CS 61B
Spring 2018

Asymptotic Analysis

Discussion 7: February 27, 2018

1 Asymptotic Notation

$\boxed{1.1}$ Order the following big-O runtimes from smallest to largest.

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

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

$\boxed{1.2}$ 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. $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$.

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

- True, although $\Theta(\cdot)$ is a better bound.
- 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.
- True, although $\Theta(\cdot)$ is a better bound.
- False, $O(\cdot)$.
- True.
- True.
- False, $\Omega(\cdot)$.

2 Analyzing Runtime

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

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

Worst: $\Theta(M + N)$, Best: $\Theta(N)$ The trick is that `j` is initialized outside the loops!

2.2 Give the worst case and best case runtime where N = `array.length`. Assume `mrpoolsort(array)` is in $\Theta(N \log N)$.

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

Worst: $\Theta(N^2)$, Best: $\Theta(N \log N)$ Remember sorting in the beginning!

Achilles Added Additional Amazing Asymptotic And Algorithmic Analysis Achievements

(a) What is `mystery()` doing?

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

(b) Using an ADT, describe how to implement `mystery()` with a better runtime. Then, if we make the assumption an **`int`** can appear in the `array` at most twice, develop a solution using only 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. Can do constant space by sorting then going through, but sorting is generally in $O(n \log n)$ time.

2.3 Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of M and N. Assume that `comeOn()` is in $\Theta(1)$ and returns a boolean.

```
for (int i = 0; i < N; i += 1) {
    for (int j = 1; j <= M; ) {
        if (comeOn()) {
            j += 1;
        } else {
            j *= 2;
        }
    }
}
```

For `comeon()` the worst case is $\Theta(NM)$ and the best case is $\Theta(N \log M)$. To see this, note that in the best case `comeon()` always returns false. Hence `j` multiplies by 2 each iteration. The inner loop would execute relative to $\log M$ and the outer loop iterates `N` times. In the worst case, `comeon()` always returns false, thus the inner loop iterates `M` times.

3 Have You Ever Went Fast?

3.1 Given an **int** 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.

```java
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;
}
```

(a) How can we improve this solution? *Hint*: Does order matter here?

```java
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;
}
```

(b) What is the runtime of both the original and improved algorithm?

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

4 CTCI *Extra*

4.1 **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. For example, the 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 is the size of each array.

```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;
}
```

4.2 **Intersect** 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. For example, the 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;
}
```