A2 Computer Science Project

Contents:

[**Analysis**](#_el8cd4pjbjuh) **4**

[Problem Definition](#_urrxwsmvzp7u) 4

[Stakeholders](#_gxq0ynshw38t) 4

[Existing Solutions](#_gpmjqkr1vcey) 4

[Investigation](#_kwt2p0hgxa9t) 7

[Solution Requirements and Success Criteria](#_5l91wboy5mr5) 10

[Computational Methods](#_jykczlwg7ef1) 11

[System Requirements](#_39otaji6p4tr) 11

[**Design**](#_33ovf3mb2ph3) **12**

[Objective](#_3tdqwumdiv08) 12

[Interface Design](#_7s5d8xjmvnz5) 12

[Main Menu](#_ro7wozckycfn) 13

[Original design](#_oqay015ot8tf) 13

[Version 1.1](#_ardnbb160djh) 13

[Version 1.2](#_t3uf9p3sbjzk) 14

[Sorts and searches](#_v7d7blo62x7r) 14

[Original design](#_cccc6338wwg5) 14

[Version 1.1](#_qas68m6hf6bo) 15

[Version 1.2](#_2fl13pbdx7pu) 16

[Decomposition and Flowcharts](#_1r6ve18295ev) 16

[Main Menu Decomposition](#_vudzn3qnbqbg) 17

[Sort/Search decomposition](#_9ehl1cs9l4dr) 17

[Main Menu Flowchart](#_kg3vxtampvs3) 18

[Bubble Sort Flowchart](#_eru7awep2gmj) 20

[Linear Search Flowchart](#_sar8kp9xulj8) 22

[Data Dictionary](#_2c31wpjeygmo) 24

[Sample Pseudocode](#_cybiq63w8xby) 26

[Main Menu](#_w1mrwwtquwp9) 26

[Bubble Sort](#_vj1qjm9b31hu) 27

[Linear Search](#_4bt55wvm1ptn) 28

[**Development of the coded solution**](#_9q4lolkvptld) **28**

[Plan for each iteration](#_rwhept8gjitx) 28

[Iteration 1](#_5q8fo6wu9qzh) 28

[Iteration 2](#_n4fz55xhqjj) 28

[Iteration 3](#_28lwnfytzyt8) 29

[Main Menu](#_z93qg8jbo58d) 30

[Implementation in iteration 1.1](#_umciwy8jm6rh) 30

[Form Initialisation](#_r0mks6sgolpc) 30

[Exit](#_gfx57vyyalj0) 30

[Creating user array](#_numq5m1eylqd) 30

[Creating random array](#_xibvw3io0ydj) 31

[Array size text box validation](#_bl7c04am4oae) 31

[Array data text box validation](#_m9amoxgh1qvy) 32

[Add number button with validation](#_uewes36lf09p) 33

[Update array button](#_bdo79b40ei7d) 33

[Start button with validation](#_honmkttknduq) 34

[Show list button with validation](#_xoeb5zxaadhx) 34

[Array creation help button](#_he41hwnkdv9e) 34

[Draw bars button](#_zeg39e4l24bt) 35

[Draw bars function](#_kkx9u8ln8kse) 35

[Run-time graphs](#_548gg163bxvr) 36

[Sorts](#_9fl0ge3br1iy) 37

[Bubble Sort](#_2osf1zhd58ac) 37

[Information](#_djg3we1395zu) 37

[Implementation in Iteration 1](#_whvnrlxuewfn) 38

[Button](#_bekm0pae2dm) 38

[Function](#_8wzkjko9ua1z) 38

[Insertion Sort](#_57dwv5qup2nd) 39

[Information](#_r6vy8aep6l2r) 39

[Implementation in iteration 2](#_s5ipg87axndw) 39

[Button](#_pu09e8yi17f) 39

[Function](#_uxb9shnzz8ec) 40

[Merge Sort](#_u1l0ti31w1jk) 40

[Information](#_tp828dchbe9z) 40

[Implementation in iteration 2](#_34x3fuhlvv4c) 40

[Button](#_a9ozzwudph57) 41

[Functions](#_guvn773agmj9) 41

[Splitting](#_kel5bkh5s3o) 41

[Splitting with showing steps - shows changed code](#_62yxwedwl1d) 42

[Merging](#_geddghzmnu7) 42

[Merging with showing steps - shows changed code](#_ytyr8eankn3a) 43

[Quick Sort](#_z8lelgkgtww0) 43

[Information](#_m552tuvu2z8b) 43

[Implementation in iteration 2](#_5n0y4s5bxfap) 43

[Button](#_3rljvy62spra) 44

[Functions](#_p81mfbsfs41d) 44

[Quick Sort](#_vh2yi44fsl6p) 44

[Partition](#_h5vx2mwjkihc) 45

[Partition allowing for duplicate numbers](#_g1q3lf5ynfdy) 46

[Searches](#_rwhept8gjitx) 46

[Linear Search](#_rrkalivih2sz) 46

[Information](#_bt6llltv5gqv) 46

[Implementation in iteration 1](#_v7l2puuetuxy) 47

[Button](#_pbejmvvheink) 47

[Function](#_daxhcqheg341) 47

[Binary Search](#_kryrxatfngwz) 47

[Information](#_du7skbdoiayl) 47

[Implementation in iteration 2](#_k99nuawxq6po) 48

[Button](#_fwsfpjph22r2) 48

[Function](#_2inqxaoxemlk) 48

[Testing](#_rwhept8gjitx) 48

[Testing Table](#_sd0v07xc18yf) 48

[**Evaluation**](#_dfal966q04u2) **48**

[Meeting the requirements](#_vflwm99hmrkl) 48

[Results of each iteration](#_gmybxi6gmk7l) 48

[Iteration 1](#_ym5q2861kag5) 48

[Iteration 2](#_7i2r4mqx2m5b) 48

[Iteration 3](#_k29ttv9kv3lh) 49

[Feedback from users](#_sswr90xfp5e1) 49

[Iteration 1](#_99xvghc0n3p4) 49

[Iteration 2](#_n7w2087suur1) 49

[Iteration 3](#_r7yih5lpyt3k) 49

[Limitations](#_m7isandcww0k) 49

[Improvements](#_m7isandcww0k) 49

[**Bibliography**](#_wbvhgd8oopgh) **49**

[**Links - to be removed before submission**](#_1cdf8fdz03g2) **51**

# 

# 

# 

# 

# 

# 

# 

# 

# Analysis

## Problem Definition

There are many different sorting and searching algorithms used in the computer science industry today. Sorting algorithms provide students with some backbone concepts of computer science such as data structures, big O notation and efficiency. Some of these are taught in GCSE and all the way up to A-level and university classes and may appear to be confusing to some students, especially younger ones.

This solution will present a graphical representation of some common algorithms that are both efficient and inefficient. It will highlight that there is no ‘perfect’ algorithm and many choices are based on the context of the data used. This solution is intended for educational purposes.

## Stakeholders

The stakeholders are made up of those with an interest in Computer Science. With those that are just curious about the subject to those that are studying it at GCSE, Vocational and A-Level. Potential users are those that have an interest in computing and/or maths and want to know how sorting and searching algorithms work, as well as how they perform in comparison to each other.

Teachers and tutors may use this also, to teach their students about these algorithms in a simple way. Users will select an algorithm, give it data to populate the array or have it randomly generated and watch how each one performs. Students will use this program for a few minutes at a time, depending on the algorithm used, how complex the array is and the number of algorithms they want to look at.

This could be a useful revision tool as it gives students quick access to a variety of algorithms from a laptop / desktop pc.

The designer will maintain the code, keep it neat and easy to understand and respond to the feedback of stakeholders/users.

## Existing Solutions

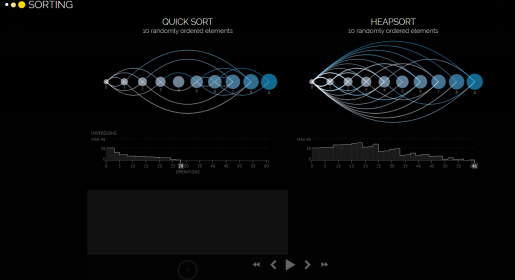
Two current solutions are [sorting.at](http://sorting.at/) and [visualgo.net/bn/sorting](https://visualgo.net/bn/sorting). These are both web-based solutions to sorting algorithms. Both of them have a limited amount of input available to the user. Sorting.at can have multiple sorts running at the same time, thanks to this it is easier to see which sort performs better depending on context.

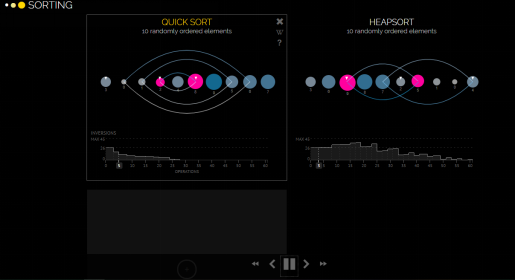
However it may seem a little complicated at first but it does link to the respective Wikipedia article to explain each sort.

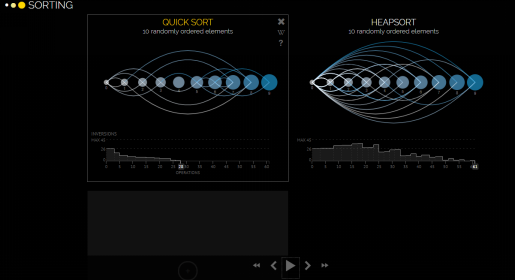
visualgo.net’s representation is much more intuitive and the pseudocode for each sort goes alongside the visuals as it runs. It also gives more freedom to the user with the data being used.

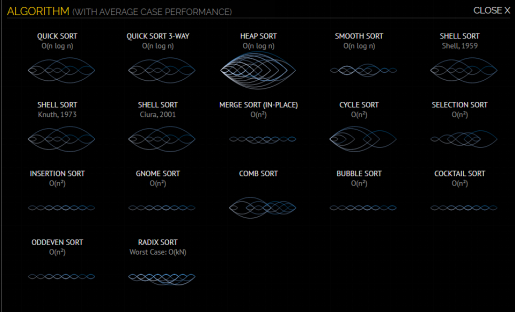
From research into existing solutions, I have concluded that my own solution would benefit largely from having pseudocode alongside each sort to aid the explanation process, this is because users are likely to be in the age range of 14 - 18 (and those new to coding concepts) and these users may find it much easier to understand basic pseudocode rather than code in a specific language alongside it. Pseudocode also allows for users to carry it over into another programming language if they wish to try code something similar for themselves.

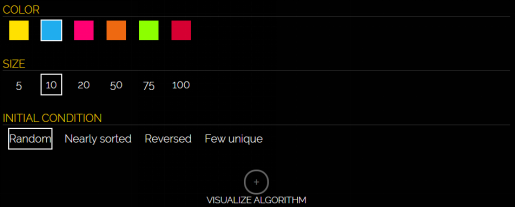
[Sorting.at](http://sorting.at/)



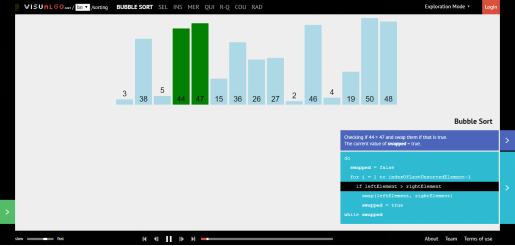


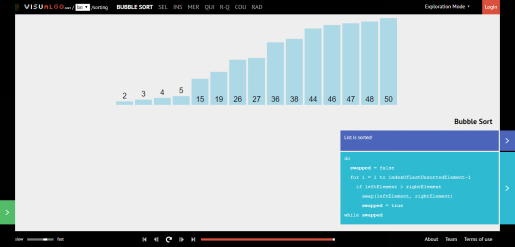






[visualgo.net/bn/sorting](https://visualgo.net/bn/sorting)

****

****

## Investigation

A questionnaire was posted online and shown to clients in order to gather more data on what potential users wanted from the program. The questions and results are as follows:

* Should both sorting and searching algorithms be in the same program?
  + Yes, put them both together (11)
  + No, make a separate program for each (8)
* how much data would you like to be usable in the program?
  + 10 integers (1)
  + 20 integers (4)
  + 30 integers (1)
  + 40 integers (0)
  + 50 integers (4)
  + 100 integers (7)
  + Other (please specify) (1)
    - As much as the user wants
* Please list any sorting/searching algorithms you would like to see/know of. N/A if you know none
  + Pancake sort (1)
  + Stooge sort (1)
  + Comb sort (1)
  + Bubble sort (9)
  + Merge sort (4)
  + Insertion sort (2)
  + Quick sort (1)
  + Bogosort (2)
  + Stalin sort (1)
  + Binary search (4)
  + Linear search (4)
  + Fibonacci search (1)
  + As many as possible (1)
  + N/A (5)
* If you own a desktop/laptop computer, what are it’s specifications? (RAM and clock speed + cores)
  + <2GB RAM and <1.8Ghz quad-core (6)
  + >= 2GM RAM and >1.8Ghz quad-core (10)
  + N/A (1)
  + Other (please specify) (2)
    - 4GB RAM 1.7Ghz dual-core (1)
    - 8GB RAM 1.7Ghz dual-core(1)
* With a visual representation of sorting and searching algorithms using bars, what colour should the bars and background be?
  + Dark grey background and light blue/purple bars (1)
  + Black background and white bars with different colour to highlight current element (1)
  + Black background and white bars (2)
  + Black background and red bars (1)
  + Grey background and red bars (1)
  + Grey background and white bars (1)
  + Anything with good contrast(2)
  + Green bars and white background (1)
  + Blue bars and white background (2)
  + Customisable by user (1)
  + Orange bars and black background(1)
  + N/A (4)
* Should users be able to input their own data or should it be randomised?
  + Yes, users should be able to input their own data (14)
  + No, data should be randomised (4)
  + N/A (1)
* Should pseudocode (simplified version of code) be shown alongside each sort/search?
  + Yes (13)
  + No (5)
  + N/A (1)
* What font and font colour should be used?
  + Roboto, white(2)
  + Ariel, black (2)
  + Comic sans, red (1)
  + Comic sans, black (2)
  + Black colour (1)
  + Carrier, black (1)
  + Jetbrains mono, black (1)
  + Consolas, black (1)
  + User decides (1)
  + Times new roman, black (1)
  + N/A (6)
* How large should integers used in the program be allowed to be?
  + <10 (1)
  + <20
  + <30
  + <40 (1)
  + <50 (3)
  + <100 (11)
  + Other (please specify) (3)
    - >100 (1)
    - <65535 (1)
    - As much as the user wants (1)
* Should results be stored in a text file to be referred to afterwards?
  + Yes (14)
  + No (4)
  + N/A (1)

## Solution Requirements and Success Criteria

The solution requirements and success criteria consist of data collected from the research on existing solutions, the research survey and my own thoughts.

1. To show a list of all searches and sorts available to the user for ease of use.
   1. Font size must be large enough so that all users can read it.
   2. Font must be clear so that all users can read. For example no joke fonts such as Wingdings.
   3. Font, background and bars must be in contrasting colours. For example a black background with white bars (grey when selected) and blue text
2. To allow users to enter their own data if they want to know how a certain data set is processed with each sort, or let it be randomised.
   1. data must be integers - strings cannot be sorted, floats cannot due to restraints of library being used.
   2. data must be positive - negative integers would not work, due to nature of how displaying bars will work.
   3. integers must not be higher than 100 - due to visual restraints.
   4. array must contain more than one item - otherwise it will always be sorted
   5. array must contain no more than 100 items - due to time restrictions, some algorithms can be slow
   6. Duplicate data is allowed
3. To provide a visual representation of the data, this makes it easier for those who don’t understand how sorts/searches work.
   1. bars must be clear to see - can easily see where bars are and their relative size.
   2. animation must appear fluid - can’t be too slow, may not be possible depending on technical specifications of the user’s device.
   3. bars will be white before the sort begins, red when not sorted, green when fully sorted and grey when highlighted.
4. To provide pseudocode for each sort to allow users to replicate each sort/search in their preferred programming language if they wish to.
   1. text of pseudocode must appear alongside the visual representation - to know which sort goes with each code
   2. text must follow some of the same rules as the sort/search list: large and clear. - so users can read it
   3. colour will be orange - to stand out from other text.
5. To store results in a text file for sorts to allow users to look back on previous data.
   1. store steps of each sort in the file - see step by step afterwards what happened
   2. file will be .txt - this keeps file sizes to a minimum, and all devices can read .txt files.

## Computational Methods

The program will be created using a divide and conquer approach. It will be split up into several procedures that will each handle an aspect of the solution. This will allow for decomposition and code reuse as one procedure can be used at several points in the program. For example creating an array will happen a lot during the program depending on how many algorithms the user wants to try. Because of this a procedure will be created to make arrays that can then be called at various stages in the program’s execution. Additionally, merge sort and quick sort utilise the divide and conquer approach themselves.

A core concept of the sorting and searching algorithm world is Big O notation, the algorithms used in the program are tractable as none go above O(n^2). As such the use of a heuristic approach is unnecessary. The algorithms used in this program will take a reasonable time to complete e.g. no O(n!) time complexity.

Another instrumental concept in this solution is visualisation, by the end of program development, users will be able to see in a simple - abstracted (reduced from complex inner workings of algorithm to simple bars sized relatively to each other) - way what happens in the various algorithms. This allows (some) complex procedures to be reduced to a simple array of bars, providing a straightforward view for users. This will be especially helpful for younger users who may be interested in the topic but don’t yet have the knowledge to understand the inner workings of each algorithm.

## System Requirements

* 1.8 GHz or faster processor. Quad-core or better recommended
* 2 GB of RAM; 8 GB of RAM recommended (2.5 GB minimum if running on a virtual machine)

-Taken from <https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements>

→ Actual requirements will be substantially less than this

* A computer using the Windows OS because the file will be .exe so not usable on Linux, Mac or mobile phones

# Design

## Objective

The interface should look clean and be user friendly. The following are the ways in which I intend to achieve the objective, these were obtained partly through the research survey - though are subject to change until the solution is complete.

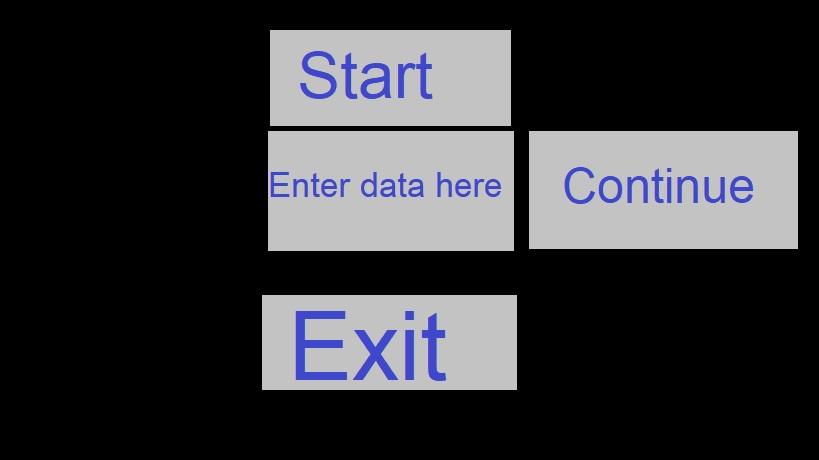
1. Black background so that the screen is not too bright so comfortable for night use
2. White bars to act as a contrast to the background
3. Red bars for when the list is not sorted
4. Green bars for when the whole list is sorted
5. Grey bars for when bar(s) are being selected
6. Grey boxes around menu buttons
7. Yellow (previously blue)text in Comic Sans - Roboto is unavailable in Visual Studio - for menu and more contrast for easy viewing
8. Orange text for pseudocode to separate it from the rest

## Interface Design

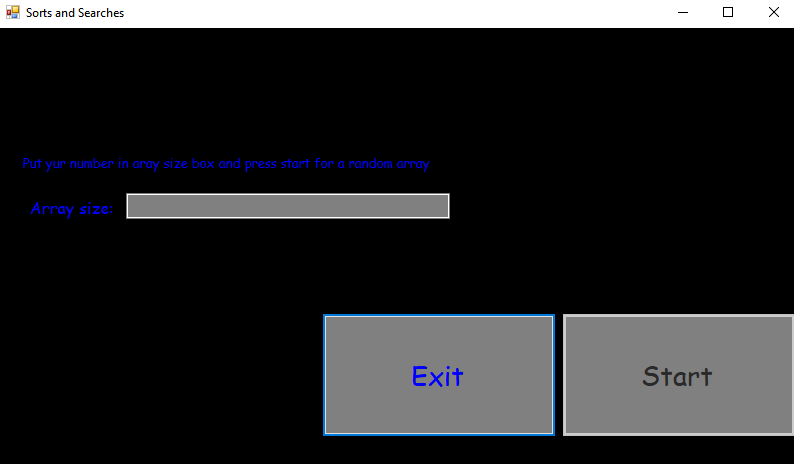
These are some basic designs for the main menu and sort and search screen. These were designed based on the questionnaire sent to stakeholders and other potential users, they were designed with simplicity in mind.

### Main Menu

#### Original design

****

#### Version 1.1

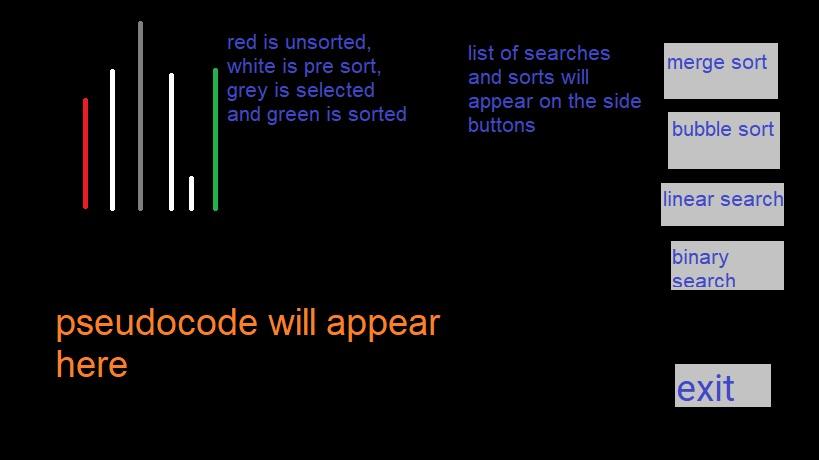
****

#### Version 1.2

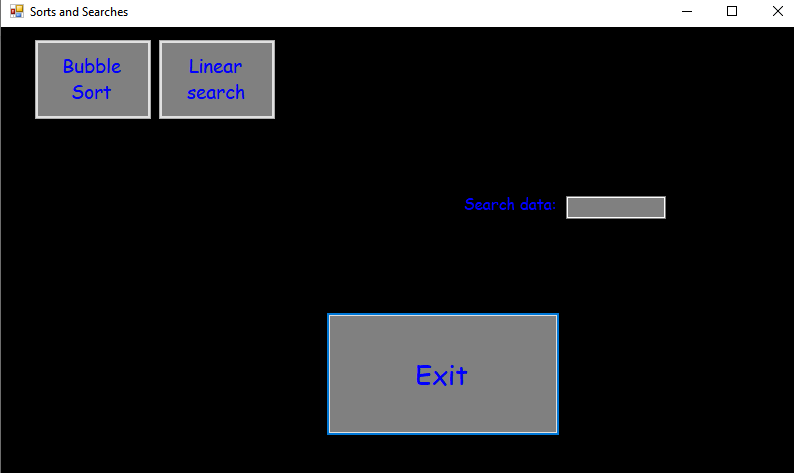
****

### Sorts and searches

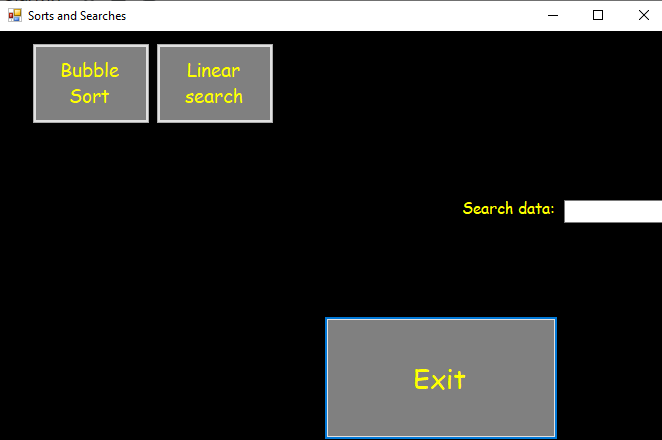
#### Original design



#### Version 1.1

****

#### Version 1.2



## Decomposition and Flowcharts

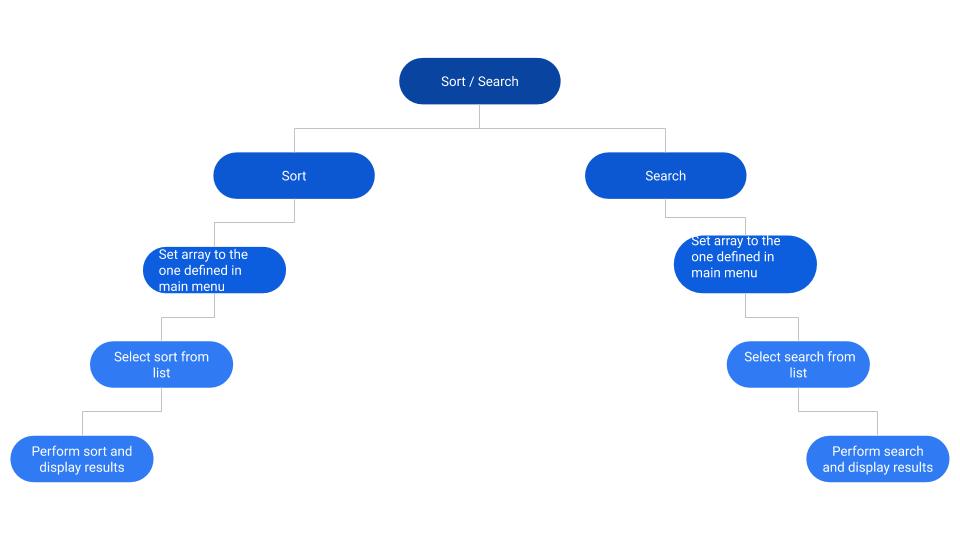
Below are the decompositions and flowcharts of the program. These outline the main features of it in a simple format.

### Main Menu Decomposition

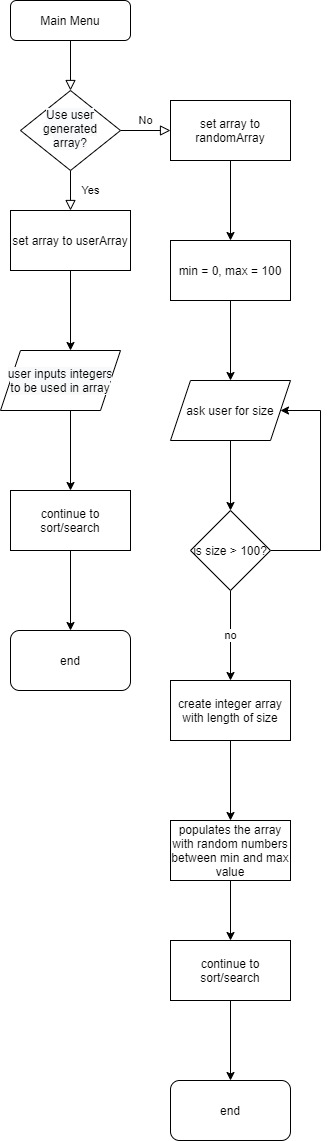


### Sort/Search decomposition

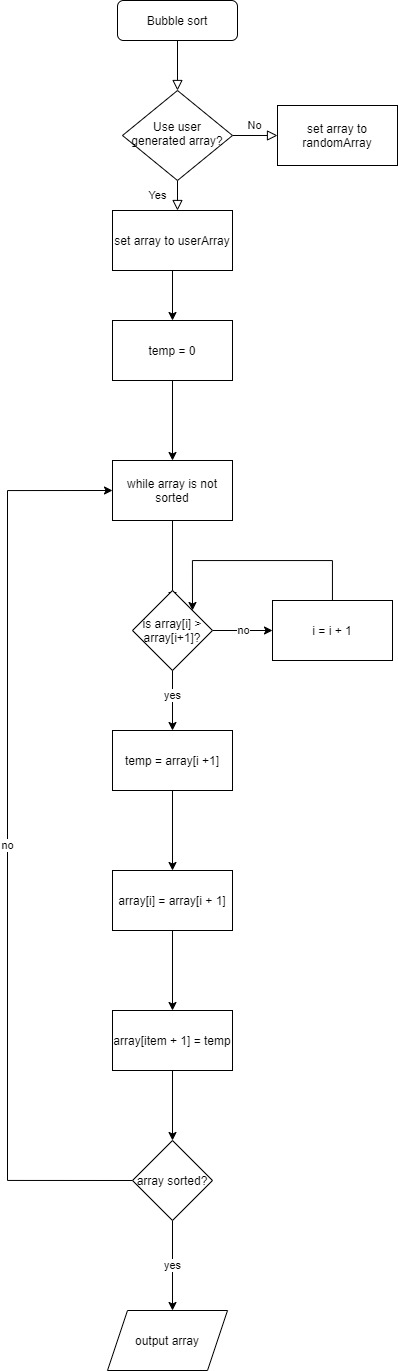
Details of one of each (will be updated with others) sort and search will be displayed through pseudocode.



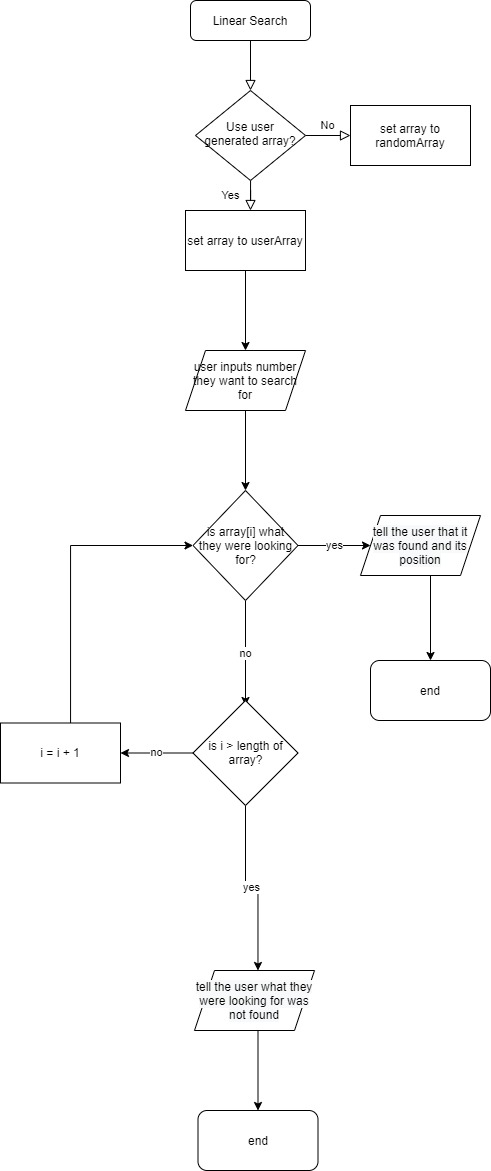
### Main Menu Flowchart

****

### Bubble Sort Flowchart

****

### Linear Search Flowchart



## Data Dictionary

Pseudocode Data Dictionary: - (actual code Data Dictionary will be below it once development of the coded solution has begun, the pseudocode one will act as a framework for the real one)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identifier** | **Description** | **Data type** | **Validation / specification** | **Scope** |
| strRandomOrUser | Will be used to store whether the user wants to choose their own array or have it randomly generated by the program | string | “random” or “user” - case will not matter, program will convert it to lowercase | MainMenu |
| intLength | Stores the length of the randomly generated array | integer | Less than 100 | MainMenu |
| constMin | Stores the lowest number that the random number generator will pick | constant | 0 | MainMenu |
| constMax | Stores the highest number that the random number generator will pick | constant | 100 - due to visual/memory limits | MainMenu |
| randomArray | An array of the random numbers | array | Length is decided by user  Integers only | MainMenu, BubbleSort, LinearSearch |
| strData | Stores the integers picked by the user | string | should only contain integers within the string | MainMenu |
| userArray | Stores the integers picked by the user in an array | array | Integers only | MainMenu, BubbleSort, LinearSearch |
| strSplit | Splits the string of inputs into individual integers using the .split() function | string | It should turn the string into individual integers | MainMenu |
| strContinue | Stores if the user wants to continue with the program or exit | string | "continue" or "exit" - case will not matter, program will convert it to lowercase | MainMenu |
| intTemp | A temporary variable | integer | 0 to begin with. Will always be an integer | BubbleSort |
| strArrayChoice | Stores if the user wants to use random array or user array | string | Y/N case doesn't matter | BubbleSort, LinearSearch |
| array | Stores all the integers to be sorted | array | integers only | BubbleSort, LinearSearch |
| intFind | Stores the number that the user wants to search for | integer | between 0 and 100 | LinearSearch |

## Sample Pseudocode

This section contains some sample pseudocode from each area of the program. These have been written as if to be used in a ‘console application’. However the solution will be a ‘windows forms application’, as such the pseudocode will not strictly follow how the C# code will look, due to the GUI nature of the program rather than a CLI application.

### Main Menu

//Main menu

Start MainMenu

strRandomOrUser = input(“Enter your own array data or use a randomised array. Please enter random or user - case is not important”) //tells the user they have the choice of choosing their own array and using a random one and gets them to choose

If randomOrUser == “random” do //if the user chooses a random array

intLength = input (“How long do you want the array to be? Max = 100 due to visual restraints”) //asks the user how many integers they want in their array

constMin =0 //sets the lowest possible number to 0

constMax = 100 //sets the highest possible number to 100

randomArray[] = int[length] //creates an integer array with length specified by the user

For i in length do //creates a loop with size of array

randomArray[i] = randomInt (min, max) //sets the first number to a random integer between 1 and 100. This will repeat itself until the array is full of randomly generated numbers.

else //if the user chooses to enter their own integers

strData = input (“Enter your integers with a space in between each one. Please enter a maximum of 100 that are up under or equal to 100 in size. If you don’t do this the program will not work.”) //gets the user to input their integers

userArray = int[] //creates an integer array

strSplit = data.split() //splits up the input into individual integers form the input string

For i in split do //for each integer in the splitted list

userArray[j] = i // sets the first number to the first number from the splitted list. This will repeat until all elements of the split are in the array

strContinue = input(“Continue to sort or search or exit? Please enter continue or exit - case does not matter”) //asks the user if they want to continue to the sort/search or exit the program

if strContinue == “continue” do

continue()

else

exit()

### Bubble Sort

//Bubble sort

Start BubbleSort

IntTemp = 0 //creates a variable with an integer value of 0

strArrayChoice = input (“Use random array? Y/N”) //asks the user if they want to use the random array

if arrayChoice == “Y” then //if the user picks yes

array = randomArray //sets the array used in sort to be a random one

else //if they didn’t pick random

array = userArray //uses a user made array instead

for item in array do //creates a loop with size of the array

print(array[item]) //displays each element in the list

for item in array -1 do //creates a loop with size of the array - 1

for item in array -1 do

if array[item] > array[item+1] //if the first number is bigger than the second

temp = array[item+1] //gives the temporary variable the value of the first number

array[item] = array[item+1] //sets the first number to the second number

array[item+1] = temp //sets the second number to the temporary variable - the larger number

//this swaps the numbers e.g. 7,2 will become 2,7

//this will repeat for each number in the list continuously until all numbers are sorted in ascending order

For item in array do

print(array[item]) //displays all the numbers in the list. At this point they should be in order

end

### Linear Search

//Linear search

Start LinearSearch

strArrayChoice = input (“Use random array? Y/N”) //asks the user if they want to use the random array

if strArrayChoice == “Y” then //if the user picks yes

array = randomArray //sets the array used in sort to be a random one

else //if they didn’t pick random

array = userArray //uses a user made array instead

intFind = input (“Search for which number: ”) //asks the user to input a number to search for

For item in array do //creates a loop with size of the array

If array[item] == find //if the current number is the number that is being searched

print (find+” found at position”+item.position) //displays the number and its position

else //if the number was not found

Print (find+” not found in array”) tells the user that it was not found in the array

end

# Development of the coded solution

## Plan for each iteration

### Iteration 1

Iteration 1 should include a random array feature and at least bubble sort and linear search for some basic functionality. If possible it should also have the ability to put in your own array. Some levels of validation should be present. User interface will be basic at this point and may not be the most usable.

### Iteration 2

Iteration 2 should include bubble sort, insertion sort, merge sort, quick sort, linear search and binary search. It should have the option to have either a random array or user generated array implemented adequately. The interface should be easy to use and more validation should be present

### Iteration 3

As the final iteration it should involve all sorts and searches mentioned in the analysis section, if that is not possible then all ones mentioned in the AQA A Level CS specification (those in Iteration 2). The interface should be easy to use and understand for anyone, and sufficient help should be provided by the program if users need it, along with pseudocode accompanying the sort/search process. Add the option to save steps/results to a text file.

All of this should have been achieved (by iteration 3) to meet the needs of users:

1. *To show a list of all searches and sorts available to the user for ease of use.*
   1. *Font size must be large enough so that all users can read it.*
   2. *Font must be clear so that all users can read. For example no joke fonts such as Wingdings.*
   3. *Font, background and bars must be in contrasting colours. For example a black background with white bars (grey when selected) and blue text*
2. *To allow users to enter their own data if they want to know how a certain data set is processed with each sort, or let it be randomised.*
   1. *data must be integers - strings cannot be sorted, floats cannot due to restraints of library being used.*
   2. *data must be positive - negative integers would not work, due to nature of how displaying bars will work.*
   3. *integers must not be higher than 100 - due to visual restraints.*
   4. *array must contain more than one item - otherwise it will always be sorted*
   5. *array must contain no more than 100 items - due to time restrictions, some algorithms can be slow*
   6. *Duplicate data is allowed*
3. *To provide a visual representation of the data, this makes it easier for those who don’t understand how sorts/searches work.*
   1. *bars must be clear to see - can easily see where bars are and their relative size.*
   2. *animation must appear fluid - can’t be too slow, may not be possible depending on technical specifications of the user’s device.*
   3. *bars will be white before the sort begins, red when not sorted, green when fully sorted and grey when highlighted.*
4. *To provide pseudocode for each sort to allow users to replicate each sort/search in their preferred programming language if they wish to.*
   1. *text of pseudocode must appear alongside the visual representation - to know which sort goes with each code*
   2. *text must follow some of the same rules as the sort/search list: large and clear. - so users can read it*
   3. *colour will be orange - to stand out from other text.*
5. *To store results in a text file for sorts to allow users to look back on previous data.*
   1. *store steps of each sort in the file - see step by step afterwards what happened*
   2. *file will be .txt - this keeps file sizes to a minimum, and all devices can read .txt files.*

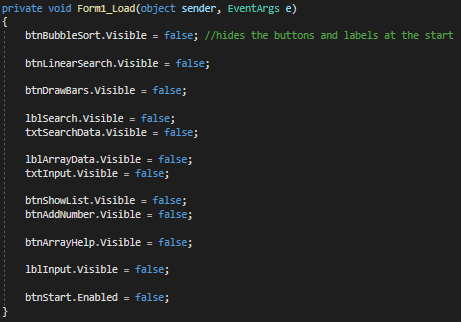
## Main Menu

The purpose of the main menu is to allow the user to input their own array if they choose to or to have a random one generated for them. They must always supply a length of the array (as of iteration 1.1)

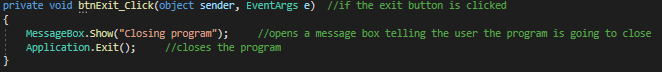
### Implementation in iteration 1.1

Includes code not currently implemented or functioning commented out

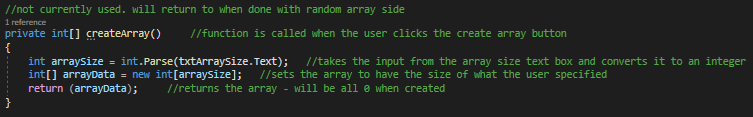
#### Form Initialisation



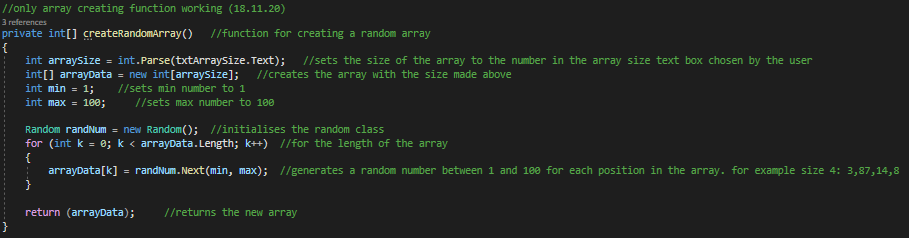
#### Exit



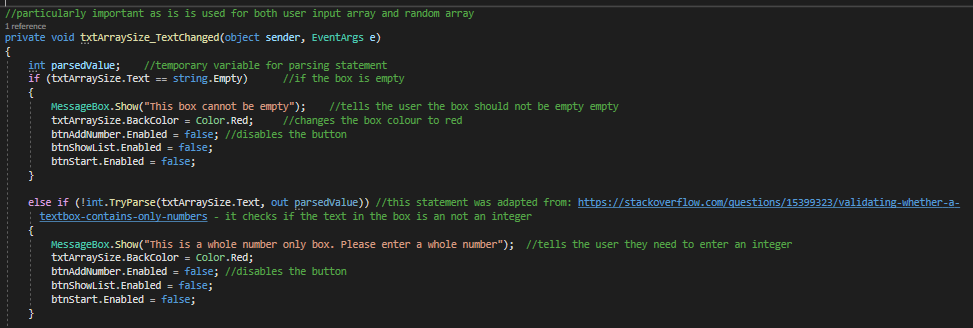
#### Creating user array

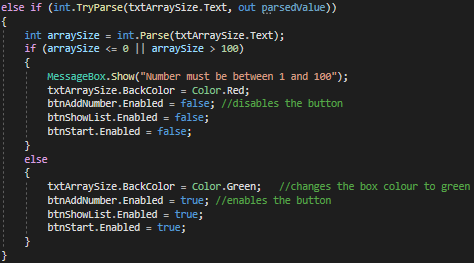


#### Creating random array

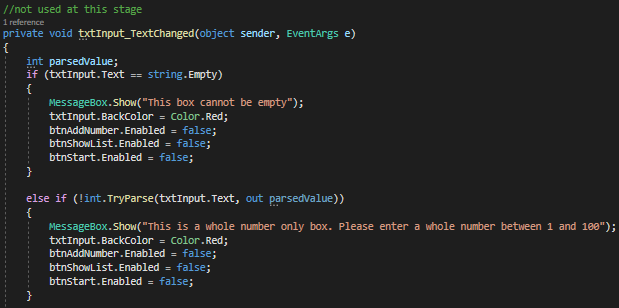


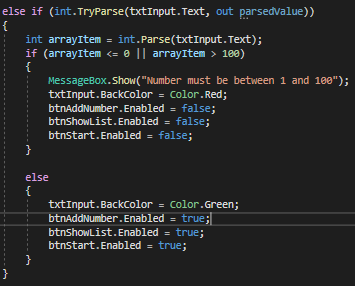
#### Array size text box validation





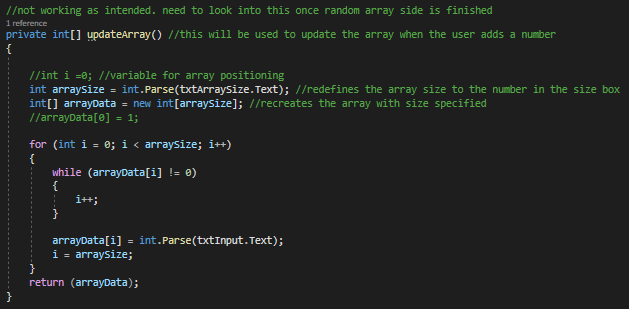
#### Array data text box validation



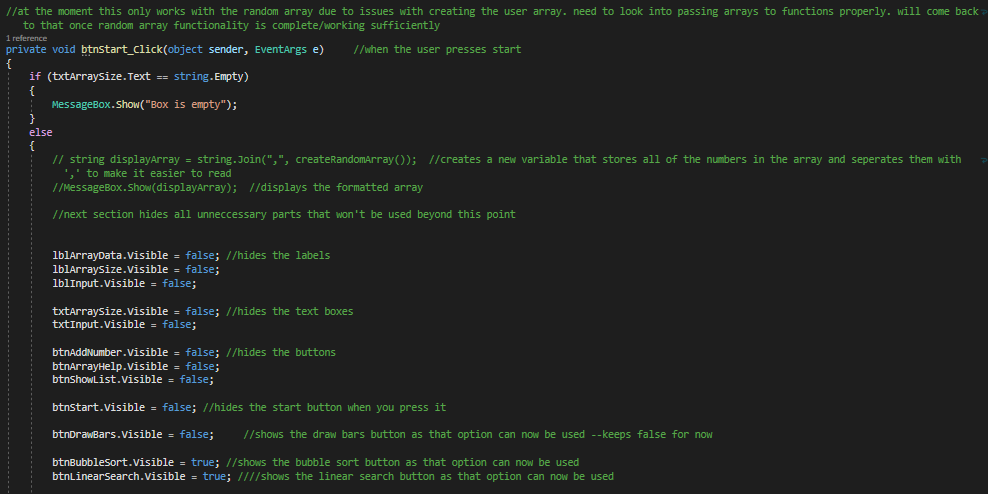


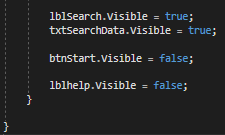
#### Add number button with validation

#### Update array button

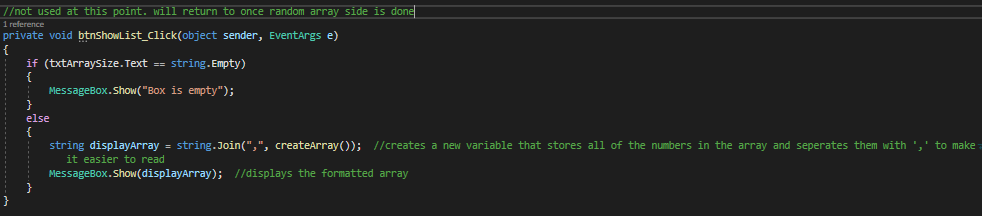


#### Start button with validation

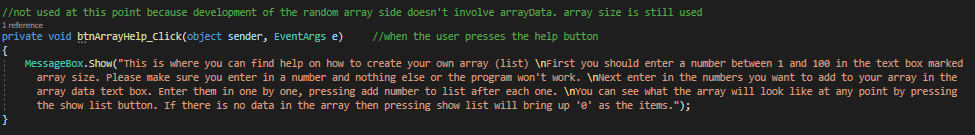




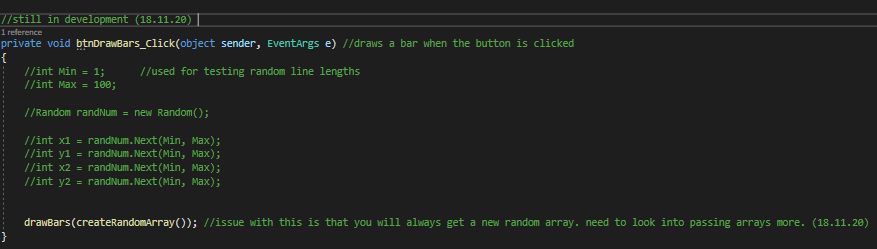
#### Show list button with validation



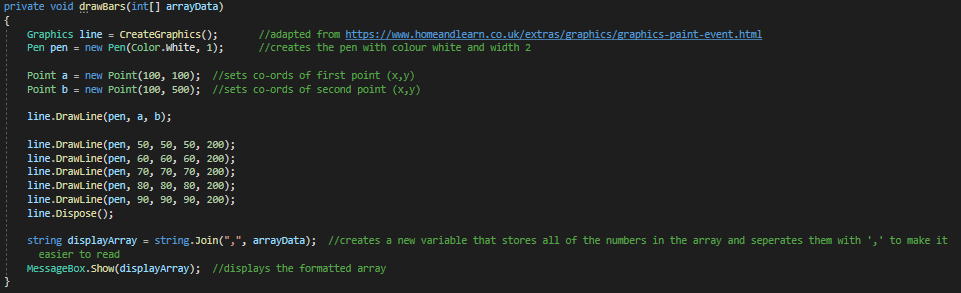
#### Array creation help button



#### Draw bars button

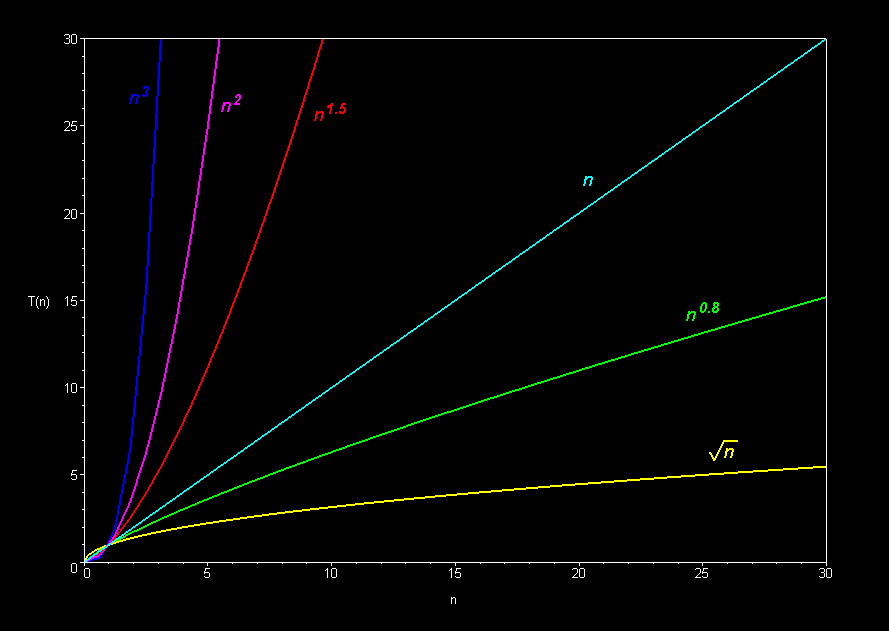


#### Draw bars function

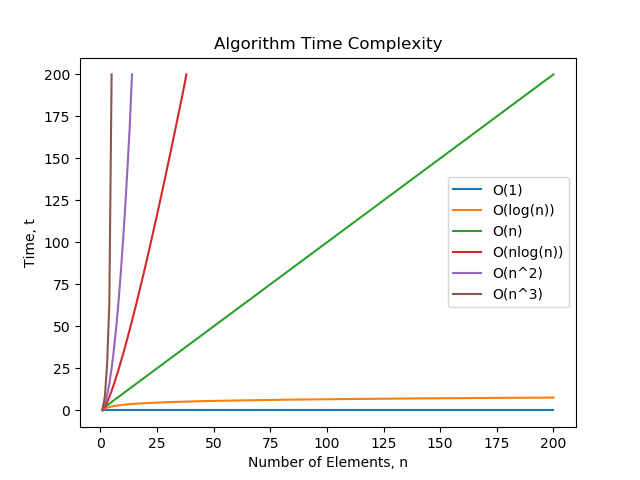


## 

## Run-time graphs



* <http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html>



* <https://blog.mbedded.ninja/programming/algorithms-and-data-structures/algorithm-time-complexity/>

## Sorts

### Bubble Sort

#### Information

The Bubble sort is an extremely simple comparison quadratic sorting algorithm which works by comparing two adjacent values [n] and [n+1] and swaps them if they are in the wrong order. This is repeated until every item in the array is in its correct position.

Its simplicity makes it very useful for educating students as an introduction to sorting algorithms but that is where its use ends. Bubble sort is very slow when working with large data sets.

This is due to its worst case and average time complexity (expressed with Big O notation) of О(n^2), where n = the number of items being sorted. This means the time it takes on average to complete the algorithm goes up with the square of the amount of items in the data set.

For example the time it would take to perform a bubble sort on 10 items would be 10^2 so 100. Whereas with 50 items it would be 50^2 so 2500, 25 times longer. The difference in time becomes massive the larger your data set is, the Bubble sort is already quite inefficient so each step takes longer to begin with.

Luckily (in the case of this program) the difference in time will not be too noticeable. This is due to the restrictions placed on the size of the array. However users will still be able to see the inefficiencies in the sort itself.

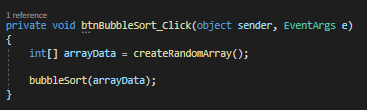
~~Sorting 1,000,000 integers would take 15,625x more time than sorting 8,000. Sorting 8,000 takes about one second on average (on my laptop, with an average over 1000 runs) so 1,000,000 would take 15,625 seconds to sort. That’s equivalent to 4.34 hours if workings out are correct. Which would take 4,340 hours (about 25 weeks) to get 1000 runs to work out its average. Luckily we already have its Big O notation so we wouldn't have to do all of this manually now.~~

Link to time complexity research sheet [here](https://docs.google.com/spreadsheets/d/1AVD-SZpJ_Gq0hwVAwH5pO3TF520FG0tLugvahSdoepM/edit?usp=sharing)

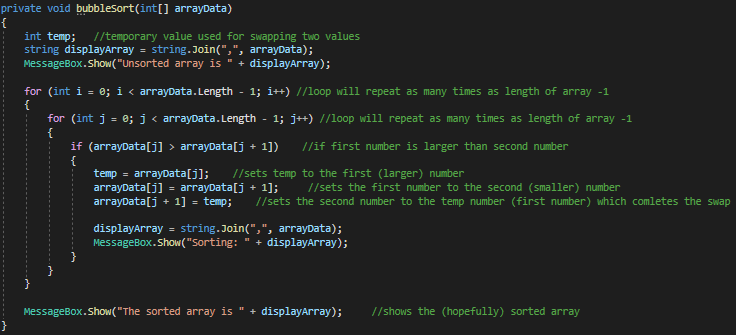
#### Implementation in Iteration 1

Bubble sort was the easiest sort to implement due to its simplicity. No issues.

##### Button



##### Function



### Insertion Sort

#### Information

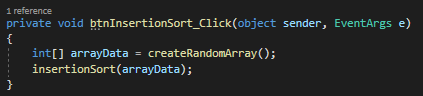
Insertion sort is another simple comparison quadratic sorting algorithm. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

It shares the same weaknesses and strengths of bubble sort, it is slow on large lists/arrays, with its worst case and average time complexity of О(n^2). It is still relatively quick on small arrays. It may share the same time complexity with bubble sort on average but it is a more efficient algorithm, therefore it completes its steps quicker, leading to it being faster than bubble sort but still heavily outclasses by sorts like merge sort and quicksort.

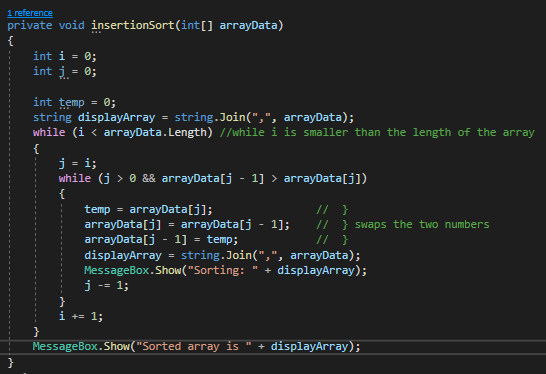
Variations of insertion sort include the shell sort which has variations of itself. These are generally more efficient but their time complexity depends on the gap used when comparing values.

#### Implementation in iteration 2

##### Button



##### Function



### Merge Sort

#### Information

Merge sort is an efficient comparison sorting algorithm that words by process of ‘divide and conquer’. It splits the array into n (number of items in array) sublists. So each sublist contains one item to begin with. It then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

This scales well to very large lists/arrays, because its worst-case and average time complexity is O(n log n).

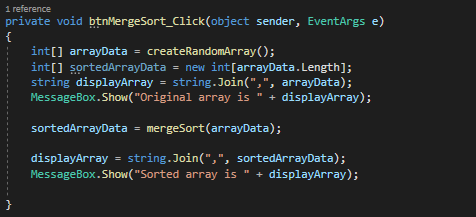
#### Implementation in iteration 2

Merge sort relies on using two functions:

1. Recursively splitting the array
2. Sort and merging the arrays into one

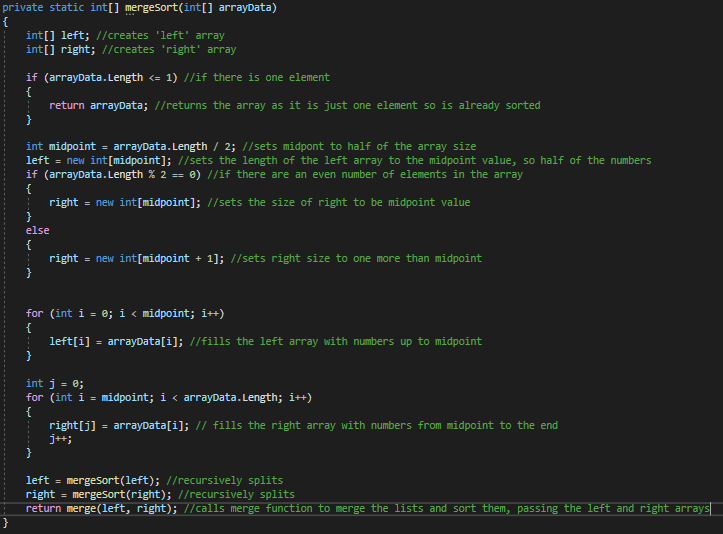
~~Because of this at the moment it is not possible to show the steps of the sort~~ (resolved 08/12/2020). So unlike the previous sorts, the output is displayed through the button rather than the sort function. Despite it being a more complex algorithm it was fairly simple to implement.

##### Button

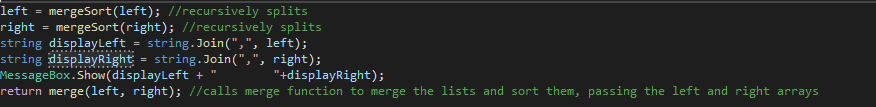


##### Functions

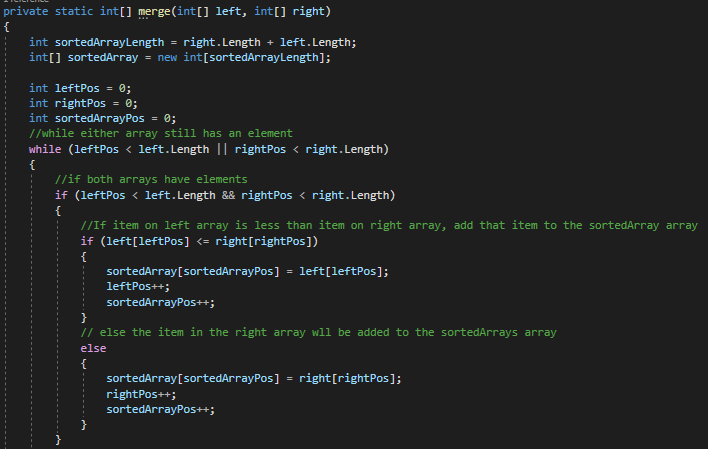
###### Splitting

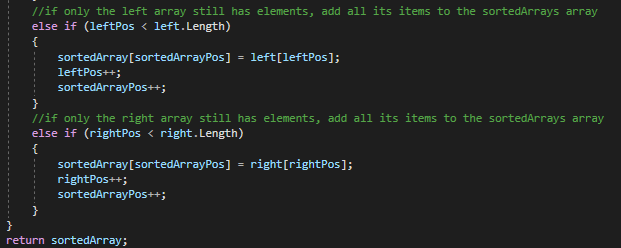


###### Splitting with showing steps - shows changed code

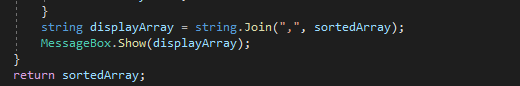


###### Merging





###### Merging with showing steps - shows changed code



### Quick Sort

#### Information

Quicksort is another divide and conquer algorithm, however this one relies on a partition operation. To partition an array, an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. The lesser and greater sublists are then recursively sorted.

This yields an average time complexity of O(n log n) when a favourable partition is chosen. However its worst-case is O(n^2). This is usually when the partition is the end or start of the list, otherwise it is very rare. In practice choosing a random pivot almost certainly yields O(n log n) performance.

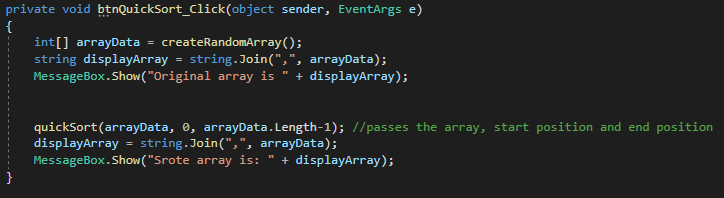
Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.

There are faster sorting algorithms that do not rely on comparison such as radix sort, these algorithms are quite complex ~~- quite outside my range of understanding~~. They are also not required to be learnt for the AQA CS A-level or GCSE specification. Because of this they will not be covered in this program because users (if they are in high school or college) will not need to know them. They will only be included if there is sufficient time before project submission which is unlikely.

#### Implementation in iteration 2

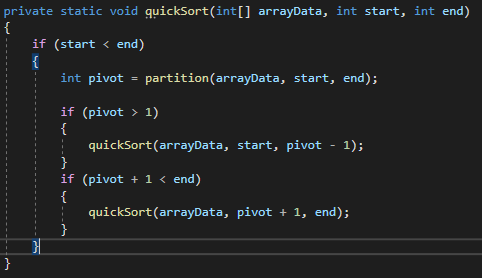
Many implementations of quick sort use a pivot of the first or last element of the array. Whilst this is simple and does function, it will likely lead to the time complexity of O(n^2). This is the worst case scenario, as such a random pivot will be chosen to hopefully reach the average time complexity of O(n log n) - a much more favourable outcome as it will scale better with larger lists. This will use Lomuto's partition Scheme (probably).

##### Button

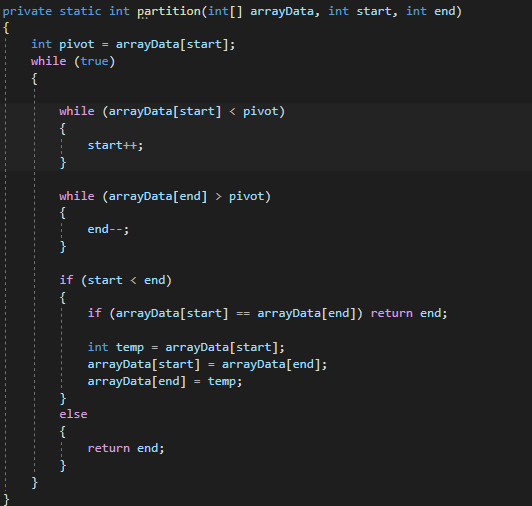


##### Functions

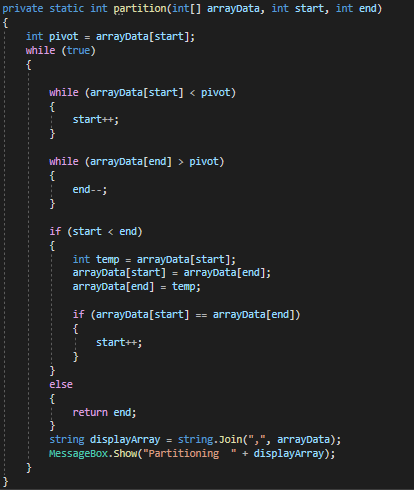
###### Quick Sort



###### Partition



###### Partition allowing for duplicate numbers



## Searches

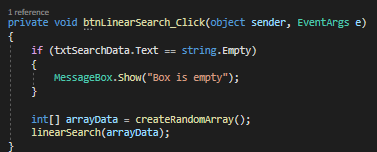
### Linear Search

#### Information

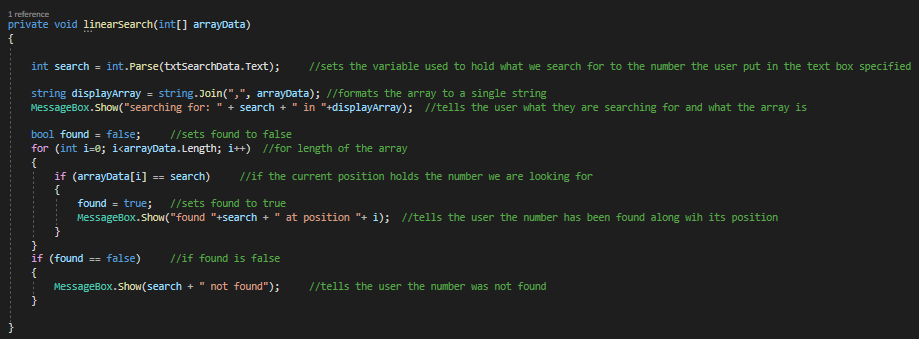
Linear search is a simple search algorithm that sequentially checks each element in a list until a match is found. It is effective on short lists but is much weaker when compared to binary search. Its average time complexity is O(n/2) but worst is O(n). A benefit of linear search is that, unlike binary search, the list does not have to be sorted beforehand

#### Implementation in iteration 1

##### Button



##### Function



### Binary Search

#### Information

Binary search is an algorithm that works by splitting a list and comparing its middle element to the target value. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. Its average and worst case time complexity is O(log n). Binary search performs well with large lists and there are very few downsides to it. If you are searching through a very small list then linear search would be better and the list must be sorted. But with the use of an efficient sorting algorithm, the time spent is still minimal.

#### Implementation in iteration 2

-

##### Button

-

##### Function

-

## Testing

Test key:

1.2

Where 1 is the test number and 2 is any **reiteration** of the test e.g. 1.3 is test 1, iteration 3 (not program iteration)

### Testing Table

Testing table is [here](https://docs.google.com/spreadsheets/d/1UXuMe1nUW32QiIM5kEr3nzeHhPYz2ZhCCTlATzgOA2w/edit?usp=sharing)

# Evaluation

## Meeting the requirements

### Results of each iteration

#### Iteration 1

Iteration 1 did not go entirely as planned. This was mostly due to issues with creating graphics and with passing the array into functions. Users could not fully create their own array at this stage. Because of this those options are hidden but remain in code until fixed. However a random array could successfully be produced. And a linear search or bubble sort can be shown via text on that array.The issues mentioned should be resolved by iteration 2.

#### Iteration 2

Iteration 2 included an overhaul of the validation system in the main menu. Rather than each text box having its own personalised procedure that it would follow, it simply called the validate function which utilised objects. Due to this code is now a lot more clean.

Merge sort was also added, along with its steps.

#### Iteration 3

### Feedback from users

#### Iteration 1

The user interface had some issues with it which were mostly to do with the colour choices. “Blue on black is not easy on the eyes.” Because of this the general text colour will be yellow to add more of a contrast.

The text box background colour was grey, when trying to click in the box using the mouse, the cursor disappeared, so the background colour of text boxes will be changed to white. The background colour of buttons will remain grey as they do not suffer from the same issue.

A major issue with the program is that when sorting through large amounts of data, the program takes a while to complete and cannot be closed without the use of Task Manager. This is because the only available sort (currently) is bubble sort which is very slow when being used on a large data set. To resolve this there will be a button or checkbox to disable/enable showing steps to prevent this

#### Iteration 2

-

#### Iteration 3

-

## Limitations

-

## Improvements

-

# Bibliography

Author Surname/Corporate name, Initial(s) Year (page revised/created), Title of page, viewed Day Month Year, <URL>.

Microsoft, 2019, Visual Studio 2019 System Requirements | Microsoft Docs, 04/05/2020, <<https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements>>

Physics & Maths Tutor, 2015, Advanced Notes - 2.2.2 Computational Methods - OCR Computer Science A-level, 30/11/2020, <<https://pmt.physicsandmathstutor.com/download/Computer-Science/A-level/Notes/OCR/2.2-Problem-Solving-and-Programming/Advanced/2.2.2.%20Computational%20Methods.pdf>>

Stack Overflow, 2019, c# - Validating whether a textbox contains only numbers, 28/10/2020, <<https://stackoverflow.com/questions/15399323/validating-whether-a-textbox-contains-only-numbers>>

Home and Learn, (no date), The .NET Paint Event, 18/11/2020, <<https://www.homeandlearn.co.uk/extras/graphics/graphics-paint-event.html>>

mbedded.ninja, 2019, Algorithm Time Complexity | mbedded.ninja, 25/11/20, <<https://blog.mbedded.ninja/programming/algorithms-and-data-structures/algorithm-time-complexity/>>

Wikipedia, 2020, Bubble sort, 01/11/2020, <<https://en.wikipedia.org/wiki/Bubble_sort>>

Wikipedia, 2020, Insertion sort, 01/11/2020, <<https://en.wikipedia.org/wiki/Insertion_sort>>

Wikipedia, 2020, Shellsort, 29/11/2020, <<https://en.wikipedia.org/wiki/Shellsort>>

Wikipedia, 2020, Merge sort, 29/11/2020, <<https://en.wikipedia.org/wiki/Merge_sort>>

Wikipedia, 2020, Quicksort, 30/11/2020, <<https://en.wikipedia.org/wiki/Quicksort>>

Wikipedia, 2020, Radix sort, 30/11/2020, <<https://en.wikipedia.org/wiki/Radix_sort>>

Wikipedia, 2020, Linear search, 01/11/2020, <<https://en.wikipedia.org/wiki/Linear_search>>

Wikipedia, 2020, Binary search, 30/11/2020, <<https://en.wikipedia.org/wiki/Binary_search_algorithm>>

Wikipedia, 2020, Big O Notation, 25/11/2020, <<https://en.wikipedia.org/wiki/Big_O_notation>>

Marshall, J., 2002, Running Time Graphs, 25/11/2020, <[http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.htm](http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html)l>

Wikipedia, 2020, Best, worst and average case, 25/11/2020, <<https://en.wikipedia.org/wiki/Best,_worst_and_average_case>>

Wikipedia, 2020, Time complexity, 25/11/2020, <<https://en.wikipedia.org/wiki/Time_complexity>>

GeeksforGeeks, 2020, Hoare’s vs Lomuto partition scheme in QuickSort, 25/11/2020, <<https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/>>

w3resource, 2020, C# Sharp Exercises: Quick sort, 9/12/2020, <[https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-9.php](https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-9.ph)>

# Links - to be removed before submission

[Trello](https://trello.com/b/6UfNTG7G/a2-coursework-project)

[Testing Sheet](https://docs.google.com/spreadsheets/d/1UXuMe1nUW32QiIM5kEr3nzeHhPYz2ZhCCTlATzgOA2w/edit?usp=drive_web&ouid=100034094965736566380)

[Testing Screenshots](https://docs.google.com/document/d/1PLmgdo8eaHeSJ5BAbnN7PVw2SCbmGIvyifH4Pe0HlFQ/edit)

[User Feedback](https://docs.google.com/spreadsheets/d/1ZcVQazH55Hb_fP14NvCEYUGx3x8dOfkBe3oryotnMws/edit#gid=0)

[Time Complexity Sheet](https://docs.google.com/spreadsheets/d/1AVD-SZpJ_Gq0hwVAwH5pO3TF520FG0tLugvahSdoepM/edit)

[Project Handbook](https://docs.google.com/document/d/1_8YPu71e3Le8tcWchRzLSo5U3SybZl8w/edit?rtpof=true)

[Project Mark Scheme](https://learn-eu-central-1-prod-fleet01-xythos.content.blackboardcdn.com/5d3f01e8dc9d8/1043542?X-Blackboard-Expiration=1607612400000&X-Blackboard-Signature=B2Gvz20zssmT3elQ7kdGbaeEn7d5baBzu59rHLY4k54%3D&X-Blackboard-Client-Id=177359&response-cache-control=private%2C%20max-age%3D21600&response-content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27MarkingsheetFullSyllabus%25282%2529.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20201210T090000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAZH6WM4PL5M5HI5WH%2F20201210%2Feu-central-1%2Fs3%2Faws4_request&X-Amz-Signature=3ea2878f20c1eaee6af623caaec00ec85b61b17d7824fe755db2b3d4ee7c78a8)