- Big O Notation
  - Constant time ($O(1)$) complexity
  - Linear Time ($O(n)$) complexity
  - Quadratic Time ($O(n^2)$) complexity
  - Logarithmic Time ($O(\log n)$) complexity
- Validating Algorithms - Deterministic Algorithm - Randomized Algorithm - Exact Algorithm - Approximate Algorithm
- Data Structures - Stacks - Queues - Trees - Types of Trees.
- Sorting and Searching Algorithms. - Sorting Algorithms - Bubble Sort - Insertion Sort - Merge Sort

Approaches

Greedy

Divide and Conquer

Parallel

Spreading calculations over a number of processors to decrease total processing time

Approximation

Calculating an approximate solution with an easily determinable error

Generalization

Adapting a similar problem's solution to arrive at an adequate solution

The Mathematics of Algorithms

Size of a Problem Instance

An instance of a problem is the input data set given to an algorithm. The behaviour of an algorithm is described by the rate of growth of execution time as a function of the size of the input problem

instance, i.e. $f(x)$ where x represents the size of the problem instance and therefore $f(x)$ being the execution time at a problem instance size of x, as result $f'(x)$ representing the behaviour of an algorithm.

An algorithm's rate of growth determines how it will perform on increasingly large problem instances

Big O Notation

Used to quantify the performance of various algorithms as input size grows.

Constant time ($O(1)$) complexity

An algorithm that takes the same amount of time to run independent of the size of the input data

```
def getFirst(my_list):
    return my_list[0]
```

The function `getFirst` will always take the same amount of time to retrieve the first element in a passed list

Linear Time ($O(n)$) complexity

An algorithm whose execution time is directly proportional to the size of the input

```
def getSum(my_list):
    sum = 0
    for item in list:
        sum += item
    return sum
```

Notice that in the main loop of the function `getSum` the number of iterations it performs increases linearly with an increasing value of n resulting in $O(n)$ complexity

Quadratic Time ($O(n^2)$) complexity

An algorithm whose execution time is proportional to the square of the input size

```
def getSum(my_list):
    sum = 0
    for row in my_list:
         for item in row:
```

```
            sum += 0
    return sum
```

Note that the nested inner loop within the other main loop gives this code a complexity of $O(n^2)$

Logarithmic Time ($O(\log n)$) complexity

An algorithm whose execution time is proportional to the logarithm of the input size, i.e. with each iteration the input size decreases by a constant multiple factor.

```
def searchBinary(my_list, item):
    first = 0
    last = len(my_list) - 1
    foundFlag = False
    while first <= last and not foundFlag:
        mid = (first + last) // 2
        if my_list[mid] == item:
            foundFlag = True
        else:
            if item < my_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return foundFlag
```

Validating Algorithms

Deterministic Algorithm

A particular input always generates the exact same output

Randomized Algorithm

A random sequence of numbers is taken along with an input which alters the output every time the algorithm is run.

Exact Algorithm

Algorithms expected to produce a precise solution without introducing any assumptions or approximations

Approximate Algorithm

Algorithms that deal with a complex problem by making assumptions.

Data Structures

Stacks A linear data structure to store a one dimensional list. I can store items in either Last-in, First-out (LIFO) or First-In, Last-Out (FILO).

```
class Stack:
    def __init__(self) -> None:
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)
```

Queues Stores n elements in a single dimensional structure. Elements are added and removed in First-in, First-out FIFO format. A queue has a rear and a front.

- Dequeue: Item is removed from the front of the queue.
- Enqueue: Item is added to the rear of the queue

```
class Queue(object):
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Trees A data structure used to represent hierarchical relationships among data elements that need to be stored or processed.

- Node: An element in the tree.
- Root: Starting data element of a tree/ A node with no parent.
- Level of a node: Distance between a node and the root.
- Sibling nodes: Two nodes at the same level in the tree.
- Child node: A node is said to be a child node in relation to another if it directly linked to that node and it's node level is less than the proposed node.
- Parent node: A node is said to be a parent node in relation to another if it directly linked to that node and it's node level is greater than the proposed node.
- Degree of a node: The number of children a node has.
- Branches: Links between nodes in the tree.
- Degree of a tree: The greatest number of children a node in the tree has.
- Subtree: A section of a tree with a chosen node acting as a root node and all the preceding children of this node acting as the nodes of the new tree.
- Leaf: A node with no children.
- Internal node: Any node that is neither a root node or a leaf node, i.e. having at least one parent and one child.

Types of Trees.

- Binary Trees: A tree with a degree of two.
- Full tree: A tree with all nodes having either two or no children.
- Perfect tree: A tree with all leaf nodes being of the same level.
- Ordered tree: A tree in which the children of a node are organized in order according to some criteria, for example, left to right in ascending order in which nodes at the same level will increase in value while moving from left to right.

Sorting and Searching Algorithms.

Sorting Algorithms

Bubble Sort

```
START
INPUT Dataset
Last_Index := Length of Dataset - 1
FOR Pass_Number = Last_Index to 0
    FOR Index = 0 to Pass_Number
        IF Dataset[Index] > Dataset[Index + 1] THEN
```

```
        Swap the values of indexes Index and Index + 1 in the Dataset
    ENDFOR
ENDFOR
RETURN Dataset
```

```
def BubbleSort(list):
    last_index = len(list) - 1
    for passNo in range(last_index, 0, -1):
        for i in range(passNo):
            if list[i] > list[i + 1]:
                list[i], list[i + 1] = list[i + 1], list[i]
    return list
```

- $O(N^2)$

Based on iterations/passes, the number of which is determined by $N - 1$, where N is the size of the dataset. The goal of each pass is pushing the highest value to the top/end of the list, by comparing adjacent neighbour values. If the value in the current position is lower than the value in the index lower than this position the values are swapped.

Insertion Sort

```
START
INPUT Dataset
FOR Index = 1 to Length of Dataset
    Prev_Index := Index - 1
    Next_Element := Dataset[Index]
    WHILE Dataset[Prev_Index] > Next_Element and_Prev_Index >= 0
        Dataset[Prev_Index + 1] := Dataset[Prev_Index]
        DECREMENT Prev_Index
    ENDWHILE
    Dataset[Prev_Index + 1] = Next_Element
ENDFOR
RETURN Dataset
```

```
def InsertionSort(list):
    for i in range(1, len(list)):
        j = i - 1
        element_next = list[i]
        while (list[j] > element_next) and (j >= 0):
            list[j + 1] = list[j]
            j -= 1
        list[j + 1] = element_next
    return list
```

- $O(N)$ when dataset is already sorted
- $O(N^2)$ when dataset is unsorted

Each iteration we remove a data point from the working dataset and insert it into its right position.

In the first iteration we take two data points and sort then, in the following iteration we select a third data point and determine it's correct position. This continues until the dataset is sorted.

Merge Sort

```
Function Merge(Dataset, Start, MidPoint, End)
    A := 0
    B := 0
    C := Start

    Left := Items of the Dataset from index Start to MidPoint
    Right := Items of the Dataset from the index MidPoint + 1 to Start
    WHILE A < Length of Left and B < Length of Right
        IF Left[A] < Right[B] THEN
            Dataset[C] = Left[A]
            INCREMENT A
        ELSE
            Dataset[C] = Right[B]
            INCREMENT B
        ENDIF
        INCREMENT C
    ENDWHILE

    WHILE A < Length of Left
        Dataset[C] = Left[A]
        INCREMENT A
        INCREMENT C
    ENDWHILE

    WHILE B < Length of Right
        Dataset[C] = Right[B]
        INCREMENT B
        INCREMENT C
    ENDWHILE

Function MergeSort(Dataset, Start, End)
    IF Start < End THEN
        MidPoint := (End - Start) / 2 + Start
        MergeSort(Dataset, Start, MidPoint) // Left
        MergeSort(Dataset, MidPoint + 1, End) // Right
        Merge(Dataset, Start, MidPoint, End)
    ENDIF
    RETURN Dataset
```

```
def MergeSort(list):
    if len(list) > 1:
        mid = len(list) // 2
        left = list[:mid]
        right = list[mid:]

        MergeSort(left)
```

```
        MergeSort(right)

        a = 0
        b = 0
        c = 0

        while a < len(left) and b < len(right):
            if left[a] < right[b]:
                list[c] = left[a]
                a += 1
            else:
                list[c] = right[b]
                b += 1
            c += 1

        while a < len(left):
            list[c] = left[a]
            a += 1
            c += 1

        while b < len(right):
            list[c] = right[b]
            b += 1
            c += 1
    return list
```

- $O(n \log n)$

Based on a divide and conquer strategy. In the first phase (splitting) the algorithm continually divides the dataset into two parts recursively until the size of t dataset is equal to a predefined threshold. In the second phase (merging) the algorithm then merges the parts until a result is arrived at.

Shell Sort

```
START
INPUT Dataset
Distance := half the length of the dataset (floor division)
WHILE Distance > 0
    FOR Index := Distance to Full Length of the Dataset
        Temp := Dataset[Index]
        Secondary_Index := Index
        WHILE Secondary_Index >= Distance and Dataset[Secondary_Index -
            Distance] > Temp
            Dataset[Secondary_Index] = Dataset[Secondary_Index -
               Distance]
            Secondary_Index = Secondary_Index - Distance
        ENDWHILE
        Dataset[Secondary_Index] = Temp
    ENDFOR
    Distance = half the current value of Distance (floor division)
ENDWHILE
```

```
RETURN Dataset
```

```
def ShellSort(list):
    distance = len(list) // 2
    while distance > 0:
        for i in range(distance, len(list)):
            temp = list[i]
            j = i
            while j >= distance and list[j - distance] > temp:
                list[j] = list[j - distance]
                j -= distance
        list[j] = temp
        distance //= 2
    return list
```

- Reasonably good performance for datasets with under 6,000 elements
- $O(N)$ when dataset is partially sorted

Similar to the bubble sort but instead of comparing immediate neighbours each pass, elements are selected and compared according to a fixed gap, i.e. a pair of elements in the initial pass (sub-list). In the subsequent passes the number of items in each sub-list increase by the factor of number items in the previous sub-list. This continues until the number of sub-lists is equal to 1. At this point we assume the list to be sorted.