Dive In

Learn

Quick Sort
And
Merge Sort

Easy and quick understanding

01

QUICK SORT

01

Quick Sort:

QuickSort is a sorting algorithm based on the divide-and-conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Let's look into its algorithm in next page:

Algorithm of the quick sort

```
Quick_sort(arr,start,end)
{
    If(start<end)
    {
        Piv_pos=partition(arr,start,end)
        Quick_sort(arr,start,Piv_pos-1)
                    //used to sort the left side of the array
         Quick_sort(arr,Piv_pos+1,end)
                    //used to sort the right side of the array
    }
}
```

Partition algorithm

```
Partition(arr,start,end)
{
   i=start+1;
   pivot=arr[start]
   for(int j=start+1;j<=end;j++)
   {

        //Rearrange the array by putting element which are
    less than the pivot on one side and which are greater than to
                                                the other side.
      if(arr[j]<pivot)
      {
        swap(arr[i],arr[j]);
         i++;
      }
   }
   swap(arr[start],arr[i-1]);
   return i-1;
}
```

How does the Quick sort algorithm works..?

The Quick Sort algorithm is a divide-and-conquer sorting technique.

Here's a step-by-step breakdown.

Choose a Pivot:
A pivot element is chosen from the array. In your description, the first element is used as the pivot.

Partitioning:
The array is rearranged so that all elements less than the pivot come before it and all elements greater than the pivot come after it. The partition function does this and returns the final position of the pivot.

Recursive Sorting:

The array is then divided into two sub-arrays

The left sub-array contains elements less than the pivot.
The right sub-array contains elements greater than the pivot.
The Quick Sort function is called recursively on these two sub-arrays.

Base Case:

The recursion stops when the size of the array is 0 or 1, as these are inherently sorted.

Corrected Explanation of the Partition Function
Here's a clearer description of the partition function.

Initialization

Start with i initialized to start + 1 and choose pivot as arr[start].

Rearranging Elements:

Loop through the elements from start + 1 to end. For each element arr[j]

If arr[j] is less than the pivot, swap arr[i] and arr[j], then increment i by 1.

Final Position of Pivot: After the loop, swap the pivot element (arr[start]) with arr[i - 1]. This places the pivot in its correct sorted position.

Return the Pivot Index:

Finally, return i - 1, which is the index of the pivot.

Complete Algorithm Overview

Here's a revised version of your explanation

The Quick Sort algorithm works by selecting a pivot element (in this case, the first element) and partitioning the array around this pivot.

The partitioning process rearranges the array such that all elements less than the pivot are on the left and all elements greater than the pivot are on the right.

The algorithm recursively applies the same process to the left and right sub-arrays. The recursion continues until the base case is reached, where the sub-array has a length of 0 or 1, meaning it's already sorted.

By continually partitioning and sorting, Quick Sort efficiently sorts the entire array.

Complexity

Best Time Complexity:

$O(n \log n)$

Worst Time Complexity:

$O(n^2)$

when the smallest or largest element is always chosen as the pivot, leading to unbalanced partitions

Space Complexity:

O(log n) due to recursive calls.

02

MERGE SORT

02

Merge Sort:

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Let's look into its algorithm in next page:

Algorithm for dividing

```
mergesort(arr,lb,ub)
{
  if(lb<ub)
  {
    mid=(lb+ub)/2;
    mergesort(arr,lb,mid);
    mergesort(arr,mid+1,ub);
    merge(arr,lb,mid,ub);
  }
}
```

Algorithm for merging

```
void merge(int a[], int beg, int mid, int end)
{

    n1 = mid - beg + 1;
     n2 = end - mid;

    LeftArray[n1], RightArray[n2];
                                        //temporary arrays

                                    /* copy data to temp arrays */
   for (int i = 0; i < n1; i++)
   LeftArray[i] = a[beg + i];
  for (int j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0;                       /* initial index of first sub-array */
   j = 0;                       /* initial index of second sub-array */
   k = beg;                     /* initial index of merged sub-array */
```

```
while (i < n1 && j < n2)
 {
    if(LeftArray[i] <= RightArray[j])
     {
          a[k] = LeftArray[i];
         i++;
     }
      else
      {
         a[k] = RightArray[j];
         j++;
      }
      k++;
   }
   while (i<n1)
   {
      a[k] = LeftArray[i];
      i++;
      k++;
   }

   while (j<n2)
   {
      a[k] = RightArray[j];
      j++;
      k++;
   }
}
```

Working of Merge sort Algorithm

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

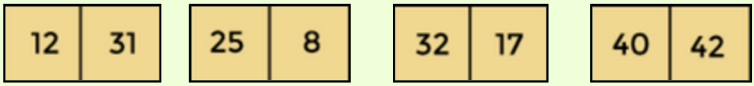
Let the elements of array are,

12	31	25	8	32	17	40	42

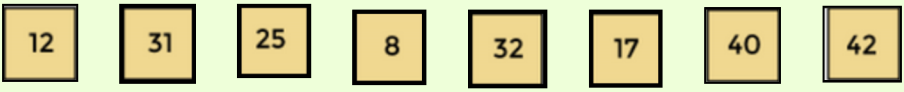
According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



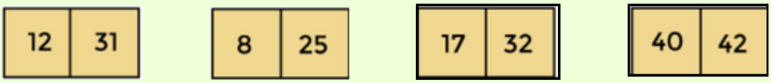
Now, again divide these arrays to get the atomic value that cannot be further divided.



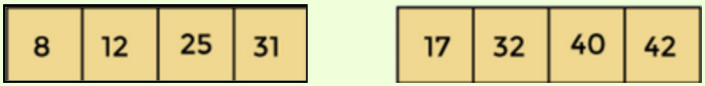
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

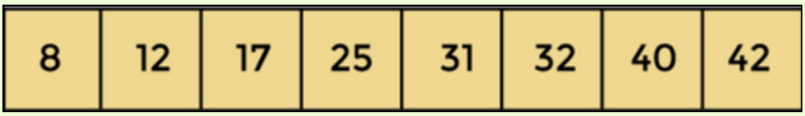
So, first compare 12 and 31, both are in sorted positions.
Then compare 25 and 8, and in the list of two values, put 8 first followed by 25.
Then compare 32 and 17, sort them and put 17 first followed by 32.
After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays.
After the final merging of above arrays,
the array will look like



Now, the array is completely sorted.

Complexity

Best Time Complexity:

O(n log n)

Worst Time Complexity:

O(n log n)

Space Complexity:

O(log n) due to recursive calls.

HOPE YOU ENJOYED LEARNING

Credit goes to

SANGAMESH S_23S028
RAGUL RAJ T_23S026
SURYA PRAKASH S_23DS036
KAUSHIKRAM K S_23DS014

....