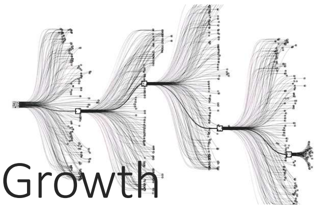


Orders of Growth

61A SUMMER 2016

GUEST LECTURER: JONGMIN JEROME BAEK (JBAEK080@BERKELEY.EDU)

There's more than one way to do it

Usually, there are many ways to *solve the same problem*. In jargon: there are many ways to *implement the same functional abstraction*.

e.g. Factoring N:

```
def factor_naive(N):
    factors = []
    i = 1
    while i <= N:
        if N % i == 0:
            factors += [i]
    return factors
```

```
def factor_clever(N):
    factors = []
    i = 1
    while i <= N ** 0.5:
        if N % i == 0:
            factors += [i, N/i]
    return factors
```

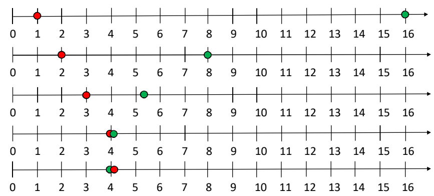
There's more than one way to do it

For the past two weeks, we talked about how to solve a problem *correctly.*

But what if there are *multiple* ways to solve a problem correctly? For example, `square` may be implemented in two ways.

Are some ways "better" than others?

The cleverness: proof by picture



There's more than one way to do it

```
def square(n):
    a, b, total = 0, n, 0
        while b:
            a, b = a + 1, b - 1
            total = total + a + b
    return total
```

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

The cleverness: dense math proof

1. Checking if a divides N is the same as checking if $\frac{N}{a}$ divides N.
2. If $\sqrt{N} + 1$ divides N, then $\frac{N}{\sqrt{N}+1}$ divides N.
3. $\frac{N}{\sqrt{N}+1}$ is evidently smaller than $\frac{N}{\sqrt{N}} = \sqrt{N}$.
4. We've already checked all numbers smaller than $\sqrt{N}$, so there's no need to check $\sqrt{N}+1$.

Why do we care?

Speed!

The naïve way divides N times. The clever way divides √N times.

This may not seem like a big deal, but…

Why do we care?

Naïve way: Try all sequences of moves and find the best sequence of moves.

There are approximately 150 turns in a game and 200 possible choices per step. This is 200^{150} possible games.

If each atom in the Universe did one step of computation each nanosecond, the Universe would need to start over about four times to try all sequences of moves and find the best sequence of moves.

Why do we care?

"Amazon calculated that a page load slowdown of just one second could cost it $1.6 billion in sales each year. Google has calculated that by slowing its search results by just four tenths of a second they could lose 8 million searches per day."

http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

Why do we care?

Clever way: As far as we know, there is no clever way to solve the game of Go! There are some ways to generate approximate solutions, such as using Google's AlphaGo, or using the human brain. More on this at the end of lecture, if we have time.

We simply do not know how to solve some really hard but really important problems. (We can find approximations.)

Why do we care?

To take an extreme example, consider the game of Go:

Go is a very complicated game. It is probably the most complex board game that is widely played by humans.

Two players play each other. Each turn, a player can make one of approximately 200 choices. There are about 150 turns.

How do we care?

How do we claim *mathematically* that one function runs faster than another?

A function is executed on some input. For example: in the factoring function, the input is the number we want to factor.

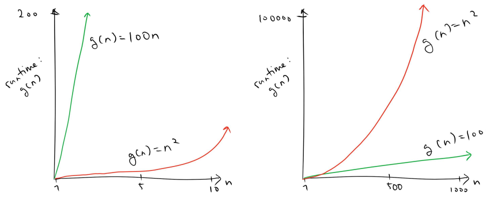
How long a function takes to run *depends on the size of the input.*

How do we care?

What we want to know: given some function, such as `factor`, how long will this function take to run?

Roughly, how many lines of code will this function need to execute? How does this depend on the size of the input?

Mole's eye view vs. Bird's eye view



How do we care?

For every function *f(N)*,

We can find a function *g(N)*:

g(N) describes how long *f(N)* takes to run as a function of *N*.

We don't care about…

More precisely, we don't care about

- Constant factors
- Any term that is not the largest term

For example:

- $g_1(N) = 3N^2 + N$ **vs.** $g_2(N) = N^2 + 6$: we treat both functions as *about the same*; we say both take *quadratic time* and denote this as $\Theta(N^2)$.
- $g_3(N) = 35N$ **vs.** $g4(N) = N + 49$: we treat both functions as *about the same*; we say both take *linear time* and denote this as $\Theta(N)$.

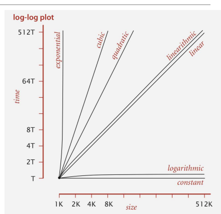
We don't care about…

We don't need to know *exactly* how the run-time grows as the size of the input increases.

We only want a rough idea.

To get a rough idea, we look from a birds-eye view.

That is, we look at the shape of *g(N)* as *N* gets really, really big.



Typical orders of growth

Why the lack of care?

Mathematical rigorousness: it saves you the headache of trivialities and it's useful to think of g(n) as n approaches infinity

Moore's law: computers are always getting exponentially faster

Constant factors/small terms are easier to reduce: even if you reduce constant factors, that doesn't tell you much about the *nature of the problem*

Mathematical Definition

N: size of the input

R(N): how long it takes to run the function on input of size N

$$R(N) = \theta(g(N))$$

Means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot g(N) \leq R(N) \leq k_2 \cdot g(N)$$

For all n larger than some minimum m

Linear Time

$\Theta(N)$

Pretty fast

Runtime increases linearly with input

```
def sum_all(lst):
    result = 0
    for e in lst:
        result += e
    return result
```

Constant Time

$\Theta(1)$

Fastest

Runtime doesn’t depend on size of input

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

def add(x, y):
    return x + y
```

Quadratic Time

$\Theta(N^2)$

Not fast

Runtime increases quadratically with input

```
def print_all_pairs(lst):
    for i in lst:
        for j in lst:
            print(i, j)
```

Logarithmic Time

$\Theta(\log N)$

Very fast

Runtime increases with input, but very little

```
def exp_decay(x):
  if x == 0:
      return 1
  else:
      return exp_decay(x//2) + 1
```

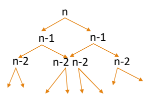
Exponential Time

$\Theta(2^N)$

Intractable

Runtime increases *extremely* fast with input

```
def fork_bomb(n):
    if n == 0:
        return 1
    return fork_bomb(n - 1) +
           fork_bomb(n - 1)
```



How to determine orders of growth

In CS61A, we particularly care about orders of growth when writing iterative or recursive code.

Iteration:

- How long does each iteration take?
- How many times do we loop?

Recursion:

- How long does each call to the function take?
- How many times do we call the function?

Why do we care?

What seems harder: factoring n, or finding the nth Fibonacci number?

Counter to intuition, factoring is a much harder problem!

Some problems are inherently harder than others. Theoretical computer science is in the business of classifying problems by how hard they are.

P vs. NP and the Computational Complexity Zoo:

https://www.youtube.com/watch?v=YX40hbAHx3s

Examples

```
def mystery1(n):  Θ(N)
      x = 0
      for i in range(n):
            x += 1
      return x

def mystery2(n):  Θ(N)
      if n == 0:
            return 0
      return mystery2(n-1) + 1
```

```
def mystery3(n):  Θ(N²)
      x = 0
      for i in range(n):
            for j in range(i):
                  x += 1
      return x

def mystery4(n):  Θ(logN)
      x = 1
      while x < n:
            x *= 2
      return x
```

Examples

```
def mystery5(n):  Θ(1)
   if n > 0:
      return mystery5(-n)+
             mystery5(-n)
   else:
      return 0
def mystery6(n):  Θ(1)
   i = 0
   while i > 0:
      i += 1
   return n
```

```
def mystery7(n):  Θ(2^N)
   if n == 0:
      return 1
   return mystery8(n-1) +
          mystery8(n-1)
def mystery8(n):  Θ(2^N)
   if n == 0:
      return 1
   return mystery7(n-1) +
          mystery7(n-1)
```