01010101 01010100 01000011 00100000 01001100 01000101 01000101 01000100 01010011

A Taste of A-level Computer Science

‘Thinking Recursively’

**Contents**

[Preface 2](#_Toc41740888)

[1. Programming language constructs 3](#_Toc41740889)

[2. Factorial, Iteratively 4](#_Toc41740890)

[2.1 Natural Numbers 4](#_Toc41740891)

[2.2 Factorial 4](#_Toc41740892)

[3. Factorial, Recursively 5](#_Toc41740893)

[3.1 Recursion 5](#_Toc41740894)

[3.2 The general case 5](#_Toc41740895)

[3.3 Stack Overflow 6](#_Toc41740896)

[3.4 The Base Case 7](#_Toc41740897)

[4. Maintaining Order 8](#_Toc41740898)

[5. Binary Trees 10](#_Toc41740899)

[5.1 Binary Search Tree 10](#_Toc41740900)

[5.2 Building a Tree 10](#_Toc41740901)

[5.3 Implementing a Binary Tree 12](#_Toc41740902)

[5.4 Searching a tree 14](#_Toc41740903)

[5.5 Tree Traversal 16](#_Toc41740904)

# Preface

At UTC Leeds, students in Year 12 studying A-level learn the **C#** programming language. Our GCSE students learn the **Visual Basic** programming language, which means that students who join us in Year 12 who have used **Python** at their previous school, are able to start their C# learning journey together with our own students – as a new language for everyone in September!

The work in this booklet is not specific to any programming language – rather the principles and logic behind programming. Code is presented in pseudo-code, independent of any language, and you can work through the tasks by writing programs in any language you like (answers to all tasks are available in Visual Basic and Python – please email me to request code solutions to tasks).

You are encouraged to address the tasks throughout this document by building a working program. In all cases pseudocode is given, but in some cases, it is left incomplete to challenge you – for additional reinforcement you can pull out the text boxes behind the task box to reveal complete pseudocode templates. You should then attempt to implement the pseudocode by writing a program – and ideally getting it to work! Don’t worry if that part seems to elude you though – I intend to provide video conferencing lessons for prospective students.

Damian Whitehouse

Subject Leader of Computer Science

damian.whitehouse@utcleeds.co.uk

***Don’t look a fool***

C# is pronounced “C Sharp” never “C Hash”.

C# is a Microsoft proprietary programming language, which extends the C programming language to include object-orientation, reimagining C++.

C# is very similar to Java.

# 1. Programming language constructs

All third generation (imperative) programming languages have constructs for sequence, selection and iteration.

**Sequence** – statements are executed in order and program control, when passed to a subroutine, returns back to the position of the call when the subroutine finishes, e.g.

CipherText = Encrypt(PlainText)

in this statement, program control is passed to subroutine Encrypt which returns control to this statement and returns a value to be assigned to the variable CipherText.

**Selection** – choosing a code pathway based on a question, e.g.

If age >= 14 And age <= 18 Then

 Output(“You can join UTC Leeds.”)

Else

 Output(“You do not fall within our age range.”)

EndIf

only one of the output statements is executed, depending on the value of variable age.

**Iteration** – repeating statements on a loop: which may be count controlled, e.g.

For i = 1 to 10

 Output(i \* 2)

EndFor

or condition controlled, e.g.

There is also another example of **sequencing** here – program control is passed to the CountFactors subroutine, and then control returns back to the statement which made the call, to evaluate the While condition.

Sub CountFactors(n)

 Count = 0

 For i = 1 To n

 If n Mod i = 0 Then

 Count = Count + 1

 EndIF

 EndFor

 Return Count

End

While CountFactors(n) > 2

 Output(“Not prime, try again.”)

EndWhile

the For loop executes exactly ten times, but the While loop executes as many times a needed until the user enters a prime number.

***Task 1***

Write a program that continuously prompts the user to enter a prime number, until they do, and outputs “That is a prime number” after the loop.

Is that all there is to programming?

# 2. Factorial, Iteratively

## 2.1 Natural Numbers

In mathematics and computer science, the natural numbers are depicted as set ℕ, used for counting (cardinal numbers) and ordering (ordinal numbers).

The set ℕ contains all whole numbers from 1 to ∞.

## 2.2 Factorial

The factorial function is denoted as an exclamation mark, e.g. Factorial(4) is denoted as 4!

Factorial is the operation of multiplying any natural number with all the natural numbers that are smaller than it, e.g.

4! = 4 \* 3 \* 2 \* 1 = 24

Like the infinite set ℕ, calculating factorial could have infinite terms

n! = n \* (n-1) \* (n-2) \* … \* 2 \* 1

Of course, we don’t need n to be very large before n! is too large for a 64-bit computer to handle (even in a programming language using a 128-bit numeric variable).

*Template code pull out* 🡪

Sub Factorial(n)

 f = 1

 For i = 2 To n

 f = f \* i

 EndFor

 Return f

End

***Task 2***

Develop this pseudocode into a working program by writing a subroutine which calculates factorial. The program should output the factorial of a number entered at run-time.

n = Input()

Output(Factorial(n))

Sub Factorial(n)

 …

End

# 3. Factorial, Recursively

## 3.1 Recursion

There is more to controlling program flow than sequence, selection and iteration constructs. The fourth imperative language construct is recursion. Actually, recursion can be used instead of iteration – although iteration is still the right choice for some situations.



This lithograph art by M.C. Escher, created in 1948, encapsulates the essence of recursive programming.

## 3.2 The general case

Let’s consider a different way to calculate 4!

4! = 4 \* 3!

The hard work of calculating 3! is assumed, and we just multiply that by 4.

If we generalise this approach,

n! = n \* (n-1)!

Let’s use this idea to pull a programming fast one…

Modify your factorial subroutine so that it does not use a loop.

Sub Factorial(n)

 Return n \* Factorial(n – 1)

End

A subroutine that calls itself – no way! That’s a recursive subroutine.

## 3.3 Stack Overflow

Not quite there yet – we have infinite recursion and that stack overflow is a nasty run-time error!

The first call made would actually be the last one to return – as the value it would return is based on what another call returns – except it never does.

Subroutine calls use a First In Last Out data structure called a stack – often referred to as a Last In First Out data structure.

The trouble is, this recursion gets deeper into a chain of recursive calls that never go back to return anything, as more calls keep getting pushed onto the stack. Eventually the stack runs out of space to handle any more subroutine calls, causing a run-time error, specifically a stack overflow error.

Suppose we call the subroutine Factorial(3), look at the trace table below, the value of n is decremented infinitely.

|  |  |
| --- | --- |
| **n** | **Return** |
| 3 |  |
| 2 |  |
| 1 |  |
| 0 |  |
| -1 |  |
| -2 |  |
| -3 |  |
| -4 |  |
| -5 |  |
| -6 |  |
| … |  |
| -∞ |  |

We need one of the subroutine calls to return a value, so the recursive chain can unwind and the first call made can return the answer to the part of the program which made the first call.

## 3.4 The Base Case

For a recursive subroutine to work, we need at least one possible pathway through the code that ends the subroutine, without making another recursive call – this prevents an infinite loop and allows the recursion to unwind in reverse order.

Look again at the general case,

n! = n \* (n-1) \* (n-2) \* … \* 1

What is the final value use in the calculation? What value for n should result in that value being returned?

**1! = 1**

When n = 1 we should return 1.

This is the base case – the only value for n that cannot be calculated, it is defined.

The trace table shows the order in which the calls are made, and then the unwinding in reverse to calculate the result, e.g. if we make the call Factorial(3) further calls are made, with the first call to return anything being Factorial(1), this allows the recursion to unwind and the earlier calls to return a value.

***Task 3***

*Modify your subroutine to include a base case*

Sub Factorial(n)

 Return n \* Factorial(n – 1)

End

Sub Factorial(n)

 If n = 1

 Return 1

 Else

 Return n \* Factorial(n - 1)

 EndIf

End

*Template code pull out* 🡨

|  |  |
| --- | --- |
| **n** | **Return** |
| 3 |  |
| 2 |  |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |

The first call made, Factorial(3) finally returns the value 6.

# 4. Maintaining Order

Let’s consider the four basic algorithms on the GCSE course

* linear search – slow
* binary search – very fast (but only works on sorted lists)
* bubble sort – very slow
* merge sort – fast

Suppose we need to store a list of values in an array/list. The values are entered as they come, but an ordered output is needed, and we need to be able to search the list often.

Because new items are appended to the end of the list – we need to perform a sort after each append in order to maintain an ordered list.

1st insert is “Rod”, 2nd is “Zippy”, sorting makes no change

|  |  |
| --- | --- |
| 0 | Rod |
| 1 | Zippy |

3rd insert is “Jane”,

|  |  |
| --- | --- |
| 0 | Rod |
| 1 | Zippy |
| 2 | Jane |

Sorting rearranges array

|  |  |
| --- | --- |
| 0 | Jane |
| 1 | Rod |
| 2 | Zippy |

4rd insert is “Freddy”,

|  |  |
| --- | --- |
| 0 | Jane |
| 1 | Rod |
| 2 | Zippy |
| 3 | Freddy |

Sorting rearranges array

|  |  |
| --- | --- |
| 0 | Freddy |
| 1 | Jane |
| 2 | Rod |
| 3 | Zippy |

***Task 4***

Build this program to sort a list

list = {}

list.Add("Rod")

list.Add("Zippy")

list.Add("Jane")

list.Add("Freddy")

Sub Add(list, item)

 list.Add(item)

 Sort(list)

End

Sub Sort(list)

 …

End

As there is no getting away from the need to sort the list after each addition – a time costly bubble sort or a more efficient merge sort could be used.

For small lists, who cares? For very large lists – even a merge sort isn’t as good as the awesome method which follows – a binary search tree.

# 5. Binary Trees

A binary tree is a rooted tree (based on a hierarchy) where each node can have at most two children – one left and one right child.

## 5.1 Binary Search Tree

A binary search tree is structured in such a way that the value of a node’s left child must be less than its own value and the value of the right child must be greater than its own.

Instead of appending to and sorting a list – we shall instead build a binary tree.

## 5.2 Building a Tree

We need to start somewhere, so the first value entered will always become the root node of the tree.

1st value is “Rod”, so tree looks like this



2nd value is “Zippy”, so we go to the right of “Rod”



3rd value is “Jane”, so we go to the left of “Rod”



4th value is “Freddy”, so left of “Rod”, but “Rod” already has a left child node – so where do we put “Freddy”?

For each node we visit, starting from the root node, we compare to the value being inserted and either go left or right, repeating until we find a vacancy to insert into.



***Task 5.2***

Complete the tree for the values given *(solution behind box)*

~~Rod~~

~~Zippy~~

~~Jane~~

~~Freddy~~

Bungle

George

Jeffrey



## 5.3 Implementing a Binary Tree

We need to define a node class, which contains a data value along with a pointer to each child node, initially set to nothing.

Class Node

 left = Nothing

 right = Nothing

 data = ""

End Class

When we create a new node instance, we will assign a data value.

The tree itself is just a single variable pointing to the root node. Initially nothing, for an empty tree.

root = Nothing

We now need a subroutine to add new nodes to the tree.

Sub Add(inData, n)

 If n is Nothing Then

 n = new Node()

 n.data = inData

 End If

End Sub

This basic subroutine will cope with adding the first node to the tree, e.g. “Rod”.



Add("Rod", root)

Coding to add the next node will require some recursive thinking!

Add("Zippy", root)

Our Add subroutine will need to consider what to do when n is not Nothing.

*Template code pull out* 🡪

Sub Add(inData, n)

 If n is Nothing Then

 n = new Node()

 n.data = inData

 Else If inData < n.data Then

 Add(inData, n.left)

 Else

 Add(inData, n.right)

 End If

End Sub

***Task 5.3***

Extend the Add subroutine to make two recursive calls, to go to the left or right of the given node.

Write a program to implement the binary tree.

Sub Add(inData, n)

 If n is Nothing Then

 n = new Node()

 n.data = inData

 Else If … Then

 …

 Else

 …

 End If

End Sub

## 5.4 Searching a tree

Suppose we are looking for a specific value, e.g. search for “strawberry”



First, we examine the root node of the tree, e.g. compare “strawberry” to “banana”, this allows us to decide whether to go left or right.

Then, we examine the next node, e.g. compare “strawberry” to “peach”, and decide to go left or right.

Then, we examine the next node, e.g. compare “strawberry” to “pear”, and decide to go left or right. As “pear” has no right child node, we conclude that “strawberry” does not exist in the tree.

Summary – to search the entire binary tree for “strawberry” we only had to examine the values “banana”, “peach”, “pear”.

***Task 5.4.1***

What is the maximum number of nodes that would need to be examined to search a binary tree containing 256 nodes?

2? = 256

In mathematical terms

Log2 256 = 8

*Pull down for answer* 🡫

The algorithm to search a binary tree has two recursive cases - go left or go right. It also has two base cases – the value is found or it isn’t in the tree.

*Pull out for code template* 🡪

Sub Search(value, n)

 If n is Nothing Then

 Return False

 ElseIf n.data = value Then

 Return True

 ElseIf value < n.data Then

 Search(value, n.left)

 Else

 Search(value, n.right)

 EndIf

End Sub

***Task 5.4.2***

Complete and code this subroutine into your program.

Sub Search(value, n)

 If n is Nothing Then

 Return False

 ElseIf n.data = value Then

 Return True

 ElseIf … Then

 …

 Else

 …

 EndIf

End Sub

## 5.5 Tree Traversal

When searching for something, we stop looking when we find it, so a search is not guaranteed to visit every node in a data structure.

To traverse a data structure means to visit every node, in some predefined sequence.

We are going to start at the root node and always go left, passing each node on its **left** side, e.g.

from “F” to “D” to “B” to “A”



### Backtracking

Once we get as far left as possible, we move around the node, passing **underneath** it.

Sometimes we may pass underneath and immediately to the node’s **right**, e.g. “A”.

But sometimes we may pass underneath a node and then to its right child, before returning back to it, e.g. from “B” to “C” then back to “B”.

Backtracking is where recursive subroutine calls unwind.

### In-order Traversal

There are actually three ways to traverse a binary tree.

An in-order traversal outputs the tree in alphabetical order (assuming the data values are characters/strings – it would of course output in numerical order if the data values were numbers).

The steps of the in-order traversal algorithm are:

* Traverse left subtree
* Output node
* Traverse right subtree

To refine that as a recursive subroutine, in high-level pseudocode:

Sub InOrderTraversal(node)

 If left node exists Then recurse left

 Output node

 If right node exists then recurse right

End

*Pull out for code template* 🡪

Sub InOrderTraversal(node)

 If node.left is not Nothing Then

 InOrderTraversal(node.left)

 EndIf

 Output node.data

 If node.right is not Nothing Then

 InOrderTraversal(node.right)

 EndIf

End

***Task 5.5***

Expand this high-level pseudocode and add a traversal subroutine to your program to output the tree values in the correct order.

Sub InOrderTraversal(node)

 If left node exists Then recurse left

 Output node

 If right node exists then recurse right

End