Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

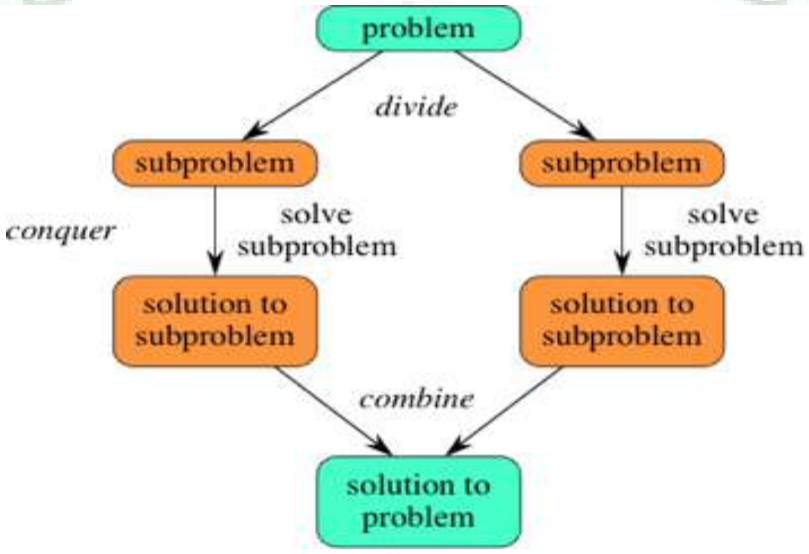
What is Divide and Conquer?

In computer science, many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems. In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Main Parts

Generally, divide-and-conquer algorithms have three parts −

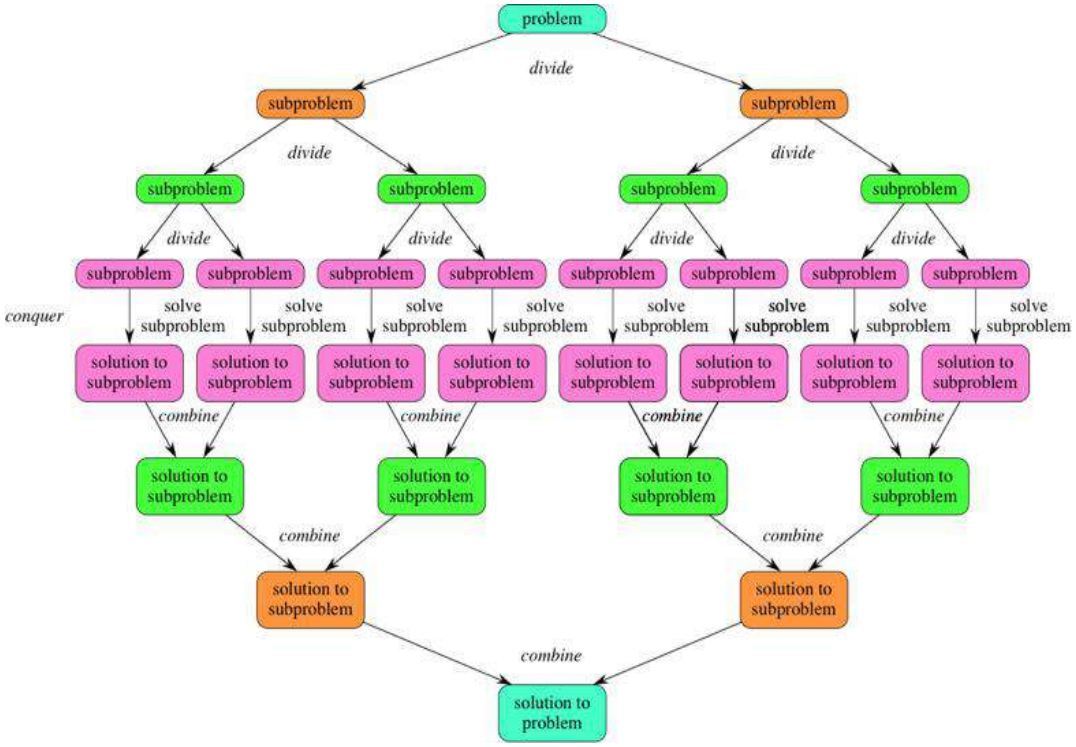
- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.



If we expand out two more recursive steps, it looks like this:

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



Pros and Cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion; hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Merge sort
- Binary search

What are the basic steps?

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Basic steps for divide and conquer methodology are

1. Base Case, solve the problem directly if it is small enough
2. Divide the problem into two or more similar and smaller subproblems
3. Recursively solve the subproblems
4. Combine solutions to the subproblems

Divide and Conquer Methodology for Sorting

The general algorithm for sorting a list of values by using divide and conquer methodology is as follow: (A is a list of values)

1. Base case at most one element (left ≥ right), return

2. Divide A into two subarrays: FirstPart, SecondPart
   - Two Subproblems:
     - Sort the FirstPart
     - Sort the SecondPart

3 Recursively
   - Sort FirstPart
   - Sort SecondPart

4 Combine sorted FirstPart and sorted SecondPart

➢ Merge Sort

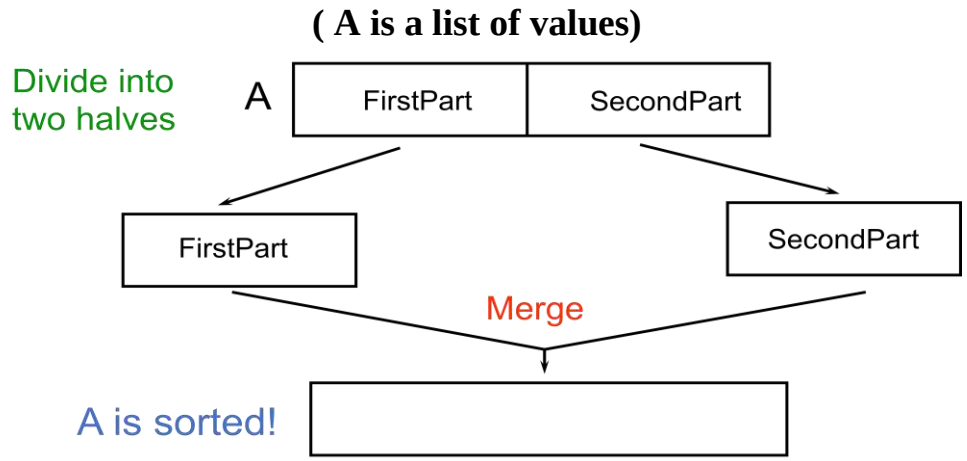
Merge Sort is a divide-and-conquer algorithm used for sorting arrays or lists. It works by dividing the input array into smaller subarrays, recursively sorting them, and then merging the sorted subarrays to produce a fully sorted array.

Key Features:

- Stable Sort: Maintains the relative order of equal elements.
- Time Complexity: Always O(nlog⁡n)O(nlog n), regardless of input order.
- Space Complexity: Requires extra memory for merging, with O(n)O(n) space usage.
- Algorithm Type: Recursive.

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



Merge Sort Algorithm

```
Merge-Sort (A, left, right)
    if    left ≥ right   return
    else
        middle ← (left+right)/2⌋
        Merge-Sort (A, left, middle)
        Merge-Sort (A, middle+1, right)
        Merge (A, left, middle, right)
```

Merge(A, left, middle, right)

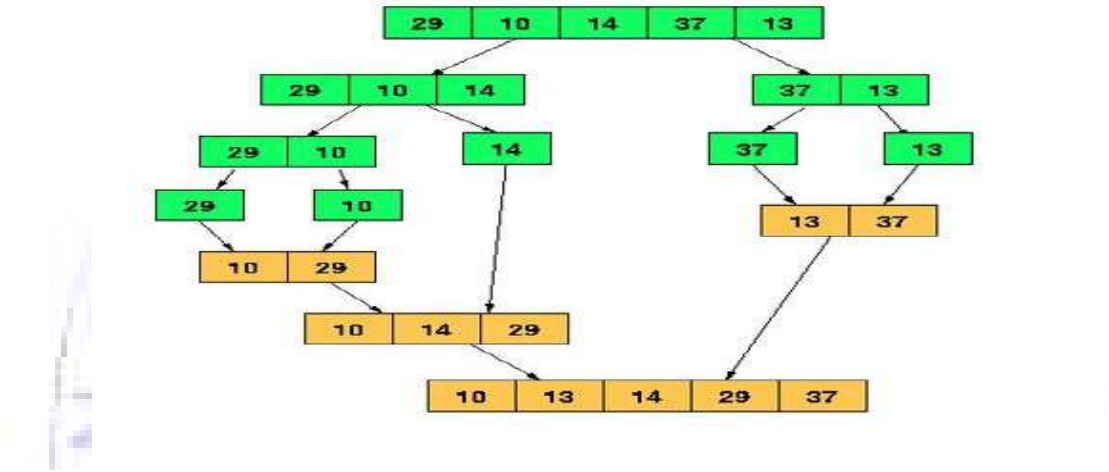
```
n1 ← middle – left + 1    // n1 is a size of left part
n2 ← right – middle       // n2 is a size of right part
create array L[n1], R[n2]
for i ← 0 to n1-1 do L[i] ← A[left +i]
for j ← 0 to n2-1 do R[j] ← A[middle+j+1]
 i ← j ← 0
k ←left
while i < n1 &  j < n2
                    if L[i] < R[j]
                     A[k++] ← L[i++]
                    else
                     A[k++] ← R[j++]
while i < n1
                  A[k++] ← L[i++]
while j < n2
                  A[k++] ← R[j++]
```

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Time Complexity (Merge Sort): O(nlogn)

Example (Merge Sort)

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.



Examples

Example1:

Input Array: [38, 27, 43, 3, 9, 82, 10]

Process:

1. Divide: Recursively split the array:
   - Split into: [38, 27, 43] and [3, 9, 82, 10]
   - Further split [38, 27, 43] into [38, 27] and [43]
   - Continue until subarrays contain single elements: [38], [27], [43], [3], [9], [82], [10]
2. Conquer: Recursively sort and merge:
   - Merge [38] and [27] into [27, 38]
   - Merge [27, 38] and [43] into [27, 38, 43]
   - Merge [3], [9], and [10] into [3, 9, 10]
   - Merge [3, 9, 10] and [82] into [3, 9, 10, 82]
3. Merge: Combine sorted subarrays:
   - Merge [27, 38, 43] and [3, 9, 10, 82] into [3, 9, 10, 27, 38, 43, 82].

Output: [3, 9, 10, 27, 38, 43, 82]

Example 2:

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Input Array: [5, 2, 9, 1, 7, 6]

Process:

1. **Divide:**
   - [5, 2, 9] and [1, 7, 6]
   - [5, 2] and [9], [1, 7] and [6]
   - [5], [2], [9], [1], [7], [6]
2. **Conquer:**
   - Merge [5] and [2] into [2, 5]
   - Merge [1] and [7] into [1, 7]
   - Merge [2, 5] and [9] into [2, 5, 9]
   - Merge [1, 7] and [6] into [1, 6, 7]
3. **Merge:**
   - Combine [2, 5, 9] and [1, 6, 7] into [1, 2, 5, 6, 7, 9].

Output: [1, 2, 5, 6, 7, 9]

Recursive Definition:

Merge Sort can be defined recursively as follows:

1. If the array contains 1 or 0 elements, it is already sorted.
2. Otherwise, divide the array into two halves.
3. Recursively sort both halves.
4. Merge the two sorted halves into a single sorted array.

Time Complexity

Using Recurrence Tree:

Expand step by step:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$= 2\left[2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right] + cn$$

$$= 4T\left(\frac{n}{4}\right) + cn + cn$$

$$= 4\left[2T\left(\frac{n}{8}\right) + c\frac{n}{4}\right] + 2cn$$

$$= 8T\left(\frac{n}{8}\right) + cn + cn + cn$$

At the k-th level:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k \cdot cn$$

Base Case:

When $\frac{n}{2^k} = 1$, $k = \log_2 n$:

$$T(1) = O(1)$$

Substitute $k = \log_2 n$:

$$T(n) = 2^{\log_2 n} T(1) + \log_2 n \cdot cn$$

$$T(n) = n \cdot O(1) + cn \log_2 n$$

Final Solution:

$$T(n) = O(n \log n)$$

➢ Quick Sort

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general purpose sort and it consumes relatively fewer resources during execution.

Like merge sort, quick sort uses divide-and-conquer, and so it's a recursive algorithm. The way that quick sort uses divide-and-conquer is a little different from how merge sort does.

In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quick sort is the opposite: all the real work happens in the divide step. In fact, the combine step in quick sort does absolutely nothing.

Quick sort has a couple of other differences from merge sort. Quick sort works in place and its worst-case running time is as bad as selection sort's and insertion sort's: $\Theta(n^2)$.

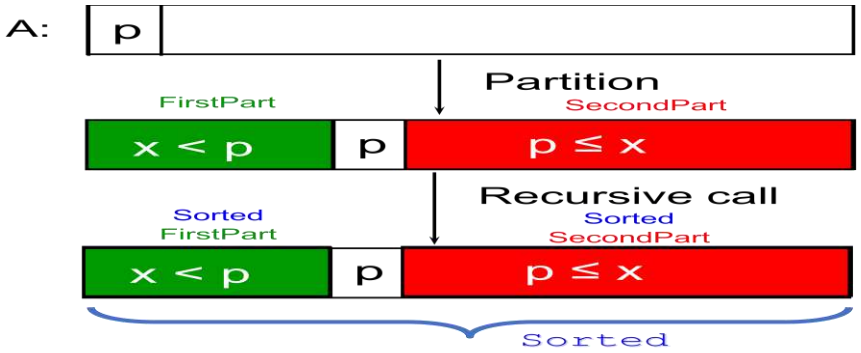
But its average-case running time is as good as merge sort's: $\Theta(n\log n)$. So why think about quick sort when merge sort is at least as good?

That's because the constant factor hidden in the big-Θ notation for quick sort is quite good. In practice, quick sort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

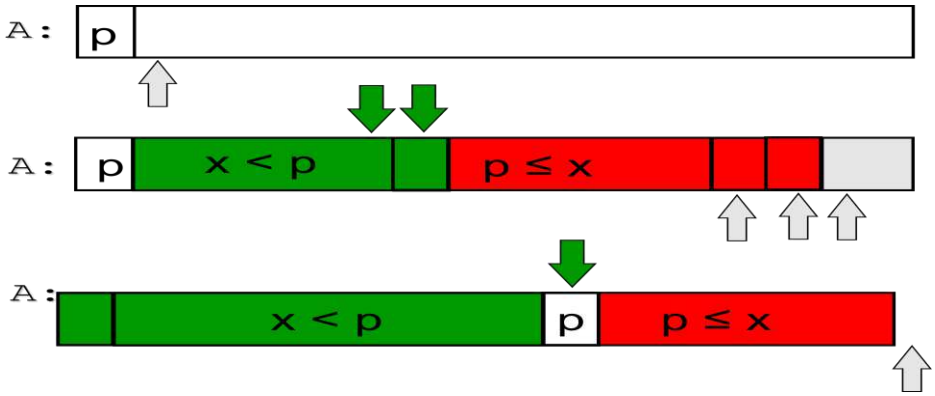
Quick Sort Main Parts

- **Divide**
    - Pick any element **p** as the **pivot**, e.g, the first element
    - Partition the remaining elements into
        - **First Part, which contains all elements < p**
        - **Second Part, which contains all elements ≥ p**
- **Recursively sort** the First Part and Second Part
- **Combine**: no work is necessary since sorting is done in place

Algorithm



Partitions



Quick Sort Algorithm

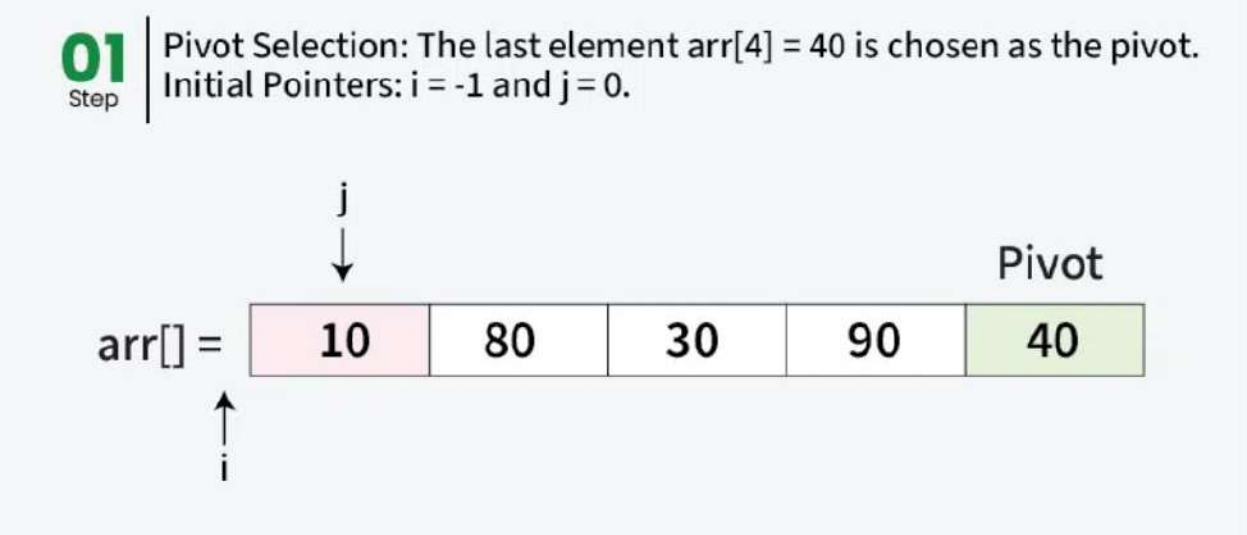
Quick-Sort (A, left, right)

```
{
    if   left ≥ right  return
     else
            middle ← Partition (A, left, right)
            Quick-Sort (A, left, middle–1)
            Quick-Sort (A, middle+1, right)
    end if
}
```

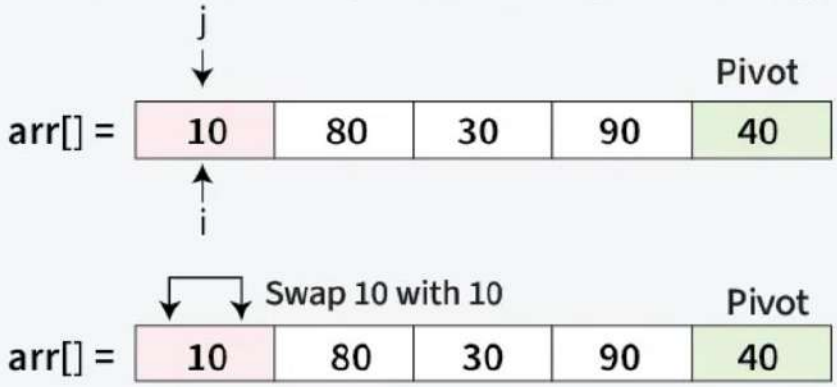
Partition (A, left, right)

```
{
    x ← A[left]
    i ← left
    for j ← left+1 to right
        if A[j] < x then
            i ← i + 1
            swap (A[i], A[j])
        end if
    end for j
    swap (A[i], A[left])
    return i
}
```

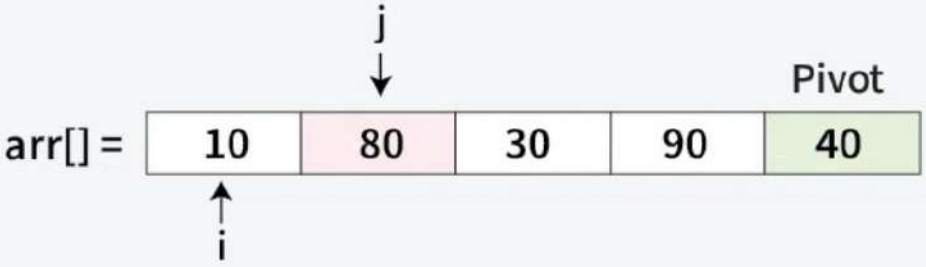
Example: Quick Sort



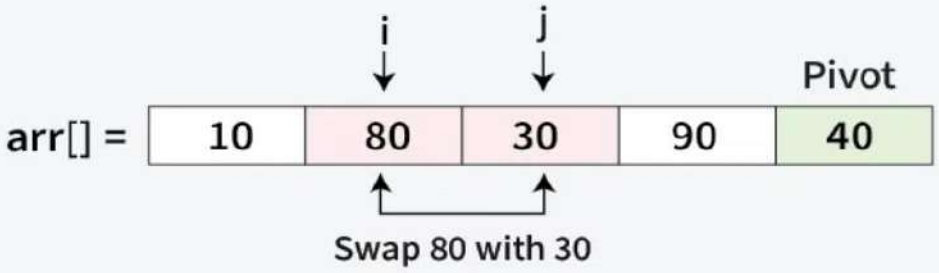
02 Step | Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1

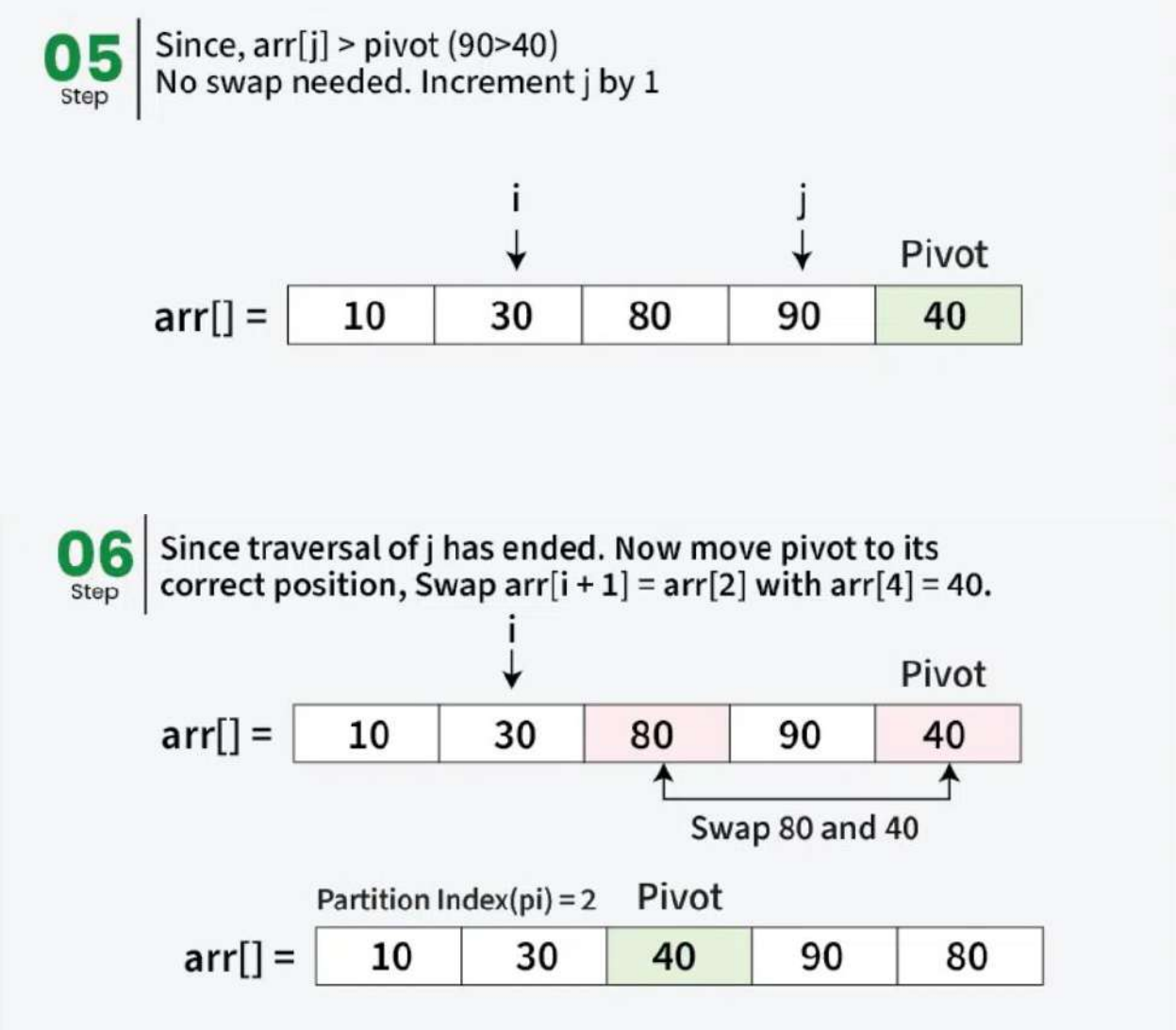


03 Step | Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1



04 Step | Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1





Divide: by choosing any element in the sub-array array[p..r]. Call this element the **pivot**. Rearrange the elements in array[p..r] so that all other elements in array[p..r] that are less than or equal to the pivot are to its left and all elements in array[p..r] that are greater than pivot are to the pivot's right. We call this procedure **partitioning**. At this point, it doesn't matter what order the elements to the left of the pivot are in relative to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.

As a matter of practice, we'll always choose the rightmost element in the sub-array, array[r], as the pivot. So, for example, if the sub-array consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we

choose 6 as the pivot. After partitioning, the sub-array might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let **q** be the index of where the pivot ends up.

Conquer: by recursively sorting the sub-arrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+1..r] (all elements to the right of the pivot, which must be greater than the pivot).

Combine: by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in array[p..q-1], are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in array[q+1..r], are greater than the pivot and are sorted. The elements in array[p..r] can't help but be sorted!

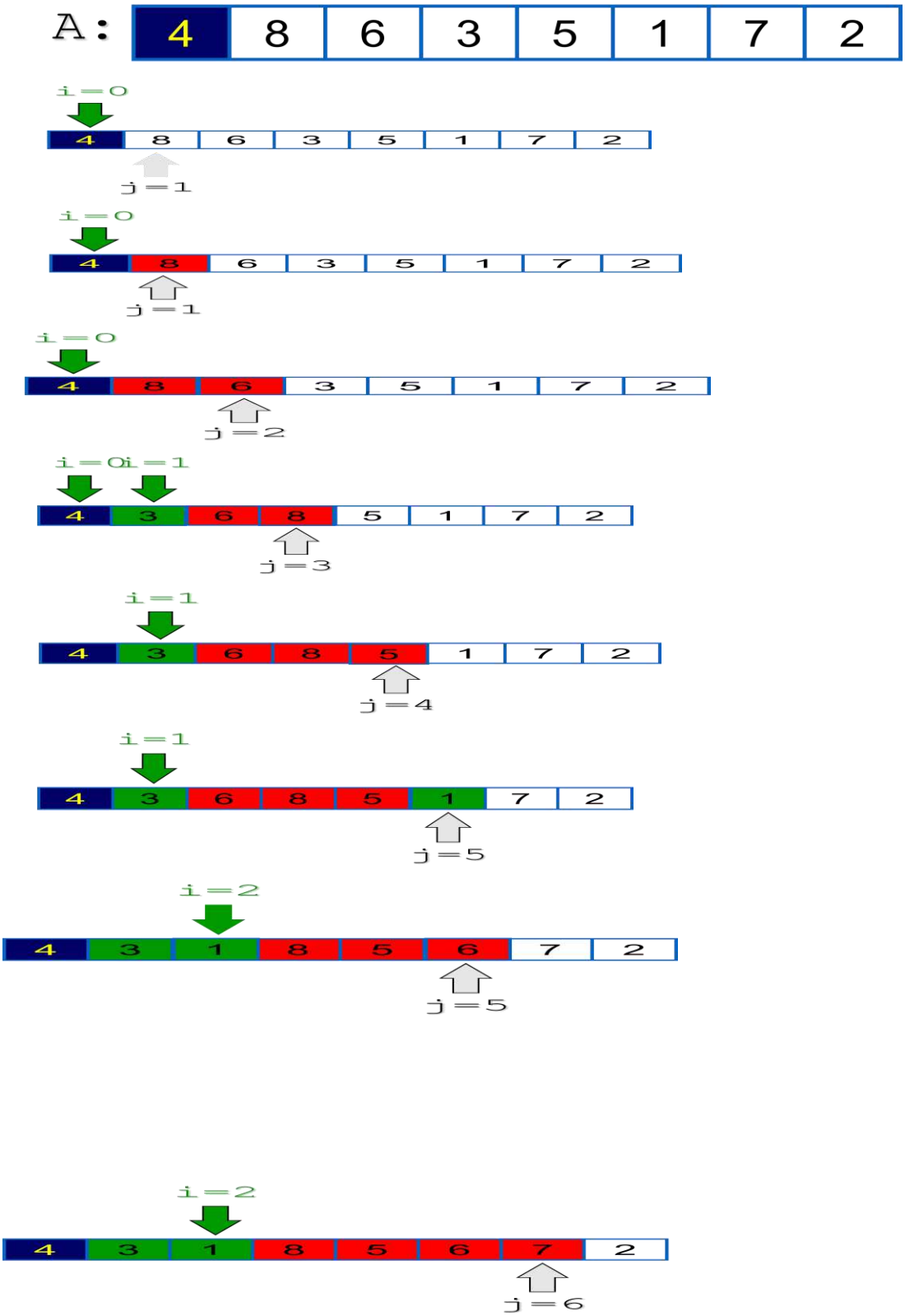
Think about our example. After recursively sorting the sub-arrays to the left and right of the pivot, the sub-array to the left of the pivot is [2, 3, 5], and the sub-array to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the sub-array has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The sub-array is sorted.

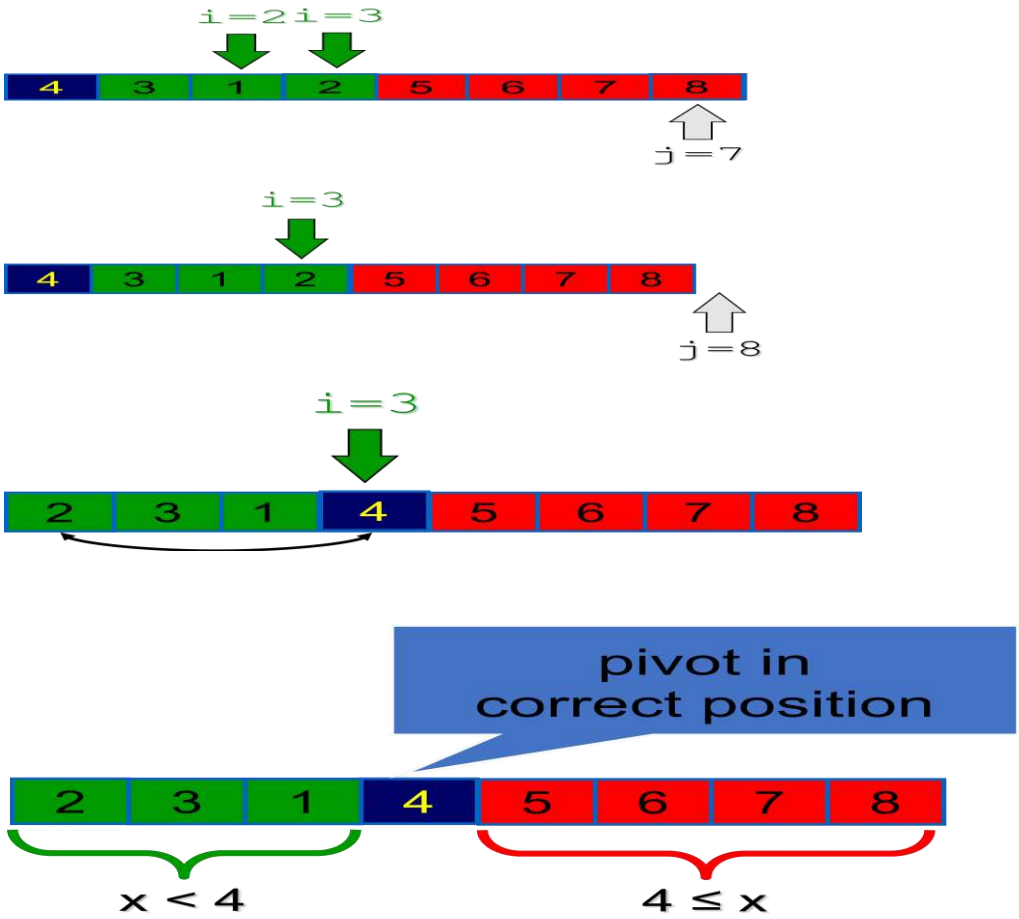
The base cases are sub-arrays of fewer than two elements, just as in merge sort. In merge sort, you never see a sub-array with no elements, but you can in quick sort, if the other elements in the sub-array are all less than the pivot or all greater than the pivot.

Let's go back to the conquer step and walk through the recursive sorting of the sub-arrays. After the first partition, we have sub-arrays of [5, 2, 3] and [12, 7, 14, 9, 10, 11], with 6 as the pivot.

To sort the sub-array [5, 2, 3], we choose 3 as the pivot. After partitioning, we have [2, 3, 5]. The sub-array [2], to the left of the pivot, is a base case when we recurs, as is the sub-array [5], to the right of the pivot.
To sort the sub-array [12, 7, 14, 9, 10, 11], we choose 11 as the pivot, resulting in [7, 9, 10] to the left of the pivot and [14, 12] to the right. After these sub-arrays are sorted, we have [7, 9, 10], followed by 11, followed by [12, 14].

Quick Sort Example


A:
4 8 6 3 5 1 7 2
i=0
j=1
i=0
j=1
i=0
j=2
i=0i=1
j=3
i=1
j=4
i=1
j=5
i=2
j=5
i=2
j=6



Recurrence Relation of Quick Sort

Let T(n)T(n)T(n) represent the time complexity of Quick Sort for an input of size nnn. The recurrence relation can be expressed as:

$$T(n)=T(k)+T(n-k-1)+O(n)$$
$$T(n)=T(k)+T(n-k-1)+O(n)$$
$$T(n)=T(k)+T(n-k-1)+O(n)$$

- The **partitioning step** takes O(n)O(n)O(n) time.
- The array is divided into two parts, one of size kkk and the other of size n−k−1n - k - 1n−k−1.
- In the best and average case, the pivot divides the array into equal halves:

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

Using the **Master Theorem**, this results in O(nlog⁡n)O(n \log n)O(nlogn).

- In the worst case (when the pivot is always the smallest or largest element):

  T(n)=T(n−1)+O(n)T(n) = T(n - 1) + O(n)T(n)=T(n−1)+O(n)

  This results in O(n2)O(n^2)O(n2).