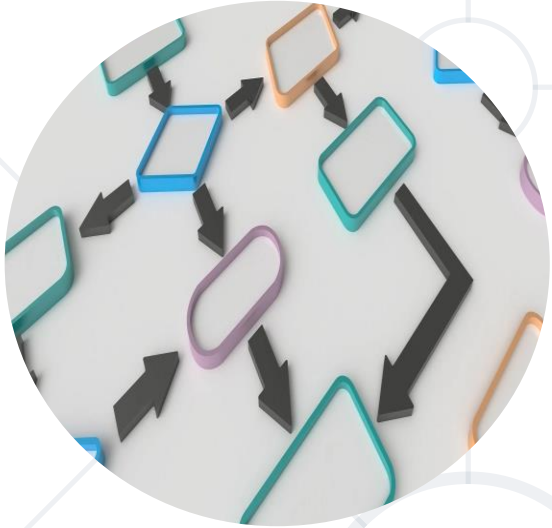
Algorithms Introduction



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-advanced

Table of Contents

1. Algorithmic **Complexity**
2. **Recursion** and **Recursive** Algorithms
3. **Brute-Force** Algorithms
4. **Greedy** Algorithms and **Greedy** Failure Cases
5. **Searching** Algorithms
   - Linear Search and Binary Search
6. Simple **Sorting** Algorithms
   - Selection Sort, Bubble Sort, Insertion Sort

O(n)

Algorithmic Complexity

Asymptotic Notation

Algorithm Analysis

- Why should we **analyze algorithms**?
  - Predict the **resources** the algorithm will need
    - Computational time (**CPU** consumption)
    - Memory space (**RAM** consumption)
    - Communication **bandwidth** consumption
    - **Hard disk** operations

Get Number of Steps

- Calculate the maximum steps to find the result

```
def get_operations_count(n):
    counter = 0

    for i in range(n):
        for j in range(n):
            counter += 1

    return counter
```

$O(n^2)$

- The input(**n**) of the function is the main source of steps growth

Simplifying Step Count

- Initialization of `counter`: **1 time**
- Creating `range(n)` for the outer loop: **1 time**
- Outer loop comparison: **n + 1 times**
- Creating `range(n)` for the inner loop: **n times**
- Inner loop comparison: **n * (n + 1) times**
- Incrementing `counter`: **n * n times**
- Return statement: **1 time**

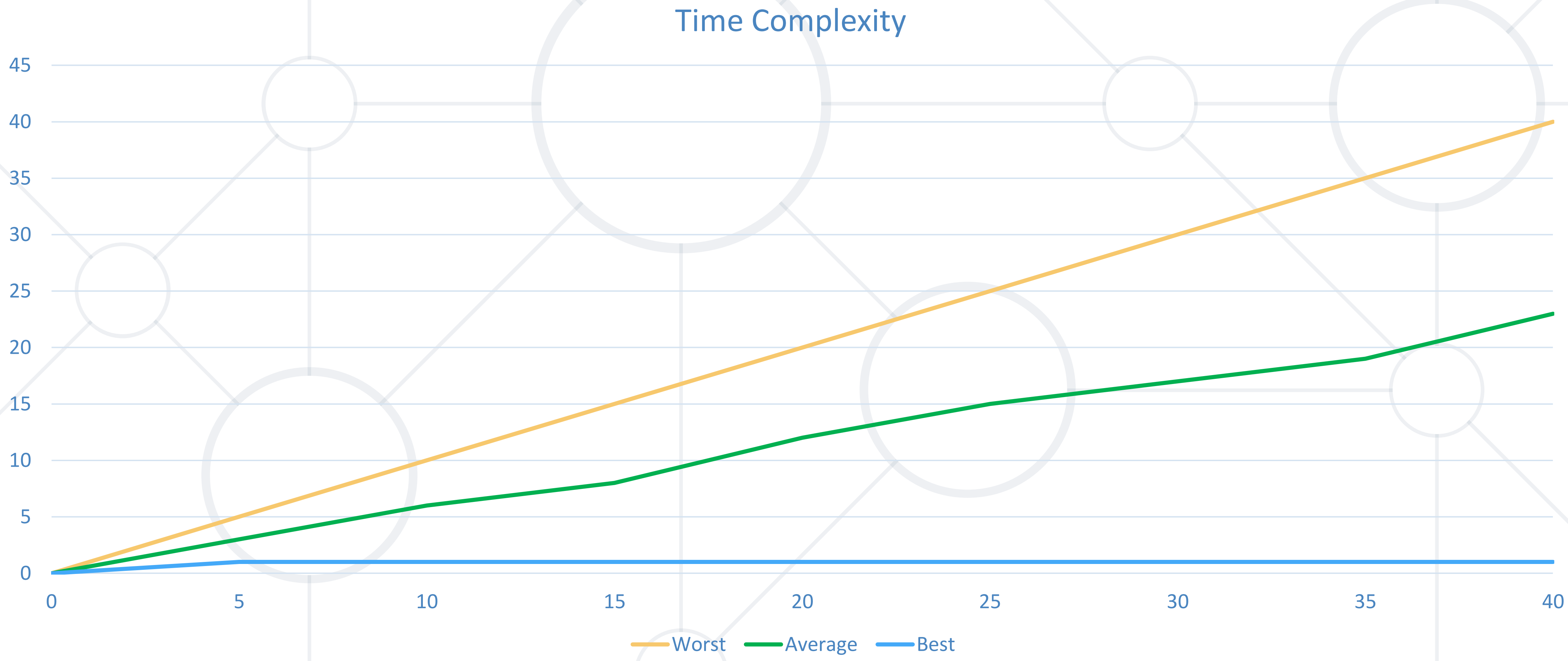
Time Complexity

Software University

- **Worst-case**
  - An **upper** bound on the running time
- **Average-case**
  - **Average** running time
- **Best-case**
  - The **lower** bound on the running time (the optimal case)

Time Complexity

- Therefore, we need to measure **all** the possibilities:



Time Complexity

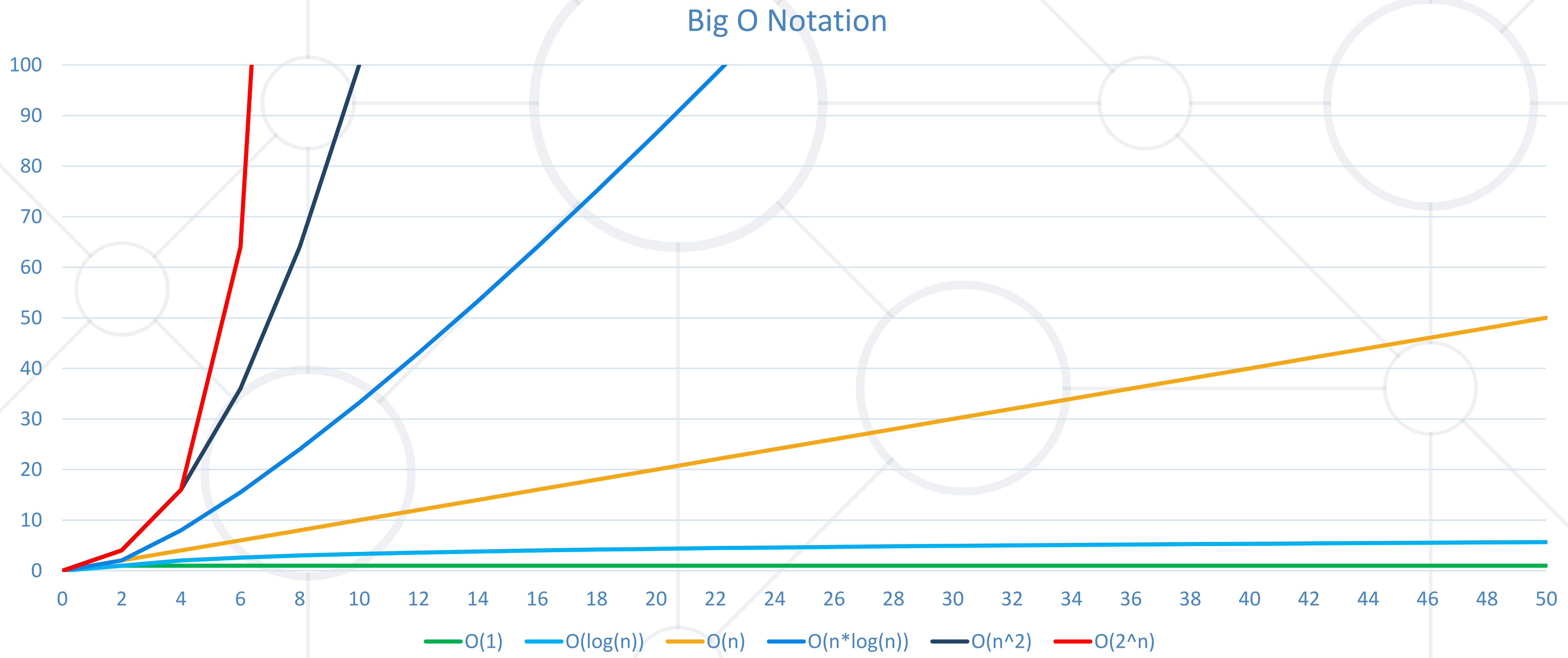
- From the previous chart we can deduce:
  - For smaller size of the input (**n**) we **don't care much for the runtime**
  - So we measure the time as **n** approaches **infinity**
  - If an algorithm **must scale**, it **should compute** the result within a **finite and practical time**
  - We're concerned about the **order of an algorithm's complexity**, not the actual time in terms of **milliseconds**

Asymptotic Notations

- Descriptions that allow us to examine an algorithm's running time
- There are **three** common asymptotic notations:
  - Big **O – O(f(n))**
  - Big **Theta – Θ(f(n))**
  - Big **Omega – Ω(f(n))**

Asymptotic Functions

- Below are some examples of **common algorithmic** grow:



Typical Complexities

Complexity	Notation	Description
constant	O(1)	n = 1 000 → 1-2 operations
logarithmic	O(log n)	n = 1 000 → 10 operations
linear	O(n)	n = 1 000 → 1 000 operations
linearithmic	O(n*log n)	n = 1 000 → 10 000 operations
quadratic	O(n2)	n = 1 000 → 1 000 000 operations
cubic	O(n3)	n = 1 000 → 1 000 000 000 operations
exponential	O(n^n)	n = 10 → 10 000 000 000 operations

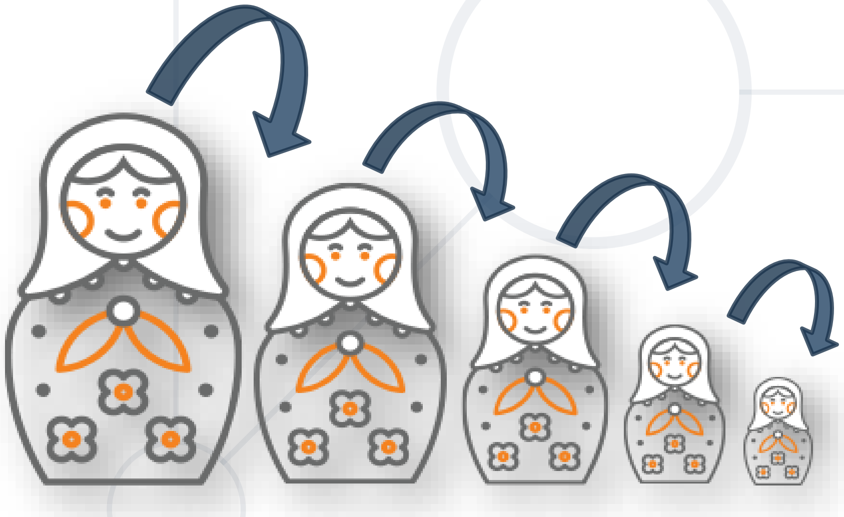
Recursion

What is Recursion?

- **Method** of solving a problem where the solution depends on solutions to smaller instances of the same problem
- A common **computer programing tactic** is to **divide** a problem into **sub-problems** of the same type as the original, **solve** those sub-problems, and **combine** the **results**

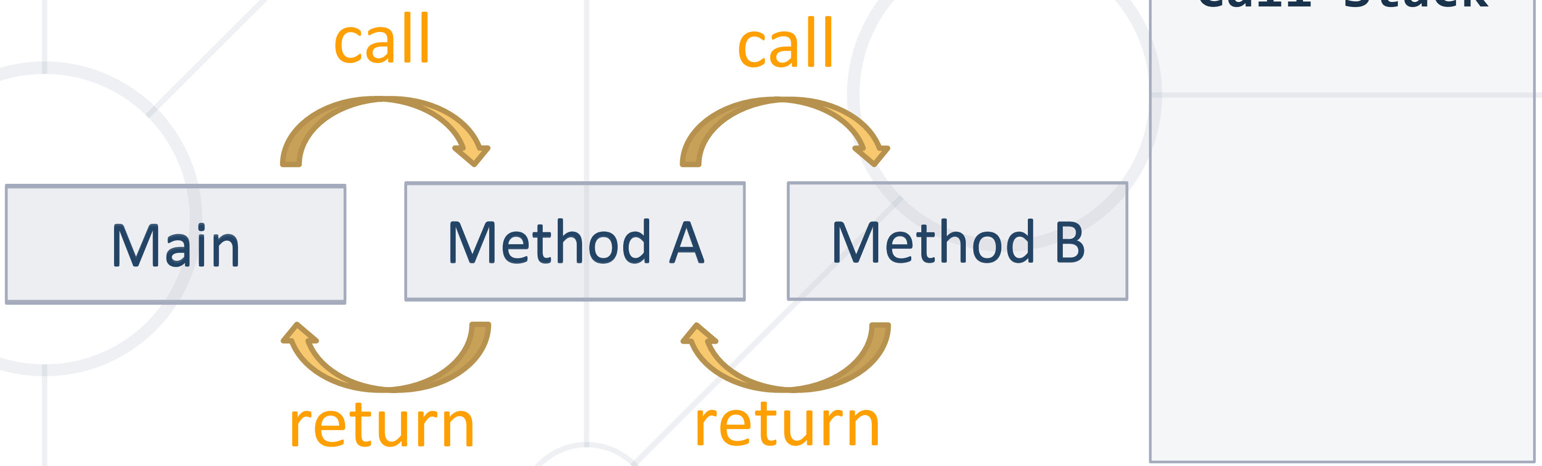
What is Recursion?

- A function or a method that **calls itself one or more** times until a specified **condition** is **met**
  - After the recursive call the rest code is processed **from** the **last** one called **to** the **first**



Call Stack

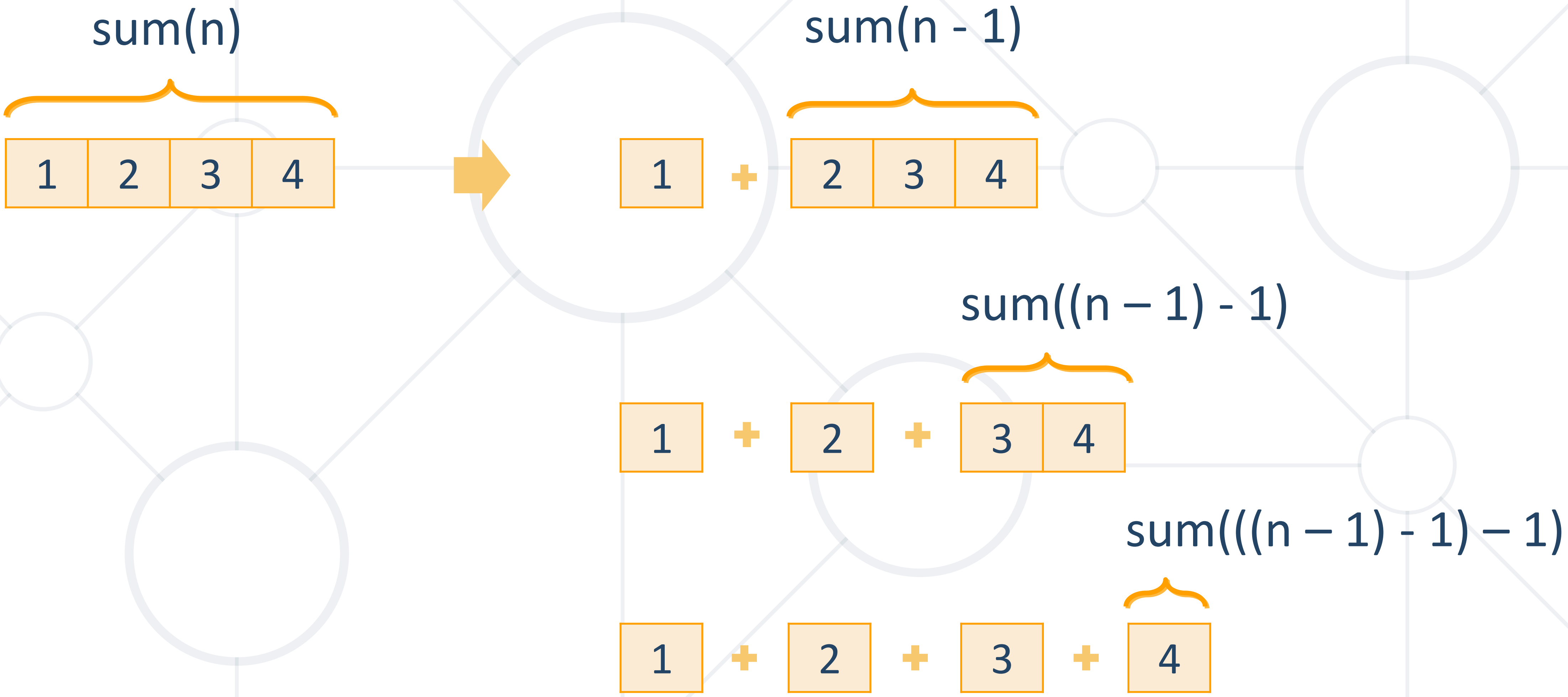
- "The stack" is a small **fixed-size** chunk of memory (e.g. 1MB)
- Keeps track of **the point** to which each active subroutine should **return control** when it **finishes executing**



Other Definition

- Problem solving technique (In CS)
  - Involves a **function calling itself**
  - The function should have a **base case**
  - **Each step** of the recursion should **move towards** the **base case**

Array Sum – Example



Problem: Recursive Array Sum

- Create a **recursive method** that
  - Finds the sum of all numbers stored in an `array`
  - Read numbers from the console

```
1 2 3 4  →  10
```

```
-1 0 1  →  0
```

Solution: Recursive Array Sum

```
def calc_sum(numbers, idx):
    if idx == len(numbers) - 1:
        return numbers[idx]

    return numbers[idx] + calc_sum(numbers, idx + 1)
```

Base case

Recursive call

Problem: Recursive Factorial

- Create a **recursive method** that calculates **n!**
  - Read **n** from the console

Input		Output
`5`	→	`120`
`10`	→	`3628800`

Recursive Factorial – Example

- Recursive definition of **n!** (n factorial):

```
n! = n * (n-1)! for n > 0
0! = 1
```

3!	=	3 * 2!
2!	=	2 * 1!
1!	=	1 * 0!
0!	=	1

Solution: Recursive Factorial

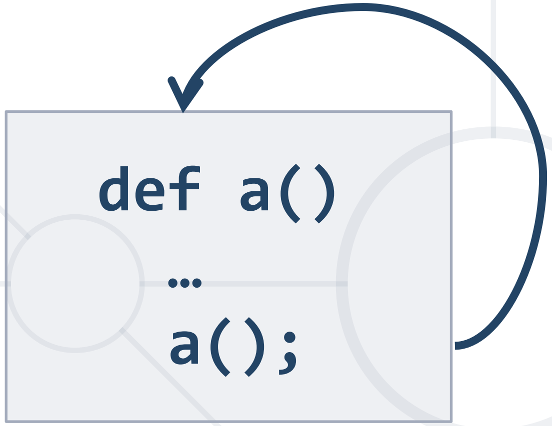
```
def get_factorial(num):
    if num == 0:
        return 1

    return num * get_factorial(num - 1)
```

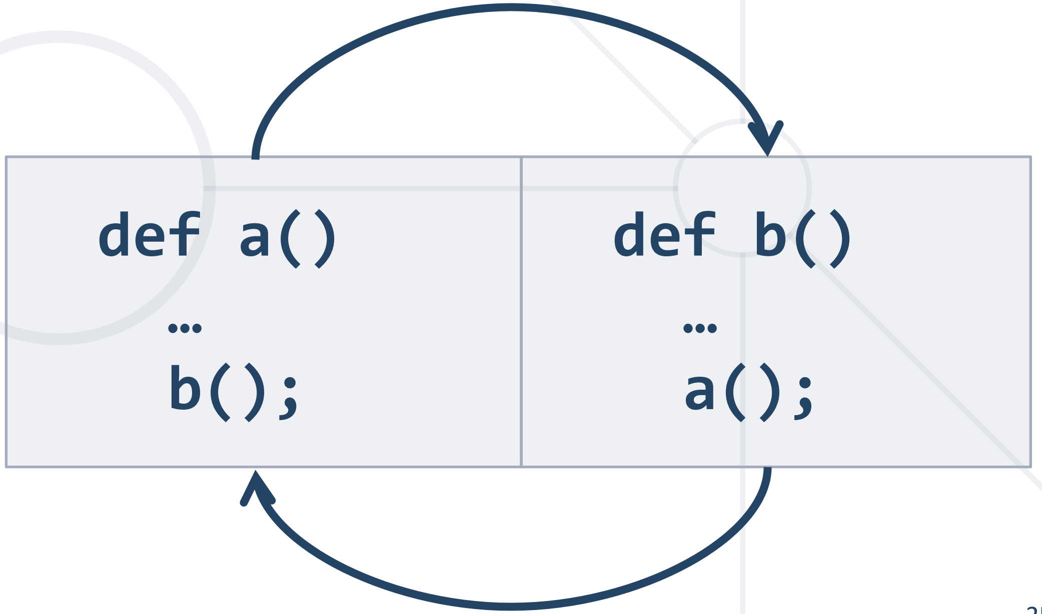
Base case

Recursive call

Direct and Indirect Recursion

- **Direct recursion**
  - A method directly calls itself
- **Indirect recursion**
  - Method **A** calls **B**, method **B** calls **A**
  - Or even **A** → **B** → **C** → **A**





Recursion Pre-Actions and Post-Actions

- Recursive methods have **three** parts:
  - **Pre-actions** (before calling the recursion)
  - **Recursive calls** (step-in)
  - **Post-actions** (after returning from recursion)

```
def recursion()
  # Pre-actions
  recursion()
  # Post-actions
```

Problem: Recursive Drawing

- Create a **recursive method** that draws the following figure

```
5
```

```
*****
****
***
**
*
#
##
###
####
#####
```

Pre-Actions and Post-Actions – Example

```python
def print_figure(n):
    if n <= 0:
        return

    # TODO: Pre-action: print n asterisks

    # Recursive call
    print_figure(n - 1)

    # TODO: Post-action: print n hashtags
```

Performance: Recursion vs. Iteration

- Recursive calls are **slower**
- Parameters and return values **travel** through the stack
- Good for branching problems

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

- No function call **cost**
- Creates **local** variables
- Good for linear problems (no branching)

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Brute-Force Algorithms

Brute-Force Algorithms

- Trying all possible combinations
- Picking the best solution
- Usually slow and inefficient

Brute-Force Algorithms

0	0	0	0	0

Brute-Force Algorithms

0	0	0	0	1

Brute-Force Algorithms

0	0	0	0	2

Brute-Force Algorithms

9 9 9 9 9

10 x 10 x 10 x 10 x 10 = 100,000 combinations

Greedy Algorithms

Greedy Algorithms

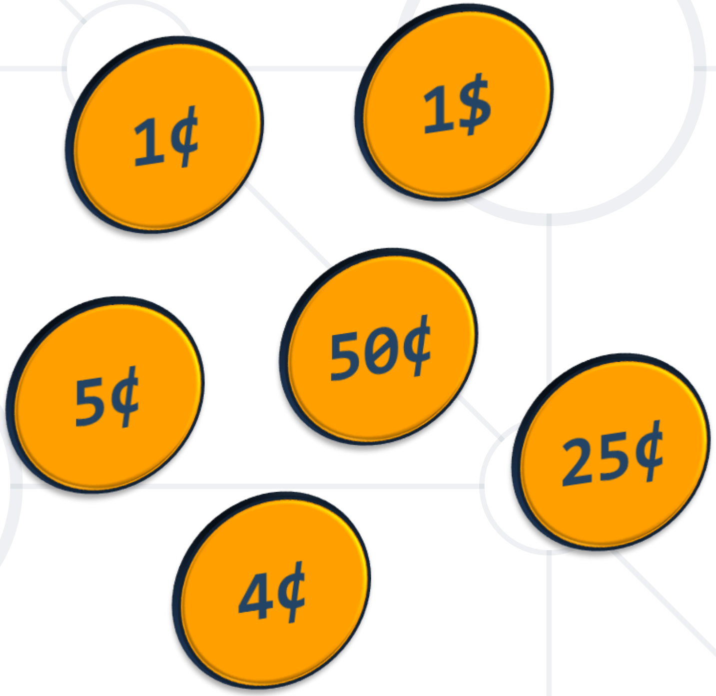
- Used for solving optimization problems
- Usually more efficient than the other algorithms
- Can produce a **non-optimal** (incorrect) result
- Pick the **best local** solution
  - The optimum for a **current** position and point of view
- Greedy algorithms assume that always choosing a **local** optimum leads to the **global** optimum

Optimization Problems

- Finding the best solution from all possible solutions
- Examples:
  - Find the **shortest** path from Sofia to Varna
  - Find the **maximum increasing subsequence**
  - Find the shortest route that visits each city and returns to the origin city

Problem: Sum of Coins

- Write a program, which gathers a sum of money, using the least possible number of coins
- Consider the US **currency coins**
  - **0.01**, **0.02**, **0.05**, **0.10**
- **Greedy algorithm** for "Sum of Coins":
  - Take the largest coin while possible
  - Then take the second largest
  - etc.



Sum of Coins Visualization

Target: 18

Actual: 0

Sum of Coins Visualization

Target: 18

Actual: 10

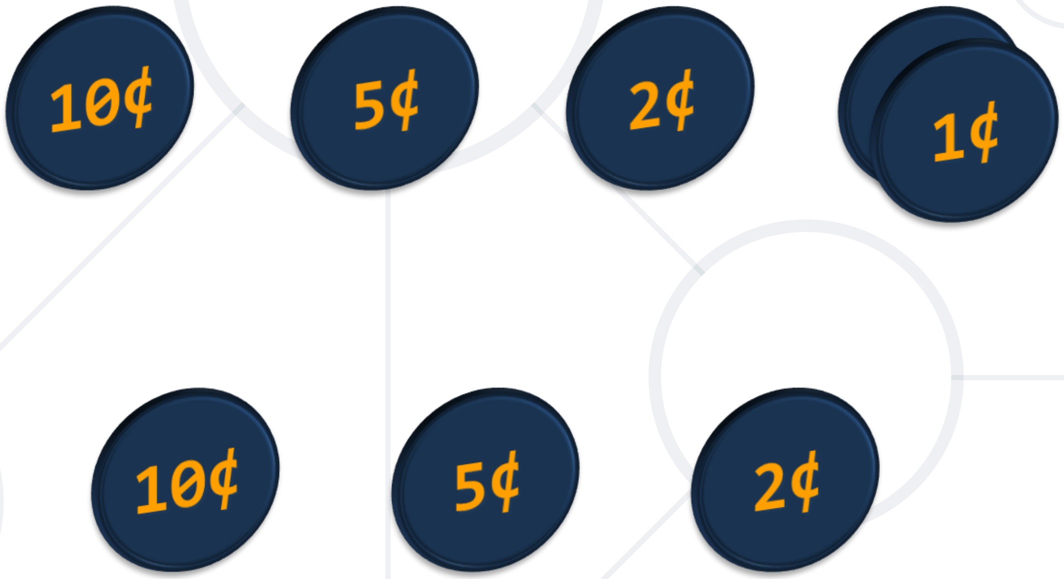
Sum of Coins Visualization

Target: 18

Actual: 15

Sum of Coins Visualization

Target: 18



Actual: 17

Sum of Coins Visualization

Target: 18

Actual: 18

Greedy Failure Cases

Sum of Coins Failure Visualization

Target: 18

Actual: 0

Sum of Coins Failure Visualization

Target: 18

Actual: 10

Sum of Coins Failure Visualization

Target: 18

Actual: 10

Sum of Coins Failure Visualization

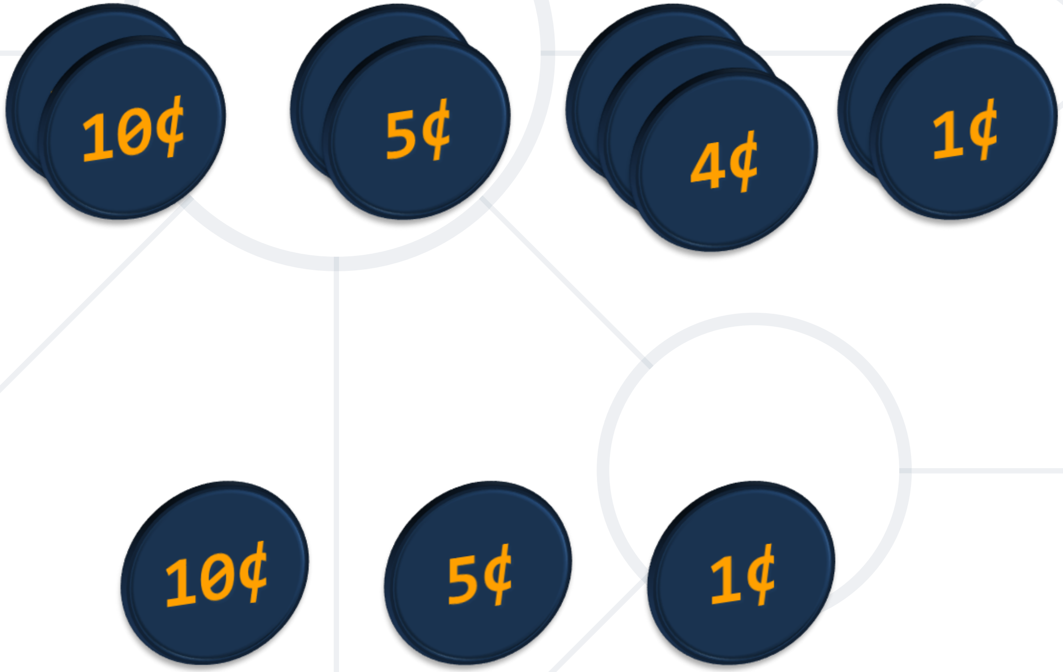
Target: 18

Actual: 15

Sum of Coins Failure Visualization

Target: 18

Actual: 16



Sum of Coins Failure Visualization

Target: 18

Actual: 17

Sum of Coins Failure Visualization

Target: 18

Actual: 18

Sum of Coins Failure Visualization

Target: 18

Solution: Sum of Coins

```python
def choose_coins(coins, target_sum):
    coins.sort(reverse=True)
    index = 0
    used_coins = {}

    while target_sum != 0 and index < len(coins):
   # Next slide

coin_input = input()
coins = list(map(int, coin_input.split(", ")))
target_sum = int(input()
print(choose_coins(coins, target_sum))
```

Solution: Sum of Coins

```python
while target_sum != 0 and index < len(coins):
    count_coins = target_sum // coins[index]
    target_sum %= coins[index]
    if count_coins > 0:
        used_coins[coins[index]] = count_coins
    index += 1

if target_sum != 0:
    return "Error"
else:
    # Sum found
```

Problem: Set Cover

- Write a program that finds the smallest subset of **S**, the union of which = **U** (if it exists)
- You will be given a **set** of integers **U** called "**the Universe**"
- And a set **S** of **n** integer sets whose union = **U**

```
Universe: 1, 2, 3, 4, 5
Number of sets: 4
1
2, 4
5
3
```

→

```
Sets to take (4):
{ 2, 4 }
{ 1 }
{ 5 }
{ 3 }
```

Solution: Set Cover

```
def set_cover(universe, sets):
    universe_set = set(universe)
    chosen_sets = []
    remaining_sets = sets.copy()

    while universe_set and remaining_sets:
    // Next slide

universe = list(map(int, input().split(", ")))
n = int(input())
sets = []
```

Solution: Set Cover

```
    while universe_set and remaining_sets:
            best_set = max(remaining_sets, key=lambda s:
   len(universe_set.intersection(s)))
            if not universe_set.intersection(best_set):
                break
            chosen_sets.append(best_set)
            universe_set -= set(best_set)
            remaining_sets.remove(best_set)

        if universe_set:
            return None
        return chosen_sets
```

Solution: Set Cover

```
for _ in range(n):
    set_elements = list(map(int, input().split(", ")))
    sets.append(set_elements)

result = set_cover(universe, sets)
```

Solution: Set Cover

```python
if result is None:
    print("No solution exists")
else:
    for i in range(len(result)):
        result[i] = sorted(result[i])

    print(f"\nSets to take ({len(result)}): ")
    for s in result:
        print("{", ", ".join(map(str, s)), "}")
```

Searching Algorithms

Linear and Binary Search

Search Algorithm

- **Search algorithm** - an algorithm for finding an item with specified properties among a collection of items
- Different types of searching algorithms:
  - For sub-structures of a given structure
    - A graph, a string, a finite group
  - Search for the **min / max** of a function, etc.

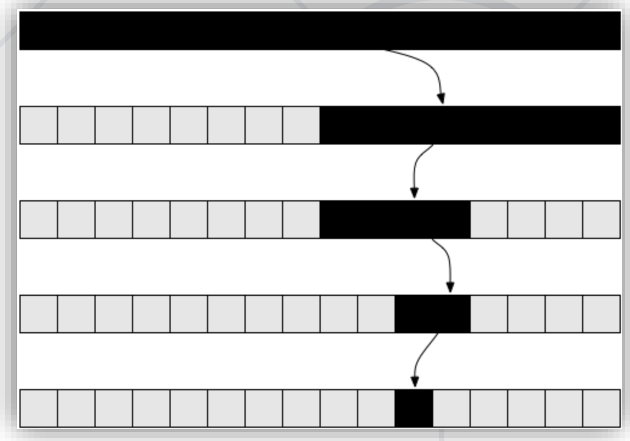
Linear Search

- **Linear search** finds a particular value in a list
  - Checking every one of the elements
  - One at a time, in sequence
  - Until the desired one is found
- Worst & average performance: **O(n)**

```
for each item in the list:
  if that item has the desired value,
    return the item's location
return nothing
```

Binary Search

- *Binary search* finds an item within an ordered data structure
- At each step, compare the input with the middle element
  - The algorithm repeats its action to the left or right sub-structure
- Average performance: **O(log(n))**
- See the *visualization*



Binary Search (Iterative)

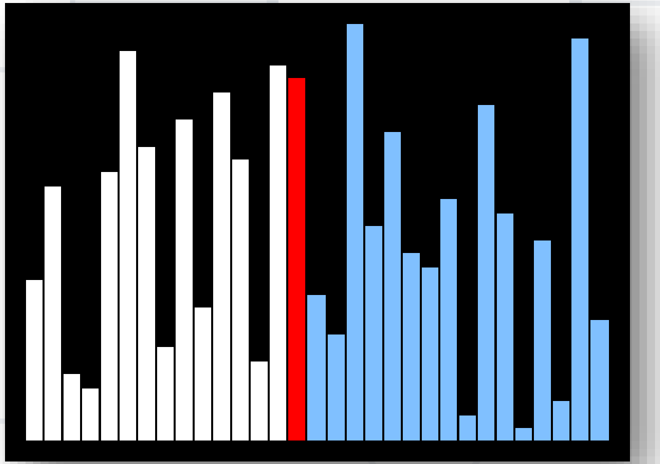
```
def binary_search(numbers, target):
    left = 0
    right = len(numbers) - 1
    while left <= right:
        mid_idx = (left + right) // 2
        mid_el = numbers[mid_idx]
        if mid_el == target:
            return mid_idx
        if mid_el < target:
            left = mid_idx + 1
      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

Simple Sorting Algorithms

Selection, Bubble and Insertion Sort

What is a Sorting Algorithm?

- **Sorting algorithm**
  - An algorithm that rearranges elements in a list
    - In non-decreasing order
  - Elements must be **comparable**
- More formally
  - The **input** is a sequence / list of elements
  - The **output** is a rearrangement / **permutation** of elements
    - In non-decreasing order



Sorting – Example

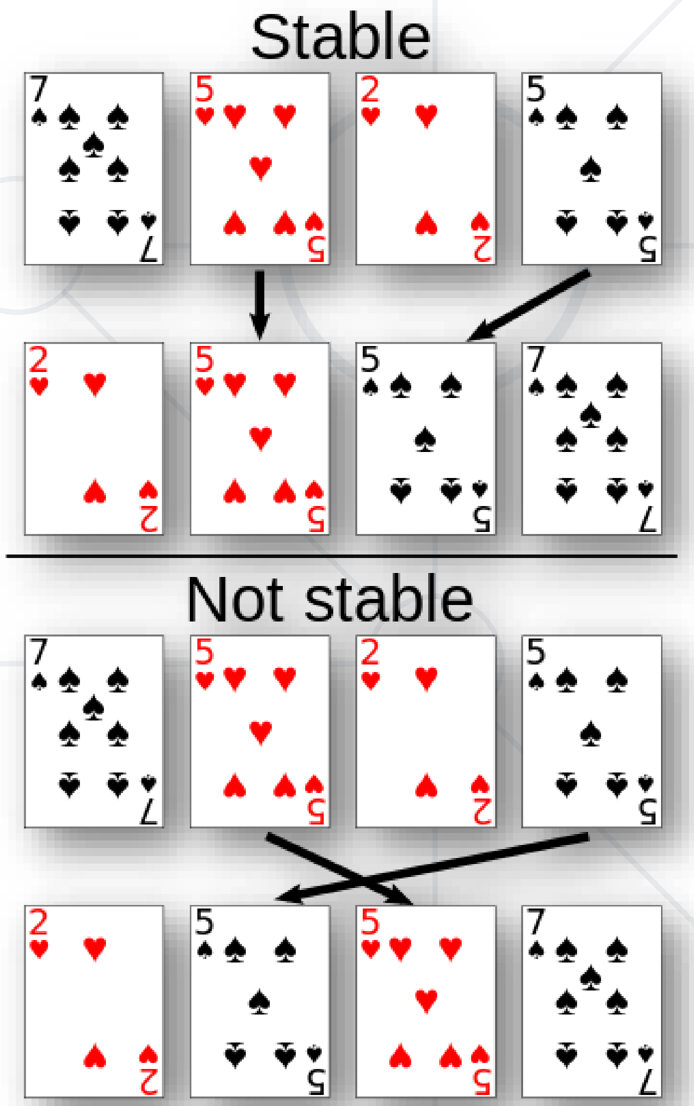
- Efficient sorting algorithms are important for:
  - Producing human-readable output
  - Canonicalizing data – making data uniquely arranged
  - In conjunction with other algorithms, like binary searching
- Example of sorting:

Sorting Algorithms: Classification

- Sorting algorithms are often classified by:
  - Computational **complexity** and memory usage
    - Worst, average and best-case behavior
  - **Recursive** / non-recursive
  - **Stability** – stable / unstable
  - **Comparison-based** sort / non-comparison based

Stability of Sorting

- **Stable** sorting algorithms
  - Maintain the order of equal elements
  - If two items compare as equal, their relative order is preserved
- **Unstable** sorting algorithms
  - Rearrange the equal elements in unpredictable order
- Often **different elements** have the same **key** used for equality comparing

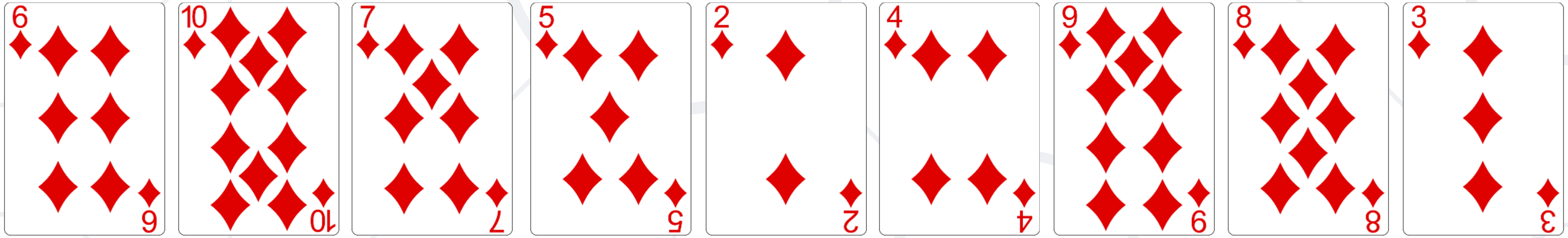


Selection Sort

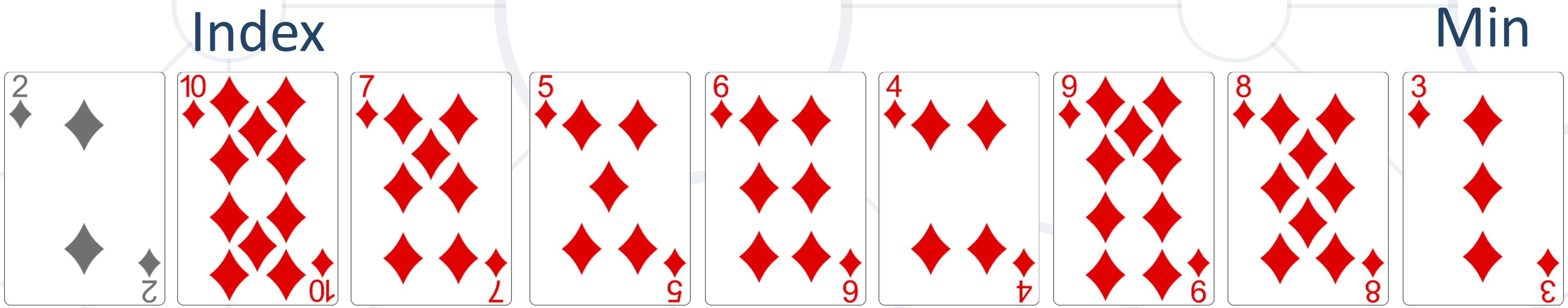
- **Selection sort** – simple, but inefficient algorithm
  - Swap the first with the min element on the right, then the second, etc.
  - Memory: **O(1)**
  - Time: **$O(n^2)$**
  - Stable: No
  - Method: Selection

Selection Sort Visualization

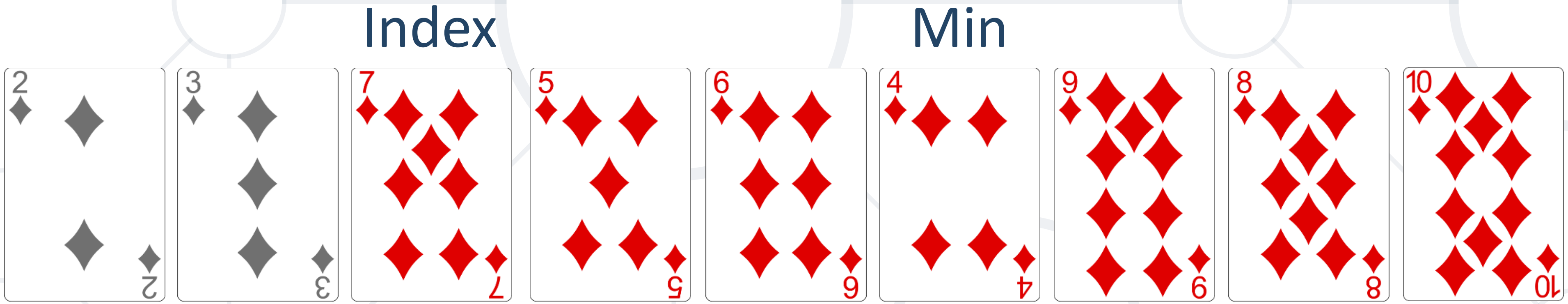
Index Min



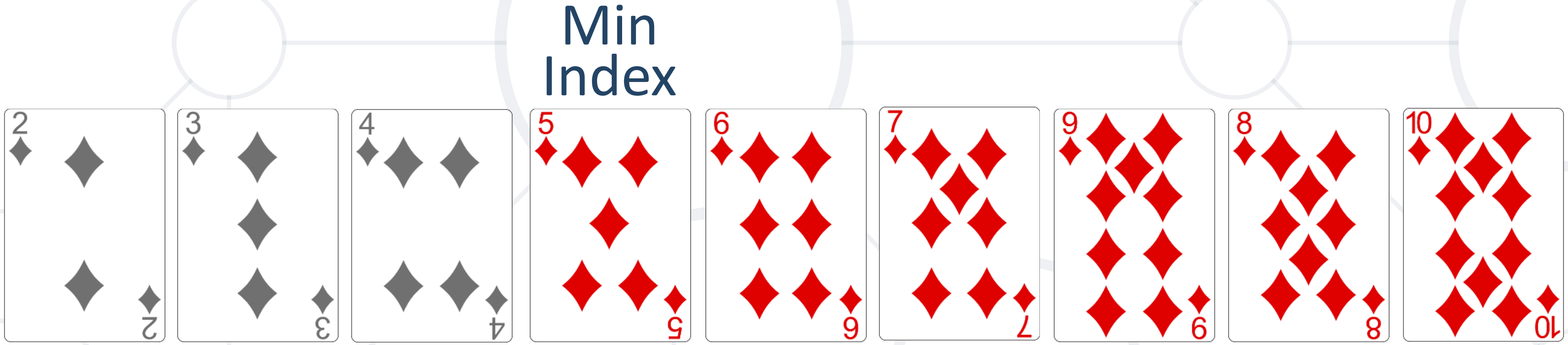
Selection Sort Visualization



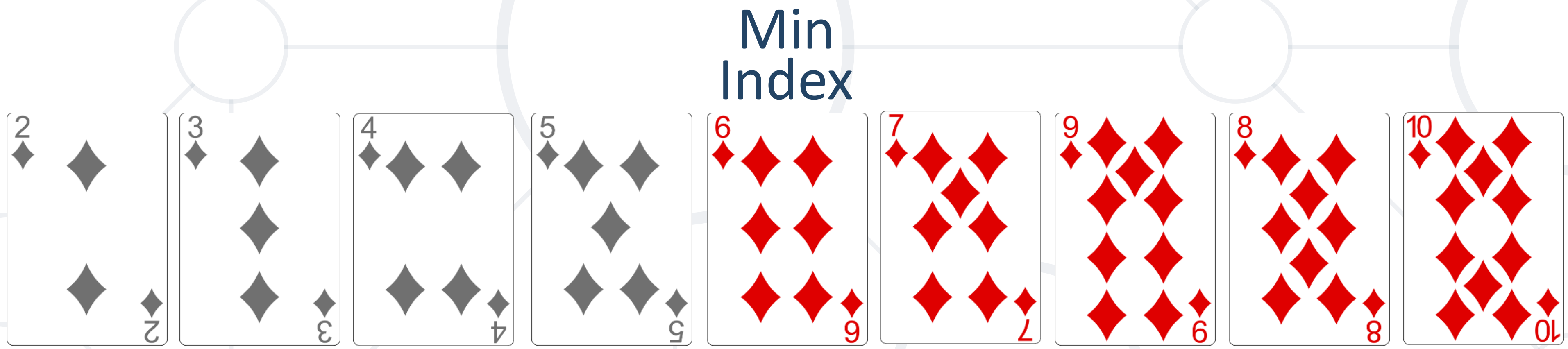
Selection Sort Visualization



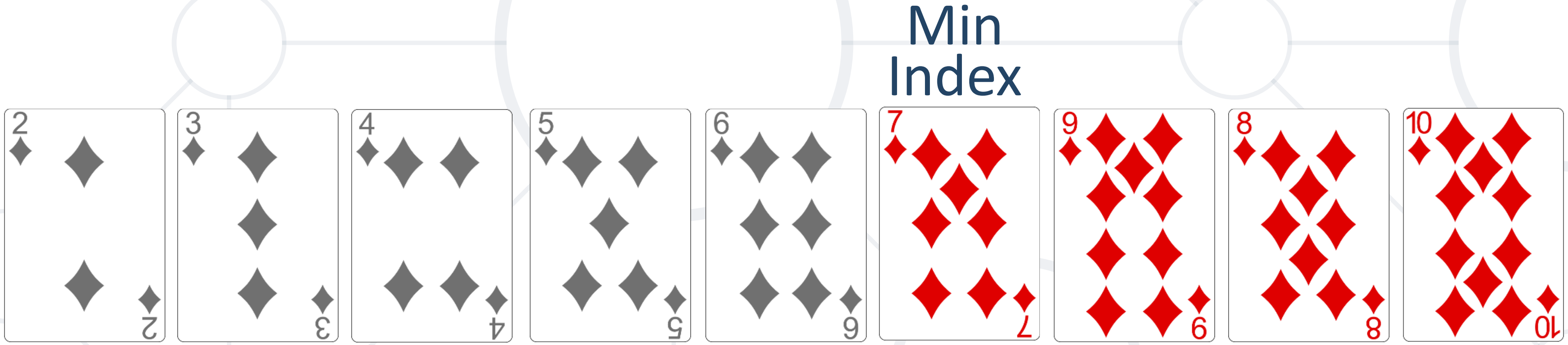
Selection Sort Visualization



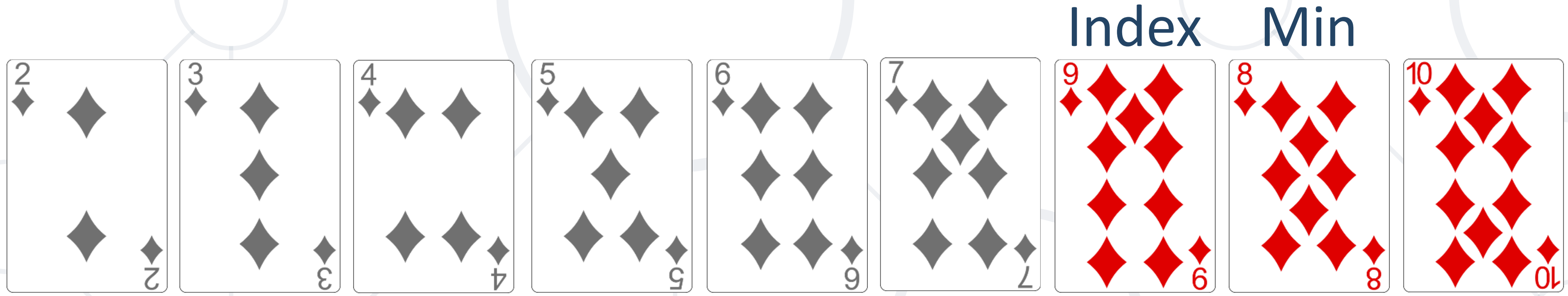
Selection Sort Visualization



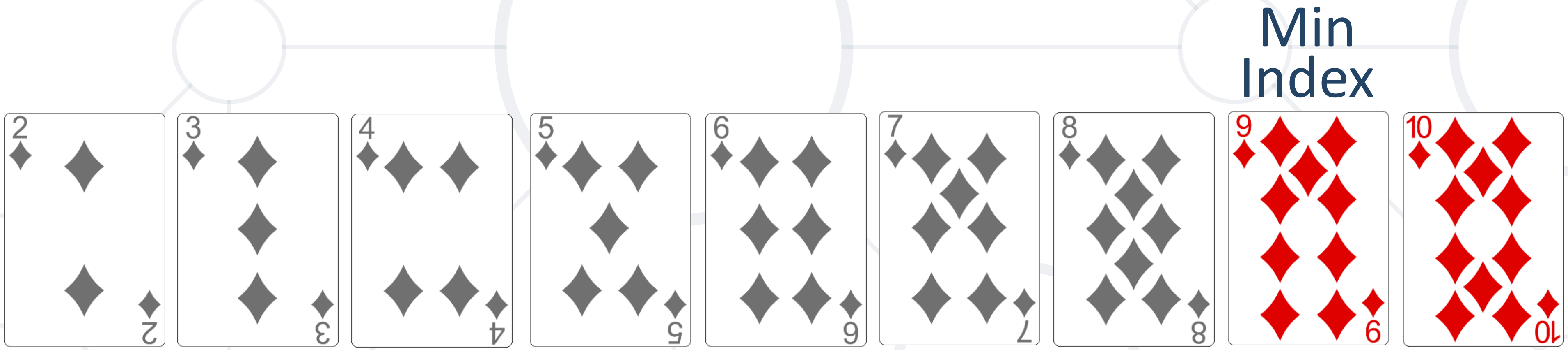
Selection Sort Visualization



Selection Sort Visualization

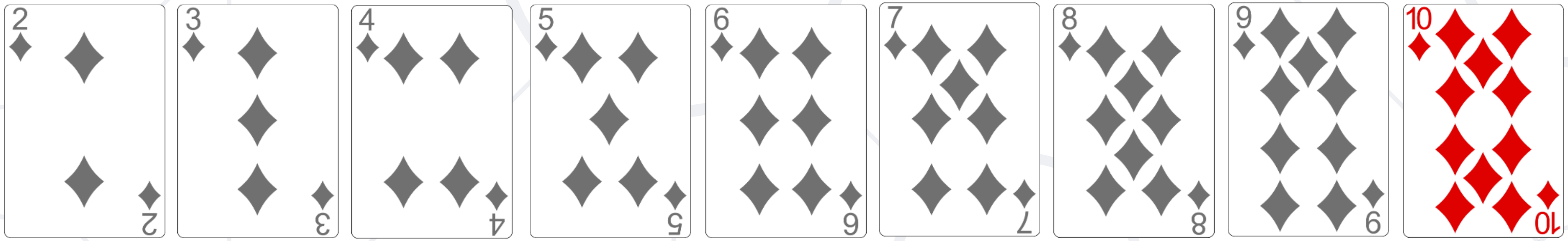


Selection Sort Visualization

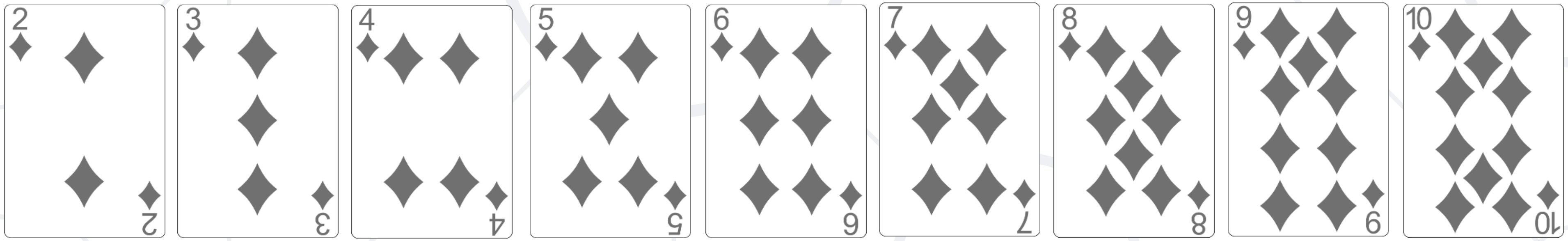


Selection Sort Visualization

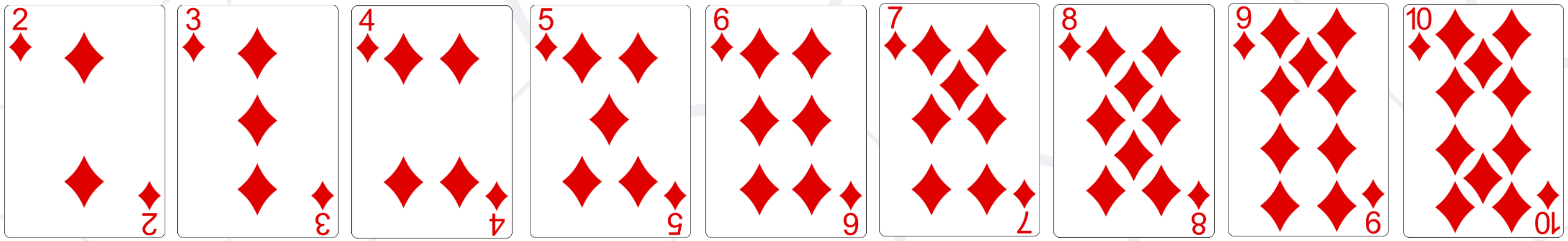
Min Index



Selection Sort Visualization



Selection Sort Visualization



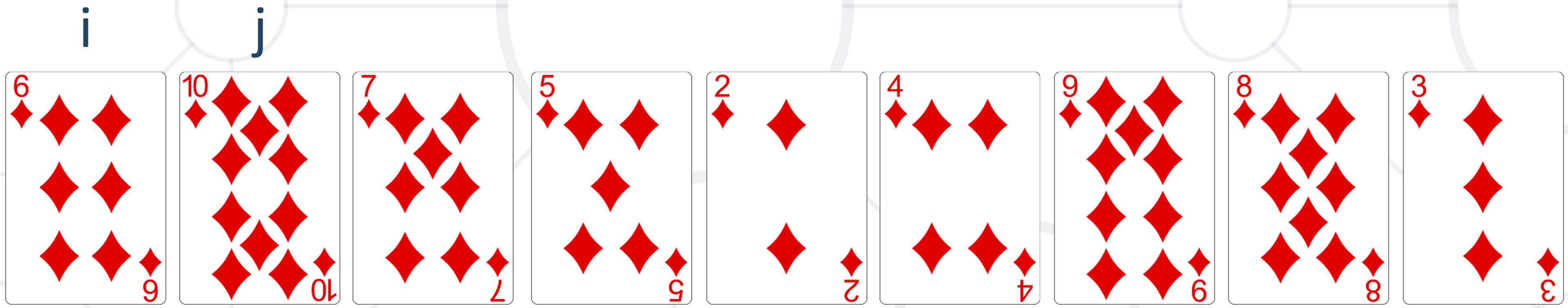
Selection Sort

```
def selection_sort(nums):
    for idx in range(len(nums)):
        min_idx = idx
        for curr_idx in range(idx + 1, len(nums)):
            if nums[curr_idx] < nums[min_idx]:
                min_idx = curr_idx

        nums[idx], nums[min_idx] = nums[min_idx], nums[idx]
```

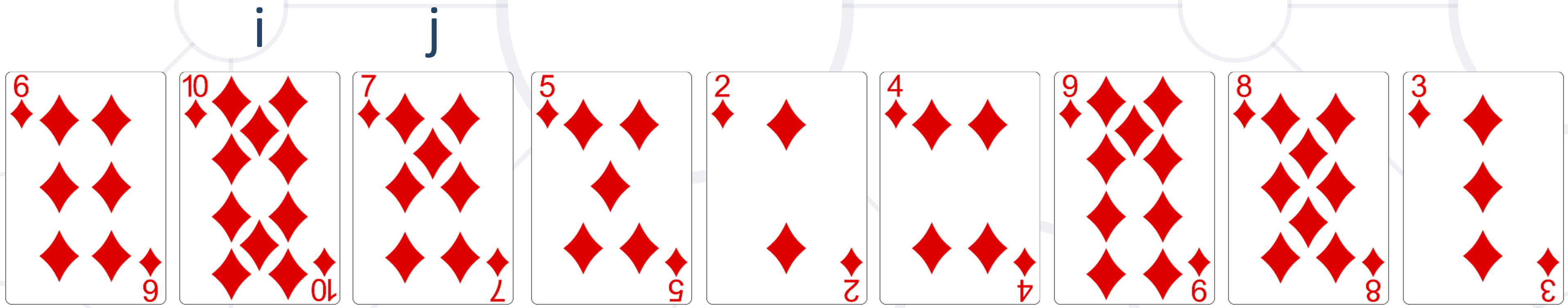
Bubble Sort

- **Bubble sort** – simple, but inefficient algorithm
- Swaps to neighbor elements when not in order until sorted
  - Memory: **O(1)**
  - Time: **$O(n^2)$**
  - Stable: Yes
  - Method: Exchanging

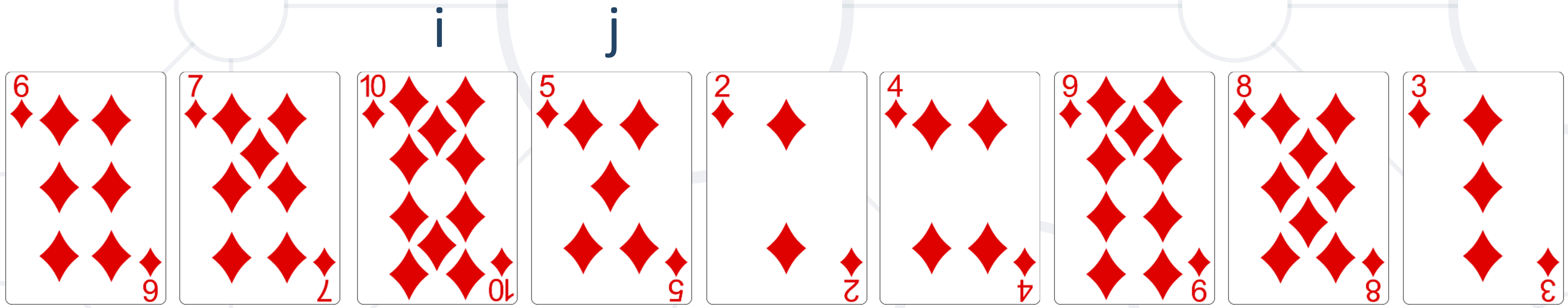
Bubble Sort Visualization



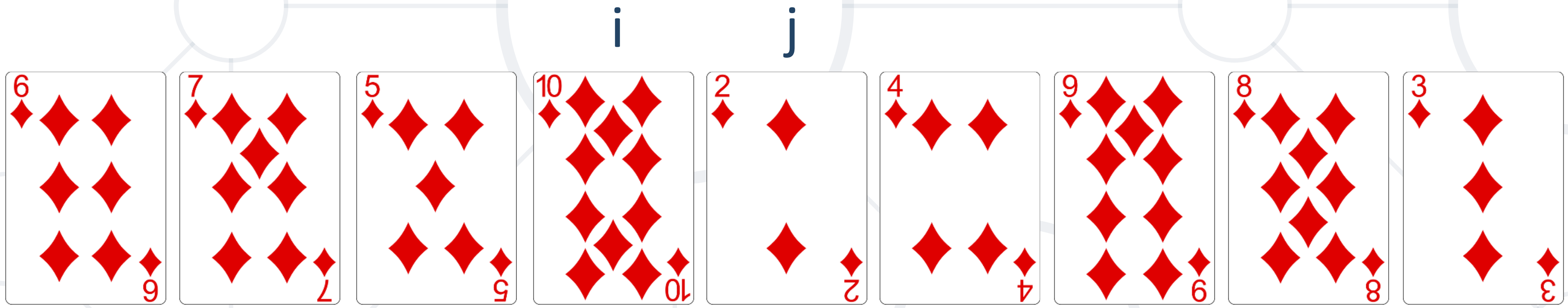
Bubble Sort Visualization



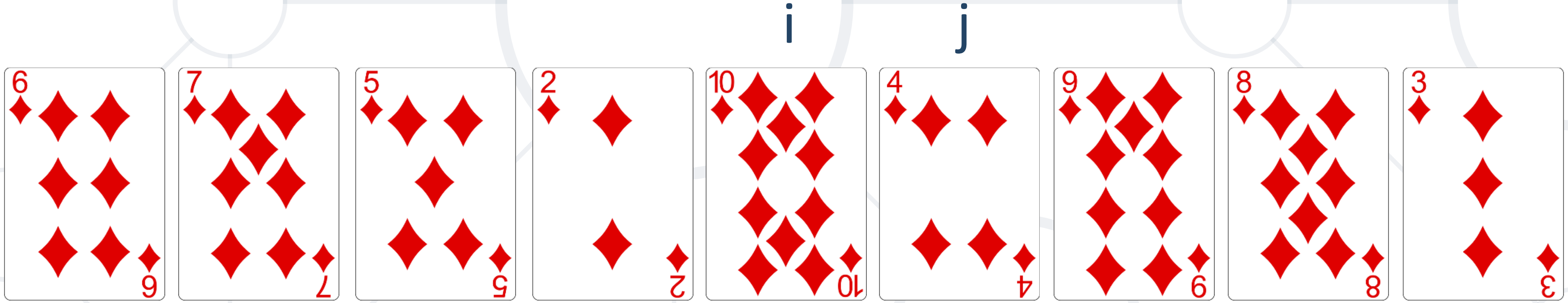
Bubble Sort Visualization



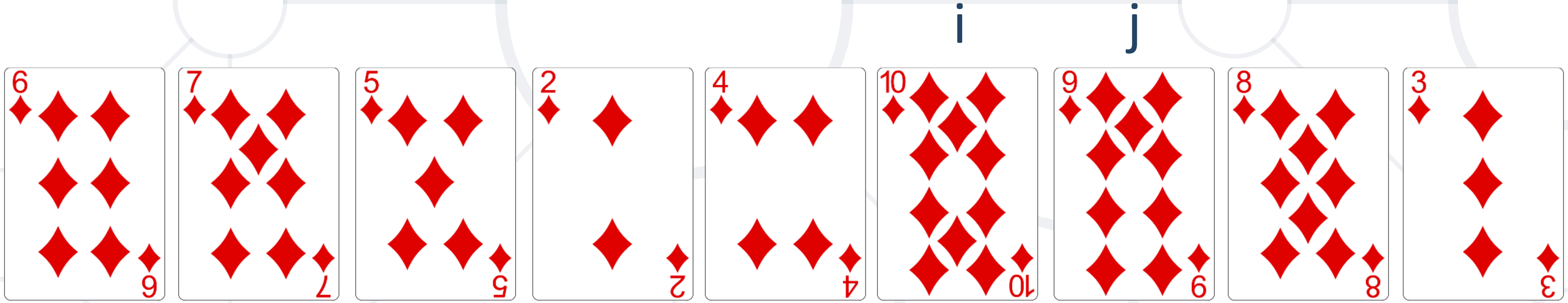
Bubble Sort Visualization



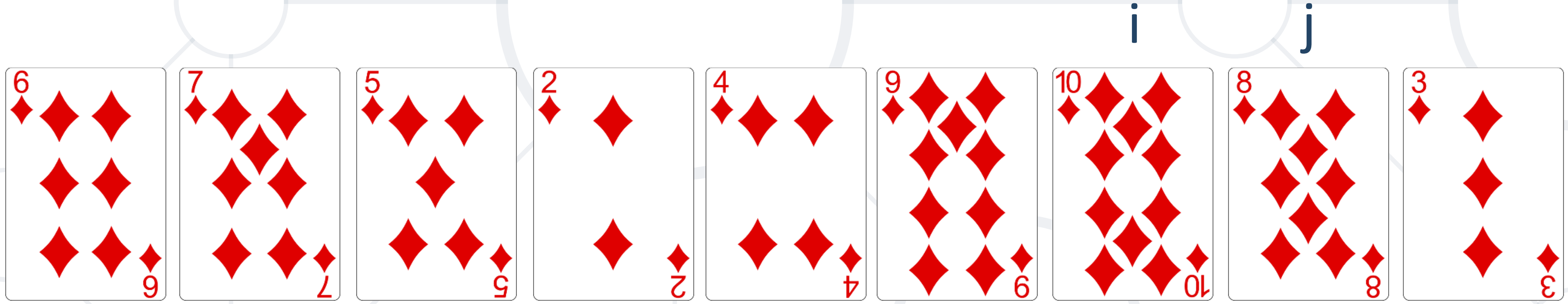
Bubble Sort Visualization



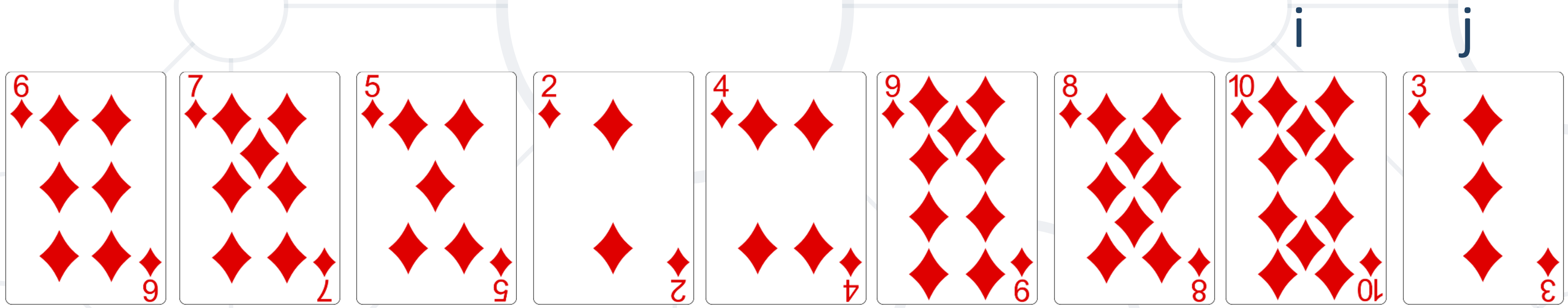
Bubble Sort Visualization



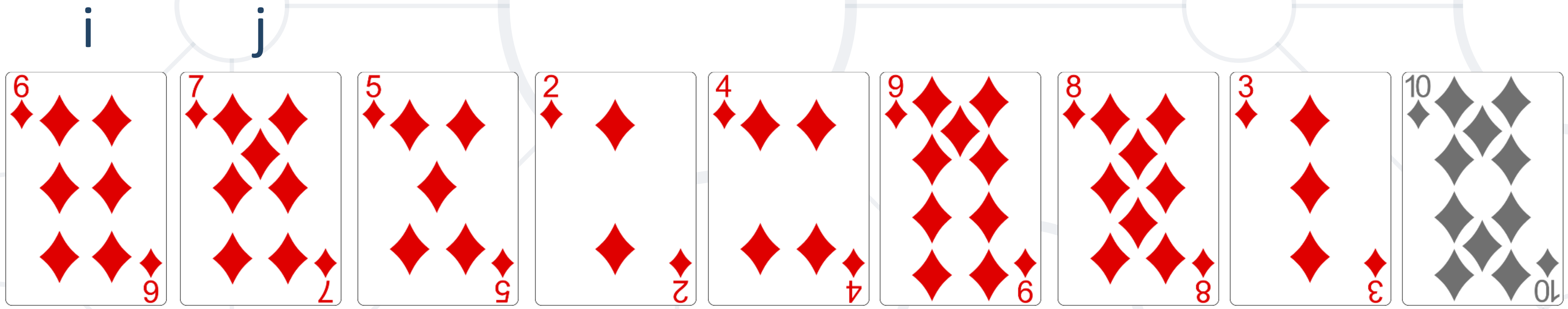
Bubble Sort Visualization



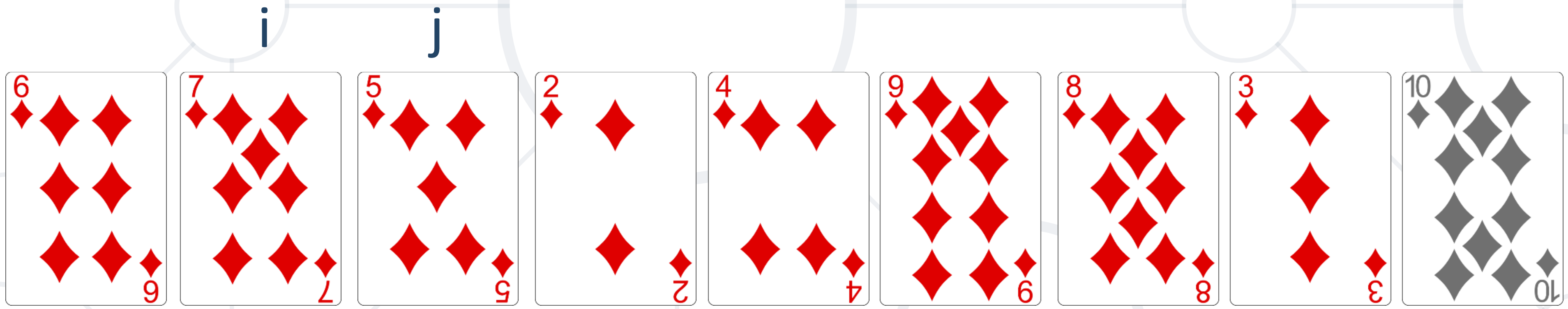
Bubble Sort Visualization



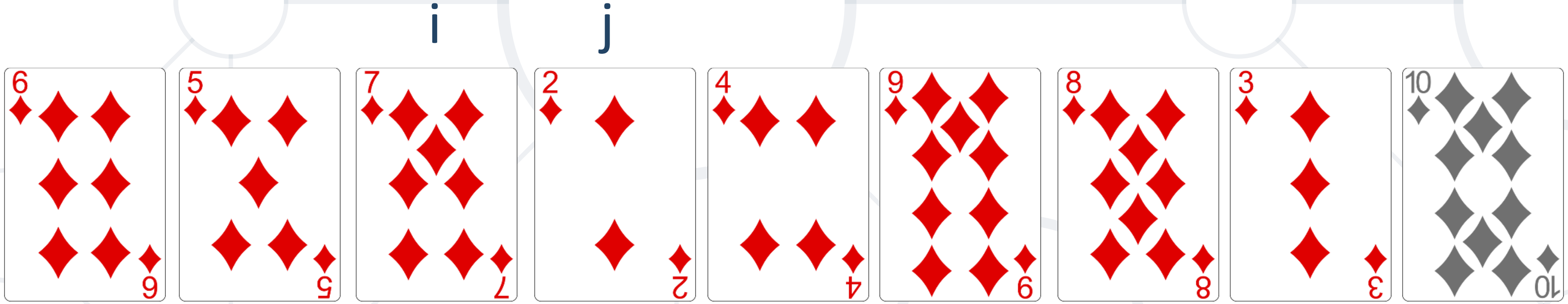
Bubble Sort Visualization



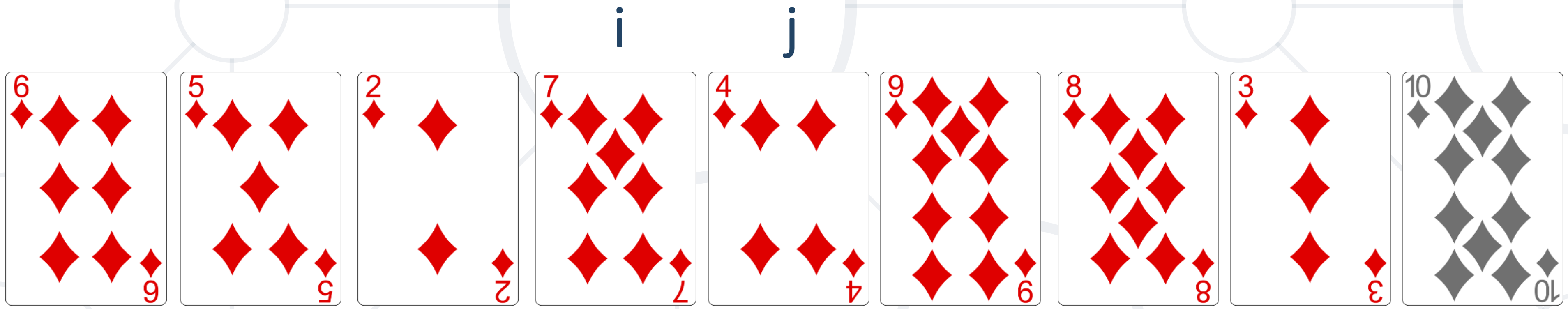
Bubble Sort Visualization



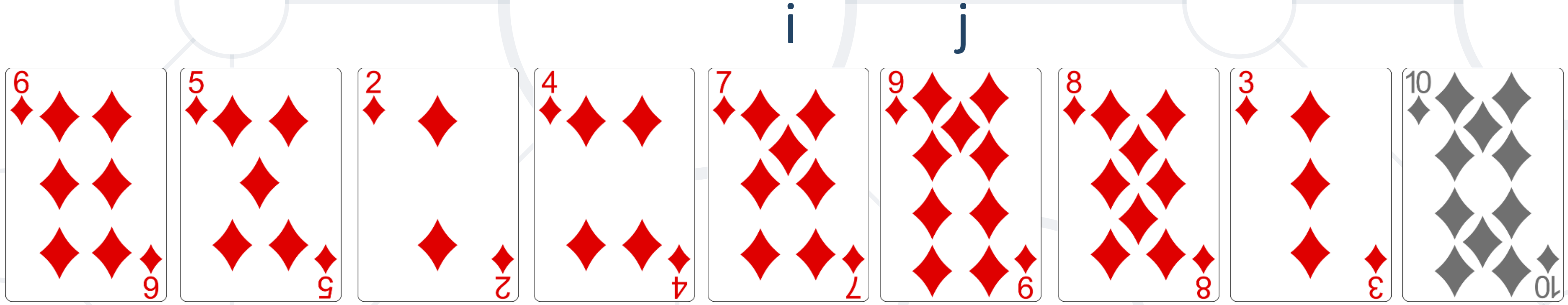
Bubble Sort Visualization



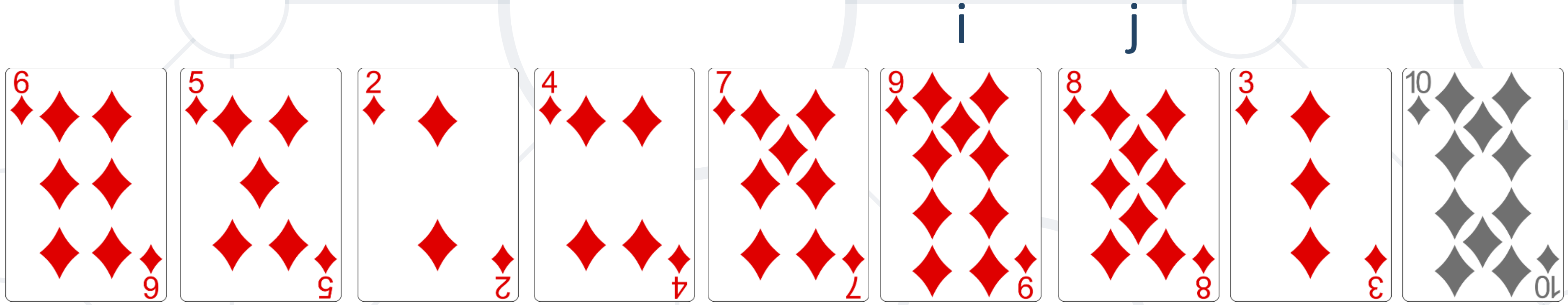
Bubble Sort Visualization



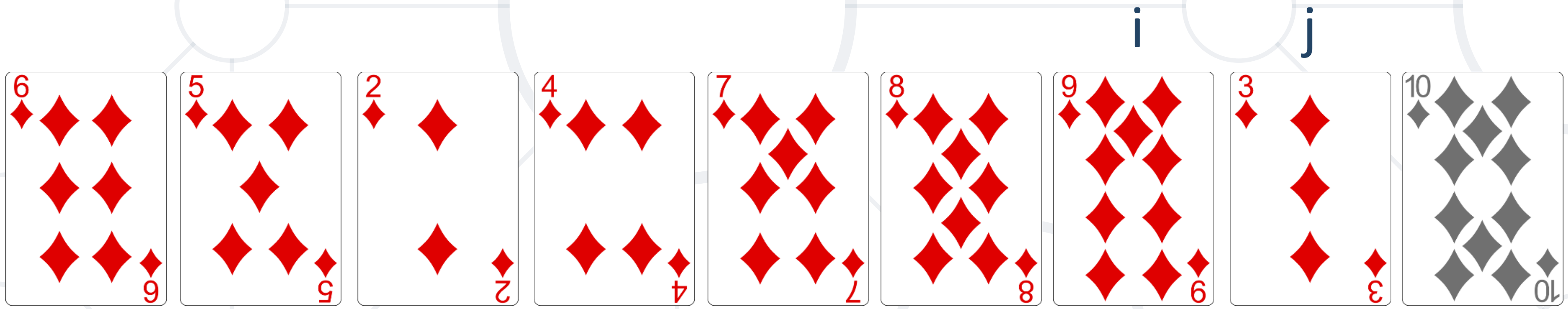
Bubble Sort Visualization



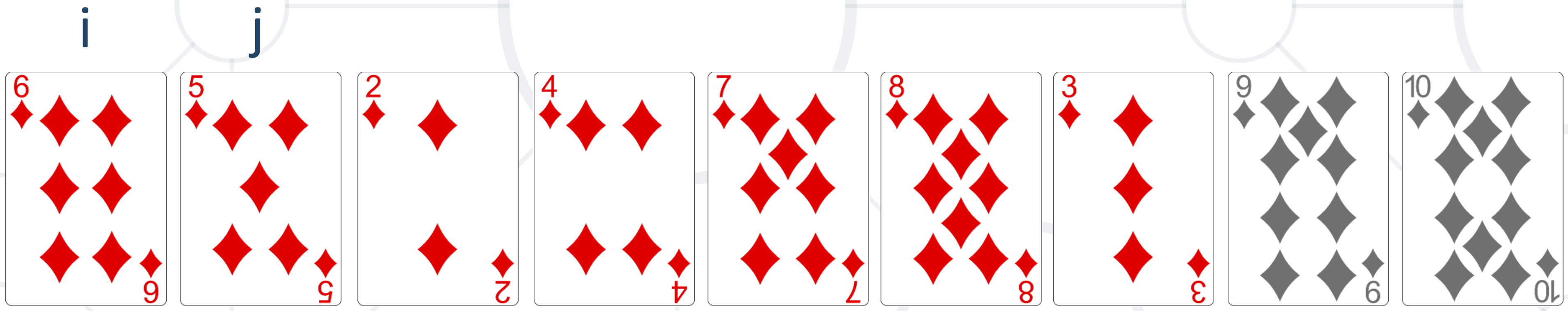
Bubble Sort Visualization



Bubble Sort Visualization



Bubble Sort Visualization



Bubble Sort

```
def bubble_sort(nums):
    is_sorted = False
    i = 0

    while not is_sorted:
        is_sorted = True
        for j in range(1, len(nums) - i):
            if nums[j - 1] > nums[j]:
                nums[j], nums[j - 1] = nums[j - 1], nums[j]
                is_sorted = False

        i += 1
```

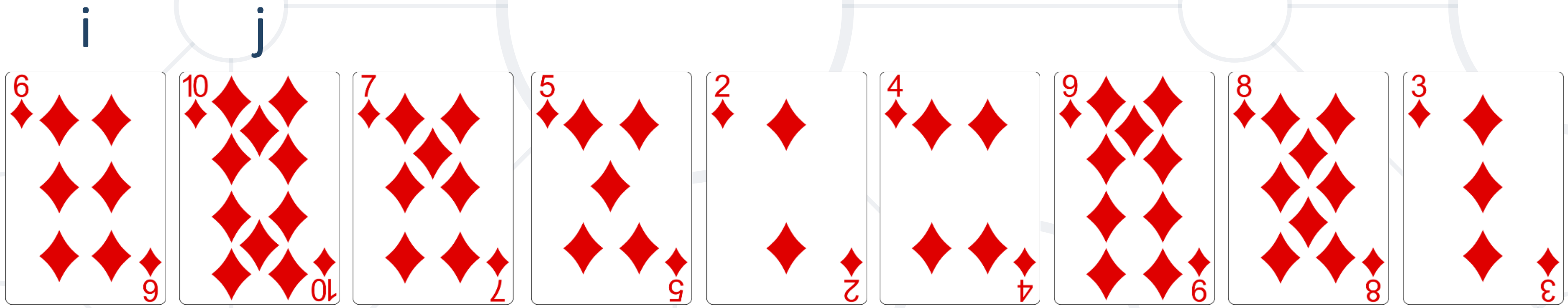
Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection	n^2	n^2	n^2	1	No	Selection
Bubble	n	n^2	n^2	1	Yes	Exchanging

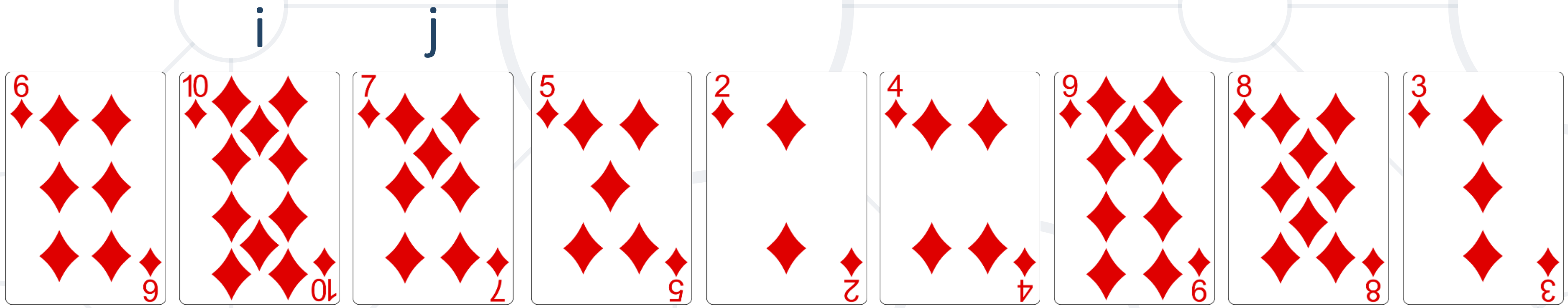
Insertion Sort

- **Insertion Sort** – simple, but inefficient algorithm
  - Move the first unsorted element left to its place
  - Memory: **O(1)**
  - Time: **$O(n^2)$**
  - Stable: **Yes**
  - Method: **Insertion**

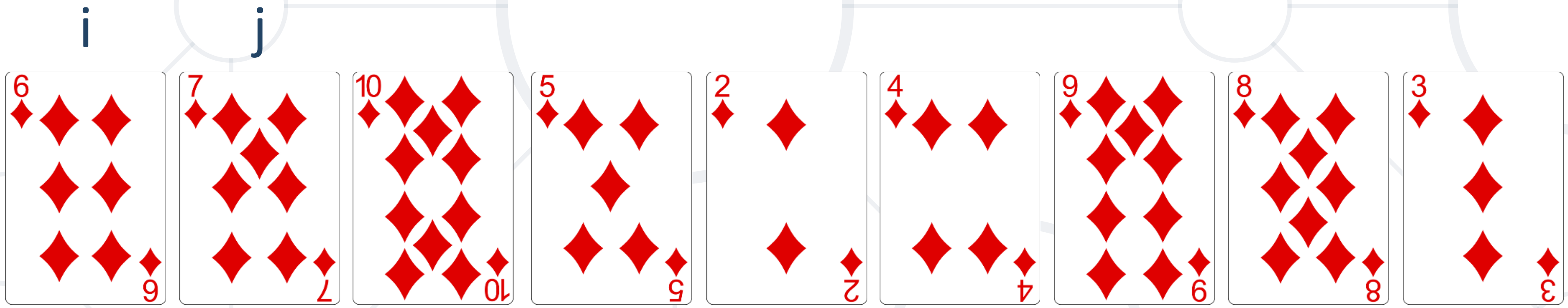
Insertion Sort Visualization



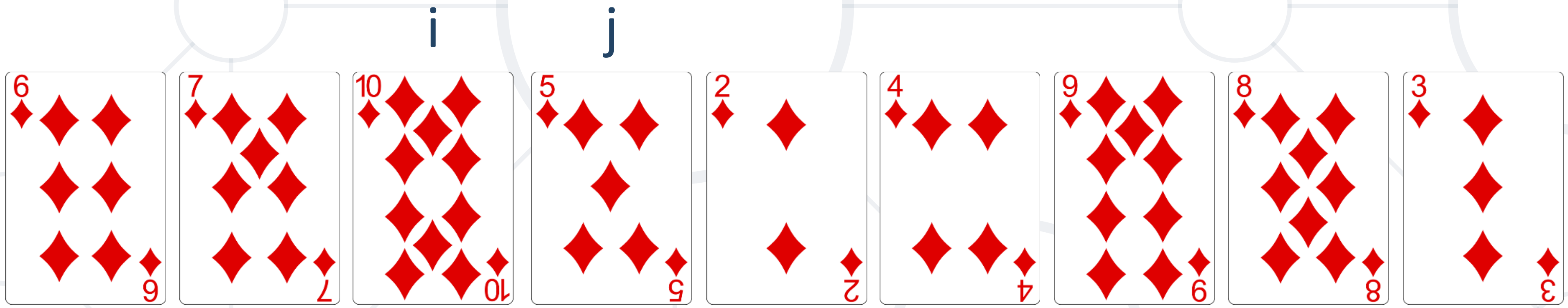
Insertion Sort Visualization



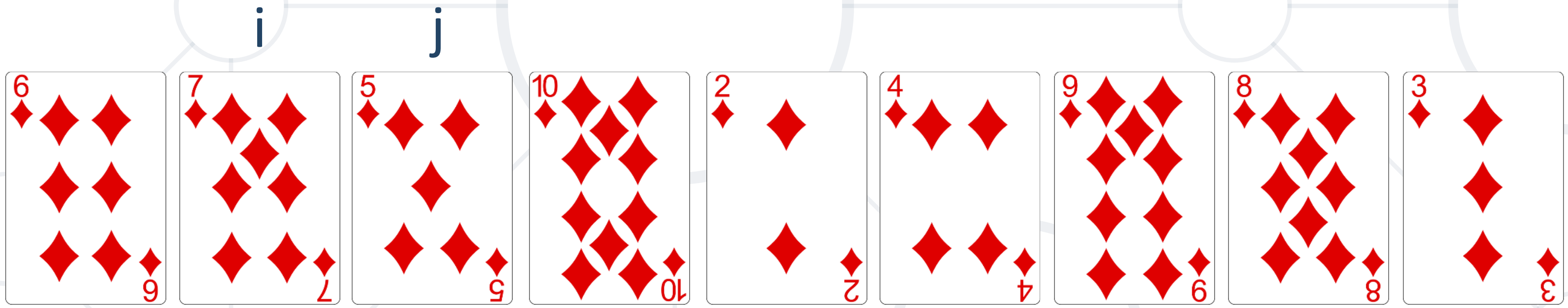
Insertion Sort Visualization



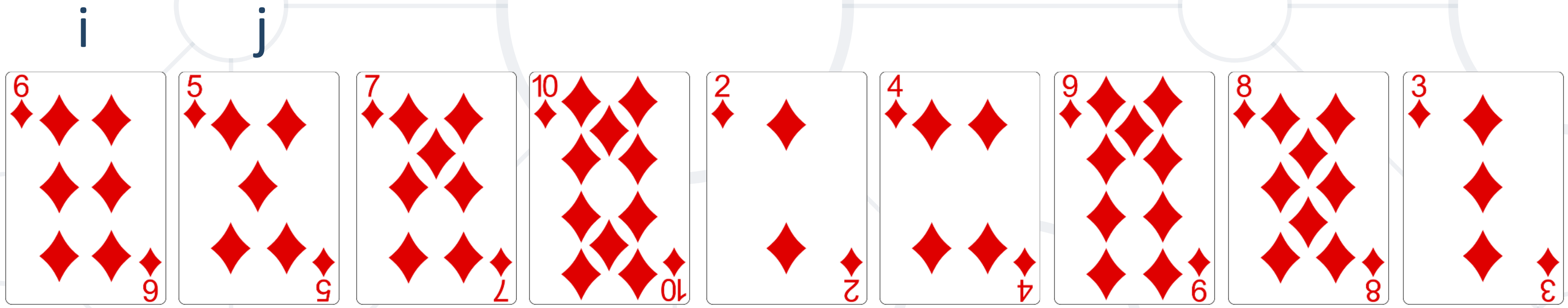
Insertion Sort Visualization



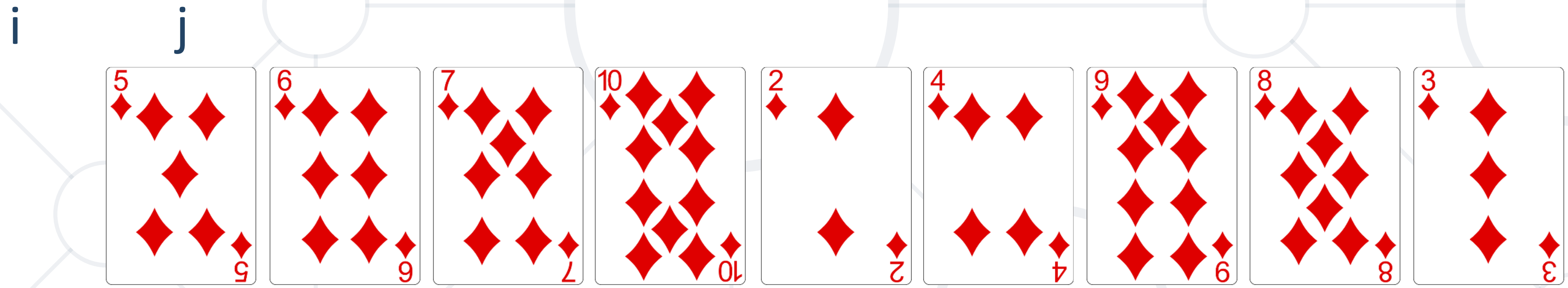
Insertion Sort Visualization



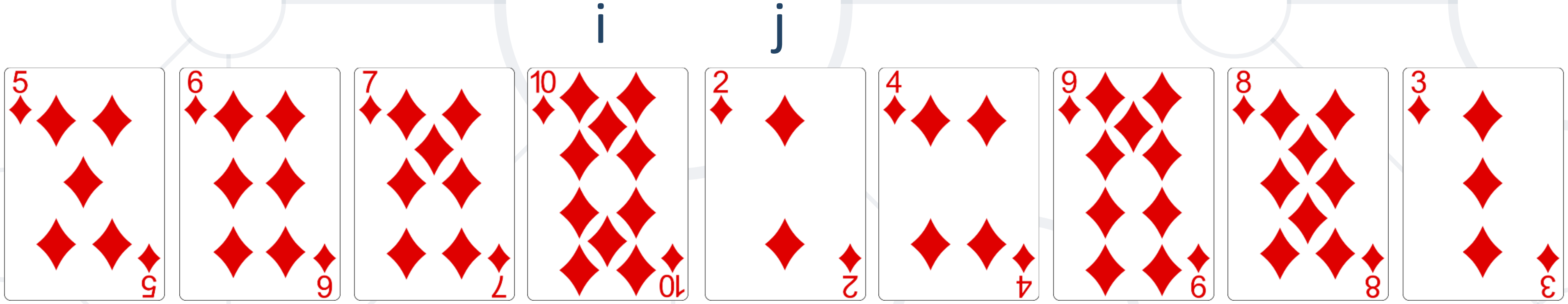
Insertion Sort Visualization



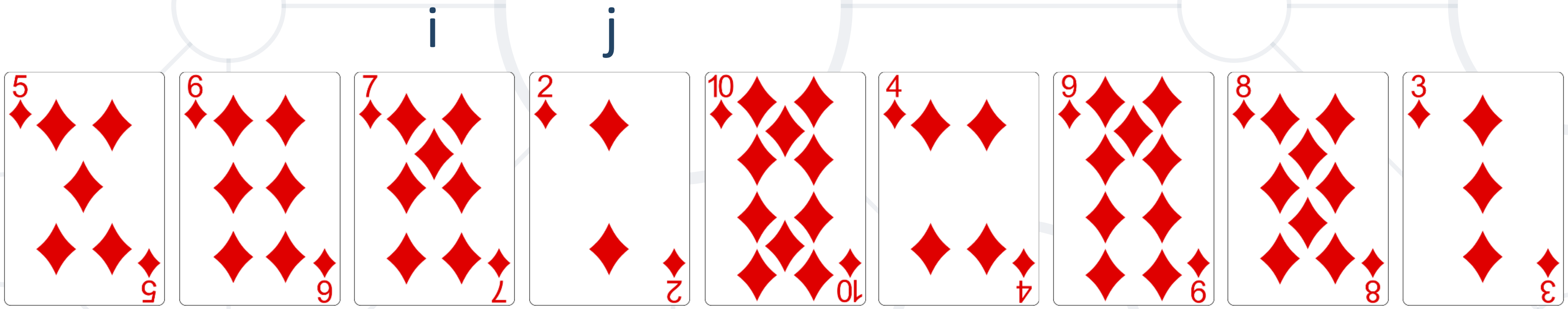
Insertion Sort Visualization



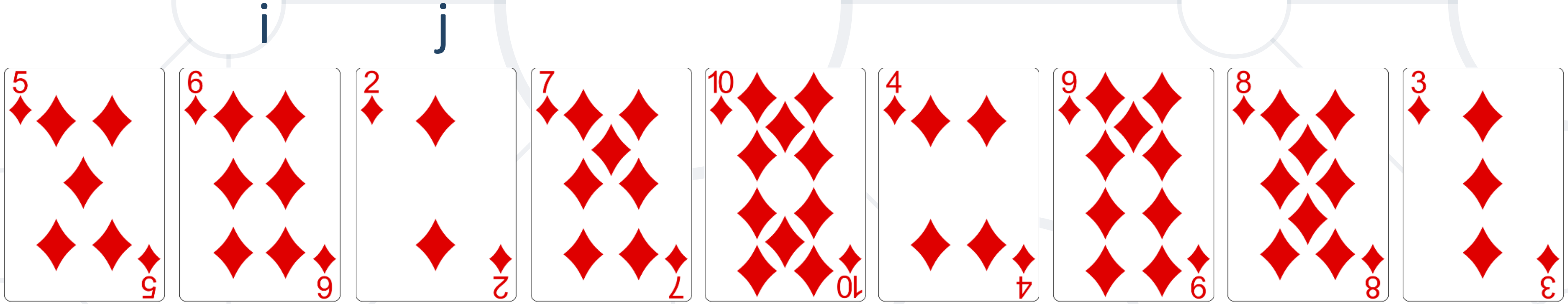
Insertion Sort Visualization



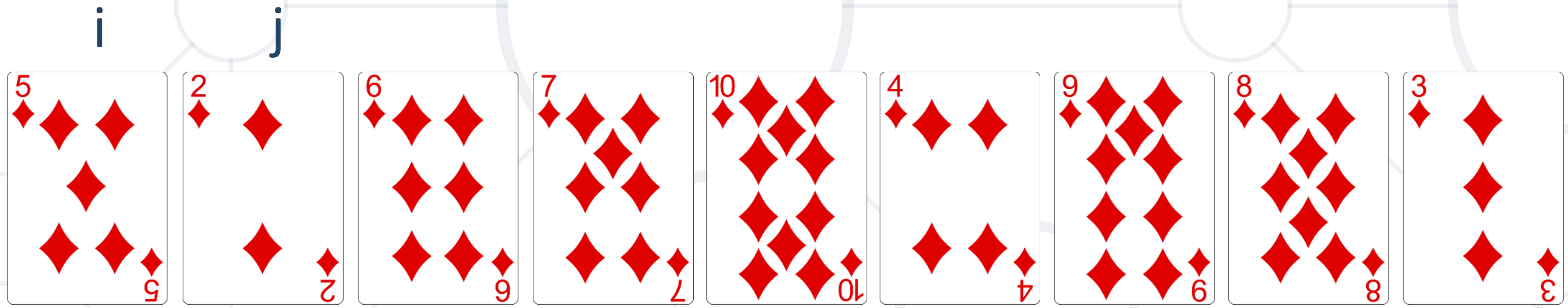
Insertion Sort Visualization



Insertion Sort Visualization

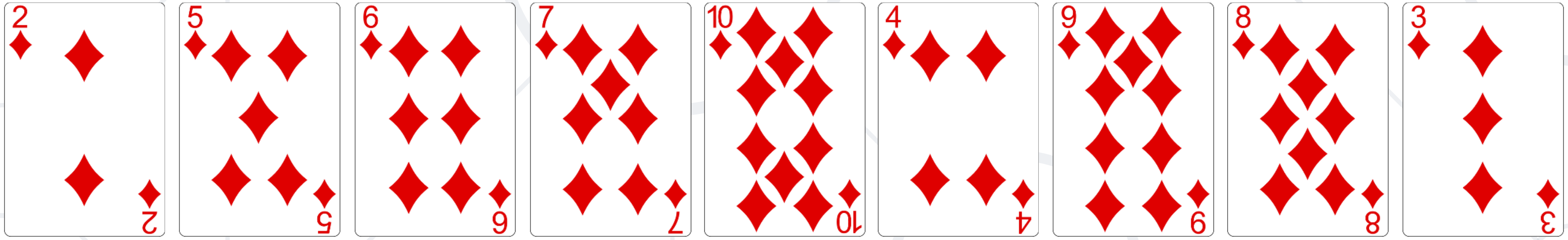


Insertion Sort Visualization



Insertion Sort Visualization

i j



Insertion Sort

```python
def insertion_sort(nums):
    for i in range(1, len(nums)):
        key = nums[i]
        j = i - 1

        while j >= 0 and nums[j] > key:
            nums[j + 1] = nums[j]
            j -= 1

        nums[j + 1] = key
```

Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection	n^2	n^2	n^2	1	No	Selection
Bubble	n	n^2	n^2	1	Yes	Exchanging
Insertion	n	n^2	n^2	1	Yes	Insertion

Summary

- Algorithmic Complexity
- Recursion - a function calls itself
- Brute-Force - trying all the possible solutions
- Greedy - picking a locally optimal solution
- Searching algorithms:
  - Binary Search, Linear Search
- Sorting algorithms:
  - Selection sort, Bubble sort, Insertion sort

Software University

Questions?

SoftUni

- Software University
- SoftUni Creative
- SoftUni Digital
- SoftUni Foundation
- SoftUni Kids
- СофтУни БУДИТЕЛ
- Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg