**Algorithms**

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

* Algorithms characteristics

Graphical user interface, text, application

Description automatically generated  
Text

Description automatically generated

* Common Algorithms

- Search Algorithms

- Sorting Algorithms

- Computational Algorithms

- Collection Algorithms

Table

Description automatically generated

Graphical user interface, text

Description automatically generated

* **Recursion**

When a function calls itself

A screenshot of a computer

Description automatically generated with medium confidence

- function count down using recursion:

def countdown(x):

    if(x==0):

        return

    else:

        print(x," ")

        countdown(x-1)

countdown(4)

Output:

4  
3  
2  
1

* **Bubble sort**

Graphical user interface, text

Description automatically generated

Implement in python:

def bubble\_sort(list):

    for i in range(0,len(list)-1):

        for j in range(len(list)-1):

            if(list[j]>list[j+1]):

                temp = list[j]

                list[j] = list[j+1]

                list[j+1] = temp

    return list

* **Merge sort**

Graphical user interface, text

Description automatically generated with medium confidence

Implement in python with recursion:

def mergeSort(dataset):

    if len(dataset) > 1:

        mid = len(dataset) // 2

        leftarr = dataset[:mid]

        rightarr = dataset[mid:]

        # recursively break down the arrays

        mergeSort(leftarr)

        mergeSort(rightarr)

        i=0 # index into the left array

        j=0 # index into the right array

        k=0 # index into merged array

        # while both arrays have content

        while i < len(leftarr) and j < len(rightarr):

            if leftarr[i] < rightarr[j]:

                dataset[k] = leftarr[i]

                i += 1

            else:

                dataset[k] = rightarr[j]

                j += 1

            k += 1

        # if the left array still has values, add them

        while i < len(leftarr):

            dataset[k] = leftarr[i]

            i += 1

            k += 1

        # if the right array still has values, add them

        while j < len(rightarr):

            dataset[k] = rightarr[j]

            j += 1

            k += 1

* **Quick sort**

Graphical user interface, text, application

Description automatically generated

Implement in python:

def quickSort(dataset, first, last):

    if first < last:

        # calculate the split point

        pivotIdx = partition(dataset, first, last)

        # now sort the two partitions

        quickSort(dataset, first, pivotIdx-1)

        quickSort(dataset, pivotIdx+1, last)

def partition(datavalues, first, last):

    # choose the first item as the pivot value

    pivotvalue = datavalues[first]

    # establish the upper and lower indexes

    lower = first + 1

    upper = last

    # start searching for the crossing point

    done = False

    while not done:

        # advance the lower index

        while lower <= upper and datavalues[lower] <= pivotvalue:

            lower += 1

        # advance the upper index

        while datavalues[upper] >= pivotvalue and upper >= lower:

            upper -= 1

        # if the two indexes cross, we have found the split point

        if upper < lower:

            done = True

        else:

            # exchange the two values

            temp = datavalues[lower]

            datavalues[lower] = datavalues[upper]

            datavalues[upper] = temp

    # when the split point is found, exchange the pivot value

    temp = datavalues[first]

    datavalues[first] = datavalues[upper]

    datavalues[upper] = temp

    # return the split point index

    return upper

* Unordered list search (Linear search)

Complexity: O(n)

def find\_item(item, itemlist):

    for i in range(0, len(itemlist)):

        if item == itemlist[i]:

            return i

    return None

* Ordered list search (Binary search)

Complexity: O(log n)

def binSearch(item, list):

    listSize = len(list) -1

    low = 0

    upper = listSize

    ind = (low+upper) // 2

    while(low <= upper):

        ind = (low+upper) // 2

        if list[ind] < item :

            low = ind + 1

        elif list[ind] > item :

            upper = ind - 1

        else:

            return ind

    return -1

Unique filtering with hash table

items = ["apple", "pear", "orange", "banana", "apple",

         "orange", "apple", "pear", "banana", "orange",

         "apple", "kiwi", "pear", "apple", "orange"]

# create a hashtable to perform a filter

filter = dict()

for item in items:

    filter[item] = 0

result = set(filter.keys())

print(result)