Design and Analysis of Algorithms

Algorithm Design Techniques

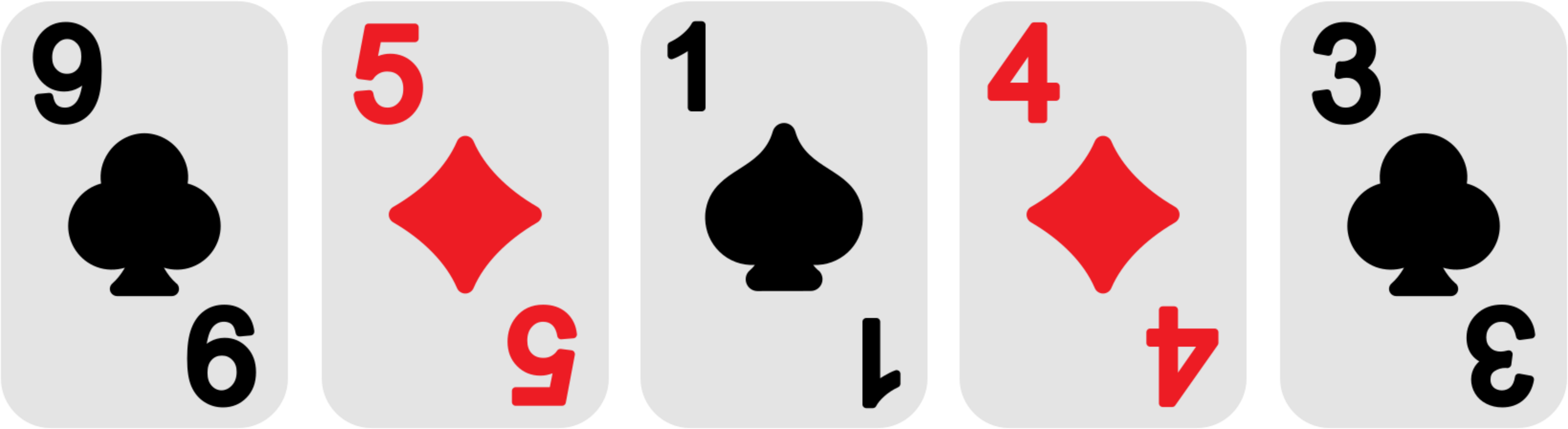
- **Brute Force and Exhaustive Search**
- Divide-and-Conquer
- **Decrease-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Sorting Algorithm

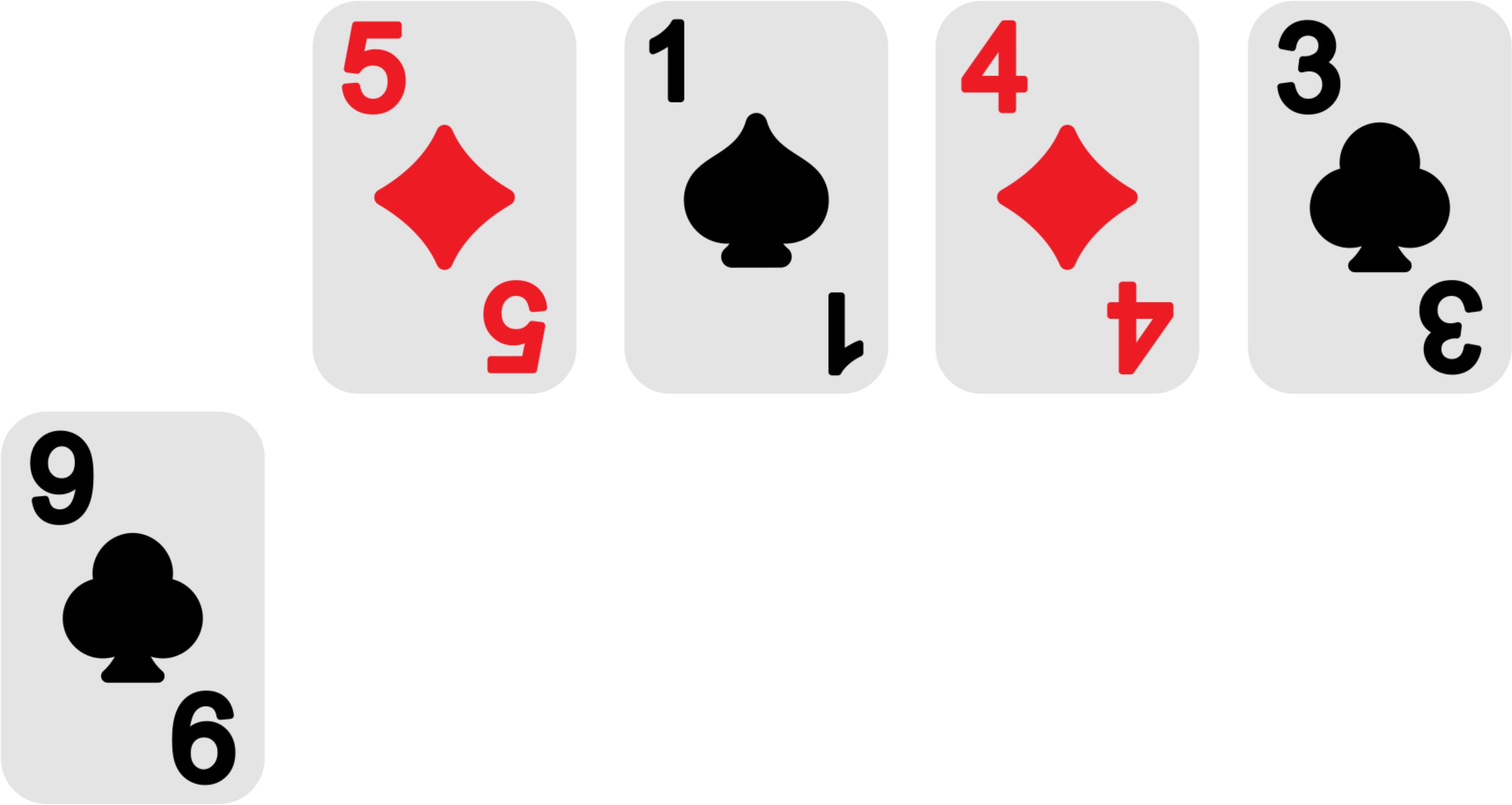
Brute Force

- Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- In this section, we consider the application of the brute-force approach to the problem of sorting.
- As we mentioned, many algorithms have been developed for solving this very important problem.
- Now, ask yourself a question: "What would be the most straightforward method for solving the sorting problem?"

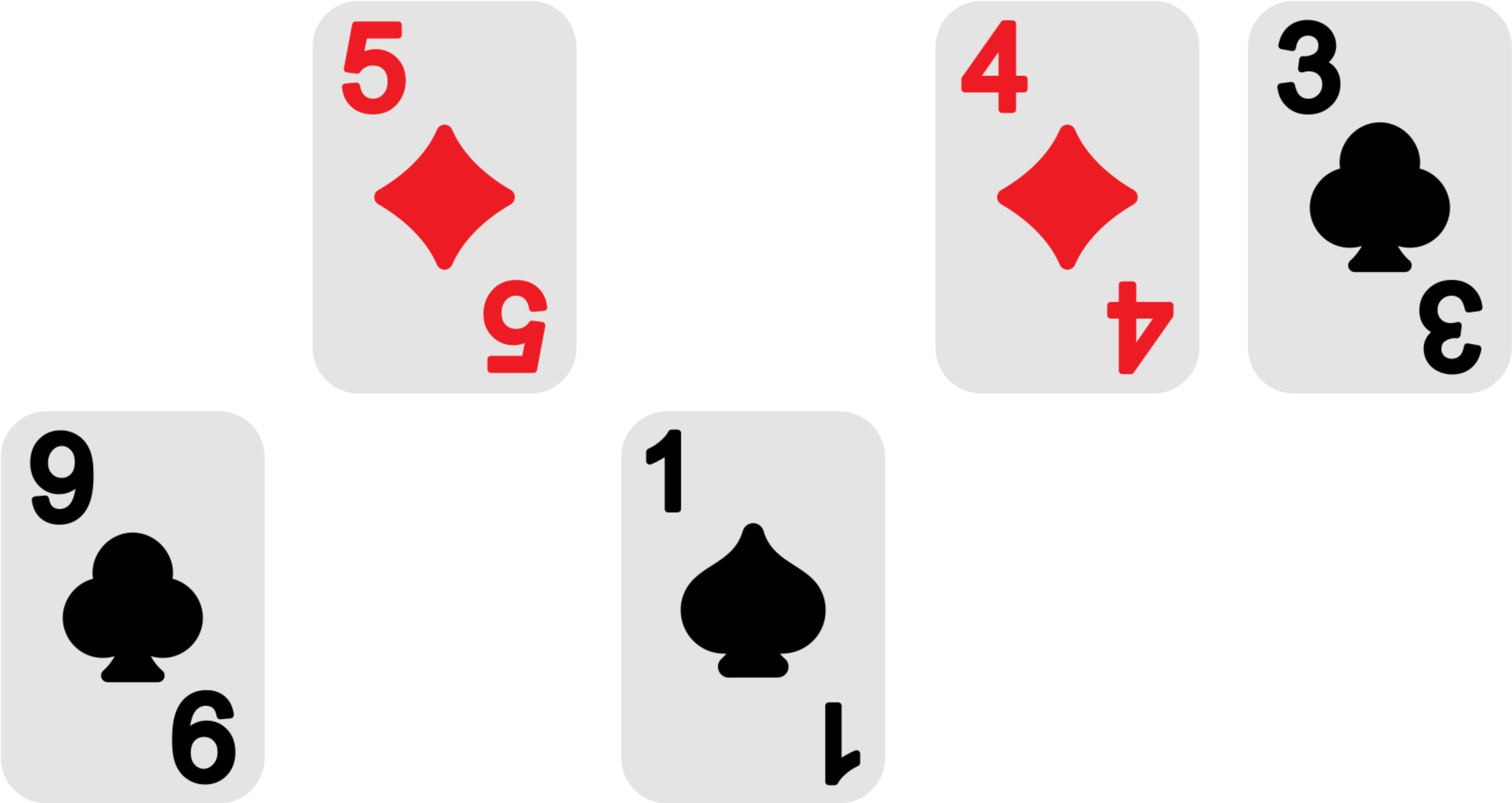
Sorting Playing Cards



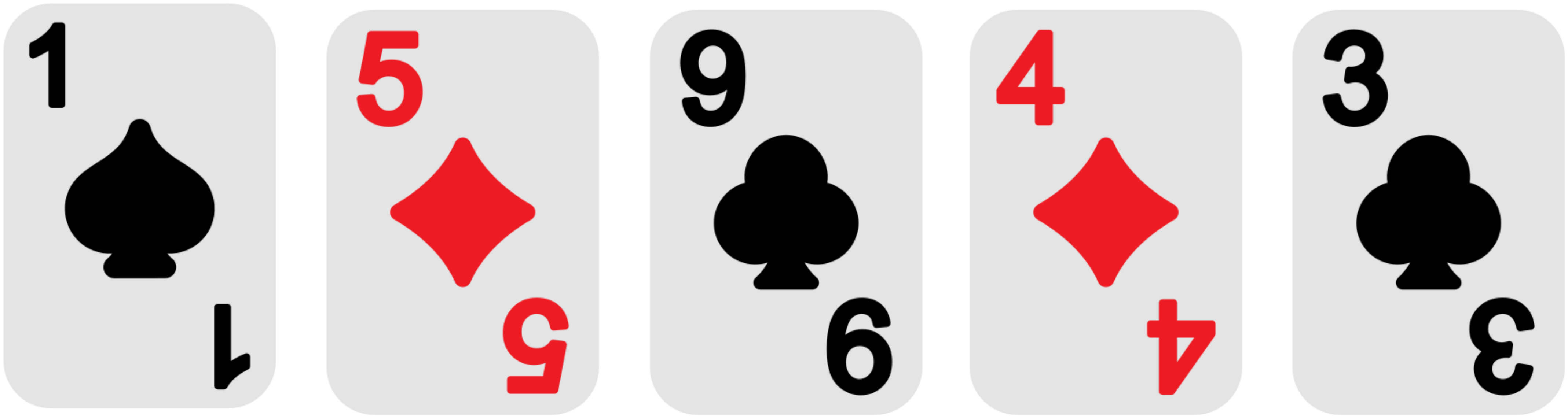
Sorting Playing Cards



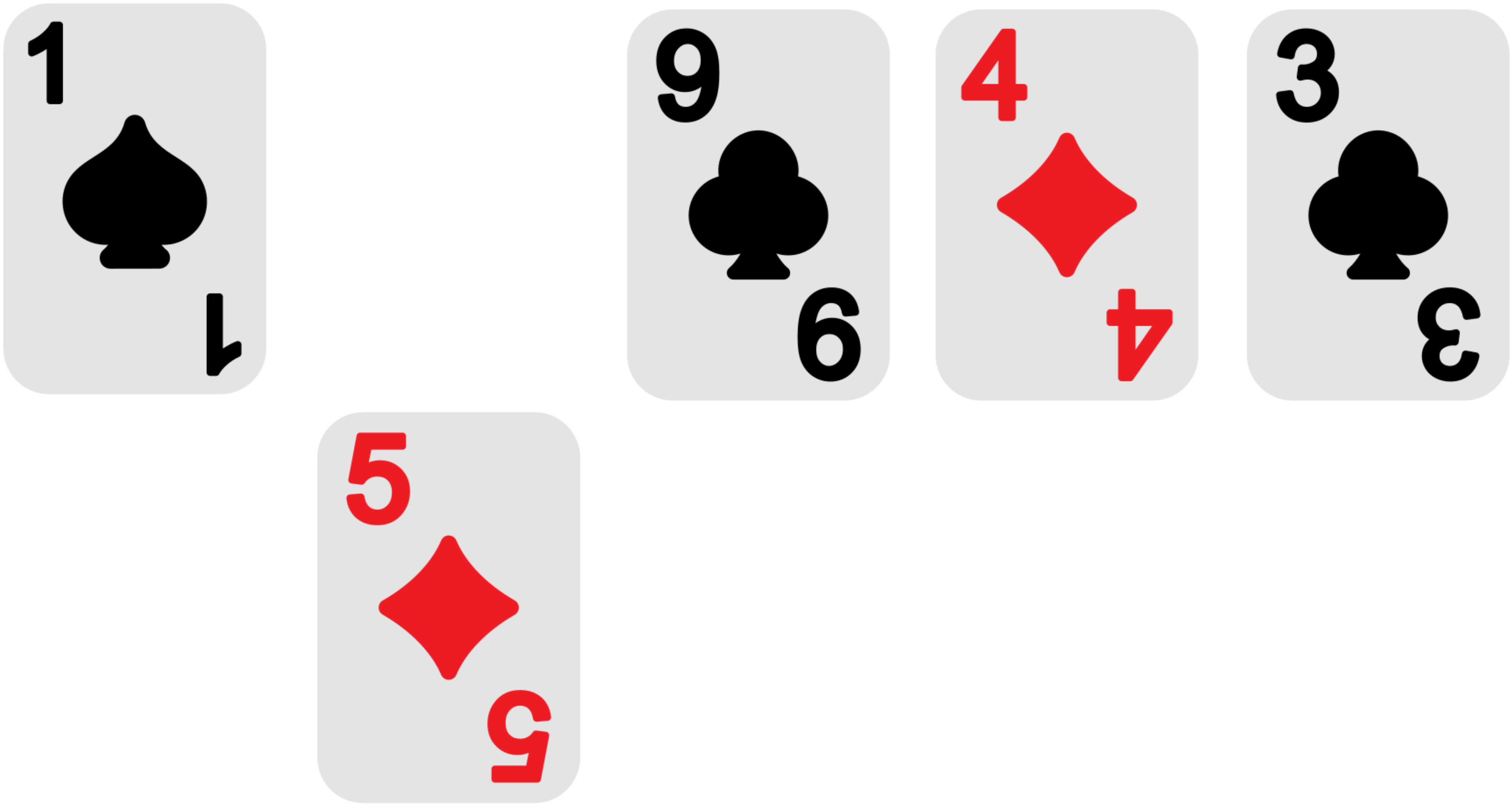
Sorting Playing Cards



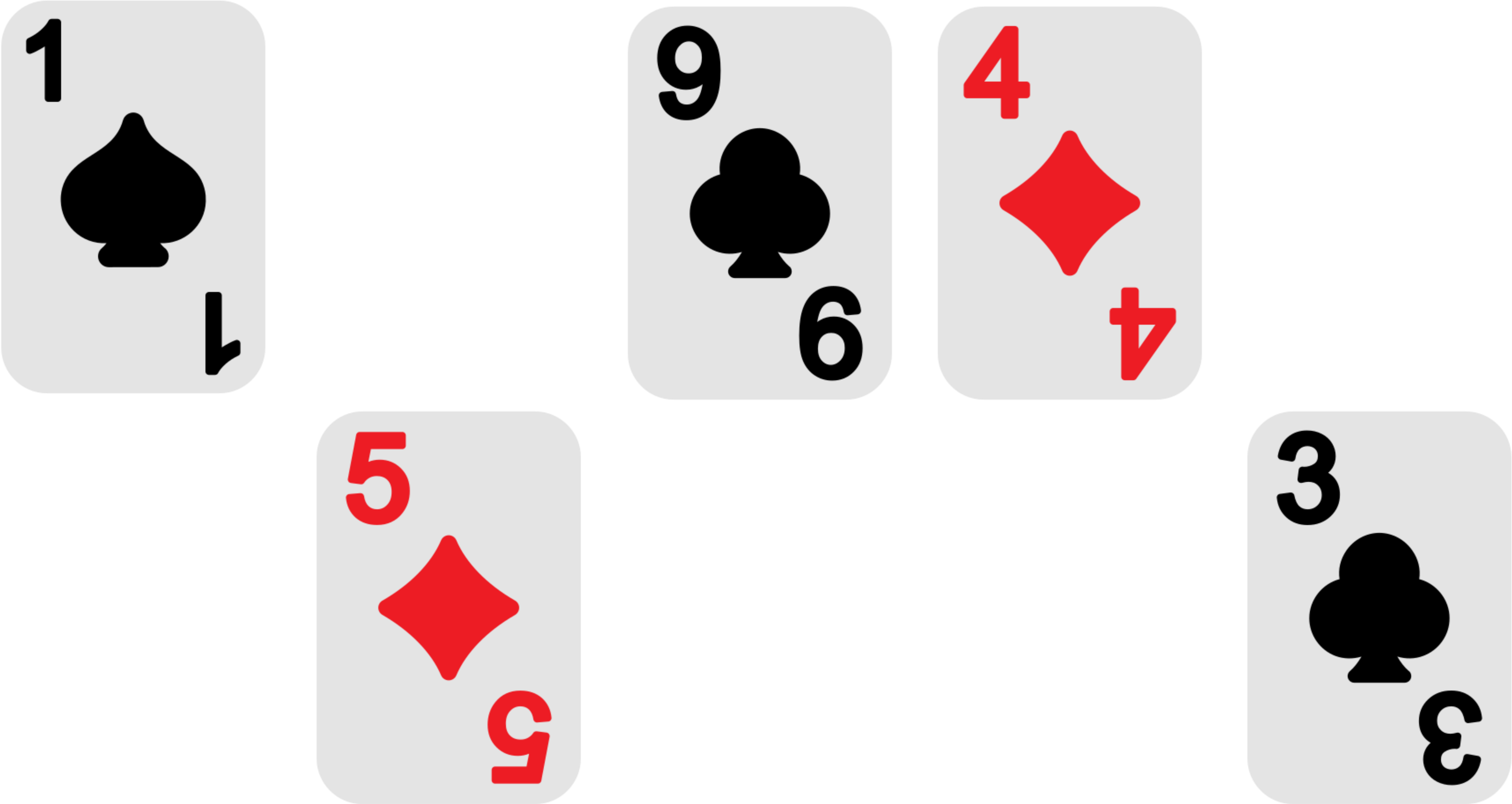
Sorting Playing Cards



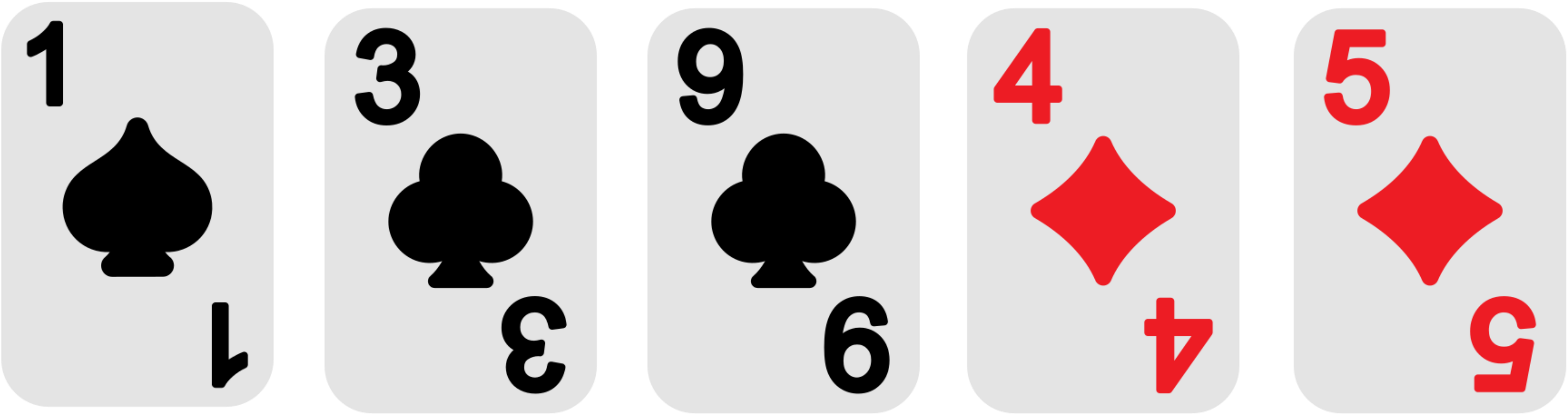
Sorting Playing Cards



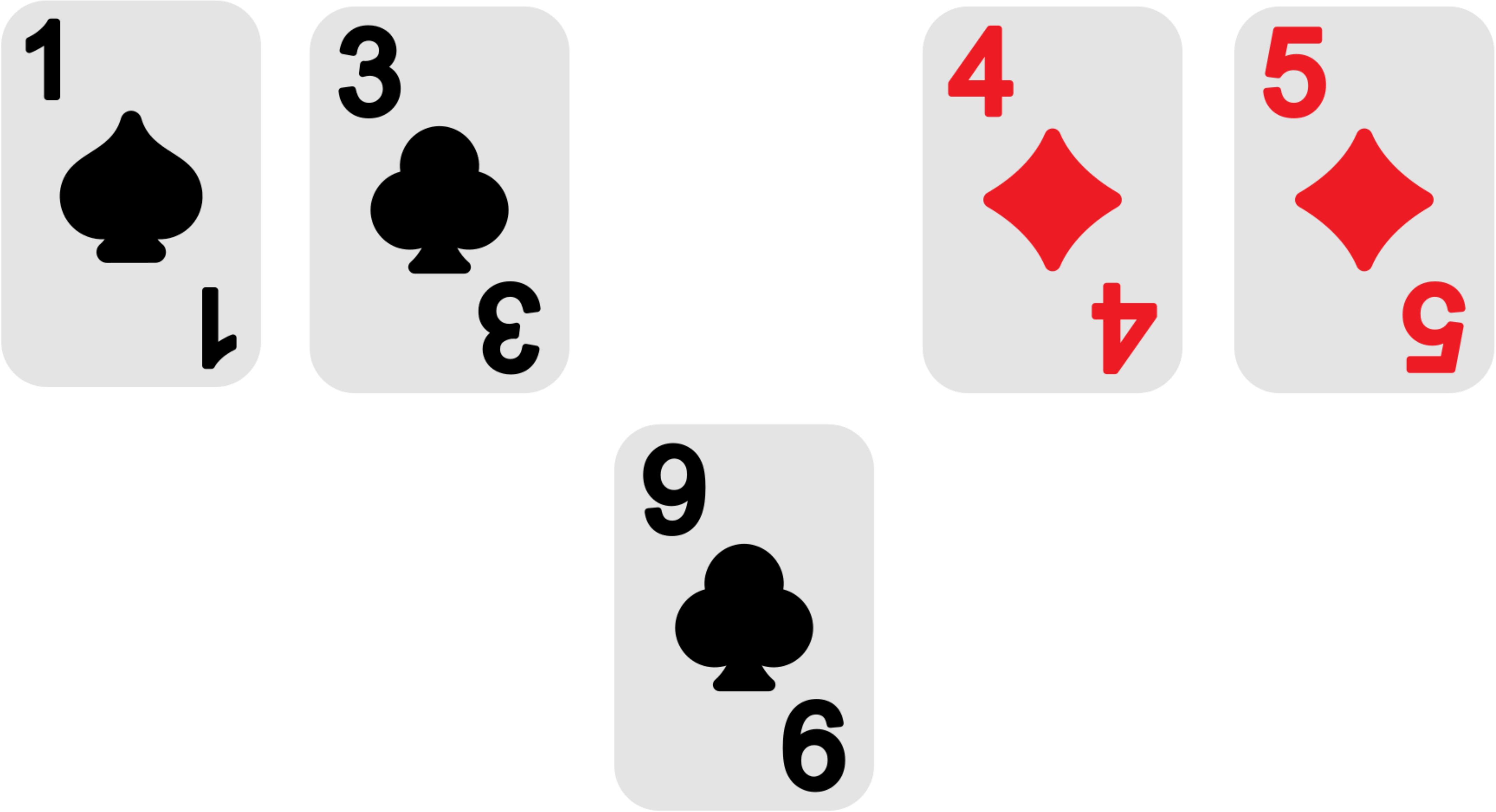
Sorting Playing Cards



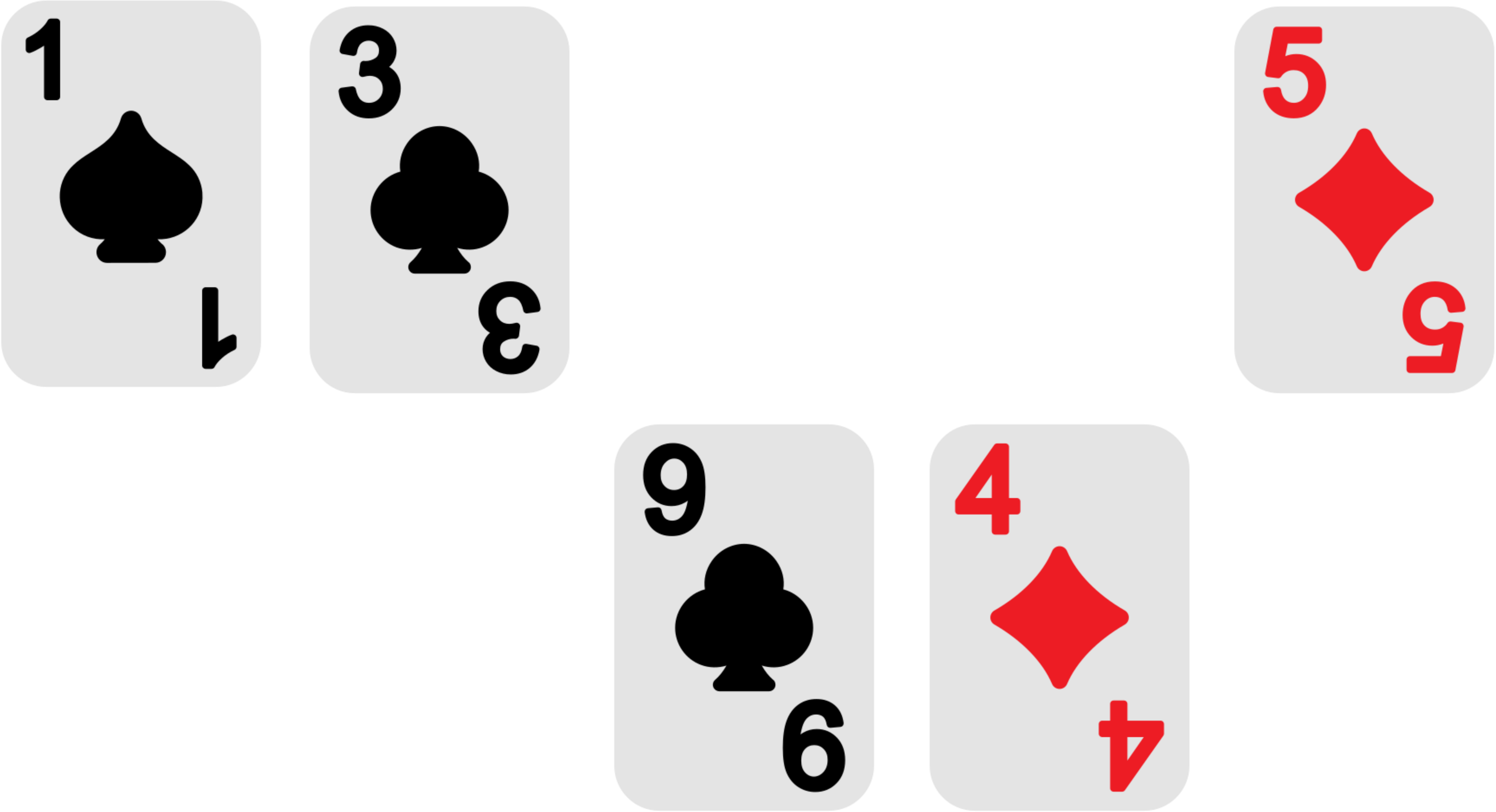
Sorting Playing Cards



Sorting Playing Cards

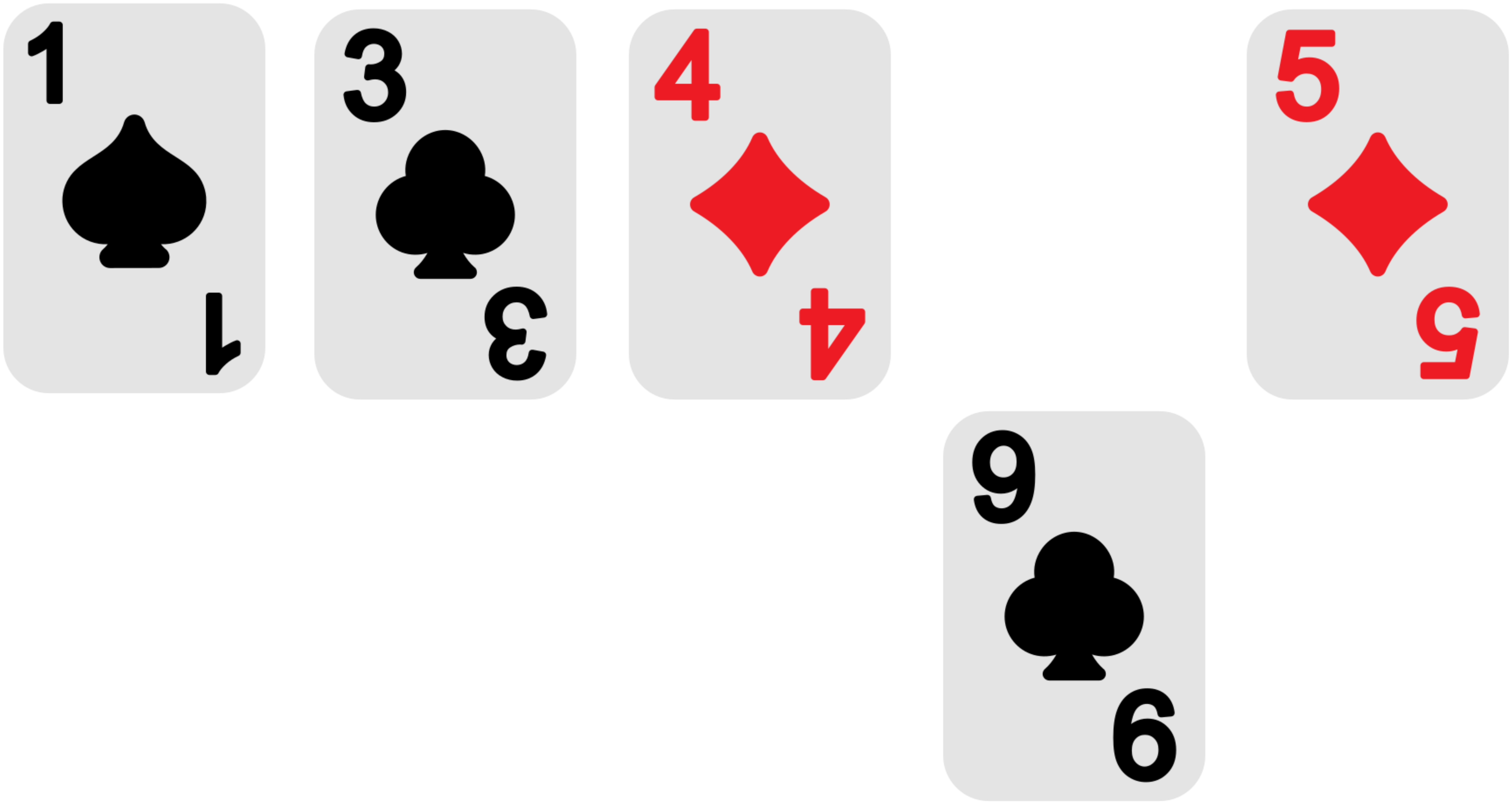


Sorting Playing Cards

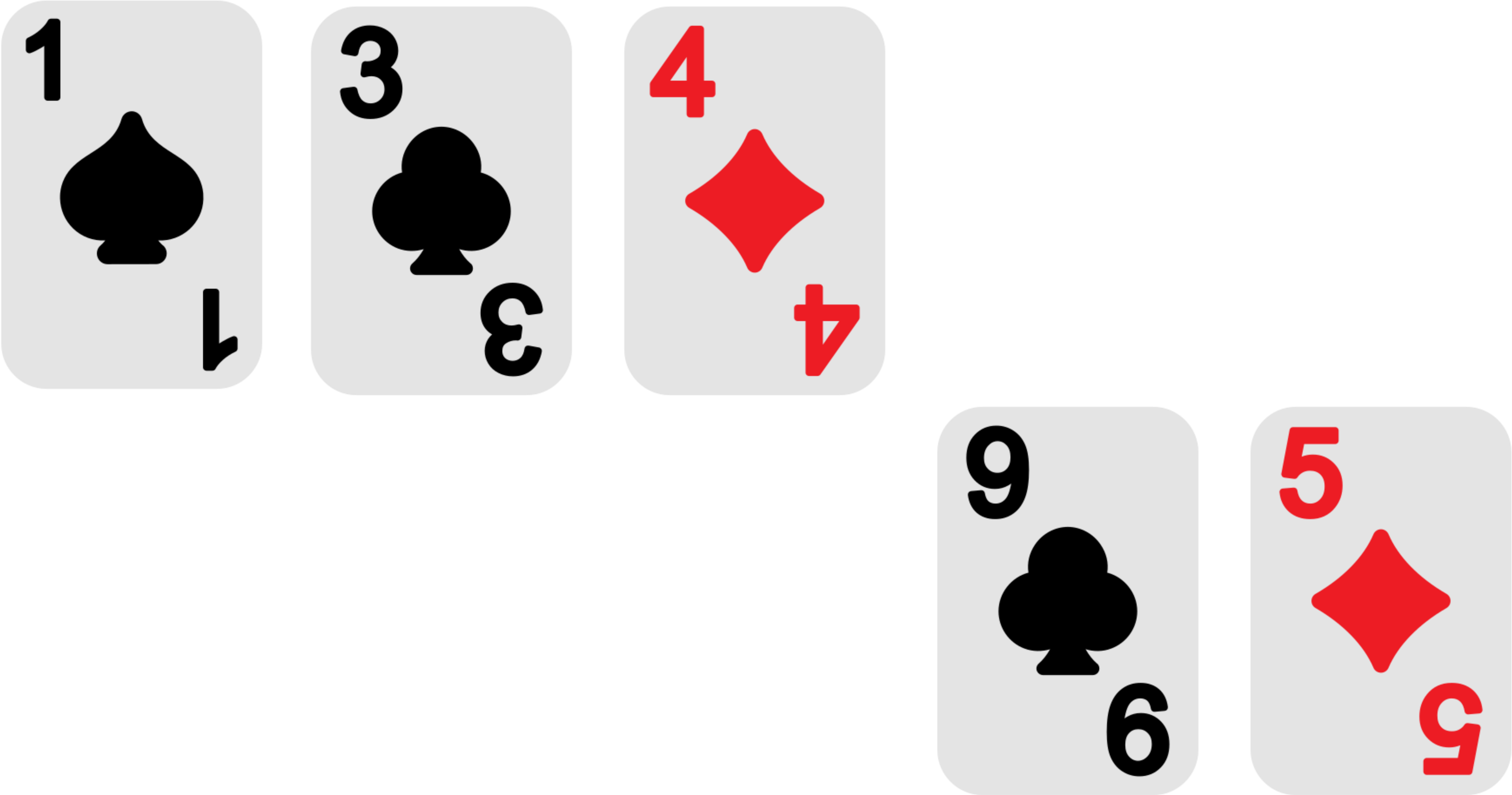


Sorting Playing Cards

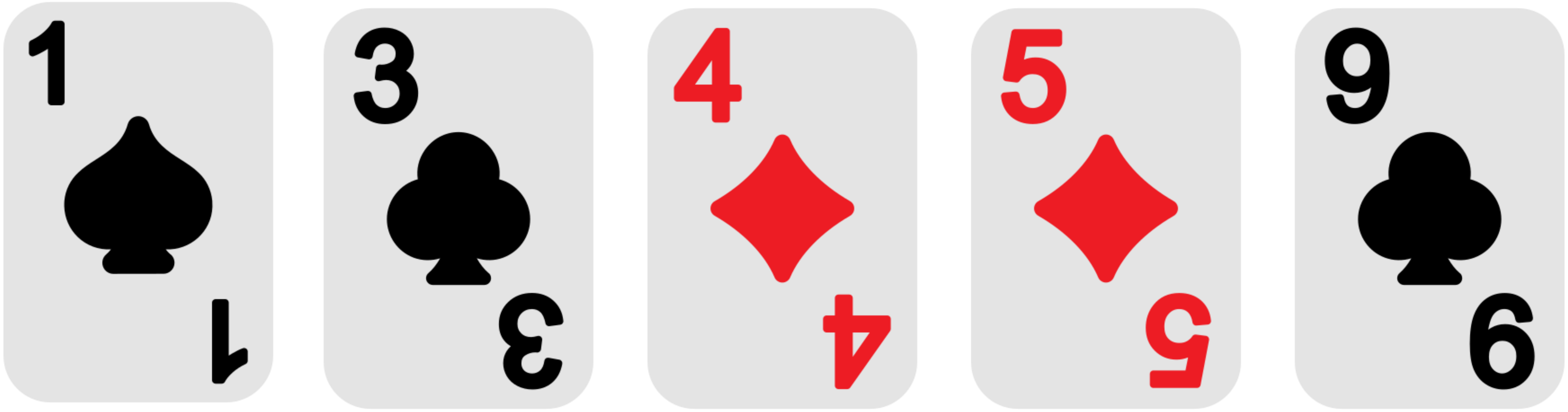
Sorting Playing Cards



Sorting Playing Cards



Sorting Playing Cards



Selection Sort: Pseudocode

ALGORITHM *SelectionSort*(A[0..$n-1$])

//Sorts a given array by selection sort
//Input: An array A[0..$n-1$] of orderable elements
//Output: Array A[0..$n-1$] sorted in nondecreasing order

```
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]
            min ← j
    swap(A[i], A[min])
```

Selection Sort: Analysis

- The analysis of selection sort is straightforward.
- The input size is given by the number of elements *n*.
- The basic operation is the key comparison.
- The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n-1)n}{2}$$

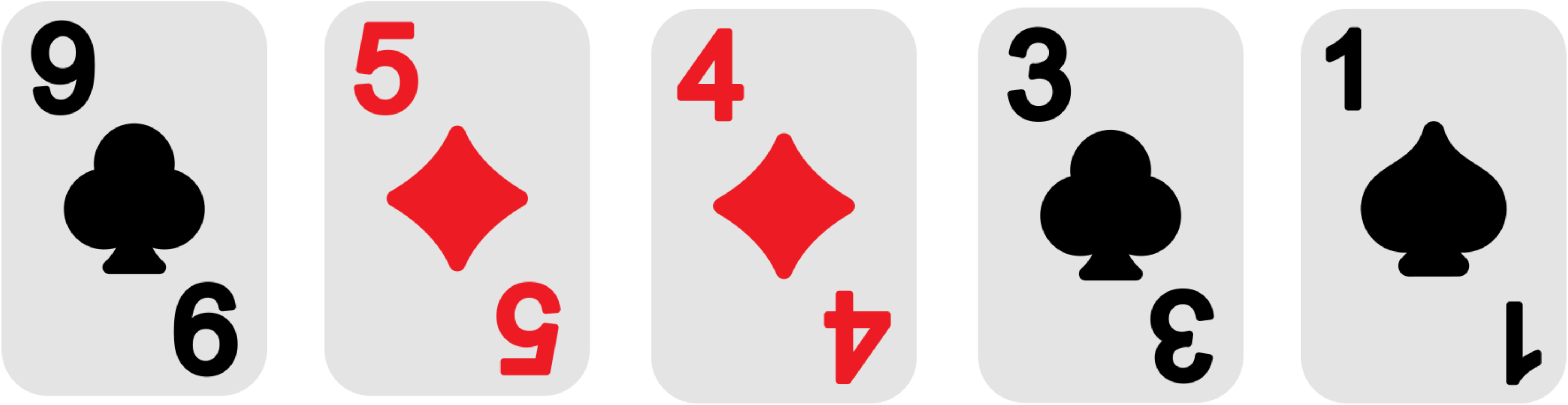
Working of Selection Sort

Sort 6, 4, 1, 8, 5

6		4		**1**		8		5
1	|	**4**		6		8		5
1		4	|	**6**		8		**5**
1		4		5	|	**8**		**6**
1		4		5		6	|	8

Selection Sort: Best Case

Selection Sort: Worst Case



Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	$O(1)$
Stability	No

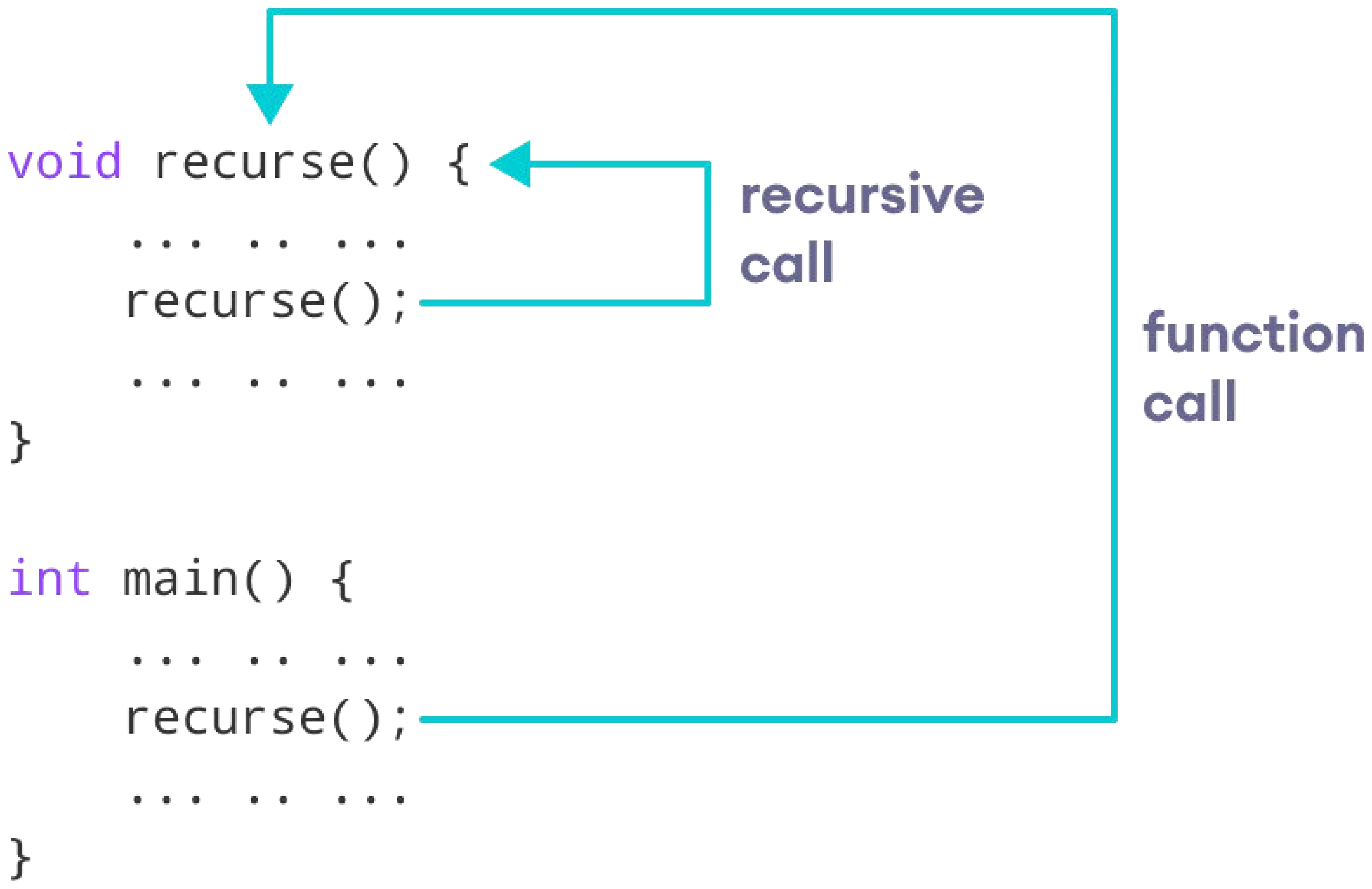
Sorting Algorithms

- Bubble Sort
- **Selection Sort**
- **Insertion Sort**
- Merge Sort
- Quicksort
- Counting Sort
- Radix Sort
- Bucket Sort
- Heap Sort
- Shell Sort

Recursion

- Recursion is a coding technique used in many algorithms.
- A recursive function is a function that calls itself.
- A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

Recursion


void recurse() {
... .
recurse();
... .
}
int main() {
... .
recurse();
... .
}
recursive call
function call

Base Case and Recursive Case

- Because a recursive function calls itself, it’s easy to write a function incorrectly that ends up in an infinite loop.
- When you write a recursive function, you have to tell it when to stop recursing.
- That’s why every recursive function has two parts: the base case, and the recursive case.
- The recursive case is when the function calls itself.
- The base case is when the function doesn’t call itself again, so it doesn’t go into an infinite loop.

Sum of Integers

ALGORITHM $Sum(n)$

//Computes the sum of integers from 1 to n recursively

//Input: A nonnegative integer n

//Output: The sum of integers from 1 to n

if $n = 0$

  return 0

else

  return $Sum(n - 1) + n$

Factorial

- In mathematics, the notation $n!$ represents the factorial of the nonnegative integer n.
- The factorial of n is the product of all the integers from 1 to n.

$$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \ldots \times 2 \times 1$$

$$n! = n \times (n - 1)!$$

- For example,

  $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$

  $4! = 4 \times 3 \times 2 \times 1 = 24$

  $0! = 1$

Factorial

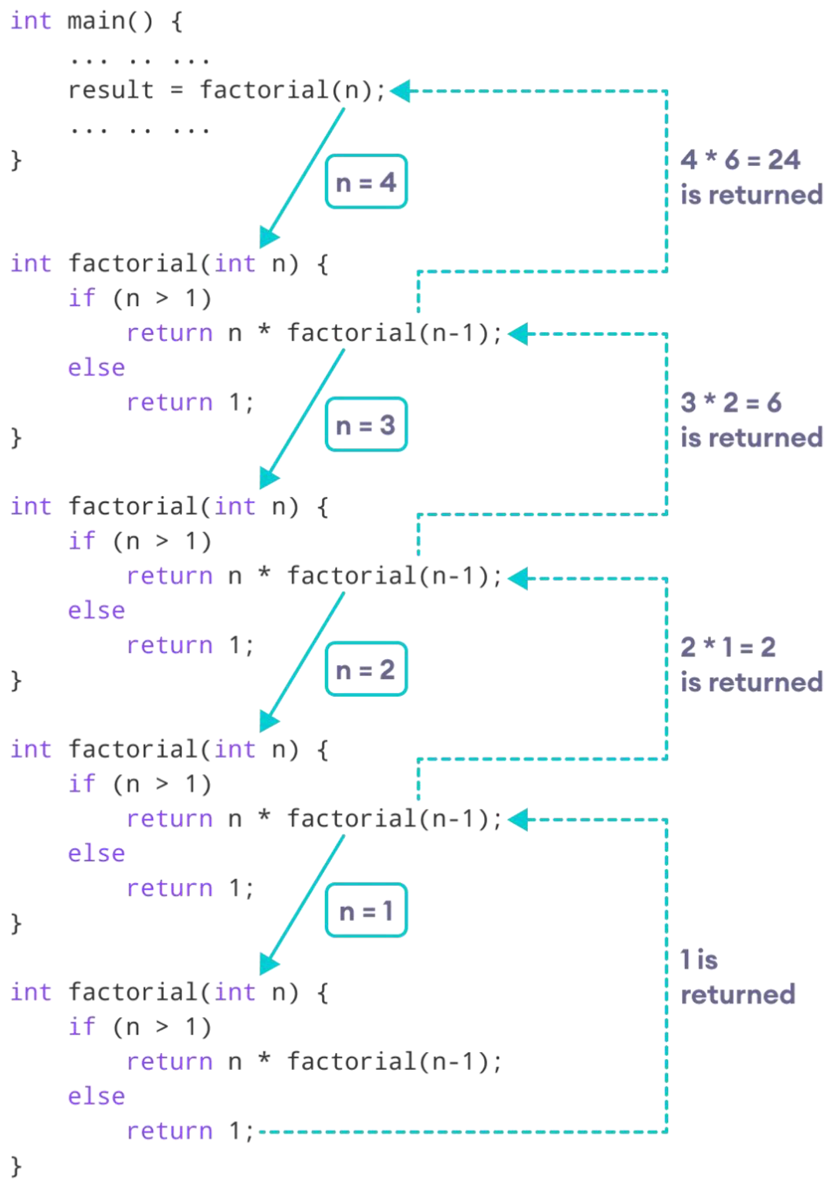
By definition, we can compute $F(n) = F(n-1).n$ with the following recursive algorithm.

ALGORITHM F(n)

//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!

if $n = 0$
    return 1 // 0! = 1
else
    return $F(n-1) * n$ // $n! = n \times (n-1)!$

Factorial


int main() {
... .
result = factorial(n);
... .
}
n = 4
4 * 6 = 24
is returned
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 3
3 * 2 = 6
is returned
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 2
2 * 1 = 2
is returned
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 1
1 is
returned
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}