Data Structures and Algorithms

# Definitions

Definition of Data Structure

It organizes and stores data, they all differ in the way that they store the data.

Definition of Algorithm

It described the steps you must perform to accomplish a specific task.

Definition of Implementation

The code you write to perform the task, an algorithm can have many implementations.

# Big-O Notation

Definition

The Big-O Notation gives a way of comparing the time and space complexity of different algorithms in an objective manner. It measures the worst-case time or space complexity relative to the input.

* **O (1) – Constant**
* **O (log2n) – Logarithmic**
* **O (n) – Linear**
* **O (nlog2n) – Linearithmic**
* **O (n2) – Quadratic**
* **O (2n) – Exponential**
* **O (n!) – Factorial**

By using Big-O Notation, developers can predict how algorithms will perform as the input size grows and select the most efficient algorithm based on scalability and resource use.

# Stable and Unstable Sort

Definition

A stable sort is when the algorithm preserves the relative ordering of duplicate items, an unstable sort does not preserve the relative order.

# Arrays

Definition

They are not a dynamic data structure, which means once you create an array you cannot change its size, it is a continuous block in memory. Every element occupies the same amount of space in memory.

If we know the index of an element, the time to retrieve the element will be the same.

Using the *formula* **x + i \* y** to reach the memory address of the **ith** element, the size of each element being **y** and starting at the memory address **x**.

For arrays:

**O (1) –** Retrieve with index

**O (n) –** Retrieve without index

**O (n) –** Add an element to a full array

**O (1) –** Add an element to the end of an array (has space)

**O (1) –** Insert or update an element at a specific index

**O (1) –** Delete an element by setting it to null

**O (n) –** Delete an element by shifting elements

The general rule:

If it doesn’t have a loop, it’s *Constant Time*

If it requires a single loop, it’s *Linear Time*

If it requires a loop inside of a loop, it’s *Quadratic Time*

# Bubble Sort

It’s an in-place and stable algorithm and it has a time complexity of **O (n2)**.

public class bubbleSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        for (int lastUnsortedIndex = intArray.length - 1; lastUnsortedIndex > 0; lastUnsortedIndex--) {

            for (int i = 0; i < lastUnsortedIndex; i++) {

                if (intArray[i] > intArray[i + 1]) {

                    swap(intArray, i, i + 1);

                }

            }

        }

        for (int i = 0; i < intArray.length - 1; i++) {

            System.out.println(intArray[i]);

        }

    }

    public static void swap(int[] array, int i, int j) {

        if (i == j) {

            return;

        }

        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

}

# Selection Sort

It’s an in-place and unstable algorithm and it has a time complexity of **O(n2)**, but does not require as much swapping as the *Bubble Sort*.

public class selectionSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        for (int lastUnsortedIndex = intArray.length - 1; lastUnsortedIndex > 0; lastUnsortedIndex--) {

            int largest = 0;

            for (int i = 1; i <= lastUnsortedIndex; i++) {

                if (intArray[i] > intArray[largest]) {

                    largest = i;

                }

            }

            swap(intArray, largest, lastUnsortedIndex);

        }

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

    public static void swap(int[] array, int i, int j) {

        if (i == j) {

            return;

        }

        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

}

# Insertion Sort

It’s an in-place and stable algorithm and it has a time complexity of **O (n2)**.

public class insertionSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        for (int firstUnsortedIndex = 1; firstUnsortedIndex < intArray.length; firstUnsortedIndex++) {

            int newElement = intArray[firstUnsortedIndex];

            int i;

            for (i = firstUnsortedIndex; i > 0 && intArray[i - 1] > newElement; i--) {

                intArray[i] = intArray[i - 1];

            }

            intArray[i] = newElement;

        }

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

}

# Shell Sort

It’s an in-place and unstable algorithm and it has a time complexity of **O (n2)** but it can perform much better because of the gap, it doesn’t require as much shifting as insertion sort so it usually performs better and it doesn’t require as much swapping as bubble sort so it usually performs better, can utilize the Knuth Sequence to calculate the gap **(3k** **– 1) / 2** where k is the length of the array and the result should be a close value to the length of the array, without being greater than the array.

public class shellSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        for (int gap = intArray.length / 2; gap > 0; gap /= 2) {

            for (int i = gap; i < intArray.length; i++) {

                int newElement = intArray[i];

                int j = i;

                while (j >= gap && intArray[j - gap] > newElement) {

                    intArray[j] = intArray[j - gap];

                    j -= gap;

                }

                intArray[j] = newElement;

            }

        }

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

}

# Recursion

Factorial Algorithm

A method is a recursive method when it calls itself

**Iterative** implementation usually runs faster and uses less memory than the **Recursive** one

Seach up*: Tail recursion*

public class factorialAlgorithm {

    public static void main(String[] args) {

    }

    public static int recursiveFactorial(int num) {

        if (num == 0) {

            return 1; // If this condition doesn't exist, stack overflow happens

        }

        return num \* recursiveFactorial(num - 1); // Uses call stack

    }

    public static int iterativeFactorial(int num) {

        if (num == 0) {

            return 1;

        }

        int factorial = 1;

        for (int i = 1; i <= num; i++) {

            factorial \*= i;

        }

        return factorial;

    }

}

# Merge Sort

It’s a not in-place and stable alrogithm and it has a time complexity of **O (nlog2n)**.It’s a Divide and Conquer algorithm, divided into two phases: Splitting (logical) and Merging. An unsorted array will be divided into two smaller arrays (extra goes right) until all the arrays have one element each – these are sorted. Every left/right pair of sibling arrays will be merged into a sorted array, then again until you have a single sorted array. The merging phase is not in-place, it uses temporary arrays.

public class mergeSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        sort(intArray, 0, intArray.length);

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

    public static void sort(int[] input, int start, int end) {

        if (end - start < 2) {

            return;

        }

        int mid = (start + end) / 2;

        sort(input, start, mid);

        sort(input, mid, end);

        merge(input, start, mid, end);

    }

    public static void merge(int[] input, int start, int mid, int end) {

        if (input[mid - 1] <= input[mid]) {

            return;

        }

        int i = start;

        int j = mid;

        int tempIndex = 0;

        int[] temp = new int[end - start];

        while (i < mid && j < end) {

            temp[tempIndex++] = input[i] <= input[j] ? input[i++] : input[j++];

        }

        System.arraycopy(input, i, input, start + tempIndex, mid - i);

        System.arraycopy(temp, 0, input, start, tempIndex);

    }

}

# Quick Sort

It’s an in-place and unstable algorithm and it has a time complexity of **O (nlog2n)**.Same as Merge Sort it’s also a Divide and Conquer algorithm, it uses a pivot element to partition the array into two parts, smaller elements will go to the left and right elements will go right, pivot ends up being in sorted position, process is now repeated for left and right array and eventually every element has been the pivot.

public class quickSort {

    public static void main(String[] args) {

        int[] intArray = { 20, 35, -15, 7, 55, 1, -22 };

        sort(intArray, 0, intArray.length);

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

    public static void sort(int[] input, int start, int end) {

        if (end - start < 2) {

            return;

        }

        int pivotIndex = partition(input, start, end);

        sort(input, start, pivotIndex);

        sort(input, pivotIndex + 1, end);

    }

    public static int partition(int[] input, int start, int end) {

        int pivot = input[start];

        int i = start;

        int j = end;

        while (i < j) {

            while (i < j && input[--j] >= pivot);

            if ( i < j) {

                input[i] = input[j];

            }

            while (i < j && input[++i] <= pivot);

            if (i < j) {

                input[j] = input[i];

            }

        }

        input[j] = pivot;

        return j;

    }

}

# Counting Sort

It’s a not in-place algorithm and can be a stable algorithm with a few extra steps, the time complexity is **O (n)** since assumptions are being made, a linear time complexity can be achieved. An algorithm that makes assumptions about the data, doesn’t use comparison, only works with non-negative discrete values and counts the occurrence of each value which need to be within a specific range.

public class countingSort {

    public static void main(String[] args) {

        int[] intArray = { 2, 5, 9, 8, 2, 8, 7, 10, 4, 3 };

        sort(intArray, 1, 10);

        for (int i = 0; i < intArray.length; i++) {

            System.out.println(intArray[i]);

        }

    }

    public static void sort(int[] input, int min, int max) {

        int[] countArray = new int[(max - min) + 1];

        for (int i = 0; i < input.length; i++) {

            countArray[input[i] - min]++;

        }

        int j = 0;

        for (int i = min; i <= max; i++) {

            while (countArray[i - min] > 0) {

                input[j++] = i;

                countArray[i - min]--;

            }

        }

    }

}

# Radix Sort

It’s an in-place and stable algorithm, it has a time complexity of **O (n)**, but often runs slower because of the overheac. It’s an algorithm that makes assumptions about the data, and it has to have the same width and radix, starts at the rightmost position and must use a stable sort algorithm at each stage.

public class radixSort {

    public static void main(String[] args) {

        int[] radixArray = { 4725, 4586, 1330, 8792, 1594, 5729 };

        sort(radixArray, 10, 4);

        for (int i = 0; i < radixArray.length; i++) {

            System.out.println(radixArray[i]);

        }

    }

    public static void sort(int[] input, int radix, int width) {

        for (int i = 0; i < width; i++) {

            radixSingleSort(input, i, radix);

        }

    }

    public static void radixSingleSort(int[] input, int position, int radix) {

        int numItems = input.length;

        int[] countArray = new int[radix];

        for (int value : input) {

            countArray[getDigit(position, value, radix)]++;

        }

        for (int j = 1; j < radix; j++) {

            countArray[j] += countArray[j - 1];

        }

        int[] temp = new int[numItems];

        for (int tempIndex = numItems - 1; tempIndex >= 0; tempIndex--) {

            temp[--countArray[getDigit(position, input[tempIndex], radix)]] = input[tempIndex];

        }

        for (int tempIndex = 0; tempIndex < numItems; tempIndex++) {

            input[tempIndex] = temp[tempIndex];

        }

    }

    public static int getDigit(int position, int value, int radix) {

        return value / (int) Math.pow(radix, position) % radix;

    }

}

# Lists

Definition

It’s an ordered collection of elements that allows duplicates and permits elements to be inserted or accessed by their position (index) in the Iist.

ArrayList

Dynamic array implementation of the List Interface. Provides fast retrieval but slower insertion and deletion in comparison to LinkedList

ArrayList<String> arrayList = new ArrayList<>();

LinkedList

Each item in the list is called a node, the first item in the list is the head of the list.

If you have a singly (one link between nodes) linked list you should insert and delete items to the front of the list to maintain **O (1)**, otherwise you have to traverse the array

If you have a doubly linked list, the list has a head and tail and for them the time complexity is **O (1)** but for inbetween nodes time is **O (n)**. Allows for fast insertion and deletion at the cost of slower random access.

LinkedList<Integer> linkedList = new LinkedList<>();

Vector

Similar to ArrayList but synchronized, making it thread-safe, often considered legacy class

Vector<Double> vector = new Vector<>();

# Abstract Data Type

Definition

It doesn’t dictate how the data is organized but it does dictate the operations you can perform, abstract data types are usually an interface.

# Stacks

They are an abstract data type, LIFO – last in, first out so there is no random access. The item at the top of the stack is the only one we’re allowed to access. Operations are *push/pop/peek.* The ideal data structure to back a stack is linked list. The time complexity will be **O (1)** when using a linked list and **O (n)** for an array (should be used when either memory is tight or if you know the maximum number of items that will ever be on the stack).

Stack<Integer> stack = new Stack<>();

# Queues

They are an abstract data type, FIFO – first in, first out, item is added at the end of the queue, remove or peek the item at the front of the queue. Time complexity depends on what is the back of the interface.

A **Deque (Deck)** is a double ended queue, supports insertion and removal at both ends.

Queue<String> queue = new LinkedList<>();

# Hashtables

Also an abstract data type, provides access to data using keys and consists of pais key/value pair, it is optimized for retrieval (when you know the key). The **load factor** tells how full a hash table is, it’s used to decide when to resize the array backing the hash table, can’t be too low (empty space) and can’t be too full (increase likelihood of collisions).

Map<String, Employee> hashMap = new HashMap<String, Employee>();