**Proposer Details**

| Group Number | *G-07* |
| --- | --- |
| Registration Number of Group Members | 2020-CS-24 |

**Sorting Algorithms**

|  |  |
| --- | --- |
| **Selection Sort** |  |
| **Description** | This algorithm works by finding minimum element in the unsorted part of array and then it puts it in the beginning of the array. Similarly, if there’s another smaller element than current element at the beginning it then moves it to the beginning. In this way it can pick item form unsorted part and compare it with the sorted part and then moves elements to their appropriate positions. It is an “**unstable**” algorithm based on placing and moving of array elements. |
| **Pseudo Code** | Selection-Sort(Array):  n = Array.length()  for i = 0 to n-1:  minmum\_index = i  for j = i+1 to n:  if Array[j] <= Array[min]:  // store index (j) into minimum  If minimum != i:  // swap i with minimum  return Array |
| **Code** | def selection\_sort(A):  n = len(A) # length  for idx in range(n-1):  min = idx  for j in range(idx+1, n):  if A[j] <= A[min]: # smaller element found  min = j # store index  if min != idx: # swap  temp = A[idx]  A[idx] = A[min]  A[min] = temp  return A |
| **Time Complexity** | Best-case time complexity: O(n²)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization:** when we’re at first index i = 0 Sub-array has one element and one element is always sorted.  **Maintenance:** when first iteration begins we move from i = 0 to i = n and at each iteration if A[minimum] is larger than any other element then it is swapped with that element. Thus, the sub-Array is always sorted.  **Termination:** Outer Loop terminates when i = n. Now when i > n (A.length) the previous sub-Array is sorted and the largest value is at the last index thus we don’t have to check the last element as it is moved to its rightful place. Whole array is sorted.  Thus, Algorithm is correct. |
| **Advantages** | * It is best for small array (little number of inputs). * No additional space is required as we’re manipulating a single array. * if array is already sorted it gives linear runtime[O(n)]. |
| **Disadvantages** | * It requires n2 number of steps for sorting array. Which is worse in accordance with other algorithms out there. * It is at its worst when there’s an array with greater number of inputs. * It is less efficient as its performance is dependent on the initial arrangement of array. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Bubble Sort** |  |
| **Description** | This algorithm work by continuously swapping two elements in an array based on their comparison with each other. It continues until all the elements aren’t in perfect order. Its execution takes almost same time as of Selection-Sort but it is a “**Stable**” algorithm. Since it uses flag for swapping if array is already sorted it takes less time for its execution. |
| **Pseudo Code** | Bubble-Sort(Array)  n = Array.length  for i = 0 to n-1 do:  swapped = false  for j = 0 to n-1 do:  // compare elements  if Array[j] > Array[j+1] then  // swap  swap(Array[j], Array[j+1] )  swapped = true  if(not swapped) then  break  return Array |
| **Code** | def bubble\_sort(A):  n = len(A) # length of Array  for i in range(n-1):  swaped = False # flag  for j in range(n-1):  if A[j] >= A[j+1]:  # swap with next element  temp = A[j+1]  A[j+1] = A[j]  A[j] = temp  swaped = True  # if not swaped then list is in order  if swaped == False :  break  return A |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization:** Initially the subarray A[1..i−1] is empty. This is the largest element of the subarray.  **Maintenance:** After the execution of the inner loop, A[i] will be the smallest element of the subarray A[i..n]. And in the beginning of the outer loop, A[1..i − 1] consists of elements that are smaller than the elements of A[i..n]A[i..n], in sorted order.  **Termination:** The loop terminates when i = A.length. At that point the array A[1..n]A[1..n] will consists of all elements in sorted order. |
| **Advantages** | * When list is small it is best because it then can sort list efficiently. * It is simple to implement and easy to understand. * it costs a very minimum memory overhead. |
| **Disadvantages** | * it is not suitable for real life applications, due to its time complexity. * It takes more operations when input list is larger thus consuming more of the time. * for every array with n inputs it takes n2 operations to sort. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Insertion** **Sort** |  |
| **Description** | It checks element for its rightful place in list and places it there. This setup continues for each element and at each iteration. Like previous algorithms its **stable.** It has same Runtime as previous algorithms. |
| **Pseudo Code** | Insertion-Sort(Array):  n = Array.length  for j = 2 to n  min = Array[j]  // Insert Array[j] into the sorted sequence  j = i – 1  while i > 0 and Array[i] > min  Array[i+1] = Array[i]  i = i – 1  Array[j+1] = min  return Array |
| **Code** | def insertion\_sort(A):  for i in range(1, len(A)):  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  return A |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Initialization**: The subarray consists of first element and it is sorted since it is a single element.  **Maintenance**: In each iteration 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**: When it reaches last element in the array, which means entire array is sorted. As sub-Array has expanded and covered the whole array. |
| **Advantages** | * Efficient for small number of input. * Simple to understand and implement. * Efficient in system where data insertion is minimal. * Doesn’t use any additional memory. |
| **Disadvantages** | * Runtime complexity limits its uses. * Bad for large number of inputs. * Even in average case its complexity is O(n2) |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Merge Sort** |  |
| **Description** | Merge sort is quite efficient algorithm. It uses Divide & Conquer approach. It is Recursive algorithm that Divides array into sub-arrays and then after placing them into right place it merges them again into a single array thus giving the output. It’s a **stable** algorithm. It best and worst case complexity is same which leaves some loose ends. |
| **Pseudo Code** | Merge-Sort(A, left, right)  n = A.length  If n = 1 // Base Case return single element  Return A  Else // size is greater than make more partitions  mid = (left + right)/2 // find middle  Merge-Sort ( A , left , mid ) // make left and right  Merge-Sort( A , mid + 1 , right) //  Merge(A, left , mid , right)  Merge(A,low,m,high):  Right array of size m+1-low  Left array of size high-m  for i = 0 to low to m+1:  L[i] = A[i]  for j= m+1 to high+1  R[i] = A[j]  i=0  j=0  k = low  while i < L.length:  A[k] = Left[i]  i += 1  k += 1    while j < R.length:  A[k] = Right[j]  j += 1  k += 1  return A |
| **Code** | def Merge\_Sort(A, l, r):  # BASE CASE  if l < r:  m = int(l + (r-l)/2)  Merge\_Sort(A, l, m)  Merge\_Sort(A, m+1, r)  Merge(A, l, m, r)  def Merge(A, l, m, r):  global S  size\_1 = m - l + 1 # size of left array  size\_2 = r - m # size of right array  # temporary arrays  L = []  R = []  # copying elements in temporary arrays  # and checkng if element < 0  for i in range(0, size\_1):  if A[l + i] <= 0:  L.append(A[l + i])  else:  R.append(A[l + i])  for j in range(0, size\_2):  if A[m + 1 + j] <= 0:  L.append(A[m + 1 + j])  else:  R.append(A[m + 1 + j])  # indexes of temp arrays  i = 0  j = 0  k = l  # copying in first Array  while i < len(L):  A[k] = L[i]  i += 1  k += 1  while j < len(R):  A[k] = R[j]  j += 1  k += 1 |
| **Time Complexity** | Best-case time complexity: O(nlgn)  Worst-case time complexity: O(nlgn) |
| **Proof of Correctness** | **Initialization:** Subarray S is empty. L[1] is the smallest element of L, while R[1] is the smallest element of R. So it is true.  **Maintenance:** if L[i] ≤ R[j]. Then L[i] is the smallest element not yet copied to S. The current nonempty part of S consists of the k − 1 smallest elements, so after the loop is over and L[i] is copied to S, the nonempty part of S will consist of the k smallest elements. Incrementing k (in the for loop update) and i re-establishes the loop invariant for the next iteration.  **Termination:** when k = m + 1 (Condition where loop terminates). S contains the m smallest elements of L and R, in sorted order. |
| **Advantages** | * It can sort larger input runtime complexity is far more less than previous algorithms O(nlgn) which is same for best and worst cases. * It is good for real life applications. * Its execution is sequential. It doesn’t require much seeking. |
| **Disadvantages** | * Its less efficient for small number of input. * It requires memory overhead for sub-array creation. * Even if array is already sorted then it’ll take same time that makes it worse O(nlgn) |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Quick Sort** |  |
| **Description** | Quick sort uses Divide & Conquer approach like Merge sort.  It picks a pivot and partitions array around the pivot, in such a way that lesser elements are moved to left and larger are moved to right.  we can pick first, last, random, or median as pivot  partitioning the array is main procedure.  partition is done so element as pivot is at its correct position and put all smaller elements to left and put all greater elements to right  it must take linear time.  It is “**Unstable**”. |
| **Pseudo Code** | Quick-Sort(A, lowest, highest)  // base case if single element then do nothing  if lowest >= highest  return  pivotIdx = Partition(A, lowest, highest)  Quick-Sort(A, lowest, pivotIdx – 1)  Quick-Sort(A, pivotIdx + 1, highest)  Partition(A, first, last)  A[PivotIdx] = A[last]  PivotIdx = first  loop idx from first to last  if A[idx] <= A[PivotIdx]  Swap(A[idx],A[PivotIdx])  pivotIdx = PivotIdx + 1  return PivotIdx – 1 |
| **Code** | def quick\_Sort(A, l, h):  if (l < h):  pi = partition(A, l, h)  quick\_Sort(A, l, pi - 1)  quick\_Sort(A, pi + 1, h)  def partition (A, l, h):  pivot = A[h];  i = (l - 1)  for j in range(l,h):  if (A[j] < pivot):  i+=1  A[i],A[j]=A[j],A[i]  A[i + 1],A[h]=A[h],A[i + 1]  return (i + 1) |
| **Time Complexity** | Best-case time complexity: O(nlgn)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** | **Basic Step:** when l < k since it contains one or no element so it is sorted trivally.  **Induction Step:** we take middle point as pivot and then element less than pivot are in L[] and Greater are in R[] when recursion occurs further R[i] and L[i] are sorted based on pivots and further sub-Divisions. Thus at each recursion pivot is at its rightful place.  Pivot P moves almost every element to its place in rightful manner till array is sorted. |
| **Advantages** | * Its runtime is O(nlgn) which is good for many practical applications and is widely used. * It consumes minimal memory overhead. * Its faster than Merge sort in average case. |
| **Disadvantages** | * Its runtime may vary depending upon the initial array. * Worst-case (Reverse Sorted Array) may cause O(n2) runtime complexity. * It’s not stable that may question its performance issues. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Counting Sort** |  |
| **Description** | It is based on hashing technique. Every number is stored at a unique index. Output array is long as maximum number in input array. Then based on their value each number is stored at their rightful place. In case of duplicates counter is increased in respective index. In case of negative number, we first make them positive by adding the minimum number in each and then by subtracting it from the output array. All is done in linear time.  It’s a **“Stable”** algorithm. |
| **Pseudo Code** | Counting-Sort(A)  k = maximum in A  C[0..........k] // C contains k zero elements  // count each element  for j = 0 to A.length  C[A[j]] = C[A[j]] + 1  // C[i] contains position of each element  for i = 1 to k  C[i] = C[i] + C[i-1]    // array to actual shape without zero entries  for j = n-1 downto 0  B[C[A[j]]-1 ] = A[j]  C[A[j]] = C[A[j]] - 1  return B |
| **Code** | def Counting\_Sort(A):  low = min(A)  if low < 0:  low = low \* -1  for i in range(len(A)):  A[i] = A[i] + low  else:  low = 0  k = max(A)  C = [0] \* (k + 1)  out = [0] \* len(A)  for i in range(len(A)-1):  C[A[i]] += 1  for i in range(k):  C[i] += C[i - 1]  for i in range(len(A)-1, -1, -1):  j = A[i]  C[j] -= 1  out[C[j]] = A[i]  for i in range(len(out)):  out[i] = out[i] - low  return out |
| **Time Complexity** | Best-case time complexity: O(n+k)  Worst-case time complexity: O(n+k) |
| **Proof of Correctness** | **Base Case:** In the beginning C is array of length A.Max having all entries zeros such that C = [0,0,0,0…….Max] this is sorted trivially.  **Inductive Step:** in each step C[i] is increased based on presence of number. Again C is constructed in such a way that last having one is largest in array. So if C[K] has value 1 then it is largest number. Note that a random element may have more than one value but it means that duplicate (Since C is list of frequencies having largest number’s frequency in the end).  Thus when C[K+1] is initialized as 1 it means it is present in list and is the largest number. Thus, When values are initialized we get a sorted list. |
| **Advantages** | * It takes linear time which is better than other algorithms. * It is good for real time applications where time is priority. * It is best for small range key values. |
| **Disadvantages** | * It requires more additional space. * It is inefficient for greater range of key values. * It can only sort integers and alike values(Discrete values) |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Radix Sort** |  |
| **Description** | Radix sort works with digits iteratively. first it'll pick least significant digit in a number and will sort it with using some stable sort (like Insertion sort). Next run with next digit (2nd digit from left) and repeat the above procedure. It'll go on till whole array is sorted. |
| **Pseudo Code** | Radix-Sort(A, d)  digits = digits of max number in A  for i = 1 to digits  // sort A[] according to ith digit  // using insertion (Stable sort)  insertion-Sort(A, i) |
| **Code** | def RADIX\_SORT(A, digits):  digits = str(max(A))  digits = len(digits)  B = [0] \* len(A)  for i in range(0, digits):  # insertion sort to sort array A on digit i  B = insertion\_sort(A, i+1)  return B |
| **Time Complexity** | d = digits  Best-case time complexity: O(nd)  Worst-case time complexity: O(nd) |
| **Proof of Correctness** | **Base case:** when there are 1-digit number it is correct since it uses insertion sort for sorting individual digits.  **Inductive step:** Assume the numbers are sorted by their d lowest order digits. Sort using digit d+1 two numbers that have the same digit d+1 preserve their original order by stability Two numbers that differ at digit d+1 are placed in the correct order.  The process goes on and at last when sorting first digit n from the right the numbers are nearly at their correct position by sorting previous digits this generates a full sorted array.  Thus Algorithm is correct. |
| **Advantages** | * It takes linear time for sorting. * It’s quite good for long lists of medium range inputs. * It relatively requires less space than counting sort. |
| **Disadvantages** | * It can only sort integers * Its constant is larger in worst case making it inefficient. * It is less flexible and needs to be re-written for different data. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Bucket Sort** |  |
| **Description** | It divides the input array into 2D array (buckets).  Each list contains some elements. Next it sorts individual buckets using some stable algorithm (Again insertion sort)  Finally, the algorithm concatenates the buckets into single list. it simply copies the elements of each bucket into the output array.  it is “**Stable**” algorithm. |
| **Pseudo Code** | Bucket-Sort(A)  n = A.length  B[i……n-1] be the new array  for i = 0 to n-1  make B[i] another empty list at **i**  for i = 0 to n  B[n \* A[i]] = A[i]  for i = 0 to n-1  // sort B[i] using stable sort  insertion-sort(B[i])  Concatenate B[i]…..B[n] into single lists |
| **Code** | def BUCKET\_SORT(A):    n = len(A)  B =[[] for i in range(n)]    for i in range(n):  B[int(n\*A[i])].append(A[i])  print(int(n\*A[i]))    for i in range (n - 1):  B[i] = insertion\_sort(B[i])    C = list(chain.from\_iterable(B)) # concatenation for 2D list    return C |
| **Time Complexity** | k = number of buckets  Best-case time complexity: O(n + k)  Worst-case time complexity: O(n2) |
| **Proof of Correctness** | **Basic Step:** If there’s only one element in array then it is already sorted.  **Inductive Hypothesis:** Consider there are k buckets with n integers each they’ll be sorted by insertion sort so each bucket is sorted.  **Inductive step:** When dealing with K+1 buckets we first need to distribute numbers equally in each bucket which will be done when B[n \* A[i]] thus when we apply insertion sort in last list we get a sorted bucket. Which in turn gives us a 2D list on concatenation we see each bucket hold different values in sorted manner which gives off a whole sorted list. |
| **Advantages** | * Buckets can be processed independently from each other. * Array is divided into sub-arrays which leads to Divide and Conquer. * It does the job in linear time. |
| **Disadvantages** | * Bad distribution leads to extra work for no benefit. * It depends upon input array which makes it less-reliable. * For large input it costs memory overhead. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Shell Sort** |  |
| **Description** | This algorithm is kin of insertion sort but it uses gaps and minimizes the comparisons.  It first calculate gap like (k = length) k/2, k/4,....,1  then comparison starts from the beginning of the list and after gap it changes positions of elements if they are not in the right order. This continues until element on the left is smaller than current element then gap is further divided into half and process is repeated. |
| **Pseudo Code** | SHELL-SORT(A,n)  gap = N/2, N/4, N/8, ...., 1  for gap=n/2, gap=0, gap/=2  //Perform gapped insertion sort for this gap size.    for i=gap to n with increment of 1  temp=A[i]    for j=i till j>=gap and A[j-gap]>temp    A[j]= A[j-gap]  j = j - gap  // put temp in its correct location  A[j]= temp; |
| **Code** | def Shell\_Sort(A):  n = len(A)  gap = n/2  while gap > 0:  for i in range(gap,n):  Temp = A[i]  j = i  while j >= gap and A[j-gap] > Temp:  arr[j] = arr[j-gap]  j -= gap  arr[j] = Temp  gap /= 2 |
| **Time Complexity** | Best-case time complexity: O(n \* lgn)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** |  |
| **Advantages** | * Requires no extra memory. * Faster than other n-squared algorithms like bubble sort. * Effective for finite numbers in an array. |
| **Disadvantages** | * It is Unstable. * Dependent on input array that hinders its performance. * Complex and difficult for understanding. |
| **Dry Run** |  |

|  |  |
| --- | --- |
| **Cocktail Sort** |  |
| **Description** | It is just like bubble sort but a n improved version. It is “**Stable**” algorithm.  It has two stages   * Left to Right: loop starts from left. Adjacent items are compared and if left value is greater than Right value, then values are swapped. At the end of first iteration, largest number will reside at the end of the array. * Right to Left: starting from the item before sorted item, to start of the array. Here adjacent items are compared and are swapped if required. |
| **Pseudo Code** | Cocktail-Sort(A)  swap = false  start = 0  end = A.length - 1  while (swap is True):  swap = False // in case array is already sorted  for i in range(start, end):  if (A[i] > A[i + 1]):  then swap( A[ i ], A[ i+1 ] )  swap := true;  if swapped = false then break  swap = false  end = end-1  for i in range(end, start-1, -1): // downto loop  if (A[i] > A[i + 1]):  swap (A[i + 1], A[i])  swap = True  start = start + 1 |
| **Code** | def cocktailSort(A):  n = len(A)  swap = True  start = 0  end = n-1  while (swap == True):  swap = False #in case array is already sorted    # Left to Right  for i in range(start, end):  if (A[i] > A[i + 1]):  A[i], A[i + 1] = A[i + 1], A[i]  swapped = True  if (swap == False):  break  swap = False  end = end-1  for i in range(end-1, start-1, -1):  if (A[i] > A[i + 1]):  A[i], A[i + 1] = A[i + 1], A[i]  swap = True  start = start + 1 |
| **Time Complexity** | Best-case time complexity: O(n)  Worst-case time complexity: O(n²) |
| **Proof of Correctness** |  |
| **Advantages** | * It is faster than bubble sort. * It requires no additional space. * Best for small input. |
| **Disadvantages** | * No better than bubble sort when input is larger. * It is less efficient than algorithms with nlgn time. * It may cost time in order to save memory. |
| **Dry Run** | Description |