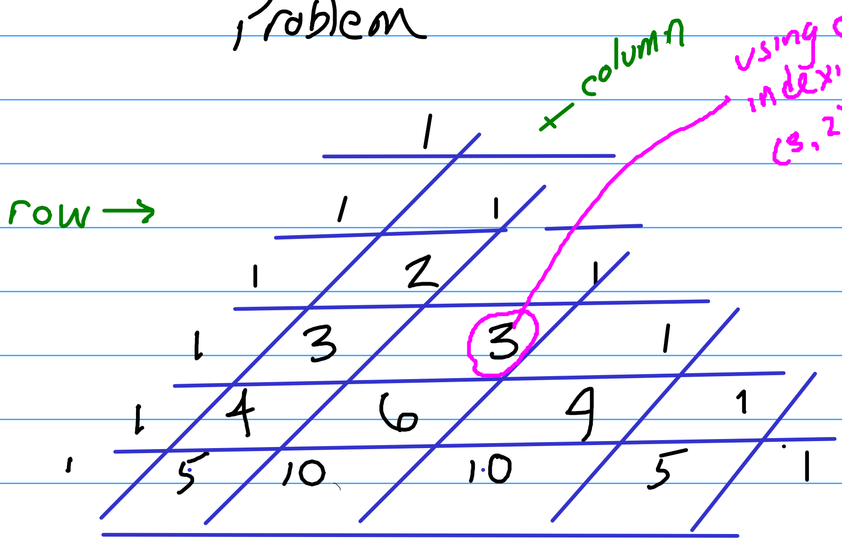
Classes 10 and 11 CSc 335 February 29 and March 5 2024

① Exercise 1.12 from A & S — The Pascal's Triangle Problem



using 0-based indexing, This is the (3,2) entry. Given the def of Pascal's triangle, its natural to want the value $\binom{3}{2}$ in this position.

We are to write a program which inputs the "index" of an entry in this table and which returns the corresponding value

What is the definition of index in this case? After some thought, we decide to use row-column indexing, where columns are diagonal. Given this, there is still the question of whether to use 0-based or 1-based indices.

For example — should the leftmost column be the 0th column or the 1st column?

Since we can in all likelihood develop a program for either choice, we ask whether there is another criterion we can use to make the decision.

As you probably know, the numbers in the Triangle are precisely the binomial coefficients — ie — the numbers $\binom{n}{k}$ which occur as the coefficients in the exponsion of $(x+y)^n$. So it makes sense that the convention we use lines up with this.

Let's compute some values for $\binom{n}{k}$.

$\binom{0}{0} = 1$

$\binom{1}{0} = 1$

$\binom{1}{1} = 1$

$\binom{2}{0} = 1$

$\binom{2}{1} = 2$

$\binom{2}{2} = 1$

Recall $\binom{n}{k} = \dfrac{n!}{k!\,(n-k)!}$

$\longrightarrow$

$$
\begin{array}{ccccccc}
 & & & \binom{0}{0} & & & \\
 & & \binom{1}{0} & & \binom{1}{1} & & \\
 & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\
\binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3}
\end{array}
$$

Suggested: check out 1-based indexing and work out the program

looks like we are best served by using 0-based indexing for both the rows and the columns.

With this in mind, we can tighten the program's specification

;; pre: row and col are integers such that

;; $\binom{row}{col}$ makes sense as a binomial coefficient

;; ie - both are non-negative integers <ins>and</ins>

;; col $\leq$ row

program

;; post: returns $\binom{row}{col}$

Start by understanding the recursion

→ if col = 0, return 1

→ if col = row, return 1

does this catch row = 0 forces output of 1 ?

→ otherwise

just the sum

(pas row col) is computed from

(pas (- row 1) (- col 1))

and

(pas (- row 1) col)

$\binom{row-1}{col-1}$ $\binom{row-1}{col}$

... $\binom{row}{col}$ ...

The recursion is on the row index - only

col not reduced, so can't induct on col

Tentative IH: The recursive calls work so long as the pre-condition is satisfied AND the col index is no more than row-1.

we know col $\leq$ row and we know neither row nor col is 0 and we know col < row. So clearly

col-1 $\leq$ row-1 and also

col $\leq$ row-1 (because everything is an integer)

Termination is clear because for each recursive call, the row index is reduced by 1 — and the program halts when row = 0.

---

is our design ready to be tested as code?

→ design roles for variables

→ divide and conquer strategy — ie — The shape of the IH and IS — is in place

→ termination argument is in place

Looks like we're ready to code!

```
(define (pas row col)
  (cond ((= row 0)   1)
        ((or (= col 0) (= col row))  1)
        (else (+ (pas (- row 1) (- col 1))
                 (pas (- row 1) col)))))
```

Next: check for consistency with the development

eg - does this work as well?

```
(define (pas row col)
  (cond ((= col 0)   1)
        ((= col row)  1)
        (else ... as above) ...)
```

logically equivalent?

if so - the 2nd version would be a bit more efficient

Next: run a few tests

Next: (if time) write out a concise proof, which effectively summarizes your development.

That's a wrap!

In the typed notes, you'll find a version which makes a different choice for the indexing

---

What about an iterative solution?

→ could use iterative factorial in the $\binom{n}{k}$ formula given above, but considerable care needs to be taken to avoid unnecessary duplication of effort.

→ another way would be to make use of lists or vectors, with the idea of computing the r^{th} row (represented as a list, say) from the $(r-1)^{st}$ row. Will come back.

This brings into question what is meant by the phrase "iterative programs work in constant space" because — clearly — these rows grow longer and longer as the computation proceeds.

Similarly, the space needed for

(fact (- n 1))

increases as n increases.

What we mean is: "the stack of calls does not grow"

2

```
; Exercise 1.11 - as given in the text

; a function f is defined by the rule that f(n) = n if n < 3 and
; f(n) = f(n-1) + 2*f(n-2) + 3*f(n-3) if n >= 3.  Write a procedure that
; computes f by means of a recursive process.  Write a procedure that
; computes f by means of an iterative process.

; here is a procedure that computes f by means of a recursive process

(define (f n)
  (cond ((< n 3) n)
      (else (+ (f (- n 1)) (* 2 (f (- n 2))) (* 3 (f (- n 3)))))))
```

basis step

direct transliteration of the given function. pf?

IS is the claim that the values returned by the rec. calls are correctly combined, according to the def

all assumed to compute correctly - because the arguments are all < n

[induct on n]

```
; some thought is required to compute f by means of an iterative process

; design roles
```

Somewhat similar to the Fib. problem.

```
; w = f(m)
; x = f(m-1)
; y = f(m-2)
; z = f(m-3)
```

our goal is to prove that this scheme function correctly implements *

```
; idea for iterative computation

; w <- x + 2y + 3z
; x <- w
; y <- x
; z <- y
```

After m ↦ m+1, what is now f(m) becomes f(m-1).

[ie - we have changed the indexing]

```
; where the updates, just as parameter updates in scheme,  do not interfere with one ar

(define (f-iterative n)

  (define (f-iter m w x y z)
    (cond ((= m n) w)
        (else (f-iter (+ m 1) (+ w (* 2 x) (* 3 y)) w x y))))

  (cond ((< n 3) n)
      (else (f-iter 3 (+ 2 (* 2 1) (* 3 0)) 2 1 0))))
```

The conjunction of these design roles = the invariant

stopping case: m=n

we are supposed to compute f(n)

Design idea: increment m until it reaches n, all the while keeping track

of the information
we need to compute
$f(n+1)$.