Class 07 CSc 335 Sections M and R 02/15/24
Early Examples of Program Development

---

(1) Computing b^n, where, for simplicity, we shall assume that $n \geq 0$ is an integer, and b is just a number.

Specifically - just $\underbrace{b * b * \ldots * b}_{n \text{ times}}$

The first question: should we use recursion, or iteration? Generally it is much easier to use recursion - so we start with that. Having decided that, the next question is: how will we 'divide and conquer'? Specifically, what related (but smaller) problem will we solve, and how will we use that solution to solve the larger (main) problem?

IH

IS

$\underbrace{b * \ldots * b}_{n-1 \text{ times}} \quad * \quad b$

lazy ellipsis

Note that the first step in the induction is NOT "what is the basis step?". The reason is clear: we can't know what the basis step is until we know what we're trying to prove.

But at this point - with the divide and conquer strategy in place, we can ask: what problem of this kind cannot be subdivided? And this is the basis step — for the strategy we propose, the $n=0$ case cannot be subdivided — so we must compute it directly (without recursion).

Please notice that the strategy does not further decompose — specifically, it does NOT spell out

Evil ellipsis

"... and then we decompose the b^{n-1} problem, and then the b^{n-2} problem, ..., and so on until the b^0 problem is reached".

The divide and conquer strategy has some serious magic: the divide step includes a black-box component. We do NOT ask how b^{n-1} is computed. We simply ASSUME that it is. This relies on the Principle of Induction.

Let's take a moment to understand the difference between the EVIL and lazy ellipses.

Here it is: the lazy ellipsis can always be replaced. For example,

$$\underbrace{b * \cdots * b}_{n-1 \text{ times}}$$

can be replaced by $\prod_{i=1}^{n-1} b$.

Notational Aside

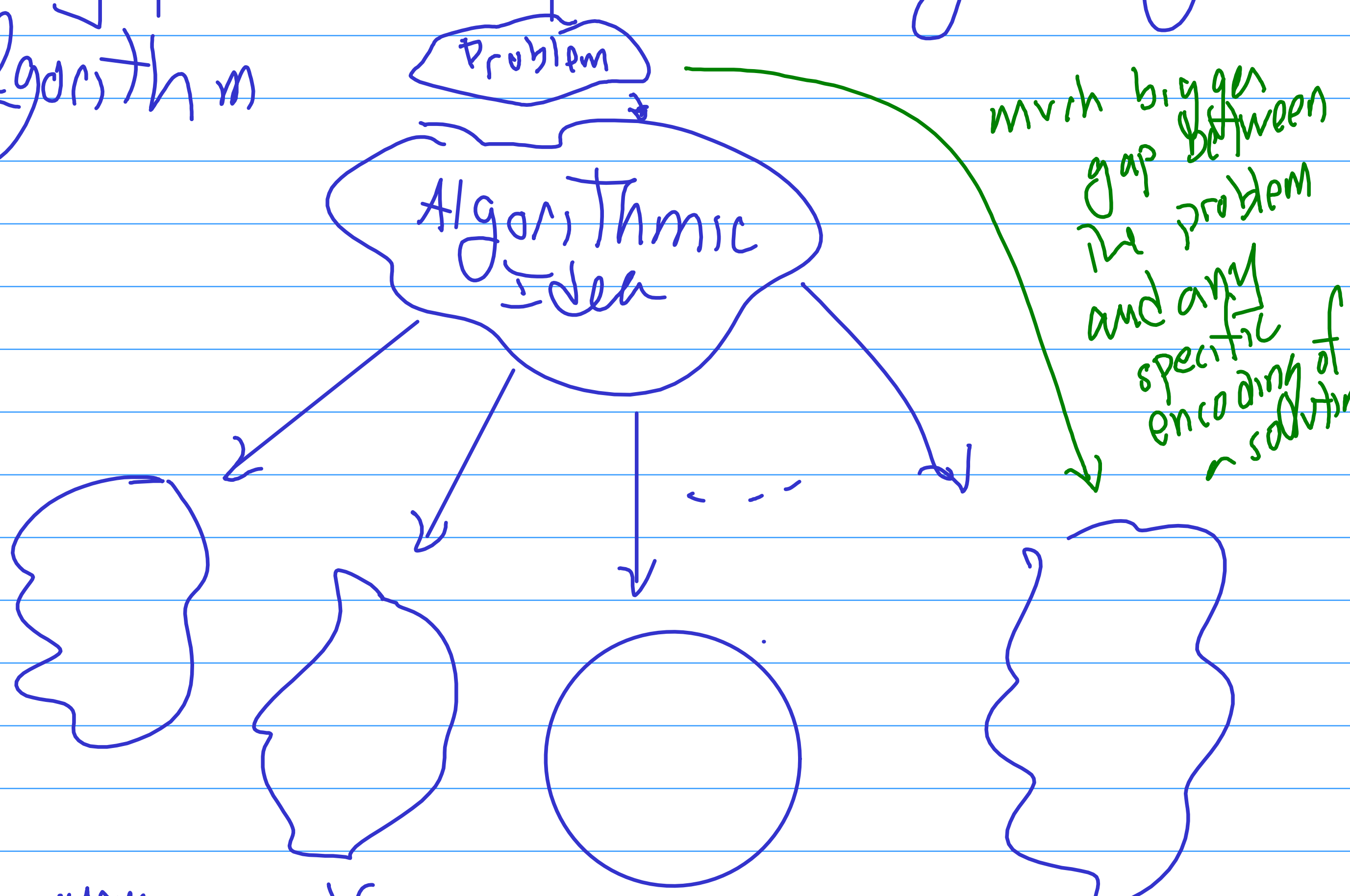
Hey, wait: i doesn't occur in the term, so what does this mean? The idea is to regard $\prod_{i=1}^{n-1} b$ as $\prod_{i=1}^{n-1} f(i)$, where $f(i)$ is the constant b for all values of i.

In other words, we wrote $b * \cdots * b$, but we did not need to.

For the EVIL ellipsis, there is no way of replacing it. One way or another, a description at that level involves the concept "eventually." You need to carry out an induction.

Another point, having to do with workflow: design in English, not in code.

Supporting argument: you will find that the human-language designs are more abstract than the computer-language designs. By this I mean that usually there are many possible computer codings of a given algorithm



many possible programs

Each program makes many specific

choices regarding eg data representation, precisely what to do next and so on. As far as solving the orig. problem is concerned, they are over-constrained.

Related: programming languages are horrors for designing solutions. So you don't want to use them for this!

Let's get back to b^n. At this point we might generate some guess-code

```
;pre: b is a number and n ≥ 0 is an integer
  (define (myexpt b n)
    (cond ((zero? n) 1)
          (else (* b (myexpt b (- n 1))))))
;post: returns b^n
```

does the pre hold here?

IH: The recursive call works, provided the pre-cond holds at the point of the call.

We have to show — in this particular case — that PRE $\wedge$ $n \neq 0$ $\Rightarrow$ b is a number and $n-1 \geq 0$

"Axiom of non-interference" if we were being very formal

This is clear: ① $n \geq 0 \wedge n \neq 0 \Rightarrow n-1 \geq 0$
② if b was a number, then — because the program never changes b — it is still a number.

So (myexpt b (- n 1)) actually does return b^{n-1}; the program multiplies this by b, hence giving the correct result.

So — are we done? Nope — still have to check the basis step.

We have to show that

$PRE \wedge (n=0) \Longrightarrow$

1 is the right answer

But this is not a valid implication, because our current PRE allows $b=0$. In case $n=0$, the result 0^0 is undefined — the program should certainly not return 1.

So, what to do?

No input checking in 335!

No error message outputs in this course!

We can change the precondition — one possibility is

$\vec{P}_1$ $b \neq 0$ is a number and $n \geq 0$ is an integer

Another — less restrictive (weaker), and so superior if it works — is

$\vec{P}_2$ b is a number and $n \geq 0$ is an integer and (b and n are not both 0)

P_1 solves our problem — for now

$b \neq 0 \wedge n = 0 \implies b^n = 1$

(This is the basis step)

You can check that the induction step is not bothered by this new precondition because ... the new precondition implies the old precondition

but it would be nice if we could use P_2. One way might be

```
; P2
(define (another-expt b n)
  (cond ((zero? b) 0)
                    ; if b≠0, (myexpt b n) computes the correct result!
        (else (myexpt b n))))
; b^n is returned.
```

$\boxed{\text{Homework}}$ - you might try using P_2 with the original program (ie, no checking whether $b=0$) — and giving a 'no-unwinding' analysis of what goes wrong.

The unwinding analysis might go as follows: if $b=0$, then since eventually $n=0$, we will be confronted with 0^0 — so the whole plan fails.

HINT: The fault rests in an invalid implication!

The termination argument - which is essentially the same as for the recursive factorial - needs yet to be given.

where it seems that one function will suffice

Grading Rubric for a problem of this kind:

1. specification — ie ~ pre and post

2. decide whether to use recursion or iteration — usually start with recursion because it's easier

2.1 Having decided on recursion

→ functional decomposition: if more than one function seems to be necessary

→ divide & conquer strategy (D & C)

→ IH: what will we induct on?

→ either the IS or the basis

for these intro number problems IS is usually some basic algebra

basis step is given in terms of the D & C

→ check that the pre condition holds when the recursive call is made

→ check that pre & basis ⇒ post condition

→ Termination arg

if any of these steps break, revise — possibly starting over

2. Invariant first / code later development of an iterative exponentiation program.

Note: The DI needs to include a rough idea of the guess invariant as well as a hint at a termination idea

We start with a design idea: we wish to compute b^n ($b \neq 0$, $n \geq 0$ — b is a number and n is an integer) in the following way: using an accumulator variable result, we will increase a counter – count – from 0 up to n, with b^count kept in result.

When we stop, count = n, and result should be b^n. We have a guess-invariant:

$$\text{result} = b^{\text{count}}$$

→ is this strong enough? [Yes: when count = n, we can return result]

→ is it weak enough? [Yes - if we set result initially to 1, we have result = b^{count} when count = 0]

→ is it preservable? IE: if result = b^{count} for the current call, is this still true on the next call? Our design idea was to increase count — presumably by 1 — so all we need do to maintain the equation is to multiply result by b.

* should have asked this first! termination is really part of the design idea

does the proposed design terminate? [Yes - incrementing count from 0 to integer $n \geq 0$ will terminate]

Generic Design Idea

"Make progress towards termination while keeping the invariant true"

guess-code

once this is a local function, the b and n parameters are no longer needed in expt-iter

```
(define (myexpt b n)
  some call to expt-iter )

(define (expt-iter b n count result)
  (cond ((= count n) result)
        (else (expt-iter b n (+ count 1)
                              (* b result))
        )))
```

suggested: make this a local function inside the wrapper

Let's test ... need first to set up
The initial call to expt-iter

The initial call is supposed to
make the invariant true the
first time expt-iter is called

```
(define (myexpt b n)
  (expt-iter b n 0 1))
```

Is there an alternate design? As engineers,
we need to do better than settle for the
first idea that occurs to us!

Try to look at your designs critically

For the program we just wrote, one
criticism might be that we need
at each step to compare count to n

* If we were to start count at n, and decrease it to 0, would our current invariant still work?

* If we were to start count at n, and decrease it to 0, would we still need all the variables?

Let's take the first question:

currently, our GI is

$$result = b^{count}$$

Does anything go wrong when we change the design?

"it is not weak enough"

One problem: on start, count = n, and so result would need to be b^n. And this means that

The problem has already been solved – there is no reason to run the program!

"does not imply post when conjoined with the stopping condition"

<u>Another problem:</u> ~~It's~~ also not strong enough – in that it does not imply the postcondition when $n = 0$ (ie, when the proposed loop stops)

Yet it seems that we ought to be able to compute b^n by counting n down to 0 – so perhaps the thing to do is to adjust our gI !

$$result = b^{n - count} \; ?$$

looks better!

so the exponent is 0 when count = n, and is n when count = 0

H/W

Suggested for you:

Work this out

ie - modify the previously given code so that this gI works