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

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. 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, $T_w(n) \in \Theta\left(n^4\right)$.

4. Deques

The most recent homework introduces the concept of a Deque, which you will be working with in this section to solve the problems. Some information about deques is copied below from the homework description.

Deque (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back). Deques can do everything that both stacks and queues can do.

Specifically, any deque implementation must have exactly the following operations.

- `public void addFirst(T item)`: Adds an item of type T to the front of the deque.
- `public void addLast(T item)`: Adds an item of type T to the back of the deque.
- `public boolean isEmpty()`: Returns true if deque is empty, false otherwise.
- `public int size()`: Returns the number of items in the deque.
- `public void printDeque()`: Prints the items in the deque from first to last, separated by a space. Once all the items have been printed, print out a new line.
- `public T removeFirst()`: Removes and returns the item at the front of the deque. If no such item exists, returns null.
- `public T removeLast()`: Removes and returns the item at the back of the deque. If no such item exists, returns null.
- `public T get(int index)`: Gets the item at the given index, where 0 is the front, 1 is the next item, and so forth. If no such item exists, returns null. Must not alter the deque!

You will see two deque implementations in the homework: a deque implemented with a circular array and a deque implemented with linked nodes.

(a) Write a method `removeRandom` that accepts a Deque<Integer> as a parameter and chooses at random to remove and return the first or last element in the given deque. It should be equally likely to remove and return the first or last element. You should use `Math.random()` to help you randomly select the first or last element. `Math.random()` randomly returns a real number in the range [0, 1) (greater than or equal to 0.0 and less than 1.0).

Assume that the given deque contains at least one element.

Solution:

```
public static int removeRandom(Deque<Integer> deque) {
  if (Math.random() < 0.5) {
    return deque.removeFirst();
  }
  return deque.removeLast();
}
```

(b) Write a method `evenOdd` that accepts a List<Integer> as a parameter and returns a Deque<Integer> where all even numbers in the given list appear before the odd numbers in the given list.

For example, if you are given the list [1, 2, 3, 4, 5, 6] then `evenOdd` should return the deque [6, 4, 2, 1, 3, 5].

Solution:

```
public static Deque<Integer> evenOdd(List<Integer> list) {
  Deque<Integer> deque = new LinkedDeque<>(); // ArrayDeque also works
  for (int n : list) {
    if (n % 2 == 0) {
      deque.addFirst(n);
    } else {
      deque.addLast(n);
    }
  }
  return deque;
}
```

(c) Write a method `isReverse` that accepts two Deque<String> as parameters and returns true if the second deque is exactly the reverse of the first deque, false otherwise.

For example, if the first deque stores `["a", "b", "c", "d"]` and the second deque stores `["d", "c", "b", "a"]`, then `isReverse` should return true because the second deque is exactly the first deque reversed.

You may modify the given deques to solve this problem. **You may not use the Deque get method to solve this problem.**

Solution:

One possible solution appears below:

```
public static boolean isReverse(Deque<String> deque1, Deque<String> deque2) {
  if (deque1.size() != deque2.size()) {
    return false;
  }
  while (!deque1.isEmpty()) {
    if (!deque1.removeFirst().equals(deque2.removeLast())) {
      return false;
    }
  }
  return true;
}
```

(d) Implement `isReverse` **without modifying the input deques.** That is to say, if the first deque stores `["a", "b", "c", "d"]`, then after a call to `isReverse`, deque1 should remain in the state `["a", "b", "c", "d"]`.

You may not use the Deque get method to solve this problem.

Solution:

One possible solution appears below. Note, in particular, that **we cannot return false within the loop** because we need the input deques to be in the same state that they were in originally.

```
public static boolean isReverse(Deque<String> deque1, Deque<String> deque2) {
  if (deque1.size() != deque2.size()) {
    return false;
  }
  boolean isReverse = true;
  for (int i = 0; i < deque1.size(); i++) {
    String elt1 = deque1.removeFirst();
    String elt2 = deque2.removeLast();
    if (!elt1.equals(elt2)) {
      isReverse = false;
    }
    // Preserve inputs by adding elt1 and elt2 back to their respective deques.
    deque1.addLast(elt1);
    deque2.addFirst(elt2);
  }
  return isReverse;
}
```

(e) Questions (c) and (d) restricted your use of the `get` method. Why might we want to restrict usage of a method like `get` for a generic deque?

Solution:

Deques can be implemented using a circular array or linked nodes. When working with linked nodes, there isn't an easy way to access data in the middle since we need to continuously follow references to access a node in the middle of a long chain (think about a linked list – there isn't an easy way to know what the third element in a linked list is. You would need to start at the beginning and hop through each node's next pointer). For a linked node implementation, we would expect $\mathcal{O}(n)$ runtime for `get` if n is the size of the deque.

A deque implemented with a circular array, on the other hand, does not have this same problem since we can easily access an element in an array given an index. We would expect $\mathcal{O}(1)$ runtime for `get` for the array implementation.

When writing these methods, we don't know what implementation of a deque the client chose, so it would be best to avoid calls to `get` to avoid the potential runtime addition from continuously calling this method.

Food for thought

5. 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.