CS61A Lecture 7
Complexity and Orders of Growth

Jon Kotker and Tom Magrino
UC Berkeley EECS
June 27, 2012

Computer Science in the News

Bot With Boyish Personality Wins Biggest Turing Test

Eugene Goostman, a chatbot with the personality of a 13-year-old boy, won the biggest Turing test ever staged, on 23 June.

Turing test: Measure of machine intelligence proposed by Alan Turing. A human talks via a text interface to either a bot or a human: the human has to determine which (s)he is talking to.

Turing suggested that if a machine could fool the human 30% of the time, it passed the test. Eugene passed 29% of the time.

Eugene was programmed to have a "consistent and specific" personality.

http://www.newscientist.com/blogs/onepercent/2012/06/bot-with-boyish-personality-wi.html

2

Today

- Time complexity of functions.
- Recursion review.

3

Problem Solving: Algorithms

An ***algorithm*** is a step-by-step description of how to perform a certain task.

For example, how do we bake a cake?

Step 1: Buy cake mix, eggs, water, and oil.

Step 2: Add the cake mix to a mixing bowl.

… and so on.

Image: http://pixelperfectmag.com/wp-content/uploads/2012/05/portal-cake.jpg

4

Problem Solving: Algorithms

The functions we write in Python *implement* algorithms for computational problems.

For a lot of problems, there are *many different* algorithms to find a solution.

How do we know which algorithm is *better*?

5

Comparison of Algorithms

How do we know which algorithm (and the function that implements it) is *better*?

- Amount of time taken. We will focus on this.
- Size of the code.
- Amount of non-code space used.
- Precision of the solution.
- Ease of implementation.

… among other metrics.

6

Comparison of Algorithms

Which function is better?
Which function takes lesser time?

ITERATIVE

```
def fib(n):
    if n == 0:
        return 0
    prev, curr, k = 0, 1, 1
    while k < n:
        prev, curr = curr, curr + prev
        k = k + 1
    return curr
```

VS

RECURSIVE

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Comparison of Algorithms

The *iterative* version of `fib` is quicker than the (naïve) *recursive* version of `fib`.

In module 2, we will see that we can make the two equally efficient.

Difference is only visible for *larger* inputs.

Idea:
Measure the *runtime* of a function for ***large*** inputs.
Computers are already quick for small inputs.

Runtime Analysis

How do we measure the ***runtime*** of a function?

Simplest way: Measure with a stopwatch.

Is this the *best* way?

http://fc00.deviantart.net/fs46/i/2009/187/0/a/Bill_Weasley___s_pocket_watch_by_Remus_Chocolade.jpg

Runtime Analysis

Measuring raw runtime depends on *many* factors:

- Different computers can have different runtimes.
- Same computer can have different runtimes on the same input.
  Other processes can be running at the same time.
- Algorithm needs to be implemented first!
  Can be tricky to get right.
- Function can take prohibitively long time to run.

Runtime Analysis

Problem:

How do we *abstract* the computer away?

Can we compare runtimes *without* implementing the algorithms?

Runtime Analysis: Big Data

Humans are producing a *lot* of data *really quickly.*

News

World's data will grow by 50X in next decade, IDC study predicts

IT execs will likely have trouble finding enough people with the skills and experience to manage it, analysts say

By Lucas Mearian
June 28, 2011 01:23 PM ET
10 Comments + Briefcase What's this?

Computerworld - In 2011 alone, 1.8 zettabytes (or 1.8 trillion gigabytes) of data will be created, the equivalent to every U.S. citizen writing 3 tweets per minute for 26,976 years. And over the next decade, the number of servers managing the world's data stores will grow by ten times.

Interestingly, the amount of data people create by writing email messages, taking photos, and downloading music and movies is minuscule compared to the amount of data being created about them, the EMC-sponsored study found.

http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts

Runtime Analysis

Big Idea:

Determine how the ***worst-case runtime*** of an algorithm ***scales*** as we scale the input.

The less the runtime scales as the input scales, the better the algorithm.

It can handle *more* data *quicker*.

Announcements

- Waitlist is cleared. If you're still on the waitlist by the end of this week, please let us know!
- Next week, we will move to **105 Stanley** for the rest of the summer.
- Midterm 1 is on **July 9**.
  - We will have a review session closer to the date.
- If you need accommodations for the midterm, please notify DSP by the end of this week.
- HW1 grade should be available on `glookup`.

Beware: Approximations Ahead

http://www.gamesdash.com/limg/1/283/beware-of-the-sign.jpg

Orders of Growth

```
def add_one(n):
    return n + 1

def mul_64(n):
    return n * 64

def square(n):
    return n * n
```

Time taken by these functions is roughly *independent* of the input size.

These functions run in ***constant time***.

Orders of Growth

```
def add_one(n):
    return n + 1

def mul_64(n):
    return n * 64

def square(n):
    return n * n
```

Approximation: Arithmetic operations and assignments take constant time.

Orders of Growth

```
def fact(n):
    k, prod = 1, 1
    while k <= n:
        prod = prod * k
        k = k + 1
    return prod
```

Constant-time operations

This loop runs n times.

Constant-time operations

Total time for all operations is proportional to n.

Orders of Growth

```
def fact(n):
    k, prod = 1, 1
    while k <= n:
        prod = prod * k
        k = k + 1
    return prod
```

Time taken by this function scales roughly *linearly* as the input size scales.

This function runs in ***linear time***.

Orders of Growth

```
def sum_facts(n):
    '''Adds factorials of
    integers from 1 to n.'''
    sum, k = 0, 1
    while k <= n:
        sum += fact(k)
        k = k + 1
    return sum
```

Constant-time operations

This loop runs n times.

For the kth loop, `fact` runs in time proportional to k.

Constant-time operations

Orders of Growth

Time taken by `sum_facts` is proportional to

Call to `fact` inside loop

Constant time operations per loop

Constant time operations outside loop

$$\underbrace{\underset{k=1}{1} + \underset{k=2}{2} + \dots + \underset{k=n}{n}}_{\text{Call to fact inside loop}} + an + b$$

$$= \frac{1}{2} n \cdot (n+1) + an + b$$

$$= \frac{1}{2} n^2 + (a + 1/2)n + b$$

Orders of Growth

The constants a and b do not actually matter.

For really large values of n, n^2 suppresses n.

n	1	10	100	1000	10000
n^2	1	100	10000	1000000	100000000

For really large values of n,

$$\frac{1}{2} n^2 + (a + 1/2)n + b \approx \frac{1}{2} n^2.$$

Orders of Growth

One more approximation:

We only care about how the runtime **scales** as the input size **scales**, so the constant factor is *irrelevant*.

$\frac{1}{2} n^2$ scales similarly to n^2.

For example, if the input size *doubles*, both functions *quadruple*.

Orders of Growth

```
def sum_facts(n):
    '''Adds factorials of
    integers from 1 to n.'''
    sum, k = 0, 1
    while k <= n:
        sum += fact(k)
        k = k + 1
    return sum
```

Time taken by this function scales roughly *quadratically* as the input size scales.

This function runs in ***quadratic time***.

Orders of Growth

A few important observations:

1. We only care about ***really large input values***, since computers can deal with small values really quickly.
2. We can ***ignore any constant factors*** in front of *polynomial* terms, since we want to know how the runtime **scales**.
3. We care about the ***worst-case*** runtime. If the function can be linear on some inputs and quadratic on other inputs, it runs in quadratic time overall. This can happen if your code has an `if` statement, for example.

"For large input values"

How do we communicate the worst-case ***asymptotic*** runtime to other computer scientists?

25

Big-O Notation

Let $f(n)$ be the runtime of a function.
It depends on the input size n.

We can then say

$$f(n) \in \underbrace{O(g(n))}_{\textit{Set of functions}}$$

"Belongs to"

26

Big-O Notation

Intuitively, $g(n)$ is an "upper bound" on $f(n)$.

$$f(n) \in O\big(g(n)\big)$$

if there are two integers c, N such that

"For large input values"

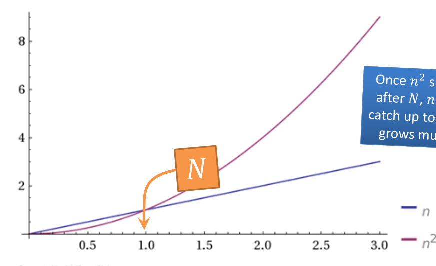
for all $n > N$,

$$f(n) < c \cdot g(n).$$

"Ignore the constant factors"

27

Big-O Notation: Example



N

Once n^2 surpasses n after N, n can never catch up to n^2, and n^2 grows much faster.

28

Big-O Notation: Example

$$\boxed{n} \in O(\boxed{n^2})$$

if there are two integers c, N such that

for all $n > \boxed{1}$,

$$f(n) < \boxed{1} \cdot g(n).$$

29

Big-O Notation

In this class, we are not going to worry about finding the values of c and N.

We would like you to get a basic intuition for how the function behaves for *large inputs*.

CS61B, CS70 and CS170 will cover this topic in much more detail.

30

Big-O Notation

Remember:

Constant factors *do not matter*.

Larger powered polynomial terms *suppress* smaller powered polynomial terms.

We care about the *worst-case* runtime.

Big-O Notation

Constant factors do not matter.

Size of input (N)	$t_1(n) = 3n^3$	$t_2(n) = 19{,}500{,}000n$
10	3.10 microseconds	200 milliseconds
100	3.0 milliseconds	2.0 seconds
1000	3.0 seconds	20 seconds
10000	49 minutes	3.2 minutes
100000	35 days (est.)	32 minutes
1000000	95 years (est.)	5.4 hours

From *Programming Pearls* (Addison-Wesley, 1986)

Jon Bentley ran two different programs to solve the same problem. The cubic algorithm was run on a Cray supercomputer, while the linear algorithm was run on a Radio Shack microcomputer. The microcomputer beat out the super computer for large n.

Big-O Notation

Which of these are correct?

- $\frac{1}{2}n^2 \in O(n^2)$
- $n^2 \in O(n)$
- $15000n + 3 \in O(n)$
- $5n^2 + 6n + 3 \in O(n^2)$

Big-O Notation

Which of these are correct?

- $\frac{1}{2}n^2 \in O(n^2)$ **Correct**
- $n^2 \in O(n)$ **Incorrect**
- $15000n + 3 \in O(n)$ **Correct**
- $5n^2 + 6n + 3 \in O(n^2)$ **Correct**

Big-O Notation

How does this relate to **asymptotic** runtime?

If a function runs in ***constant time***, its runtime is in $O(1)$.
("Its runtime is bounded above by a constant multiple of 1.")
If a function runs in ***linear time***, its runtime is in $O(n)$.
("Its runtime is bounded above by a constant multiple of n.")
If a function runs in ***quadratic time***, its runtime is in $O(n^2)$.
("Its runtime is bounded above by a constant multiple of n^2.")

Common Runtimes

"POLYNOMIAL"

Class of Functions	Common Name	Commonly found in
$O(1)$	Constant	Searching and arithmetic
$O(\log n)$	Logarithmic	Searching
$O(\sqrt{n})$	Root-n	Primality checks
$O(n)$	Linear	Searching, sorting
$O(n \log n)$	Linearithmic/loglinear	Sorting
$O(n^2)$	Quadratic	Sorting
$O(n^3)$	Cubic	Matrix multiplication
$O(2^n)$	Exponential	Enumeration

There are many problems for which the worst-case runtime is exponential. There has yet been no proof that these problems have polynomial solutions, and there has been no proof that a polynomial solution does not exist.
One example is the problem of finding the shortest tour through a set of cities.

Common Runtimes

Generally, "efficient" code is code that has a *polynomial* asymptotic runtime. The lower the power on the polynomial, the better.

Big-Theta and Big-Omega Notation

We defined earlier $f(n) \in O(g(n))$
$g(n)$ is an ***upper bound*** on $f(n)$.

If $f(n) = O(g(n))$, then $g(n) \in \Omega(f(n))$.
$f(n)$ is a ***lower bound*** on $g(n)$.

If $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$, then
$f(n) = \Theta(g(n))$.
$g(n)$ is a ***tight bound*** on $f(n)$.

Which Algorithm is Better?

```
def sum1(n):
    ''' Adds all numbers from 1 to n. '''
    sum, k = 0, 1
    while k <= n:
        sum += k
        k += 1
    return sum

def sum2(n):
    ''' Adds all numbers from 1 to n. '''
    return (n * (n+1))/2
```


Which Algorithm is Better?

```
def sum1(n):
    ''' Adds all numbers from 1 to n. '''
    sum, k = 0, 1
    while k <= n:
        sum += k
        k += 1
    return sum

#The second one is better
def sum2(n):
    ''' Adds all numbers from 1 to n. '''
    return (n * (n+1))/2
```


Conclusion

- One measure of efficiency of an algorithm and the function that implements it is to measure its runtime.
- In *asymptotic runtime analysis*, we determine how the runtime of a program scales as the size of the input scales.
- Big-O, Big-Omega and Big-Theta notations are used to establish relationships between functions for *large input sizes*.
- ***Preview***: The other computational player: data.