**Proposer Details**

| Group Number | *G-17* |
| --- | --- |
| Registration Number of Group Members | 2020-CS-134  2020-CS-157 |

**Sorting Algorithms**

**Insertion Sort:**

|  |  |
| --- | --- |
| **Description** | Insertion sort involves finding the right place for a given element in a sorted list. So, in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position. |
| **Pseudo Code** | Insertion\_Sort(A):  for i=1 to A.length  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** |  |
| **Time Complexity** | The best-case time complexity of Insertion Sort is: O(n)  The worst-case time complexity of Insertion Sort is: O(n²) |
| **Proof of Correctness** |  |
| **Three Strengths** | * Space requirement is minimum. * Main advantage of insertion sort is its simplicity. * It has good ability to sort small lists. |
| **Three Weakness** | * It does not perform well as other sorting algorithms perform. * It requires n number of steps for sorting. * It does not deal well with huge lists. |
| **Dry Run** |  |

**Selection Sort:**

|  |  |
| --- | --- |
| **Description** | In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. |
| **Pseudo Code** | Selection-Sort(A):  for i=1 to A.length-1:  ji=i  for j=i+1 to A.length:  if A[j]<A[ji]:  ji=j  temp=A[i]  A[i]=A[ji]  A[ji]=temp |
| **Code** |  |
| **Time Complexity** | The best and worst-case time complexity of Selection Sort is: O(n²) |
| **Proof of Correctness** |  |
| **Three Strengths** | * It performs well on small lists. * No temporary storage is required. * It performs well on items that have already been sorted. |
| **Three Weakness** | * Poor efficiency while dealing with huge lists. * As bubble sort it requires n number of steps for sorting. * It is just suitable for lists having small number of elements. |
| **Dry Run** |  |

**Merge Sort:**

|  |  |
| --- | --- |
| **Description** | This is a divide and conquer algorithm. In this algorithm we split a list in half and keeps splitting the list by 2 until it only has single element. Then we merge the sorted list. We keep doing this until we get a sorted list with all the elements of the unsorted input list. |
| **Pseudo Code** | Merge (A, a, m, b):  for t=1 to 9:  if A[i] < A[j]:  B[t]=A[i]  i++  else:  B[t]=A[j]  j++ |
| **Code** |  |
| **Time Complexity** | Time complexity of Merge Sort is O(n log n) in best and worst case. |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is suitable for huge lists. * It has a consistent running time. * It does not under-goes the whole list several times. |
| **Three Weakness** | * Extra space required to run subarrays. * Slow with respect to other sorting algorithms. * Goes through whole sorting process even the array is sorted. |
| **Dry Run** |  |

**Bubble Sort:**

|  |  |
| --- | --- |
| **Description** | Simplest sorting algorithm. Iterates over the list, in each iteration it compares elements in pairs and keeps swapping them such that the larger element is moved towards the end of the list. |
| **Pseudo Code** | BubbleSort(A):  for i=1 to A.length-1:  for j=A.length to i+1:  if A[j]<A[j-1]:  exchange A[j] with A[j-1] |
| **Code** |  |
| **Time Complexity** | The best-case time complexity of Bubble Sort is: O(n)  The worst-case time complexity of Bubble Sort is: O(n²) |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is popular and easy to implement. * It does not required any temporary storage. * It is easy to understand. |
| **Three Weakness** | * It does not deal well with huge lists. * It is just used for academic teaching not for real life. * It requires n number of steps for sorting. |
| **Dry Run** |  |

**Hybrid Sort:**

|  |  |
| --- | --- |
| **Description** | Hybrid Sort is a [hybrid](https://en.wikipedia.org/wiki/Hybrid_algorithm) [stable](https://en.wikipedia.org/wiki/Category:Stable_sorts) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), derived from [merge sort](https://en.wikipedia.org/wiki/Merge_sort) and [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort), designed to perform well on many kinds of real-world data. |
| **Pseudo Code** |  |
| **Code** |  |
| **Time Complexity** | Time complexity of Hybrid Sort is O(n²). |
| **Proof of Correctness** |  |
| **Three Strengths** | * No additional storage is required. * It is said to be the best sorting algorithm. * Space requirement is minimum. |
| **Three Weakness** | * It does not perform well as other sorting algorithms perform. * It requires n number of steps for sorting. |
| **Dry Run** |  |

**Quick Sort:**

|  |  |
| --- | --- |
| **Description** | In this algorithm we partition the list around a pivot element, sorting values around the pivot. In my solution I used the last element from the list as pivot value. Best performance is achieved when the pivot value splits the list in two almost equal halves. |
| **Pseudo Code** | QuickSort(A ,p ,r):  if p<r:  q=Partition (A, p, r)  QuickSort(A, p,q-1):  QuickSort(A, q+1, r):  Initial Call QuickSort(A,1, A.length) |
| **Code** |  |
| **Time Complexity** | Time complexity of Quick Sort in best-case is O(nlogn).  Time complexity of Quick Sort in worst-case is O(n²). |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is said to be the best sorting algorithm. * It is able to deal well with huge lists. * No additional storage is required. |
| **Three Weakness** | * Its worst-case performance is equal to average case performance of insertion sort. * It is fragile. * It is destructive sort if array is already sorted. |
| **Dry Run** |  |

**K-Select/Quick Select:**

|  |  |
| --- | --- |
| **Description** | Quick-select/K-Select is a selection algorithm to find the Kth smallest element in an unordered list. It is related to the quick sort sorting algorithm. |
| **Pseudo Code** | quickSelect(list, left, right, k)  if left=right  return list[left]  Select a pivotIndex between left and right  pivotIndex=partition(list, left, right, pivotIndex)  if k=pivotIndex  return list[k]  else if k<pivotIndex  right=pivotIndex–1  else  left=pivotIndex+1 |
| **Code** |  |
| **Time Complexity** | The best-case time complexity of K-Select/Quick Select is: O(n)  The worst-case time complexity of K-Select/Quick Select is: O(n²) |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is efficient and has good average case. * These are often used in real world implementations. * It has low time complexity. |
| **Three Weakness** | * It is sensitive to the pivot that is chosen. * It has poor worst case time complexity. * It is not more stable. |
| **Dry Run** |  |

**Counting Sort:**

|  |  |
| --- | --- |
| **Description** | This algorithm does not do comparison between the elements. We use the mathematical properties of the integers to sort. We count how many time a number has come and store the count in the array where index is mapped to key’s value. |
| **Pseudo Code** | CountingSort(A,B,n,k):  Let C be array 0….k  for i=0 to k:  C[i]=0  for j=1 to n:  C[A[j]]+=1  for i=1 to k:  C[i]+=C[i-1]  for j=n downto 1:  B[C[A[j]]]=A[j]  C[A[j]]-=1 |
| **Code** |  |
| **Time Complexity** | Time complexity of Counting Sort in best-case and worst-case is O(k+n). |
| **Proof of Correctness** |  |
| **Three Strengths** | * Counting sort has better time complexity. * Counting sort’s run time is shorter as compared to other algorithms. * This is stable and non-comparison sort. |
| **Three Weakness** | * Counting sort only works when the range of potential items in the input is known ahead of time. * If the range of potential values is big, then counting sort requires a lot of space. * Counting sort can only be used for arrays with integer elements. |
| **Dry Run** |  |

**Heap Sort:**

|  |  |
| --- | --- |
| **Description** | We create two segments of the list one sorted and one unsorted. In this we use heap data structure to efficiently get the max element from the unsorted segment of the list. Heap method uses recursion to get the max element at the top. |
| **Pseudo Code** | Heapsort(A):  Build-Max-Heap(A)  for i=A.length downto 2:  exchange A[1] with A[i]  A.heap-size-=1  MAX-Heapify(A,1) |
| **Code** |  |
| **Time Complexity** | Time complexity of Heap Sort in best-case and worst-case is O(n log n). |
| **Proof of Correctness** |  |
| **Three Strengths** | * The Heap sort algorithm is widely used because of its efficiency. * The Heap sort algorithm can be implemented as an in-place sorting algorithm. * Space requirement is minimum. |
| **Three Weakness** | * It takes more time to compute. * Memory management is more complicated. * It is not stable. |
| **Dry Run** |  |

**Bucket Sort:**

|  |  |
| --- | --- |
| **Description** | Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable [sorting algorithms](https://www.programiz.com/dsa/sorting-algorithm) or recursively applying the same bucket algorithm. |
| **Pseudo Code** | BucketSort(A)  n = A.length  Let B[0, . . . , n − 1] be a new array  for i = 0 to n - 1  B[i] ← 0  for i = 1 to n  B[bnA[i]c] ← A[i]  for i = 0 to n-1  sort list B[i] using insertion sort  concatenate the lists B[0], B[1], . . . , B[n − 1]  return B |
| **Code** |  |
| **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** |  |

**Shell Sort:**

|  |  |
| --- | --- |
| **Description** | Shell Sort involves sorting elements which are away from each other. We sort a large sub-list of a given list and go on reducing the size of the list until all elements are sorted. |
| **Pseudo Code** | ShellSort(A, k):  for z=k.length downto 1:  b=k[z]  for i=b+1 to n:  j=i  temp=A[i]  while(j>=b) AND A[j-b]>temp:  A[j]=A[j-b]  j=j-b  A[j]=temp |
| **Code** |  |
| **Time Complexity** | Time complexity of Quick Sort in best and worst-case is O(nlogn). |
| **Proof of Correctness** |  |
| **Three Strengths** | * It is efficient for medium sized lists. * It is fastest as compared to all sorting algorithms. * It is more suitable than insertion sort. |
| **Three Weakness** | * It is only efficient for finite number of elements. * It is complex algorithm. * Limited to use for small size arrays. |
| **Dry Run** |  |

**Radix Sort:**

|  |  |
| --- | --- |
| **Description** | Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit. |
| **Pseudo Code** | RadixSort(A ,d):  for i=1 to d:  n=A.length  exp=10^(i-1)  for j=1 to n:  for k=j+1 to n:  if(A[j]/exp)%10>(A[k]/exp)%10  exchange A[i] with A[j] |
| **Code** |  |
| **Time Complexity** | The time complexity of Radix Sort is: O(d(n + k)) |
| **Proof of Correctness** |  |
| **Three Strengths** | * Algorithm is fast when the keys are short. * It is a stable sort. * It has better efficiency as compared to other sorting algorithms. |
| **Three Weakness** | * It takes more space as compared to other sorting algorithms. * It is much less flexible than other sorting algorithms. * The constant for Radix sort is greater compared to other sorting algorithms. |
| **Dry Run** |  |

**Tree Sort:**

|  |  |
| --- | --- |
| **Description** | Tree sort is a sorting algorithm that is based on [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) data structure. It first creates a binary search tree from the elements of the input list or array and then performs an in-order traversal on the created binary search tree to get the elements in sorted order. |
| **Pseudo Code** | Treesort(A):  tree=RBTree()  for element in A:  tree.insert(element)  i=0  for element in A.traverse():  A[i]=element  i+=1 |
| **Code** |  |
| **Time Complexity** | The time complexity of Tree Sort is: O(n²) |
| **Proof of Correctness** |  |
| **Three Strengths** | * We can make changes very easily. * Sorting is as fast as in quick sort. * It has better worst case complexity. |
| **Three Weakness** | * It requires separate memory for sorting. * Worst case occur when the elements in array is already sorted. * In worst case the time complexity is O(n^2). |
| **Dry Run** |  |