AIM :
Implement Bubble sort,Merge sort, Insertion Sort and greedy search algorithm

---

Introduction

Sorting

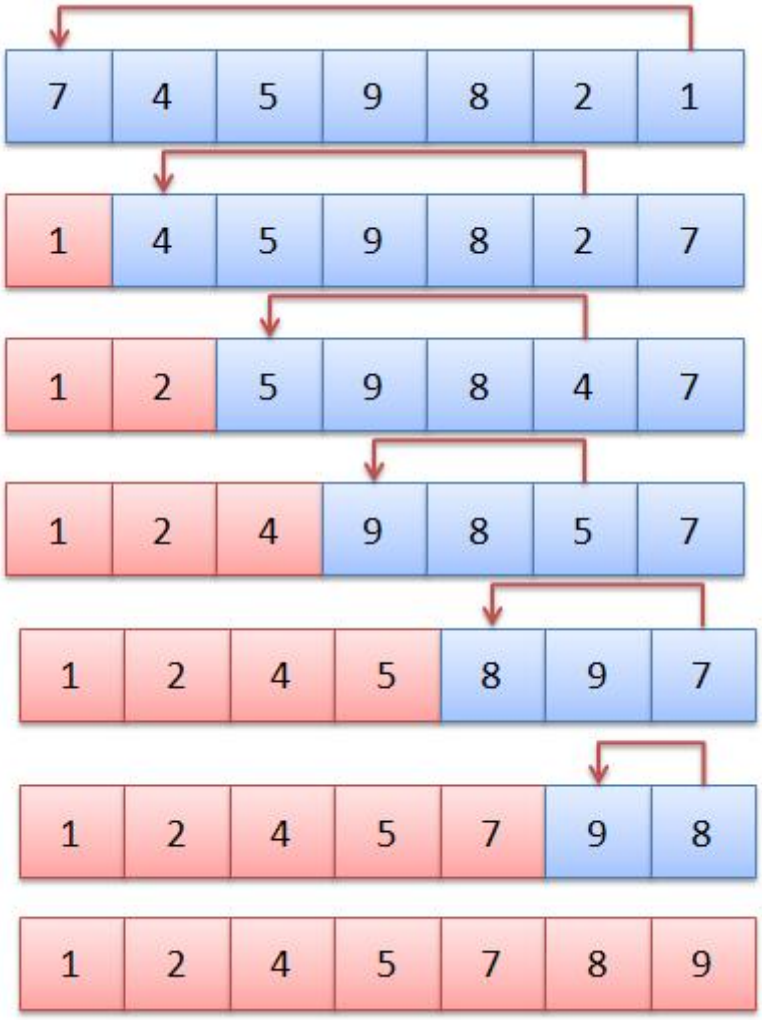
Sorting is any process of arranging items systematically. in computer science, sorting algorithms put elements of list in a certain order

Selection Sort algorithm

This algorithm work by repeatedly finding the minimum element in the list and placing it at the beginning. this way maintains two lists:

1. the sublist of already-sorted elements which is filled from left to right
2. the sublist of the remaining unsorted elements that need to be sorted

In other words , this algorithms works by iterating over list and swapping each element with minimum(or maximum)element found in the unsorted list with that in the sorted list



In [1]:

```
def selection_sort(lst):
    """
    Selection sort function
    :param lst: List of integers
    """

    # Traverse through all lst elements
    for i in range(len(lst)):
        # Find the minimum element in unsorted lst
        min_index = i
        for j in range(i + 1, len(lst)):
            if lst[min_index] > lst[j]:
                min_index = j

        # Swap the found minimum element with the first element
        lst[i], lst[min_index] = lst[min_index], lst[i]


# Driver code to test above
if __name__ == '__main__':

    lst = [3, 2, 1, 5, 4]
    selection_sort(lst)  # Calling selection sort function

    # Printing Sorted lst
    print("Sorted lst: ", lst)
```

```
Sorted lst:  [1, 2, 3, 4, 5]
```

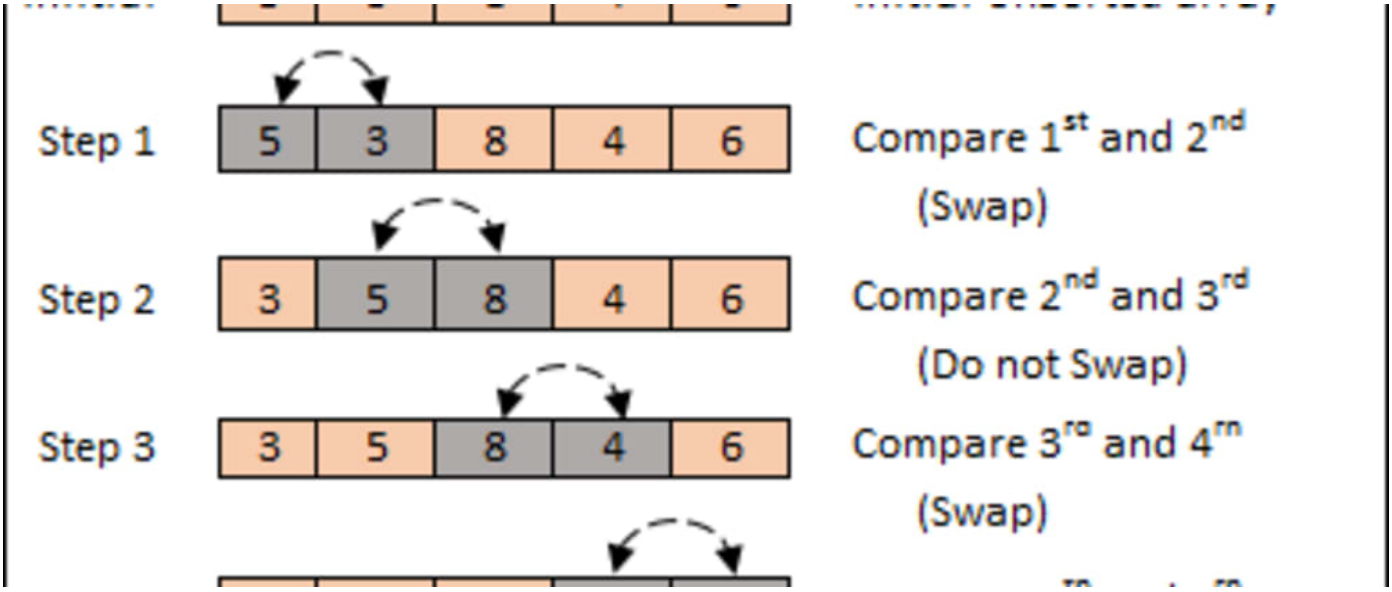
Time Complexity

The time complexity of this code is in O(n2) because finding a minimum number in the list requires iterating over the entire list for each element of the given list. the quadratic time complexity makes it impractical for large inputs

Bubble Sort algorithm

This is another famous sorting algorithm also known as sinking soet it works by comparin adjacent paris of elements and swapping tham if them if they are in the wrong order. This is repeated until the list is sorted

Think of it this way: a bubble passed over the list catches the maximum/minimum element and brings it over to the right side



In [2]:

```
def bubble_sort(lst):
    """
    Bubble sort function
    :param lst: lst of unsorted integers
    """

    # Traverse through all list elements
    for i in range(len(lst)):

        # Last i elements are already in place
        for j in range(0, len(lst) - i - 1):

            # Traverse the list from 0 to size of lst - i - 1
            # Swap if the element found is greater than the next element
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]


# Driver code to test above
if __name__ == '__main__':

    lst = [3, 2, 1, 5, 4]
    bubble_sort(lst)  # Calling bubble sort function

    print("Sorted list is: ", lst)
```

```
Sorted list is:  [1, 2, 3, 4, 5]
```

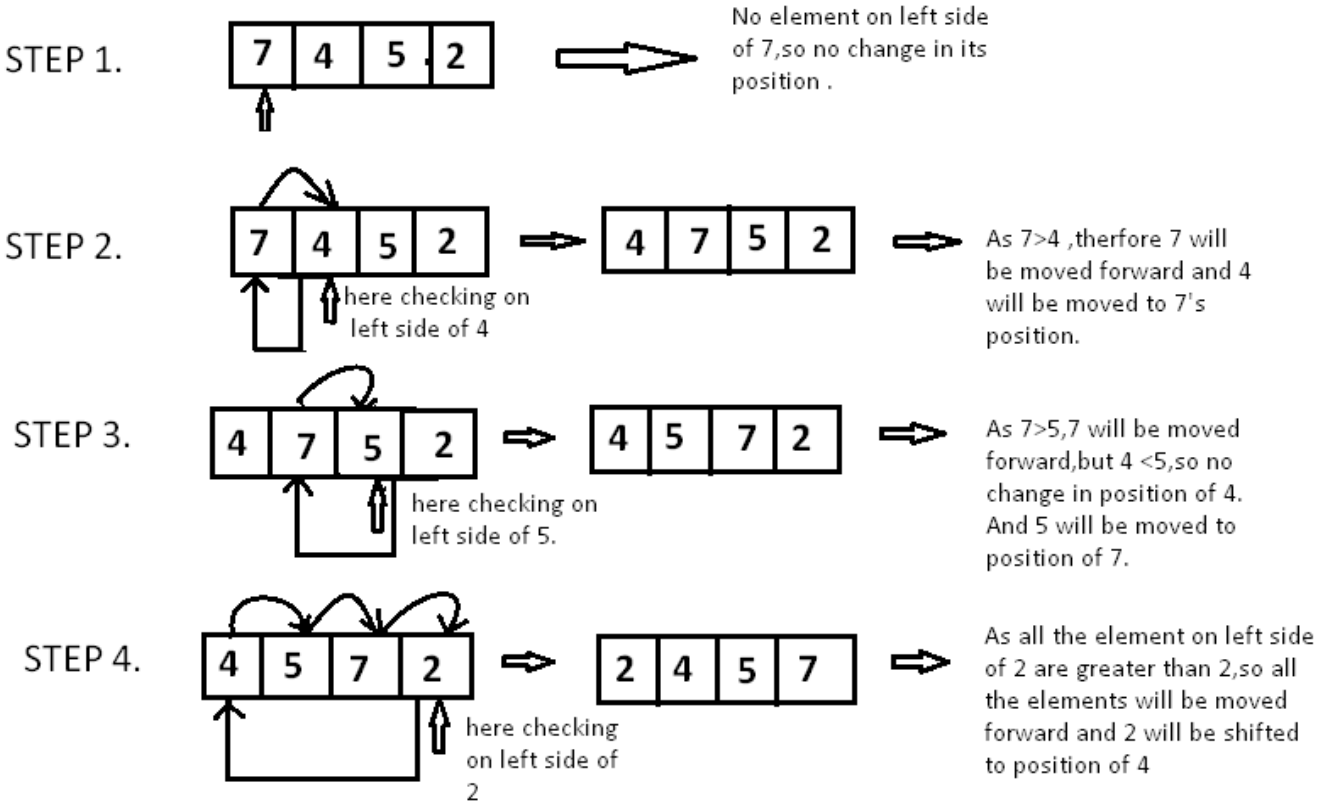
Time Complexity

The time complexity of this code is O(n2) again this algorithm is not very effcient.

insertion Sort algorithm

Insertion sort is another famous sorting algorithms and works the way you would naturally sort in real life

It iterates over the given list,figures out whaht the correct position of every element is, and inserts it there



In [4]:

```
def insertion_sort(lst):
    """
    Insertion sort function
    :param lst: lst of unsorted integers
    """

    # Traverse through 1 to len(lst)
    for i in range(1, len(lst)):

        key = lst[i]

        # Move elements of lst greater than key, to one position ahead
        j = i - 1
        while j >= 0 and key < lst[j]:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = key


# Driver code to test above
if __name__ == '__main__':

    lst = [3, 2, 1, 5, 4]
    insertion_sort(lst)  # Calling insertion sort function

    print("Sorted list is: ", lst)
```

```
Sorted list is:  [1, 2, 3, 4, 5]
```

Time complexity

The algorithm is O(n^2) O(n 2) , which, again, makes it a poor choice for large input sizes. However, notice that the complexity is actually n^2 n 2

only when the input list is sorted in reverse. So, the ‘best-case’ complexity (when the list is sorted in the correct order) is \Omega(n) Ω(n) .

Merge Sort algorithm

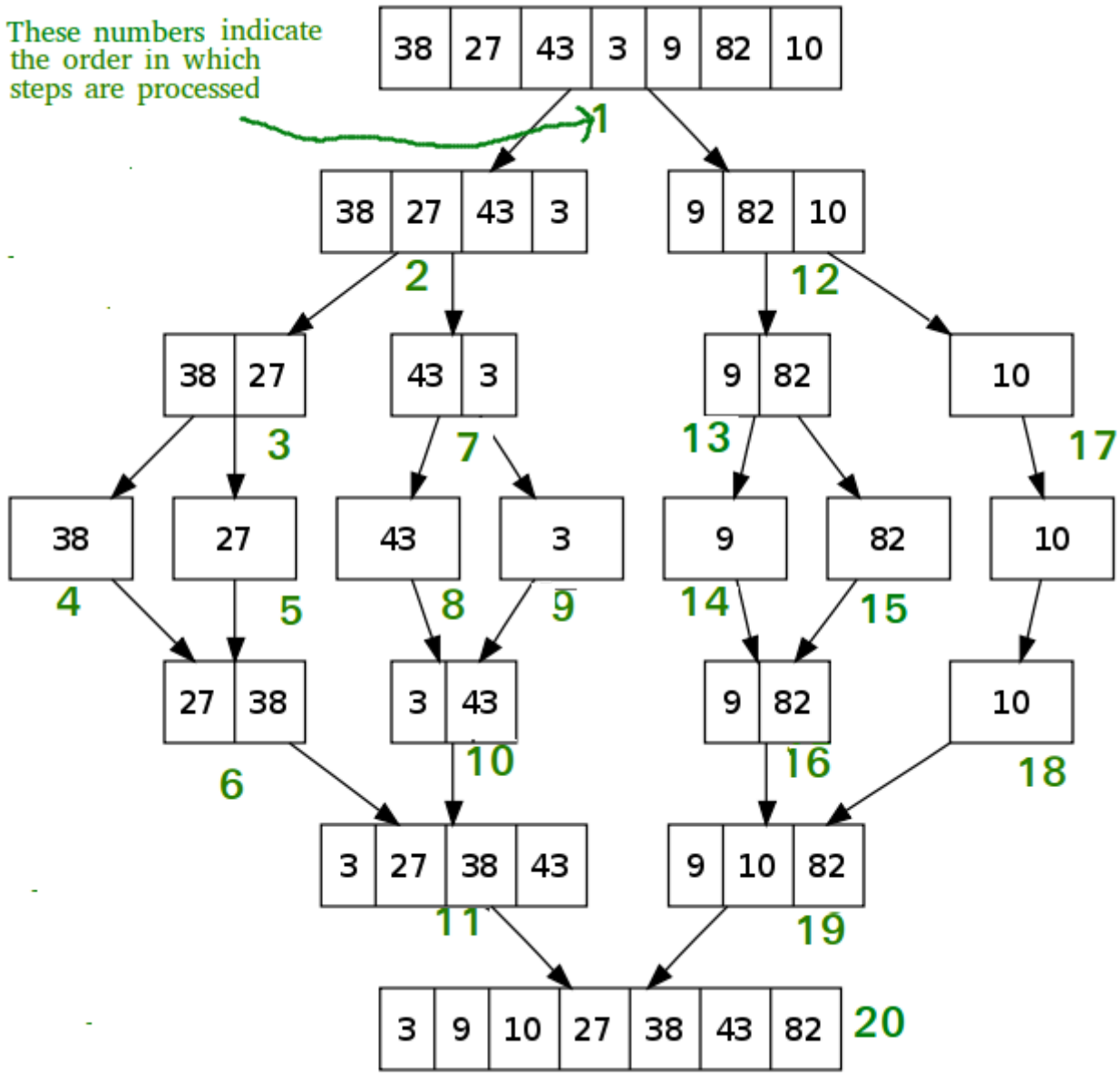
Merge sort is one of the most promient divide-and-conquer sorting algorithms in the modern era.it can be used to sort the values in any traversable data structure such as list

the theory

Merge sort works by splitting the input list two halves, repeating the process on those halves, and finally merging the two sorted halves toghether

The algorithm first moves from top to bottom , dividing the list into smaller and smaller parts until only the separate elements remain

Implementation



In [5]:

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

        # Recursive call on each half
        mergeSort(left)
        mergeSort(right)

        # Two iterators for traversing the two halves
        i = 0
        j = 0

        # Iterator for the main list
        k = 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
              # The value from the left half has been used
              myList[k] = left[i]
              # Move the iterator forward
              i += 1
            else:
                myList[k] = right[j]
                j += 1
            # Move to the next slot
            k += 1

        # For all the remaining values
        while i < len(left):
            myList[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            myList[k]=right[j]
            j += 1
            k += 1

myList = [54,26,93,17,77,31,44,55,20]
mergeSort(myList)
print(myList)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

This is the recursive approach for implementing merge sort. The steps needed to obtain the sorted array through this method can be found below:

1. The list is divided into **left** and **right** in each recursive call until two adjacent elements are obtained.
2. Now begins the sorting process. The **i** and **j** iterators traverse the two halves in each call. The **k** iterator traverses the whole lists and makes changes along the way.
3. If the value at **i** is smaller than the value at **j**, **left[i]** is assigned to the **myList[k]** slot and **i** is incremented. If not, then **right[j]** is chosen.
4. This way, the values being assigned through **k** are all sorted.

5. At the end of this loop, one of the halves may not have been traversed completely. Its values are simply assigned to the remaining slots in the list.