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?

```python
# A possible implementation of square using iteration
def square(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a + 1, b - 1
        total = total + a + b
    return total

# A possible implementation of square using recursion
def square(n):
    return n * n
```

The cleverness: proof by picture

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 $\sqrt{N}$ times.
This may not seem like a big deal, but...

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.”


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 \texttt{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?

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. \( g_4(N) = N + 49 \): we treat both functions as about the same; we say both take linear time and denote this as \( \Theta(N) \).

Why the lack of care?
Mathematical rigor: 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 doesn't depend on size of input

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

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

---

### Constant Time

**\( \Theta(1) \)**

Fastest

Runtime doesn’t depend on size of input

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

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

---

### Logarithmic Time

**\( \Theta(\log N) \)**

Very fast

Runtime increases with input, but very little

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

---

### Quadratic Time

**\( \Theta(N^2) \)**

Not fast

Runtime increases quadratically with input

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

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

---

### Exponential Time

**\( \Theta(2^N) \)**

Intractable

Runtime increases extremely fast with input

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

```
1
2
5
8
13
21
34
```
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?

Examples

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

```python
Ө(N)
```

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

```python
Ө(N²)
```

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

```python
Ө(N²)
```

Examples

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

```python
Ө(2^N)
```

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=YX40hb4Hx3s