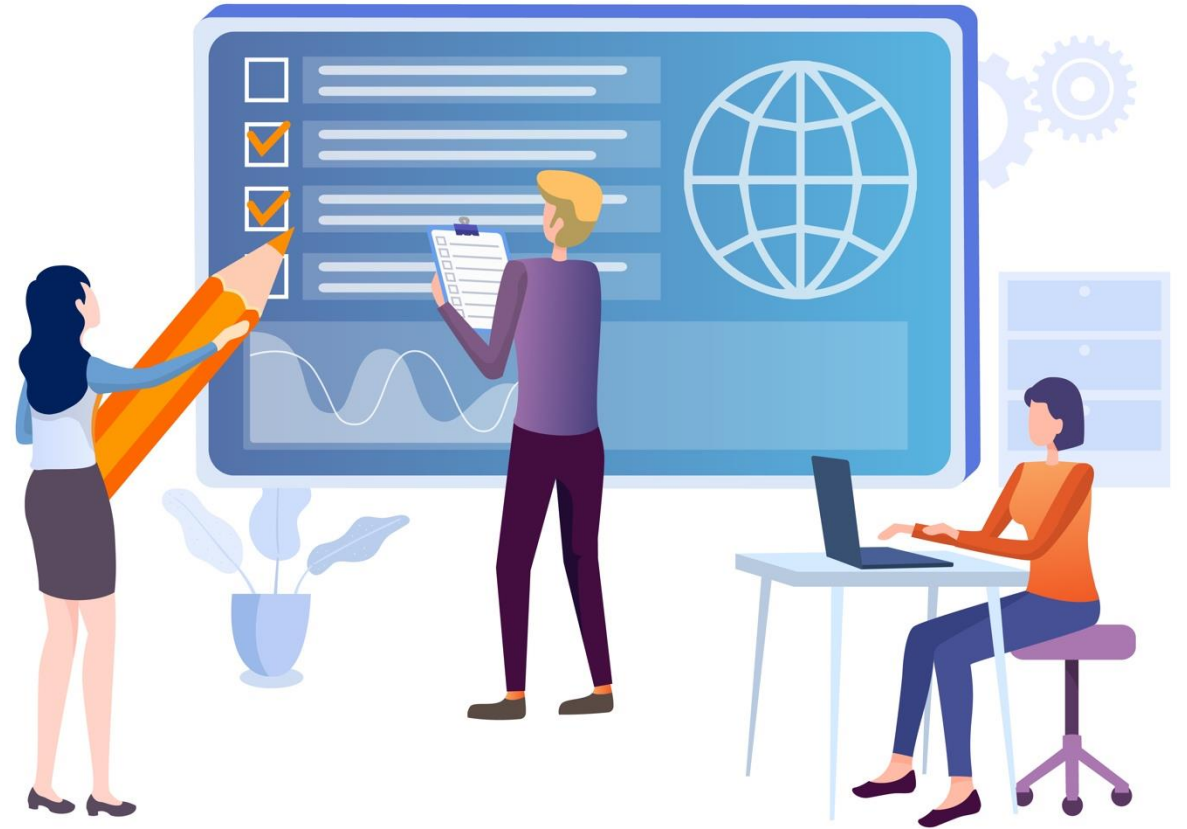


2. Introduction to algorithms

Quiz of the day

Get ready!



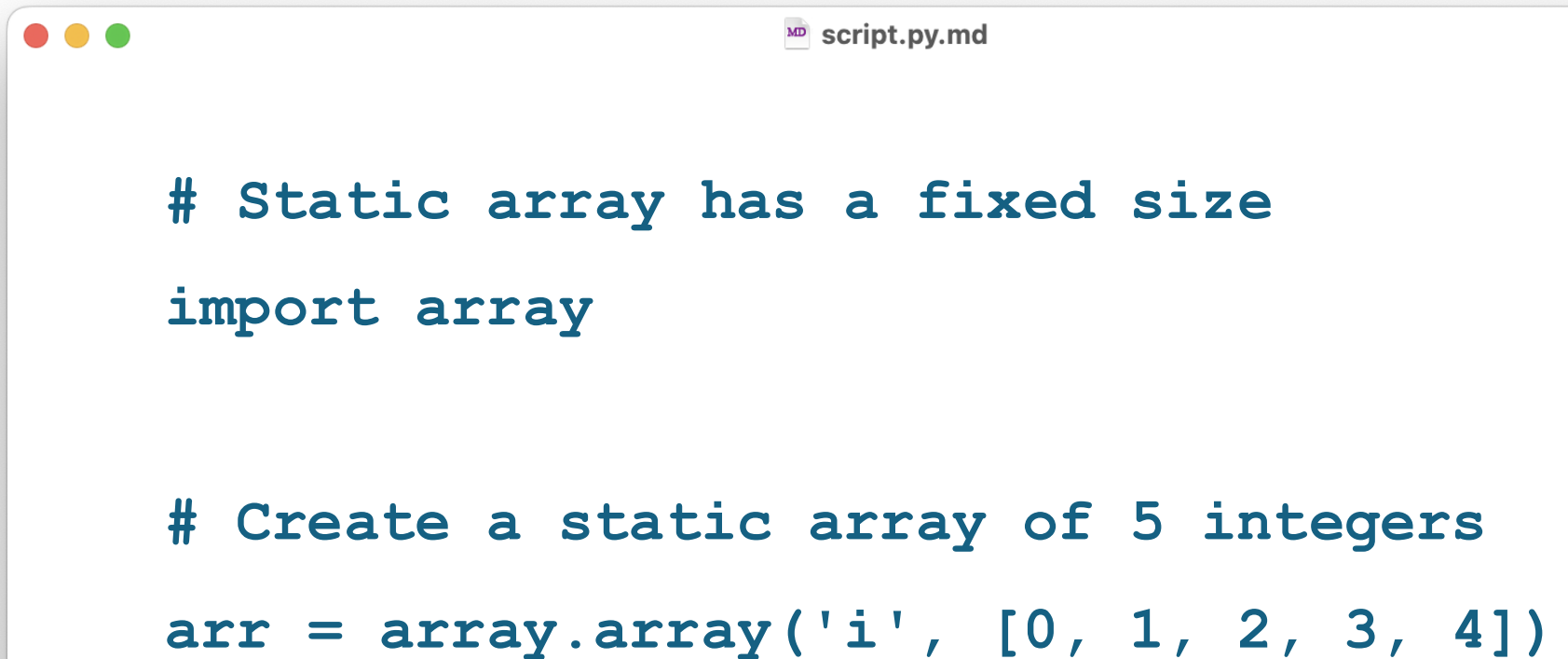
Agenda

- ▶ Algorithms for problem-solving!
 - Searching
- ▶ Computational complexity (second part)
- ▶ Sorting (bubble sort, merge etc.)
- ▶ **Race lab!**
 - Join me in 404-405 & online



Did you know?

- ▶ In Python, the built-in list type is implemented as a dynamic array with automatic resizing.



```
# Static array has a fixed size
import array

# Create a static array of 5 integers
arr = array.array('i', [0, 1, 2, 3, 4])
```

Did you know?

▸ **Static array:**

- Fixed size once declared

▸ **Dynamic array:**

- Starts with a certain amount of space (capacity).
- When you add more elements than it can hold, it **automatically resizes** — usually by **doubling** the capacity.
- Internally, it copies all elements into a **new, larger block of memory**.

Matrix Multiplication Performance Comparison (500×500 Matrices)

Method	Time (seconds)
Pure Python (loops)	20.82
NumPy <code>@</code> operator	0.15

- Matrix multiplication using NumPy is over 100x faster than using nested for loops
- A significant portion of NumPy's performance advantage does come from using static arrays, but that's just one part of a broader picture (~5x faster)


Properties of Python lists

Operation	Average Time Complexity
<code>append(item)</code>	$O(1)$ <i>amortized</i>
<code>pop()</code>	$O(1)$
<code>pop(0)</code>	$O(n)$ ← ⚠ shifts items
<code>insert(index, val)</code>	$O(n)$
<code>del list[i]</code>	$O(n)$
<code>list[i]</code> access	$O(1)$

* Amortized time is the average time per operation over a sequence of operations — even if some individual operations are expensive.

Hack: Process in reverse

- ▶ If you're removing elements from a list, starting from the back avoids index shifting



```
for i in range(len(my_list) - 1, -1, -1):  
    if my_list[i] == "delete":  
        del my_list[i]
```

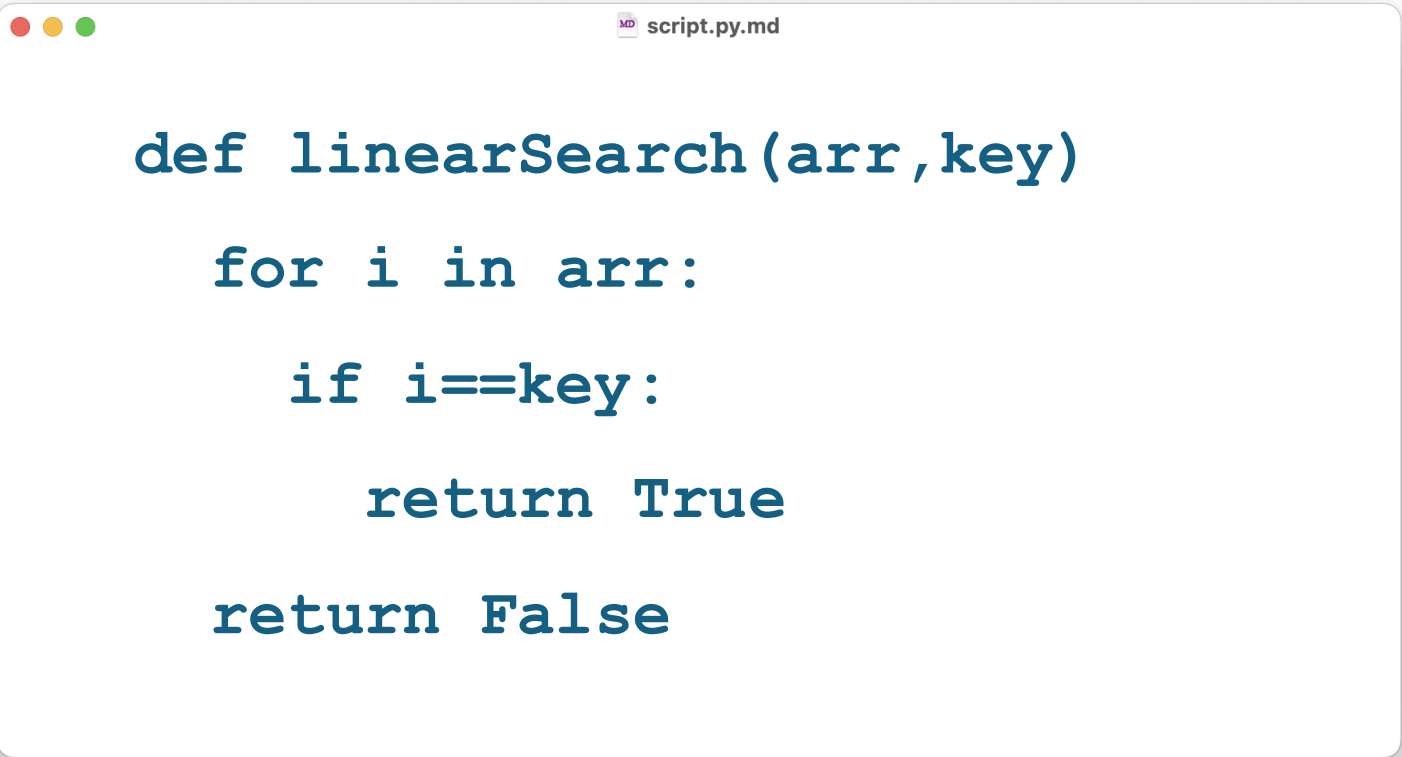

Hack: Two-pointer technique 😊

- ▶ Comparing elements, from both ends is efficient!
 - ▶ Instead of checking every pair or looping over the whole structure multiple times, you check two elements in one pass

```
script.py.md

left, right = 0, len(word) - 1
while left < right:
    if word[left] != word[right]:
        return False
    left += 1
    right -= 1
```

Linear Search algorithm



```
def linearSearch(arr,key)
    for i in arr:
        if i==key:
            return True
    return False
```

- An array of elements and a key.
- Does the key exist in the array?

Let's see an example

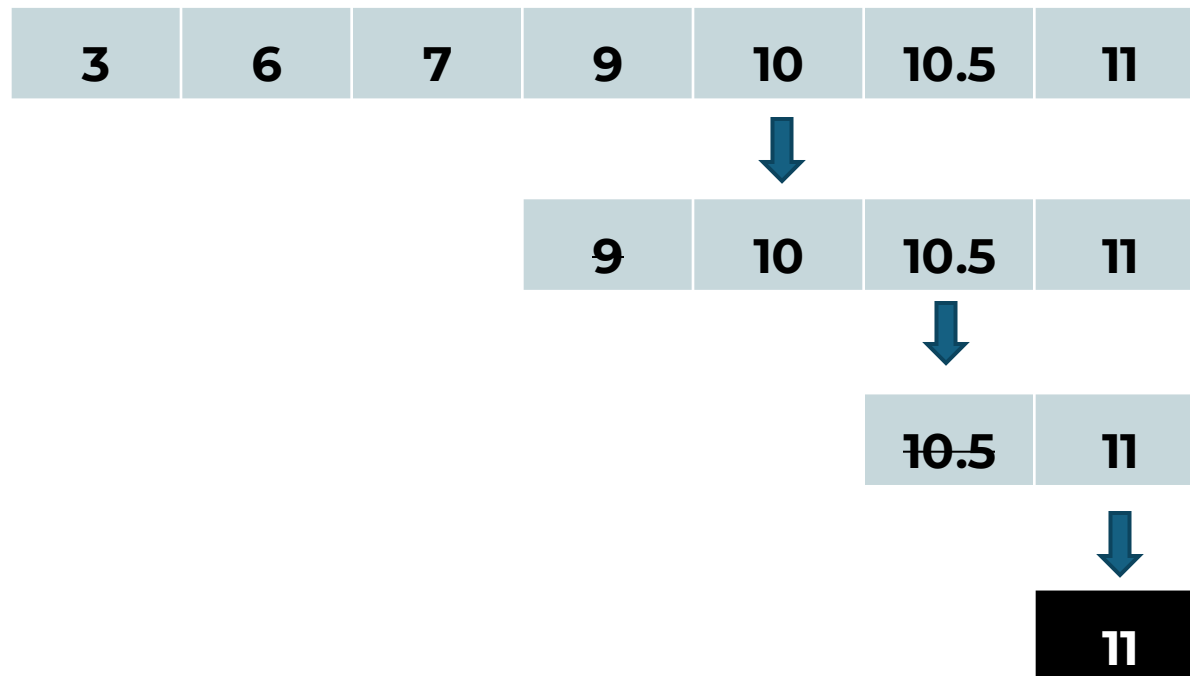
- ▶ $A = [4, 9, 12, 16, 18], \text{key}=6$
- ▶ In principle, there are n comparisons before we can say "no".
 - We will spend five operations to search for a key.
 - We care about the worst-case scenario: 5 operations = $O(5) = O(n)$.
- ▶ What about this case?
 - $B = [4, 9, 12, 16, 18], \text{key}=4$
 - Still, we care about the worst-case scenario... $O(n)$

Group quiz

- ▶ You just opened a new deck of cards!
 - You removed the two jokers
 - You have 52 cards in order!
 - What is the time complexity to search for a King of hearts? **$O(n)$ / Position 39!**
- ▶ What if you don't know? **Linear search (n)**
- ▶ Can you do it better? **Yes – Deck is human-friendly ordered**

Can we do better?

► $A = [3, 6, 7, 9, 10, 10.5, 11]$, $\text{key}=11$



The search space is repeatedly divided in half during each iteration...

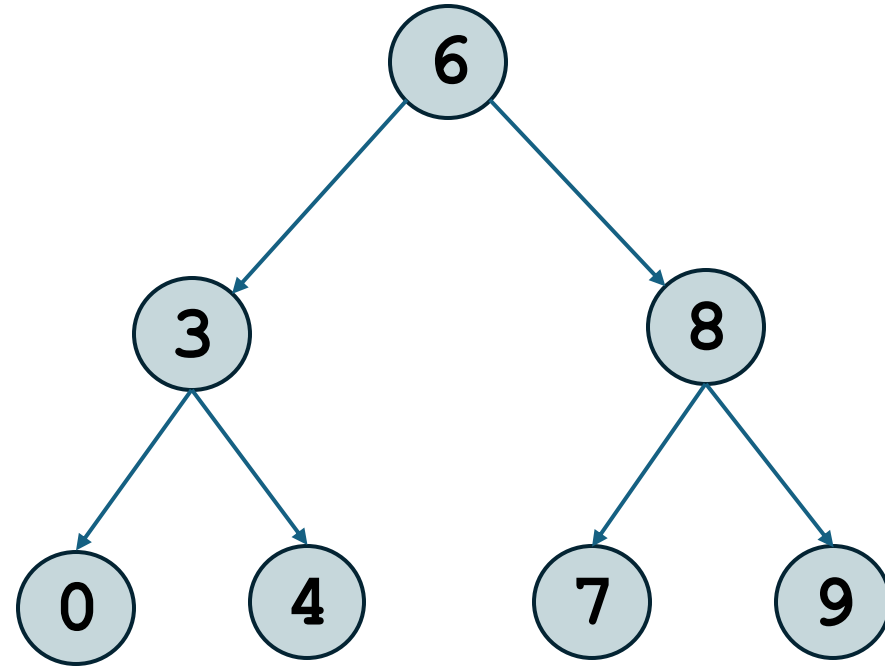
$O(\log n)$

Binary search example

[0, 3, 4, 6, 7, 8, 9]

↑ ↑ ↑
L **M** **R**

Binary Search



Binary Search Tree

Binary search

- ▶ Search in a **sorted array** by repeatedly dividing the search interval in half.
 - Begin with an interval covering the whole array.
 - If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
 - Otherwise, narrow it to the upper half.

```
def binary_search(arr, target):  
    left, right = 0, len(arr) - 1  
    while left <= right:  
        mid = left + (right - left) // 2  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] < target:  
            left = mid + 1  
        else:  
            right = mid - 1  
    return -1
```

target=13

0	3	6
[10,13,14,16,17,18,19]		
↑	↑	↑
L	M	R

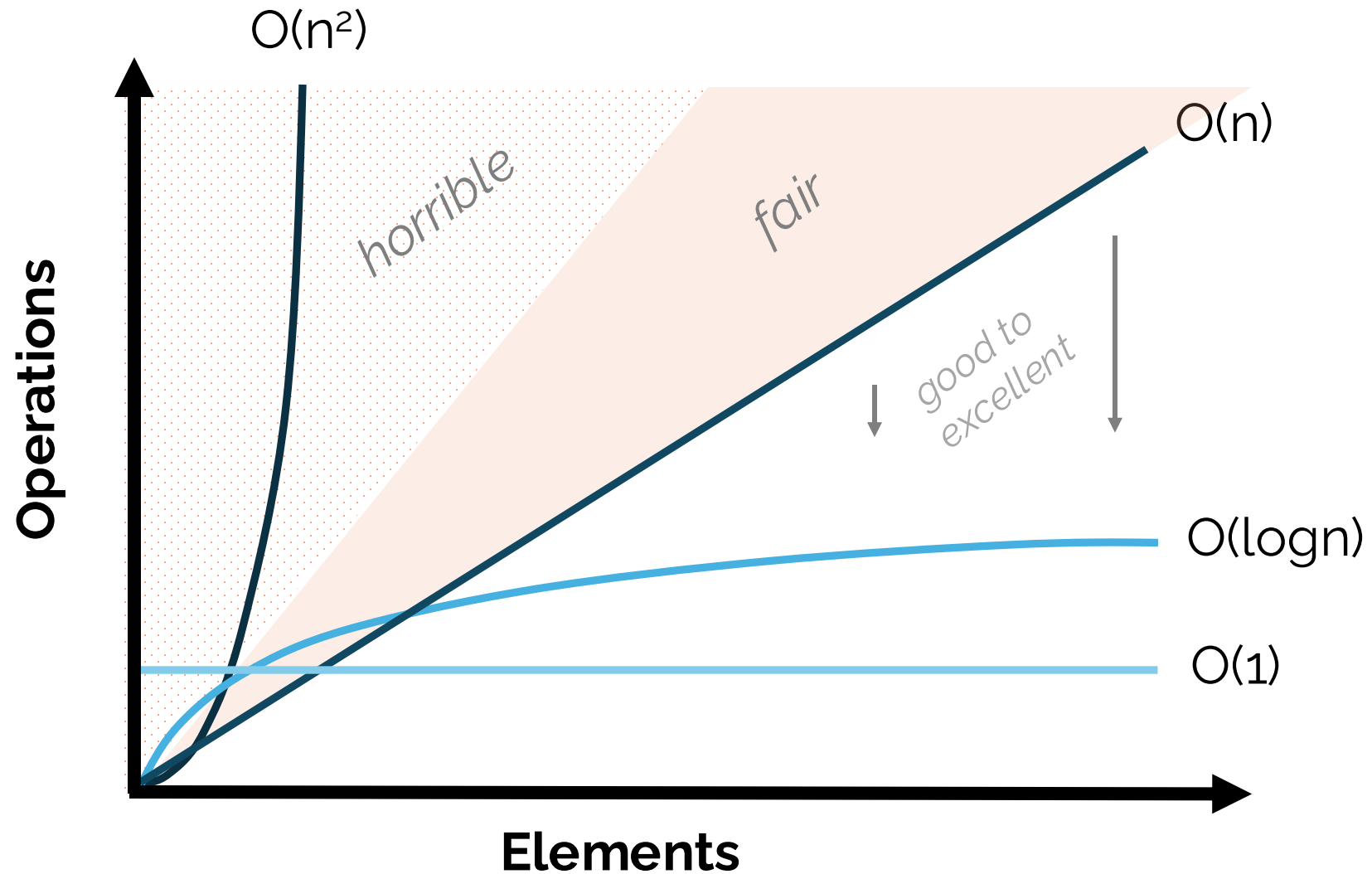
Quiz

- ▶ How many iterations to find 23? **3**

[2, 5, 8, 12, 16, 23, 38, 56, 72]

— — —

Big-O Complexity Chart



Group quiz

► Write down all the possible permutations of ABC?

- **ABC**
- **ACB**
- **BAC**
- **BCA**
- **CAB**
- **CBA**

Can you think of a mathematical formula to generalise this example?

$$3! = 1 * 2 * 3 = 6$$

$O(N!)$ [Factorial time]

- ▶ Generating all possible permutations of a set of elements is inherently factorial.
 - Consider a set of N elements
 - There are $N!$ permutations to be generated

Group quiz

- ▶ You have been tasked with guessing a password.
 - The password includes three digits, each of which can be either 1 or 0. A digit may appear multiple times or not at all in the combination. Write down all the possible combinations!
 - Can you think of a mathematical formula to generalise this example?

$$2^3 = 8 = 2^n$$

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

$O(2^N)$ [Exponential time]

- ▶ All possible combinations!
 - Scenario: In security testing or hacking, generating all possible passwords up to a certain length, given a specific character set (brute-force password cracking).

Computational complexity

- ▶ Time

- Number of operations to complete a task

- ▶ Space

- Amount of extra memory used for completing a task



Big O (worst case)

- ▶ **$O(1)$ [Constant time]**

- Accessing an element in an array by index.

- ▶ **$O(n)$ [Linear time]**

- Linear search in an array.

- ▶ **$O(\log n)$ [Logarithmic time]**

- Binary search in a sorted array.

- ▶ **$O(n^2)$ [Quadratic time]**

- Sort data by repeatedly swapping the adjacent elements if they are in the wrong order.

Quiz

► Write down all the possible permutations of ABC?

- **ABC**
- **ACB**
- **BAC**
- **BCA**
- **CAB**
- **CBA**

Can you think of a scenario where generating all possible permutations could be useful?

Software testing

Can you think of a mathematical formula to generalise this example?

$$3! = 1 * 2 * 3 = 6$$

$O(N!)$ [Factorial time]

- ▶ Generating all possible permutations of a set of elements is inherently factorial.
 - Consider a set of N elements
 - There are $N!$ permutations to be generated

Quiz

- ▶ You have been tasked with guessing a password.
 - The password includes three digits, each of which can be either 1 or 0. A digit may appear multiple times or not at all in the combination. Write down all the possible combinations!
 - Can you think of a mathematical formula to generalise this example?

$$2^3 = 8 = 2^n$$

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

$O(2^N)$ [Exponential time]

- ▶ All possible combinations!
 - Scenario: In security testing or hacking, generating all possible passwords up to a certain length, given a specific character set (brute-force password cracking).

Question

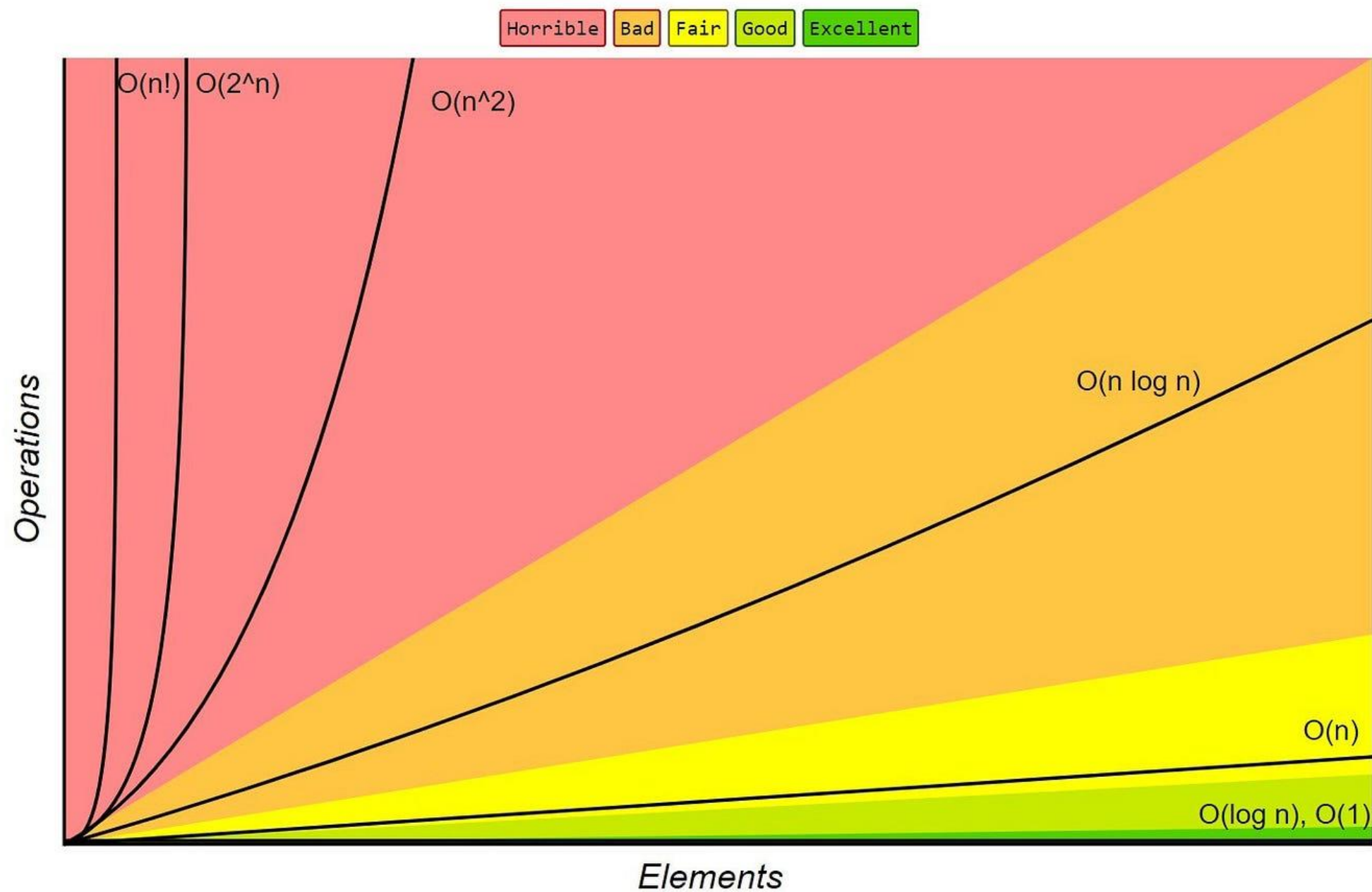
- What is worse, factorial or exponential?

$$2^5 = 2 * 2 * 2 * 2 * 2 = 32$$

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

- By the time you reach $n=10$,
 - $10! = 3,628,800$
 - $2^{10} = 1,024$
- The factorial function grows much faster!

Big-O Complexity Chart



Group discussion

- ▶ What is the most efficient way to swap elements of a list?

- `alist = [2, 4, 6, 8, 12, 14]`

- `swap(alist, 6, 8) → [2, 4, 8, 6, 12, 14]`

`[2, 4, 8, 6, 12, 14]`

↑ ↑ ↑ ↑ ↑ ↑

Time complexity? $O(n/2) \rightarrow O(n)$

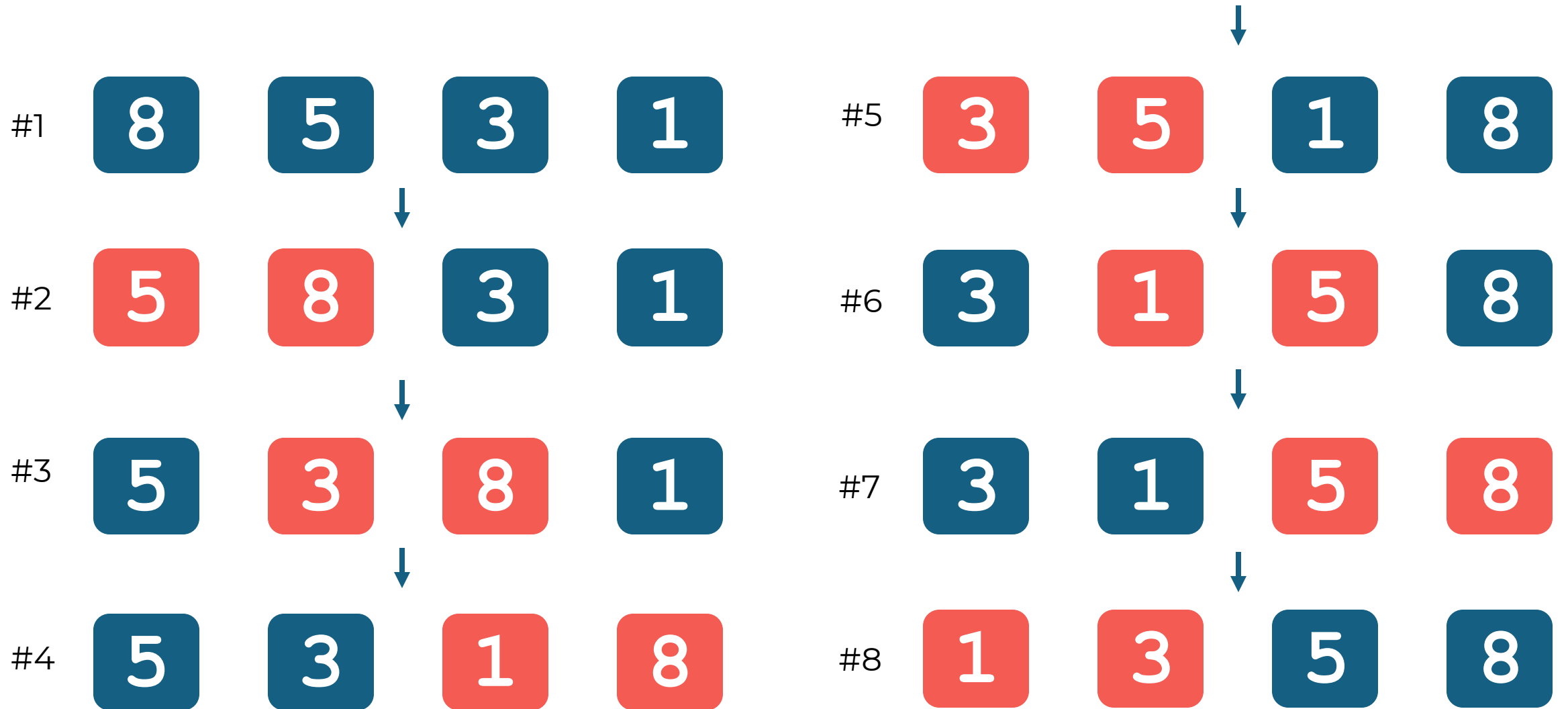
Space complexity? $O(1)$ 😊

Sorting methods

What is sorting?

- ▶ Sorting refers to ordering data in an increasing or decreasing manner according to some linear relationship among the data items.
- ▶ Bubble sort:
 - Repeatedly steps through the list to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order.

Bubble sort



What is the complexity of bubble sort?

► Time:

- Repeatedly traversing the array and comparing adjacent items, swapping them if they are in the wrong order.
 - Each complete pass through an array of length n compares elements in pairs, leading to $n-1$ comparisons (first pass), $n-2$ (second), and so on, down to 1 comparison in the last pass.
 - $(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} \approx O(n^2)$

► Space:

- $O(1)$ – in-place swapping, no extra space is used 😊

Bubble sort



script

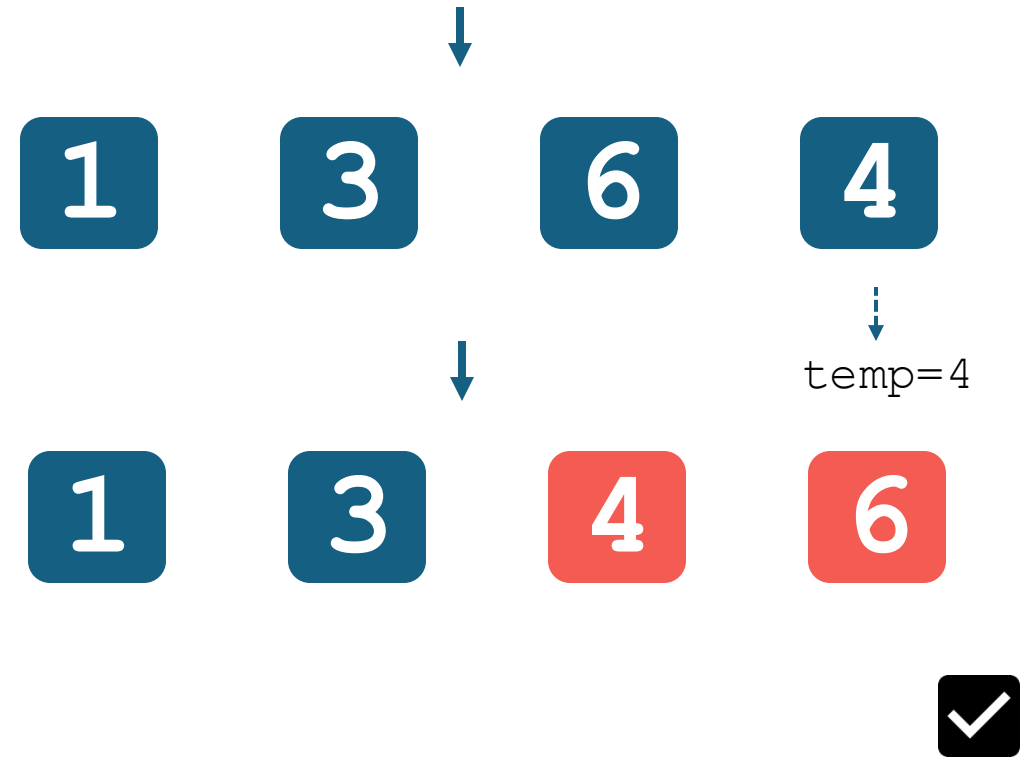
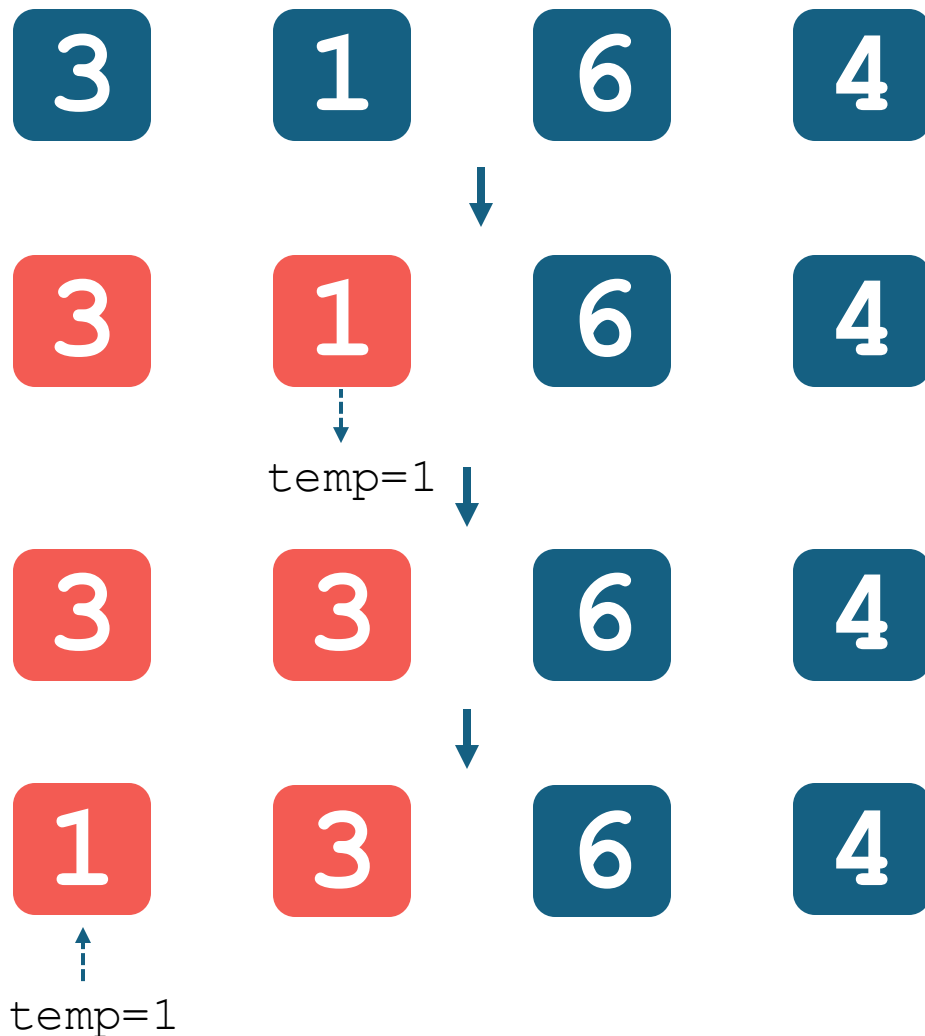
```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n-1-i):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]  
    return arr
```

Insertion sort

- ▶ Iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.
 - It is a stable sorting algorithm (elements with equal values maintain relative order).

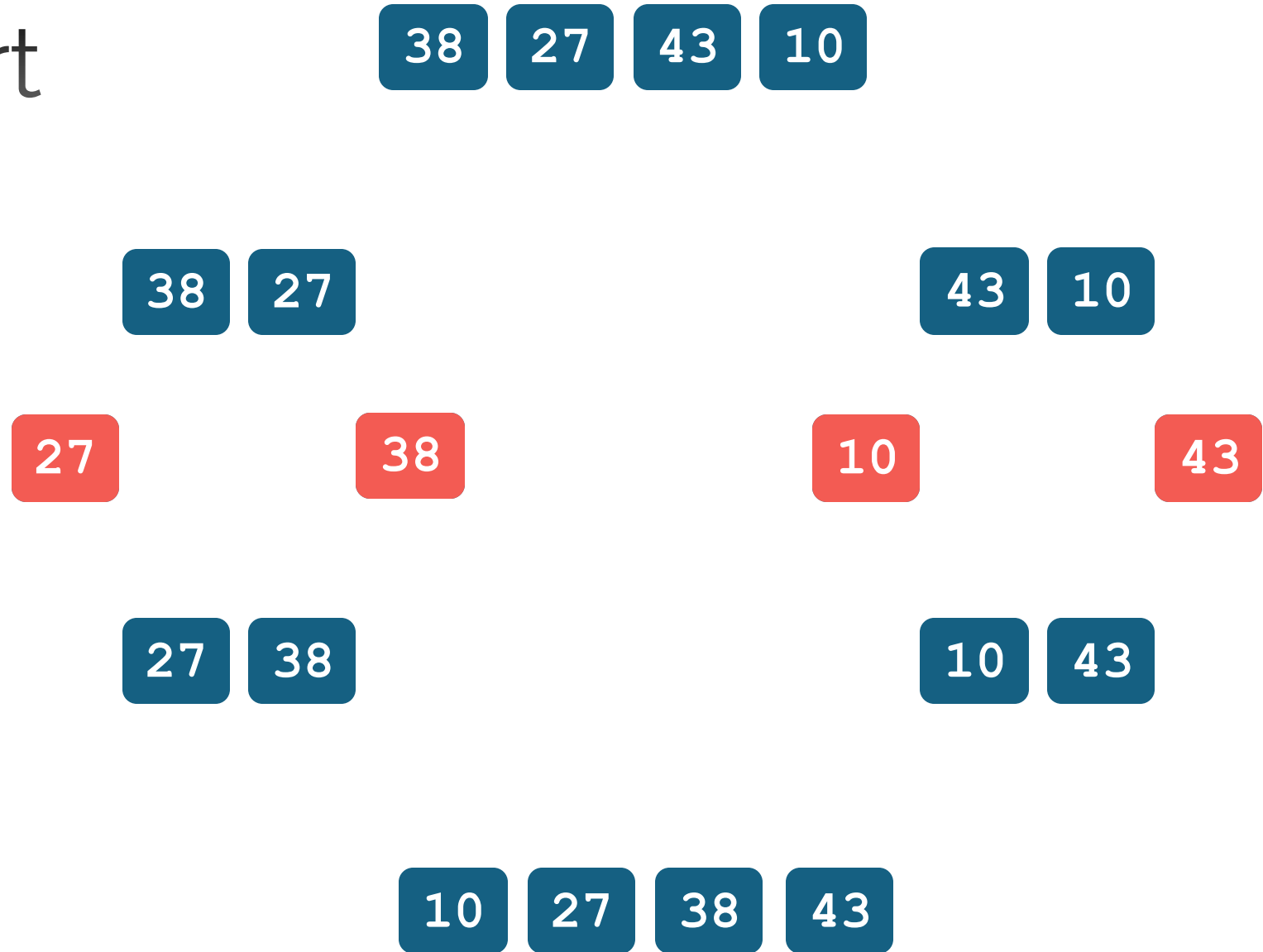


Insertion sort



Time: $O(n^2)$
Space: $O(1)$

Merge sort



Time: $O(n \log n)$

Space: $O(n)$

What is $O(n \log n)$?

► Merge Sort:

- Divides the array into two halves, sorts each half, and then merges the sorted halves.
- Each divide (and subsequent merge) step takes linear time, and the division happens $\log n$ times (the depth of the recursion tree).



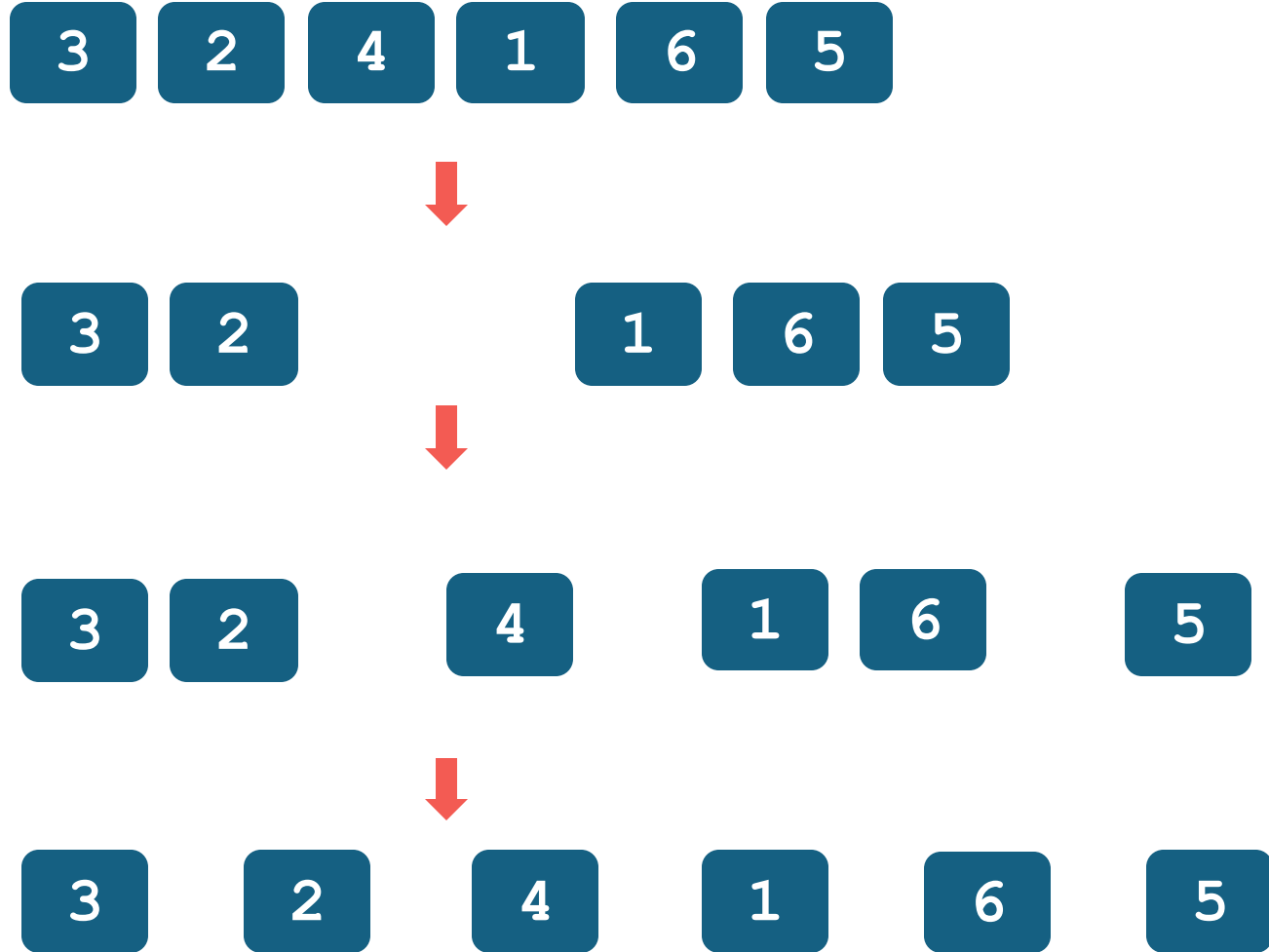
Quiz

- ▶ Using pen and paper sort the following numbers using merge sort!

3 2 4 1 6 5

- ▶ First, split the array into smaller sub-arrays until each sub-array has only one element.
- ▶ Start merging them back together in a sorted order.
- ▶ Merge the two sorted sub-arrays

Step 1: Split the Array



First, split the array into smaller sub-arrays until each sub-array has only one element.

Step 2: Merge and Sort



Step 3: Final merge

2 3 4

1 5 6



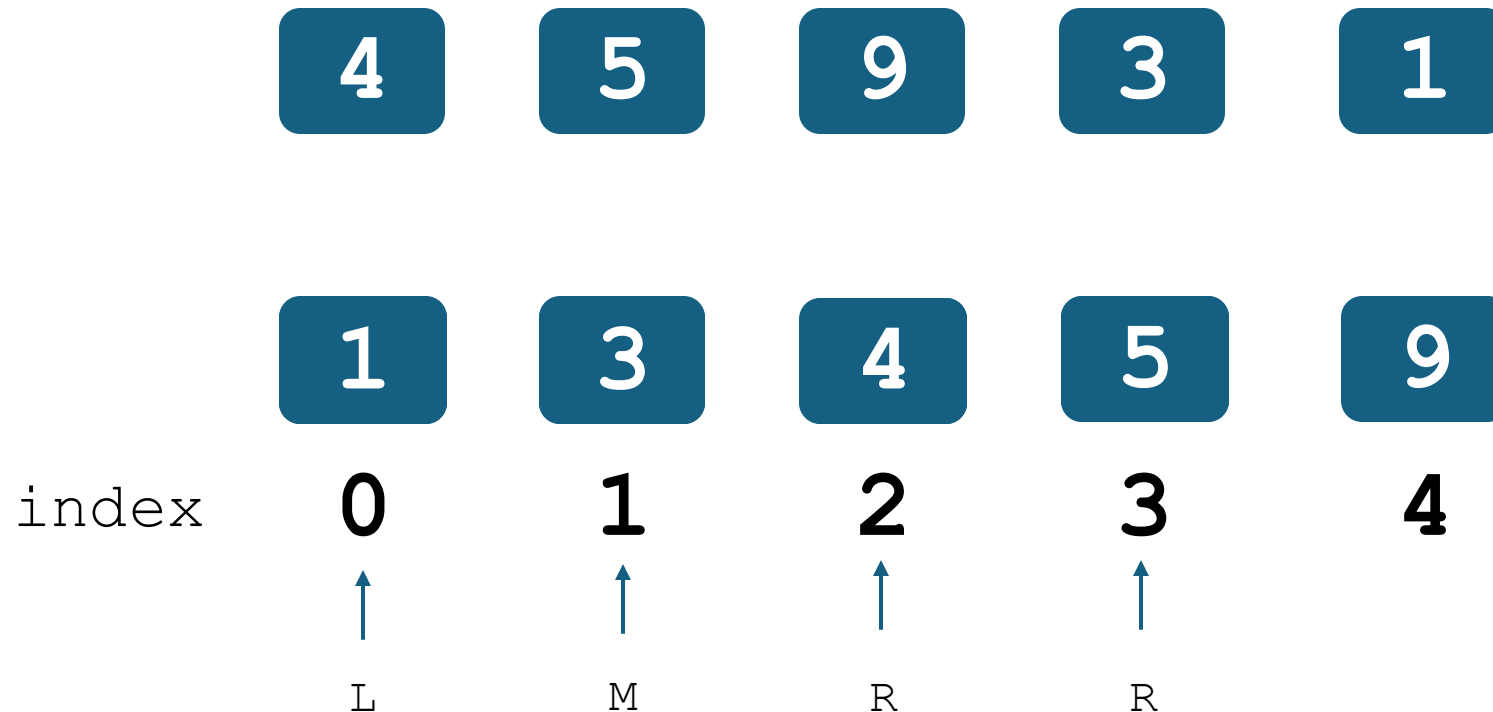
1 2 3 4 5 6

Tim sort

- ▶ Tim Sort is a hybrid sorting algorithm derived from merge sort and insertion sort. It was designed to perform well on many kinds of real-world data.
 - Tim Sort is the default sorting algorithm used by Python's `sorted()` and `list.sort()` functions.
- ▶ Time:
 - $O(n \log n)$
- ▶ Space
 - $O(n)$

TimSort uses binary search insertion

- Instead of using linear search to find the location where an element should be inserted, it uses binary search.



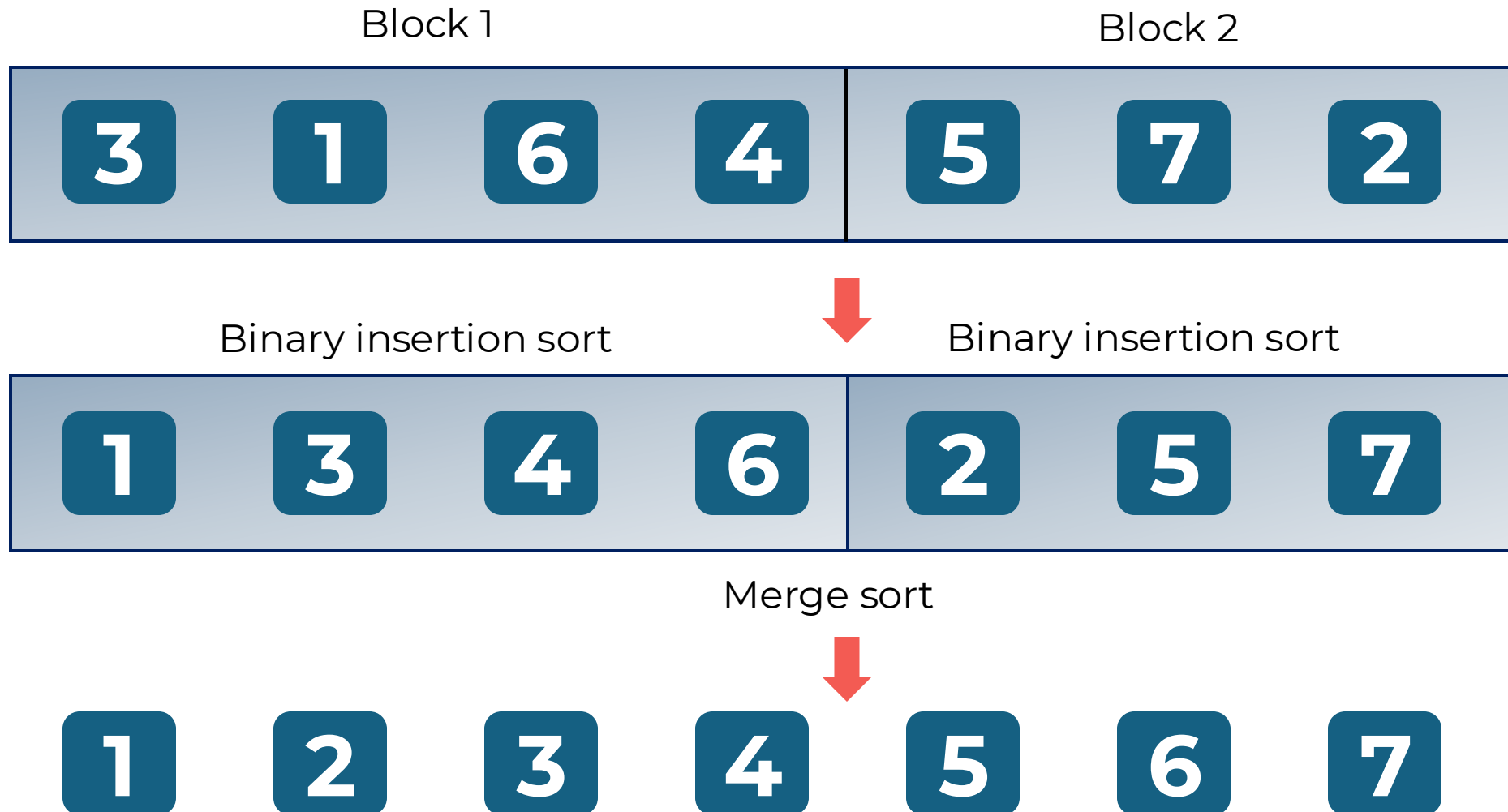
Tim sort steps

- ▶ Tim sort assumes that some data is already ordered...
- ▶ We call this a run:
 - **arr** = {5,7,8,9,3, ... }
 - **Run1** is {5,7,8,9}
- ▶ Each run has a minimum length of elements (min run):
 - **arr** = {8,12,3,9,14, ... }, with min run = 3
 - Run is {8,12,3}
 - Tim sort performs binary insertion, so the run becomes {3,8,12}

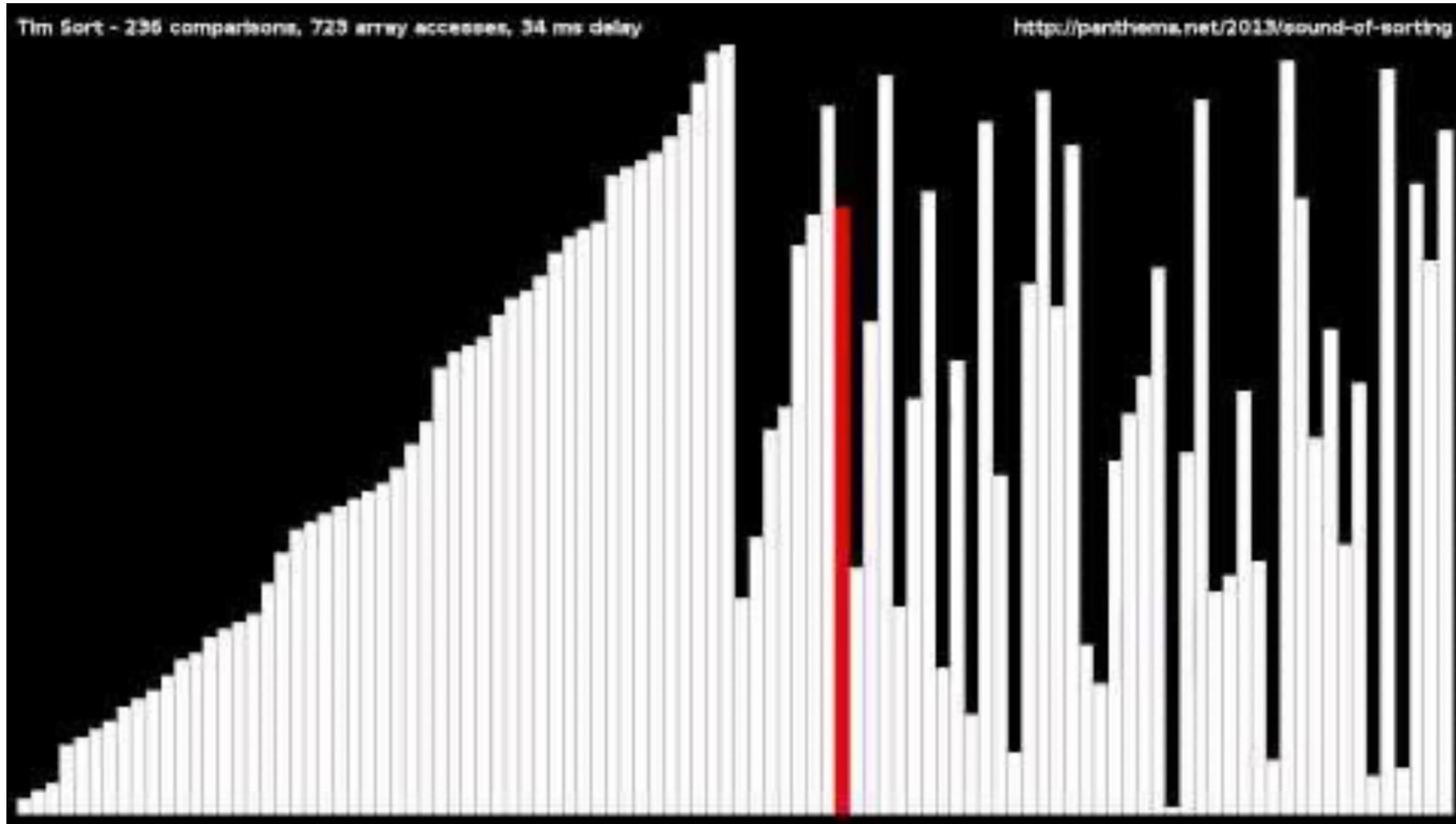
Tim sort steps

- ▶ Tim sort has some cool feature
 - If the min run is met, and the next element follows the same pattern (ascending order), then increase the size of the run...
 - `arr = {5,7,8,9}`, `min run = 3`
 - `run1 = {5,7,8} → {5,7,8,9}`
- ▶ If the run is in descending order, then blindly reverse!
 - `run1 = {8,4,2,1}` becomes `{1,2,4,8}`

The whole picture!



Tim sort: The fastest sort!



Let's compare them!

Algorithm	Time	Space
Linear search	$O(n)$	$O(1)$
Binary search	$O(\log n)$	$O(1)$
Bubble sort	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n)$
Tim sort	$O(n \log n)$	$O(n)$

Thank you!

- The lab starts soon!
(404-405)

O (no!)



Lab 2

Big Data Analytics

Lab activities



- ▶ Complete [lab 2 activities](#) and exercises.
 - To complete the exercises, you will need to [download the "rockyou.txt" from Kaggle](#).
- ▶ Use your preferred python IDE.