Asymptotic Analysis

Overview

Asymptotic analysis is essentially the method we use to measure how fast a particular function runs. We describe the runtimes of functions with three different bounds.

Big O Bound (O)
When we say a function f is $O(g)$, we mean that f is upper bounded by g. This means that as the inputs, n, of these functions grow to infinity, then $f(n)$ will be smaller than $g(n)$. Formally, big O is defined as, if $f(n) = O(g)$, then there exists positive constants c and k such that $0 < f(n) < cg(n)$ for all $n \geq k$.

Big Omega Bound (Ω)
When we say a function f is $\Omega(g)$, we mean that f is lower bounded by g. This means that as the inputs, n, of these functions grow to infinity, then $f(n)$ will be greater than $g(n)$. Formally, big omega is defined as, if $f(n) = \Omega(g)$, then there exists positive constants c and k such that $0 < cg(n) < f(n)$ for all $n \geq k$.

Big Theta Bound (Θ)
When we say a function f is $\Theta(g)$, we mean that f is upper bounded and lower bounded by g. This means as the inputs, n, of these functions grow to infinity, then $f(n)$ will be between different values of $g(n)$. Formally, big O is defined as, if $f(n) = \Theta(g)$, then there exists positive constants c_1, c_2 and k such that $c_1 g(n) < f(n) < c_1 g(n)$ for all $n \geq k$. To prove a function is Θ bounded by another function, we just have to show the first function is upper and lower bounded by the second function. For example, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$.

Asymptotic Relationship Between Two Functions

When given two functions f and g, we can use limits to find the asymptotic relationship between these two functions.

- If $f(n) = O(g(n))$, then we know that f is upper bounded by g, so as the input, n, approaches ∞, $g(n)$ is larger than $f(n)$. In terms of limits this is $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- If $f(n) = \Omega(g(n))$, then we know that f is lower bounded by g, so as the input, n, approaches ∞, $g(n)$ is smaller than $f(n)$. In terms of limits this is $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
- If $f(n) = \Theta(g(n))$, then we know that f is upper and lower bounded by g, so as the input, n, approaches ∞, $g(n)$ is larger than $f(n)$. In terms of limits this is $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, where c is some constant.

Finding the Asymptotic Bound
There are two main rules when finding the asymptotic bound of a function.

1. Keep dominant term and drop all lesser terms
2. Drop all constants in front of dominant term

For example, if we have $f(x) = 500x^3 + 3x^2 - 5x + 9$, then we drop $3x^2 - 5x + 9$ because they are all lesser terms and also drop the 500 in front of x^3 since it is a constant, leaving us with $f(x) = \Theta(x^3)$. The reason why we can do this can be shown with limits. If we were to find the $\lim_{x\to\infty} \frac{500x^3+3x^2-5x+9}{x^3}$, we also drop lesser terms and constants, arriving at 500.

Calculating Runtime

Normal Functions

Conceptually, the runtime of a function is how long it takes to execute, so simply summing all the actions taken during the function call is sufficient to find the runtime. This means that if the function calls other functions, knowing the runtimes of those functions is crucial to calculate the runtime of the given function. Aside from knowing the runtimes of functions called within the given function, knowing how to analyze loops is also important.

Single Loops

Recall that the runtime is all the "work" done done by a function or operation. For single loops, this often times is simply how much work is done per iteration * the number of iterations. This is because the work done per iteration is usually the same. If it is different per iteration, then you will have to resort to using a summation to help you find the runtime.

```
public void f1(int n) {
    for (int i = 0; i < n; i+=1) {
        System.out.println("it be f1");
    }
}
public void f2(int n) {
    for (int i = 1; i < n; i++) {
        g(i) // where g(n) is Theta(n);
    }
}
```

For f1, each iteration of the loop runs in $\Theta(1)$, so their runtimes are simply how many iterations of the loop there are, resulting in a runtime of $\Theta(n)$. In f2, since each iteration of the loop does different work, we need to come up with a summation to solve. We see that $g(n)$ runs in linear time, so the summation becomes $1+2+3+4+...+n$ or $\sum_{i=1}^{n} i$. Solving this summation results in $\Theta(n^2)$ (explanation in *Important Summations*).

Multiple Loops

With multiple loops, there are two main forms this manifests in. Loops could be independent of the other loops (easier) or loops could be dependent on other loops (harder).

```
public void f3(int n) {
    for (int i = 0; i < n; i+=1) {
        for (int j = 0; j < n; j+=1) {
            for (int k = 0; k < n; k+=1) {
                System.out.println("Independent Loops");
            }
        }
    }
}
```

If you examine these three nested loops, you will see that each loop is independent of the others (the variable counter only appears in the loop that it is counting for). The runtime of these nested loops, is simply the product of the number of iterations of each loop. In this case it would be $n * n * n = n^3 \implies \Theta(n^3)$. You can only do this when the loops are **INDEPENDENT** of each other.

For loops that are dependent on each other, you will more often than not require a summation to solve. Consider these nested loops.

```
public void f4(int n ) {
    for (int i = 0; i < n; i+=1) {
        for (int j = i; j < n; j+=1) {
            System.out.println("Dependent Loops");
        }
    }
}
```

These loops are **NOT** independent of each other because of the line *int j = i*. The number of times the inner loop runs depends on what iteration the outer loop is on. For these loops, I like to look at each iteration of the outer loop and see how much work is done by the inner loop and sum that. There will usually be a pattern that you can spot after 3 or 4 iterations.

Outer Loop Iteration	Work of Inner Loop
1	n
2	n - 1
3	n - 2
⋮	⋮
n - 1	2
n	1

The runtime of f4 is then the sum of the right column, $n + (n-1) + (n-2) + ... + 2 + 1$. Reversing the order of this summation results in $1 + 2 + 3 + ... + n$, which we've seen is $\Theta(n^2)$, so the runtime of f4 is $\Theta(n^2)$.

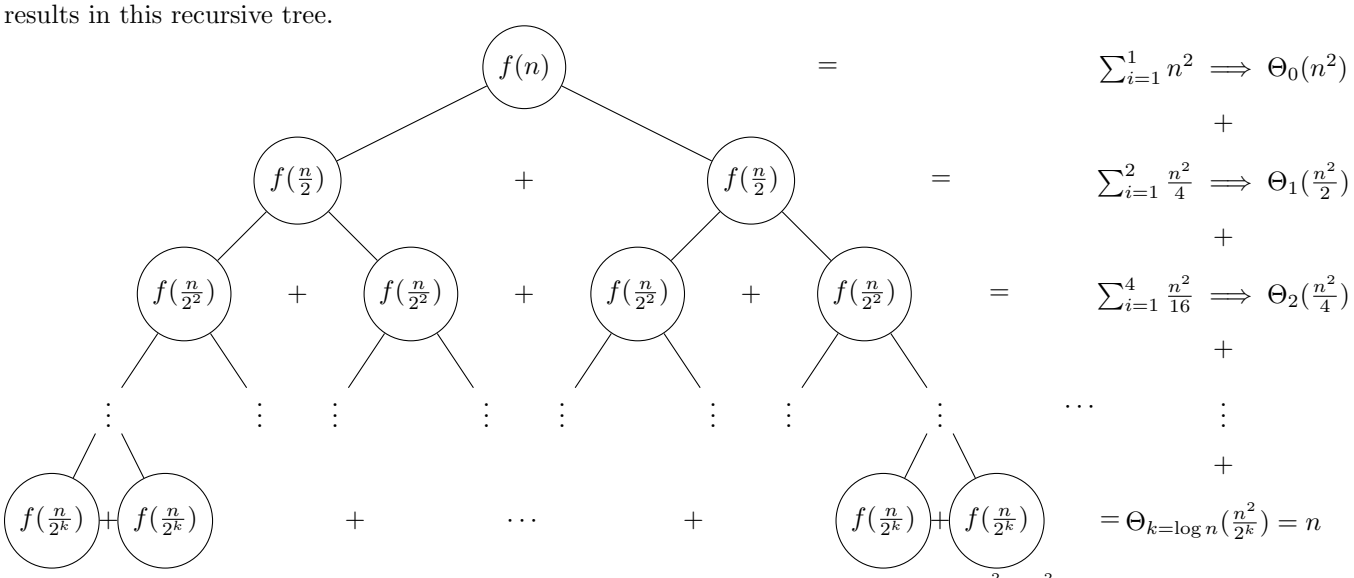
Normal Functions Cont.
If there are multiple groups of loops or operations in a function, the runtime is simply the sum of all the parts. Conceptually each of these loops and operations will execute and contribute to the work done by the function. If there are conditionals, you may need to express the runtime with O and Ω because executing the conditional will cause the function to do more work, whereas if it is skipped the function does less work. If executing the conditional, just add the work done inside of the conditional, whereas if not executing the conditional, don't add that amount of work.

Recursive Functions
When solving recursive functions, there are two main steps: solving for the nonrecursive section (normal section) and analyzing the recursive structure. Solving for the nonrecursive section is simply what we did in the earlier. This will tell us the runtime of each function call **with respect to the input**. Consider this function

```
public int f(int n) {
    if (n == 1) {
        return n;
    }
    for (int i = 0; i < n; i+=1) {
        for (int j = i; j < n; j+=1) {
            System.out.println("Dependent Loops");
        }
    }
    return f(n/2) + f(n/2);
}
```

The nested for loops is what we saw earlier to be $\Theta(n^2)$. This means that if the input to the function is n, then the runtime is n^2, but if the input to the function is $\frac{n}{2}$, then the runtime is $(\frac{n}{2})^2 = \frac{n^2}{4}$. This is what it

means to be $\Theta(n^2)$ with respect to the input. After we've solved the nonrecursive section, it's time to put together the recursive tree. We see the branching factor is 2 and the input is half of the original input. This results in this recursive tree.



The runtime of f is then the sum of the entire right column. This turns out to be $n^2+\frac{n^2}{2}+\frac{n^2}{4}+...+4n+2n+n$. We can reverse the order to arrive at $n+2n+4n+...+\frac{n^2}{4}+\frac{n^2}{2}+n^2 = n(1+2+4+...+\frac{n}{4}+\frac{n}{2}+n) \implies \Theta(n^2)$ (explanation in *Important Summations*). If each level of the recursive tree does the same amount of work, then we can simply do work per level * number of levels to calculate the runtime.

Important Summations

1. $1+2+3+4+...+n = \sum_{i=1}^{n} i \implies \Theta(n^2)$, where n is the last term
   The $\Theta(n^2)$ comes from the mathematical formula for this arithmetic sequence, which is $\frac{n(n+1)}{2}$, where n is the value of the last term. Conceptually, this is adding the first an last terms to get $n+1$, the second term and second to last term to also get $n+1$, and so on. There are then $\frac{n}{2}$ $(n+1)$ terms since two terms combine to make one $n+1$ term, resulting in $\frac{n(n+1)}{2} \implies \Theta(n^2)$.
   Another way of thinking about it through visual. If you stack blocks each of height equal to that term's value, you get a triangle, with length n for n terms and height of n since the last term is of value n. The area of this triangle is approximately the summation, which turns out to be $\frac{n^2}{2} \implies \Theta(n^2)$.

2. $1+2+4+8+...+n = \sum_{i=0}^{\log n} 2^i \implies \Theta(n)$, where n is the last term
   The $\Theta(n^2)$ comes from the mathematical formula for this geometric sum, which is $\frac{a_1}{1-r}$, where a_1 is the first term of the sequence and r is the ratio between terms. Rearranging the current form of the sequence, we can arrive at $n(\frac{1}{n}+\frac{2}{n}+\frac{4}{n}+...+\frac{1}{4}+\frac{1}{2}+1)$. Then taking the reverse order, we have $n(1+\frac{1}{2}+\frac{1}{4}+...+\frac{2}{n}+\frac{1}{n})$. We see that this is a geometric sum * n. From the formula, we get that the geometric series is $\approx \frac{1}{1-\frac{1}{2}} = 2$, so the overall result is $\approx 2n \implies \Theta(n)$.
   Another way of thinking about this is through a pattern. We see that 1 is 1 less than the next term, 2. Next, $1+2=3$, which is 1 less than the next term, 4. $1+2+4=7$, which is 1 less than the next term, 8. From this we see that the sum of all the terms will always be 1 less than the next term. This results in $(n-1)+n = 2n-1 \implies \Theta(n)$.

TL;DR

Big O (Upper Bound): $f(n) = O(g(n))$ if $0 < f(n) < cg(n)$ for all $n \geq k$
Big Omega (Lower Bound): $f(n) = \Omega(g(n))$ if $0 < cg(n) < f(n)$ for all $n \geq k$
Big Theta: $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
Limits: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \implies f(n) = O(g(n)) \\ c \implies f(n) = \Theta(g(n)) \\ \infty \implies f(n) = \Omega(g(n)) \end{cases}$
Finding Asymptotic Bound:

1. Drop lesser terms
2. Drop constants

Calculating Runtime:

1. Find runtime of nonrecursive part
2. If function not recursive then done, else find branching factor and how input changes in recursive calls
3. Analyze recursive tree
4. Solve resulting summation
5. ***Important Notes**
   (a) If **independent** or work done per level/iteration is the same, **short cuts can be applied**
   (b) If **dependent** or work done per leve/iteration is **not** the same, **use summations**

Summations:

- $1 + 2 + 3 + 4 + ... + n = \sum_{i=1}^{n} i \implies \Theta(n^2)$
- $1 + 2 + 4 + 8 + ... + n = \sum_{i=0}^{\log n} 2^i \implies \Theta(n)$
- ***NOTE: the n in the final bounds correspond to the last term of the summation, so adjust the bounds accordingly**

Geometric Series: $\sum_{i=0}^{n} r^i \begin{cases} O(1) & r \leq 1 \\ O(n) & r = 1 \\ O(r^n) & r \geq 1 \end{cases}$