Programming Abstractions

CS106B

Cynthia Lee

Topics du Jour:

- Last time:
  - Performance of Fibonacci recursive code
  - Look at growth of various functions
    - Traveling Salesperson problem
    - Problem sizes up to number of Facebook accounts
  - Formal mathematical definition
- **This time: Big-O performance analysis**
  - Simplifying Big-O expressions
  - Analyzing algorithms/code
    - Just a bit for now, but we'll be applying this to all our algorithms as we encounter them from now on
- **Head start on Wednesday's topic: make your own classes!**
  - Needed for Boggle assignment, we are starting to see a little bit in MarbleBoard assignment as well.

Translating code to a f(n) model of the performance

(n = size of vector)

	Statements	Cost
1	double findAvg (Vector<int>& grades){	
2	double sum = 0;	1
3	int count = 0;	1
4	while (count < grades.size()) {	$n + 1$
5	sum += grades[count];	n
6	count++;	n
7	}	
	)	1
	.size();	
		1
11	return 0.0;	
12	}	
ALL		**3n+5**

Do we really care about the +5?
Or the 3 for that matter?

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
10	**1,024**	10,240 (.000003s)	1,048,576 (.0003s)	1.80×10^{308}
30	**1,300,000,000**	39000000000 (13s)	1690000000000000000 (18 years)	$2.3 \times 10^{391,338,994}$

# of Facebook accounts

Big-O

We say a function f(n) is **"big-O"** of another function g(n), and write "f(n) is **O**(g(n))" iff there exist positive constants c and n_0 such that:

$$f(n) \le c\, g(n) \textit{ for all } n \ge n_0.$$

What you need to know:

O(X) describes an "upper bound"—**the algorithm will perform <u>no worse</u> than X**

- We ignore constant factors in saying that
- We ignore behavior for "small" n

Image has been put in the public domain by its author.
http://commons.wikimedia.org/wiki/File:Kitten_(06)_by_Ron.jpg

Simplifying Big-O Expressions

- We always report Big-O analyses in simplified form and generally give the tightest bound we can
- Some examples:

Let f(n) = $\cancel{3}\log_2 n + \cancel{4}\,n\log_2 n + n$………..*f(n) is O(* $n\log n$ *).*

Let f(n) = $\cancel{546} + \cancel{34}n + \cancel{2}n^2$…………………..*f(n) is O(* n^2 *).*

Let f(n) = $2^n + \cancel{14}n^2 + \cancel{4}n^3$…………………..*f(n) is O(* 2^n *).*

Let f(n) = $\cancel{100}$……………………………..*f(n) is O(* 1 *).* "constant"

Big-O

Applying to algorithms

Applying Big-O to Algorithms

- Some code examples:

```
for (int i = data.size() - 1; i >= 0; i--){
    for (int j = 0; j < data.size(); j++){
        cout << data[i] << data[j] << endl;
    }
}
```

n

is O(n^2 *) where n is* `data.size()`*.*

Applying Big-O to Algorithms

- Some code examples:

```
for (int i = data.size() - 1; i >= 0; i -= 3){
    for (int j = 0; j < data.size(); j += 3){
        cout << data[i] << data[j] << endl;
    }
}
```

$\frac{1}{3}n$

$\frac{1}{3}n$

is O(n^2*) where n is* `data.size()`.

$\frac{1}{9}n^2$

Applying Big-O to Algorithms

- Some familiar examples:

Binary search………………….*is O(*$\log_2 n$*) where n is* size of array/vector.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

mn $\quad 9n^2$

Fauxtoshop edge detection...*is O(* *) where n is* .

$O(n^2)$

$O(n \cdot m)$ $\quad n$

R -1 C -1	R -1 C +0	R -1 C +1		
R +0 C -1	R +0 C +0	R +0 C +1		
R +1 C -1	R +1 C +0	R +1 C +1		

for (rows) $\quad n$
for (cols) $\quad n$
for $\quad 3$
for $\quad 3$

Applying Big-O to Algorithms

- Some code examples (assume data.size() >= 5):

```
for (int i = 0; i < data.size(); i += (data.size() / 5)) {
    cout << data[i] << endl;
}
```

is O(1 *) where n is* `data.size()`*.*

Big-O Extra Slides

Interpreting graphs using the formal definition

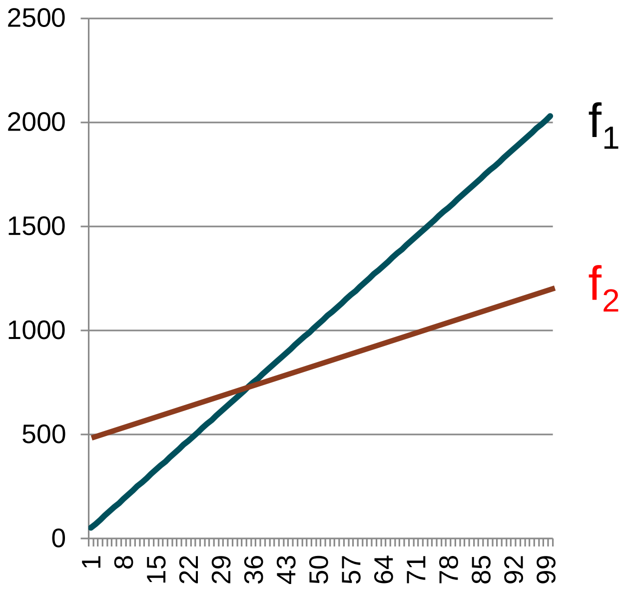
f_2 is $O(f_1)$

"f(n) is **O**(g(n))" iff

$$\exists c, n_0 > 0, s.t. \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

A. TRUE
B. FALSE

Why or why not?

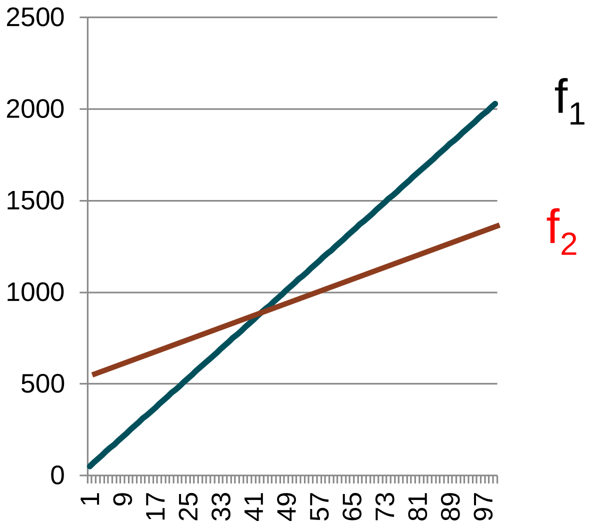


f(n) is O(g(n)), if there are positive constants c and n_0 such that $f(n) \leq$ **c** $* g(n)$ for all $n \geq n_0$.

Because we ignore the constant coefficient that determines slope, f1 and f2 look the "same" in Big-O analysis

f_2 is $O(f_1)$ ***and*** f_1 is $O(f_2)$

- Math version: *We can move f_2 above f_1 by multiplying by* **c** *(we can change the slope of f_2 by a constant factor)*



"f(n) is **O**(g(n))" iff

$$\exists c, n_0 > 0, s.t. \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

f_3 is $O(f_1)$

A. TRUE

B. FALSE

The constant c cannot rescue us here "because calculus."