Algorithms And Data Structures I

Course Notes

Felipe Balbi

May 25, 2020

Contents

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 0.0.1 (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:

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

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

Take mean to get new candidate:

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

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.

Figure 1 shows an 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.

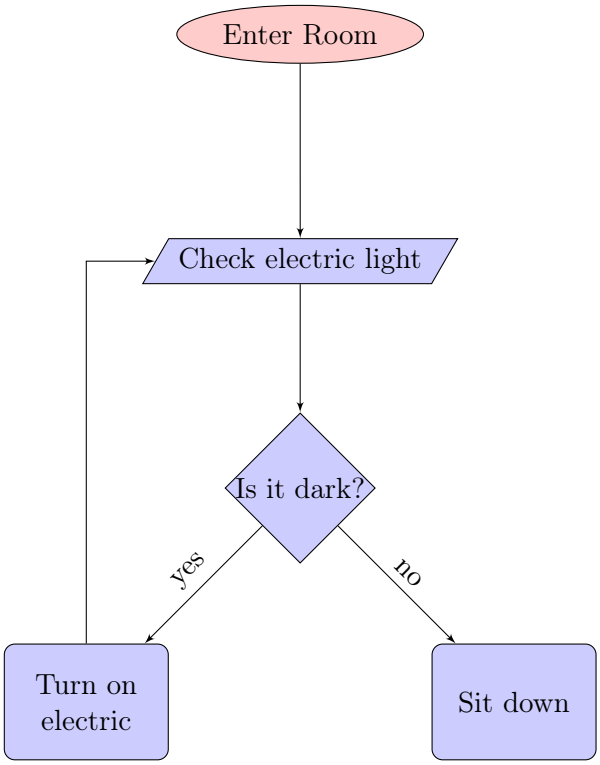


Figure 1: Example Flowchart

Heron's Method Flowchart

Figure 2 shows Heron's Method Flowchart

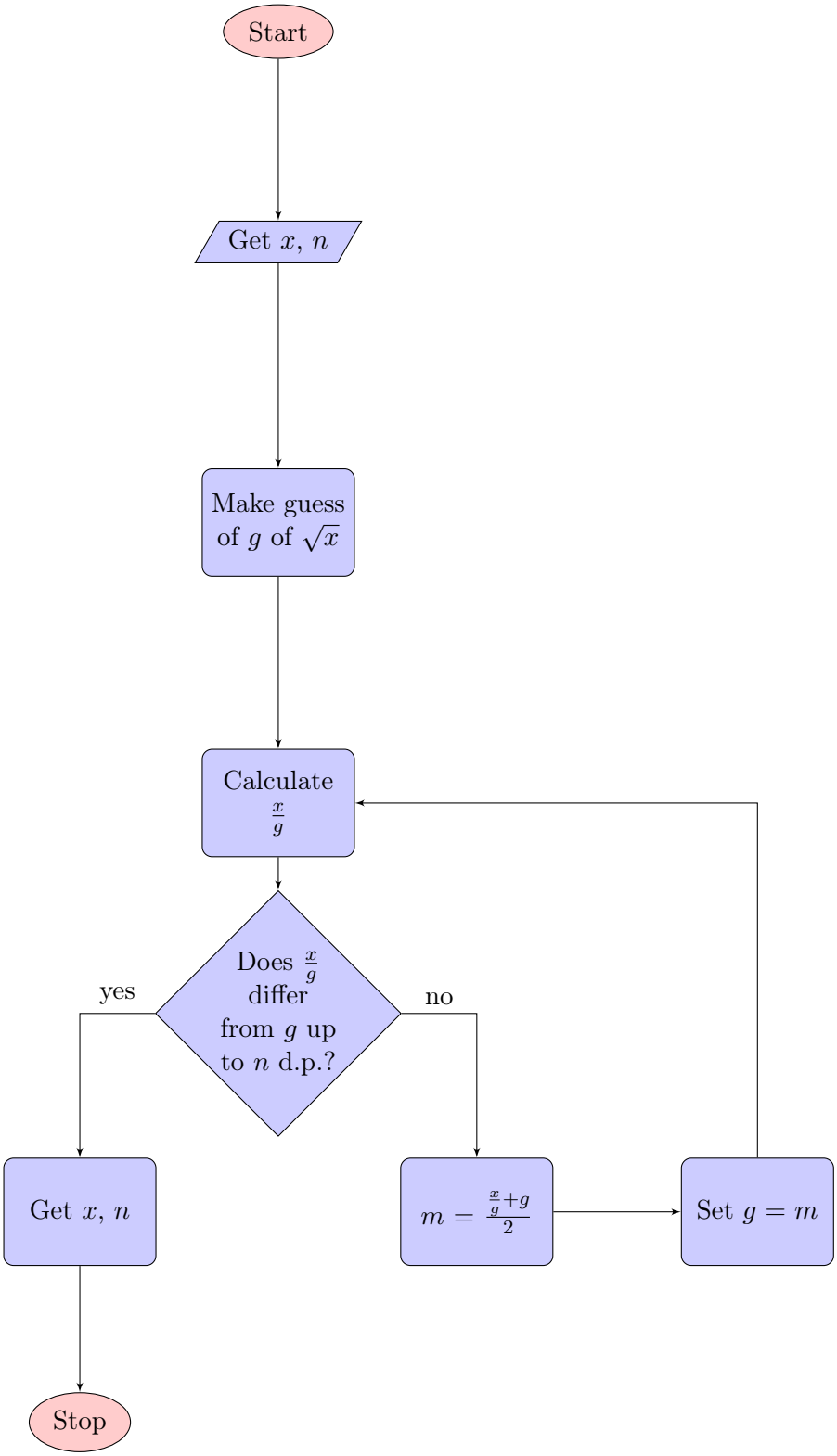


Figure 2: Heron's Method

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 the one shown on figure 3.

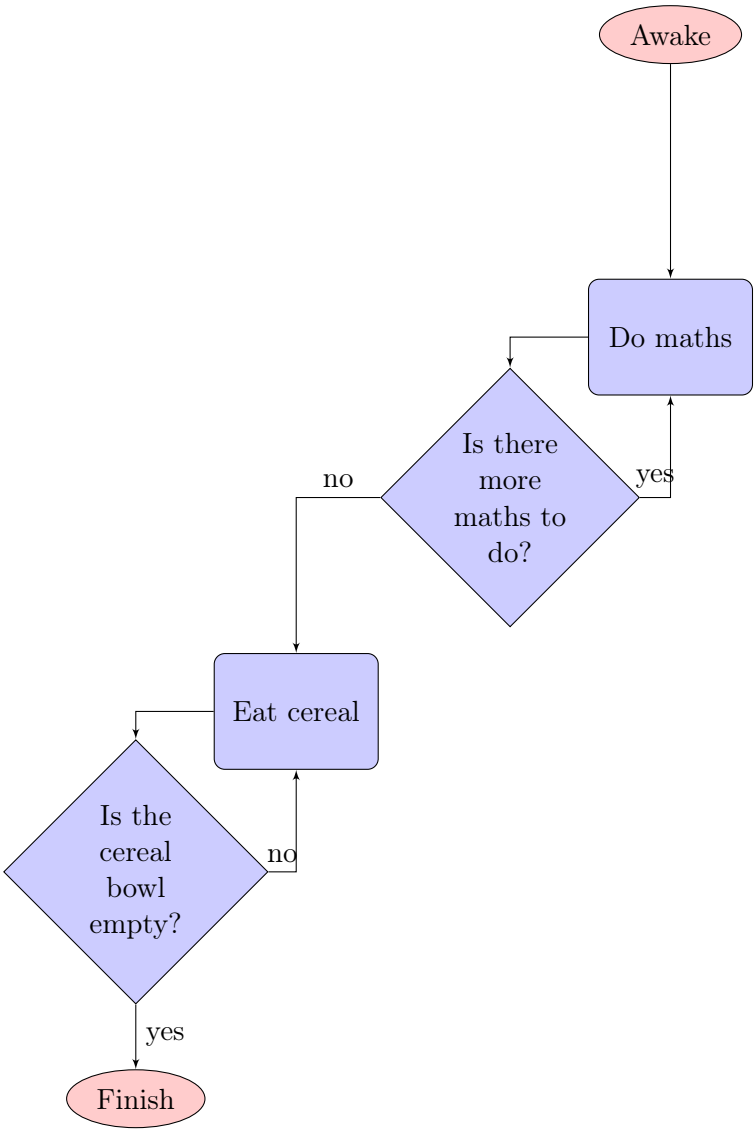


Figure 3: Prof. Matty's Morning Routine

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, the flowchart is show in figure 4.

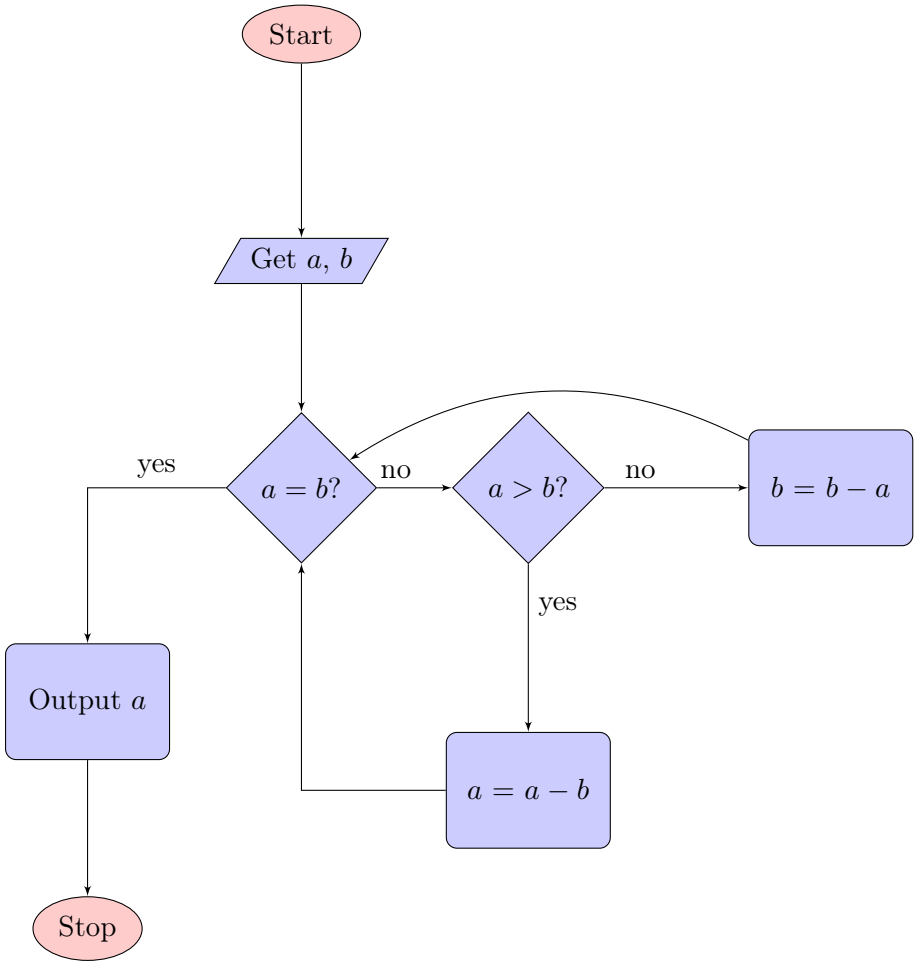


Figure 4: Prof. Matty's Morning Routine

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: $y \leftarrow$ **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: $x \leftarrow x + 3$

1. 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
```

2. 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

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.

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 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
```

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
```

Algorithm 10 shows an example of both.

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
```

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 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 figure 5 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**

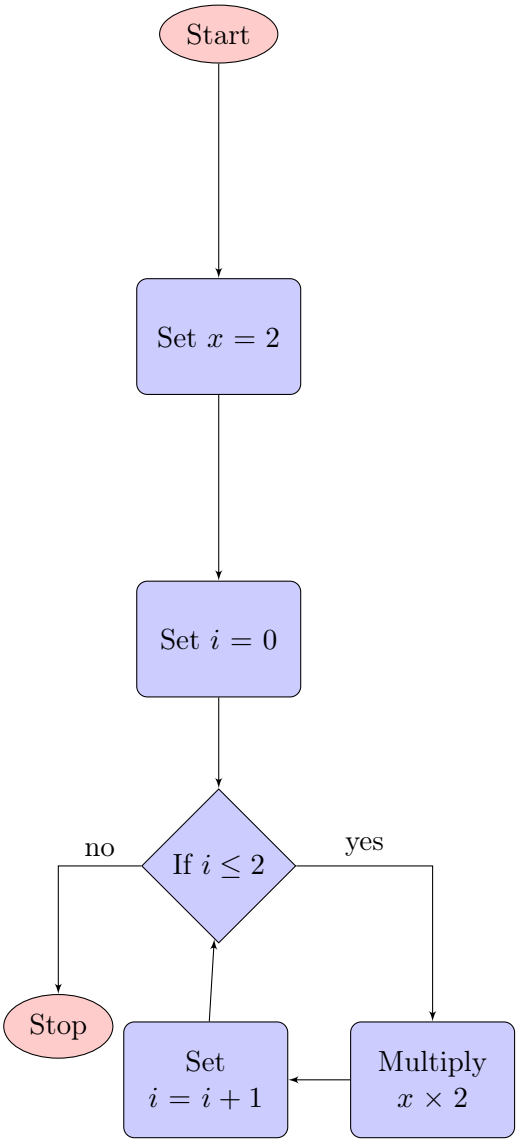


Figure 5: Pseudocode Loop Flowchart

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

2.3.4 My example in pseudocode

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

Algorithm 12 Heron's Method Pseudocode

```
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
```

```
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.

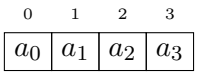
In computer science we care about how data is stored and processed on a **computer**.

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.

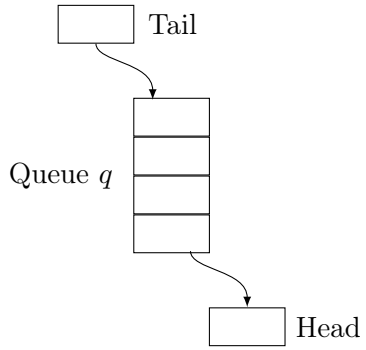
While vectors may seem a bit too simplistic at first, they are a building block for more complex data structures and provide just enough power for us to describe more complex structures.

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

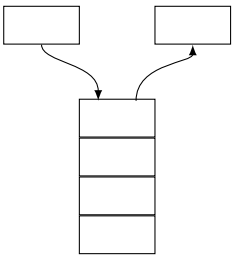
Week 6

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.

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

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.

Definition of an Array

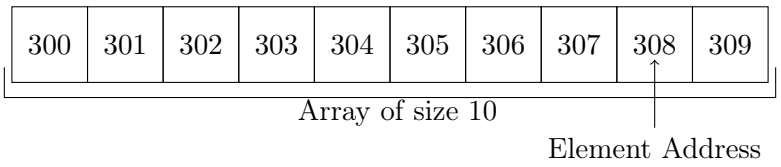
An array is essentially a block of memory on which we can store a collection of data. Typically, the elements of an Array are all of the same type, whether they are integers,

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 [1]. In this way, we can view the Array as a line of integers.

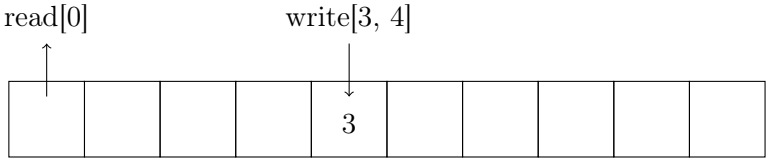
Much like a book, which have consecutively numbered pages of content, the addresses of the array are like the page numbers and the contents of the array are like the contents of a given page.

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



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*.

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

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

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

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*.

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.

```
1: function LinearSearch(v, item)
2:     for 1 ≤ i ≤ LENGTH[v] do
3:         if v[i] = item then
4:             return i
5:         end if
6:     end for
7:     return FALSE
8: end function
```

4.2.3 Searching π

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

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.

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

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 6 we have a visual representation of a pointer.

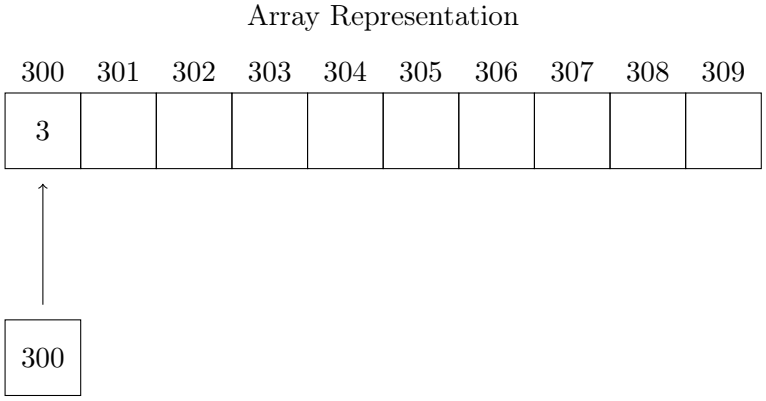


Figure 6: 10-element Array and a Pointer

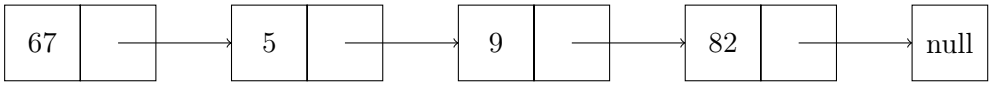


Figure 7: Singly Linked List

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, 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 7.

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.

Week 9

Learning Objectives

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

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.

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.

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 8 shows a vector randomly sorted, i.e. it is unsorted. After looking at figure 8 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 13.

Algorithm 13 Swap Function

```
1: function SWAP(vector, i, j)
2:     x ← vector[j]
3:     vector[j] ← vector[i]
4:     vector[i] ← x
5:     return vector
6: end function
```

Given algorithm listing 13, 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 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 14 shows an implementation of *Bubble Sort* in pseudocode.

Algorithm 14 Bubble Sort

```
1: function BUBBLE SORT(vector)
2:     n ← LENGTH[vector]
3:     for 1 ≤ i ≤ n − 1 do
4:         count ← 0
5:         for 1 ≤ j ≤ n − 1 do
6:             if vector[j + 1] < vector[j] then
7:                 Swap(vector, j, j + 1)
8:                 count ← count + 1
9:             end if
10:        end for
11:        if count = 0 then
12:            break
13:        end if
14:    end for
15:    return vector
16: 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

9	8	7	6	5	4	3	2	1	0

Figure 8: Shuffled vector

(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.

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.

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 15 shows it.

Algorithm 15 Shift Function

```
1: function SHIFT(array, i, j)
2:     if i ≤ j then
3:         return array
4:     end if
5:     store ← array[i]
6:     for 0 ≤ k ≤ (i − j − 1) do
7:         array[i − k] ← array[i − k − 1]
8:     end for
9:     array[j] ← store
10:    return array
11: end function
```

Listing 16 contains the pseudocode for *Insertion Sort*.

Algorithm 16 Insertion Sort

```
1: function INSERTIONSORT(vector)
2:     for 2 ≤ i ≤ LENGTH[vector] do
3:         j ← i
4:         while (vector[i] < vector[j − 1]) ∧ (j > 1) do
5:             j ← j − 1
6:         end while
7:         Shift(vector, i, j)
8:     end for
9:     return vector
10: end function
```

Week 10

Learning Objectives

- Explain the bubble sort in terms of comparisons
- Understand insertion sort and how it differs from bubble sort
- Relate and translate iteration in pseudocode to iteration in JavaScript

5.3.1 Loops in JavaScript

After drinking so much of the Theoretical Cool-Aid, it's time to enjoy some programming with JavaScript.

We can download Node.js from here and install it on our computer. One installed, we can use it to run javascript code outside of the browser environment.

In order to write some javascript, we're going to need a text editor. Professor Matty shows two options: Atom and Sublime Text. There are other very good options, such as Visual Studio Code, Brackets, WebStorm and many others. Take your pick.

After installing an editor, we can start writing some JavaScript code, starting by converting the morning routing pseudocode into actual JavaScript

```
/* create variable done and assign it the value 5 */
var done = 5;
var maths = 0;

/* a for loop that runs from 1 to 5 */
for (var i = 1; i <= done; i++) {
  maths++;
  /* maths += 1
   * maths = maths + 1
   */
}

var cereal = 10;

while (cereal > 0) {
```

```
  cereal--;
}
```

5.3.3 Bubble sort in JavaScript

In order to implement Bubble Sort, first we need to implement the *Swap* function.

```
function swap(array, i, j)
{
  var x = array[j];
  array[j] = array[i];
  array[i] = x;
}
```

Given *Swap*, we can implement Bubble Sort:

```
function bubbleSort(array)
{
  var n = array.length;

  for (var i = 0; i <= n - 2; i++) {
    var count = 0;

    for (var j = 0; j <= n - 2; j++) {
      if (array[j + 1] < array[j]) {
        swap(array, j, j + 1);
        count++;
      }
    }

    if (count == 0)
      break;
  }

  return array;
}
```

In order to test this, we need to define an unsorted array and try to sort it:

```
arr = [10, 9, 8, 7, 6, 5, 4, 4, 2, 1];
console.log(bubbleSort(arr));
```

This returns the sorted array `[1, 2, 4, 4, 5, 6, 7, 8, 9, 10]`.

Week 11

Learning Objectives

- Explain the model of random-access machines
- Explain asymptotic growth of functions and worst-case time complexity

6.0.3 Introduction to Topic 6

Assuming we can solve a problem with a multitude of solutions, how do we decide which solution is the best for the particular situation?

During this topic, we discuss the Random-Access Machine model of computation in order to start comparing algorithms.

The Random-Access Machine models the behavior a generic Central Processing Unit (CPU). This generic, or abstract CPU model is useful in studying and predicting why certain tasks take longer than other to execute.

6.1.1 Random-access machine model

Modern Computers follow the same basic architecture: the Von Neumann Architecture.

The Von Neumann Architecture, named after John von Neumann, a Hungarian-American Mathematician, looks a bit like the block diagram shown in figure 9.

Note that Random-Access Memory, or *RAM*, is the memory that's accessible to the CPU, which makes it mutable as a result. Values stored in RAM change as a result of computations.

External Memory is a memory type that is not directly accessible by a CPU. When we need to perform computations on data that's sitting on external memory, the data must first be loaded onto RAM.

At a basic level, the data available in *RAM* is represented as binary digits, or *bits*. Modern computers, usually don't deal with with bits directly; rather they compute in terms of bytes, which is a collection of 8 bits.

Data and program instructions are stored in registers and the CPU can access each register if given the correct address.

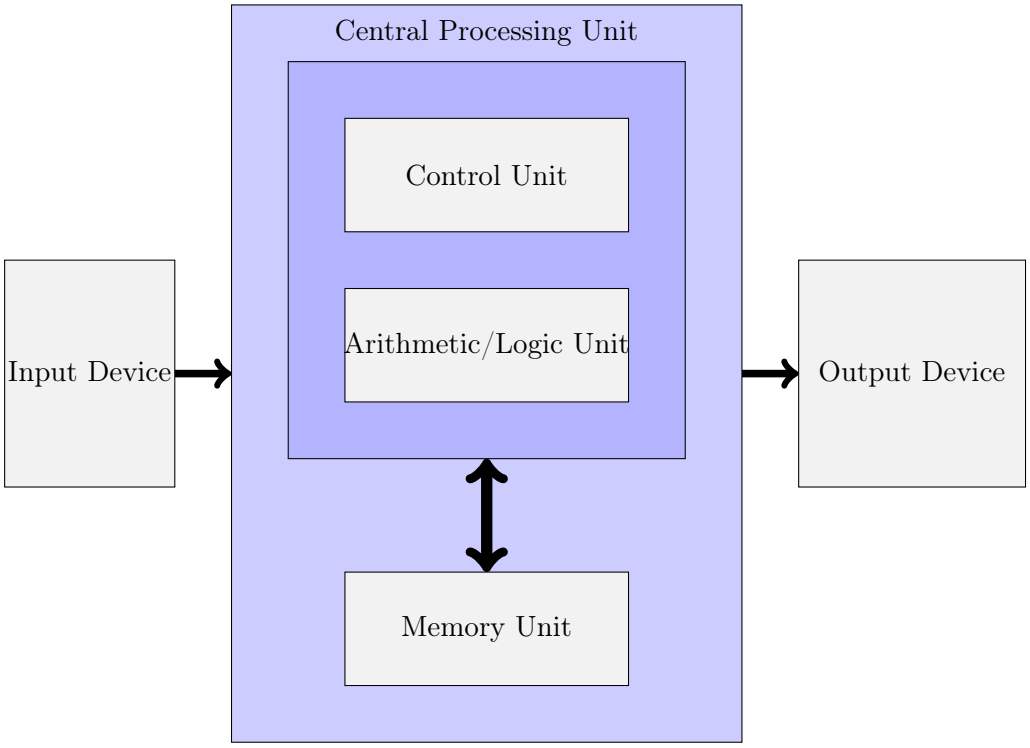


Figure 9: Von Neumann Architecture

"Random-access" refers to how the machine can access data in any register in one single operation. This means that if we know the address of a register, the CPU can access it directly.

The RAM Model can be visualized in figure 10. The Central Processing Unit is composed of a Program Counter (PC), a Control Unit (CU), and an array of registers. The Memory is composed of an array for Input values, an array for Output values and an array for the Program Instructions.

The Control Unit can read data from the Input and store data to the Output. It can also read Instructions from the Program. It can also edit the Program Counter to keep track of where it is in the program execution.

Each of the registers can store an arbitrary integer value. For the Program Counter, which is nothing but a specialized register, the integer stored in it must be positive because it keeps track of where we are in the program.

Depending on the value stored in all of these registers, the Control Unit will perform a particular operation.

The Control Unit can **read**, **write**, and **copy** values from the memory units. It can also perform simple arithmetic operations such as addition, subtraction, multiplication, and division.

In addition to the operations mentioned before, the Control Unit can also perform conditional operations (if... then...).

6.1.3 Counting operations

The RAM Model will be used to abstract away all the particular details of computers and focus on the common structure. By doing this, we can clearly account for all the things done in an implementation of an algorithm.

We can also use our RAM Model to implement abstract data structures. For example and array could be described in terms of a consecutive set of registers where each register address would be the index of a particular element of the array. A linked list can be implemented by using data in registers or memory units to store locations of where the machine looks next.

Another reason to introduce the RAM Model was to illustrate how can account for operations in a model of computing. When comparing different algorithms, we can count the number of operations necessary to solve the problem. We assume that operations are carried out sequentially and each operation takes one time-step.

In the following example, we count the operations carried out for computing the factorial of a number based on code listing 17.

We can convert the pseudocode into a set of instructions that will look like so:

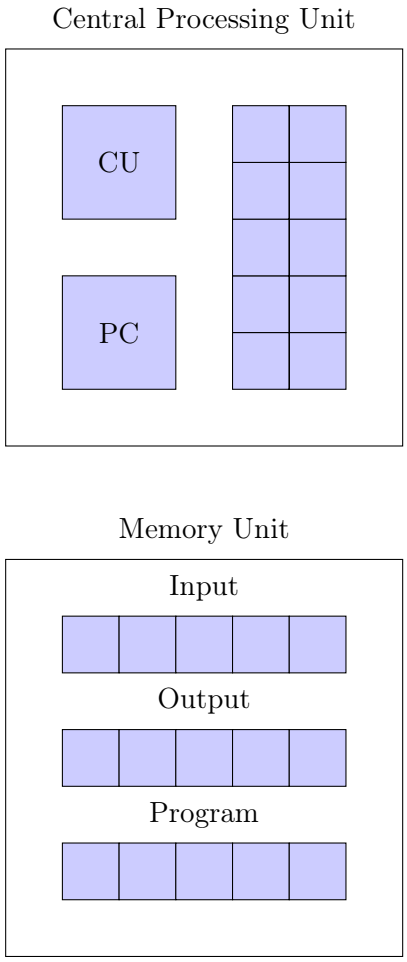


Figure 10: Random-access Machine Model

Algorithm 17 Factorial function

```
1: function FACTORIAL(n)
2:     a ← 1
3:     for 1 ≤ i ≤ n do
4:         a ← a × i
5:     end for
6:     return a
7: end function
```

1. Retrieve n
2. Store n in register
3. Store a in register
4. Store i in register
5. Check if i is less then or equal to n. Go to step **6** if yes, otherwise step **9**
6. Multiply i and a and store result
7. Increase i by 1
8. Go to step **5**
9. Store a in output and stop

Our program counter will change from 1 to 9 depending on which instruction we're executing. Steps **5** through **8** will repeat n times so we can multiply from 1 up to n. This means that with larger n, the program will execute for longer time because more time-steps will be needed.

Consequently, time is a resource that gets used up in any implementation and we are interested in how much of this resource is necessary to complete a particular task.

Another important resource we keep track of in Computer Science is the amount of memory (or space) needed for a given task. From the point of view of the RAM Model, we look at the number of registers needed to complete a task and call this the space requirement.

Different algorithms will have different time and space requirements. It's important to point out, however, that space requirements on algorithms always give a **lower bound** on the number of time-steps required to complete the computation. Well, we can operate on a single memory unit in one time-step. If we have n memory units to compute, we will require at least n time-steps.

Considering that the number of time-steps will always be larger than the space requirements, studying these time resources says something about the space resources. We will, therefore, focus on time resources.

6.2.1 Comparing function growth

The number of operations from our set of instructions available in section 6.1.3 Counting operations is counted as follows:

Steps 1 - 4 happen once each, so that's 4 operations. Steps 5 - 8 will happen a total of n times each, since we're talking about 4 steps happening n times, we have a $4n$ steps. Step 9 happens once at the end of the program.

Therefore, the total number operations for the factorial program is $4n + 5$.

Counting operations like this can be a rather tedious process, however the important piece of information here was that the time requirement grows with the input n.

In other words, the number of operations is a function of n, $f(n)$. In this case, our function is of the form $f(n) = an + b$. This means that we have a linear function.

Linear functions grow linearly with the input n. The amount of constant growth is given by the constant a.

There are other functions that can model time requirement growth. Polynomial functions ($f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$), exponential functions ($f(n) = a^n$), and logarithmic functions ($f(n) = log_a n$) are the most commonly employed.

Depending on the kind of function, it will tell us the resources required for completing the task as the input grows. This brings to discussing the growth of functions. If the input grows by 1, how much will the function grow?

Function	$f(n+1) - f(n)$
2^n	$2^{n+1} - 2^n = 2^n(2-1) = 2^n$
n^2	$(n+1)^2 - n^2 = n + 1$
n	$(n+1) - n = 1$
$log_2 n$	$log_2(n+1) - log_2(n) = log_2(\frac{n+1}{n}) \leq \frac{1.5}{n}$

When choosing algorithms to solve problems, we should choose the one where the function representing the number of required operations grows the slowest, avoiding exponential growth when we can.

Asymptotic growth of functions studies the behavior of function growth as n gets larger and larger. We can show that a function with an exponential component will always, eventually overtake a polynomial function and, likewise, a polynomial function will always, eventually, overtake a logarithmic function.

The study of asymptotic growth motivates the Big O notation.

6.2.3 Big O notation

When comparing the asymptotic growth of functions, we need only concern ourselves with the terms that grow the fastest. For example, in the function $f(n) = 2^n + 3n$, 2^n grows much faster than $3n$. Likewise, in the function $f(n) = 1000n^2 + n$, the term $1000n^2$ grows the fastest. Furthermore, the constant 1000 is irrelevant, since what really describes the growth of the function is the n^2 component.

Big O Notation emcompasses both of these details by ignoring constants and focussing only on the fastest growing components of functions.

$f(n)$	$\mathcal{O}(f(n))$
$2^n + 3n$	$\mathcal{O}(2^n)$
$1000n^2 + n$	$\mathcal{O}(n^2)$
35	$\mathcal{O}(1)$

Because Big O groups all functions by their fastest growing parts ignoring constants, we can think of Big O as representing classes or sets of functions.

When we say that $n^2 + 2 = \mathcal{O}(n^2)$ we're saying that $n^2 + 2$ is a member of the set of functions referred to as $\mathcal{O}(n^2)$, not necessarily that it's **equal** to it.

When dealing with logarithmic functions, the base is irrelevant as changing bases is the same as multiplying by a constant since $\log_2 n = \frac{\log_3 n}{\log_3 2}$. Thus, we don't need to worry about the base of the logarithm. Therefore, when applying Big O to logarithms, we simply say $\mathcal{O}(\log n)$.

One thing to note, however, is that $\mathcal{O}(2^n)$ is not the same as $\mathcal{O}(4^n)$ since the latter grows faster than the former.

The rigorous, mathematical definition of Big O is:

$$f(n) \in \mathcal{O}(g(n)) \exists k > 0 \exists n_0 \mid \forall n > n_0 |f(n)| \leq k \cdot g(n)$$

A consequence of this is that each Big O classes live inside one another. Figure 11 shows a hierarchy of Big O classes.

In algorithmic analysis we will **always** try to ascertain the smallest Big O class in which a function lives as this will help us compare and decide which algorithm is best for the application.

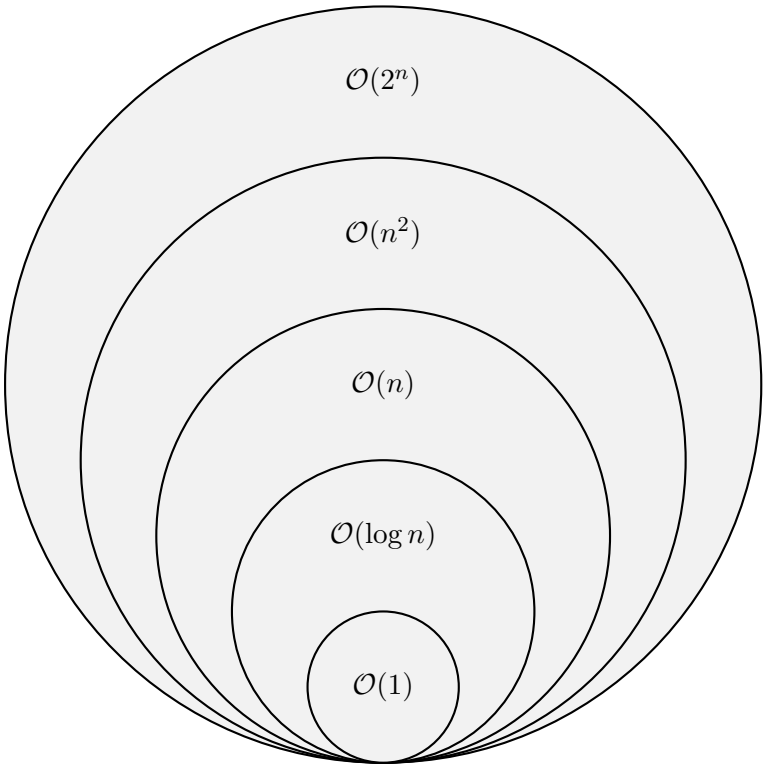


Figure 11: Big O Classes

Week 12

Learning Objectives

- Explain the model of random-access machines
- Explain asymptotic growth of functions and worst-case time complexity
- Describe the worst-case time complexity of the bubble and insertion sort algorithms

6.3.1 Worst-case time complexity

When we say that the number of operations of an algorithms is $\mathcal{O}(n)$, what we really mean to say is that the Time Complexity of performing the algorithm is $\mathcal{O}(n)$.

Time Complexity is a quantification of the time it takes to run an algorithm. In other words, it's the quantification of the number of time-steps required to carry out the computation.

Here, *complexity* refers to the resources required, which means that the *Space Complexity* of an algorithm is the amount of memory space required to complete the computation.

The $\mathcal{O}$ classes represent complexity, however this would be the **smallest** $\mathcal{O}$ in which the number of operations of space requirements live.

We need to understand what happens when we want to e.g. run Bubble Sort on an input vector that's already sorted. If the input is already sorted, no swaps will happen but comparisons are still necessary. Furthermore, if the input is somewhat sorted, some swaps will happen and if it's completely reversed many swaps will happen.

This means that the Time Complexity will vary from input to input in general.

In order to get around this variability in the number of operations, we have another tool: *Worst-case Analysis*.

The worst-case input is the input that results in the most operations needed, or the input with largest time complexity. For example, if we're looking at the linear search algorithm, the best case is that the value we're looking for is in the first element of the array. Conversely, the worst case is when the element we're looking is not in the vector at all.

To find the worst-case time complexity, we establish the smallest $\mathcal{O}$ class in which the number of operations as a function lives.

Looking back at the Linear Search algorithm (reproduced below), we can see that the part of the algorithm that varies with the input is the for loop, which will run for as many iterations as the length of the input array. This means that the worst-case time complexity will be n where n is the length of the array, in other words, the worst-case time complexity is $\mathcal{O}(n)$.

```
1: function LinearSearch(v, item)
2:     for 1 ≤ i ≤ LENGTH[v] do
3:         if v[i] = item then
4:             return i
5:         end if
6:     end for
7:     return FALSE
8: end function
```

In the case of Bubble Sort14, the best case is when the input is already sorted and we need *n-1* comparisons and a single pass of the algorithm. Therefore, the best-case is $\mathcal{O}(n)$. The worst-case will need *n-1* passes each with *n-1* comparisons. Therefore the worst-case is $\mathcal{O}(n^2)$.

6.3.4 Input size

We need to standardize how numbers are stored so we can properly count storage usage. We will assume that the input is a string of bits. A number n can be stored with $\mathcal{O}(\log n)$ bits.

We can let $m = \mathcal{O}(\log n)$. For our factorial function, the time complexity was $\mathcal{O}(n)$. If we want to know the time complexity in terms of the size of the input (m), it will be $\mathcal{O}(2^m)$, since the number n is exponentially larger than the number m.

What this means is that for the amount of space our computer uses to store the input, the number of time-steps (operations) required to compute the factorial on that number grows exponentially in that space.

Week 13

Learning Objectives

- Explain the binary search algorithm and the importance of sorting
- Describe the worst-case time complexity of binary search and compare it with that of linear search

7.0.3 Introduction to Topic 7

The focus of this topic will be the Binary Search Algorithm. This is a very efficient algorithm to search sorted vectors and arrays.

7.1.1 Binary search

Let's assume we have a sorted vector. If we pick any element at random, all other elements to the left will be less than or equal to the one we picked and all objects to the right will be greater than or equal to the one we picked. Figure 12 depicts this information.

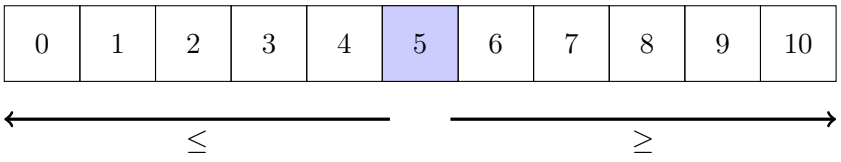


Figure 12: A Sorted Vector

We can rely on this fact to guide how we search to the value we're looking for. If the element we're looking for is larger than the one we picked at random, we don't need to look at any of the values to the left, because they will **all** be smaller than what we're looking for.

In practice, we don't pick an element at random, we pick the midpoint; i.e. the value at the middle of the vector. This way, we split the vector into two smaller vectors and, if necessary, we will **again** apply Binary Search to the smaller vector by choosing the midpoint and comparing against what we're trying to find.

The midpoint of the vector is calculated as:

$$\lfloor \frac{leftmost\_index + rightmost\_index}{2} \rfloor$$

The reason for calculating the midpoint in terms of the indexes of the leftmost and rightmost indexes is because it works when breaking the original vector in subvectors.

7.1.4 Coding up binary search

Code listing 18 shows a pseudocode for Binary Search algorithm.

Algorithm 18 Binary Search

```
1: function BinarySearch(v, x)
2:     n ← LENGTH[v]
3:     L ← 1
4:     R ← n
5:     while R ≥ L do
6:         m ← ⌊(L+R)/2⌋
7:         if v[m] = x then
8:             return TRUE
9:         else if v[m] > x then
10:            R ← m − 1
11:        else
12:            L ← m + 1
13:        end if
14:    end while
15:    return FALSE
16: end function
```

7.2.1 Worst-case complexity of binary search

The best-case running time for Binary Search is when the value we're looking for is at the midpoint of the vector. This means that our very first comparison will find the value. In this case, the total number of operations required is $\mathcal{O}(1)$.

When looking for the worst-case input, we find that there are multiple situations which will result is maximum number of operations required. One such case is when the value we're looking is at the very first element of the vector.

Because we're recursively halving the problem at each iteration, the worst time complexity of this problem is $\mathcal{O}(\log n)$.

We cannot use Binary Search in **unsorted** arrays because we cannot guarantee that when we exclude half the array, we're not excluding the value we're looking for.

7.2.3 Binary search is optimal

Assuming the input data is sorted, we can search very quickly with binary search.

During the video the professor builds an argument as to why Binary Search is an optimal algorithm. The gist of it is that the computation of any searching algorithm can be modeled as a tree of decisions and the height of the tree is optimal path to any one solution. He then shows that the height of the tree is $\log n$ and that's the same complexity of Binary Search, therefore Binary Search is optimal.

Week 14

Learning Objectives

- Explain the binary search algorithm and the importance of sorting
- Describe the worst-case time complexity of binary search and compare it with that of linear search
- Explain how search algorithms can be used in contexts beyond searching a data structure

7.3.1 Search problems and abstraction

In summary, Professor Matty goes about how we can reimagine problems in order to reduce them to problems we already know how to solve.

He provides an example of how we can reduce the original birthday guests problem down to a simpler problem of searching an array.

Week 15

Learning Objectives

- Explain the concept of decrease and conquer
- Apply the tool of recursion to algorithms

8.1.1 Decrease and conquer

Recursion involves self-reference. This self-reference can be used to solve problems by reducing them to smaller instances of themselves.

From an algorithmic perspective, there are two main approaches:

Decrease and Conquer recursively breaking down a problem into a smaller problem of the same type

Divide and Conquer recursively breaking down a problem into two or more parts of the same type

Decrease and Conquer relies on the idea that if we have a problem we reduce it to a smaller problem of the same type and solve that. General inputs can, then, be reduced to the simpler problem for which we have a solution.

Recursion, here, is used to reduce the problem.

Using the factorial as an example:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

Note that this can be written as:

$$n! = n \cdot (n-1)!$$

And $(n-1)!$ can be written as:

$$(n-1)! = (n-1) \cdot (n-2)!$$

And so on. We need a base case, i.e. a simple instance of this problem which is easy to solve. We know that $0! = 1$, so that can be our base case.

This results in pseudocode listing 19 for the factorial function:

Algorithm 19 Recursive Factorial Function

```
1: function FACTORIAL(n)
2:     if n = 0 then
3:         return 1
4:     end if
5:     return n× FACTORIAL(n − 1)
6: end function
```

Note here that if we take away the base case (the *if* statement), this function would recurse forever. Recursively calling the function on the same input would also result in infinite recursion.

Anything that can be solved recursively, can also be solved iteratively. Similarly, anything that can be solved iteratively, can be solved recursively.

8.1.3 Recursive Euclidean algorithm

Pseudocode listing 20 depicts the recursive implementation of Euclides Algorithm:

Algorithm 20 Recursive Greatest Common Divisor

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

8.2.1 Recursive searching and sorting

A recursive implementation is straightforward. Say we have a vector with n elements. If the value we're looking for, is not in the first element, then we have $n - 1$ elements to look.

With that in mind, we can produce the pseucode listing

Bubble Sort can also be implemented recursively. The key to reducing the problem size is realising that after the first pass of the algorithm, the largest value will be pushed to the rightmost element, therefore we don't need to look at it anymore.

```
1: function SEARCH(v, l, item)
2:     n ← LENGTH[v]
3:     if l > n then
4:         return FALSE
5:     else if v[l] = item then
6:         return TRUE
7:     end if
8:     return SEARCH(v, l + 1, item)
9: end function
10: function LINEARSEARCH(v, item)
11:     return SEARCH(v, 1, item)
12: end function
```

This results in the pseudocode listing 21.

Algorithm 21 Recursive bubble sort

```
1: function SORT(vector, r)
2:     if r ≤ 1 then
3:         return vector
4:     end if
5:     for 1 ≤ j ≤ r − 1 do
6:         if vector[j + 1] < vector[j] then
7:             SWAP(vector, j, j + 1)
8:         end if
9:     end for
10:    SORT(vector, r − 1)
11:    return vector
12: end function
13: function BUBBLE SORT(vector)
14:    n ← LENGTH[vector]
15:    return SORT(vector, n)
16: end function
```

Insertion Sort has a different key to reducing the problem. The base case here is if we have a vector of a single element which, clearly, is already sorted. Pseudocode listing 22 depicts the recursive version of Insertion Sort.

8.2.3 Permutations revisited

We can generate permutations recursively by realising that in order to generate the permutations for n letters, we need the permutations for $n - 1$ letters. Pseucode listing 23 depicts a possible solution.

Algorithm 22 Recursive insertion sort

```
1: function SORT(vector, r)
2:     if r ≤ 1 then
3:         return vector
4:     end if
5:     SORT(vector, r − 1)
6:     j ← r
7:     i ← r
8:     while (vector[i] < vector[j − 1]) ∧ (j > 1) do
9:         j ← j − 1
10:    end while
11:    Shift(vector, i, j)
12:    return vector
13: end function
14: function INSERTIONSORT(vector)
15:    n ← LENGTH[vector]
16:    return SORT(vector, n)
17: end function
```

Algorithm 23 Recursive permutations

```
1: function PERMUTATIONS(vector)
2:     if LENGTH[vector] ≤ 1 then
3:         return vector as a dynamic array
4:     end if
5:     new DynamicArray s
6:     for 1 ≤ i ≤ LENGTH[vector] do
7:         new Vector v(LENGTH[vector] − 1)
8:         v[1 : i − 1] ← vector[1 : i − 1]
9:         v[i : LENGTH[vector] − 1] ← vector[i + 1 : LENGTH[vector]]
10:        new DynamicArray w ← PERMUTATIONS(v)
11:        new Vector p(LENGTH[vector])
12:        p[1] ← vector[i]
13:        for 1 ≤ j ≤ LENGTH[w] do
14:            p[2 : LENGTH[vector]] ← w[j]
15:            s[LENGTH[s] + 1] ← p
16:        end for
17:    end for
18:    return s
19: end function
```

Week 16

Learning Objectives

- Explain the general concept of decrease and conquer
- Apply the tool of recursion to algorithms
- Explain how recursion is implemented using the call stack

8.3.1 Recursive binary search

Pseudocode listing 24 shows a Recursive version of Binary Search. Much like all the other recursive search algorithms, the implementation is divided up into two functions.

Algorithm 24 Recursive Binary Search

```
1: function SEARCH(v, item, L, R)
2:     if L > R then
3:         return FALSE
4:     end if
5:     m ← ⌊(L+R)/2⌋
6:     if v[m] = item then
7:         return TRUE
8:     else if v[m] > item then
9:         R ← m − 1
10:    else
11:        L ← m + 1
12:    end if
13:    return SEARCH(v, item, L, R)
14: end function
15: function BINARYSEARCH(v, item)
16:    R ← LENGTH[v]
17:    L ← 1
18:    return SEARCH(v, item, L, R)
19: end function
```

8.3.3 Call stack

How is recursion handled during execution? How can a function call itself and still keep track of the correct parameters and returned values?

There's something called the Call Stack, which is a data structure that stores information about called functions within a program.

The call stack:

- works in the background of the program while it's running
- manages what functions are called and when
- manages all the data used by a function

The details of the Call Stack itself will depend on the choice of programming languages, CPU architecture, machine language and several other factors.

Every time a function is called, all the variables and arguments relevant to that function are pushed onto the stack and, together, form a Stack Frame.

The point here is that even when a function calls **itself**, we still get a new stack frame pushed onto the stack. Because of that we can solve problems recursively.

Week 17

Learning Objectives

- Explain the concept of divide and conquer
- Explain and implement merge sort and quicksort

9.0.3 Introduction to Topic 9

We're dealing with two new sorting algorithms: Quicksort and Merge sort.

Both of these algorithms are Divide-and-conquer Algorithms. Divide and conquer is a form of multi-branched recursion, in which problems are divided (rather than reduced) into smaller problems and each sub-problem is solved recursively.

Quicksort works by partitioning our input according to the value of a specific element. Merge sort works by halving the input multiple times and running merge sort on these halves.

9.1.1 Quicksort

Every divide and conquer algorithm works by dividing the problem into 2 or more smaller problems and solving these smaller problems individually. Later, we combine the solved sub-problems to get to the solution of the full problem.

Recursion is a simple way of implementing this methodology.

As with other sorting algorithms, we assume we're given an unsorted array of numbers and want to return a sorted array of numbers in ascending order.

Given the input, we break it into one or two smaller vectors and call Quicksort individually on these smaller vectors using recursion.

To partition the array, we compare its elements against a special element called the *pivot*. How the *pivot* is chosen depends on a particular implementation of Quicksort.

Prof. Matty's implementation chooses the *pivot* at the midpoint of the vector, i.e.:

$$pivot = \lfloor \frac{left+right}{2} \rfloor$$

where:

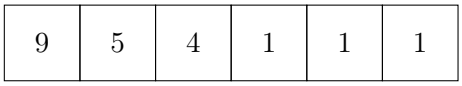


Figure 13: Unsorted array

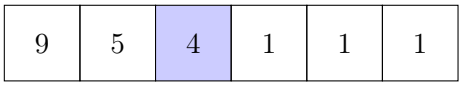


Figure 14: Pivot

left leftmost index

right rightmost index

Figure 13 shows an unsorted array.

Given an unsorted array, we choose a *pivot* in figure 14.

One the *pivot* has been chosen, we partition the vector into two smaller vectors such that elements smaller than the *pivot* are on the left sub-vector and elements larger than the *pivot* are on the right sub-vector, see figure 15.

With this completed, we run Quicksort on the sub-vectors. The base case here is a vector of one element which is already sorted. This means we need to choose new *pivots* (figure 16).

Following, we apply Quicksort again and split into smaller vectors, as in figure 17.

Which leads us to a new set of pivots, as in figure 18.

Now we have reached the base case of a vector of one element, we just return it and we have a sorted vector.

All of this can be implemented without creating extra vectors, by relying on *Swap* operations. What we do is that we iterate over the vector from both sides, approaching the *pivot* from both sides.

We make a variable i starting at the first index and j starting at the last index. Then we compare both against the *pivot*. Depending on the result of the comparison, we *Swap* i^{th} and j^{th} elements.

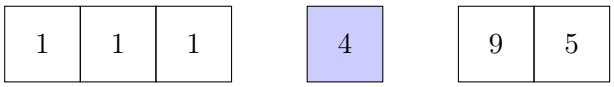


Figure 15: Sub-vectors

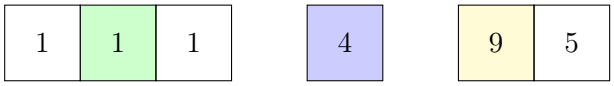


Figure 16: New pivots

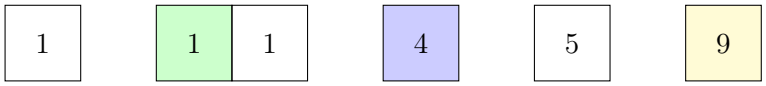


Figure 17: Split into smaller vectors

9.1.3 Quicksort and induction

We know how quicksort works (by splitting the problem into multiple small problems until we reach a base case, which is easy to solve). Now we need to establish the correctness of the algorithm, i.e. we must understand why it works.

In order to prove the correctness of the recursive algorithm, we can connect it to the concept of Mathematical Induction. More specifically, we're going to rely on strong induction.

A mathematical strong induction proof has two requirements:

Base case proof that the base case $P(1)$ is true

Inductive step Assuming $P(1), \ldots, P(k)$ to be true, produce a proof that $P(k+1)$ is also true

The base case for Quicksort is simple: it's a vector with a single element (or without any elements) which is, clearly, already sorted.

For the inductive step, we can build an argument that Quicksort splits the vector in three parts: *pivot*, left vector and right vector. The *pivot* can be considered as a vector of one element, which is already sorted as per the base case.

As for the left and right vector, the largest possible vector is of length k, which is assumed to be sortable. In other words, a vector of $n+1$ elements, can be split into a vector of $n-1$ elements, a *pivot* or 1 element, and another vector of 1 element. All three are known to be sortable by Quicksort.

Figure 18: More pivots

9.2.1 Merge sort

Merge Sort works by splitting a vector into 1 element vectors then merging these vectors so that while merging they are kept in sorted order.

To illustrate, the following figures walks through the merging of two vectors of 3 elements each.

Figure 19: Initial state

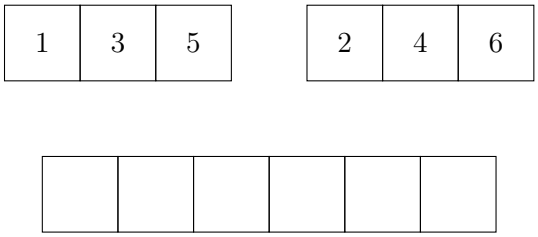


Figure 20: Create empty vector

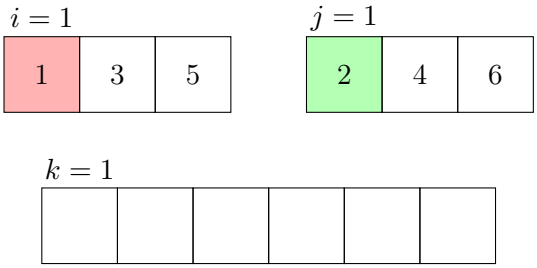


Figure 21: Compare elements

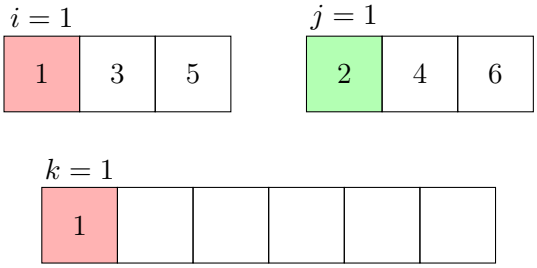


Figure 22: Move smallest to new vector

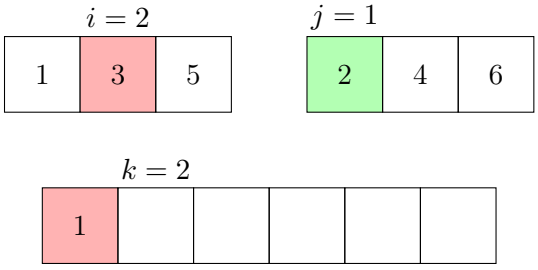


Figure 23: Increment i and k

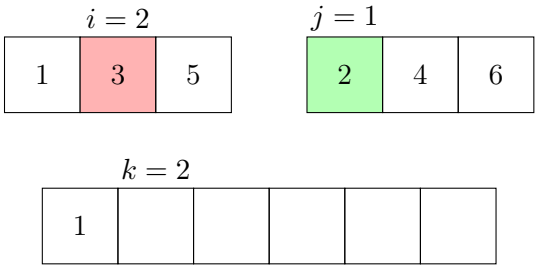


Figure 24: Compare elements

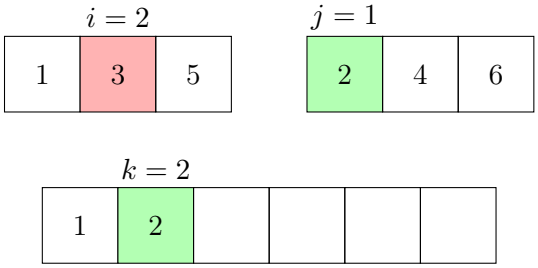


Figure 25: Move smallest to new vector

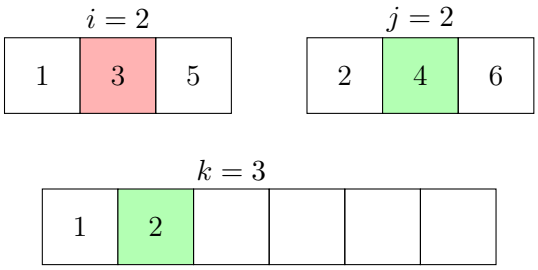


Figure 26: Increment j and k

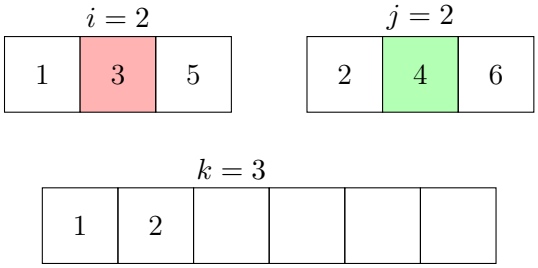


Figure 27: Compare elements

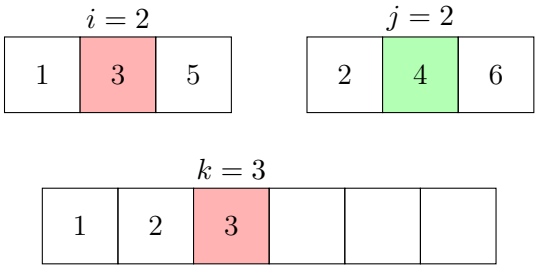


Figure 28: Move smallest to new vector

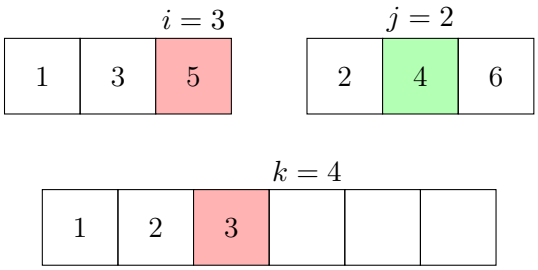


Figure 29: Increment i and k

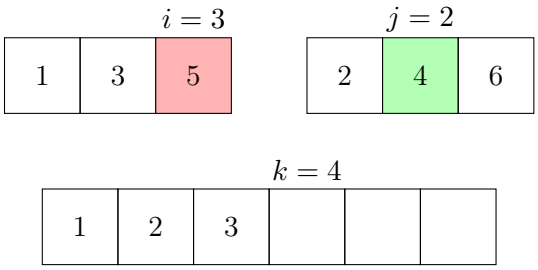


Figure 30: Compare elements

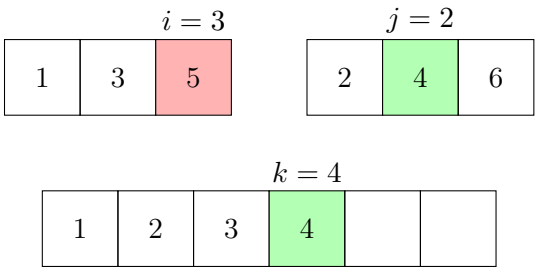


Figure 31: Move smallest to new vector

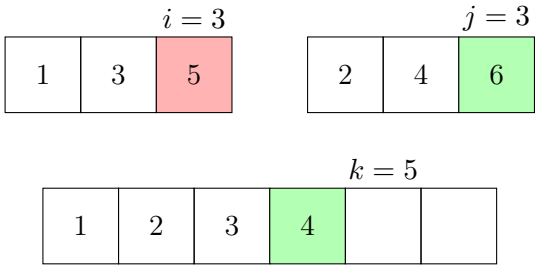


Figure 32: Increment j and k

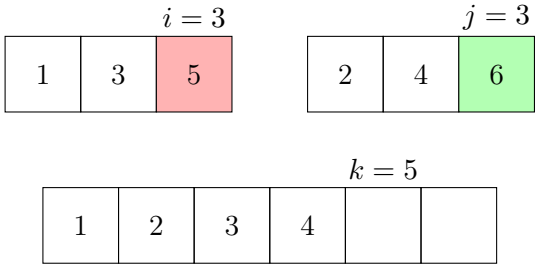


Figure 33: Compare elements

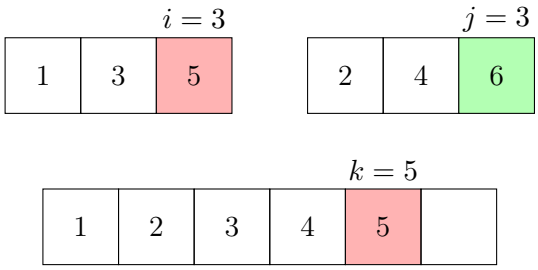


Figure 34: Move smallest to new vector

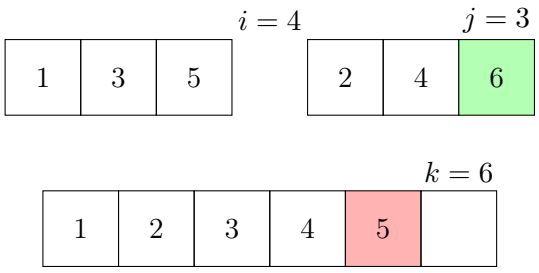


Figure 35: Increment i and k

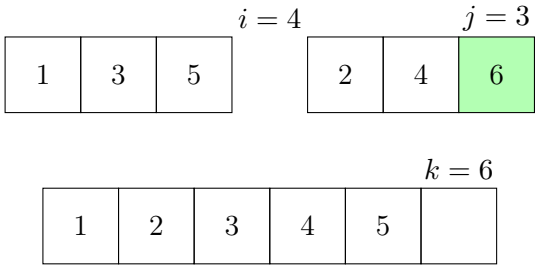


Figure 36: Only one element left

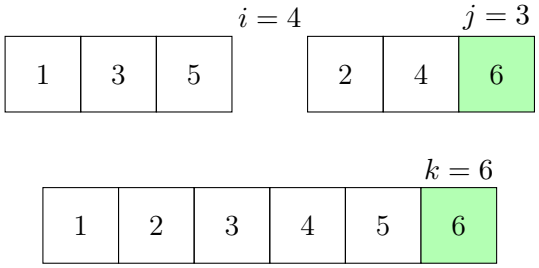


Figure 37: Move it to the new vector

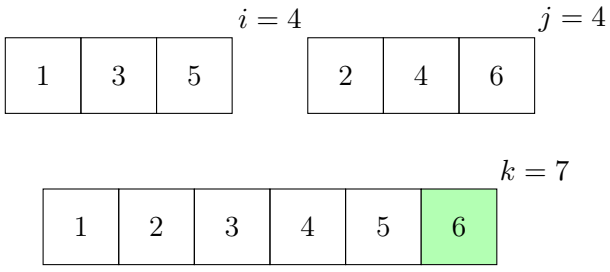


Figure 38: Increment j and k

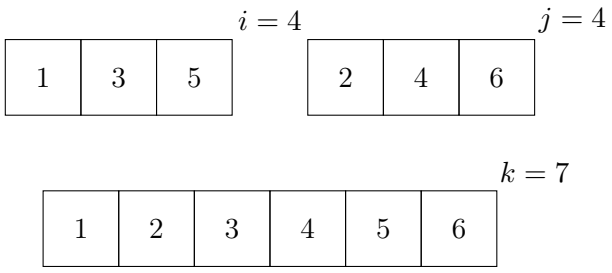


Figure 39: Algorithm terminated

Merge sort pseudocode

Algorithm 25 Merge sort

```
1: function MERGE(w, v)
2:     m ← LENGTH[w]
3:     n ← LENGTH[v]
4:     new Vectors(m + n)
5:     i ← 1
6:     j ← 1
7:     k ← 1
8:     while (i ≤ m) ∧ (j ≤ n) do
9:         if w[i] < v[j] then
10:            s[k] ← w[i]
11:            i ← i + 1
12:        else
13:            s[k] ← v[j]
14:            j ← j + 1
15:        end if
16:        k ← k + 1
17:    end while
18:    while i ≤ m do
19:        s[k] ← w[i]
20:        i ← i + 1
21:        k ← k + 1
22:    end while
23:    while j ≤ n do
24:        s[k] ← v[j]
25:        j ← j + 1
26:        k ← k + 1
27:    end while
28:    return s
29: end function
30: function MERGESORT(vector)
31:    n ← LENGTH[vector]
32:    if n = 1 then
33:        return vector
34:    end if
35:    m ← ⌊(n+1)/2⌋
36:    new VectorL(m)
37:    new VectorR(n − m)
38:    L ← vector[1 : m]
39:    R ← vector[m + 1 : n]
40:    return MERGE(MERGESORT(L), MERGESORT(R))
41: end function
```

Week 18

Learning Objectives

- Explain the concept of divide and conquer
- Explain and implement merge sort and quicksort
- Compare the worst-case time complexity of certain sorting algorithms

Essential Reading

- Cormen, T.H., C.E. Leierson, R.L. Rivest and C. Stein Introduction to algorithms. (Cambridge, MA: MIT Press, 2009) 3rd edition.
  - Section 7
  - PDF

9.3.1 Worst-case time complexity of Quicksort and merge sort

After discussing merge sort and quicksort, we need to analyze their worst-case time complexity. Looking back at section 6.3.1 Worst-case time complexity, Bubble sort and insertion sort had worst-case time complexity of $\mathcal{O}(n^2)$ for vectors of length n.

Quicksort, again, has a worst-case time complexity of $\mathcal{O}(n^2)$. We can show that this happens when the largest element is the *pivot* as this will result in the vector being broken into one vector of length $n-1$ instead of vectors of length $\approx \frac{n}{2}$.

If, for some reason, every time we look for another *pivot*, it happens to be the next largest number. This is shown in figure 40.

What the figure shows is that each time through the algorithm, we will execute $n-1$ comparisons on a vector of length $n-1$. This will repeat for each iteration which leads us to conclude that the worst-case complexity belongs to the class of $\mathcal{O}(n^2)$.

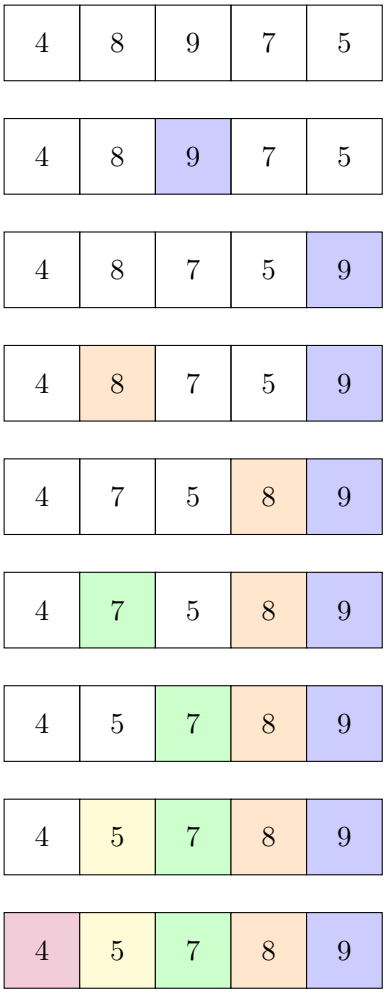


Figure 40: Quicksort worst-case

If this algorithm performs the same as bubblesort and insertion sort, why bother? The reason is that the likelyhood of facing a worst case input is very minimal. A typical input will, generally speaking, allow us to halve the input at each run through the algorithm and this will take $\mathcal{O}(\log n)$ time. Each time we halve the vector, we need to compare the elements' values with the value of the pivot, and this take $\mathcal{O}(n)$ time. Therefore, the typical time complexity of Quicksort is proportional to $\mathcal{O}(n \log n)$.

A follow-up question might be *What is typical, anyway?*. It means that if we have a set of **all** possible vectors and pick one at random, there's a high probability that it will possess a particular property or a set of properties. Conversely, there's a low probability that we will get the worst-case input.

We refer to this as the Average-case and its time complexity is that of a typical input.

Quicksort average-case time complexity is $\mathcal{O}(n \log n)$ and due to this, it's widely used in libraries of programming languages.

It's, for example, part of C's standard library (at least those conforming to POSIX.1-2001)[1] and some browsers' javascript engines (like Chrome, the new Microsoft Edge, Brave, etc). Firefox's javascript engine, however, uses merge sort.

Merge sort has the same time complexity for best, average and worst-case: $\mathcal{O}(n \log n)$[2].

We can compare quicksort and merge sort by their space requirements. Merge sort will need at least $\mathcal{O}(n)$ extra space to carry out its computations. Quicksort can run all its operations in place, without any extra memory being used, except for the extra space in the stack for the recursive calls. In the worst-case, we will have $\mathcal{O}(n)$ recursive calls so the number of elements on the stack will be $\mathcal{O}(n)$.

The worst-case space complexity of bubble sort is $\mathcal{O}(1)$, this is because at each run of the algorithm there is a constant amount of variables needed (number of passes and indices of both elements being compared). Likewise, insertion sort has worst-case space complexity of $\mathcal{O}(1)$, since all we need to keep track of are the indices of the elements being compared and values being shifted.

The table below summarizes all of this information:

Algorithm	Worst-case Time Complexity	Worst-case Space Complexity
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Quicksort	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

[1] Binary search is also part of C's standard library
[2] Comparison of Sorting Algorithms on Wikipedia

Week 19

Learning Objectives

- Explain what a decision problem is
- Explain the basic concept of a complexity class

Essential Reading

- Revision Exercises

10.0.3 Introduction to Topic 10

Can we conceive quick and efficient algorithms to solve certain problems? We begin to consider what such questinos by grouping problems in groups called Complexity Classes. The study of relationships between these groups is called Computational Complexity.

The subject of Computational Complexity operates at a much more abstract and theoretical level, but there is much that can be said about with regards to why it is difficult to find efficient solutions to certain problems.

It has been known for decades that there exist problems which we don't know how to solve. One such problem, as demonstrated by Alan Turing, is the Halting Problem. The problem statement goes like so: Given a mathematical description of a computer and a program, does the program terminate in finite time? Alan Turing claimed that it's not possible to prove that the program terminates in a finite amount of time-steps.

10.1.1 Introduction to complexity classes

While discussing time-complexity, we favoured one algorithm over another by choosing the one whose time-complexity grew slower over time. For example, we prefer an algorithm that grows at $\mathcal{O}(n^2)$ over an algorithm that grows at $\mathcal{O}(2^n)$. This is because an exponential function grows faster than any polynomial function.

This notion, allows us to make a distinction between efficient and innefficient algorithms. Therefore, we want to classify **problems** with regards to whether they can be solved by an efficient algorithm or not.

A problem is, therefore, hard if there is no efficient algorithm to solve it.

We need a standard method of how to think about time complexity. We have discussed before that the standard way of quantifying the input is the size of the input, where the input is an array of bits (each element of the array is a single bit).

The appropriate way of quantifying computational complexity is, therefore, the size of the input according to the standard way of encoding the input into an array with each element being at most a bit.

The worst-case complexity will, then, be computed in terms of the size of the input.

Problems can be viewed as a set of inputs, where all inputs satisfy some particular property as defined by the problem. The complexity class is, therefore, a set of all these problems according to some classification. In other words, a Complexity Class is a set of sets.

10.1.3 Decision problems

The kinds of problems discussed in Computational Complexity are referred to as Decision Problems because a machine will decide whether to *accept* or *reject* the input. For example, regular languages are sets of words that accepted by finite state automata. FSA are machines that can solve a particular set of decision problems.

The essential detail here, however, is that output of the problem is binary.

Because the output is binary, we can partition the outputs into two disjoint sets: The set of accepted inputs, and the set of rejected inputs.

Therefore, a problem is the set of accepted inputs because they share a common property.

Talking about languages, every input can be considered as a word, since it's just a string of symbols, and a language is a collection of such strings.

It's often said that a particular decision problem defines a language since its accepted inputs belong to a language and the rejected inputs do **not** belong to that language.

For example, the following problem:

$$\text{If } x^2 = n \text{ is an integer, is } x \text{ an integer?}$$

This is a decision problem since the answer (or output) is **true** or **false**. The real question here is the following:

$$\text{Given an integer } n, \text{ accept } n \text{ if } \sqrt{n} \in \mathbb{Z}?$$

Yet another way to phrase this problem is:

$$\text{The language } \mathcal{L} \text{ of perfect squares}$$

Let's look at two possible solutions:

1. Calculate $\sqrt{n}$

2. Calculate $1^2, 2^2, 3^2, \ldots$

Either method will be able to decide if a particular number n belongs to the language of perfect squares or not.

Given the input in standard encoding, what is the worst-case time complexity as a **function of the size of the input**?

We know that the input is stored in standard encoding, so the size of the input is $s = \mathcal{O}(\log n)$. Given this, in order to calculate the worst-case time complexity for method 1, we need to know the worst-case time complexity of computing the square root using e.g. Heron's Method. One can argue that the time complexity of Heron's Method is $\mathcal{O}(s)$; this is because Heron's method halved the distance between two numbers at each iteration in order to produce a new candidate for the square root.

Because of this halving of the distance, the number of steps taken to reach something close to the square root is $\mathcal{O}(\log n)$ which is essentially the size s of the input, hence the time complexity is $\mathcal{O}(s)$.

With method 1, we can see that its worst-case time complexity isn linear in size of the input.

When it comes to method 2, we need to iterate over all integer starting from one until an integer $k \geq n$. Let's remind ourselves that n is exponentially larger than the input size ($\log n$). This means that the worst-case time complexity will be on the order of the square root of an exponential in s. This is much worse than method 1.

If we want to think about the resources required to run this computation, all we need is a RAM model machine that operates in time that is at most polynomial in the size of the input. This means that the problem of decide the language of perfect squares belongs to the set of problems that can be decide by a machine and the RAM model in at most a number of time-steps that is polynomial in the size of the inputs.

10.2.1 Particular complexity classes

As mentioned before, complexity classes are sets of sets. More specifically, they are sets of languages.

We have already established that we can decide if a number is a perfect square in linear time in the RAM model of computation. Let's consider a different problem now where the input is an array where each element in a bit value or nothing and another number x in binary form. The size s of the input is, therefore, the length n of the array plus the size of x.

The problem is to determine if we can find a string of x consecutive bit values 1 in the input array. One possible algorithm to solve this is to perform linear search in the array and keep count of the number of consecutive 1s that appear in that array.

This algorithm will have a worst-case time complexity of $\int$, meaning it's linear in the size s of the input. Consequently, we have two problems (i.e. two languages) which can be solved in linear time with the RAM model of computation. Therefore, we can collect these two languages and place them in the same set.

In order to circumvent minor details in the model of computation chosen (RAM vs Turing Machine, for example) we group our problems into large complexity classes to allow for such minor differences in accounting. For example, we will collect every problem that can be decided by a computational machine in at most polynomial time under the same complexity class called P.

P generally captures the idea that it is the set of all problems that can be solved efficiently.

Another complexity class, EXP, is the set of all languages that can be decided in at most **exponential** ($\mathcal{O}2^{poly(n)}$) time in the size of input

Yet another complexity class is called NP. It's the set the all languages that can be decided by an algorithm in the RAM model, provided that model has access to a proof of polynomial size, with worst-case time complexity at most polynomial in the size of the input.

10.2.3 NP

NP is P without the requirement of the proof, therefore P is in NP. Moreover, NP is in EXP.

Because of this, we need a way of distinguishing between the harder problems in NP from all the others. To solve this, the concept of NP-hardness was introduced.

A problem Q is called NP-hard if the same efficient algorithm to solve Q, could be used to also solve a problem R in NP. A problem is called NP-complete if it is both NP-Hard and in the class of NP.

NP-complete are the hardest problems within the complexity class of NP. Because of this, if we can show that there exists an efficient algorithm to solve NP-complete problems, then we show that P is equal to NP and get ourselves one million USD. It is widely believed, however, that P is not equal to NP.

Week 20

Learning Objectives

- Explain why NP-complete problems are an obstacle to general-purpose efficient algorithms

Solution to Revision Exercises

- PDF

10.3.1 NP problems and searching

If a problem is shown to be NP-complete, an efficient algorithm is far from obvious. While that's true, there are methods to approach a solution to NP-complete problem even if the solution is not efficient.

One approach may be enumerate all proofs and test them one by one. The proofs are of polynomial size, so they will be of the form $\mathcal{O}(2^p)$ where p is the polynomial of the proof.

We have used a similar method in Topic 2 when testing all possible guest arrangements for the birthday problem. This idea of trying all possible candidates to a problem relates to Linear Search, where we test each element of the input array until we find what we're looking for.

With NP-complete problems we need to try all possible proofs to see if any of them results in acceptance of the input, otherwise it results in rejection. It's like having to search an exponentially large vector to solve our problem.

This means we would have an exponential algorithm to solve NP-complete problem. If we could apply Binary Search for this problem, we would have a logarithm of an exponential, which would result in a constant n. This would imply that $P = NP$, which is widely believed to **not** be the case.

As a result, Binary Search may not be useful to solve NP-complete problems. This is contrast to other problems where Binary Search is useful.

For example, our previous problem of determining if a number is a perfect square, we can use Binary Search to improve the second solution method. Basically, we would form an exponentially large vector where each element is the square of the index, then we run Binary search, which would run in linear time.

Another approach to solving NP-complete problems is Backtracking, just like we did with the labyrinth problem in the beginning of topic 4.

10.3.2 New approaches to computing

Our discussion has been specific to particular kinds of models of computing (RAM model or Turing Machine). Could it be that our models of computation are imposing limits in how we think about computation?

There are other approaches to computation. One of which is a randomised computation. In this method, a computer utilises random numbers are extra resource to help it decide what operation to do next. This means that the order of computation is not fixed (i.e. non-deterministic).

Randomised algorithms are used to e.g. simulate complex behavior. Random numbers are also at the core of cryptography.

In terms of our understanding of computational complexity, randomised algorithms do not offer a dramatic improvement over standard approaches. It's also widely believes that whatever can be done with a randomised algorithm, can also be achieved with a non-randomised algorithm.

A more dramatic change from our current view of Algorithms and Data Structures is Quantum Computing. A Quantum Computer can search an unsorted database faster than linear search. Observations such as this shows that a RAM model cannot simulate a Quantum Computer and we must, therefore, develop new algorithms and new data structures taylored for this new way of computing.