**Proposal Details**

| Group Number | 20 |
| --- | --- |
| Registration Number of Group Members | 2020-CS-23 Muhammad Rizwan  2020-CS-10 Asad Mehmood |

**Sorting algorithms Details**

|  |  |
| --- | --- |
| **Algorithm** | **Bubble sort** |
| Description | It is a type of sorting in which we repeatedly compare adjacent elements and swap them if they are in wrong order. The process is repeated until array is sorted. |
| Pseudo code | Let n = length of array  For i=0 to n-1 do  For j=0 to n-1 do  If array[j]>array[j+1] do  Swap array[j] and array[j+1] |
| Code | def **bubble\_sort**(array):      for i in **range**(**len**(array)-1):          for j in **range**(**len**(array) - 1):              if array[j] > array[j+1]:                  temp=array[j]                  array[j] = array[j+1]                  array[j+1]=temp    **print**(array) |
| Time complexity | for i in **range**(**len**(array)-1): n times    for j in **range**(**len**(array) - 1): (n)(n-1)times       if array[j] > array[j+1]: (n)(n-2) times           temp=array[j] (n)(n-2) times           array[j] = array[j+1] (n)(n-2) times           array[j+1]=temp (n)(n-2) times  The best case has time complexity : O(n)  The average case has time complexity: O(n^2)  The worst case has time complexity :O(n^2) |
| Proof of correctness |  |
| Strengths | * Easy to understand * Very short code * Takes low memory |
| Weaknesses | * Takes longer time to sort * Not suitable for larger inputs |
|  |  |
| **Algorithm** | **Merge sort** |
| Description | This type of sorting is based on divide and conquer rule. In this sorting, array is divided again and again until it remains with single element then it is sorted and combined recursively. |
| Pseudo code | Merge-sort(A,p,r):  If p<r do  Q=p+r/2  Merge-sort(A,p,q)  Merge-sort(A,q+1,r)  Merge(A,p,q,r)  Merge(A,p,q,r):  num1=q-p+1  num2=r-q  let Left[1,…,num1+1] and Right[1,….,num2+1] be new arrays  for i=1 to num1 do  Left[i]=A[p+i-1]  for j=1 to num2 do  Right[j]=A[q+j]  Left[num1+1] = ∞  Right[num2+1] = ∞  i = 1  j = 1  for k = p to r:  if Left[i]<Right[j]:  A[k]=Left[i]  i = i+1  else A[k]= = Right[j]:  j = j+1 |
| Code | import **sys**  def **merge**(arr,p,q,m):      Left=[]      Right=[]      for i in **range**(p,m+1):          Left.**append**(arr[i])      for i in **range**(m+1,q+1):          Right.**append**(arr[i])      Left.**append**(**sys**.maxsize)      Right.**append**(**sys**.maxsize)      j=0      i=0      for idx in **range**(p,q+1):          if Left[i]<=Right[j]:              arr[idx]=Left[i]              i+=1          else:              arr[idx]=Right[j]              j+=1    def **merge\_sort**(arr,p,q):      if p<q:          mid=(p+q)/2          mid=**int**(mid)  **merge\_sort**(arr,p,mid)  **merge\_sort**(arr,mid+1,q)    **merge**(arr,p,q,mid) |
| Time complexity | import **sys**  def **merge**(arr,p,q,m):      Left=[] O(1)      Right=[] O(1)      for i in **range**(p,m+1): O(n)          Left.**append**(arr[i]) O(n-1)      for i in **range**(m+1,q+1): O(n)          Right.**append**(arr[i])       O(n-1)      Left.**append**(**sys**.maxsize) O(1)      Right.**append**(**sys**.maxsize) O(1)      j=0 O(1)      i=0 O(1)      for idx in **range**(p,q+1): O(n)          if Left[i]<=Right[j]: O(n-1)              arr[idx]=Left[i] O(n-1)              i+=1 O(n-1)          else: O(n-1)              arr[idx]=Right[j] O(n-1)              j+=1 O(n-1)    def **merge\_sort**(arr,p,q):      if p<q: O(1)          mid=(p+q)/2 O(1)          mid=**int**(mid) O(1)  **merge\_sort**(arr,p,mid)  **merge\_sort**(arr,mid+1,q)    **merge**(arr,p,q,mid)  The best case has time complexity : O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity : O (n.log n) |
| Proof of correctness | **Initialization**: The loop invariant holds prior to the first iteration of the loop. Here, i = j = 1, and S is completely empty. L[1] is the smallest element of L, while R[1] is the smallest element of R, so the initialization step holds.  **Maintenance**: To see that each iteration maintains the loop invariant, suppose that 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:** At termination, k = m+ 1. By the loop invariant, S contains the m smallest elements of L and R, in sorted order. This is the result that we wanted (i.e., the merging of the two sorted arrays to produce a new sorted array) |
| Strengths | * Takes lesser time to sort * Constant time for any number of inputs * Works on divide and conquer technique |
| Weaknesses | * Takes same time even for smaller inputs * Not faster than quick sort * The code will run even if the array is already sorted |
|  |  |
| **Algorithm** | **Bucket sort** |
| Description | This type of sorting distributes every element in buckets. Then every bucket is sorted by using insertion sort algorithm. |
| Pseudo code | Let arr be the array containing unsorted elements.  Let b\_array be a new array containing 10 more arrays.  For i in arr do  index = 10 \* j  index = int(index)  b\_array[idx] = j  for i = 1 to 10 do  b\_array[i]= insertion\_sort(b\_array[i])  for i = 1 to 10 do  for j= 0 to b\_array.Length do  array[k]= b\_array[i][j]  k++ |
| Code | def **bucketSort**(array):      b\_arrays=[]      for i in **range**(10):          b\_arrays.**append**([])      for j in array:          idx = **int**(10 \* j)          b\_arrays[idx].append(j)      for i in **range**(10):          b\_arrays[i] = **insert\_sort**(b\_arrays[i])      k=0      for i in **range**(10):          for j in **range**(**len**(b\_arrays[i])):              array[k] = b\_arrays[i][j]              k += 1  **print**(array)    def **insert\_sort**(array):      for i in **range**(1, **len**(array)):          key = array[i]          j = i – 1          while j >= 0 and array[j] > key:              array[j + 1] = array[j]              j=j-1          array[j+1] = key      return array |
| Time complexity | def **bucketSort**(array):      b\_arrays=[]      for i in **range**(10):          b\_arrays.**append**([])      for j in array:          idx = **int**(10 \* j)          b\_arrays[idx].append(j)      for i in **range**(10):          b\_arrays[i] = **insert\_sort**(b\_arrays[i])      k=0      for i in **range**(10):          for j in **range**(**len**(b\_arrays[i])):              array[k] = b\_arrays[i][j]              k += 1  **print**(array)    def **insert\_sort**(array):      for i in **range**(1, **len**(array)):          key = array[i]          j = i – 1          while j >= 0 and array[j] > key:              array[j + 1] = array[j]              j=j-1          array[j+1] = key      return array  The best case has time complexity : O (n+k)  The average case has time complexity: O (n)  The worst case has time complexity : O (n^2) |
| Proof of correctness |  |
| Strengths | * We sort buckets, which are smaller parts of the original array * Its independent of the comparison approach * It is a stable sort |
| Weaknesses | * Only works for decimal values * Consumes more memory due to empty buckets * Cannot be used for words and letters * Depends on insertion sort |
|  |  |
| **Algorithm** | **Cycle sort** |
| Description | **Cycle sort** is a comparison-based sorting algorithm. It divides array into the number of cycles. Then each of them can be rotated to produce a sorted array. |
| Pseudo code | Cycle\_sort(arr):  For k= 0 to arr.Length do  temp=arr[k]  index=k  for i = k+1 to arr.Length do  if arr[i]<temp:  index = index + 1  while temp= =arr[index] do  index = index+1  if index != k do  swap temp with arr[index]  while index != k do  index= k  for i= start + 1 to arr.Length do  if arr[i] < temp do  index = index +1  while temp = = arr[index] do  index =index +1  swap temp with arr[index] |
| Code | def **cycleSort**(array):     cycle = 0     for k in **range**(0, **len**(array) - 1):          temp = array[k]          idx = k          for i in **range**(k + 1, **len**(array)):              if array[i] < temp:                  idx += 1          if idx == k:              continue          while temp == array[idx]:              idx += 1          temp2=array[idx]          array[idx]=temp          temp=temp2          cycle += 1          while idx != k:              idx = k              for i in **range**(k + 1, **len**(array)):                  if array[i] < temp:                      idx += 1              while temp == array[idx]:                  idx += 1              temp2=array[idx]              array[idx]=temp              temp=temp2              cycle += 1     return cycle |
| Time complexity | def **cycleSort**(array):     cycle = 0     for k in **range**(0, **len**(array) - 1): n times          temp = array[k] n-1 times          idx = k n-1 times          for i in **range**(k + 1, **len**(array)): ∑ i=0-n (              if array[i] < temp:                  idx += 1          if idx == k:              continue          while temp == array[idx]:              idx += 1          temp2=array[idx]          array[idx]=temp          temp=temp2          cycle += 1          while idx != k:              idx = k              for i in **range**(k + 1, **len**(array)):                  if array[i] < temp:                      idx += 1              while temp == array[idx]:                  idx += 1              temp2=array[idx]              array[idx]=temp              temp=temp2              cycle += 1     return cycle  The best case has time complexity : O (n^2)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Proof of correctness |  |
| Strengths | * It is memory efficient * It is also storage efficient |
| Weaknesses | * The time complexity is always n^2 * It is unstable sort |
|  |  |
| **Algorithm** | **Radix sort** |
| Description | Radix sort is a sorting algorithm that sorts data with according to position of digits in numbers. First the unit places are used as reference to sort numbers, then tens place and so on. Radix sort uses counting sort at last to sort an array of numbers. |
| Pseudo code | Radix\_sort(arr):  Largest number=max(arr)  m=max number of elements in largest number  Create a new array of size of arr  For i= 0 to m do  Call the counting sort  Count\_sort(arr,numb):  Largest=max(array)  Let count\_array be a new array[0,….,largest+1]  For i= 0 to n: |
| Code | def **count\_sort**(array, const):      large = **max**(array)      n = **len**(array)      final = [0] \* (n)      count\_array = [0] \* (large+1)        for i in **range**(0, n):          idx =  array[i] // const          count\_array[idx % 10] += 1        for i in **range**(1, large+1):          count\_array[i] += count\_array[i - 1]      i = n - 1        while i >= 0:          idx = array[i] // const          final[count\_array[idx % 10] - 1] = array[i]          count\_array[idx % 10] -= 1          i -= 1      i = 0      for i in **range**(0, **len**(array)):          array[i] = final[i]    def **radixSort**(array):      const = 1      max\_value = **max**(array)      while max\_value / const > 0:  **count\_sort**(array, const)          const = const \* 10 |
| Time complexity | def **count\_sort**(array, const):      large = **max**(array)      n = **len**(array)      final = [0] \* (n)      count\_array = [0] \* (large+1)        for i in **range**(0, n):          idx =  array[i] // const          count\_array[idx % 10] += 1        for i in **range**(1, large+1):          count\_array[i] += count\_array[i - 1]      i = n - 1        while i >= 0:          idx = array[i] // const          final[count\_array[idx % 10] - 1] = array[i]          count\_array[idx % 10] -= 1          i -= 1      i = 0      for i in **range**(0, **len**(array)):          array[i] = final[i]    def **radixSort**(array):      const = 1      max\_value = **max**(array)      while max\_value / const > 0:  **count\_sort**(array, const)          const = const \* 10  The best case has time complexity : O (n)  The average case has time complexity: O (nk)  The worst case has time complexity : O (nk) |
| Proof of correctness |  |
| Strengths | * It is stable sort * Faster than many algorithms |
| Weaknesses | * Takes more memory * Difficult to code * Depends on counting sort |
|  |  |
| **Algorithm** | **Insertion Sort** |
| Description | In this type of sorting, the first element in array is taken as sorted and second element is taken as ‘key’ and it is compared with every element in array to place them in their correct position. It is repeated until array is sorted. |
| Pseudo code | Insertion\_Sort(A):  for j = 2 to A.length  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 |
| Code | def insertSort(A):  n = len(A)  for i in range(1,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  #insertion sort in descending order  def insertSort(A):  n = len(A)  for i in range(1,n):  key = A[i]  j = i-1  while j>=0 and key > A[j]:  A[j+1] = A[j]  j = j-1  A[j+1] = key |
| Time complexity | Insertion\_Sort(A):  for j = 2 to A.length n  key = A[j] n-1  i = j – 1 n-1  while i > 0 and A[i] > key  A[i+1] = A[i]  i = i-1 – 1)  A[i+1] = key n - 1  The best case has time complexity : O (n)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1 : Initialization**  We begin by showing that the loop invariant holds before the first loop iteration, when j = 2. Subarray A [1 .. j-1], therefore, contains only one element A [1], which is actually the first element of A [1]. In addition, this subarray is sorted, indicating that the invariant loop holds before the first loop iteration.  **Step 2: Maintenance:**  Informally, the body of the for loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right until it finds the proper position for A[j], at which point it inserts the value of A[j]. The subarray A[j..1] then consists of the elements originally in A[j..1] , but in sorted order.  **Step 3 : Termination:**  The condition causing the for loop to terminate is that j > A:length = n. Because each loop iteration increases j by 1, we must have j = n + 1 at that time. Substituting n + 1 for j in the wording of loop invariant, we have that the subarray A[1..n] consists of the elements originally in A[1..n], but in sorted order. Observing that the subarray A[1..n] is the entire array, we conclude that the entire array is sorted. |
| Strengths | * It is a stable sort algorithm * It works fast on small input array (O(n)) |
| Weaknesses | * It does not perform well on large input array * It takes (O(n^2)) time to perform sorting on large input array |
|  |  |
| **Algorithm** | **Selection sort** |
| Description | In this type of sorting, we repeatedly search for smallest element in array and locate it in the beginning of array. |
| Pseudo code | Selection\_sort(A):  for i=0 to A.length:  minvalue\_index = i  for j in range(i + 1,n):  if A[j] < A[minvalue\_index]  minvalue\_index = j  swap (min\_element, first unsorted position)  end Selection\_sort |
| Code | def selectionSort(A):  n = len(A)  for i in range(n-1):  minValueIndex = i  for j in range(i + 1,n):  if A[j] < A[minValueIndex] :  minValueIndex = j  if minValueIndex != i :  temp = A[i]  A[i] = A[minValueIndex]  A[minValueIndex] = temp  return A  # selection sort in descending order  def selectionSort(A):  n = len(A)  for i in range(n-1):  minValueIndex = i  for j in range(i + 1,n):  if A[j] > A[minValueIndex] :  minValueIndex = j  if minValueIndex != i :  temp = A[i]  A[i] = A[minValueIndex]  A[minValueIndex] = temp  return A |
| Time complexity | def selectionSort(A):  n = len(A) 1  for i = 1 to n-1 : n  minValueIndex = i n - 1  for j = i + 1 to n:  if A[j] > A[minValueIndex] :  minValueIndex = j  if minValueIndex != i : n - 1  temp = A[i] n - 1  A[i] = A[minValueIndex] n - 1  A[minValueIndex] = temp n - 1  The best case has time complexity : O (n^2)  The average case has time complexity: O (n^2)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1: 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 3 of the code sets minValueindex = 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.  **Step 2: Maintenance**  Before pass j, we assume that min indexes the smallest element in the subarray A[i..j-1]. During iteration A[j] ≥ A[min],the if statement is not true, so nothing is executed. But now min indexes the smallest element of A[i..j]. Line 6 switches min to index this new location and hence after the loop iteration finishes, min indexes the smallest element in subarray A[i..j].  **Step 3 : 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. |
| Strengths | * It performs well on small size arrays. * This algorithm does not require a lot of space for sorting. Only one extra variable is used to store temp value. |
| Weaknesses | * It does not perform well on large size array. * It take O(n^2) time to perform sorting. |
|  |  |
| **Algorithm** | **Quick sort** |
| Description | Quick sort is just like Merge sort as it also uses divide and conquer rule. In this algorithm, we first select pivot and on the basis of pivot, we divide smaller and larger elements to their correct side. |
| Pseudo code | def quickSort(arr[], low, high)  if (low < high)  pi = partition(arr, low, high);  quickSort(arr, low, pi - 1)  quickSort(arr, pi + 1, high)  def partition (arr[], low, high)  pivot = arr[high];  i = (low - 1)  for j = low to (high – 1)  if (arr[j] < pivot)  i++  swap arr[i] and arr[j]  swap arr[i + 1] and arr[high])  return (i + 1) |
| Code | def quick\_sort(A,p,r):  if p < r:  pi = partition(A,p,r)  quick\_sort(A,p,pi-1)  quick\_sort(A,pi+1,r)    def partition(A,p,r):  x = A[right]  i = p - 1  for j = p to r:  if (A[j] < x):  i +=1  temp = A[i]  A[i] = A[j]  A[j] = temp  temp\_2 = A[i+1]  A[i+1] = A[r]  A[r] = temp\_2  return (i+1) |
| Time complexity | def quick\_sort(A,p,r):  if p < r:  pi = partition(A,p,r)  quick\_sort(A,p,pi-1)  quick\_sort(A,pi+1,r)    def partition(A,p,r):  x = A[right]  i = p - 1  for j = p to r:  if (A[j] < x):  i +=1  temp = A[i]  A[i] = A[j]  A[j] = temp  temp\_2 = A[i+1]  A[i+1] = A[r]  A[r] = temp\_2  return (i+1)  The best case has time complexity : O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1: Initialization:**  Prior to the first iteration of the loop, i = p – 1 and j = p. Because no values lie between p and i and no values lie between i + 1 and j - 1, the first two conditions of the loop invariant are trivially satisfied.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . Because of the swap, we now have that A[i] < x, and condition 1 is satisfied. Similarly, we also have that A[j – 1] > x, since the item that was swapped into A[j – 1] is, by the loop invariant, greater than x.  **Step 3 : Termination:**  At termination, j == r. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x. |
| Strengths | * It is also in-place algorithm. * It works very well on short sized arrays |
| Weaknesses | * It is not a stable sort algorithm * It takes much time on large size arrays which is O(n^2) |
|  |  |
| **Algorithm** | **Counting sort** |
| Description | This sorting works by iterating through elements, counting the occurrence of every element present in array and use these counts to compute an element index in the final sorted array. |
| Pseudo code | Counting\_sort(A,B,k)  Let C[0 .. k] be a new array  for i = 0 to k  C[i] = 0  for j = 1 to A.length  C[A[j]] +=1  for i = 1 to k  C[i] += C[i-1]  for j = A.length **downto** 1  B[C[A[j]]] = A[j]  C[A[j]] -= 1 |
| Code | def countSort(A):  ans = []  smallest = min(A)  smallest = abs(smallest)  for i in range(0,len(A)):  A[i] += smallest  largest = max(A)  output = [0] \* len(A)  count = [0] \* (largest+1)  for i in range(0,len(A)):  j = A[i]  count[j] +=1  for i in range(1,largest+1):  count[i] += count[i-1]  for i in reversed(range(0,len(A))):  j = A[i]  count[j] -=1  output[count[j]] = A[i]  output[count[j]] -= smallest  print(output) |
| Time complexity | def countSort(A):  ans = [] 1  smallest = min(A) 1  smallest = abs(smallest) 1  for i in range(0,len(A)): n+1  A[i] += smallest n  largest = max(A) 1  output = [0] \* len(A) 1  count = [0] \* (largest+1) 1  for i in range(0,len(A)): n+1  j = A[i] n  count[j] +=1 n  for i in range(1,largest+1): n+1  count[i] += count[i-1] n  for i in reversed(range(0,len(A))): n+1  j = A[i] n  count[j] -=1 n  output[count[j]] = A[i] n  output[count[j]] -= smallest n  print(output)1  The best case has time complexity : O (n + k)  The average case has time complexity: O (n + k)  The worst case has time complexity : O (n + k) |
| Proof of correctness | Loop invariant  **Step 1: Initialization:**  Prior to the first iteration of the loop, i = p – 1 and j = p. Because no values lie between p and i and no values lie between i + 1 and j - 1, the first two conditions of the loop invariant are trivially satisfied.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . Because of the swap, we now have that A[i] < x, and condition 1 is satisfied. Similarly, we also have that A[j – 1] > x, since the item that was swapped into A[j – 1] is, by the loop invariant, greater than x.  **Step 3 : Termination:**  At termination, j == r. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x. |
| Strengths | * It is a stable sort algorithm * Since it is not a comparison based algorithm |
| Weaknesses | * If range of input value is much larger then this algorithm requires a very large space. |
|  |  |
| **Algorithm** | **Shell sort** |
| Description | It is type of sorting which first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. The interval between the elements is reduced based on the sequence used. |
| Pseudo code | def shellSort()  A : array of items  interval = A.length // 2  while interval > 0 do:    for i = interval to n:  valueToInsert = A[i]  inner = i;  while inner >= interval && A[inner - interval] > valueToInsert do:  A[inner] = A[inner - interval]  inner = inner - interval  end while  A[inner] = valueToInsert  interval = interval //2; |
| Code | def shellSort(array, n):  interval = n // 2  while interval > 0:  for i in range(interval, n):  temp = array[i]  j = i  while j >= interval and array[j - interval] > temp:  array[j] = array[j - interval]  j -= interval  array[j] = temp  interval //= 2 |
| Time complexity | def shellSort(array, n):  interval = n // 2  while interval > 0:  for i in range(interval, n):  temp = array[i]  j = i  while j >= interval and array[j - interval] > temp:  array[j] = array[j - interval]  j -= interval  array[j] = temp  interval //= 2  The best case has time complexity : O (n.log n)  The average case has time complexity: O (n.log n)  The worst case has time complexity : O (n^2) |
| Proof of correctness | Loop invariant  **Step 1: Initialization:**  Prior to the first iteration of the loop, i = p – 1 and j = p. Because no values lie between p and i and no values lie between i + 1 and j - 1, the first two conditions of the loop invariant are trivially satisfied.  **Step 2: Maintenance:**  We consider two cases, depending on the outcome of the test. When A[j] > x; the only action in the loop is to increment j . After j is incremented, condition 2 holds for A[j – 1] and all other entries remain unchanged. When A[j] < x; the loop increments i, swaps A[i] and A[j] , and then increments j . Because of the swap, we now have that A[i] < x, and condition 1 is satisfied. Similarly, we also have that A[j – 1] > x, since the item that was swapped into A[j – 1] is, by the loop invariant, greater than x.  **Step 3 : Termination:**  At termination, j == r. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x. |
| Strengths | * It is a stable sort algorithm * Since it is not a comparison-based algorithm |
| Weaknesses | * If range of input value is much larger then this algorithm requires a very large space. |
|  |  |
| **Algorithm** | **Heap Sort** |
| Description | It is also a comparison based sorting algorithm. It is improved version of selection sort in which we first divide array and find largest element from unsorted array and put it in sorted array. |
| Pseudo Code | heap\_Sort(A):  A.heap-size = A:length  for i = [A.length/2] downto 1  MAX-HEAPIFY(A,i)  MAX-HEAPIFY(A,i)  l = left(i)  r = right(i)  if l <= A.heap-size and A[l] > A[i]  largest = l  else  largest = i  if r <= A.heap-size and A[r] > A[largest]  largest = r  if largest != i  exchange A[i] with A[largest]  MAX-HEAPIFY(A,largest) |
| Code | def heapify(arr, n, i):  largest = i  left = 2 \* i + 1  right = 2 \* i + 2  if left < n and arr[i] < arr[l]:  largest = left  if right < n and arr[largest] < arr[r]:  largest = right  if largest != i:  arr[i],arr[largest] = arr[largest],arr[i]  heapify(arr, n, largest)  def heapSort(arr):  n = len(arr)  for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)  for i in range(n-1, 0, -1):  arr[i], arr[0] = arr[0], arr[i]  heapify(arr, i, 0) |
| Time Complexity | def heapify(arr, n, i):  largest = i 1  left = 2 \* i + 1 1  right = 2 \* i + 2 1  if left < n and arr[i] < arr[l]: 1  largest = left 1  if right < n and arr[largest] < arr[r]: 1  largest = right 1  if largest != i: 1  arr[i],arr[largest] = arr[largest],arr[i] 1  heapify(arr, n, largest) 1  def heapSort(arr):  n = len(arr) 1  for i in range(n // 2 - 1, -1, -1): (n/2)+1  heapify(arr, n, i) (n/2)  for i in range(n-1, 0, -1): (n)  arr[i], arr[0] = arr[0], arr[i] n-1  heapify(arr, i, 0) n-1 |
| Proof of correctness | **Initialization:** Prior to the first iteration of the loop, i = [n/2]. Each node [n/2] + 1; [n/2]+2 … n is a leaf and is thus the root of a trivial max-heap.  **Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A,i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes i + 1; i + 2,…, n are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.  **Termination:** At termination, i = 0. By the loop invariant, each node 1,2, …. , n is the root of a max-heap. In particular, node 1 is. |
| Strengths | It takes nlgn time for any input size array. |
| Weakness | It is not a stable sort algorithm |