Computation II

Lecture 12

João Fonseca

jpfonseca@novaims.unl.pt

Adapted from Prof. Dr. Leonardo Vanneschi's class materials

NOVA Information Management School

Agenda

- Sorting Algorithms
  - Insertion sort
  - Merge sort
  - Quick sort

Sorting

We sort many things in our everyday lives: A handful of cards; bills and other piles of paper; jars of spices; and so on…

And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around.

Sorting is also one of the most frequently performed computing tasks.

Sorting

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised.

Some of these algorithms are straightforward adaptations of schemes we use in everyday life. Others are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer.

Sorting

While introducing this central problem in computer science, this part of the course also has the secondary purpose of illustrating many important issues in algorithm design and analysis.

For instance, the collection of sorting algorithms presented will illustrate that *divide-and-conquer* is a powerful approach to solving a problem, and that there are multiple ways to do the dividing.

For instance: Mergesort divides a list in half, Quicksort divides a list into big values and small values.

Sorting

This part of the course covers several standard algorithms, appropriate for sorting a collection of records (values) that fit in the computer's main memory.

Basic Concepts of Sorting

Definition of the sorting problem

the sorting problem consists in *arranging a set of values in non-decreasing order*.

For instance, given the sequence of values:
7 3 2 4 3 8 5
Rearrange the sequence in order to obtain:
2 3 3 4 5 7 8

For simplicity, in the continuation, we will limit the study to *sequences of numbers*. But the same method can apply, for instance, to strings (that can be sorted in alphabetical order) or for any other type of object on which it is possible to define an order.

The function swap

Basically, all the algorithms that we will study in the next slides make use of a function that swaps (i.e. exchanges) two elements in a vector.

So, *from now on, we will consider that we have this function already implemented*.

```
function swap
    input: a vector A and two indices of A called i and j
    effect: swapping elements A[i] and A[j]
```

The function swap - Example

Let us assume that the vector A is:

0	1	2	3	4	5	6	7
2	**7**	**9**	**5**	**3**	**8**	**1**	**4**

Then the function **swap(A, 2, 5)** modifies the vector in the following way:

0	1	2	3	4	5	6	7
2	**7**	**8**	**5**	**3**	**9**	**1**	**4**

The function swap - Example

Conceptually, the function swap *CANNOT be implemented without using a supplementary variable*, that is used to temporary store one of the values that must be swapped!

So, the algorithm that implements the function swap is:

```
def swap (a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

a is a list, ***i*** and ***j*** are indexes

(Remark: in Python, you can do it in one shot: array[i], array[j] = array[j], array[i])

The function swap - Example

swap(A, 2, 5)

0	1	**2**	3	4	**5**	6	7
2	7	9	5	3	8	1	4

temp = 9

0	1	**2**	3	4	**5**	6	7
2	7	8	5	3	8	1	4

0	1	**2**	3	4	**5**	6	7
2	7	8	5	3	9	1	4

Sorting Algorithms - Basic Operations

For calculating the *computational complexity* of the sorting algorithms that we will study, we will ***count*** the number of times the following operations are executed:

- ***Number of comparisons*** between values (number of tests if a value is smaller or bigger than another)
- ***Number of swaps*** in the array

We make this choice because these are the two operations that are more used in the sorting algorithms (and so they can be used as basic operations).

Insertion Sort

Imagine that you have a stack of phone bills from the past two years and that you wish to organize them by date.

A fairly natural way to do this might be to look at the *first two* bills and put them in order. Then take the *third* bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the *sorted pile* that you have already made.

This naturally intuitive process is the inspiration for our first sorting algorithm, the Insertion Sort

Insertion Sort

Insertion Sort iterates through a list of elements.

Each record is inserted in turn at the correct position within a sorted list composed of those elements already processed.

In the next slides, we see the algorithm in different levels of abstraction and details.

Insertion Sort

```
function Insertion Sort
        input: an array of n non-sorted numbers called A
        output: the array A sorted


for all the elements of index i in A

    bring element i backwards in vector A, until it is in a position where all the
    elements until itself are sorted
```

Insertion Sort

```
function Insertion Sort
        input: an array of n non-sorted numbers called A
        output: the array A sorted


for all the elements of index i in A
  go backward in vector A and for all elements of index j from i to 0
     if the element in position j is smaller then the element in position j-1 swap
            the elements in position j and j-1
```

Insertion Sort

function Insertion Sort
    input: an array of n non-sorted numbers called A
    output: the array A sorted

```
for all i from 0 to n-1
  for all j from i to 0
    if (A[j] < A[j-1])
          swap (A, j, j-1)
```

Insertion Sort

```
function Insertion Sort
        input: an array of n non-sorted numbers called A
        output: the array A sorted


for all i from 0 to len(A)-1:
  for all j from i to 0 (backwards)
     if (A[j] < A[j-1])
            swap (A, j, j-1)
```

Insertion Sort

function Insertion Sort
input: an array of n non-sorted numbers called A
output: the array A sorted

```
for all i from 0 to len(A)-1:
    for all j from i to 0 (backwards)
        if (A[j] < A[j-1])
            swap (A, j, j-1)
```

```python
def sort_insertion(a):
    for i in range(1, len(a)):
        for j in range(i, 0, -1):
            if a[j] < a[j-1]:
                swap(a, j, j-1)
```

Example of an unsorted array: [1 2 1 8 9 8 0 7 3 7]
Example of a sorted array: [0 1 1 2 3 7 7 8 8 9]
Total: compares=45 & swaps=18

Insertion Sort

function Insertion Sort

input: an array of n non-sorted numbers called A
output: the array A sorted

for all i from 0 to len(A)-1:
  for all j from i to 0 (backwards)
    if (A[j] < A[j-1])
      swap (A, j, j-1)
	def sort_insertion(a):
	  for i in range(1, len(a)):
	    for j in range(i, 0, -1):
	      if a[j] < a[j-1]:
	        swap(a, j, j-1)

Example of an unsorted array: [1 2 1 8 9 8 0 7 3 7]
Example of a sorted array: [0 1 1 2 3 7 7 8 8 9]
Total: compares=45 & swaps=18

Insertion Sort

What if the array is almost sorted?

```
def sort_insertion(a):
    for i in range(1, len(a)):
        for j in range(i, 0, -1):
            if a[j] < a[j-1]:
                swap(a, j, j-1)
```

Example of an unsorted array: [0 1 1 2 3 7 8 7 8 9]
Example of a sorted array: [0 1 1 2 3 7 7 8 8 9]
Total: compares=45 & swaps=1

Insertion Sort - A Possible Improvement

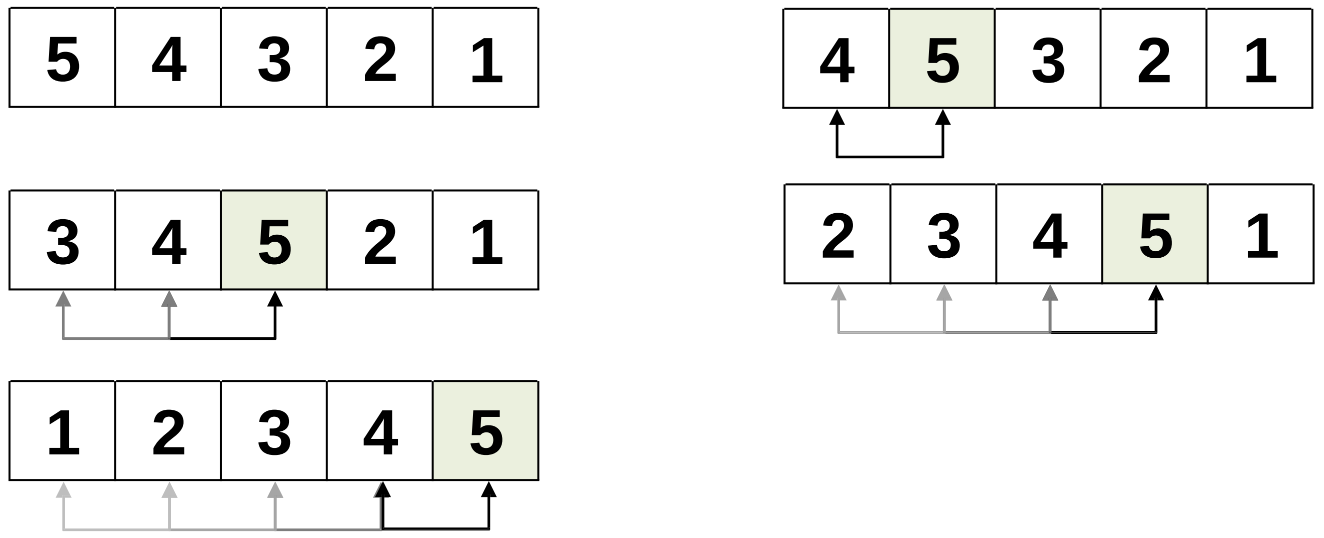
In this improvement, we have exploited the idea that the left part of A (from position 0 to position i) is already sorted. So, whenever one of the tests *a[j] < a[j- 1]* fails, there is no point in making also all the others. It means that the element to insert is already in its correct (sorted) position, and so the internal loop can terminate.

```
for all I from 0 to len(A)-1:
        for all j from i to 0, and when ((j>0) and (A[j] < A[j-1]))
                swap(A, j, j-1);
```

```
def sort_insertion_(a):
    for i in range(1, len(a)):
        j = i  # aka key
        while a[j] < a[j-1] and j > 0:
          swap(a, j, j-1)
          j = j - 1
```

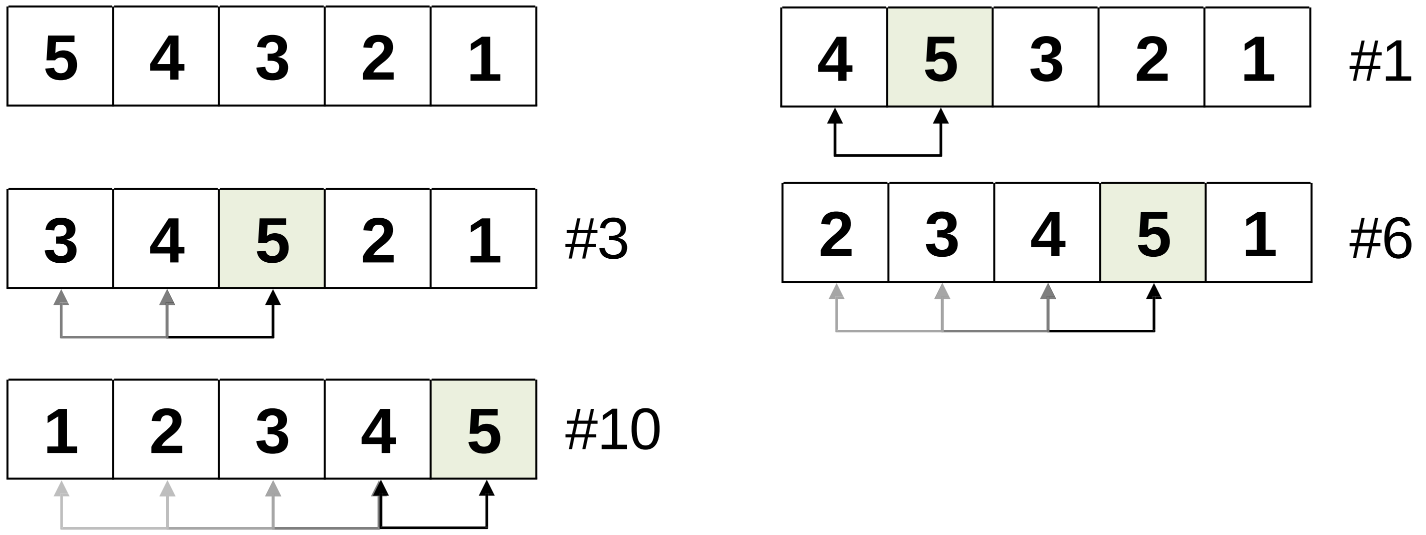
Insertion Sort - Example

Consider the worst case scenario for an array of size 5:



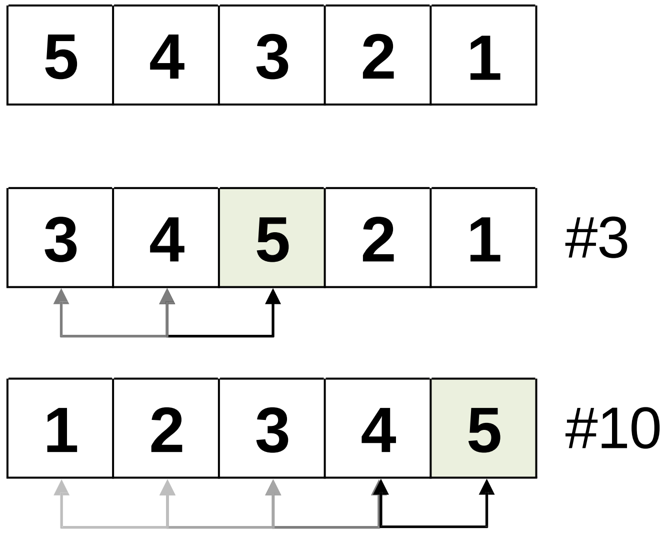
Insertion Sort - Example

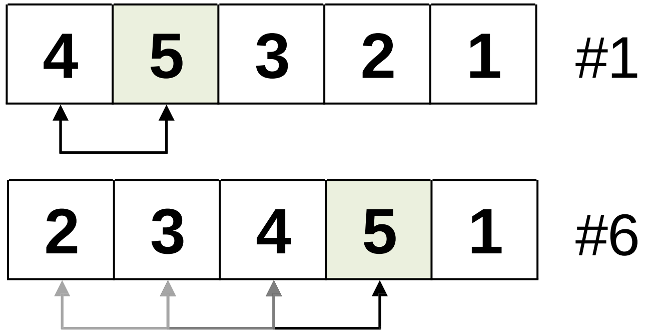
Let’s count the **number of swaps**:

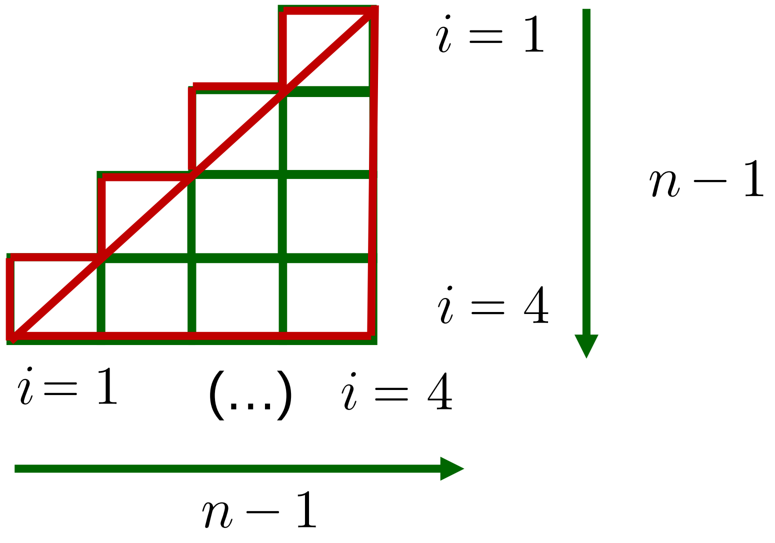


Insertion Sort - Example

Let’s represent the number of swaps graphically (using one *square* per swap):

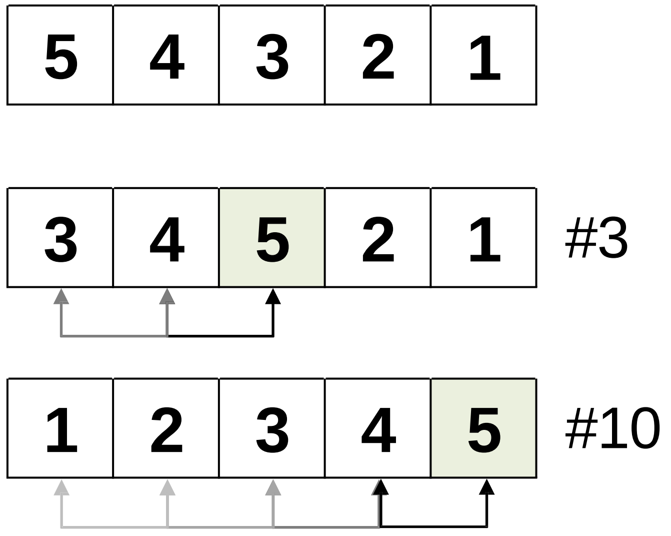


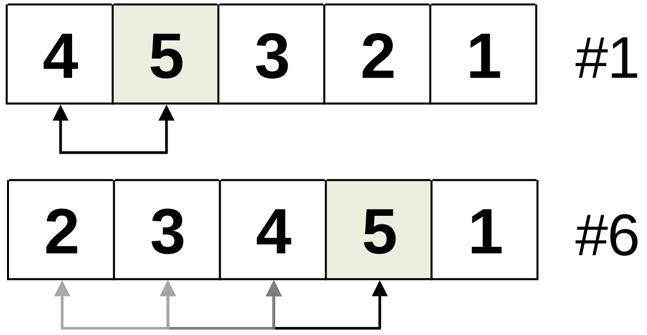


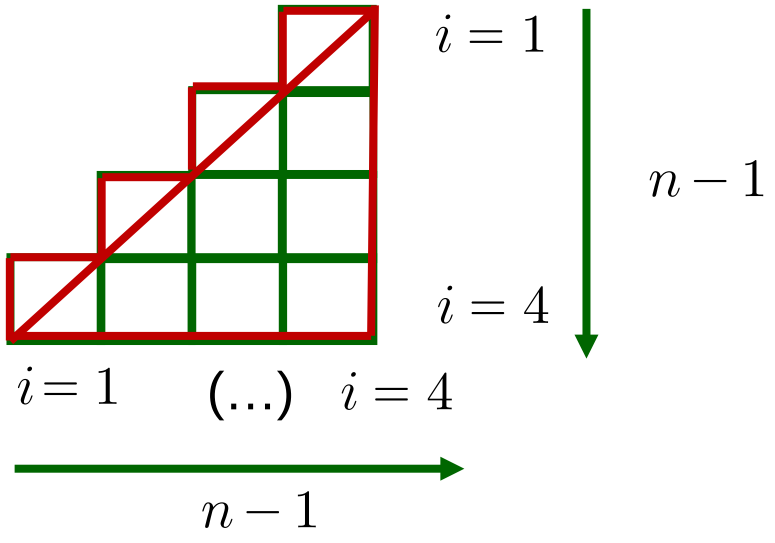


Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:



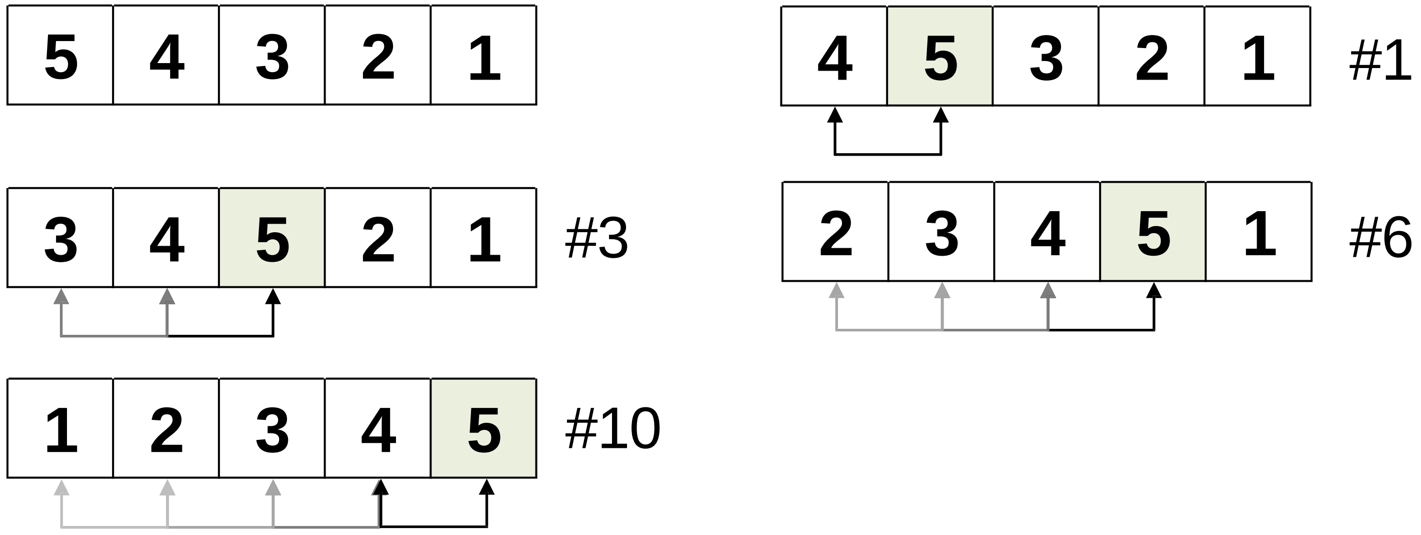


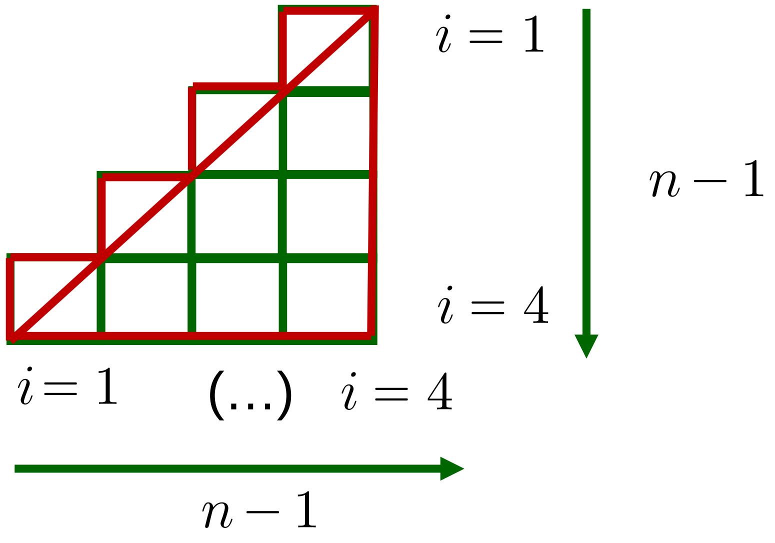


$$A = \frac{(n-1)(n-1)}{2} + (n-1)\frac{(1*1)}{2}$$

Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:

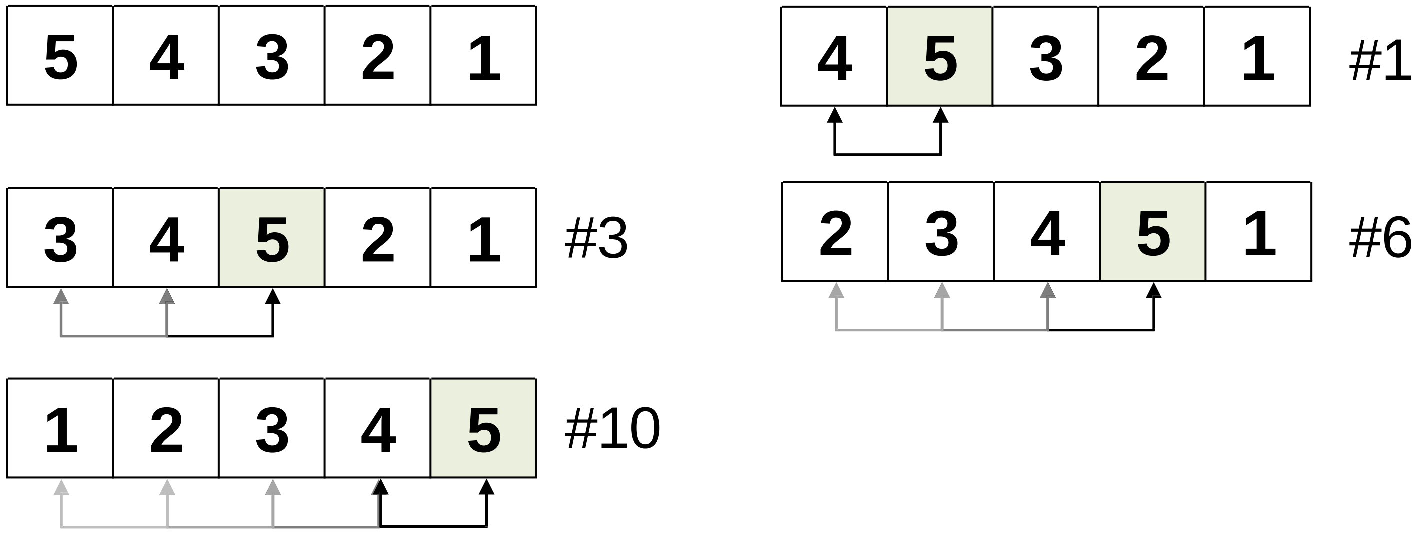


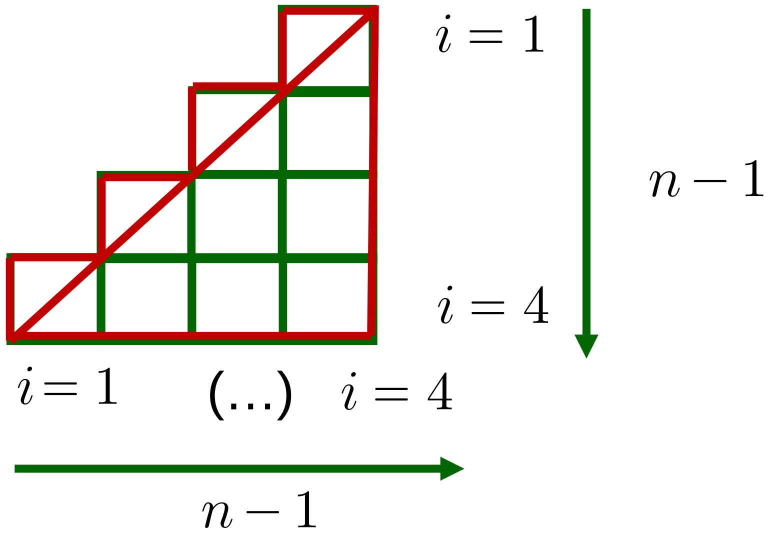


$$A = \frac{(n-1)(n-1) + (n-1)}{2}$$

Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:

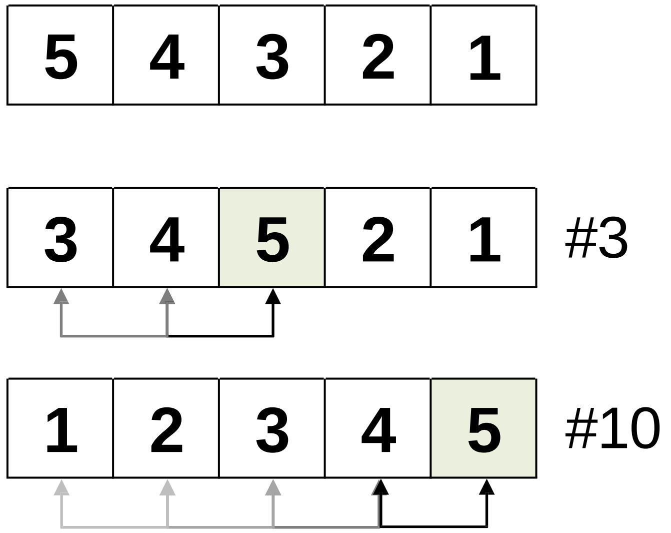


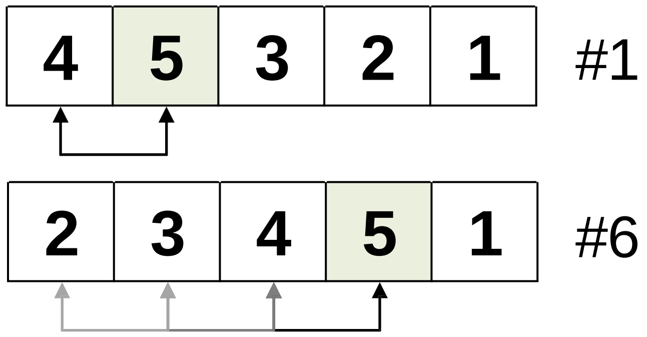


$$A = \frac{(n-1)((n-1)+1)}{2}$$

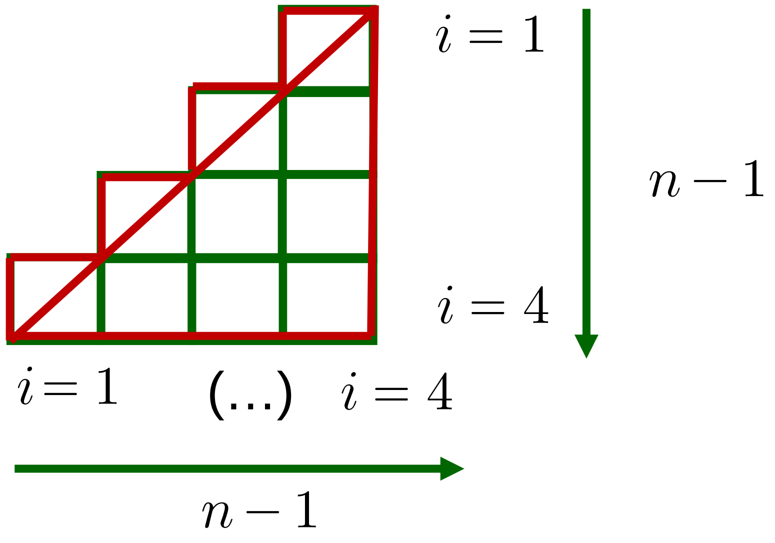
Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:



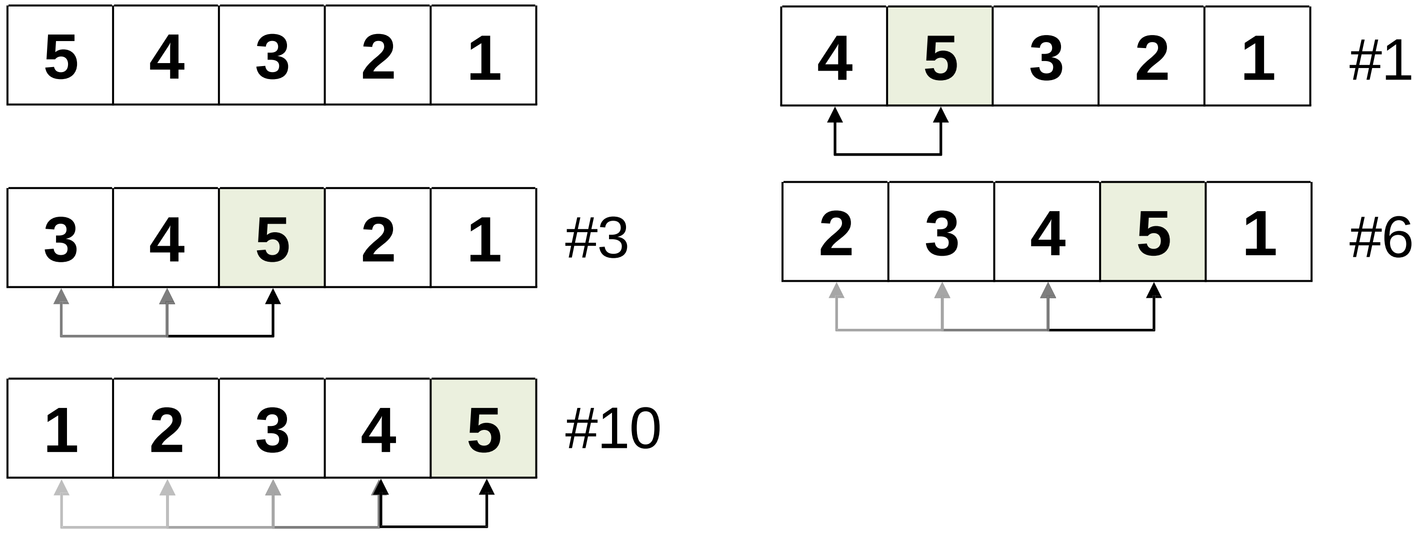


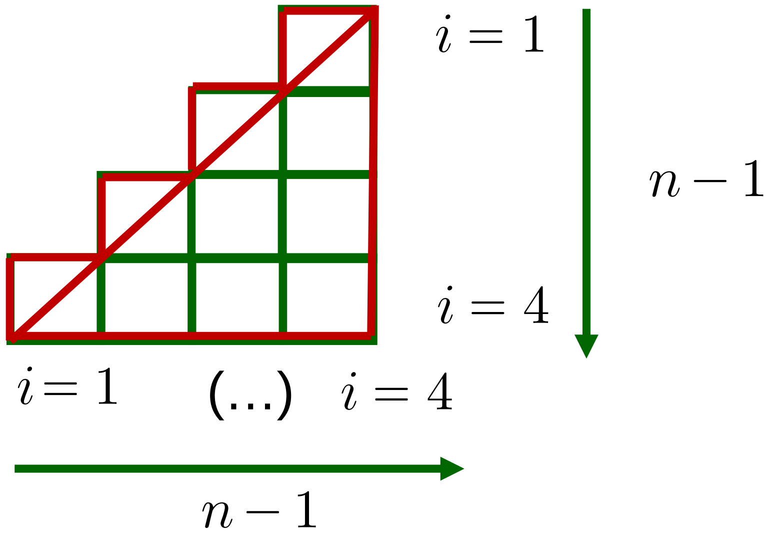
$$A = \frac{(n-1)(n-1+1)}{2}$$



Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:

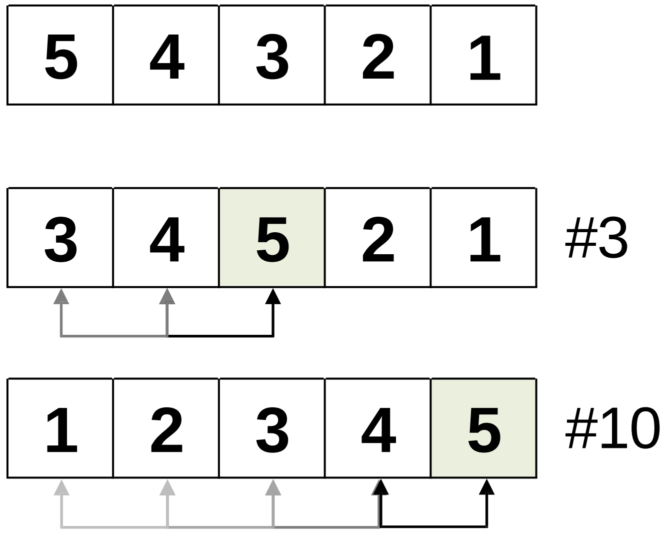


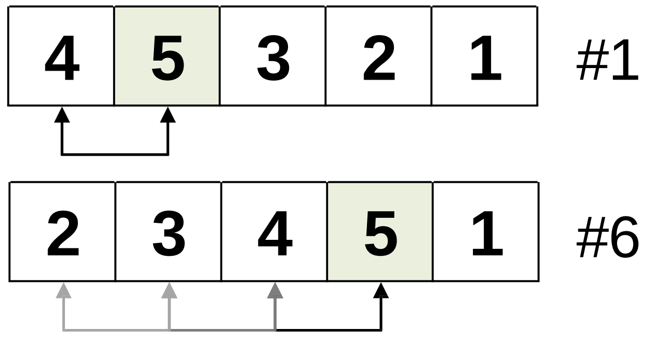


$$A = \frac{(n-1)(n)}{2}$$

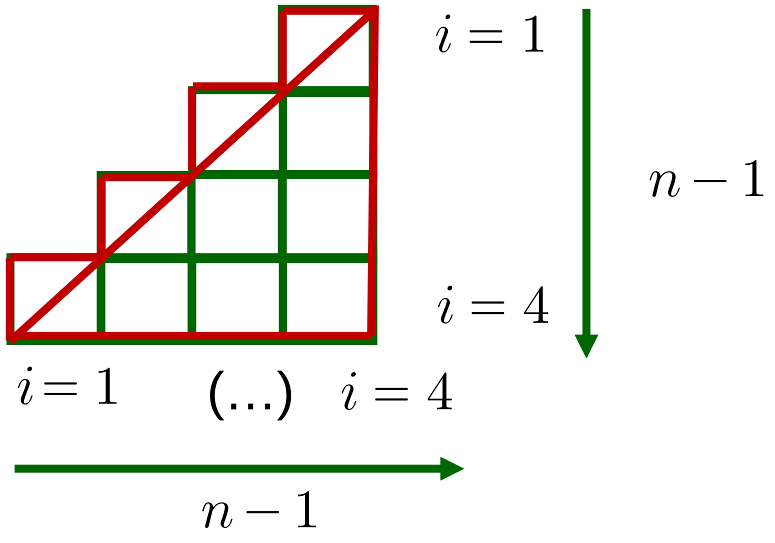
Insertion Sort - Example

The area of these *squares* represents the time complexity of insertion sort:





$$A = \frac{n(n-1)}{2} \rightarrow O(n^2)$$



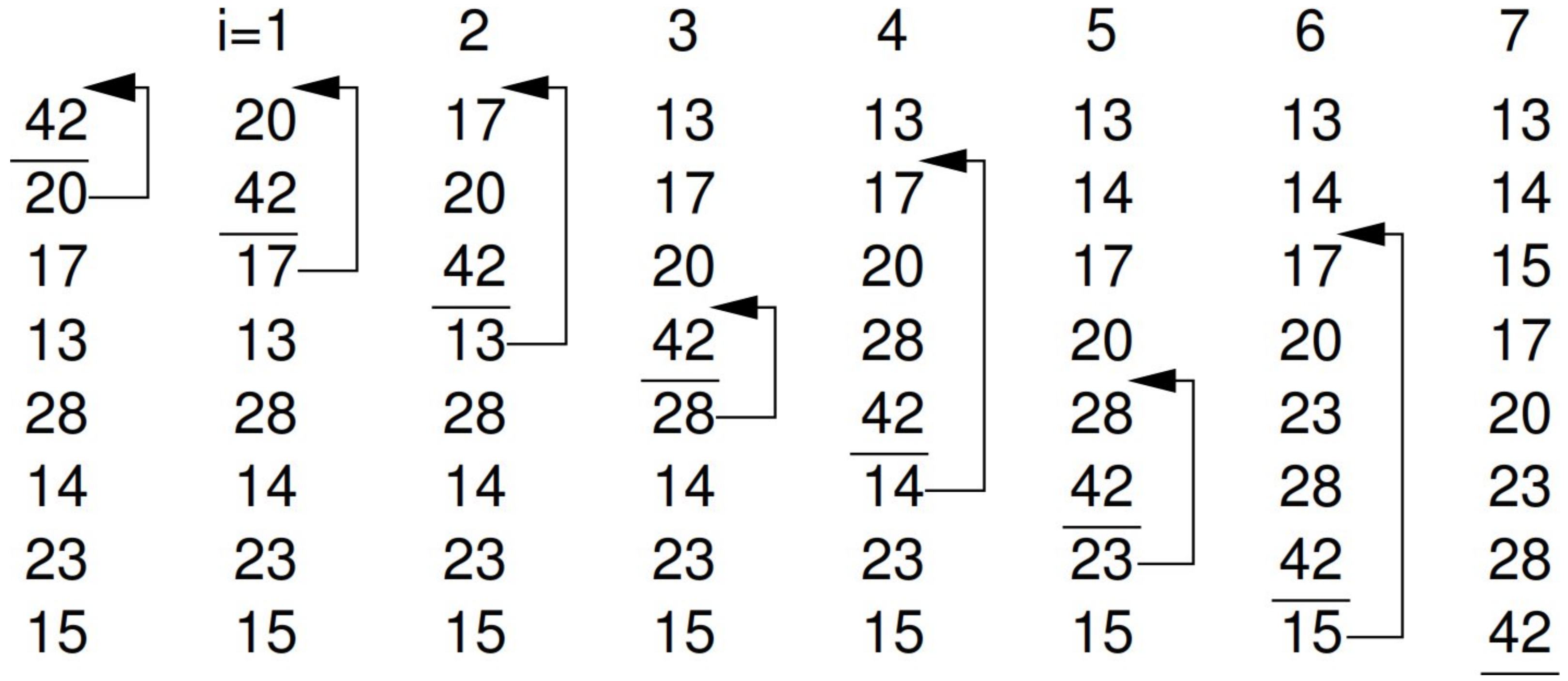
Insertion Sort - Example

We want to apply Insertion Sort to sort the following array:

42		13
20		14
17		15
13	⇒	17
28		20
14		23
23		28
15		42

Insertion Sort - Example

Here is the execution of the Insertion Sort, step by step. Each column shows the array after the iteration with the indicated value of i in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the upward motions of values through the array.



Insertion Sort - Computational Complexity

The body of Insertion Sort is made up of two nested **for** loops.

The outer **for** loop is executed **n - 1 times**.
The inner **for** loop is harder to analyze because the number of times a swap is executed depends on how many keys in positions 1 to i – 1 have a value less than that of the key in position i. In the ***worst case***, each record must make its way to the top of the array. This would occur if the keys are initially arranged from highest to lowest, in the reverse of sorted order. In this case, the number of comparisons will be one the first time through the for loop, two the second time, and so on.

Thus, the total number of comparisons will be:

$$\sum_{i=2}^{n} i = \Theta(n^2)$$

Why? See next slide...

Insertion Sort - Computational Complexity

In contrast, consider the ***best-case*** cost. This occurs when the values begin in sorted order from lowest to highest.

In this case, every pass through the inner for loop will fail immediately, and no values will be moved.

The total number of ***comparisons*** (*considering the improved version of the algorithm!*) will be ***n - 1***.

Thus, the cost for Insertion Sort in the best case is **Θ(n)**.

Insertion Sort - Computational Complexity

While the best case is significantly faster than the worst case, *the worst case is usually a more reliable indication of the “typical” running time*.

However, there are situations where we can expect the input to be in sorted or nearly sorted order.

One example is when an already sorted list is *“slightly disordered”*; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight.

Insertion Sort - Computational Complexity

What is the ***average-case*** cost of Insertion Sort?

When value i is processed, the number of times through the inner for loop depends on how far “out of order” the value is. In particular, the inner for loop executes a swap once for each value greater than the value i that appears in array positions 0 through i-1.

For example, in the leftmost column of the previous example, the value 15 is preceded by five values greater than 15.

Each such occurrence is called an inversion. The number of inversions (i.e., the number of values greater than a given value that occur prior to it in the array) will determine the number of comparisons and swaps that must take place.

We need to determine what the *average number of inversions* will be for the record in position i.

Insertion Sort - Computational Complexity

We expect on average that *half of the values in the first i – 1 array positions be greater than the value at position i*.

Thus, the average case should be about *half the cost of the worst case*, which is still **$\Theta(n^2)$**.

So, *the average case is no better than the worst case* in asymptotic complexity.

Insertion Sort - Computational Complexity

Counting comparisons or swaps yields similar results.

In fact, each time through the inner for loop yields both a comparison and a swap, except the last (i.e., the comparison that fails the inner **for** loop's test), which has no swap.

Thus, the number of swaps for the entire sort operation is n - 1 less than the number of comparisons.

This is 0 in the best case, and $\Theta(n^2)$ in the average and worst cases.

Merge Sort

A natural approach to problem solving is ***divide and conquer***.

In terms of sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow.

A simple way to do this would be to ***split the list in half***, ***sort the halves***, and then ***merge*** the sorted halves together. This is the idea behind Mergesort.

Merge Sort

A pseudocode sketch of Mergesort is as follows:

```
function MergeSort
        input: an array of n non-sorted numbers called A
        output: the array A sorted

   if the number of elements in A is smaller or equal to 1
   then terminate;     // in this case we have nothing to sort
                        //  but this piece of code MUST be here!
                        //  why?
   else
        •  Put in an array A1 the first half of the elements in A;
        •  Put in an array A2 the second half of the elements in A;
        •  Sort A1 applying MergeSort;
        •  Sort A2 applying MergeSort;
        •  Join together A1 and A2 (that are now both sorted) using a
           function called Merge;
```

Merge Sort

A pseudocode sketch of Mergesort is as follows:

```
function MergeSort
            input: an array of n non-sorted numbers called A
            output: the array A sorted

     if the number of elements in A is smaller or equal to 1
     then terminate;      // in this case we have nothing to sort
                          //  but this piece of code MUST be here!
                          //  why?
    else
          •   Put in an array A1 the first half of the elements in A;
          •   Put in an array A2 the second half of the elements in A;
          •   Merge (MergeSort(A1), MergeSort(A2));

                      //  Of course we still have to define the function Merge
                      //  this is done in the next slide
```

The Function Merge

Before discussing how to implement Mergesort, we will first examine the **Merge** function.

Merging two sorted sublists is quite simple. The function merge examines the first element of each sublist and picks the smaller value as the smallest element overall. This smaller value is removed from its sublist and placed into the output list.

Merging continues in this way, *comparing the front elements of the sublists and continually appending the smaller* to the output list until no more input elements remain.

Merge Sort

The are many ways to implement the Merge Sort.

In particular, there are many ways of:

- splitting array A into two halves
- implementing function Merge

About how to split array A into two halves, it is convenient not to create supplementary (and, in this case, useless) structures.
It is better to use ***indices***. The Merge Sort method will thus receive, as parameters, the array to be sorted and two indices: the *beginning and the end of the portion of the array to be sorted*.
It's objective will be to sort the portion included between these two indices.
Then, we will call the Merge Sort method passing 0 and the index of the last lement (and in this way, all the vector will be sorted).

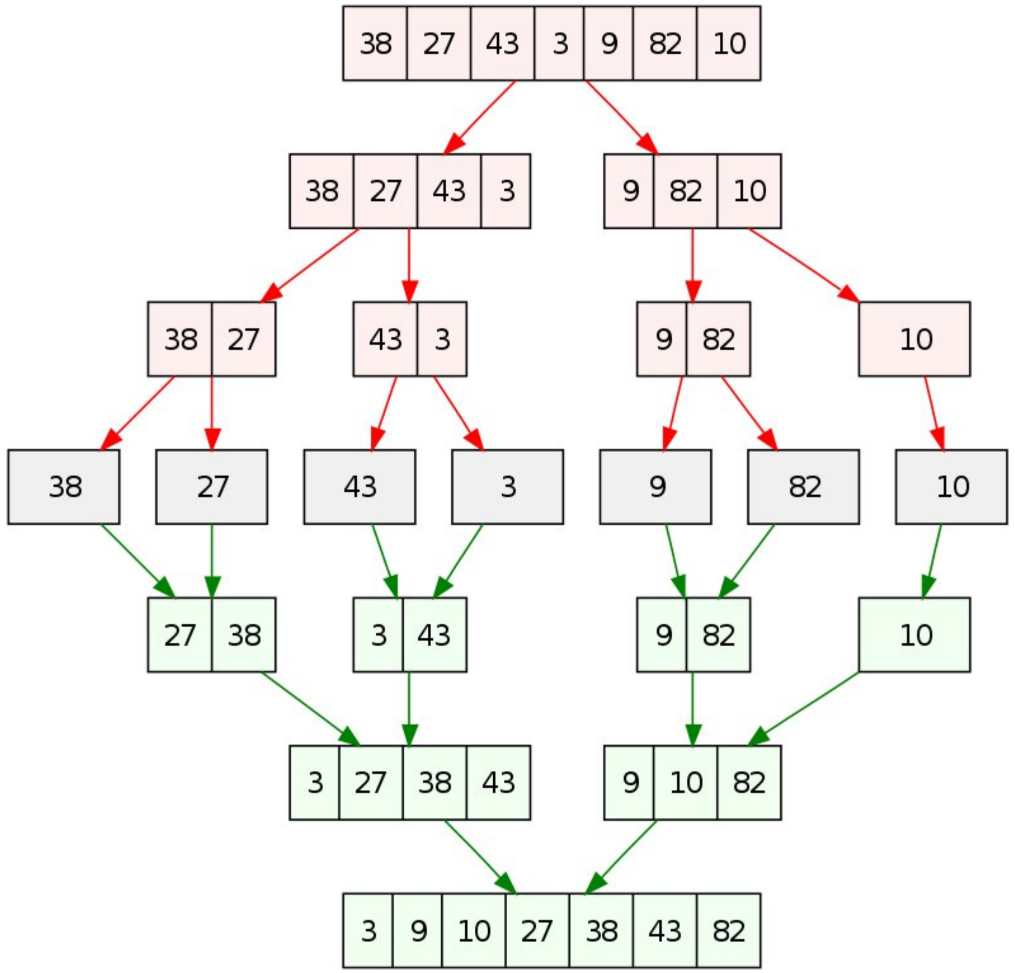
Merge Sort - “High level” Example

An illustration of Mergesort. The rows in red show the recursive division of the array into sub-arrays until the size is 1.

The grey row shows the 7 values comprising the array as individual sub-arrays (each sorted within itself).

The rows in green are created by merging the sub- arrays in a sorted manner until one array remains.

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/600px-Merge_sort_algorithm_diagram.svg.png



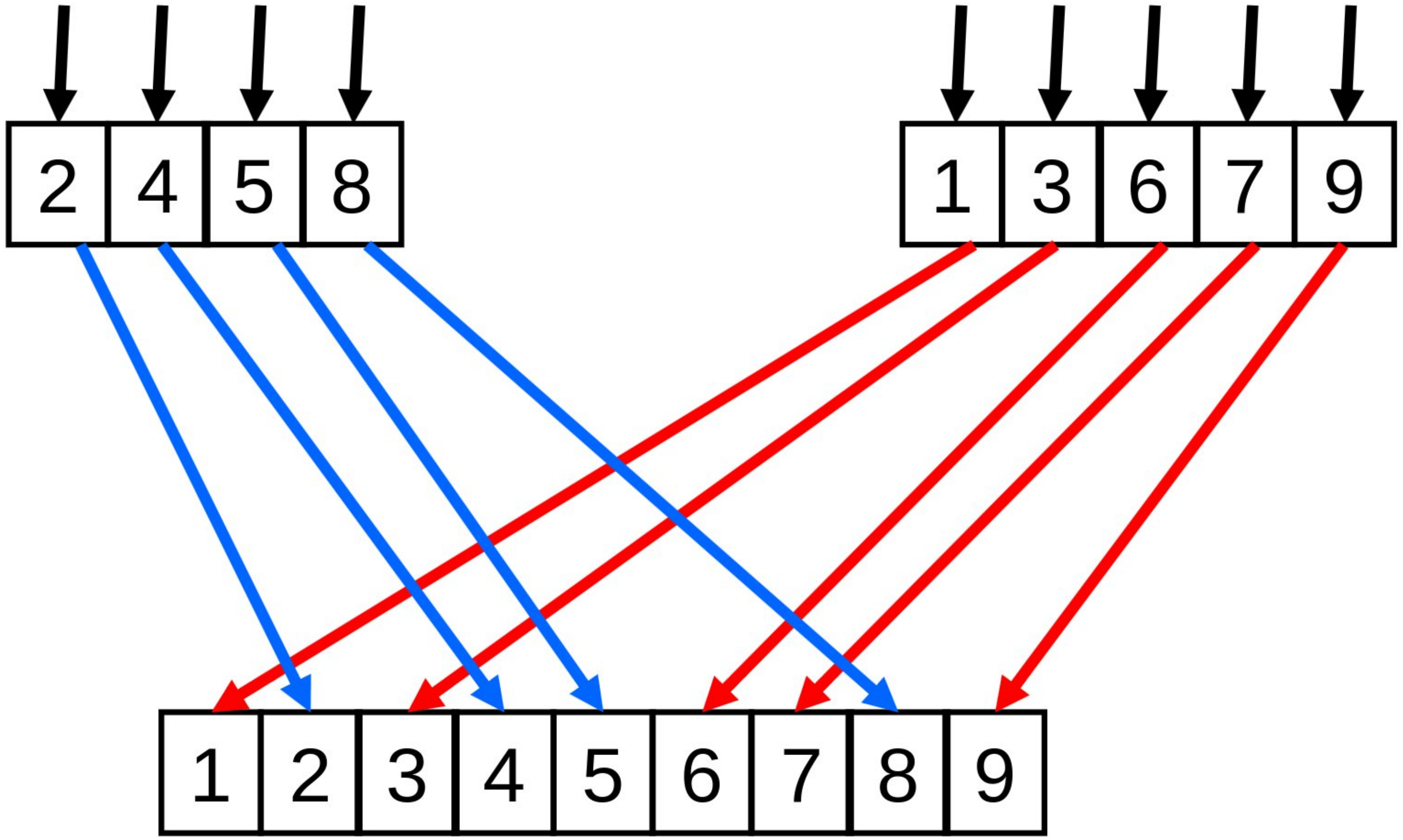
Merge Sort

```
def sort_merge(a):
    if len(a) == 1:
        return a

    idx_half = len(a) // 2
    a_left = merge_sort(a[: idx_half])
    a_right = merge_sort(a[idx_half:])
    return merge(a_left, a_right)
```

Mergesort is recursively called until subarrays of size 1 have been created, requiring *log n* levels of recursion.

The Merge Function - Idea



The Merge Function

```
def merge(a_left, a_right):
  a_sorted = []
  i = j = 0
  while i < len(a_left) and j < len(a_right):
    if a_left[i] < a_right[j]:
      a_sorted.append(a_left[i])
      i += 1
    else:
      a_sorted.append(a_right[j])
      j += 1

  if i < len(a_left):
    a_sorted.extend(a_left[i:])
  if j < len(a_right):
    a_sorted.extend(a_right[j:])
  return a_sorted
```

Function extend, example:

```
L1 = [1, 4]
L2 = [2, 3, 5]
L.extend(L2)
print(L)

# Prints: [1, 4, 2, 3, 5]
```

This function merges two sorted subvectors into a sorted vector (more efficient versions of this method can exist).

Binary Search - Computational Complexity

A list can be divided into lists of size 1 by repeatedly splitting in $\boldsymbol{\Theta}(\log \boldsymbol{n})$ time.

Recall that the worst-case complexity for the binary search is precisely $\boldsymbol{\Theta}(\log \boldsymbol{n})$: virtually splitting a given array of size *n* into halves until reaching the maximum recursion depth (when the array is size 1).

Binary Search - computational complexity

Recall the rationale regarding binary search

Number of step	**Length of the sequence**	**Number of comparisons done so far**
1	$N/2^0$	1
2	$N/2^1$	2
3	$N/2^2$	3
4	$N/2^3$	4
5	$N/2^4$	5
…	…	…
X	**1**	**X**

$$\boxed{N/(2^{x-1})}$$

Binary Search - computational complexity

Let us solve the equation of the previous slide:

$1 = N / 2^{X-1}$

$2^{X-1} = N$

Logarithm rule:

$$b^x = n \rightarrow \log_b n = x$$

$\mathbf{\log_2 N = X - 1}$

$\mathbf{X = \log_2 N + 1}$

Binary Search - computational complexity

So, in the **worst case** (the case in which the searched element is not in the sequence), we do X = $\log_2 N + 1$ comparisons.

In the **average case**, we can expect to make half of these comparisons: X = $(\log_2 N + 1) / 2$

In *both cases*, the complexity of the binary search is:

$$\boldsymbol{\Theta}(\log \boldsymbol{n})$$

Remark that in the best case (the case in which the searched element is the central one of the sequence) we just do one comparison, so the computational complexity is constant: $\boldsymbol{\Theta}(\mathbf{1})$

Merge Sort - Computational Complexity

Thus, **the depth of the recursion is *log n*** for *n* elements in the array *a*. The first level of recursion can be thought of as working on one array of size *n*, the next level working on two arrays of size *n/2*, the next on four arrays of size *n/4*, and so on. The bottom of the recursion has *n* arrays of size 1.

Thus, *n* arrays of size 1 are merged (requiring n total steps), *n/2* arrays of size 2 (again requiring n total steps), *n/4* arrays of size 4, and so on.

```
def sort_merge(a):
  if len(a) == 1:
    return a

  idx_half = len(a) // 2
  a_left = merge_sort(a[: idx_half])
  a_right = merge_sort(a[idx_half:])
  return merge(a_left, a_right)
```

Merge Sort - Computational Complexity

The merging part takes time T(m) where m is the total length of the two subarrays being merged. The array to be sorted is repeatedly split in half until subarrays of size 1 are reached, at which time they are merged to be of size 2, these merged to subarrays of size 4, and so on. **At each of the *log n* levels of recursion, Θ(*n*) work is done.**

```
def merge(a_left, a_right):
  a_sorted = []
  i = j = 0
  while i < len(a_left) and j < len(a_right):
    if a_left[i] < a_right[j]:
      a_sorted.append(a_left[i])
      i += 1
    else:
      a_sorted.append(a_right[j])
      j += 1

  if i < len(a_left):
    a_sorted.extend(a_left[i:])
  if j < len(a_right):
    a_sorted.extend(a_right[j:])
  return a_sorted
```

Merge Sort - Computational Complexity

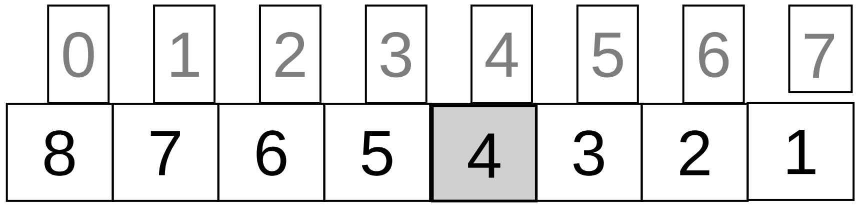
At each of the *log n* levels of recursion, Θ*(n)* work is done, for a total cost of:

$$\Theta(n \log n)$$

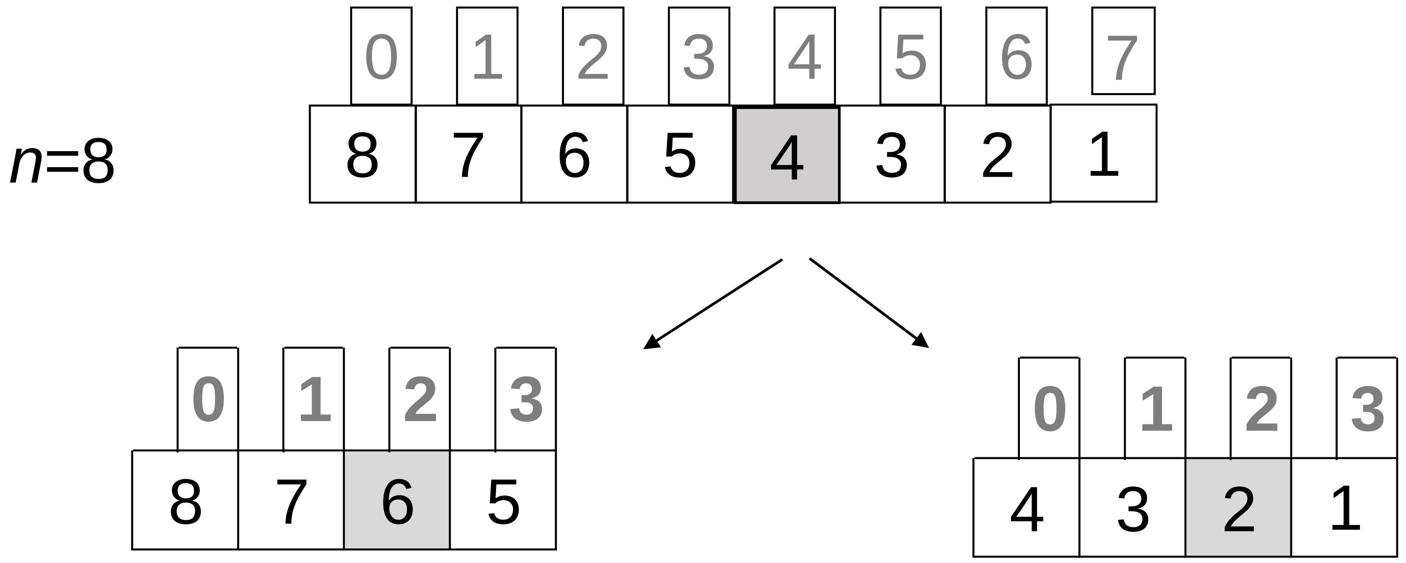
This cost is unaffected by the relative order of the values being sorted, thus this analysis holds for the *best, average, and worst cases*.

Merge Sort - example

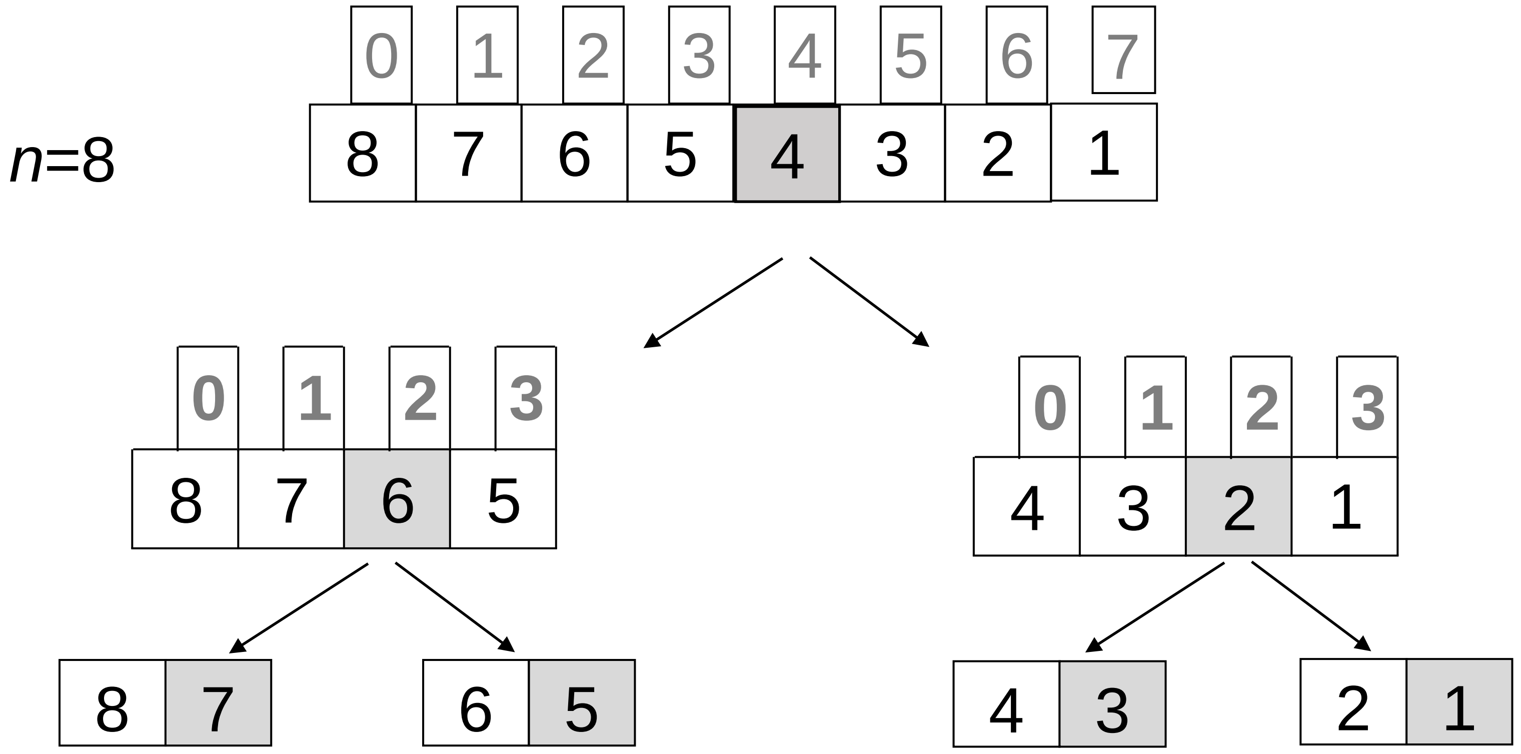
n=8



Merge Sort - example

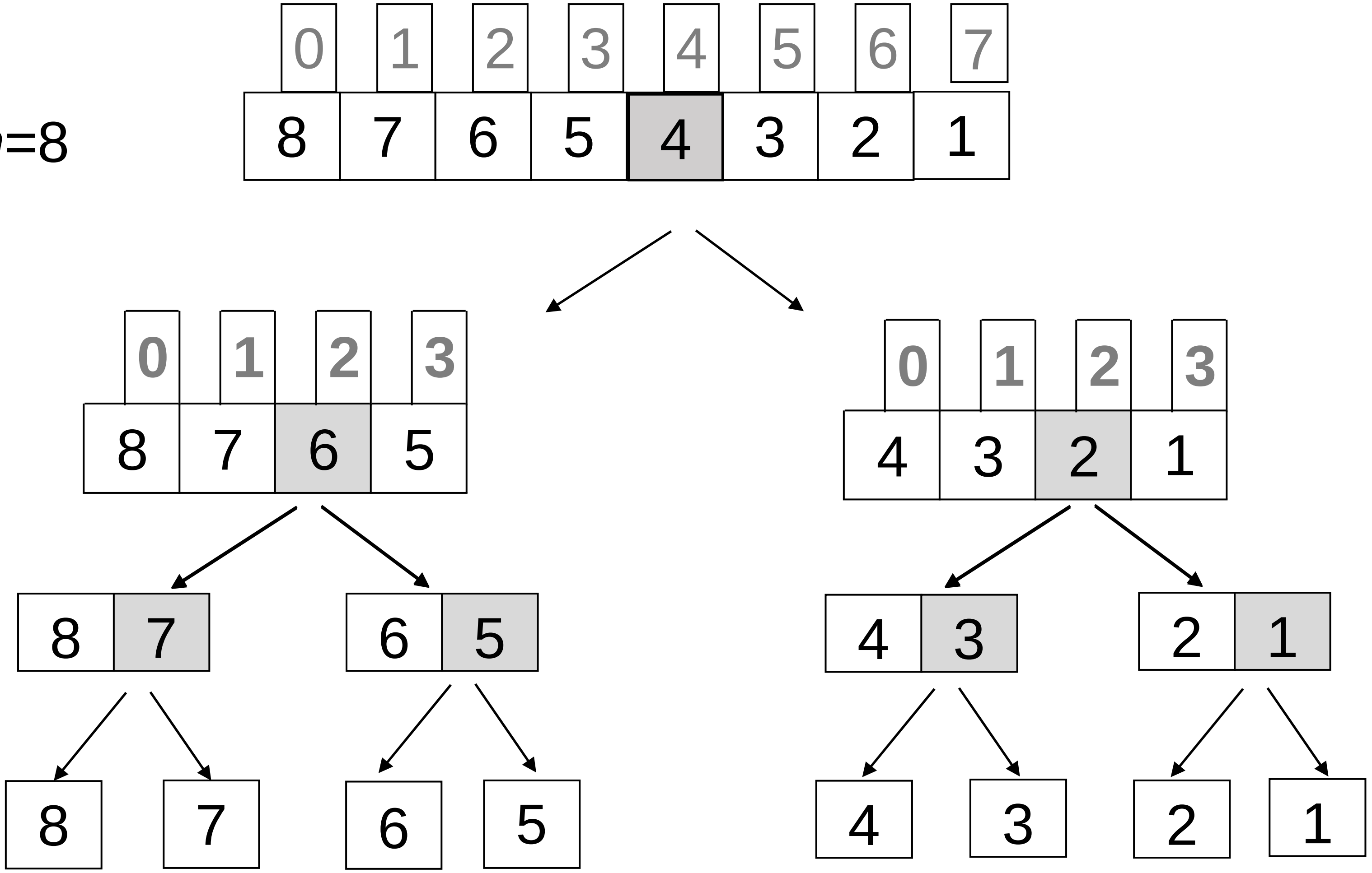


Merge Sort - example

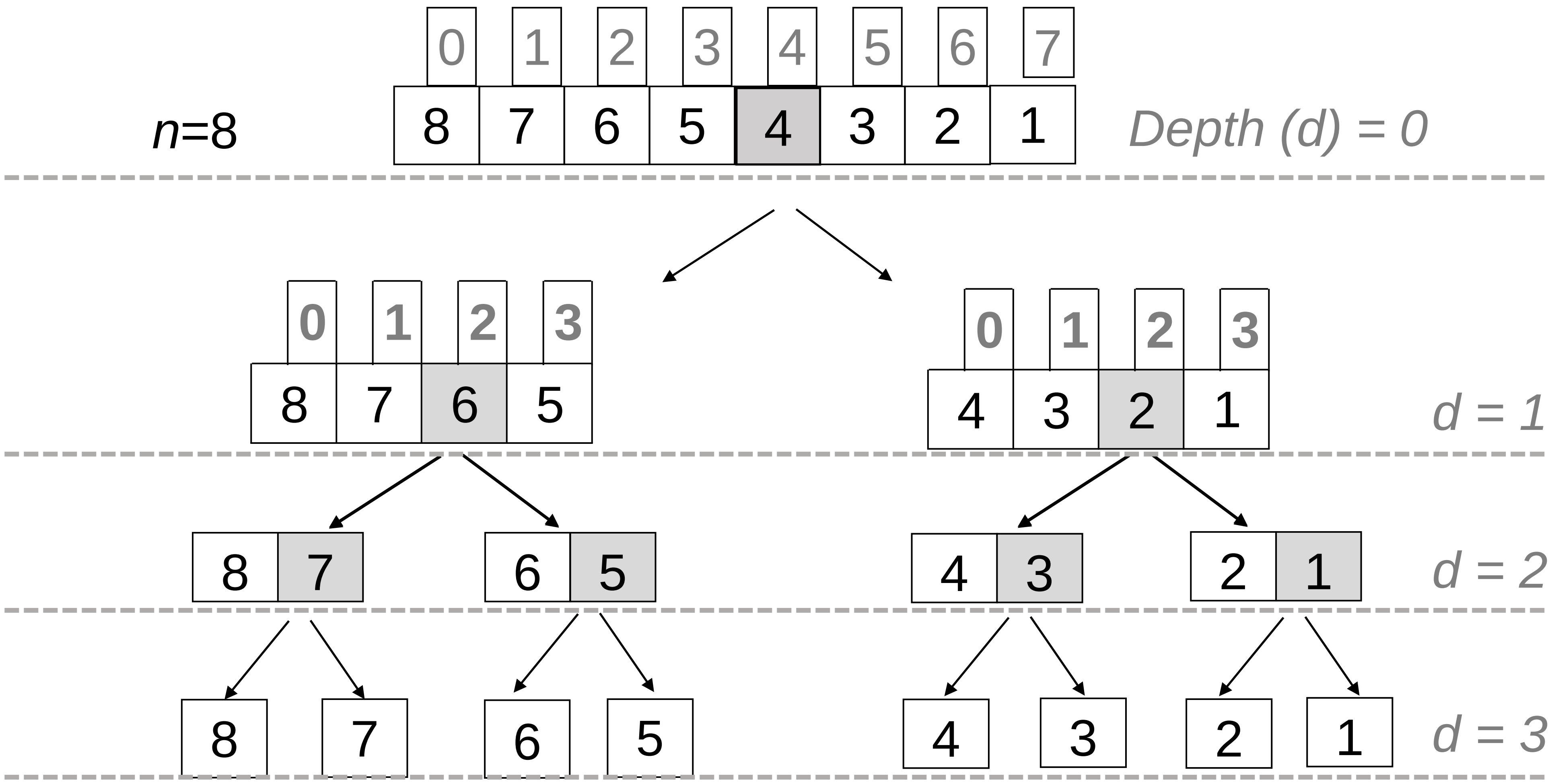


Merge Sort - example

n=8

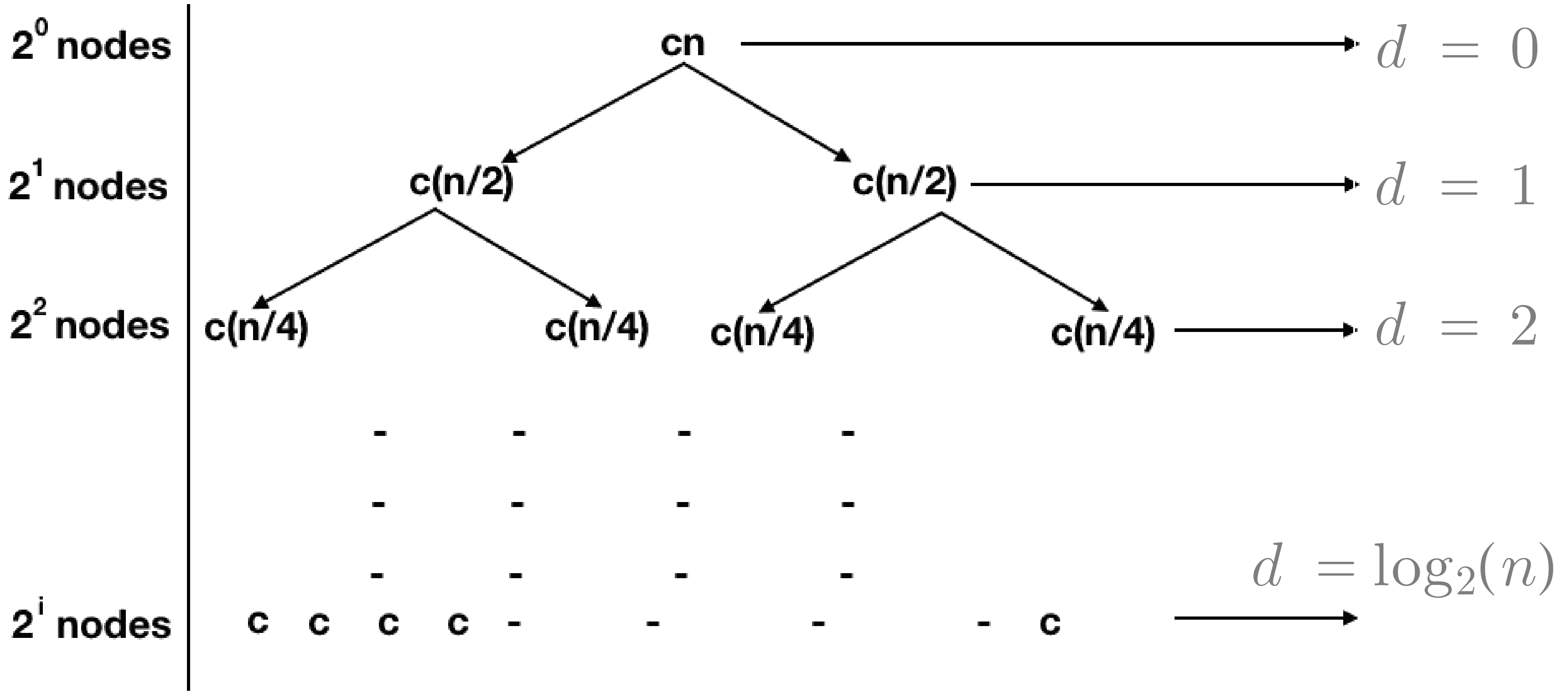


Merge Sort - example



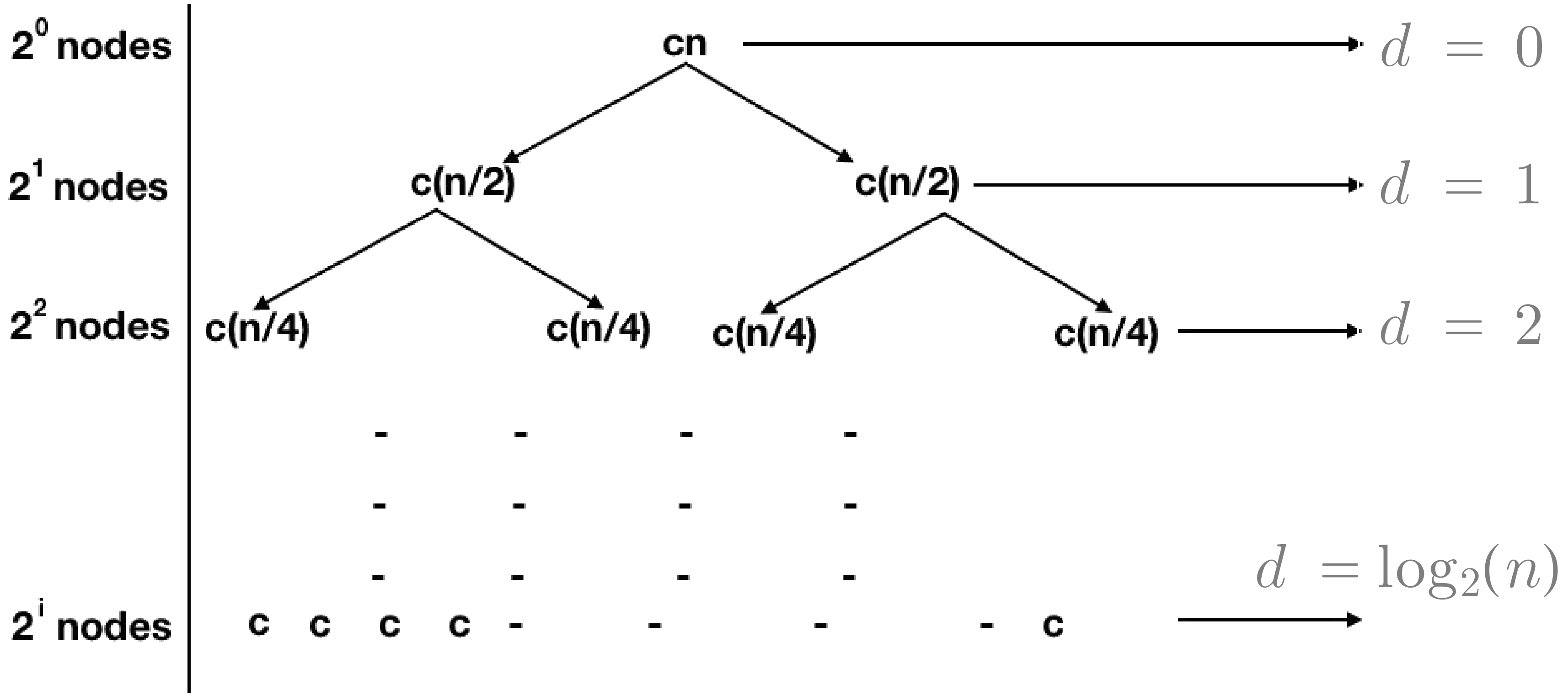
Merge Sort - example

Given an *n-sized* array, the depth of the recursive call tree for the merge sort algorithm is given by $\log_2(n)$. **For $n = 8$, $\log_2(8) = 3$** .



Merge Sort - example

In this sense, we converted the recurrence relations into a *tree.* **The computational effort can be seen as the sum of the costs at all the levels of the tree's depth across all the branches.**



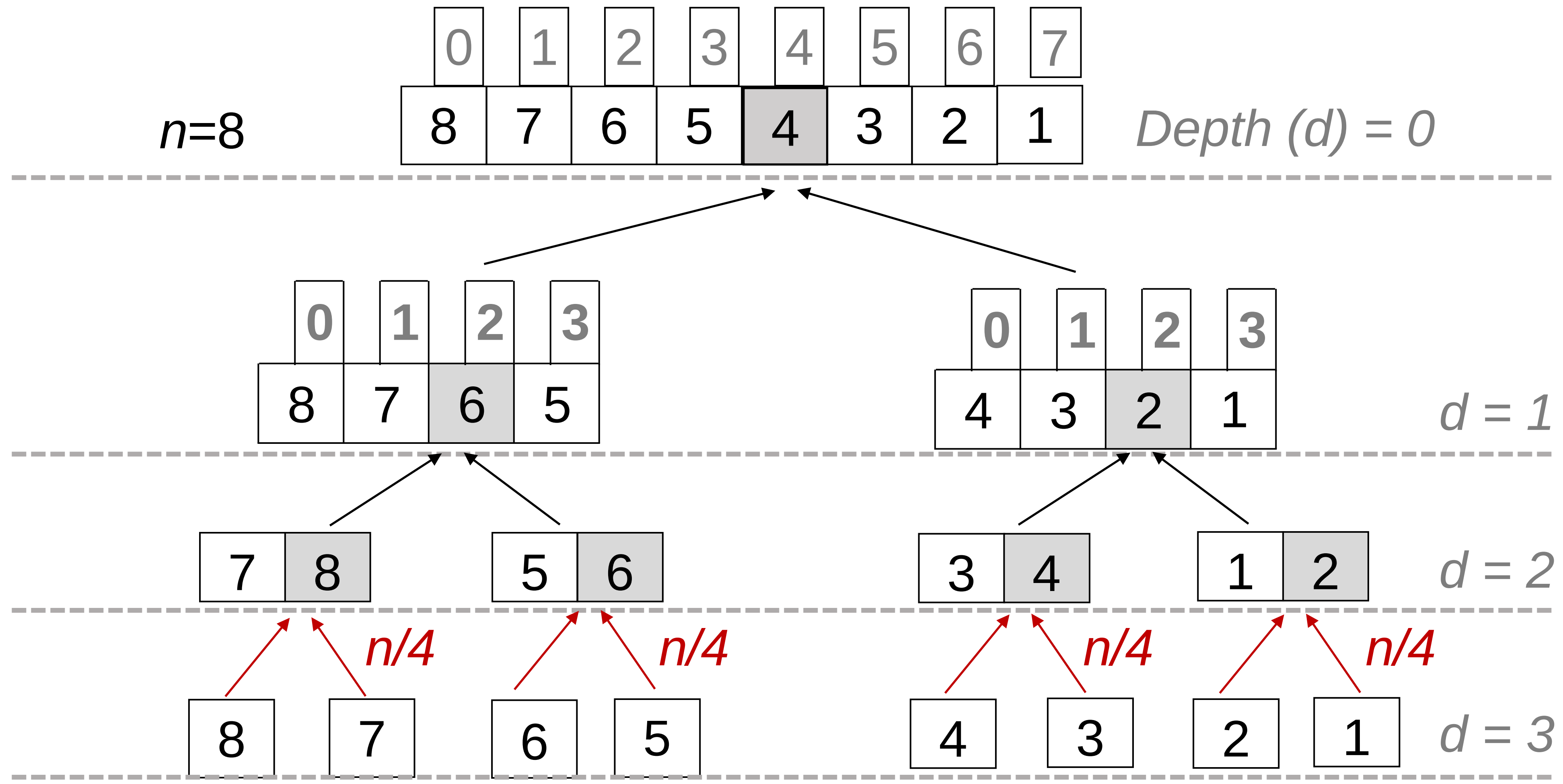
Merge Sort - Computational Complexity

The merging part takes time T(m) where m is the total length of the two subarrays being merged. The array to be sorted is repeatedly split in half until subarrays of size 1 are reached, at which time they are merged to be of size 2, these merged to subarrays of size 4, and so on. **At each of the *log n* levels of recursion, Θ(*n*) work is done.**

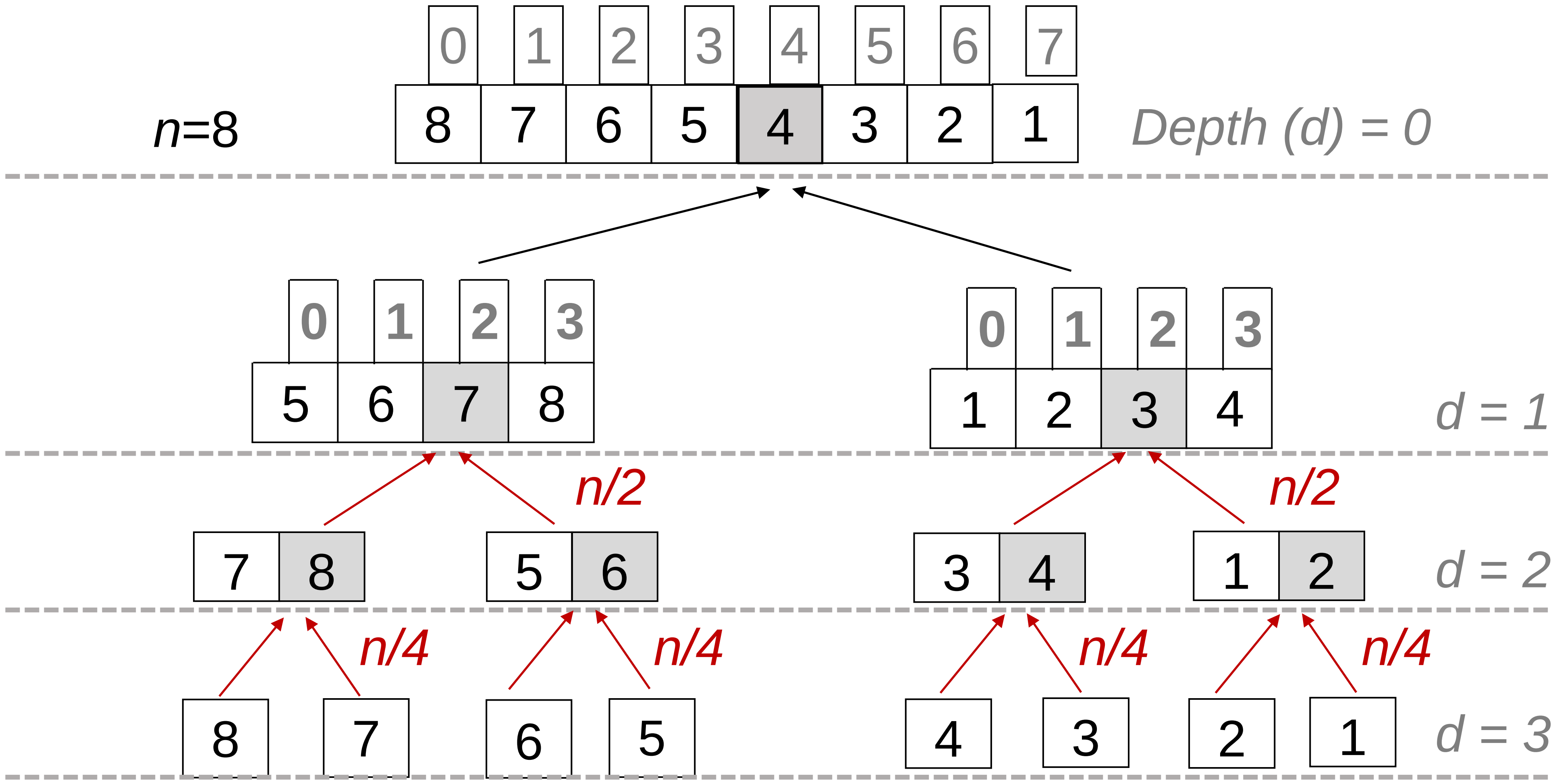
```
def merge(a_left, a_right):
  a_sorted = []
  i = j = 0
  while i < len(a_left) and j < len(a_right):
    if a_left[i] < a_right[j]:
      a_sorted.append(a_left[i])
      i += 1
    else:
      a_sorted.append(a_right[j])
      j += 1

  if i < len(a_left):
    a_sorted.extend(a_left[i:])
  if j < len(a_right):
    a_sorted.extend(a_right[j:])
  return a_sorted
```

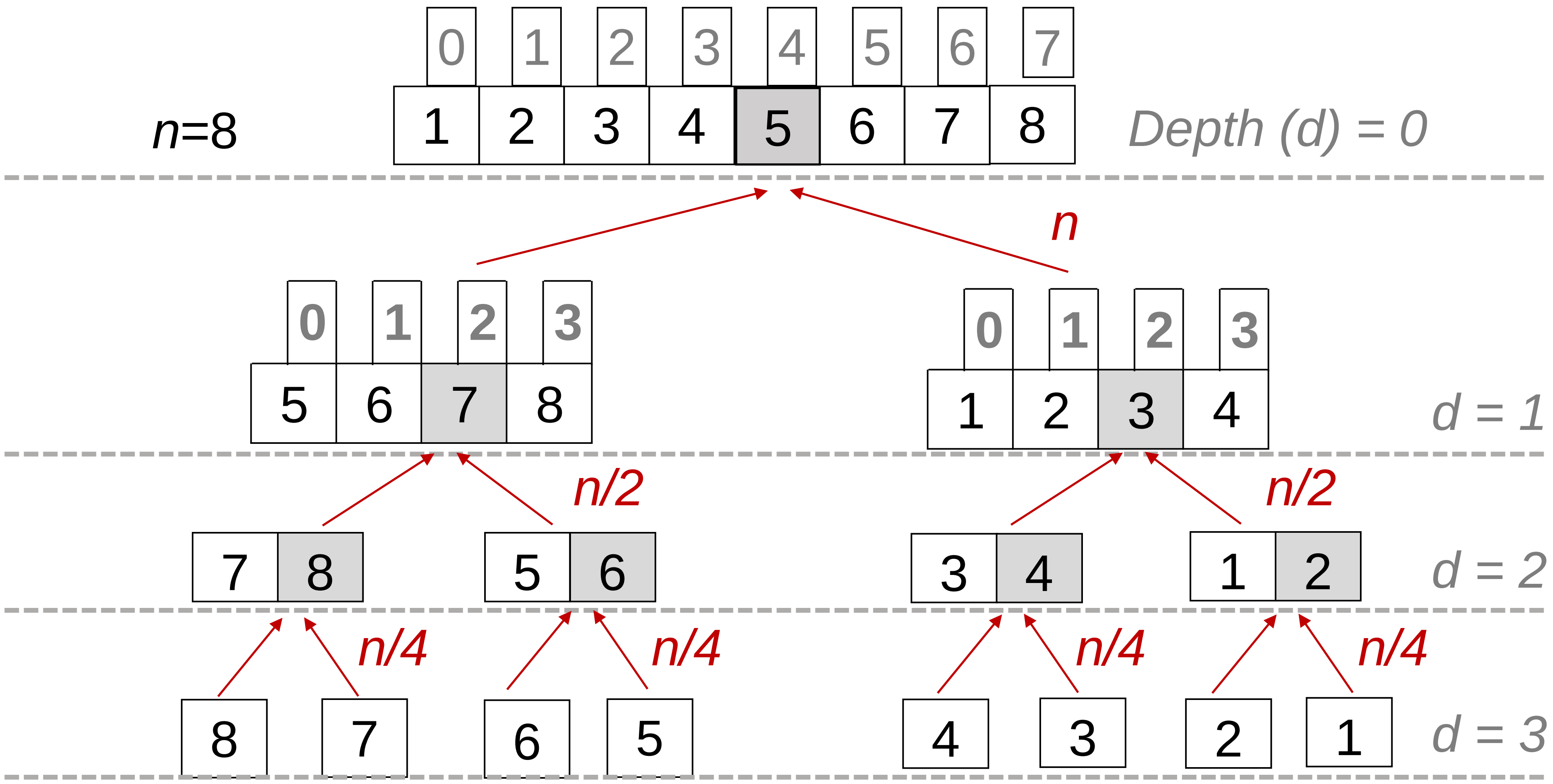
Merge Sort - example



Merge Sort - example

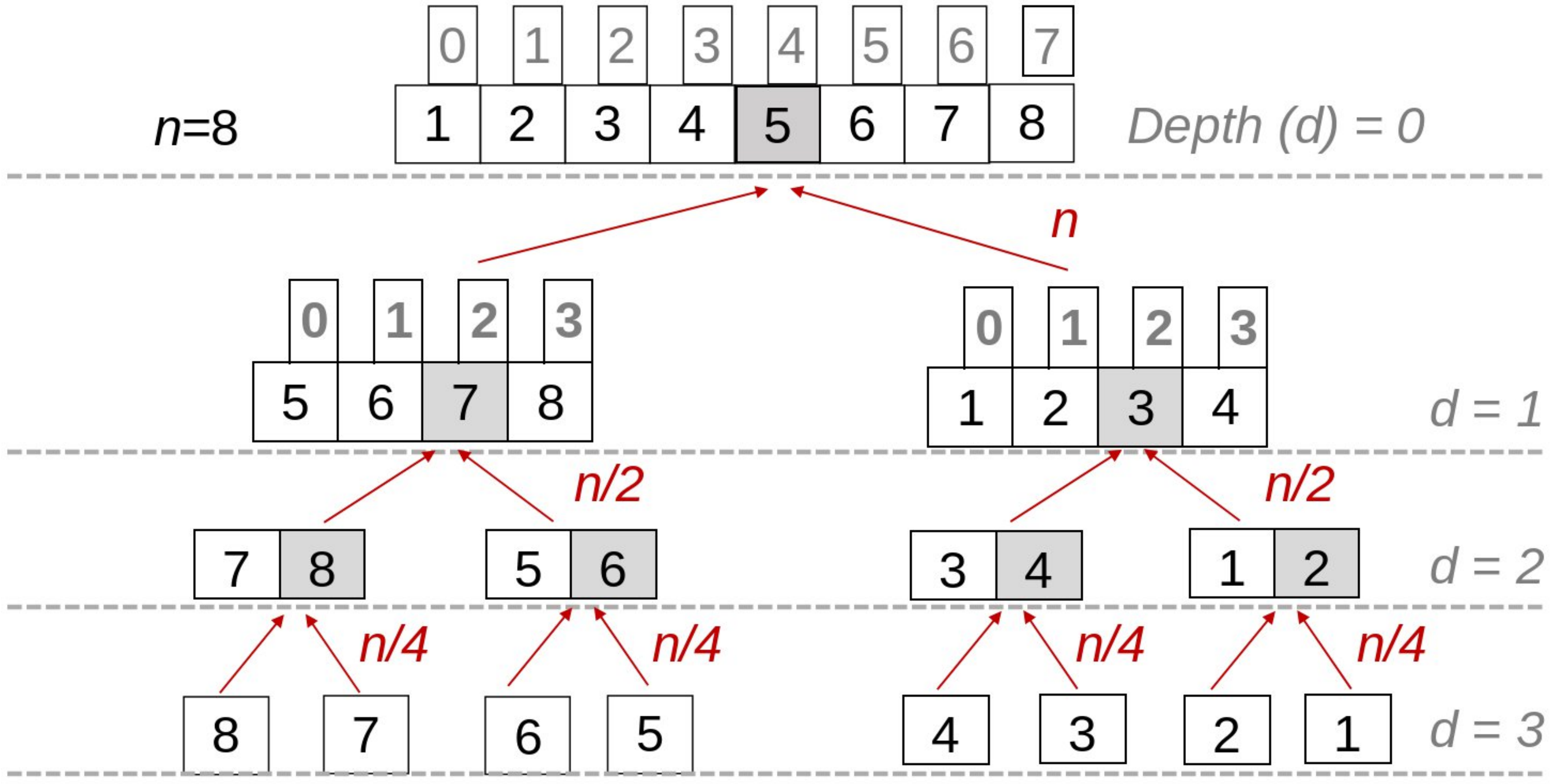


Merge Sort - example



Merge Sort - example

Summation of the costs at all the levels of the tree's depth: **n + 2(n/2) + 4(n/4) = 3n,** where $3 = \log_2(8)$. In general terms: $O\left(n \log_2 n\right)$



Quick Sort

While Merge Sort uses the most obvious form of divide and conquer (split the list in half then sort the halves), it is not the only way that we can break down the sorting problem.

And we saw that doing the merge step for Mergesort when using an array implementation is not so easy.

So perhaps a different divide and conquer strategy might turn out to be more efficient and/or simple?

Quick Sort

Quicksort is aptly named because, when properly implemented, it is among the fastest known general-purpose in-memory sorting algorithm in the average case.

Although the average runtime is equivalent to that of the merge sort, quick sort *does not require the extra array(s)*, so it is more space efficient.

Quicksort is widely used and is typically the default algorithm implemented in many libraries, such as numpy.sort()

Interestingly, Quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications (covered later).

Quick Sort

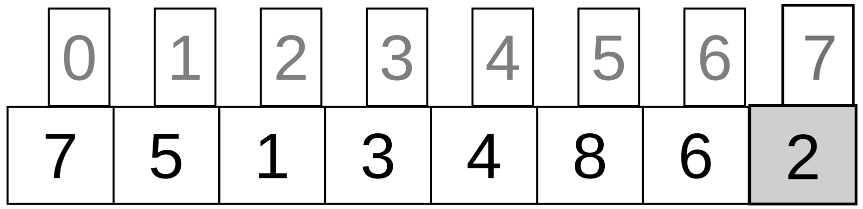
Quicksort *first selects a value called the **pivot***. The records are then rearranged in such a way that *the h values less than the pivot are placed in the first, or leftmost, h positions in the array*, *and the values greater than or equal to the pivot are placed in the last, or rightmost, n - h positions*. This is called a partition of the array.

The values placed in a given partition ***need not (and typically will not) be sorted with respect to each other***. All that is required is that all values end up in the correct partition.

The pivot value itself is placed in position *h* (the *correct/right* position if the array was sorted). ***Quicksort then proceeds to sort the resulting subarrays*** now on either side of the pivot, one of size *h* and the other of size *n - h - 1*.

How are these values sorted? A: using Quicksort on the resulting subarrays would (via *recursion*)

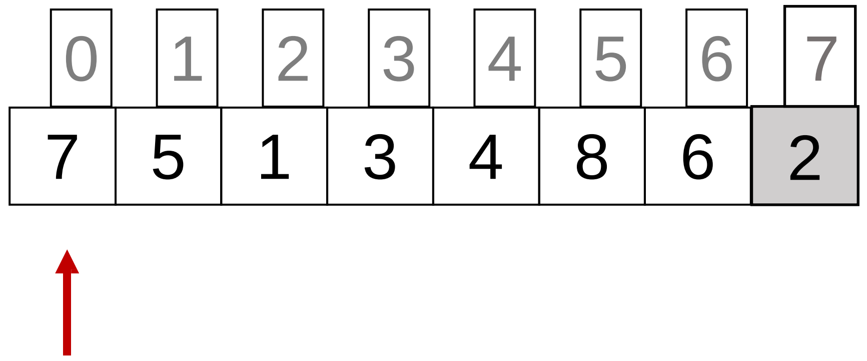
Quick Sort - Example



The value 2 was selected as the *pivot*. The next step consists of rearranging the array in terms of the *pivot*, so that values smaller than the *pivot* are placed on the left, and larger elements are placed on the right, and the pivot is re-located to the correct position if the array was sorted.

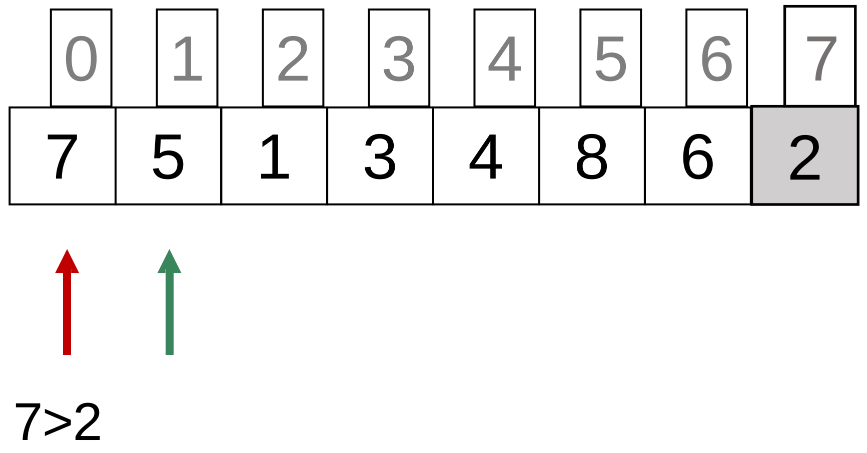
Quick Sort - Example

Compare the *pivot* with other values in the array (starting at *i=0*):



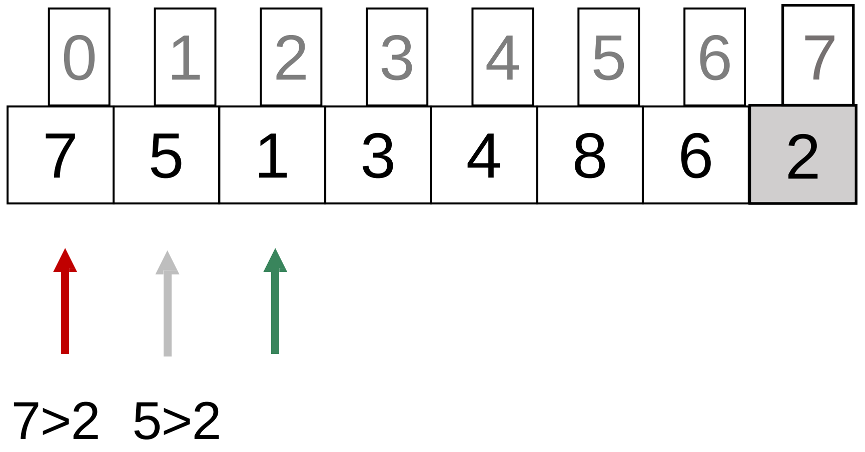
Quick Sort - Example

Since ***7>2***, fix **the pointer** at *i=0* and use **another pointer** for iterating over other elements (until finding a value that is smaller or equal to the *pivot*):



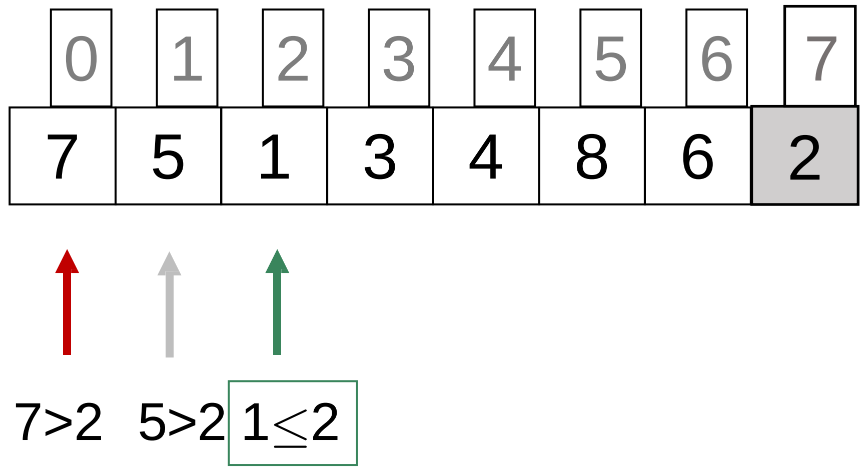
Quick Sort - Example

Since ***5>2***, move **that other pointer** one index forward:

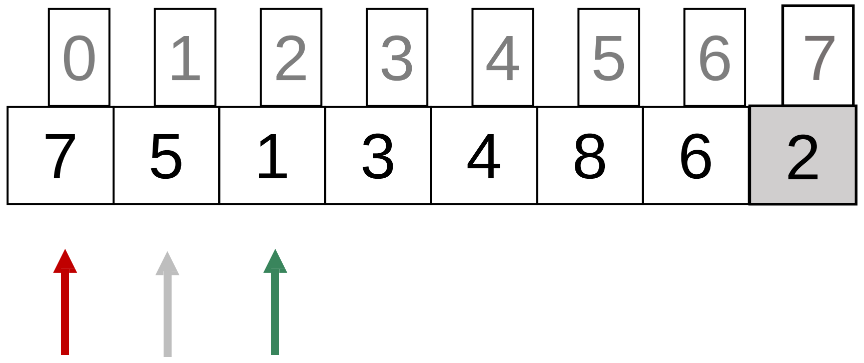


Quick Sort - Example

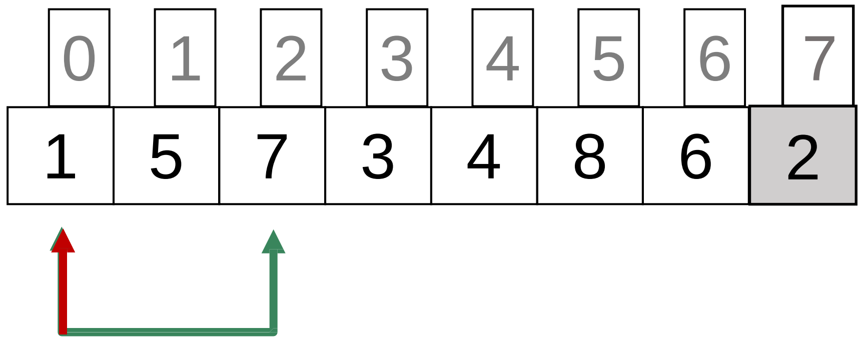
Now $\mathbf{1} \leq \mathbf{2}$, meaning that we found one value that is smaller than the *pivot*.



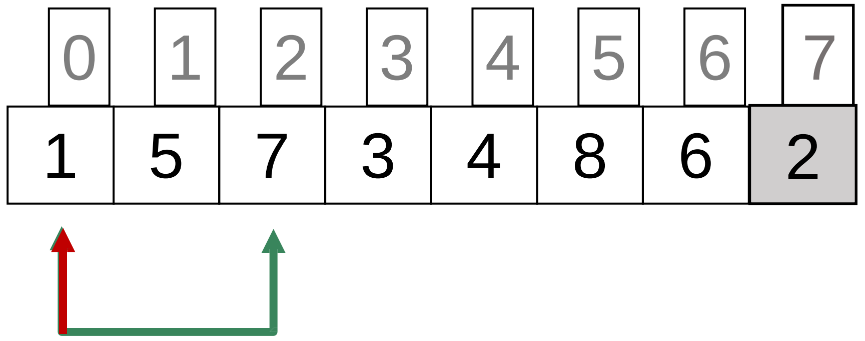
Quick Sort - Example



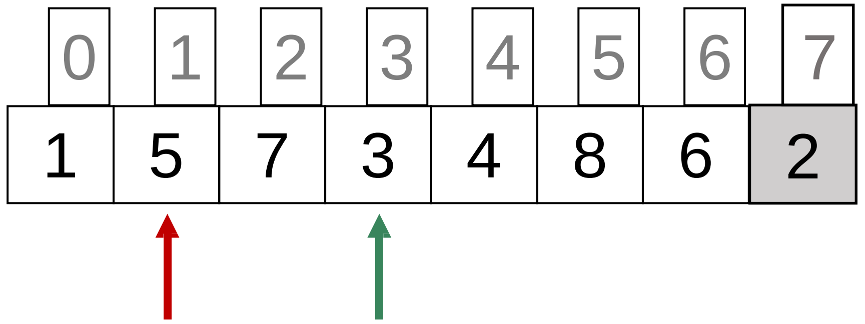
Swap this value with that of the previously fixed pointer (at *i=0*), which basically represents the first-found value that is larger than the pivot.



Quick Sort - Example

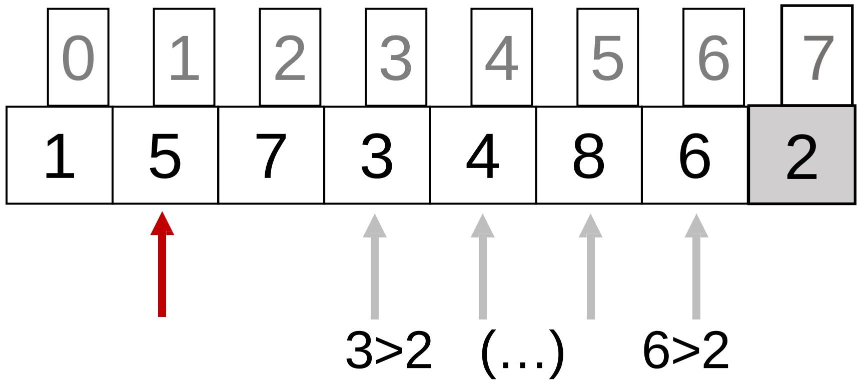


Move **the pointer** previously fixed at $i = 0$ to $i = 1$, which basically represents the second value found larger than the pivot. Move **that other pointer** (which loops over the array's values) one index forward.

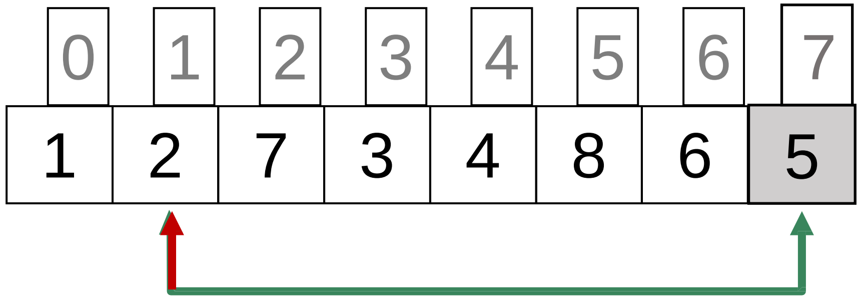


Quick Sort - Example

The procedure is repeated for all indices up to 7 (the *pivot*'s index). No other value smaller or equal to 2 was found.



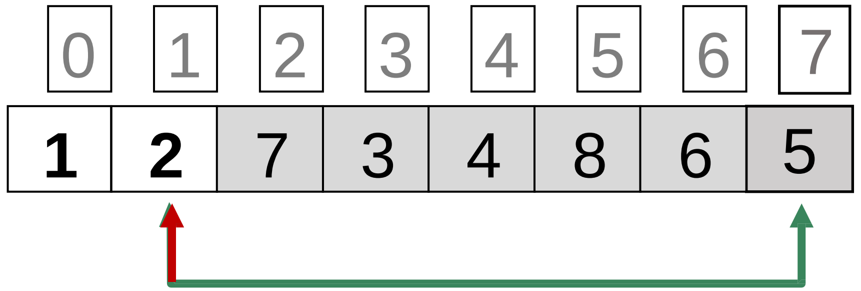
When all elements were traversed and compared with the *pivot*, swap the *pivot* with the previously fixed pointer (at *i=1*).



Quick Sort - Example

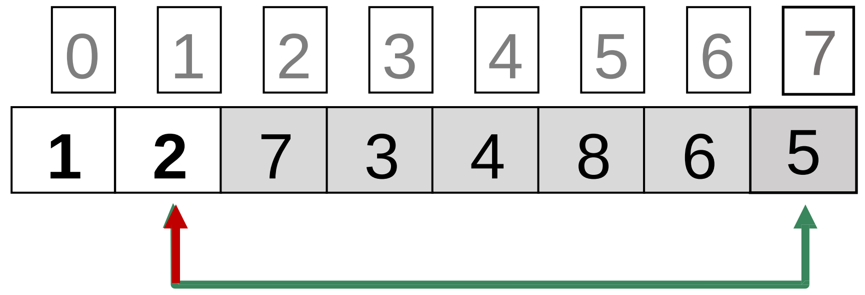
Notice that, at this stage, the pivot was placed in the *correct/right* (sorted) position, and the array was split in two parts around it:

1. **smaller or equal elements**
2. **greater elements**



Quick Sort - Example

This process is then recursively repeated for these two halves until all the input array gets sorted:



Quick-sort half 1:

0	1
1	2

Quick-sort half 2:

2	3	4	5	6	7
3	4	5	6	7	8

Quick Sort

- Pick an element, called a **pivot**, from the array
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). *After this partitioning, the pivot is in its final position*. This is called the **partition** operation.
- ***Recursively*** apply the above steps to the sub-array of elements with *smaller* values and separately to the sub-array of elements with *greater* values.

Quick Sort

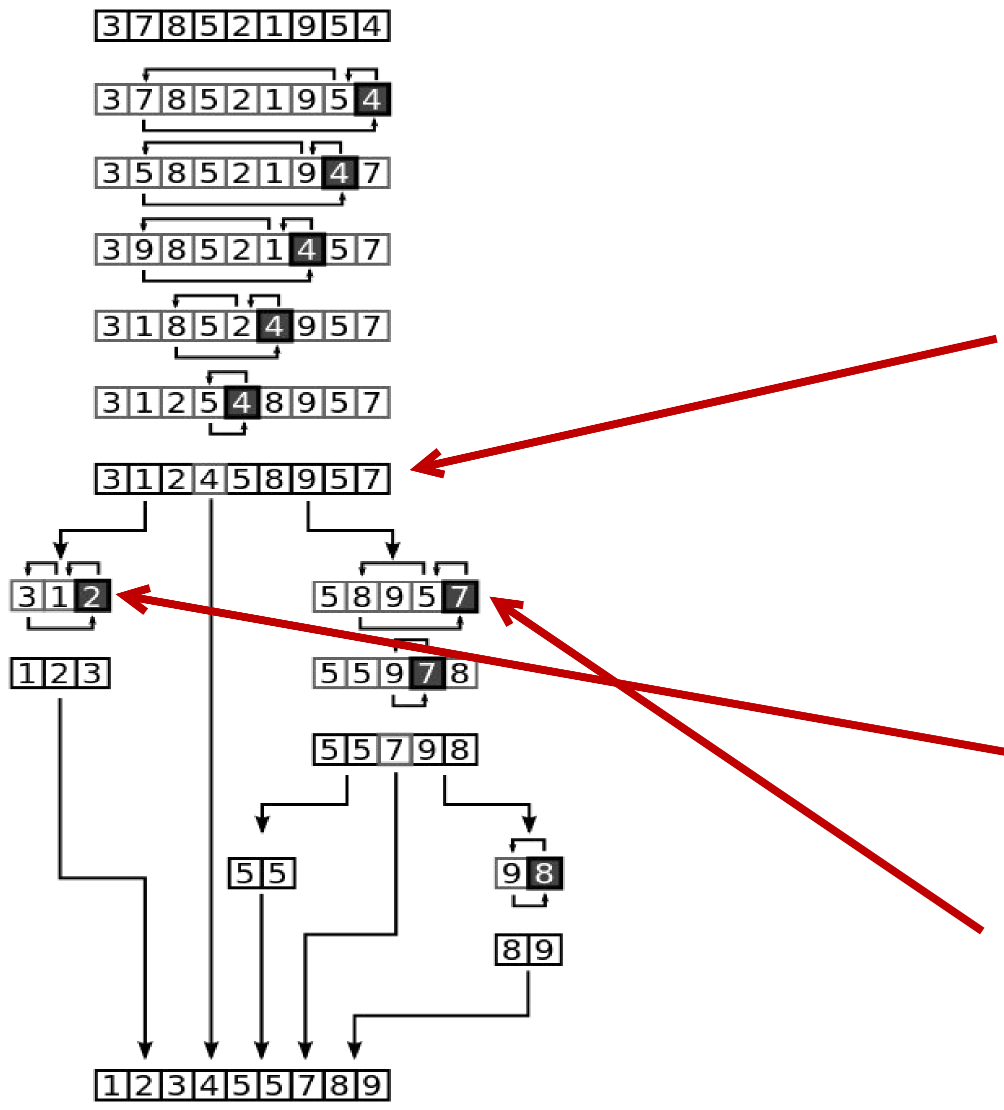
A pseudocode sketch of Quick Sort is as follows:

```
function quickSort
        input: an array of n non-sorted numbers called A
               the leftmost index of the portion to be sorted i
               the rightmost index of the portion to be sorted k
        output: the portion of array A from position i to k sorted

   if i < k then
                p := partition(A, i, k)
                quickSort(A, i, p - 1)
                quickSort(A, p + 1, k)
```

Where *the partition function determines a pivot value, puts all the elements smaller than the pivot to the left of the pivot and all the values larger than pivot to the right of the pivot* and then returns the ***index*** of the pivot.

Quick Sort



The partition function takes the element on the right hand of the array (number 4 in this case) and brings the array in a state in which **all numbers smaller than 4 are on the left of 4, and all the numbers larger than 4 are on the right of 4**.

Now 4 is in its final position!

All we need to do is to **re-apply recursively the quicksort** to the part of the array that is **at the left of number 4** and to the part of the array that is **at the right of number 4**.

Quick Sort - Complexity

Before beginning the analysis of the time computational complexity of quicksort, it is good to remind that quicksort has the advantage of being able to sort *"in-place"*, i.e., *without the need for a temporary array* (as opposite, for instance, to merge sort).

Now let's study time complexity

Quick Sort - Complexity

Let’s begin by analyzing the computational **complexity of the partition function**.

Basically, it is composed by one cycle that allows to scan the elements of the array from a position called “low” to a position called “high”. In the worst case, low = 0, while high = n -1 (where n is the size of the array), so the number of iterations of this cycle is equal to n.

In general, we analyze just a portion of the array, so the number of iterations is c·n.

In any case, the complexity of the partition function is **Θ(n)**.

Quick Sort - Complexity

Now we focus on the complexity of the whole quicksort algorithm.

Recall that quicksort involves the partition function, and 2 recursive calls.

Thus, the basic quicksort relation is:

$T(n) = \Theta(n) + T(h) + T(n-h-1) = c \cdot n + T(h) + T(n-h-1)$

where h is the size of the first (i.e. left) sub-block after partitioning.

To find the solution for this relation, we'll consider three cases:

1. The Worst case
2. The Best-case
3. The Average-case

All depends on the value of the pivot!!

Quick Sort - Complexity

The ***worst case*** happens when *the pivot is the smallest (or largest) element* of the portion of array that must be sorted.

In such a case, if n is the size of the portion that has to be sorted, we have one empty sub-block, one element (pivot) in the "correct" place and one sub-block of size (n-1):

$$T(n) = \Theta(n) + T(0) + T(n-1)$$

$$T(n) = c \cdot n + 0 + T(n-1)$$

$$T(n) = \underbrace{c \cdot n + c \cdot n + \dots + c \cdot n}_{\text{n times}}$$

So, the complexity of the quicksort in the **worst case** is $\boldsymbol{\Theta(n^2)}$, which is not better than the bubblesort... (in other words, it is "quite poor"!)

Quick Sort - Complexity

The **best case** happens when *the pivot is the median element* of the portion of array that must be sorted.

In such a case, if n is the size of the portion that has to be sorted, we have one sub-block of size n/2 (or n/2 – 1), one element (pivot) in the “correct” place and one sub-block of size n/2 (or n/2 – 1):

$$T(n) = \Theta(n) + 2 \cdot T(n/2)$$

So, the complexity of the quicksort in the **best case** is **Θ(n log n)**, which is comparable with mergesort.

Quick Sort - Complexity

In an **average case**, we have to recursively apply the quicksort to a block of size k and to a block of size n-1-k, so:

$$\mathbf{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[\mathbf{T}(k) + \mathbf{T}(n-1-k)], \quad \mathbf{T}(0) = \mathbf{T}(1) = c$$

It is possible to prove that the **closed-form** solution to this recurrence relation is **Θ(n log n)**.

So, the complexity of the quicksort in the **average case** is **Θ(n log n)**, which is again comparable with mergesort.

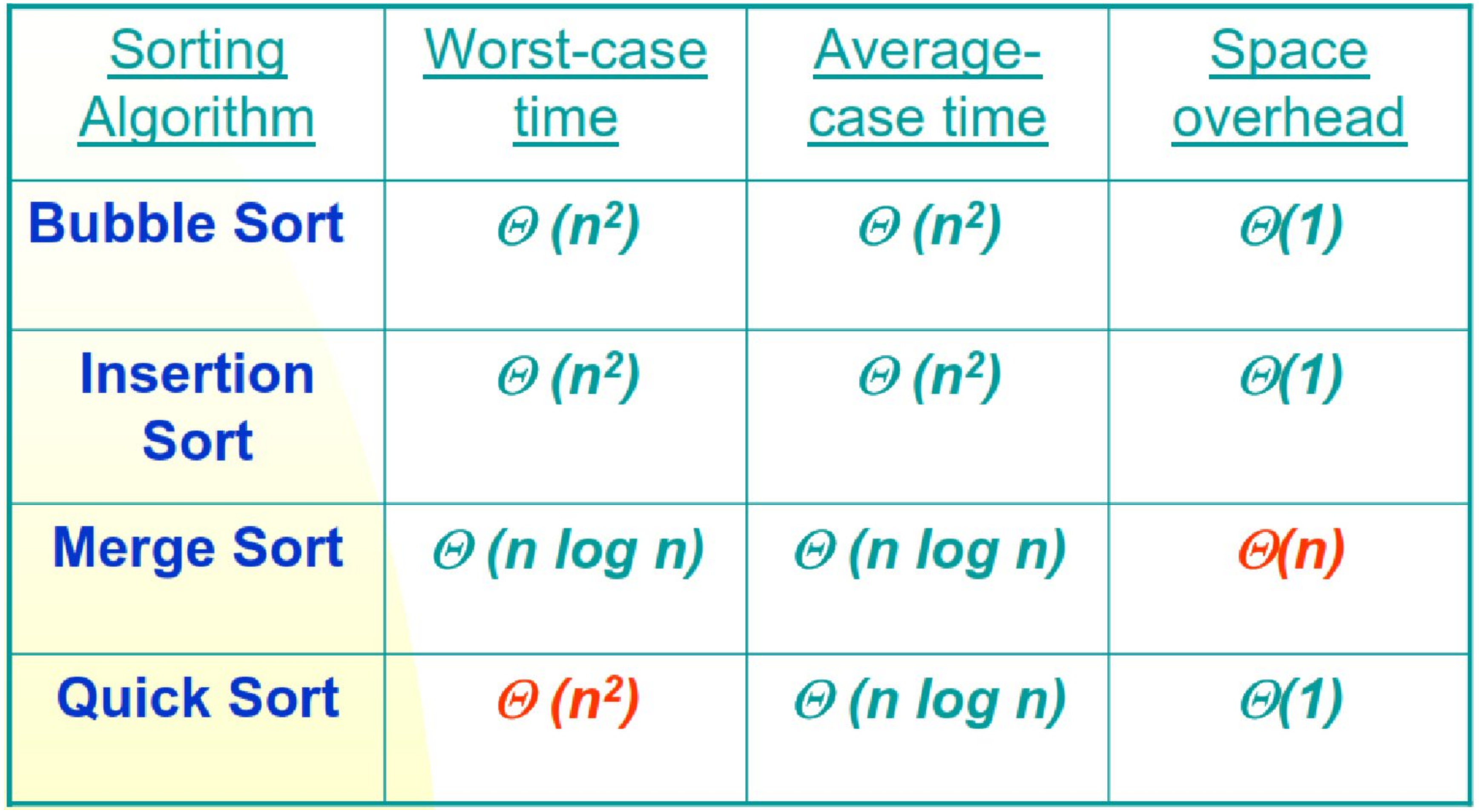
Quick Sort - Complexity

It is good to repeat that the worst case happens when we choose, as pivot, the maximum or the minimum element in the array.

Unless the unlucky case in which many elements of the array are identical between each other, as the size n of the array increases, the probability of occasionally choosing the max or min as pivot decreases... becoming quite a small probability for big sizes of the array.

We can informally conclude that *quicksort will always have good behavior, except for some unlucky cases*.

Sorting Algorithms – Let’s sum up



Sorting Algorithm	Worst-case time	Average-case time	Space overhead
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$