze the runtime of our `find(x)` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case?

Solution:

In the best possible case, our tree will be completely balanced. In the worst possible case, it will be completely unbalanced, resembling a linked list.

(b) Write a recurrence to represent the worst-case runtime for `find(x)` in terms of n, the number of elements contained within our tree. Then, provide a Θ bound.

Solution:

The recurrence representing the worst-case runtime of `find(x)` is:

$$T_w(n) = \begin{cases} 1 & \text{when } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

That is, every time we recurse, we are able to eliminate only one node from the span of possibilities we must consider.

This recurrence is in $\Theta(n)$.

(c) Assuming we have an optimally structured tree, write a recurrence for the runtime of `find(x)` (again in terms of n). Then, provide a Θ bound.

Solution:

The recurrence representing the best-case runtime of `find(x)` is:

$$T_w(n) = \begin{cases} 1 & \text{when } n = 0 \\ 1 + T(n/2) & \text{otherwise} \end{cases}$$

That is, every time we recurse, we are able to eliminate about half of the nodes we must consider.

This recurrence is in $\Theta(\log(n))$.

9. 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.