



**TASK**

# Sorting and Searching

Visit our website

# Introduction

## WELCOME TO THE SORTING AND SEARCHING TASK!

In 2007 the former CEO of Google, Eric Schmidt, asked the then presidential candidate Barack Obama what the most efficient way to sort a million 32-bit integers was (watch it [here!](#)). Obama answered that the bubble sort would not be the best option. Was he correct? By the end of this task you might be able to answer that question! In this lesson you will learn about data sources and types, data structures, and how to use these, including ordering and finding data in different types of data structures.

## ABSTRACT DATA TYPES (ADTs)

Abstract data types are different from “just plain” data types, in that the term *abstract data type* refers to a way of understanding a conceptual abstraction, a semantic model for storing, accessing, and manipulating data, including the values, operations, and behaviours defined as being allowed for users to perform for each specific abstract data type. Some examples are given in the table below.

Abstract Data Type	Explanation	Example
Queues	<p>Queues are part of the Python queue package (i.e. to use queues you will need to import an external library).</p> <p>Queues follow the FIFO principle which stands for “first in first out” - i.e. the first object added to the queue becomes the first extracted from the queue.</p> <p>This is similar to how an ATM queue works. Imagine there is one ATM and a row of people. The first person to arrive will be the first one to leave as well.</p>	<pre>from queue import Queue  # Creating new Queue students = Queue()  # Adding students to queue students.put("Kylie") students.put("Julia") students.put("Glitch")  # Iterate over all students while not students.empty():     # Get current student     # This removes the student     from the queue     current_student =</pre>

	<p>The same principle can be applied to a Python queue- whichever data element you enter first will be the first element to leave the queue.</p> <p>It's also important to know that the queue is a type of linked list so you will also need to import the linked list class.</p>	<pre>students.get()     print(current_student)  # Output: # Kylie # Julia # Glitch</pre>
Stacks	<p>All lists in Python can implicitly be used as stacks.</p> <p>Stacks follow the rule of LIFO which stands for last in first out - i.e. the last object added to the queue becomes the first extracted from the queue.</p> <p>As an example of this, think of a stack of paperwork on a desk. The sheet of paper at the bottom was the first thing added to that stack; however, this will be the last thing we see as we work through the stack.</p> <p>The same principle applies to a stack in Python, when adding (we use the <i>.push()</i> method to add to a stack) anything after that value is entered will be removed first (we can use the <i>.pop</i> method to remove data from a stack)</p>	<pre># A list can also be used as a stack my_pets = []  # Add 3 entries to the stack my_pets.append("Dog") my_pets.append("Cat") my_pets.append("Ferret") print(my_pets) # Output =&gt; ['Dog', 'Cat', 'Ferret']  # Stacks use pop() method my_pets.pop() print(my_pets) # Output =&gt; ['Dog', 'Cat']</pre>

## Graphs

The graph data structure shows the relationship between multiple elements.

For example, on most social media platforms, whenever you follow someone new, you may get new suggestions on who to follow based on the person you recently followed.

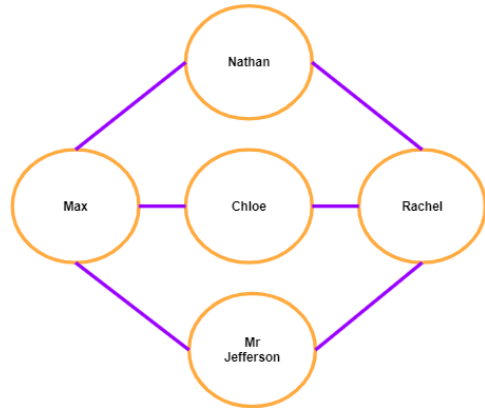
This is because the person you follow has a relationship with other people outside your social circle.

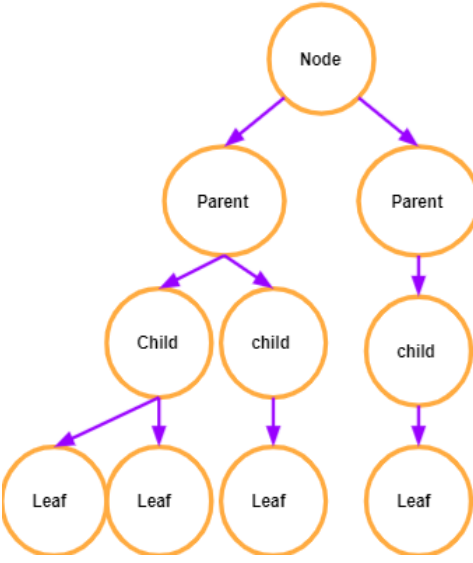
In the example on the side, we have a couple of people's names and their connections with one another.

When using graphs, we have the following definitions.

**Vertex** - A vertex in this example is the different people we have listed.

**Edge** - An edge is a line that shows the relationship between each person.



<p>Trees</p>	<p>Trees are very similar to a family tree or a business structure.</p> <p>Trees start off with a 'node'; a node is the starting point of the tree and does not have a previous generation.</p> <p>Trees then have children. Children are the portions of data that grow off from the node. Children have a parent node (in this case, the original node would be the parent); if two children stem from the same node, they are then declared siblings.</p> <p>Once we have children, we can then have parents. Parents are the original source of the data that extends from them.</p> <p>And finally, we have a leaf. A leaf is the last set of data to appear in the tree. It is the child of a parent node but does not have any children of its own.</p>	
--------------	--	--

## DATA STRUCTURES TO STORE ABSTRACT DATA TYPES

Let's recap what we know about data structures

### ARRAYS

The array is a data structure, provided by Python, which stores a variable number of elements of any type. An array is used to store a collection of data. However, it is more useful to think of an array as a collection of variables.

## Declaring Lists

Declaring lists in Python is a very simple step. To create an empty list, you simply need to specify the following syntax:

```
my_arr = []
```

Python is a weakly-typed language, which means that all data is treated equally (for the most part). Lists don't need to be restrained by a single data type, and so a list that looks like

```
my_arr = [1,2,3,"4"]
```

Is completely valid in Python.

## Finding the Length of a List

In Python, it is possible to view the associated methods and attributes of an object using the `dir()` method. If you use this to inspect lists using

```
print(dir(my_arr))
```

You will get a whole lot of gobbledygook. Among this gobbledygook is an attribute called `'__len__'`. What this means is that you can use the in-built Python method `len()` on it, which will return you the length of the array.

```
my_arr = [1,2,3,"4"]  
print(len(my_arr))  
# Returns 4
```

## Array Indices

To access array elements you use an index. Array indices range from 0 to `len(my_arr) - 1`. For example, the array created above (`my_arr`), holds four elements with indices from 0 to 3. You can represent each element in the array using the following syntax:

```
arrayVariable[index]
```

For example, `my_arr[1]` represents the second element in the array `my_arr`.

## Assigning Values within a List

To assign values to the elements we will use the syntax:

```
arrayVariable[index] = value
```

The example below initialises the array referenced by the variable `numArray`:

```
my_arr[0] = 23.6  
my_arr[1] = 45.12  
my_arr[2] = 8.4  
my_arr[3] = 77.7
```

## Slicing in Lists

Python has a powerful syntax with their lists, which is termed slicing. This means that there are certain nuances to accessing lists that aid in making your code easier and quicker.

For every list, the index `-1` gets you the last element, and the index `-2` gets you the second last element, etc.

```
print(my_arr[-1])  
# Returns "4"
```

You can also access a subset of the list using slicing. The general syntax is:

```
my_arr[first_element:last_element]
```

It should be noted that the `first_element` is inclusive, and the `last_element` is exclusive. Continuing with our previous example:

```
print(my_arr[1:3])  
# Returns [2, 3]
```

And, by not specifying a `first_element`, you simply start at the beginning of the list

```
print(my_arr[:3])  
# Returns [1, 2, 3]
```

There is one more trick to array slicing:

```
my_arr[first_element:last_element:step]
```

This last element, step, simply tells you how many steps to jump forward at a time. For example:

```
print(my_arr[::-2])  
# Returns [1, 3]
```

This starts at position 0, proceeds to the end of the list, and jumps forward 2 steps each time. This means that there is a quick and easy way to reverse any list:

```
print(my_arr[::-1])  
# Returns ['4', 3, 2, 1]
```

## SORTING AND SEARCHING DATA

Disorganised data does not provide us with information, as discussed in the previous lesson. We have to have methods for arranging data logically - sorting it - and for looking through it to find things (whether we want to just read an element, or insert/delete data at a specific point, for example inserting a Surname in the correct place in a set of data alphabetically sorted data about Employees). In this lesson, we're going to take a close look at the ways we can go about sorting and searching data.

### Algorithms

As you know, an algorithm is a clear, defined way of solving a problem - a set of instructions that can be followed to solve a problem, and that can be repeated to solve similar problems. Every time you write code, you are creating an algorithm — a set of instructions that solves a problem.

There are a number of well-known algorithms that have been developed and extensively tested to solve common problems. In this lesson, you will be learning about algorithms used to sort and to search through a data structure that contains a collection of data (e.g. a List). In other words, in this lesson, you are going to learn about **algorithms** that manipulate the data in those data structures.

There are two main benefits that you should derive from learning about the sorting and searching algorithms in this lesson:

1. You will get to analyse and learn about algorithms that have been developed and tested by experienced software engineers, providing a model for writing good algorithms.




2. As these algorithms have already been implemented by others, you will see how to use them without writing them from scratch, i.e. you don't have to reinvent the wheel. Once you understand how they work, you can easily implement them in any coding language in which you gain proficiency.

Let's consider algorithms for searching and sorting data.

## SORTING ALGORITHMS

If you want to sort a set of numbers into ascending order, how do you normally go about it? The most common way people sort things is to scan through the set, find the smallest number, and put it first; then find the next smallest number and put it second; and so on, continuing until they have worked through all the numbers. This is an effective way of sorting, but for a program, it can be time-consuming and memory-intensive.

There are a number of algorithms for sorting data, such as the Selection Sort, Insertion Sort, Shell Sort, Bubble Sort, Quick Sort, and Merge Sort. These all have different levels of operational complexity, and complexity to code. An important feature to consider when analysing algorithms and their performance is how efficient they are - how much they can effectively do in how long. With sorting algorithms, we're interested in knowing how quickly a particular algorithm can sort a list of items. We use something called **Big O notation** to evaluate and describe the efficiency of code. Big O notation, also called Landau's Symbol, is used in mathematics, complexity theory, and computer science. The O refers to *order of magnitude* - the rate of growth of a function dependent on its input. An example of Big O notation in use is shown below:

Big - O  
  
$$f(x) = O(g(x))$$

Don't be intimidated by the maths - you don't have to understand the intricacies of it to be able to understand algorithm efficiency using Big O Notation. We'll take a very brief, high-level look at it below, which should be enough to assist you in understanding the Big O complexities of the algorithms we look at in this lesson.

There are seven main mathematical functions that are used to represent the complexity and performance of an algorithm. Each of these mathematical

functions is shown in the image below. Each function describes how an algorithm's complexity changes as the size of the input to the algorithm changes.

As you can see from the image, some functions (e.g.  $O(\log n)$  or  $O(1)$ ) describe algorithms where the efficiency of the algorithm *stays the same regardless of how big the input to the algorithm becomes* (shaded green to indicate excellent performance in the image). A sorting function with this level of Big O performance would continue to sort things really well and really quickly even when sorting very large datasets. Conversely, other functions (e.g.  $O(n!)$  and  $O(2^n)$ ), describe a situation where the bigger the input gets, the worse the complexity and efficiency get. Sorting functions with this level of efficiency might work okay for small datasets, or datasets that are already largely ordered, but become very inefficient and slow as the size and/or degree of disorder in the input data they're handling increases.

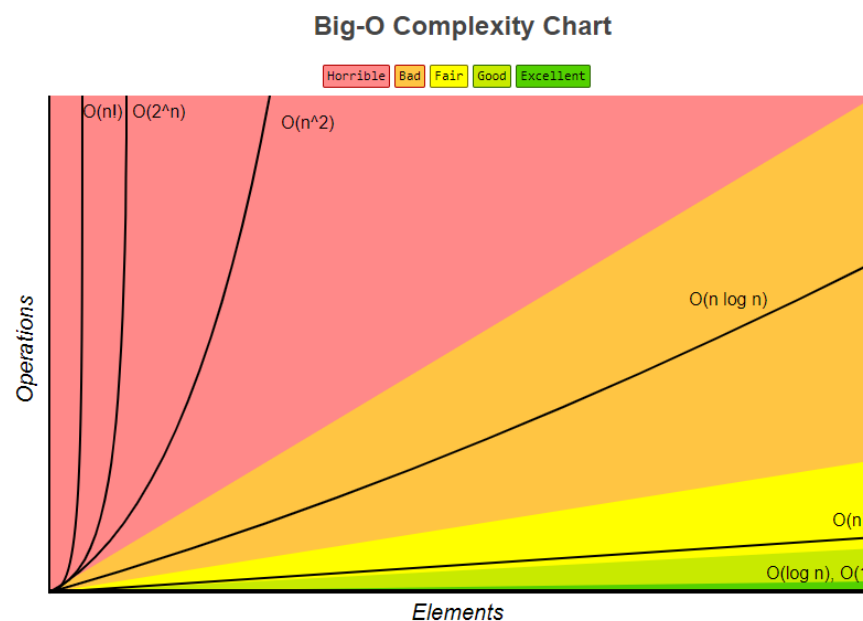


Image source: <http://bigocheatsheet.com/>

Keep the above image in mind and refer back to it as we discuss different sorting and searching algorithms and refer to their complexity and performance using Big O notation.

As previously stated, there are a number of algorithms for sorting data. The table below provides a comparison of the time complexity of some of these.

Algorithm	Time Complexity		
	Best	Average	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$

Image source: <http://bigocheatsheet.com/>

If you want to have a look at animated visual representations of sorting methods in action, [this is a great resource](#) that lets you run various sorts at your chosen speed, or step through them one operation at time so that you can trace exactly what is happening. Sitting down with the algorithm for a particular sorting method and stepping through an animated sort is a great way to understand it. You can further deepen your understanding by creating a [trace table](#) of the examples you're working with. This topic is a common favourite of interviewers in tech interviews, and so it is worth ensuring you understand it well. If you would like more information on Big O notation, [Wikipedia has quite a comprehensive entry on the topic](#); if you prefer to watch than read, this [2 hour video course](#) will teach you a lot.

In the next section we will take a detailed look at three of the most popular sorting methods (and equally popular topics for interview questions!): **Bubble Sort, Quick Sort, and Merge Sort.**

## BUBBLE SORT

Bubble Sort got its name because the sorting algorithm causes the larger values to 'bubble up' to the top of a list. Performing a Bubble Sort involves comparing an item ( $a$ ) with the item next to it ( $b$ ). If  $a$  is bigger than  $b$ , they swap places. This process continues until all the items in the list have been compared, and then the process starts again; this happens as many times as one less than the length of the list to ensure that enough passes through the list are made. Let's look at the example below:

8	3	1	4	7	<b>First iteration:</b> Five numbers in random order.
---	---	---	---	---	---

3	<b>8</b>	1	4	7	3 < 8 so 3 and 8 swap
3	1	<b>8</b>	4	7	1 < 8, so 1 and 8 swap
3	1	4	<b>8</b>	7	4 < 8, so 4 and 8 swap
3	1	4	7	<b>8</b>	7 < 8, so 7 and 8 swap (do you see how 8 has bubbled to the top?)
<b>3</b>	1	4	7	8	<b>Next iteration:</b>
1	<b>3</b>	4	7	8	1 < 3, so 1 and 3 swap. 4 > 3 so they stay in place and the iteration ends

Bubble Sort is  $O(n^2)$  and is written below (Adamchick, 2009):

```
def bubble_sort(arr):
    for i in range(len(arr) - 1, -1, -1):
        for j in range(1, i + 1):
            if arr[j - 1] > arr[j]:
                # Swap the values
                arr[j - 1], arr[j] = arr[j], arr[j - 1]
    return arr
```

Practise this concept with some number sequences of your own. It can be a little tricky to comprehend initially, but becomes quite simple as you become more familiar with it.

## QUICK SORT

Quick Sort, as the name suggests, is a fast sorting algorithm. We start with a *pivot* point (usually the first element, but can be any element). We then look at the next element. If it is bigger than the pivot point, it stays on the right, and if it is smaller than the pivot point, it moves to the left of it. We do this for every number in the list going through the entire list once. At this point we have divided the array into values less-than-pivot, pivot, and greater-than-pivot. These sublists each get a new pivot point and the same iteration applies. This 'divide and conquer' process continues until each element has become a pivot point. See the example below:

<b>4</b>	3	2	8	7	5	1	<b>Pivot point: 4</b>
----------	---	---	---	---	---	---	-----------------------

(3)	2	1)	4	(8	7	5)	Divided list with new pivot points for each (3 and 8)
(2	1)	3	4	(7	5)	8	Divided list with new pivot points each (2 and 7)
(1)	2	3	4	(5	7	8	Divided list with new pivot points each (1 and 5)
1	2	3	4	5	7	8	Nothing changes with above pivot points so the list is sorted

Quick Sort is  $O(\log n)$  and is written below (Dalal, 2004):

```
def quick_sort(arr, low, high):
    if low < high:
        mid = partition(arr, low, high)
        arr = quick_sort(arr, low, mid - 1)
        arr = quick_sort(arr, mid + 1, high)
    return arr
```

This is the partition method that is used to select the pivot point and move the elements around:

```
def partition(arr, low, high):
    # The pivot point is the first item in the subarray
    pivot = arr[low]

    # Loop through the array. Move items up or down the array so that they
    # are in the proper spot with regards to the pivot point
    while low < high:
        # Can we find a number smaller than the pivot point:
        # Keep moving the high marker down the array until we find this
        # or until high==low
        while low < high and arr[high] >= pivot:
            high -= 1

        if low < high:
            # Found a smaller number, swap it into position
            arr[low] = arr[high]
            # Now look for a number larger than the pivot point
            while low < high and arr[low] <= pivot:
                low += 1
```

```

    if low < high:
        # Found one! Move it into position
        arr[high] = arr[low]
    # Move the pivot back into the array and return its index
    arr[low] = pivot
    return low

```

Note how the partition method is actually doing most of the heavy lifting, while the quicksort method just calls the partition method. This is a bit confusing at first, but keep practising to help you get the hang of the concept.

## MERGE SORT

Like Quick Sort, Merge Sort is also a 'divide and conquer' strategy. In this strategy, we break apart the list to then put it back together in order. To start, we break the list into two and keep dividing until each element is on its own. Next, we start combining them two at a time and sorting as we go. Have a look at the example below. We start by dividing up the elements:

4	3	2	8	7	5	1	<b>List of numbers</b>
(4	3	2	8)	(7	5	1)	The list is divided into 2
(4	3)	(2	8)	(7)	(5	1)	Divided lists are divided again
(4)	(3)	(2)	(8)	(7)	(5)	(1)	Divided until each element is on its own

Next, we start to pair the elements back together, sorting as we go.

### First merge

(4)	(3)	(2)	(8)	(7)	(5)	(1)	<b>Individual elements</b>
(4	3)	(2	8)	(7	5)	(1)	Elements are paired and compared
(3	4)	(2	8)	(5	7)	(1)	Sorted paired lists

Now we combine again. We sort as we combine by comparing the items:

### Second merge part 1

(3	4)	(2	8)	
<b>2</b>				1. 3 vs. 2 — $3 > 2$ so 2 becomes index 0
2	<b>3</b>			2. 3 vs 8 — $3 < 8$ so 3 becomes index 1
2	3	<b>4</b>		3. 4 vs. 8 — $4 < 8$ so 4 becomes index 2
2	3	4	<b>8</b>	4. 8 is leftover so becomes index 3

### Second merge part 2

(5	7)	(1)	
<b>1</b>			5. 5 vs. 1 — $5 > 1$ so 1 becomes index 0
1	<b>5</b>		6. 5 vs. 7 — $5 < 7$ so 5 becomes index 1
1	5	<b>7</b>	7. 7 is leftover so becomes index 2

Finally, we sort and merge the last two lists:

### Third merge

(2	3	4	8)	(1	5	7)	
<b>1</b>							1. 2 vs. 1 — $1 < 2$ so 1 becomes index 0
1	<b>2</b>						2. 2 vs. 5 — $2 < 5$ so 2 becomes index 1
1	2	<b>3</b>					3. 3 vs. 5 — $3 < 5$ so 3 becomes index 2
1	2	3	<b>4</b>				4. 4 vs. 5 — $4 < 5$ so 4 becomes index 3
1	2	3	4	<b>5</b>			5. 8 vs. 5 — $8 > 5$ so 5 becomes index 4
1	2	3	4	5	<b>7</b>		6. 8 vs. 7 — $8 > 7$ so 7 becomes index 5
1	2	3	4	5	7	<b>8</b>	7. 8 is leftover so becomes index 6

Merge Sort is  $O(n \log(n))$  and is written below (University of Cape Town, 2014):

```
def merge_sort(items):
    n = len(items)
    temporary_storage = [None] * n
    size_of_subsections = 1

    while size_of_subsections < n:
        for i in range(0, n, size_of_subsections * 2):
            i1_start, i1_end = i, min(i + size_of_subsections, n)
            i2_start, i2_end = i1_end, min(i1_end + size_of_subsections, n)
            sections = (i1_start, i1_end), (i2_start, i2_end)
            merge(items, sections, temporary_storage)
            size_of_subsections *= 2

    return items
```

The above method finds the midpoint of the array and recursively divides it until all the elements are the only ones in their own arrays. Once this is done the following method is called recursively to put them back together in order (University of Cape Town, 2014):

```
def merge(items, sections, temporary_storage):
    (start_1, end_1), (start_2, end_2) = sections
    i_1 = start_1
    i_2 = start_2
    i_t = 0

    while i_1 < end_1 or i_2 < end_2:
        if i_1 < end_1 and i_2 < end_2:
            if items[i_1] < items[i_2]:
                temporary_storage[i_t] = items[i_1]
                i_1 += 1
            else: # the_list[i_2] >= items[i_1]
                temporary_storage[i_t] = items[i_2]
                i_2 += 1
            i_t += 1
```



```

elif i_1 < end_1:
    for i in range(i_1, end_1):
        temporary_storage[i_t] = items[i_1]
        i_1 += 1
        i_t += 1

else: # i_2 < end_2
    for i in range(i_2, end_2):
        temporary_storage[i_t] = items[i_2]
        i_2 += 1
        i_t += 1

for i in range(i_t):
    items[start_1 + i] = temporary_storage[i]

```

Python includes its own **sort()** method, which uses the [Timsort algorithm](#).

```

my_list = [54,26,93,17,77,31,44,55,20]
my_list.sort()
print(my_list)
# Returns [17, 20, 26, 31, 44, 54, 55, 77, 93]

```

## SEARCHING ALGORITHMS

There are two main algorithms for searching through elements in code, namely **linear search** and **binary search**. Linear search is closest to how we, as humans, search for something. If we are looking for a particular book on a messy bookshelf, we will look through all of them until we find the one we're looking for, right? This works very well if the group of items are not in order, but it can be quite time-consuming. Binary search, on the other hand, is much quicker but it only works with an ordered collection.

In the next section we will take a detailed look at these two searching methods. If you want to have a look at visual representations of any of the searching methods above, have a look [here](#).

## LINEAR SEARCH

As mentioned, linear search is used when we know that the elements are not in order. We start by knowing what element we want, and we go through the list comparing each element to the known element. The process stops when we either find an element that matches the known element, or we get to the end of the list. Linear search is  $O(n)$  and is executed below (Dalal, 2004):

```
def sequential_search(item, arr):  
    # If item is not found, return None  
    # Iterate over list. If we find our item, break loop  
    # and return index  
    for i in range(len(arr)):  
        if arr[i] == item:  
            return i
```

## BINARY SEARCH

Binary search works if the list is in order. If we go back to our book example, if we were looking for *To Kill a Mockingbird* and the books were organised in alphabetical order, we could simply go straight to 'T', refine to 'To' and so on until we found the book. This is how binary search works. We start in the middle of the list and determine if our sought-for element is smaller than (on the left of) or bigger than (on the right of) that element. By doing this we instantly eliminate half of the elements in the list to search through! We continue to halve the list until either we find our sought-after element, or until all possibilities have been eliminated, i.e. there are no elements found to match our sought-after element. Let's look at an example:

We are looking for the number 63 in the following list:

3	10	63	80	120	6000	7400	8000
---	----	----	----	-----	------	------	------

In the list above, the midpoint is between 80 and 120, so the value of the midpoint will be 100 ( $(120+80) \div 2 = 100$ ). 63 is less than 100, so we know 63 must be in the left half of the list:

3	10	63	80	<del>120</del>	<del>6000</del>	<del>7400</del>	<del>8000</del>
---	----	----	----	----------------	-----------------	-----------------	-----------------

Let's half what's left. The midpoint is now 37.5, which is smaller than 63, so our element must be on the right side.

3	<del>10</del>	63	80	<del>120</del>	<del>6000</del>	<del>7400</del>	<del>8000</del>
---	---------------	----	----	----------------	-----------------	-----------------	-----------------

We're left with our last two elements. The midpoint of 63 and 80 is 71.5, which is greater than 63, so our element must be on the left side.

3	<del>10</del>	63	<del>80</del>	<del>120</del>	<del>6000</del>	<del>7400</del>	<del>8000</del>
---	---------------	----	---------------	----------------	-----------------	-----------------	-----------------

And it is! We've found our element.

Binary search is  $O(\log n)$  (UCT, 2014) and is written below (Dalal, 2004):

```
def binary_search(item, arr):
    low, high = 0, len(arr) - 1

    # Keep iterating until low and high cross
    # Returns None if item not found
    while high >= low:
        # Find midpoint
        mid = (low + high) // 2

        # If item is found at midpoint, return
        if arr[mid] == item:
            print(arr[mid])
            return mid

        # Else, if item at midpoint is less than item,
        # search the second half of the list
        elif arr[mid] < item:
            low = mid + 1

        # Else, search first half
        else:
            high = mid - 1
```

# Compulsory Task 1

- Create a Python script called **array\_lists.py**
- Design a class called Album. The class should contain:
  - The data fields *albumName* (String), *numberOfSongs* (int) and *albumArtist* (String).
  - A `__str__` method that returns a string that represents an Album object in the following format:  
(*albumName*, *albumArtist*, *numberOfSongs*)
- Create a new list called *albums1*, add 5 albums to it and print it out.
- Sort the list according to the *numberOfSongs* and print it out. (You may want to [understand the key parameter in the sort method](#)).
- Swap the element at position 1 of *albums1* with the element at position 2 and print it out.
- Create a new list called *albums2*.
- Add 5 albums to the *albums2* List and print it out.
- Copy all of the albums from *albums1* into *albums2*.
- Add the following two elements to *albums2*:
  - (Dark Side of the Moon, Pink Floyd, 9)
  - (Oops!... I Did It Again, Britney Spears, 16)
- Sort the courses in *albums2* alphabetically according to the album name and print it out.
- Search for the album “Dark Side of the Moon” in *albums2* and print out the index of the album in the List.

## Compulsory Task 2

In a newly created Python script called **merge\_sort.py**:

- Modify the Merge sort algorithm to order a list of strings by string length from the longest to the shortest string.
- Run the modified Merge sort algorithm against 3 string lists of your choice. Please ensure that each of your chosen lists is not sorted and has a length of at least 10 string elements.

## Compulsory Task 3

Using the following array: [27, -3, 4, 5, 35, 2, 1, -40, 7, 18, 9, -1, 16, 100]

- Create a Python script called **sort\_and\_search.py**  
Which searching algorithm would be appropriate to use on the given list?
- Implement this searching algorithm to search for the number 9. Add a comment to explain why you think this algorithm was a good choice.
- Research and implement the Insertion sort on this array.
- Implement a searching algorithm you haven't tried yet in this Task on the sorted array to find the number 9. Add a comment to explain where you would use this algorithm in the real world.



Rate us

## Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this lesson, task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.



## REFERENCES

Adamchik, V. (2009). Sorting. Retrieved 25 February 2020, from <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html>

Dalal, A. (2004). Searching and Sorting Algorithms. Retrieved 25 February 2020, from <http://www.cs.carleton.edu/faculty/adalal/teaching/f04/117/notes/searchSort.pdf>

University of Cape Town. (2014). Sorting, searching and algorithm analysis — Object-Oriented Programming in Python 1 documentation. Retrieved 25 February 2020, from [https://python-textbok.readthedocs.io/en/1.0/Sorting\\_and\\_Searching\\_Algorithms.html](https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html)