Searching, Sorting and Greedy Algorithms

Searching, Sorting and Greedy Algorithms

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

1. Searching Algorithms
   - Linear Search
   - Binary Search
2. Simple Sorting Algorithms
   - Selection, Bubble Sort and Insertion
3. Advanced Sorting Algorithms
   - QuickSort, MergeSort
4. Greedy Algorithms

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 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.

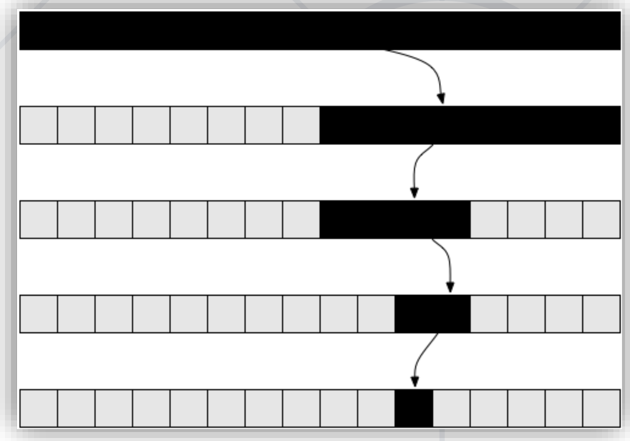
Linear Search

- **Linear search** finds a particular value in a list
  - 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



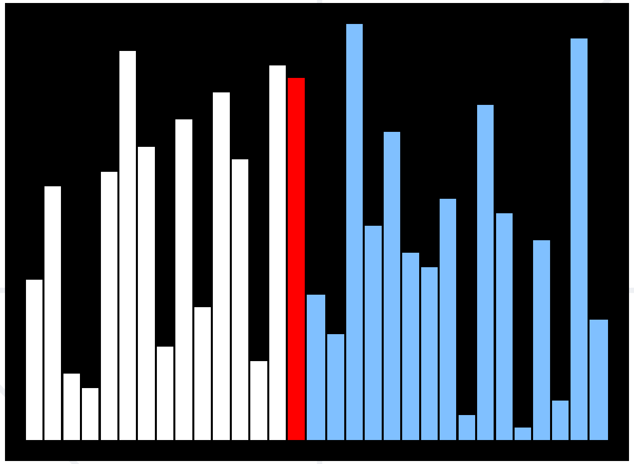
Binary Search (Iterative)

```
static int BinarySearch(int[] numbers, int searchNumber) {
  var left = 0;
  var right = numbers.Length - 1;
  while (left <= right) {
    var mid = (left + right) / 2;
    if (numbers[mid] == searchNumber)
      return mid;
    if (searchNumber > numbers[mid])
      left = mid + 1;
    else
      right = mid - 1;
  }
  return -1;
}
```

Simple Sorting Algorithms

Selection Sort and Bubble 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 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:

Unsorted list

10	3	7	3	4

→ sorting →

Sorted list

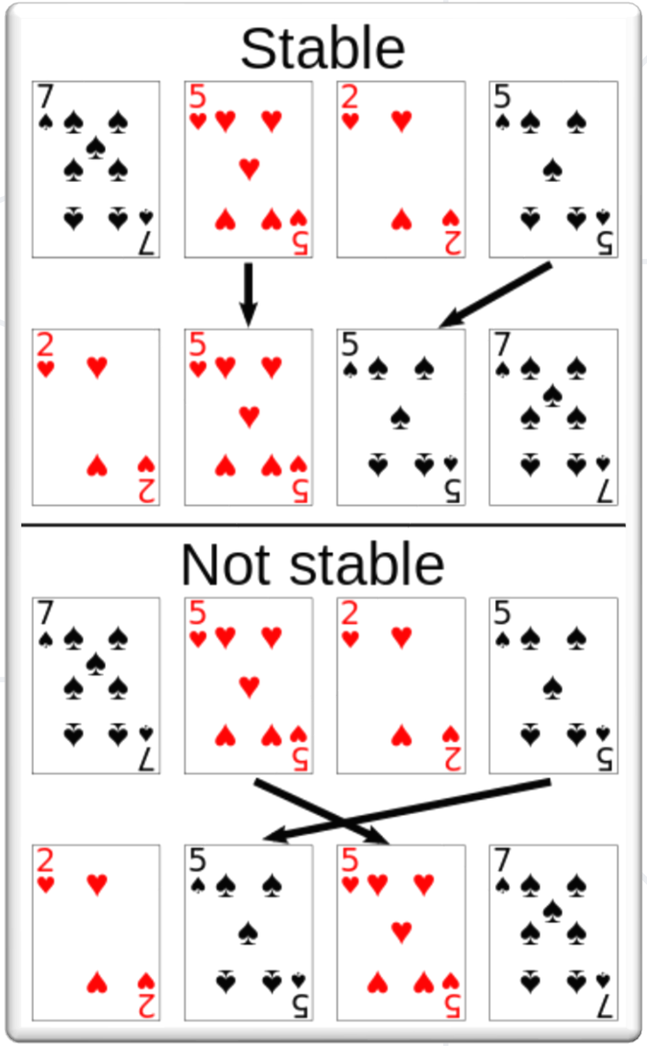
3	3	4	7	10

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 **same key** used for equality comparing

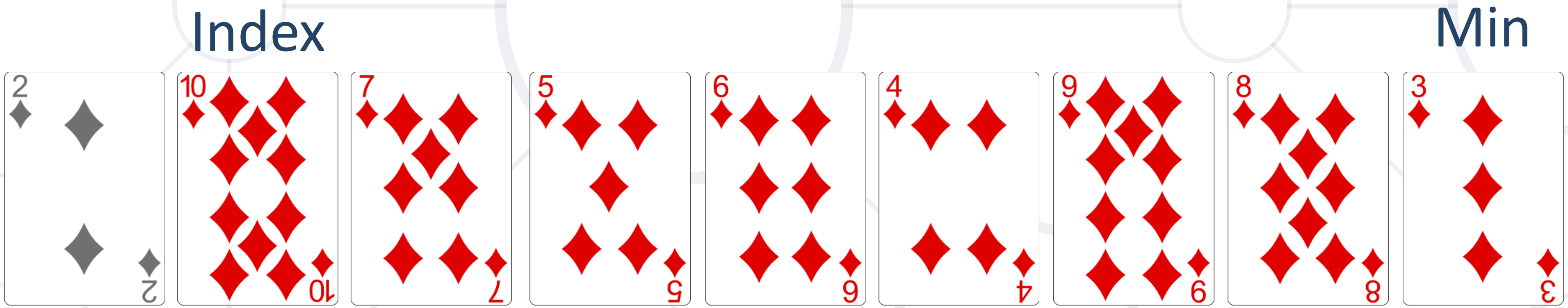


Selection Sort

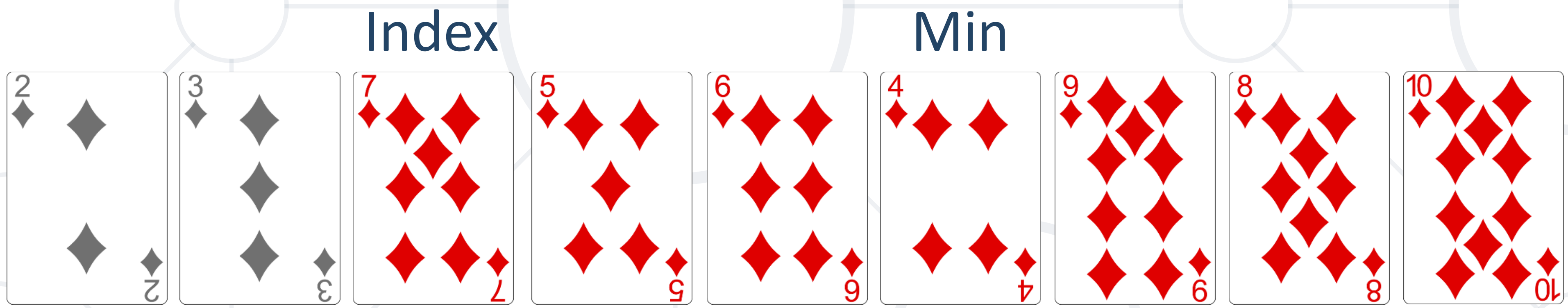
- **Selection sort** – simple, but inefficient algorithm
  - Swap the first with the min element on the right, then the second, etc.
  - Memory: **O(1)**
  - Time: **$O(n^2)$**
  - Stable: No
  - Method: Selection

Selection Sort Visualization

Selection Sort Visualization

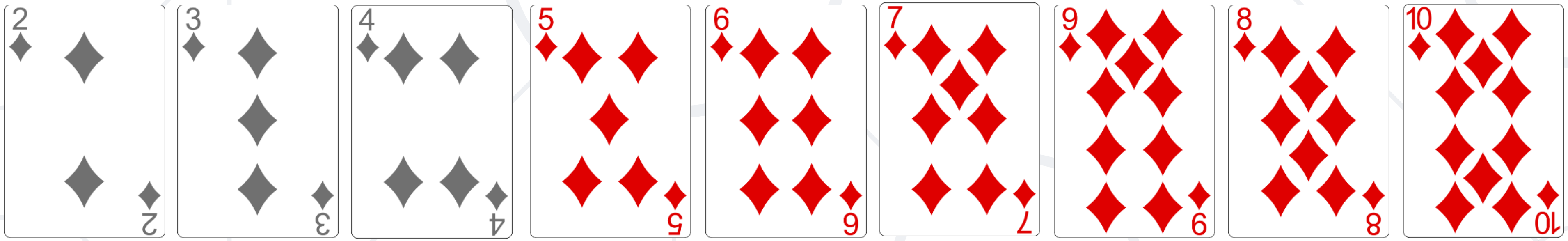


Selection Sort Visualization



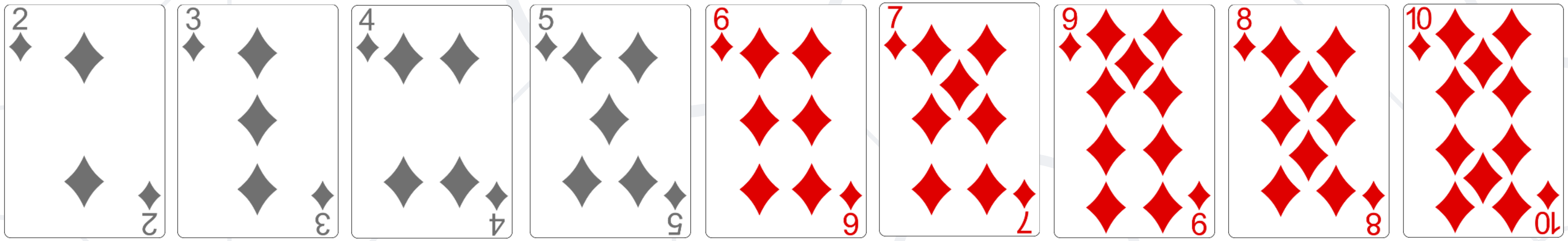
Selection Sort Visualization

Min
Index



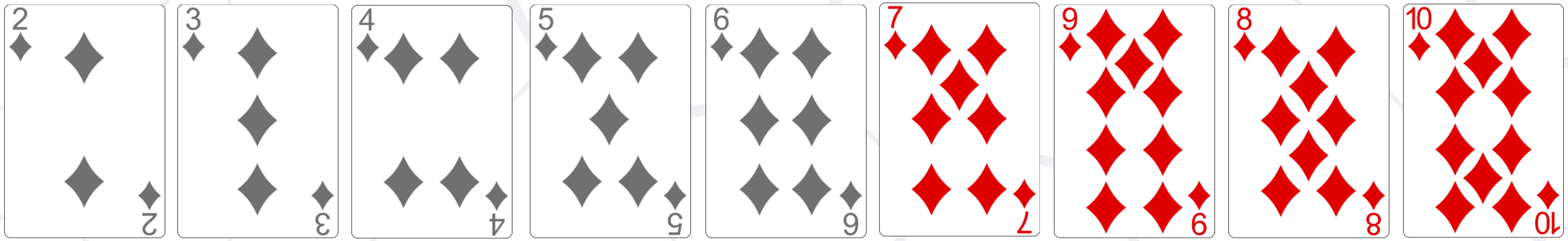
Selection Sort Visualization

Min
Index

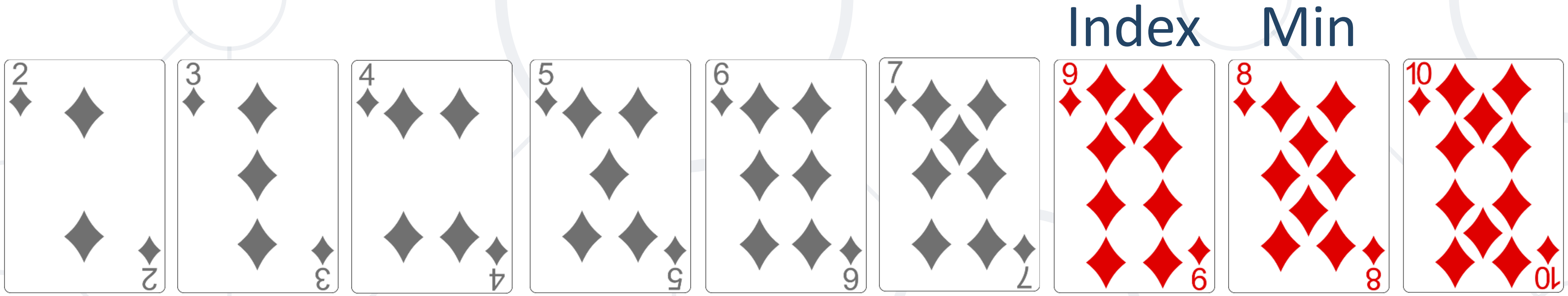


Selection Sort Visualization

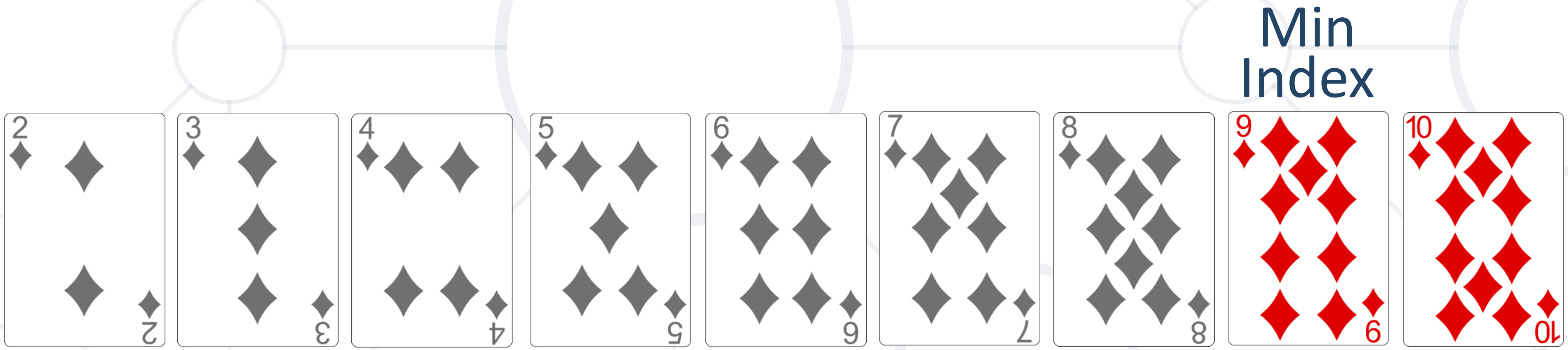
Min Index



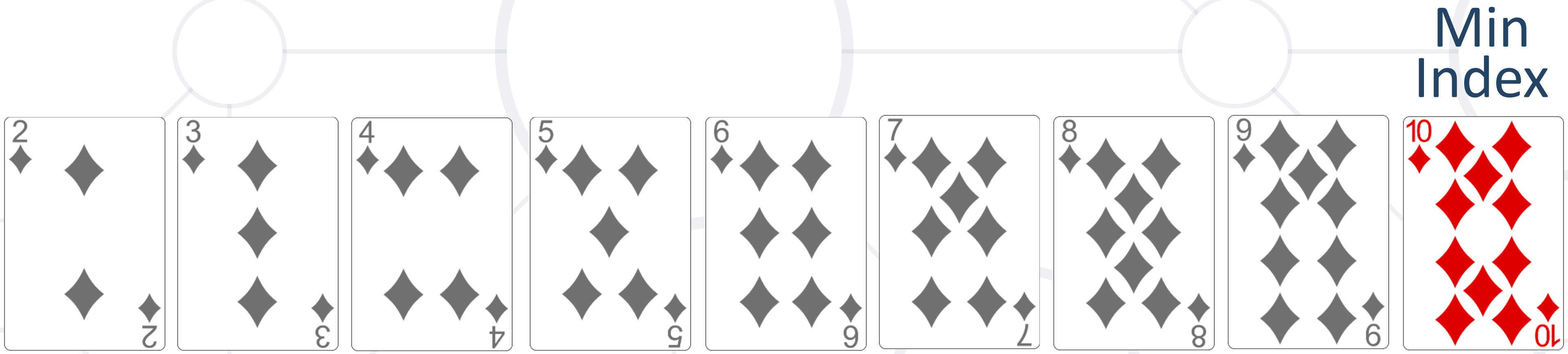
Selection Sort Visualization



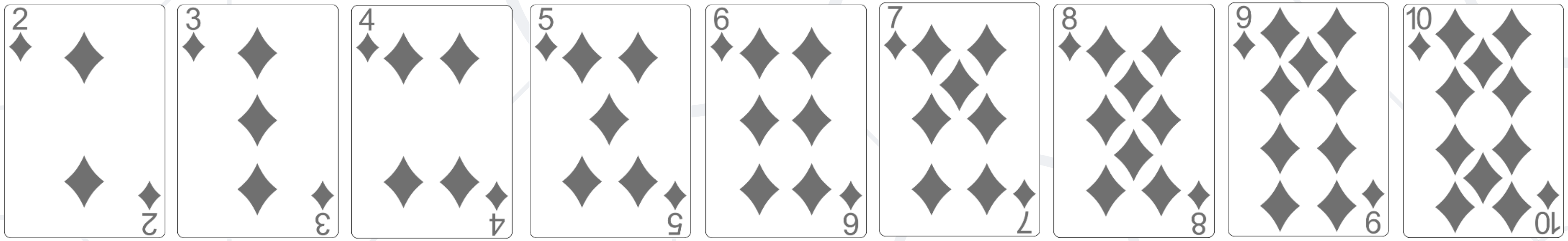
Selection Sort Visualization



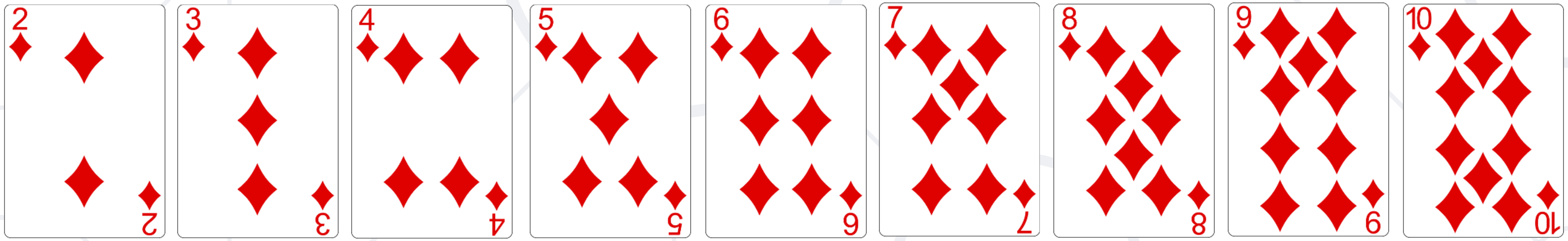
Selection Sort Visualization



Selection Sort Visualization



Selection Sort Visualization



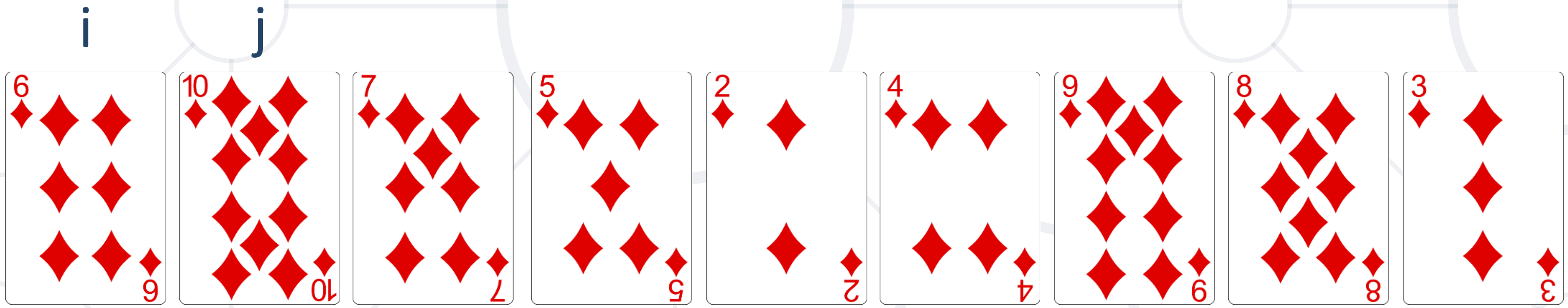
Selection Sort Code

```
for (int index = 0; index < arr.Length; index++) {
    var min = index;
    for (int curr = index + 1; curr < arr.Length; curr++) {
        if (arr[curr] < arr[min]) {
            min = curr;
        }
    }
    Swap(arr, index, min);
}
```

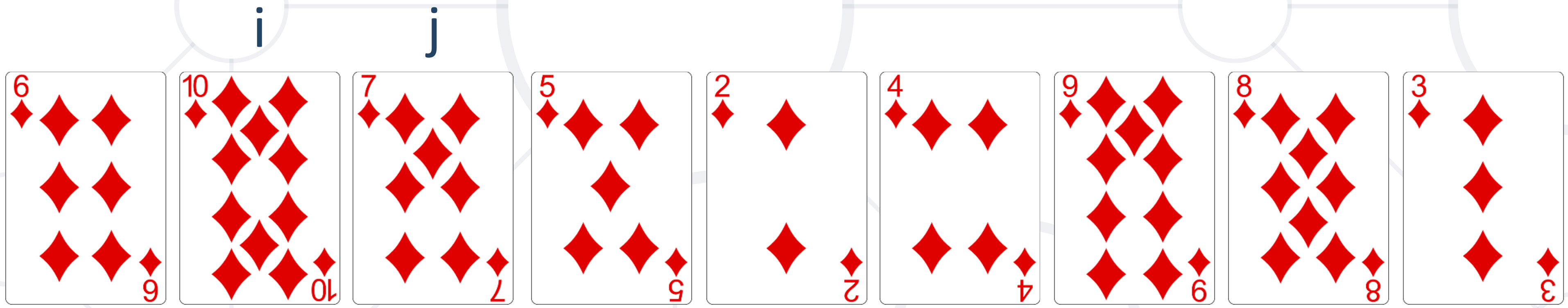
Bubble Sort

- **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

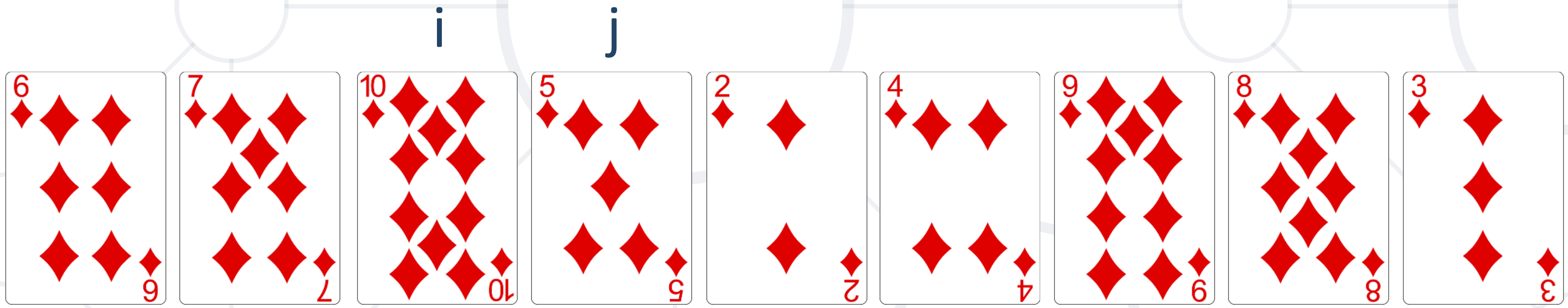
Bubble Sort Visualization



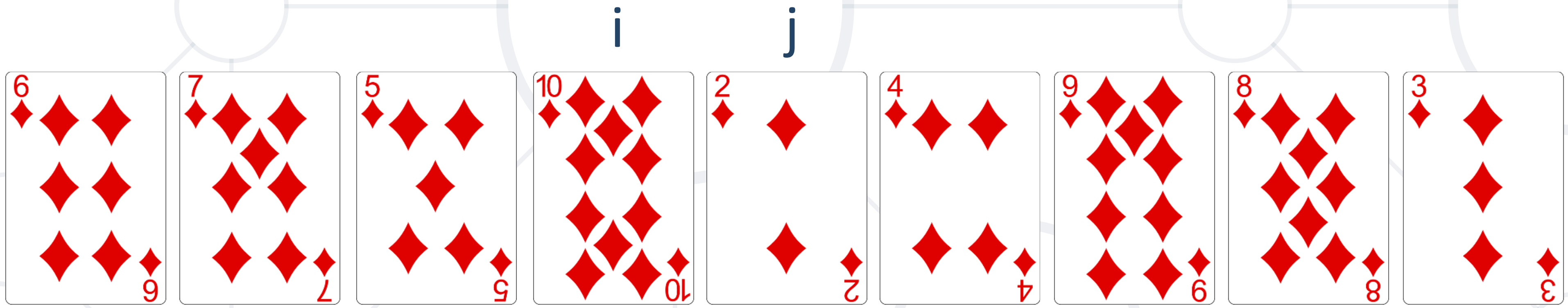
Bubble Sort Visualization



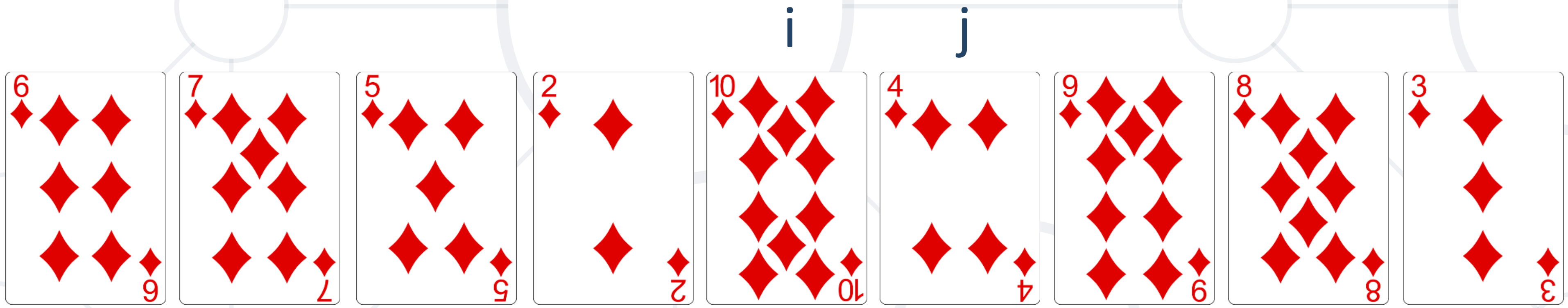
Bubble Sort Visualization



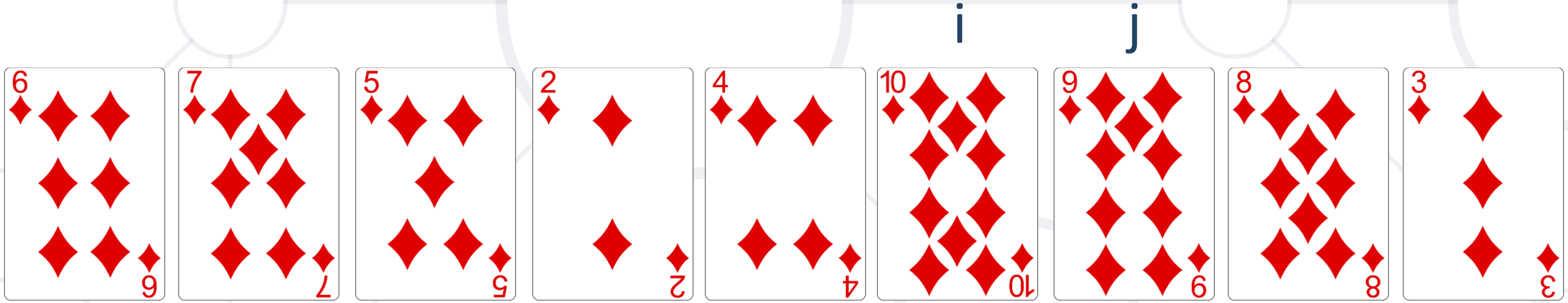
Bubble Sort Visualization



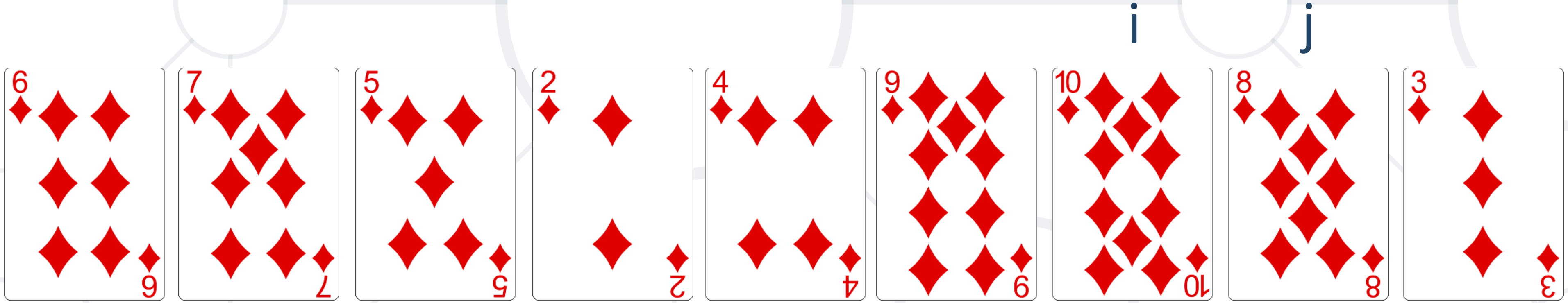
Bubble Sort Visualization



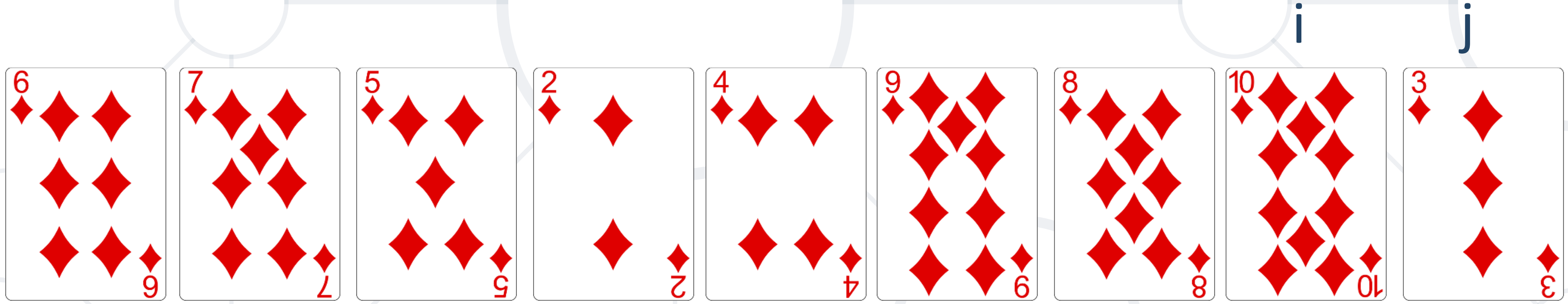
Bubble Sort Visualization



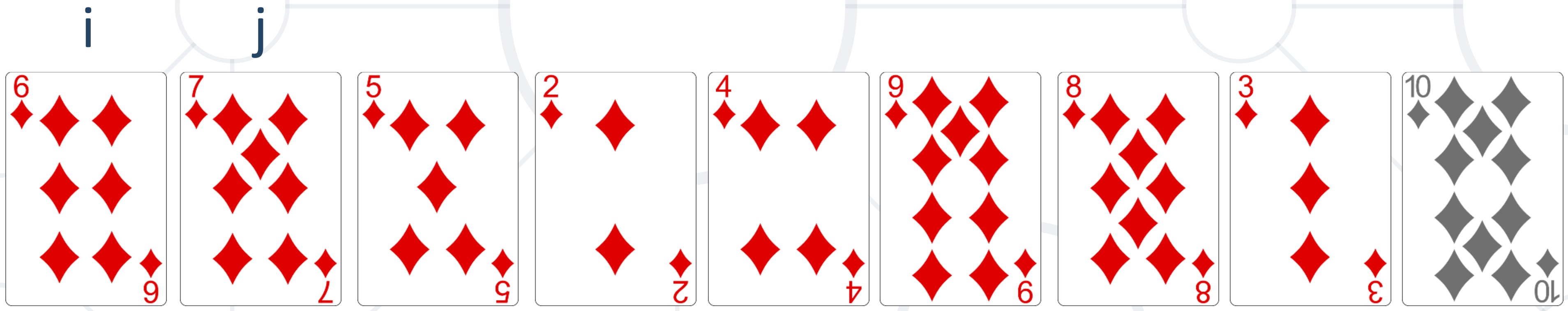
Bubble Sort Visualization



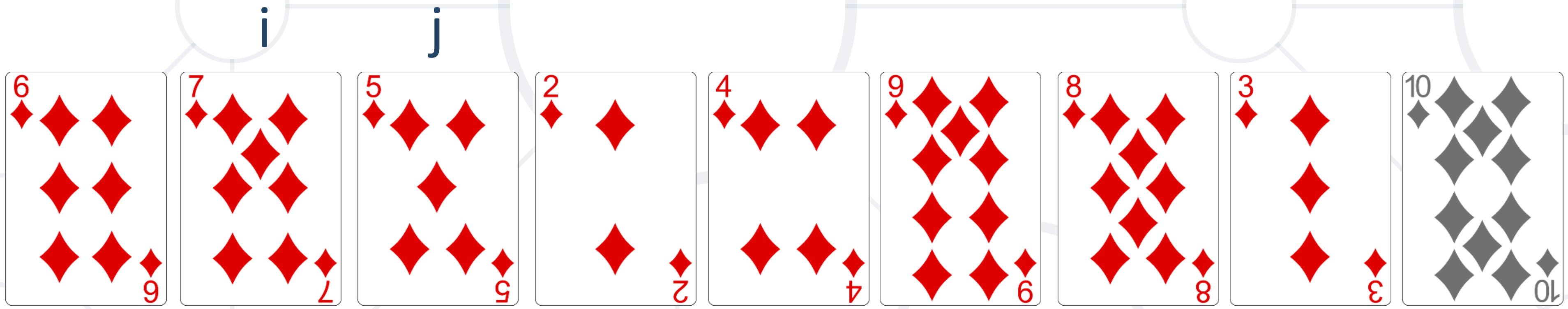
Bubble Sort Visualization



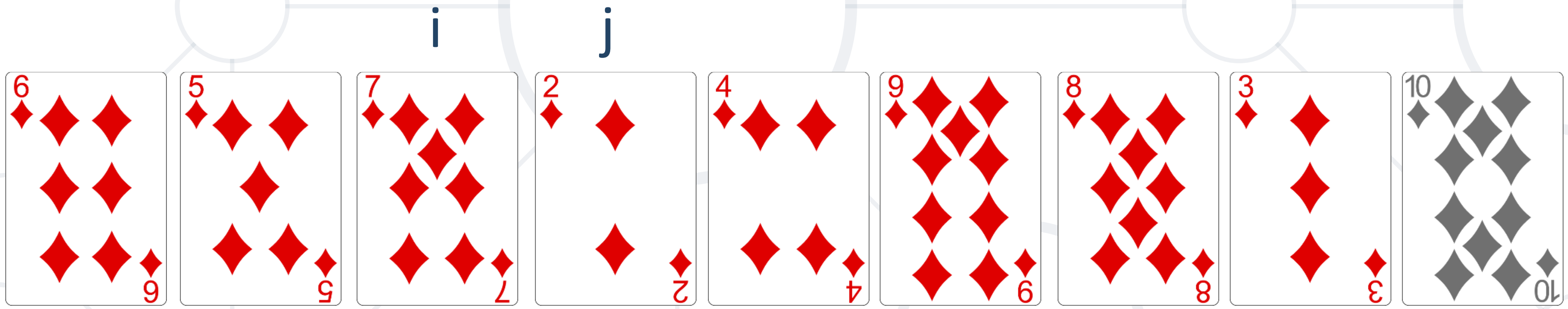
Bubble Sort Visualization



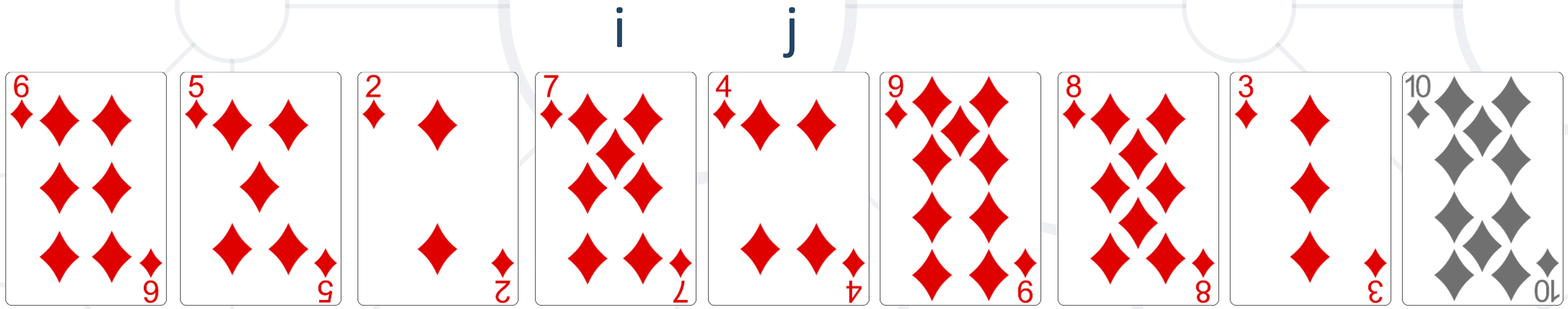
Bubble Sort Visualization



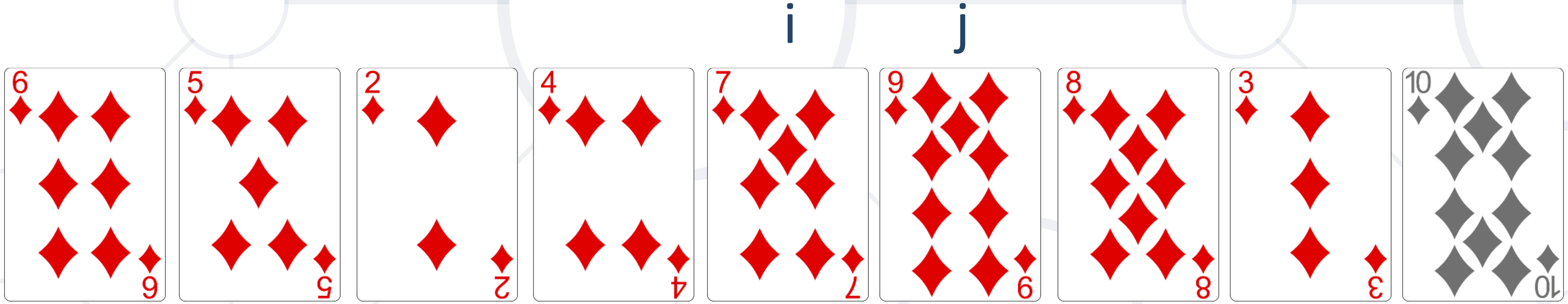
Bubble Sort Visualization



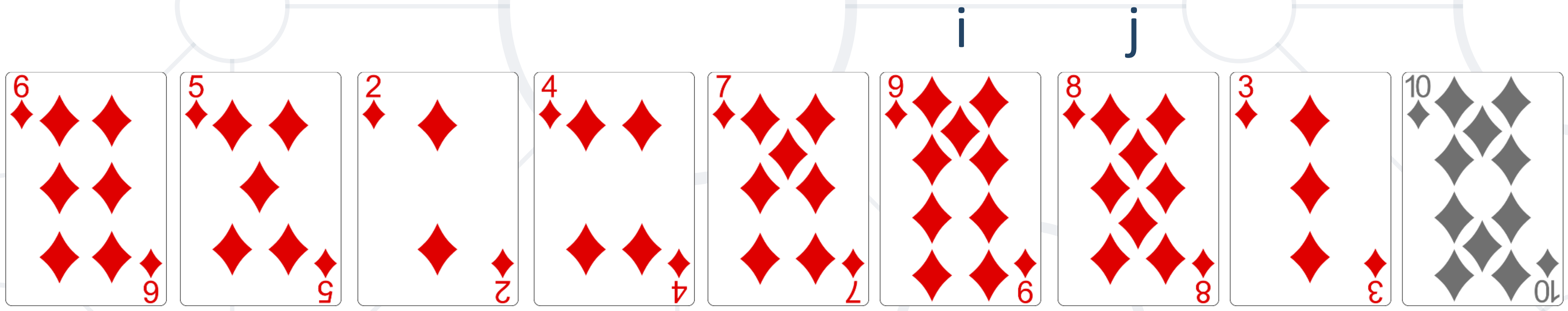
Bubble Sort Visualization



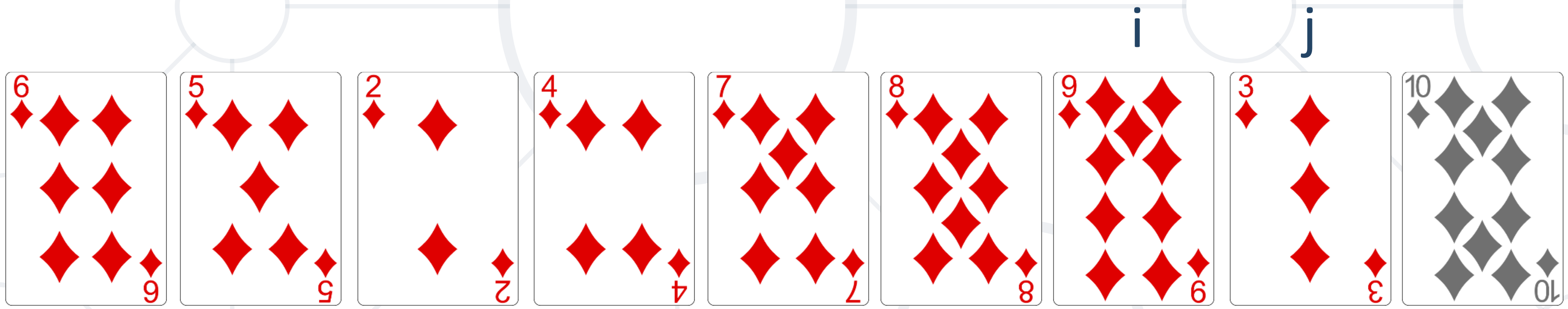
Bubble Sort Visualization



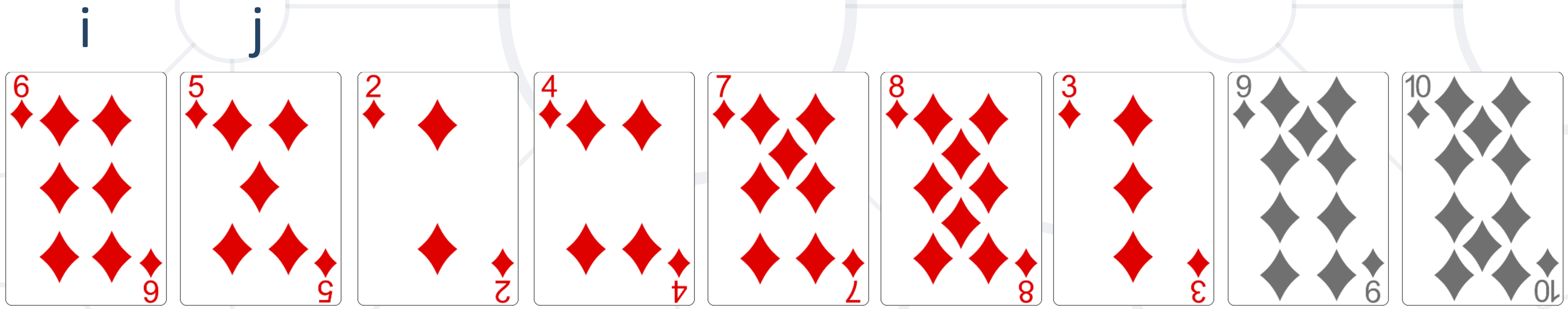
Bubble Sort Visualization



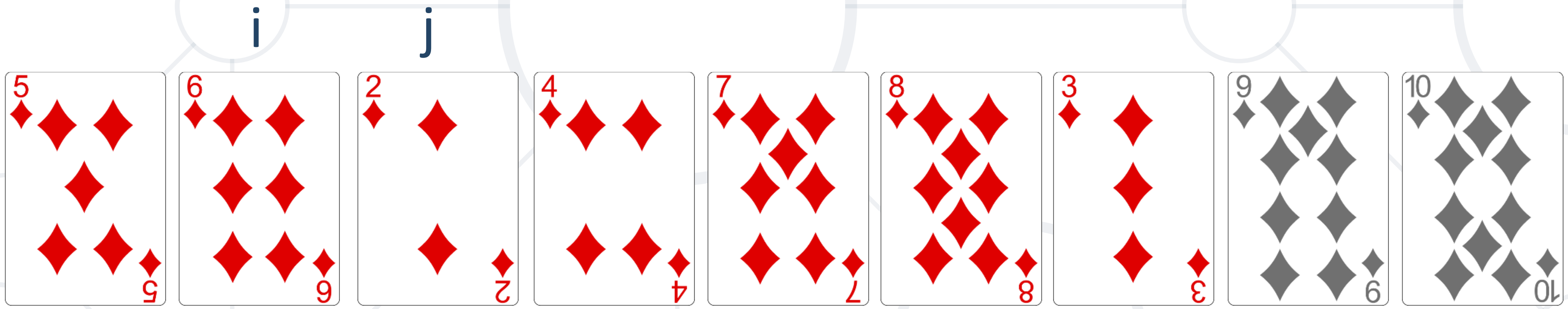
Bubble Sort Visualization



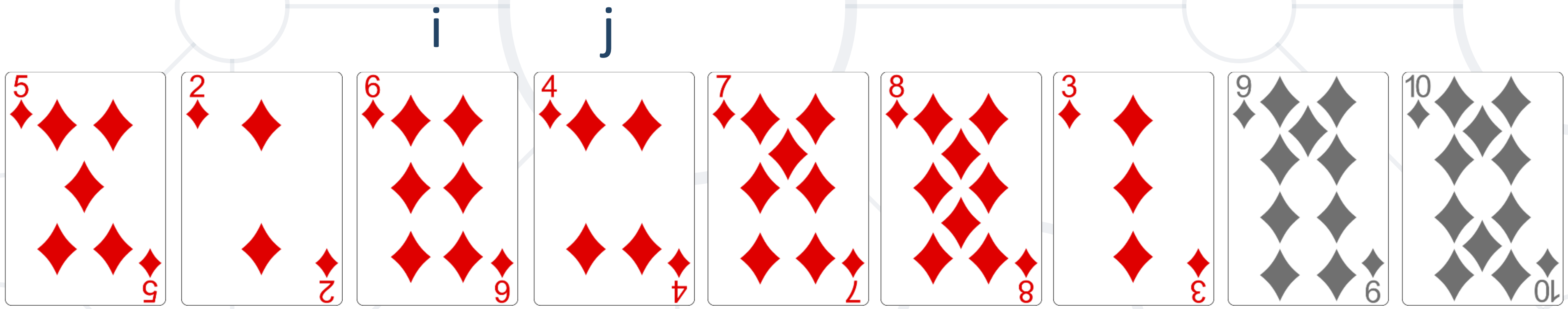
Bubble Sort Visualization



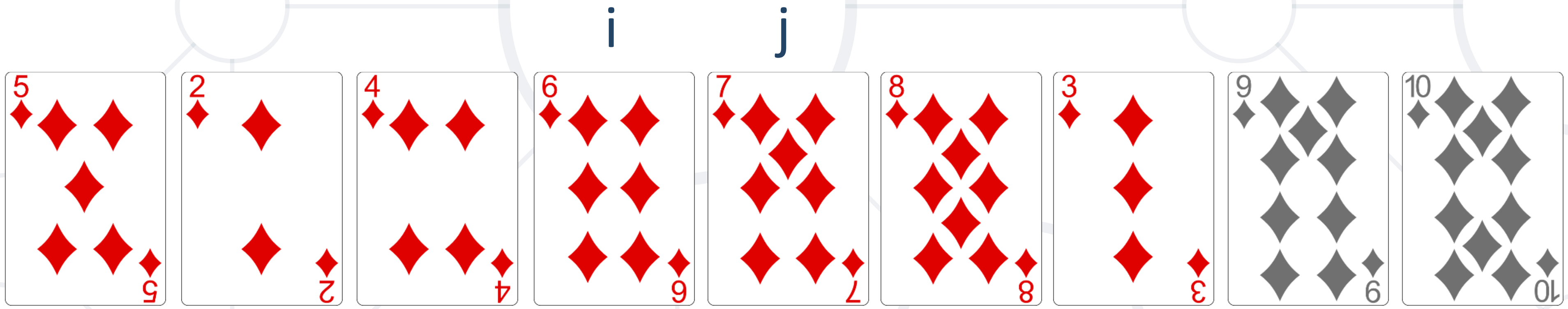
Bubble Sort Visualization



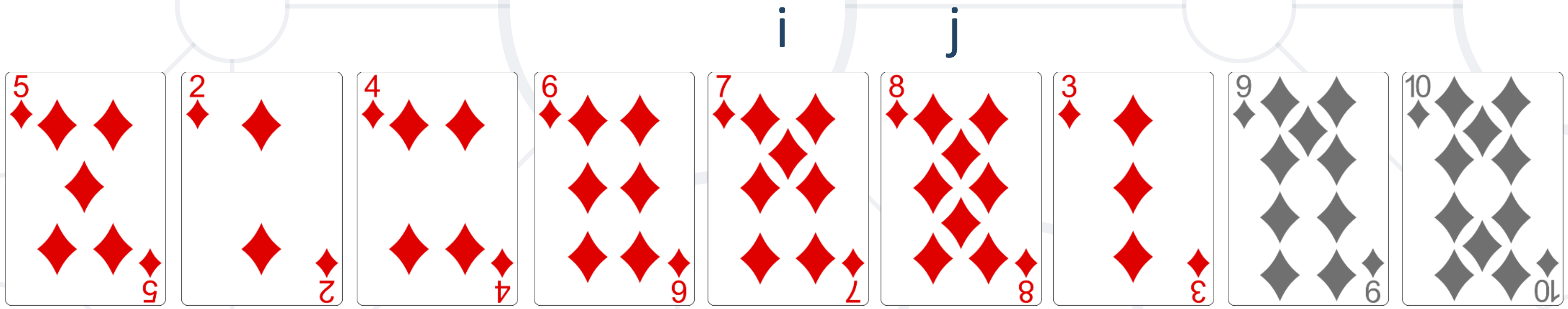
Bubble Sort Visualization



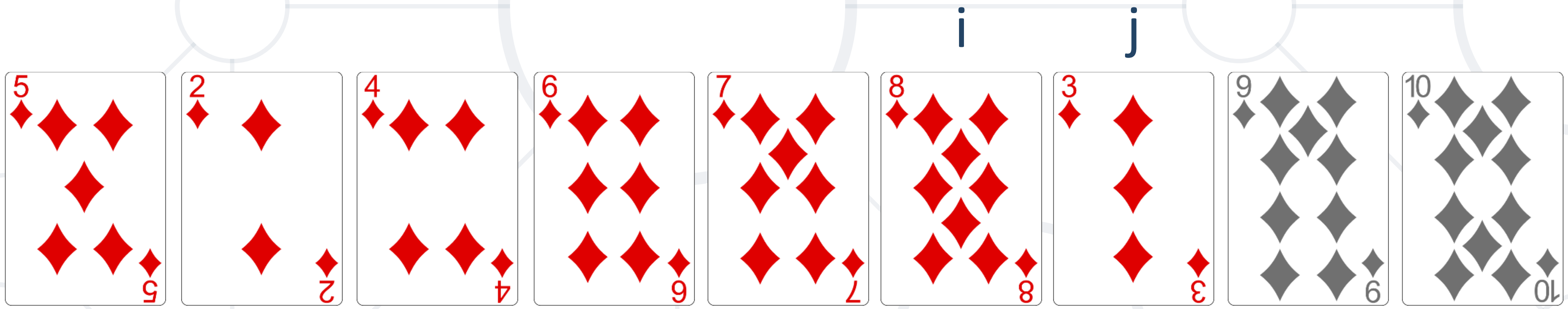
Bubble Sort Visualization



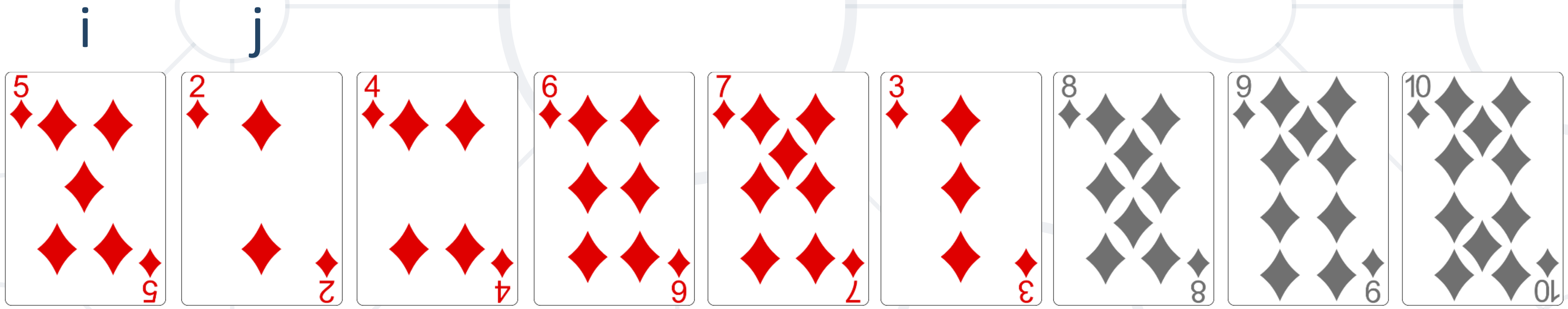
Bubble Sort Visualization



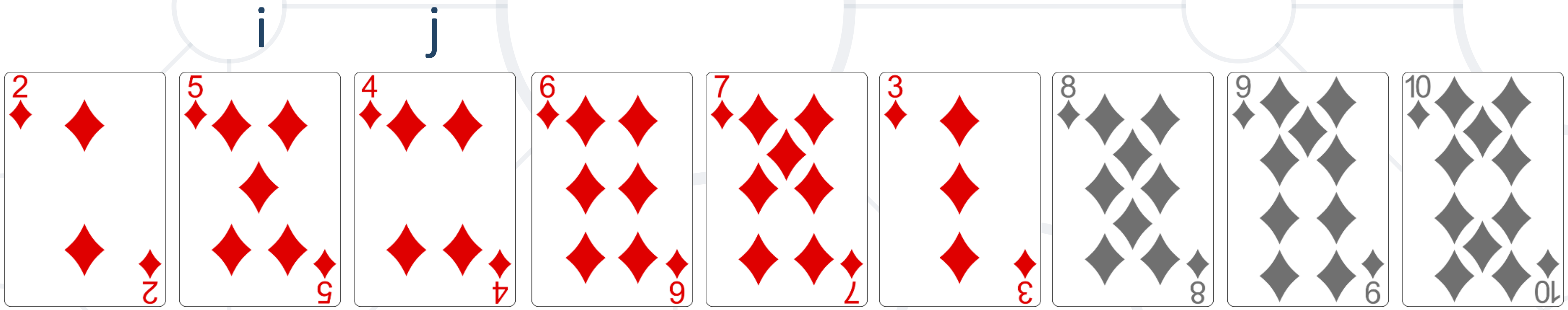
Bubble Sort Visualization



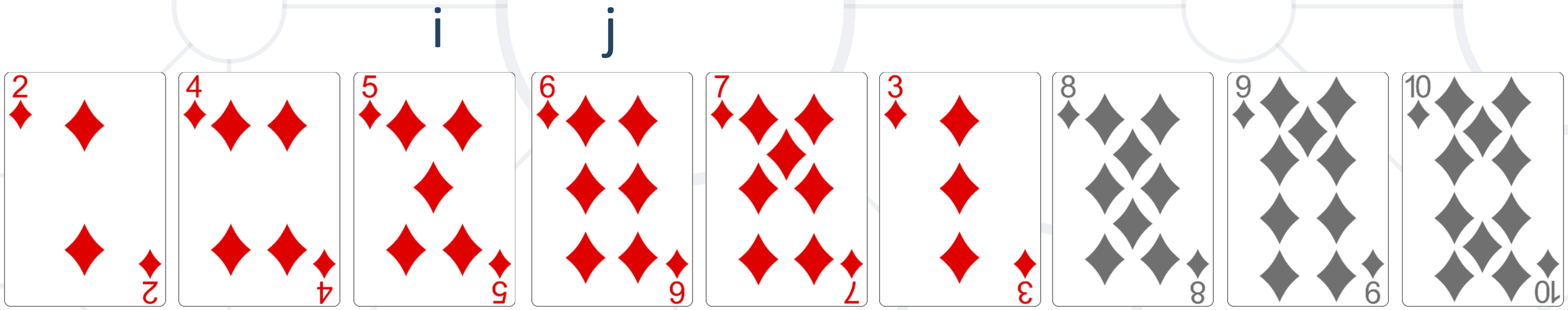
Bubble Sort Visualization



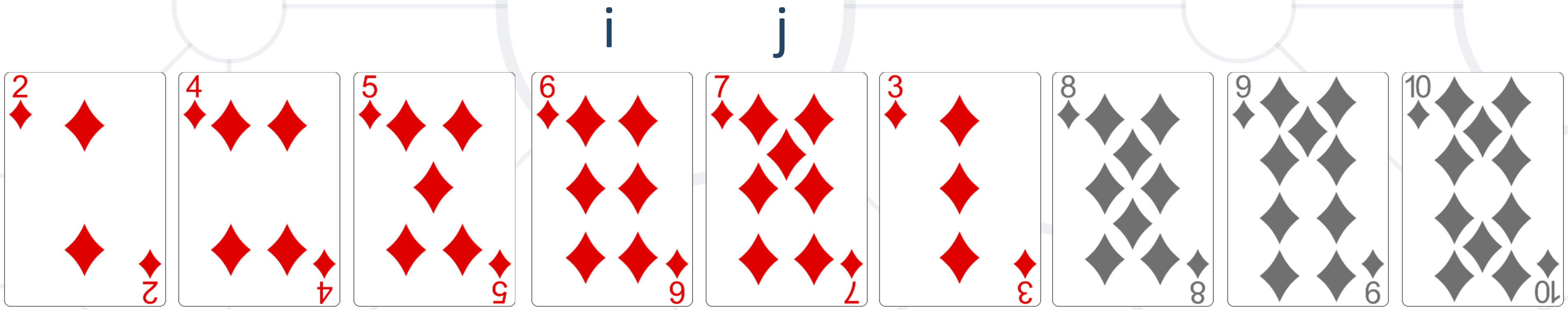
Bubble Sort Visualization



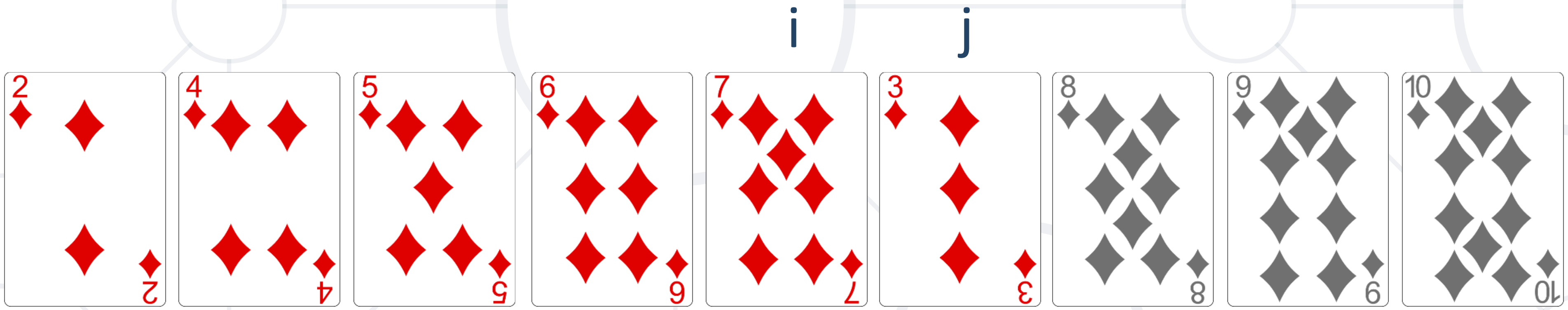
Bubble Sort Visualization



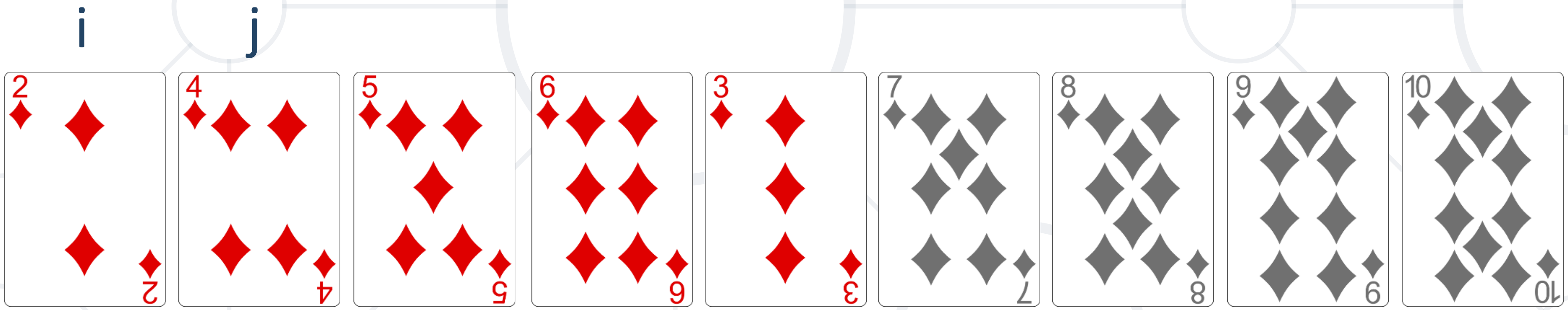
Bubble Sort Visualization



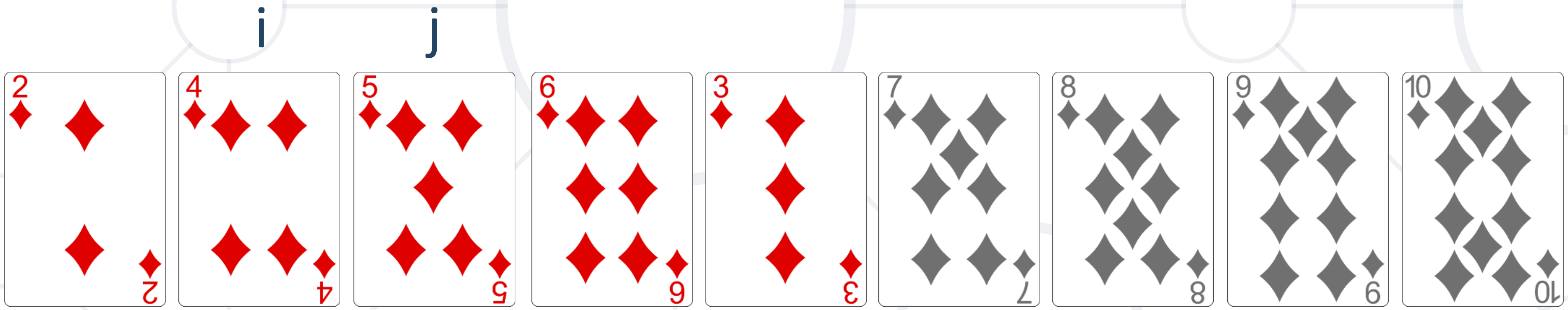
Bubble Sort Visualization



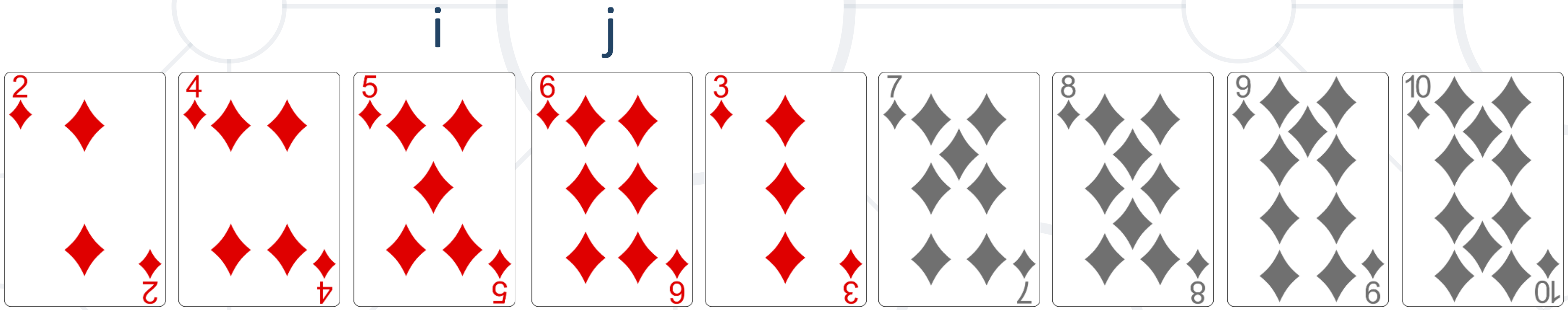
Bubble Sort Visualization



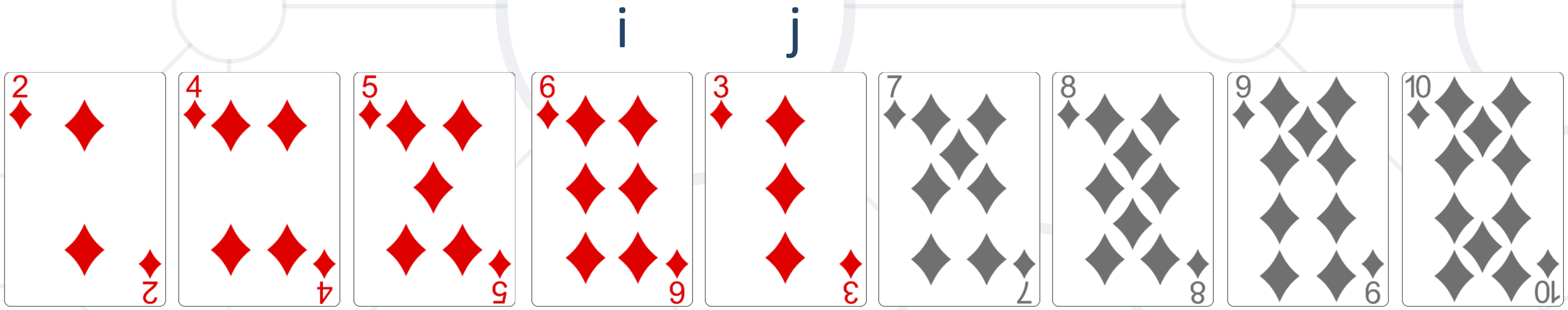
Bubble Sort Visualization



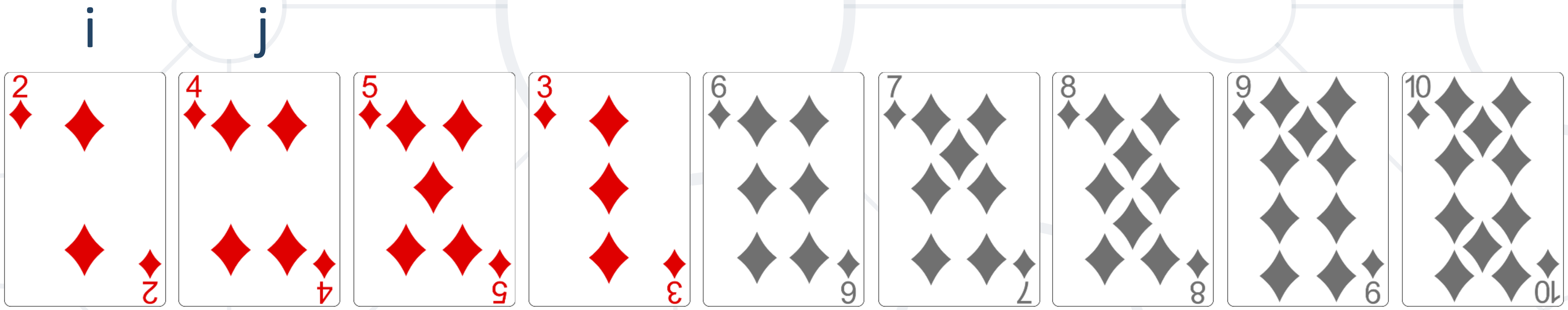
Bubble Sort Visualization



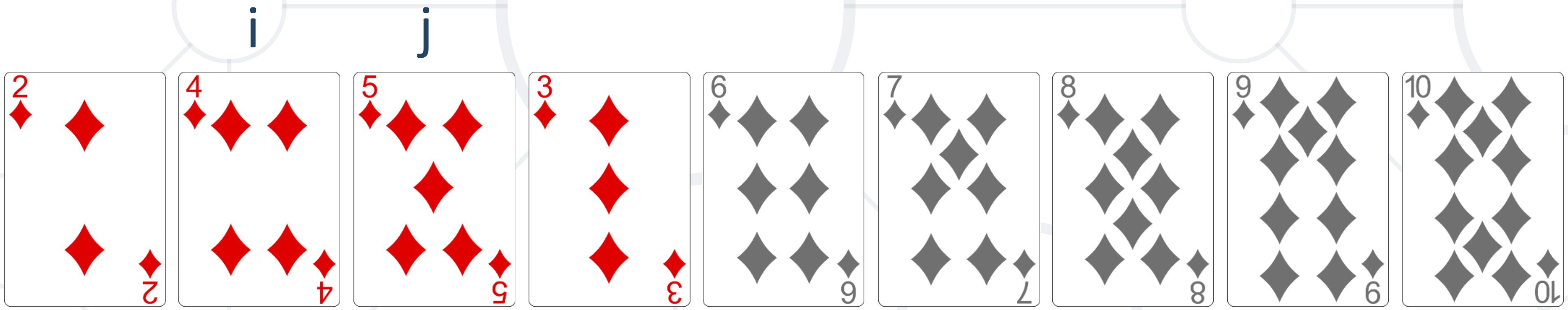
Bubble Sort Visualization



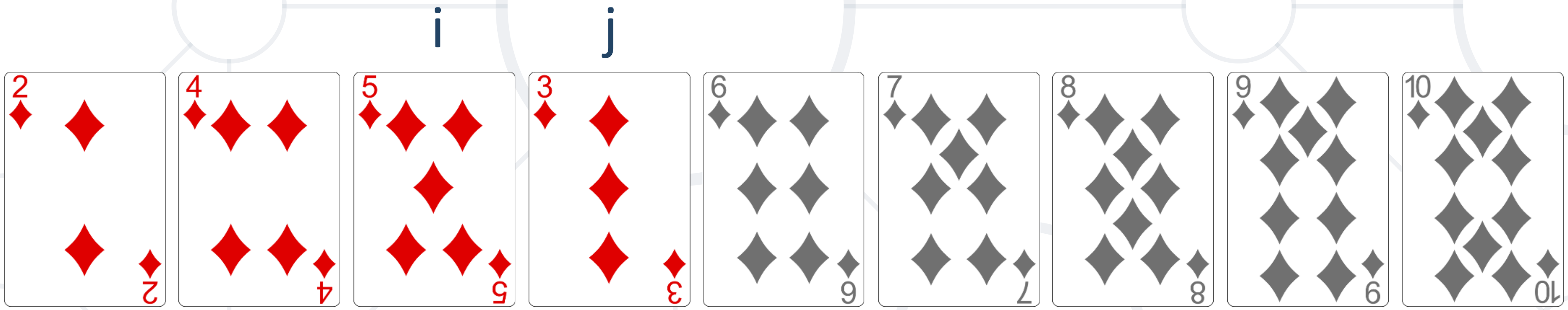
Bubble Sort Visualization



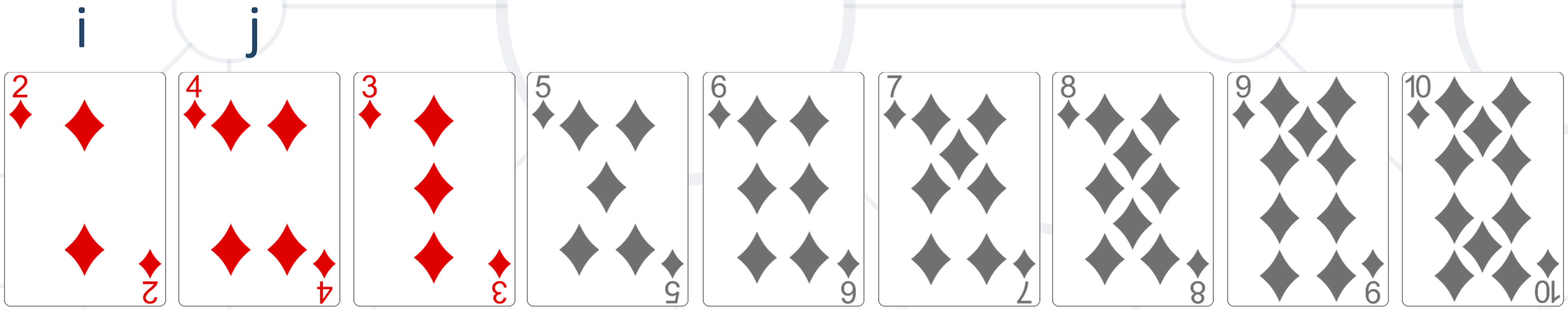
Bubble Sort Visualization



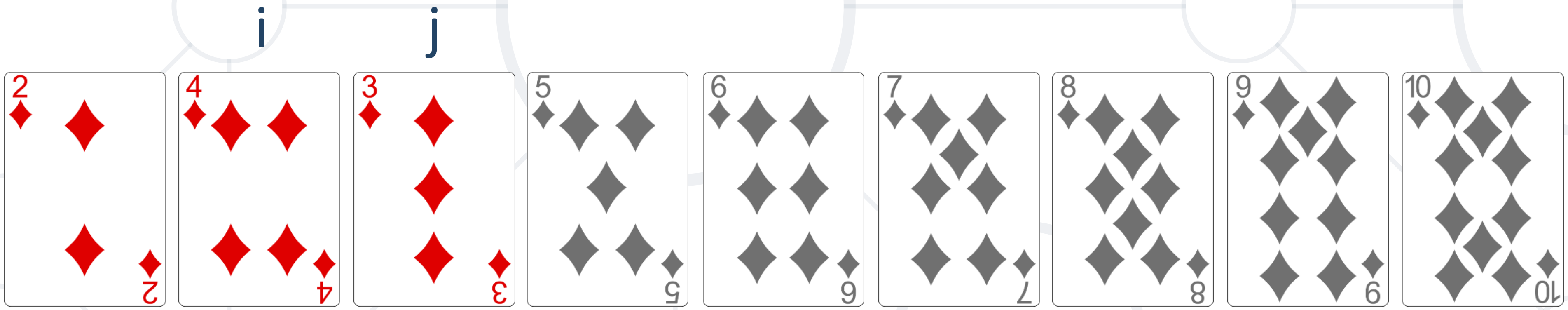
Bubble Sort Visualization



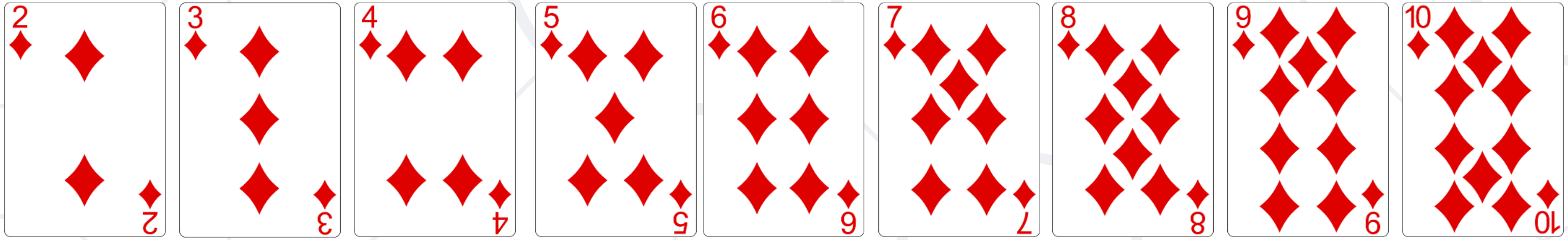
Bubble Sort Visualization



Bubble Sort Visualization



Bubble Sort Visualization



BubbleSort

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

Bubble Sort (2)

```
var numbers = new [] {1, 3, 4, 2, 5, 6};
var isSorted = false;
var i = 0;
while (!isSorted) {
  isSorted = true;
  for (int j = 1; j < numbers.Length - i; j++) {
    if (numbers[j - 1] > numbers[j]) {
      isSorted = false;
      Swap(numbers, j - 1, j);
    }
  }
  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

```
for (int i = 1; i < arr.Length; i++)
{
  var j = i;
  while (j > 0 && arr[j] < arr[j - 1])
  {
    Swap(arr, j, j - 1);
    j--;
  }
```

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

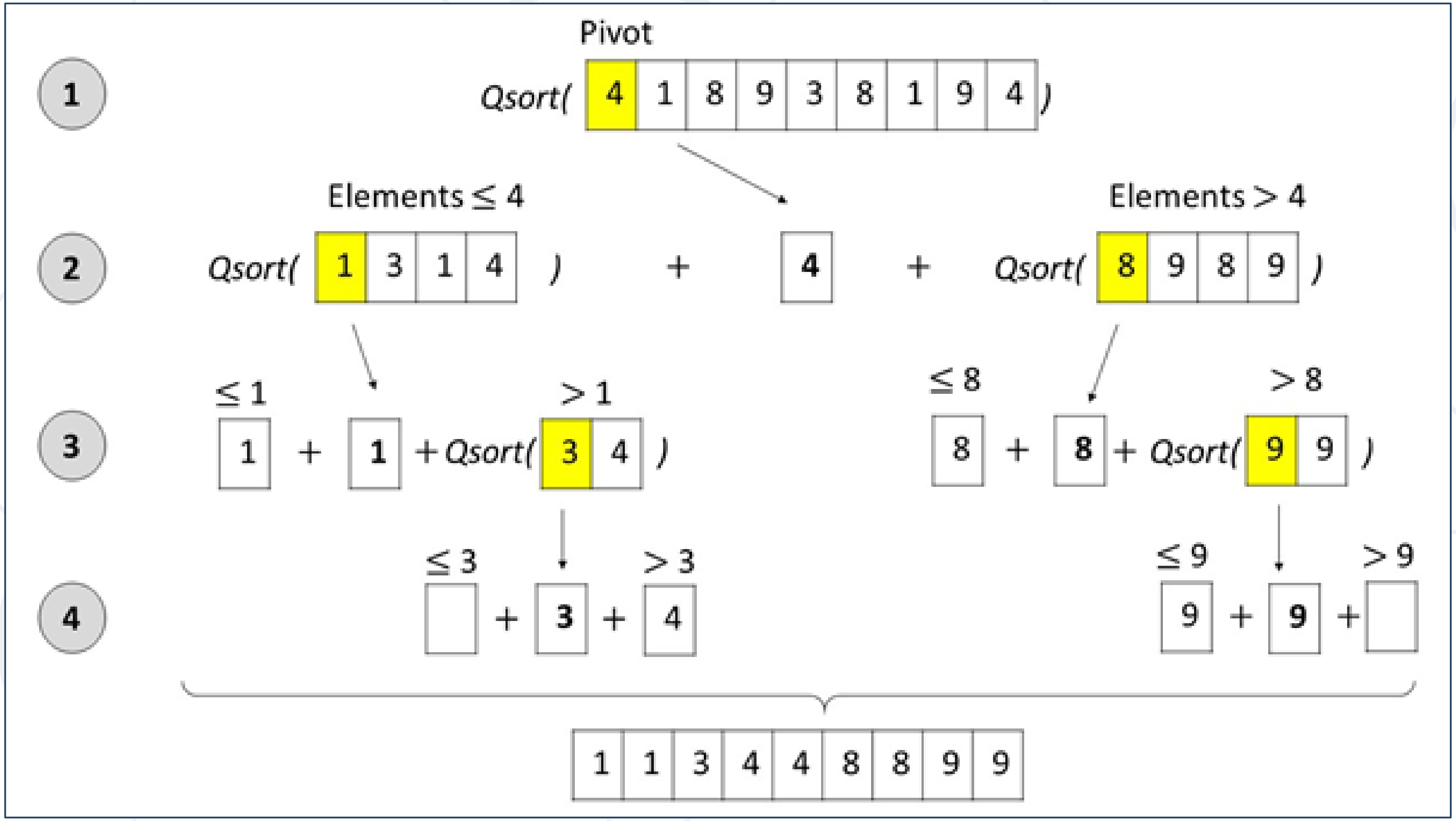
Advanced Sorting Algorithms

QuickSort, MergeSort

Quick Sort

- **QuickSort** – efficient sorting algorithm
  - Choose a pivot; move smaller elements left & larger right; sort left & right
  - Memory: **O(log(n))** stack space (recursion)
  - Time: **$O(n^2)$**
  - Stable: **Depends**
  - Method: **Partitioning**

Quick Sort: Conceptual Overview



Quick Sort (1)

```
public static void QuickSortHelper(
  int[] array, int startIdx, int endIdx)
{
  if (startIdx >= endIdx)
    return;
  var pivotIdx = startIdx;
  var leftIdx = startIdx + 1;
  var rightIdx = endIdx;
  while (leftIdx <= rightIdx) {
    // TODO: Continues on the next slide
  }
  // TODO: Continues on slide Quick Sort (3)
}
```

Quick Sort (2)

```
if (array[leftIdx] > array[pivotIdx] &&
      array[rightIdx] < array[pivotIdx]) {
  Swap(array, leftIdx, rightIdx);
}

if (array[leftIdx] <= array[pivotIdx]) {
  leftIdx += 1;
}

if (array[rightIdx] >= array[pivotIdx]) {
  rightIdx -= 1;
}
```

Quick Sort (3)

```
Swap(array, pivotIdx, rightIdx);

var isLeftSubArraysSmaller =
  rightIdx - 1 - startIdx < endIdx - (rightIdx + 1);
if (isLeftSubArraysSmaller) {
  QuickSortHelper(array, startIdx, rightIdx - 1);
  QuickSortHelper(array, rightIdx + 1, endIdx);
} else {
  QuickSortHelper(array, rightIdx + 1, endIdx);
  QuickSortHelper(array, startIdx, rightIdx - 1);
}
```

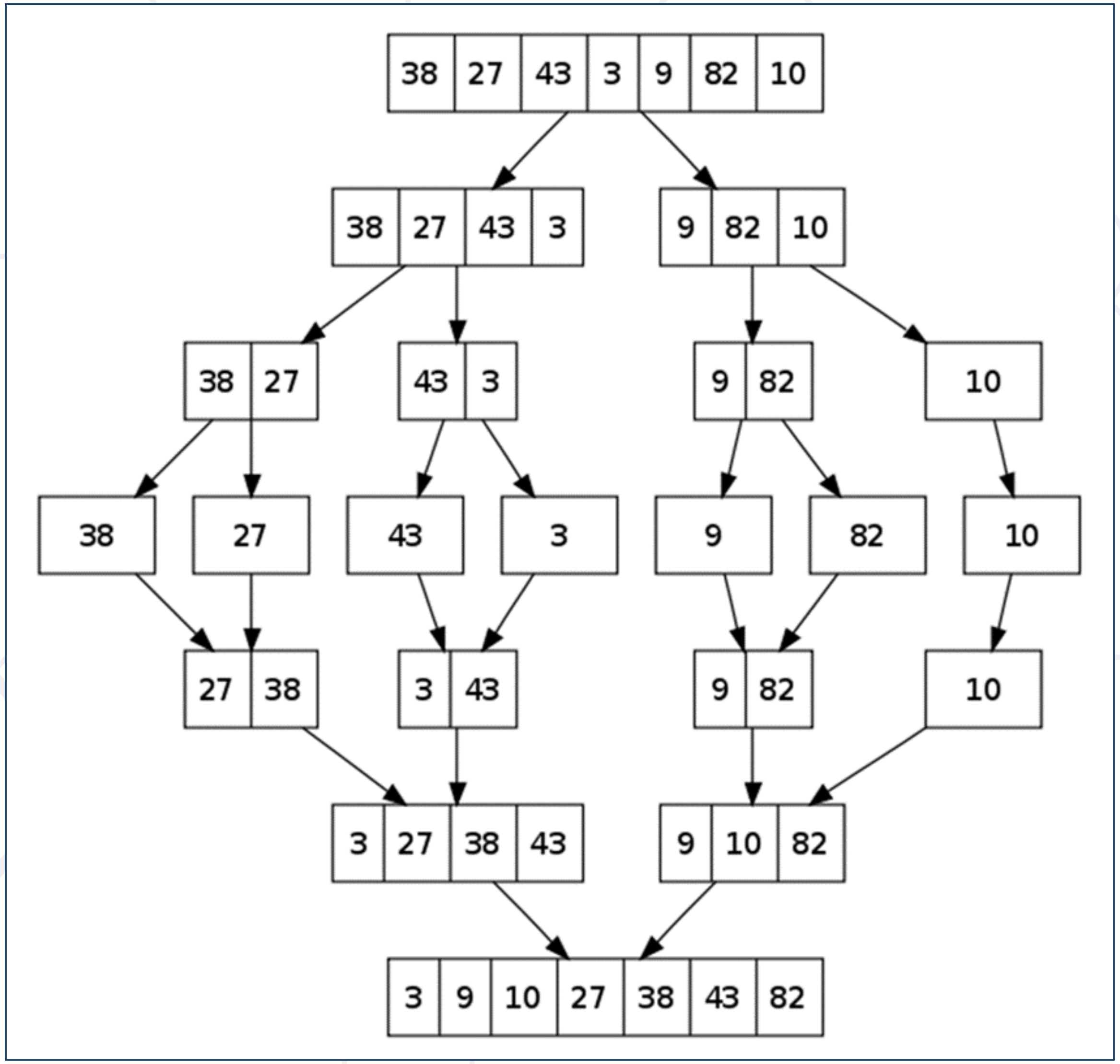
Comparison of Sorting Algorithms

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	n	n^2	n^2	1	Yes	Insertion
Quick	n * log(n)	n * log(n)	n^2	1	Depends	Partitioning

Merge Sort

- **Merge sort** is efficient sorting algorithm
- Divide the list into sub-lists (typically 2 sub-lists)
  1. Sort each sub-list (recursively call merge-sort)
  2. Merge the sorted sub-lists into a single list
- Memory: **O(n)** / **O(n*log(n))**
- Time: **O(n*log(n))**
- Highly parallelizable on multiple cores / machines → up to **O(log(n))**

Merge Sort: Conceptual Overview



Merge Sort (1)

```
// Memory: O(n*log(n))
public static int[] MergeSort(int[] array)
{
  if (array.Length == 1)
    return array;

  var middleIdx = array.Length / 2;
  var leftHalf = array.Take(middleIdx).ToArray();
  var rightHalf = array.Skip(middleIdx).ToArray();

  return MergeArrays(MergeSort(leftHalf), MergeSort(rightHalf));
}
```

Merge Sort (2)

```
public static int[] MergeArrays(int[] left, int[] right) {
  var sorted = new int[left.Length + right.Length];
  var sortedIdx = 0; var leftIdx = 0; var rightIdx = 0;
  while (leftIdx < left.Length && rightIdx < right.Length) {
    if (left[leftIdx] < right[rightIdx]) {
      sorted[sortedIdx++] = left[leftIdx++];
    } else {
      sorted[sortedIdx++] = right[rightIdx++];
    }
  }
  // TODO: Take remaining elements either from the left or right
  return sorted;
}
```

Merge Sort (3)

```
while (leftIdx < left.Length) {
  sorted[sortedIdx] = left[leftIdx];
  sortedIdx += 1;
  leftIdx += 1;
}

while (rightIdx < right.Length) {
  sorted[sortedIdx] = right[rightIdx];
  sortedIdx += 1;
  rightIdx += 1;
}
```

Merge Sort

```
// Memory: O(n)
public static int[] MergeSort(int[] array)
{
  if (array.Length <= 1)
    return array;

  var copy = new int[array.Length];
  Array.Copy(array, copy, array.Length);

  MergeSortHelper(array, copy, 0, array.Length - 1);

  return array;
}
```

Merge Sort (2)

```
public static void MergeSortHelper(
  int[] source, int[] copy, int leftIdx, int rightIdx)
{
  if (leftIdx >= rightIdx)
    return;

  var middleIdx = (leftIdx + rightIdx) / 2;
  MergeSortHelper(copy, source, leftIdx, middleIdx);
  MergeSortHelper(copy, source, middleIdx + 1, rightIdx);

  MergeArrays(source, copy, leftIdx, middleIdx, rightIdx);
}
```

Merge Sort (3)

```
public static void MergeArrays(
  int[] source, int[] copy, int startIdx, int middleIdx, int endIdx)
{
  var sourceIdx = startIdx;
  var leftIdx = startIdx; var rightIdx = middleIdx + 1;
  while (leftIdx <= middleIdx && rightIdx <= endIdx) {
    if (copy[leftIdx] < copy[rightIdx])
      source[sourceIdx++] = copy[leftIdx++];
    else
      source[sourceIdx++] = copy[rightIdx++];
  }
  // TODO: Take remaining elements either from the left or right
}
```

Merge Sort (4)

Software University

```
while (leftIdx <= middleIdx)
{
  source[sourceIdx] = copy[leftIdx];
  leftIdx += 1;
  sourceIdx += 1;
}

while (rightIdx <= endIdx)
{
  source[sourceIdx] = copy[rightIdx];
  rightIdx += 1;
  sourceIdx += 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	n	n^2	n^2	1	Yes	Insertion
Quick	n * log(n)	n * log(n)	n^2	1	Depends	Partitioning
Merge	n * log(n)	n * log(n)	n * log(n)	1	Yes	Merging

Greedy Algorithms

Picking Locally Optimal Solution

Greedy Algorithms

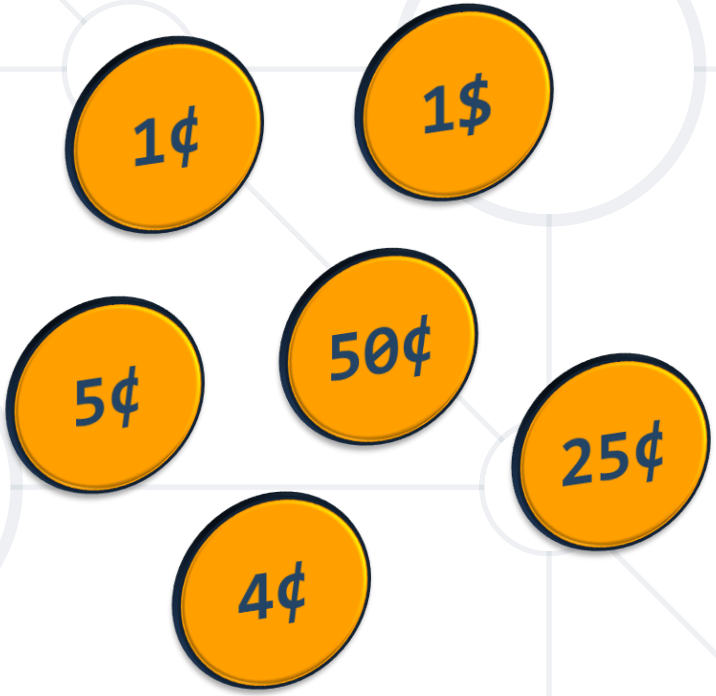
- Used for solving optimization problems
- Usually more efficient than the other algorithms
- Can produce a **non-optimal** (incorrect) result
- Pick the **best local** solution
  - The optimum for a **current** position and point of view
- Greedy algorithms assume that always choosing a **local** optimum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

10¢ 5¢ 2¢ 2¢ 1¢ 1¢

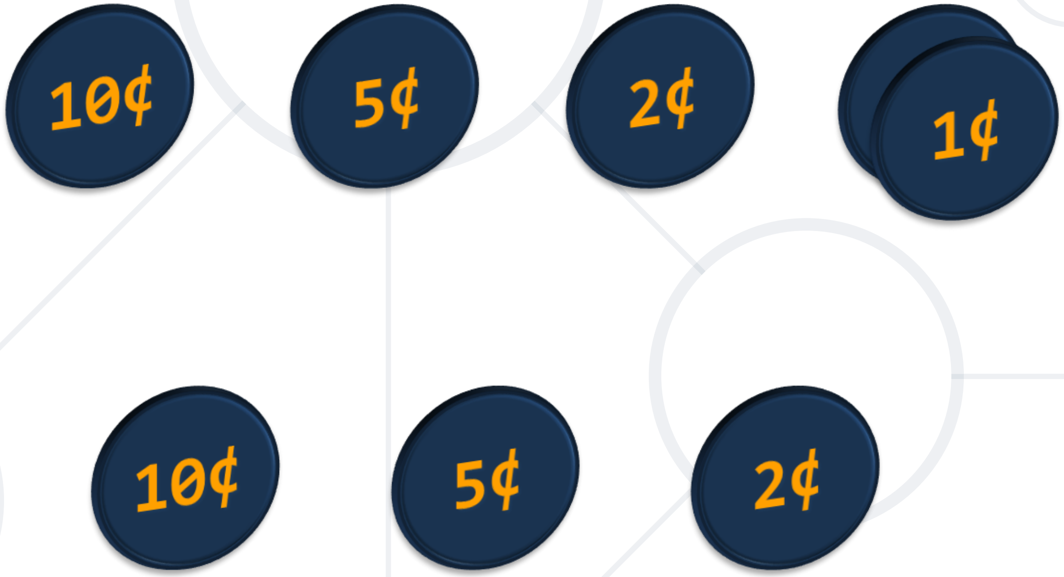
Actual: 15

10¢ 5¢

Sum of Coins Visualization

Target: 18

Actual: 17



Sum of Coins Visualization

Target: 18

Actual: 18

Solution: Sum of Coins (1)

```
var coins = // Read an array of integers and sort it in desc.
var target = int.Parse(Console.ReadLine());
var counter = 0;
var coinsIndex = 0;
var sb = new StringBuilder();
while (target > 0 && coinsIndex < coins.Length) {
  var currentCoin = coins[coinsIndex++];
  var coinsCount = target / currentCoin;
  if (coinsCount > 0) {
    counter += coinsCount;
    target -= currentCoin * coinsCount;
    sb.AppendLine($"{coinsCount} coin(s) with value {currentCoin}");
  }
}
// TODO: Print the output
```

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**

```
1, 2, 3, 4, 5
4
1
2, 4
5
3
```

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

Solution: Set Cover

```
// Read the input elements – universe and sets
var selectedSets = new List<int[]>();
while (universe.Count > 0) {
  var currentSet = sets
    .OrderByDescending(s => s.Count(e => universe.Contains(e)))
    .FirstOrDefault();
  foreach (var number in currentSet)
  { universe.Remove(number); }

  sets.Remove(currentSet);
  selectedSets.Add(currentSet);
}
// TODO: Pirnt the output
```

Greedy Failure Cases

Greedy Algorithms Often Fail

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 Substructure and Greedy Choice Property

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

Summary

- **Searching** algorithms
  - Binary Search, Linear Search
- **Slow** sorting algorithms:
  - Selection sort, Bubble sort, Insertion sort
- **Fast** sorting algorithms:
  - Quick sort, Merge sort, etc.
  - How to choose the most appropriate algorithm?
- **Greedy Algorithms**

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

License

Trainings @ Software University (SoftUni)

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