НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Data Structures and Organization

(p.2 – Time complexity / Arrays)

Yevhen Berkunskyi,
NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua

How well will it scale?

- From an architectural perspective, scalability refers to how flexible your app is as your features are increasing.
- From a database perspective, scalability is about the capability of a database to handle an increasing amount of data and users.
- For a web server, being scalable can mean that it can serve a high number of users accessing it at the same time.

How well will it scale?

- For algorithms, scalability refers to how the algorithm performs in terms of execution time and memory usage as the input size increases.
- With a small amount of data, any algorithm may still feel fast.
- However, as the amount of data increases, an expensive algorithm can become crippling.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Time complexity. Big O notation

- Time complexity is a measure of the time required to run an algorithm as the input size increases.
- In this lecture, we'll go through the most common time complexities and learn how to identify them.

Constant time

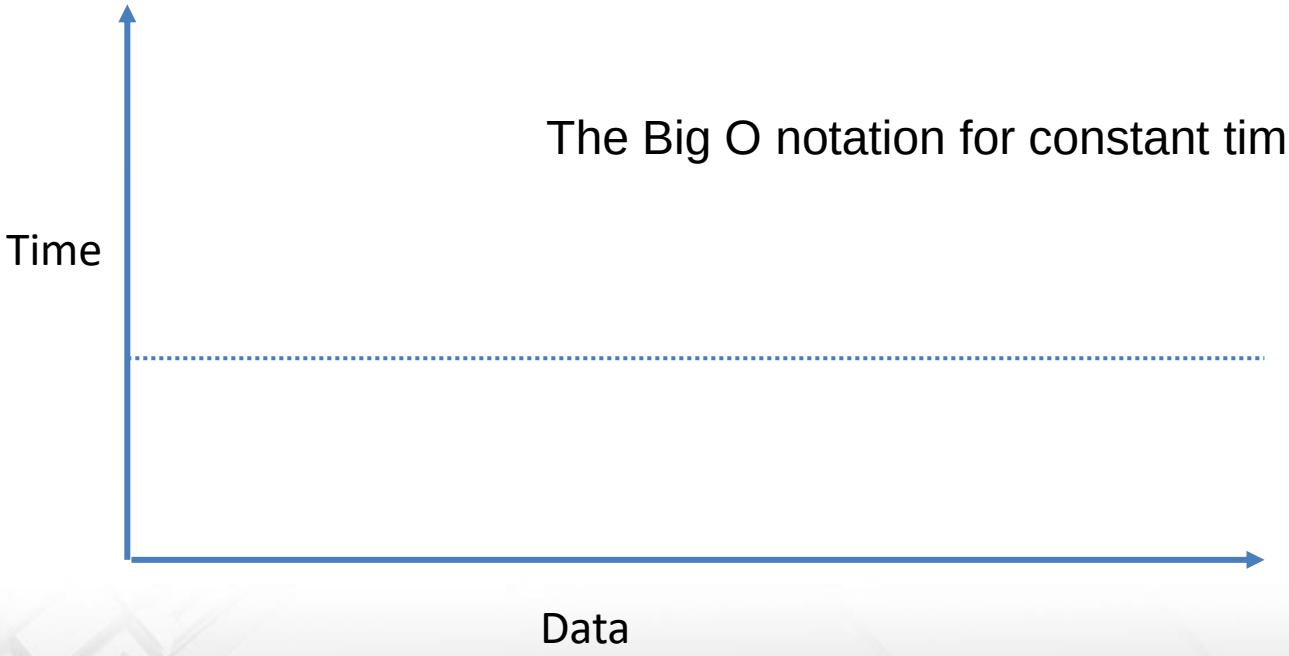
- A constant time algorithm is one that has the same running time regardless of the size of the input. Consider the following:

```
fun checkFirst(names: List<String>) {
    if (names.firstOrNull() != null) {
        println(names.first())
    } else {
        println("no names")
    }
}
```

The size of names does not affect the running time of this function. Whether names has 10 items or 10 million items, this function only checks the first element of the list.

Constant time

- Here's a visualization of this time complexity in a plot between time versus data size:



As input data increases, the amount of time the algorithm takes does not change.

Linear Time

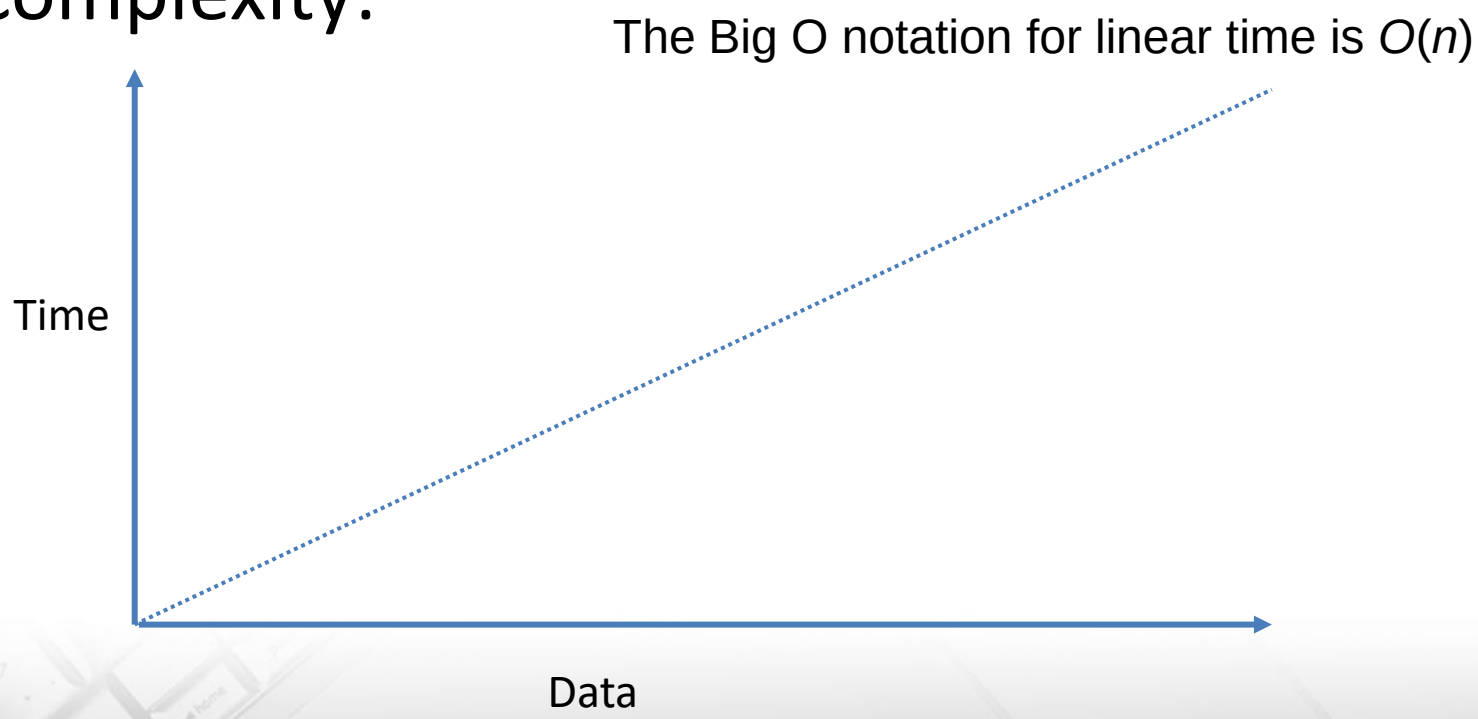
- Consider the following snippet of code:

```
fun printNames(names: List<String>) {
    for (name in names) {
        println(name)
    }
}
```

- This function prints all the names in a `String` list.
- As the input list increases in size, the number of iterations is increased by the same amount.

Linear Time

- This behavior is known as **linear time complexity:**



Note about complexity

- **Note**: What about a function that has two loops over all of the data and a calls six different $O(1)$ methods? Is it $O(2n + 6)$?
- Time complexity only gives a high-level shape of the performance. Loops that happen a set number of times are not part of the calculation.
- All constants are dropped in the final Big O notation. In other words, $O(2n + 6)$ is surprisingly equal to $O(n)$

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Quadratic time

- More commonly referred to as **n squared**, this time complexity refers to an algorithm that takes time proportional to the square of the input size.
- Consider the following code:

```
fun multiplicationBoard(size: Int) {
  for (number in 1..size) {
    print(" | ")
    for (otherNumber in 1..size) {
      print("$number x $otherNumber = ${number * otherNumber} |")
    }
    println()
  }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Quadratic time

- If you call this function using a small number, like 2, you'll get the following output:

```
| 1 x 1 = 1 | 1 x 2 = 2 |
| 2 x 1 = 2 | 2 x 2 = 4 |
```

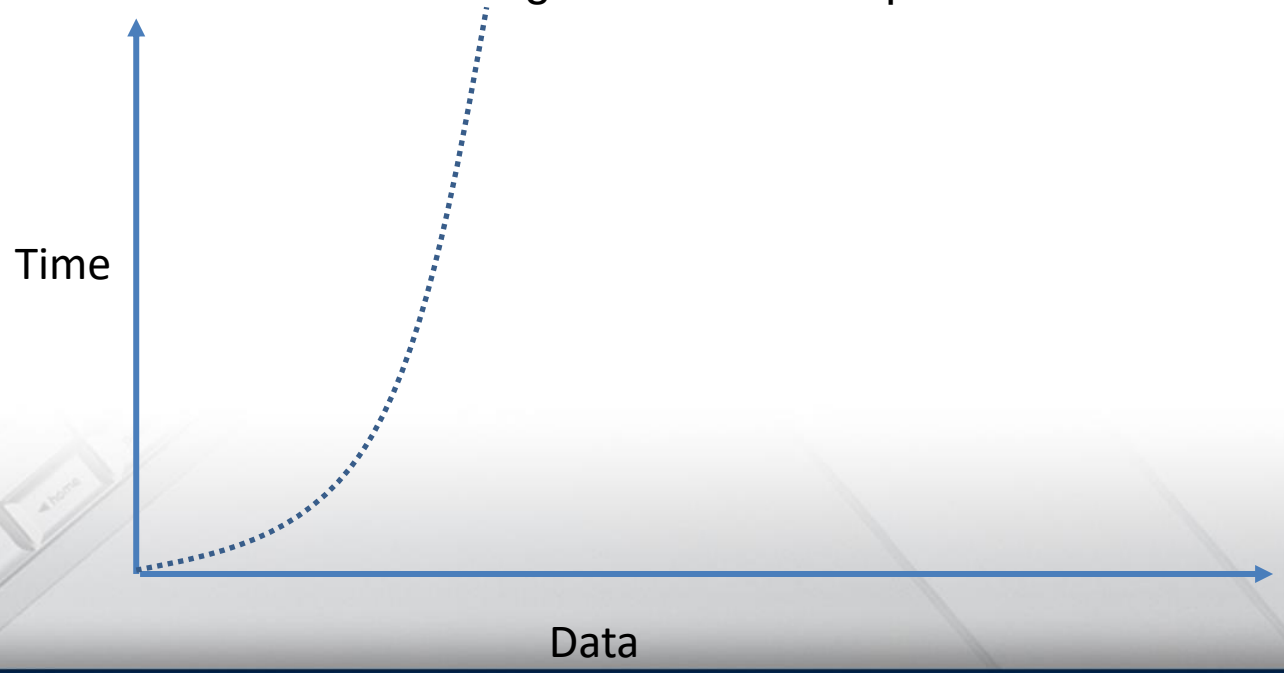
- If the input is 10, it'll print the full multiplication board of 10 × 10. That's 100 print statements.
- If you increase the input size by one, it'll print the product of 11 numbers with 11 numbers, resulting in 121 print statements.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Quadratic time

- As the size of the input data increases, the amount of time it takes for the algorithm to run increases drastically.
- Thus, n squared algorithms don't perform well at scale.

The Big O notation for quadratic time is $O(n^2)$.



Note

- No matter how inefficiently a linear time *O(n)* algorithm is written, for a sufficiently large *n*, the linear time algorithm will always execute faster than a super optimized quadratic algorithm.
- Companies put millions of dollars of R&D into reducing the slope of those constants that Big O notation ignores.
- For example, a GPU optimized version of an algorithm might run 100× faster than the naïve CPU version while remaining *O*(*n*)

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Arrays

What about arrays?

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

What arrays are?

- The array is the most commonly used data storage structure; it’s built into most programming languages.
- An array is a data structure represented as a group of cells of the same type, with one common name.
- Arrays are used to process a large number of similar data.

Class as Data Structure Unit

- Suppose you're coaching kids-league basketball, and you want to keep track of which players are present at the practice field
- What you need is an attendance-monitoring program for your laptop—a program that maintains a database of the players who have shown up for practice.
- You can use a simple data structure to hold this data.

What kind of data we'll use for player?

Let it be Player class:

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

The Player class

```
public class Player {
    private int num;
    private String name;

    public Player(int num, String name) {
        this.num = num;
        this.name = name;
    }

    @Override
    public String toString() {
        return "{" + num + ", " + name + "}";
    }
   ...
}
```

```
data class Player(var num:Int, var name:String)
```

Actions

- Insert a player into the data structure when the player arrives at the field
- Check to see whether a particular player is present, by searching for the player's number in the structure
- Delete a player from the data structure when that player goes home

These three operations: **insertion**, **searching**, and **deletion** — will be the fundamental ones in most of the data storage structures

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Example

Ordered Array

- Imagine an array in which the data items are arranged in order of ascending key values—that is, with the smallest value at index 0, and each cell holding a value larger than the cell below.
- Such an array is called an *ordered array*.
- When we insert an item into this array, the correct location must be found for the insertion: just above a smaller value and just below a larger one.
- Then all the larger values must be moved up to make room.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Binary Searching

- Why would we want to arrange data in order?
- One advantage is that we can speed up search times dramatically using a *binary search*.

The Guess-a-Number Game

- Binary search uses the same approach you did as a kid (if you were smart) to guess a number in the well-known children's guessing game.
- In this game, a friend asks you to guess a number she's thinking of between 1 and 100.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

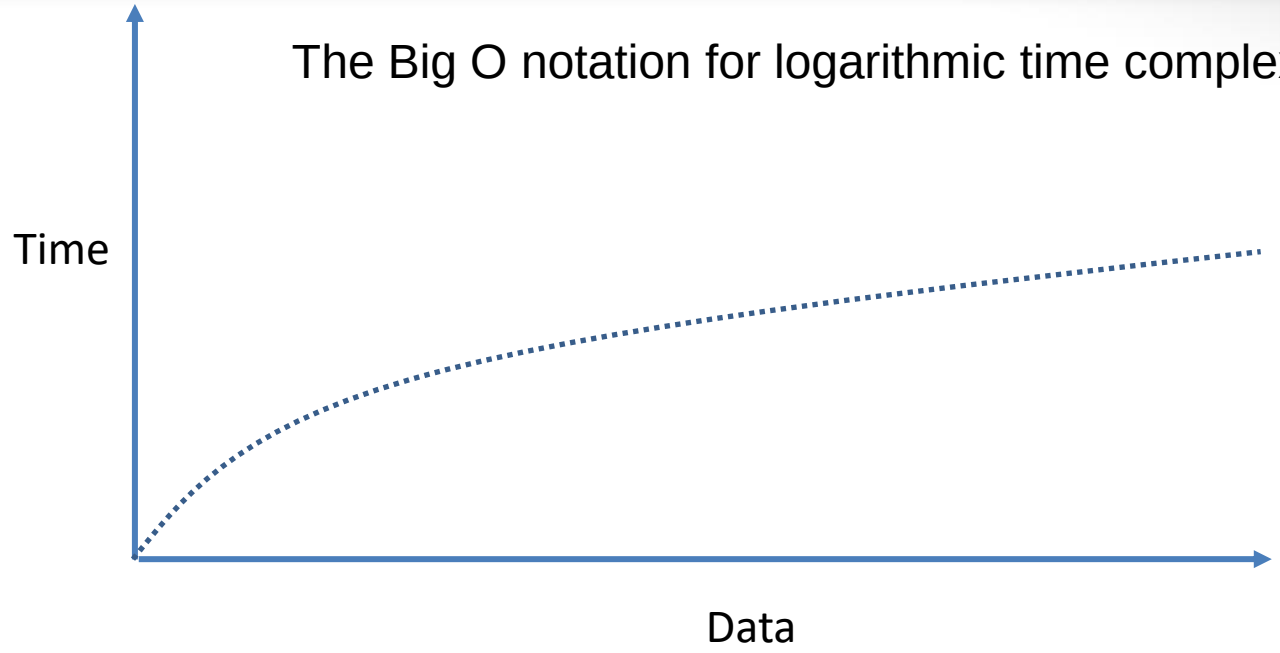
Example

Logarithmic time

- The algorithm first checks the middle value to see how it compares with the desired value.
- If the middle value is bigger than the desired value, the algorithm won't bother looking at the values on the right half of the list; since the list is sorted, values to the right of the middle value can only get bigger.
- In the other case, if the middle value is smaller than the desired value, the algorithm won't look at the left side of the list.
- This optimization cuts the number of comparisons by half

Logarithmic time

The Big O notation for logarithmic time complexity is $O(\log n)$



- As input data increases, the time it takes to execute the algorithm increases at a slower rate. This can be explained by considering the impact of halving the number of comparisons you need to do.

Simple Sorting

The three algorithms in this presentation all involve two steps, executed over and over until the data is sorted:

1. **Compare two items.**
2. **Swap two items, or copy one item.**

However, each algorithm handles the details in a different way

- **Bubble Sort**
- **Selection Sort**
- **Insertion Sort**

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Bubble Sort

The bubble sort is notoriously slow, but it's conceptually the simplest of the sorting algorithms and for that reason is a good beginning for our exploration of sorting techniques.

Here are the rules you're following:

1. Compare two elements.

2. If the one on the left is greater, swap them.

3. Move one position right.

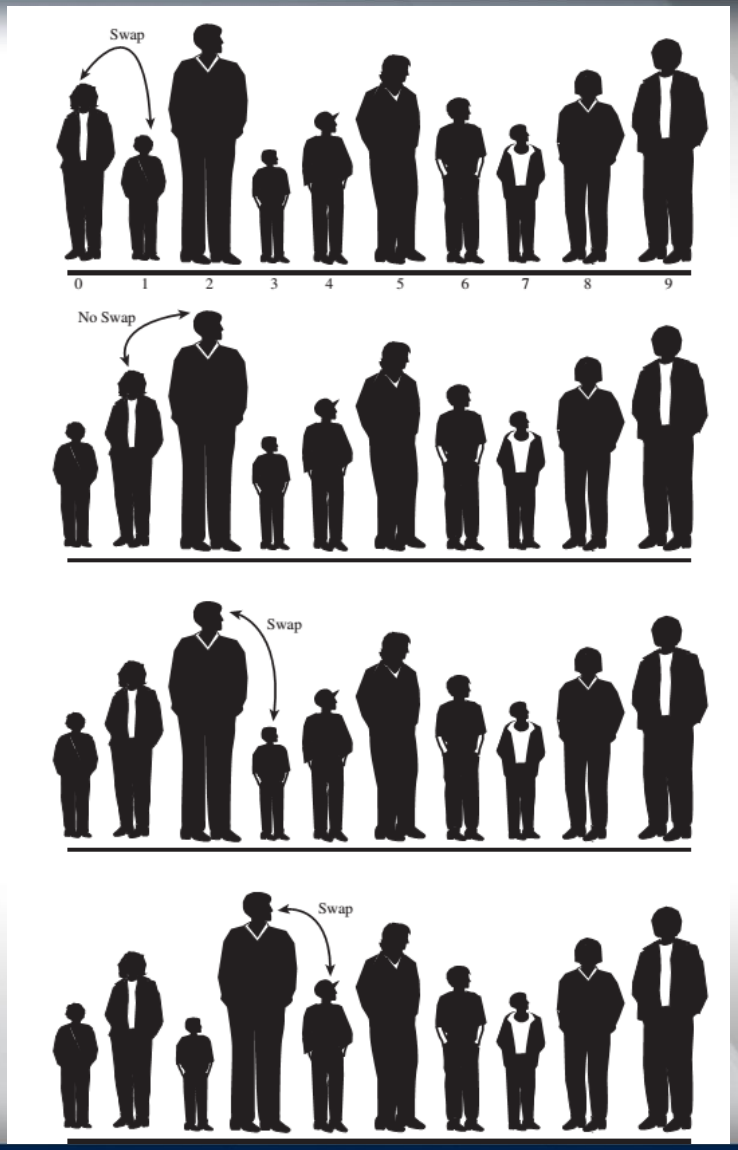
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Bubble Sort

1. Compare two elements.
2. If the one on the left is greater,
   swap them.
3. Move one position right.

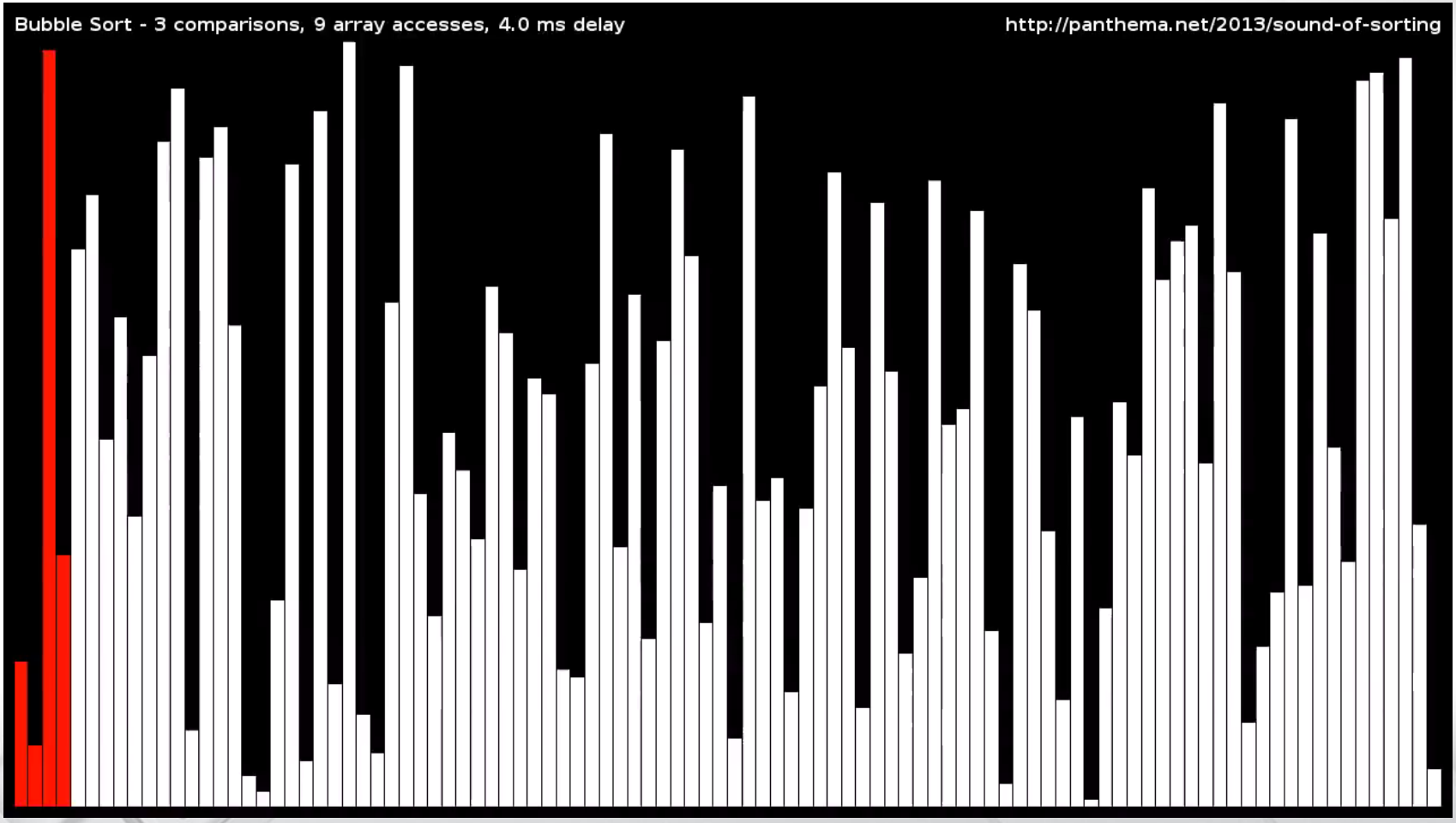
Efficiency of the Bubble Sort

(N–1) + (N–2) + (N–3) + ... + 1 = N(N–1)/2*



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Bubble Sort



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Selection Sort

A Brief Description

What's involved in the selection sort is making a pass through all the elements and picking (or *selecting*, hence the name of the sort) the smallest one. This smallest element is then swapped with the element on the left end of the line, at position 0.

Now the leftmost element is sorted and won't need to be moved again.

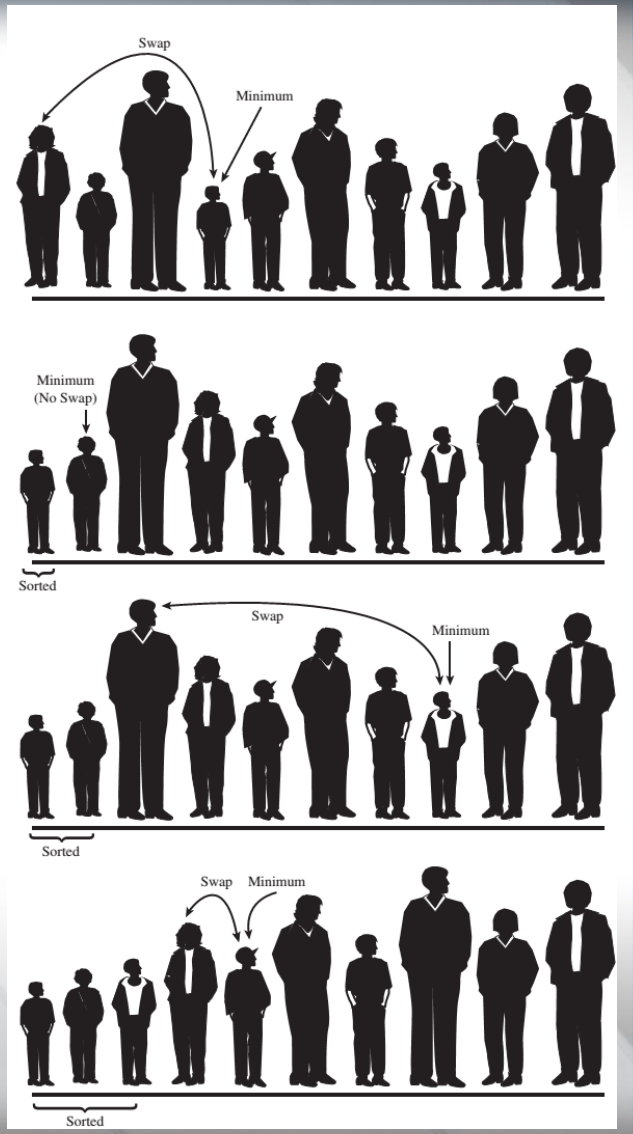
Notice that in this algorithm the sorted elements accumulate on the left (lower indices), whereas in the bubble sort they accumulated on the right.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Selection Sort

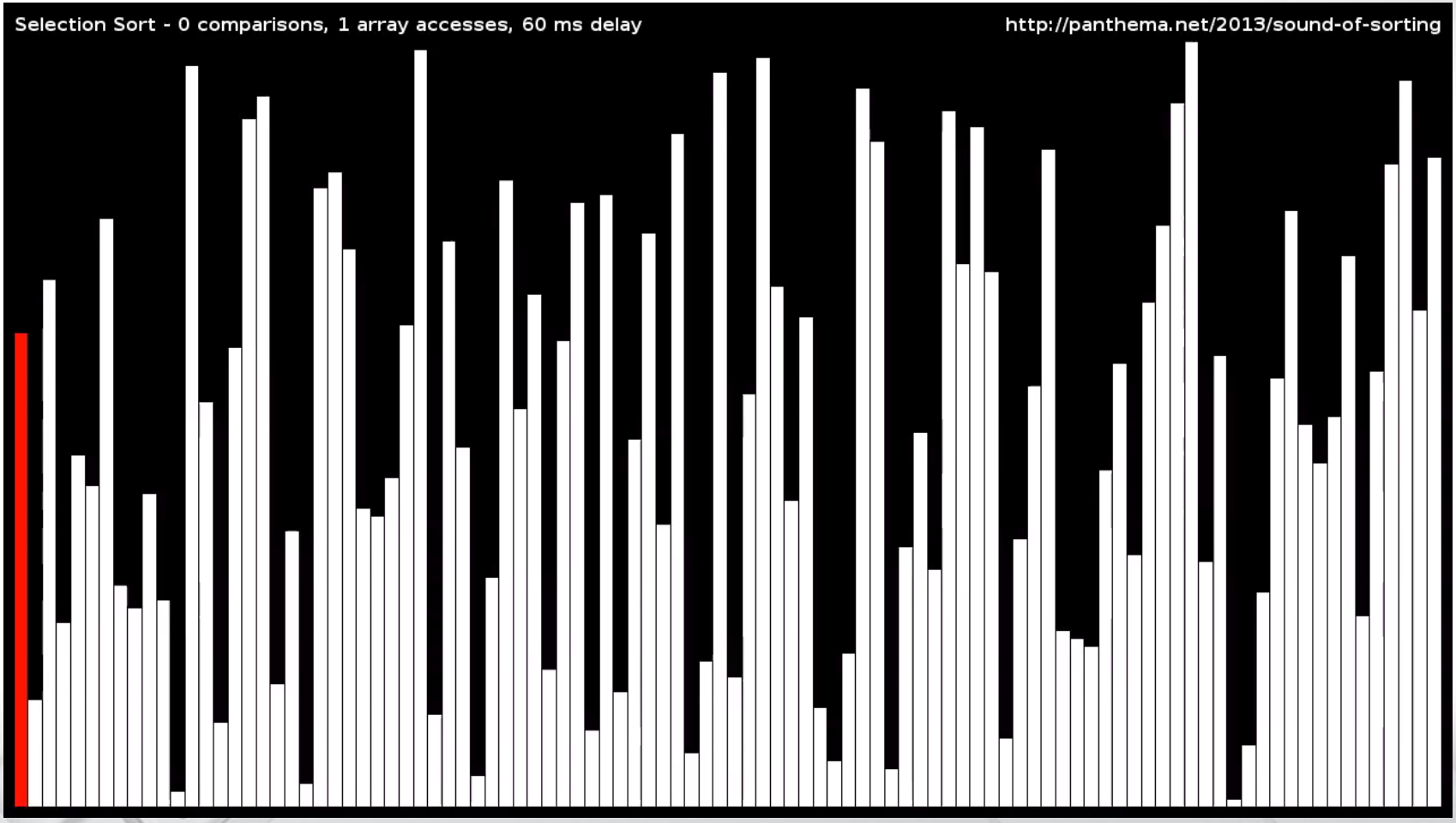
Efficiency of the Selection Sort

The selection sort performs the same number of comparisons as the bubble sort: N*(N-1)/2



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

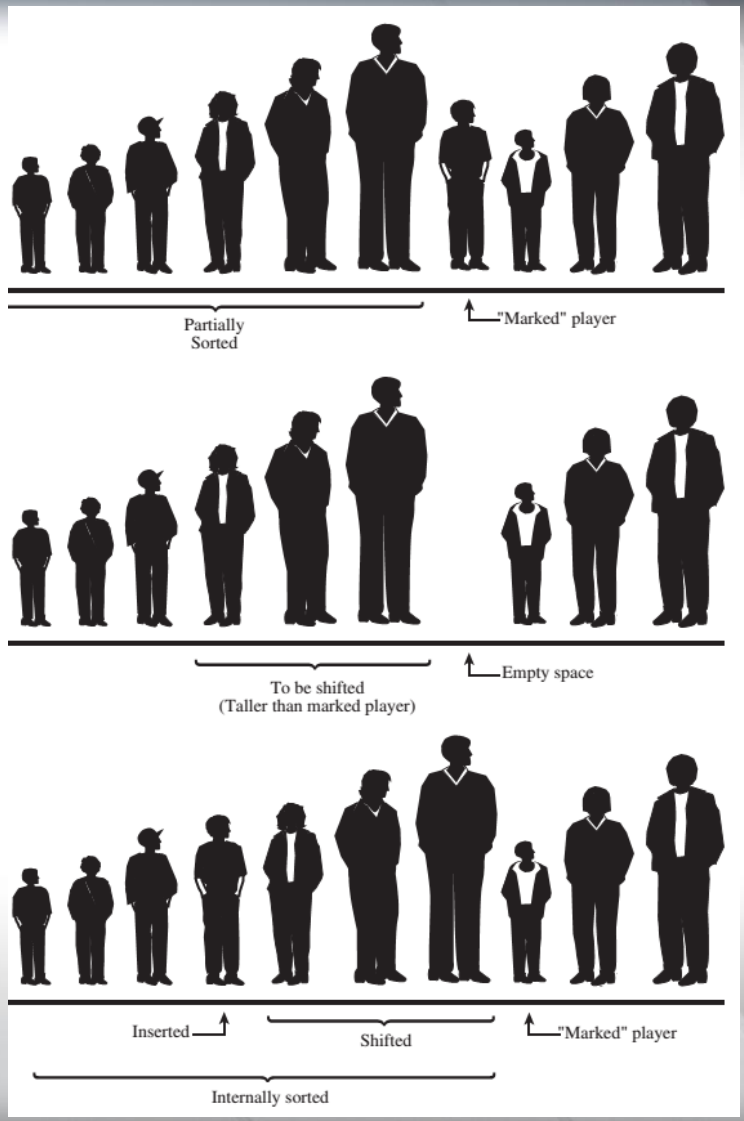
Selection Sort



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

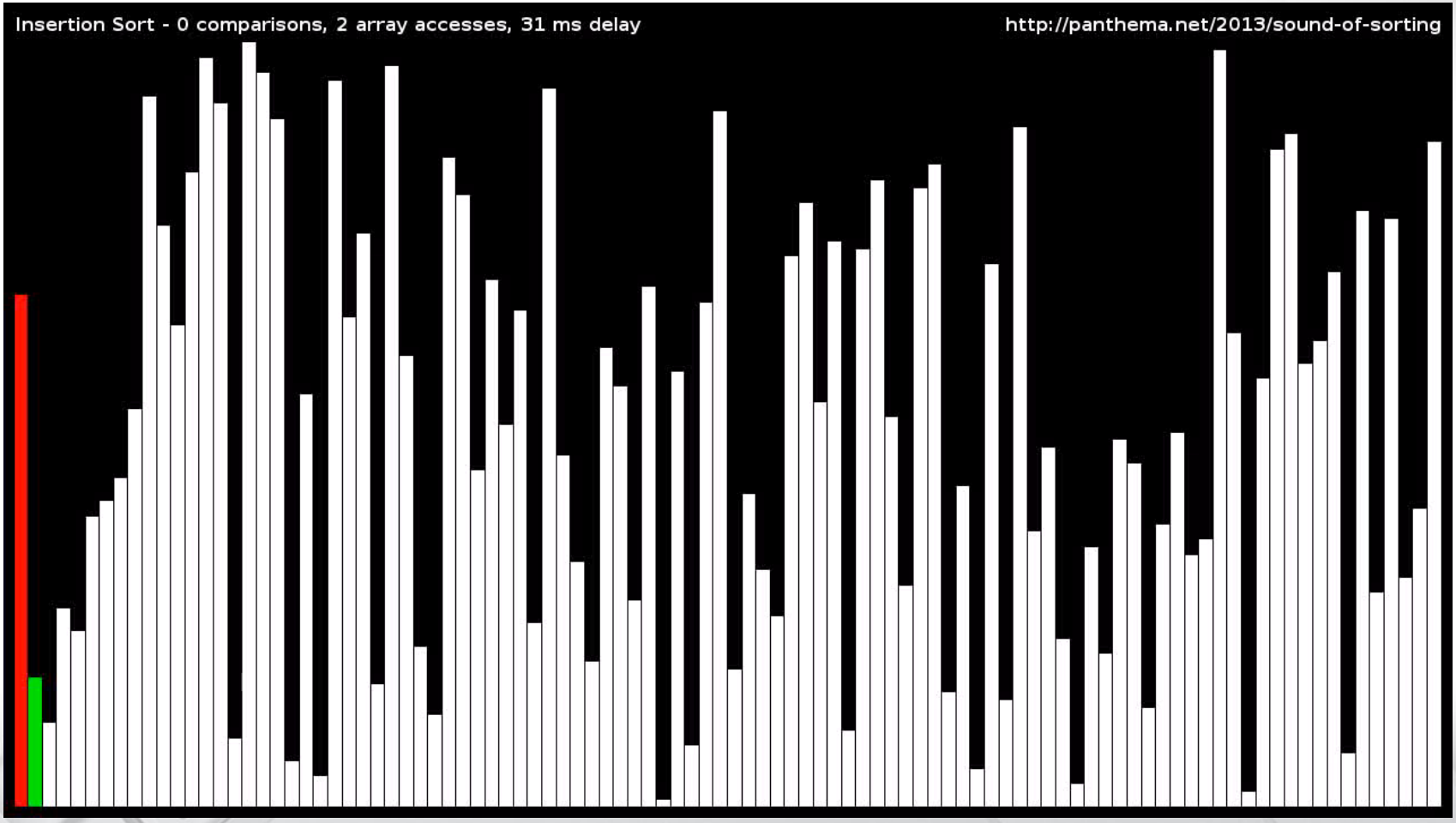
Insertion Sort

- At this point there's an imaginary marker somewhere in the middle of the line.
- The elements to the left of this marker are *partially sorted*. This means that they are sorted among themselves; each one is greater than the element from left.
- However, the elements aren't necessarily in their final positions because they may still need to be moved when previously unsorted elements are inserted between them.



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Insertion Sort



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

QuickSort

Quicksort is an efficient sorting algorithm that selects a pivot element from the array and partitions the other elements into two smaller slices or windows, according to whether they are less than or greater than the pivot element.

- Subsequently, the two newly created slices are sorted recursively. The base cases of the recursion are array slices of size zero or one, which are already sorted.
- It's important to note that this sorting algorithm has a time complexity of $O(n \log n)$ on average and $O(n^2)$ for the worst case. However, the worst-case scenario is rare, and the algorithm performs well in practice.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

QuickSort

```
fun partitionHoare(arr: IntArray, low: Int, high: Int): Int {
    val pivot = arr[low]
    var start = low - 1
    var end = high + 1
    while (true) {
        do {
            start++
        } while (arr[start] < pivot)
        do {
            end--
        } while (arr[end] > pivot)
        if (start >= end) {
            return end
        }
        val temp = arr[start]
        arr[start] = arr[end]
        arr[end] = temp
    }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

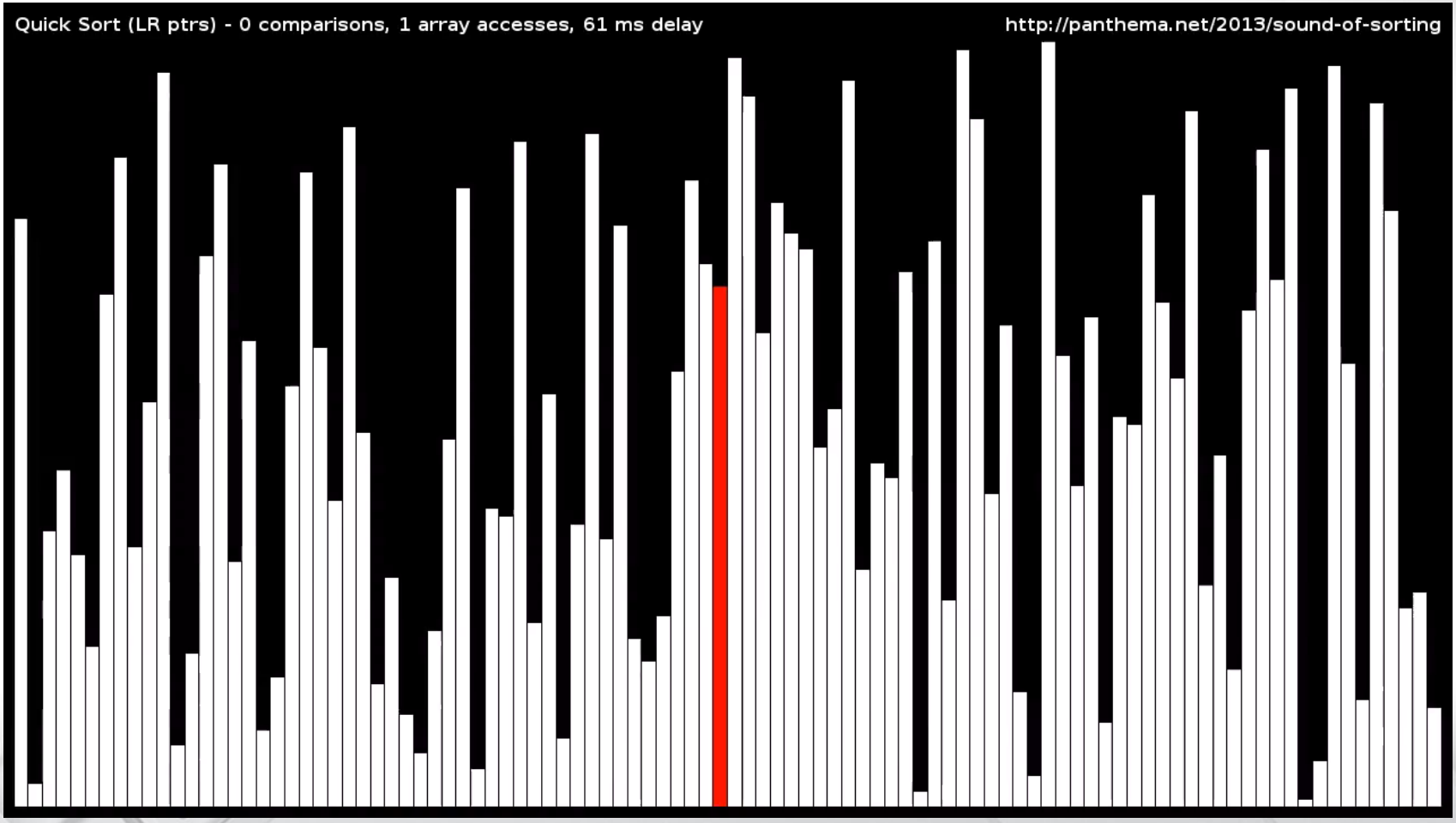
QuickSort

```
fun quickSortHoare(
    arr: IntArray,
    low: Int = 0,
    high: Int = arr.size - 1) {

    if (low < high) {
        val pivot = partitionHoare(arr, low, high)
        quickSortHoare(arr, low, pivot)
        quickSortHoare(arr, pivot + 1, high)
    }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

QuickSort

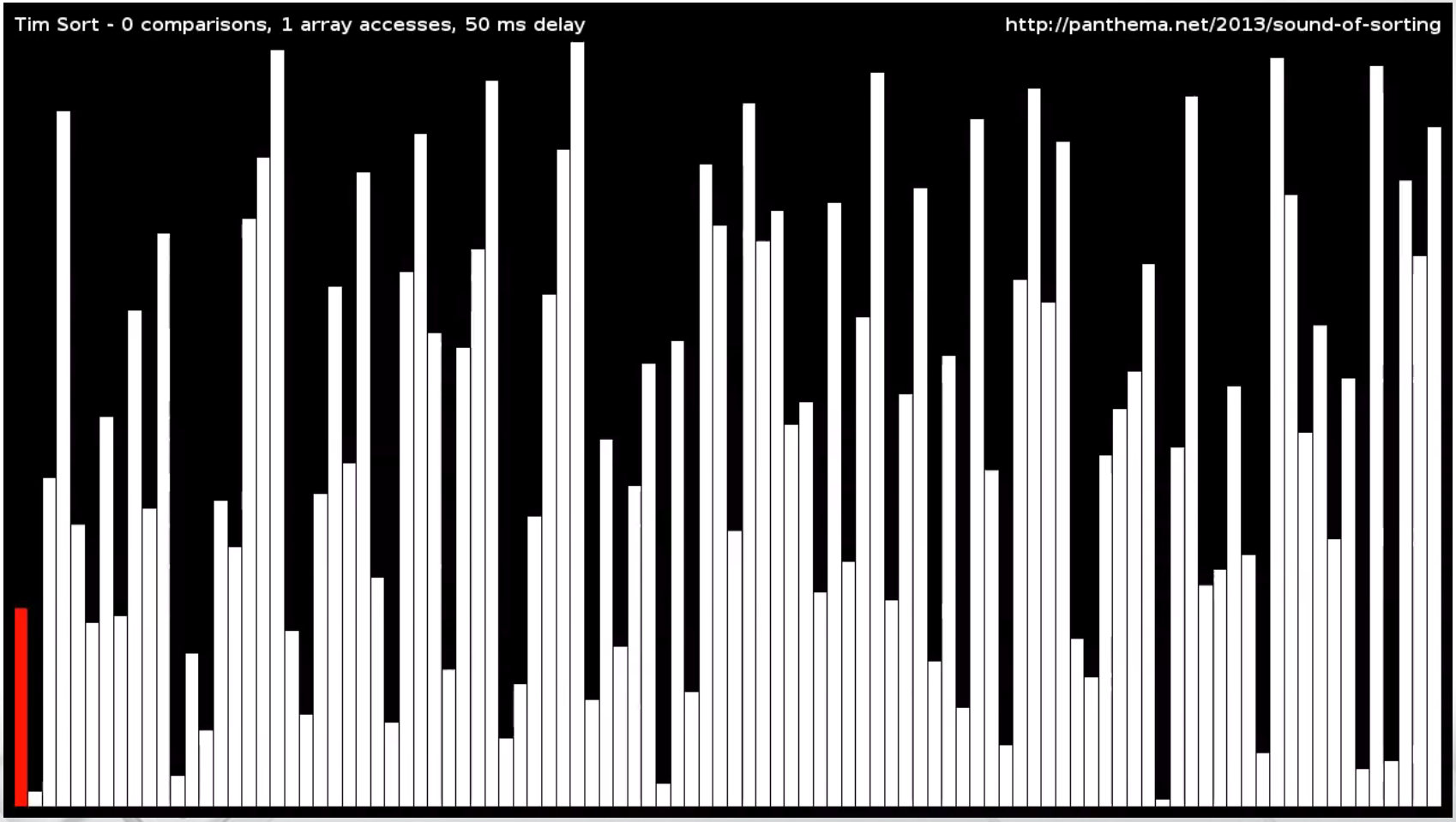


Quasilinear time

- Another common time complexity you'll encounter is **quasilinear time *O(n log n)***.
- Algorithms in this category perform worse than linear time but dramatically better than quadratic time. They are among the most common algorithms you'll deal with.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

TimSort



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Questions?

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Data Structures and Organization

(p.2 – Time complexity / Arrays)

Yevhen Berkunskyi,
NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua