Sorting Algorithms

Chapter No.	23
Status	Completed

▼ Selection Sort

- Selection sort is a sorting algorithm that divides an array into a sorted part and an unsorted part, searches for the smallest element in the unsorted part and swaps this element with the first element in the unsorted part

▼ Process

1. Iterate through the array from index 0 to the end and find the smallest element.
2. Swap that smallest element with the element with index 0.
3. Iterate through the array from index 1 to the end and find the smallest element.
4. Swap that smallest element with the element with index 1.
5. Repeat this process a total of (n-1) times, where n is the length of the array, decreasing the number of elements that are iterated through by 1 each time the process is repeated

- O(n^2)
- In-place

```
def selectionsort(lis):
    n = len(lis)
    to_swap = 0

    for i in range(n-1):
        smallest = to_swap

        for j in range(to_swap, n):
            if lis[j] < lis[smallest]:
                smallest = j

        lis[to_swap], lis[smallest] = lis[smallest], lis[to_swap]
        to_swap += 1
```

▼ Bubble Sort

- Bubble sort is a sorting algorithm that performs pairwise comparison between adjacent elements in an array and swaps the two elements if they are not in the correct order
- This process continues until the array is sorted and no more swaps have to be made
- The smallest or largest element is 'bubbled' towards the end of the array

▼ Process

1. Compare the zeroth and first elements and swap them if the zeroth element is larger than the first element.
2. Compare the first and second elements and swap them if the first element is larger than the second element.
3. Continue to perform pairwise comparison until the last two elements in the array have been compared. The largest element is now the last element in the array and this constituted as one pass of the array from the zeroth to the last element.
4. Perform a second pass of the array from the zeroth to the second last element.
5. Perform a total of (n-1) passes, where n is the length of the array, reducing the number of comparisons made in a pass by 1 for each consecutive pass, until no comparisons are made in a pass

- O(n^2)
- In-place

```
def bubblesort(lis):
    n = len(lis)
    end = n-1

    for i in range(n-1):
        for j in range(end):
            if lis[j] > lis[j+1]:
                lis[j], lis[j+1] = lis[j+1], lis[j]
        end -= 1
```

▼ Insertion Sort

- Insertion sort is a sorting algorithm that divides an array into a sorted part and unsorted part and inserts elements from the unsorted part into their correct index in the sorted part

▼ Process

1. Divide the array into a sorted part and unsorted part.
2. At the beginning, the sorted part consists of only the first element and the unsorted part consists of the second element to the last element
3. Add the first element in the unsorted part into the sorted part and repeatedly swap it with the element before it until the sorted part of the array is sorted.
4. Repeat this process of inserting elements from the unsorted part into the sorted part a total of (n-1) times, where n is the length of the array, until there are no elements in the unsorted part

- O(n^2)
- In-place

```
def insertionsort(lis):
    n = len(lis)
    unsorted = 1

    for i in range(0, n-1):
        to_insert = unsorted
        to_compare = to_insert-1

        while lis[to_insert] < lis[to_compare] and to_compare >= 0:
            lis[to_insert], lis[to_compare] = lis[to_compare], lis[to_insert]
            to_insert -= 1
            to_compare -= 1

        unsorted += 1
```

▼ Merge Sort

- Merge sort is a recursive sorting algorithm that operates on the divide-and-conquer approach

▼ Process

1. The array is 'divided' by splitting the array into two smaller arrays of equal or near-equal length
2. These smaller arrays are recursively 'conquered' by splitting them into even smaller arrays, until each array has a length of 1 and does not need to be sorted
3. The sorted arrays, each with a length of 1, are then recursively merged into longer sorted arrays, until all the sorted arrays have been merged into a single sorted array

- O(n log n)
- Not-in-place

```
def combine(lis1, lis2): #combines two sorted lists into one sorted list
    for i in lis1:
        if i > lis2[-1]:
            lis2.append(i)
        else:
            insert_index = 0
```

```
            for j in range(len(lis2)):
                if i > lis2[j]:
                    insert_index += 1
                    continue

            lis2.insert(insert_index, i)
    return lis2

def mergesort(lis):
    if len(lis) <= 1:
        return lis

    divide_index = len(lis)//2

    left_lis = lis[0:divide_index]
    right_lis = lis[divide_index:]

    sorted_left_lis = mergesort(left_lis)
    sorted_right_lis = mergesort(right_lis)
    return combine(sorted_left_lis, sorted_right_lis)
```

▼ Quicksort

- Quicksort is a recursive sorting algorithm that operates on the divide-and-conquer approach

▼ Process

1. The array is 'divided' into two smaller arrays by choosing an element to be the pivot and rearranging the array so that all elements smaller than the pivot are on the left of the pivot and all elements larger than the pivot are on the right of the pivot
2. These smaller arrays are recursively 'conquered' by splitting them into even smaller arrays and rearranging the smaller arrays, until each array has a length of 1 and does not need to be rearranged
3. Since the sort is performed in-place, the smaller arrays are recursively combined

- O(n^2)
- In-place

```
def partition(lis, L, R): #sorts the list into [elements smaller than pivot, pivot, elements larger than pivot] AND returns the ir
    i = L
    j = L

    while j < R:
        if lis[j] > lis[R]:
            j += 1
        else:
            lis[i], lis[j] = lis[j], lis[i]
            i += 1
            j += 1

    lis[i], lis[j] = lis[j], lis[i]

    return i


def quicksort(lis, L, R): #L = index of first element, R = index of last element
    if len(lis) <= 1: #base case for original list
        return lis

    if R < L: #base case for recursion
        return lis

    pivot_index = partition(lis, L, R)

    quicksort(lis, L, pivot_index-1)
    quicksort(lis, pivot_index+1, R)
```