Lecturer: Dominic Carr

Computational Thinking With Algorithms

Student: Richard Deegan G00387896

Contents

[Preliminaries 2](#_Toc71897974)

[Introduction 3](#_Toc71897975)

[Sorting Algorithms 3](#_Toc71897976)

[Bubble Sort 3](#_Toc71897977)

[Introduction 3](#_Toc71897978)

[Space and time complexity 3](#_Toc71897979)

[How Bubble Sort works with original diagram 3](#_Toc71897980)

[Merge Sort 3](#_Toc71897981)

[Introduction 3](#_Toc71897982)

[Space and time complexity 3](#_Toc71897983)

[How Merge Sort works with original diagram 3](#_Toc71897984)

[Counting Sort 3](#_Toc71897985)

[Introduction 3](#_Toc71897986)

[Space and time complexity 3](#_Toc71897987)

[How Counting Sort works with original diagram 3](#_Toc71897988)

[Selection Sort 3](#_Toc71897989)

[Introduction 3](#_Toc71897990)

[Space and time complexity 4](#_Toc71897991)

[How Selection Sort works with original diagram 4](#_Toc71897992)

[Insertion Sort 4](#_Toc71897993)

[Introduction 4](#_Toc71897994)

[Space and time complexity 4](#_Toc71897995)

[How Insertion Sort works with original diagram 4](#_Toc71897996)

[Implementation and Benchmarking 4](#_Toc71897997)

[Results of the benchmarking 4](#_Toc71897998)

[Summary of Results- Graph and Table 4](#_Toc71897999)

# Preliminaries

For this project you will write an application which will be used to benchmark five different sorting algorithms. You will also write a report which introduces the algorithms you have chosen and discusses the results of the benchmarking process. The five sorting algorithms must be chosen according to the following criteria:

1. A simple comparison-based sort (Bubble Sort, Selection Sort or Insertion Sort)

2. An efficient comparison-based sort (Merge Sort, Quicksort or Heap Sort)

3. A non-comparison sort (Counting Sort, Bucket Sort or Radix Sort)

4. Any other sorting algorithm of your choice

5. Any other sorting algorithm of your choice.

For the purpose of this assignment brief the following Algorithms were chosen:

* Bubble Sort
* Merge Sort
* Counting Sort
* Selection
* Insertion

# Introduction

Algorithms are a set of processes or rules that must be followed in order to solve problems. A common analogy for a computer algorithm is that of a food recipe. They are similar in that they are both a set of instructions that must be followed to get a desired outcome. Many different algorithms can be created to achieve the same result, just like recipes, in this instance the flavour and taste may be different where in the case of algorithms, maybe the run time will be different. In general, there are seven key features that form the basis of well-designed algorithms:

1. Input: Input data is well defined when beginning the algorithms operations
2. Output: Output data is well defined when the algorithm has finished.
3. Finiteness: An algorithm must always end unless it will run forever.
4. Unambiguous: Each step/set of code must be well defined and logical in each case.
5. Correctness: The algorithm must arrive at a correct solution, within error rates.
6. Feasibility: It should be feasible to run the algorithm considering resources.
7. Efficiency: The algorithm must be as quick as possible given the size of the task.

# Time and Space Complexity

Considering the features above, two key concepts emerge when considering algorithms; that is space and time complexity. Space complexity refers to the amount of space or memory it takes an algorithm to run its function. It includes both auxiliary space and input space. Space complexity S(p) of any algorithm p is S(p) = A + Sp(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I. The example below illustrates the concept.

A picture containing graphical user interface

Description automatically generated

Here we have three variables R,P, Q and one constant. S(p) is equal to 1 + 3. Now space is dependent on the data types of the given constant types and variables, this will be multiplied accordingly. [1].

Time complexity is the amount of time it takes an algorithm to complete its task as a function of the input length. The time complexity of algorithms is usually expressed using notation such as Big O, Big Theta, or Big Omega. Essentially we are considering how well the algorithm deals with the scaling of numbers, as the input data size increases its important to know how well an algorithm will scale accordingly. For a programmer, its highly valuable to know the effect of space and time complexity on algorithms, this will enable the programmer to solve problems in the most efficient way possible.

# Big O Notation

Describing the run time of an algorithm is a complex task and all efforts are focused on finding a function that best approximates the runtime. Asymptotic notations are used to describe the run time and have roots from Number Theory []. Big O notation is used to describe asymptotic behaviours of functions. The growth rate of the functions is called “order”, and it is order of magnitudes in which concerns the programmer.

**Big-oh (O):** Let f(n) and g(n) be functions such that: - ∃k > 0 ∃n0 ∀n > n0 |f(n)| ≤ |g(n) · k| then f(n) ∈ O(g(n)) or with some abuse of notation f(n) = O(g(n)). [Krueger et al., 1999] essentially Big-Oh represents the upper limit of resources required for the algorithm to run, that is the maximum amount of time and space. It describes the algorithm in the worst-case scenario, in essence it describes how a function grows slower or at least as slow as another.

**Big Omega(Ω):** Omega is used to describe the complexity of an algorithm in the best case. Essentially it represents the lower limit of resources required for the algorithm to run. This is the best-case scenario and exhibits linear growth in execution time as n is increased. []

**Big-Theta (Θ):** Let f(n) and g(n) be functions such that :- ∃k1 > 0 ∃k2 > 0 ∃n0 ∀n > n0 g(n) · k1 ≤ f(n) ≤ g(n) · k2 then f(n) ∈ Θ(g(n)) or with abuse of notation f(n) = Θ(g(n)).[] Essentially the theta notation is utilised to specify the runtime of an algorithm is no greater or less than a certain “order”. It describes the complexities of the algorithm in the average case by representing both big-oh (O) and Big Omega(Ω) expressions.

The following is a list of common types of orders and their names:

Graphical user interface, text, application

Description automatically generated

[Danziger, P., 2010]

Common Sorting algorithms include the following, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort, Radix Sort, and Bucket Sort. Sorting algorithms.

# Performance, Comparator functions and Time Complexity.

There are two main key features when determining the performance an algorithm.

1. Input Size: The size of the data (n) which is inputted into the algorithm must be considered, or the number of data items in a single instance.
2. Operation count: The number of operations f(n) required by the algorithm, or its running time.

# Diagram, schematic Description automatically generated

Adrean mejia (2020).

Considering the above diagram, if we use statements with basic operations such as comparisons, assignments, or reading a variable, we can therefore assume that they take a constant time. That is 0(1). As long as there is a fixed number of operations we can say that time will be constant, even if we have multiple statements.

Conditional statements are critical for coding algorithms, and calculating the time complexity considering the worst case or big -oh, we must take the worst case performance. For instance, an “if” block of code takes 0(n log n) time complexity, while the associated “else” part of the code takes a runtime of O(1), in this case 0(n log n) is used as it is the worst case.

Loop statements are determined by figuring out the runtime of the block inside the loop and multiplying it by the number of times the program repeats the loop. All loops that grow in proportional to the input size will have a linear time complexity, 0(n). If the array is looped only half the way then its still 0(n). Constant time loops are different in that the runtime is constant , that is if the code is not dependent on the input size it will always run x times and therefore possess a time complexity of 0(n).

For logarithmic time loops where an array is split in half for each iteration. The while loop will execute for the number of times we can split the array length in half. This is calculated by suing the log function. For example, of the array length is 8, then we can say log2(8) = 3.

Nested loops are highly inefficient as the algorithm must loop over each element within the nested loop as well as the original loop, the expected time complexity would be O(n^2).

The time complexity of functions depends on the runtime of the function. If the run time of say 3 functions are all constant at O(1) then the final runtime would be O(n^3).

For recursive functions, it’s possible to visualise each function invocation as a node. Figure xxxxxx helps illustrates this in greater detail. The total number of calls in a complete node tree is 2^n -1. Fn(4) is not complete and the last level has only two nodes, that is Fn(1) and Fn(0), while a complete node tree would have eight nodes. For this instance, the run time is exponential O(2^n), however it wont get any worse as the upper limit is 2^n.

Diagram

Description automatically generated

# Stable and Unstable algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. On the other hand, in an unstable algorithm, straw or spork may be interchanged e.g selection sort, quick sort, heap sort. [] In essence, sorting stability ensures records with the same key keep the relative order, before and after the sort. This may be important depending on the problem you are trying to solve. However, if the retention of that relative order is not necessary then an unstable algorithm may be used, in this case it will free up some memory space.

Diagram, table

Description automatically generated

In place sorting: this type of sorting algorithm uses a fixed additional amount of working space that is independent of the input size. Insertion sort and bubble sort would be examples of this. In other words, these algorithm uses constant extra space for producing the output. It sorts the list only by modifying the order of the elements within the list.

https://www.geeksforgeeks.org/sorting-terminology/

Ideal sorting algorithm.

The ideal sorting algorithm would have the following (news.ycombiator, 2016):

* Stable: Equal keys aren’t reordered.
* Operates in place, requiring O(1) extra space.
* Worst-case O(n·lg(n)) key comparisons.
* Worst-case O(n) swaps.
* Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

# In place sorting

# Comparison and non-comparison based.

Introduce the concept of sorting and sorting algorithms, discuss the relevance of concepts such as complexity (time and space), performance, in-place sorting, stable sorting, comparator functions, comparison-based and non-comparison-based sorts, etc.

Time efficiency considers the time or number of operations required for the computer takes to run a program (or algorithm in our case) • Space efficiency considers the amount of memory or storage the computer needs to run a program/algorithm

# Sorting Algorithms

## Bubble Sort – Comparison Based Algorithm

### Introduction

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs [7].

A picture containing text, electronics, keyboard

Description automatically generated

Figure xxxx illustrates this in practice. The shaded numbers are compared, hence bubble sort is a comparison based algorithm.  If there are n items in the list, then there are n−1n−1 pairs of items that need to be compared on the first pass. At the start of the second pass, the largest number is now in the correct place. There are now n-1 items in the array to sort out, or in other words, there will be at least n – 2 pairs. The exchange operation is referred to as “swap”, in Python is possible to perform a simultaneous assignment with a statement such as “a, b=b, a” will result in two assignments being done at the same time.

Diagram

Description automatically generated

### Space and time complexity

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These “wasted” exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop [8].

The **time complexity** of Bubble Sort is **O(n2)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best ase time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

* Worst Case Time Complexity [ Big-O ]: **O(n2)**
* Best Case Time Complexity [Big-omega]: **O(n)**
* Average Time Complexity [Big-theta]: **O(n2)**
* Space Complexity: **O(1)**

### How Bubble Sort works with original diagram

The Python code used to implement Bubble Sort is shown in below. Essentially two loops exist, the outer loop runs in increments of one from the second last array item, while the inner loop runs from the first array item in the list to the items in the outer list being ran. The output of the program is the following: [5,15, 18, 22, 22, 55, 66, 91].

Text

Description automatically generated

## Merge Sort

### Introduction

### Space and time complexity

Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n\*log n).

* Worst Case Time Complexity [ Big-O ]: O(n\*log n)
* Best Case Time Complexity [Big-omega]: O(n\*log n)
* Average Time Complexity [Big-theta]: O(n\*log n)
* Space Complexity: O(n)

Time complexity of Merge Sort is O(n\*Log n) in all the 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves.

It requires equal amount of additional space as the unsorted array. Hence its not at all recommended for searching large unsorted arrays [8].

### How Merge Sort works with original diagram

## Counting Sort

### Introduction

### Space and time complexity

### How Counting Sort works with original diagram

## Selection Sort

### Introduction

### Space and time complexity

### How Selection Sort works with original diagram

## Insertion Sort

### Introduction

### Space and time complexity

### How Insertion Sort works with original diagram

# Implementation and Benchmarking

Description of the process when implementing the application

## Results of the benchmarking

Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm? In this section you should use both a table and a graph to summarise the results

## Summary of Results- Graph and Table

# References

[1]. TutorialsPoint.com <https://www.tutorialspoint.com/time-and-space-complexity-in-data-structure>

[2] Danziger, P., 2010. Big o notation. *Source internet: http://www. scs. ryerson. ca/mth110/Handouts/PD/bigO. pdf, Retrieve: April*.

[3]Krueger, R., Rudnicki, P. and Shelley, P., 1999. Asymptotic notation. Part II: Examples and problems. Journal of Formalized Mathematics, 11(199), p.9.

[4]Introduction to Asymptotic Notations Big O (O), Big Omega (Ω) and Big Theta (Ө) <https://madhawacperera.medium.com/introduction-to-asymptotic-notations-big-o-o-big-omega-%CF%89-and-big-theta-%D3%A9-cb84d0d1caf7>

[5]Adrean mejia (2020). How to find time complexity of an algorithm?

<https://adrianmejia.com/how-to-find-time-complexity-of-an-algorithm-code-big-o-notation/>

[6]stackoverflow(2021).https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important#:~:text=A%20sorting%20algorithm%20is%20said,Sort%2C%20Bubble%20Sort%2C%20etc.

[]7unestone.academ(2021). <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html>

[8]Bubble sort. https://www.studytonight.com/data-structures/bubble-sort