CS 61BL — Asymptotics

Summer 2022 — Recurring Section 6: Tuesday July 12, 2021

1 Asymptotics

(a) Order the following big-O runtimes from smallest to largest.

$$O(\log n), O(1), O(n^n), O(n^3), O(n \log n), O(n), O(n!), O(2^n), O(n^2 \log n)$$

$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2 \log n) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$

(b) Are the statements in the right column true or false? If false, correct the asymptotic notation ($\Omega(\cdot)$, $\Theta(\cdot)$, $O(\cdot)$). Be sure to give the tightest bound. $\Omega(\cdot)$ is the opposite of $O(\cdot)$, i.e. $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$. *Hint: Make sure to simplify the runtimes first.*

$f(n) = 20501$	$g(n) = 1$	$f(n) \in O(g(n))$
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$

i) False. Although this bound is technically correct, it is NOT the tightest bound. $\Theta(\cdot)$ is a better bound.
ii) False, $O(\cdot)$. Even though n^3 is strictly worse than n^2, n^2 is still in $O(n^3)$ because n^2 is always as good as or better than n^3 and can never be worse.
iii) False. Again, even though technically 2^{2n} is in $O(4^n)$, it is not a tight bound. $\Theta(\cdot)$ is a better bound.
iv) False, $O(\cdot)$. $\log(n^{100}) = 100 * log(n) = log(n)$. Thus, $log(n)$ is in $O(nlogn)$ because $log(n)$ is always better than $nlogn$.
v) True. The 3^n term is worse than all the terms in $g(n)$.
vi) True. The n^2 term dominates the lower order terms - $nlogn$ and $logn$, so $f(n)$ and $g(n)$ both run in $\Theta(n^2)$.

(c) Give the worst case and best case runtime in terms of M and N. Assume `ping` is in $\Theta(1)$ and returns an **int**.

```
for (int i = N; i > 0; i--) {
    for (int j = 0; j <= M; j++) {
        if (ping(i, j) > 64) break;
    }
}
```

Worst: $\Theta(MN)$, Best: $\Theta(N)$ We repeat the outer loop N times, no matter what. For the inner loop, we see the amount of times we repeat it depends on the result of `ping`. In the best case, it returns true immediately, such that we'll only ever look at the inner loop once and then break the inner loop. In the worst case, `ping` is always false and we complete the inner loop M times for every value of N in the outer loop.

(d) Below we have a function that returns true if every int has a duplicate in the array, and false if there is any unique int in the array. Assume `sort(array)` is in $\Theta(N \log N)$ and returns `array` sorted.

```java
public static boolean noUniques(int[] array) {
    array = sort(array);
    int N = array.length;
    for (int i = 0; i < N; i += 1) {
        boolean hasDuplicate = false;
        for (int j = 0; j < N; j += 1) {
            if (i != j && array[i] == array[j]) {
                hasDuplicate = true;
            }
        }
        if (!hasDuplicate) return false;
    }
    return true;
}
```

1. Give the worst case and best case runtime where $N =$ `array.length`.

   Its runtime is $\Theta(NlogN+N^2) = \Theta(N^2)$ for the worst case the if statement always sets x to true. The best case is if we we don't set x to be true in the very first loop, which allows us to only go through the entire array once giving us $\Theta(NlogN + N) = \Theta(NlogN)$.

2. Try to come up with a way to implement `noUniques()` that runs in $\Theta(NlogN)$ time. Can we get any faster?

   We should rely on the fact that a sorted array means all duplicates will be adjacent. `curr` represents the current item we are checking, and we check the item after curr (since our array is sorted) to see if a duplicate exists. There is a possible $\Theta(N)$ solution, but that involves data structures we haven't covered yet!

```java
public static boolean noUniques(int[] array) {
    array = sort(array);
    int N = array.length;
    int curr = array[0];
    boolean unique = true;
    for (int i = 1; i < N; i += 1) {
        if (curr == array[i]) {
            unique = false;
        } else if (unique) {
            return false;
        } else {
            unique = true;
            curr = array[i];
        }
    }
```

```
        return !unique;
    }
```

2 Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("We will miss you next semester Akshit :(");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.

Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = ______) {
        System.out.println("This is one is low key hard");
    }
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = j + 1) {
        System.out.println("This is one is low key hard");
    }
}
```

(b) Desired runtime: $\Theta(log(N))$

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < ______; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Any constant would work here, 2 was chosen arbitrarily.

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < 2; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < ______; j = j + 1) {
        System.out.println("This is one is high key hard");
    }
```

```
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2, i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < ______; i = i * 2) {
    for (int j = 1; j < N * N; j = ______) {
        System.out.println("yikes");
    }
}
```

```
for (int i = 1; i < Math.pow(2, N); i = i * 2) {
    for (int j = 1; j < N * N; j = j + 1) {
        System.out.println("yikes");
    }
}
```

3 Extra: Golden Rule of Equals

The **Golden Rule of Equals** says:

> Given variables `b` and `a`, the assignment statement `b = a` copies all the bits from `a` into `b`.

Passing parameters obeys the same rule: copy the bits to the new scope.

What is wrong with this definition of `swap`? How can we fix it?

```
class SimpleSwap {
    public static void swap(int a, int b) {
        int temp = b;
        b = a;
        a = temp;
    }
    public static void main(String[] args) {
        int x = 2, y = 5;
        System.out.println("x: " + x + ", y: " + y);
        swap(x, y);
        System.out.println("x: " + x + ", y: " + y);
    }
}
```

```
x: 2, y: 5
x: 2, y: 5
```

In the main method, `x` and `y` won't actually be swapped. Within `swap`, we can change what `a` and `b` point to, but we can't change the variables that were declared in `main`. We can fix this by either in-lining the `swap` functionality in the `main` method or returning and reassigning the swapped values using an object.

Meta: Emphasize "Golden Rule of Equals" (pass-by-value). The bits are copied over. It is helpful to talk about the 8 types of primitive variable types (**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**), and that Java stores the actual value in the variable. So when a primitive variable is passed to a function, its value itself is copied over.

How is this implementation of `swap` different?

```
class Coordinate {
    int x, y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class SwapObject {
    public static void swap(Coordinate p) {
        int temp = p.x;
```

```
        p.x = p.y;
        p.y = temp;
    }
    public static void main(String[] args) {
        Coordinate p = new Coordinate(2, 5);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
        swap(p);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
    }
}
```

When calling `swap` with a `Coordinate` object, we're passing a reference to the original `Coordinate` object. The object's instance variables can be changed from within `swap` and will remain changed after we exit from the function.

It can be useful to mention that Java stores only the address of an object in a variable of non-primitive type, and so when passed to a function, the address of the original object gets passed around.

4 Extra: Abstract Data Types

A **list** is an ordered sequence of items: like an array, but without worrying about the length or size.

```
interface List<E> {
    boolean add(E element);
    void add(int index, E element);
    E get(int index);
    int size();
}
```

A **set** is an unordered collection of unique elements.

```
interface Set<E> {
    boolean add(E element);
    boolean contains(Object object);
    int size();
    boolean remove(Object object);
}
```

A **map** is a collection of key-value mappings, like a dictionary in Python. Like a set, the keys in a map are unique.

```
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
    boolean containsKey(Object key);
    Set<K> keySet();
}
```

Define a procedure, `sumUp`, which returns **`true`** if any two values in the `array` sum up to `n`.

```java
public static boolean sumUp(int[] array, int n) {

}
```

```java
public static boolean sumUp(int[] array, int n) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        if (seen.contains(n - value)) {
            return true;
        }
        seen.add(value);
    }
    return false;
}
```

Now define a procedure, `isPermutation`, which returns **`true`** if a string `s1` is a permutation of `s2`. For example, `"atc"` and `"tac"` are permutations of `"cat"`.

```java
public static boolean isPermutation(String s1, String s2) {
```

```
}
```

```
public static boolean isPermutation(String s1, String s2) {
    Map<Character,Integer> characterCounts = new HashMap<>();
    for (char c : s1.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count + 1);
    }
    for (char c : s2.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count - 1);
    }
    for (char c : characterCounts.keySet()) {
        if (characterCounts.get(c) != 0 ) {
            return false;
        }
    }
    return true;
}
```