ECE430.217 Data Structures

Quicksort

Textbook: Weiss Chapter 7.7

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

In this topic we will look at quicksort:

- The idea behind the algorithm
- The run time and worst-case scenario
- Strategy for avoiding the worst-case: median-of-three
- Implementing quicksort in place
- Examples

Strategy

Previously we have seen two $O(n \ln(n))$ sorting algorithms:

- **Heap sort** requires no additional memory (i.e., in-place sorting)
- **Merge sort** requires $\Theta(n)$ additional memory (i.e., out-of-place sorting)

Now we will look at **quick sort**

- A recursive algorithm
  - Use an object in the array (a pivot) to divide the two
- Almost in-place sorting
  - Average case: $O(n \ln(n))$ time and $O(\ln(n))$ memory
  - Worst case: $O(n^2)$ time and $O(n)$ memory

We will look at strategies for avoiding the worst case

NOTE

Quicksort operates in-place on the data to be sorted. However, quicksort requires $O(\log n)$ stack space pointers to keep track of the subarrays in its divide and conquer strategy. Consequently, quicksort needs $O(\log^2 n)$ additional space. Although this non-constant space technically takes quicksort out of the in-place category, quicksort and other algorithms needing only $O(\log n)$ additional pointers are usually considered in-place algorithms.

https://en.wikipedia.org/wiki/In-place_algorithm

- CLRS states quicksort is in-place sorting

Idea: Quicksort

Merge sort splits the array into sub-lists and sorts them.

- The larger problem is split into two sub-problems based on *location* in the array

The idea of **quicksort**:

- Pick an object in the array
- Then partition the remaining objects into two groups:
  - Smaller ones than the chosen one
  - Larger ones than the chosen one

Example: Quicksort

For example, given

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	**44**	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location if the list was sorted

- Keep repeating this on the left and right groups

Run-time analysis: Best case

In the **best case**, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

What happens if you are unlucky (i.e., the worst case)?

Run-time analysis: Worst case

Suppose we choose **the smallest element** as our pivot

80	38	95	84	66	10	79	**2**	26	87	96	12	43	81	3

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

The run time is $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

- If you keep picking up the smallest

Avoiding Worst: Median of three

Best if you can pick the median element in the list as our pivot:

80	38	95	84	66	10	79	2	26	87	96	12	43	81	3

Unfortunately, you may need O(n) to find the median

Alternate strategy: take the median of a subset of entries

– Median-of-three: Inspect three elements, and take the median out of three

Avoiding Worst: Median of three

Sorting the elements based on 44 results in two sub-lists, each of which must be sorted (again, using quicksort)

Select the 26 to partition the first sub-list:

Select 81 to partition the second sub-list:

Runtime impacts of Median-of-three

Assumption

- The input array: $A[1 \ldots n]$
- The sorted output array: $A'[1 \ldots n]$

What's the good pivot? (See CLRS Problem 7-5. p209)

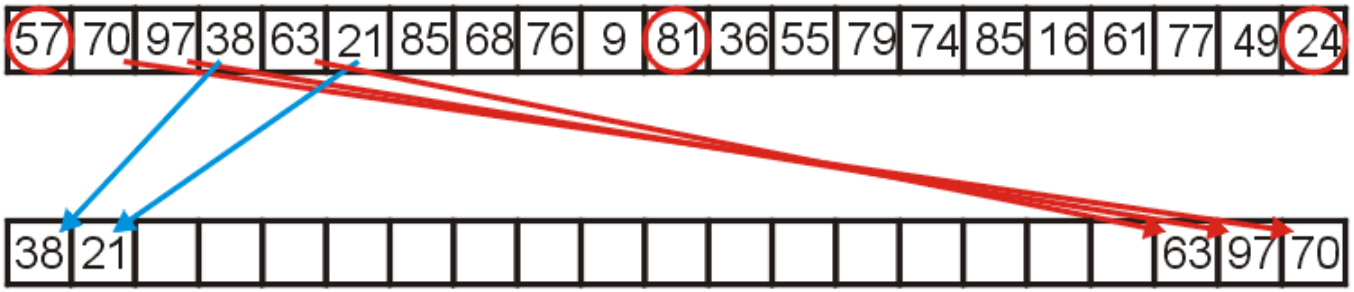
- 1) Median: i.e., pick x, where $x = A'[\frac{n+1}{2}]$
  - Median-of-three increases 50% of chances to pick the median compared to the ordinary random pick.
- 2) Somewhere in the middle: i.e., pick x, where $n/3 < i < 2n/3$
  - Probability of picking such x with Median-of-three: 13/27
  - Probability of picking such x with ordinary random pick: 1/3

Non In-place Implementation

If we allocate an additional array, we can implement the partitioning by copying elements either to the front or the back of the additional array

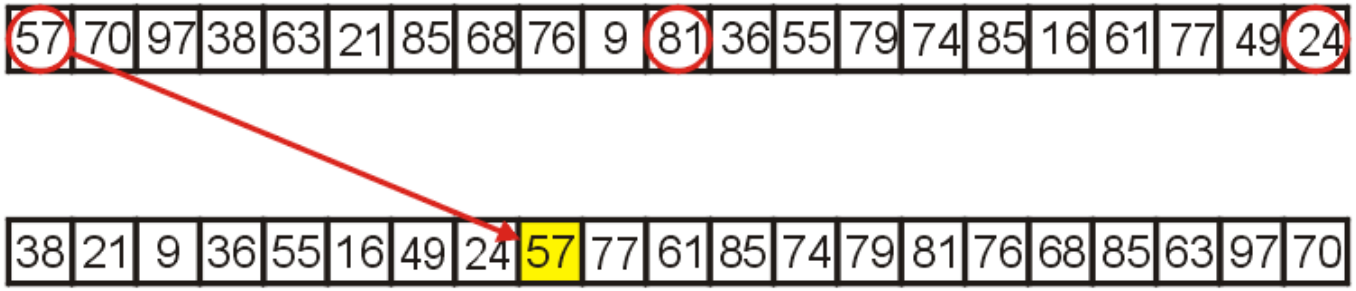
For example, consider the following:

- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array



Non In-place Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole



Non In-place Implementation

Then why not simply using merge sort?

- Merge sort has no worst case runtime.
- Merge sort always divides an array into two equal or near-equal arrays

Q. Can we implement (almost) in-place quicksort?

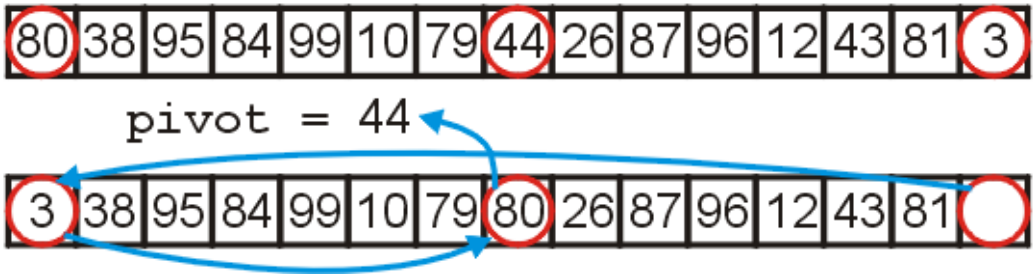
In-place Quicksort Algorithm

Step #1. Examined the first, middle, and last entries

Step #2. Choose the median of these to be the pivot

Step #3. Then,

- move the smallest entry (out of three) to the first entry
- move the largest entry (out of three) to the middle entry



In-place Quicksort Algorithm

Step #4. Find two out-of-order entries:

- Starting from the front, the entry larger than the pivot
- Starting from the back, the entry smaller than the pivot

Step #5. Once you find those two, swap the two out-of-order entries.

Step #6. Go back to step #4 until you sweep through entire entries

Quicksort Example

Consider the following unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Notation:

```
quicksort( array, begin_idx, num_entries);
```

Assumption:

We will use insertion sort if the number of entries are smaller than 6

Quicksort Example

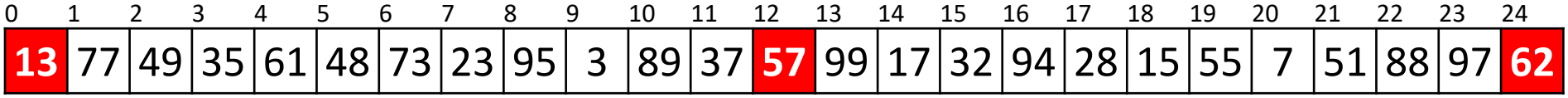
We call `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

`quicksort( array,  0, 25 )`

Quicksort Example

We are calling `quicksort( array, 0, 25 )`



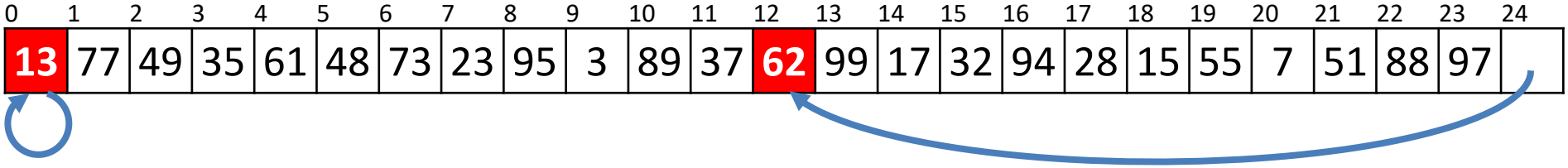
First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
```

`quicksort( array,  0, 25 )`

Quicksort Example

We are calling `quicksort( array, 0, 25 )`

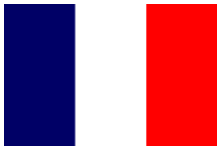


First, $25 - 0 > 6$, so find the midpoint and the pivot

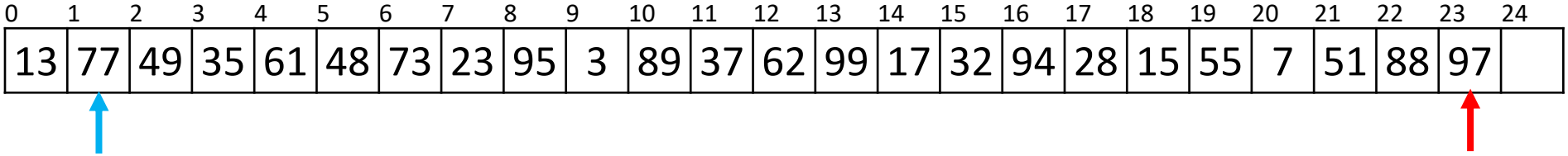
```
midpoint = (0 + 25)/2; // == 12
pivot = 57;
```

`quicksort( array,  0, 25 )`

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



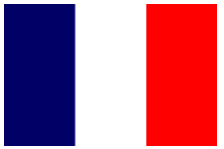
Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

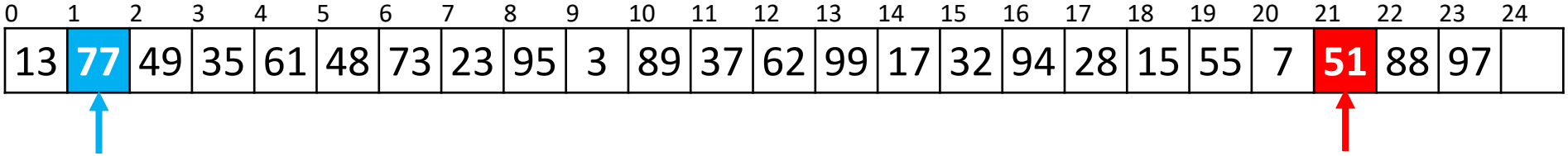
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



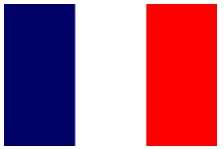
Searching forward and backward:

```
low = 1;
high = 21;
```

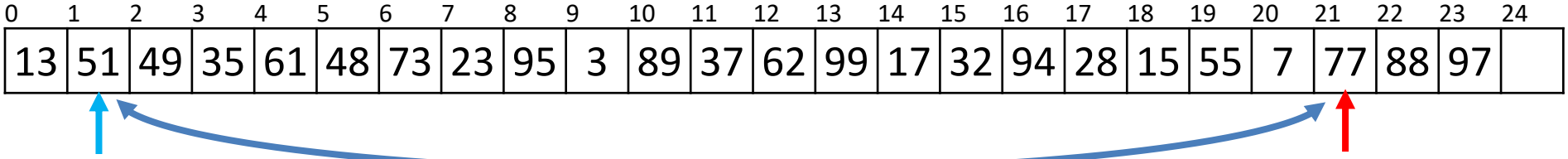
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Searching forward and backward:

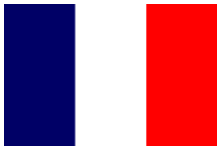
```
low = 1;
high = 21;
```

Swap them

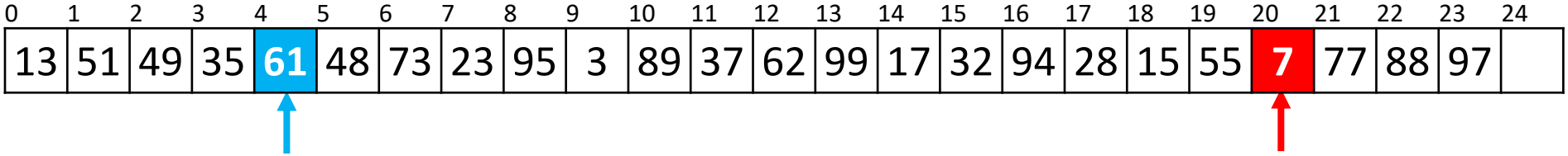
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



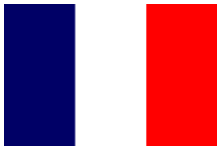
Continue searching

```
low = 4;
high = 20;
```

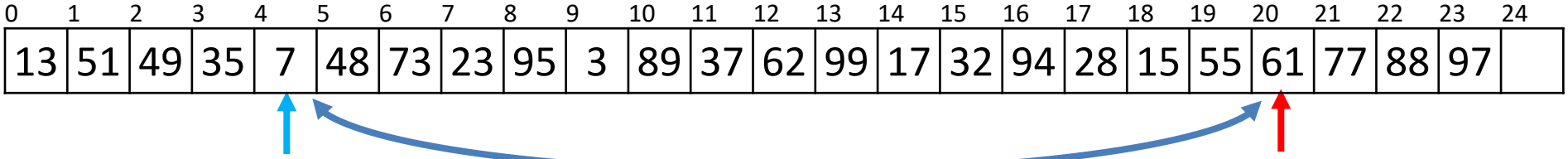
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Continue searching

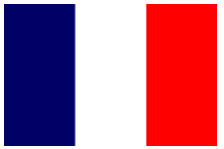
```
low = 4;
high = 20;
```

Swap them

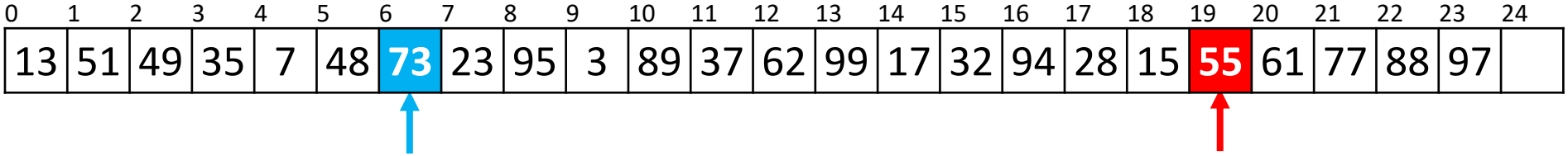
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



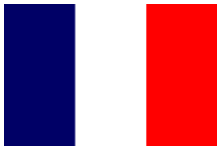
Continue searching

```
low = 6;
high = 19;
```

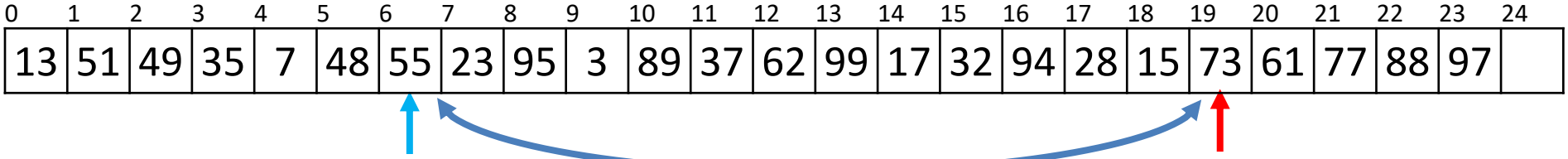
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Continue searching

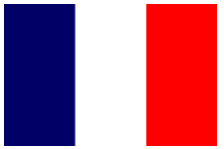
```
low = 6;
high = 19;
```

Swap them

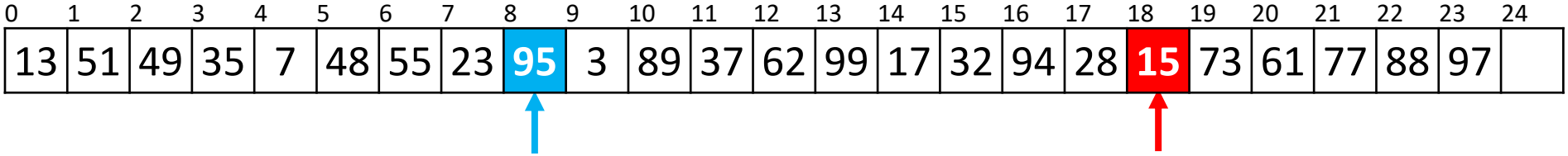
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



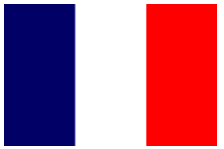
Continue searching

```
low = 8;
high = 18;
```

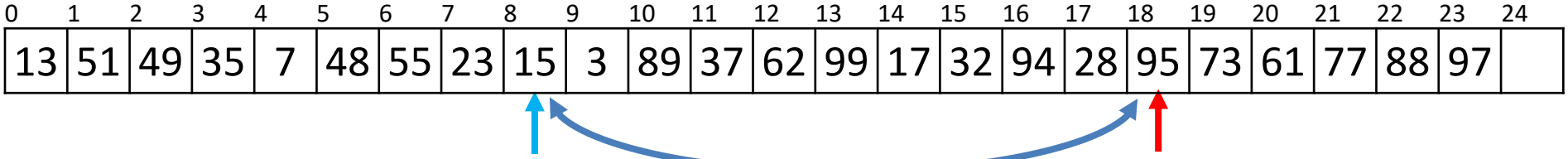
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Continue searching

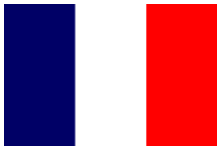
```
low = 8;
high = 18;
```

Swap them

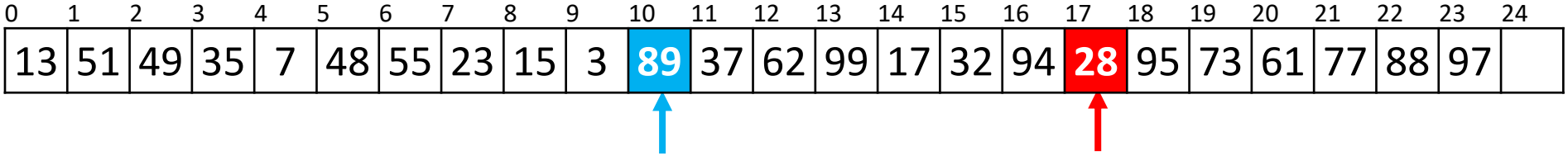
`pivot = 57;`

`quicksort( array,  0, 25 )`

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



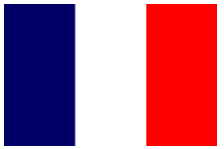
Continue searching

```
low = 10;
high = 17;
```

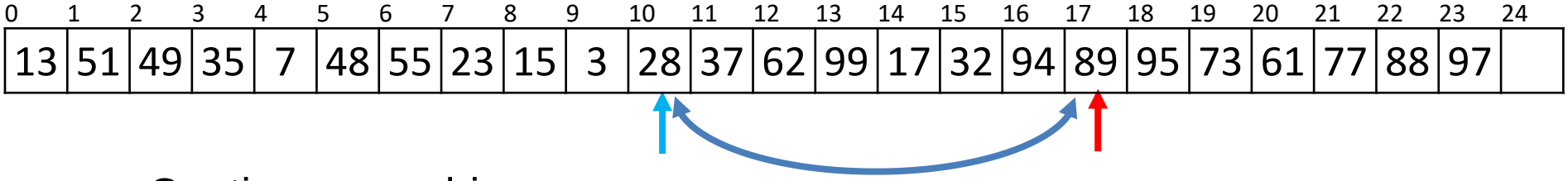
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Continue searching

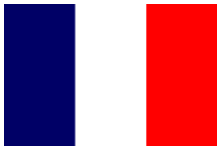
```
low = 10;
high = 17;
```

Swap them

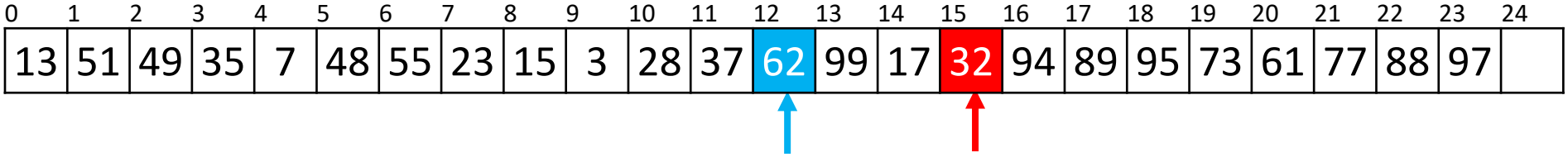
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



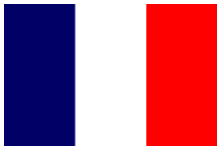
Continue searching

```
low = 12;
high = 15;
```

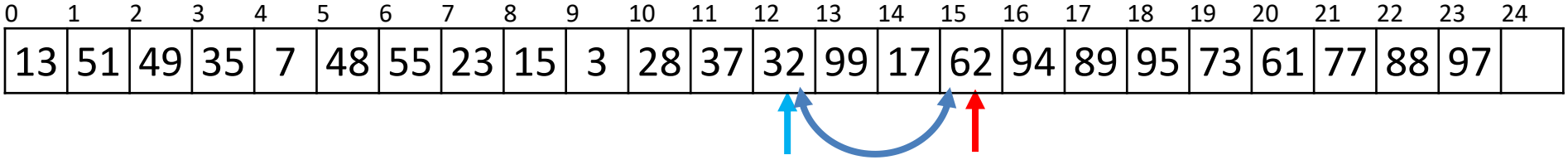
```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`



Continue searching

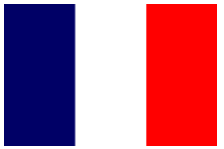
```
low = 12;
high = 15;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

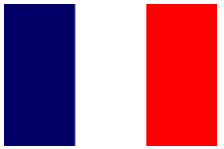
Continue searching

```
low = 13;
high = 14;
```

```
pivot = 57;
```

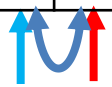
```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	



Continue searching

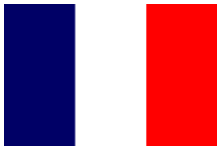
```
low = 13;
high = 14;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example



We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 14;
high = 13;
```

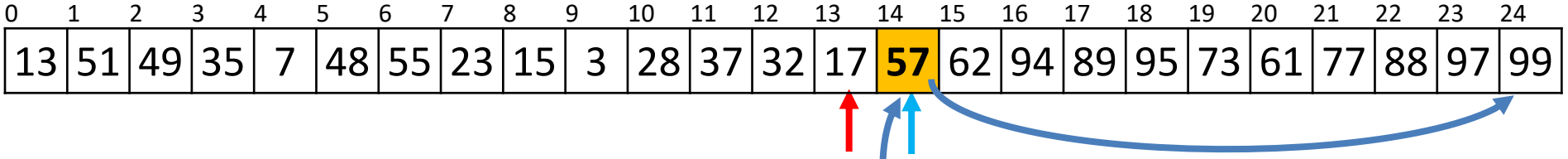
Now, `low > high`, so we stop

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort Example

We are calling `quicksort( array, 0, 25 )`



Continue searching

```
low = 14;
high = 13;
```

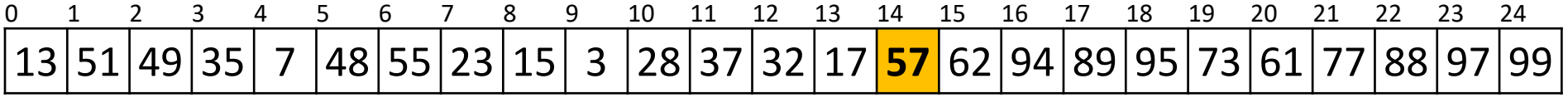
Now, `low > high`, so we stop

`pivot = 57;`

`quicksort( array,  0, 25 )`

Quicksort Example

We are calling `quicksort( array, 0, 25 )`



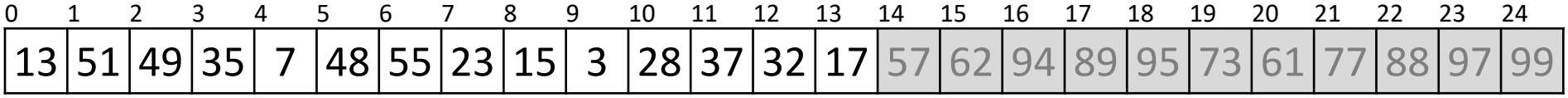
We now begin calling quicksort recursively on the first half

```
quicksort( array, 0, 14 );
```

`quicksort( array,  0, 25 )`

Quicksort Example

We are executing `quicksort( array, 0, 14 )`



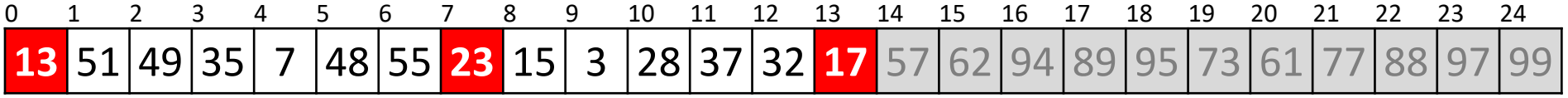
First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example

We are executing `quicksort( array, 0, 14 )`



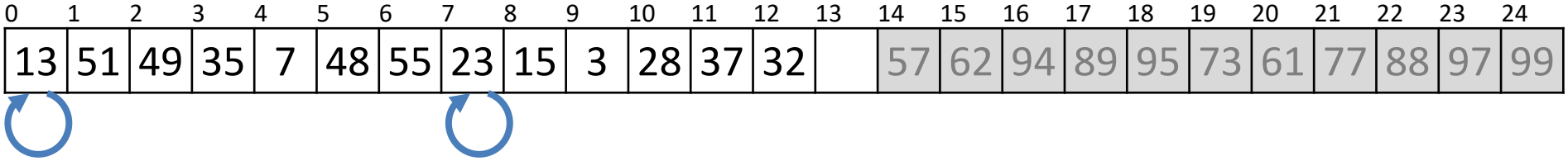
First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
pivot = 17
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example

We are executing `quicksort( array, 0, 14 )`



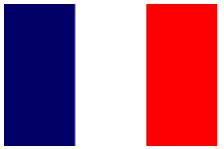
First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

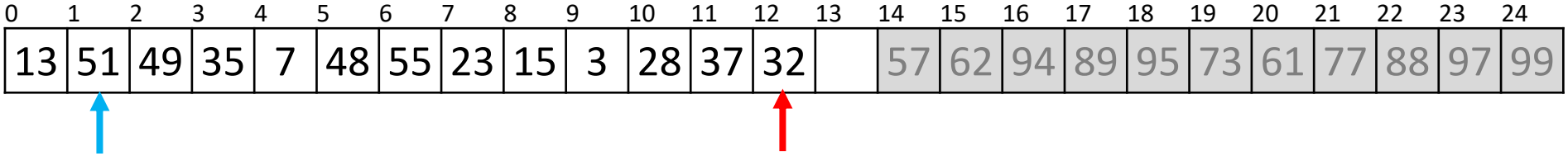
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example



We are executing `quicksort( array, 0, 14 )`



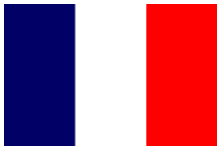
Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

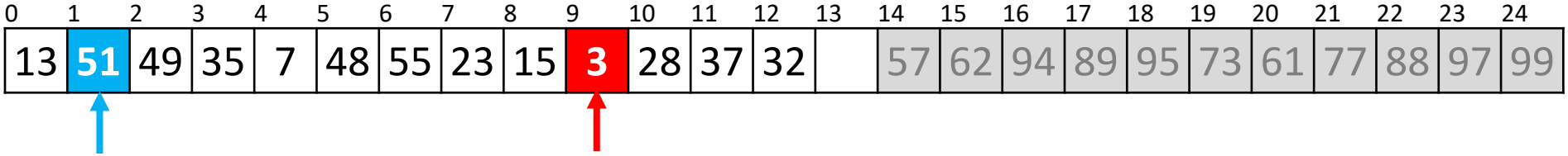
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example



We are executing `quicksort( array, 0, 14 )`



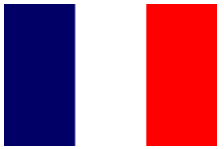
Searching forward and backward:

```
low = 1;
high = 9;
```

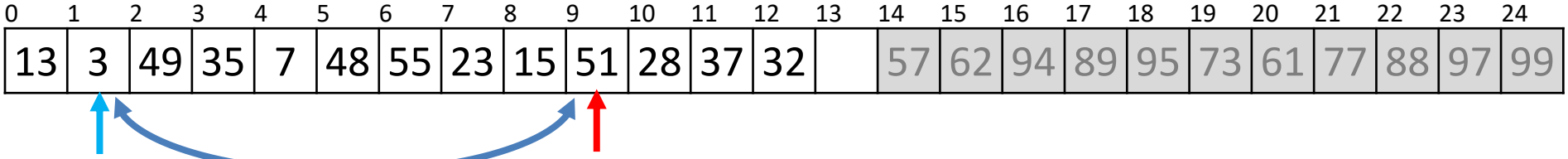
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example



We are executing `quicksort( array, 0, 14 )`



Searching forward and backward:

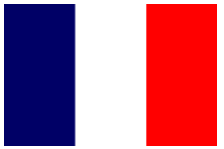
```
low = 1;
high = 9;
```

Swap them

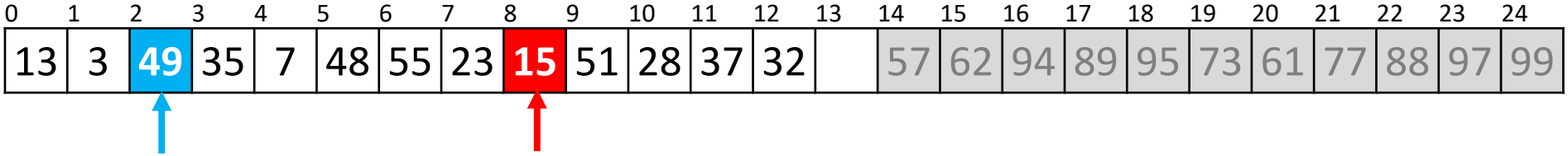
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example



We are executing `quicksort( array, 0, 14 )`



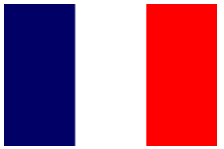
Searching forward and backward:

```
low = 2;
high = 8;
```

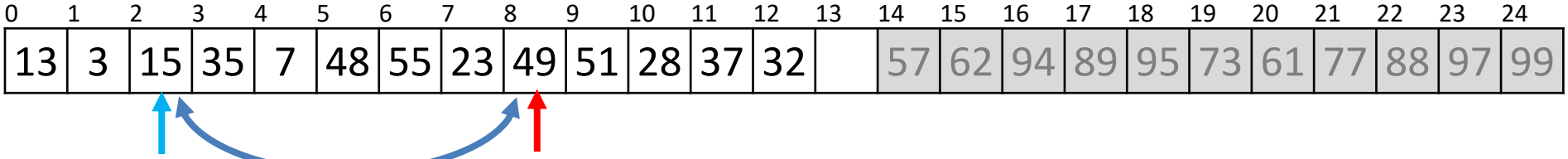
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort Example



We are executing `quicksort( array, 0, 14 )`



Searching forward and backward:

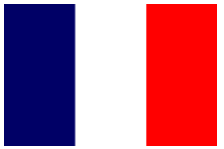
```
low = 2;
high = 8;
```

Swap them

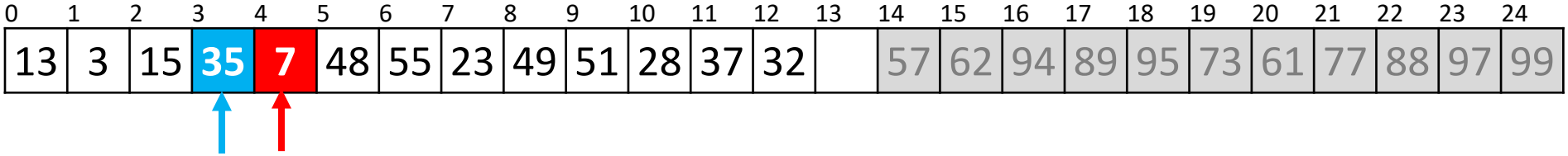
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We are executing `quicksort( array, 0, 14 )`



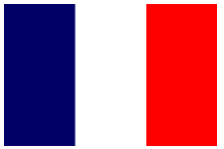
Searching forward and backward:

```
low = 3;
high = 4;
```

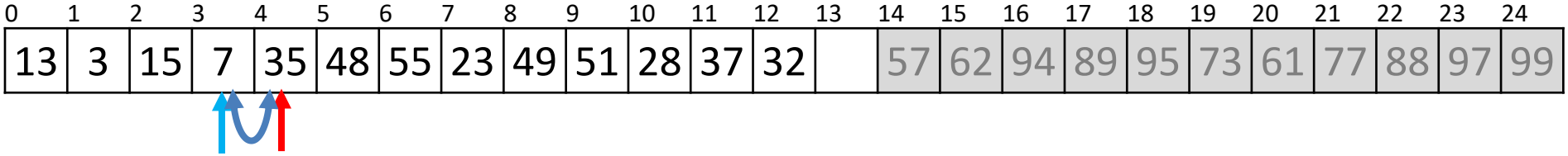
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We are executing `quicksort( array, 0, 14 )`



Searching forward and backward:

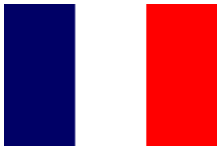
```
low = 3;
high = 4;
```

Swap them

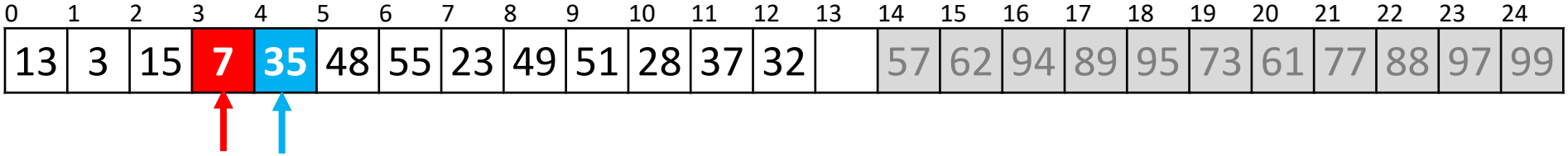
```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We are executing `quicksort( array, 0, 14 )`



Searching forward and backward:

```
low = 4;
high = 3;
```

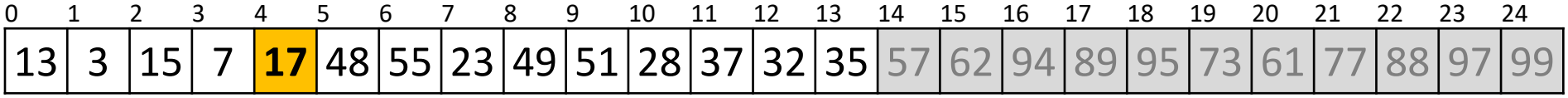
Now, `low > high`, so we stop

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`



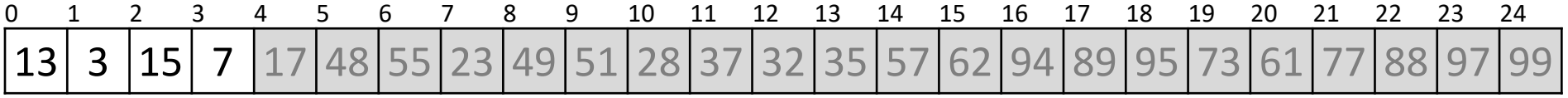
We continue calling quicksort recursively

```
quicksort( array, 0, 4 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 4 )`

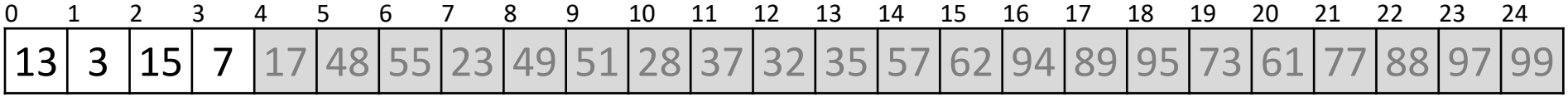


Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

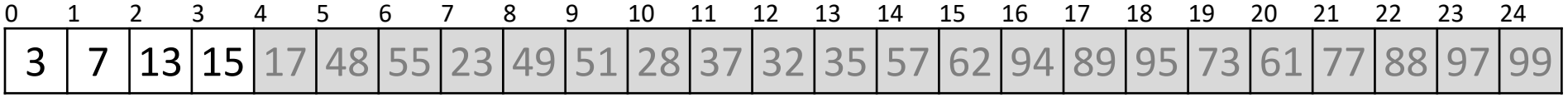
Insertion sort just sorts the entries from `0` to `3`




```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

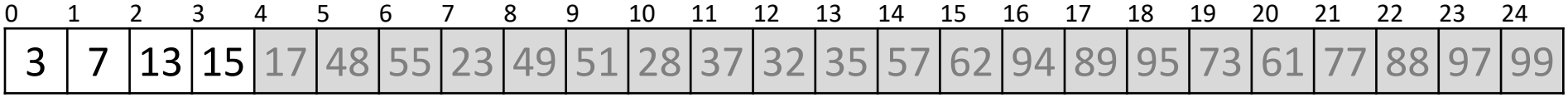


– This function call completes and so we exit

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

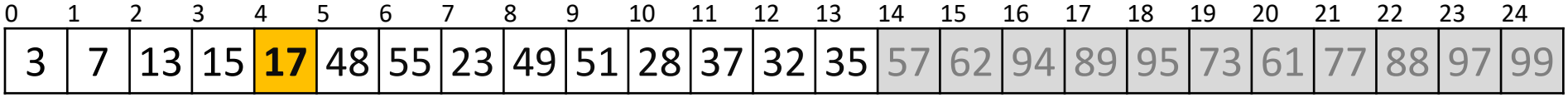
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 14 )`



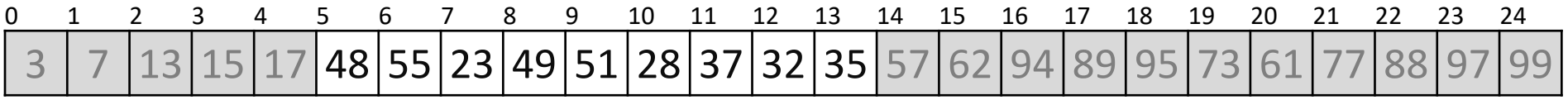
We continue calling quicksort recursively on the second half

```
quicksort( array, 0,  4 );
quicksort( array, 5, 14 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`



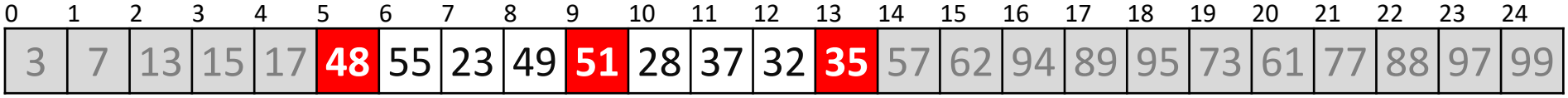
First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`



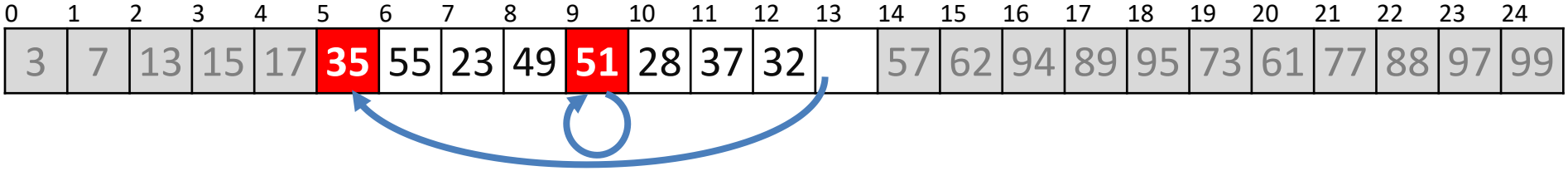
First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

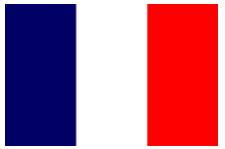


First, $14 - 5 > 6$, so find the midpoint and the pivot

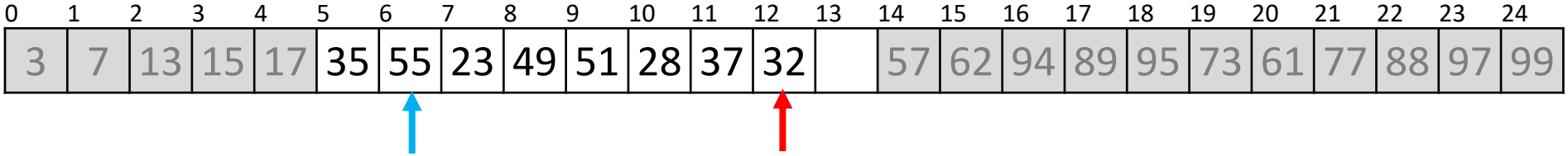
```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



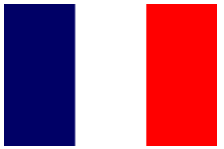
Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

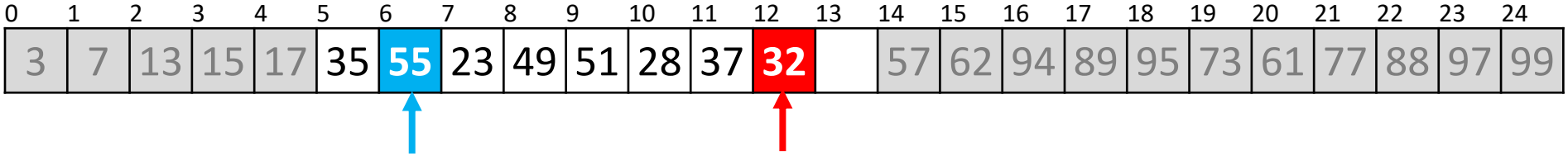
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



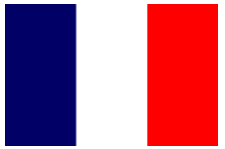
Searching forward and backward:

```
low = 6;
high = 12;
```

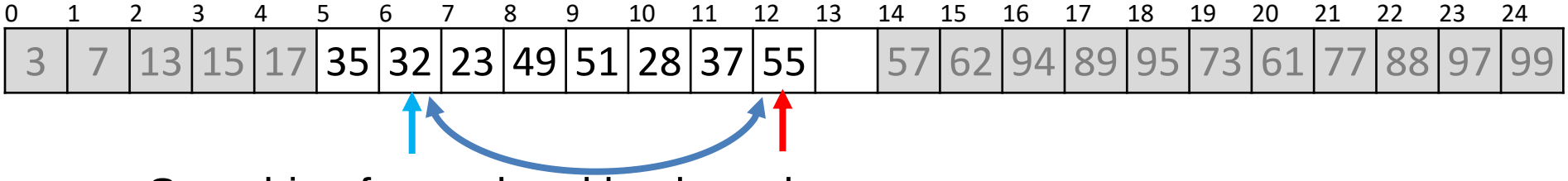
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



Searching forward and backward:

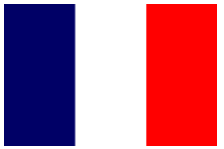
```
low = 6;
high = 12;
```

Swap them

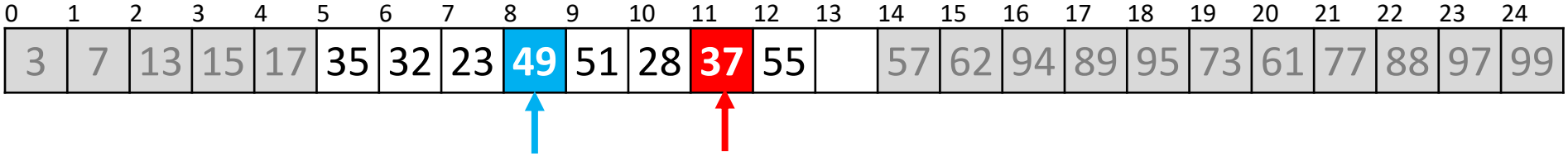
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



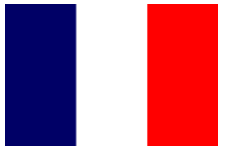
Continue searching

```
low = 8;
high = 11;
```

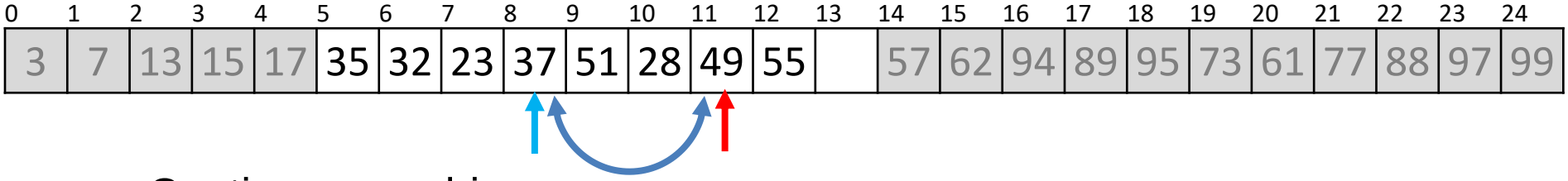
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



Continue searching

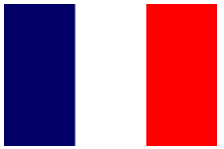
```
low = 8;
high = 11;
```

Swap them

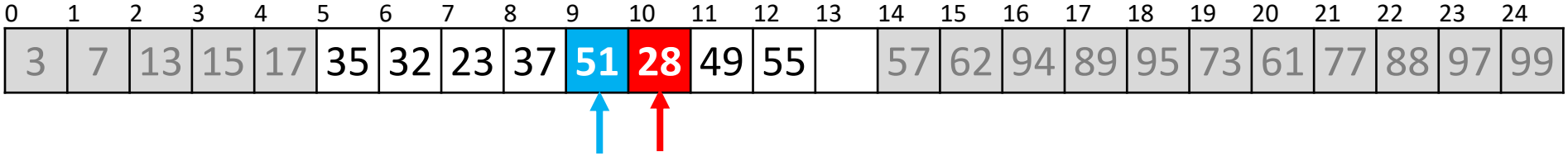
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



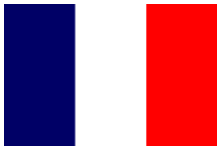
Continue searching

```
low = 8;
high = 11;
```

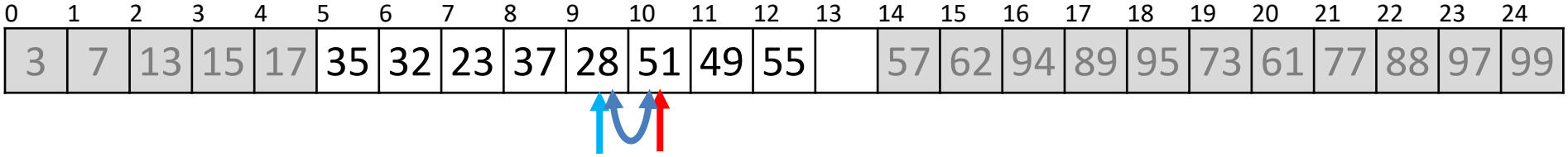
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



Continue searching

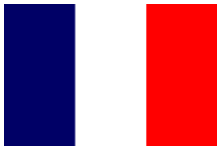
```
low = 8;
high = 11;
```

Swap them

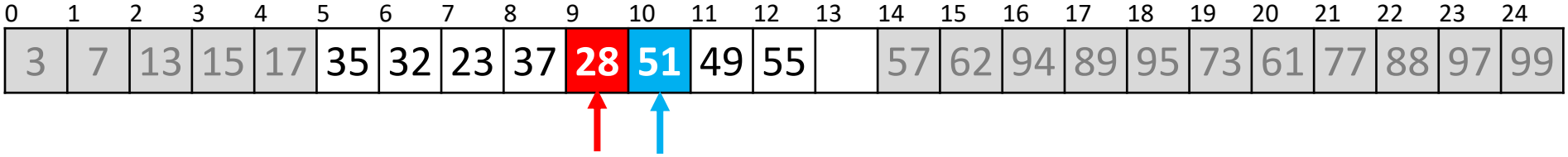
```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We now are calling `quicksort( array, 5, 14 )`



Continue searching

```
low = 8;
high = 11;
```

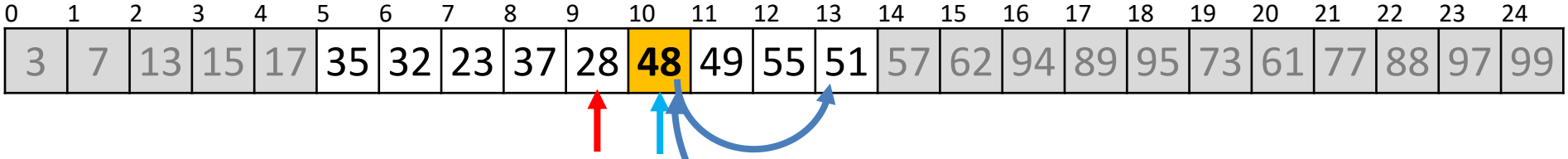
Now, `low > high`, so we stop

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`



Continue searching

```
low = 8;
high = 11;
```

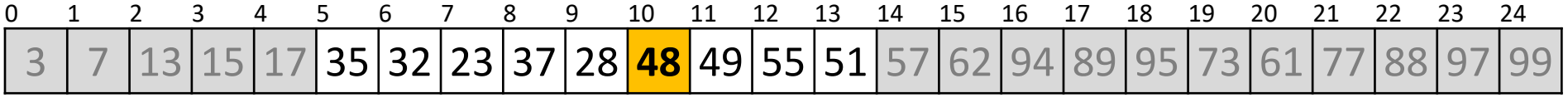
Now, `low > high`, so we stop

`pivot = 48;`

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`



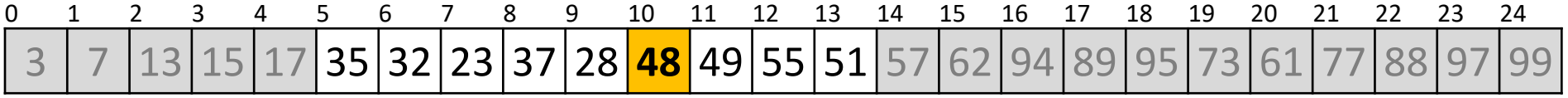
We now begin calling quicksort recursively on the first half

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`



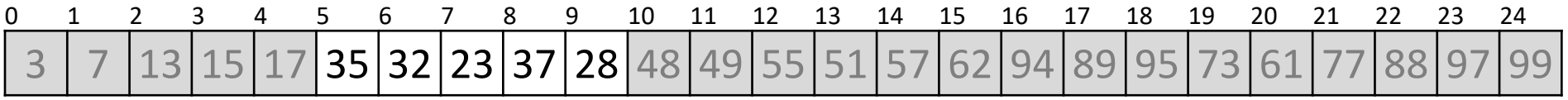
We now begin calling quicksort recursively

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 5, 10 )`

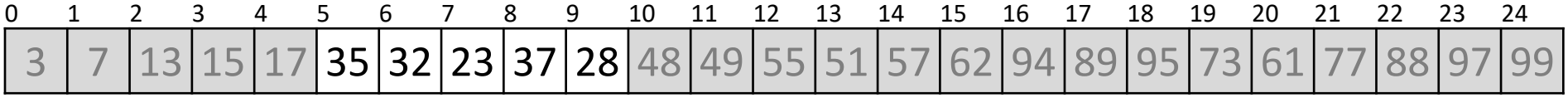


Now, $10 - 5 \leq 6$, so find we call insertion sort

```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

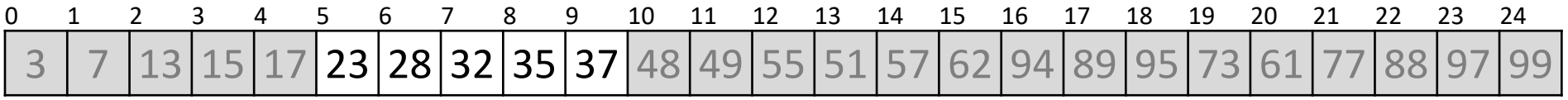
Insertion sort just sorts the entries from 5 to 9




```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 5 to 9

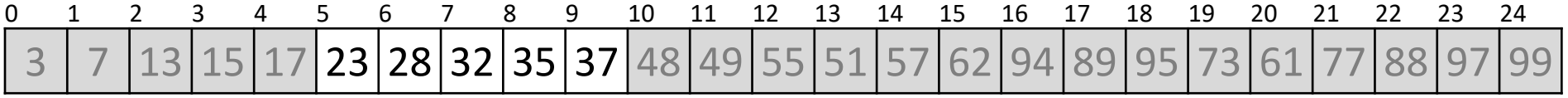


– This function call completes and so we exit

```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

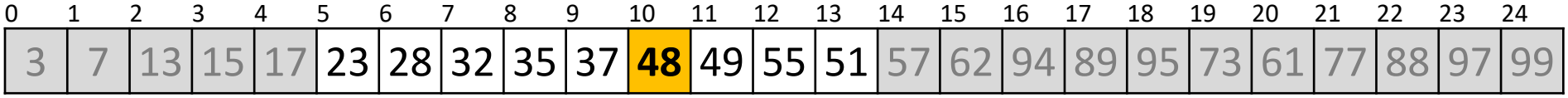
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 5, 14 )`



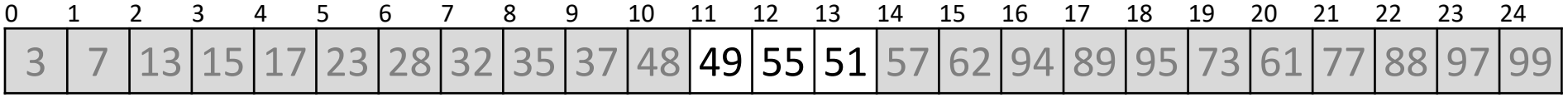
We continue calling quicksort recursively on the second half

```
quicksort( array, 5, 10 );
quicksort( array, 6, 14 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 11, 15 )`

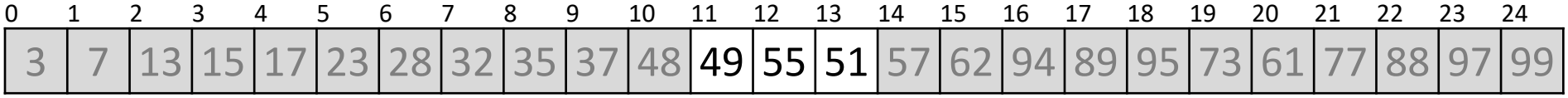


Now, $15 - 11 \leq 6$, so find we call insertion sort

```
quicksort( array,  6, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

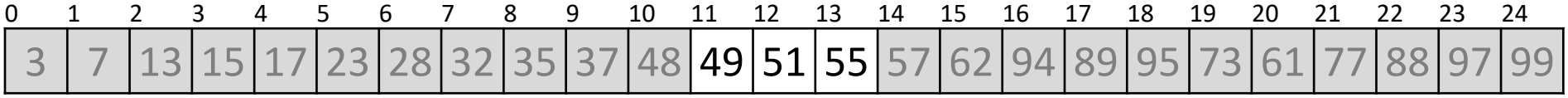
Insertion sort just sorts the entries from 11 to 14




```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 11 to 14

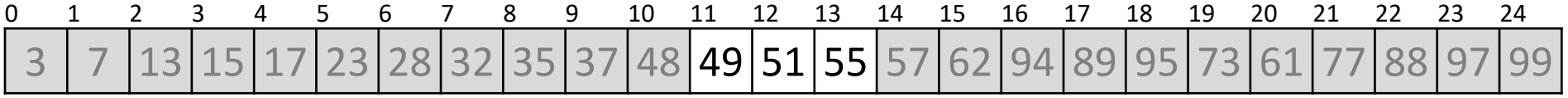


– This function call completes and so we exit

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

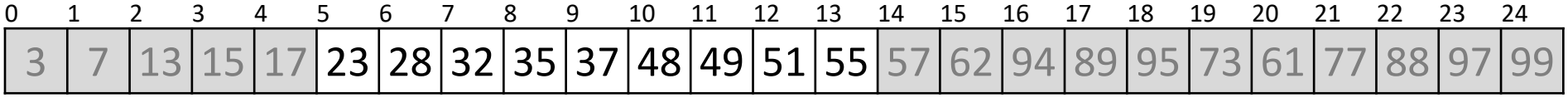
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

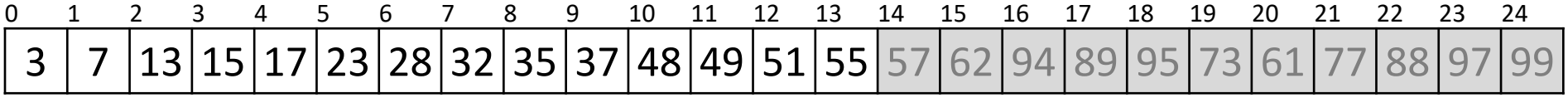
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

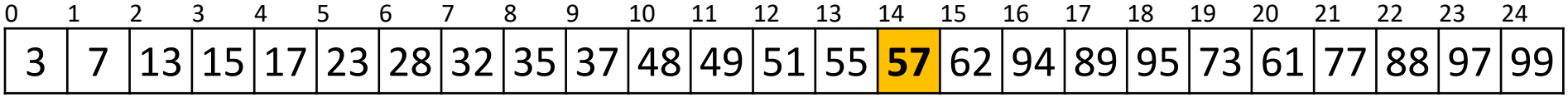
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 25 )`



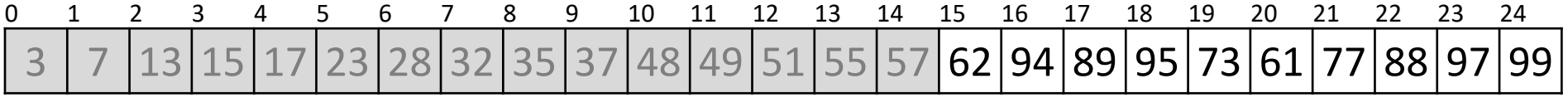
We continue calling quicksort recursively on the second half

```
quicksort( array, 0, 14 );
quicksort( array, 15, 25 );
```

`quicksort( array,  0, 25 )`

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`



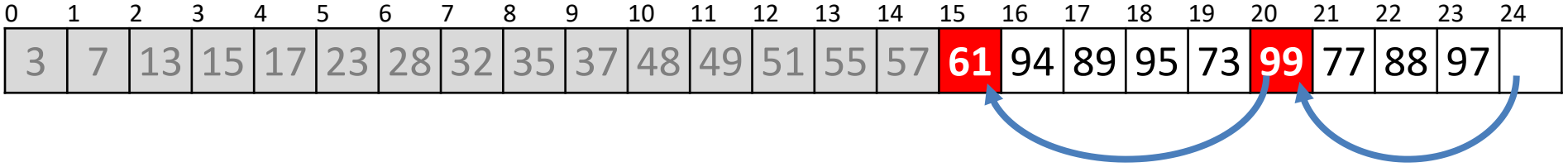
First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

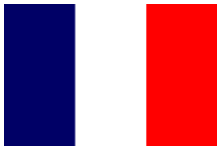


First, $25 - 15 > 6$, so find the midpoint and the pivot

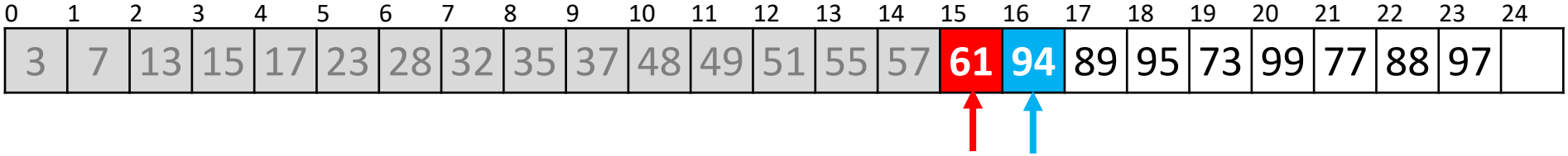
```
midpoint = (15 + 25)/2; // == 20
pivot = 62;
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example



We are back to executing `quicksort( array, 15, 25 )`



Searching forward and backward:

```
low = 16;
high = 15;
```

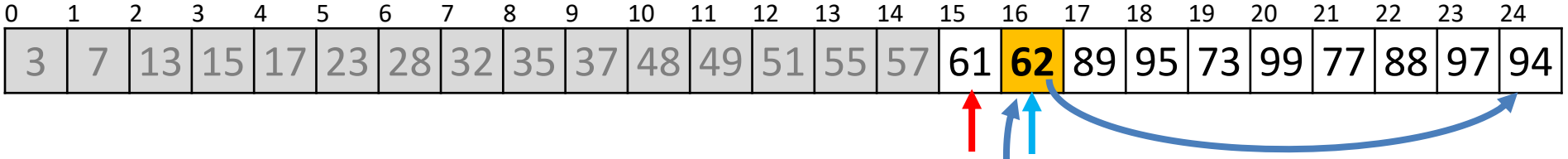
Now, `low > high`, so we stop

```
pivot = 62;
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`



Searching forward and backward:

```
low = 16;
high = 15;
```

Now, `low > high`, so we stop

- Note, this is the worst-case scenario
- The pivot is the second smallest element

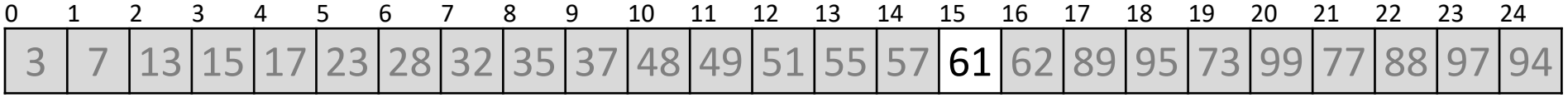
`pivot = 62;`

`quicksort( array, 15, 25 )`

`quicksort( array,  0, 25 )`

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`



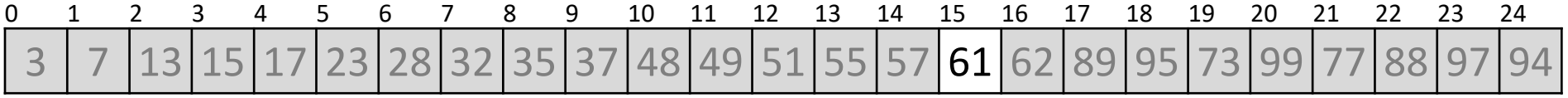
We continue calling quicksort recursively on the first half

```
quicksort( array, 15, 16 );
```

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 15, 16 )`

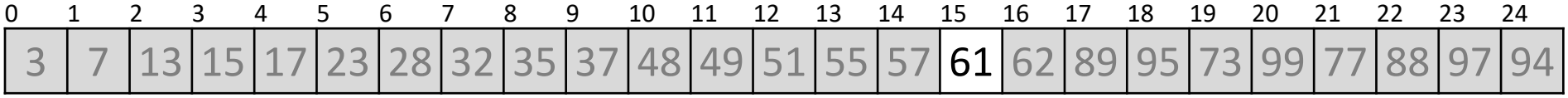


Now, $16 - 15 \leq 6$, so find we call insertion sort

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

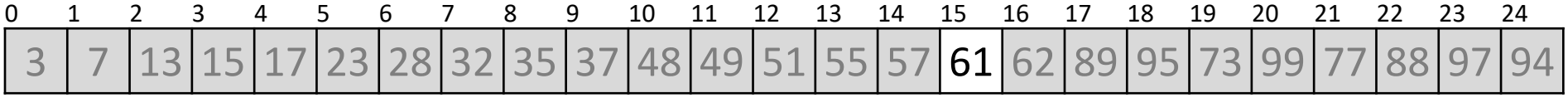
Insertion sort immediately returns




```
insertion_sort( array, 15, 16 )
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

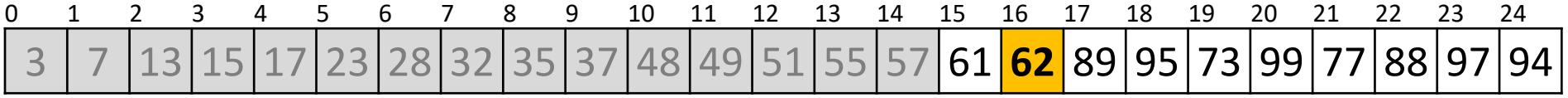
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`



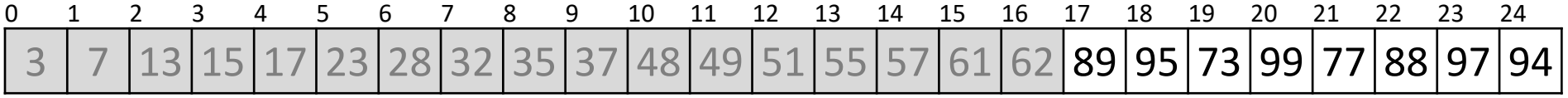
We continue calling quicksort recursively on the second half

```
quicksort( array, 15, 16 );
quicksort( array, 17, 25 );
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



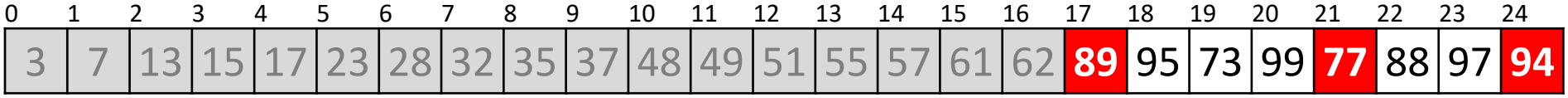
First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



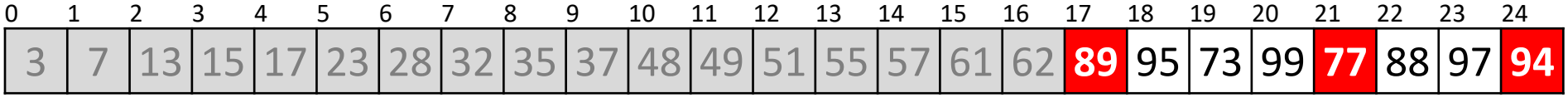
First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



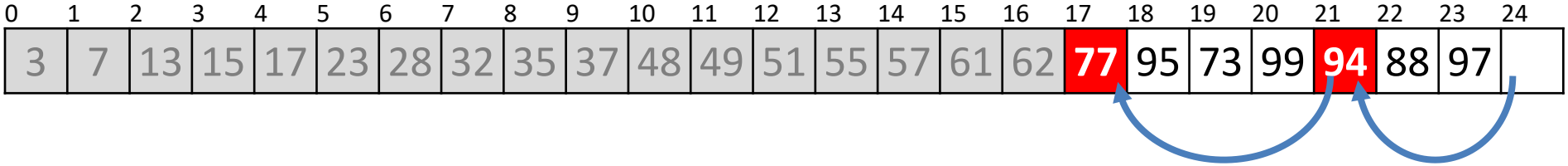
First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

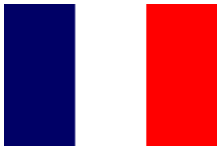


First, $25 - 17 > 6$, so find the midpoint and the pivot

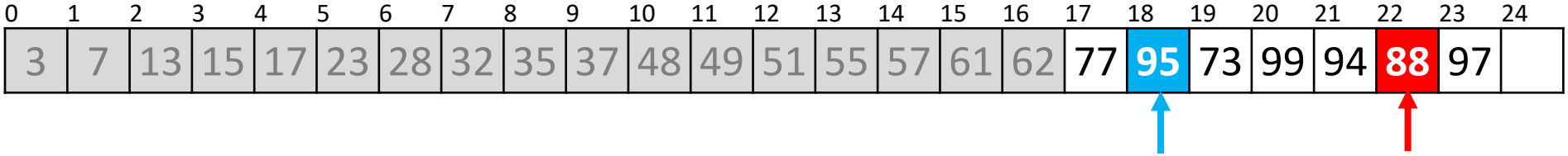
```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example



We are now calling `quicksort( array, 17, 25 )`



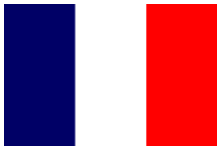
Searching forward and backward:

```
low = 18;
high = 22;
```

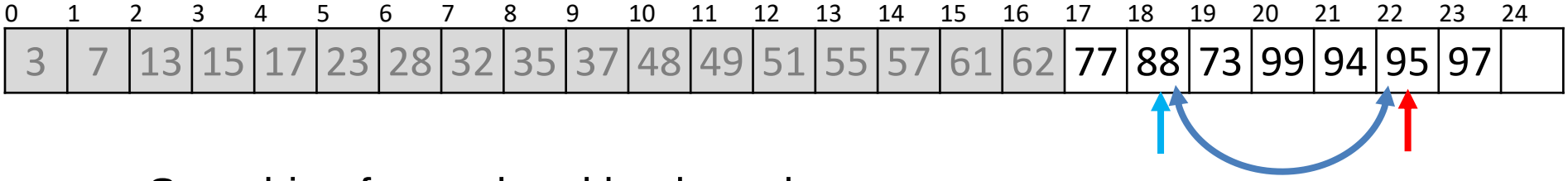
```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example



We are now calling `quicksort( array, 17, 25 )`



Searching forward and backward:

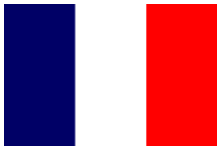
```
low = 18;
high = 22;
```

Swap them

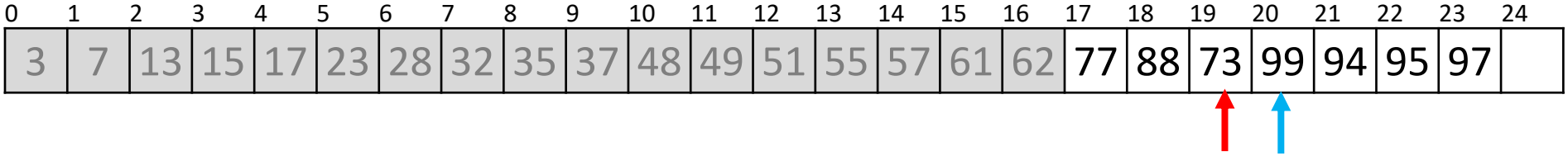
```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example



We are now calling `quicksort( array, 17, 25 )`



Searching forward and backward:

```
low = 20;
high = 19;
```

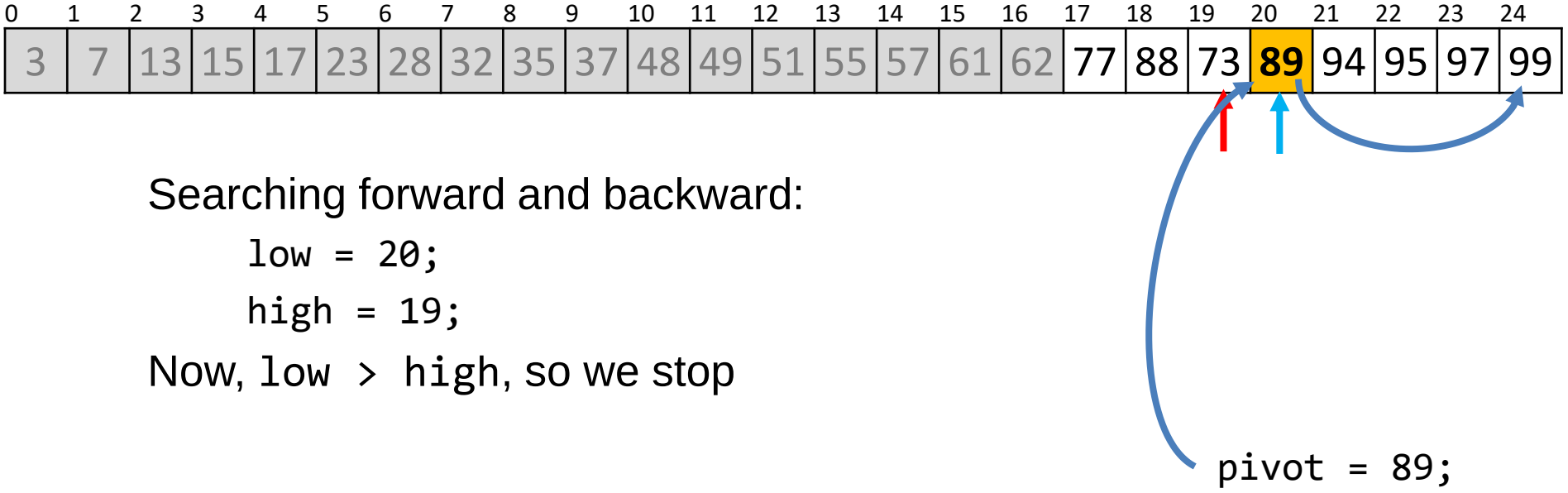
Now, `low > high`, so we stop

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



Searching forward and backward:

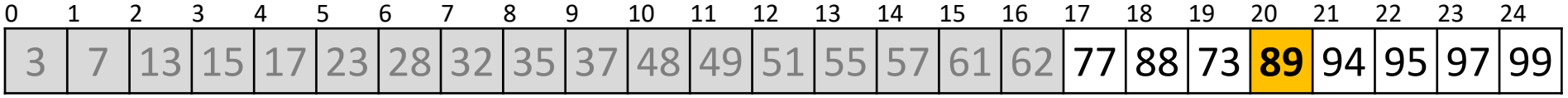
```
low = 20;
high = 19;
```

Now, `low > high`, so we stop

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



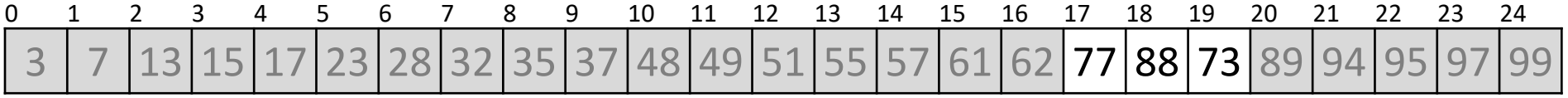
We start by calling quicksort recursively on the first half

```
quicksort( array, 17, 20 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now executing `quicksort( array, 17, 20 )`

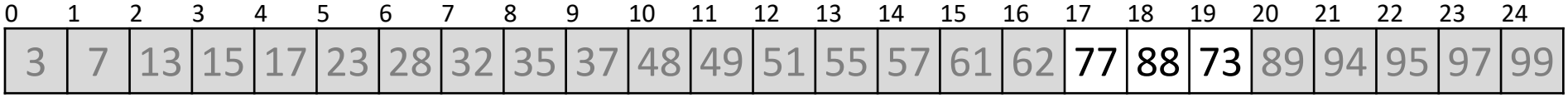


Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

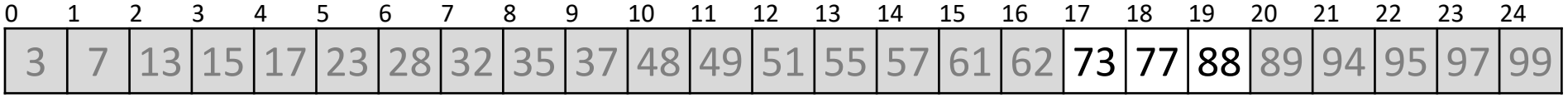
Insertion sort just sorts the entries from 17 to 19




```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 17 to 19

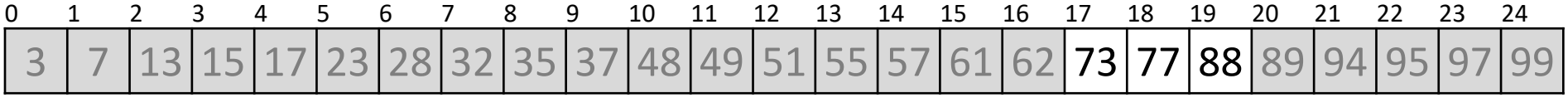


– This function call completes and so we exit

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

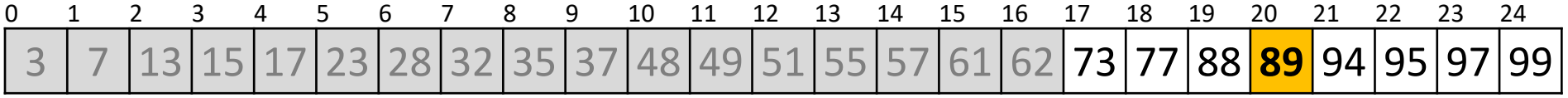
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

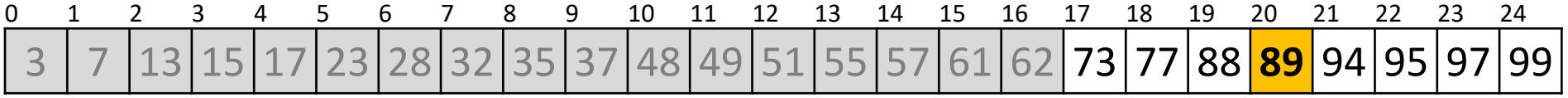
We are back to executing `quicksort( array, 17, 25 )`




```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 17, 25 )`



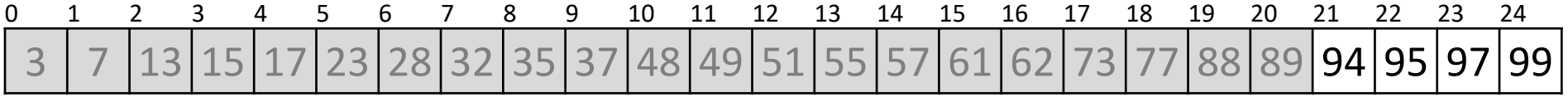
We continue by calling quicksort on the second half

```
quicksort( array, 17, 20 );
quicksort( array, 21, 25 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 21, 25 )`

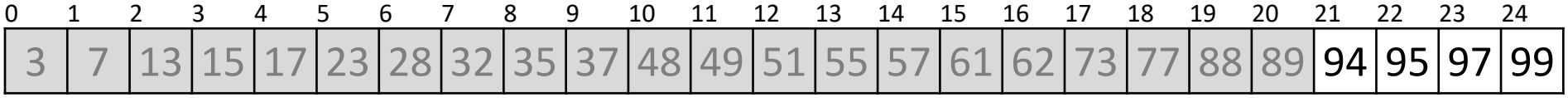


Now, $25 - 21 \leq 6$, so find we call insertion sort

```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

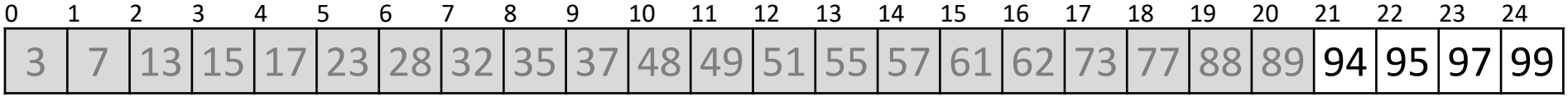
Insertion sort just sorts the entries from 21 to 24




```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 21 to 24

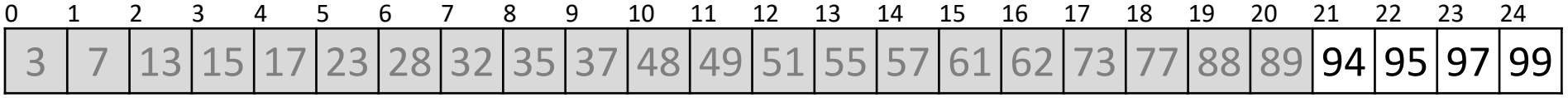


- In this case, the sub-array was already sorted
- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

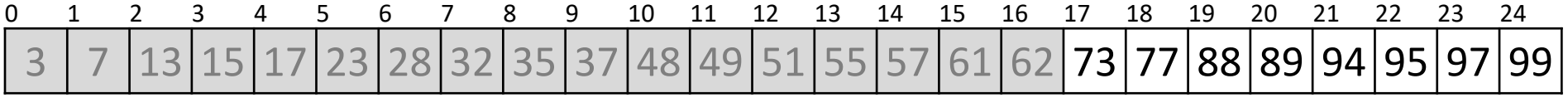
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

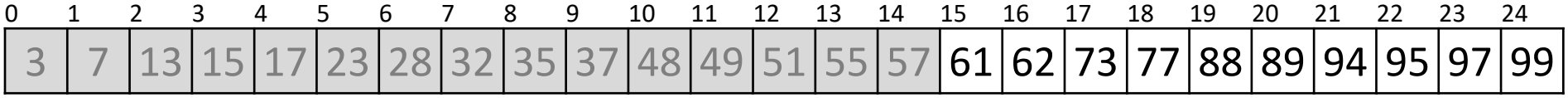
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

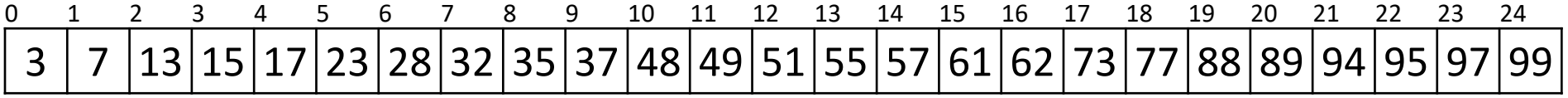
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

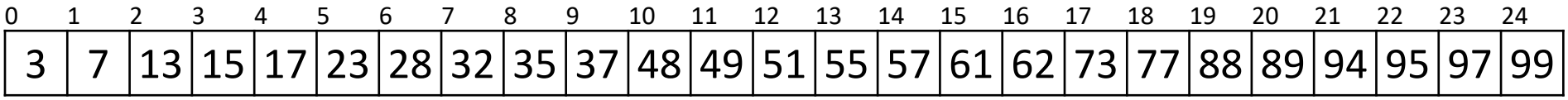
This call to `quicksort` is now also finished, so it, too, exits




```
quicksort( array,  0, 25 )
```

Quicksort example

We have now used quicksort to sort this array of 25 entries



Runtime Analysis

- The runtime of quicksort T(n) is the addition of
  - Runtime of the two recursive calls
  - The linear time spent to partition the array (i.e., cn)

- Suppose i is the number of elements in the first partition

$$T(0) = T(1) = 1$$

$$T(n) = T(i) + T(n - i - 1) + cn$$

- **The pivot selection would determine the worst/best/average cases!**

Runtime Analysis: Worst Case

- Given the runtime of quicksort

$$T(n) = T(i) + T(n - i - 1) + cn$$

- The worst case implies that the pivot is always the smallest element.
  - So i is always 0.

$$T(n) = T(n - 1) + cn, n > 1$$

- Then telescope all following equations:

$$T(n - 1) = T(n - 2) + c(n - 1)$$
$$T(n - 2) = T(n - 3) + c(n - 2)$$
$$T(2) = t(1) + c(2)$$

- Adding up yields the following:

$$T(n) = t(1) + c\sum_{i=2}^{n} i = \Theta(n^2)$$

Runtime Analysis: Best Case

- Given the runtime of quicksort

$$T(n) = T(i) + T(n - i - 1) + cn$$

- The best case implies that the pivot splits the elements half

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

- Divide by n. Then telescope all following equations:

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$
$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$
$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$
$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

- Adding up yields the following:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c \cdot \log n \qquad T(n) = cn \cdot \log n + n = \Theta(n \cdot \log n)$$

Runtime Analysis: Average Case

- Given the runtime of quicksort

$$T(n) = T(i) + T(n - i - 1) + cn$$

- The pivot is picked equally likely to the size of the first/second partitions

$$E[T(i)] = \frac{1}{n}\sum_{j=0}^{n-1} T(j)$$

$$E[T(n - i - 1)] = \frac{1}{n}\sum_{j=0}^{n-1} T(n - j - 1)$$

- So the average of T(n) will be:

$$T(n) = \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn \quad (\text{eq. } 7.14)$$

Check details from textbook, Weiss 7.7.5 p320)

Runtime Analysis: Average Case

- Multiplying n to eq.7.14:

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + cn^2 \quad \text{(eq. 7.15)}$$

- Replacing *n* into *(n-1)*

$$(n-1)T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \quad \text{(eq. 7.16)}$$

- Subtract eq.7.16 from eq.7.15

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

- Rearrange the terms and drop the insignificant *–c* on the right

$$nT(n) = (n+1)T(n-1) + 2cn$$

- Divide by n(n+1) and then telescope

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

- The sum telescope gives

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1} \frac{1}{i}$$

- From Euler's constant, we know

$\gamma = \lim_{n\to\infty}\left(-\log n + \sum_{k=1}^{n} \frac{1}{k}\right)$, where $\gamma \approx 0.577$

$$\sum_{i=3}^{n+1} \frac{1}{i} \approx \log(n+1) + \gamma - \frac{3}{2}$$

- Therefore,

$$\frac{T(n)}{n+1} = O(\log n)$$

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

Memory Requirements

The additional memory is required: $\mathbf{O}(\ln(n))$

- Each recursive function call places its local variables, parameters, *etc*., on a stack
- Average case: The depth of the recursion tree is $\mathbf{O}(\ln(n))$
- Worst case: The depth of the recursion tree is $\mathbf{O}(n)$

Run-time Summary

To summarize all three $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Heap Sort	$O(n \ln(n))$		$O(1)$	
Merge Sort	$O(n \ln(n))$		$O(n)$	
Quicksort	$O(n \ln(n))$	$O(n^2)$	$O(\ln(n))$	$O(n)$

Divide and Conquer and Recursive Algorithm

- **Divide and conquer** algorithms have three stages
  - **#1. Divide**
  - **#2. Conquer**
  - **#3. Combine**

- **In the case of quick sort,**
  - **#1. Divide:** Pick pivot and rearrange the array. Then split the array into two sub-arrays
  - **#2. Conquer:** Sort the resulting sub-arrays **recursively** (using the same quick sort)
  - **#3. Combine:** None

Summary

This topic covered quicksort

- On average faster than heap sort or merge sort
- Uses a pivot to partition the objects
- Using the median of three pivots is a reasonably means of finding the pivot
- Average run time of $\mathbf{O}(n \ln(n))$ and $\mathbf{O}(\ln(n))$ memory
- Worst case run time of $\mathbf{O}(n^2)$ and $\mathbf{O}(n)$ memory

References

Wikipedia, http://en.wikipedia.org/wiki/Quicksort

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.1, 2, 3.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, p.137-9 and §9.1.

[3] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §7.1, p.261-2.

[4] Gruber, Holzer, and Ruepp, *Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms*, 4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007.