Basic Algorithms

Recursion, Greedy, Sorting and Searching

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Table of Contents

Have a Question?

sli.do

#java-advanced

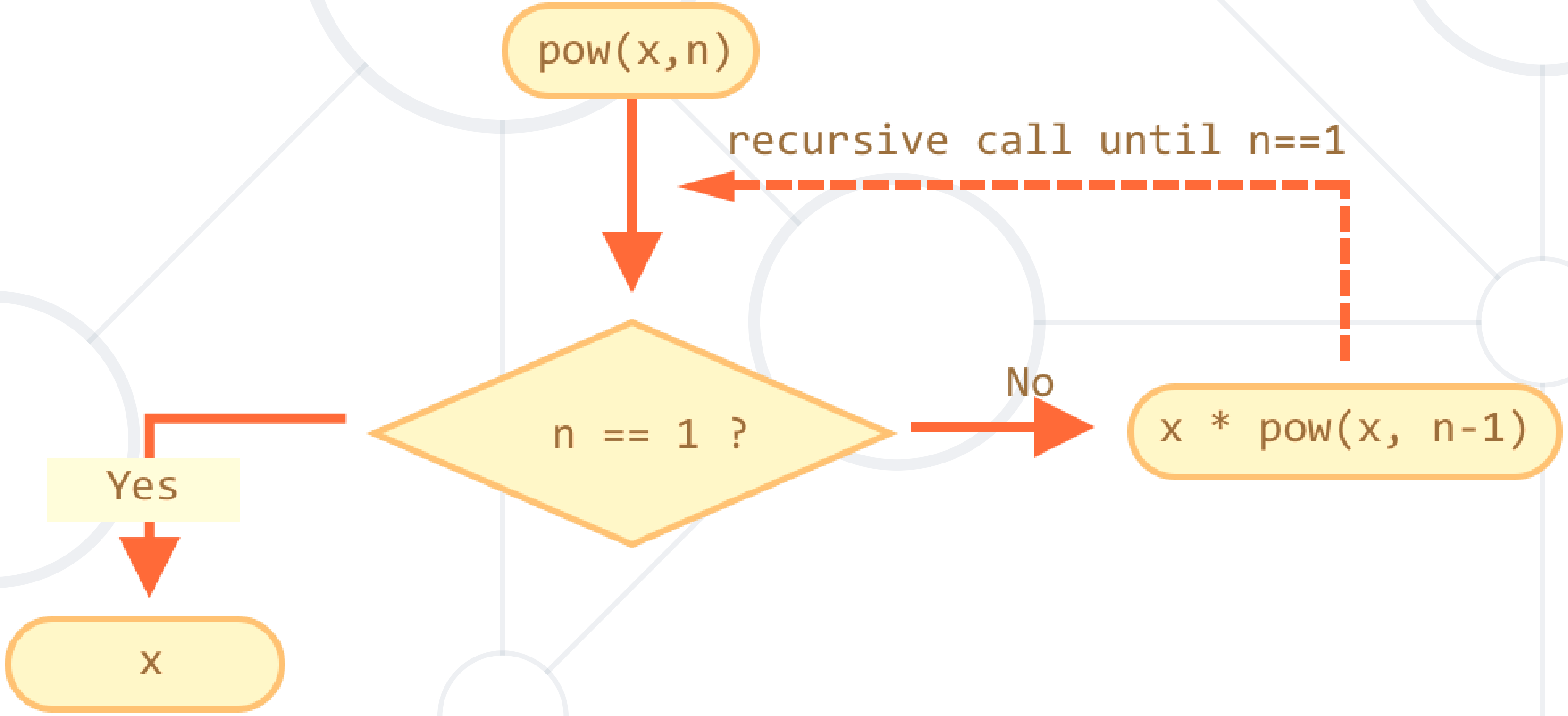
Recursion

What is Recursion?

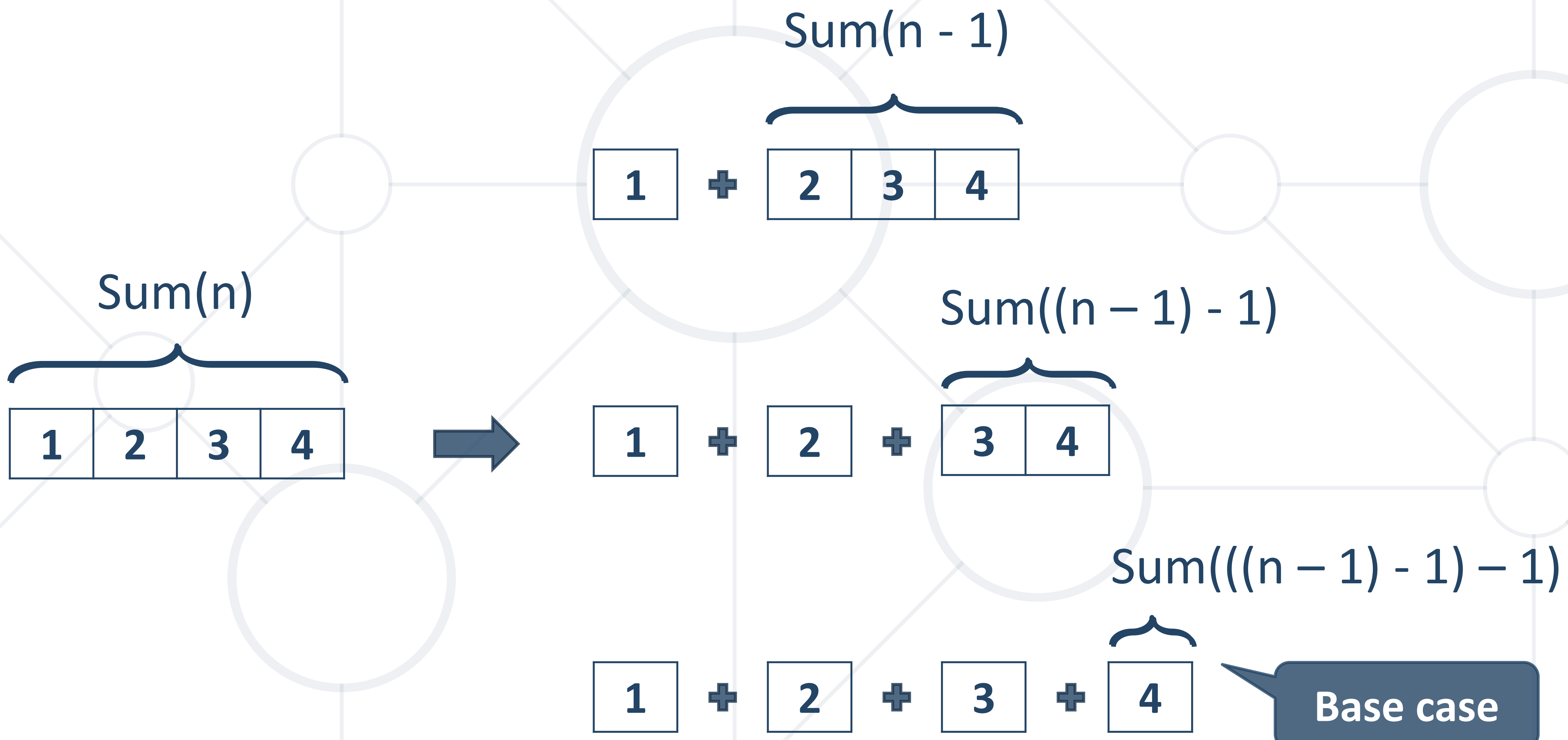
- A function or a method that **calls itself one or more** times until a specified **condition** is **met**
- When it is, the rest of **each** repetition is processed **from** the **last** one called **to** the **first**

How Does It Work?

- The function or method has a **base case**
- **Each step** of the recursion should **move towards** the **base case**



Example: Array Sum



Example: Recursive Factorial

- Recursive definition of n! (n factorial):

5	→	120
10	→	3628800

- Pseudocode

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

N!

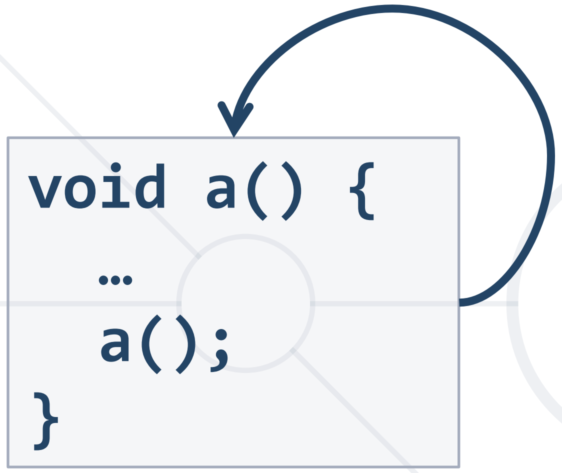
Recursion Pre-Actions and Post-Actions

- Recursive methods have 3 parts:
  - **Pre-actions** (before calling the recursion)
  - **Recursive calls** (step-in)
  - **Post-actions** (after returning from recursion)

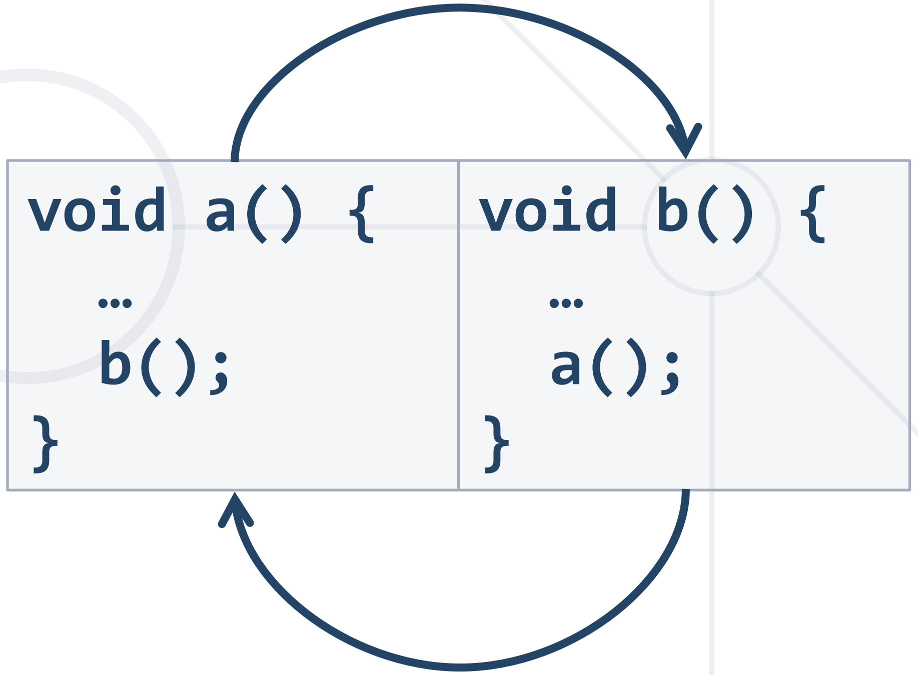
```
static void recursion() {
  // Pre-actions
  recursion();
  // Post-actions
}
```

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



```
void a() {
  …
  a();
}
```

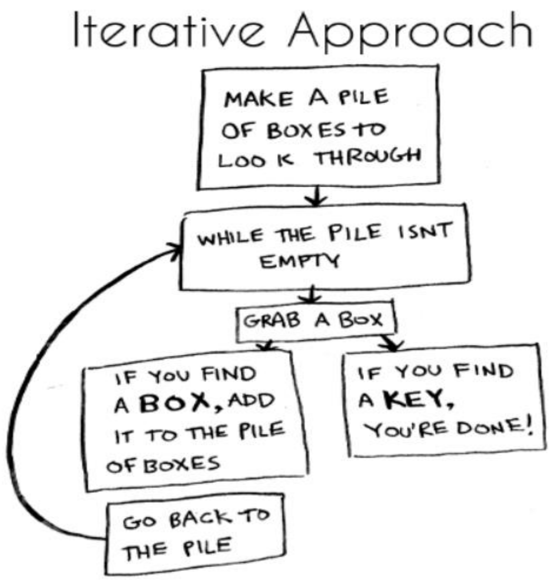


```
void a() {
  …
  b();
}
```

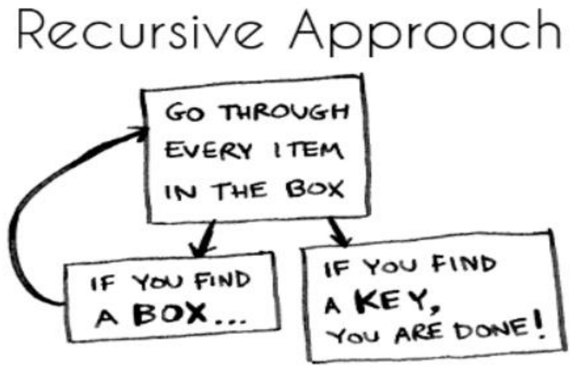
```
void b() {
  …
  a();
}
```

Iterative vs. Recursive Approach

- A function repeats a defined process until a condition fails
- A function that calls itself repeatedly until a certain condition is met



Iterative Approach
MAKE A PILE OF BOXES TO LOOK THROUGH
WHILE THE PILE ISNT EMPTY
GRAB A BOX
IF YOU FIND A BOX, ADD IT TO THE PILE OF BOXES
IF YOU FIND A KEY, YOU'RE DONE!
GO BACK TO THE PILE



Recursive Approach
GO THROUGH EVERY ITEM IN THE BOX
IF YOU FIND A BOX...
IF YOU FIND A KEY, YOU ARE DONE!

Recursion

Problem: Recursive Array Sum

- Write a **recursive method** that:
  - Finds the sum of all numbers stored in an `int[] array`
  - Read numbers from the console

```
1 2 3 4  →  10
```

```
-1 0 1   →  0
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Recursive Array Sum

```
static int sum(int[] array, int index) {
  if (index == array.length - 1) {
    return array[index];
  }
  return array[index] + sum(array, index + 1);
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Problem: Recursive Factorial

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

```
5
```

```
120
```

```
10
```

```
3628800
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Recursive Factorial

```
static long factorial(int num) {
  if (num == 0) {
    return 1;
  }
  return num * factorial(num - 1);
}
```

Base case

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

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

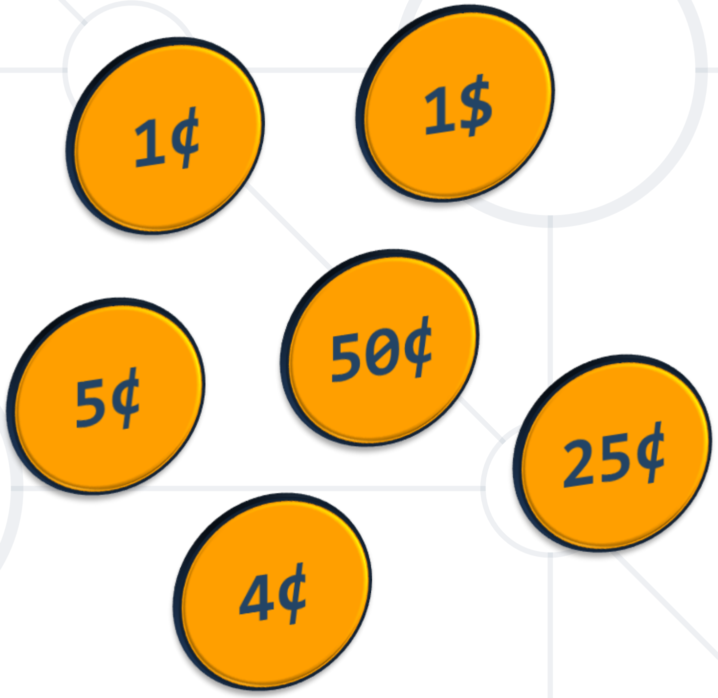
Optimization Problems

- 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

Greedy Algorithms

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.



Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

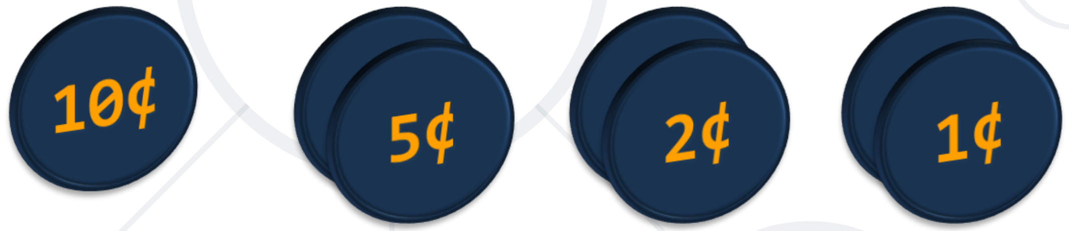
Sum of Coins Visualization

Target: 18

Actual: 0

Sum of Coins Visualization

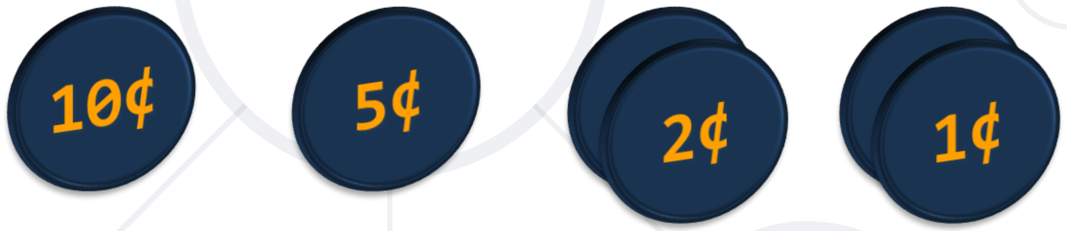
Target: 18



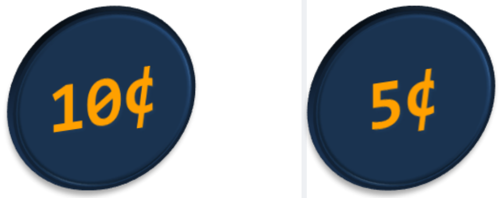
Actual: 10

Sum of Coins Visualization

Target: 18

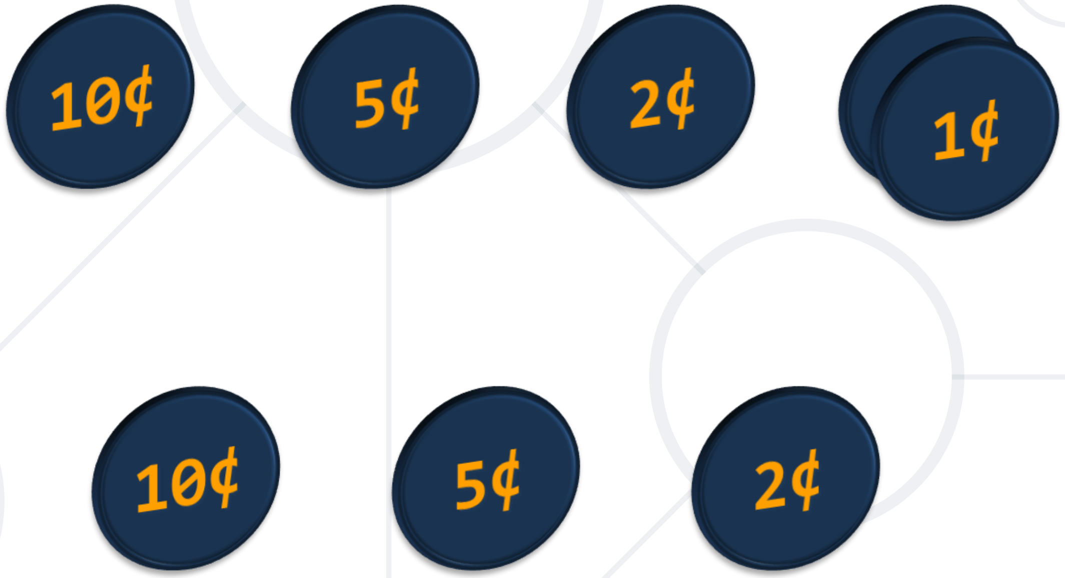


Actual: 15



Sum of Coins Visualization

Target: 18



Actual: 17

Sum of Coins Visualization

Target: 18

10¢
5¢
2¢
1¢
10¢
5¢
2¢
1¢

Actual: 18

Greedy Algorithm for Sum of Coins

Solution: Sum of Coins (1)

```java
public static Map<Integer, Integer>
                      chooseCoins(int[] coins, int targetSum) {
  List<Integer> sortedCoins = Arrays.stream(coins).boxed()
            .sorted(Collections.reverseOrder())
            .collect(Collectors.toList());
  Map<Integer, Integer> chosenCoins = new LinkedHashMap<>();
  int currentSum = 0; int coinIndex = 0;
  // Next slide
  if (currentSum != targetSum)
    throw new IllegalArgumentException();
  return chosenCoins;
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Sum of Coins (2)

```
while (currentSum != targetSum && coinIndex < sortedCoins.size()) {
  int currentCoin = sortedCoins.get(coinIndex);
  int remainder = targetSum - currentSum;
  int numberOfCoins = remainder / currentCoin;
  if (currentSum + currentCoin <= targetSum) {
    chosenCoins.put(currentCoin, numberOfCoins);
    currentSum += numberOfCoins * currentCoin;
  }
  coinIndex++;
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

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 }
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Set Cover (1)

```java
public static List<int[]> chooseSets(
              List<int[]> sets, List<Integer> universe) {
  List<int[]> selectedSets = new ArrayList<>();
  Set<Integer> universeSet = new HashSet<>();
  for (int element : universe) { universeSet.add(element);}
  while (!universeSet.isEmpty()) {
    // Next Slide
  }
  return selectedSets;
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Set Cover (2)

```java
int notChosenCount = 0;
int[] chosenSet = sets.get(0);
for (int[] set : sets) {
  // Next slide
}
selectedSets.add(chosenSet);
for (int elem : chosenSet) {
  universeSet.remove(elem);
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Solution: Set Cover (3)

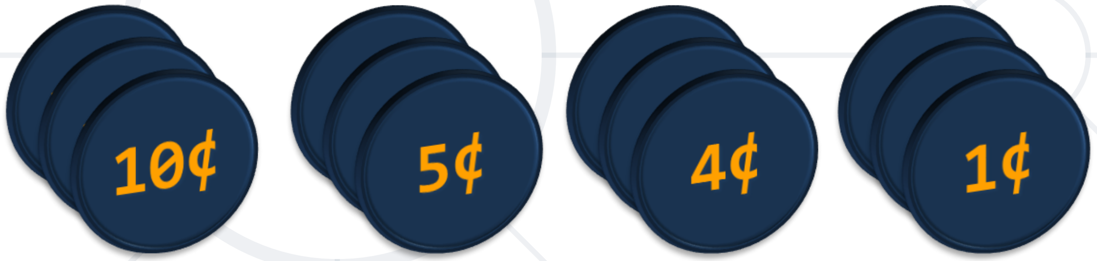
```
int count = 0;
for (int elem : set) {
  if (universeSet.contains(elem)) {
    count++;
  }
}
if (notChosenCount < count) {
  notChosenCount = count;
  chosenSet = set;
}
```

Check your solution here: https://judge.softuni.bg/Contests/1701/Basic-Algorithms-Lab

Greedy Failure Cases

Sum of Coins Failure

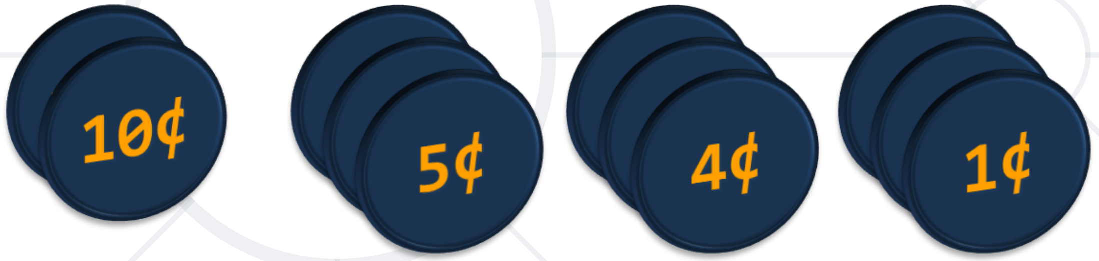
Target: 18



Actual: 0

Sum of Coins Failure

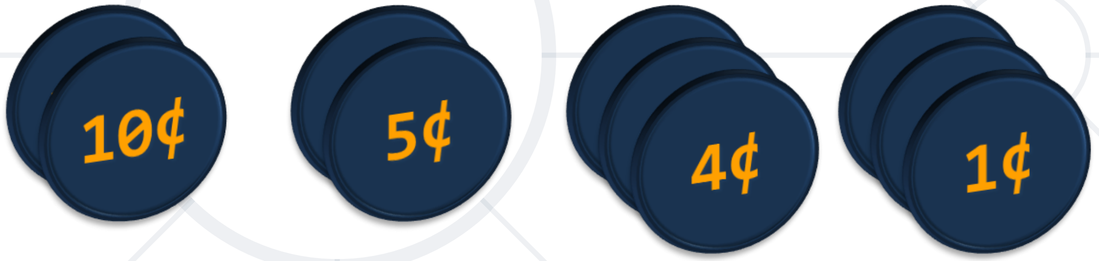
Target: 18



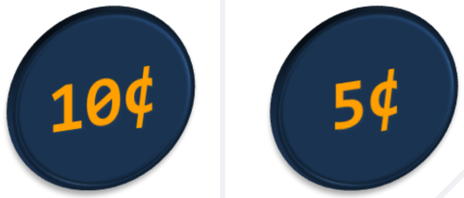
Actual: 10

Sum of Coins Failure

Target: 18

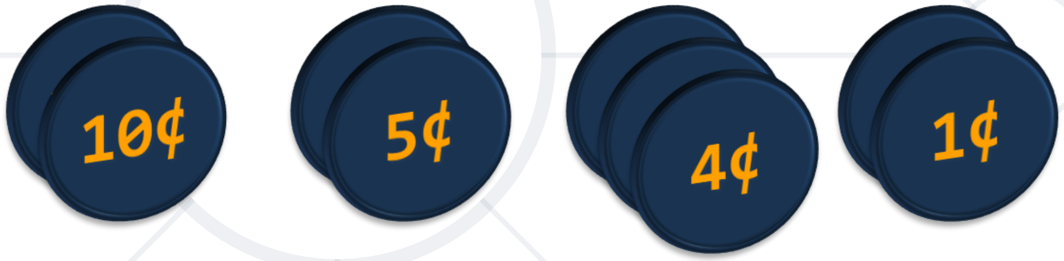


Actual: 15

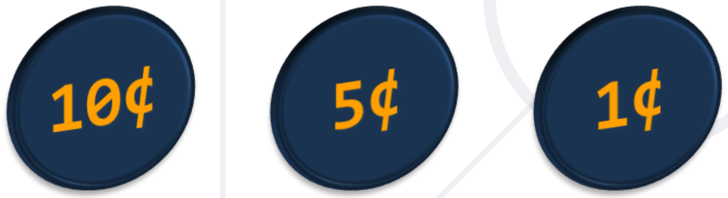


Sum of Coins Failure

Target: 18

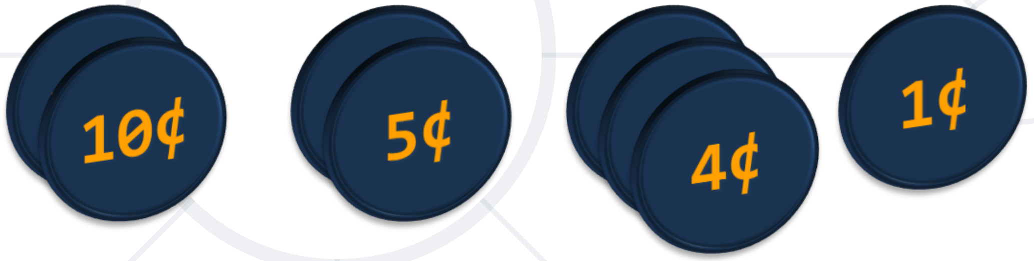


Actual: 16

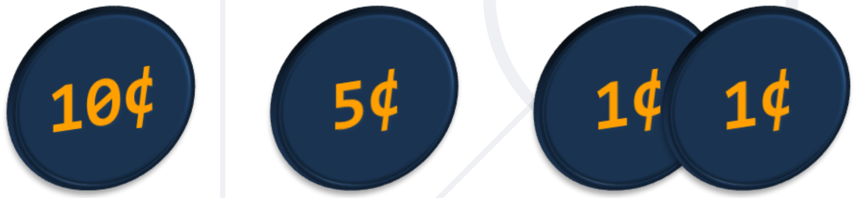


Sum of Coins Failure

Target: 18

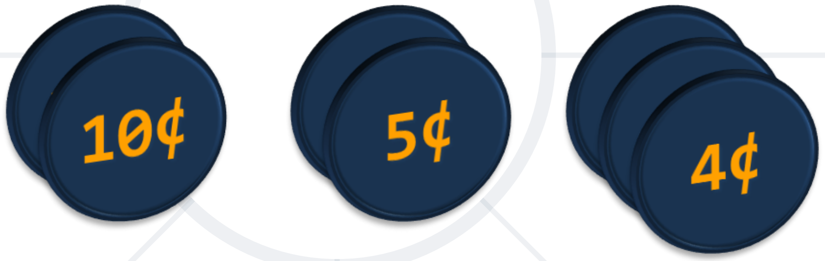


Actual: 17

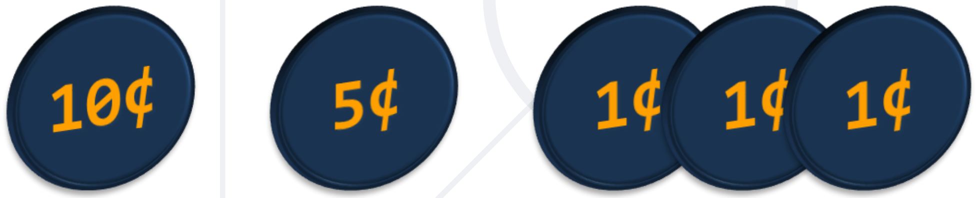


Sum of Coins Failure

Target: 18

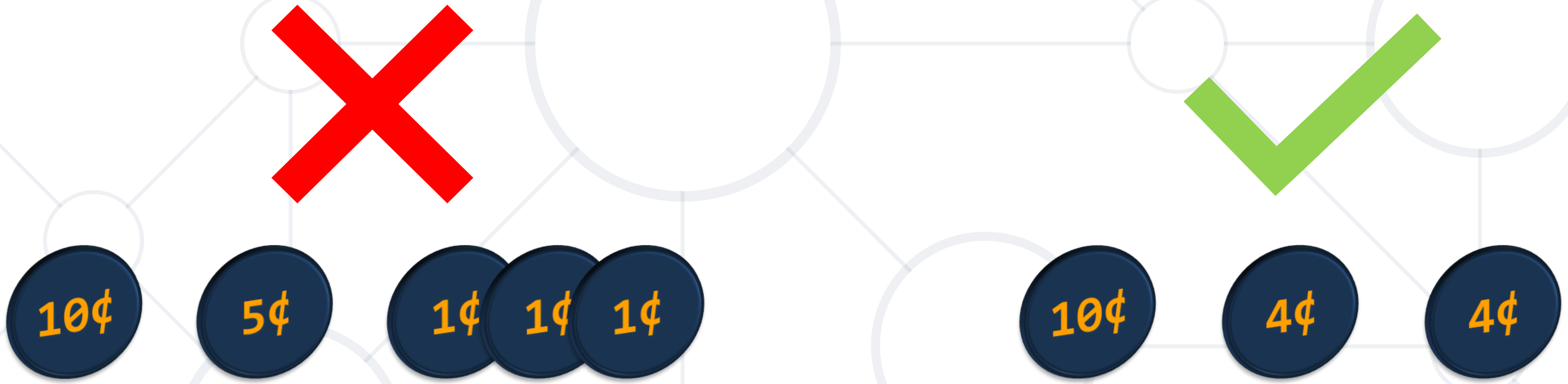


Actual: 18



Sum of Coins Failure

Target: 18



Optimal Greedy Algorithms

Optimal Greedy Algorithms

- Suitable problems for greedy algorithms have these properties:
  - **Greedy choice property**
  - **Optimal substructure**
- Any problem having the above properties is guaranteed to have an optimal greedy solution

Greedy Choice Property

- **Greedy choice** property
  - **A global optimal solution** can be obtained by greedily selecting a **locally optimal** choice
  - Sub-problems that arise are solved by consequent greedy choices
    - Enforced by optimal substructure

Optimal Substructure Property

- **Optimal substructure** property
  - After each greedy choice the problem remains an optimization problem of the same form as the original problem
  - **An optimal global solution contains the optimal solutions of all its sub-problems**

Greedy Algorithms: Example

- The "**Max Coins**" game
  - You are given a set of coins
  - You play against another player, alternating turns
  - Per each turn, you can take up to three coins
  - Your goal is to have as many coins as possible at the end

Max Coins – Greedy Algorithm

- A simple **greedy strategy** exists for the "Max Coins" game

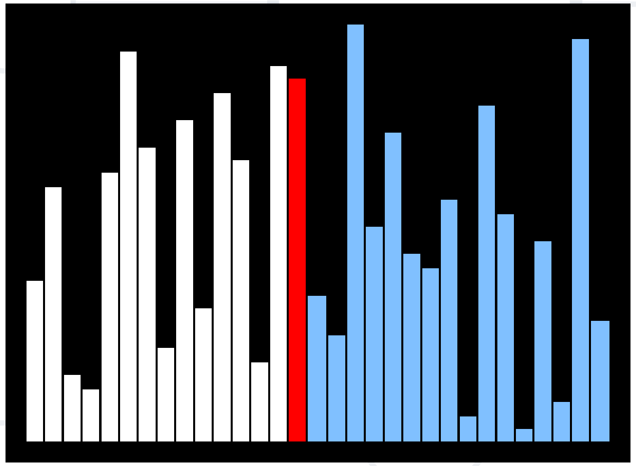
```
At each turn take the maximum number of coins
```

- Always choose the local maximum (at each step)
  - You don't consider what the other player does
  - You don't consider your actions' consequences
- The **greedy algorithm** works optimally here
  - It takes as many coins as possible

Simple Sorting Algorithms

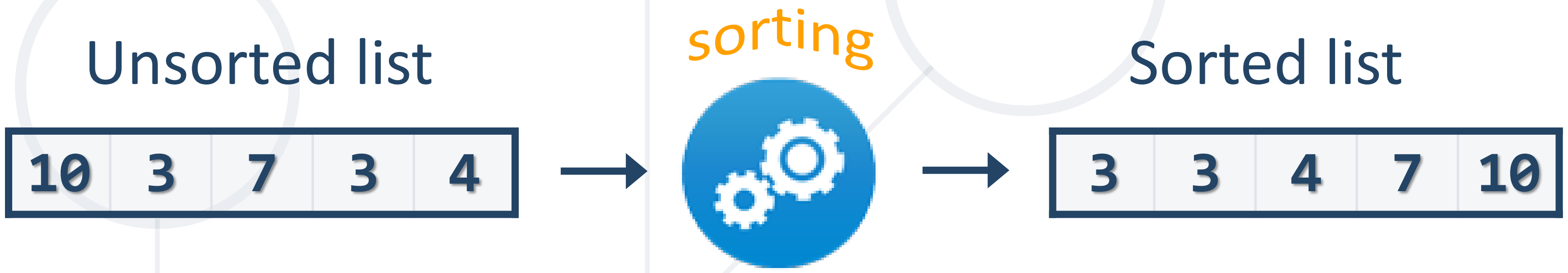
What is a Sorting Algorithm?

- **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 an 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:

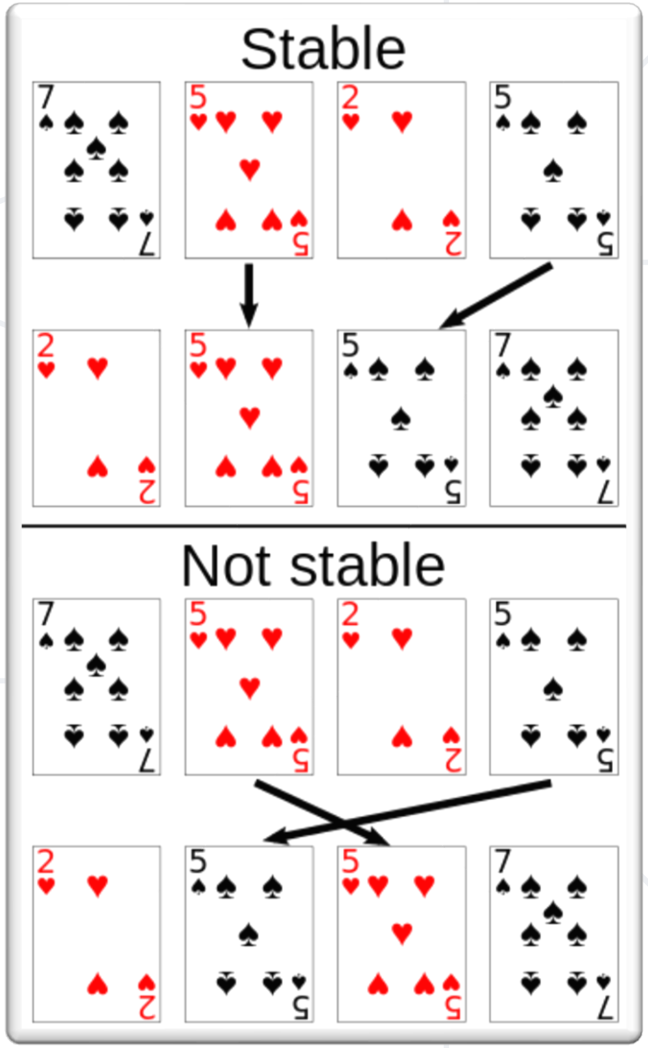


Sorting Algorithms: Classification

- 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
  - Sorting **method**: insertion, exchange (bubble sort and quicksort), selection (heapsort), merging, serial / parallel, etc.

Stability of Sorting

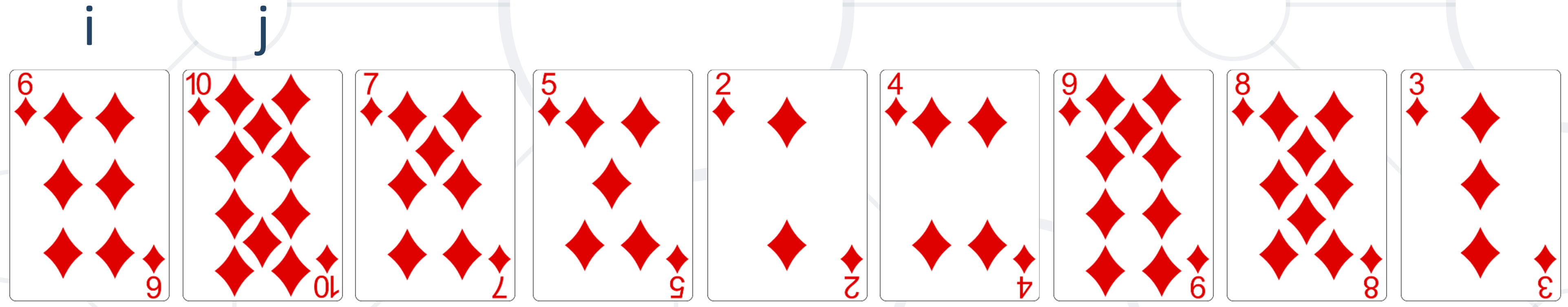
- **Stable** sorting algorithms
  - Maintain the order of equal elements
  - If two items compare as equal, their relative order is preserved
- **Unstable** sorting algorithms
  - Rearrange the equal elements in unpredictable order
- Often **different elements** have **same key** used for equality comparing



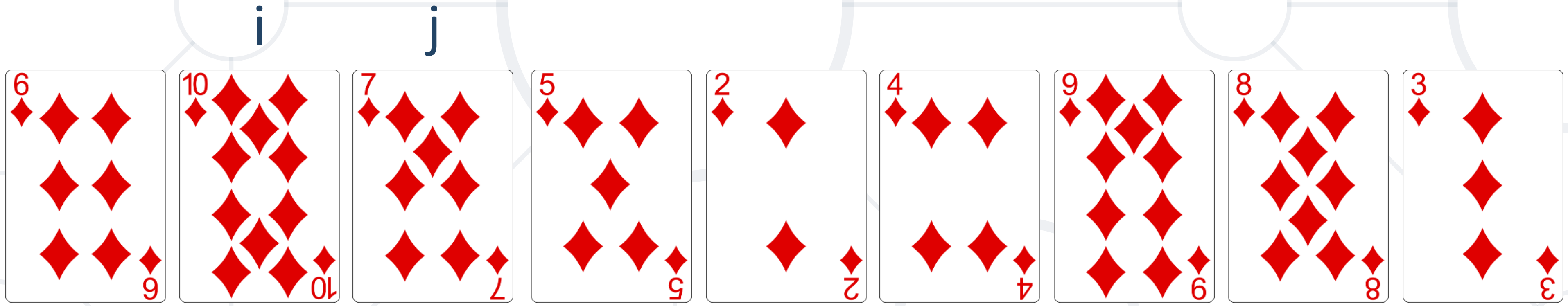
Bubble Sort

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

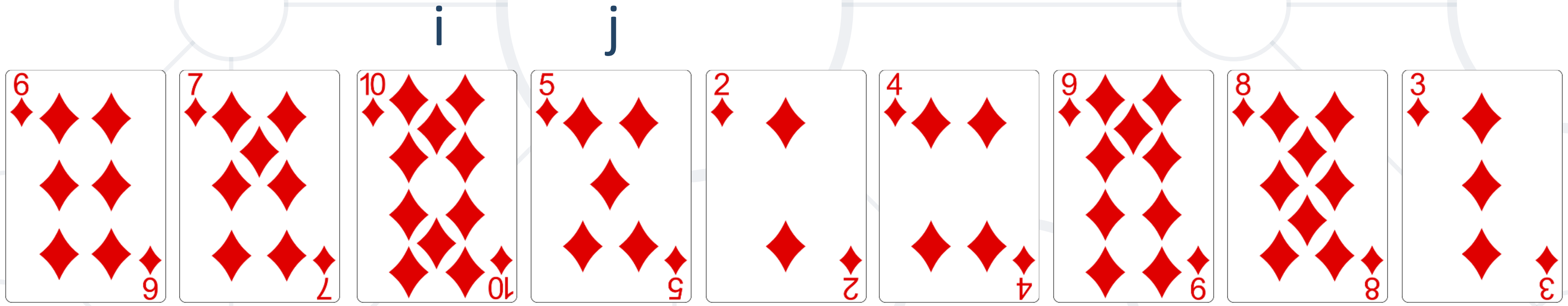
Bubble Sort Visualization



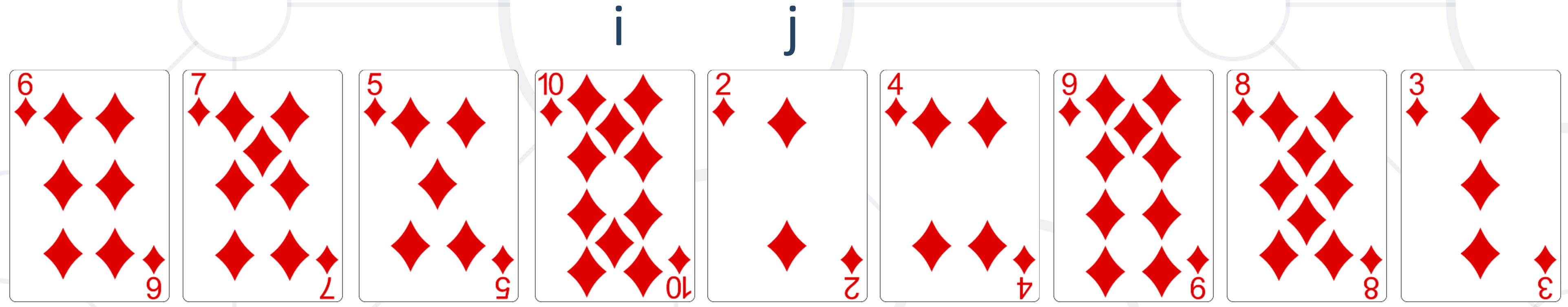
Bubble Sort Visualization



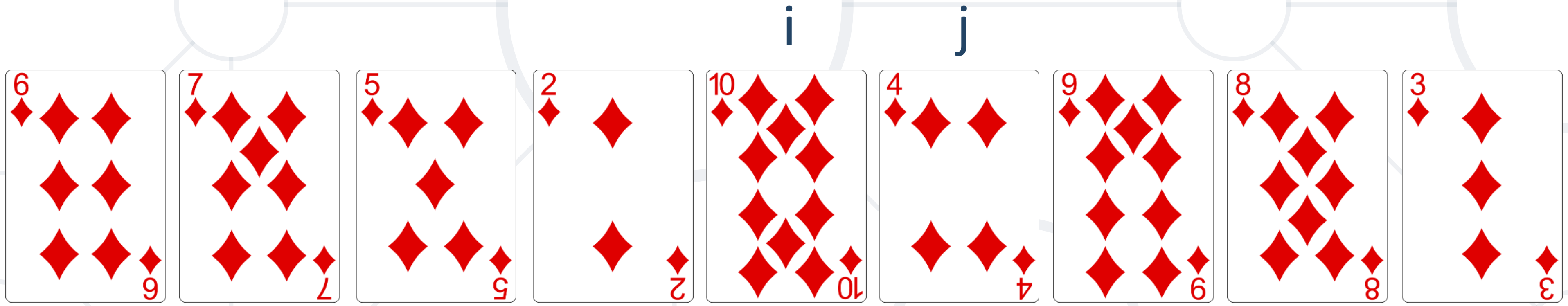
Bubble Sort Visualization



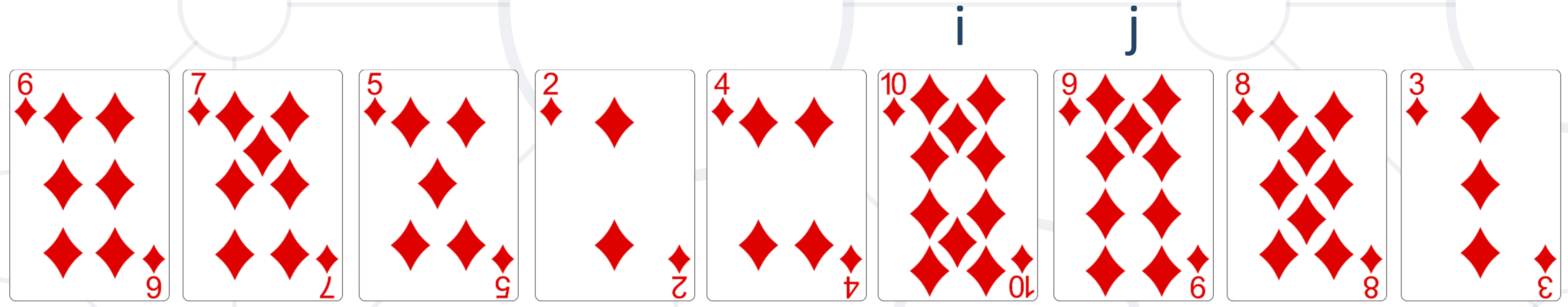
Bubble Sort Visualization



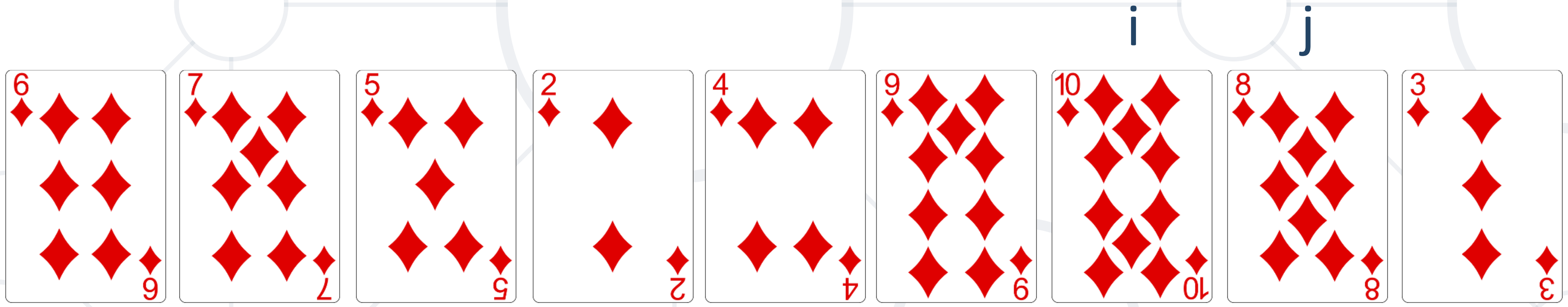
Bubble Sort Visualization



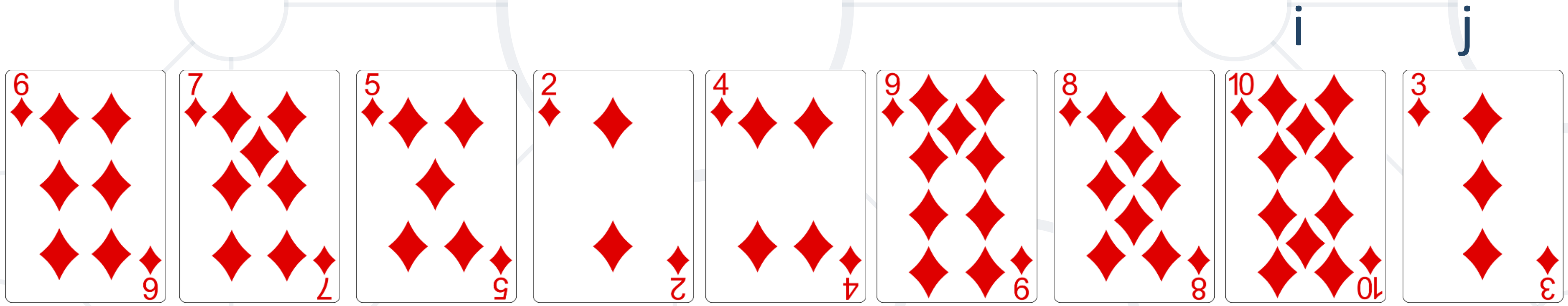
Bubble Sort Visualization



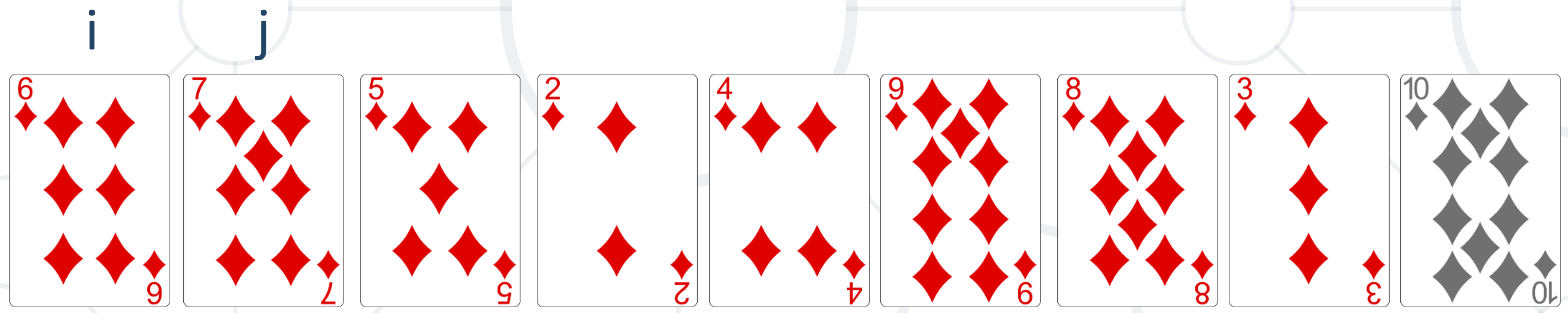
Bubble Sort Visualization



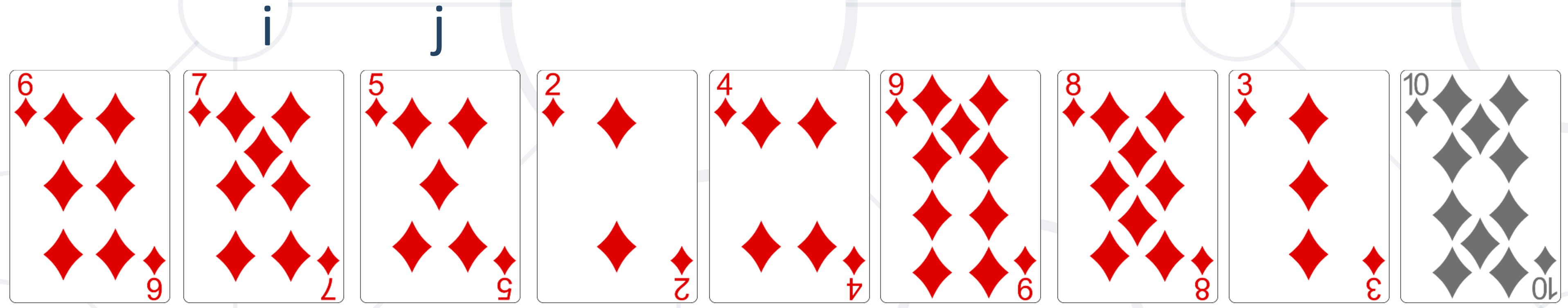
Bubble Sort Visualization



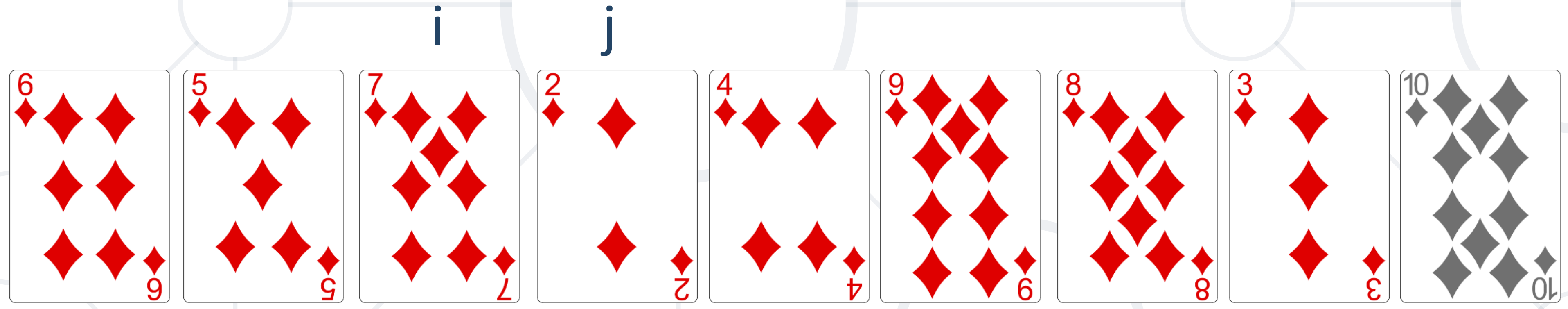
Bubble Sort Visualization



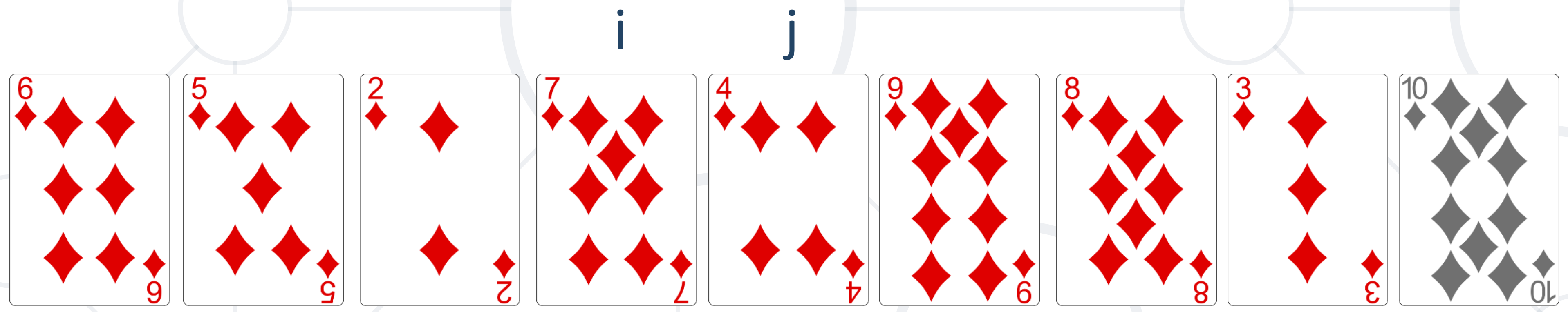
Bubble Sort Visualization



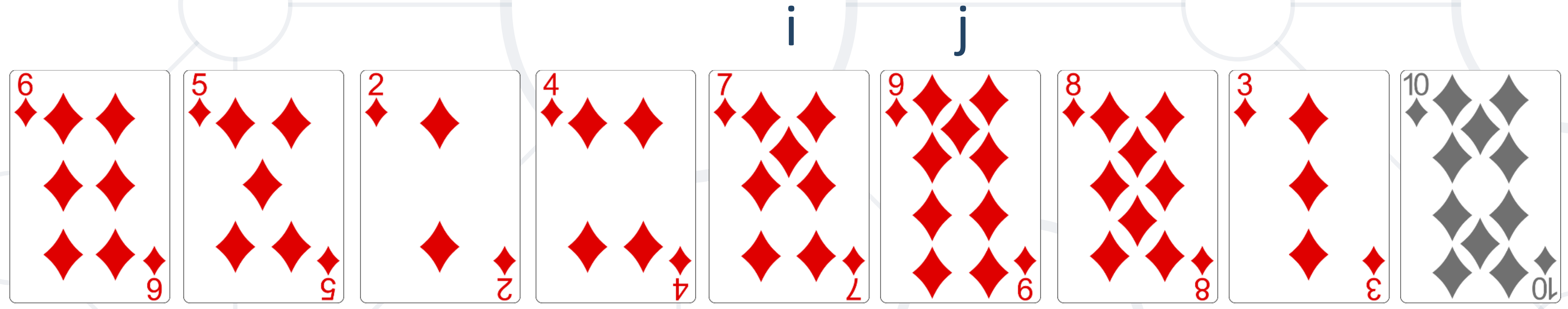
Bubble Sort Visualization



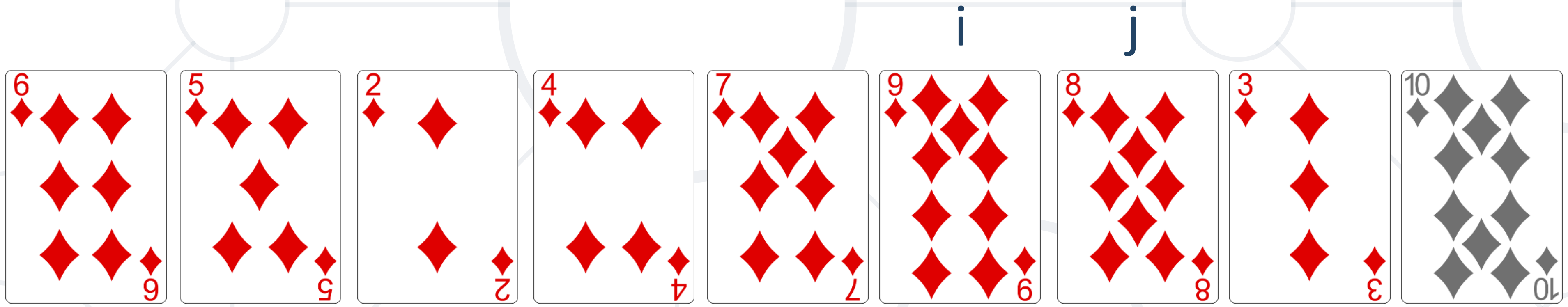
Bubble Sort Visualization



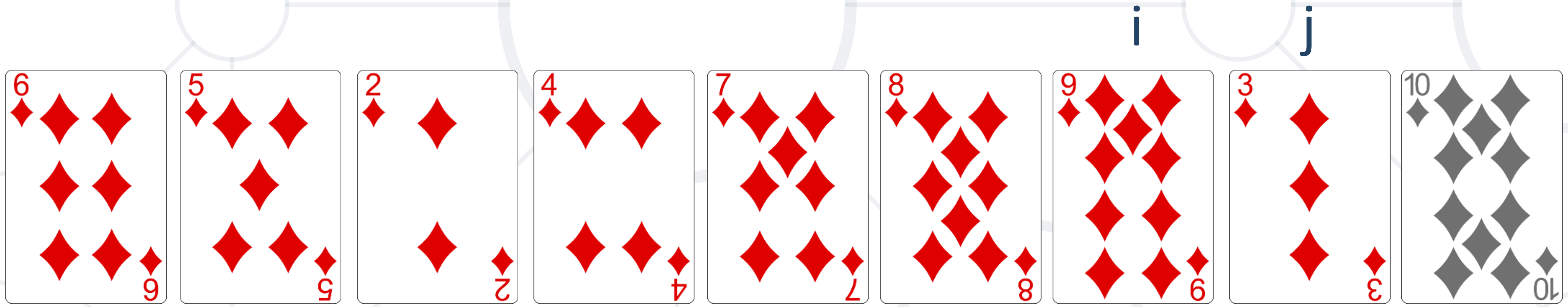
Bubble Sort Visualization



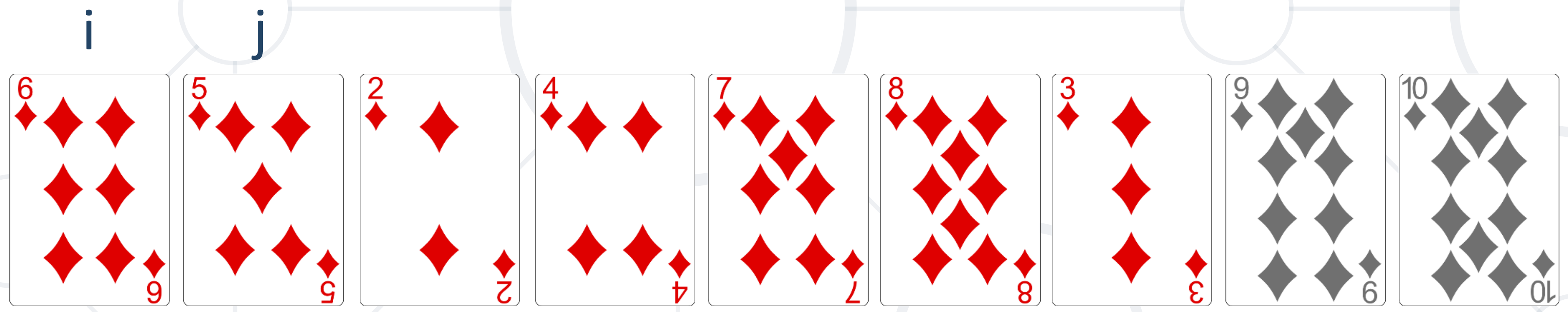
Bubble Sort Visualization



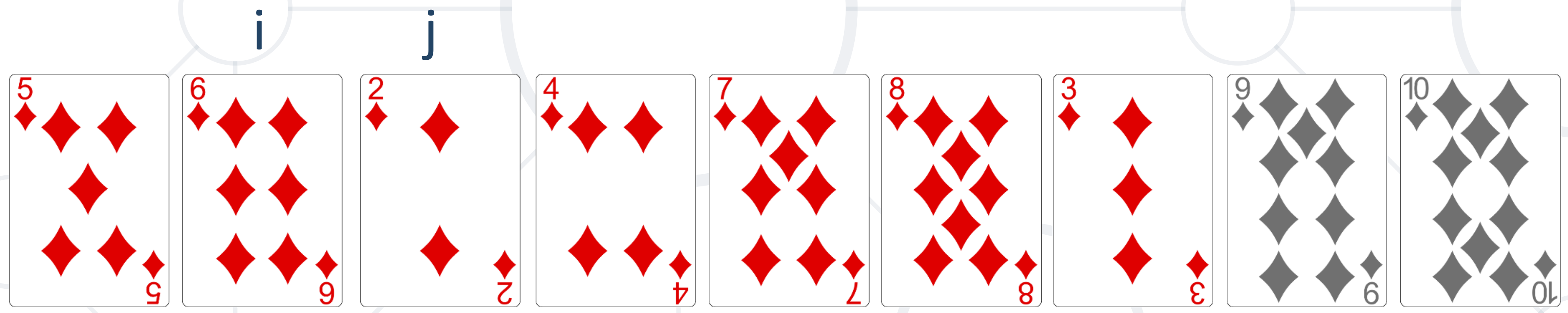
Bubble Sort Visualization



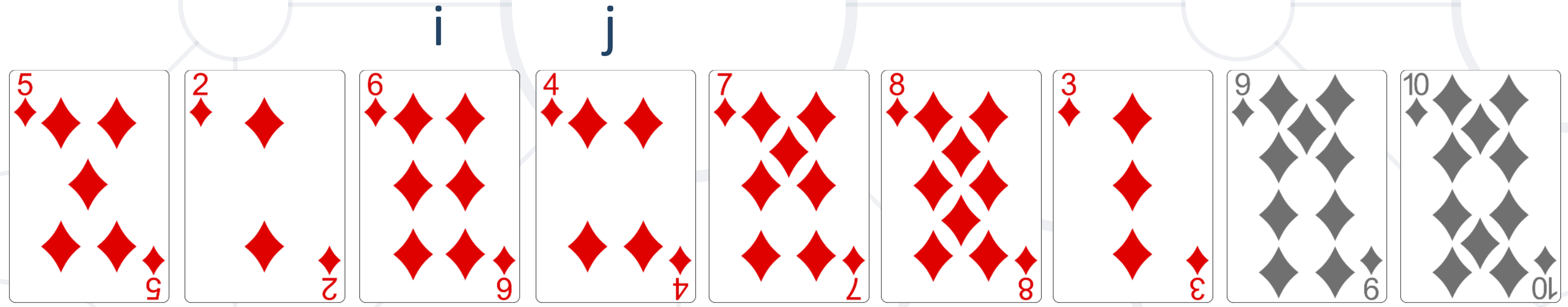
Bubble Sort Visualization



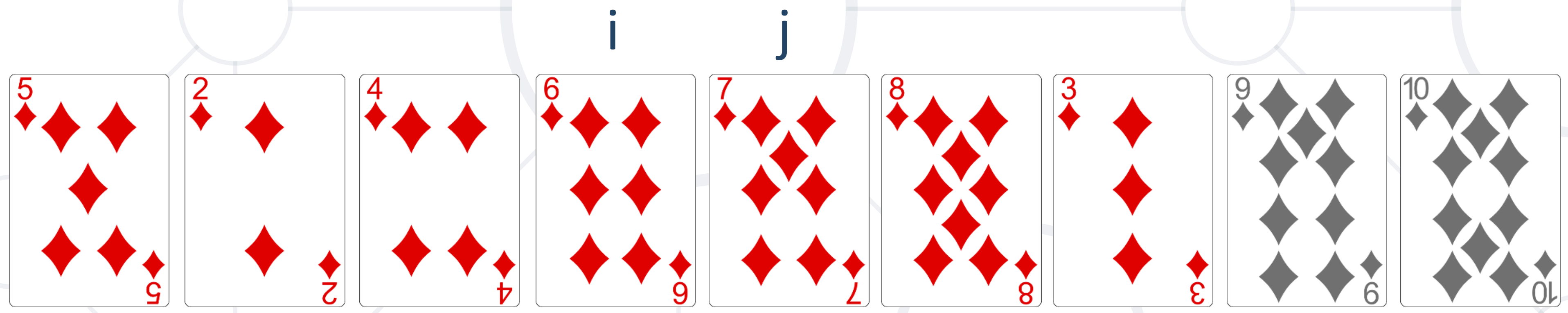
Bubble Sort Visualization



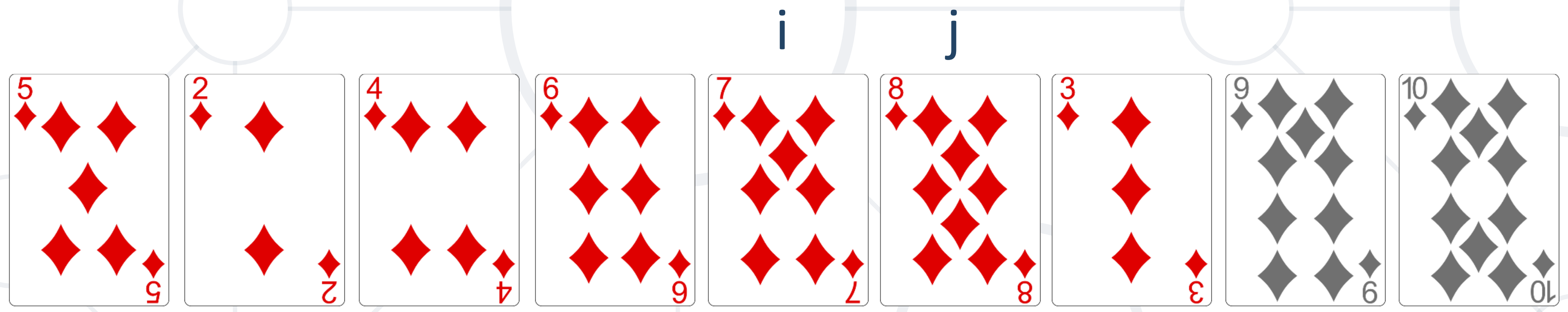
Bubble Sort Visualization



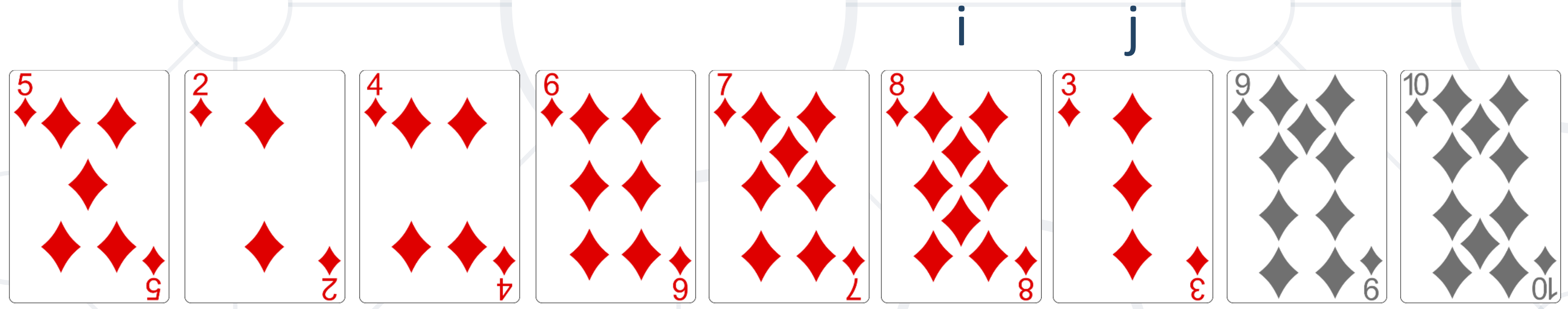
Bubble Sort Visualization



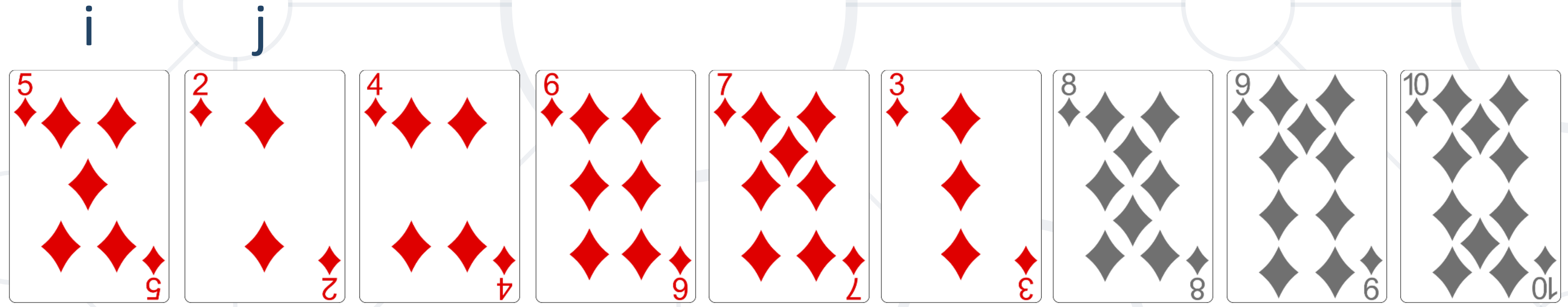
Bubble Sort Visualization



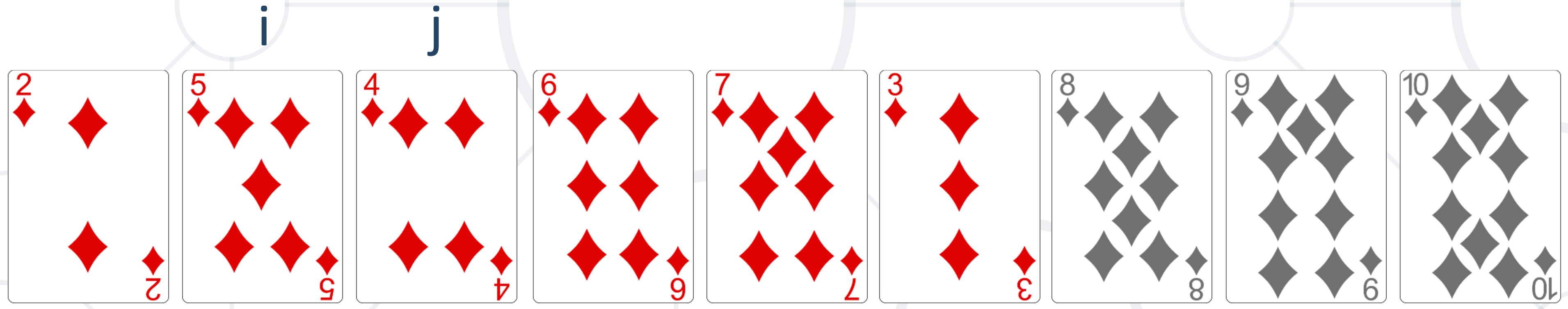
Bubble Sort Visualization



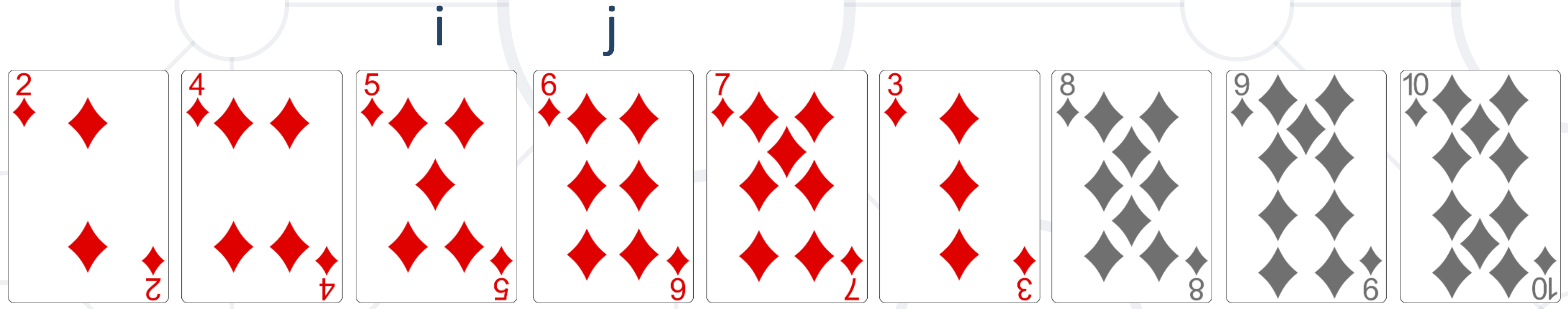
Bubble Sort Visualization



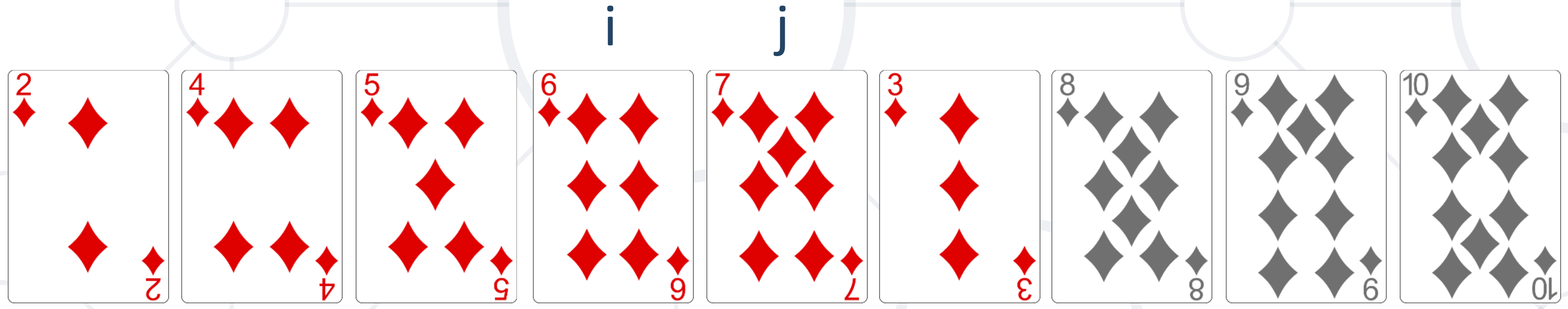
Bubble Sort Visualization



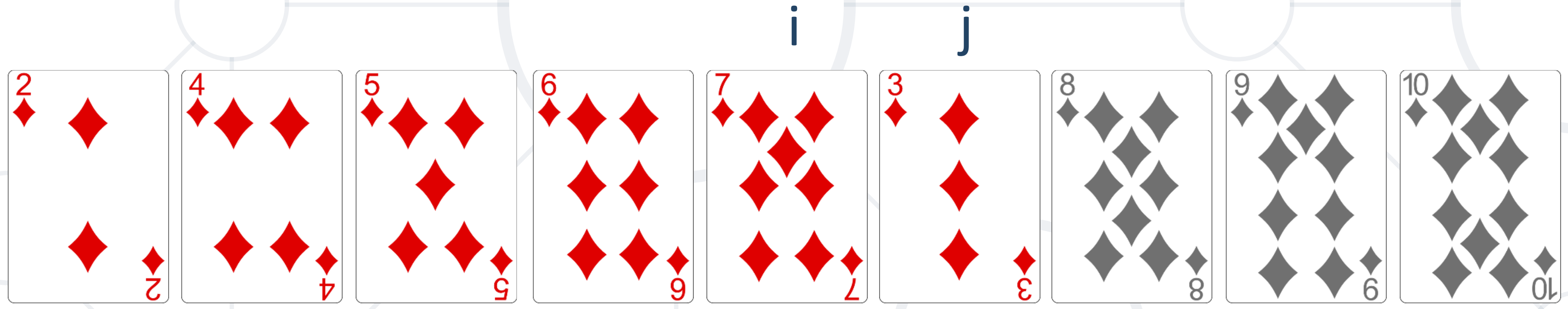
Bubble Sort Visualization



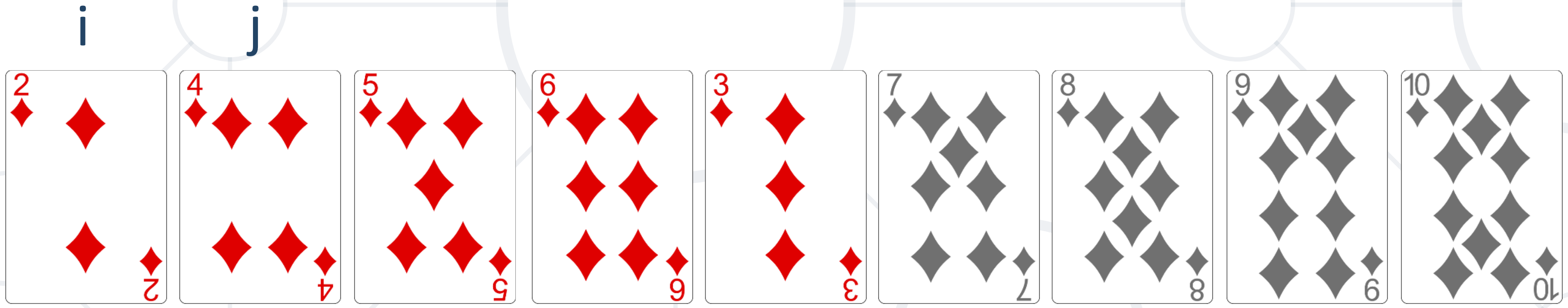
Bubble Sort Visualization



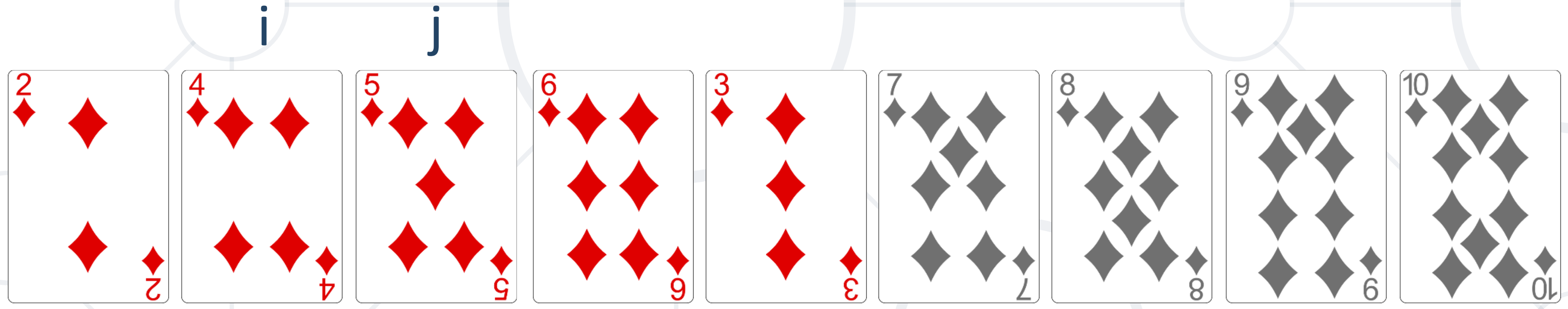
Bubble Sort Visualization



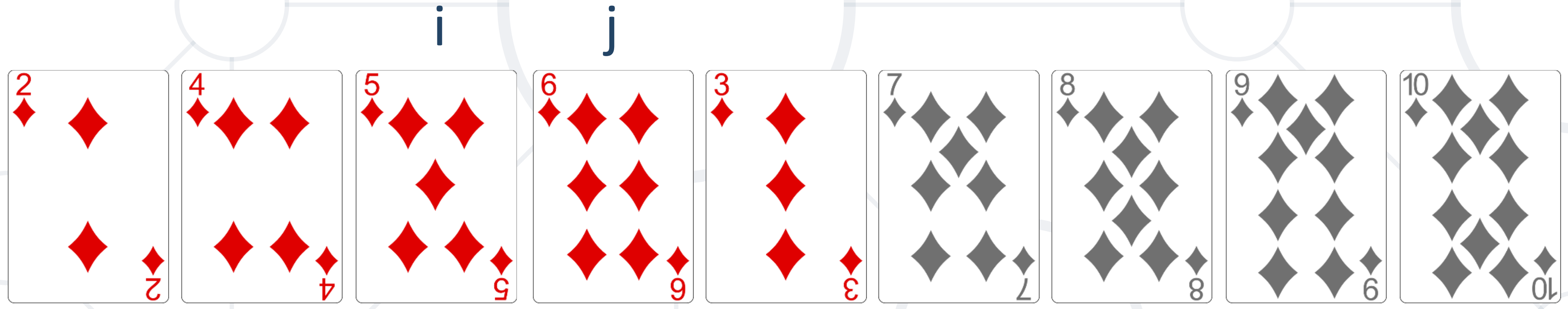
Bubble Sort Visualization



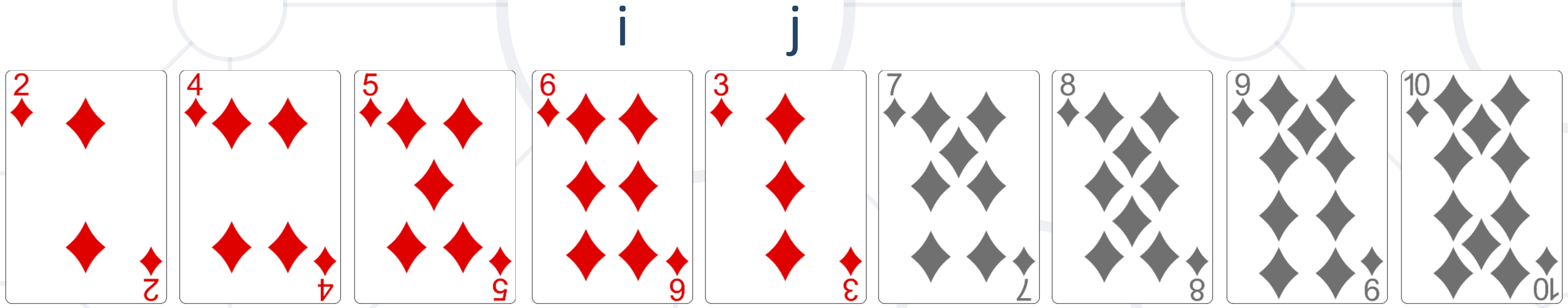
Bubble Sort Visualization



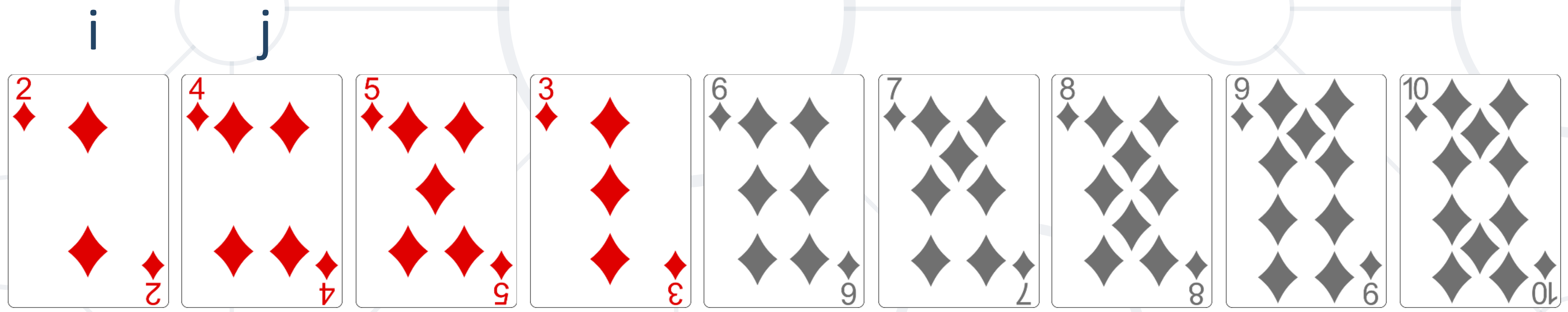
Bubble Sort Visualization



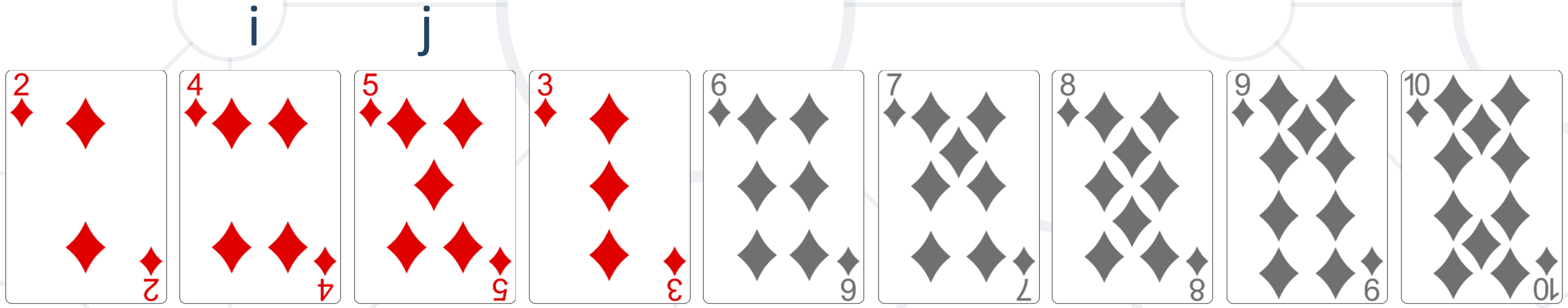
Bubble Sort Visualization



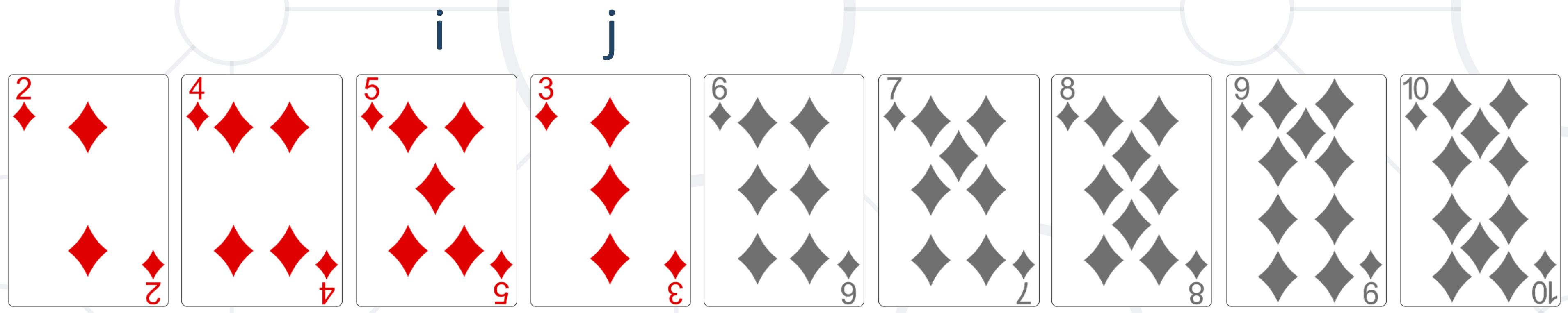
Bubble Sort Visualization



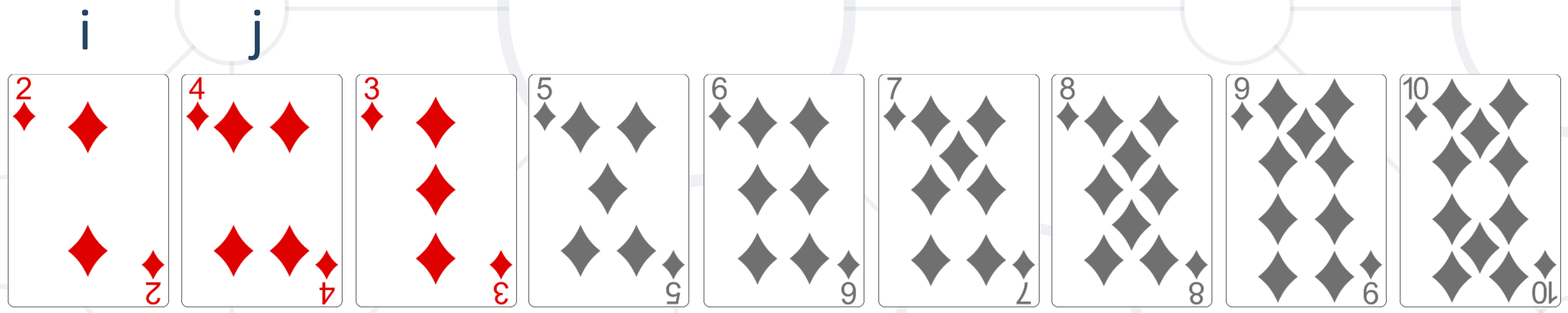
Bubble Sort Visualization



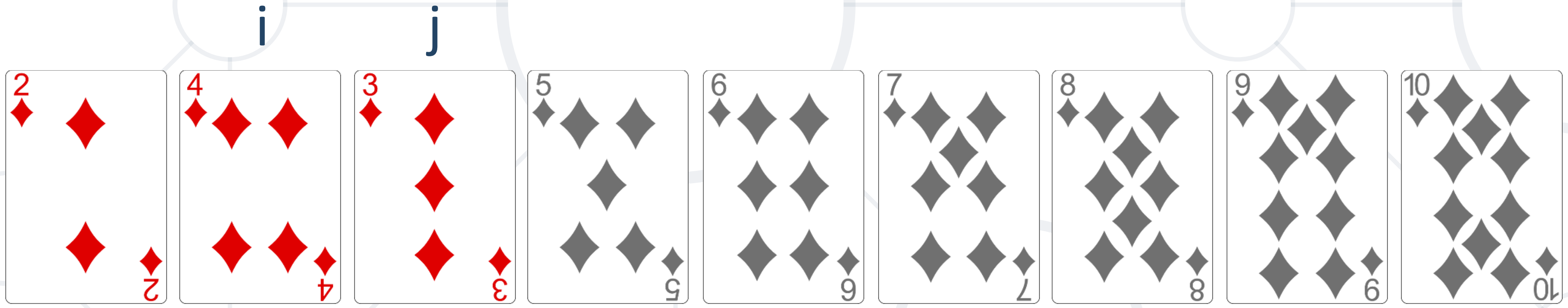
Bubble Sort Visualization



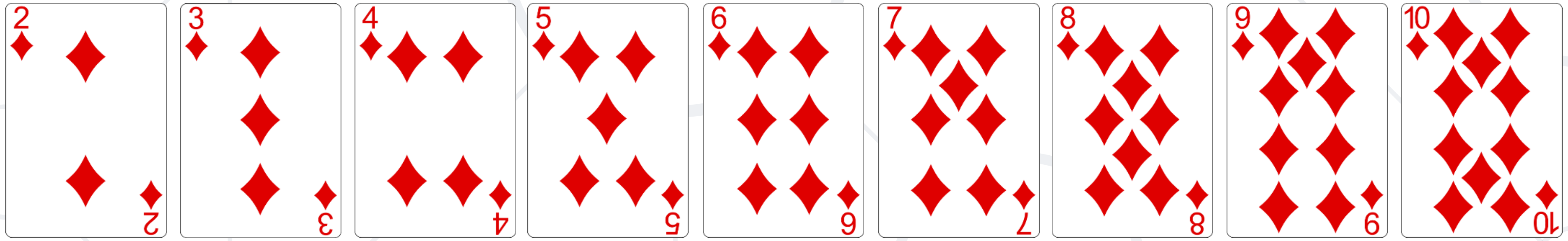
Bubble Sort Visualization



Bubble Sort Visualization



Bubble Sort Visualization



BubbleSort

BubbleSort

```
int[] numbers = {1, 3, 4, 2, 5, 6};
for (int i = 0; i < numbers.length; i++) {
  for (int j = i + 1; j < numbers.length - 1; j++) {
    if (numbers[i] > numbers[j]) {
      int tempNumber = numbers[i];
      numbers[i] = numbers[j];
      numbers[j] = tempNumber;
    }
  }
}

//TODO: Print numbers
```

Searching Algorithms

Search Algorithm

- **Search algorithm** == an algorithm for finding an item with specified properties among a collection of items
- Different types of searching algorithms:
  - For virtual search spaces
    - Satisfy specific mathematical equations
    - Try to exploit partial knowledge about a structure
  - For sub-structures of a given structure
    - A graph, a string, a finite group
  - Search for the min / max of a function, etc.

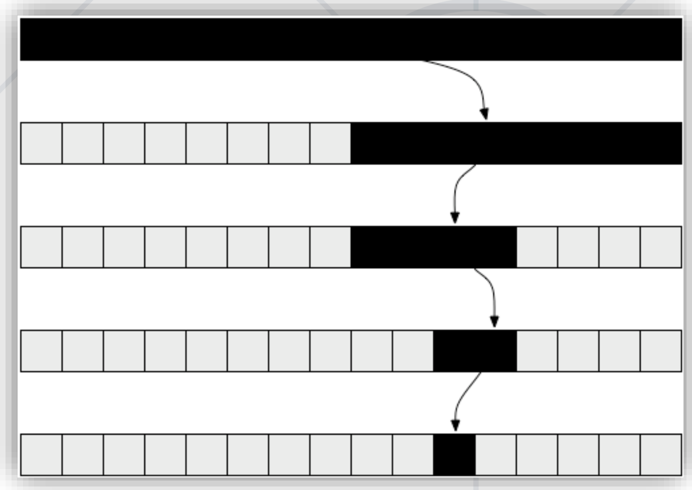
Linear Search

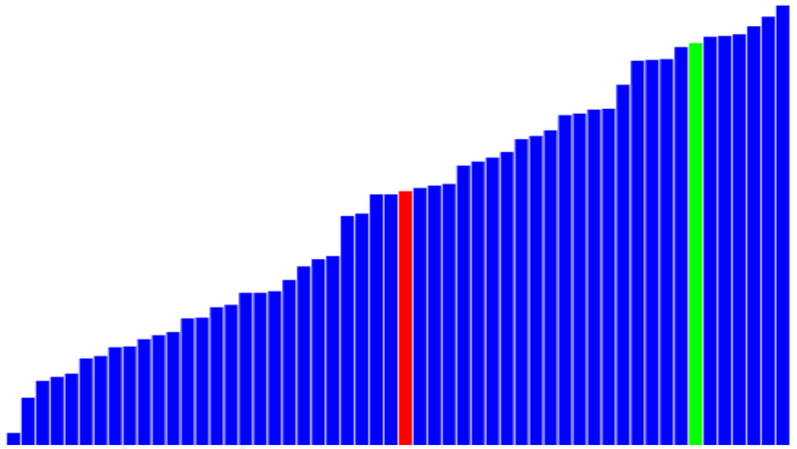
- Linear search finds a particular value in a list (visualize)
  - Checking every one of the elements
  - One at a time, in sequence
  - Until the desired one is found
- Worst & average performance: O(n)

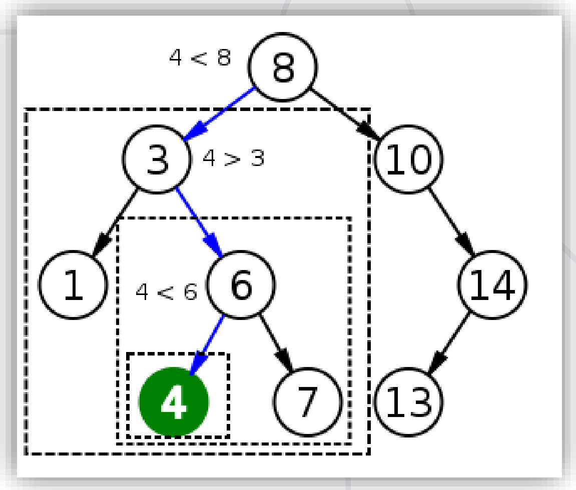
```
for each item in the list:
  if that item has the desired value,
    return the item's location
return nothing
```

Binary Search

- Binary search finds an item within a 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)

```
int binarySearch(int arr[], int key, int start, int end) {
  while (end >= start) {
    int mid = (start + end) / 2;
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else
      return mid;
  }
  return KEY_NOT_FOUND;
}
```

Summary

- **Recursion** – a method or a function that calls itself
- **Brute-Force** - trying all the possible solutions
- **Greedy** - picking a locally optimal solution
- **Sorting**
  - Bubble Sort
- **Searching**
  - Linear and Binary

Questions?

SoftUni
Software University
SoftUni Svetlina
SoftUni Creative
SoftUni Digital
SoftUni Foundation
SoftUni Kids

SoftUni Diamond Partners

SUPER HOSTING .BG

INDEAVR Serving the high achievers

SmartIT

XS SOFTWARE

zühlke empowering ideas

INFRAGISTICS

Coca-Cola HBC Bulgaria

DRAFT KINGS

Postbank Решения за твоето утре

SOFTWARE GROUP

Educational Partners

Software University

eee.bg

VIRTUAL RACING SCHOOL

codexio

Trainings @ Software University (SoftUni)

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

License