LESSON

Sorting and Searching

Visit our website

Introduction

WELCOME TO THE SORTING AND SEARCHING LESSON!

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. At the end there is also a task for you to put into practice what you have learned in this lesson.

INTENDED LEARNING OUTCOMES

You will know that you have succeeded at this task if, by the time you complete it, you are able to:

- Demonstrate an understanding of how abstract data types are stored on computers.
- Demonstrate an understanding of sort techniques used to sort data held in data structures.
- Demonstrate an understanding of search techniques used to search datasets.

Refer to the self-assessment table provided after the programming task near the end of the document for a checklist to assess your own degree of achievement of each of these learning outcomes. Try to gauge your own learning and address any weak areas before submitting your task, which will become part of your formative assessment portfolio.

CONCEPT: DATA SOURCES

Data scientists work with data that comes from various sources. Sometimes records in a database are entered manually, in a traditional data-capture process; sometimes data is downloaded from the Internet or obtained using a web API, or it may be obtained from an open-source or proprietary data supply source, as is the case with many databases used for big-data research. Wherever data comes from, it generally ends up as a file on a computer.

Before we begin working with files, let us consider File Systems. Your computer harddrive is organised into a hierarchical structure of files and directories (washington.edu, n.d.):

- **Files:** A file is a resource on your computer that contains and stores information. There are different types of files that serve different purposes - storing audio, video, text, settings and metadata, program files, and so on. Some common file types you will encounter containing data extracted from databases are CSV, XML, JSON and Python files. You will learn about working with some of these files in this task.

- **Directories:** A directory is a cataloguing structure which contains references to other computer files and possibly other directories. Simply put, a directory is a location for storing files on your computer.

Your file system starts with a **root directory**, typically denoted on Unix by a forward slash ('/'), and on Windows by a drive letter (e.g. 'C:/') (washington.edu, n.d.). Now that we are familiar with file systems, we can dive into the different types of files that can be used as sources of data.

This lesson addresses Unit Standard 115373: Demonstrate an understanding of sort and search techniques used in computer programming.

Specific Outcomes:

1. **SO1: Demonstrate an understanding of how abstract data types are stored on computers.**
2. **SO2: Demonstrate an understanding of sort techniques used to sort data held in data structures. (Range: Including but not limited to: Selection sort, Insertion sort, Bubble sort (at least 2))**
3. **SO3: Demonstrate an understanding of search techniques. (Notes: Demonstrate an understanding of search techniques used to efficiently find data held in data structures) (Range: Including but not limited to: Binary search)**

UNDERSTANDING ABSTRACT DATA TYPES (SO1)

DATA TYPES

Data types define the types and ranges of values that variables can have, and thus what can be stored in these variables. Some examples are given in the table below.

Data type	Explanation	Example

Integer numbers	Integers (**`int`**) are whole numbers (i.e. numbers that have no decimal points).
	
	This is any number from -2147483648 to 2147483648
	
	This is often the most commonly used data type.
		`int number1 = 5;`
		`int number2 = 9865;`
		`int number 3 = -5986;`

Floating point numbers	Floating-point numbers (**`float`**) are similar to integers. The critical difference is that floats can contain decimal points. With floats, in Java, it's essential to add an 'f' at the end of your float (should it contain a decimal point) so that Java knows it has floating-point numbers.	`float num1 = 5.5879f;` `float num2 = -87.48f;`

Long numbers	Long numbers (**`long`**) are very similar to integers; the significant difference is that they can hold much *larger* numbers.
	
	A long variable value can be any number from -9223372036854775808 to 9223372036854775808
	
	A key point to remember with long values is that if you compile your program for only 64-bit-based machines, your
		`long num1 = 489;`
		`long num2 = 999999999;`
		`long num3 = -156;`

	program won't work on 32 bit based systems. Typically integers are the preferred data type, but if you need a number more significant than an ints range, you should use the long data type.	

Double numbers	Doubles (**`double`**) are floats with a much more extensive range.
	
	Doubles can have a total of 15 decimal places.
		`double num1 = 159.126;`
		`double num2 = 1234.156165156156;`
		`double num3 = -156.8778;`

Boolean	Booleans (**`boolean`**, commonly known as true or false values) have only two values, true/false. Booleans are commonly used in **`if`** statements and loops as they are used to tell the program when (and when not) to run a particular portion of code.	`boolean bool1 = true;` `boolean bool2 = false;`

Characters	Characters (**`char`**) are any single character. If you take a look at your keyboard, every key is one character (including spaces, punctuation, and numbers - however, numbers assigned to a **`char`** variable are essentially text and not treated numerically).
	
	It's important to remember that when creating a **`char`** variable in Java, the character assigned to the variable should always be encased inside **single** quotation marks.
		`char char1 = 'a';`
		`char char2 = '5';`
		`char char3 = '!';`
		`char char4 = '☺';`

Strings	Strings (**`string`**) can be considered an upgrade to	`String string1 = "Logan";`

	characters, in that this data type can hold values that are the combination of multiple characters into a single variable. Strings are thus used to hold all text longer than a single character. However, strings can also hold only one character. It's important to remember that when creating a **`string`** variable in Java, the text assigned to the variable should always be placed inside **double** quotation marks.	`String string2 = "Serge is 24";` `String string3 = "a";`

Objects	Objects are a crucial component in Java; almost anything you see in Java will be an object.
	Whenever you see the keyword "new" in Java, it's our way of telling the program that we are creating a new object. In creating an object, we use whatever the object name is in the creation statement. In the example, the objects being created are Pokemon, and so a Pokemon constructor for new objects is called.
		`Pokemon furret = new Pokemon("Normal", 2, "Keen eye");`
		
		`Pokemon pikachu = new Pokemon("Electric", 5, "thunderbolt");`

ABSTRACT DATA TYPES (ADTs)

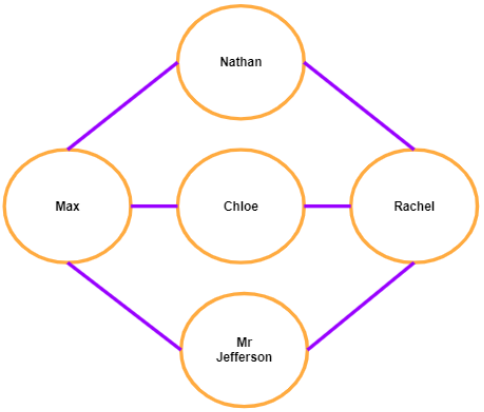
Abstract data types are different from "just plain" data types, in that the term *abstract data type* refers to a way of understanding a conceptual abstraction, a semantic model for storing, accessing, and manipulating data, including the values, operations, and behaviours defined as being allowed for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

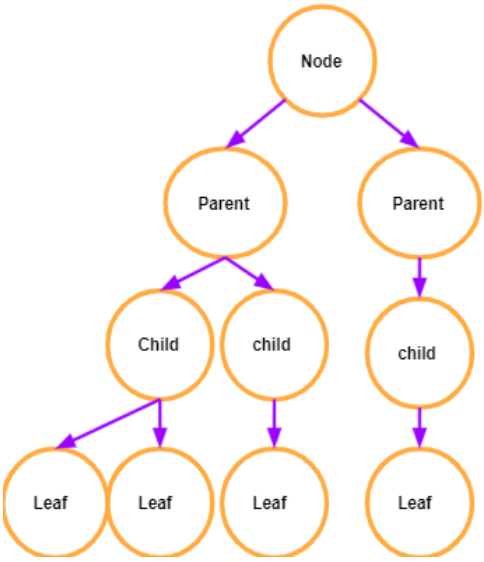
Abstract Data Type	Explanation	Example

Queues	Queues are part of the java.util 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 Java 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.
		`import java.util.LinkedList;`
		`import java.util.Queue;`
		`//Creating new Queue`
		`Queue<String> students = new LinkedList<>();`
		
		`//Adding students to queue`
		`students.add("Kylie");`
		`students.add("Julia");`
		`students.add("Glitch");`
		
		`//output of 'students' will be -> [Kylie, Julia, Glitch]`
		
		`//removes the first student added`
		`students.remove();`
		
		`//output of students will be -> [Julia, Glitch]`

Stacks	Similar to queues, stacks are part of the java.util class. As with queues, you will need to import this class to make use of a stack.
	
	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.
		`import java.util.Stack;`
		`//Create new stack object`
		`Stack<String> myPets = new Stack<>();`
		
		`//Pushing our items to the stack`
		`myPets.push("Dog");`
		`myPets.push("Cat");`
		`myPets.push("Ferret");`
		
		`//Output of MyPets -> [Dog, Cat, Ferret]`

	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 Java, when adding (we use the *.push()* 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)
		`//Removing the last element`
		`myPets.pop();`
		
		`//Output of MyPets -> [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 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.	
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.	

1.1 DATA STRUCTURES TO STORE ABSTRACT DATA TYPES

ARRAYS

The array is a data structure, provided by Java, which stores a fixed number of elements of the same 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 of the same type.

Declaring Arrays

To use an array in a program, a variable to reference the array must be declared, and the array's element type must be specified. The array can have any element type. All elements that are in the array will have the same data type. To declare a variable, use the following syntax:

```
elementType[] arrayVariable;
```

The following example declares a variable `numArray` that references an array that stores elements of the type `double`.

```
double[] numArray;
```

Creating Arrays

After declaring an array variable you can create an array using the new operator. The syntax is as follows:

```
elementType[] arrayVariable = new elementType[arraySize];
```

The following example declares an array variable called numArray, creates an array of 5 double elements and assigns the reference to the array to numArray.

```
double[] numArray = new double[5];
```

The array size must be given when space for an array is allocated. The size cannot be changed after an array is created. To obtain the size of an array, use `arrayVariable.length`.

Numeric data types are assigned the default value 0, `char` types are assigned the value `\u0000` and boolean types are assigned the value `false` when an array is created.

Array Indices

To access array elements you use an index. Array indices range from 0 to `arrayVariable.length - 1`. For example, the array created above (`numArray`), holds five elements with indices from 0 to 4. You can represent each element in the array using the following syntax:

```
arrayVariable[index];
```

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

Initialising an Array

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**:

```
numArray[0] = 23.6;
numArray[1] = 45.12;
numArray[2] = 8.4;
numArray[3] = 77.7;
numArray[4] = 1.34;
```

You can use shorthand notation in Java, known as the array initialiser, which combines the declaration, creation and initialisation of an array in one statement. The syntax for an array initialiser is as follows:

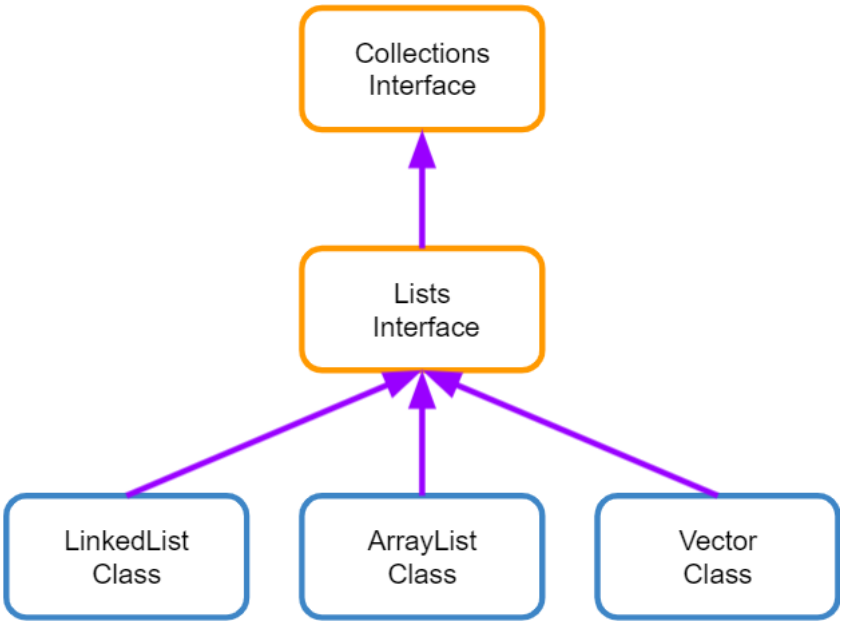
```
elementType[] arrayVariable = {value1, value2, ..., valueN};
```

For example, the following array initialiser statement is equivalent to the example above in which each element was assigned a value individually:

```
double[] numArray = {23.6, 45.12, 8.4, 7.77, 1.34};
```

LISTS

Lists are an interface in Java that is used to implement a large number of different list types. In the image below, you can see the relationship between a list and the classes that implement it.



Unlike many other data structures in Java, a list is not created in the way you may be familiar with.

```
List<String> listOne = new List<String>;
```

This will cause your program to contain a syntax error. The idea of a list is to assist us in implementing the following...

```
//ArrayList
List<String> arrayList = new ArrayList<>();
```

```
//LinkedList
List<String> linkedList = new LinkedList<>();
```

```
//Stacks
List<String> stackList = new Stack<>();
```

```
//Vectors
List<String> vectorList = new Vector<>();
```

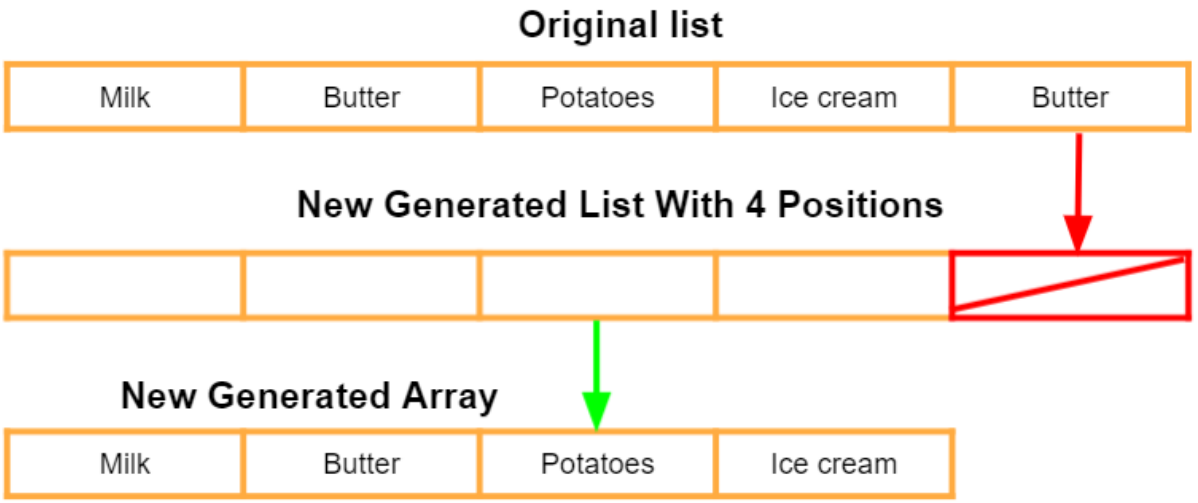
ARRAY LISTS

Array lists are our most commonly used data structure; these ArrayLists have an unlimited number of slots for how much data they can store.

So then, why would we use any other data structure if arrays are so powerful? The reason for us using different data structures (other than features) is because array lists can be viewed as "slower" (more information about speed when you learn about time complexity.)

ArrayLists become slower with the more data we feed it.

When adding/removing data Java takes all the data from our current list and copies it over to an entirely new list with the added/removed data we provided, take a look at this example below.



As you can see above, our original array list has a total of 5 values, but we've accidentally added butter twice. So when we remove it, our program needs to create a whole new list with only four values and then copy that data over to an entirely new array list.

While this may not be a massive problem with arrays with a small amount of data, imagine we have an array with over 9000 sets of data, and we need to remove one; this would cause our program to take a long time to complete this process.

Declaring and creating arraylists

Declaring array lists is slightly different compared to creating an array. Because ArrayLists have no size limits (i.e. after creating an array list, you can add or remove data at any time)

With ArrayList, you have to specify what data type your array will be using...

```
List<String> stringList = new ArrayList<>();
List<Integer> intList = new ArrayList<>();
List<Double> doubleList = new ArrayList<>();
```

As you can see above, inside our diamond brackets **<>** we place our data type. Unlike when creating a variable, we used the complete word (rather than int, we would use Integer).

Adding/Removing data from an arraylist

Now that we've created an ArrayList, how do we add values to our list? Well, there are a few ways we can do this.

- **Arraylist default values**
    - Let's say we already have data that we know we want to be placed inside our array; we can use something known as **Arrays.asList** to achieve this.

```
List<String> computerParts = new ArrayList<>(Arrays.asList("test1",
"test2", "test3"));
```

    - As you can see within our brackets at the end of our array creation, we've added the **asList** method. This can take any number of values, and these will automatically be placed inside the array when it's created.
- **Adding data later on**
    - As we know, sometimes we won't know all the data we need right at the start. This is why we may need to add data later on within the program. This is where the **.add** method comes into play.

```
List<String> computerParts = new ArrayList<>();

computerParts.add("i9 processor");
```

```
computerParts.add("GTX 3080");
computerParts.add("5TB SSD");
computerParts.add("16GB RAM stick");

System.out.println(computerParts);
/*output
[test1, test2, test3, i9 processor, GTX 3080, 5TB SSD, 16GB RAM stick]
*/
```

- In the example above, you can see that all the data we added to our array is printed out. You can use this at any part of your program.

- **Removing data**
  - Similarly to how we may want to add data at a later point in our program, we may want to do the same thing with removing data from our array. To achieve this, we'll make use of the **.remove()** method. This takes in one parameter, which is the index of the value you want to remove.
  - Let's say that our last PC already has a 16GB ram stick; we no longer need to purchase this from the store, so we want to remove it from our list.

```
computerParts.remove(3);
System.out.println(computerParts);
/*output
[i9 processor, GTX 3080, 5TB SSD]
*/
```

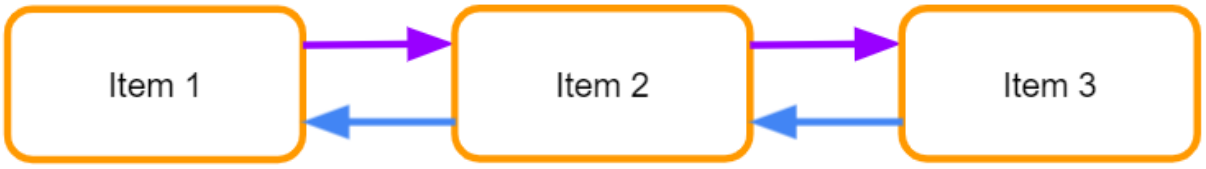
  - We removed the third element because the RAM stick was the 4th item in our list (as we count from 0 in programming).

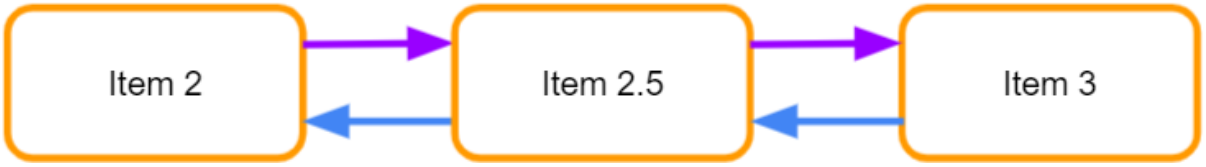
LINKED LISTS

Linked lists differ from Arraylists in the sense of how they add and remove data. ArrayLists only contain one list, which is copied over to a new list after an element changes.

Linked lists data types are actually created with two lists. This is where performance is boosted compared to an ArrayList. Additionally, linked lists contain many of the same features as an ArrayList but with the combination of a queue (first in, first out).

So let's take a look at this in more detail.





Because of the way LinkedLists are created, the way elements are placed into a list is by taking the one/two nodes (item1, item2, and item3 are nodes) and setting the value between those two nodes. This is where it becomes more efficient than ArrayList because we're not entirely recreating the array. As we know, sometimes we won't know all the data we need right at the start. This is why we may need to add data later on within the program. This is where the .**add** method comes into play.

Declaring and creating LinkedLists

Because we're making use of the lists interface the way we create a linkedList is very similar to other list creations.

```
List<String> linkedList = new LinkedList<>();
```

/* If you want to know how to make a list that uses other data types and not just a string refer back to "Declaring and creating ArrayLists" */

Adding/Removing data from an LinkedList

Now that we've created an ArrayList, how do we add values to our list? Well, there are a few ways we can do this.

- **LinkedList default values**
    - Let's say we already have data that we know we want to be placed inside our array; we can use something known as **Arrays.asList** to achieve this.

```
List<String> testSubjects = new LinkedList<>(Arrays.asList("CN2971",
"KS8G"));
```

    - As you can see within our brackets at the end of our array creation, we've added the **asList** method. This can take any number of values, and these will automatically be placed inside the array when it's created.
- **Adding data later on**

- As we know, sometimes we won't know all the data we need right at the start. This is why we may need to add data later on within the program. This is where the **.add** method comes into play.

```
List<String> testSubjects = new LinkedList<>(Arrays.asList("CN2971",
"KS8G"));

testSubjects.add("SP");
testSubjects.add("MMKM");

System.out.println(testSubjects);
/*output
* [CN2971, KS8G, SP, MMKM]
*/
```

- In the example above, you can see that all the data we added to our array is printed out. You can use this at any part of your program.

- **Removing data**
  - Similarly to how we may want to add data at a later point in our program, we may want to do the same thing with removing data from our array. To achieve this, we'll make use of the **.remove()** method. This takes in one parameter, which is the index of the value you want to remove.
  - Let's say that our last subject was released back into the wild, so we want to remove it from our list.

```
testSubjects.remove(3);

System.out.println(testSubjects);
/*output
* [CN2971, KS8G, SP]
*/
```

- We removed the third element because the MMKM subject was the 4th item in our list (as we count from 0 in programming).

ArrayList vs LinkedList

So now that we've gone over these two arrays, which one should you use?

In many cases, you will almost always use an ArrayList. The benefits in terms of data searching and memory usage compared to a LinkedList are far more superior.

The times we use LinkedList is in the scarce case that it needs to be used in a specific algorithm, and even with that in mind, those events will only appear very rarely.
It's suggested that you play around with both and do your own research for both of these; however, ArrayLists will most likely be the most commonly used data structure you'll be using.

CONCEPT: 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.

CONCEPT: 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:

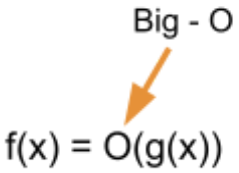
1. You will get to analyse and learn about algorithms that have been developed and tested by experienced software engineers, providing a model for writing good algorithms.
2. As these algorithms have already been implemented by others, you will see how to use them without writing them from scratch, i.e. 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.

2. SORTING ALGORITHMS (SO2)

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:



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.

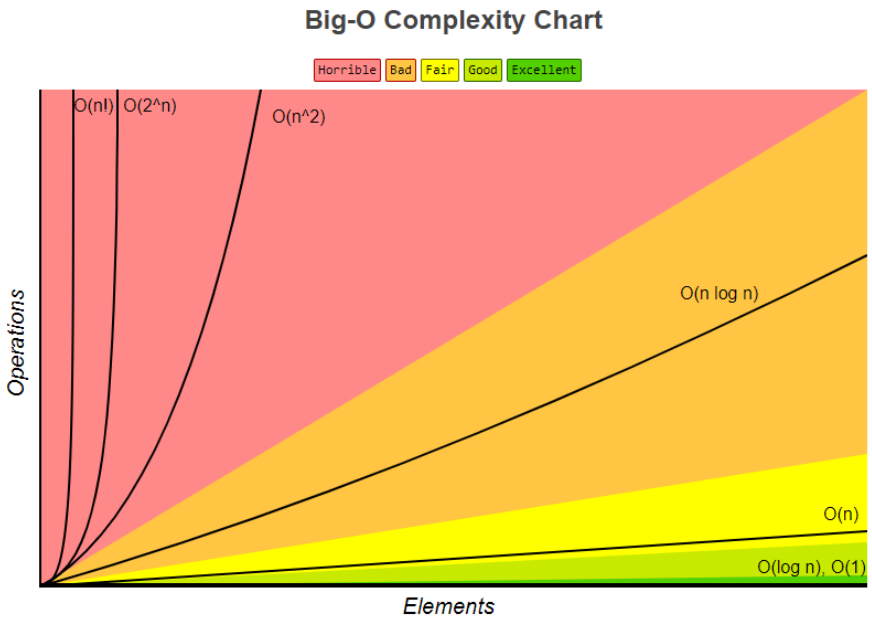


Image source: **http://bigocheatsheet.com/**

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
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: **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.

SORTING: DEEP DIVE

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

```
void bubbleSort(int ar[]) {
    for (int i = (ar.length - 1); i >= 0; i--) {
        for (int j = 1; j <= i; j++) {
            if (ar[j-1] > ar[j]) {
                int temp = ar[j-1];
                ar[j-1] = ar[j];
                ar[j] = temp;
            }
        }
    }
}
```

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:

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 above pivot points so the list is sorted

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

```
public void quicksort(int[] list, int low, int high) {
    if (low < high) {
        int mid = partition(list,low,high);
        quicksort(list, low, mid-1);
        quicksort(list, mid+1, high);
    }
}
```

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

```
public int partition(int[] list, int low, int high) {

    // The pivot point is the first item in the subarray
    int pivot = list[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.
```

```
    do {
        // 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 && list[high] >= pivot) {
            high--;
        }
        if (low < high) {
            // found a smaller number; swap it into position
            list[low] = list[high];
            // now look for a number larger than the pivot point
            while (low < high && list[low] <= pivot) {
                low++;
            }

            if (low < high) {
                // found one! move it into position
                list[high] = list[low];
            }
        }
    } while (low < high);

    // move the pivot back into the array and return its index
    list[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	**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)	
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)	
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)	
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 (Dalal, 2004):

```
public void mergeSort(int[] list, int low, int high) {
    if (low < high) {
        // find the midpoint of the current array
        int mid = (low + high)/2;
        // sort the first half
        mergeSort(list, low, mid);
        // sort the second half mergeSort(list,mid+1,high);
        // merge the halves
        merge(list, low, high);
    }
}
```

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 (Dalal, 2004):.

```
public void merge(int[] list, int low, int high) {

    // temporary array stores the ''merge'' array within the method.
     int[] temp = new int[list.length];

     // Set the midpoint and the end points for each of the subarrays
    int mid = (low + high)/2;
    int index1 = 0;
    int index2 = low;
    int index3 = mid + 1;

    // Go through the two subarrays and compare, item by item,
    // placing the items in the proper order in the new array
    while (index2 <= mid && index3 <= high) {
        if (list[index2] < list[index3]) {
        temp[index1] = list[index2];
        index1++; index2++;
        }
        else {
        temp[index1] = list[index3];
        index1++; index3++;
        }
    }

    // if there are any items left over in the first subarray, add them to
    // the new array
```

```
    while (index2 <= mid) {
    temp[index1] = list[index2];
    index1++;
    index2++;
    }

    // if there are any items left over in the second subarray, add them
    // to the new array
    while (index3 <= high) {
    temp[index1] = list[index3];
    index1++;
    index3++;
    }

    // load temp array's contents back into original array
    for (int i=low, j=0; i<=high; i++, j++) {
    list[i] = temp[j];
    }
}
```

As mentioned earlier, Merge Sort is used in the Collections Framework as `Collections.sort()`. For example:

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {

        List<Integer> myList = new ArrayList<>();
        myList.add(10);
        myList.add(2);
        myList.add(23);
        myList.add(14);

        System.out.println("List: " + myList);

        Collections.sort(myList);
        System.out.println("Sorted List: " + myList);
    }
}
```

3. SEARCHING ALGORITHMS (SO3)

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.

SEARCHING: DEEP DIVE

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. Let us look at these two in more detail. 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):

```
public int sequentialSearch(int item, int[] list) {
    // if index is still -1 at the end of this method, the item is not
    // in this array.
    int index = -1;
    // loop through each element in the array. if we find our search
    // term, exit the loop.
    for (int i=0; i<list.length; i++) {
        if (list[i] == item) {
            index = i;
            break;
        }
    }
    return index;
}
```

Output:

```
List: [10, 2, 23, 14]
Sorted List: [2, 10, 14, 23]
```

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) (UCT, 2014) and is written below (Dalal, 2004):

```
public int binarySearch(int item, int[] list) {
    // if index = -1 when the method is finished, we did not find the
    // search term in the array
    int index = -1;

    // set the starting and ending indexes of the array; these will
```

```
    // change as we narrow our search
    int low = 0;
    int high = list.length-1;
    int mid;

    // Continue to search for the search term until we find it or
    // until our ''low'' and ''high'' markers cross
    while (high >= low) {
        mid = (high + low)/2; // calculate the midpoint of the current array
        if (item < list[mid]) { // value is in lower half, if at all
            high = mid - 1;
        }
        else if (item > list[mid]) {
            // value is in upper half, if at all
            low = mid + 1;
            }
        else {
                // found it! break out of the loop
            index = mid;
            break;
        }
    }
    return index;
}
```

It is worth noting that the Java Collections Framework uses the binary search algorithm in the form of `Collections.binarySearch()`. This means that you need to sort your list before you search it. See the example below:

```
import java.util.*;

public class BinarySearch{
   public static void main(String[] args) {

      List<String> list = new ArrayList<>();
      list.add("Java");
      list.add("Programming");
      list.add("Is");
      list.add("Fun");

      // Sort the list into alphabetical order
      Collections.sort(list);
      System.out.println("Sorted List: " + list);

      int index = Collections.binarySearch(list, "Java");
      System.out.println("'Java' in List is at " + index);
```

```
      index = Collections.binarySearch(list, "Python");
      System.out.println("'Python' in List is at " + index);
    }
}
```

Output:

```
Sorted List: [Fun, Is, Java, Programming]
'Java' in List is at 2
'Python' in List is at -5
```

Note what the output prints as 'Python's' index: -5. This is because if you were to insert this element into the list, it would have an index value of 4. It sounds strange but it is actually calculated using *(-(insertion_index) - 1)*, which is *(-(-5) -1)*.

Compulsory Task 1

- Create a Java file called **ArrayLists.java**

- Design a class called Album. The class should contain:
    - The data fields *albumName* (String), *numberOfSongs* (int) and *albumArtist* (String).
    - A constructor that constructs a Album object with the specified *albumName*, *numberOfSongs*, and *albumArtist*.
    - The relevant *get* and *set* methods for the data fields.
    - A *toString()* method that returns a string that represents an Album object in the following format:
      (*albumName, albumArtist, numberOfSongs*)

- Create a new ArrayList called *albums1*, add 5 albums to it and print it out *(You may want to look back to the Java Collections Framework task for guidance)*.

- Sort the list according to the *numberOfSongs* and print it out.

- Swap the element at position 1 of *albums1* with the element at position 2 and print it out.

- Create a new ArrayList called *albums2*.

- Using the *addAll* method 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

Using the following ArrayList: [“blue”, “six”, “hello”, “game”, “unorthodox”, “referee”, “ask”, “zebra”, “run”, “flex”]

- Create a Java file called **BubbleSort.java**
- Implement the Bubble sort algorithm on the ArrayList and print out the sorted list.

Compulsory Task 3

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

- Create a Java file called Sort&Search.java
  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 the 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.

Self-assessment table

Intended learning outcome (SAQA US 115373)	Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- I can successfully:	
Demonstrate an understanding of how abstract data types are stored on computers (SO1).	Identify different abstract data types used in computer programming. Range: Including but not limited to: Queue, stack, graph, tree data types in a computer.	Identify different data structures used to store abstract data types in a computer. Range: Including but not limited to: Arrays, lists, linked lists.
Demonstrate an understanding of sort techniques used to sort data held in data	Demonstrate different types of sort techniques Range: Including but not limited to: Selection sort,	Demonstrate the working of different types of sort techniques Range: Including but not

structures (SO2).	Insertion sort, Bubble sort (at least 2)		limited to: Selection sort, Insertion sort, Bubble sort (at least 2)	
Demonstrate an understanding of search techniques (SO3).	Demonstrate (identify and apply) different types of search techniques Range: Including but not limited to: Binary search		Explain the working of different types of search techniques Range: Including but not limited to: Binary search	
Intended learning outcomes (Hyperion extension)	Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- I can successfully:			
Understand sorting algorithms	Understand how bubble sort works and be able to implement it in code	Understand how merge sort works and be able to implement it in code	Understand how merge sort works and be able to implement it in code	
Understand searching algorithms	Understand how linear search works	Implement linear sort in code	Understand how binary search works	Implement binary search in code
Understand computational complexity	Calculate the computational complexity of sorting algorithms		Calculate the computational complexity of searching algorithms	

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**