**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** | def Insert(arr):  arr\_size = len(arr)  for j in range(1,arr\_size):  key = arr[j]  i = j-1  while i>=0 and arr[i]>key:  arr[i+1] = arr[i]  i = i-1  arr[i+1] = key  return arr |
| **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** | **Proof by loop Invariant:**  **Initialization** - The subarray starts with the first element of the array, and it is (obviously) sorted to begin with.  **Maintenance** - Each iteration of the loop expands the subarray, but keeps the sorted property. An element gets inserted into the array only when it is greater than the element to its left. Since the elements to its left have already been sorted, it means is greater than all the elements to its left, so the array remains sorted.  **Termination** - The code will terminate after has reached the last element in the array, which means the sorted subarray has expanded to encompass the entire array. The array is now fully sorted. |
| **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** | def Selection\_Sort(A):  for i in range(len(A)):  min = i  for j in range(i+1, len(A)):  if A[min\_idx] > A[j]:  min = j  A[i], A[min] = A[min], A[i]  return |
| **Time Complexity** | The best and worst-case time complexity of Selection Sort is: O(n²) |
| **Proof of Correctness** | 1 for i = 1 to n-1  2 min = i  3 for j = i+1 to n  4 if A[j] < A[min]  5 min = j  6 swap A[i] with A[min]  First we prove the correctness of the inner loop: lines 3 to 5  **Initialization:**  Prior to the first iteration of the loop, j=i+1. So the array segment A[i..j-1]  is really just spot A[i]. Since line 2 of the code sets min = i, we have that  min indexes the smallest element (the only element) in subarray A[i..j-1] and  hence the loop invariant is true.  **Maintenance:**  Before pass j, we assume that min indexes the smallest element in the subarray  A[i..j-1]. During iteration j we have two cases: either A[j] < A[min]  or A[j] ≥ A[min]. In the second case, the if statement on line 4 is not true,  so nothing is executed. But now min indexes the smallest element of A[i..j].  In the first case, line 5 switches min to index location j since it is the smallest.  If min indexes an element less than or equal to subarray A[i..j-1] and now  A[j] < A[min], then it must be the case that A[j] is less than or equal to  elements in subarray A[i..j-1]. Line 5 switches min to index this new location  and hence after the loop iteration finishes, min indexes the smallest element in  subarray A[i..j].  **Termination:**  At termination of the inner loop, min indexes an element less than or equal to all  elements in subarray A[i..n] since j = n+1 upon termination. This finds the  smallest element in this subarray and is useful to us in the outer loop because we  can move that next smallest item into the correct location. |
| **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** | def Merge(arr,l,mid,r):  n1 = mid-l+1  n2 = r-mid  arr1 = []  arr2 = []  for n in range(n1):  arr1.append(arr[l+n])  for n in range(n2):  arr2.append(arr[mid+1+n])  i=0  j=0  k=l  while i<n1 and j<n2:  if arr1[i]<arr2[j]:  arr[k] = arr1[i]  i = i+1  k = k+1  else :  arr[k] = arr2[j]  j = j+1  k = k+1  while i<n1:  arr[k] = arr1[i]  i = i+1  k = k+1  while j<n2:  arr[k] = arr2[j]  j = j+1  k = k+1    def MergeSort(arr,l,r):  if l<r:  mid = math.floor((l+r)/2)  MergeSort(arr,l,mid)  MergeSort(arr,mid+1,r)  Merge(arr,l,mid,r) |
| **Time Complexity** | Time complexity of Merge Sort is O(n log n) in best and worst case. |
| **Proof of Correctness** | **1 Correctness of Merge**  We would like to prove that MergeSort works correctly. To do this, let us first look at the correctness of the merge function, as this is iterative, and we have done proofs like this before. Recall  that in such cases we want to find a loop invariant which is a condition that holds every time the  internal loop (or loops) is executed, and helps us prove correctness. In this case, let us focus on  the tmp array. At iteration k, suppose that the indices in the two parts of the array a are i and j.  Then, our loop invariant will be: tmp[k] ≤ a[l], ∀l ∈ {i, . . . m} and tmp[k] ≤ a[l], ∀l ∈ {j, . . . q}.  In other words, the element we just copy at position k is the minimum of the remaining elements.  Why does this condition help us? Since the tmp array is filled from left to right, all the elements  left in a will be put in later, at positionstmp[k+1], . . . tmp[q−p+1]. Hence, this condition means  that in the end of the merge, tmp[k] ≤ tmp[l], ∀k < l, which means that after copying, a will be  correctly sorted between p and q.  To see why the loop invariant is correct, recall that we assume that when merge is called, the  two parts of a between p and m and between m + 1 and q are already sorted. Hence:  a[i] ≤ a[l], ∀l ∈ {i + 1, . . . m}  and  a[j] ≤ a[l], ∀l ∈ {j + 1, . . . q}.  So a[i] and a[j] are the smallest remaining elements. If a[i] ≤ a[j], then tmp[k] will get value a[i],  and we also have that a[i] ≤ a[l], ∀l ∈ {j, . . . q}, The similar reasoning holds in the other case.  **2 Correctness of MergeSort**  Now that we know Merge works correctly, we will show that the entire algorithm works correctly,  using a proof by induction. For the base case, consider an array of 1 element (which is the base  case of the algorithm). Such an array is already sorted, so the base case is correct.  For the induction step, suppose that MergeSort will correctly sort any array of length less than  n. Suppose we call MergeSort on an array of size n. It will recursively call MergeSort on two  1  arrays of size n/2. By the induction hypothesis, these calls will sort these arrays correctly. Hence,  after the recursive calls, array a will be sorted between indices p, . . . m and m+1, . . . q respectively.  We have already showed that merge works correctly, hence after executing it, a will be correctly  sorted between p and q. This concludes our proof.  Recall that when you design recursive algorithms, you have to “put your faith” in the recursion,  assume it will work, then specify the processing that follows it. Induction basically gives you the  mathematical tool to prove that your “faith leap” is indeed justified. |
| **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** | def bubbleSort(arr):  n = len(arr)    # Traverse through all array elements  for i in range(n):    # Last i elements are already in place  for j in range(0, n-i-1):    # traverse the array from 0 to n-i-1  # Swap if the element found is greater  # than the next element  if arr[j] > arr[j+1] :  arr[j], arr[j+1] = arr[j+1], arr[j] |
| **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** | a. A'A consists of the elements in AA but in sorted order.  b. Loop invariant: At the start of each iteration of the for loop of lines 2-4, the subarray A[j..n]A[j..n] consists of the elements originally in A[j..n]A[j..n] before entering the loop but possibly in a different order and the first element A[j]A[j] is the smallest among them.  Initialization: Initially the subarray contains only the last element A[n]A[n], which is trivially the smallest element of the subarray.  Maintenance: In every step we compare A[j]A[j] with A[j - 1]A[j−1] and make A[j - 1]A[j−1] the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.  Termination: The loop terminates when j = ij=i. According to the statement of loop invariant, A[i]A[i] is the smallest among A[i..n]A[i..n] and A[i..n]A[i..n] consists of the elements originally in A[i..n]A[i..n] before entering the loop.  c. Loop invariant: At the start of each iteration of the for loop of lines 1-4, the subarray A[1..i − 1]A[1..i−1] consists of the i - 1i−1 smallest elements in A[1..n]A[1..n] in sorted order. A[i..n]A[i..n] consists of the n - i + 1n−i+1 remaining elements in A[1..n]A[1..n].  Initialization: Initially the subarray A[1..i − 1]A[1..i−1] is empty and trivially this is the smallest element of the subarray.  Maintenance: From part (b), after the execution of the inner loop, A[i]A[i] will be the smallest element of the subarray A[i..n]A[i..n]. And in the beginning of the outer loop, A[1..i − 1]A[1..i−1] consists of elements that are smaller than the elements of A[i..n]A[i..n], in sorted order. So, after the execution of the outer loop, subarray A[1..i]A[1..i] will consists of elements that are smaller than the elements of A[i + 1..n]A[i+1..n], in sorted order.  Termination: The loop terminates when i = A.lengthi=A.length. At that point the array A[1..n]A[1..n] will consists of all elements in sorted order.  d. The iith iteration of the for loop of lines 1-4 will cause n − in−i iterations of the for loop of lines 2-4, each with constant time execution, so the worst-case running time of bubble sort is \Theta(n^2)Θ(n  2  ) which is same as the worst-case running time of insertion sort. |
| **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** | def countSort(arr):    # The output character array that will have sorted arr  output = [0 for i in range(len(arr))]    # Create a count array to store count of individual  # characters and initialize count array as 0  count = [0 for i in range(256)]    # For storing the resulting answer since the  # string is immutable  ans = ["" for \_ in arr]    # Store count of each character  for i in arr:  count[ord(i)] += 1    # Change count[i] so that count[i] now contains actual  # position of this character in output array  for i in range(256):  count[i] += count[i-1]    # Build the output character array  for i in range(len(arr)):  output[count[ord(arr[i])]-1] = arr[i]  count[ord(arr[i])] -= 1    # Copy the output array to arr, so that arr now  # contains sorted characters  for i in range(len(arr)):  ans[i] = output[i]  return ans |
| **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** |  |