Consolidation & Homeworks

Yan Huang

Goal for Today

- Consolidate your learning from the past few lectures
- HW2

List Comprehension and **`zip`**

Using "zip" we can define a function that returns the list of all <u>positions</u> of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

List Comprehension and **`zip`**

positions :: Eq a ⇒ a → [a] → [Int]

Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x’,i) ← zip xs [0..], x == x’]
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means [’a’, ’b’, ’c’] :: [Char].

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

```
count     :: Char → String → Int
```

count c s = length [i | i ∈ s, i == c]

For example:

```
> count 's' "Mississippi"
4
```

length "ssss" = 4

Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

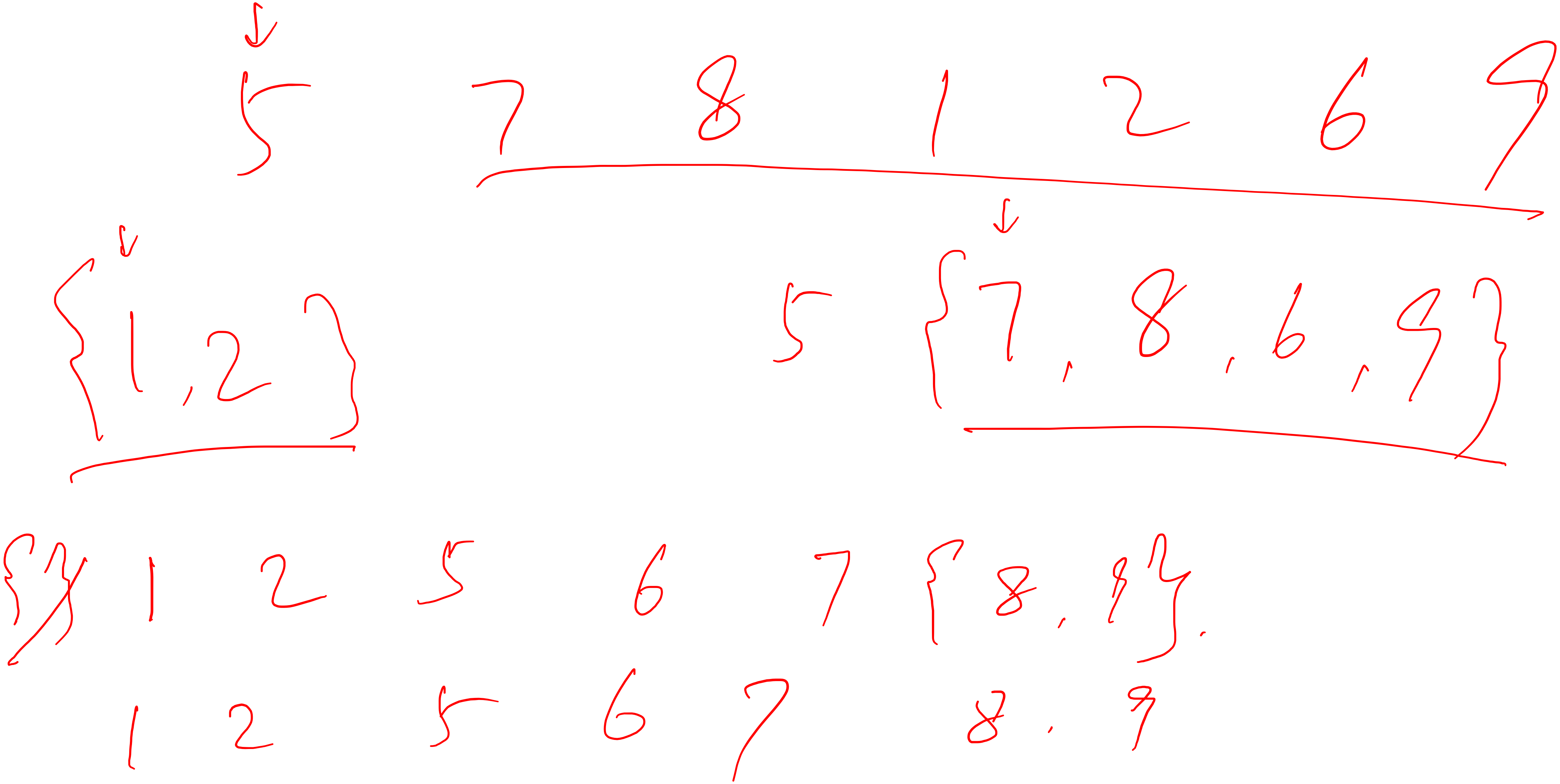
```
count      :: Char → String → Int
count x xs =
   length [x’ | x’ ← xs, x == x’]
```

For example:

```
> count ’s’ "Mississippi"
4
```

Recursive Functions and Quick Sort

Quick Sort



Quick Sort

The quicksort algorithm for sorting a list of values can be specified by the following two rules:

- The empty list is already sorted;

- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

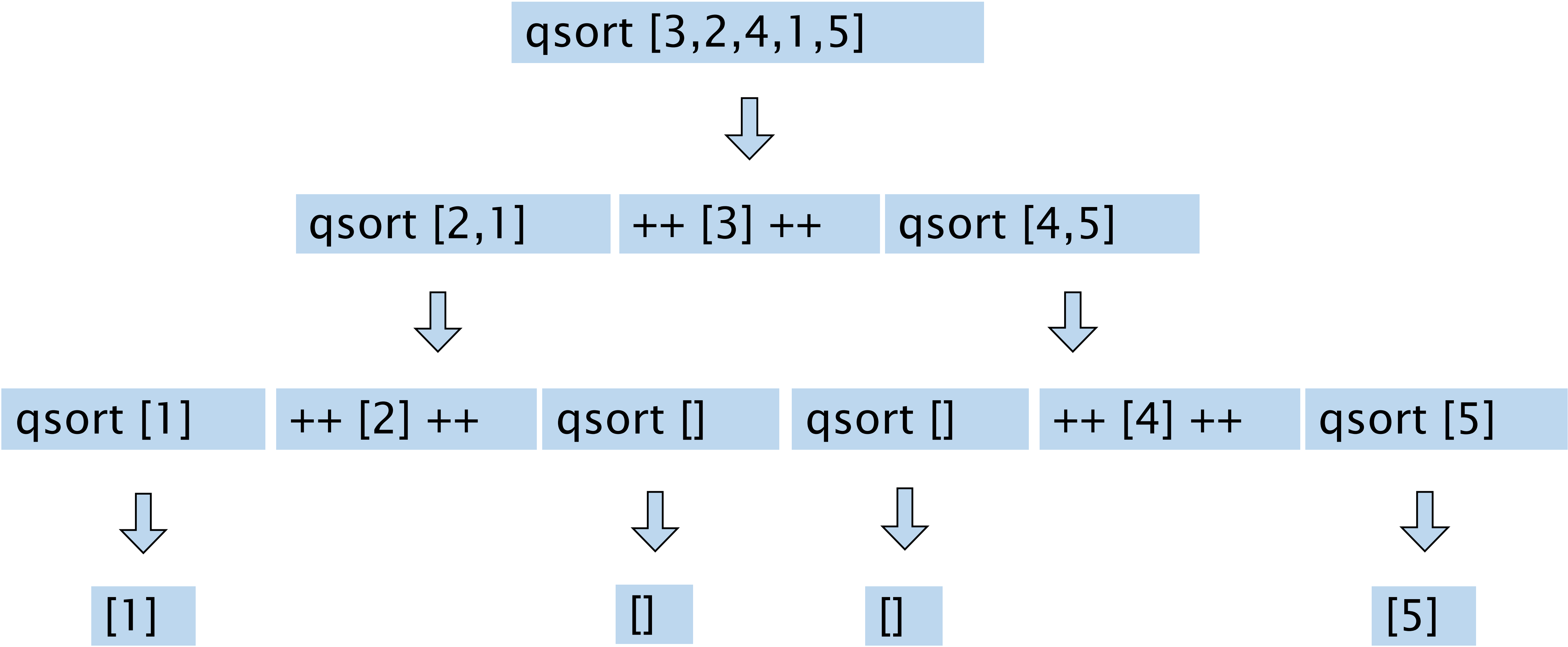
Using recursion, this specification can be translated directly into an implementation:

Using recursion, this specification can be translated directly into an implementation:

```
qsort        :: Ord a ⇒ [a] → [a]
qsort []      = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

- This is probably the <u>simplest</u> implementation of quicksort in any programming language!

For example (abbreviating qsort as q):



Homework 2

P1

('a', 'b', 'c')　　(Char, Char, Char)

:type ('a', 'b', 'c')

P2　swap :: (a, b) -> (b, a).
swap (x, y) = (y, x)

f :: (Integral a) => a -> Rational
f x = toRational $ sum xs

dotprod [1, 2, 3] [5, 6, 8]

= 1 × 5 + 2 × 6 + 3 × 8

dotproduct xs ys = sum [(x * y) | (x, y) ∈ zip xs ys]