BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ELEMENTARY SORTING ALGORITHMS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort

Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort

Sorting problem

Ex. Student records in a university.

	Chen	3	A	991-878-4944	308 Blair
	Rohde	2	A	232-343-5555	343 Forbes
	Gazsi	4	B	766-093-9873	101 Brown
item →	Furia	1	A	766-093-9873	101 Brown
	Kanaga	3	B	898-122-9643	22 Brown
	Andrews	3	A	664-480-0023	097 Little
key →	Battle	4	C	874-088-1212	121 Whitman

Sort. Rearrange array of N items into ascending order.

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Sample sort client

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial, but 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

Goal. Sort any type of data.

Ex 2. Sort strings from file in alphabetical order.

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = In.readStrings(args[0]);
      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
```

Sample sort client

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
```

Callbacks

Goal. Sort any type of data.

Q. How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

Callback = reference to executable code.

- Client passes array of objects to `sort()` function.
- The `sort()` function calls back object's `compareTo()` method as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: class-type functors.
- C#: delegates.
- Python, Perl, ML, Javascript: first-class functions.

Callbacks: roadmap

client

```
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());
   }
}
```

object implementation

```
public class File
implements Comparable<File>
{
   ...
   public int compareTo(File b)
   {
      ...
      return -1;
      ...
      return +1;
      ...
      return 0;
   }
}
```

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
   public int compareTo(Item that);
}
```

sort implementation

```
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 (a[j].compareTo(a[j-1]) < 0)
            exch(a, j, j-1);
         else break;
}
```

key point: no dependence on `File` data type

Total order

A total order is a binary relation $\leq$ that satisfies

- Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Ex.

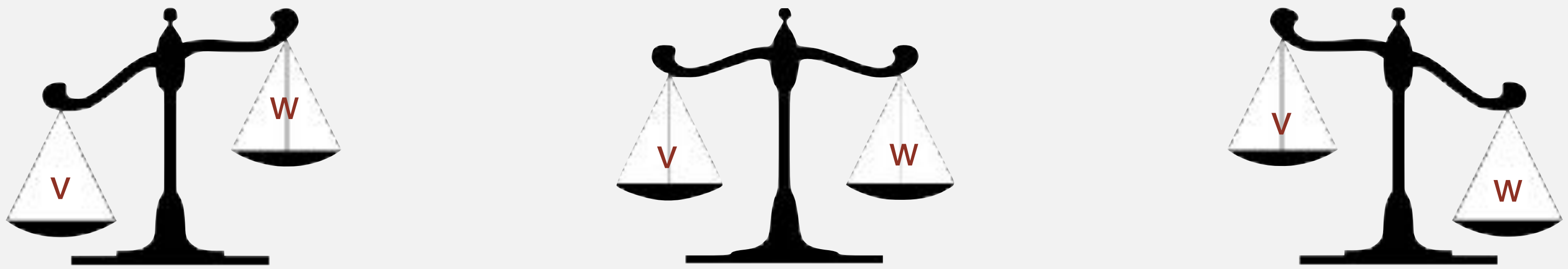
- Standard order for natural and real numbers.
- Alphabetical order for strings.
- Chronological order for dates.
- ...

an intransitive relation

Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

- Is a total order.
- Returns a negative integer, zero, or positive integer if v is less than, equal to, or greater than w, respectively.
- Throws an exception if incompatible types (or either is `null`).



less than (return −1)   equal to (return 0)   greater than (return +1)

Built-in comparable types. `Integer`, `Double`, `String`, `Date`, `File`, ...

User-defined comparable types. Implement the `Comparable` interface.

Implementing the Comparable interface

Date data type. Simplified version of `java.util.Date`.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

only compare dates to other dates

Two useful sorting abstractions

Helper functions. Refer to data 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`.

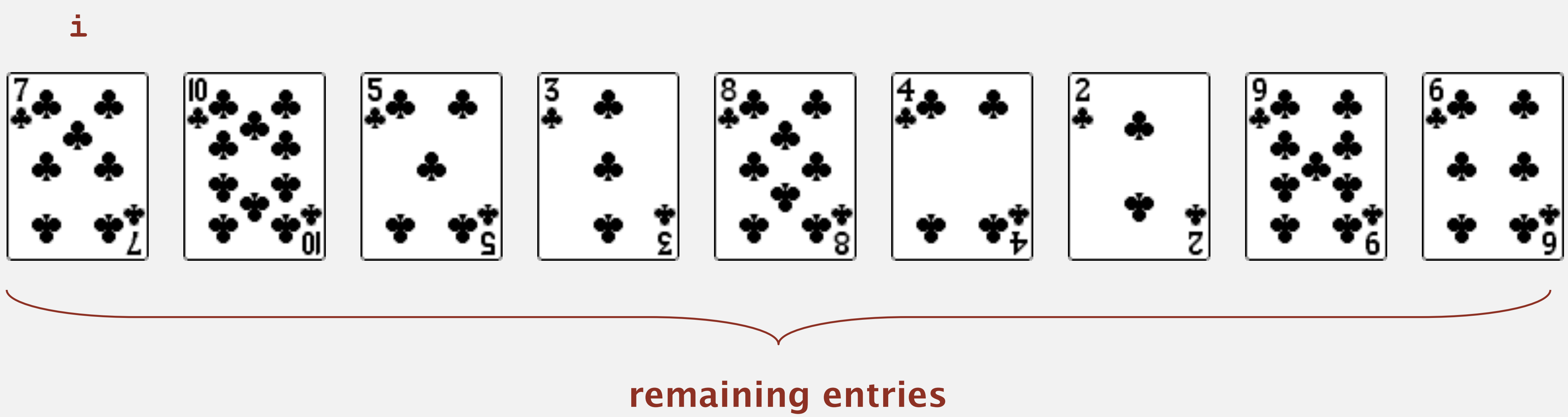
```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort

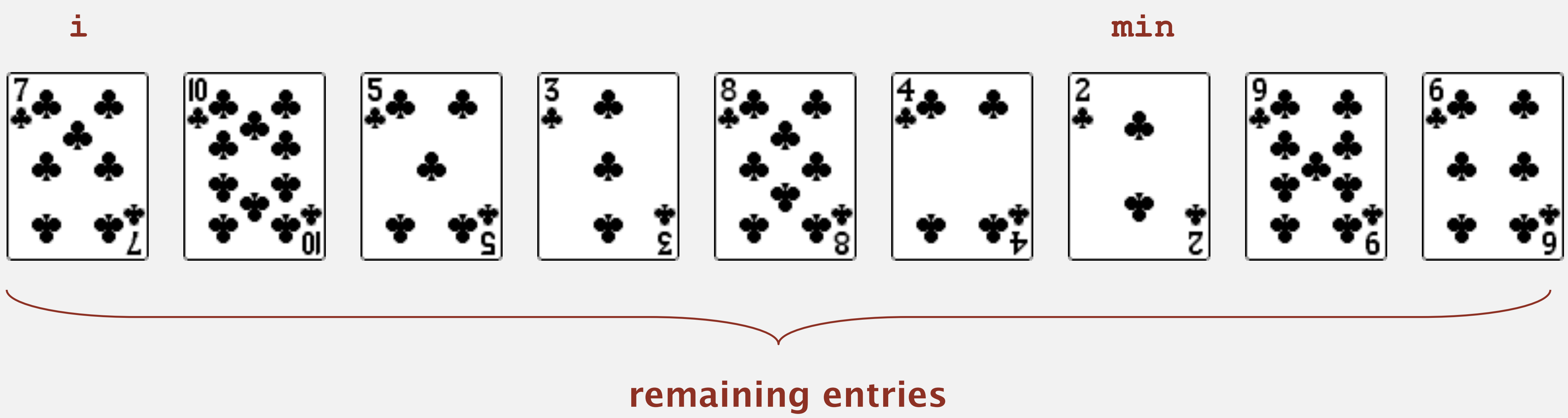
Selection sort

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



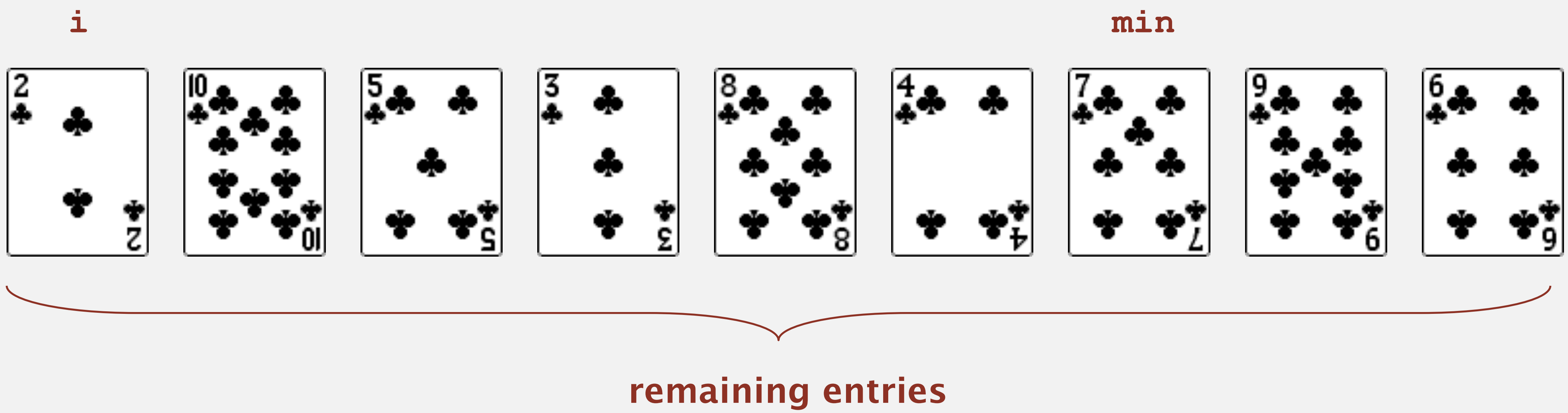
Selection sort

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



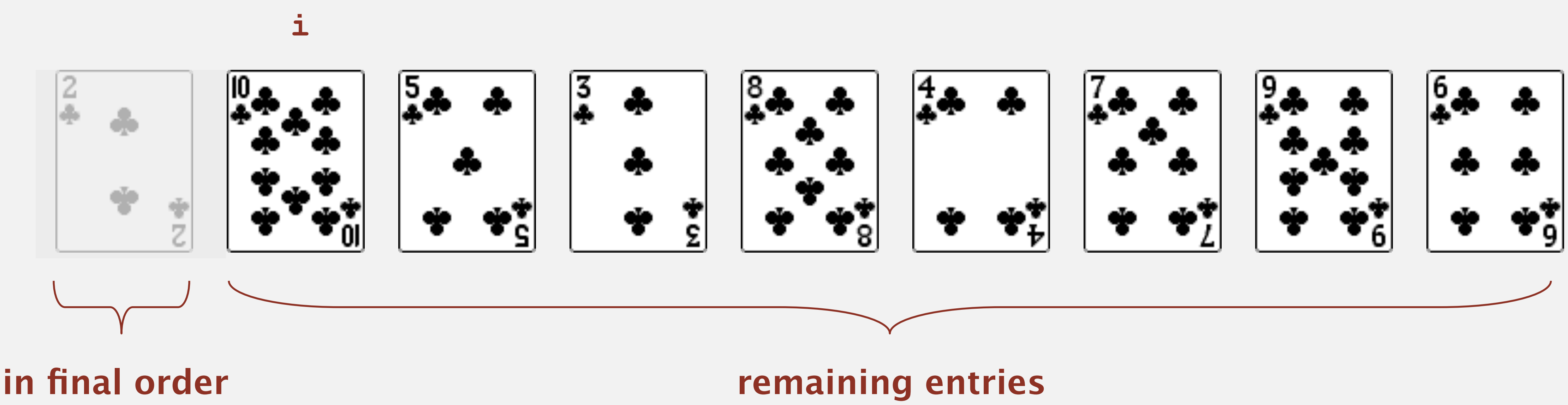
Selection sort

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



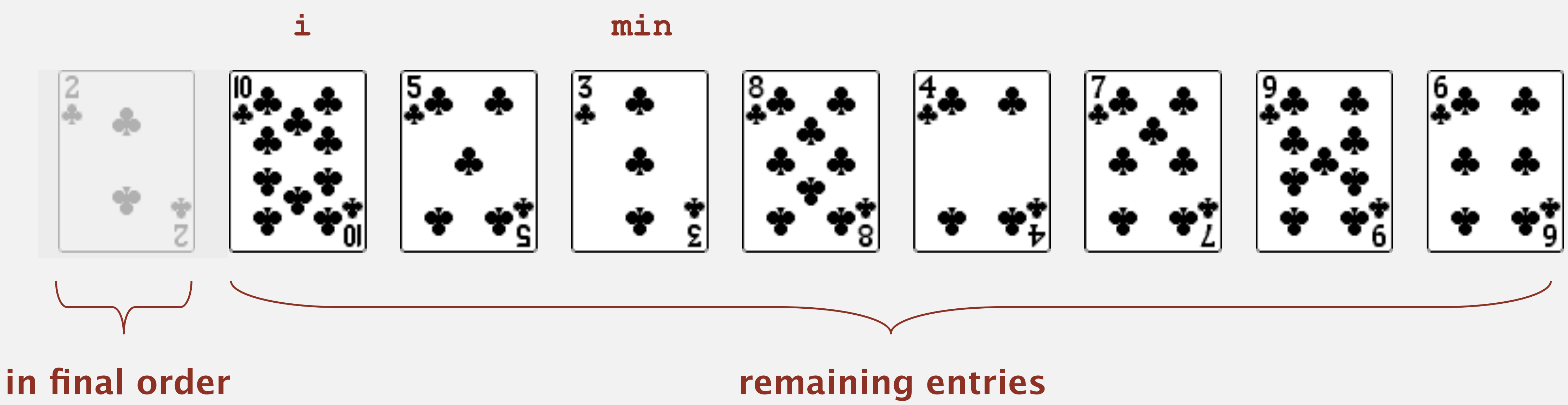
Selection sort

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



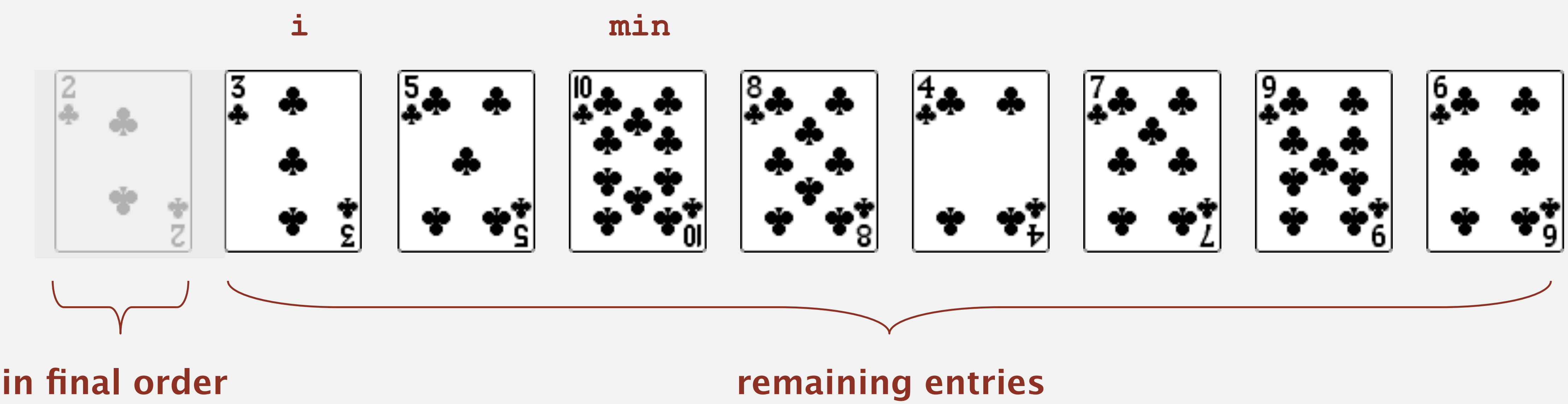
Selection sort

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



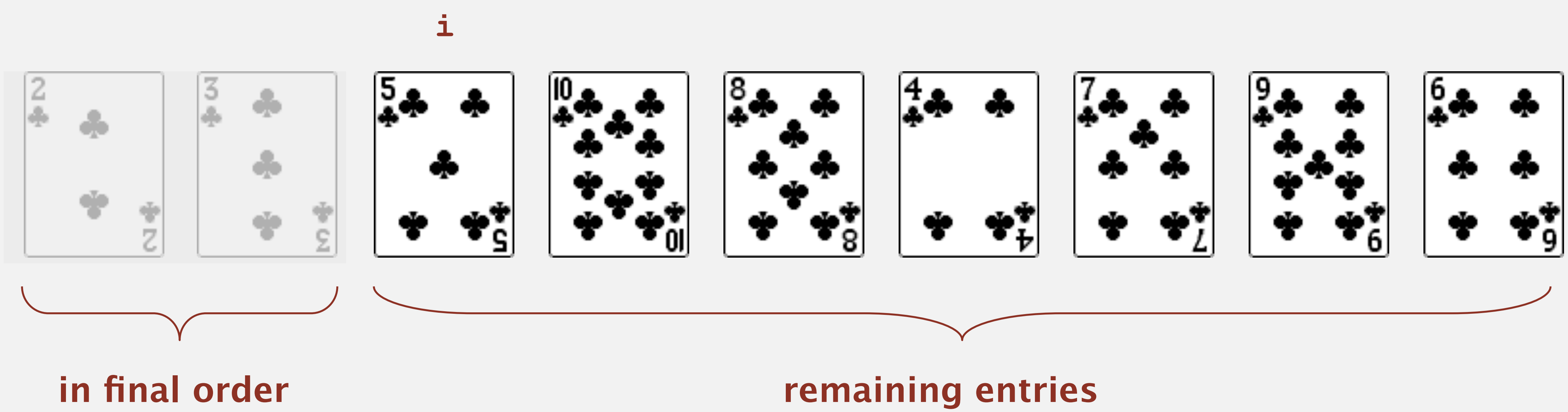
Selection sort

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



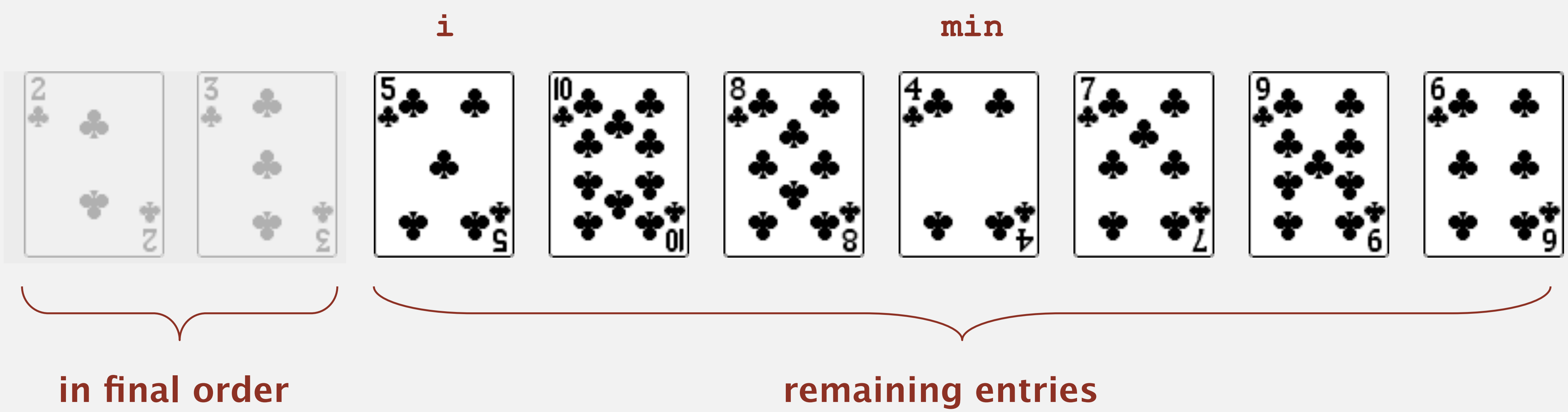
Selection sort

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



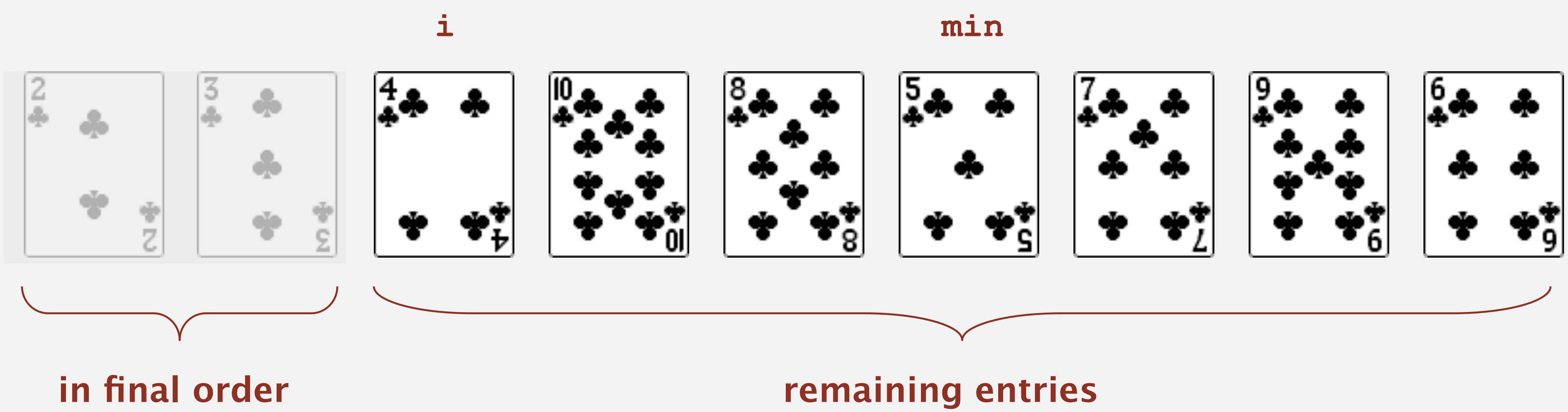
Selection sort

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



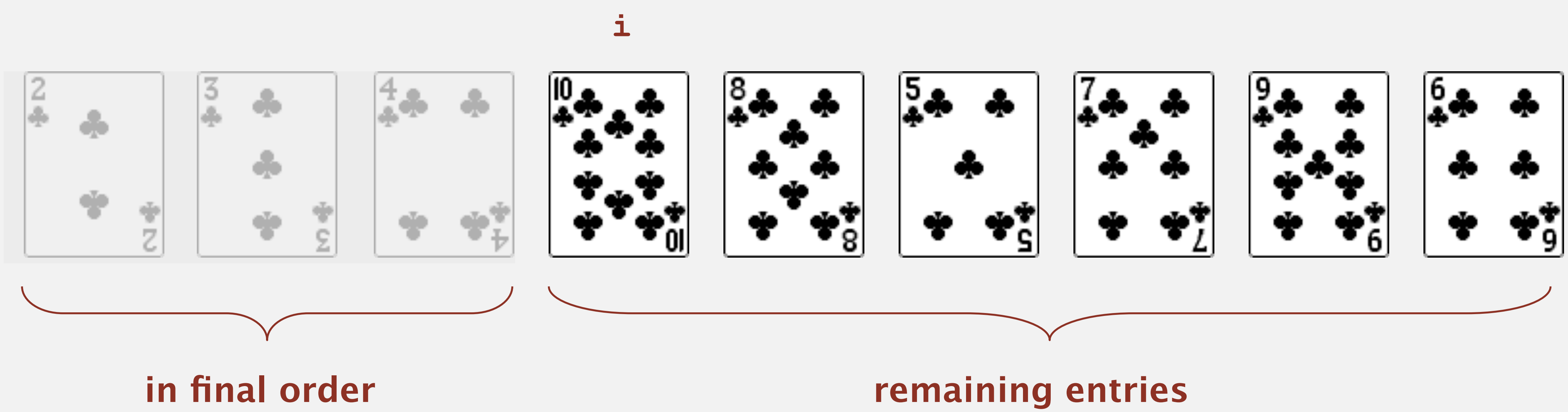
Selection sort

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



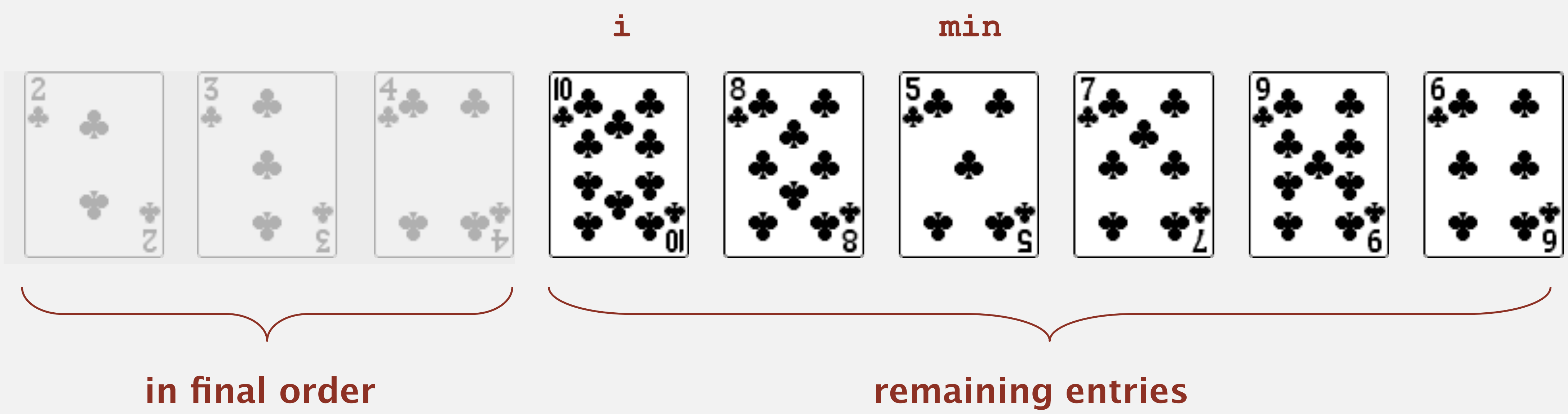
Selection sort

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



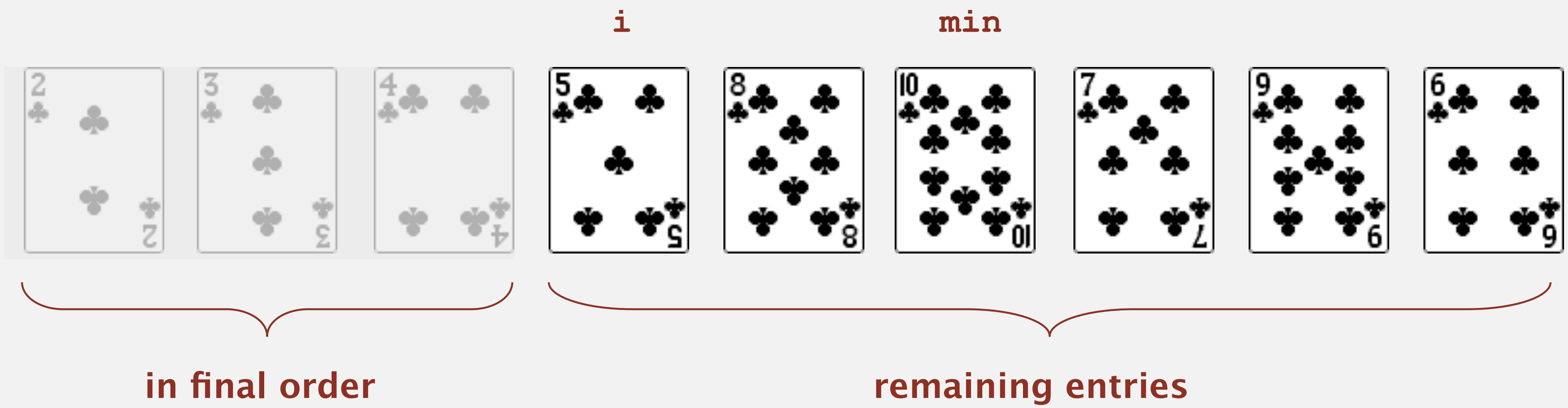
Selection sort

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



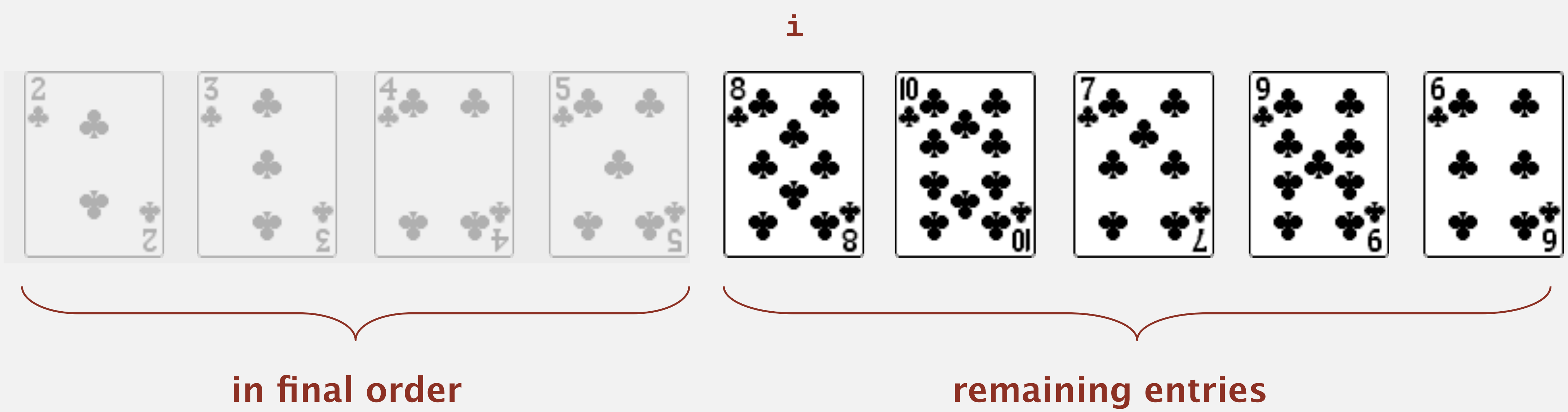
Selection sort

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



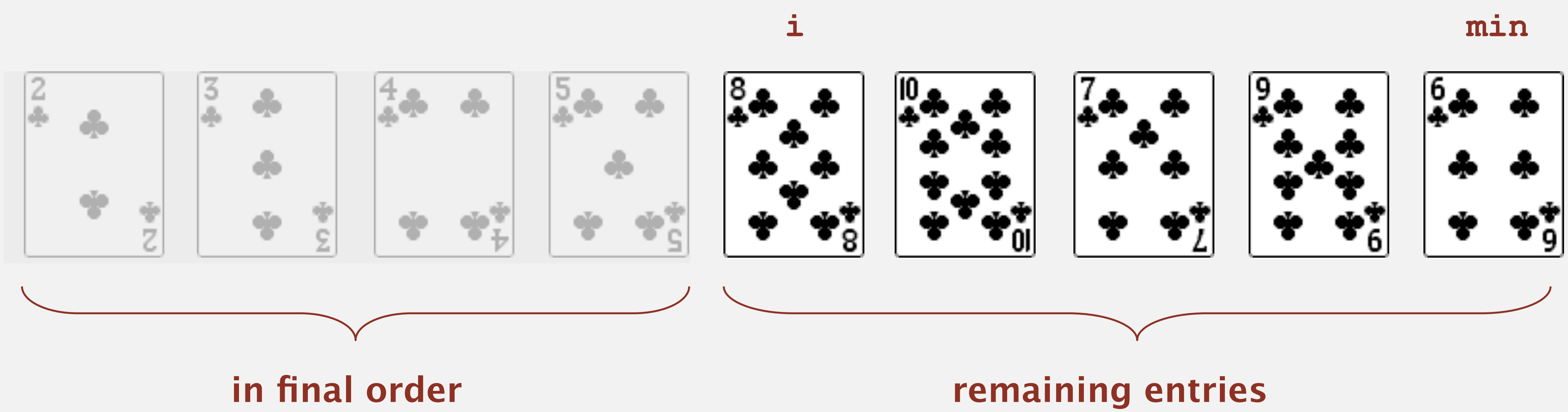
Selection sort

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



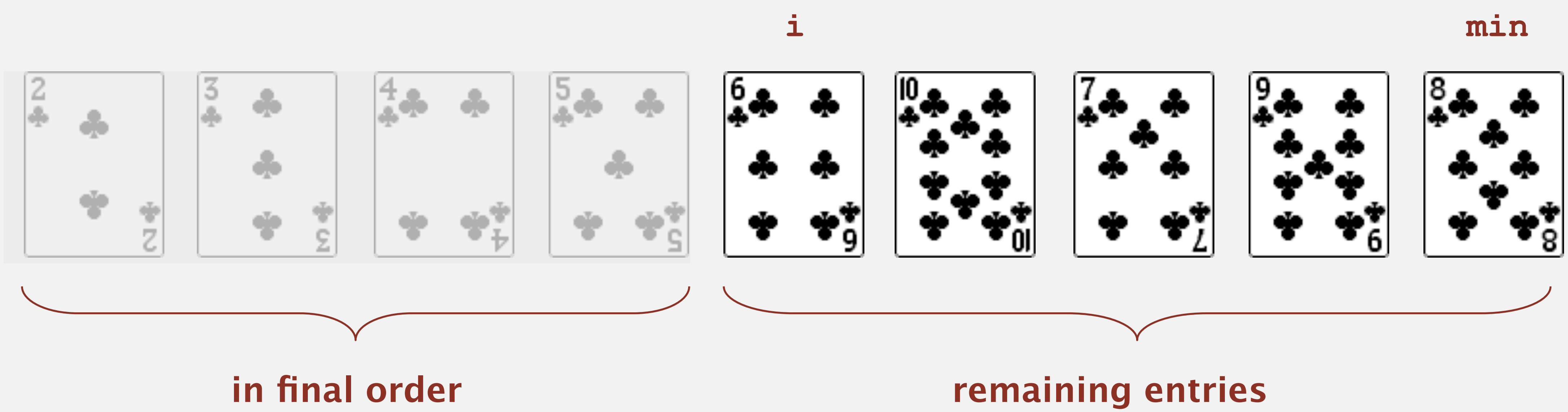
Selection sort

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



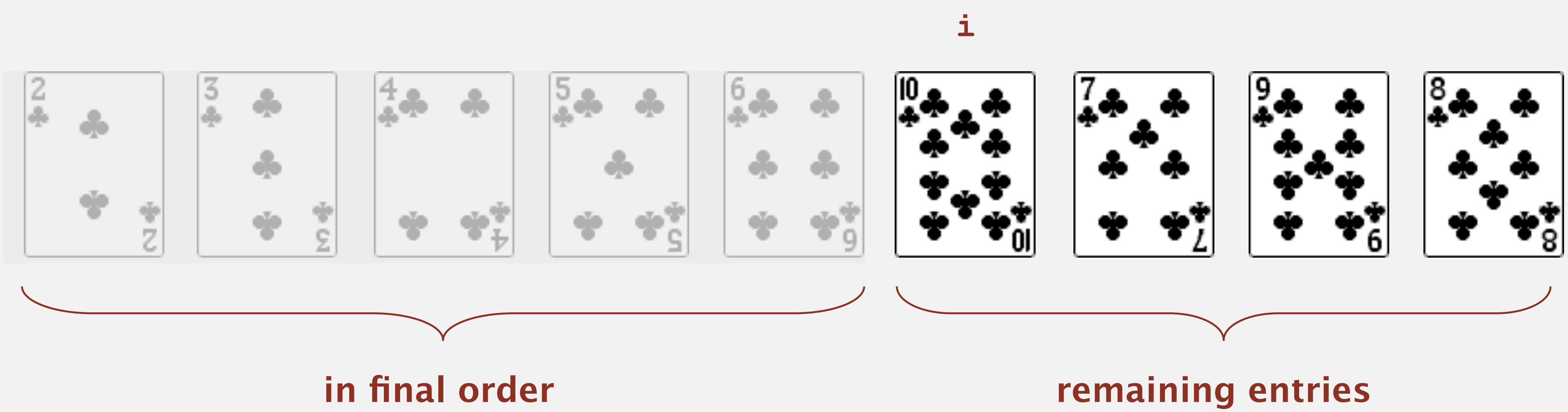
Selection sort

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



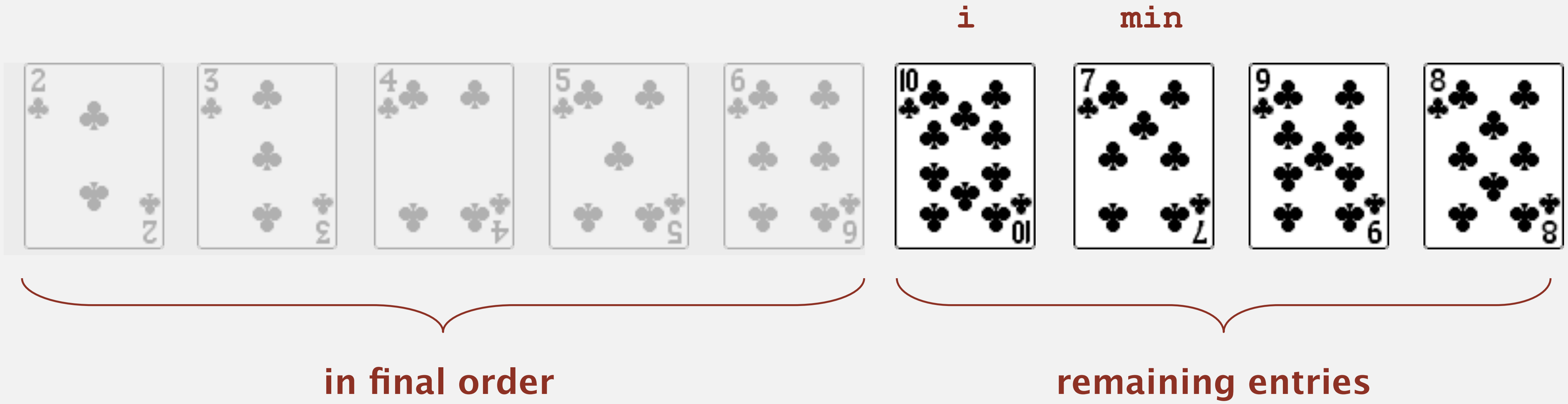
Selection sort

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



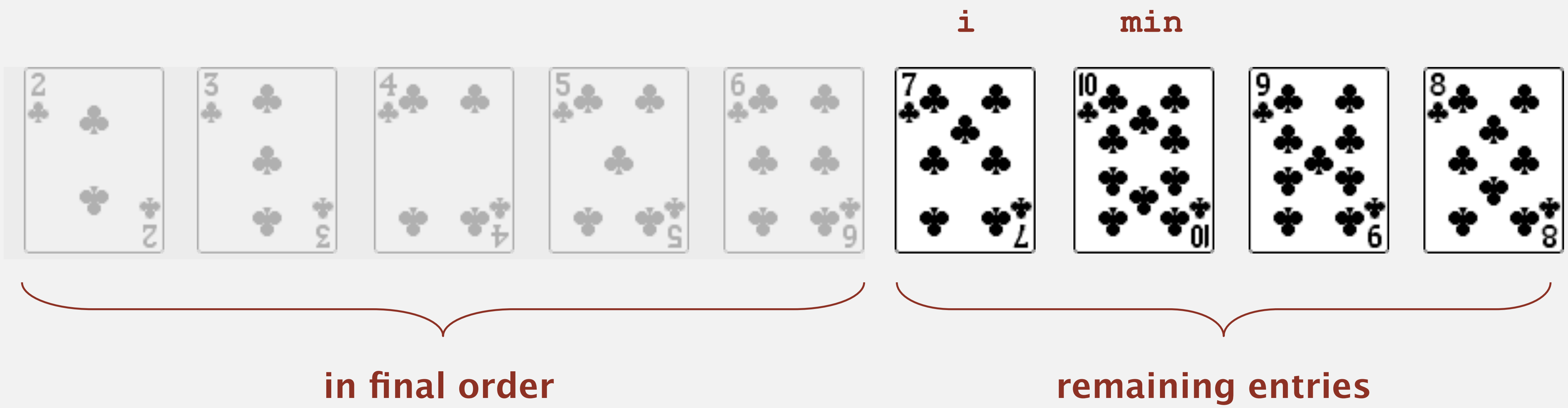
Selection sort

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



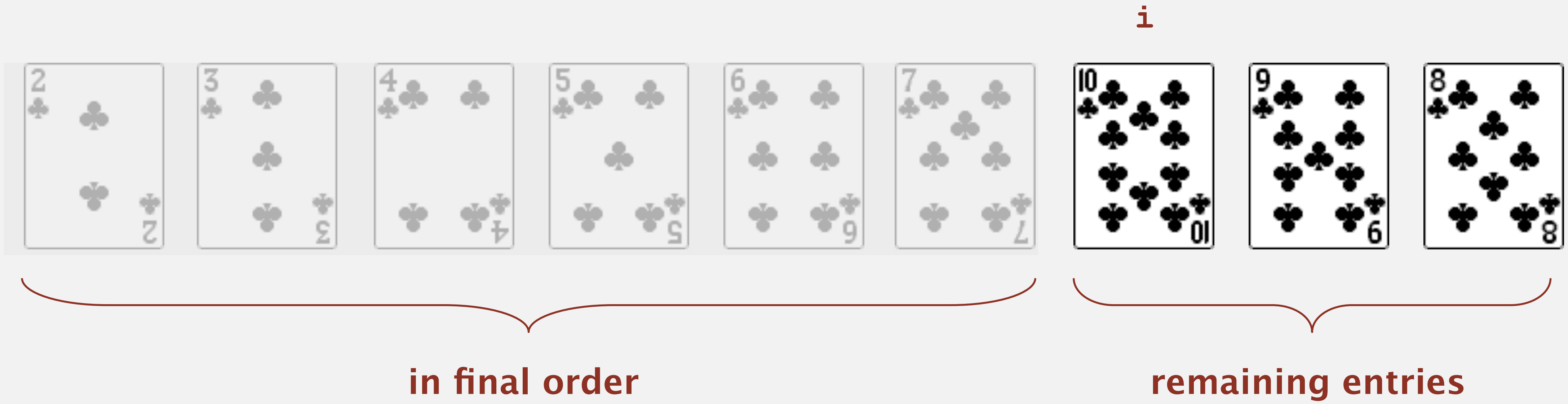
Selection sort

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



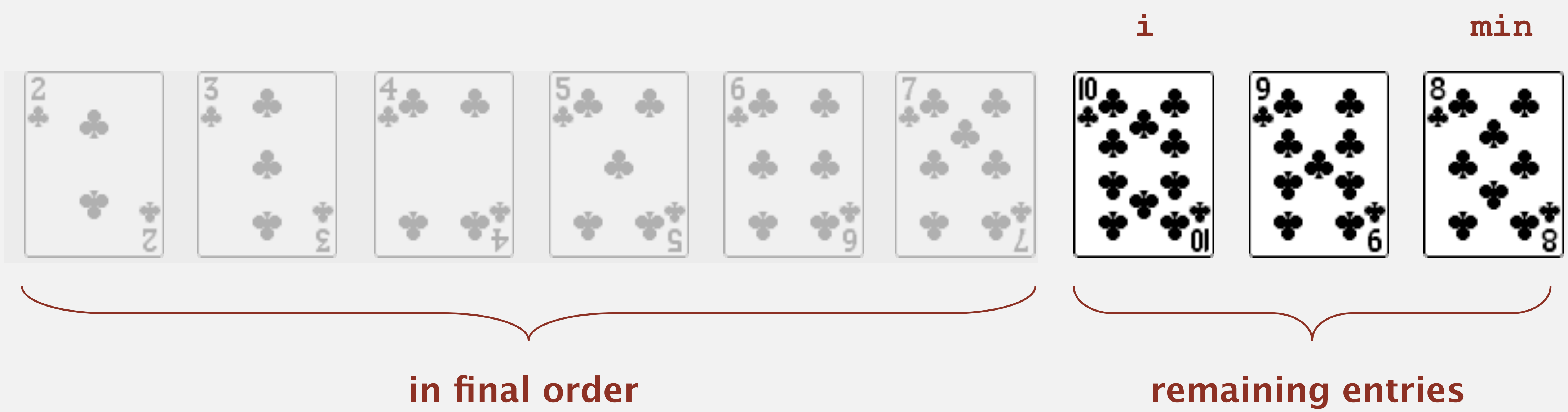
Selection sort

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



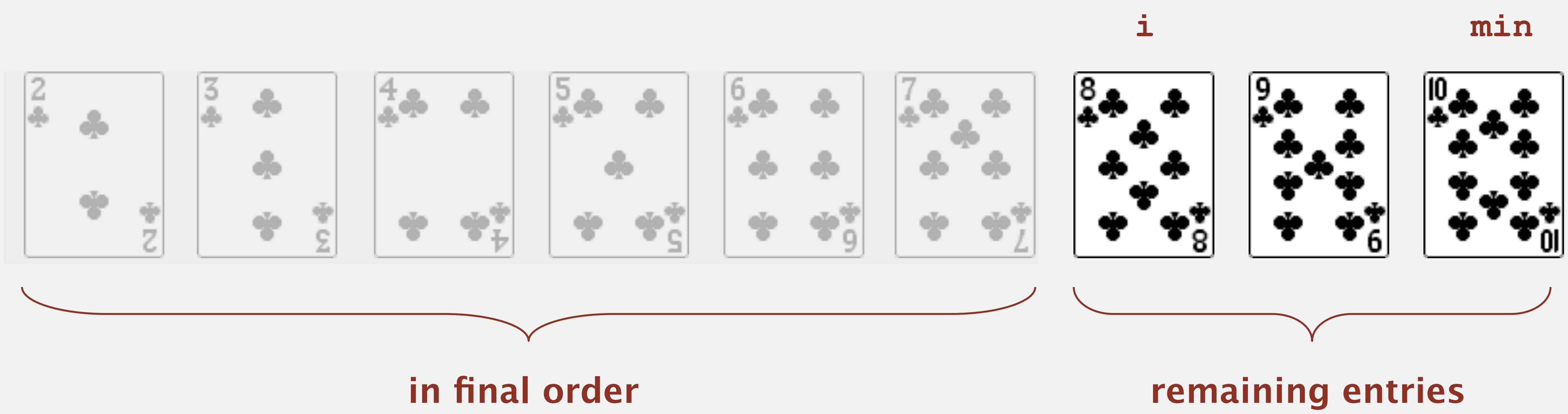
Selection sort

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



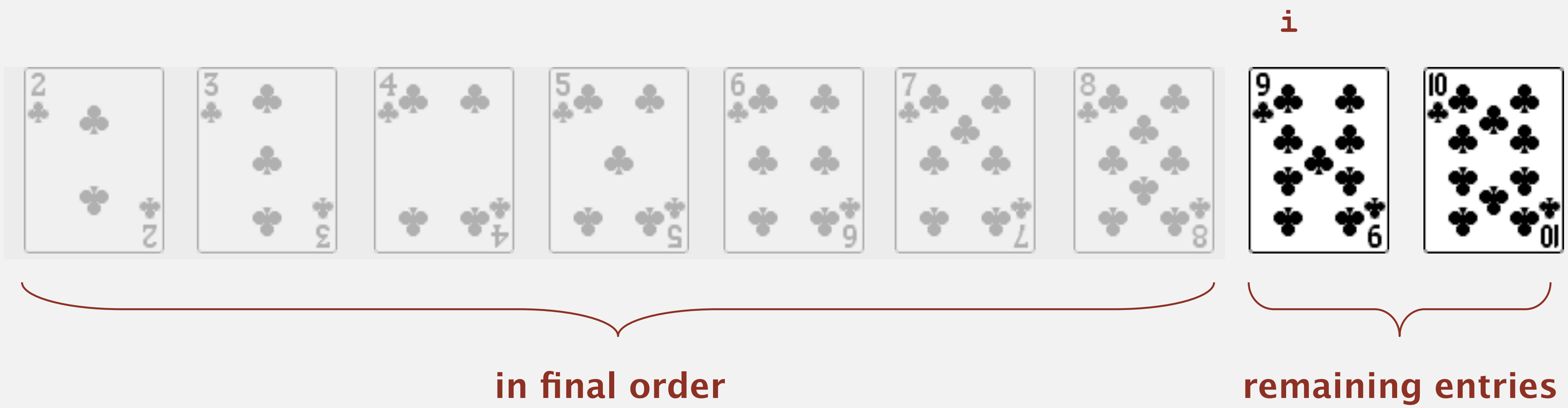
Selection sort

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



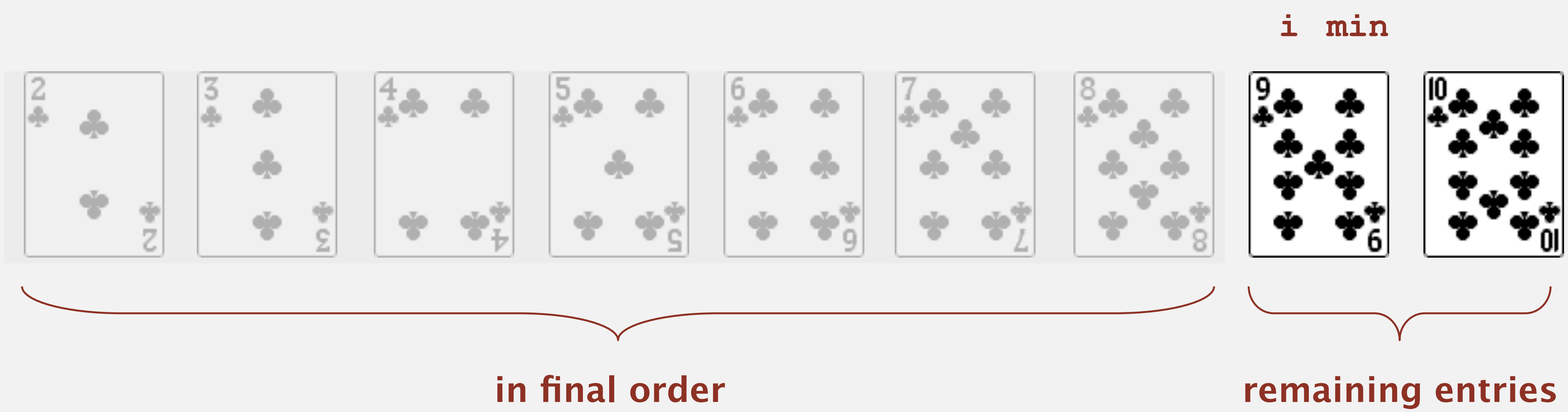
Selection sort

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



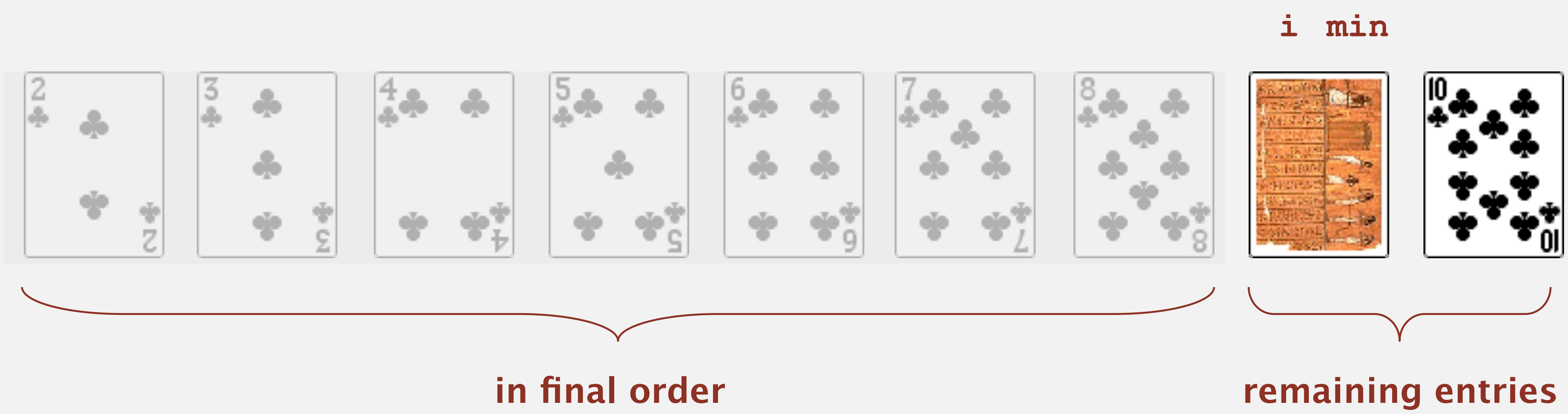
Selection sort

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



Selection sort

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



Selection sort

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

Selection sort

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

Selection sort

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

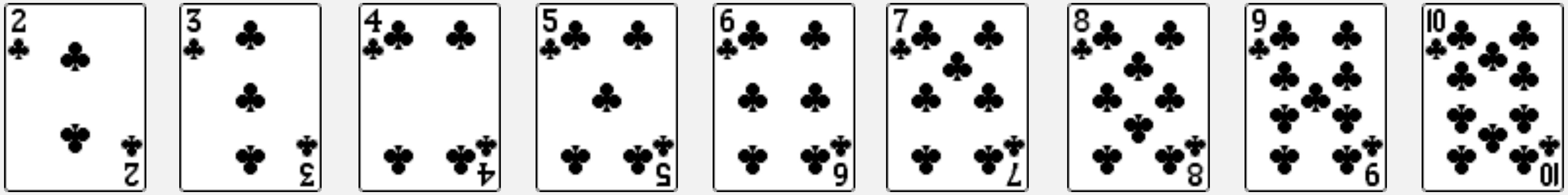
Selection sort

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

in final order

Selection sort

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



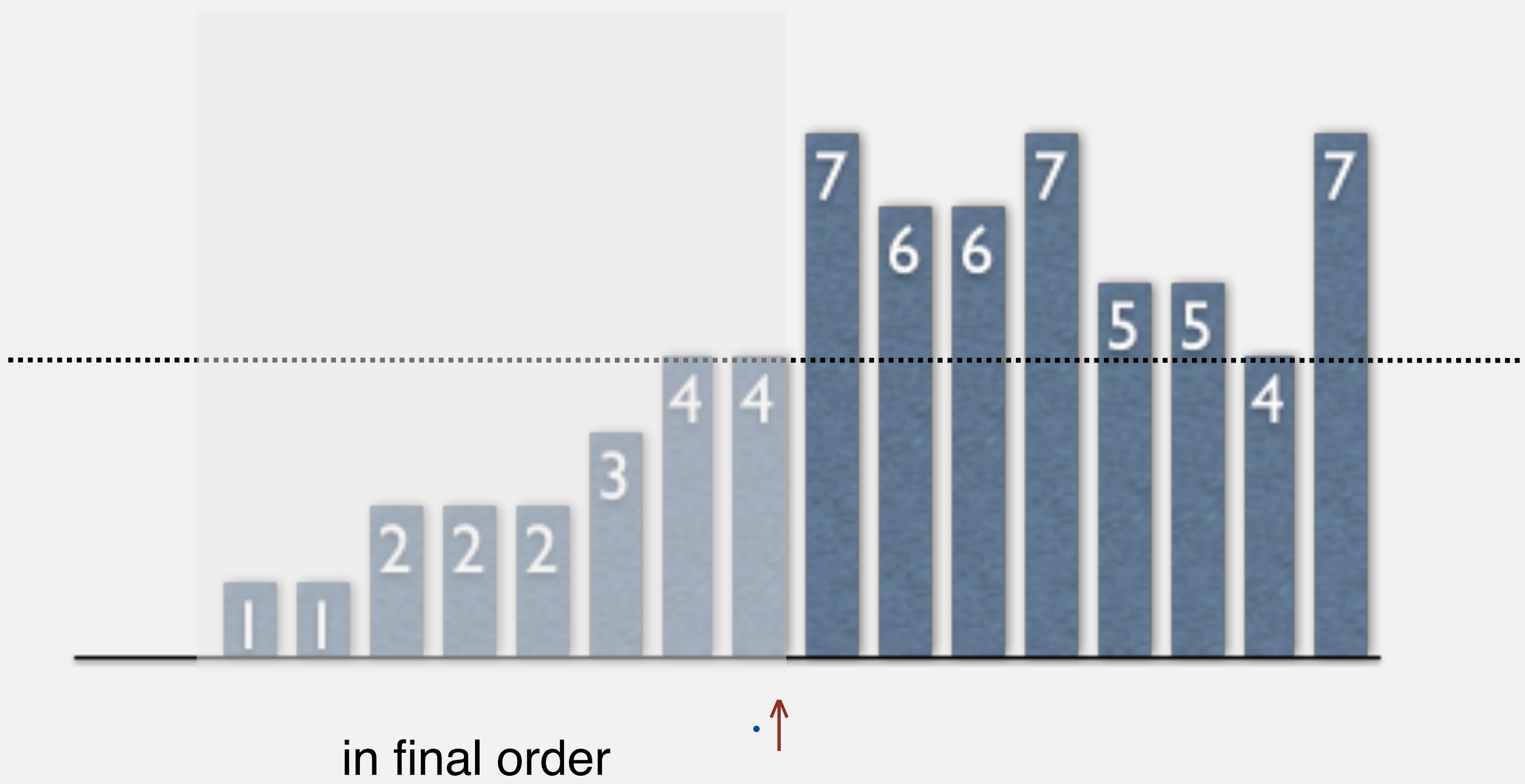
sorted

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.

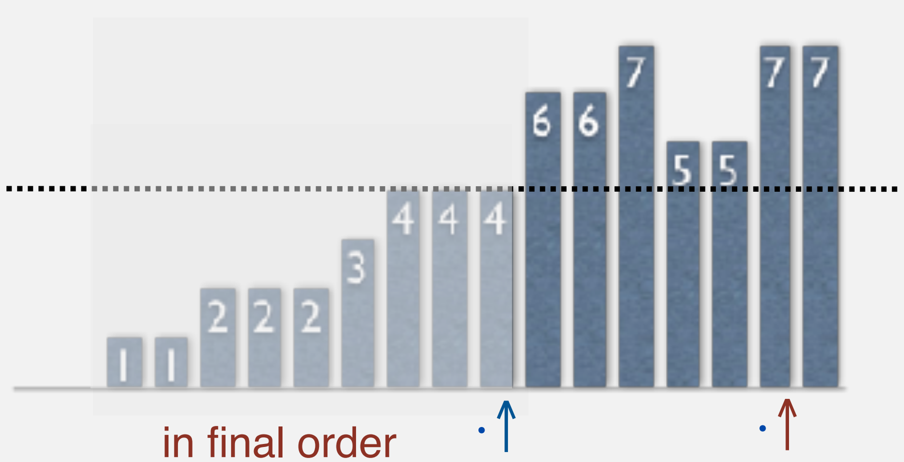
```
i++;
```

- 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);
```



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)
   {  /* as before */   }

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

Poll time!

What is the best case complexity of Selection Sort in terms of number of comparisons?

a. ~N

b. ~N logN

c. ~N^2/2

d. ~N^2

Visit the following link to submit your answers:

https://forms.gle/oEj7U1dtuVXPFzvd6

SCAN ME

Selection sort: mathematical analysis

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

		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

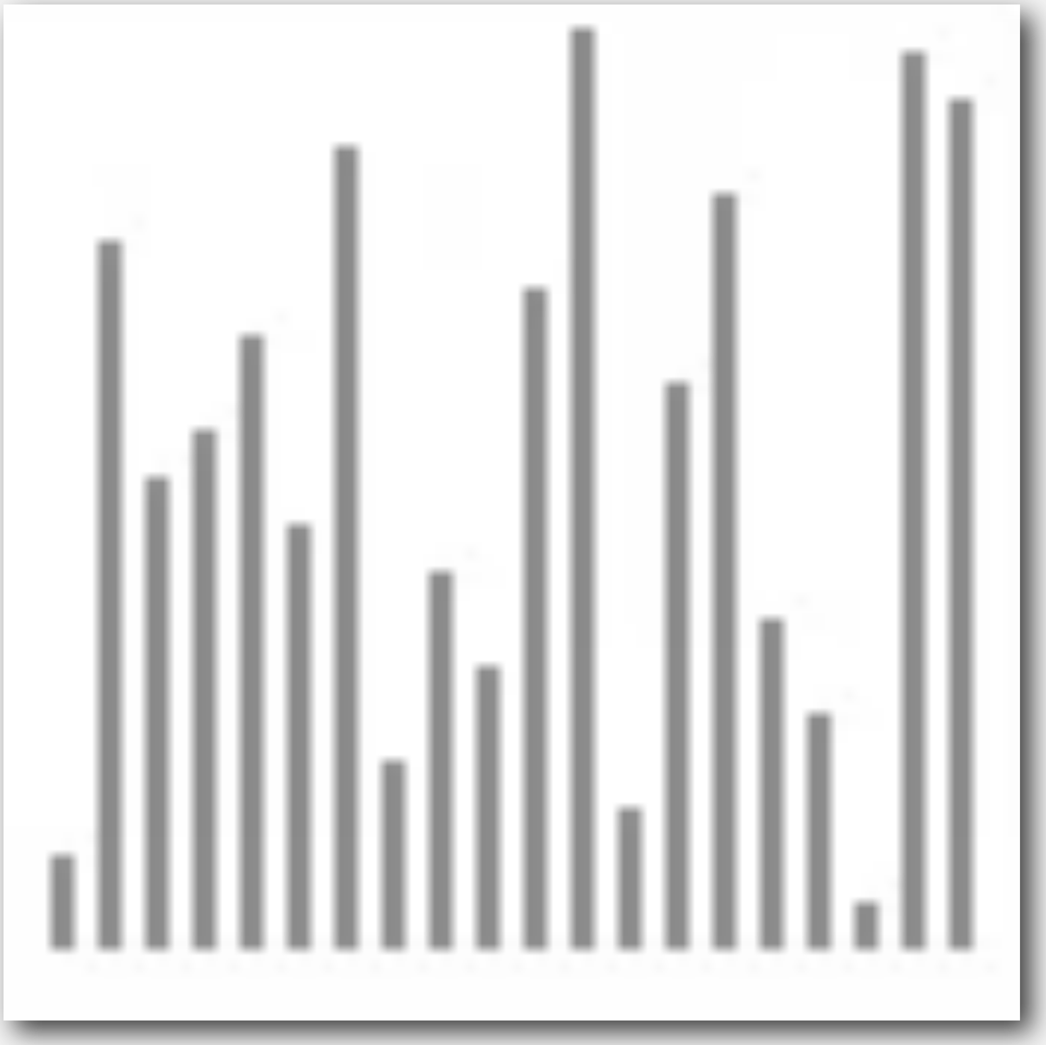
Trace of selection sort (array contents just after each exchange)

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

Data movement is minimal. Linear number of exchanges.

Selection sort: animations

20 random items



▲ algorithm position

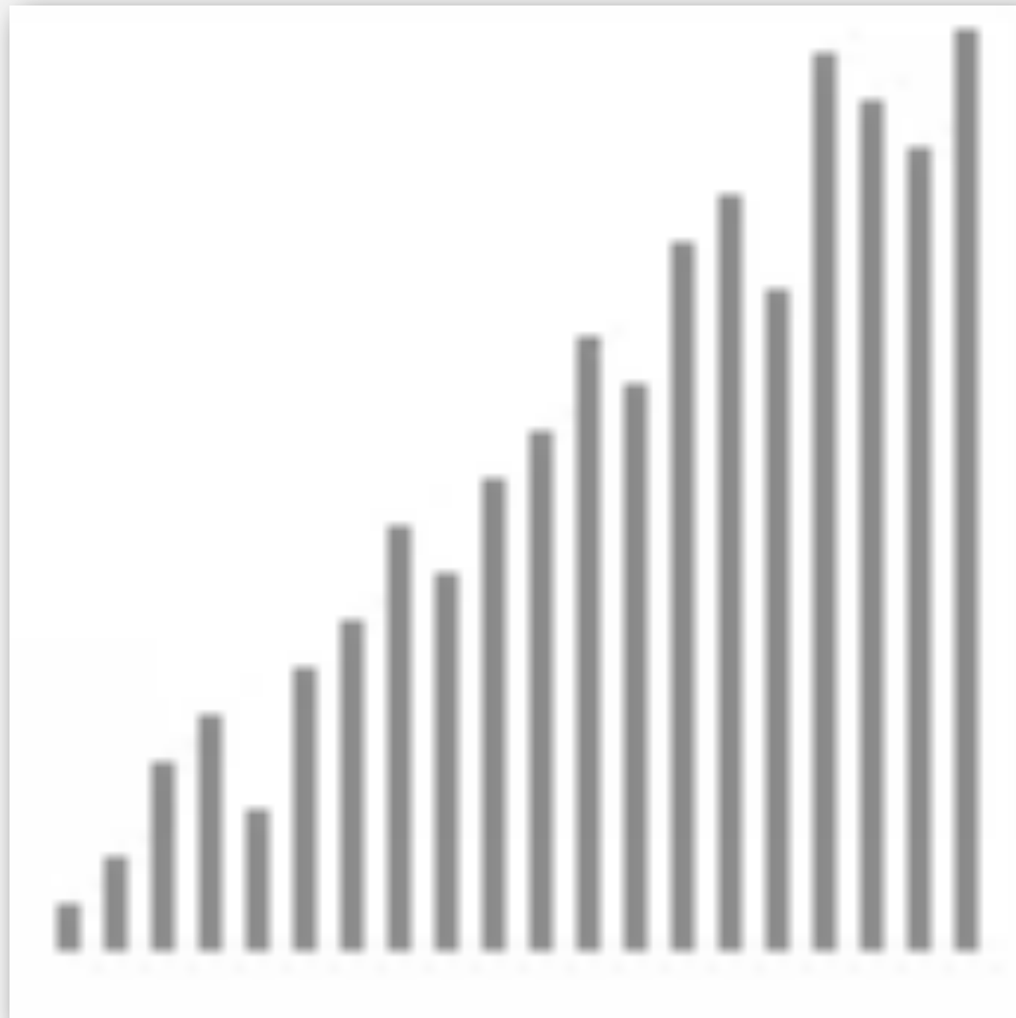
in final order

not in final order

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

Selection sort: animations

20 partially-sorted items



algorithm position

in final order

not in final order

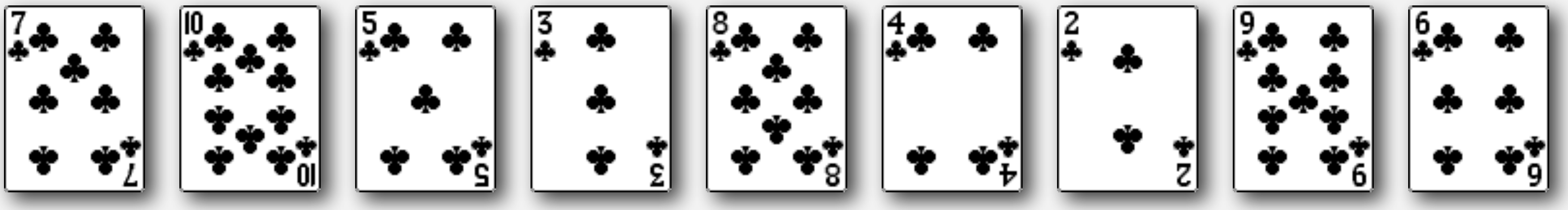
http://www.sorting-algorithms.com/selection-sort

Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort

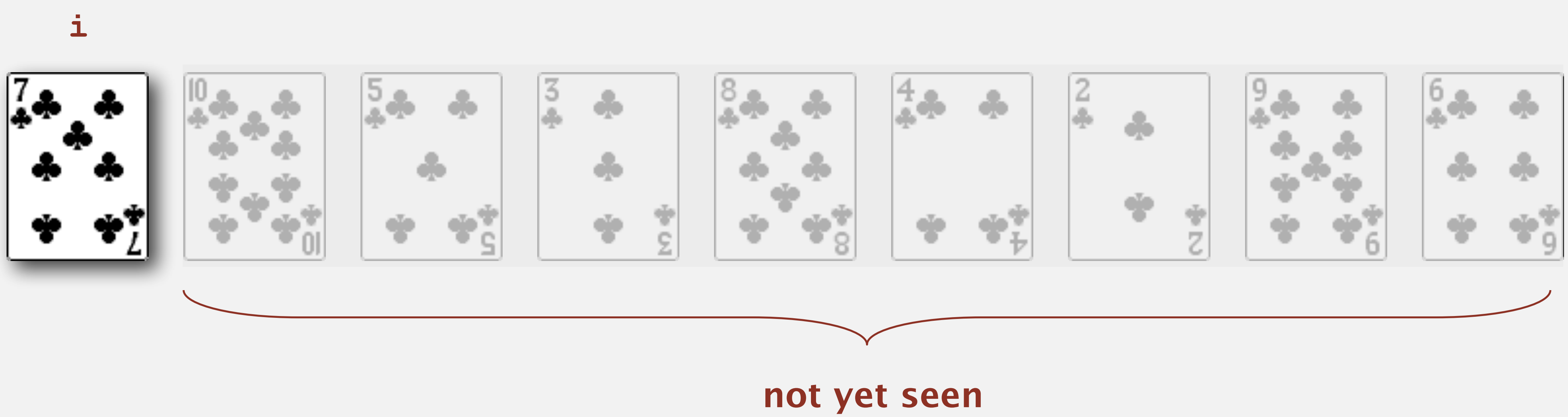
Insertion sort

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



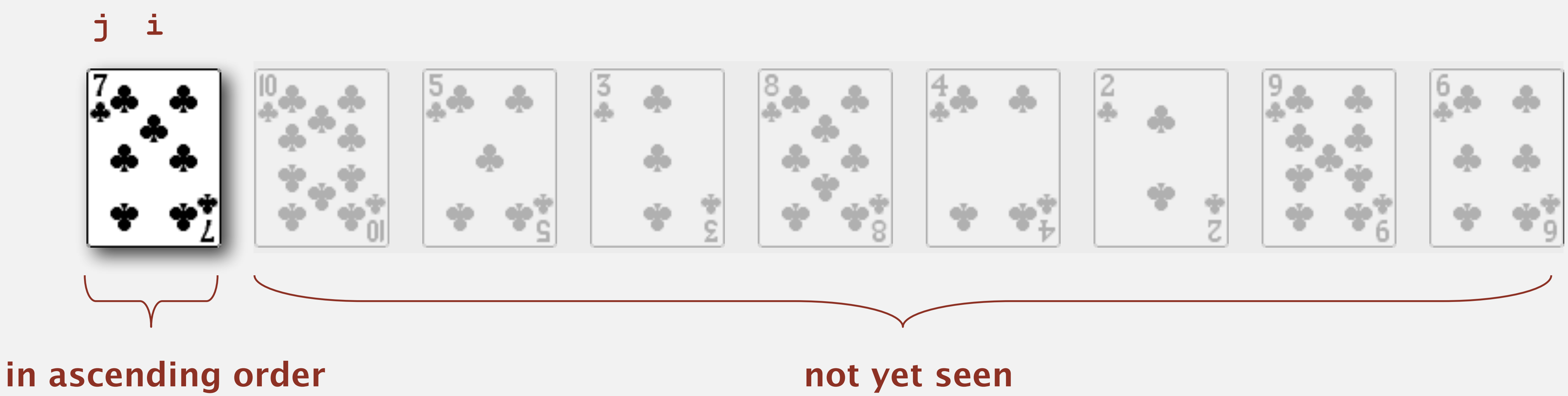
Insertion sort

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



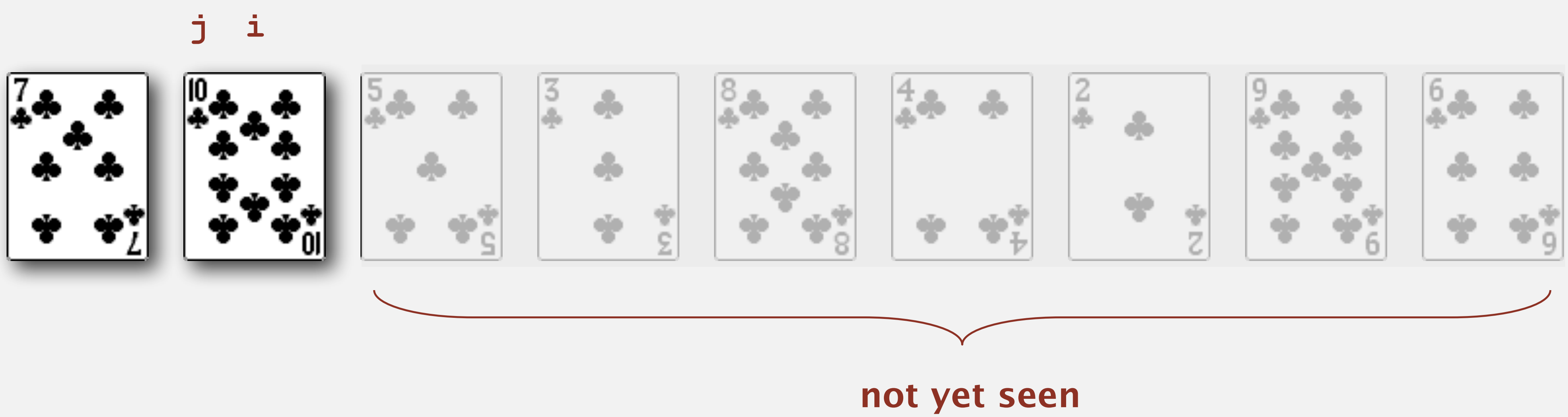
Selection sort

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



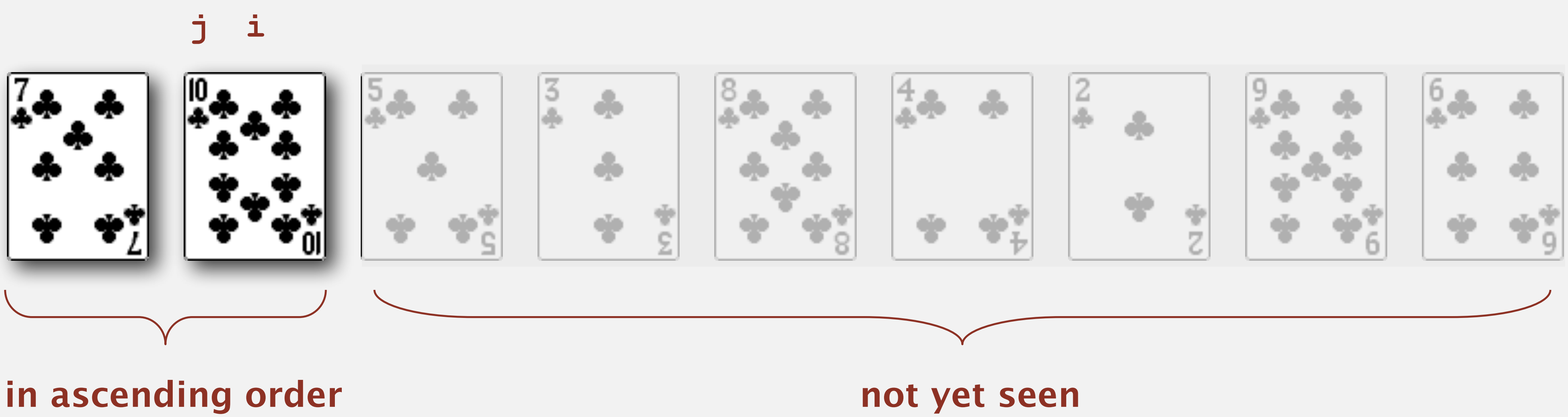
Insertion sort

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



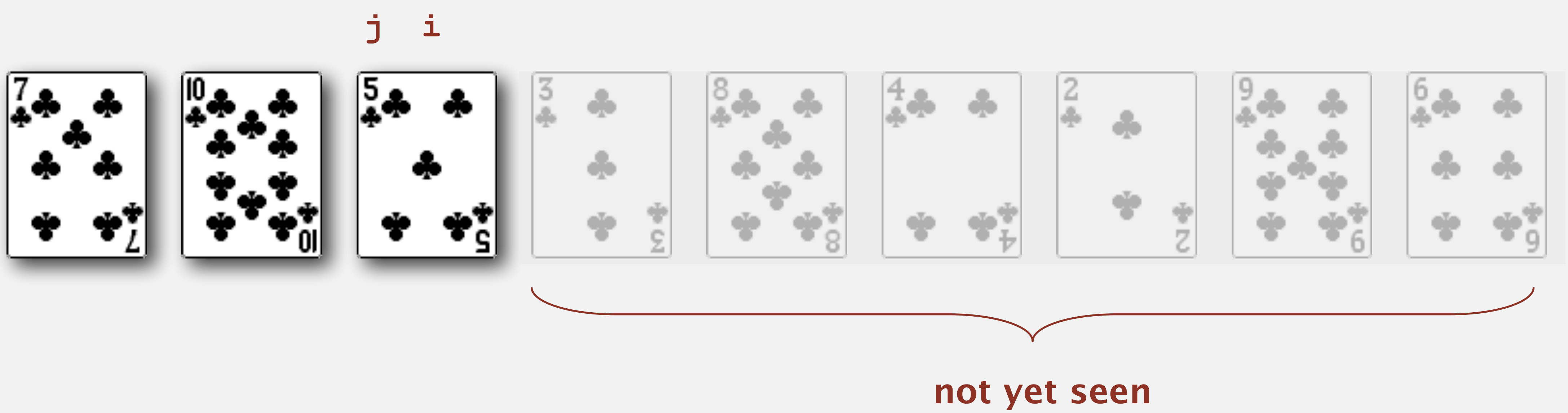
Insertion sort

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



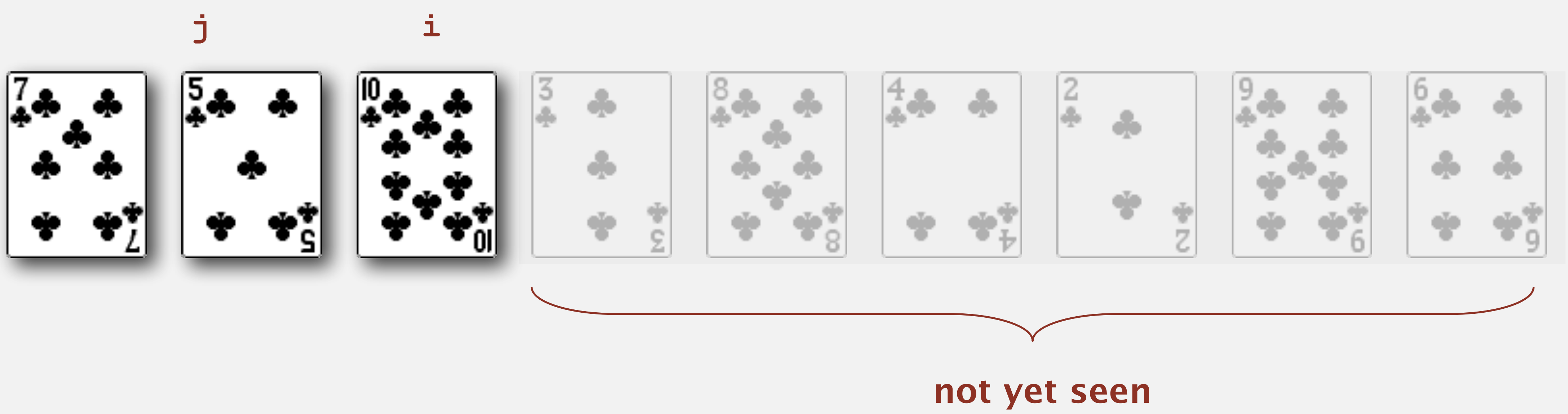
Insertion sort

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



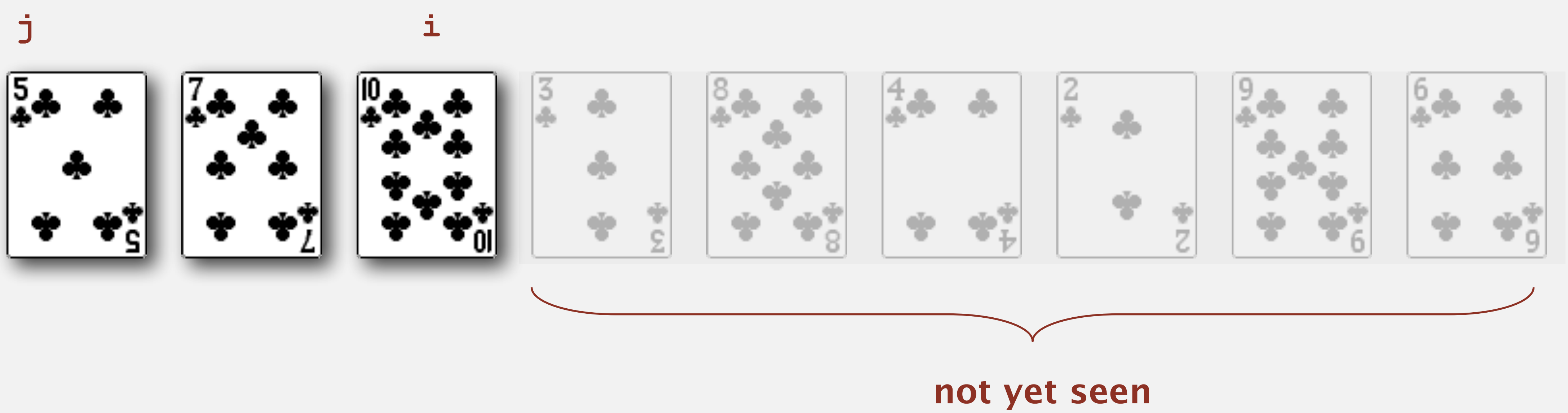
Insertion sort

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



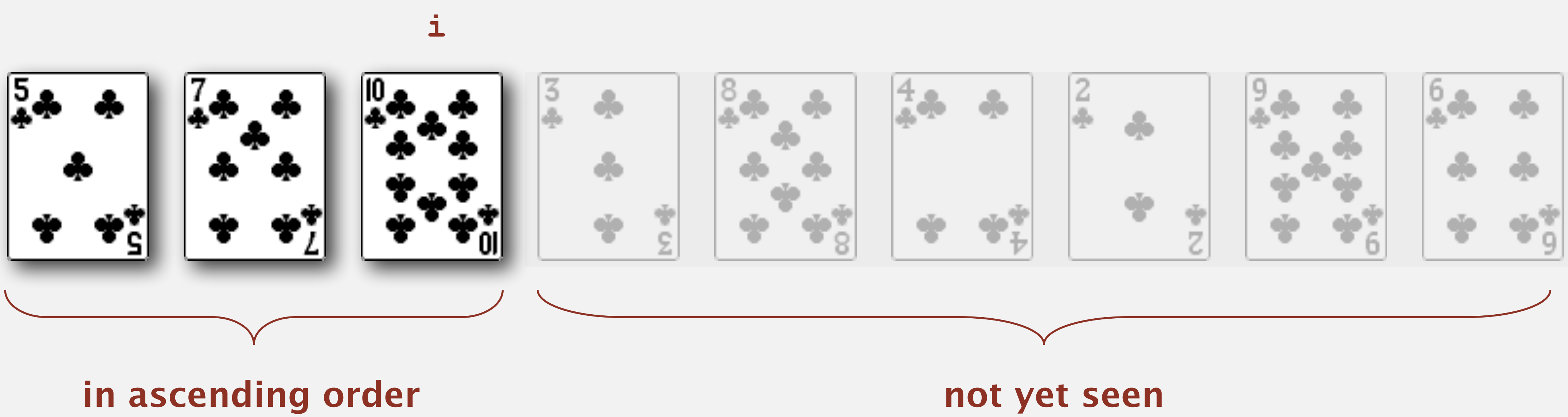
Insertion sort

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



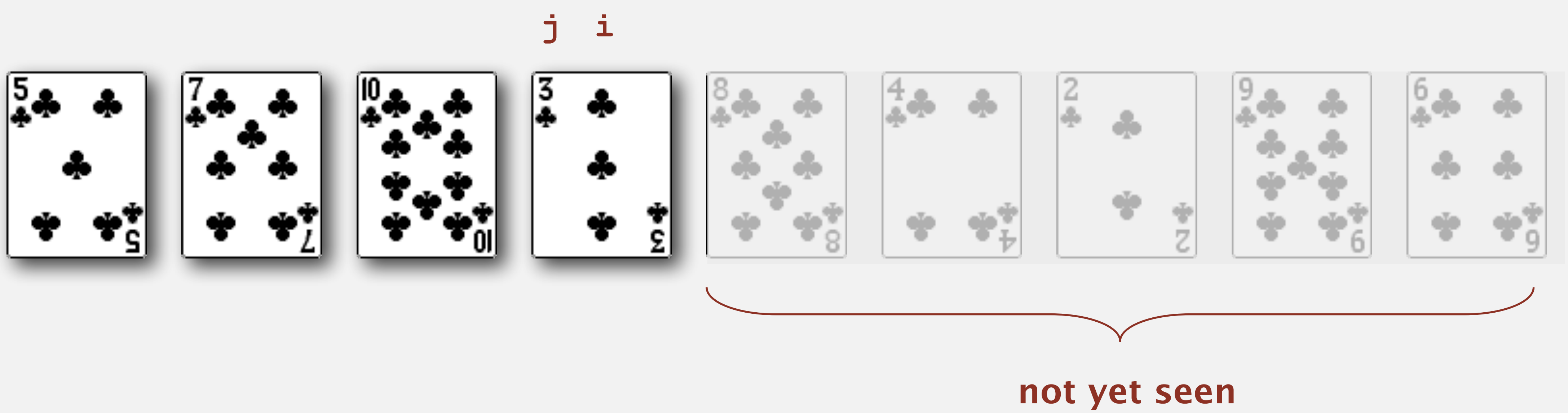
Insertion sort

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



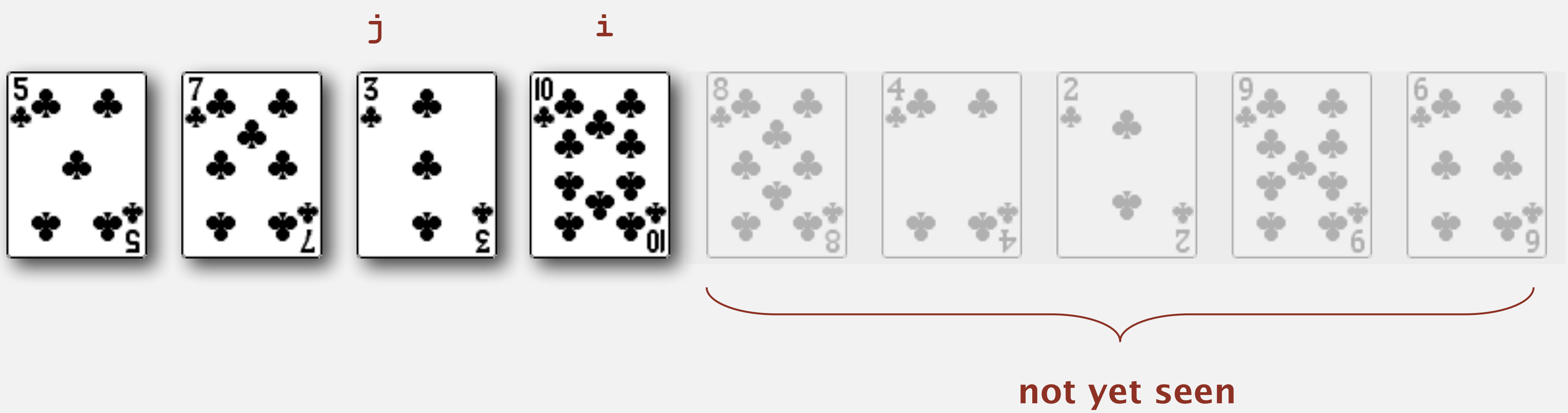
Insertion sort

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



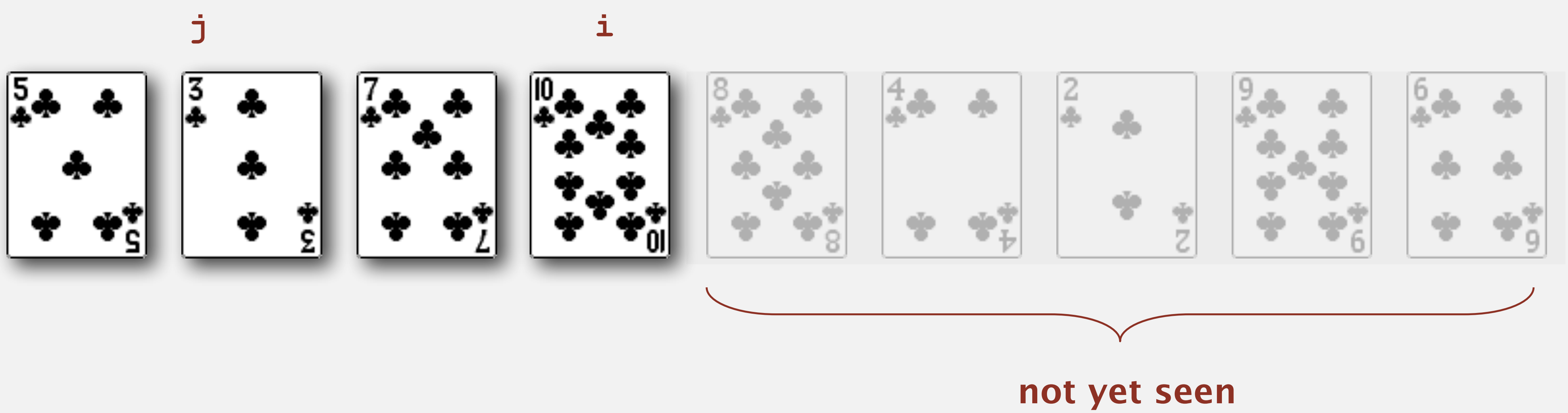
Insertion sort

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



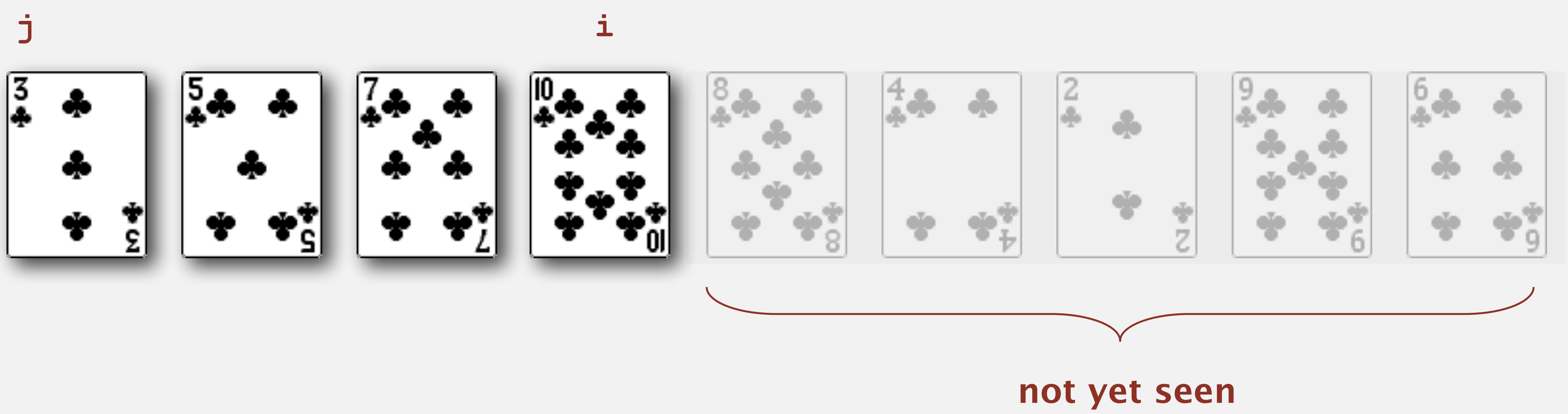
Insertion sort

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



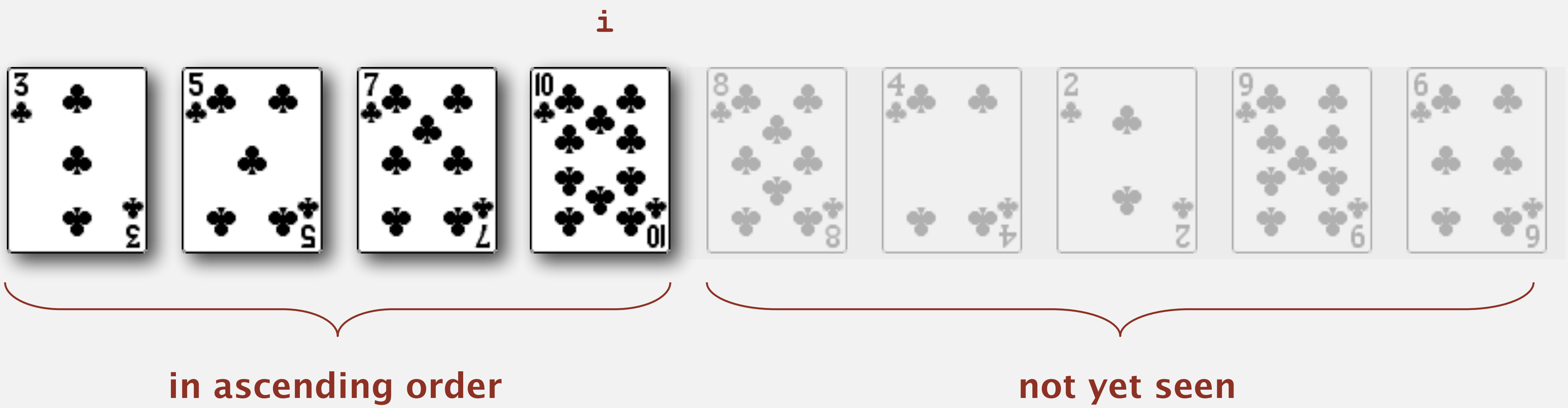
Insertion sort

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



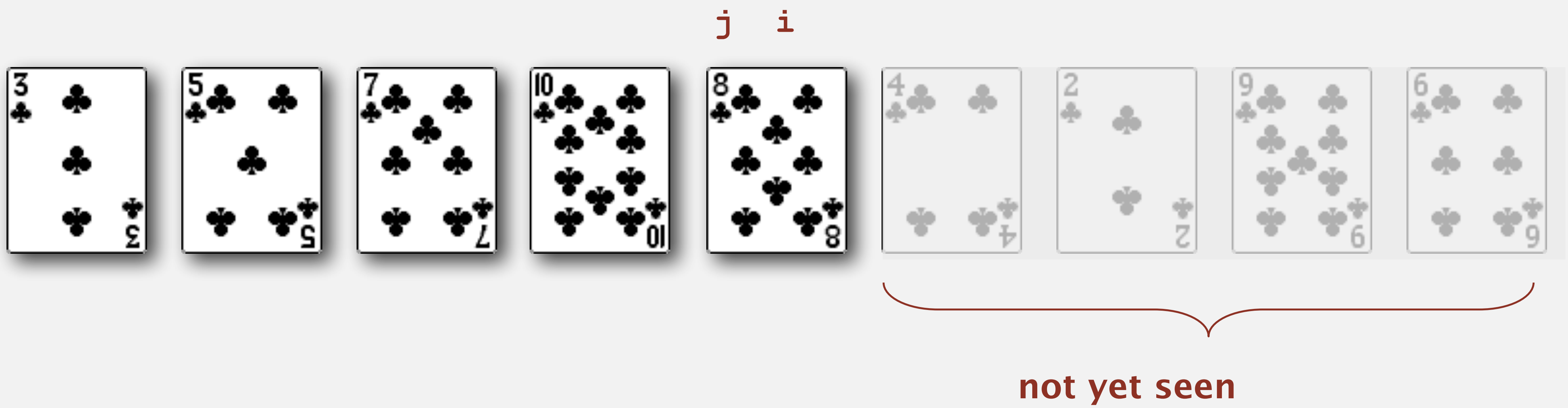
Insertion sort

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



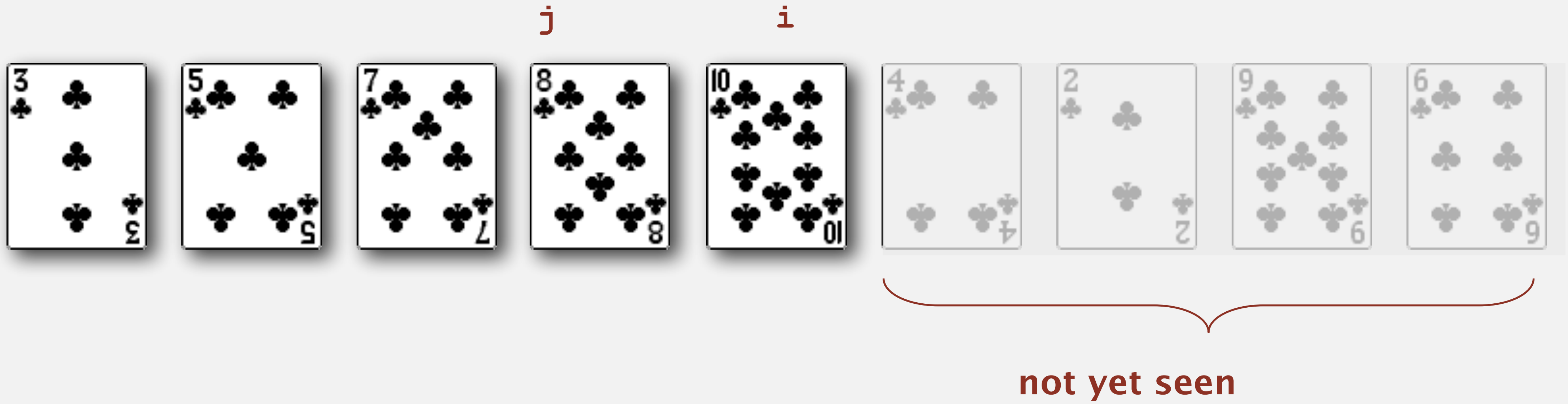
Insertion sort

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



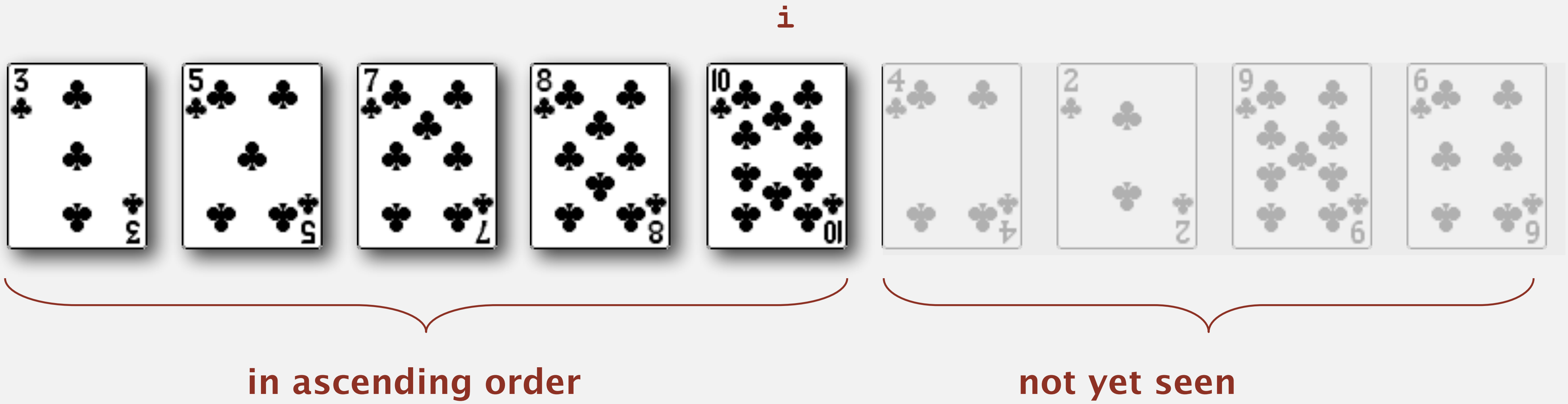
Insertion sort

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



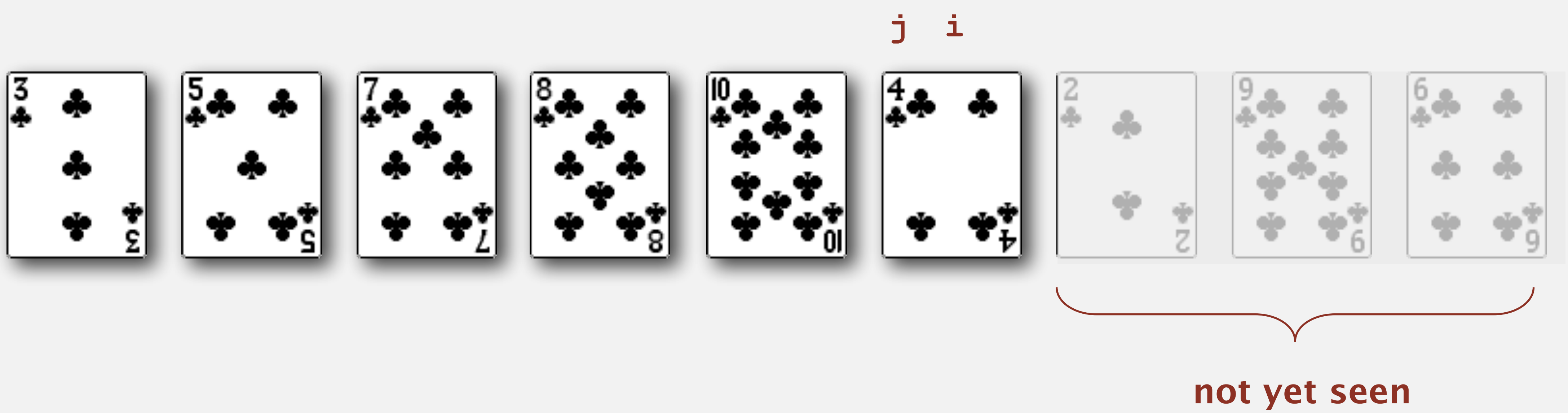
Insertion sort

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



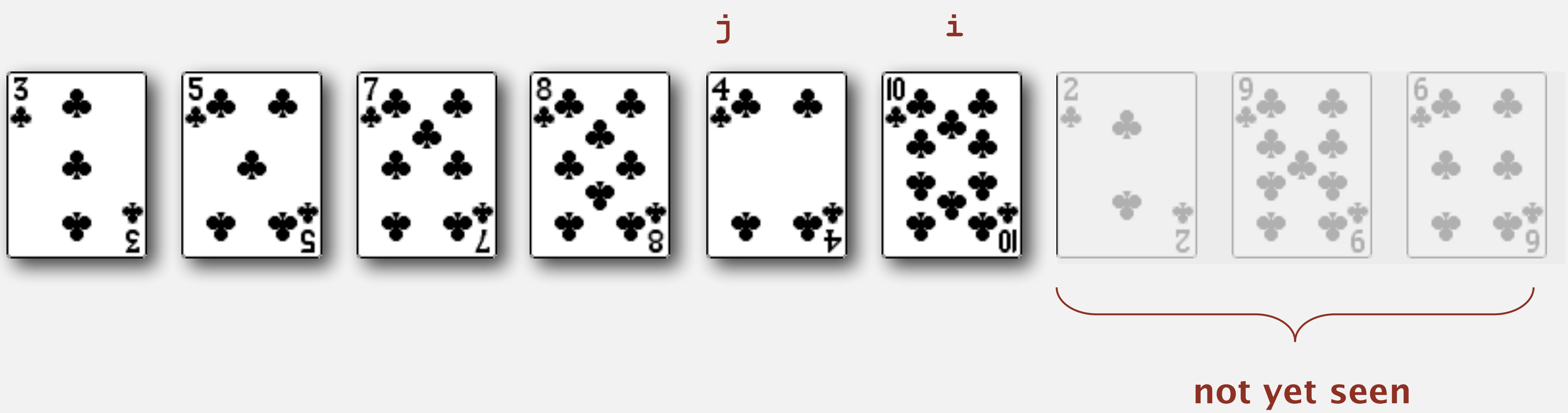
Insertion sort

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



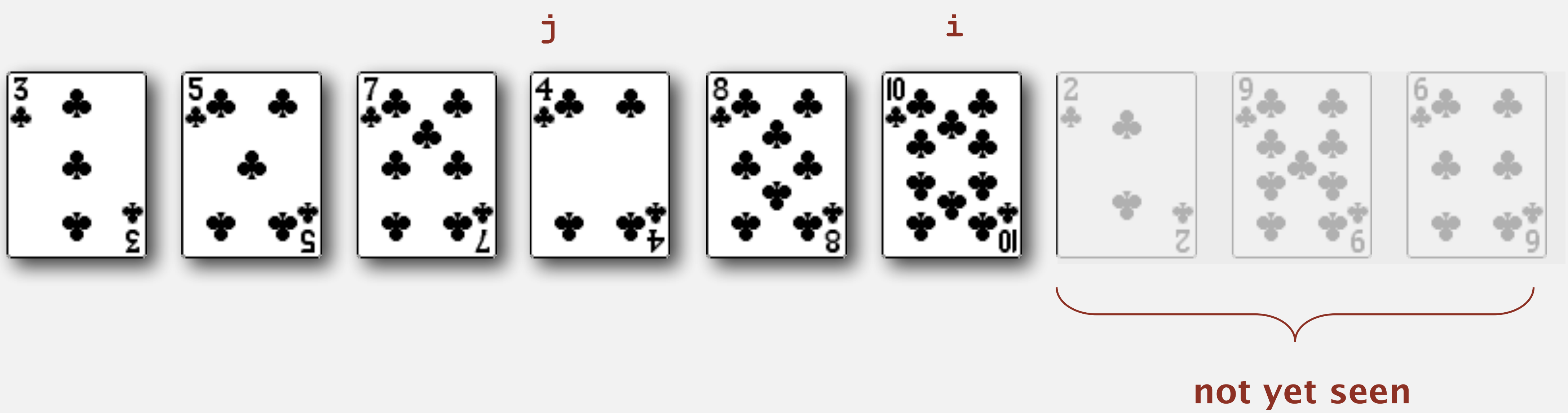
Insertion sort

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



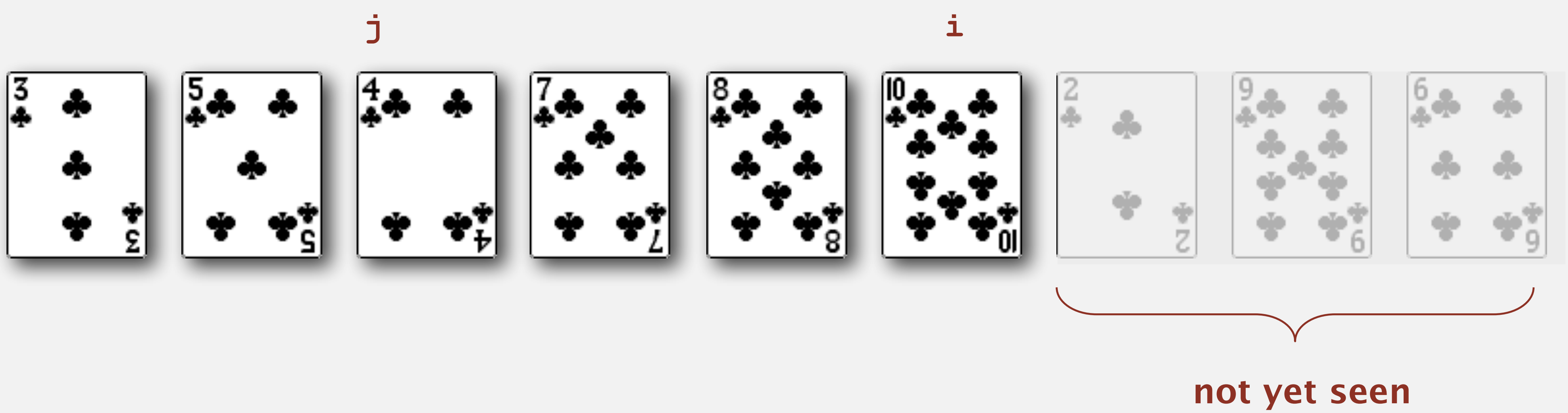
Insertion sort

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



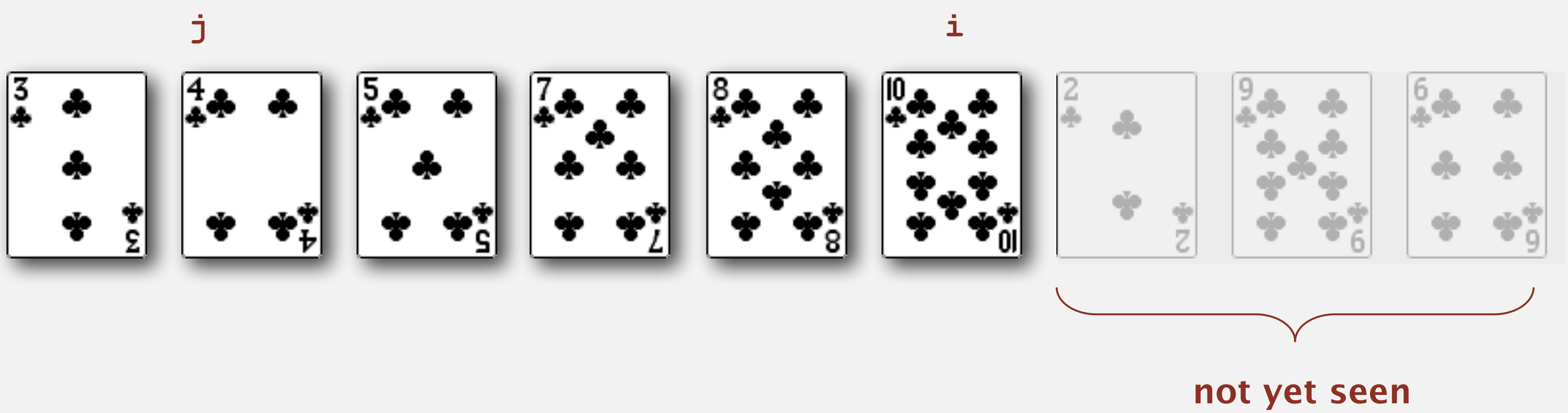
Insertion sort

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



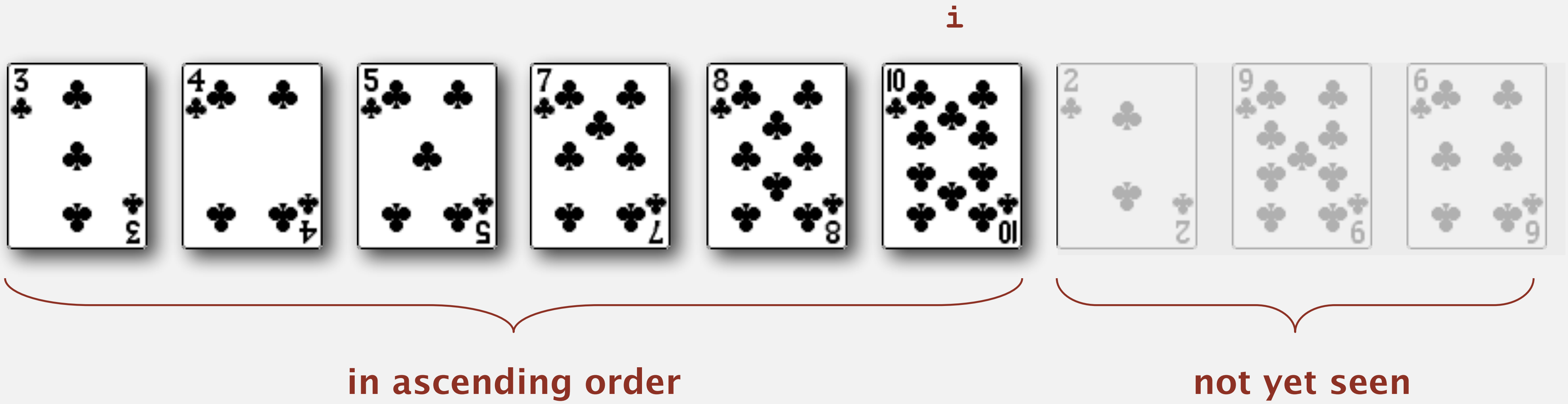
Insertion sort

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



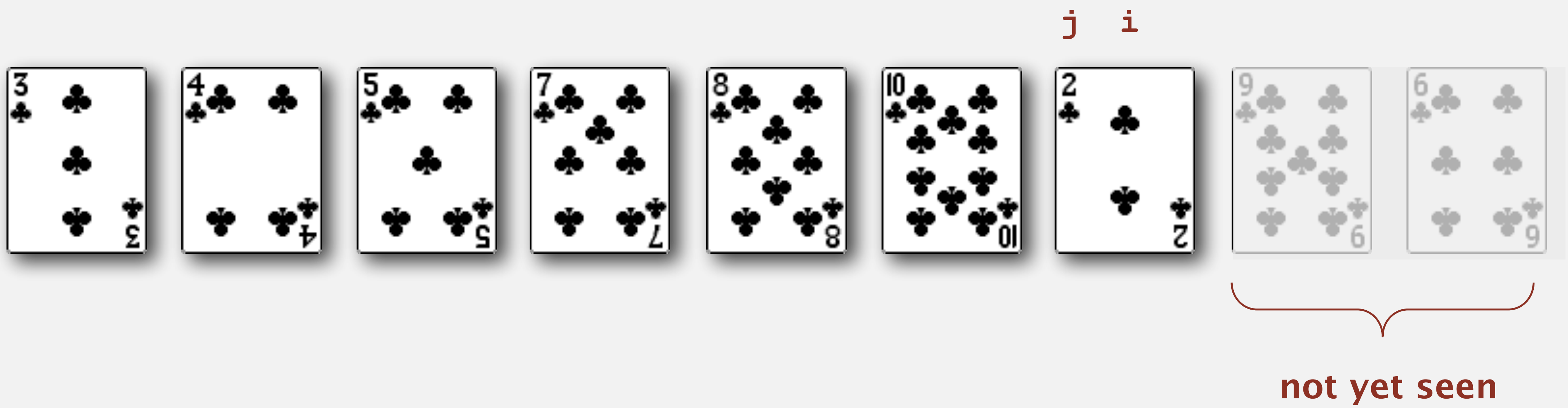
Insertion sort

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



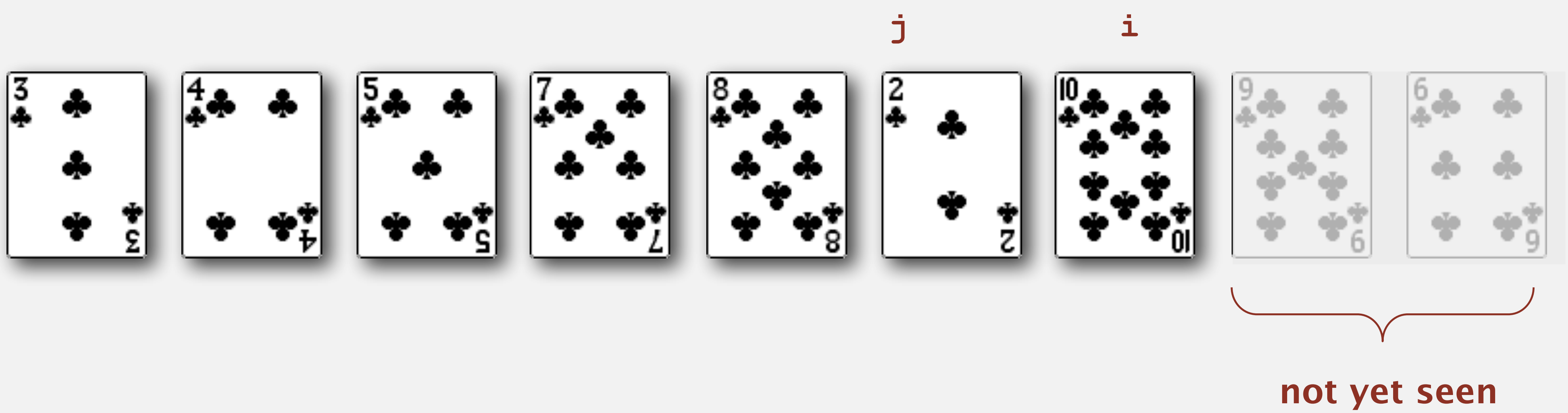
Insertion sort

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



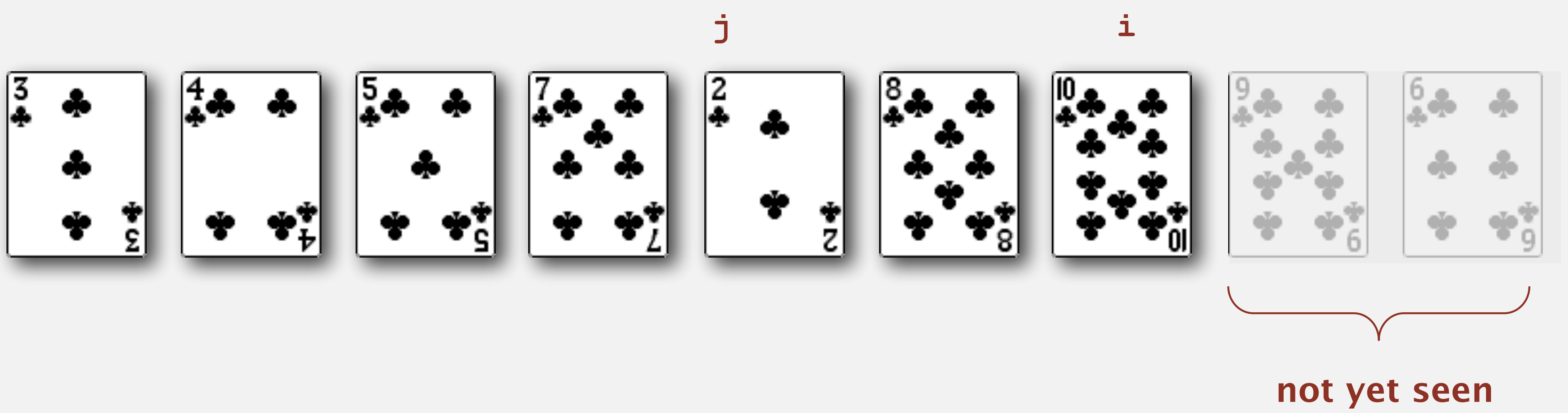
Insertion sort

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



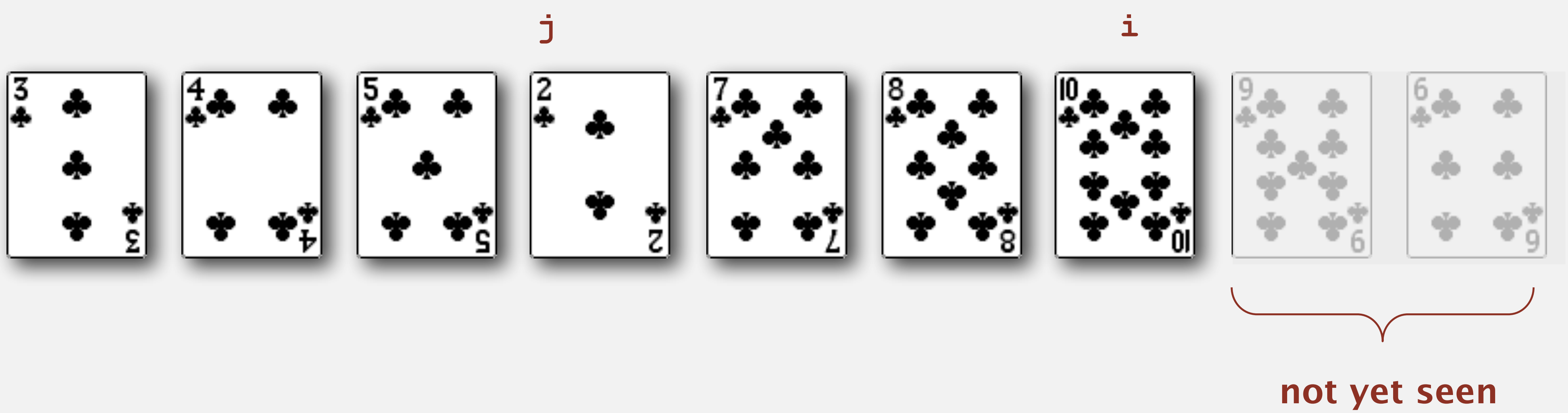
Insertion sort

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



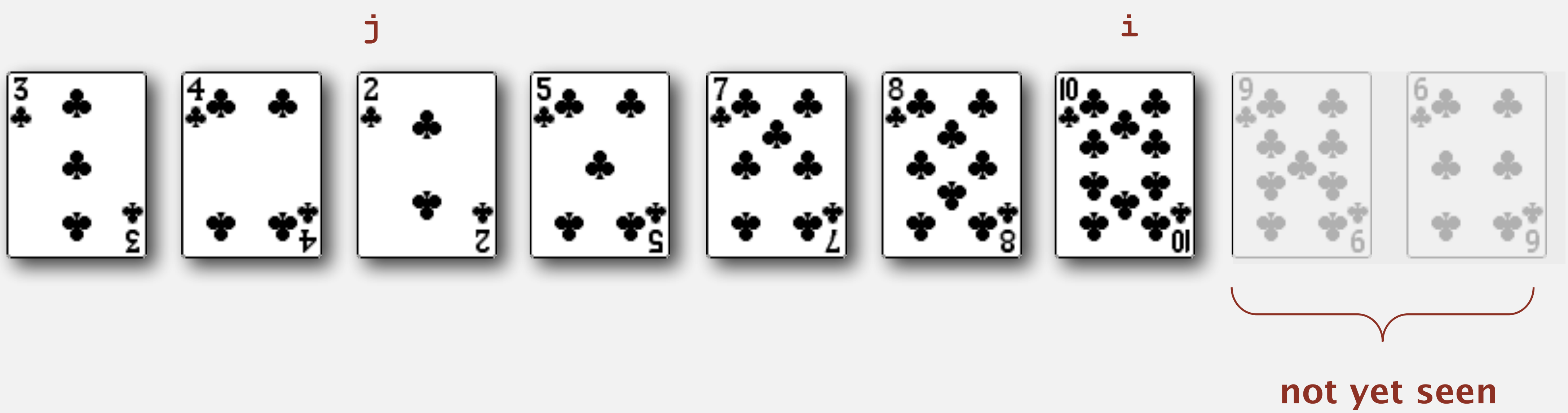
Insertion sort

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



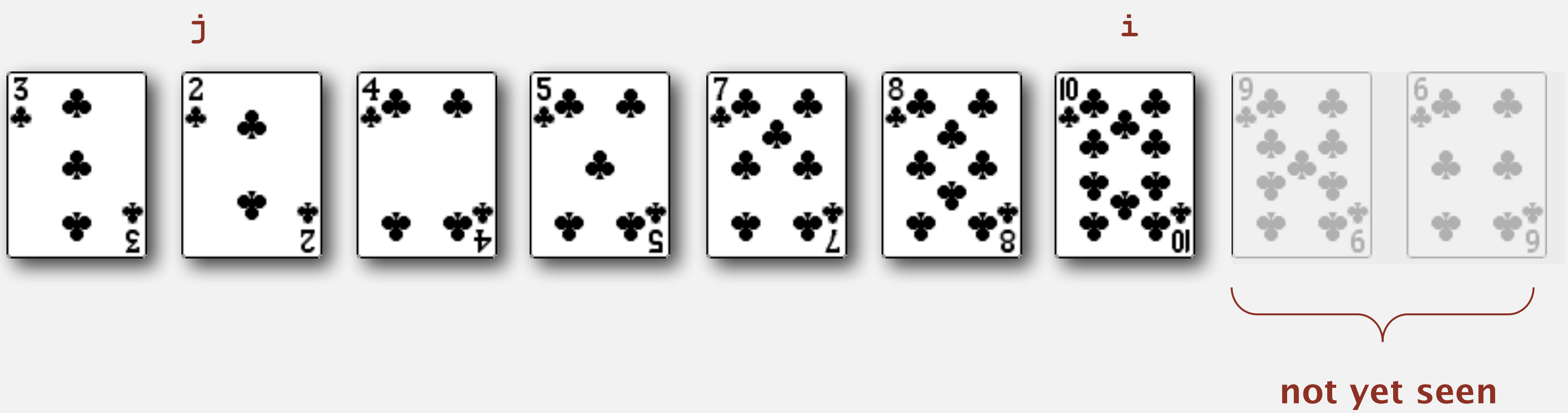
Insertion sort

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



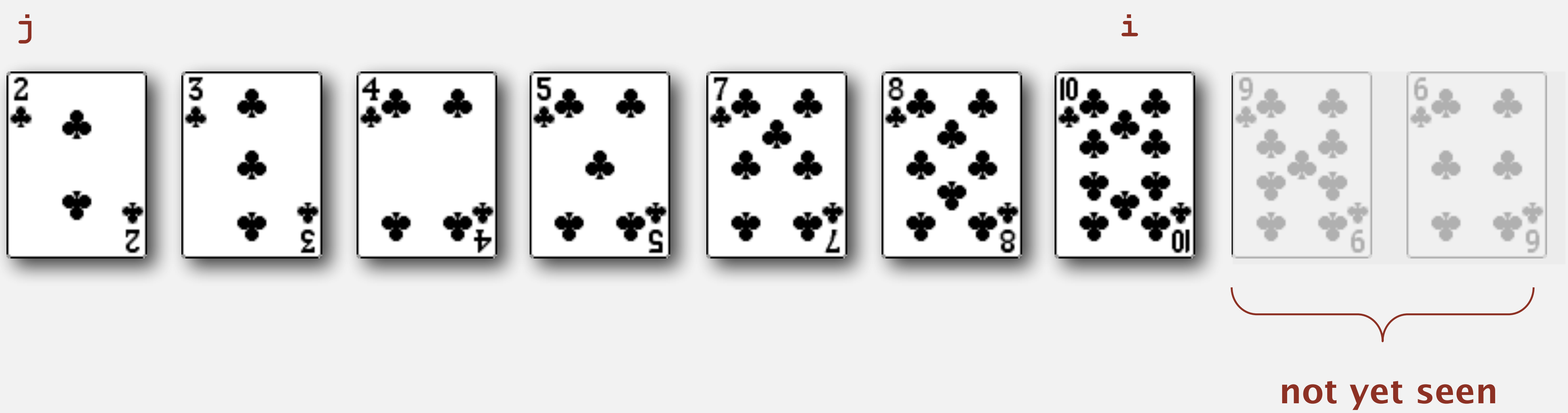
Insertion sort

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



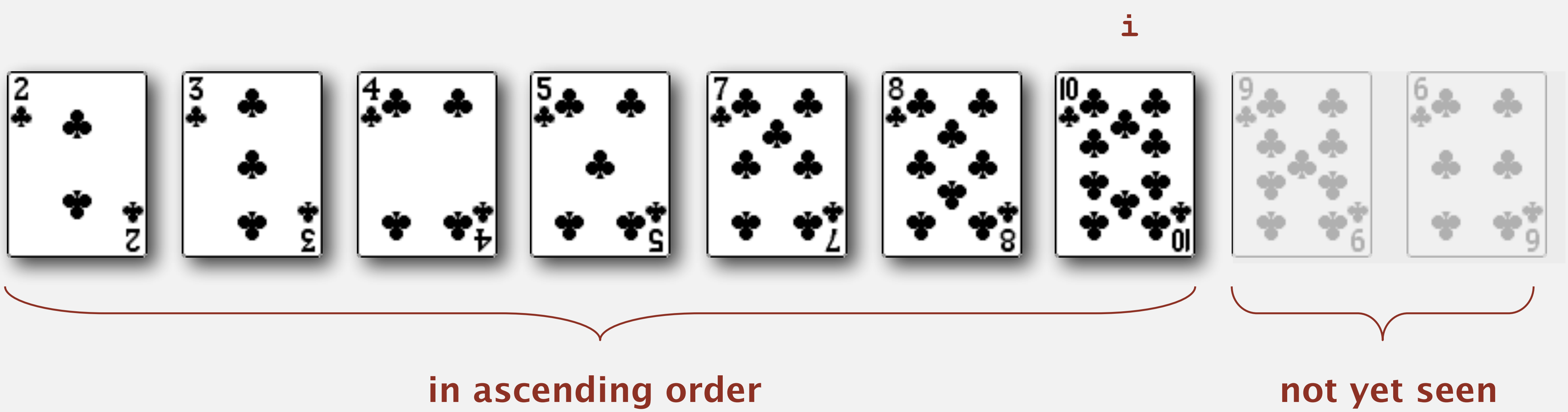
Insertion sort

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



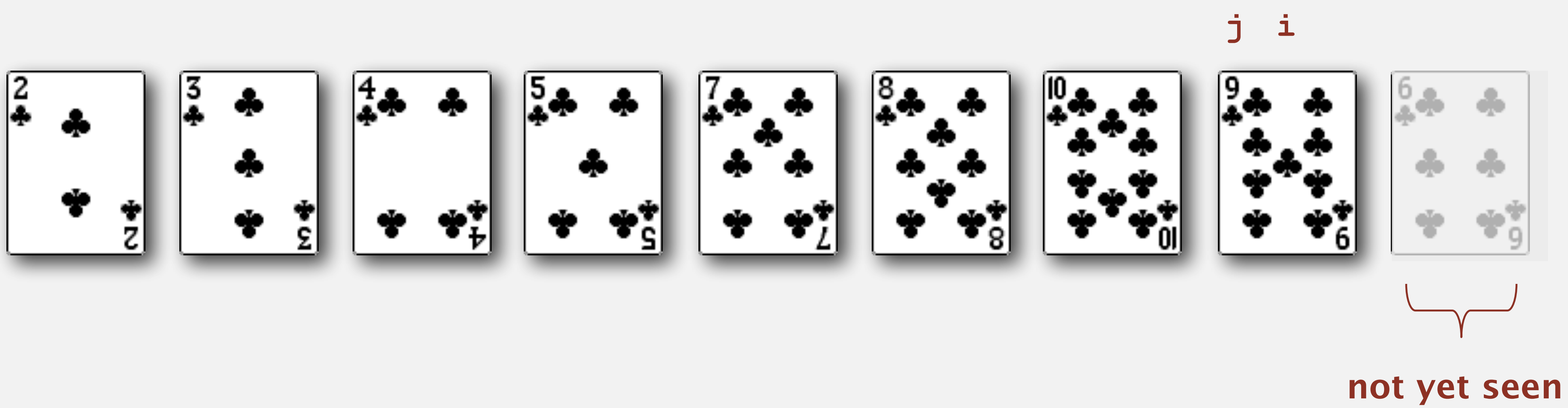
Insertion sort

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



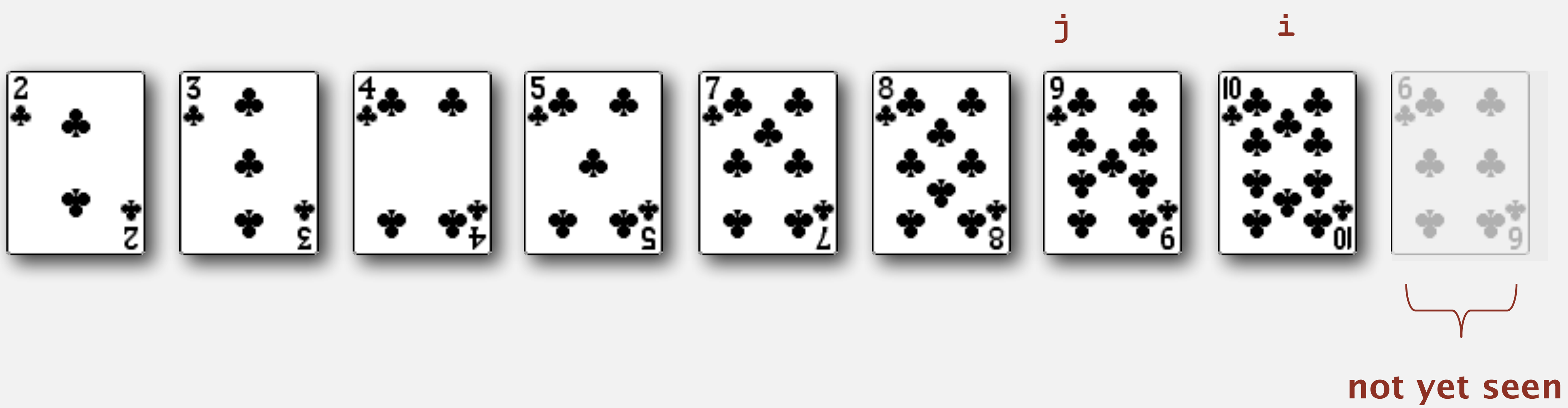
Insertion sort

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



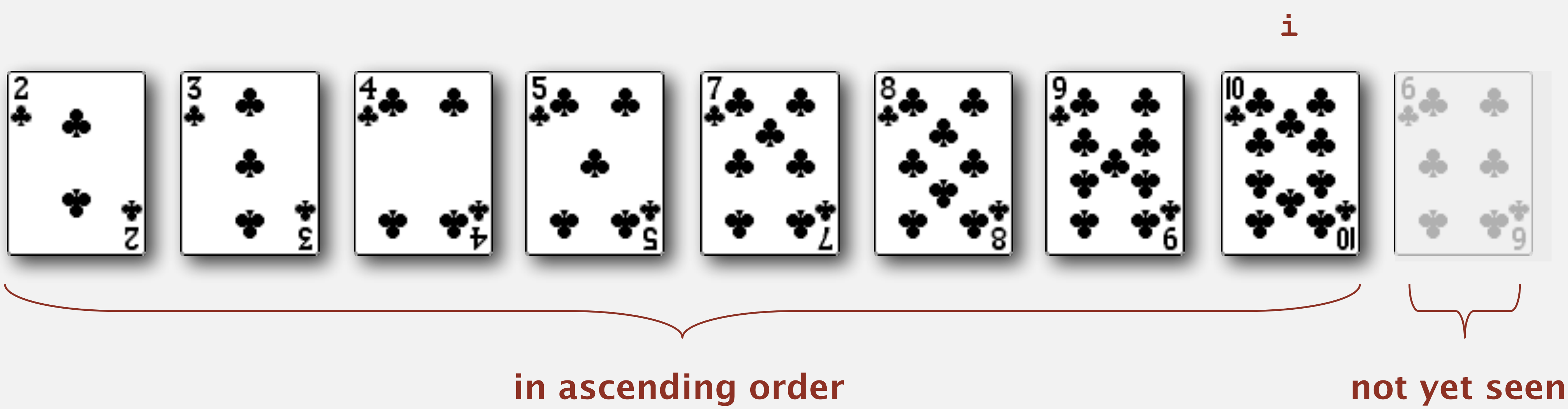
Insertion sort

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



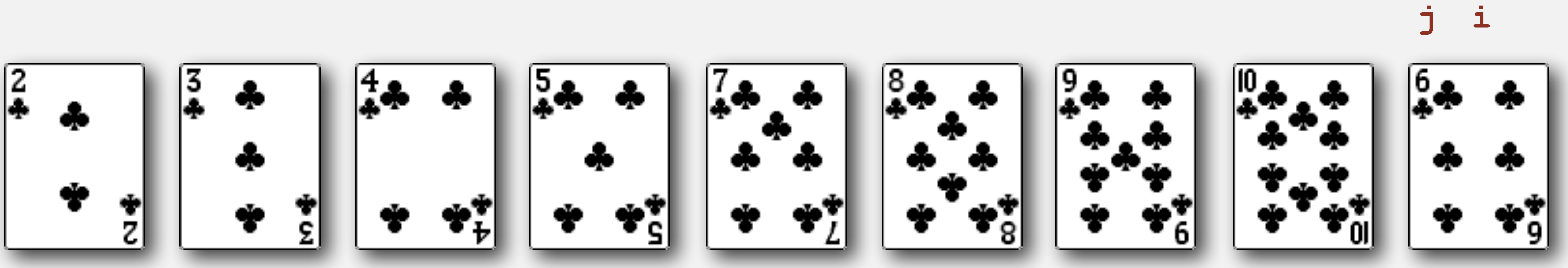
Insertion sort

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



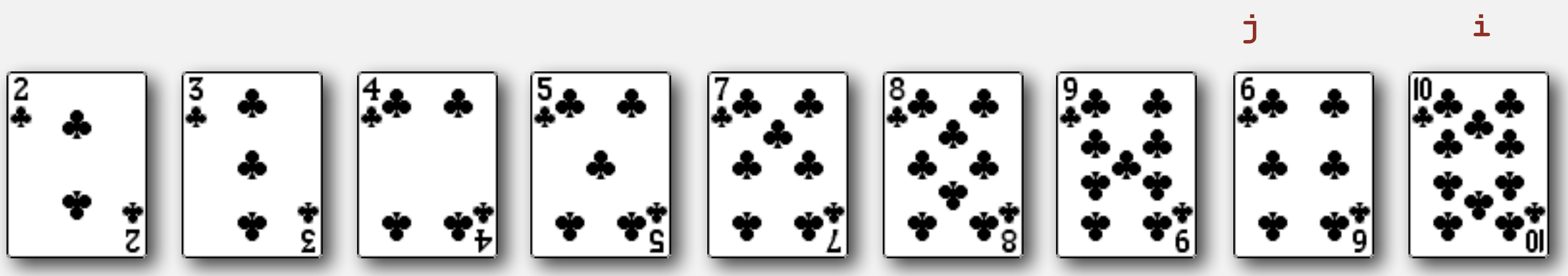
Insertion sort

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



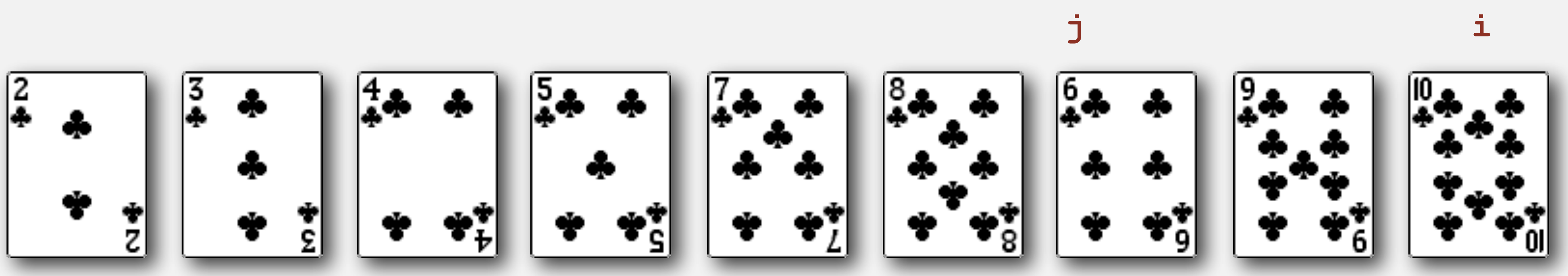
Insertion sort

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



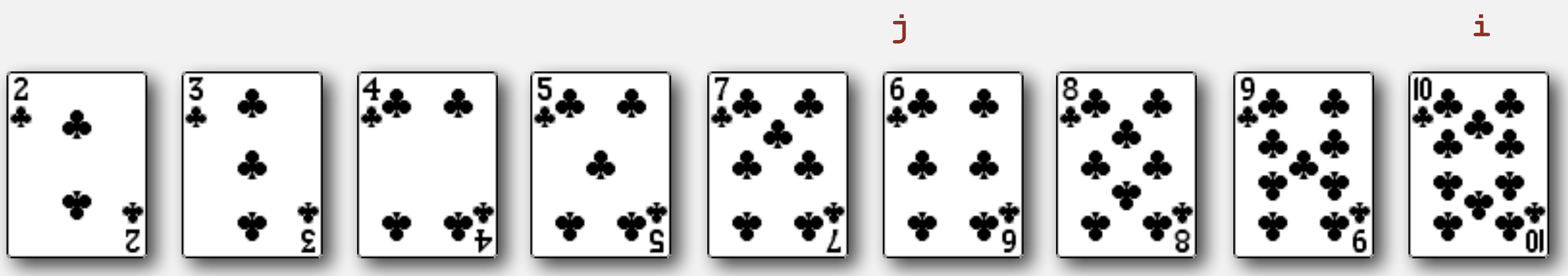
Insertion sort

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



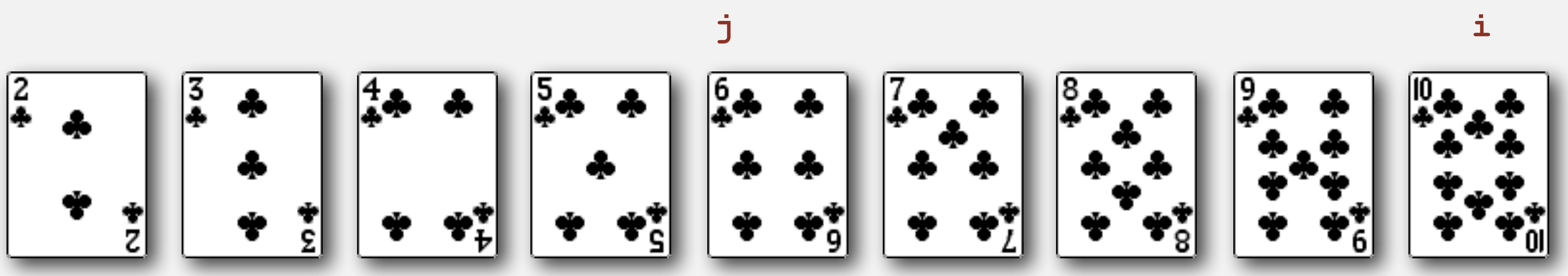
Insertion sort

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



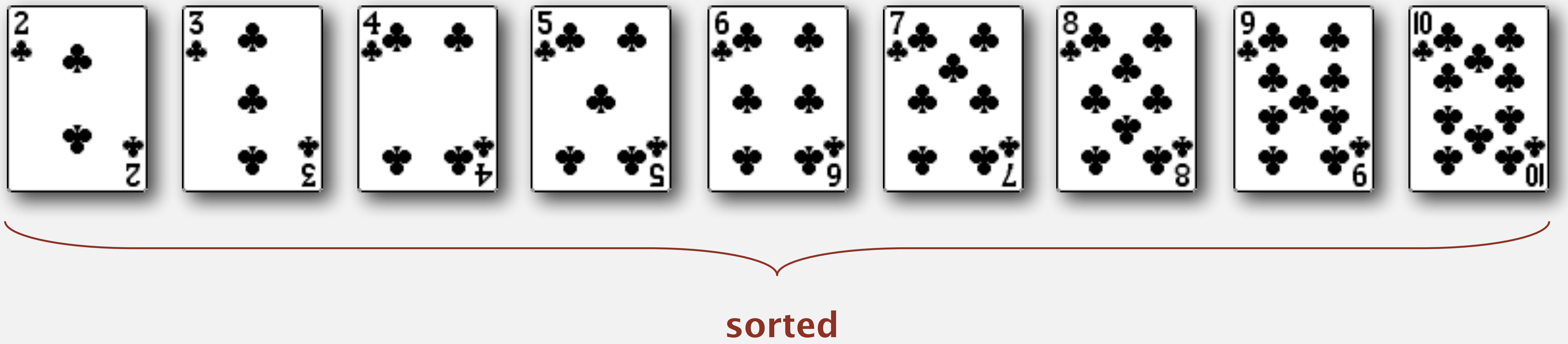
Insertion sort

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



Insertion sort

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

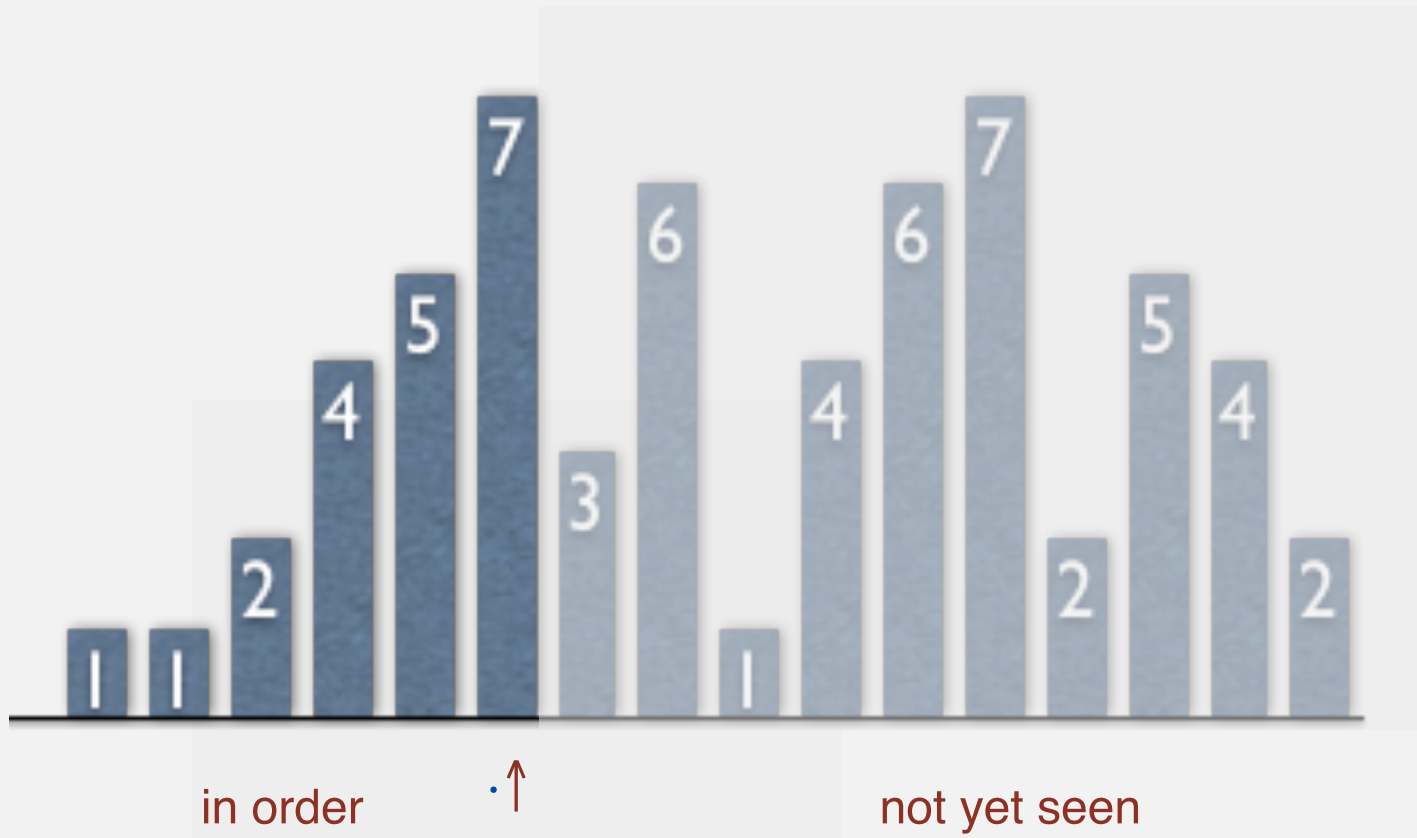


Insertion sort

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(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

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

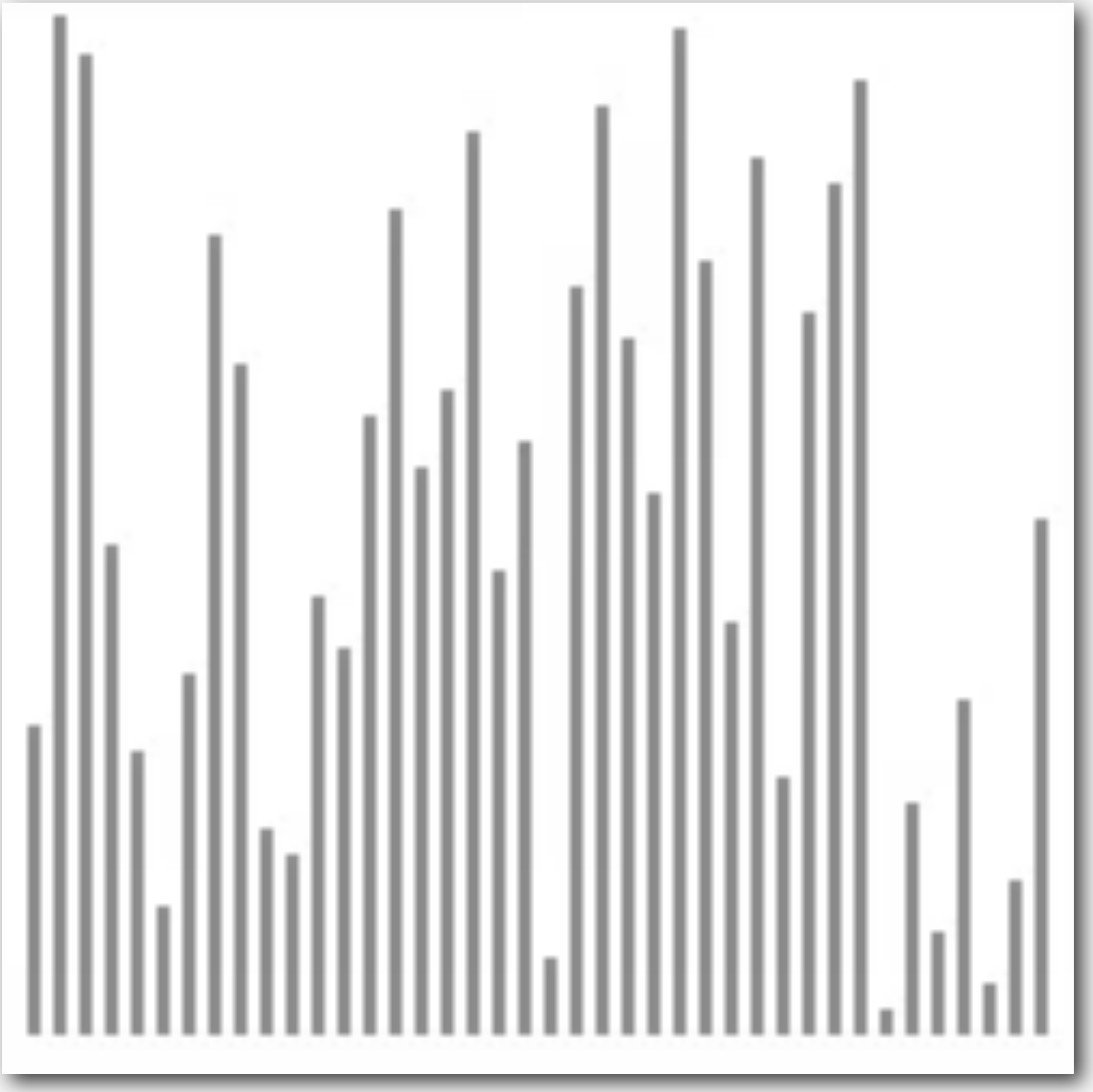
entry in red is a[j]

entries in black moved one position right for insertion

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

Insertion sort: animation

40 random items



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

▲ algorithm position

in order

not yet seen

Poll time!

What is the worst case of Insertion Sort?

a. A randomly-ordered array with distinct elements
b. An array in ascending order with distinct elements
c. An array in descending order with distinct elements

Visit the following link to submit your answers:
https://forms.gle/f4CgsSrmj8gwT6sJA

SCAN ME

Insertion sort: best and worst case

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

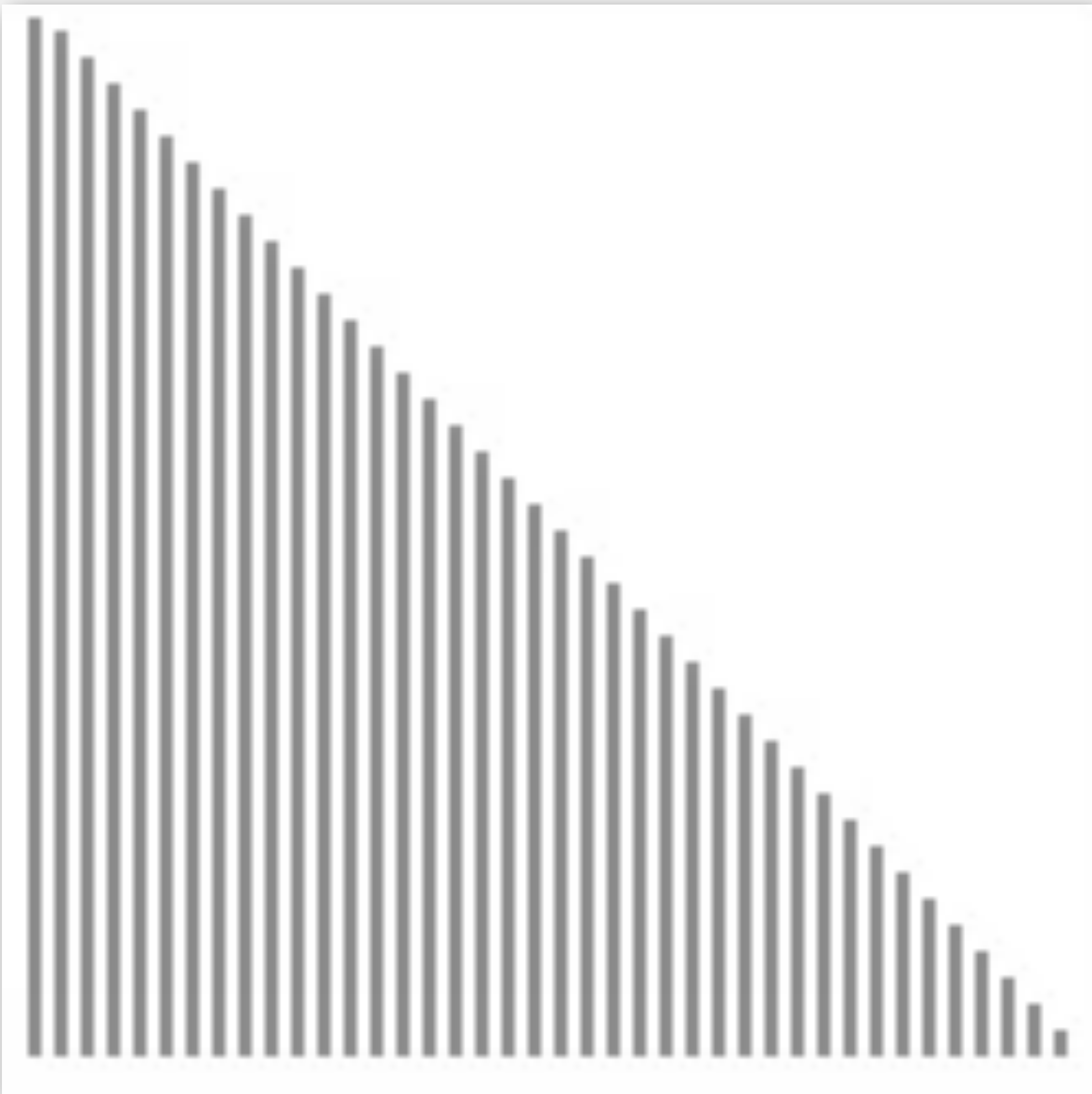
```
A E E L M O P R S T X
```

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.

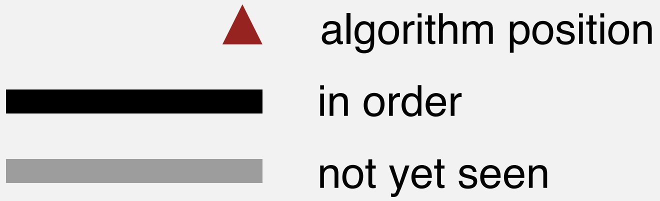
```
X T S R P O M L E E A
```

Insertion sort: animation

40 reverse-sorted items



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 $\leq c\,N$.

- Ex 1. A subarray of size 10 appended to a sorted subarray of size N.
- Ex 2. An array of size N with only 10 entries out of place.

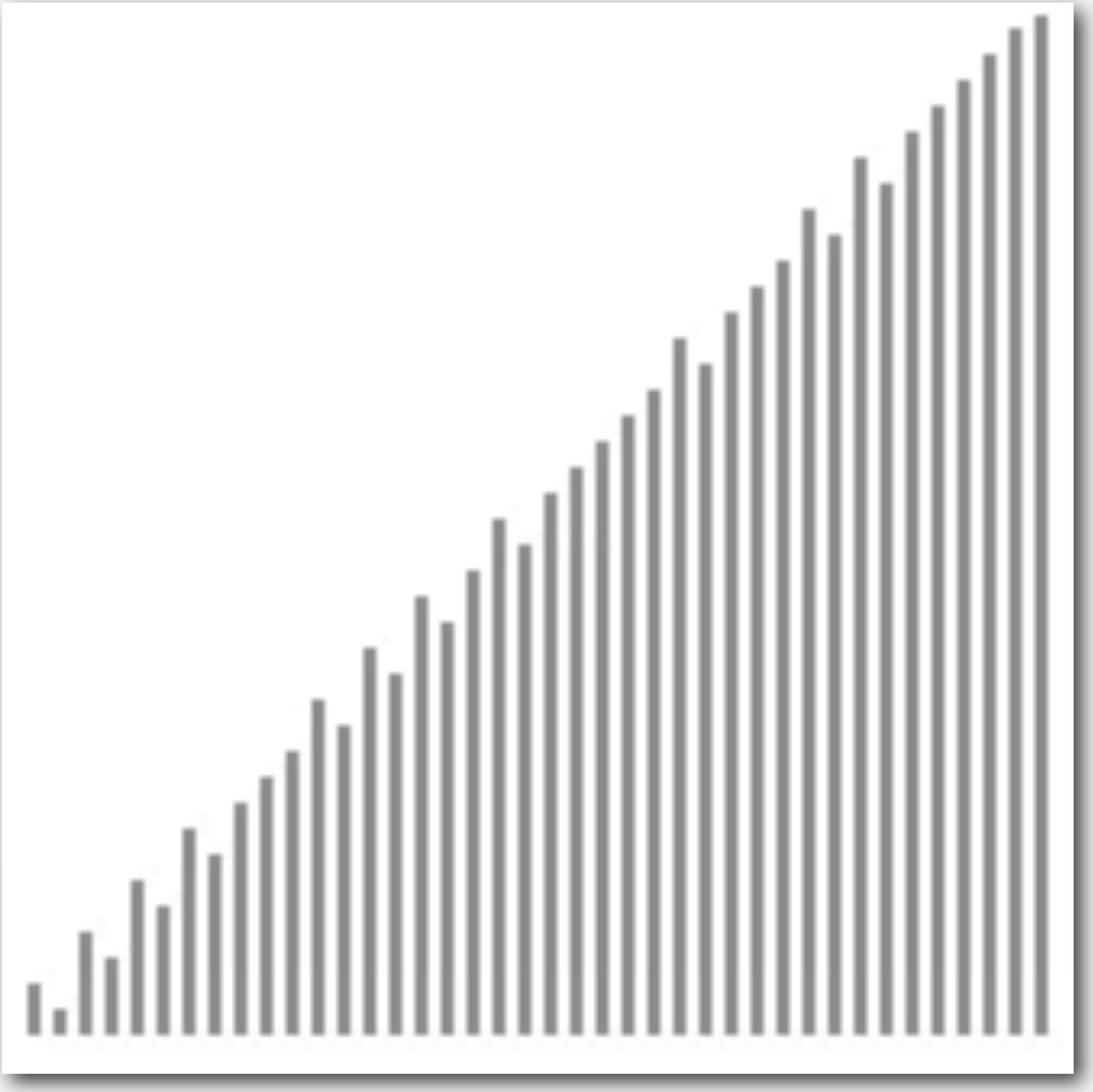
Proposition. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

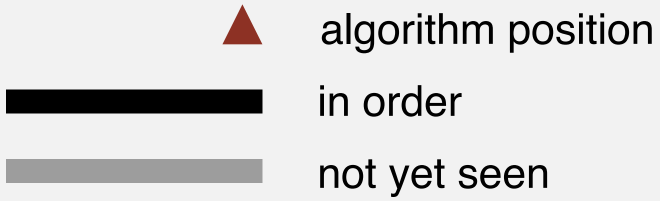
number of compares = exchanges + (N – 1)

Insertion sort: animation

40 partially-sorted items



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



Elementary Sorting Algorithms

- **Sorting review**
- Rules of the game
- Selection sort
- Insertion sort
- **Shellsort**

Shellsort overview

Idea. Move entries more than one position at a time by *h*-sorting the array.

an h-sorted array is h interleaved sorted subsequences

h = 4

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

Shellsort. [Shell 1959] *h*-sort the array for decreasing seq. of values of *h*.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

h-sorting

How to h-sort an array? Insertion sort, with stride length h.

3-sorting an array

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

Why insertion sort?

- Big increments $\Rightarrow$ small subarray.
- Small increments $\Rightarrow$ nearly in order. [stay tuned]

Shellsort example: increments 7, 3, 1

input

S	O	R	T	E	X	A	M	P	L	E

7-sort

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

3-sort

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

1-sort

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

result

A	E	E	L	M	O	P	R	S	T	X

Shellsort: intuition

Proposition. A g-sorted array remains g-sorted after h-sorting it.

7-sort

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

3-sort

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

still 7-sorted

Challenge. Prove this fact—it's more subtle than you'd think!

Shellsort: which increment sequence to use?

Powers of two. 1, 2, 4, 8, 16, 32, ...

No.

Powers of two minus one. 1, 3, 7, 15, 31, 63, ...

Maybe.

3x + 1. 1, 4, 13, 40, 121, 364, ...

OK. Easy to compute.

merging of $(9 \times 4^i) - (9 \times 2^i) + 1$ and $4^i - (3 \times 2^i) + 1$

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Good. Tough to beat in empirical studies.

=

Interested in learning more?

- See Section 6.8 of Algs, 3rd edition or Volume 3 of Knuth for details.
- Do a JP on the topic.

Shellsort: Java implementation

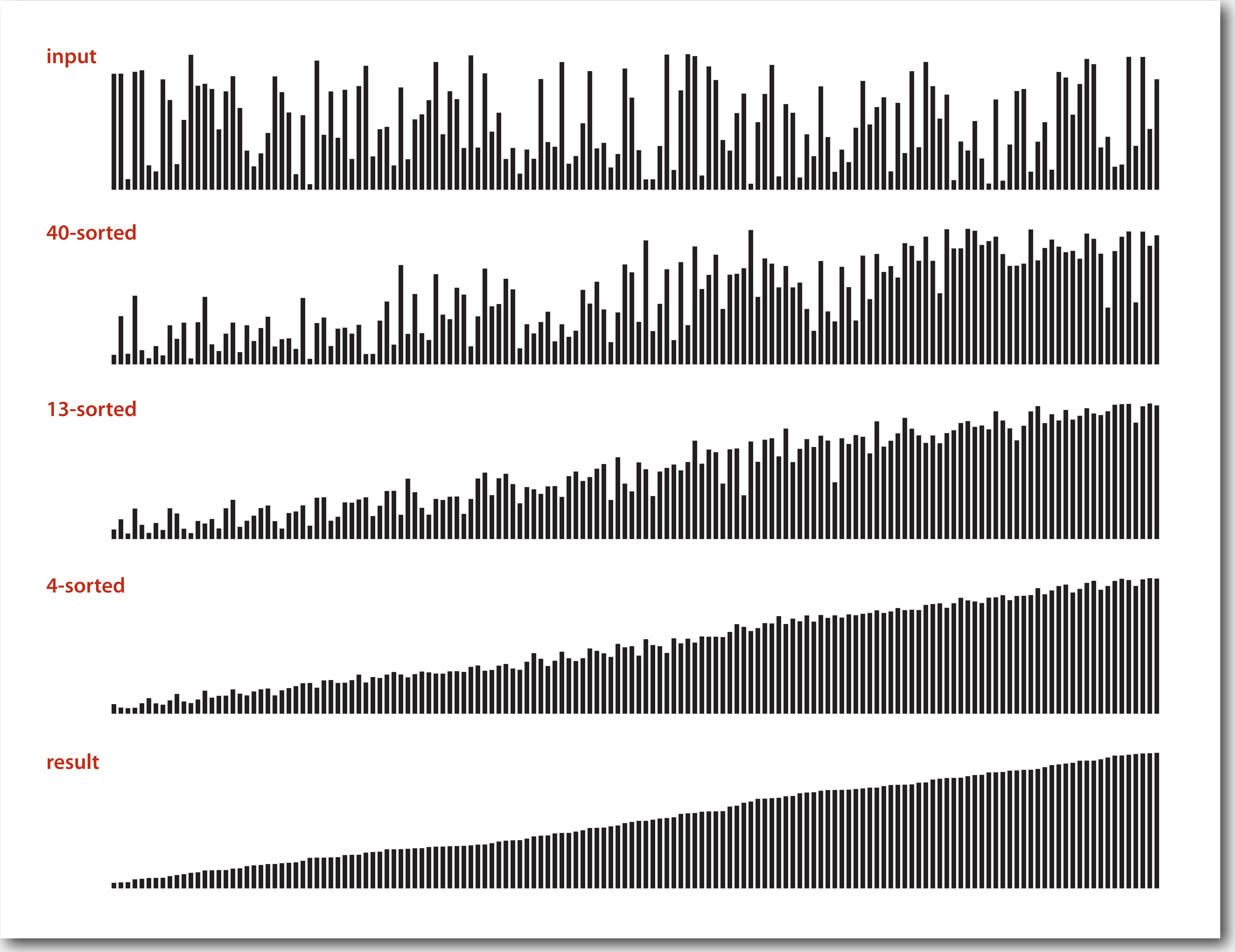
```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...

      while (h >= 1)
      {  // h-sort the array.
         for (int i = h; i < N; i++)
         {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }

         h = h/3;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static boolean void(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

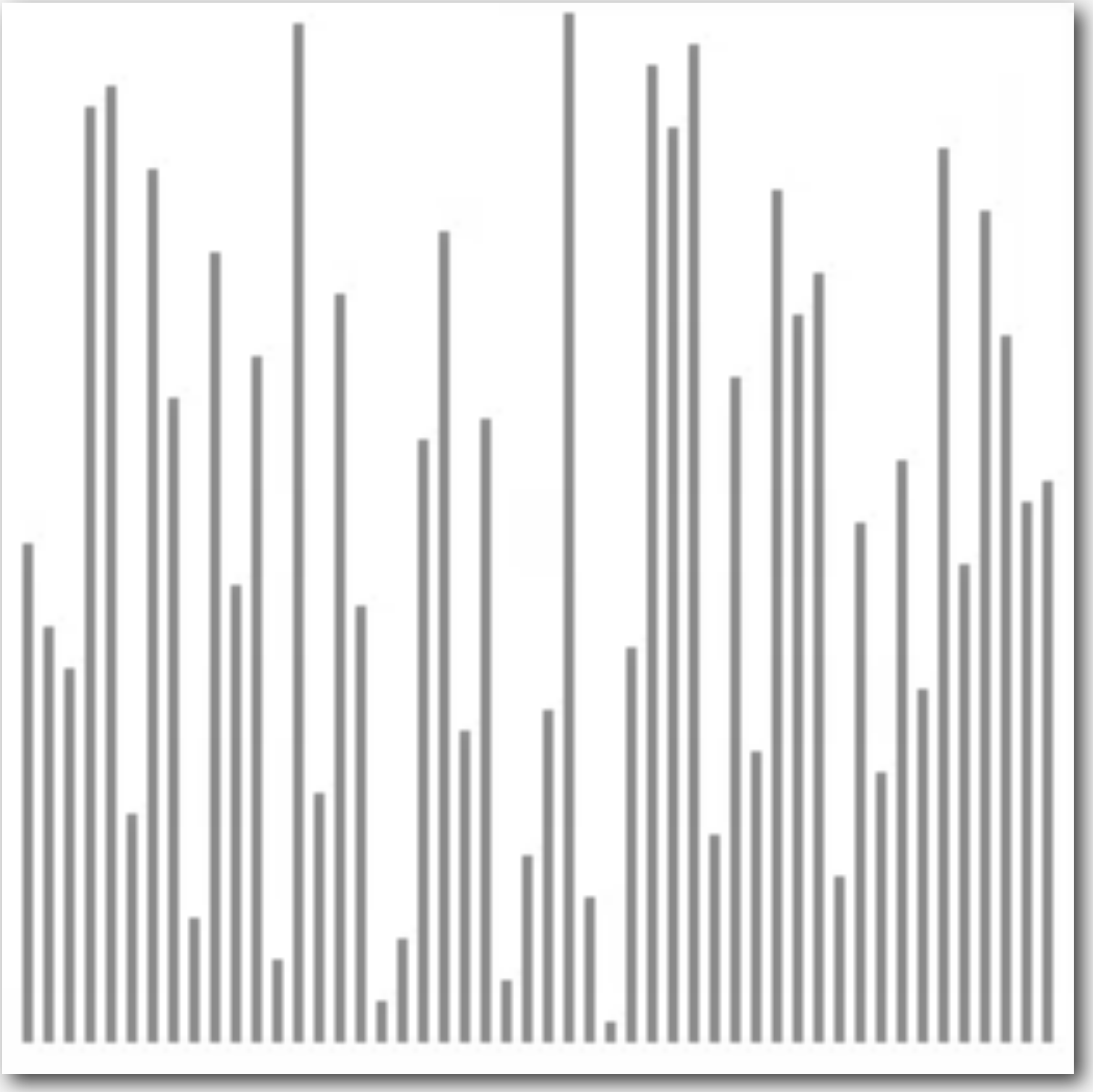
move to next increment

Shellsort: visual trace

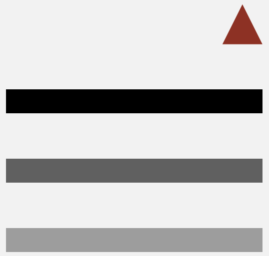


Shellsort: animation

50 random items

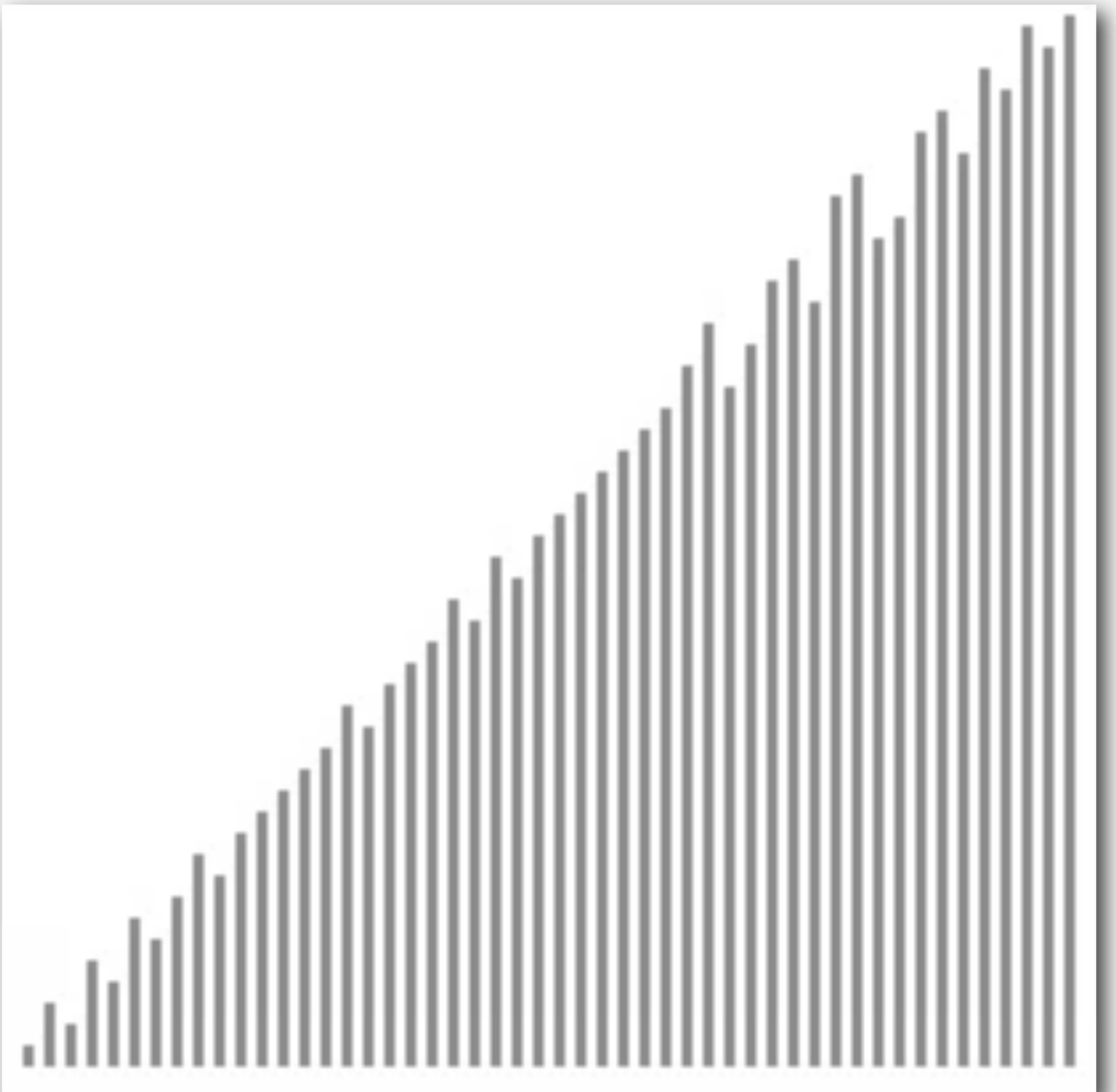


http://www.sorting-algorithms.com/shell-sort



Shellsort: animation

50 partially-sorted items



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

algorithm position

h-sorted

current subsequence

other elements

Shellsort: analysis

Proposition. The worst-case number of compares used by shellsort with the 3x+1 increments is $O(N^{3/2})$.

Property. The number of compares used by shellsort with the 3x+1 increments is at most by a small multiple of N times the # of increments used.

N	compares	$N^{1.289}$	2.5 N lg N
5.000	93	58	106
10.000	209	143	230
20.000	467	349	495
40.000	1022	855	1059
80.000	2266	2089	2257

measured in thousands

Remark. Accurate model has not yet been discovered (!)

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.

- Fast unless array size is huge.
- Tiny, fixed footprint for code (used in embedded systems).
- Hardware sort prototype.

Simple algorithm, nontrivial performance, interesting questions.

- Asymptotic growth rate?
- Best sequence of increments? ← open problem: find a better increment sequence
- Average-case performance?

Lesson. Some good algorithms are still waiting discovery.