Lecture 5: Algorithm Desgin and Development

Instructor: Musard Balliu, musard@chalmers.se

http://www.cse.chalmers.se/~musard

QUESTIONS?

Plan

- Last time
  1. subroutines
  2. arrays

- Today's Plan:

  1. algorithm design and development

Designing algorithms

Consider the following (exam worthy!) exercise:

Every array of integers is composed of segments that "go up" (non-decreasing, in which no element is smaller than the preceding one) and that "go down" (non-increasing, in which no element is larger than the preceding one). A segment that "goes up" followed by a segment that "goes down" is called a "hill".

Write a program to compute the length of the first hill in a given array.

Understanding the problem

The first thing to do is to make sure you understand the problem. Make sure you know what is given and what is sought, and that you understand all the terms involved. For example:

- *What is given?* An array.
  Give it a name! OK, `x`.
  Can you give an example? Sure, `x = {1, 2, 3, 2, 1}`.
  That's not a great example, we'll see that in a bit.
- *What is asked for?* A program to compute the length of the first hill of `x`.
  What is a hill? A hill is a segment that goes up followed by a segment that goes down.
  Can you give an example? Sure, `{1, 2, 3, 2, 1}`.
  Do you see a problem with this?

Understanding the problem

The first thing to do is to make sure you understand the problem. Make sure you know what is given and what is sought, and that you understand all the terms involved. For example:

- *What is given?* An array.
  Give it a name! OK, `x`.
  Can you give an example? Sure, `x = {1, 2, 3, 2, 1}`.
  That's not a great example, we'll see that in a bit.
- *What is asked for?* A program to compute the length of the first hill of `x`.
  What is a hill? A hill is a segment that goes up followed by a segment that goes down.
  Can you give an example? Sure, `{1, 2, 3, 2, 1}`.
  Do you see a problem with this? Yes, the example `x` is not general enough. Let's take `x = {1, 2, 0, -1, 3, 3, 7}`.
  What is the first hill of `x`*?* It's `{1, 2, 0, -1 }`.
  What is the program supposed to give us? The length of this hill, 4.

Corner cases

- *Consider corner cases!* What are "corner cases"?

  They are extreme cases, such as when the array is empty. OK, if the array is empty, then I'd say there are no hills. In which case, there is no length to return! What do I do?!

  Don't panic. Sometimes the exercise doesn't specify corner cases, in which case you are free to choose what you want. The examiner can't complain. But here we actually have an answer. An empty segment is both non-decreasing and non-increasing. Moreover an empty segment can be seen as being composed out of an empty non-decreasing segment and an empty non-increasing one. So it does have a hill, the empty hill, whose length is zero. OK, I guess that makes sense...

More corner cases

What if the array has only one element? Well, I guess you can say that it is composed of an empty non-decreasing segment and a one-element non-increasing segment. It's like a hill with a very "abrupt" left side. Or the other way around (a very abrupt right side). So it starts with a hill of length one.

What if the array has two elements? Well, if the two elements "go up", like $\{1, 2\}$, you can say it starts with a two element non-decreasing segment, followed by an empty non-increasing segment, so it starts with a hill of length two (very abrupt on its right side). And it's the same if they "go down" (abrupt on its left side).

Normal cases

Hmm, we have that arrays of length zero always start with a hill of length zero, arrays of length one always start with a hill of length one, arrays of length two always start with a hill of length two! Is it always the case that an array of length n starts with a hill of length n? Nonsense, those are just the corner cases. For three elements we can have length two: `{1, -1, 1}`.

So, the normal case is when we have three or more elements? I'd say so. When there are less than three elements, we don't even need to look at them.

And in the normal case, a hill is always of length at least two? Yes

Common strategies

One of the best things you can do is to examine small examples.

- *Let's look again at our* `x`. OK, `x = {1, 2, 0, -1, 3, 3, 7}`.

  What did you say the first hill was? It's `{1, 2, 0, -1 }`.

  How did you know when to stop? Well, it goes up to 2 and then down to -1, after which it goes up again.

  So when do you stop? When I go up again.

  But not when you first go up? No, of course not!

Examining examples

Don't forget corner cases!

- *What about our last example, the normal three element case?* That was `{1, -1, 1}`.

  What is the first hill? It's `{1, -1}`.

  How did you know when to stop? Well, I stopped when I was going up.

  But this was the first time you were going up! You said you need to "go up again"! Well, yes. . . . I actually meant "go up after I've been going down".

Pseudocode: making a plan

Record your findings in code-like English.

- *So, what have we learned?* Well, the following:

```
if x is empty, return 0
if x has only one element, return 1
if x has two elements, return 2
else
go through the array
  keeping track of whether up or down
  if you're going down and find
      yourself going up, you're done
return something
```

Pseudocode

Gradually refine the pseudocode.

- *The first three lines are a bit repetitive.* Yes, they can be shortened to one.

```
if x has two elements or less, return the length of x
else
go through the array
  keeping track of whether up or down
  if you're going down and find
      yourself going up, you're done
return something
```

Pseudocode

Gradually refine the pseudocode.

- How do we "go through the array"? With a loop.

  What sort of loop? I don't know. A `for` loop?

  A `for` *loop is usually used to traverse the whole array, or to take a number of steps known beforehand. Do we need to go through the whole array?* No, in general not. The first hill is usually not going to be the entire array.

  Do we know how many steps we're going to take? Not beforehand, no. Otherwise we'd already know the answer!

  That means we probably don't want a `for` *loop.* Then maybe a `while` loop?

Refining the pseudocode

When do we use a `while` *loop?* When we need to repeat a certain action as long as a condition holds.
What is the action here? We need to go through the array, checking if we're going up or down.
And what is the condition? We keep going as long as we're not going up after we've been going down.
Let's try to put that in the pseudocode! OK:

```
if x has two elements or less, return the length of x
else
while(goOn) {
  go through the array
  if you're going up after you've been going down
     then goOn = false
}
return something
```

Refining the pseudocode

What should `goOn` *be initially?* We should go on, so it should be `true`.

So we're going to go through the loop at least once? Yes, the array has at least three elements, we need to see how big the first hill is.

Then we should probably use a `do-while` *loop!* OK:

```
if x has two elements or less, return the length of x
else
goOn = true
do {
  go through the array
  if you're going up after you've been going down
     then goOn = false
} while(goOn)
return something
```

Refining the pseudocode

We need to be more precise. What does it mean, `go through the array`*?* It means, look at the next element.

How do you look at an element? By indexing, `x[i]`.

What is `i`*?* Well, initially it's zero, then one, and so on.

So we need to keep track of it? Yes.

And increase it at every step? Yes.

Refining the pseudocode

Let's write it in! OK:

```
if x has two elements or less, return the length of x
else
goOn = true
i = 0
do {
  look at x[i]
  if you're going up after you've been going down
     then goOn = false
  i++
} while(goOn)
return something
```

Refining the pseudocode

Let's see how this works so far. We're at our first time in the loop. What is the state of the computation? The what of who?

We talked about this in the beginning: the values of the variables in scope. Ah, OK. Let's see: we have `x`, our array, a boolean `goOn` which is `true`, and `i`, which is zero.

So what are we looking at? `x[0]`.

And what do we need to do? Er...`if you're going up` and so on.

Can we do that by looking at `x[0]`*?* No, we can't. That's just the first element. We haven't been "going" anywhere!

Refining the pseudocode

Indeed. In order to execute the action inside the loop, we must already have done some work outside the loop! Yes, such as seeing whether we're first going up or straight down...

How do we do that? We need to look at `x[0]` and `x[1]`. If `x[0]` `<=` `x[1]`, we're non-decreasing, so we're "going up". Otherwise, we're "going down" already.

Refining the pseudocode

Let's put that in the pseudocode! OK:

```
if x has two elements or less, return the length of x
else
goOn = true
if x[0] <= x[1]
  going up
else
  going down
i = 2
do {
  look at x[i]
  if you're going up after you've been going down
     then goOn = false
  i++
} while(goOn)
return something
```

Refining the pseudocode

General lesson: when there are corner cases, then there is some work to be done either before the loop, or after the loop, in order to be able to execute the statements inside the loop in a uniform fashion!

Now that we've done some work before the loop, let's see again what happens when we're in the loop the first time. What is the state? Again, `x` is our array, `goOn` is `true`, we now know whether we're going up or down, `i` is two, and we're looking at `x[2]`.

What do we need to do? Determine if we're now going up or down.

How do we do that? We look at `x[1]`. If it's smaller than `x[2]`, we're going up. If it's bigger than `x[2]`, we're going down.

And if they are equal? Then we're going up.

Why? That's what we said outside the loop.

Refining the pseudocode

But look: if `x = {1, 3, 2, 2}`*, what is the first hill?* It's the whole of `x`.

But after we examine the first 2*, we know we're going down.* Yes.

And when we examine the second 2*, you say we're going up!* Well...

Which means the second 2 *is not part of the hill!* That's wrong.

So what do we do if the preceding element is equal to the current one? We need to keep the direction in which we're going: if we were going up, we're still going up, and the same for going down.

Refining the pseudocode

General lesson: always examine the corner cases! (Equality, for example, is usually a corner case for "less than or equal").

We're getting close, now. Yes:

```
if x has two elements or less, return the length of x
else
goOn = true
if x[0] <= x[1]
  going up
else
  going down
i = 2
```

Continued on the next slide

Refining the pseudocode

Continued from the previous slide

```
do {
  if x[i-1] < x[i]
     we're going up
  if x[i-1] == x[i]
     we're going in the same direction as before
  if x[i-1] > x[i]
     we're going down
  if you're going up after you've been going down
     then goOn = false
  i++
} while(goOn)
return something
```

Refining the pseudocode

How do we keep track of the direction? With a boolean variable.

Just one? Yes, if `goingUp` is `false`, then we're going down.

Good, let's put that in!. OK:

```
if x has two elements or less, return the length of x
else
goOn = true
if x[0] <= x[1]
  goingUp = true
else
  goingUp = false
i = 2
```

Continued on the next slide

Refining the pseudocode

Continued from the previous slide

```
do {
  if x[i-1] < x[i]
     goingUp = true
  if x[i-1] == x[i]
     goingUp stays the same
  if x[i-1] > x[i]
     goingUp = false
  if you're going up after you've been going down
     then goOn = false
  i++
} while(goOn)
return something
```

Refining the pseudocode

Oops. Yes, when I get to the last `if`, I only know where I'm going now, but not where I was going before!

So what do you do? Well, I could have another variable, `goingUpBefore`, to remember the direction. No, wait, that's not necessary. All I need is to move the decision about `goOn` inside the first `if`! After all, that's the only time that the condition `going up after you've been going down` can apply!

Refining the pseudocode

Correct, let's do that! OK:

```
do {
  if x[i-1] < x[i]
     if goingUp == false
        // we were going down
        // now we're going up:
        // we're done!
        goOn = false
  if x[i-1] == x[i]
     goingUp stays the same
  if x[i-1] > x[i]
     goingUp = false
  i++
} while(goOn)
return something
```

Refining the pseudocode

We're close to the end! What should we return? I don't know!

Well, what is the type of what we want to return? An integer, the length of the hill.

And what integers do we have in the state? Er, the elements of the array are integers, the length of the array is an integer, and. . . aha, and `i`.

So what should we return? Er, `i`?

Refining the pseudocode

Hmm, well let's go through an example! A short one, please! OK, what happens if `x = {1, -1, 1}`? *(. . . thinking, thinking. . .)* At the end of the loop, I have `i = 3`!

What do we want to return? We want `2`. I see! The length of the hill is equal to the index of the first element after the hill! And we're doing one more increment before the test. Maybe we shouldn't do that if we know that `goOn` is `false`, then we could return `i`.

Refining the pseudocode

Good idea, put it in! OK:

```
do {
  if x[i-1] < x[i]
     if goingUp == false
        // we were going down
        // now we're going up:
        // we're done!
        goOn = false
  if x[i-1] == x[i]
     goingUp stays the same
  if x[i-1] > x[i]
     goingUp = false
  if (goOn)
     i++
} while(goOn)
return i
```

Refining the pseudocode

Remember the general lesson: always consider corner cases!
Again?!

Always! What happens if you try with `{1, 1, 1}`*?*
Uh. . . *(. . . thinking, thinking. . .)* Oops, I have a problem: I'm trying to look at `x[3]`, which doesn't exist!

So what happened? I incremented `i`, but there was no next element to look at. I need to test whether there is, in fact, a next element, otherwise we should stop.

How do we stop? We set `goOn` to `false`.

Yes, put that in and we can go on to writing Java!

Refining the pseudocode

```
do {
  if x[i-1] < x[i]
     if goingUp == false
        // we were going down
        // now we're going up:
        // we're done!
        goOn = false
  if x[i-1] == x[i]
     goingUp stays the same
  if x[i-1] > x[i]
     goingUp = false
  if (goOn)
     i++
     if i == x.length
        goOn = false
} while(goOn)
return i
```

Implementing the pseudocode

We are implementing a subroutine with the signature

```
int hillLength(int[] x)
```

Before actually implementing the code, it is a good idea to implement a number of tests in a `main`. Do not forget to test corner cases, as well as corner cases!

Initial code
Solution

Reviewing the implementation

Even if you have a program that works, it is necessary to review it. Very often, you can improve it, generalize it, or at least ensure that you will be able to solve similar problems faster in the future.

In the solution presented on the previous slide, the transliteration of the pseudocode was somewhat exaggerated, resulting in the useless line

```
goingUp = goingUp;
```

A more interesting change is to add a parameter to `hillLength` to start the computation from a given index, instead of at the beginning of the array.

Initial code
Solution

Designing algorithms: summary

1. Makes sure you understand the problem. Consider corner cases!
2. Go through a couple of examples.
3. Record your findings in pseudocode.
4. Refine the pseudocode. Don't forget to consider corner cases!
5. Implement the pseudocode.
6. Review, improve, and generalize the implementation.

Readings

Make sure you understand the design of the algorithm developed in this lecture.

Read sections 3.2 and 4.6 of Eck's book.