Algorithms

CS 330 : Discrete Structures

def. Algorithm: a sequence of instructions that describes, unambiguously, how to solve a problem in a finite amount of time.

English (carefully!)
pseudocode
code (language/paradigm)?

questions related to algorithms:

- how do we specify them? — English (carefully!) / pseudocode / code (language/paradigm)?
- does one exist to solve a given problem? — solvability/decidability
- does it always give the correct answer? — correctness (proofs!)
- how long does it take?
- how much memory does it take?

$\}$ — time & space complexity

- does it complete in a reasonable amount of time? — tractability (P=NP)
- how do we design them? — many algorithmic paradigms to help! (e.g., brute-force, greedy, divide+conquer)

e.g., finding the max of two numbers x, y :

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

correct? (use proof by cases)

e.g., finding the max of a sequence of values $a_1, a_2, \ldots a_n$

```
def max_seq([a_1, a_2, ..., a_n]):
    m ← a_1
    for i ← [2 .. 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.

e.g., Insertion sort

```
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, $[a_1, \ldots, a_i]$ 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:

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

i.e., G does the opposite of what H reports about P

- what is the result of H(G, G)?
- <u>contradiction</u>! 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 $100! = 100 \times 99 \times 98 \times .. \times 1$)

— 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)

determined by

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 # 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 :

```
def max_seq([a_1, a_2, ..., a_n]):      # times
    m = a_1                             ← 1
    for i ← [2 .. n]:                   ← n - 1
        m = max(m, a_i)                 ← T_max(n-1) ≈ 2(n-1)
    return m                            ← 1
```

$T(n) = 3(n-1) + 2 = 3n - 1$

length of input list.

e.g., find timing function for insertion_sort:

```
def insertion_sort([a_1, a_2, ..., a_n]):      # 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~~
```

$\boxed{\text{worst-case analysis!}}$

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

e.g., $1+2+3+4+5 = 15$

$= 3 \times 5$ (avg. value × # terms)

$= \frac{(5+1)}{2} \times 5$

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

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

$= \frac{n(n-1)}{2}$

e.g., find timing function for insertion_sort:

```
def insertion_sort([a_1, a_2, ..., a_n]):        # times
    for i <- [2..n]:                  <-- n - 1
        for j <- [i..2]:              <-- 1 + 2 + ... + (n-1) = n(n-1)/2
            if a_{j-1} > a_j:         <-- "
                swap a_{j-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 :

$$T_{max\_seq}(n) = 3n - 1$$

$$T_{insertion\_sort}(n) = \frac{3n^2}{2} - \frac{n}{2} - 1$$

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-Oh" notation

$f(x)$ is $O(g(x))$ $\left(\text{or } \begin{cases} f(x) \in O(g(x)) \\ f(x) = O(g(x)) \end{cases} \right)$

$x \in \mathbb{R}$

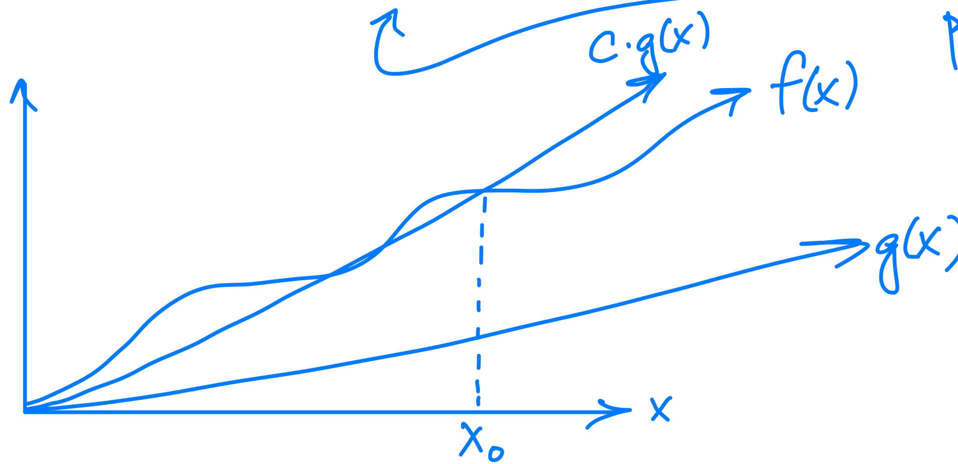
"abuse" of notation!

$\Longleftrightarrow \exists \underbrace{C, x_0}_{} \in \mathbb{R}$

known as "witnesses" (there are infinitely many)

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

when dealing w/ 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 $\forall x \geq 1 \left((x \leq x^2) \wedge (1 \leq x^2) \right)$

$$\therefore \forall x \geq 1 \left(x^2+2x+1 \leq x^2+2x^2+x^2 \right)$$

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

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 $a_n \neq 0$, $f(x) \in O(x^n)$

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:

$$T_{max\_seq}(n) = 3n - 1 \in O(n)$$

$$T_{insertion\_sort}(n) = \frac{3n^2}{2} - \frac{n}{2} - 1 \in O(n^2)$$

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

$n! \in O(?)$ $\quad n! \in O(n^n)$ for $c=1, n_0=1$

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

$\log(n!) \in O(?)$ $\quad \log(n!) \in O(n \log n)$ for $c=1, n_0=1$

$$\log(n!) \leq \log(n^n)$$
$$\leq n \log n$$

useful big-O relationships:

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

$x^b \in O(x^a)$, but $x^a \notin O(x^b)$

$b^x \in O(a^x)$, but $a^x \notin O(b^x)$

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

$x^b \in O(a^x)$, but $a^x \notin O(x^b)$

$(\log_a x)^b \in O(x^c)$, but $x^c \notin O((\log_a x)^b)$

combinations of functions:

- if $f_1(x) \in O(g_1(x))$ and $f_2(x) \in O(g_2(x))$

  then $f_1(x) + f_2(x) \in O(\max(g_1(x), g_2(x)))$

- if $f_1(x) \in O(g(x))$ and $f_2(x) \in O(g(x))$

  then $f_1(x) + f_2(x) \in O(g(x))$

- if $f_1(x) \in O(g_1(x))$ and $f_2(x) \in O(g_2(x))$

  then $f_1(x) \cdot f_2(x) \in O(g_1(x) \cdot g_2(x))$

T/F?

$42 \in O(3)$ ✓

$4x+100 \in O(x)$ ✓

$3x^2 - 42x \in O(100x)$ ✗

$6x^2 + \log x \in O((\log x)^3)$ ✗

$x^2 - 3^x \in O\left(\frac{x^3}{100}\right)$ ✗

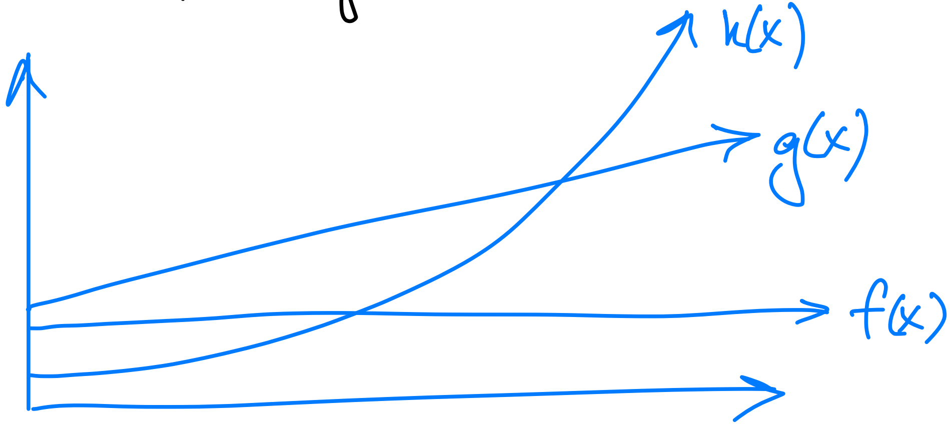
$\log x \in O(\sqrt{x})$ ✓

$x + x \log x \in O(x)$ ✗

$5x^{100} \in O(2^x - 1000)$ ✓

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!)



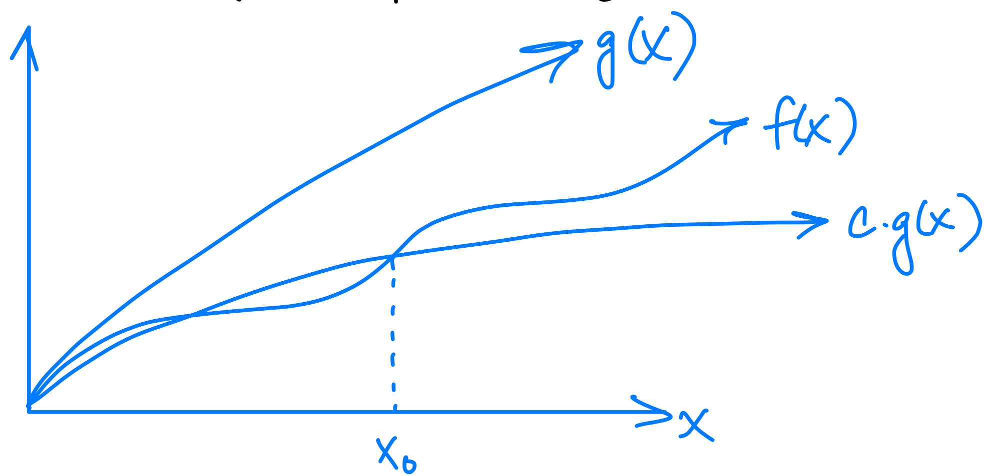
$f(x) \in O(h(x))$
$g(x) \in O(h(x))$

$f(x)$ is $\Omega(g(x))$ $\left(\text{or } \begin{cases} f(x) \in \Omega(g(x)) \\ f(x) = \Omega(g(x)) \end{cases} \right)$

$x \in \mathbb{R}$

$\iff \exists\, c, x_0 \in \mathbb{R}$

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



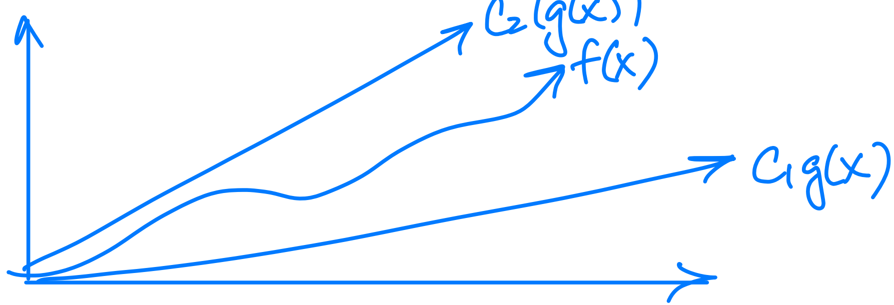
$f(x)$ is $\Theta(g(x))$ $\left(\text{or } \begin{cases} f(x) \in \Theta(g(x)) \\ f(x) = \Theta(g(x)) \end{cases}\right)$

$x \in \mathbb{R}$

$\iff$ $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$

i.e., if $\exists\, c_1, c_2, x_0 \in \mathbb{R}$

where $\forall x \geq x_0 \left(c_1 g(x) \leq |f(x)| \leq c_2 g(x) \right)$



$f(x)$ "sandwich" on $g(x)$ 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 Θ-notation.

but remember, Θ 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:

6 $f_1(x) = 8x^3 + 12x^2 - 13$

9 $f_2(x) = 42^x(x^2+1)$

3 $f_3(x) = (\log x)^2$

1 $f_4(x) = 2400$

10 $f_5(x) = x!$

8 $f_6(x) = 42^x$

2 $f_7(x) = \log(\log x)$

5 $f_8(x) = x^2(\log x)^3$

4 $f_9(x) = 32x + 1000$

7 $f_{10}(x) = 1.5^x$

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

MILLENNIUM PRIZE
$1,000,000

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?

PHEW!

EEK!

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

YAY!

if P ≠ NP (which most folks believe), we can stop trying to find efficient exact solutions, and focus on partial / approximation algorithms.

DANG IT!