Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- If the key is smaller, go left.
- If the key is bigger, go right.
- Equal, found.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- If the key is smaller, go left.
- If the key is bigger, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

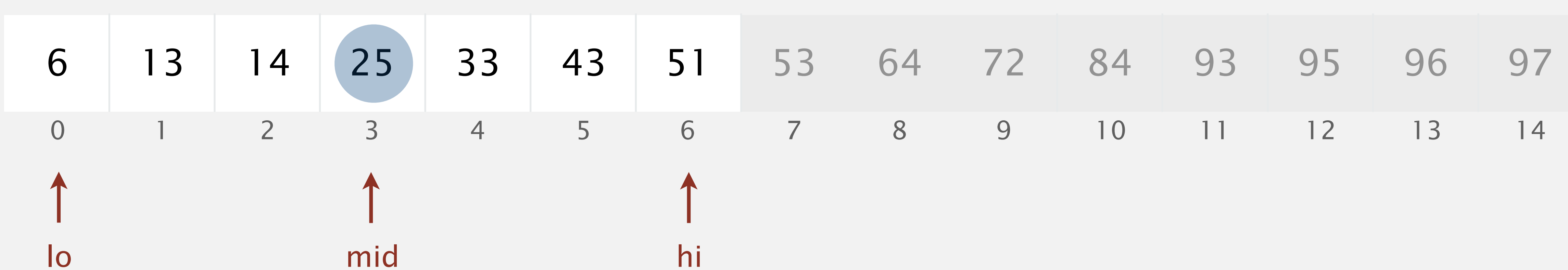
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- If the key is smaller, go left.
- If the key is bigger, go right.
- Equal, found.

successful search for 33



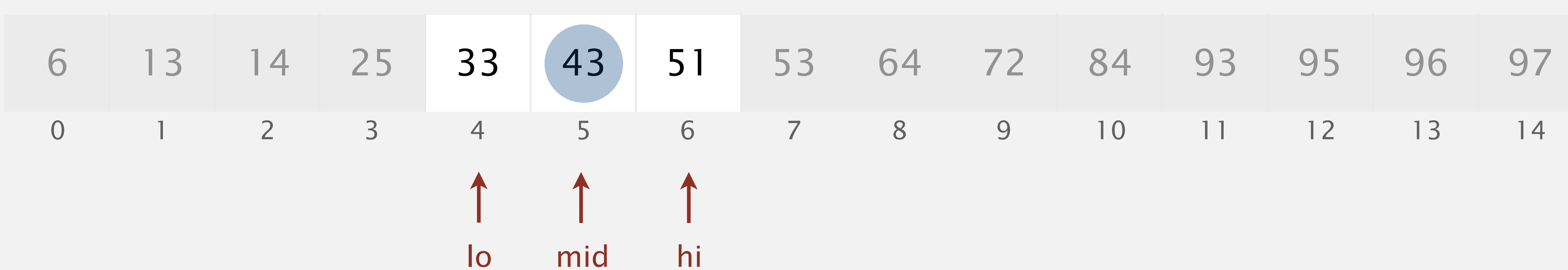
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- If the key is smaller, go left.
- If the key is bigger, go right.
- Equal, found.

successful search for 33



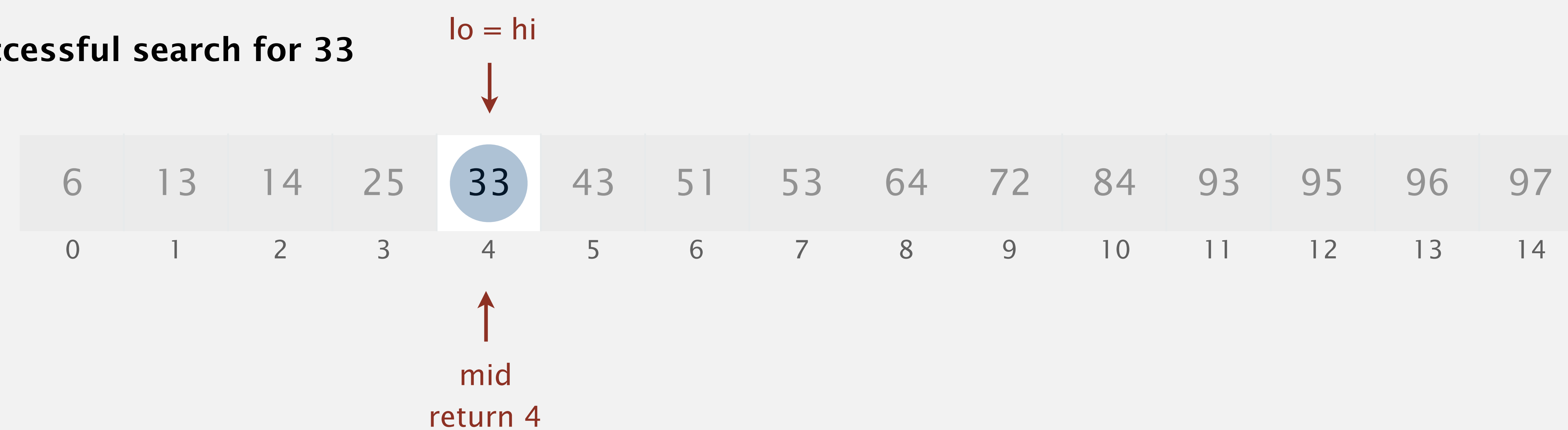
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- If the key is smaller, go left.
- If the key is bigger, go right.
- Equal, found.

successful search for 33



Binary search: Java implementation

Invariant. If `key` appears in array `a[]`, then `a[lo]` ≤ `key` ≤ `a[hi]`.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length - 1;
   while (lo <= hi)
   {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;
}
```

one "3-way compare"

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N.

Def. $T(N)$ = # key compares to binary search a sorted subarray of size $\leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

left or right half (floored division) — $T(N/2)$

the compare — 1

Pf sketch. [assume N is a power of 2]

$$
\begin{aligned}
T(N) &\leq T(N/2) + 1 && \text{[given]} \\
&\leq T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\
&\leq T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\
&\;\vdots \\
&\leq T(N/N) + \underbrace{1 + 1 + \ldots + 1}_{\lg N} && \text{[stop applying, } T(1) = 1 \text{]} \\
&= 1 + \lg N
\end{aligned}
$$

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Types of analyses

Best case. Lower bound on cost.

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for "random" input.
- Provides a way to predict performance.

this course

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. "Expected" cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

Theory of algorithms

Goals.

- Establish "difficulty" of a problem.
- Develop "optimal" algorithms.

Approach.

- Suppress details in analysis: analyze "to within a constant factor."
- Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better.

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10\ N^2$ $5\ N^2 + 22\ N \log N + 3N$ $\vdots$	classify algorithms
Big O	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\ N^2$ $100\ N$ $22\ N \log N + 3\ N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$ $\vdots$	develop lower bounds

Theory of algorithms: example 1

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 1-SUM = "*Is there a* 0 *in the array?* "

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all *N* entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s–.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than "to within a constant factor" to predict performance.

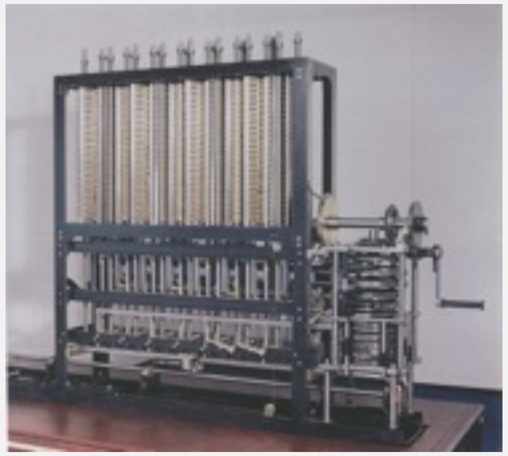
Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}\ N^2$ $10\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\ N^2$ $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

This course. Focus on approximate models: use Tilde-notation

Turning the crank: summary



Empirical analysis.

- Execute program to perform experiments.
- Assume power law.
- Formulate a hypothesis for running time.
- Model enables us to make predictions.

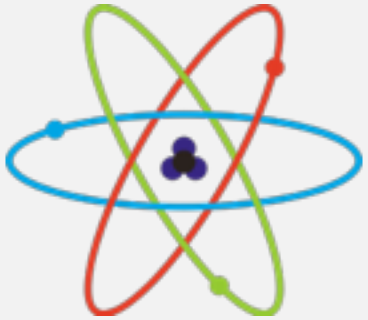
Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

$$\sum_{h=0}^{\lfloor \lg N \rfloor} \lceil N/2^{h+1} \rceil \, h \;\sim\; N$$

Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.

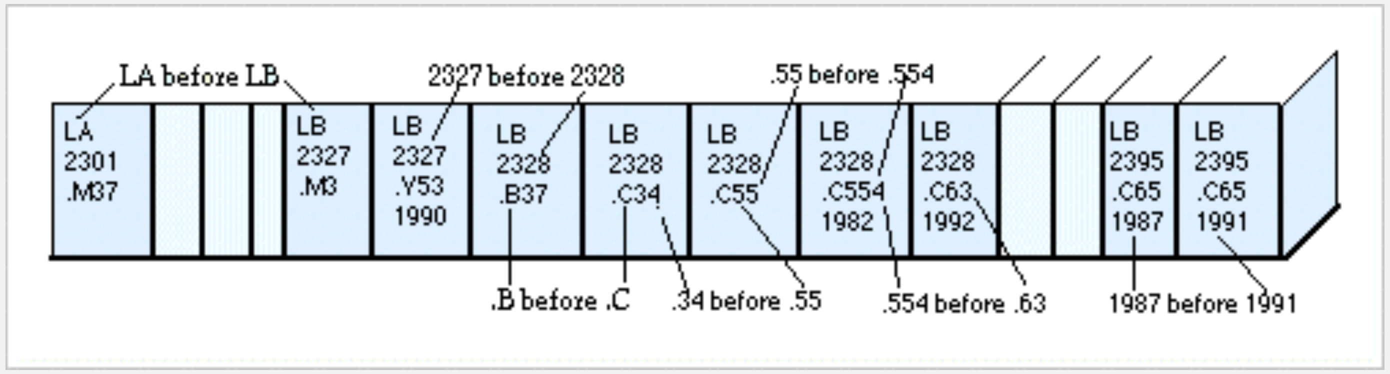


http://algs4.cs.princeton.edu

2.1 Elementary Sorts

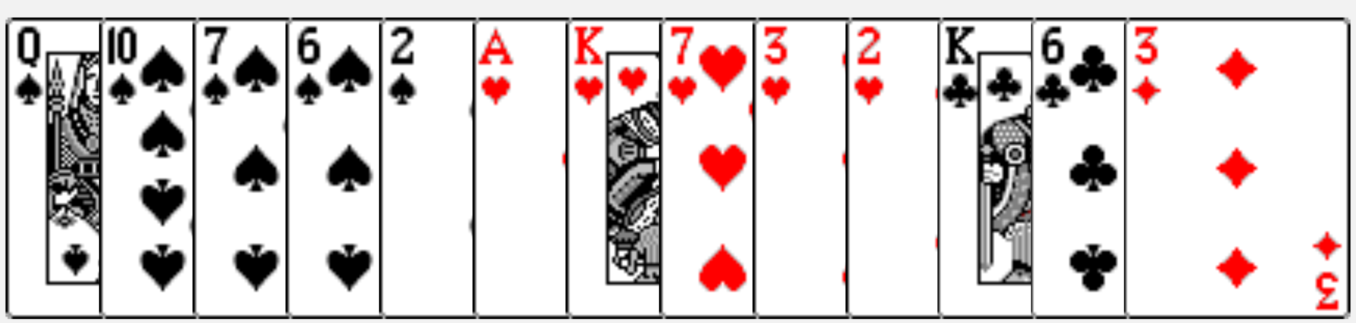
- *rules of the game*
- *selection sort*
- *insertion sort*

Sorting applications

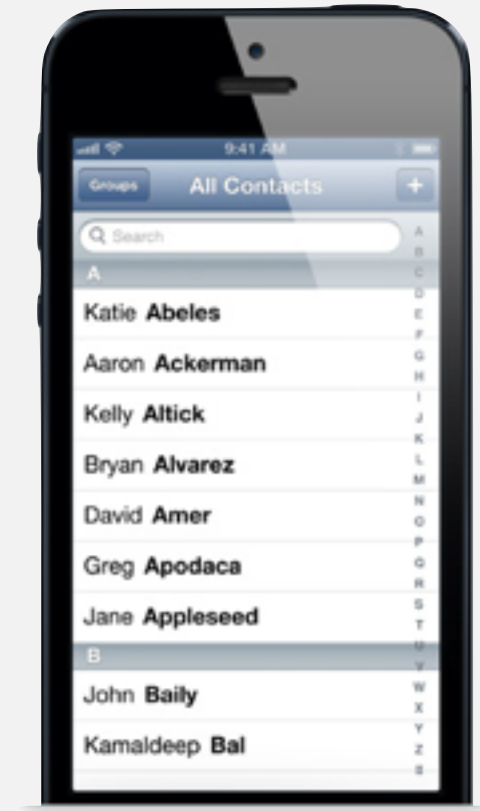


Library of Congress numbers

FedEx packages



playing cards



contacts

Hogwarts houses

Sample sort client 1

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial (stay tuned for an application)

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Sample sort client 2

Goal. Sort any type of data.

Ex 2. Sort strings in alphabetical order.

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAllStrings();
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter < words3.txt
all bad bed bug dad ... yes yet zoo
[suppressing newlines]
```

Sample sort client 3

Goal. Sort any type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;

public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

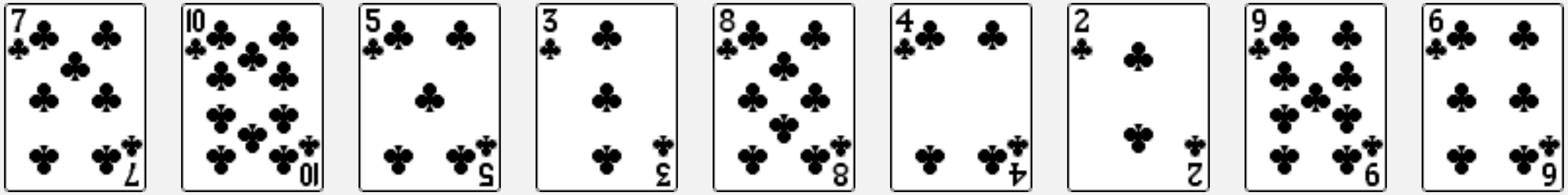
http://algs4.cs.princeton.edu

2.1 ELEMENTARY SORTS

- *selection sort*
- *insertion sort*

Selection sort demo

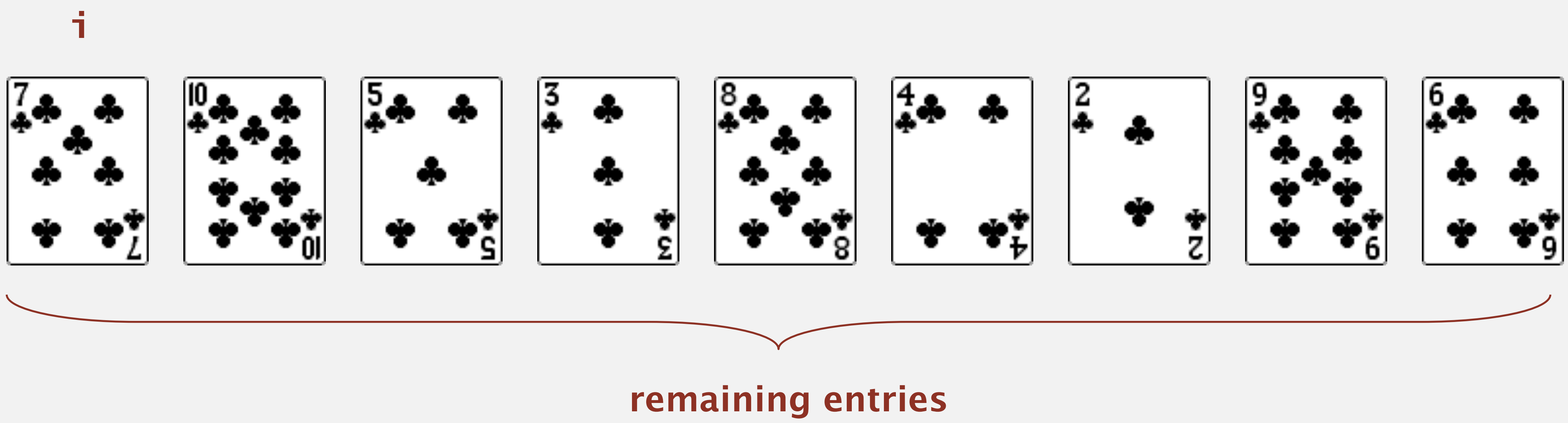
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



initial

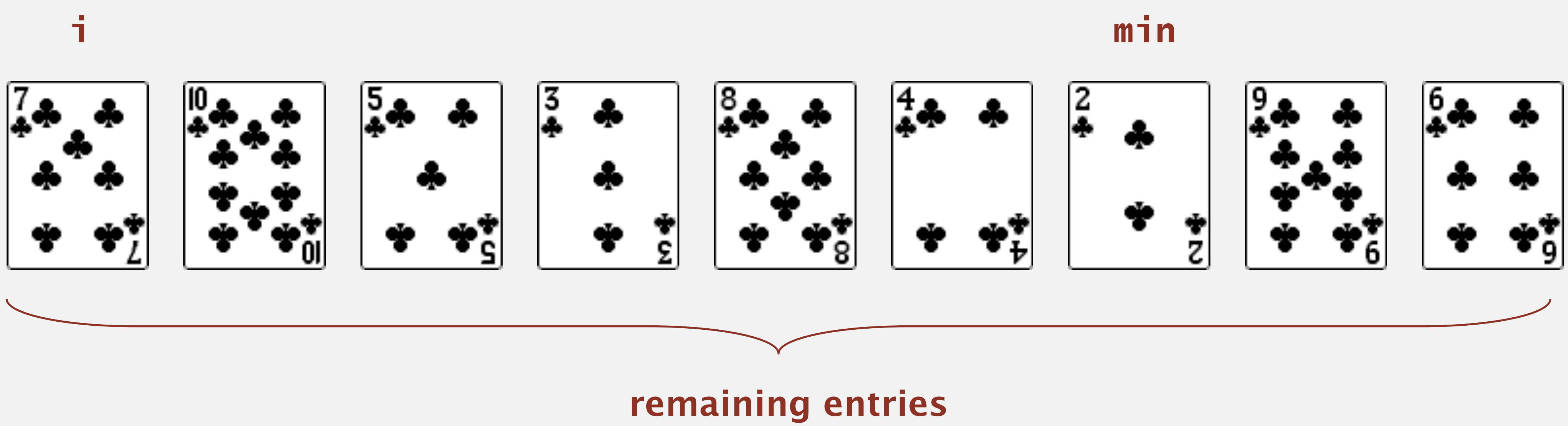
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



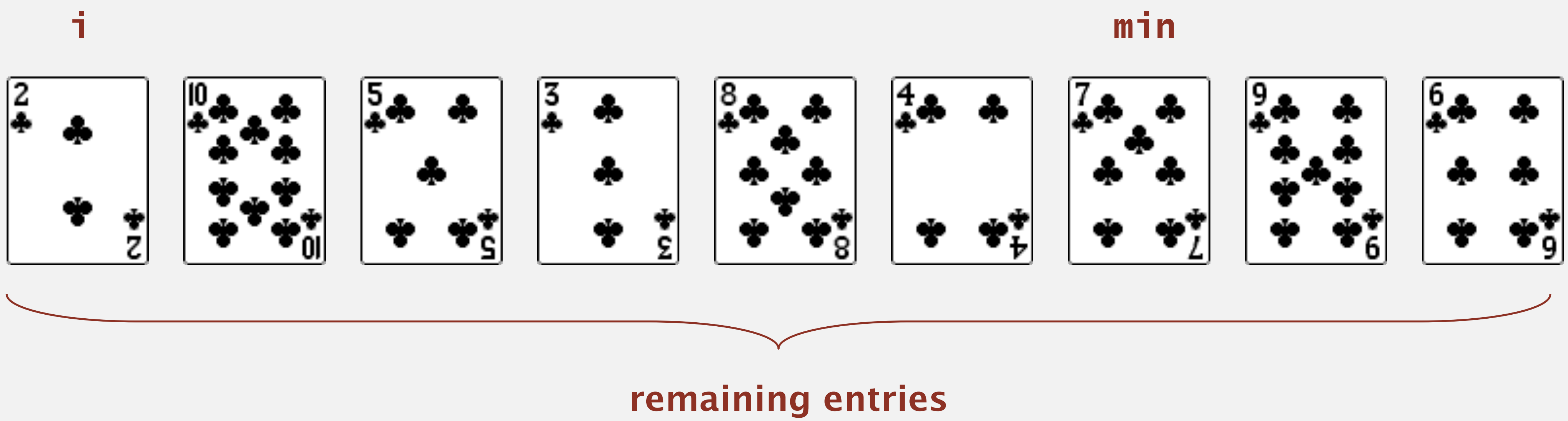
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



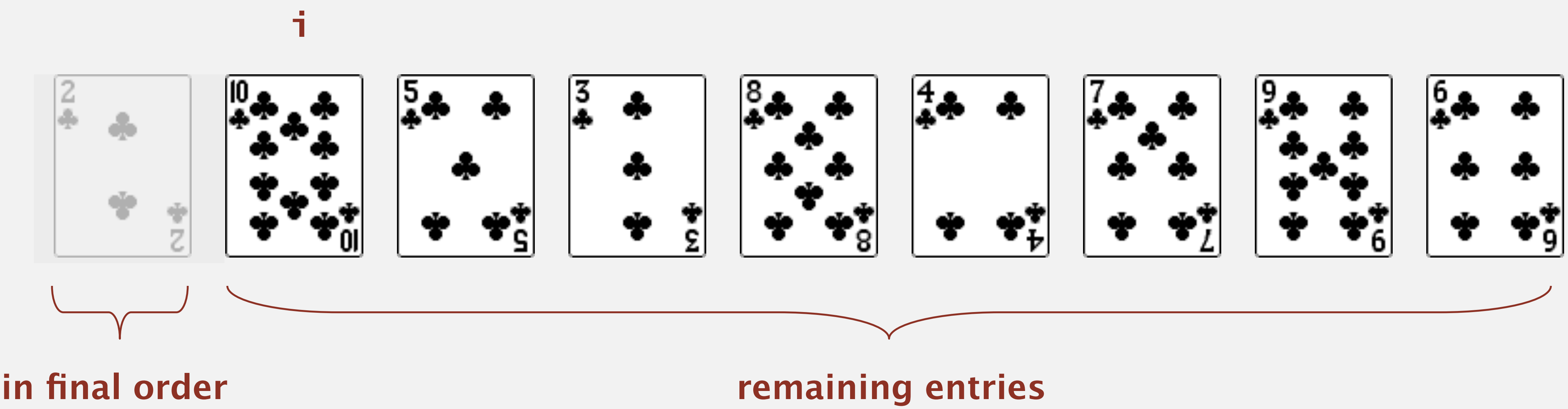
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



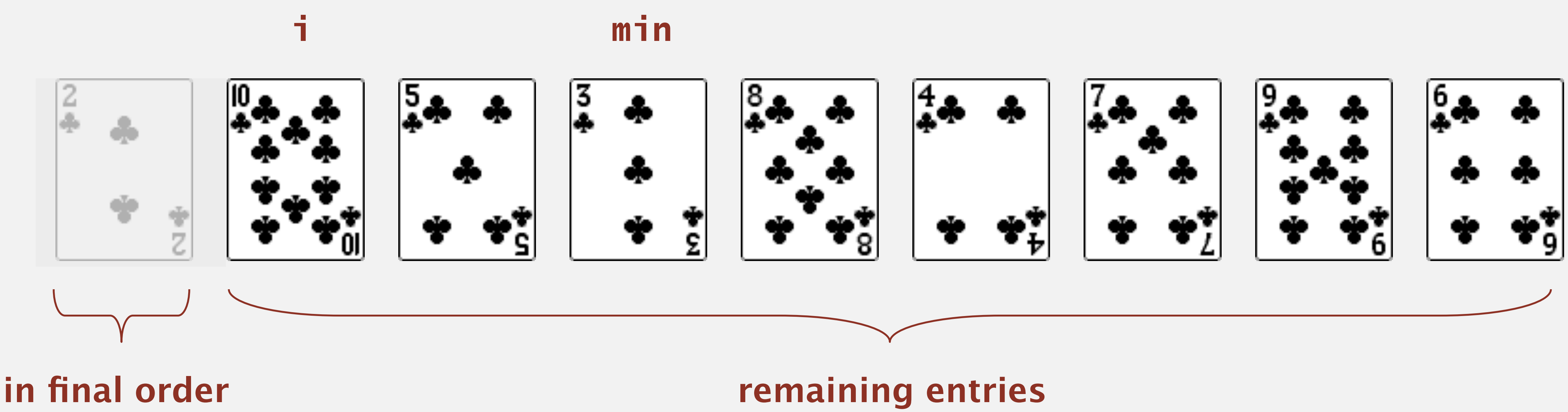
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



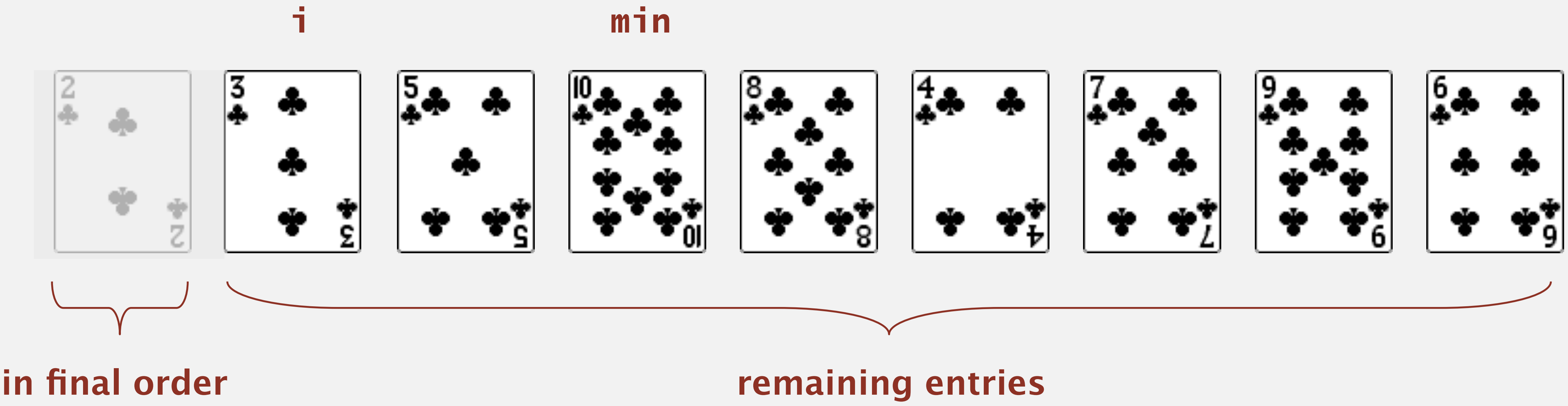
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



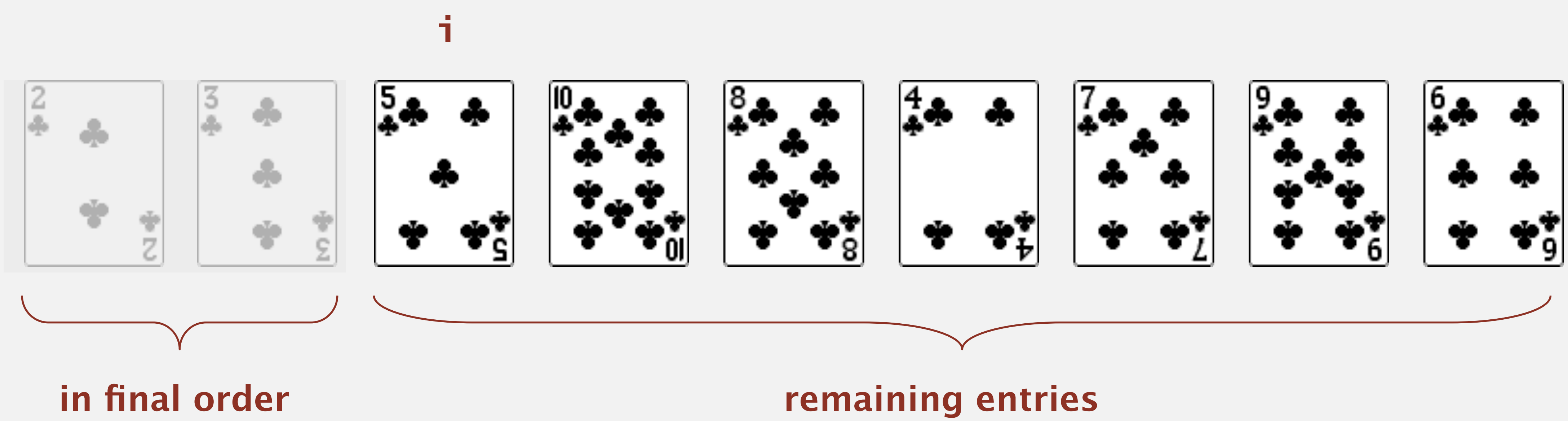
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



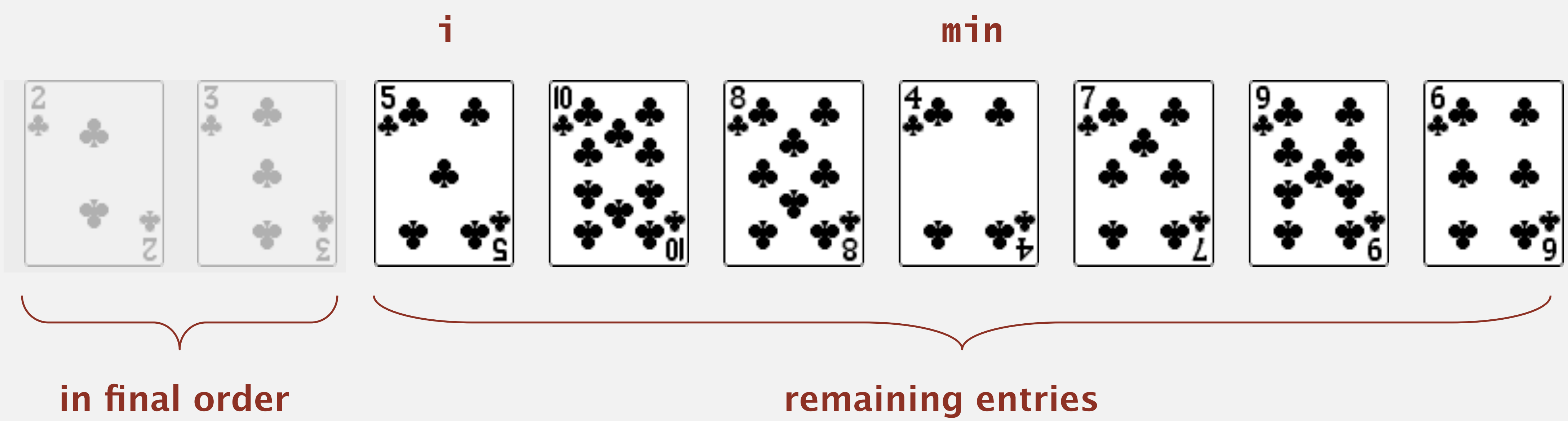
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



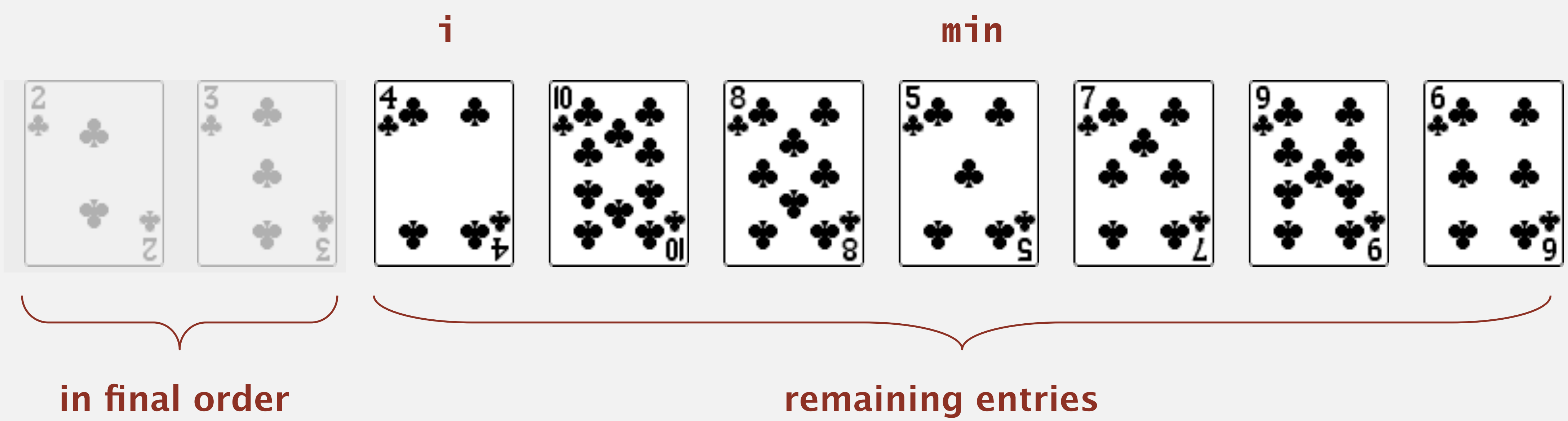
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



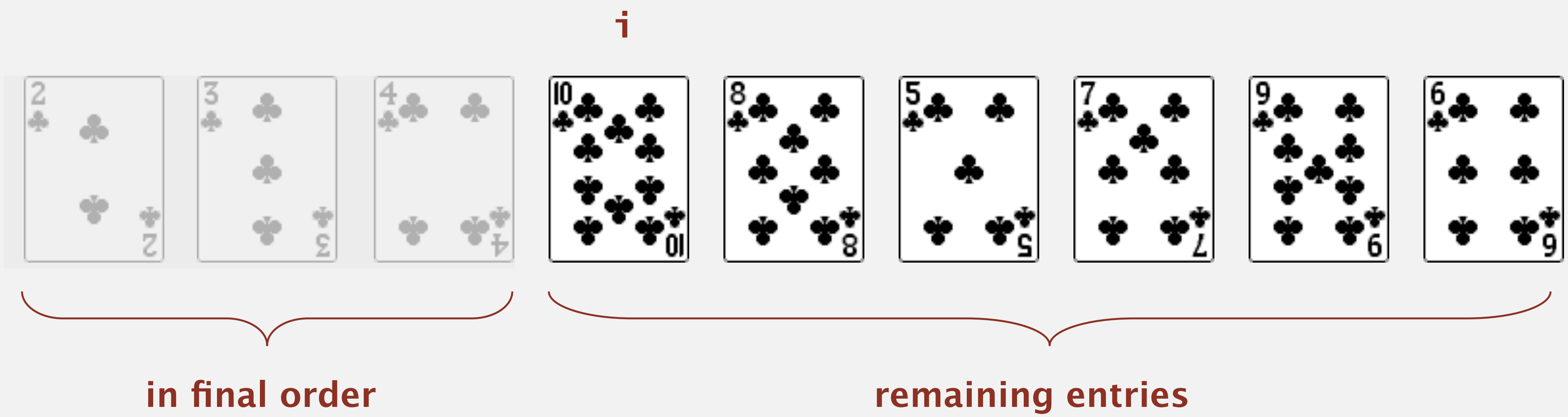
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



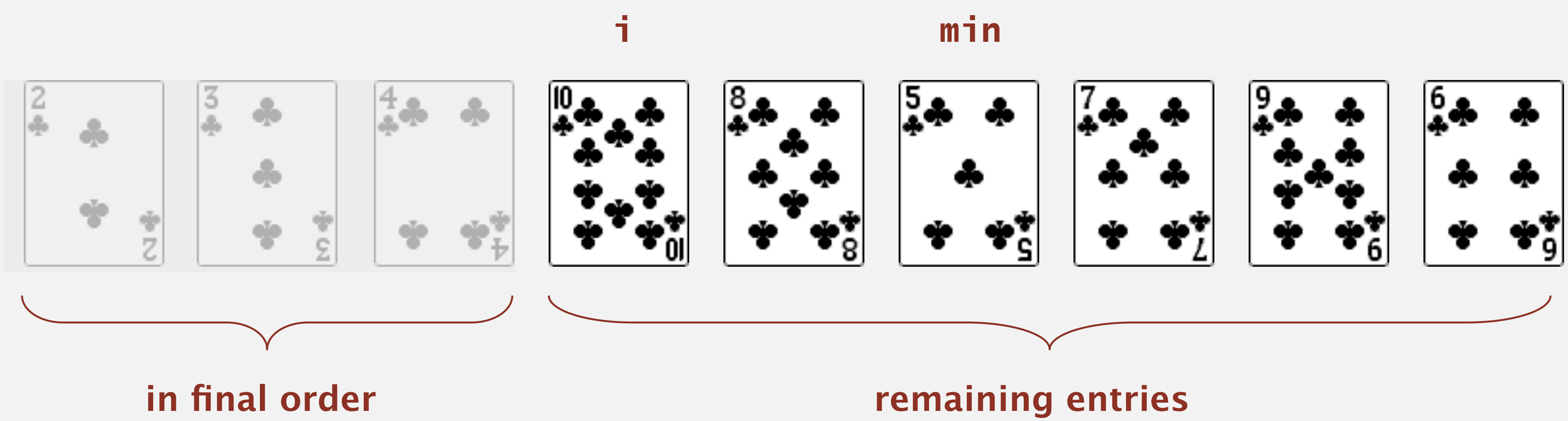
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



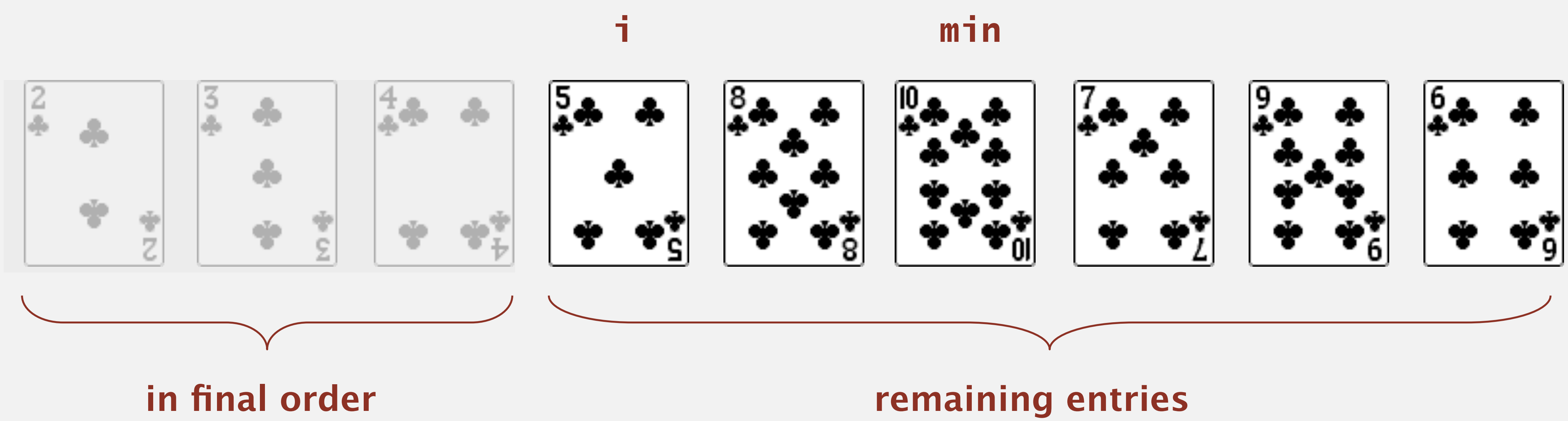
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



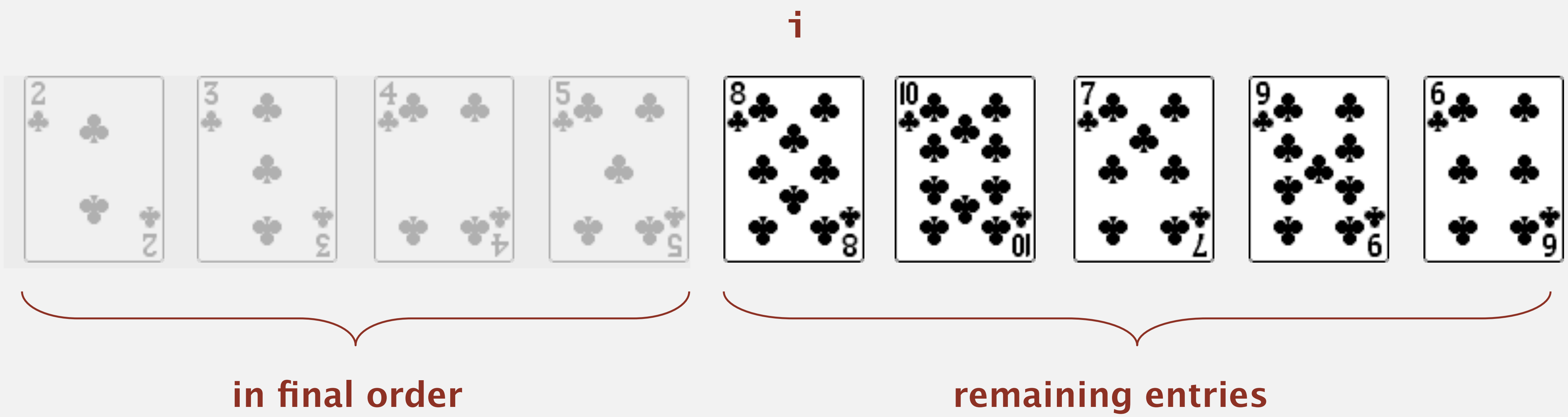
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



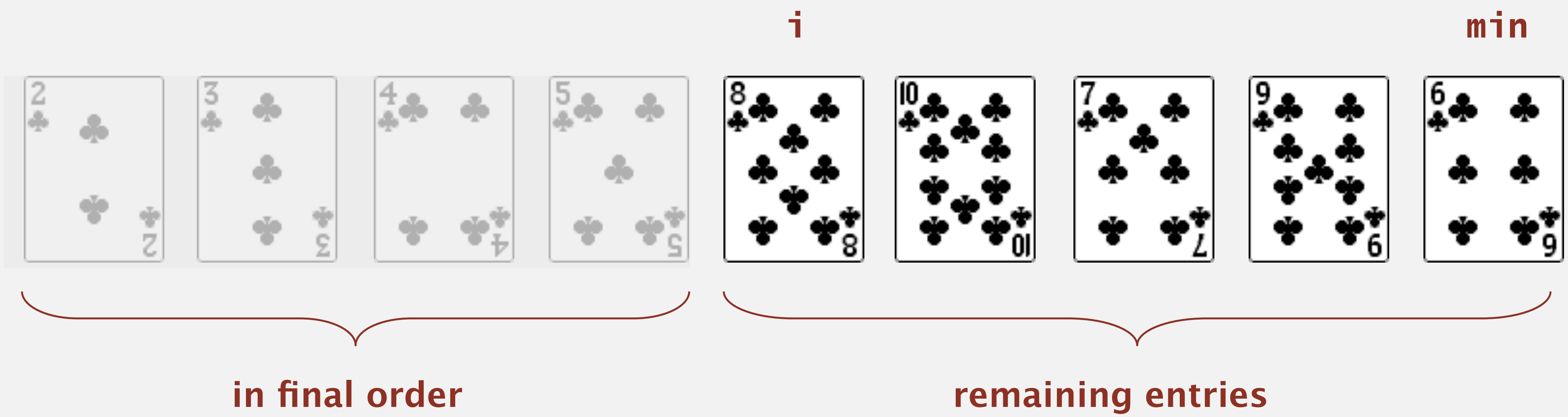
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



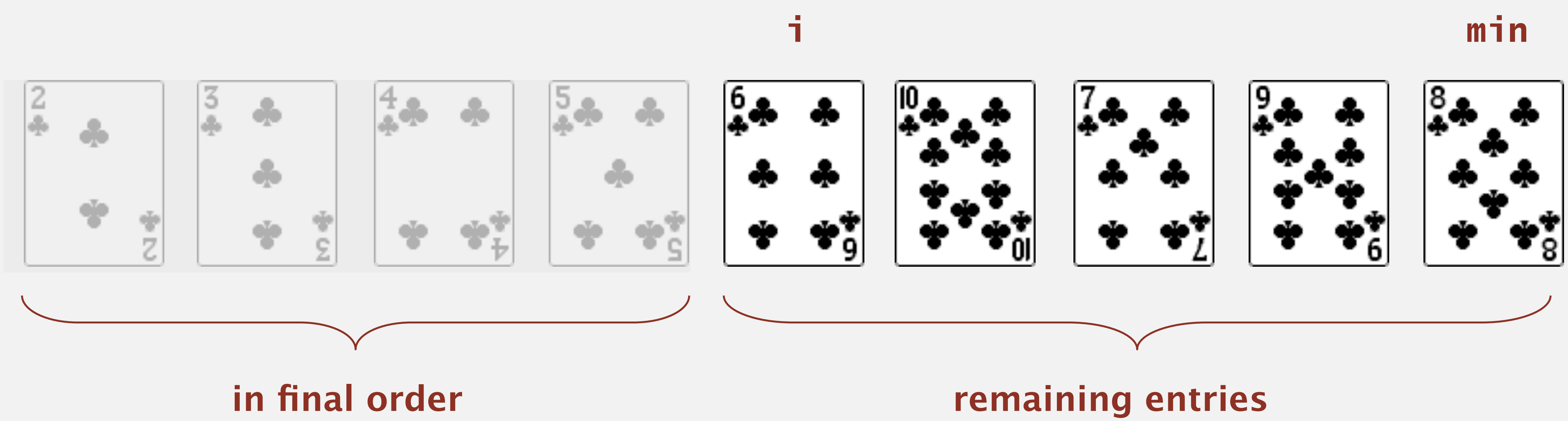
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



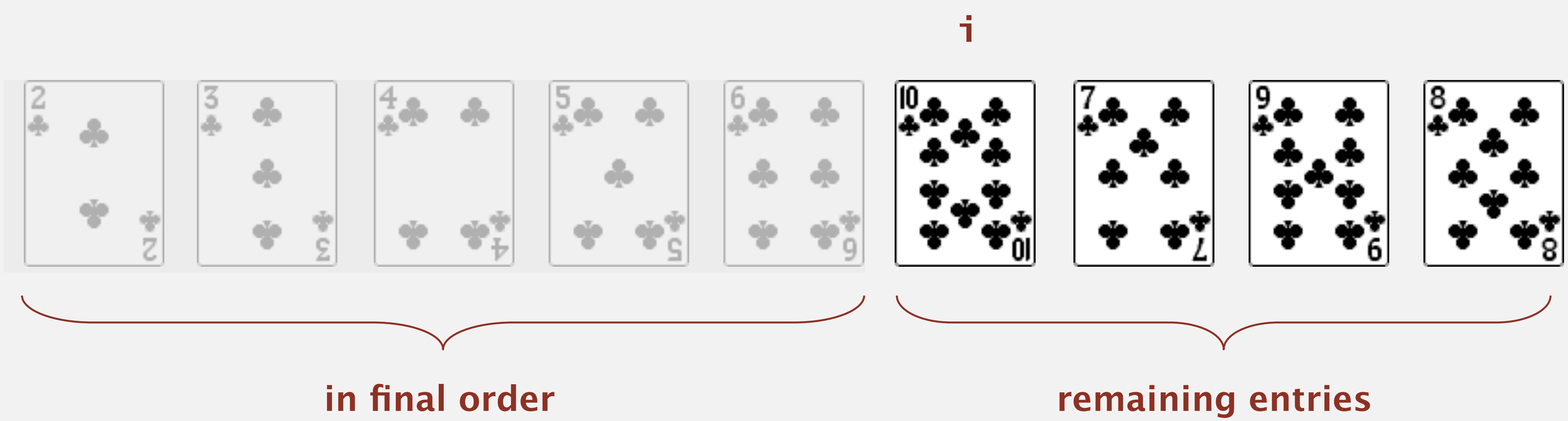
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



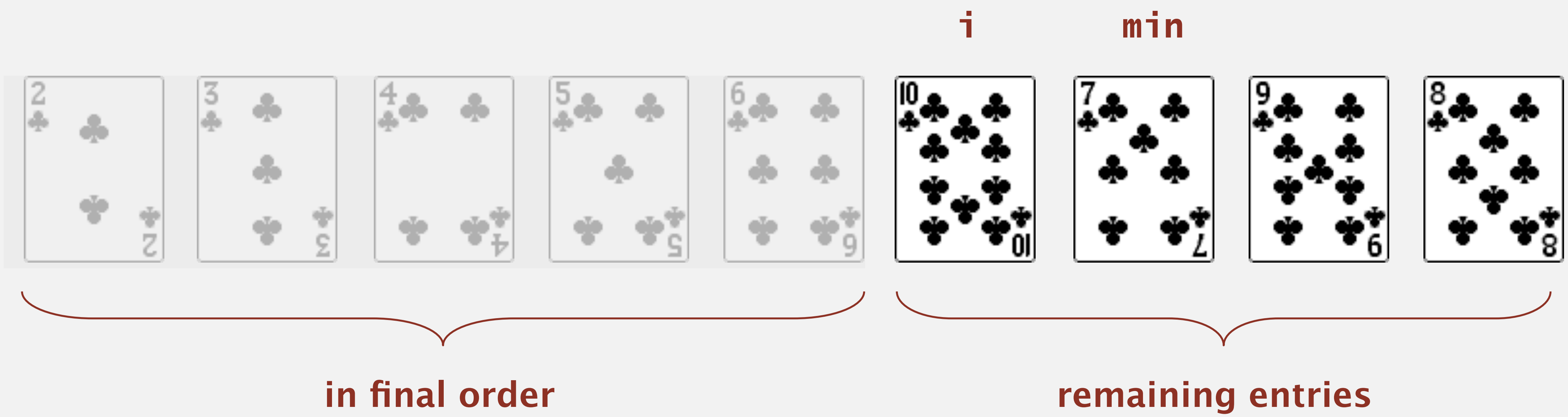
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



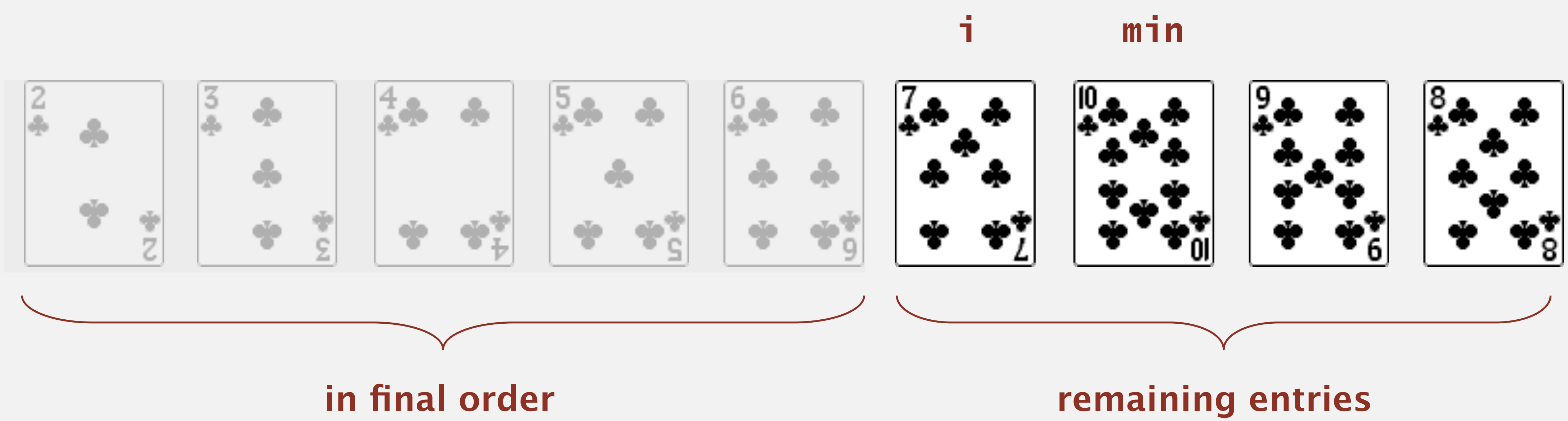
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



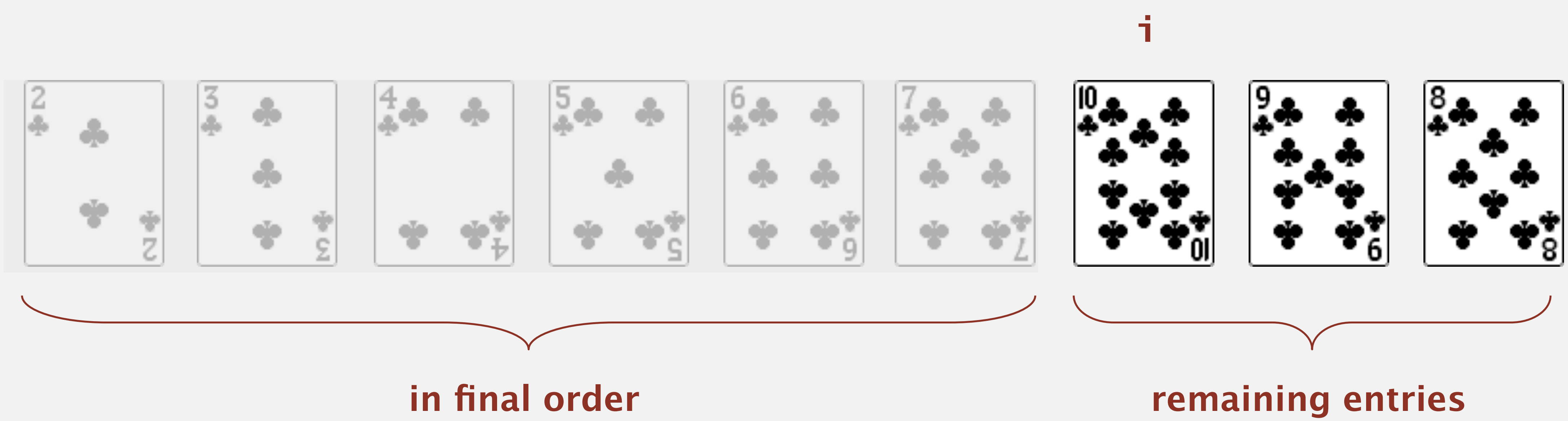
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



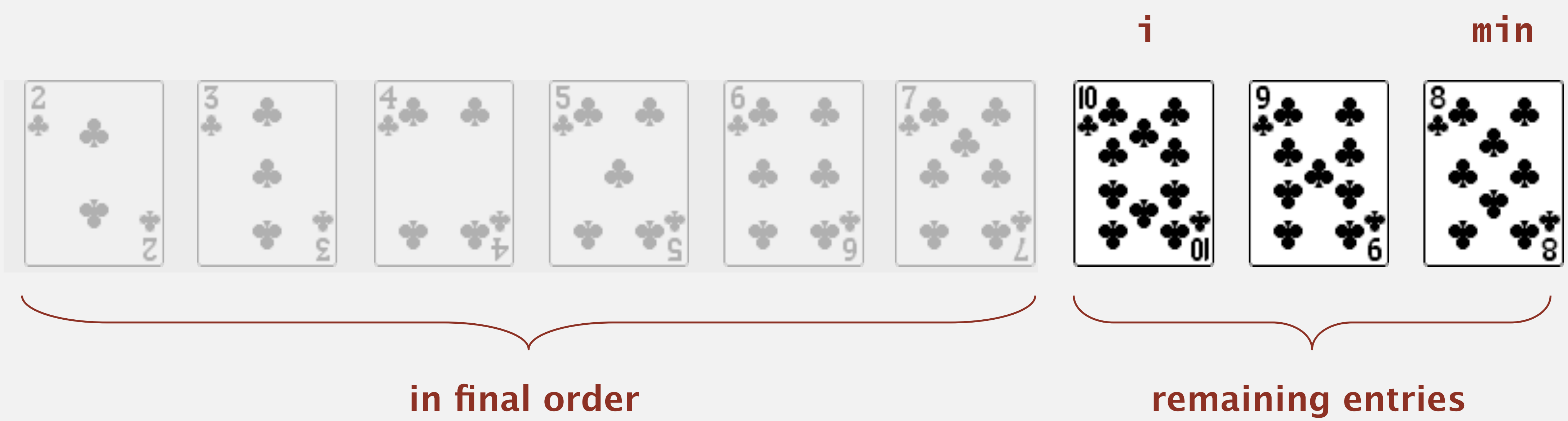
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



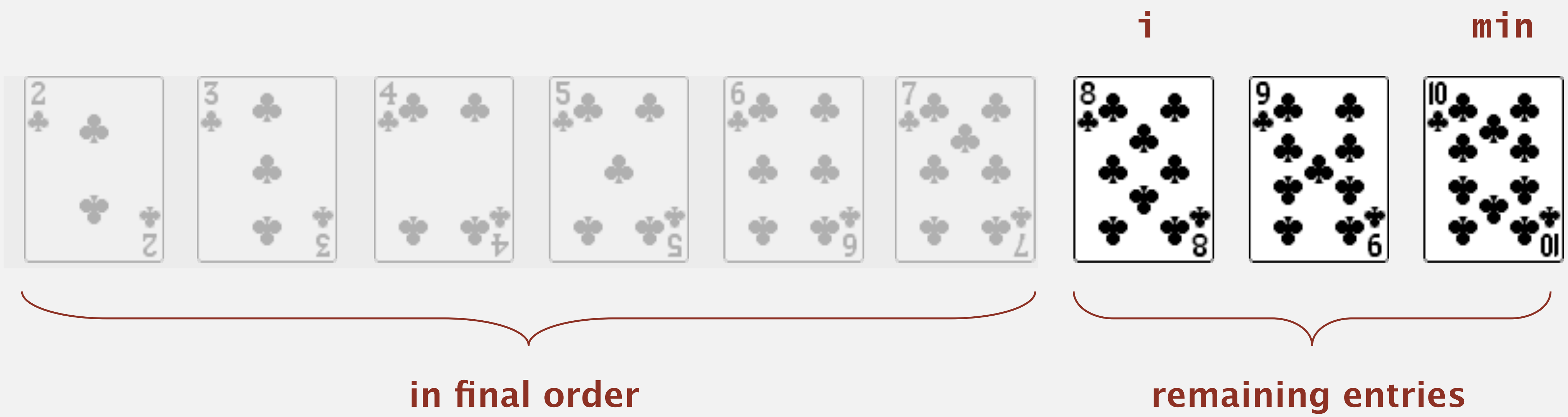
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



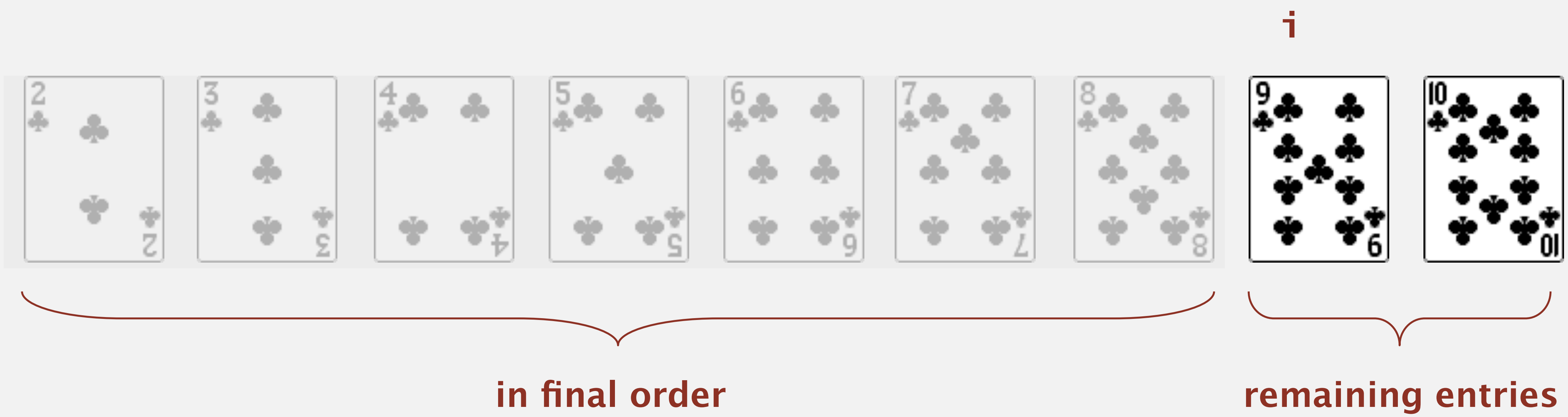
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



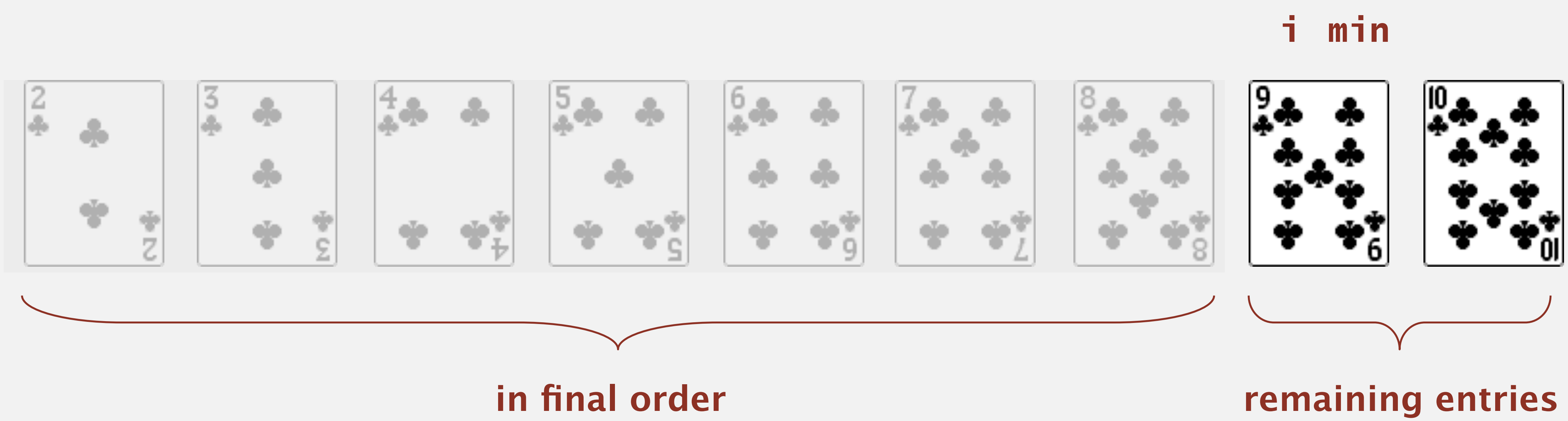
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



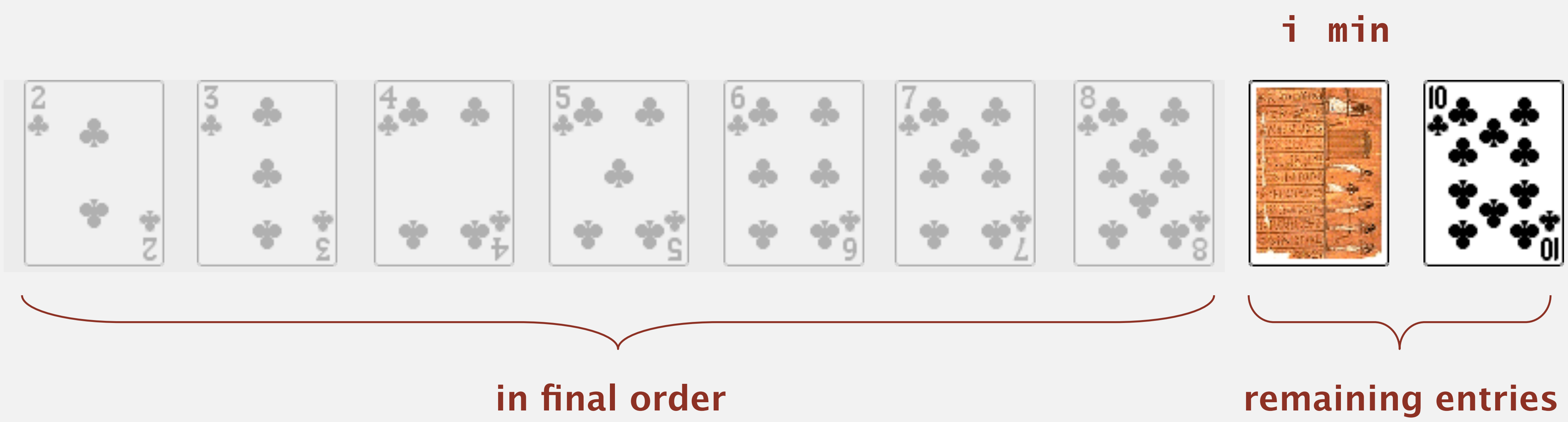
Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.

Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.

Selection sort demo

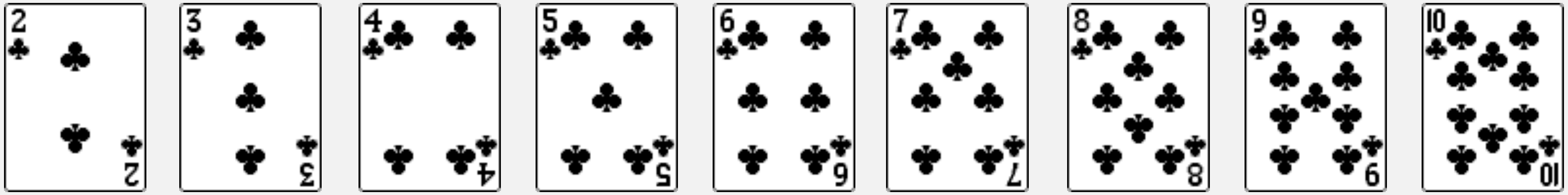
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.

Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.

Selection sort demo

- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



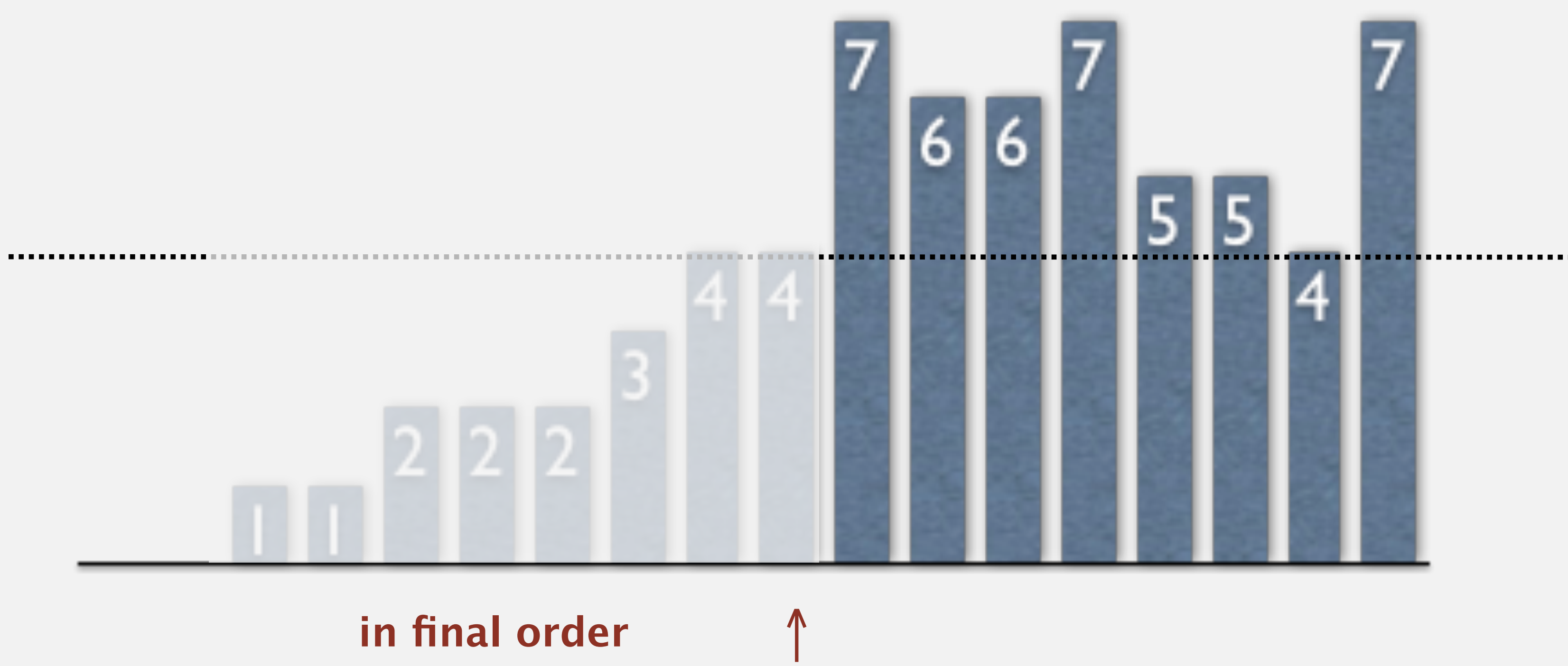
sorted

Selection sort

Algorithm. ↑ scans from left to right.

Invariants.

- Entries the left of ↑ (including ↑) fixed and in ascending order.
- No entry to right of ↑ is smaller than any entry to the left of ↑.



Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

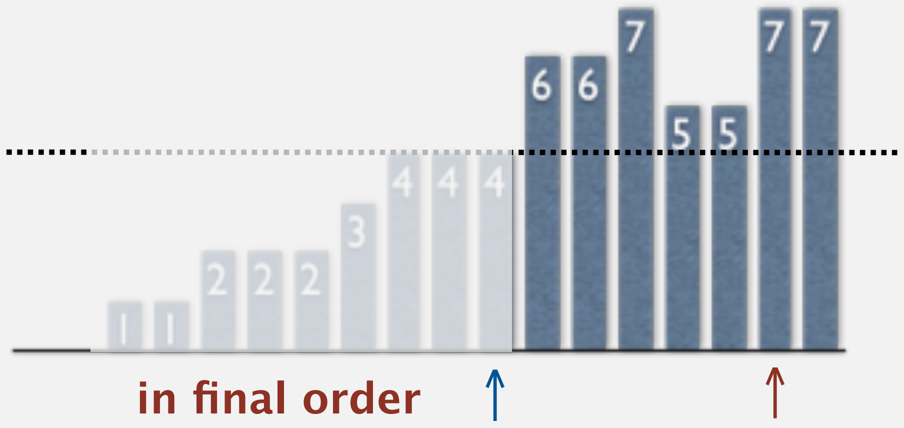
```
i++;
```

- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```



Two useful sorting abstractions

Helper functions. Refer to data only through compares and exchanges.

Less. Is item `v` less than `w` ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array `a[]` at index `i` with the one at index `j`.

```
private static void exch(Object[] a, int i, int j)
{
   Object swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

Selection sort: Java implementation

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* see previous slide */  }

   private static void exch(Object[] a, int i, int j)
   {  /* see previous slide */  }
}
```

http://algs4.cs.princeton.edu/21elementary/Selection.java.html

Generic methods

Oops. The compiler complains.

```
% javac Selection.java
Note: Selection.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
 % javac -Xlint:unchecked Selection.java
 Selection.java:83: warning: [unchecked] unchecked call to
compareTo(T) as a member of the raw type java.lang.Comparable
        return (v.compareTo(w) < 0);
                          ^
1 warning
```

Q. How to silence the compiler?

Generic methods

Pedantic (type-safe) version. Compiles without any warnings.

generic type variable
(type inferred from argument; must be Comparable)

```
public class SelectionPedantic
{
    public  static <Key extends Comparable<Key>> void sort(Key[] a)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    {  /* as before */  }

    private static Object void exch(Object[] a, int i, int j)
    {  /* as before */  }
}
```

http://algs4.cs.princeton.edu/21elementary/SelectionPedantic.java.html

Remark. Use type-safe version in system code (but not in lecture).

Selection sort: animations

20 random items

▲ algorithm position

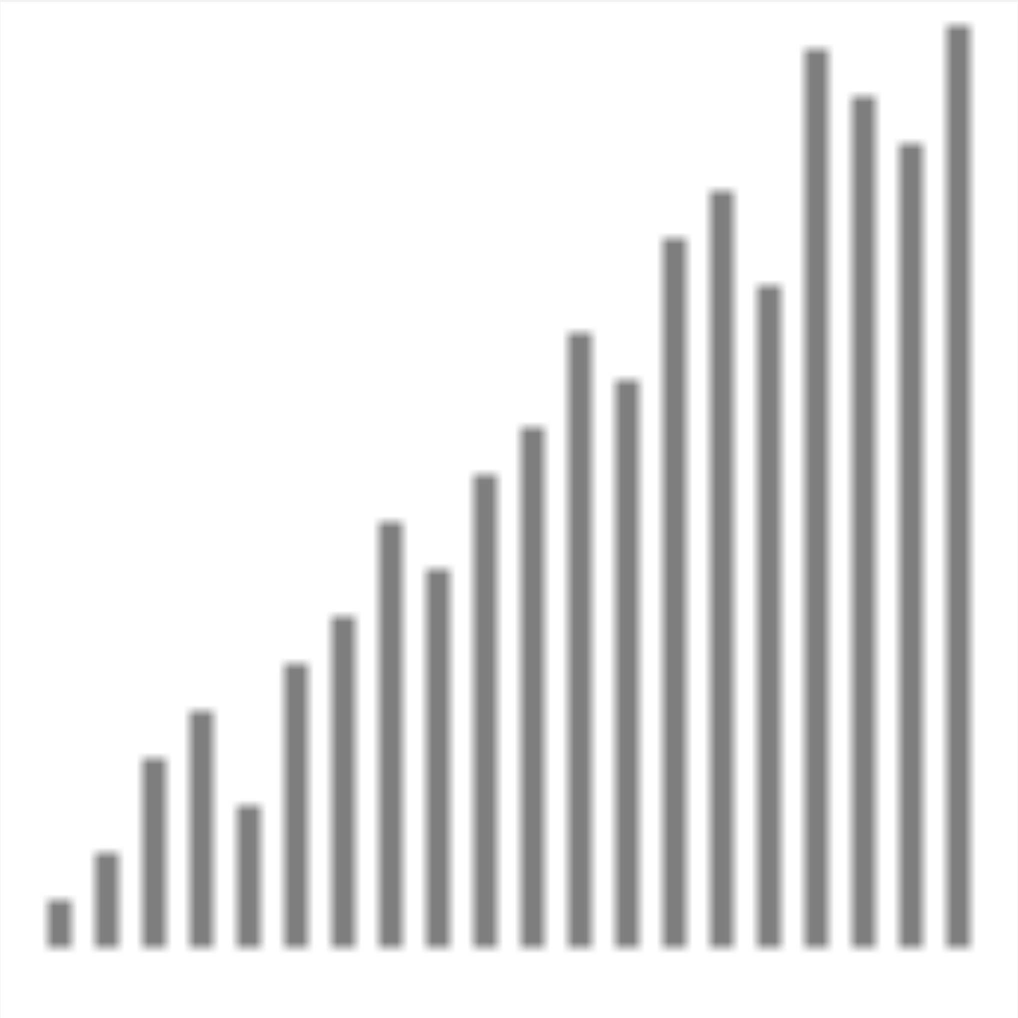
in final order

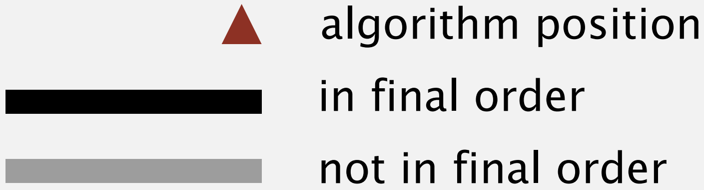
not in final order

http://www.sorting-algorithms.com/selection-sort

Selection sort: animations

20 partially-sorted items





http://www.sorting-algorithms.com/selection-sort

Selection sort: mathematical analysis

Proposition. Selection sort uses $(N-1) + (N-2) + \ldots + 1 + 0 \sim N^2/2$ compares and N exchanges to sort any array of N items.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if input is sorted.

Data movement is minimal. Linear number of exchanges—exactly N.

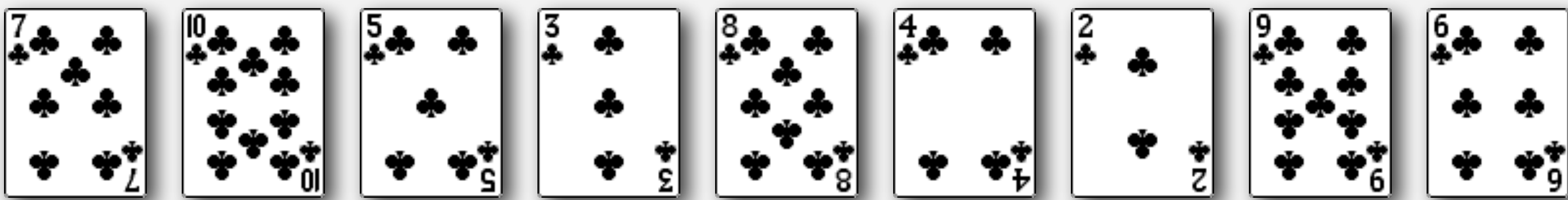
http://algs4.cs.princeton.edu

2.1 ELEMENTARY SORTS

- *selection sort*
- *insertion sort*

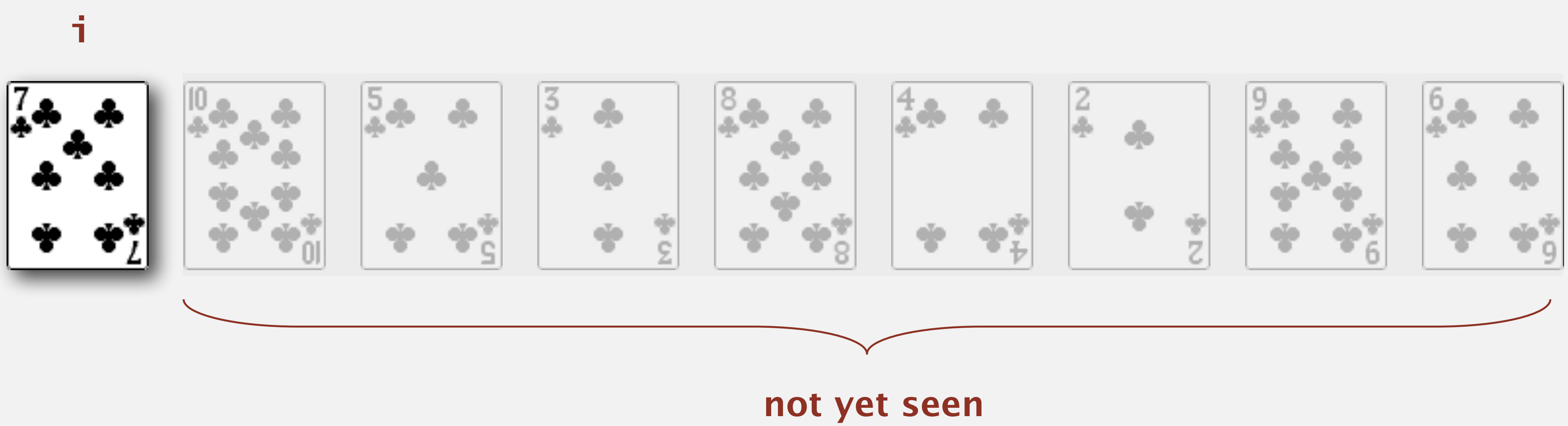
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



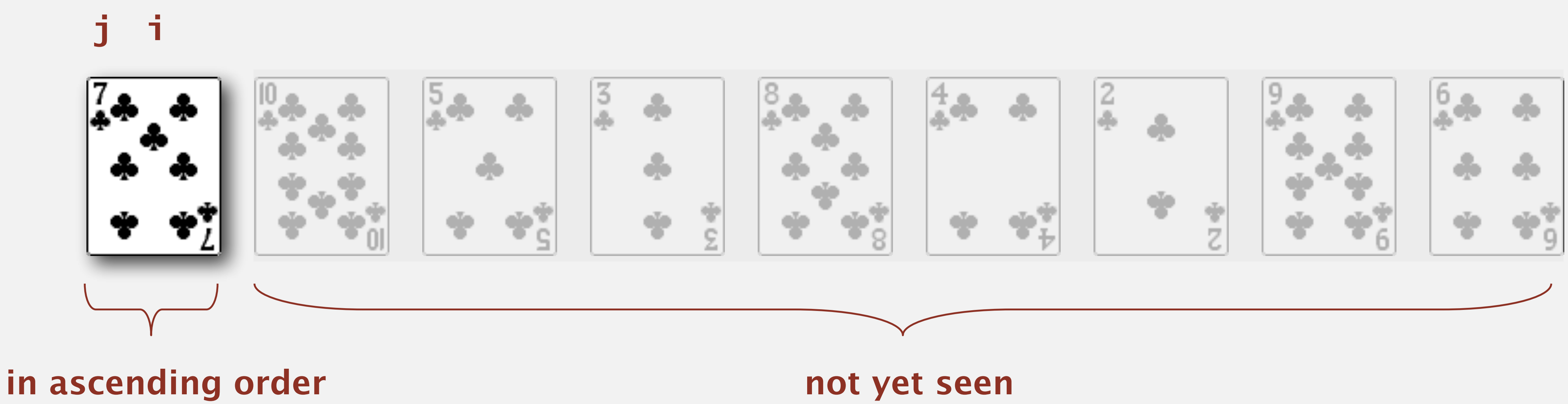
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



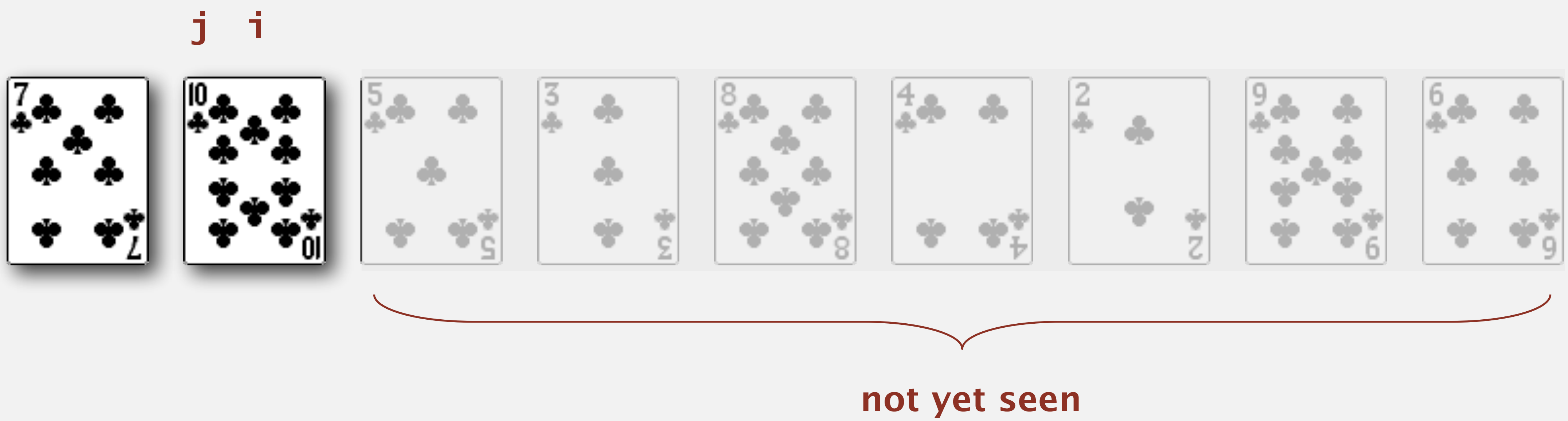
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



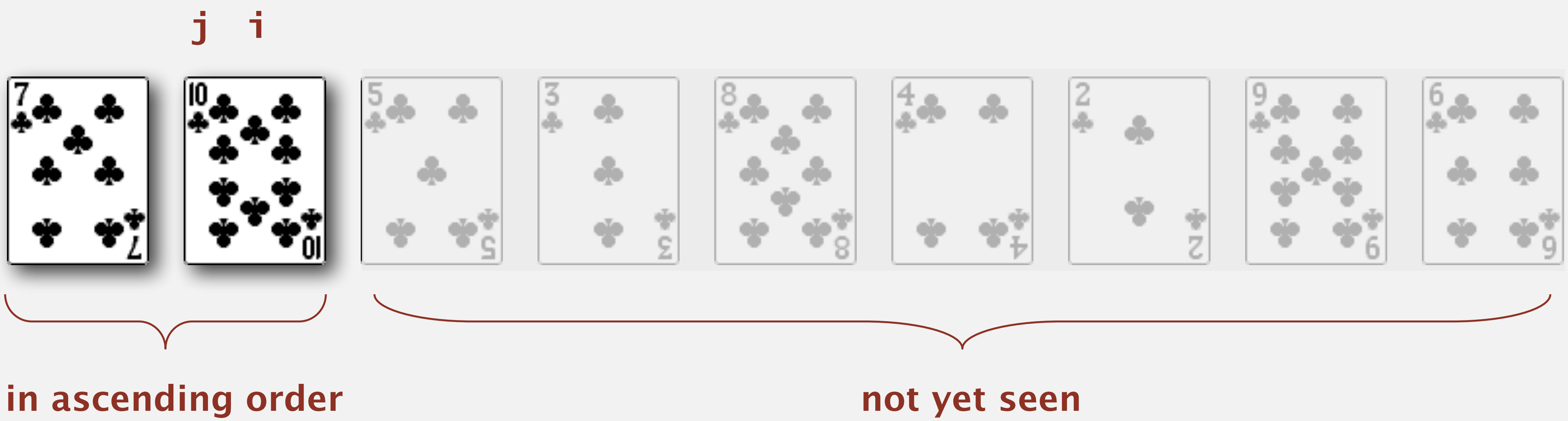
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



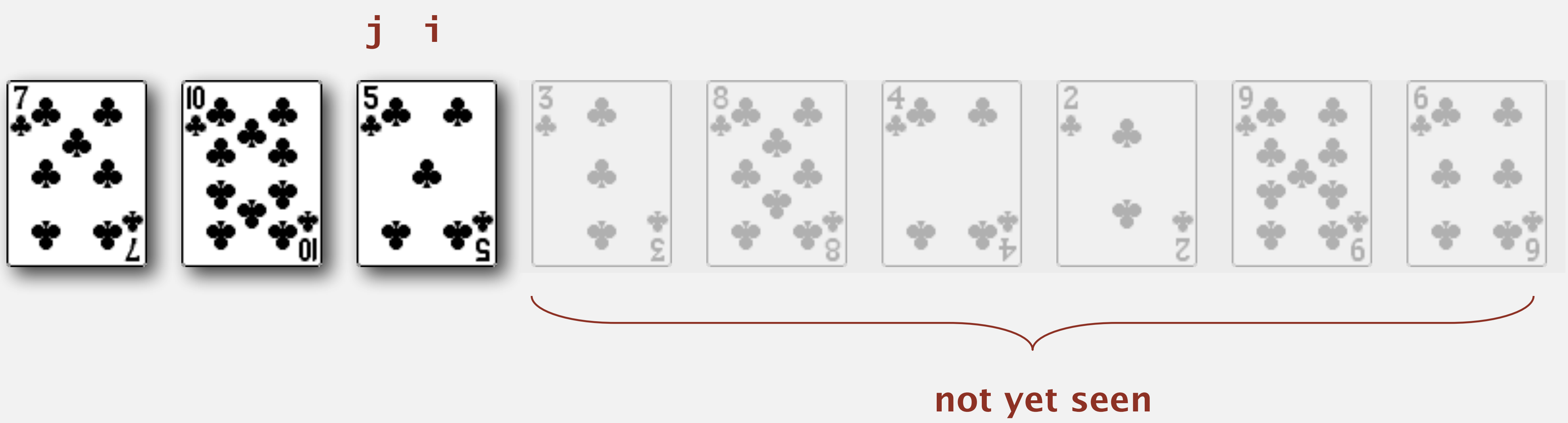
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



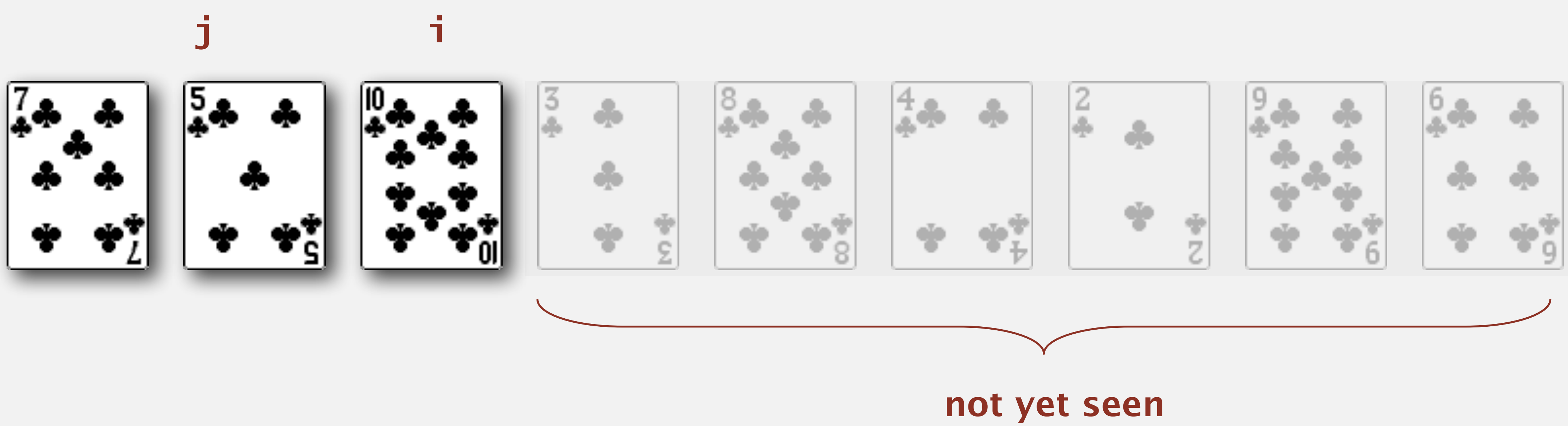
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



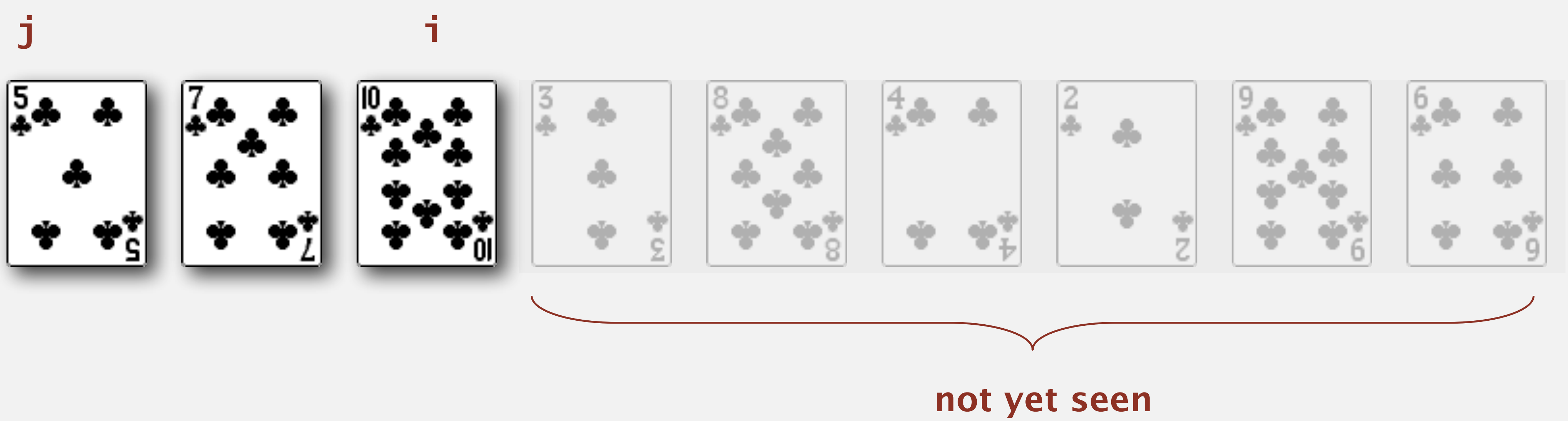
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



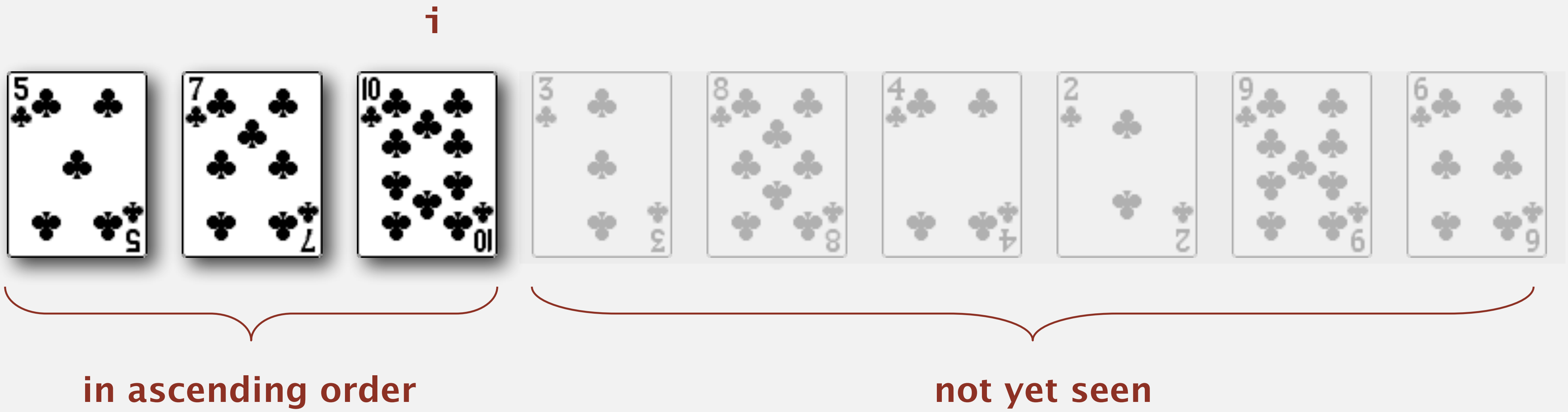
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



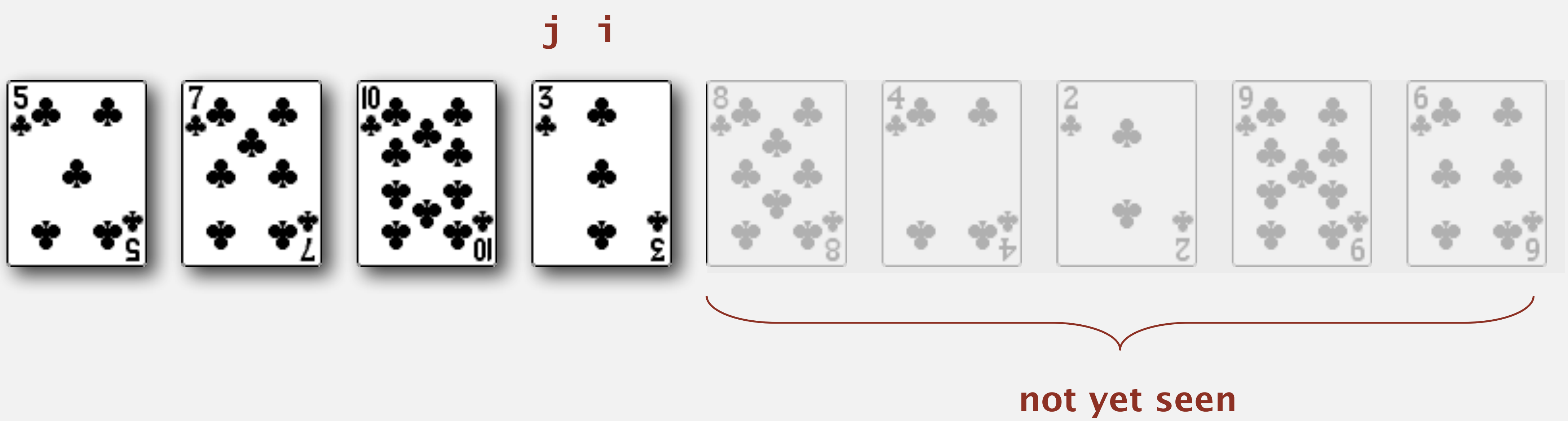
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



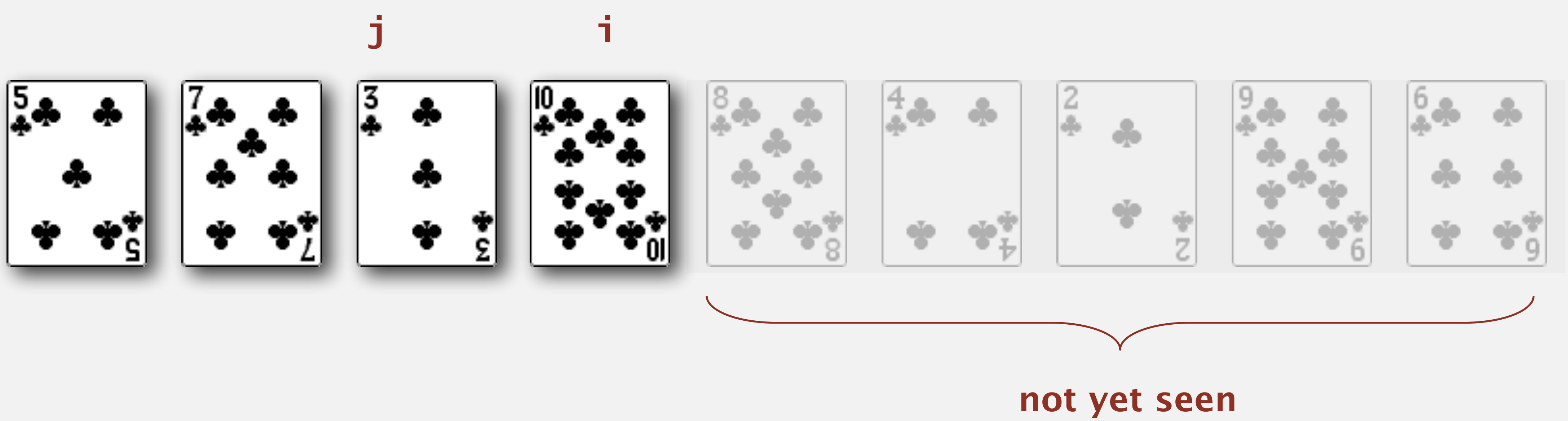
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



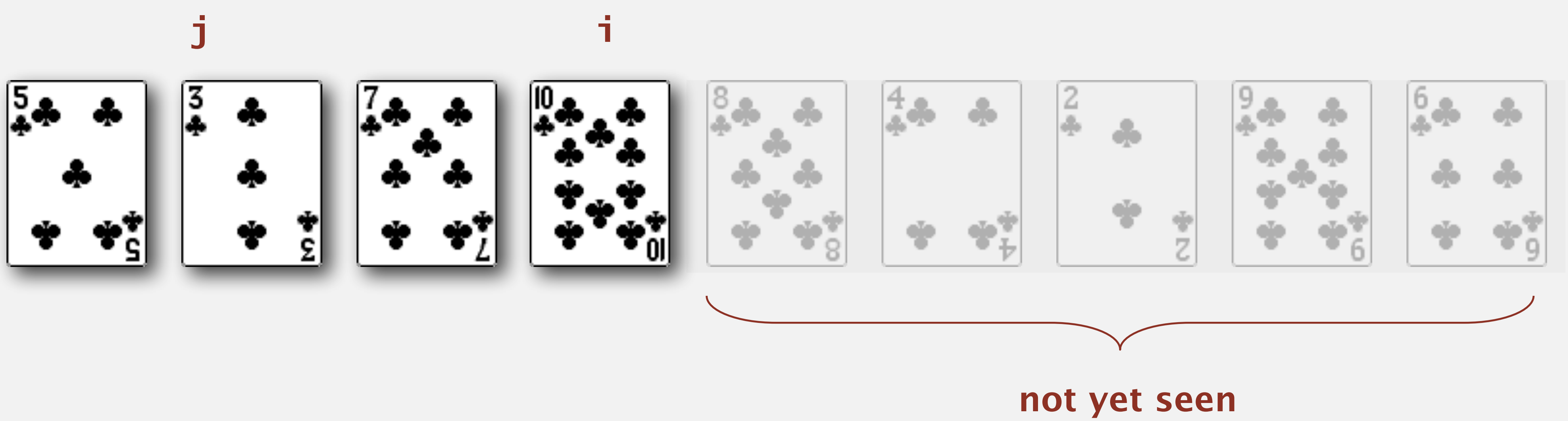
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



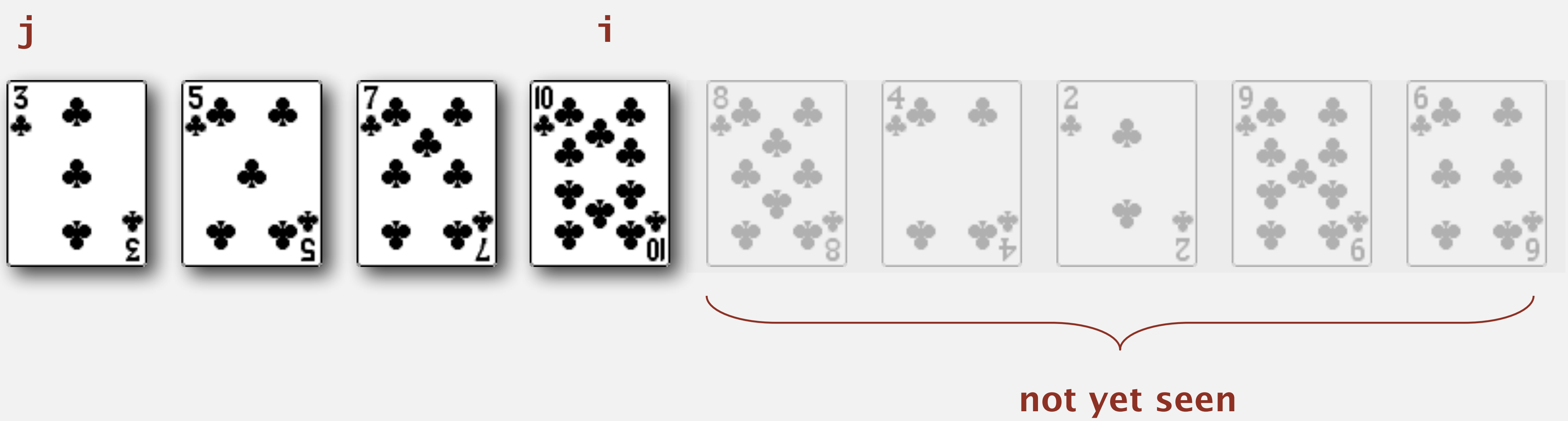
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



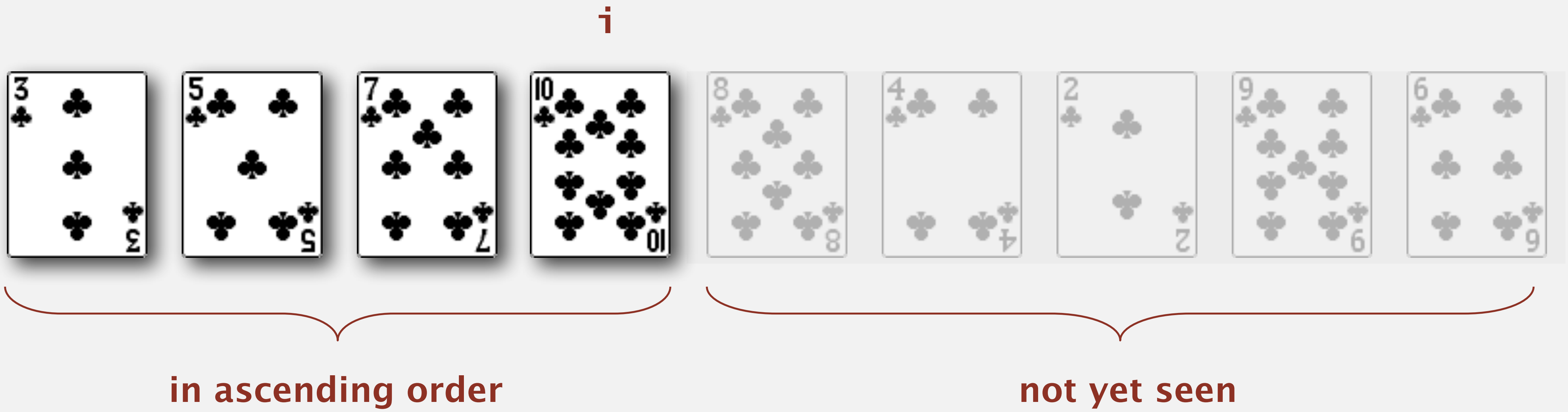
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



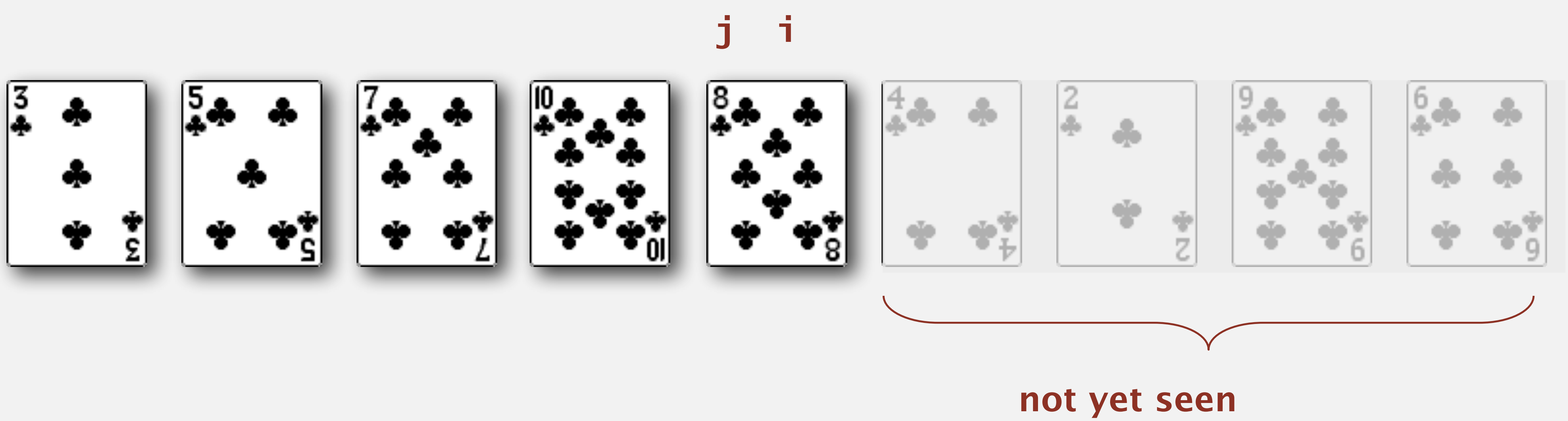
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



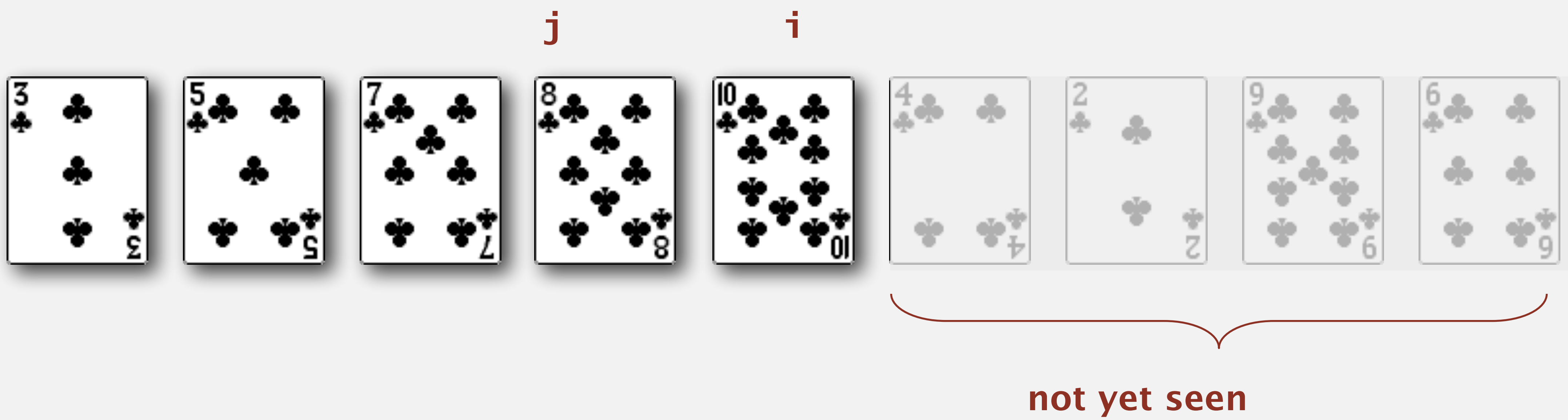
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



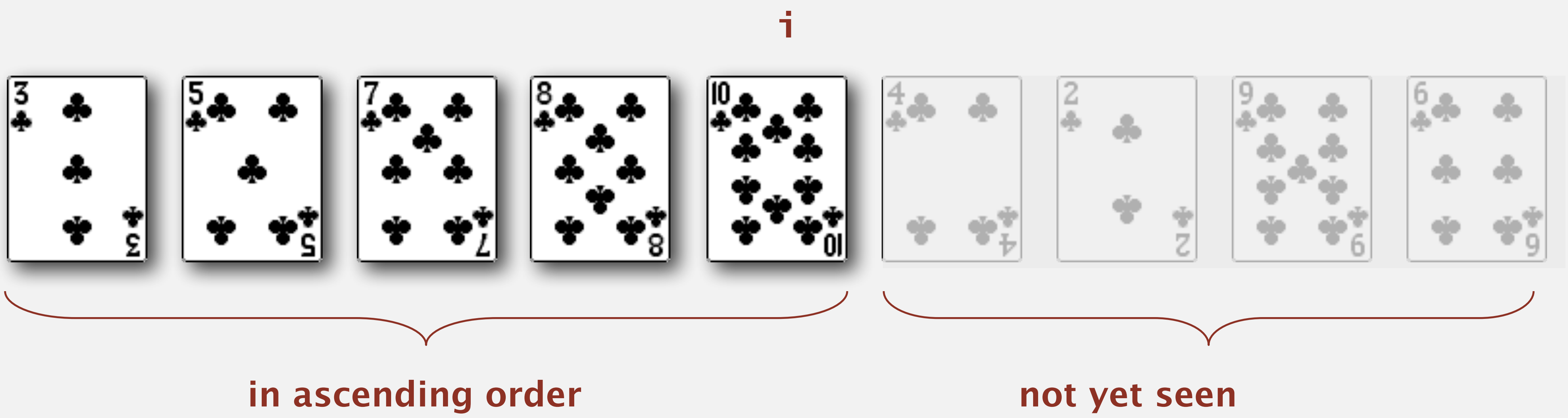
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



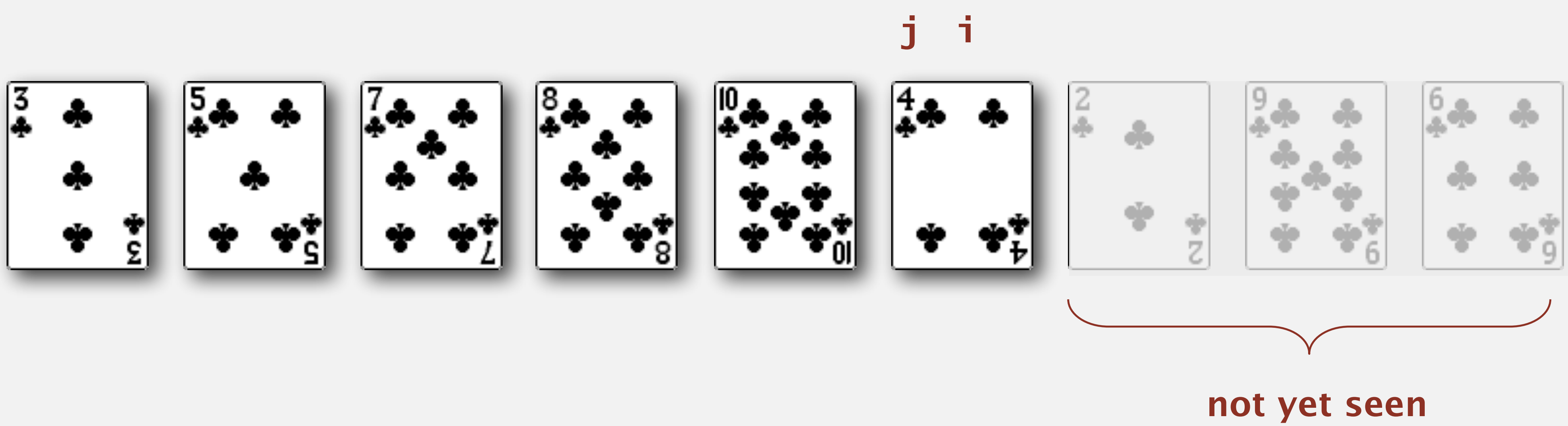
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



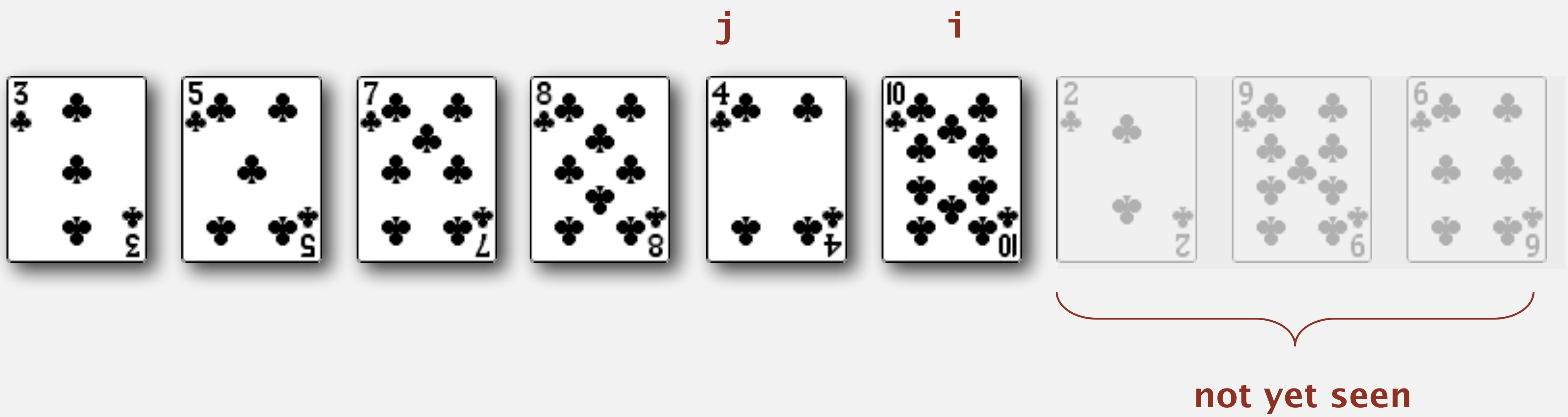
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



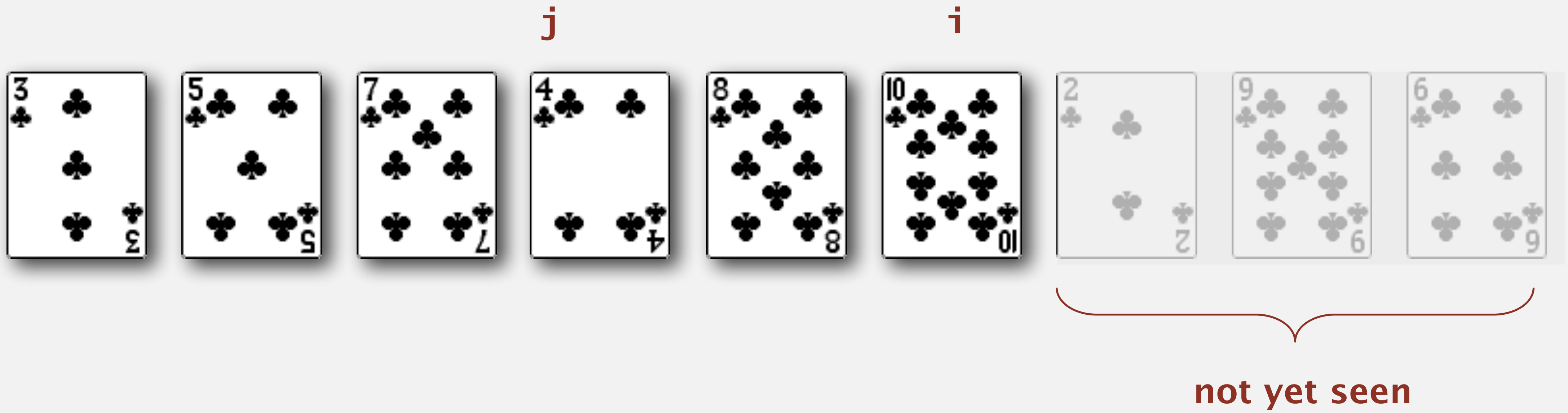
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



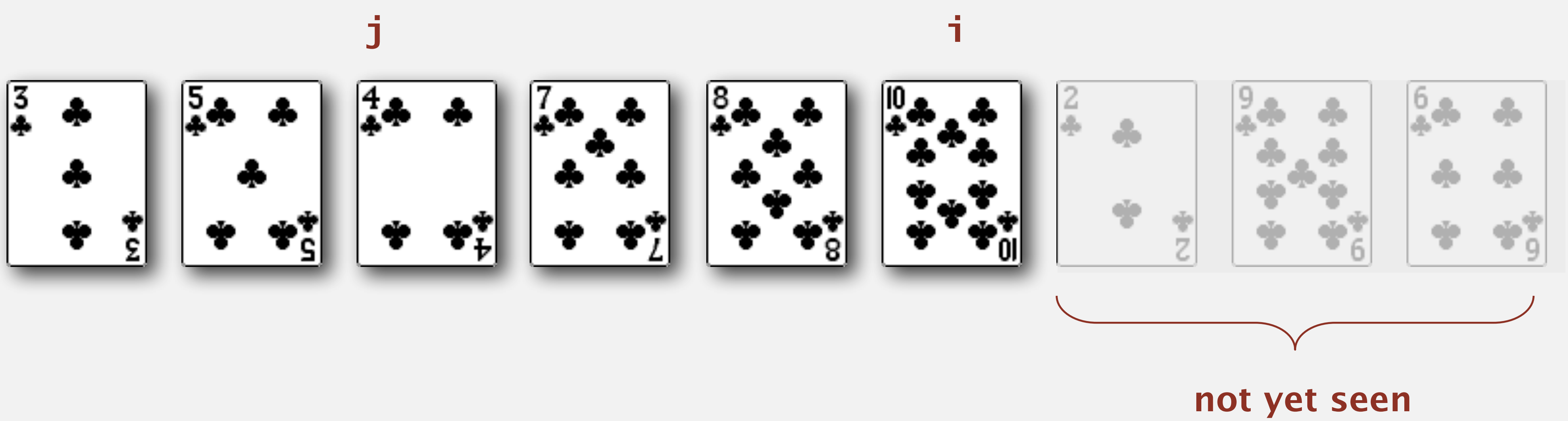
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



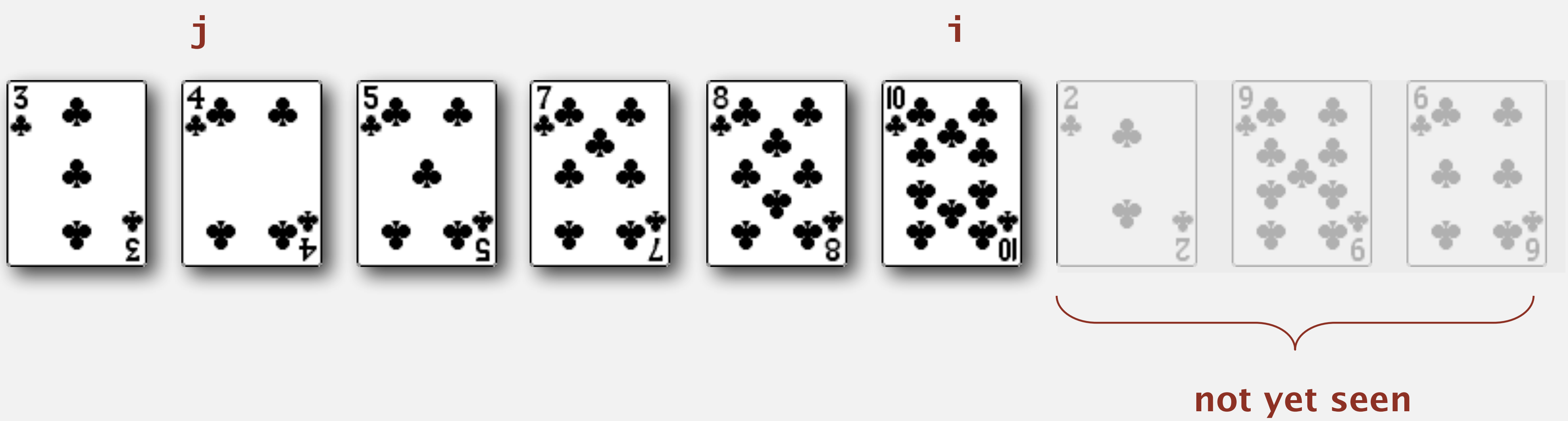
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



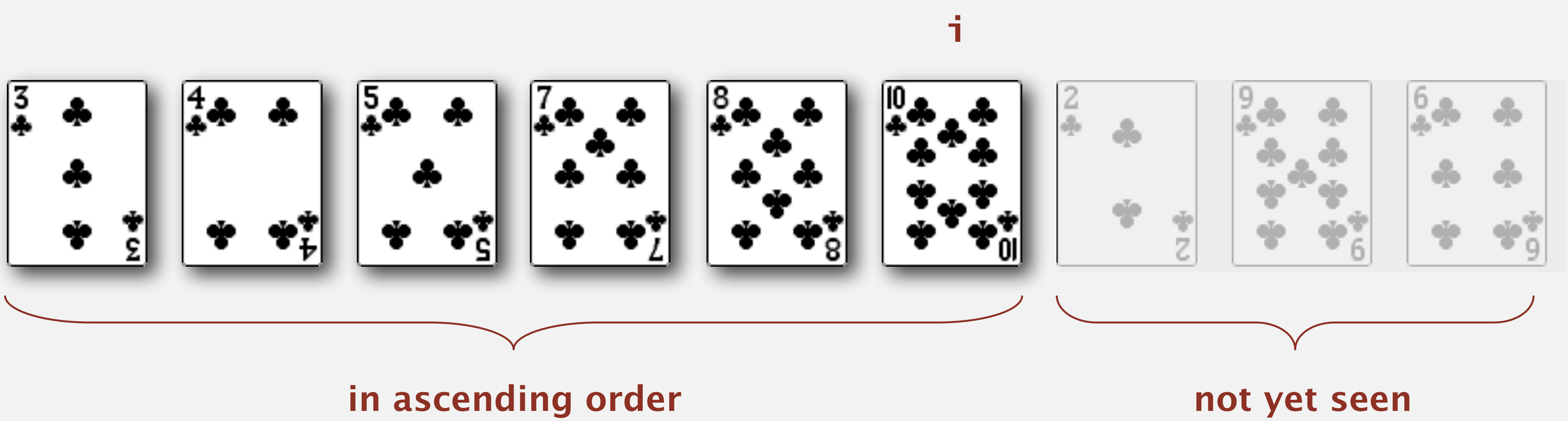
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



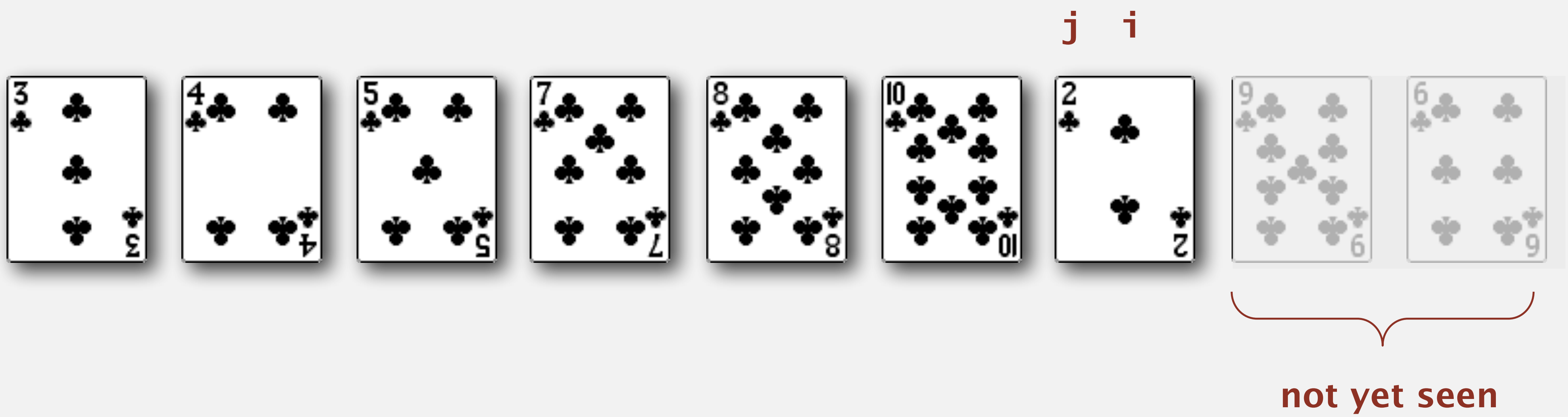
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



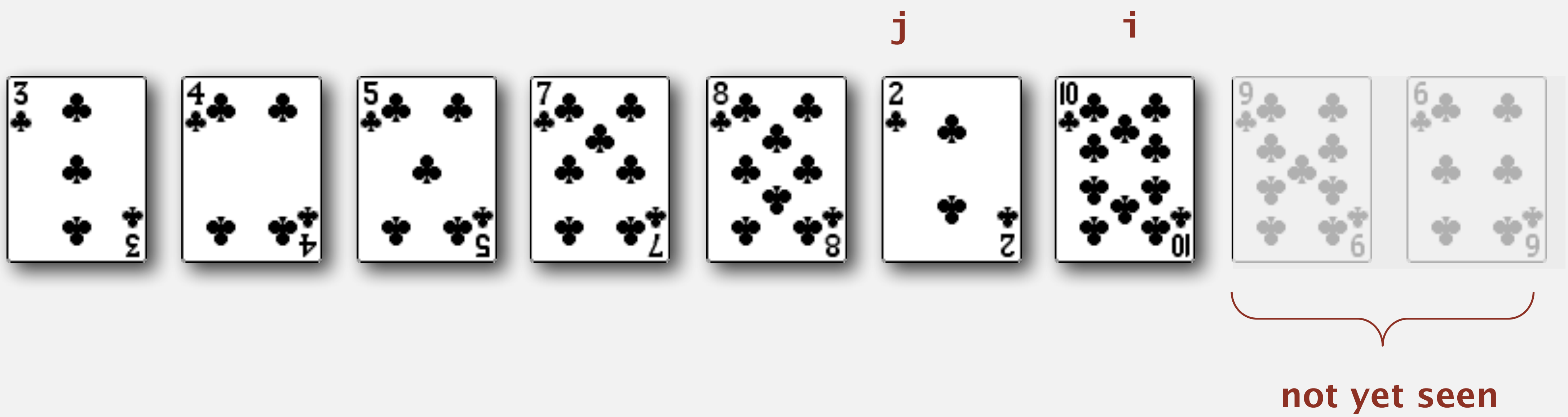
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



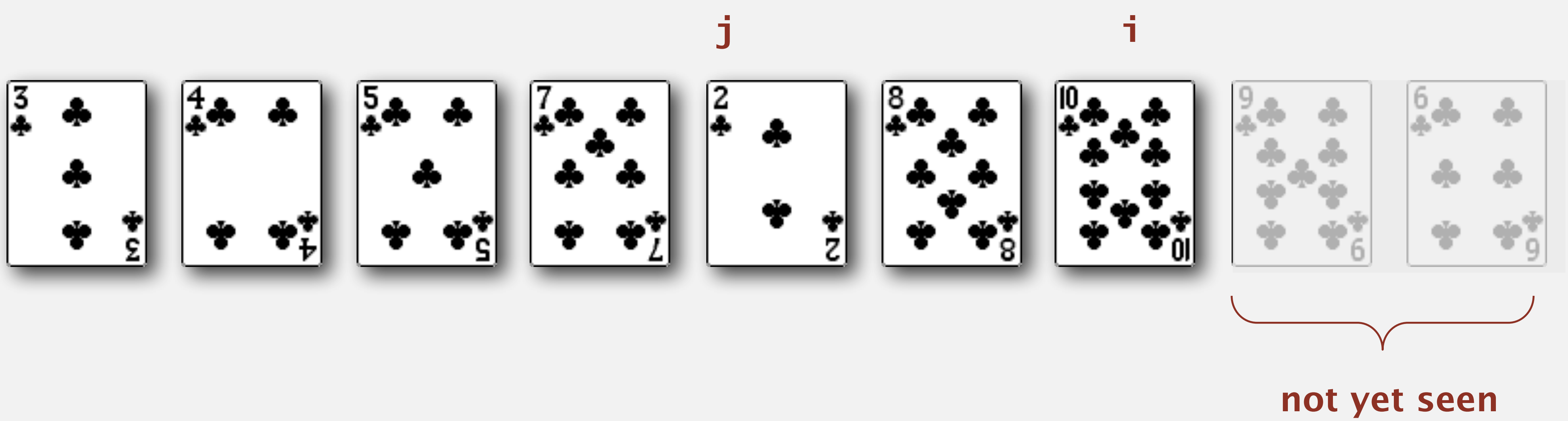
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



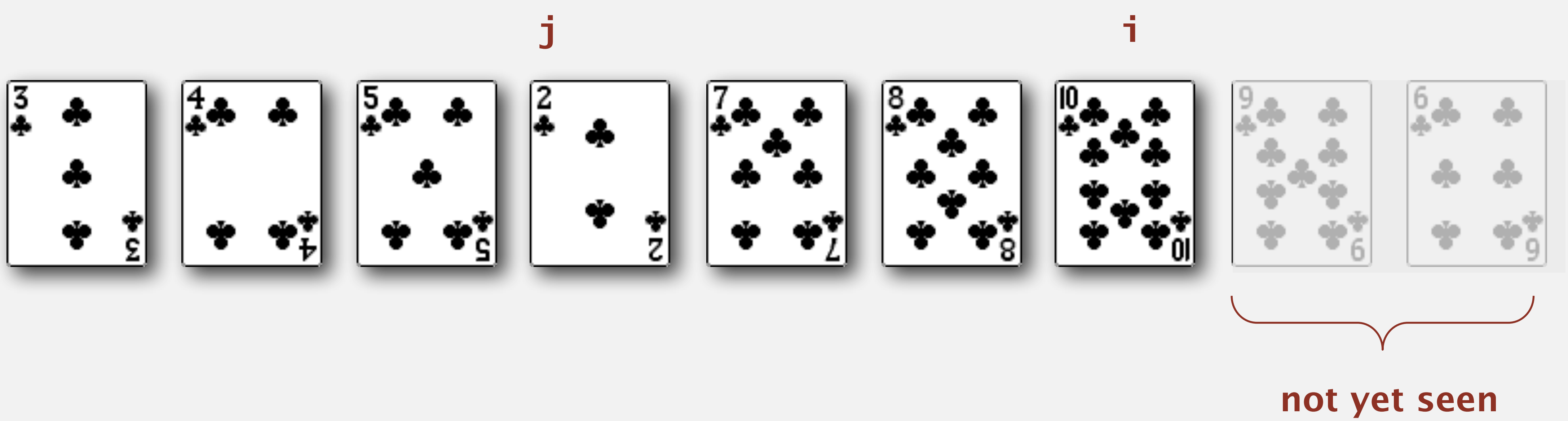
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



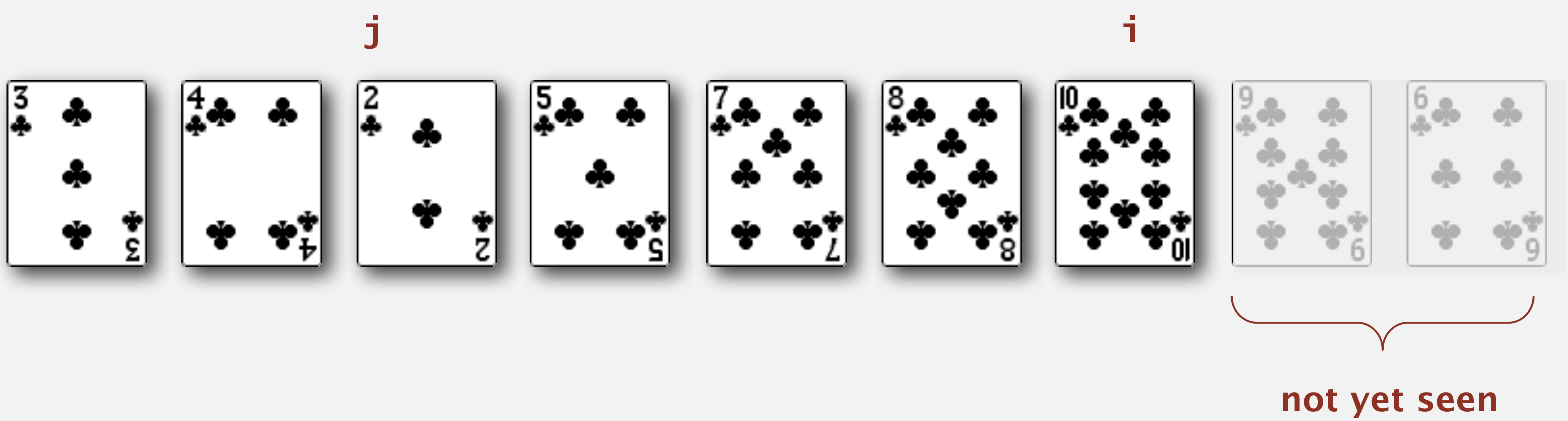
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



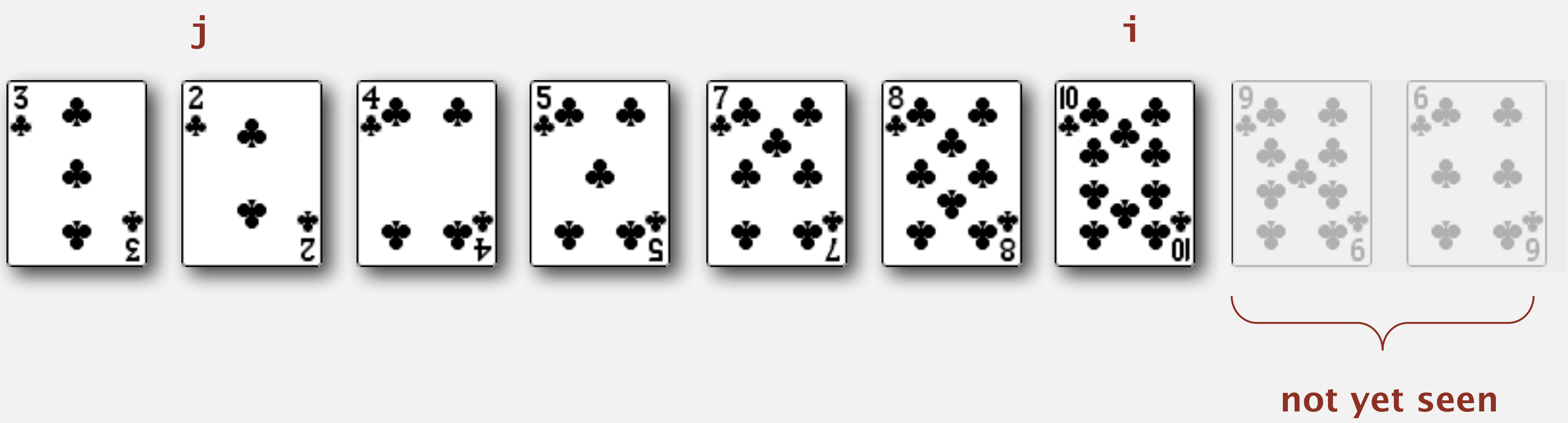
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



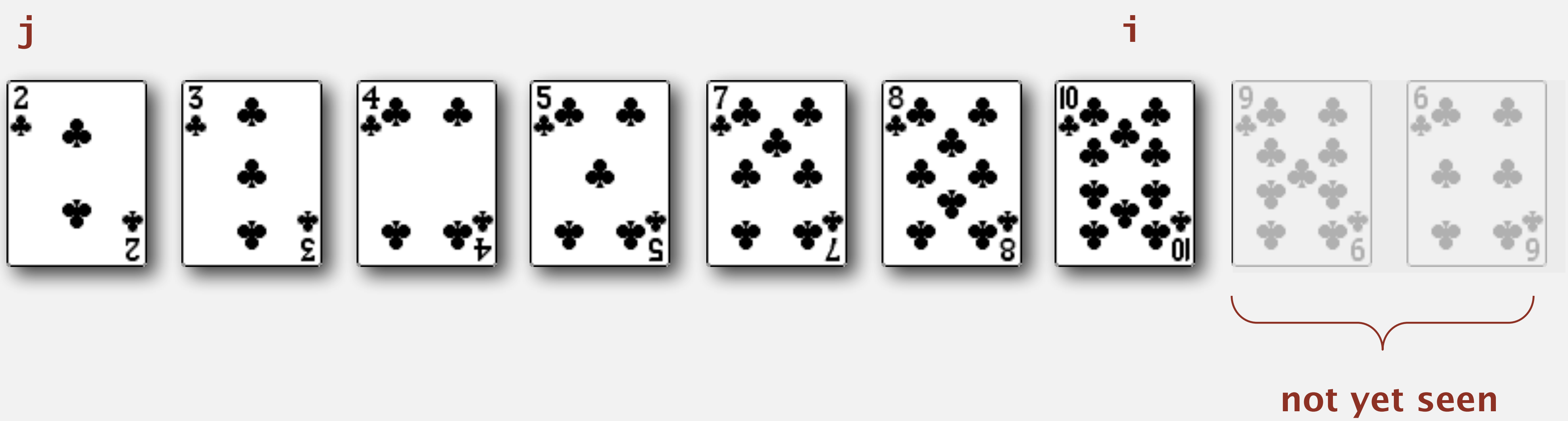
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



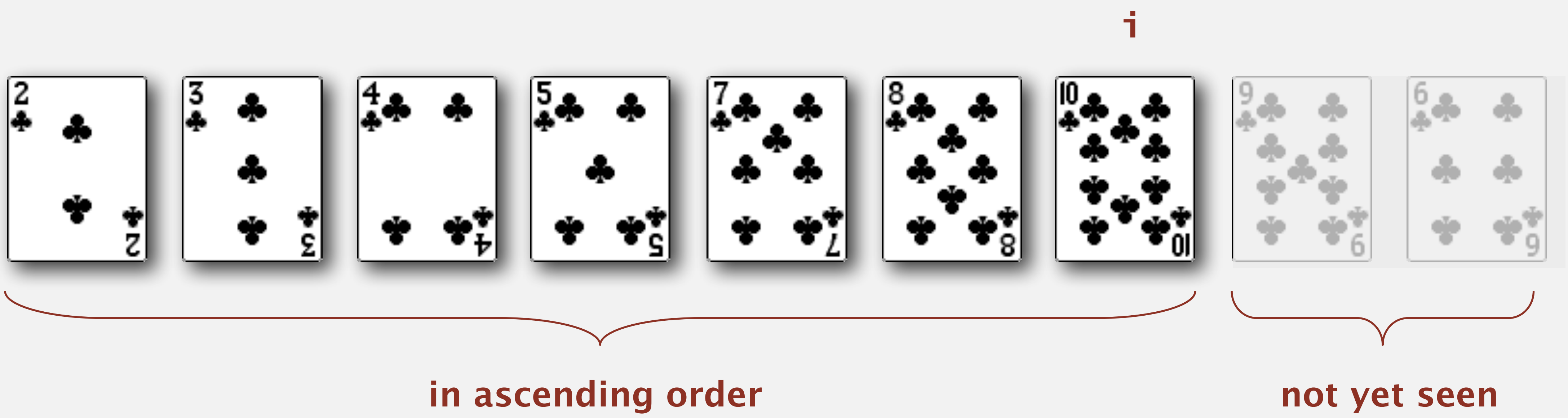
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



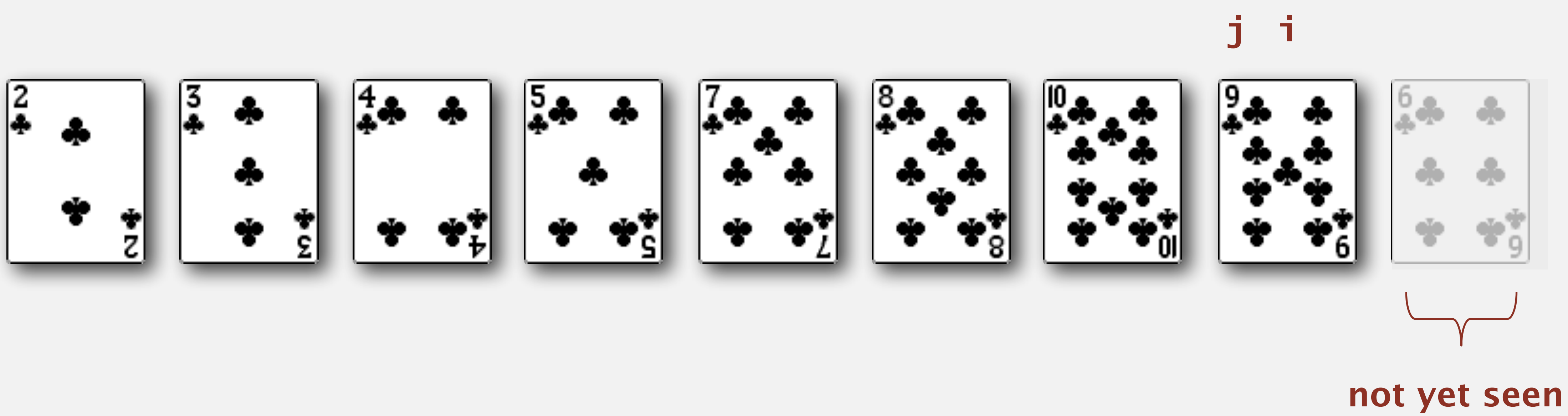
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



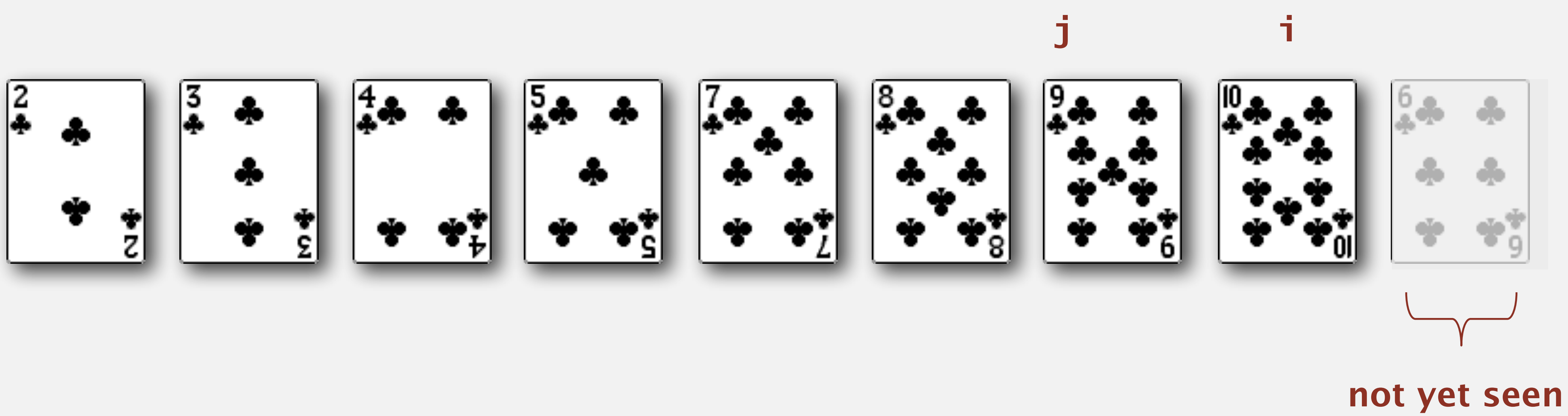
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



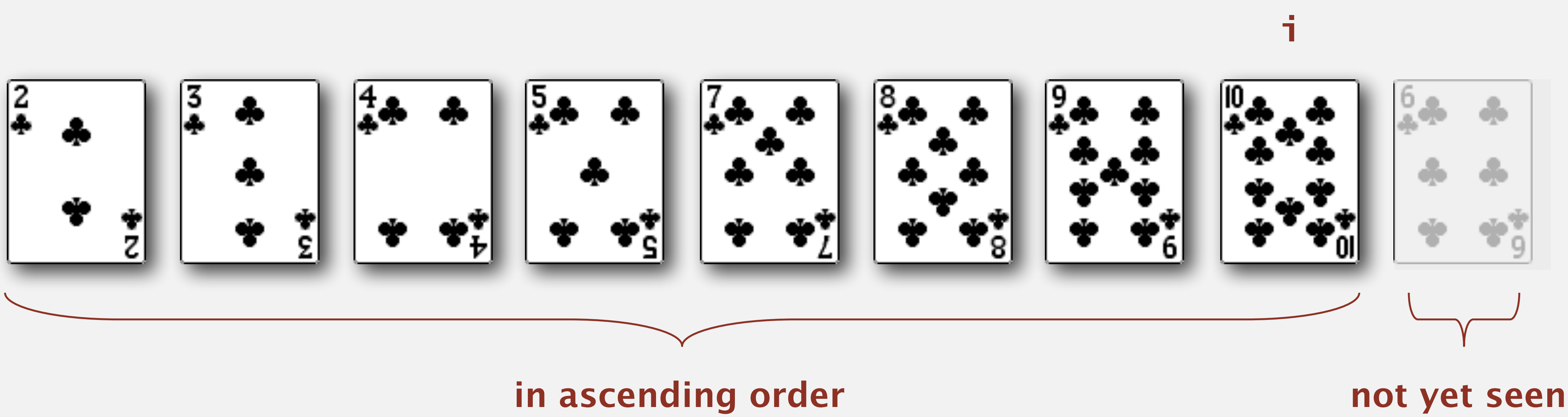
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



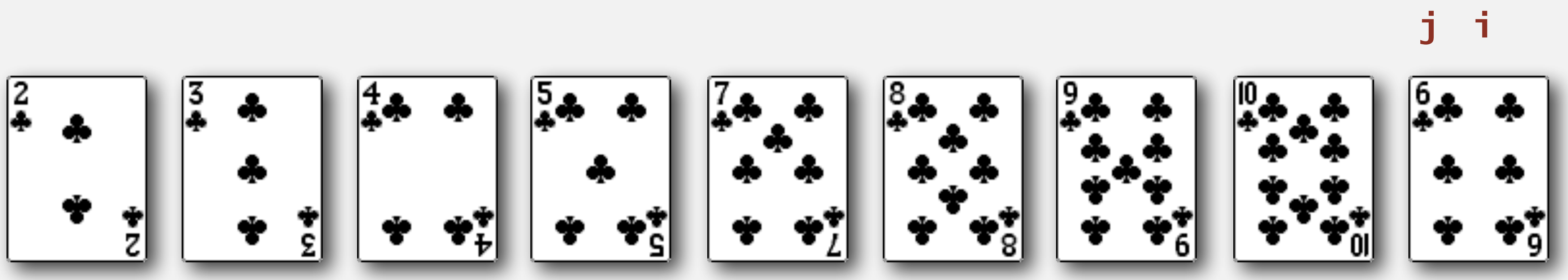
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



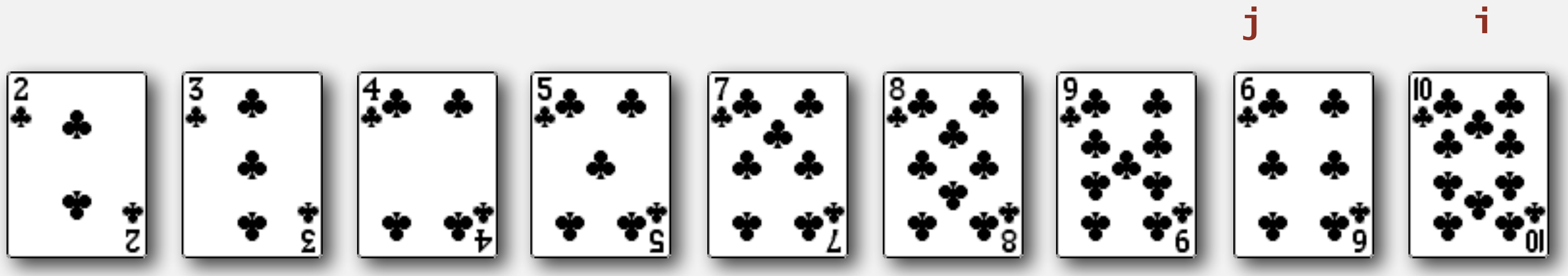
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



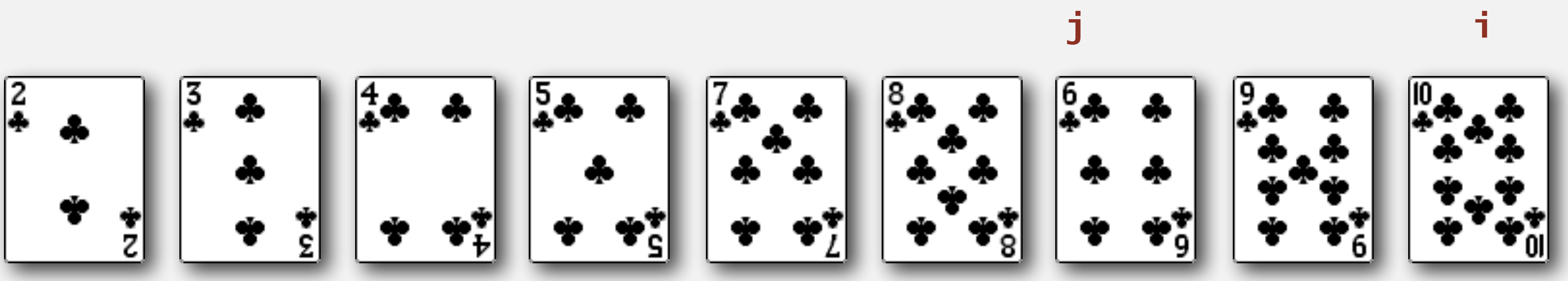
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



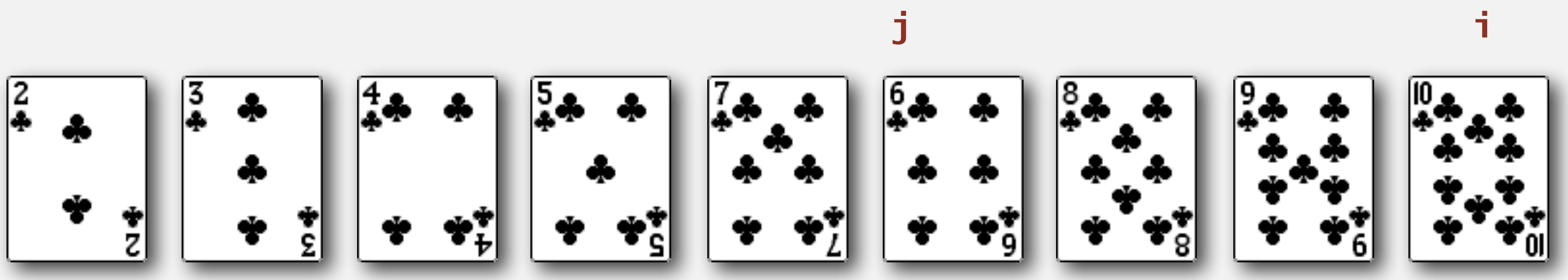
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



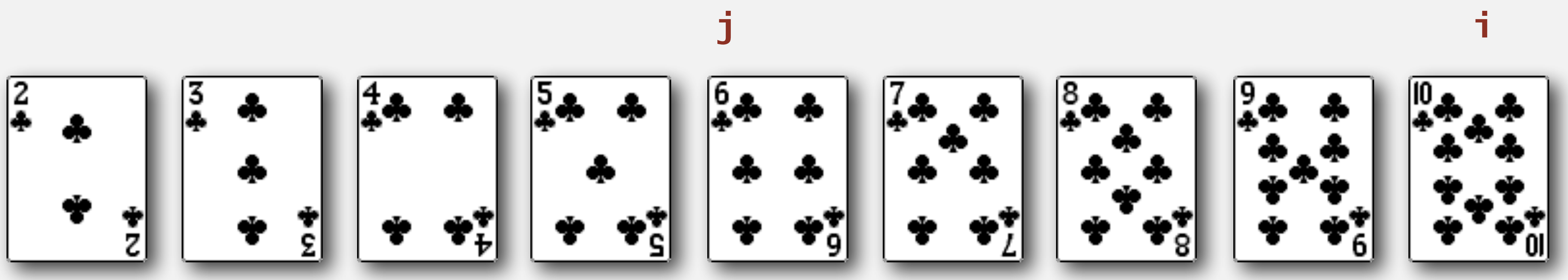
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



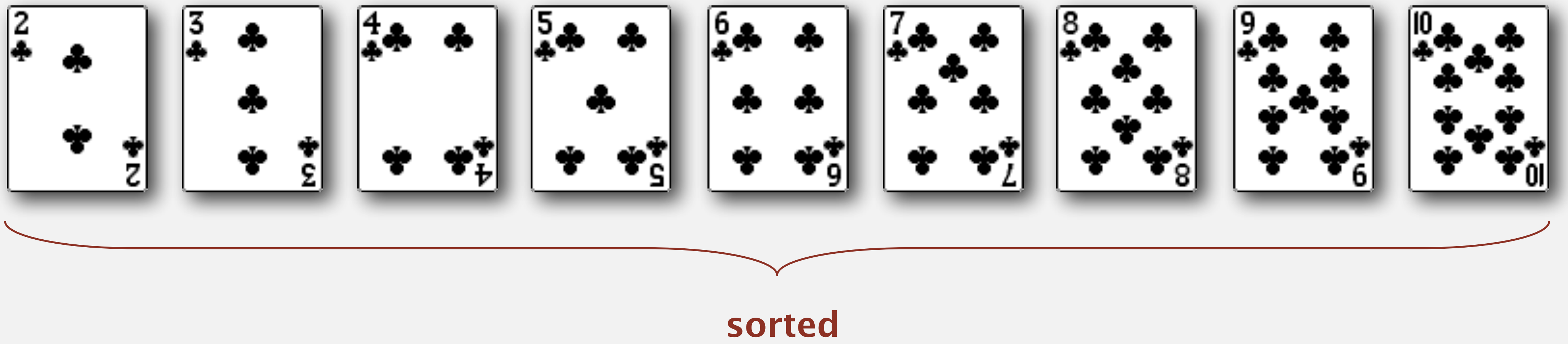
Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.

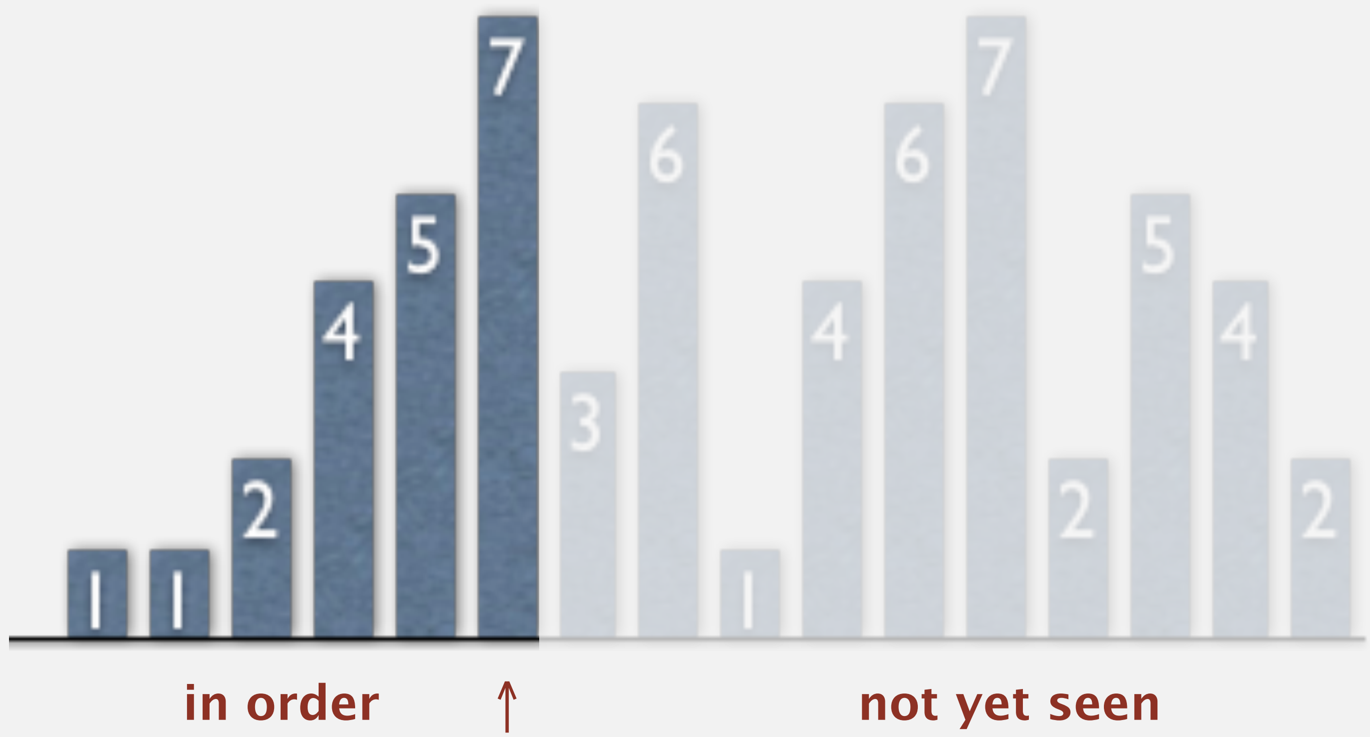


Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

- Entries to the left of ↑ (including ↑) are in ascending order.
- Entries to the right of ↑ have not yet been seen.



Insertion sort: inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Moving from right to left, exchange `a[i]` with each larger entry to its left.

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```

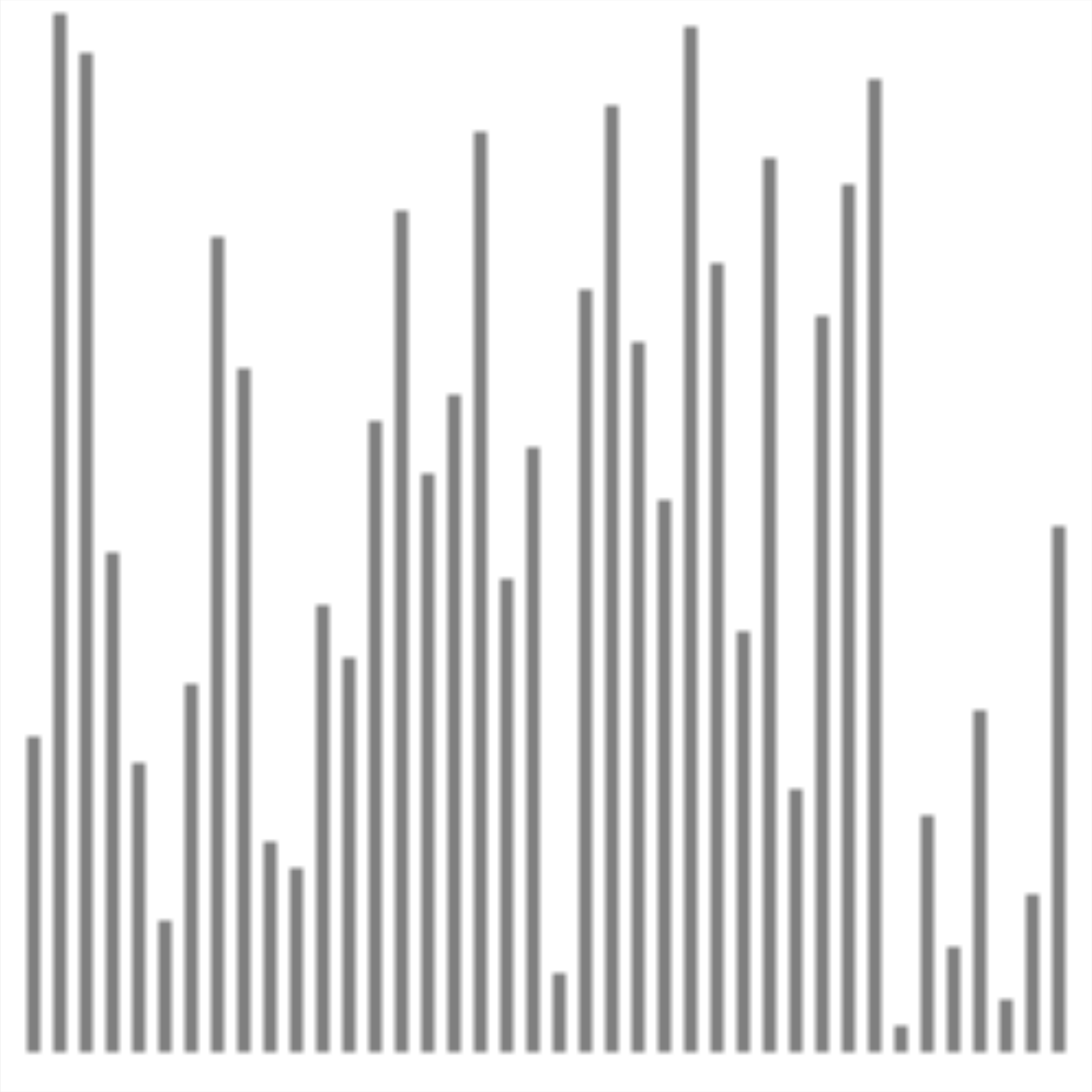
Insertion sort: Java implementation

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Object[] a, int i, int j)
   {  /* as before */  }
}
```

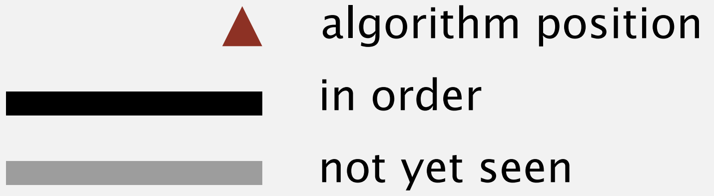
http://algs4.cs.princeton.edu/21elementary/Insertion.java.html

Insertion sort: animation

40 random items



http://www.sorting-algorithms.com/insertion-sort



Insertion sort: mathematical analysis

Proposition. To sort a randomly-ordered array with distinct keys, insertion sort uses $\sim \frac{1}{4} N^2$ compares and $\sim \frac{1}{4} N^2$ exchanges on average.

Pf. Expect each entry to move halfway back.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

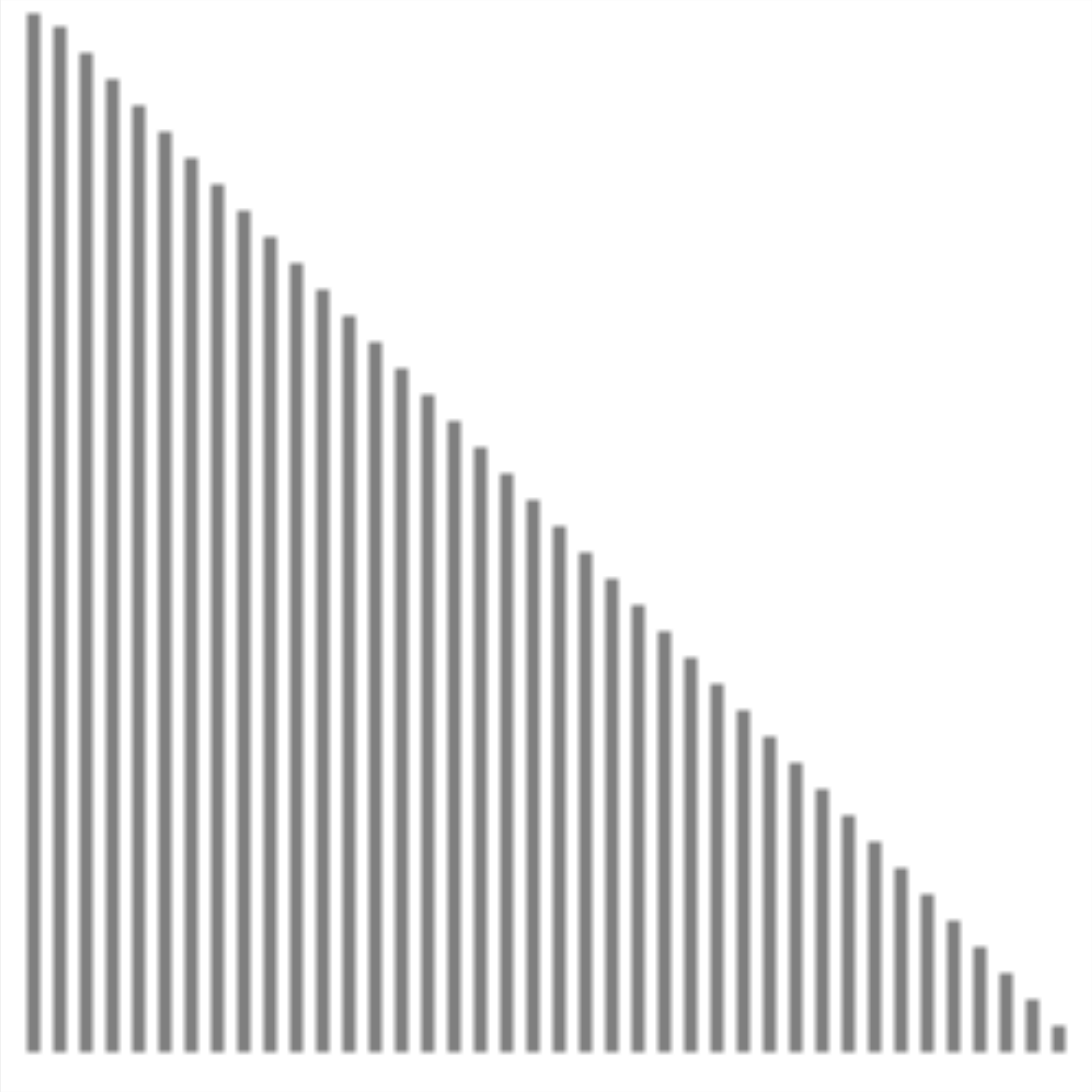
entry in red is `a[j]`

entries in black moved one position right for insertion

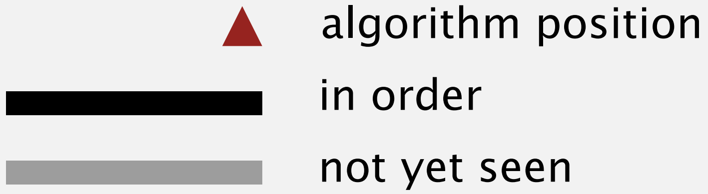
Trace of insertion sort (array contents just after each insertion)

Insertion sort: animation

40 reverse-sorted items



http://www.sorting-algorithms.com/insertion-sort



Insertion sort: analysis

Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

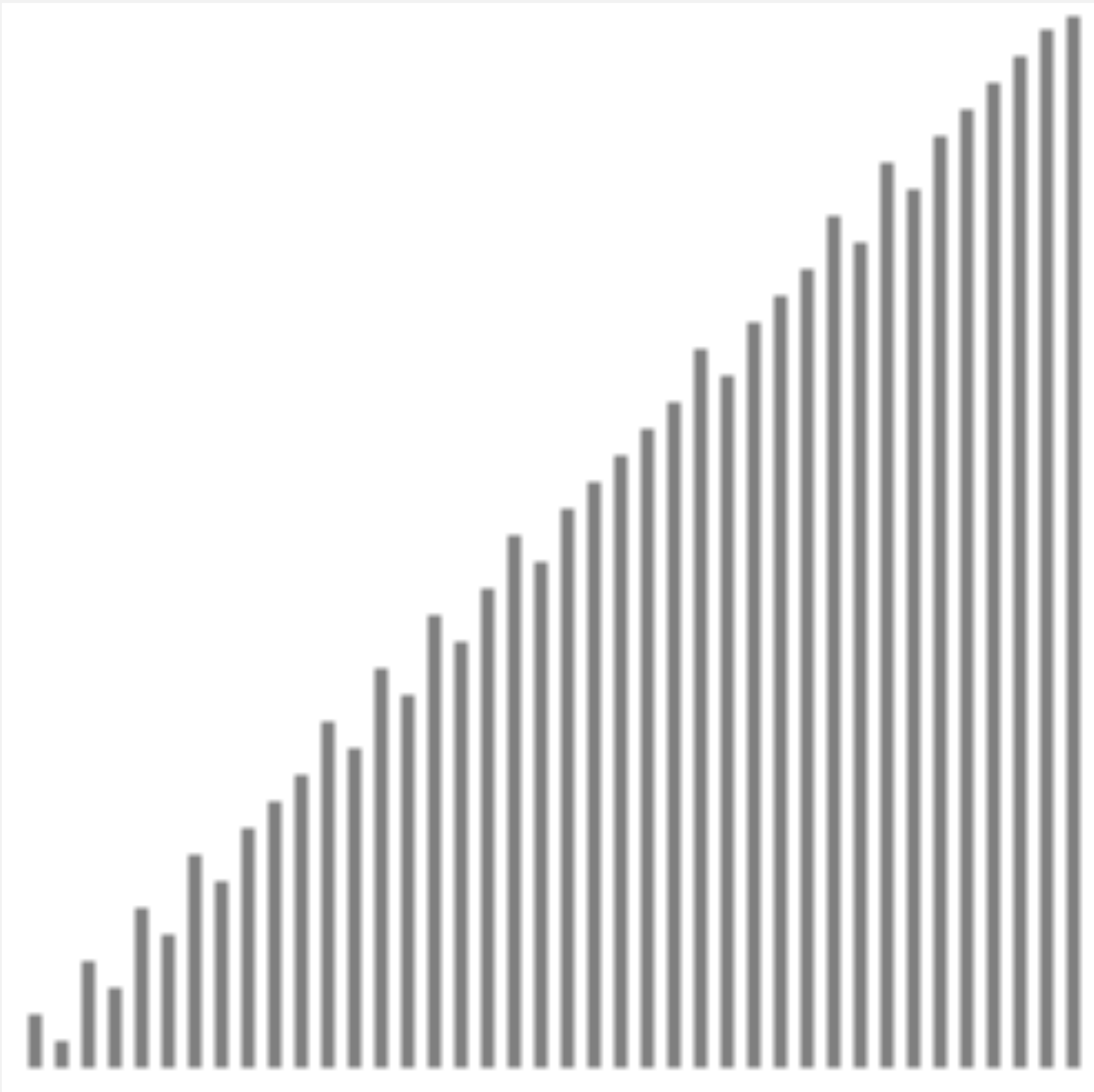
X T S R P O M L F E A

Best case. If the array is in ascending order, insertion sort makes $N-1$ compares and 0 exchanges.

A E E L M O P R S T X

Insertion sort: animation

40 partially-sorted items



▲ algorithm position

in order

not yet seen

http://www.sorting-algorithms.com/insertion-sort

Insertion sort: partially-sorted arrays

Def. An inversion is a pair of keys that are out of order.

```
A E E L M O T R X P S
```

T-R T-P T-S R-P X-P X-S

(6 inversions)

Def. An array is partially sorted if the number of inversions is $\le c\,N$.

- Ex 1. A sorted array has 0 inversions.
- Ex 2. A subarray of size 10 appended to a sorted subarray of size N.

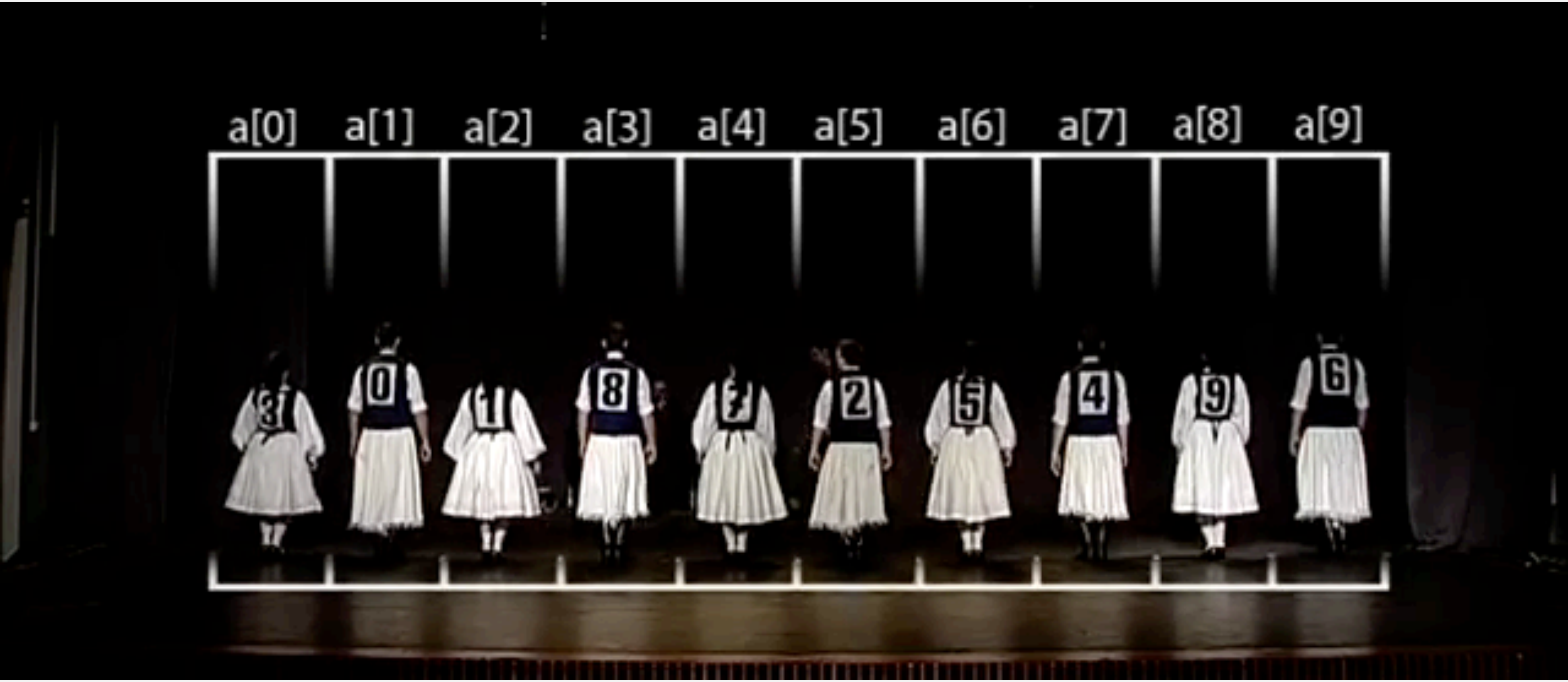
Proposition. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

number of compares ≤ exchanges + (N – 1)

Insertion sort demo

- In iteration `i`, swap `a[i]` with each larger entry to its left.



https://www.youtube.com/watch?v=ROalU379l3U