Data Structures and Algorithms (CEN3016)

Dr. Muhammad Umair Khan

Assistant Professor

Department of Computer Engineering

National University of Technology

Merge Sort

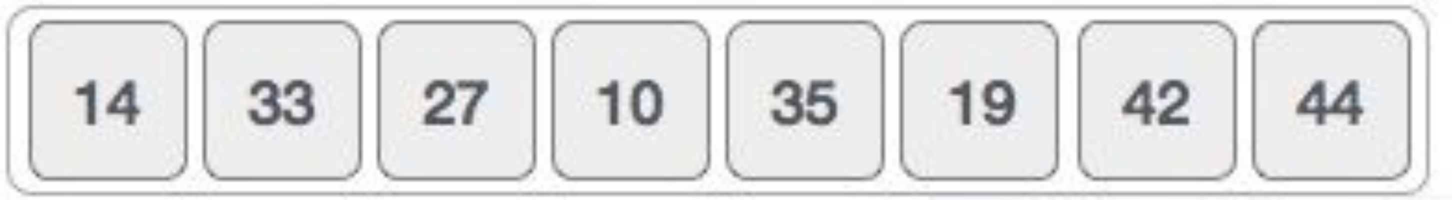
Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

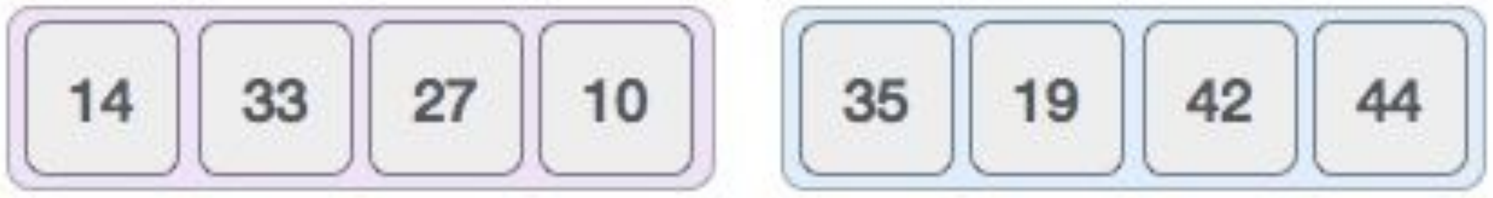
Merge Sort

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

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

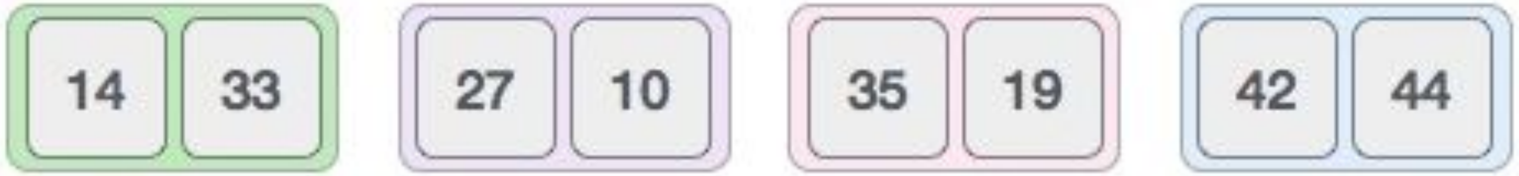


We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

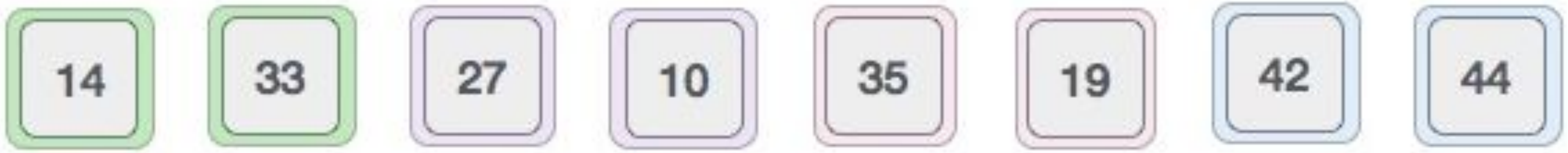


Merge Sort

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



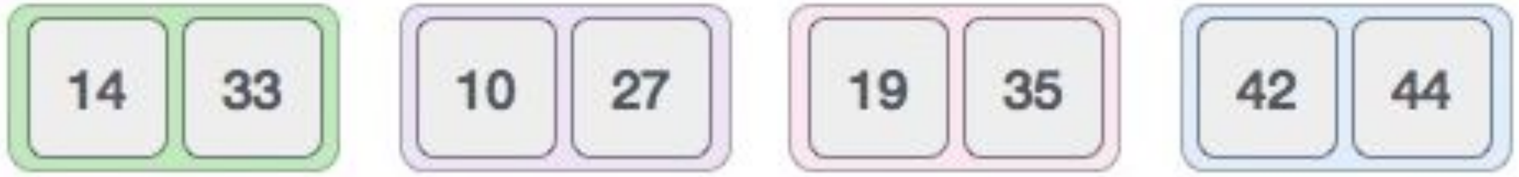
We further divide these arrays and we achieve atomic value which can no more be divided.



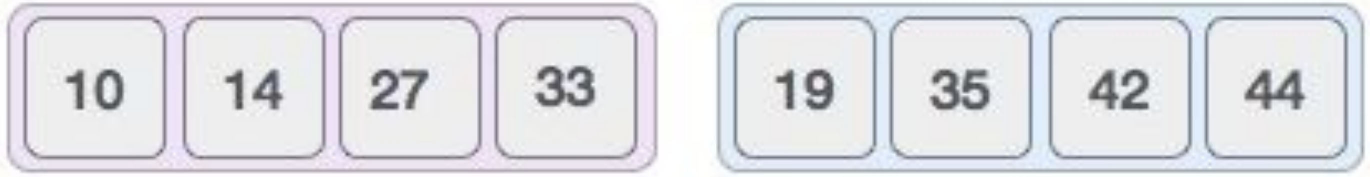
Now, we combine them in exactly the same manner as they were broken down.

Merge Sort

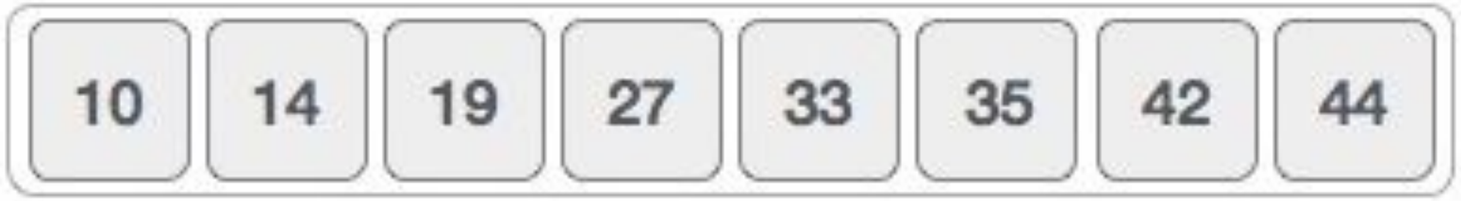
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed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



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 in a sorted order.



After the final merging, the list should look like this

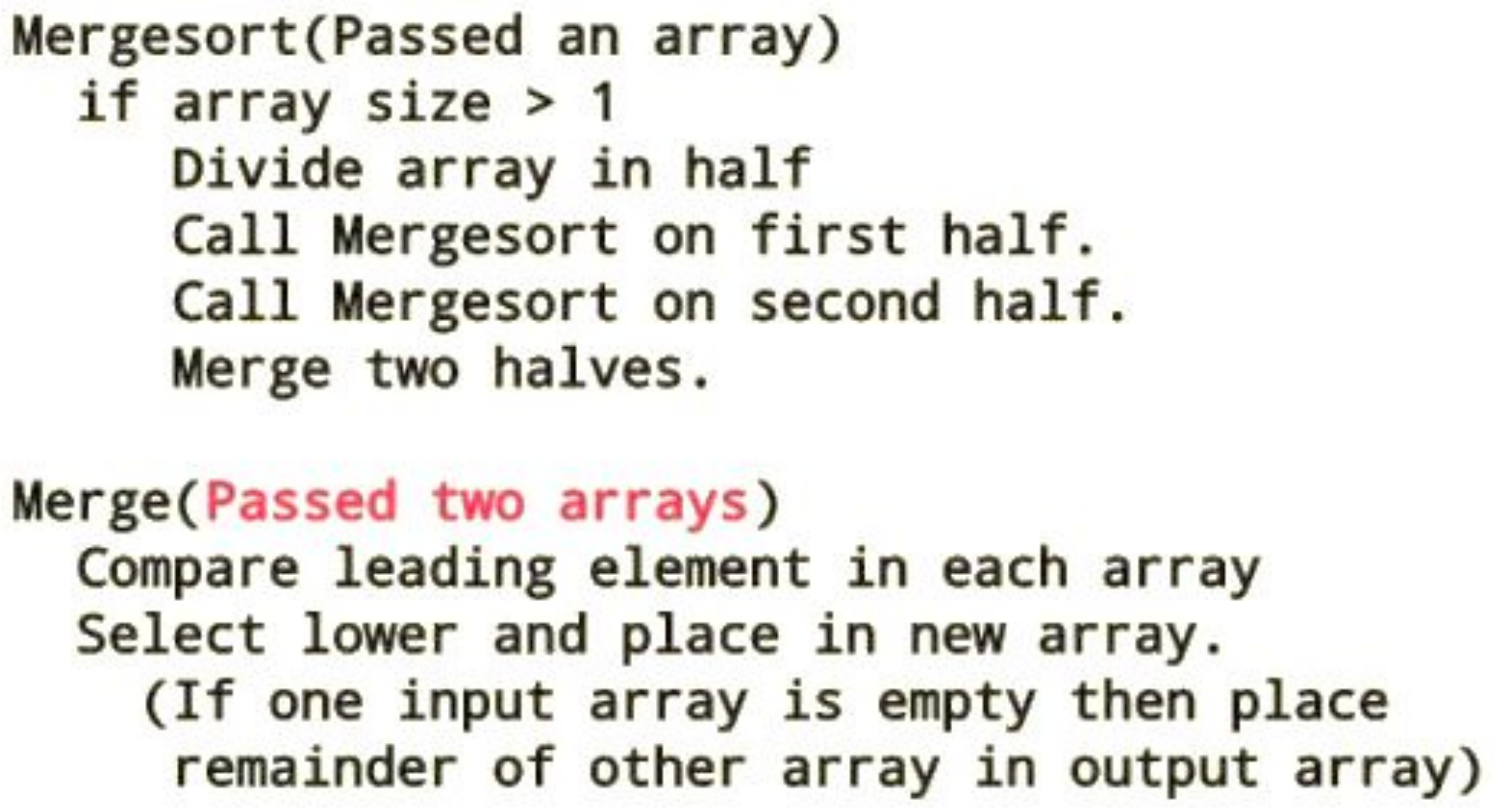


Algorithm

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, it is 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.

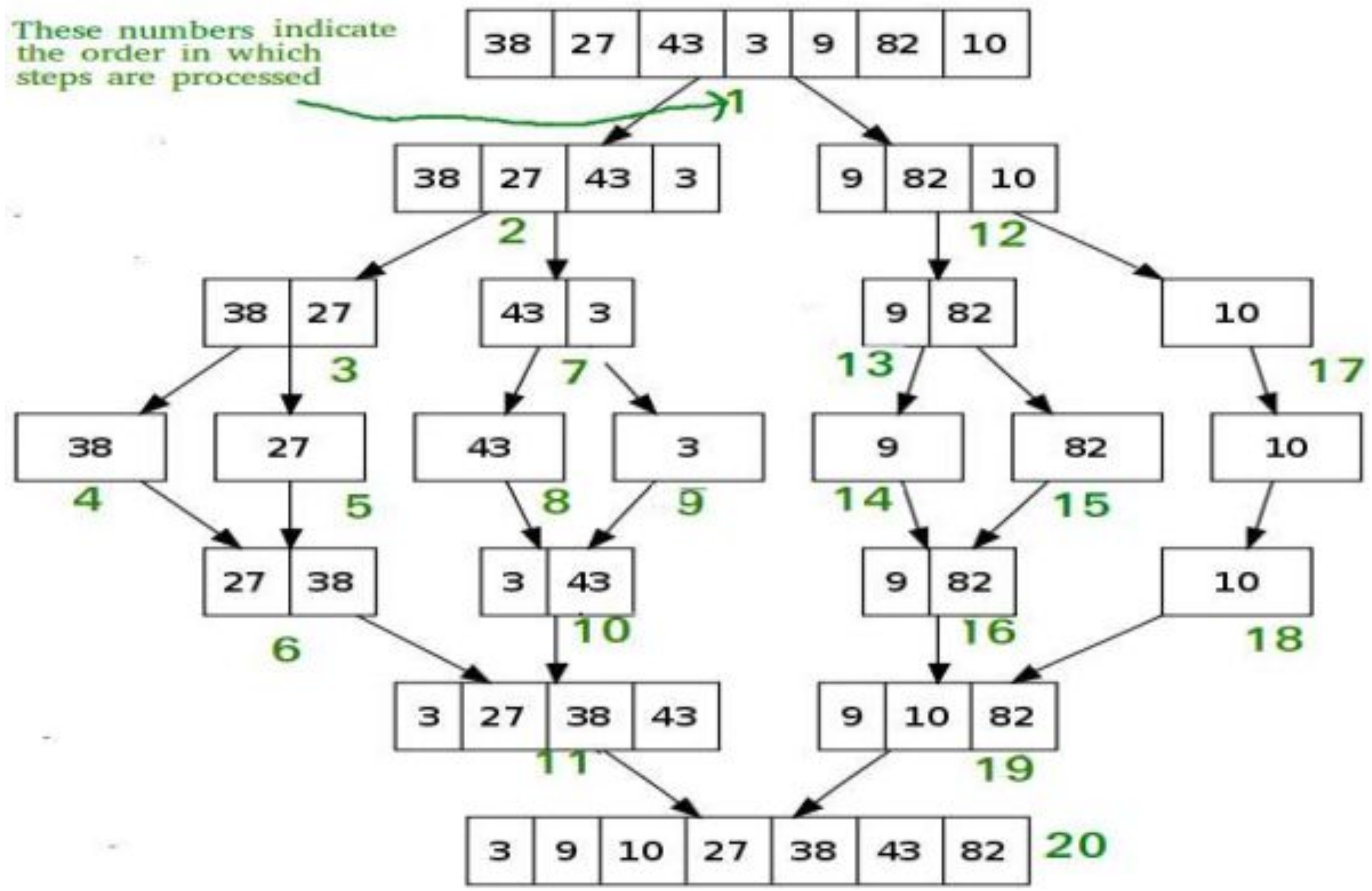
Merge Sort



```
Mergesort(Passed an array)
  if array size > 1
     Divide array in half
     Call Mergesort on first half.
     Call Mergesort on second half.
     Merge two halves.

Merge(Passed two arrays)
  Compare leading element in each array
  Select lower and place in new array.
    (If one input array is empty then place
     remainder of other array in output array)
```

Merge Sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
2
9 82 10
12
38 27
3
43 3
7
9 82
13
10
17
38
4
27
5
43
8
3
9
9
14
82
15
10
27 38
6
3 43
10
9 82
16
10
18
3 27 38 43
11
9 10 82
19
3 9 10 27 38 43 82
20

Merge Sort

void MergeSort(int LIST[], int lo, int hi)

- {
- int mid;
- if(lo < hi)
- {
- mid = (lo + hi)/ 2;
- MergeSort(LIST, lo, mid);
- MergeSort(LIST, mid+1, hi);
- ListMerger(LIST, lo, mid, hi);
- }
- }

- void ListMerger(int List[], int lo, int mid, int hi)
- {
- int TempList[hi-lo+1];

//temporary merger array

- int i = lo, j = mid + 1; //i is for left-hand,j is for right-hand
- int k = 0;

//k is for the temporary array

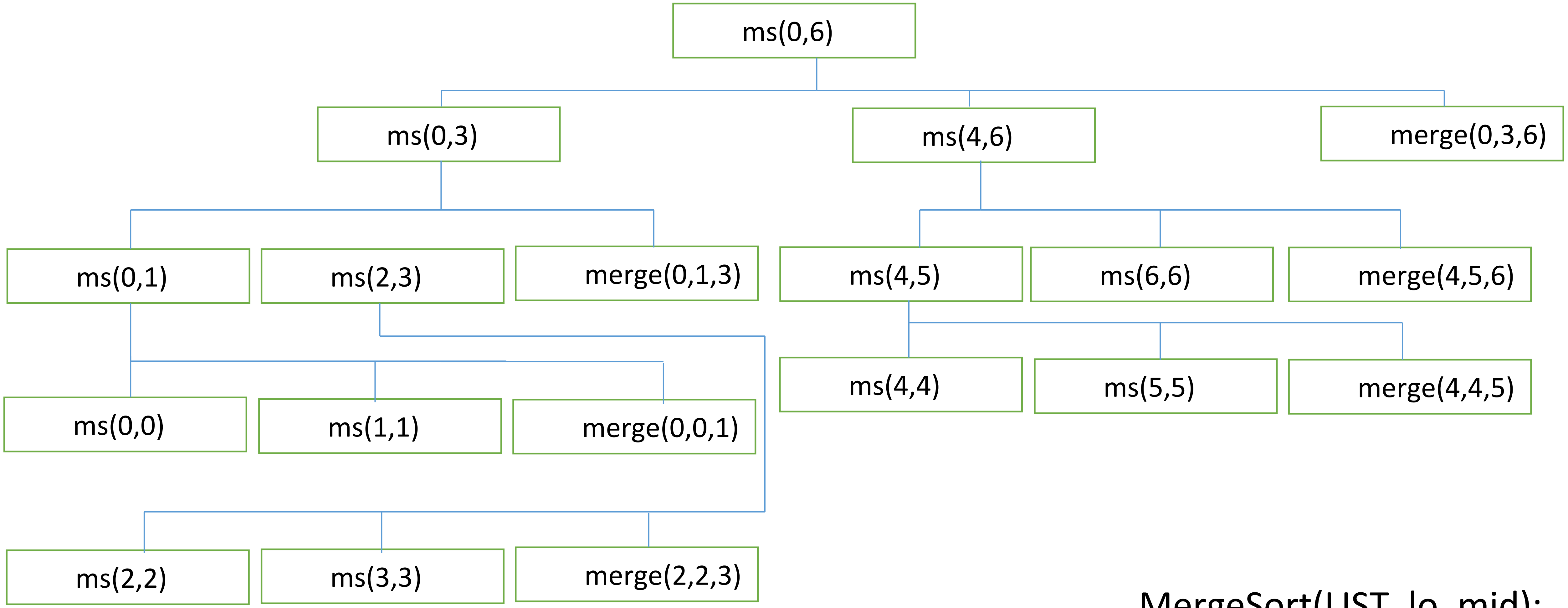
- while(i <= mid && j <=hi)
- {
- if(List[i] <= List[j])
- TempList[k++] = List[i++];
- else
- TempList[k++] = List[j++];
- }

Merge Sort

//remaining elements of left-half
- while(i <= mid)
- TempList[k++] = List[i++];
- //remaining elements of right-half
- while(j <= hi)
- TempList[k++] = List[j++];
- //copy the mergered temporary List to the original List
- for(k = 0, i = lo; i <= hi; ++i, ++k)
- List[i] = TempList[k];
- •

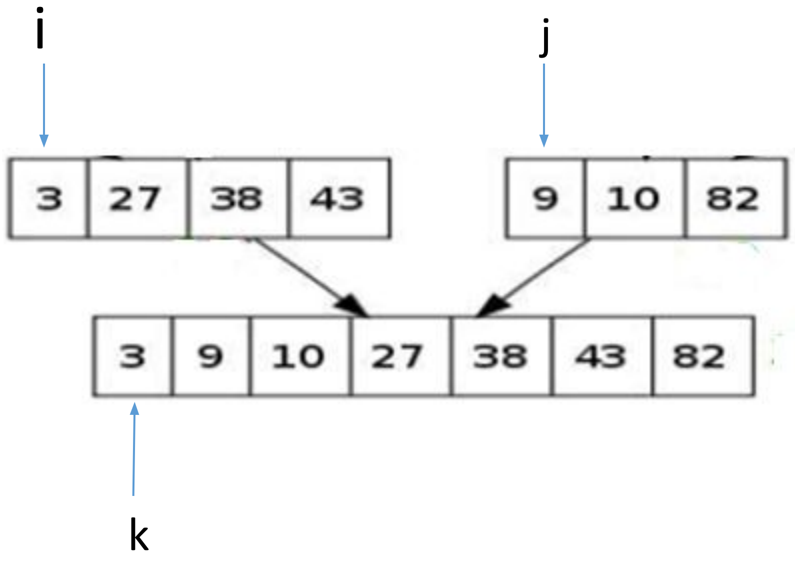
}

Merge Sort



```
MergeSort(LIST, lo, mid);
MergeSort(LIST, mid+1, hi);
ListMerger(LIST, lo, mid, hi);
```

Merge Sort



i	j	k
3	9	
27	10	
38	82	
43		

Performance

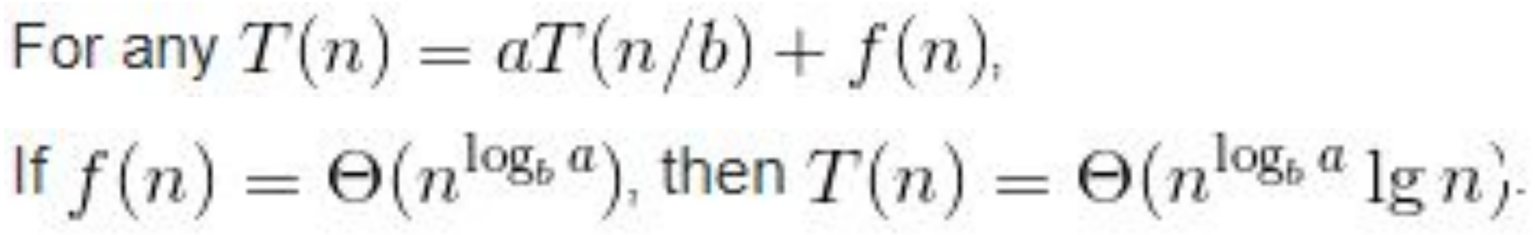
Time Complexity:
Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + O(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $O(n\log n)$. Time complexity of Merge Sort is $O(n\log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

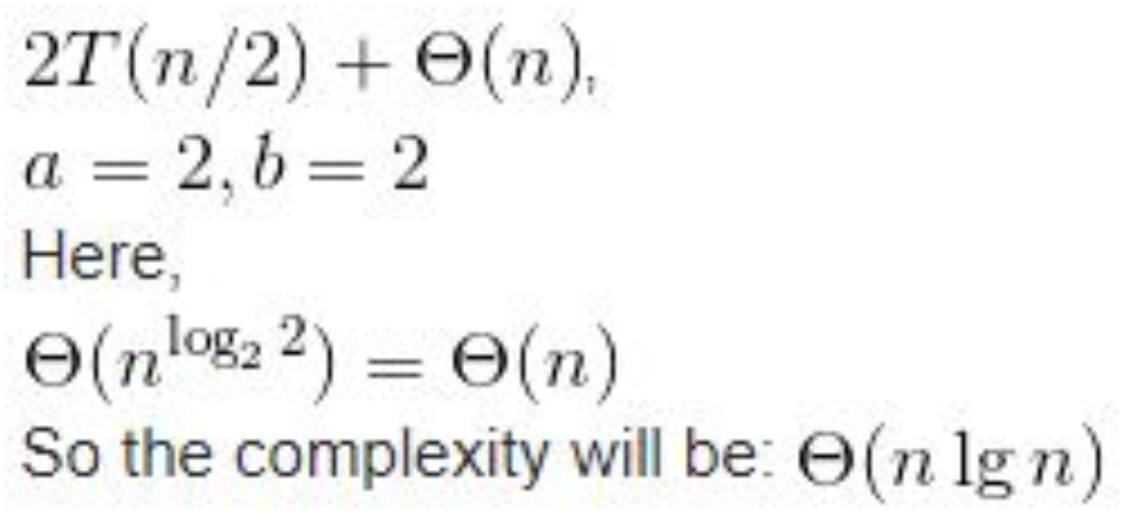
Master theorem

Master theorem states that,



For any $T(n) = aT(n/b) + f(n)$,

If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.



$2T(n/2) + \Theta(n)$,

$a = 2, b = 2$

Here,

$\Theta(n^{\log_2 2}) = \Theta(n)$

So the complexity will be: $\Theta(n \lg n)$

Performance

- **Auxiliary Space:** O(n)
- **Algorithmic Paradigm:** Divide and Conquer
- **Sorting In Place:** No in a typical implementation
- **Stable:** Yes

Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Introduction

- Fastest known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: O(N^2)
    - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like mergesort

Quicksort

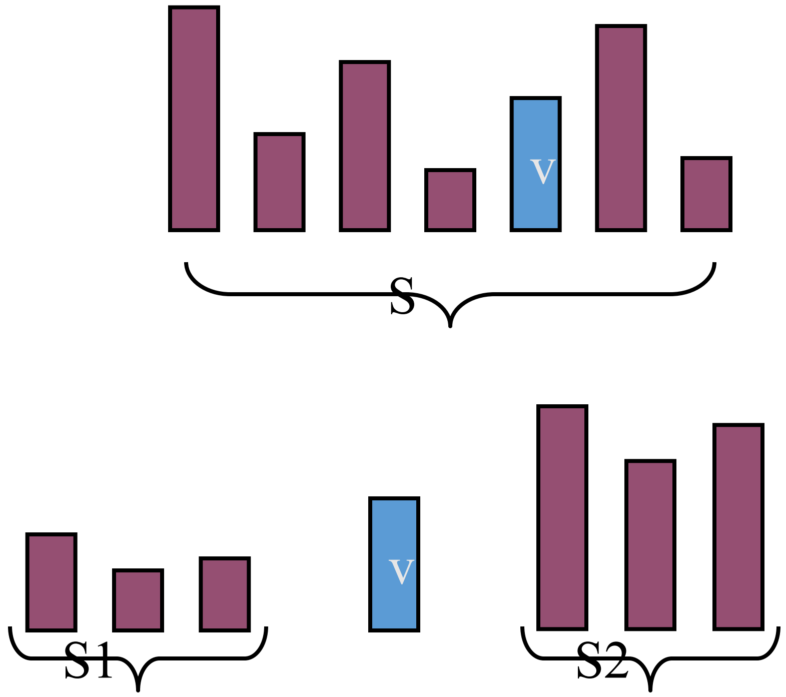
- Divide step:
  - Pick any element (***pivot***) v in S
  - Partition S – {v} into two disjoint groups

    S1 = {x ∈ S – {v} | x <= v}

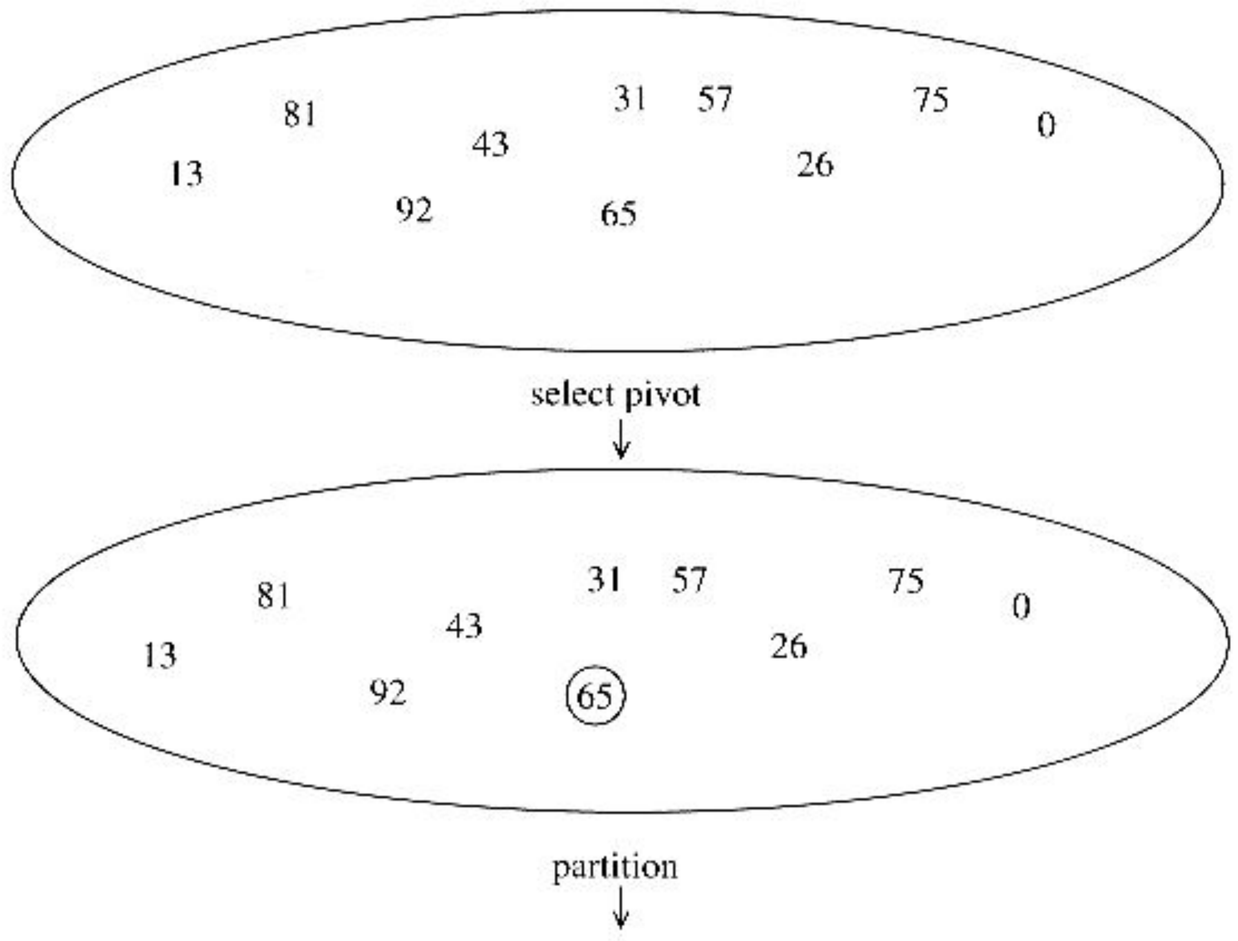
    S2 = {x ∈ S – {v} | x ≥ v}

- Conquer step: recursively sort S1 and S2

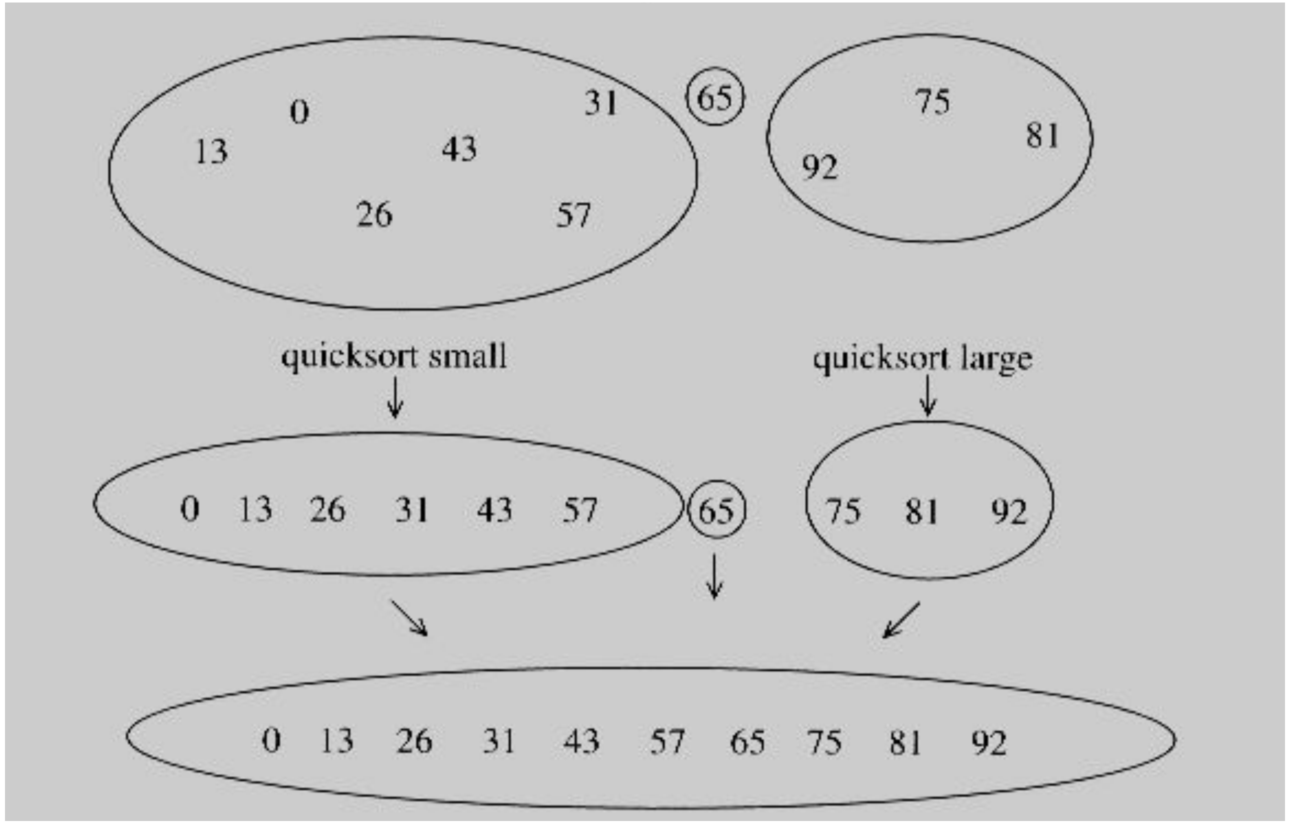
- Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)



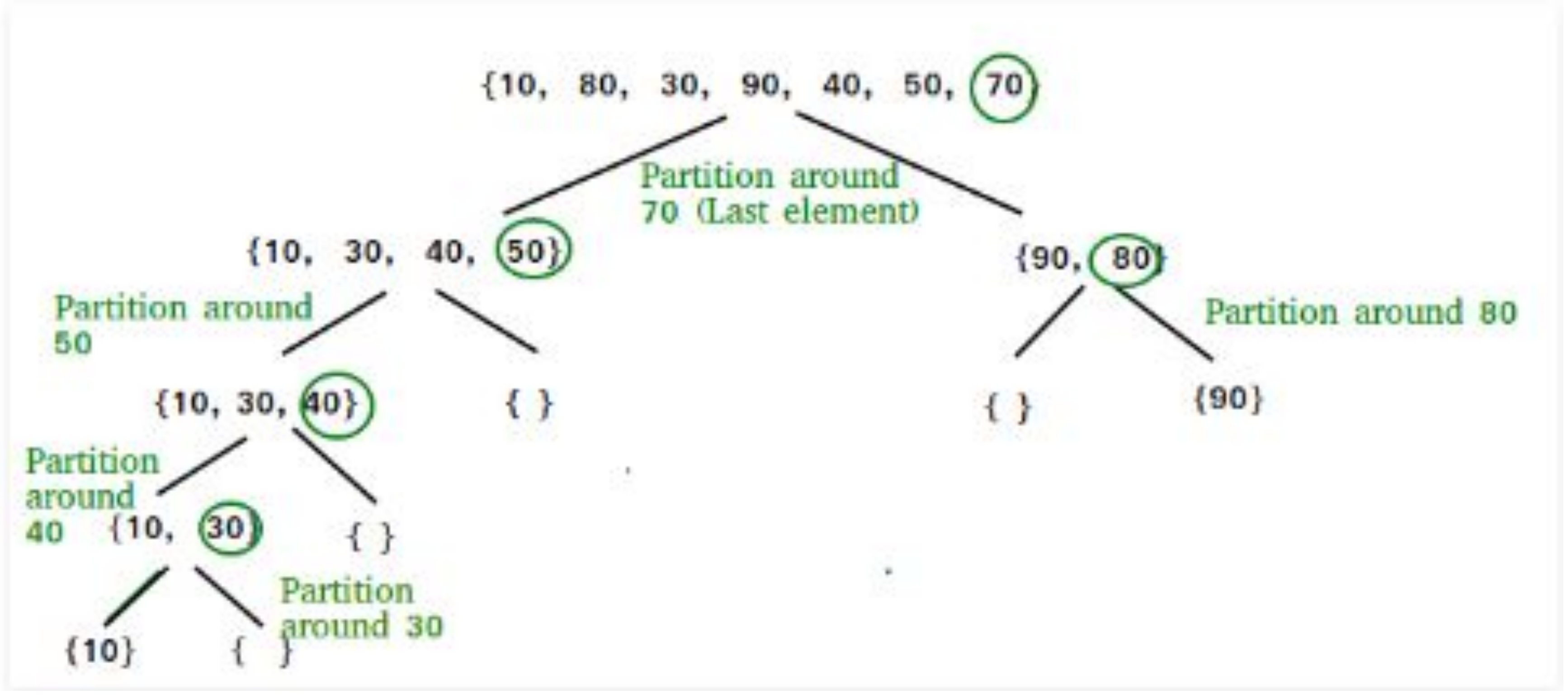
Example


81
31
57
75
0
43
26
13
92
65
select pivot
81
31
57
75
0
43
26
13
92
65
partition

Example

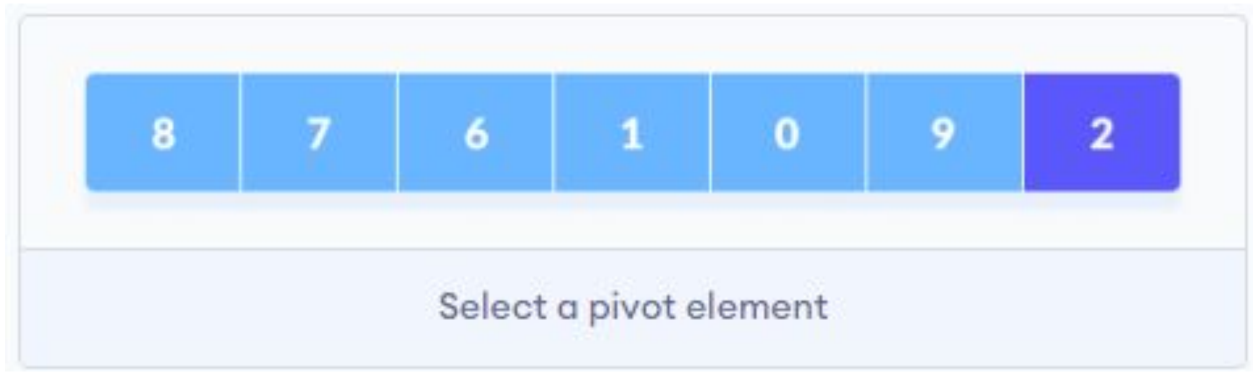

0
31
65
75
81
13
43
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Example

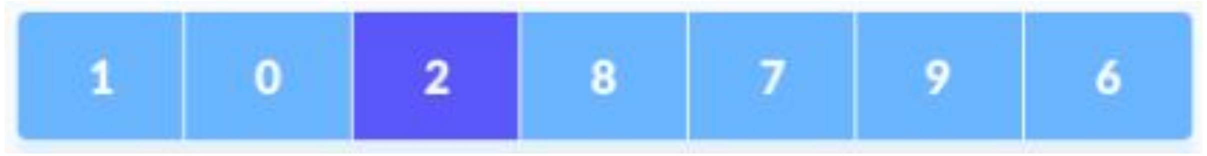

{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

How QuickSort Works?

A pivot element is chosen from the array. You can choose any element from the array as the pivot element.
Here, we have taken the rightmost (ie. the last element) of the array as the pivot element.



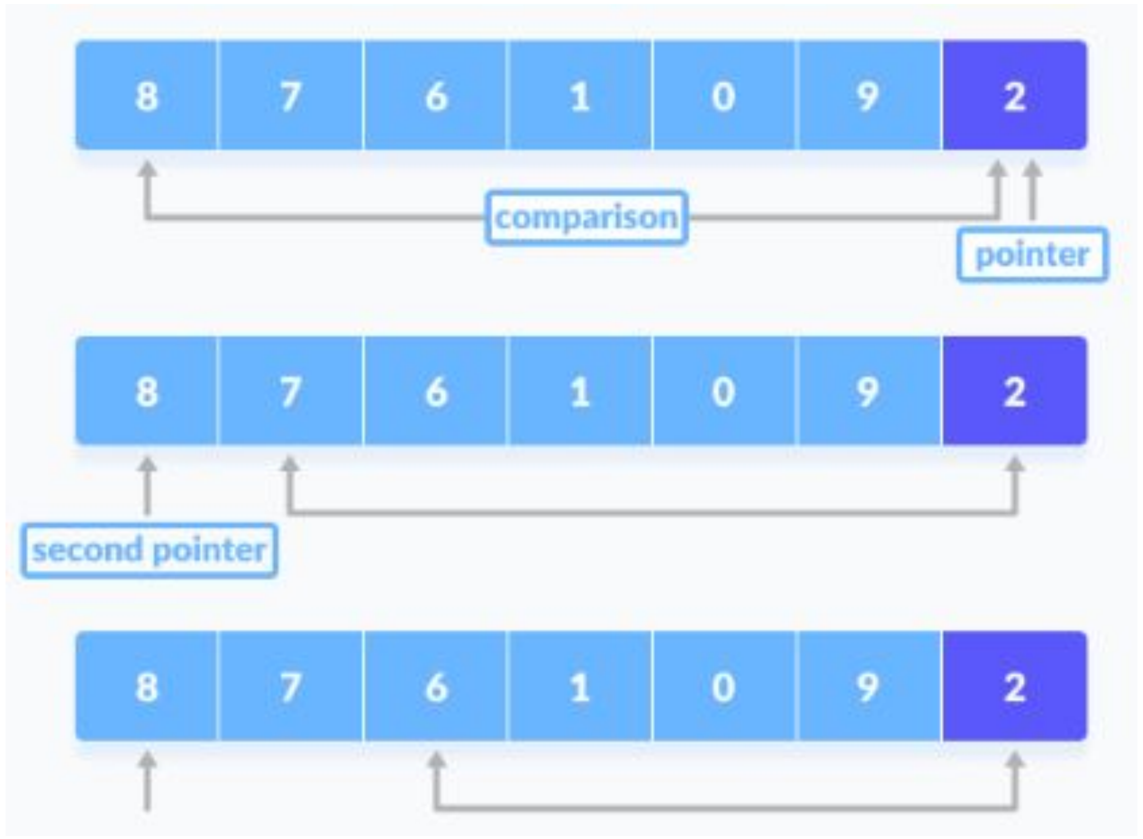
The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.

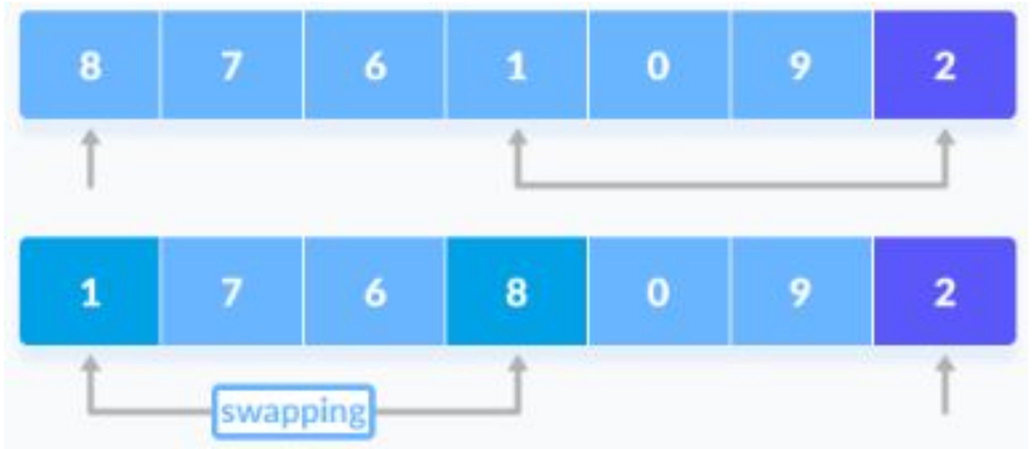


How QuickSort Works?

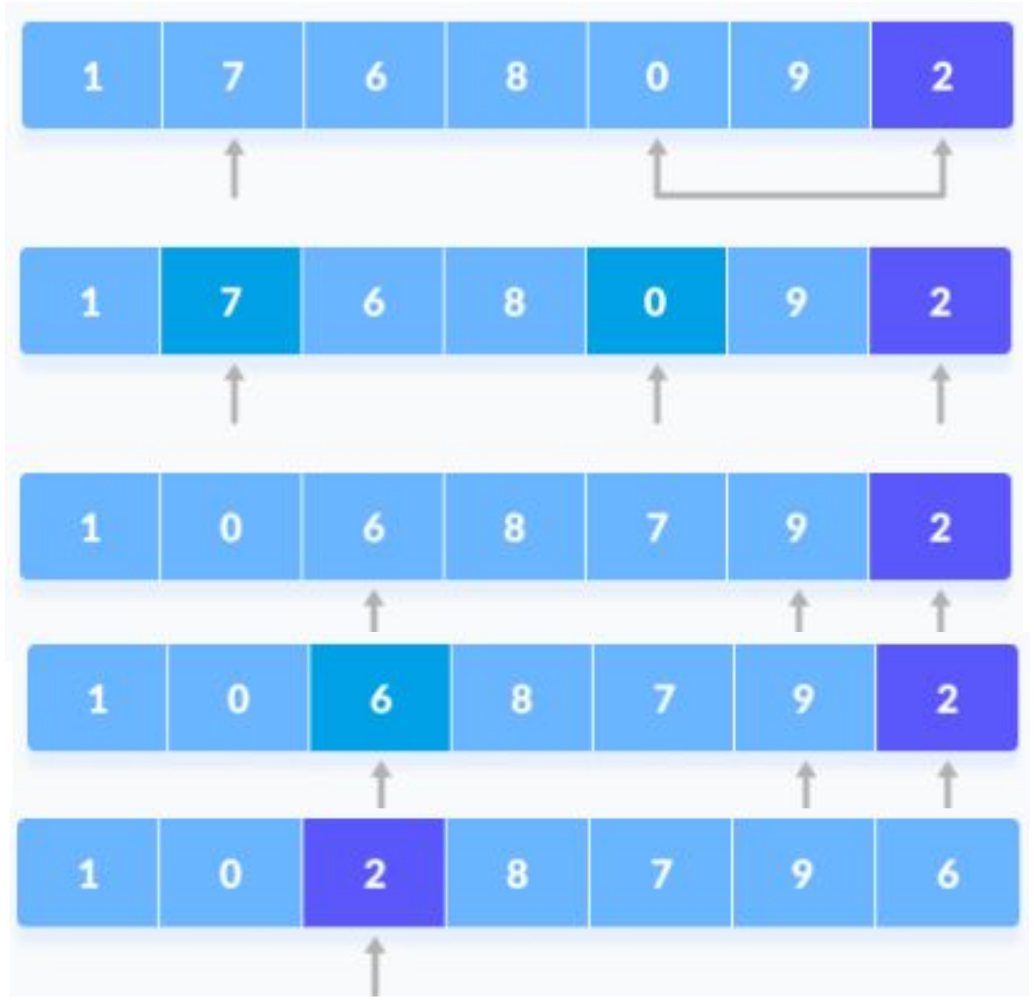
The above arrangement is achieved by the following steps.
A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. If the element greater than the pivot element is reached, a second pointer is set for that element.

Now, the pivot element is compared with the other elements (a third pointer). If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.


8 7 6 1 0 9 2
comparison
pointer
8 7 6 1 0 9 2
second pointer
8 7 6 1 0 9 2

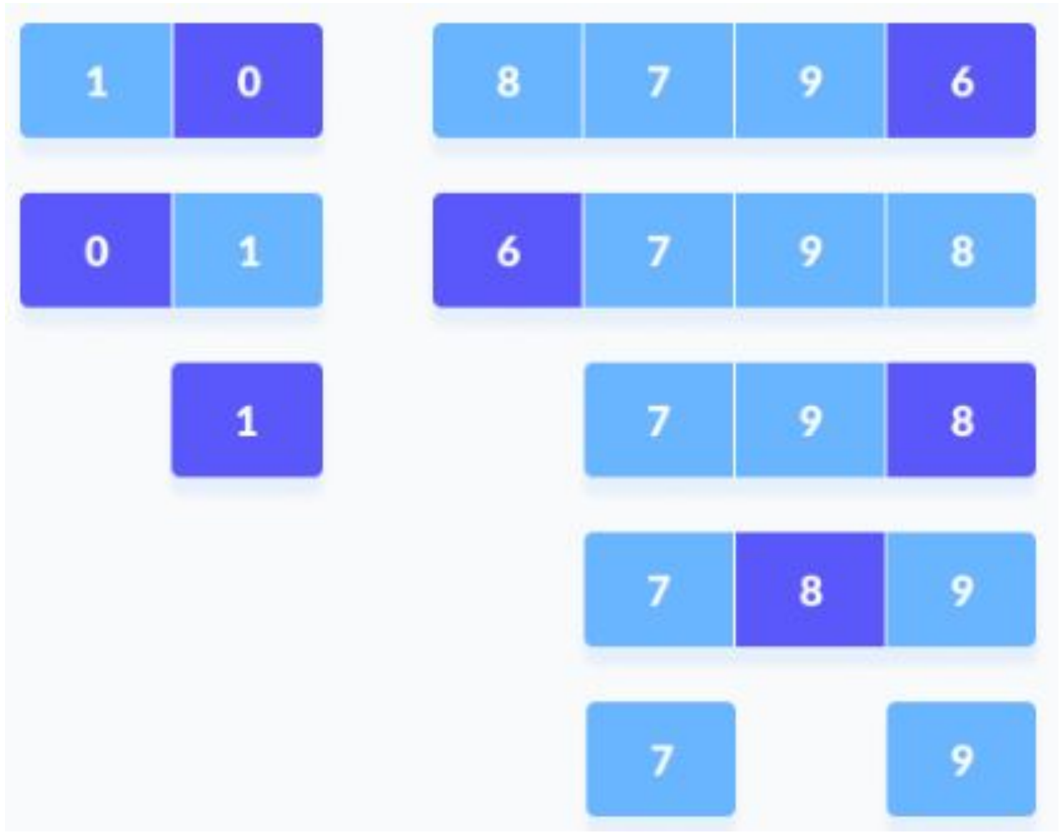

8 7 6 1 0 9 2
1 7 6 8 0 9 2
swapping

The process goes on until the second last element is reached. Finally, the pivot element is swapped with the second pointer.



Now the left and right subparts of this pivot element are taken for further processing in the steps below.

Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed at their right position. Then, step 2 is repeated.



Pseudo-code

Input: an array a[left, right]

```
QuickSort (a, left, right) {
    if (left < right) {
      pivot = Partition (a, left, right)
      Quicksort (a, left, pivot-1)
      Quicksort (a, pivot+1, right)
    }
}
```

Compare with MergeSort:

```
MergeSort (a, left, right) {
      if (left < right) {
       mid = divide (a, left, right)
       MergeSort (a, left, mid-1)
       MergeSort (a, mid+1, right)
       merge(a, left, mid+1, right)
      }
}
```

In-place Partition

- If use additional array (not in-place) like MergeSort
    - Straightforward to code like MergeSort

- Many ways to implement
- Even the slightest deviations may cause surprisingly bad results.
    - Not stable as it does not preserve the ordering of the identical keys.

Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
    - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
    - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

Quicksort Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time, i.e. O(1)
  - partitioning: linear time, i.e. O(N)
  - running time of the two recursive calls

Average-Case Analysis

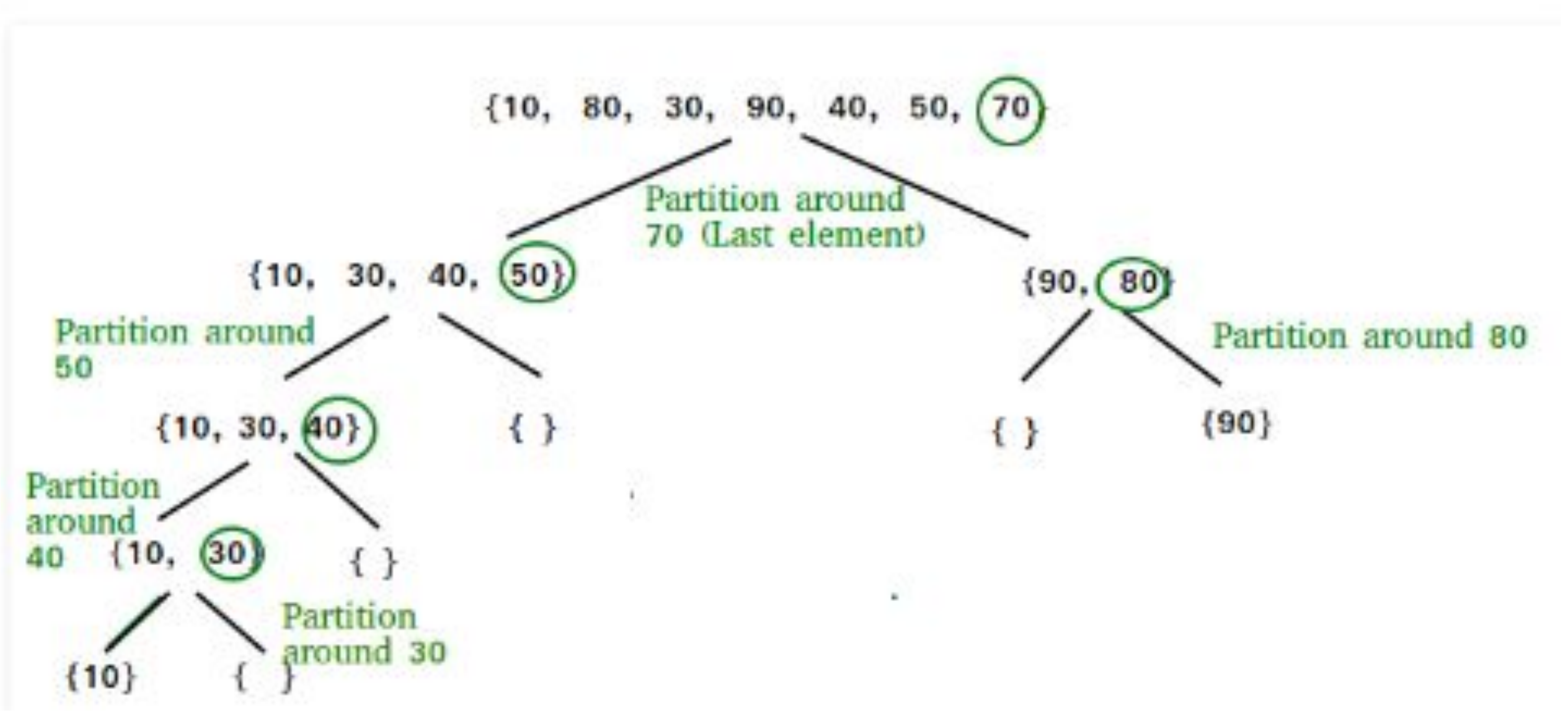
- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N)

Quick Sort vs Merge Sort

Quick sort is an internal algorithm which is based on divide and conquer strategy. In this:

The array of elements is divided into parts repeatedly until it is not possible to divide it further.
It is also known as “partition exchange sort”.
It uses a key element (pivot) for partitioning the elements.
One left partition contains all those elements that are smaller than the pivot and one right partition contains all those elements which are greater than the key element.

Quick Sort vs Merge Sort

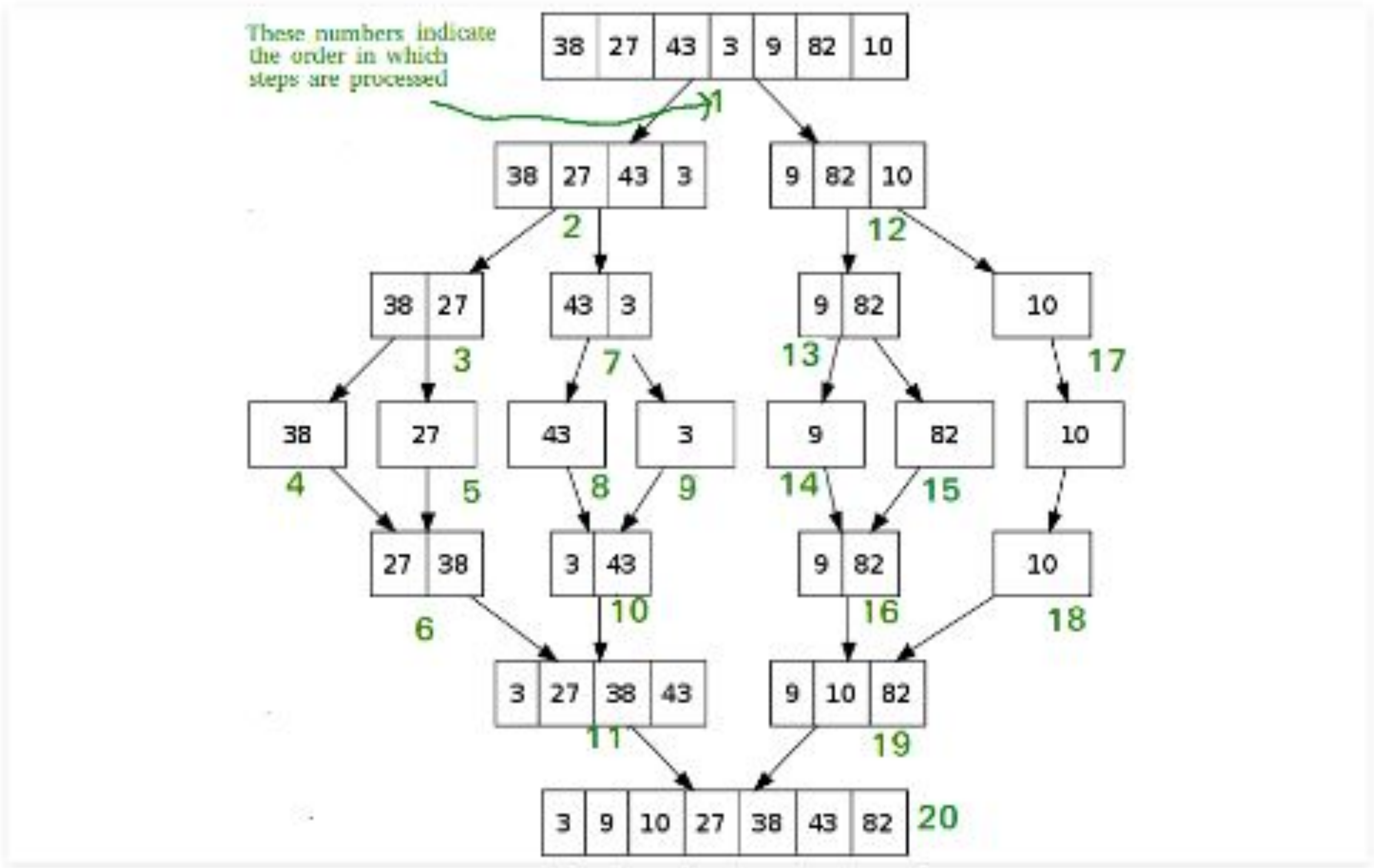


Quick Sort vs Merge Sort

Merge sort is an external algorithm and based on divide and conquer strategy. In this:

The elements are split into two sub-arrays (n/2) again and again until only one element is left.
Merge sort uses additional storage for sorting the auxiliary array.
Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
At last, the all sub arrays are merged to make it ‘n’ element size of the array.

Quick Sort vs Merge Sort



Quick Sort vs Merge Sort

1. Partition of elements in the array :

   In the merge sort, the array is parted into just 2 halves (i.e. n/2).

   whereas

   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.

2. Worst case complexity :

   The worst case complexity of quick sort is $O(n^2)$ as there is need of lot of comparisons in the worst condition.

   whereas

   In merge sort, worst case and average case has same complexities O(n log n).

Quick Sort vs Merge Sort

3.Efficiency :
Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
whereas
Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.

4.Sorting method :
The quick sort is internal sorting method where the data is sorted in main memory.
whereas
The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

Quick Sort vs Merge Sort

5. Stability :
   Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.
   whereas
   Quick sort is unstable in this scenario. But it can be made stable using some changes in code.
6. Preferred for :
   Quick sort is preferred for arrays.
   whereas
   Merge sort is preferred for linked lists.
7. Locality of reference :
   Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).