



Professional Development  
Service for Teachers

An tSeirbhís um Fhorbairt  
Ghairmiúil do Mhúinteoirí



# LEAVING CERTIFICATE COMPUTER SCIENCE

## Algorithms Manual for LCCS Teachers

# Algorithms

**A Manual for Teachers**

of

**Leaving Certificate Computer Science**



© PDST 2021

This work is made available under the Creative Commons Share Alike 3.0 Licence <https://creativecommons.org/licenses/by-sa/3.0/>. You may use and re-use this material (not including images and logos) free of charge in any format or medium, under the terms of the Creative Commons Attribution Share Alike Licence.

Please cite as: Algorithms Manual for LCCS Teachers, PDST Dublin, 2021

---

## Table of Contents

<b>Section 1: Introduction to Algorithms</b> .....	4
Activity #1 .....	8
<b>Section 2: Searching and Sorting Algorithms</b> .....	13
A Simple Sort Algorithm .....	18
The Simple (Selection) Sort .....	20
Insertion Sort .....	25
Bubble Sort .....	32
Quicksort .....	39
Linear Search .....	44
Binary Search .....	51
Activity #2 .....	60
<b>Section 3: Analysis of Algorithms</b> .....	67
Big O .....	68
Activity #3 .....	74
Task A – Analysis of Search Algorithms .....	75
Task B – Analysis of Sorting Algorithms .....	81
<b>Section 4: Critical Reflection: Thoughts on unconscious and algorithmic bias</b> ...	85
Unconscious Bias .....	89
Algorithmic Bias .....	90
<b>Section 5: Final Reflection</b> .....	99

---

## Section 1

### Introduction to Algorithms

In recent years the word algorithm has been slowly creeping out from behind the walls of high-tech companies and the computer science lecture halls of universities and making its way into the public gallery modern society. And the reason for this is simple: algorithms are all around us. They have evolved to shape the way we live our daily lives, the way we think, and perhaps most significantly, who we are. *But what exactly is an algorithm?*

An algorithm is a set of rules for getting a specific output from a specific input. Each step must be so precisely defined that it can be translated into computer language and executed by machine

Donald Knuth (1977)

It is difficult to think of any aspect of modern society that remains untouched by algorithms – application areas include: arts, entertainment, education, banking, finance, insurance, healthcare, medicine, media, social media, travel, tourism, crime, justice, transport, politics, public services, communications, retail, security, manufacturing, military and much, much, more. Sales, marketing, sports, games, astronomy, exploration, science and technology, construction, engineering, agriculture, food, research and development. The list is endless. There are algorithms to recommend our next purchases, the next book to read, the next song to listen to, the next YouTube video to watch – algorithms to maintain playlists, find the perfect partner, schedule our busy lives, pay for and deliver our shopping and so on ad infinitum.

The ubiquitous nature of algorithms and their influence on modern life should be patently clear. And for this reason alone the benefits of having a general understanding of the way they operate should also be clear. Simply put, life can be made easier when one has some level of understanding about the algorithms that are used to drive and support it.

When it comes to the study of algorithms (as is the case with Leaving Certificate Computer Science) their importance takes on an even greater significance. The study of algorithms enables us to provide opportunities for students to ask questions that are fundamental to computer science. Questions such as ...

- What is computable?

- Does an algorithm guarantee a correct solution?
- How optimal is this solution?
- What is the worst case time complexity?

According to Knuth<sup>1</sup> an algorithm has the following five important features:

1. ***Finiteness:*** *An algorithm must always terminate after a finite number of steps. A procedure that has all the characteristics of an algorithm except that it possibly lacks finiteness may be called a **computational method** e.g. reactive processes*
2. ***Definiteness:*** *Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. Algorithms that are expressed using natural languages give rise to the possibility of ambiguity. To get around this difficulty, formally defined programming languages or computer languages are designed for specifying algorithms. An expression of a computational method in a computer language is called a **program**.*
3. ***Input:*** *An algorithm has zero or more inputs, taken from a specified set of objects: quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs.*
4. ***Output:*** *An algorithm has one or more outputs, which have a specified relation to the inputs.*
5. ***Effectiveness:*** *All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.*

A less formal definition of ‘algorithm’ is a step-by-step procedure for solving a problem or accomplishing some end.<sup>2</sup> According to this definition ordinary everyday instructions such as those found in recipe books or any set of instructions (e.g. making a cup of coffee, furniture flat-pack assembly instructions, Lego, changing the oil in a car etc.) could be called algorithms. No computation necessary - what do you think? This definition tells us that basically, if you can clearly describe how to do something, then you can make an algorithm for it.

It is worth noting that there is a big difference between inventing an algorithm and using it. Inventing an algorithm can be very difficult – there can be multiple solutions to the same

---

<sup>1</sup> Source: Knuth, D The Art of Computer Programming (Vol. 1, Fundamental Algorithms, 3rd ed.)

<sup>2</sup> <https://www.merriam-webster.com/dictionary/algorithm>

problem - and the use of computational thinking skills is essential, whereas using an algorithm is just a matter of following the algorithm's instructions.

- ✓ Algorithms are way of capturing intelligence and sharing it with others
- ✓ They provide general solutions to problems (but some problems are so hard that they cannot be solved by algorithms e.g. The Halting Problem)
- ✓ They can be expressed in a variety of different ways – programs, pseudo-code, flowcharts etc.
- ✓ Common elements of algorithms include data acquisition, computation, sequence, selection, iteration and a means to report the output.
- ✓ There is a close relationship between algorithms and data structures.
- ✓ The essential features of all algorithms are correctness and effectiveness

### *Rule Based Algorithms vs. Machine Learning Algorithms*

The distinction between rule-based algorithms and AI/machine learning algorithms is very important and therefore worth discussing.

Rule based algorithms are the traditional algorithms that are written by humans typically using programming constructs such as sequence, selection and iteration. These are the classic algorithms that can be debugged and tested, and behave in a deterministic fashion. We will see later that these type of algorithms can be studied, verified and rigorously analysed.

Although many rule-based algorithms pre-date computer algorithms (Euclid's algorithm for finding the greatest common divisor of two numbers and The Babylonian square-root algorithm (sometimes called Hero's method) are just two examples), there can be little doubt that since the 1950s and the rise in popularity of computers there has been somewhat of an explosion of interest and the development of new rule-based algorithms. This is largely down to the fact that because of their speed and reliability, computers are an ideal tool for running algorithms.

In the next section we will be taking a detailed look at a variety of searching and sorting algorithms (i.e. linear and binary searches, simple (selection) sort, insertion sort, bubble sort and quicksort algorithms) but there are quite literally thousands of other rule based algorithms too. Some classic examples include Google's Page Rank algorithm (written by

Larry Page and Sergey Brin), Dijkstra's shortest path algorithm, Cooley-Tukey algorithm (used to break down signals into frequencies), Moore's Algorithm (used for scheduling and resource allocation) and a wide variety of Greedy (heuristic) algorithms just to name a few.

All operating systems and the vast majority of application software are built using many of these rule-based algorithms. Common examples include word-processing, spreadsheet and database packages, web browsers, graphic/multimedia systems. Other business examples include Customer Relationship Management (CRM) systems, Point-Of-Sale (POS) and stock control systems, Automated Teller Machine (ATM) systems, sales, purchasing, invoicing and accounting systems. Online systems we use to communicate with each other, purchase goods, play games, book cinema or concert tickets, holidays, taxis, flights, hotels, and stream movies and music to our devices are all built from rule-based algorithms.

Machine learning algorithms (and AI) differ from rule-based algorithms in a number of respects. These type of algorithms are designed so that they can be 'trained' over time using a combination of very large volumes of data and human input. These inputs are used by the algorithms to build large and complex mathematical models which are then used to make inferences and predictions. Unlike rule-based algorithms, machine learning algorithms are characterised by a statistical randomness that gives rise to non-deterministic (stochastic) behaviours.

Machine learning algorithms (and AI) were discussed earlier in the section on unconscious bias and are the subject of much debate at the moment. It is probably fair to claim that the recent surge in popularity of machine-learning algorithms is being met by many people with a mix of excitement and a certain degree of trepidation – excitement at the positive potential they hold for society, but trepidation caused by the inability in certain cases by their designers to explain their behaviour.

For an excellent introduction to algorithms watch the BBC4 documentary, *The Secret Rules of Modern Living*<sup>3</sup> produced and directed by David Briggs and presented by Professor Marcus du Sautoy. A nice worksheet to accompany the video is available at the link referenced below.<sup>4</sup>

<sup>3</sup> <https://www.youtube.com/watch?v=kiFfp-HAu64>.

<sup>4</sup> <https://csilvestriblog.files.wordpress.com/2015/09/the-secret-rules-of-modern-living-algorithms.pdf>

## Activity #1: Introduction to Algorithms

Read the scenario below carefully and then watch the video *The Secret Rules of Modern Living*, Marcus Du Sautoy (<https://www.youtube.com/watch?v=kiFfp-HAu64>) from 23:44 to 26:53

*The Stable Marriage Problem (David Gale and Lloyd Shapely, 1962 and later Alvin Roth)*

Suppose you had a group of men and a group of women who wanted to get married. The goal is to find stable matches between two sets of people who have different preferences and opinions on who is their best match.

The central concept is that the matches should be stable: There should be no two people who prefer each other to the partners they actually got e.g. an unstable match would be if Mary and John like each other better than their partners. The problem is to develop a formula to pair everyone off as happily as possible.

Sometimes solutions to problems can have varied (and unexpected) applications. In what other contexts do you think the Gale-Shapley algorithm could be applied?



## Discussion

It is interesting to note how algorithmic solution(s) to some famous (and not so famous) problems have found applications in entirely different (and unexpected) contexts.

The original problem context for the Gale-Shapley algorithm was college admissions i.e. how to match students to colleges so that everyone got a place, but more importantly were happy even if they didn't get their first choice. However, it is quite likely that some of the following applications of solutions to the Stable Marriage Problem were not anticipated in 1962 when Gale-Shapley first posed the problem and invented its solution:

- As recently as 2004 Alvin Roth adapted the Gale-Shapley algorithm to help transplant patients find donors (it is estimated that thousands of lives being saved as a result<sup>5</sup>). Both Shapley and Roth received the Nobel Prize in 2012 for this work. (David Gale passed away in 2008)
- In the 1990s, Roth, with backing from the National Science Foundation, began looking at the National Residency Match Program (NRMP), a system that assigns new doctors to hospitals around the country (USA). The NRMP was struggling because new doctors and hospitals were often both unsatisfied with its assignments. Roth used Gale and Shapely's work to reshape the NRMP matching algorithm so that it produced matches that were more stable.
- Another application was found in assigning (client) users to servers in a large distributed Internet service.
- General solutions to the SMP are also applied in the areas of in economics, stock markets and marketing recommendation systems (basically any scenario which involves supply and demand or matching sellers to buyers).

Can you think of any other contexts where solutions to the Stable Marriage Problem could be applied? What about love?

---

<sup>5</sup> <https://medium.com/@UofCalifornia/how-a-matchmaking-algorithm-saved-lives-2a65ac448698>

## Further Work

This work can be carried out in your own time following the workshop.

Consider potential areas of application for solutions to the following problems/scenarios.

### *Scenario 1: The Secretary Problem (aka The Optimal Stopping Problem)*

Suppose that you are in an ice cream parlour with a hundred different flavours of ice cream: chocolate-mint, peanut butter, pepper, coffee-chocolate-garlic, and many more! Because you do not know any of these strange combinations, the friendly ice cream vendor allows you to taste some! You can try a little spoon of a kind of ice cream and have to decide whether you want a full serving or want to eat something else. Unspoken rules of politeness say that if you have declined a flavour to try a new one, you can never choose that previous flavour again. Which strategy will lead to the best bowl of ice cream?



### *Scenario 2: Two Machine Scheduling*

When you wash your clothes they have to pass through the washer and the dryer in sequence, and different loads will take different amounts of time in each. A heavily soiled load might take longer to wash but the usual time to dry; a large load may take the usual time to wash but a longer time to dry. If you have several loads of laundry to do on the same day, what's the best way to do them?

(This problem originated from a mathematician called Selmar Johnson. The scenario Johnson examined was bookbinding, where each book needs to be printed on one machine and then bound on another. Problem is to minimise the total time for the two machines to complete all their jobs.)



### *Scenario 3: The Elevator Algorithm (aka Karp's algorithm or Knuth's One Tape Sort<sup>6</sup>)*

How would you design an elevator algorithm that is **fair**, both to its passengers and the waiting public?

<sup>6</sup> Knuth, Donald, *The Art Of Computer Programming*. Vol 3, pp 357-360. "One tape sorting"

## Scenario 4: The Travelling Salesman Problem (TSP)

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city once, and only once, and returns to the origin city?



## Scenario 5: The Bridges of Königsberg

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other, or to the two mainland portions of the city, by seven bridges. Can you devise a walk through the city that would cross each of those bridges once and only once?

Solutions involving either of the following are unacceptable:

- reaching an island or mainland bank other than via one of the bridges, or
- accessing any bridge without crossing to its other end



There are many more interesting scenarios/problems to those presented on the previous pages. You are encouraged to research some for yourself and use the space provided on the next page to start recording your findings. Here are some ideas to get you started:

- Discuss methods for pairing socks!
- What about washing dishes?
- What is *The Dining Philosopher Problem*?
- Discuss the nature of the following two scenarios (are they the same?):
  - 1) Suppose we have  $n$  tasks to complete, each with a time estimate – how can we delegate the tasks to two people as evenly as possible?
  - 2) Suppose you needed to divide a large number of assorted crates into two equal weight groups?

Browse to <https://classicproblems.com/> to read more interesting Computer Science problems and bolster your knowledge of algorithms in the process.

**Additional Notes**

## Section 2

### Searching and Sorting Algorithms

#### Introduction

Sorting and searching are at the very heart of what computers do. Most of us can think of situations from our past where we needed to search for something e.g. a name in a contact list, or a song from a playlist. Maybe even an email or a book from a library or a product from an online catalogue. How many times do we use a search engine in a single day?

In today's digital world searching and sorting algorithms are crucial for efficient retrieval and processing of large volumes of data. Without question they are among the most important and the most frequently used algorithms in computer science. In fact, it is estimated that over 25% of computing time is spent on sorting with some installations spending more than 50% of their time sorting<sup>7</sup>. From an environmental perspective that can add up to a lot of energy and greenhouse gases.

Many of us will appreciate that there are classes of algorithms that, for a given input, will compute all possible outputs. Sometimes the amount of output can run well beyond orders of magnitude that humans are capable of dealing with. Consider as examples algorithms for finding the shortest possible route between two points or an algorithm to compute all the possible winning moves from this point in a chess game. Other classes of algorithms work by processing very large amounts of input data just to generate a relatively small amount of output. For example, your favourite social media application might use an algorithm which trawls through its database of millions (and even billions!) of registered users just so that it can present them to you as suggested 'friends' to connect with or to follow. The presentation of the results of these algorithms in sorted order is often as important as the underlying algorithm that was used to gather them in the first place.

Sorting makes it possible to view the same underlying data in multiple ways. For example, products may be presented to an online user in order of price or some other metric such as rating. A football league table sorted in alphabetic order by team would probably look very different to the same table sorted on points. It is a combination of the frequency of the types of computations referred to above, and the sheer volume of the data that make sorting and searching algorithms so important.

---

<sup>7</sup> Source: Fundamentals of Data Structures in Pascal (Horowitz and Shani, Pg. 335)

Finally, and also very importantly, as we will see later when we study the binary search algorithm, sorting makes it possible to search very large data sets in very little time. They also enable easy detection of duplicate values and facilitate the comparison of lists.

### *Research Exercise*

This exercise can be carried out in your own time following the workshop.

**Find out how much time computers spend searching and sorting**

**List as many applications as you can that use a searching/sorting algorithm.**

So, what do we mean by sorting?

An algorithm that maps the following input/output pair is called a sorting algorithm:

Input: A list (aka array),  $L$ , that contains  $n$  orderable elements (often called *keys*):  
 $L[0, 1, \dots, n - 1]$ .

Output: A sorted permutation of  $L$  such that,  
 $L[0] \leq L[1] \leq \dots \leq L[n - 1]$ .

For example,  $[a, b, c, d]$  is sorted alphabetically,  $[1, 2, 3, 4, 5]$  is a list of integers sorted in *increasing order*, and  $[5, 4, 3, 2, 1]$  is a list of integers sorted in *decreasing order*.

By convention, empty lists and lists consisting of only one element (singletons) are always sorted. This is a key point for the base case of many sorting algorithms.

When the (sorted) output occupies the same memory as was used to hold the original (unsorted) input the sorting is said to have been done *in place*. This is a desirable feature for sorting algorithms to have because it means they have little or no additional space requirements (on top of the size of the list that is being sorted).

What is searching?

An algorithm that maps the following input/output pair is called a search algorithm:

Input: An list,  $L$ , that contains  $n$  orderable elements (often called *keys*)  $L[0, 1, \dots, n - 1]$  and some target value commonly referred to as an *argument*.

Output: If the argument is found in  $L$  it is conventional to return its zero-based positional offset (i.e. the index) and if the argument is not found some implementations return the length of the list while others return  $-1$ . (Either of these two outputs can be used to indicate that the argument doesn't exist in  $L$ .)

For example, a search to find argument 'c' in the list  $L$ ,  $[d, a, c, b]$  would return 2 and a search to find argument 'z' (or any other value not on  $L$ ) in the same list would return either 4 or  $-1$ .

For any given problem, it is quite possible that there is more than one algorithm that represents a correct solution. Two good examples of this are the problems of searching and sorting. Dozens of different algorithms have been written to solve this problem. LCCS names these six.

Linear (sequential) Search

Binary Search

Quicksort

Simple (selection) Sort

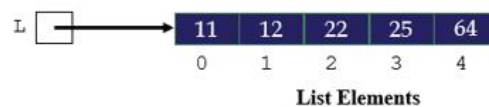
Bubble Sort

Insertion Sort

For the purpose of this workshop we will confine our attention to searching and sorting numeric data (as opposed to alphanumeric or data of any other datatype) that are stored using the list data structure (aka an array).

When we come to look at the implementation of some of these algorithms we will find it is necessary to have a working knowledge of lists i.e. indexing and traversals, the use of comparison operators (the law of trichotomy and the law of transitivity), and how to swap/exchange values. It will also be necessary to have a knowledge of iteration and useful to have an understanding of recursion.

Question: What does the function shown below do?



```
1 def isSorted(L):  
2     for i in range(1, len(L)):  
3         if L[i] < L[i-1]:  
4             return False  
5  
6     return True
```



**List Traversal and The Swap Operation**

### List Traversal

Pass over each element in the list one at a time

`L = [18, 27, 15, 13, 22]`

```
for index in range(len(L)):
    print(L[index])
```

Equivalent to ...

```
print(L[0])
print(L[1])
print(L[2])
print(L[3])
print(L[4])
```

Output Displayed

18  
27  
15  
13  
22

### The Swap Operation

Let's say we wanted to exchange `L[2]` and `L[1]`

`L = [18, 27, 15, 13, 22]`

```
temp = L[2]
L[2] = L[1]
L[1] = temp
```

Python supports a single statement swap:

```
L[2], L[1] = L[1], L[2]
```

A more detailed description of the various search and sort algorithms is presented in the following pages. (In the workshop we skip directly to Activity # 2 on page 60.)

## A Simple Sort Algorithm

We will start our discussion of sort algorithms with this presentation of what is perhaps the simplest sort of all.

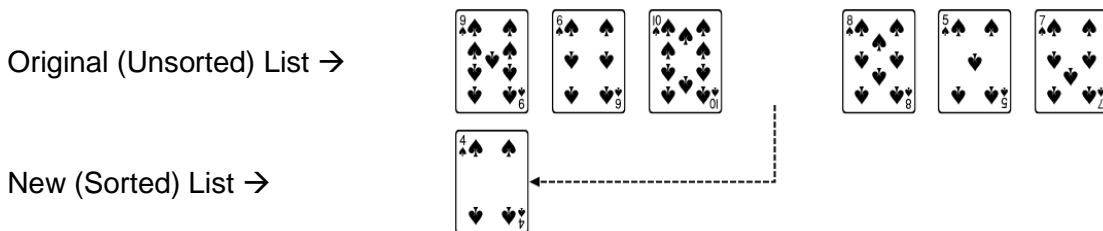
Let's consider the process of sorting the seven unsorted cards shown here.



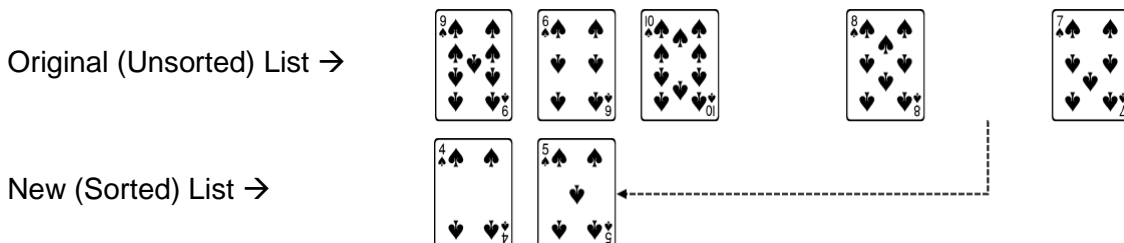
The desired output is:



One approach is to start by finding the smallest card in the unsorted list and moving it into a new list. The smallest card is 4 and this is moved to the new list as illustrated.



We proceed by moving the next smallest card (i.e. 5) from the original list and adding it to the end of the new list.



This process of finding the smallest card from the unsorted list and moving it to the end of the sorted list continues until there are no cards left in the original list and all the cards are sorted.

In general, the simple sort works by repeatedly selecting the smallest item from an unsorted list and moving it to a second list. Once all the items have been removed from the unsorted list, the second list will contain the items in sorted order.

The sequence of steps is as follows:

1. Initialise an unsorted list
2. Initialise an empty sorted list
3. Repeat as long as there are items in the unsorted list
4. Find the smallest item
5. Move the smallest item to the sorted list
6. Stop

These steps can be translated into the following Python code.

```
# A Very Simple Sort v1

unsorted_list = [9, 6, 10, 4, 8, 5, 7] # the list to be sorted
sorted_list = [] # the initial (empty) sorted list

# Loop over every element in the unsorted list
for i in range(len(unsorted_list)):
    smallest = min(unsorted_list) # min returns the smallest
    sorted_list.append(smallest) # append the smallest to the sorted list
    unsorted_list.remove(smallest) # remove the smallest from unsorted_list
```

It is important to note that the above code exploits the `min` built-in function to find the smallest item. The actual algorithm for `min` involves comparing each element to every other element in the list.

### *A note on performance*

The above technique is not considered to be a very efficient algorithm. The main reason for this is that it requires twice as much memory as the size of the original sorted list i.e. in order to sort a list of size  $N$ , the algorithm the space requirements are  $2N$ . This becomes impractical when the number of items in the list becomes large.

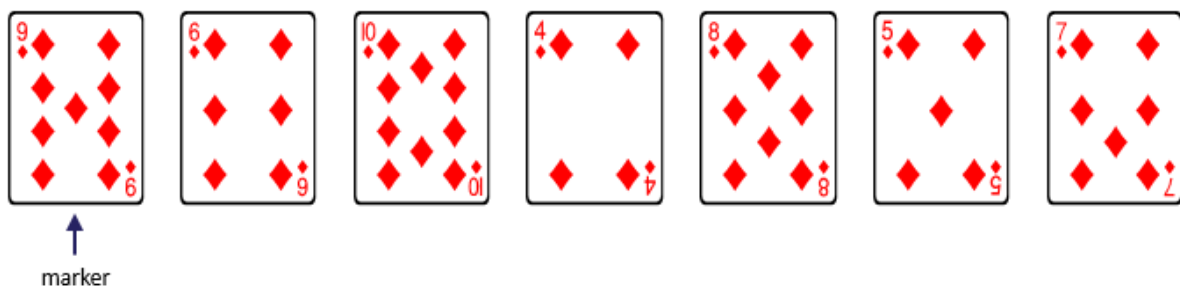
## The Simple (Selection) Sort

The selection sort algorithm is a variation of the algorithm just presented with one important difference – the items are sorted ‘in place’ i.e. without the need for a second list.

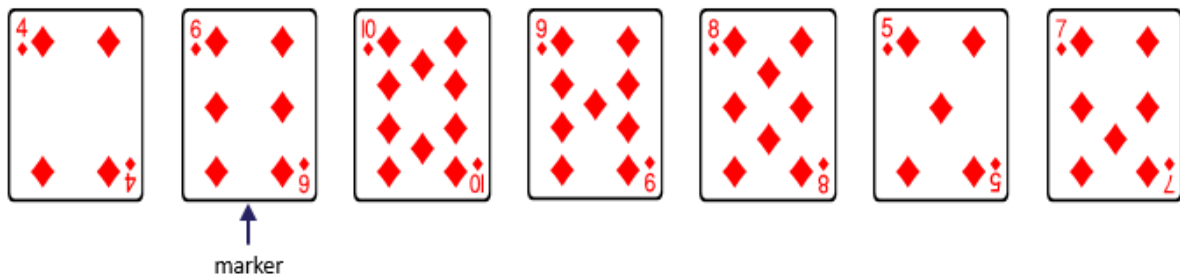
The algorithm maintains a *marker* such that at all times:

- all items to the right of the marker are unsorted
- all items to the left of the marker have been sorted.

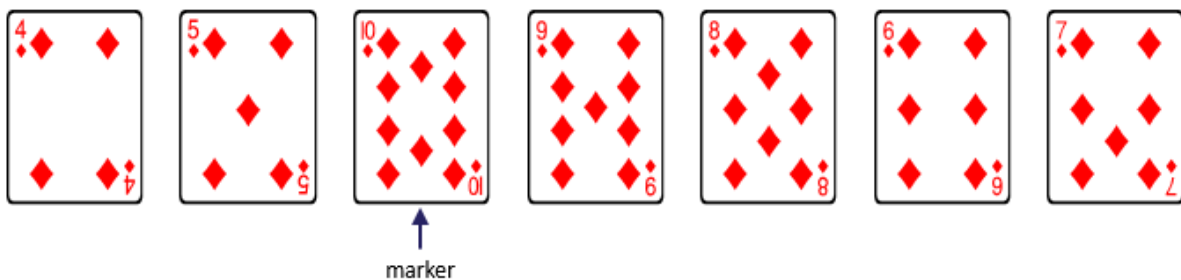
This example shows an unsorted list with the marker in its initial position pointing to the first item in the list.



The algorithm proceeds by finding the smallest item to the right of the marker – in this case 4 – and then swapping this item with the item at the marker. The marker is then advanced to the next position as illustrated.

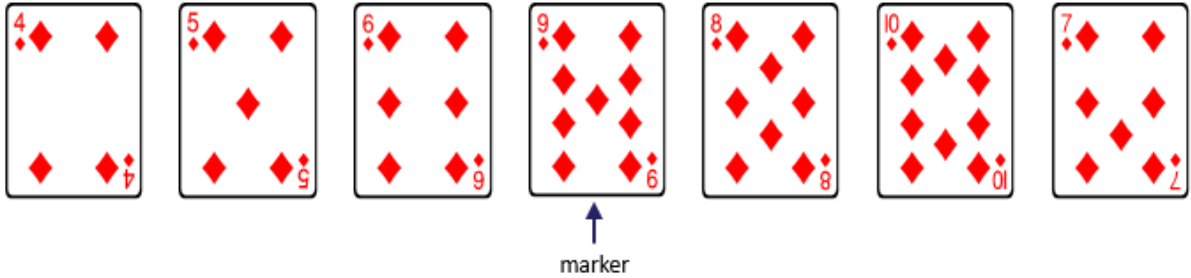


In the next pass the smallest item (to right of the marker, i.e.5) is swapped with the item pointed to by the marker (i.e. 6). This leaves the list looking like this:

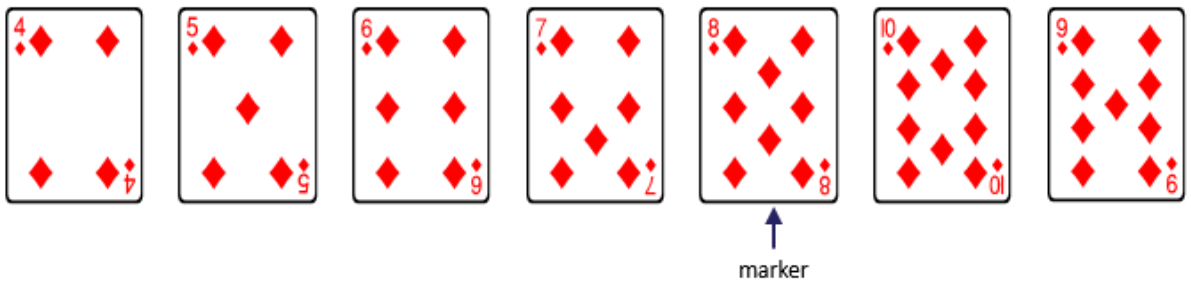


This process continues in a systematic fashion until all the items in the list have been processed.

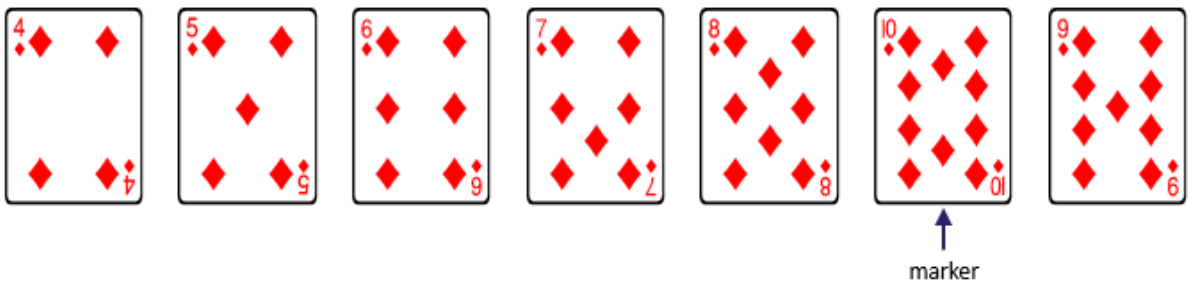
6 has just been swapped with 10. The next swap will be 9 and 7.



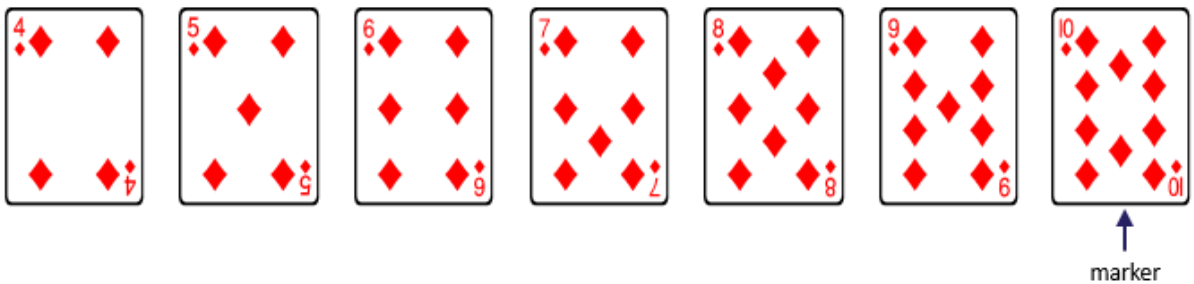
9 has just been swapped with 7 and the marker is advanced to 8 – since no item to the right of the marker is smaller than 8 the list will remain unchanged.



The list remains unchanged and the marker is advanced to 10.



9 is swapped with 10 and the list is sorted.



The steps in the selection sort algorithm are as follows:

1. Initialise an unsorted list
2. Initialise a marker
3. Loop across every list item
4. Find the minimum item to the right of the marker
5. Swap this item with the item at the marker
6. Advance the marker to the right one position
7. Stop

These steps are annotated in the Python implementation shown here.

```
# Simple (Selection) Sort v1

# 1. Initialise an unsorted list
L = [9, 6, 10, 4, 8, 5, 7]
# 2. Initialise a marker
marker = 0

# 3. Traverse through all list items
while marker < len(L):
    # 4. Find the minimum item to the right of the marker
    index_of_min = marker
    for j in range(marker+1, len(L)):
        if L[index_of_min] > L[j]:
            index_of_min = j

    # 5. Exchange the smallest item with the item at the marker
    temp = L[marker] # save the item at the marker
    L[marker] = L[index_of_min] # copy 1
    L[index_of_min] = temp # copy 2

    # 6. Advance the marker to the right by 1 position
    marker = marker+1

# 7. Stop
```

- The values to be sorted are stored in a list called `L`.
- The variable `marker` is used to store the index that will contain the next item to be sorted. All items to the left of `marker` are sorted and all items to the right of `marker` are yet to be processed.

- The variable `index_of_min` is the index of the smallest item to the right of the marker. The item at this position will be swapped with the item at the marker – this is the heart of the algorithm.

The illustration below depicts the changing values of `marker` and `index_of_min` as the algorithm sorts a list of 7 items in `L`.

List index →	0	1	2	3	4	5	6	marker	index_of_min
L	9	6	10	4	8	5	7	0	3
L	4	6	10	9	8	5	7	1	5
L	4	5	10	9	8	6	7	2	5
L	4	5	6	9	8	10	7	3	6
L	4	5	6	7	8	10	9	4	4
L	4	5	6	7	8	10	9	5	6
L	4	5	6	7	8	9	10	6	

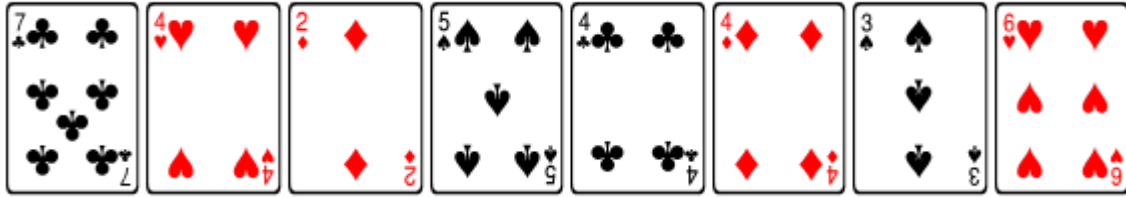
**Key:**  Value of item at marker  
 Value of minimum item to the right of marker

### Exercises – Simple (Selection) Sort

1. Use the simple (selection) sort algorithm to sort the list [7, 8, 5, 2, 4, 6, 3] shown below. (Fill in the blanks in the same manner as above.)

List index	0	1	2	3	4	5	6	marker	index_of_min
aList	7	8	5	2	4	6	3	0	
aList								1	
aList								2	
aList								3	
aList								4	
aList								5	
aList	2	3	4	5	6	7	8	6	

2. Perform a simple (selection) sort on the face values of the following cards.





## Insertion Sort

We can develop our understanding of the insertion sort as follows:

- a list with one item is already sorted.
- a list with two items can be sorted by sorting the second item relative to the first. If the second item is greater than the first, the two items are already sorted and nothing further needs to be done; otherwise we obtain our sorted list by swapping the two items.
- a list with three items can be sorted by sorting the first two items (as just described) and then sorting the third item relative to the first two.
- a list with four items can be sorted by sorting the first three items (as just described) and then sorting the fourth item relative to the first three.
- And so on.

### Example

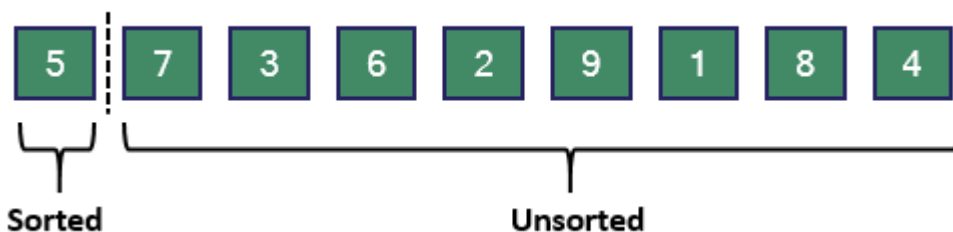
Let's say we were asked to sort the list of numbers shown below in ascending order.



The desired output is:



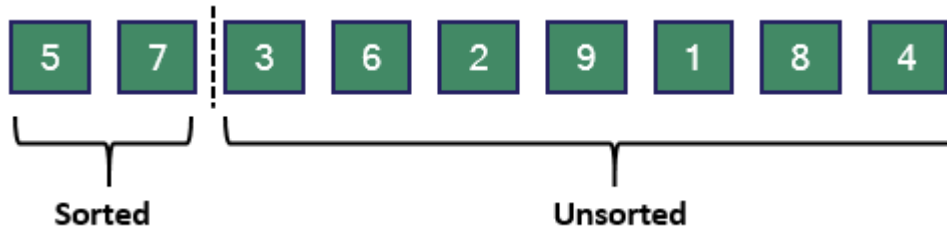
The insertion sort starts at the leftmost item. It sets a marker between the first and second item. Everything to the left of the marker is always sorted and everything to the right of the marker remains to be sorted. This is illustrated as follows:



The algorithm proceeds as follows until the entire list is sorted:

1. Select the first item from the unsorted list (in this case 7)
2. Insert the selected item into the correct position within the sorted this (this is done by swapping this item to the left until it arrives at the correct position)
3. Advance the marker to the right by one position

After following these three instructions the 7 remains in the same position (as it is already sorted relative to 5) and the marker is advanced to the right by 1. The list now looks like this:

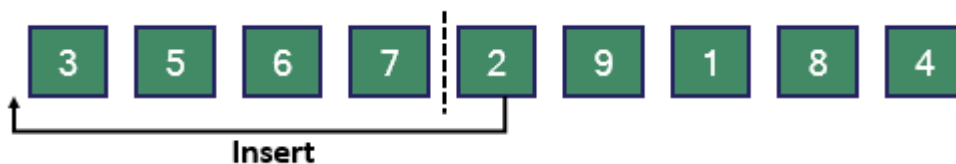


The next item to sort is 3 (because this is the first item in the unsorted list) and so 3 is inserted into the sorted list. This is what the list looks like after 3 has been inserted. The next item to insert will be 6.



By this point you may be wondering how, on each pass, the selected item gets inserted into its correct position. We'll come to this soon – for the moment it's important to grasp the outer loop which iterates over each item in the list.

At this point 6 has been inserted into the sorted list and the next item to sort is 2.



This is what the list looks like after 2 has been inserted. The next item to sort is 9.



Since 9 is already sorted relative to the list on its left the list will remain unchanged. We just advance the marker to the right by one position and consider the next item to sort which is 1.

This is what the list looks like before 1 is inserted into its correct (sorted) position.



This is what the list looks like after 1 has been inserted. The next item to sort is 8.



This is what the list looks like after 8 has been inserted. The next (and final) item to sort is 4.



This is what the list looks like after 4 has been inserted. There are no more items to the right of the marker and so the algorithm terminates.



Because we have maintained the list to the left of the marker in a sorted state throughout, we can safely conclude that this final list is sorted.

### Reflection Exercise

1. How many insertions do you think would be necessary if the initial list was
  - a) already sorted e.g. 1, 2, 3, 4, 5
  - b) in reverse order e.g. 5, 4, 3, 2, 1
  
2. Generalise your answer to 1 for a list of any size (i.e. a list of size  $n$ .)

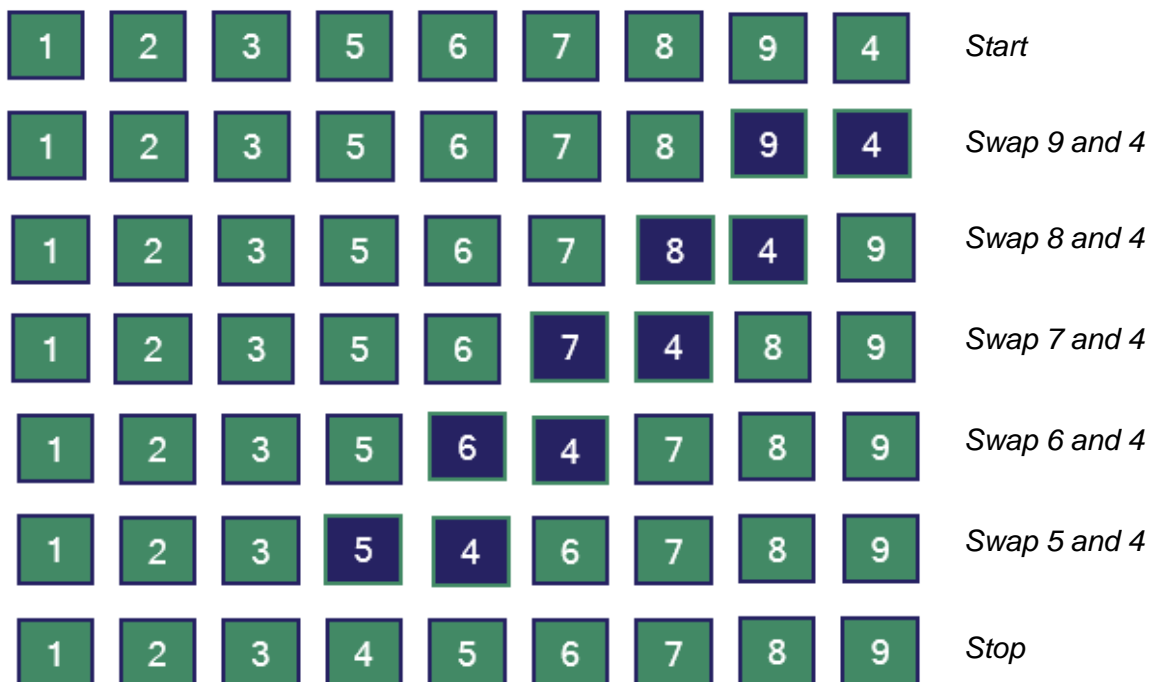
## Inserting the item to its correct position

Before we look at an implementation of the insertion sort, it is helpful to understand how step 2 of the algorithm works. Step 2 says *insert the selected item into the correct position within the sorted this*. How do we do this?

Consider the transition (shown here) that takes place in the final step of our example. The question is: *how does the 4 get inserted into the correct position?*



The answer is: *4 is repeatedly swapped back with all larger numbers to its left*. The step-by-step sequence of swaps are illustrated below:



The algorithm for this swap sequence is shown in the code below.

```

# repeatedly swap a[j] with larger numbers to its left
while (a[j] < a[j-1] and j>0):
    temp = a[j]
    a[j] = a[j-1]
    a[j-1] = temp
    j = j-1
  
```

The full Python implementation of the insertion sort is shown below:

```
# 1. Initialise an unsorted list
the_list = [5, 7, 3, 6, 2]
# 2. Initialise a marker
marker = 1

# 3. Traverse through all list items
while (marker < len(the_list)):
    # 4. Insert the selected item to its correct position
    j = marker
    while (the_list[j] < the_list[j-1] and j>0):
        tmp = the_list[j]
        the_list[j] = the_list[j-1]
        the_list[j-1] = tmp
        j = j-1

    # 6. Advance the marker to the right by 1 position
    marker = marker+1
```

Starting with `the_list` comprising of [5, 7, 3, 6, 2] the table below highlights the comparisons and exchanges that take place on each pass of the insertion sort algorithm.

Pass	State of List (before-> after)	Comment
1	[5, 7, 3, 6, 2] -> [5, 7, 3, 6, 2]	5 and 7 are compared but not exchanged since they are both in order relative to one another
2	[5, 7, 3, 6, 2] -> [5, 3, 7, 6, 2] [5, 3, 7, 6, 2] -> [3, 5, 7, 6, 2]	7 and 3 are compared and exchanged 5 and 3 are compared and exchanged
3	[3, 5, 7, 6, 2] -> [3, 5, 6, 7, 2]	7 and 6 are compared and exchanged
4	[3, 5, 6, 7, 2] -> [3, 5, 6, 2, 7] [3, 5, 6, 2, 7] -> [3, 5, 2, 6, 7] [3, 5, 2, 6, 7] -> [3, 2, 5, 6, 7] [3, 2, 5, 6, 7] -> [2, 3, 5, 6, 7]	7 and 2 are compared and exchanged 6 and 2 are compared and exchanged 5 and 2 are compared and exchanged 3 and 2 are compared and exchanged

The total number of passes is four. The total number of comparison operations is eight and the total number of exchanges is seven.

## Exercises – Insertion Sort

1. Explain what is going on at each stage of the insertion sort algorithm below.  
Make sure to identify all comparison and exchange operations.

Data	Comment							
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>7</td><td>8</td><td>5</td><td>2</td><td>4</td><td>6</td><td>3</td></tr> </table>	7	8	5	2	4	6	3	This is the initial unsorted list.
7	8	5	2	4	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>7</td><td style="border-left: 1px dashed black;">8</td><td>5</td><td>2</td><td>4</td><td>6</td><td>3</td></tr> </table>	7	8	5	2	4	6	3	
7	8	5	2	4	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>7</td><td>8</td><td style="border-left: 1px dashed black;">5</td><td>2</td><td>4</td><td>6</td><td>3</td></tr> </table>	7	8	5	2	4	6	3	
7	8	5	2	4	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>5</td><td>7</td><td>8</td><td style="border-left: 1px dashed black;">2</td><td>4</td><td>6</td><td>3</td></tr> </table>	5	7	8	2	4	6	3	
5	7	8	2	4	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>2</td><td>5</td><td>7</td><td>8</td><td style="border-left: 1px dashed black;">4</td><td>6</td><td>3</td></tr> </table>	2	5	7	8	4	6	3	
2	5	7	8	4	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>2</td><td>4</td><td>5</td><td>7</td><td>8</td><td style="border-left: 1px dashed black;">6</td><td>3</td></tr> </table>	2	4	5	7	8	6	3	
2	4	5	7	8	6	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>2</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td style="border-left: 1px dashed black;">3</td></tr> </table>	2	4	5	6	7	8	3	
2	4	5	6	7	8	3		
<table border="1" style="display: inline-table; text-align: center;"> <tr><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td style="border-left: 1px dashed black;"></td></tr> </table>	2	3	4	5	6	7	8	
2	3	4	5	6	7	8		

Total number of comparison operations:

Total number of exchanges:

2. Perform an insertion sort on the following list of integers:



## Bubble Sort

The bubble sort algorithm works by repeatedly comparing adjacent element and swapping them if they are out of order. The effect is that on each pass of the bubble sort, the largest unsorted item ‘bubbles’ towards the end of the list into its sorted position.

The algorithm is summarised below for an ascending order sort:

1. Initialise an unsorted list
2. Traverse across every element in the list
3. Compare all adjacent elements starting from the beginning
4. If the elements are out of order, then swap them

### Example

Let’s look at how the bubble sort algorithm sorts the list of numbers shown here into ascending order.



After Pass	State of List (at the end of the pass)	Explanation
1		After pass 1, 7 has ‘bubbled’ up to the top of the list.
2		After pass 2, 6 has bubbled into its sorted position.
3		After pass 3, 5 has bubbled into its position.
4		After pass 4, 3 has bubbled into its position.
5		After pass 5, 2 has bubbled into its position.

Notice that 5 passes over the list were required in order to sort the 5 items. In general, the bubble sort will take  $n$  passes to sort a list of  $n$  items.



We now examine what happens in pass 1 in greater detail. The following illustrations depict the exchanges that take place in pass 1, and in particular, explain how 7 bubbles to the end of the list.

This is the initial list.



The first two numbers to be compared are 5 and 7. Since these two numbers are in order no exchange is necessary.



The algorithm then proceeds by comparing the next adjacent pair i.e. 7 and 3. Since they are out of order they must be swapped.



This is what the list looks like after 7 and 3 have been swapped.



The algorithm then compares 7 and 6 and since these two numbers are out of order they must be swapped.



6 and 7 have been swapped.



7 and 2 are the next adjacent pair to be compared. Since 7 is greater than 2 they are swapped.



This is the final state of the list after pass 1. As there are no more adjacent pairs the algorithm proceeds to pass 2.



Notice that in the above list of 5 items there are 4 comparisons. In general, for a list of  $n$  elements, the bubble sort will make  $n - 1$  comparisons on each pass.

### Reflection Exercise

Do you think the bubble sort is an efficient algorithm? Justify your answer.

We will now look at a Python implementation of the bubble sort algorithm.

```
# Bubble Sort v1

# 1. Initialise an unsorted list
L = [5, 7, 3, 6, 2]

print("INPUT (initial list): ", L)

# 2. Traverse across every element in the list
for i in range(len(L)):
    # 3. Compare all adjacent elements starting from the beginning
    for j in range(len(L)-1):
        # 4. if the elements are out of order, then swap them
        if L[j] > L[j+1]:
            temp = L[j+1]
            L[j+1] = L[j]
            L[j] = temp

print("OUTPUT (sorted list): ", L)
```

The exchanges that take place on each pass are highlighted below

Pass	Exchanges (before -> after)	Comment
1	[5, 7, 3, 6, 2] -> [5, 7, 3, 6, 2] [5, <b>7</b> , <b>3</b> , 6, 2] -> [5, 3, 7, 6, 2] [5, 3, <b>7</b> , <b>6</b> , 2] -> [5, 3, 6, 7, 2] [5, 3, 6, <b>7</b> , <b>2</b> ] -> [5, 3, 6, 2, 7]	This sequence of exchanges was detailed on the previous page. Notice that after 4 comparisons and 3 exchanges 7 has bubbled up to the end of the list
2	[ <b>5</b> , <b>3</b> , 6, 2, 7] -> [3, 5, 6, 2, 7] [3, 5, 6, 2, 7] -> [3, 5, 6, 2, 7] [3, 5, <b>6</b> , <b>2</b> , 7] -> [3, 5, 2, 6, 7] [3, 5, 2, 6, 7] -> [3, 5, 2, 6, 7]	Notice that 5 and 3 are initially exchanged. 5 and 6 are compared but not exchanged because 6 is bigger. 6 and 2 are then exchanged. This brings 6 to its sorted position.
3	[3, 5, 2, 6, 7] -> [3, 5, 2, 6, 7] [3, <b>5</b> , <b>2</b> , 6, 7] -> [3, 2, 5, 6, 7] [3, 2, 5, 6, 7] -> [3, 2, 5, 6, 7] [3, 2, 5, 6, 7] -> [3, 2, 5, 6, 7]	3 is compared to 5 but there is no exchange (as they are in order). Then 5 is compared to 2 and they are exchanged. 5 is compared to 6 and then 7 but no exchanges ensue and so 5 is in its sorted position.
4	[ <b>3</b> , <b>2</b> , 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7]	3 is exchanged with 2 to bring it to its final sorted position. No further exchanges take place.
5	[2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7] [2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7]	Although each pair of adjacent items are compared, no exchanges take place as the list happens to be sorted. The comparisons are 2 with 3, 3 with 5, 5 with 6 and 6 with 7.

By this stage it should be evident that the bubble sort is not a very efficient algorithm. We will discuss two inefficiencies:

1. The first inefficiency derives from the fact the outer loop traverses over every element in the list – even if the list is already sorted (and no matter how many items the algorithm thinks it has left to sort).

To highlight this problem let us consider how the algorithm behaves if it is presented with a list that was already sorted e.g.  $L = [1, 2, 3, 4]$ . The algorithm proceeds to make 4 passes over the data - each pass compares the adjacent elements (3 comparisons: 1 with 2, 2 with 3 and 3 with 4). No exchange ever ensues since elements are all in the required order giving a total of 12 unnecessary comparison operations.

Now consider the algorithm's behaviour if the initial list look like this:  $[4, 2, 3, 1]$ . By the end of the first pass 4 would have bubbled to the end and the list would be sorted. Despite this, the algorithm would continue with three more 'exchange-less' passes. In this case we we have 9 unnecessary comparison operations

In order to eliminate this inefficiency, we introduce a flag called `exchange`. The outer loop is modified so that the program traverses across every element as long as `exchange` has a value of `True`. The flag is initialised to `False` at the start of each pass and set to `True` only when an exchange occurs.

```
# Bubble Sort v2
# 1. Initialise an unsorted list
aList = [1, 2, 3, 4]

exchange = True
i = 0
# 2. Traverse across every element as long as there are exchanges
while (i < len(L)) and (exchange == True): # or just 'exchange'
    exchange = False # assume that there will be no exchanges
    # 3. Compare all adjacent elements starting from the beginning
    for j in range(len(L)-1):
        # 4. if the elements are out of order, then swap them
        if L[j] > L[j+1]:
            temp = L[j+1]
            L[j+1] = L[j]
            L[j] = temp
            exchange = True # we've done an exchange!

    i = i+1 # increment the loop counter
```

Although it might seem trivial, this is a decent improvement on the previous version of the algorithm. The algorithm now recognises (by the absence of any exchanges) when the list is sorted and can terminate accordingly. Consider how many comparison operations this would save in a sorted list of 1,000,000 items.

- The second inefficiency in the algorithm derives from the fact that the algorithm ignores the items it has already sorted on previous passes. To illustrate this point clearly let us return to our earlier example. The table below highlights the (unnecessary) comparisons that are made involving items that have already been sorted.

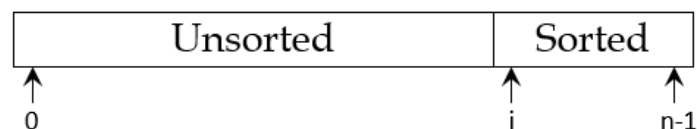
Pass	State of List (before-> after)	Comment
1	[5, 7, 3, 6, 2] -> [5, 3, 6, 2, 7]	After pass 1, 7 has been moved into its sorted position. There are no unnecessary comparisons.
2	[5, 3, 6, 2, 7] -> [3, 5, 2, 6, 7]	6 is unnecessarily compared to 7 at the end of pass 2 (because since 7 has already been sorted the comparison cannot result in an exchange).
3	[3, 5, 2, 6, 7] -> [3, 2, 5, 6, 7]	5 is unnecessarily compared to 6 and 6 is unnecessarily compared to 7 at the end of pass 2
4	[3, 2, 5, 6, 7] -> [2, 3, 5, 6, 7]	There are 3 unnecessary comparisons
5	[2, 3, 5, 6, 7] -> [2, 3, 5, 6, 7]	All 4 comparisons are unnecessary.

Each pass makes  $n - 1$  comparisons even though the comparisons involving the sorted items cannot result in an exchange. The solution is to reduce the number of iterations of the inner loop by 1 on each pass of the data.

This is done in our final implementation of the bubble sort which is shown on the next page.

The algorithm works by maintaining a variable,  $i$  such that for a list of length  $n$ :

- all items  $A[0 \dots i - 1]$  are unsorted and
- all items  $A[i \dots n - 1]$  are sorted



```
# Bubble Sort v3

# 1. Initialise an unsorted list
L = [5, 7, 3, 6, 2, 4, 1]

print("INPUT (initial list): ", L)

exchange = True
n = len(L)
i = 0
# 2. Traverse across every element as long as there are exchanges
while (i < n) and exchange:
    print("BEFORE PASS %d: %s " % (i+1, L))
    exchange = False # assume that there will be no exchanges
    # 3. Compare all unsorted adjacent elements
    for j in range(n-i-1):
        # 4. if the elements are out of order, then swap them
        print("%s " % L, end="-> ")
        if L[j] > L[j+1]:
            L[j], L[j+1] = L[j+1], L[j] # Canonical swap!
            exchange = True # we've done an exchange!

        print("%s " % L)

    print("AFTER PASS %d: %s " % (i+1, L))
    i = i+1 # increment the loop counter

print("OUTPUT (sorted list): ", L)
```

Take some time to study the code and understand how the `for` loop highlighted in the above code is used to improve the efficiency of earlier versions of the bubble sort algorithm.

Notice the use of the `print` statements to display the states of the list as the sort progresses - the output is shown on the next page.

As an exercise you might consider modifying the code so that it computes the following:

- the number comparisons on each pass
- the total number of exchanges on each pass
- the total number of comparisons
- the total number of exchanges

## Exercise – Bubble Sort

The data shown on the left below was generated by our final implementation of the bubble sort algorithm shown on the previous page. Use the right hand column to explain the progress of the algorithm.

INPUT (initial list): [5, 7, 3, 6, 2, 4, 1]	
BEFORE PASS 1: [5, 7, 3, 6, 2, 4, 1]	<b>Pass 1:</b>
[5, 7, 3, 6, 2, 4, 1] -> [5, 7, 3, 6, 2, 4, 1]	<i>5 is compared with 7. No exchange</i>
[5, 7, 3, 6, 2, 4, 1] -> [5, 3, 7, 6, 2, 4, 1]	<i>7 is exchanged with 3</i>
[5, 3, 7, 6, 2, 4, 1] -> [5, 3, 6, 7, 2, 4, 1]	
[5, 3, 6, 7, 2, 4, 1] -> [5, 3, 6, 2, 7, 4, 1]	
[5, 3, 6, 2, 7, 4, 1] -> [5, 3, 6, 2, 4, 7, 1]	
[5, 3, 6, 2, 4, 7, 1] -> [5, 3, 6, 2, 4, 1, 7]	
AFTER PASS 1: [5, 3, 6, 2, 4, 1, 7]	
BEFORE PASS 2: [5, 3, 6, 2, 4, 1, 7]	<b>Pass 2:</b>
[5, 3, 6, 2, 4, 1, 7] -> [3, 5, 6, 2, 4, 1, 7]	<i>5 is exchanged with 3</i>
[3, 5, 6, 2, 4, 1, 7] -> [3, 5, 6, 2, 4, 1, 7]	<i>5 is compared with 6. No exchange</i>
[3, 5, 6, 2, 4, 1, 7] -> [3, 5, 2, 6, 4, 1, 7]	
[3, 5, 2, 6, 4, 1, 7] -> [3, 5, 2, 4, 6, 1, 7]	
[3, 5, 2, 4, 6, 1, 7] -> [3, 5, 2, 4, 1, 6, 7]	
AFTER PASS 2: [3, 5, 2, 4, 1, 6, 7]	
BEFORE PASS 3: [3, 5, 2, 4, 1, 6, 7]	<b>Pass 3:</b>
[3, 5, 2, 4, 1, 6, 7] -> [3, 5, 2, 4, 1, 6, 7]	
[3, 5, 2, 4, 1, 6, 7] -> [3, 2, 5, 4, 1, 6, 7]	
[3, 2, 5, 4, 1, 6, 7] -> [3, 2, 4, 5, 1, 6, 7]	
[3, 2, 4, 5, 1, 6, 7] -> [3, 2, 4, 1, 5, 6, 7]	
AFTER PASS 3: [3, 2, 4, 1, 5, 6, 7]	
BEFORE PASS 4: [3, 2, 4, 1, 5, 6, 7]	<b>Pass 4:</b>
[3, 2, 4, 1, 5, 6, 7] -> [2, 3, 4, 1, 5, 6, 7]	
[2, 3, 4, 1, 5, 6, 7] -> [2, 3, 4, 1, 5, 6, 7]	
[2, 3, 4, 1, 5, 6, 7] -> [2, 3, 1, 4, 5, 6, 7]	
AFTER PASS 4: [2, 3, 1, 4, 5, 6, 7]	
BEFORE PASS 5: [2, 3, 1, 4, 5, 6, 7]	<b>Pass 5:</b>
[2, 3, 1, 4, 5, 6, 7] -> [2, 3, 1, 4, 5, 6, 7]	
[2, 3, 1, 4, 5, 6, 7] -> [2, 1, 3, 4, 5, 6, 7]	
AFTER PASS 5: [2, 1, 3, 4, 5, 6, 7]	
BEFORE PASS 6: [2, 1, 3, 4, 5, 6, 7]	<b>Pass 6:</b>
[2, 1, 3, 4, 5, 6, 7] -> [1, 2, 3, 4, 5, 6, 7]	
AFTER PASS 6: [1, 2, 3, 4, 5, 6, 7]	
BEFORE PASS 7: [1, 2, 3, 4, 5, 6, 7]	<b>Pass 7:</b>
AFTER PASS 7: [1, 2, 3, 4, 5, 6, 7]	<i>No Exchange</i>
OUTPUT (sorted list): [1, 2, 3, 4, 5, 6, 7]	

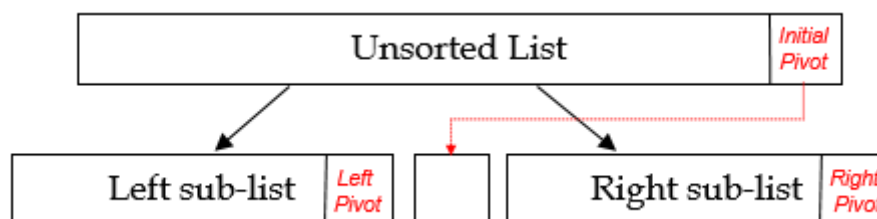
## Quicksort

The quicksort algorithm was developed in 1962 by the famous British computer scientist, Tony Hoare. As its name suggests, quicksort, is a very efficient sorting algorithm (considered to be the fastest general purpose sorting algorithm). Quicksort belongs to a special class of algorithms called *divide-and-conquer* algorithms and owes much of its efficiency to divide-and-conquer as a general problem solving technique. (Merge sort is another popular example of a divide-and-conquer sorting algorithm and later in this manual we will see how binary search uses the divide-and-conquer technique is used to search for some arbitrary value in a list of keys.)

The general principle of divide-and-conquer is to solve large problems by decomposing or breaking them down into smaller sub-problems and solving these smaller problems recursively, and then combining the results to form a complete solution.

In particular, the quicksort algorithm operates by dividing its list into two partitions around some special value called a *pivot*. The lists are divided so that all the elements in the first partition are less than or equal to the pivot and all the elements in the second partition are greater than the pivot. By sorting the sub-lists using the exact same technique we eventually reach the point where all elements are sorted.

The illustration below depicts an unsorted list with the last element chosen as an initial pivot. The list is partitioned into two sub-lists – a left sub-list and a right-sub-list. All the elements in the left sub-list are less than the pivot and all the elements in the right-sub-list are greater than the pivot. The algorithm proceeds by sorting the two sub-lists recursively.



The diagram depicts the initial pivot sorted with respect to the two sub-lists.

The steps of the quicksort algorithm can be expressed recursively as follows:

STEP 1. Choose the rightmost element in the list as the `pivot`

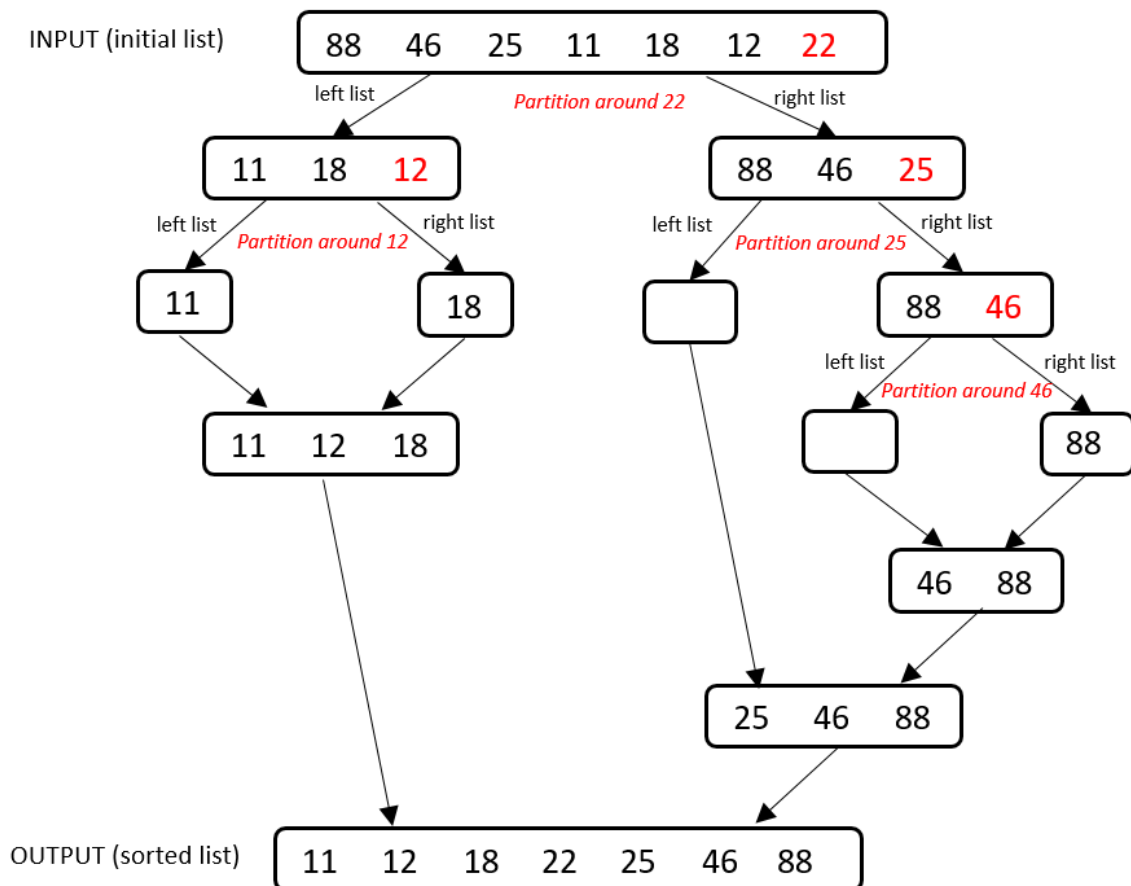
STEP 2. Create three empty lists called `left_list`, `middle_list` and `right_list`

STEP 3. for each element (key) in the list

- if element is  $<$  pivot add it to `left_list`
- if element is  $=$  pivot add it to `middle_list`
- if element is  $>$  pivot add it to `right_list`

STEP 4. The result is a list made up by applying steps 1-3 to `left_list`, followed by the elements in `middle_list`, followed by applying steps 1-3 to `right_list`

Each list is partitioned until it contains just one element. These steps are illustrated in the graphic below starting with an unsorted list [88, 46, 25, 11, 18, 12, 22] with 22 as the pivot.





A Python implementation of the quicksort algorithm is shown below:

```
def quick_sort(L):  
    left_list = []  
    middle_list = []  
    right_list = []  
  
    # Base case  
    if len(L) <=1:  
        return(L)  
  
    # Set pivot to the last element in the list  
    pivot = L[len(L)-1]  
  
    # Iterate through all elements (keys) in L  
    for key in L:  
        if key < pivot:  
            left_list.append(key)  
        elif key == pivot:  
            middle_list.append(key)  
        else:  
            right_list.append(key)  
  
    # Repeat the quicksort on the sub-lists and combine the results  
    return quick_sort(left_list) + middle_list + quick_sort(right_list)
```

The crux of the algorithm is the partitioning process described in step 3 on the previous page. This process is applied recursively to every left and right list i.e. quicksort the left sub-list and quicksort the right sub-list, until the list is either empty or contains a single element. (This is the base case used to end the recursion.) The final sorted list is assembled by concatenating these base case lists together.

The algorithm can be tested using the following driver code:

```
# Driver code ...  
L = [88, 46, 25, 11, 18, 12, 22]  
print("INPUT (initial list): ", L)  
print("OUTPUT (sorted list): ", quick_sort(L))
```

When the program is run the following output is generated:

```
INPUT (initial list):  [88, 46, 25, 11, 18, 12, 22]  
OUTPUT (sorted list):  [11, 12, 18, 22, 25, 46, 88]
```

## Notes:

- 1) The same functionality of the final line of code in the function (i.e. the return statement) could be achieved by using the following three lines:

```
sorted_left_lists = quick_sort(left_list)
sorted_right_lists = quick_sort(right_list)
return sorted_left_lists + middle_list + sorted_right_lists
```

- 2) The choice of pivot value is important and several different techniques are employed. In some implementations the middle element is chosen as the pivot; in others it is the first element; more advanced implementation select the pivot based on the arithmetic mean of the list elements. The implementation shown here use the last element for the pivot.

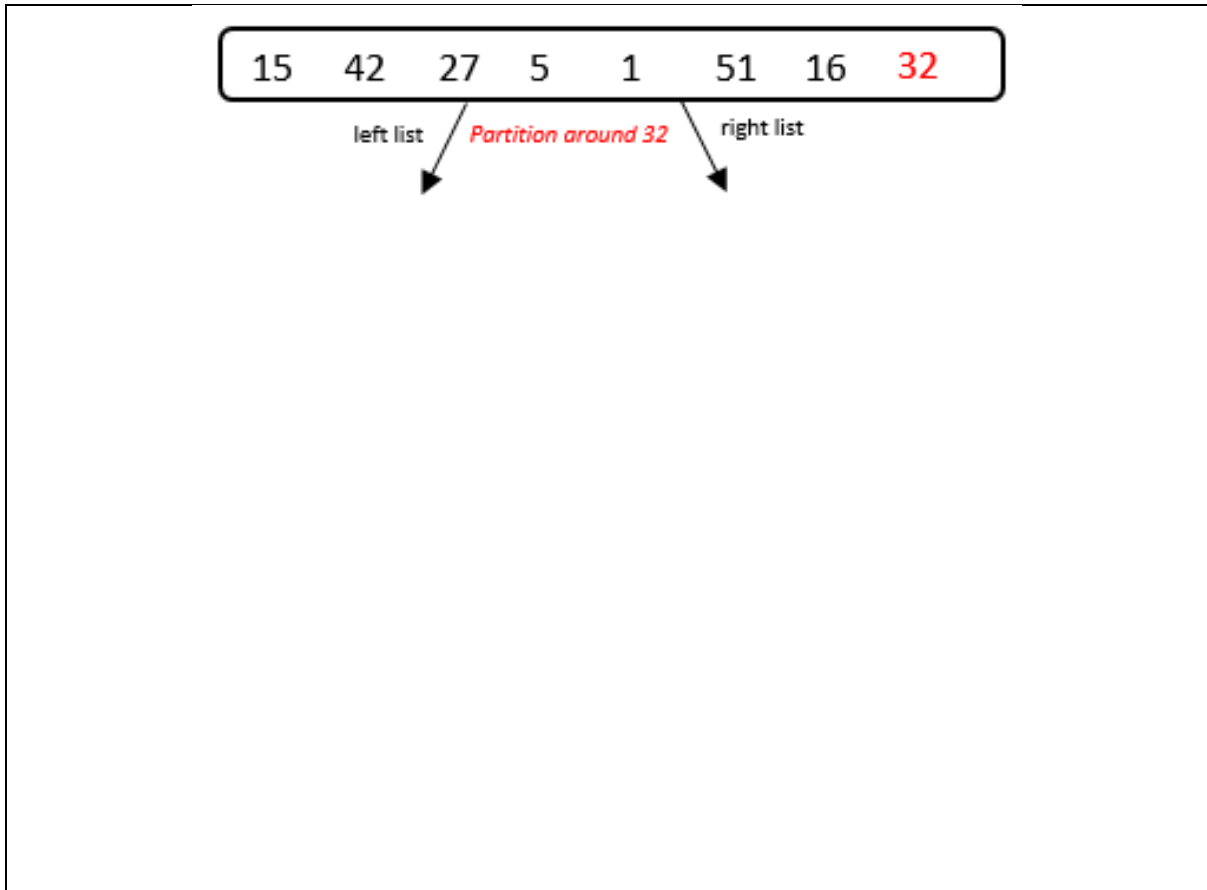
A useful exercise is to consider how the performance of the algorithm would be impacted if the pivot chosen was either the smallest or the largest element in the list.

- 3) This is not the most efficient implementation of the quicksort possible – in fact, it is a very inefficient version of quicksort (and is used here because of its simplicity relative to other versions of the same algorithm). The inefficiency of this implementation is mainly down to its reliance on additional external memory in order to store the left and right sub-lists. For very large lists this becomes highly inefficient and even infeasible.

More efficient implementations do not require the use of additional memory and can perform the sort using ‘in place’ memory. Such techniques work by exchanging elements either side of the pivot that are found to be out of order relative to the pivot. For example, elements that are larger than the pivot and to its left might be exchanged with elements that are smaller than the pivot and to its right.

## Exercise

Show, in the style of the quicksort tree diagram depicted earlier, how the following list of integers could be sorted using a quicksort. The initial pivot is 32 - shown here in red.



Use the space below to explain in your own words how the quicksort algorithm works:

## Linear Search

Let's say we were asked the question: *does the list below contain the number fourteen?* Without thinking twice, most of us would scan down through the list until we arrive at the number fourteen. This intuitive response is called a linear search.

15	4	41	13	24	14	12	21
----	---	----	----	----	----	----	----

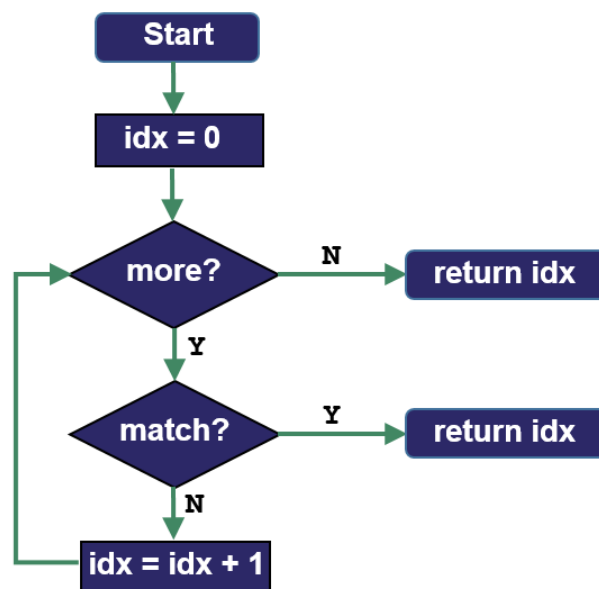
As we scan each element we perform a quick Boolean calculation. `True` or `False` - is the element I am looking at equal to fourteen? If the result is `true`, we have found the required element and the search can end; otherwise, if the result is `False` we automatically (and very quickly) move on to the next element and repeat the Boolean calculation. This process continues until either we find fourteen, or we reach the end of the list, by which time we can conclude that the fourteen is not contained in the list.

The linear search algorithm is also called a *sequential search*. The sequential nature of the process is illustrated below.

<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	Is 15 the same as 14? No. Move to next element.
15	4	41	13	24	14	12	21		
<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	Is 4 the same as 14? No. Move to next element.
15	4	41	13	24	14	12	21		
<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	Is 41 == 14? No. Move to next element.
15	4	41	13	24	14	12	21		
<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	13 == 14? No. Next element
15	4	41	13	24	14	12	21		
<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	if 24 == 14: Found Else: Next element
15	4	41	13	24	14	12	21		
<table border="1" style="background-color: #1a2b4d; color: white; width: 100%;"> <tr> <td>15</td><td>4</td><td>41</td><td>13</td><td>24</td><td>14</td><td>12</td><td>21</td> </tr> </table> <p style="text-align: center;">↑</p>	15	4	41	13	24	14	12	21	if 14 == 14: <b>Found (so STOP!)</b> Else: Next element
15	4	41	13	24	14	12	21		

Given a list of elements to search through (i.e. keys), and a target value to search for (i.e. an argument), the steps of the linear (sequential) search algorithm can be expressed as follows:

1. Set a marker at the start of the list (called `idx` in the flowchart below)
2. Loop through steps 3 – 7 as long as there are more numbers to compare
3. Compare the current element to the target value
4. If they match:
  5. Return the value of the marker (`idx`)
6. If they are not equal:
  7. Advance the marker right by one position (`idx = idx+1`)
8. Return the value of the marker (`idx`)



When the above algorithm is applied to find the number fourteen in the list show below it will result in a value of 5. This is the index position of the target element in the list. (Recall, that a list index is a zero-based positional offset.)

0	1	2	3	4	5	6	7
15	4	41	13	24	14	12	21

It is important to note that when the target value is not found in the list, the algorithm returns the length of the list. For example, if the algorithm was applied to find the number 22 in the above list the result will be 8 (because the length of this list is 8). When a target value is found in a list, the search operation is said to be successful; otherwise unsuccessful.

Unsuccessful searches can be inferred by the calling code simply by comparing the returned value to the list length. If the value returned by the linear search algorithm is equal to the list length, then the code can deduce that the search was unsuccessful. (This is because list lengths are one-based i.e. the length of a list is always one more than the index of the final element.

In summary, the linear search algorithm works by starting at the first list element and working its way from left-to-right, it compares each element with the target value until either a match is found or the end of the list has been reached.

Some advantages and disadvantages of the linear search algorithm are as follows:

### *Advantages*

1. Simplicity. The linear search is intuitive to most. It is relatively easy to understand and implement.
2. It does not require the data to be stored in any particular order.

### *Disadvantage*

The main disadvantage of the linear search algorithm lies in its lack of efficiency. The more elements there are in a list the greater the amount of time it will take to search for any specific element. In fact, the amount of time it takes to find a target value increases in proportion to the number of elements in the list to search. Therefore, it will take ten times longer to find an element in a list of 1,000 elements than it would for a list of 100 elements. This is called linear time complexity, or  $O(n)$  for short.

One Python implementation of the linear search algorithm is shown in the code below.

```
def linear_search_v1(v, L):  
    i = 0  
    while i < len(L): # more?  
        if L[i] == v: # match?  
            return i # successful  
  
        i = i + 1  
  
    return i # unsuccessful
```

The function `linear_search_v1` is defined to return the position of target value, `v` in list, `L` if successful; otherwise the length of the list will be returned.

The algorithm can be tested using the following driver code – the user is prompted to enter a target value to search for. This is stored in the variable, `argument`.

```
# Driver code ...
keys = [15, 4, 41, 13, 24, 14, 12, 21]
argument = int(input("Enter a target value: "))

result = linear_search_v1(argument, keys)

if (result != len(keys)):
    print("%d found at position %d" %(argument, result))
else:
    print("%d not found. Return value is %d" %(argument, result))
```

Some sample runs are illustrated below:

```
>>> %Run linear_searches.py
Enter a target value: 14
14 found at position 5

>>> %Run linear_searches.py
Enter a target value: 15
15 found at position 0

>>> %Run linear_searches.py
Enter a target value: 12
12 found at position 6

>>> %Run linear_searches.py
Enter a target value: 22
22 not found. Return value is 8
```

A number of common variations on this implementation of the linear search algorithm exist. Some of these variations are shown on the next page.

Version 2 of our linear search algorithm uses a Boolean variable called `match` to indicate whether a match has been found (or not) by the algorithm. Initially, `match` is set to `False` and the search continues as long as it remains `False` (i.e. `not match` will be `True` when `match` is `False`) and there are more elements to compare (i.e. `i < len(L)`).

```
def linear_search_v2(v, L):  
    i = 0  
    match = False  
  
    while not match and i < len(L):  
        if L[i] == v: # match?  
            match = True  
        else:  
            i = i + 1  
  
    return i
```

This next version is a refinement on the one above. Basically, the logic for finding a match and testing for the end of the list are combined into one Boolean expression which becomes the loop guard. The need for an additional `if-else` test inside the loop is removed. The elegance of this solution lies in the fact that the loop body needs only to contain a single statement (`i = i + 1`) to advance to the next element.

```
def linear_search_v3(v, L):  
    i = 0  
    while i < len(L) and L[i] != v: # more? and match?  
        i = i + 1  
  
    return i
```

Version 4 of our algorithm shown below uses a `for` loop instead of a `while` loop. Notice that `len(L)` is returned in this version to indicate that the search was unsuccessful.

```
def linear_search_v4(v, L):  
    for i in range(len(L)):  
        if L[i] == v:  
            return i  
  
    return len(L)
```



This next version – perhaps the simplest of all – uses a `for` loop and exploits the Python `'in'` operator.

```

def linear_search_v5(v, L):
    i = 0
    for element in L:
        if element == v:
            return i
        i = i + 1

    return len(L)
  
```

One interesting question worth exploring is:

*How could the linear search algorithm be improved if it was known that the list to be searched was already sorted?*

Finally, it is worth noting linear search can be implemented recursively as follows:

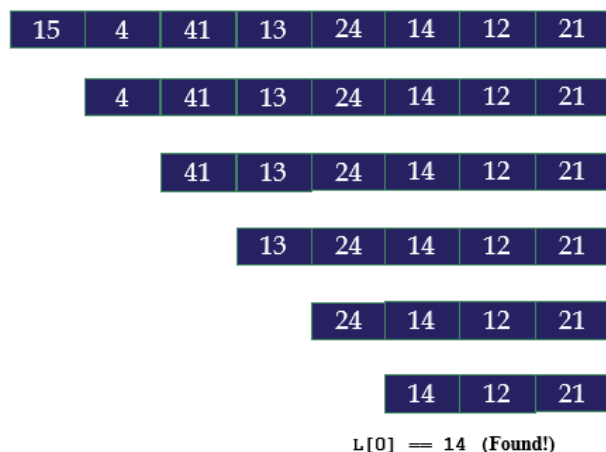
```

def linear_search_v6(v, L, index=0) :
    if len(L) != 0:
        if L[0] == v:
            return index

        r = linear_search_v6(v, L[1:], index+1)
        if r != -1:
            return r

    return -1
  
```

The sequence of lists passed into the recursive function are stacked as show. (This is based on the same example we used earlier i.e. the target value is 14.)



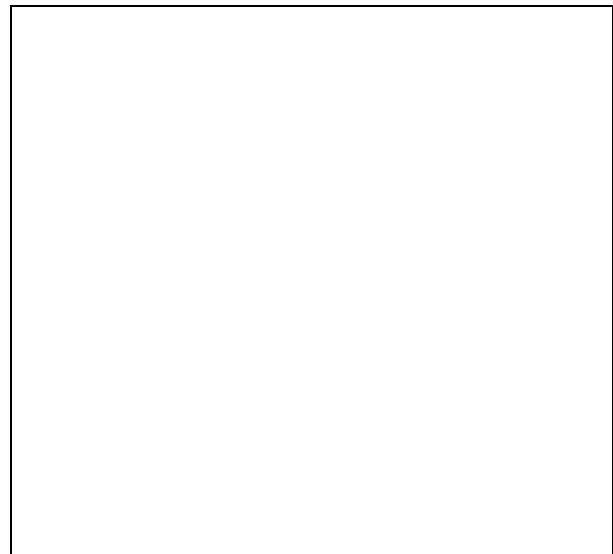
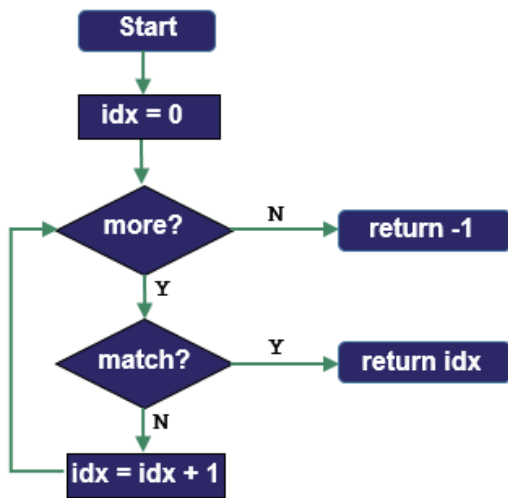
The approach taken is to compare the target value, `v` with first element in `L`. If element is found at the first position (`L[0]`), the index is returned.

Otherwise, recur for the remainder of the list (`L[1:]`).

## Exercise

Use the flowchart below to explain the process of finding the number 26 in the following list of values:

idx	0	1	2	3	4	5	6
values	21	17	-1	26	22	-5	24



Explain the meaning of *more?* in the above flowchart?

Explain the meaning of *match?* in the above flowchart?

## Binary Search

Many people are familiar with the following (guessing) game.

Think of a number between 1 and 32. Now ask someone to guess the number you are thinking of. In each turn, if the guess is not correct, tell your opponent whether the number is too high or too low and ask them to try again. Keep going until he or she guesses your number. How many guesses did it take? Go again. Play the game a few times taking note of the number of guesses it took to find the secret number each time.

Can you explain why the maximum number of guesses it will take to correctly guess any number you can think of between 1 and 32 would be 5? Or is it 6? What if the problem space was doubled i.e. how many guesses would be needed to guarantee success for any number between 1 and 64?

The strategy used by most in the above game is the same strategy employed by the binary search algorithm. It is also the same strategy that people would have used to look up telephone numbers from an alphabetically sorted list of names contained in what was called a phone book back in the 20<sup>th</sup> century!

The binary search algorithm is an example of a *divide-and-conquer* algorithm. Divide-and-conquer is problem solving technique which works by repeatedly reducing the problem (divide) and then attempting to solve the problem (conquer) on the new problem space. In this case the approach is to repeatedly divide the portion of the list that could contain the item in two (i.e. half), until either the item is found or the list cannot be divided any further.

Instead of testing the list's first element, the binary search starts with the element in the middle. If that element happens to contain the target value, then the search is over. If the target value is less than the middle element of the list, we restrict the search to the first half of the list; otherwise we search the second half of the list. Either way, half of the list's elements are eliminated from further searching on each iteration and the procedure is repeated for the half of the list that potentially contains the value. This process continues until the value being searched for is either found, or there are no more elements to test.

Donald Knuth is famously quoted as saying that *an algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it*. So let's put Knuth's advice to practice and try the binary search algorithm.

## Binary search pseudo-code

The pseudo-code for the binary search algorithm is as follows:

```

1. Set low = 0
2. Set high = length of list - 1
3. Set mid =  $\frac{\text{low} + \text{high}}{2}$ , rounded down to an integer
4. If the value at the mid position is the same as the target value
   Return mid
   Else If the value at the mid position is less than the target value
   Set low = mid + 1
   Else If the value at the mid position is greater than the target value
   Set high = mid - 1
5. As long as low doesn't 'cross over' high, go back to step 3 above
6. Return -1

```

Let's say we were tasked with applying the above algorithm to search for a target value of 28 in the following list of 16 values. Notice the index numbers from 0...15 are displayed over each list element and, crucially, that the list has already been sorted.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

In the first three steps of the algorithm we set the variables `low`, `high` and `mid` to 0, 15 and 7 respectively.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37
							↑								↑
							low								high
								↑							
								mid							

We now move to line 4 of the algorithm and since 14 is less than 28 we change the value of `low` to `mid+1` which is 8. The value of `mid` is computed to be  $(8 + 15)/2$  which is 11 (rounded down). Our state now look like this.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37
								↑			↑				↑
								low			mid				high

Since 25 is less than 28 we change the value of `low` again, this time to 12. The new value for `mid` becomes 13 and the state can be visualised as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

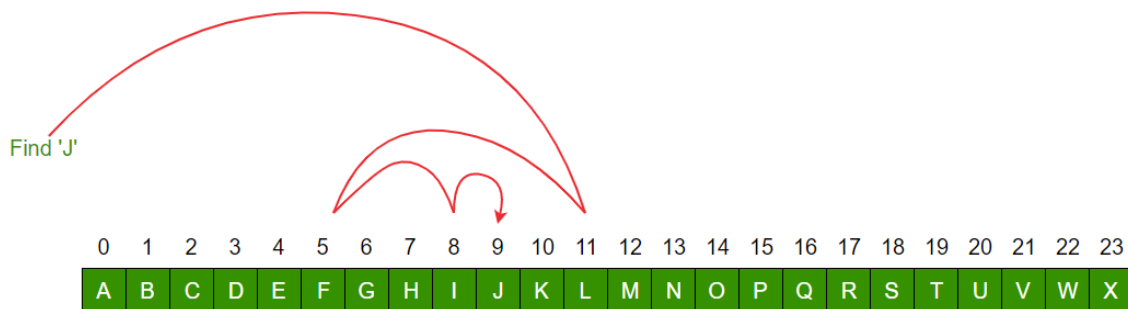
↑     ↑     ↑  
 low  mid    high

Since the next comparison finds the target value, the algorithm can terminate successfully.

The use of trace tables can be very helpful in carrying out a binary search. A trace table for this example might look as follows:

low	mid	high	Rough work
0	7	15	L[7] is 14. 14 < 28 so move <code>low</code> to the right of <code>mid</code> and re-compute <code>mid</code> mid now becomes 11
8	11	15	L[11] is 25. 25 < 28 so move <code>low</code> to the right of <code>mid</code> and re-compute <code>mid</code> mid now becomes 13
12	13	15	L[13] is 28. Found!

The graphic below taken from [geeksforgeeks.org](http://geeksforgeeks.org) is a nice illustration of how the binary search finds the letter 'J' in the list made up of the first 24 letters of the alphabet ('A' – 'X')



A Python implementation of the binary search algorithm is shown below in the function `binary_search`. The function is defined to return the position of some target value, `v` in a list, `L` if successful; otherwise the length of the list will be returned.

```
def binary_search(v, L):

    low = 0
    high = len(L)-1

    while (low <= high):
        mid = (low+high)//2

        if L[mid] == v:
            return mid
        elif L[mid] < v:
            low = mid + 1
        else:
            high = mid - 1

    return len(L)
```

The algorithm can be tested using the driver code shown below. The list, `keys` is first initialised. The user is then prompted to enter a target value to search for. This is stored in the variable, `argument`.

```
# Driver code ...
keys = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
argument = int(input("Enter a target value: "))

result = binary_search(argument, keys)
if (result != len(keys)):
    print("%d found at position %d" %(argument, result))
else:
    print("%d not found. Return value is %d" %(argument, result))
```

Some sample runs are shown below.

Sample Run #1 Look for <code>v</code> , 28 in <code>L</code>	Enter a target value: 28 28 found at position 13
Sample Run #2 Look for <code>v</code> , 57 in <code>L</code>	Enter a target value: 57 57 not found. Return value is 16

### Exercise

Given the list,  $L$  of sixteen integers shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Describe the binary search path to search  $L$  for the following target values,  $v$ .

- a) 19      b) 12      c) 15

A trace table with the initial values of `low`, `mid` and `high` already filled in is provided to get you started.

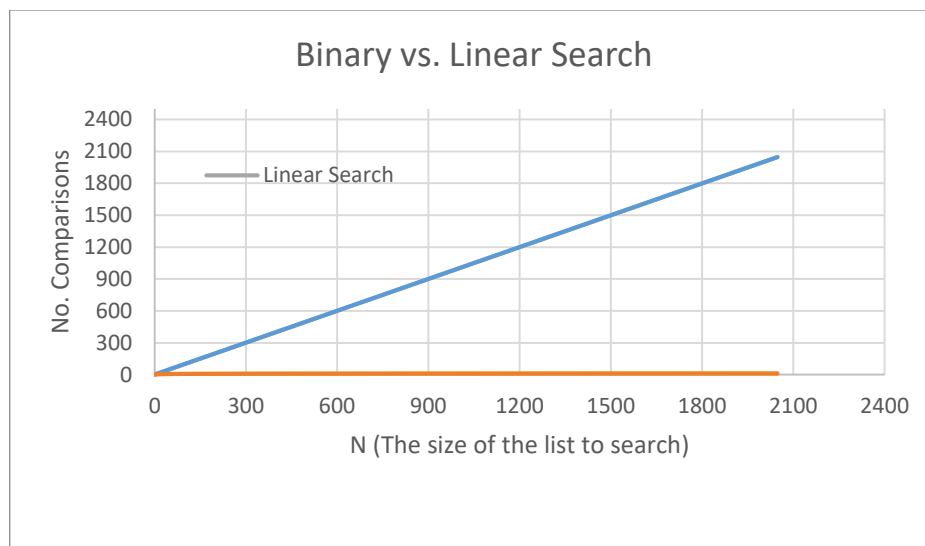
low	mid	high	Rough work
0	7	15	

The main advantage and disadvantage of the binary search are as follows.

### Advantage

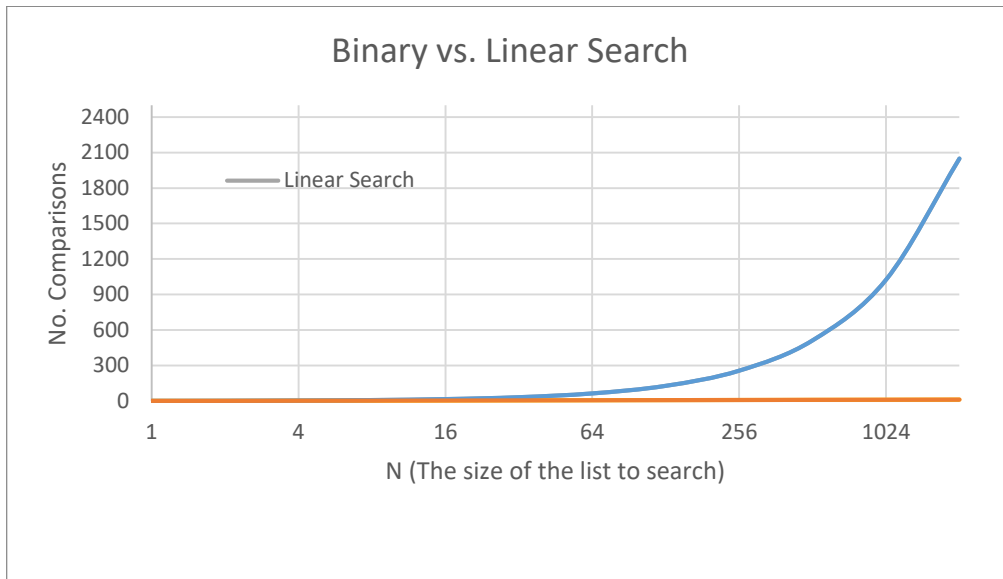
The binary search is much a more efficient algorithm than the linear search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 1,000 elements. If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 500. If the item is not found on the second attempt, the number of elements that remains to be searched is 250. This process continues until the binary search has either located the desired item or determined that it is not in the array. With 1,000 elements this takes no more than 10 comparisons. Compare this to the performance of the linear search which for this scenario would need to make an average number of 500, and a worst case of 1,000 comparisons to achieve the same result.

The following charts illustrate how the two search algorithms stack up against each other in terms of performance. We are already aware that the performance of the linear search increases in proportion to the number of items in the list to search. This linearity is clearly shown by the blue line below. However, notice how the performance cost of the binary search (shown by the brown line) barely rises above the x-axis using this scale.

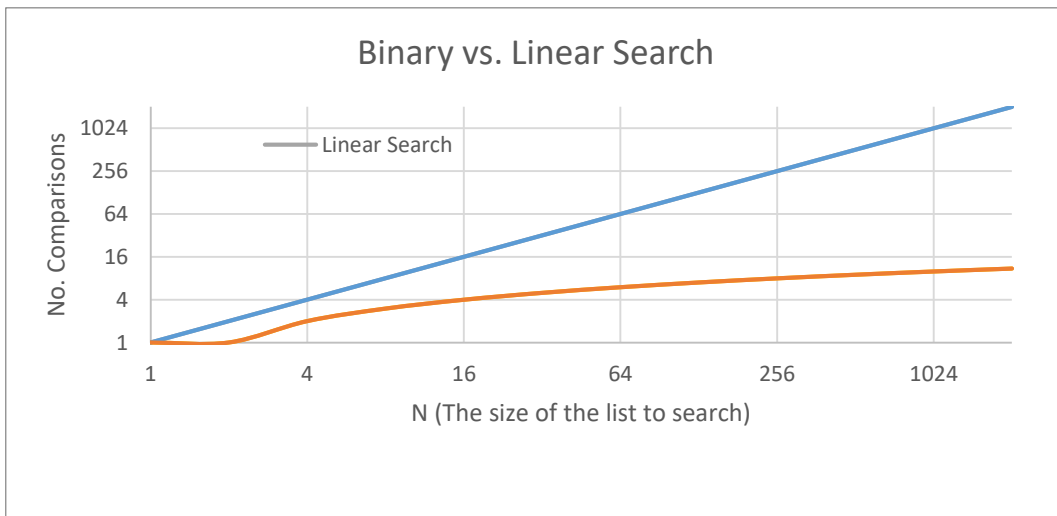


The next graph shows the same data but this time the x-axis is scaled logarithmically. Again the rise in cost of the binary search is barely noticeable as the size of the list grows. Notice, however that the cost of the linear search appears to grow exponentially with respect to the size of the list to search.





The final graphs shown below uses a log-log scale i.e. both x- and y-axes are scaled logarithmically.



Here we can finally see the true logarithmic nature of the efficiency of the binary search emerge. In particular, notice that the performance of the binary search is a logarithmic function of the size of the problem space. Furthermore, the graph is evidence that *binary search is exponentially faster than its linear counterpart.*

### *Disadvantage*

The main drawback of the binary search is that the elements must be sorted beforehand.

## Recursive Implementation

The code below shows a recursive implementation of the binary search. The function searches for  $v$  in  $L$  between  $L[\text{low}]$  and  $L[\text{high}]$ .

```
def recursive_binary_search(v, L, low, high):  
  
    # JE: Uncomment this next line to see the search space  
    #print("v(%d) L(%s) low(%d) high(%d)" % (v, str(L[low:high+1]), low, high))  
  
    if low > high:  
        return len(L) # Not Found!  
  
    mid = (low + high)//2  
  
    if v == keys[mid]: # Found!  
        # v is at mid in L so breakout of recursion  
        return mid  
  
    elif v < keys[mid]:  
        # v is in the lower half of L so recur on L up to mid-1  
        return recursive_binary_search(v, L, low, mid-1)  
  
    # v is in the upper half of L so recur on L from mid+1  
    return recursive_binary_search(v, L, mid+1, high)
```

As is the case with all recursive algorithms there is a base case and a reduction step. In the base case the function returns without making a recursive call, and in the reduction step the function makes a recursive call (i.e. it calls itself) and in so-doing moves one step closer to the base case.

In this example, there are two base cases as follows:

- 1) The list is empty (this occurs when  $\text{low} > \text{high}$ ) and
- 2) The middle element in the list is the value being searched for

The recursive call depends on the outcome of a comparison between the middle element in the list being searched and the target value:

- if the target value is less than the middle element the function recurs on first (lower) half of the list i.e. `recursive_binary_search(v, L, low, mid-1)`
- if the target value is greater than the middle element the function recurs on second (upper) half of the list i.e. `recursive_binary_search(v, L, mid+1, high)`

The recursive binary search algorithm can be tested using the driver code shown below. The list, `keys` is first initialised. The user is then prompted to enter a target value to search for. This is stored in the variable, `argument`.

The initial call to the recursive function to search for `argument` in `keys` is highlighted in bold. Note that the search is confined to work within the index range that is specified by the last two arguments i.e. 0 and 15 in the case of this example.

```
# Driver code ...
keys = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
argument = int(input("Enter a target value: "))

result = recursive_binary_search(argument, keys, 0, len(keys)-1)

if (result != len(keys)):
    print("%d found at position %d" %(argument, result))
else:
    print("%d not found. Return value is %d" %(argument, result))
```

Three separate sample runs to search for 14, 28 and 38 in `keys` are shown below. The values of variables, `v`, `L`, `low` and `high` at each step of the recursion process are shown for information purposes.

```
Enter a target value: 14
v(14) L([2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]) low(0) high(15)
14 found at position 7
```

```
Enter a target value: 28
v(28) L([2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]) low(0) high(15)
v(28) L([17, 19, 22, 25, 27, 28, 33, 37]) low(8) high(15)
v(28) L([27, 28, 33, 37]) low(12) high(15)
28 found at position 13
```

```
Enter a target value: 38
v(38) L([2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]) low(0) high(15)
v(38) L([17, 19, 22, 25, 27, 28, 33, 37]) low(8) high(15)
v(38) L([27, 28, 33, 37]) low(12) high(15)
v(38) L([33, 37]) low(14) high(15)
v(38) L([37]) low(15) high(15)
v(38) L([]) low(16) high(15)
38 not found. Return value is 16
```

## Activity #2: Developing an understanding of basic sorting algorithms

The main objective of this activity is that each participant gains a procedural understanding of the simple (selection) sort, the insertion sort and the bubble sort algorithms.

For this activity participants are divided into groups (4 individuals per group is ideal) and each group is assigned with an initial algorithm to study.

**Instructions :**


1. Read the algorithm provided and make notes in your workbook
2. Each table to agree common understanding of their assigned algorithm and prepare a demonstration which they will use to teach others
3. Participants move around the room (in teams of 2) explaining and demonstrating the algorithm they have learned to others

Simple  
(selection)  
Sort

Insertion  
Sort

Bubble  
Sort

Activity to be followed by a short discussion at the end.



### Stages 1 and 2 (15 minutes)

Everyone spends five minutes reading the assigned algorithm to themselves.

In the next five to ten minutes the algorithm is discussed in groups. The aim of this discussion is to ensure that everyone has a concrete understanding of how the algorithm works – points of confusion are clarified and a strategy for explaining how the algorithm works to others is agreed upon.

### Stage 3 (10 minutes x 2)

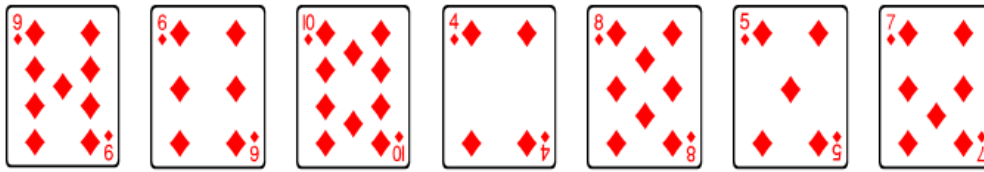
Two people (pairs) from each group remain at their original table while the other pair move to the another table (e.g. 1 ↔ 4; 2 ↔ 5; 3 ↔ 6). The pairs explain/demonstrate their algorithms to one another (no more than 5 minutes per pair!).

This is repeated once so that everyone has had an opportunity to learn each of the three elementary sorting algorithms.

**A detailed description of each algorithm is provided elsewhere in this manual.**

## Activity 2.1 The Simple (Selection) Sort

Let's say we're tasked with sorting the values of some list,  $L$ , arranged as follows:

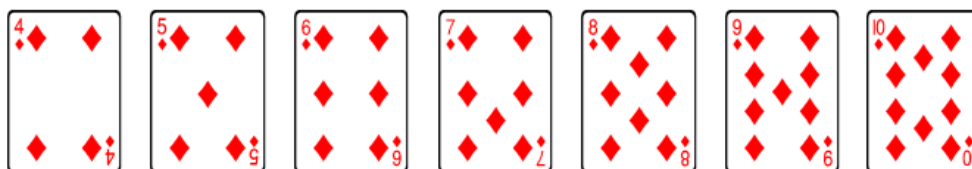


Place your index finger as a marker under the first element (i.e. the 9 of diamonds) and proceed as follows:

- find the smallest value to the right of your marker and swap the two values
- move your marker (index finger) one place to the right
- repeat this process until the marker reaches the end of the list

Use the space below to trace the state of the list as you progress:

When you reach the end the list should be sorted as follows:



Use the space below to describe your own understanding of how the simple (selection) sort algorithm works.

## Activity 2.2 The Bubble Sort

The bubble sort repeatedly ‘bubbles’ larger items towards the (sorted) end of the list. Given an unsorted list,  $L$  as input:



The table below depicts the state of  $L$  at the end of each pass of the bubble sort algorithm.

After Pass #	State of List (at the end of the pass)	Notes (what exchanges take place?)
1		
2		
3		
4		
5		

How many exchanges would take place if the initial list was:

**a)** already sorted, **b)** in reverse order?

a) Already Sorted e.g. [1, 2, 3, 4, 5]

b) Reverse Order e.g. [5, 4, 3, 2, 1]

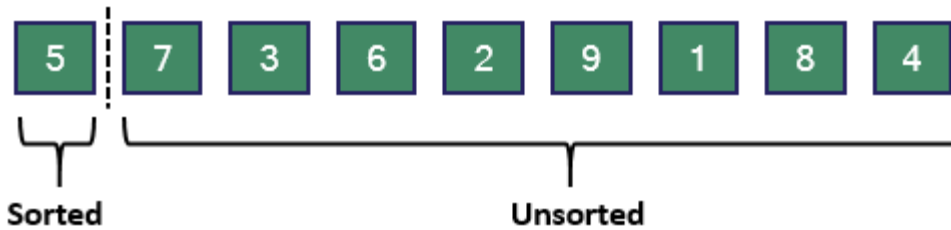
Use the space below to explain how the bubble sort algorithm works:



## Activity 2.3 The Insertion Sort

In any list the first item is always considered sorted with respect to all the items to its left. Then working from left to right each subsequent item is inserted into its correct place with respect to the previously sorted items.

Follow the instructions below to sort the following list:



Place your index finger as a marker under the first item in the unsorted list (i.e. in this case the first selected item will be 7) and proceed as follows:

- insert the selected item into its correct place within the sorted list (to the left). This is done by repeatedly swapping back (leftwards) with all larger neighbours to the left
- move your marker (index finger) one place to the right (the next selected item in this example will be 3)
- repeat this process until the marker reaches the end of the list

Use the space below to trace the state of the list at the end of each pass:

The final sorted list will look like this:



Use the space below to explain how the insertion sort algorithm works:

## Section 3 - Analysis of Algorithms

### Introduction to Algorithmic Efficiency (Complexity)

Now that we have developed an understanding of how some search/sort algorithms work, the next logical step is to examine just how *well* they work. In this section we will analyse the performance of algorithms. In computer science this is often referred to as algorithmic efficiency or complexity. Two common measures of algorithmic efficiency are space and time – the former provides an indication of the demands an algorithm places on memory in terms of space requirements, while the later focuses on the time requirements of an algorithm. For the most part, we will be confining the remainder of our discussion to time complexity.

The study of time complexity provides us with a framework which can be used to compare algorithms and understand how *well* they perform in relation to one another. Before we can begin to compare algorithms in terms of their performance however, we must first devise (or at least agree upon) some system that is both impartial and reliable.

On the surface it might seem fair and make sense to simply time how long it takes an algorithm to run in minutes and seconds (or milliseconds) and use this as a measure of performance. As it turns out however this would be neither fair nor reliable. This is because a computer's performance can depend on a variety of different factors (e.g. processor clock speed, word size, bus width and amount of available memory), and so, an algorithm that takes 1000 milliseconds to run on one computer might run in just 10 milliseconds on another (one hundred times faster!). In fact, depending on the processor load, the time taken to run an algorithm could potentially vary significantly from run to run on the same processor.

Furthermore, the running time of an algorithm is likely to vary in accordance with the size of its input. Intuitively it is easy to understand that a particular sorting algorithm will sort 1,000 integers must faster than it will sort 1,000,000 integers. However, as we will soon learn to appreciate (hopefully!), it is the specific techniques and nuances employed by algorithms that have a much greater bearing on performance than the size of the input.

And then there are questions such as what is the fastest time an algorithm can run in i.e. what is the best case performance? Or is there an average performance time for a particular algorithm? What about a worst case?

As it turns out it is this final question (regarding worst case) that computer scientists are most interested in. The reason for this is that a worst case running time gives users a bottom line guarantee that an algorithm will finish at worst within a particular timeframe, and for this reason worst case scenario is used as a metric for comparing algorithms.

From the preceding section it should be evident that something other than exact running time as a metric for time complexity is needed. That something is Big-O.

## Big O

Big O is a notation used in Computer Science to describe the worst case running time (or space requirements) of an algorithm in terms of the size of its input usually denoted by  $n$ .

By using Big-O notation, algorithms can be broadly classified into one of the groups described below. The running time (or space requirements) of algorithms within the same classification is of the same order of growth with respect to  $n$ .

The imprecise nature of Big-O is important to understand from the outset. For example, an algorithm found to take  $2n^3 + n^2 - 4n + 3$  time to complete would be described as having a complexity of  $O(n^3)$ . This is because the higher order term will dominate the other terms for sufficiently large values of  $n$ . The lower order terms and constant value can therefore be ignored. Big-O provides an *order of magnitude* and can be thought of as a qualitative descriptor as much as a quantitative one.

A description and examples of some common Big-O values is now presented.

### $O(1)$

An algorithm described in this manner will always run within some *constant time* (sometimes called bounded time) regardless of the size of the input. Such algorithms are said to take 'order of 1', or  $O(1)$  time to complete.

While it is possible that two different  $O(1)$  algorithms may take significantly different times to complete this does not matter. The important point is that we know that  $O(1)$  algorithms will complete within some constant time.

To take an analogy, let's say it's the weekend and you were preparing to do some serious study but before you get started you first need to clear your room/desk. The time required to

do this work doesn't depend in any way on the number of subjects you intend to study. It will be completed within some constant amount of time regardless of whether you will study two or ten subjects.

### $O(n)$

If the length of time it takes to run an algorithm increases in proportion to the size of the input the algorithm is said to run in *linear time*. Such algorithms have an  $O(n)$  complexity.

The linear (sequential) search algorithm used to find some target value (argument) in a list that contains  $n$  values is a classic example of an  $O(n)$  algorithm. This is because in the worst case scenario every element in the list will have to be examined in order to find the target value. These algorithms are characterised by the following loop structure.

```
for i in range(n):  
    print(i) # this line will be executed n times
```

Once again it is important to remember that the absolute time is not the important factor. On average it will take much less time to search for a value in shorter lists than longer ones. Recall, Big-O provides us with an objective classification scheme which can be used to compare algorithms based on worst case scenarios.

To continue with our earlier analogy – let  $n$  be the number of subjects you are going to study and let us say that you had decided to allocate a fixed amount of time to each subject. It makes sense therefore that the more subjects you study the longer it will take to finish your study. Twice as many subjects will require twice the amount of time.

### $O(n^2)$

Now let's say that you decided to use a slightly different approach to your study. Instead of allocating the same fixed amount of time to each subject you decide to allocate fixed units of time to reading individual pages of notes. You start by reading one page for the first subject, two for the second, three for the third and so on. By the time you have reached your  $n^{\text{th}}$  subject you will need to read  $n$  pages of notes. The amount of time it takes to complete your study in this case is known as *quadratic time* and is written as  $O(n^2)$ .

Algorithms of this type are characterised by loops nested to one level. For each of the  $n$  iterations carried out by the outer loop, the inner loop will perform  $n$  iterations of its own. This

is illustrated in the snippet of Python code below in which the print statement appears within a nested for loop and will be executed  $n^2$  times.

```
for i in range(n):
    for j in range(n):
        print(i, j) # this line will be executed n squared times
```

The three elementary sort algorithms – selection sort, insertion sort and bubble sort – are all examples of algorithms whose time complexity is quadratic. Furthermore, it is noteworthy that algorithms in this class are impractical to use when it comes to dealing with large volumes of data. Just think about it – if the size of a list doubles it would take four times longer to sort; increasing the size of a list threefold will result in a nine-fold increase in time. Not to labour the point too much, it would take 100 times longer to sort a list of 1000 items than it would to take to sort a list just 10 times smaller. Quadratic time algorithms are simply unsustainable.

### $O(\log_2 n)$

These class of algorithms are said to be logarithmic. For algorithms that have logarithmic time complexity it means that as the value  $n$  increases, the time complexity of your program increases by a logarithmic factor.

Such algorithms are characterised by cutting the size of the input in half in each step as it moves towards a solution. Take for example the following analysis of a binary search:

List Size (n)	Maximum number of comparisons (c)
1	1
2	2
4	3
8	4
16	5
32	6
etc.	etc.

As can be seen from the table above the maximum number of comparisons (steps) the binary search algorithm needs to perform in order to find some target value just increases by one each time the size of the input list is doubled.

The relationship between the size of the input ( $n$ ) and the maximum number of comparisons ( $c$ ) required is given by:

$$n = 2^{c-1}$$

Therefore,

$$c = \log_2 n + 1$$

So binary search has  $O(\log_2 n)$  time complexity which is a very impressive and desirable feature for any algorithm to have. More importantly however is the fact that we can use this to calculate the maximum number of comparisons it will take to search a list of any size i.e. we can guarantee an upper bound. For example a list with  $2^{30} \approx 1 \text{ billion}$  elements will take no more than 31 comparisons. This is very useful information for software designers to have at hand when they need to choose the most appropriate algorithm for the system they are working on.

$O(n \log_2 n)$

When it comes to analysing worst case time complexity of algorithms that sort by using a series of head-to-head comparisons, it is a proven fact that the best we can hope to achieve is  $O(n \log_2 n)$ , also called “linearithmic” time. “Linearithmic” complexity lies somewhere between linear and quadratic. It is not an understatement to say that algorithms with this class of time complexity result in seismic improvements in performance.

“Linearithmic” algorithms are characterised by an approach to problem solving known as *divide-and-conquer*. Depending on the particular algorithm there will be  $n$  divisions and each division will take  $\log n$  steps to conquer or vice versa.

Two examples of algorithms that fall into this class of time complexity are quicksort and merge sort.

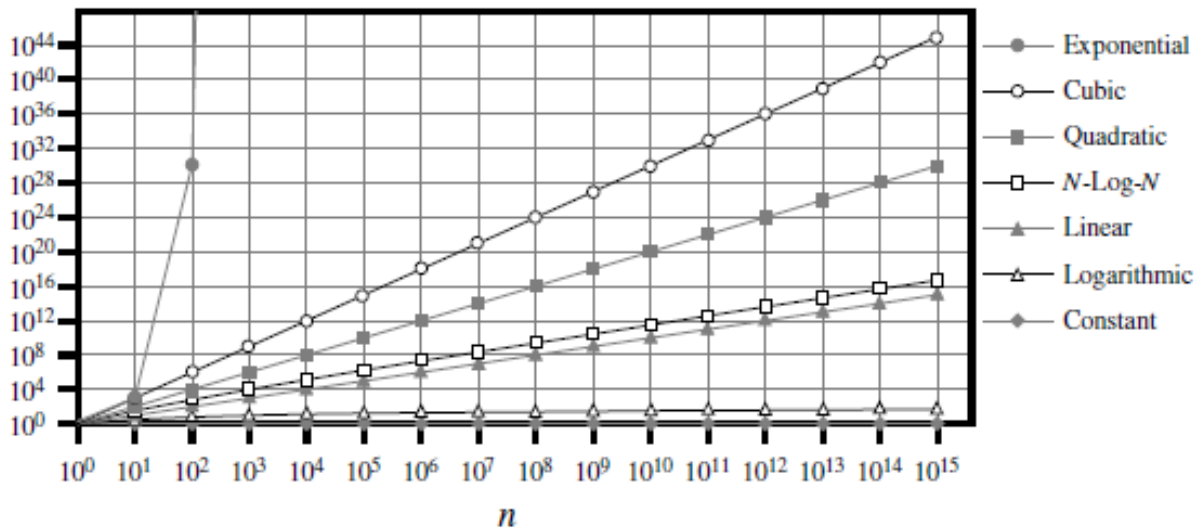
### *Intractable Problems*

Finally, it is worth noting that algorithms can have time complexities that are exponential,  $O(2^n)$  and even worse, factorial,  $O(n!)$ . The solution to the Travelling Salesman Problem is an example of a factorial time algorithm. Algorithms of this nature are said to be *intractable* as their running time makes them infeasible even for very small values of  $n$ . (This is evidenced by the values in the rightmost two columns in the table at the top of the next page.)

## Summary Graphs and Tables

The growth rates in computation time for the common time complexity functions discussed in the preceding section are depicted in tabular and graphical<sup>8</sup> format below.

N	Constant	Linear	Quadratic	Logarithmic	Linearithmic	Exponential	Factorial
1	1	1	1	1	1	2	1
2	1	2	4	1	2	4	2
4	1	4	16	2	8	16	24
8	1	8	64	3	24	256	40320
16	1	16	256	4	64	65536	2.09228E+13
32	1	32	1024	5	160	4294967296	2.63131E+35
64	1	64	4096	6	384	1.84467E+19	1.26887E+89
128	1	128	16384	7	896	3.40282E+38	3.8562E+215



<sup>8</sup> Source: Data Structures and Algorithms in Python (Goorich et. al., Page 122)



This table summarises the time complexities of some common search and sort algorithms.

	<b>Best Case</b>	<b>Average Case</b>	<b>Worst Case</b>
<b>Linear Search</b>	$O(1)$	$O(n)$	$O(n)$
<b>Binary Search</b>	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
<b>Simple (selection) Sort</b>	$O(n^2)$	$O(n^2)$	$O(n^2)$
<b>Bubble Sort</b>	$O(n)$	$O(n^2)$	$O(n^2)$
<b>Insertion Sort</b>	$O(n)$	$O(n^2)$	$O(n^2)$
<b>Quicksort</b>	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$

## Activity #3: Analysis of Algorithms

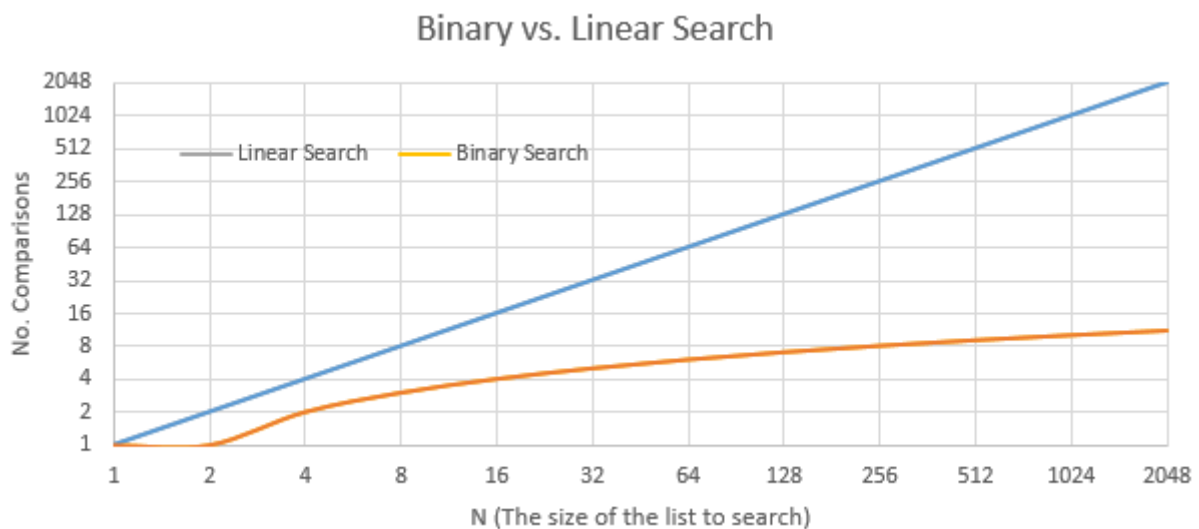
The main objective of this task is to enable participants to plan and carry out their own detailed analysis of algorithms.

In this activity teachers will work in pairs (pair programming) – all pairs are assigned the same task outlined below.

Each pair opens the assigned Python analysis framework

The code is run and adapted according to the instructions provided on the following pages in order to complete each of the assigned tasks.

The task is to use the analysis framework to test the assertion that the linear search is exponentially slower than the binary search.



**A detailed description of each algorithm is provided elsewhere in this manual.**

## Task A: Analysis of Search Algorithms

### Linear Search

The code below defines a function called `linear_search`. The function works by looking for some value  $v$  in a list  $L$ . If  $v$  is found, the index of its position in  $L$  is returned; otherwise the function returns `-1`.

```
import random

def linear_search(v, L):
    global comparisons

    for index in range(len(L)):
        comparisons = comparisons + 1
        if L[index] == v:
            return index

    return -1 # not found

# Driver code ...
print("%s\t\t %s\t\t %s" %("List Size", "Found Index", "#Comparisons"))
for list_size in [1, 10, 100, 1000, 10000, 100000, 1000000]:
    some_list = list(range(list_size))
    random.shuffle(some_list) # randomise the list
    comparisons = 0
    target = -1 # worst case because -1 never exists
    pos = linear_search(target, some_list)
    print("%d\t\t %d\t\t %d" %(len(some_list), pos, comparisons))
```

1. Run the above program and record your output in the table below.

List Size	Found Index	#Comparisons
1		
10		
100		
1,000		
10,000		
100,000		
1,000,000		

2. Based on what you have learned from the previous question:

a) what is the worst case time complexity for the linear search?

b) explain the significance of the following line in the code:

```
random.shuffle(some_list) # randomise the list
```

3. Replace the line:

```
target = -1 # worst case because -1 never exists
```

with the line:

```
target = random.randrange(len(some_list)) # average case
```

Run the program again and record your output in the table below.

List Size	Found Index	#Comparisons
1		
10		
100		
1,000		
10,000		
100,000		
1,000,000		

4. Based on the previous question what is the average case time complexity for the linear search? (See: <https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/>)

## Binary Search

The code below consists of a definition of a binary search function (`binary_search`) and a test harness used to test this function. The function works by looking for some value `v` in a list `L`. If `v` is found, the index of its position in `L` is returned; otherwise the function returns `-1`.

Read the program carefully.

```
import random

def binary_search(v, L):
    global comparisons
    low = 0
    high = len(L)-1

    while (low <= high):
        index = (low+high)//2
        comparisons = comparisons + 1

        if L[index] == v:
            return index
        elif L[index] < v:
            low = index + 1
        else:
            high = index - 1

    return -1

# Driver code (test harness)...
print("%s\t\t %s\t\t %s" %("List Size (N)", "Found Index", "#Comparisons"))
for list_size in [1, 10, 100, 1000, 10000, 100000, 1000000]:
    some_list = list(range(list_size))
    comparisons = 0
    target = -1 # worst case because -1 never exists
    pos = binary_search(target, some_list)
    print("%d\t\t %d\t\t %d" %(len(some_list), pos, comparisons))
```

The program performs a binary search on seven different lists (`some_list`). The length of each list varies, starting at 1 and increasing in powers of 10 up to 1,000,000. The contents of each list is generated using the `range` function which returns a sequence of all the integers from zero up to the size of the list minus 1. So for example, the third list which has a length of 100 will contain the all integers, 0 through to 99.

For each list, a call is made to `binary_search` to look for a `target` value of `-1`. Since this value does not exist, the test harness enables us to count the maximum number of `comparisons` that the binary search makes for lists of different sizes.

1. Run the binary search program and record your output in the table below.

List Size	Found Index	#Comparisons
1		
10		
100		
1,000		
10,000		
100,000		
1,000,000		

2. Replace the line:

```
target = -1 # worst case because -1 never exists
```

with the line:

```
target = random.randrange(len(some_list)) # average case
```

Run the program again and record your output in the table below.

List Size	Found Index	#Comparisons
1		
10		
100		
1,000		
10,000		
100,000		
1,000,000		

3. What conclusion (if any) can you draw from the above results?

4. Use a calculator (or some other means) to complete the table below.

<b>N</b>	<b><math>\log_2 N</math></b>
1	
10	
100	
1,000	
10,000	
100,000	
1,000,000	

5. Repeat question 3.

6. What is meant by the best case time complexity of a search algorithm?

7. Record the best, average and worst case time complexities for linear and binary searches in the table below.

	<b>Best Case</b>	<b>Average Case</b>	<b>Worst Case</b>
<b>Linear Search</b>			
<b>Binary Search</b>			

8. Modify the search programs provided so that you can complete the table below.  
How do the linear and binary search algorithms behave when the list size doubles?  
Consider cases where the target element exists and does not exist.

List Size	Binary Search		Linear Search	
	Target Does Not Exist	Target Exists	Target Does Not Exist	Target Exists
1				
2				
4				
8				
16				
32				
64				
128				
256				
512				
1024				
2048				
4096				
8192				
16384				
32768				
65536				

9. Plot a graph of the above results.  
(You could use a spreadsheet package or a Python library)

10. In what situations (if any) does the linear search outperform the binary search



## Task B: Analysis of Sorting Algorithms

### Exercise 1: Simple (Selection) Sort

In this exercise participants explore the number of comparisons and exchanges made for the simple (selection) sort algorithm.

```
import random

def simple_selection_sort(L):

    swaps = 0
    comparisons = 0
    #print("Before: ", L)

    # Traverse over all list elements
    for i in range(len(L)):

        # Find the minimum to the right of i
        min_idx = i
        for j in range(i+1, len(L)):
            comparisons = comparisons + 1
            if L[j] < L[min_idx]:
                min_idx = j

        # Swap minimum element with the current element
        L[i], L[min_idx] = L[min_idx], L[i]
        swaps = swaps + 1

    #print("After: ", L)
    print("N(%d), #Comparisons (%d), #Swaps (%d) "%(len(L), comparisons, swaps))

# Driver code ...
# ... run this code for a list sizes 5, 10, 100 and 1000
# ... run for already sorted, reversed and randomised lists
# ... record the #comparisons and #swaps in the manual
L = list(range(5)) # generate the ordered list
#L.reverse() # uncomment this line to reverse the list
#random.shuffle(L) # uncomment this line to randomise the list

simple_selection_sort(L)
```

Take some time to study the code and understand how the algorithm works.

When the above listing is run it will display the number of comparisons and the number of swaps performed by the selection sort on a list of five already sorted elements (L).

1. Run (and re-run) the program on the previous pages to record the number of comparisons and swaps performed by the simple (selection) sort algorithm on ordered, reversed, and randomised lists of lengths 5, 10, 100 and 1000. (Twelve runs.)

Record your results in the table below.

Notes:

- 1) To change the list length, modify the argument passed into the `range` function:
- 2) To reverse the list, just uncomment out the following line:  
`#L.reverse()`
- 3) To randomise the list, just uncomment out the following line:  
`#random.shuffle(L)`

List Length (N)	List already Sorted		Initial List Reversed		Randomised List	
	#Compares	#Swaps	#Compares	#Swaps	# Compares	#Swaps
5						
10						
100						
1000						

(Aside. The program was kept as simple as possible for clarity purposes. Of course, it would be possible, and maybe even worthwhile, to modify the code so that all of the required information can be generated in a single run.)

2. Use the above results to investigate the proposition that the simple selection sort uses  $\sim N^2/2$  comparisons and  $N$  swap operations to sort a list of length  $N$ .

## Exercise 2: Insertion Sort, Bubble Sort and Quicksort

Add definitions for these three sort algorithms to the program used for exercise 1. Now run the program in the same manner as you did for the previous exercise and complete the tables below for each of the sort algorithms.

### A. Insertion Sort

List Length (N)	List already Sorted		Initial List Reversed		Randomised List	
	#Compares	#Swaps	#Compares	#Swaps	# Compares	#Swaps
5						
10						
100						
1000						

### B. Bubble Sort

List Length (N)	List already Sorted		Initial List Reversed		Randomised List	
	#Compares	#Swaps	#Compares	#Swaps	# Compares	#Swaps
5						
10						
100						
1000						

### C. Quicksort

List Length (N)	List already Sorted		Initial List Reversed		Randomised List	
	#Compares	#Swaps	#Compares	#Swaps	# Compares	#Swaps
5						
10						
100						
1000						

Now answer the following questions.

- Use your data to verify the following for the insertion sort algorithm.

	Comparisons	Swaps	Notes
Best	$N - 1$	0	
Average	$\sim \frac{N^2}{4}$	$\sim \frac{N^2}{4}$	
Worst	$\sim \frac{N^2}{2}$	$\sim \frac{N^2}{2}$	

- Explain why, on average, the insertion sort is (slightly) faster than the simple (selection) sort.

- State which of the three algorithms you would recommend if the initial list was:
  - already (or almost) fully sorted
  - in reverse order
  - very large (made up of millions of elements)

## Section 4

### **Critical Reflection**

*Thoughts on unconscious bias and algorithmic bias*

The following warmup activities are based on research work carried out at The Centre for Advancement of Science and Mathematics Teaching and Learning (CASTeL) in Dublin City University.

#### **Warmup Activity #1**

Words that come to mind when you think about teenage girls.

Words that come to mind when you think about teenage boys.



**Warmup activity #2**

Read the following scenario once and complete the short survey that follows.

A builder, leaning out of the van, shouts “nice legs” to a nurse passing by. The same nurse arrives at work, and casually mentions this to a senior doctor. The doctor said, “I’d never say that”. The doctor has two grown up children who are 22 and 30. They get on very well. One is a Sergeant in the Army; the other is training to be a beauty therapist. The doctor divorced last year and is currently dating someone else.

Now complete the following survey. Tick the appropriate box for each statement ...

	<b>True</b>	<b>False</b>	<b>Don't know</b>
The builder was driving a van	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The van was travelling quicker than the nurse	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
There was at least one man in the van	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Not every man mentioned would shout ‘nice legs’	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The doctor is no longer living with his wife	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The doctor has a new girlfriend	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The doctor’s son is in the army	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The youngest child is training to be a beauty therapist	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
At some point a man spoke to a woman	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
At least two of the people mentioned are men	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
A woman was shouted at	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

The scenario did not provide enough information to answer *True* or *False* to any of the statements.

	True	False	Don't know
The builder was driving a van			<input checked="" type="checkbox"/>
The van was travelling quicker than the nurse			<input checked="" type="checkbox"/>
There was at least one man in the van			<input checked="" type="checkbox"/>
Not every man mentioned would shout "nice legs"			<input checked="" type="checkbox"/>
The doctor is no longer living with his wife			<input checked="" type="checkbox"/>
The doctor has a new girlfriend			<input checked="" type="checkbox"/>
The doctor's son is in the army			<input checked="" type="checkbox"/>
The youngest child is training to be a beauty therapist			<input checked="" type="checkbox"/>
At some point a man spoke to a woman			<input checked="" type="checkbox"/>
At least two of the people mentioned are men			<input checked="" type="checkbox"/>
A woman was shouted at			<input checked="" type="checkbox"/>

Consider why you might have ticked either *True* or *False* for any of the above statements.

What is **unconscious bias** and why is it important?





## *What is Unconscious Bias?*

- An automatic mental shortcut used to process information and make decisions quickly.

Unconscious Bias is ...

- Natural
- A rapid categorisation of people
- Created by social influence
- Unintentional
- Used by everyone
- Most likely to be acted on when one is stressed or tired
- A bad thinking habit

Attributes that can trigger unconscious bias include gender, ethnicity, religion/belief, perceived sexual orientation, attractiveness, disability, clothing, piercings/tattoos, hairstyle, body language, accent, personality, friends/family, age, height etc.

Some of the factors that contribute to unconscious bias include ...

- Media (traditional and social media)
- Language
- Socialisation (TV, School, Religion, Parents)
- Relationships
- Role Models
- Education
- Patterns in Society

What does unconscious bias have to do with **algorithms**?

## Algorithmic Bias

In an age of Artificial Intelligence (AI) and machine learning fuelled by big data, unconscious bias can be replicated, reinforced and amplified by algorithms. AI systems are not new (recall Alan Turing’s Test from the 1950s and Eliza and other ‘expert systems’ from the 1960s and 70s) and although they have had a chequered history, there seems to be a widespread general recognition that we are on the cusp of entering a ‘golden age’ of machine learning and AI. This has the potential to have a profound impact on our lives and society in general.

In today’s society ordinary everyday decisions traditionally made by humans are increasingly being made by algorithms. In an ideal world we would like to think that these decisions would be made free from any bias. However, this is not always the case – the use of algorithms to sway voters’ opinions and influence the outcomes of elections in various jurisdictions around the world in recent years have been widely documented.<sup>910</sup>

## What does Unconscious Bias have to do with Algorithms?



“some of the biggest problems in the industry aren’t technical – they’re people (egos etc.) .... diversity creates better solutions and opportunities because of wider experience set, perspectives and **less bias** ...

*James Whelton, CoderDojo Co-Founder*

Algorithms are particularly well suited to processing unimaginably large amounts of data – the kinds of data we have come to associate with the ‘infinite scroll’ through pages and pages of social posts and product listings. But how do these algorithms decide to rank content and on what basis do they place material at the top of these lists? Although very few

<sup>9</sup> <https://www.wired.com/story/the-great-hack-documentary/>

<sup>10</sup> <https://www.theguardian.com/world/2018/dec/17/revealed-how-italy-populists-used-facebook-win-election-matteo-salvini-luigi-di-maio>

know the precise answer to this question, it is generally agreed that these ‘recommender’ algorithms favour material that triggers emotional responses. Such systems which reward reaction can lead to extremification, can be divisive, and can result in the polarisation of groups and communities. The power of algorithms, and in particular their influence on our own biases, should not be underestimated.

Further examples of algorithmic bias are unfortunately all too commonplace – can you imagine how the Nigerian man, Chukwuemeka Afigbo felt when he discovered that the automatic hand-soap dispenser did not work for black hands but recognised white hands perfectly<sup>11</sup><sup>12</sup>? Or Jackie Alciné, an Africa American software engineer, who in 2015 discovered that AI systems were tagging images of himself and his friends with the word ‘Gorillas’<sup>13</sup>. And it’s not just black hands and faces. In his book *Coders: The Making of a New Tribe and the Remaking of the World*, Clive Thompson points out how algorithms trained in China, Japan and South Korea have been found to struggle to recognise Caucasian faces but work well on East Asian ones.

The root problem in many of these cases stems from the fact that the samples in the underlying dataset used to train the AI were simply not broad enough. AI facial recognition systems which have been trained using a disproportionate number of white faces are more likely to recognise white faces with a greater degree of accuracy than they are to recognise faces of another skin colour. As Hannah Fry points out in her book, *Hello World: Being Human in the Age of Algorithms*, this can very quickly turn into a self-reinforcing or feedback loop. In situations when the AI is designed to make decisions this can have particularly devastating effects on the lives of individuals.

Take for example a policing algorithm used to predict locations where crime is likely to occur based on statistics from where it has occurred in the past. It is not difficult to imagine how such algorithms could lead to a lop-sided concentration of policing effort being directed towards areas of ethnic minorities and social disadvantage, and how in turn, such an algorithm has the potential to exacerbate certain biases.

Similarly, algorithms designed to ‘assist’ judges when it comes to handing down sentences may well have been trained on pre-existing crime data that has already been ‘contaminated’

---

<sup>11</sup> <https://x.ai/blog/ai-lacks-intelligence-without-different-voices/>

<sup>12</sup> [https://twitter.com/nke\\_ise/status/897756900753891328?lang=en](https://twitter.com/nke_ise/status/897756900753891328?lang=en)

<sup>13</sup> <https://www.theguardian.com/technology/2015/jul/01/google-sorry-racist-auto-tag-photo-app>

with bias. Both Thompson and Fry describe how independent research<sup>14</sup> (ProPublica) found that one such algorithm called COMPASS was almost twice as likely to label a black defendant as getting a high-risk recidivist score than a white defendant (even though the algorithm did not explicitly include race as a factor). Can you imagine the public outrage that followed when the ProPublica report was published in 2016?

Of course there are those who would argue while they might not be biased themselves, the algorithms are merely reflecting realities and therefore might be correct. What if racism or sexism are accurate reflections of the world? Should we allow our AI algorithms to continue to propagate biases or does AI present an opportunity to redress and reduce these biases (an algorithmic version of political correctness, if you will)? Either way there is a moral choice to be made – the decision by AI designers *to ignore bias* in their algorithms must surely be considered every bit as moral as the opposite decision *not to ignore bias* is. Gladly, many companies seem to be taking a proactive approach to dealing with bias. Twitter for example, recently announced they are giving all employees training on unconscious bias<sup>15</sup>. In his book *The Master Algorithm*, Pedro Domingos presents some more technical, but yet compelling, techniques for addressing algorithmic bias.

### *Further Reading*

<https://ideal.com/unconscious-bias/>

<https://www.nytimes.com/2019/11/15/technology/algorithmic-ai-bias.html><https://www.pressreader.com/nigeria/daily-trust/20130705/282252368125380>  
<https://www.propublica.org/series/machine-bias/p3>

<https://www.newscientist.com/article/2166207-discriminating-algorithms-5-times-ai-showed-prejudice/#ixzz66Guztaak>

---

<sup>14</sup> <https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing>

<sup>15</sup> <https://www.irishtimes.com/business/technology/inside-twitter-how-the-company-deals-with-difficult-conversations-1.4069288>

Examples of algorithmic bias.

We will return to the topic of algorithms later – right now we turn our attention to our personal and professional relationship with unconscious bias.

What does unconscious bias have to do with YOU?

*What does Unconscious Bias have to do with YOU?*

In our everyday work as teachers it is good to be aware of our own unconscious bias. This awareness can help us to explain our actions and, if necessary, even tweak our own behaviours so that our work as professional practitioners may be enhanced.

Unconscious bias can arise as a result of deeply held assumptions which we hold about the world around us. These assumptions have been shaped over years of personal and professional experiences, and sculpted by culture and society, they define our very identity. Our values, attitudes and dispositions are inextricably bound with our unconscious biases and assumptions.

**What does  
Unconscious Bias  
have to do with  
YOU?**



Barbara Larieve points out that when we experience something new, the experience is not pure. Everything is contextually bound and every new experience is filtered through this context as well as our past experiences. (This can explain why two people might walk away from the same exact same event such as a concert with entirely different opinions.)

It is reasonable to ask: How many of us assume, when we go into a class, that our students will take the same meaning and significance from our own actions (e.g. what we say and do) as we place on them ourselves?

Brookfield<sup>16</sup> in his work on critical reflection talks about the importance of being aware of our assumptions. In particular, he points out that *sincerity of intention does not guarantee purity*

---

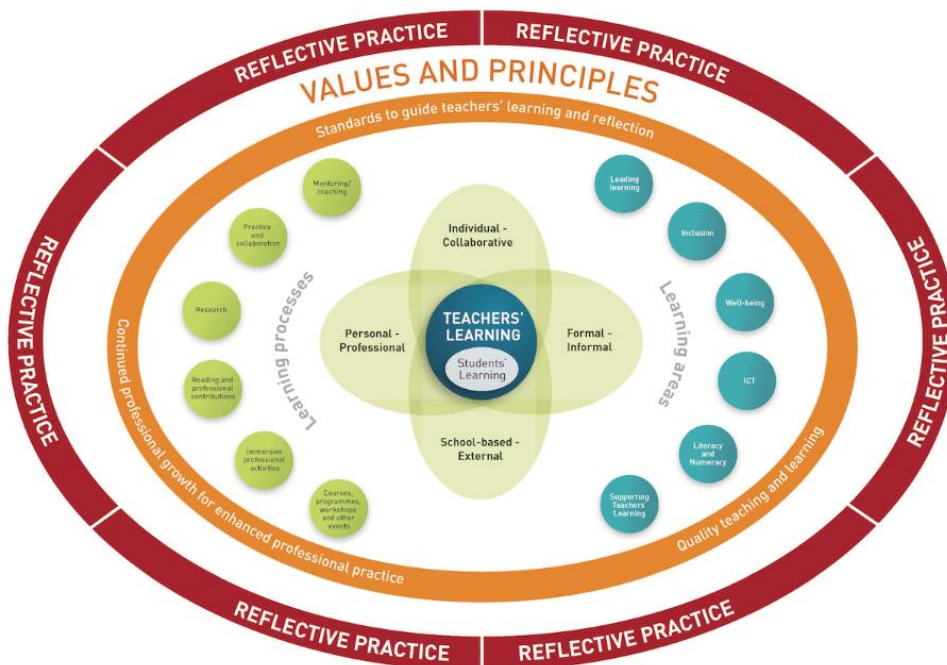
<sup>16</sup> Source: Becoming a Critically Reflective Teacher

of practice. John Dewey says 'We do not learn from experience... we learn from reflecting on experience'<sup>17</sup>.

It should make sense therefore that efforts to deepen our understanding of own biases and assumptions are worth pursuing, and one way to do this is through critical reflection on a habitual basis.

Critical reflection enables us to 'check in' on our own unexamined assumptions and the impact they can have on our own work as professional practitioners.

Reflective practice is considered so important that it envelopes the entire Cosán Framework as illustrated below.



But critical reflection is not easy. It is about challenging our biases, assumptions and questioning our own existing practices. This can be uncomfortable for many as it involves some or all of the following:

- exposing unexamined beliefs
- making visible our reflective loops
- facing deeply rooted personal attitudes concerning human nature, human potential and human learning (human capital)

<sup>17</sup> Source: Dewey, 1933, p.78

Efforts to become critically reflective involve negotiations of feelings of frustration, insecurity and rejection. These feelings are all very common and occur naturally as part of any change process. Dissonance is a natural part of professional learning and growth (Timperley). It is not pleasant to question beliefs and discover that (at least some of) our long standing, 'tried and trusted' approaches to teaching may have been misguided.

There is a nice quotation from Confucius shown below which captures the above sentiment.



By three methods we may learn wisdom: First, by reflection, which is noblest; Second, by imitation, which is easiest; and third by experience, which is the bitterest.

Dewey identifies the need for the reflective practitioner to be open-minded, wholehearted and responsible. Why do you think these three characteristics might be necessary?



**Reflection Activity Worksheet**

In his book, *The Courage To Teach*, Parker J. Palmer asks the question: *Who am I to teach?*

In order to answer this, it might be helpful to critically reflect on your own values as a teacher. A good starting point might be to question/identify your own unconscious biases and assumptions.

What are my values as a teacher?

## References/Further Reading

Brookfield, S. (2017). *Becoming a Critically Reflective Teacher*. 2nd Ed. San-Francisco: Jossey-Bass

Dewey, J. (1933). *How we think: A restatement of the relation of reflective thinking to the educative process*. New York: D.C. Heath and Company.

Hargreaves, A. & O'Connor, M.T. (2018). *Solidarity with solidity: The case for collaborative professionalism*. Phi Delta Kappan

Larrivee, B., (2000) *Transforming Teaching Practice: becoming the critically reflective teacher*, Reflective Practice, Vol. 1, No. 3

Palmer, Parker J. (1998). *The courage to teach: exploring the inner landscape of a teacher's life*. San Francisco: Jossey-Bass

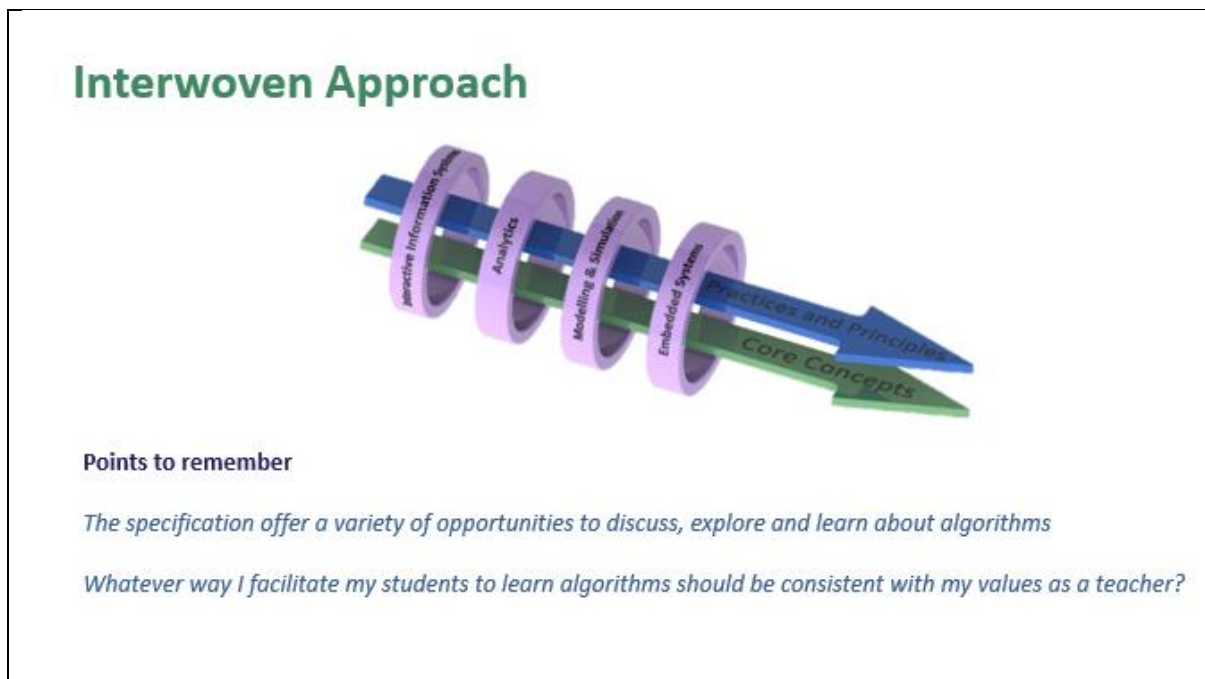
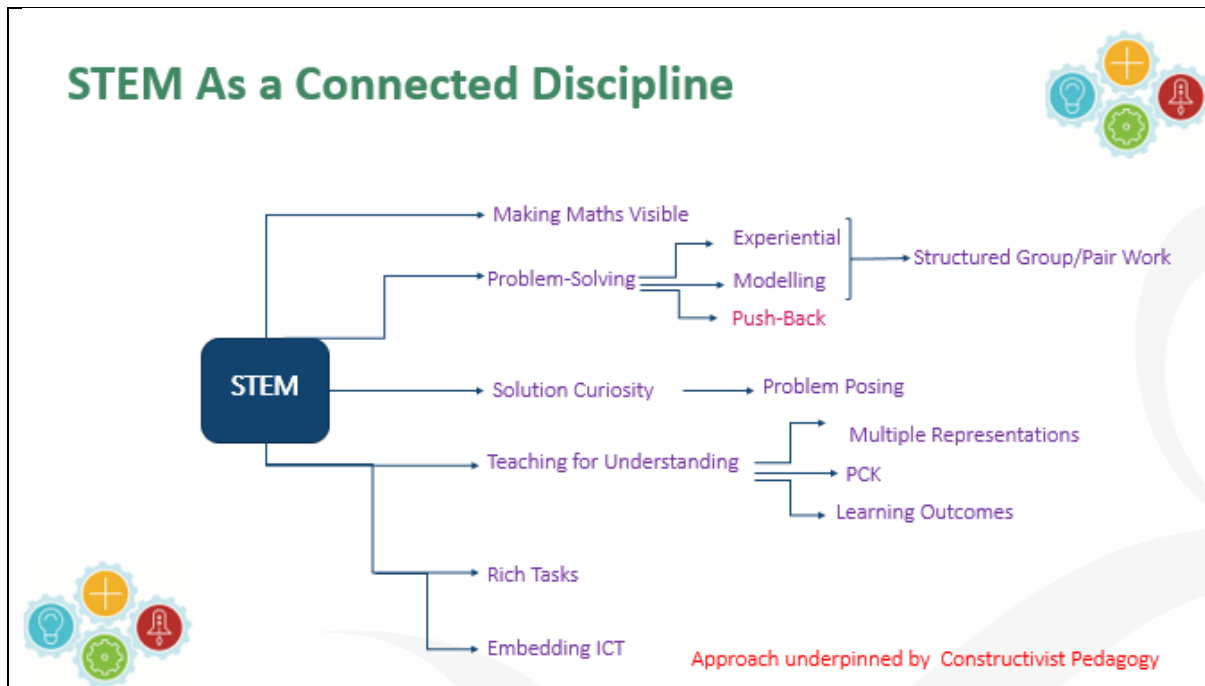
Schön, D. (1983). *The Reflective Practitioner: How professionals think in action*. London: Temple Smith

Spalding, E. & Wilson, A. (2002). *Demystifying reflection: A study of pedagogical strategies that encourage reflective journal writing*. Teachers College Record, 104, 1393-1421

Further information on models of reflection such as Brookfield's four lenses (self, peers, students and research), Gibbs' Reflective Cycle and Rolfe et al's Framework can be found on the Teaching Council's website at the following link:

<https://www.teachingcouncil.ie/en/Teacher-Education/Teachers-learning-CPD-/Cosan-Support-Materials/Reflecting-on-Professional-Learning/>

## Section 5 – Final Reflection



### Two question to consider

1. How can I connect Computer Science with other subjects in my school?
2. What other areas of the LCCS course does algorithms have the potential to link in with?  
(How can the LOs be interwoven?)

How will I provide students with opportunities to learn more about algorithms (in a manner that is consistent with my own values)?

**BLANK PAGE**

**BLANK PAGE**