Algorithms and Data Structures

Whitepaper

Author: Nathan Clarke

Date: 29 Oct 2019

Table of Contents

Overview

Algorithms are defined by the steps which are performed for the solution for a certain task or a problem.

Data structures are the means of how data is organised in the computer's memory.

In a computer program if the data is optimally organised it will improve the efficiency of how it will be processed and the fewer steps required for computing would reduce the time and amount of resources required.

So, this is why algorithms and data structures are critical to the performance of a program.

The example code in this whitepaper will be shown using Java.

Basic Algorithms

Sorting

One of the most fundamental and most useful algorithms is sorting. There are many reasons to why we want to sort data in some order.

For example, let's say we have a group of employees and want to determine how long (in years) they have been employed, then have it sorted in descending order (highest to lowest value).

We would firstly retrieve the employee start date, compare it against the current date and then return the difference in years.

We can store these values into an int array.

```
    int[] employedYears = { 7, 3, 2, 9, 11, 1 };
```

We now have the data in an unordered list, the next task to sort the data in descending order. We can do this using the Java 8 Stream to first box the array to sort in reverse order.

```
import java.util.Arrays;
import java.util.Collections;

public class SortingAlgorithmImpl1 {

    static int[] employedYears = { 7, 3, 2, 9, 11, 1 };

    static int[] descIntArr(int[] arrList) {
        return
                Arrays
                    .stream(arrList).boxed()
                    .sorted(Collections.reverseOrder())
                    .mapToInt(Integer::intValue)
                    .toArray();
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(descIntArr(employedYears)));
    }
}
```

The array has now been sorted in descending order; the console output result can be seen below.

[11, 9, 7, 3, 2, 1]

Although the code above works and there are many sorting algorithms some are faster, slower, uses more resources or fewer resources this is

a great reason to compare the differences and determine the best algorithm.

Another implementation of the descending sorting algorithm can be seen below:

```
import java.util.Arrays;

public class SortingAlgorithmImpl2 {

    static int[] employedYears = { 7, 3, 2, 9, 11, 1 };

    static void SelectionSort (int[] list)
    {
        for (int i = list.length - 1; i > 0; i --)
        {
            int index = 0, temp;   //initialize to subscript of first element
            for(int j = 1; j <= i; j ++) //locate smallest element between
positions 1 and i.
            {
                if(list[j] < list[index])
                    index = j;
            }
            temp = list[index];   //swap smallest found with element in position i.
            list[index] = list[i];
            list[i] = temp;
        }
        System.out.println(Arrays.toString(list));
    }

    public static void main(String[] args)
    {
        SelectionSort(employedYears);
    }
}
```

Linear Search

We now know how to sort an array data, though we may want to search for a specific element within an array. How do we go about doing this?

The linear search algorithm can be used to search through an entire array to find an element.

In the example below, it checks whether an element is found within an array and returns its position, else the element does not exist.

```
public class LinearSearchAlgorithm {

    static int[] employedYears = { 7, 3, 2, 9, 11, 1 };

    static void linearSearch(int[] list, int value) {
        int n = list.length;
        for (int i = 0; i < n; i++) {
            if (list[i] == value) {
                System.out.println("The element has been found at position: " + i);
```

```
                return;
            }
        }
        System.out.println("The element is not in the array.");
        return;
    }

    public static void main(String[] args) {
        //Value Exists in employedYears Array
        linearSearch(employedYears, 3);
        //Value Does Not Exist in employedYears Array
        linearSearch(employedYears, 12);
    }
}
```

Bubble Sort

The bubble sort algorithm incrementally goes through an array and compares two values a pair at a time to determine whether it's in order. If the values are in order nothing changes if they are not the values swapped around. It then moves onto the next pair and repeats the process.

We will use the following array values as an example.

{ 7, 3, 2, 9, 11, 1}

We compare the first two values 7 and 3 to see whether they are in order (ascending order). If the values are not in order, they are swapped.

We can see 7 and 3 are not in order so they are swapped.

{ 3, 7, 2, 9, 11, 1};

The next pair is 7 and 2 which is not in order and is swapped.

{ 3, 2, 7, 9, 11, 1};

The next pair is 7 and 9 which is in order, so we continue to the next pair.

{ 3, 2, 7, 9, 11, 1}

The next pair is 9 and 11 which is in order, so we continue to the next pair.

{ 3, 2, 7, 9, 11, 1 };

The final pair is 11 and 2 which is not in order and is swapped.

{ 3, 2, 7, 9 , 1, 11 };

Iteration one has been complete, and we repeat this process until the array becomes ordered and no more values can be swapped.

{ 1, 2, 3, 7, 9, 11 };

The program below will execute the bubble sorting algorithm.

```
import java.util.Arrays;

public class BubbleSortAlgorithm {

    static int[] employedYears = { 7, 3, 2, 9, 11, 1 };

    static void bubbleSort(int[] list) {
        int n = list.length;
        boolean swapped;
        do
        {
            swapped = false;
            for (int i = 0; i < n-1; i++) {
                if (list[i] > list[i+1]) {
                    int temp = list[i];
                    list[i] = list[i+1];
                    list[i+1] = temp;
                    swapped = true;
                }
            }
        } while (swapped == true);
        System.out.println(Arrays.toString(list));
    }

    public static void main(String[] args) {
        bubbleSort(employedYears);
    }
}
```

The array has now been sorted in ascending order; the console output result can be seen below.

[1, 2, 3, 7, 9, 11]

Algorithmic Analysis

Merge Sort

Another means of sorting is the merge sort. Merge sort has two main steps, the first being splitting and then merging.

Let's say we have the following numbers: 7, 8, 1, 4, 5, 6, 3, 2 and we want to sort in ascending order using the merge sort algorithm.

Note * (The data within the split arrays have been referred to as groups: Group 1 | Group 2 | Group 3 | Group 4.)

We would first start by splitting the numbers, so we have two groups.

7, 8, 1, 4 | 5, 6, 3, 2

The number will be split again, so we have four groups.

7, 8 | 1, 4 | 5, 6 | 3, 2

This process is repeated until each group only contains two values. The groups are then checked to determine whether the values need to be switched.

7, 8 | 1, 4 | 5, 6 | Switched to 2, 3

Each of the numbers in the groups has been sorted, now we need to merge them.

We start by getting groups 1 and 2 and begin the merging.

7, 8 | 1, 4

The values of both groups are then compared. When we compare value 1 against 7 and can see 1 is smaller, therefore, it will be moved to the front of group 1.

We then compare values 4 against 7, 4 also smaller, therefore, both numbers can be placed before group 1 and the merge between the two groups has been completed.

1, 4, 7, 8

The same merge will be performed on groups 3 and 4.

5, 6 | 2, 3

We then compare values 2 against 5, 2 is smaller, therefore, it will be moved to the front of group 3.

We then compare values 3 against 5, 3 also smaller, therefore, both numbers can be placed before group 3 and the merge between the two groups has been completed.

2, 3, 5, 6

Now we have two groups

1, 4, 7, 8 | 2, 3, 5, 6

The values from group 2 are compared against the values of group 1 to find out which one is smaller and then we'll merge.

We will compare values 1 against 2, 1 is smaller so we compare 2 against 4, two is smaller so we can place 2 after 1.

We keep moving up the group 2 list and to compare against the group 1 list. This will process will be repeated until we have a sorted list.

1, 2, 3, 4, 5, 6, 7, 8

The program below will execute the merge sort algorithm.

```
import java.util.Arrays;

public class MergeSortAlgorithm {

    static void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two sub arrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temp arrays
        int _l[] = new int [n1];
        int _r[] = new int [n2];

        // Copy data to temp arrays
```

```
        for (int i = 0; i < n1; ++i)
            _l[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            _r[j] = arr[m + 1+ j];

        // Initial indexes of first and second sub arrays
        int i = 0, j = 0;

        // Initial index of merged sub array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (_l[i] <= _r[j])
            {
                arr[k] = _l[i];
                i++;
            }
            else
            {
                arr[k] = _r[j];
                j++;
            }
            k++;
        }

        // If reminder exists copy elements of _l[]
        while (i < n1)
        {
            arr[k] = _l[i];
            i++;
            k++;
        }

        // If reminder exists copy elements _r[]
        while (j < n2)
        {
            arr[k] = _r[j];
            j++;
            k++;
        }
    }

    static void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            // Get the middle point in the array
            int m = (l+r)/2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    public static void main(String args[])
    {
        int[] employedYears = { 7, 8, 1, 4, 5, 6, 3, 2 };

        //Displaying Unsorted Array
        System.out.println("Unsorted Array: " + Arrays.toString(employedYears));

        //Sort Array
```

```
        sort(employedYears, 0, employedYears.length-1);

        //Display Sorted Array
        System.out.println("Sorted array: " + Arrays.toString(employedYears));
    }
}
```

Binary Search

The binary search algorithm searches for a specific value in an ordered list. The array must be in a sorted order to be able to use this algorithm.

For the following tutorial we'll use the following array dataset:
{1, 2 ,3 ,4, 5, 6, 7, 8, 9}

The binary search works by going to the middle of the list (median) and then determines whether that's the value it is looking for. If it's not the case it will perform a check to determine whether the value is greater or less than the current median. In the case, the value is greater all other values below and equal to the current median are then excluded from the search.

The new median for the remainder of the data is then found, and the process is repeated until the value is found or there are no more values left in the case it doesn't exist.

The program below will execute the binary search algorithm.

```
public class BinarySearch {

    static int[] employedYears = { 1, 2, 3, 4, 5, 6, 7, 8 };

    static boolean binarySearch(int findValue, int[] list, int low, int high) {
        if (low > high) {
            System.out.println("Value has not been found.");
            return false;
        }

        int middle = (low + high) / 2;

        if (findValue == list[middle]) {
            System.out.println("The value has been found at position: " + middle);
            return true;
        }
        else if (findValue > list[middle]) {
            return binarySearch(findValue, list, middle+1, high);
        }
        else {
            return binarySearch(findValue, list, low, middle-1);
        }
    }

    public static void main(String args[])
    {
```

```
        binarySearch(7, employedYears, 0, 9);
    }
}
```

Big O Time and Space Notation

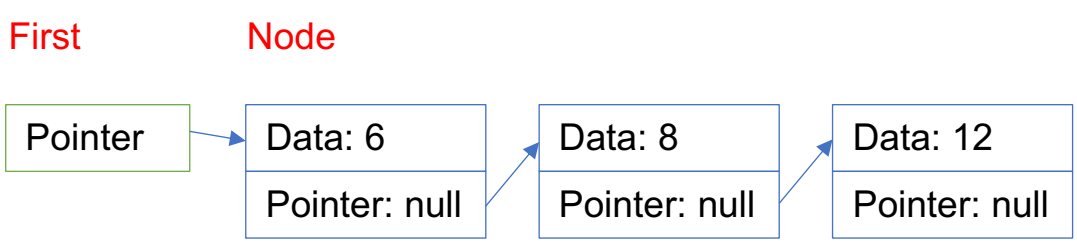
The underlying principle of the Big O time and space notation is to analysis how many steps it takes to complete an algorithm being able to determine how much memory space the algorithm needs in relation to its input.

This is quite a complex and mathematical process which I will not get into here. However, I would recommend using a tool that analyses how much memory you're using and how efficient your code is there are many tools you can find online which do this.

Basic Data Structures

Linked Lists

A linked list is a linear data structure that does not store the elements in contiguous memory locations. As shown in the illustration below, the elements in a linked list are linked using pointers:



Stack

A special type of the linked list is the stack, which is based on the last in first out data structure (LIFO). This means the last element that goes in will be the first element to be removed and so on.

Generally, a stack will only support fast access to one side of the list rather than both and prevents elements that are not on the top of the stack to be access or removed.

The stack has three main operations: pushing, pop, peek. The push operation moves new elements to the top of the stack. The pop operation removes and returns the top element from the stack. The peek operation simply returns the top element of the stack without removing it.

The program below will execute the stack data structure.

```
import java.util.Stack;

public class StackImpl {

    public static void main(String args[])
    {
        // creates a new stack
        Stack stack = new Stack();

        // add values to the stack
        stack.push(8); // [8]
        stack.push(12); // [8, 12]

        // Displaying the stack
        System.out.println(stack.peek()); // prints 8

        // add value to the stack
```

```
        stack.push(16); // [8, 12, 16]

        //Casting stack.pop() value to int and remove last in
        int x = (int) stack.pop();

        // Displaying the stack
        System.out.println(x); // prints 16
        System.out.println(stack); // prints [8, 12]
    }
}
```

Queue

Another special type of linked list is a queue, which is based on the first-in, first-out data structure (FIFO). This means that the first element that goes in will be the first element out, and so on.

Similarly, to the stack, the queue algorithm doesn't remove the median value and will only support the removal at the head of the queue and the insertion at the end of the queue.

The queues have three main operations: enqueue, dequeue, and peek. The enqueue is used to add an element to the end of a queue. The dequeue used to remove the first element in the queue. The peek operation just like the stack simply returns the first value in the queue, without removing it.

The program below will execute the queue data structure.

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueImpl {

    public static void main(String args[])
    {
        Queue queue = new LinkedList();

        //add value to queue
        queue.add(8); // [8]
        queue.add(12); // [8, 12]

        //Display queue
        System.out.println(queue.peek()); // prints 8

        //add value to queue
        queue.add(16); // [8, 12, 16]

        //Delete the head
        queue.poll();

        //Display queue
        System.out.println(queue); // prints [12, 16]
    }
}
```