Time complexity and the Big O
With a specific **computational model** in mind:

**Definition:**

The **running time** of an algorithm $A$ is a **function**

$$T_A : \mathbb{N} \rightarrow \mathbb{N}$$

defined by

$$T_A(n) = \max_{\text{instances} \ I \text{ of size } n} \{ \# \text{ steps } A \text{ takes on } I \}$$

**worst-case**

Write $T(n)$ when $A$ is clear from context.
The Random-Access Machine (RAM) model

memory access e.g. A[94] takes 1 step

+ , - , / , *, <, >, etc. e.g. 245*12894 takes 1 step

Actually:

Assume arithmetic operations take 1 step IF numbers are bounded by poly(n).

Unless specified otherwise, we use this model.
Need one more level of abstraction

There is an Algorithm that decides PALINDROME in time

\[ T(n) = \frac{1}{2}n^2 + \frac{3}{2}n + 1. \]

Breaking News 1: Intel releases a new processor, native support for vector arithmetic
Breaking News 2: Microsoft releases 64 bit OS

For algorithms at a higher level (like, in C, or pseudocode), it’s not exactly clear what counts as “1” time step

Even for slightly more complicated algorithms, it’s nearly impossible to calculate so precisely
Palindrome T has running time

\[ T(n) = \frac{1}{2} n^2 + \frac{3}{2} n + 1 \]

We want to use the right level of abstraction!

The key takeaway of this T(n):

it’s “quadratic”; that is, proportional to \( n^2 \).

This leads us to...
Next Great Idea:

**Big-O notation**

The (T)CS way to compare functions:

$$
\begin{align*}
O(\cdot) & \leq \Omega(\cdot) \\
& = \Theta(\cdot)
\end{align*}
$$
Big O

Our notation for $\leq$ when comparing functions.

The right level of abstraction!

“Sweet spot”

- coarse enough to suppress details like *programming language, compiler, architecture, …*

- sharp enough to make comparisons between different *algorithmic approaches*. 
Informal: An upper bound that ignores constant factors + ignores small values of n

\[ f(n) = O(g(n)) \] roughly means

\[ f(n) \leq g(n) \] up to a constant (for “large enough” n)

\[ n^2 + 100n + 500 \text{ is } O(n^2) \]
Big O

The graph shows the function $n^2 + 100n + 500$ compared to $n^2$. The green curve represents $n^2$, and the red curve represents $n^2 + 100n + 500$. As $n$ increases, the red curve approaches the green curve, indicating that $n^2 + 100n + 500$ grows asymptotically as $n^2$. This is consistent with the Big O notation, which describes the upper bound of the growth rate of a function.
Big O

\[ n^2 + 100n + 500 \]
Big O

For $f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+$

$$f(n) = O(g(n))$$  
roughly means

$$f(n) \leq g(n)$$  
up to a constant factor  
(for large $n$)

Formal Definition:

For $f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+$, we say $f(n) = O(g(n))$ if there exist constants $C, n_0 > 0$ such that for all $n \geq n_0$, we have $f(n) \leq Cg(n)$  
($C$ and $n_0$ cannot depend on $n$. )
Formal Definition:

For $f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+$, we say $f(n) = O(g(n))$ if there exist constants $C, n_0 > 0$ such that for all $n \geq n_0$, we have $f(n) \leq Cg(n)$.

($C$ and $n_0$ cannot depend on $n$.)
Formal Definition:

For \( f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \), we say \( f(n) = O(g(n)) \) if there exist constants \( C, n_0 > 0 \) such that for all \( n \geq n_0 \), we have \( f(n) \leq C g(n) \) (\( C \) and \( n_0 \) cannot depend on \( n \)).

Example:

\[ f(n) = 3n^2 + 10n + 30 \quad g(n) = n^2 \]

Take \( C = 4, \quad n_0 = 13 \)

When \( n \geq 13 \), \( 10n + 30 \leq 10n + 3n = 13n \leq n^2 \).

So \( f(n) = 3n^2 + 10n + 30 \leq 3n^2 + n^2 = 4n^2 = 4g(n) \)
Big O

\[ f(n) = 3n^2 + 10n + 30 \quad g(n) = n^2 \]

\[ 4g(n) \quad f(n) \quad g(n) \]

\[ n_0 = 13 \]
Same example, different $C, n_0$:

\[ f(n) = 3n^2 + 10n + 30 \quad g(n) = n^2 \]

Take $C = 43, \quad n_0 = 1 \quad \text{When } n \geq 1,$

\[ f(n) = 3n^2 + 10n + 30 \leq 3n^2 + 10n^2 + 30n^2 \]

\[ = 43n^2 = 43g(n) \]
Big O practice

1000n is $O(n)$

0.00000001n is $O(n)$

$0.1n^2 + 10^{20}n + 10^{10000}$ is $O(n^2)$

$n$ is $O(2^n)$

10^{10}$ is $O(1)$

0.00000001$n^2$ is not $O(n)$

$n \log n$ is not $O(n)$

Note on notation:

People usually write $4n^2 + 2n = O(n^2)$

Another valid notation: $4n^2 + 2n \in O(n^2)$
<table>
<thead>
<tr>
<th>Class</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant</strong></td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Logarithmic</strong></td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>Square-root</strong></td>
<td>$O(\sqrt{n}) = O(n^{0.5})$</td>
</tr>
<tr>
<td><strong>Linear</strong></td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Quasi-linear</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Quadratic</strong></td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td><strong>Polynomial</strong></td>
<td>$O(n^k)$</td>
</tr>
<tr>
<td><strong>Exponential</strong></td>
<td>$O(2^{n^k})$</td>
</tr>
</tbody>
</table>
## Run time scaling

<table>
<thead>
<tr>
<th>Running-time:</th>
<th>Ratio:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$cn$</td>
<td>$2n$</td>
</tr>
<tr>
<td>$cn^2$</td>
<td>$(2n)^2$</td>
</tr>
<tr>
<td>$cn^3$</td>
<td>$(2n)^3$</td>
</tr>
<tr>
<td>$cn^k$</td>
<td>$(2n)^k$</td>
</tr>
<tr>
<td>$c2^n$</td>
<td>$2^{2n}$</td>
</tr>
</tbody>
</table>
Big O
### $n$ vs $2^n$

<table>
<thead>
<tr>
<th>$2^n$</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>1,048,576</td>
<td>20</td>
</tr>
<tr>
<td>1,073,741,824</td>
<td>30</td>
</tr>
<tr>
<td>1,152,921,504,606,846,976</td>
<td>60</td>
</tr>
</tbody>
</table>
Exponential running time

If your algorithm has exponential running time e.g. $\sim 2^n$

Then it is (usually) not practical.
Exponential running time: Example

Given a set of integers, determine if there is a (non-empty) subset that sum to 0.

| 4 | -3 | -2 | 7 | 99 | 5 | 1 |

Exhaustive Search (Brute Force Search):
Try every possible subset and see if it sums to 0.

Number of subsets is $2^n$

So running time is at least $2^n$
Differences between functions

\[ \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll 3^n \]
Poll

Select all that apply.

\( \log(n!) \) is:

- \( O(n) \)
- \( O(n \log n) \)
- \( O(n^2) \)
- \( O(2^n) \)

Beats me
Big Omega

\( O(\cdot) \) is like \( \leq \)

\( \Omega(\cdot) \) is like \( \geq \)

**\( O(\cdot) \)**

**Informal:** An **upper bound** that ignores **constant factors** and ignores **small n**.

**\( \Omega(\cdot) \)**

**Informal:** A **lower bound** that ignores **constant factors** and ignores **small n**.

\[ f(n) = \Omega(g(n)) \iff g(n) = O(f(n)) \]
Big Omega

Formal Definition:

For \( f, g : \mathbb{N}^+ \to \mathbb{R}^+ \) we say \( f(n) = \Omega(g(n)) \) if there exist constants \( c, n_0 > 0 \) such that for all \( n \geq n_0 \) we have \( f(n) \geq cg(n) \)

(\( c \) and \( n_0 \) cannot depend on \( n \))
Big Omega

Some Examples:

\[10^{-10} n^4 \text{ is } \Omega(n^3)\]

\[0.001 n^2 - 10^{10} n - 10^{30} \text{ is } \Omega(n^2)\]

\[n^{0.0001} \text{ is } \Omega(\log n)\]
Theta

\( O(\cdot) \) is like \( \leq \)
\( \Omega(\cdot) \) is like \( \geq \)
\( \Theta(\cdot) \) is like \( = \)
Theta

Formal Definition:
For $f, g : \mathbb{N}^+ \to \mathbb{R}^+$ we say $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

Equivalently:
There exist constants $c, C, n_0$ such that

for all $n \geq n_0$ we have $cg(n) \leq f(n) \leq Cg(n)$
Some Examples:

\[ 0.001n^2 - 10^{10}n - 10^{30} \text{ is } \Theta(n^2) \]

\[ 1000n \text{ is } \Theta(n) \]

\[ 0.00001n \text{ is } \Theta(n) \]
Putting everything together

Now we really understand what this means:

“The (asymptotic) complexity of algorithm A is $O(n^2)$”
(which means $T_A(n) = O(n^2)$.)

Make sure you are specifying:

- the computational model
  > what constitutes a *step* in the model
- the length of the input
Goals for the today

1. What is the right way to study complexity?
   - using the right language and level of abstraction
   - upper bounds vs lower bounds
   - polynomial time vs exponential time

2. Appreciating the power of algorithms.
   - analyzing some cool (recursive) algorithms
What is the running time as a function of input length?

- logarithmic
- linear
- log-linear
- quadratic
- exponential
- beats me

def isPrime(n):
    if (n < 2):
        return False
    for factor in range(2, n):
        if (n % factor == 0):
            return False
    return True
Poll Answer

def isPrime(n):
    if (n < 2):
        return False
    for factor in range(2,n):
        if (n % factor == 0):
            return False
    return True

# iterations: \approx n

n = 2^{\log_2 n} = 2^{\text{len}(n)}
```python
def sum(A, B):
    for i from 1 to B do:
        A += 1
    return A
```

What is the running-time of this algorithm?
Integer Addition

\[
\begin{array}{c}
36185027886661311069865932815214971104 \quad \text{A} \\
+ \quad 65743021169260358536775932020762686101 \quad \text{B} \\
\hline
101928049055921669606641864835977657205 \quad \text{C}
\end{array}
\]

\# steps to produce \( C \) is \( O(n) \)
**Integer Multiplication**

**Input:** 2 $n$-digit numbers $x$ and $y$.

**Output:** The product of $x$ and $y$.

**Grade-School Algorithm:**

\[
\begin{array}{c}
5 6 7 8 \\
\times 1 2 3 4 \\
\hline
2 2 7 1 2 \\
1 7 0 3 4 \\
1 1 3 5 6 \\
\hline
7 0 0 6 6 5 2
\end{array}
\]

$n$ rows

$\rightarrow O(n)$ operations

$\rightarrow O(n)$ operations

$\rightarrow O(n)$ operations

$\rightarrow O(n)$ operations

**Total:** $O(n^2)$
You might think:

Probably this is the best, every must be digit multiplied with every other?

However: you can always do better (until you can prove you can’t)

What algorithm does Python use?
Integer Multiplication

\[
x = \begin{array}{c} a \\ b \end{array}
\]
\[
y = \begin{array}{c} c \\ d \end{array}
\]

\[
x \cdot y = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d)
\]
\[
= ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd
\]

Use recursion!
**Integer Multiplication**

\[ x = \begin{array}{cc} a & b \\ \hline 5 & 6 \\ 7 & 8 \end{array}, \quad y = \begin{array}{cc} c & d \\ \hline 1 & 2 \\ 3 & 4 \end{array} \]

\[ x = a \cdot 10^{n/2} + b \]
\[ y = c \cdot 10^{n/2} + d \]

\[ x \cdot y = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) \]
\[ = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd \]

- Recursively compute \( ac, ad, bc, \) and \( bd \).
- Do the multiplications by \( 10^n \) and \( 10^{n/2} \) \( O(n) \)
- Do the additions. \( O(n) \)

\[ T(n) = 4T(n/2) + O(n) \]
Integer Multiplication

Level 0

Level 1

Level 2

# distinct nodes at level j: \(4^j\)

work done per node at level j: \(cn/2^j\)

# levels: \(\log_2 n\)

Total cost: \(\sum_{j=0}^{\log_2 n} cn2^j \in O(n^2)\)
Hmm, we don’t really care about $ad$ and $bc$. We just care about their sum. Maybe we can get away with 3 recursive calls.
Integer Multiplication

\[ x \cdot y = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) \]
\[ = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd \]

\[ (a + b)(c + d) = ac + ad + bc + bd \]

- Recursively compute \( ac, bd, (a+b)(c+d) \).
- \( ad + bc = (a+b)(c+d) - ac - bd \)

\[ T(n) \leq 3T(n/2) + O(n) \quad \text{Is this better?} \]
**Integer Multiplication**

- **Level 0:**
  - $n$

- **Level 1:**
  - $n/2$ (3 nodes)

- **Level 2:**
  - n/4 (9 nodes)

# distinct nodes at level $j$: $3^j$

work done per node at level $j$: $c(n/2^j)$

# levels: $\log_2 n$

Total cost: $\sum_{j=0}^{\log_2 n} c(n/2^j)$

$cn(3^j/2^j)$ per level
Integer Multiplication

Level
0

1

2

Total cost:

$$\sum_{j=0}^{\log_2 n} cn\left(\frac{3^j}{2^j}\right) \leq Cn\left(3^{\log_2 n} / 2^{\log_2 n}\right)$$

$$= C3^{\log_2 n}$$

$$= Cn^{\log_2 3} \in O(n^{\log_2 3})$$

Karatsuba Algorithm
You might think:

Probably this is the best, what else can you really do?

Again: you can always do better

Cut the integer into 3 parts of length $n/3$ each.
Replace 9 multiplications with only 5.

\[
T(n) \leq 5T(n/3) + O(n) \\
T(n) \in O(n^{\log_3 5})
\]

Can do $T(n) \in O(n^{1+\epsilon})$ for any $\epsilon > 0$. 
Integer Multiplication

Fastest known: \( n(\log n)^{2^{O(\log^* n)}} \)  

Martin Fürer  
(2007)