Asymptotic Complexity Sorting

CS143: lecture 11

Konstantinos Ameranis, July 16

Asymptotic Complexity

Asymptotic Complexity

Asymptotic Complexity

- In two papers Konstantinos found two algorithms for the same problem

Asymptotic Complexity

- In two papers Konstantinos found two algorithms for the same problem
- How do we compare which one is faster?

Asymptotic Complexity

- In two papers Konstantinos found two algorithms for the same problem
- How do we compare which one is faster?
- One idea is to compare which one runs faster on various inputs

Asymptotic Complexity

- In two papers Konstantinos found two algorithms for the same problem
- How do we compare which one is faster?
- One idea is to compare which one runs faster on various inputs
- Here are the timings reported in the papers:

Asymptotic Complexity

- In two papers Konstantinos found two algorithms for the same problem
- How do we compare which one is faster?
- One idea is to compare which one runs faster on various inputs
- Here are the timings reported in the papers:

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- For small inputs Algorithm A seems better

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- For small inputs Algorithm A seems better
- But for larger inputs Algorithm B seems to achieve better times

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- For small inputs Algorithm A seems better
- But for larger inputs Algorithm B seems to achieve better times
- Except the last input where Algorithm A finishes much much faster

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Reading further in the papers reveal more information

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Reading further in the papers reveal more information
- Algorithm A is implemented in C and Algorithm B is implemented in Python

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Reading further in the papers reveal more information
- Algorithm A is implemented in C and Algorithm B is implemented in Python
- Algorithm B was run in a computing cluster on an Intel Xeon CPU and Algorithm A was run in a Pentium 4 desktop CPU

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Reading further in the papers reveal more information
- Algorithm A is implemented in C and Algorithm B is implemented in Python
- Algorithm B was run in a computing cluster on an Intel Xeon CPU and Algorithm A was run in a Pentium 4 desktop CPU
- The last input is actually a special case where Algorithm A bypasses most of the work needed

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- It depends

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- It depends
  - Implementation

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- It depends
  - Implementation
  - Architecture

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- It depends
  - Implementation
  - Architecture
  - Input

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

- Which one is better?
- It depends
  - Implementation
  - Architecture
  - Input
  - …

Size (n)	Algorithm A	Algorithm B
100	0.01s	0.12s
500	0.05s	0.22s
1,000	0.13s	0.55s
5,000	1.03s	1.12s
10,000	2.32s	2.18s
10,000	5.41s	3.18s
50,000	10.44s	6.27s
100,000	15.38s	10.46s
1,000,000	232.48s	103.72s
1,000,000	32.43s	108.37s

Asymptotic Complexity

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

+ Pros:

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

+ Pros:

  - Exact computation

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

➕ Pros:

  - Exact computation

❌ Cons:

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

➕ Pros:

  - Exact computation

❌ Cons:

  - Very time consuming to compute exactly

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

➕ Pros:

  - Exact computation

❌ Cons:

  - Very time consuming to compute exactly
  - For different sizes which one is better changes

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

➕ Pros:

  - Exact computation

❌ Cons:

  - Very time consuming to compute exactly
  - For different sizes which one is better changes

- For small inputs all algorithms will take similar times

Asymptotic Complexity

- A better idea:
  Count basic operations needed as a function of input in the worst case

➕ Pros:

  - Exact computation

❌ Cons:

  - Very time consuming to compute exactly
  - For different sizes which one is better changes
- For small inputs all algorithms will take similar times
- But for large inputs a linear and a quadratic algorithm will have very different run times

Asymptotic Complexity

Asymptotic Complexity

- Let $f(n) = 80n$ and $g(n) = 6n^2$

Asymptotic Complexity

- Let $f(n) = 80n$ and $g(n) = 6n^2$
- $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{80n}{6n^2} = \lim_{n \to \infty} \frac{80}{6n} = 0$$

Asymptotic Complexity

- Let $f(n) = 80n$ and $g(n) = 6n^2$
- $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{80n}{6n^2} = \lim_{n \to \infty} \frac{80}{6n} = 0$$
- Therefore as $n \to \infty$ $g(n)$ grows at least as fast as $f(n)$

Asymptotic Complexity

- Let $f(n) = 80n$ and $g(n) = 6n^2$
- $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{80n}{6n^2} = \lim_{n \to \infty} \frac{80}{6n} = 0$$
- Therefore as $n \to \infty$ $g(n)$ grows at least as fast as $f(n)$
- From this point on we will say that $f(n) \in O\left(g(n)\right)$

Asymptotic Complexity

Symbol	Limit definition	Constant definition	Relation	Name
$f(n) \in O\left(g(n)\right)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}$	$\exists\ n_0 \in \mathbb{N}, c \in \mathbb{R} : \forall\ n > n_0 \quad f(n) \leq c \cdot g(n)$	$\leq$	Big-O
$f(n) \in \Omega\left(g(n)\right)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} > 0$	$\exists\ n_0 \in \mathbb{N}, c \in \mathbb{R} : \forall\ n > n_0 \quad f(n) \geq c \cdot g(n)$	$\geq$	Big-Omega
$f(n) \in \Theta\left(g(n)\right)$	$0 < \limsup_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}$	$\exists\ n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R} : \forall\ n > n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$=$	Theta
$f(n) \in o\left(g(n)\right)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$\forall c \in \mathbb{R} \exists\ n_0 \in \mathbb{N} : \forall\ n > n_0 \quad f(n) \leq c \cdot g(n)$	$<$	little-o
$f(n) \in \omega\left(g(n)\right)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$	$\forall c \in \mathbb{R} \exists\ n_0 \in \mathbb{N} : \forall\ n > n_0 \quad f(n) \geq c \cdot g(n)$	$<$	little-omega

Asymptotic Complexity

L'Hôpital's rule

Asymptotic Complexity

L'Hôpital's rule

- Complexity functions tend to be increasing and unbounded

Asymptotic Complexity

L'Hôpital's rule

- Complexity functions tend to be increasing and unbounded
- As $n \to \infty$ both $f(n)$ and $g(n)$ go to ∞

Asymptotic Complexity

L'Hôpital's rule

- Complexity functions tend to be increasing and unbounded
- As $n \to \infty$ both $f(n)$ and $g(n)$ go to ∞
- In this case L'Hôpital's rule applies

Asymptotic Complexity

L'Hôpital's rule

- Complexity functions tend to be increasing and unbounded
- As $n \to \infty$ both $f(n)$ and $g(n)$ go to ∞
- In this case L'Hôpital's rule applies
- $$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \limsup_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Asymptotic Complexity

L'Hôpital's rule

- Complexity functions tend to be increasing and unbounded
- As $n \to \infty$ both $f(n)$ and $g(n)$ go to ∞
- In this case L'Hôpital's rule applies
- $$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \limsup_{n\to\infty} \frac{f'(n)}{g'(n)}$$
- For example: $$\limsup_{n\to\infty} \frac{3+80n}{3+6n+6n^2} = \limsup_{n\to\infty} \frac{80}{6+12n} = 0$$

Asymptotic Complexity

Properties

Asymptotic Complexity

Properties

- Coefficients can be dropped: $f(n) = c \cdot g(n) = O\left(g(n)\right)$

Asymptotic Complexity

Properties

- Coefficients can be dropped: $f(n) = c \cdot g(n) = O\left(g(n)\right)$
  - Using the constant definition the proof is straightforward

Asymptotic Complexity

Properties

- Coefficients can be dropped: $f(n) = c \cdot g(n) = O\left(g(n)\right)$
  - Using the constant definition the proof is straightforward
- In a polynomial only the largest power matters

Asymptotic Complexity

Properties

- Coefficients can be dropped: $f(n) = c \cdot g(n) = O\left(g(n)\right)$
  - Using the constant definition the proof is straightforward
- In a polynomial only the largest power matters

$$f(n) = a_k n^k + \ldots + a_0 = O(n^k)$$

Asymptotic Complexity

Properties

- Coefficients can be dropped: $f(n) = c \cdot g(n) = O\left(g(n)\right)$
  - Using the constant definition the proof is straightforward
- In a polynomial only the largest power matters

$$f(n) = a_k n^k + \ldots + a_0 = O(n^k)$$

$$\lim_{n\to\infty} \frac{a_k n^k + \ldots + a_0}{n^k} = \lim_{n\to\infty} \frac{a_k n^k}{n^k} + \ldots + \frac{a_0}{n^k} = a_k + \ldots + 0 = a_k \in \mathbb{R}$$

Asymptotic Complexity

Properties

Asymptotic Complexity

Properties

- Product: If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$ then $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$

Asymptotic Complexity

Properties

- Product: If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$ then $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$
- Sum: If $f_1(n) = O\left(g(n)\right)$ and $f_2(n) = O\left(g(n)\right)$ then $f_1(n) + f_2(n) = O\left(g(n)\right)$

Asymptotic Complexity

Properties

- Product: If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$ then $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$
- Sum: If $f_1(n) = O\left(g(n)\right)$ and $f_2(n) = O\left(g(n)\right)$ then $f_1(n) + f_2(n) = O\left(g(n)\right)$

Sorting

Sorting

Putting things in order

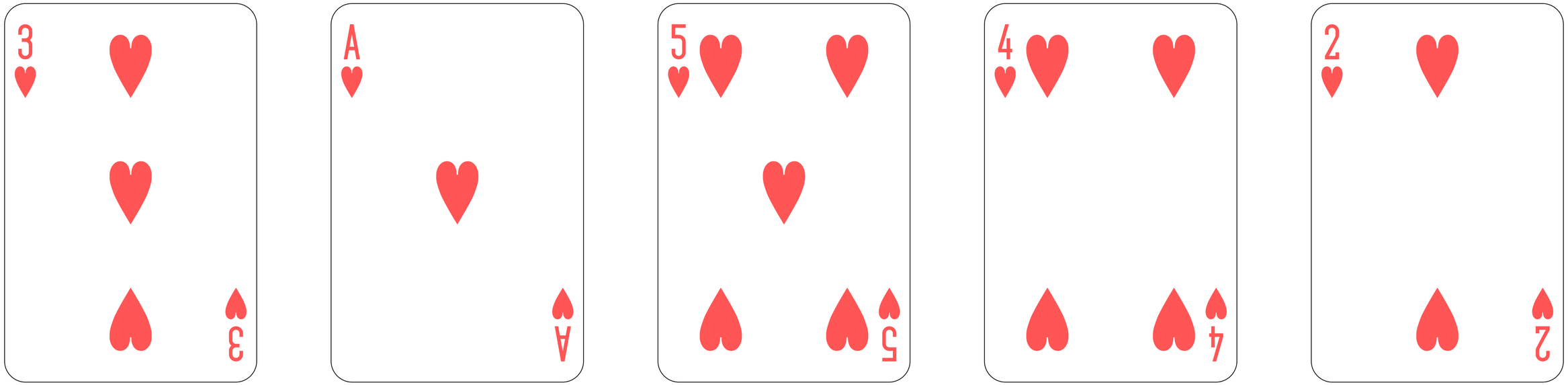
- What do we have:
  - A *list* of *n* elements
  - A comparison function: $\leq$
- What do we want:
  - The *list* has all the same elements as it started with
  - If $i \leq j$, `list[i]` $\leq$ `list[j]`

Sorting

Example

Sorting

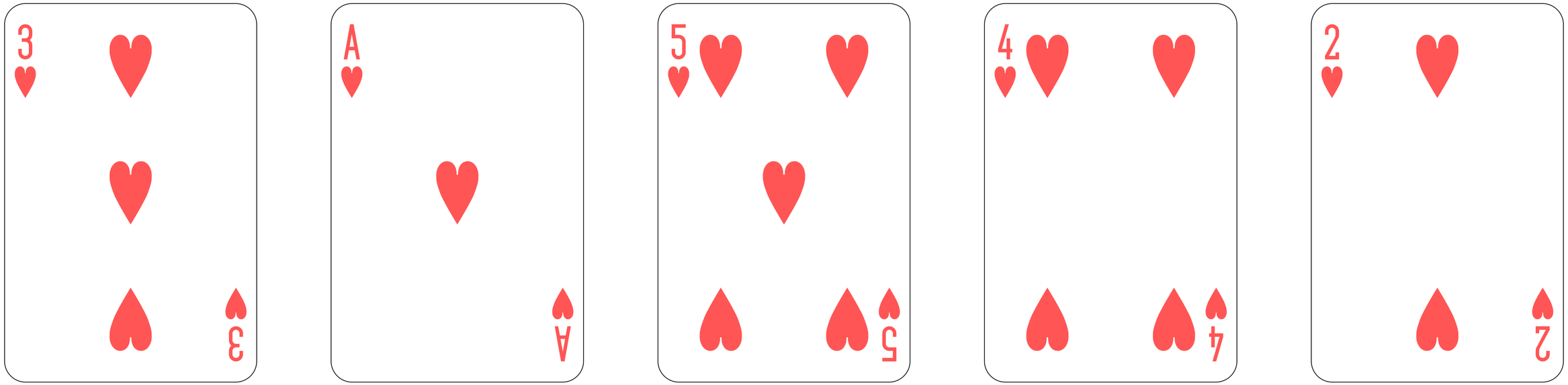
Example



Sorting

Example 1 (how I do it)

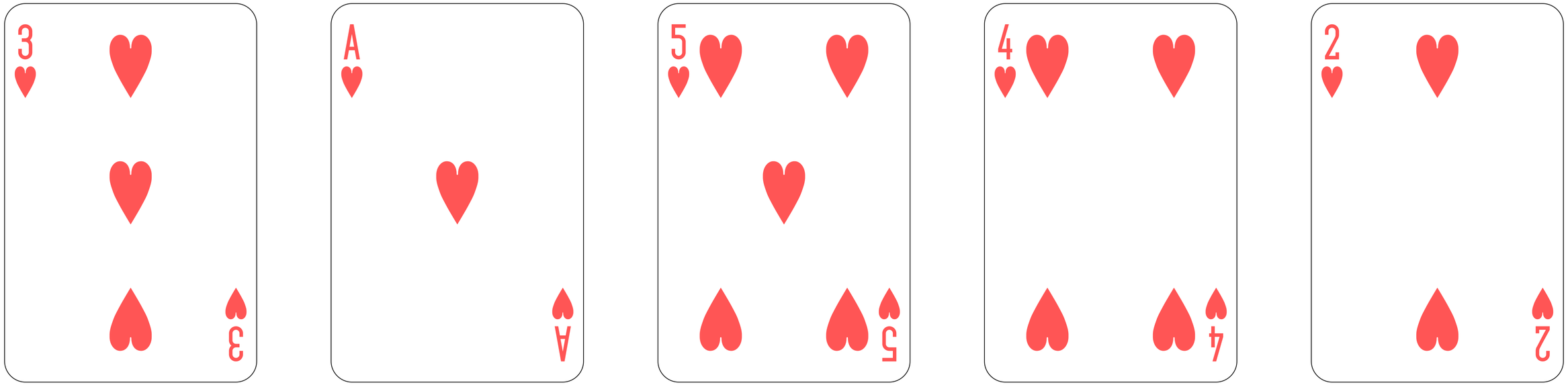
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

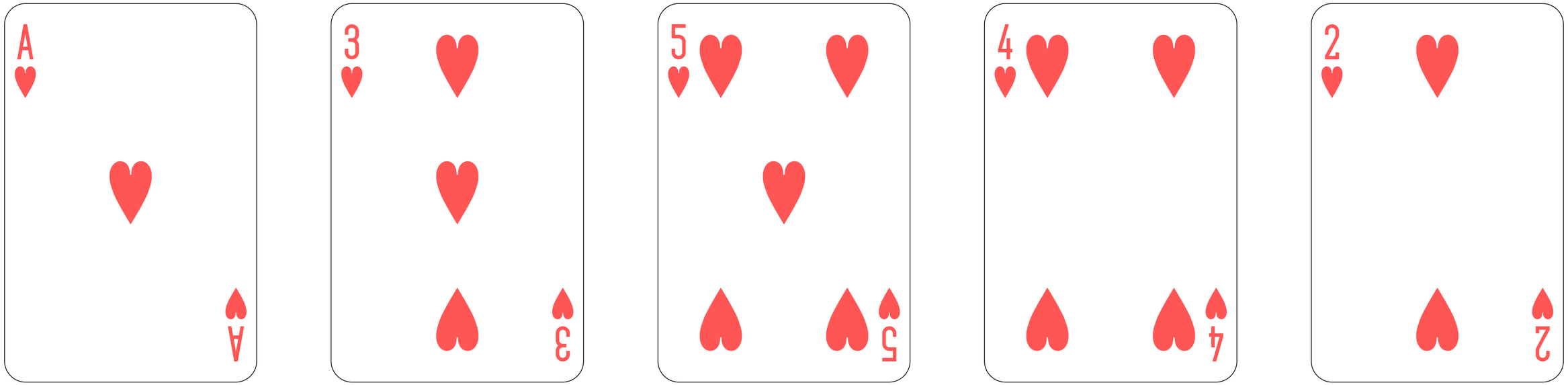
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

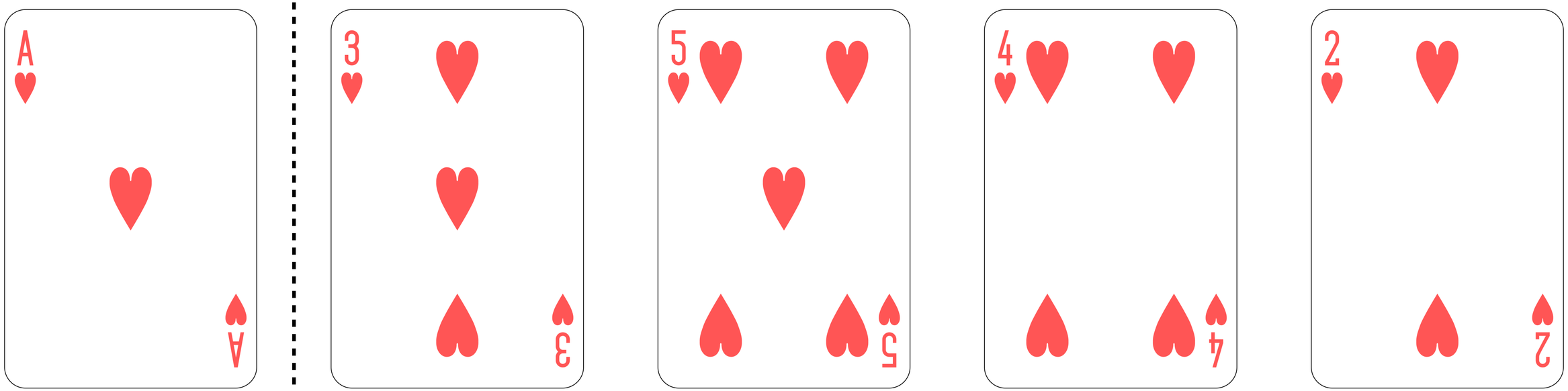
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

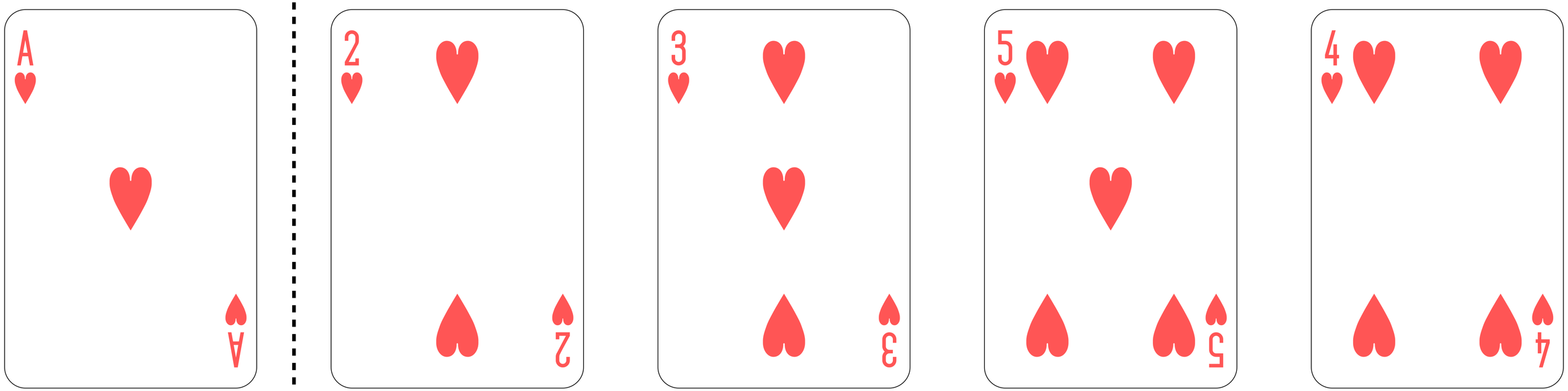
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

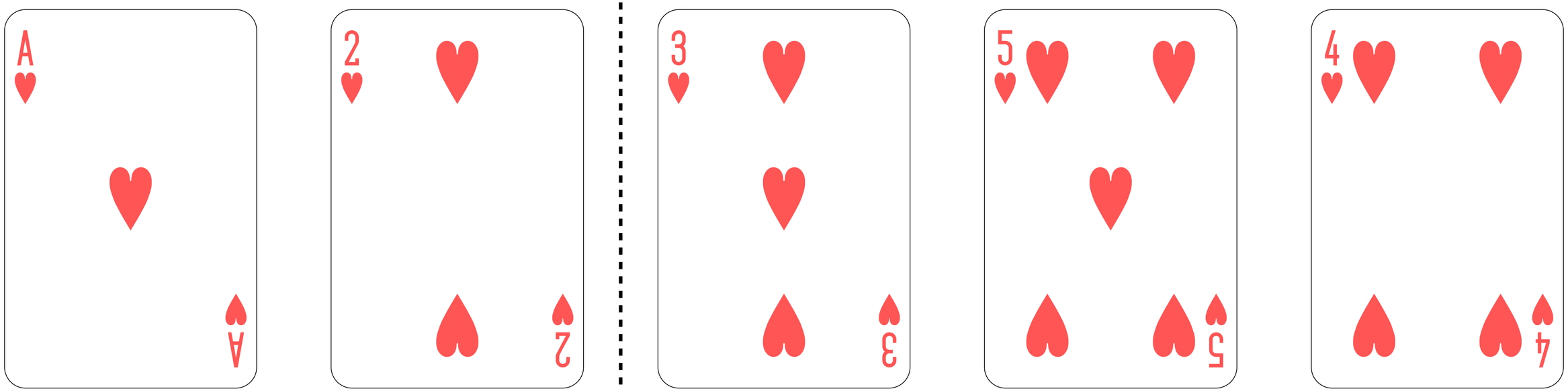
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

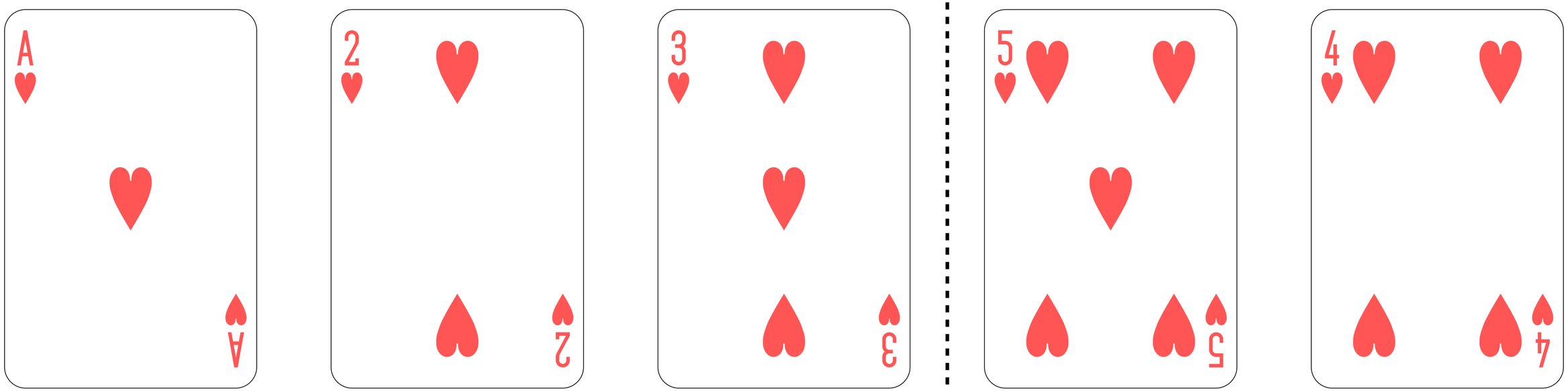
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

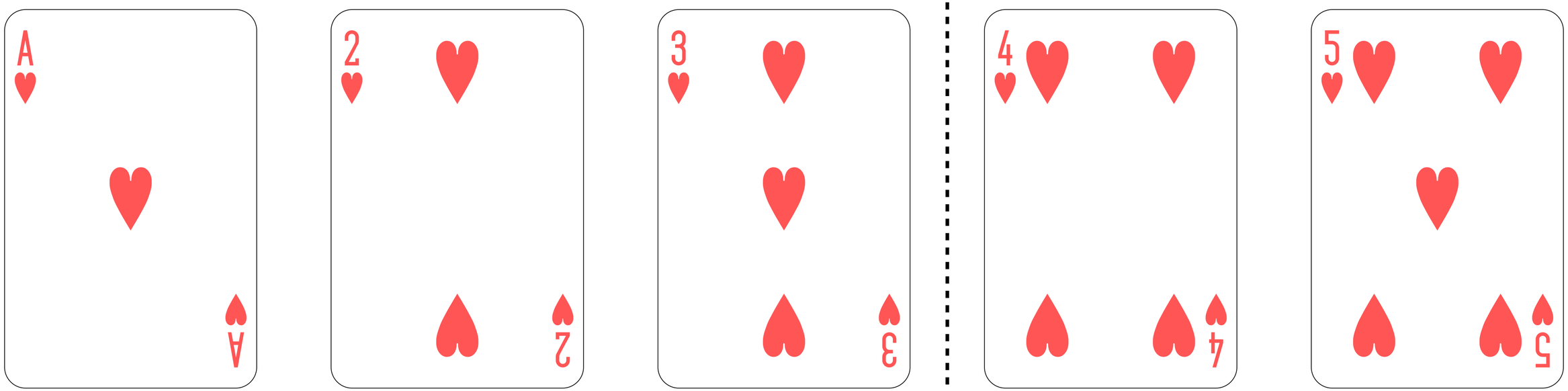
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

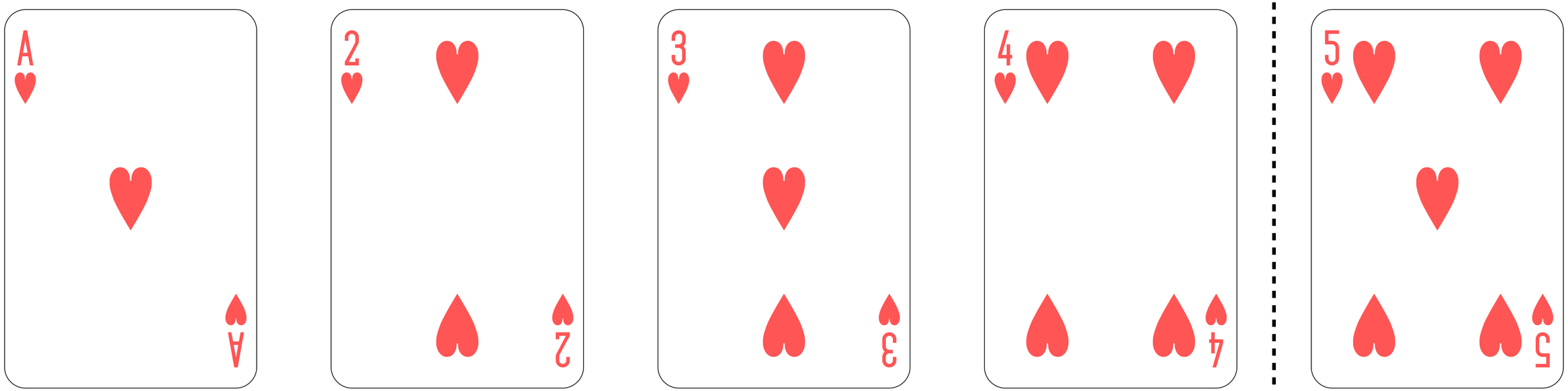
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

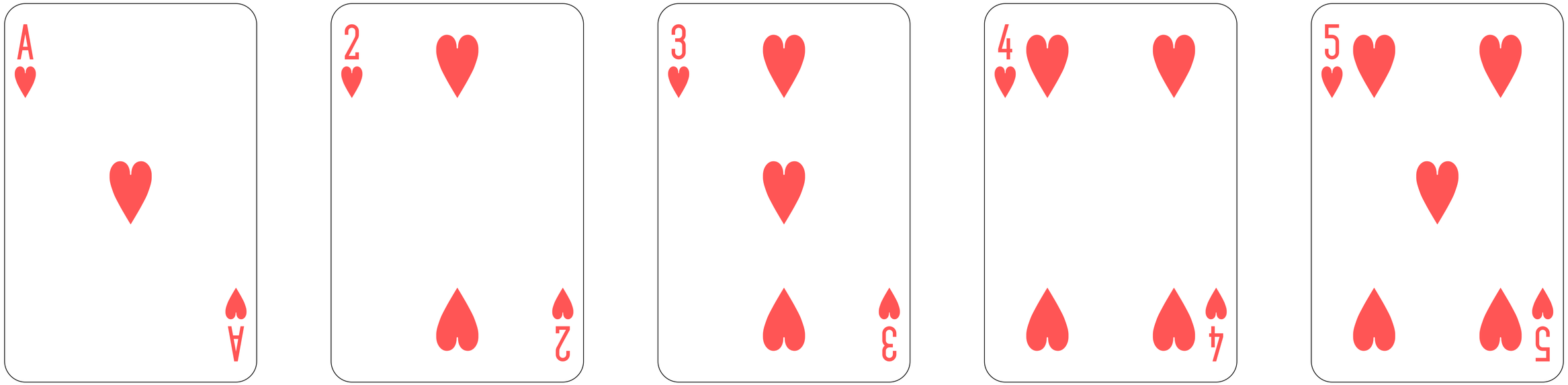
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- Why swap instead of pushing things over?

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- Why swap instead of pushing things over?
  - It's more efficient and we don't care about the order of the unsorted part

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- Why swap instead of pushing things over?
  - It's more efficient and we don't care about the order of the unsorted part
- This is called *selection sort* -- we *select* the one we want repeatedly.

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- How many comparisons do we need to do?

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- How many comparisons do we need to do?
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- How many comparisons do we need to do?
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$
- How many swaps?

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- How many comparisons do we need to do?
  - $(n-1)+(n-2)+\ldots+1 = n(n-1)/2 = O(n^2)$
- How many swaps?
  - $n-1$

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

- Everything left of the line is sorted.

Sorting

Example 1 (how I do it): Algorithm

```
For i = 1 to n - 1:
    min = index of smallest in A[i : n]
    swap A[min] and A[i]
```

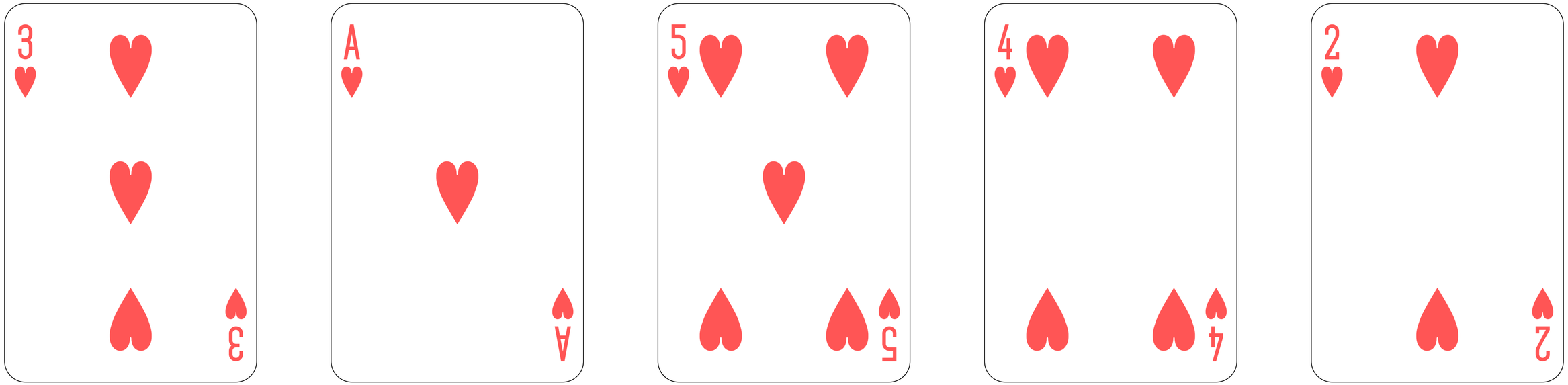
- Everything left of the line is sorted.
- Scanning the right (unsorted) part, and putting it to the end of the left (sorted) part.

Sorting

Example 2

Sorting

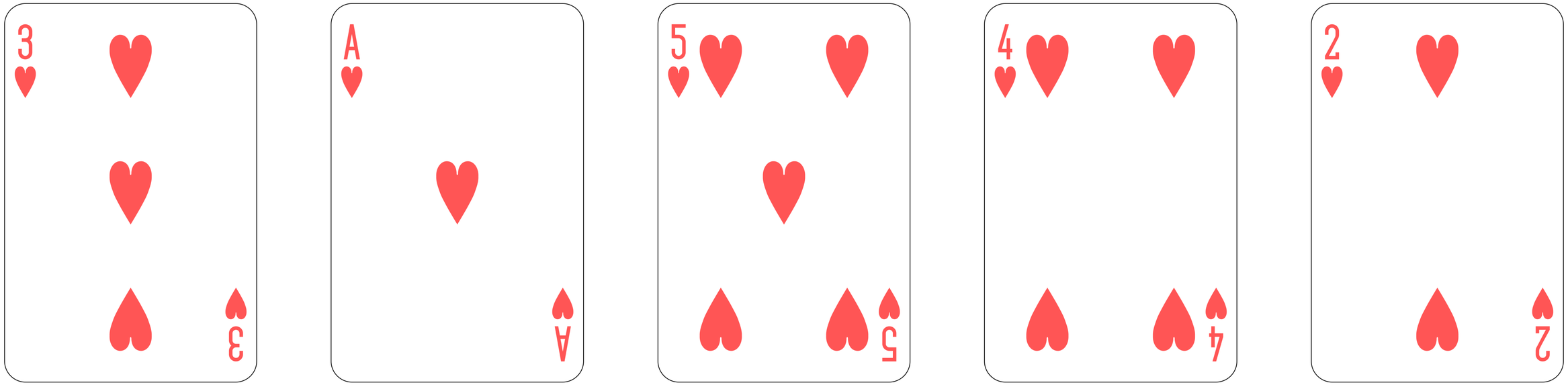
Example 2



Sorting

Example 2

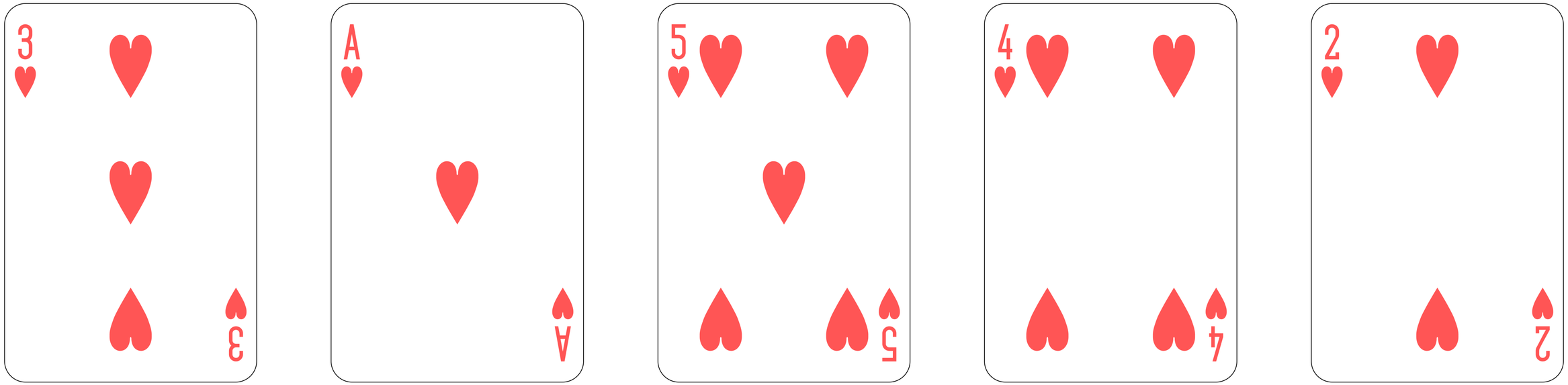
- Pick the first unsorted and insert it into the right place



Sorting

Example 2

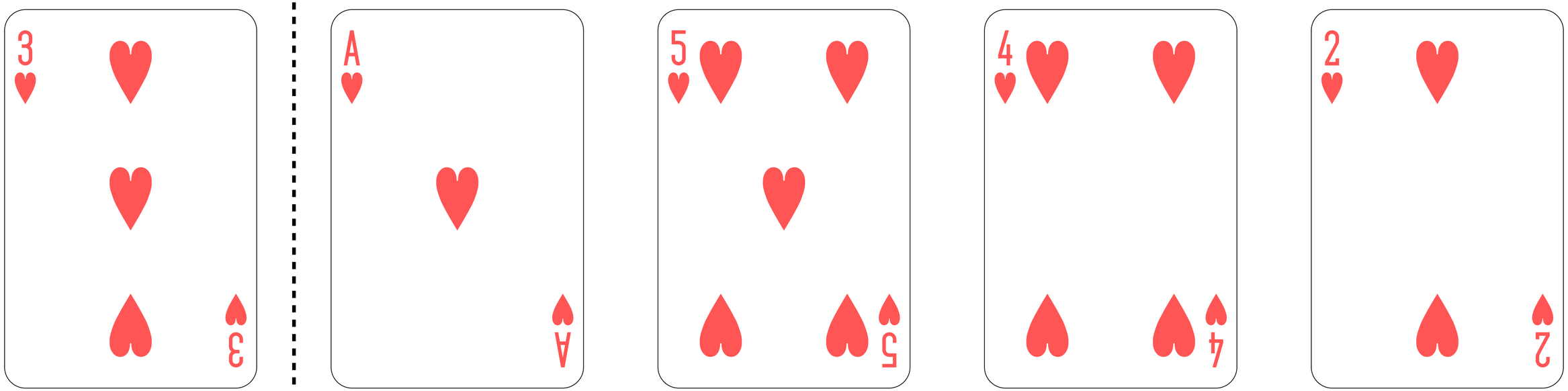
- Pick the first unsorted and insert it into the right place



Sorting

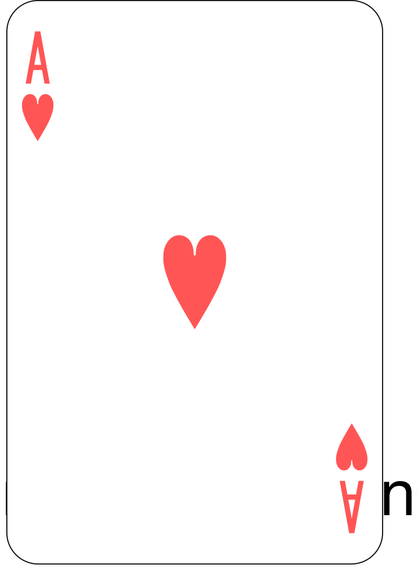
Example 2

- Pick the first unsorted and insert it into the right place

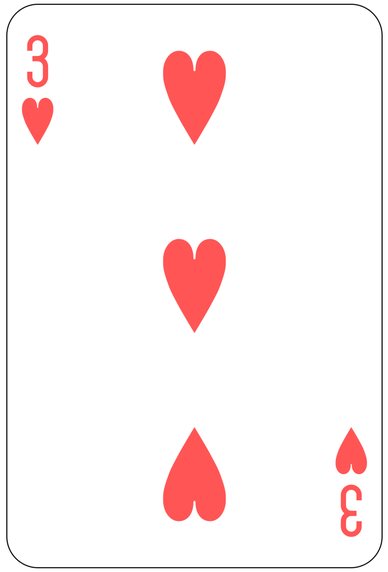


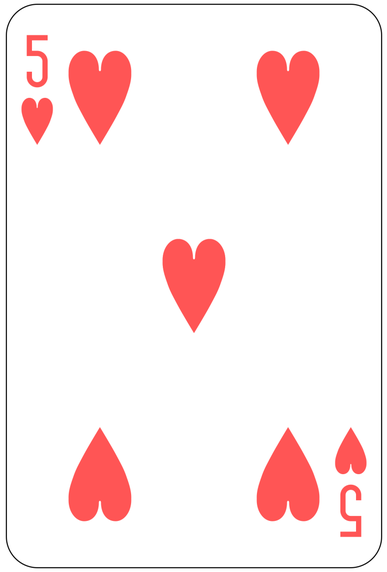
Sorting

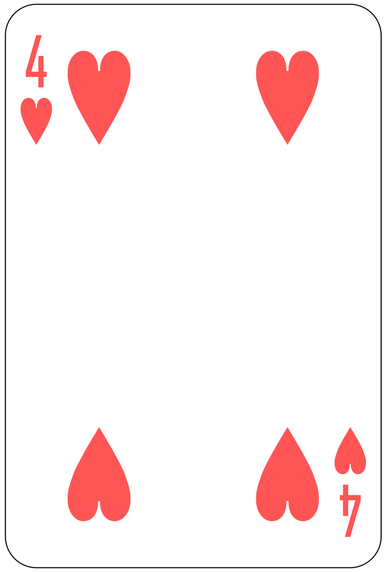
Example 2

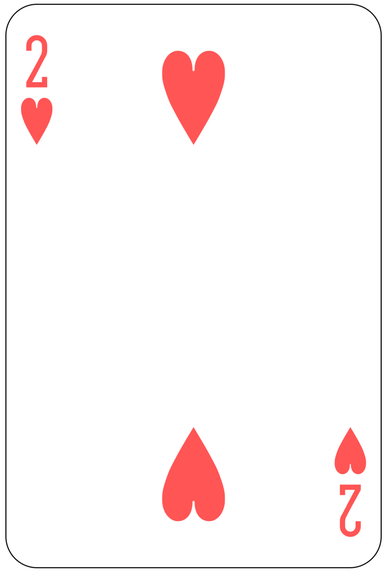


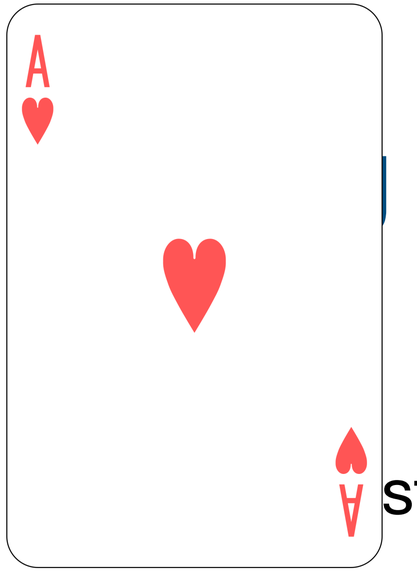
- Pick the first u nsert it into the right place



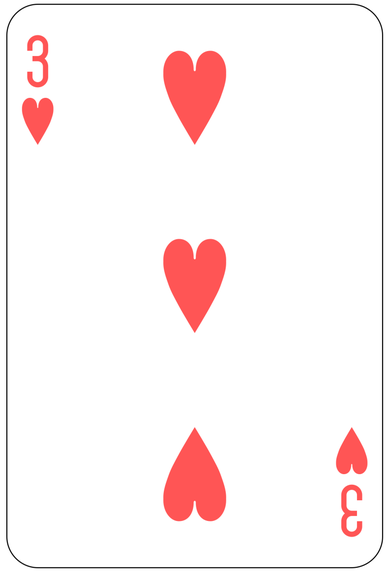


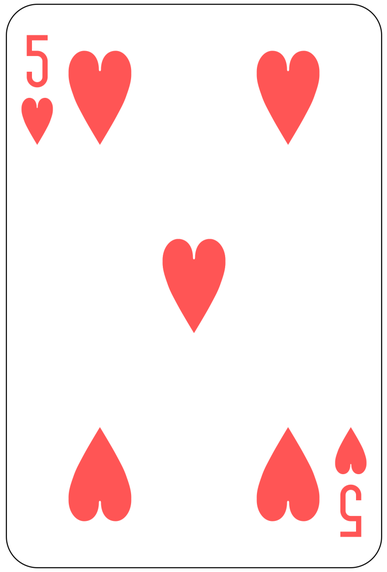


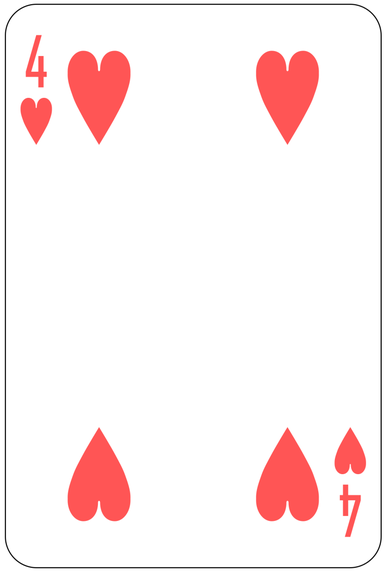


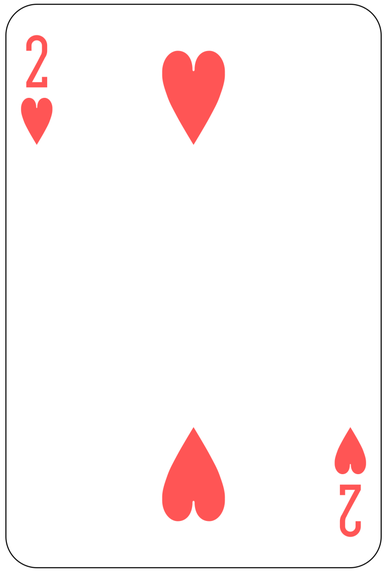


st unsorted and insert it into the right place





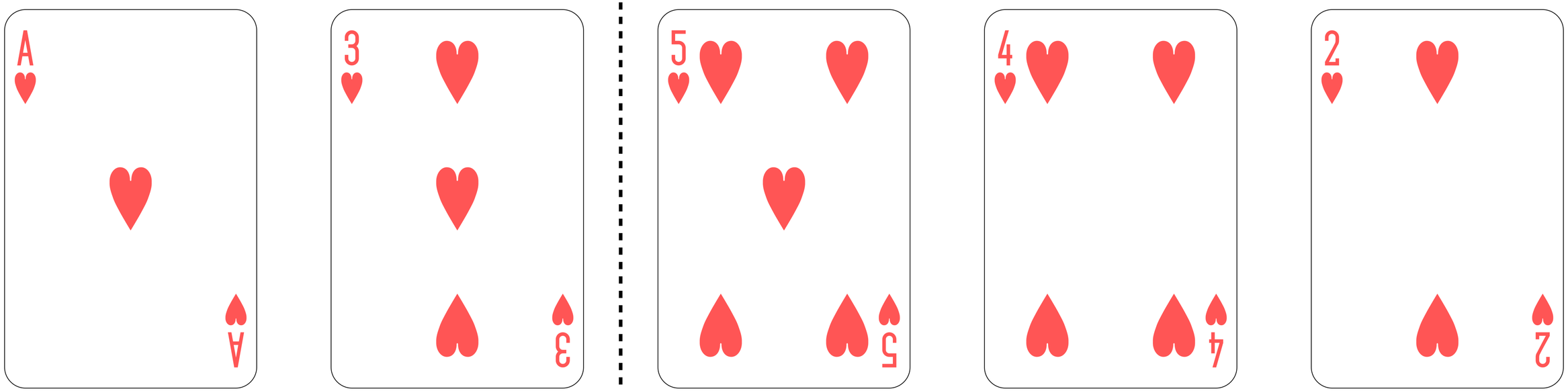




Sorting

Example 2

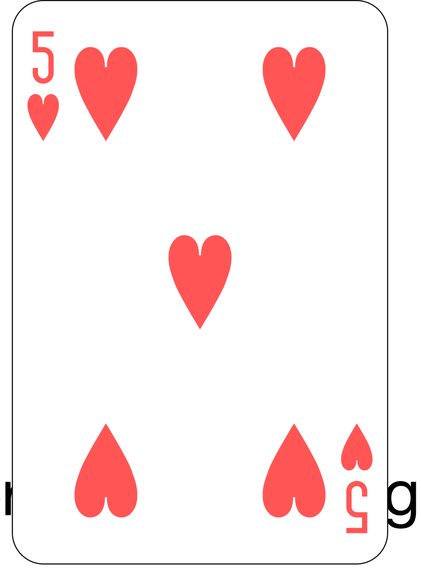
- Pick the first unsorted and insert it into the right place

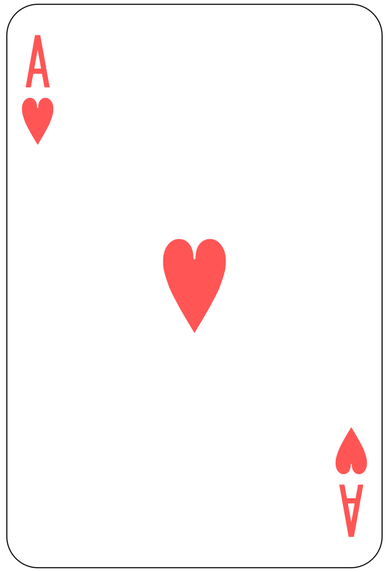


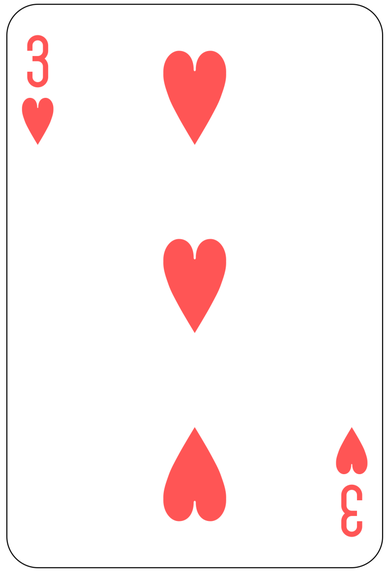
Sorting

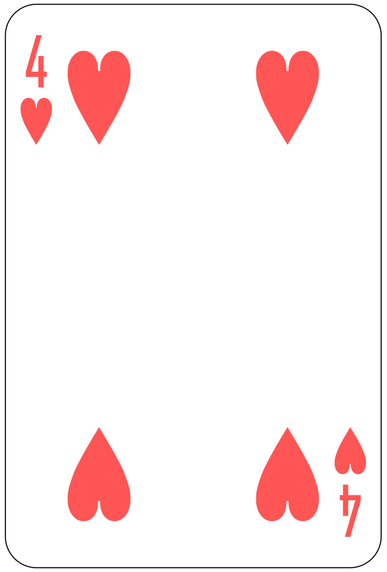
Example 2

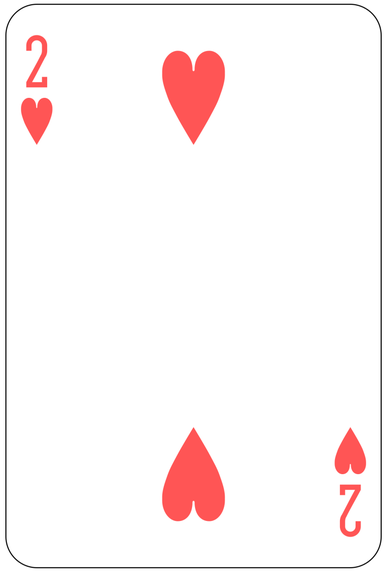
- Pick the first unsorted and inser ght place





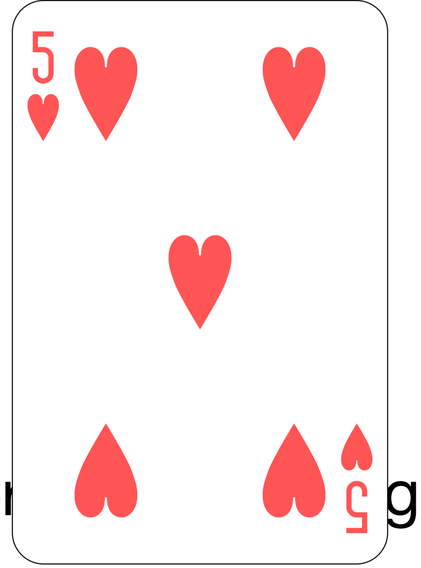




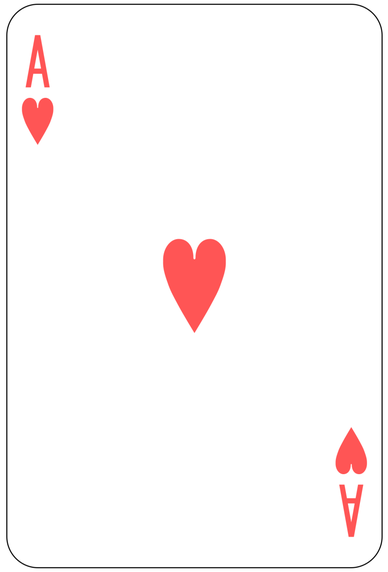


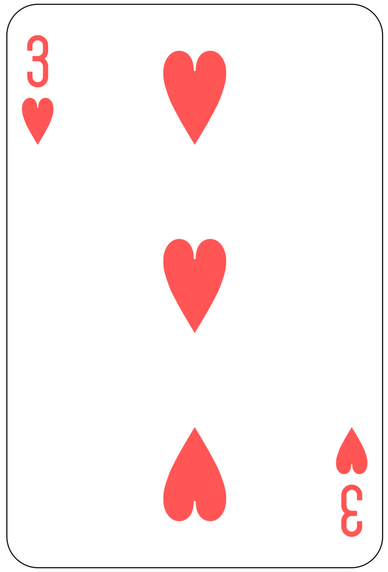
Sorting

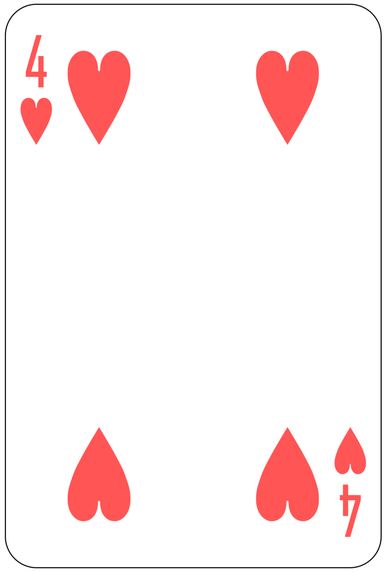
Example 2

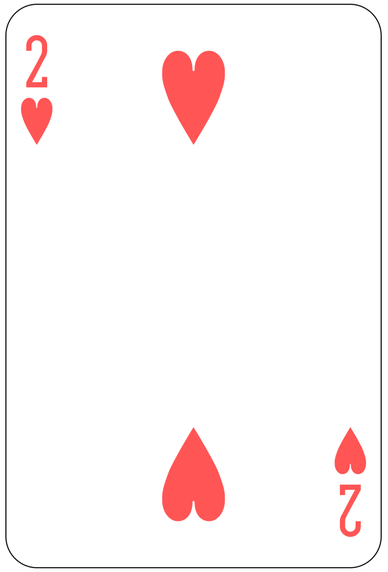


- Pick the first unsorted and inser ght place





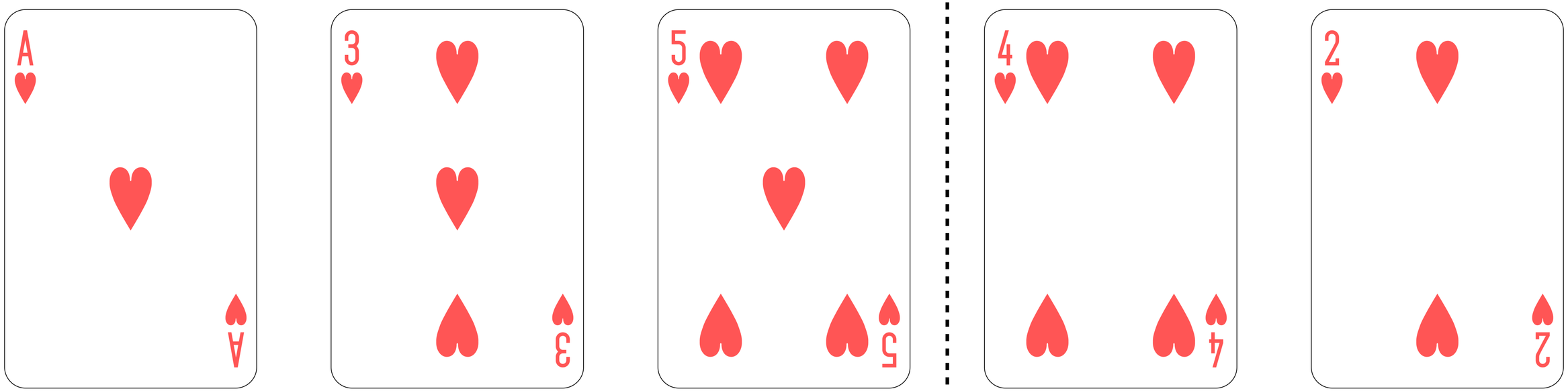




Sorting

Example 2

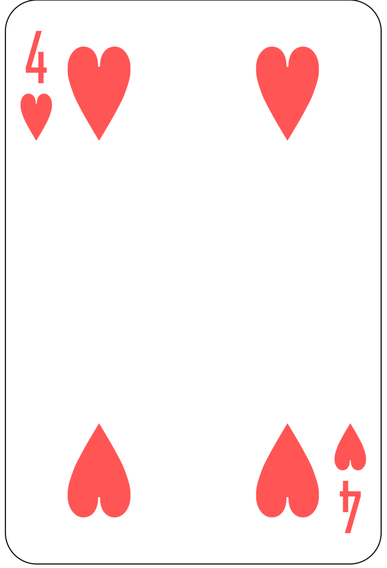
- Pick the first unsorted and insert it into the right place

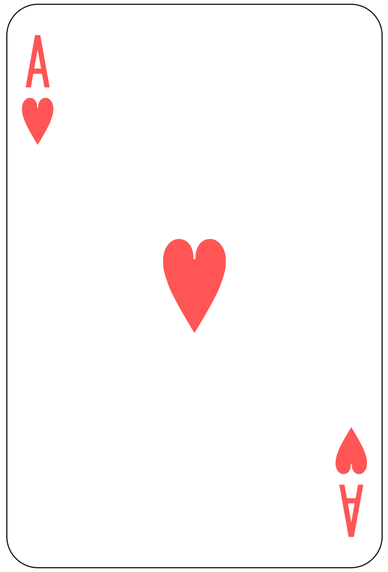


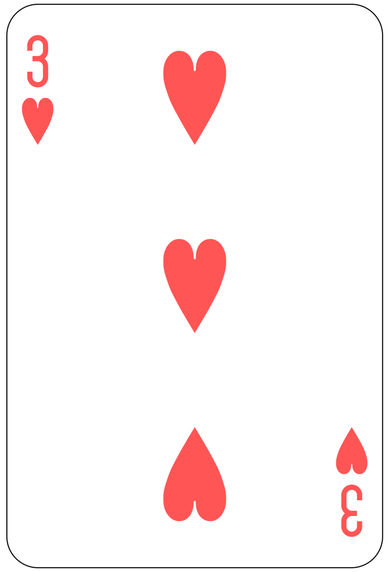
Sorting

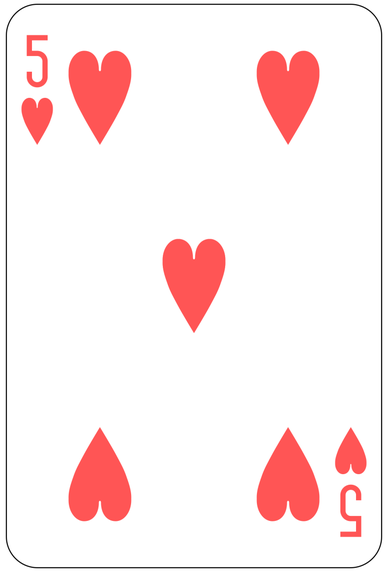
Example 2

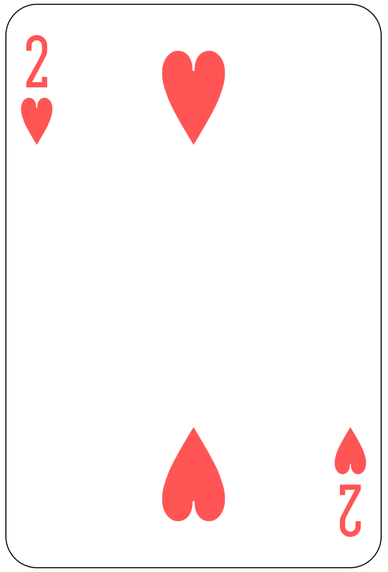
- Pick the first unsorted and insert it into the right





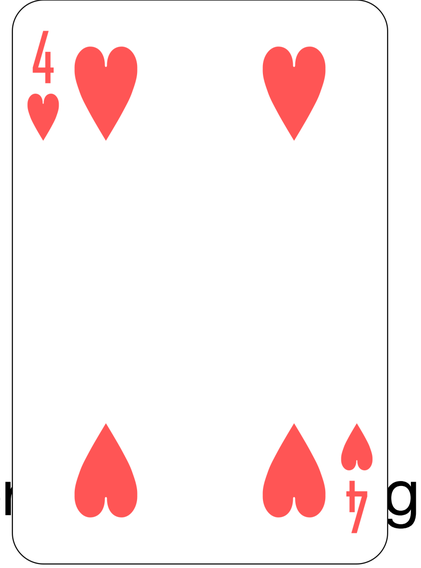




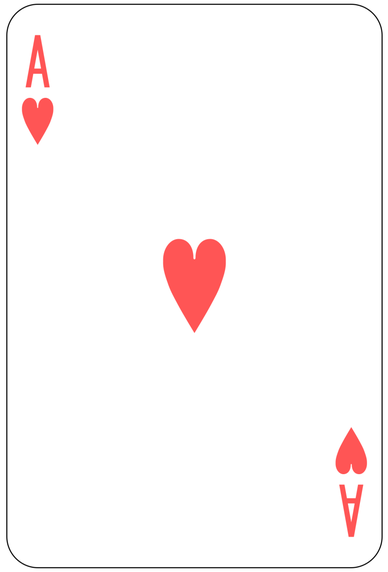


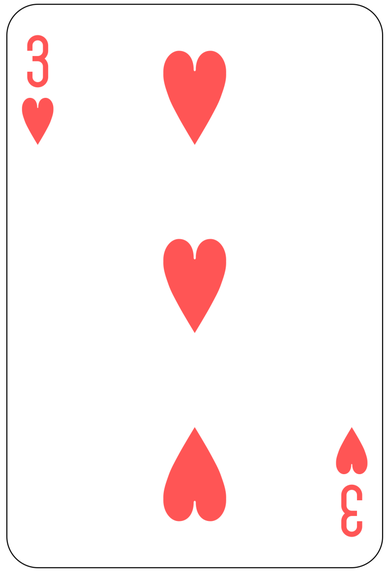
Sorting

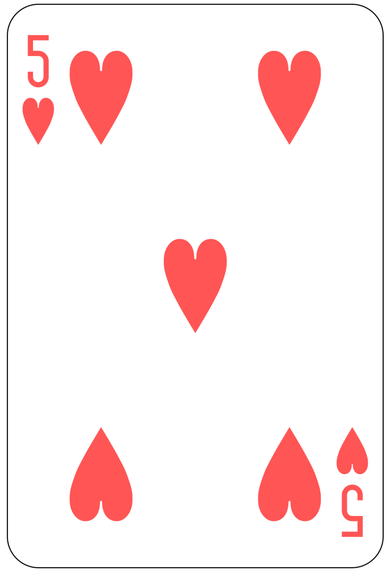
Example 2

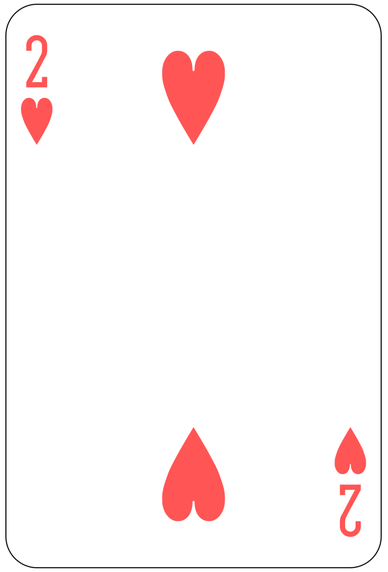


- Pick the first unsorted and inser ght place





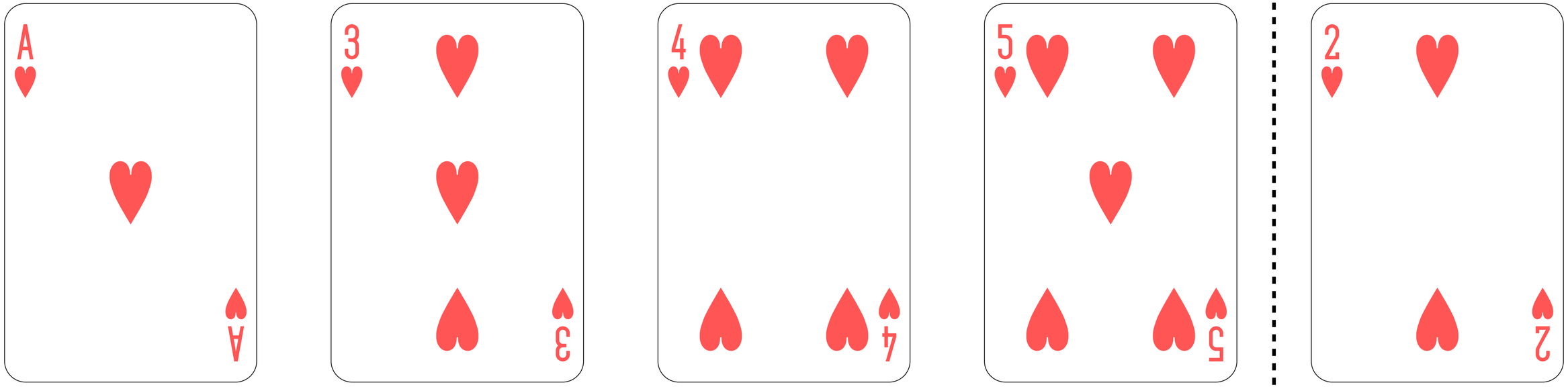




Sorting

Example 2

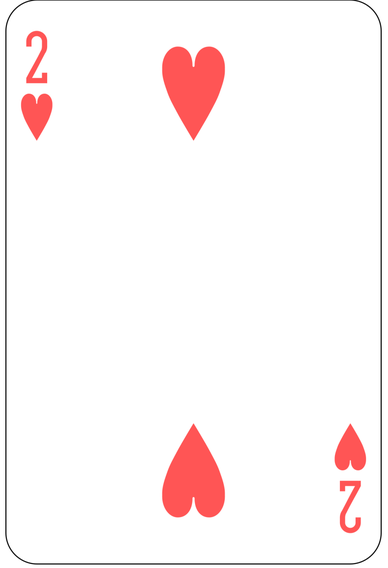
- Pick the first unsorted and insert it into the right place

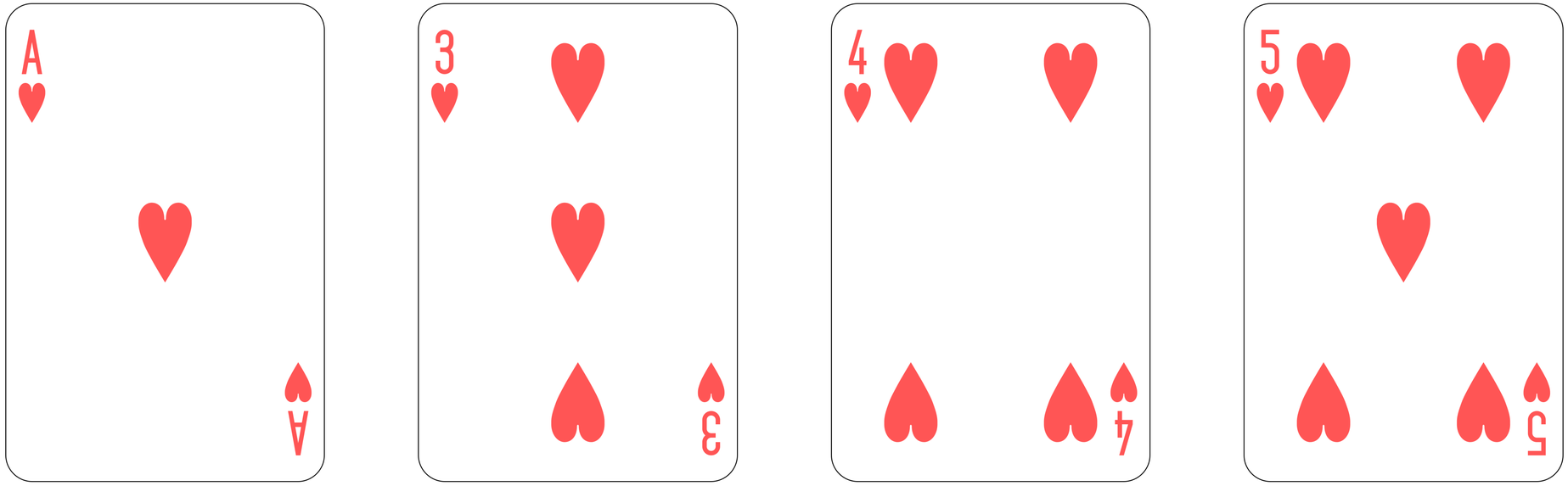


Sorting

Example 2

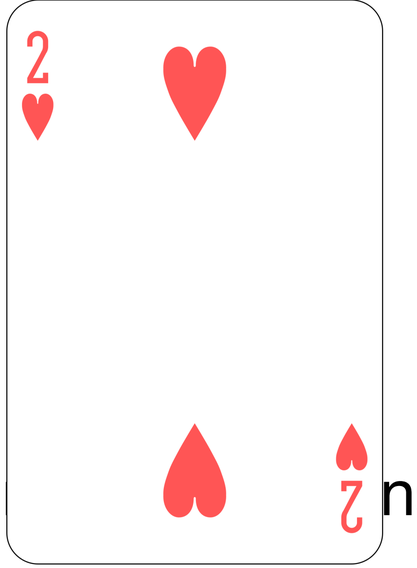
- Pick the first unsorted and insert it into the right place



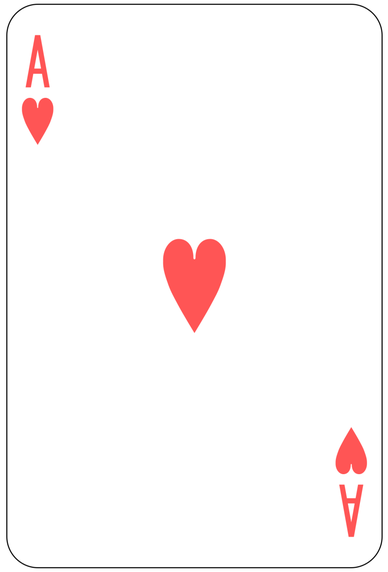


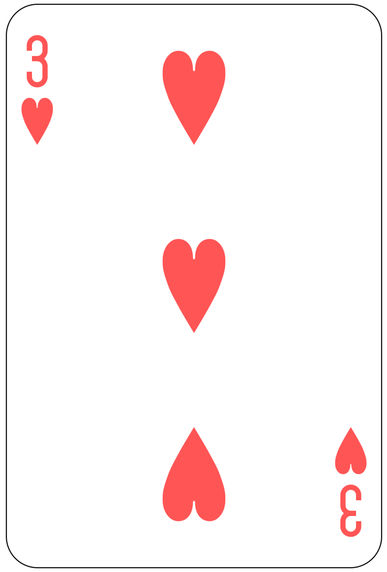
Sorting

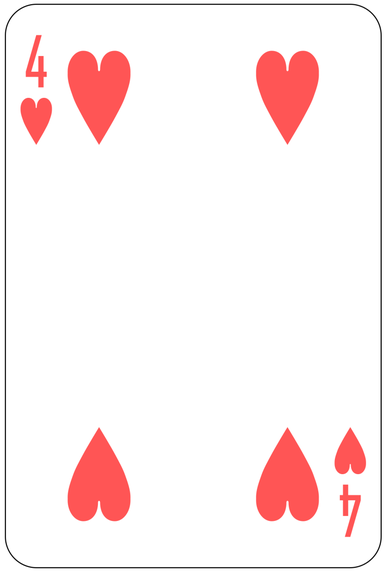
Example 2

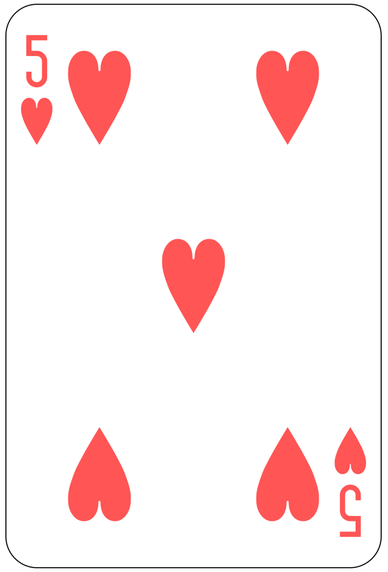


- Pick the first u nsert it into the right place





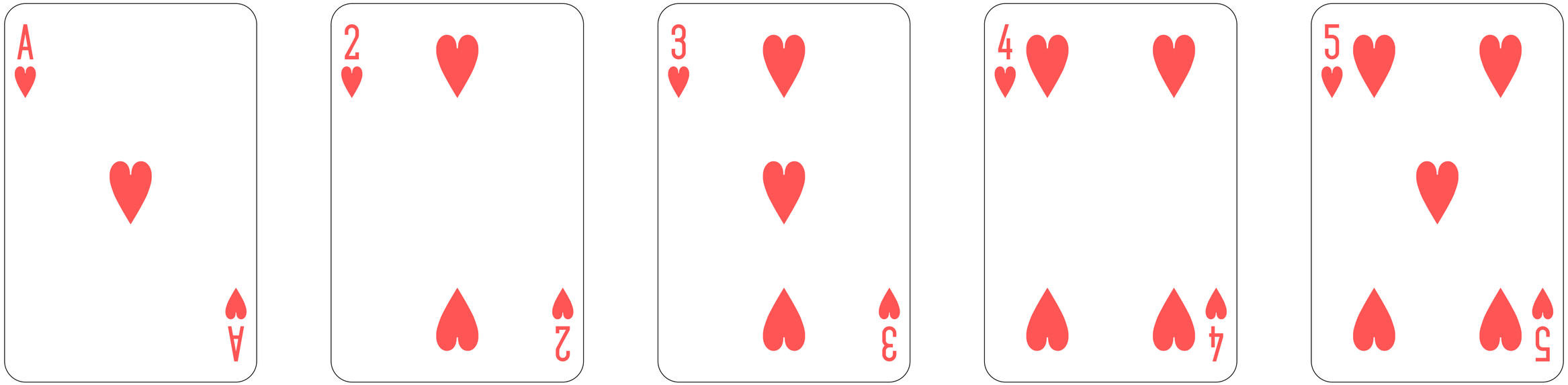




Sorting

Example 2

- Pick the first unsorted and insert it into the right place



Sorting

Example 2: Algorithm

```
for i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

Sorting

Example 2: Algorithm

```
for i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Take the first unsorted and insert it into the right place in the sorted pile

Sorting

Example 2: Algorithm

```
for i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Take the first unsorted and insert it into the right place in the sorted pile
- Inserting means shifting everything after the card by one place

Sorting

Example 2: Algorithm

```
for i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Take the first unsorted and insert it into the right place in the sorted pile
- Inserting means shifting everything after the card by one place
- This is called *insertion sort*.

Sorting

Example 2: Algorithm

```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

Sorting

Example 2: Algorithm

```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Comparison: worst-case $O(n^2)$

Sorting

Example 2: Algorithm

```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Comparison: worst-case $O(n^2)$
- Swap: worst-case $O(n^2)$

Sorting

Example 2: Algorithm

```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

Sorting

Example 2: Algorithm

```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Everything left of the line is sorted

Sorting

Example 2: Algorithm

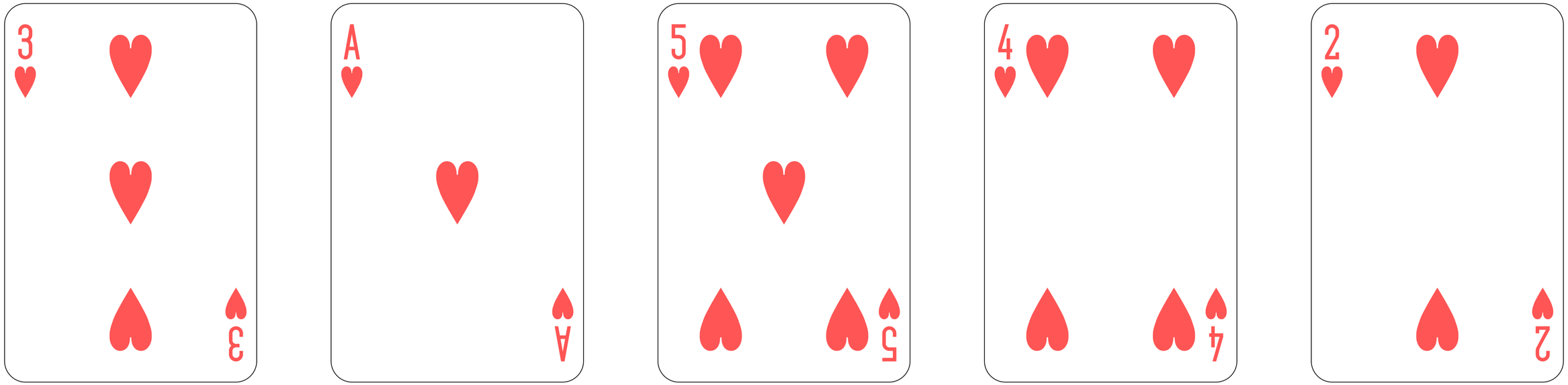
```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

- Everything left of the line is sorted
- Take the first one on the right, scanning the left to find a place.

Sorting

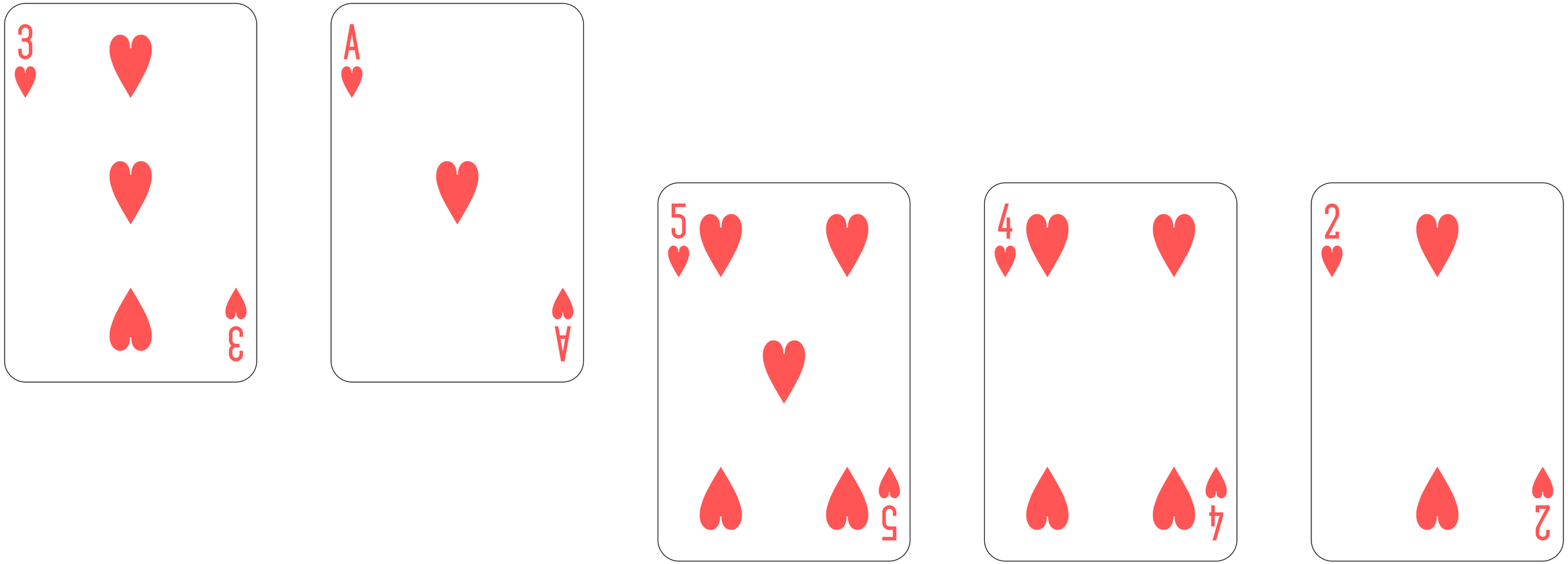
Example 3

- Take two cards, swap them if out of order.



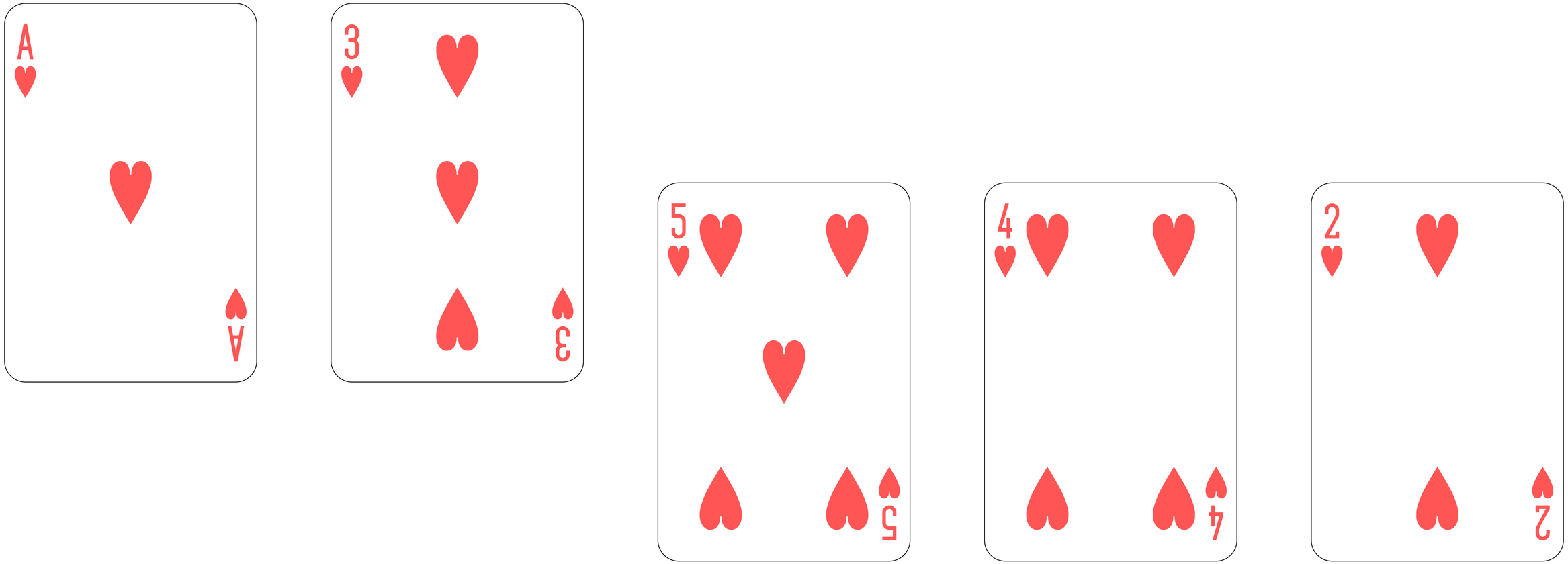
Sorting

Example 3



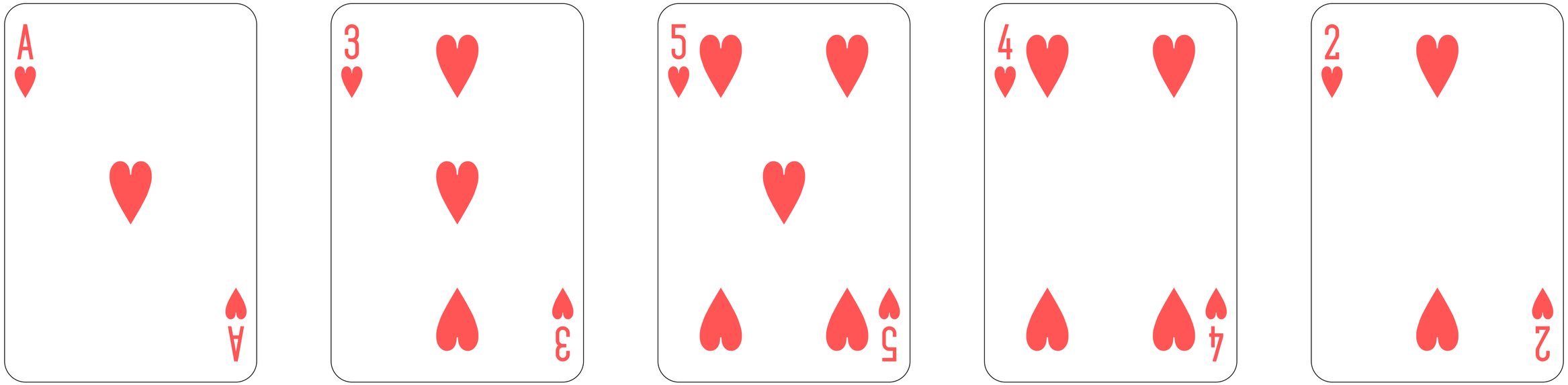
Sorting

Example 3



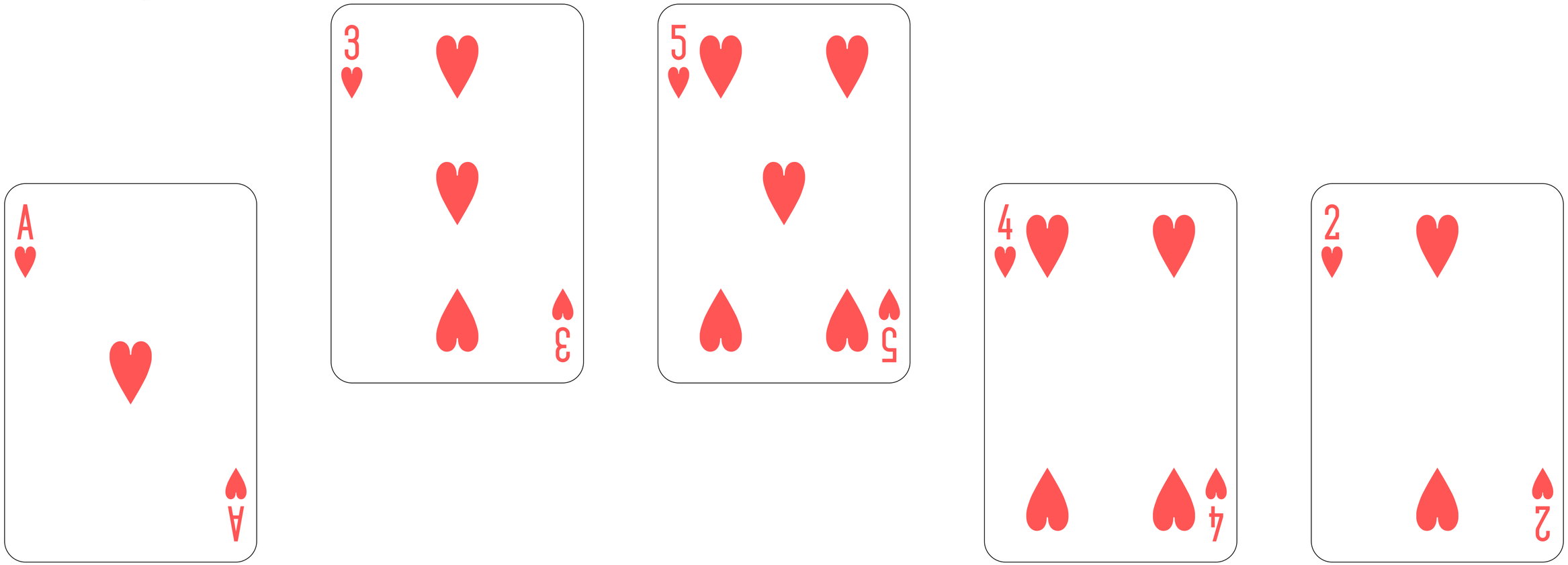
Sorting

Example 3



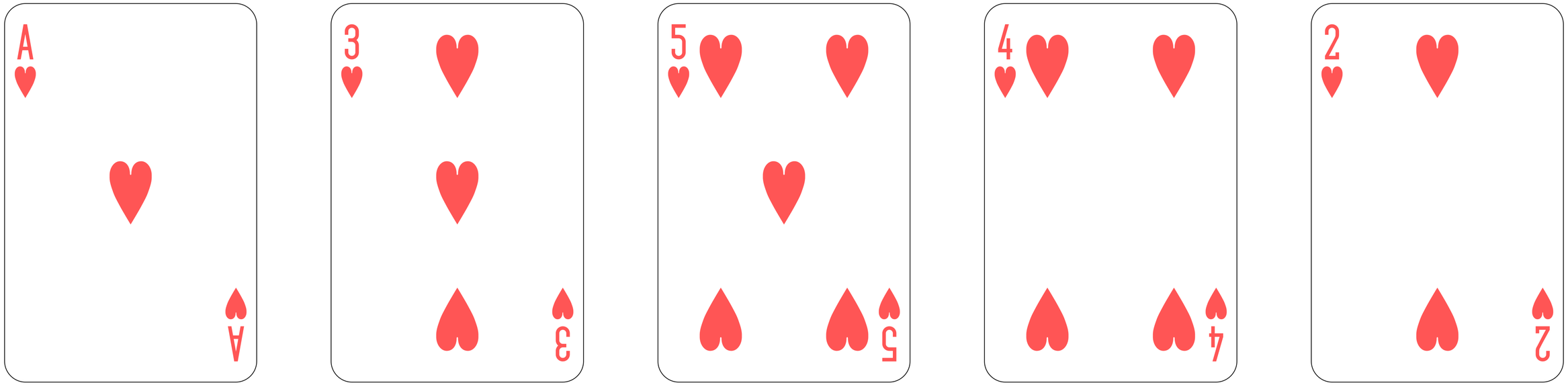
Sorting

Example 3



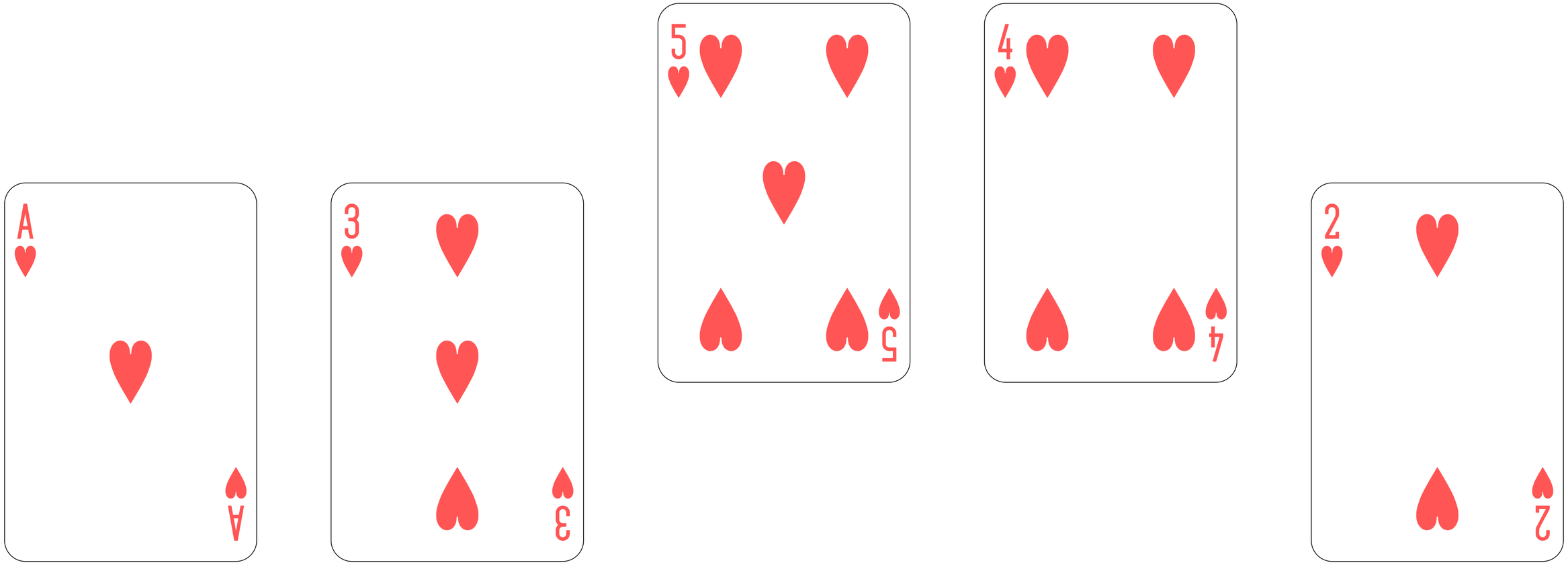
Sorting

Example 3



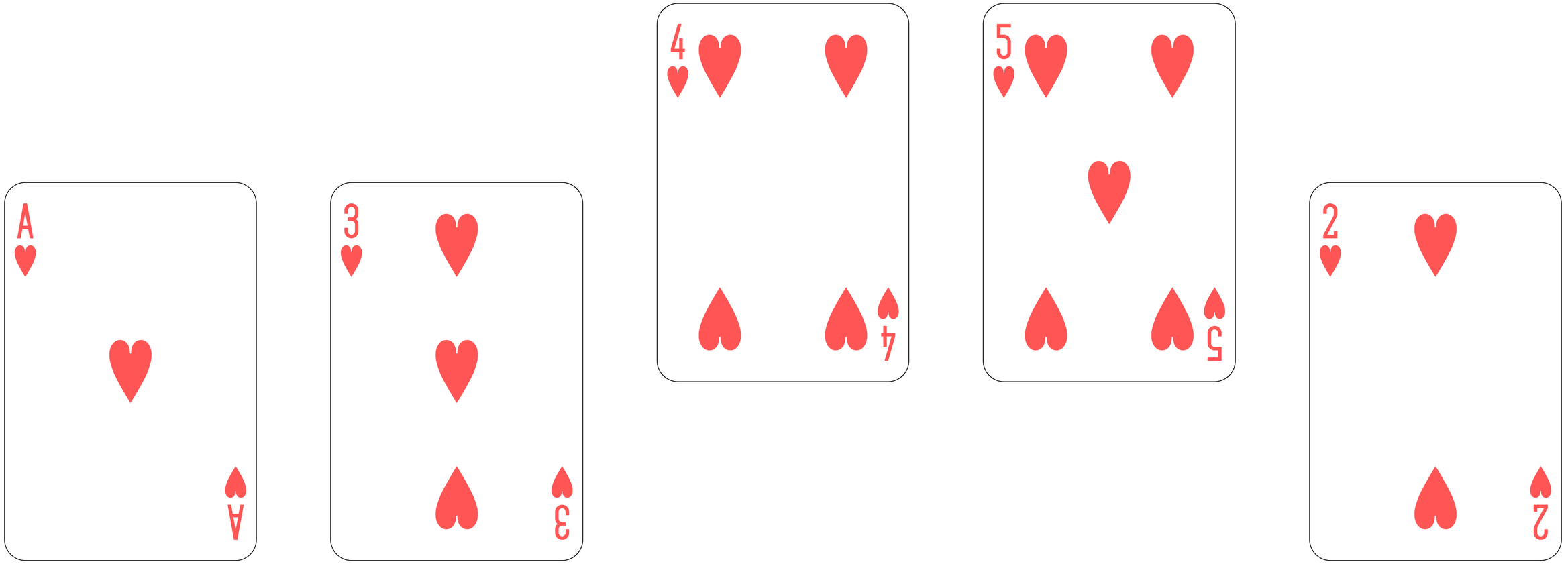
Sorting

Example 3



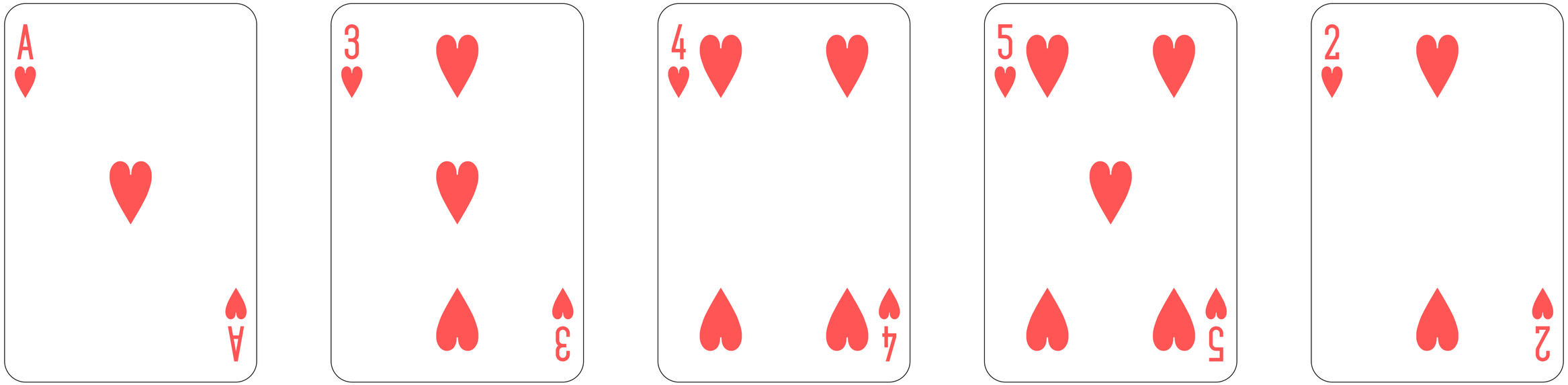
Sorting

Example 3



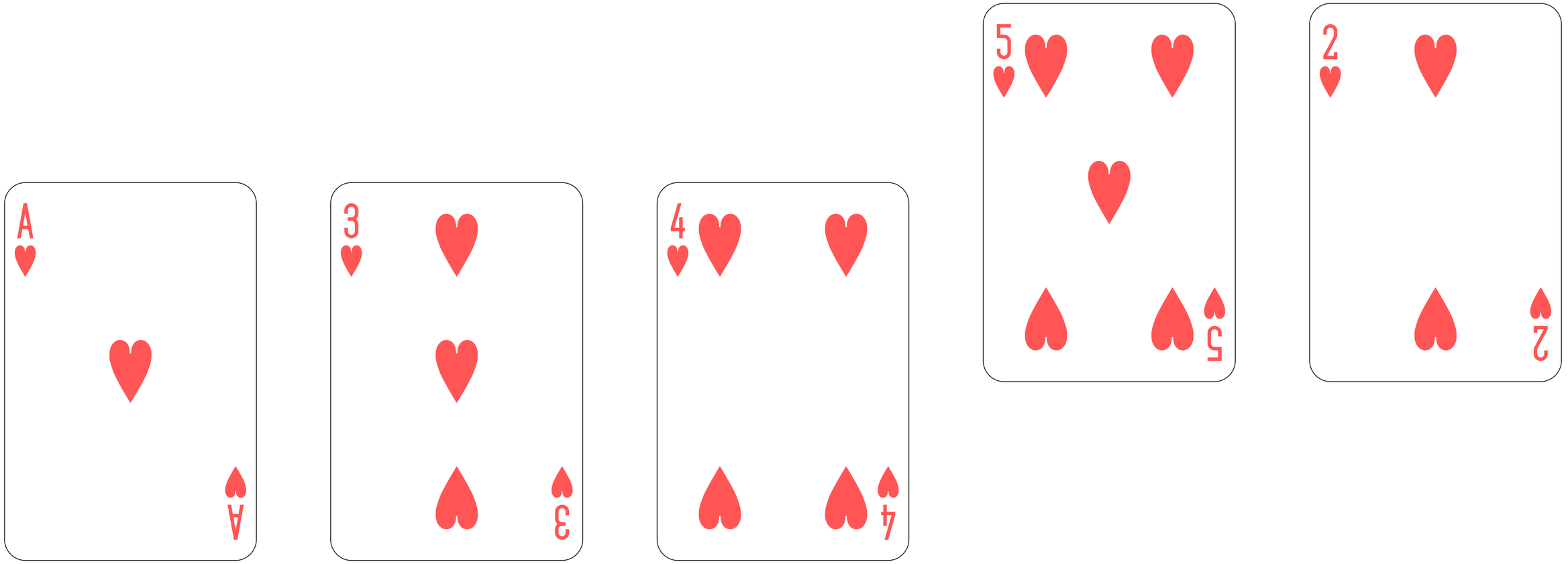
Sorting

Example 3



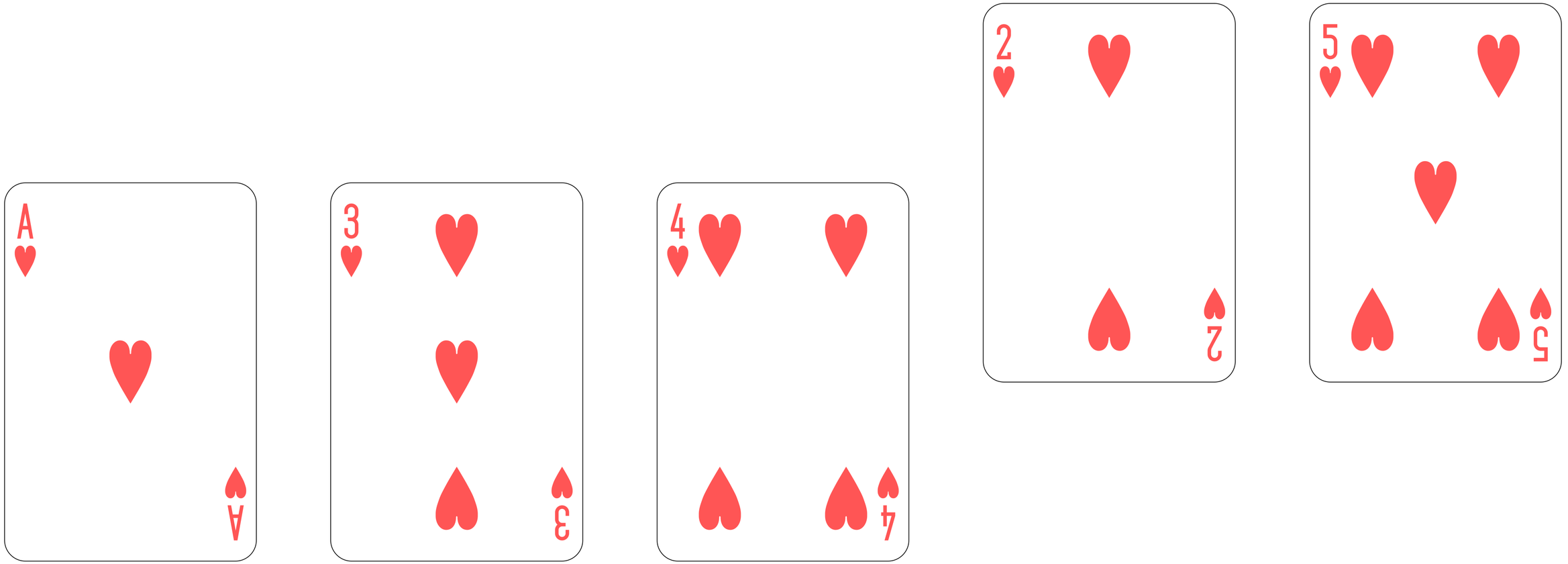
Sorting

Example 3



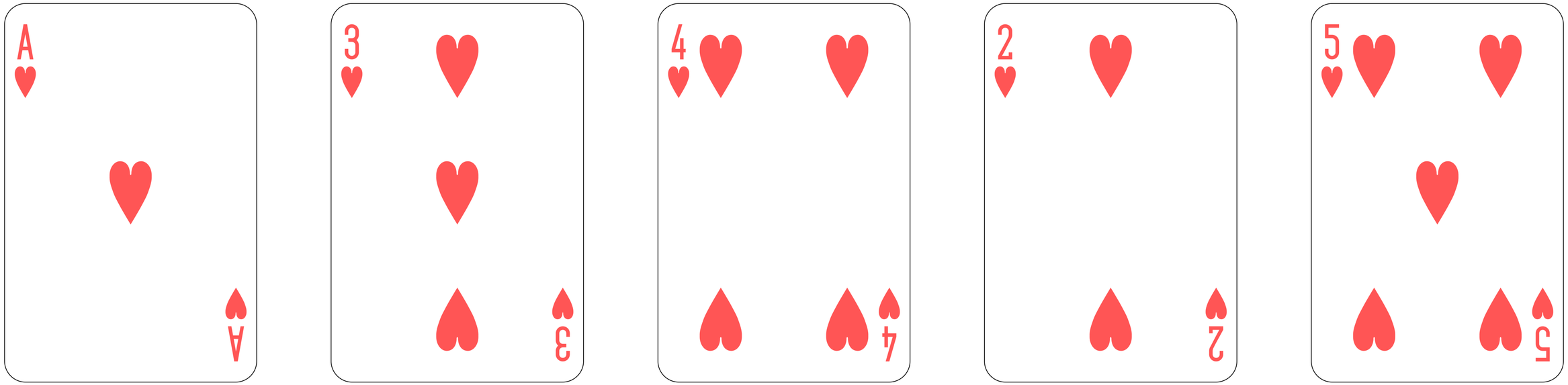
Sorting

Example 3



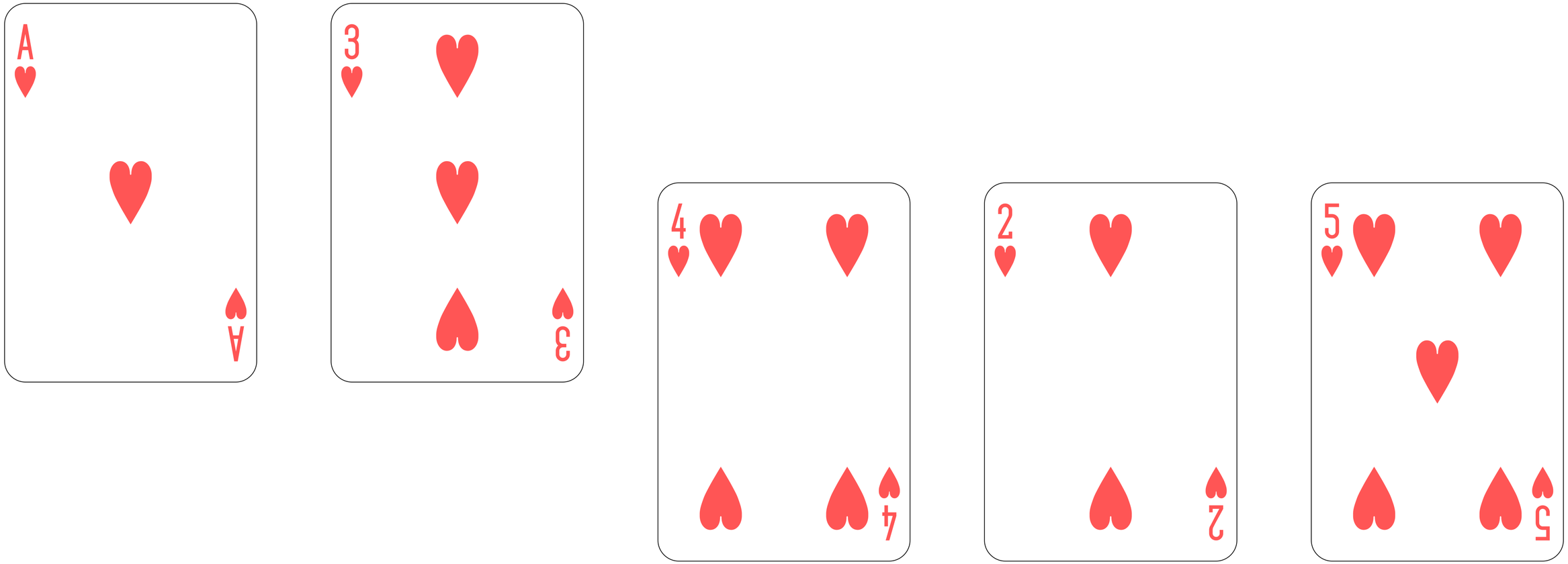
Sorting

Example 3



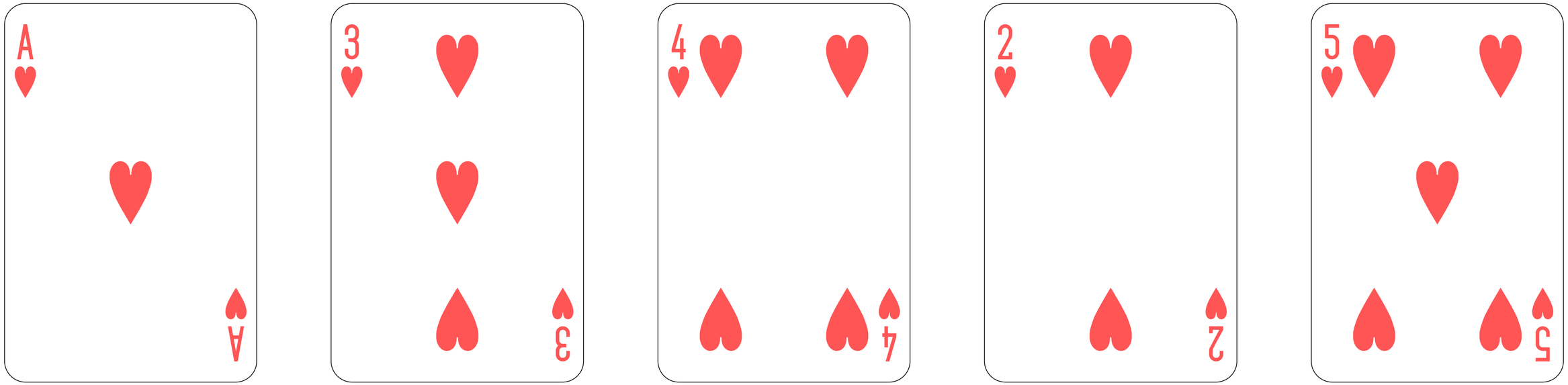
Sorting

Example 3



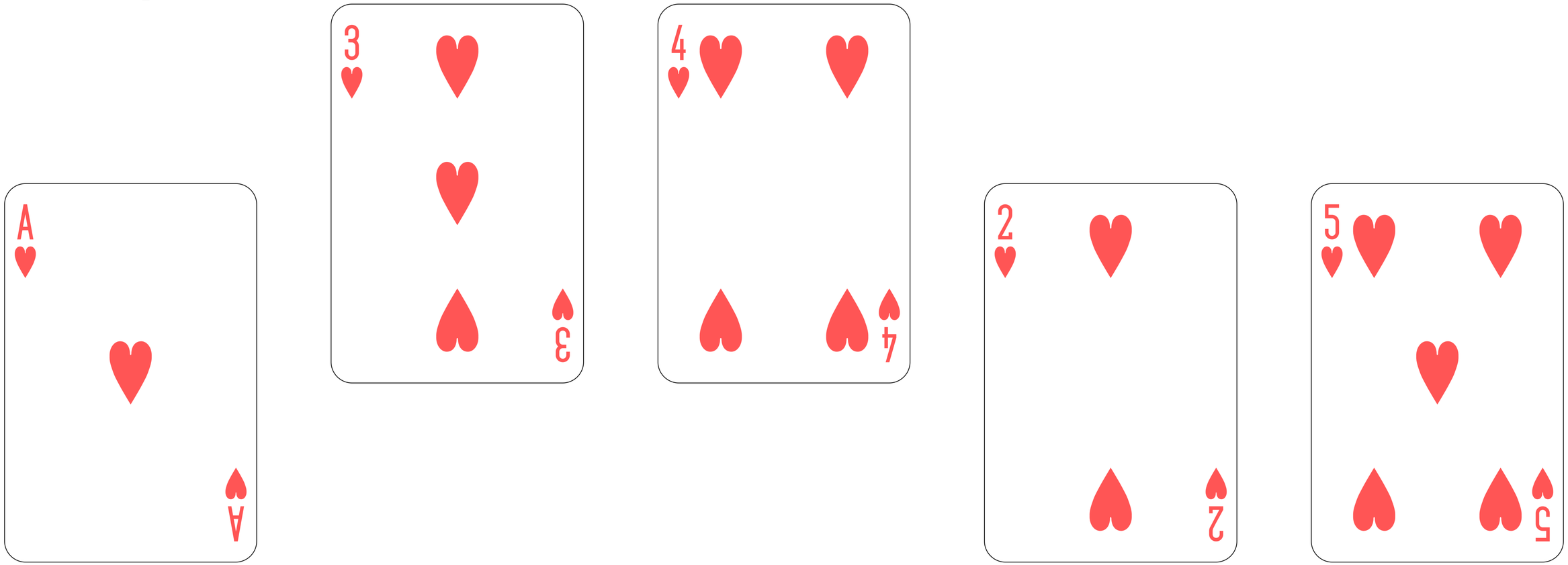
Sorting

Example 3



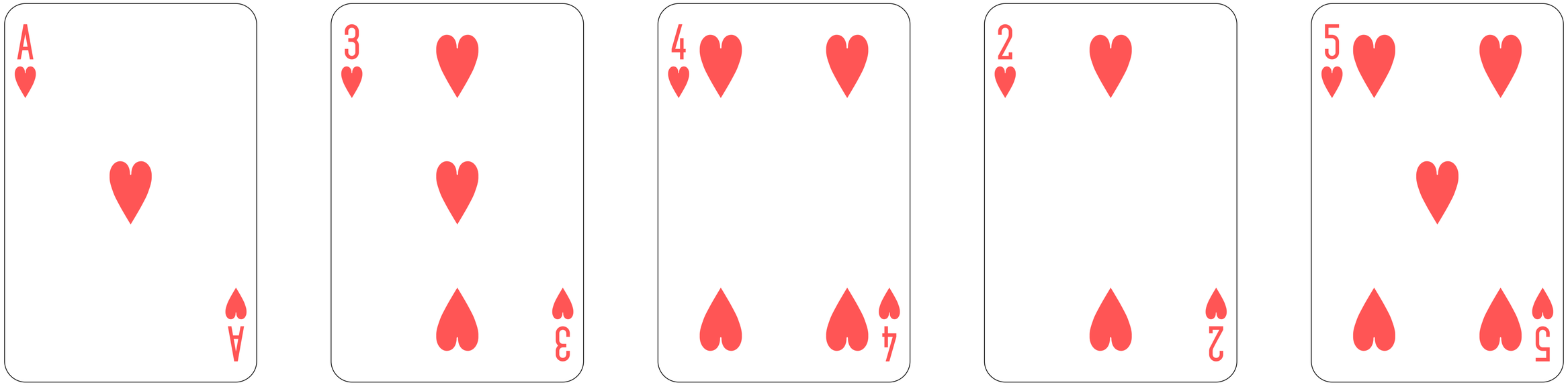
Sorting

Example 3



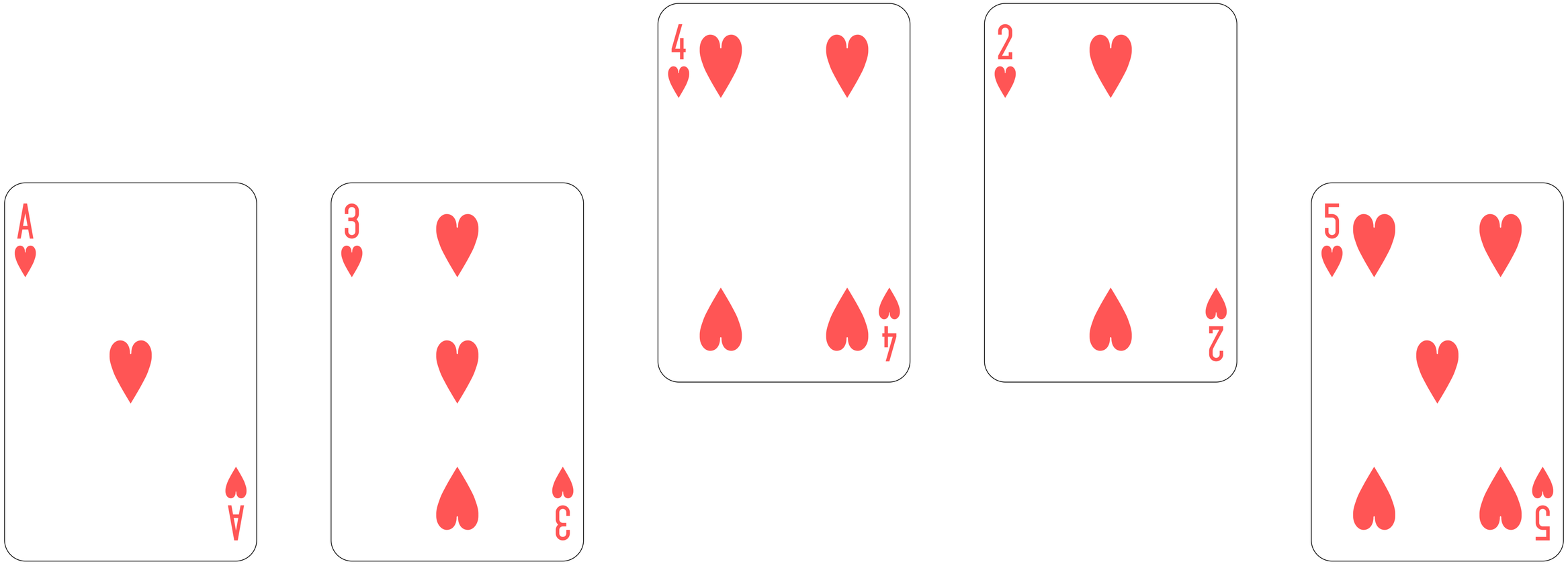
Sorting

Example 3



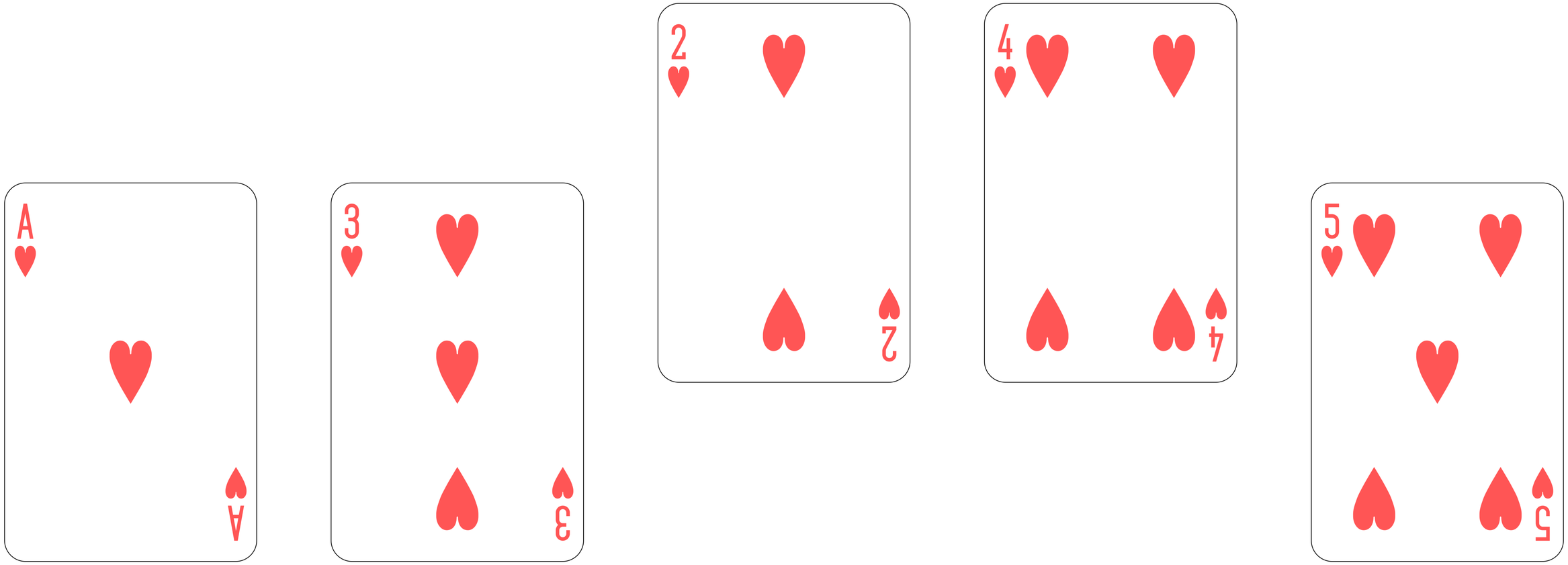
Sorting

Example 3



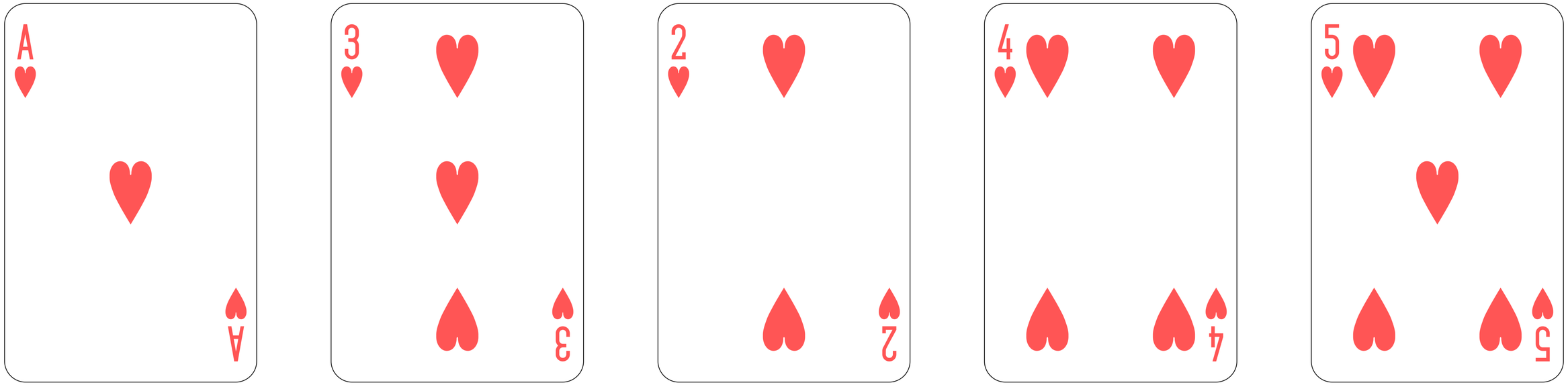
Sorting

Example 3



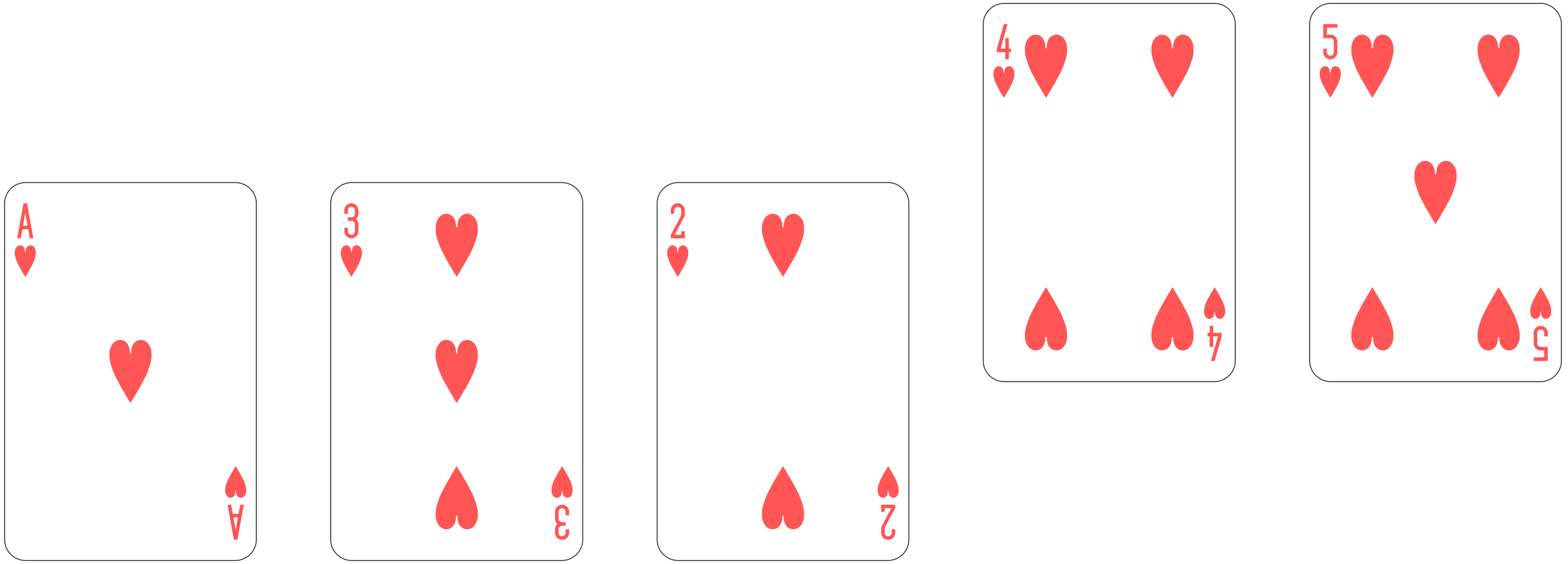
Sorting

Example 3



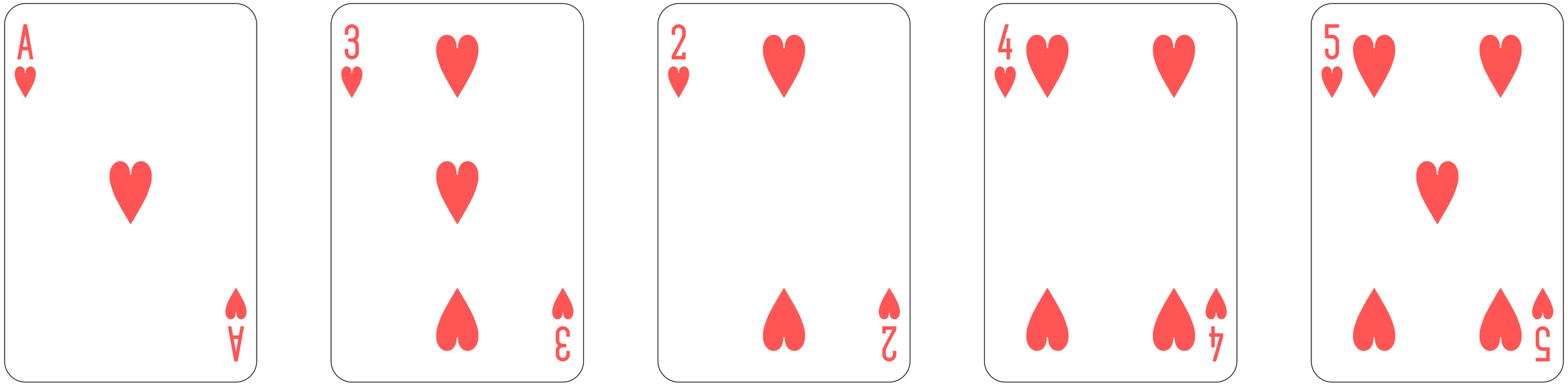
Sorting

Example 3



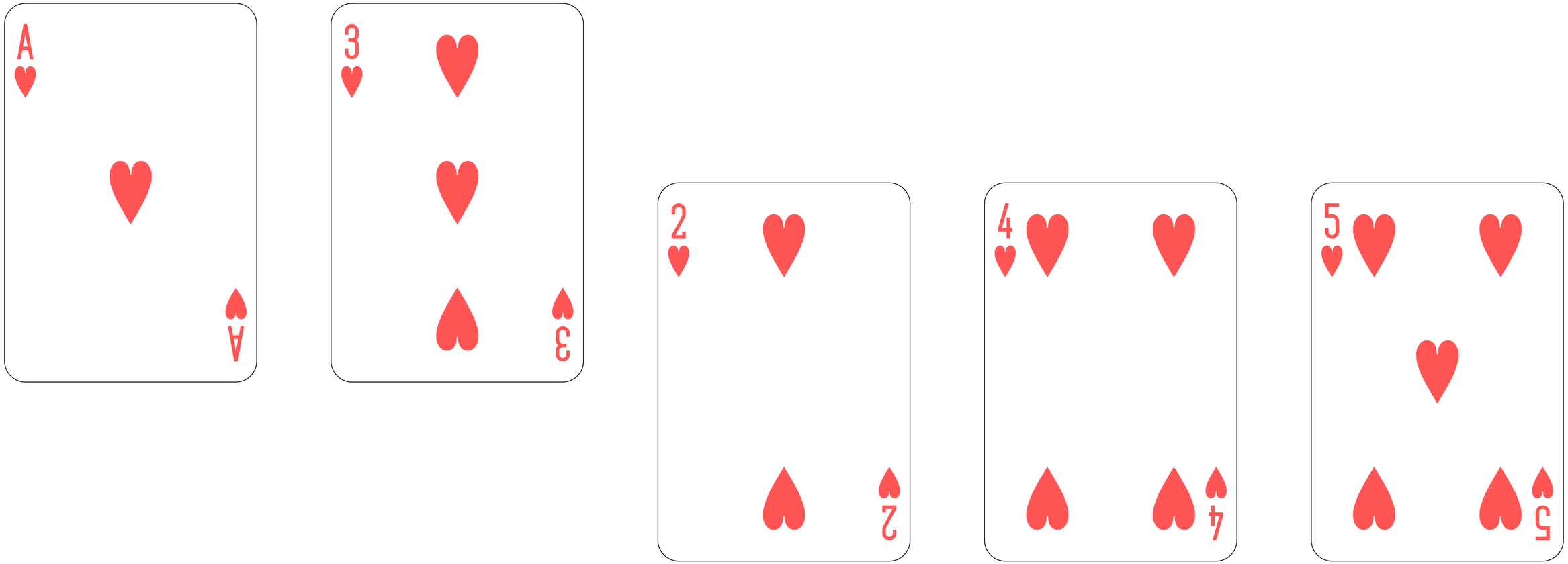
Sorting

Example 3



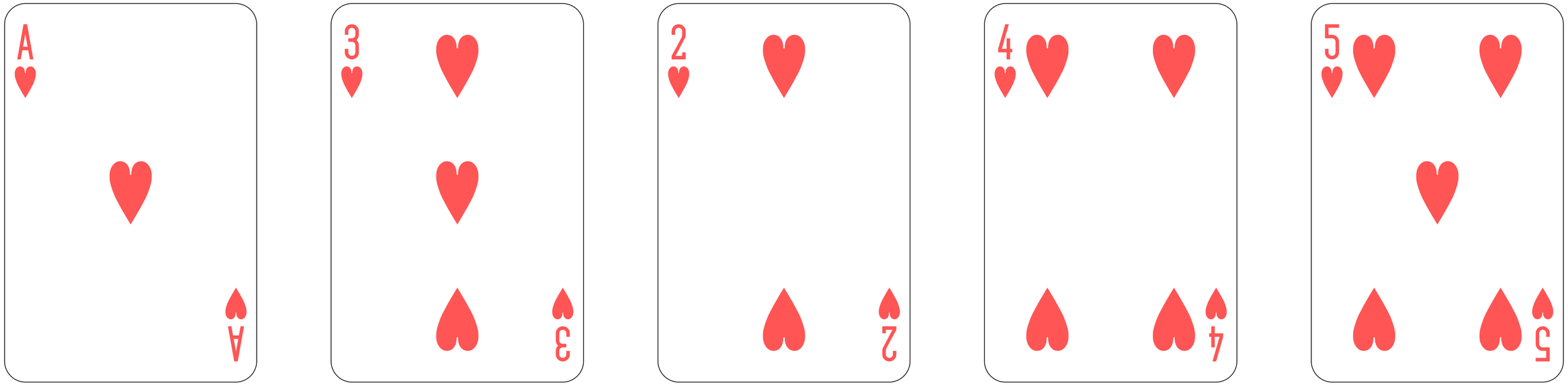
Sorting

Example 3



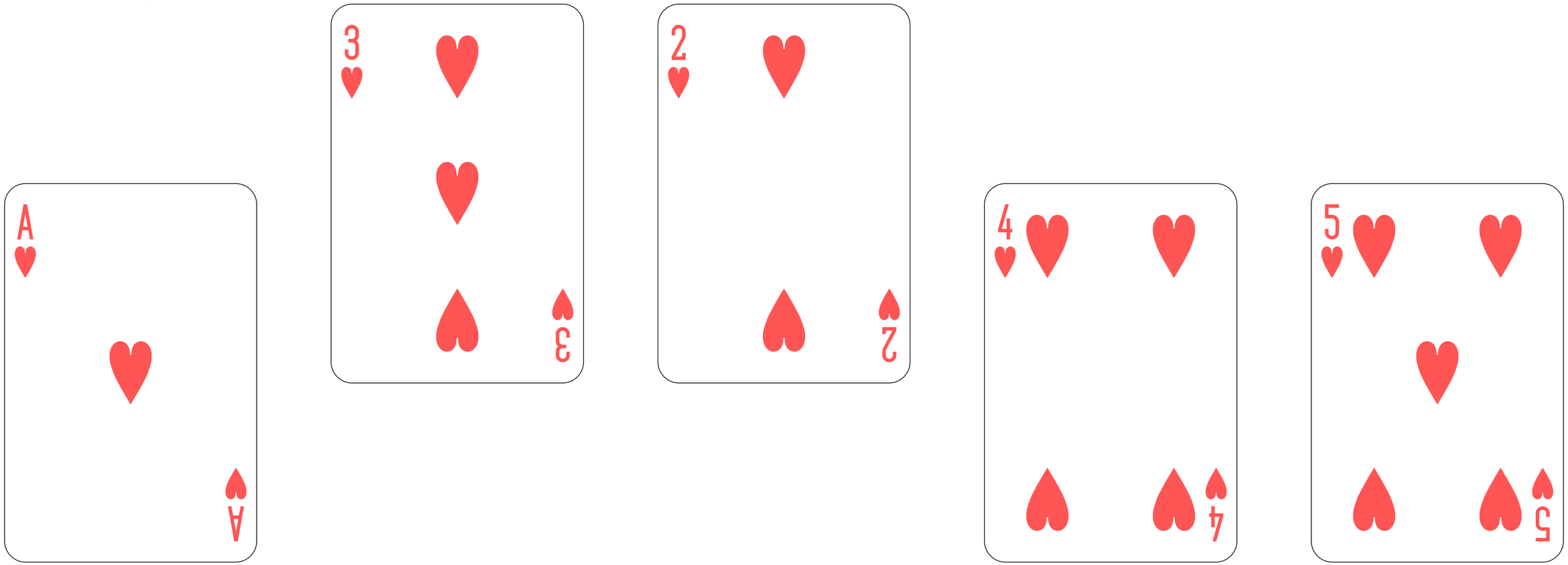
Sorting

Example 3



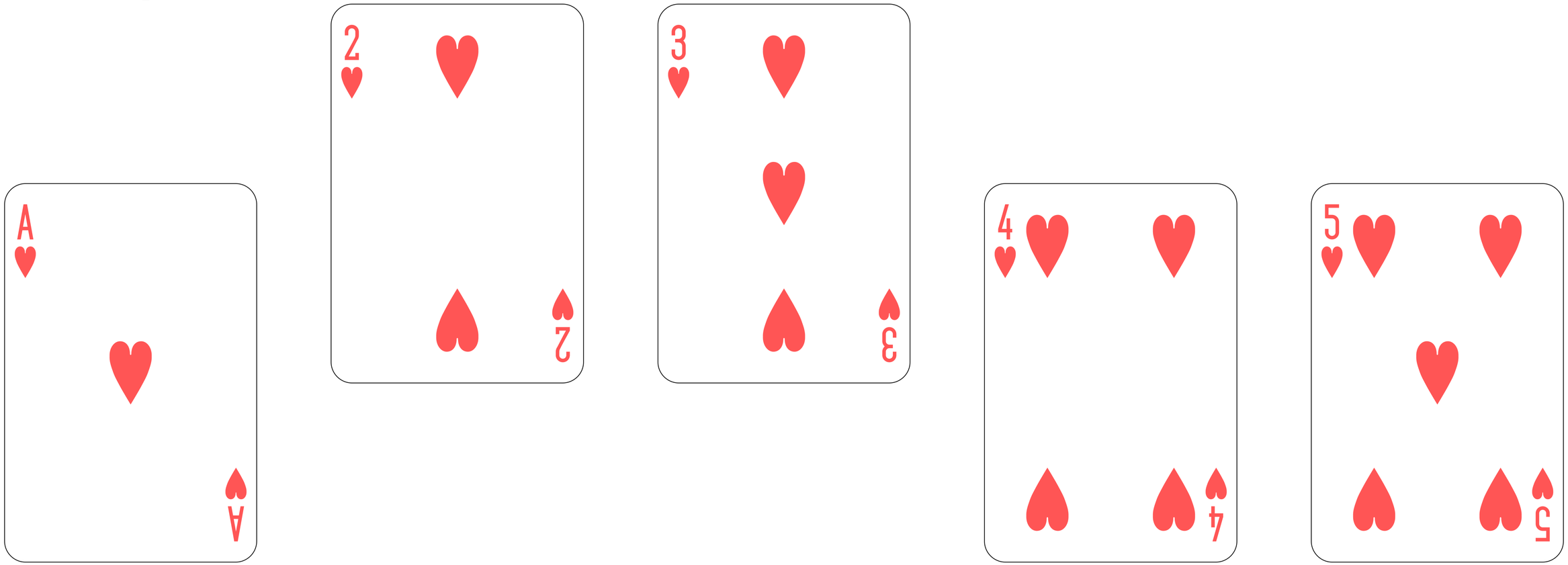
Sorting

Example 3



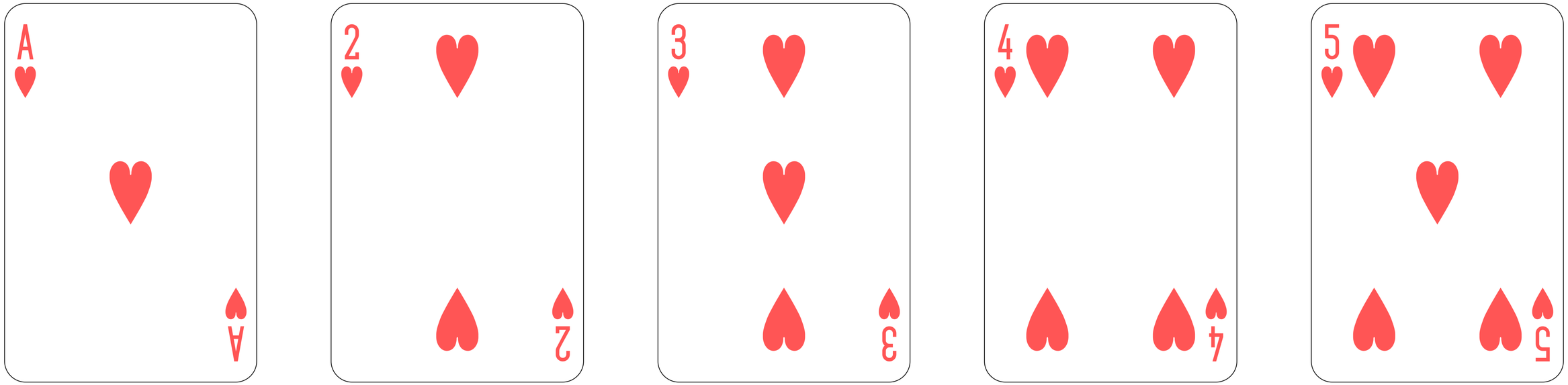
Sorting

Example 3



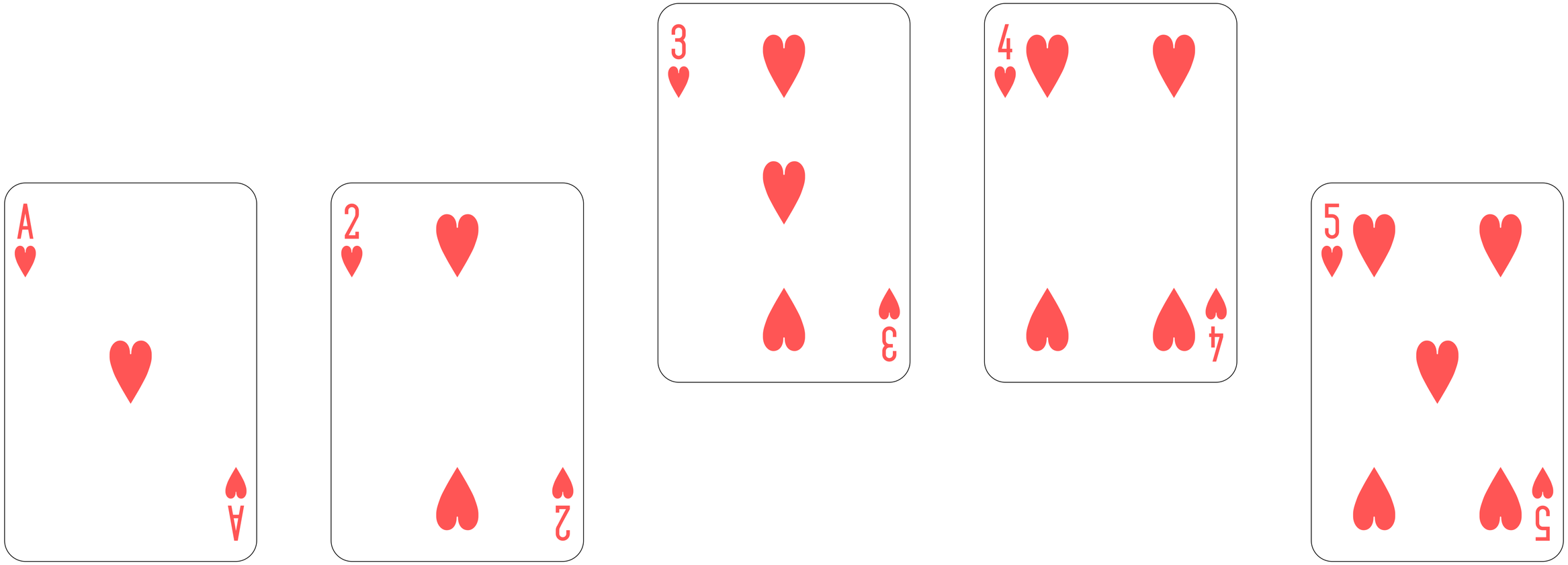
Sorting

Example 3



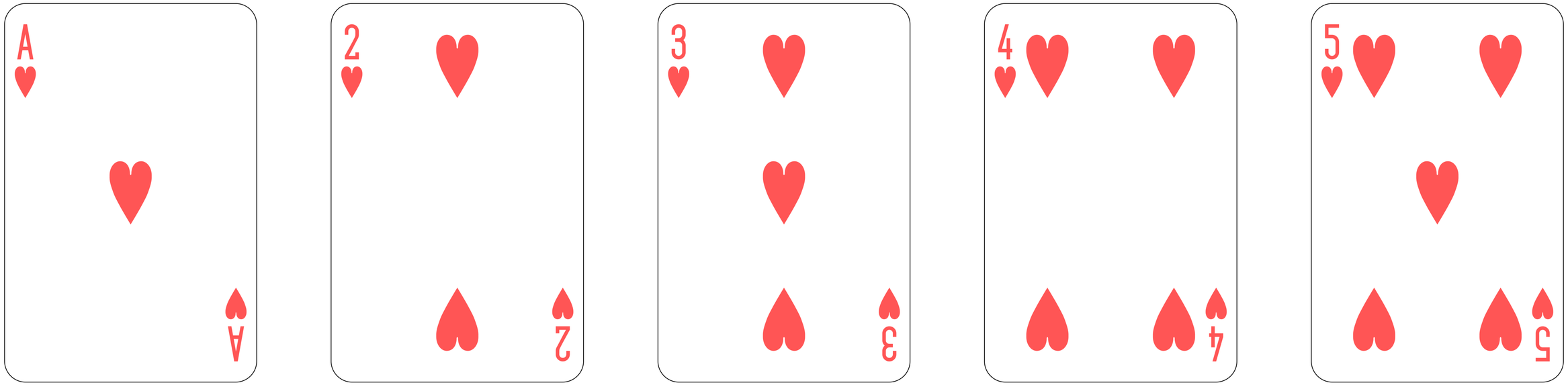
Sorting

Example 3



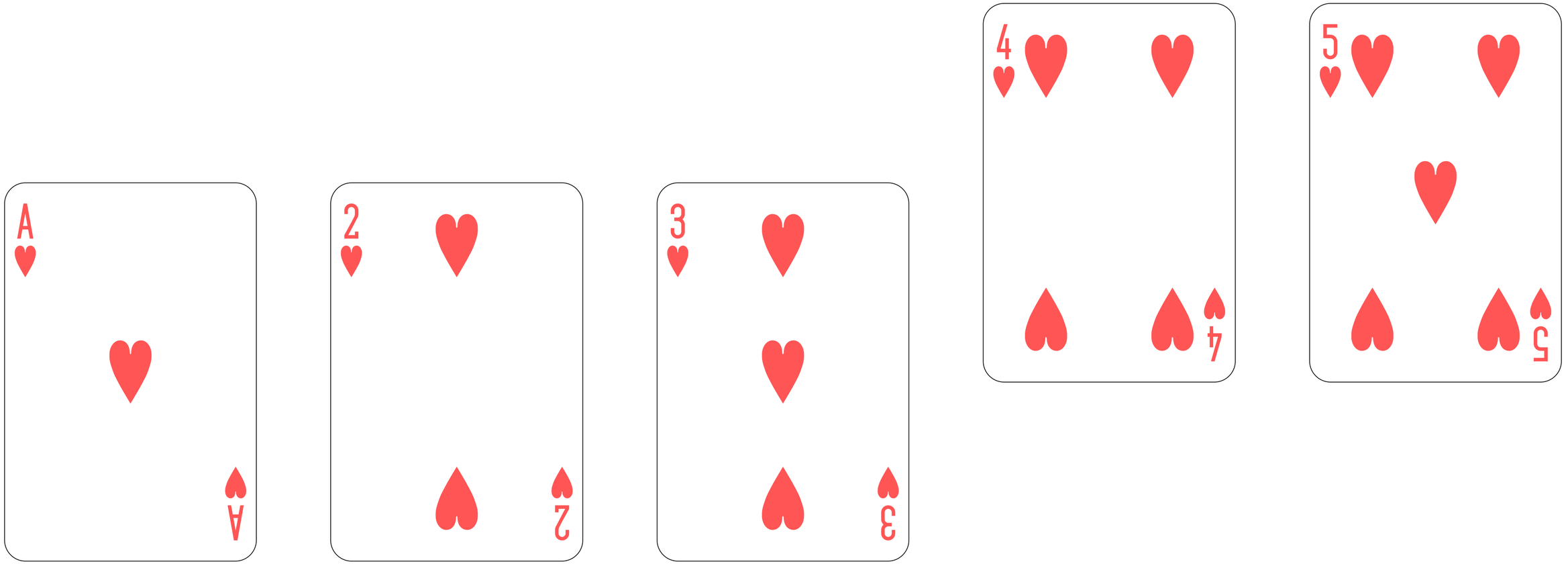
Sorting

Example 3



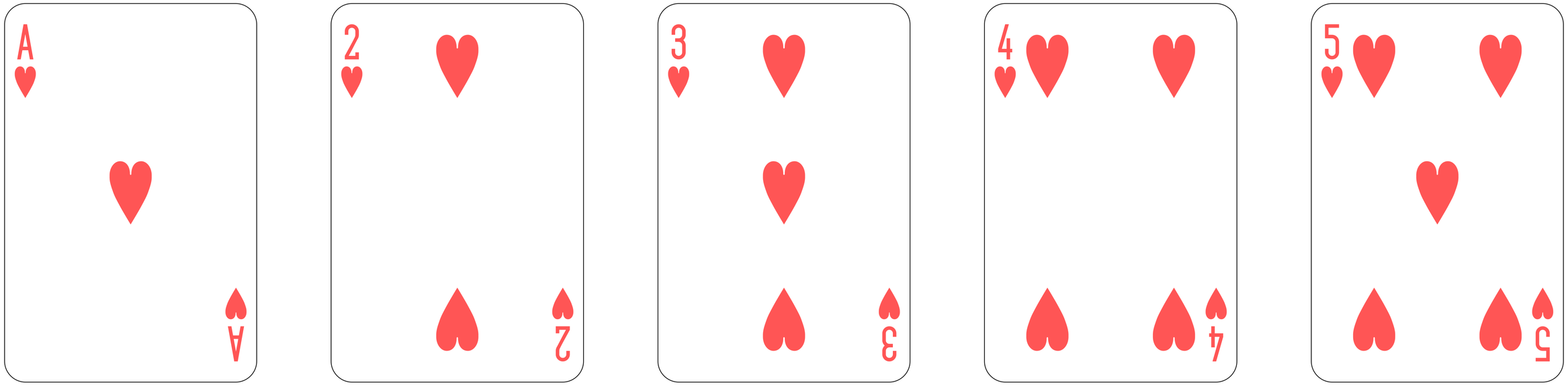
Sorting

Example 3



Sorting

Example 3



Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- Swap two adjacent elements if they are out of order

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- Swap two adjacent elements if they are out of order
- How many rounds do we need to sort the entire list?

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- Swap two adjacent elements if they are out of order
- How many rounds do we need to sort the entire list?
  - n. Why?

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- Swap two adjacent elements if they are out of order
- How many rounds do we need to sort the entire list?
  - n. Why?
  - Every round, the largest element is pushed to the right.

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- Swap two adjacent elements if they are out of order
- How many rounds do we need to sort the entire list?
  - n. Why?
  - Every round, the largest element is pushed to the right.
- $O(n^2)$ comparisons, $O(n^2)$ swaps

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- The largest element "bubbles" up.

Sorting

Example 3: Algorithm

```
while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```

- The largest element "bubbles" up.
- This is called *bubble sort*.

Sorting

Sorting

- Three $O(n^2)$ algorithms: Insertion sort, selection sort, bubble sort

Sorting

- Three $O(n^2)$ algorithms: Insertion sort, selection sort, bubble sort
- There are better algorithms

Sorting

- Three $O(n^2)$ algorithms: Insertion sort, selection sort, bubble sort
- There are better algorithms
  - We will revisit after learning about trees!

Sorting

- Three $O(n^2)$ algorithms: Insertion sort, selection sort, bubble sort
- There are better algorithms
  - We will revisit after learning about trees!
- It's a whole can of worms

Sorting

- Three $O(n^2)$
- There are be
  - We will re
- It's a whole

rt

Sort Benchmark Home Page

New: We are happy to announce the 2022 winners listed below. The new, 2022 records are listed in green. Congratulations to the winners!

Background

Until 2007, the sort benchmarks were primarily defined, sponsored and administered by Jim Gray. Following Jim's disappearance at sea in January 2007, the colleagues and sort benchmark winners. The Sort Benchmark committee members include:

- Chris Nyberg of Ordinal Technology Corp
- Mehul Shah of Aryn.ai
- George Porter of UC San Diego Computer Science & Engineering Dept

Top Results

	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Cloud	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.	2022, $0.97 / TB **Exoshuffle-CloudSort** 100 TB for $97 40 Amazon EC2 i4i.4xlarge nodes 1 Amazon EC2 r6i.2xlarge node Amazon S3 storage Frank Sifei Luan UC Berkeley Stephanie Wang UC Berkeley and Anyscale Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong UC Berkeley SangBin Cho, Eric Liang Anyscale Ion Stoica UC Berkeley and Anyscale
	2016, 37 TB	2016, 55 TB

Function Pointers

Function Pointers

Function Pointers

- Your code lives in memory too!

Function Pointers

- Your code lives in memory too!
- ...so they have addresses

Function Pointers

- Your code lives in memory too!
- ...so they have addresses
- ...so just like we have pointers to data, we have pointers to functions as well

Function Pointers

- Your code lives in memory too!
- ...so they have addresses
- ...so just like we have pointers to data, we have pointers to functions as well
- What's the point?

Function Pointers

- Your code lives in memory too!
- ...so they have addresses
- ...so just like we have pointers to data, we have pointers to functions as well
- What's the point?
  - We can pass functions around!

Function Pointers

Example

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is
  - A function pointer called `cmp`

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is
  - A function pointer called `cmp`
  - The function that `cmp` points to takes two `void *` and returns `int`

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is
  - A function pointer called `cmp`
  - The function that `cmp` points to takes two `void *` and returns `int`
  - It tells the sorting function how to compare two arbitrary elements

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is
  - A function pointer called `cmp`
  - The function that `cmp` points to takes two `void *` and returns `int`
  - It tells the sorting function how to compare two arbitrary elements
  - (negative if 1 < 2, 0 if 1 == 2, positive if 1 > 2)

Function Pointers

Example

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }

        if (!swapped) {
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }

        if (!swapped) {
            break;
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }

        if (!swapped) {
            break;
        }
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }

        if (!swapped) {
            break;
        }
    }
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
    for (;;) {
        int swapped = 0;
        for (int i = 0; i < l->length; i++) {
            if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                void *tmp = l->elems[i];
                l->elems[i] = l->elems[i + 1];
                l->elems[i + 1] = tmp;
                swapped = 1;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *));

int strcmp_wrapper(void *s1, void *s2) {
    return strcmp(s1, s2);
}

int main(void) {
    struct alist l;
    alist_sort(&l, &strcmp_wrapper);
    return 0;
}
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *));

int strcmp_wrapper(void *s1, void *s2) {
    return strcmp(s1, s2);
}

int main(void) {
    struct alist l;
    alist_sort(&l, &strcmp_wrapper);
    return 0;
}
```

^ optional