

# Sorting Algorithms

Algorithms, Data and Security  
A.Y. 2024/25

**Valeria Cardellini**

Global Governance, 3rd year  
Science and Technology Major

## Sorting

---

- Given a set of items, the goal is to sort the items in the set
- Examples: sort alphabetically a list of names, a list of numbers, etc...
- Basic algorithm used in many problems
  - E.g., if you sort, then you can do binary search
- See visualization <https://visualgo.net/bn/sorting>

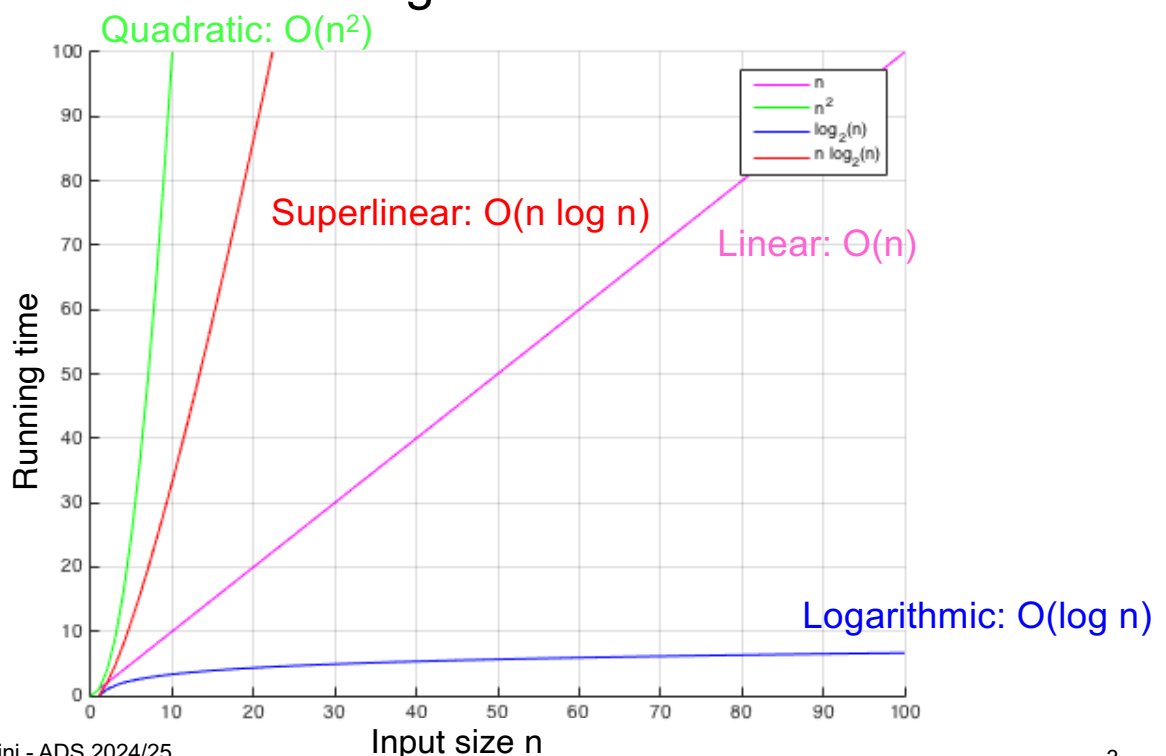
# Sorting algorithms and running times

- We will study four sorting algorithms
  - We will sort an array (or list) of integer numbers in **non-decreasing order** (i.e., from smallest to largest)
    - E.g., 1 2 2 3 3 4 5 6 6
  - All the algorithms we will consider are **comparison-based**, i.e., they sort by comparing elements
- Running times:  $O(n^2)$ ,  $O(n \log n)$ 
  - $n$  is the number of items to sort

$n$	10	100	1000	$10^6$	$10^9$
$n \log_2 n$	$\sim 33.3$	$\sim 665$	$\sim 10^4$	$\sim 2 \times 10^7$	$\sim 3 \times 10^{10}$
$n^2$	100	$10^4$	$10^6$	$10^{12}$	$10^{18}$

## Running times

- Some common big O functions



# SelectionSort

- Idea: repeatedly select the smallest element (i.e., the minimum) from the unsorted portion of the list and swaps it with the first element of the unsorted portion
  - Incremental approach
- Example

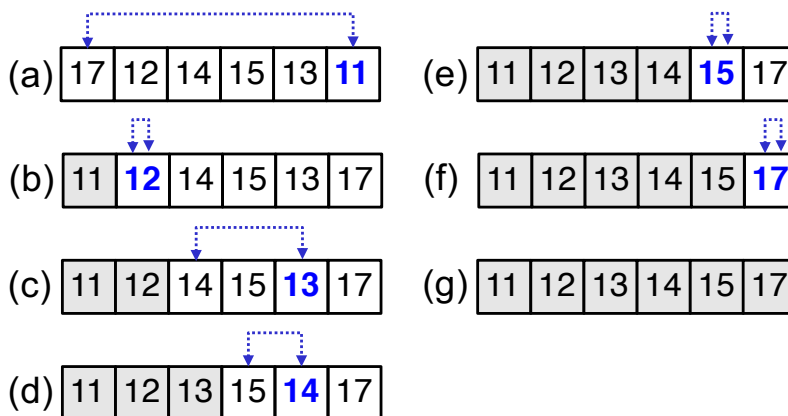


Figure (from top to bottom):  
white: still unsorted  
grey: already sorted  
blue: minimum and swap

## SelectionSort: how to find minimum

- How to find the minimum element from a particular position onwards
  - We will refer to that position as  $i$
- smallest* place marker keeps track of the position of the smallest value we have seen so far
- Initialize *smallest* to  $i$
- Look at every value from  $i+1$  onwards and update *smallest* only when you find a value smaller than the one at *smallest*

```
smallest =  $i$ 
for  $j = i + 1$  to  $n$  do
    if  $L[j] < L[\textit{smallest}]$  then
        smallest =  $j$ 
    end if
end for
```

## SelectionSort: how to find minimum

- Example of finding minimum value from a particular position ( $i=1$ ) onwards

17	12	14	15	13	11
1	2	3	4	5	6

```
smallest = i
for j = i + 1 to n do
  if L[j] < L[smallest] then
    smallest = j
  end if
end for
```

*smallest* = 1

*j* = 2

*smallest* = 2      because  $L[2] < L[\textit{smallest}]$

*j* = 3

*j* = 4

*j* = 5

*j* = 6

*smallest* = 6

because  $L[6] < L[\textit{smallest}]$

→ The minimum value is  $L[6]$ , that is 11

} when  $j=3,4,5$  we do not update *smallest*

## SelectionSort: algorithm

**algorithm** SelectionSort(*list* *L*)

for *i* = 1 to *n* - 1 do

*smallest* = *i*

    for *j* = *i* + 1 to *n* do

        if  $L[j] < L[\textit{smallest}]$  then

*smallest* = *j*

        end if

    end for

    if  $i \neq \textit{smallest}$  then

        swap  $L[i]$  and  $L[\textit{smallest}]$

    end if

end for

Find minimum in the unsorted portion of the list

Swap minimum with the leftmost unsorted element (if needed)

Running time?

## SelectionSort: running time

---

- Let's focus on the **number of comparisons**
- At step  $k$ , finding minimum takes  $O(n-k)$  time
  - Finding minimum value from position 1 onwards:  $n-1$  comparisons
  - Finding minimum value from position 2 onwards:  $n-2$  comparisons
  - ...
  - Finding minimum value from position  $n-1$  onwards: 1 comparison
- In total,  $(n-1) + (n-2) + \dots + 1$  comparisons
  - Let's express  $(n-1) + (n-2) + \dots + 1$  as  $\sum_{i=1}^{n-1} i$

## SelectionSort: running time

---

- At step  $k$ , finding minimum takes  $O(n-k)$  time
- In total, the running time is:

$$\sum_{k=1}^{n-1} O(n-k) = O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

To solve: set  $i=n-k$  and use the **arithmetic series formula** to express the sum by its closed form value

$$\sum_{k=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

# InsertionSort

---

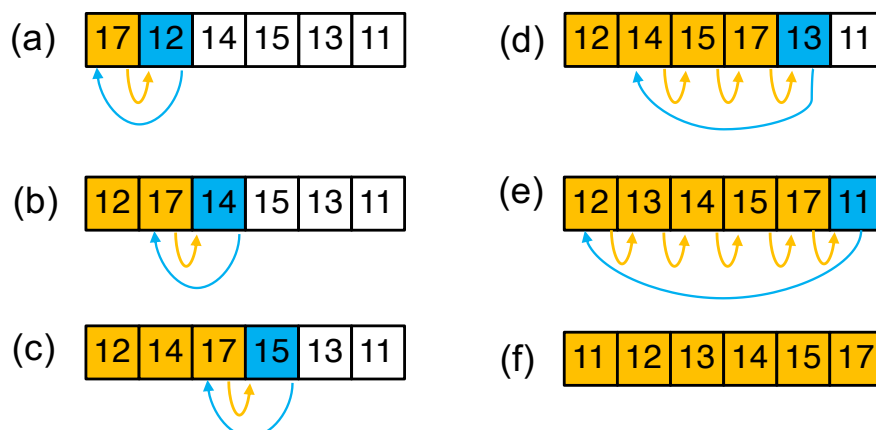
- Insertion sort works the way you might sort a hand of playing cards



# InsertionSort

---

- Idea: **extend sorted portion** from  $k-1$  to  $k$  items by **inserting** the  $k$ -th item in the proper position among the first  $k-1$  items
  - Incremental approach
- Example



## InsertionSort: algorithm

---

```
algorithm InsertionSort(list  $L$ )  
  for  $i = 2$  to  $n$  do  
     $value = L[i]$   
     $j = i - 1$   
    ▷ Insert  $L[i]$  into the sorted subarray  $L[1 : i - 1]$   
    while  $j > 0$  and  $L[j] > value$  do  
       $L[j + 1] = L[j]$   
       $j = j - 1$   
    end while  
     $L[j + 1] = value$   
  end for
```

Running time?

## InsertionSort: running time

---

- Inserting  $k$ -th item among the first  $k-1$  items takes time  $O(k)$  in the worst case
- In total, the running time is

$$O\left(\sum_{k=1}^{n-1} k\right) = O(n^2)$$

To solve: use again the arithmetic series formula

$$\sum_{k=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

# BubbleSort

---

- Scan the list several times
- At each scan compare adjacent pairs, and swap them if they are not in the right order
- If no items were swapped during a scan, you can stop as the list is sorted
- Why BubbleSort? Larger items "bubble" to the end of the list

# BubbleSort

---

- Example

(a) 

17	12	14	15	13	11
----	----	----	----	----	----

12 17

14 17

15 17

13 17

11 17

(b) 

12	14	15	13	11	17
----	----	----	----	----	----

12 14

14 15

13 15

11 15

(c) 

12	14	13	11	15	17
----	----	----	----	----	----

(c) 

12	14	13	11	15	17
----	----	----	----	----	----

12 14

13 14

11 14

(d) 

12	13	11	14	15	17
----	----	----	----	----	----

12 13

11 13

(e) 

12	11	13	14	15	17
----	----	----	----	----	----

11 12

(f) 

11	12	13	14	15	17
----	----	----	----	----	----



## BubbleSort: running time

---

- Scan the list several times
  - At each scan compare adjacent pairs, and swap them if they are not in the right order
  - If no items were swapped during a scan, you can stop as the list is sorted
- Each scan requires  $O(n)$  time
- We have at most  $O(n)$  scans
- Therefore, running time is  $O(n^2)$
- Is the algorithm correct? Yes
  - After  $k$ -th scan, the  $k$  largest items are at the end of the list and they are correctly sorted

## Can we do better?

---

- The three sorting algorithms described so far are simple to understand but relatively slow
  - Running time:  $O(n^2)$
- Let us analyze a faster sorting algorithm

# MergeSort

---

- Based on an algorithm design technique known as **divide and conquer**
- Known also as **divide et impera** since it was used by Romans to conquer and rule other people
- BinarySearch is another algorithm based on divide and conquer

## Divide and conquer

---

- **Top-down** approach:
  1. **Divide** the original problem into two or more **subproblems** that are similar to the original problem but smaller in size
  2. **Conquer** the subproblems by solving them *recursively*. If the subproblem sizes are small enough, just solve the subproblems directly without recursing
  3. **Combine** the subproblem solutions to form a solution to the original problem

# Divide and conquer applied to MergeSort

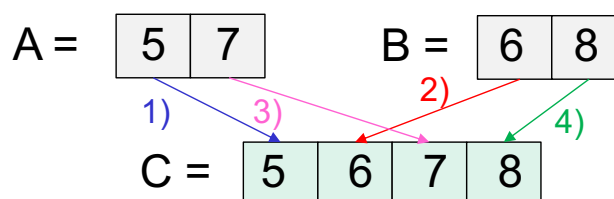
---

- Let's apply divide and conquer to MergeSort
  1. **Divide**: **split** input (list to be sorted) in two (roughly equal) halves
  2. **Conquer**: **solve** the two **subproblems** **recursively**
    - If the list has one element, it is sorted
    - Otherwise, return to step 1 to sort it
  3. **Combine**: **merge** the two sorted lists

## How to merge?

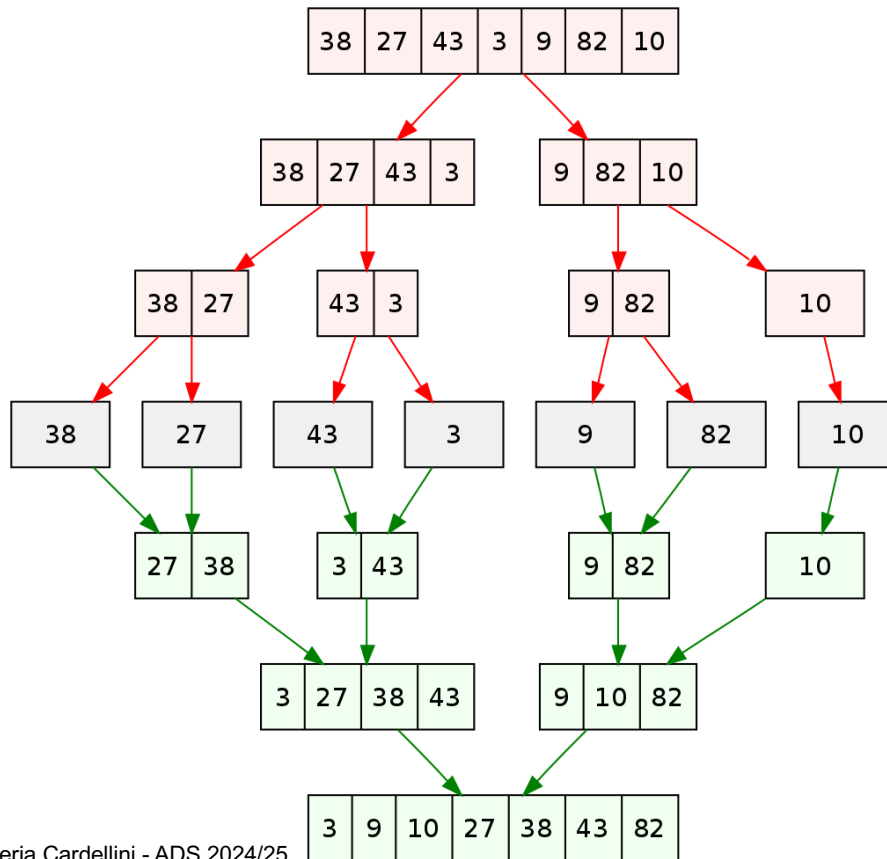
---

- Two sorted lists A e B can be merged as follows:
  - Choose minimum from A and B, copy it to output list C and remove it from A or B. Repeat until either A or B becomes empty
  - Take all the elements in the remaining non-empty list and copy them at the end of C
  - Example:



- Running time for merging:  $O(n)$ , where  $n$  is the total number of elements

## MergeSort: example



Valeria Cardellini - ADS 2024/25

22

## MergeSort: with words

- **Divide** the array  $L[l : r]$  to be sorted into two subarrays, each of half the size. To do so, compute the midpoint  $m$  of  $L[l : r]$  (taking the average of  $l$  and  $r$ ), and divide  $L[l : r]$  into subarrays  $L[l : m]$  and  $L[m+1 : r]$
- **Conquer** by sorting each of the two subarrays  $L[l : m]$  and  $L[m+1 : r]$  *recursively* using MergeSort
- **Combine** by merging the two sorted subarrays  $L[l : m]$  and  $L[m+1 : r]$  back into  $L[l : r]$ , producing the sorted output

Valeria Cardellini - ADS 2024/25

23

# MergeSort: pseudocode

---

**algorithm** MergeSort(*list*  $L$ , *integer*  $l$ , *integer*  $r$ )

**if**  $l < r$  **then**

$m = \lfloor (l + r) / 2 \rfloor$

    ▷ find midpoint of  $L[l : r]$

    MergeSort( $L$ ,  $l$ ,  $m$ )

    ▷ call MergeSort for first half

    MergeSort( $L$ ,  $m + 1$ ,  $r$ )

    ▷ call MergeSort for second half

    Merge( $L$ ,  $l$ ,  $m$ ,  $r$ )

    ▷ merge the two sorted halves

**end if**

# Merge: pseudocode

---

**algorithm** Merge(*list*  $L$ , *integer*  $l$ , *integer*  $m$ , *integer*  $r$ )

$n_1 = m - l + 1$

  ▷ length of  $L[l:m]$

$n_2 = r - m$

  ▷ length of  $L[m+1:r]$

  Let  $A[1 \dots n_1 + 1]$  and  $B[1 \dots n_2 + 1]$  be new arrays

**for**  $i = 1$  to  $n_1$  **do**

  ▷ copy  $L[l:m]$  into  $A[1:n_1]$

$A[i] = L[l + i - 1]$

**end for**

**for**  $j = 1$  to  $n_2$  **do**

  ▷ copy  $L[m+1:r]$  into  $B[1:n_2]$

$B[j] = L[m + j]$

**end for**

$A[n_1 + 1] = \infty$

$B[n_2 + 1] = \infty$

$i = 1$

  ▷  $i$  indexes the smallest remaining element in  $A$

$j = 1$

  ▷  $j$  indexes the smallest remaining element in  $B$

**for**  $k = l$  to  $r$  **do**

  ▷ at each step copy the smallest unmerged element back into  $L[l:r]$

**if**  $A[i] \leq B[j]$  **then**

$L[k] = A[i]$

$i = i + 1$

**else**

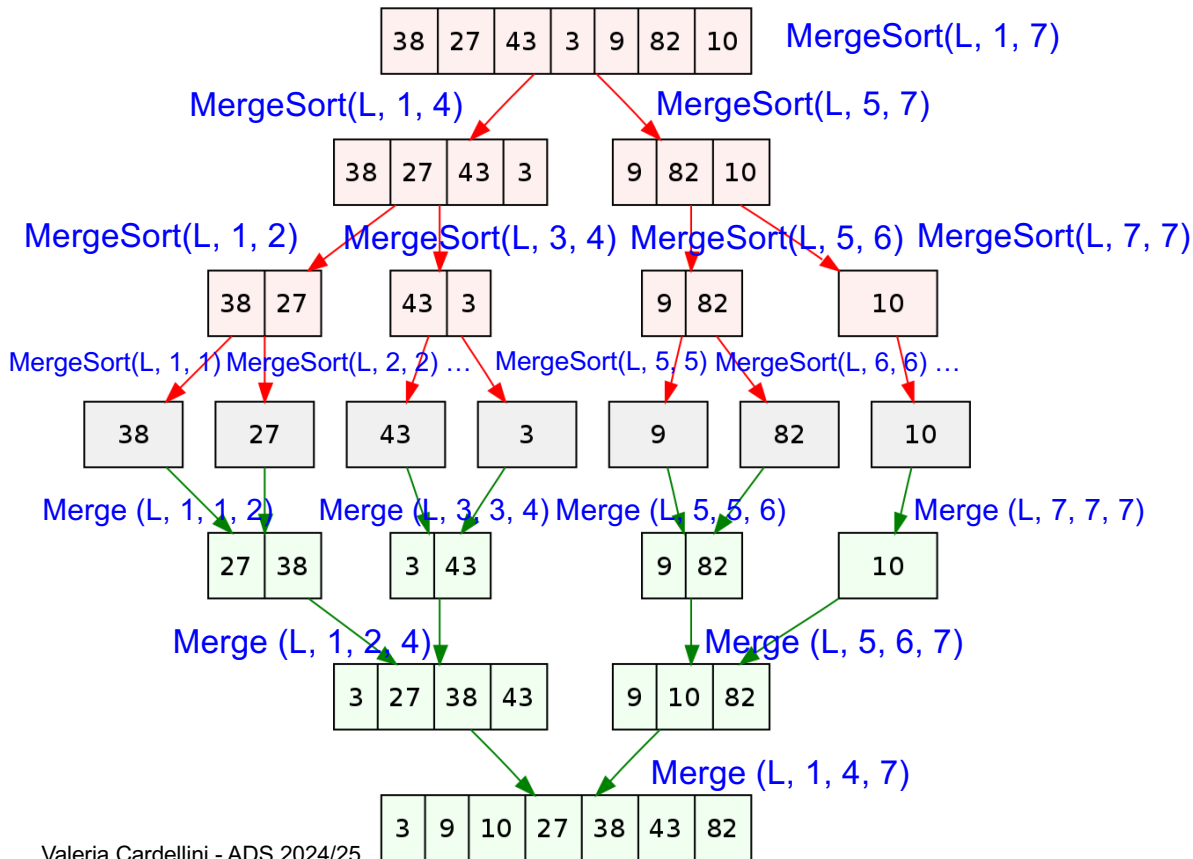
$L[k] = B[j]$

$j = j + 1$

**end if**

**end for**

# MergeSort and its call tree: example



Valeria Cardellini - ADS 2024/25

26

## MergeSort: running time

- Let  $T(n)$  be the number of comparisons done by MergeSort to sort  $n$  elements:
  - $T(n/2)$  comparisons to sort the first half ( $n/2$  elements)
  - $T(n/2)$  comparisons to sort the second half ( $n/2$  elements)
  - $cn$  comparisons to merge the two sorted subarrays, for some constant  $c$
- $T(n) = 2T(n/2) + cn$

## MergeSort: running time

---

- Let  $T(n)$  be the number of comparisons done by MergeSort to sort  $n$  elements:

$$T(n) = 2T(n/2) + cn$$

$$T(1) = 1$$

- $T(1) = 1$  (if the list has only one element, nothing to do)
- It is a recurrence. How do we solve it?

## MergeSort: running time

---

- We have to solve

$$T(n) = 2T(n/2) + cn$$

- If this is true, then we can also say

$$T(n/2) = 2T(n/4) + cn/2$$

- Putting this back into the recurrence, we obtain:

$$T(n) = 2T(n/2) + cn = 4T(n/4) + 2cn/2 + cn$$

- i.e.,

$$T(n) = 4T(n/4) + 2cn$$

## MergeSort: running time

---

- We have

$$T(n) = 2T(n/2) + cn = 4T(n/4) + 2cn$$

- We can also say

$$T(n/4) = 2T(n/8) + cn/4$$

- Putting this back into the recurrence, we obtain:

$$T(n) = 4T(n/4) + 2cn = 8T(n/8) + 4cn/4 + 2cn$$

- i.e.,

$$T(n) = 8T(n/8) + 3cn$$

## MergeSort: running time

---

- We can go on like this:

$$T(n) = 8T(n/8) + 3cn = 16T(n/16) + 4cn = \dots$$

- In general, for  $k = 1, 2, 3, \dots$

$$T(n) = 2^k T(n/2^k) + kcn$$

- When do we stop? Cannot go on forever...



## MergeSort: running time

---

- We have:

$$T(n) = 2^k T(n/2^k) + kcn$$

- When do we stop? Cannot go on forever...
- We can stop when  $k$  is such that  $n/2^k = 1$
- i.e.,  $n = 2^k$ , i.e.,  $k = \log_2 n$ . For this value of  $k$ :

$$T(n) = 2^k T(n/2^k) + kcn = nT(1) + c n \log_2 n$$

- Namely:

$$T(n) = n + c n \log_2 n = O(n \log n)$$

## MergeSort: summary

---

- The number of comparisons  $T(n)$  done by MergeSort to sort  $n$  elements is described by the following recurrence:

$$T(n) = 2T(n/2) + cn$$

- The solution of this recurrence is

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

- MergeSort sorts  $n$  elements in time  $O(n \log n)$

## Take-away

---

- We described four sorting algorithms:
  - SelectionSort, InsertionSort, BubbleSort:  $O(n^2)$
  - MergeSort:  $O(n \log n)$

n	10	100	1000	$10^6$	$10^9$
$n \log_2 n$	$\sim 33$	$\sim 664$	$\sim 10^4$	$\sim 2 \times 10^7$	$\sim 3 \times 10^{10}$
$n^2$	100	$10^4$	$10^6$	$10^{12}$	$10^{18}$

- If we have to sort 1 billion ( $n=10^9$ ) of items, and each step takes 10 nanosec. ( $10^{-8}$  sec.):
  - $n^2$  steps take:  $10^{18} \times 10^{-8} \text{ sec} = 10^{10} \text{ sec} \sim 317 \text{ years!}$
  - $n \log_2 n$  steps take:  $\log_2(3 \times 10^{10}) \times 10^{-8} \text{ sec} \sim 5.6 \text{ sec!}$

## Exercise

---

- Apply the four algorithms to sort the following list in non-decreasing order: 16, 9, 48, 11, 21, 34, 10, 9
- To check your solution, you can also sort the list using <https://visualgo.net/bn/sorting> by first creating the list of numbers and then sorting it using the different algorithms

# References

---

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms, 4th ed., MIT Press, 2022
- C. Demetrescu, I. Finocchi, G. F. Italiano. Algoritmi e Strutture Dati, Mc-Graw Hill, 2008 (in Italian)