Lecture 6:

1. Append + reverse
2. Predicting running time of programs

```
(* Purpose: output a list with
        the same elements in
        the opposite order! *)

(* E.g.   reverse [1,2,3,4] = [4,3,2,1]
                                    *)

fun reverse(l: int list): int list =
    case l of
        [] => []
                        put x at the back end of
      | x::xs => append(reverse(xs), [x])
```

"Induction on an example"

reverse [1, 2, 3, 4] should be [4, 3, 2, 1]

template structural recursion

??? put 1 at the end

reverse [2, 3, 4] $\xrightarrow{\text{IH}}$ [4, 3, 2]

put 2 at end

reverse [3, 4] $\longrightarrow$ [4, 3]

$\cancel{xs :: x}$

$[2,3] :: 1$

$\stackrel{?}{=} [2,3,1]$???

$x :: xs$

$1 :: [2,3] = [1,2,3]$

(* Purpose: make a new list
with all the elts of one list
before all the elts of another
*)

(* E.g. append([4,3], [1,2]) = [4,3,1,2]
*)

```
fun append(l1 : int list, l2 : int list) : int list =
    case l1 of
        [] => l2
      | x :: xs => x :: append(xs, l2)
```

[built in @   l1 @ l2   append(l1, l2)] append([4, 3], [1, 2]) $\xrightarrow{\text{should be}}$ [4, 3, 1, 2]

4 :: 3 :: 1 :: 2 :: []

append([3], [1, 2]) $\xrightarrow{\text{IH}}$ [3, 1, 2]

put 4 at the front

Analyzing/predicting running time

resources

1. running time
2. space/memory
3. network traffic
4. power

→ make predictions based on "size" of the data

1. program $\longrightarrow$ recurrence

2. recurrence $\longrightarrow$ closed form

3. closed form $\longrightarrow$ big-O

Size: length of l_1 and/or
length of l_2

Predict: number of steps append
takes

$\text{append}([\,], l_2)$

$\mapsto \text{case } [\,] \text{ of } [\,] \Rightarrow l_2 \mid \ldots$

$\mapsto l_2$

2 steps

$\text{append}(x :: xs, l_2)$

$\mapsto \text{case } x :: xs \text{ of } [\,] \Rightarrow l_2 \mid x :: xs \Rightarrow x :: \text{append}(xs, l_2)$

$\mapsto x :: \text{append}(xs, l_2)$

$\mapsto \vdots$

$\mapsto \vdots$

$\mapsto \vdots$

$\mapsto x :: v$ → value of recursive call

2 steps plus steps of recursive call

cost recurrence: math function from the size of the input to the cost (#steps)

work

$Work_{append}(n)$ — length of l_1

$$W_{append}(0) = 2$$

length of input

$$W_{append}(n) = 2 + \underbrace{W_{append}(n-1)}_{\text{recursive call}}$$

$n \neq 0$

Closed form: non-recursive def. of the same function

$$\text{Closed form}(n) = 2n + 2$$

Theorem $\text{Wopperd}(n) = 2n+2$ closed form

Proof. Ind. on n

Case for 0

$\text{Wopperd}(0) = 2 = 2\cdot 0+2$ ✓

Case 1+k:

IH $\text{Wopperd}(k) = 2k+2$

TS: $\text{Wopperd}(k+1) = 2(k+1)+2$

$= 2+\text{Wopperd}(k)$

$\underset{IH}{=} 2+2k+2 \quad =$

Big-O notation

① ignore constant factors

② ignore small inputs

→ rough comparisons between algorithms

→ rough predictions

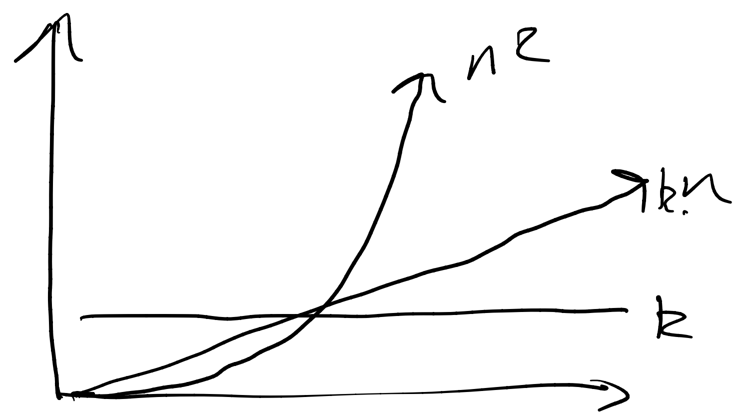
buckets

$O(1) \subset O(\log n) \subseteq O(n) \subseteq O(n \log n) \subset \ldots$

constant linear

$\ldots \subset O(n^2) \subset O(n^3) \subseteq O(2^n)$

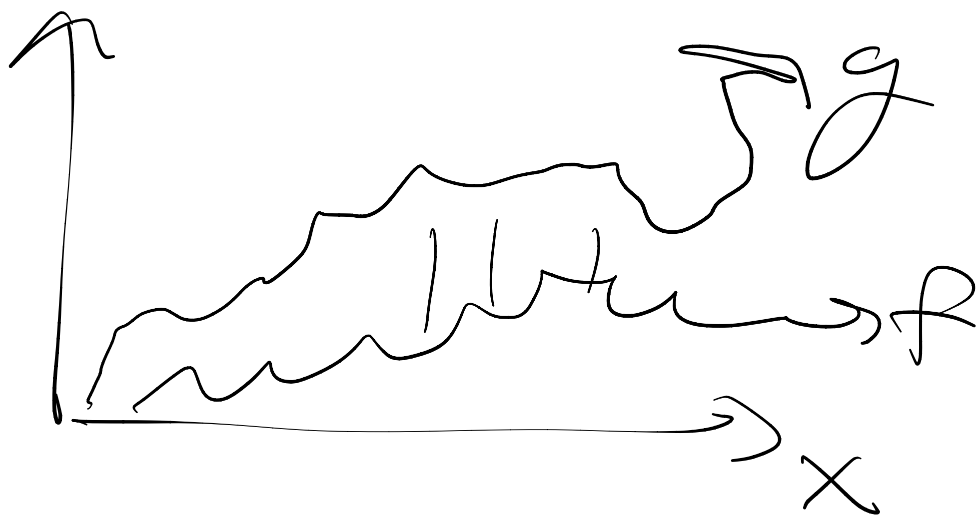
quadratic cubic exponential



1. approx I to $O(-)$

f is $O(g)$ means

for all x, $f(x) \leq g(x)$



Eg.

x is $O(2x)$ but $2x$ is not $O(x)$

2) ignore constant factors

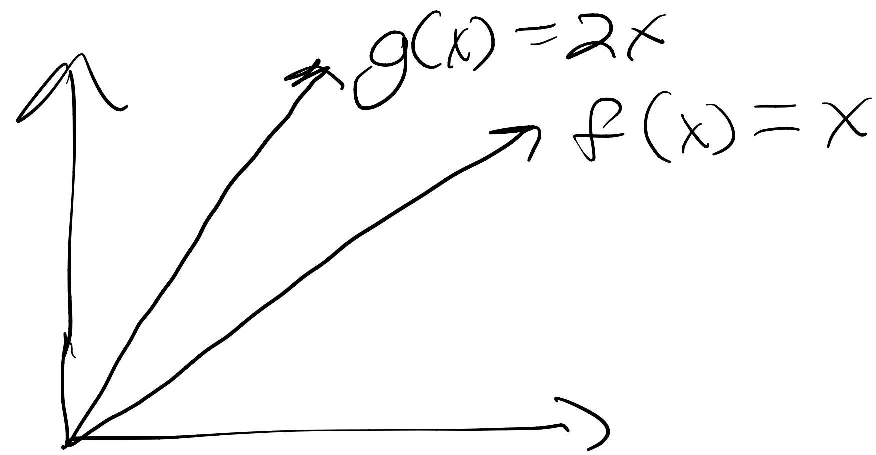
f is $O(g)$ means

there exists a k such that

for all x,

$$f(x) \le k\, g(x)$$

E.g.



x is $O(2x)$

$2x$ is $O(x)$

$2x \le kx$

for $k = 2$

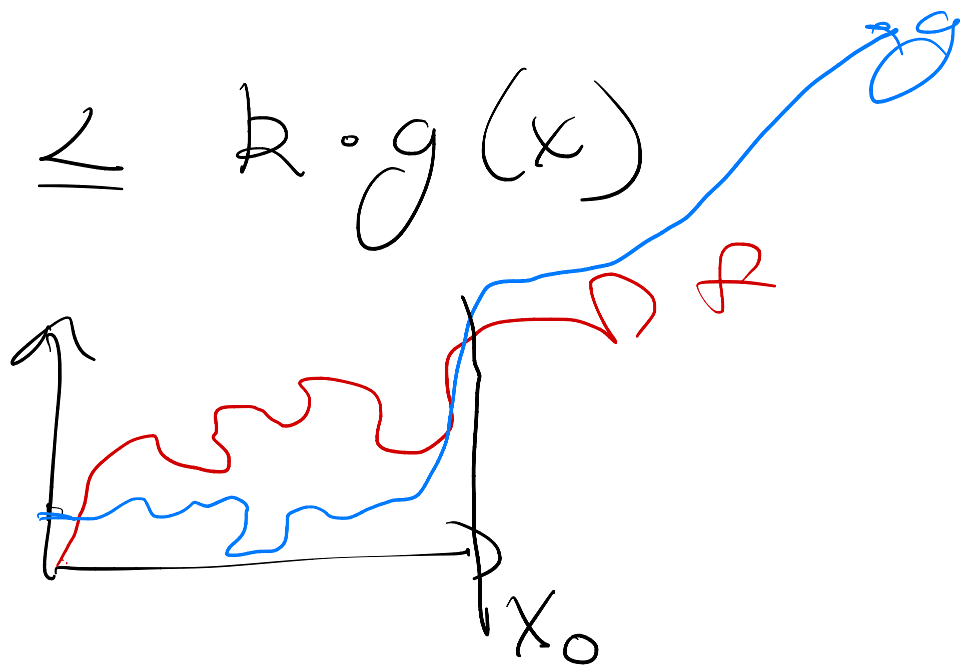
③ ignore small inputs

f is $O(g)$ iff

there exists a k and x_0 such that

for all $x \geq x_0$

$$f(x) \leq k \cdot g(x)$$



$$\text{E.g. } \boxed{W_{append}(x) = 2x + 2} \text{ is } O(x)$$

x — length of l_1

approx 2: $\exists k$ s.t. for all $x \geq 0$

$$2x + 2 \leq kx$$

$$2 \cdot 0 + 2 = 2 \not\leq k \cdot 0 = 0$$

real def $\exists k\ x_0$ s.t. for all $x \geq x_0$

$$2x + 2 \leq kx$$

$$\leq 4x$$

E.g. $\boxed{x_0 = 1,\ k = 4}$

$x = 0$ no

$x = 1$ yes

$x = 2$ yes ...

Wrapped(n) n = length of l_1

Wrapped(n) has closed form

$$2n+2$$

$2n+2$ is $O(n)$

fun reverse(l: int list) : int list =
case l of
[] => []
| x::xs => append(reverse(xs), [x])

put x at the back e

(1) recurrence $W_{rev}(0) = k_0$ ← some constant

$$W_{rev}(\underset{\uparrow}{n}) = k_1 + W_{rev}(n-1) + W_{append}(n-1)$$

length of l

size of input to rec.

size of input to append

length(reverse xs)
= length xs

$$W_{red}(n) = k_1 + W_{red}(n-1) + \underbrace{W_{opp}(n-1)}$$

$$\leq k_1 + W_{red}(n-1) + k_{\frac{n}{2}}$$

$$\approx 1 + W_{red}(n-1) + n$$

$$\approx n + W_{red}(n-1)$$

$$\underbrace{n + (n+1) + (n-2) + (n-3) + \cdots}$$

summation of n

Closed form

$$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \text{ is } O(n^2)$$

(* Purpose: reverse l into r *)

(* E.g. revTwoPiles([1,2,3], [4,5,6])

= [3,2,1,4,5,6] *)

```
fun revTwoPiles(l: int list,
                r: int list) : int list =
  case l of
    [] => r
  | x::xs => revTwoPiles(xs, x::r)
```

```
(* reverse l in linear time *)

fun fastReverse(l: int list): int list =
    revTwoPiles(l, [])
```

$$\underbrace{W_{fastReverse}(n)}_{O(n)} = k + \underbrace{W_{revTwoPiles}(n)}$$

n — length of l

$$W_{revTwoPiles}(0) = k_0 \quad 1$$

$$W_{revTwoPiles}(n) = k_1 + W_{revTwoPiles}(n-1) \quad 1 \quad O(n)$$

n — length of l

$$\boxed{\begin{aligned} &W(0) = \text{constant} \\ &W(n) = 1 + W(n-1) \quad \text{is } O(n) \end{aligned}}$$

Harder problems

can be

easier to solve

efficiently!