Sorting Algorithms

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- In this topic, we will introduce sorting, including:
  - Definitions
  - Assumptions
  - *In-place* sorting
  - Sorting techniques and strategies
  - Overview of run times

- Lower bound on run times
- Define inversions and use this as a measure of *unsortedness*

Definition

- Sorting is the process of:
  - Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, ..., a_{n-1})$$

e.g., numbers, and returning a reordering

$$(a'_0, a'_1, ..., a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$$

- The conversion of an Abstract List into an Abstract Sorted List

Assumption

- In these topics, we will assume that:
  - Arrays are to be used for both input and output,
  - We will focus on sorting objects and leave the more general case of sorting records based on one or more fields as an implementation detail

In-place Sorting

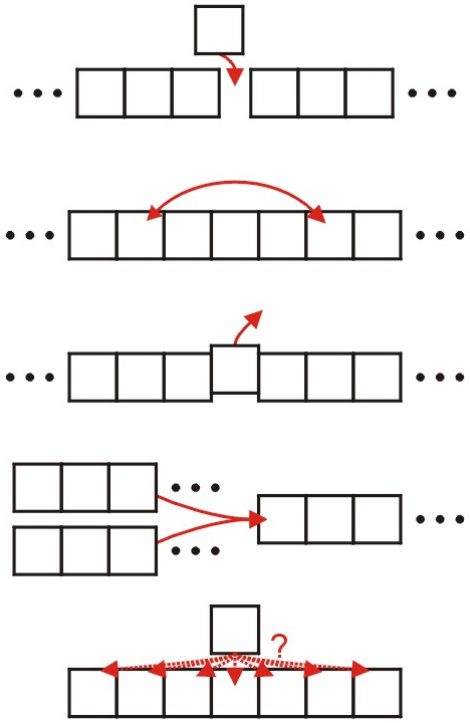
- In-place sorting algorithms are performed in-place, that is, with the allocation of at most $\Theta(1)$ additional memory

- Out-of-place sorting algorithms require the allocation of second array of equal size
  - Requires $\Theta(n)$ additional memory

- We will prefer in-place sorting algorithms

Classifications

- The operations of a sorting algorithm are based on the actions performed:
  - Insertion
  - Exchanging
  - Selection
  - Merging
  - Distribution



Run-time

- The run time of the sorting algorithms we will look at fall into one of three categories:

$$\Theta(n) \qquad \Theta(n \ln(n)) \qquad O(n^2)$$

- We will examine average- and worst-case scenarios for each algorithm

- The run-time may change significantly based on the scenario

Run-time

- We will review the more traditional $O(n^2)$ sorting algorithms:
  - Insertion sort
- Some of the faster $\Theta(n \ln(n))$ sorting algorithms:
  - Heap sort, Quicksort, and Merge sort
- And linear-time $O(n)$ sorting algorithms
  - Bucket sort and Radix sort
  - We must make assumptions about the data

Summary

- Introduction to sorting, including:
  - Assumptions
  - In-place sorting ($O(1)$ additional memory)
  - Sorting techniques
    - insertion, exchanging, selection, merging, distribution
  - Run-time classification: $O(n)$ $O(n \ln(n))$ $O(n^2)$

- A general sorting algorithm must be $\Omega(n \ln(n))$

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Sorting_algorithm
http://en.wikipedia.org/wiki/Sorting_algorithm#Inefficient.2Fhumorous_sorts

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

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

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

[5] 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.

Insertion Sort

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic discusses the insertion sort
- We will discuss:
  - the algorithm
  - an example
  - run-time analysis
    - worst case
    - average case
    - best case

Background

- Consider the following observations:
  - A list with one element is sorted
  - In general, if we have a sorted list of *k* items, we can insert a new item to create a sorted list of size *k* + 1

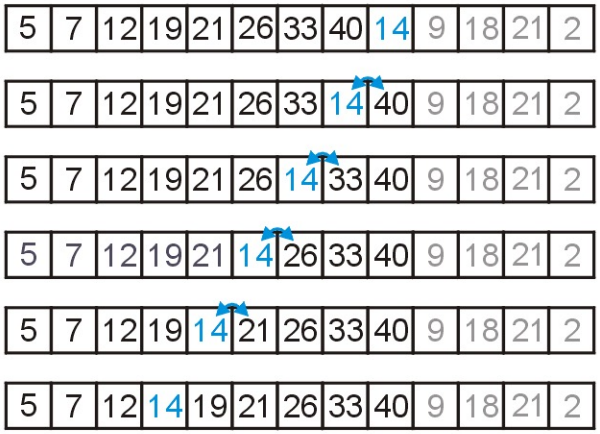
Background

- For example, consider this sorted array containing of eight sorted entries

- Suppose we want to insert 14 into this array leaving the resulting array sorted

Background

- Starting at the back, if the number is greater than 14, copy it to the right
  - Once an entry less than 14 is found, insert 14 into the resulting vacancy



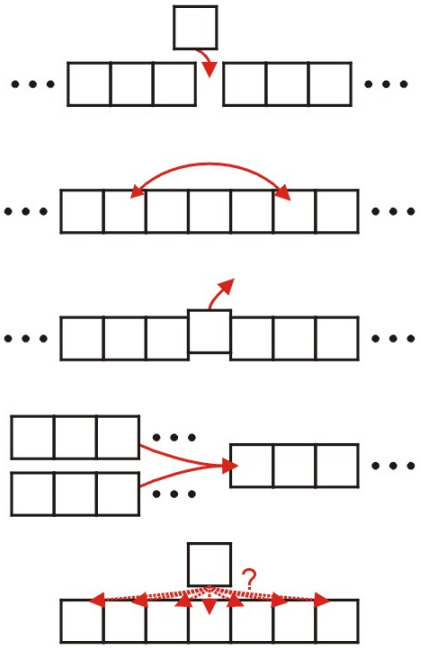
The Algorithm

- For any unsorted list:
  - Treat the first element as a sorted list of size 1

- Then, given a sorted list of size $k - 1$
  - Insert the kth item (in the unsorted list) into the sorted list
  - The sorted list is now of size $k + 1$

The Algorithm

- Recall the five sorting techniques:
  - Insertion
  - Exchange
  - Selection
  - Merging
  - Distribution

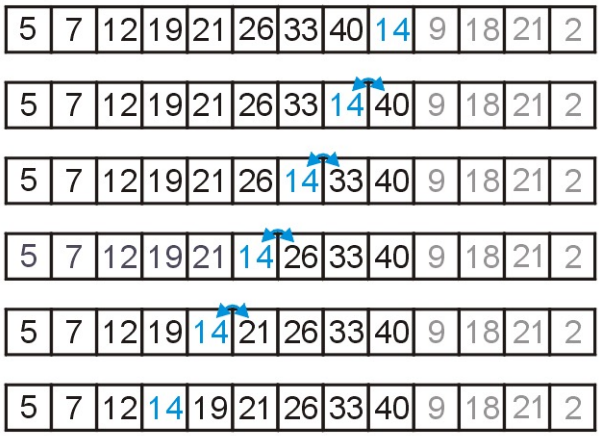


- Clearly insertion falls into the first category

Implementation

- Code for this would be:

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```



Inversions

- Before jumping into the runtime analysis, let's talk about inversions
  - The number of inversions impact the runtime of insertion sort
  - In fact, this is **true for all swap/exchange based sorting algorithms**

- Consider the following list:

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

- To what degree is this list sorted (or unsorted)?

Inversions

- The list requires only a few exchanges to make it sorted

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 12 16 25 26 33 35 42 45 56 58 67 74 75 81 83 86 88 95 99

Inversions

- Given any list of *n* numbers, there are

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

  pairs of numbers

- For example, the list (1, 3, 5, 4, 2, 6) contains the following 15 pairs:

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	(3, 2)	(3, 6)
		(5, 4)	(5, 2)	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

- You can notice that 11 of these pairs of numbers are in order. The remaining four pairs are *reversed*, or *inverted*

Inversions

- Given a permutation of *n* elements

$$a_0, a_1, \ldots, a_{n-1}$$

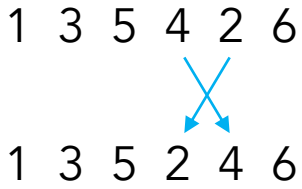
an inversion is defined as a pair of entries which are reversed

- That is, (a_j, a_k) forms an inversion if

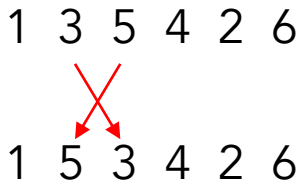
$$j < k \text{ but } a_j > a_k$$

Inversions

- Exchanging (or swapping) two adjacent entries either:
  - **removes an inversion**, *e.g.*, (4,2)



or **introduces a new inversion**, *e.g.*, (5, 3) with



Number of Inversions

- There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of numbers in any set of n objects

- Consequently, each pair contributes to either
  - the set of ordered pairs, or
  - the set of inversions

- For a random ordering, we would expect approximately half of all pairs are inverted.

  So, the number of inverted pairs are

$$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \mathbf{O}(n^2)$$

Runtime Analysis

- The number of inversions impact the runtime of insertion sort
  - Worst: $O(n^2)$
    - Reverse sorted
  - Average: $O(n^2)$
    - The random list has $O(n^2)$ inversions on average
  - Best: O(n)
    - Very few inversions (i.e., O(n))

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Insertion_sort

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.1, p.80-82.

[3] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, p.2-4, 6-9.

[4] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §8.2, p.262-5.

[5] Edsger Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM 11 (3), pp.147–148, 1968.

[6] Donald Knuth, *Structured Programming with Goto Statements*, Computing Surveys 6 (4): pp.261–301, 1972.

Heap Sort

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic covers the simplest $\Theta(n \ln(n))$ sorting algorithm: *heap sort*

- We will:
  - define the strategy
  - analyze the run time
  - convert an unsorted list into a heap
  - cover some examples

- Extra: in-place sorting

Heap Sort

- Let's use min-heap to implement sorting
  - Inserting *n* objects into a min-heap first
  - Then popping *n* objects will pop in sorted order

- Heap sort strategy
  - Given an unsorted list with *n* objects, place them into a heap, and take them out

Run time Analysis of Heap Sort

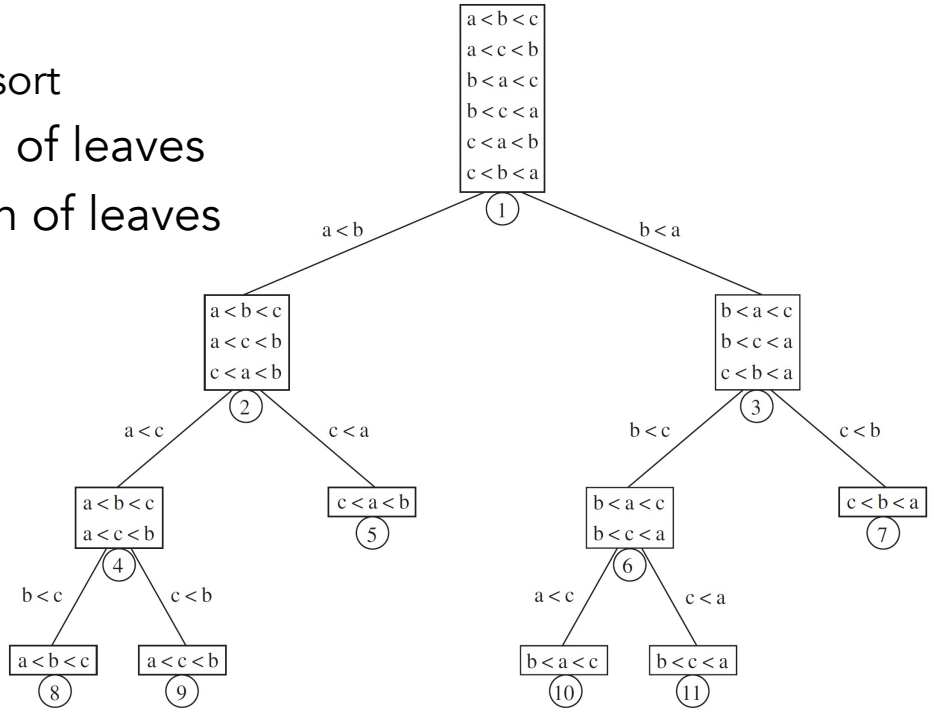
- Taking an object out of a heap with *n* items requires **O**(ln(*n*)) time
- Therefore, taking *n* objects out requires

$$\sum_{k=1}^{n} \ln(k) = \ln\left(\prod_{k=1}^{n} k\right) = \ln(n!)$$

- Question: What is the asymptotic bound of ln(*n*!)?
- Let's step back, and think about the runtime of generic sorting algorithms

General Lower Bound of Sorting

- A decision tree
  - Abstract representation of generic comparison-based sorting algorithm
  - A decision tree is a binary tree
    - Each node represents a set of possible ordering
    - Each edge represents a comparison operation
  - The depth of a leaf node
    - The number of comparisons to sort
  - Worst case: the maximum depth of leaves
  - Average case: the average depth of leaves



General Lower Bound of Sorting

- **Lemma (7.1, Weiss p.324)**
  - Let T be a binary tree of depth d. Then T has at most 2^d leaves
- **Lemma (7.2)**
  - A binary tree with L leaves must have depth at least $\lceil \log L \rceil$
- **Theorem (7.6)**
  - Any sorting algorithm that uses only comparisons requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case
  - **Proof:** A decision tree to sort N elements must have N! leaves

General Lower Bound of Sorting

- **Theorem (7.7)**
  - Any sorting algorithm that uses only comparisons requires $[NlogN]$ comparisons
  - **Proof**
    - $\log(N!) = \log\{N(N-1)(N-2)\cdots(1)\}$

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 1$$

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(\frac{N}{2})$$

$$\geq \frac{N}{2}\log(\frac{N}{2})$$

$$\geq \frac{N}{2}\log N - \frac{N}{2}$$

$$= \Omega(N\log N)$$

HeapSort: In-place Implementation

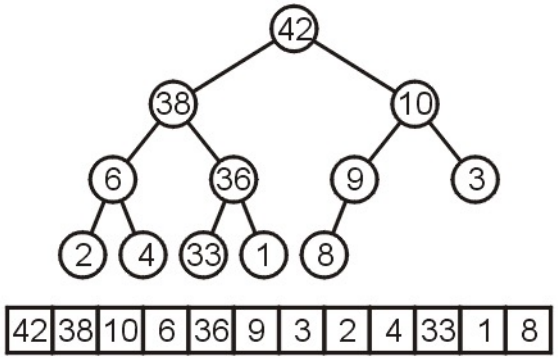
- Problem:
  - This solution requires additional memory, that is, a min-heap of size n

- This requires $\Theta(n)$ memory and is therefore not in place

- Is it possible to perform a heap sort in place, that is, require at most $\Theta(1)$ extra memory (a few extra variables)?

In-place Implementation

- Instead of implementing a min-heap, consider a max-heap:
  - A heap where the maximum element is at the top of the heap and the next to be popped

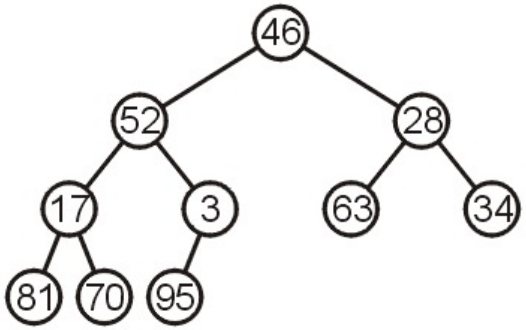


In-place Heapification

- Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

- This array represents the following complete tree:

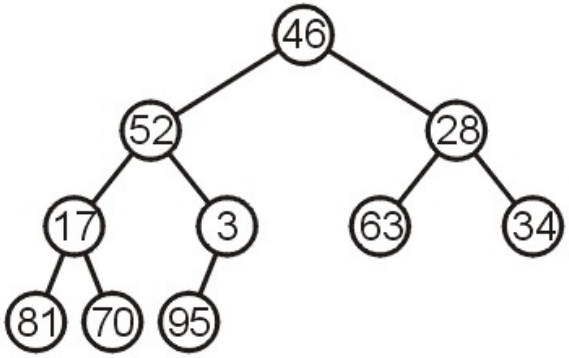


- This is neither a min-heap, max-heap, or binary search tree

- Goal: In-place conversion of this complete tree into a max heap

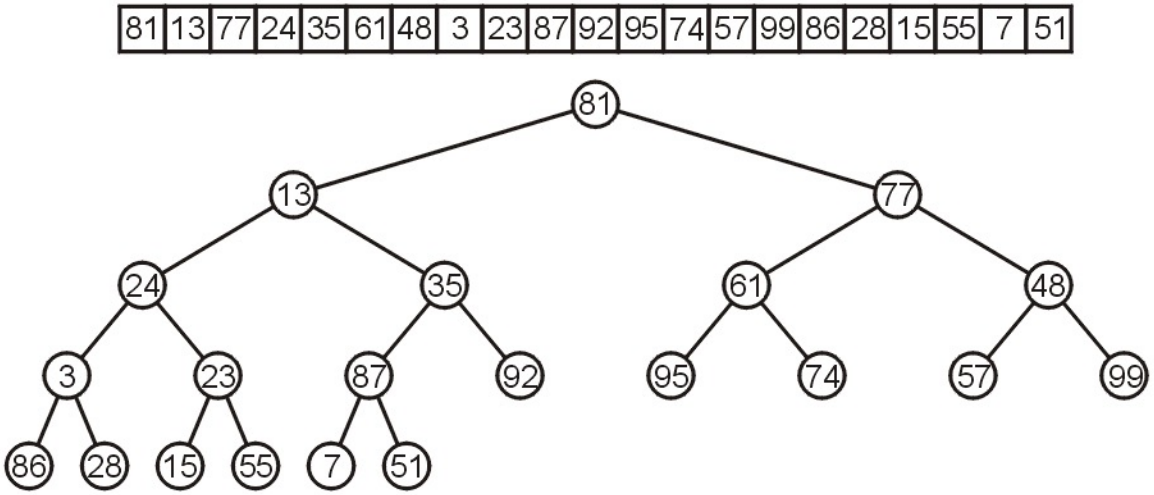
In-place Heapification

- Two strategies:
  - Top-down: Assume 46 is a max-heap and keep inserting the next element into the existing heap (similar to the strategy for insertion sort)
  - Bottom-up: Assume all leaf nodes are already max heaps. So make corrections so that previous nodes also form max heaps



In-place Heapification

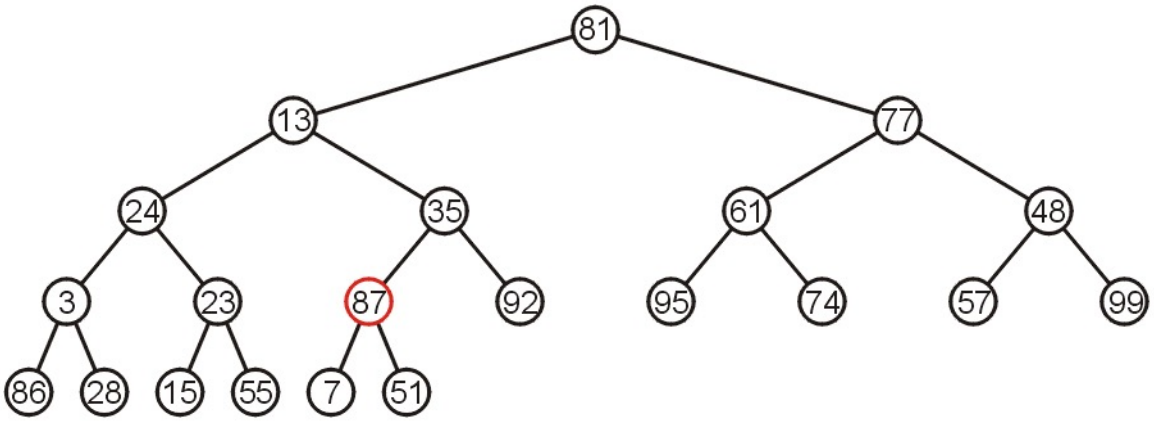
- Let's work bottom-up: each leaf node is a max heap on its own



In-place Heapification

- Starting at the back, we note that all leaf nodes are trivial heaps

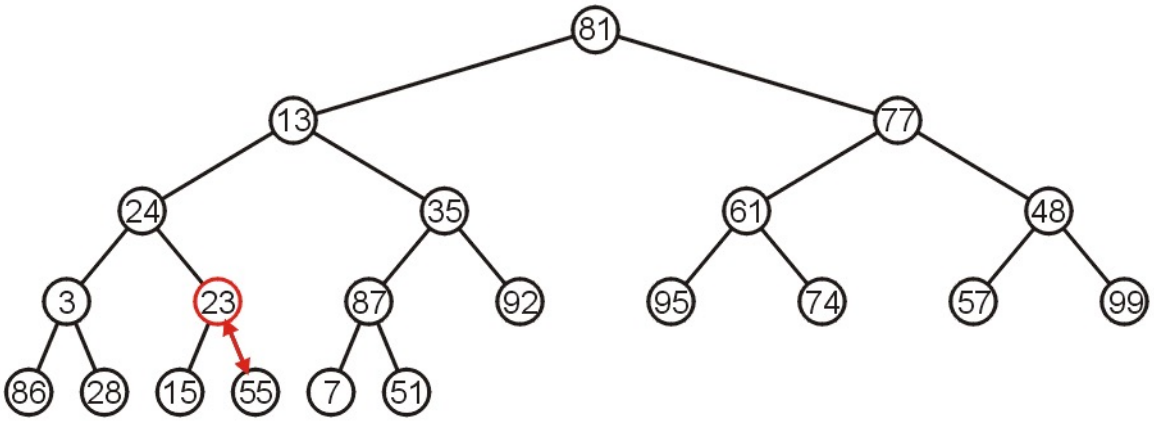
- Also, the subtree with 87 as the root is a max-heap



In-place Heapification

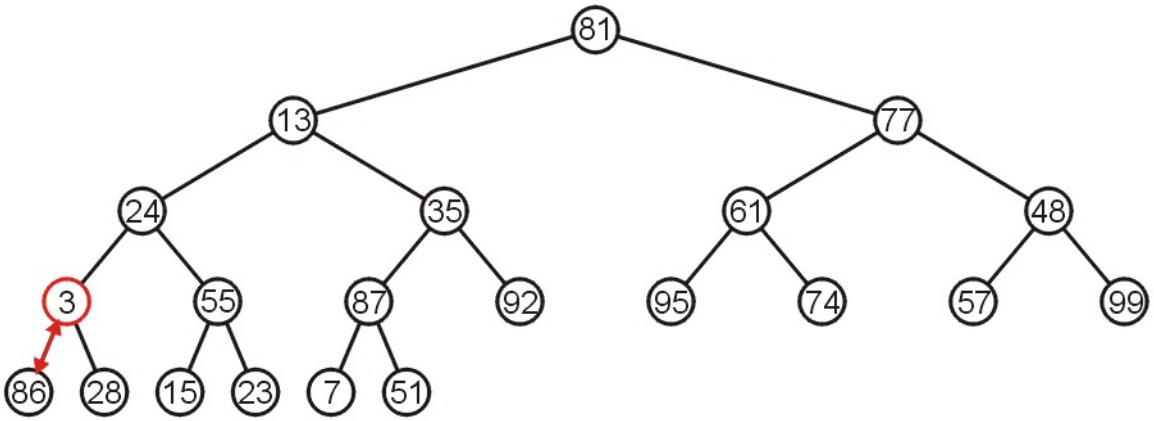
- The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap

- This process is termed *percolating down*



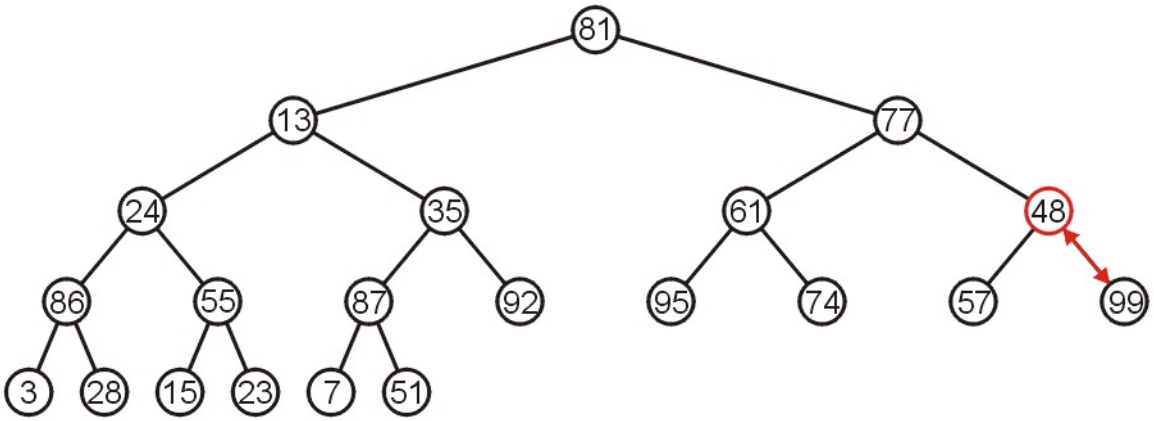
In-place Heapification

- The subtree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86



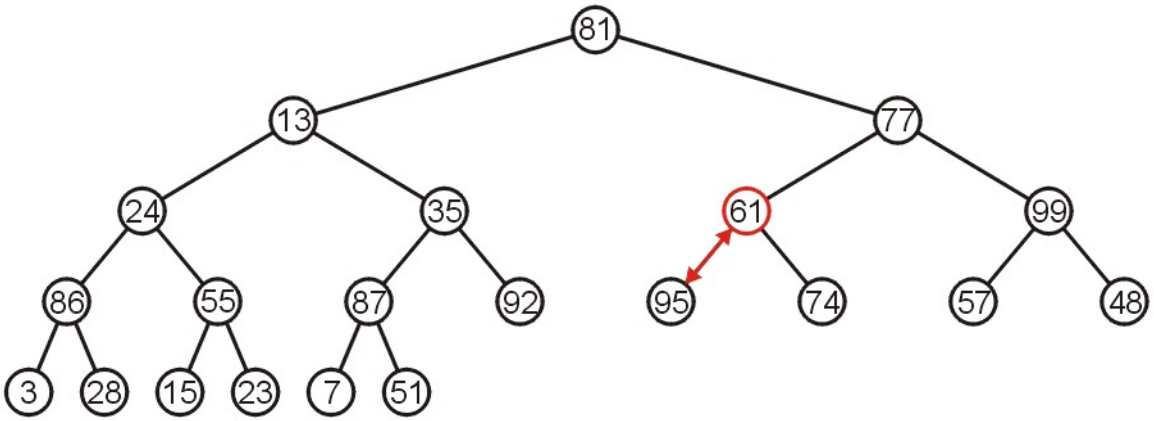
In-place Heapification

- Starting with the next higher level, the subtree with root 48 can be turned into a max-heap by swapping 48 and 99



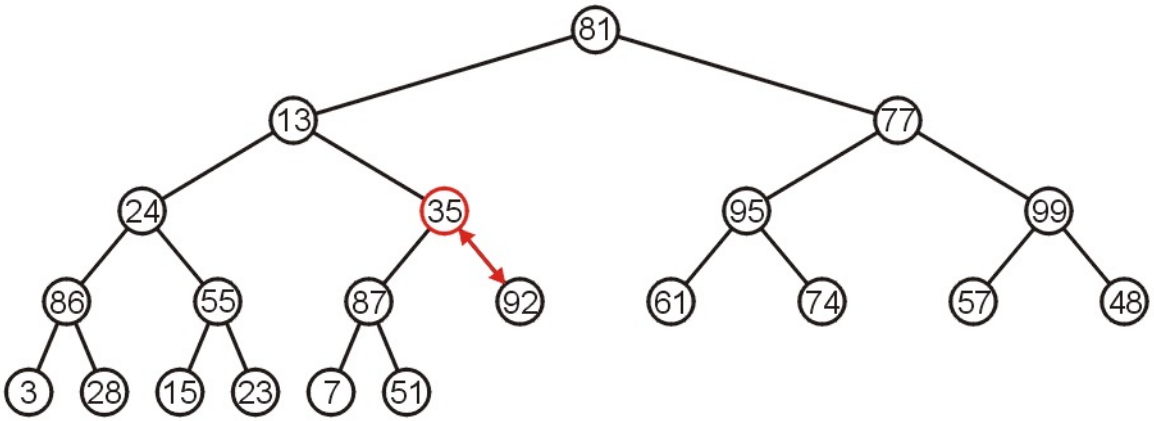
In-place Heapification

- Similarly, swapping 61 and 95 creates a max-heap of the next subtree



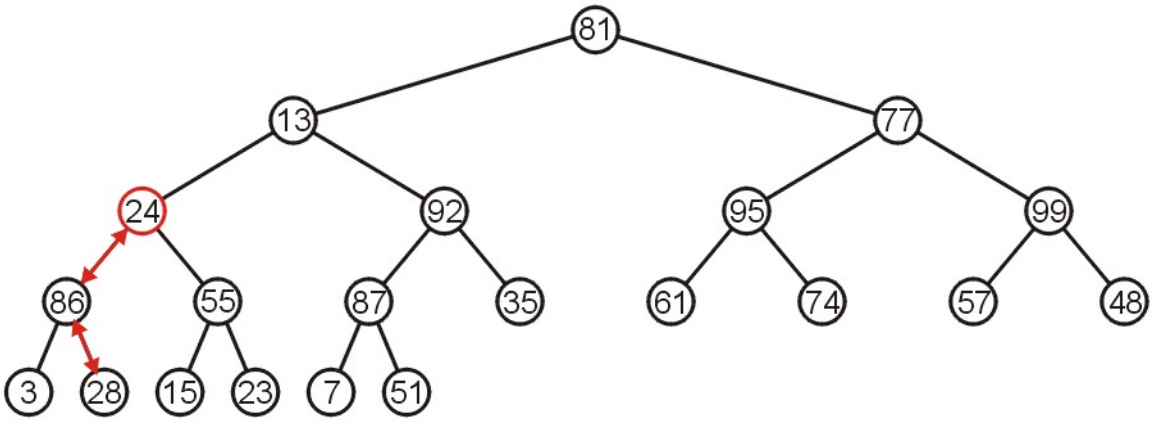
In-place Heapification

- As does swapping 35 and 92



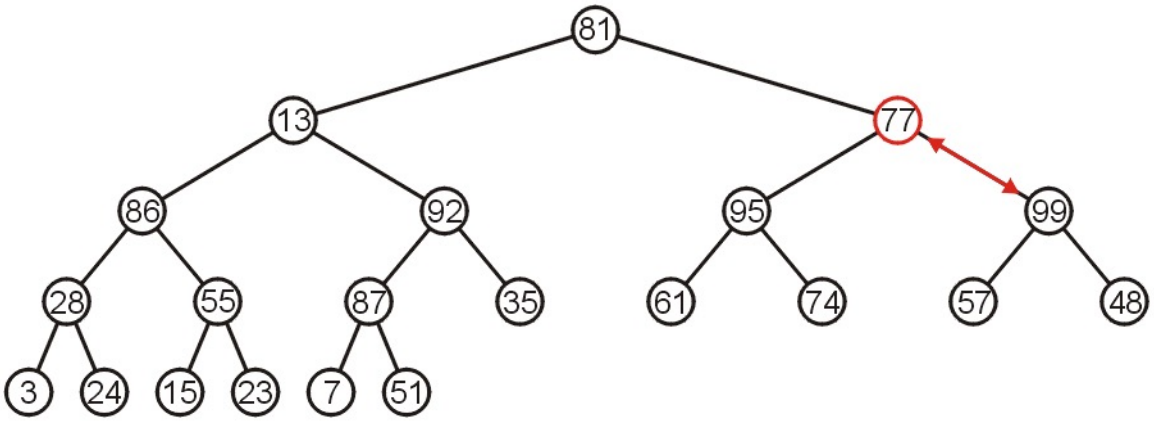
In-place Heapification

- The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28



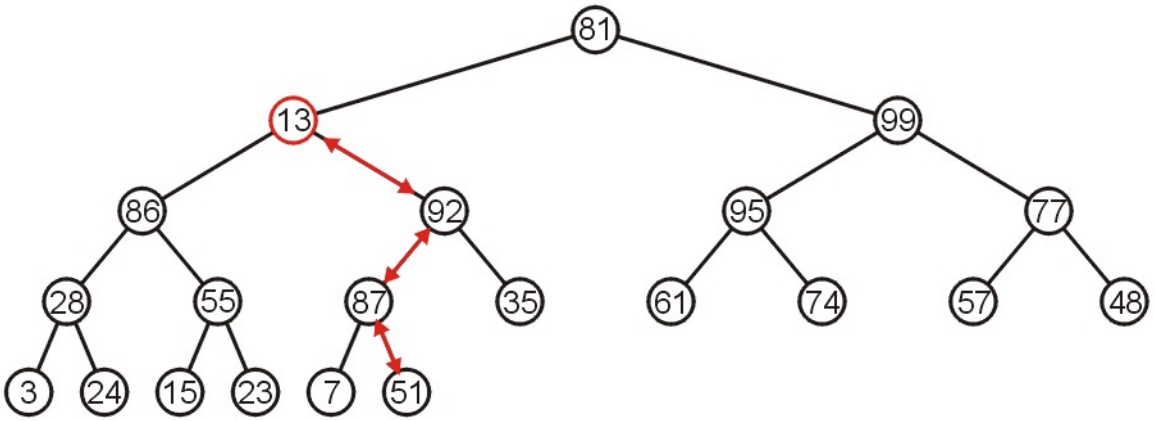
In-place Heapification

- The right-most subtree of the next higher level may be turned into a max-heap by swapping 77 and 99



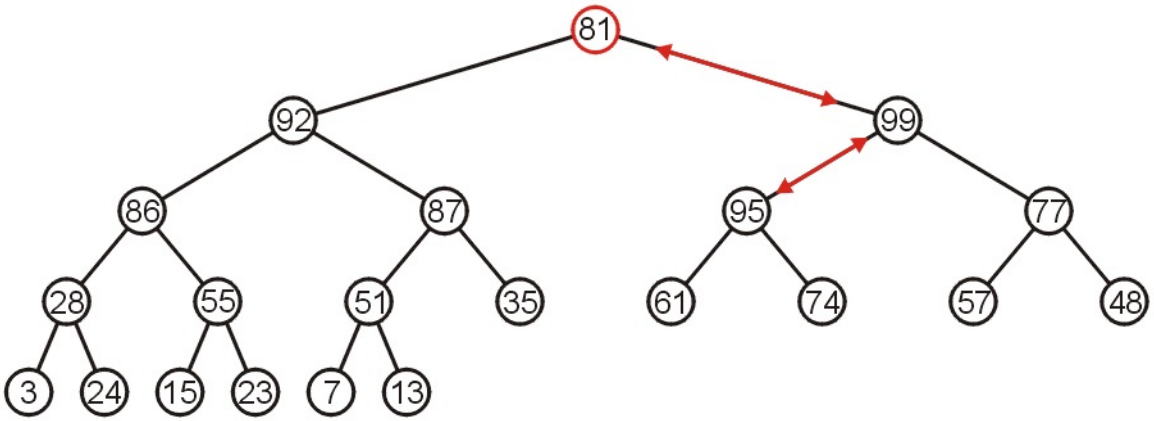
In-place Heapification

- However, to turn the next subtree into a max-heap requires that 13 be percolated down to a leaf node



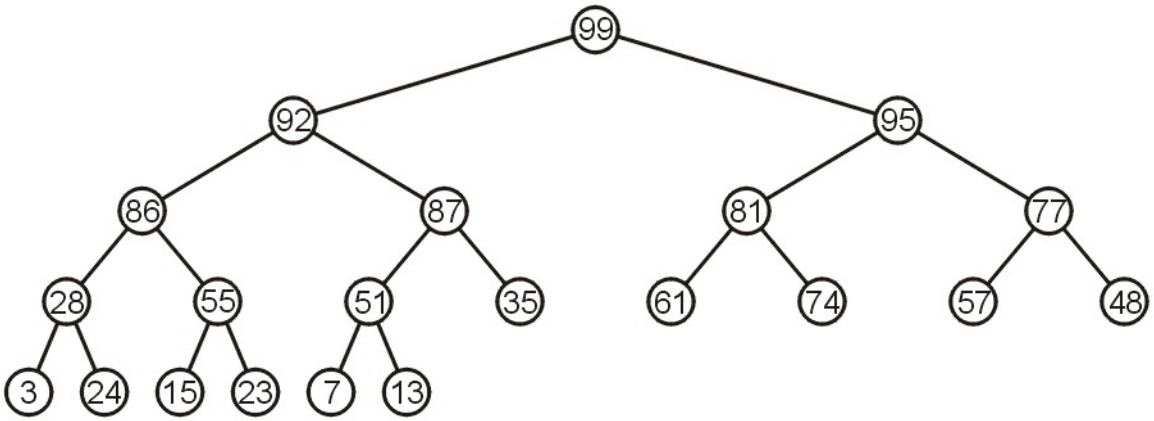
In-place Heapification

- The root need only be percolated down by two levels



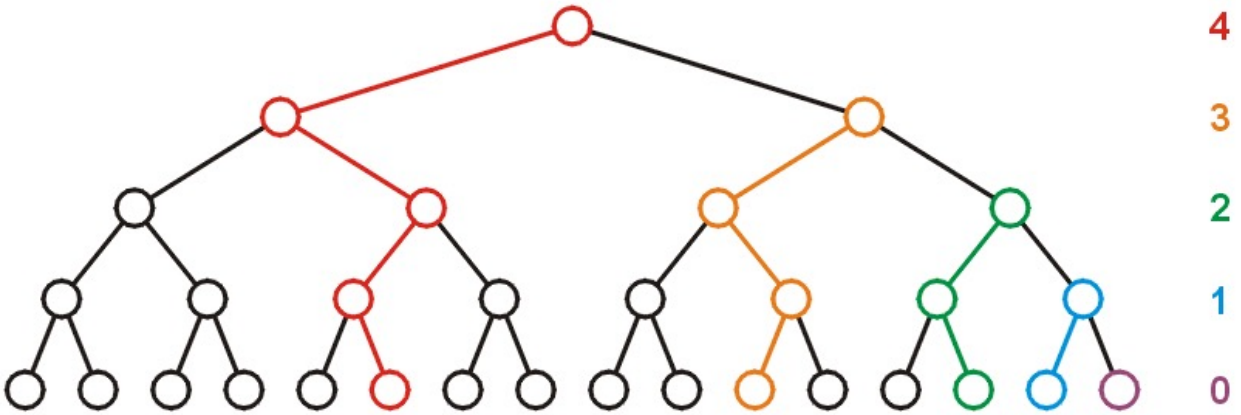
In-place Heapification

- The final product is a max-heap



Run-time Analysis of Heapify

- Considering a perfect tree of height *h*:
  - The maximum number of swaps which a second-lowest level would experience is 1, the next higher level, 2, and so on



Run-time Analysis of Heapify

- At depth k
  - There are at most 2^k nodes
  - Nodes at depth k percolate down at most $h - k$ levels
  - In the worst case, this requires a total of $2^k(h - k)$ swaps

- Writing this sum mathematically, we get:

$$\sum_{k=0}^{h} 2^k(h-k) = \left(2^{h+1} - 1\right) - (h+1)$$

- Recall that for a perfect tree
  - $n = 2^{h+1} - 1$ and $h + 1 = \lg(n + 1)$

$$\sum_{k=0}^{h} 2^k(h-k) = n - \lg(n+1) = \Theta(n)$$

- So, bottom-up heapification takes $\Theta(n)$

Runtime Analysis of Heap Sort

- 1. Heapification runs in $\Theta(n)$
- 2. Popping n items from a heap of size n, runs in $\Theta(n \ln(n))$ time
- So, the total algorithm will run in $\Theta(n \ln(n))$ time

Run-time Summary

- The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

Summary

- We have seen our first in-place $\Theta(n \ln(n))$ sorting algorithm:
  - Convert the unsorted list into a max-heap as complete array
  - Pop the top n times and place that object into the vacancy at the end
  - It requires $\Theta(1)$ additional memory—it is truly in-place

- It is a nice algorithm; however, we will see two other $n \ln(n)$ algorithms;
  - Merge sort requires $\Theta(n)$ additional memory
  - Quick sort requires $\Theta(\ln(n))$ additional memory

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Heapsort

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.3, p.144-8.

[3] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, Ch. 7, p.140-9.

[4] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §7.5, p.270-4.

Merge Sort

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic covers merge sort
  - A recursive divide-and-conquer algorithm
  - Merging two lists
  - The merge sort algorithm
  - A run-time analysis

Merge Sort

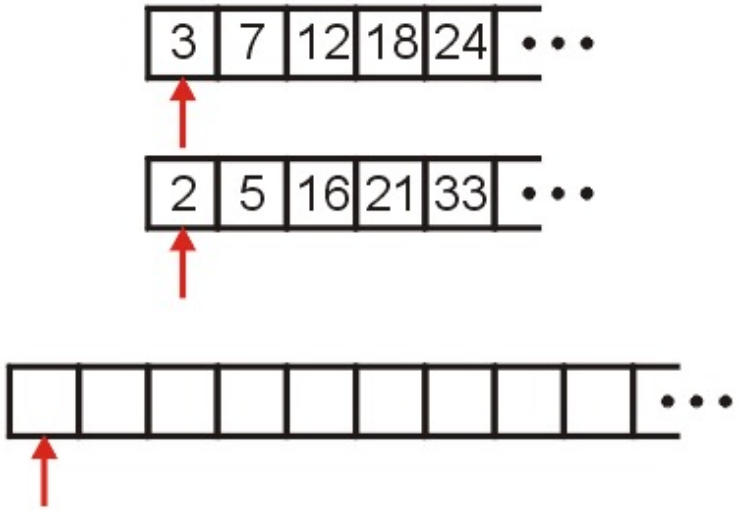
- The merge sort algorithm is defined recursively:
  - If the list is of size 1, it is sorted—we are done;
  - Otherwise:
    - Divide an unsorted list into two sub-lists,
    - Sort each sub-list recursively using merge sort, and
    - Merge the two sorted sub-lists into a single sorted list

- This is the first significant *divide-and-conquer* algorithm we will see

- Question: How quickly can we recombine the two sub-lists into a single sorted list?

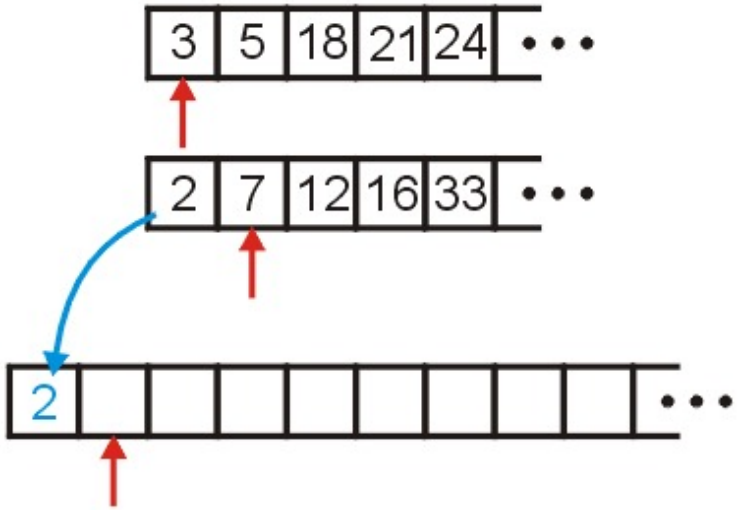
Merging Example

- Consider the two sorted arrays and an empty array
- Define three indices at the start of each array



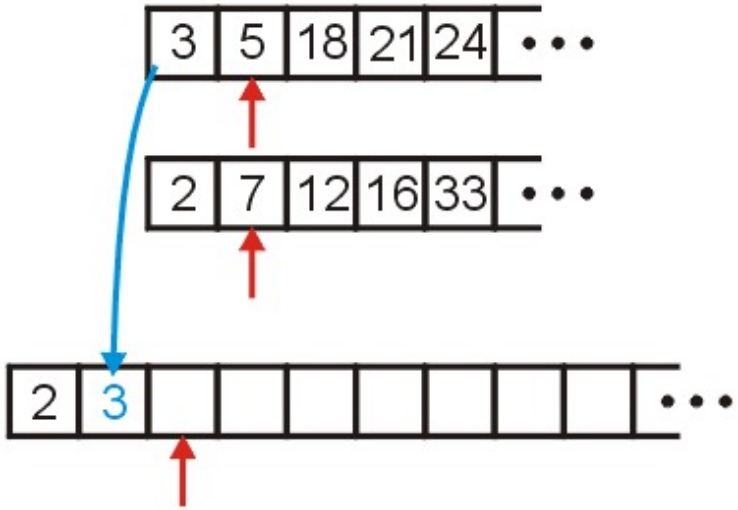
Merging Example

- We compare 2 and 3: 2 < 3
  - Copy 2 down
  - Increment the corresponding indices



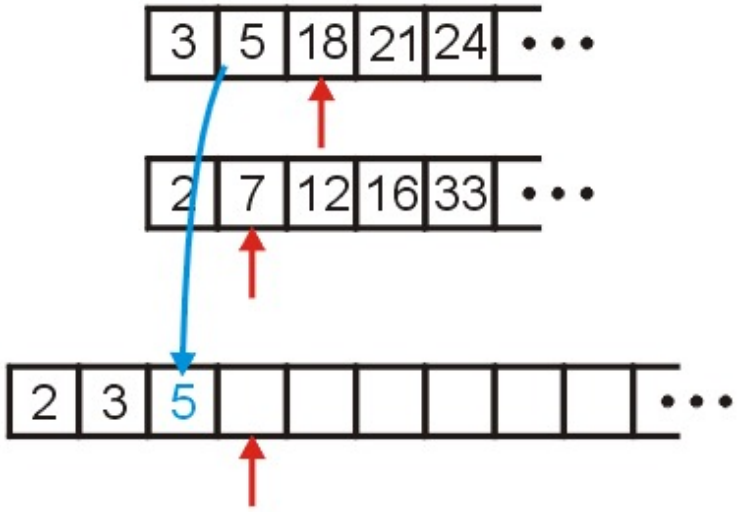
Merging Example

- We compare 3 and 7
  - Copy 3 down
  - Increment the corresponding indices



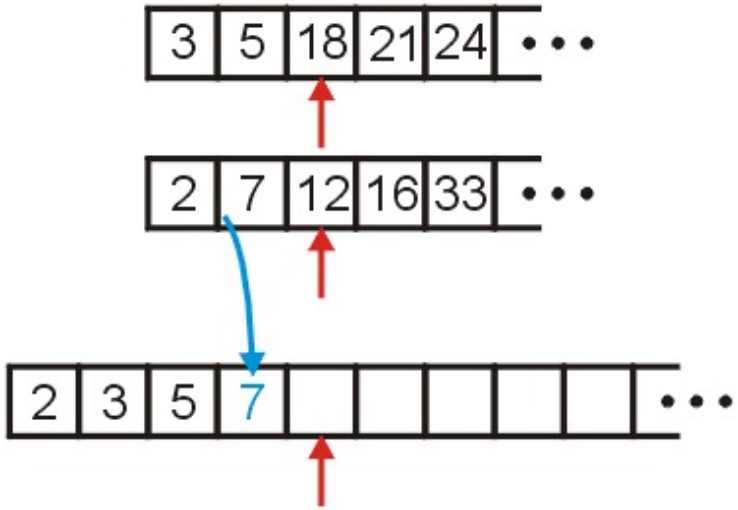
Merging Example

- We compare 5 and 7
  - Copy 5 down
  - Increment the appropriate indices



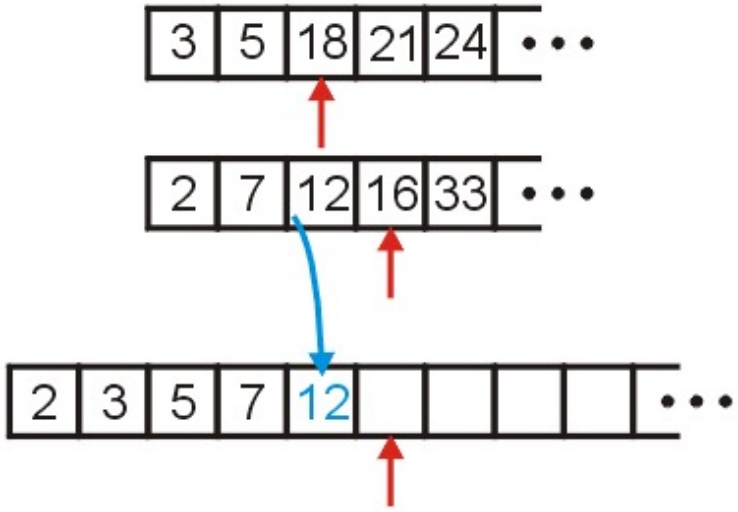
Merging Example

- We compare 18 and 7
  - Copy 7 down
  - Increment...



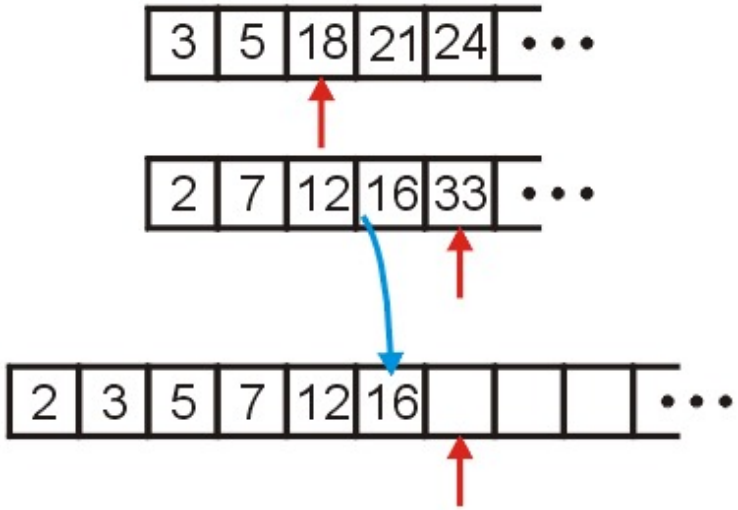
Merging Example

- We compare 18 and 12
  - Copy 12 down
  - Increment...



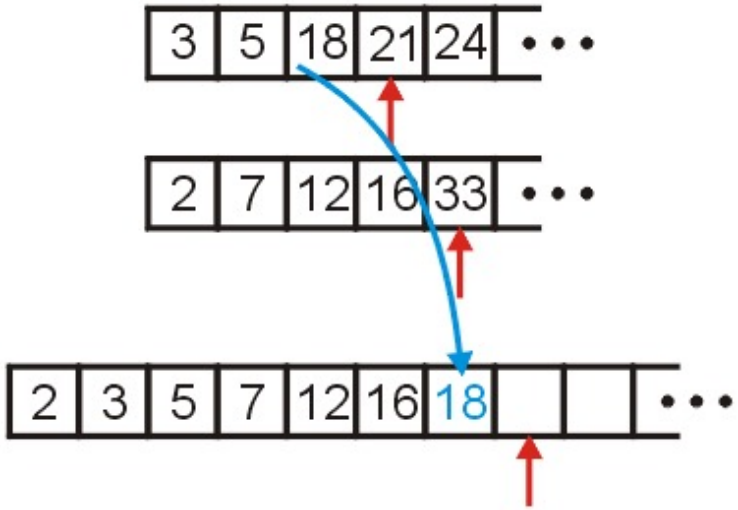
Merging Example

- We compare 18 and 16
  - Copy 16 down
  - Increment...



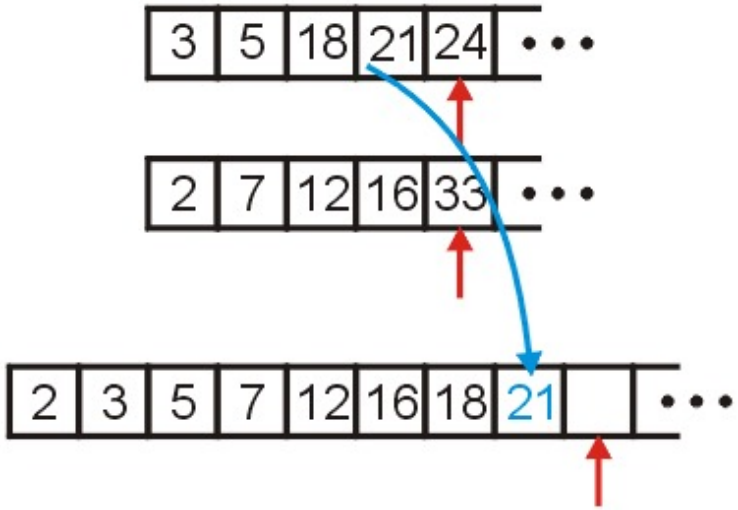
Merging Example

- We compare 18 and 33
  - Copy 18 down
  - Increment...



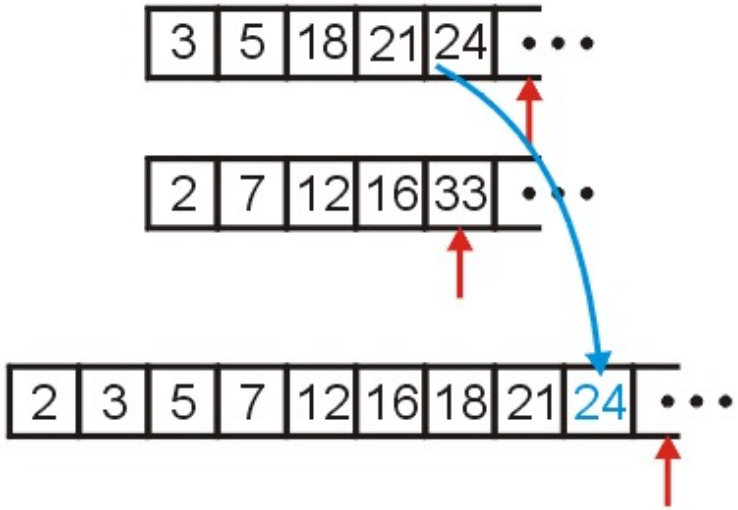
Merging Example

- We compare 21 and 33
  - Copy 21 down
  - Increment...



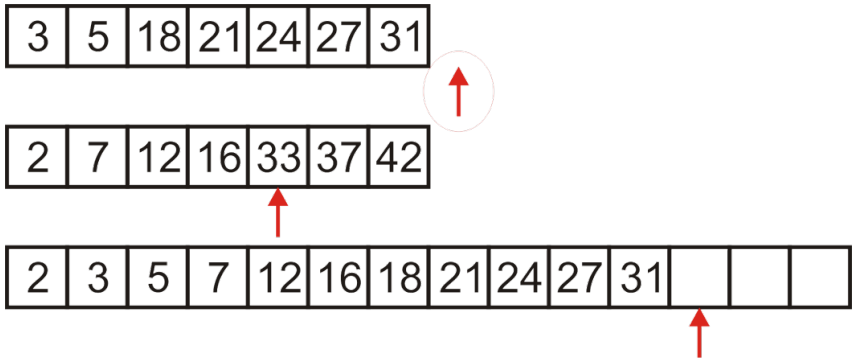
Merging Example

- We compare 24 and 33
  - Copy 24 down
  - Increment...

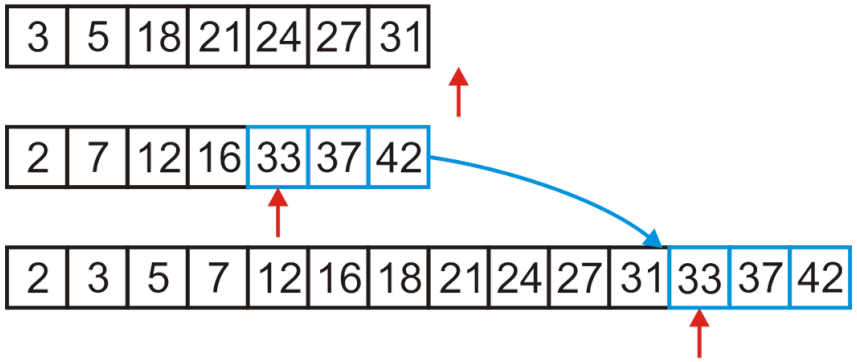


Merging Example

- We would continue until we have passed beyond the limit of one of the two arrays



- After this, the rest can be simply copied



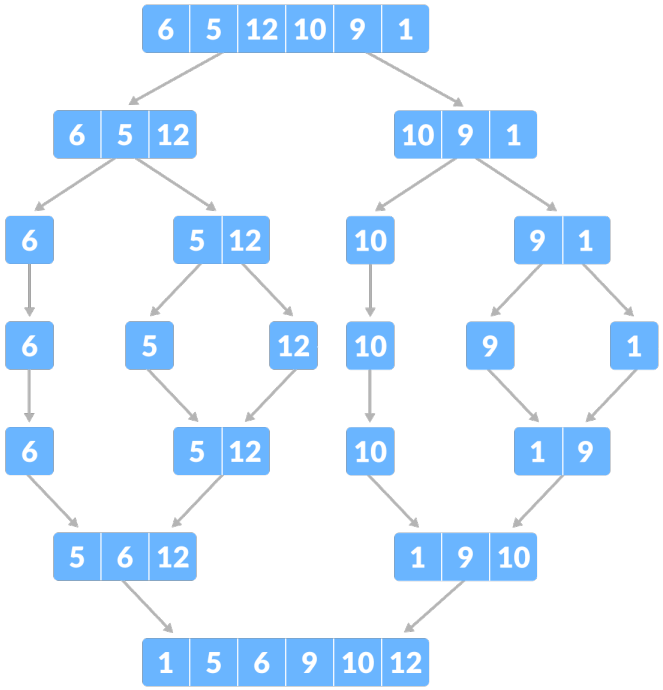
Analysis of merging

- Suppose the sorted arrays, array1 and array2, are of size n1 and n2, respectively
  - Then merging would be performed in $\Theta(n_1 + n_2)$ time

- Problem: We cannot merge two arrays in-place
  - This algorithm always required the allocation of a new array
  - Therefore, the memory requirements are also $\Theta(n)$

The Algorithm

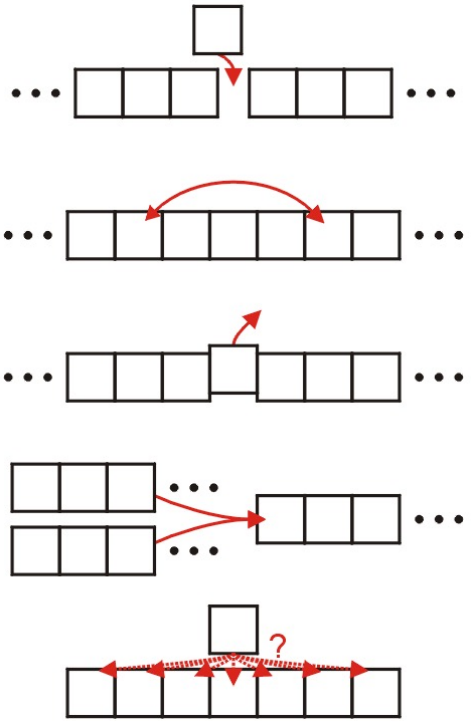
- The algorithm:
  - Split the list into two approximately equal sub-lists
  - Recursively call merge sort on both sub lists
  - Merge the resulting sorted lists



Ref: https://www.programiz.com/dsa/merge-sort

The Algorithm

- Recall the five sorting techniques:
  - Insertion
  - Exchange
  - Selection
  - Merging
  - Distribution



- Clearly merge sort falls into the fourth category

Run-time Analysis of Merge Sort

- Thus, the time required to sort an array of size $n > 1$ is:
  - the time required to sort the first half,
  - the time required to sort the second half, and
  - the time required to merge the two lists

Representing these with the recurrence relation:

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Divide by n,

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

The equation holds for any n, which is power of 2

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

Then telescope a sum of all equations,

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

Finally we get

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

Run-time Summary

- The following table summarizes the run-times of merge sort

Case	Run Time	Comments
Worst	$O(n \ln(n))$	No worst case
Average	$O(n \ln(n))$	
Best	$O(n \ln(n))$	No best case

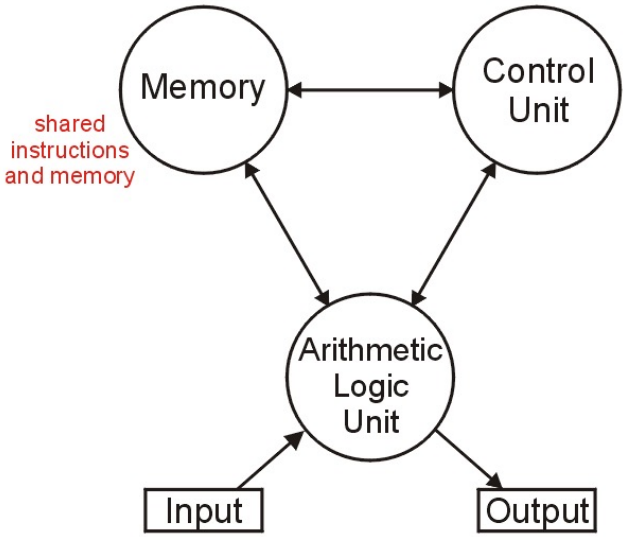
Comments

- Merge sort requires an additional array
  - Heap sort does not require

- Next we see quick sort
  - Faster, on average, than either heap or quick sort
  - Requires $o(n)$ additional memory

Merge Sort

- The (likely) first implementation of merge sort was on the ENIAC in 1945 by John von Neumann
  - The creator of the *von Neumann architecture* used by all modern computers:



http://en.wikipedia.org/wiki/Von_Neumann

Summary

- This topic covered merge sort:
  - Divide an unsorted list into two equal or nearly equal sub lists,
  - Sorts each of the sub lists by calling itself recursively, and then
  - Merges the two sub lists together to form a sorted list

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Sorting_algorithm

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

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

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

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

Quick Sort

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

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

- 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

Idea: Quicksort

- **Merge sort** splits the array 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

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

- 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)?

Worst-case scenario

- 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

Median of three as pivot

- 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

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: $A'[1 \ldots n]$

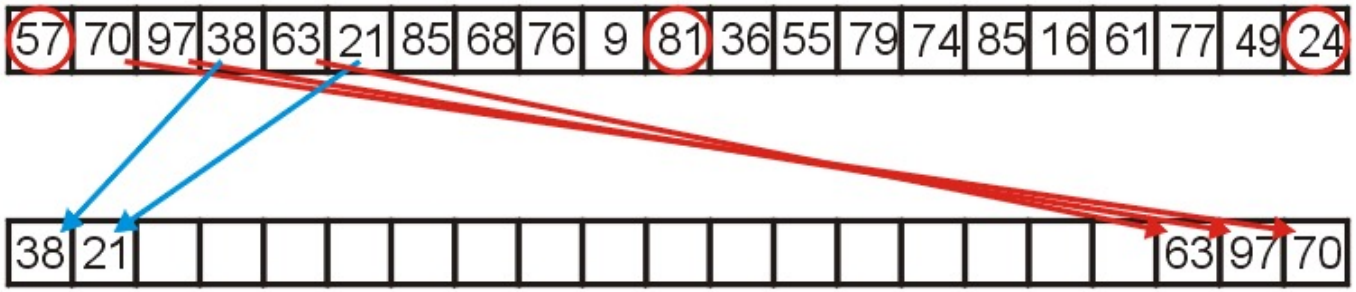
What's the good pivot? (See CLRS Problem 7-5. p209)

- 1) Median: i.e., pick x, where $x = A'[\frac{n+1}{2}]$
  - Median-of-three increases 50% of chances 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

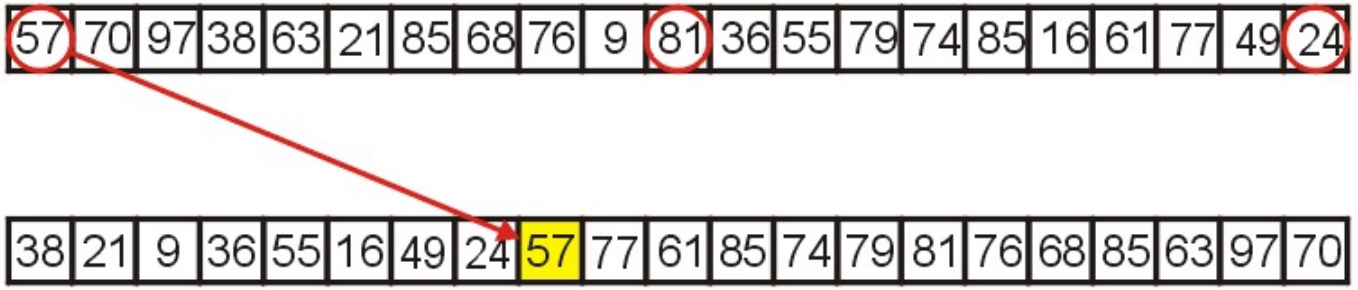
- If we allocate an additional array, we can implement the partitioning by copying elements either to the front or the back of the additional array

- 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



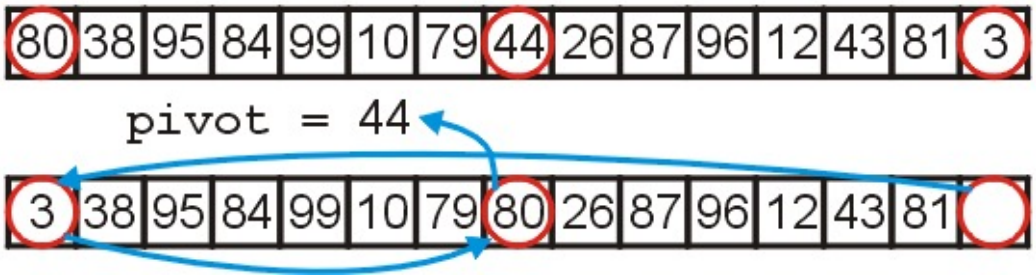
Non In-place Implementation

- Then why not simply using merge sort?
  - Merge sort has no worst case runtime.
  - It 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

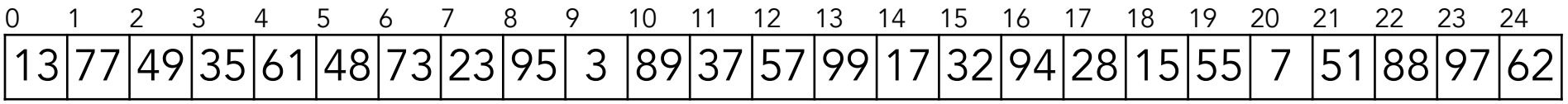
- Step #4. Find two out-of-order entries:
  - Starting from the front, the entry larger than the pivot
  - Starting from the back, the entry smaller than the pivot

- 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



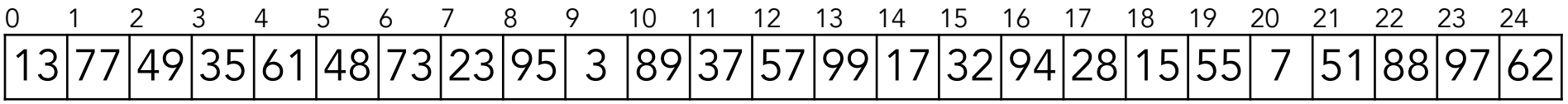
- Notation:

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

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

Quicksort Example

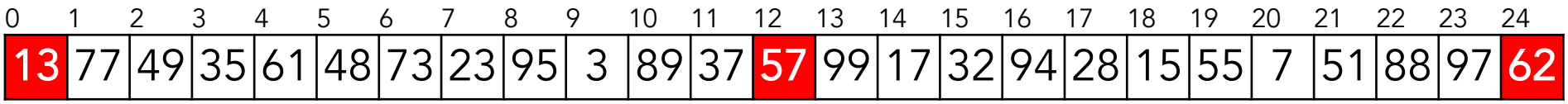
- We call `quicksort( array, 0, 25 )`



`quicksort( array, 0, 25 )`

Quicksort Example

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



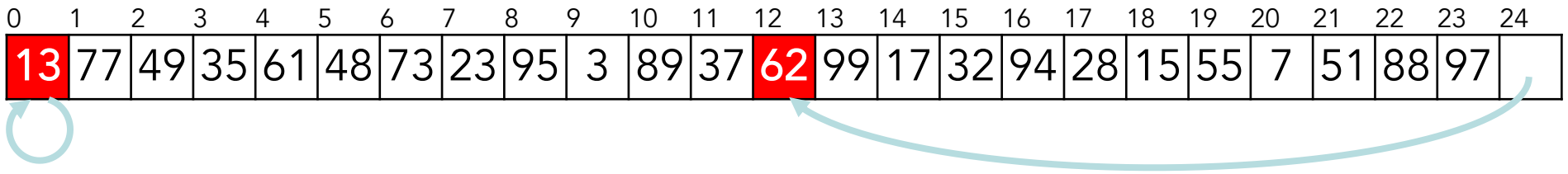
- First, 25 – 0 > 6, so find the midpoint and the pivot

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

`quicksort( array,  0, 25 )`

Quicksort Example

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



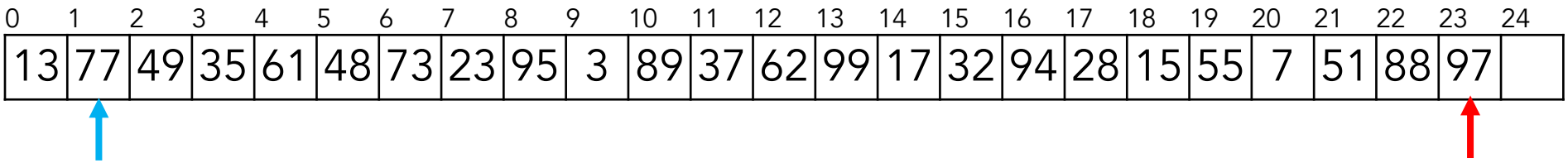
- First, 25 – 0 > 6, so find the midpoint and the pivot

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

`quicksort( array,  0, 25 )`

Quicksort Example

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



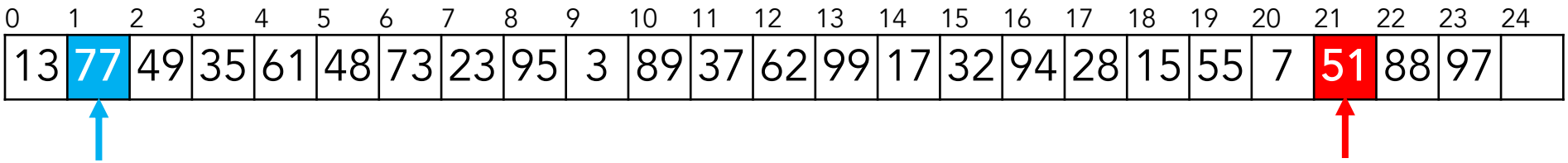
- Starting from the front and back:
  - Find the next element greater than the pivot
  - The last element less than the pivot

```
pivot = 57;
```

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

Quicksort Example

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



- Searching forward and backward:

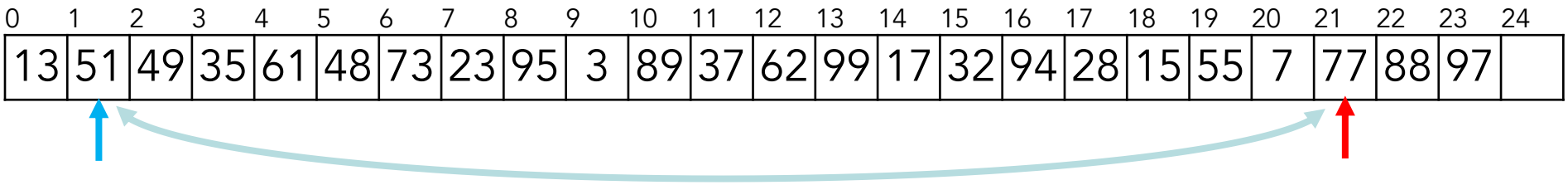
```
low = 1;
high = 21;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Searching forward and backward:

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

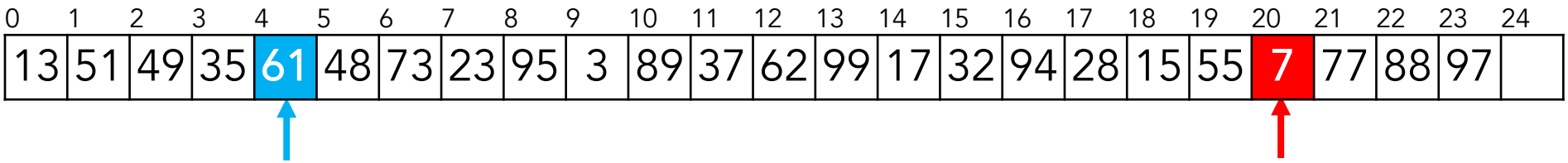
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

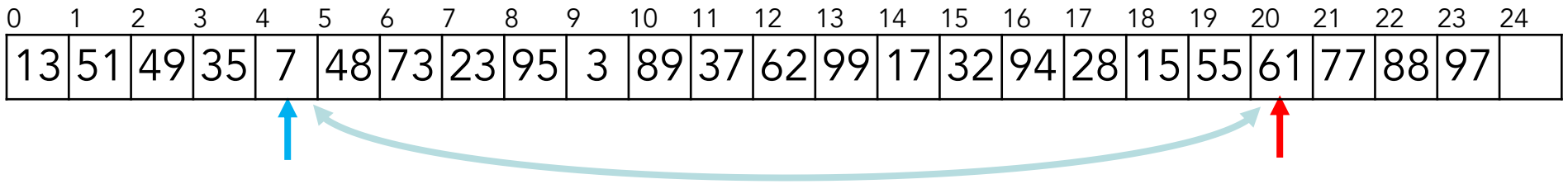
```
low = 4;
high = 20;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

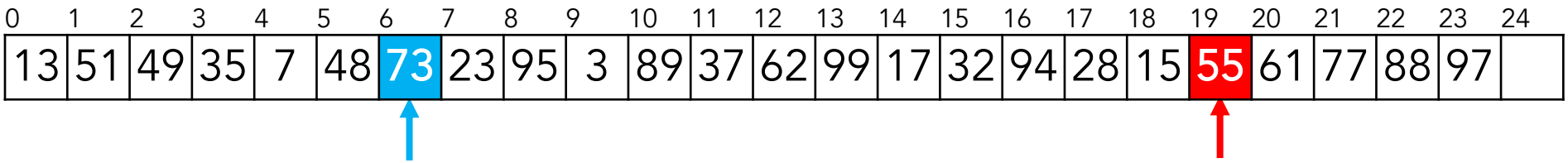
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

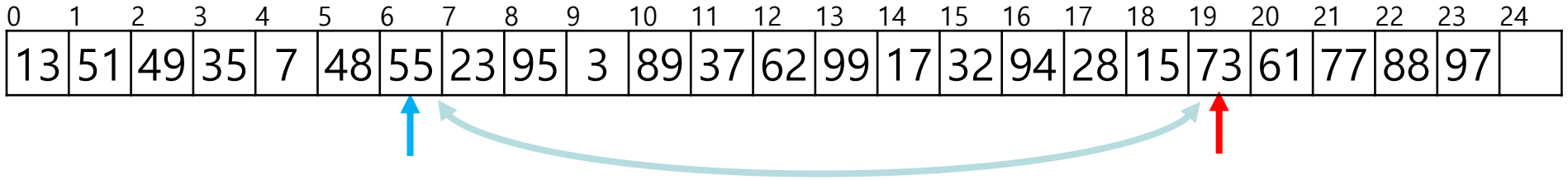
```
low = 6;
high = 19;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

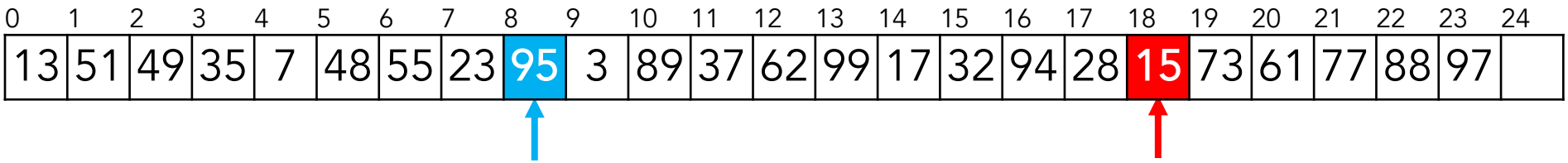
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

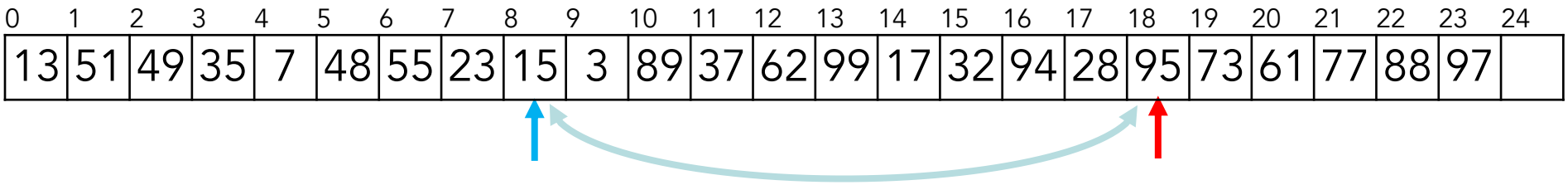
```
low = 8;
high = 18;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

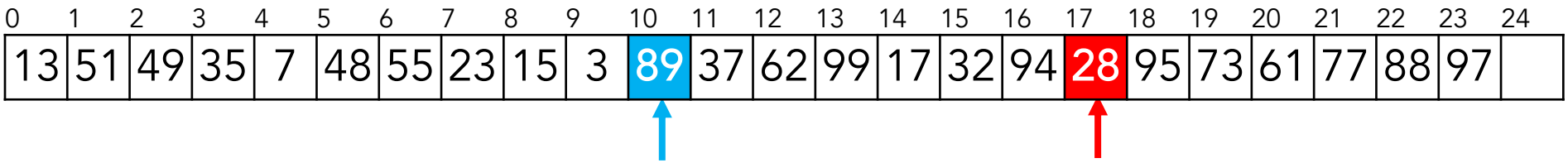
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

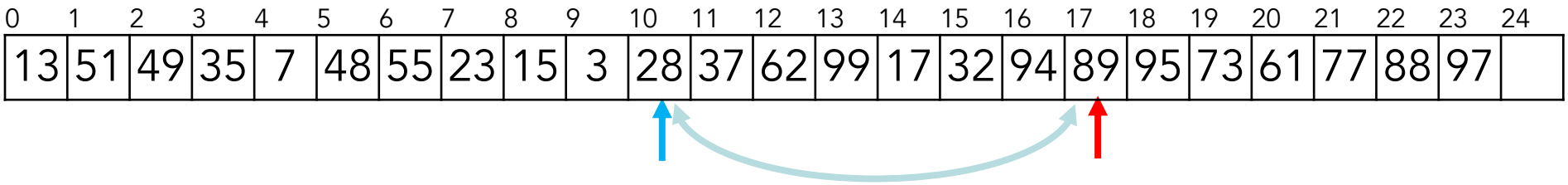
```
low = 10;
high = 17;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

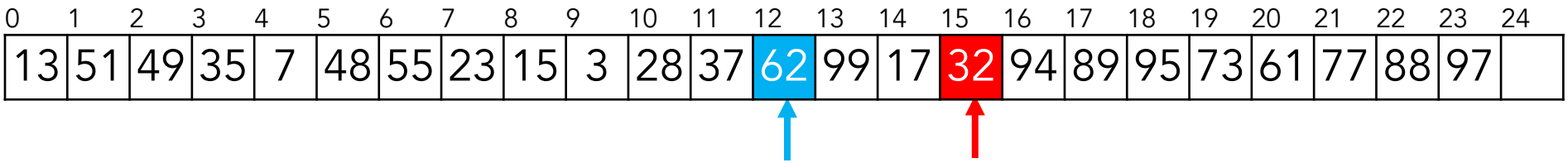
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

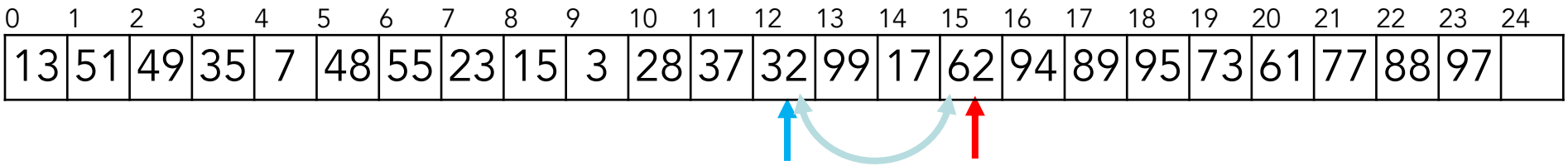
```
low = 12;
high = 15;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

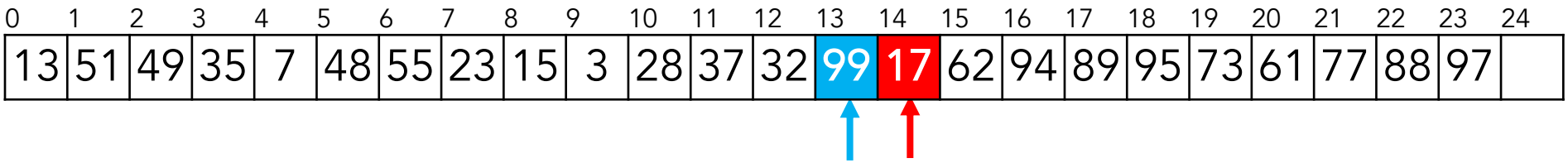
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

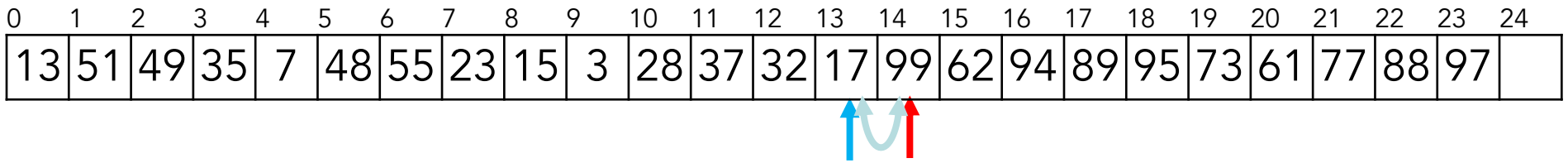
```
low = 13;
high = 14;
```

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

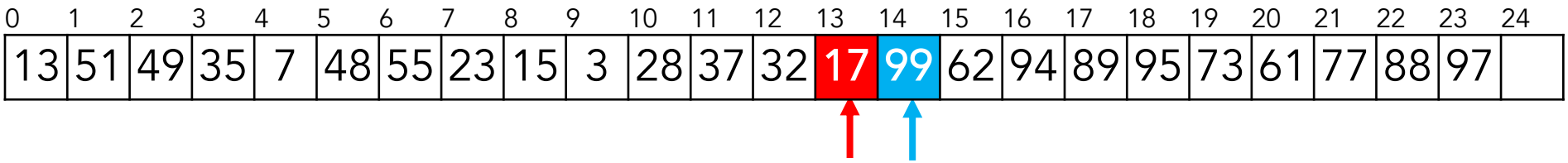
- Swap them

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

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

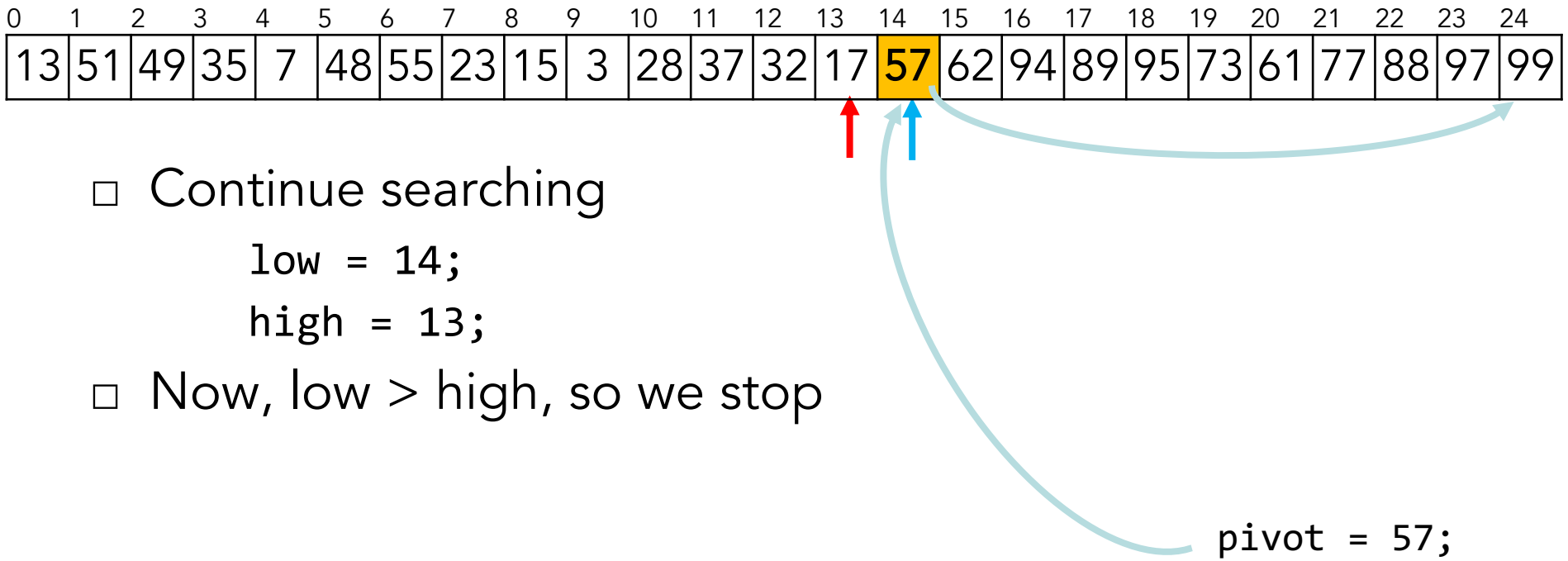
- Now, low > high, so we stop

```
pivot = 57;
```

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

Quicksort Example

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



- Continue searching

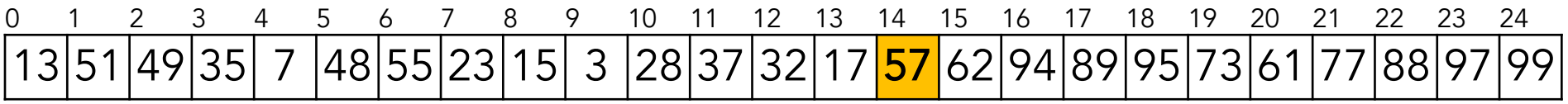
```
low = 14;
high = 13;
```

- Now, low > high, so we stop

`quicksort( array, 0, 25 )`

Quicksort Example

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



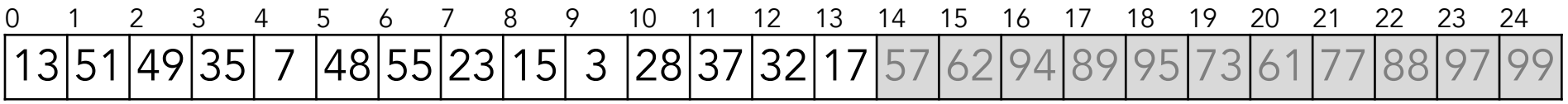
- We now begin calling quicksort recursively on the first half

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

`quicksort( array,  0, 25 )`

Quicksort Example

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



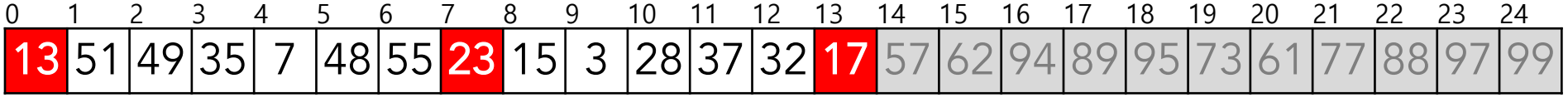
- First, 14 – 0 > 6, so find the midpoint and the pivot

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

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

Quicksort Example

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



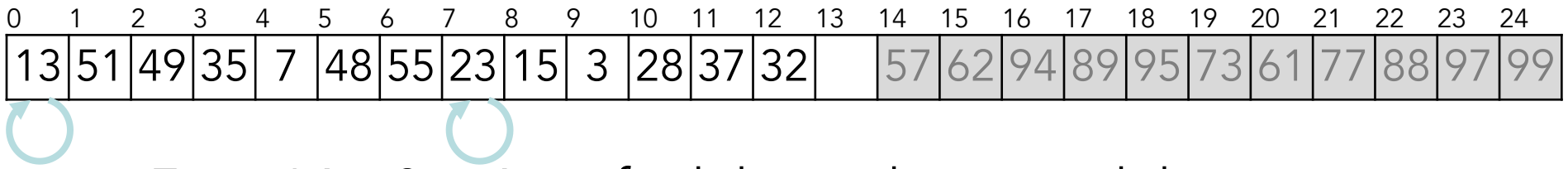
- First, 14 – 0 > 6, so find the midpoint and the pivot

```
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

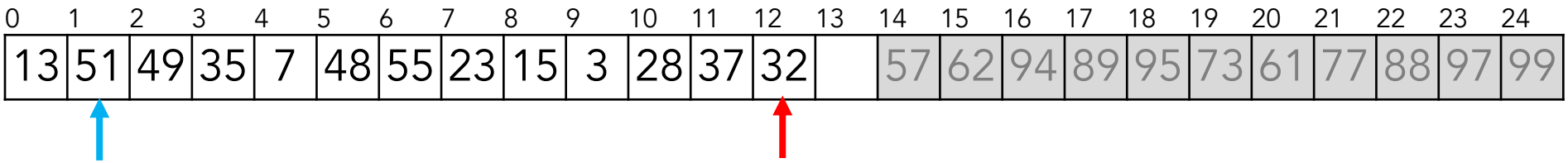
```
midpoint = (0 + 14)/2; // == 7
```

```
pivot = 17;
```

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

Quicksort Example

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



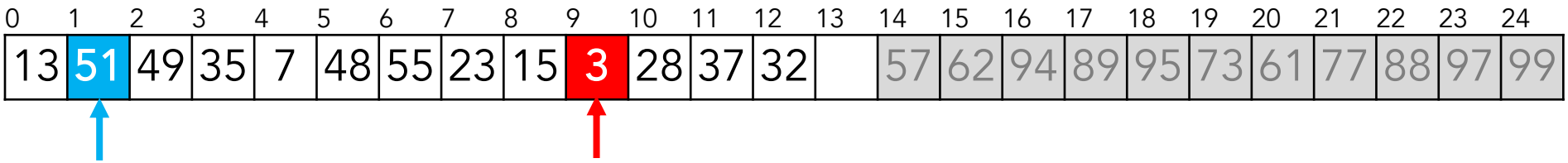
- Starting from the front and back:
  - Find the next element greater than the pivot
  - The last element less than the pivot

```
pivot = 17;
```

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

Quicksort Example

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



- Searching forward and backward:

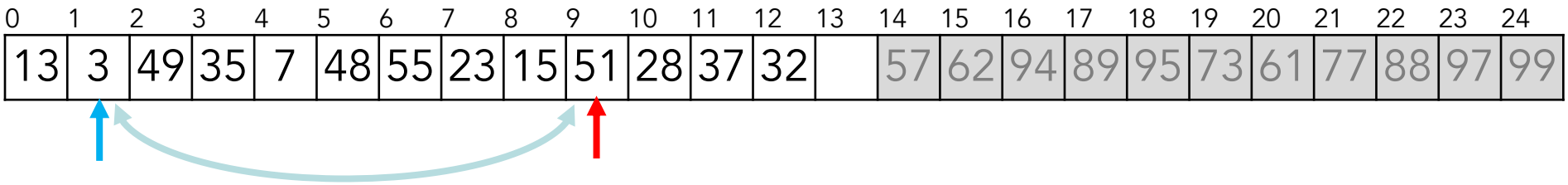
```
low = 1;
high = 9;
```

```
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;
```

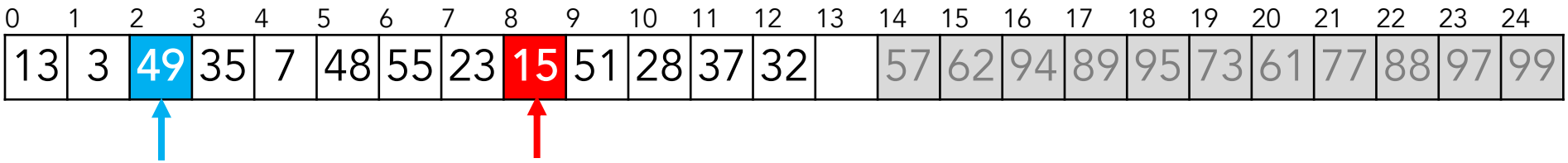
- Swap them

```
pivot = 17;
```

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

Quicksort Example

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



- Searching forward and backward:

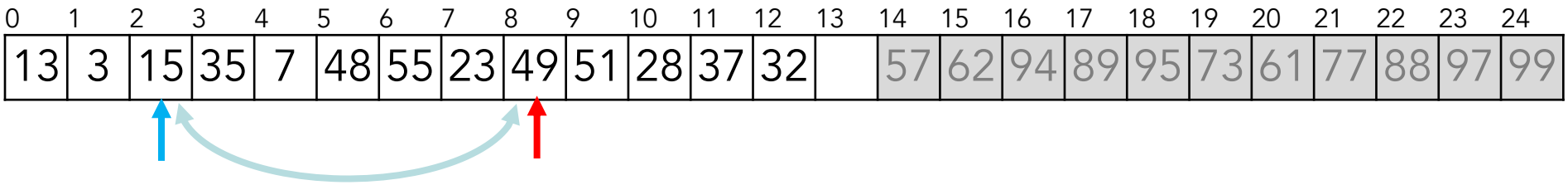
```
low = 2;
high = 8;
```

```
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;
```

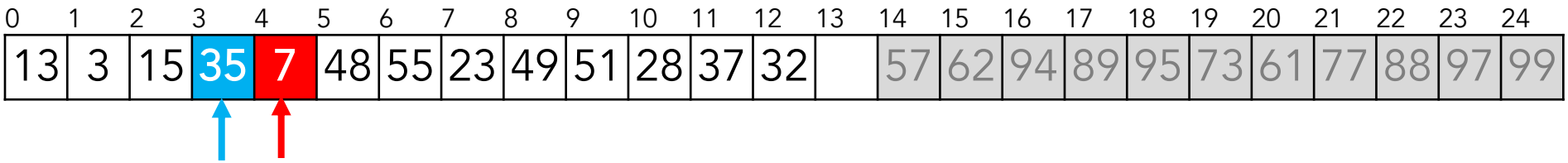
- Swap them

```
pivot = 17;
```

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

Quicksort Example

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



- Searching forward and backward:

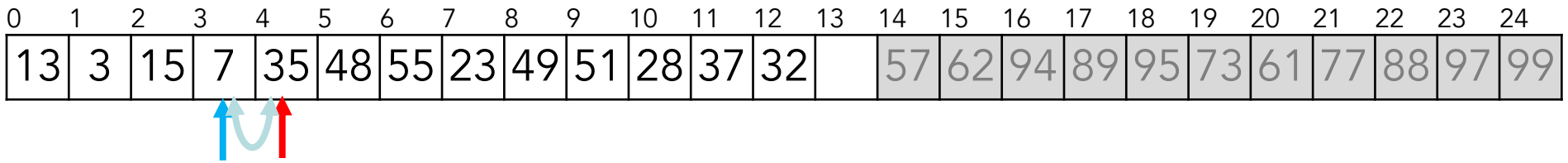
```
low = 3;
high = 4;
```

```
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;
```

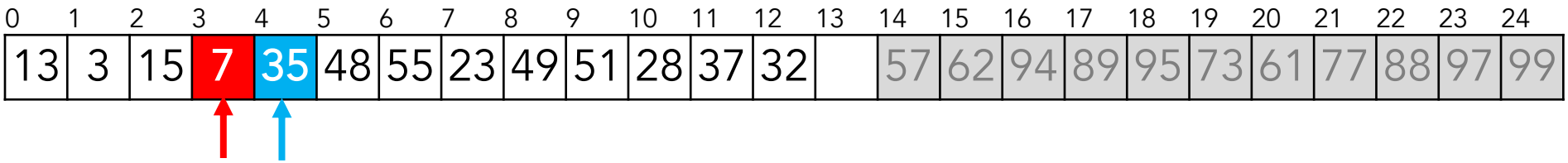
- Swap them

```
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;
```

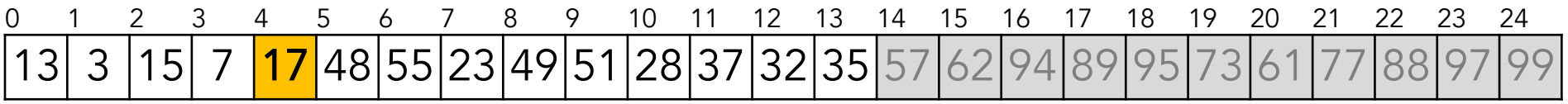
- Now, low > high, so we stop

```
pivot = 17;
```

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

Quicksort Example

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



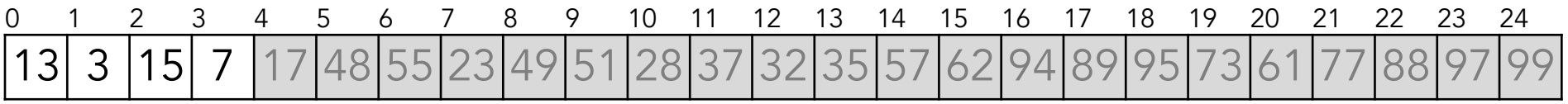
- We continue calling quicksort recursively

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

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

Quicksort Example

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

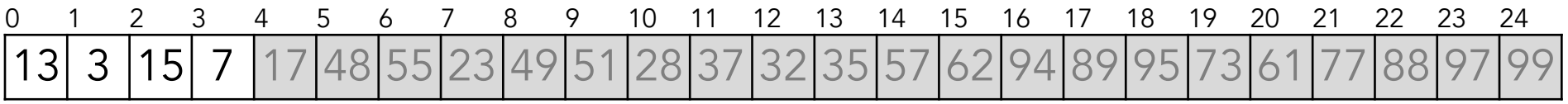


- Now, 4 – 0 ≤ 6, so find we call insertion sort

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

Quicksort Example

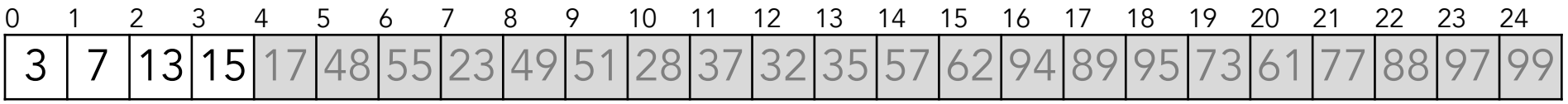
- Insertion sort just sorts the entries from 0 to 3




```
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

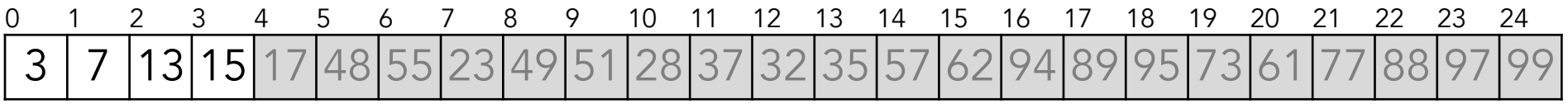


  - This function call completes and so we exit

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

Quicksort Example

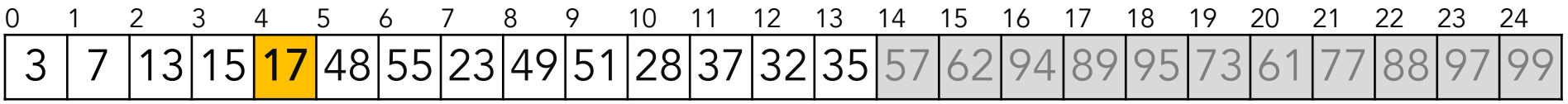
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

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



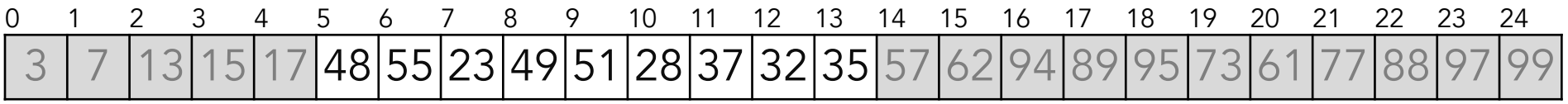
- We continue calling quicksort recursively on the second half

```
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 )`



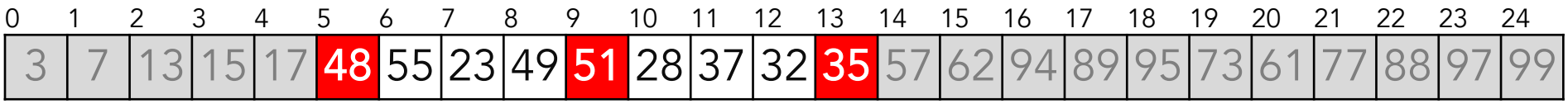
First, 14 – 5 > 6, so find the midpoint and the pivot

```
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 )`



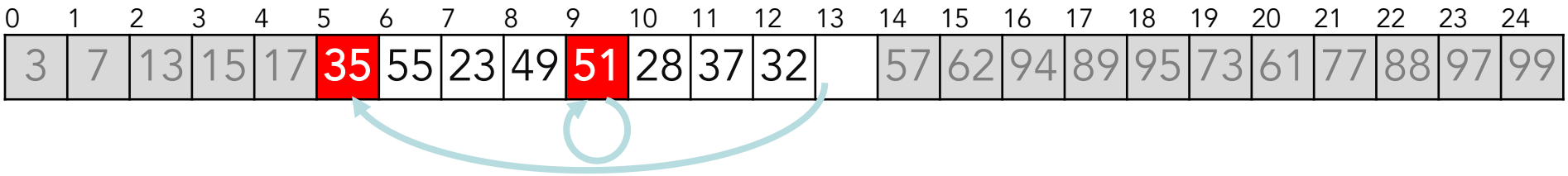
- First, 14 – 5 > 6, so find the midpoint and the pivot

```
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 )`



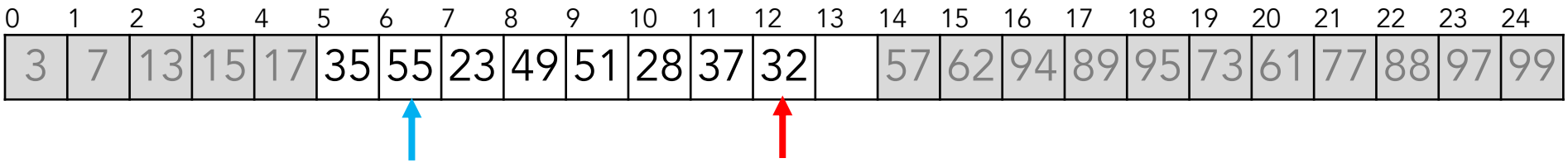
- First, 14 – 5 > 6, so find the midpoint and the pivot

```
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 )`



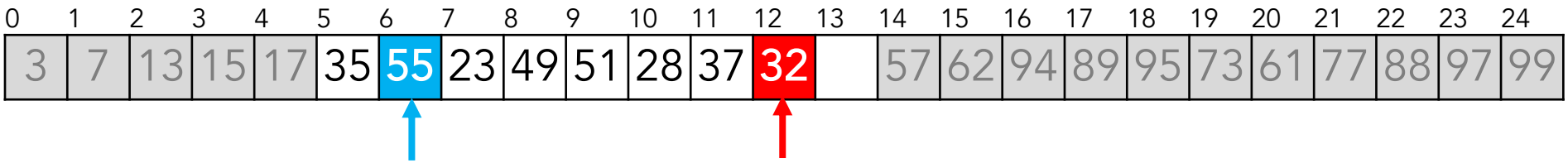
- Starting from the front and back:
  - Find the next element greater than the pivot
  - The last element less than the pivot

```
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:

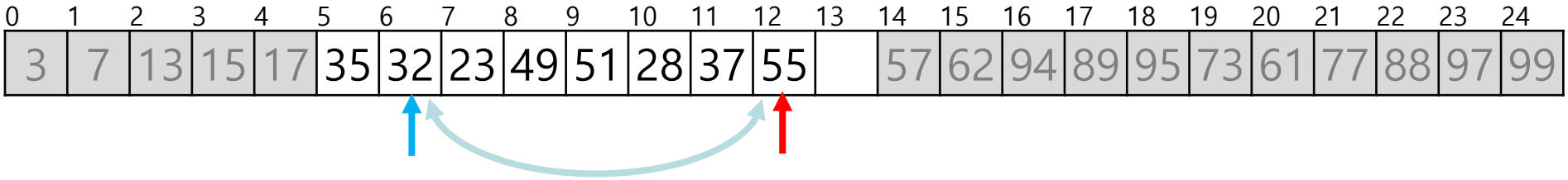
```
low = 6;
high = 12;
```

```
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;
```

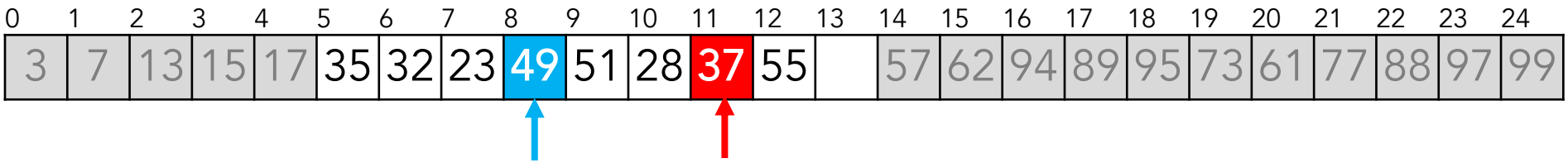
- Swap them

```
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

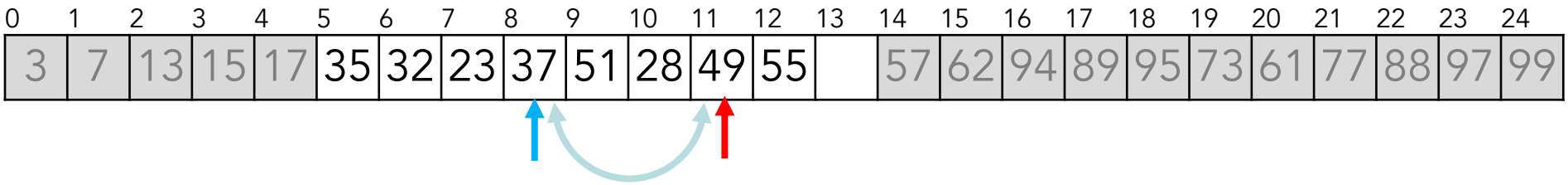
```
low = 8;
high = 11;
```

```
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;
```

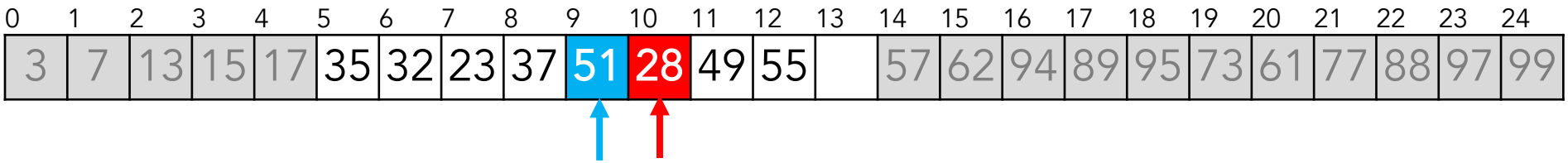
- Swap them

```
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

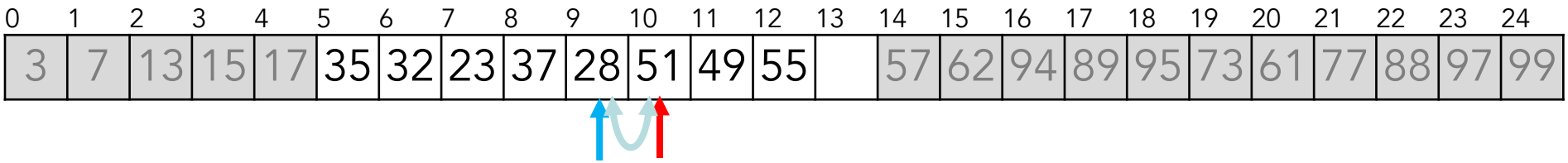
```
low = 8;
high = 11;
```

```
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;
```

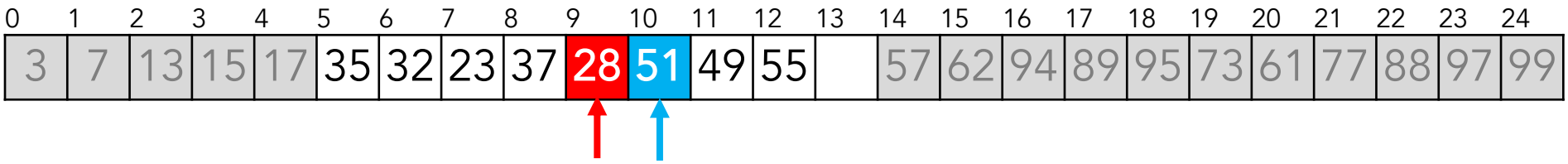
- Swap them

```
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;
```

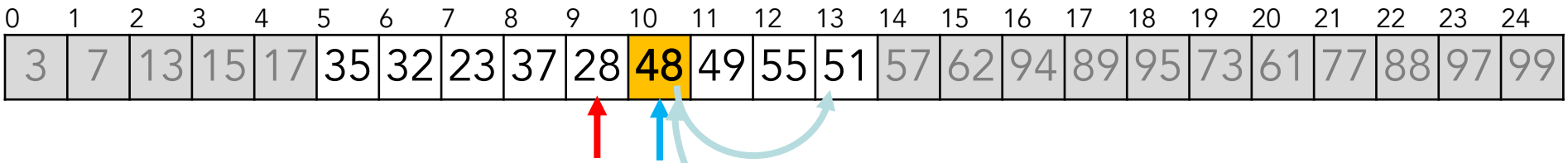
- Now, low > high, so we stop

```
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;
```

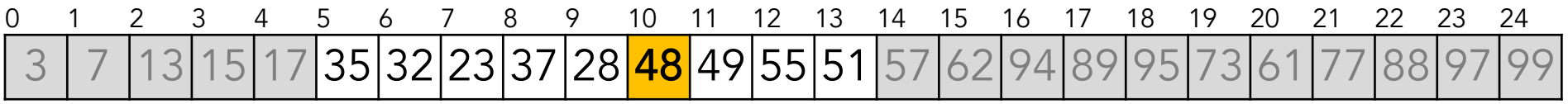
- Now, low > high, so we stop

`pivot = 48;`

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

Quicksort Example

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



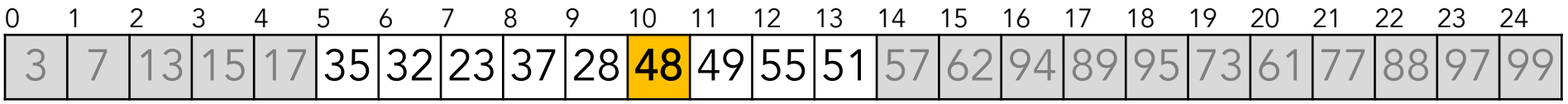
- We now begin calling quicksort recursively on the first half

```
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 )`



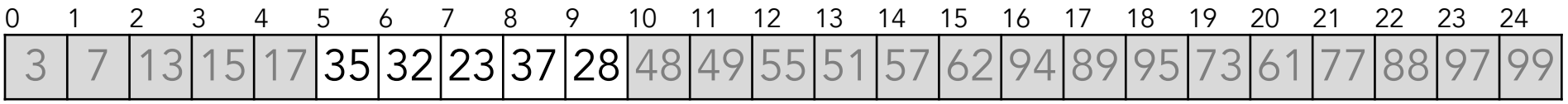
- We now begin calling quicksort recursively

```
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 )`

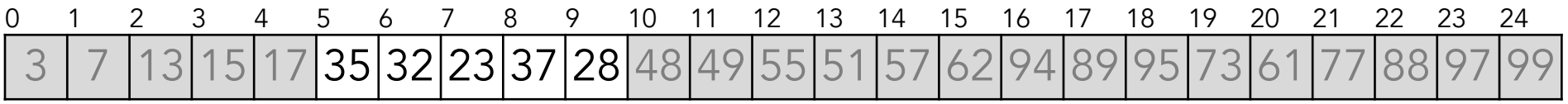


- Now, 10 – 5 ≤ 6, so find we call insertion sort

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

Quicksort Example

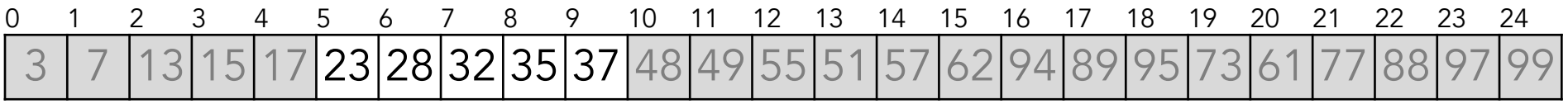
- Insertion sort just sorts the entries from 5 to 9




```
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

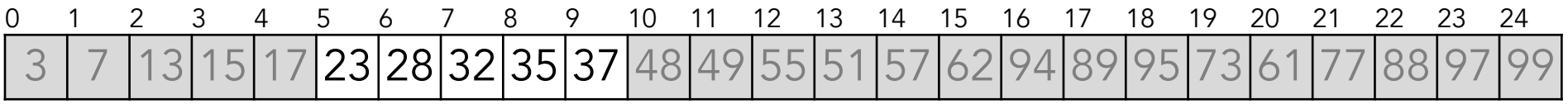


  - This function call completes and so we exit

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

Quicksort Example

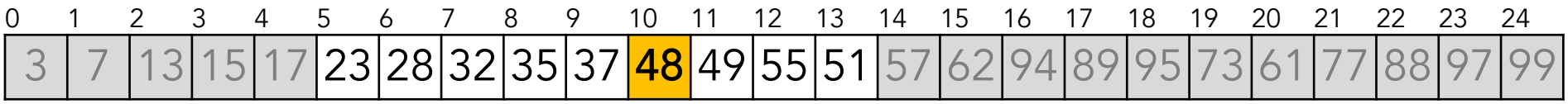
- This call to quicksort is now also finished, so it, too, exits




```
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 )`



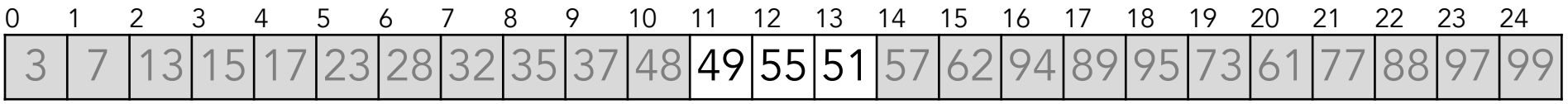
- We continue calling quicksort recursively on the second half

```
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 )`

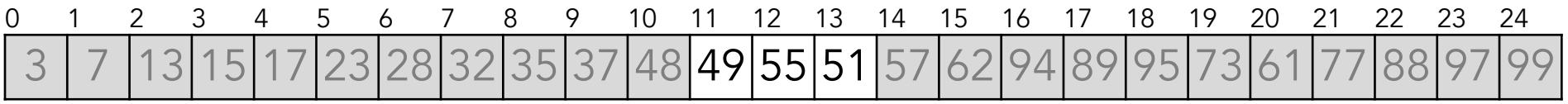


- Now, 15 – 11 ≤ 6, so find we call insertion sort

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

Quicksort Example

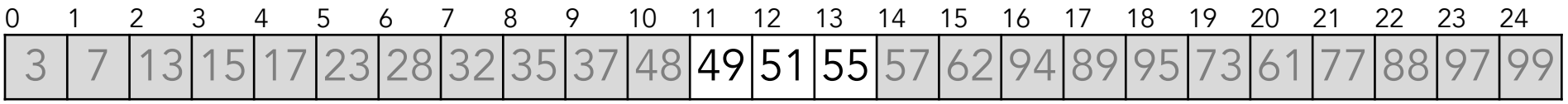
- Insertion sort just sorts the entries from 11 to 14




```
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

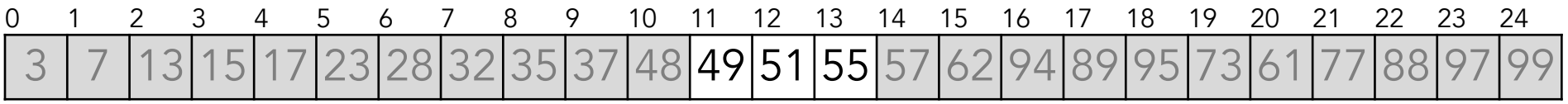


  - This function call completes and so we exit

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

Quicksort Example

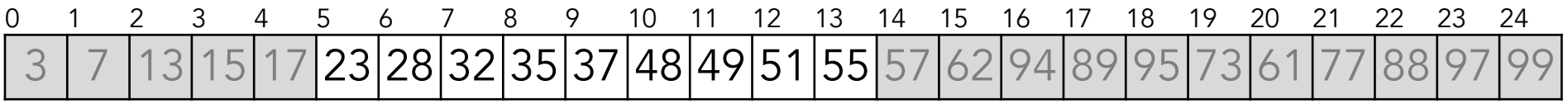
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

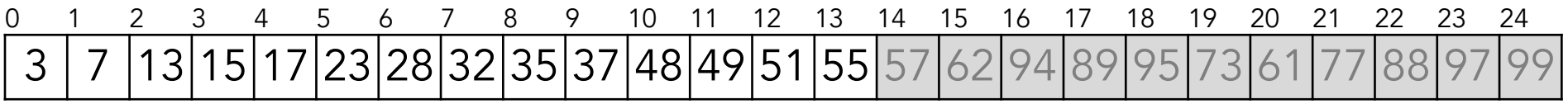
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

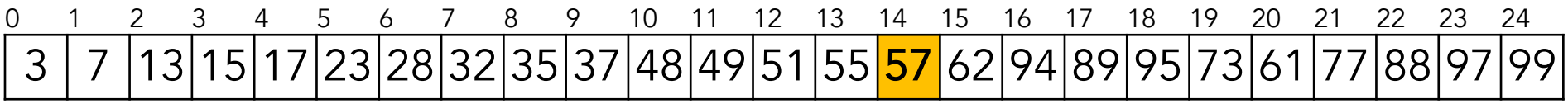
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

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



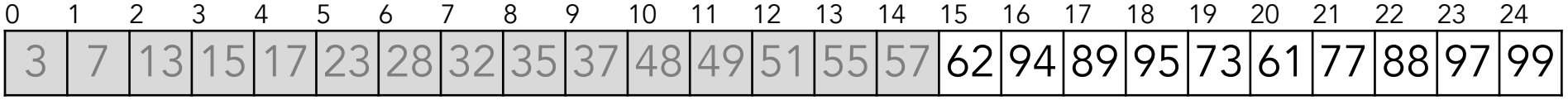
- We continue calling quicksort recursively on the second half

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

`quicksort( array,  0, 25 )`

Quicksort Example

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



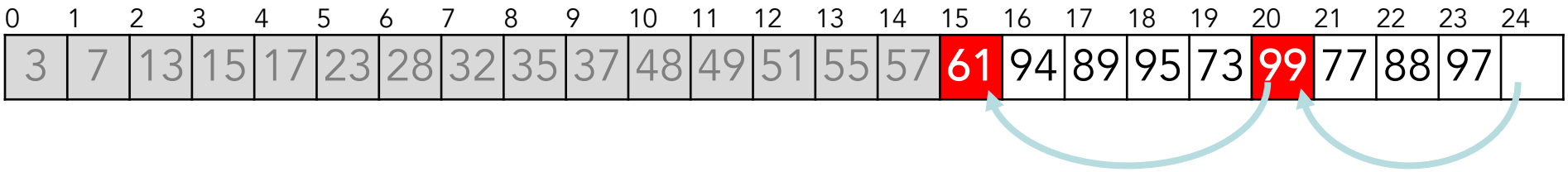
- First, 25 – 15 > 6, so find the midpoint and the pivot

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

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

Quicksort Example

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



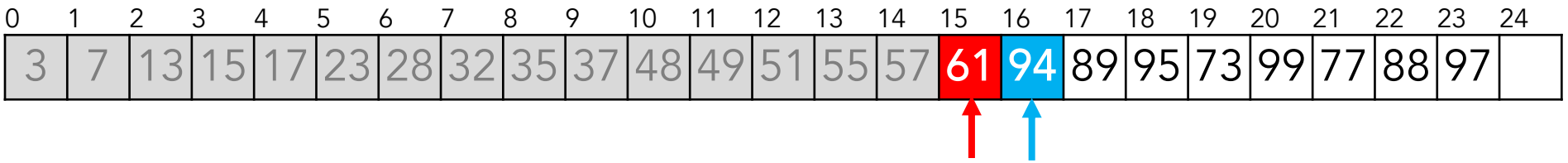
- First, 25 – 15 > 6, so find the midpoint and the pivot

```
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;
```

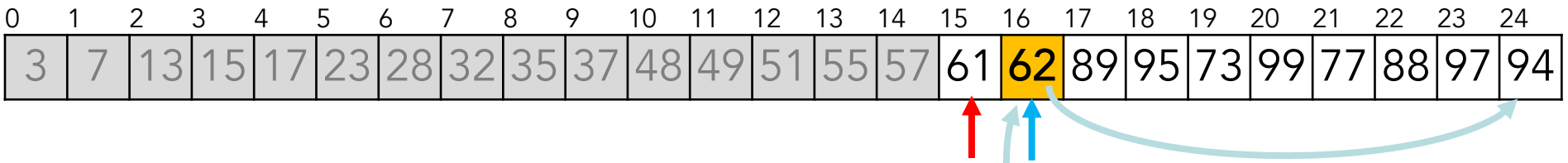
- Now, low > high, so we stop

```
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;
```

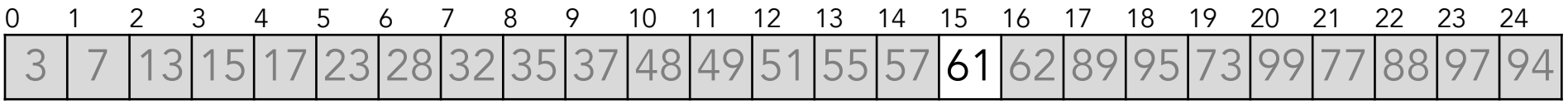
- Now, low > high, so we stop
  - Note, this is the worst-case scenario
  - The pivot is the second smallest element

```
pivot = 62;
```

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

Quicksort Example

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



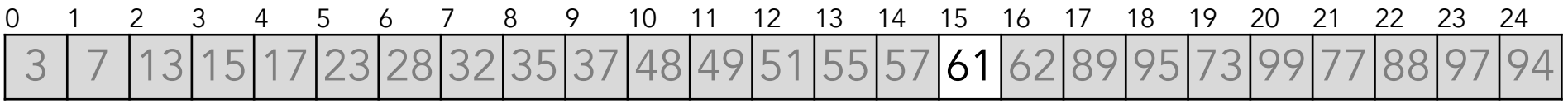
- We continue calling quicksort recursively on the first half

```
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 )`

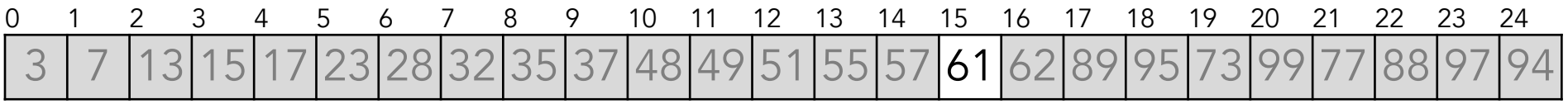


- Now, 16 – 15 ≤ 6, so find we call insertion sort

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

Quicksort Example

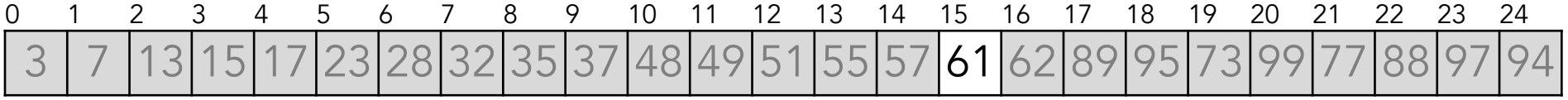
- Insertion sort immediately returns




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

Quicksort Example

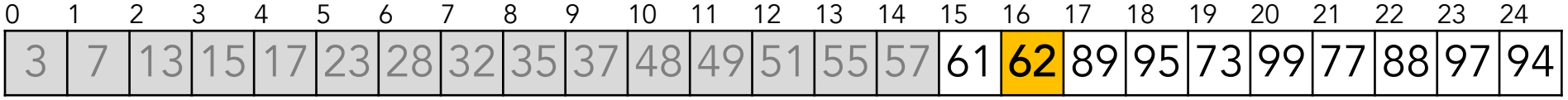
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

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



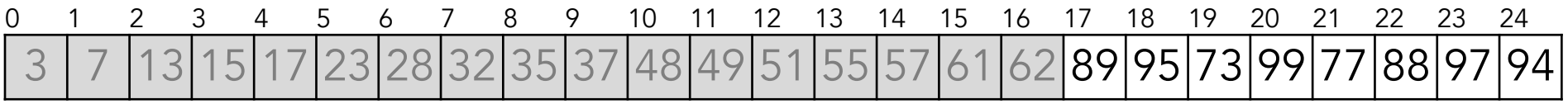
- We continue calling quicksort recursively on the second half

```
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 )`



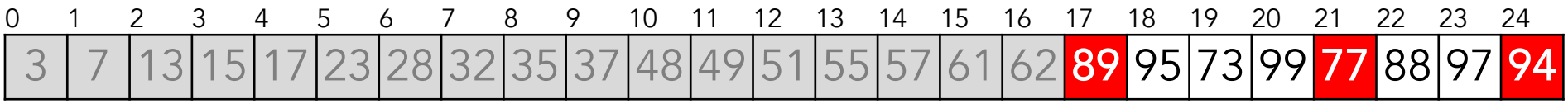
- First, 25 – 17 > 6, so find the midpoint and the pivot

```
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 )`



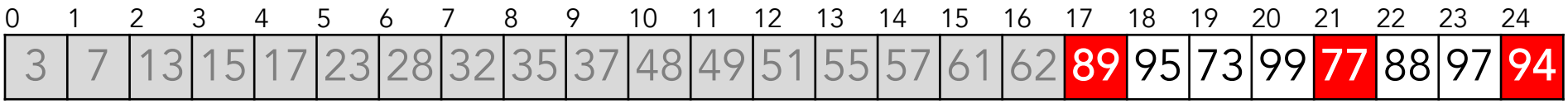
- First, 25 – 17 > 6, so find the midpoint and the pivot

```
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 )`



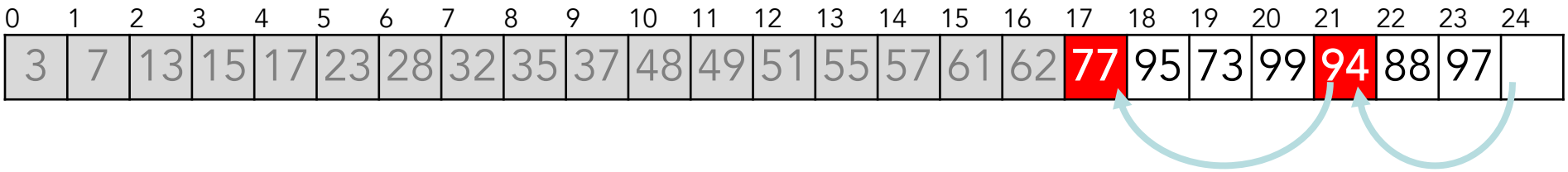
- First, 25 – 17 > 6, so find the midpoint and the pivot

```
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 )`



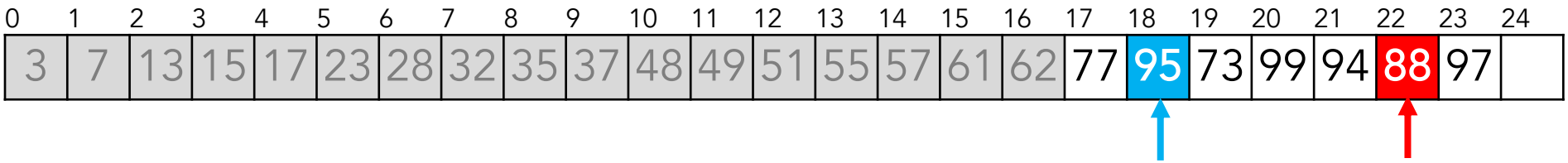
- First, 25 – 17 > 6, so find the midpoint and the pivot

```
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:

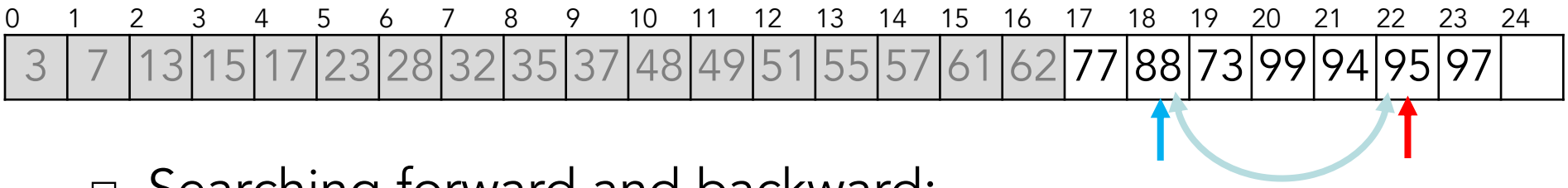
```
low = 18;
high = 22;
```

```
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;
```

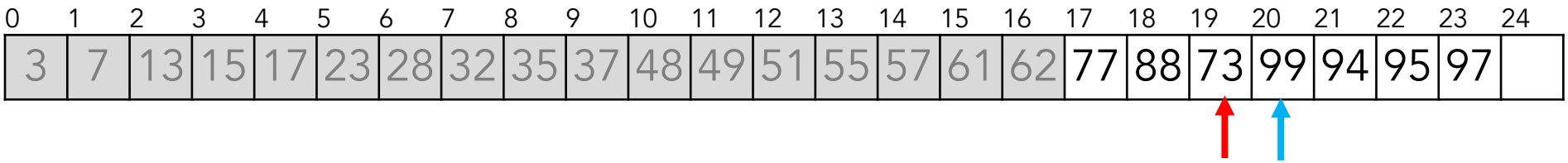
- Swap them

```
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;
```

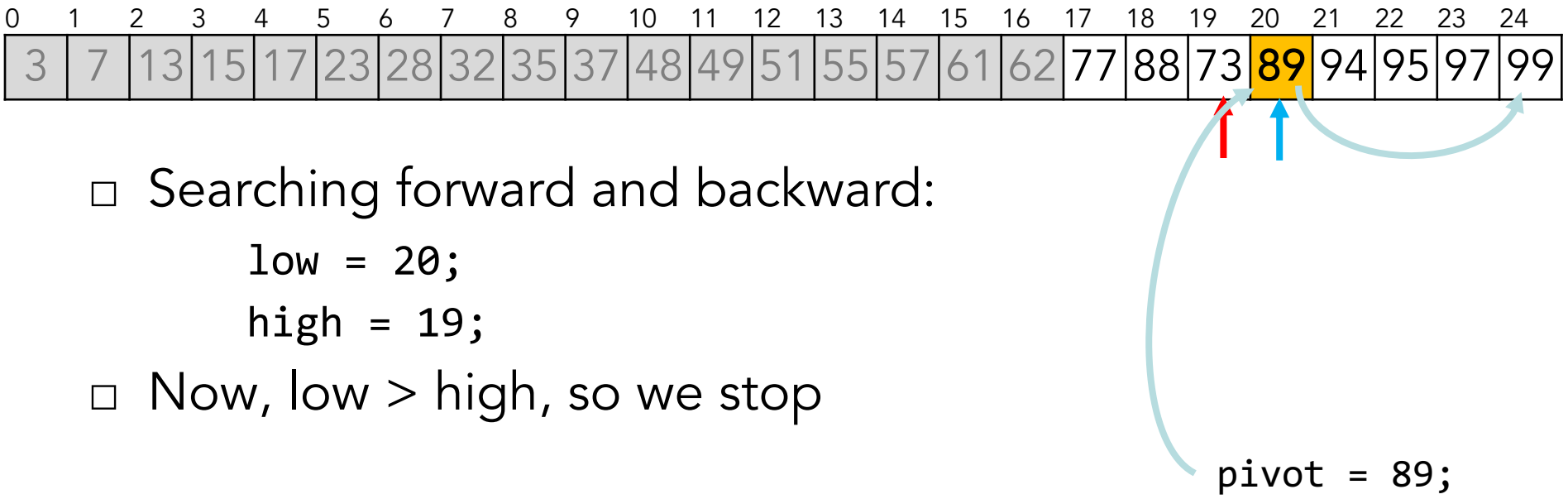
- Now, low > high, so we stop

```
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:

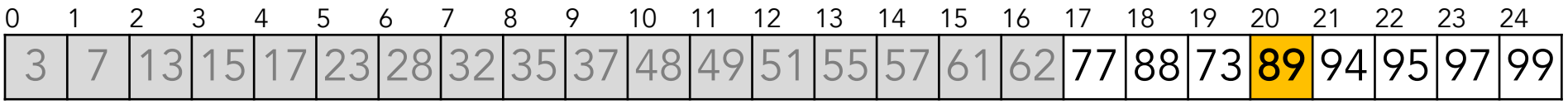
```
low = 20;
high = 19;
```

- 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 )`



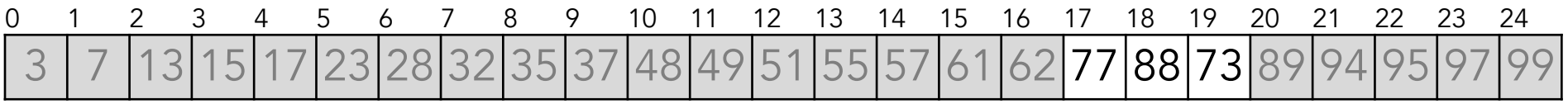
- We start by calling quicksort recursively on the first half

```
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 )`

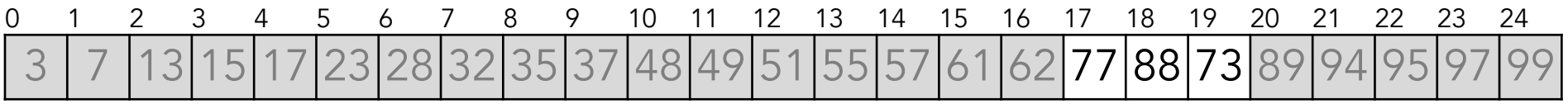


- Now, 4 – 0 ≤ 6, so find we call insertion sort

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

Quicksort Example

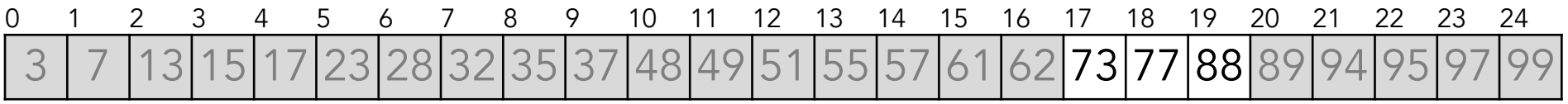
- Insertion sort just sorts the entries from 17 to 19




```
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

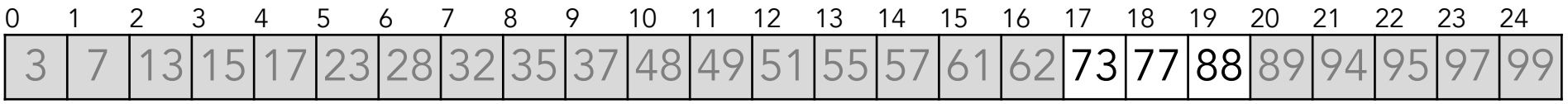


  - This function call completes and so we exit

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

Quicksort Example

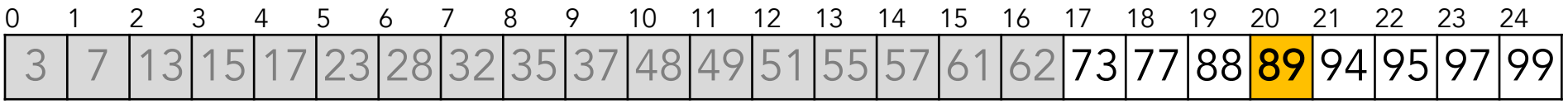
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

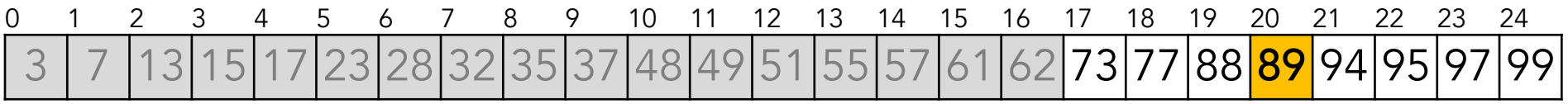
- We are back to executing `quicksort( array, 17, 25 )`




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

Quicksort Example

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



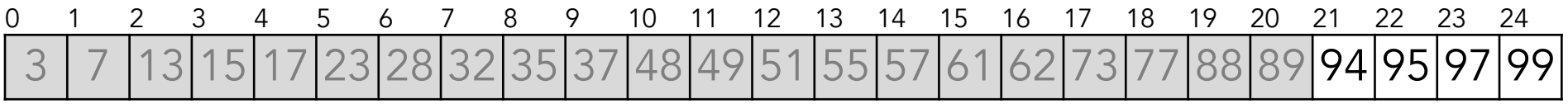
- We continue by calling quicksort on the second half

```
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 )`

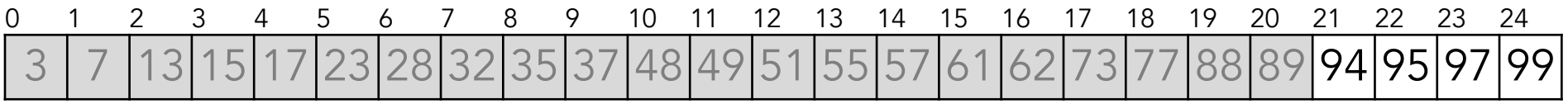


- Now, 25 – 21 ≤ 6, so find we call insertion sort

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

Quicksort Example

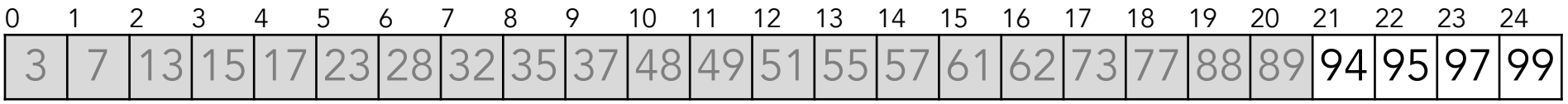
- Insertion sort just sorts the entries from 21 to 24




```
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

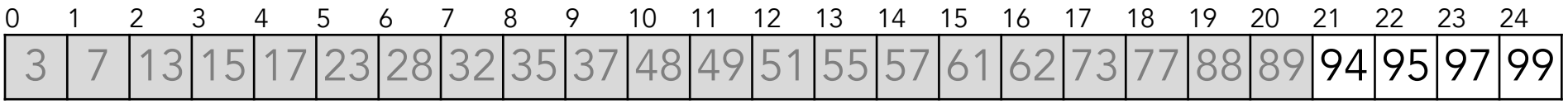


  - In this case, the sub-array was already sorted
  - This function call completes and so we exit

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

Quicksort Example

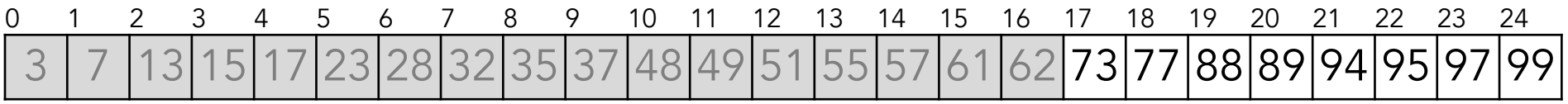
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

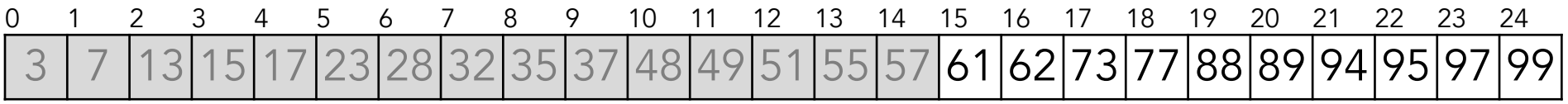
- This call to quicksort is now also finished, so it, too, exits




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

Quicksort Example

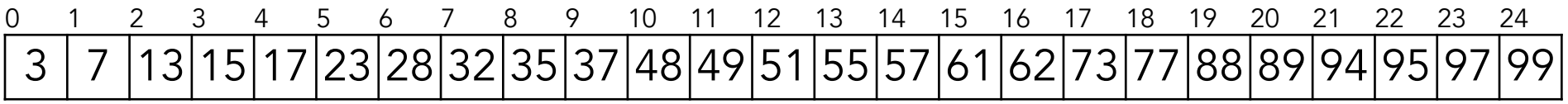
- This call to quicksort is now also finished, so it, too, exits




```
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

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
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Runtime Analysis

- The runtime of quicksort T(n) is the addition of
  - Runtime of the two recursive calls
  - The linear time spent in the partition (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:

$$T(n) = \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn$$

- Solving this is a bit more tricky than worst/best cases
  - Check our textbook, Weiss 7.7.5 (p318)
- In the end you will get

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

Memory Requirements

- The additional memory is required: **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 **O**(ln(*n*))
  - Worst case: The depth of the recursion tree is **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	$O(n \ln(n))$	$O(n^2)$	$O(\ln(n))$	$O(n)$

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 $O(n \ln(n))$ and $O(\ln(n))$ memory
  - Worst case run time of $O(n^2)$ and $O(n)$ memory

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Quicksort

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

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

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

[5] 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.

Bucket Sort

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- Recall:
  - All exchange (swapping) based sorting takes $\Theta(n^2)$
  - All comparison based sorting takes $\Theta(n \ln(n))$

- Bucket sort makes assumptions about the data being sorted
  - Consequently, we can achieve better than $\Theta(n \ln(n))$ run times

- We will look at:
  - a supporting example
  - the algorithm
  - run times (no best-, worst-, or average-cases)
  - summary and discussion

Supporting example

- Suppose we are sorting a large number of local phone numbers, for example, all residential phone numbers in the 02 area code region

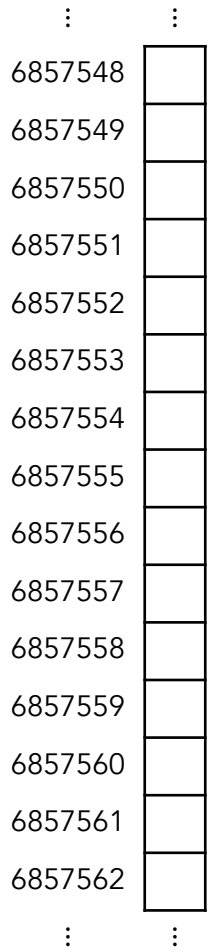
- Goal: We want something faster than heap/merge/quicksort!

Supporting example

- Consider the following scheme:
  - Create a bit vector with 10 000 000 bits
    - This requires $10^7/1024/1024/8 \approx 1.2$ MB
  - Set each bit to 0 (indicating false)
  - For each phone number, set the bit indexed by the phone number to 1 (true)
  - Once each phone number has been checked, walk through the array and for each bit which is 1, record that number

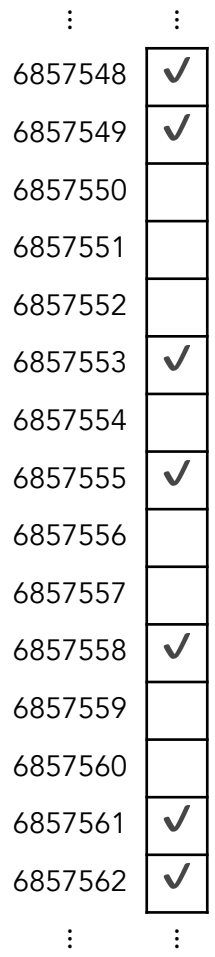
Supporting example

- For example, consider this section within the bit array



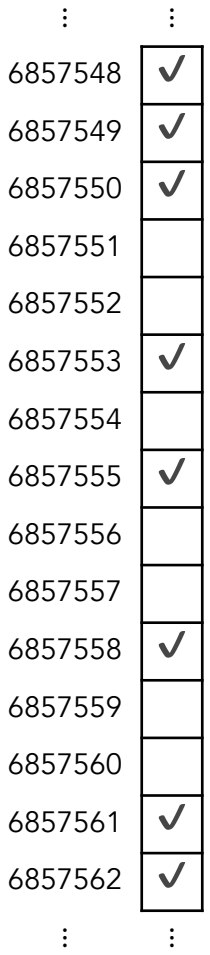
Supporting example

- For each phone number, set the corresponding bit
  - For example, 685-7550 is a residential phone number



Supporting example

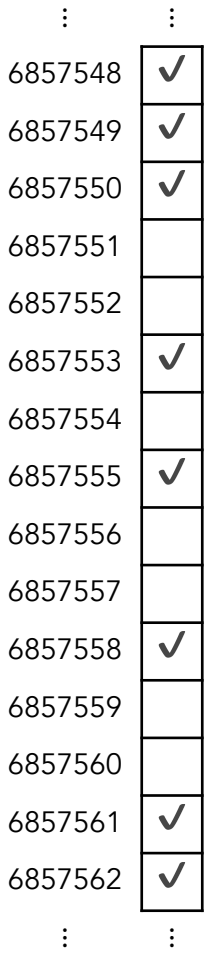
- For each phone number, set the corresponding bit
  - For example, 685-7550 is a residential phone number



Supporting example

- At the end, we just take all the numbers out that were checked:

…,685-7548, 685-7549, 685-7550,
685-7553, 685-7555, 685-7558,
685-7561, 685-5762, …



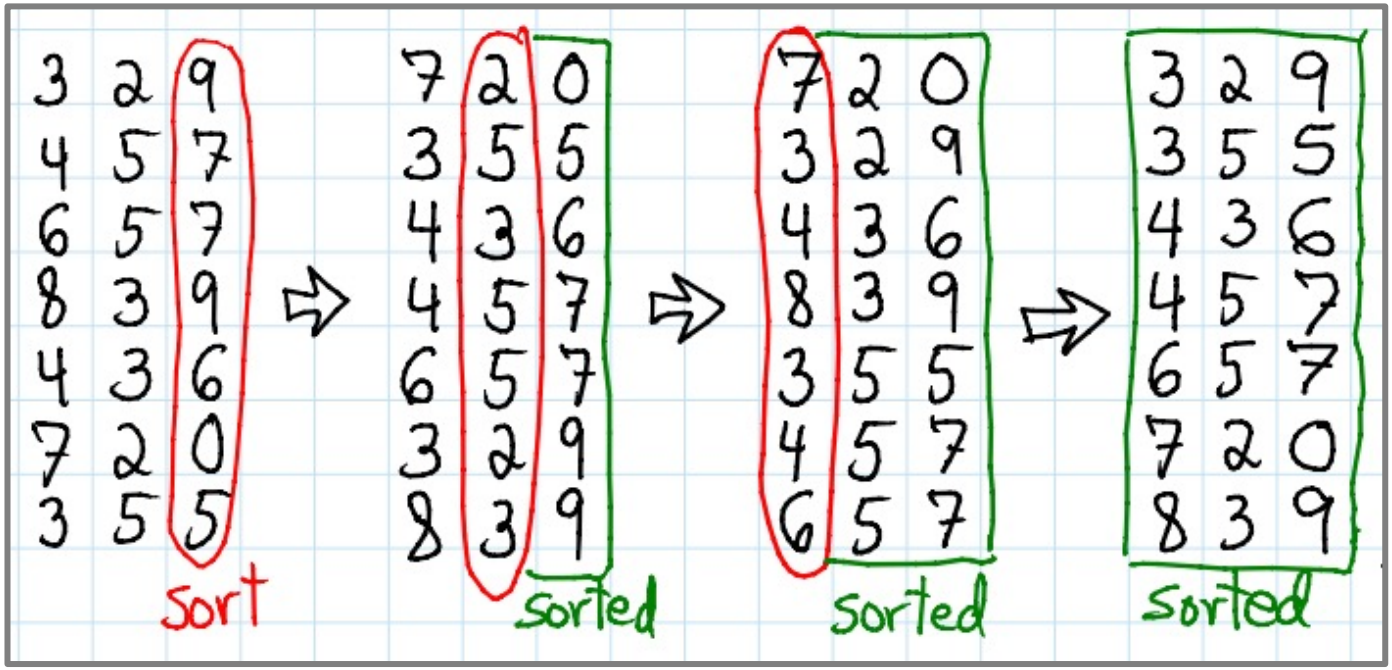
Supporting example

- In this example, the number of phone numbers (4 000 000 or n) is similar to the size of the array (10 000 000 or m)

- The runtime of such an algorithm is $\Theta(n+m)$:
  - we make one pass through the data (n),
  - we make another pass through the array and extract the phone numbers which are true (m)

Radix Sort

- **Radix sort:** Use bucket sort in a more complex arrangement
  - Keeping down the number of buckets
  - Applying bucket sorts in a row



Summary

- By assuming that the data falls into a given range, we can achieve $\Theta(n)$ sorting run times

- As any sorting algorithm must access any object at least once, the run time must be $\Theta(n)$

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Bucket_sort

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

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

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

[5] 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.