CS 61B Spring 2026

Asymptotics II

Exam-Level Discussion 07: February 23, 2026

1 Algorithm Analysis

(a) Say we have a function **findMax** that iterates through an unsorted int array one time and returns the maximum element found in that array. Give the tightest lower and upper bounds ($\Omega(\cdot)$ and $O(\cdot)$) of **findMax** in terms of `N`, the length of the array. Is it possible to define a $\Theta(\cdot)$ bound for **findMax** in the general case?

Because the array is unsorted, we don't know where the max will be, so we have to iterate through the entire array to ensure that we find the true max. Therefore, we know that we can never go faster than linear time with respect to the length of the array. Since the function is both lower and upper bounded by N, we can say that the function is theta-bounded by N as well ($\Theta(N)$).

(b) Give the worst case and best case runtime in terms of M and N. Assume **ping** runs 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 case runtime: $\Theta(N)$	Best case runtime: $\Theta(N)$

We repeat the outer loop N times, no matter what. For the inner loop, 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.

(c) 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.

```
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;
}
```

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

Worst case runtime: Θ (N)	Best case runtime: Θ (N)

Notice that we call **`sort`** at the beginning of the function, which we are told runs in $\Theta(N \log N)$.

First, we consider the best case. We notice that if **`hasDuplicate`** is false after the inner loop (i.e. **`! hasDuplicate`** has truth value **`true`**) we can exit the **`for`** loop early via the return statement on line 11. Thus, the best case is when we never set **`hasDuplicate`** to be **`true`** during the first time we run the inner loop. In this case, we can return after only looping through the array once, giving us $\Theta(N \log N + N) = \Theta(N \log N)$.

For the worst case, we notice that if **`hasDuplicate`** is always set to **`true`** by the inner loop, we never return on line 11. Thus, we consider the worst case where **`hasDuplicate`** is always set to **`true`** in every loop, forcing us to have to loop fully through both the inner and outer loop. One such input is an array of all the same integer! Since we have to fully loop through both loops, our worst-case runtime is $\Theta(N \log N + N^2) = \Theta(N^2)$.

2 The Re-Cursed Swamp

To help visualize the solutions better, a video walkthrough of this problem is linked here!

Sometimes, it isn't possible to give a theta bound for an entire function. In these cases, it's best to analyze the best and worst-case inputs and evaluate the runtime on those to produce a "tightest" upper and lower bound.

(a) Consider the function below...

```
public static int curse(int N) {
    if (N % 2 = 0 || N <= 0) {
        return 0;
    } else {
      for (int i = 0; i < N; i++) {
        System.out.println("You have been cursed!");
      }
      return curse(N - 2);
    }
}
```

What type of input will result in the best-case runtime? What type of input will result in the worst-case runtime?

The best-case input is any arbitrarily large even number. In this case, the top conditional will immediately cause the function to return, giving it

Give a tight Ω and O bound that correspond to the lower and upper bounds on this function's runtime. That is, don't just say sometihng like $O(2^N)$ which is technically true, but useless.

Feel free to use the tree-drawing technique from questions 1 and 2!

Lower bound: Ω (N^2)	Upper bound: O (N^2)

(b) Give the tightest runtime bound(s) for the function below. We can assume the **System.arraycopy** method takes $\Theta(N)$ time, where N is the number of elements copied. The official signature is **System.arrayCopy(Object sourceArr, int srcPos, Object dest, int destPos, int length)**. Here, **srcPos** and **destPos** are the starting points in the source and destination arrays to start copying and pasting in, respectively, and **length** is the number of elements copied.

```
public static void silly(int[] arr) {
    if (arr.length <= 1) {
      return;
    }

    int newLen = arr.length / 2;
    int[] firstHalf = new int[newLen];
    int[] secondHalf = new int[newLen];

    System.arraycopy(arr, 0, firstHalf, 0, newLen);
    System.arraycopy(arr, newLen, secondHalf, 0, newLen);

    silly(firstHalf);
    silly(secondHalf);
}
```

$\Theta(N)$

At each level, we do N work, because the call to **`System.arraycopy`**. You can see that at the top level, this is N work. At the next level, we make two calls that each operate on arrays of length N/2, but that total work sums up to N. On the level after that, in four separate recursive function frames we'll call **`System.arraycopy`** on arrays of length N/4, which again sums up to N for that whole layer of recursive calls.

Now we look for the height of our recursive tree. Each time, we halve the length of N, which means that the length of the array N on recursive level k is roughly $N * \left(\frac{1}{2}\right)^k$. Then we will finally reach our base case $N \leq 1$ when we have $N * \left(\frac{1}{2}\right)^k = 1$. Doing some math, we see this can be transformed into $N = 2^k$, which means $k = \log_2(N)$. In other words, the number of layers in our recursive tree is $\log_2(N)$. If we have $\log_2(N)$ layers with $\Theta(N)$ work on each layer, we must have $\Theta(N \log(N))$ runtime.

(c) Given that **exponentialWork** runs in $\Theta(3^N)$ time with respect to input N, give the tightest runtime bound(s) for **yellowWood**.

Hint: This one is hard! Drawing trees will be of utmost importance. If you suspect that the runtime cannot be theta-bounded, drawing one for the best case and one for the worst case can be a good idea.

```
public void yellowWood(int N) {
    if (Math.random() > 0.9) {
      twoPathsDiverge(N, 2);
    } else {
      twoPathsDiverge(N, 1);
    }
}

private void twoPathsDiverge(int N, int j) {
  if (N <= 1) {
    return;
  }
  exponentialWork(N);
  for (int i = 0; i < 3; i++) {
    twoPathsDiverge(N - j, j);
  }
}
```

$\Omega(3^N)$, $O(N * 3^N)$

For the best case, the first level has $\Theta(3^N)$ work, the next level has $\Theta(3^{N-1})$ work, and so on until the last level which has approximately $\Theta\left(3^{\frac{N}{2}}\right)$ work. This gives the sum $\Theta(3^N) + \Theta(3^{N-1}) + ... + \Theta\left(3^{\frac{N}{2}}\right) = \Theta(3^N)$.

For the worst case, each level has $\Theta(3^N)$ work. Given that there are $\sim N$ layers, summing up the tree results in $O(N * 3^N)$ for an upper bound.

3 Asymptotics is Fun!

(a) Using the function `g` defined below, what is the runtime of the following function calls? Write each answer in terms of `N`. Feel free to draw out the recursion tree if it helps.

```
public static void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= x; i++) {
        g(N - 1, i);
    }
}
```

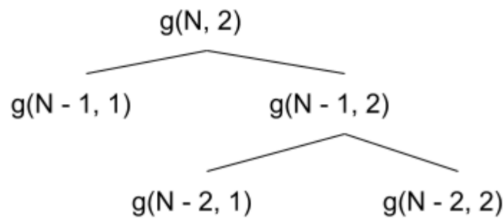
g(N, 1): Θ (N)	g(N, 2): Θ (N)

`g(N, 1)`: $\Theta(N)$

Explanation: When `x` is 1, the loop gets executed once and makes a single recursive call to `g(N - 1)`. The recursion goes `g(N)`, `g(N - 1)`, `g(N - 2)`, and so on. This is a total of `N` recursive calls, each doing constant work.

`g(N, 2)`: $\Theta(N^2)$

Explanation: When `x` is 2, the loop gets executed twice. This means a call to `g(N)` makes 2 recursive calls to `g(N - 1, 1)` and `g(N - 1, 2)`. The recursion tree looks like this:



From the first part, we know `g(..., 1)` does linear work. Thus, this is a recursion tree with N levels, and the total work is $(N-1)+(N-2)+...+1 = \Theta(N^2)$ work.

(b) Suppose we change line 6 to `g(N - 1, x)` and change the stopping condition in the for loop to `i <= f(x)` where `f` returns a random number between 1 and `x`, inclusive. For the following function calls, find the tightest Ω and big O bounds. Feel free to draw out the recursion tree if it helps.

```
public static void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= f(x); i++) {
        g(N - 1, x);
    }
}
```

g(N, 2): Ω (N)	g(N, 2): O (2^N)
g(N, N): Ω (N)	g(N, N): O (N^N)

Explanation: Suppose `f(x)` always returns 1. Then, this is the same as case 1 from (a), resulting in a linear runtime.

On the other hand, suppose `f(x)` always returns x. Then `g(N, x)` makes `x` recursive calls to `g(N - 1, x)`, each of which makes `x` recursive calls to `g(N - 2, x)`, and so on, so the recursion tree has 1, x, x^2 ... nodes per level. Outside of the recursion, the function `g` does `x` work per node. Thus, the overall work is $x * 1 + x * x + x * x^2 + ... + x * x^{N-1} = x(1 + x + x^2 + ... + x^{N-1})$.

Plug in `x = 2` to get $2(1 + 2 + 2^2 + ... + 2^{N-1}) = O(2^N)$ for our first upper bound. Plug in `x = N` to get $N(1 + N + N^2 + ... + N^{N-1}) = O(N^N)$ (ignoring lower-order terms).