Programming Languages and Techniques (CIS120)

Lecture 4

Jan 20, 2012

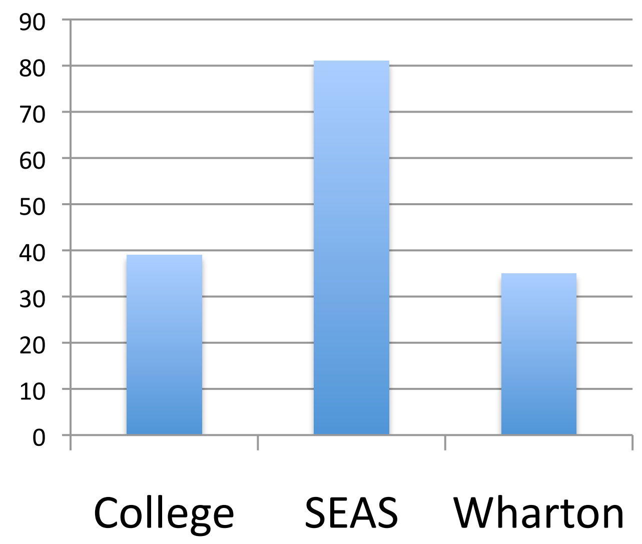
Lists II, Tuples, and Patterns

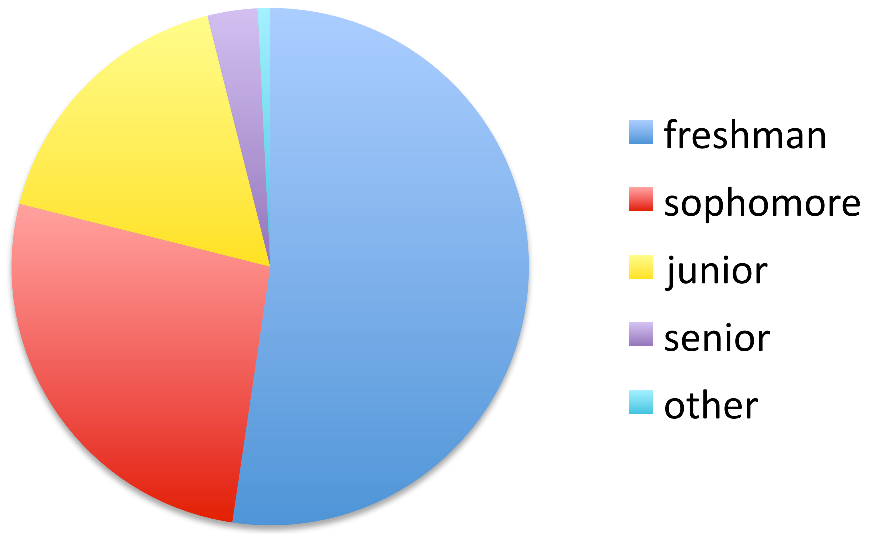
Announcements

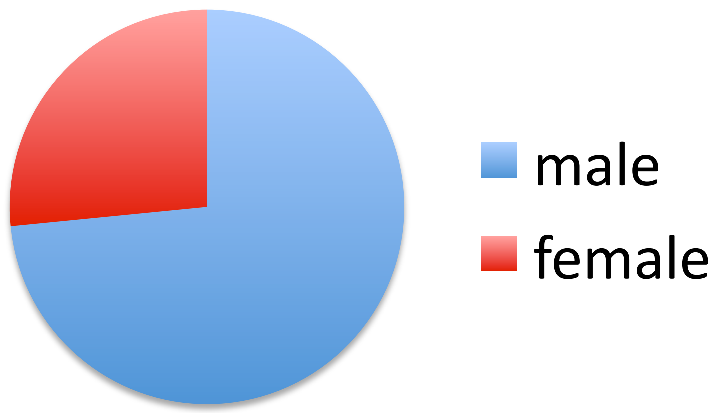
- Homework 1: OCaml Finger Exercises
  - Due: Monday, Jan 23rd at 11:59:59pm (midnight)
  - Don't use '@' for Problem 7. ☺

- TA office hours: on website and Piazza
- Questions?
  - Post to Piazza (privately if need to include code)
  - Office hours (Weirich: 4-5PM today, none on Monday)

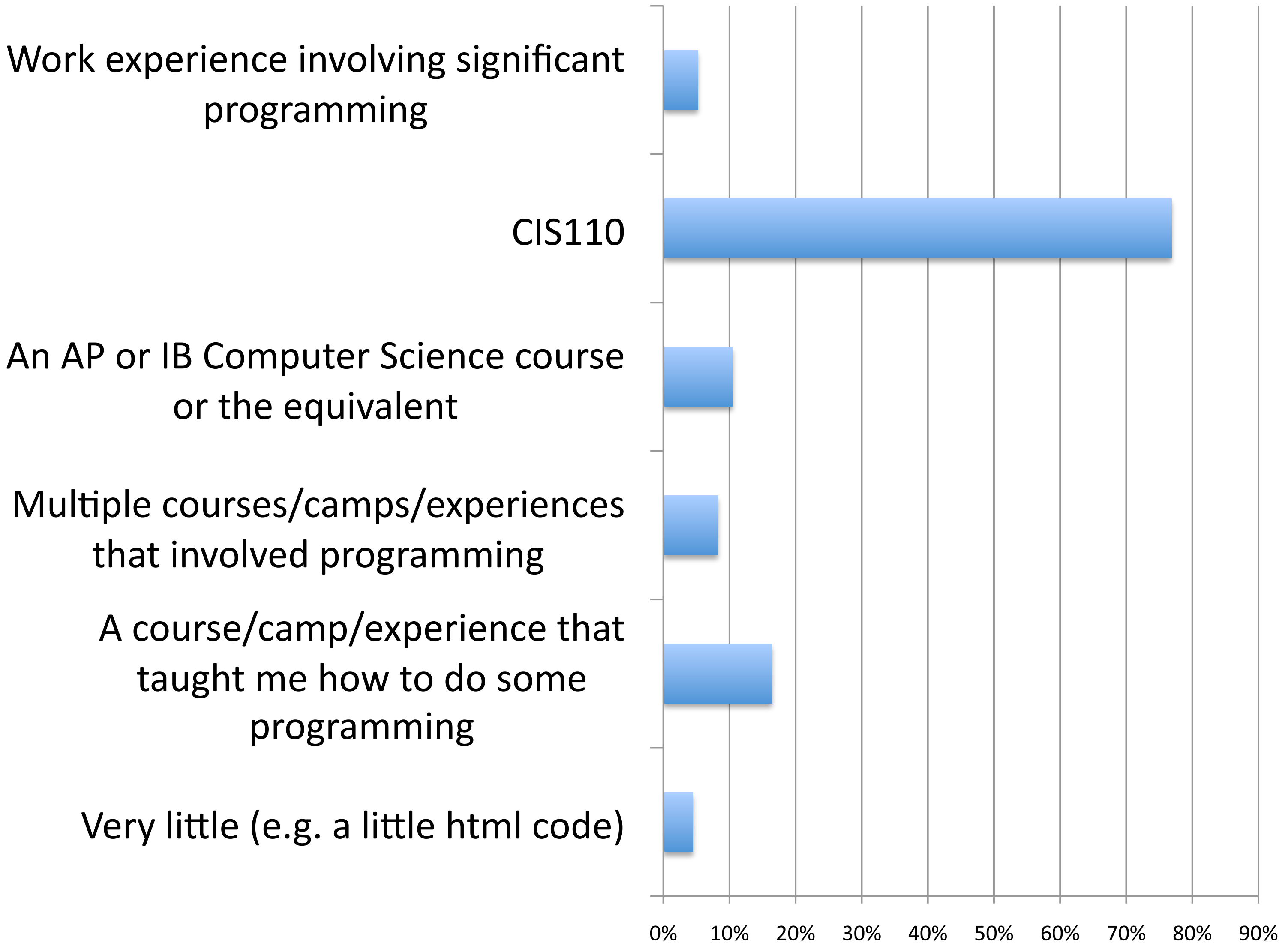
Who is CIS 120?



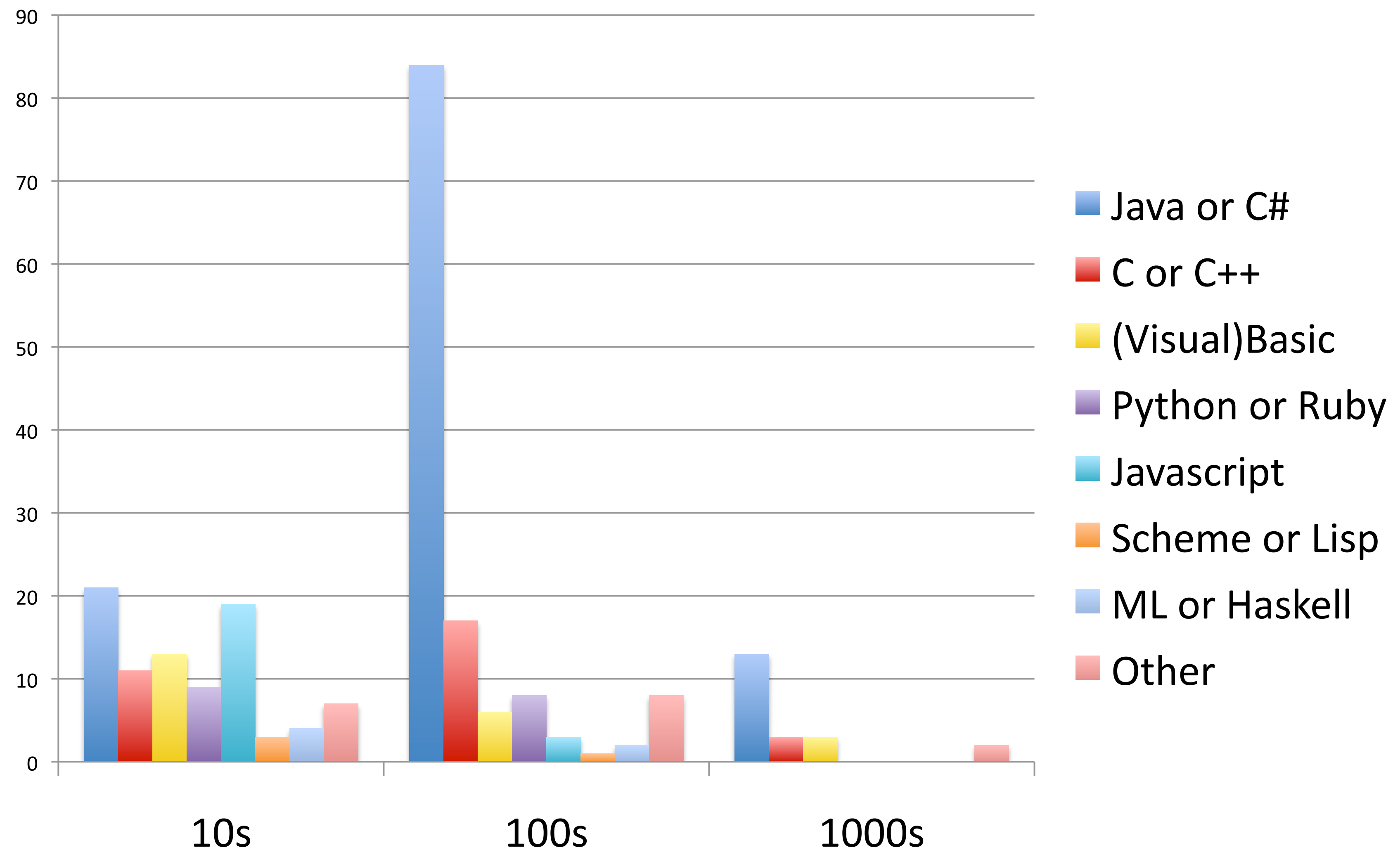




Prior experience with programming



Size of largest program



Lists

List Types*

The type of lists of integers is written

```
int list
```

The type of lists of strings is written

```
string list
```

The type of lists of booleans is written

```
bool list
```

The type of lists of lists of strings is written

```
(string list) list
```

etc.

*Note that lists in OCaml are *homogeneous* – all of the list elements must be of the same type. If you try to create a list like [1; “hello”; 3;true] you will get a type error.

What is a list?

- A list is either:

`[ ]`	the *empty* list, sometimes called *nil*, or
`v::tail`	a *head* value v, followed by a list of the remaining elements, the *tail*

- Here, the '`::`' operator *constructs* a new list from a head element and a shorter list.
  - This operator is pronounced "cons" (for "construct")

- Importantly, there is no other way to create a list.

Example Lists

To build a list, cons together elements, ending with the empty list:

`1::2::3::4::[]`	a list of four numbers
`“abc”::”xyz”::[]`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Explicitly parenthesized

‘`::`’ is an ordinary operator like + or ^, except it takes an element and a *list* of elements as inputs:

`1::(2::(3::(4::[])))`	a list of four numbers
`“abc”::(”xyz”::[])`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Convenient List Syntax

Much simpler notation: enclose a list of elements in `[` and `]` separated by `;`

`[1;2;3;4]`	a list of four numbers
`[“abc”;”xyz”]`	a list of two strings
`[true]`	a list of one boolean
`[]`	the empty list

Using Recursion Over Lists

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty. *Pattern matching* determines which.

```
let rec number_of_songs (pl : string list) : int =
  begin match pl with
  | [] -> 0
  | ( song :: rest ) -> 1 + number_of_songs rest
  end
```

If the lists is non-empty, then “song” is the first song of the list and “rest” is the remainder of the list.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

Calculating With Lists

- Calculating with lists is just as easy as calculating with arithmetic expressions:

$$
\begin{array}{lll}
 & (2+3)::(12\ /\ 5)::[] & \\
\longmapsto & 5::(12\ /\ 5)::[] & \text{because } 2+3 \Rightarrow 5 \\
\longmapsto & 5::2::[] & \text{because } 12/5 \Rightarrow 2
\end{array}
$$

A list is a value whenever all of its elements are values.

Calculating with Cases

- Consider how to run a match expression:

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

$\longmapsto$

1 + 10

$\longmapsto$

11

Note: `[1;2;3]` equals `1::(2::(3::[]))`

It doesn't match the pattern [] so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is 1 and `rest` is `(2::(3::[]))`.
So, substitute 1 for `first` in the second branch

Calculating with Recursion

```
    number_of_songs [“Monster”;”Teeth”]
⟼      (substitute the list for pl in the function body)
    begin match “Monster”::(“Teeth”::[]) with
      | [] -> 0
      | (song :: rest) -> 1 + (number_of_songs rest)
    end
⟼      (second case matches with rest = “Teeth”::[]
    1 + (number_of_songs “Teeth”::[])
⟼      (substitute the list for pl in the function body)
    1 + (begin match “Teeth”::[] with
          | [] -> 0
          | (song :: rest) -> 1 + (number_of_songs rest)
         end
⟼      (second case matches again, with rest = [])
    1 + (1 + number_of_songs [])
⟼      (substitute [] for pl in the function body)
    …
```

```
let rec number_of_songs (pl : string list) : int =
  begin match pl with
  | [] -> 0
  | ( song :: rest ) -> 1 + number_of_songs rest
  end
```

Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec number_of_songs (pl : string list) : int =
   begin match pl with
   | [] -> 0
   | ( song :: rest ) -> 1 + number_of_songs rest
   end
```

```
let rec contains (pl:string list) (s:string) : bool =
   begin match pl with
   | [] -> false
   | ( song :: rest ) -> s = song || contains rest s
   end
```

Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[ ]` calculates the value (`f  [ ]`) directly.

The branch for `hd::rest` calculates

(`f(hd::rest)`) given hd and (`f  rest`).

Design Pattern for Recursion

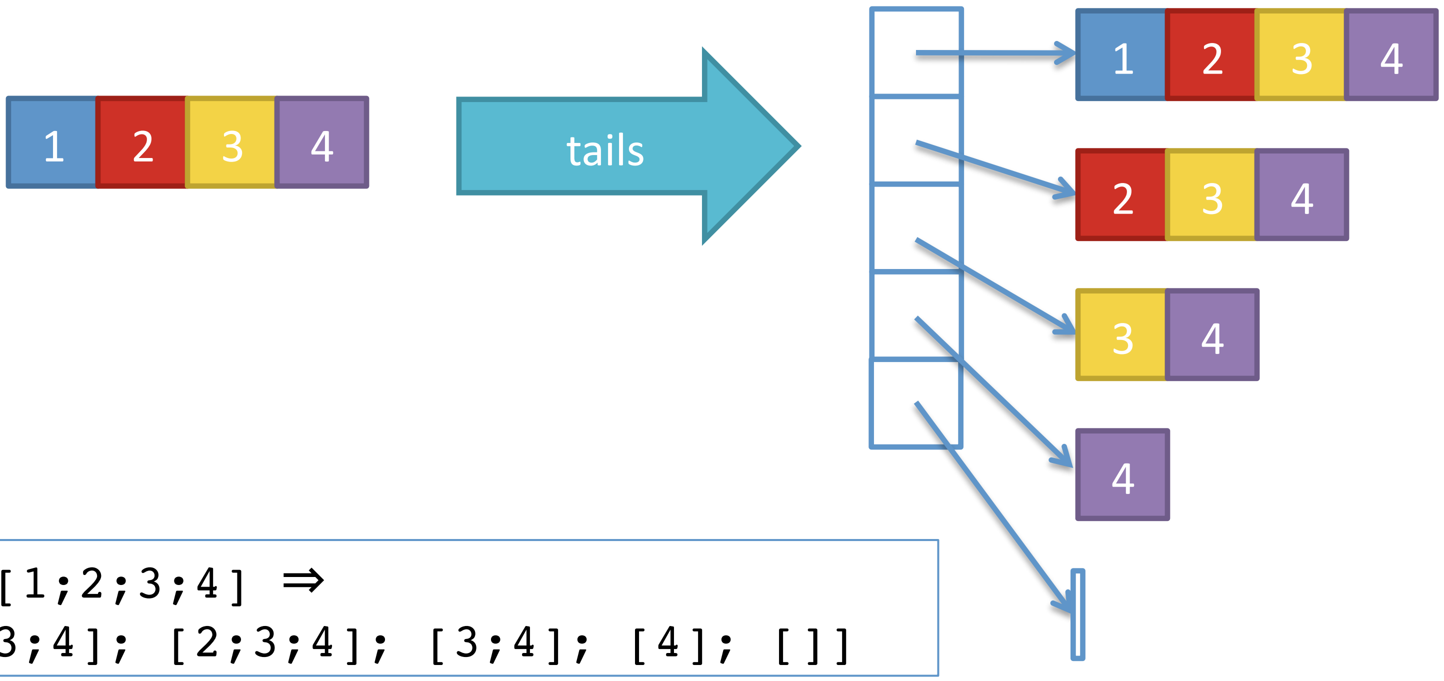
1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
   - If the main input to the program is an immutable list, look for a recursive solution…
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?
     - Is there a direct solution for the empty list?

More List Examples

see lists.ml

tails

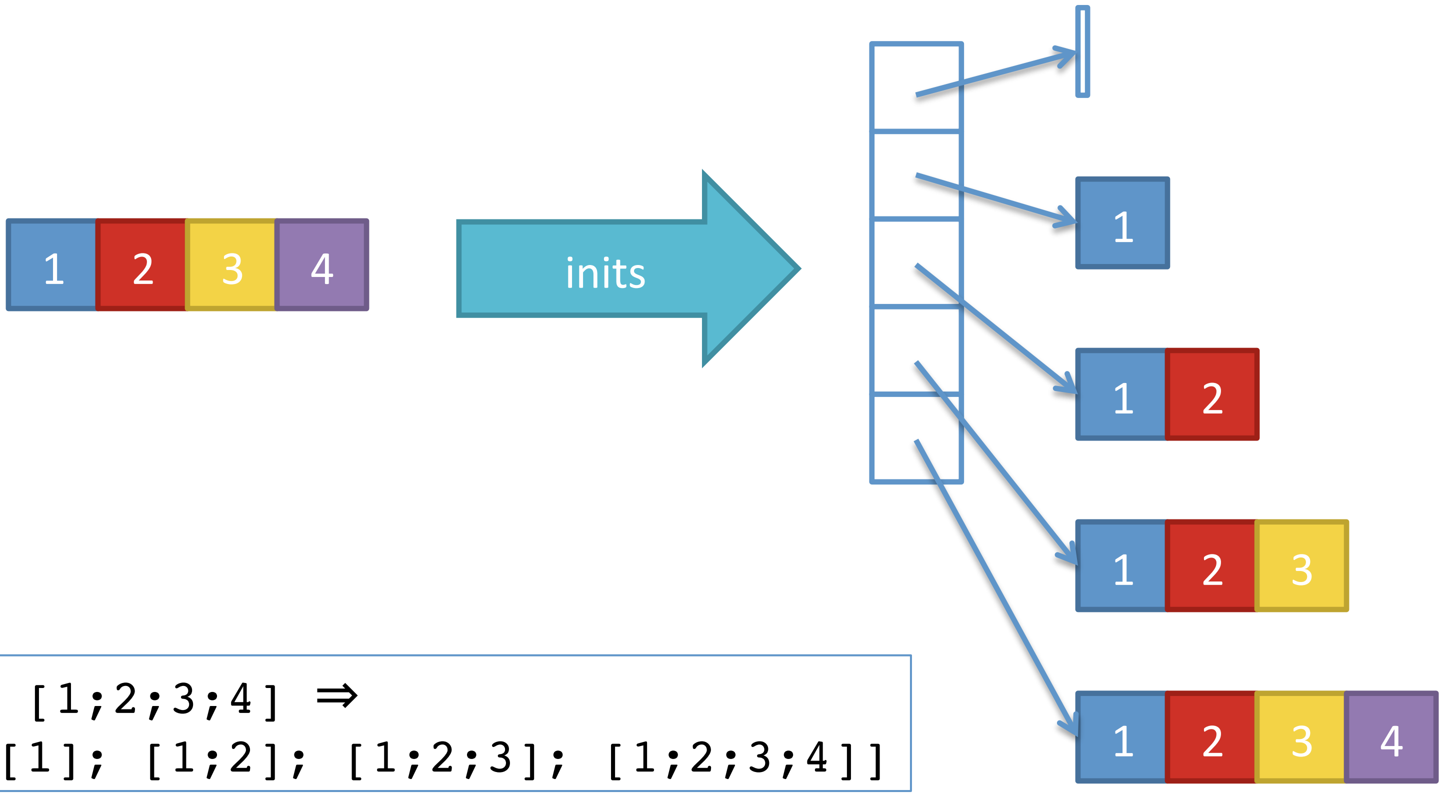
- Design problem: Given a list of integers, produce all suffixes of a given list, starting with the full list and removing the first element at each step



```
tails [1;2;3;4] ⇒
[[1;2;3;4]; [2;3;4]; [3;4]; [4]; []]
```

inits

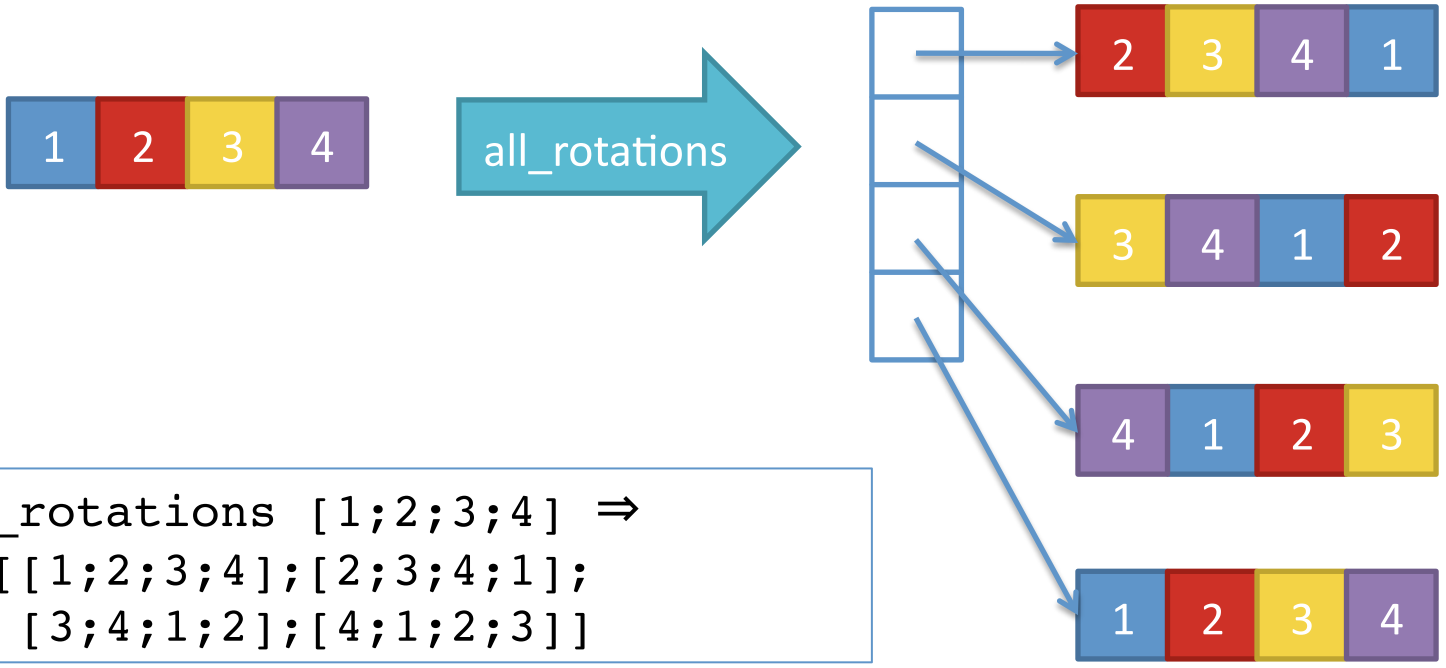
- Design problem: Given a list, produce all *initial prefixes* of the list.



```
inits [1;2;3;4] ⇒
[[]; [1]; [1;2]; [1;2;3]; [1;2;3;4]]
```

Challenge: All rotations

- Design problem: Given a list, produce all *rotations* of the list.



```
all_rotations [1;2;3;4] ⇒
   [[1;2;3;4];[2;3;4;1];
    [3;4;1;2];[4;1;2;3]]
```