Complexity Analysis & Sorting

Sorting Revisited

- The time complexity of the selection sort algorithm is $O(N^2)$.
- Its execution time is *long* when N is large enough.
- Is it possible to solve the sorting problem *more efficiently*?

Sorting using Recursion

- In a $O(N^2)$ time algorithm, *doubling* N *increases* the running time by a factor of *four*.

- The reverse is also true: *halving* N *decreases* the running time by the same factor of *four*.

- This suggests that *dividing* an array in half and *recursively* sorting the subarrays might reduce the sorting time.

Divide-and-Conquer

N

$O(N^2)$

$T(N) \le cN^2$

N/2 | N/2

$T(N) \le 2c(N/2)^2 = cN^2/2$

- Recall the ***divide-and-conquer*** approach.
  - *Divide* the problem into smaller pieces.
  - (*Divide* an array in half.)

  - *Tackle* each sub-task either directly or by recursion.
  - (Recursively *sort* each subarray.)

  - *Combine* the solutions of the parts to form the solution of the whole.
  - (*Merge* two sorted subarrays to one sorted array.)

The Merge Sort Algorithm

`array`	56	25	37	58	95	19	73	30

- An array of size 0 or 1 must already be sorted.
- When n > 1,
  - *divide* the array into two subarrays of smaller size;

`arr1`	56	25	37	58

`arr2`	95	19	73	30

  - *recursively sort* each subarray;

`arr1`	25	37	56	58

`arr2`	19	30	73	95

  - *merge* the sorted subarrays back into the original array.

`array`	19	25	30	37	56	58	73	95

Merge Sort Implementation

```
void MergeSort(int *array, int n) {
   int n1, n2, *arr1, *arr2;

   if (n > 1) {
      n1 = n / 2;     // size of 1st subarray
      n2 = n - n1;    // size of 2nd subarray
      arr1 = (int *)malloc(n1 * sizeof(int));
      arr2 = (int *)malloc(n2 * sizeof(int));
      Divide(array, arr1, n1, arr2, n2);
      MergeSort(arr1, n1);
      MergeSort(arr2, n2);
      Merge(array, arr1, n1, arr2, n2);
      free(arr1);
      free(arr2);
   }
}
```

A *test* to stop or continue

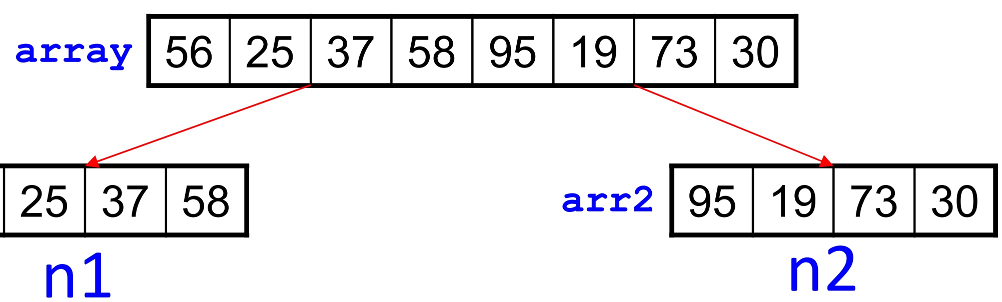
Two *recursive calls* to continue recursion

An *end case* to terminate the recursion (nothing to do)

Merge Sort Implementation (Divide)

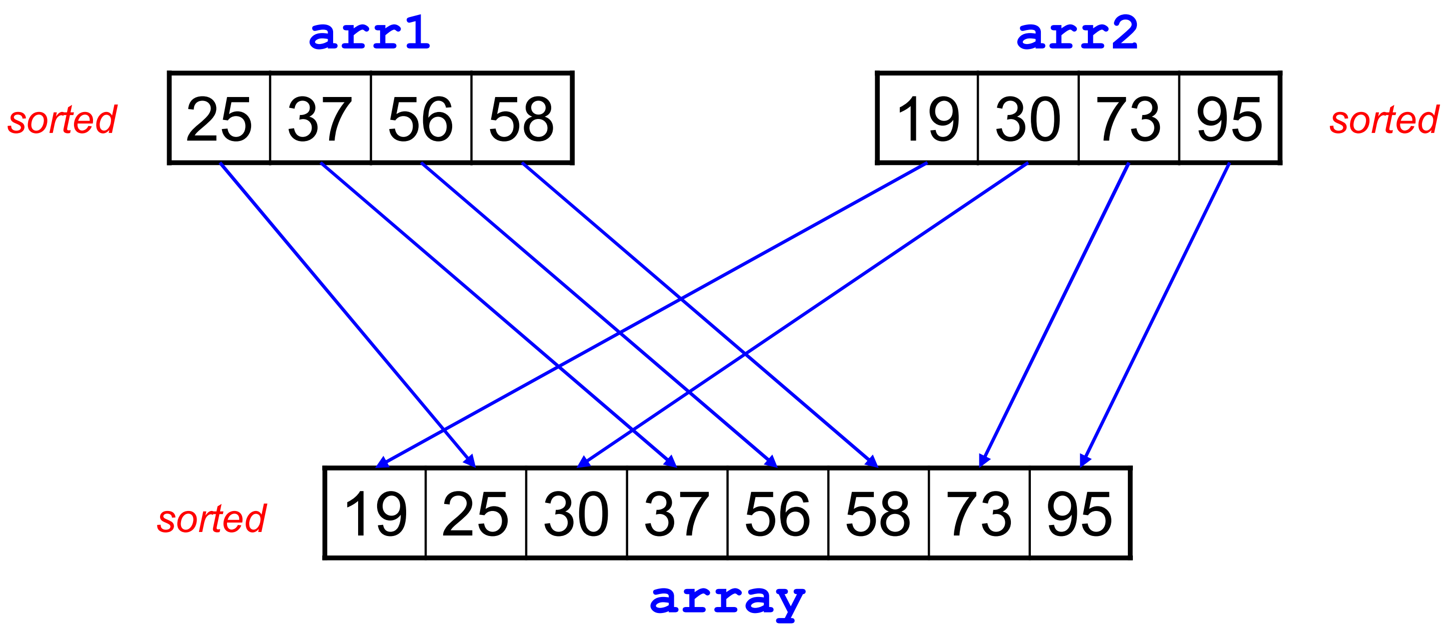
```
void Divide(int *array, int *arr1, int n1, int *arr2, int n2) {
   int i;
   for (i = 0; i < n1; i++)
      arr1[i] = array[i];
   for (i = 0; i < n2; i++)
      arr2[i] = array[i + n1];
}
```

- The front half of **`array`** is placed in **`arr1`**.
- The rear half of **`array`** is placed in **`arr2`**.

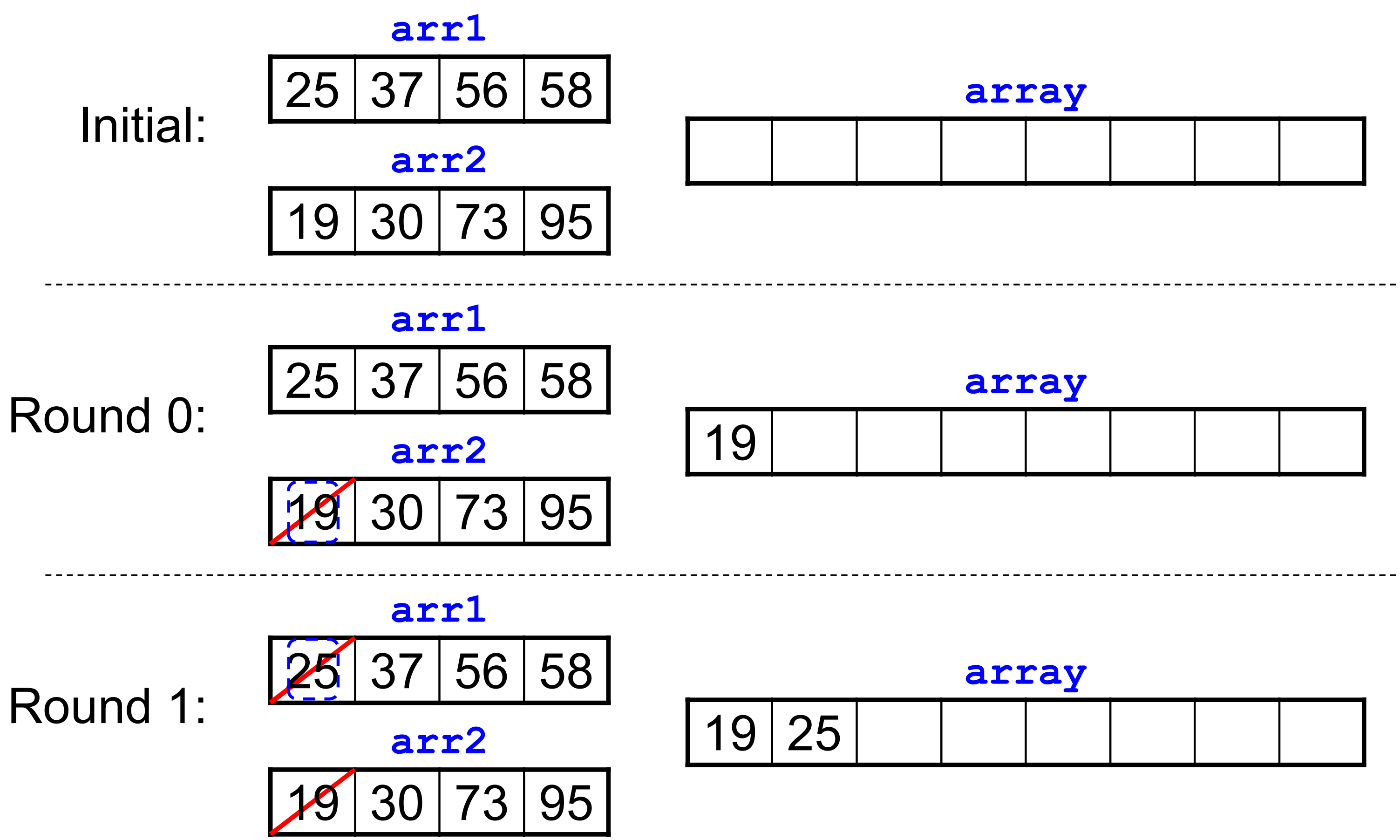


Merging Two Arrays

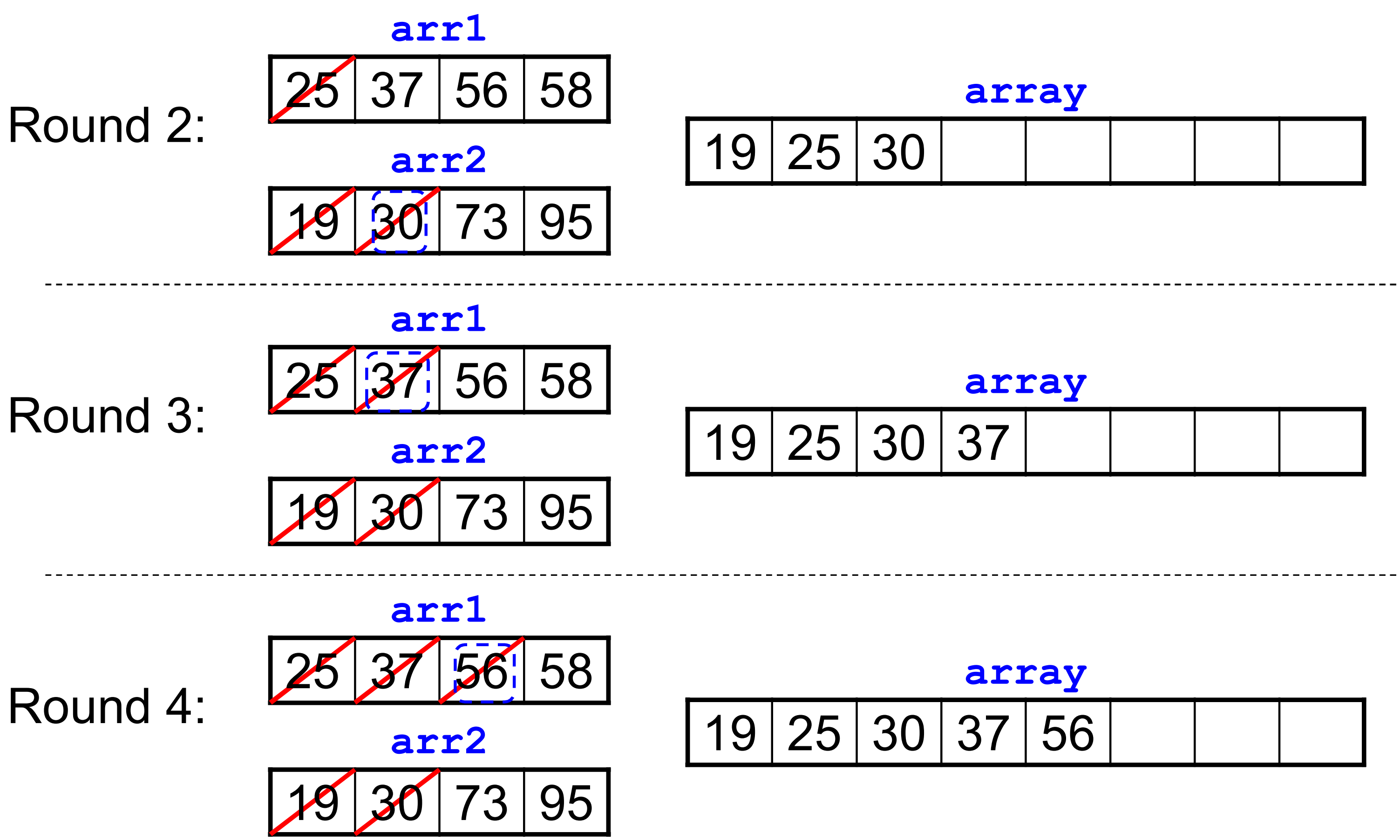
- Merge two *sorted* arrays **`arr1`** and **`arr2`** to one *sorted* array **`array`**.



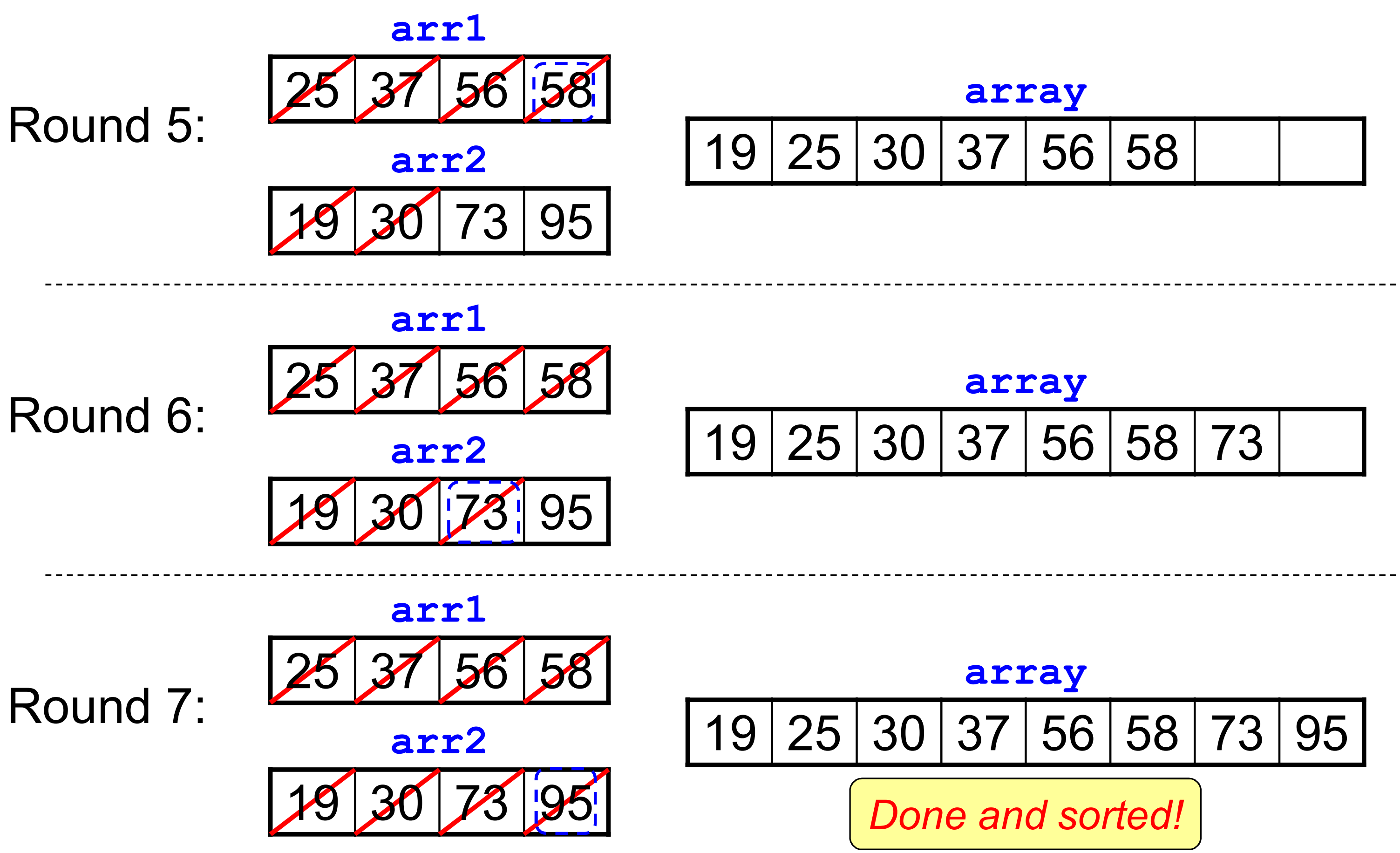
Merging Two Arrays



Merging Two Arrays



Merging Two Arrays



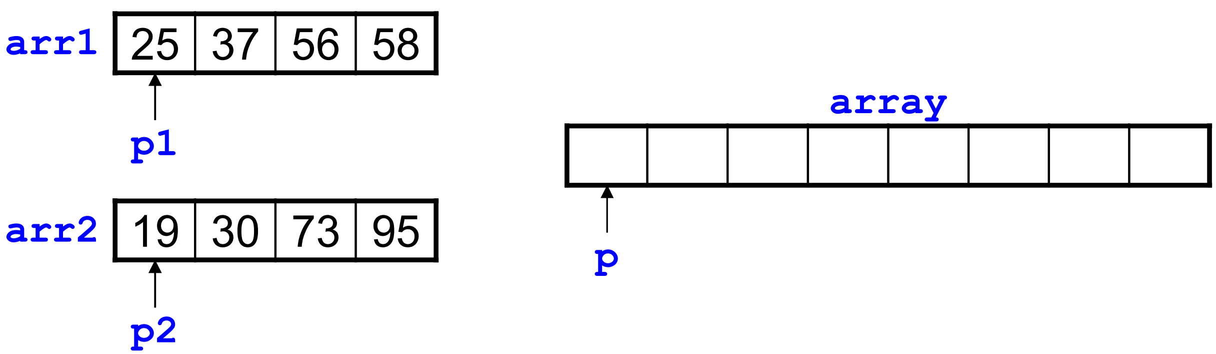
Merging Two Arrays: Implementation (Merge)

```
void Merge(int *array, int *arr1, int n1, int *arr2, int n2) {
   int p1 = 0, p2 = 0, p = 0;

   while (p1 < n1 && p2 < n2) {
      if (arr1[p1] <= arr2[p2])
         array[p++] = arr1[p1++];
      else
         array[p++] = arr2[p2++];
   }

   while (p1 < n1)
      array[p++] = arr1[p1++];

   while (p2 < n2)
      array[p++] = arr2[p2++];
}
```

Merging Two Arrays: Implementation

- **p1**, **p2**, and **p** are indices to the arrays **arr1**, **arr2**, and **array** respectively.

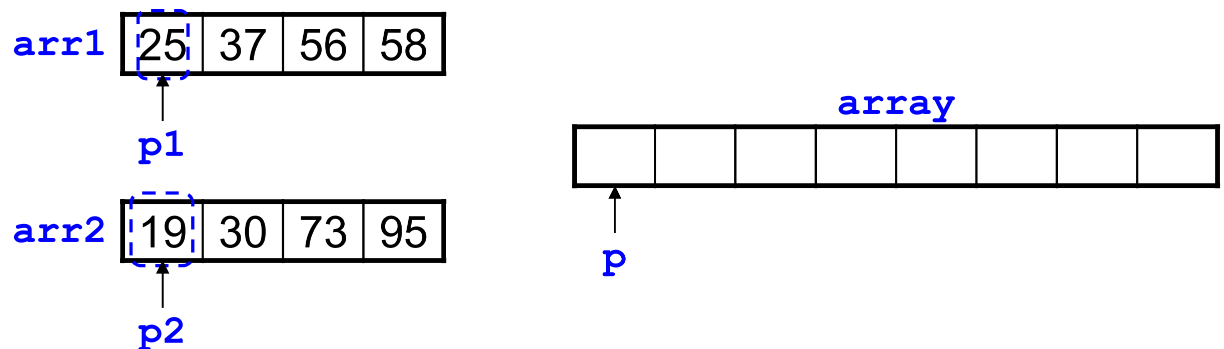
```
int p1 = 0, p2 = 0, p = 0;
while (p1 < n1 && p2 < n2) {
   if (arr1[p1] <= arr2[p2])
      array[p++] = arr1[p1++];
   else
      array[p++] = arr2[p2++];
}
```



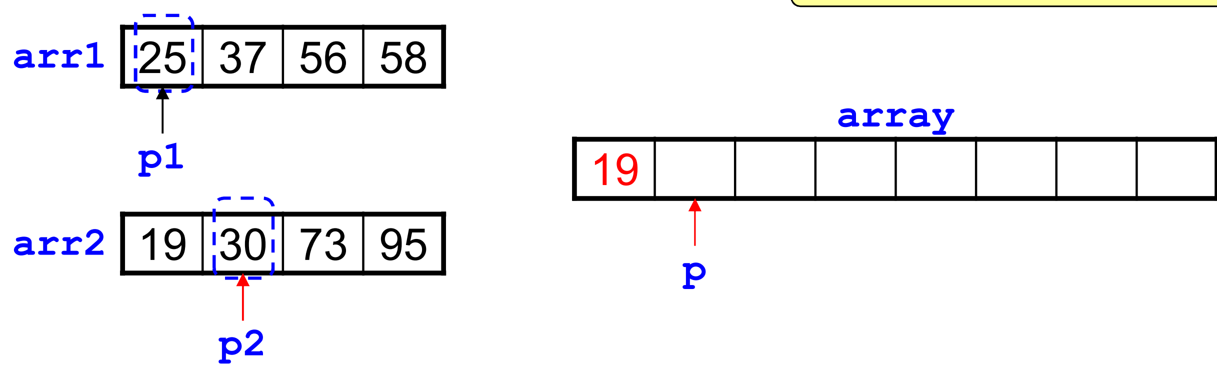
```
while (p1 < n1 && p2 < n2) {
    if (arr1[p1] <= arr2[p2])
        array[p++] = arr1[p1++];
    else
        array[p++] = arr2[p2++];
}
```

Copy either `arr1[p1]` or `arr2[p2]`, whichever smaller, to `array[p]`.

And advance the pointer of that array.



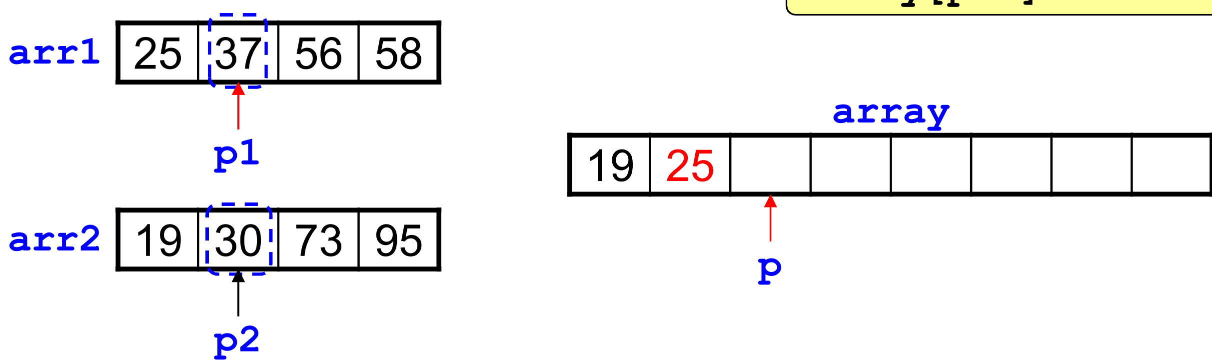
`array[p++] = arr2[p2++];`



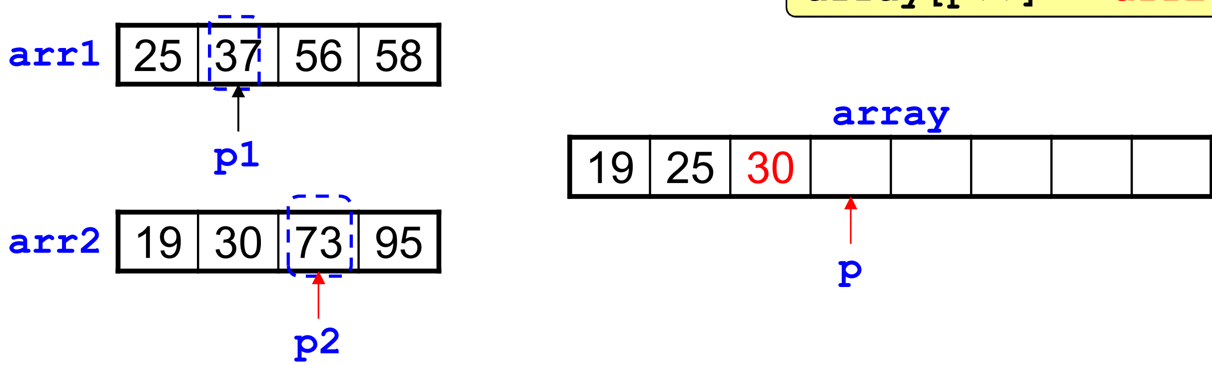
```
while (p1 < n1 && p2 < n2) {
   if (arr1[p1] <= arr2[p2])
      array[p++] = arr1[p1++];
   else
      array[p++] = arr2[p2++];
}
```

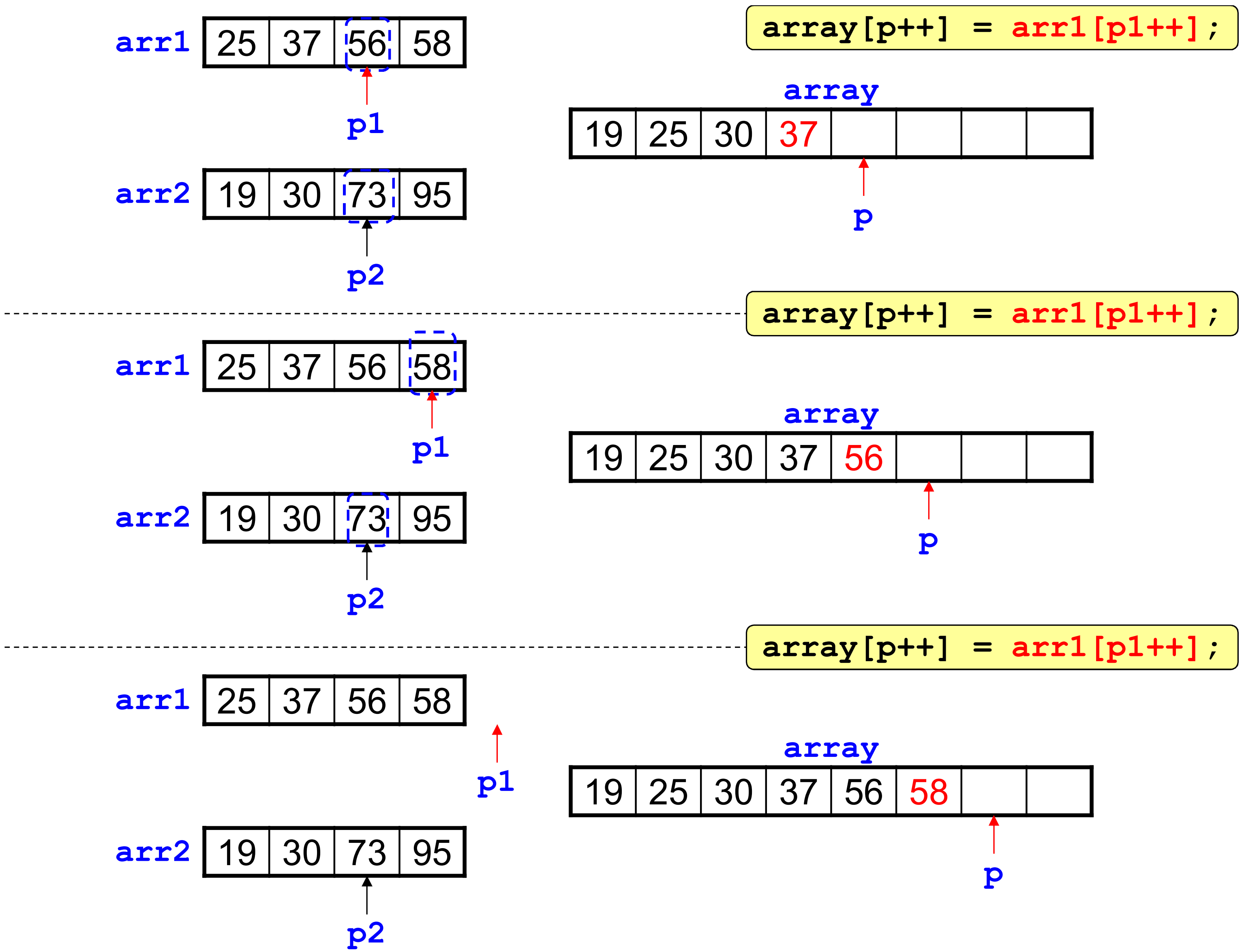
Copy either `arr1[p1]` or `arr2[p2]`, whichever smaller, to `array[p]`.

`array[p++] = arr1[p1++];`



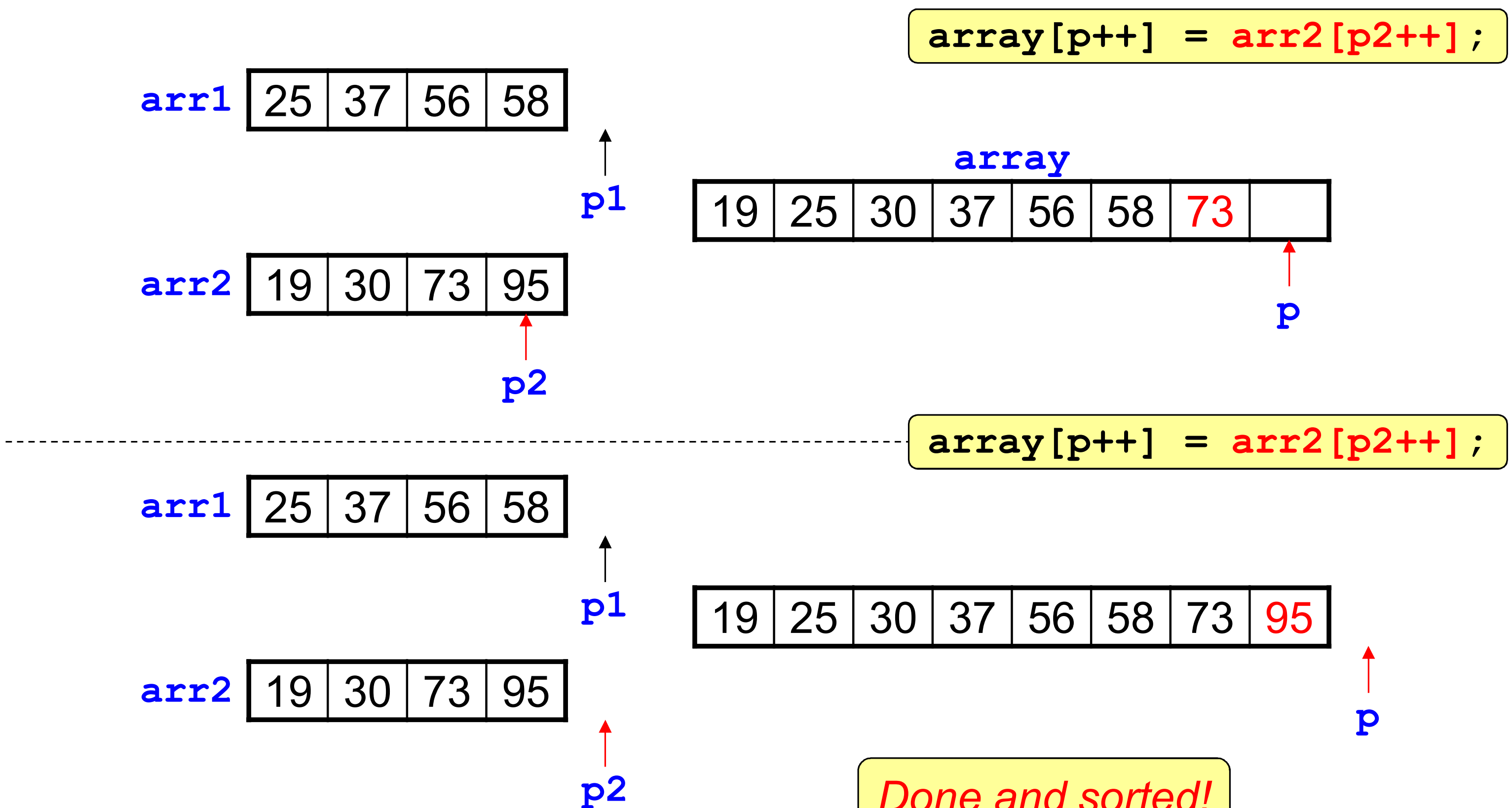
`array[p++] = arr2[p2++];`




array[p++] = arr1[p1++];
arr1
25 37 56 58
p1
arr2
19 30 73 95
p2
array
19 25 30 37
p
array[p++] = arr1[p1++];
arr1
25 37 56 58
p1
arr2
19 30 73 95
p2
array
19 25 30 37 56
p
array[p++] = arr1[p1++];
arr1
25 37 56 58
p1
arr2
19 30 73 95
p2
array
19 25 30 37 56 58
p

```
while (p1 < n1)
   array[p++] = arr1[p1++];

while (p2 < n2)
   array[p++] = arr2[p2++];
```

Flush the remaining elements of either `arr1` or `arr2` to `array`.

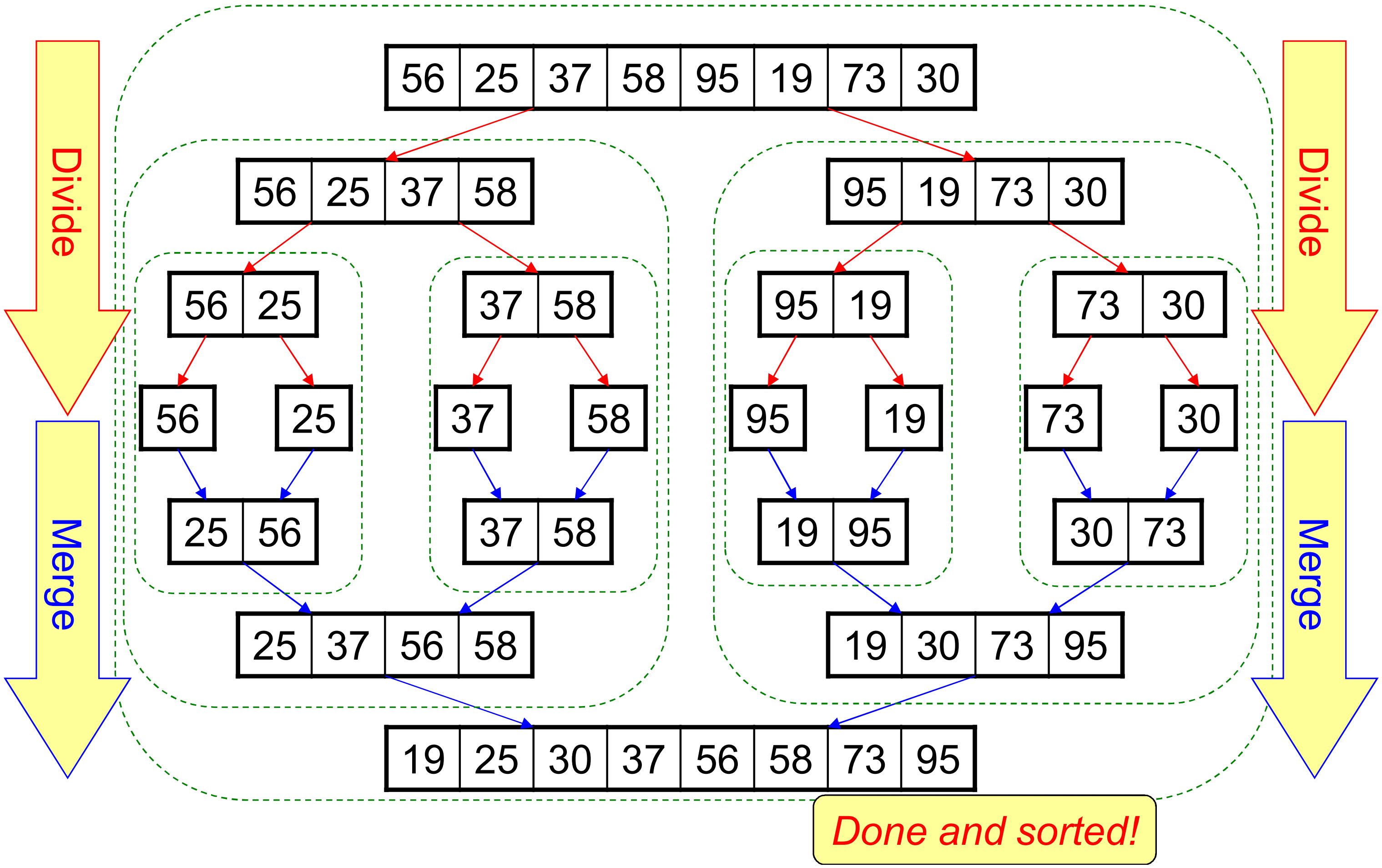


Done and sorted!

Merge Sort Implementation

```
void MergeSort(int *array, int n) {
   int n1, n2, *arr1, *arr2;

   if (n > 1) {
      n1 = n / 2;     // size of 1st subarray
      n2 = n - n1;    // size of 2nd subarray
      arr1 = (int *)malloc(n1 * sizeof(int));
      arr2 = (int *)malloc(n2 * sizeof(int));
      Divide(array, arr1, n1, arr2, n2);
      MergeSort(arr1, n1);
      MergeSort(arr2, n2);
      Merge(array, arr1, n1, arr2, n2);
      free(arr1);
      free(arr2);
   }
}
```

Merge Sort: Example Run



How Efficient is Merge Sort?

- To know the time complexity of merge sort, we first need to know those of **`Divide`** and **`Merge`**.

```
void Divide(int *array, int *arr1, int n1, int *arr2, int n2) {
   int i;
   for (i = 0; i < n1; i++)
      arr1[i] = array[i];
   for (i = 0; i < n2; i++)
      arr2[i] = array[i + n1];
}
```

`n1` iterations

`n2` iterations

`n1` + `n2` = N.

Thus, time complexity is O(N).

Efficiency of Merge Sort

```
void Merge(int array, int *arr1, int n1, int *arr2, int n2) {
   int p1 = 0, p2 = 0, p = 0;

   while (p1 < n1 && p2 < n2) {
      if (arr1[p1] <= arr2[p2])
         array[p++] = arr1[p1++];
      else
         array[p++] = arr2[p2++];
   }

   while (p1 < n1)
      array[p++] = arr1[p1++];

   while (p2 < n2)
      array[p++] = arr2[p2++];
}
```

These statements are executed exactly N times in *all* loops.

Thus, time complexity is O(N).

Efficiency of Merge Sort

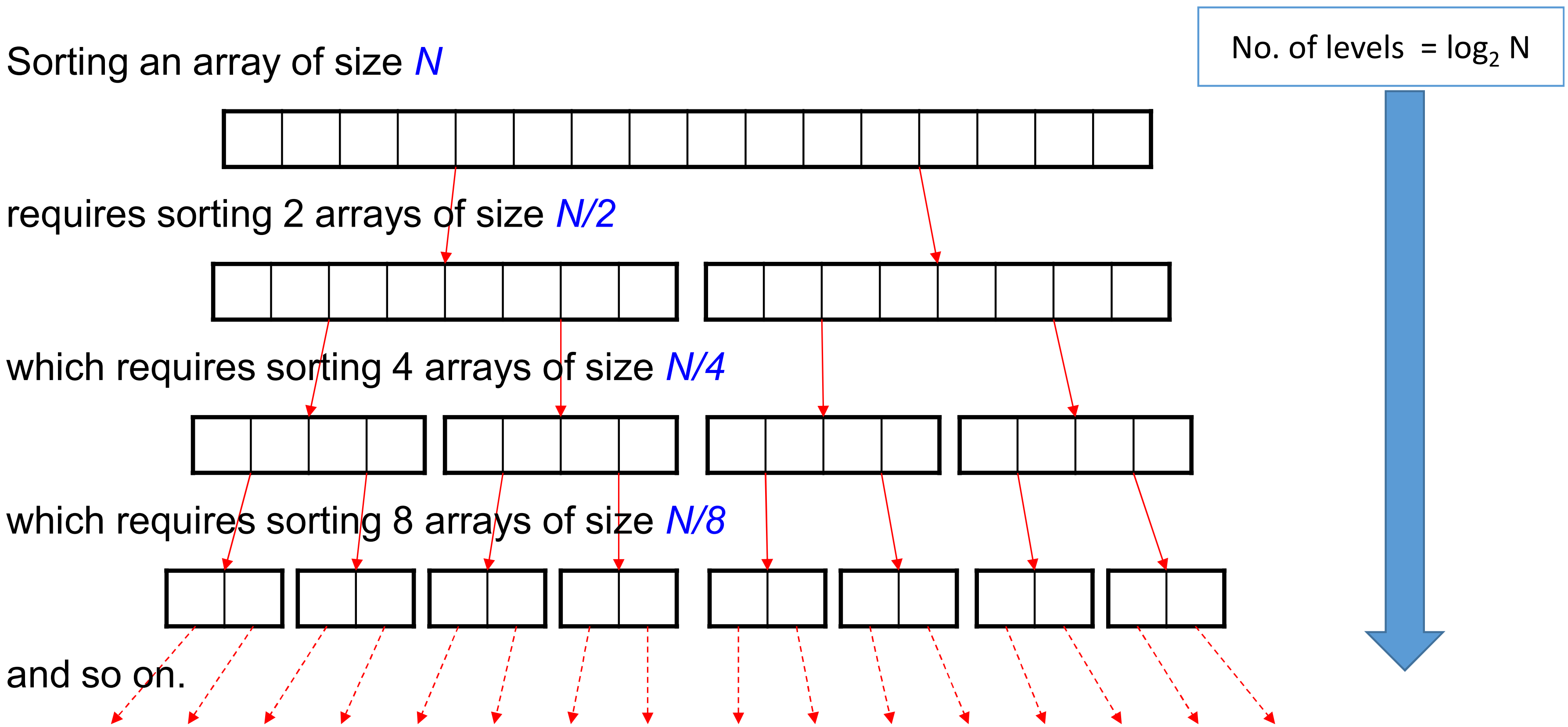
```
void MergeSort(int *array, int n) {
   int n1, n2, *arr1, *arr2;

   if (n > 1) {
      n1 = n / 2;     // size of 1st subarray
      n2 = n - n1;    // size of 2nd subarray
      arr1 = (int *)malloc(n1 * sizeof(int));
      arr2 = (int *)malloc(n2 * sizeof(int));
      Divide(array, arr1, n1, arr2, n2);
      MergeSort(arr1, n1);
      MergeSort(arr2, n2);
      Merge(array, arr1, n1, arr2, n2);
      free(arr1);
      free(arr2);
   }
}
```

O(N)

O(?)

O(N)

Recursive Decomposition



The time spent at each level is *always* O(N).

Computational Complexity of Merge Sort

- The time spent at each level is $O(N)$.
- There are totally k levels where $2^k = N$, i.e.,

$$k = \log_2 N$$

- Hence, the time complexity of merge sort is

$$O(N \log_2 N)$$

- We usually omit the logarithmic base in writing complexities. That is, we usually write:

$$O(N \log N)$$

N^2 vs N log N

- N log N grows much slower than N^2.

N	N^2	N log N
10	100	33
100	10,000	664
1,000	1,000,000	9,966
10,000	100,000,000	132,877
100,000	10,000,000,000	1,660,964

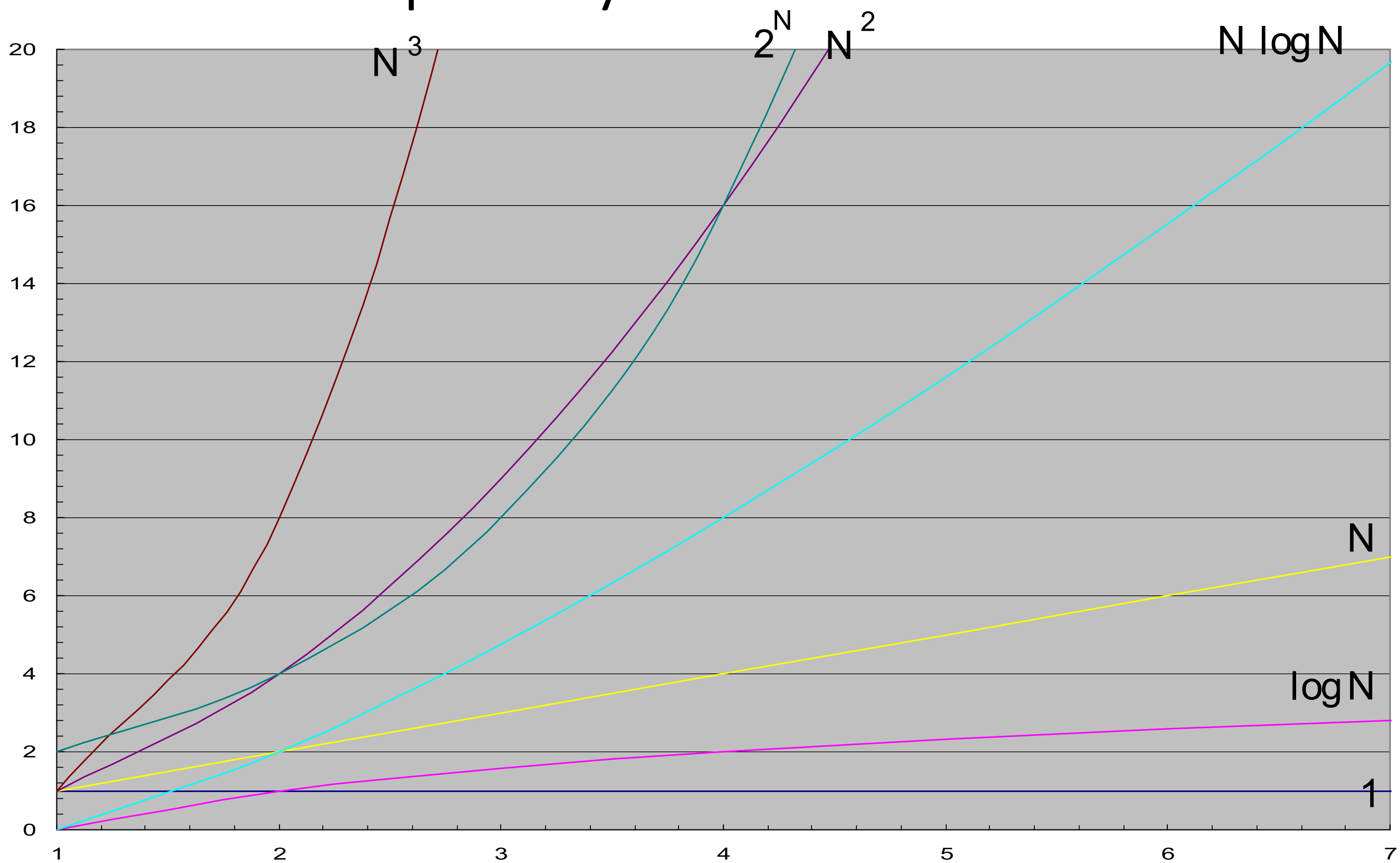
Selection Sort vs Merge Sort

N	S. Sort Time (s)	M. Sort Time (s)
10,000	0.265	< 0.001
20,000	1.061	0.015
40,000	4.260	0.031
100,000	26.797	0.062
110,000	32.557	0.078
120,000	38.721	0.093
140,000	54.142	0.093
200,000	110.032	0.109
1,000,000	2663.429	0.625

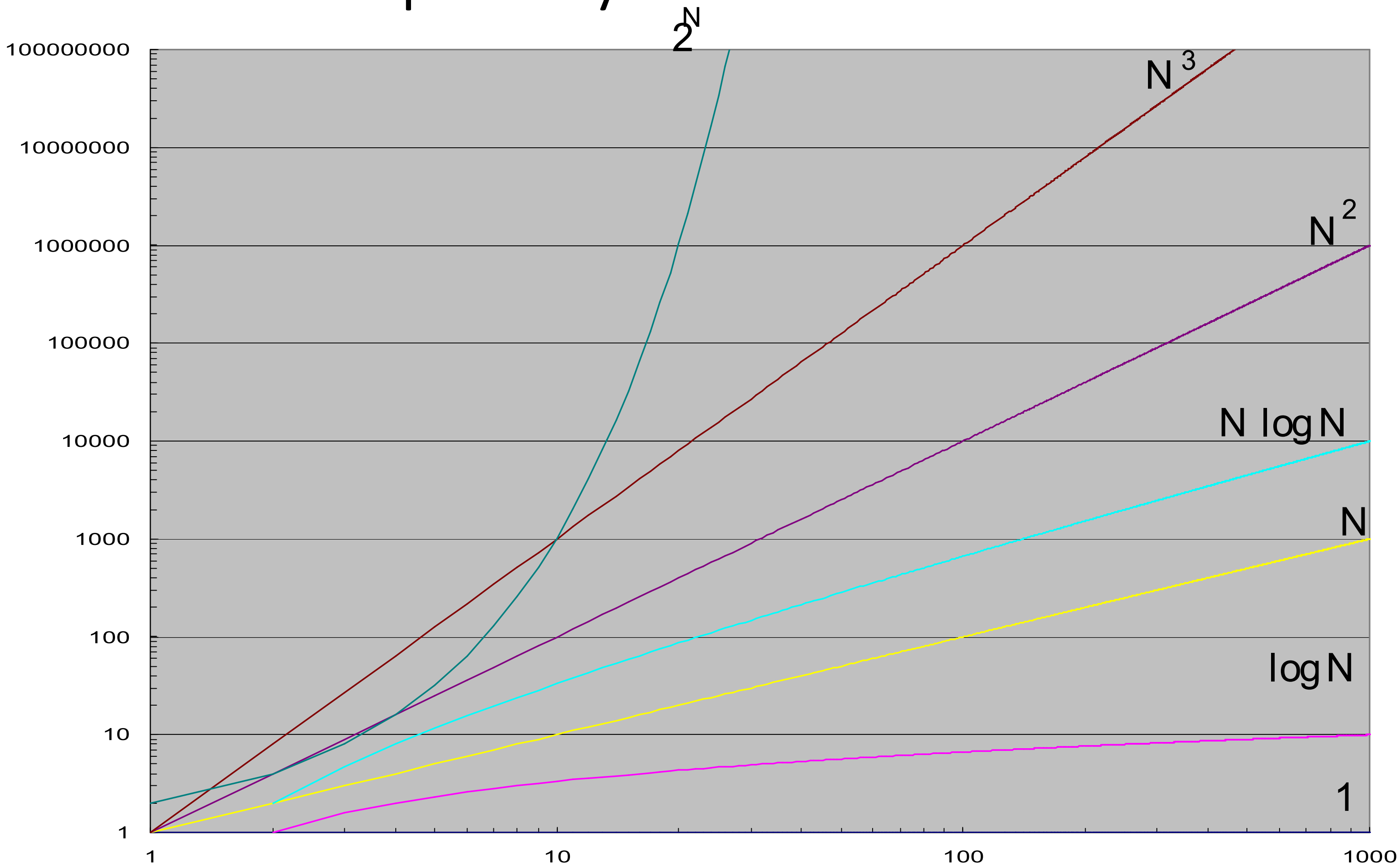
Common Complexity Classes

Big-O	Name	Example
$O(1)$	constant	Returning the 1st element in an array
$O(\log N)$	logarithmic	Binary search in a sorted array
$O(N)$	linear	Linear search in an array
$O(N \log N)$	N log N	Merge sort
$O(N^2)$	quadratic	Selection sort
$O(N^3)$	cubic	Conventional matrix multiplication
$O(2^N)$	exponential	Tower of Hanoi

Common Complexity Classes



Common Complexity Classes

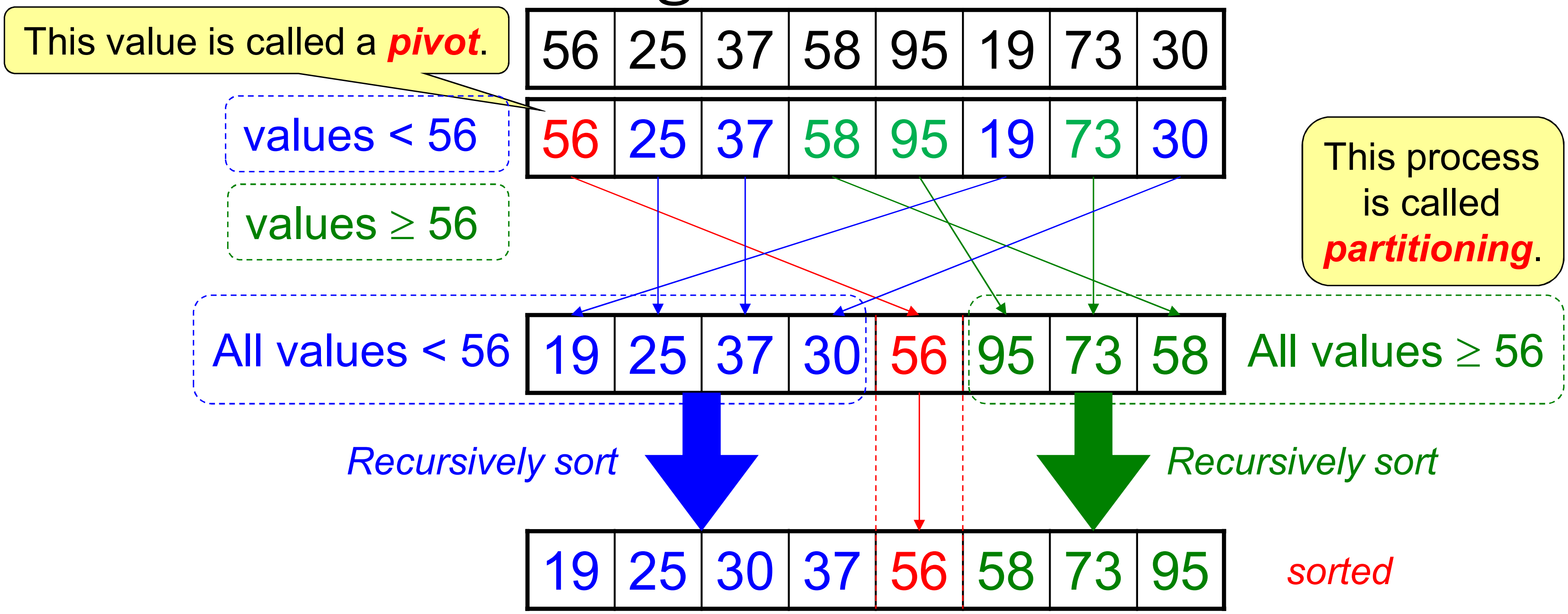


Quicksort

- Besides merge sort, ***Quicksort*** is another sorting algorithm that makes use of recursion.

- It mainly differs from merge sort in the way to dividing up the array.

The Quicksort Algorithm



The Quicksort Algorithm

- An array of size 0 or 1 must already be sorted.
- Otherwise,
  - Choose a *pivot* to rearrange the array elements so that
    - *small* values are moved toward the *beginning*
    - *large* values are moved toward the *end*;
  - Recursively sort each subarray.

The Quicksort Algorithm

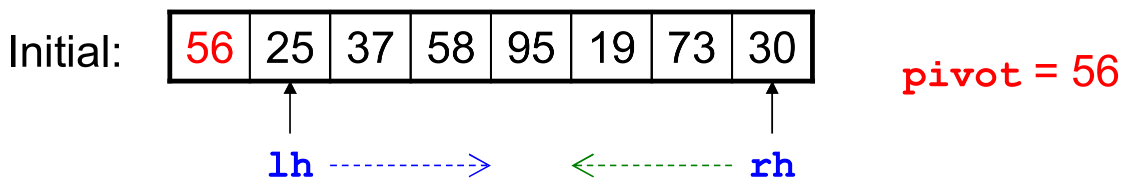
- Unlike merge sort, Quicksort does *not* require
  - merging the subarrays;
  - using additional space to store the subarrays.

- *Any* array element can be chosen as the pivot. The simplest choice is the first element in the array.

56	25	37	58	95	19	73	30

Partitioning an Array

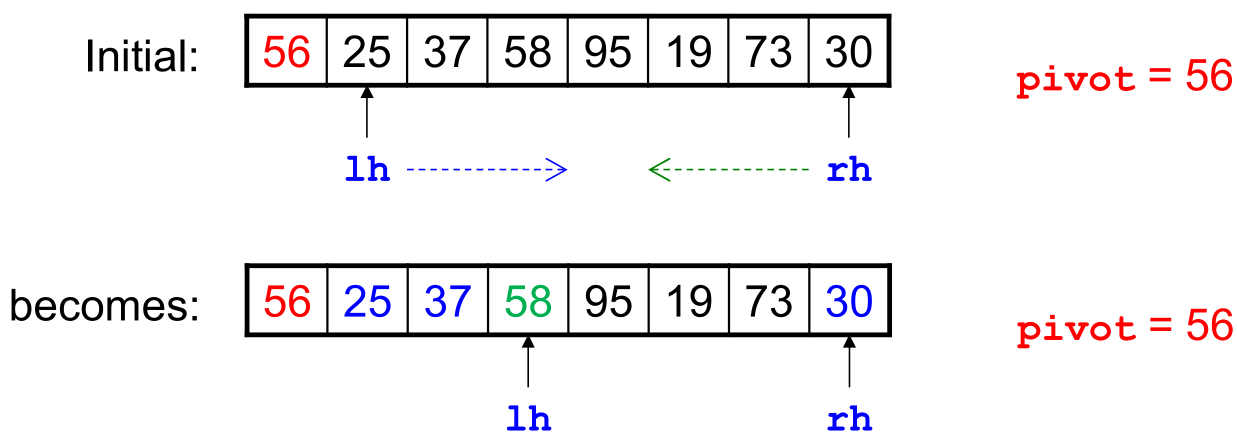
- We use 2 indices **`lh`** and **`rh`** to the array.



- During partitioning, **`rh`** moves left and **`lh`** moves right until they meet each other.

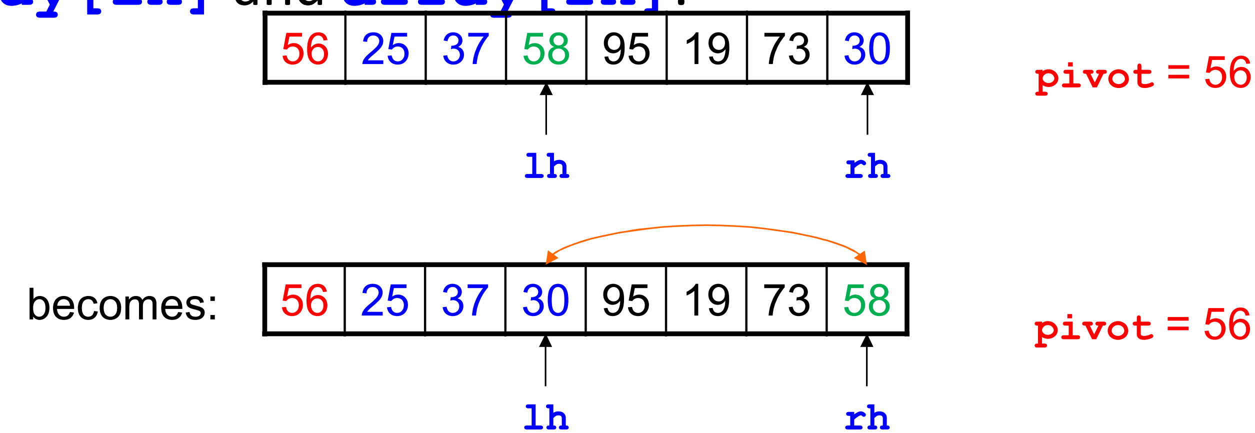
Partitioning an Array

- **rh**: moves left until **array[rh]** < **pivot** or meets **lh**.
- **lh**: moves right until **array[lh]** ≥ **pivot** or meets **rh**.



Partitioning an Array

- Swap **`array[lh]`** and **`array[rh]`**.



- Repeat the two processes (moving and swapping) until **`lh`** meets **`rh`**.

`rh` moves left until `array[rh]` < `pivot` or meets `lh`.
`lh` moves right until `array[lh]` ≥ `pivot` or meets `rh`.

- Move **`rh`** and **`lh`**.

56	25	37	30	95	19	73	58

`lh` - - - -> <- - - - - `rh`

`pivot` = 56

becomes:

56	25	37	30	95	19	73	58

`lh` `rh`

`pivot` = 56

- Swap **`array[lh]`** and **`array[rh]`**.

becomes:

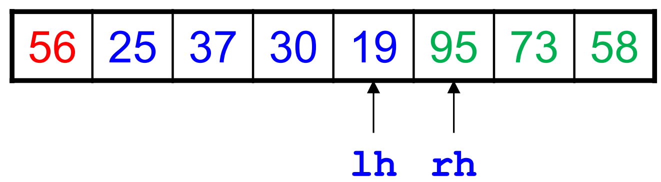
56	25	37	30	19	95	73	58

`lh` `rh`

`pivot` = 56

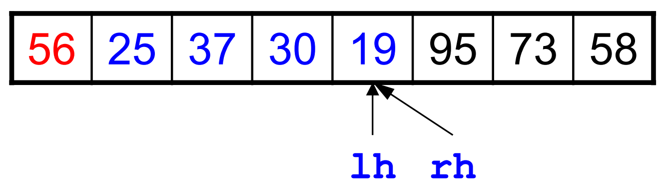
rh moves left until **array[rh]** < **pivot** or meets **lh**.
lh moves right until **array[lh]** ≥ **pivot** or meets **rh**.

- Move **rh** and **lh**.



pivot = 56

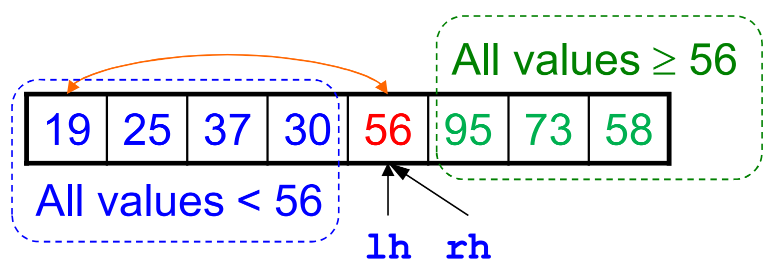
becomes:



pivot = 56

- **lh** meets **rh**. Now swap **array[lh]** and **array[0]** (the pivot).

becomes:



pivot = 56

```
int Partition(int *array, int n) {
   int lh = 1, rh, pivot, temp;

   pivot = array[0];
   rh = n - 1;
   do {
      while (lh < rh && array[rh] >= pivot)
         rh--;
      while (lh < rh && array[lh] < pivot)
         lh++;
      if (lh != rh) {
         temp = array[lh];
         array[lh] = array[rh];
         array[rh] = temp;
      }
   } while (lh != rh);
   if (array[lh] >= pivot)
      return 0;
   array[0] = array[lh];
   array[lh] = pivot;
   return lh;
}
```

`rh` moves left. `lh` moves right.

Swap `array[lh]` and `array[rh]`.

Exit loop when `lh == rh`.

Special case: pivot is the smallest element.

Swap `array[0]` and `array[lh]`.

Returns the position of the pivot.

Quicksort Implementation

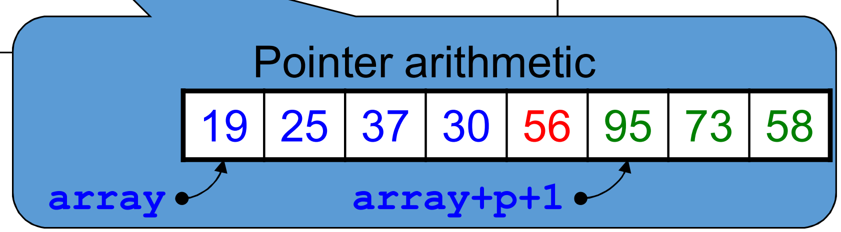
- Using the **Partition** function, Quicksort can be implemented very easily.

```
void Quicksort(int *array, int n) {
    int p;

    if (n > 1) {
        p = Partition(array, n);
        Quicksort(array, p);
        Quicksort(array + p + 1, n - p - 1);
    }
}
```

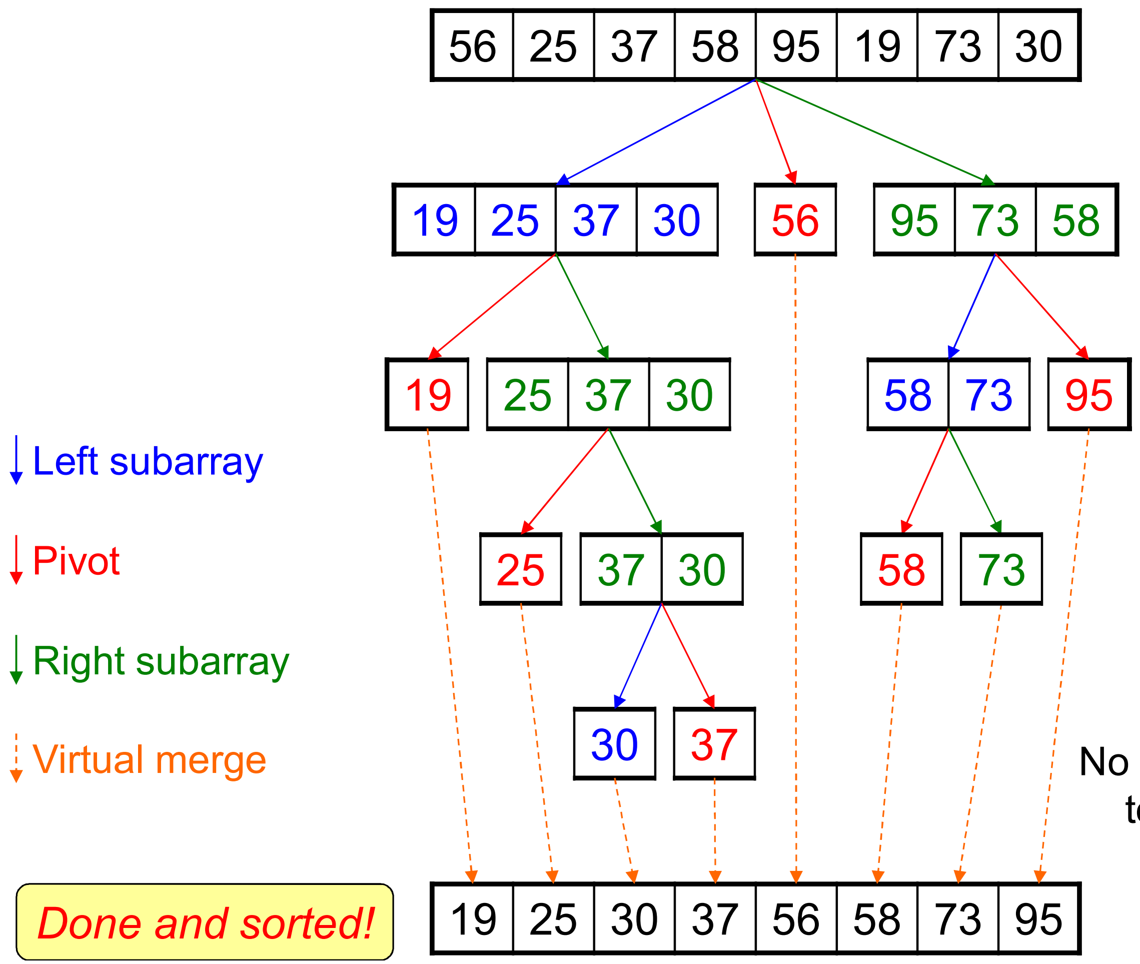
A *test* to stop or continue

Two *recursive calls*

An *end case* (nothing to do)



Quicksort: Example Run



No actual computation
to combine result

Computational Complexity of Quicksort

- To know the time complexity of Quicksort, we first need to know that of **`Partition`**.

```
int Partition(int *array, int n) {
   int lh = 1, rh, pivot, temp;

   pivot = array[0];
   rh = n - 1;
   do {
      while (lh < rh && array[rh] >= pivot)
         rh--;
      while (lh < rh && array[lh] < pivot)
         lh++;
      if (lh != rh) {
         temp = array[lh];
         array[lh] = array[rh];
         array[rh] = temp;
      }
   } while (lh != rh);
   if (array[lh] >= pivot)
      return 0;
   array[0] = array[lh];
   array[lh] = pivot;
   return lh;
}
```

This loop is executed most frequently.

They are totally executed N - 2 = O(N) times.

19	25	37	30	56	95	73	58

`lh` **`rh`**

Thus, computational complexity is O(N).

Computational Complexity of Quicksort

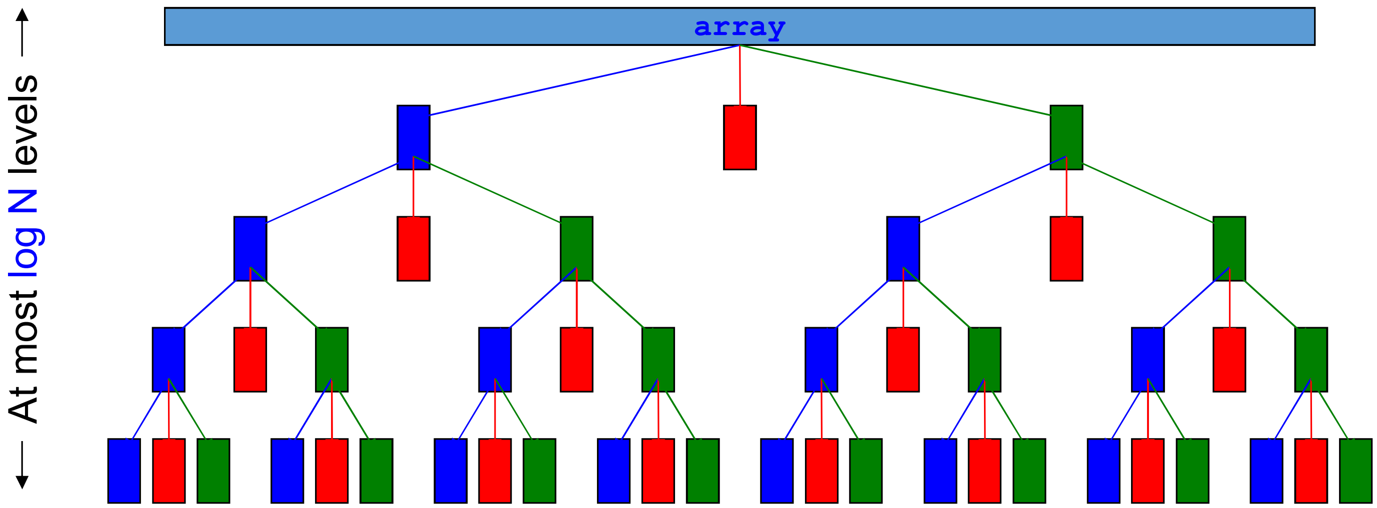
```
void Quicksort(int *array, int n) {
   int p;

   if (n > 1) {
      p = Partition(array, n);
      Quicksort(array, p);
      Quicksort(array + p + 1, n - p - 1);
   }
}
```

O(N) time

O(?) time

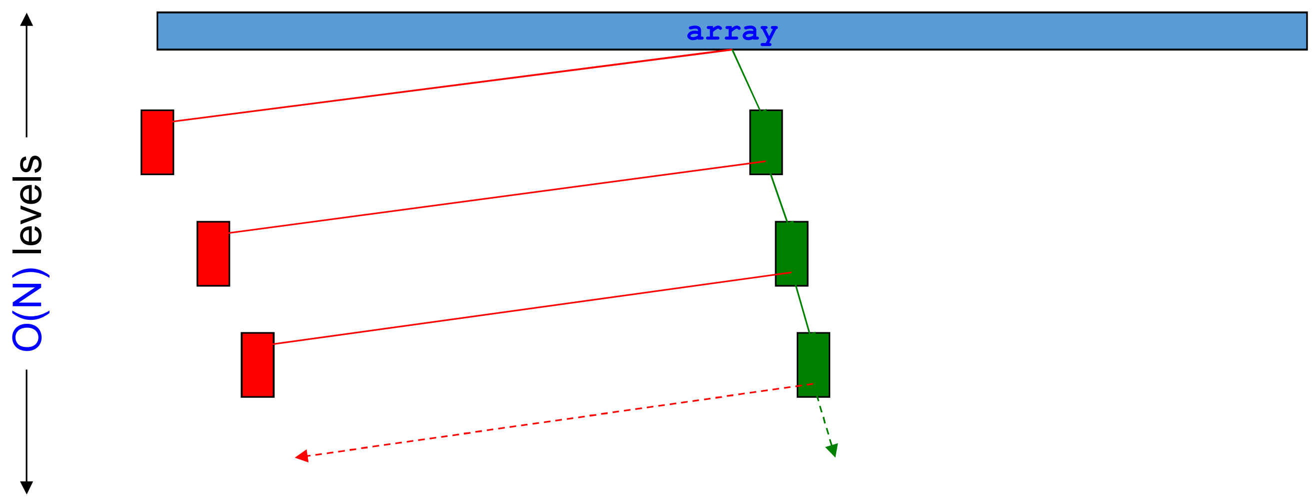
Quicksort: Average Case

- In *average* case, the left and right subarrays have equal size.
- There are at most log N levels. Each level executes in O(N) time. Thus, *average case* time complexity is O(N log N).



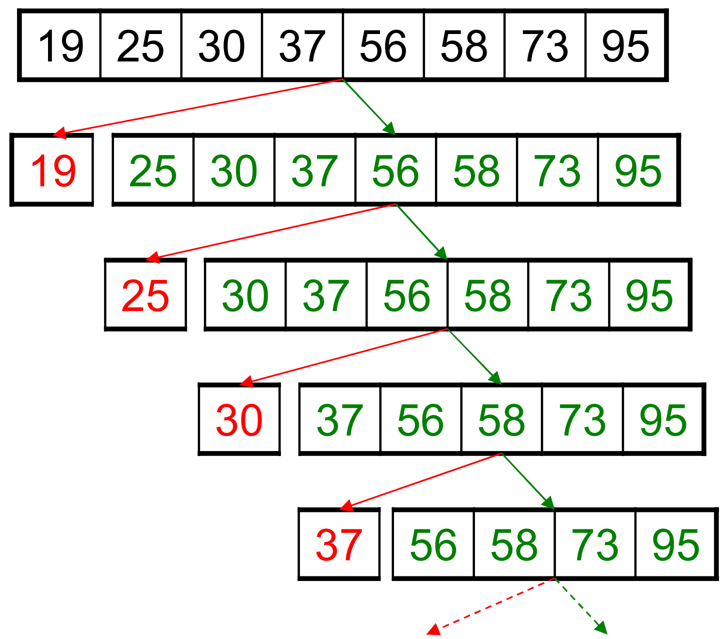
Quicksort: Worst Case

- In the *worst* case, one subarray is much larger than the other.
- There are O(N) levels. Each level executes in O(N) time. Thus, *worst case* time complexity is O(N^2).



Quicksort: Worst Case

- The worst case happens when the input array is already sorted.



Selection Sort, Merge Sort, and Quicksort

- In practice, Quicksort is faster than most other sorting algorithms.

N	Sel. Sort Time (s)	Merge Sort Time (s)	Quicksort Time (s)
40,000	4.260	0.031	0.015
100,000	26.797	0.062	0.015
200,000	110.032	0.109	0.046
400,000	444.961	0.235	0.094
1,000,000	2663.429	0.625	0.313
2,000,000	> 10000	1.296	0.766