CS 61B Discussion 7 Spring 2016

1 Welcome O & Omega, Θ's Not Alone!

Order the following big-O runtimes from most to least efficient:

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

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

Are the statements in the right column true or false? If false, correct the asymptotic notation (Ω, Θ, O). Be sure to give the tightest bound. $\Omega(\cdot)$ is the opposite of $O(\cdot)$, i.e. $f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$.

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

$\star$The 1st and 3rd ones are **true**, but Θ is a better bound.

$\star\star$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.

2 Analyzing Runtime

Give the worst case runtime in $O(\cdot)$ notation. Extra: Give the best case runtime in $\Omega(\cdot)$.

A. Use M and N in your result. `ping()` is a constant time, $O(1)$, function that returns an int.

Worst: $O(M+N)$, Best: $\Omega(N)$

the trick is that j is initialized outside the loops!!!!

```
int j = 0;
for (int i = N; i > 0; i--) {
    for (; j <= M; j++) // Can do without {} because there is "1" line after
        if (ping(i, j) > 64)
            break; // ends the loop (only 1 loop, so only the inner loop here)
}
```

B. Use N in your result, where N is the length of `arr`.

Worst: $O(N^2)$, Best: $\Omega(N\log N)$

remember sorting in the begining!!!! thats why nlogn is best (the loop's best case is just N)

```
public static boolean mystery(int[] arr) {
    arr = mrpoolsort(arr); // creates sorted copy of arr in Θ(NlogN) time
    int N = arr.length;
    for (int i = 0; i < N; i += 1) {
        boolean x = false;
        for (int j = 0; j < N; j += 1) {
            if (i != j && arr[i] == arr[j])
                x = true;
        }
        if (!x)
            return false;
    }
    return true;
}
```

Achilles Added Additional Amazing Asymptotic And Algorithmic Analysis Achievements: What is `mystery()` doing?

mystery() returns **true if** every **int** has a duplicate in the array (1,2,1,2 would **return true**) and **false if** there is any unique **int** in the array (1,2,2 is **false**).

Using an ADT, can you rewrite `mystery()` with a better runtime? What about if we make the assumption an int can appear in `arr` at most twice, is there a better way (uses constant memory)?

A $\Theta(N)$ algorithm is to use a Map and **do** key=element and value=number of appearances, then make sure all values are >1. Uses $O(N)$ memory however.

There is none - XOR does not work **for** 1, 2, 3. Can **do** constant space by sorting then going through, but its $O(n \log n)$

3 Have You Ever Went Fast?

Given an integer x and a **sorted** array A[] of N distinct integers, design an algorithm to find if there exists indices i and j such that `A[i] + A[j] == x`.

Let's start with the naive solution:

```
public static boolean findSum(int[] A, int x) {
    for (int i = 0; i < A.length; i++){
        for (int j = 0; j < A.length; j++){
            if (A[i] + A[j] == x)
                return true;
            }
        }
    }
    return false;
}
```

Can we do this faster? Hint: Does order matter here?

```
public static boolean findSumFaster(int[] A, int x){
    int left = 0;
    int right = A.length - 1;
    while (left <= right){
        if (A[left] + A[right] == x)
            return true;
        else if (A[left] + A[right] < x)
            left++;
        else
            right--;
    }
    return false;
    }
```

What is the runtime of both these algorithms?

Naive: Worst = $O(N^2)$, Best = $\Omega(1)$.
Optimized: Worst = $O(N)$, Best = $\Omega(1)$

4 Basic Interview Type Questions (Extra for Experts)

Union: Write the code that returns an array that is the union between two given arrays. The union of two arrays is a list that includes everything that is in both arrays, with no duplicates. Assume the given arrays do not contain duplicates. Ex: Union of 1,2,3,4 and 3,4,5,6 is 1,2,3,4,5,6
Hint: The method should run in $O(M+N)$ time where M and N are the sizes of the two arrays.

```java
public static int[] union(int[] A, int[] B) {
    HashSet<Integer> set = new HashSet<Integer>();
    for (int num : A) {
        set.add(num);
    }
    for (int num : B) {
        set.add(num);
    }
    int[] unionArray = new int[set.size()];
    int index = 0;
    for (int num : set) {
        unionArray[index] = num;
        index += 1;
    }
    return unionArray;
}
```

Intersection: Now do the same as above, but find the intersection between both arrays. The intersection of two arrays is the list of all elements that are in both arrays. Again assume that neither array has duplicates. Ex: Intersection of 1,2,3,4 and 3,4,5,6 is 3,4.
Hint: Think about using ADTs other than arrays to make the code more efficient.

```java
public static int[] intersection(int[] A, int[] B) {
    HashSet<Integer> setOfA = new HashSet<Integer>();
    HashSet<Integer> intersectionSet = new HashSet<Integer>();
    for (int num : A) {
        setOfA.add(num);
    }
    for (int num : B) {
        if (setOfA.contains(num)) {
            intersectionSet.add(num);
        }
    }
    int[] intersectionArray = new int[intersectionSet.size()];
    int index = 0;
    for (int num : intersectionSet) {
        intersectionArray[index] = num;
        index += 1;
    }
    return intersectionArray;
}
```

What is the runtime, $\Omega(\cdot)$ and $O(\cdot)$, of the above algorithm?

$\Theta(N+M)$