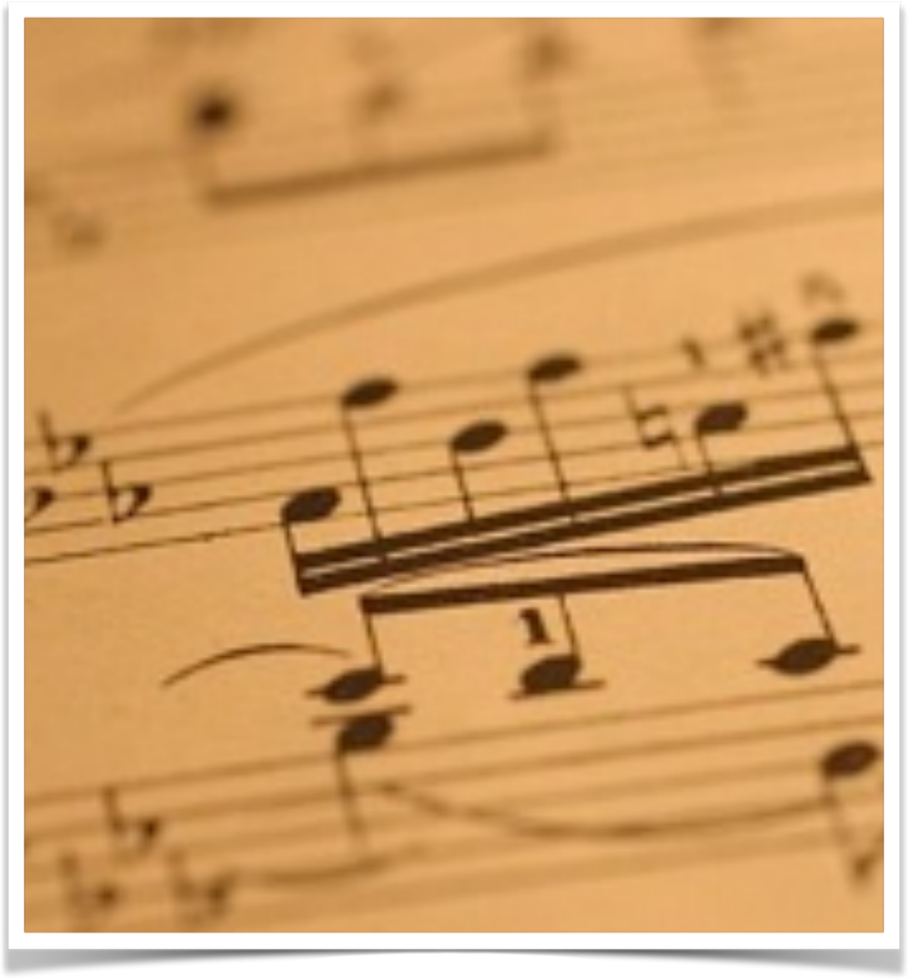
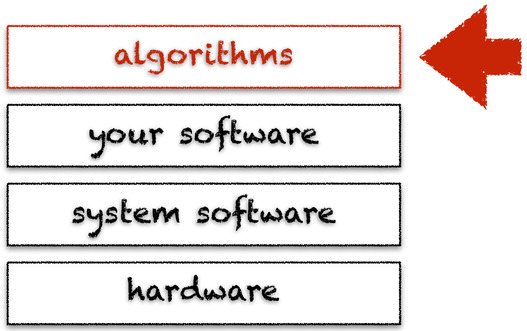


algorithms & computational complexity

learning objectives

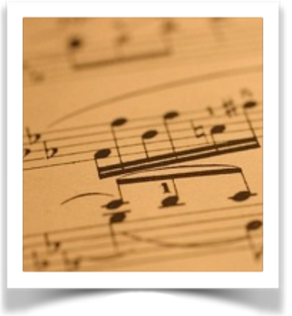


- learn basic principles of algorithmic design
- learn how those principles are used for sorting
- learn how algorithmic complexity is computed

today

what is logic
in computing

how do
they relate?



what is an
algorithm

how to measure
algorithmic complexity?

logic

the intellectual tool for reasoning about the truth and falsity of statements

boolean algebra

assume that p , q and r are boolean variables (or statements) and that $T = true$, $F = false$, we have:

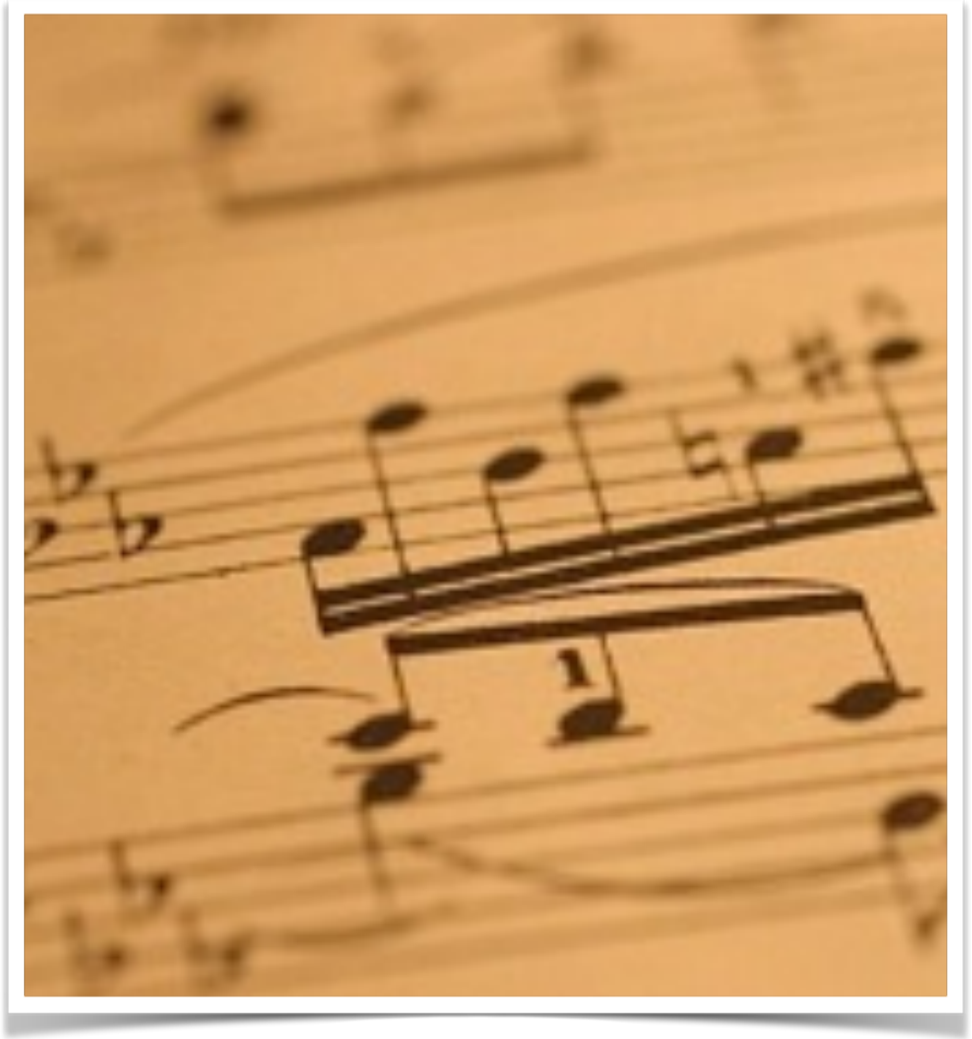
p	$\neg p$
F	T
T	F

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

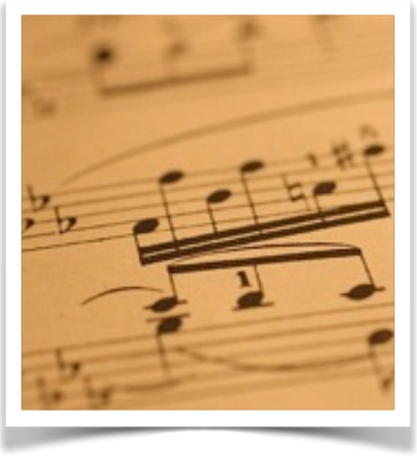
Associative Rules:	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
Distributive Rules:	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
Idempotent Rules:	$p \wedge p \Leftrightarrow p$	$p \vee p \Leftrightarrow p$
Double Negation:	$\neg\neg p \Leftrightarrow p$	
DeMorgan's Rules:	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
Commutative Rules:	$p \wedge q \Leftrightarrow q \wedge p$	$p \vee q \Leftrightarrow q \vee p$
Absorption Rules:	$p \vee (p \wedge q) \Leftrightarrow p$	$p \wedge (p \vee q) \Leftrightarrow p$
Bound Rules:	$p \wedge F \Leftrightarrow F \quad p \wedge T \Leftrightarrow p$	$p \vee T \Leftrightarrow T \quad p \vee F \Leftrightarrow p$
Negation Rules:	$p \wedge (\neg p) \Leftrightarrow F$	$p \vee (\neg p) \Leftrightarrow T$

$\neg \Leftrightarrow$ **not**
$\vee \Leftrightarrow$ **or**
$\wedge \Leftrightarrow$ **and**



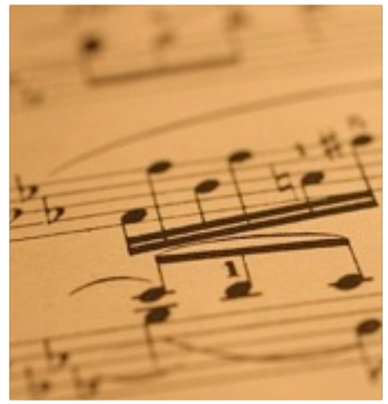
what's an algorithm?

origins



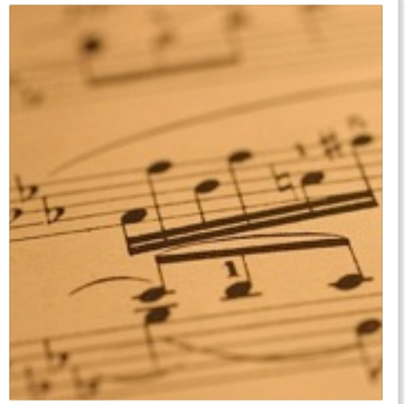
the word "algorithm" comes from Muhammad ibn Musa al-Khwarizmi (780-850), the name of a Persian mathematician who worked in the House of Wisdom, in Bagdad

definition



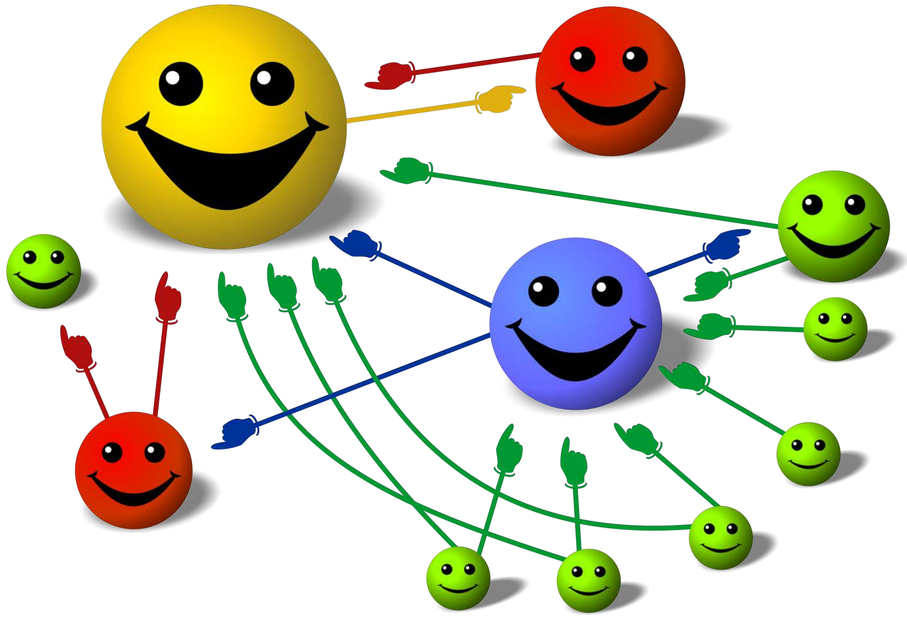
an algorithm is a well-defined computational procedure that takes some input values and produces some output values as the solution of a well-specified problem

definition



an algorithm can be expressed in a natural language (e.g., English), as a computer program (e.g., in scala), or even in some hardware design, via the appropriate layout of transistors

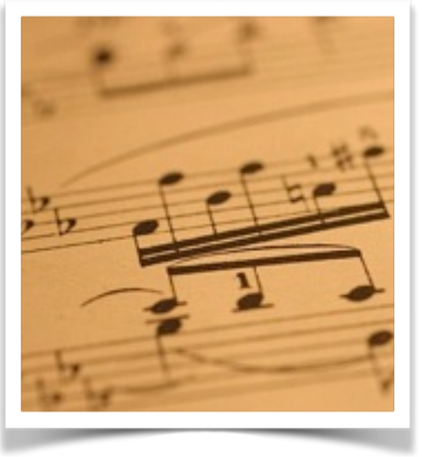
google

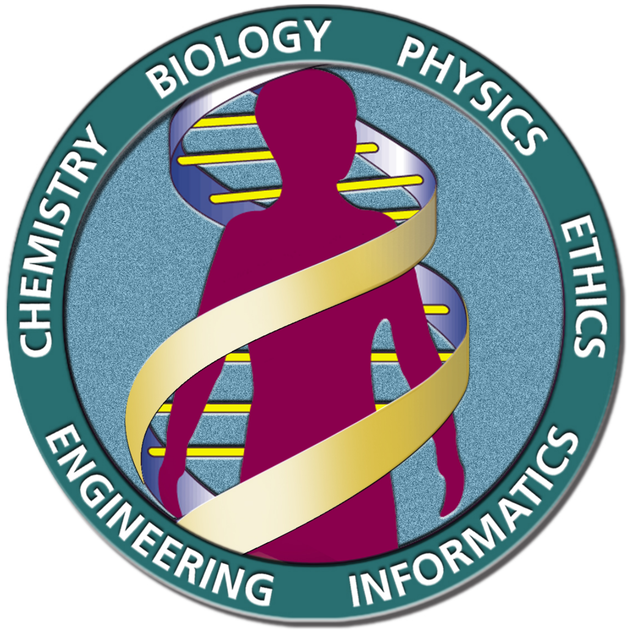


the PageRank algorithm was
developed at Stanford University by
Larry Page and Sergey Brin as part of a research project,
which led to a functional prototype at the origin of Google Inc.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

the human genome project





this project aimed at identifying the 20'000-25'000 genes in human DNA, based on 3.3 billion chemical base pairs, and at developing sophisticated algorithms for analyzing this data

from math to algorithms

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x \geq 0 \\ \sqrt{-x} & \text{if } x < 0 \end{cases}$$

```
function f(x : real)
if x ≥ 0
    f ← sqrt(x)
else
    f ← sqrt(–x)
```

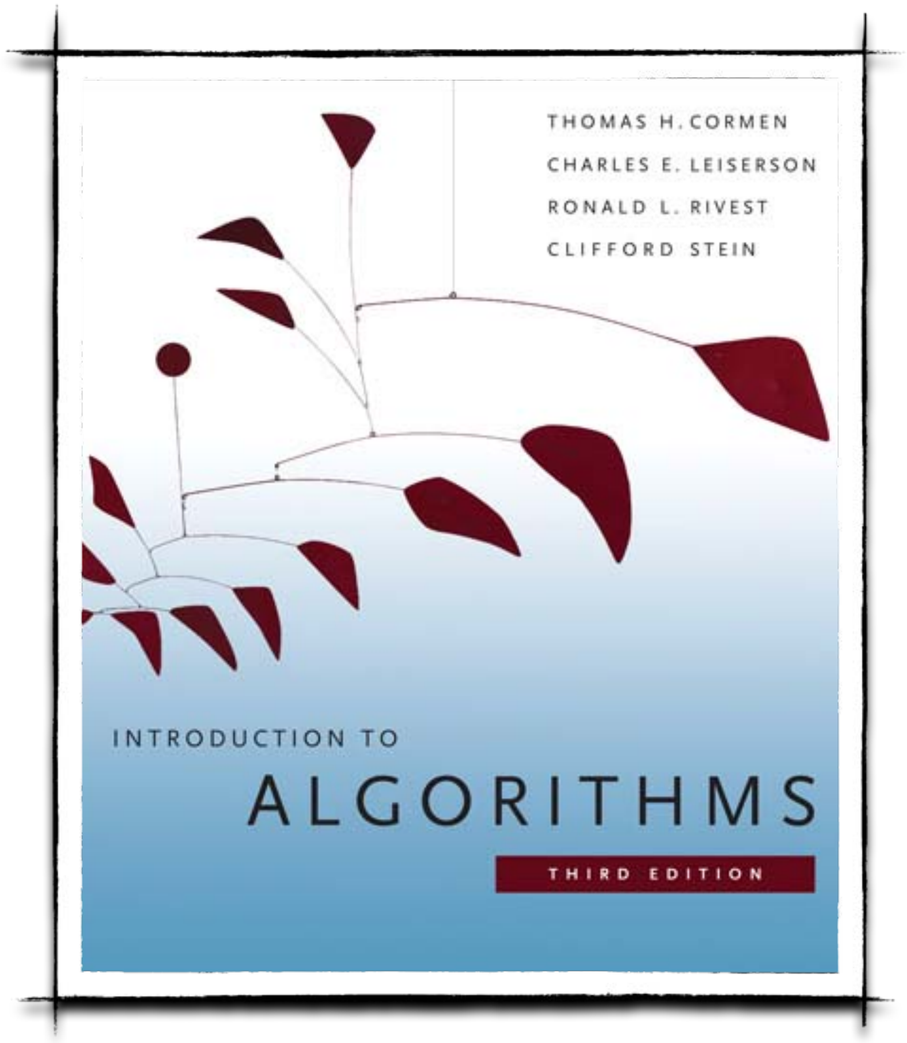
```
def f(x: Double) : Double = {
  if (x < 0) Math.sqrt(-x) else Math.sqrt(x)
}
```

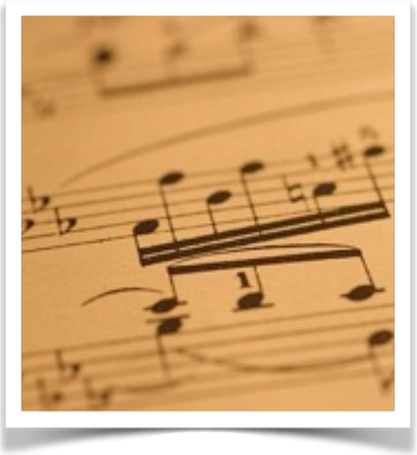
$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i$$

```
function f(x : array[1..n] of real)
f ← 0
for i = 1 to n do
    f ← f + x[i]
```

```
def f(X: List[Double]) : Double = {
  var sum = 0.0
  for (x <- X) {
    sum = sum + x
  }
  sum
}
```

book





INTRODUCTION TO ALGORITHMS

BY T. H. CORMEN ET AL.

3RD EDITION

MIT PRESS, 2009

the sorting problem

specification

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

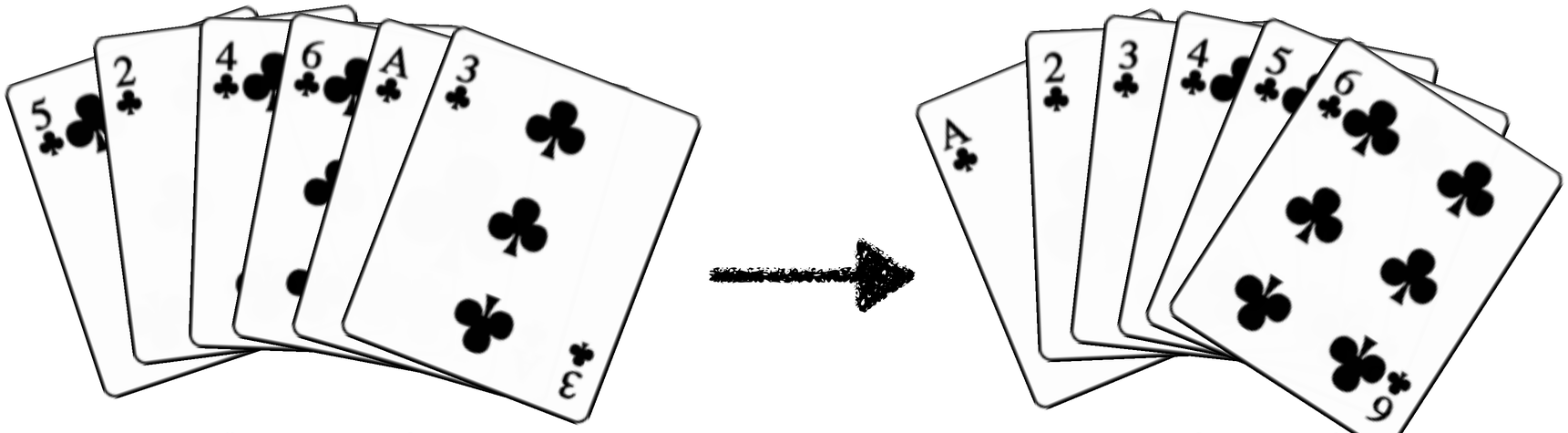
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

the sequence of numbers is stored in arrays and numbers are also referred to as keys

example

input: $A = \langle 5, 2, 4, 6, 1, 3 \rangle$

output: $A = \langle 1, 2, 3, 4, 5, 6 \rangle$



input

output

sorting algorithms

there exists various sorting

- insertion sort
- merge sort
- heap sort
- quick sort
- bucket sort
- etc...

insertion sort

pseudo-code

```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

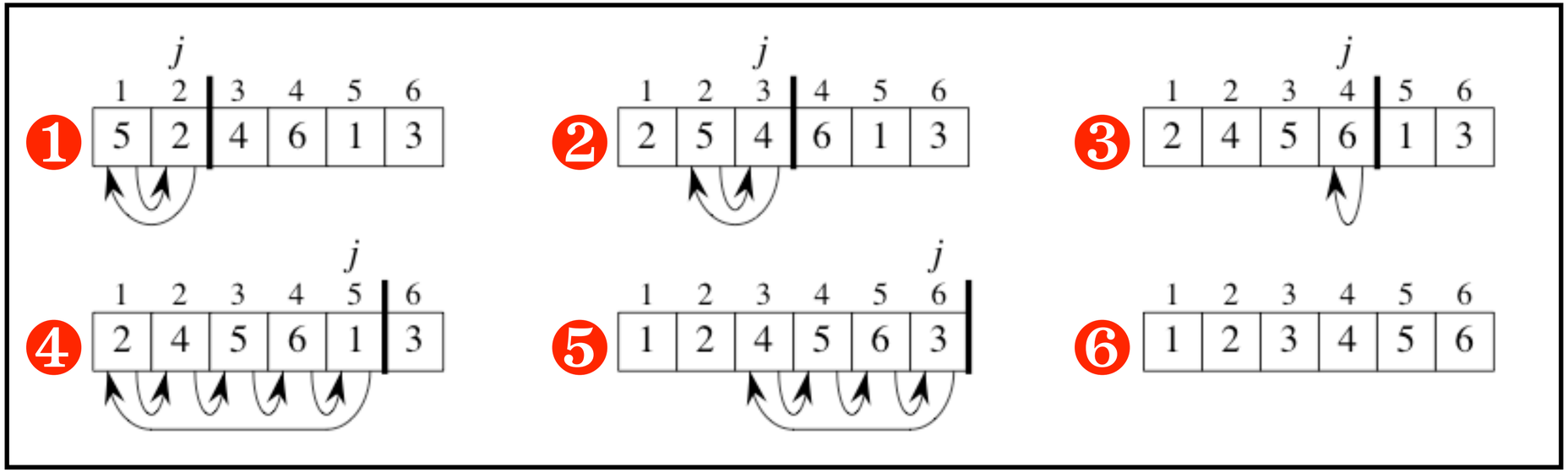
$A[1..n]$ is an array of integer of size n

array A is sorted in place

example

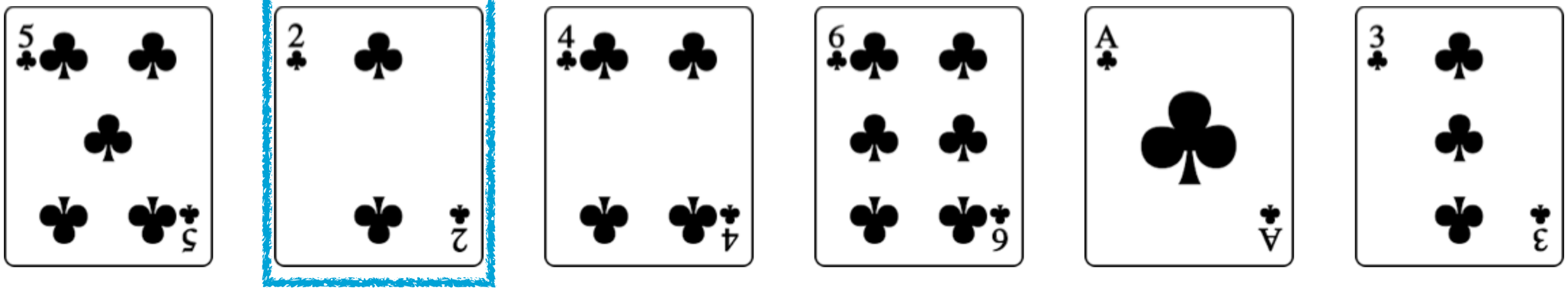
overview

```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

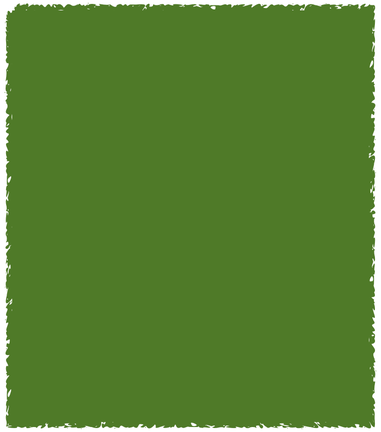


insertion sort

1 j=2 3 4 5 6



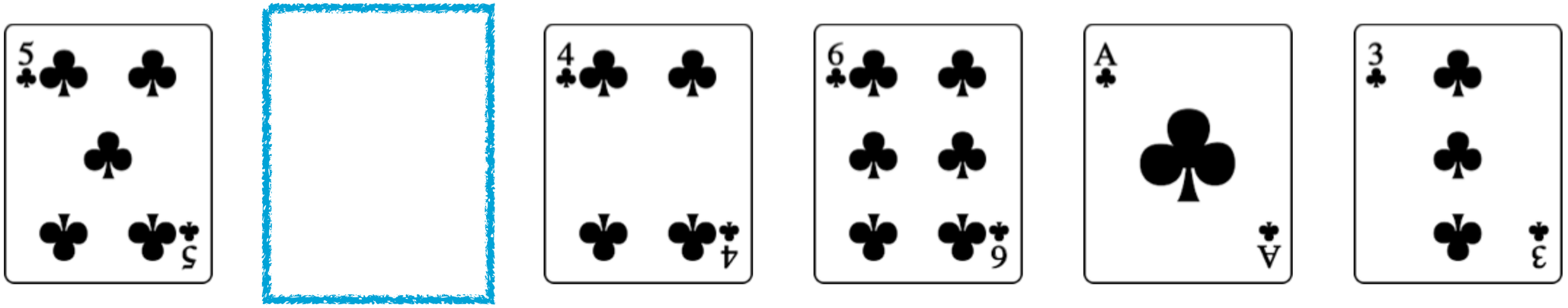
key



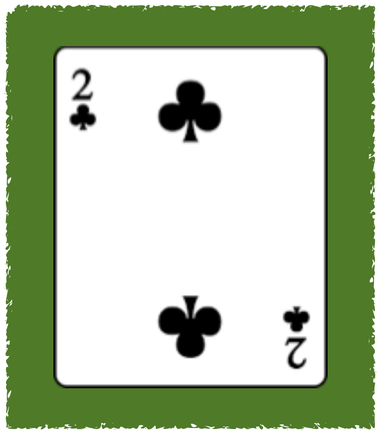
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 j=2 3 4 5 6



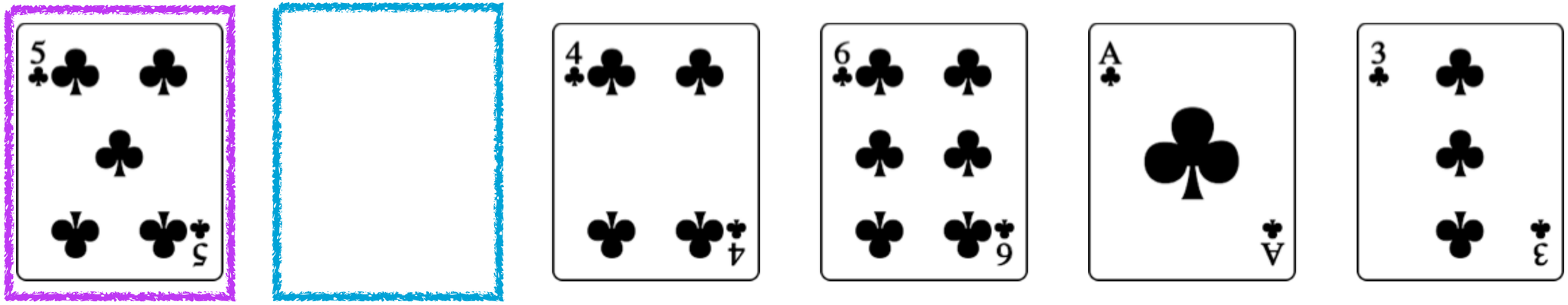
key



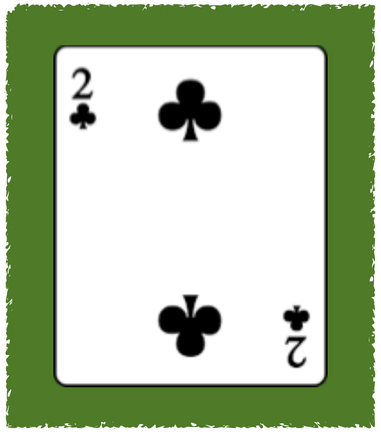
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 j=2 3 4 5 6



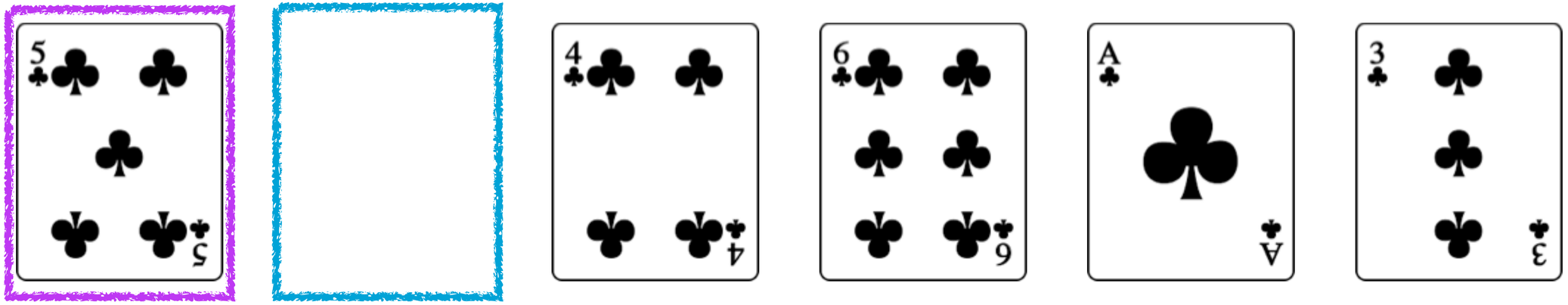
key



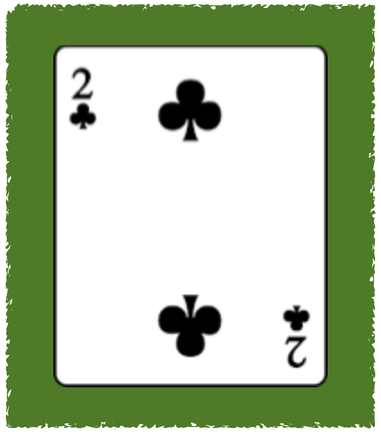
```
for j ← 2 to n
    do key ← A[j]
    i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 j=2 3 4 5 6



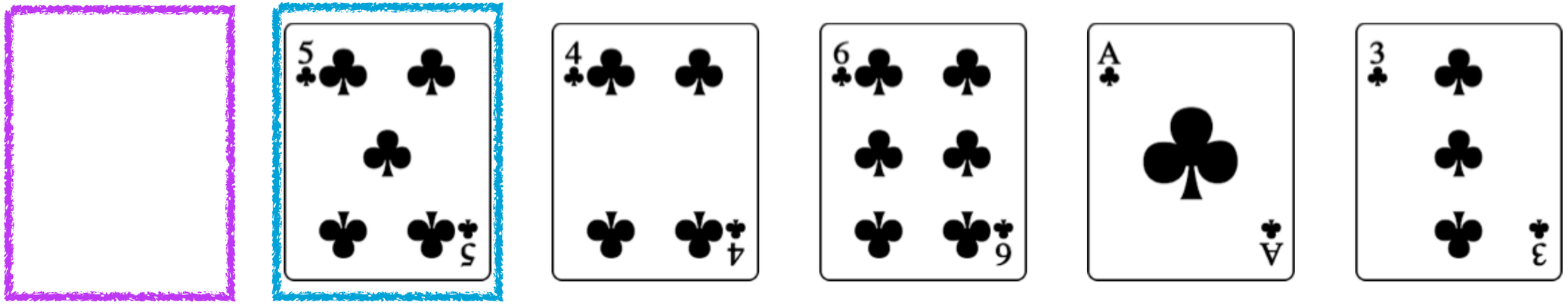
key



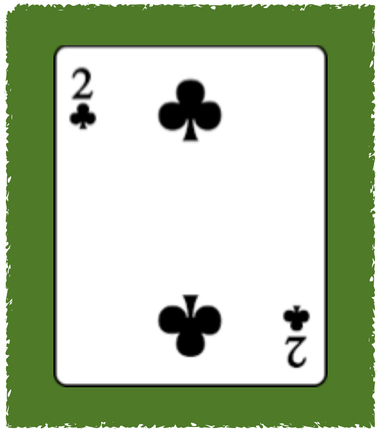
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
    → while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 j=2 3 4 5 6



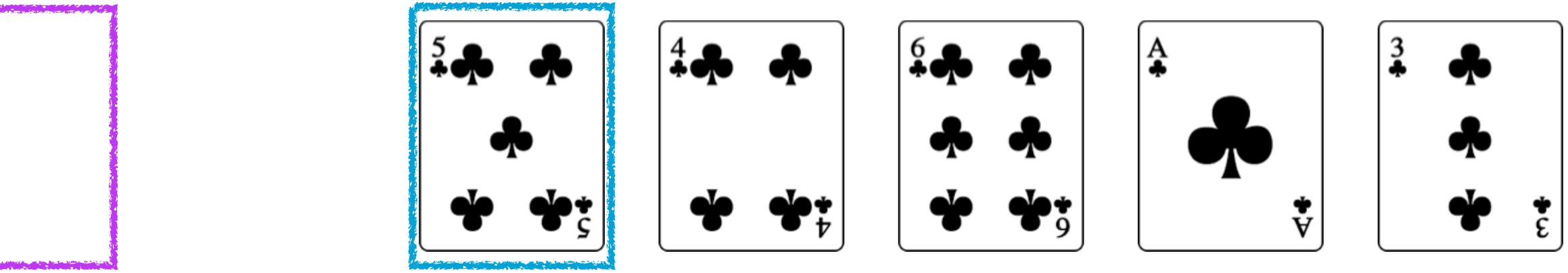
key



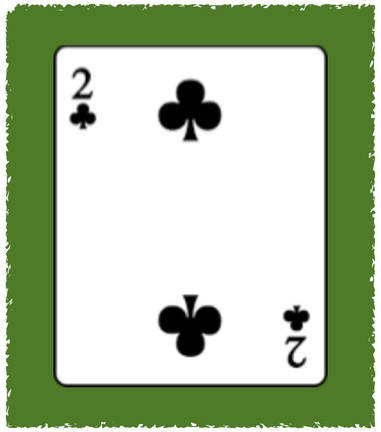
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i= 0 1 j= 2 3 4 5 6



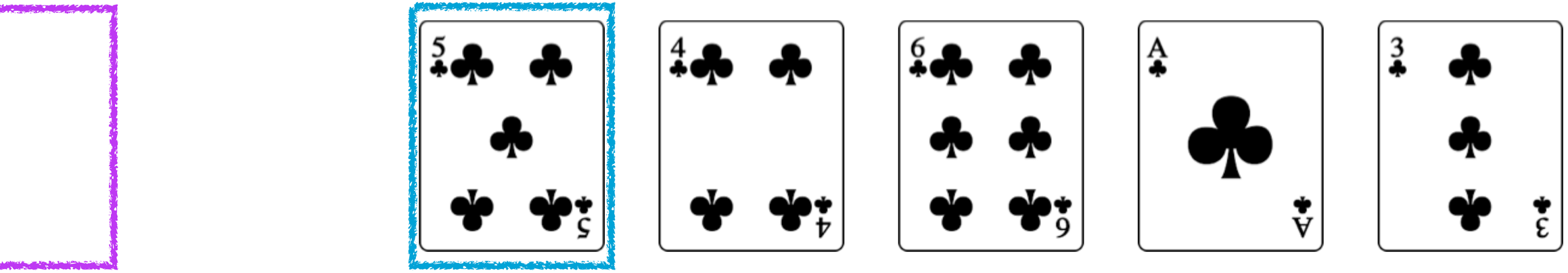
key



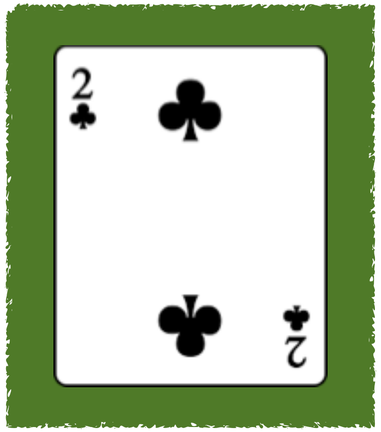
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

i= 0 1 j= 2 3 4 5 6



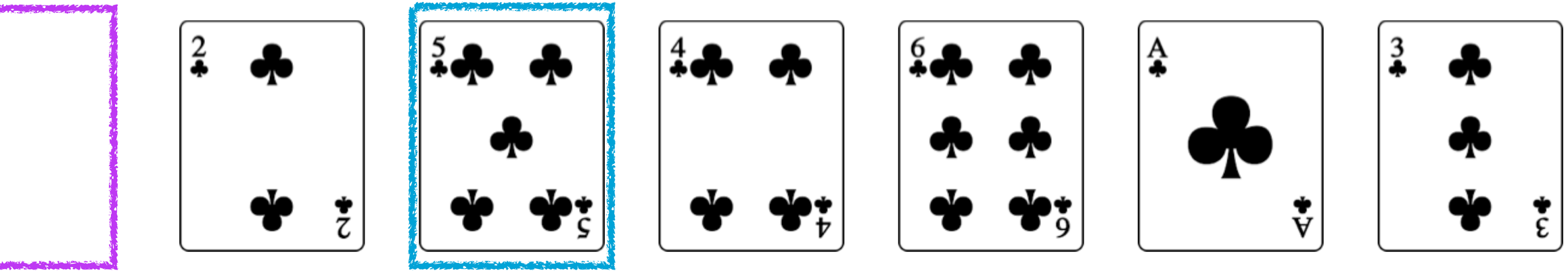
key



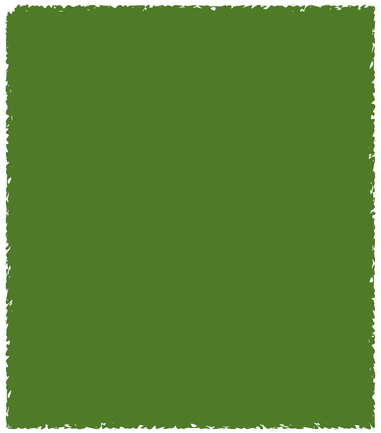
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
    → while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i= 0 1 j= 2 3 4 5 6



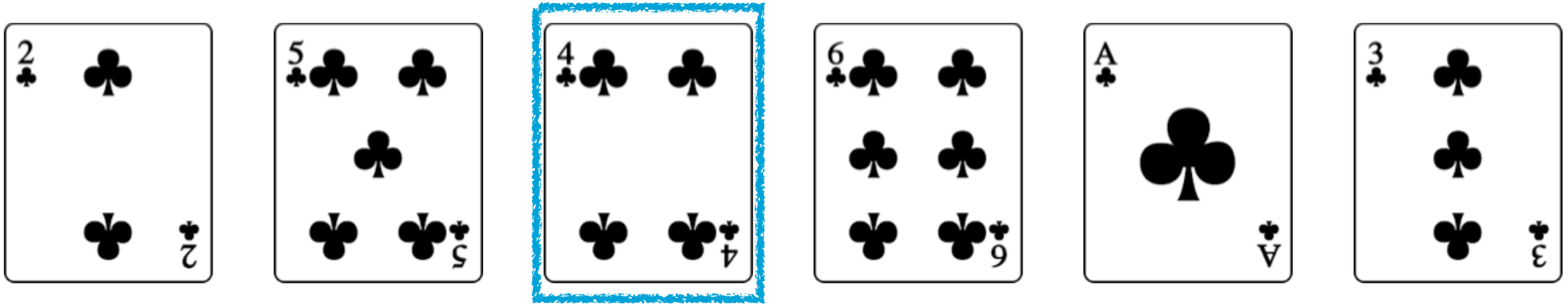
key



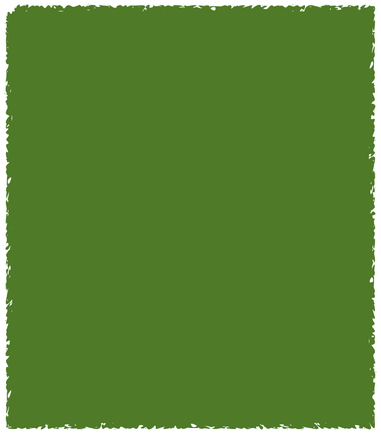
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 j=3 4 5 6



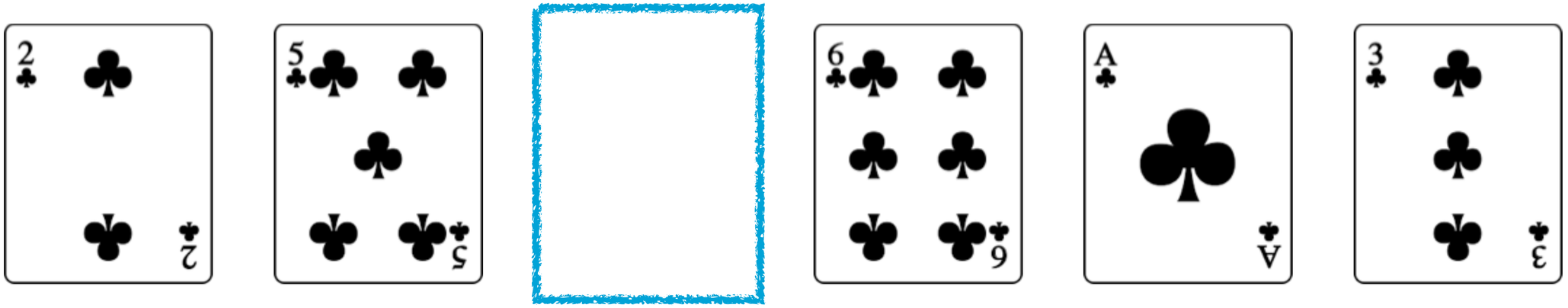
key



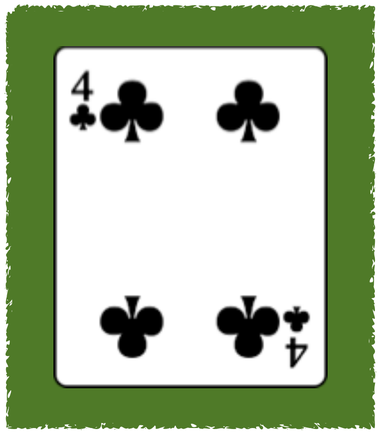
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 j=3 4 5 6



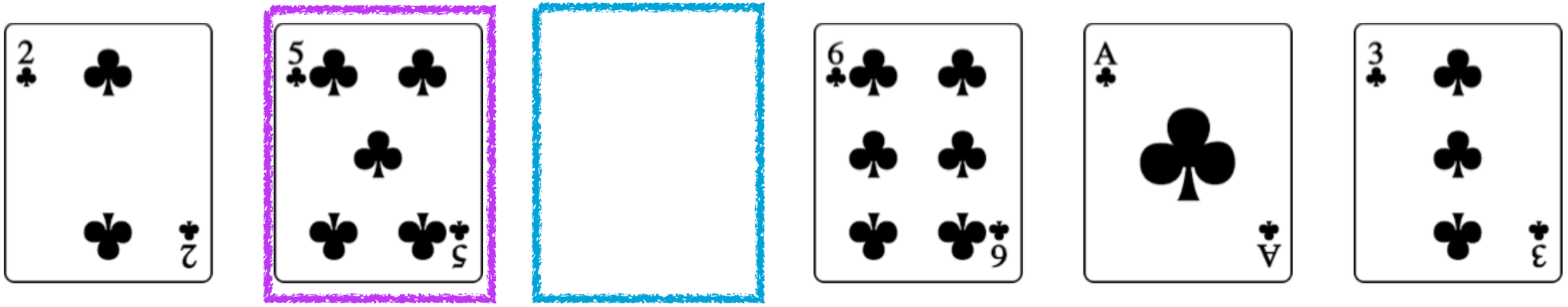
key



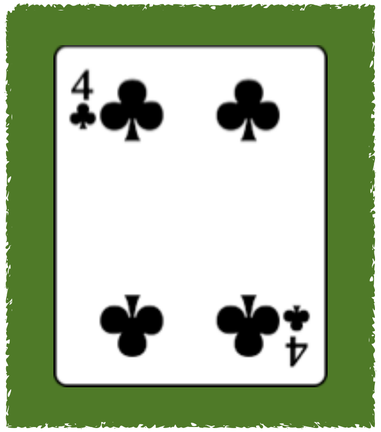
```
for j ← 2 to n
do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i=2 j=3 4 5 6



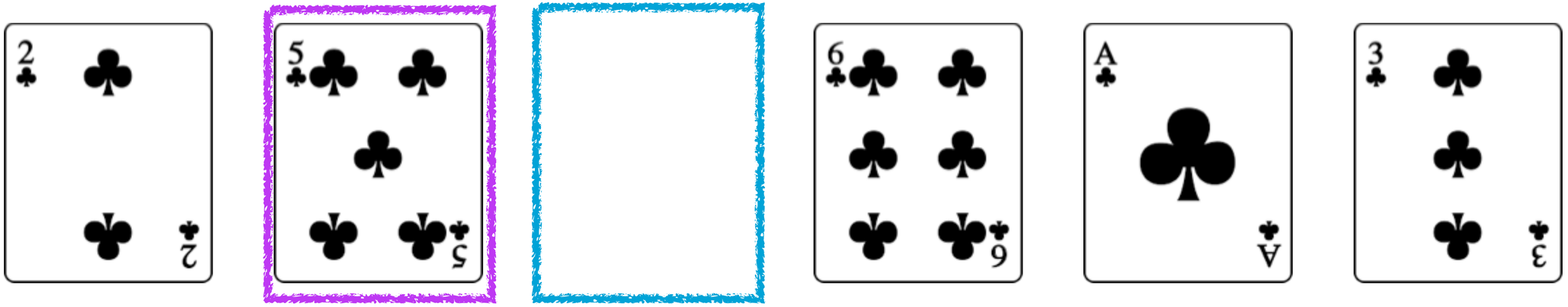
key



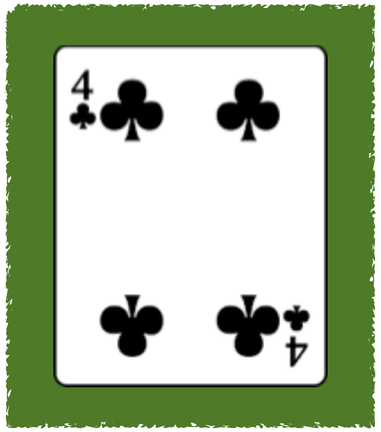
```
for j ← 2 to n
    do key ← A[j]
    i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

insertion sort

1 i=2 j=3 4 5 6



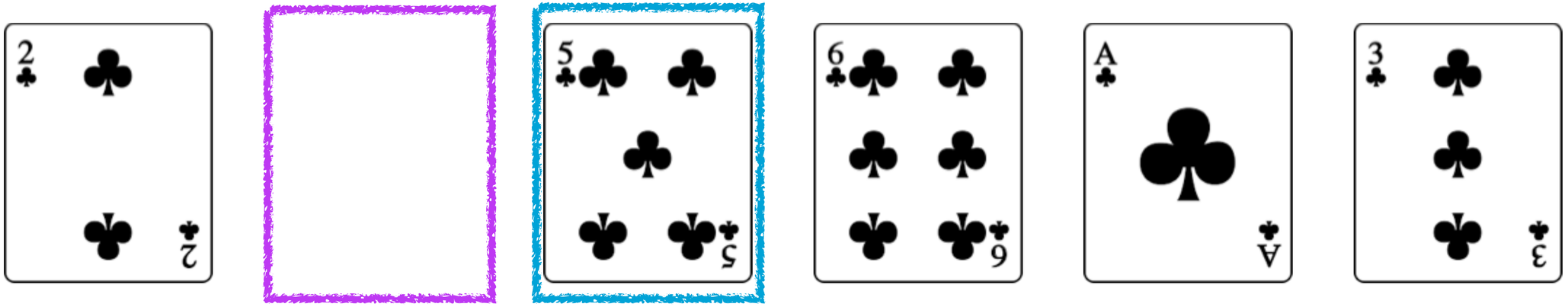
key



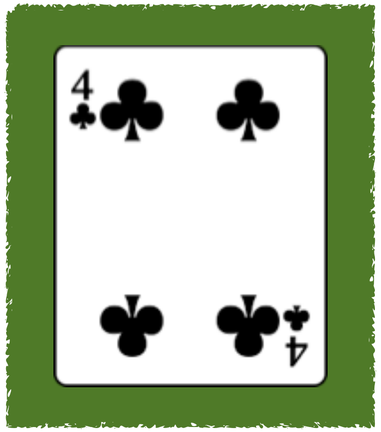
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i=2 j=3 4 5 6



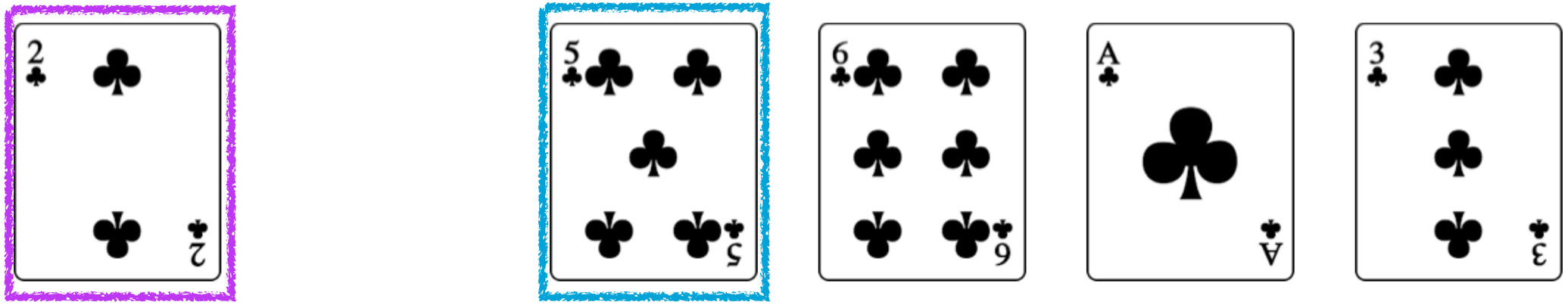
key



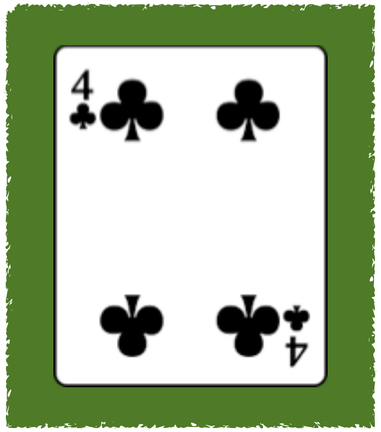
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 j=3 4 5 6



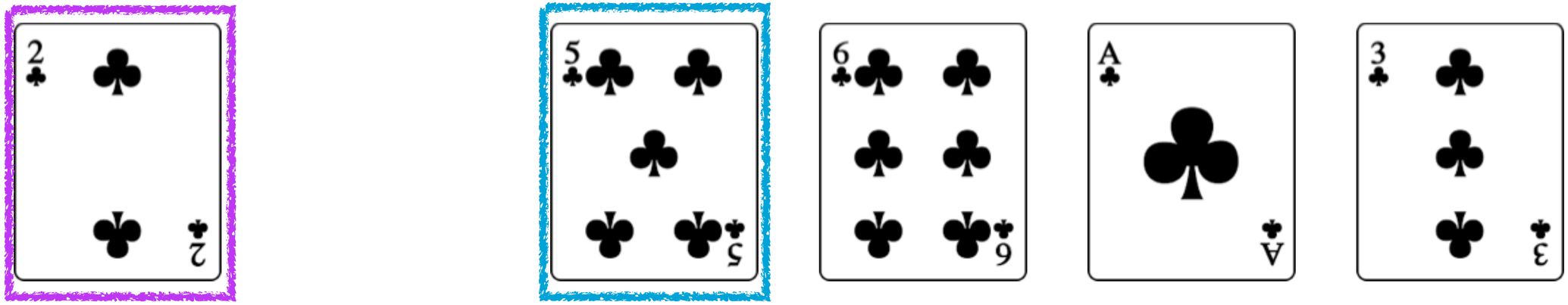
key



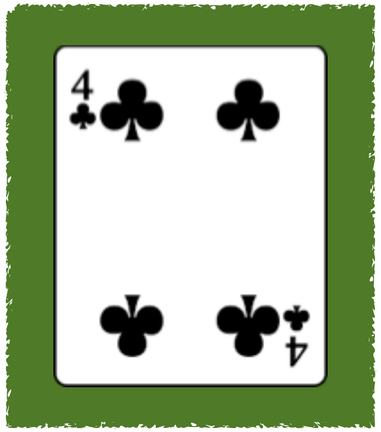
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 j=3 4 5 6



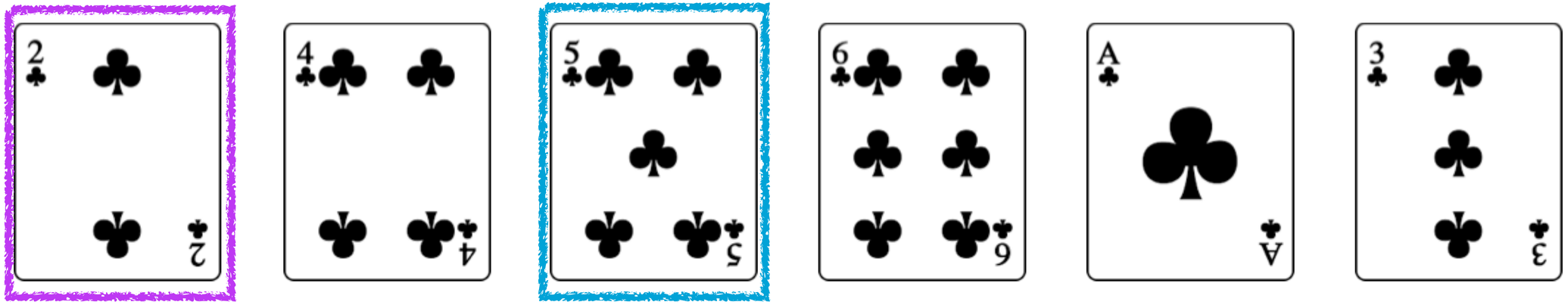
key



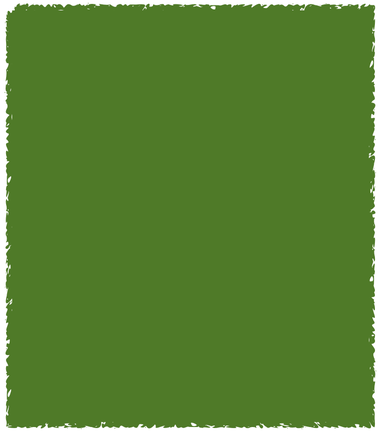
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
    → while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 j=3 4 5 6



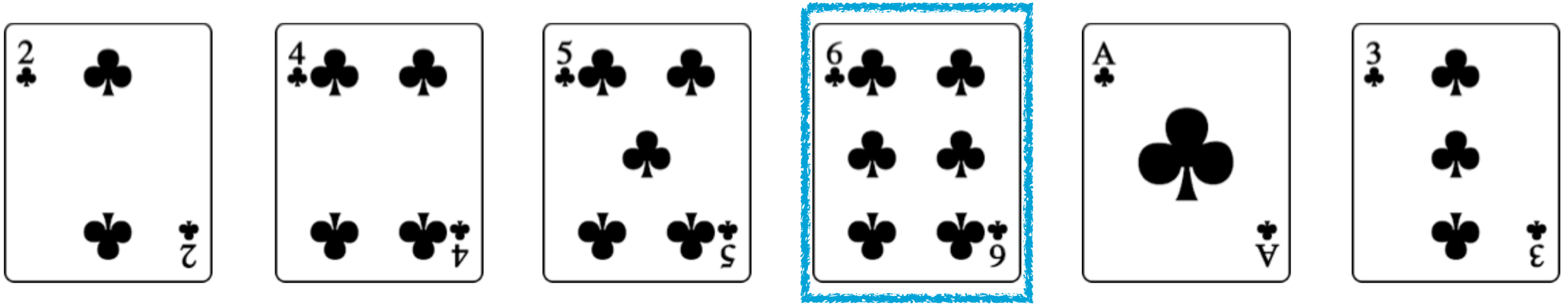
key



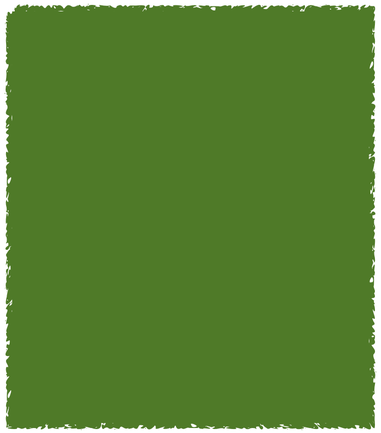
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 j=4 5 6



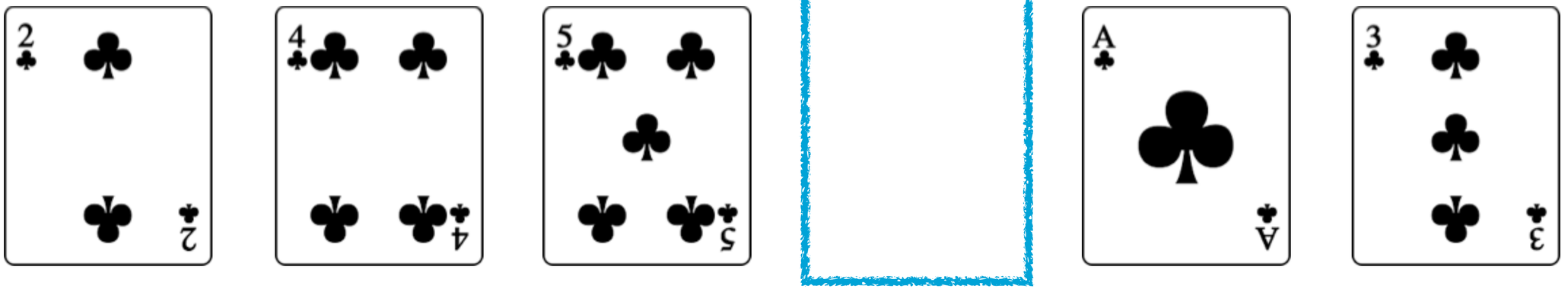
key



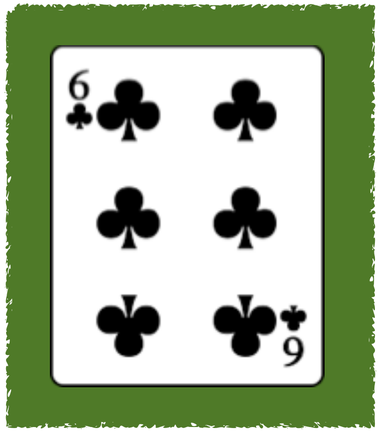
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 j=4 5 6



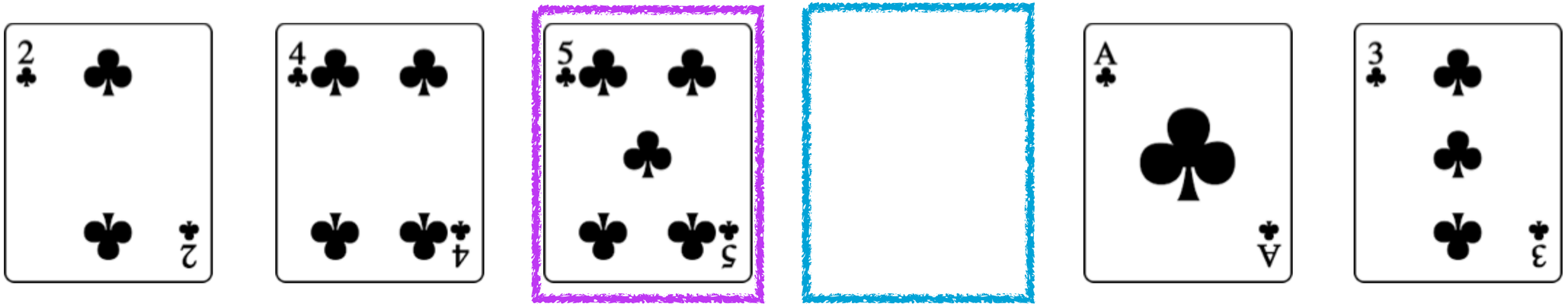
key



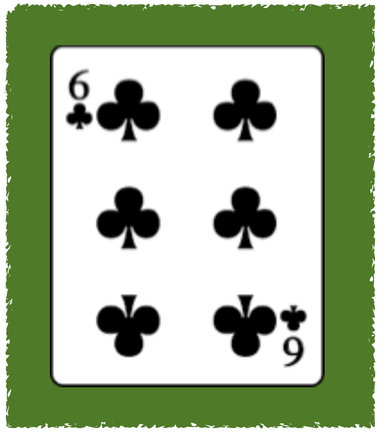
```
for j ← 2 to n
do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 j=4 5 6



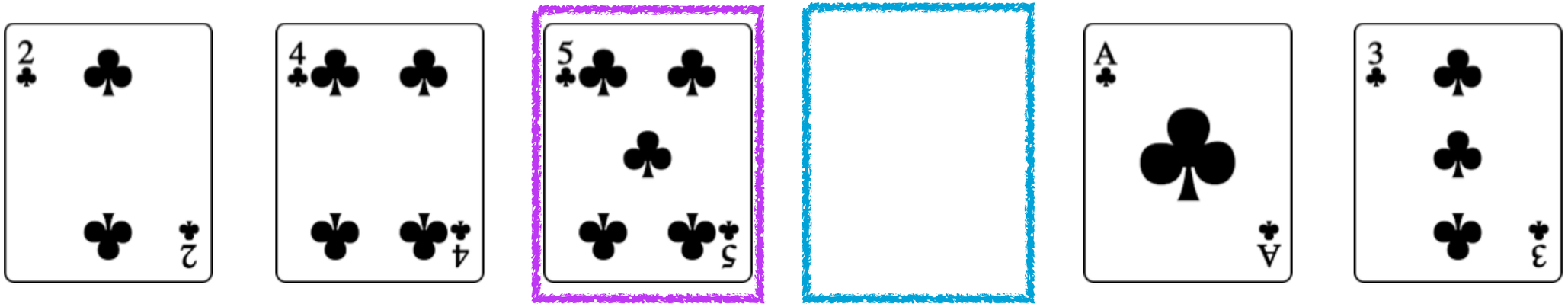
key



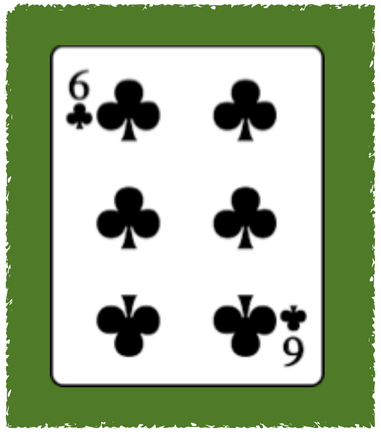
```
for j ← 2 to n
    do key ← A[j]
    i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

insertion sort

1 2 i=3 j=4 5 6



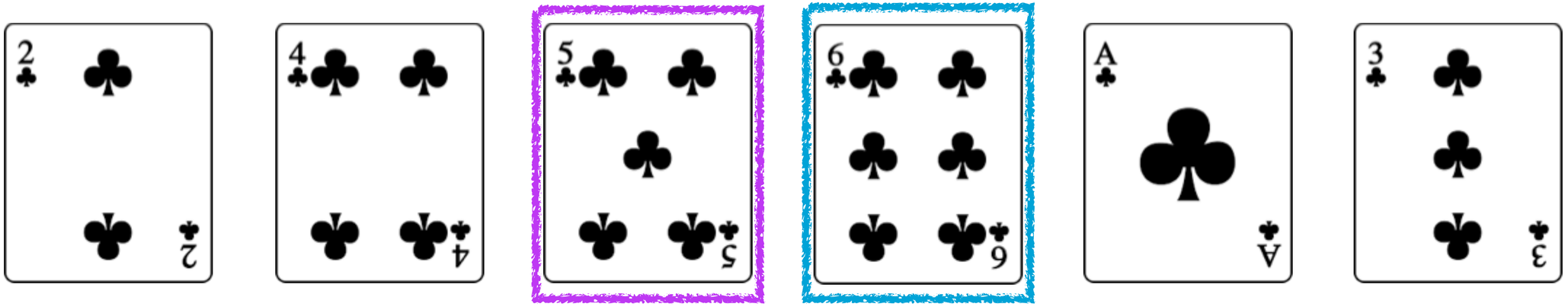
key



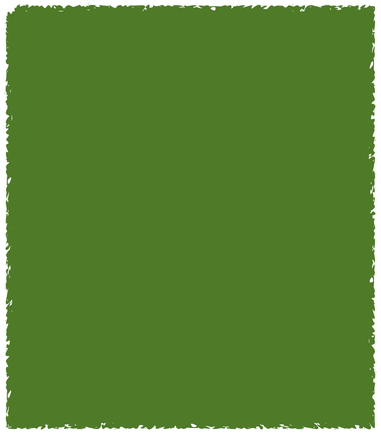
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 j=4 5 6



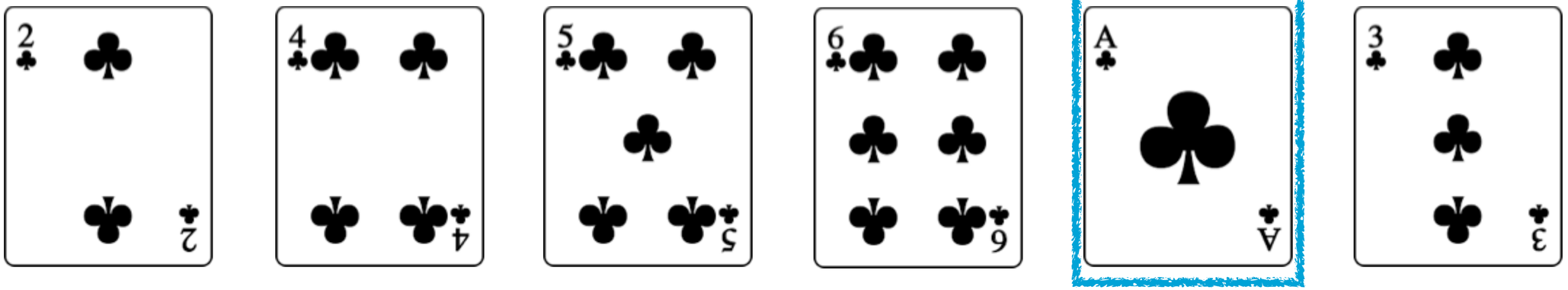
key



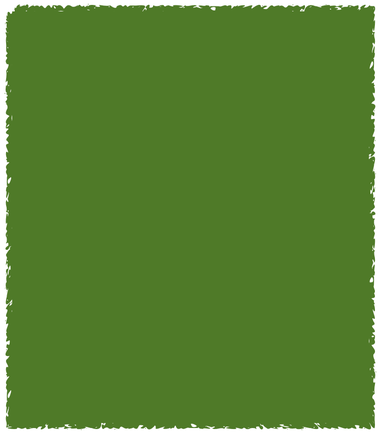
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 j=5 6



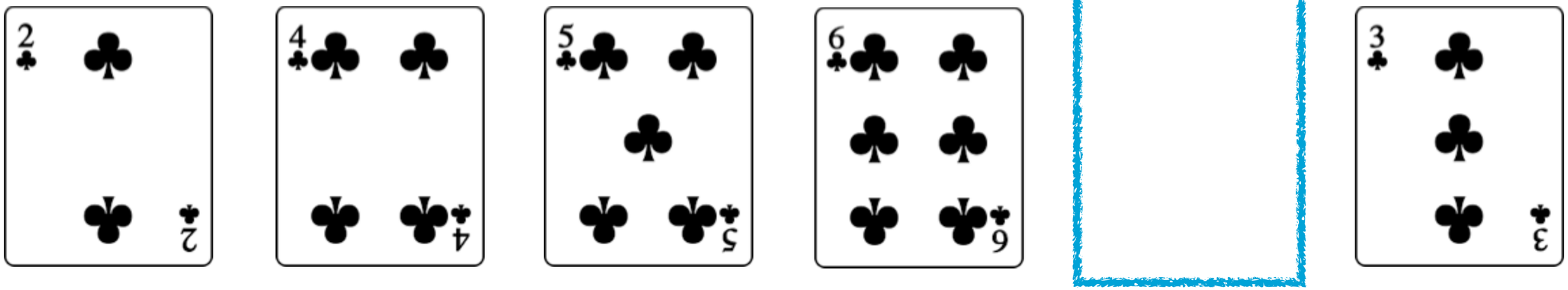
key



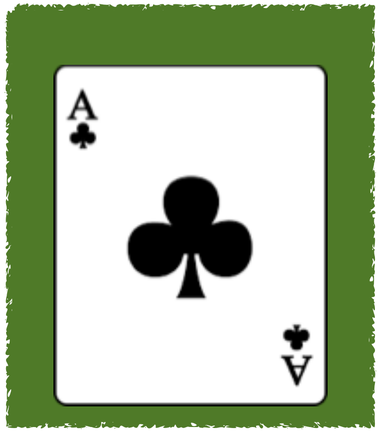
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 j=5 6



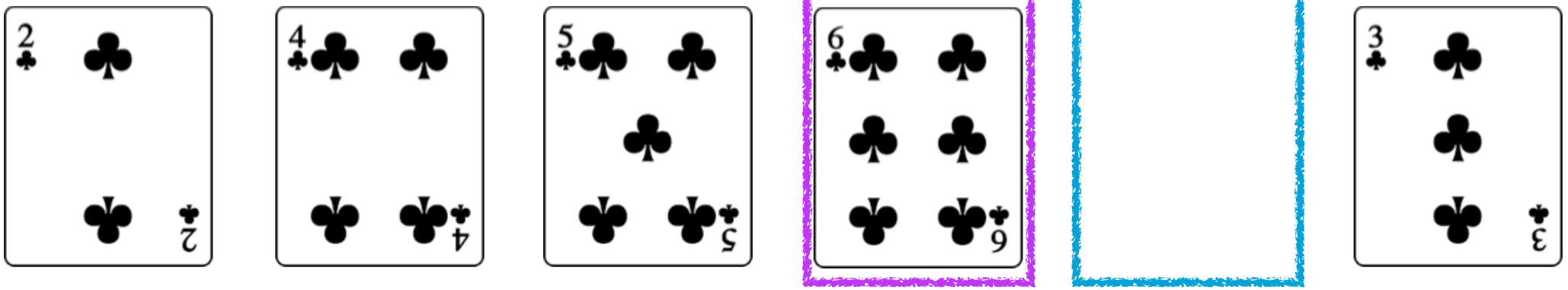
key



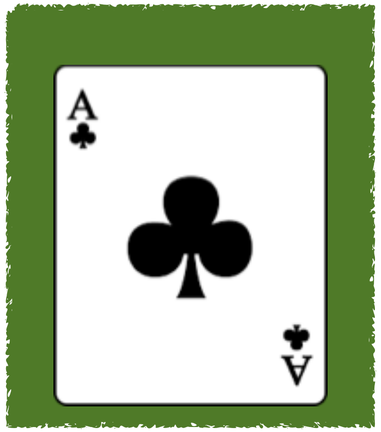
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 i=4 j=5 6



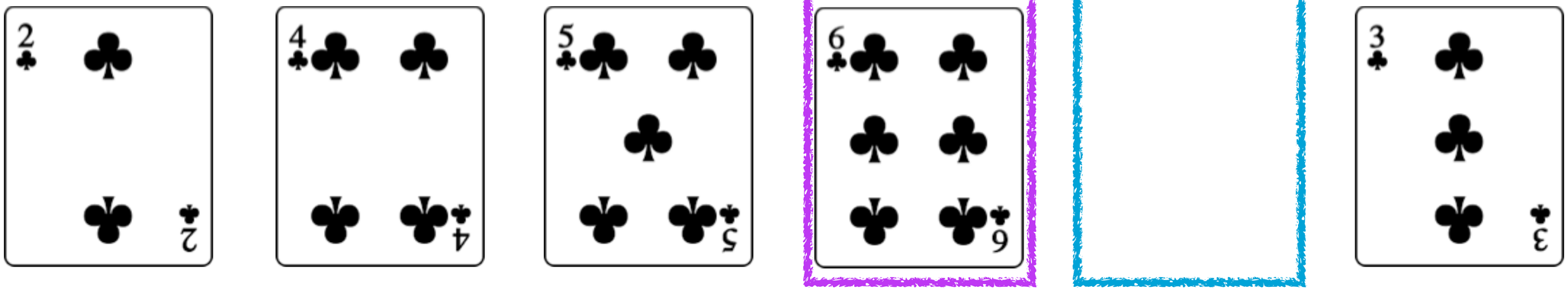
key



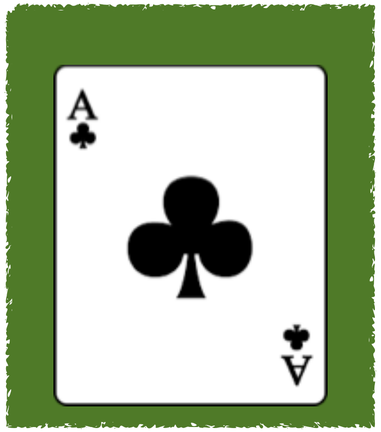
```
for j ← 2 to n
    do key ← A[j]
    i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

insertion sort

1 2 3 i=4 j=5 6



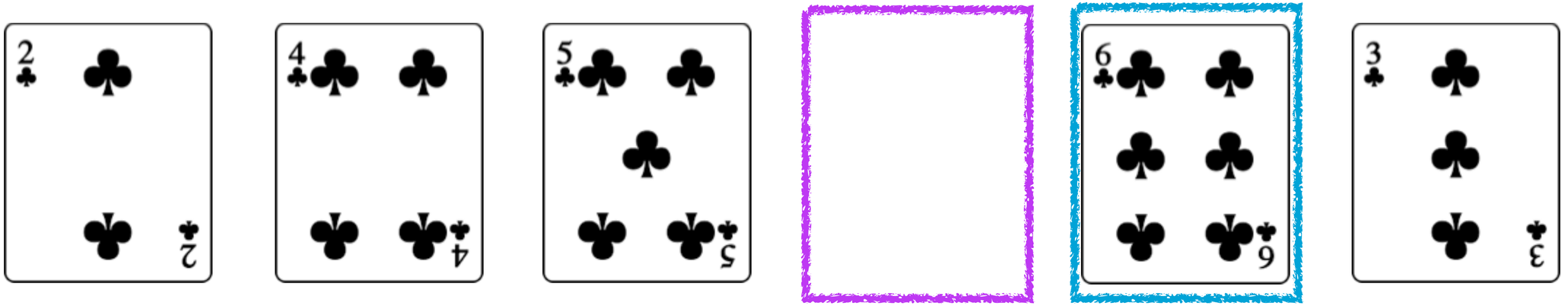
key



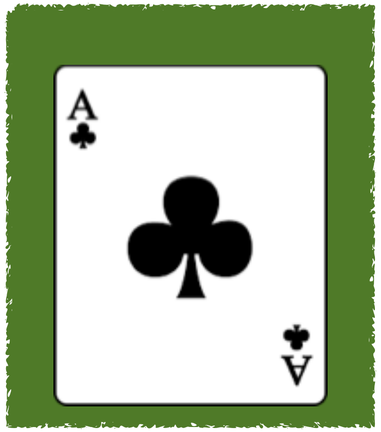
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
    → while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 i=4 j=5 6



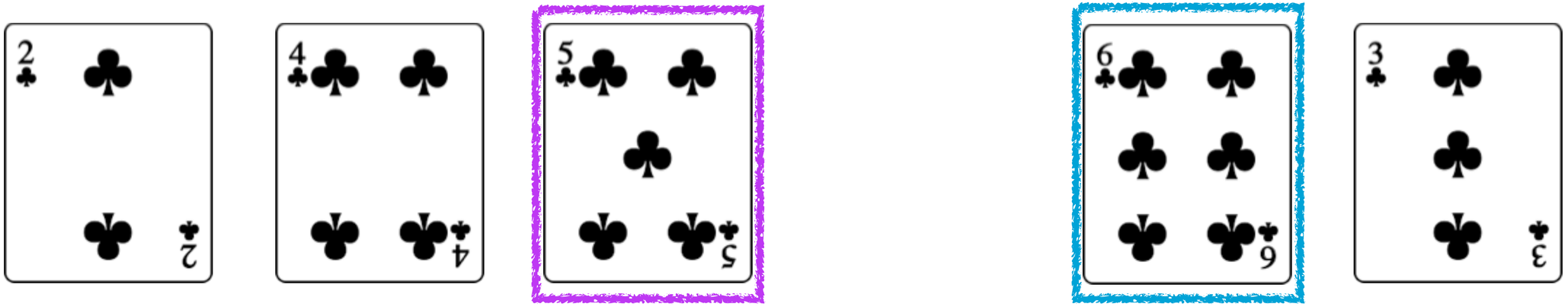
key



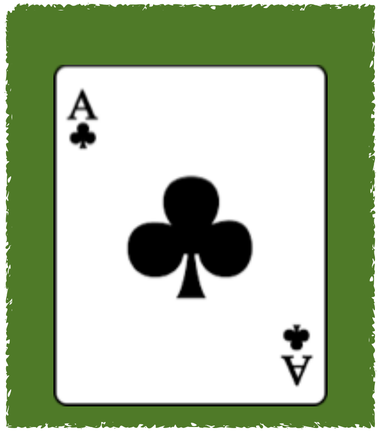
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 j=5 6



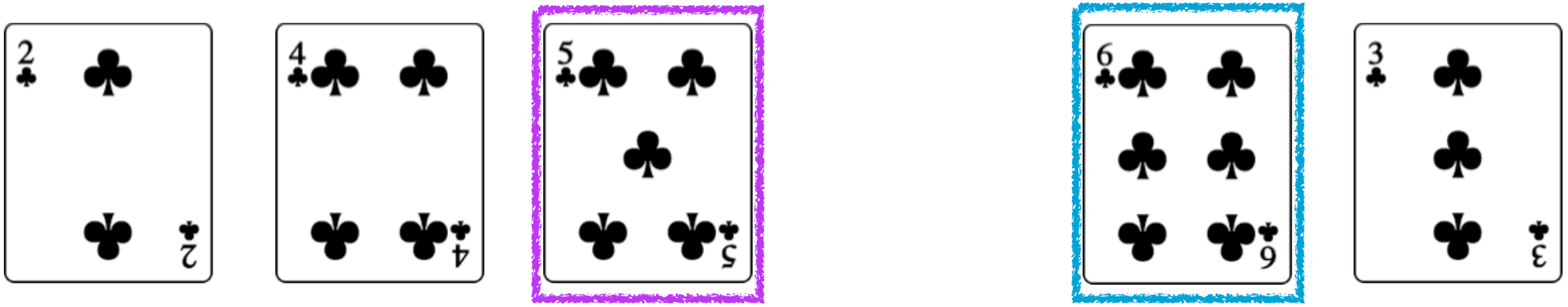
key



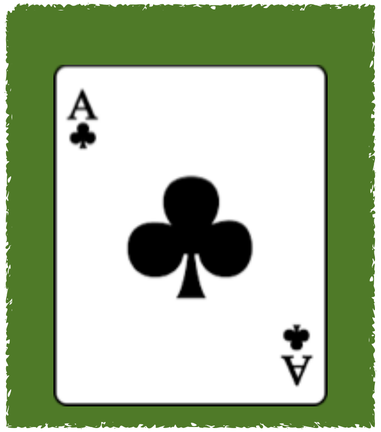
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 j=5 6



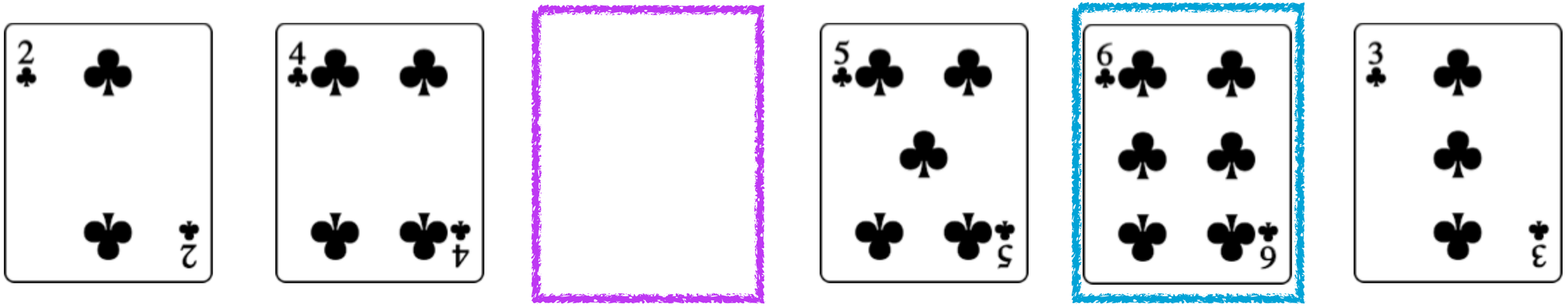
key



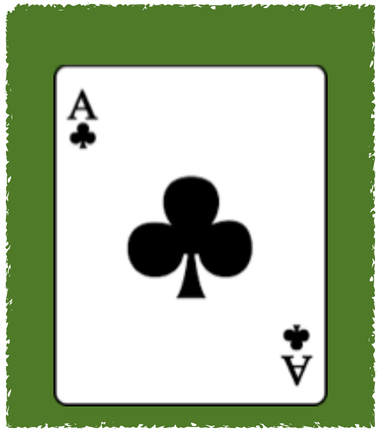
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 j=5 6



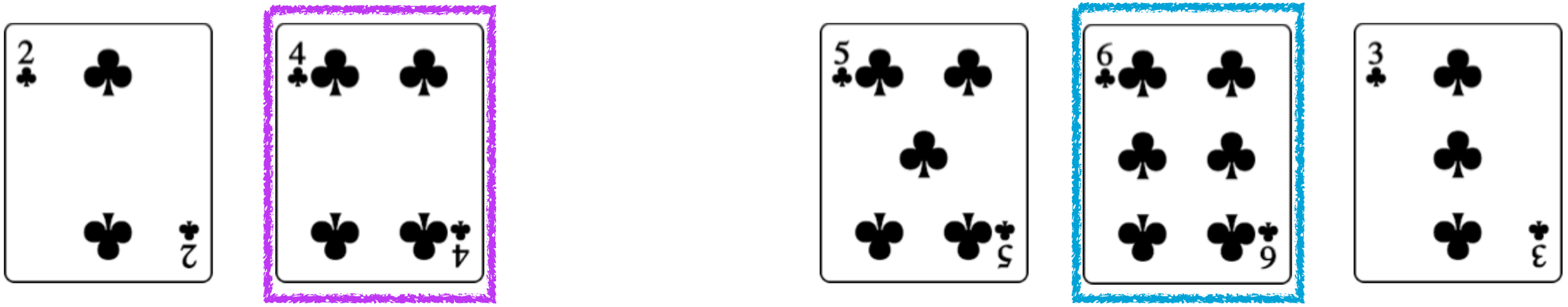
key



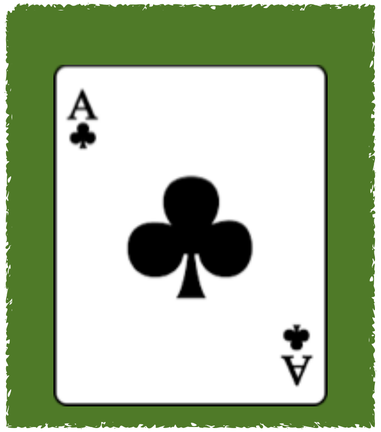
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i=2 3 4 j=5 6



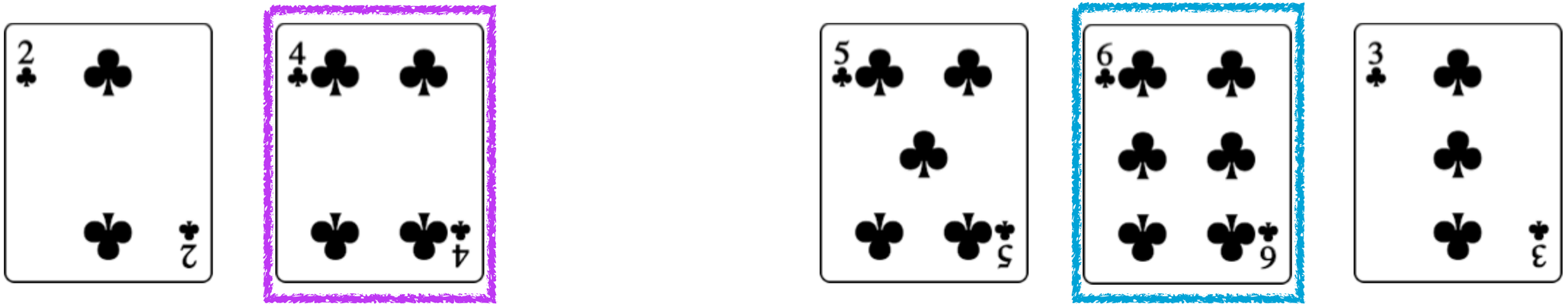
key



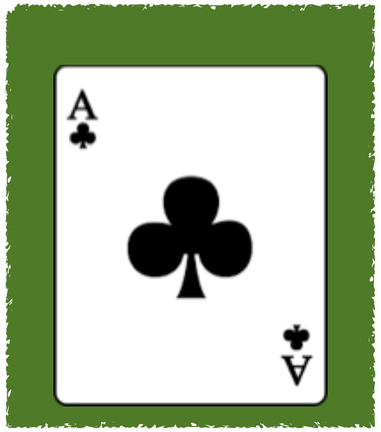
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i=2 3 4 j=5 6

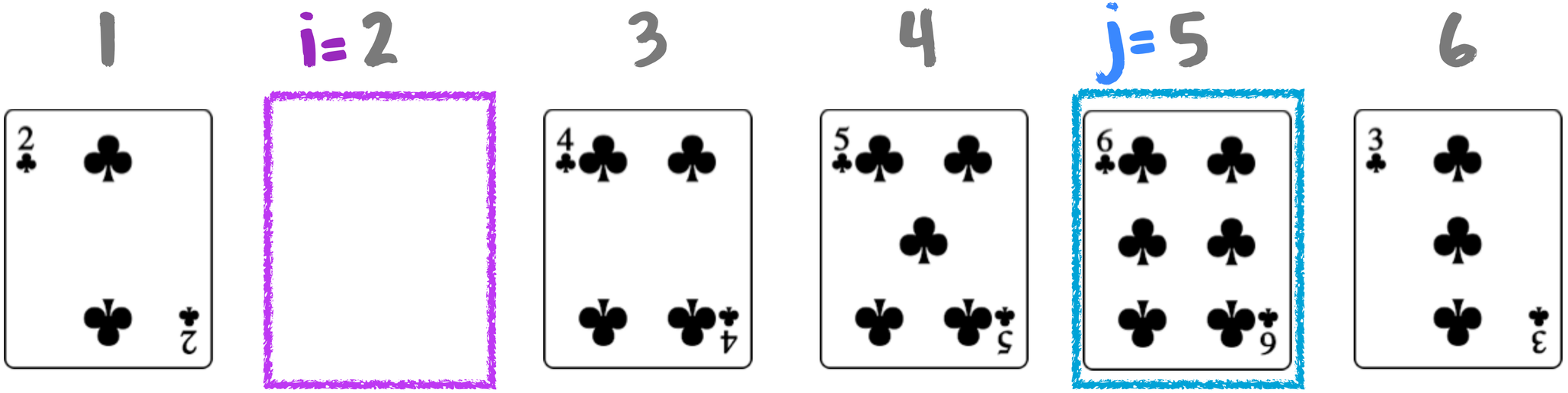


key

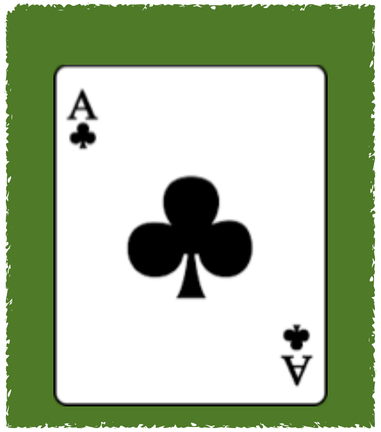


```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort



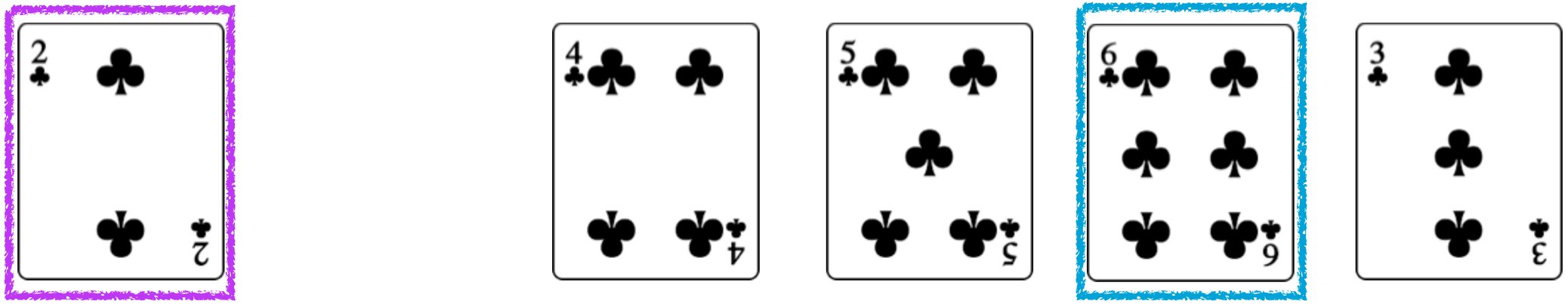
key



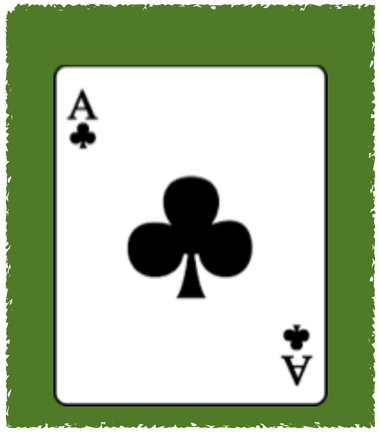
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 3 4 j=5 6



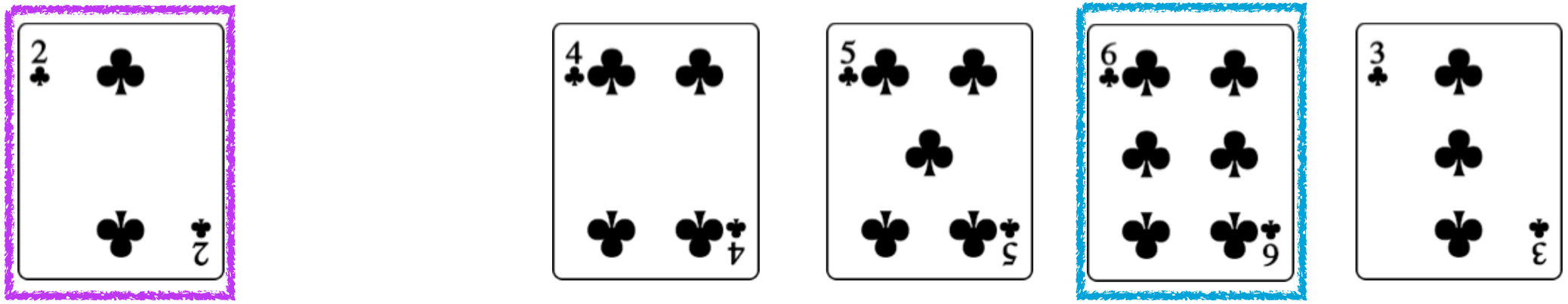
key



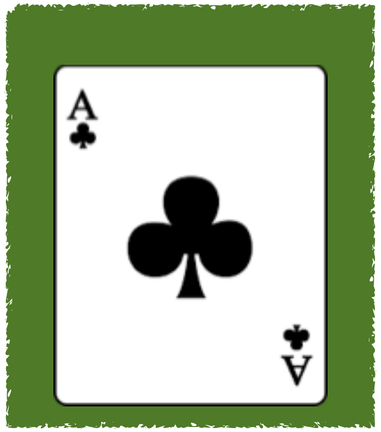
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 3 4 j=5 6



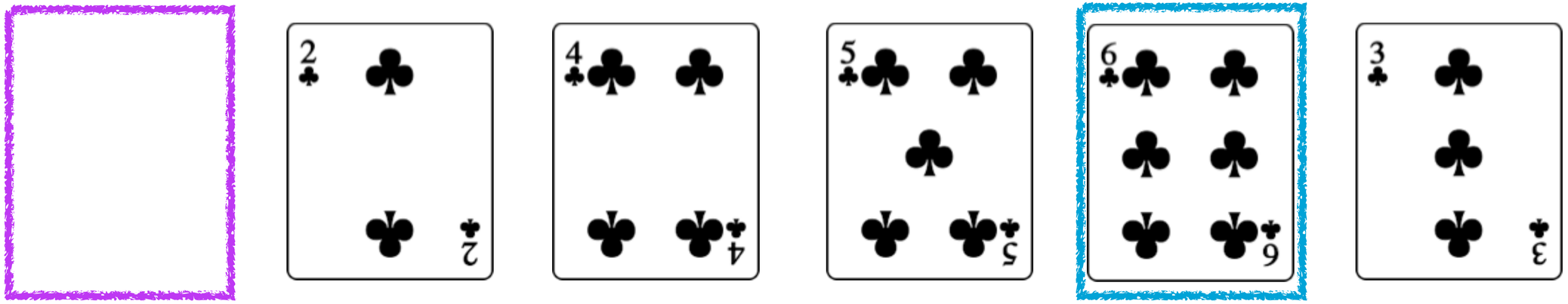
key



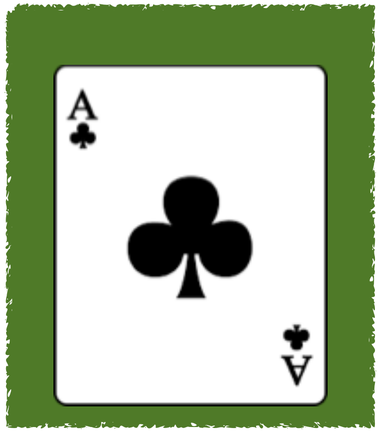
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=1 2 3 4 j=5 6



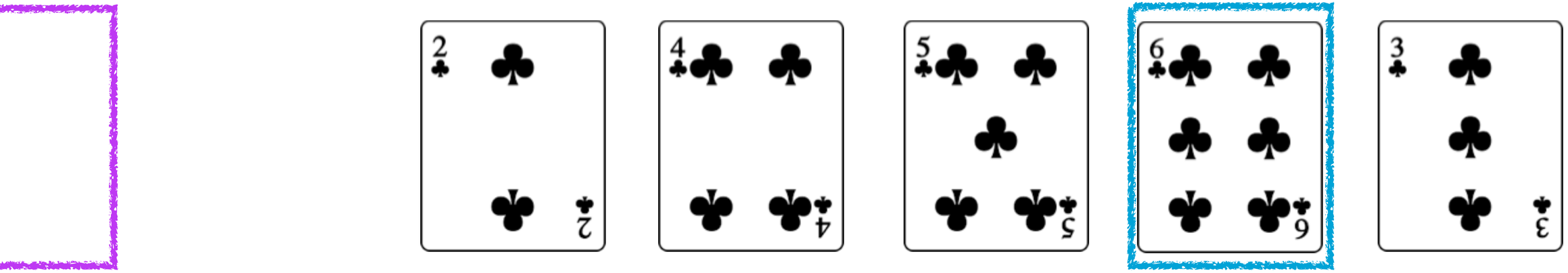
key



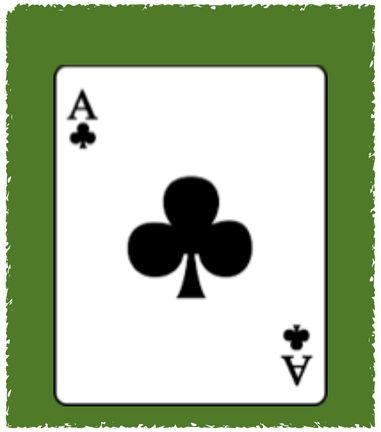
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=0 1 2 3 4 j=5 6



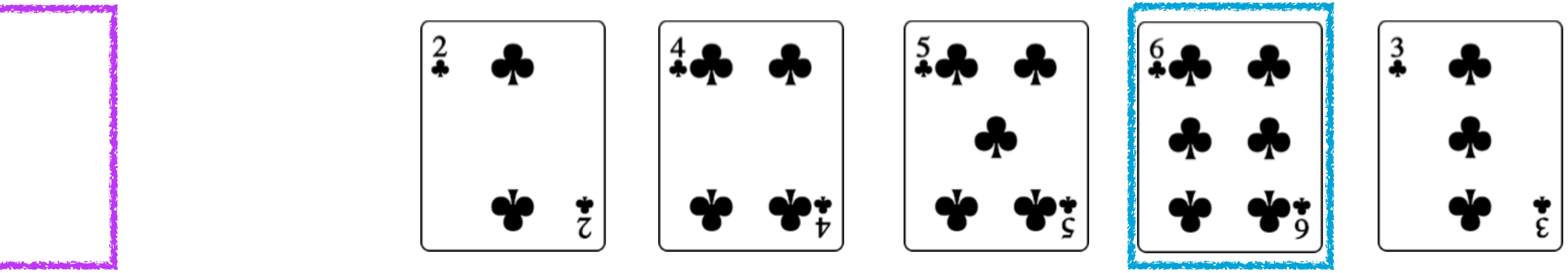
key



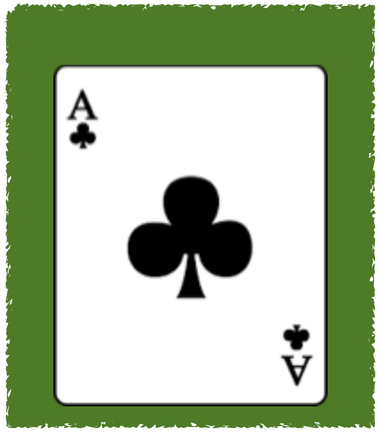
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=0 1 2 3 4 j=5 6



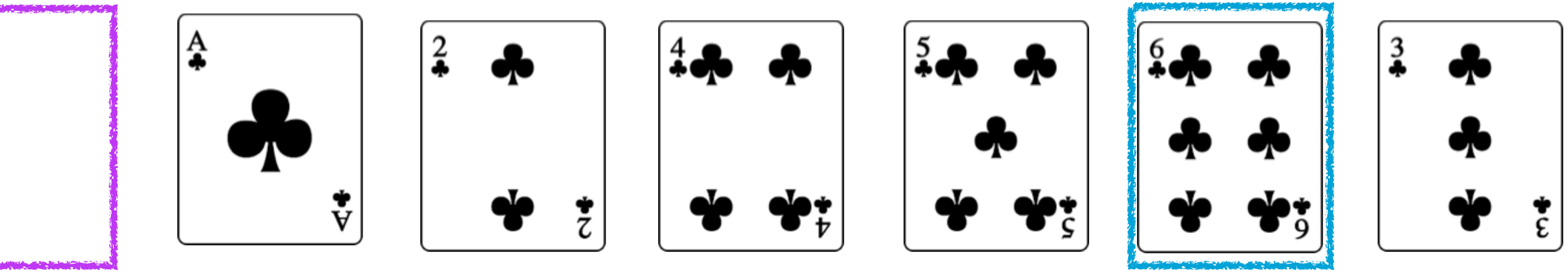
key



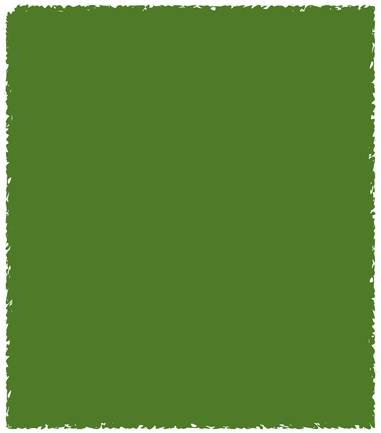
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

i=0 1 2 3 4 j=5 6



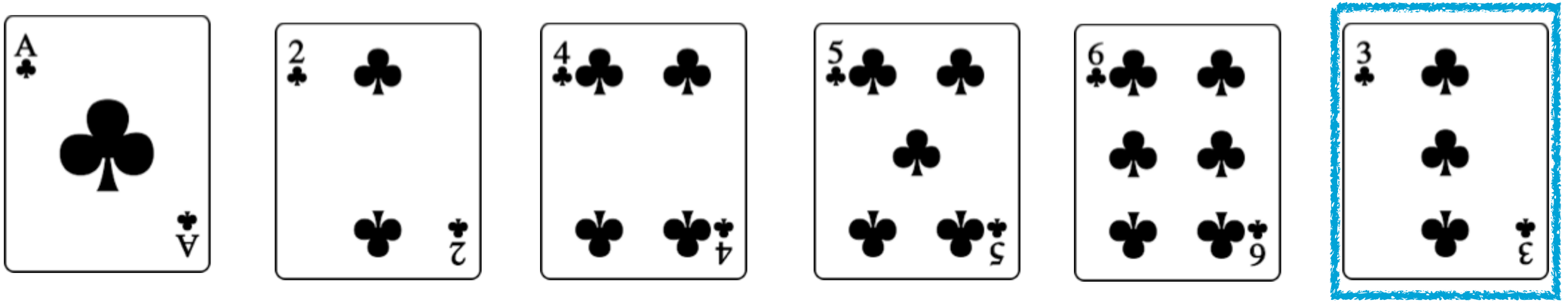
key



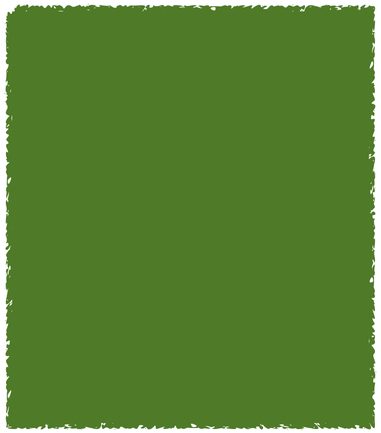
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 5 j=6



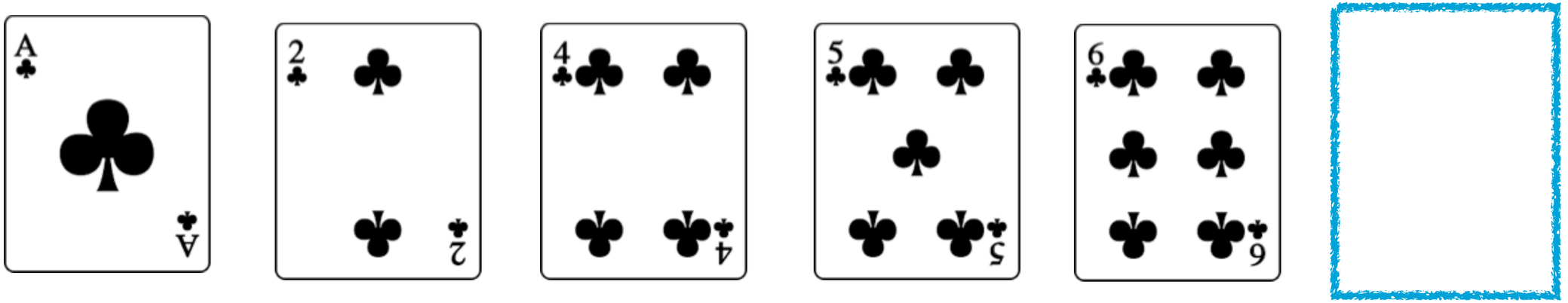
key



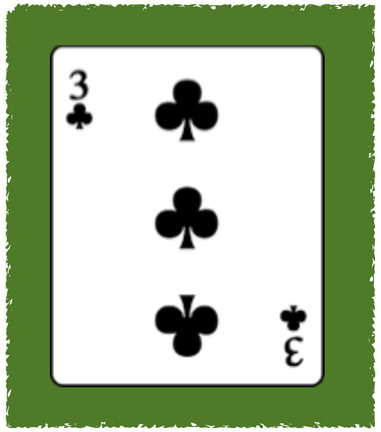
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 5 j=6



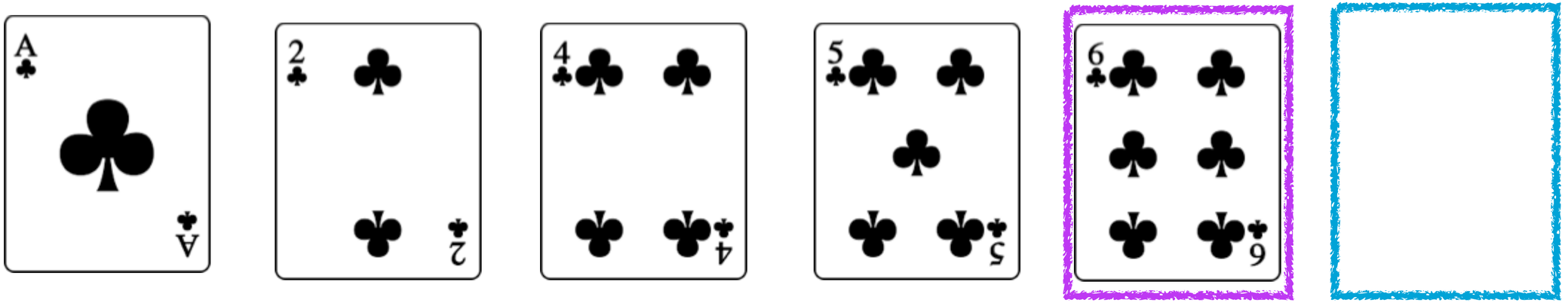
key



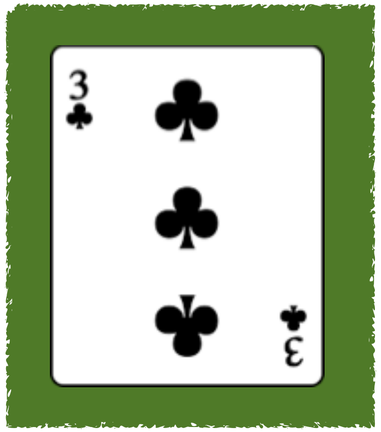
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 i=5 j=6



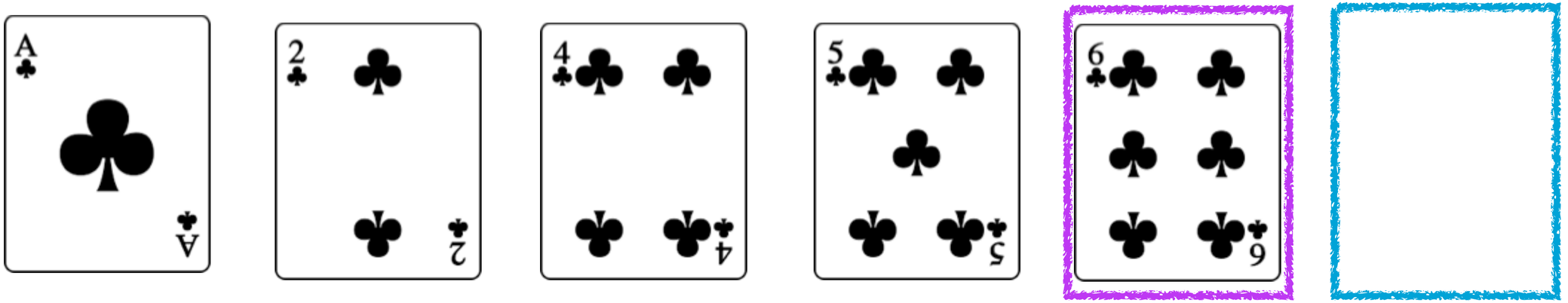
key



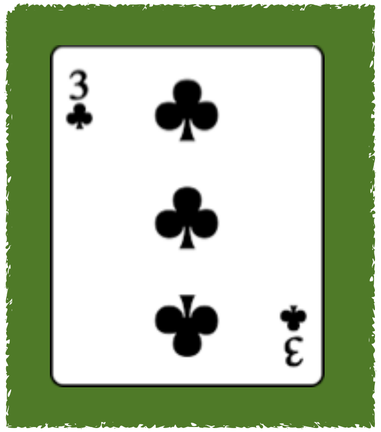
```
for j ← 2 to n
    do key ← A[j]
    i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

insertion sort

1 2 3 4 i=5 j=6



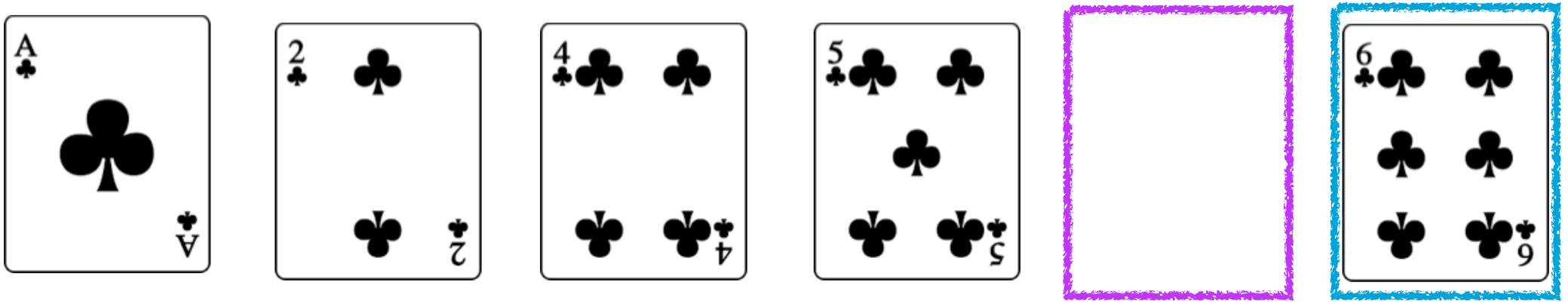
key



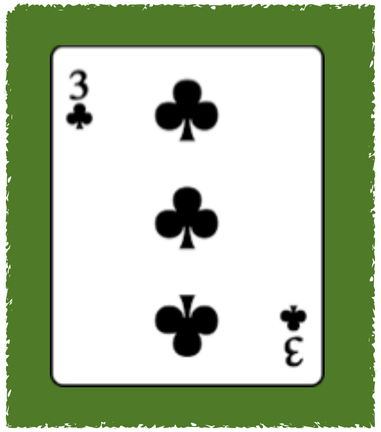
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
    → while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 4 i=5 j=6



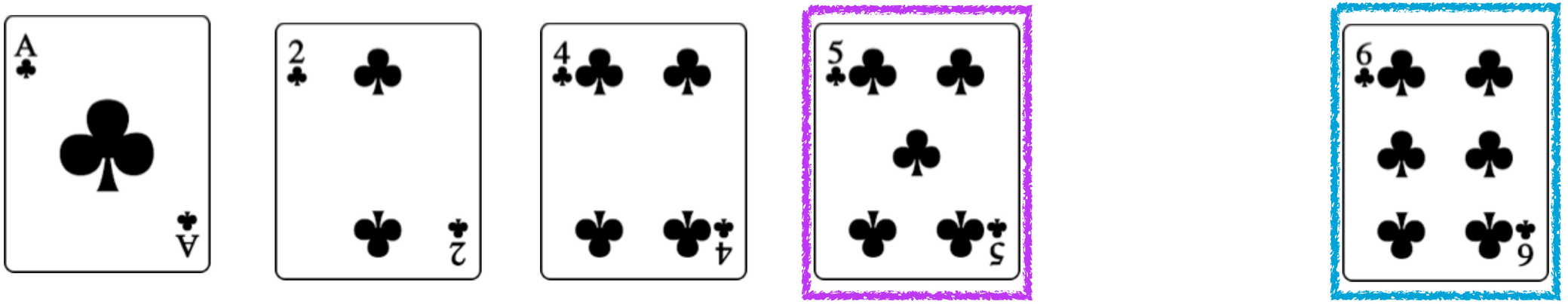
key



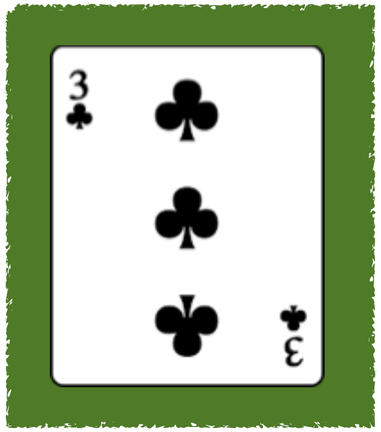
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 i=4 5 j=6



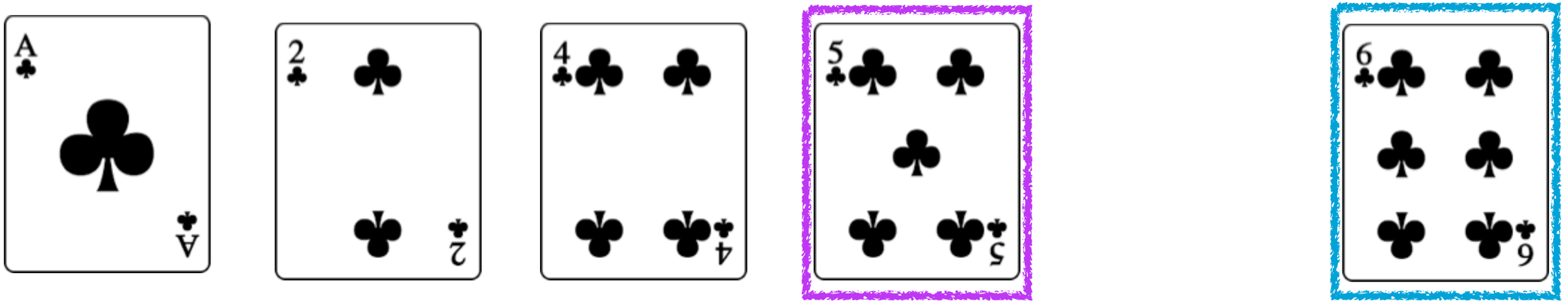
key



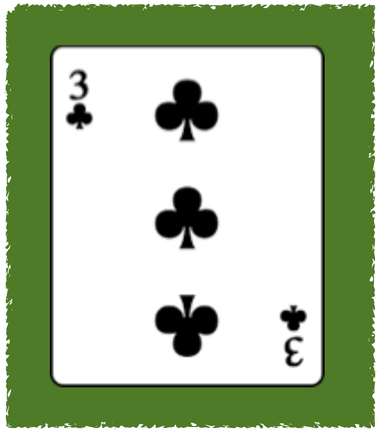
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 i=4 5 j=6



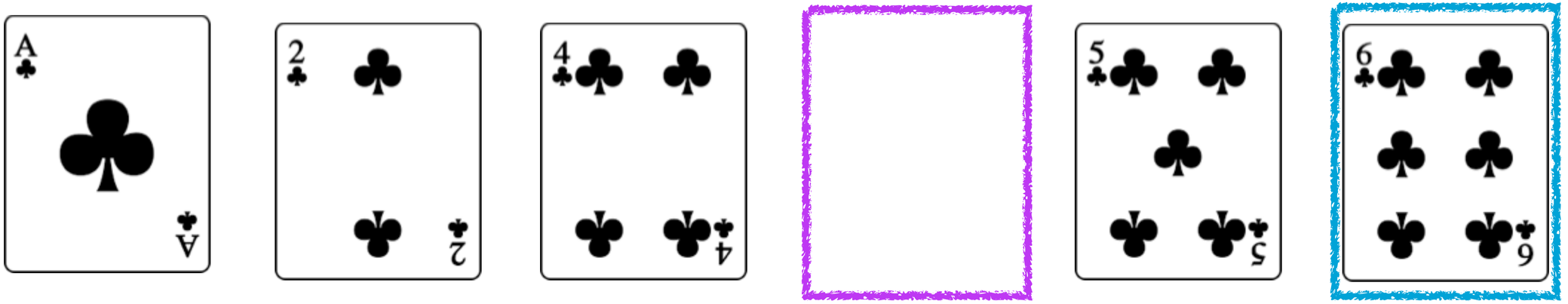
key



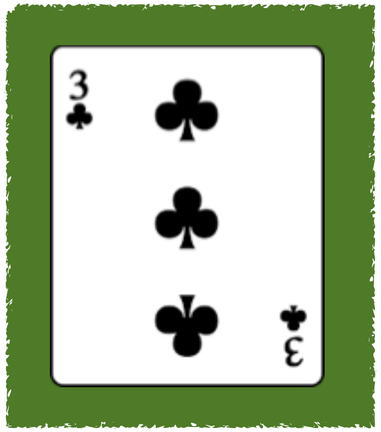
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 3 i=4 5 j=6



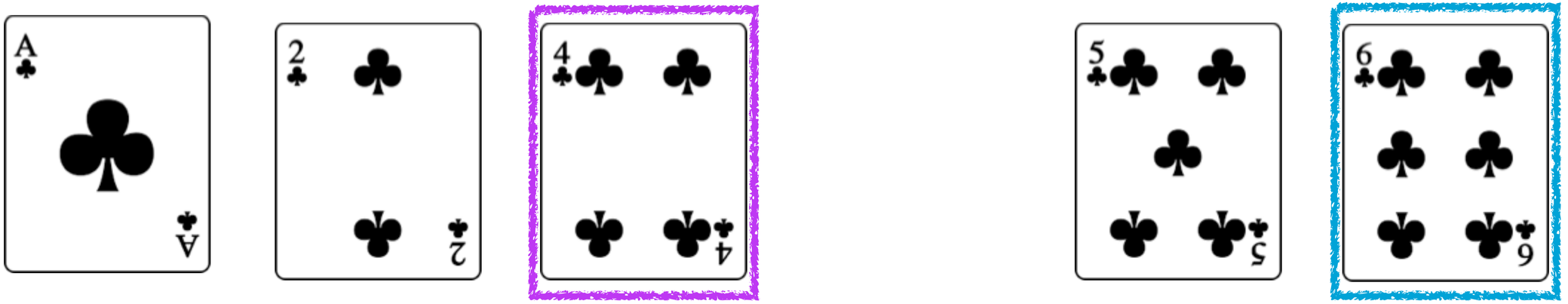
key



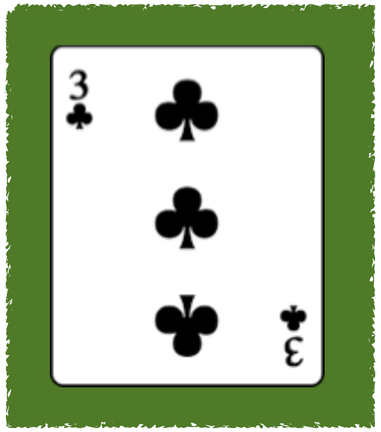
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
    →  do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 5 j=6



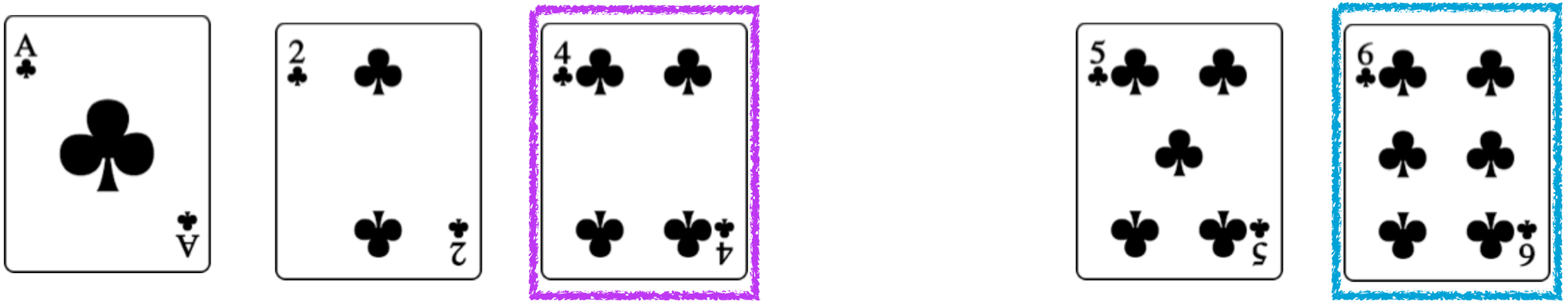
key



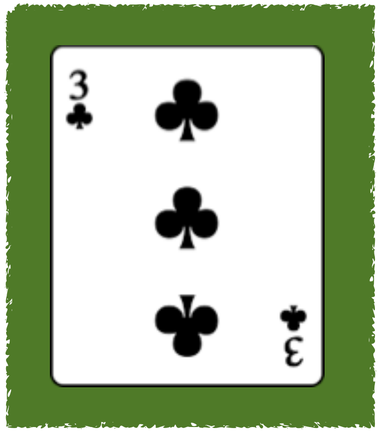
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 5 j=6



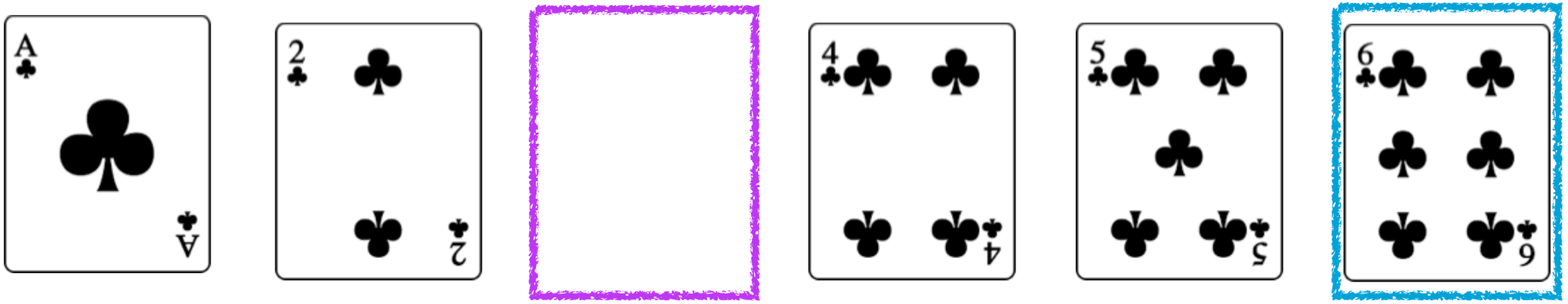
key



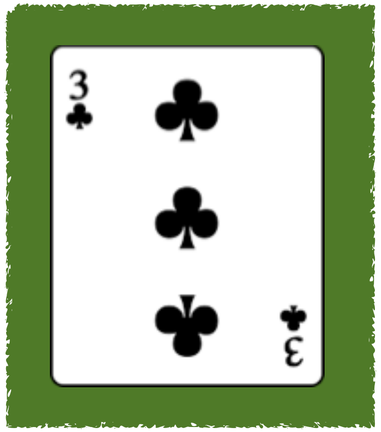
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 2 i=3 4 5 j=6



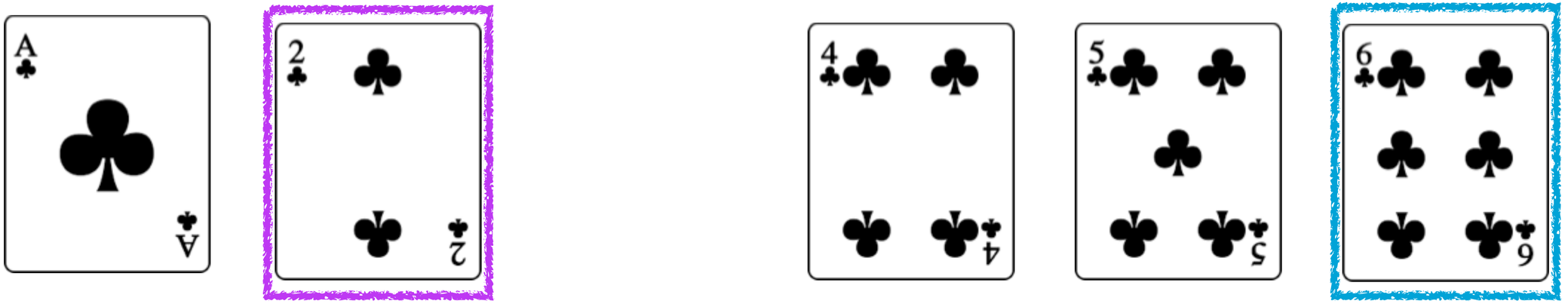
key



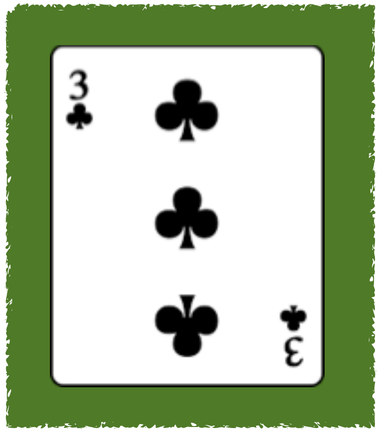
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
    →  do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i= 2 3 4 5 j= 6



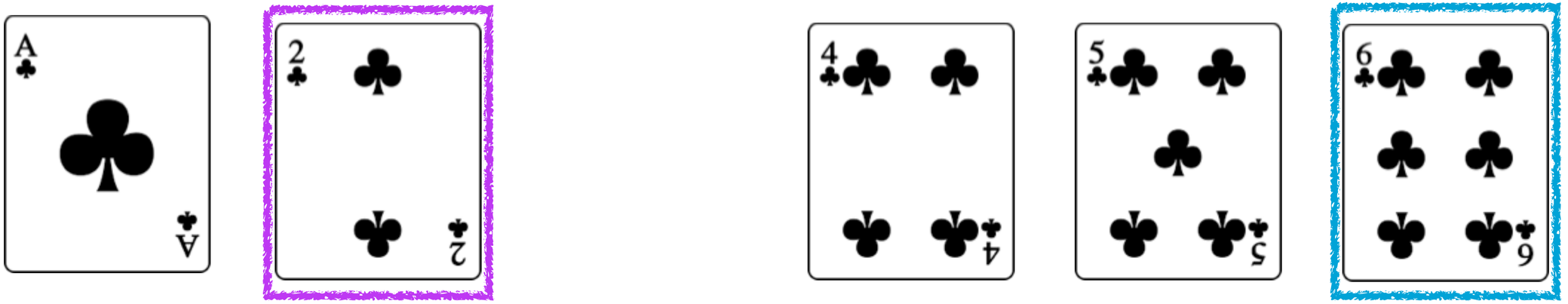
key



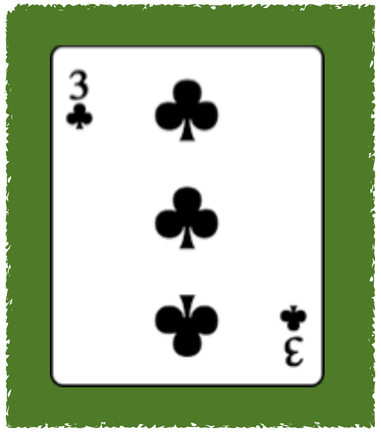
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
           → i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 i= 2 3 4 5 j= 6



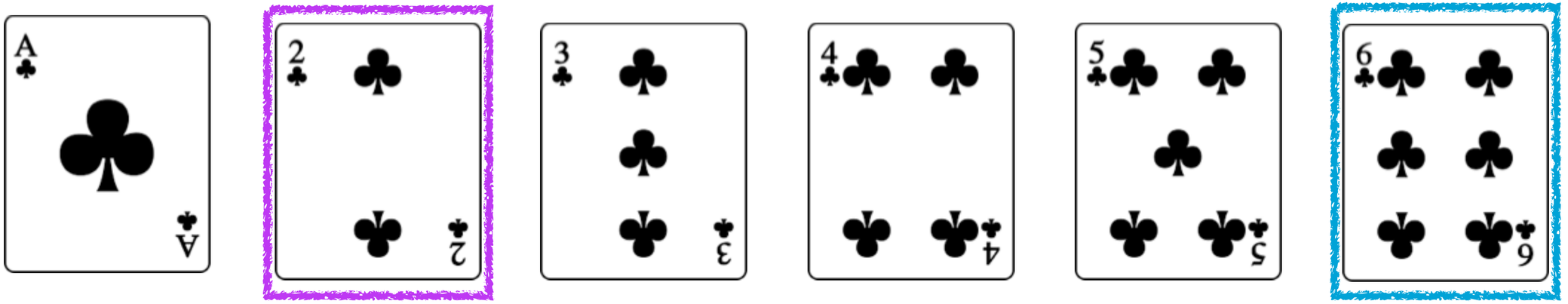
key



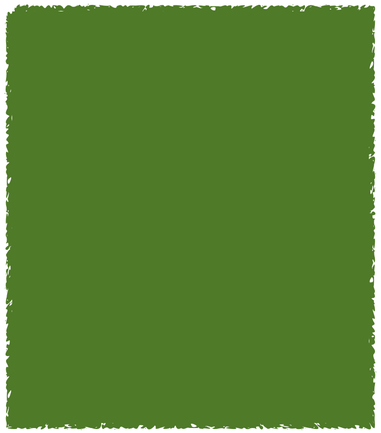
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

insertion sort

1 | i= 2 | 3 | 4 | 5 | j= 6

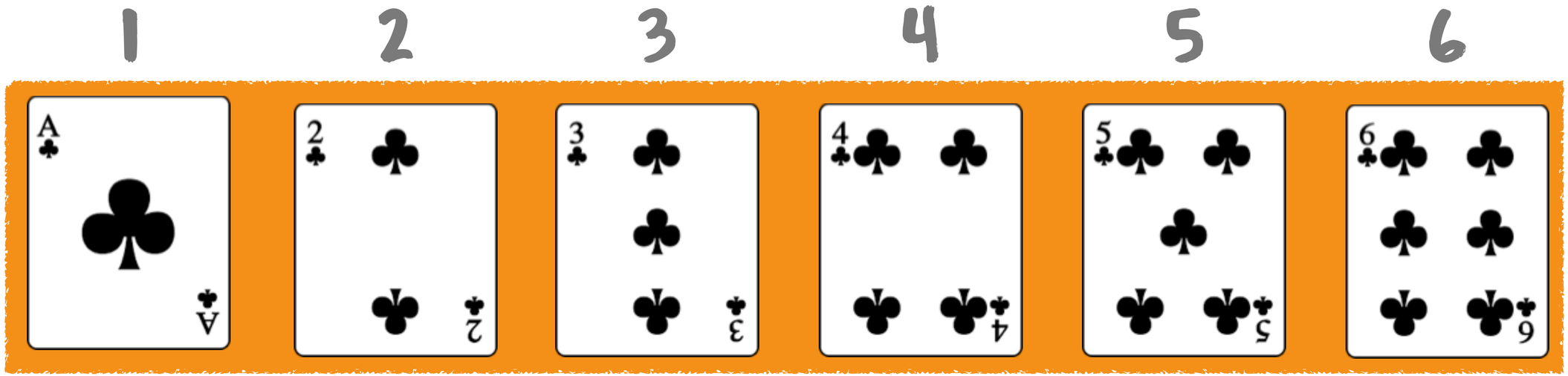


key



```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
          do A[i + 1] ← A[i]
             i ← i − 1
       A[i + 1] ← key
```

insertion sort




```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

correctness

an algorithm is correct
if for any input, it
terminates with the
correct output

loop invariant

a boolean condition that must remain true throughout the loop execution

used to prove that an algorithm gives the correct answer

loop invariant

proving a loop invariant is similar to a mathematical proof by induction

mathematical induction

1. prove base case (true for n = 0 or n = 1)
2. prove inductive step (true for n ⇒ true for n+1)

loop invariant

1. prove invariant holds before the loop starts
2. prove invariant holds from iteration to iteration

loop invariant

in mathematical induction, the
inductive step is used infinitively

for algorithms, we have to show that the
loop terminates and that the invariant is
still true after the loop

loop invariant

1. **initialization**
   prove the invariant is true before the first iteration

2. **maintenance**
   prove the invariant is true before some iteration
   ⇒ invariant true before the next iteration

3. **termination**
   when loop terminates, the invariant gives us a useful property for showing that the algorithm is correct

example

```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

invariant

at the start of each iteration, the for loop consists of the elements originally in $A[1..j-1]$ but in sorted order

initialization: before the first iteration, $j = 2$ so the $A[1..j-1]$ subarray is simply element $A[1]$, which is trivially sorted

maintenance: at each iteration, we shuffle elements of subarray $A[1..j-1]$ to the right until proper position for $A[j]$ is found, where it is inserted; so, at the start of next iteration, the new augmented $A[1..j-1]$ is also sorted

termination: the loop stops when $j = n + 1$, so we then have $A[1..j-1] = A[1..n]$, the whole array, which we know to be sorted thanks to the maintenance property we just proved

the same problem

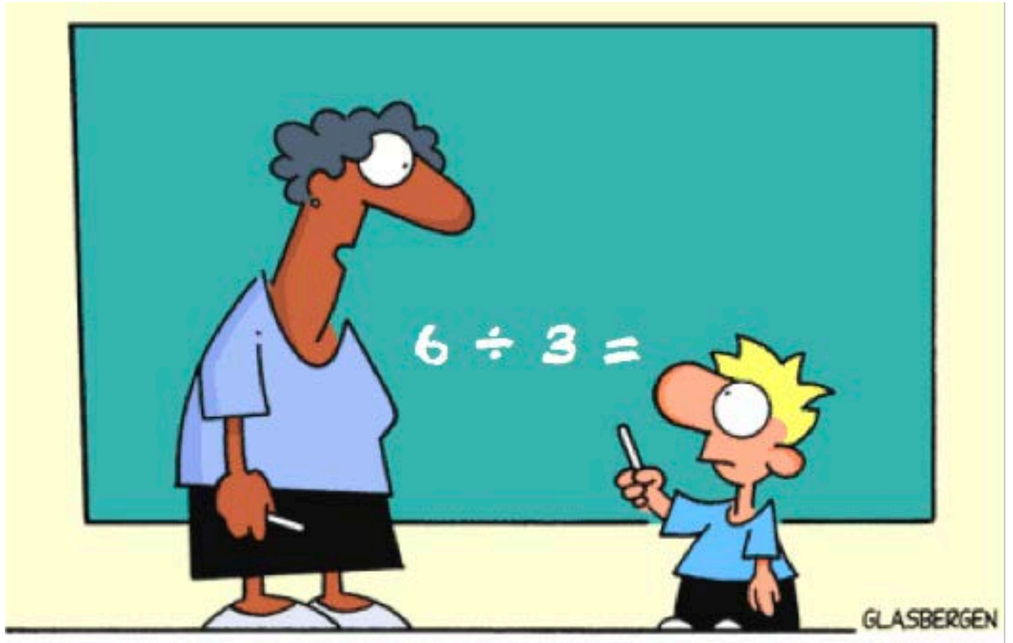
but different algorithms...

..may lead to different performance

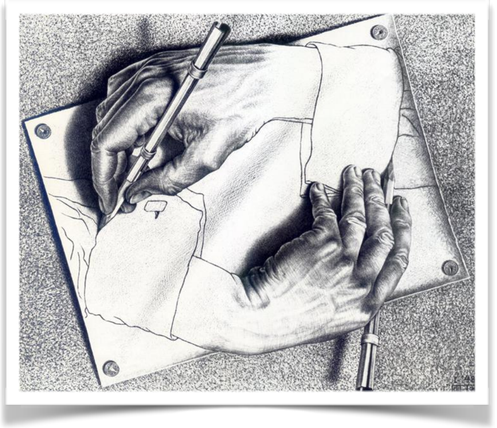
divide & conquer

a classical way to solve complex problems

break the initial problem into several subproblems that are easier to solve than the original problem

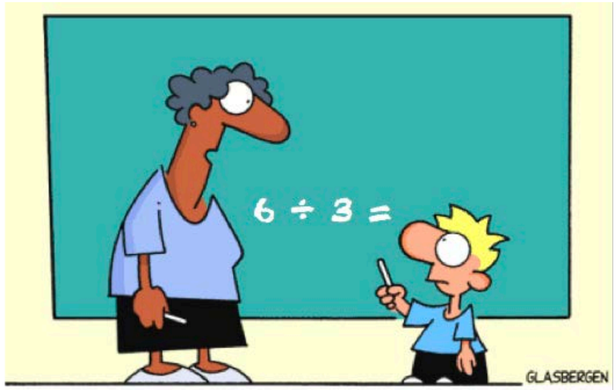


divide & conquer



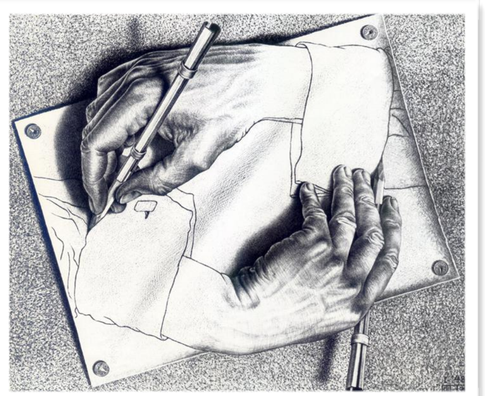
recursion as a special case

- **divide** the initial problem into several subproblems that are smaller instances of the original problem
- **conquer** by computing solutions to those subproblems recursively
- **combine** the smaller solutions into a solution to the initial problem



No, we're just learning how to divide. When you get to business school, you'll learn how to divide and conquer

factorial as example

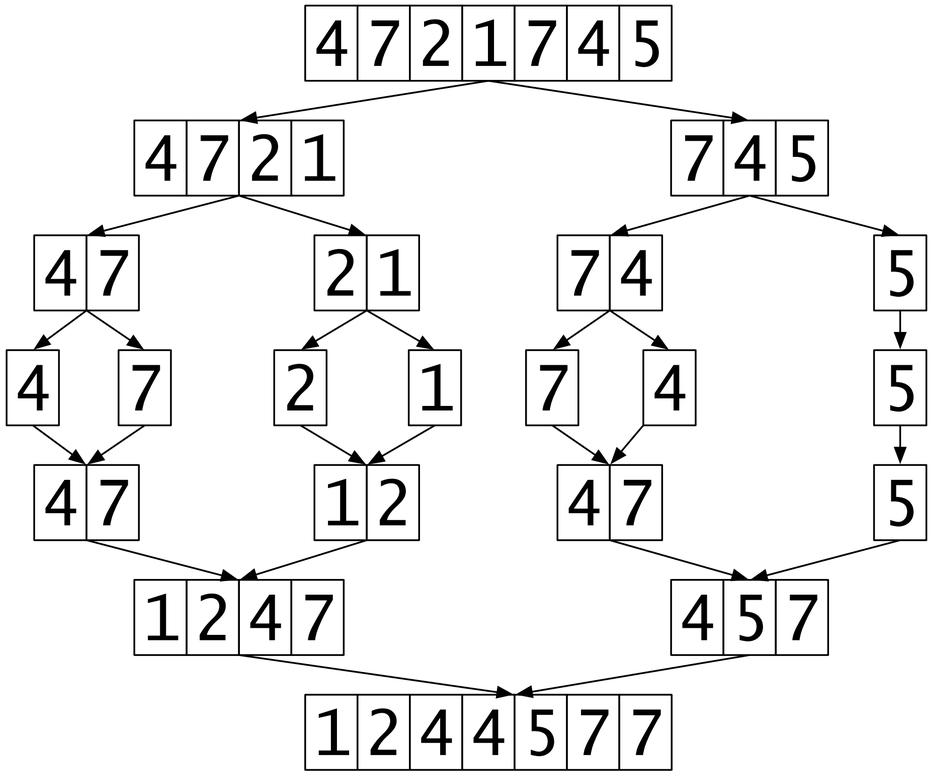


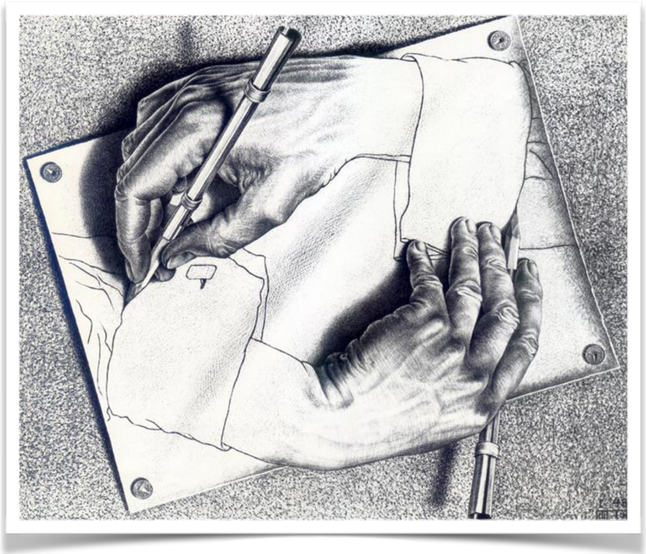
```
def factorial(n: Int) : Int = {
  if (n == 0 || n == 1) {
    1
  }
  else {
    n * factorial(n-1)
  }
}
```

initial call →	f(4) =
1st recursive call →	= 4 * f(3)
2nd recursive call →	= 4 * 3 * f(2)
3rd recursive call →	= 4 * 3 * 2 * f(1)
3rd recursive call returns →	= 4 * 3 * 2 * 1
2nd recursive call returns →	= 4 * 3 * 2
1st recursive call returns →	= 4 * 6
initial call returns →	= 24

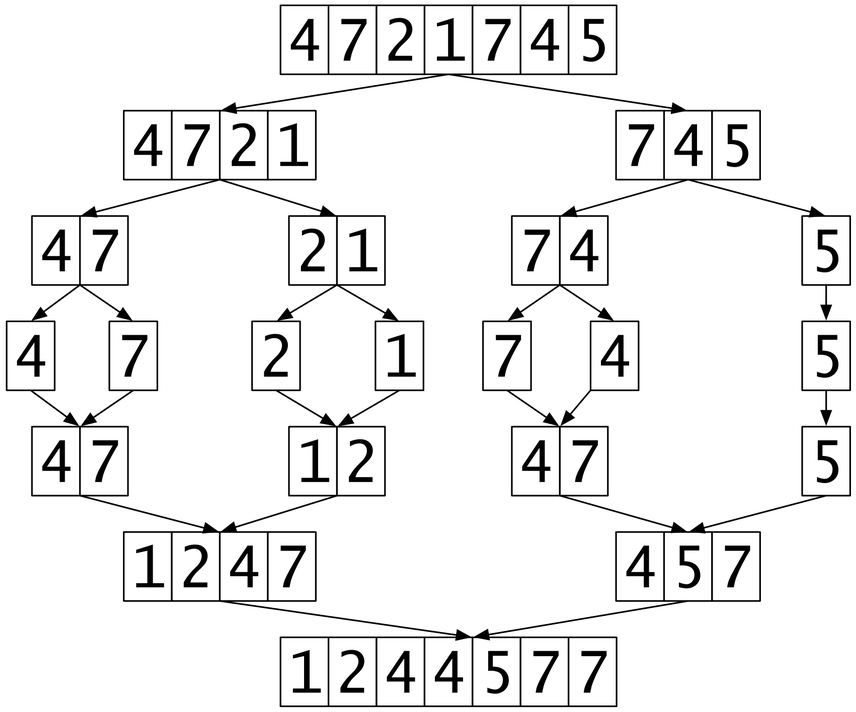
merge sort

divide & conquer





merge sort



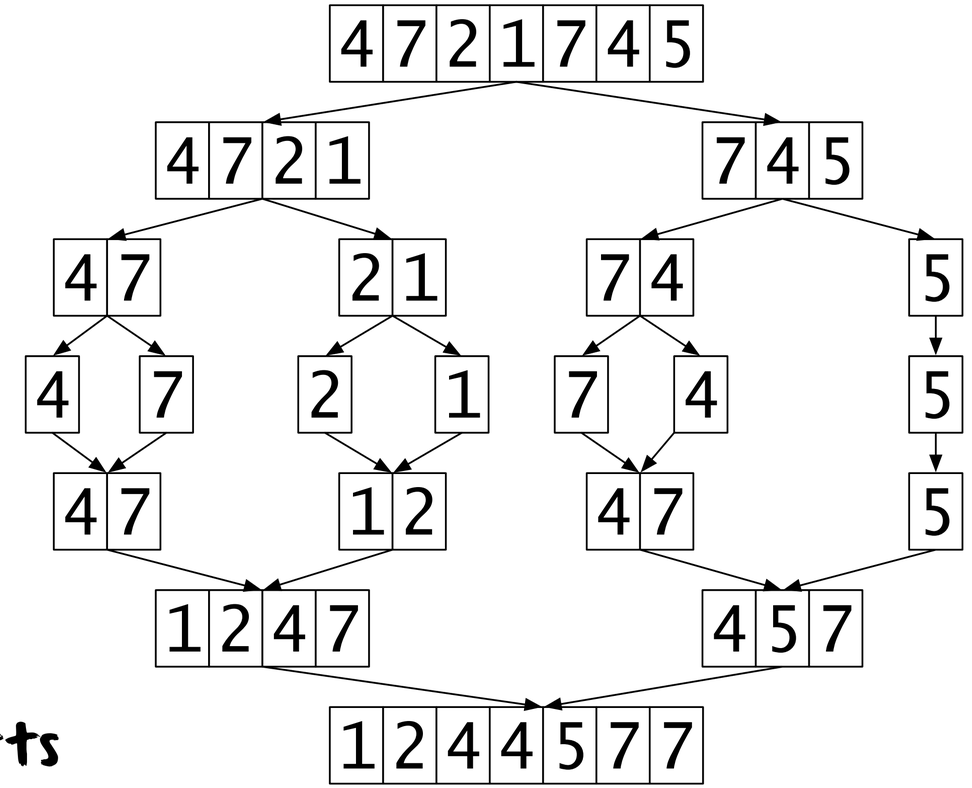
divide: break the sequence of n numbers into pairs

conquer: sort the subsequences recursively using merge sort

combine: merge the sorted subsequences to produce the final sorted array

merge sort

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```



- function MERGE-SORT(A, p, r) sorts array A between indices p and r
- initially, $p = 1$ and $r = n$

merge sort

function MERGE(A, p, q, r)

assumes:

- $1 \le p \le q < r \le n$
- subarrays $A[p..q]$ and $A[q+1..r]$ are sorted

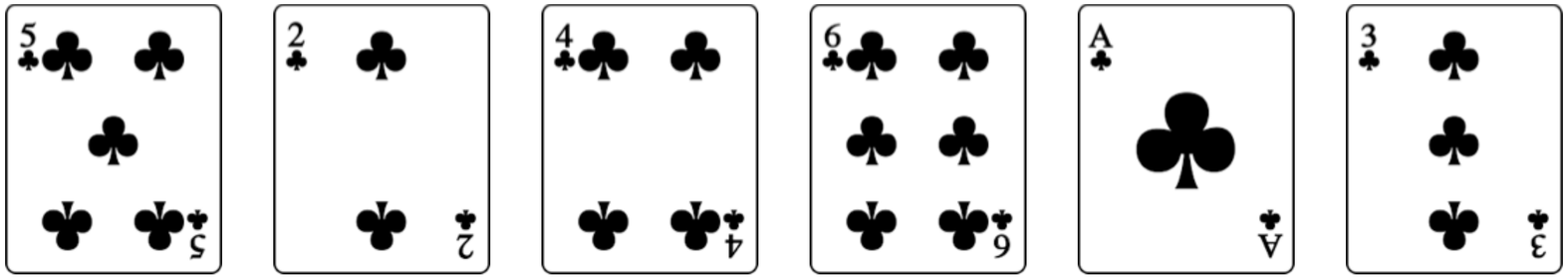
an example:

❶	1 7 8 9	2 4 6	
❷	7 8 9	2 4 6	1
❸	7 8 9	4 6	1 2
❹	7 8 9	6	1 2 4
❺	7 8 9		1 2 4 6
❻			1 2 4 6 7 8 9

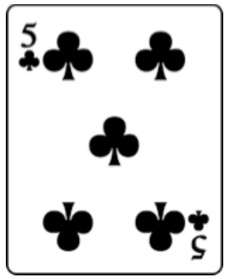
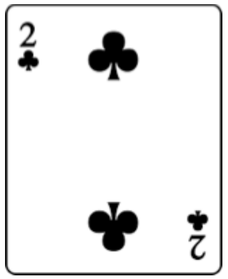
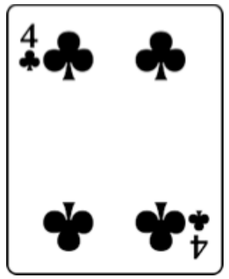
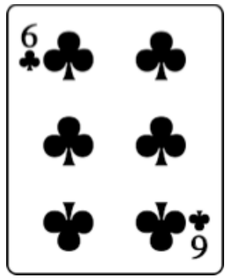
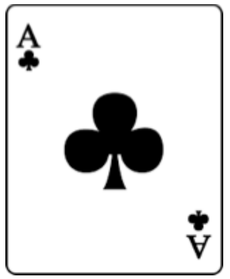
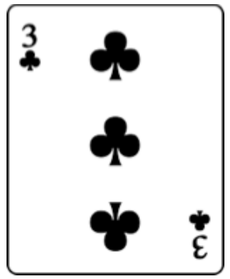
MERGE(A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$
for $i \leftarrow 1$ **to** n_1
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
    do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r
    do if $L[i] \le R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$

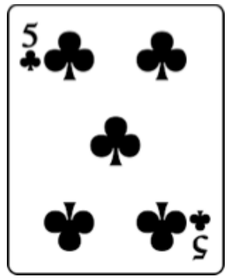
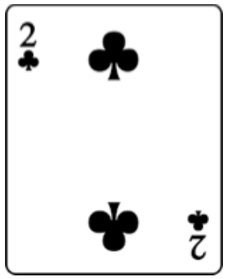
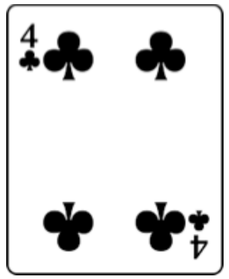
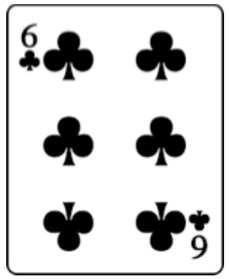
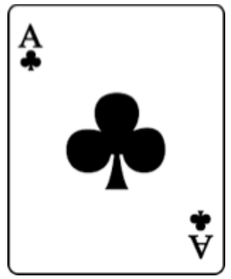
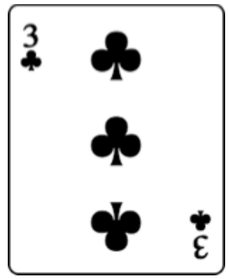
merge sort



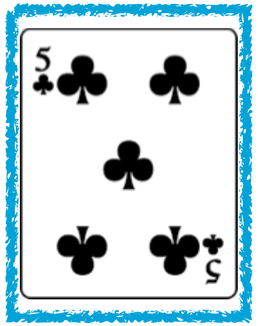
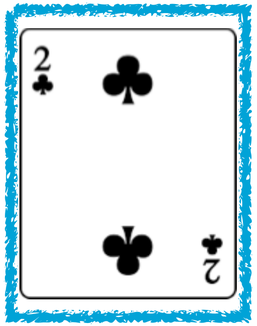
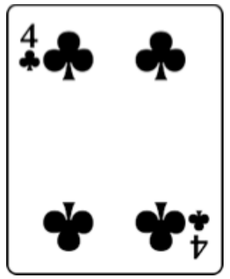
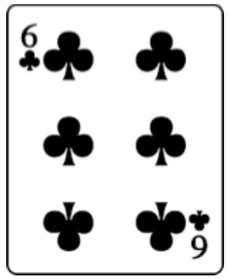
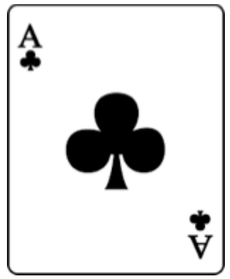
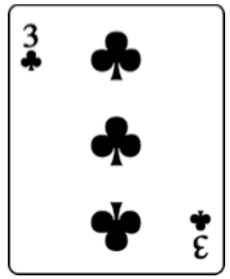
merge sort

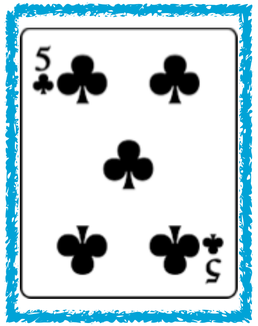
merge sort

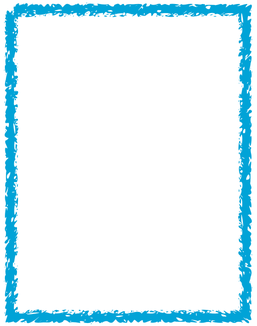







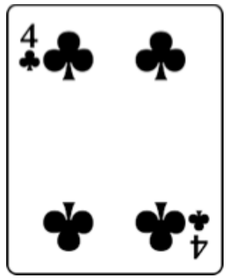
merge sort

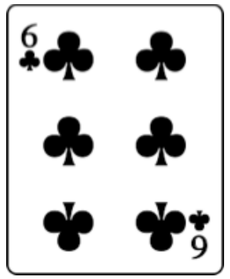







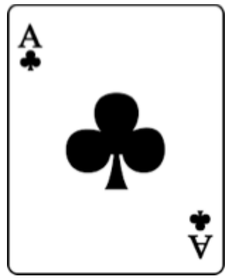
merge sort

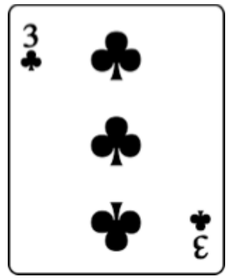


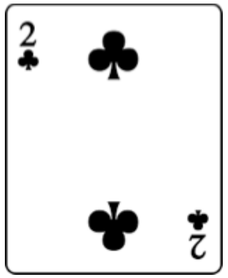




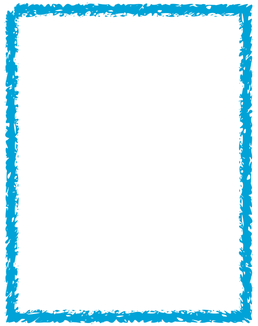


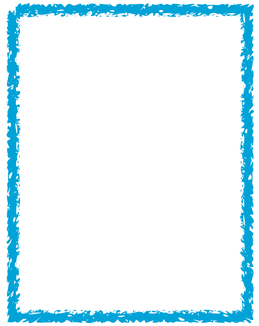


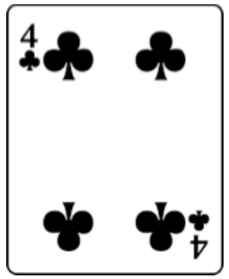


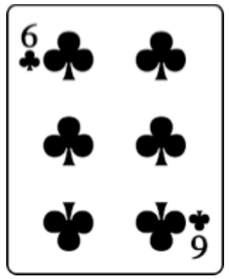


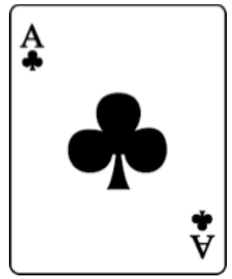
merge sort

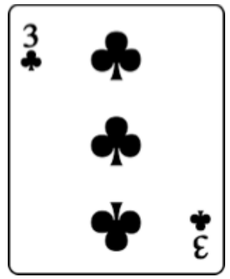


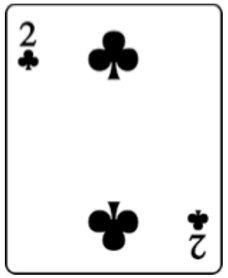


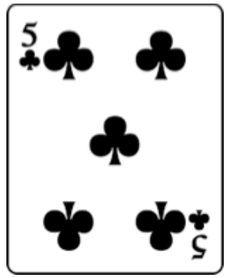




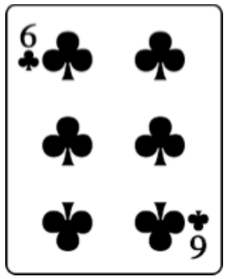


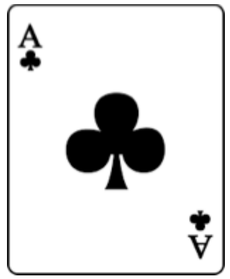


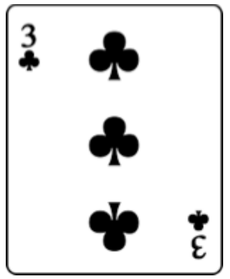


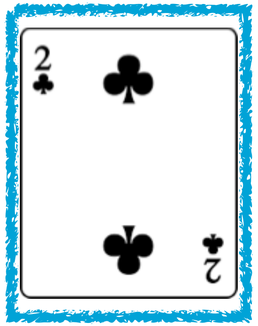


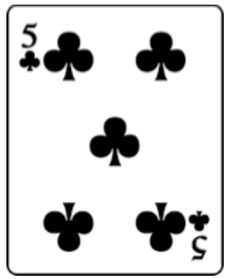
merge sort

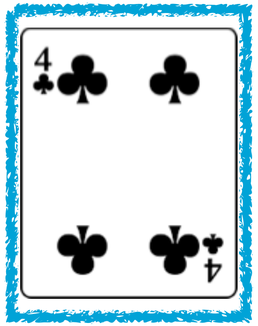




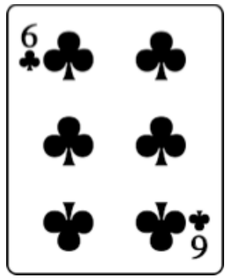


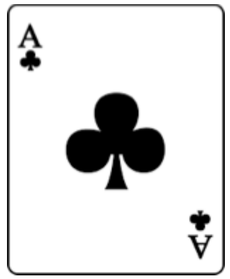


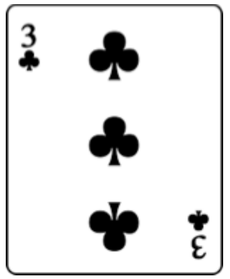


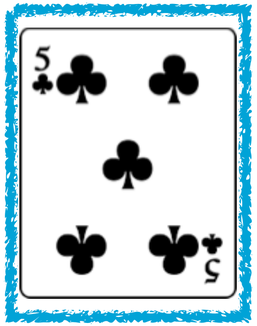


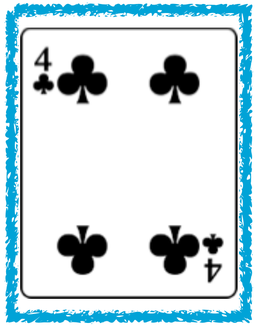
merge sort

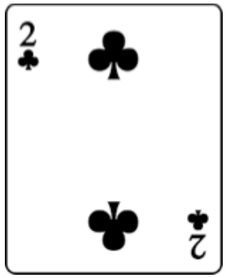




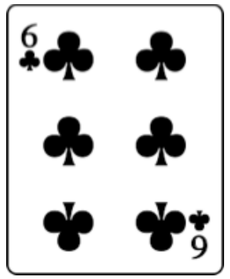


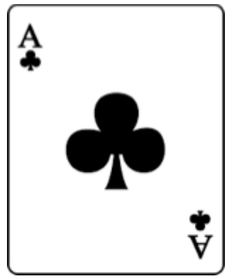


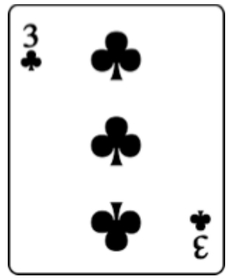


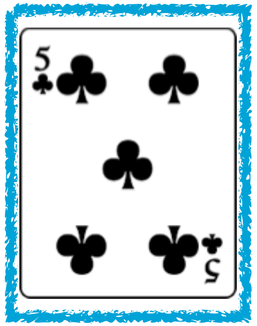


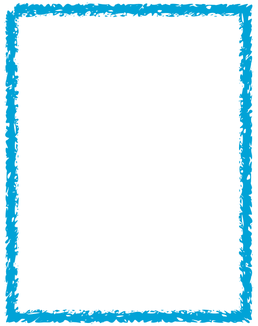
merge sort

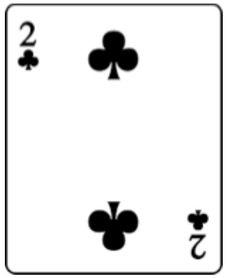


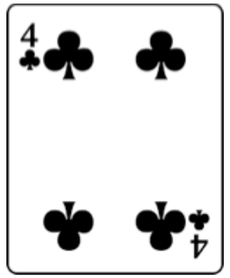




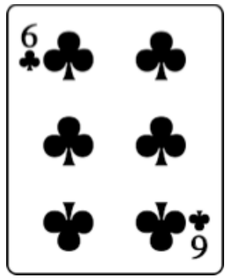


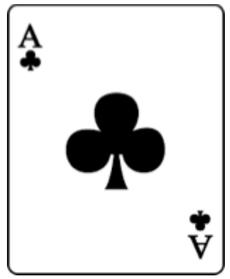


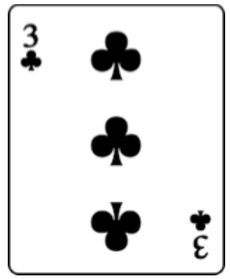


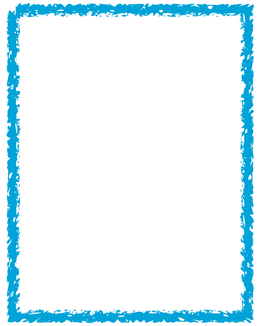


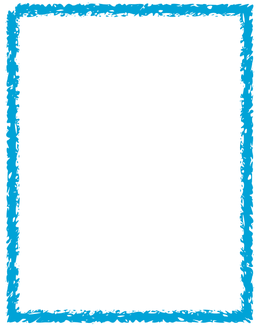
merge sort

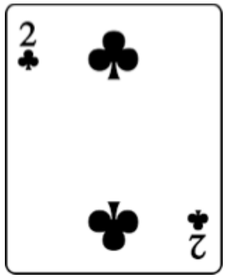


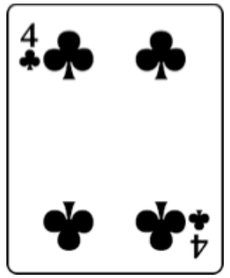


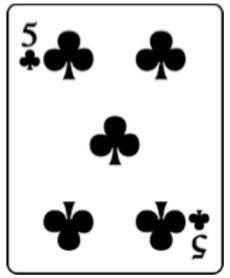




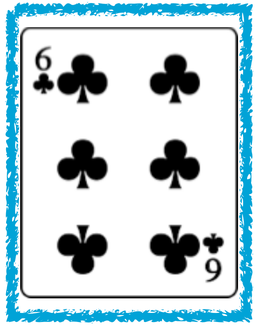


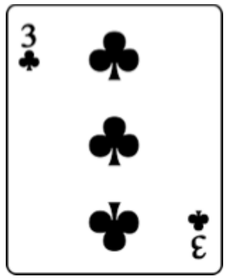


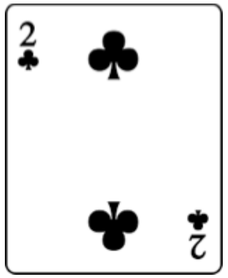


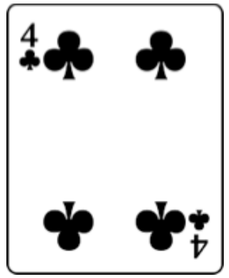


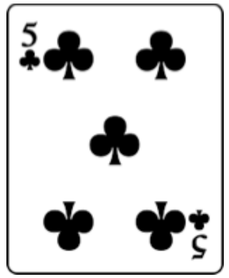
merge sort



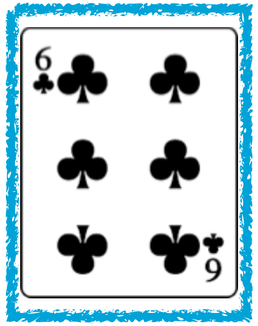


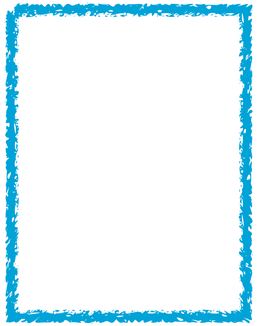


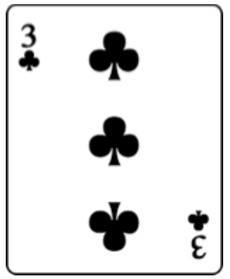


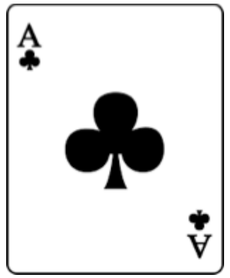


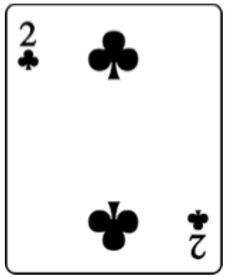
merge sort

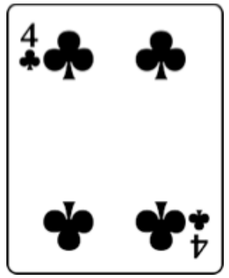


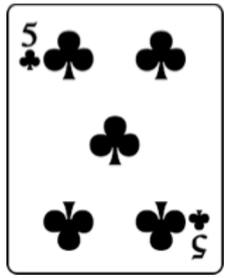




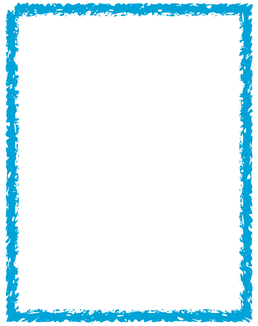


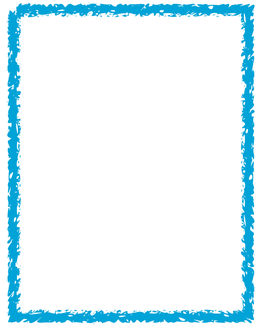


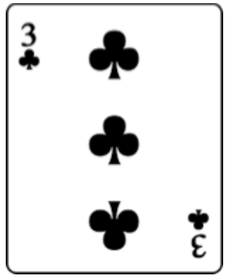


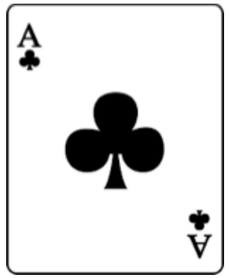


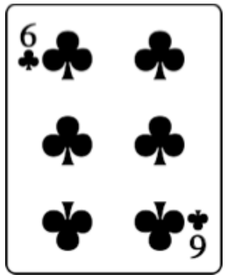
merge sort

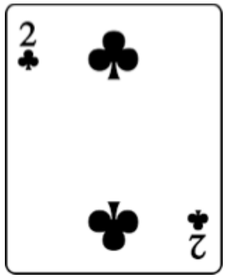


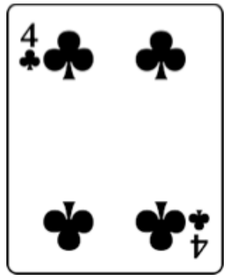


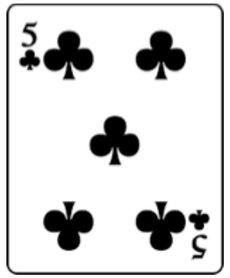




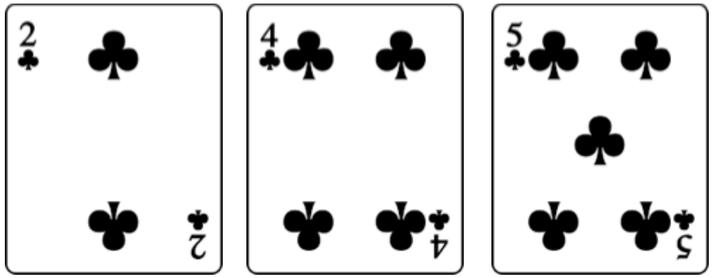


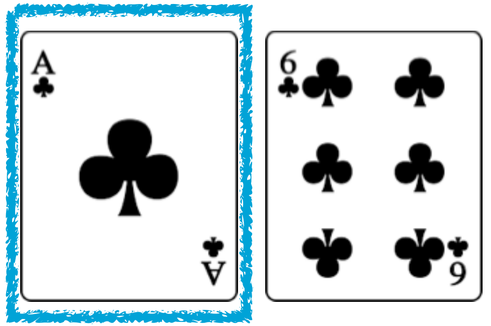


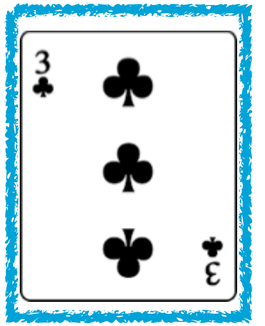




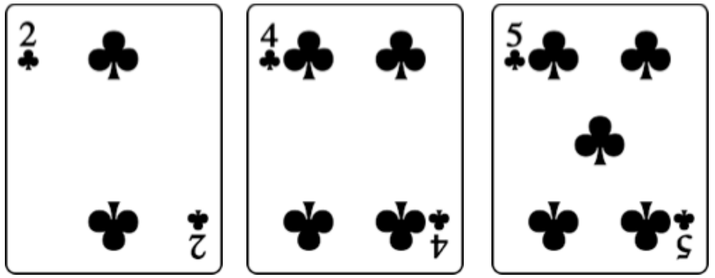
merge sort

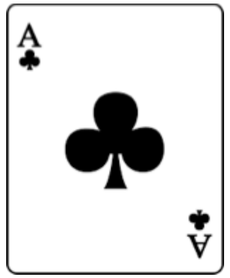


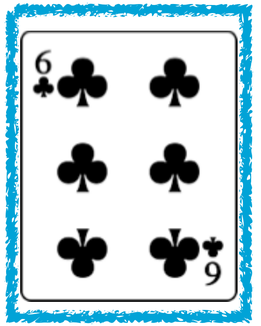


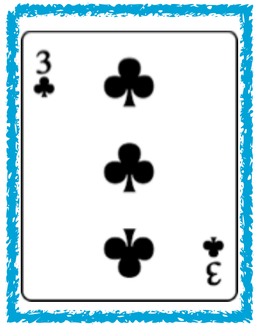


merge sort

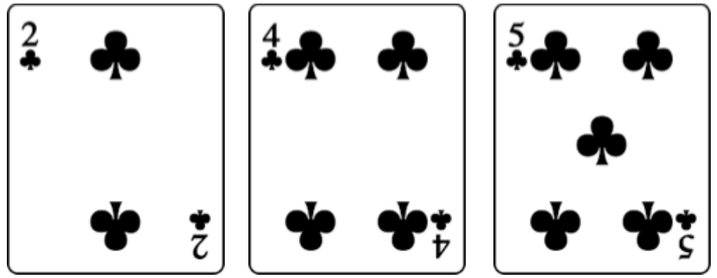


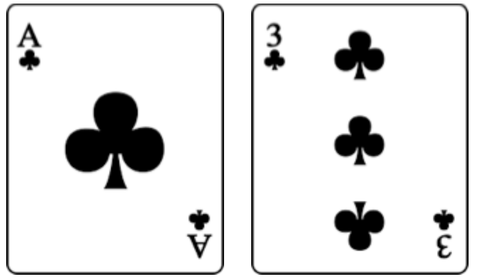


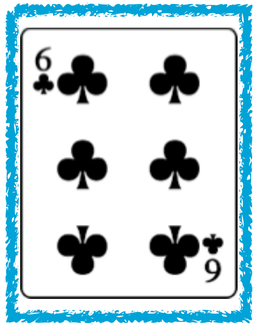


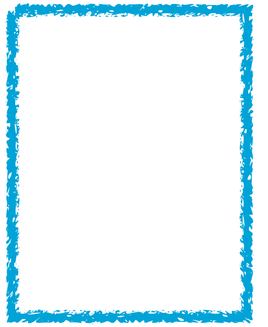


merge sort

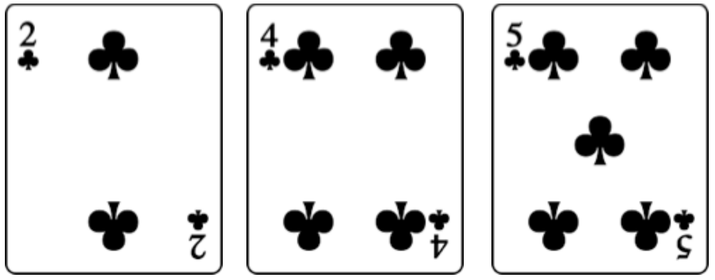


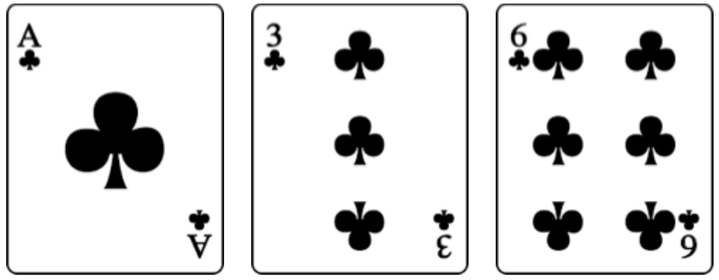




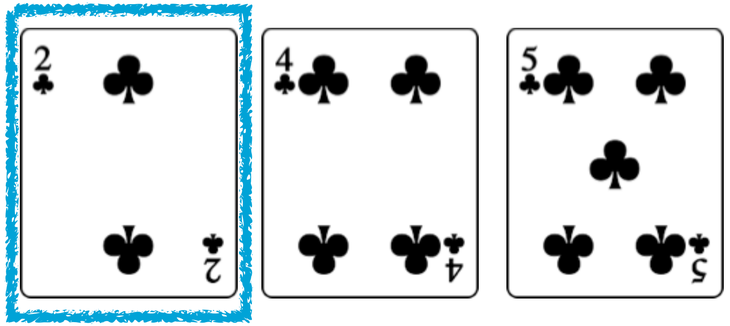


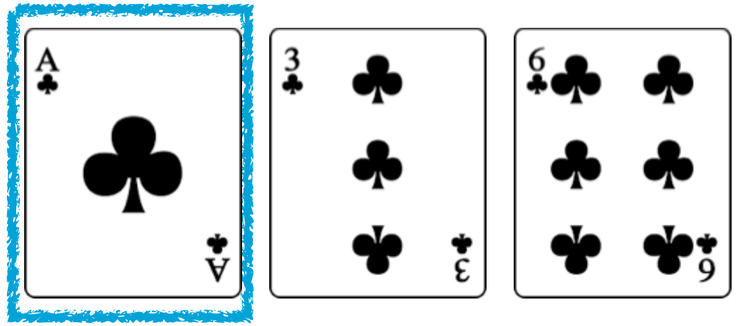
merge sort



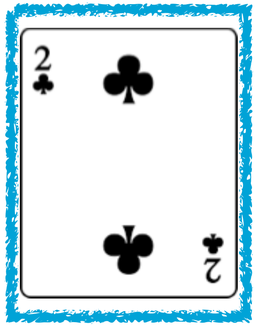


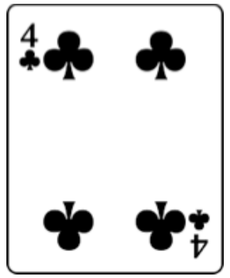
merge sort

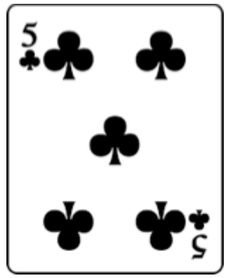


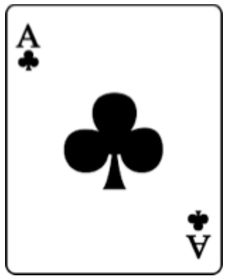


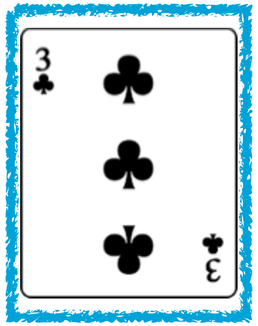
merge sort

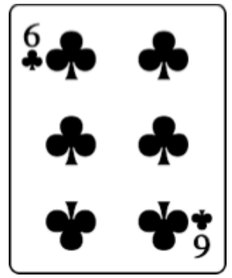




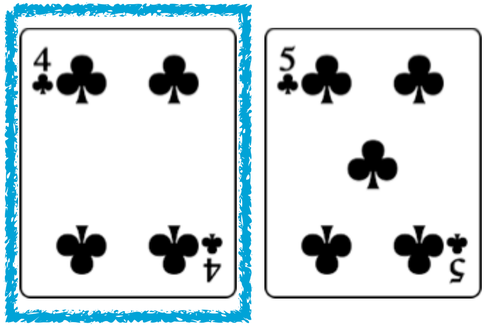


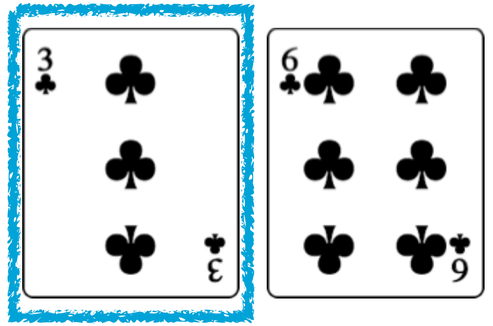


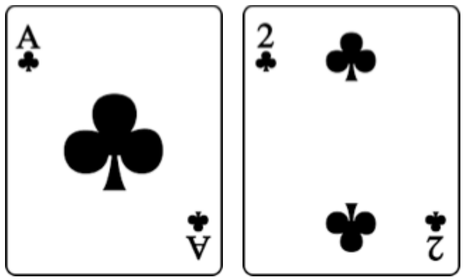




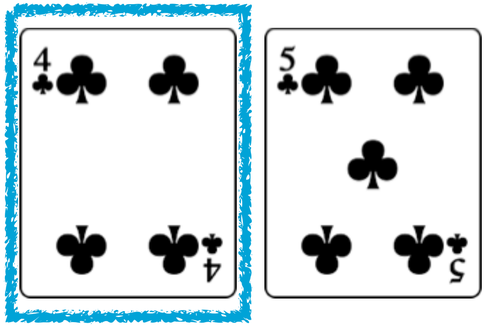
merge sort

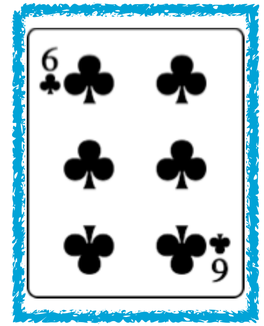


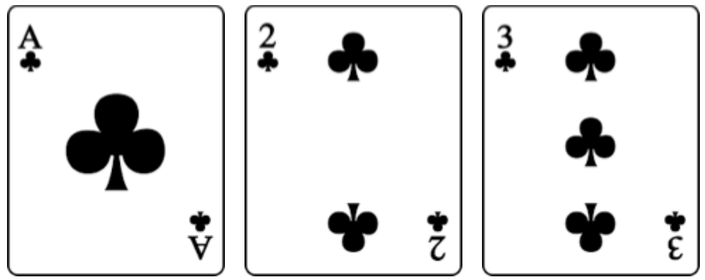




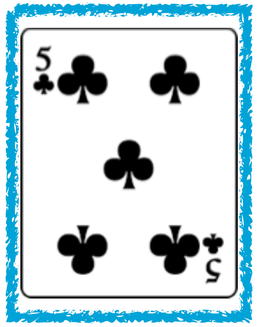
merge sort

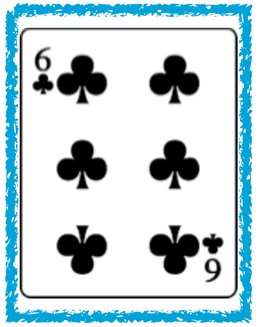


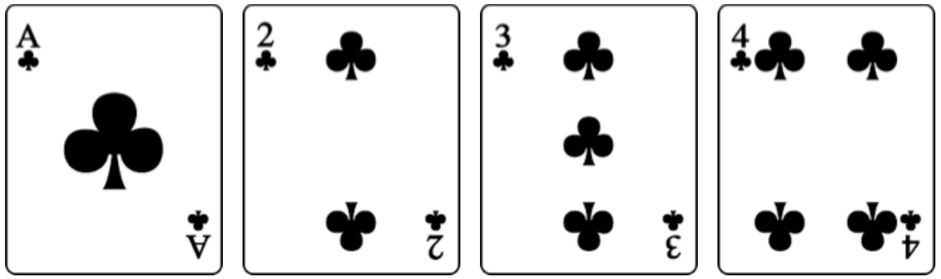




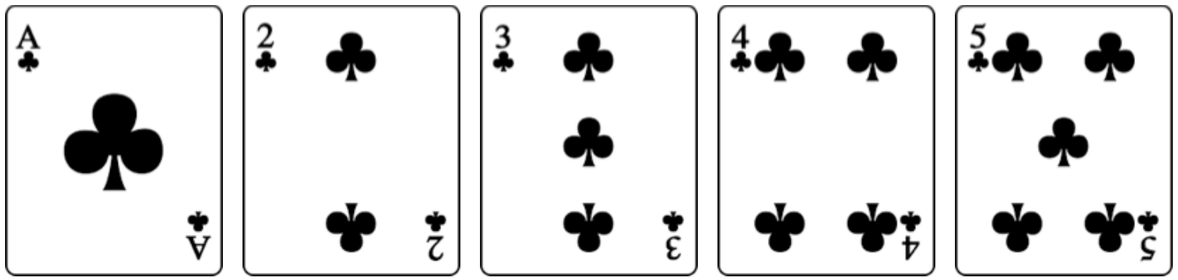
merge sort



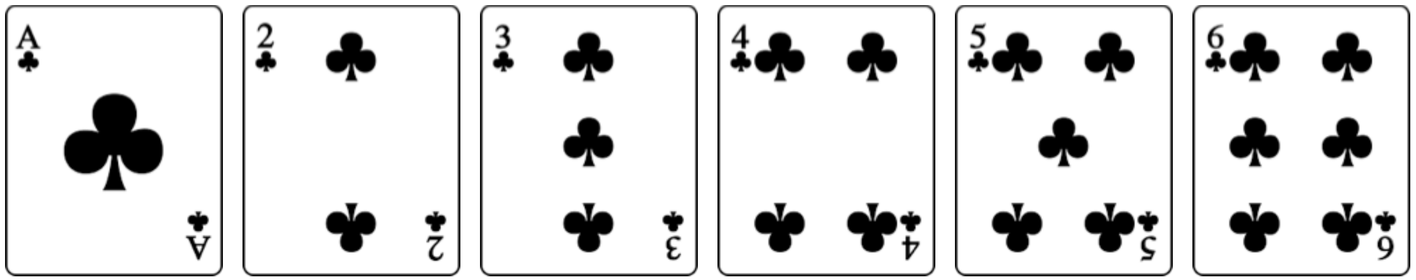




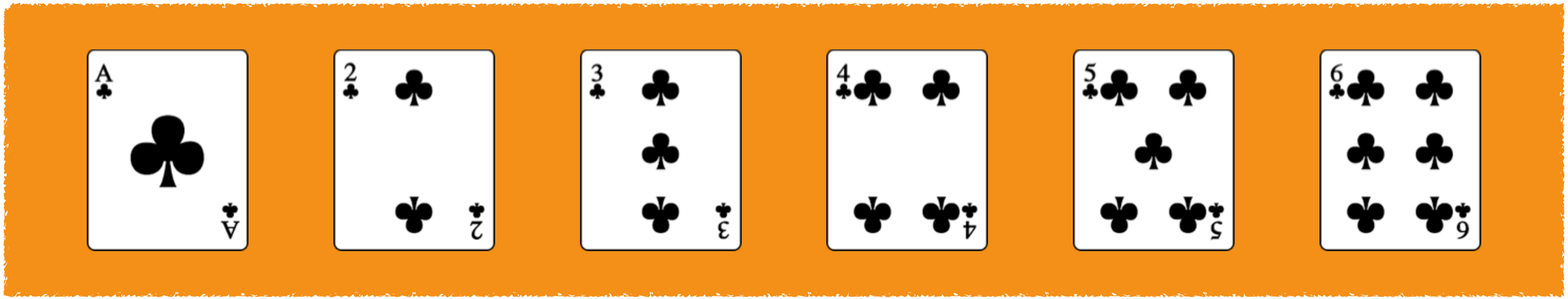
merge sort



merge sort



merge sort



algorithm analysis

aim of analysis

algorithm analysis aims at predicting the resources needed by an algorithm to produce its output, as a measure of its efficiency

typical resources

computational time

memory footprint

network bandwidth

example

hereafter, we mainly focus on
computational time as a measure
of the algorithm efficiency

usually, the computational time
depends on the size of the data
taken as input by the algorithm

example

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

let $T(n)$ be the running time of the insertion sort, so:

$$T(n) = \sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

example

let $T(n)$ be the running time of the insertion sort, so:

$$T(n) = \sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \; .$$

best case scenario

best case: the array is already sorted

* $A[1..n] \le key$ at start of each while loop
* all t_j are equal to 1

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; .
\end{aligned}
$$

best case form: $an + b$ (linear function of n)

this is known as the **lower bound** of the algorithm

worst case scenario

worst case: the array is sorted in reverse order

* $A[1..n] > key$ throughout each while loop
* key compared with all numbers left to the j-th position, i.e., to $j-1$ numbers

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

worst case form: $an^2 + bn + c$ (quadratic function of n)

this is known as the **upper bound** of the algorithm

best case vs. worst case

we are usually interested in the worse case,
which represents the maximum growth rate,
also known as order of growth

we usually keep only the higher power of n
and say that the insertion sort algorithm has
a worse-case running time of $O(n^2)$
(it reads "order of n square")

costs

vs.

insertion sort I_A

$c_1 n^2$ steps to sort n numbers, with c_1 a constant independent of n

merge sort M_A

$c_2 n \log_2 n$ steps to sort n numbers, with c_2 a constant independent of n

VS.

c_1 and c_2 depend on how the algorithm was actually implemented and compiled

let's assume that $c_1 = c_2 = 2$
(good programmer and compiler)

let's assume I_A and M_A
both run on a computer executing
1 billion (10^9) instructions per second

input data

finally, let's assume I_A and M_A have to

sort 1 million (10^6) numbers

Q: how much time will it take?

results

A: insertion sort I_A

$$\frac{2 \times (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}}$$

$= 2000 \text{ seconds} = 33 \text{ minutes}$

A: merge sort M_A

$$\frac{2 \times 10^6 \log_2 10^6 \text{ instructions}}{10^9 \text{ instructions/second}}$$

$= 0.04 \text{ seconds} = 40 \text{ milliseconds}$

with 10 million numbers, the difference is even bigger

2.3 days for I_A and less than a second for M_A !

let's penalize M_A

insertion sort I_A

let's assume that $c_1 = 2$

(good programmer & compiler)

merge sort M_A

let's assume that $c_2 = 50$

(lousy programmer & compiler)

let's penalize M_A

let's assume I_A runs on a computer executing 1 billion (10^9) instructions per second

let's assume M_A runs a computer executing only 10 million (10^7) instructions per second

task

let's assume I_A and M_A have to

sort 1 million (10^6) numbers

with their respective implementations and
on their respective computers

Q: how much time will it take?

results

A: insertion sort I_A

$$\frac{2 \times (10^6)^2 \textit{ instructions}}{10^9 \textit{ instructions/second}}$$

$= 2000 \textit{ seconds} = 33 \textit{ minutes}$

A: merge sort M_A

$$\frac{50 \times 10^6 \log_2 10^6 \textit{ instructions}}{10^7 \textit{ instructions/second}}$$

$= 100 \textit{ seconds} = 1.6 \text{ minute}$

with 10 million numbers, the difference is even bigger

2.3 days for I_A and just under 20 minutes for M_A !