What is an Algorithm?

Algorithms are used to solve problems, in the context of computer science, the problem must be a mathmatical one.

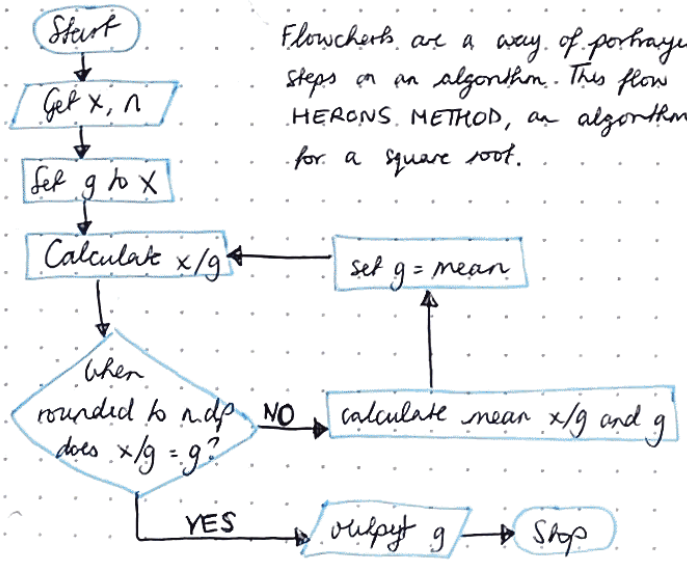
eg, you cannot ask 'should I take a holiday?' but you could evaluate: x days of holiday
y days used
x > y ? TRUE or FALSE

It is possible that a problem can be solved multiple ways, for example 'if $x^2 = 2$ is x an integer?' could be done:

1. Calculate $\sqrt{2}$ and evaluate if answer is an integer
2. Square integers from 1 to 2 and see if the result is 2.

A general all purpose algorithm that can solve many examples of the same question is always preferable over a highly specific one.

Introduction to flowcharts.



Flowcharts are a way of portraying the various steps in an algorithm. This flow chart is for HERONS METHOD, an algorithm to find an approx for a square root.

PSEUDOCODE

Pseudocode is a simple way of describing an algorithm in a format similar to computer languages.

In certain instances it a good alternative flow charts, which can be time consuming. It is also a step closer to programming languages which makes translating it to code more straightforward.

descretisation is the process of taking a continuous quantity and turning it into steps. Eg if we built a program to control a thermostat, we would provide it with discrete steps (eg 0.5°C) which we would increase or decrease by until the desired temperature was reached.

THE ASSIGNMENT SYMBOL ←

eg 1. x ← 2
2. y ← TRUE

We should use more descriptive variables than above, and never use spaces in the name.

SELF-REFERENTIAL

eg 1. x ← 2
2. x ← x + 2

Once the variable is initialised you can then use the value to change itself as above

Example 1 - THERMOSTAT PSEUDOCODE

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

functions is a general concept in programing, it takes in input and returns an output.

Below is a function that evaluates an input and determines if it is even:

```
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
```

A return statement causes the function to stop. Everything inside the function is called the 'body'. And the body is made up of 'statements'.

loops, iteration is the idea of repeating something multiple times. There are two types of loops FOR LOOPS and WHILE LOOPS. In for loops we create a variable that acts as a loop counter. When we hit the target value the loop stops. eg:

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

a while loop continues until the statement is no longer satisfied. eg:

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

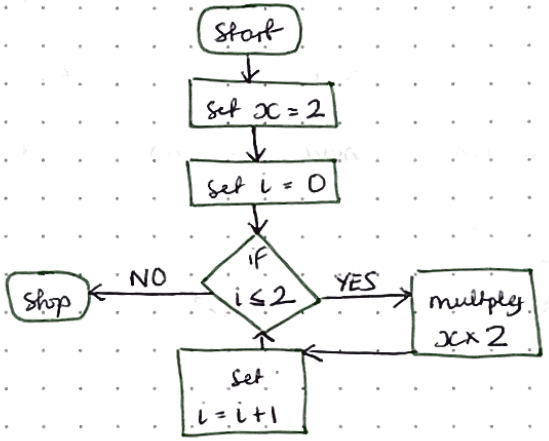
BREAK → will cause the loop to stop
CONTINUE → will skip to the next iteration

flow chart to pseudocode →

Pseudocode	Flowchart
Assignments	Basic Actions
if... else	Diamond
function	Start terminal
end function	Stop terminal
return	Parallelogram

For example, conversion of a basic for loop between pseudocode and flowchart.

```
1. x ← 2
2. for 0 ≤ i ≤ 2 do
3.     x ← x × 2
4. end for
```



ABSTRACT DATA

How we structure data is important in every day life and computing. If we have a list and want to add something to it we would usually add it to the bottom of the list.

However we also need need to consider how data is stored and processed by the computer.

When looking at abstract data structures we mostly ignore the technical details and focus on the fundamental structure and how the data can be amended.

Vectors can be a useful abstraction of memory. It is a finite fixed size sequential data collection:

Vector v

0	1	2	3
a_0	a_1	a_2	a_3

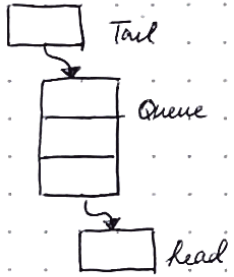
The number of elements in a vector is fixed and the number of elements in the vector is its length. Eg in the above example the length is 4.
The address of the element is also known as the elements index.

You can perform the following operations on vectors:

Operation	Pseudocode	Description
length	LENGTH [v]	Returns number of elements
Select [k]	v[k]	Returns k^{th} element of v
Store! [0,k]	v[k] ← 0	Sets k^{th} element to 0
Construct new vector	new Vector w(n)	Makes new vector w length n

As we don't always know the number of elements to be stored in a vector and a vectors length cannot be modified we can get around this by storing the data elsewhere and storing the reference to that data in the vector.

Queues occur when there is a resource which is not immediately available. They operate on a FIFO basis (first in, first out).



We can add new items to the queue meaning its length is dynamic. Elements must be added at the tail and removed at the head.

Operation	Pseudocode	Description
head	HEAD [q]	Returns the head
dequeue!	DEQUEUE [q]	Removes element
enqueue![0]	ENQUEUE [0,q]	Add to the tail
empty?	EMPTY [q]	Is the queue empty?
Construct new queue	newQueue q	New queue q

Stacks are very similar to queues only they operate on a LIFO basis (last in first out). The top of the stack is the only element that's accessable

Operation	Pseudocode	Description
push! [0]	PUSH [0,S]	Adds a new element
top	TOP [S]	Returns top element
pop!	POP [S]	Removes the top element
empty?	EMPTY [S]	Checks if stack is empty
Construct new stack	newStack S	Makes a new stack S

Arrays and Searching

Different abstract data structures affect how we search an array. In CS we make a distinction between an ABSTRACT DATA TYPE and a DATA STRUCTURE.
Where the data type is what is being stored and the data structure is how it is being stored, Eg a vector as an abstract data type, but an array is a concrete data structure.

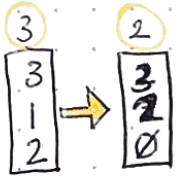
An array is simply a block of memory where we can store data (typically of the same type). The size of an array cannot be changed, but we can copy the contents of the array to a new larger one.

A dynamic array is an abstract data structure, a dynamic array is like a vector but does not have a fixed length. As a result it has all the same operations:

Operation	Pseudocode	Description
length	LENGTH [d]	Returns number of elements
select [k]	d [k]	Returns the k^{th} element
Store! [0,K]	d [k] ← 0	Sets k^{th} element to 0
RemoveAt! [K]	d [k] ← Ø	Remove k^{th} element, return k
InsertAt! [0,k]	d [k] ← 0	Inserts new element, increase length

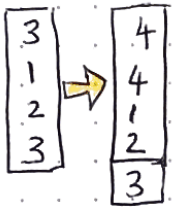
When we go through removing and adding elements of a dynamic array we effectively have to copy them to a new array with the updated length.
Typically we assume index 0 contains the length of the array.

→ removeAt![2]



1. write![element 3, 2]
   write the contents of element 3 to index 2
2. write![Ø, 3]
   write nothing to index 3
3. write![2, 0]
   update the arrays length

→ insertAt![element 4, 2]



1. write![4, 0]
   write the length 4 to index 0
2. write![element 1, 1]
   copy element 1 from old to new array
3. write![element 4, 2]
   write the new element to index 4
4. write![element 2, 3]
   copy element 2 from old array to new array
5. Write![element 3, 4]
   copy element 3 from old array to new array

VECTORS, STACKS, QUEUES and DYNAMIC ARRAYS are all represented by data in a line. For this reason, they are known as LINEAR DATA STRUCTURES.

linear search algorithms

When looking at designing search algorithms it is important to consider which data structure you are looking at. When looking at 2 questions:

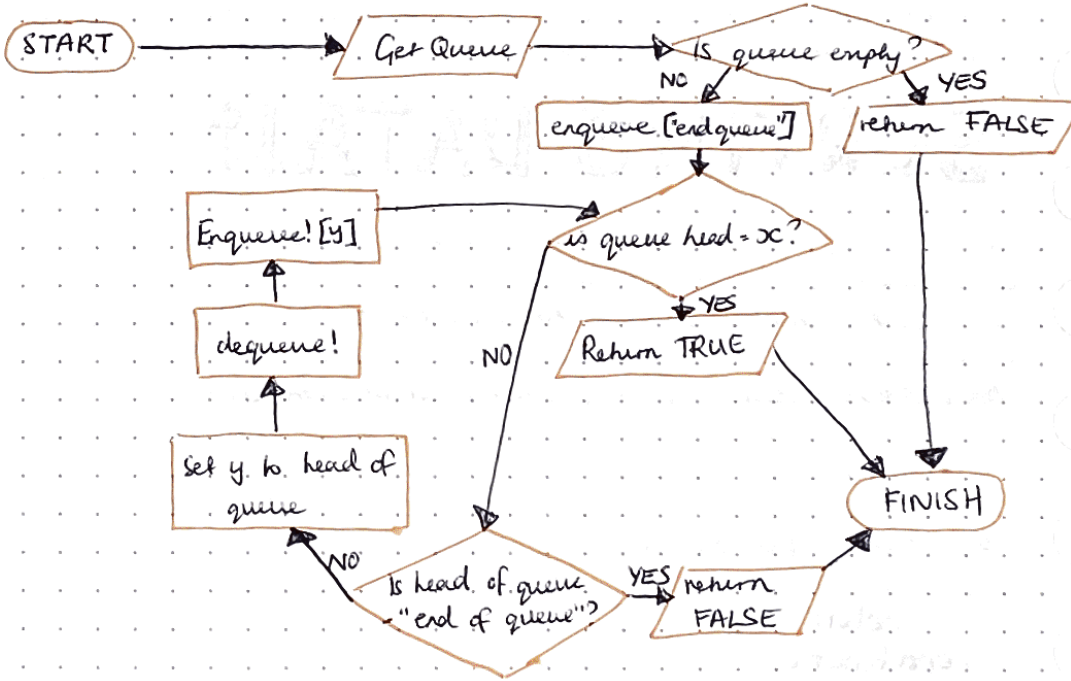
1. Is there an element of value 6?
2. Which element contains the value 6?

The first requires a simple bolean response, the 2nd requires the index number, it wouldn't care about the index number in a queue or a stack as all the elements in the data structure are not available.

For a vector or array we can use the following algorithm:

```
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
```

We can't use this search algorithm for queues and stacks, we can pull out each item and compare it, but we need to ensure the original stack and queue are not destroyed:



linked list

A **pointer** is a variable that contains a memory address. The pointer does not store the value at that address, only the address. Whenever we want to access the value at the address the pointer is pointing to we must **DEREFERANCE** the pointer by going to the address indicated.

A **linked list node** is an array of two items, the first item is a value, the second item is a pointer pointing to another node.
- The pointer pointing to the next node is 'next'
- The pointer pointing to the start is 'head'
- The last pointer to 'null' address ends the list

While it is fairly easy to insert additional nodes it is harder to simply go to a particular element as you must transverse the list from the start.

This ability to shrink or grow as needed means linked lists are preferable to implementing STACKS and QUEUES vs using dynamic arrays. Where changing an array requires copying everything to a new array.

SORTING DATA

Whenever sorting data we need to know what data structure we are using, as well as what data we are sorting.

For example numbers may be sorted ascending or descending, but alpha characters may be sorted alphabetically (lexicographical order).

An algorithm to swap places for a vector could be written as:

```
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
```


bubble Sort a bubble sort works by traversing the vector, comparing the items and swapping them. It requires at least n-1 passes to fully sort the vector. The pseudocode for this would be:

```
1.  function BUBBLESORT (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]
7.                  Swap (vector, j, j+1)
8.                  count ← count+1
9.              end if
10.         end for
11.         if count ← 0 then
12.             break          ◁ if no changes made on first pass
13.         end if                 then list is sorted.
14.     end for
15.     return vector
16. end function
```

Since we are only using vector operations we can use the bubble sort directly for arrays.

BUBBLE SORT ON A STACK - we implement an empty stack. To start, we PUSH the top of our stack onto the empty stack. Then we compare the second item to the new stack. If the top of the secondary stack is smaller we leave as is. If not we swap the items. On the first pass the largest item will be at the top, we loop again until all sorted.

Insertion Sort works by starting at the first element and sorting as you go. Eg element 5 is compared to 1-4 to find out where it should sit. See the following page for the algorithm:

```
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
```

if element j is smaller than previous elements but larger than element k. We move it into position k+1 and shift all elements to the right.

```
1  Function INSERTION SORT (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
```

if element i is less than j-1 and j > 0. aka compare i to all the elements before. If you find an element smaller than i call SHIFT()

BUBBLE SORT IN JAVASCRIPT

Swap function

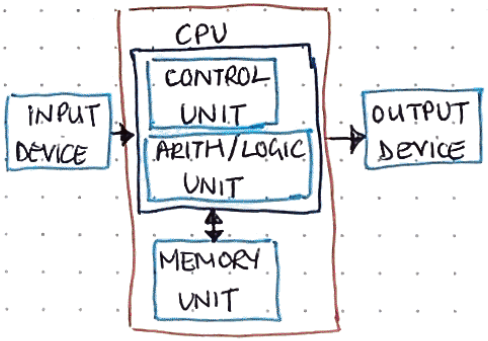
```
1. function swap (array, i, j)
2. {
3.   var x = array.~~length~~[j]
4.   array[j] = array[i];
5.   array[i] = x;
6. }
```


We can then implement the bubble sort:

```
1  function bubbleSort (array)
2  {
3    var n = array.length;
4
5    for (var i = 0; i <= n-2; i++ {
6      var count = 0;
7
8      for (var j = 0; j <= n-2; j++) {
9        if (array[j+1] < array[j]) {
10         swap (array, j, j+1);
11         count++;
12       }
13     }
14
15     if (count == 0)
16       break;
17     }
18
19   return array;
20 }
```

RANDOM ACCESS MACHINE

Before we consider the best type of algorithm for a problem we need to consider the computer architecture. The random access machine models the behaviour of the CPU



most modern computers follow this architecture.
RAM changes based on computations
Modern computers deal in bytes rather than bits (8 bits = 1 byte)

Random access refers to how the machine can access data in any register in one single operation. eg if we know the address we can access it.

The CPU unit has
- A program counter, keeps track of where we are
- A control unit, read, writes, copies and performs basic operations
- An array of registers where information is stored for current computation

Using this knowledge we can consider how particular data structures are implemented by the computer. For example an array would be information stored in consecutive registers where each register address is the index of the array.

With this knowledge we can consider ~~who~~ how many steps a computer must take to solve a particular algorithm. For example the below algorithm is converted into the below steps

```
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. If $i \leq n$ go to 6 else go to 9
6. multiply i and a and store result
7. Increase i by 1
8. Goto step 5
9. Store a in output and stop

When examining the efficiency of an algorithm we consider a couple of factors.

TIME - In the above example the bigger n is the longer it takes to run as it must loop n number of times

MEMORY - We also need to consider the number of registers needed to ~~use~~ do the calculation.

For the factorial program steps 1-4 happen once, 5-8 happens n times and step 9 happens once. Therefore the total number of operations can be described as:

$$4 + 4n + 1 \text{ or } 4n + 5$$

When considering the time we look at the most significant operation. Since $4n + 5$ is a linear function, it grows with n.

An important question to consider is if the function input grows by 1 how big is the step up in time eg

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

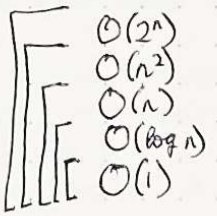
The study of this growth motivates big O notation

Big O Notation

When we consider big O notation we are only interested in the fastest growing components ignoring the constants eg

$$2^n + 3n \Rightarrow O(2^n)$$
$$1000n^2 + n \Rightarrow O(n^2)$$
$$35 \Rightarrow O(1)$$

It is important to note that when we talk of $2^n + 3n = O(2^n)$ we are saying that it belongs to the class of $O(2^n)$ but **not** that it is equal to it.



It is important to envisage these as nested, but we are interested in finding the lowest level.

TIME COMPLEXITY is a quantification of the time it takes to run an algorithm. If we consider each step takes a chunk of time we are looking at the number of time steps

The time complexity will vary depending on the type of input, for example on a bubble sort, best case scenario is the initial array is already sorted. Worst case scenario is that it is in descending order. The O is different for each input.

In order to get around this we consider WORST CASE SCENARIO

IMPUT SIZE

At the moment we have standerised our number storage, but it is worth noting that as an input number gets bigger it may get stored across multiple registers. And therefore more time is needed to recall each number.

Binary Search

The binary search algorithm is a very efficient method of searching a sorted array.

Take the following array, where we are searching for 2

0 1 2 3 4 5 6 7 8 9 10

We take the mid point 5, since $2 < 5$ we can discard the right

0 1 2 3 4 5

We take the midpoint 2, since $2 = 2$ we have found our item so we return TRUE

The binary search algorithm is depicted in pseudocode as follows:-

```
1  function BINARY SEARCH (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
```

The best case scenario for binary search, is that the value is at the midpoint so the number of operations is $O(1)$.

The worst case scenario is if the item is not found or if it is next to the very first midpoint. Since we are halving each time the complexity is not quite n, but rather $O(\log n)$.

We cannot use binary sort on unsorted arrays, because we run the risk of excluding the item when we discard parts of the array.

Binary search is the optimal search method.

Decrease and Conquer

Recursion involves self-reference. The self-reference can be used to solve problems by reducing them to smaller chunks.

There are 2 main approaches

1. DECREASE AND CONQUER 2. DIVIDE AND CONQUER

DECREASE + CONQUER

Consider $n!$ or $n \times (n-1) \times (n-2) \ldots$ $(3! = 3 \times 2 \times 1)$

Since $n! = n \times (n-1)!$ is the same thing ~~thing~~, we can look at trying to solve it using recursion.

We first start with the base case, a simple instance of this problem that is easy to solve. In this case it is $0! = 1$.

We then write the algorithm for this base case:

```
1. function FACTORIAL (n)
2.    if n = 0 then
3.       return 1
4.    end if
5.    return n x FACTORIAL (n-1)
6. end function
```

note it is important to have the base case statement to prevent the recursion from looping infinitely.

> NOTE: Anything that can be solved recursively can be solved iteratively and vice versa.

Recursion can be preffered over iteration as it can make a far more elegant and easy to understand solution.

```
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
```

This is Euclids algorithm for finding the greatest common ~~denominator~~ divisor for inputs a and b.

Recursive searching and sorting

Searching recursively is fairly straightforward to implement, if we have a vector with n elements and the value we are looking for is not in the first element then we have a further n-1 elements to search.

The bubble sort can be implemented recursively, we just need to know that after the first pass the uppermost value will be the highest number so can be ignored.

```
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 BUBBLESORT (vector)
14    n ← LENGTH [vector]
15    return SORT (vector, n)
16 end function
```

if you are at the last element of the vector then return sorted vector

go through one pass of the vector

recursively loop through the next pass

CALL STACK

There are several key questions when considering recursive equation

- HOW IS RECURSION HANDLED DURING EXECUTION?
- HOW CAN A FUNCTION CALL ITSELF AND KEEP TRACK OF VALUES?

The call stack is the data structure that stores information about called functions. The call stack

- WORKS IN THE BACKGROUND OF THE PROGRAM WHILE ITS RUNNING
- MANAGES WHAT FUNCTIONS ARE CALLED AND WHEN
- MANAGES ALL THE DATA USED BY THE FUNCTION

The details of the stack will depend on PROGRAMMING LANGUAGE, 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 to the stack forming the STACK FRAME.

Even when we call the function from the function we still get a new stack frame pushed onto the stack, because of that we can solve problems recursively.

```
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
```

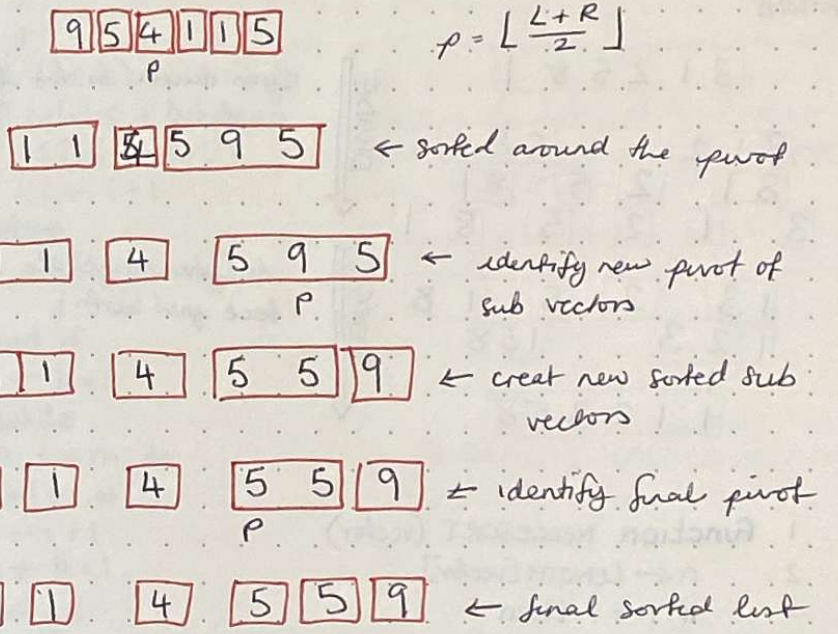
Quick Sort

is a divide and conquer algorithm where the problem is split into smaller subproblems which is each solved recursively.

Quicksort works by partitioning our input according to the value of a specific element.

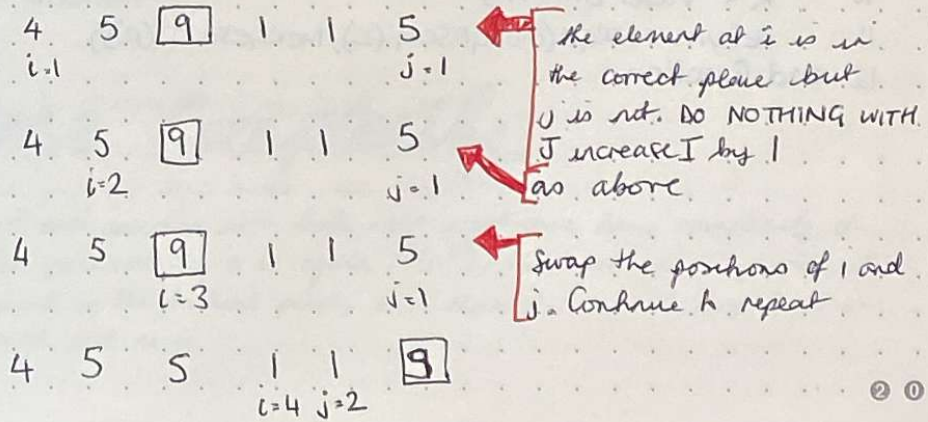
We take the unsorted vector and split it typically at the midpoint. This is called the pivot.

We then sort the data by comparing each element to the pivot and moving it to the left if smaller and the right if larger



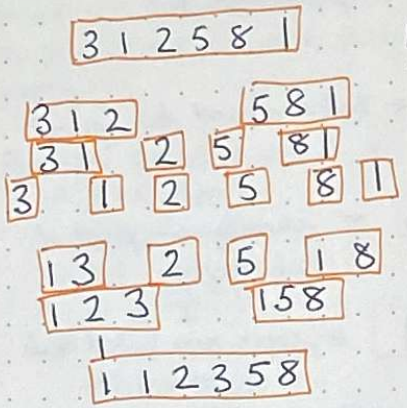
when it comes to sorting the numbers to the corect side of the pivot we store the value of the pivot element, starting from the outside we compare each left and right item to the pivot value and swap them to the right side.

eg



merge sort

Merge sort works by splitting out the vector into single element vectors as these are combined they are then sorted. A visual representation of this is as follows



DIVIDE — you divide to the bottom level

MERGE — As you merge the vectors back you sort

```
1  function MERGESORT (vector)
2      n ← LENGTH [vector]
3      if n = 1 then
4          return vector
5      end if
6      m ← ⌊(n+1)/2⌋
7      new VECTORL (m)
8      new VECTORR (n-m)
9      L ← vector [1 : m]
10     R ← vector [m+1 : n]
11     return MERGE(MERGESORT (L), MERGESORT (R))
12 end function
```

This will continue down to the base case before wrapping all the way back up.


```
1  function MERGE (v, v)
2      m ← LENGTH [v]
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 v[i] < v [j] then
10             s [k] ← v [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] ← v [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
```

compare the left vector to the right vector and put the lowest item into vector s

this handles any remaining elements of vector v not sorted

this handles any remaining elements of vector v not sorted

time Complexity

Bubble sort and insertion sort both have worst case time complexity of $O(n^2)$. for quicksort big o is again $O(n^2)$, the worst case is when the largest element is the initial pivot, and that the second largest is in the next pivot and so on

if it has the same O as bubble and insertion, why bother with quicksort?

The worst case scenario in this instance is not very likely to happen. Because we are dividing the problem for each iteration it's typical big O is $O(\log n)$

The average case is beyond the scope of this module, but since quicksort has a good typical big O, it is frequently used in libraries.

MERGE SORT has the same time complexity for best, average and worst case scenarios, $O(n \log n)$.

When it comes to memory, merge sort requires at least $O(n)$ extra space.

Algorithm	Worst case time	Worst case space
Bubble Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Quicksort	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n)$

COMPUTATIONAL Complexity

Can we conceive quick and efficient algorithms to solve certain problems?

Firstly we group problems into complexity classes, the study between these classes and their relationships is called computational complexity

Computational complexity works at an abstract level, a famous example is the Halting problem. Can a computer confirm if another computer will run infinitely or solve the problem.

We look at big O and time complexity as a way of determining how efficient an algorithm is. If there is no efficient algorithm for solving a problem, then the problem is hard.

Complexity classes contain problems not algorithms.

Decision Problems

The kind of problems considered for computational complexity are called decision problems as a machine will either accept or reject the input.

Think finite automata, but the output is binary

We can split the outputs into two distinct sets, of inputs that are accepted or rejected.

The language is all of the accepted inputs to the program.

Consider the following problem

If $x^2 = n$ is an integer, is x an integer?

Since we are seeking a yes/no output this is a decision problem. So the real question is

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

It could also be phrased as

The language L of perfect squares

This could be solved by finding $\sqrt{n}$ or by $1^2, 2^2, 3^2, 4^2 \ldots$

Either method is capable of deciding whether n belongs to the language of perfect squares or not. But which is the most efficient

With the first method all we need to do is find the $\sqrt{n}$ and then identify if the result is an integer. We can see that time complexity is linear in nature and is dependent on the size of the input in terms of storage

For the second method you would need to iterate over $1^2, 2^2, 3^2$ until you got to a number larger than n before you could evaluate the answer. Thus the time complexity is exponential in nature and much worse in terms of efficiency than the ~~the~~ first solution

Complexity Classes

As mentioned before, complexity classes are sets of sets, or more specifically sets of languages.

Imagine we have an array as an input, where each element is storing a bit value or nothing. We are also given a number x in binary form.

The size of the input is then the size of the array plus the size of x.

The problem is to identify if the value 1 appears x times in the array. A method of doing this is using linear search and then count of each time it comes across 1.

This means it has a complexity that is in line with it's input size s or $O(s)$. As 7 elements in the array = 7 elements to pass over.

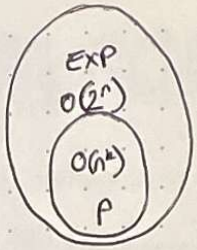
Since this is also linear in nature, we would put it in the previous perfect square problem in the same class

It is worth noting that the type of model used may change the time complexity slightly due to how data is represented eg RAM model vs Turing machine.

To solve these small occurring differences we group these problems into slightly larger classes

P is set of all languages that can be decided by an algorithm in the Ram model with worst case time complexity at most polynomial in the size of input
THESE ARE EFFICIENT $O(s)$

EXP is the set of all languages that can be decided by an algorithm in the RAM model with worst case time complexity at most exponential in the size of the input $O(2^{poly(s)})$



It is worth noting that P is contained within EXP.
This containment is strict.

NP is the class of problems that can be VERIFIED in polynomial time, but SOLVED in non-polynomial (aka exponential time)

NP-hard are problems that are at least as hard as any problem in NP complete. Where the same efficient algo to Q could be used to solve R

NP-complete are the hardest problems. They are part of NP-hard AND NP. They are unlikely to have a solution in polynomial time