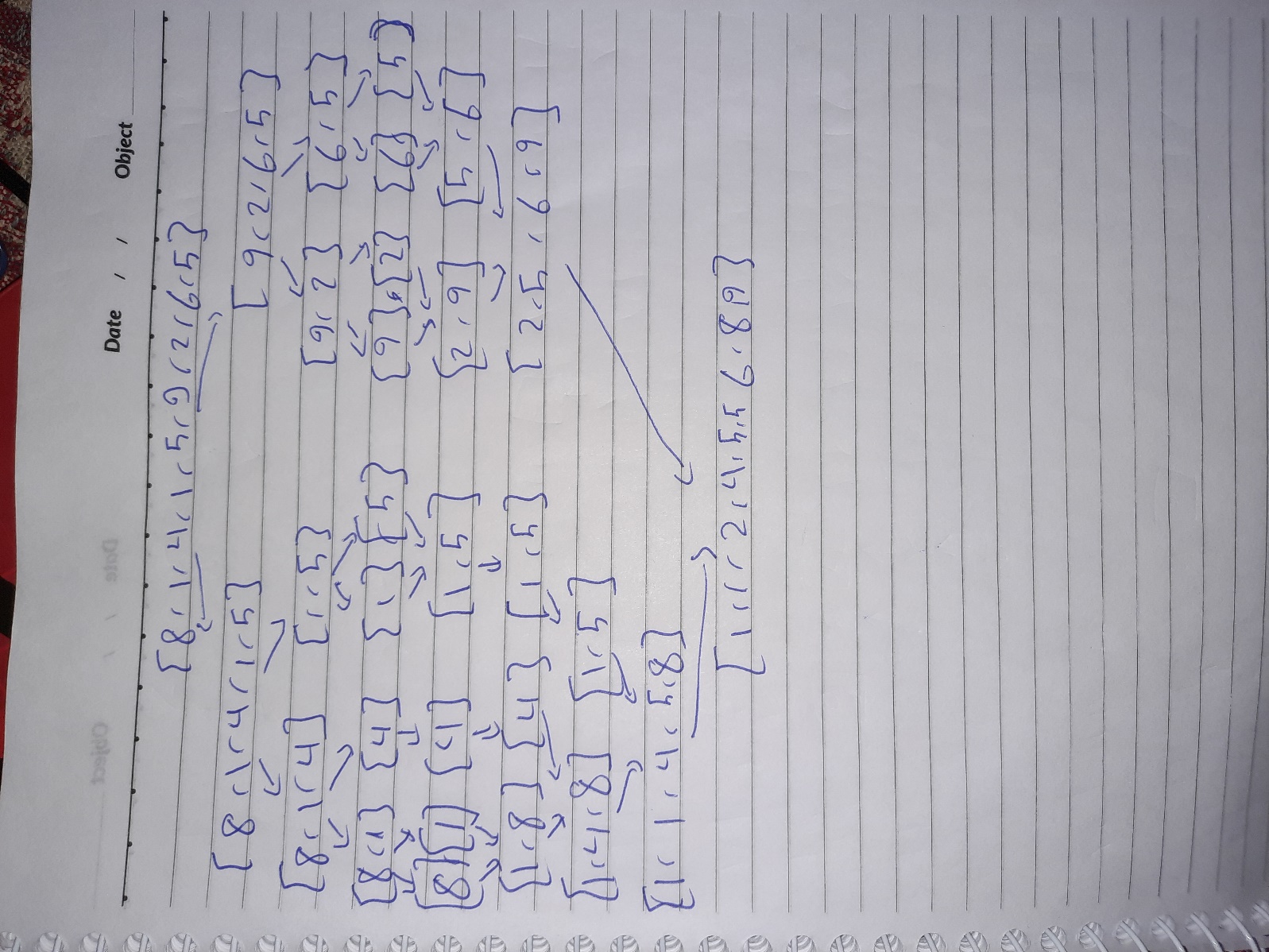
**BY: Fadi Alahmad Alomar 120180049**

# Question 1  
a) Insertion sort:  
[8,1,4,1,5,9,2,6,5]  
swap(arr[0],arr[1])  
[1,8,4,1,5,9,2,6,5]  
swap(arr[1],arr[2])  
[1,4,1,8,5,9,2,6,5]  
swap(arr[0],arr[1])  
[1,1,4,8,5,9,2,6,5]  
swap(arr[4],arr[3])  
[1,1,4,5,8,9,2,6,5]  
swap(arr[6],arr[5])  
[1,1,4,5,8,2,9,6,5]  
swap(arr[5],arr[4])  
[1,1,4,5,2,8,9,6,5]  
swap(arr[4],arr[3])  
[1,1,4,2,5,8,9,6,5]  
swap(arr[3],arr[2])  
[1,1,2,4,5,8,9,6,5]  
swap(arr[7],arr[6])  
[1,1,4,2,5,8,6,9,5]  
swap(arr[6],arr[5])  
[1,1,4,2,5,6,8,9,5]  
swap(arr[8],arr[7])  
[1,1,4,2,5,6,8,5,9]  
swap(arr[7],arr[6])  
[1,1,4,2,5,6,5,8,9]  
swap(arr[6],arr[5])  
[1,1,4,2,5,5,6,8,9]  
  
b) Heap sort:  
max heapify arr  
[9,6,8,5,5,4,2,1,1]  
swap(arr[0],arr[-1])  
[1,6,8,5,5,4,2,1,9]  
heapify arr[:-1]  
[8,6,4,5,5,1,2,1,9]  
swap(arr[0],arr[-2])  
[1,6,4,5,5,1,2,8,9]  
heapify arr[:-2]  
[6,5,4,1,5,1,2,8,9]  
swap(arr[0],arr[-3])  
[2,5,4,1,5,1,6,8,9]  
heapify arr[:-3]  
[5,5,4,1,2,1,6,8,9]  
swap(arr[0],arr[-4])  
[1,5,4,1,2,5,6,8,9]  
heapify arr[:-4]  
[5,2,4,1,1,5,6,8,9]  
swap(arr[0],arr[-5])  
[1,2,4,1,5,5,6,8,9]  
heapify arr[:-5]  
[4,2,1,1,5,5,6,8,9]  
swap(arr[0],arr[-6])  
[1,2,1,4,5,5,6,8,9]  
heapify arr[:-6]  
[2,1,1,4,5,5,6,8,9]  
swap(arr[0],arr[-7])  
[1,1,2,4,5,5,6,8,9]

c) Merge sort  
  
d) Quick sort  
[8, 1, 4, 1, 5, 9, 2, 6, 5] ,p = 5 ,l = 0 ,r = 8  
[8, 1, 4, 1, 5, 9, 2, 6, 5] the important part of the array on which the algorithm is working  
swapping arr[0], arr[8]  
array after swapping = [5, 1, 4, 1, 5, 9, 2, 6, 8] ,l = 1 ,r = 7  
swapping arr[4], arr[6]  
array after swapping = [5, 1, 4, 1, 2, 9, 5, 6, 8] ,l = 5 ,r = 5  
swapping arr[5], arr[6]  
array after swapping = [5, 1, 4, 1, 2, 5, 9, 6, 8] ,l = 5 ,r = 4  
[5, 1, 4, 1, 2, 5, 9, 6, 8] full array after modification  
  
[5, 1, 4, 1, 2, 5, 9, 6, 8] ,p = 6 ,l = 6 ,r = 8  
[9, 6, 8] the important part of the array on which the algorithm is working  
swapping arr[6], arr[7]  
array after swapping = [6, 9, 8] ,l = 7 ,r = 6  
swapping arr[6], arr[6]  
array after swapping = [6, 9, 8] ,l = 7 ,r = 6  
[5, 1, 4, 1, 2, 5, 6, 9, 8]  
  
[5, 1, 4, 1, 2, 5, 6, 9, 8] ,p = 9 ,l = 7 ,r = 8  
[9, 8] the important part of the array on which the algorithm is working  
swapping arr[7], arr[8]  
array after swapping = [8, 9] ,l = 8 ,r = 7  
swapping arr[8], arr[8]  
array after swapping = [8, 9] ,l = 8 ,r = 7  
[5, 1, 4, 1, 2, 5, 6, 8, 9] full array after modification  
  
[5, 1, 4, 1, 2, 5, 6, 8, 9] ,p = 4 ,l = 0 ,r = 4  
[5, 1, 4, 1, 2] the important part of the array on which the algorithm is working  
swapping arr[0], arr[4]  
array after swapping = [2, 1, 4, 1, 5] ,l = 1 ,r = 3  
swapping arr[2], arr[3]  
array after swapping = [2, 1, 1, 4, 5] ,l = 3 ,r = 2  
swapping arr[3], arr[3]  
array after swapping = [2, 1, 1, 4, 5] ,l = 3 ,r = 2  
[2, 1, 1, 4, 5, 5, 6, 8, 9] full array after modification  
  
[2, 1, 1, 4, 5, 5, 6, 8, 9] ,p = 1 ,l = 0 ,r = 2  
[2, 1, 1] the important part of the array on which the algorithm is working  
swapping arr[0], arr[2]  
array after swapping = [1, 1, 2] ,l = 1 ,r = 1  
swapping arr[1], arr[1]  
array after swapping = [1, 1, 2] ,l = 2 ,r = 0  
[1, 1, 2, 4, 5, 5, 6, 8, 9] full array after modification  
  
[1, 1, 2, 4, 5, 5, 6, 8, 9] ,p = 1 ,l = 0 ,r = 1  
[1, 1] the important part of the array on which the algorithm is working  
swapping arr[0], arr[1]  
array after swapping = [1, 1] ,l = 1 ,r = 0  
[1, 1, 2, 4, 5, 5, 6, 8, 9] full array after modification  
  
[1, 1, 2, 4, 5, 5, 6, 8, 9] Final result

PYTHON CODE FOR QUICK SORT:

def quickR(x, l, r):  
 if l >= r:  
 return  
 p = (l + r) // 2  
 lc = rc = False  
 lo = l  
 ro = r  
 po = x[p]  
 pp = p  
 print(x, ",p =", po, ",l =", lo, ",r =", ro)  
 print(x[lo:ro + 1], "the important part of the array on which the algorithm is working")  
 while l <= r:  
 while x[l] < po:  
 l += 1  
 while x[r] > po:  
 r -= 1  
 if l <= r:  
 print("swapping arr[{}], arr[{}]".format(l, r))  
 x[r], x[l] = x[l], x[r]  
 if x[p] != po:  
 if x[r] == po:  
 pp = r  
 lc = True  
 else:  
 pp = l  
 rc = True  
 l += 1  
 r -= 1  
 print("array after swapping =", x[lo:ro + 1], ",l =", l, ",r =", r)  
 if pp != p and lc:  
 print("swapping arr[{}], arr[{}]".format(l, pp))  
 x[pp], x[l] = x[l], x[pp]  
 print("array after swapping =", x[lo:ro + 1], ",l =", l, ",r =", r)  
 elif pp != p and rc:  
 print("swapping arr[{}], arr[{}]".format(pp, r))  
 x[pp], x[r] = x[r], x[pp]  
 print("array after swapping =", x[lo:ro + 1], ",l =", l, ",r =", r)  
 if rc:  
 print(x)  
 print()  
 quickR(x, r + 1, ro)  
 quickR(x, lo, r - 1)  
 else:  
 print(x, "full array after modification")  
 print()  
 quickR(x, l + 1, ro)  
 quickR(x, lo, l - 1)  
  
  
x = [8, 1, 4, 1, 5, 9, 2, 6, 5]  
quickR(x, 0, 8)  
print(x, " Final result")

# Question 3  
transfer the two strings into arrays then sort them using heap sort, each character has an ascii value thus it can be treated as a number when comparing it to other characters  
going through both the arrays and if they are the same then the words are an anagram if the arrays are not the same then the words are not an anagram

# Question 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Best Time | Worst Time | Average Time | Space Required | Advantages | Disadvantages |
| Insertion Sort | O(n) | O(n2) | O(n2) | O(1) | 1- Suitable for small-sized arrays  2- Takes less time in case of sorted arrays  3-Simple and easy to implement | Relatively slow with large-sized arrays compared to other sorting algorithms |
| Merge Sort | O(nlogn) | O(nlogn) | O(nlogn) | O(n) | 1- Consistent runtime making it efficient  2- Quicker for large lists than Insertion Sort | 1- Requires additional memory (for merging sub- arrays)  2- Takes the same time even if the list is already sorted |
| Quick Sort | O(nlogn) | O(n2) | O(nlogn) | O(logn) | 1- Regarded as the best sorting algorithms  2- Deals well with large arrays | 1- The worst-case is similar to the average of insertion sort and other slow sorting algorithms  2- Insertion Sort is more suited for sorted arrays |