Hyperiondev

TASK

Sorting and Searching

Visit our website

Introduction

In today's data-driven world, the ability to efficiently search and sort through large volumes of information is crucial. Whether you're organising your email, searching through databases, or processing big data, mastering these fundamental skills can dramatically improve the performance of your applications. In this lesson, you will learn about abstract data types, explore various data structures, and understand how to effectively use them for sorting and searching. By the end of this task, you will be well-equipped to handle data more efficiently and make informed decisions about the best techniques to use in different scenarios.

ABSTRACT DATA TYPES

Abstract data types are different from regular data types. Abstract data types provide a conceptual model for storing, accessing, and manipulating data, including the allowed values, operations, and behaviours. Examples are provided below.

Queues

Queues are part of the Python queue package, i.e. to use queues you will need to import an external library. 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.

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. 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. 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.

```
from queue import Queue

# Creating new Queue
students = Queue()

# Adding students to the 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 = students.get()
    print(current_student)

# Output:
# Kylie
# Julia
# Glitch
```

Stacks

All lists in Python can implicitly be used as stacks. Stacks follow the rule of LIFO which stands for '**last in first out**'. This means the last object added to the stack is the first one to be removed.

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.

The same principle applies to a stack in Python, when adding (we use the `.append()` method to add to a stack) anything after that value is entered will be removed first (we can use the `.pop` method to remove data from a stack).

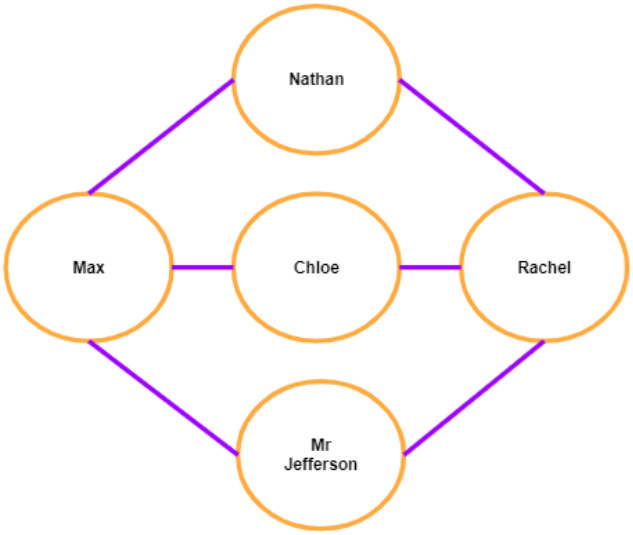
```
# A list can also be used as a stack
my_pets = []

# Add 3 entries to the stack
my_pets.append("Dog")
my_pets.append("Cat")
my_pets.append("Ferret")
print(my_pets)
# Output => ['Dog', 'Cat', 'Ferret']

# Stacks use pop() method
my_pets.pop()
print(my_pets)
# Output => ['Dog', 'Cat']
```

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 below, 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.

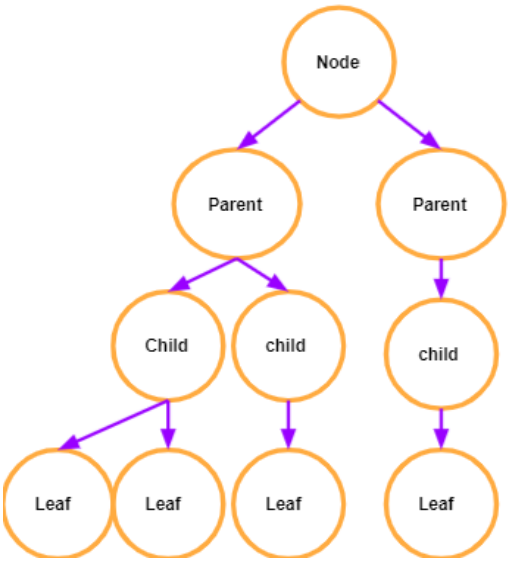


Trees

Trees are very similar to a family tree or a business structure. Trees start off with a 'node'; a node is the starting point of the tree and does not have a previous generation.

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.

Once we have children, we can then have parents. Parents are the original source of the data that extends from them. 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.



DATA STRUCTURES TO STORE ABSTRACT DATA TYPES

There are several data structures including lists, nested lists, tuples, sets, and dictionaries. We will be focussing on lists here but you can explore the other data structures in the **Python documentation**. Let's recap what we know about lists.

Lists

The list is a data structure provided by Python that stores a variable number of elements of any type. A list is used to store a collection of data. It is more useful to think of a list as a collection of elements.

Declaring lists

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

```
empty_list = []
```

Python is a dynamically typed language, meaning that variable types are interpreted at runtime and can change as the program executes. Additionally,

Python is strongly typed, which means that it enforces type constraints during operations to prevent unintended behaviour.

Lists in Python do not need to be restricted to a single data type, so a list containing elements of different types, like the one shown below, is completely valid.

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

Finding the length of a list

In Python, you can view the associated methods and attributes of an object using the `dir()` function. For example, if you inspect a list using `dir()`:

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

print(dir(example_list))
```

You will get a list of all the methods and attributes associated with the list object. Among these, you will find **special methods** (those with double underscores, also known as dunder methods), such as `__len__`. These special methods allow for more advanced functionality. The presence of `__len__` indicates that you can use the built-in Python function `len()` to get the length of the list. For example:

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

print(len(example_list))  # Prints 4.
```

List indices

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

```
example_list[index]
```

For example, **example_list[1]** represents the second element in the list **example_list**. The example below makes use of the **example_list** to retrieve each element within the list.

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

# Accessing elements by index
element_0 = example_list[0]  # Accesses the first element, 1
element_1 = example_list[1]  # Accesses the second element, 2
element_2 = example_list[2]  # Accesses the third element, 3
element_3 = example_list[3]  # Accesses the fourth element, "4"
```

Assigning values within a list

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

```
example_list[index] = value
```

The example below initialises the list referenced by the variable **example_list**:

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

# Modifying elements by index
example_list[0] = 23.6
example_list[1] = 45.12
example_list[2] = 8.4
example_list[3] = 77.7

print(example_list)  # Output: [23.6, 45.12, 8.4, 77.7]
```

Slicing in lists

Python has a powerful syntax for working with lists, including negative indexing and slicing. Slicing allows you to access specific portions of a list, making your code more efficient and readable.

In Python lists, negative indices can be used to access elements from the end of the list. For example, index **-1** retrieves the last element, **-2** retrieves the second-to-last element, and so on.

```
example_list = [1, 2, 3, "4"]
print(example_list[-1])  # Using negative indexing outputs "4"
```

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

```
example_list[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:

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

print(example_list[1:3])  # Outputs [2, 3]
```

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

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

print(example_list[:3])  # Outputs [1, 2, 3]
```

In Python lists, slicing can be further extended by adding a **`step`** parameter to the syntax:

```
example_list[first_element:last_element:step]
```

The step parameter in list slicing specifies how many elements to skip each time when creating a slice. For example:

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

print(example_list[::2])  # Outputs [1, 3]
```

This slicing operation starts iterating from the beginning of the list (position **`0`**), continues until the end of the list, and jumps forward by **`2`** steps each time. Using a negative step in slicing allows for an easy reversal of the list, as shown in the example below:

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

print(example_list[::-1])  # Outputs ['4', 3, 2, 1]
```

SORTING AND SEARCHING DATA

Disorganised data does not always provide us with enough information. We need methods for arranging data logically, like sorting it and finding things quickly. This includes tasks like retrieving specific elements or adding and removing data at specific points. For example, inserting a surname in alphabetical order into a list of employees will be easier after sorting the list of employees. In this lesson, we're going to take a look at the ways we can go about sorting and searching data.

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 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.

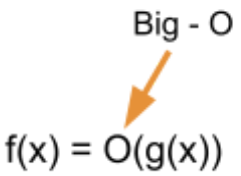
Let's consider algorithms for searching and sorting data.

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 Selection Sort, Insertion Sort, Shell Sort, Bubble Sort, QuickSort, and Merge Sort. These all have different levels of operational complexity and difficulty 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:



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.

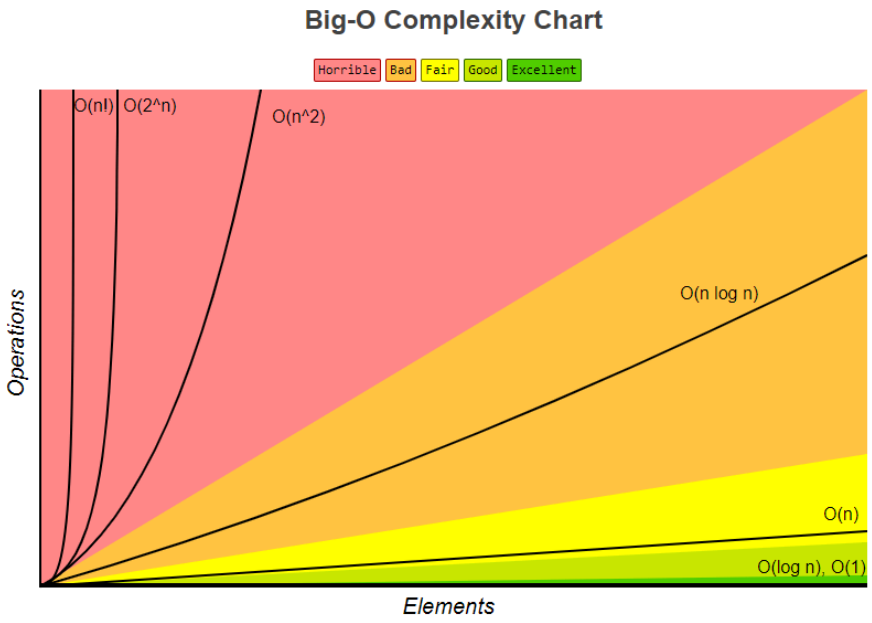


Image source (Big-O Cheat Sheet, 2013)

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.

Keep the Big O complexity chart 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
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)

Image source (Big-O Cheat Sheet, 2013)

If you want to have a look at animated visual representations of sorting methods in action, **this is a great resource**. It lets you run various sorts at your chosen speed, or step through them one operation at a 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 popular choice for tech interviews, so it is worth ensuring you understand it well. If you would like more information on Big O notation, **Wikipedia has a comprehensive entry** on the topic.

Let's look at popular sorting methods (and equally popular topics for interview questions): **Bubble Sort, QuickSort,** 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	**First iteration:** Five numbers in random order.
3	**8**	1	4	7	3 < 8 so 3 and 8 swap
3	1	**8**	4	7	1 < 8, so 1 and 8 swap
3	1	4	**8**	7	4 < 8, so 4 and 8 swap
3	1	4	7	**8**	7 < 8, so 7 and 8 swap (do you see how 8 has bubbled to the top?)
3	1	4	7	8	**Next iteration:**
1	**3**	4	7	8	1 < 3, so 1 and 3 swap. 4 > 3 so they stay in place and the iteration ends

Bubble Sort is $O(n^2)$:

```
def bubble_sort(items):
    # Traverse through all elements in the list
    for i in range(len(items) - 1, -1, -1):
        # Traverse the list from 1 to i
        for j in range(1, i + 1):
            # Swap if the element is greater than the next element
            if items[j - 1] > items[j]:
                items[j - 1], items[j] = items[j], items[j - 1]
    return items


# Example usage:
example_list = [1, 3, 2, 6, 7, 4, 11, 5]
sorted_list = bubble_sort(example_list)
print(sorted_list)
```

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

QuickSort

QuickSort, as the name suggests, is a fast sorting algorithm. We start with a *pivot* point (usually the first element, but it 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 list 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:

4	3	2	8	7	5	1	**Pivot point:** 4
(**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 the above pivot points so the list is sorted

QuickSort is O(log n) and is written below (Dalal, 2004):

```
def quick_sort(items, low, high):
    if low < high:
        # Partition the list and get the pivot index
        mid = partition(items, low, high)

        # Recursively sort the left partition
        items = quick_sort(items, low, mid - 1)

        # Recursively sort the right partition
        items = quick_sort(items, mid + 1, high)

    return items
```

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

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

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

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

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

Note how the **partition** function is doing most of the heavy lifting, while the **quick_sort** function just calls the partition method. This is a bit confusing at first, try experimenting with the **quick_sort** and **partition** functions to see how they handle lists of different data types. Keep in mind that you will need to ensure that both functions are present within the file when testing the **quick_sort** function, as it relies on the **partition** function.

Example usage combining both the **quick_sort** and **partition** functions:

```
def quick_sort(items, low, high):
    if low < high:
        # Partition the list and get the pivot index
        mid = partition(items, low, high)

        # Recursively sort the left partition
        items = quick_sort(items, low, mid - 1)
```

```
        # Recursively sort the right partition
        items = quick_sort(items, mid + 1, high)

    return items


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

    # Loop through the list. Move items up or down the list 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 list until we find this
        # or until high == low
        while low < high and items[high] >= pivot:
            high -= 1

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

            if low < high:
                # Found one! Move it into position
                items[high] = items[low]

    # Move the pivot back into the list and return its index
    items[low] = pivot
    return low



# Example usage:
example_list = [33, 10, 59, 26, 41, 58, 18]
sorted_list = quick_sort(example_list, 0, len(example_list) - 1)
print(sorted_list)
```

Merge Sort

Like QuickSort, Merge Sort is also a 'divide and conquer' strategy. In this strategy, we break apart the list and 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	**List of numbers**
(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)	**Individual elements**
(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)	**Pairs to compare**
2				1. 3 vs. 2 — 3 > 2 so 2 becomes index 0
2	**3**			2. 3 vs 8 — 3 < 8 so 3 becomes index 1
2	3	**4**		3. 4 vs. 8 — 4 < 8 so 4 becomes index 2
2	3	4	**8**	4. 8 is leftover so becomes index 3

Second merge part 2

(5	7)	(1)	**Pairs to compare**
1			5. 5 vs. 1 — 5 > 1 so 1 becomes index 0
1	**5**		6. 5 vs. 7 — 5 < 7 so 5 becomes index 1
1	5	**7**	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)	**Pairs to merge**
1							1. 2 vs. 1 — 1 < 2 so 1 becomes index 0
1	**2**						2. 2 vs. 5 — 2 < 5 so 2 becomes index 1
1	2	**3**					3. 3 vs. 5 — 3 < 5 so 3 becomes index 2
1	2	3	**4**				4. 4 vs. 5 — 4 < 5 so 4 becomes index 3
1	2	3	4	**5**			5. 8 vs. 5 — 8 > 5 so 5 becomes index 4
1	2	3	4	5	**7**		6. 8 vs. 7 — 8 > 7 so 7 becomes index 5
1	2	3	4	5	7	**8**	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):
    # Get the length of the input list
    items_length = len(items)

    # Create temporary storage for merging
    temporary_storage = [None] * items_length

    # Initialise the size of subsections to 1
    size_of_subsections = 1

    # Iterate until the size of subsections is less than the length of the
list
    while size_of_subsections < items_length:
        # Iterate over the list in steps of size_of_subsections * 2
        for i in range(0, items_length, size_of_subsections * 2):
            # Determine the start and end indices of the two subsections
            # to merge.
            first_section_start, first_section_end = i, min(
                i + size_of_subsections, items_length
            )
            second_section_start, second_section_end = first_section_end, min(
                first_section_end + size_of_subsections, items_length
            )

            # Define the sections to merge
            sections = (first_section_start, first_section_end), (
                second_section_start,
                second_section_end,
            )

            # Call the merge function to merge the subsections
            merge(items, sections, temporary_storage)

        # Double the size of subsections for the next iteration
        size_of_subsections *= 2

    # Return the sorted list
    return items
```

The above function finds the midpoint of the list and recursively divides it until all the elements are the only ones in their own lists.

Once this is done the following **merge** function is called recursively to put them back together in order (University of Cape Town, 2014):.

```
def merge(items, sections, temporary_storage):
    # Unpack the sections tuple to get the start and end indices
    # of each section.
    (first_section_start, first_section_end), (
        second_section_start,
        second_section_end,
    ) = sections

    # Initialise indices for the two sections and temporary storage
    left_index = first_section_start
    right_index = second_section_start
    temp_index = 0

    # Loop until both sections have been fully merged
    while left_index < first_section_end or right_index < second_section_end:
        # Check if both sections still have elements to compare
        if left_index < first_section_end and right_index < second_section_end:
            # Compare elements from both sections
            if items[left_index] < items[right_index]:
                # Place the smaller element into temporary storage
                temporary_storage[temp_index] = items[left_index]
                left_index += 1
            else:  # items[right_index] <= items[left_index]
                temporary_storage[temp_index] = items[right_index]
                right_index += 1
            temp_index += 1

        # If section 1 still has elements left to merge
        elif left_index < first_section_end:
            # Copy remaining elements from section 1 to temporary storage
            for i in range(left_index, first_section_end):
                temporary_storage[temp_index] = items[left_index]
                left_index += 1
                temp_index += 1

        # If section 2 still has elements left to merge
        else:  # right_index < second_section_end
            # Copy remaining elements from section 2 to temporary storage
            for i in range(right_index, second_section_end):
                temporary_storage[temp_index] = items[right_index]
                right_index += 1
                temp_index += 1
```

```
    # Copy sorted elements from temporary storage back to the original list
    for i in range(temp_index):
        items[first_section_start + i] = temporary_storage[i]
```

Example usage combining both the `merge_sort` and `merge` functions:

```
def merge_sort(items):
    # Get the length of the input list
    items_length = len(items)

    # Create temporary storage for merging
    temporary_storage = [None] * items_length

    # Initialise the size of subsections to 1
    size_of_subsections = 1

    # Iterate until the size of subsections is less than the length of the
list
    while size_of_subsections < items_length:
        # Iterate over the list in steps of size_of_subsections * 2
        for i in range(0, items_length, size_of_subsections * 2):
            # Determine the start and end indices of the two subsections
            # to merge.
            first_section_start, first_section_end = i, min(
                i + size_of_subsections, items_length
            )
            second_section_start, second_section_end = first_section_end, min(
                first_section_end + size_of_subsections, items_length
            )

            # Define the sections to merge
            sections = (first_section_start, first_section_end), (
                second_section_start,
                second_section_end,
            )

            # Call the merge function to merge the subsections
            merge(items, sections, temporary_storage)

        # Double the size of subsections for the next iteration
        size_of_subsections *= 2

    # Return the sorted list
```

```
    return items


def merge(items, sections, temporary_storage):
    # Unpack the sections tuple to get the start and end indices
    # of each section.
    (first_section_start, first_section_end), (
        second_section_start,
        second_section_end,
    ) = sections

    # Initialise indices for the two sections and temporary storage
    left_index = first_section_start
    right_index = second_section_start
    temp_index = 0

    # Loop until both sections have been fully merged
    while left_index < first_section_end or right_index < second_section_end:
        # Check if both sections still have elements to compare
        if left_index < first_section_end and right_index <
second_section_end:
            # Compare elements from both sections
            if items[left_index] < items[right_index]:
                # Place the smaller element into temporary storage
                temporary_storage[temp_index] = items[left_index]
                left_index += 1
            else:  # items[right_index] <= items[left_index]
                temporary_storage[temp_index] = items[right_index]
                right_index += 1
            temp_index += 1

        # If section 1 still has elements left to merge
        elif left_index < first_section_end:
            # Copy remaining elements from section 1 to temporary storage
            for i in range(left_index, first_section_end):
                temporary_storage[temp_index] = items[left_index]
                left_index += 1
                temp_index += 1

        # If section 2 still has elements left to merge
        else:  # right_index < second_section_end
            # Copy remaining elements from section 2 to temporary storage
            for i in range(right_index, second_section_end):
                temporary_storage[temp_index] = items[right_index]
```

```
                right_index += 1
                temp_index += 1

    # Copy sorted elements from temporary storage back to the original list
    for i in range(temp_index):
        items[first_section_start + i] = temporary_storage[i]


# Example usage:
example_list = [33, 10, 59, 26, 41, 58, 18]
sorted_list = merge_sort(example_list)
print(sorted_list)
```

Python includes its own `sort()` method, which uses the **TimSort algorithm**:

```
example_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
example_list.sort()
print(example_list)  # Outputs [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 search methods. Have a look at **visual representations of the search methods** mentioned above to aid your understanding.

Linear search

As mentioned, linear search, also known as sequential 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 demonstrated below (Dalal, 2004):

```python
def sequential_search(target, items):
    # Iterate over the list. If we find the target item, return its index.
    for index in range(len(items)):
        if items[index] == target:
            return index
    # If the target item is not found, return None.
    return None


# Example usage:
items_list = [50, 10, 40, 20, 30]
target_item = 30
result = sequential_search(target_item, items_list)

if result is not None:
    print(f"Item {target_item} found at index {result}.")
else:
    print(f"Item {target_item} not found in the list.")
```

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) ÷ 2 = 100). 63 is less than 100, so we know 63 must be in the left half of the list:

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

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~~	~~10~~	63	80	~~120~~	~~6000~~	~~7400~~	~~8000~~

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~~	~~10~~	63	~~80~~	~~120~~	~~6000~~	~~7400~~	~~8000~~

And it is! We've found our element.

Binary search is O(log n) (University of Cape Town, 2014) and is written below (Dalal, 2004):

```
def binary_search(target, items):
    low, high = 0, len(items) - 1

    # Keep iterating until the low and high cross
    while high >= low:
        # Find midpoint
        mid = (low + high) // 2

        # If item is found at midpoint, return its index
        if items[mid] == target:
            return mid
        # Else, if item at midpoint is less than target,
        # search the second half of the list
        elif items[mid] < target:
            low = mid + 1
        # Else, search the first half
        else:
            high = mid - 1

    # Returns None if item not found
    return None


# Example usage:
sorted_items = [10, 20, 30, 40, 50]
target_item = 30
result = binary_search(target_item, sorted_items)
```

```
if result is not None:
    print(f"Item {target_item} found at index {result}.")
else:
    print(f"Item {target_item} not found in the list.")
```

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task 1

1. Create a Python script called **album_management.py**.
2. Design a class called `Album`. The class should contain:
    - A constructor which initialises the following instance variables:
        - `album_name` - Stores the name of an album.
        - `number_of_songs` - Stores the number of songs within the album.

        - `album_artist` -Stores the album's artist.
    - A `__str__` method that returns a string that represents an Album object in the following format:
      (*`album_name`*, *`album_artist`*, *`number_of_songs`*).
3. Create a new list called `albums1`, add five `Album` objects to it, and print out the list.
4. Sort the list according to the `number_of_songs` and print it out. (You may want to examine **the key parameter in the sort method**).
5. Swap the element at position 1 (index `0`) of `albums1` with the element at position 2 (index `1`) and print it out.
6. Create a new list called `albums2`.
7. Add five `Album` objects to the `albums2` list, and print out the list.
8. Copy all of the albums from `albums1` into `albums2`.
9. Add the following two albums to `albums2`:
    - (Dark Side of the Moon, Pink Floyd, 9)
    - (Oops!... I Did It Again, Britney Spears, 16)
10. Sort the albums in `albums2` alphabetically according to the album name and print out the sorted list.
11. Search for the album *Dark Side of the Moon* in `albums2` and print out the index of the album in the albums2 list.

Auto-graded task 2

In a newly created Python script called **merge_sort.py**:

1. Modify the merge sort algorithm provided in the example usage section above to order a list of strings by string length from the longest to the shortest string.
2. 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.

Auto-graded task 3

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

1. Create a Python script called **sort_and_search.py**. Consider which searching algorithm would be appropriate to use on the given list?
2. Implement this search algorithm to search for the number 9. Add a comment to explain why you think this algorithm was a good choice.
3. Research and implement the **Insertion sort** on this list.
4. Implement a searching algorithm you haven't tried yet in this task on the sorted list to find the number 9. Add a comment to explain where you would use this algorithm in the real world.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

REFERENCES

Csizmar Dalal, A. (2004). *Searching and sorting algorithms: CS117,* Fall 2004 supplementary lecture notes. Retrieved 12 July 2024, from **https://ocd.lcwu.edu.pk/cfiles/Computer%20Science/CS-206/searchSortW12.pdf**

Rowell, E. (2013). Big-O Cheat Sheet: Know thy complexities. Retrieved 16 July 2024, from **http://bigocheatsheet.com/**

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**