|  |  |
| --- | --- |
| Group Number | 43 |
| Registration Number of Group Members | 2020-CS-31  2020-CS-44 |
| Algorithm’s Name | Merge Sort |
| Description | Merge sort divides the input array into two arrays and does it recursively until their size is 1 and then compares every index with each other for sorting. |
| Pseudo code of Algorithm | Merge(A, p, q, r)  n1 = q - p + 1  n2 = r - q  let L[1…n1+1] and R[1…n2+1] be new arrays  **for** i = 1 **to** n1  L[i] = A[p + i - 1]  **for**  j = 1 **to** n2  R[j] = A[q + j]  L[n1 + 1] = ∞  R[n2 + 1] = ∞  i = 1  j = 1  **for** k = p **to** r  **if** L[i] <= R[j]  A[k] = L[i]  **else** A[k] = R[j]  j = j + 1 |
| Python Code | def merge(self, A, start, mid, end):            index1 = mid - start + 1          index2 = end - mid            left = []          right= []            for i in range(0,index1):              left.append(A[start + i])          for j in range(0,index2):              right.append(A[mid + j + 1])            left.append(math.inf)          right.append(math.inf)            i = 0          j = 0            for k  in range(start,end+1):              if left[i] <= right[j]:                  A[k] = left[i]                  i += 1              else:                  A[k] = right[j]                  j += 1        def merge\_sort(self, A, start, end):          if start < end:              mid = int((start + end)/2)              self.merge\_sort(A, start, mid)              self.merge\_sort(A, mid+1, end)              self.merge(A, start, mid, end) |
| Time Complexity Analysis | Time Complexity of merge sort is **O(n\*lg(n))** in all worst average and best cases. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we have k=p, so the subarray A is empty. This subarray contains the smallest elements of k-p=0, since i and j both = 0 therefore L[i] and R[j] both are smallest element of the array which have not yeh copied in A.  **Maintenance:** To check that each iteration maintains the loop invariant, we assume that L[i] <= R[j], then L[i] is the smallest element not yet copied. After that the subarray A will contain the smallest elements. After the incrementing of the loop it reestablishes the loop invariant.  **Termination:** As for the loop invariant termination will be at k=r+1, after that the subarray A contains the largest elements of both L and R merged arrays. |
| Three Strengths | * It is Faster for larger inputs. * It uses a consistent running time. * It uses a divide and conquer principle which splits the array into half. |
| Three Weakness | * Slower with smaller inputs. * It goes through the whole procedure even when the list is almost sorted. * Uses more memory. |
| Dry Run | Unsorted Array: [7, 6, 5, 65, 34]  ------------  Partitioned: 7 6  Right: [6, inf]  Left: [7, inf]  Merged: [6, 7, 5, 65, 34]  ------------  Partitioned: 6 7 5  Right: [5, inf]  Left: [6, 7, inf]  Merged: [5, 6, 7, 65, 34]  ------------  Partitioned: 65 34  Right: [34, inf]  Left: [65, inf]  Merged: [5, 6, 7, 34, 65]  ------------  Partitioned: 5 6 7 34 65  Right: [34, 65, inf]  Left: [5, 6, 7, inf]  Merged: [5, 6, 7, 34, 65]  ------------  Sorted Array: [5, 6, 7, 34, 65] |

|  |  |
| --- | --- |
| Algorithm’s Name | Insertion Sort |
| Description | Insertion sort starts from a single index and then goes to next indexes comparing adjacent indexes and sorting. |
| Pseudo code of Algorithm | 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  End while  A[j+1] ← key  End for |
| Python Code | def perform\_sorting(self, array: List):          for idx in range(1,len(array)):              key = array[idx]              i = idx-1              while i >= 0 and array[i] > key:                  array[i+1] = array[i]                  i = i - 1              array[i+1] = key          return array |
| Time Complexity Analysis | Time Complexity of insertion sort is **O(n2)** for worst and average cases and for best case it is **O(n)**. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop the subarray A will have unsorted elements of the array, the loop invariant is satisfied at the beginning of the loop.  **Maintenance:** If the loop invariant is true at start, then body of loop works by checking the next element of the array A[j-1], A[j-2], A[j-3] and compares it with the rest of the array that it has sorted and then increment itself to the next element.  **Termination:** As for the loop invariant termination will be at j>n, after that the subarray A will be sorted. |
| Three Strengths | * Simple to understand. * It performs well for small input. * It takes very less space. |
| Three Weakness | * It is slower with time of O(n2). * It performs bad for large inputs. * It is a lot slower for reverse sorted input. |
| Dry Run | Unsorted Array: [6, 0, 20, 5]  ------------  Index changed before loop: 0  Index changed in loop: -1  Sorted array till now: [6, 6, 20, 5]  ------------  Key: 0  Index changed before loop: 1  Key: 20  Index changed before loop: 2  Index changed in loop: 1  Sorted array till now: [0, 6, 20, 20]  ------------  Index changed in loop: 0  Sorted array till now: [0, 6, 6, 20]  ------------  Key: 5  ------------  Sorted Array: [0, 5, 6, 20] |

|  |  |
| --- | --- |
| Algorithm’s Name | Selection Sort |
| Description | Selection sorts an array by finding the minimum (or maximum) value and putting it in the start of the new array. |
| Pseudo code of Algorithm | for i = 1 to n - 1  min = i  for j = i+1 to n  if list[j] < list[min] then  min = j;  end if  end for  if indexMin != i then  swap list[min] and list[i]  end if  end for |
| Python Code | def perform\_sorting(self, array: List):          for outer in range(0, len(array)):              min = outer              for inner in range(outer+1, len(array)):                  if array[inner] < array[min]:                          min= inner                temp = array[outer]              array[outer] = array[min]              array[min] = temp          return array |
| Time Complexity Analysis | Time Complexity of Insertion Sort is **O(n2)** for all best, worst and average cases. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, the subarray A will have unsorted elements of the array, the line will set min to 1, also the loop invariant will be satisfied at the beginning of the loop.  **Maintenance:** If the loop invariant is true at start, then body of loop works by checking the condition and storing the smallest element of the array at A[j] and then increment itself to the next element.  **Termination:** As for the loop invariant termination will be at j=n+1, after that the subarray A will be sorted. |
| Three Strengths | * It performs well for small inputs. * No additional temporary storage is required. * Arrangements of data does not affect its performance. |
| Three Weakness | * It is slower in time with O(n2). * It doesn’t perform good with large inputs. * It will take the same time even if the array is almost sorted. |
| Dry Run | Unsorted Array: [11, 10, 8, -1]  ------------  Supposed minimum: 0  Actual minimum: 1  Actual minimum: 2  Actual minimum: 3  Sorted array till now: [-1, 10, 8, 11]  Supposed minimum: 1  Actual minimum: 2  Sorted array till now: [-1, 8, 10, 11]  Supposed minimum: 2  Sorted array till now: [-1, 8, 10, 11]  Supposed minimum: 3  Sorted array till now: [-1, 8, 10, 11]  ------------  Sorted Array: [-1, 8, 10, 11] |

|  |  |
| --- | --- |
| Algorithm’s Name | Bubble Sort |
| Description | Bubble sort works by adjacently swapping the adjacent elements by checking the condition. |
| Pseudo code of Algorithm | bubbleSort( list : array of items )  loop = list.count;  for i = 0 to loop-1 do:  swapped = false  for j = 0 to loop-1 do:  if list[j] > list[j+1] then  swap( list[j], list[j+1] )  swapped = true  end if |
| Python Code | def perform\_sorting(self, array: List):          n = len(array)          for outer in range(0,n):              for inner in range(0,n-outer-1):                  if(array[inner] > array[inner + 1]):                      temp = array[inner]                      array[inner] = array[inner+1]                      array[inner+1] = temp          return array |
| Time Complexity Analysis | Time Complexity of Bubble Sort is **O(n2)** for worst and average cases and **O(n)** for best case. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, the array will have unsorted elements of the array and the loop invariant will be satisfied at the beginning of the loop.  **Maintenance:** If the loop invariant is true at start, then body of loop checks the adjacent elements of the array array[i], array[i+1] and swapping them if the condition is true array[i]>array[i+1], and then increment itself to the next element.  **Termination:** As for the loop invariant termination will be at i=A.length, after that the array will be sorted. |
| Three Strengths | * Easy to understand. * Easy to implement. * Doesn’t take large amount of memory. |
| Three Weakness | * It is slower with time of O(n2). * It doesn’t perform well with large inputs. * Adjacent swapping makes it a lot slower for reverse sorted input. |
| Dry Run | Unsorted Array: [11, 52, 8, -1]  ------------  Alternative one Index: 0  Alternative two Index: 0  Alternative two Index: 1  Alternative two Index: 2  Sorted array till now: [11, 8, -1, 52]  Alternative one Index: 1  Alternative two Index: 0  Alternative two Index: 1  Sorted array till now: [8, -1, 11, 52]  Alternative one Index: 2  Alternative two Index: 0  Sorted array till now: [-1, 8, 11, 52]  Alternative one Index: 3  Sorted array till now: [-1, 8, 11, 52]  ------------  Sorted Array: [-1, 8, 11, 52] |

|  |  |
| --- | --- |
| Algorithm’s Name | Quick Sort |
| Description | Quick sort chooses an element as pivot and then compares the element, the element which is smaller than pivot goes to its left and vice versa, and does this recursively. |
| Pseudo code of Algorithm | 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) |
| Python Code | def Partition(self, array, low, high):          pivot = array[high]          i = low - 1          for idx in range(low, high):              if array[idx] < pivot:                  i+=1                  temp = array[i]                  array[i] = array[idx]                  array[idx] = temp            temp = array[i+1]          array[i+1] = array[high]          array[high] = temp          return i+1        def quick\_sort(self, array, low, high):          if low < high:              pivot = self.Partition(array, low, high)              self.quickSort(array, low, pivot - 1)              self.quickSort(array, pivot + 1, high) |
| Time Complexity Analysis | Time Complexity of Quick Sort is **O(n2)** for worst case and **O(n\*lg(n))** for best and average case. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, i = p-1 and j = p with this the first two conditions of loop invariant will be satisfied.  **Maintenance:** If the loop invariant is true at start, then j is incremented and according to the pivot the array is formed and the elements will be swapped according arr[j] < pivot and by using recursion it is maintained.  **Termination:** As for the loop invariant termination will be at j = r, after that the array will be sorted. |
| Three Strengths | * It is efficient. * No additional storage is required. * It performs well for large inputs. |
| Three Weakness | * It is not stable. * It worst case is slower with time complexity of O(n2). * It is recursive. |
| Dry Run | Unsorted Array: [13, 22, 8, 14]  ------------  Pivot: 14  Sorted array till now: [13, 22, 8, 14]  Sorted array till now: [13, 8, 22, 14]  Sorted array till now: [13, 8, 14, 22]  Pivot: 8  Sorted array till now: [8, 13, 14, 22]  ------------  Sorted Array: [8, 13, 14, 22] |

|  |  |
| --- | --- |
| Algorithm’s Name | Heap Sort |
| Description | Heap sort selects the largest element of the array from the unsorted side and then puts it in the sorted array. |
| Pseudo code of Algorithm | Heapsort(A)     BuildHeap(A)     for i <- length(A) downto 2 {        exchange A[1] <-> A[i]        heapsize <- heapsize -1        Heapify(A, 1)      BuildHeap(A)     heapsize <- length(A)     for i <- floor( length/2 ) downto 1        Heapify(A, i)      Heapify(A, i)     le <- left(i)     ri <- right(i)     if (le<=heapsize) and (A[le]>A[i])        largest <- le     else        largest <- i     if (ri<=heapsize) and (A[ri]>A[largest])        largest <- ri     if (largest != i) {        exchange A[i] <-> A[largest]        Heapify(A, largest) |
| Python Code | def heapify(self, array, n, i):          max\_num = i          left = 2 \* i + 1          right = 2 \* i + 2            if left < n and array[i] < array[left]:              max\_num = left              if right < n and array[max\_num] < array[right]:                  max\_num = right                if max\_num != i:                  array[i], array[max\_num] = array[max\_num], array[i]                  self.heapify(array, n, max\_num)        def heapSort(self, array):          n = len(array)          for idx in range(n // 2 - 1, -1, -1):              self.heapify(array, n, idx)              for i in range(n-1, 0, -1):                  array[i], array[0] = array[0], array[i]                  self.heapify(array, i, 0) |
| Time Complexity Analysis | Time Complexity of Heap sort is **O(n\*lg(n))** in all worst average and best cases. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop each node i + 1, i + 2, … n, everything is leaf so it is a heap, so loop invariant will be satisfied.  **Maintenance:** If the loop invariant is true at start, subtrees at children of node I are mac heaps, decrementing i reestablishes the loop invariant for next iteration.  **Termination:** As for the loop invariant termination will be at i = 0, after that the array will be sorted. |
| Three Strengths | * It is efficient. * Its memory usage is minimal. * It is simple to understand. |
| Three Weakness | * It’s worse case comes in O(n\*lg(n)). * It is unstable sort. * Memory management is complex. |
| Dry Run | Unsorted Array: [13, 22, 66, 14]  ------------  Sorted array till now: [13, 22, 66, 14]  Sorted array till now: [66, 22, 14, 13]  Sorted array till now: [66, 22, 14, 13]  Sorted array till now: [22, 14, 66, 13]  Sorted array till now: [22, 14, 66, 13]  Sorted array till now: [14, 22, 66, 13]  Sorted array till now: [66, 22, 14, 13]  Sorted array till now: [66, 22, 14, 13]  Sorted array till now: [22, 13, 14, 66]  Sorted array till now: [22, 13, 14, 66]  Sorted array till now: [14, 13, 22, 66]  Sorted array till now: [13, 14, 22, 66]  ------------  Sorted Array: [13, 14, 22, 66] |

|  |  |
| --- | --- |
| Algorithm’s Name | Counting Sort |
| Description | In counting sort, we create an array to the highest number found in the array, after that we again check the array and which variable is found in that array in incremented. |
| Pseudo code of Algorithm | CountingSort(input)  k = range of elements of array  count ← array of k + 1 zeros  output ← array of same length as input  for i = 0 to length(input) - 1 do  j = key(input[i])  count[j] += 1  for i = 1 to k do  count[i] += count[i - 1]  for i = length(input) - 1 down to 0 do  j = key(input[i])  count[j] -= 1  output[count[j]] = input[i]  return output |
| Python Code | def perform\_sorting(self, array: List):          max\_var =  max(array)          min\_var =  min(array)          key = (max\_var - min\_var) + 1            counts = []          output = []            for idx in range(0 , key):              counts.append(0)            for i in range(0 ,  len(array)):              output.append(0)            for j in range(0 , len(array)):              k = self.find\_key(array[j] , min\_var)              counts[k] += 1            for a in range(1 , key):              counts[a] += counts[a-1]            for i in range(len(array) - 1, -1, -1):              j = self.find\_key(array[i] , min\_var)              counts[j] -= 1              output[counts[j]] = array[i]            return output        def find\_key(self, element, min):          key = (min \* -1) + element          return key |
| Time Complexity Analysis | Time Complexity of Counting sort is **O(n+k)** for best and averaged case and **O(n2)** for worst case. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we assume an array with the range of its highest to lowest element.  **Maintenance:** Assume the loop invariant is true at start, we create an array with index of range from the highest to lowest element with C[i] = 0 after that we increment the index after we compare from the original array and increment the indexes j = key(input[i]), count[j] += 1 and then start a loop while decrementing the array C from that array and copying the elements in the new array.  **Termination:** As for the loop invariant termination will be at i = 0, after that the array will be sorted. |
| Three Strengths | * It is a fast sorting algorithm with time complexity of O(n). * It is a stable sort. * It is good for inputs in which the range of array is not too large. |
| Three Weakness | * It can take a lot of space for many inputs. * It works bad when the difference of element of input array is quite large. * It cannot be used for string inputs. |
| Dry Run | Unsorted Array: [3, 8, 4, 11]  -----------------------------  Counts array: [1, 1, 0, 0, 0, 1, 0, 0, 1]  Traversed Counts array [1, 2, 2, 2, 2, 3, 3, 3, 4]  Sorted array till now: [3, 4, 8, 11]  Counts array after every number sort: [0, 1, 2, 2, 2, 2, 3, 3, 3]  -----------------------------  Sorted Array: [3, 4, 8, 11] |

|  |  |
| --- | --- |
| Algorithm’s Name | Radix Sort |
| Description | Radix sort uses digit to digit sorting starting from least significant to most significant digit and then sorts the array. |
| Pseudo code of Algorithm | Radix-Sort(A, d)      for j = 1 to d do              int count[10] = {0};              for i = 0 to n do                  count[key of(A[i]) in pass j]++              for k = 1 to 10 do                  count[k] = count[k] + count[k-1]              for i = n-1 downto 0 do                  result[ count[key of(A[i])] ] = A[j]                  count[key of(A[i])]--           for i=0 to n do                  A[i] = result[i]      end for(j)   end func |
| Python Code | def countingSort(self, array, place):          n = len(array)          counts = []          output = []            for idx in range(0 , 10):              counts.append(0)            for i in range(0 ,  len(array)):              output.append(0)            for i in range(0, n):              idx = array[i] // place              counts[idx % 10] += 1            for i in range(1, 10):              counts[i] += counts[i - 1]            for i in range(len(array) - 1, -1, -1):              index = array[i] // place              output[counts[index % 10] - 1] = array[i]              counts[index % 10] -= 1            for i in range(0, n):              array[i] = output[i]        def radixSort(self, array):          max\_elem = max(array)          place = 1          while max\_elem // place > 0:              self.countingSort(array, place)              place = place \* 10 |
| Time Complexity Analysis | Time Complexity for radix sort is **O(n)** for best and average cases and **O(n2)** for worst cases. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we assume every digit as independent and sorts them independently.  **Maintenance:** Assume the loop invariant is true at start, we create an array and first compare them with their least significant digit and store it into the new array and then move it through by incrementing to most significant digit.  **Termination:** As for the loop invariant termination will be at i = d, after that the array will be sorted. |
| Three Strengths | * It is a stable sort. * It is quite fast with time complexity of O(n). * It is fast when the range of array is less. |
| Three Weakness | * It is quite slower for worst case. * It is very less flexible. * It takes more space. |
| Dry Run | Unsorted Array: [6, 15, 4, 7]  -----------------------------  Counts array: [0, 0, 0, 0, 1, 1, 1, 1, 0, 0]  Traversed Counts array [0, 0, 0, 0, 1, 2, 3, 4, 4, 4]  Sorted array till now: [4, 15, 6, 7]  Counts array after every number sort: [0, 0, 0, 0, 0, 1, 2, 3, 4, 4]  -----------------------------  Counts array: [3, 1, 0, 0, 0, 0, 0, 0, 0, 0]  Traversed Counts array [3, 4, 4, 4, 4, 4, 4, 4, 4, 4]  Sorted array till now: [4, 6, 7, 15]  Counts array after every number sort: [0, 3, 4, 4, 4, 4, 4, 4, 4, 4]  Sorted Array: [4, 6, 7, 15] |

|  |  |
| --- | --- |
| Algorithm’s Name | Bucket Sort |
| Description | Bucket sort distributes the elements of the array into a number of buckets and then uses a different sorting algorithm to sort each bucket. |
| Pseudo code of Algorithm | function bucketSort(array, k) is  buckets ← new array of k empty lists  M ← the maximum key value in the array  for i = 1 to length(array) do       insert *array[i]* into *buckets[floor(k × array[i]/M)]*  for i = 1 to k do       nextSort(buckets[i])  return the concatenation of buckets[1], ...., buckets[k] |
| Python Code | def perform\_sorting(self, array: List):          bucket = []          for idx in range(0,10):              bucket.append([])            for element in array:              index = math.floor(element\*10)              bucket[index].append(element)              for bucketIDX in range(0 , len(bucket)):              bucket[bucketIDX] = InsertionSort.insertion\_sort(bucket[bucketIDX])            original\_idx = 0          for bucketidx in range(0 , len(bucket)):              for idx in range(0 , len(bucket[bucketidx])):                  array[original\_idx] = bucket[bucketidx][idx]                  original\_idx += 1          return array |
| Time Complexity Analysis | Time Complexity for Bucket sort is **O(n)** for best and average case and **O(n2)** for worst case**.** |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we assume arrays as buckets.  **Maintenance:** Assume the loop invariant is true at start, we create an array with indexes from 0 to 9 after that we put the values by using their most significant bit into those assumed arrays and sort those arrays using any different sorting algorithms and then sort accordingly.  **Termination:** As for the loop invariant termination will be at i = n-1, after that the array will be sorted. |
| Three Strengths | * It is efficient. * It is stable. * It takes less memory. |
| Three Weakness | * It is quite slower for worst case. * It doesn’t work good for array with close range elements. * It is less flexible. |
| Dry Run | Unsorted Array: [0.667, 0.529, 0.566, 0.424, 0.205, 0.3124]  Bucket: []  Bucket: []  Bucket: [0.205]  Bucket: [0.3124]  Bucket: [0.424]  Bucket: [0.529, 0.566]  Bucket: [0.667]  Bucket: []  Bucket: []  Bucket: []  Buckets: [0.205, 0.529, 0.566, 0.424, 0.205, 0.3124]  Buckets: [0.205, 0.3124, 0.566, 0.424, 0.205, 0.3124]  Buckets: [0.205, 0.3124, 0.424, 0.424, 0.205, 0.3124]  Buckets: [0.205, 0.3124, 0.424, 0.529, 0.205, 0.3124]  Buckets: [0.205, 0.3124, 0.424, 0.529, 0.566, 0.3124]  Buckets: [0.205, 0.3124, 0.424, 0.529, 0.566, 0.667]  Sorted Array: [0.205, 0.3124, 0.424, 0.529, 0.566, 0.667] |

|  |  |
| --- | --- |
| Algorithm’s Name | Gnome Sort |
| Description | Gnome sort works with a single element at a time and then places it to its proper place by a number of swaps. |
| Pseudo code of Algorithm | procedure optimizedGnomeSort(a[]):  for pos in 1 to length(a):       gnomeSort(a, pos)    procedure gnomeSort(a[], upperBound):  pos := upperBound  while pos > 0 and a[pos-1] > a[pos]:       swap a[pos-1] and a[pos]       pos := pos - 1 |
| Python Code | def perform\_sorting(self, array: List):          n = len(array)          idx = 0          while idx < n:              if idx == 0:                  idx +=1              elif array[idx] >= array[idx-1]:                  idx +=1              else:                  array[idx] , array[idx - 1] = array[idx- 1] , array[idx]                  idx -=1          return array |
| Time Complexity Analysis | Time Complexity for gnome sort is **O(n2)** for all worst and average cases but **O(n)** for best case. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we assume the elements by putting it to its right place by swapping.  **Maintenance:** Assume the loop invariant is true at start, we check the elements by their index with each other and by putting it to the its right place. After the incrementing of the loop, it reestablishes the loop invariant.  **Termination:** As for the loop invariant termination will happen when index < n is false, after that the array will be sorted. |
| Three Strengths | * It is easy to understand. * It is stable. * It simply moves by swapping the adjacent elements. |
| Three Weakness | * It worst and average case time is quite slow. * It is similar to bubble sort. * It is less flexible. |
| Dry Run | Unsorted Array: [9, 1, 0, 3]  Index: 0  Index: 1  Sorted array till now: [9, 1, 0, 3]  -----------------------------  Index: 0  Sorted array till now: [1, 9, 0, 3]  -----------------------------  Index: 1  Sorted array till now: [1, 9, 0, 3]  -----------------------------  Index: 2  Sorted array till now: [1, 9, 0, 3]  -----------------------------  Index: 1  Sorted array till now: [1, 0, 9, 3]  -----------------------------  Index: 0  Sorted array till now: [0, 1, 9, 3]  -----------------------------  Index: 1  Sorted array till now: [0, 1, 9, 3]  -----------------------------  Index: 2  Sorted array till now: [0, 1, 9, 3]  -----------------------------  Index: 3  Sorted array till now: [0, 1, 9, 3]  -----------------------------  Index: 2  Sorted array till now: [0, 1, 3, 9]  -----------------------------  Index: 3  Sorted array till now: [0, 1, 3, 9]  -----------------------------  Index: 4  Sorted array till now: [0, 1, 3, 9]  -----------------------------  Sorted Array: [0, 1, 3, 9] |

|  |  |
| --- | --- |
| Algorithm’s Name | Shell Sort |
| Description | Shell sort sorts the elements far from each other, then progressively reduces the gap between the elements to be compared. |
| Pseudo code of Algorithm | procedure 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        end while     A[inner] = valueToInsert     end for     interval = (interval -1) /3;     end while  end procedure |
| Python Code | def perform\_sorting(self, array: List):          gap = int(len(array)/2)          while gap > 0:              i = 0              inner = gap              while inner < len(array):                  if array[i] > array[inner]:                      array[i], array[inner] = array[inner], array[i]                    i += 1                  inner += 1                    idx = i                  while idx - gap  > -1:                      if array[idx - gap] > array[idx]:                          array[idx - gap], array[idx] = array[idx], array[idx-gap]                      idx -= 1              gap = int(gap / 2)          return array |
| Time Complexity Analysis | Time complexity of shell sort is **O(n\*lg(n))** for best and average case and **O(n2)** for worst case. |
| Proof of Correctness | **Initialization:** Before the first iteration of the loop, we assume the elements which are far from each other and then reduces dynamically.  **Maintenance:** Assume the loop invariant is true at start, we check the elements by their indexes which and sort those which are far from each other k - gap > -1 by using this condition we swap the values which are not in the right order.  **Termination:** As for the loop invariant termination will happen when gap <= 0, after that the array will be sorted. |
| Three Strengths | * It does not require extra space. * It is only efficient for finite number of elements. * It average case is also quite faster. |
| Three Weakness | * It is complex and difficult to understand. * It is not stable. * It worst case scenario is quite slow. |
| Dry Run | Unsorted Array: [13, 22, 8, 14]  ------------  Gap: 2  Sorted array till now: [8, 22, 13, 14]  Sorted array till now: [8, 14, 13, 22]  Sorted array till now: [8, 14, 13, 22]  Gap: 1  Sorted array till now: [8, 14, 13, 22]  Sorted array till now: [8, 14, 13, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Sorted array till now: [8, 13, 14, 22]  Gap: 0  ------------  Sorted Array: [8, 13, 14, 22] |