- 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

```python
1  def getFirst(my_list):
2      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

```python
1  def getSum(my_list):
2      sum = 0
3      for item in list:
4          sum += item
5      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

```python
1  def getSum(my_list):
2      sum = 0
3      for row in my_list:
4          for item in row:
```

```
5                sum += 0
6        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.

```
1  def searchBinary(my_list, item):
2      first = 0
3      last = len(my_list) - 1
4      foundFlag = False
5      while first <= last and not foundFlag:
6          mid = (first + last) // 2
7          if my_list[mid] == item:
8              foundFlag = True
9          else:
10             if item < my_list[mid]:
11                 last = mid - 1
12             else:
13                 first = mid + 1
14     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).

```python
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

```python
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

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

```
 7          Swap the values of indexes Index and Index + 1 in the Dataset
 8      ENDFOR
 9  ENDFOR
10  RETURN Dataset
```

```python
1  def BubbleSort(list):
2      last_index = len(list) - 1
3      for passNo in range(last_index, 0, -1):
4          for i in range(passNo):
5              if list[i] > list[i + 1]:
6                  list[i], list[i + 1] = list[i + 1], list[i]
7      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

```
 1  START
 2  INPUT Dataset
 3  FOR Index = 1 to Length of Dataset
 4      Prev_Index := Index - 1
 5      Next_Element := Dataset[Index]
 6      WHILE Dataset[Prev_Index] > Next_Element and_Prev_Index >= 0
 7          Dataset[Prev_Index + 1] := Dataset[Prev_Index]
 8          DECREMENT Prev_Index
 9      ENDWHILE
10      Dataset[Prev_Index + 1] = Next_Element
11  ENDFOR
12  RETURN Dataset
```

```python
1  def InsertionSort(list):
2      for i in range(1, len(list)):
3          j = i - 1
4          element_next = list[i]
5          while (list[j] > element_next) and (j >= 0):
6              list[j + 1] = list[j]
7              j -= 1
8          list[j + 1] = element_next
9      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

```
1   Function Merge(Dataset, Start, MidPoint, End)
2       A := 0
3       B := 0
4       C := Start
5
6       Left := Items of the Dataset from index Start to MidPoint
7       Right := Items of the Dataset from the index MidPoint + 1 to Start
8       WHILE A < Length of Left and B < Length of Right
9           IF Left[A] < Right[B] THEN
10              Dataset[C] = Left[A]
11              INCREMENT A
12          ELSE
13              Dataset[C] = Right[B]
14              INCREMENT B
15          ENDIF
16          INCREMENT C
17      ENDWHILE
18
19      WHILE A < Length of Left
20          Dataset[C] = Left[A]
21          INCREMENT A
22          INCREMENT C
23      ENDWHILE
24
25      WHILE B < Length of Right
26          Dataset[C] = Right[B]
27          INCREMENT B
28          INCREMENT C
29      ENDWHILE
30
31  Function MergeSort(Dataset, Start, End)
32      IF Start < End THEN
33          MidPoint := (End - Start) / 2 + Start
34          MergeSort(Dataset, Start, MidPoint) // Left
35          MergeSort(Dataset, MidPoint + 1, End) // Right
36          Merge(Dataset, Start, MidPoint, End)
37      ENDIF
38      RETURN Dataset
```

```
1   def MergeSort(list):
2       if len(list) > 1:
3           mid = len(list) // 2
4           left = list[:mid]
5           right = list[mid:]
6
7           MergeSort(left)
```

```
 8          MergeSort(right)
 9
10          a = 0
11          b = 0
12          c = 0
13
14          while a < len(left) and b < len(right):
15              if left[a] < right[b]:
16                  list[c] = left[a]
17                  a += 1
18              else:
19                  list[c] = right[b]
20                  b += 1
21              c += 1
22
23          while a < len(left):
24              list[c] = left[a]
25              a += 1
26              c += 1
27
28          while b < len(right):
29              list[c] = right[b]
30              b += 1
31              c += 1
32      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

```
 1  START
 2  INPUT Dataset
 3  Distance := half the length of the dataset (floor division)
 4  WHILE Distance > 0
 5      FOR Index := Distance to Full Length of the Dataset
 6          Temp := Dataset[Index]
 7          Secondary_Index := Index
 8          WHILE Secondary_Index >= Distance and Dataset[Secondary_Index -
                  Distance] > Temp
 9              Dataset[Secondary_Index] = Dataset[Secondary_Index -
                      Distance]
10              Secondary_Index = Secondary_Index - Distance
11          ENDWHILE
12          Dataset[Secondary_Index] = Temp
13      ENDFOR
14      Distance = half the current value of Distance (floor division)
15  ENDWHILE
```

```
16  RETURN Dataset
```

```python
 1  def ShellSort(list):
 2      distance = len(list) // 2
 3      while distance > 0:
 4          for i in range(distance, len(list)):
 5              temp = list[i]
 6              j = i
 7              while j >= distance and list[j - distance] > temp:
 8                  list[j] = list[j - distance]
 9                  j -= distance
10          list[j] = temp
11          distance //= 2
12      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.

QuickSort

```
 1  Function QuickSort(Dataset, Low, High)
 2      IF Size of dataset is less than two THEN
 3          RETURN
 4      ENDIF
 5
 6      IF Low >= High THEN
 7          RETURN
 8      ENDIF
 9
10      Pivot := Last item in dataset
11      PivotIndex := Index of last item in datasets
12      FOR Index := Index of first item in list to last
13          IF Dataset[Index] <= Pivot THEN
14              INCREMENT PivotIndex
15              Swap the values of Dataset[PivotIndex] and Dataset[Index]
16          ENDIF
17      ENDFOR
18      INCREMENT PivotIndex
19      Swap the values of Dataset[PivotIndex] and Dataset[High]
20
21      QuickSort(Dataset, Low, DECREMENT PivotIndex)
22      QuickSort(Dataset, INCREMENT PivotIndex, High)
```

```
1  def QuickSort(list):
2      if len(dataset) < 2:
3          return dataset
4
5      pivot = dataset[0]
6      less = [item for item in dataset[1:] if item <= pivot]
7      greater = [item for item in dataset[1:] if item >= pivot]
8      return QuickSort(less) + [pivot] + QuickSort(greater)
```

- Speed depends on pivot choice but generally $O(n \log n)$ and worst case $O(n^2)$