Searching and Sorting Algorithms

Binary Search, Selection, Bubble Sort, Insertion, QuickSort and MergerSort

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.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

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.

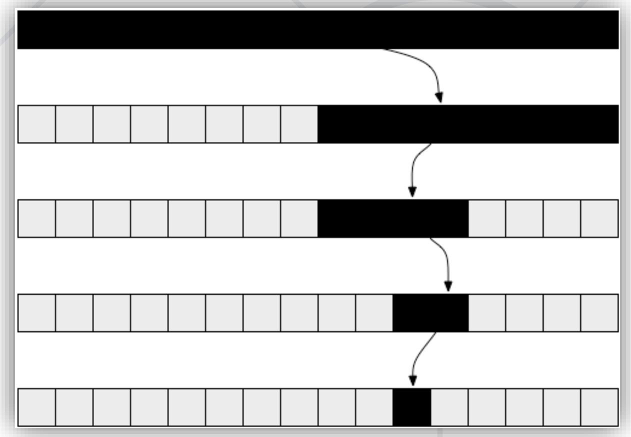
Linear Search

- **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)

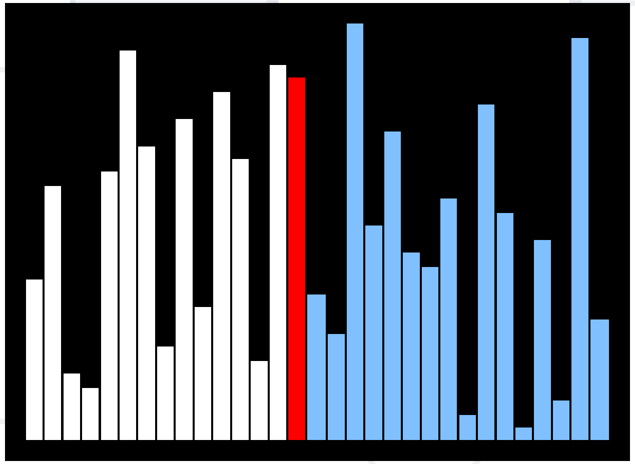
```python
def binary_search(numbers, target):
    left = 0
    right = len(numbers) - 1
    while left <= right:
        mid_idx = (left + right) // 2
        mid_el = numbers[mid_idx]
        if mid_el == target:
            return mid_idx
        if mid_el < target:
            left = mid_idx + 1
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

Simple Sorting Algorithms

Selection, Bubble and Insertion Sort

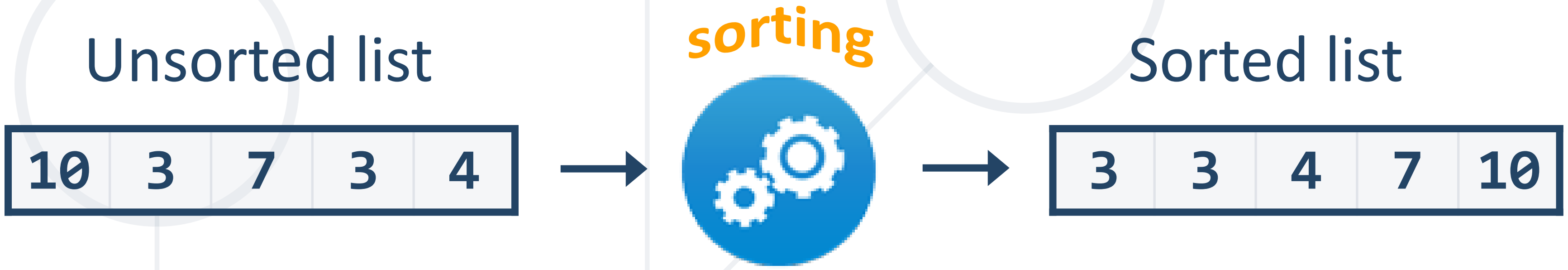
What is a Sorting Algorithm?

- **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:

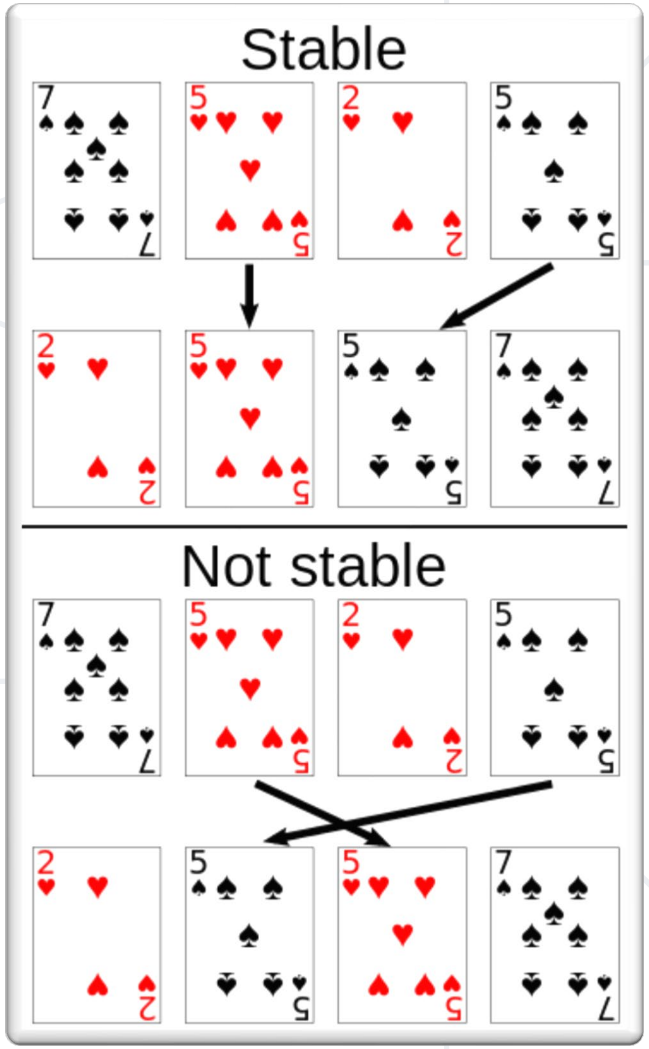


Sorting Algorithms: Classification

- 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

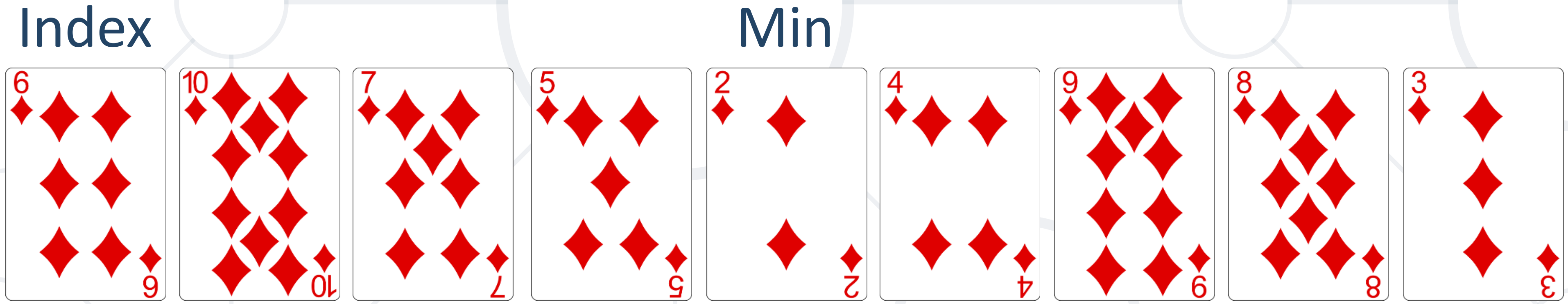
- **Stable** sorting algorithms
  - Maintain the order of equal elements
  - If two items compare as equal, their relative order is preserved
- **Unstable** sorting algorithms
  - Rearrange the equal elements in unpredictable order
- Often **different elements** have **same key** used for equality comparing



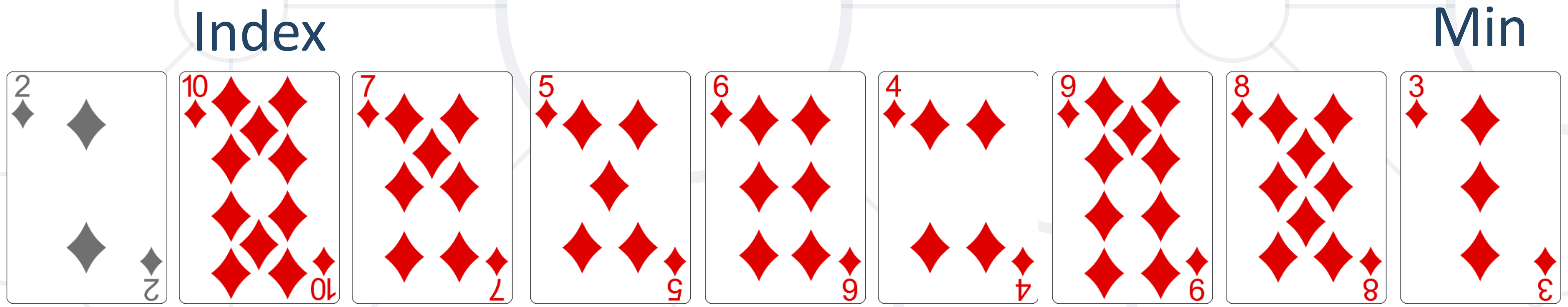
Selection Sort

- **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

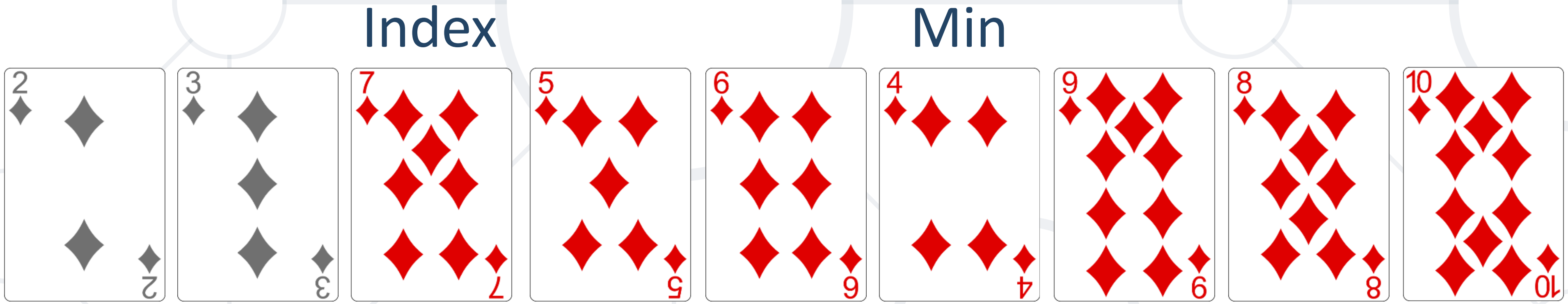
Selection Sort Visualization (1)



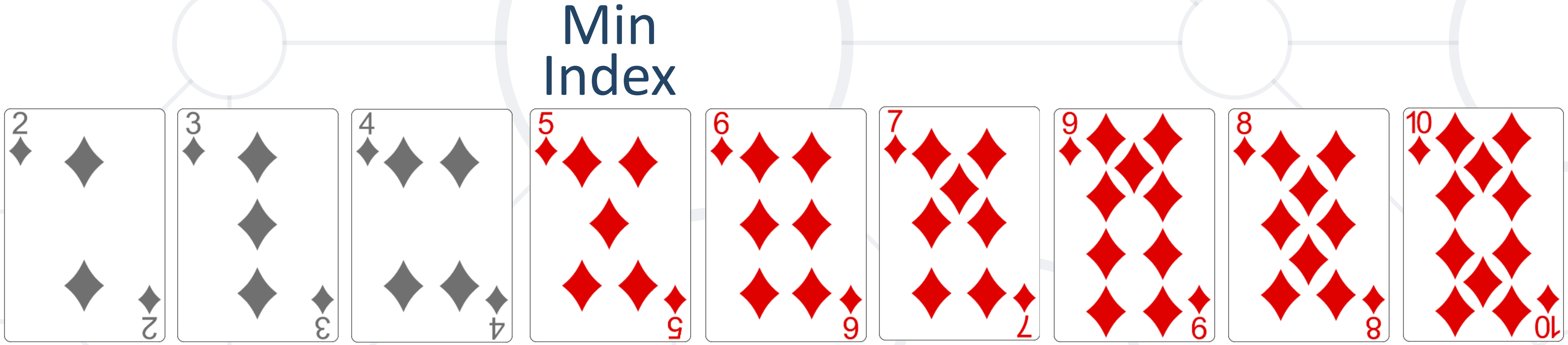
Selection Sort Visualization (2)



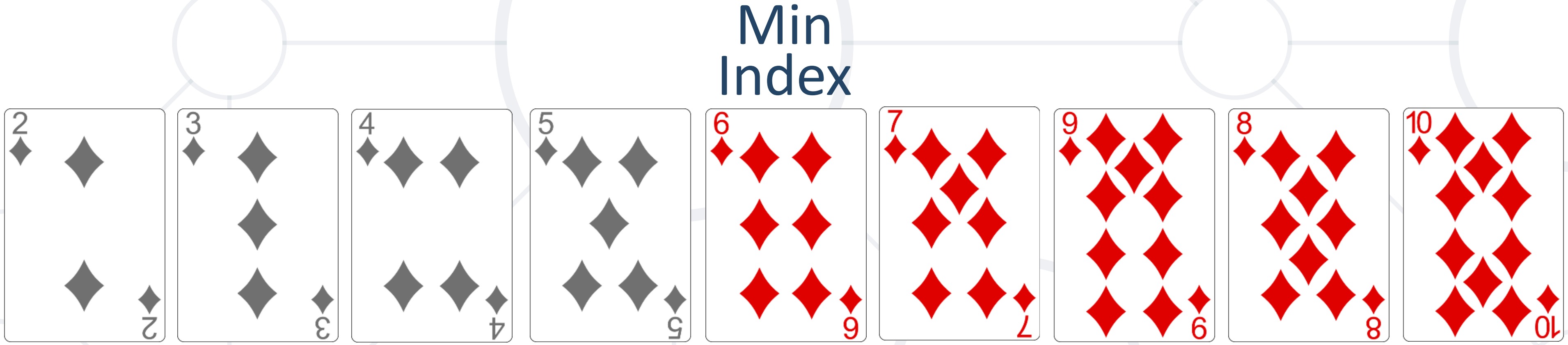
Selection Sort Visualization (3)



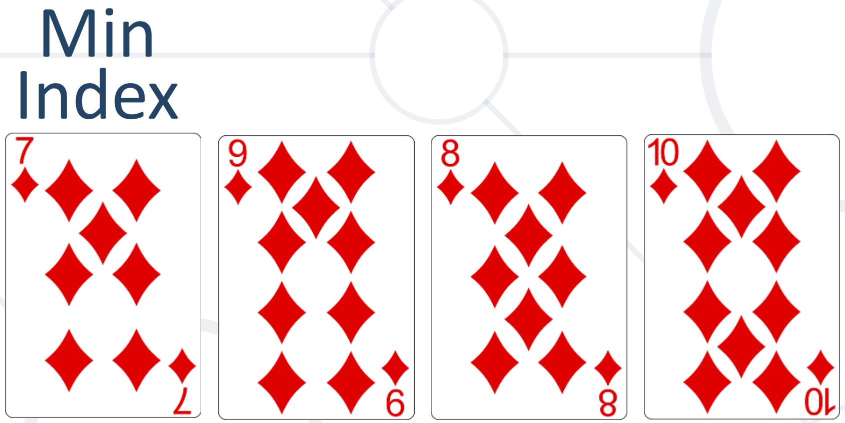
Selection Sort Visualization (4)

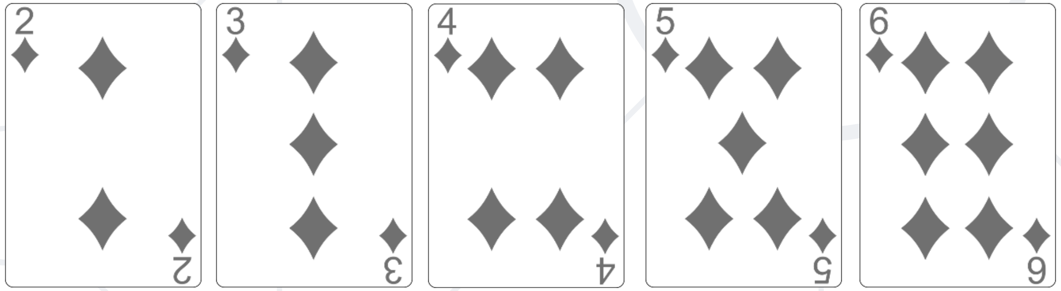


Selection Sort Visualization (5)

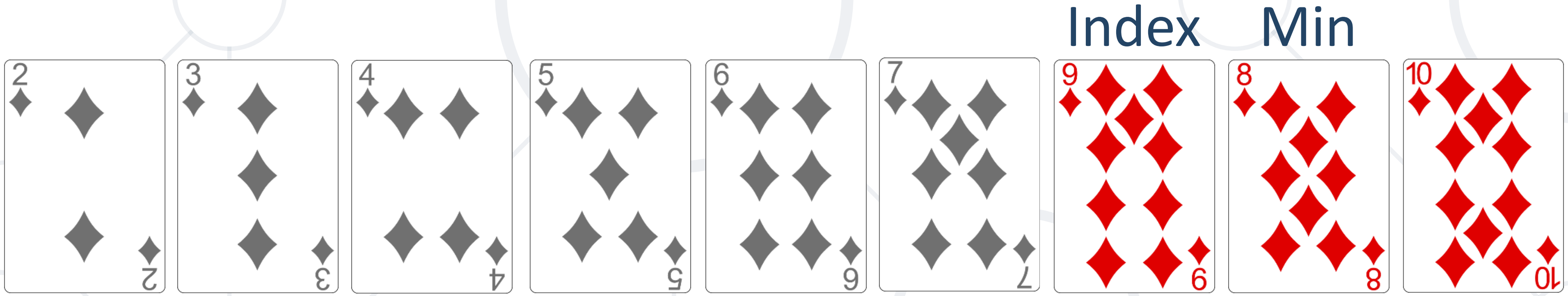


Selection Sort Visualization (6)

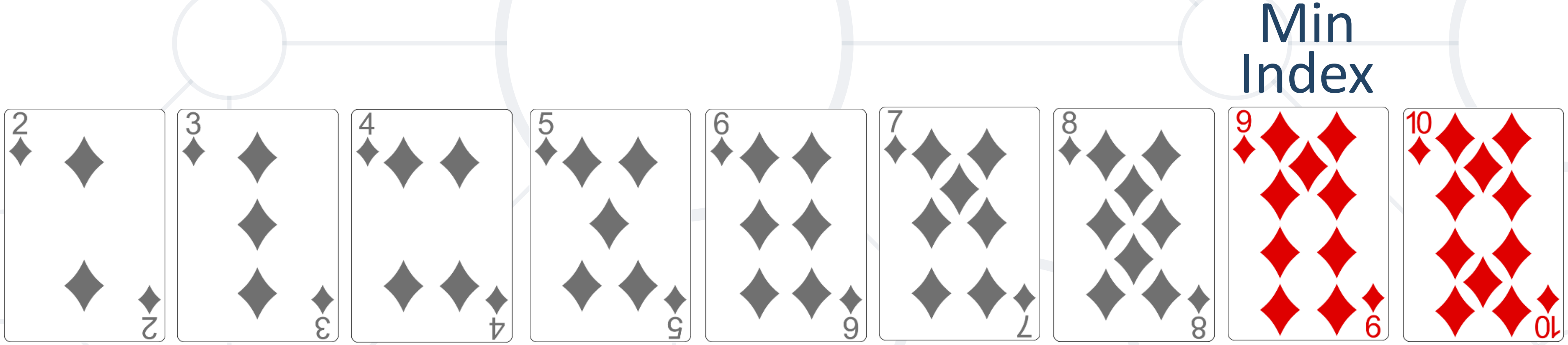




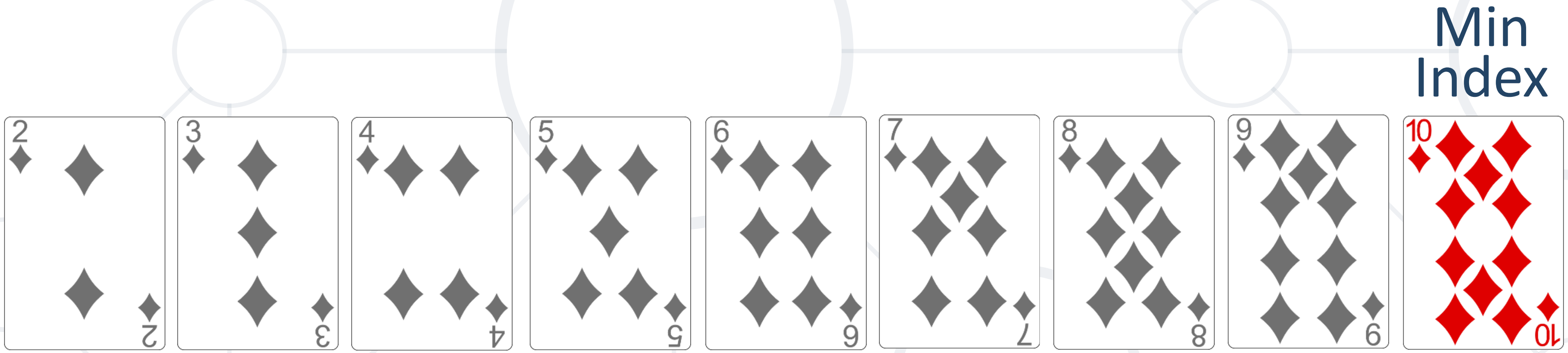
Selection Sort Visualization (7)



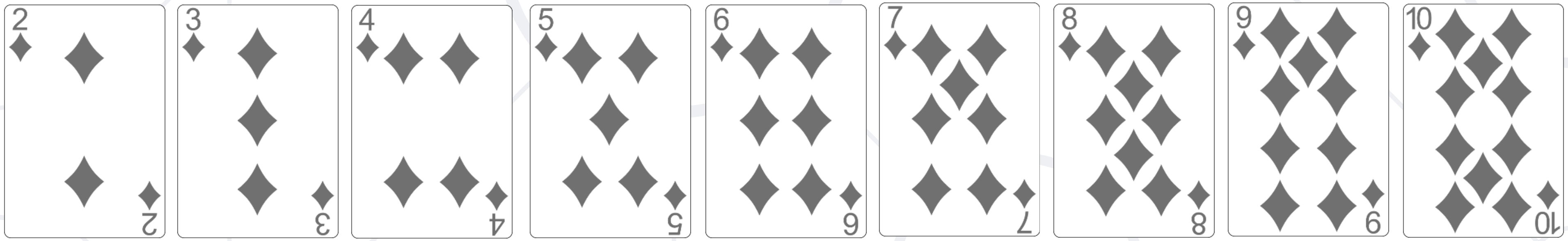
Selection Sort Visualization (8)



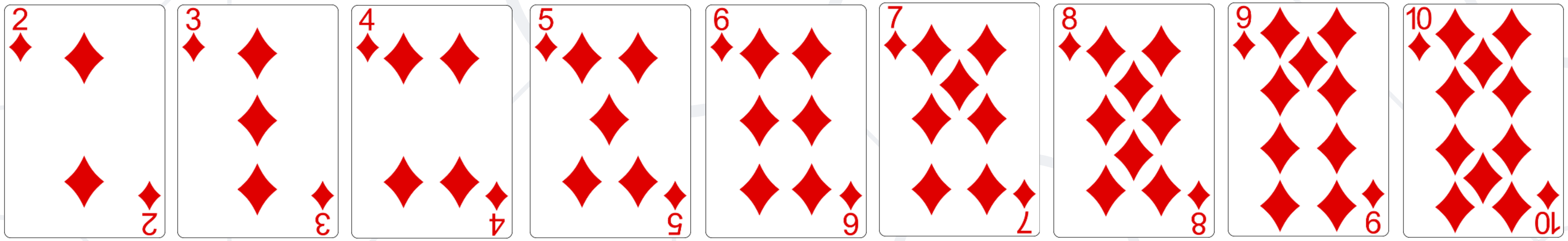
Selection Sort Visualization (9)



Selection Sort Visualization (10)



Selection Sort Visualization (11)



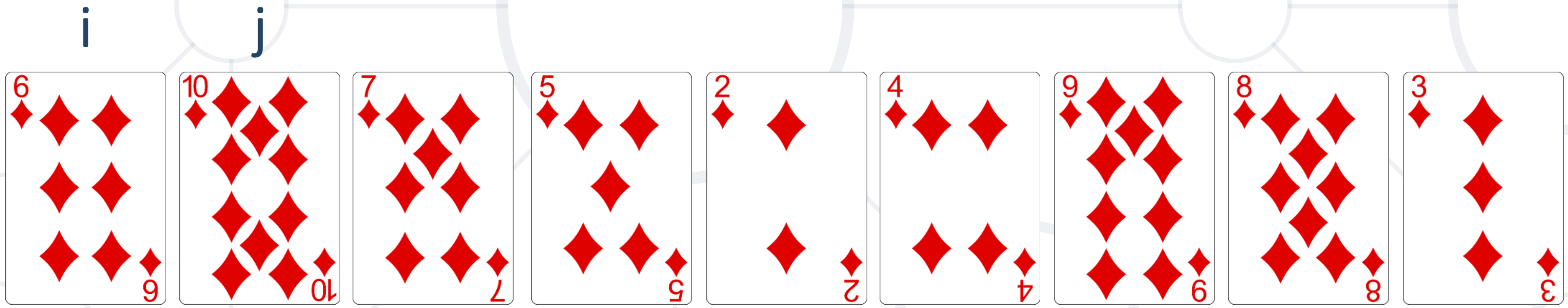
Selection Sort Code

```
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]
```

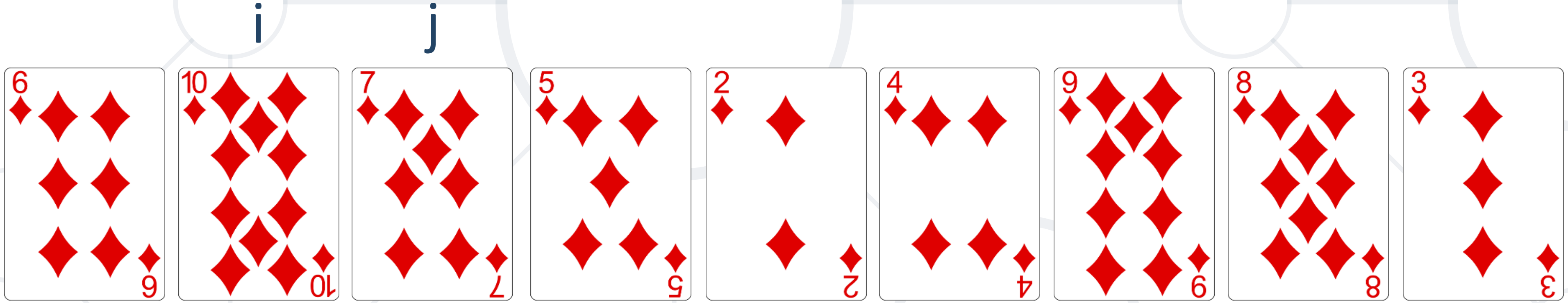
Bubble Sort

- **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

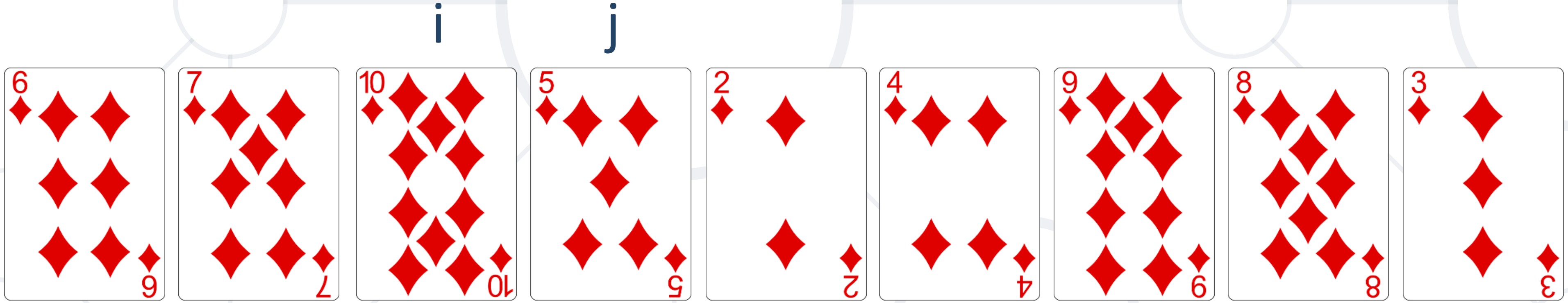
Bubble Sort Visualization (1)



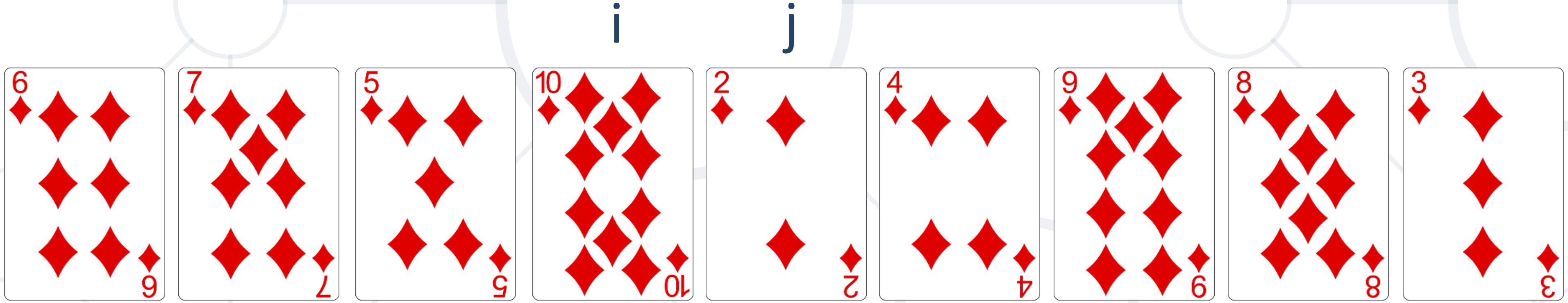
Bubble Sort Visualization (2)



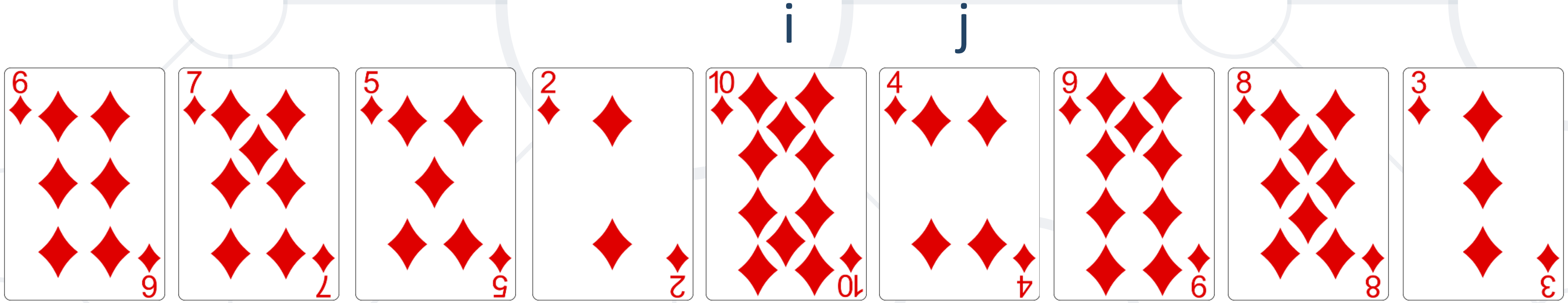
Bubble Sort Visualization (3)



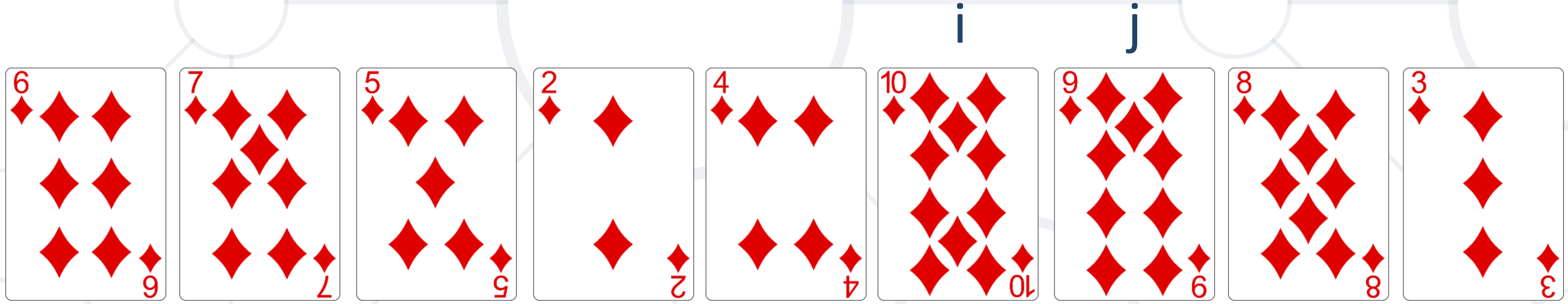
Bubble Sort Visualization (4)



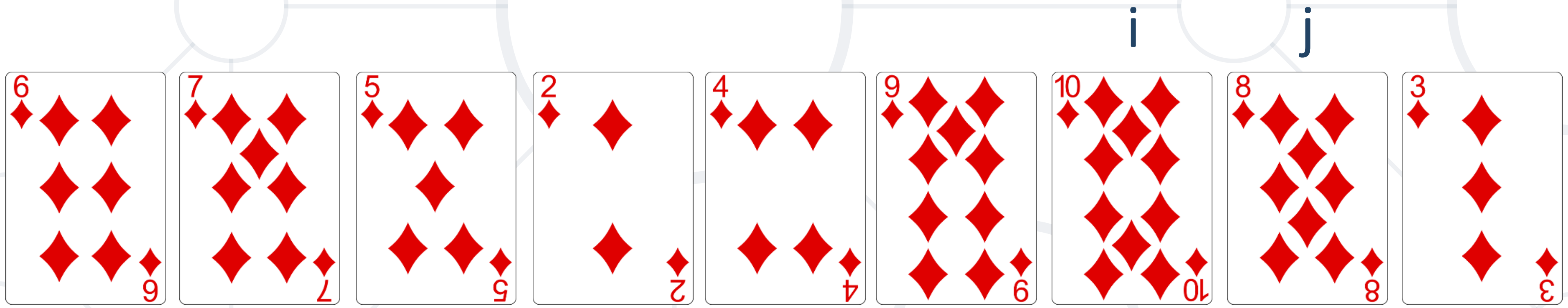
Bubble Sort Visualization (5)



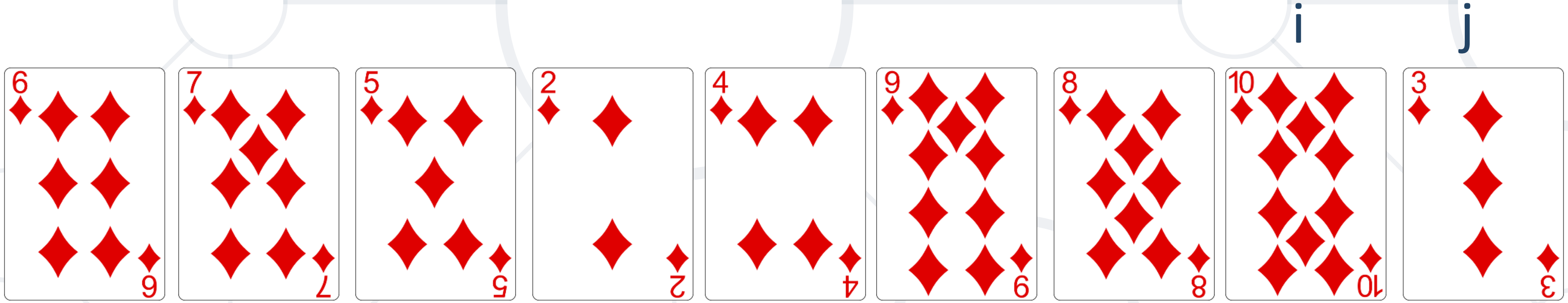
Bubble Sort Visualization (6)



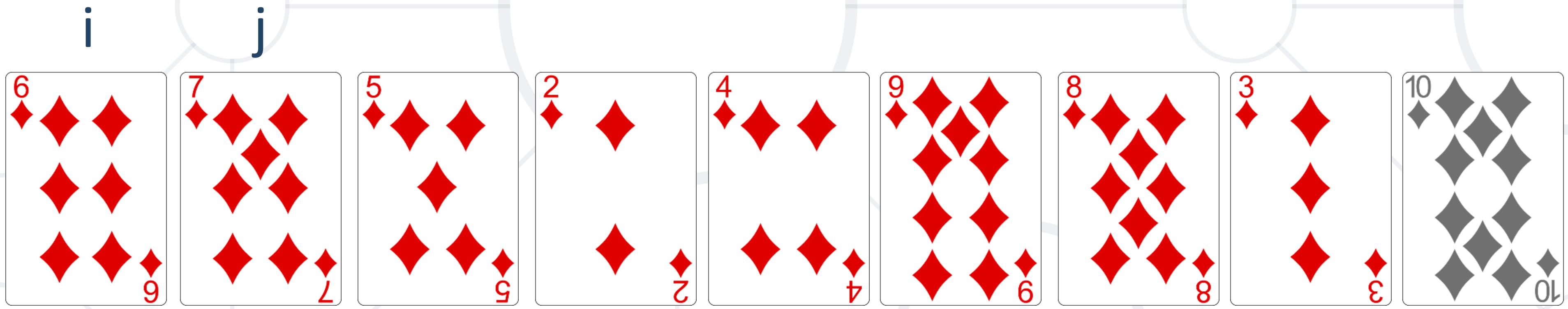
Bubble Sort Visualization (7)



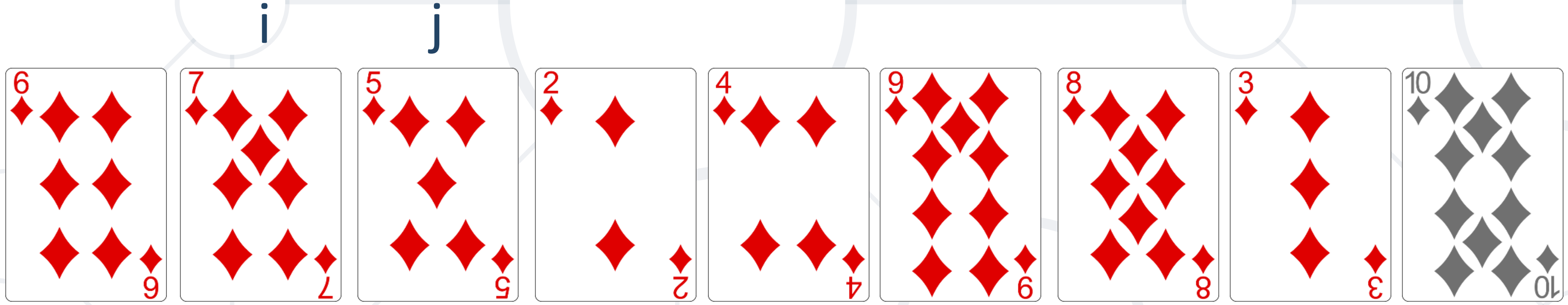
Bubble Sort Visualization (8)



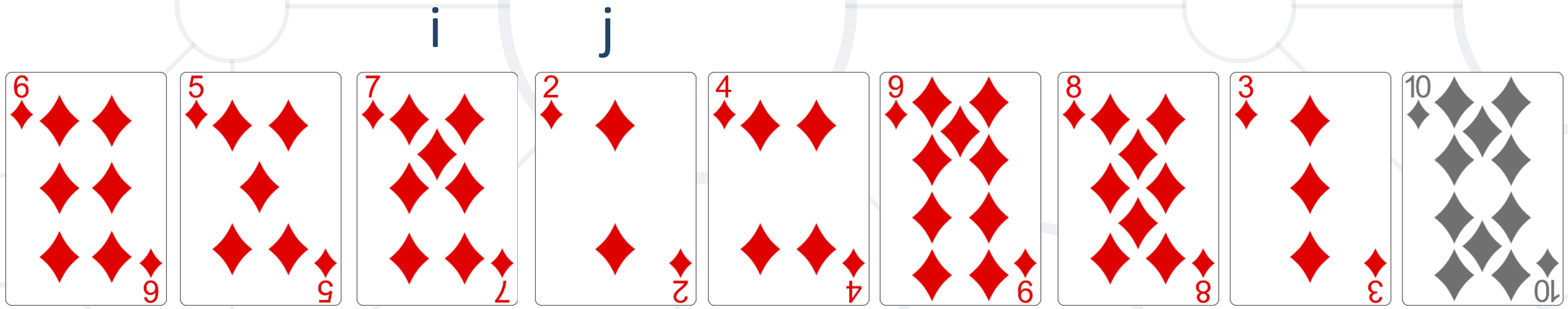
Bubble Sort Visualization (9)



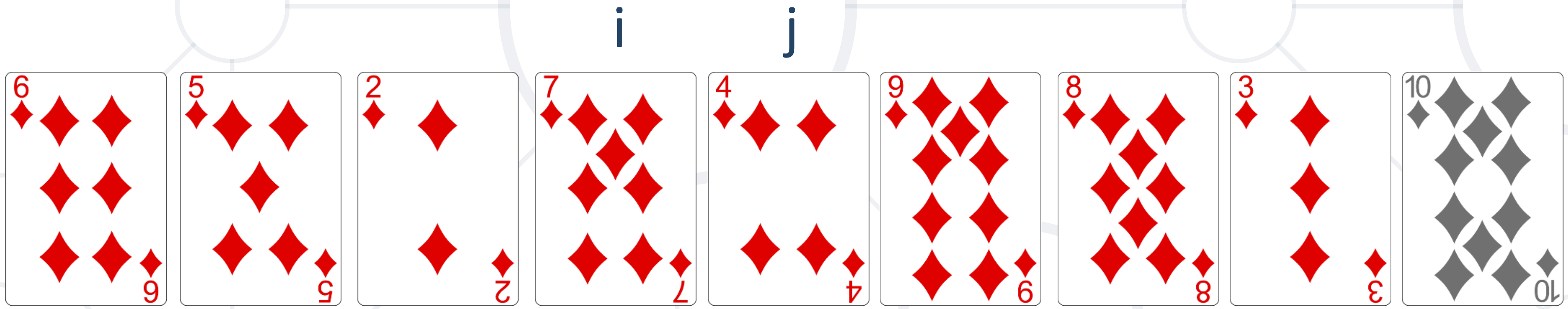
Bubble Sort Visualization (10)



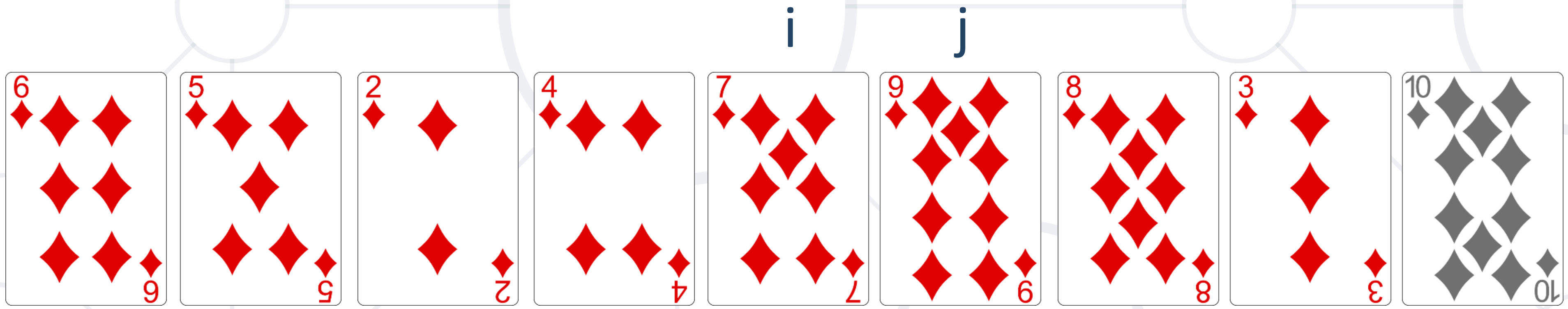
Bubble Sort Visualization (11)



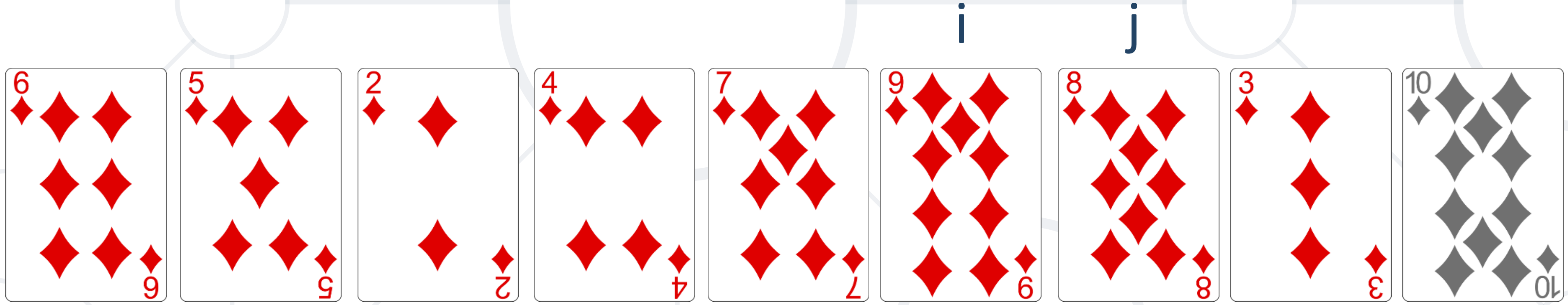
Bubble Sort Visualization (12)



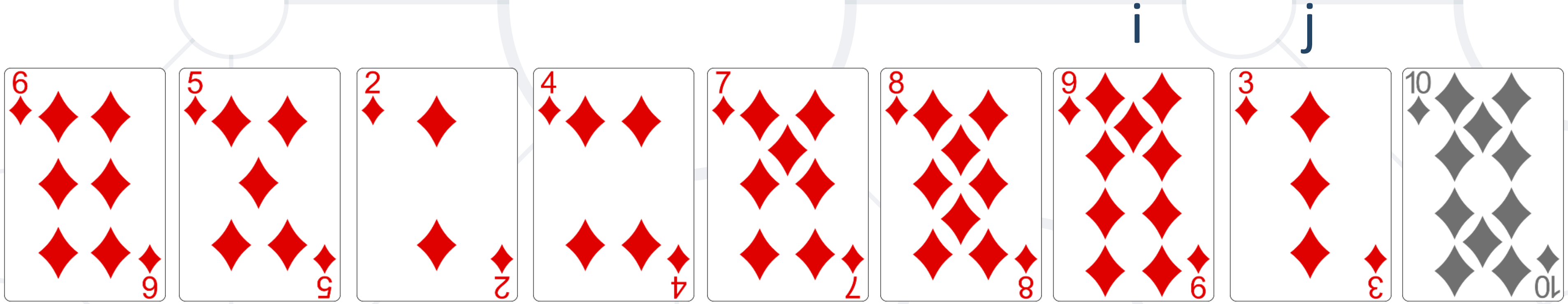
Bubble Sort Visualization (13)



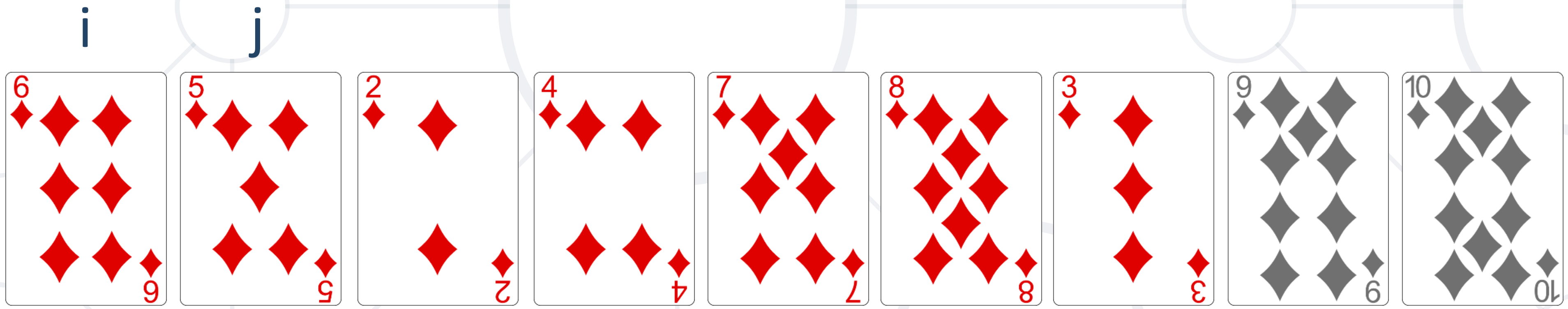
Bubble Sort Visualization (14)



Bubble Sort Visualization (15)



Bubble Sort Visualization (16)



BubbleSort (1)

```
nums = [1, 3, 4, 2, 5, 6]
for i in range(len(nums)):
    for j in range(1, len(nums) - i):
        if nums[j - 1] > nums[j]:
            nums[j], nums[j - 1] = nums[j - 1], nums[j]
```

Bubble Sort (2)

```
nums = [1, 3, 4, 2, 5, 6]
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
```

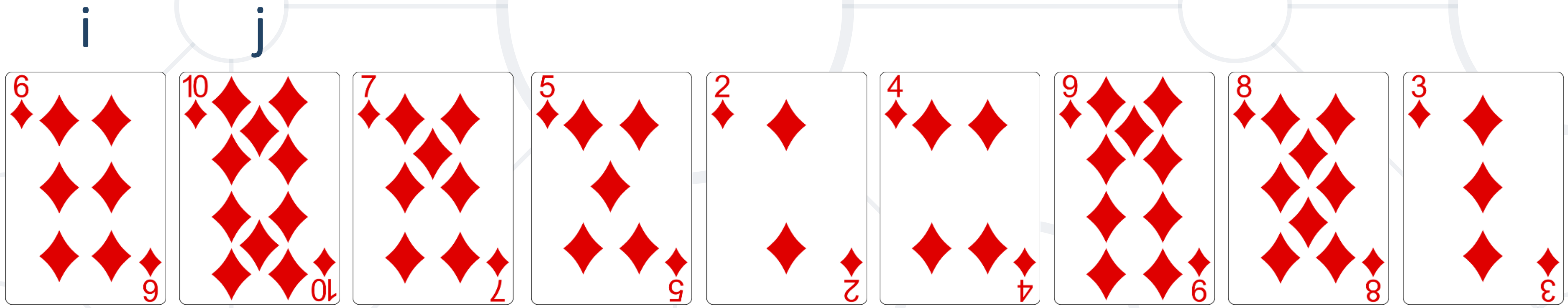
Comparison of Sorting Algorithms

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

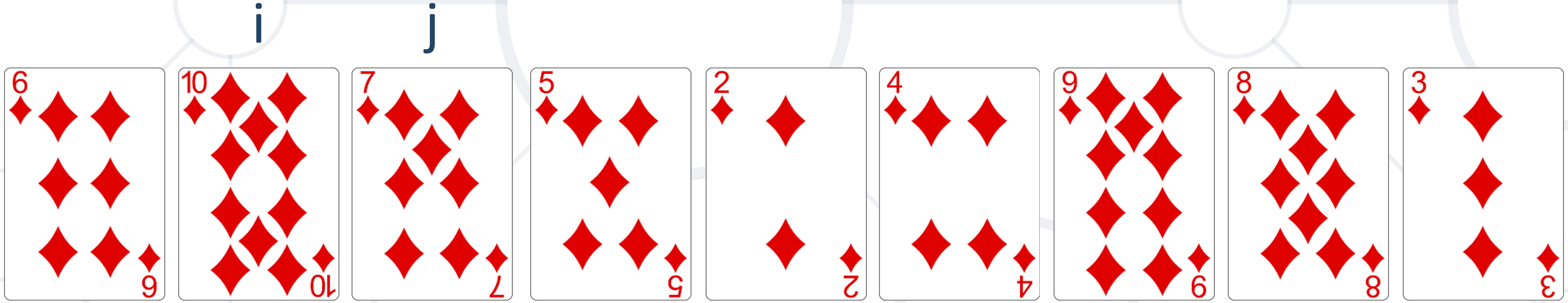
Insertion Sort

- **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**

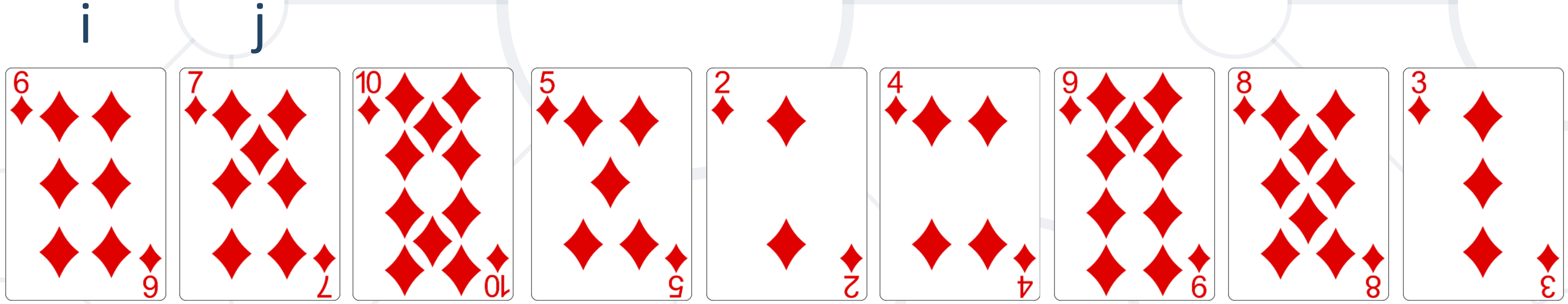
Insertion Sort Visualization (1)



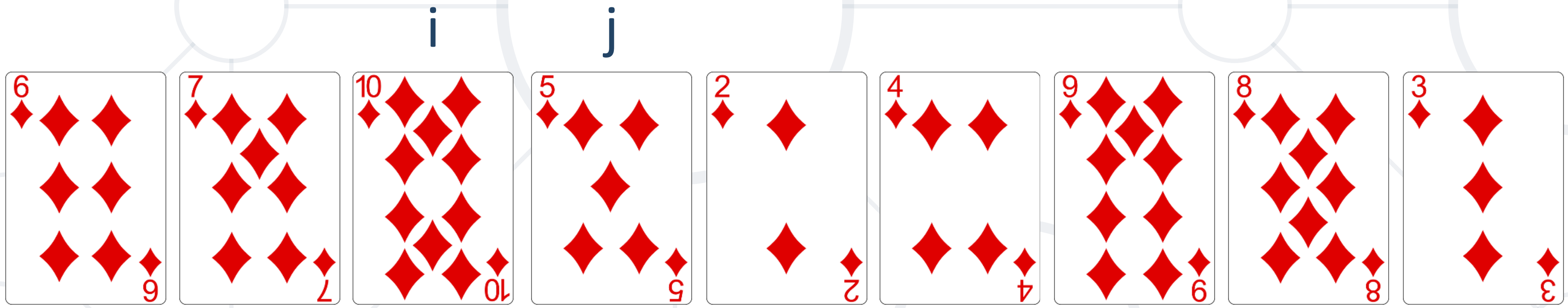
Insertion Sort Visualization (2)



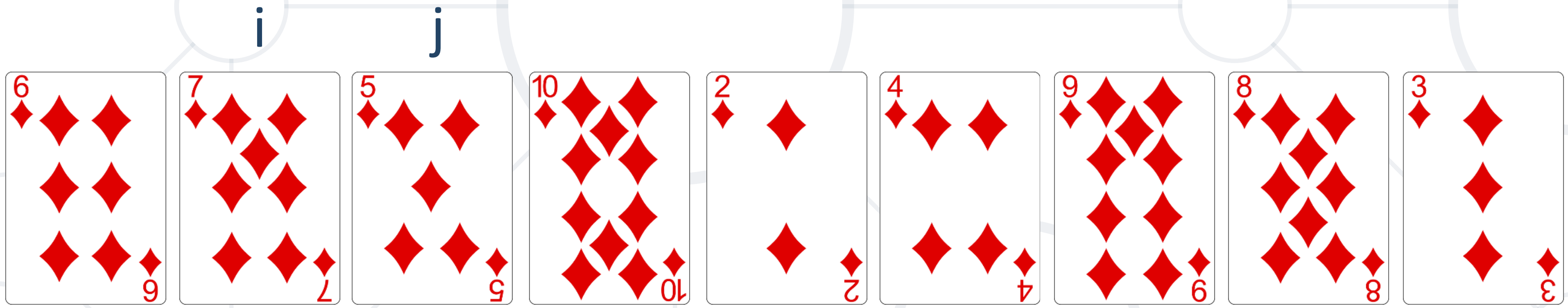
Insertion Sort Visualization (3)



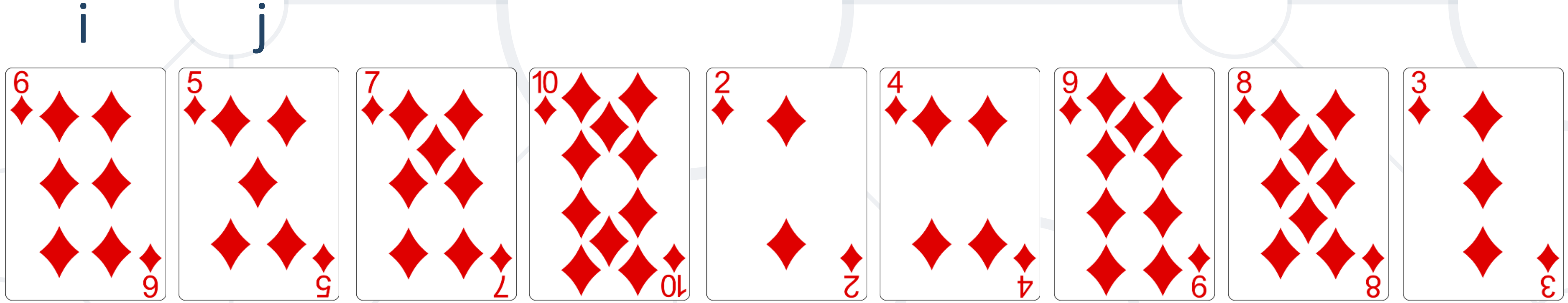
Insertion Sort Visualization (4)



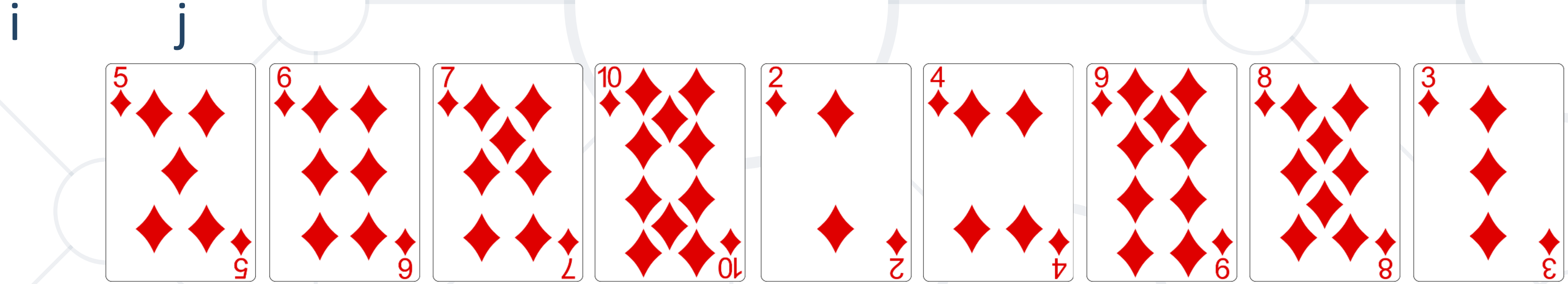
Insertion Sort Visualization (5)



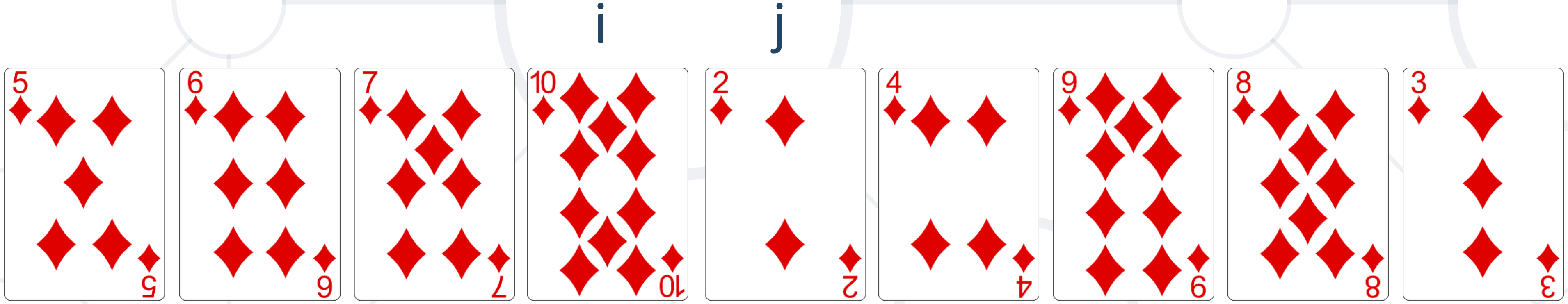
Insertion Sort Visualization (6)



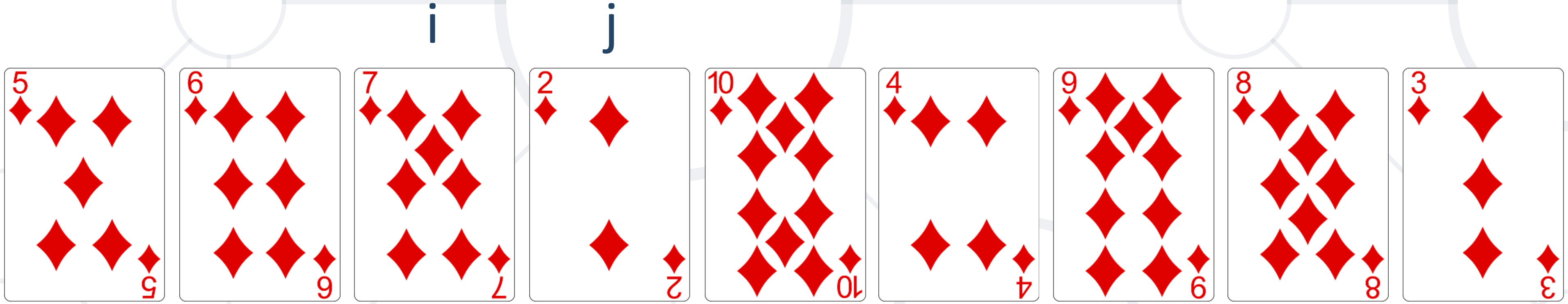
Insertion Sort Visualization (7)



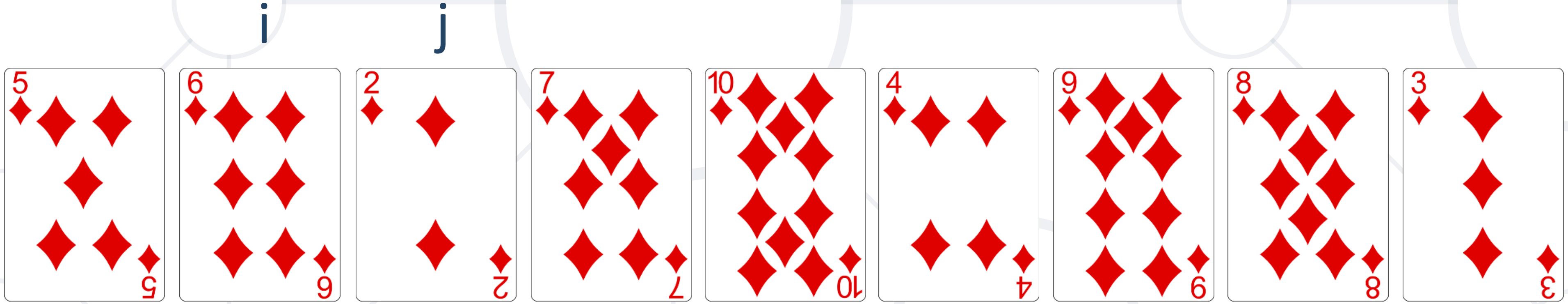
Insertion Sort Visualization (8)



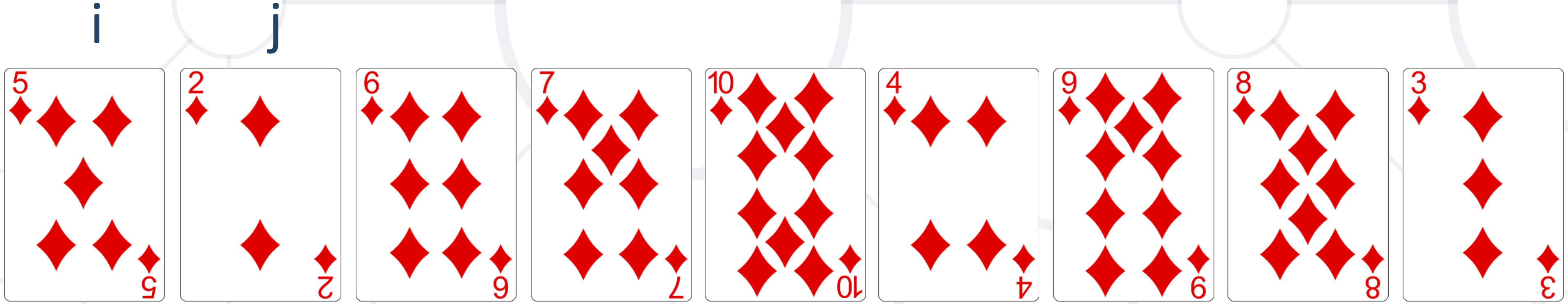
Insertion Sort Visualization (9)



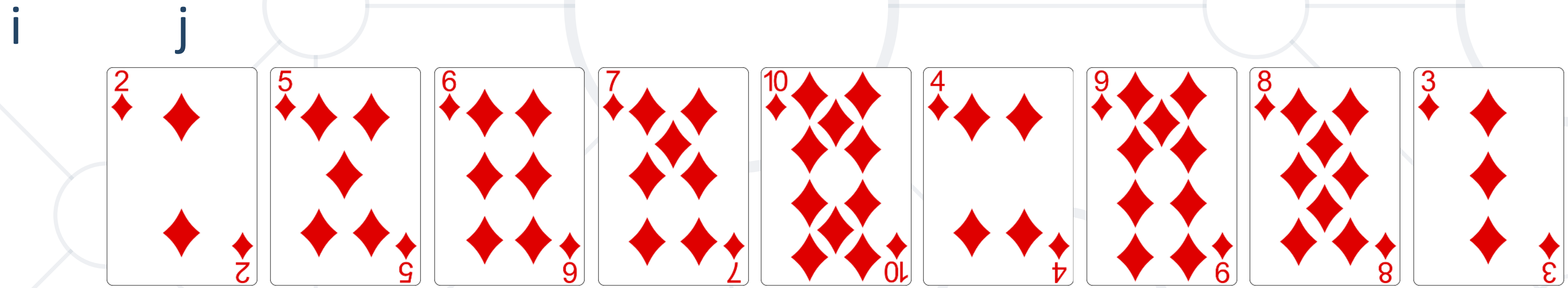
Insertion Sort Visualization (10)



Insertion Sort Visualization (11)



Insertion Sort Visualization (12)



Insertion Sort

```
for i in range(len(nums)):
    j = i
    while j > 0 and nums[j] < nums[j - 1]:
        nums[j], nums[j - 1] = nums[j - 1], nums[j]
        j -= 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

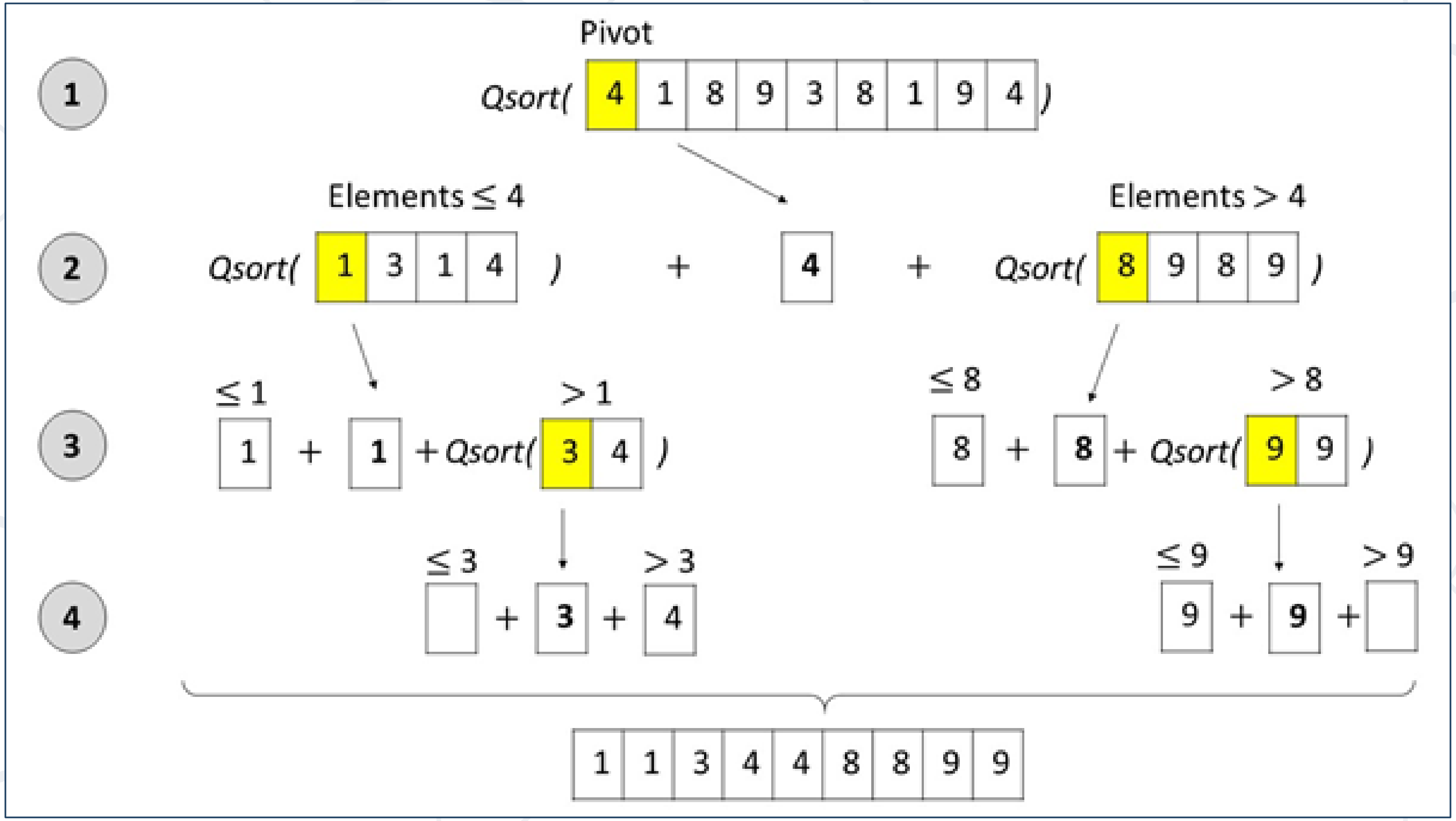
Advanced Sorting Algorithms

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: **$O(n^2)$**
    - When the pivot element divides the array into two **unbalanced sub-arrays** (huge difference in size)
  - Stable: **Depends**
  - Method: **Partitioning**

Quick Sort: Conceptual Overview



Quick Sort

```
def quick_sort(nums, start, end):
    if start >= end:
        return
    pivot, left, right = start, start + 1, end
    while left <= right:
        if nums[left] > nums[pivot] > nums[right]:
            nums[left], nums[right] = nums[right], nums[left]
        if nums[left] <= nums[pivot]:
            left += 1
        if nums[right] >= nums[pivot]:
            right -= 1
    nums[pivot], nums[right] = nums[right], nums[pivot]
    quick_sort(nums, start, right - 1)
    quick_sort(nums, right + 1, end)
```

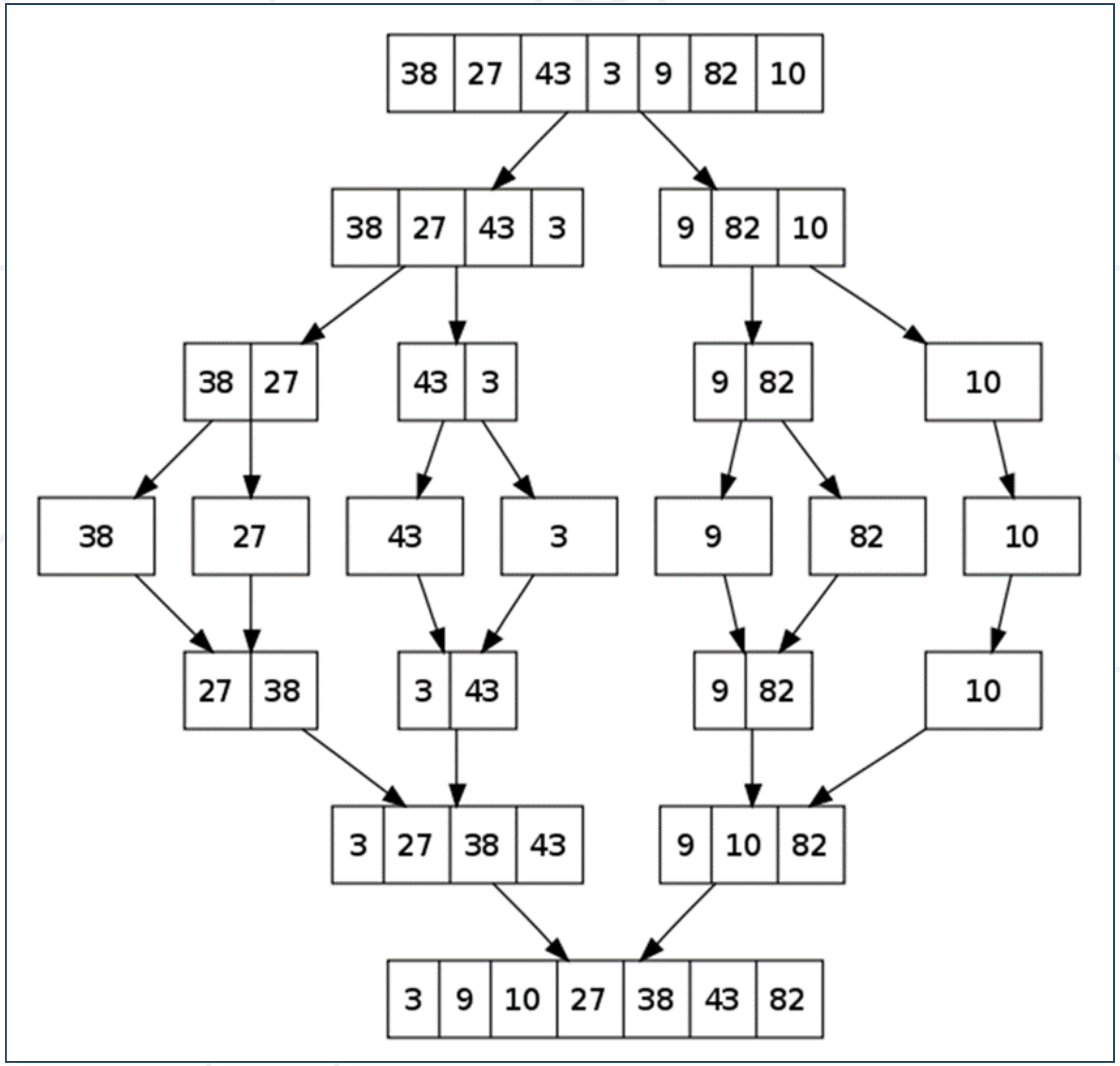
Comparison of Sorting Algorithms

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	n * log(n)	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



38 27 43 3 9 82 10

38 27 43 3 | 9 82 10

38 27 | 43 3 | 9 82 | 10

38 | 27 | 43 | 3 | 9 | 82 | 10

27 38 | 3 43 | 9 82 | 10

3 27 38 43 | 9 10 82

3 9 10 27 38 43 82

Merge Sort (1)

```
# Memory: O(n*log(n))
def merge_sort(nums):
    if len(nums) == 1:
        return nums
    mid_idx = len(nums) // 2
    left = nums[:mid_idx]
    right = nums[mid_idx:]
    return merge_arrays(merge_sort(left), merge_sort(right))
```

Merge Sort (2)

```
def merge_arrays(left, right):
    sorted_arr = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            sorted_arr.append(left[left_idx])
            left_idx += 1
        else:
            sorted_arr.append(right[right_idx])
            right_idx += 1
    # TODO: Take remaining elements either from the left or right
    return sorted_arr
```

Merge Sort (3)

```
while left_idx < len(left):
    sorted_arr.append(left[left_idx])
    left_idx += 1

while right_idx < len(right):
    sorted_arr.append(right[right_idx])
    right_idx += 1
```

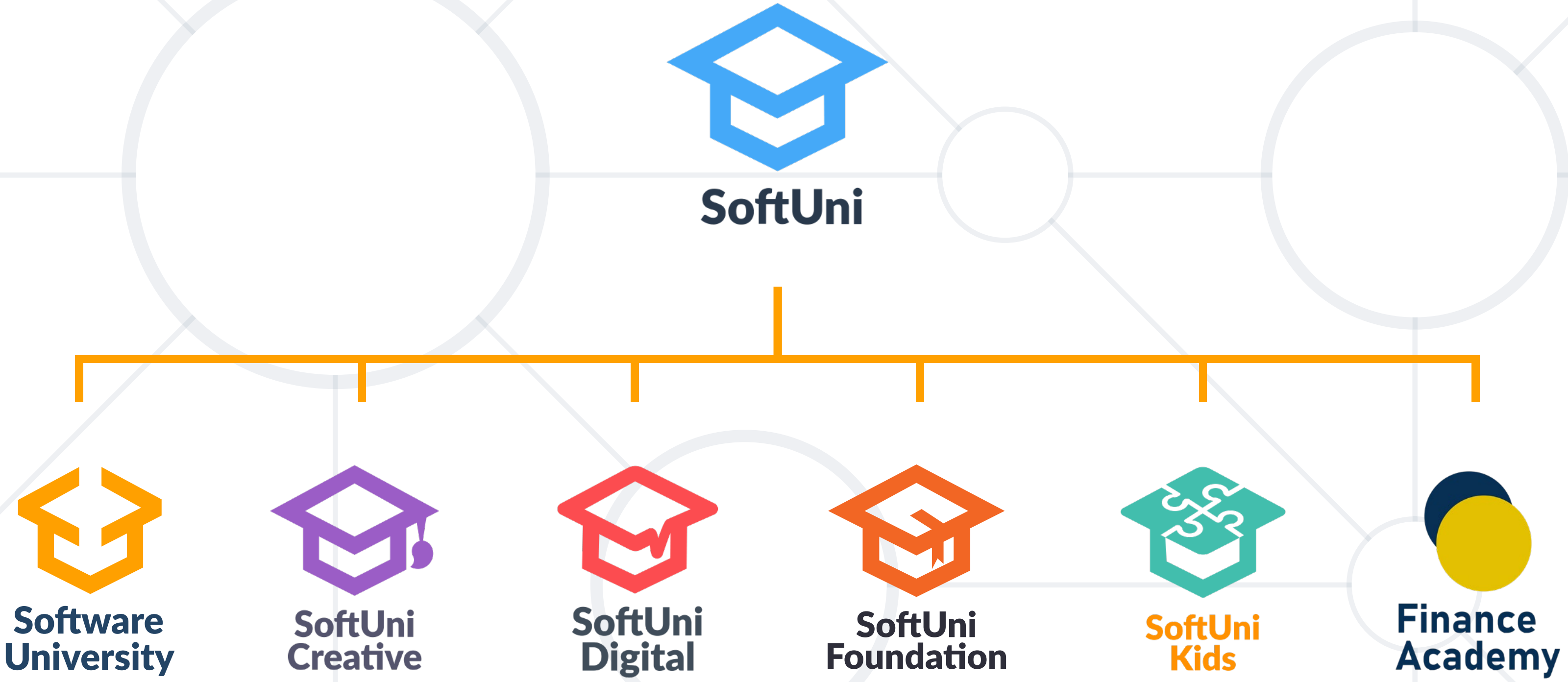
Comparison of Sorting Algorithms

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	n * log(n)	Depends	Partitioning
Merge	n * log(n)	n * log(n)	n * log(n)	n	Yes	Merging

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?

Questions?



SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

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