13/5/20

Data Structures - Merge Sort Algorithm

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –

14	33	27	10	35	19	42	44

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

14	33	27	10

35	19	42	44

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

14	33

27	10

35	19

42	44

We further divide these arrays and we achieve atomic value which can no more be divided.

14	33	27	10	35	19	42	44

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

14	33

10	27

19	35

42	44

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all i a sorted order.

10	14	27	33

19	35	42	44

After the final merging, the list should look like this –

10	14	19	27	33	35	42	44

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Step 1 - if it is only one element in the list it is already sorted, return.
Step 2 - divide the list recursively into two halves until it can no more be divided.
Step 3 - merge the smaller lists into new list in sorted order.
```

...all now see the pseudocodes for merge sort functions. As our algorithms point out two main functions – ...ge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

[redacted]

 13/5/20

Data Structure and Algorithms - Quick Sort

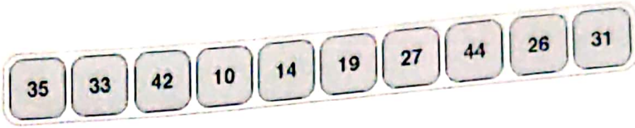
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n^2), respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.

Unsorted Array



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

```
Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot
```

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

13/5/20

ck Sort Algorithm

sing pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

```
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively
```

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

[redacted]