**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** | Proof of correctness using loop invariance is shown below:  The elements in ‘array’[1….i-1], are the smallest of the array in sorted order  **Initialization:** At i = 0, the array is empty, since there are no smallest elements in the array and the condition holds.  **Maintenance:** There are 2 sub-arrays, one consists of [0 … i] which is sorted and other consists of [i+1 … n] elements order.  At i = i+1, the min\_index is equal to the value in A[i]. The for loop then finds the smallest element in the next sub-array, and if it does, it swaps with the A[i].  In the next iteration, the sub array ‘array’ [0...i] does indeed consist of the smallest numbers in sported order, as now the sub-array becomes [1 … i].  **Termination:** The loop invariant terminates when i = n +1, i.e., i > n  Therefore, selection Sort is correct. |
| **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 worst 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.  Therefore, Insertion sort is correct. |
| **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 \* log n | | Worst Case | n \* log n | |
| **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  Therefore, Merge Sort is correct. |
| **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** | Proof of correctness using loop invariance is shown below:  **Initialization:** At i = 0, the array is empty, since there are no smallest elements in the array and the condition holds.  **Maintenance:** There are 2 sub-arrays, one consists of [0 … i] which is sorted and other consists of [i+1 … n] elements order.  At i = i+1, the algorithm keeps traversing the array, swapping any two unsorted element together.  In the next iteration, the sub array ‘array’ [0...i-1] does indeed consist of the smallest numbers in sported order, as now the sub-array becomes [1 … i].  **Termination:** The loop invariant terminates when i = n +1, i.e., i > n  Therefore, Bubble Sort is Correct. |
| **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** | Proof of correctness using loop invariance is shown below:  **Initialization:** At the start of each Iteration, I = low-1 and j = low, it creates 3 conditions:   * If low<= k <= low-1, then array[k] <= x, true because no value of k satisfies the equation * If low <= k <= low-1, then array[k] > pivot, true because no value of k satisfies the equation * If k = high, the array[k] = pivot, but since no changes have been made to pivot, this condition is also true   **Maintenance:** At the next Iteration, if A[j] <= pivot A[j] will be swapped with the first element of the right sub-array and the index of last element of left sub-array is increased as I = I +1.  And if A[j] > pivot, then the only change is the last index of the right sub-array and the conditions remains valid because the last index is greater then pivot.  **Termination:** when j = high and therefore the array ‘A’ has been partitioned into 3 sub-arrays, one contains elements less than pivot. 2nd contain greater elements than pivot and 3rd contain element that are equal to pivot.  Therefore, Quick Sort is Correct. |
| **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** |  |

**Tim Sort:**

|  |  |
| --- | --- |
| **Description** | The **Tim sort** algorithm is considered a **hybrid** sorting algorithm made with the help of merge sort and insertion sort. It takes advantage of common patterns of data and solve the large-Scale real-world data.  Insertion sort is the best method to sort when data is already or partially sorted or the length of run is smaller than MIN\_RUN and merge sort is best when the input is large. MIN\_RUN is a value which is a power of 2 not more than 32 (or 64).  Tim Sort applies insertion sort on the small subarrays whose length is less than 32 or 64. And then merge these sorted arrays. |
| **Pseudo Code** | def TIMSORT(array)  minRun = 24  size1 = len(array)  for i in range(0,len(arr),RUN)  arr[x:x+RUN] = self.InsertionSort\_ascend(arr[x:x+RUN],col)  size2 = minRun  while size2 < size1  for i in range(0,size1,size2\*2)  mid = i + size2-1  end = min((i + size2\*2-1,(size1-1)))  left = array[i:mid]  right = array[mid+1:end+1]  arr = merge(left,right)  size2 = size2 \* 2  return array |
| **Code** | def timsort\_ascending(self,arr,col):          RUN = 2          for x in range (0,len(arr),RUN):              arr[x:x+RUN] = self.InsertionSort\_ascend(arr[x:x+RUN],col)          r = RUN          while r < len(arr):              for x in range(0,len(arr),2\* r):                  arr[x:x+ 2\*r] = self.merge\_asccending(arr[x:x+r],arr[x+r:x+2\*r],col)              r = r\*2          return arr |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | n | | Worst Case | n \* log n | |
| **Proof of Correctness** | Proof of correctness using loop invariance is shown below:  Tim sort divides the array into chunks of 32 elements. Then apply insertion sort into these chunks and then use merge sort in these sorted chunks to make a single sorted array  **Initialization**: Consider each chunk as separated arrays and subarray [0 to i] of each array. Each sub array is sorted because initially i is equal to 0 and the array containing only one element is considered to be sorted.  **Maintenance**: with each iteration a key value (array[i+1]) is compared with the elements of the sorted part and placed in its correct position followed by an increment of i by one. Thus, maintaining the loop invariant.  **Termination**: when i approaches the size of original array, the array is completely sorted meaning array [0 to i] is sorted  After termination each of the chunks are merged using merge sort into a single sorted array  Therefore, Tim Sort is correct. |
| **Three Strengths** | * It works very well for a larger number of inputs. * It has better time complexity which is n log n. * It is a stable sort. |
| **Three Weakness** | * More memory is used to store elements of sub arrays * It will perform the whole process even if the array is already sorted. |

**Shell Sort:**

|  |  |
| --- | --- |
| **Description** | Shell sort is the advanced version of Insertion Sort. It Sorts the element based on the interval basis. It avoids large shifts as used to happens in Insertion Sort. For example, if we take the interval of 4 then it will compare the first element with the 5th element and then reduces the interval as the iterations continues. |
| **Pseudo Code** | shellSort()  A : array of items    while interval < A.length /3 do:  interval = interval \* 3 + 1  end while    while interval > 0 do:  for outer = interval; outer < A.length; outer ++ do:  valueToInsert = A[outer]  inner = outer;  while inner > interval -1 && A[inner - interval] >= valueToInsert do:  A[inner] = A[inner - interval]  inner = inner - interval    A[inner] = valueToInsert  interval = (interval -1) /2; |
| **Code** | n = len(array)              gap = n//2              while gap >= 1 :                for i in range(gap,n):                  temp = array[i]                  j = i - gap                  while j >= 0 and array[j][col] > temp[col] :                    array[j+gap] = array[j]                    j -= gap                  array[j+gap] = temp                gap = gap//2              return array |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | N \* log N | | Average Case | N\* log N | | Worst Case | n2 | |
| **Proof of Correctness** | Proof of correctness using loop invariance is almost same as insertion sort as insertion sort is proved right then its conditions is also almost same:  **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.  Therefore, Shell sort is correct. |
| **Three Strengths** | * This algorithm is quite efficient for medium-sized data sets * 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 is not a stable sort. * Its time complexity of the worst case is n^2 |

**Cock Tail Sort:**

|  |  |
| --- | --- |
| **Description** | Cocktail Sort is also known as Bi-directional Bubble Sort. It extends by appling bubble sort by sorting the given array from both sides. |
| **Pseudo Code** | Cocktail\_Sort(A : list of sortable items) is  do  swapped := false  for each i in 0 to length(A) − 2 do:  if A[i] > A[i + 1] then  swap(A[i], A[i + 1])  swapped := true  end if  end for  if not swapped then  // we can exit the outer loop here if no swaps occurred.  break do-while loop  end if  swapped := false  for each i in length(A) − 2 to 0 do:  if A[i] > A[i + 1] then  swap(A[i], A[i + 1])  swapped := true  end if  end for  while swapped // if no elements have been swapped, then the list is sorted |
| **Code** | isSwapped = True              start = 0              end = len(A) - 1              while (isSwapped == True):                  isSwapped = False                  i = start                  while ( i < end ):                      if (A[i][col] > A[i + 1][col]):                          A[i], A[i + 1] = A[i + 1], A[i]                          isSwapped = True                      i += 1                  if (isSwapped == False):                      break                  isSwapped = False                  i = end - 2                  while(  i > start - 1):                      if (A[i][col] > A[i + 1][col]):                          A[i], A[i + 1] = A[i + 1], A[i]                          isSwapped = True                      i = i - 1                  start = start + 1              return A |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | N2 | | Average Case | n2 | | Worst Case | n2 | |
| **Proof of Correctness** | Proof of correctness using loop invariance is almost the same as bubble sort because it is derived from it. is shown below:  **Initialization:** At i = 0, the array is empty, since there are no smallest elements in the array and the condition holds.  **Maintenance:** There are 2 sub-arrays, one consist of [0 … i] which is sorted and other consists of [i+1 … n] elements order.  At i = i+1, the algorithm keeps traversing the array, swapping any two unsorted element together.  In the next iteration, the sub array ‘array’ [0...i-1] does indeed consist of the smallest numbers in sported order, as now the sub-array becomes [1 … i].  **Termination:** The loop invariant terminates when i = n +1, i.e., i > n  Now reverse all the conditions for next for loop.  Therefore, CockTail Sort is Correct. |
| **Three Strengths** | * It is also Stable sort. * It can detect whether the list is already sorted or not. * 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. |

**Comb Sort:**

|  |  |
| --- | --- |
| **Description** | Comb Sort is the advanced form of bubble Sort. It is also a comparison-based sorting algorithm. Comb Sort uses a gap of size more than 1. The gape in the comb sort starts with the larger value and then shrinks by a factor of 1.3. It means that after the completion of each iteration, the gap is divided by the shrink factor of 1.3. The iteration continues until the gap is 1. At the end the array is sorted and returned. |
| **Pseudo Code** | combsort(array input)  gap := input.size  loop until gap = 1 and swaps = 0  gap := int(gap/1.3)  if gap < 1  //minimum gap is 1  gap := 1  end if  i := 0  swaps := 0  loop until i + gap >= input.size  if input[i] > input[i+gap]  swap(input[i], input[i+gap])  swaps := 1 //Flag a swap has occurred, so the  //list is not guaranteed sorted  i := i + 1 |
| **Code** | def divideNum(self,n):            n = n / 1.3            return 1 if n < 1 else int(n)      def CombSort\_ascend(self,A,col):          if (col != 5):              n = len(A)              swap = True              while not n == 1 or swap == True:                  n = self.divideNum(n)                  swap = False                  for i in range(0, len(A) - n):                        if A[i][col] > A[i + n][col]:                            A[i], A[i + n] = A[i + n], A[i]                          swap = True              return A |
| **Time Complexity** | |  |  | | --- | --- | | **Cases** | **Time Complexity** | | Best Case | N | | Average Case | n2 | | Worst Case | n2 | |
| **Proof of Correctness** | Proof of correctness using loop invariance is almost the same as bubble sort because it is derived from it. is shown below:  **Initialization:** At i = 0, the array is empty, since there are no smallest elements in the array and the condition holds.  **Maintenance:** There are 2 sub-arrays, one consists of [0 … i] which is sorted and other consists of [i+1 … n] elements order.  At i = i+n, the algorithm keeps traversing the array, swapping any two unsorted element after the gap number.  In the next iteration, the sub array ‘array’ [0...i-1] does indeed consist of the smallest numbers in sported order, as now the sub-array becomes [1 … i].  **Termination:** The loop invariant terminates when i = n +1, i.e., i <= 1  Therefore, COMB Sort is Correct. |
| **Three Strengths** | * It is also Stable sort. * It can detect whether the list is already sorted or not. * 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. |