Section 02: Asymptotic Analysis

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.)

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

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

(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

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 $\mathcal{O}(n)$.

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;
    }
}
```

(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++;
    }
}
```

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.

(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].

(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.**

(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.

(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?

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.