Lecture 3: How to measure efficiency

Data Structures and Algorithms

Announcements

- Course background survey due by Friday
- HW 1 is Due Friday
- Alex has Office Hours after class (2:30-4:30) CSE 006, will help with setup

  - If you have any questions about your setup please come to office hours so we can iron out all the wrinkles before the partnered projects begin next week.

- HW 2 Assigned on Friday – Partner selection forms due by 11:59pm **Thursday**

https://goo.gl/forms/rVrVUkFDdsqI8pkD2

Review: Sequential Search

sequential search: Locates a target value in an array / list by examining each element from start to finish.

- How many elements will it need to examine?

- Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

i

- What is the best case? O(1) if we are looking for first element
- What is the worst case? O(n) if we are looking for last element
- What is the complexity class? O(n) ALWAYS based on pesimistic case.

Review: Binary Search

binary search: Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

- How many elements will it need to examine?
- Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

min (index 0) mid (index 8) max (index 16)

- What is the best case? O(1) - the middle
- What is the worst case? The beginning
- What is the complexity class? log(n)

Analyzing Binary Search

What is the pattern?

- At each iteration, we eliminate half of the remaining elements

How long does it take to finish?

- 1st iteration – N/2 elements remain
- 2nd iteration – N/4 elements remain
- Kth iteration - N/2^k elements remain
- Done when N/2^k = 1

index	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

Analyzing Binary Search

$$\frac{N}{2^K} = 1$$

$$N = 2^K$$

$$\log_2 N = \log_2 2^K$$

$$\boxed{\log_2 N = K}$$

$O(\log n)$

↑ we lost our base!

Logarithms

$\log_b a = x$ mean

x solves

$b^? = a$

$\log_b b^z = x$

$b^? = b^z \rightarrow z$

Analyzing Binary Search

Finishes when $N / 2^K = 1$

$N / 2^K = 1$

-> multiply right side by 2^K

$N = 2^K$

-> isolate K exponent with logarithm

$Log_2 N = k$

Is this exact?

- N can be things other than powers of 2
- If N is odd we can't technically use Log_2
- When we have an odd number of elements we select the larger half
- Within a fair rounding error

Asymptotic Analysis

asymptotic analysis: how the runtime of an algorithm grows as the data set grows

Approximations / Rules

- Basic operations take "constant" time
  - Assigning a variable
  - Accessing a field or array index

adding, subtracting, print out a string, etc…

- Consecutive statements
  - Sum of time for each statement

dont consider time it takes moving between lines

- Function calls
  - Time of function's body

dont consider time it takes to call function

- Conditionals
  - Time of condition + maximum time of branch code

fits with our pesimistic viewpoint

- Loops
  - Number of iterations x time for loop body

if

else

← longest

Modeling Case Study

Goal: return ‘true’ if a sorted array of ints contains duplicates

Solution 1: compare each pair of elements

```
public boolean hasDuplicate1(int[] array) {
   for (int i = 0; i < array.length; i++) {
      for (int j = 0; j < array.length; j++) {
         if (i != j && array[i] == array[j]) {
            return true;
         }
      }
   }
   return false;
}
```

Solution 2: compare each consecutive pair of elements

```
public boolean hasDuplicate2(int[] array) {
   for (int i = 0; i < array.length - 1; i++) {
      if (array[i] == array[i + 1]) {
            return true;
      }
   }
   return false;
}
```

Modeling Case Study: Solution 2

$\boxed{T(n)}$ where n = array.length

-> work inside out

Solution 2: compare each consecutive pair of elements

```
public boolean hasDuplicate2(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        if (array[i] == array[i + 1]) { +4
            return true; +1
        }
    }
    return false; +1
}
```

If statement = 5

x n - 1

4 operations: add i + 1; access array[i+1]; access array[i]; check equality

Assume WORST case; i.e. enters if statement every single time

T(n) = 5 (n-1) + 1

linear time complexity class O(n)

Modeling Case Study: Solution 1

Solution 1: compare each consecutive pair of elements

```
public boolean hasDuplicate1(int[] array) {
    for (int i = 0; i < array.length; i++) { x n
        for (int j = 0; j < array.length; j++) { x n
            if (i != j && array[i] == array[j]) { +5
                return true; +1
            }
        }
    }
    return false; +1
}
```

6

6n

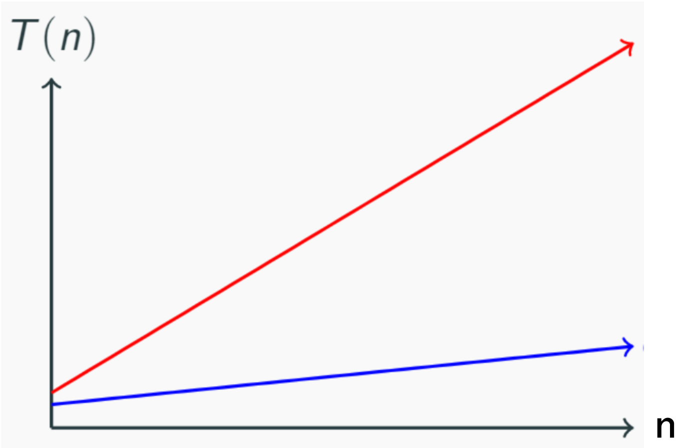
$6n^2$

$T(n) = 6\,n^2 + 1$

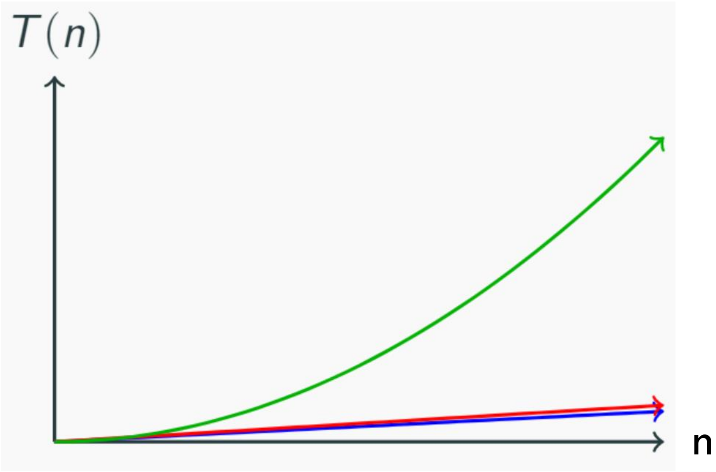
quadratic time complexity class $O(n^2)$

Comparing Functions

Function growth



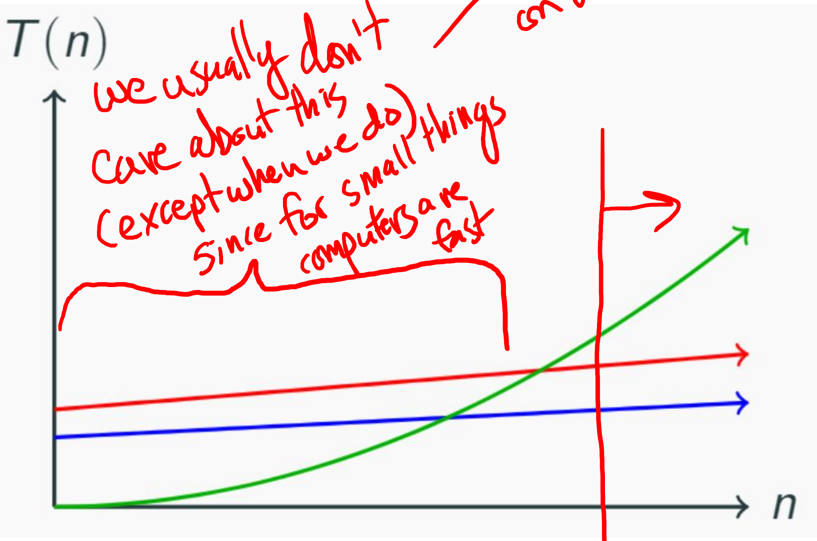
n and 4n look very different up close



n and 4n look the same over time

n^2 eventually dominates n

When do we care? Lots of runs on small data!



n^2 doesn't start off dominating the linear functions

It eventually takes over…

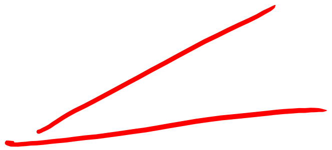
Function comparison: exercise

$O(n) = O(5n+3)$ $\quad$ $O(n) \leq O(5n+3)$

$f(n) = n \leq g(n) = 5n + 3$? **True** – all linear functions are treated as equivalent

$f(n) = 5n + 3 \leq g(n) = n$? **True**

$f(n) = 5n + 3 \leq g(n) = 1$? **False** $\quad$ $O(5n - 1000000) \geq 1$



$f(n) = 5n + 3 \leq g(n) = n^2$? **True** – quadratic will always dominate linear

$f(n) = n^2 + 3n + 2 \leq g(n) = n^3$? **True**

$f(n) = n^3 \leq g(n) = n^2 + 3n + 2$? **False**

In this case, its not a true <= sign; it refering to whether one function dominates the other (do they have different big O run times?)

Definition: function domination

Definition: Domination

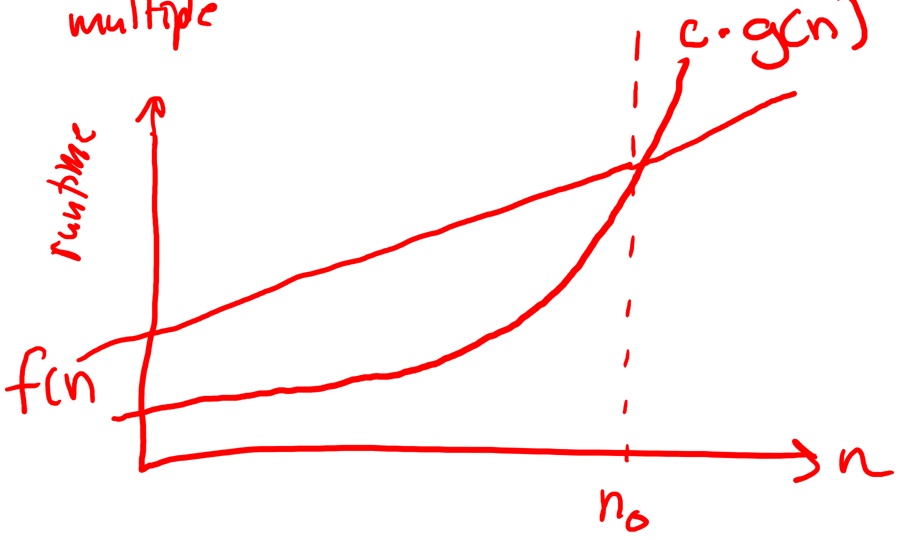
A function f(n) is **dominated** by g(n) when…

There exists two constants c > 0 and $\boxed{n_0 > 0}$

Such that for all values of $n \geq n_0$

$f(n) \leq c * g(n)$

i.e. g(n) is eventually always bigger than f(n), up to a constant multiple



Example:

Is f(n) = n dominated by g(n) = 5n + 3 ?

c = 1

$n_0 = 1$

Yes!

$2n = n+3$

$n_0 = 10$

$f(n) = n$

$g(n) = n+3$

$n \geq 1$

$c = 1$

$c \cdot g(n) = n+3 \geq n \ \forall\, n > n_0 = 1$

$c \cdot f(n) = 2n \geq n+3 \ \forall\, n > 10$

$c = 2$

Exercise: Function Domination

Demonstrate that $5n^2 + 3n + 6$ is dominated by n^3 by finding a c and n_0 that satisfy the definition of domination

n=1 LHS $5+3+6 = 14$

$5+3+6 = 14$

$\boxed{5n^2 + 3n + 6} \le \boxed{5n^2 + 3n^2 + 6n^2}$ when $n \ge 1$

$5n^2 + 3n^2 + 6n^2 = \boxed{14n^2}$

$5n^2 + 3n + 6 \le 14n^2$ for $n \ge 1$

$14n^2 \le c*n^3$ for c = ? n >= ?

$\frac{14}{n}$ -> c = 14 & n >= 1

Definition: Big O

If f(n) = n ≤ g(n) = 5n + 3 ≤ h(n) = 100n and

h(n) = 100n ≤ g(n) = 5n + 3 ≤ f(n) n

Really they are all the "same"

Definition: Big O

O(f(n)) is the "family" or "set" of all functions that are dominated by f(n)

Text

all functions that are ~ as fast or faster than f(n) runtime

Question: are O(n), O(5n + 3) and O(100n) all the same?

True! By convention we pick simplest of the above -> O(n) ie "linear"

f(n) = 1 in O(n) ?

Definitions: Big Ω

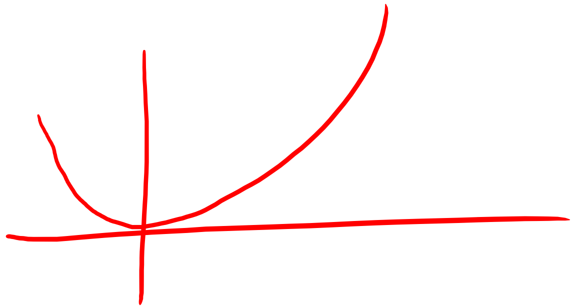
"f(n) is greater than or equal to g(n)"

F(n) dominates g(n) when:

There exists two constants such that c > 0 and n0 > 0

Such that for all values n >= n0

F(n) >= c * g(n) is true



n^2

Definition: Big Ω

Ω(f(n)) is the family of all functions that dominates f(n)

all functions that are ~equal or larger than f(n) (i.e. runs slower)

$\Omega(n^2)$

$\rightarrow n^3$

$\rightarrow n^5$

$\rightarrow n!$

$\boxed{\in}$ Element Of

f(n) is dominated by g(n)

Is that the same as

"f(n) is contained inside O(g(n))"

Yes!

f(n) ∈ g(n)

$n = O(n)$

$n \le O(n)$

$n \in O(n)$

Examples

$4n^2 \in \Omega(1)$

true

$4n^2 \in \Omega(n)$

true

$4n^2 \in \Omega(n^2)$

true

$4n^2 \in \Omega(n^3)$

false

$4n^2 \in \Omega(n^4)$

false

$4n^2 \in O(1)$

false

$4n^2 \in O(n)$

false

$4n^2 \in O(n^2)$

true

$4n^2 \in O(n^3)$

true

$4n^2 \in O(n^4)$

true

Definition: Big O

O(f(n)) is the "family" or "set" of all functions that are dominated by f(n)

Definition: Big Ω

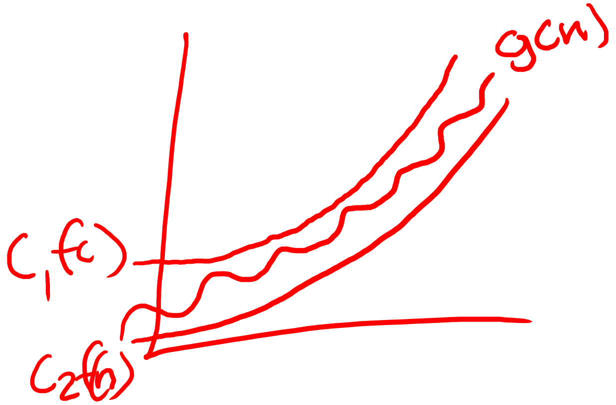
Ω(f(n)) is the family of all functions that dominates f(n)

Definitions: Big Θ

We say f(n) ∈ Θ(g(n)) when both

f(n) ∈ O(g(n)) and f(n) ∈ Ω (g(n)) are true

Which is only when f(n) = g(n)



Definition: Big Θ
Θ(f(n)) is the family of functions that are equivalent to f(n)

approximately same runtime; all same order of magnitude (i.e. 4n^2 is in Big O- of n^2)

Industry uses "Big Θ" and "Big O" interchangeably

Summary

$O(f(n)) \leq f(n) == \Theta(f(n)) \leq \Omega(f(n))$

	Dominated by $f(n) \in O(g(n))$	Dominates $f(n) \in \Omega(g(n))$
$f(n)$		
$O(1)$		
$O(\log n)$		
$O(n)$		
$O(n^2)$		
$O(n^3)$		

Justifying the "Rules"

Approximations / Rules

- Basic operations take "constant" time
  - Assigning a variable
  - Accessing a field or array index
- Consecutive statements
  - Sum of time for each statement
- Function calls
  - Time of function's body
- Conditionals
  - Time of condition + maximum time of branch code
- Loops
  - Number of iterations x time for loop body

c_1

c_2

c_3

$\vdots$

c_n

$c_1 + c_2 + \dots + c_n \leq C \cdot n$

$\max_n c_n$

+1 per line

$c + 1 \leq c'$

A Slightly Harder example

```
public void mystery(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n * n; j++) {
            System.out.println("Hello");
        }
        for (int j = 0; j < 10; j++) {
            System.out.println("world");
        }
    }
}
```

+1 n^2 $O(n^2)$

+1   10   n

no n! $10 \Rightarrow O(1)$

$O(n^2) + O(1)$
$= O(n^2 + 1)$
$= O(n^2)$ $\times n$

$n \cdot O(n^2) = O(n^3)$

Remember: work outside in

Solution: $T(n) = n(n^2 + 10) = n^3 + 10n$

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```

~~+1 n n $f(n) = n^2$~~

Keep an eye on loop bounds!

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!"); +c
    }
}
```

+c → 0 + 1 + 2 + 3 +…+ n-1 → n

Summation

$1 + 2 + 3 + 4 + \ldots + n = \sum_{i=1}^{n} i$

Definition: Summation
$\sum_{i=a}^{b} f(i)$ = f(a) + f(a + 1) + f(a + 2) + … + f(b-2) + f(b-1) + f(b)

$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$

Simplifying Summations

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
      System.out.println("Hello!");
   }
}
```

$$\underbrace{\underbrace{+c}_{\text{inner loop}} \;\to\; 0 + 1 + 2 + 3 + \ldots + n\text{-}1}_{} \;\to\; n$$

+c

0 + 1 + 2 + 3 +…+ n-1

n

(0c + 1c + 2c + 3c + … + i-1c)
+ (0c + 1c + 2c + 3c + … + i-1c)
+ (0c + 1c + 2c + 3c + … + i-1c)
+ repeat n times

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} c = \sum_{i=0}^{n-1} ci$$ Summation of a constant

$$= c\sum_{i=0}^{n-1} i$$ Factoring out a constant

$$= c\frac{n(n-1)}{2}$$ Gauss's Identity

$$= \frac{c}{2}n^2 - \frac{c}{2}n$$ $O(n^2)$

Function Modeling: Recursion

```
public int factorial(int n) {
   if (n == 0 || n == 1) {  +3
      return 1; +1                +c
   } else {
      return n * factorial(n - 1);  +????
   }
}
```

Function Modeling: Recursion

```
public int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
}
```

$+c_1$

$+T(n-1)$

$+c_2$

$$T(n) = \begin{cases} C_1 & \text{when } n = 0 \text{ or } 1 \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

Definition: Recurrence

Mathematical equivalent of an if/else statement
f(n) =

Unfolding Method

$$T(n) = \begin{cases} C_1 & \text{when } n = 0 \text{ or } 1 \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

$$T(3) = C_2 + T(3-1) = C_2 + (C_2 + T(2-1)) = C_2 + (C_2 + (C_1)) = 2C_2 + C_1$$

$$T(n) = C_1 + \sum_{i=0}^{n-1} C_2$$

Summation of a constant

$$T(n) = C_1 + (n-1)C_2$$