* LAPTOPSTER



Sorting Algorithms

Project Supervisor

Mr. Samyan Qayyum Wahla

Project Members(Group 41)

|  |  |
| --- | --- |
| Member 1 Esha Tur Razia | Member 1 2020-CS-71. |
| Member 2 Sana Shafique | Member 2 2020-CS-84. |

Department of Computer Science

University of Engineering and Technology, Lahore Pakistan

|  |  |  |
| --- | --- | --- |
| Quick Sort | | |
| |  | | --- | | Pseudo code |   quickSort(list, small, large)  if (small < large)  p = partition(list, small, large)  quickSort(list, small, p - 1);  /\* prior p \*/  quickSort(list, p + 1, large);  /\* after the p \*/  partition (list, small, large)  pivotPoint = list[large]  i = (small - 1)  for (j = small; j <= large- 1; j++)  if (list[j] < pivotPoint)  i++  swap list[i] and list[j]  swap list[i + 1] and list[large]  return (i + 1) | | |
| Time complexity | Big-O(n log(n)) | |
| |  | | --- | | Description |   In this, we randomly select a pivot point and array is divided on the basis of that pivot and then sort it. It works on the principle of divide and conquer. | | |
|  | | |
| Strength | | |
| * Its time complexity is big-o(n log(n)) * Its faster than merge sort * Additional memory is not required | | |
| Weakness | | |
| * Not stable * Depending on number of elements of array running time differs * If array is sorted running time might degrades | | |
| Proof of correctness | | |
| Picking a pivot value randomly can help to avoid worst case scenarios like (perfectly sorted array). | | |

|  |
| --- |
| Code in python |
| def quickSort(arr,small,large)  if len(arr) == 1  return arr  if low < high  p = partition(arr, small, large)  quickSort(arr, small, p - 1)  quickSort(arr, p + 1, large)  else  p = 0  def partition(arr, small, large)  i = (small - 1)  pivot\_point = arr[large]  for j in range(small,large)  if arr[j] <= pivot  i = i + 1  arr[i], arr[j] = arr[j], arr[i]  arr[i+1], arr[high] = arr[high], arr[i+1]  return (i + 1)  if \_\_name\_\_ == '\_\_main\_\_':  arr = [4, 1, 3, 9, 7, 8]  n = len(arr)  quickSort(arr, 0, n-1)  print("Sorted array is:")  for i in range(n):  print("%d" arr[i])n"), |

|  |  |  |
| --- | --- | --- |
| Merge Sort | | |
| |  | | --- | | Pseudo code |   def mergeSort( arr )  if ( n == 1 )  return arr  array1 = arr[0] to arr[n/2]  array2 = arr[n/2+1] to arr[n]  array1 = mergeSort( array1 )  array2 = mergeSort( array2 )  return merging( array1, array2 )  def merging( arr\_a, arr\_b )  arr\_c  while ( arr\_a and arr\_b contains members )  if ( arr\_a[0] > arr\_b[0] )  adding arr\_b[0] at the ending of arr\_c  then remove arr\_b[0] from arr\_b  else  adding arr\_a[0] at the ending of arr\_c  then remove arr\_a[0] from arr\_a  while ( arr\_a contains members )  adding arr\_a[0] at the ending of arr\_c  then remove arr\_a[0] from arr\_a  while ( arr\_b contains members )  adding arr\_b[0] at the ending of arr\_c  then remove arr\_b[0] from arr\_b  return arr\_c | | |
| Time complexity | Big-O(n log(n)) | |
| |  | | --- | | Description |   In this we simply split the given array into two parts and then sort them recursively until they are sorted and then merge them at the end. | | |
|  | | |
| Strength | | |
| * Merge sort is much faster than straight forward methods of sorting * Stable algorithm * It’s a recursive algorithm that calls itself over and over | | |
| Weakness | | |
| * Sub elements use more space * Slower than others | | |
| Proof of correctness | | |
| It requires big-o(n) space on arrays and linked lists, it demands a constant space. | | |

|  |
| --- |
| Code in python |
| def Merge\_Sort(arr):  first = []  second = []  if len(arr) > 1:  mid = len(arr)//2  first = arr[:mid]  second = arr[mid:]  Merge\_Sort(first)  Merge\_Sort(second)  i = 0  j = 0  k = 0  while i < len(first) and j < len(second):  if first[i] < second[j]:  arr[k] = first[i]  i = i + 1  else:  arr[k] = second[j]  j = j + 1  k = k + 1      while i < len(first):  arr[k] = first[i]  i = i + 1  k = k + 1    while j < len(second):  arr[k] = second[j]  j = j + 1  k = k + 1  def print\_List(arr):  for i in range(len(arr)):  print(arr[i], end=" ")        if \_\_name\_\_ == '\_\_main\_\_':  arr = [4, 1, 3, 9, 7]  print("Given array is", end="\n")  print\_List(arr)  Merge\_Sort(arr)  print("Sorted array is: ", end="\n")  print\_List(arr) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Code in python | | | | | |
| Bubble Sort | | |
| |  | | --- | | Pseudo code |   def BubbleSort( arr )  looping = arr.count;  for i = 0 to looping-1  do:  swap = false  for j = 0 to looping-1  do:  /\* check elements which are adjacent to each other and compare them\*/  if arr[j] > arr[j+1] then  swap( arr[j], arr[j+1] )  swap = true  if(! swapped)  break  return list | | |
| Time complexity | Big-O(n) | |
| |  | | --- | | Description |   In this, we simply compares the consecutive elements from the given array and then swap them until they are sorted. | | |
|  | | |
| Strength | | |
| * Use less space * Few lines of code | | |
| Weakness | | |
| * Complexity is big-o(n^2) * Take much time for sorting | | |
| Proof of correctness | | |
| As it takes a large time, it is not a preferred sorting algorithm. | | |

|  |
| --- |
| Code in python |
| def bubble\_Sort(arr):  for i in range(len(arr) - 1):  min = i  for j in range( i + 1, len(arr)):  if(arr[j] < arr[min]):  min = j  if(min != i):  arr[i], arr[min] = arr[min], arr[i]  return arr  if \_\_name\_\_ == '\_\_main\_\_':  arr = [5, 8, 9, 3, 5, 7, 1, 3, 4, 9, 3, 5, 1, 8, 4]  print("Given array: ")  bubble\_Sort(arr)  printArr(arr) |

|  |  |  |
| --- | --- | --- |
| Insertion Sort | | |
| |  | | --- | | Pseudo code |   for i = 1 to n  key = Arr [i]  j = i – 1  while j > = 0 and Arr[j] > key  Arr[j+1] = Arr[j]  j = j – 1  Arr[j+1] = key | | |
| Time complexity | Big-O(n) | |
| |  | | --- | | Description |   In this we split the input in two parts one is sorted and the other is unsorted part then take any element from the unsorted part and replace it at the desired position. | | |
|  | | |
| Strength | | |
| * Efficient * Take less usage of space | | |
| Weakness | | |
| * It takes time despite array being sorted * Fails on huge sets of information | | |
| Proof of correctness | | |
| In the best case scenario(already sorted) every insert requires a constant time. | | |

|  |
| --- |
| Code in python |
| def Insertion\_sort(arr):  # for loop iteration in python from j to the length of array  j = 1  for j in range(len(arr)):  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    def print\_List(arr):  for i in range(len(arr)):  print(arr[i], end=" ")    arr = [5, 7, -8, 9, 10, 4, -7, 0, -12, 1, 6, 2, 3, -4, -15, 12]  print("Given array is", end="\n")  print\_List(arr)  Insertion\_sort(arr)  print("Sorted array is: ", end="\n")  print\_List(arr) |

|  |  |  |
| --- | --- | --- |
| Selection Sort | | |
| |  | | --- | | Pseudo code |   def selection (arr)  for i = 1 to n - 1  /\* current element will be converted to min\*/  min = i  /\* checking if the element is minimum \*/  for j = i+1 to n  if arr[j] < arr[min]  then min = j;  if index of min != i  then swapping arr[min] and arr[i] | | |
| Time complexity | Big-O(n^2) | |
| |  | | --- | | Description |   In this smallest element is selected from the initially given input and then it is swapped with the left most element and the process is repeated until the sorted array is formed. | | |
|  | | |
| Strength | | |
| * Data arrangement is not affected * Fewer operations are used | | |
| Weakness | | |
| * Large amount of time is required | | |
| Proof of correctness | | |
| Even if the array is already sorted it requires time to scan the entire array. | | |

|  |
| --- |
| Code in python |
| def Selection\_Sort(arr):  for i in range(len(arr) - 1):  min = i  for j in range( i + 1, len(arr)):  if(arr[j] < arr[min]):  min = j  if(min != i):  arr[i], arr[min] = arr[min], arr[i]  return arr  if \_\_name\_\_ == '\_\_main\_\_':  arr = [5, 8, 9, 3, 5, 7, 1, 3, 4, 9, 3, 5, 1, 8, 4]  print("Given array: ")  Selection\_Sort(arr)  printArr(arr) |

|  |  |  |
| --- | --- | --- |
| Count Sort | | |
| |  | | --- | | Pseudo code |   def CountingSort(arr)  for i = 0 to length(arr) - 1  do  j = key(arr[i])  num[j] += 1  for i = 1 to k  do  num[i] += num[i - 1]  for i = length(arr) - 1 down till zero  do  j = key(arr[i])  num[j] -= 1  result[num[j]] = arr[i]  return result | | |
| Time complexity | Big-O(n+k) | |
| |  | | --- | | Description |   This works by counting presence of each element in given input. | | |
|  | | |
| Strength | | |
| * Less time complexity | | |
| Weakness | | |
| * For large range of values it requires more space | | |
| Proof of correctness | | |
| Non negative key values have range of k. | | |

|  |
| --- |
| Code in python |
| import random,math  def getKey(n):  return n  def countingSort(list,k,getKey):  count = [0]\*k  for n in list  count[getKey(n)] = count[getKey(n)] + 1  for i in range(k)  if i ==0:  count[i] = count[i]  else:  count[i] += count[i-1]  output = [None]\*len(list)  for i in range(len(list)-1, -1, -1)  sortkey = getKey(list[i])  output[count[sortkey]-1] = list[i]  count[sortkey] -=1  return output  random.seed(0)  arr = [random.randint(0,20) for n in range(10)]  print("Unsorted array")  print(arr)  print("\nSorted array using basic counting sort")  output = countingSort(arr, max(arr) +1, getKey)  print(output) |

|  |  |  |
| --- | --- | --- |
| Heap Sort | | |
| |  | | --- | | Pseudo code |   HeapSort(arr)  buildHeap(arr)  for i < len(arr) down till 2 {  interchange arr[h] with arr[i]  heapLen < heapLen h  Heapify(arr, h)  BuildHeap(arr)  heapLen < len(arr)  for i < floor( len/2 ) down till 1  Heapify(arr, i)  Heapify(arr, i)  left <- left(i)  right <- right(i)  if (left <= heapSize) and (arr[left] > arr[i])  highest <- left  else  highest <- i  if (right <= heapLen) and (arr[right] > arr[highest])  highest <- right  if (highest != i) {  exchange arr[i] with arr[highest]  Heapify(arr, highest) | | |
| Time complexity | Big-O(n+k) | |
| |  | | --- | | Description |   In this algorithm, we sort the given input by placing the minimum element at the start and repeat the process until the whole array is sorted it is similar to selection sort algorithm | | |
|  | | |
| Strength | | |
| * Simple to understand * Efficient * Use less memory | | |
| Weakness | | |
| * Uses big-o(1) memory space * Worst case with nLog(n) | | |
| Proof of correctness | | |
| Heap sort is stable if the underlying sorting algorithm is stable. | | |

|  |
| --- |
| Code in python |
| def heapify(arr, n, i):      higher = i      left = 2 \* i + 1  right = 2 \* i + 2      if left < n and arr[i] < arr[l]:          higher = left        if right < n and arr[highest] < arr[right]:          highest = right        if highest != i:          arr[i],arr[highest] = arr[highest],arr[i]            heapify(arr, n, highest)    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)    arr = [ 12, 11, 13, 5, 6, 7]  heapSort(arr)  n = len(arr)  print ("Sorted array is")  for i in range(n):      print ("%d" %arr[i]), |

|  |  |  |
| --- | --- | --- |
| Radix Sort | | |
| |  | | --- | | Pseudo code |   radixSort(arr, pass) // p is the number of passes   for j = 1 to pass  do  int countThearray[10] = {0}   for i = 0 to n  do  countThearray[key of(arr[i]) in pass j]++   for k = 1 to 10 do    countThearray[r] = countThearray[r] + countThearray[r-1]   for i = n-1 down to 0  do  resultOfarray[ countThearray[key of(arr[i])] ] = arr[j]  countThearray[r]  countThearray[key of(arr[i])]   for i=0 to n  do  arr[i] = resultOfarray[i] | | |
| Time complexity | Big-O(nk) | |
| |  | | --- | | Description |   By using counting sort it sorts elements on the basis of significance of the elements i.e from least significant to most significant. | | |
|  | | |
| Strength | | |
| * When range of elements is small then fast * Use dc3 algorithm * Stable | | |
| Weakness | | |
| * Less flexible * Depends on digits or letters * More space | | |
| Proof of correctness | | |
| Radix sort is stable if the underlying sorting algorithm is stable. | | |

|  |
| --- |
| Code in python |
| def countingSort(arr, e1):  n = len(arr)  output = [0] \* (n)  count = [0] \* (10)  for i in range(0,n):  index = (arr[i]/e1)  count[int((index)%10)] ++  for i in range(1,10):  count[i] += count[i-1]  i = n-1  while i>=0:  index = (arr[i]/e1)  output[ count[ int((index)%10) ] - 1] = arr[i]  count[int((index)%10)] --  i --  i = 0  for i in range(0,len(arr)):  arr[i] = output[i]  def radixSort(arr):  max1 = max(arr)  while max1/exp > 0  countingSort(arr,exp)  exp \*= 10\n  arr = [110, 45, 65,50, 90,602, 24, 2, 66]  radixSort(arr)  for i in range(len(arr))  print(arr[i]) |

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| Linear Search | | |
| |  | | --- | | Pseudo code |   def linearSearch(arr)  while x < arr.length  do  if(arr[x] == r]  return x  else  x = x + 1  return -1 | | |
| Time complexity | Big-O(n) | |
| |  | | --- | | Description |   It simply compares the element in a specific sequence until the element is found or matched. | | |
|  | | |
| Strength | | |
| * Small and medium can be searched quickly * List does not required to be sorted * Not affected by deleting and inserting | | |
| Weakness | | |
| * Time consuming for large arrays * Linear search algorithm is worst case * Less efficient | | |
| Proof of correctness | | |
| * It searches the element by sequentially comparing the array elements. | | |

|  |
| --- |
| Code in python |
| def linearSearch(arr, n, k):   for i in range(0, n):          if (arr[i] == k):              return i   return -1      arr = [1 ,3, 5, 4, 7, 9]  k = 7    n = len(arr)  ans = linear\_Search(arr, n, k)  if(ans == -1):      print("Element not found")  else:      print("Element found at index: ", ans) |