FALL 2018 CS DIVERSITY & INCLUSION TOWN HALL

ALL ARE WELCOME
WE WANT TO HEAR YOU!

Thursday November 29, 4P-5P
CIT 3rd Floor Atrium
Diversity & Inclusion Student Advocates 2019-2020
Applications Now Open!

https://tinyurl.com/2019diversityadvocate

(application closes November 28th)
OCaml has built-in operator @ for appending two lists

\[ [17; 18; 22] \, \@ \, [53; 32] \] evaluates to \[ [17; 18; 22; 53; 32] \]

**Quiz:** Write `append: 'a list * 'a list -> 'a list` without using @

**Input:** two lists

**Output:** the list consisting of the items in the first input list followed by those in the second input list

```ocaml
let rec append = function
  | [], lst -> lst
  | first::rest, lst -> first::append (rest, lst)
```
Running time of `append`

Let $f(m,n)$ be worst-case time for appending an $m$-element list and an $n$-element list

**OCaml runtime analysis:** pattern-matching takes constant time (assume no “fancy” stuff)

$f(0,n) \leq b$

$f(m,n) \leq a + f(m-1, n)$

**Solution:** $f(m,n) \leq am + b$

A linear-time procedure.

```ocaml
let rec append = function
  | [], lst -> lst
  | first::rest, lst -> first::append (rest, lst)
```
reverse procedure

Quiz: Write reverse: 'a list -> 'a list. You are allowed to use @ operator. Also: analyze running time.

let rec reverse = function
    [] -> []
  | first::rest -> reverse rest @ [first]

Let \( g(n) \) be reverse’s worst-case running time for lists of length \( n \)

We said appending has running time \( f(m,n) \leq am + b \)
\[ g(0) \leq c \]
\[ g(n) \leq d + a \cdot (n-1) + b + g(n-1) \]

Solution: \( g(n) \) is \( \text{O}(n^2) \)

Is there a faster algorithm?
Usual strategy for recursion

1. Given an input, derive the recursive input from input in a straightforward way  
ed.g. cdr
2. Derive the output from the recursive output in some way that depends on problem.

For example, you might find this strategy useful in solving the following problem

input: matrix, represented as 'a list list
output: list consisting of the diagonals of the matrix, represented as 'a list list

Example: Diagonals of

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{array}
\]

are [1], [2;5], [3;6;9], [4;7;10], [8;11], and [12]

You can use usual strategy (I used three helper procedures, including take and drop)
Introduction to tail recursion

Usual strategy

1. Given an input, derive the recursive input from input in a straightforward way
e.g. cdr
2. Derive the output from the recursive output in some way that depends on problem.

Tail-recursion strategy

Output is same as recursive output
Add a new argument, the partial-solution argument
Base case just returns the value of that argument
Because of extra argument, need a helper procedure
Recursion happens only in helper procedure
Body of original procedure just calls helper with partial-solution argument being trivial value
Why tail recursion?

Consider how a computer would carry out a recursive computation

Consider append
Takes two arguments

\[
\text{append } ([1;2;3], [4;5;6]) \\
\text{append } ([2;3], [4;5;6]) \\
\text{append } ([3], [4;5;6]) \\
\text{append } ([], [4;5;6])
\]

\[
[4;5;6] \\
3 :: [4;5;6] \\
2 :: [3;4;5;6] \\
1 :: [2;3;4;5;6]
\]

Once recursive call returns, OCaml must cons head of first argument onto result of recursive call

Therefore OCaml must store first argument binding while waiting for recursive call to return.

Uses a stack data structure

The stack stores little local environments, called stack frames

Number of stack frames stored = recursion depth

Managing the stack is a bit expensive

Some languages limit stack size.

Tail recursion does not require stack to grow.
Tail-recursive append

Quiz: Write `append` tail-recursively

Must still be linear time

**Hint:** at the end, doesn’t return just the value of the partial-result—has to do some additional processing.

**Hint:** Uses `reverse`