**Sorting Algorithms:**

| Group Number | *G-04* |
| --- | --- |
| Registration Number of Group Members | 2020-CS-110  2020-CS-104 |

**Sorting Algorithms**

**Selection Sort:**

|  |  |
| --- | --- |
| **Description** | Selection sort is one of the simplest procedures of sorting. It is a comparison-based Algorithm. In the first iteration, it first finds the smallest element of the given array and then swaps it with the first element of the array and then the array divides into 2 arrays. One is sorted which contains 1 element and the other is sorted which contains all other elements. Then in the next iteration, it finds the smallest element in the unsorted array and swaps it with the first element of the unsorted array and it becomes part of the sorted Array. Similarly, it continues to sort the given elements, and at the end sorted array is returned. |
| **Pseudo Code** | list: array of items  n: Length of list  for i = 1 to n - 1  min = i  for j = i+1 to n  if list[j] < list[min]  min = j  if indexMin != i  swap list[min] and list[i] |
| **Code** | def Selection(arr,size):      for i in range(0,size-1):          min = i          for j in range(i+1,size):              if(arr[j]<arr[min]):                  min=j          if(min !=i):              arr[i],arr[min]=arr[min],arr[i]      return arr |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n2 | | Average Case | n2 | | Worst Case | n2 | |
| **Proof of Correctness** |  |
| **Three Strengths** | 1. It works very well for a small number of inputs.  2. It will perform excellently on the array that is already sorted, as no element would be swapped in this case.  3. It does not require a lot of space, as it works in the original array and no other new array is used. |
| **Three Weakness** | 1. It would not perform well for a large number of inputs.  2. The time complexity is n^2 in the worse case and it will consume more time.  3. Its efficiency decreases with the increase in a number of inputs.  4. It is not a stable sort. |
| **Dry Run** |  |

**Insertion Sort:**

|  |  |
| --- | --- |
| **Description** | Insertion sort is also a comparison-based sorting algorithm. It is inspired by the way in which we sort playing cards.  Before starting the implementation, we divide the given array into a sorted array(which contains only the first element of the array) and an unsorted array(which contains the remaining elements of the array considered to be unsorted). Now the first iteration will start from the second element of the array considered to be the first element of the unsorted array and take it as “Key element” and compare it with the sorted array elements and place it at its right place by comparison with sorted elements. Then similarly second-time first element of the unsorted array will be taken and will be placed in a sorted array and at the end, a sorted array will be returned. |
| **Pseudo Code** | INSERTION-SORT(A)  for i = 1 to n  key =A [i]  j = i – 1  while j > = 0 and A[j] > key  A[j+1] = A[j]  j =j – 1  A[j+1] = key |
| **Code** | def insertion\_sort(A):        for i in range(1, len(A)):          key = A[i]          j = i - 1            # Compare key with each element on the left of it until an element smaller than it is found          # For descending order, change key<array[j] to key>array[j].          while j >= 0 and key < A[j]:              A[j + 1] = A[j]              j = j - 1            # Place key at after the element just smaller than it.          A[j + 1] = key |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n | | Average Case | n2 | | Worst Case | n2 | |
| **Proof of Correctness** | Proof of correctness using loop invariance is shown below:  Initialization: As j=2 therefore, A [1 … j-1] = A [1] array of single element is always sorted.  Maintenance: For a particular j, A [1 … j-1] is sorted while loop -> A [j] in the correct position A [1 … j ] is now sorted. At the beginning of next iteration j becomes j+1 A [ 1 … j] is sorted -> which means loop invariant holds.  Termination: The ‘for’ loop terminates when j > n (i.e., j = n+1) The subarray is A [1 … n] by definition is in sorted order. |
| **Three Strengths** | * It works very well for a small number of inputs. * It would not spend much time if the array is already sorted, * It does not require a lot of space, as it works in the original array and no other new array is used * It is stable sort |
| **Three Weakness** | * It works very well for a small number of inputs. * Its time complexity of the worst case is n^2 * It would take more time if the array is entirely unsorted and it has to do n-1 swapping. |
| **Dry Run** |  |

**Merge Sort:**

|  |  |
| --- | --- |
| **Description** | Merge sort is a famous sorting algorithm. It uses a divide and conquer paradigm for sorting. It divides the problem into sub-problems and solves them individually and then the sub-problems are further divided into more sub-problems and are solved. It then combines the results of sub-problems to get the solution to the original problem. e.g we have divided a given array into 2 arrays A, B and then we will merge it into a new array C in a sorted manner. It is also a stable Sort. |
| **Pseudo Code** | mergeSort(array, low,high)  If size =1  Return array  Else  m=(low+ high)/2  mergeSort ( array , low , m )  mergeSort(array , m+1 , high);  merge(array, low , m , high)  def Merge(A,a,m,b):  R= [] of size m+1-a  L= [] of size B-m  for i in range a to m+1:  L.append(A[i])  for j in range m+1 to n+1  R.append(A[j])  i=0  j=0  for k in range a to b+1  if(i<len(L) and j<len(R) and L[i]<R[j]):  A[k]=L[i]  i=i+1  elif(j<len(R) and i<len(L)and L[i]>=R[j]):  A[k]=R[j]  j=j+1  elif(i<len(L)):  A[k] =L[i]  i=i+1  elif(j<len(R)):  A[k]=R[j]  j=j+1  return A |
| **Code** | def merge(X,a,m,b):        left\_copy = X[a:m + 1]  # Copy first half into one Array      right\_copy = X[m+1:b+1] # Copy Second Half into Second Array        # Necessary Variables      left\_ind = 0      right\_ind = 0      sort\_ind = a        # This Loop will Copy the Element from Both Left and Right Array in a sorted Manner      while left\_ind < len(left\_copy) and right\_ind < len(right\_copy):          if left\_copy[left\_ind] <= right\_copy[right\_ind]:              X[sort\_ind] = left\_copy[left\_ind]              left\_ind = left\_ind + 1          # Opposite from above          else:              X[sort\_ind] = right\_copy[right\_ind]              right\_ind = right\_ind + 1          # Regardless of where we got our element from          # move forward in the sorted part          sort\_ind = sort\_ind + 1        # We ran out of elements either in left\_copy or right\_copy      # so we will go through the remaining elements and add them      while left\_ind < len(left\_copy):          X[sort\_ind] = left\_copy[left\_ind]          left\_ind = left\_ind + 1          sort\_ind = sort\_ind + 1      while right\_ind < len(right\_copy):          X[sort\_ind] = right\_copy[right\_ind]          right\_ind = right\_ind + 1          sort\_ind = sort\_ind + 1 |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n | | Worst Case | n\*logn | |
| **Proof of Correctness** | Proof of correctness using loop invariance is shown below:  Initialization: Prior to the first iteration, k = a, so the subarray X[ a … k-1] is empty and k-a = 0 elements. I = j = 1, no left\_copy[i] and right\_copy[j] contain smallest elements not copied back into X.  Maintenance: Lets consider left\_copy[i] <= right\_copy[j]  Left\_copy[i] -> smallest element not copied into A  X[a … k-1] contain k-a smallest elements.  Now X[k] = Left\_copy[i] therefore, X[a … k] contains k-a+1 smallest elements.  Now I is incremented, k is incremented in the for loop.  This help re-establish the loop invariant.  Termination: k = r+1. By definition of loop invariant. A[a … k-1] which is A[a … r] contains k-a = r-a+1 smallest elements in sorted order |
| **Three Strengths** | * It works very well for a larger number of inputs. * It has better time complexity which is n log n. * It has a constant running time. * It is a stable sort. * It is a recursive sorting algorithm. |
| **Three Weakness** | * Merge sort is comparatively slow for small number of inputs. * More memory is used to store elements of sub arrays * It will perform the whole process even if the array is already sorted. |
| **Dry Run** |  |

**Bubble Sort:**

|  |  |
| --- | --- |
| **Description** | Bubble sort is another comparison-based and easiest sorting algorithm to implement. It doesn’t use any extra space while sorting. It uses multiple passes through an array. In each pass, it compares the next element of itself and then swaps the pair if they are in the wrong order and this process goes on. And it continues until a full scan is passed without any swapping of any element means the given array is sorted. |
| **Pseudo Code** | procedure bubbleSort (list: array of items)  loop = list.count;  for i = 0 to loop-1:  swapped = false  for j = 0 to i:  if list[j] > list[j+1]:  swap (list[j], list[j+1])  swapped = true  if (not swapped)  break |
| **Code** | def Bubble(A,size):      n=size-1      for i in range(0,n):          swap=0          for j in range(i):              if(A[j]>A[j+1]):                  temp=A[j]                  A[j]=A[j+1]                  A[j+1]=temp                  swap=1          if(swap==0):              break      return A |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n | | Average Case | n2 | | Worst Case | n2 | |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is also Stable sort. * It can detect whether the list is already sorted or not. * In the case of the sorted array the time complexity is O (n). * It also works better for a small number of inputs |
| **Three Weakness** | * It works poorly for a larger number of inputs. * It has O (N^2) time complexity. * It would use more memory space. |
| **Dry Run** |  |

**Quick Sort:**

|  |  |
| --- | --- |
| **Description** | Quick Sort is a famous sorting algorithm. It is also a comparison-based algorithm but involves a divide and conquer approach as well. It also follows a recursive algorithm. It divides the given array into 2 arrays using a partitioning element also known as a pivot and the division is done in a way that all the elements on the left side of the pivot are smaller than the pivot and all the elements on the right side of the pivot are greater than the pivot. Also, pivot reaches its original position. |
| **Pseudo Code** | quickSort(arr [], low, high)  if (low < high)  pi = partition (arr, low, high)  quicksort (arr, low, pi - 1)  quicksort (arr, pi + 1, high)  partition (arr [], low, high)  pivot = arr[high]  i = (low - 1)  for (j = low; j <= high- 1; j++)  if (arr[j] < pivot)  i++  swap arr[i] and arr[j]  swap arr [i + 1] and arr[high])  return (i + 1) |
| **Code** | def quickSort(A,low,high):      if( low < high):          # pi is partitioning index, arr[pi] is now          # at right place          pi = partition(A, low, high)          quickSort(A,low, pi -1)          quickSort(A, pi + 1, high)  def partition(A,low,high):      # pivot (Element to be placed at right position)      pivot = A[high]      i = (low - 1) # Index of smaller element and indicates the                    # right position of pivot found so far      # If current element is smaller than the pivot      for j in range(low,high):          if(A[j] < pivot):              i += 1              # increment index of smaller element              A[i], A[j] = A[j], A[i]        A[i+1], A[high] = A[high], A[i+1]      return (i+1) |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n\*logn | | Worst Case | n2 | |
| **Proof of Correctness** |  |
| **Three Strengths** | * It works very well for a larger number of inputs. * It has better time complexity which is n log n. * No additional storage is required in case of quicksort * If the array split is half then there will be O(n\*(lg\*n)) |
| **Three Weakness** | * It is recursive and if recursion is not available to us then the implementation would be more difficult. * Its time complexity in the worst case is n\* n if the array divides into arrays of 1 and (n-1) . * It has a In-consistent running time. * It is not a stable sort. |
| **Dry Run** |  |

**Counting Sort:**

|  |  |
| --- | --- |
| **Description** | Counting sort is a sorting technique that is based on the range of input values. It is used to sort elements in linear time. Firstly, we will find the max value in the input array and then will make an array of that max number. Then we will count the number of occurrences of each array element from 0 to length -1 and assign it to the new array. Then this new array will be used to again retrieve the sorted version of the input array. It has linear runtime but it's not the best algorithm because the space complexity is quite high and it's only suitable when the number of input arrays is low. |
| **Pseudo Code** | CountingSort(input)  k = range of elements of array  count ← array of k + 1 zeros  output ← array of same length as input  for i = 0 to length(input) - 1  j = key(input[i])  count[j] += 1  for i = 1 to k do  count[i] += count[i - 1]  for i = length(input) - 1 to 0  j = key(input[i])  count[j] -= 1  output[count[j]] = input[i]  return output |
| **Code** | # Function to Find Maximum Element  def max\_num(A):      maxa = A[0]      for i in range(1,len(A)):          if( maxa < A[i]):              maxa = A[i]      return maxa  # Function to find Mininum Element  def min\_num(A):      minn = A[0]      for i in range(1,len(A)):          if( minn > A[i]):              minn = A[i]      return minn  # Counting Sort Function  def counting\_sort(A,B):      minn = min\_num(A)  # This Will Find Mininum Element of Input Array        # If Minimum Element is less than Zero than      # Add the positive of that Number in all elements to make it them all positive one      if(minn < 0):          for j in range(0,len(A)):              A[j] = A[j] + abs(minn)        # Find maximum element in input array to make that number array      k = max\_num(A)      C = [0]\*(k+1)        # This Loop will add 1 to the index of number in input array      # e.g 5 is Number than at 5 index in array C 1 will be added.      for j in range(0,len(A)):          C[A[j]] = C[A[j]] + 1        # Not add the previous index into next one in C array      for i in range(1,k+1):          C[i] = C[i] + C[i-1]        # Now place the elements at there right place.      for j in range(len(A)-1,-1,-1):          B[C[A[j]]-1 ] = A[j]          C[A[j]] = C[A[j]] - 1        # if the minimum element was negative then again minus that      # number from all elementsto attain the original array      if(minn < 0):          for j in range(0,len(A)):              B[j] = B[j] - abs(minn) |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | N | | Average Case | N | | Worst Case | N | |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is not a comparison-based sorting. * It has better time complexity which is n * It is a stable sort algorithm |
| **Three Weakness** | * I It cannot be used for non-integer values. * It requires more space. * It would be inappropriate if at least one elements of the array are too large. |
| **Dry Run** |  |

**Bucket Sort:**

|  |  |
| --- | --- |
| **Description** | Bucket sort is a sorting algorithm in which we divide the array into number of buckets and then we use a comparison-based sorting algorithm to sort individual buckets and then we combine these buckets. And at the end we get out sorted Array. |
| **Pseudo Code** | bucketSort(arr[], n)  1) Create n empty buckets  2) Do following for every array element arr[i].  .......a) Insert arr[i] into bucket[n\*array[i]]  3) Sort individual buckets using insertion sort.  4) Concatenate all sorted buckets. |
| **Code** | def BucketSort(A):      B=[]      for i in range(10):          B.append([])      i=0      for i in range(len(A)):          a=int(A[i]\*10)          B[a].append(A[i])      for i in range(10):          B[i]=InsertionSort(B[i])      k=0      for i in range(10):          for j in range(len(B[i])):              A[k]=B[i][j]              k=k+1      return A  def InsertionSort(A):      for j in range(1,len(A)):          key=A[j]          i=j-1          while(i>=0 and A[i]>key):              A[i+1]=A[i]              i=i-1          A[i+1]=key      return A |
| **Time Complexity** | The worst-case time complexity of Bucket Sort is: O(n²)  The average time complexity of Bucket Sort is: O(n+k) |
| **Proof of Correctness** |  |
| **Three Strengths** | * When elements are distributed in buckets each bucket can be processed independently. * You can sort smaller arrays. * It is efficient when the input are uniformly distributed. |
| **Three Weakness** | * Efficiency is sensitive to distribution of input values. * It is not more stable. * Cannot apply it to all data types. |
| **Dry Run** |  |

**Radix Sort:**

|  |  |
| --- | --- |
| **Description** | It is not a comparison-based sorting algorithm. It would call the counting sort digit wise on the numbers. It would firstly call the counting sort for least significant digit of the number and continues until most significant digit. In this way array would be sorted. |
| **Pseudo Code** | Radix\_Sort(Array,p)  for j = 1 to p do           int count\_array[10] = {0}  for i = 0 to n        count\_array[key of(Array[i]) in pass j]++            for k = 1 to 10 do                  count\_array[k] = count\_array[k] + count\_array[k-1]              for i = n-1 to 0                  result\_array[ count\_array[key of(Array[i])] ] = Array[j]                  count\_array[key of(Array[i])]--              for i=0 to n do                  Array[i] = result\_array[i] |
| **Code** | # Function to Find Maximum Element  def max\_num(A):      maxa = A[0]      for i in range(1,len(A)):          if( maxa < A[i]):              maxa = A[i]      return maxa  # Function to find Mininum Element  def min\_num(A):      minn = A[0]      for i in range(1,len(A)):          if( minn > A[i]):              minn = A[i]      return minn  def RadixSOrt(A,B,tens):      minn = min\_num(A)  # This Will Find Mininum Element of Input Array        # If Minimum Element is less than Zero than      # Add the positive of that Number in all elements to make it them all positive one      if(minn < 0):          for j in range(0,len(A)):              A[j] = A[j] + abs(minn)        # This is a new 2d array to store elements in specific Area.      C = [[] for i in range(10)]      # This is a array for counting the numbers      count = [0]\*10        # This Loop will add 1 to the index of number in input array      # e.g 5 is Number than at 5 index in array C 1 will be added.      for j in range(0,len(A)):          C[int(str(A[j]/tens).split(".")[1][0])].append(A[j])        count[0] = len(C[0])      # Not add the previous index into next one in C array      for i in range(1,10):          count[i] = len(C[i]) + count[i-1]        # Now place the elements at there right place.      for j in range(len(A)-1,-1,-1):          k = int(str(A[j]/tens).split(".")[1][0])          c =  count[k]-1          B[c] = C[k][len(C[k])-1]          C[k].pop()          count[k] = count[k] - 1        # if the minimum element was negative then again minus that      # number from all elementsto attain the original array      if(minn < 0):          for j in range(0,len(A)):              B[j] = B[j] - abs(minn)      return B |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | N \* k | | Average Case | N \* k | | Worst Case | N \* k | |
| **Proof of Correctness** |  |
| **Three Strengths** | * Radix sort works efficiently when the range of input is not too large. * Time complexity of radix sort is better than comparison-based algorithms and it is O (n). * Radix sort is a stable sort |
| **Three Weakness** | * It would more space even more than quicksort * Radix sort is less flexible. * The constant for the radix sort is compared greatly unlike another sorting algorithm |
| **Dry Run** |  |