8 - QUICKSORT

Topics

- Partitioning
- Quicksort – using partitioning
  - Choosing the Pivot
  - Analysis of Quicksort
  - Picking a better Pivot
    - Median of Three

3

Partitioning

Partitioning

- Partitioning is the underlying mechanism of quicksort.
- To *partition* data is to divide it into two groups, so that all the items with a key value higher than a specified amount are in one group, and all the items with a lower key value are in another.
  - Divide students into those with grade point averages higher and lower than 3.5, so as to know who deserves to be on the Dean's list.

Partitioning

- (See Workshop applet for "Partition")
- Pivot value
  - Items with a key value less than the pivot value go in the left part of the array,
  - and those with a greater (or equal) key go in the right part.
- The arrow labeled *partition* points to the leftmost item in the right (higher) subarray.
  - This value is returned from the partitioning method
- After being partitioned, the data is by no means sorted; it has simply been divided into two groups.

Partitioning

- partition(left, right, pivot)

```
1. l = left - 1, r = right + 1; // initialize pointers "l" and "r"
2. while true, do
   2.1. while l < right AND a[++l] < pivot ; // nop
   2.2. while r > left  AND a[--r] >= pivot ; // nop
   2.3. if l >= r, break;            // pointers cross
   2.4. else swap a[l] and a[r] // swap elements
3. Return l                 // return partition location "l"
4. Terminate
```

Partitioning – Java Code

- Listing 7.2, Partition.java, page 327

Efficiency of Partitioning Algorithm

- Partitioning has a complexity of O (N)

Quicksort – Using Partitioning

Quicksort

- Quicksort is the most popular sorting algorithm.
- In majority of situations, it's the fastest, operating in O(N*logN) time.
- To understand quicksort, you should be familiar with the partitioning algorithm.
- Basically, the quicksort algorithm operates by partitioning an array into two subarrays and then calling itself recursively to quicksort each of these subarrays.
- Selection of "pivot" is one important aspect.

Quicksort I

- To sort a[left...right]:
- Quicksort(a[left...right]):

1. if left < right:

   1.1. Partition a[left...right] such that:

   all a[left...p-1] are less than a[p], and

   all a[p+1...right] are >= a[p]

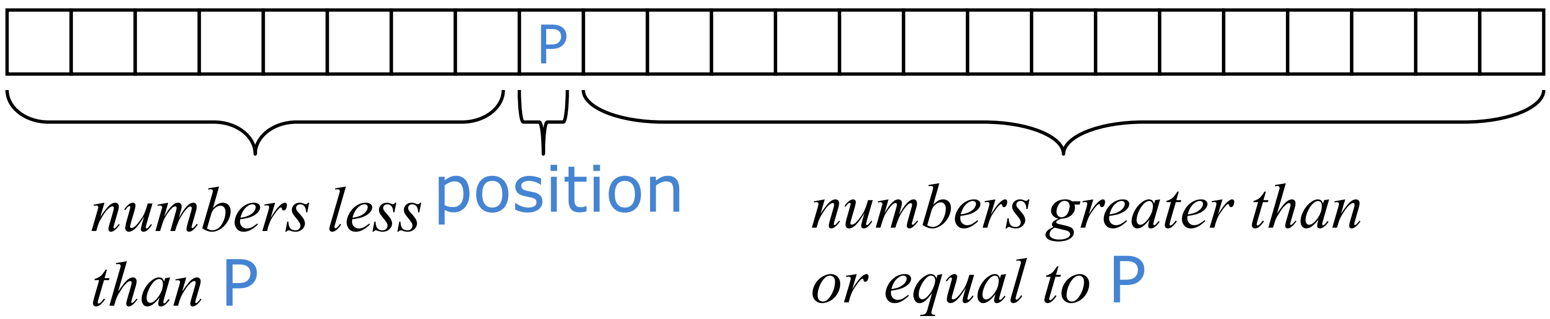
   1.2. Quicksort a[left...p-1]

   1.3. Quicksort a[p+1...right]

2. Terminate

Partitioning (Quicksort II)

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number P in the array to use as a pivot
  - We partition the array into three parts:



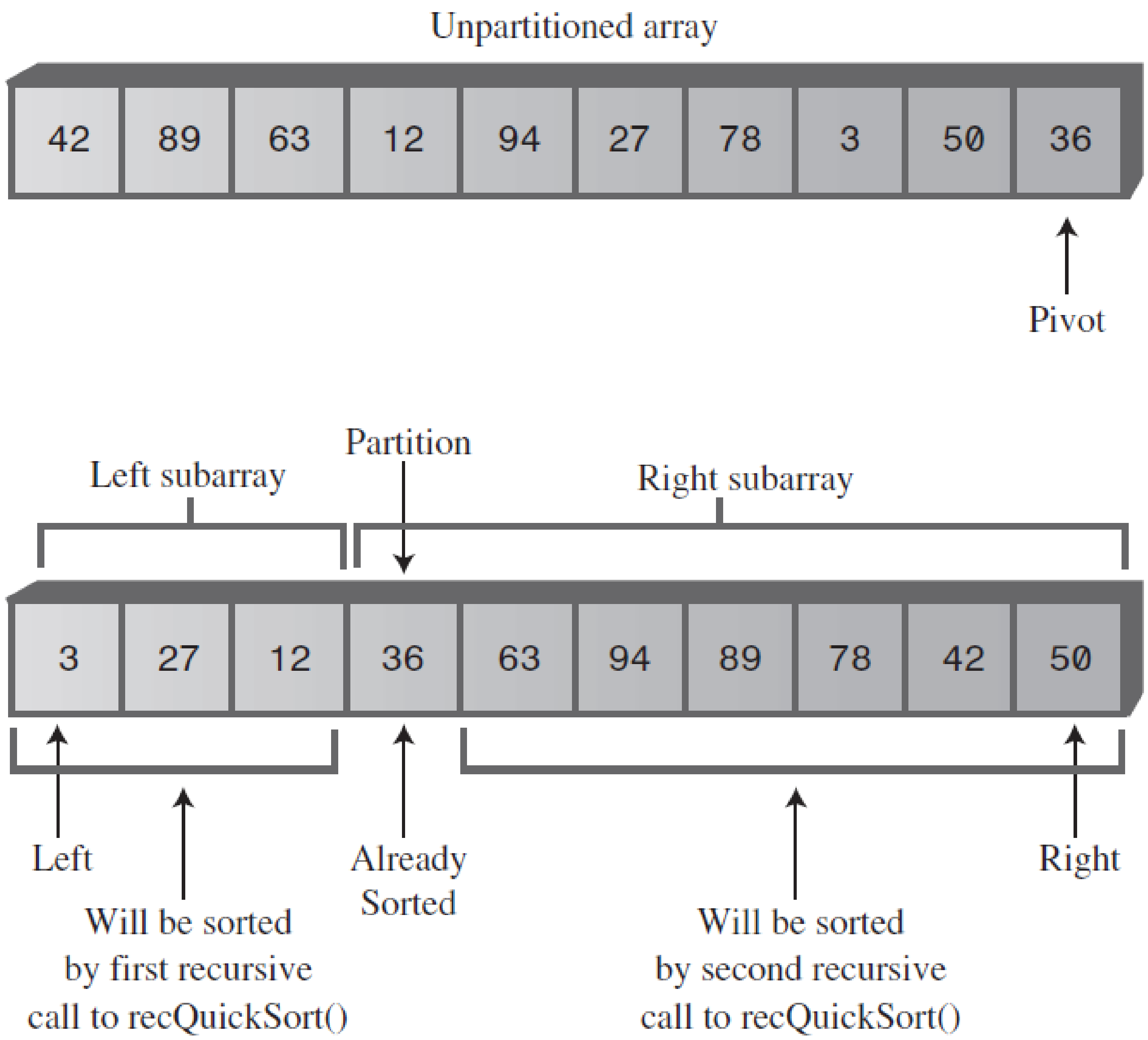
Partitioning II

- Choose an array value (say, the rightmost element) to use as the pivot
- Starting from the left end, find the first element that is greater than or equal to the pivot
- Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done

Choosing the Pivot

Partitioning - Choosing the Pivot

- The pivot value should be the key value of an actual data item; this item is called the *pivot*.
- You can pick a data item to be the pivot more or less at random.
- For simplicity, let's say we always pick the item on the right end of the subarray being partitioned.
- After the partition, the pivot needs to be placed into its proper place, between the left and right subarrays.
  - Pivot should be Swapped with the left item in the right subarray.



Recursive calls sort subarrays.

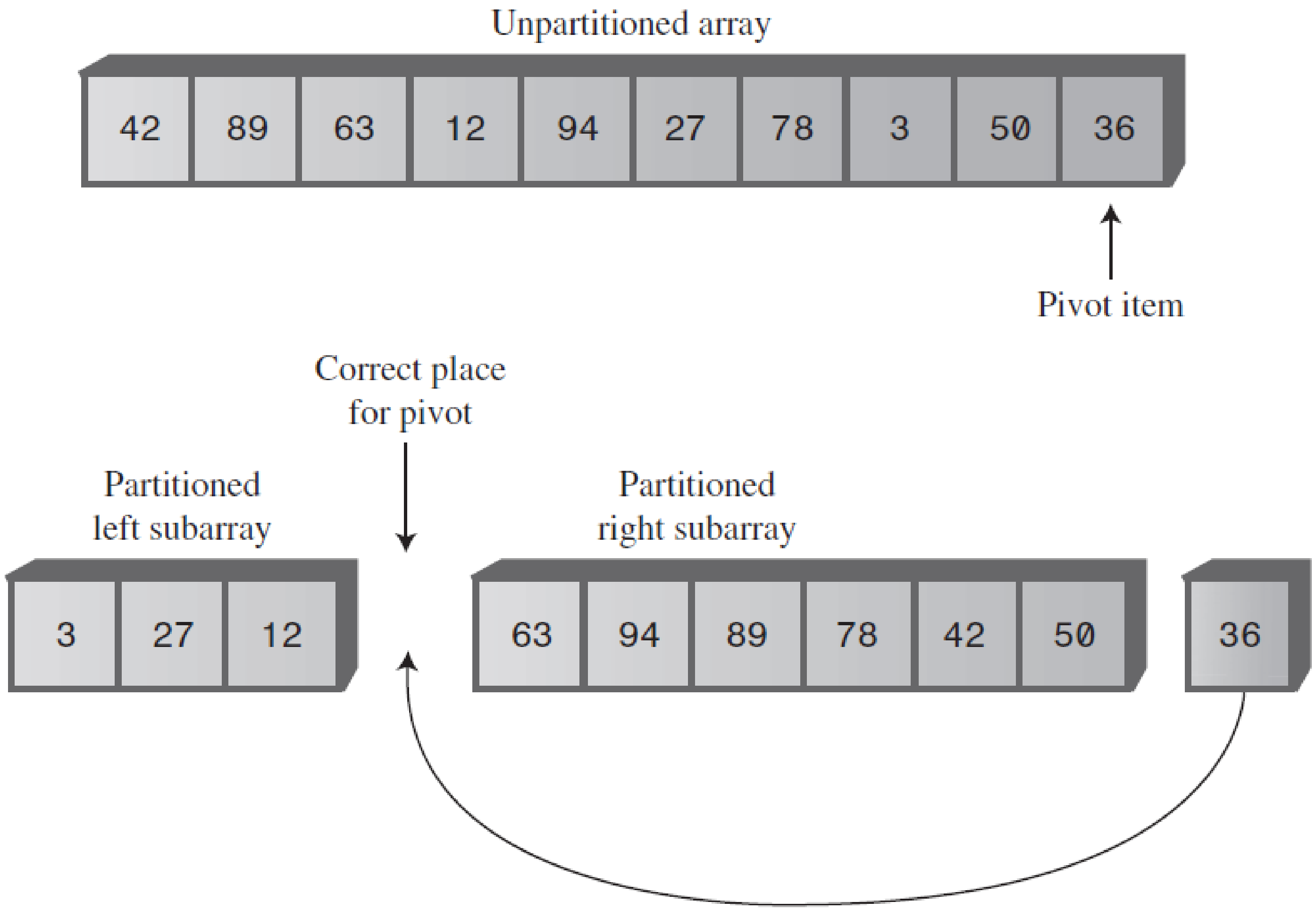


FIGURE 7.9 The pivot and the subarrays.

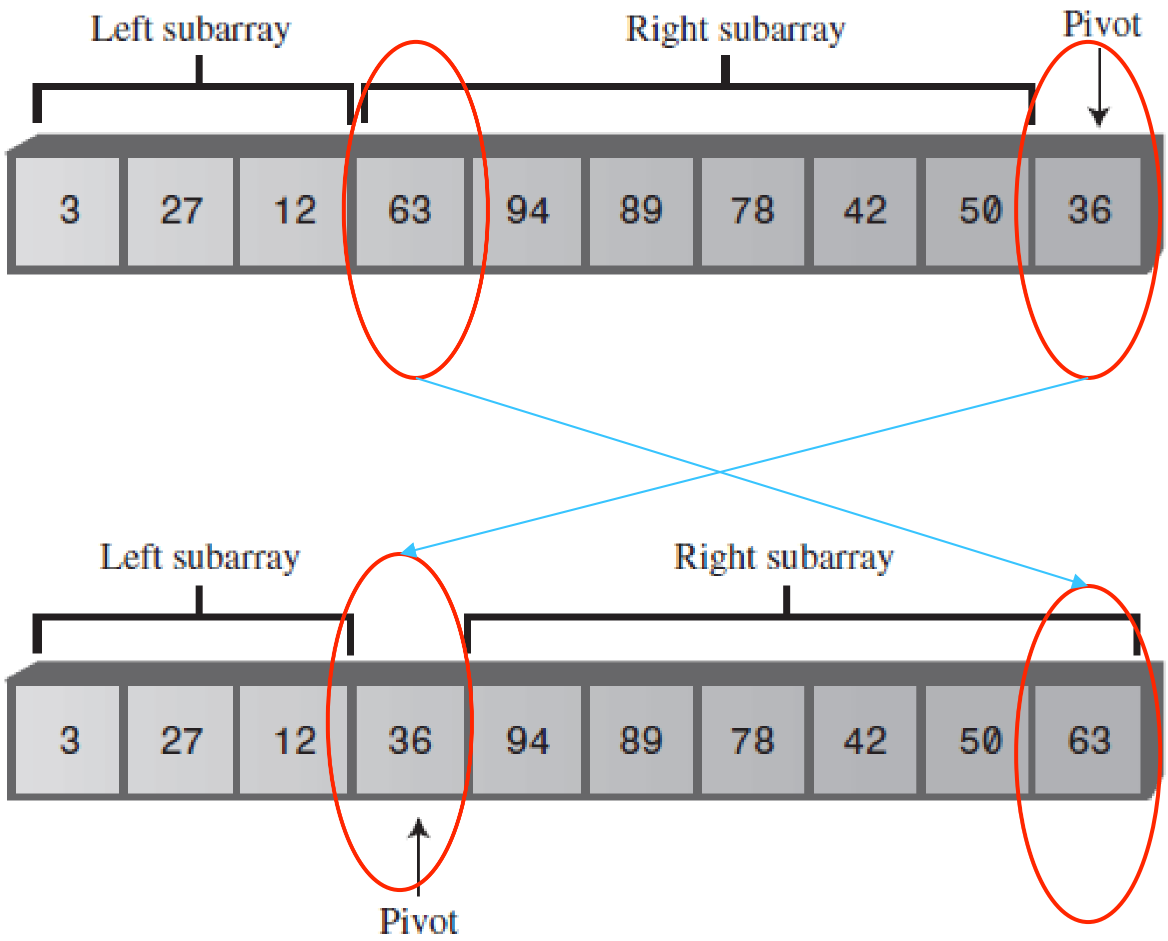


FIGURE 7.10 Swapping the pivot.

Partitioning with Pivot selection

- To partition a[left…right] and to select the pivot:

1. Set pivot = a[right], l = left - 1, r = right; // init pointers
2. while true, do
   2.1. while l < right AND a[++l] < pivot ; // nop
   2.2. while r > left AND a[--r] >= pivot ; // nop
   2.3. if l >= r, break; // pointers cross
   2.4. else swap a[l] and a[r] // swap elements
3. Swap a[right] and a[l] // swap pivot with element "l"
4. Return l // return "l"
5. Terminate

Example of partitioning

- choose pivot: 3 4 9 2 7 3 1 2 1 8 9 3 5 6 4
- search: 3 4 9 2 7 3 1 2 1 8 9 3 5 6 4
- swap: 3 3 9 2 7 3 1 2 1 8 9 4 5 6 4
- search: 3 3 9 2 7 3 1 2 1 8 9 4 5 6 4
- swap: 3 3 1 2 7 3 1 2 9 8 9 4 5 6 4
- search: 3 3 1 2 7 3 1 2 9 8 9 4 5 6 4
- swap: 3 3 1 2 2 3 1 7 9 8 9 4 5 6 4
- search: 3 3 1 2 2 3 1 7 9 8 9 4 5 6 4 (left > right)
- swap with pivot: 3 3 1 2 2 3 1 4 9 8 9 4 5 6 7

The partition method (Java)

```
static int partition(int a[], int left, int right, long p) {
    int l = left - 1, r = right; // p is pivot
    while (true) {
        while (l < right && a[++l] < p);
        while (r > left  && a[--r] >= p);
        if (l >= r)  break;
        else {
            int temp = a[l]; a[l] = a[r]; a[r] = temp;
        }
    }
    a[right] = a[l];
    a[l] = p;
    return l;
}
```

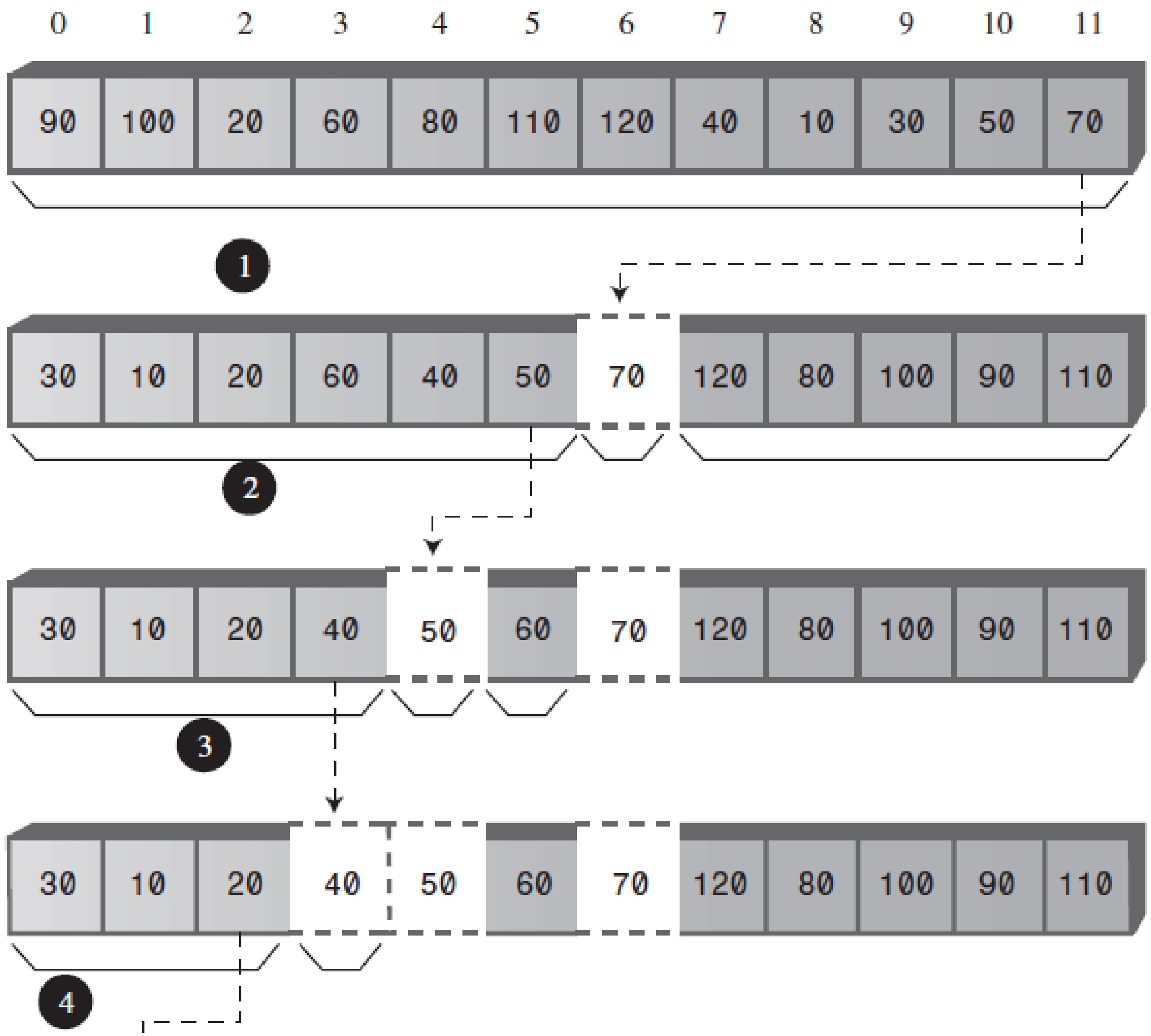
The quicksort method (in Java)

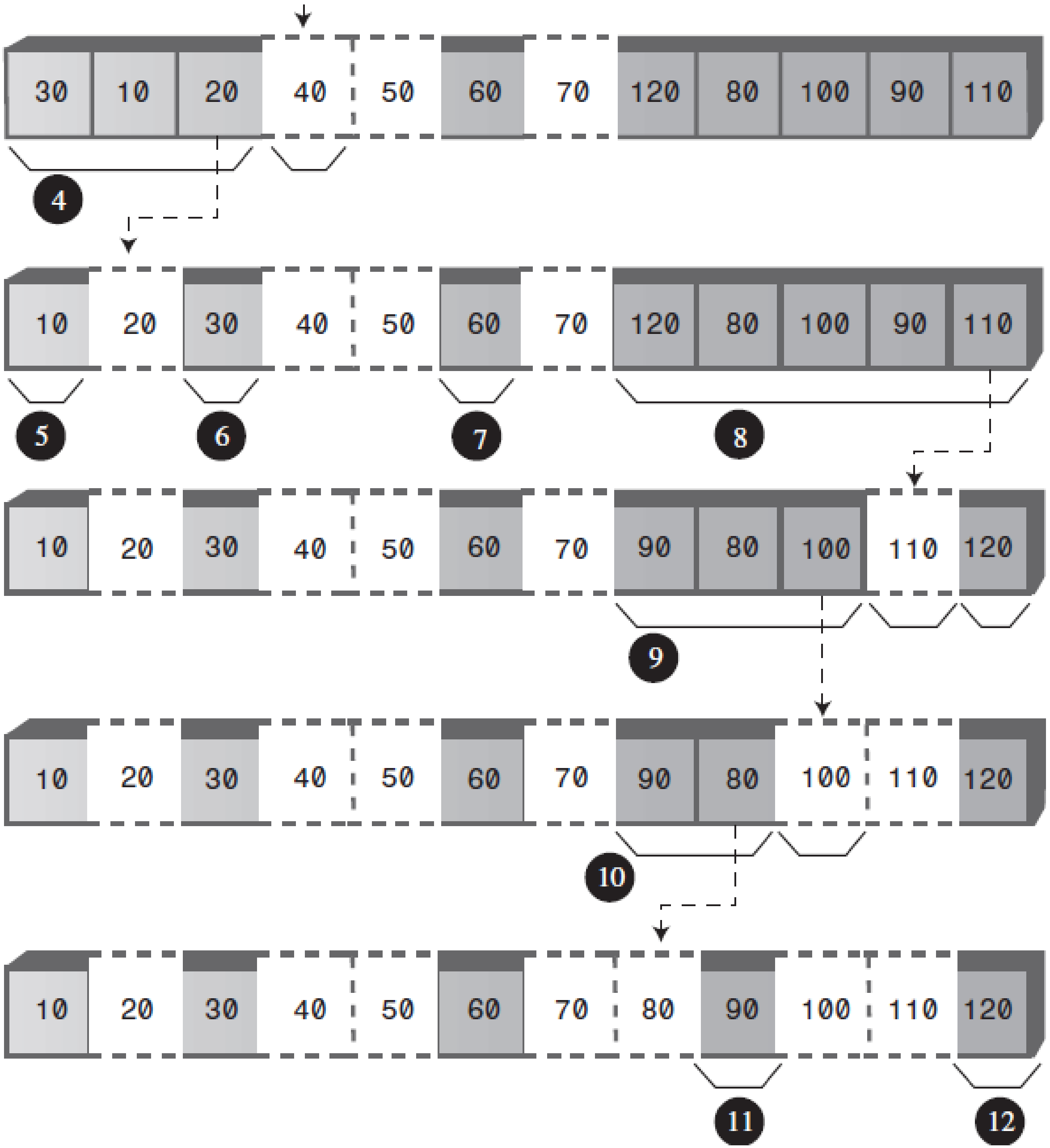
- Listing 7.3, quicksort1.java, page 337

```
static void quicksort(int[] array, int left, int right) {
  if (right – left <= 0)   // if size <= 1, already sorted (base)
       return;
  else {     // size = 2 or larger (recursive)
       int pivot = array[right];
       int p = partition(array, left, right, pivot);
       quicksort(array, left, p - 1);
       quicksort(array, p + 1, right);
    }
}
```

Quicksort in Action

- See Workshop applet “Quicksort1”


0 1 2 3 4 5 6 7 8 9 10 11
90 100 20 60 80 110 120 40 10 30 50 70
1
30 10 20 60 40 50 70 120 80 100 90 110
2
30 10 20 40 50 60 70 120 80 100 90 110
3
30 10 20 40 50 60 70 120 80 100 90 110
4


30 10 20 40 50 60 70 120 80 100 90 110
4
10 20 30 40 50 60 70 120 80 100 90 110
5
6
7
8
10 20 30 40 50 60 70 90 80 100 110 120
9
10 20 30 40 50 60 70 90 80 100 110 120
10
10 20 30 40 50 60 70 80 90 100 110 120
11
12

Analysis of Quicksort

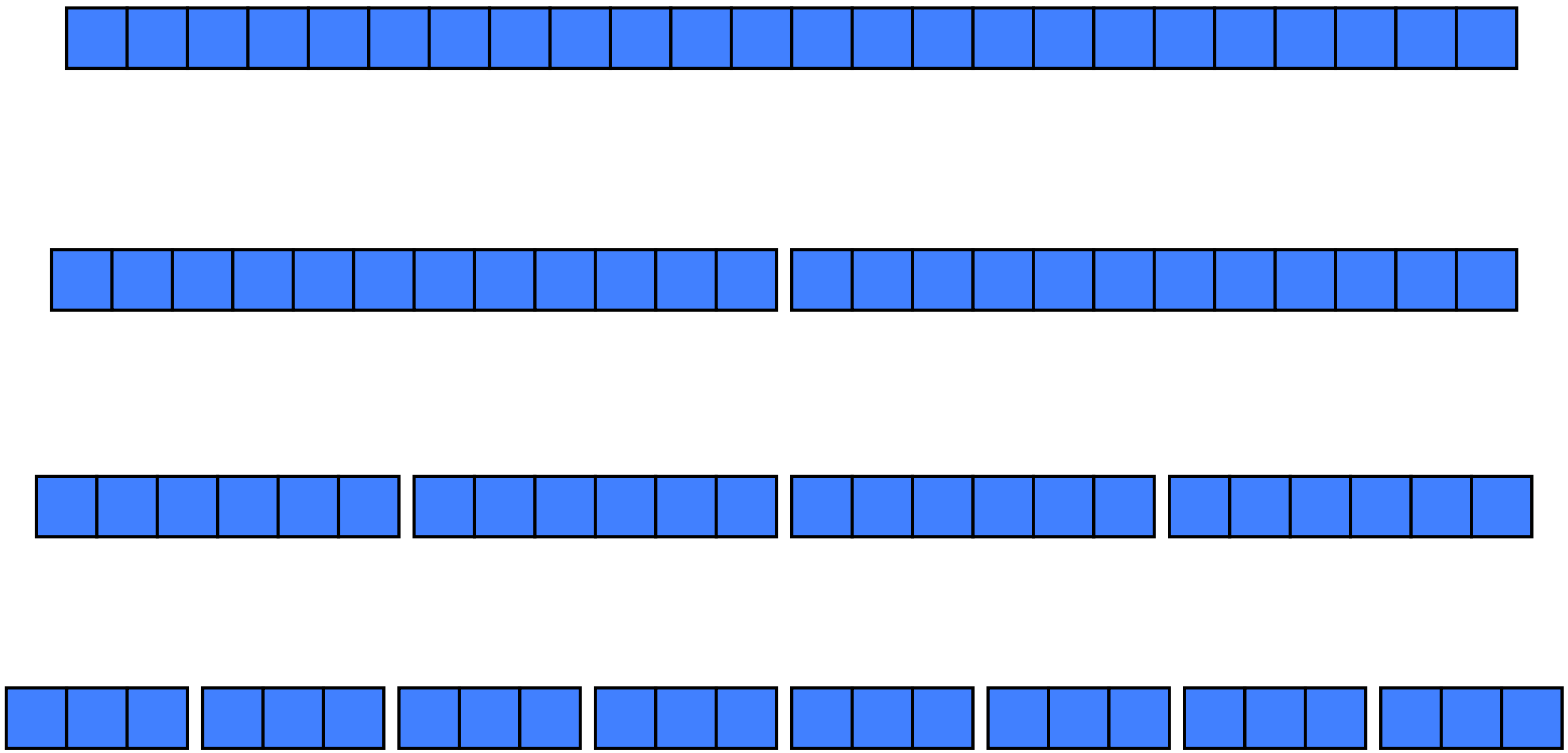
Analysis of quicksort—Best case I

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion is $\log_2 n$
  - Because that's how many times we can halve n
- However, there are many recursions!
  - How can we figure this out?
  - We note that
    - Each partition is linear over its subarray
    - All the partitions at one level cover the array

Best case II

- We cut the array size in half each time
- So the depth of the recursion is $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the average case, quicksort has time complexity $O(n \log_2 n)$

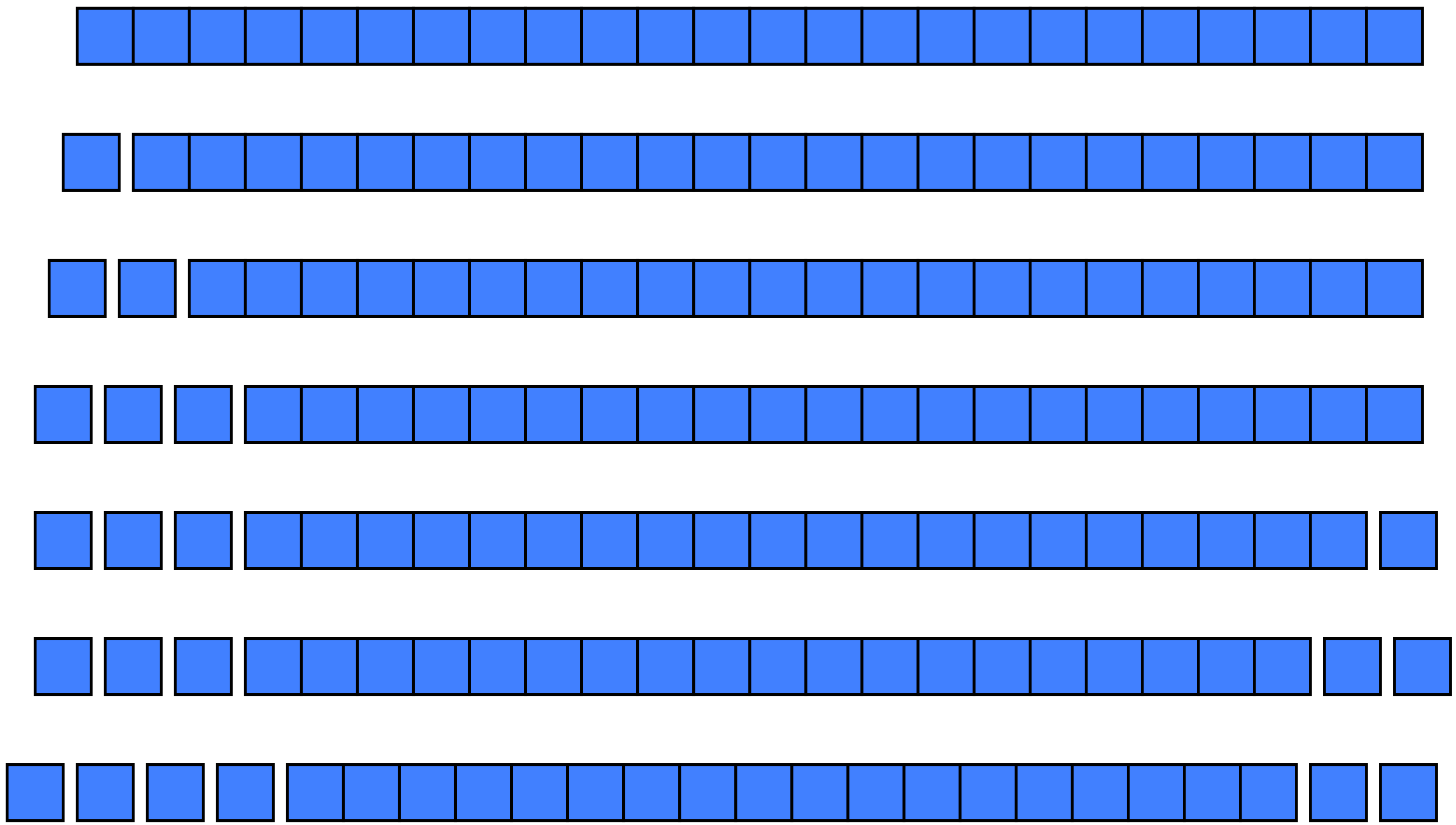
Partitioning at various levels – Best Case



Worst case

- What about the worst case?
- In the worst case, partitioning always divides the size N array into these two parts:
  - one subarray with 1 element (the pivot), and
  - one subarray with N-1 elements.
- We don't recur on the subarray with 1 element.
- Recurring on the length N-1 part requires (in the worst case) recurring to depth N-1

Worst case partitioning



Worst case for quicksort

- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - There are many arrangements that *could* make this happen
  - Here are two common cases:
    - When the array is already sorted
    - When the array is *inversely* sorted (sorted in the opposite order)

Typical case for quicksort

- If the array is sorted to begin with, Quicksort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n \log_2 n)$
- The constants are so good that Quicksort is generally the fastest algorithm known
- Most real-world sorting is done by Quicksort

Tweaking Quicksort

- Almost anything you can try to "improve" Quicksort will actually slow it down
- One *good* tweak is to switch to a different sorting method when the subarrays get small (say, 10 or 12)
  - Quicksort has too much overhead for small array sizes
- For large arrays, it *might* be a good idea to check beforehand if the array is already sorted
  - But there is a better tweak than this

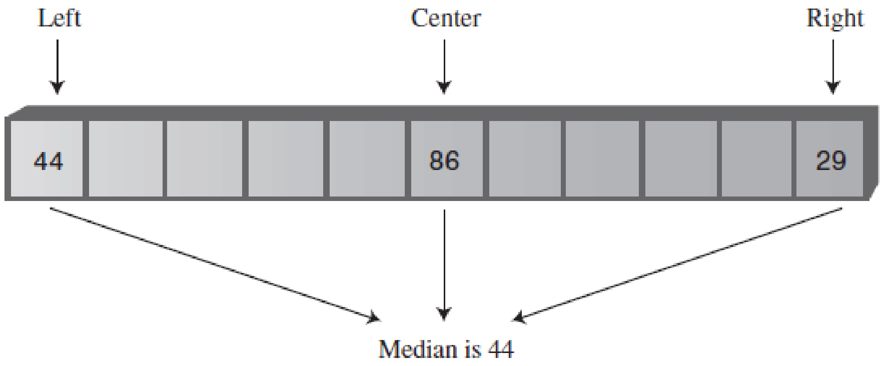
Picking a better Pivot

Picking a better pivot

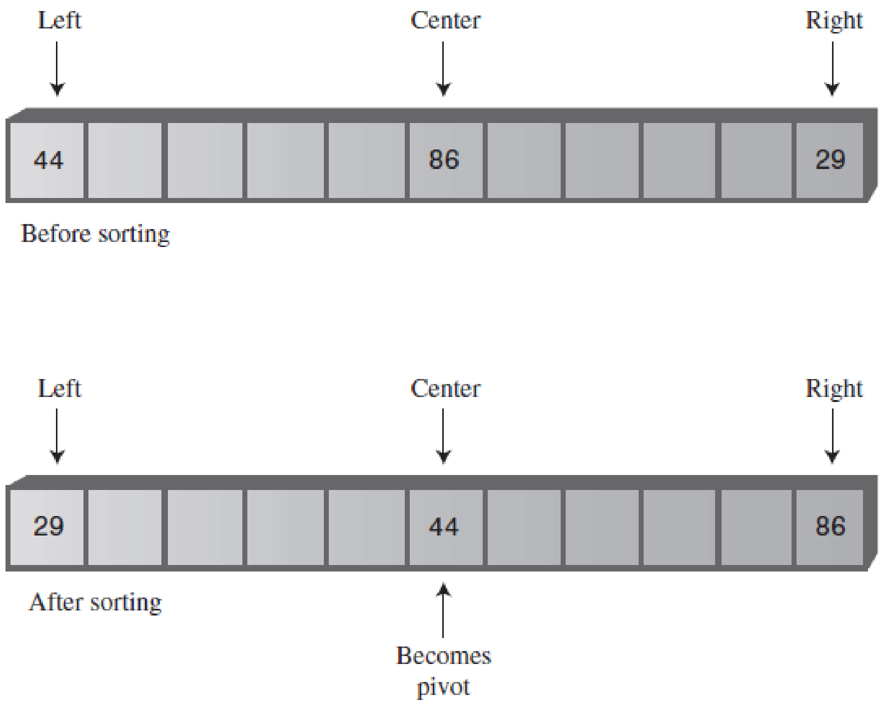
- Before, we picked the last element of the subarray to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the first element
- We could do an *optimal* quicksort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a median: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

Median of three

- Obviously, it doesn't make sense to sort the array in order to find the median to use as a pivot
- Instead, compare just *three* elements of our (sub)array—the first, the last, and the middle
  - Take the *median* (middle value) of these three as pivot
  - It's possible (but not easy) to construct cases which will make this technique $O(n^2)$
- Suppose we rearrange (sort) these three numbers so that the smallest is in the first position, the largest in the last position, and the other in the middle
  - This lets us simplify and speed up the partition loop



The median of three.



Sorting the left, center, and right elements.

Final comments

- Quicksort is the fastest known sorting algorithm
- For optimum efficiency, the pivot must be chosen carefully
- "Median of three" is a good technique for choosing the pivot
- However, no matter what you do, there will be some cases where Quicksort runs in $O(n^2)$ time

The End