Lecture 2

1. Strings

Problem 3: HW 2

String Append

(string-append "abc" "def")

⇒ "abcdef"

(string-append "ab" "bc" "de")

⇒ "abbcde"

Problem 6: HW 2

You need to compare two boolean values

(boolean=? #t #f) ⇒ false

(

Data Type: Symbols

set of Keyboard characters

Suppose you're desing checkers app.
Red, Black
We can store this information
using Symbols.

Input: 'Red ⇒ symbol Red
'Red ≠ "Red"

For Problem 2:
Grades will be in symbols

Write which outputs 'R, 'B, 'G

(n mod 3) == 1 → 'Red

(n mod 3) == 2 → 'Blue

(n mod 3) == 0 → 'Green

```
(: num-to-color : Integer → Symbol)

(define (num-to-color n)
 (if (= (remainder n 3) 0)
       'Green
   (if (= (remainder n 3) 1)
         'Red
         'Blue ) ) )
```

Recursion

Recursion → Divide and Conquer
Recursion → Mathematical Induction

Main Idea: You have a problem P. Divide it into sub problems P_1 and P_2. Suppose we can solve P_1 and P_2.
Combine the solution of P_1 and P_2 to solve P.

Example:
Write a program to sum first n natural numbers.
Input: n.
Output: 1+2+3+4 .... n.

$sum(3) = 6$

$sum(4) = 10$.

$sum(3) = 1+2+3$

$= sum(2) + 3$.

In general.

$sum(n) = sum(n-1) + n$.

(Recurrence Relation).

$sum(4) = 4 + sum(3)$

$= 4 + 3 + sum(2)$

$= 4 + 3 + 1 + sum(1)$

$= 4 + 3 + 1 + 0 + sum(-1)$

$= 4 + 3 + 1 + 0 + (-1) + sum(-2)$

sum(0) = 0 ← Base case

sum(4) = 4 + sum(3)
= 4 + 3 + sum(2)
= 4 + 3 + 2 + sum(1)
= 4 + 3 + 2 + 1 + sum(0)
= 4 + 3 + 2 + 1 + 0 ⇒ 10

→ Recurrence Relation

sum(n) = sum(n-1) + n.
sum(0) = 0 — Base Case

```
(: sum : Integer -> Integer)
(define (sum n)
  (if (= n 0) 0
      (+ (sum (- n 1)) n)))
```

(check-expect (sum 3) 6)

(sum 3) $\Rightarrow$ 3 + (sum 2)

$\Rightarrow$ 3 + 2 + (sum 2)

$\Rightarrow$ 3 + 2 + 1 (sum 0)

$\Rightarrow$ 3 + 2 + 1 + 0

= 6.

(sum -1) $\Rightarrow$ Infinite Loop

Example 2:

Write a function to count 'c', 's' letters in a string.

(count-cs "abc") ⇒ 1

(count-cs "cscs") ⇒ 4.

S = "abc␣"

Extract the last character and check if it is 'c' or 's'.

+1

Recurrence Relation

count-cs(S) = count-cs(S[1, n])
+ if(S[1] = 'c' or S[1] = 's')
else
0.

len(S) > 1

if n == 0 (string is empty)

count-cs(S) = 0.

$s[0, n-1]$ → The string you get by removing the last character.

$s[n]$ → the last character.

n → length of the string.

Racket function

```
(: count-cs : String -> Integer)
(define (count-cs s)
  (if (= (string-length s) 0)
      0
      (+ (count-cs (substring s 1))
```

removes the first charack.

```
(if (or
    (char=? #\c (string-ref s 0))
    (char=? #\s (string-ref s 0))
    1
    0))))
```

→ The first character is 'c' or 's'

string-length

string-ref

(string-ref "abc" 0) ⇒ "a"

0 1 2

(string-ref "abc" 1) ⇒ "b"

(substring "abc" 1) ⇒ "bc"

(substring "abcd" 2) ⇒ "cd"

Example 3

factorial (n) = 1 · 2 · 3 · 4 · 5 · · · n .

Recurrence for factorial

factorial (n) = factorial (n-1) · n

factorial (0) = 1 ← Base case.

Demonstrate error functions

(: fact : Integer → Integer)

```
(define (fact n)
(if (> 0 n)
    (error "Input less than zero")
```

```
(if (= x 0)
    1                    ← Base case
    (* x (fact (- x 1)))).
                  ↖
              Recurrence.
```

```
(check-error (fact -1) "Input less
                        than zero").
```

Cond

Avoid nested if statements

Example: Using Cond

Write function which takes a county as input and prints its population.

```
(: population : String → Integer)
(define (population country)

(cond
 [(string=? country "USA") 3,00,00,000]
 [(string=? country "Mexico") 1,200,000]
 [(string=? country "Canada") 200,000]
 [else (error "I don't know
              the answer")])
```

country ← Input

(string=? country 'USA) ⇒ #t

(population 'USA) ⇒ 3,000,000,000.

General Template for Recursive functions.

```
(: func : Integer -> Integer)
(define (func n)
 (cond
  [(base-case 1) result1]
  [(base-case 2) result2].
  [(base-case 3) result3]
  [else recursive-case]))
```

List

[97, 100, -15, 6]

(: my-list : (Listof Integer))

(define my-list (list 97 100 -15 6))

(Listof (Listof (Listof Integer)))
(Listof Boolean)

Write important functions for list

- first

(first my-list) ⇒ 97

- rest

(rest my-list) ⇒ '(100 -15 6)

- empty?

(empty? my-list) ⇒ #f

Example:

Write a program to add

every element of List of Integers.

(10 -11 13 14)

$\Rightarrow$ 26

First Recurrence

(sum-list nums)

= (first nums) + (sum-list (rest nums))

→ Recurrence Relation.

Sum-list ((100 -11 23 4))

= 100 + sum-list ((-11 23 4))

What is the base-case?
When the list is empty?
If list is empty, sum is 0.

(: sum-list : (Listof Integer) → Integer)

```
(define (sum-list input)
 (cond
  [(empty? input) 0]
  [else (+ (first input) (sum-list
    (rest input)))]))
```

Example

You have a list of strings which are actually books you own, you have to count how many copies of "Oedipus Rex" do you have.

count-occurence (input)

= (if (string=? (first input) "Oedipus Rex"
1
0)

```
+ count-occurence ((rest input))
```

If my input is empty, output is 0.

```
(: count-occurences : (Listof String)
   String -> Integer)

(define (count-occurences my-books
                          title)

(cond
 [(empty? my-books) 0]

 [else (+
```

```
(count-occurences (rest my-books))

(if (string=? (first my-books)
              title)
    1
    0)))
```

```
(check-expect (count-occurences
 (list "ABC" "DEF" "XYZ")
  "XYZ") 1)
```

Example

Selling Cocunuts

You're selling cocunuts. Eact time you sell a cocunut, you the raise th price by $3. Inihal, price of th cocunut is $1. If there n people, how much will you make.

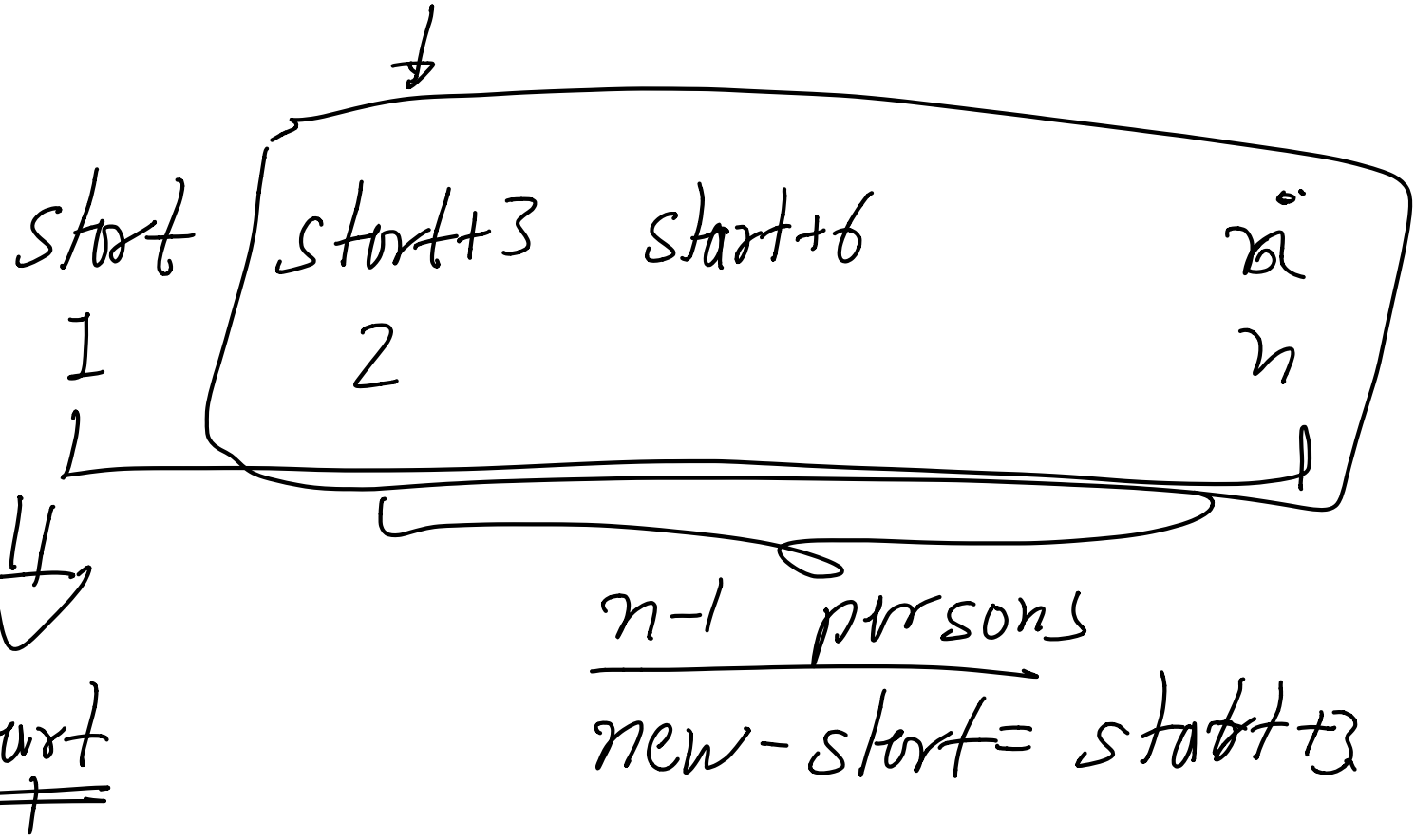
Summation of anthmehc progression.

Let's assume we have have an arbibary inhial price. start

profit(n, start)

subprobl

$= profit(n-1, start+3) + start$



If number of people is 0, then you make zero profit.

if $n = 0$, $profit(n, start) = 0$.

```
(: profit : Integer Integer
   → Integer)
(define (profit n start)

(cond
  [(= n 0)   0]
  [else (+ start (profit (- n 1)
                         (+ start 3)))])

(: my-profit : Integer → Integer)

((define (my-profit n)

   (profit n 1))).
```

Fibonacci

$F_0 = 0, \quad F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

0, 1, 1, 2, 3, 5, 8, 13

Write a racket function to compute n^{th} fibonacci.

Here we have two base cases.

C: fibo : Integer → Integer

```
(define (fibo n)

  (cond
    [(= n 0) 0]
    [(= n 1) 1]

    [else (+ (fibo (- n 1))
             (fibo (- n 2)))]))
```