Algorithms And Data Structures I Course Notes

Felipe Balbi

October 29, 2019

Contents

1 Week 1

Learning Objectives:

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

1.1 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 1.1 (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 1.2.1 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.

1.3 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.

1.3.1 Heron's Method

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

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

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:

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

Take mean to get new candidate:

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

correct to 1 d.p.

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

2 Week 2

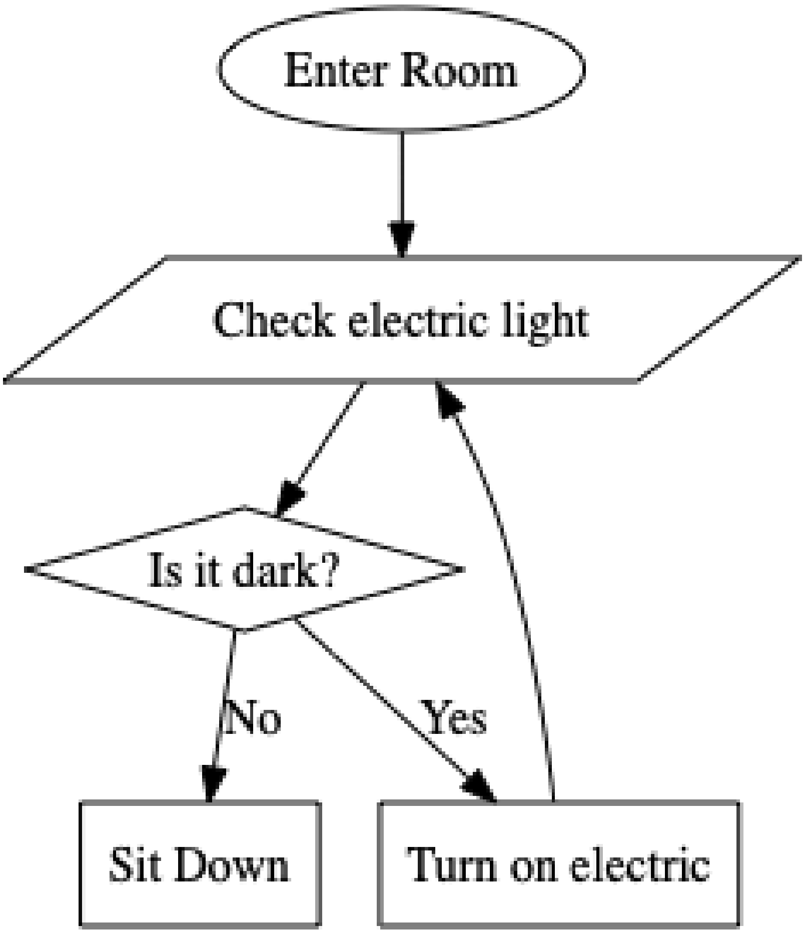
Learning Objectives:

- 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.

2.1 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.

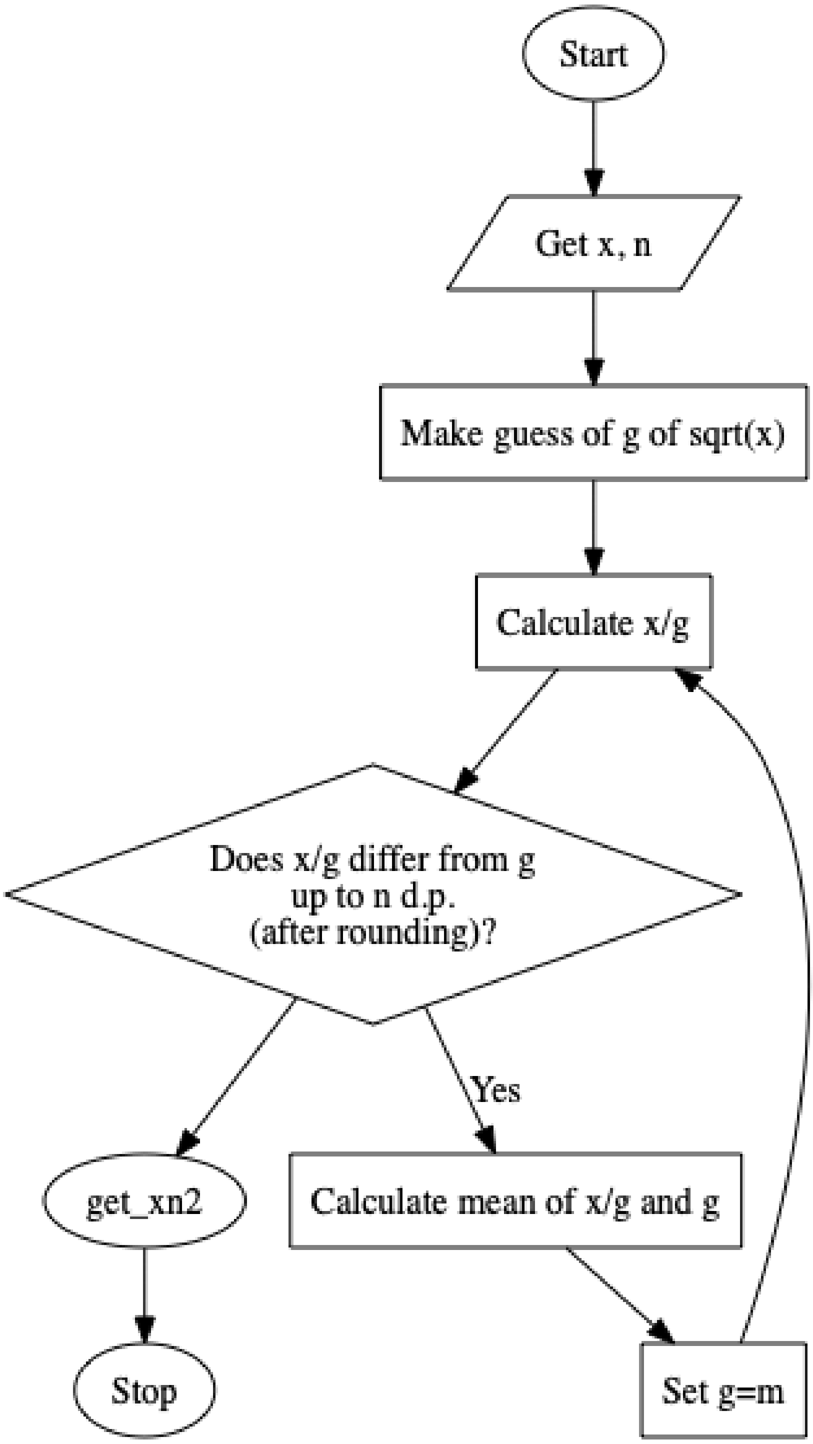
- Diamond

  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.

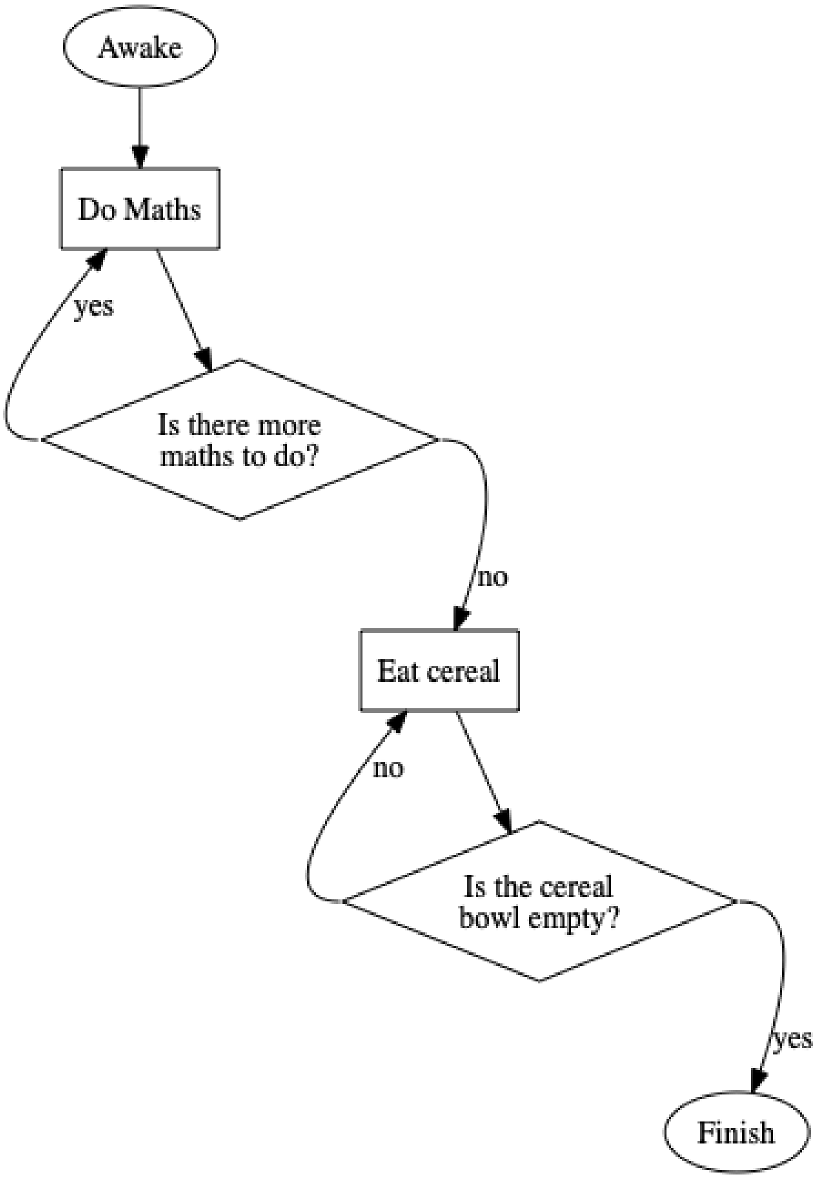
2.1.1 Heron's Method Flowchart



Heron's Method

2.2 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

2.3 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.

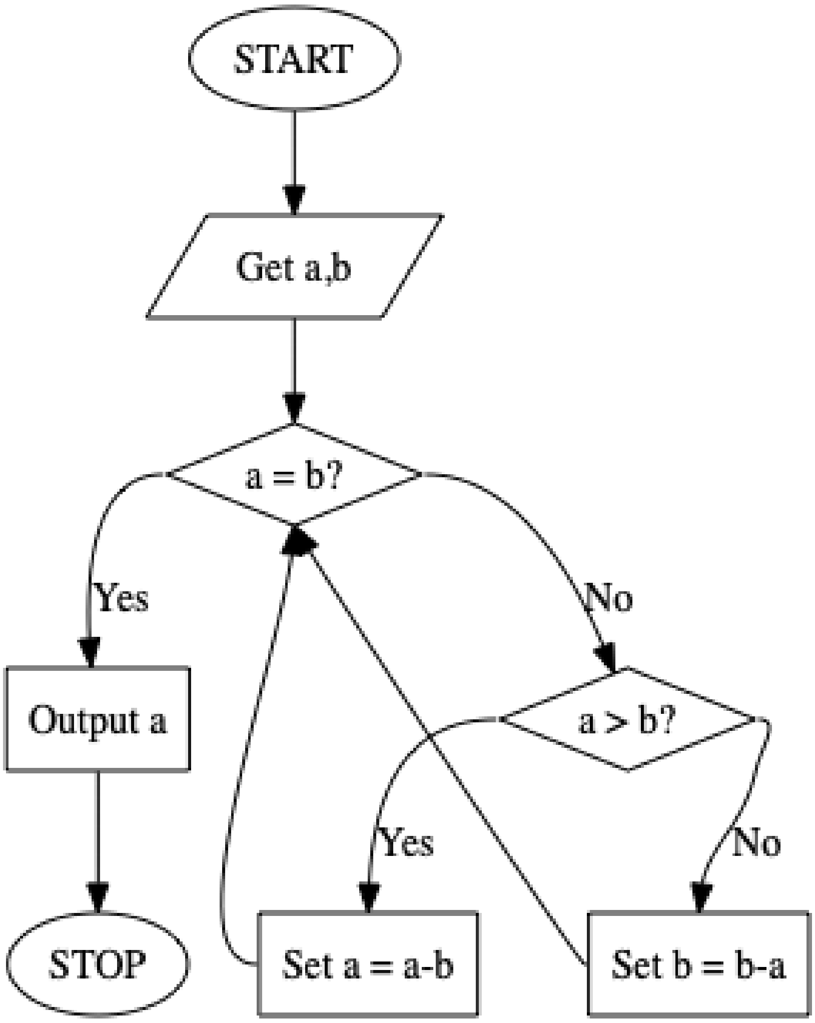
3 Week 3

Learning Objectives:

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

3.0.1 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

3.0.2 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.

3.0.3 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

```
$x \leftarrow 2$
$y \leftarrow$ 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

```
$x \leftarrow 2$
$x \leftarrow x + 3$
```

Common symbols used in pseudocode

- Assignment: ←

- 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

```
$x \leftarrow 2$
**if** $x > 1$ **then**
    $x \leftarrow x - 1$
**end if**
**if** $x < 0$ **then**
    $x \leftarrow x + 1$
**else**
    $x \leftarrow x + 10$
**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

```
$temperature \leftarrow 18$
$desired\_temperature \leftarrow 20$
**if** $temperature < desired\_temperature$ **then**
    $temperature \leftarrow temperature + 0.5$
**end if**
```

3.0.4 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

```
function EVEN($n$)
    if $n \mod 2 = 0$ then
        return $TRUE$
    else
        return $FALSE$
    end if
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.

3.0.5 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

```
$x \leftarrow 1$
for $2 \leq i \leq 10$ do
    $x \leftarrow x + i$
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

```
$x \leftarrow 1$
$y \leftarrow 0$
while $x < 11$ do
    $y \leftarrow x + y$
    $x \leftarrow x + 1$
end while
```

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

```
function IsXInteger($n$)
    $y \leftarrow FALSE$
    for $1 \leq i \leq n$ do
        if $i^2 = n$ then
            $y \leftarrow TRUE$
        end if
    end for
    return $y$
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

```
function IsXInteger($n$)
    $y \leftarrow FALSE$
    $i \leftarrow 1$
    while $i^2 \leq n$ do
        if $i^2 = n$ then
            $y \leftarrow TRUE$
        end if
        $i \leftarrow i + 1$
    end while
    return $y$
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.

3.0.6 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

```
$x \leftarrow 1$
$y \leftarrow 10$
while $x < 11$ do
    if $y = 10$ then
        $x \leftarrow x + 1$
        $y \leftarrow y - 1$
        continue
    end if
    break
end while
```

Algorithm 11 Euclidean Algorithm

```
function GCD($a$ , $b$)
    while $a \neq b$ do
        if $a > b$ then
            $a \leftarrow a - b$
        else
            $b \leftarrow b - a$
        end if
    end while
    return $a$
end function
```

4 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

4.1 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.

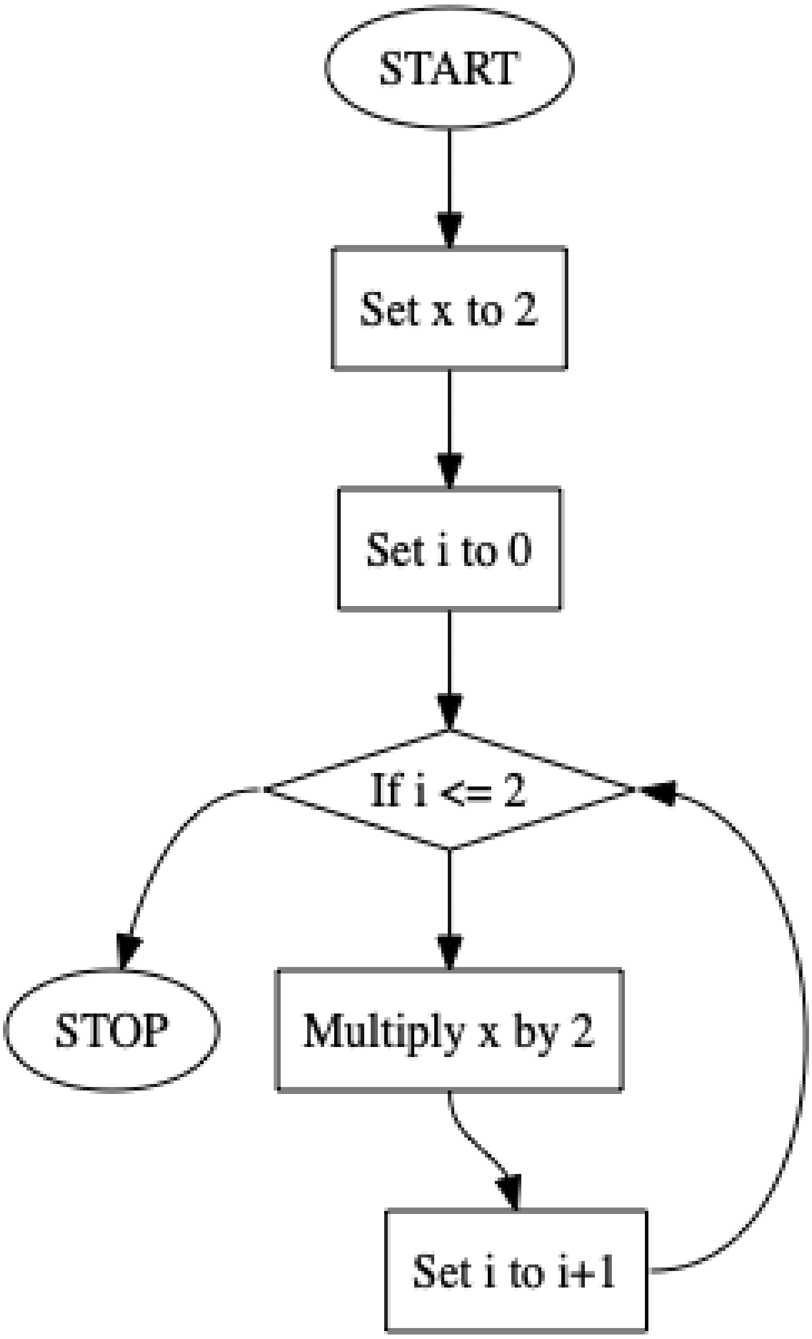
4.1.1 Basic Translations

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

4.1.2 Pseudocode loop translation

The following pseudocode 4.1.2, shows how a for loop may look like in pseudocode, while the following figure shows a flowchart version of the same thing.

```
$x \leftarrow 2$
**for** $0 \leq i \leq 2$ **do**
    $x \leftarrow x \times 2$
**end for**
```



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

```
function SquareRoot($x$ , $n$)
    $g \leftarrow x$
    while $\lfloor(g \times 10^n) + 0.5\rfloor - \lfloor\left(\frac{x}{g} \times 10^n + 0.5\right)\rfloor \neq 0$ do
        $g \leftarrow \frac{1}{2}\left(g + \frac{x}{g}\right)$
    end while
    return $g$
end function
```

4.2 2.3.4 My example in pseudocode

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

```
function Morning($conscious$, $done$, $cereal$)
    if $conscious = TRUE$ then
        $maths \leftarrow 0$
        for $1 \leq i \leq done$ do
            $maths \leftarrow maths + 1$
        end for
        while $cereal > 0$ do
            $cereal \leftarrow cereal - 1$
        end while
    end if
    return ”$ready$!”
end function
```

4.3 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.

5 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

5.1 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.

5.2 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

0	1	2	3
a_0	a_1	a_2	a_3

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.

5.2.1 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	$\textbf{new}\, Vector\, w(n)$	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.

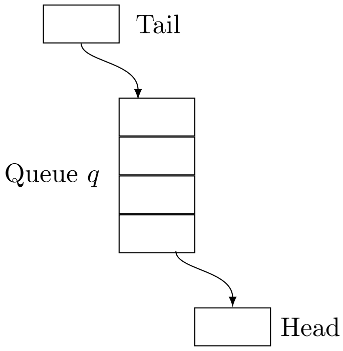
While vectors may seem a bit too simplistic at first, they are a building block for more complex data structures and provide just enough power for us to describe more complex structures.

5.3 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.

5.3.1 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	$\textbf{new}\mathit{Queue}\, q$	Makes a new queue q

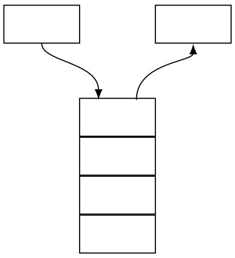
6 Week 6

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

6.1 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.



6.1.1 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	$\textbf{new}\mathit{Stack}\, s$	Makes a new stack s

6.2 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.

6.3 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.

7 Week 7

Learning Objectives

- Explain the difference between an abstract data structure and a concrete data structure
- Explain how abstract data structures can be implemented by arrays and linked lists
- Describe the linear search algorithm

7.1 4.0.3 Arrays

We start looking at searching, which is finding a desired element within a collection of data.

Different abstract data structures affect how we search.

Utilizing the basic operations and abstract data structures already defined, we can construct different algorithms. There is a distinction made in Computer Science between what's called an *Abstract Data Type* and a *Data Structure*.

An *Abstract Data Type* consists of the data we have, the values the data can take, and the allowed operations on this data. A data structure is the more concrete way in which many pieces of data are stored, managed and manipulated by the computer.

When it comes to implementation of *Abstract Data Type* we need a more concrete *Data Structure* to use.

One example is the *Vector* which is allowed an operation called **length** which tells us the number of elements in the vector. The definition of the vector, however, does not tell us how this **length** operation is actually calculated. This is why we need a concrete *Data Structure*. In this case, the *Array*.

The *Array* is a very common data structure that many programming languages have built-in support for manipulating.

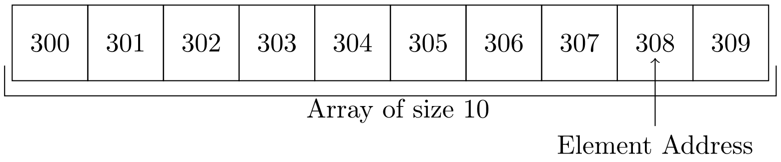
7.1.1 Definition of an Array

An array is essentially a block of memory on which we can store a collection of data. Typically, the elements of an Array are all of the same type, whether they are integers, floating points, or Booleans.

We also need a reference to the Array, so we know where in memory it's stored.

JavaScript Arrays, however, can hold data of different types.

Array Representation



Note that Arrays are a block of contiguous memory [2]. In this way, we can view the Array as a line of integers.

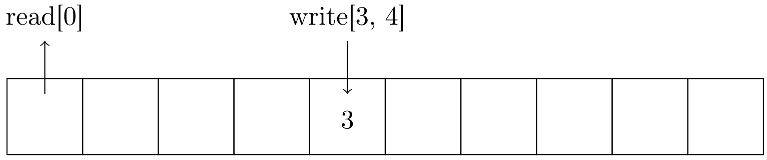
Much like a book, which have consecutively numbered pages of content, the addresses of the array are like the page numbers and the contents of the array are like the contents of a given page.

Normally, we don't deal with memory addresses directly. Programming languages give us a better way of accessing an array element with consecutive numbers starting at 0.

Just like vectors, the size of a simple array cannot be altered. But we can create a new array of larger size and copy previous elements to this new array.

Once an array is created, a computer can read values of elements and write (including overwrite) elements of the array.

Array Representation



[2] Contiguous Memory means that each address of the Array is consecutive.

7.2 4.1.1 Dynamic arrays

While vectors have immutable sizes, queues and stacks are extensible. Therefore, we can't implement queues and stacks with simple arrays of fixed sizes.

In order to implement stacks and queues we need a new abstract data structure called the *Dynamic Array*.

Since the *Dynamic Array* is an abstract data structure, we need to start defining its operations.

7.2.1 Dynamic Array Operations

The *Dynamic Array* is a collection of elements just like the vector, but it's not a fixed size data structure like the vector. Because it's like the vector, the *Dynamic Array* has all the operations of a vector.

Operation	Pseudocode	Description
length	$LENGTH[d]$	Returns number of elements
select[k]	$d[k]$	Returns k^{th} element
store![o, k]	$d[k] \leftarrow o$	Sets the k^{th} element to value o
removeAt![k]	$d[k] \leftarrow \emptyset$ $(k \leq LENGTH[d]$	Removes k^{th} element, return the element's value
insertAt![o, k]	$d[k] \hookleftarrow o$ $(k \leq LENGTH[d] + 1$	Inserts a new element, increasing the length

Given this abstract data structure, we can implement it with an array. Whenever we need to add a new element to an array, we allocate a new, larger array, copy elements from the old array to the new array and add the new element.

Like with Arrays, we will have element at index 0 to keep hold of the length of the array. When implementing *removeAt![k]* we need to go through three steps (assuming an array of 3 elements):

- /removeAt![2]
  - *write![Element 3,2]*

    Write the contents of element 3 to index 2
  - *write![,3]*

    Write **nothing** to index 3
  - *write![2,0]*

    Update the Array's length

Insertion is similar. Assuming, again, an array of 3 elements, if we want to insert a new element at index 2:

- *insertAt![Element 4,2]*

  Insert the new element 4 at index 2
  - *write![4,0]*

    Write the length 4 to index 0

- *write![Element 1,1]*

  Copy Element 1 from old to new array
- *write![Element 4,2]*

  Write new Element 4 to new array
- *write![Element 2,3]*

  Copy Element 2 from old to new array
- *write![Element 3,4]*

  Copy Element 3 from old to new array

7.2.2 Similarities among Vector, Queue, Stack and Dynamic Array

In all of these abstract data structures, the data can be formed in a line. This is because the data contained within these data structures are sequential, with one after the other.

For this reason, these data strutures are referred to as *Linear Data Structures.*

7.3 4.2.1 Linear search algorithm

When designing algorithms we have to be wary of implementation details as implementation of different data structures may be more complex or costly than others.

Given an abstract data structure such as a queue, a stack or a vector, is there an element contained within this data structure with the value 6? A similar problem is that of finding out which element within the data structure contains the value 23.

To answer these questions, we start to discuss the *Linear Serch Algorithm.*

The answer to the first problem will be a Boolean (true or false) value while for the second problem it'll be e.g. an integer. Also, the first problem, makes sense for a stack a queue but the second does not.

A vector or a dynamic array will be useful at solving the second problem, since we can return the index of the element.

What this shows is that the problem we want to solve must make sense with respect to the input.

Since we don't know anything about the contents of vector, we can't make assumptions about where the value may be. Indeed, the value may not even be at the vector at all. Therefore, we may very well start at the beginning.

7.4 4.2.3 Searching π

This video contains a solution to a problem proposed on the previous video.

```
function LinearSearch($v$, $item$)
    for $1 \le i \le LENGTH[v]$ do
        if $v[i] = item$ then
            return $i$
        end if
    end for
    return $FALSE$
end function
```

7.5 4.2.5 Searching stacks and queues

This video cotnains a solution to the problem of searching stacks and queues proposed at the end of the previous video.

8 Week 8

Learning Objectives

- Explain the difference between an abstract data structure and a concrete data structure
- Explain how abstract data structures can be implemented by arrays and linked lists
- Describe the linear search algorithm

8.1 4.3.1 Linked lists

Before we define the *Linked List* we need to gain an understanding of what a pointer is. A pointer is, simply, a variable that stores a memory address, therefore “pointing” to that memory address.

Note that the pointer does not store the value at the memory address, instead the pointer stores the address that contains the value.

We can make a useful analogy with the index of a book. The index does not contain the contents of the book, instead the index “points” us to which stores that content. Note that the index is, itself, contained within pages of the book. Similarly, the pointer is piece of memory, which an address and that address stores the address to another location in memory. In figure 1 we have a visual representation of a pointer.

Because the pointer doesn’t store the value we’re looking for, it merely points to it or *references* it, whenever want to access the value pointed to by a pointer, we must **dereference** the pointer.

To understand dereferencing, we can go back to our book analogy. Whenever we visit the book index to look up the page number to read about a certain subject, that’s the same as reading the pointer. If after reading the pointer, we,

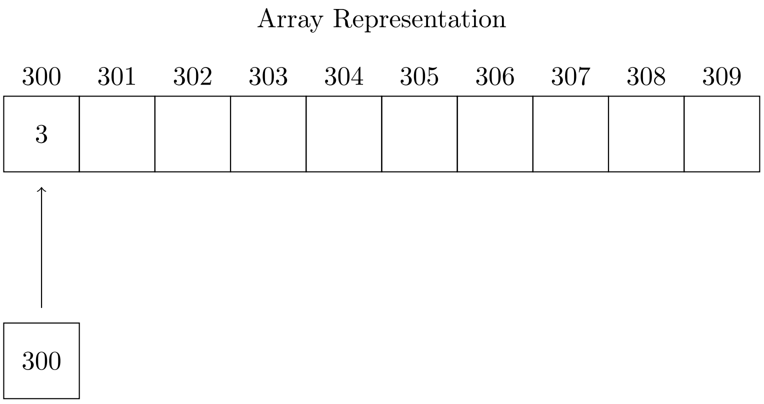


Figure 1: 10-element Array and a Pointer

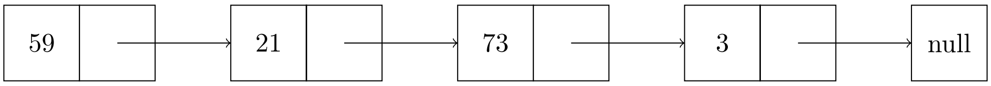


Figure 2: Singly Linked List

then, go read the page pointed to by the index, then we're **dereferencing** the page number pointer.

In other words, to dereference a pointer is to read the memory address pointed to by the pointer.

With this introduction to pointers, we can start building up our understanding of linked lists. Imagine that a linked list *node* is an array of two items where the first contains a value and the second contains a pointer to another /node. Like depicted by figure 2.

Note that each node of a linked list can be anywhere in memory. As long as we can point to it, there are no restrictions.

There is some peculiar terminology with linked lists. The pointers pointing to the next node are referred to as *next.* We also hold another pointer that points to the first item of the list and we call it *head.* The very last pointer on a singly linked list points to the a special address called *null.* This acts like a end-of-list marker.

A empty linked list consists of our *head* pointer referencing the *null* address.

One particularly benefitial property of Linked Lists is that's fairly easy (and computationally inexpensive) to insert or remove elements at arbitrary positions in the linked list.

Unlike arrays, in a linked list we **must** follow the pointers until we to the node we want.

There are three possible cases for adding a new node to existing list:

- Add node to head of the list

  In this case, we allocate memory for a new node, make its *next* pointer point to the old *head* of the list, then modify *head* so it points to our new node.

- Add node to the tail of the list

  Again, we start by allocating memory for a new node, then we traverse the list until we find a *next* pointer whose value is *null*, that means it's the end of the list. We modify that pointer to point at our new node and make the new node's *next* point to null.

- Add node anywhere else in the list

  Allocate memory for a new node, then we traverse the list until we find the exact location where we want to add a node. Modify that node's *next* field to point at our new node and make our new node's *next* field point to the following node.

Comparing with dynamic arrays, whenever we need to grow the array, we must allocate a bigger array and copy all elements over from old array to new array.

This makes linked lists more appealing for implementing stacks and queues.

When it comes to deleting nodes from a linked list, the operation is analogous to insertion and there are, again, three cases:

- Delete first node on the linked list

  Modify *head* to point to first nodes' *next* pointer. Then free the memory of the first node.

- Delete last node on the linked list

  Traverse the list until we find the penultimate node. Make its *next* pointer point to *null* and free the memory of the old last node.

- Delete node anywhere else on the list

  Traverse the list until we find the node previous to the node we want to delete. Make its *next* pointer point to the next node's *next* pointer. Then the free the memory of the element we just removed from the list.

Searching a linked list requires us to traverse the list until we find the element we're looking for.

9 Week 9

Learning Objectives

- Explain the bubble sort in terms of comparisons
- Understand insertion sort and how it differs from bubble sort

9.1 5.0.1 Solution to the Lottery Problem

We start to look at searching algorithms by analysing the Bubble Sort algorithm. We also learn that the ability to sort data makes searching for it a lot easier, which makes Sorting & Searching algorithm closely related.

9.2 5.0.3 Introduction to Topic 5

If we can find a way to sort a data structure, it generally makes it easier to ask different questions from the sorted data.

The structure of this new problem (sorting) is so that the input to the problem is a data structure and the output is another (sorted) data structure.

The output data structure should have the same values as the input – i.e. there should be **no** data loss –, albeit all elements being sorted according to some order.

9.3 5.1.1 Bubble sort

Whenever we want to sort data we need to know which operations are available for us. In other words, we need to know what kind of data structure we're dealing with.

Moreover, we need to know the kind of data is currently being held in our collection. For example, if we're dealing with numbers, we may want to sort them from smallest to largest and if we're dealing with strings of english characters, we may want to sort in alphabetical order (called *lexicographical order*).

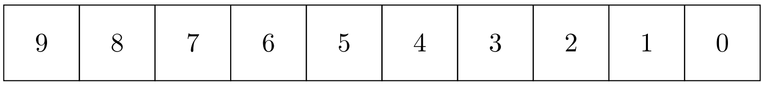
Figure 3 shows a vector randomly sorted, i.e. it is unsorted. After looking at figure 3 we may be tempted to start moving elements around and placing them in the correct places. Remember, however, that no data structure provides an operation akin to picking things up and moving them around.

Elements have a fixed location. We can read values from and write values to the elements' locations in memory. In order to *swap* the values of two elements we will use a function called *Swap*, which is provided in algorithm listing 12.

Algorithm 12 Swap Function

```
function SWAP($vector$, $i$, $j$)
    $x \leftarrow vector[j]$
    $vector[j] \leftarrow vector[i]$
    $vector[i] \leftarrow x$
    return $vector$
end function
```

Given algorithm listing 12, what remains is a method for comparing two elements and, based on such comparison, swap them. There are many ways of achieving this. One such way is the basis for the *Bubble Sort* algorithm which traverses the vector and compares adjacent elements. If the first element has a



9	8	7	6	5	4	3	2	1	0

Figure 3: Shuffled vector

value larger than the second, we swap them using our swap function, otherwise we don't do anything at all.

Bubble Sort may require multiple passes through the vector until it is fully sorted. Note that in a vector with n elements, we will need up to $n-1$ passes on the vector to fully sort it. Algorithm listing 13 shows an implementation of *Bubble Sort* in pseudocode.

Algorithm 13 Bubble Sort

```
function BUBBLE SORT($vector$)
    $n \leftarrow LENGTH[vector]$
    for $1 \leq i \leq n-1$ do
        $count \leftarrow 0$
        for $1 \leq j \leq n-1$ do
            if $vector[j+1] < vector[j]$ then
                $Swap(vector, j, j+1)$
                $count \leftarrow count + 1$
            end if
        end for
        if $count = 0$ then
            break
        end if
    end for
    return $vector$
end function
```

Note that this algorithm requires two loops. The first counts the number of passes made by the algorithm (using variable i) and the second loop is the one that actually compares (and maybe swaps) neighboring elements of the vector (using variable j). There is a also a counter *count* which is incremented each time the *Swap* function is called.

If no swaps have been done, this means the vector is sorted, so we use that to break out of the outter loop and return the sorted vector.

One interesting property of *Bubble Sort* is that after each pass of the algorithm, the largest value of the vector will be place at the end of vector, i.e. it will be sorted.

Because we're only using *vector* operations which exist on the *array* data structure, we can implement *Bubble Sort* directly for arrays.

9.4 5.1.3 Bubble sort on a stack

To implement *Bubble Sort* on a stack, we need the help of another stack which will be returned as our sorted stack.

The helper stack will start empty. From there, we pop the first item of our unsorted stack and push it to our secondary stack. Then we compare the tops of both stacks. If top of secondary stack is smaller than the top of primary stack, then we don't have anything to do, so we continue by popping another element from primary and pushing it to secondary stack. If, however, the top of secondary stack is greater than top of primary stack, then we will swap the values.

Once we reach the end of the first pass, the largest element will be in the top of the first stack. So to keep going, we pop all elements from the second stack and push them to the first stack.

9.5 5.2.1 Insertion sort

Insertion Sort is very much like we would sort a hand of cards in a card game. We would fan the cards in our hands and look at them. When we find a card that's out of order, we would pick that card, removing it from the one hand, and insert it back into the right place.

This algorithm is very similar to *Bubble Sort*, however, the comparisons are carried out in a slightly different manner. Instead of comparing all pairwise elements, we will compare element j with all previous elements. This means that we will start sorting with element 2.

If we find that element j is smaller that previous elements, but is larger than element k (with $k < j$) then we will move element j to position $k + 1$ and *shift* all other values to the right.

The Shift function at algorithm listing 14 shows it.

Algorithm 14 Shift Function

```
function SHIFT($array$, $i$, $j$)
    if $i \leq j$ then
        return $array$
    end if
    $store \leftarrow array[i]$
    for $0 \leq k \leq (i - j - 1)$ do
        $array[i - k] \leftarrow array[i - k - 1]$
    end for
    $array[j] \leftarrow store$
    return $array$
end function
```

Listing 15 contains the pseudocode for *Insertion Sort.*

Algorithm 15 Insertion Sort

```
function InsertionSort($vector$)
    for $2 \leq i \leq LENGTH[vector]$ do
        $j \leftarrow i$
        while $(vector[i] < vector[j-1]) \wedge (j > 1)$ do
            $j \leftarrow j - 1$
        end while
        $Swap(vector, i, j)$
    end for
    return $vector$
end function
```