Warm-up

In the last lecture we only covered only a few basic operations available to computers: assignments, conditionals, and numeric comparisons. The usual arithmetic operators are also available, which means that we can write pseudocode like

```
let sum $\leftarrow$ augend $+$ addend
```

Although one line, this is implicitly two instructions: first to add the augend and the addend, second to replace the value of sum with the addition's result. More complicated expressions can encode even more steps, which follow the standard order of operations.

Use this knowledge to factor the following problem and devise an algorithm solving it:

> The inputs `first_number`, `second_number`, and `third_number` each hold a number. `positive_count` should, after the algorithm runs, hold the number of inputs that are positive.

Lesson Overview

In the examples so far, we've always known ahead of time how many subproblems a problem will factor into. But sometimes the number of subproblems depends on the inputs' values, something that none of our current strategies can deal with. Two of the four kinds of factoring are affected: factoring into parallel subproblems and factoring into quotient subproblems.

Variably Many Parallel Subproblems

Consider this problem, assuming that printing a number is a basic step:

> `lower_bound` and `upper_bound` each hold an integer input. The algorithm should print out all integers that are greater than or equal to `lower_bound` as well as less than `upper_bound`. `current` is available as temporary storage.

If we knew `lower_bound` and `upper_bound`, the factorization would be trivial: there would be (`upper_bound` $-$ `lower_bound`) subproblems, each of which would be to print one number:

- Printing numbers at least `lower_bound` and less than `upper_bound`
  Splits into parallel subproblems:
  - Printing `lower_bound`
  - Printing `lower_bound` $+ 1$
  - ...
  - Printing `upper_bound` $- 1$

But we don't know these values.

Our first trick will be to introduce new variables, called **loop variables**, that describe all of the ways that the possible subproblems can differ from one another, and then treat these variables as inputs to a common, repeated subproblem called the **loop action**.

In this case, the subproblems can differ only in the number that they ask us to print, so we will use the variable `current` to store that number, making the factoring look like this:

- Printing numbers at least `lower_bound` and less than `upper_bound`
  Splits into parallel subproblems differentiated by `current`*:*
  - ⟨Subproblems about computing values for `current`. . . ⟩
  - Printing `current` (action)

The obvious next question is how to fill in that placeholder. Fortunately, there's a standard algorithmic pattern, called a **loop**, that we can look at:

1. Set the loop variables for the first subproblem (**initiation**).
2. Stop if the loop variables don't describe a subproblem that needs solving (**termination**).
3. Solve the current subproblem (action).
4. Update the loop variables for the next subproblem (**consecution**).
5. Go to Step 2.

This tells us that getting `current` to have the appropriate values involves the subproblems labeled initiation, consecution, and termination.

For our example: the first subproblem is the one where `current` is `lower_bound`, we can advance from one subproblem to the next by incrementing `current`, and we do not need to solve any subproblems where `current` is at least `upper_bound`. We can fill in the factoring accordingly:

- Printing numbers at least `lower_bound` and less than `upper_bound`
  Splits into parallel subproblems differentiated by `current`*:*
  - Setting `current` to `lower_bound` (initiation)
  - Setting `current` to `current` + 1 (consecution)
  - Checking that `current` is less than `upper_bound` (termination)
  - Printing `current` (action)

All of these are things we know how to do. Our solution in pseudocode:

```
let current ← lower_bound
while current < upper_bound do
    print current
    let current ← current + 1
end
```

The pattern

```
while ⟨condition. . . ⟩ do
    ⟨Loop body. . . ⟩
end
```

means that the computer should check whether the condition evaluates to `true`, running the **loop body** and coming back to check again if so, moving on to the next step if not.

Practice

Give a factoring for and pseudocode solving the following problem:

> `upper_bound` holds a number. The algorithm should print out all integer powers of two that are less than `upper_bound`. `power_of_two` is available as temporary storage.

Variably Many Quotient Subproblems

Now we'll extend our strategy to handle quotient subproblems. For our example, we'll take a problem from the previous lectures, finding the maximum of three numbers, and extend it to finding the maximum of many:

> `lower_bound` and `upper_bound` each hold an integer input. The algorithm should compute the maximum value of a certain function f over the integers between `lower_bound`, inclusive, and `upper_bound`, exclusive, and store it in `maximum`. `current` is available as temporary storage.

(For example, if `lower_bound` is -1 and `upper_bound` is 2, we want the maximum of $f(-1)$, $f(0)$, and $f(1)$.)

If we followed the quotient structure from maximum of three, the factoring would look like this:

- Finding the maximum of f over the integers in [`lower_bound`, `upper_bound`)
  Splits into quotient subproblems:
  - Finding the maximum of the first zero numbers
  - Finding the maximum of the first one number
    by finding the maximum of the previous result and $f($`lower_bound` $+ 1)$
  - ...
  - Finding the maximum of the first (`upper_bound` $-$ `lower_bound`) numbers
    by finding the maximum of the previous result and $f($`upper_bound` $- 1)$

Note that we have to include cases for fewer than two numbers because the difference between `upper_bound` and `lower_bound` could be anything.

We can start out with the same trick as before, identifying loop variables and unifying subproblems. But, unlike with parallel subproblems, some of our work is already done: the loop variables don't have to record any information already stored in the result variables.

As written, the first value (if any) in our partial maxima is always $f($`lower_bound`$)$, but the last value changes. We'll use `current` to keep track of where we are; the last value will be $f($`current`$)$.

That lets us unify everything except "finding the maximum of the first zero numbers". Quotient subproblems often have one exception like this; it is called the **base case**. The new factoring:

- Finding the maximum of f over the integers in [`lower_bound`, `upper_bound`)
  Splits into quotient subproblems differentiated by `current`*:*
  - ⟨Subproblems about computing values for `current`...⟩
  - Finding the maximum of zero numbers (base case)
  - Finding the maximum of $f($`lower_bound`$)$ through $f($`current`$)$
    by finding the maximum of the previous result and $f($`current`$)$ (action)

And we already know how to compute values for `current`, so we can fill those parts in.

- Finding the maximum of f over the integers in [`lower_bound`, `upper_bound`)
  Splits into quotient subproblems differentiated by `current`:
  - Setting `current` to `lower_bound` (initiation)
  - Setting `current` to `current` + 1 (consecution)
  - Checking that `current` is less than `upper_bound` (termination)
  - Finding the maximum of zero numbers (base case)
  - Finding the maximum of f(`lower_bound`) through f(`current`)
    by finding the maximum of the previous result and f(`current`) (action)

Algorithmically, our general strategy is almost the same as the one we had for variably many parallel subproblems. We just need to add a step to handle the base case and then allow termination to consider other variables, like the result variables.

0. **Set the result variables to appropriate starting values (base case).**
1. Set the loop variables for the first subproblem (initiation).
2. Stop if the loop **(or other)** variables don't describe a subproblem that needs solving (termination).
3. Solve the current subproblem (action).
4. Update the loop variables for the next subproblem (consecution).
5. Go to Step 2.

Notice that Step 0 does exactly the same thing for the result variables that Step 1 does for the loop variables. So handling the base case is also called result-variable initiation. Likewise, Step 3 does for result variables what Step 4 does for loop variables; with quotient subproblems the loop action is also called result-variable consecution.

0. Set the result variables to appropriate starting values **(result-variable initiation)**.
1. Set the loop variables for the first subproblem **(loop-variable initiation)**.
2. Stop if the loop (or other) variables don't describe a subproblem that needs solving (termination).
3. Solve the current subproblem **(result-variable consecution)**.
4. Update the loop variables for the next subproblem **(loop-variable consecution)**.
5. Go to Step 2.

This strategy gives us pseudocode like the following:

```
let maximum ← ?
let current ← lower_bound
while current < upper_bound do
    if $f$(current) > maximum then
        let maximum ← $f$(current)
    end
    let current ← current + 1
end
```

The question mark is there because it might not be immediately obvious what the maximum of zero numbers is. We can work it out by trying the algorithm on a simple problem instance, say, one where `lower_bound` is 0 and `upper_bound` is 1.

The possible execution traces, depending on whether f(`current`) > ? is false or true:

1. `maximum` becomes ?.
2. `current` becomes 0.
3. `current` < `upper_bound` evaluates to `true`.
4. f(`current`) > `maximum` evaluates to `false`.
5. `current` becomes 1.
6. `current` < `upper_bound` evaluates to `false`.
7. The result is ?, not as intended unless $f(0) = ?$.

1. `maximum` becomes ?.
2. `current` becomes 0.
3. `current` < `upper_bound` evaluates to `true`.
4. f(`current`) > `maximum` evaluates to `true`.
5. `maximum` becomes $f(0)$.
6. `current` becomes 1.
7. `current` < `upper_bound` evaluates to `false`.
8. The result is $f(0)$, as intended.

So either f(`current`) > ? or f(`current`) = ? needs to hold for any value of `current`. What number is less than or equal to all others? Negative infinity:

```
let maximum ← $-\infty$
let current ← lower_bound
while current < upper_bound do
    if $f$(current) > maximum then
        let maximum ← $f$(current)
    end
    let current ← current + 1
end
```

Homework

See Worksheet 3.