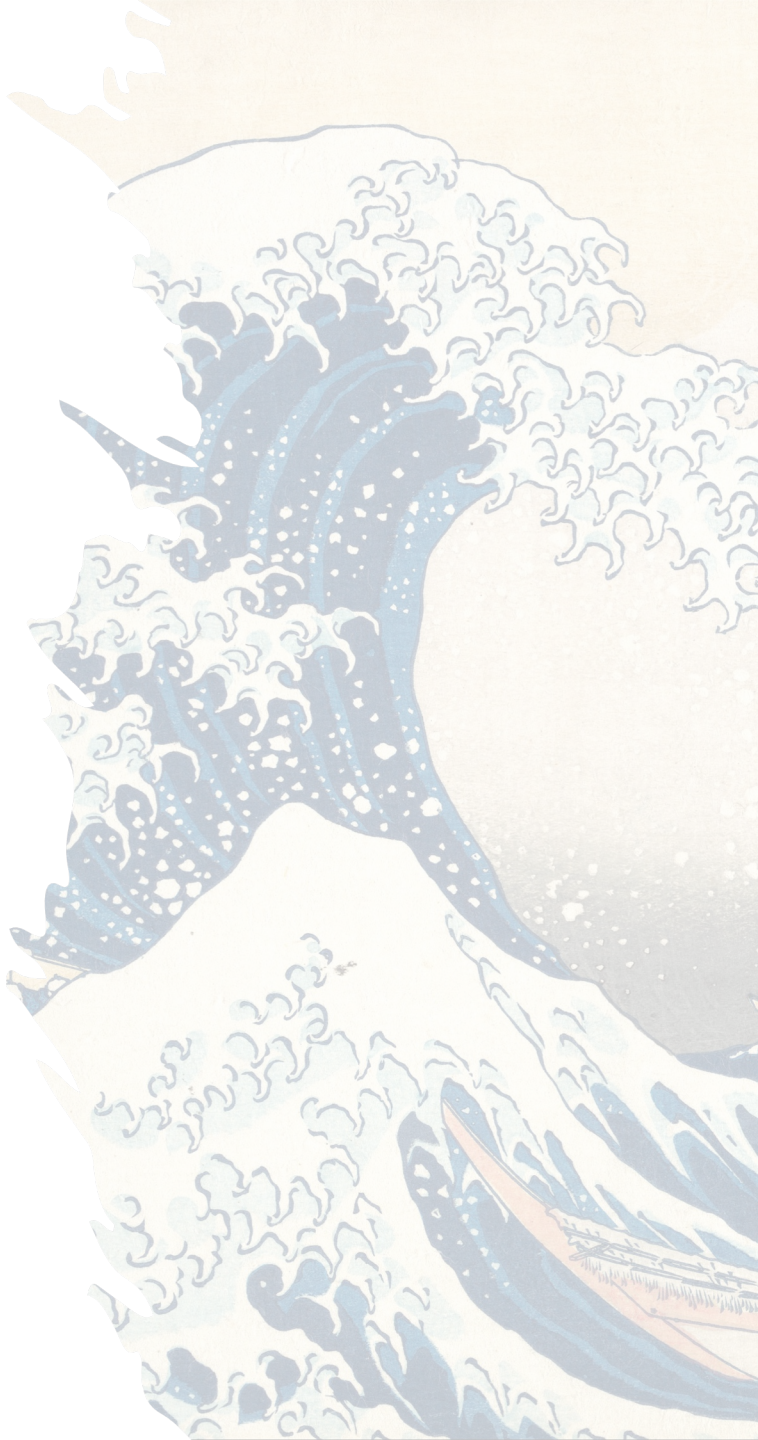
Stelios Sotiriadis

2. Introduction to algorithms

Agenda

- Algorithms for problem-solving!
- Computational complexity (second part)
- Sorting (bubble sort, merge etc.)
- Algorithmic classes:
  - Divide and conquer, Greedy and Dynamic programming
- **Race lab!**
  - Join me in 404-405 & online



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(logn) [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 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?

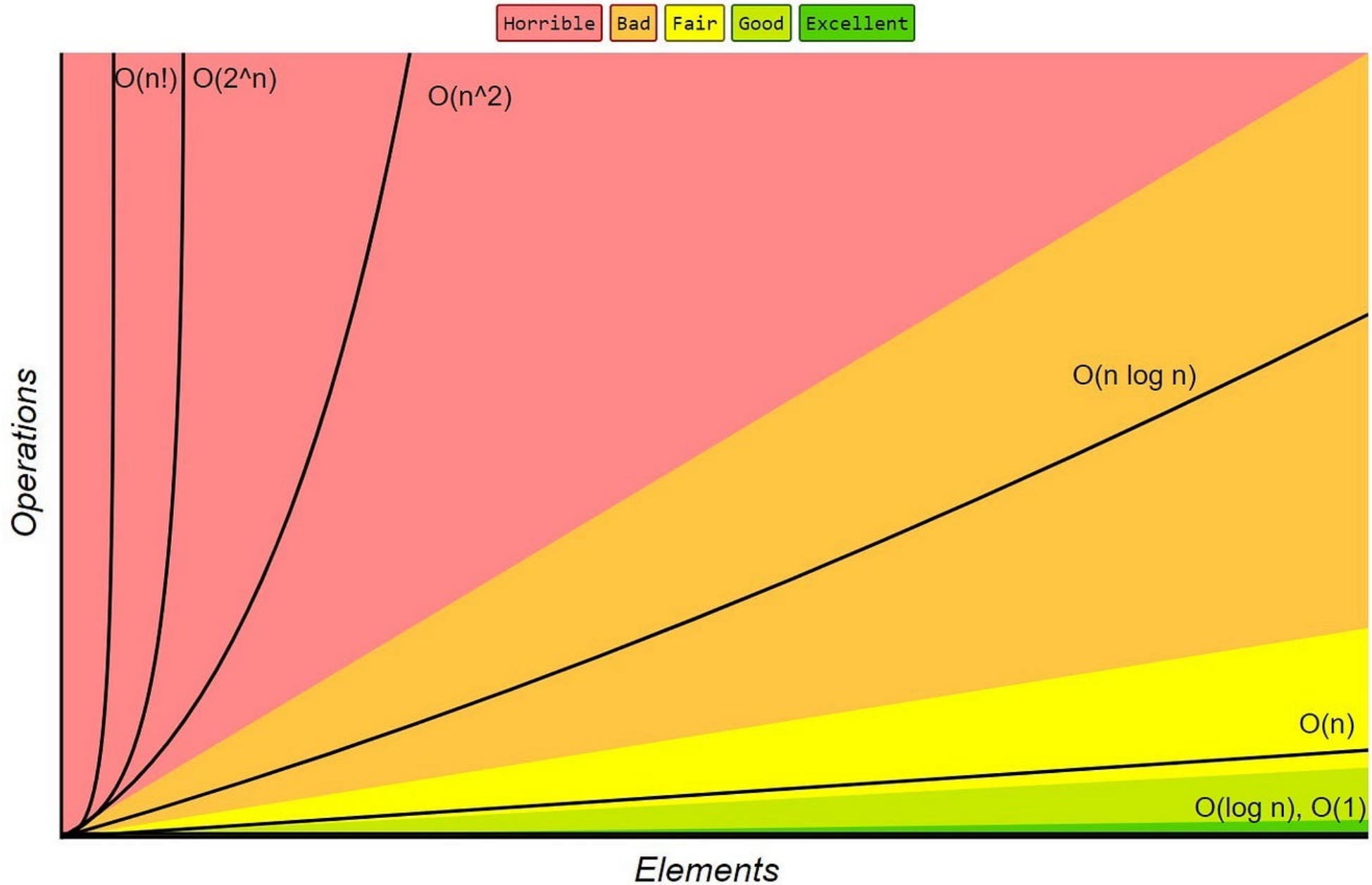
```
2^3 = 8 = 2^n
```

- 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).

Big-O Complexity Chart



Sorting methods

What is sorting?

- Sorting: an operation that segregates items into groups according to specified criteria.
- 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.
    - For each full pass through the array (which could be of length `n`), it compares elements in pairs, leading to `n-1` comparisons in the first pass, `n-2` in the second, and so on, down to `1` comparison in the last pass.
  - `O(n^2)`
- Space:
  - `O(1)` – in-place swapping, no extra space is used!

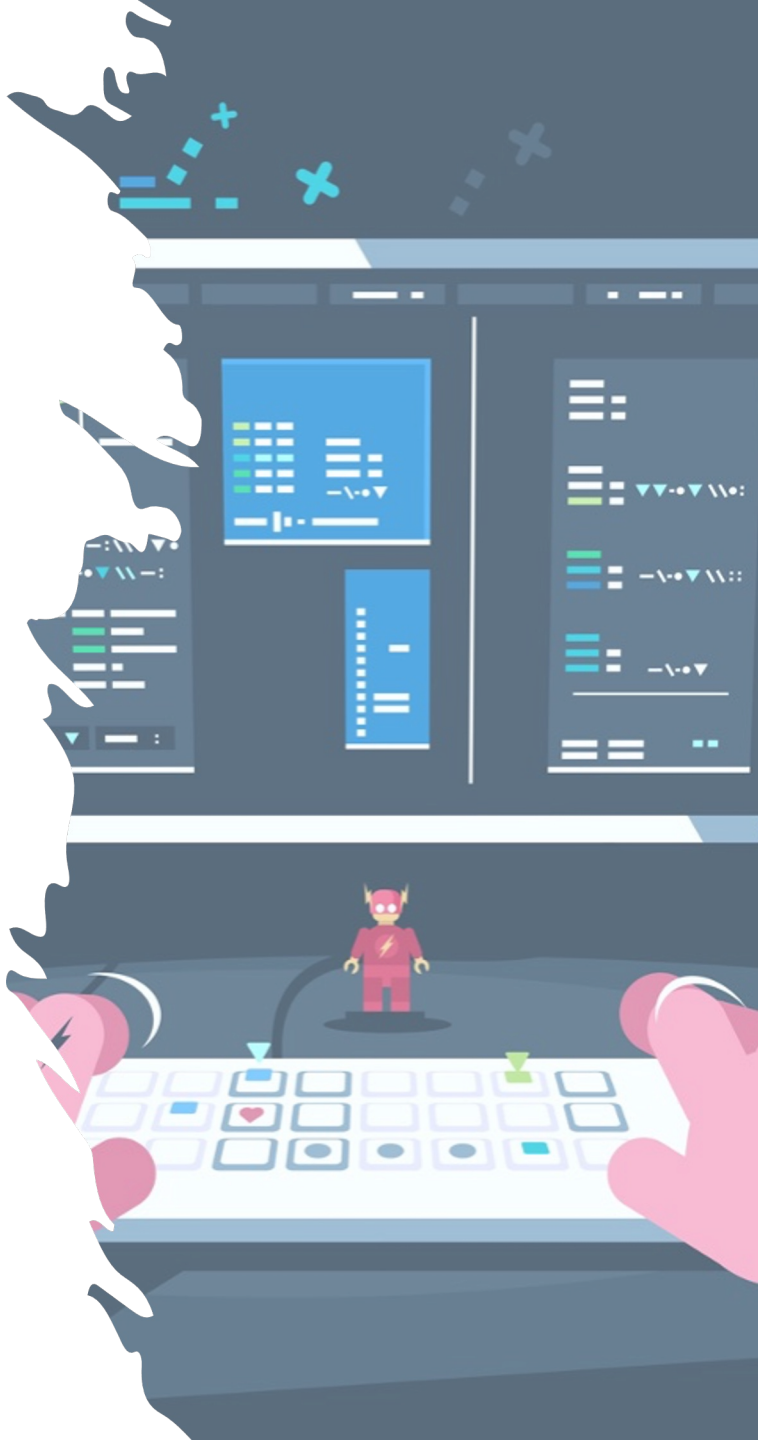
Bubble sort

script

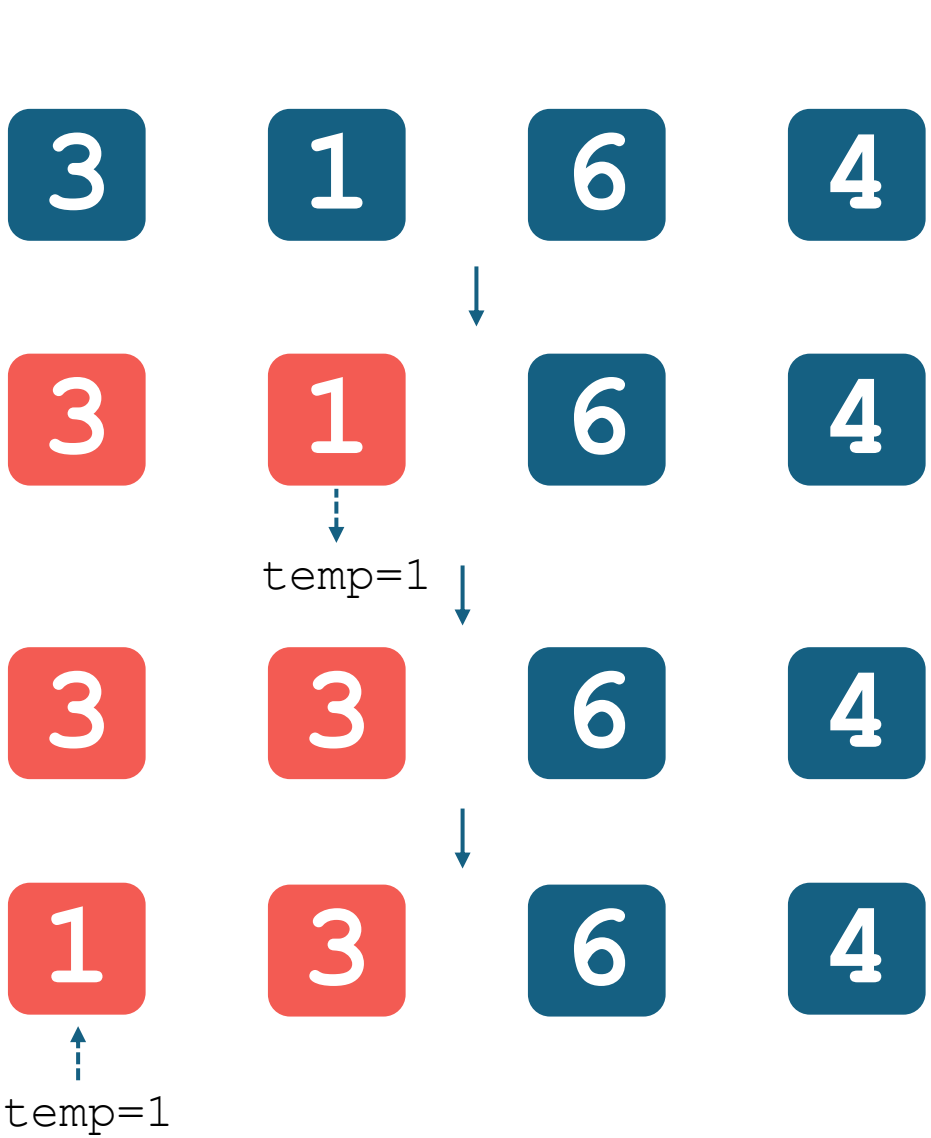
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

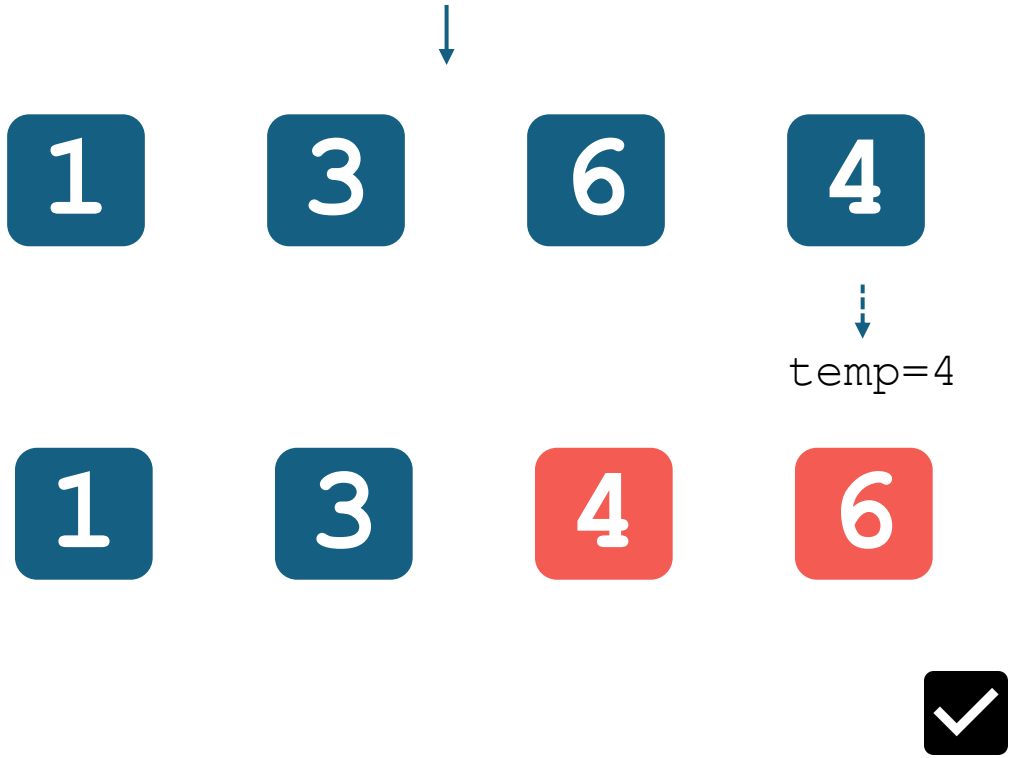
Insertion sort

- 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)



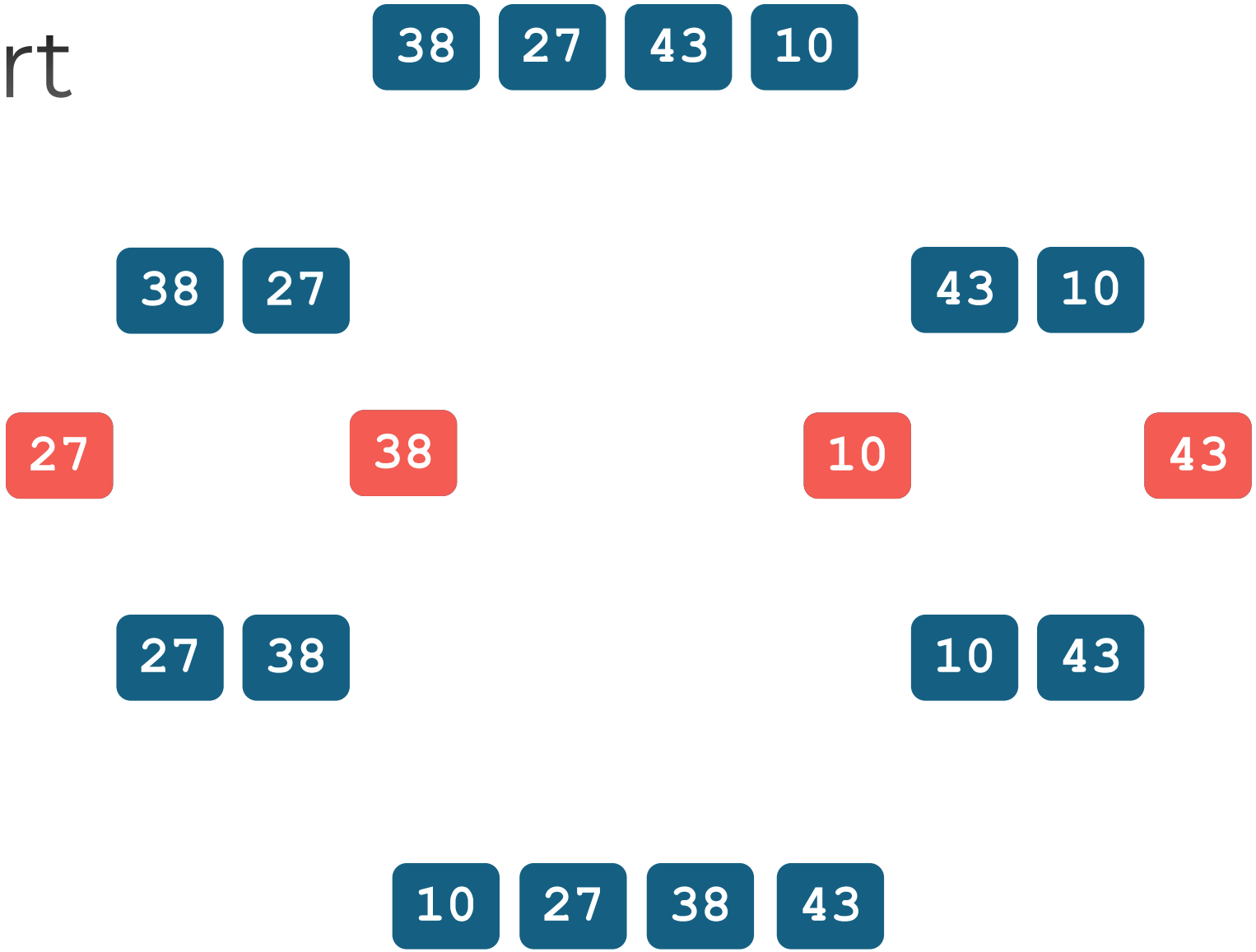
Insertion sort





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

Merge sort



Time: `O(nlogn)`
Space: `O(n)`

What is `O(nlogn)`?

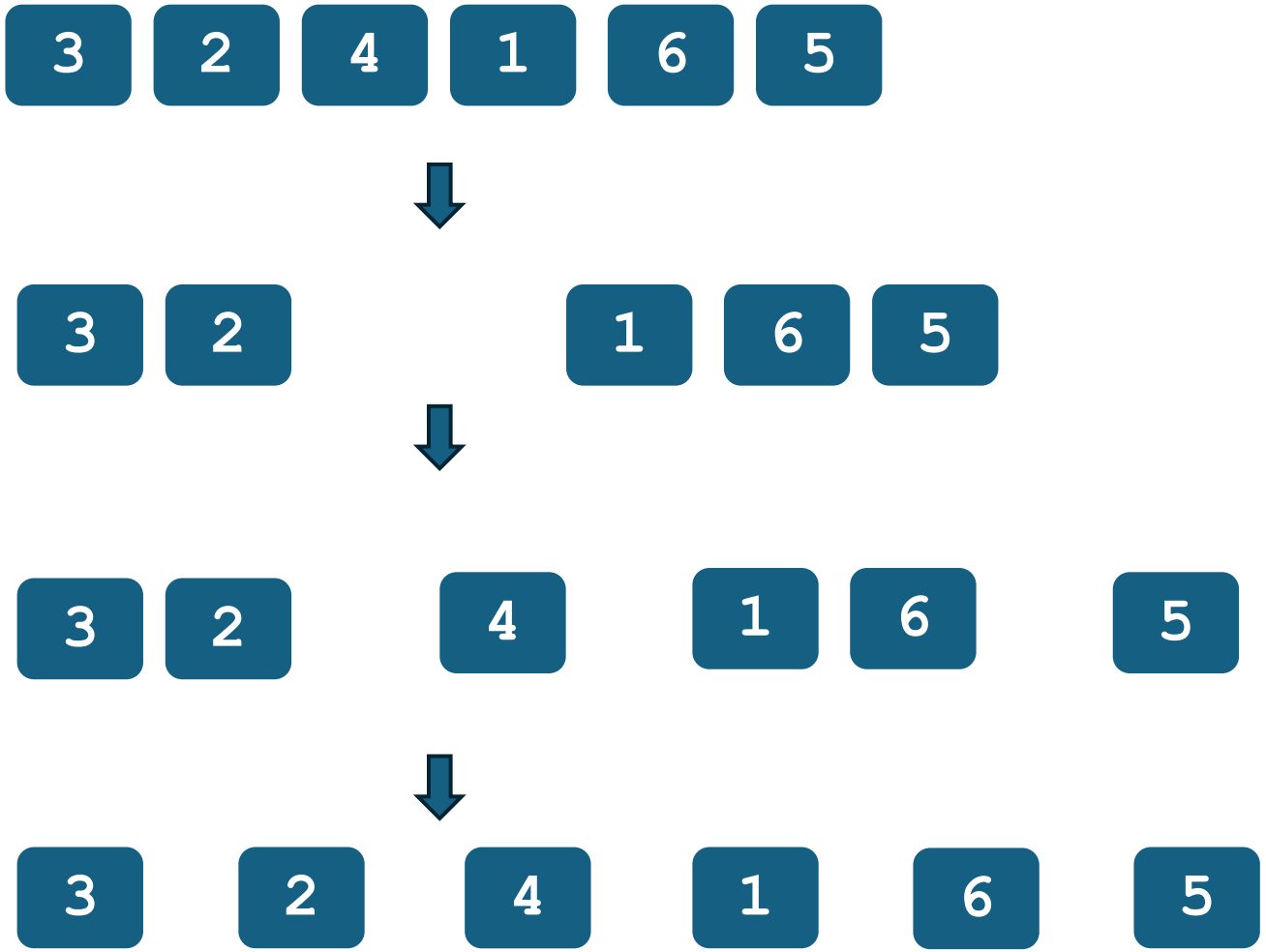
- Merge Sort:
  - Divides the array into two halves, sorts each half, and then merges the sorted halves.
  - Each divide (and subsequent merge) step takes linear time, and the division happens `logn` times (the depth of the recursion tree).

Quiz

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

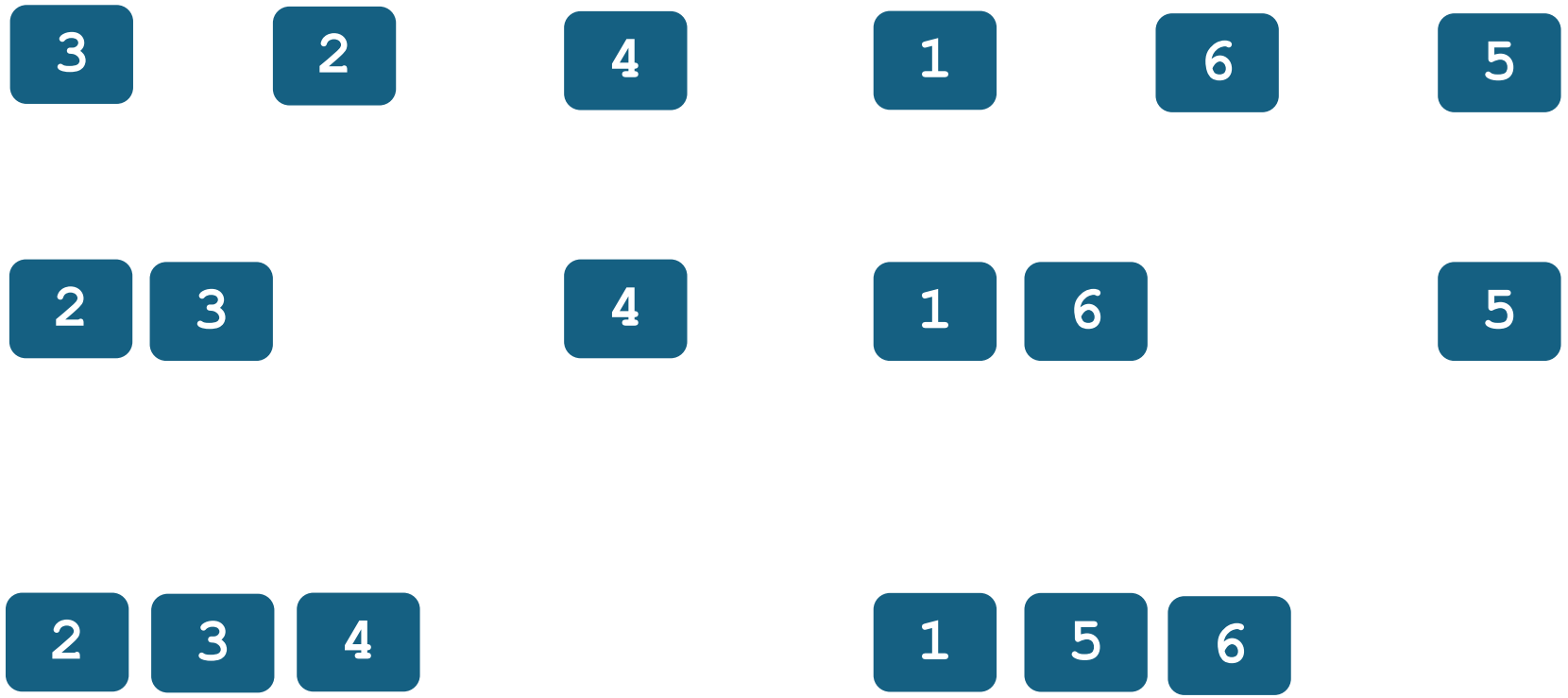
- First, split the array into smaller sub-arrays until each sub-array has only one element.
- Start merging them back together in a sorted order.
- Merge the two sorted sub-arrays

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

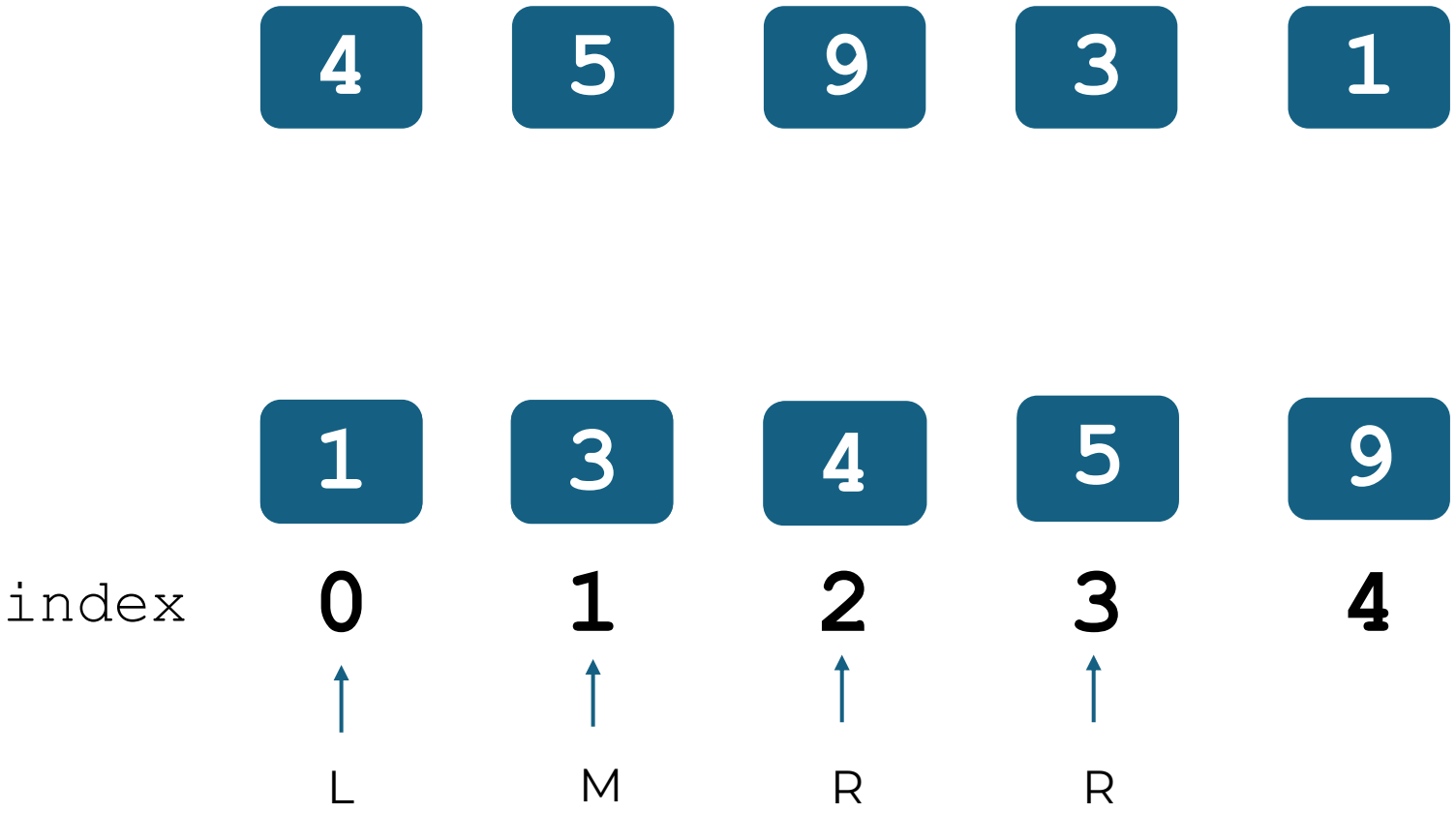


Step 3: Final merge

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)`

It uses binary insertion



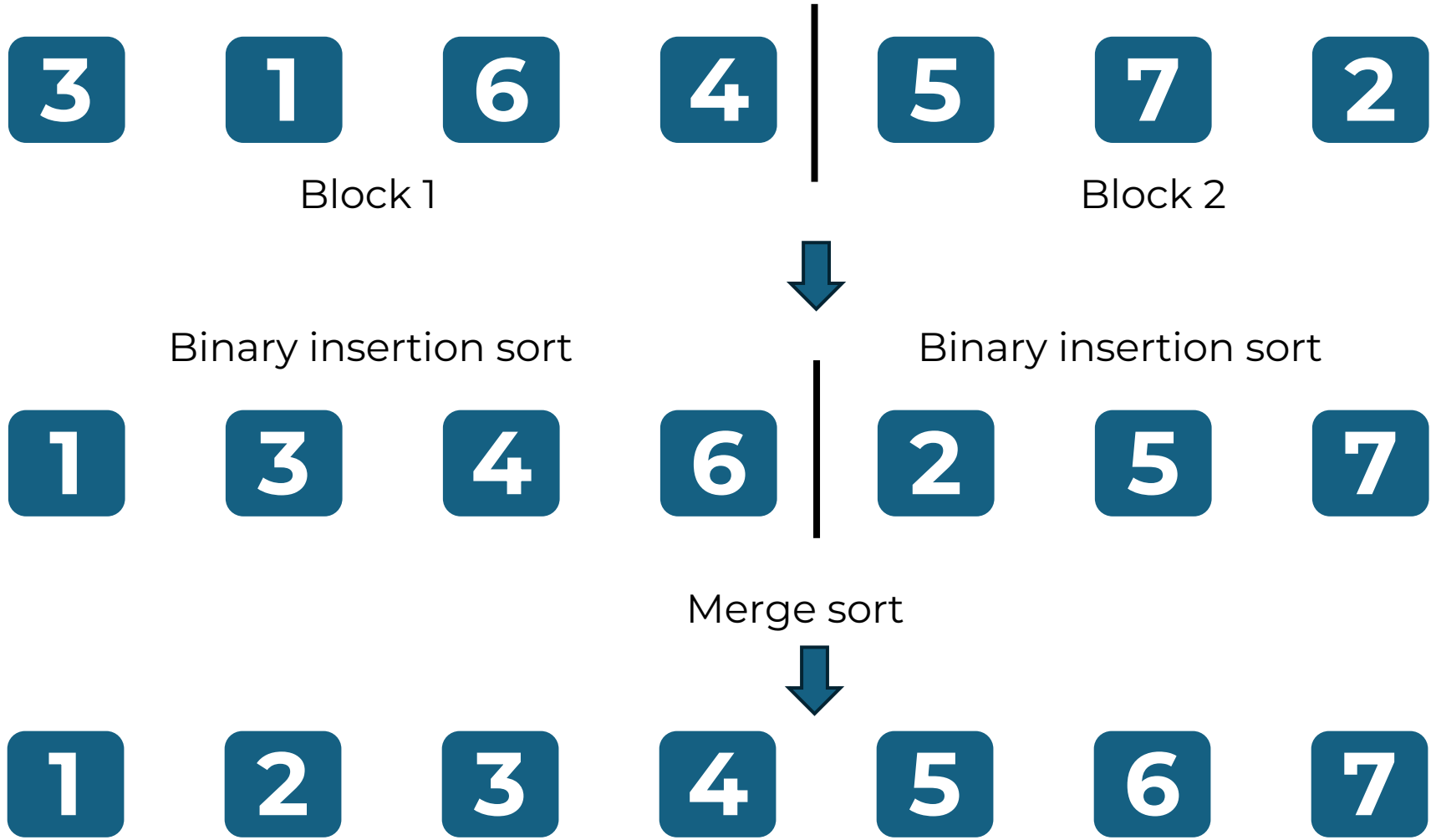
Tim sort steps

- Tim sort assumes that some data is already ordered…
- We call this run.
  - **`arr = {5,7,8,9,3, … }`**
  - Run 1 is **`{5,7,8,9}`**
- Min run:
  - Minimum length of each run
  - **`arr = {8,12,3,9,14, … }`**, min run = **`3`**
  - Run 1 is **`{8,12,3}`**
    - It will perform binary insertion to become **`{3,8,12}`**

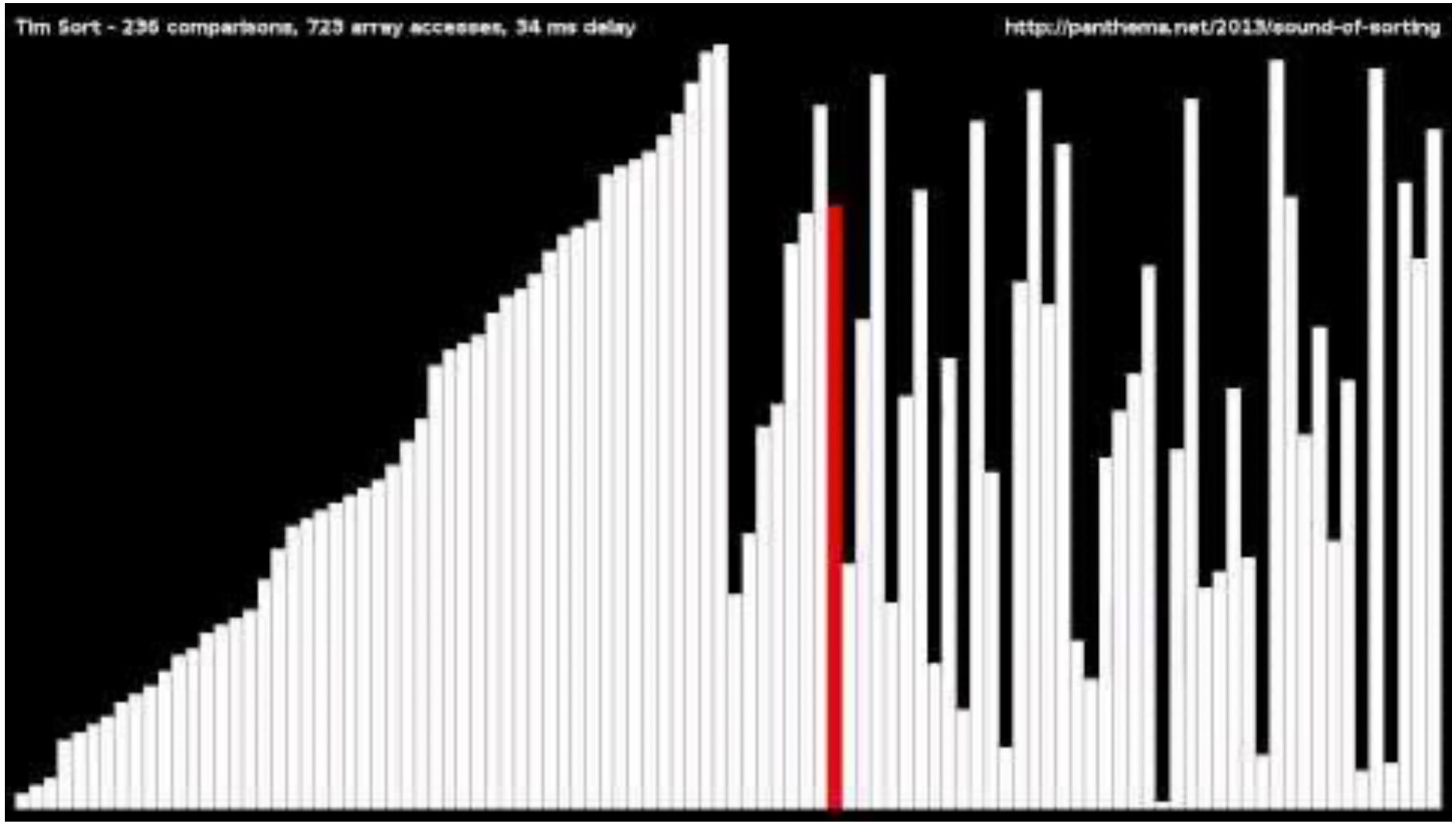
Tim sort steps

- Tim sort has some cool feature
  - If the min run is met, and the next element follows the pattern (ascending), then keep going…
  - **`arr = {5,7,8,9}, min run = 3`**
  - **`arr = {5,7,8}` → `{5,7,8,9}`**
- If the run is in descending order, then blindly reverse!
  - **`arr = {8,4,2,1}`**
  - **`arr = {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(logn)	O(1)
Bubble sort	O(n^2)	O(1)
Insertion sort	O(n^2)	O(1)
Merge sort	O(nlogn)	O(n)
Tim sort	O(nlogn)	O(n)

Divide and conquer!

Divide and conquer

- A general algorithm design paradigm
  - Divide: divide the input data `S` in two or more disjoint subsets `S1, S2, …`
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for `S1, S2, …,` into a solution for `S`
- Examples:
    - Binary search
    - Merge sort

Sometimes, we cannot do better!

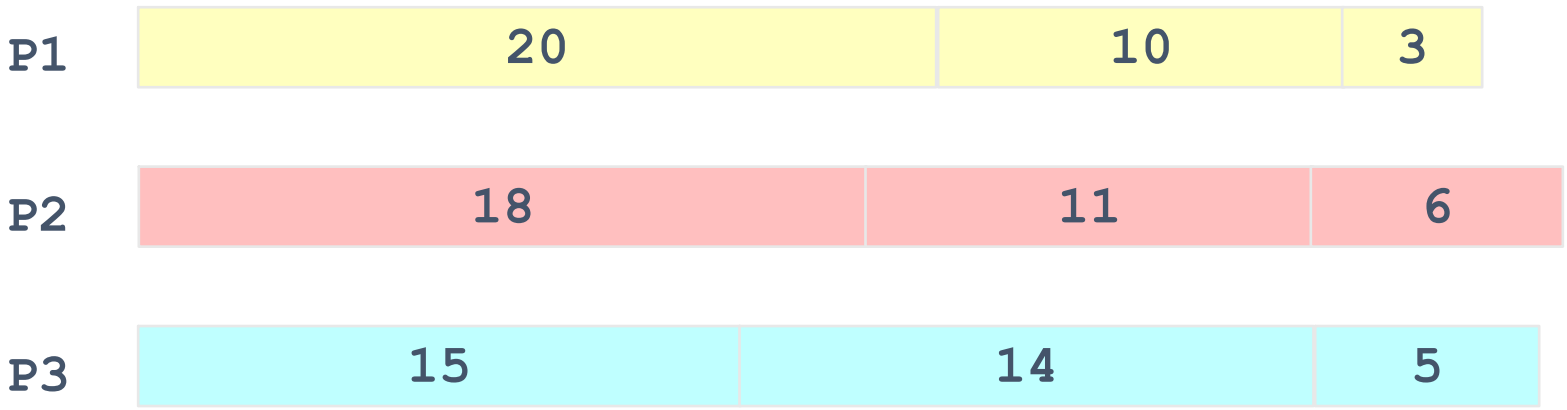
- Divide and conquer cannot be used.
- **Greedy algorithms**,
  - Unlike divide and conquer, make a series of choices that seem best at the moment with the hope of finding a global optimum.

Problem

- You must run nine Python scripts, each lasting `3, 5, 6, 10, 11, 14, 15, 18, and 20` minutes.
- You have three computers on which you can run these scripts.
- What is the best combination to run your scripts?

Solution 1

- You decide to do the longest-running jobs first on whatever processor is available. `{3, 5, 6, 10, 11, 14, 15, 18, 20}`

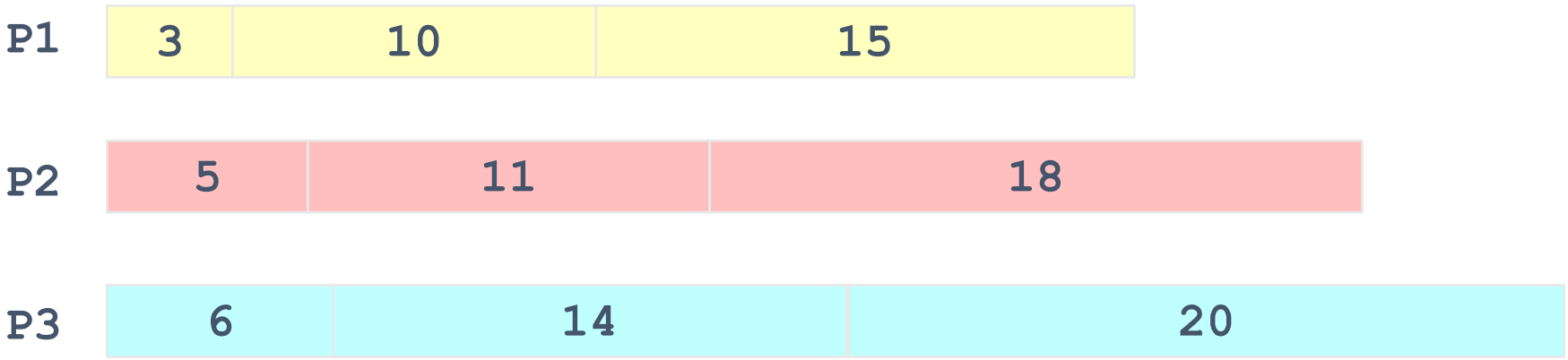


  - Time to completion: `18 + 11 + 6 = 35` minutes
  - This solution isn't bad, but we might be able to do better

Solution 2

- You decide to do the shortest-running jobs first.

`{3, 5, 6, 10, 11, 14, 15, 18, 20}`



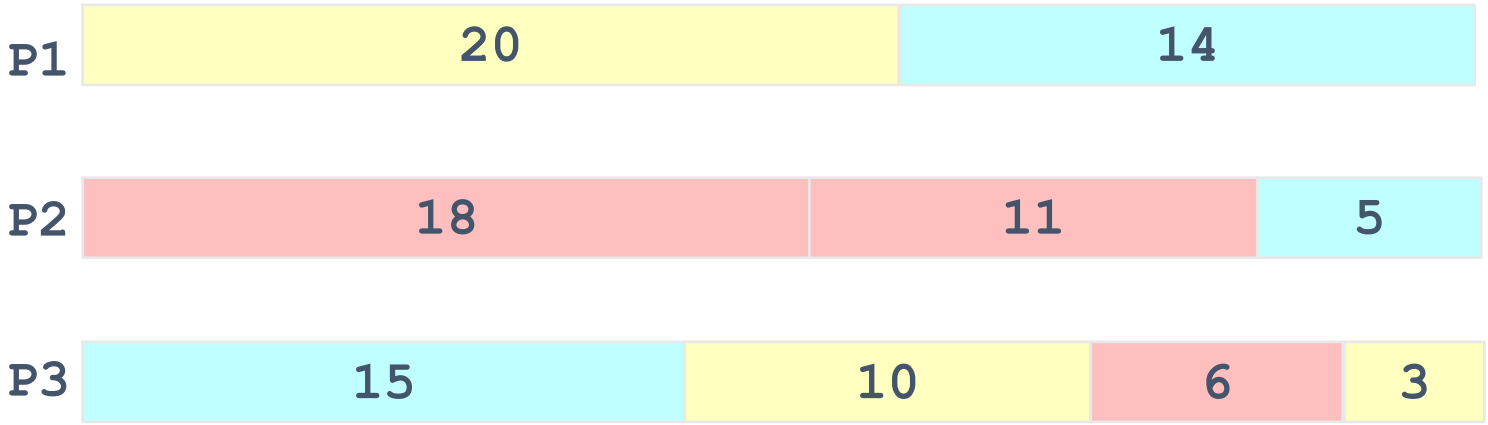
- That wasn't such a good idea; time to completion is now

  `6 + 14 + 20 = 40` minutes

Solution 3

- Better solutions do exist:
  - Greedy: at each stage was pick the maximum (or the minimum for other problems)

`{3, 5, 6, 10, 11, 14, 15, 18, 20}`



- Time to completion: `20 + 14 = 34` minutes

How to find the optimal solution?

- One way:
  - Try all possible assignments of jobs to processors
- Unfortunately, this approach can take exponential time!
  - Selecting the best option available at the moment.

Fintech example

- Resource Allocation in Finance:
  - You have £10,000 that you want to invest.
    - You decided to build a portfolio and make various investments with the hope of maximising your profit.
    - But there are some risks, like volatility levels of stocks (price can change dramatically in a short period of time)
  - How to model this scenario?
    - Each investment option has a specific return (value) and risk (weight), and the goal is to maximise the return on investment

Ali Baba and the 40 thieves...

- Ali Baba is a woodcutter who discovers the secret cave hideout of a band of forty thieves while working in the forest.
- He overhears the thieves' leader, often named Cassim, using a magical phrase to open the cave where they store their stolen treasures. Ali Baba learns the phrase and gains access to the cave.
    - Open sesame!
- Ali Baba enters the cave with a bag and takes some of the treasure for himself...
- What is the best strategy for Ali Baba to get as much as possible out of the cave?

Knapsack fractional: A maximisation problem!

`7` objects (`n=7`)

`15` is the size of a bag (`m=15`)

Objects	**O**	`1`	`2`	`3`	`4`	`5`	`6`	`7`
Profits	**P**	`10`	`5`	`15`	`7`	`6`	`18`	`3`
Weight	**W**	`2`	`3`	`5`	`7`	`1`	`4`	`1`

- The question is: How do we fill the bag so that the profit is maximized?

Knapsack fractional: A maximization problem!

Objects:	**O**	1	2	3	4	5	6	7
Profits:	**P**	10	5	15	7	6	18	3
Weight:	**W**	2	3	5	7	1	4	1
Profit by weight:	$\frac{P}{W}$	$\frac{10}{2}$=5	$\frac{5}{3} = 1.66$	$\frac{15}{5}$=3	$\frac{7}{7} = 1$	$\frac{6}{1} = 6$	$\frac{18}{4}$=4.5	$\frac{3}{1}$=3

Knapsack fractional: A maximization problem!

Objects:	**O**	1	2	3	4	5	6	7
Profits:	**P**	10	5	15	7	6	18	3
Weight:	**W**	2	3	5	7	1	4	1
Profit by weight:	$\frac{P}{W}$	5	1.66	3	1	6	4.5	3

How much should we take?

0 < amount <1

Knapsack fractional: A maximization problem!

Objects:	**O**	1	2	3	4	5	6	7
Profits:	**P**	10	5	15	7	6	18	3
Weight:	**W**	~~2~~	~~2~~ 3	~~5~~	7	~~1~~	~~4~~	~~1~~
Profit by weight:	$\frac{P}{W}$	5	1.66	3	1	6	4.5	3
		1	2/3	1	0	1	1	1

Bag size: 15

15-1=14

14-2=12

12-4=8

8-5=3

3-1=2

Knapsack fractional: A maximization problem!

n objects (n=7)

m is size of a bag (m=15)

Objects: O	1	2	3	4	5	6	7
Profits: P	**10**	**5**	**15**	**7**	**6**	**18**	**3**
Weight: W	2	3	5	7	1	4	1
P/W	5	1.3	3	1	6	4.5	3
x	1	2/3	1	0	1	1	1

Total weight: `1*2+2/3*3+1*5+0*7+1*1+1*4+1*1 = 15`

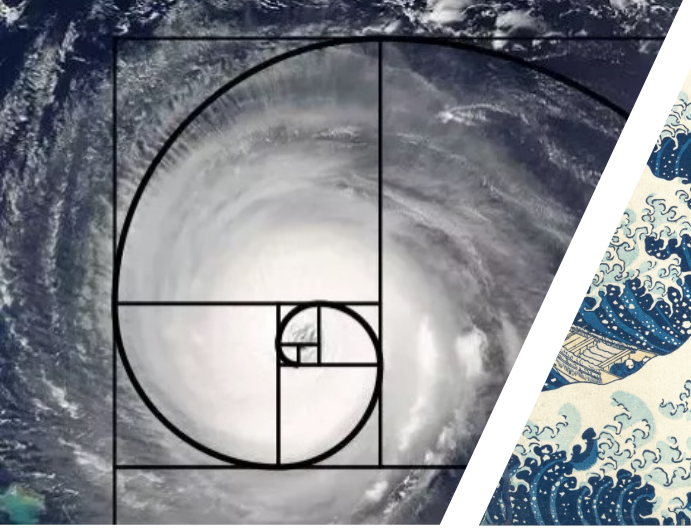
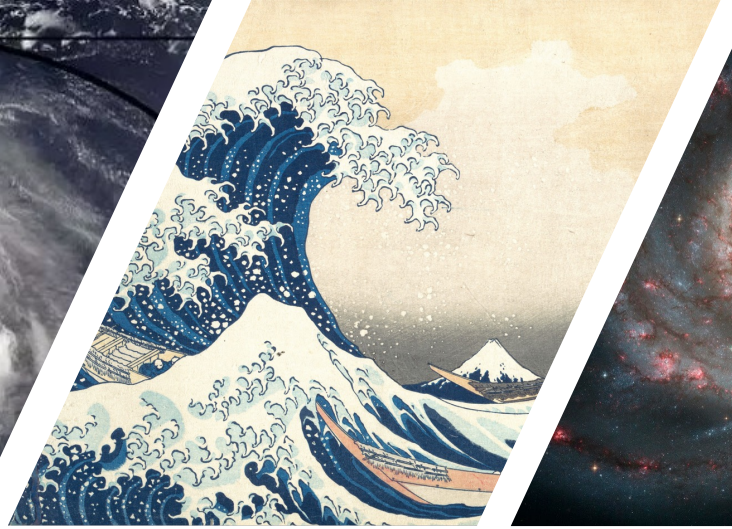
Total profit: `1*10+2/3*5+1*15+0*7+1*6+1*18+1*3 = 55.33`

We can still do better!

Dynamic programming

The Romanesco flower!

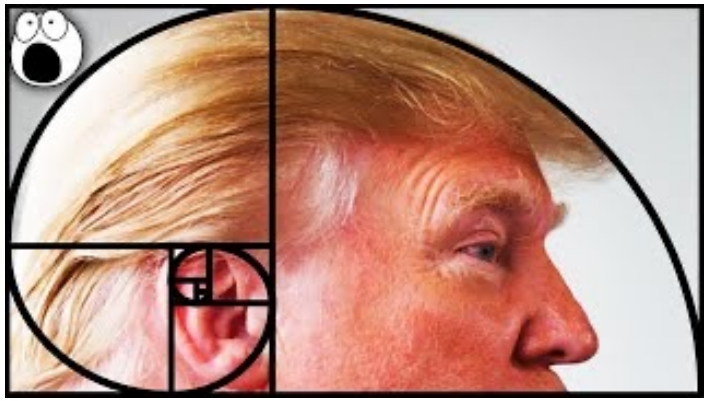
- Romanesco has flowers that form perfect geometric spiral patterns that repeat.
- Looking closely, each spiral bud is composed of a series of smaller buds
  - All buds are arranged in yet another spiral that continues at smaller levels
- The number of spirals on Romanesco's head follows the **Fibonacci sequence**!

Other examples

Fibonacci symmetry algorithm underlies our perception of attractiveness


Reflects the growth processes in nature



What is the Fibonacci sequence?

- The Fibonacci Sequence is a series of numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

- The sequence starts with **zero** and **one** and is a steadily increasing series where each number equals the sum of the preceding two numbers.
- Example: Fibonacci F_3 is found by adding $F_2 + F_1$

$$F_3 = F_2 + F_1 = 1+1 = \mathbf{2}$$

$$F_2 = F_1 + F_0 = 1+0 = 1$$

$$F_1 = 1$$

Quiz!

- What is the Fibonacci of 7?

  Answer: 13

$$F_3 = F_2 + F_1 = 1+1 = \mathbf{2}$$
$$F_2 = F_1 + F_0 = 1+0 = 1$$
$$F_1 = 1$$

$$F_0 = 0$$
$$F_1 = 1$$
$$F_2 = 1$$
$$F_3 = 2$$
$$F_4 = 3$$
$$F_5 = 5$$
$$F_6 = 8$$
$$\mathbf{F_7 = 13}$$

How to calculate the Fibonacci sequence?

script

```
function fibo(n)
    if n <= 1
        return n
    else
        return fibo(n-1) + fibo(n-2)
```

How to calculate the Fibonacci sequence?

script

```
function fibo(n)
    if n <= 1
        return n
    else
        return fibo(n-1) + fibo(n-2)
```

- If `n = 0`, then what is the `fibo(0)`?
  - Answer: **0**
- If `n = 1`, then what is the `fibo(1)`?
  - Answer: **1**

How to calculate the Fibonacci sequence?

script

```
function fibo(n)
    if n <= 1
        return n
    else
        return fibo(n-1) + fibo(n-2)
```

- If `n = 2`, then what is the `fibo(2)`?
  - Answer: `fibo(1) + fibo(0) = 1 + 0 = 1`

- If `n = 3`, then what is the `fibo(3)`?
  - Answer: `fibo(2) + fibo(1) = 1 + 1 = 2`

What is recursion in programming?

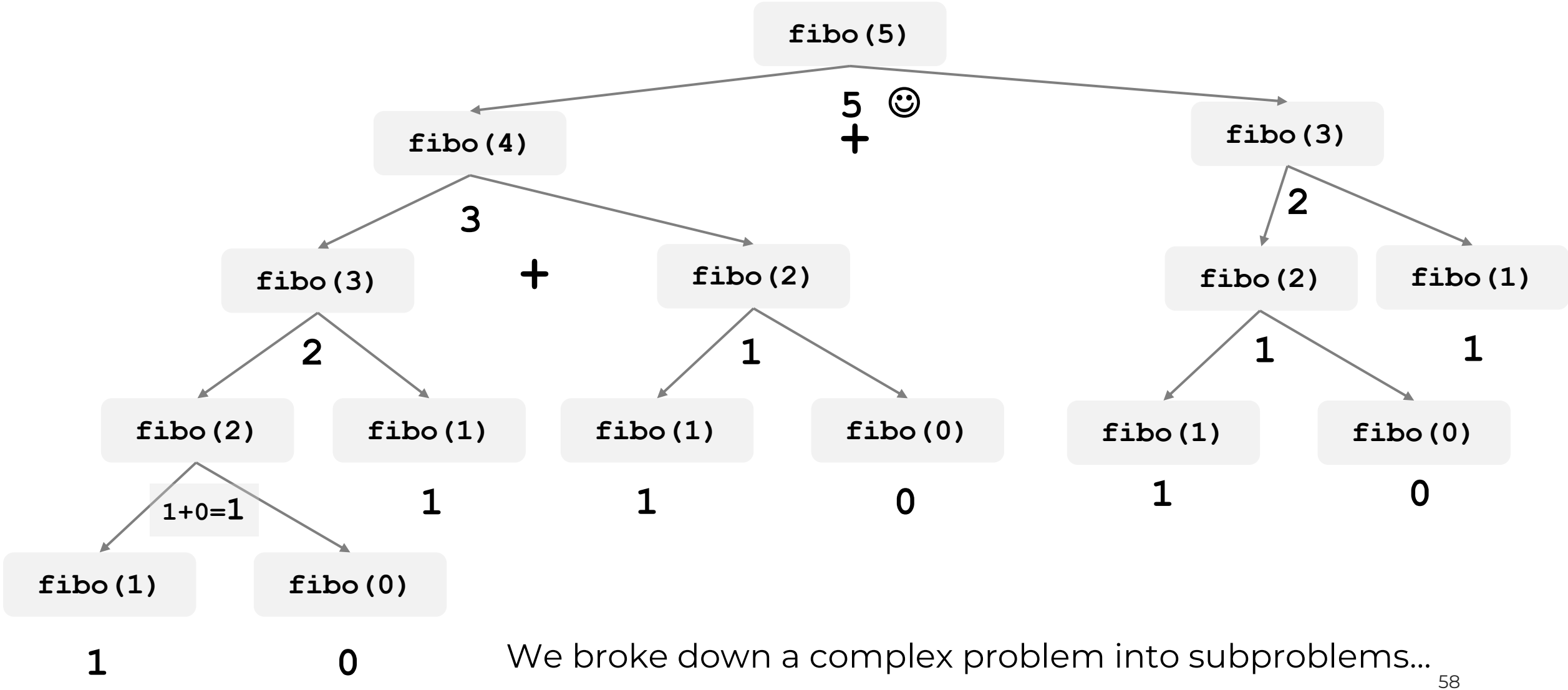
script

```
# The sum of the first 10 natural numbers


def recursiveSum(n):
    if n == 0:
        return 0
    else:
        return n + recursiveSum (n - 1)
```

Let's see in practice!

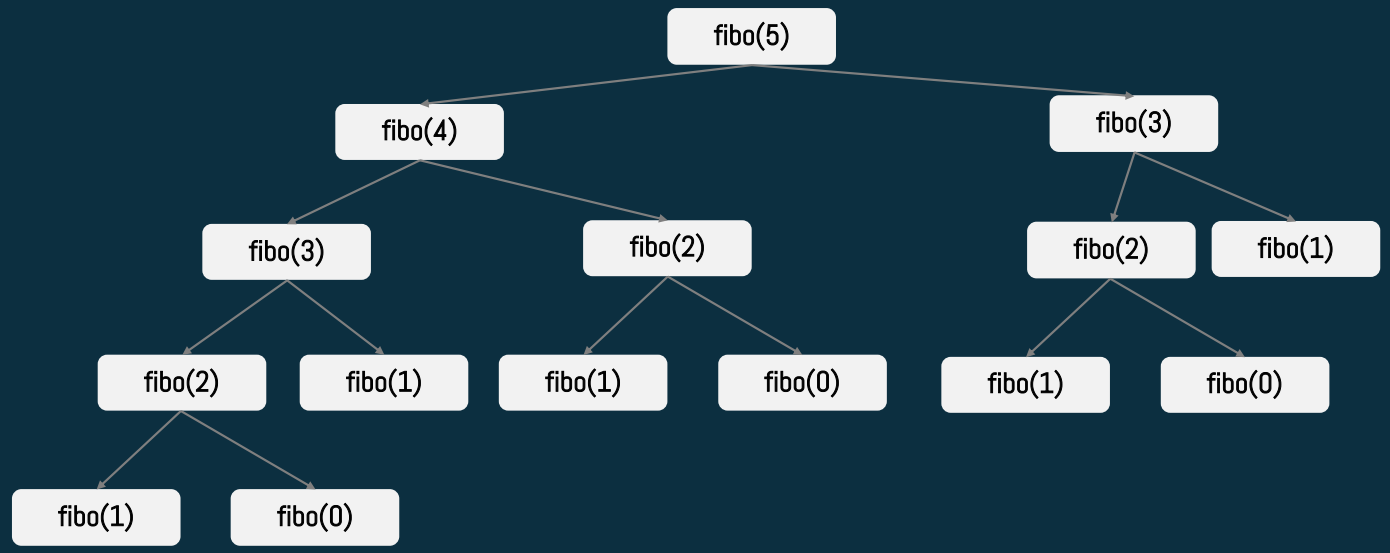
We solve it recursively!



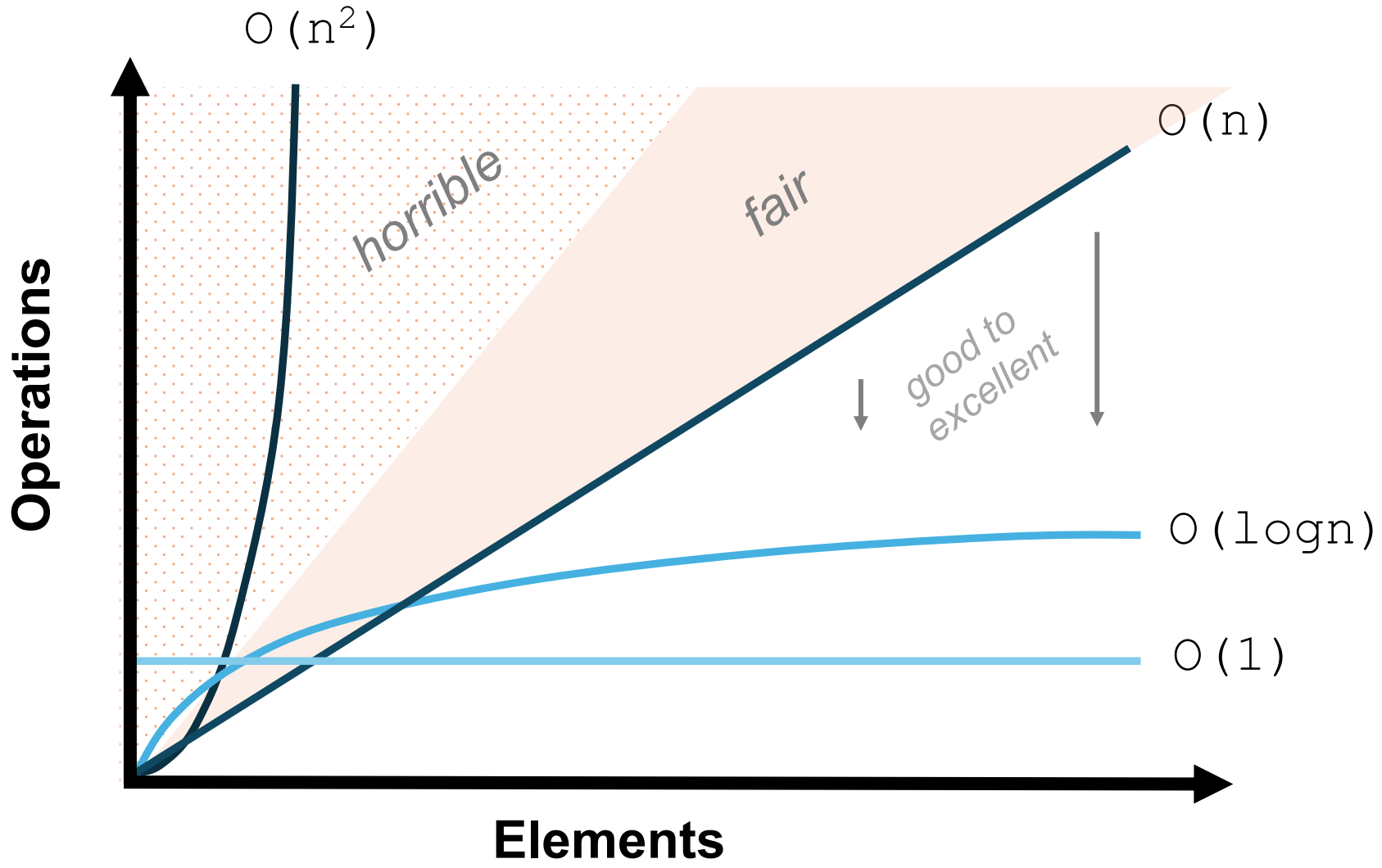
We broke down a complex problem into subproblems...

Quiz!

- What about computational complexity?
  - What is the time complexity of the Fibonacci method we just explored?
  - Answer: $O(2^n)$ (exponential)
- Is this good?
  - Answer: O(No!)

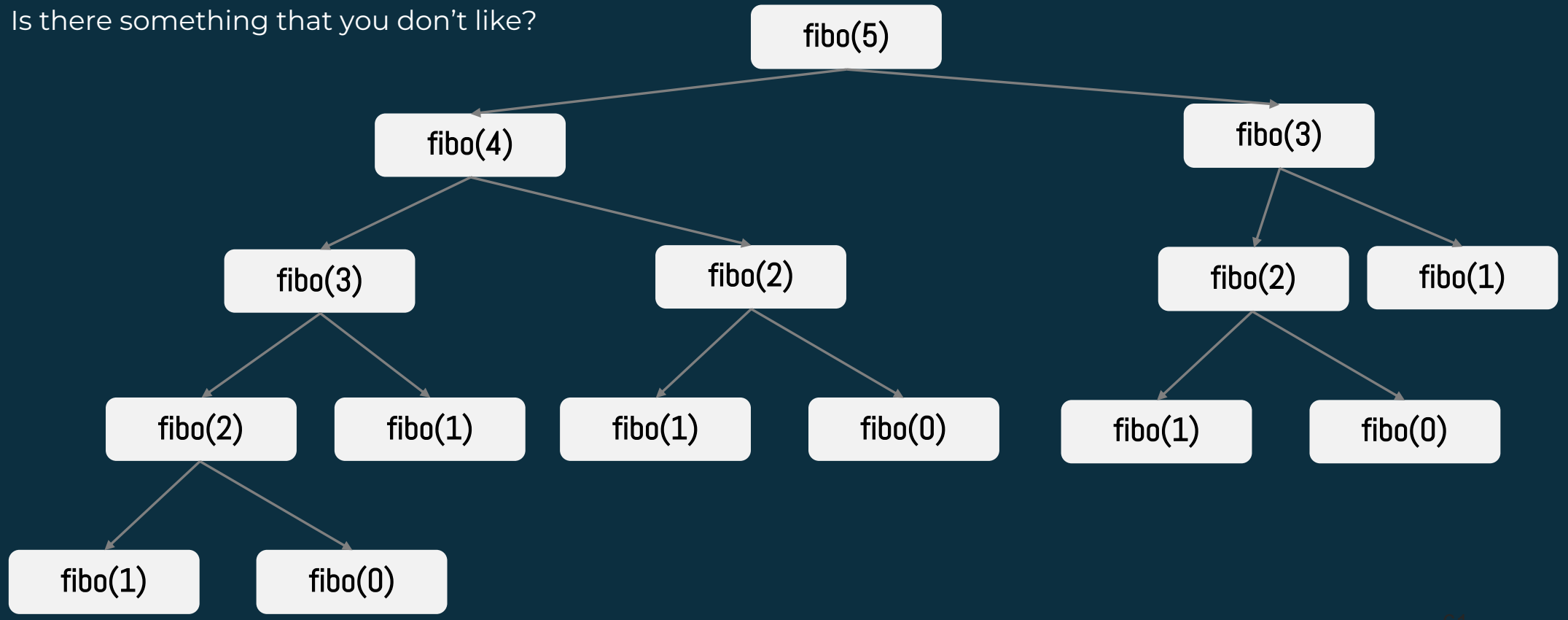


Big-O Complexity Chart



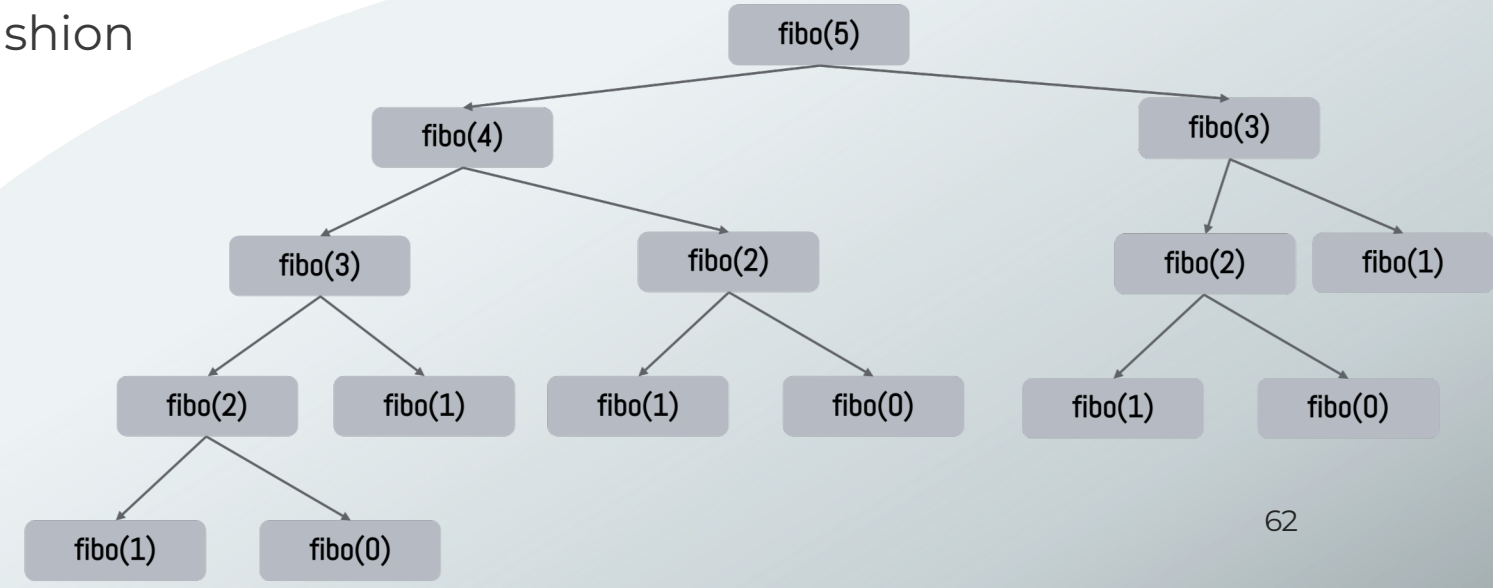
Quiz!

- Can you do better?
  - Is there something that you don't like?



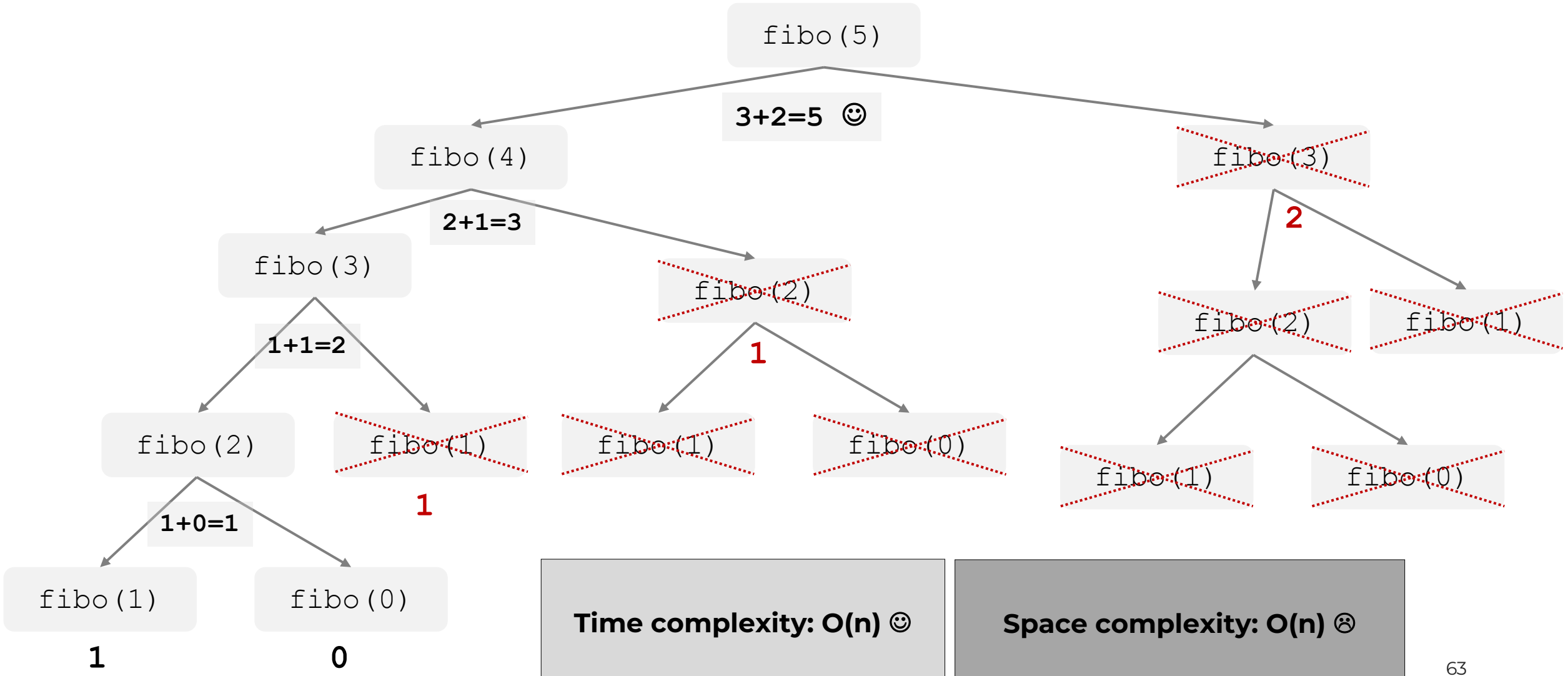
Can we do it better?

- **What if we avoid calculating the same Fibonacci numbers again and again?**
- What if we memorise the results of an existing calculation and use it in the next iteration?
  - Let's solve it in a **bottom-up** fashion



0	1	1	2	3	5
0	1	2	3	4	5

Let's use dynamic programming!



Time complexity: O(n) ☺

Space complexity: O(n) ☹

Let's recap

- Dynamic programming is a technique that breaks problems into sub-problems ☺
  - But it stores intermediate results, and that's a tradeoff ☹
  - There is no guarantee that the stored value will be used later in the execution

Dynamic programming vs. Greedy

- **Divide and conquer** breaks down a problem into two or more sub-problems
  - Until these become simple enough to be solved directly.

- **Greedy methods** (top-down) solve a problem as quickly as possible.
  - Make whatever choice seems best now and then solve the sub-problems arising after.

- **Dynamic programming** (bottom-up) solves a problem as efficiently as possible.
  - Store the result for future purposes so you do not need to compute it again.

Thank you!

- The lab starts soon!

  (404-405)

O(no!)

Lab 2

Big Data Analytics

Lab activities

- Complete lab 2 activities and exercises