CSM Week 4 Review

Asymptotic Analysis

Why use asymptotic anaylsis and how?

Suppose I'm trying to compare my code to a friend's code to see whose is better. The best way to do this is to simply measure whose code is faster. One way to measure speed is to time how fast the code runs. However, this can vary based on language, compiler, type of machine, etc. Therefore, it is not an objective way of measuring runtime!

The best way to measure runtime is to measure the **number of operations based on input size** (*size of the parameter*).

By operations, we mean:
- Basic operations (a += 1)
- Assignments (a = b)
- For/while loops
- Recursion

Runtime is often measured as a function, *f(N)*. We use notations to represent the runtime of a method with a familiar function (*more explanation in the right column and next page*). If the parameter is an array, we'll measure the runtime of the method by the length of the array (*N = array.length*).

Asymptotic analysis in CS 61B is pretty reminiscent of CS 61A orders of growth. Since we are measuring the runtime of a method at large inputs, we can still do many simplifications (think of it as we're taking the limit of the function f(N) to N -> ∞).

Drop constants: θ(N + 1) = θ(N); θ(5N) = θ(N)

Drop smaller terms: θ(N + log N) = θ(N)

Omit logarithm bases: Since the bases of a logarithmic function can be converted by a constant multiplier, the base of θ(log N) does not need to be specified!

Common Orders of Growth:
Constant — θ(1)
Logarithmic — θ(log N)
Linear — θ(N)
Polynomial — $\theta(N^x)$
Linearithmic — θ(N log N)
Exponential — $\theta(2^N)$
Factorial — θ(N!)

These orders of growth are functions that we use to define and compare runtimes.

Sometimes, it's hard to tell if the function that represents a runtime fits a function included in the list of orders of growth. Intstead, we will consider the upper bound and lower bound of the runtime function in terms of these familiar functions to make the runtime easier to compare.

Notations

O - "Big O" notation, upper bound of f(N)

Comparing two functions, f(N) and g(N):

$$0 \leq \lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty$$

Ω - "Big Omega" notation, lower bound of f(N)

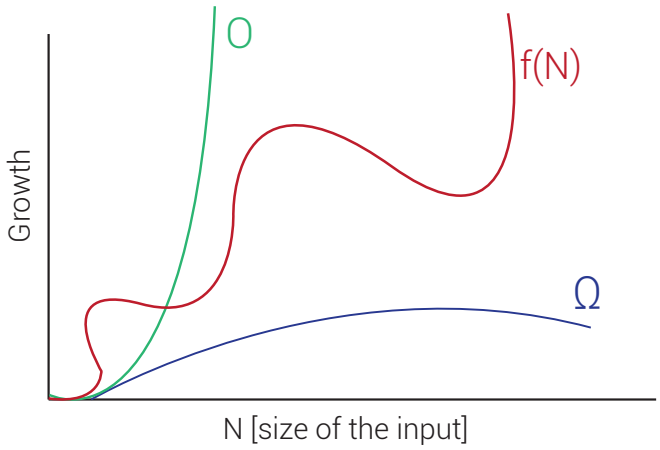
Comparing two functions, f(N) and g(N):

$$0 < \lim_{N \to \infty} \frac{f(N)}{g(N)} \leq \infty$$

θ - "Big Theta" notation, tight bound of f(N), use only if Ω = O!

Comparing two functions, f(N) and g(N):

$$0 < \lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty$$



The graph to the left is a good visualization. f(N) is the measured runtime of a method and it's difficult to tell what type of function it is. Therefore, we use O as the upper bound and Ω as the lower bound to better define f(N). Note: We might notice that O is less than f(N) at the very beginning. This is totally okay! We're only looking at *large* inputs for asymptotic analysis and O will always be greater than f(N) at larger inputs.

Best case vs Worst Case vs Overall Case

Best case: the lower bound of the method runtime, uses θ notation

Worst case: the upper bound of the method runtime, uses θ notation

Overall case: tight bound of the method runtime, uses θ notation if best case = worst case; otherwise, uses Ω notation denoting the best case runtime and O notation denoting the worst case runtime

When a question asks for the **tight asymptotic runtime bound**, it is looking for the "tightest" bound possible. This is a runtime in θ notation. If a θ bound does not exist, then the tightest bound should be the worst case runtime in O notation.

If the explanation above is still confusing, let's walkthrough the problem below! Suppose, we're measuring runtime by the N = array.length.

```
boolean checkForOne(int[] array) {
     for (int i: array) {
          if (i == 1) {
               return true;
          }
     }
     return false;
}
```

Tight Asymptotic Runtime Bound: O(N)

We are looking for the tightest bound, but no tight bound exists! The upper bound and lower bound runtime do not match.Then, we default to using the upper bound of our runtime to describe our tight bound. Note: In this course, we never use Ω to describe the tight asymptotic runtime.

Best case: Ω(1)

The best scenario is when the first element in the array is equal to 1. This implies that the array can be extremely long and it wouldn't matter because checkForOne would terminate immediately after seeing the first element is 1. Therefore, the best case runtime does not depend on input size and must be constant.

Worst case: O(N)

In the worst scenario, the array would contain no elements equal to 1. Therefore, checkForOne will iterate through the entire array and return false. Now, the runtime is dependent on the size of the array! The worst case runtime must be linear!

Overall case: Ω(1), O(N)

No tight bound exists since the best case does not equal the worst case. Therefore, the overall case must be in Ω and O notation!

Algorithm Analysis

```
public static int crimsonTuna(int[][] array) {
     if (array.length < 4) {
          return 0;
     }
     for (int i = 0; i < array.length; i++) {
          for (int j = 0; j < array[i].length; j++) {
               if (i == 4) {
                    return -1;
               }
          }
     }
     return 1;
}
```

The problem above is from last week's worksheet. It wasn't clear whether we were taking the runtime in regards to M, the number of rows in the array, or N, the number of columns in the array. Therefore, we can consider the three scenarios below! Note: The exam will be much clearer about the input we're considering! There shouldn't be anything this complicated!

1. **N -> ∞, M = some constant**

   Best case: Ω(1)
   Since M can be any number (including small numbers), the best case would be when M < 4 and the method is terminated after going into the first if-statement.

   Worst case: O(N)
   In the worst case, M is also a large input, so we would go into the double for loop. The runtime would depend on the number of columns we iterate through each row (N) until we reach the 4th row.

   Tight asymptotic runtime: O(N)

2. **N = some constant, M -> ∞**

   Best case: Ω(N)
   Unlike the first scenario, we would not be able to go into the first if-statement since M is a large input. The best case scenario would be the first scenario's worst case as well!

   Worst case: O(N)
   Same reason as the best case!

   Tight asymptotic runtime: θ(N)

3. **N -> ∞, M -> ∞**

   Best case: Ω(N)
   Same reason as the second scenario's best case!

   Worst case: O(N)
   Same reason as the best case!

   Tight asymptotic runtime: θ(N)

Difference between N and 2^N

N and 2^N have very similar summations:

$1 + 2 + 4 + 8 + ... + 2^N = 2^{N+1} - 1 \rightarrow \theta(2^N)$ Number of terms: N

$1 + 2 + 4 + 8 + ... + N = 2N - 1 \rightarrow \theta(N)$ Number of terms: log N

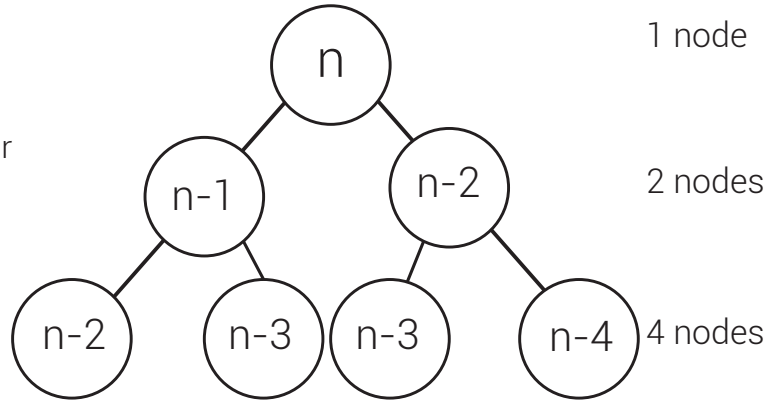
However, the context in which they are used is different:

For 2^N, let's use the fibonacci method as an example:

```
int fibonacci(int n) {
      if (n <= 1) {
            return n;
      }
      else {
            return fibonacci(n - 1) + fibonacci(n - 2);
      }
}
```

We can use a tree to draw out the number of recursive calls:

The height of the tree would be n, which is the number of times we make a recursive call before we satisfy the base case.



Since each node in this tree does the same amount of work (n), we can sum up the number of nodes to get the runtime!

We end up with the summation:

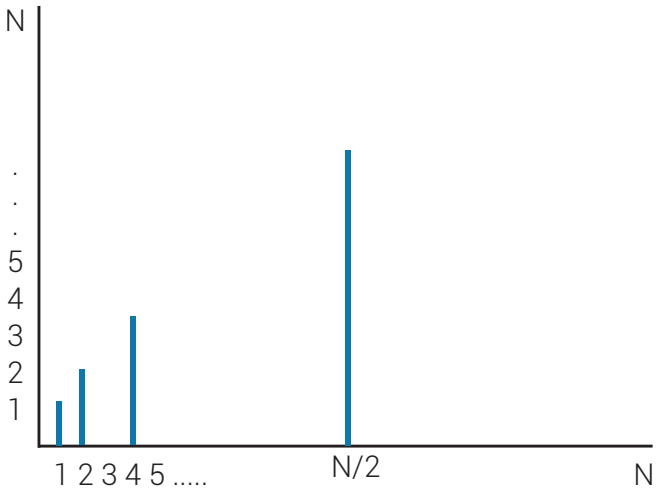
$1 + 2 + 4 + 8 + ... + 2^n = 2^{n+1} - 1 \rightarrow \theta(2^n)$

We know because we have n levels from the height, which implies we have n total terms!

For N, let's use a method called piltover:

```
int piltover(int N) {
      int[] arr = new int[N];
      for (int i = 1; i < N; i *= 2) {
            for (int j = 0; j < i; ++j) {
                  arr[j] += j;
            }
      }
      return arr;
}
```

For complicated loops, it often helps to draw out bar graphs to model what's happening. In this case, we can plot out the values of i across the x-axis and the number of operations that occur for each corresponding i value on the y-axis.



From the graph, we can tell that i is called log N times since not every number on the x-axis is used. When we set up the summation, we now know that there are log N terms total!

If we add up each i, we get the summation:

$1 + 2 + 4 + 8 + ... + N = 2N - 1 \rightarrow \theta(N)$

Note: We can also tell that the number of operations corresponding to each i is i as well! Just from looking at the graph, we can tell the relationship between each step in i is linear -> $\theta(N)$.