CS 61B Discussion 7: Asymptotics Fall 2021

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

   $f(x) \in \Theta(g(x))$

   Explanation 1: It is useful 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. (If you want to find the value of α, try using the change of base formula.)

   Explanation 2: Using the power rule of logarithms, $\log(5^x) = x \log 5$.

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.

You may find the following relations helpful:

$$1+2+3+4+\cdots+N=\Theta(N^2)$$

$$1+2+4+8+\cdots+N=\Theta(N)$$

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

```
1  public static void bars(int n) {
2      for (int i = 0; i < n; i += 1) {
3          for (int j = 0; j < i; j += 1) {
4              System.out.println(i + j);
5          }
6      }
7
8      for (int k = 0; k < n; k += 1) {
9          constant(k);
10     }
11 }
```

$\Theta(n^2)$. For an explanation, see below.

Let's start our analysis with the second for loop beginning at line 8.

1. For each loop iteration, we call `constant(k)` exactly once (line 9). We are given that `constant` runs in $\Theta(1)$ time, which means that the runtime of `constant` does not depend on the input to the method `bars` (the value of `n`).
2. The for loop runs a total of n iterations. Therefore, the total runtime of the for loop at line 8 is (total number of iterations) $\cdot$ (work per iteration) $= n \cdot 1 = n$.

   Note: You can only multiply (total number of iterations) by (work per iteration) if the amount work done for each iteration is the same for every iteration. In our case, `constant(k)` *always* runs in $\Theta(1)$ time. This will not always be the case, as we will see in a bit.

Now let's look at the nested for loop beginning at line 2.

3. For each iteration of the inner for loop, we call `System.out.println(i + j)` exactly once. Since the runtimes for calling `System.out.println` and evaluating `i + j` do not depend on the value of n, the runtime for each call to `System.out.println(i + j)` is $\Theta(1)$.
4. The inner for loop (line 3) runs a total of i times. The actual value of i changes each time the inner loop runs (it depnds on the value set by the outer for loop). Each time the inner for loop runs, its runtime is (total number of iterations) $\cdot$ (work per iteration) $= i \cdot 1 = i$ (note that we cannot evaluate i just yet, as the value of i changes over time).

5. The outer for loop (line 2) runs a total of n times. The amount of work done inside of the inner for loop is i (this was calculated from the previous step). Unlike earlier, we are **not** allowed to multiply the number of iterations by the work done for each iteration because the value of i changes for each iteration of the outer loop. Instead, we can add up the total amount of work being done across all iterations of the outer for loop. Below is a table showing the amount of work done for each iteration of the outer loop.

i	0	1	2	3	...	n - 1
work per iteration	0	1	2	3	...	n - 1

To get the total runtime of the nested for loop, all we have to do is add up the work done for all the iterations:

$$0+1+2+3+...+n-1 = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2} \implies n^2$$

.

Therefore, the total runtime for the nested for loop is $\Theta(n^2)$.

All that's left is to sum up the runtimes!

6. We calculated a runtime of n for the for loop at line 10, and n^2 for our nested for loop at line 2. Therefore, our total runtime for `bars` is the sum of the two runtimes: $\Theta(n+n^2) \implies \Theta(n^2)$.

2. Determine the runtime for `barsRearranged`.

```
1  public static void cowsGo(int n) {
2      for (int i = 0; i < 100; i += 1) {
3          for (int j = 0; j < i; j += 1) {
4              for (int k = 0; k < j; k += 1) {
5                  System.out.println("moove");
6              }
7          }
8      }
9  }
10
11 public static void barsRearranged(int n) {
12     for (int i = 1; i <= n; i *= 2) {
13         for (int j = 0; j < i; j += 1) {
14             cowsGo(j);
15         }
16     }
17 }
```

$\Theta(n)$. See the next page for an explanation.

We'll begin by determining the runtime of `cowsGo`.

1. The method `cowsGo` prints "moove" many times to standard output. Printing "moove" to standard output takes constant time (this was determined from the previous question).
2. Now we'll look at the total number of times "moove" gets printed. Note that the number of times "moove" gets printed out doesn't depend on the value of the input to `cowsGo(n)`. In fact, `cowsGo` always prints out "moove" a constant number of times, regardless of the value of `n`. Therefore, the runtime of `cowsGo` is $\Theta(1)$.

Now we'll begin analyzing `barsRearranged`.

3. The inner for loop (line 13) calls the method `cowsGo` exactly once per iteration. From the previous step, `cowsGo` has a constant runtime. Therefore, each iteration of the inner for loop does $\Theta(1)$ work.
4. The inner for loop (line 13) runs a total of i times. The value of i changes each time the inner loop runs and depends on the value set by the outer loop. Each time the inner loop runs, its runtime is (total number of iterations) $\cdot$ (work per iteration) = $i \cdot 1 = i$. Again, we cannot evaluate i just yet as its value changes over time.
5. Now, let's count the number of times the outer for loop runs. We see that i is initialized to a value of 1 and is doubled until it reaches the value of n. As a result, the outer loop runs a total of $log(n)$ times. The amount of work done inside of the of the inner for loop is i (this was calculated from the previous step). Because the value of i changes for each iteration of the outer loop, we will add up the total amount of work being done across all iterations of the outer loop to get our final runtime. Below is a table showing the amount of work done for each iteration of the outer loop.

i	1	2	4	8	...	n
work per iteration	1	2	4	8	...	n

To get the total runtime of the nested for loop, all we have to do is add up the work done for all the iterations:

$$1+2+4+8+...+n = 2^0+2^1+2^2+2^3+...+2^{log_2(n)} = \sum_{i=0}^{log_2 n} 2^i \in \Theta(2n) \implies \Theta(n)$$

.

Another way to think of it is to rearrange the sum as follows:

$$1+2+4+8+...+n = n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+...+4+2+1 = \sum_{i=0}^{log_2 n} \frac{n}{2^i} \in \Theta(2n) \implies \Theta(n)$$

.

Therefore, the total runtime for the `barsRearranged` is $\Theta(n)$.

A note on the following expression (let's call it expression 1):

$$\sum_{i=0}^{log_2 n} 2^i = \sum_{i=0}^{log_2 n} \frac{n}{2^i} \in \Theta(2n)$$

We can arrive at $\Theta(2n)$ using the equation for finding the sum of a finite geometric series, which is defined below (expression 2).

$$S_n = \sum_{i=1}^{n} a_1 r^{i-1} = a_1 (\frac{1-r^n}{1-r})$$

We can adapt the equation above to help us evaulate expression 1:

$$\begin{aligned}
\sum_{i=0}^{log_2 n} 2^i &= \sum_{i=1}^{log_2 n+1} 2^{i-1} \\
&= \sum_{i=1}^{log_2 n+1} 1 \cdot 2^{i-1} \\
&= 1 \cdot (\frac{1-2^{(\log_2 n+1)}}{1-2}) \\
&= \frac{1-(2 \cdot 2^{\log_2 n})}{-1} \\
&= -1+2n \\
&= 2n-1 \in \Theta(2n) \implies \Theta(n)
\end{aligned}$$

3 A Bit on Bits

Recall the following bit operations and shifts:

1. Mask (x & y): yields 1 only if both bits are 1.
   01110 & 10110 = 00110
2. Set (x | y): yields 1 if at least one of the bits is 1.
   01110 | 10110 = 11110
3. Flip (x ^ y): yields 1 only if the bits are different.
   01110 ^ 10110 = 11000
4. Flip all (~ x): turns all 1's to 0 and all 0's to 1.
   ~ 01110 = 10001
5. Left shift (x << left_shift): shifts the bits to the left by left_shift places, filling in the right with zeros.
   10110111 << 3 = 10111000
6. Arithmetic right shift (x >> right_shift): shifts the bits to the right by right_shift places, filling in the left bits with the current existing leftmost bit.
   10110111 >> 3 = 11110110
   00110111 >> 3 = 00000110
7. Logical right shift (x >>> right_shift): shifts the bits to the right by right_shift places, filling in the left with zeros.
   10110111 >>> 3 = 00010110

Implement the following two methods. For both problems, `i=0` represents the least significant bit, `i=1` represents the bit to the left of that, and so on.

1. Implement `isBitIOn` so that it returns a boolean indicating if the `ith` bit of `num` has a value of 1. For example, `isBitIOn(2, 0)` should return `false` (the 0th bit is 0), but `isBitIOn(2, 1)` should return `true` (the 1st bit is 1).

```
/** Returns whether the ith bit of num is a 1 or not. */
public static boolean isBitIOn(int num, int i) {
    int mask = 1 << i;
    return (mask & num) != 0;
}
```

2. Implement `turnBitIOn` so that it returns the input number but with its `ith` significant bit set to a value of 1. For example, if `num` is 1 (1 in binary is 01), then calling `turnBitIOn(1, 1)` should return the binary number 11 (aka 3).

```
/** Returns the input number but with its ith bit changed to a 1. */
public static int turnBitIOn(int num, int i) {
    int mask = 1 << i;
    return num | mask;
}
```