Sorting

CS143: lecture 8

Byron Zhong, June 28

`i++`, `++i`, and Makefile

i++, ++i

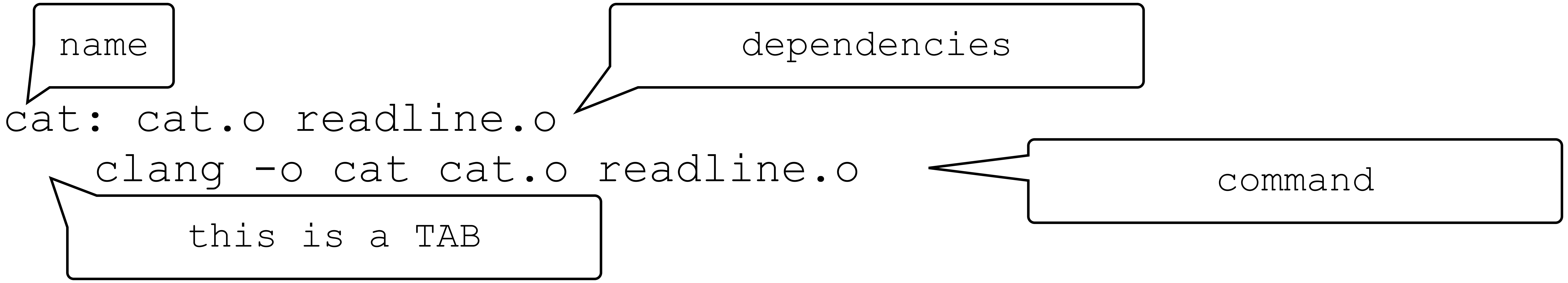
- i++, looks up i, give you the value, and +1 to i.
- ++i, looks up i, +1 to i, and give you the value.

```c
int x = 5;
printf("%d\n", x++);
printf("%d\n", ++x);
```

```
5
7
```

Makefile

- Makefile is comprised of rules.



```
cat: cat.o readline.o
	clang -o cat cat.o readline.o
```

- `make cat`
  - if cat.o and readline.o have changed, do the command
  - if not, do nothing
- But how do you make cat.o and readline.o?

Makefile

- Rules can contain patterns.

```
%.o: %.c
  clang -o $@ -c $^
```

`$^: the dependencies`

`$@: this rule's name`

- To get anything .o, run the following command, depending on that .c
- `make readline.o`
  - if readline.c has changed, do the command; otherwise, do nothing
- One rule can depend on other rules; `make` will figure out the order

Makefile

- You can also define variables in Makefile.

```
CC      = clang
CFLAGS  += -g -Wall -Wextra -Werror -pedantic -std=c11
LDFLAGS += -g

cat: cat.o readline.o
    $(CC) $(LDFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -o $@ -c $^
```

`$ to replace CC with the variable`

Separate Compilation

How C is actually compiled

file1.c
preprocessor
file1.c
compiler
file1.s
assembler
file1.o
linker

- removes comments
- execute the directives
    - finds the files to `#include`
    - replace the text with `#define`
    - `#if`

- Convert the C program into assembly
- Tokenize, parse, optimize, ...

- Convert the assembly code into machine-readable *object file*
- All functions are converted into binary

Separate Compilation

How C is actually compiled

file1.c
preprocessor
compiler
assembler
file1.o
file2.c
preprocessor
compiler
assembler
file2.o
. . .
. . .
. . .
linker
./exec

- every .o file provides some functions
- linker links all the functions together
- finds main

Separate Compilation

- Demo

Sorting

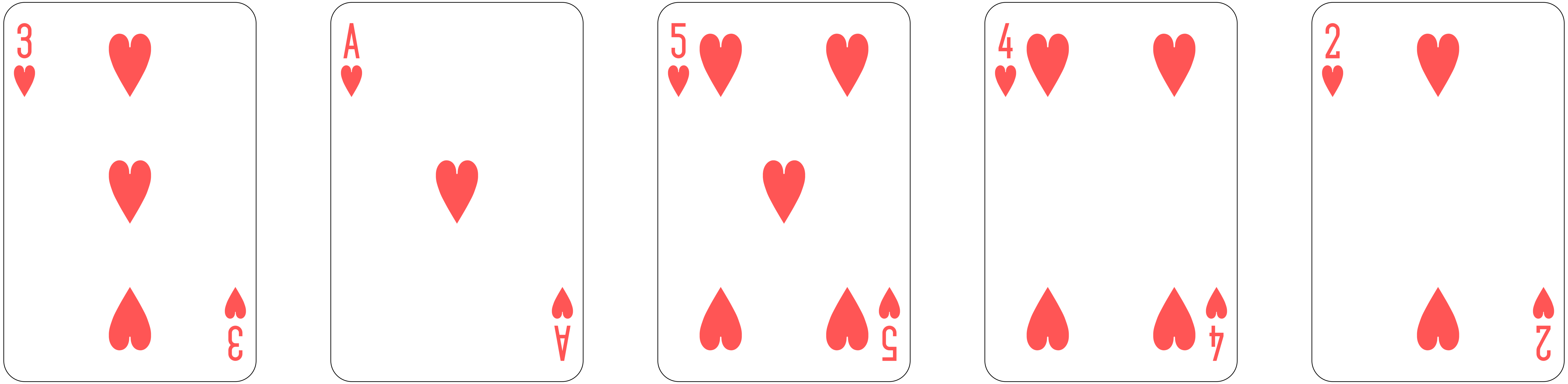
Sorting

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

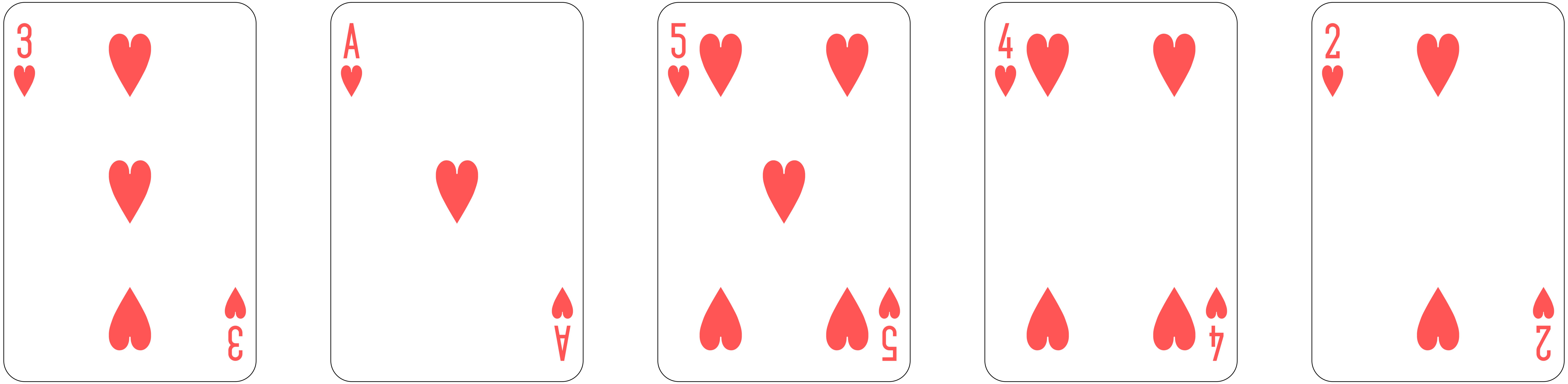
Example



Sorting

Example 1 (how I do it)

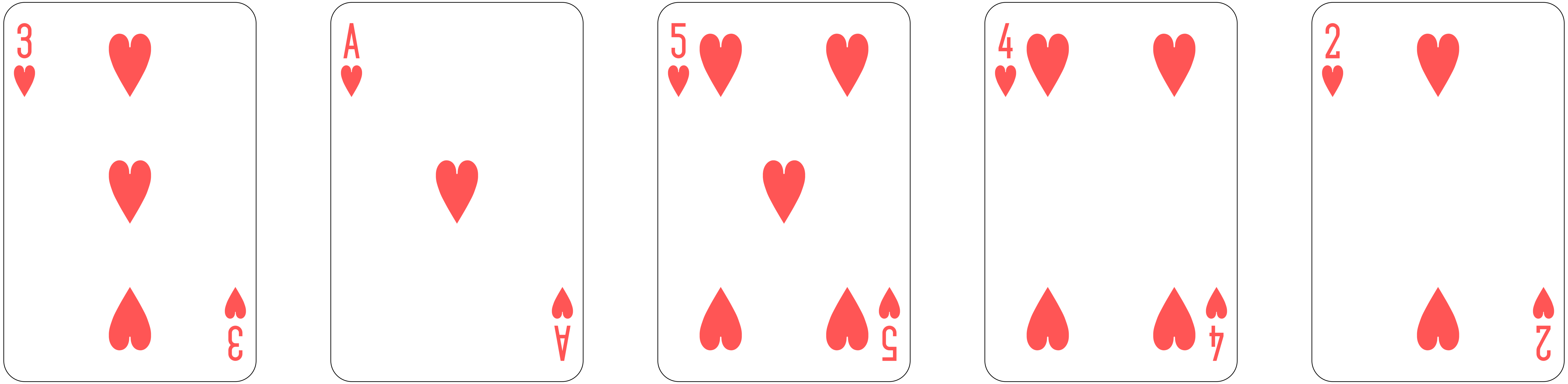
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

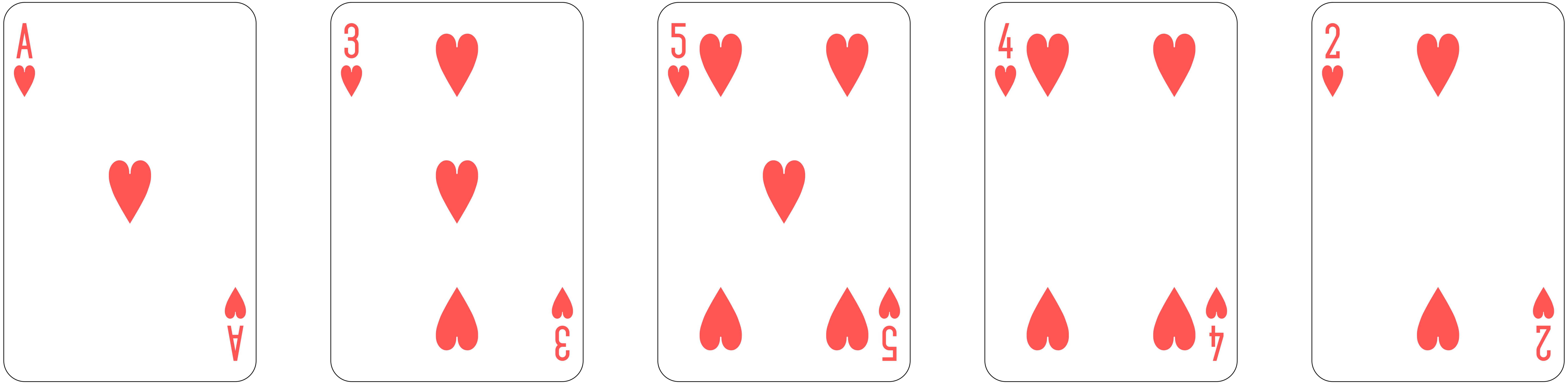
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

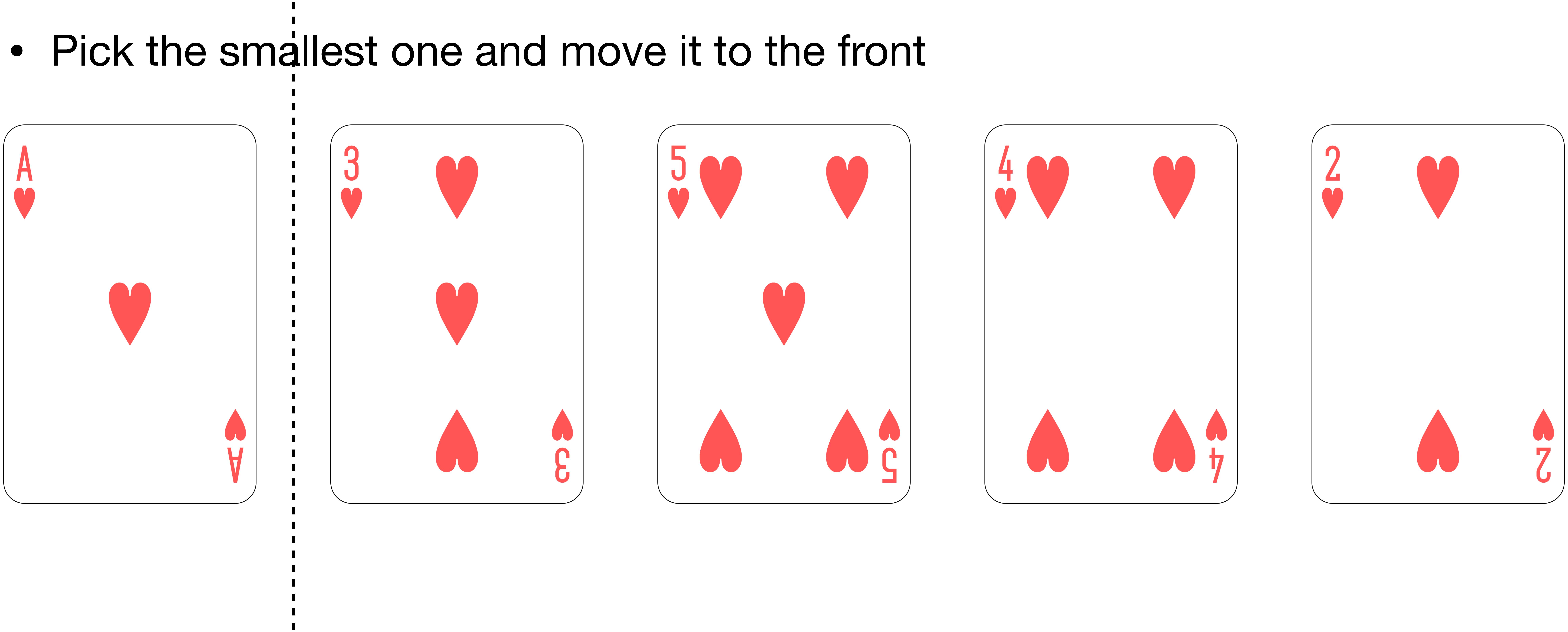
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

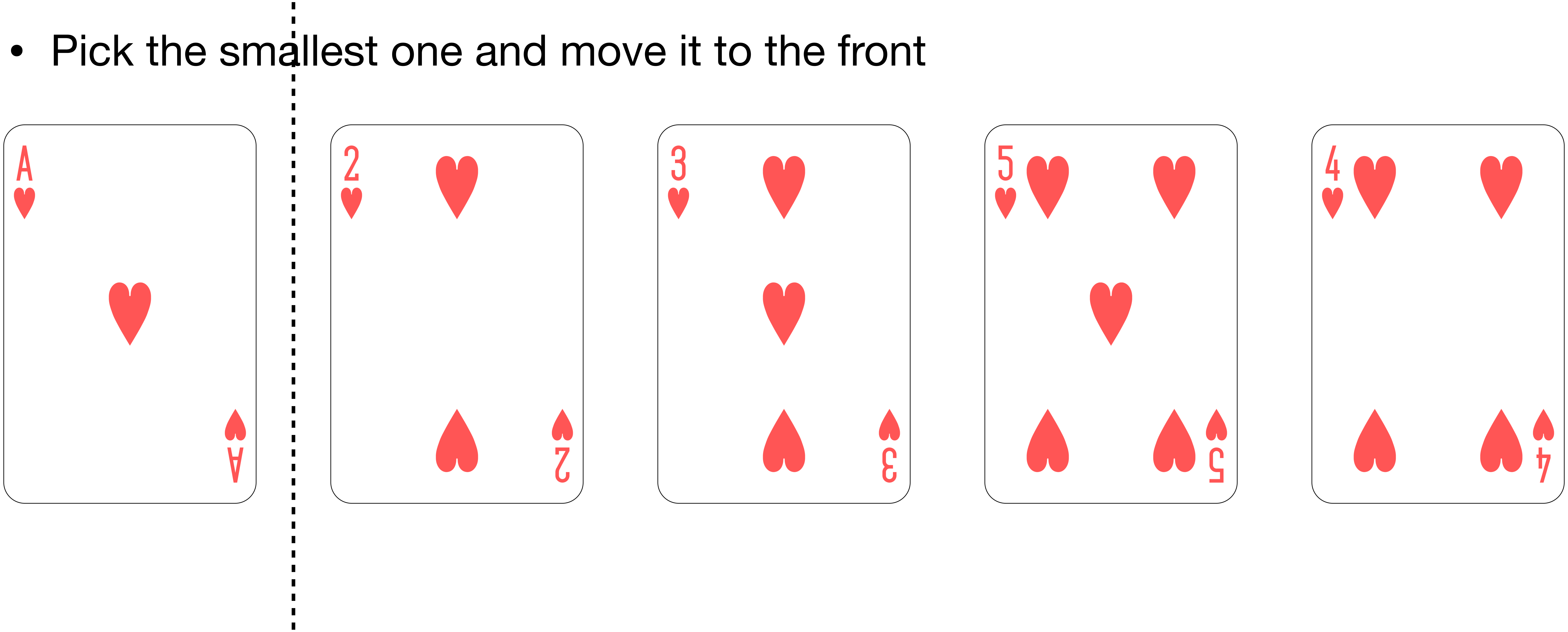
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

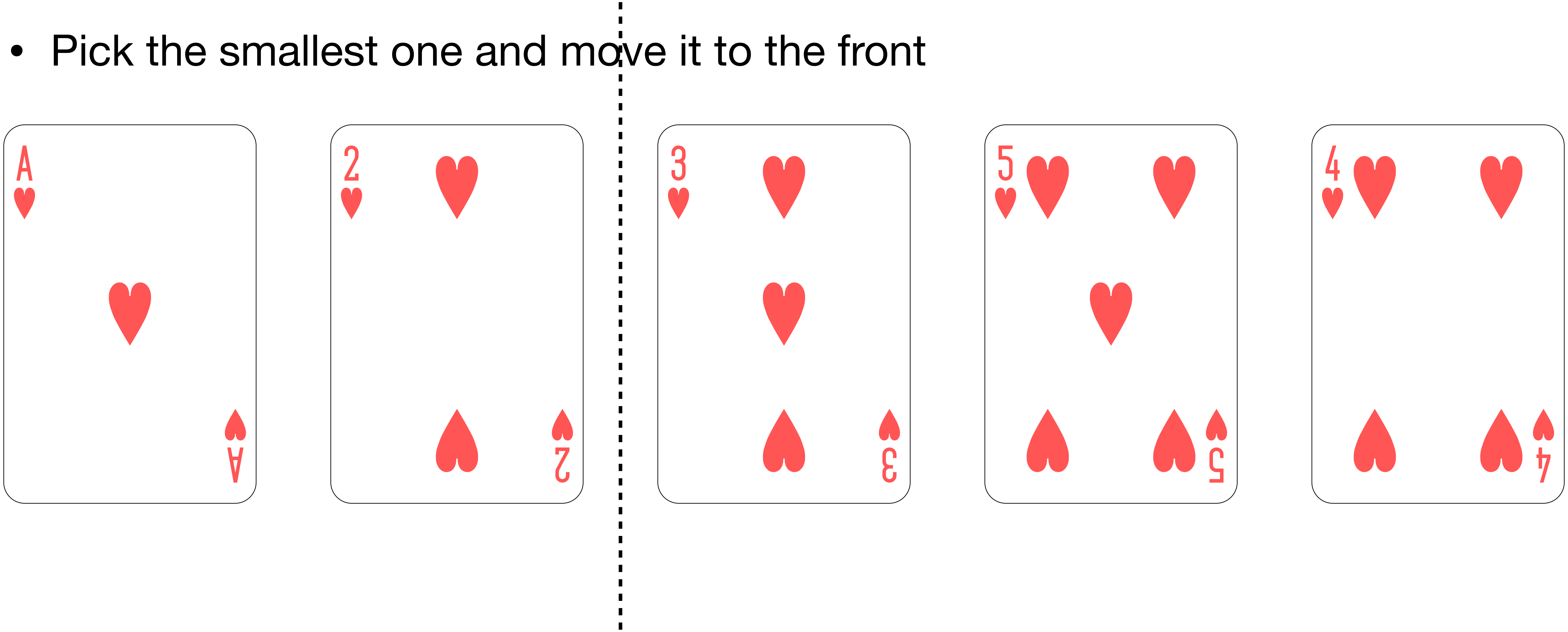
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

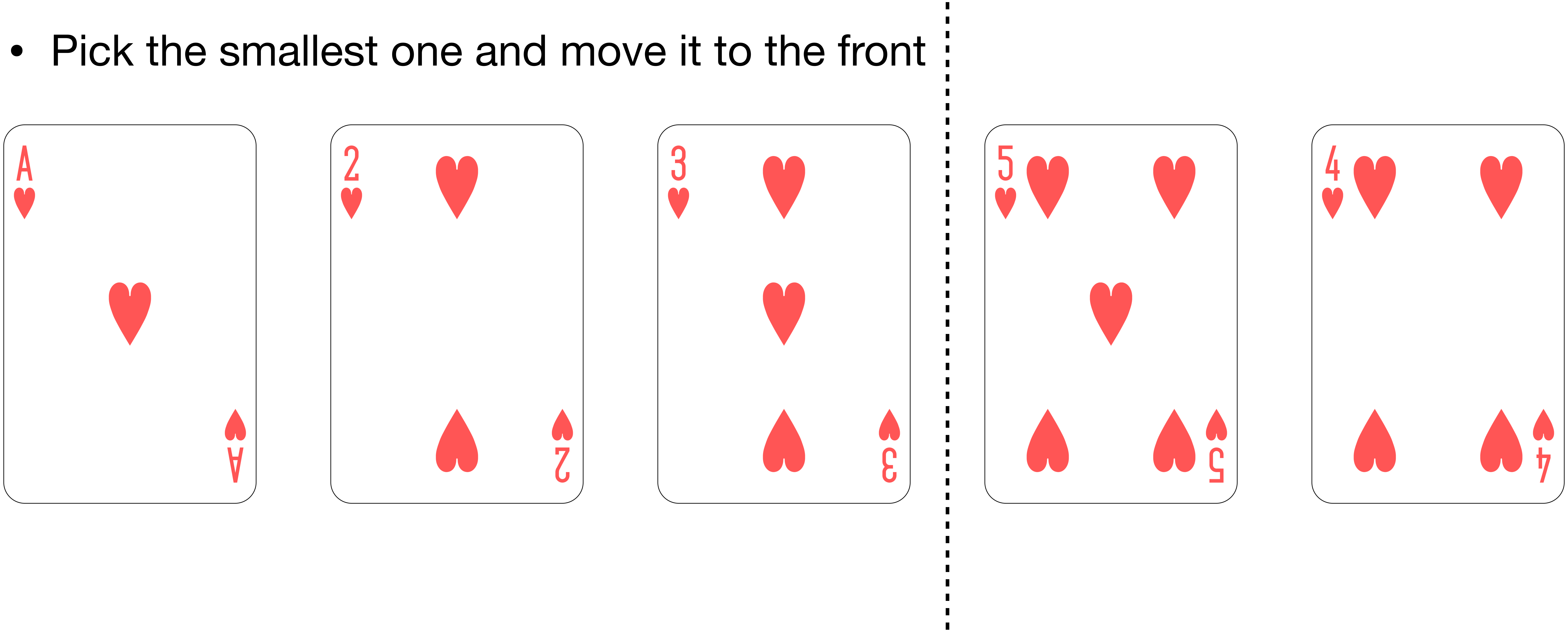
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

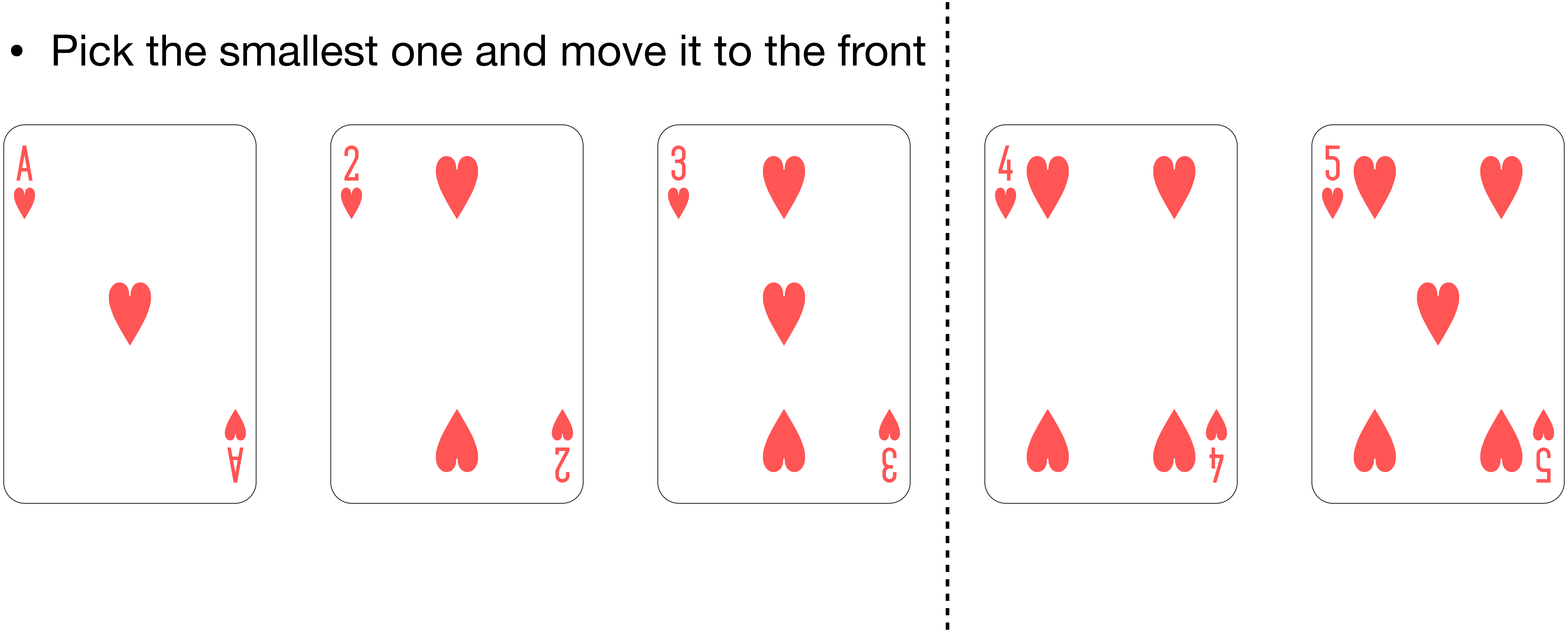
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

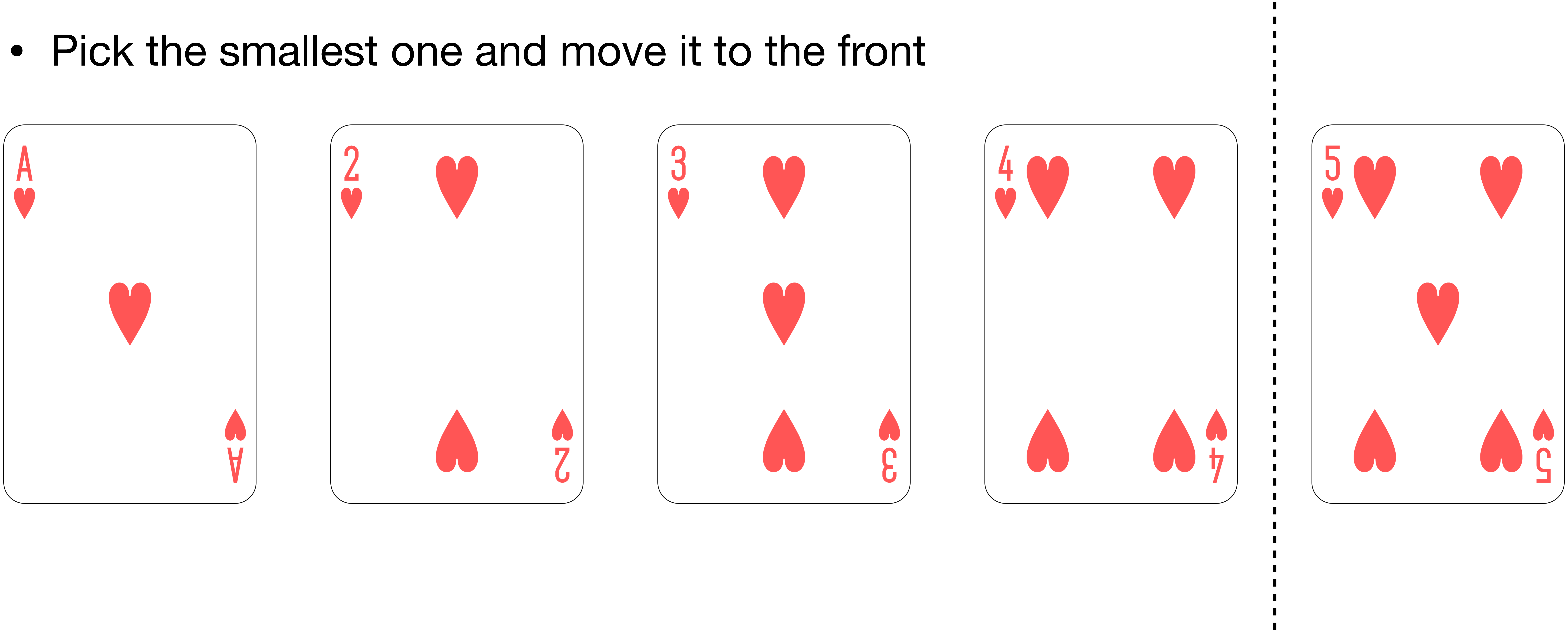
- Pick the smallest one and move it to the front



Sorting

Example 1 (how I do it)

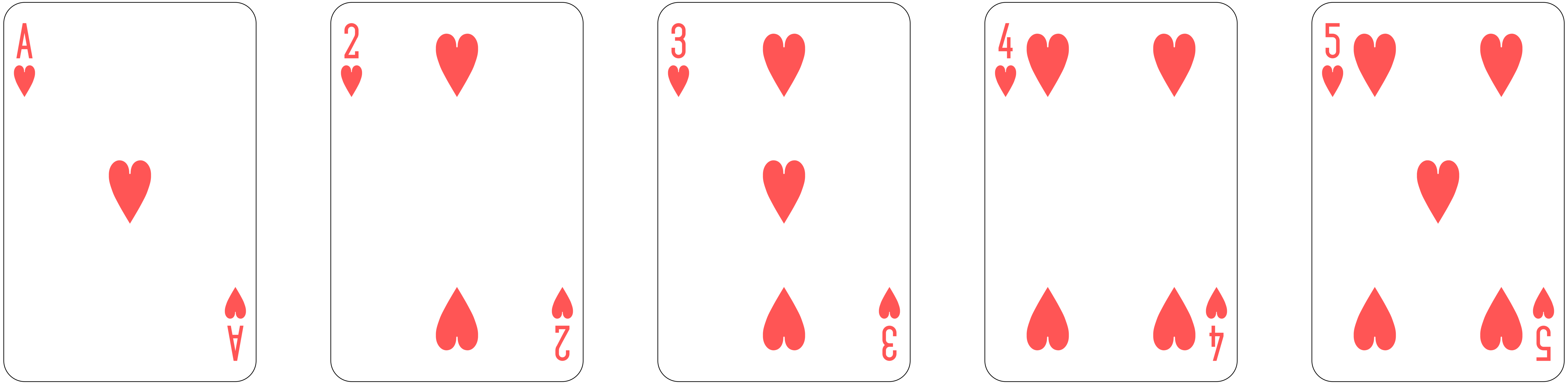
- Pick the smallest one and move it to the front



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 + 1 : 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 + 1 : 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

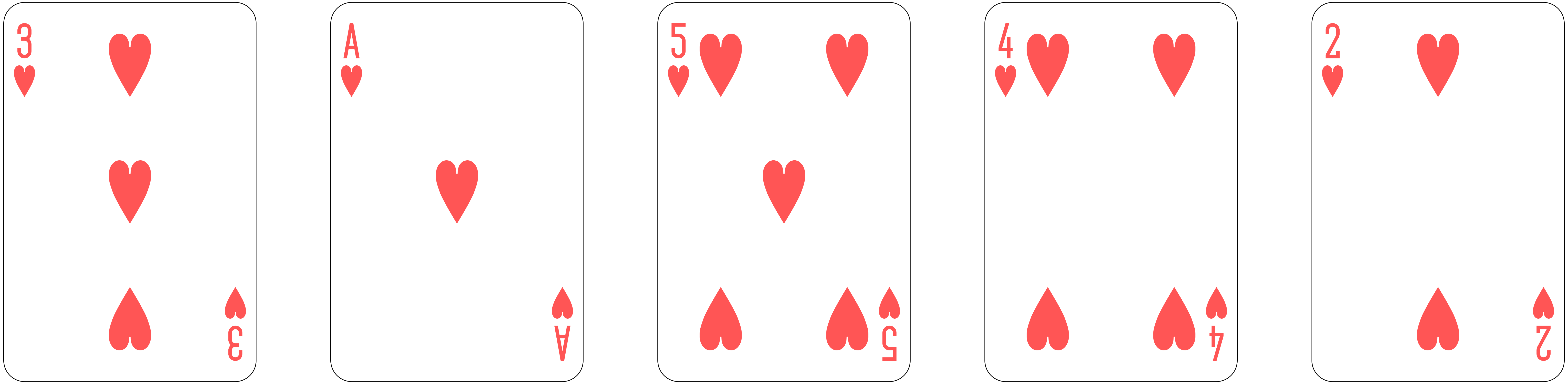
Example 1 (how I do it): Algorithm

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

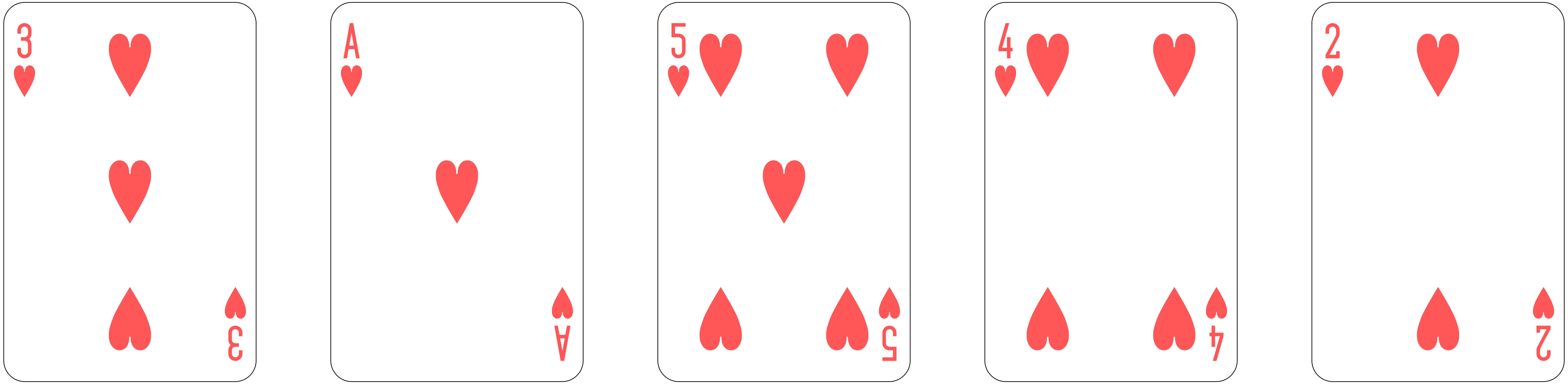
Example 2



Sorting

Example 2

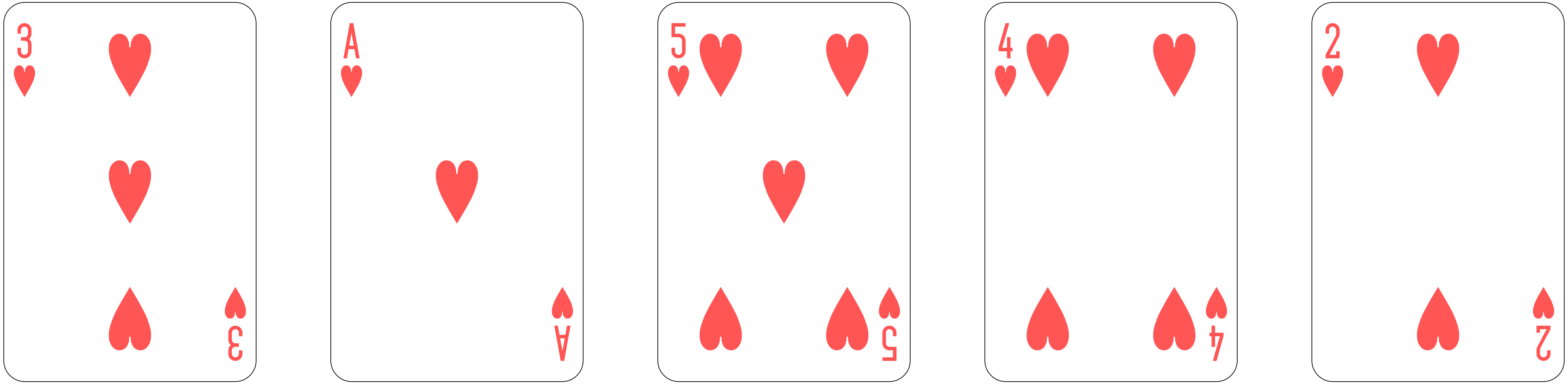
- Pick the first unsorted and insert it into the right place



Sorting

Example 2

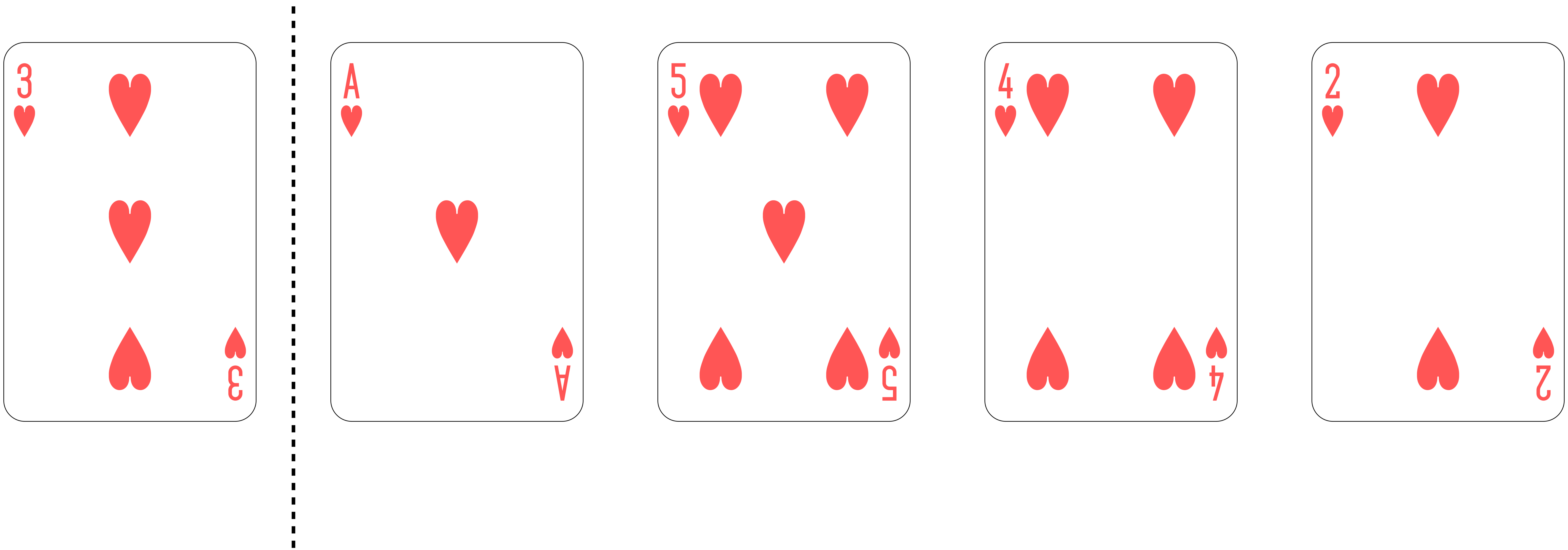
- Pick the first unsorted and insert it into the right place



Sorting

Example 2

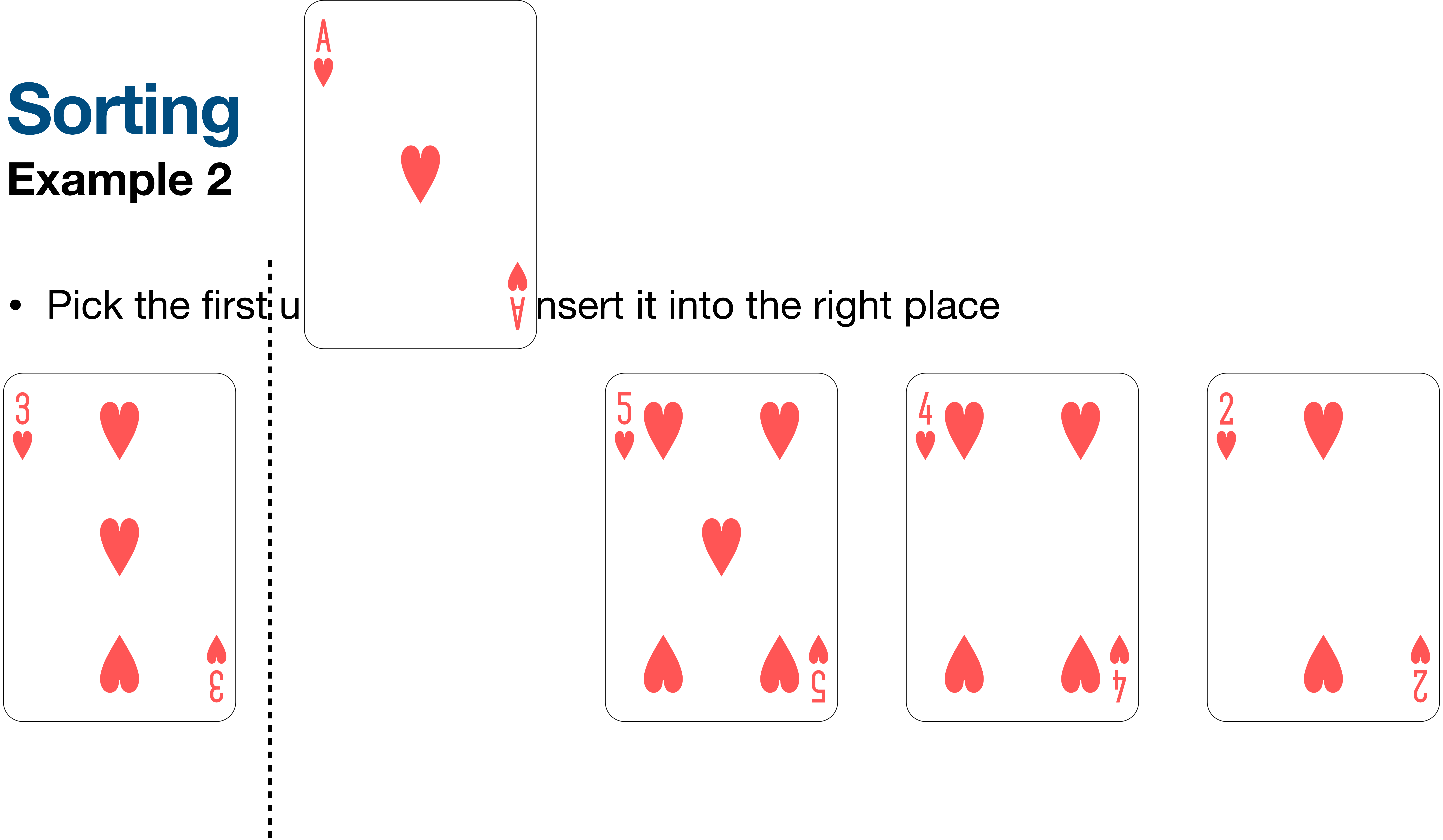
- Pick the first unsorted and insert it into the right place

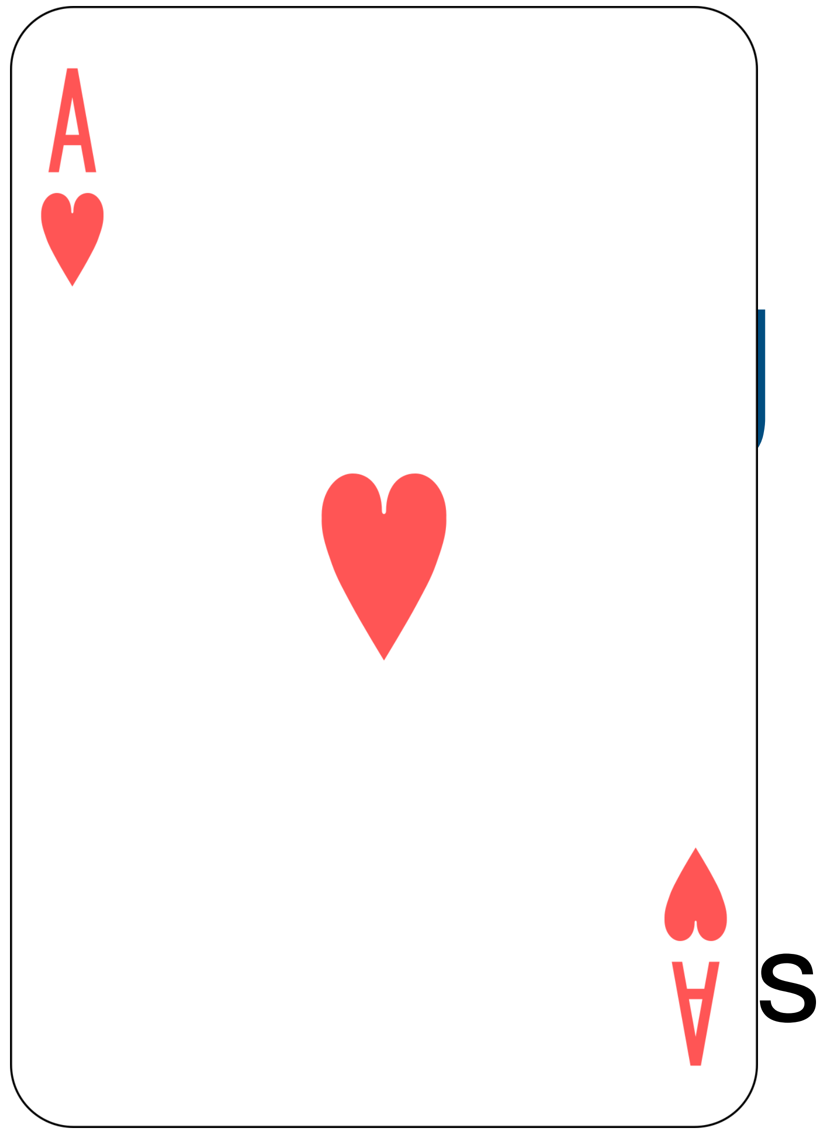


Sorting

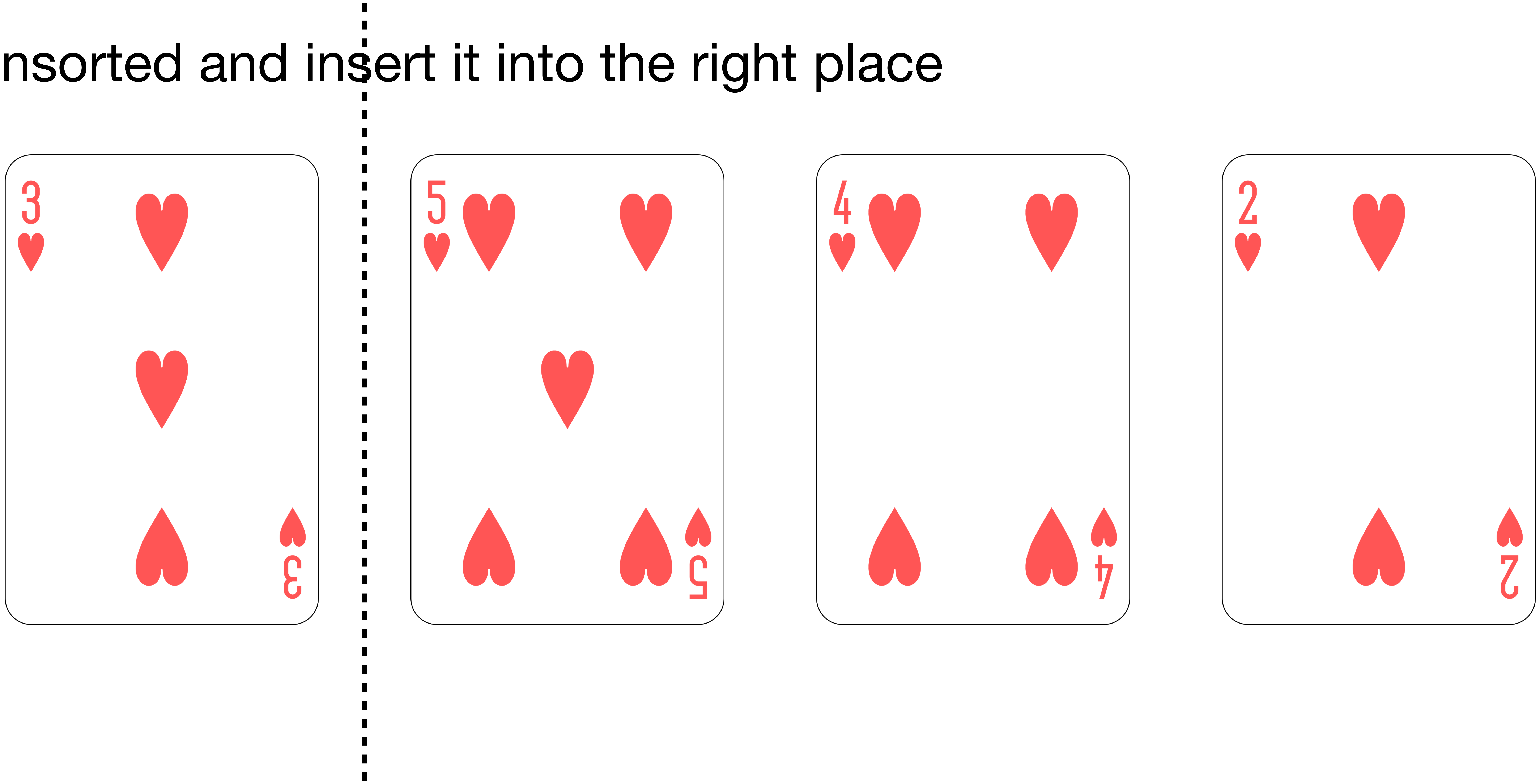
Example 2

- Pick the first u nsert it into the right place





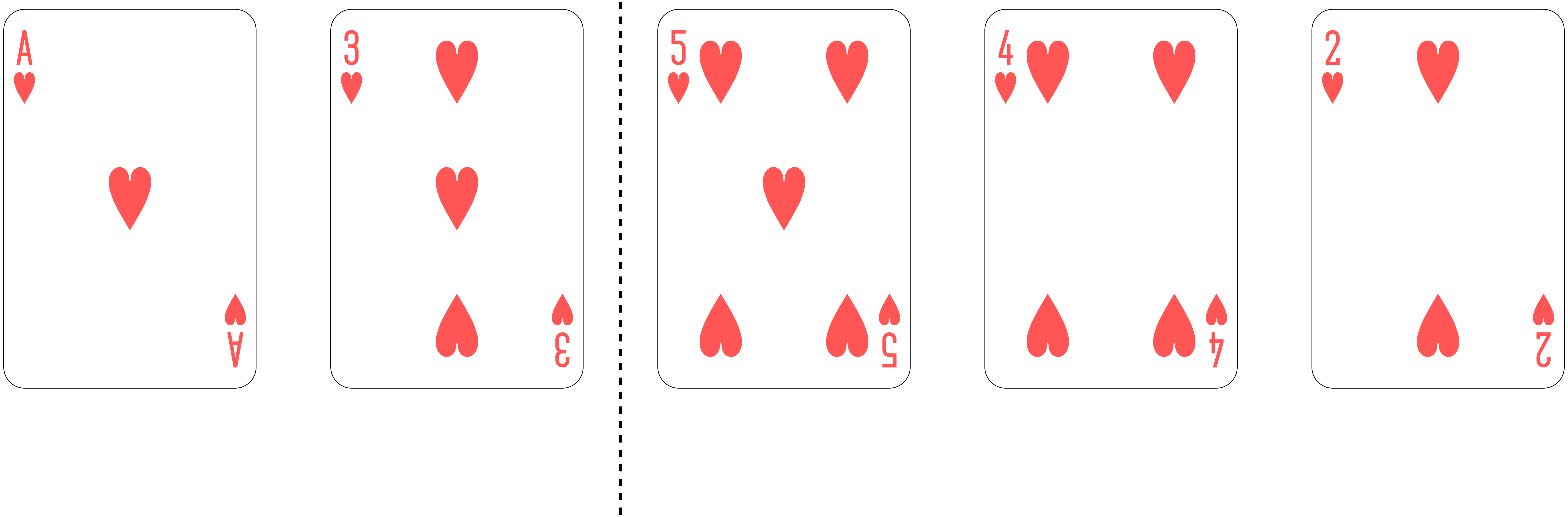
st unsorted and insert it into the right place



Sorting

Example 2

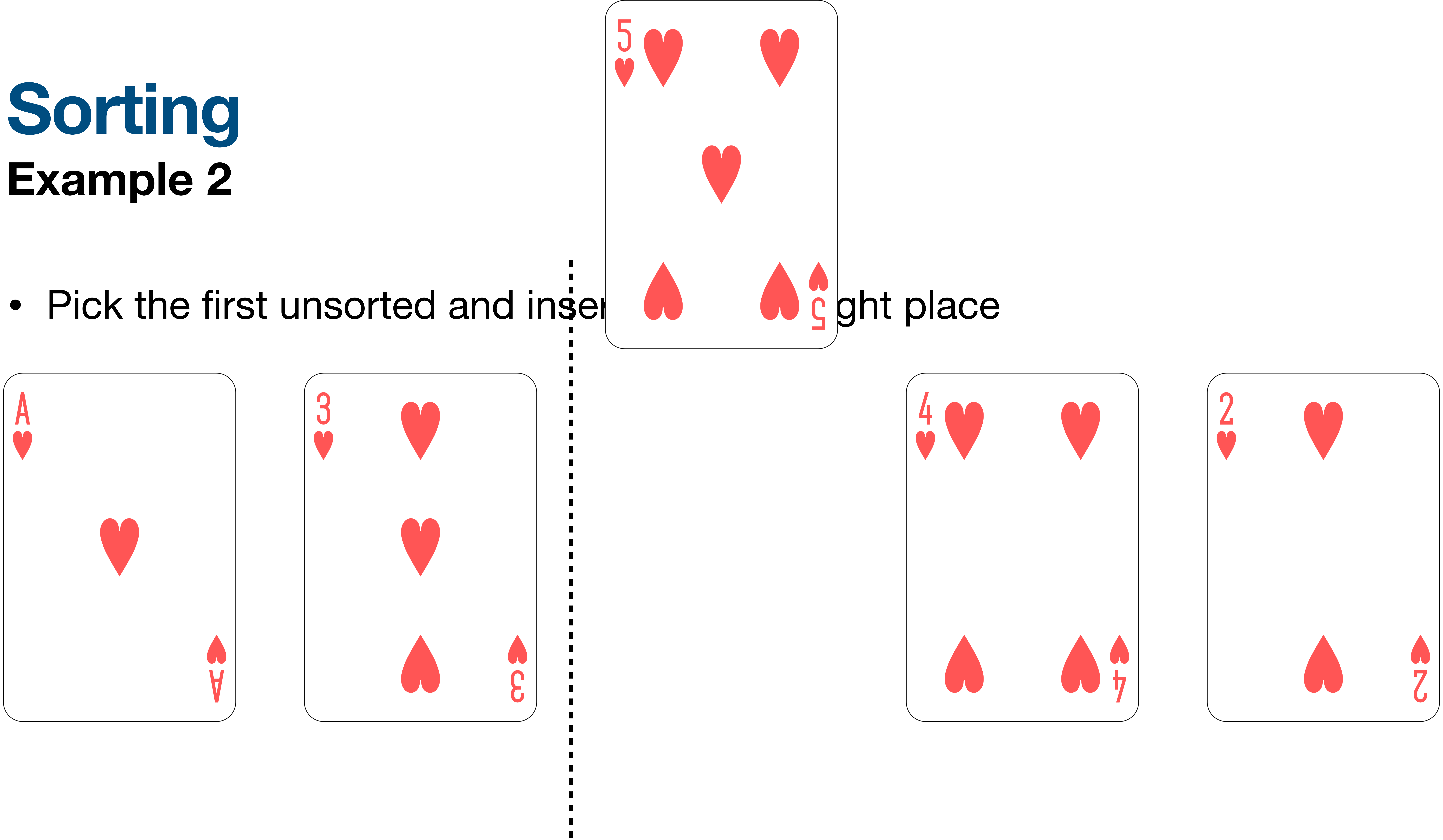
- Pick the first unsorted and insert it into the right place



Sorting

Example 2

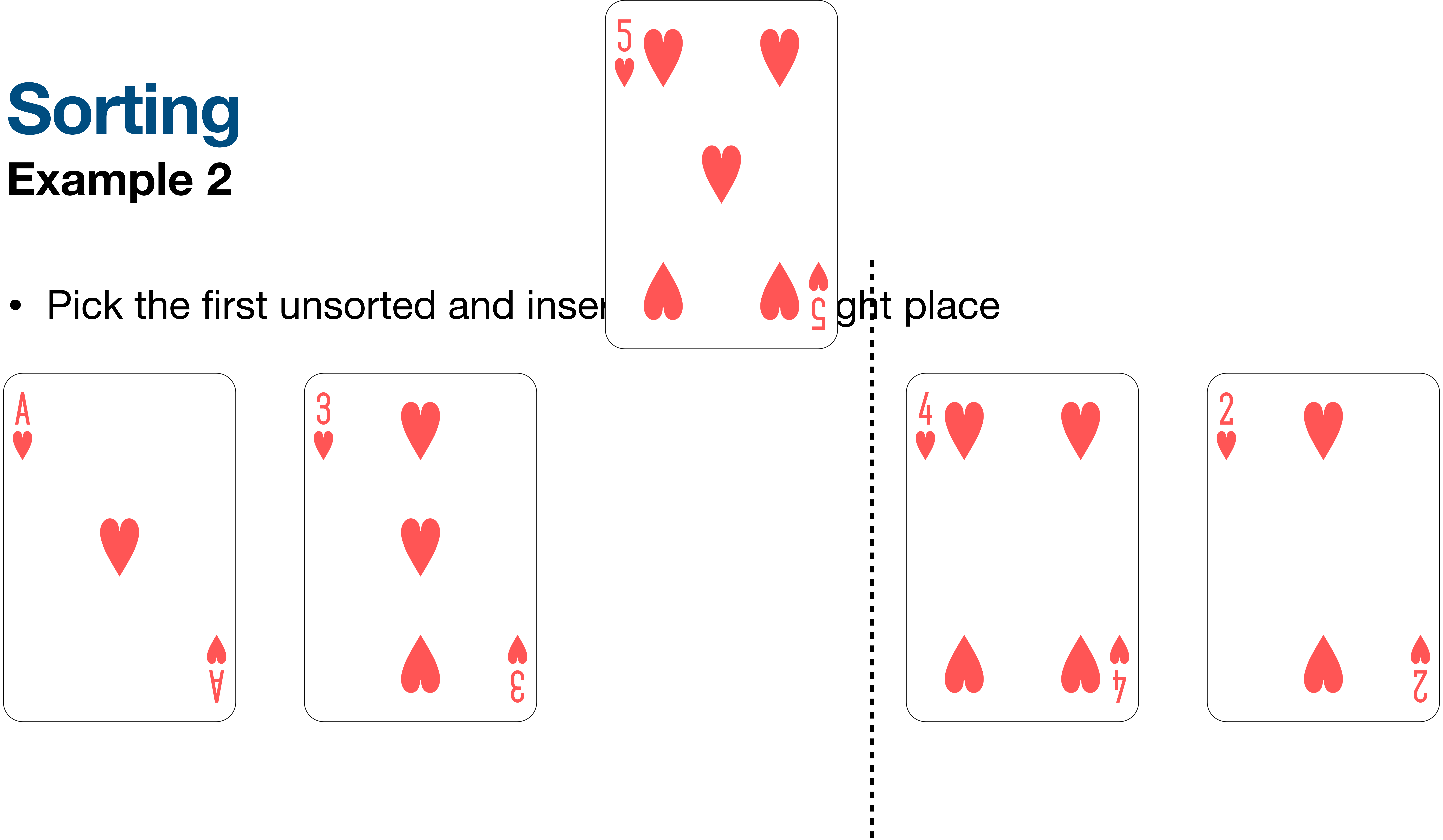
- Pick the first unsorted and inser ght place



Sorting

Example 2

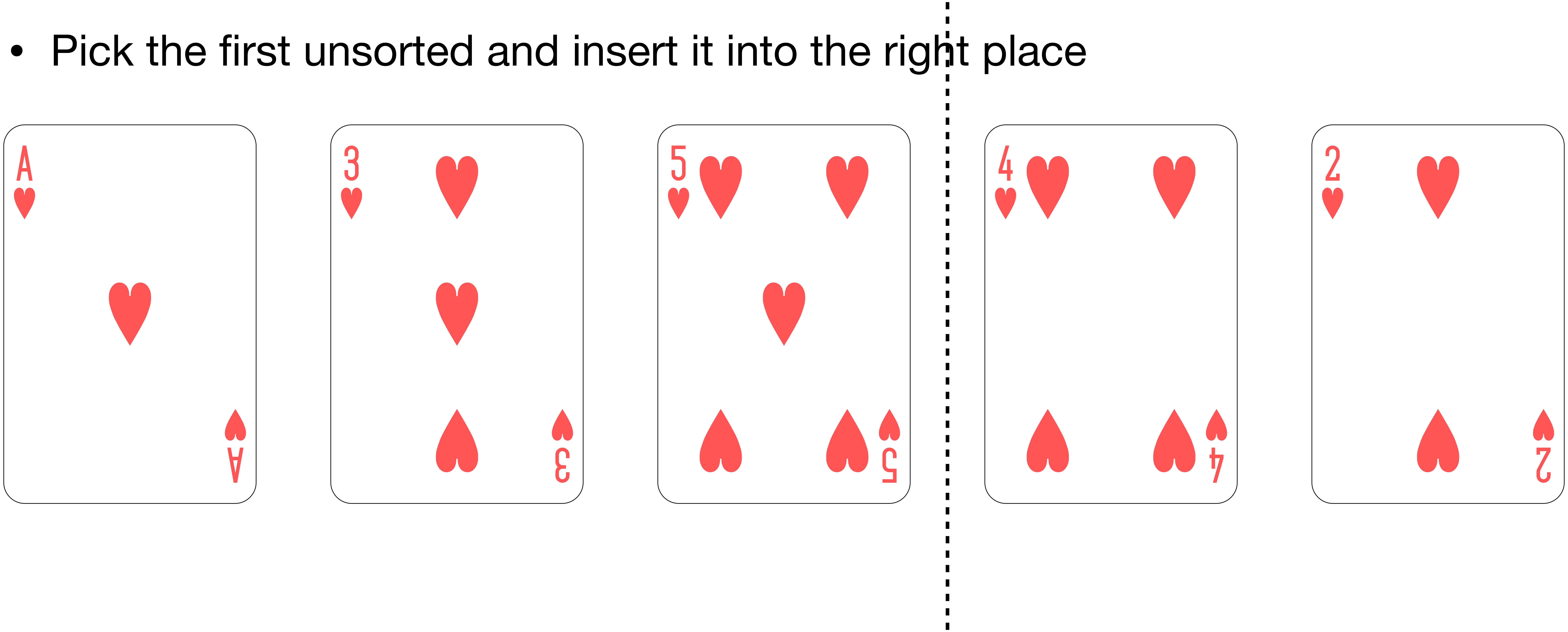
- Pick the first unsorted and inser ght place



Sorting

Example 2

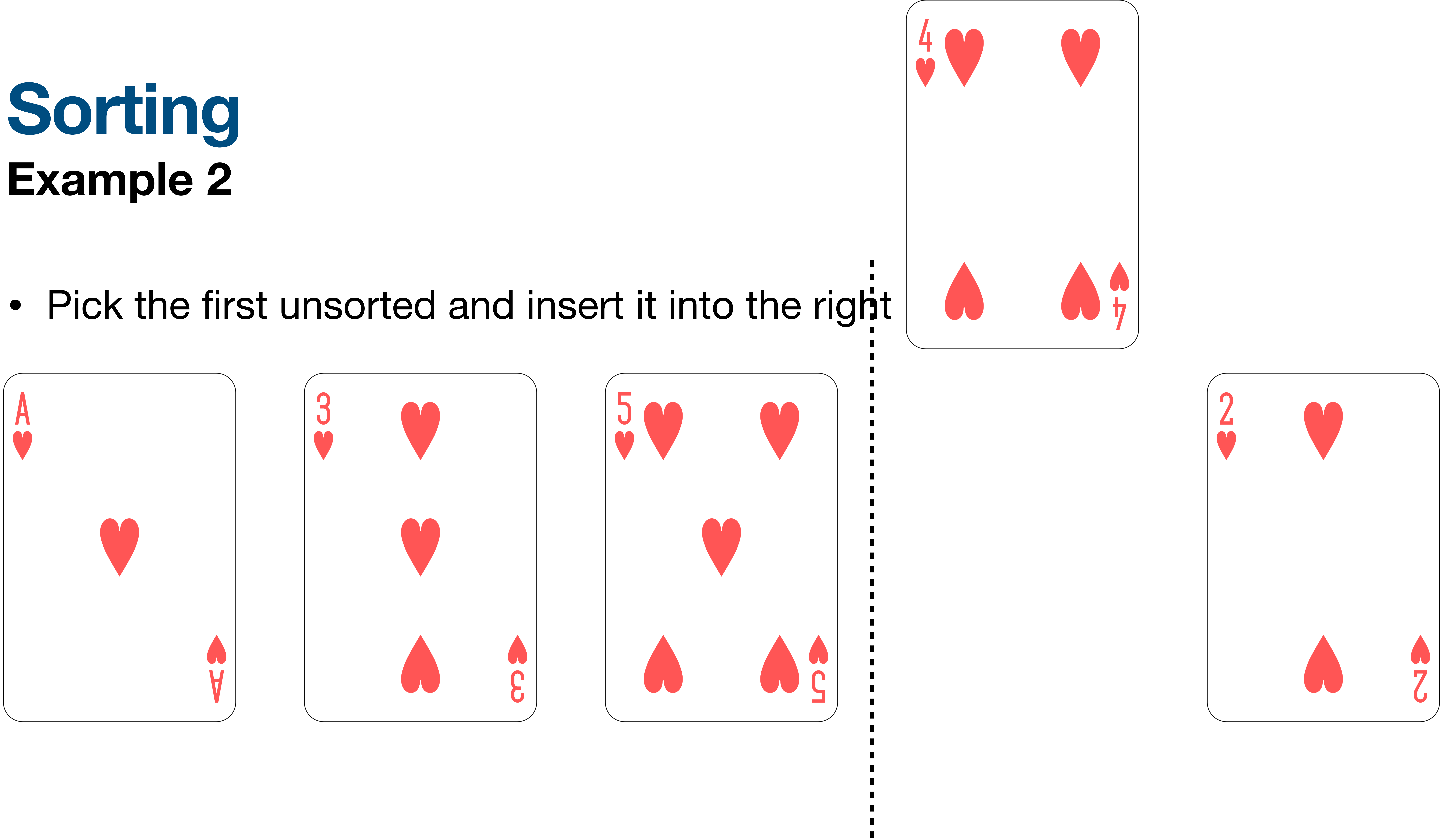
- Pick the first unsorted and insert it into the right place



Sorting

Example 2

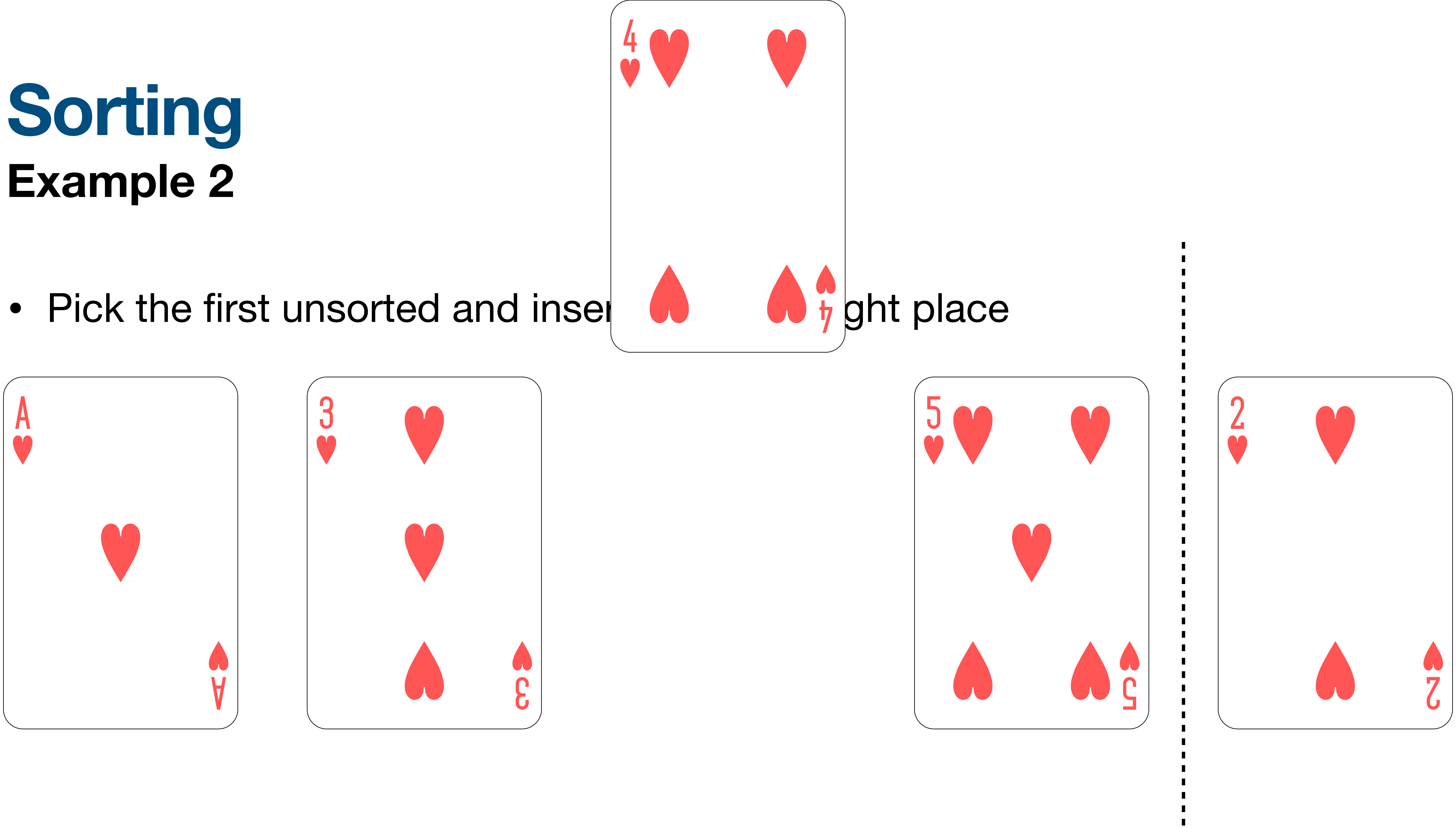
- Pick the first unsorted and insert it into the right



Sorting

Example 2

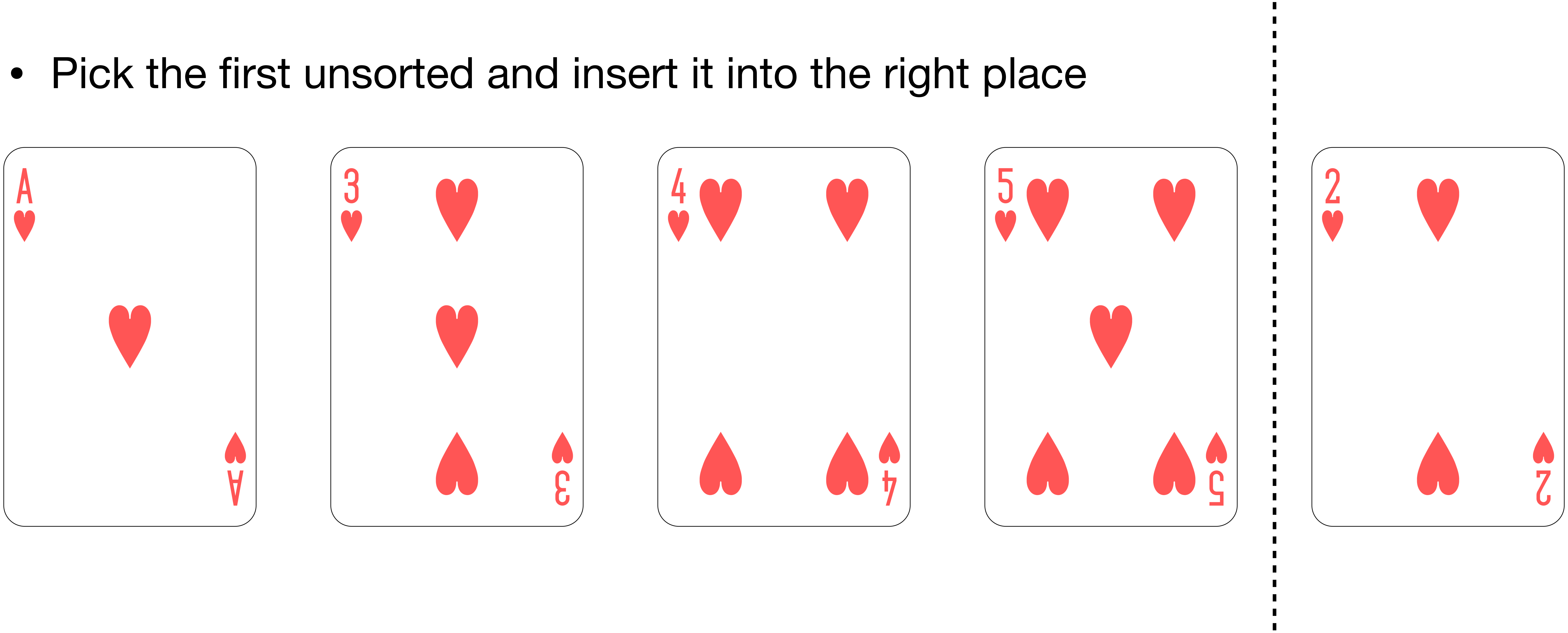
- Pick the first unsorted and inser ght place



Sorting

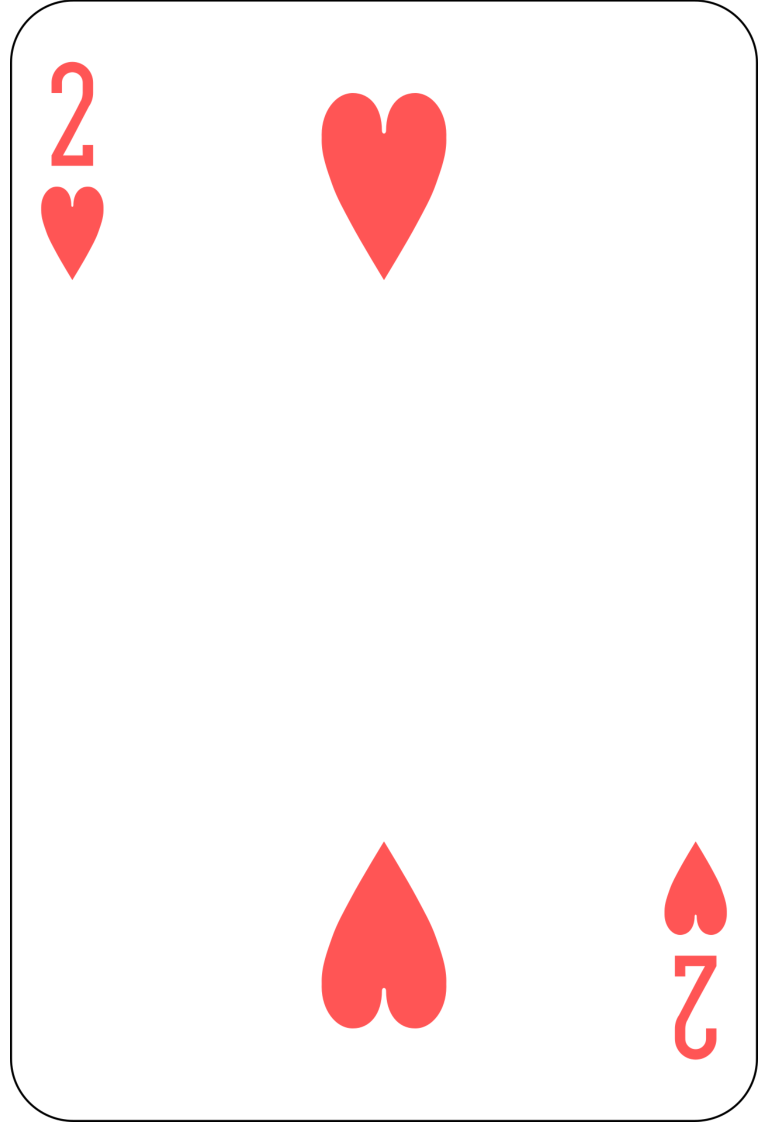
Example 2

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

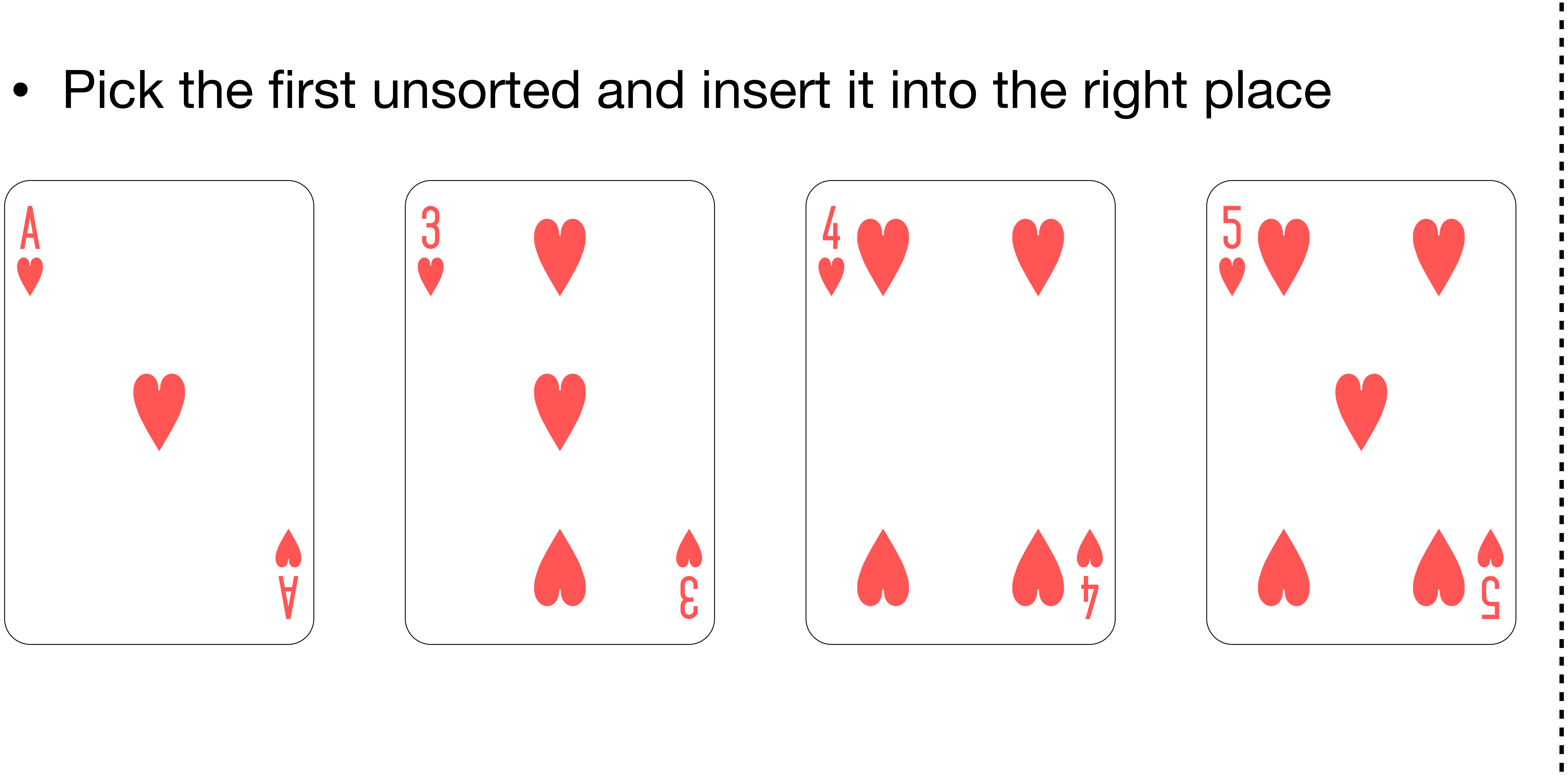


Sorting

Example 2

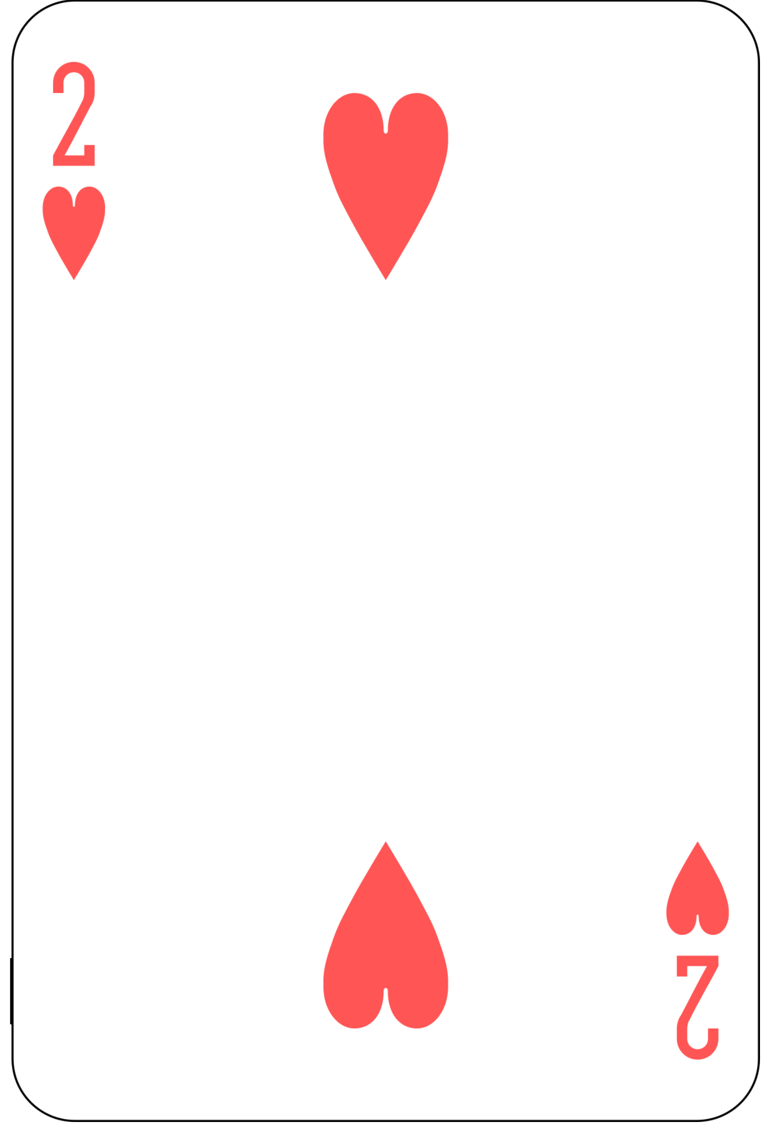


- Pick the first unsorted and insert it into the right place

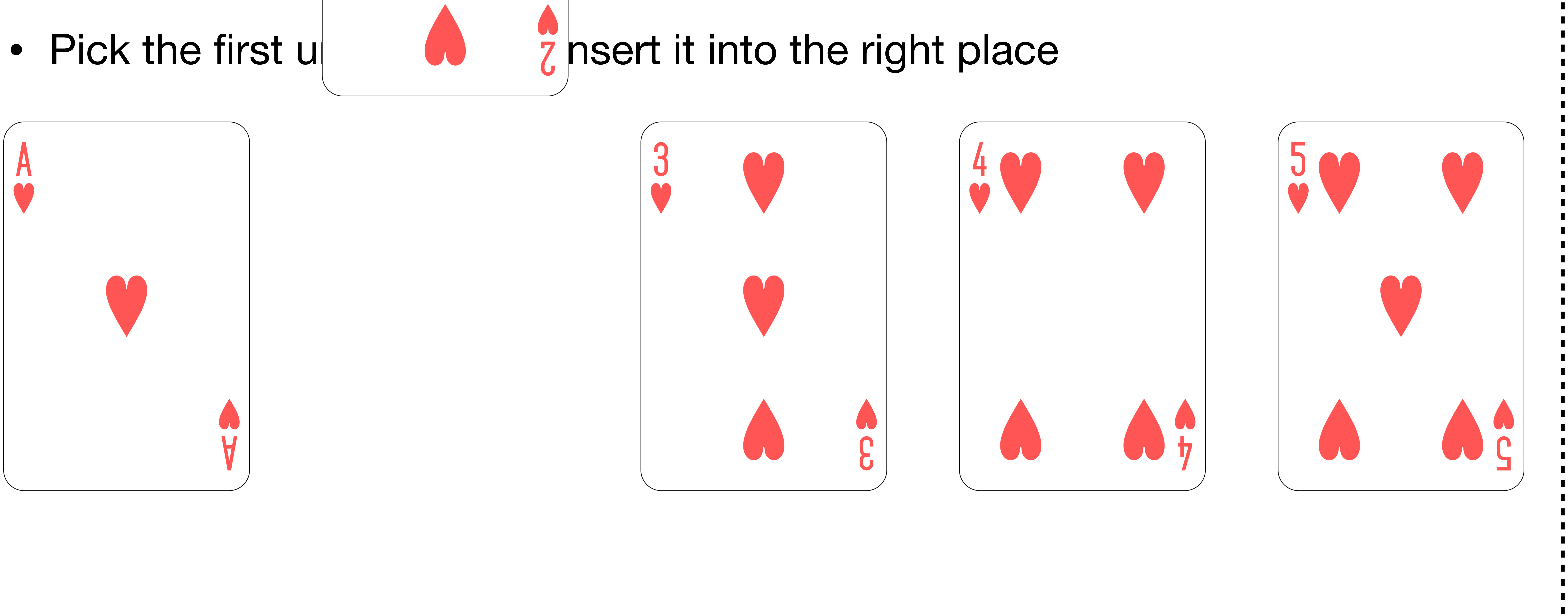


Sorting

Example 2



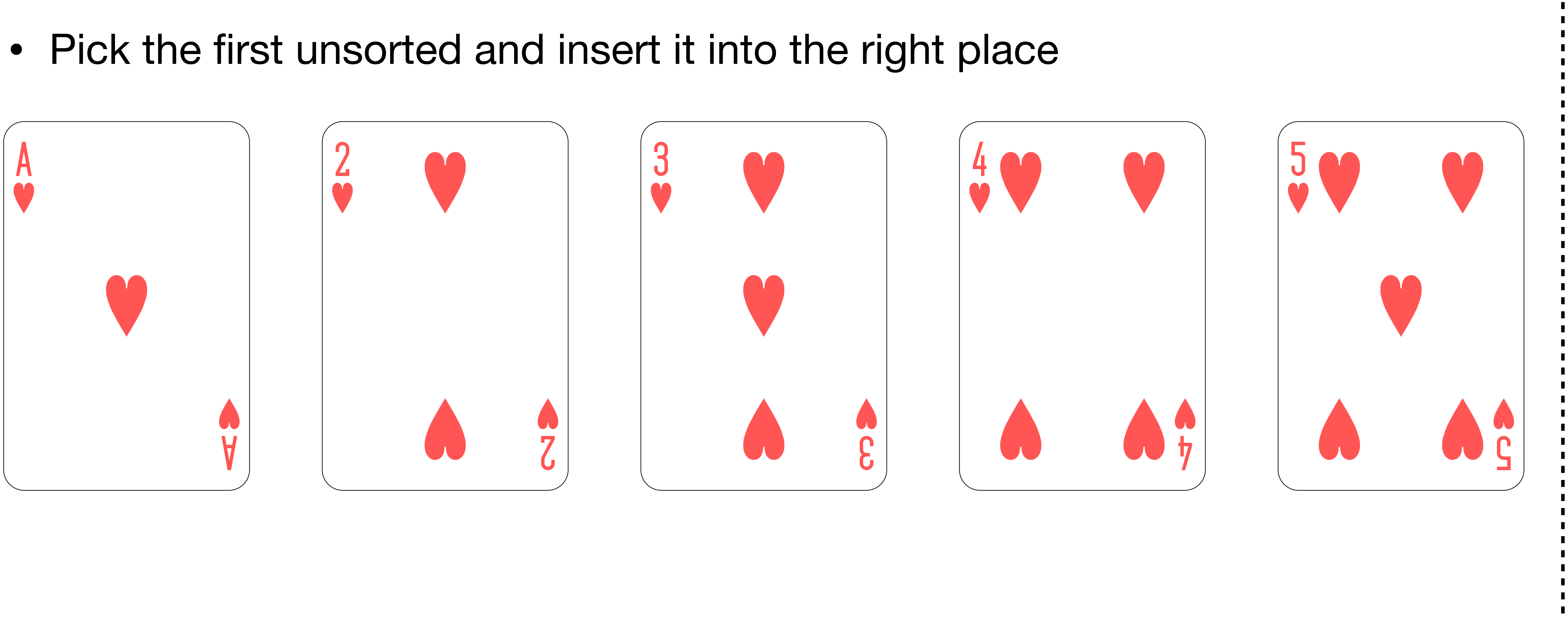
- Pick the first u nsert it into the right place



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]
```

- 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]
```

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

Sorting

Example 2: Algorithm

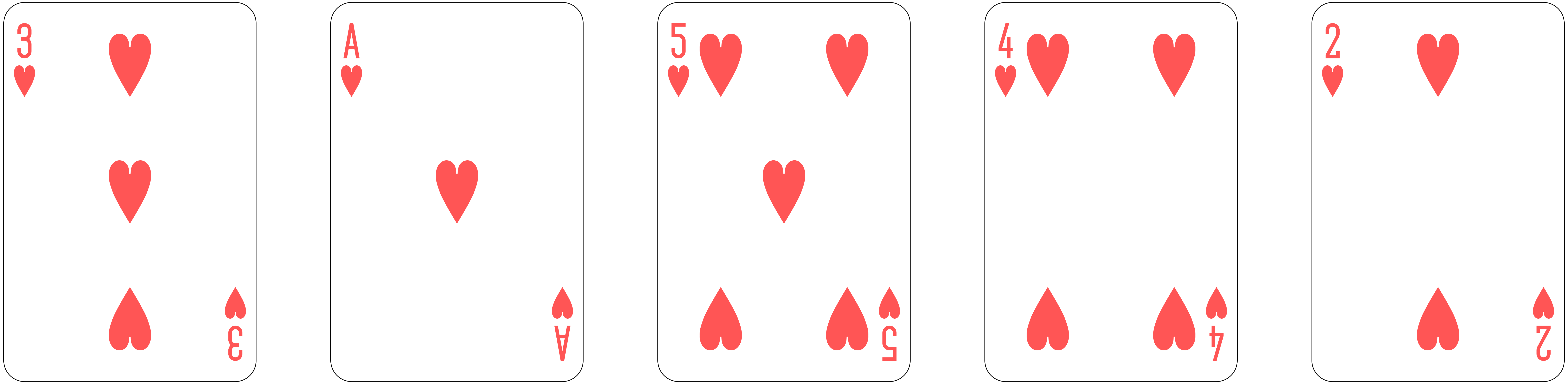
```
For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]
```

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

Sorting

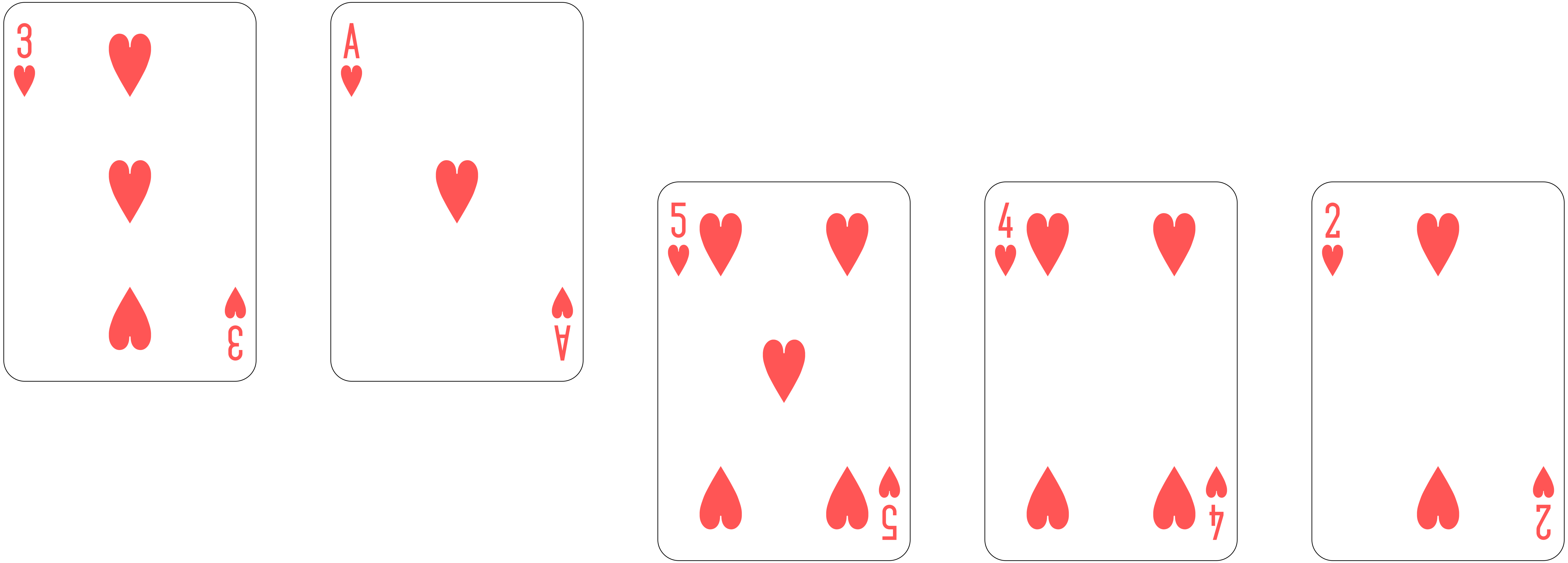
Example 3

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



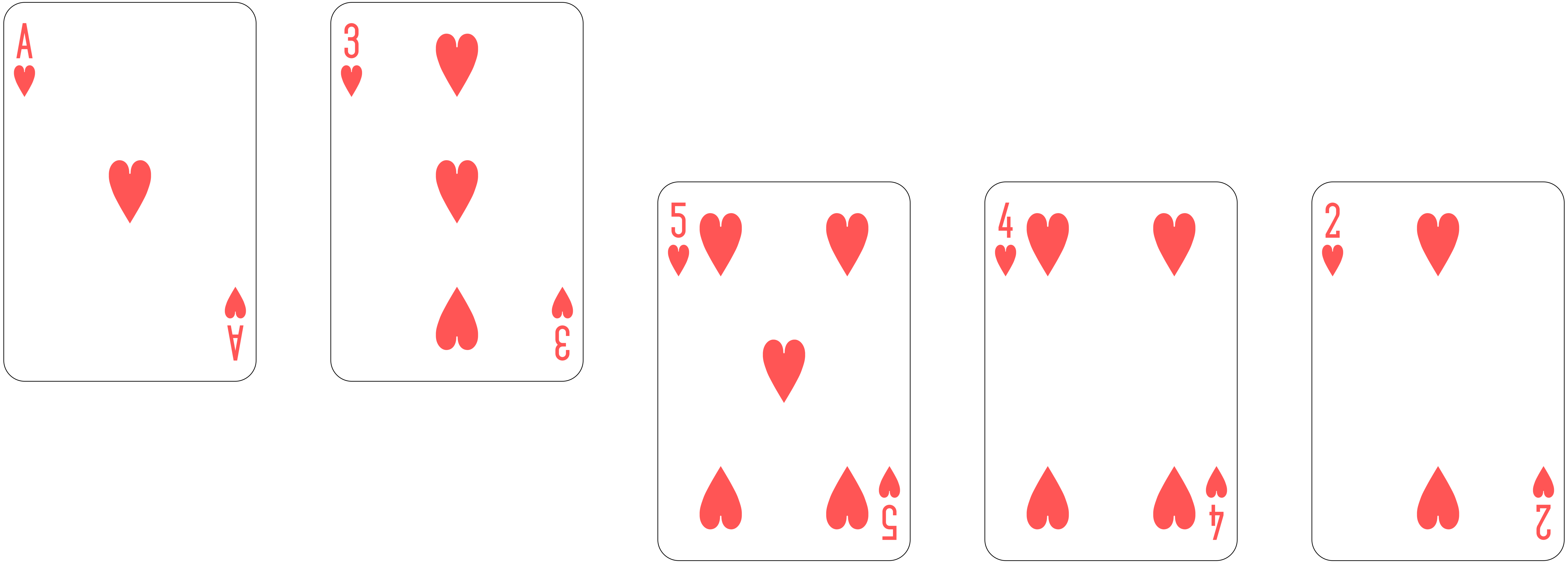
Sorting

Example 3



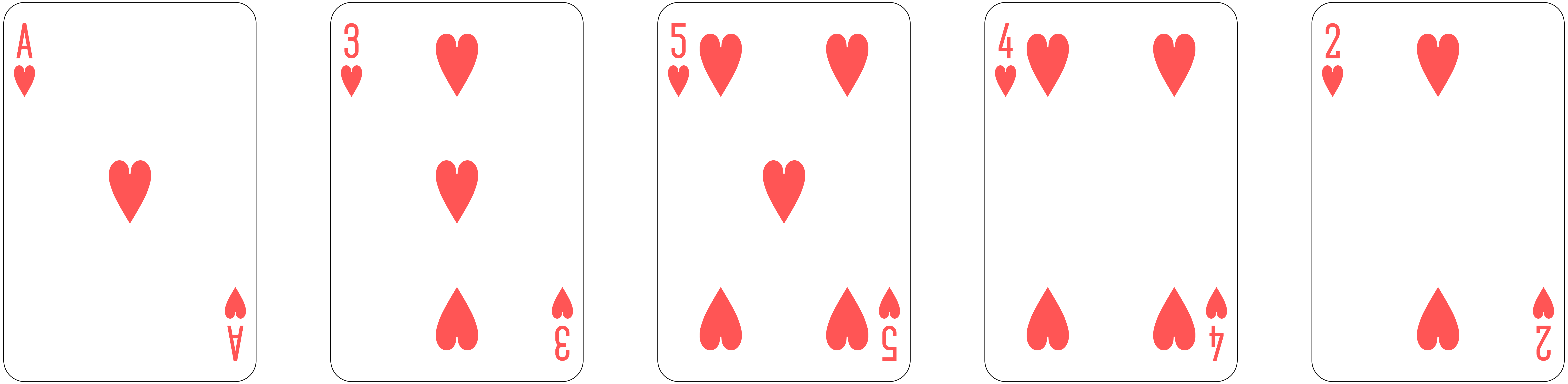
Sorting

Example 3



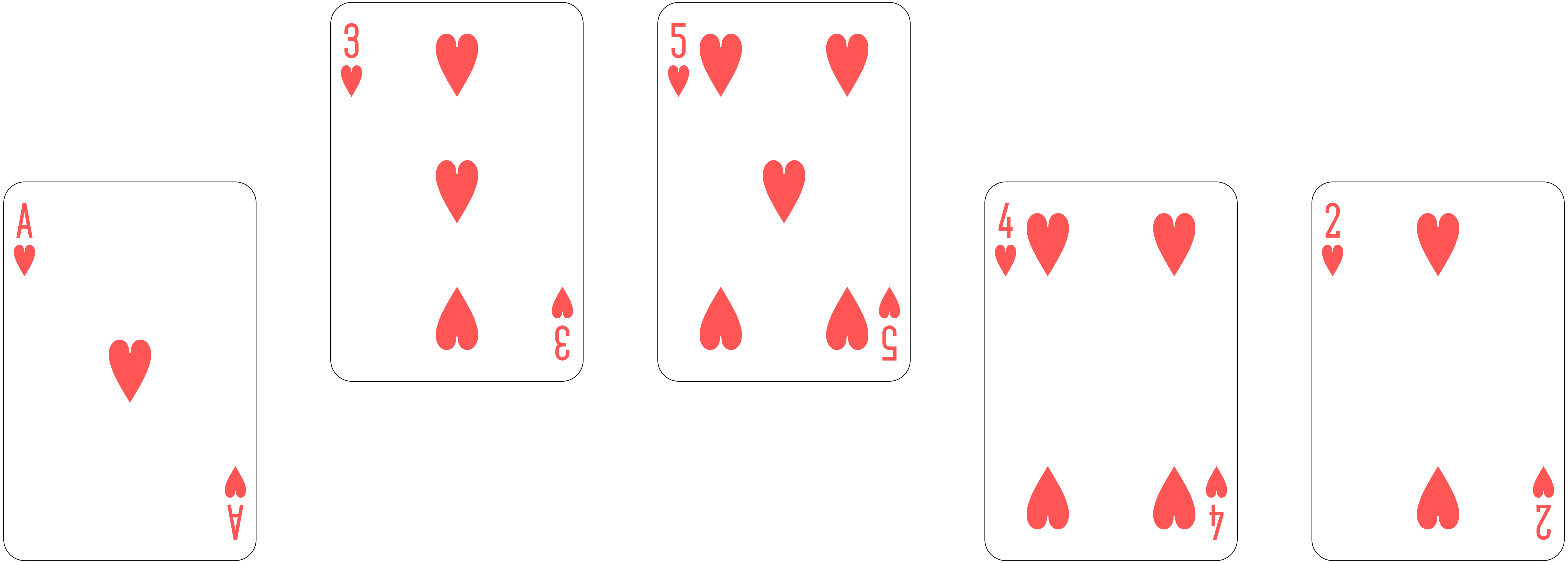
Sorting

Example 3



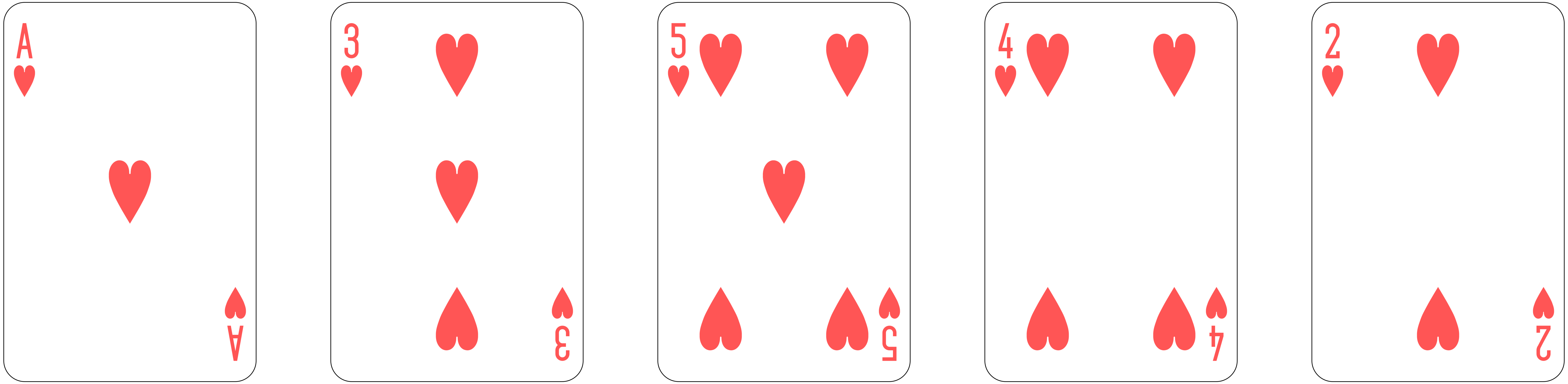
Sorting

Example 3



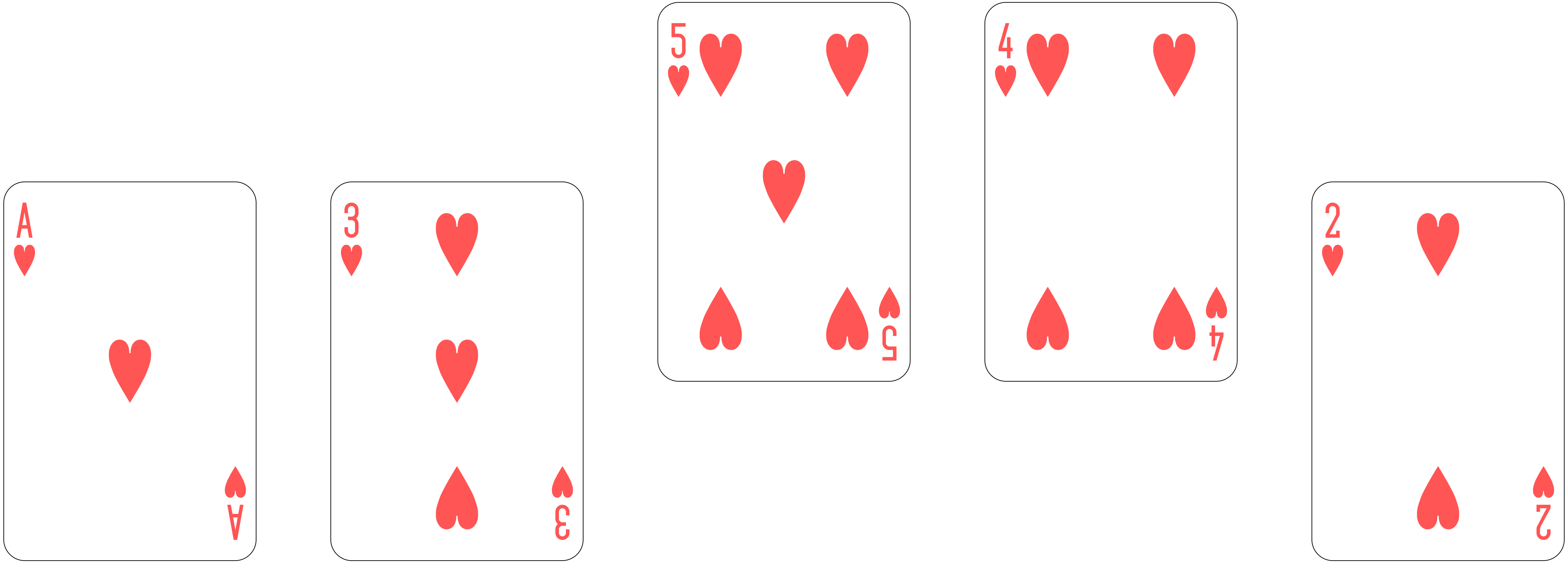
Sorting

Example 3



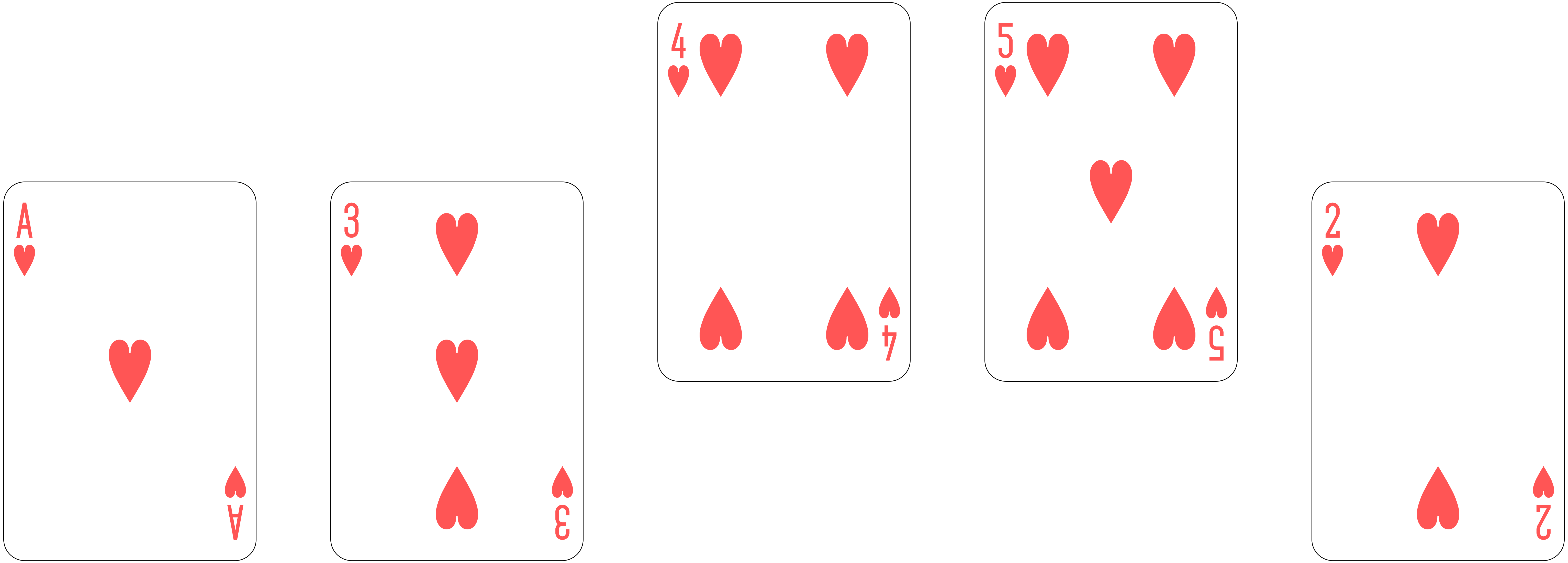
Sorting

Example 3



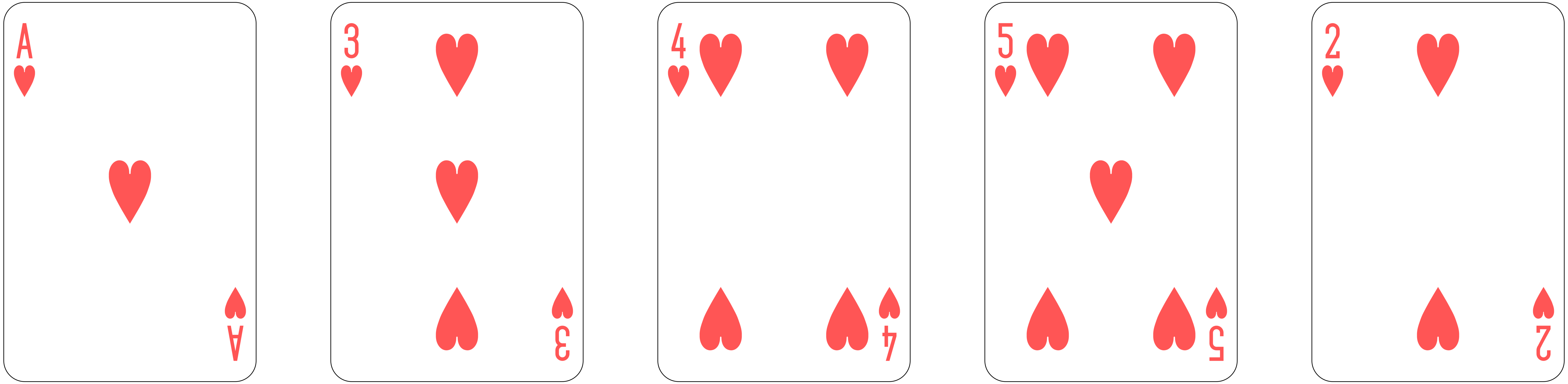
Sorting

Example 3



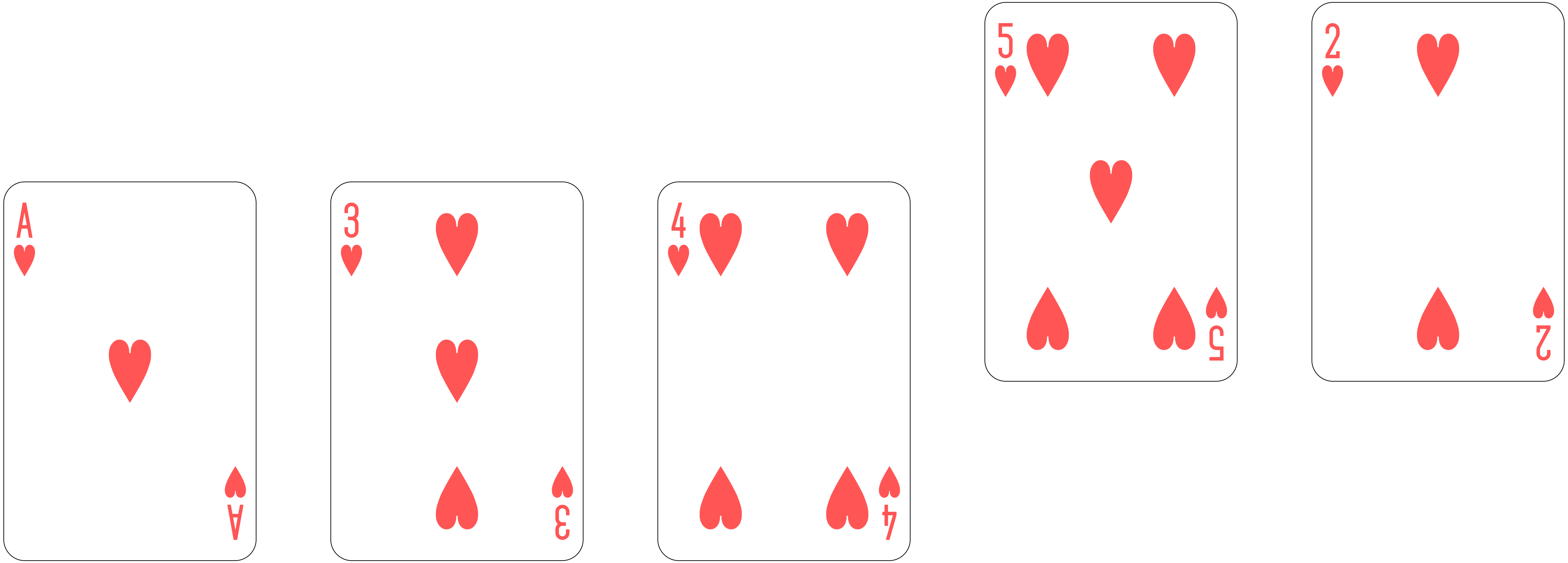
Sorting

Example 3



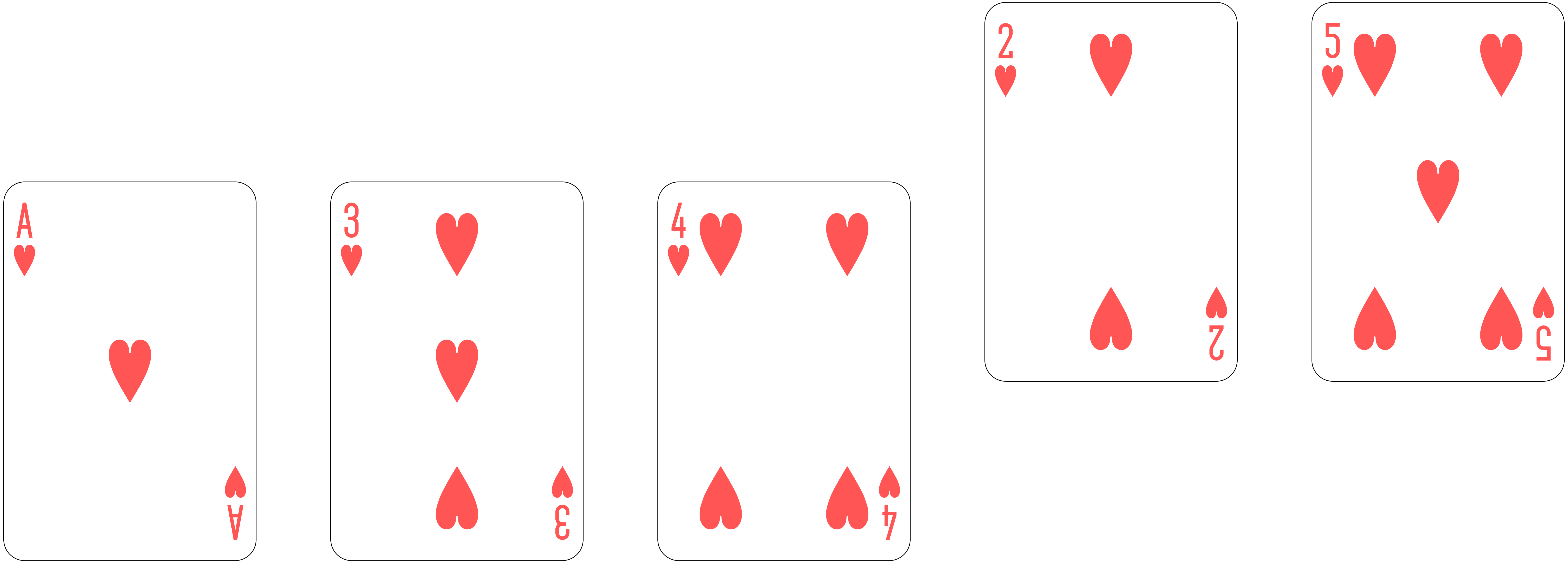
Sorting

Example 3



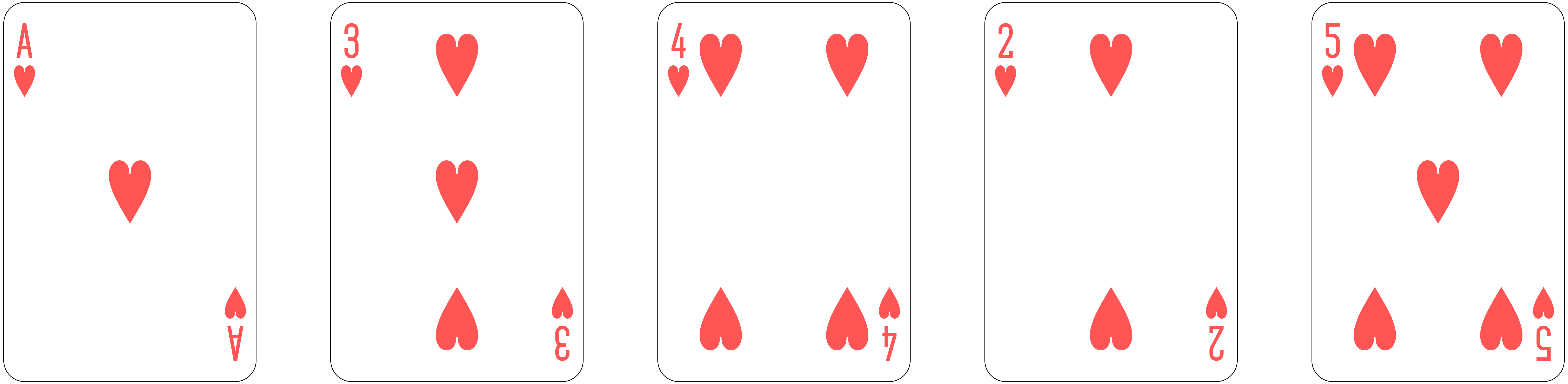
Sorting

Example 3



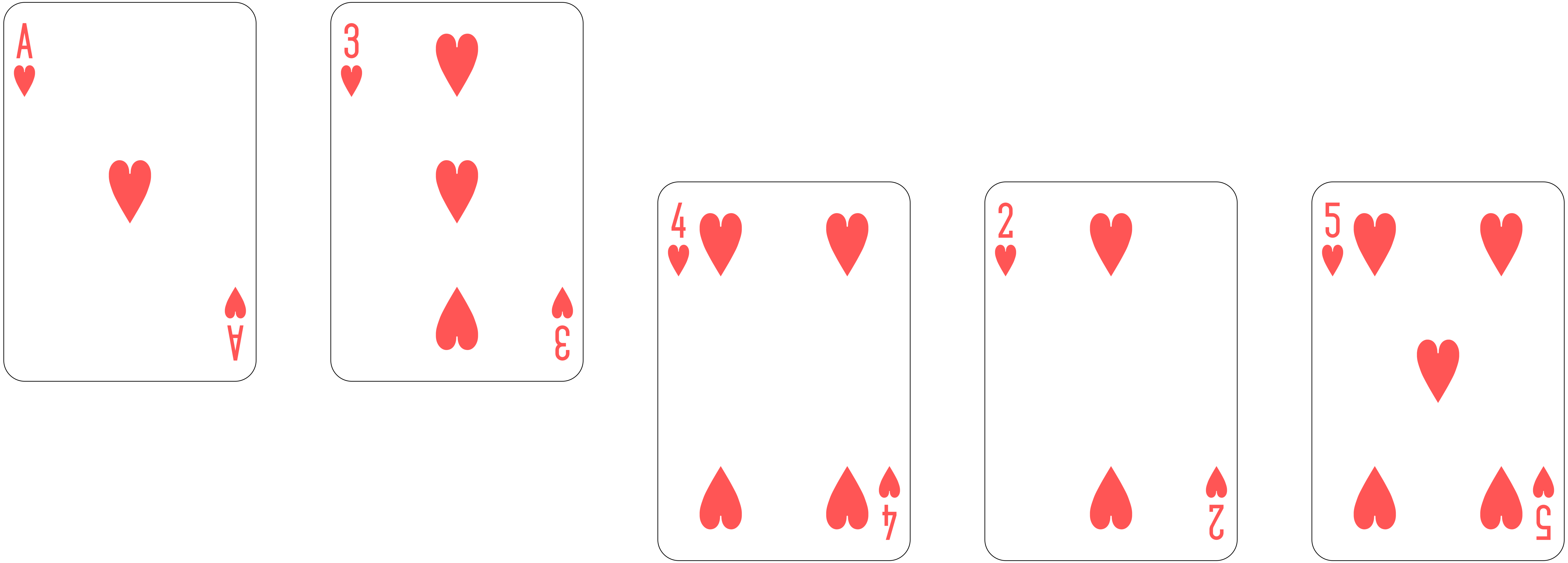
Sorting

Example 3



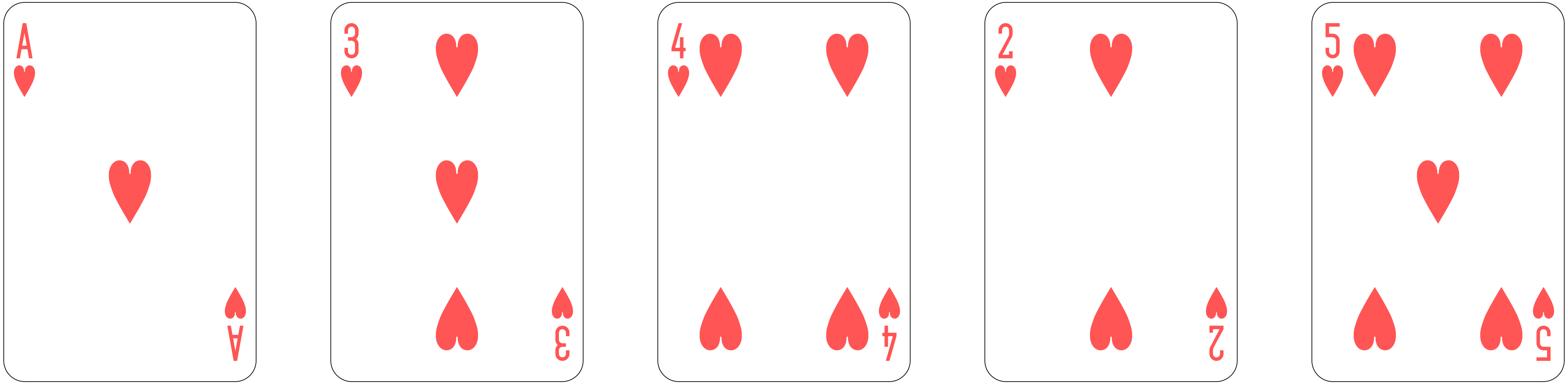
Sorting

Example 3



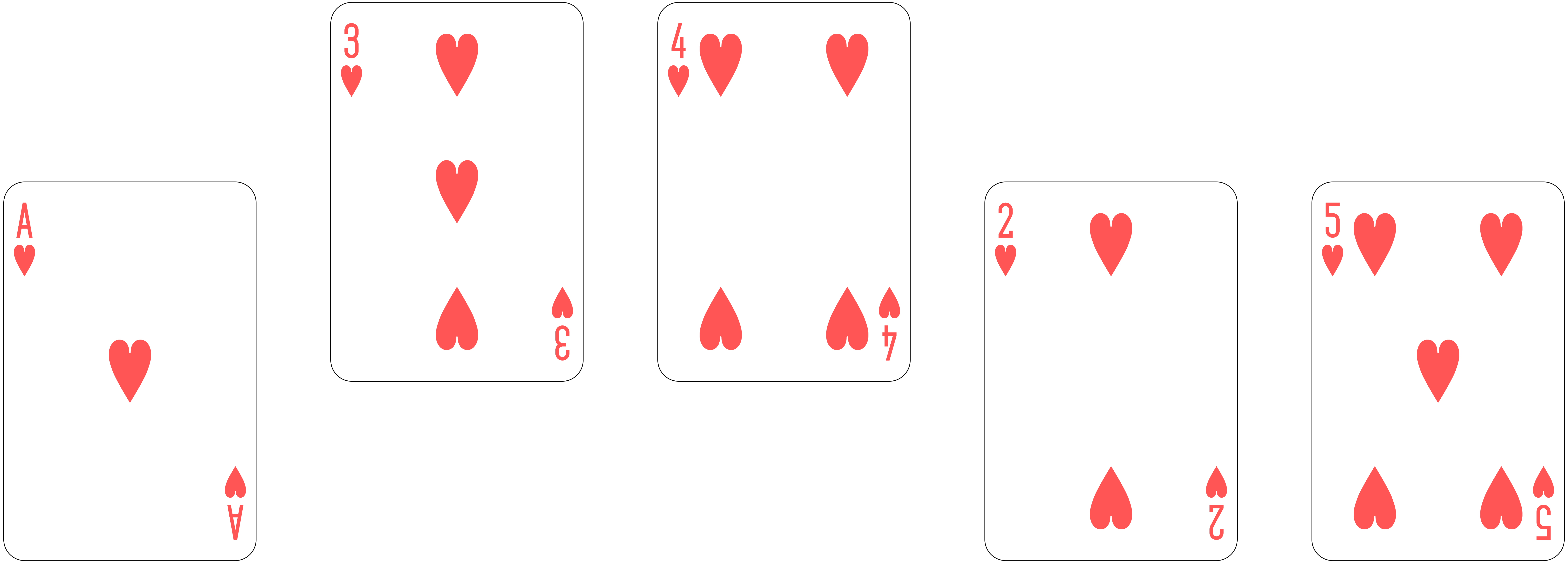
Sorting

Example 3



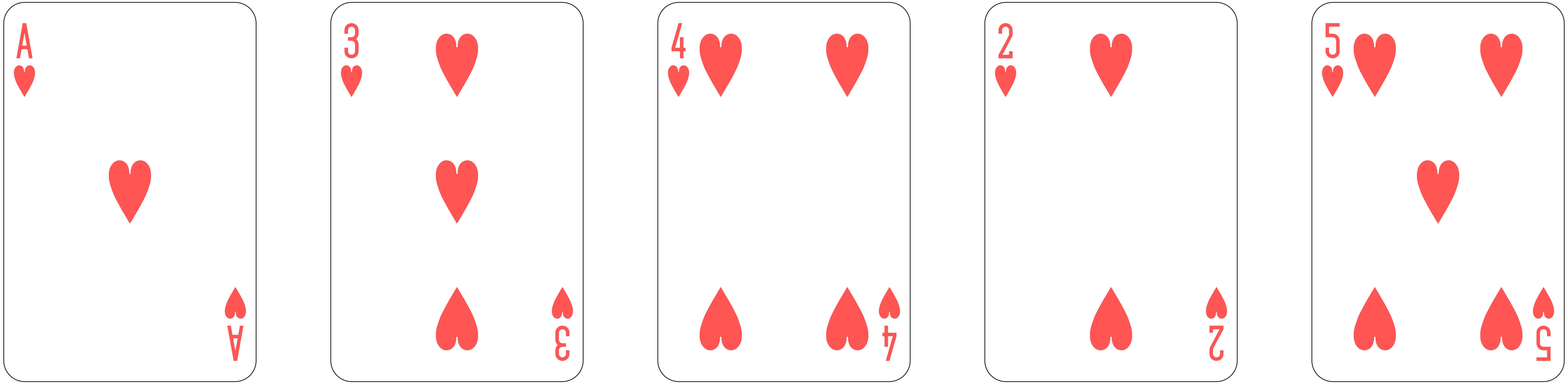
Sorting

Example 3



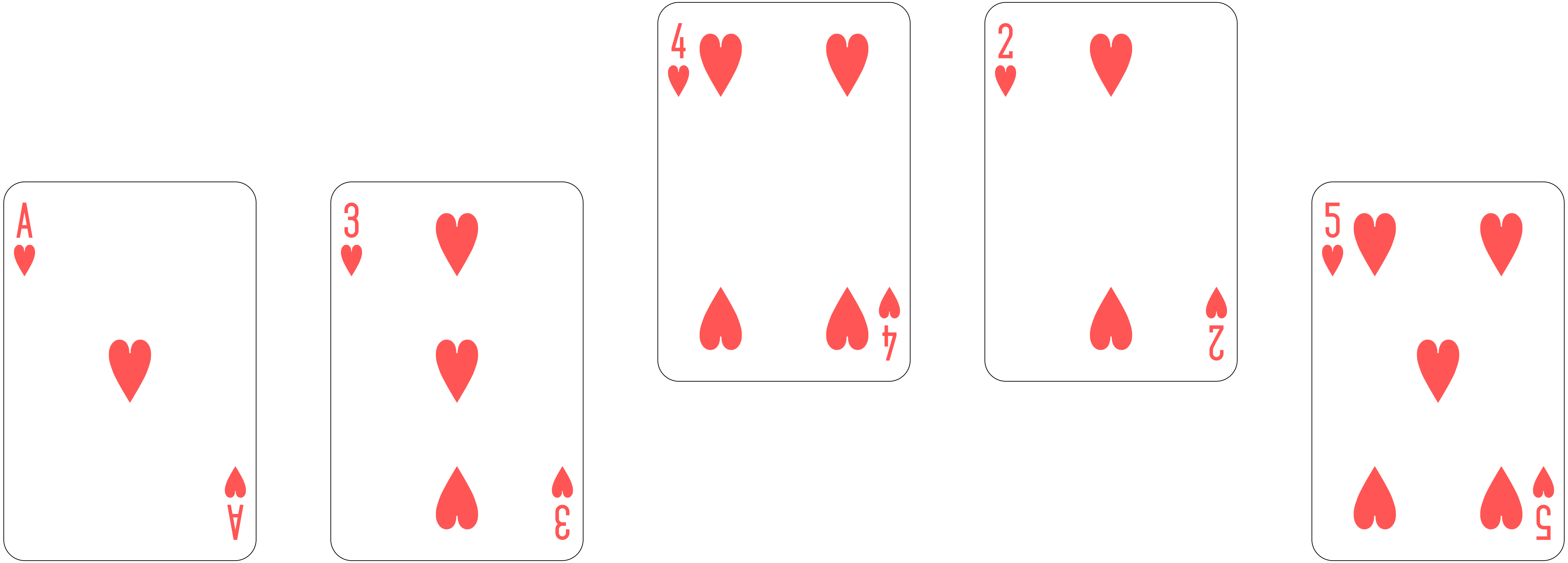
Sorting

Example 3



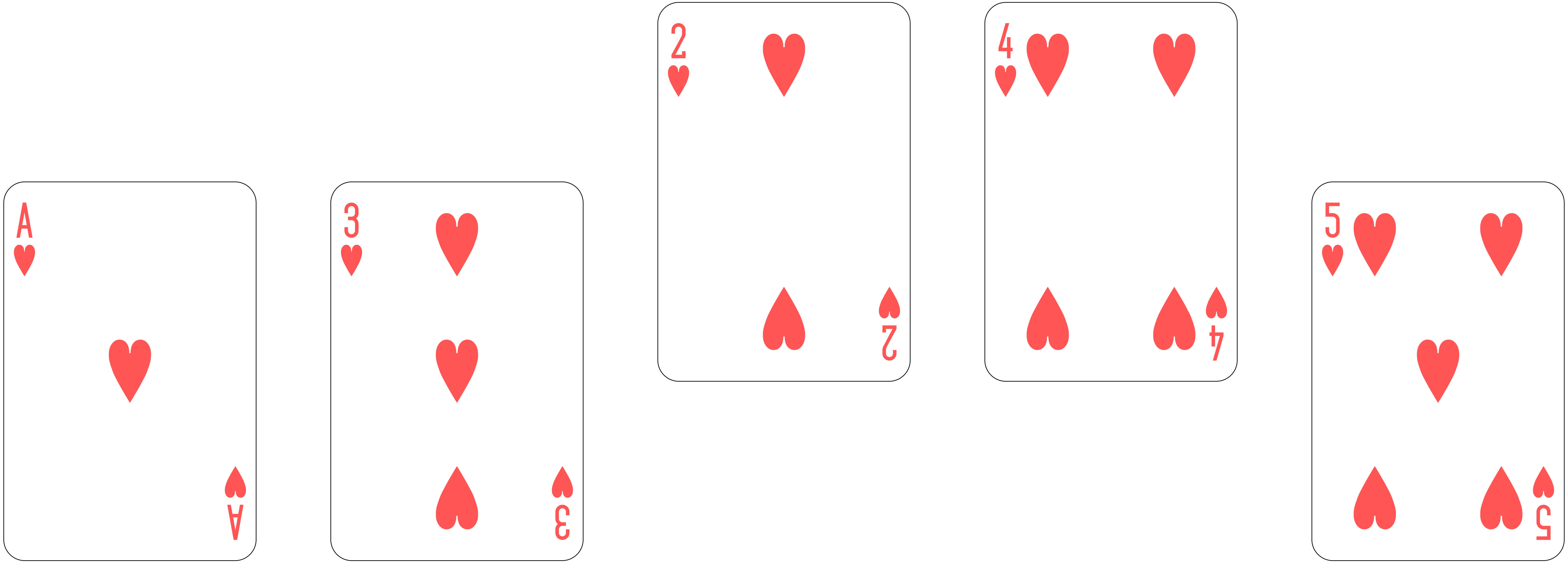
Sorting

Example 3



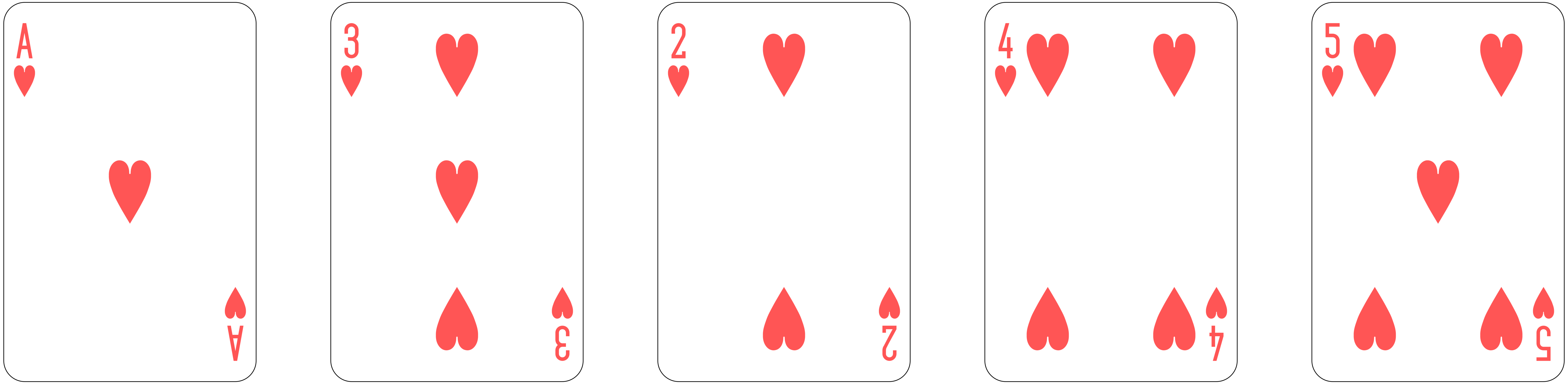
Sorting

Example 3



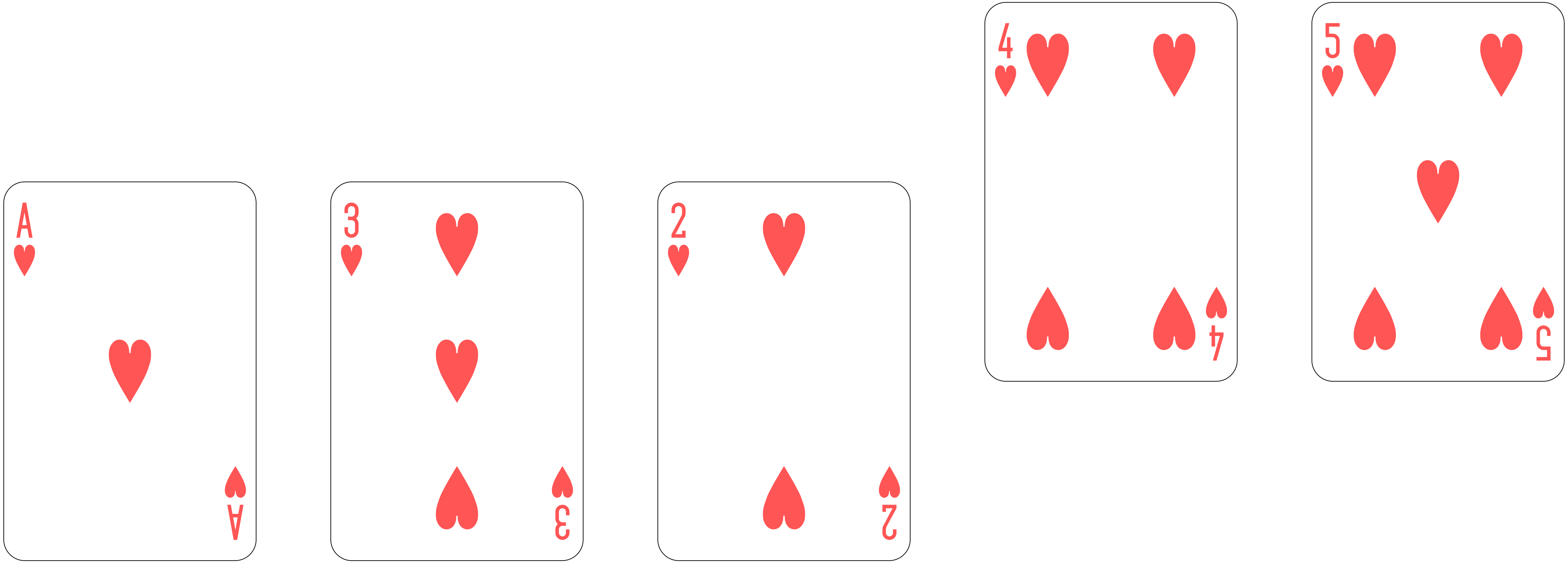
Sorting

Example 3



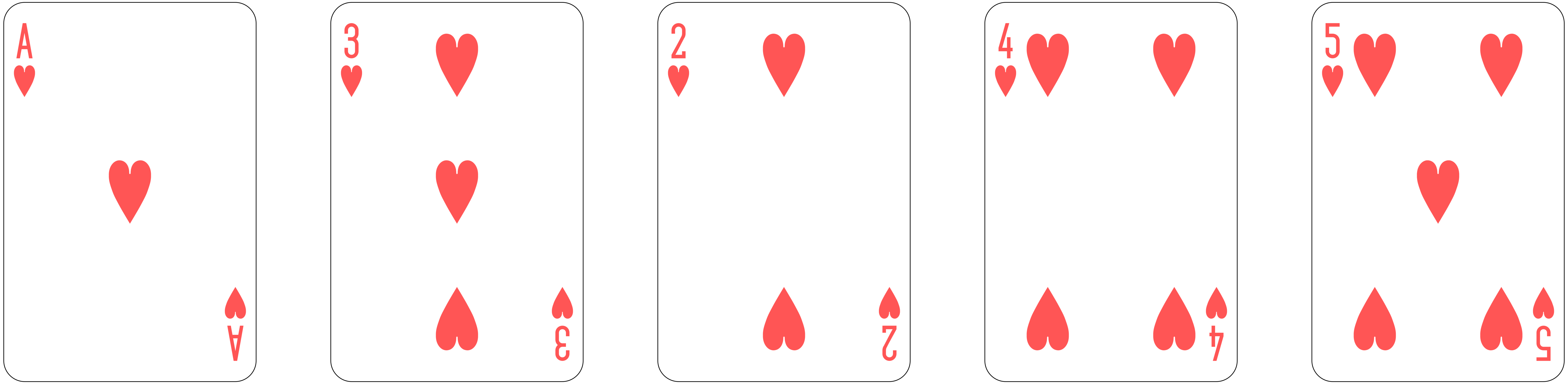
Sorting

Example 3



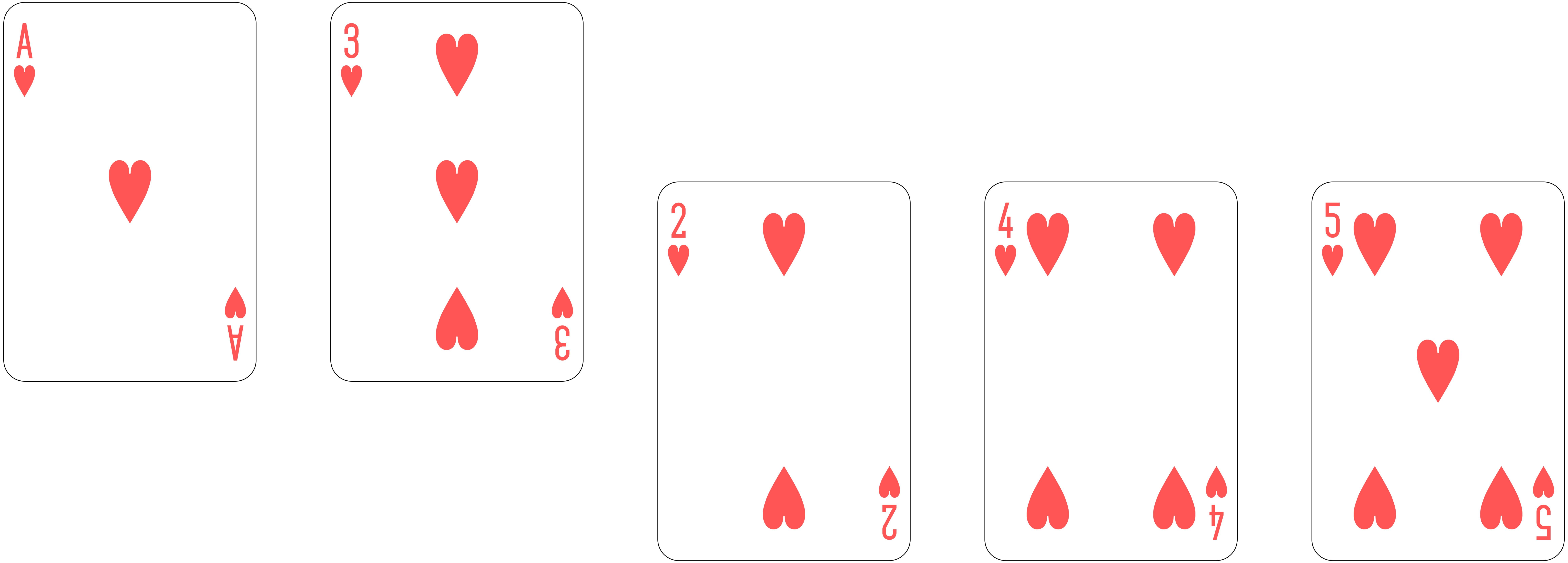
Sorting

Example 3



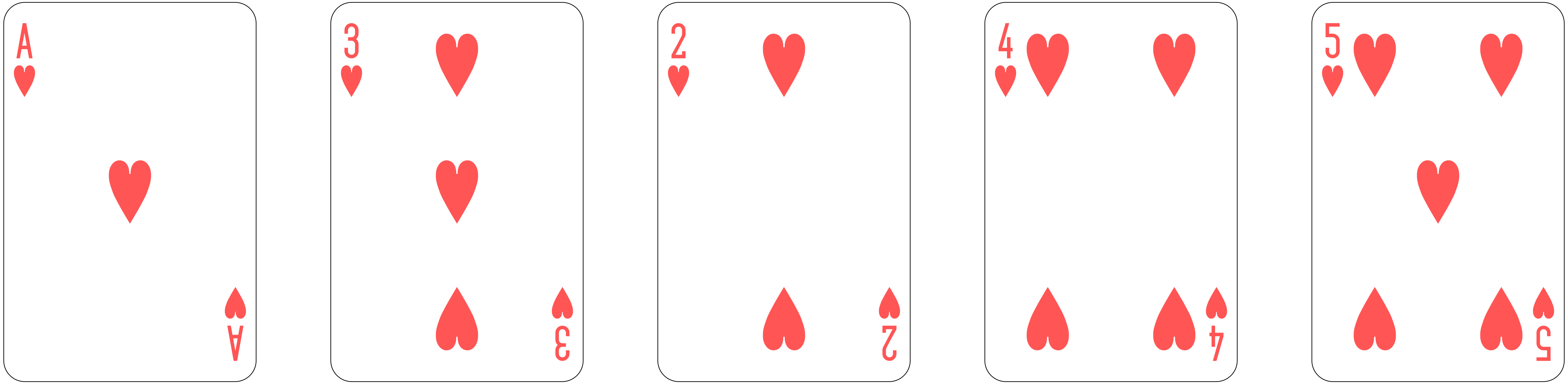
Sorting

Example 3



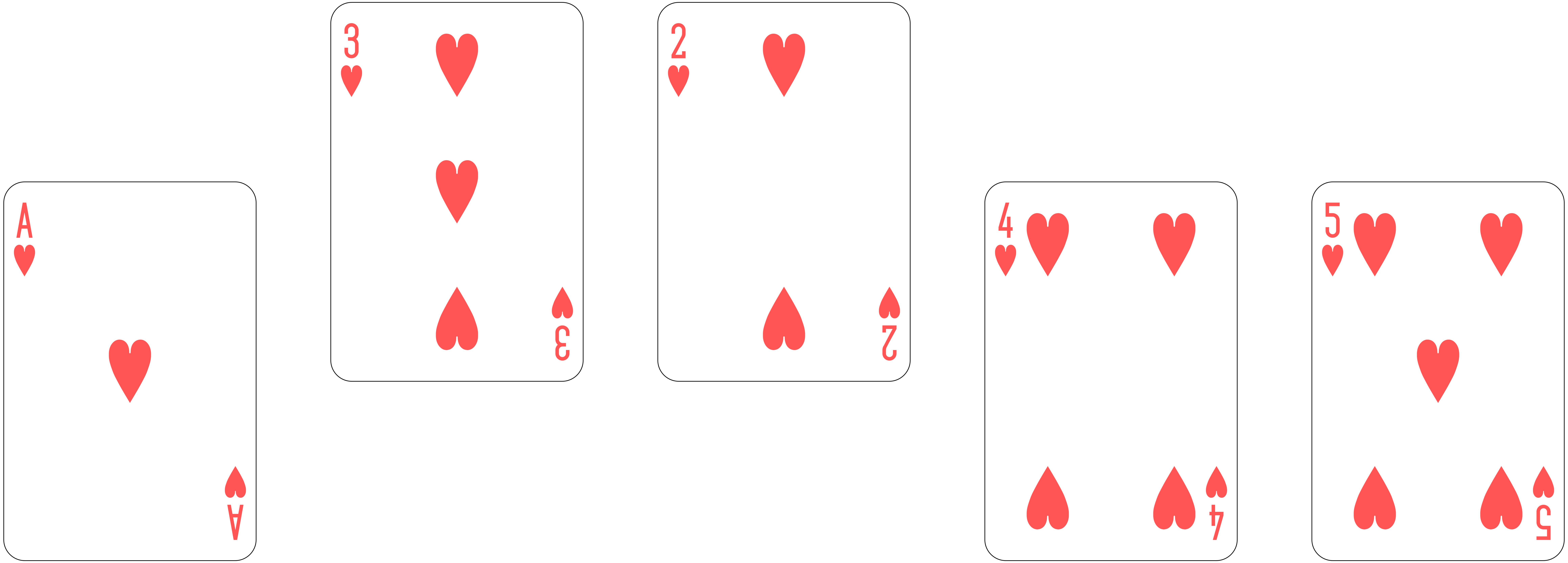
Sorting

Example 3



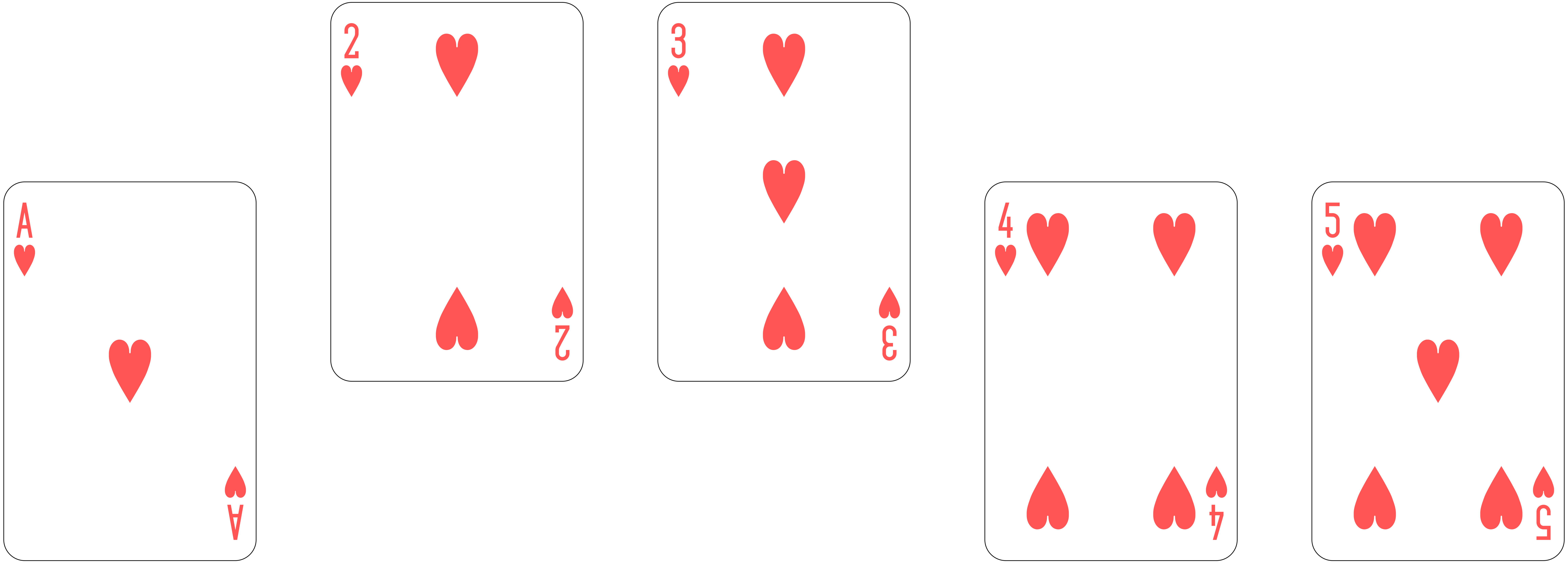
Sorting

Example 3



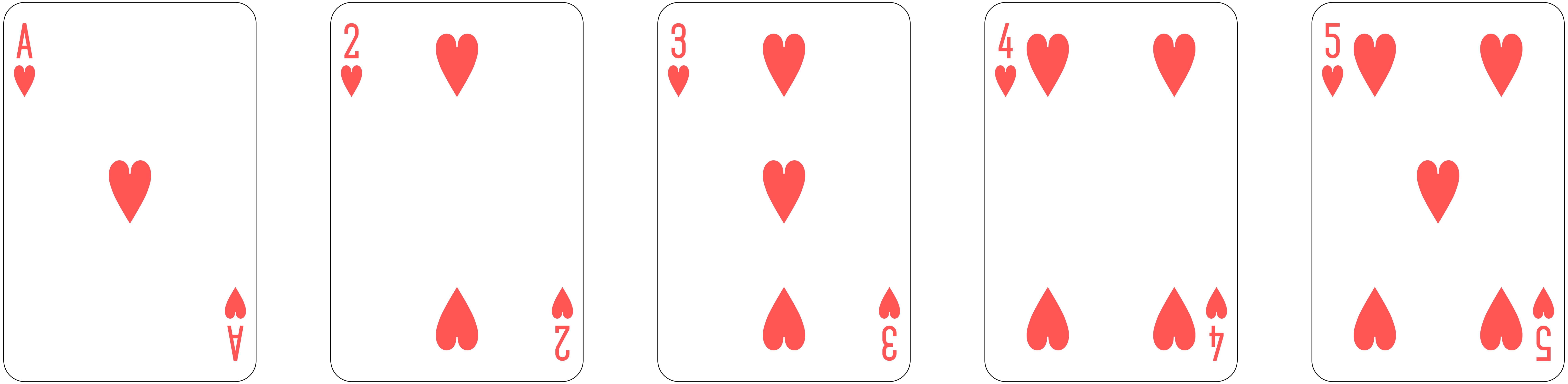
Sorting

Example 3



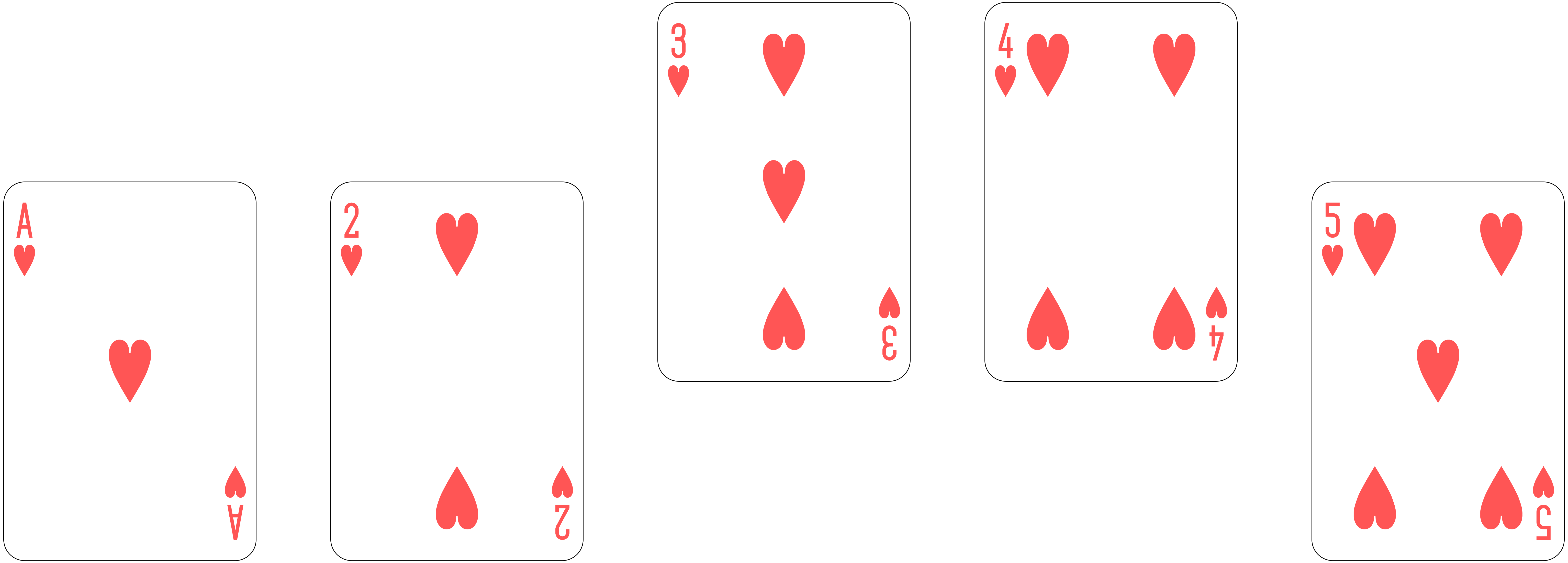
Sorting

Example 3



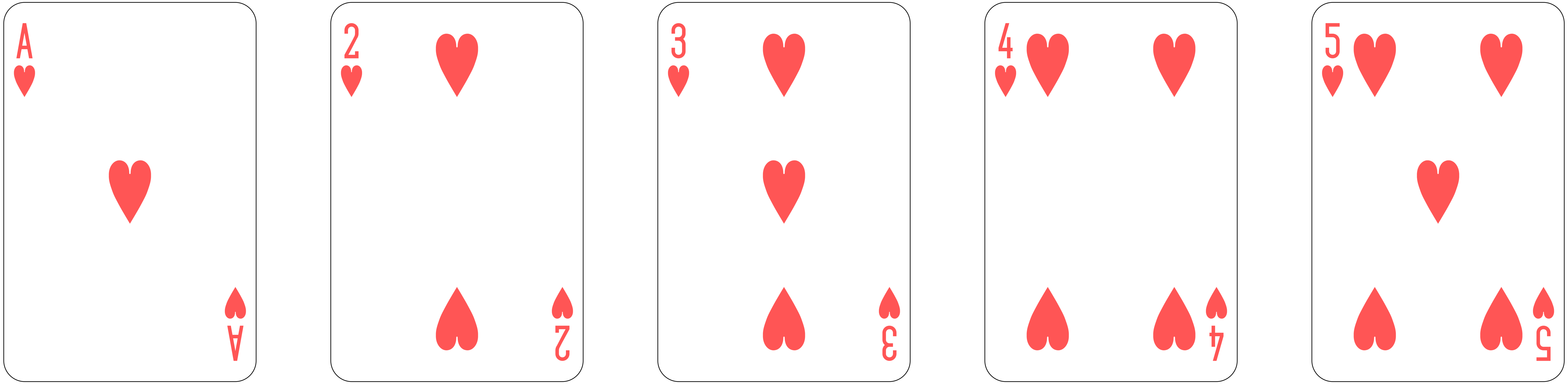
Sorting

Example 3



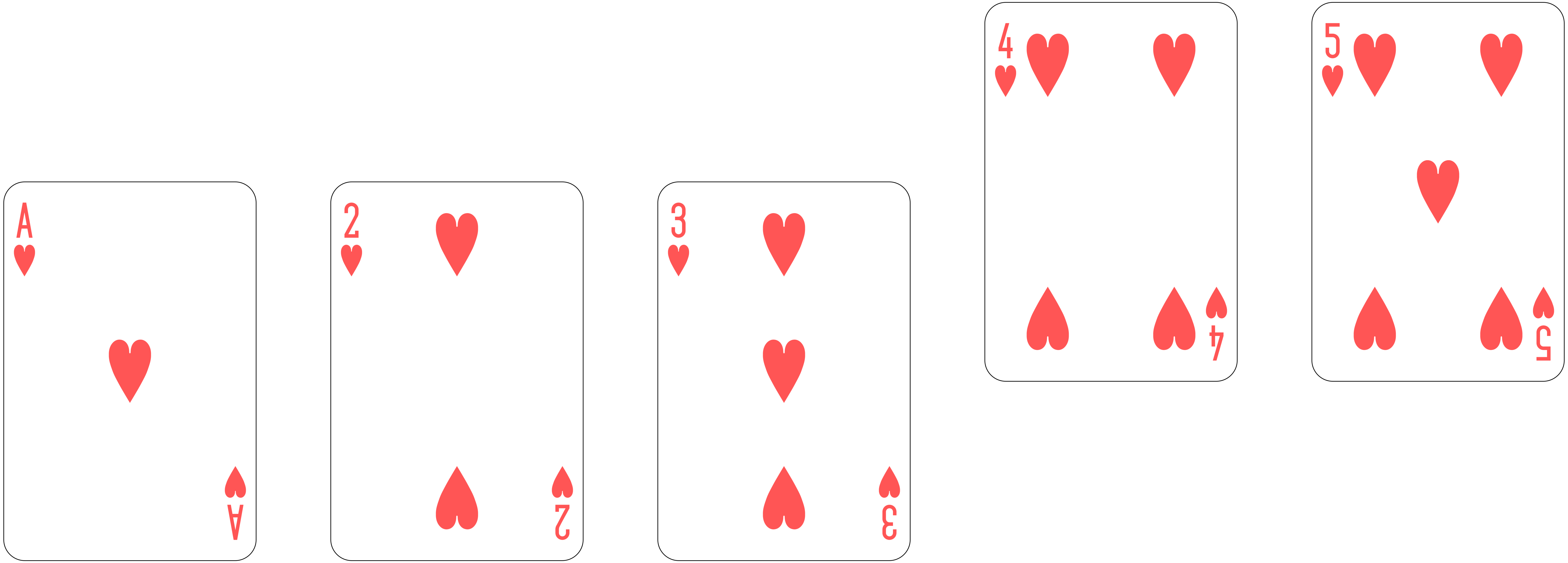
Sorting

Example 3



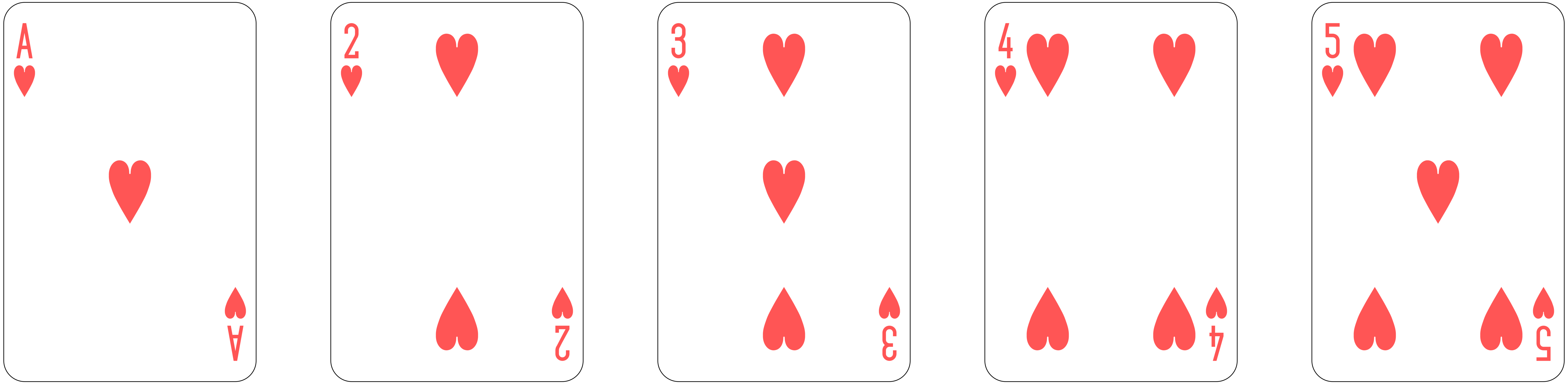
Sorting

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
```

- 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
```

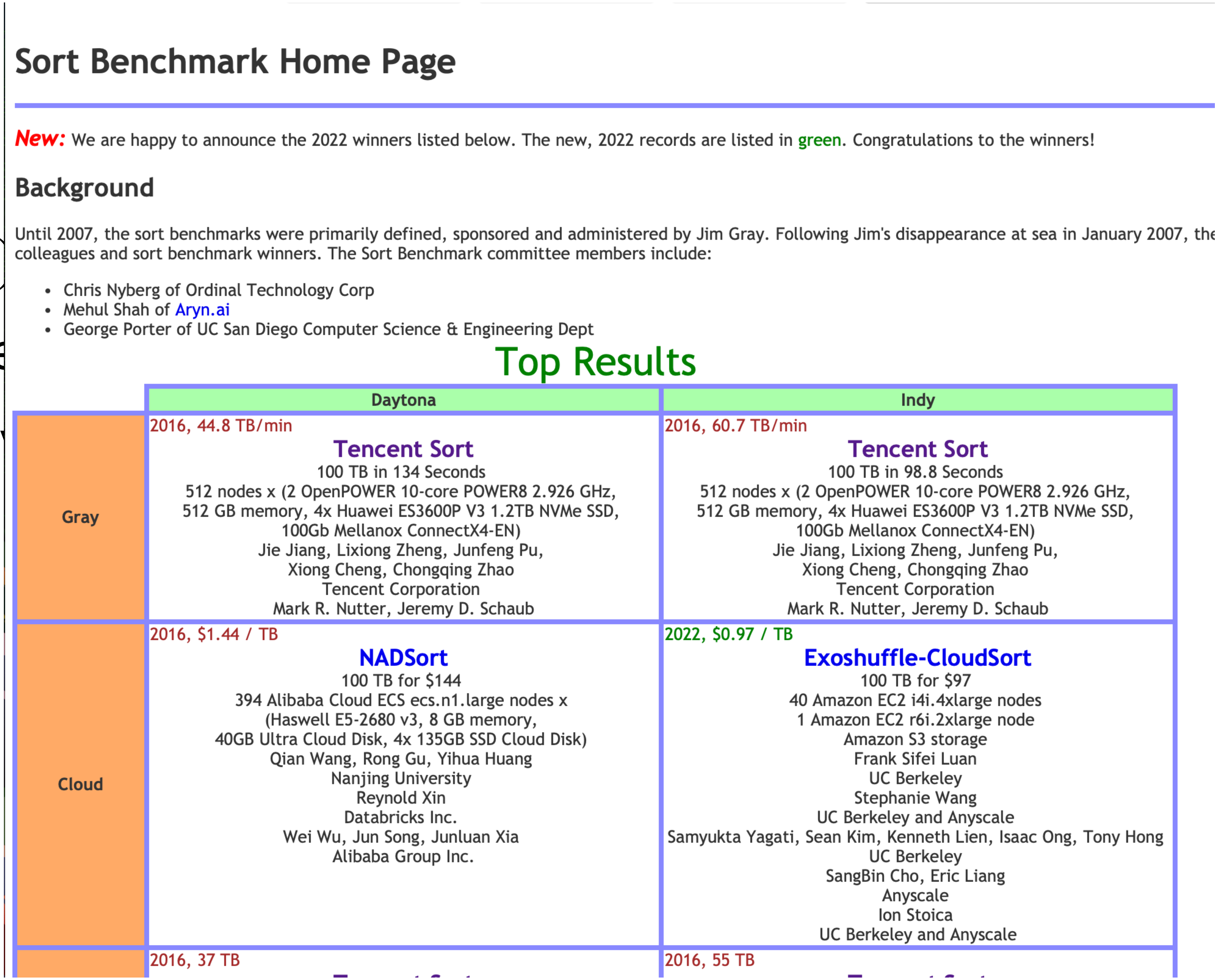
- The largest element "bubbles" up.
- This is called *bubble sort*.

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



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

- 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

```
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

```c
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

```c
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
}
```