CPE360

Analysis of Algorithms

Big-O, Theta and Omega

Good Code v/s Bad Code

How do you decide?

What we will learn

Solution A

Solution B

What is better? How do we define “better”?
Which will you pick? And how do we measure better?

Common metrics

- **Execution time (fast programs win, most of the time)**
- Run time memory
- Networking bandwidth
- Energy efficiency (battery life)
- Lines of code
- GPU optimized
- Etc., etc.,

What we need is a way to measure, communicate, and characterize solutions..

.. without even implementing in code!

#1
Algorithms, not code

Algorithms

Algorithms are a way to describe the *working steps* of a program..

..while being *independent* of programming languages

E.g., Matrix multiplication

```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
       C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```

E.g., Matrix multiplication

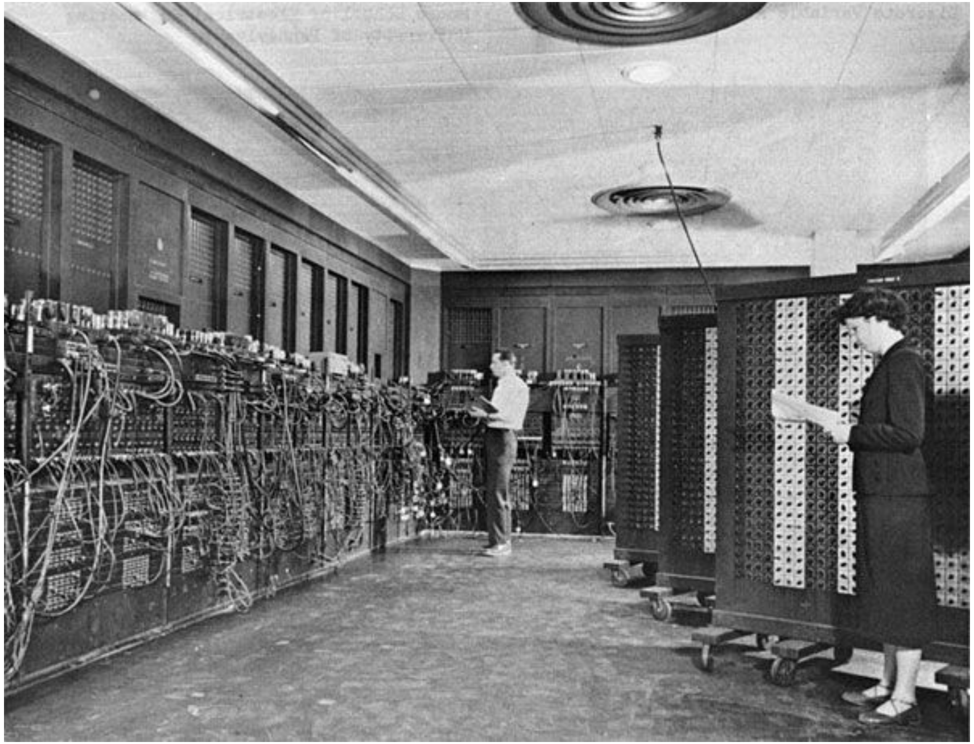
```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
       C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```

Take-aways:

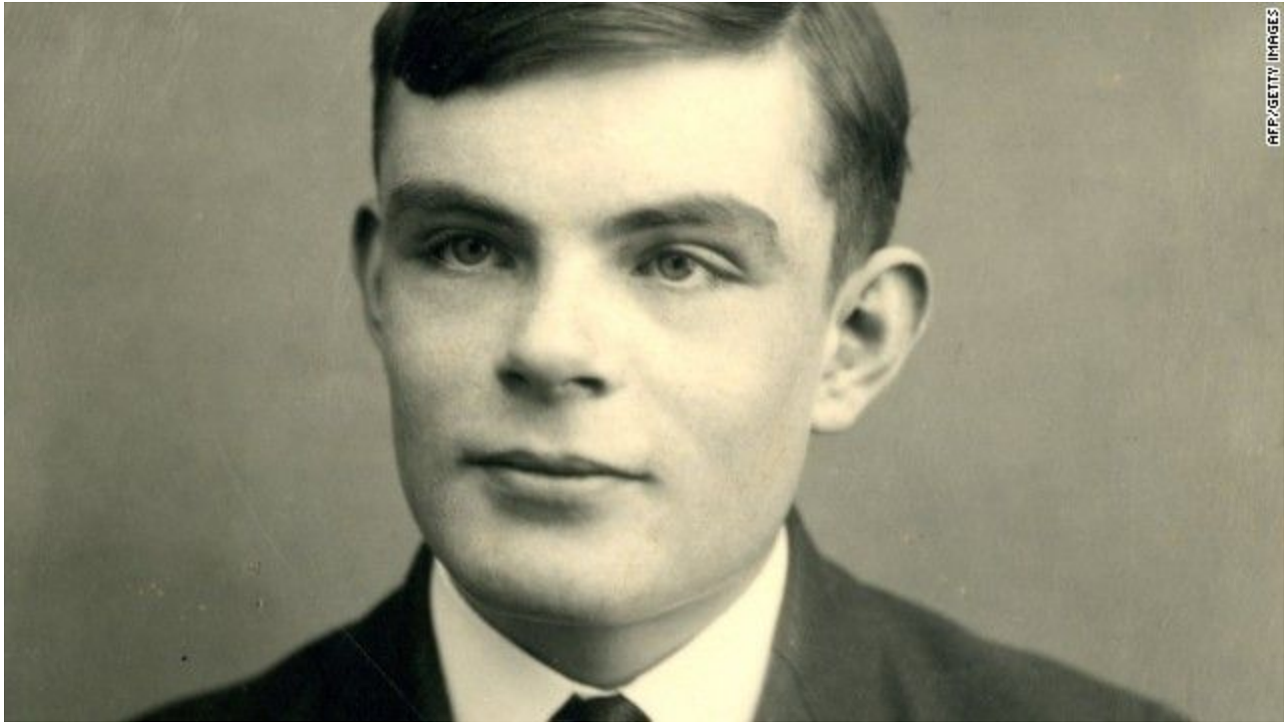
- Clearly defined input/output
- **Linear execution**
- Sequence of steps leading to output
- Steps involve iterations (e.g., loops), function calls, and decisions (if-else)

Let's rewind time

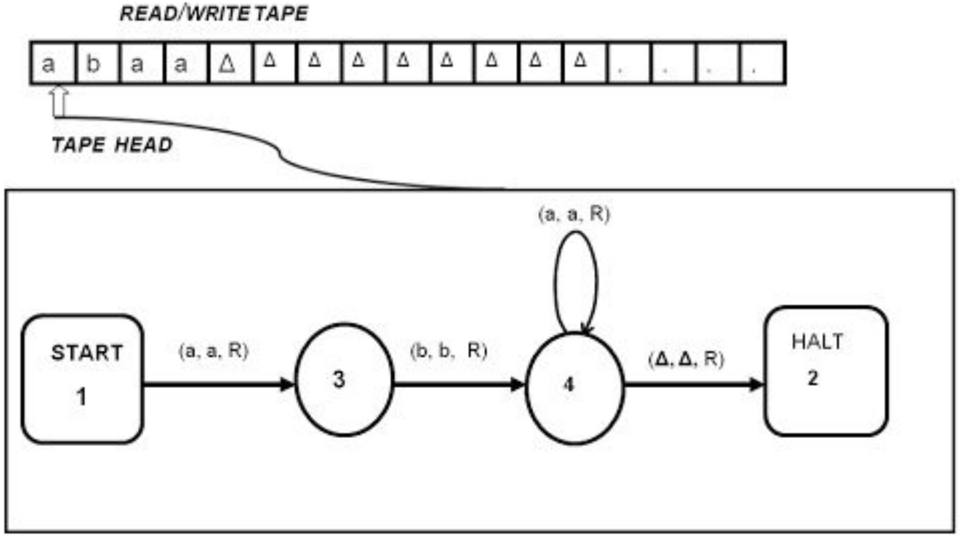
(how did we end up here?)



Alan Turing



Simplest Embodiment of a machine that can “think”

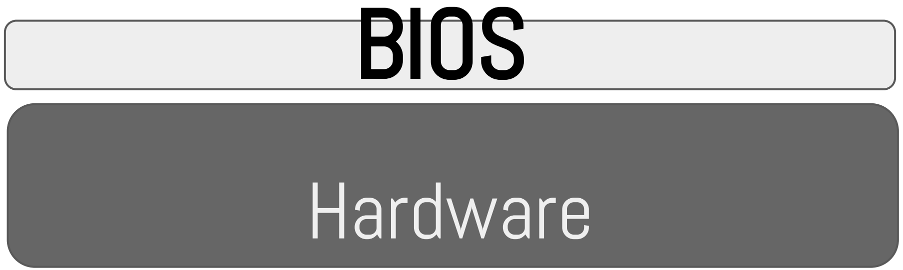


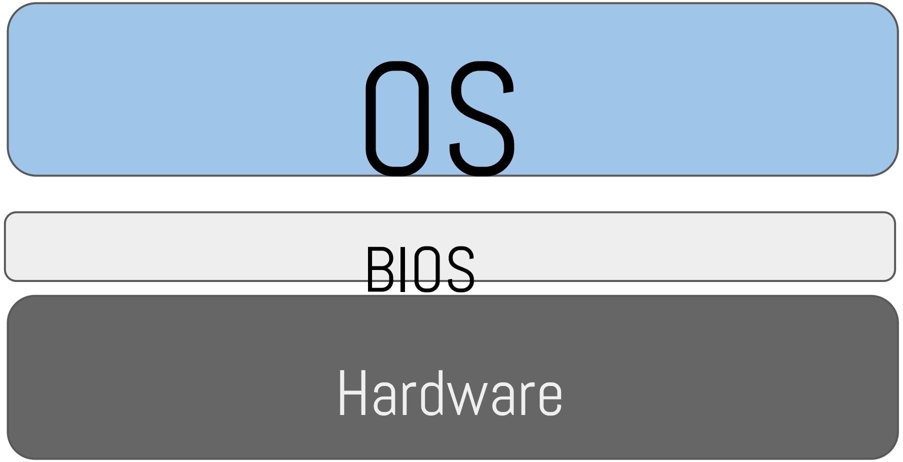
A Turing Machine for aba*

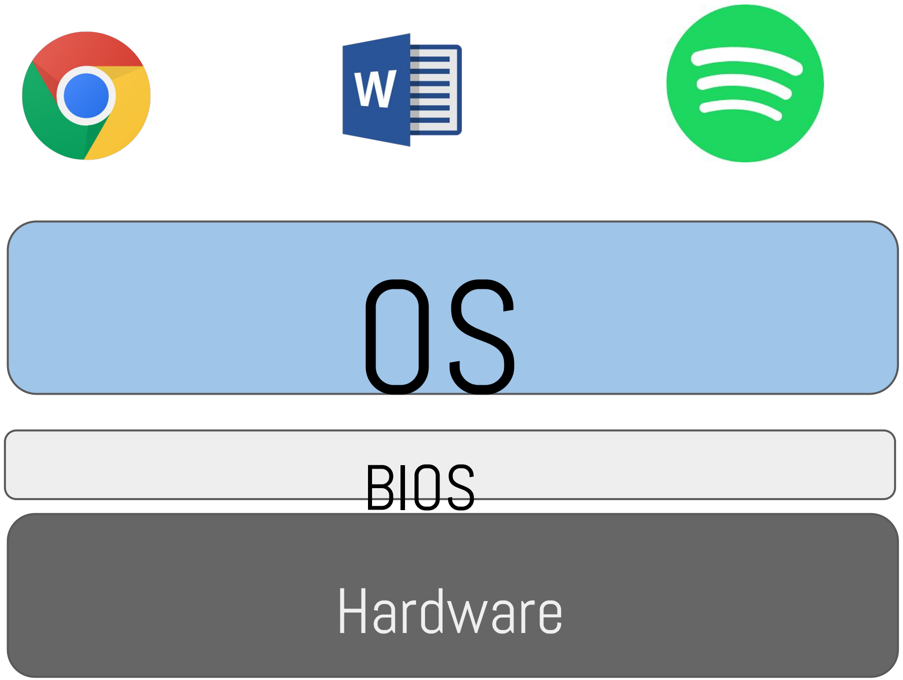
#2
Everything has a cost

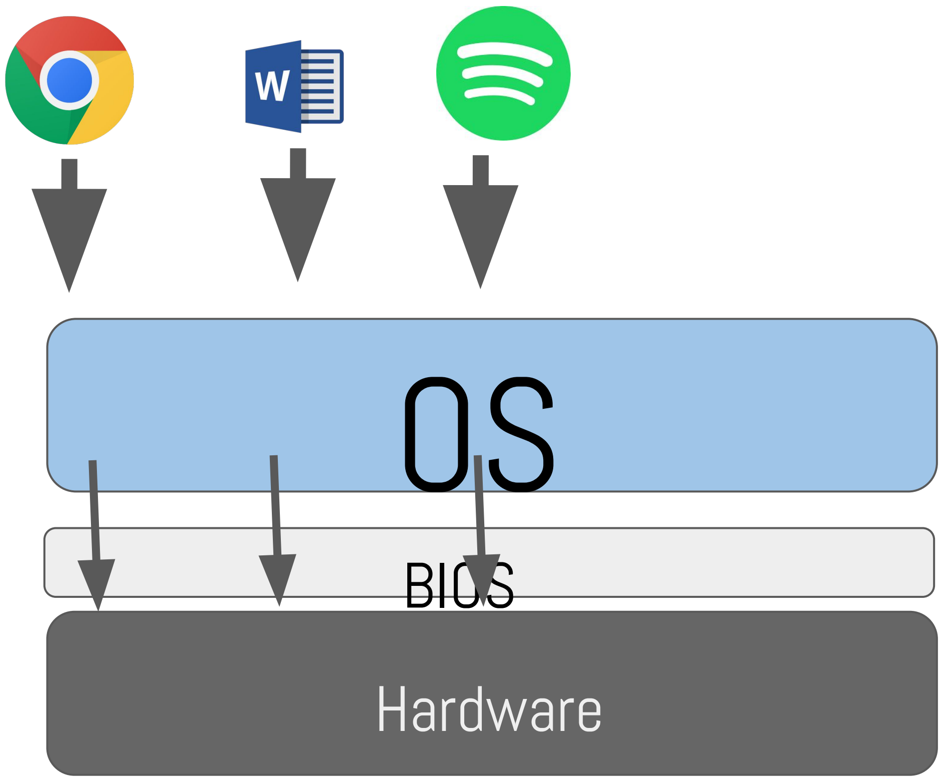
What happens when you compile/run?

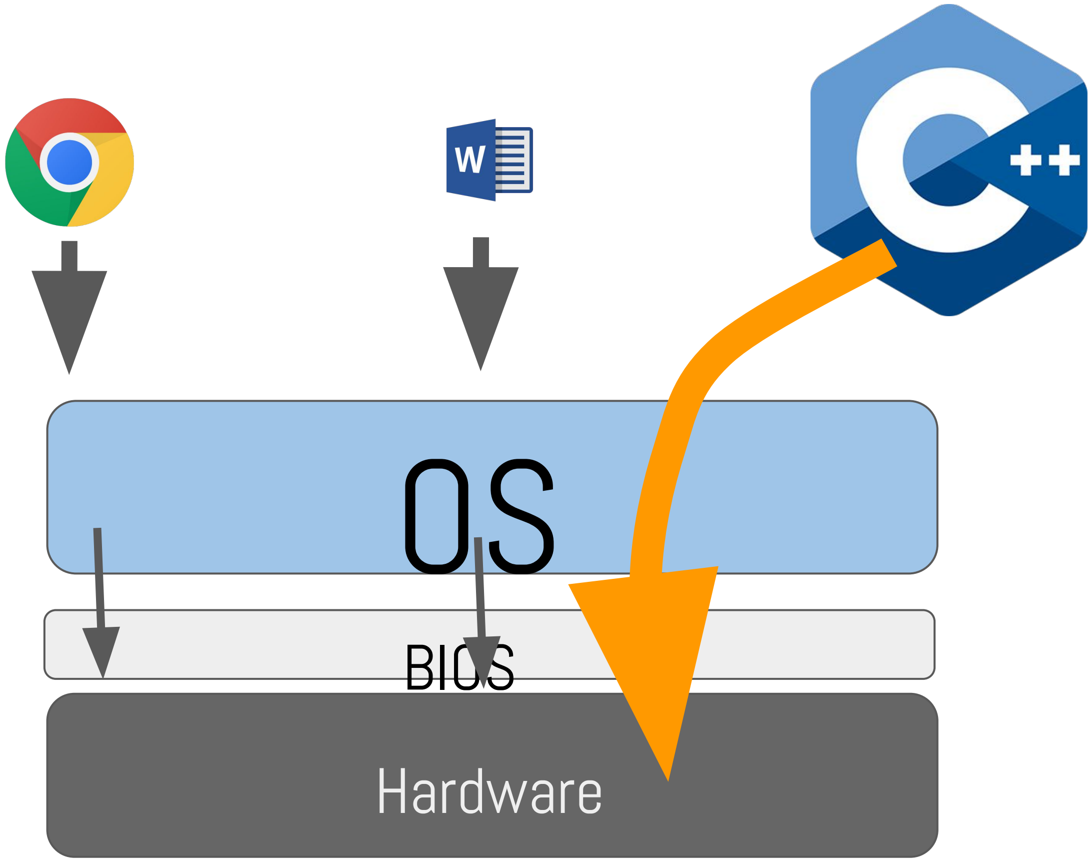
Hardware

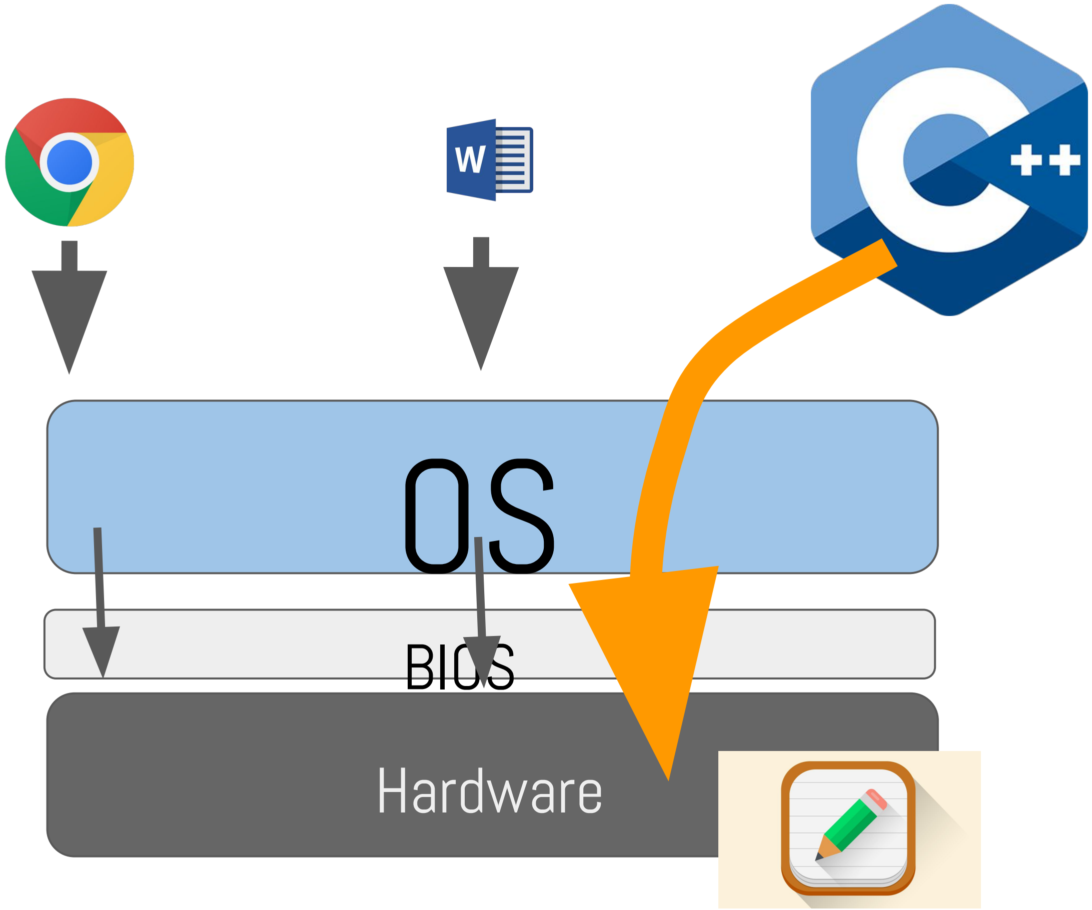

BIOS
Hardware


OS
BIOS
Hardware

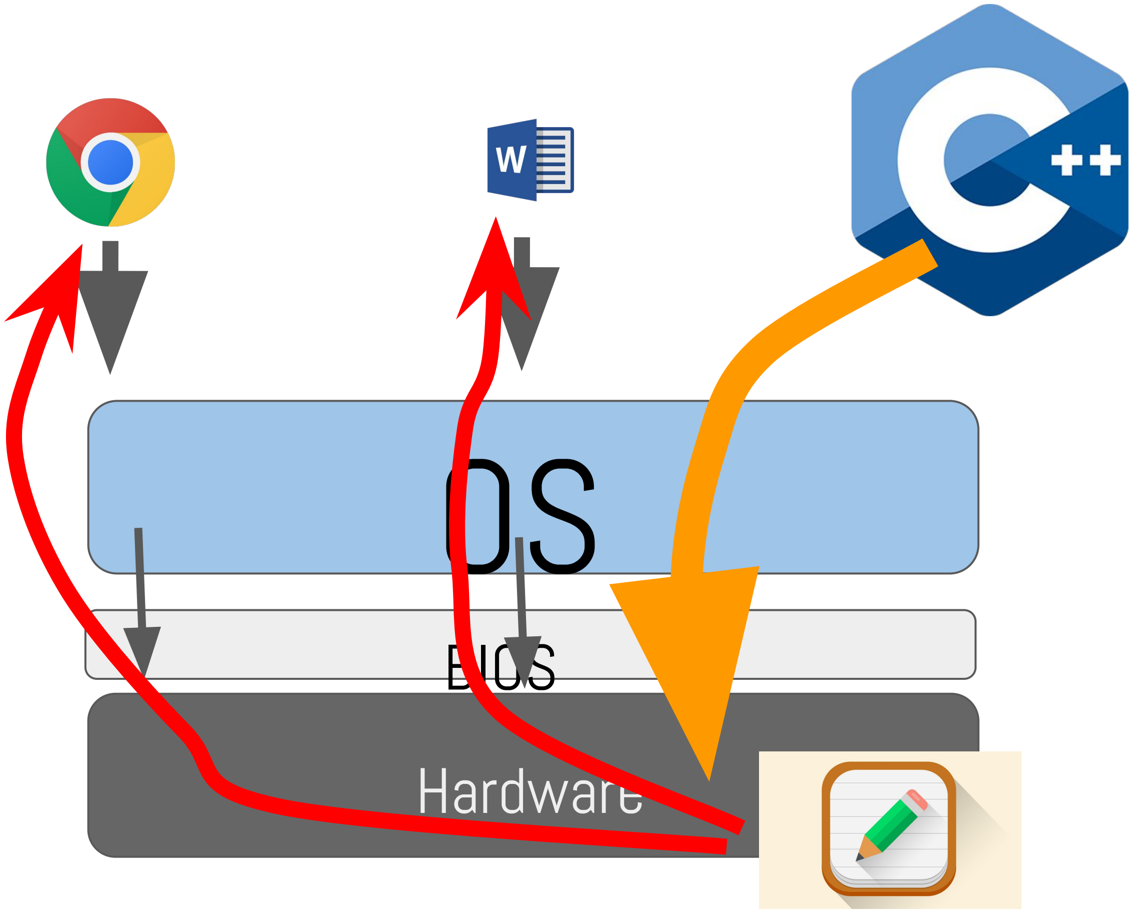

OS
BIOS
Hardware


OS
BIOS
Hardware


W
C++
OS
BIOS
Hardware

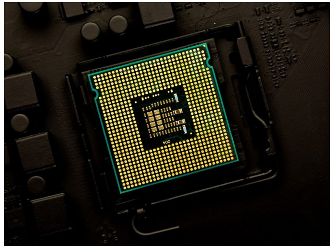


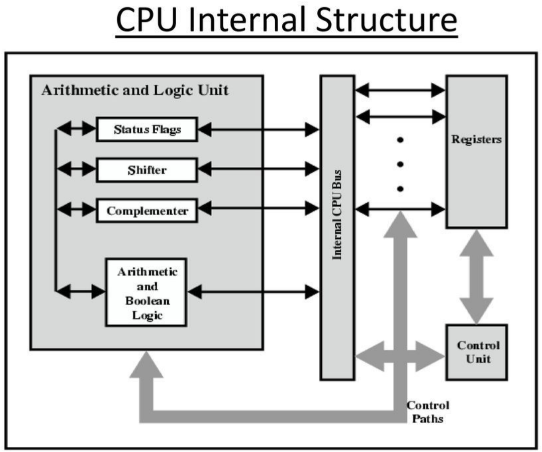
W
C++
OS
BIOS
Hardware


OS
BIOS
Hardware

CPU is the main bottleneck

Every line of code hits the CPU





CMP

ASSIGN

ADD/SUB

MUL

DIV

BRANCH

SUBROUTINE

Notion of cost

Every line of code become one of the following:

CMP (Compare)

ASSIGN (initialize variables, redefine value)

ADD/SUB (this is what your CPU is exceptionally good at)

MUL (special case of addition)

DIV (special case of MUL)

Cost Table

Loose approximation, but ratio mirrors reality

Instruction	Cost
CMP/ASSIGN	0
ADD	**1**
SUB	2
MUL	4
DIV	8

#3
Measuring Cost

Let's start simple..

```
int a = 5;
int b = 8;

b = a + b + 9;

if(b > 20) {
    b = b - 10;
}
else {
    b = b + 10;
}
```

Recall

```
int a = 5;
int b = 8;

b = a + b + 9;

if(b > 20) {
    b = b - 10;
}
else {
    b = b + 10;
}
```

Instruction	Cost
CMP/ASSIGN	0
ADD	**1**
SUB	2
MUL	4
DIV	8

Leads to..

```
int a = 5;          0
int b = 8;          0

b = a + b + 9; 2

if(b > 20) {        0
    b = b - 10;     2
}
else {
    b = b + 10;     1
}
```

Total: 5 units

Instruction	Cost
CMP/ASSIGN	0
ADD	**1**
SUB	2
MUL	4
DIV	8

Example 2: Loops

```
-------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < 10; i++) {
    b = a + i;
}
-------------------------------
```

Total cost: 20 units

```
i = 0; (1 x 0 = 0)
i < 10; (11 x 0 = 0)
i++;  (10 x 1 = 10)
b = a + i; (10 x 1 = 10)
```

Example 3: Nested Loops

```
------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        b = a + i + j;
    }
}
------------------------------
```

Analysis

```
------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        b = a + i + j;
    }
}
------------------------------
```

Total cost: 310 units

```
j = 0; (100 x 0 = 0)
j < 10; (110 x 0 = 0)
j++; (100 x 1 = 100)
b = a+i+j; (100 x 2 = 200)

i = 0; (1 x 0 = 0)
i < 10; (11 x 0 = 0)
i++;  (10 x 1 = 10)
```

Nested loops are expensive!

Try this for yourself

```
------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        for(int k = 0; k < 10; k++) {
            b = a + i + j;
        }
    }
}
------------------------------
```

Total cost: ?? units

Something more realistic

```
------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        b = a + i + j;
    }
}
------------------------------
```

Analysis

```
-------------------------------
int a = 5;
int b = 0;

for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        b = a + i + j;
    }
}
-------------------------------
```

```
j = 0; (N x 0 = 0)
j < N; (N+1) x 0 = 0
j++; (N^2 x 1 = N^2)
b = a+i+j; (N^2 x 2 = 2N^2)

i = 0; (1 x 0 = 0)
i < 10; (N + 1) x 0 = 0
i++;  (N x 1 = N)
```

Total cost: $3(N^2) + N$

Problem of choice

$$3(N^2) + N + 9$$

$$2(N^2) + 56N + 976$$

What is better? How so?

What's not very obvious is they are actually both the <u>same</u>!

#4
Big-O: The definitive way

Common problem

When algorithms are analyzed, we get a ton of math equations.

This becomes a big problem when **comparing** one algorithm to another.

There has to be a way to **simplify** these results.

Basic Question

Assume that the analysis of an algorithm produces $f(n)$

E.g., $f(n) = 2n^2 + n + 1$

Then, how efficient is this algorithm when

n→(infinity)?

Let's start simple

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

It's obvious that n^2 grows faster than 'n' or '1'. It is the most dominant term.
Let's say we denote this as $g(n) = n^2$

Let's start simple

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

It's obvious that n^2 grows faster than 'n' or '1'. It is the most dominant term.
Let's say we denote this as $g(n) = n^2$

To show that the big-*O* for $f(n)$ is n^2, we just need to do the following:

$\boldsymbol{f(n) <= C \times g(n)}$, where C is a constant > 0

$2n^2 + n + 1 <= 3 * n^2$

Then $\boldsymbol{f(n) = O(g(n))}$

(To show this, simply pick C = 4 and we are all set!)
In other words, $\boldsymbol{f(n) = O(n^2)}$

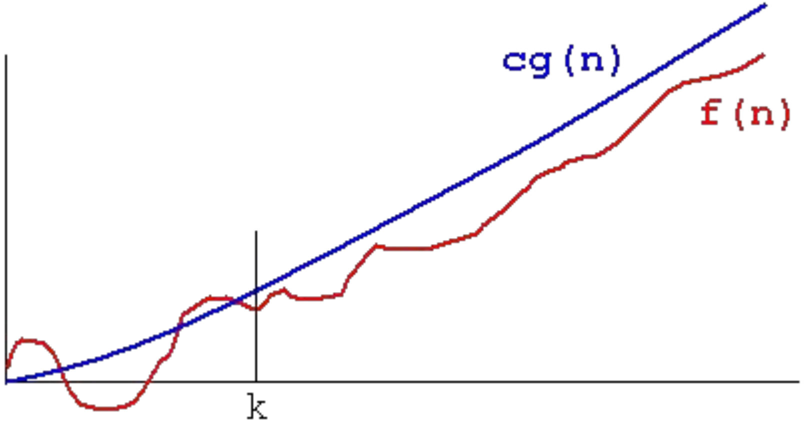
What this does

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

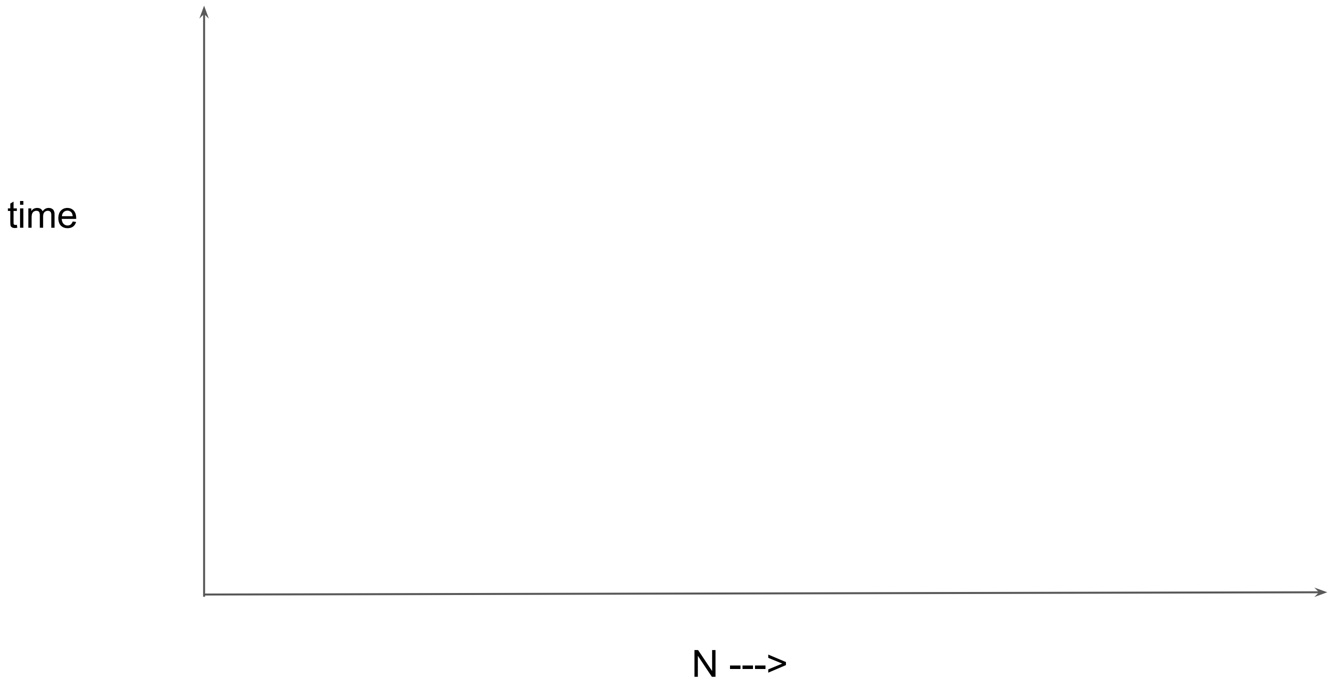
$g(n) = n^2$

Threshold "k" beyond which inequality holds true.

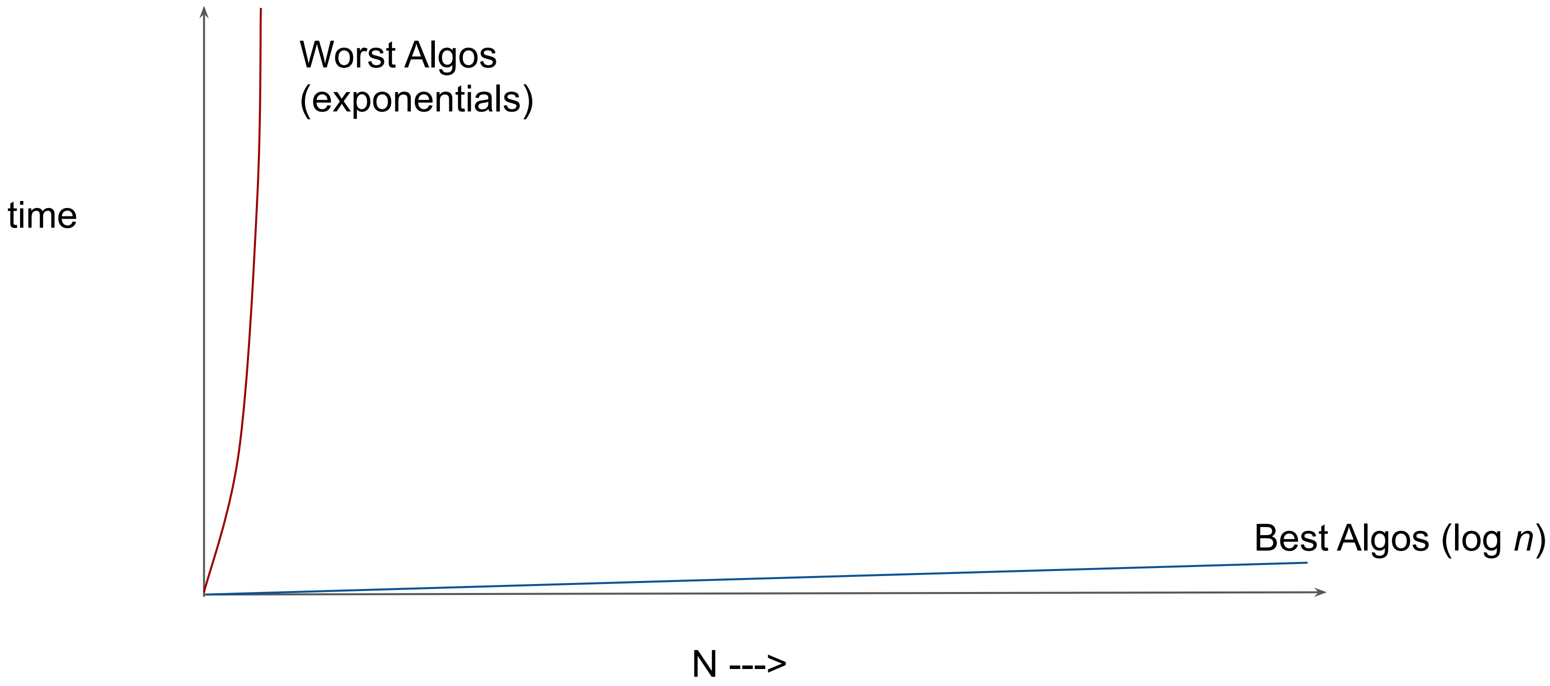
Big-O established an upper bound!



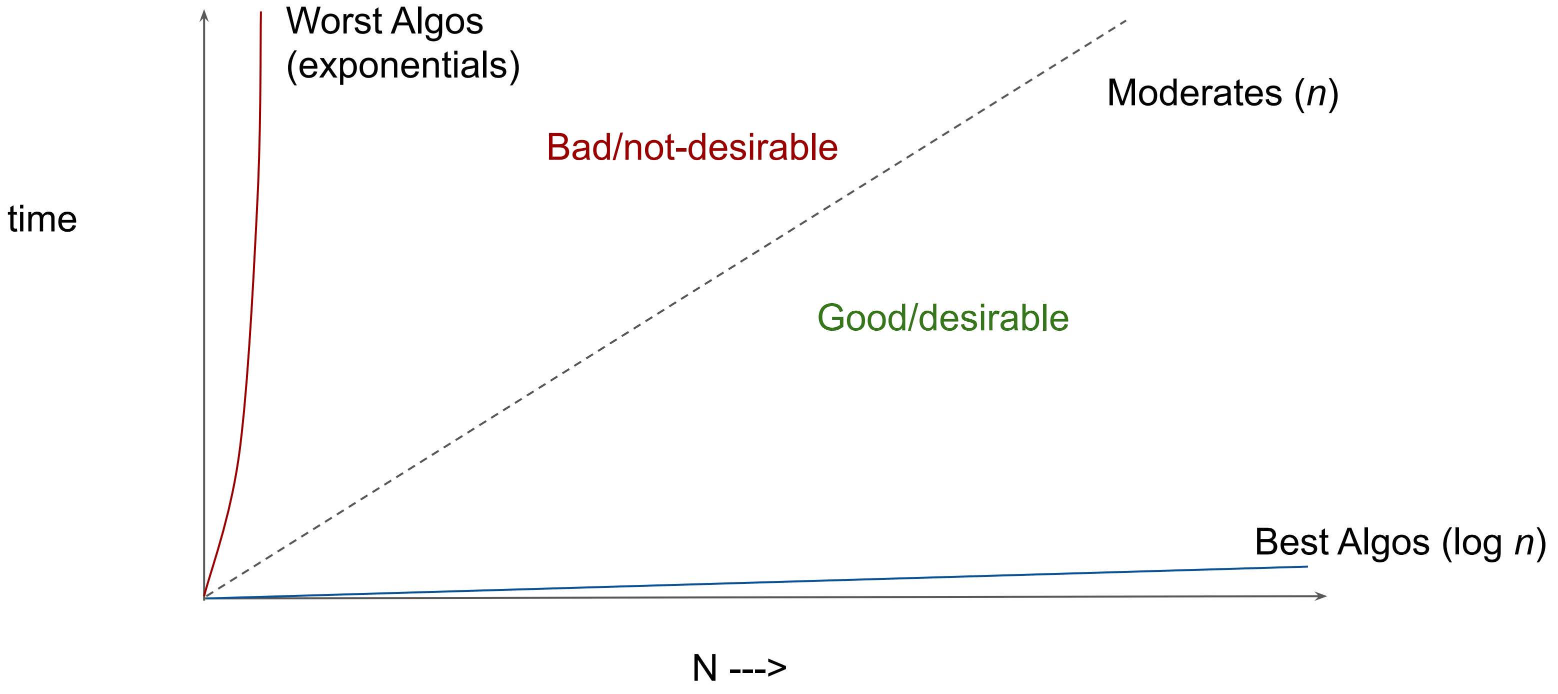
Spectrum of possibilities



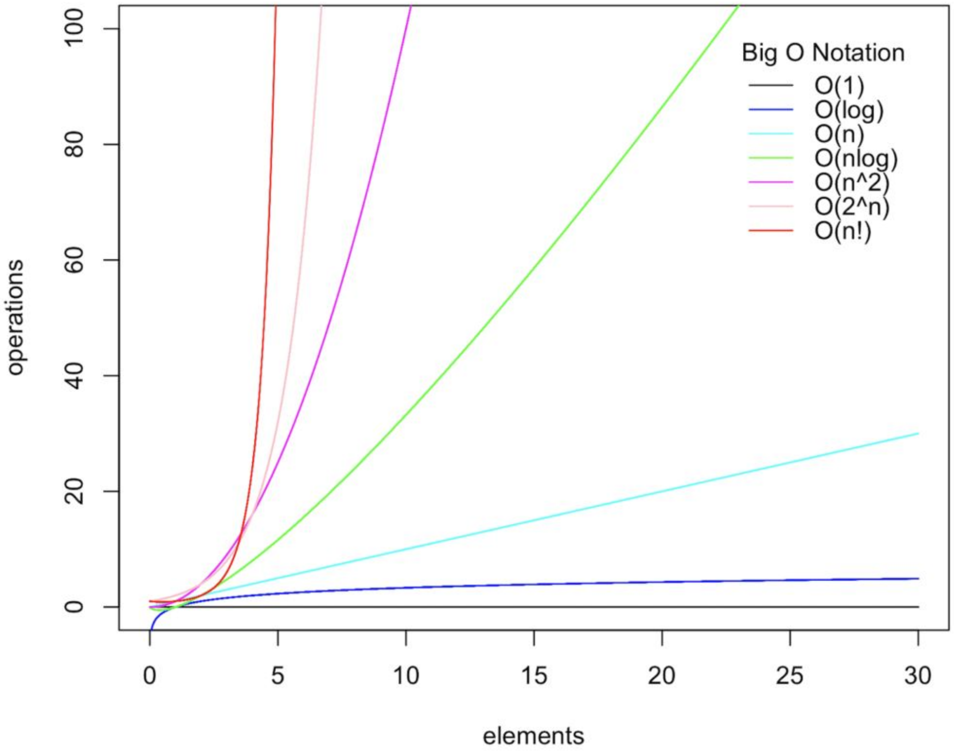
Spectrum of possibilities



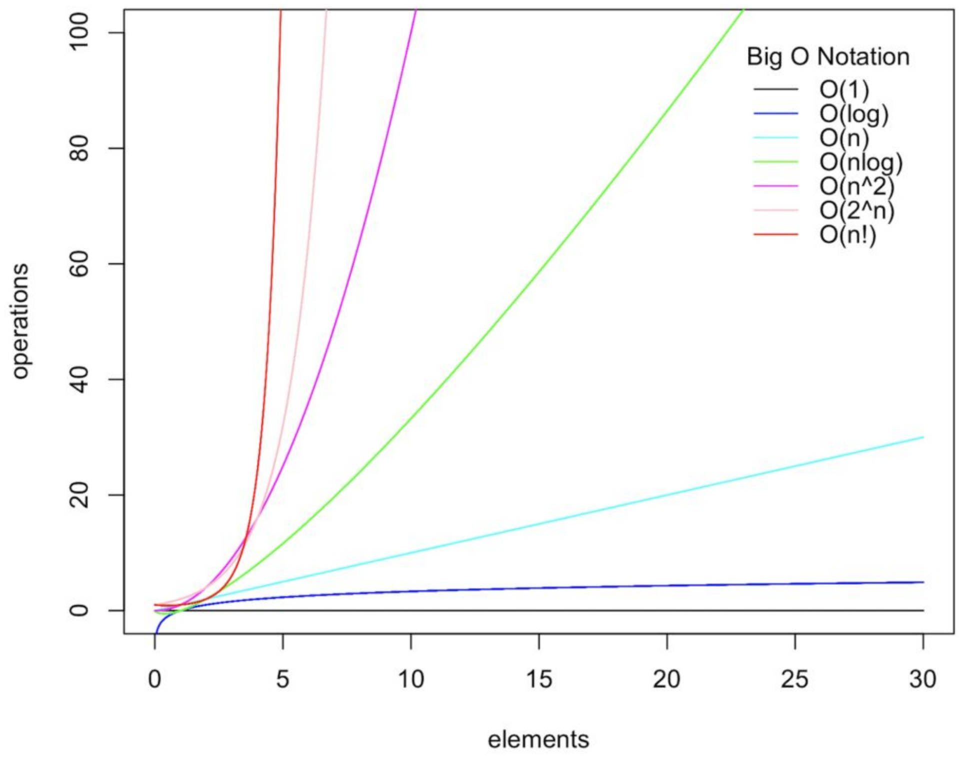
Spectrum of possibilities



Spectrum of possibilities



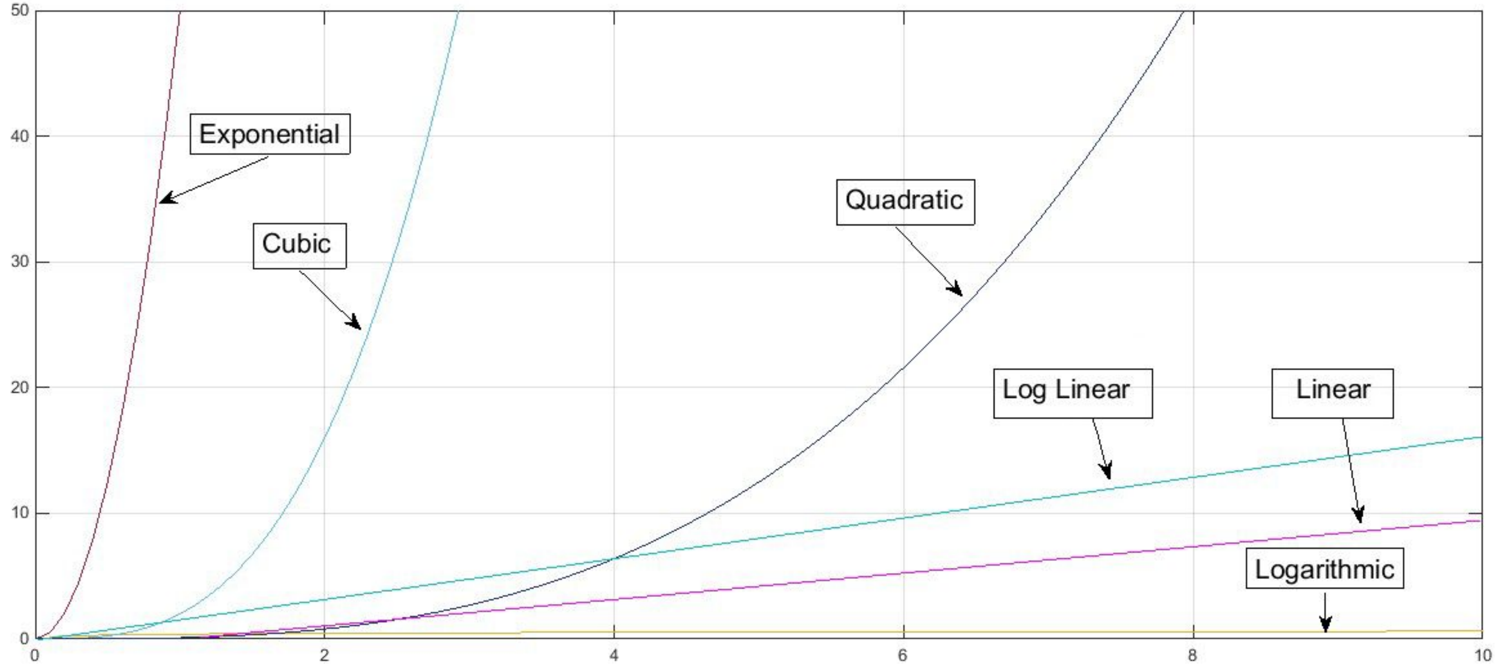
Spectrum of possibilities



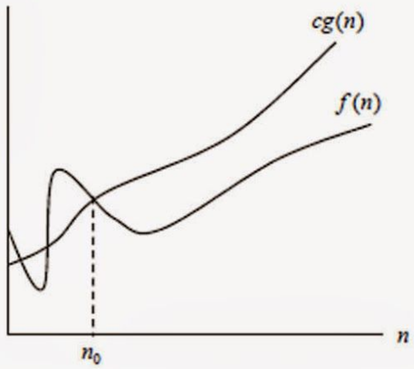
Goal is simple: **which one of these curves best describes my algorithm?**

1. Analyze algorithm to produce *f(n)*
2. From the terms in *f(n)*, pick the most dominant term. Call it *g(n)*.
3. Your *g(n)* is **one of the curves** to the left.
4. Pick a constant, C, such that this is true: C**g(n)* >= *f(n)*.
5. If you can show this, you can now say that *f(n)* = *O*(*g(n)*).
6. Finally, communicate *g(n)* to the world!

Better way to look at this



The (Big) O Notation



$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

$g(n)$ is an asymptotic upper bound for $f(n)$.

Examples:

$n^2 = O(n^2)$

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

$n^2 + 1000n = O(n^2)$

$5230n^2 + 1000n = O(n^2)$

$n = O(n^2)$

$\dfrac{n}{1200} = O(n^2)$

$n^{1.99999} = O(n^2)$

$\dfrac{n^2}{\log n} = O(n^2)$

Note: Since changing the base of a log only changes the function by a constant factor, we usually don't worry about log bases in asymptotic notation.

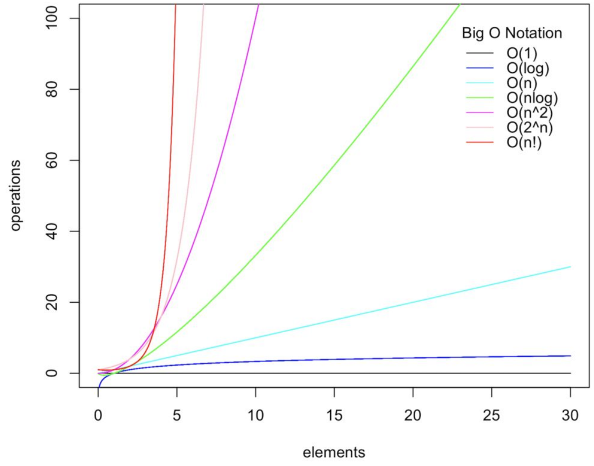
#5
Big-O: Some Examples

Big O: examples

$T(n)$	Complexity
$5n^3 + 200n^2 + 15$	$O(n^3)$
$3n^2 + 2^{300}$	$O(n^2)$
$5\log_2 n + 15\ln n$	$O(\log n)$
$2\log n^3$	$O(\log n)$
$4n + \log n$	$O(n)$
2^{64}	$O(1)$
$\log n^{10} + 2\sqrt{n}$	$O(\sqrt{n})$
$2^n + n^{1000}$	$O(2^n)$

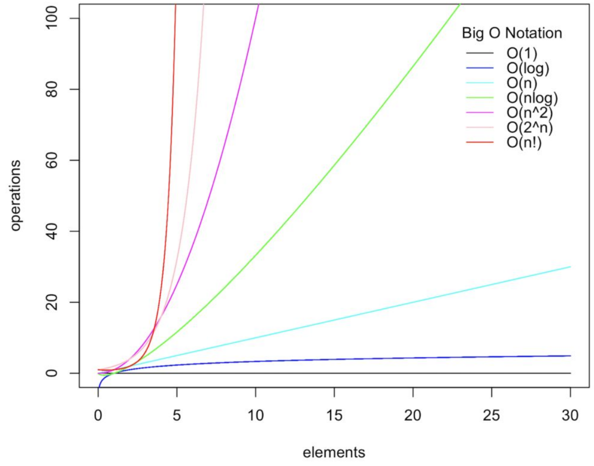
$5n^3 + 200n^2 + 15$	$O(n^3)$

- *n^3* **is most dominant**



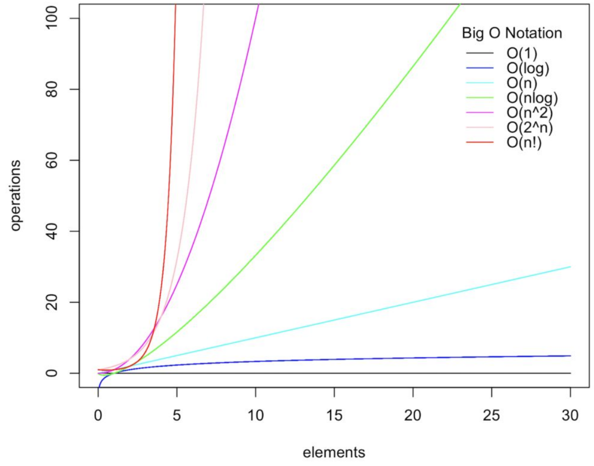
$3n^2 + 2^{300}$	$O(n^2)$

- n^2 is most dominant



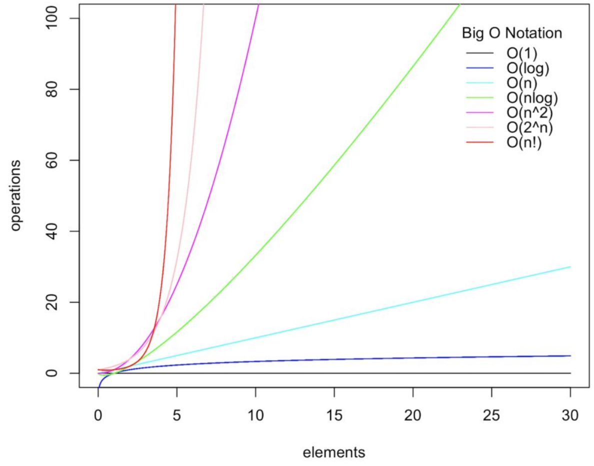
$5\log_2 n + 15\ln n$	$O(\log n)$

- *log(n)* **is most dominant**



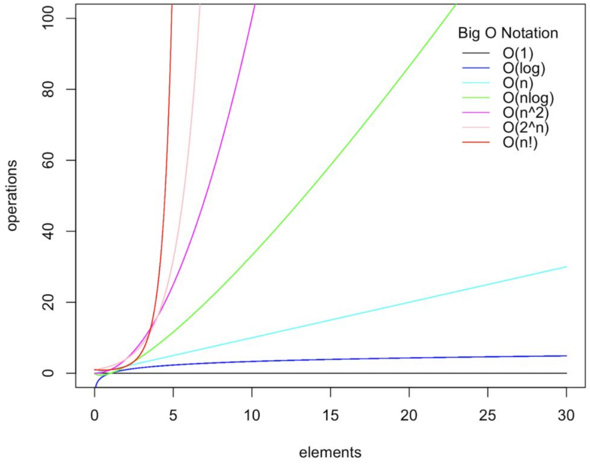
$2\log n^3$	$O(\log n)$

- Simplify: $\log n^3 = 3 \log n$
- So it's basically just $O(\log n)$



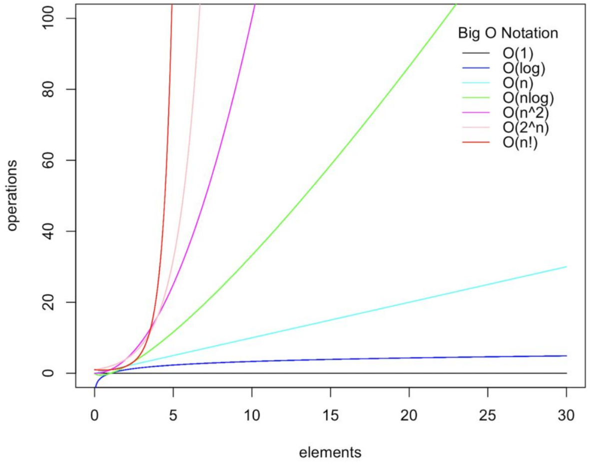
$4n + \log n$	$O(n)$

- ***n* is more dominant than log *n***



2^{64}	$O(1)$

- This is a curious case
- Notice that there are no terms with "*n*"
- In other words, this growth is independent of *n*
- As a result, this is a constant (basically a flat line)



$\log n^{10} + 2\sqrt{n}$	$O(\sqrt{n})$

$2^n + n^{1000}$	$O(2^n)$

#6
Theta and Omega Notations

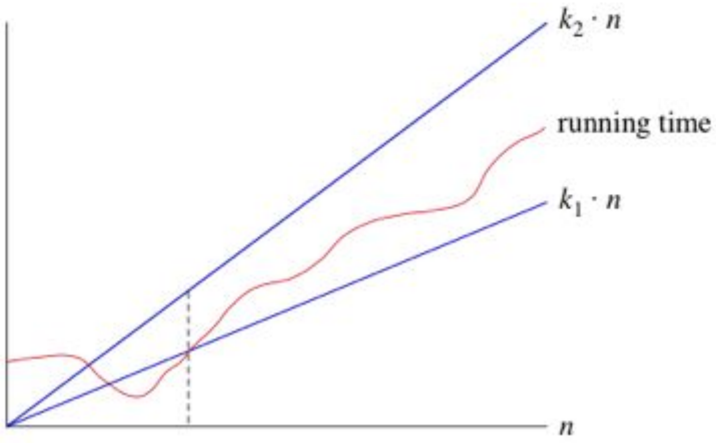
Theta: Average Case Analysis

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

$g(n) = n^2$

$k1 * g(n) <= f(n) <= k2*g(n)$

Theta establishes an upper *and* lower bound!



Omega: Best Case Analysis

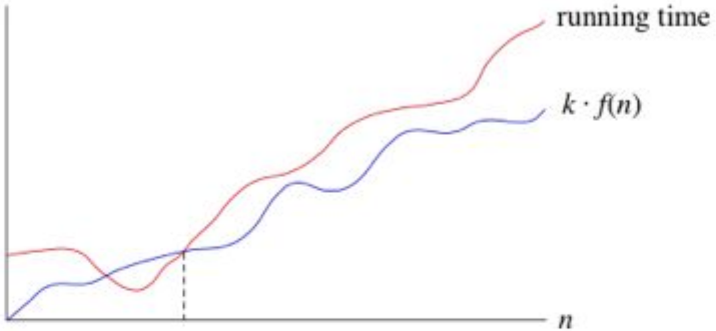
$f(n) = 2n^2 + n + 1$

$g(n) = n^2$

$f(n) > k*g(n)$

Omega **establishes a lower bound.**

P.S.: <u>Nobody</u> cares about lower bounds.



#7
P v/s NP problems

Polynomial/Non-Polynomial

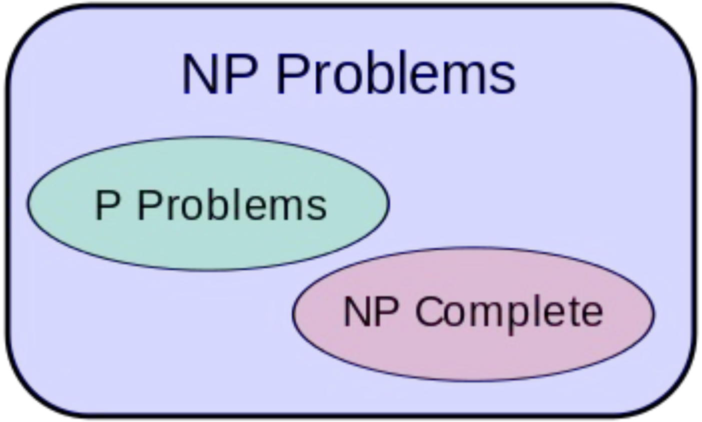
There are problems out there that are incredibly hard to solve (Non Polynomial, or **NP**).

But a given “solution” to such a problem can be verified quickly (Polynomial, or **P**).

A central debate in theoretical CS is this: can such problems also be “solved” quickly.

In other words, is **P = NP**.

This is (literally) a million dollar question.



Travelling Salesman Problem

Incredibly difficult to actually solve the problem (NP).

But.. if a solution is given, it is very easy to “verify” it’s correctness (P).

There are thousands of such “open” problems in CS.