CS 61B Discussion 6 Fall 2018

1 Basic Algorithmic Analysis

For each of the following function pairs f and g, list out the Θ, Ω, O relationships between f and g, if any such relationship exists. For example, $f(x) \in O(g(x))$.

For all the problems below, you should be able to eye the asymptotic relations without thinking about the limits which rigorously define them.

1. $f(x) = x^2$, $g(x) = x^2 + x$

   $f(x) \in \Theta(g(x))$: When comparing polynomials the only thing that matters is the degree

2. $f(x) = 50000x^3$, $g(x) = x^5$

   $f(x) \in O(g(x))$: Same as above, and $5 > 3$

3. $f(x) = \log(x)$, $g(x) = 5x$

   $f(x) \in O(g(x))$: Polynomials always grow faster than logarithms

4. $f(x) = e^x$, $g(x) = x^5$

   $f(x) \in \Omega(g(x))$: Exponential growth is always faster than polynomial growth.

5. $f(x) = \log(5^x)$, $g(x) = x$

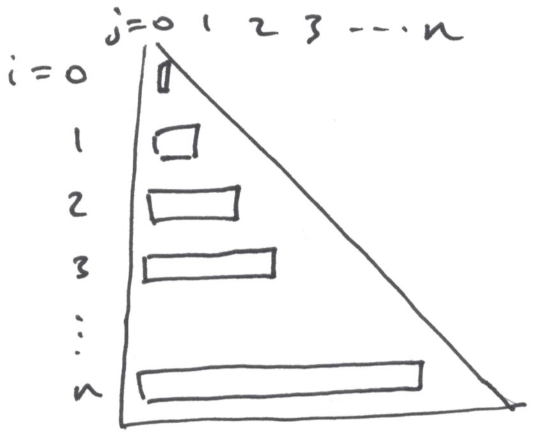
   $f(x) \in \Theta(g(x))$: It is a useful fact to remember that $\log_b(a)$ and $\log_c(a)$ differ by a constant multiple for any pair (b, c). In particular, $\log(5^x) = \alpha \log_5(5^x) = \alpha x$ for some α (remind yourself of the change-of-base formula if you're curious what α is!) You could also say that $\log(5^x) = x \log 5$ using the power rule for logarithms.

2 Practice with Runtime

For each of the following functions, find the Big-Theta expression for the runtime of the function in terms of the input variable n.

1. For this problem, you may assume that the static method *constant* runs in $\Theta(1)$ time.

   The outer nested loop runs n times, and the inner nested loop asymptotically runs n times (the actual number of times varies, but linearly with n), that means the first double-for loop runs in $\Theta(n^2)$ time (since the activity per inner loop - a print statement - runs in time independent of n). You can see this visualized using bars in the figure below. Notice that the area of the work represented by the bars is in the shape of a triangle, which has an area of $\frac{1}{2}nn \implies \Theta(n^2)$. The second loop runs in $\Theta(n)$ time, so overall we have $\Theta(n^2+n)$, but this is the same as $\Theta(n^2)$ (as shown in question 1-1), which is the answer.

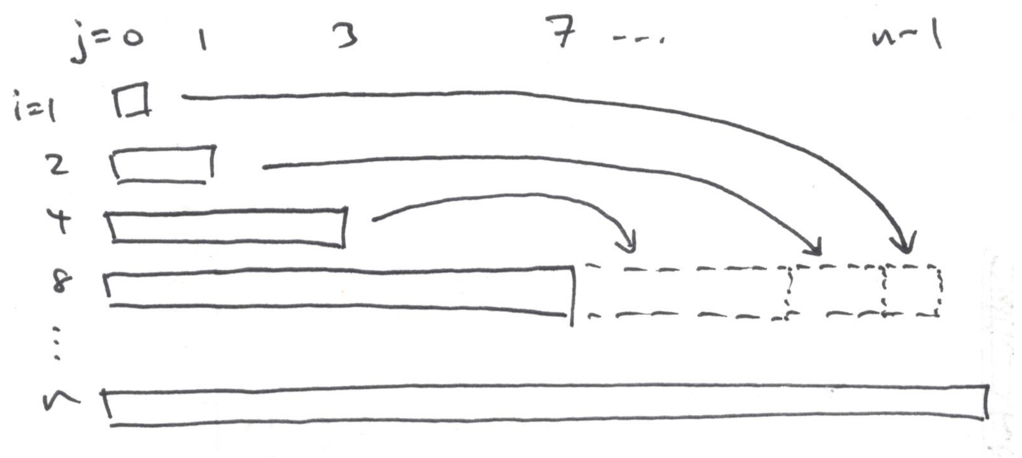


```java
public static void bars(int n) {
    for (int i = 0; i < n; i += 1) {
        for (int j = 0; j < i; j += 1) {
            System.out.println(i + j);
        }
    }

    for (int k = 0; k < n; k += 1) {
        constant(k);
    }
}
```

2. This one is trickier! Note that in the final iteration of the outer loop, the inner loop will run n times. The iteration before, the inner loop will run $\frac{n}{2}$ times, and before that $\frac{n}{4}$, and so on. Abstracting, you can see that the number of times the inner loop will run is $n + \frac{n}{2} + \frac{n}{4} + \ldots$. To figure out how many times the outer loop runs, we need to know how many times (starting with 1) you can double before reaching n: $\log_2(n)$. Therefore that sum is $\sum_{i=0}^{\log(n)} \frac{n}{2^i}$, which you might see is bounded by $2n$. Thus the total number of times the (constant-time) print statement is in $O(2n)$, and the overall runtime is $\Theta(n)$.

   Once again, you can also arrive at this answer by drawing bars. Notice that the length of the bars is doubling each time, until we get to the bar of length n. Well, we know that the bar that came before it was of length $\frac{n}{2}$, and the one before that was half of that–length $\frac{n}{4}$. How many bars can we move next to the $\frac{n}{2}$ bar so that it is as long as the n bar? Moving the $\frac{n}{4}$ bar next to it puts the combined length at $\frac{3n}{4}$. Pulling the bar before that in next to them puts it at $\frac{3n}{4} + \frac{n}{8} = \frac{7n}{8}$. In fact, we will need ALL of the preceding bars to get a total length that is (essentially) equal to n. So overall, we have (almost) $2n$ worth of bars. $2n \implies \Theta(n)$.



```java
public static void barsRearranged(int n) {
    for (int i = 1; i <= n; i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("moove");
        }
    }
}
```

3 A Bit with some Bits

Complete the following method such given a list of integers, it returns an integer such that the i^{th} bit of the return value is 1 if and only if more than half of the integers in the list have 1 in the i^{th} bit. Keep in mind that Java **int**s are 32 bits long!

For example, if bitList was $[1,3]$, then in binary this would be $[(01)_2,(11)_2]$ (with 30 more zeros in front of each number), and the result would be $(01)_2 \implies 1$, since the right-most digit was 1 for more than half the numbers, but the second-from-the-right digit was not 1 for more than half the numbers.

Note: the solution to this question isn't very complicated, but it's not short! Try breaking it down into components, and ask your neighbors for help!

I'm sure there are multiple solutions to this problem. In general, the easiest thing to do is to keep track of some variable (`result` here). At each i-stage, generate a number (`mask`) that represents a number with all 0s except for a 1 at the correct i^{th} spot. You then *AND* through all the numbers in your list and see if the result is equal to the number j. If so, you increment some counter. If, after going through your list, your counter is big enough, you flip the bit in `result` at the i^{th} spot. Using bit-*OR* is the easiest way to increment the right place in `result`.

```java
public static int bitVote(int[] bitList) {
        int result = 0;
        for (int i = 0; i < 32; i++) {
                int mask = 1 << i;
                int count = 0;
                for (int k : bitList) {
                        if ((k & mask) != 0) {
                                count++;
                        }
                }
                if (count > bitList.length/2) {
                        result = result | mask;
                }
        }
        return result;
}
```