Algorithms
CS 330: Discrete Structures
**Algorithm**: a sequence of instructions that describes, unambiguously, how to solve a problem in a finite amount of time.

Questions related to algorithms:

- How do we specify them?
- Does one exist to solve a given problem?
- Does it always give the correct answer?
- How long does it take?
- How much memory does it take?
- Does it complete in a reasonable amount of time?
- How do we design them?

- Code (language or paradigm)?
- Solvability/decidability
- Correctness (proofs?)
- Time & space complexity
- Tractability (P=NP)

Many algorithmic paradigms to help!

(e.g., brute force, greedy, divide+conquer)
e.g., finding the max of two numbers $x, y$:

```python
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

correct? (we proof by cases)
e.g., finding the max of a sequence of values $a_1, a_2, \ldots, a_n$

```python
def max_seq([a_1, a_2, \ldots, a_n]):
    m = a_1
    for i in [2 \ldots n]:
        m = max(m, a_i)
    return m
```

Correct? based on "loop invariant"—at end of

each for loop, $m$ holds max

of $a_1, \ldots, a_i$, and $i = n$ at end.
def insertion_sort([a_1, a_2, ..., a_n]):
    for i ← [2..n]:
        for j ← [i..2]:
            if a_{j-1} > a_j:
                swap a_{j-1}, a_j
            else:
                break
    correct? also based on loop invariant, [a_1, ..., a_i] is sorted at end of each inner loop, and i = n at end
e.g., determine if some program $P$ will run forever or eventually terminate (halt) on some input $X$

i.e., implement the function $H(P, x) = \begin{cases} 
\text{True} & \text{if } P(x) \text{ halts} \\
\text{False} & \text{if } P(x) \text{ runs forever}
\end{cases}$

- suppose $H$ exists

- we can implement the function $G$, like so:

```python
def G(P):
    if H(P, P):
        loop_forever()
    else:
        return
```

- what is the result of $H(G, G)$?

- contradiction! $H$ cannot exist (we cannot implement it!)
how long does it take an algorithm to complete?

trivial cases: 1) algorithm has no variables
   (e.g., compute $100! = 100 \times 99 \times 98 \times \ldots \times 1$)
   — constant runtime (can run once to get estimate)

2) algorithm performs the same # of operations
   for all inputs
   (e.g., $\max(1, 2) \cdot \max(100, 2000) \cdot \max(100000, 2000000)$)
   — constant runtime (can run on any set of inputs to get estimate)
how long does it take an algorithm to complete?

variable runtime is dependent on:

- speed of execution environment  
  - machine dependent (we ignore these factors when estimating theoretical runtime)

- number of operations executed
- cost per instruction
- size of input(s)

determined by
how long does it take an algorithm to complete?

i.e., for some algorithm with inputs $i_1, i_2, ..., in$, come up with the function $T(i_1, i_2, ..., in)$ that computes the number of operations carried out by the algorithm for the given inputs.

$T$ represents the computational/runtime complexity of the algorithm.
e.g., find timing function for max-seq:

```python
def max-seq([a₁, a₂, ..., aₙ]):  # times
    m = a₁
    for i ← [2..n]:
        m = max(m, aᵢ)
    return m
```

\[ T(n) = 3(n-1) + 2 = 3n - 1 \]

↑ length of input list.
e.g., find timing function for insertion sort:

```python
def insertion_sort([a1, a2, ..., an]):  # times
    for i in [2..n]:  # n-1
        for j in [i..2]:  # 1+2+...+(n-1)
            if aj-1 > aj:
                swap aj-1, aj  # "
                else:
                    break  # worst-case analysis!
```

(average case analysis is also useful, but often more difficult)
arithmetic series: \( 1 + 2 + \ldots + n = ? \)

e.g., \( 1 + 2 + 3 + 4 + 5 = 15 \)

\[
= 3 \times 5 \quad \text{(avg. value \times \# terms)}
\]

\[
= \left( \frac{5+1}{2} \right) \times 5
\]

i.e., \( 1 + 2 + \ldots + n = \frac{(n+1) \cdot n}{2} \)

\[
\therefore 1 + 2 + \ldots + (n-1) = \frac{(n-1)+1) \cdot (n-1)}{2}
\]

\[
= \frac{n(n-1)}{2}
\]
e.g., find timing function for insertion sort:

```python
def insertion_sort([a_1, a_2, ..., a_n]):  # times
    for i ← [2..n]:
        for j ← [i..2]:
            if a_{i-1} > a_j:
                swap a_{i-1}, a_j
            else:
                break
    T(n) = n - 1 + 3 \cdot \frac{n(n-1)}{2} = \frac{3n^2}{2} - \frac{n}{2} - 1
```
so we have:

\[ T_{\text{max-seq}}(n) = 3n - 1 \]

\[ T_{\text{transition-sort}}(n) = \frac{3n^2}{2} - \frac{n}{2} - 1 \]

we often focus on the behavior of algorithms as inputs grow large, i.e., "asymptotic" runtime complexity.

- as inputs grow large, we can ignore slower-growing terms and constants of our runtime function
- formalized in "big-Oh" notation
If $f(x)$ is $O(g(x))$ for $x \in \mathbb{R}$, then

\[
\exists C, x_0 \in \mathbb{R}, \forall x \geq x_0 \left( |f(x)| \leq C \cdot g(x) \right)
\]

This is known as the "abuse of notation." When dealing with strictly positive functions, we can lose the abs. operator.
Show that \( x^2 + 2x + 1 \in O(x^2) \)

- Need to find witnesses \( C, x_0 \) where \( \forall x \geq x_0 \left( x^2 + 2x + 1 \leq Cx^2 \right) \)

- Simplification: we know that \( \forall x \geq 1 \left( (x \leq x^2) \land (1 \leq x^2) \right) \)

\[
\forall x \geq 1 \left( x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 \right)
\]

\[
\forall x \geq 1 \left( x^2 + 2x + 1 \leq 4x^2 \right)
\]

Our witnesses \( C = 4, x_0 = 1 \)

- \( \forall x \geq 1 \left( x^2 + 2x + 1 \leq 4x^2 \right) \Rightarrow x^2 + 2x + 1 \in O(x^2) \)

- Any larger \( C \) or \( x_0 \) will also serve as witnesses!
for polynomial $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

where $a_n, a_{n-1}, \ldots, a_0 \in \mathbb{R}$ and $a_n \neq 0$, $f(x) \in \mathcal{O}(x^n)$

i.e., the term with the highest exponent (aka the degree) of a polynomial dominates its growth, asymptotically (for large x)
returning to the analysis of our algorithms:

\[ T_{\text{max-seq}}(n) = 3n - 1 \leq O(n) \]

\[ T_{\text{is\_insertion\_sort}}(n) = \frac{3n^2}{2} - \frac{n}{2} - 1 \leq O(n^2) \]

i.e., the worst-case asymptotic runtime complexity of max-seq is bounded by a linear function, and insertion sort "..." polynomial (of degree 2) function
\[ n! \in O(?), \quad n! \in O(n^n) \text{ for } C=1, n_0=1 \]

\[ n! = n \times (n-1) \times \cdots \times 2 \times 1 \]

\[ \leq n \times n \times \cdots \times n \times n \]

\[ \leq n^n \]

\[ \log(n!) \in O(?), \quad \log(n!) \in O(n \log n) \text{ for } C=1, n_0=1 \]

\[ \log(n!) \leq \log(n^n) \]

\[ \leq n \log n \]
useful big-O relationships:

\( \forall (a, b \in \mathbb{R}^+) \text{ and } (a > b > 1) \):

\[ x^b \in O(x^a), \text{ but } x^a \notin O(x^b) \]
\[ b^x \in O(a^x), \text{ but } a^x \notin O(b^x) \]

\( \forall (a, b, c \in \mathbb{R}^+) \text{ and } (a > 1) \):

\[ x^b \in O(a^x), \text{ but } a^x \notin O(x^b) \]
\[ (\log_a x)^b \in O(x^c), \text{ but } x^c \notin O((\log_a x)^b) \]
combinations of functions:

\[ \text{if } f_1(x) \in O(g_1(x)) \text{ and } f_2(x) \in O(g_2(x)) \text{ then } f_1(x) + f_2(x) \in O(\max(g_1(x), g_2(x))) \]

\[ \text{if } f_1(x) \in O(g_1(x)) \text{ and } f_2(x) \in O(g_2(x)) \text{ then } f_1(x) \cdot f_2(x) \in O(g_1(x) \cdot g_2(x)) \]
T/F?

42 ∈ \(O(3)\) ✓

\(4x + 100 \leq O(x)\) ✓

\(3x^2 - 42x \leq O(100x)\) ×

\(6x^2 + \log x \leq O((\log x)^3)\) ×

\(x + x \log x \leq O(x)\) ×

\(x^{100} \leq O(2^x - 1000)\) ✓
NB: big-O notation only describes an upper bound.

e.g.; if two algorithms have runtime complexities \( \in O(n^2) \), it is possible for them to have very different asymptotic behavior (one may be linear, and the other constant time!).
\[ f(x) = \Omega(g(x)) \quad \text{for} \quad x \in \mathbb{R} \]

\[ \iff \exists C, x_0 \in \mathbb{R} \]

where \( \forall x \geq x_0 \left( |f(x)| \geq C \cdot g(x) \right) \)

\[ f(x) \]
\[ g(x) \]
\[ c \cdot g(x) \]
\[ x \]
\[ x_0 \]
\[ f(x) = \Theta(g(x)) \quad \text{or} \quad \left\{ \begin{array}{l} f(x) \in \Theta(g(x)) \\ f(x) = \Theta(g(x)) \end{array} \right. \]

\[ \iff f(x) \text{ is } O(g(x)) \text{ and } f(x) \text{ is } \Omega(g(x)) \]

i.e., if \( \exists C_1, C_2, x_0 \in \mathbb{R} \)

where \( \forall x \geq x_0 \left(C_1 g(x) \leq |f(x)| \leq C_2 g(x)\right) \)

\[ f(x) \text{ “sandwich” on } g(x) \text{ bread} \]
if $f(x) \in \Theta(g(x))$ we say that $f(x)$ is of order $g(x)$, or that $f(x)$ and $g(x)$ are of the same order.

when possible, we prefer to compare the order of different algorithms over their big-O functions. Many texts/authors mistakenly use $O$-notation when they mean/imply $\Theta$-notation.

but remember, $\Theta$ doesn't tell us the whole story!

two algorithms that are both $\Theta(x^2)$ can still have very different runtimes in practice! (Constants matter)
rank these functions in increasing (slowest to fastest) order of growth:

\[
\begin{align*}
6. & \quad f_6(x) = 8x^3 + 12x^2 - 13 \\
9. & \quad f_2(x) = 42^x(x^2 + 1) \\
3. & \quad f_3(x) = (\log x)^2 \\
1. & \quad f_4(x) = 2400 \\
10. & \quad f_5(x) = x! \\
8. & \quad f_6(x) = 42^x \\
5. & \quad f_7(x) = \log(\log x) \\
4. & \quad f_8(x) = x^2(\log x)^3 \\
7. & \quad f_9(x) = 32x + 1000 \\
7. & \quad f_{10}(x) = 1.5^x
\end{align*}
\]
classes of problems & complexity of algorithms that solve them:

- "tractable" problem: there is a polynomial time solution (class "P")

- "intractable" problem: there is no known polynomial time solution (doesn't necessarily mean it doesn't exist!)

- "unsolvable" problem: no solution can exist (e.g., halting problem)

- class "NP": problems whose solutions can be checked in polynomial time, but for which no known polynomial time solutions exist

- class "NP-complete": all problems in this class can be "transformed" into each other, and also belong to class NP
$P = NP$?

i.e., for all problems whose solutions can be verified in polynomial time, does there exist an algorithm for solving it in polynomial time?

- **Millennium Prize**
  - \$1,000,000

- **PHEW!**

- **EEK!**

- If $P = NP$, many problems currently considered intractable could be solved relatively quickly! (e.g., decrypting/deciphering data could be done relatively easily)

- **YAY!**

- If $P \neq NP$ (which most folks believe), we can stop trying to find efficient exact solutions, and focus on partial/approximation algorithms.

- **DANG IT!**