**Sorting:**

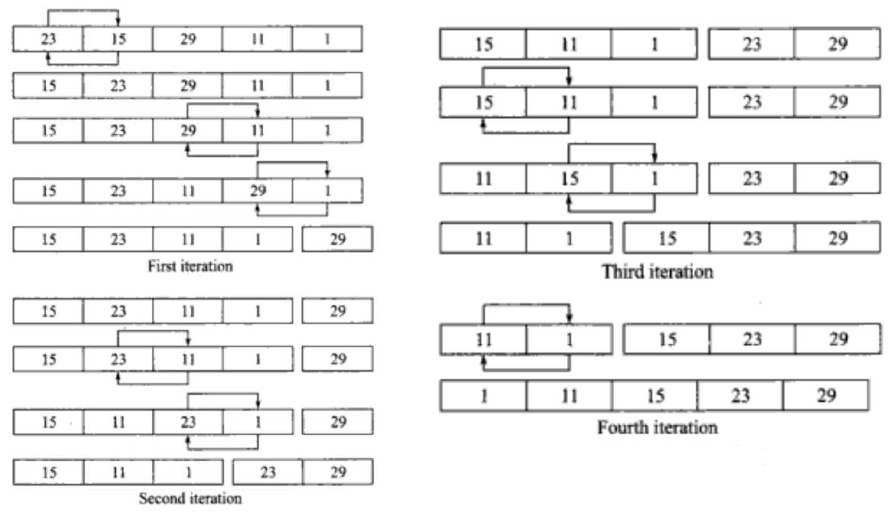
Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Types of Sorting:

* Bubble Sort
* Merge Sort
* Selection Sort
* Quick sort
* Insertion and heap Sort

**BUBBLE SORT:**

* In bubble sort, each element is compared with its adjacent element.
* We begin with the 0th element and compare it with the 1st element.
* If it is found to be greater than the 1st element, then they are interchanged.
* In this way all the elements are compared (excluding last) with their next element and are interchanged if required
* On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.



def bubblesort(list):

# Swap the elements to arrange in order

    for iter\_num in range(len(list)-1,0,-1):

        for idx in range(iter\_num):

            if list[idx]>list[idx+1]:

                temp = list[idx]

                list[idx] = list[idx+1]

                list[idx+1] = temp

list = [19,2,31,45,6,11,121,27]

bubblesort(list)

print(list)

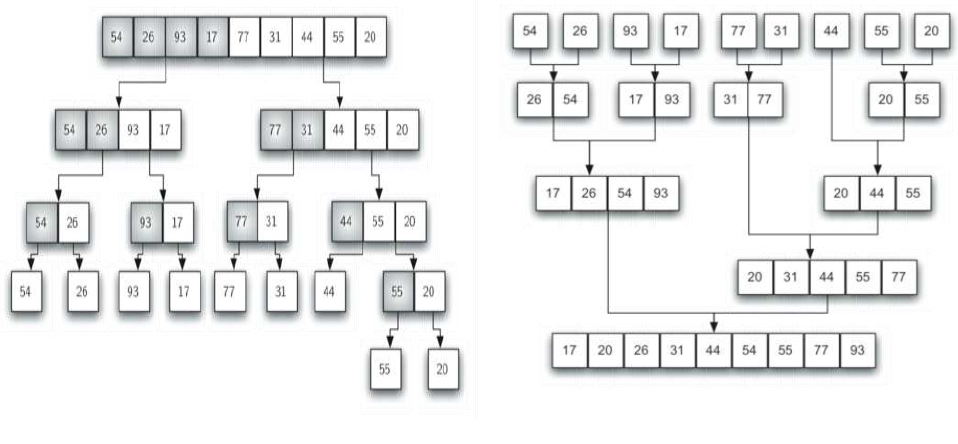
OUTPUT:

[2, 6, 11, 19, 27, 31, 45, 121]

**Merge Sort:**

Let’s start with array holding [54,26,93,17,71,31,44,55,20]

* + We can say that array[0..8] where p=0 and r=8
  + In the divide step we compute q=4
  + The conquer step has us sort the two subarrays
  + array[0..3] = [54,26,93,17]
  + array[4..8]= [71,31,44,55,20]
  + When we comeback from the conquer step, each of the two subarrays is sorted i.e.
  + array[0..3] = [17,26,54,93]
  + array[4..8]= [20,31,44,55,77]
  + Finally, the combine step merges the two sorted subarrays in first half and the second half, producing the final sorted array [17,20,26,31,44,54,55,77,93]



def merge\_sort(unsorted\_list):

    if len(unsorted\_list) <= 1:

        return unsorted\_list

# Find the middle point and devide it

    middle = len(unsorted\_list) // 2

    left\_list = unsorted\_list[:middle]

    right\_list = unsorted\_list[middle:]

    left\_list = merge\_sort(left\_list)

    right\_list = merge\_sort(right\_list)

    return list(merge(left\_list, right\_list))

# Merge the sorted halves

def merge(left\_half,right\_half):

    res = []

    while len(left\_half) != 0 and len(right\_half) != 0:

        if left\_half[0] < right\_half[0]:

            res.append(left\_half[0])

            left\_half.remove(left\_half[0])

        else:

            res.append(right\_half[0])

            right\_half.remove(right\_half[0])

    if len(left\_half) == 0:

        res = res + right\_half

    else:

        res = res + left\_half

    return res

unsorted\_list = [64, 34, 25, 12, 22, 11, 90]

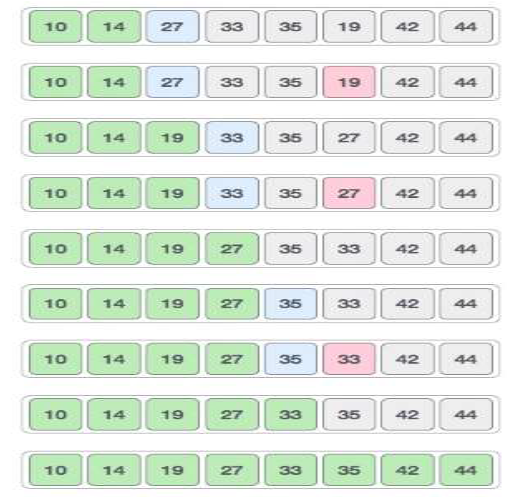
print(merge\_sort(unsorted\_list))

OUTPUT:

[11, 12, 22, 25, 34, 64, 90]

**Selection Sort:**

* + Find the least( or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.
  + Let a[n] be a linear array of n elements. The selection sort works as follows:
  + Pass 1: Find the location loc of the smallest element in the list of n elements a[0], a[1], a[2], a[3], .........,a[n-1] and then interchange a[loc] and a[0].
  + Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], .........,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.
  + Then we will get the sorted list a[0]<=a[2]<=a[3]…...<=a[n-1]



def selection\_sort(input\_list):

    for idx in range(len(input\_list)):

        min\_idx = idx

        for j in range( idx +1, len(input\_list)):

            if input\_list[min\_idx] > input\_list[j]:

                min\_idx = j

# Swap the minimum value with the compared value

        input\_list[idx], input\_list[min\_idx] = input\_list[min\_idx], input\_list[idx]

l = [19,2,31,45,30,11,121,27]

selection\_sort(l)

print(l)

OUTPUT:

[2, 11, 19, 27, 30, 31, 45, 121]

**Quick Sort:**

Quick Sort begins by partitioning the list - picking one value of the list that will be in its sorted place. This value is called a pivot. All elements smaller than the pivot are moved to its left. All larger elements are moved to its right. Knowing that the pivot is in it's rightful place, we recursively sort the values around the pivot until the entire list is sorted.

# There are different ways to do a Quick Sort partition, this implements the

# Hoare partition scheme. Tony Hoare also created the Quick Sort algorithm.

def partition(nums, low, high):

    # We select the middle element to be the pivot.

    pivot = nums[(low + high) // 2]

    i = low - 1

    j = high + 1

    while True:

        i += 1

        while nums[i] < pivot:

            i += 1

        j -= 1

        while nums[j] > pivot:

            j -= 1

        if i >= j:

            return j

        # If an element at i (on the left of the pivot) is larger than the

        # element at j (on right right of the pivot), then swap them

        nums[i], nums[j] = nums[j], nums[i]

def quick\_sort(nums):

    # Create a helper function that will be called recursively

    def \_quick\_sort(items, low, high):

        if low < high:

            # This is the index after the pivot, where our lists are split

            split\_index = partition(items, low, high)

            \_quick\_sort(items, low, split\_index)

            \_quick\_sort(items, split\_index + 1, high)

    \_quick\_sort(nums, 0, len(nums) - 1)

# Verify it works

random\_list\_of\_nums = [22, 5, 1, 18, 99]

quick\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

### **Insertion Sort:**

Like Selection Sort, this algorithm segments the list into sorted and unsorted parts. It iterates over the unsorted segment, and inserts the element being viewed into the correct position of the sorted list.

We assume that the first element of the list is sorted. We then go to the next element, let's call it x. If x is larger than the first element we leave as is. If x is smaller, we copy the value of the first element to the second position and then set the first element to x.

As we go to the other elements of the unsorted segment, we continuously move larger elements in the sorted segment up the list until we encounter an element smaller than x or reach the end of the sorted segment, and then place x in it's correct position.

def insertion\_sort(nums):

    # Start on the second element as we assume the first element is sorted

    for i in range(1, len(nums)):

        item\_to\_insert = nums[i]

        # And keep a reference of the index of the previous element

        j = i - 1

        # Move all items of the sorted segment forward if they are larger than

        # the item to insert

        while j >= 0 and nums[j] > item\_to\_insert:

            nums[j + 1] = nums[j]

            j -= 1

        # Insert the item

        nums[j + 1] = item\_to\_insert

# Verify it works

random\_list\_of\_nums = [9, 1, 15, 28, 6]

insertion\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

OUTPUT:

[1, 6, 9, 15, 28]

### **Heap Sort:**

This popular sorting algorithm, like the Insertion and Selection sorts, segments the list into sorted and unsorted parts. It converts the unsorted segment of the list to a Heap data structure, so that we can efficiently determine the largest element.

We begin by transforming the list into a **Max Heap** - a Binary Tree where the biggest element is the root node. We then place that item to the end of the list. We then rebuild our Max Heap which now has one less value, placing the new largest value before the last item of the list.We iterate this process of building the heap until all nodes are removed.

def heapify(nums, heap\_size, root\_index):

    # Assume the index of the largest element is the root index

    largest = root\_index

    left\_child = (2 \* root\_index) + 1

    right\_child = (2 \* root\_index) + 2

    # If the left child of the root is a valid index, and the element is greater

    # than the current largest element, then update the largest element

    if left\_child < heap\_size and nums[left\_child] > nums[largest]:

        largest = left\_child

    # Do the same for the right child of the root

    if right\_child < heap\_size and nums[right\_child] > nums[largest]:

        largest = right\_child

    # If the largest element is no longer the root element, swap them

    if largest != root\_index:

        nums[root\_index], nums[largest] = nums[largest], nums[root\_index]

        # Heapify the new root element to ensure it's the largest

        heapify(nums, heap\_size, largest)

def heap\_sort(nums):

    n = len(nums)

    # Create a Max Heap from the list

    # The 2nd argument of range means we stop at the element before -1 i.e.

    # the first element of the list.

    # The 3rd argument of range means we iterate backwards, reducing the count

    # of i by 1

    for i in range(n, -1, -1):

        heapify(nums, n, i)

    # Move the root of the max heap to the end of

    for i in range(n - 1, 0, -1):

        nums[i], nums[0] = nums[0], nums[i]

        heapify(nums, i, 0)

# Verify it works

random\_list\_of\_nums = [35, 12, 43, 8, 51]

heap\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

OUTPUT:

[8, 12, 35, 43, 51]