Class 9 CSc 335 February 27 and 29 2024

Tree Recursion

The first example is one that CS-1 text book writers love to use to discredit the whole idea of recursion.

We compute the n^{th} fibonacci number, where the fibonacci numbers are

0 1 1 2 3 5 8 13 21 ...

usually with 0-based indexing

0 1 2 3 4 5 6 7 8

Our program is to input an integer $n \geq 0$ and return the n^{th} fibonacci #

2 case basis

```
(define (fib n)
   (cond ((zero? n) 0)
         ((one? n) 1)
         (else (+ (fib (- n 1)) (fib (- n 2))))
   )))
```

Two recursive calls

This is just the scheme version of the definition (so not focussing here on the development) but it nonetheless brings up some interesting points.

First; how do we deal with a 2-case basis step? [Basis step: any computation done without a recursive call]

Second question: how do we deal with two recursive calls?

Let's do the second: as before, we may assume that the recursive calls work correctly PROVIDED the precondition is satisfied when the calls are made

→ need $n-1 \geq 0$ is an integer when (fib (- n 1)) is called

→ need $n-2 \geq 0$ is an integer when (fib (- n 2)) is called

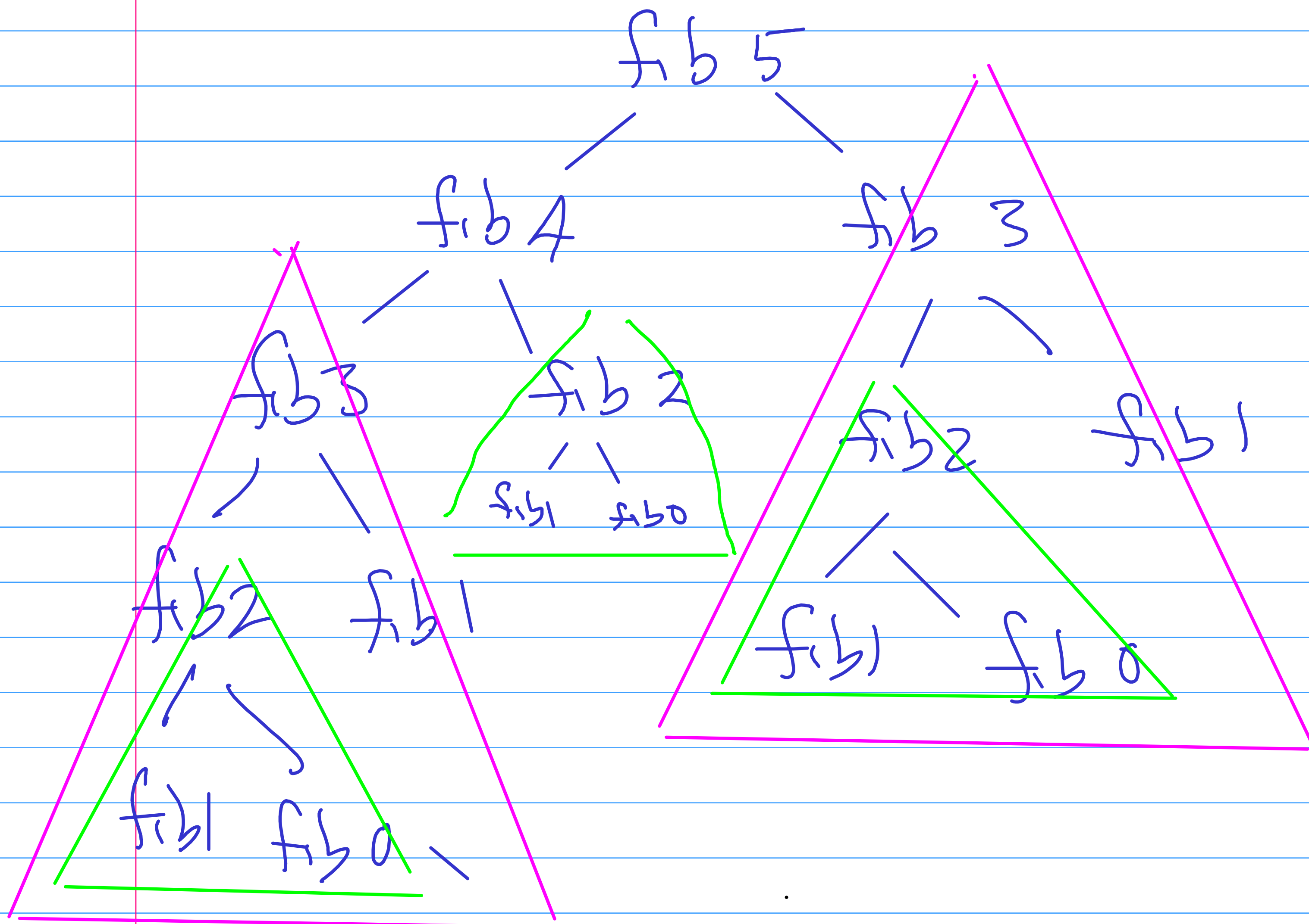
So the program computes the n^{th} fib #, for $n \geq 2$, as the sum of the $(n-1)^{st}$

and $(n-2)^{nd}$ fib #s → which is correct.

For the basis step, one must show the correctness of each case separately.

This seems easy enough — so why the abuse heaped on this poor program? The reason is that it is extremely inefficient — bad recursions are indeed to be avoided (but this does not mean that recursion is to be avoided!)

The inefficiency can be seen in a pattern of calls expansion. Eg:



The problem is that so many computations are duplicated. This program require exponential time → idea for proof is to count the number of times fib 0 and fib 1 are computed.

Could use memoization to store previously computed results to improve efficiency → but of course this requires assignment.

Let's next develop an iterative Fibonacci program.

(At the board)

```
(define (fib n)

  (define (fib-iter curr prev count)
    (cond ((= count n) curr)
          (else (fib-iter (+ curr prev) curr (+ count 1)))))

  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (fib-iter 1 0 1))))


; what are the design roles of curr and prev?
; what is the invariant?  is this version correct?
```

Question: are termination arguments always so easy?

No — go ahead and search for the '3n+1 problem' (collatz conjecture) for an example of an apparently simple while loop with completely unknown termination properties

But even here — for the recursive fib program —

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

while it is clear that the (fib (- n 1)) calls will stop —

can we be equally sure that the (fib (- n 2)) calls will stop?

Maybe we just blow through the stopping cases!

See if you can work this out.