An algorithm is a finite group of instructions that provides a step-by-step procedure for solving a problem involving...

**Computation**
- Arithmetic results
- Matrix operations
- Numerical analysis—improving accuracy

**Data Organization**
- Searching
- Sorting (alphabetization, ordering a group of #s)

**Optimization**
- Using mathematical models to find the best solution to a real-world problem.
- Graph theory
  - Find shortest paths, efficient networks.

  **Knapsock Problem**
What is an “efficient” algorithm?
- Fast
- Takes up little memory/space

Let \( n \) = the size of the data set of interest (input for the algorithm)

\( \text{Ex: } \) Searching, Sorting: \( n \) = # items in a database
\( \text{Ex: } n \) could be the “size” of a square matrix in a computation problem

Let \( f(n) \) = the “amount” of time or memory used by a particular algorithm to deal with the “worst-case scenario” involving a data set of size \( n \).

“Worst-case” analyses tend to be easier, more commonly used than other analyses.
Also: “Best-case”, “Average-case”

“Worst-case” analyses tend to be easier, more commonly used than other analyses.
Also: “Best-case”, “Average-case”

A single, really bad “pathological case” might skew a worst-case analysis “unfairly.”

\[
\begin{array}{c|c|c}
\text{1} & 12 & \text{worst-case for data size 1} \\
\text{2} & 40 & \text{2} \\
\text{3} & 84 & \text{3} \\
\end{array}
\]
Let's say $f(n)$ = worst-case running time for size $n$ data.

Assume $f(n), g(n), \ldots$ are always $> 0$.

How do we measure running times?
- Actual time in seconds, hours, \ldots on a specific computer.
- # of simple program statements/condition checks/comparisons (searching, sorting), etc.

We want $f(n)$ to be low.

Examples

1. $f(n) = 2$ (seconds) for all $n \in \mathbb{Z}^+$
   - Great! Great!

   The worst-case running time is a nice constant function of $n$. (All data sets)
   - We wish!

2. $f(n) = 2 + \lfloor \sqrt{n} \rfloor$
   - Great! Great!

   grows very slowly with $n$
1 million secret 12 days?

3. $f(n) = \left(\text{million} + \sqrt[n]{n}\right)$
   - expensive
   - fixed (start-up cost)

Yuk! Great!
Focuses on the slow growth

4. $f(n) = \left(\text{million} + (1 \text{ billion})^{\sqrt[n]{n}}\right)$

Yuk! Great!

This factor is less relevant as $n \to \infty$. "Not much" when $n = 10^{10,000}$ relative speaking.
"Big-O" Notation

\[ f(n) \text{ is } O(g(n)) \iff \]
\[ f \text{ is eventually bounded above by some constant multiple of } g(n) \iff \]
\[ f(n) \leq Cg(n) \text{ for some constant } C \]
\[ \text{and for "large enough" } n \implies n > n_0 \text{ for some } n_0 \]

Ex

\[ f(n) \leq 17\sqrt{n} \text{ for all } n > n_0 \]

So, \( f(n) \) is \( O(\sqrt{n}) \)

Big-O: "upper bound idea"

Big-\( \Omega \) (omega): "lower bound idea" \( 17\sqrt{n} \) is \( \Omega(f(n)) \)

Big-\( \Theta \) (theta): combines the two
\[ f(n) \text{ is } \Theta(g(n)) \iff \]

\[ f \text{ is eventually bounded above and below by constant multiples of } g(n) \equiv \]

\[ c_1 g(n) \leq f(n) \leq c_2 g(n) \]

\[ f(n) \text{ is } \Omega(g(n)) \quad f(n) \text{ is } O(g(n)) \]

for some constants \( c_1, c_2 \) and for "large enough" \( n \)

**Ex**

\[ \begin{array}{c}
\text{Ex}
\end{array} \]

\[ n_0 \]

\[ 10n^2 \]

\[ \frac{1}{2}n^2 \]

\[ f(n) \text{ is } \Theta(n^2) \]

We say that \( f(n) \) is "order" \( n^2 \).
Since we often use worst-case analyses, "O" is especially useful.

Well-known complexity classes:
(from best to worst)

$O(1)$  constant complexity  
\( \text{like } f(n) = 2 \)

$O(\log n)$  logarithmic  
\( O(\log n) = O(\log_{10} n) = O(\log_b n) = 0(\log_b n), b > 1 \)

$O(n^b), 0 < b < 1$  
Ex \( n^{1/2} = \sqrt{n}, n^{2/3} = \sqrt[3]{n}, ... \)

$O(n)$  linear  

$O(n \log n)$

$O(n^b), b > 1$  polynomial  
Ex \( 4n^3 + 6n^2 - n + 2 \) is \( O(n^3) \)  
\( \text{focus on leading term} \)  
\( \text{Also } O(n^4), ... \)

$O(2^n), b > 1$  exponential  
\( \text{We want the highest function} \)

$O(n!)$

$O(n^n)$
$\Theta(1) \subset \Theta(\log n) \subset \Theta(n^k)$: $\Theta(\log n) < \Theta(n^k)$

Theory: worse
(As $n \to \infty$, asymptotic behavior)

Which is better:

$f(n) = S_n^2$

is $\Theta(n^2)$

Better for small $n$

$g(n) = 1,000,000n$

is $\Theta(n)$

Eventually ($n > 200,000$), this is better.

$s_n^2 = \frac{1}{n} \cdot x_n$

at $n = 100,000$

$s_n^2$ and $1,000,000n$ scale differently.
**Rules for \( O \)**

**Sum Ex** \( 3n^4 + 7n^2 + 5\log n \) is \( O(n^4) \)

\[
\begin{align*}
&O(n^4) \\
\uparrow & \quad O(n^2) \\
&O(\log n)
\end{align*}
\]

Take the "biggest" function in the sum

**Product Ex**

\[
(2^n + n^5)(3n - 1) \quad \text{is} \quad O(n2^n)
\]

\[
\begin{align*}
&O(2^n) \\
\downarrow & \quad O(n)
\end{align*}
\]

When we multiply "pieces", we multiply their \( O \)-functions.

**Note** "\( \Theta \) means "order"**

Ex \( 4n^2 + 2n \) is \( \Theta(n^2) \)

are "roughly comparable"

Sloppy books often say "\( O \) means "order."
Cute Proofs

Example 1: \[1 + 2 + \ldots + n \leq n + n + \ldots + n = n^2\]

\[f(n) \leq n^2 \quad \text{In fact, true for all } n \in \mathbb{Z}^+\]

So, \(f(n)\) is \(O(n^2)\)

In fact, \[1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \frac{n^2 + \frac{1}{2}n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n\]

So, \(f(n)\) is \(\Theta(n^2)\), also.

Example 2: \(f(n) = n!\)

\[n! = 1 \cdot 2 \cdot 3 \ldots n \leq n \cdot n \cdot n \ldots n = n^n\]

So, \(n! \leq n^n \quad \text{In fact, true for all } n \in \mathbb{Z}^+\)

So, \(f(n)\) is \(O(n^n)\)

In fact, (Stirling's approx.)
\[n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + o(1)) \quad \text{for big } n\]

Can omit useful for proofs involving \(n!\)
SEARCHING ALGORITHMS

Linear

Look for \( x \)

[Diagram of linear search with stop condition]

\( O(n) \)

same amount of work at each step

Binary Search (assume sorted already!)

\( n = 2^k \)

[Diagram of binary search]

\( k \) steps (+1)

\( n = 2^k \)

\( k = \log n \)

\( O(\log n) \)
Searching a sorted array - $\Theta(\log n)$

Ex Binary Search

Searching an unsorted array - $\Theta(n)$

Ex Linear Search

Sorting an array - $\Theta(n \log n)$

Ex Mergesort

Merge 2 sorted arrays by comparing leftmost elements