Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 24
Recursive Sorting and Complexity

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Overview

- To review in more depth two different“divide and conquer” sorting algorithms:
    - **Merge Sort**
    - **Quick Sort**

- To be able to **implement** them and compare their efficiency for different classes of inputs

Divide and Conquer: **Sorting**

General Idea

```
def sort(array):
    if len(array) > 1:
        split(array, first_part, second_part)
        sort(first_part)
        sort(second_part)
        combine(first_part, second_part)
```

- **Merge Sort** has a simple split and a elaborate combine
- **Quick Sort** has a elaborate split and a simple combine

Merge Sort

```
def merge_sort(array):
    tmp = [None] * len(array)
    start = 0
    end = len(array)-1
    merge_sort_aux(array, start, end, tmp)
```

the array   start index   end index   temporary array

```
def merge_sort_aux(array, start, end, tmp):
```

Merge Sort

```
def merge_sort_aux(array, start, end, tmp):
    if start < end: # 2 or more still to sort
        mid = (start + end)//2

        # split into two halves
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)

        # merge
        merge_arrays(array, start, mid, end, tmp)
```

sorted result

define what the “two lists” are

Merge Sort

```python
def merge_sort_aux(array, start, end, tmp):
    if start < end: # 2 or more still to sort
        mid = (start + end)//2

        # split into two halves
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)

        # merge
        merge_arrays(array, start, mid, end, tmp)

        # copy tmp back into the original
        for i in range(start,end+1):
            array[i] = tmp[i]
```

Merge

L:

3	5	15	28	30	32

R:

10	14	22	43	50

Sorted Lists

start: 0, mid: 5, end: 10

tmp:

Merge

L:

3	5	15	28	30	32

R:

10	14	22	43	50

`i=6`

`j=5`

tmp:

3	5	10	14	15	22	28	30	32	43	50

```python
def merge_arrays(array, start, mid, end, tmp):
```

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
```

Set-up indices: i for the left, j for the right.

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):
```

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):
```

k loops through (relevant part of) tmp

```
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):







        elif array[i] <= array[j]: # array[i] is the item to copy
            tmp[k] = array[i]
            i += 1
```

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):




        elif array[i] <= array[j]: # array[i] is the item to copy
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j] # array[j] is the item to copy
            j += 1
```

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):
        if i > mid: # left finished, copy right
            tmp[k] = array[j]
            j += 1


        elif array[i] <= array[j]: # array[i] is the item to copy
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j] # array[j] is the item to copy
            j += 1
```

```python
def merge_arrays(array, start, mid, end, tmp):
    i = start
    j = mid+1
    for k in range(start, end+1):
        if i > mid: # left finished, copy right
            tmp[k] = array[j]
            j += 1
        elif j > end: # right finished, copy left
            tmp[k] = array[i]
            i += 1
        elif array[i] <= array[j]: # array[i] is the item to copy
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j] # array[j] is the item to copy
            j += 1
```

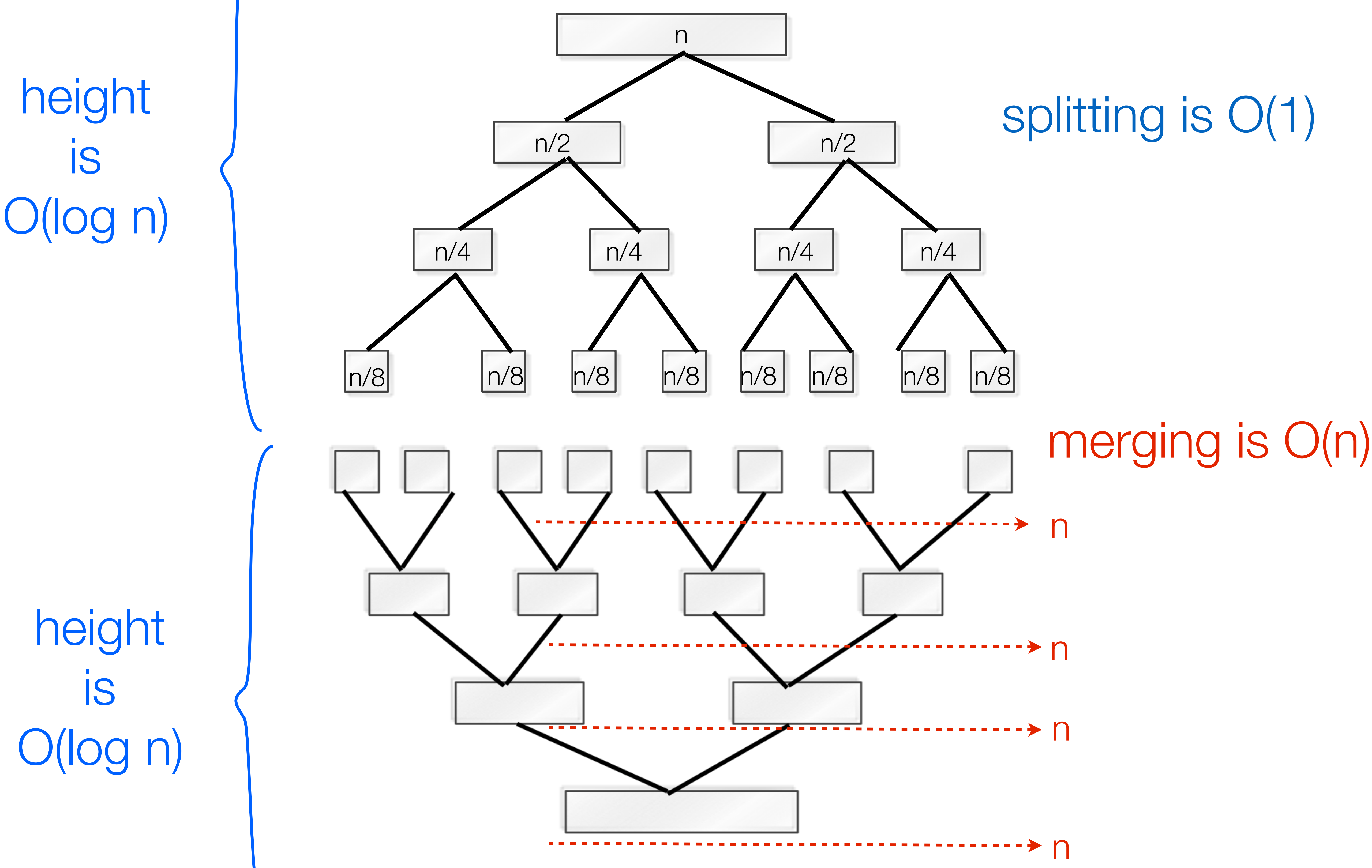
Key idea for complexity of recursion:

- How many recursive calls do we make in each version?
- How much work do we do per call?

Merge Sort Analysis

- **Natural:** Typically the method that you would use when sorting a pile of books, CDs cards, etc.
- Most of the work is in the **merging**
- Uses more **space** than other sorts
- Close to optimal in number of comparisons. Good for languages where comparison is expensive.

Merge sort



Total Running Time: O(n log n)

Quicksort

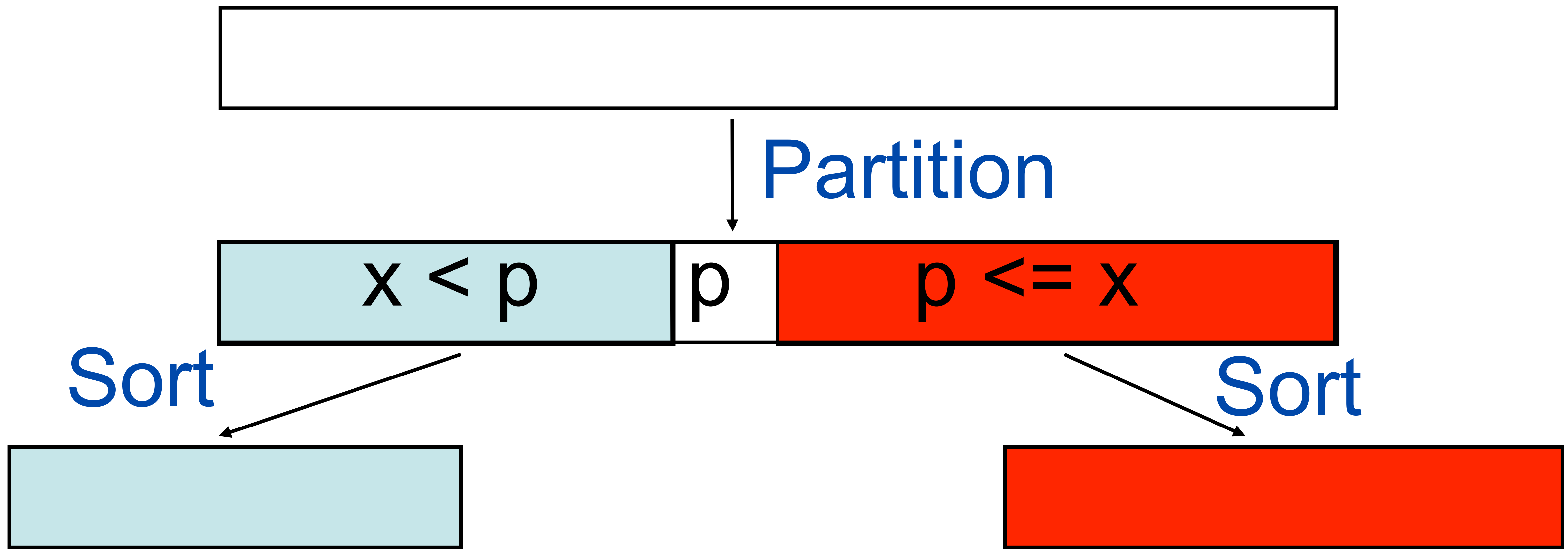
Top-10 algorithms 20th century (SIAM)

Quick Sort

- **Partition** the list
- Sort the first part (recursively)
- Sort the second part (recursively)

Partition

- Choose an item in the list, called it the **pivot**.
- The **first part** consists of all those **items** which are **less than the pivot**.
- The **second part** consists of all those **items larger than or equal to the pivot (except the pivot)**.



- **Partition:** Elaborate, based on a pivot p.

- **Combination:** Simple append, pivot in the middle.

Example Partition

`array:`

5	89	35	14	24	15	37	13	20	7	70

`start:0` **`end:10`**

Example Partition

array:

5	89	35	14	24	15	37	13	20	7	70

Randomly choose a pivot, which happens to be in the middle

Example Partition

`array:`

5	89	35	14	24	15	37	13	20	7	70

partition:

5	89	35	14	24	15	37	13	20	7	70

result

7	14	5	13	15	35	37	89	20	24	70

pivot position: 4

note that the pivot defines the boundaries

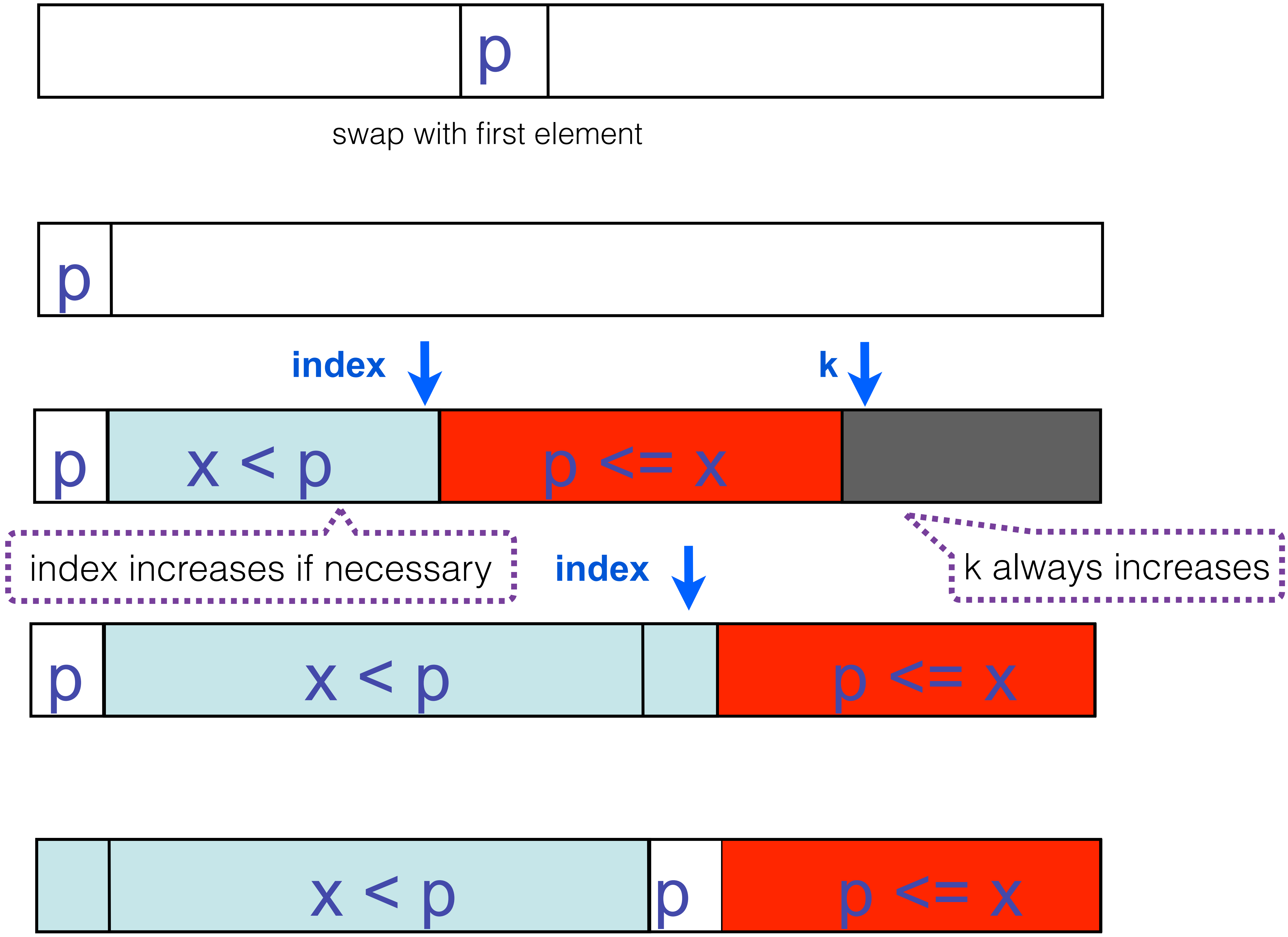
sort first half (using QS), sort second half (using QS)

Quicksort

```
def quick_sort(array):
    start = 0
    end = len(array)-1
    quick_sort_aux(array, start, end)

def quick_sort_aux(array, start, end):
    if start < end:
        boundary = partition(array, start, end)
        quick_sort_aux(array, start, boundary-1)
        quick_sort_aux(array, boundary+1, end)
```

sort second half


p
swap with first element
p
index
k
p
x < p
p <= x
index increases if necessary
index
k always increases
p
x < p
p <= x
x < p
p
p <= x

Example Partition

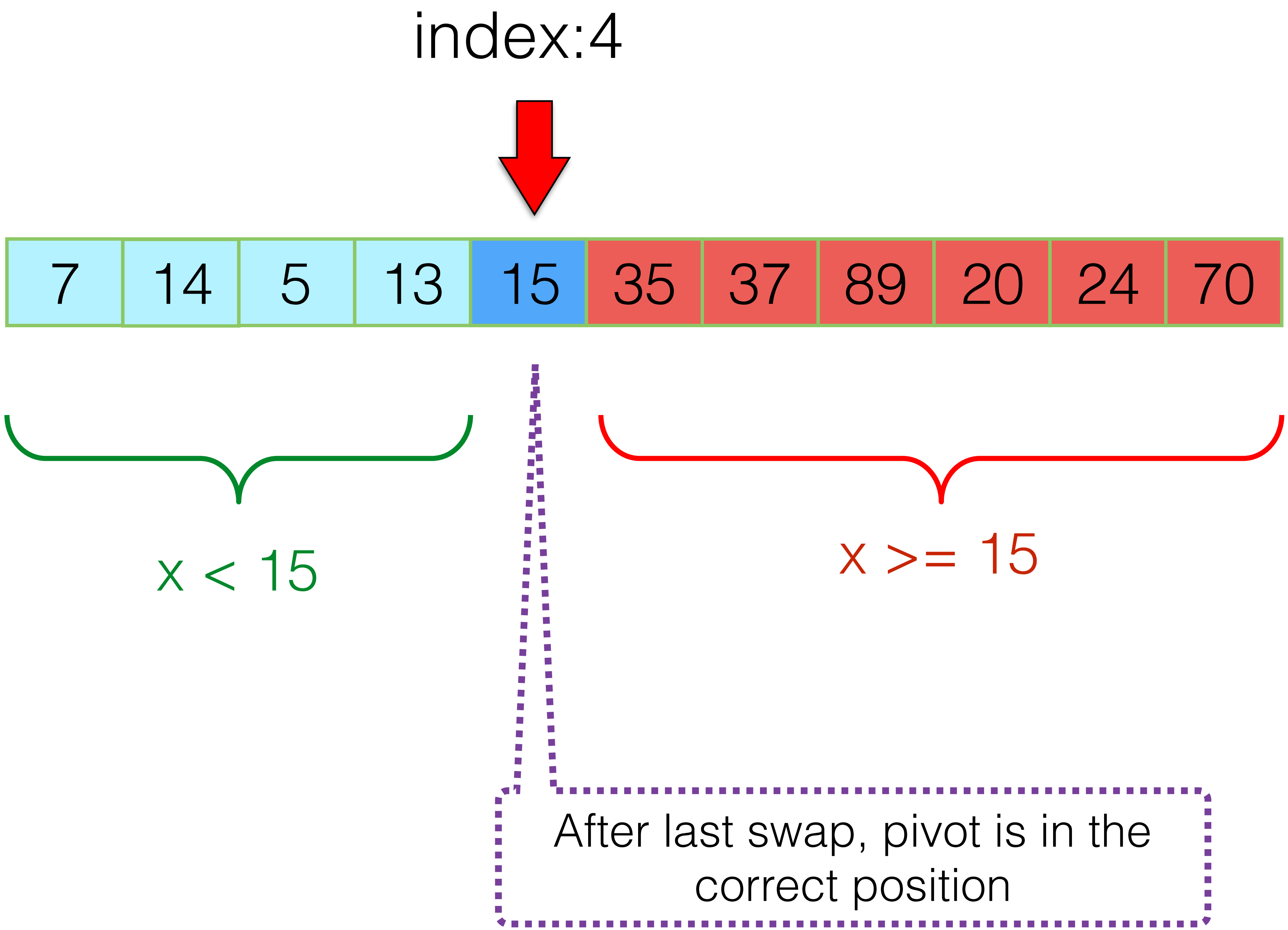
randomly pick element in position 5

array:

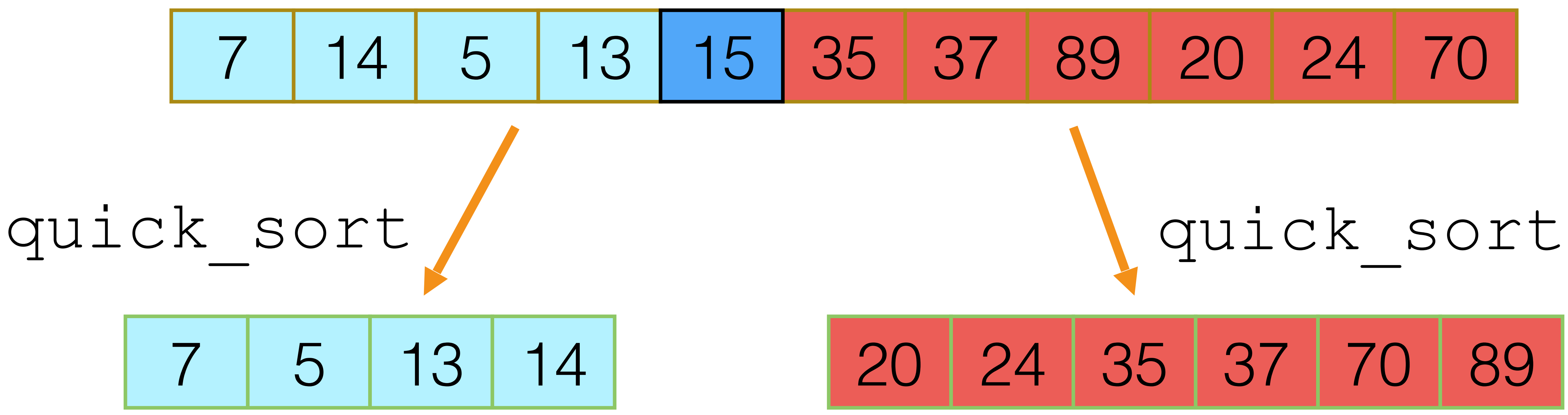
5	89	35	14	24	15	37	13	20	7	70

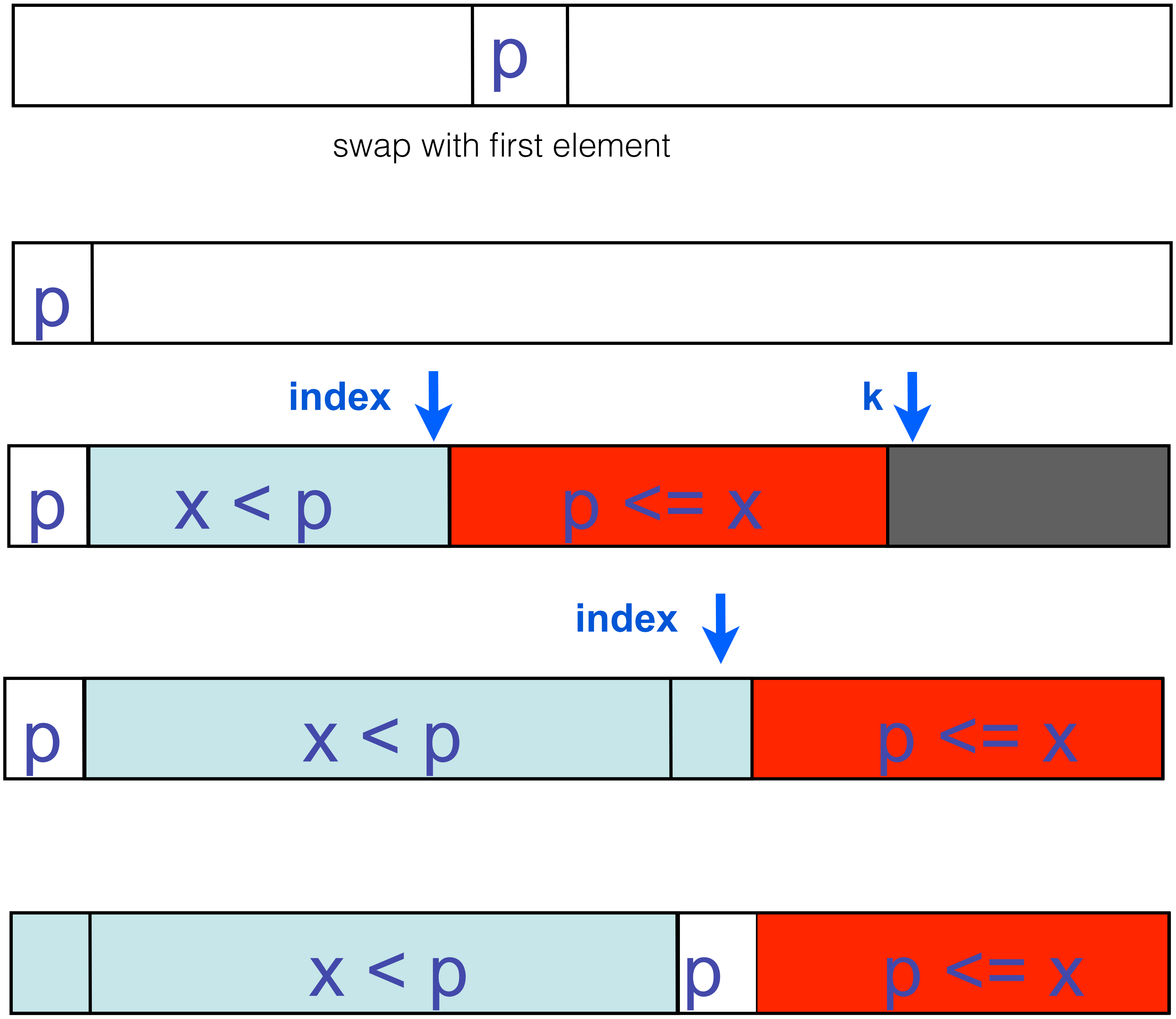
15	89	35	14	24	5	37	13	20	7	70

Example Partition



Example Partition

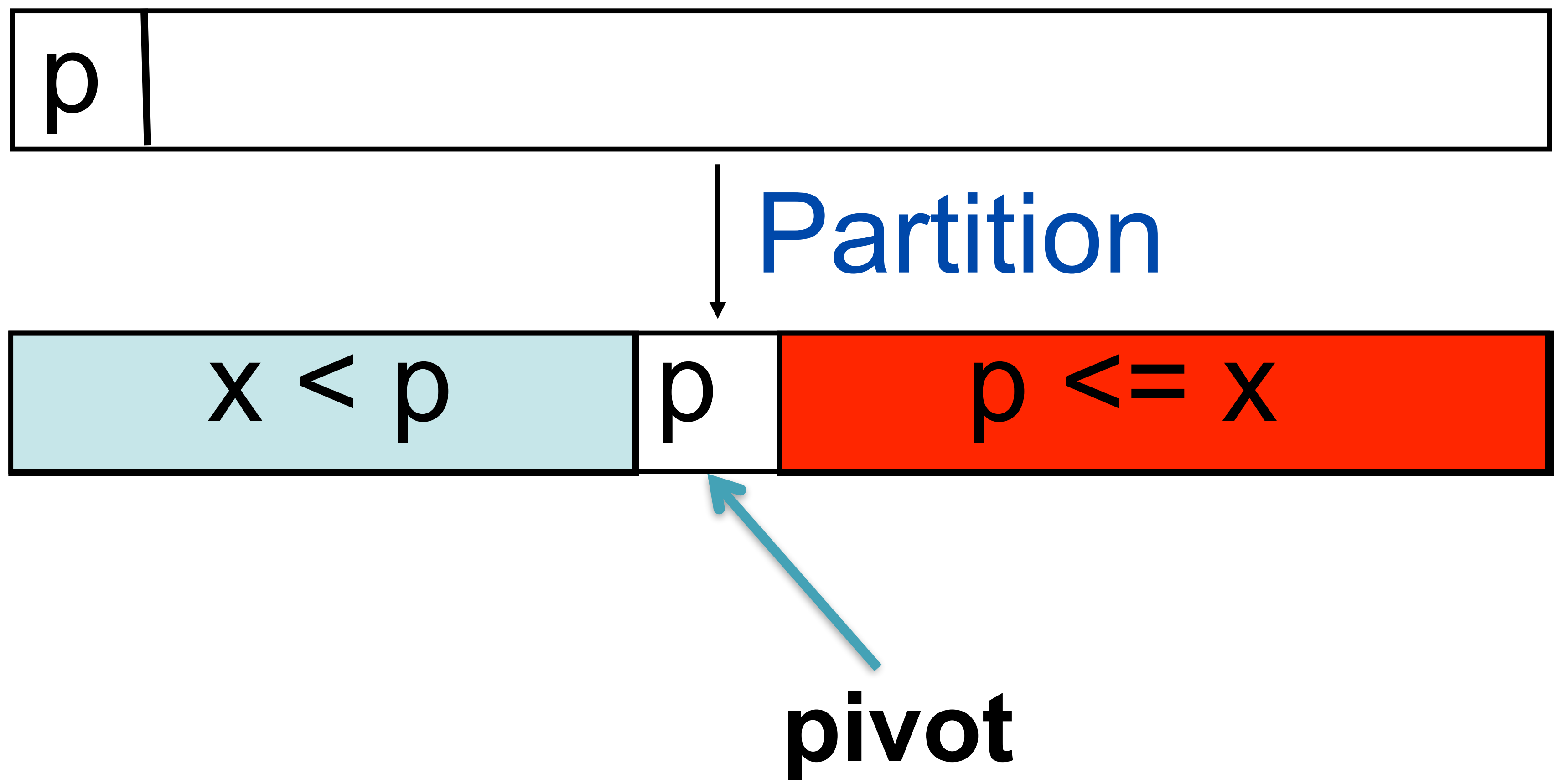



p
swap with first element
p
index
k
p
x < p
p <= x
index
p
x < p
p <= x
x < p
p
p <= x

```python
def swap(array, i, j):
    array[i], array[j] = array[j], array[i]
```

```python
def partition(array, start, end):
        mid = (start+end)//2
        pivot = array[mid]
        swap(array, start, mid)
        index = start
        for k in range(start+1,end+1):
            if array[k] < pivot:
                index += 1
                swap(array, k, index)
        swap(array, start, index)
        return index
```

Quicksort: Number of partitions depends on the pivot

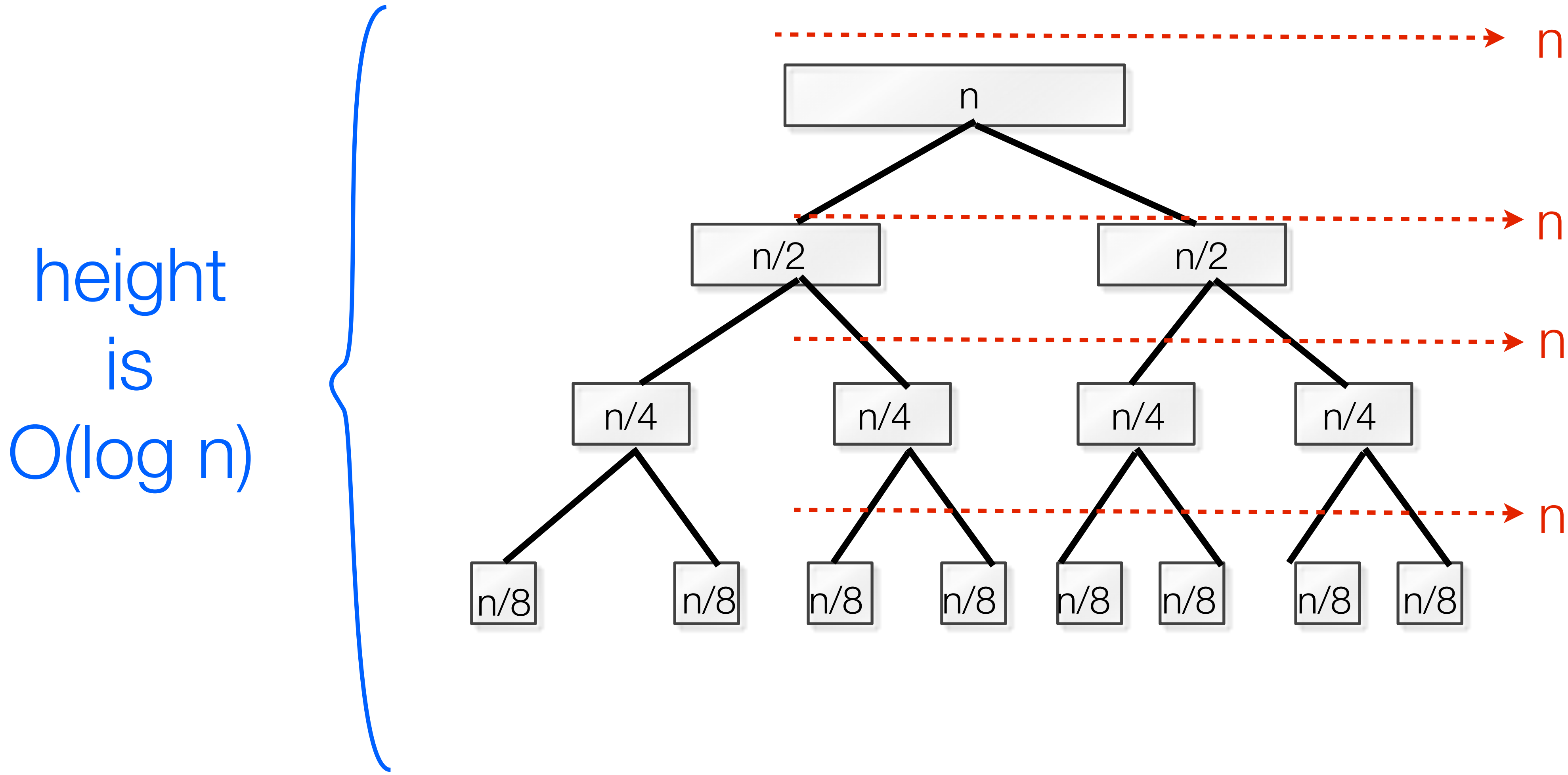


Best case: The size of the problem is reduced by half with every partition

Worst case: The size of the problem is reduced by 1 with every partition

Quick sort's best case

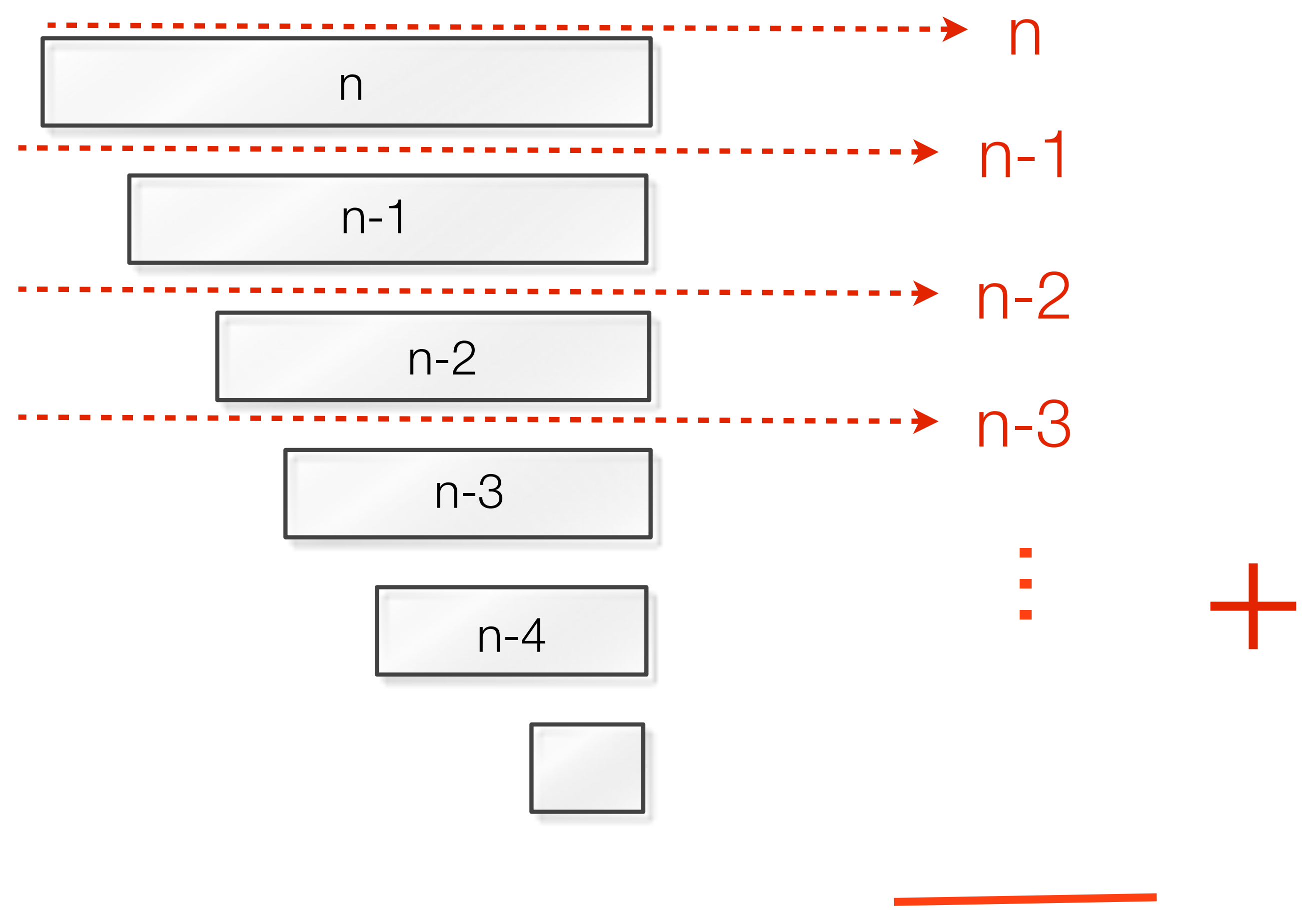
partition is O(n)



Running time in the best case: O(n log n)

Quick sort's worst case

partition is O(n)



$$\frac{n\,(n+1)}{2}$$

Running time in the worst case: $O(n^2)$

Summary

	Best case	Worst case
Quicksort	$O(n \log n)$	$O(n^2)$
Mergesort	$O(n \log n)$	$O(n \log n)$

How common is quicksort's worst case?

Not too common if choosing a random pivot.

Summary

Divide and Conquer and Recursive Algorithms (for sorting).

Merge Sort

- Easy: Split
- Elaborate: merge method

Quick Sort

- Elaborate split: partition method
- Easy combination