Discussion 8: More Asymptotics

Administrivia

- Project 2 Phase 2 due March 5th
- Labs this week will be Project 2 checkoff
    - Sign up here!
- HWs 2 and 3 upcoming due 3/14 and 3/19
- Midterm 2 far in the future 3/20

Notation: Big O, Big Omega, Big Theta

- Goal: Look at program complexity for large input
- Notations:
  - Big O - bounds above
  - Big Omega - bounds below
  - Big Theta - bounds above and below

O (Big O)

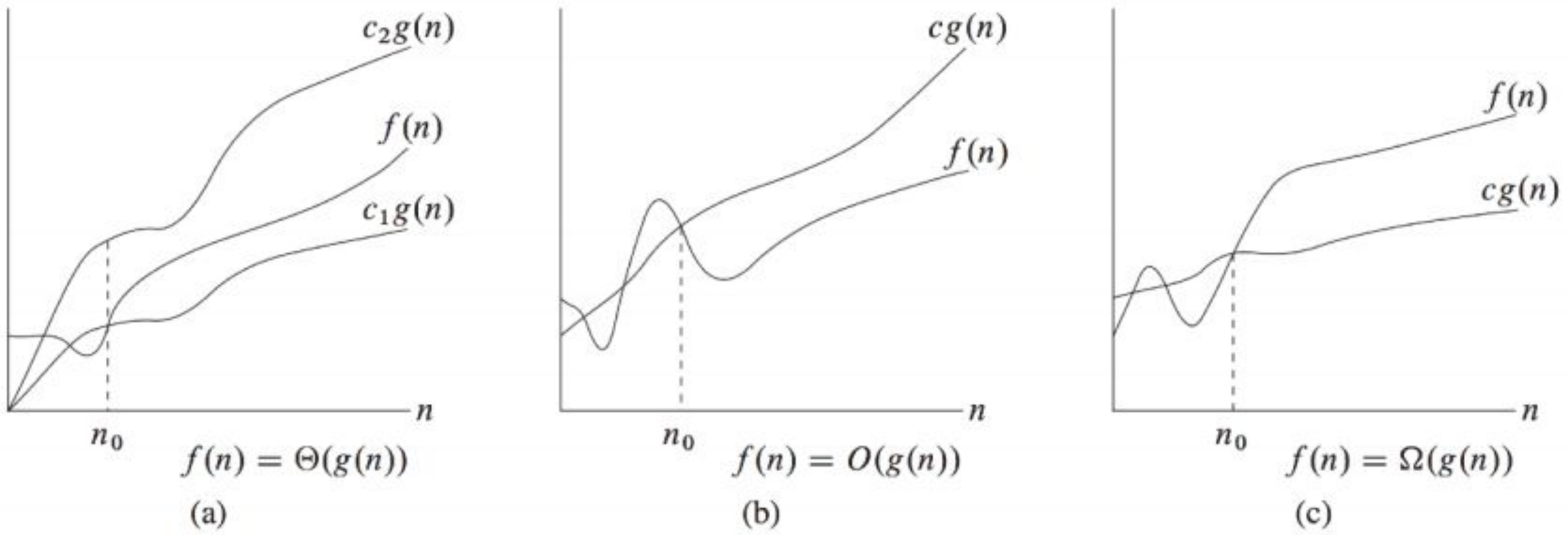
- Let $f(n)$ and $g(n)$ be positive real numbers on inputs of size n
- $f \in O(g)$ if there is a constant $c > 0$ s.t. $f(n) <= c\, g(n)$
- Upper bounded by $g(n)$ when n gets significantly large.
- Bound does not have to be tight.

Ω (Big Omega)

- Let f(n) and g(n) be positive real numbers on inputs of size n
- $f \in \Omega(g)$ if there is a constant $c > 0$ s.t. $f(n) >= c\ g(n)$
- Lower bounded by g(n) when n gets significantly large.
- Bound does not have to be tight.

Θ (Big Theta)

- Let f(n) and g(n) be positive real numbers on inputs of size n
- $f \in \Theta(g)$ if there is a constant c1 > 0 and c2 > 0 s.t.
    - C1 g(n) <= f(n) <= c2 g(n) for all c1 <= c2
- Tightly bounded by g(n) when n gets significantly large.
- • $f \in \Omega(g)$ and $f \in O(g)$



Introduction to Algorithms, Cormen, Leiserson, Rivest, Stein

Conventions: No Constants

- Drop multiplicative constants and lower order terms
- Any exponential dominates any polynomial
- Any polynomial dominates any logarithm

Common Asymptotic Sets

- O(1): constant
- O(log n): logarithmic
- O(sqrt(n)): square root
- O(n): linear
- O(n log n): linearithmic
- O(n^2): quadratic
- O(n^3): cubic
- O(2^n): exponential
- O(n!): factorial

Analyzing Runtime

```
public int[] bogosort(int[] ints) {
   while (!isSorted(ints)) {
       Collections.shuffle(ints);
   }
   return ints;
}
```

 # Let's find out the big-O runtime of this wonderful sorting algorithm!

Analyzing Runtime Example

```
public int[] bogosort(int[] ints) {
   while (!isSorted(ints)) {
       Collections.shuffle(ints);
   }
   return ints;
}
```

 # Let’s find out the big-O runtime of this wonderful sorting algorithm!
 # We enter a while loop until the list is sorted, let’s assume that isSorted runs in O(n)

Analyzing Runtime Example

```
public int[] bogosort(int[] ints) {
   while (!isSorted(ints)) {
       Collections.shuffle(ints);
   }
   return ints;
}
```

Let’s find out the big-O runtime of this wonderful sorting algorithm!
We enter a while loop until the list is sorted, let’s assume that isSorted runs in O(n)
At each iteration, we shuffle the ints. Let’s assume this can also be done in O(n)
How many times will we run this while loop if we keep shuffling?

Analyzing Runtime Example

```
public int[] bogosort(int[] ints) {
   while (!isSorted(ints)) {
       Collections.shuffle(ints);
   }
   return ints;
}
```

Let's find out the big-O runtime of this wonderful sorting algorithm!
We enter a while loop until the list is sorted, let's assume that isSorted runs in O(n)
At each iteration, we shuffle the ints. Let's assume this can also be done in O(n)
Since there are n! possible arrangements of ints, we have a 1/n! chance to correctly sort on a shuffle.

Analyzing Runtime Example

```
public int[] bogosort(int[] ints) {
    while (!isSorted(ints)) {
        Collections.shuffle(ints);
    }
    return ints;
}
```

Let's find out the big-O runtime of this wonderful sorting algorithm!
We enter a while loop until the list is sorted, let's assume that isSorted runs in O(n)
At each iteration, we shuffle the ints. Let's assume this can also be done in O(n)
Since there are n! possible arrangements of ints, we have a 1/n! chance to correctly sort on a shuffle.
The while loop dominates the runtime, taking n time to shuffle and n! potential iterations in expection.
O(n * n!) in expectation.
Unbounded in the worst case.

Techniques for Analyzing Runtime

- Annotate your code
  - Write down runtimes of called functions
- Consider end conditions for loops and consequently the number of times we could run the loop
  - What are the terminating conditions for a loop?
  - Can we algebraically express the runtime of the loop?
- Branching factor
  - Do we make additional calls and if so, how many subproblems do we make?

Analyzing Runtime Example 2

```
public void doStuff(int n) {# Let’s find out the big-O runtime of this algorithm!
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

Analyzing Runtime Example 2

```
public void doStuff(int n) {
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

 # Let's find out the big-O runtime of this algorithm!
 # We initialize i = 2 and run the loop until i <= n

Analyzing Runtime Example 2

```
public void doStuff(int n) {
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

 # Let’s find out the big-O runtime of this algorithm!
 # We initialize i = 2 and run the loop until i <= n
 # At each iteration of the while loop, we square i
 # i = i^2

Analyzing Runtime Example 2

```
public void doStuff(int n) {
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

 # Let's find out the big-O runtime of this algorithm!
 # We initialize i = 2 and run the loop until i <= n
 # At each iteration of the while loop, we square i
 # i = i^2
 # Let's try to generalize this. After a lot of iterations, we get something that looks like this: i^2^2^2^2…
 # How can we write this algebraically? .

Analyzing Runtime Example 2

```
public void doStuff(int n) {
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

Let's find out the big-O runtime of this algorithm!
We initialize i = 2 and run the loop until i <= n
At each iteration of the while loop, we square i
i = i^2
Let's try to generalize this. After a lot of iterations, we get something that looks like this: i^2^2^2^2...
i^(2^2^2^2...) = i^(2^k) <= n where k is our number of iterations

Analyzing Runtime Example 2

```
public void doStuff(int n) {
    int i = 2;
    while (i <= n) {
        i = Math.pow(i, 2);
    }
}
```

Let's find out the big-O runtime of this algorithm!
We initialize i = 2 and run the loop until i <= n
At each iteration of the while loop, we square i
i = i^2
Let's try to generalize this. After a lot of iterations, we get something that looks like this: i^2^2^2^2…
i^(2^2^2^2…) = i^2^k <= n where k is our number of iterations and we know i starts at 2
Let's solve for k: log(log(2^2^k) <= log(log(n))
k <= log(log(n)) -> O(log(log(n)))

Problem 1.1

```
public void andslam(int N) {
    if (N > 0) {
        for (int i = 0; i < N; i += 1) {
            System.out.println("datboi.jpg");
        }
        andslam(N / 2);
    }
}
```

Problem 1.1

- $\Theta(N)$
- Log n levels of work and each level is O(n)

Problem 1.2

```java
public static void andwelcome(int[] arr, int low, int high) {
    System.out.print("[ ");
    for (int i = low; i < high; i += 1) {
         System.out.print("loyal  ");
    }
    System.out.println("]");
    if (high - low > 0) {
        double coin = Math.random();
        if (coin > 0.5) {
            andwelcome(arr, low, low + (high - low) / 2);
        } else {
            andwelcome(arr, low, low + (high - low) / 2);
            andwelcome(arr, low + (high - low) / 2, high);
        }
    }
}
```

Problem 1.2

- $\Theta(N \log N)$
- Log n levels of work, but have a branching factor of 2 and each level is O(n) in the worst case

Problem 1.3

```
public int tothe(int N) {
    if (N <= 1) {
        return N;
    }
    return tothe(N - 1) + tothe(N - 1);
}
```

Problem 1.3

- $\Theta(2^N)$
- 2^i nodes per layer and each node does $O(1)$ work

Problem 1.4

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

Problem 1.4

- O(N * N!)
- n-i work per node and n! / (n-i)! nodes per layer.

Problem 2.1

- Don’t forget your definition for Ω , Θ, and O!

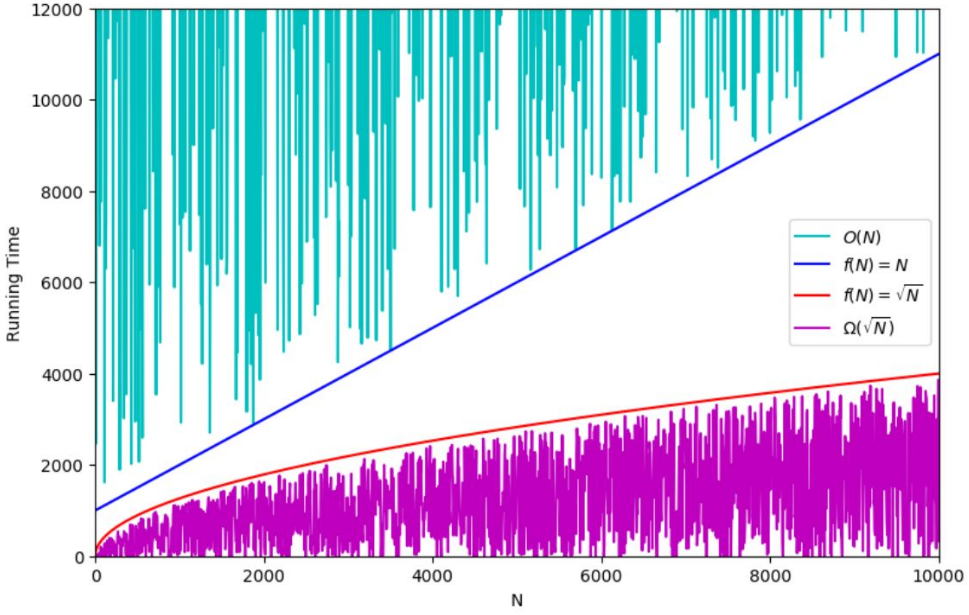
Problem 2.1

- Algorithm 1
- Neither
- Neither
- Algorithm 2
- Neither

Problem 3.1

Draw the running time graph of an algorithm that is $O(\sqrt{N})$ in the best case and $\Omega(N)$ in the worst case. Assume that the algorithm is also trivially $\Omega(1)$ in the best case and $O(\infty)$ in the worst case.

Problem 3.1



12000
10000
8000
6000
4000
2000
0
Running Time
0
2000
4000
6000
8000
10000
N
$O(N)$
$f(N) = N$
$f(N) = \sqrt{N}$
$\Omega(\sqrt{N})$

Problem 3.2: True or False?

$f(n) = 20501$	$g(n) = 1$	$f(n) \in O(g(n))$
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n$	$g(n) = (\log n)^2$	$f(n) \in O(g(n))$

Problem 3.2: True or False?

- True
- False
- True
- False
- True
- True
- False