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?

(c.e., brute-force, greedy, divide + conquer)

Correctness (proofs!)
Solvability/decidability
Time & space complexity
Tractability (P=NP)
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 range(2, n):
        m = max(m, a_i)
    return m
```
correct? (show proof by induction)
def insertion_sort([a_1, a_2, ..., a_n]):
    for i ← [2..n]:
        for j ← [i..2]:
            if a_{i-1} > a_j:
                swap a_{i-1}, a_j
            else:
                break
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., $\max(1,2), \max(1000,2000), \max(1000000,2000000)$)
   — 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
- Number of operations executed
- Cost per instruction
- Size of input(s)

Machine dependent (we ignore these factors when estimating theoretical runtime)
how long does it take an algorithm to complete?

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

\( T \) represents the computational/runtine complexity of the algorithm.
e.g., find timing function for max-sep:

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

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

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

```
def insertion_sort([a1, a2, ..., an]):  # times
    for i ← [2..n]:  ← n - 1
        for j ← [i..2]:  ← 1 + 2 + ... + (n-1)
            if a_{j-1} > a_j:  ← "
                swap a_{j-1}, a_j  ← "
            else:
                break
```

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

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

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

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

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

\:: \: 1 + 2 + \ldots + (n-1) = \left(\frac{(n-1)+1}{2}\right) \cdot (n-1)

\[
= \frac{n(n-1)}{2}
\]
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_{j-1} > a_j:
                #
                swap a_{j-1}, a_j
            else:
                break

    T(n) = n - 1 + \frac{3 \cdot n(n-1)}{2} = \frac{3n^2 - n}{2} - 1
```
so we have:

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

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-O" notation
\[ f(x) = O(g(x)) \quad \text{or} \quad \begin{cases} f(x) \in O(g(x)) \\ f(x) = O(g(x)) \end{cases} \]

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

\[ \forall x \geq x_0 \left( |f(x)| \leq C \cdot g(x) \right) \]

"abuse" of notation!

known as "witnesses" (there are infinitely many)

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

when dealing with strictly positive functions, we can lose the abs. operator.
Show that \( x^2 + 2x + 1 \in \Theta(x^2) \)

- Need to find witnesses \( C, x_0 \) where \( \forall x \geq x_0 \ (x^2 + 2x + 1 \leq Cx^2) \)
- Simplification: we know that \( \forall x \geq 1 \ ((x \leq x^2) \land (1 \leq x^2)) \)
  
  \[ \forall x \geq 1 \ (x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2) \]
  
  \[ \forall x \geq 1 \ (x^2 + 2x + 1 \leq 4x^2) \]

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

\[ \forall x \geq 1 \ (x^2 + 2x + 1 \leq 4x^2) \Rightarrow x^2 + 2x + 1 \in \Theta(x^2) \]

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

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

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

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

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

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

\[ n! = n \times (n-1) \times \ldots \times 2 \times 1 \leq n \times n \times \ldots \times n \times n \leq n^n \]

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

\[ \log(n!) \leq \log(n^n) \leq n \log n \]
useful big-O relationships:

∀(a, b ∈ ℝ⁺) and (a > b > 1):

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

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

∀(a, b, c ∈ ℝ⁺) and (a > 1):

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

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