Algorithms And Data Structures I Course Notes

Felipe Balbi

October 19, 2019

Week 1

Learning Objectives:

- Explain in broad strokes what problems and algorithms are in Computer Science.

1.1.1 What is a Problem? What is an algorithm?

In computing we deal with problems that are addressable by computers. In other words, we deal with problems that are **computable**. The underlying language used to communicate with a computer needs to be mathematical, regardless of which *Programming Language* we use.

Computers require each and every idea to be converted into a mathematical concept (a number or a truth value).

Toy example:

```
x days of holiday total
y days of holiday used
x > y ? True  or False
x - y = Amount of days left
```

Note that in the case of `x-y` a positive result implies *True* while a negative or zero result implies *False*.

Problems can usually be solved in more than one way. The study of Algorithms and Data Structures gives us tools to decide which method is *better* than the other.

Definition 0.1 (Algorithm) *A general and simple set of step-by-step instructions which, if followed, solve a particular problem.*

Keep in mind that it's highly desirable to have a general purpose algorithm that solves many instances of similar problems. For example, instead of having an algorithm to solve $x^2 = 2$, it would be better to produce an algorithm to solve $x^2 = y$ given x and y are in $\mathbb{Z}$.

1.2.1 Al-Khwarizmi and Euclid

Algorithms predate the digital computer by hundreds of years. The word *algorithm* comes from the latinized name of Persian polymath **Al-Khwarizmi** (written as *algorithmi*).

One of the first known algorithms is Euclid's algorithm for calculating the *Greatest Common Divisor* between two numbers. It was described around 300 B.C.

An algorithm is a **mathematical concept** that can be instantiated as a computer program.

1.2.4 From mathematics to digital computers

Before we think about how to concretely describe an algorithm, we need to consider how to write the input data into a computer. It is **not** always possible to input arbitrary data into a computer. For example the number π is an irrational number (actually, it's transcendental see [1], which means it has an infinite decimal expansion; therefore we can't input π into a computer, as computer memory is a finite resource. We can only approximate it.

When approximating irrational and transcendental numbers with rational numbers, we will be left with an error in our calculations. This error is referred to as the *precision* of our calculation. The smaller the error, the more precisely correct our computer handles the input to our problem.

The need for approximations came to be before digital computers. The Egyptian-Greek mathematician, Heron of Alexandria, produced an algorithm for calculating and approximation of the square root of a number. That algorithm is called Heron's Method.

Heron's Method

Say we want to calculate $x^2 = 2$, give x to 1 d.p.

We **know** x must be $1 < x < 2$, we take the mean to get a candidate:

$\frac{1+2}{2} = 1.5$

$x_g = 1.5 \rightarrow x_g^2 = \frac{9}{4} > 2$

The answer **must** be within the interval $1 < x < 1.5$.

Thus:

$$\frac{2}{x} = x < x_g \rightarrow \frac{2}{x_g} < x$$
$$\rightarrow 1.\dot{3} = \frac{4}{3} < x < \frac{3}{2} = 1.5$$

Take mean to get new candidate:

[1] `https://www.youtube.com/watch?v=WyoH_vgiqXM`

$x\prime_g = \frac{17}{12} = 1.41\dot{6}$

correct to 1 d.p.

We can repeat this process as many times as we want in order to increase accuracy.

Week 2

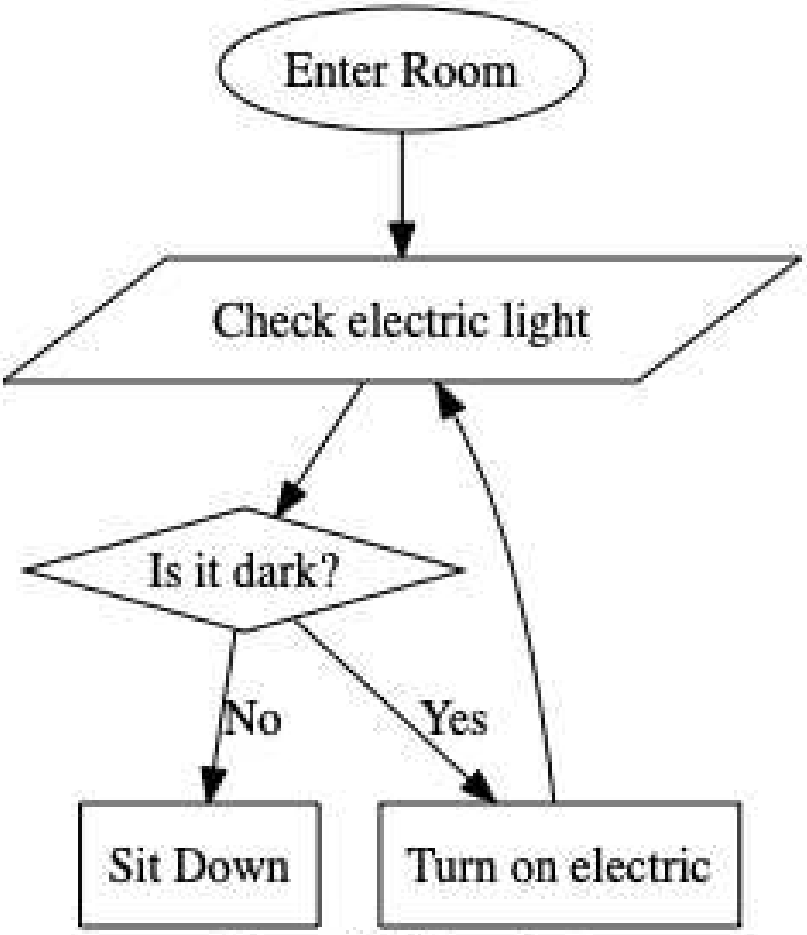
Learning Objectives:

- Recall the basic elements and construction of flowcharts.
- Express elements of simple algorithms as flowcharts.
- Explain in broad strokes what problems and algorithms are in Computer Science.

1.3.1 Introduction to flowcharts

Using flowcharts to describe algorithms. Flowcharts are abstract representations of processes such as workflow and project management. They are composed of differently shaped boxes and arrows connecting them. Boxes typically represent actions, referred to as *activities*, *states of affairs* or *decisions*. Arrows represent workflow or outcomesthat result from the decisions.

Let's look at an example below:



Example Flowchart

Flowcharts use different shapes for different meaning:

- Oval

  Ovals are Terminal nodes. They are used either as *Start* or *End* of an algorithm.

- Parallelogram

  These represent I/O actions, like gathering or displaying data.

- Arrows

  Represent control flow of the algorithm by connecting one node to another.

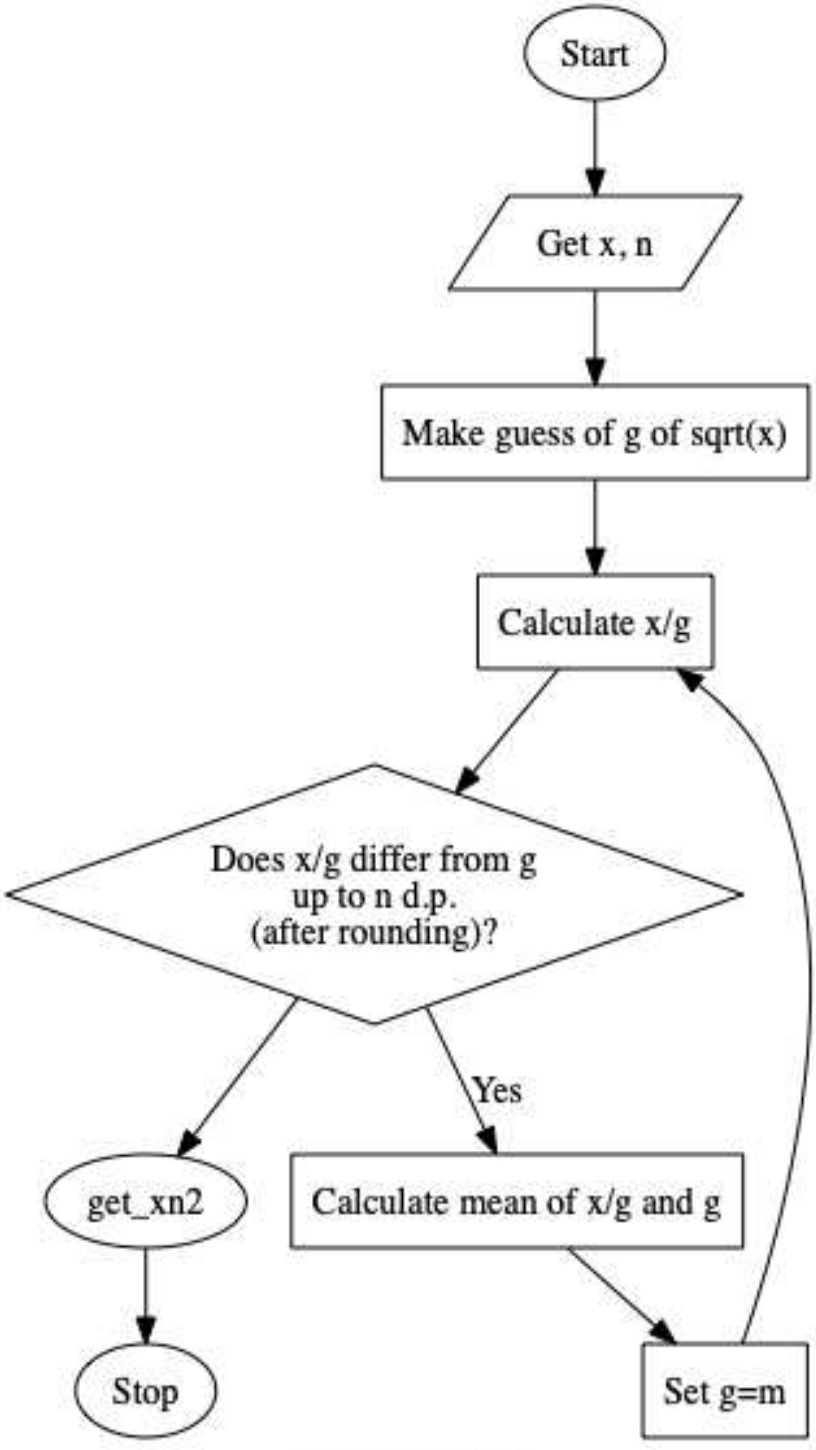
- Diamond

  Diamonds represent decisions blocks. Typically they have two outcomes: Yes/No, True/False, etc.

- Rectangle

Basic actions, turning on the light, are carried out by rectangle boxes.

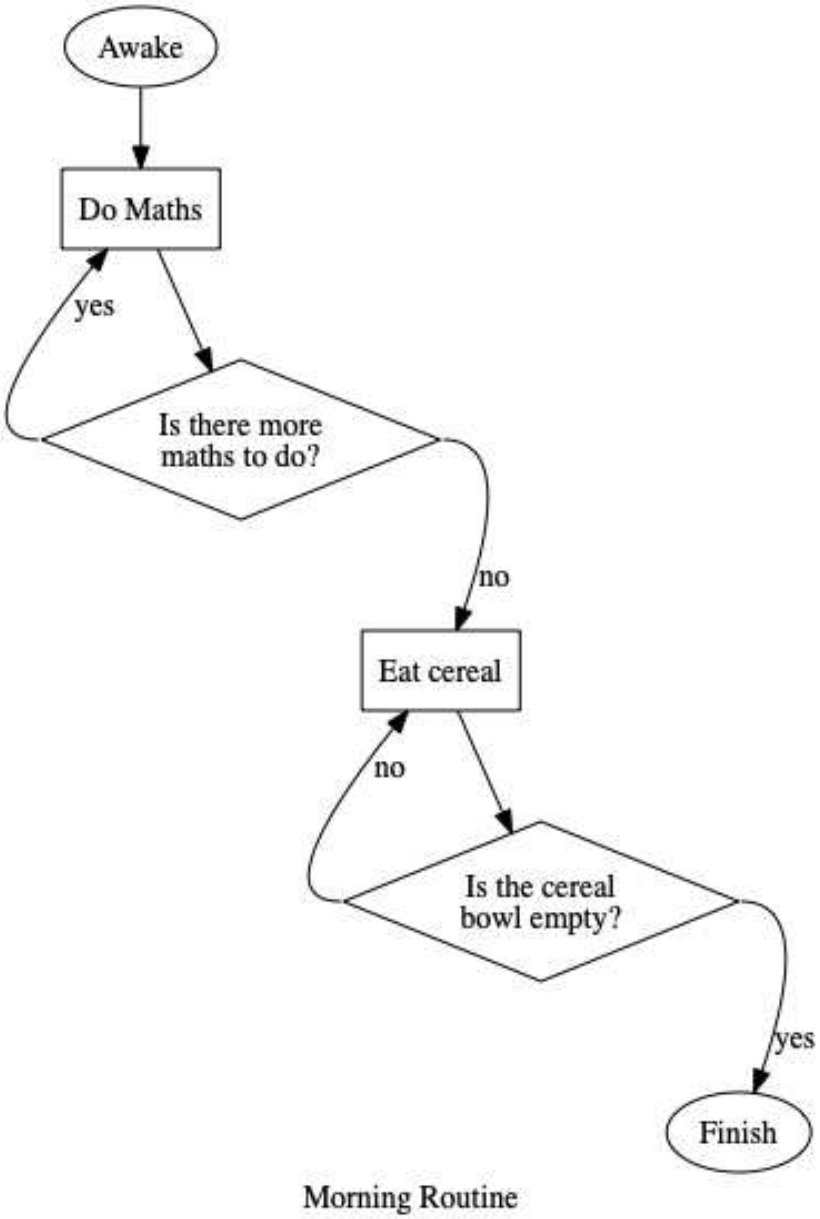
Heron's Method Flowchart



Heron's Method

1.3.4 My example of my morning routing as a flowchart

Professor Matty showed a flowchart of his morning routine. It went something like below:



Morning Routine

1.4.1 Conclusion

We learned concepts of problems, solutions, and algorithms. We learned a bit of algorithmic history and how to describe algorithms in flowcharts.

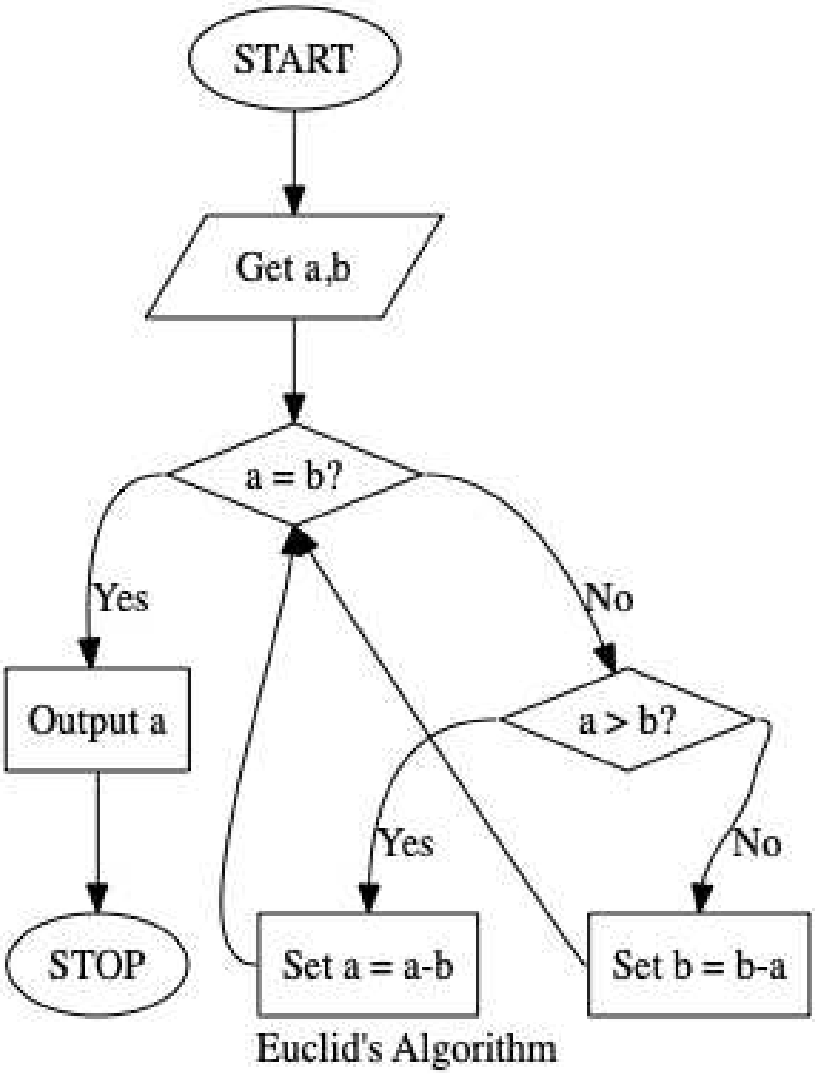
Week 3

Learning Objectives:

- Explain the necessity and concept of pseudocode.
- Describe the concept of iteration and how it is represented in pseudocode.

2.0.1 Solution to the Birthday Party Problem

The problem to be solved here is the problem of finding the *Greatest Common Divisor* of two numbers. Euclid's Algorithm is one possible algorithm for finding the GCD of two numbers.



Euclid's Algorithm

2.0.3 Introduction to Topic 2

Pseudocode is a simple way of describing and illustrating an algorithm in a language that resembles computer programs. It's widely used by Computer Scientists to discuss algorithms.

While flowcharts are an excellent way of conveying the flow of information in an algorithm and clearly showing the particular operations that are happening, drawing flowcharts is cumbersome and time-consuming.

Also, when we need to implement an algorithm, having a representation that resembles a general programming language is far better as it makes the process a little easier.

2.1.1 Discretisation and pseudocode

Discretization is the process of taking a continuous quantity and turning it into discrete steps. For example, if we were to build SW to control a thermostat until a desired temperature is reached, we would have to provide the program with discrete temperature steps (i.e. 0.5 °C) which would be incremented or decremented until the desired temperature is reached.

For this reason, pseudocode is a natural way to describe algorithms since they closely resemble computer programs. Pseudocode employs standard mathematical symbols along with a few extra bits of special notation. One such bit is the assignment symbol.

Algorithm 1 The assignment symbol

1: $x \leftarrow 2$
2: $y \leftarrow \text{TRUE}$

When using pseudocode, we should refrain from naming variables with terse names such as x, y, z, etc. As algorithms get large, it becomes to track down which letter is used for what value. Instead, we should use descriptive names such as *DesiredTemperature* for the thermostat example above. The only constraint here is that we never add **spaces** to variables names.

We read pseudocode much like English, where the order goes from left to right and from top to bottom. Assignments are also *self-referential*, which means that after a variable has been assigned a value, we assign a new value based on the variable itself:

Algorithm 2 Self-referential

1: $x \leftarrow 2$
2: $x \leftarrow x + 3$

Common symbols used in pseudocode

- Assignment: $\leftarrow$

- Arithmetic operators:
  - Addition: $+$
  - Subtraction: $-$
  - Multiplication: $\times$
  - Division: $/$
- Comparison operators:
  - Equality: $=$
  - Difference: $\neq$
  - Less than: $<$
  - Greater than: $>$
  - Less than or equal to: $\leq$
  - Greater than or equal to: $\geq$
- Logical operators
  - And: $\wedge$
  - Or: $\vee$
  - Not: $\neg$
  - if ... then ... end if

Algorithm 3 if ... then ... end if

```
1: x ← 2
2: if x > 1 then
3:     x ← x − 1
4: end if
5: if x < 0 then
6:     x ← x + 1
7: else
8:     x ← x + 10
9: end if
```

Example: Thermostat pseudocode What follows is a simple example of a real pseudocode to implement a simple algorithm that increased the temperature of a thermostat by half a degree if it's less than a threshold.

Algorithm 4 Thermostat

```
1: temperature ← 18
2: desired_temperature ← 20
3: if temperature < desired_temperature then
4:     temperature ← temperature + 0.5
5: end if
```

2.1.2 Pseudocode and functions

Pseudocode replicates other concepts from programming languages. One such concept is that of functions.

Function is a very general concept in computer science and mathematics. Functions take inputs and return outputs. For example the sum function takes two numbers as inputs and returns one number as output. Functions can return other types of values, such as Boolean values. A predicate is a function that returns a Boolean value given some input.

Here's an example function `EVEN`

Algorithm 5 The Even function

```
1: function EVEN(n)
2:     if n mod 2 = 0 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end function
```

Here, everything inside `function` and `end function` is referred to as the *body* of the function. The body is composed of *statements*. We have an `if-then` statement with a `return` statement inside of it.

A `return` statement is terminal, meaning `return` causes the function to stop.

2.2.1 Introduction to loops in pseudocode

Iteration is the idea of repeating something multiple times. Iteration is also to as *looping*. The two main looping structures are *for* loops and *while* loops. In *for* loops, we initialize a variable (e.g. i) to be our loop counter. At each iteration of the loop, we increment i by 1 until it reaches a target value, such as 10.

Algorithm 6 For Loop Example

```
1: x ← 1
2: for 2 ≤ i ≤ 10 do
3:     x ← x + i
4: end for
```

The text between **for** and **do** is referred to as the *condition* of the for loop. The text between **do** and **end for** is called the *body* of the for loop.

The basic concepts expressed in the context of the for loop, also apply to the while loop, just the structure is a little different. Here's the same algorithm from the for loop, implemented using a while loop:

With all this new vocabulary, we can implement the algorithm using pseudocode

Algorithm 7 While Loop Example

```
1: x ← 1
2: y ← 0
3: while x < 11 do
4:     y ← x + y
5:     x ← x + 1
6: end while
```

Algorithm 8 If $x^2 = n$ is x an Integer?

```
1: function IsXInteger(n)
2:     y ← FALSE
3:     for 1 ≤ i ≤ n do
4:         if i^2 = n then
5:             y ← TRUE
6:         end if
7:     end for
8:     return y
9: end function
```

The problem with this approach is that we may be squaring numbers well over than is necessary. A slight improvement can be achieved with a *while* loop.

Algorithm 9 If $x^2 = n$ is x an Integer? - While loop

```
1: function IsXInteger(n)
2:     y ← FALSE
3:     i ← 1
4:     while i^2 ≤ n do
5:         if i^2 = n then
6:             y ← TRUE
7:         end if
8:         i ← i + 1
9:     end while
10:    return y
11: end function
```

We could also employ *break* and *continue* statements to make this algorithm a little bit better. *break* stops a loop altogether while *continue* skips to the next iteration.

Algorithm 10 shows an example of both.

We should try to avoid *break* and *continue* in pseudocode.

2.2.2 Euclidean algorithm in pseudocode

With all this knowledge, we can write the Euclidean Algorithm in pseudocode. Please, refer to algorithm 11.

Algorithm 10 Break and Continue

```
1: x ← 1
2: y ← 10
3: while x < 11 do
4:     if y = 10 then
5:         x ← x + 1
6:         y ← y − 1
7:         continue
8:     end if
9:     break
10: end while
```

Algorithm 11 Euclidean Algorithm

```
1: function GCD(a , b)
2:     while a ≠ b do
3:         if a > b then
4:             a ← a − b
5:         else
6:             b ← b − a
7:         end if
8:     end while
9:     return a
10: end function
```

Week 4

Learning Objectives:

- Explain the necessity and concept of pseudocode
- Describe the concept of iteration and how it is represented in pseudocode.
- Convert a flowchart (if possible) to pseudocode

2.3.1 Conversion between flowcharts and pseudocode

After discussing the Euclidean Algorithm in both flowchart and pseudocode, we will now discuss the intricacies of converting a flowchart into pseudocode. In particular, how loops are represented in pseudocode.

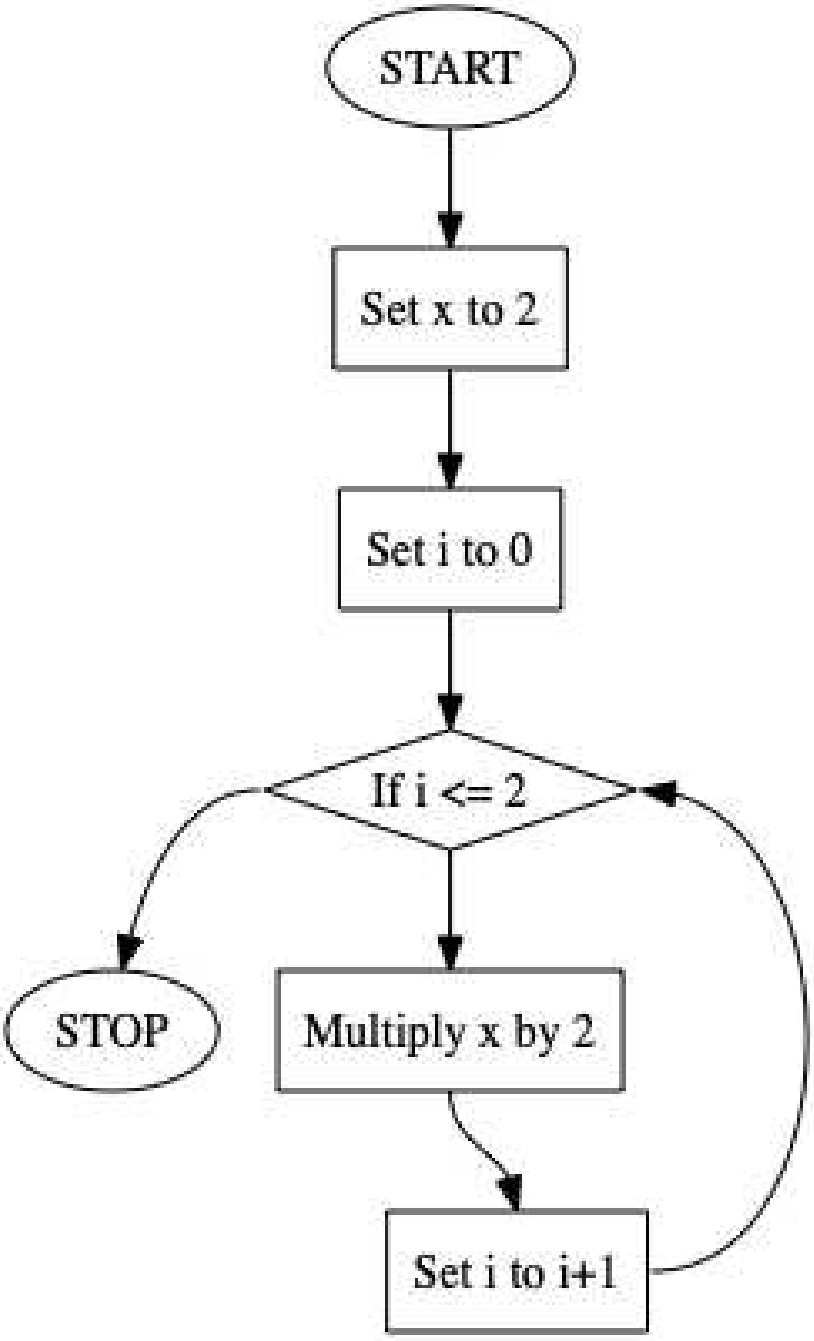
Basic Translations

Pseudocode	Flowchart
Assignments	Basic Actions
If . . . else	Diamond
function	START terminal
end function	STOP terminal
function argument	parallelogram
return	parallelogram

Pseudocode loop translation

The following pseudocode , shows how a for loop may look like in pseudocode, while the following figure shows a flowchart version of the same thing.

1: $x \leftarrow 2$
2: **for** $0 \leq i \leq 2$ **do**
3: $\quad x \leftarrow x \times 2$
4: **end for**



With this new knowledge, we can convert Heron's Method to pseudocode. See below

```
1: function SquareRoot(x , n)
2:     g ← x
3:     while ⌊(g × 10^n) + 0.5⌋ − ⌊(x/g × 10^n + 0.5)⌋ ≠ 0 do
4:         g ← 1/2 (g + x/g)
5:     end while
6:     return g
7: end function
```

2.3.4 My example in pseudocode

The lecturer showed his morning routine's pseudocode which ended up like pseudocode listing below.

```
1: function Morning(conscious, done, cereal)
2:     if conscious = TRUE then
3:         maths ← 0
4:         for 1 ≤ i ≤ done do
5:             maths ← maths + 1
6:         end for
7:         while cereal > 0 do
8:             cereal ← cereal − 1
9:         end while
10:    end if
11:    return "ready!"
12: end function
```

2.4.1 Conclusion

This is the end of topic 2: pseudocode. The main appeal of pseudocode is its universality and human friendliness. After translating flowcharts to pseudocode we were able to see how pseudocode representation can be far more compact than its flowchart version.

During this exercise, we also learned about iteration and loops. Loops, being a central idea in computer science, makes understanding them extremely important in algorithmic design and other areas.

There is more than one way of achieving looping and knowing different methods is useful depending on what we're trying to achieve.

Week 5

Learning Objectives:

- Describe the basic elements of an abstract data structure

- Explain queues, stacks and vectors in terms of their structure and operations

3.0.3 Introduction to Topic 3

How we structure data is important in everything we do in real life and in computing. We make lists of things all the time:

1. Shopping lists
2. Ingredients in a recipe
3. List of lists ;-)

We generally number items in lists and place each item in its own line. When we want to add more items, it's usually more convenient to add it to the bottom of the list.

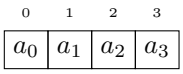
In computer science we care about how data is stored and processed on a **computer**.

When forming mathematical models of *Abstract Data Structures*, we ignore the technical details of how the data is stored (RAM, hard drive, cloud, etc) and focus on fundamental structure including how data can be amended.

3.1.1 Abstract data structure: vectors

The *Vector* is a useful abstraction of memory and simple building block of data storage. It is a **finite fixed** size sequential data collection.

Vector v



As mentioned above, a vector is fixed size. So the number of elements it has is fixed and cannot be altered. The number of elements in a vector is called the **length** of the vector. In the example above, the length is 4.

One nice aspect of vectors is that the address of element also conveys useful information. Usually, we refer to the *address* of an element as the *index* of the element within a vector.

Vector Operations

An *Operation* is a function on the vector where, if given a vector, an output can be produced, such as a number or an element of the vector.

The following table summarizes our vector operations.

Operation	Pseudocode	Description
length	$LENGTH[v]$	Returns number of elements
select[k]	$v[k]$	Returns k^{th} element from the vector
store![o, k]	$v[k] \leftarrow o$	Sets the k^{th} element of the vector to value o
construct new Vector	**new** $Vector\, w(n)$	Makes a new vector w of length n

Note that store![o, k] is the only operation that actually **modifies** the vector. Because the length of the vector is fixed, we cannot **delete** an element from the vector or **add** a new element to the vector.

It's true that we don't really know ahead of time the size of the elements stored in vectors; which means the amount of space allocated for items would have to change depending on the data type.

There is an elegant solution to this: instead of storing the data itself inside vectors, we store **references** to data. This is like storing the data in different containers and letting our vectors contain a simple number which tells us which container to look for the actual data.

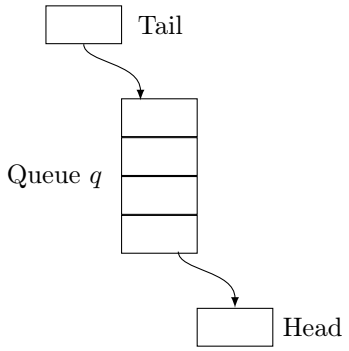
While vectors may seem a bit too simplistic at first, they are a building block for more complex data structures and provide just enough power for us to describe more complex structures.

3.2.1 Abstract data structure: queues

Queues are all around us: waiting for a table at a restaurant, waiting on a customer support line, waiting for your turn at the cashier of a shop.

The fundamental concept underlying a queue is that there is **resource** which cannot be made immediately available, so there needs to be a wait. In broad terms, the longer you have been waiting, the sooner the resource will be available to you (*First Come, First Served*, *First In, First Out (FIFO)*).

Below we can find a visual representation of a queue.



We can add new elements to the queue, which means the length of a queues is dynamic or extensible. Elements can only be added to the tail of the queue and removed from the head of the queue.

Queue Operations

Operation	Pseudocode	Description
head	$HEAD[q]$	Returns the head of the queue
dequeue!	$DEQUEUE[q]$	Removes elements from the head of the queue
enqueue![o]	$ENQUEUE[o, q]$	Adds element to the tail of the queue
empty?	$EMPTY[q]$	Returns true is queue is empty, false otherwise
construct new Queue	**new***Queue q*	Makes a new queue q

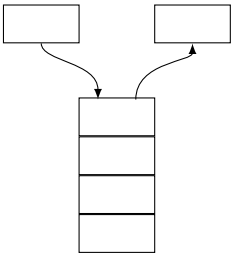
Week 6

Learning Objectives

- Describe the basic elements of an abstract data structure
- Explain queues, stacks and vectors in terms of their structure and operations
- Compare these three different abstract data structures of vectors, stacks and queues

3.3.1 Abstract data structure: stacks

The stack has certain similarities with the queue but there is one major difference. While the queue adds items to the tails and removes from the head, the stack adds **and** removes from the head. This turns it into a LIFO (*Last In, First Out*). When talking about stacks, the *head* of the stack is called the *Top* of the stack, and that's the only element that's accessible.



Stack operations

Operation	Pseudocode	Description
push![o]	$PUSH[o, s]$	Adds a new element to the top of the stack
top	$TOP[s]$	Returns the element at the top of the stack
pop!	$POP[s]$	Removes the element at the top of the stack
empty?	$EMPTY[s]$	Checks whether the stack is empty
construct new Stack	**new***Stack s*	Makes a new stack s

3.3.3 Solution to the conversion problem

Please watch the video, we're supposed to solve this by ourselves before watching the video. For reference, the task was to provide a solution to the problem of binary number representation using Stacks.

3.4.1 Summary of abstract data structures

We have covered three fundamental data structures: vector, queue and stack. We have also discussed a useful abstraction for how we manipulate data.

A vector has a fixed length while stacks and queues can grow and shrink as needed. A vector allows us to access (and modify) all of its elements while stacks allow access only to the top and queues limit access only to the head and the tail.

Dynamic sets are a collection of data that are extensible in some way.