Algorithms

1. Selection Sort

Sorting algorithm which selects the least minimum element in the given array and displaces it with the i^{th} element for every iteration.

Code snippet :

```
def selec_sort(arr):

    for i in range(0, len(arr)-1):
        min = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min]:
                min = j
        arr[min] , arr[i] = arr[i] , arr[min]
    return arr
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/Algorithms/selecSort.py

2. Bubble Sort

Sorting algorithm which selects every element and displaces it with every adjacent element based on the condition given (ascending/descending). Faster than Selection Sort.

Code snippet :

```
def bble_sort(arr):

    for i in range(0, len(arr)):
        for j in range(0, len(arr)-i-1):
            if arr[j] > arr[j+1]:
                arr[j] , arr [j+1] = arr[j+1] , arr[j]
    return arr
```

Full code at :

https://github.com/AmunRha/ChallengeSet1/blob/master/Algorithms/bbleSort.py

3. Merge Sort

One of the most efficient sorting algorithms which uses the Divide and Conquer strategy. It Divides the array into sub-arrays and merges them from the lowest sub-array (ascending/descending) which when subsequently done for every level upwards results in a sorted array.

Code snippet :

```
def merge_sort(arr):
   if len(arr) >= 2:
       # Dividing Part
       mid = int(len(arr)/2)
       a = arr[:mid]
       b = arr[mid:]
       a = merge_sort(a)
       b = merge_sort(b)
       # Merging Part
       i = j = k = 0
       while i<len(a) and j < len(b):
           if a[i] <= b[j]:
               arr[k] = a[i]
               i+=1
           else:
               arr[k] = b[j]
               j+=1
           k+=1
       while i < len(a):
           arr[k] = a[i]
           k+=1
           i+=1
       while j < len(b):
           arr[k] = b[j]
           k+=1
           j+=1
   return arr
```

4. Binary Search

Fastest search algorithm which also uses Divide and Conquer strategy. Checks if the element to search is in the left or right half of the given array and then checks again if it is in the left or right half the previously divided array, this subsequently continues till the element is either found or not found.

Code snippet :

```
def bin_srch(arr, srch_elem):

   start = 0
   end = len(arr)-1
   while start<=end :
       mid = int((start + end)/2)
       if srch_elem < arr[mid]:
           end = mid - 1
       elif srch_elem >  arr[mid]:
           start = mid + 1
       elif srch_elem == arr[mid]:
           return True, mid
   return False, -1
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/Algorithms/binSearch.py

5. Balanced Parentheses using Stack

For every inputted parentheses, the opening braces are pushed onto the stack and when a closing braces is checked the stack is popped out. If the stack is empty then the parentheses are balanced and if the stack pop creates an exception or is not empty then the parentheses are not balanced.

Code snippet :

```
def check_balanced(inp):
   stack_1 = stack_2 = stack_3 = []
   remains = False
```

```
    for char in inp:
        # Only work if the entered characters are any sort of
paranthsis
        if char=='(' or char==')' or char=='{' or char=='}' or
char=='[' or char==']':
            if char=='(':
                stack_1.append(char)
            elif char=='{':
                stack_2.append(char)
            elif char=='[':
                stack_3.append(char)
            # Catch the exception in case there is an extra closing
bracket
            try:
                if char==')':
                    stack_1.pop()
                elif char=='}':
                    stack_2.pop()
                elif char==']':
                    stack_3.pop()
            except:
                remains = True
                break
    if len(stack_1) == 0 and len(stack_2) == 0 and len(stack_3) == 0
and remains == False:
        return True
    else:
        return False
```

Full code at : https://github.com/AmunRha/ChallengeSet1/blob/master/Algorithms/balParanStack.py

6. Loop in Linked List

Using Floyd's cycle finding algorithm we can check if the given Linked List contains a loop. It works on the principle that if a slow and a fast variable, both incrementing at different values would meet each other in case there is a loop.

Code snippet :

```
def check_loop(self):
        slow = fast = self.root
        while slow and fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/Algorithms/cycleLnkList.py