

1 Recursion

- 1.1 (Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is `[3, 4, 8, 9]`

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    (x) incrementing or doubling. (y)
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    """
    if x == y:
        return [[x]]
    elif x > y:
        return []
    else:
        a = paths(x+1, y)
        b = paths(2x, y)
        return [x] + subpath for subpath in a+b
```

⊛ count-stairways
count_K

3 → 5
[3 4 5] 4:22

x → y 5 → 3

① Base case
② Recursive case

3 → 5
4 → 5
[cy] [3 ~]

2 base cases

Recursive case

→ incrementing

→ doubling

x
[x+1, y]
[2x, y]

⊛ x → x+1
→ 2x

[x+1, y], [2x, y]

[x] + [3 2]

5 Mutable Linked Lists and Trees

- 5.1 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

```
if lnk is Link.empty or lnk.rest is Link.empty:
    return
else:
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

```
while lnk is not Link.empty and lnk.rest is not Link.empty:
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
    else:
        lnk = lnk.rest
```

Recursive

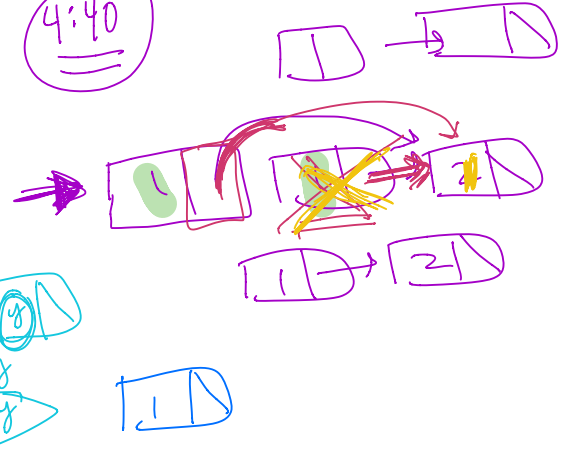
thor

1st t =

Tree(1, —)

Link(value, —)
Link attribute

4:40



7 Scheme

→ Tail Recursion

- 7.1 Write a function that takes a procedure and applies to every element in a given nested list.

The result should be a nested list with the same structure as the input list, but with each element replaced by the result of applying the procedure to that element.

Use the built-in `list?` procedure to detect whether a value is a list.

(define (deep-map fn lst)

(cond
 ((null? lst) lst)
 ((list? (car lst)) (cons (deep-map fn (car lst)) (deep-map fn (cdr lst))))
 (else (cons (fn (car lst)) (deep-map fn (cdr lst))))
)

scm> (deep-map (lambda (x) (* x x)) '(1 2 3))

(1 4 9)

scm> (deep-map (lambda (x) (* x x)) '(1 ((4) 5) 9))

(1 ((16) 25) 81)

