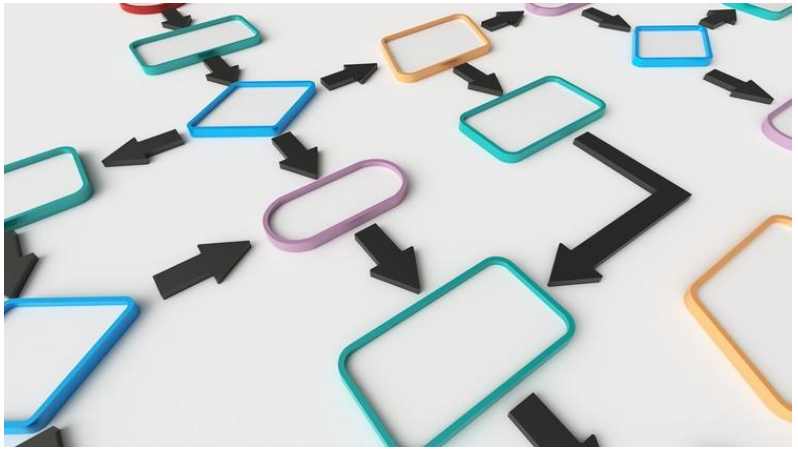
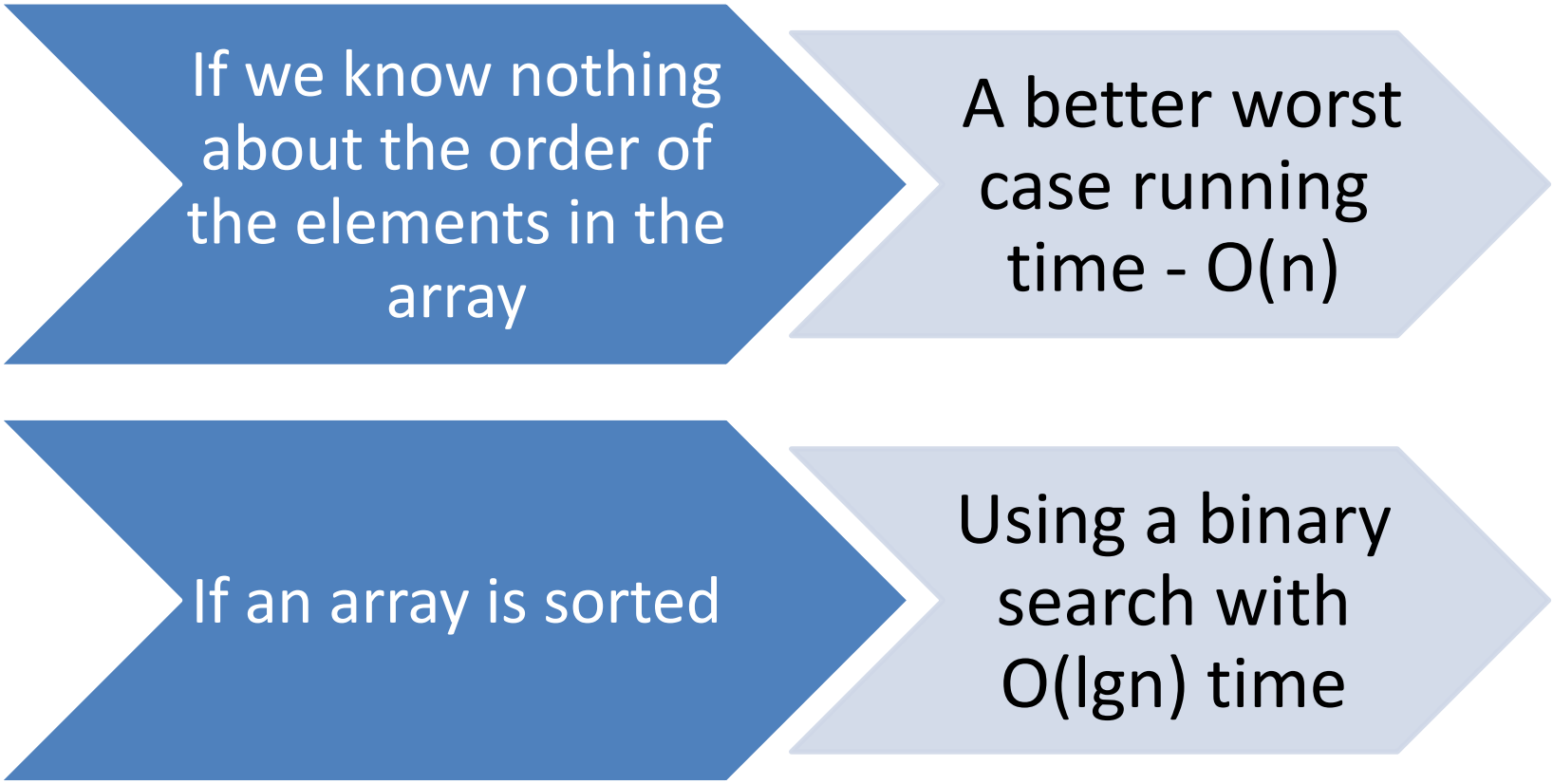
# Algorithms for Sorting and Searching

1. Binary search.
2. Selection sort.
3. Insertion sort.
4. Merge sort.
5. Quick sort.

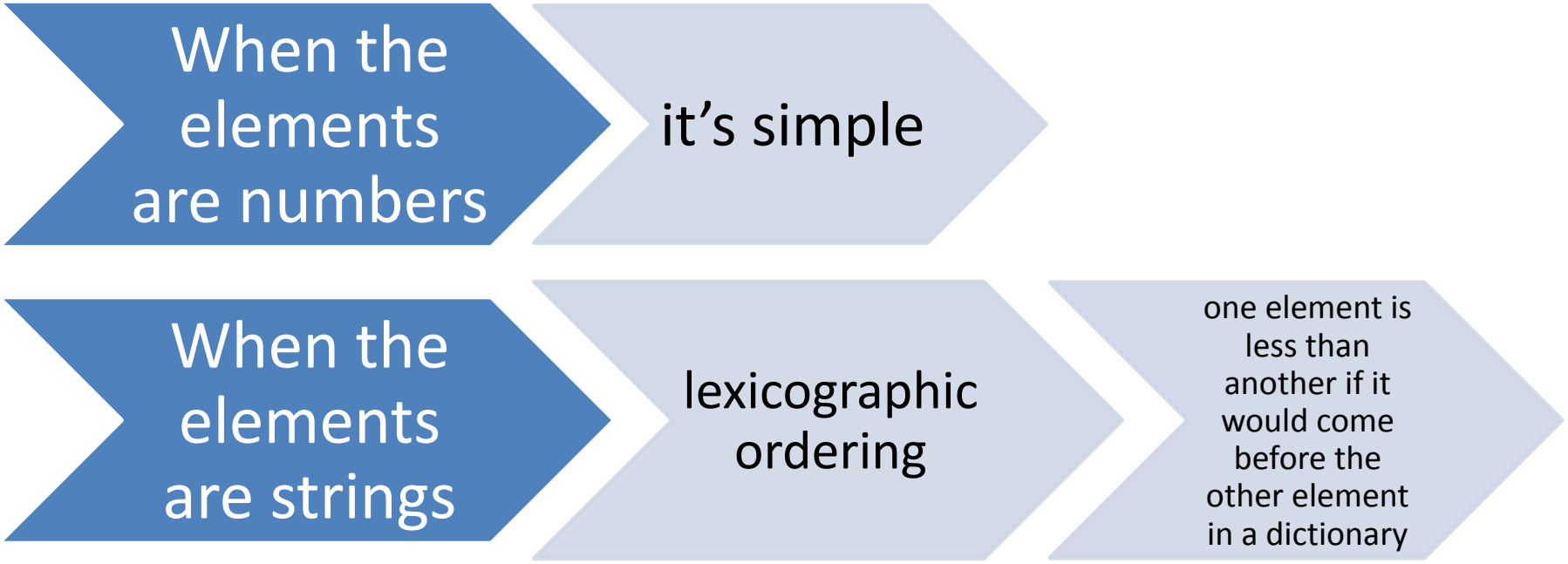


# Binary search



# Binary search

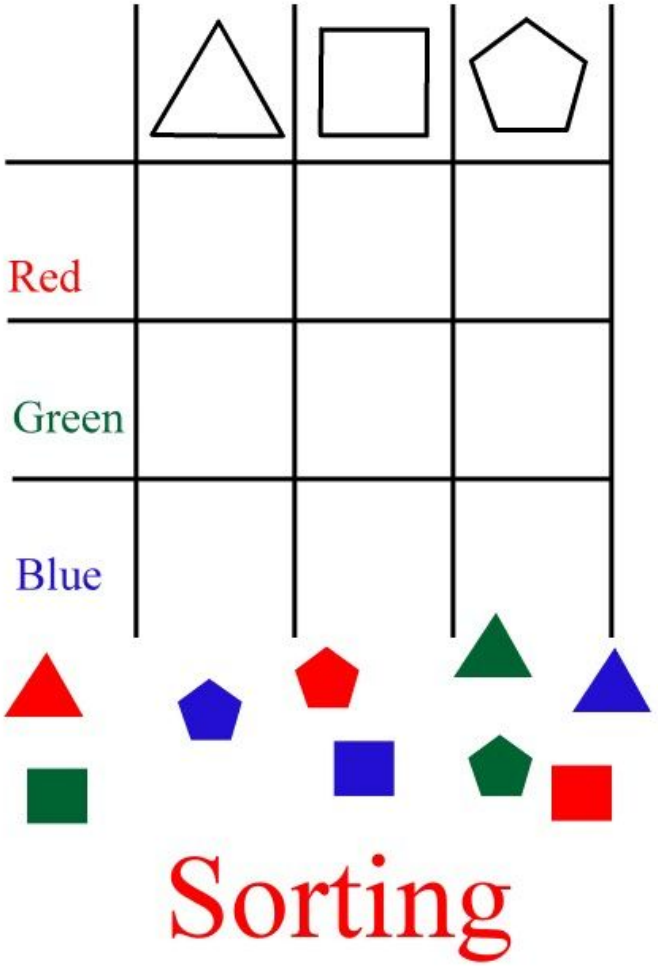
What does it mean for one element to be less than another?



# Binary search

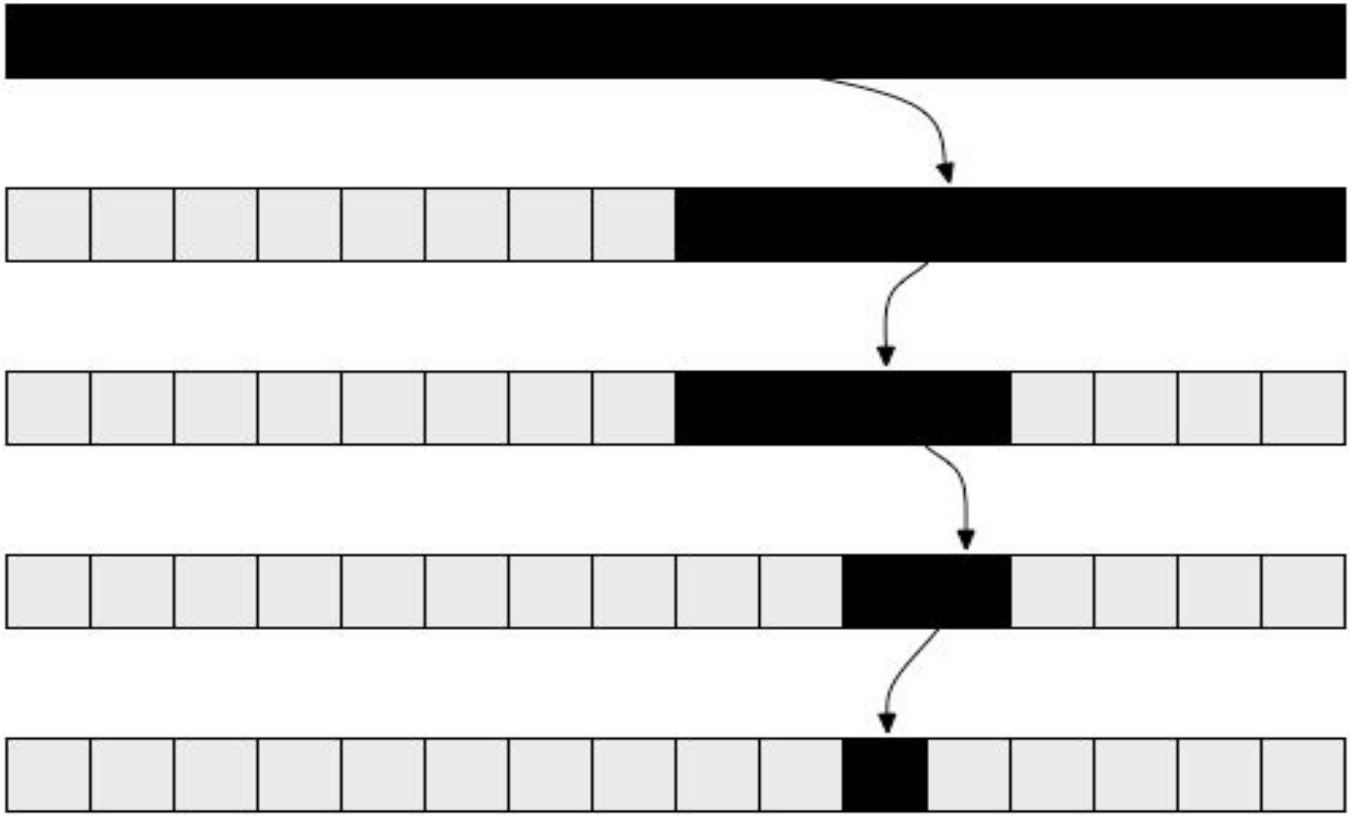
What does mean sorting?

Sorting means: to put into some well-defined order.



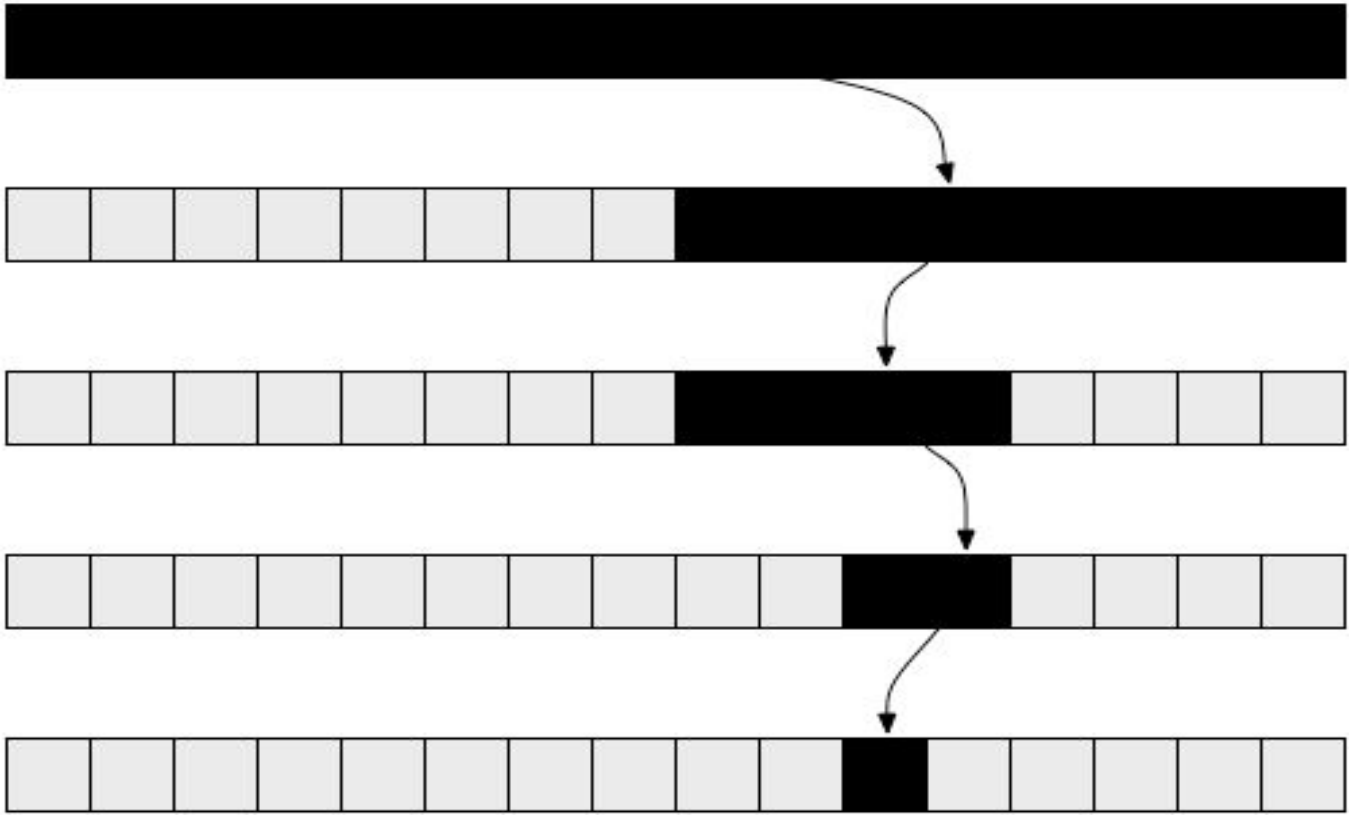
# Binary search

Binary search  requires the array being searched to be already sorted.



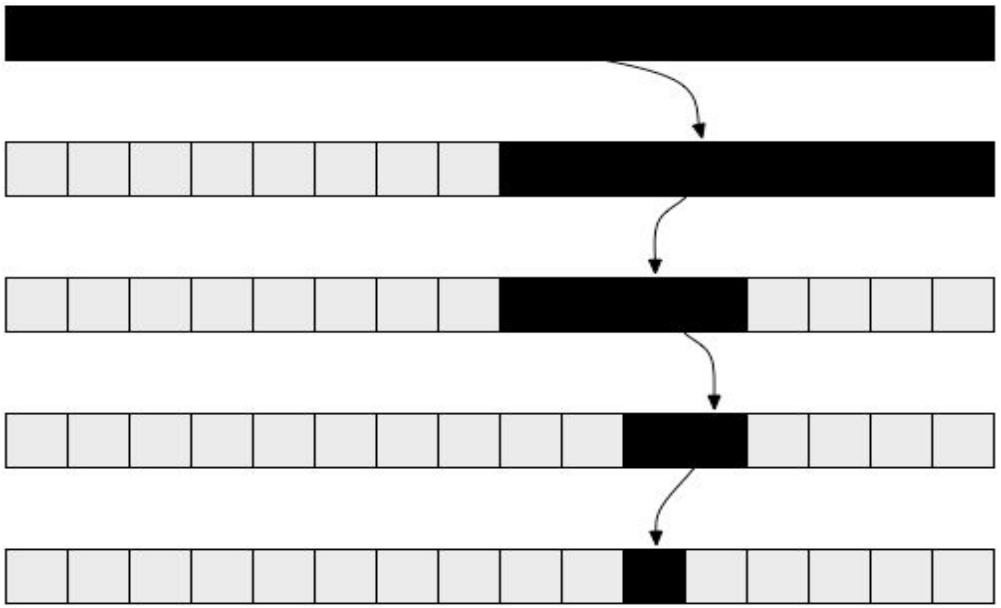
# Binary search

Binary search has the advantage that it takes only O(lgn)  time to search an n-element array.



# Binary search

The books on the bookshelf already sorted by author name.
The position of the book is named a slot.
The key is the author name.



# Binary search

*Procedure* BINARY-SEARCH$(A, n, x)$

*Inputs and Output:* Same as LINEAR-SEARCH.

1. Set $p$ to 1, and set $r$ to $n$.
2. While $p \leq r$, do the following:
   A. Set $q$ to $\lfloor (p + r)/2 \rfloor$.
   B. If $A[q] = x$, then return $q$.
   C. Otherwise ($A[q] \neq x$), if $A[q] > x$, then set $r$ to $q - 1$.
   D. Otherwise ($A[q] < x$), set $p$ to $q + 1$.
3. Return NOT-FOUND.

# Binary search

*Procedure* RECURSIVE-BINARY-SEARCH$(A, p, r, x)$

*Inputs and Output:* Inputs $A$ and $x$ are the same as LINEAR-SEARCH, as is the output. The inputs $p$ and $r$ delineate the subarray $A[p\,..\,r]$ under consideration.

1. If $p > r$, then return NOT-FOUND.
2. Otherwise ($p \leq r$), do the following:
   A. Set $q$ to $\lfloor (p + r)/2 \rfloor$.
   B. If $A[q] = x$, then return $q$.
   C. Otherwise ($A[q] \neq x$), if $A[q] > x$, then return RECURSIVE-BINARY-SEARCH$(A, p, q - 1, x)$.
   D. Otherwise ($A[q] < x$), return RECURSIVE-BINARY-SEARCH$(A, q + 1, r, x)$.

# Binary search

The running time of binary search is O(lgn).

! The array is already sorted.

# Selection sort

Remind: Each element is less than or equals to its following element in the array.

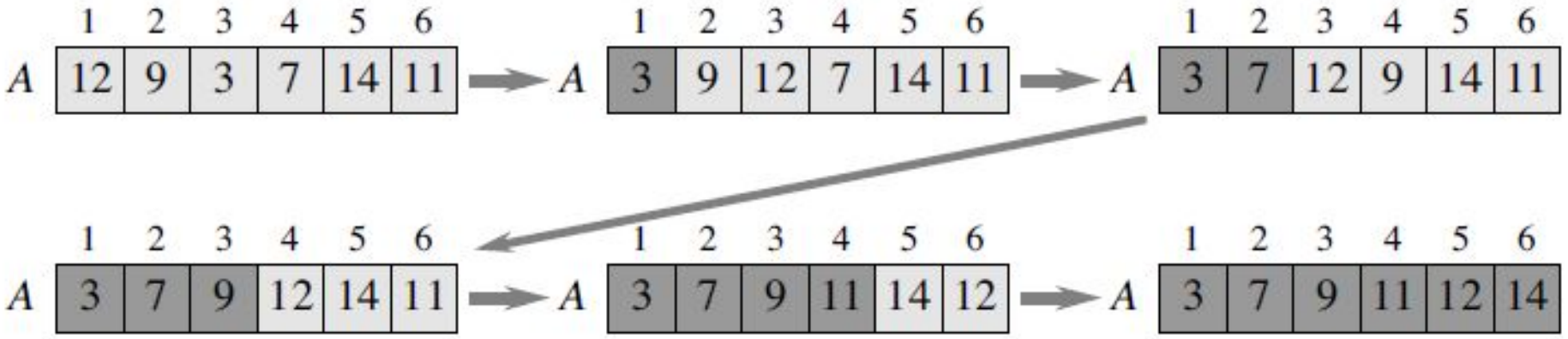
*Procedure* SELECTION-SORT$(A, n)$

*Inputs and Result:* Same as before.

1. For $i = 1$ to $n - 1$:
   A. Set *smallest* to $i$.
   B. For $j = i + 1$ to $n$:
      i. If $A[j] < A[smallest]$, then set *smallest* to $j$.
   C. Swap $A[i]$ with $A[smallest]$.

# Selection sort

Selection sort => on an array of six elements



# Selection sort

What is the running time of SELECTION-SORT?

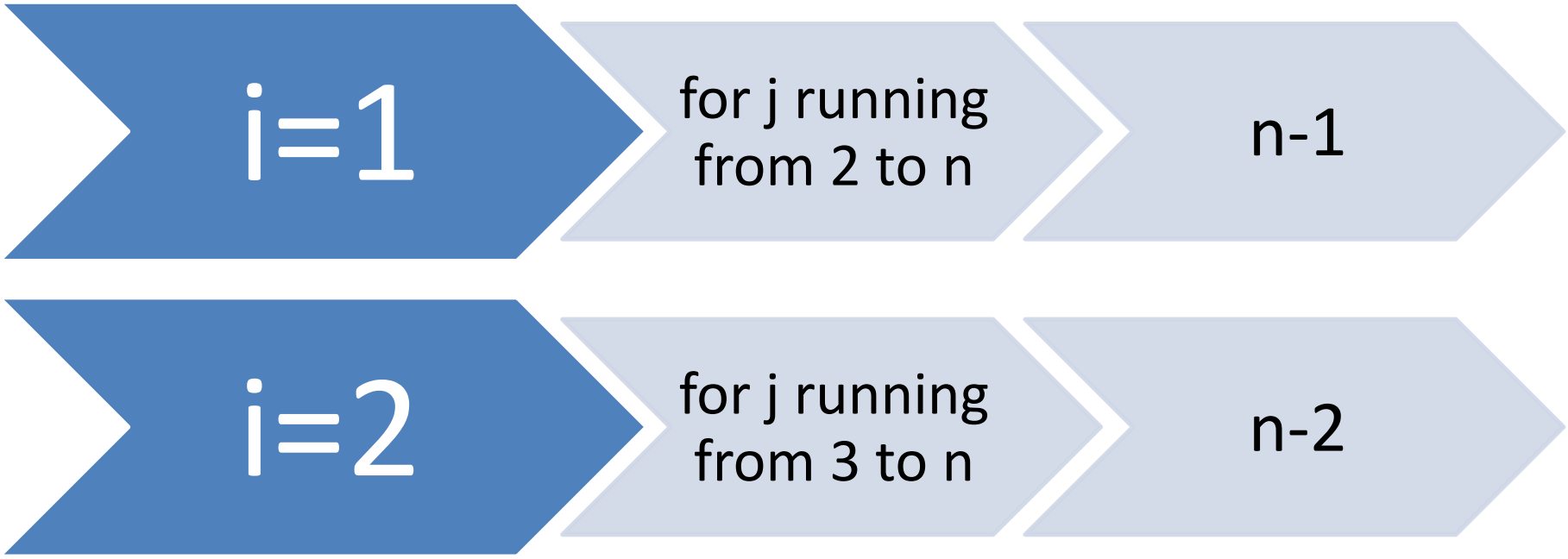
How many iterations the inner loop makes?
Remember: each iteration takes $\Theta(1)$ time.

*Procedure* SELECTION-SORT$(A, n)$

*Inputs and Result:* Same as before.

1. For $i = 1$ to $n - 1$:
   A. Set *smallest* to $i$.
   B. For $j = i + 1$ to $n$:
      i. If $A[j] < A[smallest]$, then set *smallest* to $j$.
   C. Swap $A[i]$ with $A[smallest]$.

# Selection sort




The total number of inner-loop iterations is

$(n-1)+(n-2)+(n-3)+\ldots+2+1$

# Selection sort

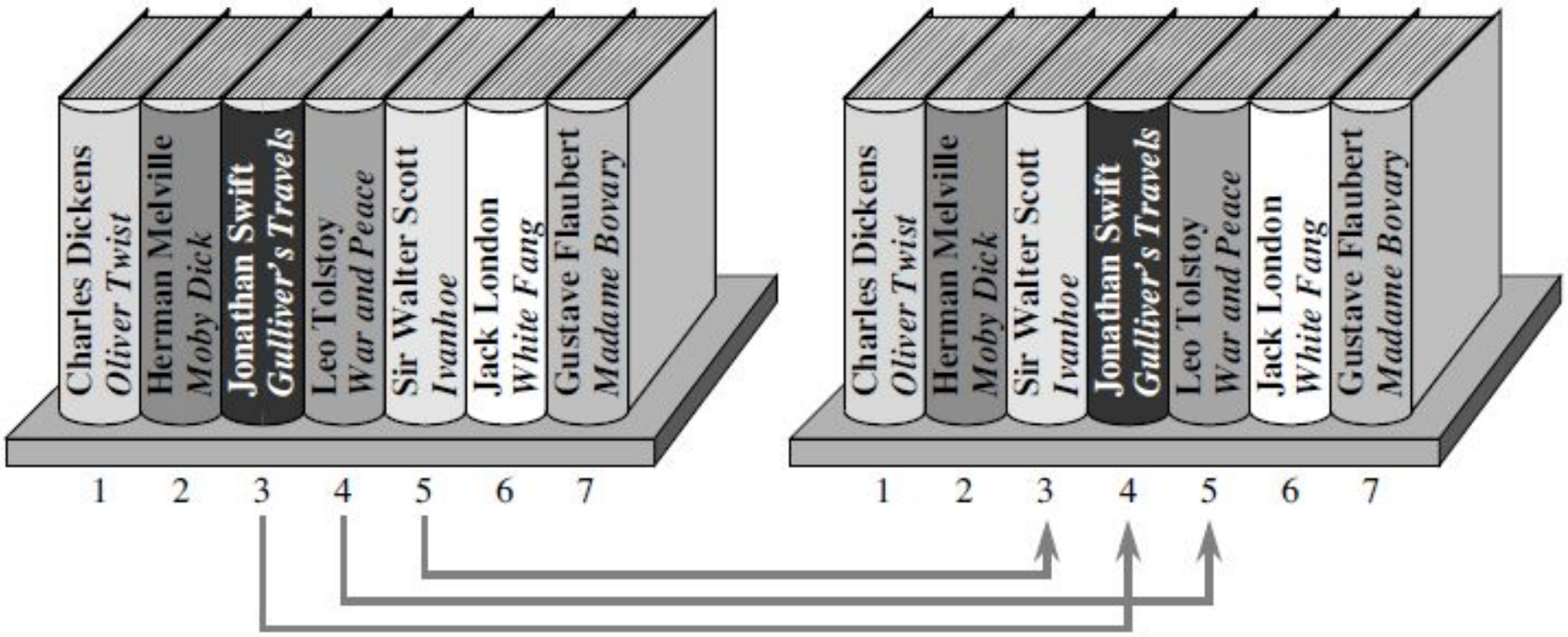
$$n+(n-1)+(n-2)+(n-3)+\ldots+2+1=$$
$$=n(n+1)/2$$

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=$$
$$=n(n-1)/2=(n^2-n)/2$$

It is sum of ***arithmetic series.***

Therefore, the running time of SELECTION-SORT is $\Theta(n^2)$.

# Insertion sort



# Insertion sort

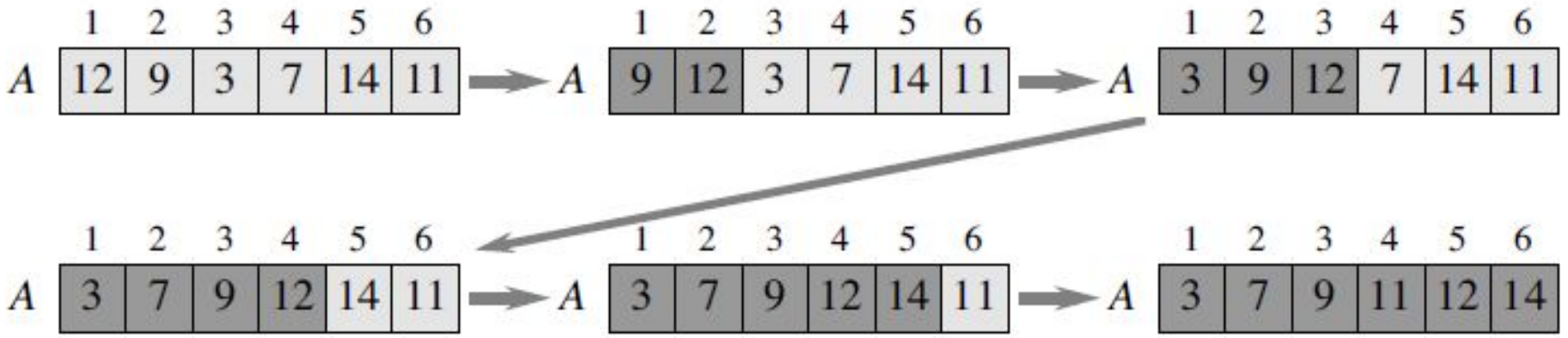
*Procedure* INSERTION-SORT$(A, n)$

*Inputs and Result:* Same as SELECTION-SORT.

1. For $i = 2$ to $n$:
   A. Set *key* to $A[i]$, and set $j$ to $i - 1$.
   B. While $j > 0$ and $A[j] > key$, do the following:
      i. Set $A[j + 1]$ to $A[j]$.
      ii. Decrement $j$ (i.e., set $j$ to $j - 1$).
   C. Set $A[j + 1]$ to *key*.

# Insertion sort

Insertion sort => on an array of six elements



# Insertion sort

What do we say about the running time of INSERTION-SORT?

| | | |
|---|---|---|
| The best case | when the inner loop makes zero iterations every time | $\Theta(n)$ |
| The worst case | when the inner loop makes the maximum possible number of iterations every time | $\Theta(n^2)$ |

# Merge sort

The running times of selection sort and insertion sort are $\Theta(n^2)$.

The running time of merge sort is $\Theta(n \lg n)$.

$\Rightarrow$ $\Theta(n \lg n)$ better because $\lg n$ grows more slowly than $n$.

# Merge sort

Disadvantages:

1. The constant factor in the asymptotic notation is higher than for the other two algorithms.

⇒ If the array size n is not large, merge sort isn't used.

2. Merge sort has to make complete copies of all input array.

⇒ If the computer memory is important, merge sort isn't used also.

# Merge sort

## Divide-and-conquer algorithm

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.
**Conquer.** The subproblems solve recursively. If they are small, than the subproblems solve as base cases.
**Combine** the solutions to the subproblems into the solution for the original problem.

# Merge sort

**Divide-and-conquer algorithm for example with bookshelf**

**Divide** all index (slot) of books in two part. The center of index's books is q equals (p + r)/2.
**Conquer**. We recursively sort the books in each of the two subproblems: [p;q] and [q+1;r].
**Combine** by merging the sorted books.

# Merge sort

*Procedure* MERGE-SORT$(A, p, r)$

*Inputs:*

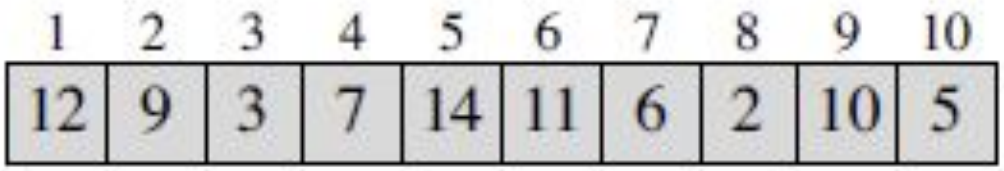
- $A$: an array.
- $p, r$: starting and ending indices of a subarray of $A$.

*Result:* The elements of the subarray $A[p\,..\,r]$ are sorted into nondecreasing order.

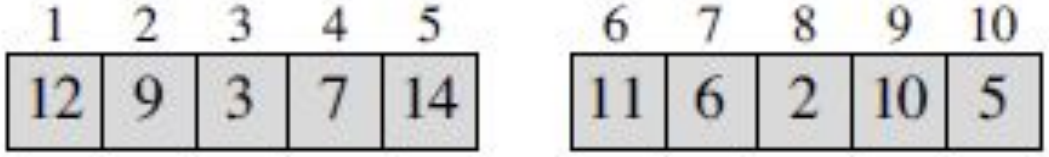
1. If $p \geq r$, then the subarray $A[p\,..\,r]$ has at most one element, and so it is already sorted. Just return without doing anything.
2. Otherwise, do the following:
   A. Set $q$ to $\lfloor (p + r)/2 \rfloor$.
   B. Recursively call MERGE-SORT$(A, p, q)$.
   C. Recursively call MERGE-SORT$(A, q + 1, r)$.
   D. Call MERGE$(A, p, q, r)$.

# Merge sort

The initial call is MERGE-SORT (A, 1, 10).




Step 2A computes q to be 5,
in steps 2B and 2C are MERGE-SORT (A, 1, 5)
and MERGE-SORT (A, 6, 10).

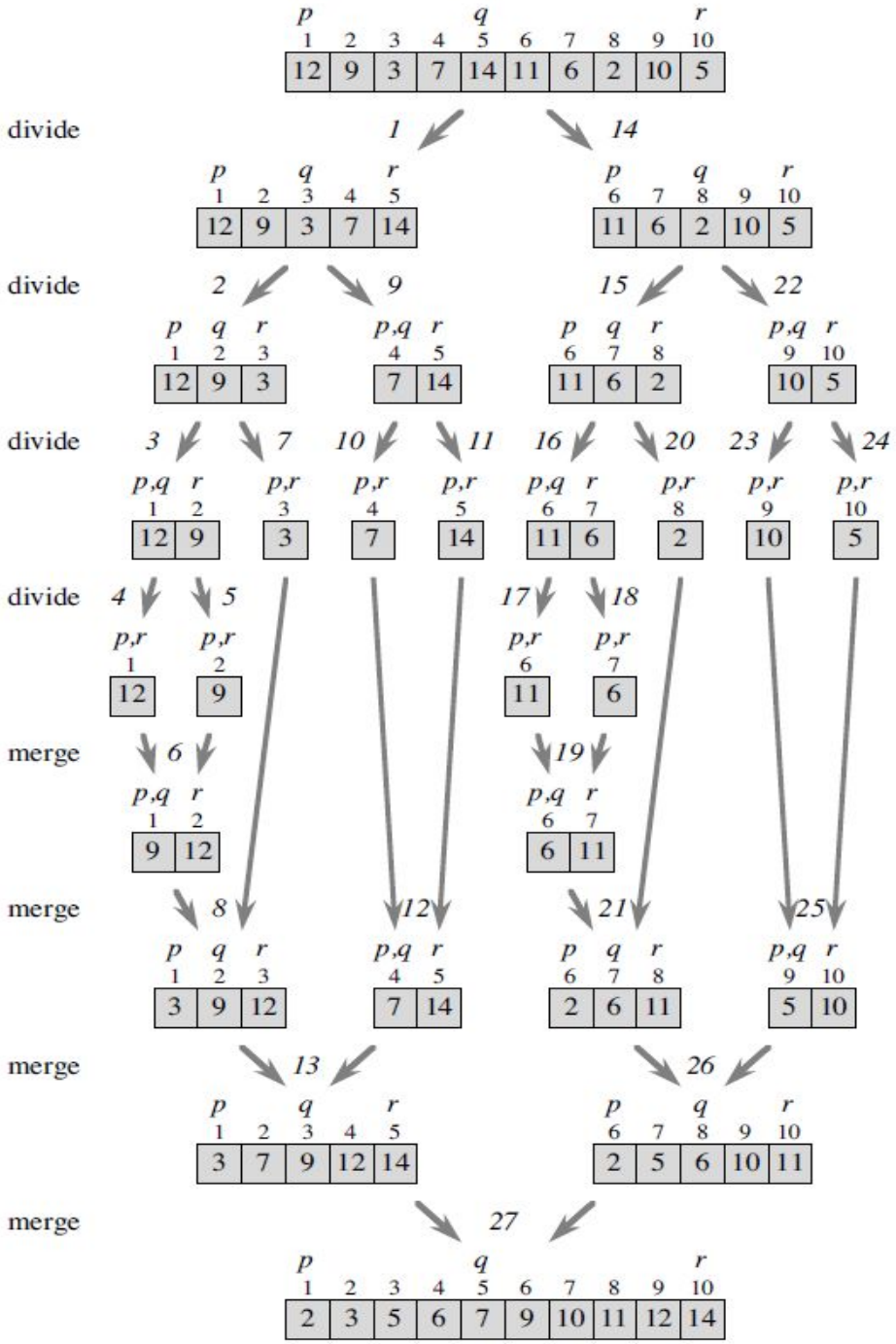



After the two recursive calls return, these two subarrays are sorted.

# Merge sort

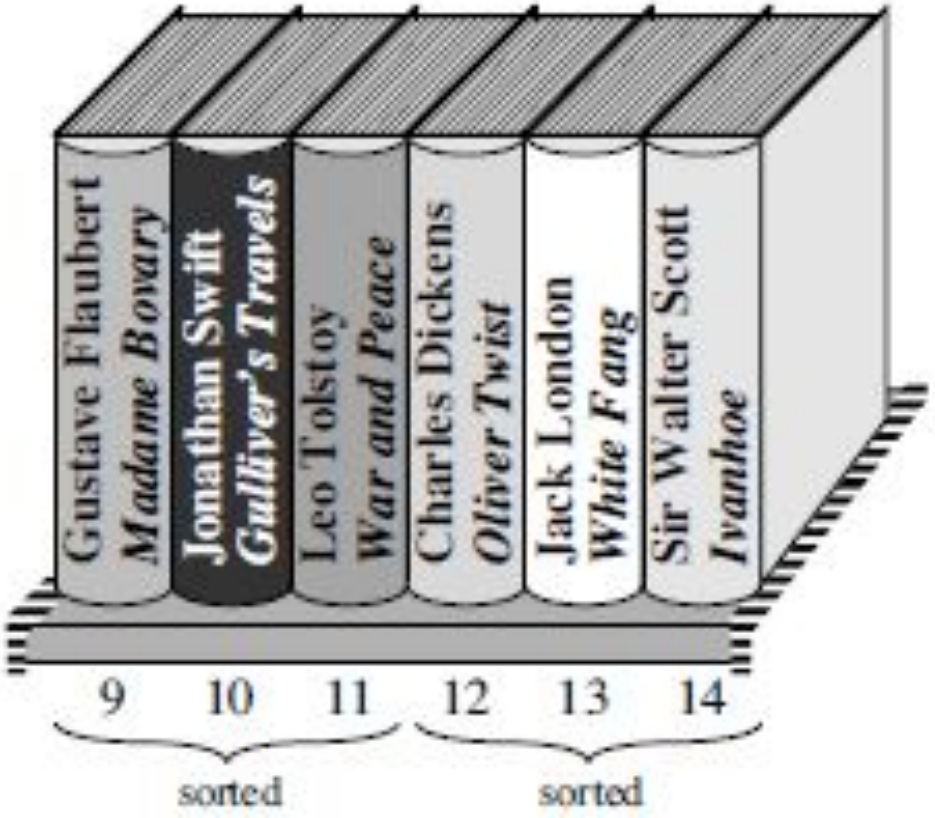
At last, the call MERGE (A, 1, 5, 10) in step 2D


The procedure MERGE is used to merge the sorted subarrays into the single sorted subarray.


p q r
1 2 3 4 5 6 7 8 9 10
12 9 3 7 14 11 6 2 10 5
divide
1
14
p q r
1 2 3 4 5
12 9 3 7 14
p q r
6 7 8 9 10
11 6 2 10 5
divide
2
9
15
22
p q r
1 2 3
12 9 3
p,q r
4 5
7 14
p q r
6 7 8
11 6 2
p,q r
9 10
10 5
divide
3
7
10
11
16
20
23
24
p,q r
1 2
12 9
p,r
3
p,r
4
7
p,r
5
14
p,q r
6 7
11 6
p,r
8
2
p,r
9
10
p,r
10
5
divide
4
5
17
18
p,r
1
12
p,r
2
9
p,r
6
11
p,r
7
6
merge
6
19
p,q r
1 2
9 12
p,q r
6 7
6 11
merge
8
12
21
25
p q r
1 2 3
3 9 12
p,q r
4 5
7 14
p q r
6 7 8
2 6 11
p,q r
9 10
5 10
merge
13
26
p q r
1 2 3 4 5
3 7 9 12 14
p q r
6 7 8 9 10
2 5 6 10 11
merge
27
p q r
1 2 3 4 5 6 7 8 9 10
2 3 5 6 7 9 10 11 12 14

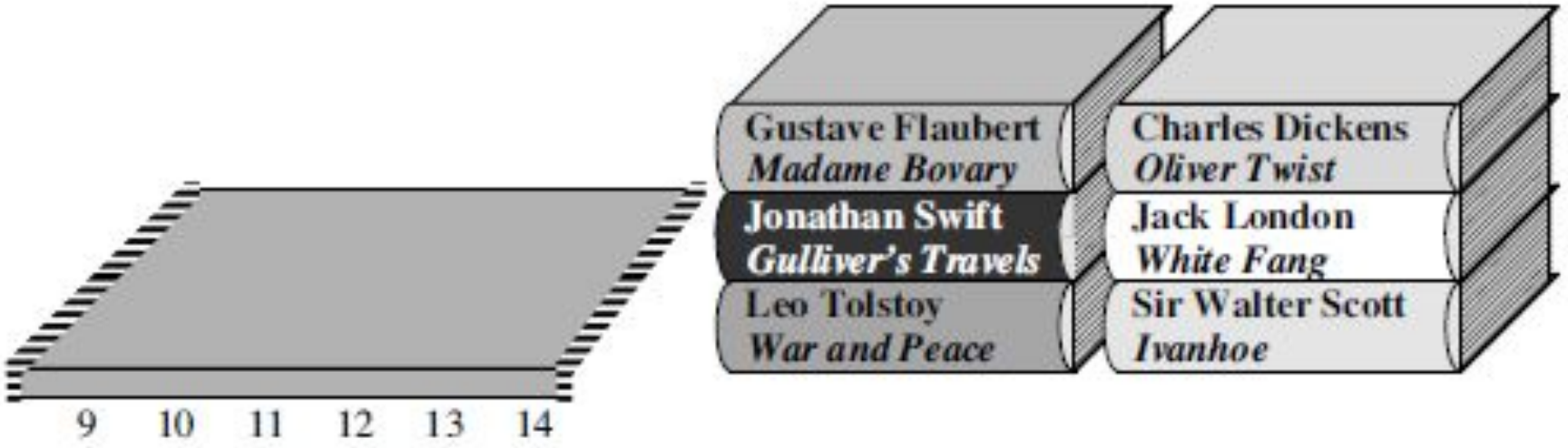
# Merge sort

Let’s look at just the part of the bookshelf from slot 9 through slot 14. We have sorted the books in slots 9–11 and that we have sorted the books in slots 12–14.



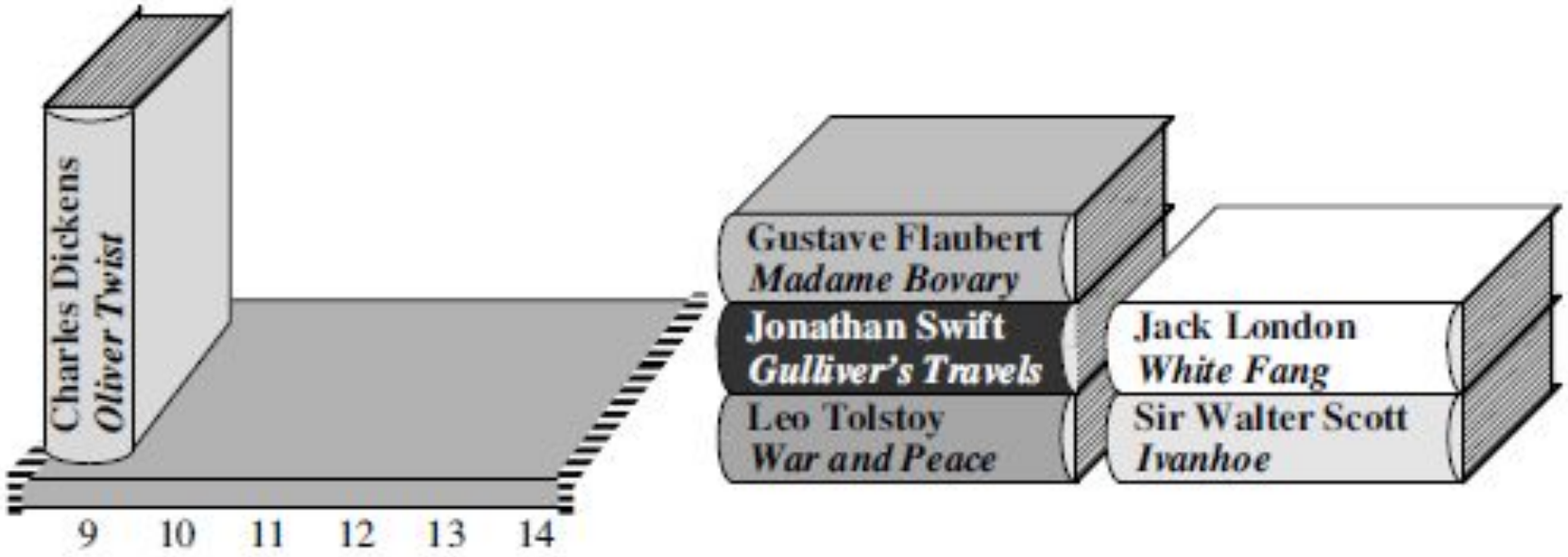
# Merge sort

We take out the books in slots 9–11 and make a stack of them, with the book whose author is alphabetically first on top, and we do the same with the books in slots 12–14.



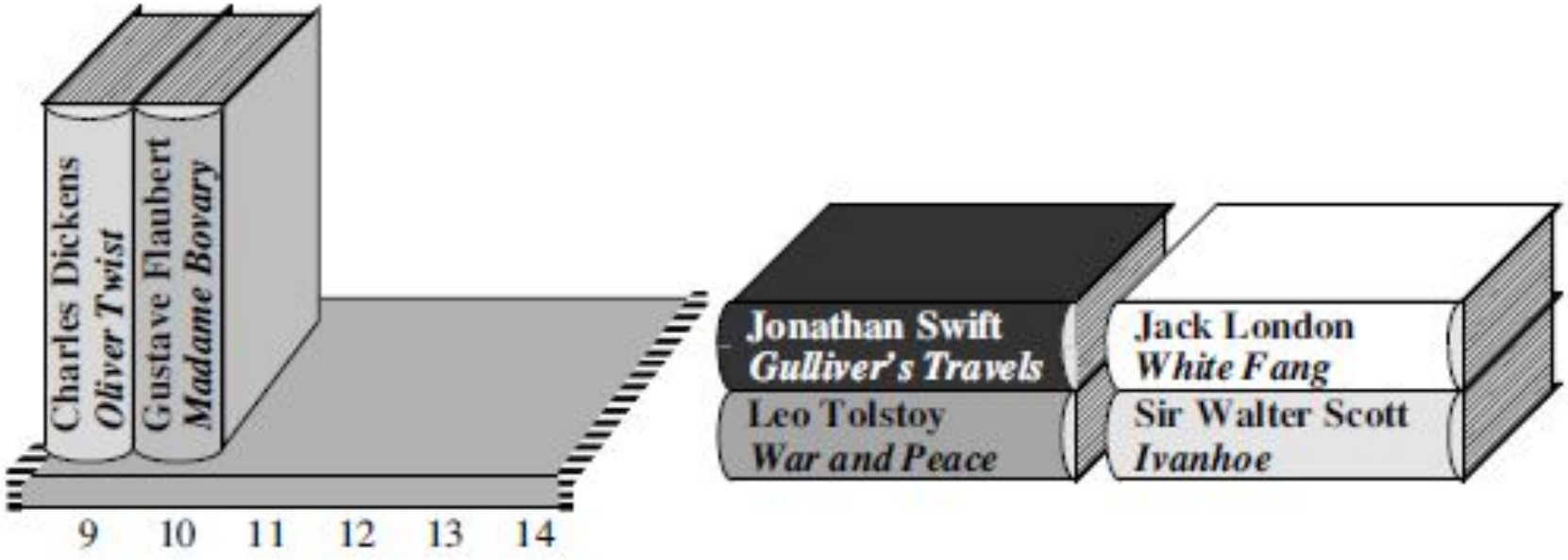
# Merge sort

Because the two stacks are already sorted, the book that should go back into slot 9 must be one of the books atop its stack.
We see that the book by Dickens comes before the book by Flaubert, and so we move it into slot 9.

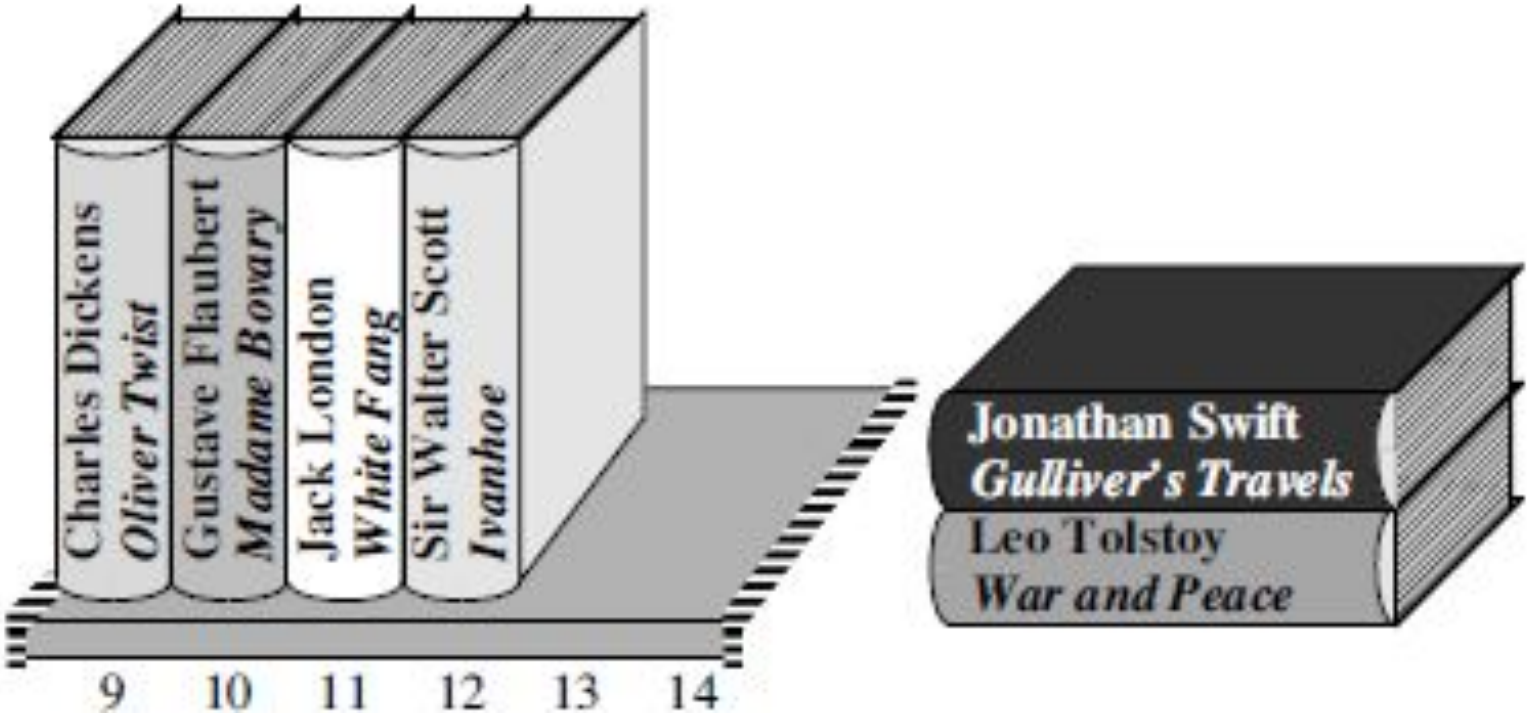


# Merge sort

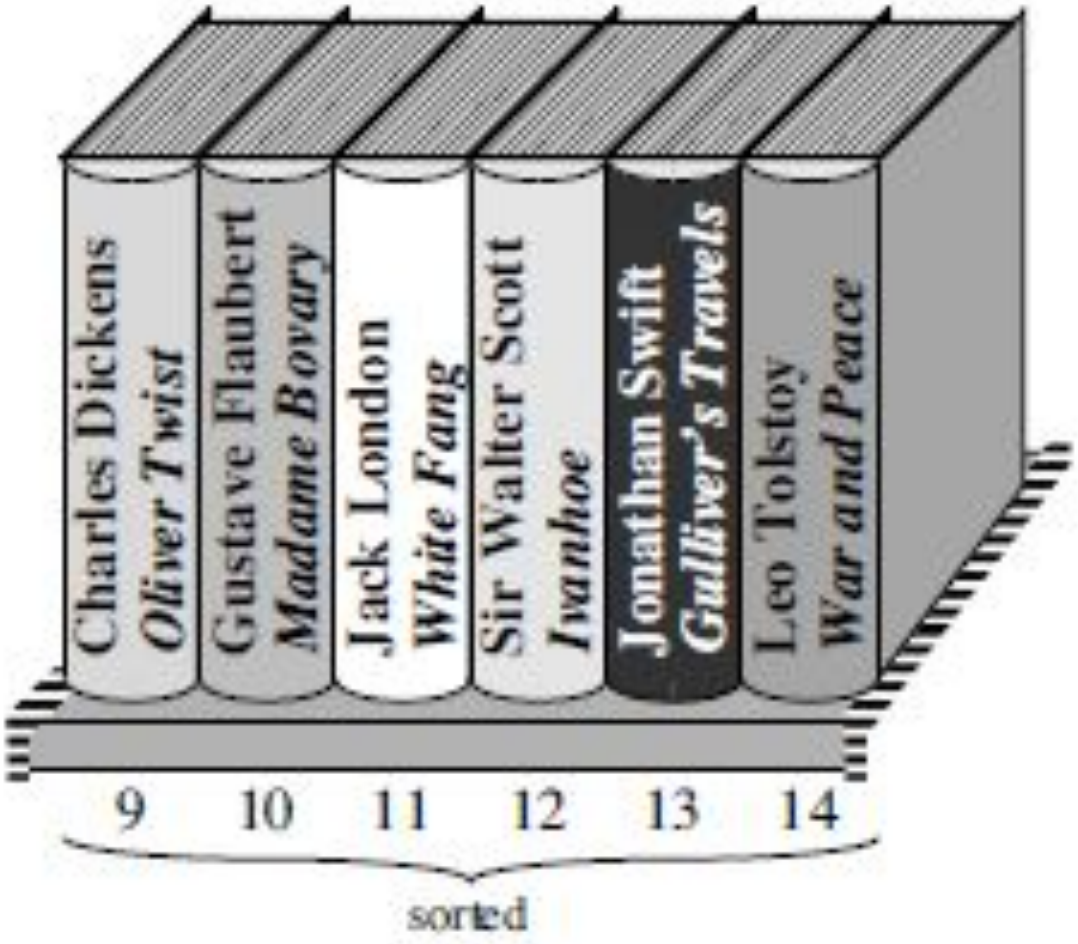
Into slot 10 must be the book still atop the first stack, by Flaubert, or the book now atop the second stack, by Jack London. We move the Flaubert book into slot 10.



# Merge sort


Charles Dickens
Oliver Twist
Gustave Flaubert
Madame Bovary
Jack London
White Fang
Sir Walter Scott
Ivanhoe
Jonathan Swift
Gulliver's Travels
Leo Tolstoy
War and Peace
9
10
11
12
13
14

# Merge sort


Charles Dickens
Oliver Twist
Gustave Flaubert
Madame Bovary
Jack London
White Fang
Sir Walter Scott
Ivanhoe
Jonathan Swift
Gulliver's Travels
Leo Tolstoy
War and Peace
9
10
11
12
13
14
sorted

*Procedure* MERGE$(A, p, q, r)$

*Inputs:*

- $A$: an array.
- $p, q, r$: indices into $A$. Each of the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ is assumed to be already sorted.

*Result:* The subarray $A[p\,..\,r]$ contains the elements originally in $A[p\,..\,q]$ and $A[q+1\,..\,r]$, but now the entire subarray $A[p\,..\,r]$ is sorted.

1. Set $n_1$ to $q - p + 1$, and set $n_2$ to $r - q$.
2. Let $B[1\,..\,n_1 + 1]$ and $C[1\,..\,n_2 + 1]$ be new arrays.
3. Copy $A[p\,..\,q]$ into $B[1\,..\,n_1]$, and copy $A[q+1\,..\,r]$ into $C[1\,..\,n_2]$.
4. Set both $B[n_1 + 1]$ and $C[n_2 + 1]$ to $\infty$.
5. Set both $i$ and $j$ to 1.
6. For $k = p$ to $r$:

   A. If $B[i] \le C[j]$, then set $A[k]$ to $B[i]$ and increment $i$.
   B. Otherwise ($B[i] > C[j]$), set $A[k]$ to $C[j]$ and increment $j$.

# Merge sort

Let's say that sorting a subarray of n elements takes time T(n). The time T(n) comes from the three components of the divide-and-conquer algorithm:

- Dividing takes constant time, because it amounts to just computing the index q.
- Conquering consists of the two recursive calls on subarrays, each with n/2 elements. It is time T(n/2).
- Combining the results of the two recursive calls by merging the sorted subarrays takes $\Theta(n)$ time.

$$\mathbf{T(n)=T(n/2)+T(n/2)+\Theta(n)=2T(n/2)+\Theta(n) \quad => \quad T(n)= \Theta(nlgn)}$$

# Quick sort

Quicksort uses the divide-and-conquer paradigm and uses recursion.

There are some differences from merge sort:

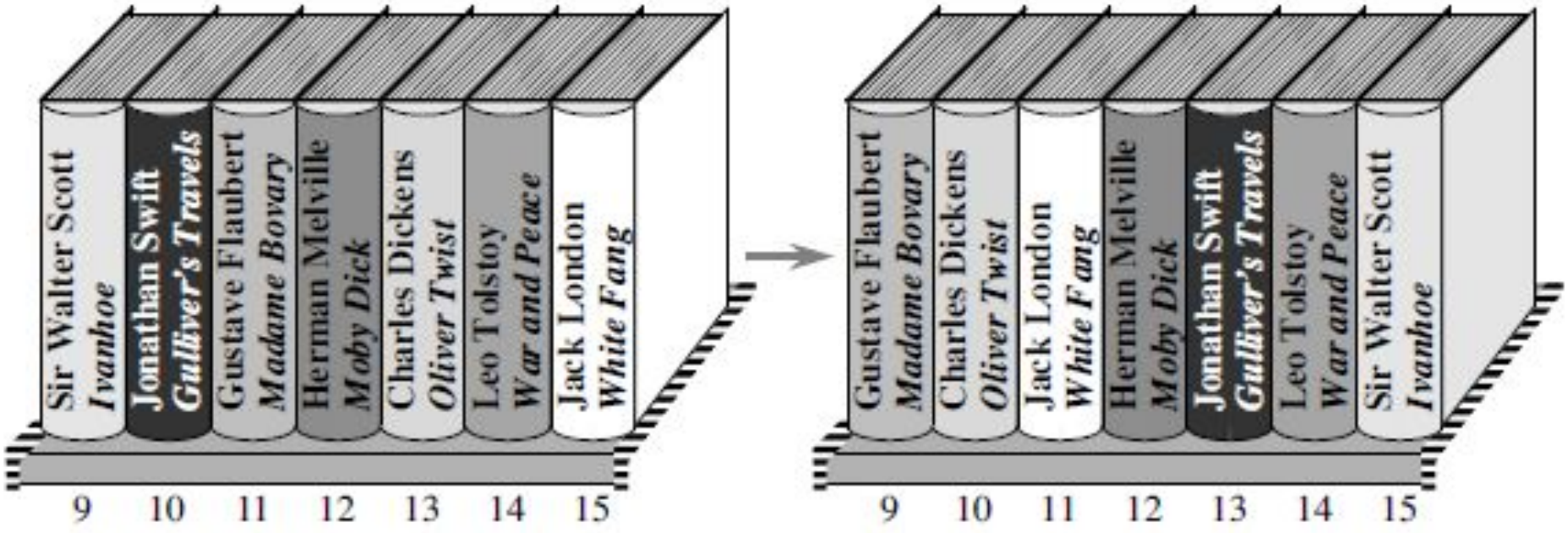
- Quicksort works in place.
- Quicksort's worst-case running time is $\Theta(n^2)$ but its average-case running time is better: $\Theta$ (nlg n).

Quicksort is often a good sorting algorithm to use in practice.

# Quick sort

1. **Divide** by first choosing any one book that is in slots p through r. Call this book the **pivot**.
   - Rebuild the books on the shelf so that all other books with author names that come before the pivot's author are to the left of the pivot, and all books with author names that come after the pivot's author are to the right of the pivot.
   - The books to the left of the book by London are in no particular order, and the same is true for the books to the right.
2. **Conquer** by recursively sorting the books to the left of the pivot and to the right of the pivot.
3. **Combine** – by doing nothing!

# Quick sort


Sir Walter Scott
Ivanhoe
Jonathan Swift
Gulliver's Travels
Gustave Flaubert
Madame Bovary
Herman Melville
Moby Dick
Charles Dickens
Oliver Twist
Leo Tolstoy
War and Peace
Jack London
White Fang
9
10
11
12
13
14
15
Gustave Flaubert
Madame Bovary
Charles Dickens
Oliver Twist
Jack London
White Fang
Herman Melville
Moby Dick
Jonathan Swift
Gulliver's Travels
Leo Tolstoy
War and Peace
Sir Walter Scott
Ivanhoe
9
10
11
12
13
14
15

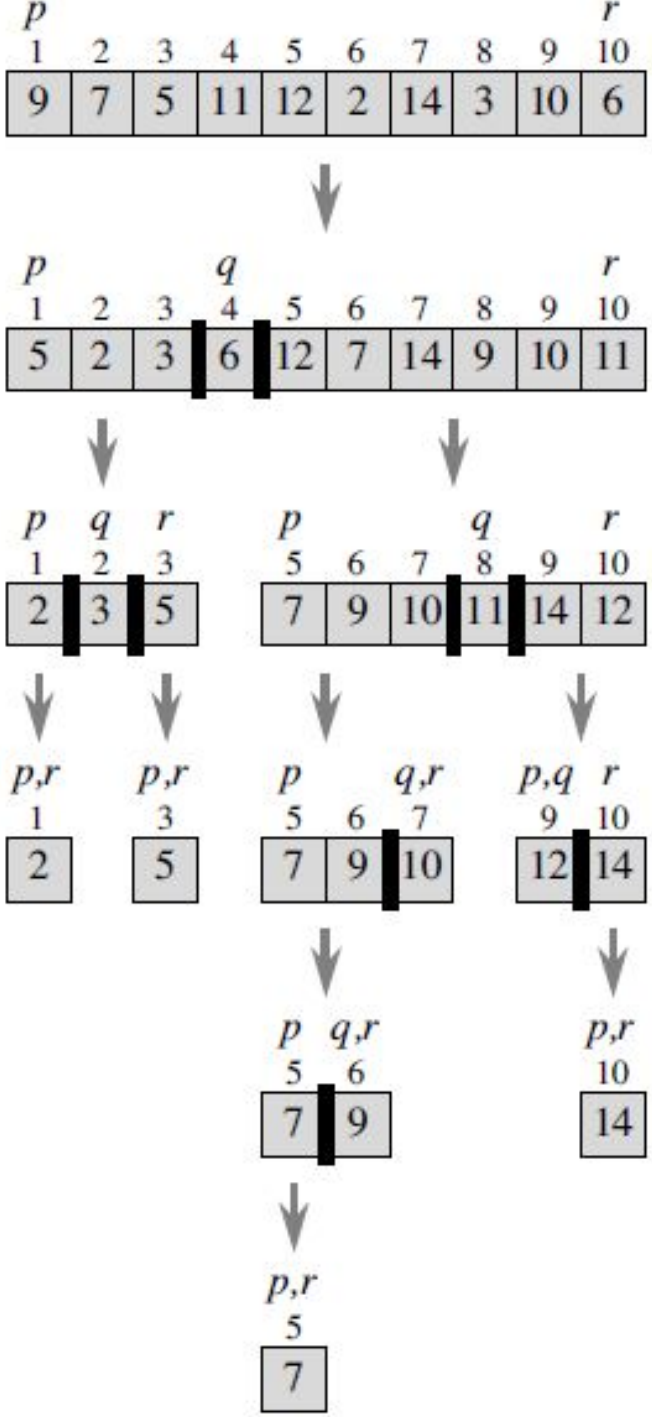
# Quick sort

*Procedure* QUICKSORT$(A, p, r)$

*Inputs and Result:* Same as MERGE-SORT.

1. If $p \geq r$, then just return without doing anything.
2. Otherwise, do the following:
   A. Call PARTITION$(A, p, r)$, and set $q$ to its result.
   B. Recursively call QUICKSORT$(A, p, q-1)$.
   C. Recursively call QUICKSORT$(A, q+1, r)$.

The procedure PARTITION (A, p, r) that partitions the subarray A[p; r], returning the index q where it has placed the pivot.


p r
1 2 3 4 5 6 7 8 9 10
9 7 5 11 12 2 14 3 10 6
p q r
1 2 3 4 5 6 7 8 9 10
5 2 3 6 12 7 14 9 10 11
p q r
1 2 3
2 3 5
p q r
5 6 7 8 9 10
7 9 10 11 14 12
p,r p,r
1 3
2 5
p q,r
5 6 7
7 9 10
p,q r
9 10
12 14
p q,r
5 6
7 9
p,r
10
14
p,r
5
7

# Quick sort

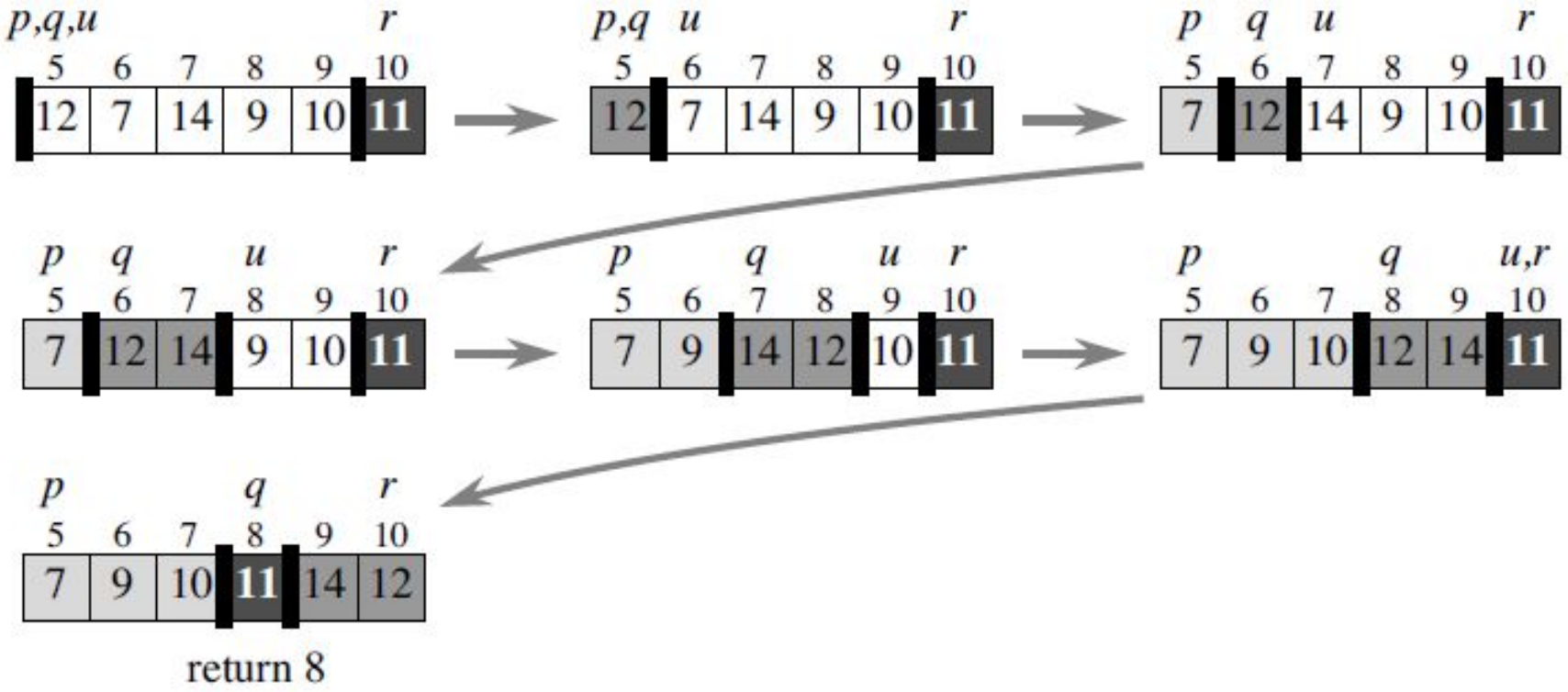
*Procedure* PARTITION$(A, p, r)$

*Inputs:* Same as MERGE-SORT.

*Result:* Rearranges the elements of $A[p\,..\,r]$ so that every element in $A[p\,..\,q-1]$ is less than or equal to $A[q]$ and every element in $A[q+1\,..\,r]$ is greater than $q$. Returns the index $q$ to the caller.

1. Set $q$ to $p$.
2. For $u = p$ to $r - 1$ do:
   A. If $A[u] \leq A[r]$, then swap $A[q]$ with $A[u]$ and then increment $q$.
3. Swap $A[q]$ with $A[r]$ and then return $q$.

# Quick sort


p,q,u
r
5 6 7 8 9 10
12 7 14 9 10 11
p,q u
r
5 6 7 8 9 10
12 7 14 9 10 11
p q u
r
5 6 7 8 9 10
7 12 14 9 10 11
p q u
r
5 6 7 8 9 10
7 12 14 9 10 11
p q u r
5 6 7 8 9 10
7 9 14 12 10 11
p q u,r
5 6 7 8 9 10
7 9 10 12 14 11
p q r
5 6 7 8 9 10
7 9 10 11 14 12
return 8

# Quick sort

In better case quicksort has the running time **Θ(nlgn).**

In the worst case quicksort has the running time **Θ($n^2$).**

# Recap

*Searching algorithms*

| Algorithm | Worst-case running time | Best-case running time | Requires sorted array? |
|---|---|---|---|
| Linear search | $\Theta(n)$ | $\Theta(1)$ | no |
| Binary search | $\Theta(\lg n)$ | $\Theta(1)$ | yes |

# Recap

*Sorting algorithms*

| Algorithm | Worst-case running time | Best-case running time | Worst-case swaps | In-place? |
|---|---|---|---|---|
| Selection sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ | yes |
| Insertion sort | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ | yes |
| Merge sort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | no |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \lg n)$ | $\Theta(n^2)$ | yes |