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

Lecture 13
Recursive sorting

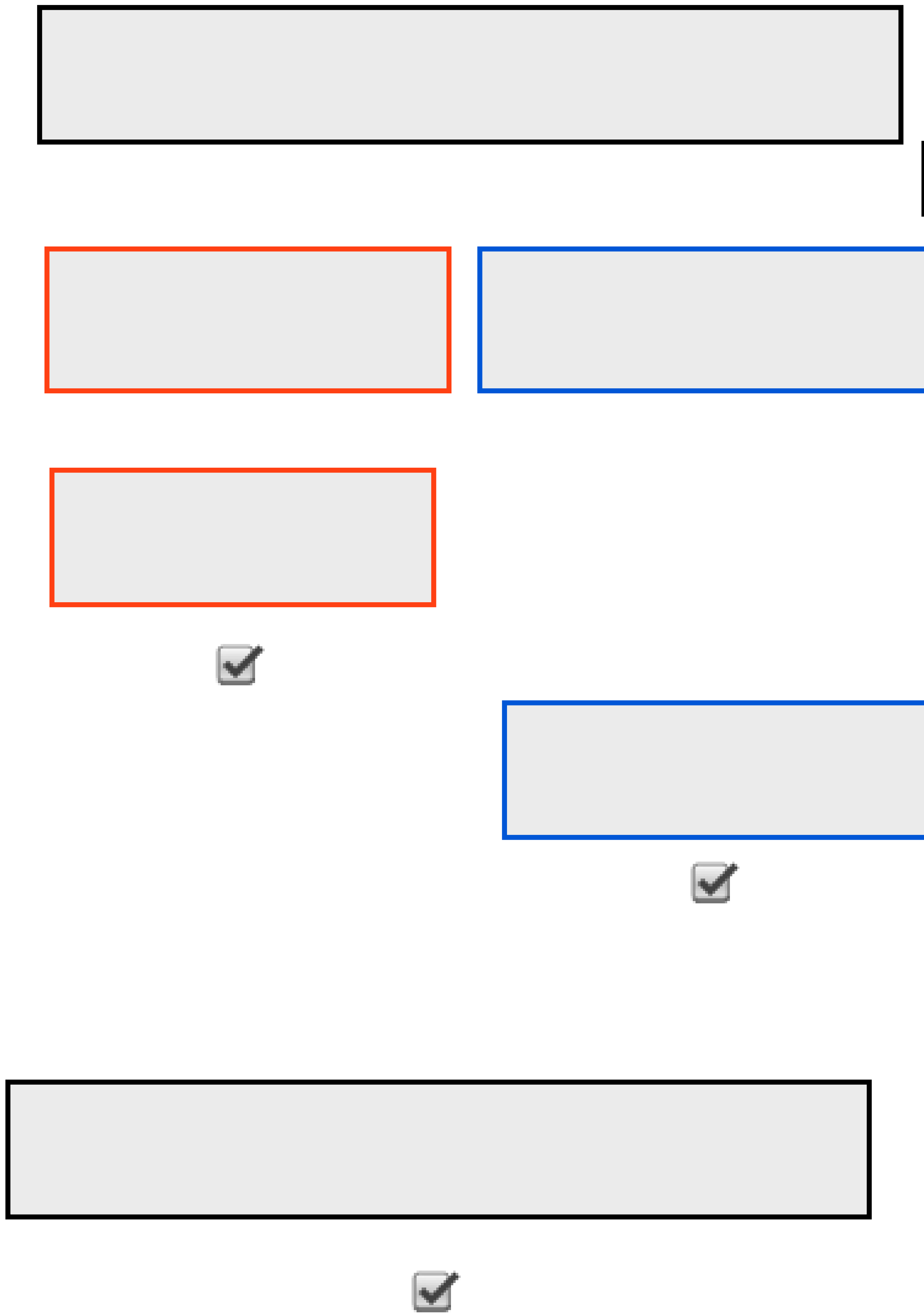
FIT 1008
Introduction to Computer Science

Can we do better?

Overview

- To review what a “divide and conquer” algorithm is
- 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



Divide the array into 2 parts

Sort the first part

Sort the second part

Combine

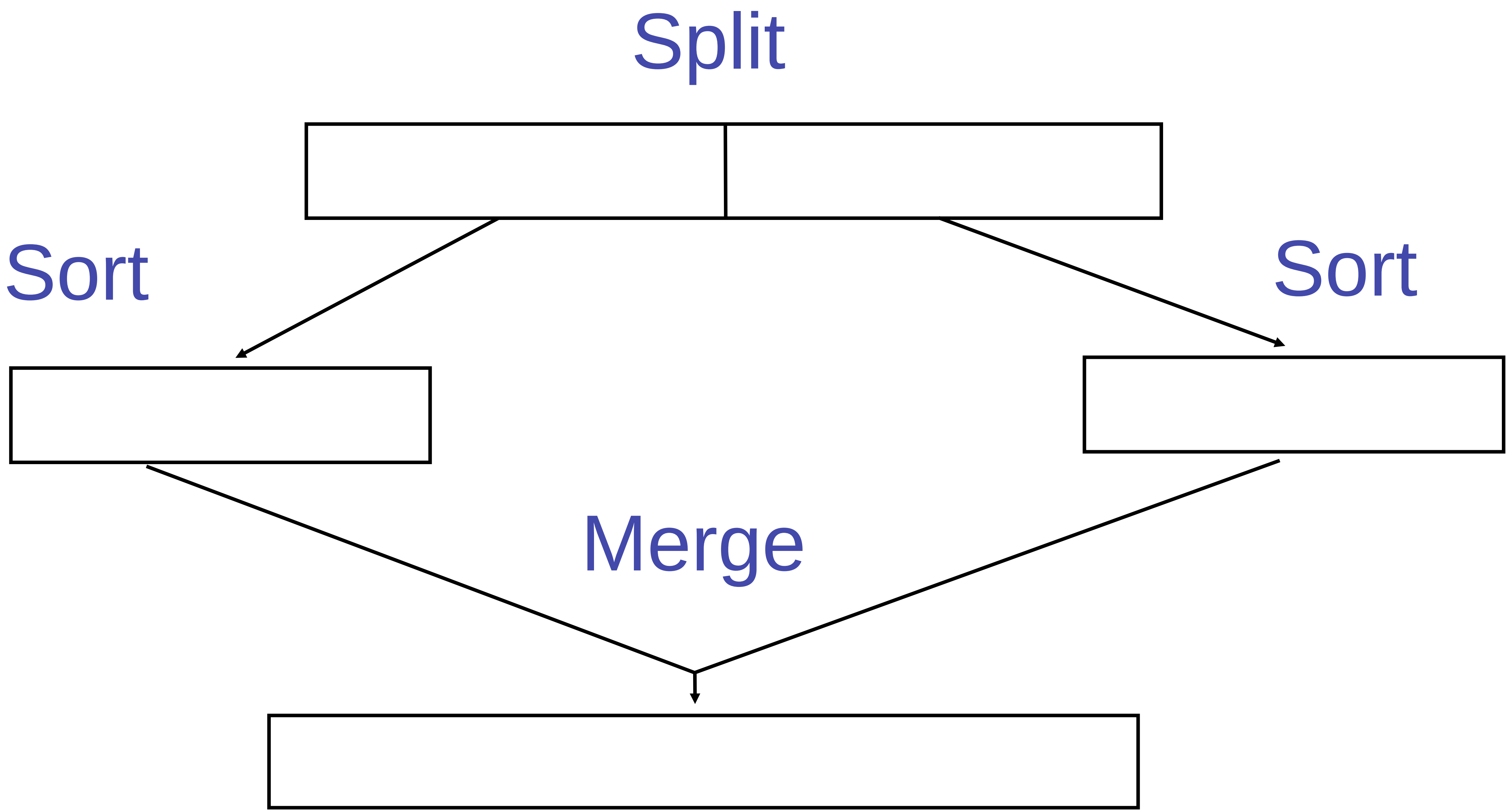
Divide and Conquer: Sorting

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)
```

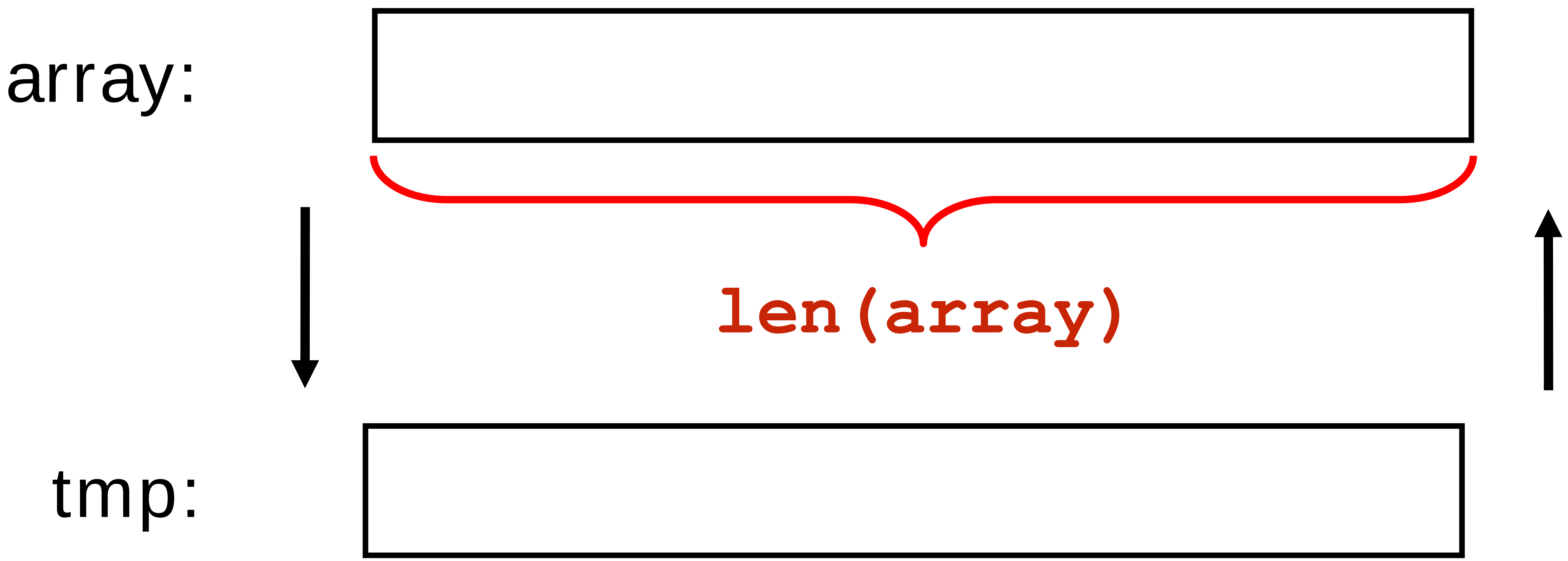
- Merge Sort has a simple split and a elaborate combine
- Quick Sort has a elaborate split and a simple combine

Merge Sort



- Split: In half.
- Merge: Take two unsorted arrays, produce a sorted one.

Merge Sort



Use a temporary array, then copy back to original array.

Merge Sort

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

No return statement, array will be changed

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)

        # 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

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):
    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
```

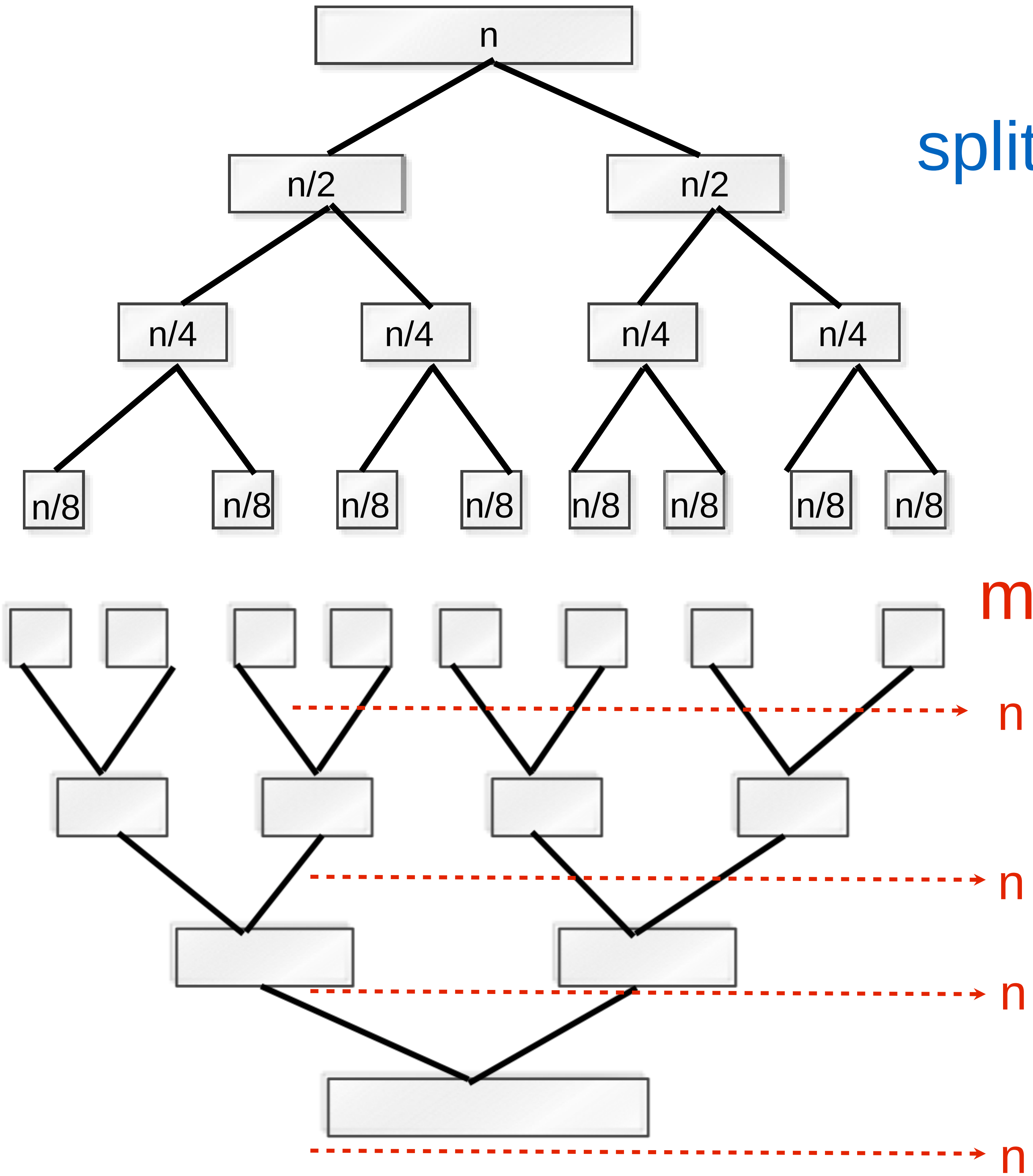
Merge Sort Analysis

- 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

height is O(log n)

splitting is O(1)



merging is O(n)

height is O(log n)

Total Running Time: O(n log n)

Quicksort

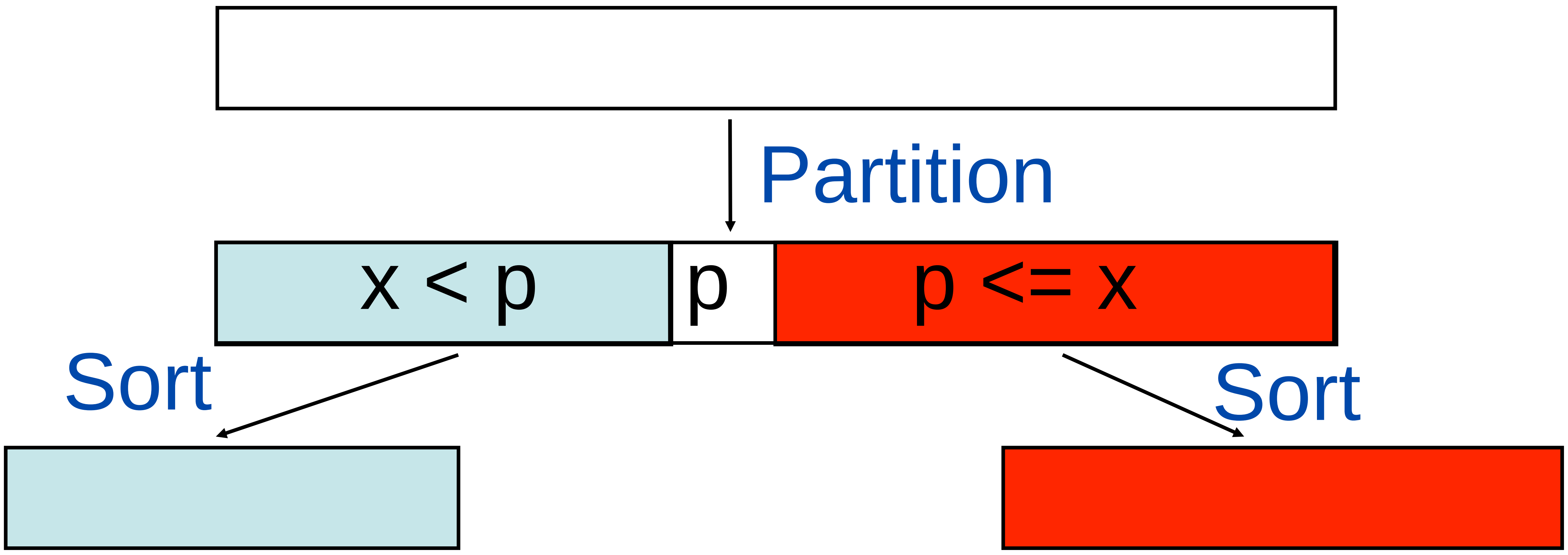
- Top-10 algorithms 20th century (SIAM)

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

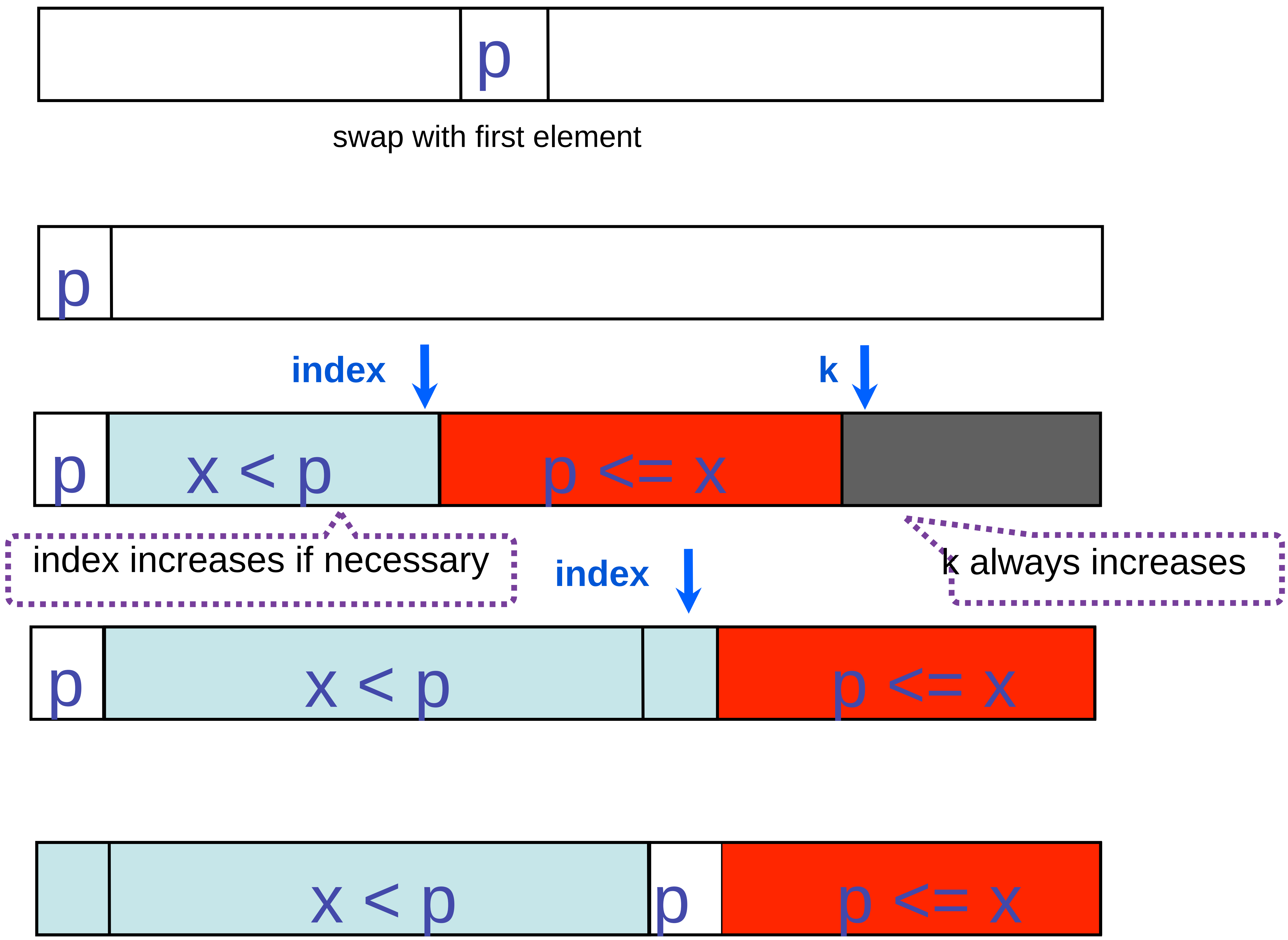
sort first half (using QS), sort second half (using QS)

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)
```

How do we partition?


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

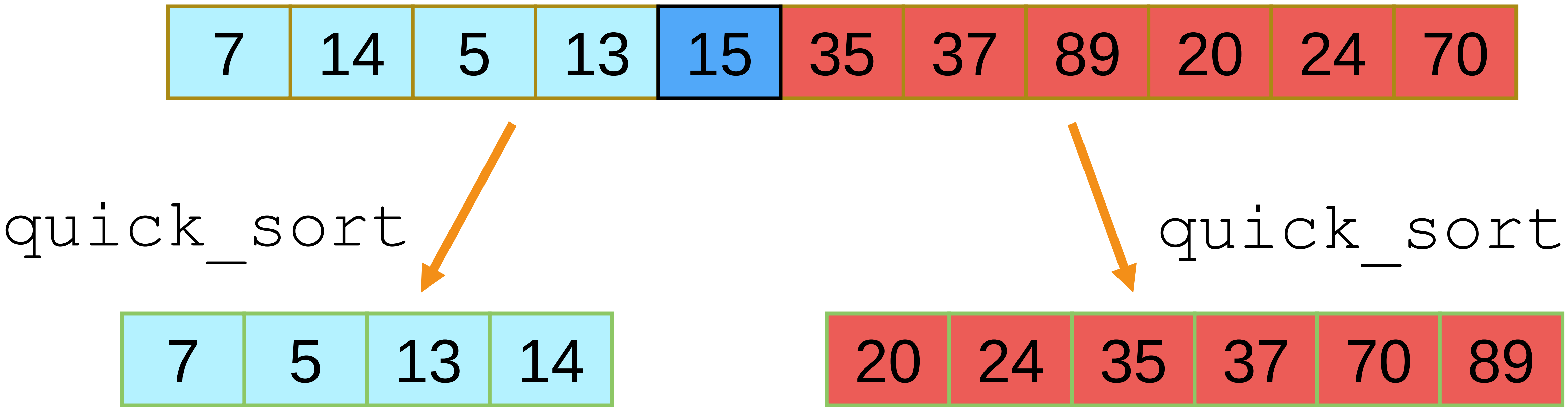
randomly pick element in position 5

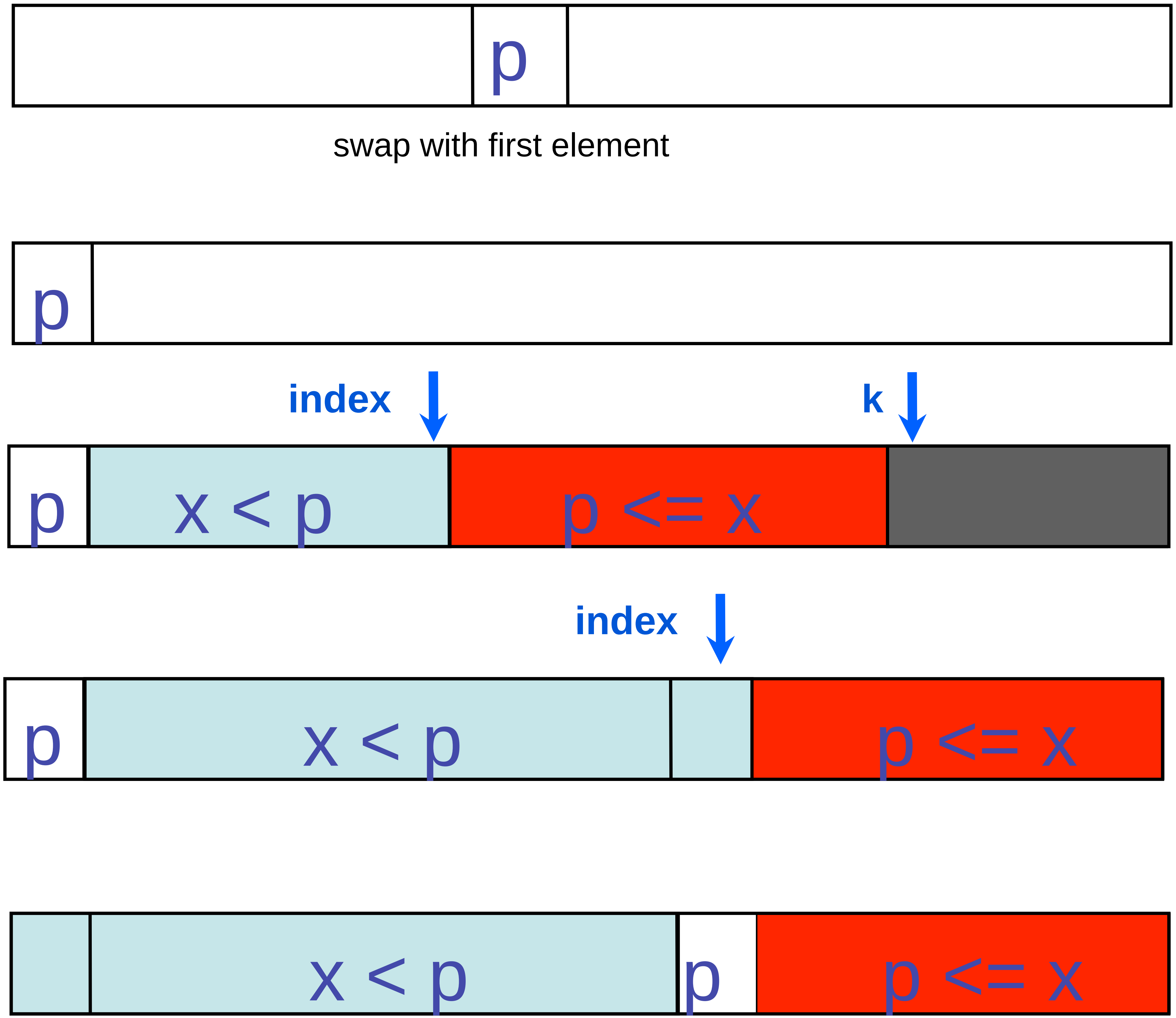
array:

5	89	35	14	24	15	37	13	20	7	70

15	89	35	14	24	5	37	13	20	7	70

Example Partition

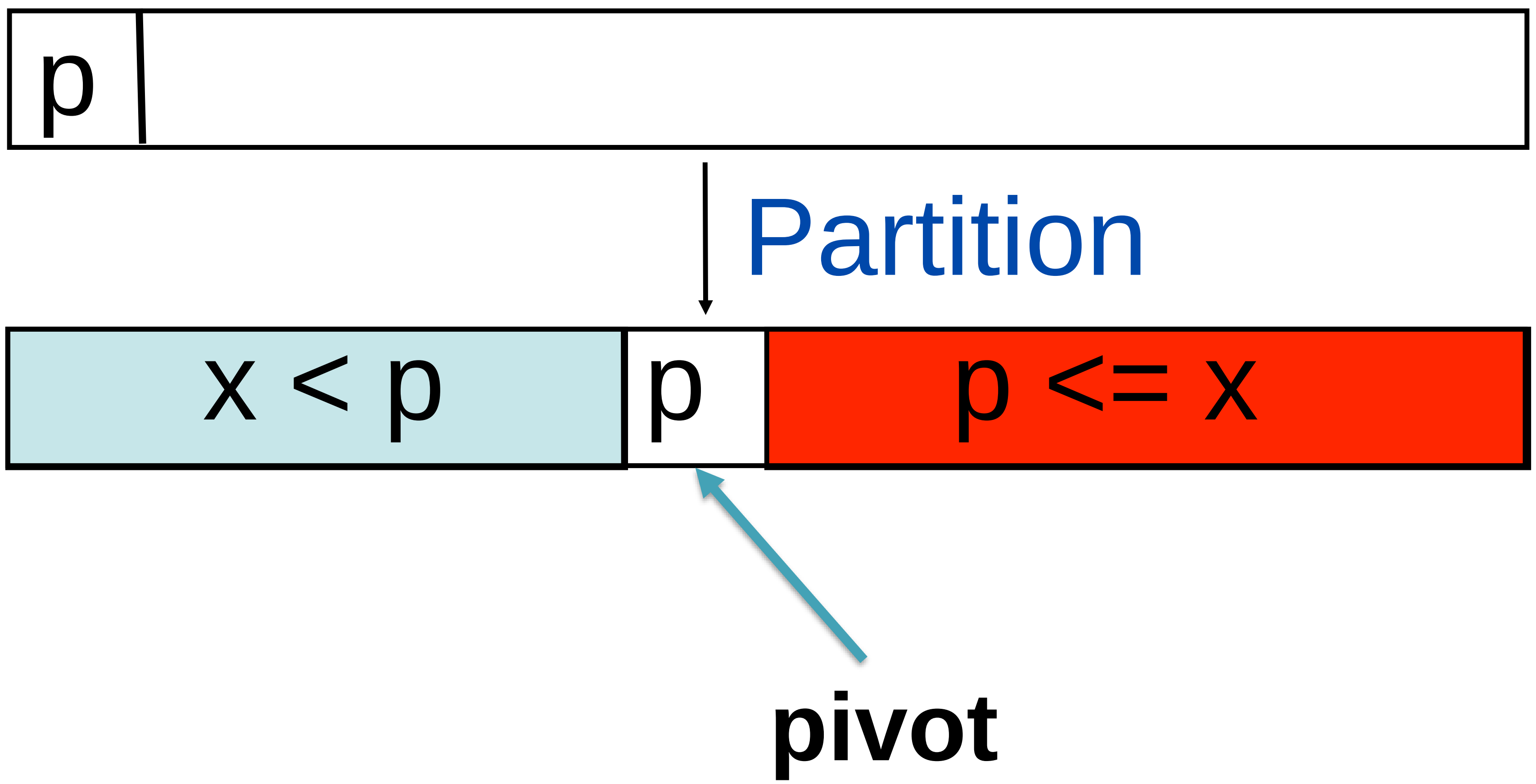



p
swap with first element
p
index
k
p
x < p
p <= x
index
p
x < p
p <= x
x < p
p
p <= x

```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



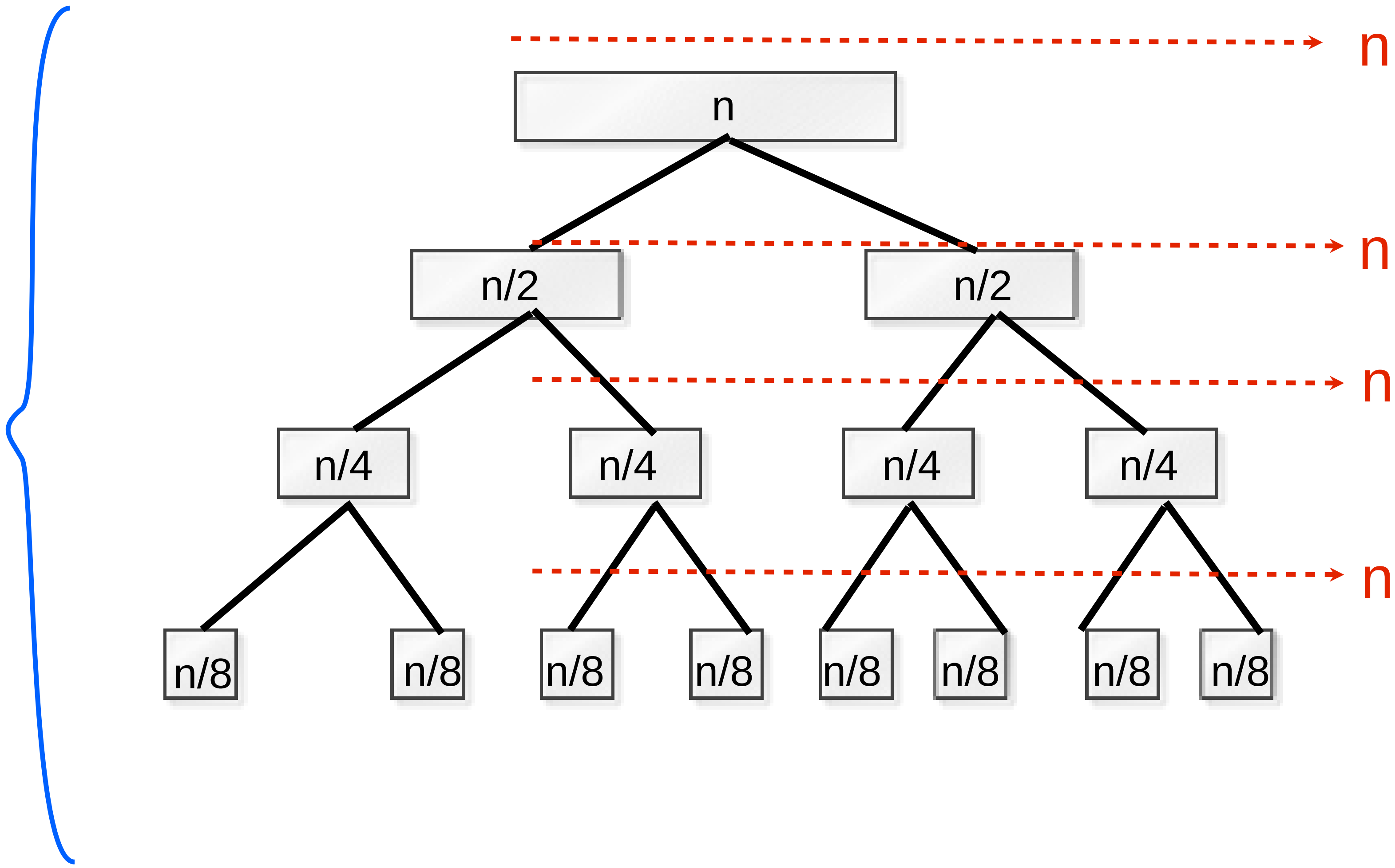
Best case: The size of the problem is reduced by half with every partition

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

Quick sort's best case

partition is O(n)

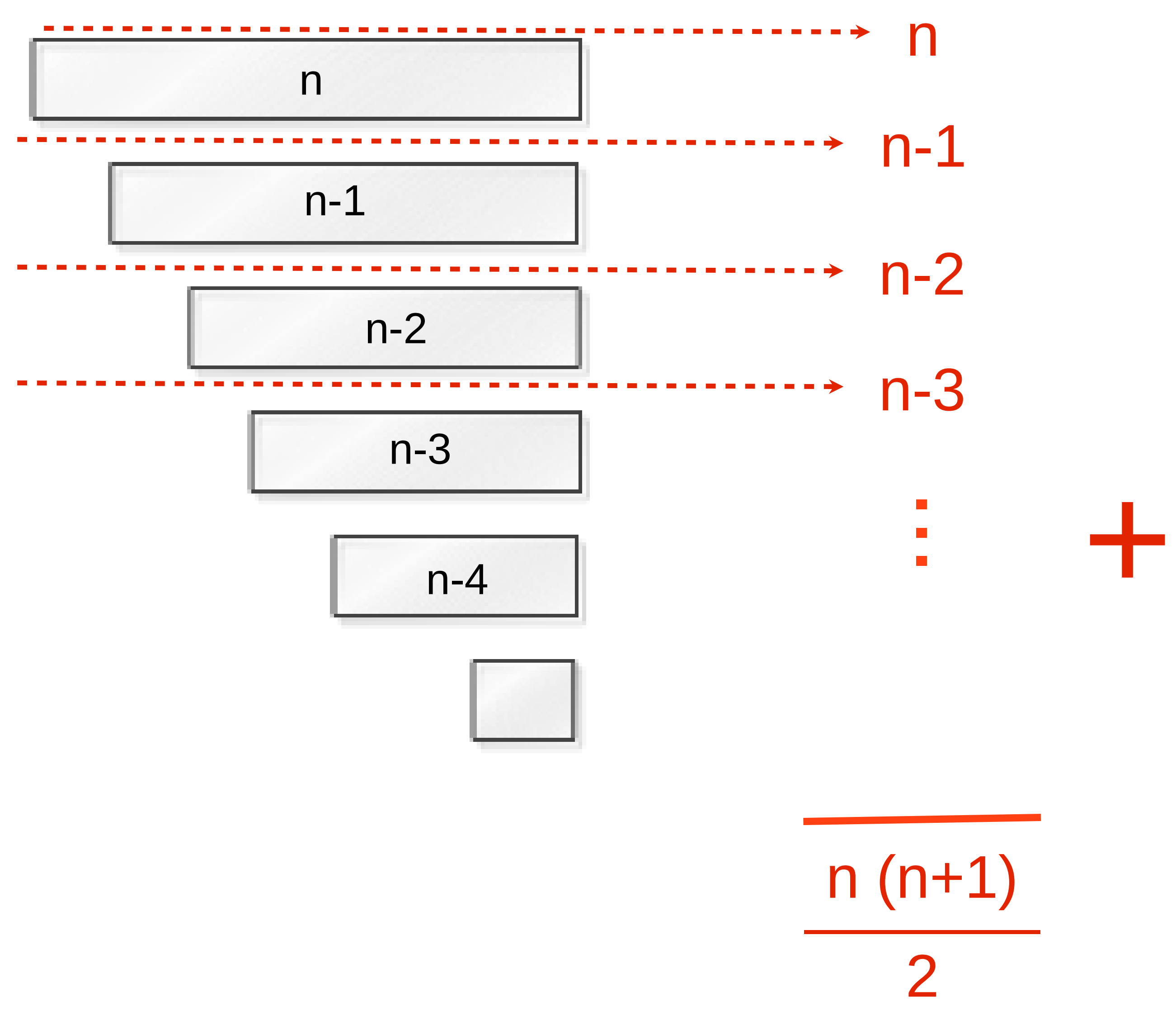
height is O(log n)



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

Quick sort’s worst case

partition is $O(n)$



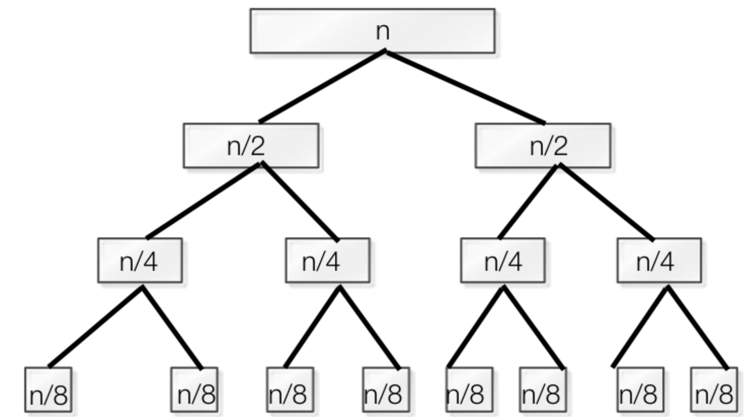
Running time in the worst case: $O(n^2)$

Another view of complexity

Quicksort

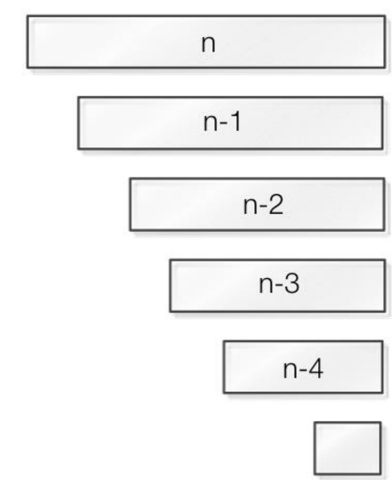
$$T(n) = O(n) + 2 \cdot T(\frac{n}{2})$$

(best case)



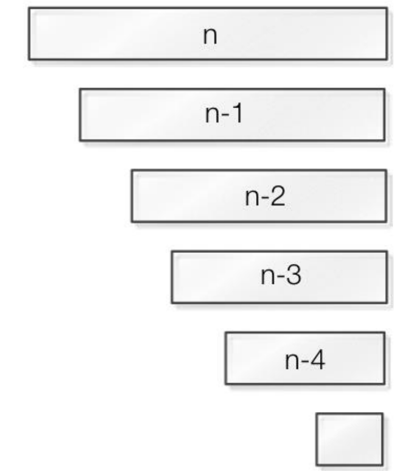
$$T(n) = O(n) + T(n-1)$$

(worst case)



Quicksort

(worst case)



$$T(n) = O(n) + T(n-1)$$

$$T(n) = O(n) + O(n-1) + T(n-2)$$

$$T(n) = O(n) + O(n-1) + O(n-2) + T(n-3)$$

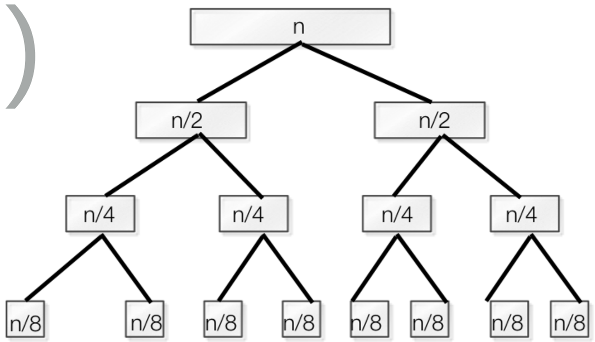
$$\vdots$$

$$T(n) = O(n) + O(n-1) + O(n-2) + \cdots + O(2) + T(1)$$

$$T(n) = O(n^2)$$

Quicksort

(best case)



$$T(n) = O(n) + 2 \cdot T(\frac{n}{2})$$

c = 1

a = 2

b = 2

2 = 2^1

case 2

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

Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^c)$$

$$T(1) = k$$

$a \geq 1, b > 1, c > 0$

case 1:

$$a > b^c \longrightarrow T(n) \in O(n^{\log_b a})$$

case 2:

$$a = b^c \longrightarrow T(n) \in O(n^c \log_b n)$$

case 3:

$$a < b^c \longrightarrow T(n) \in O(n^c)$$

cost of combining solutions

number of sub-problems to solve

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^c)$$

reduced problem size

$$T(n) = O(n) + 2 \cdot T(\frac{n}{2})$$

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

Useful for divide and conquer and recursive approaches

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