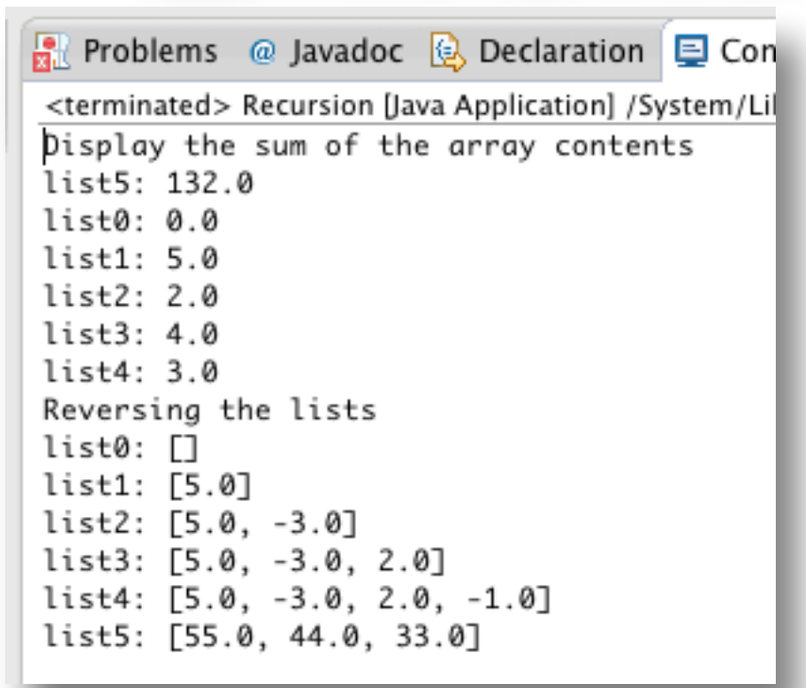
CSC 402

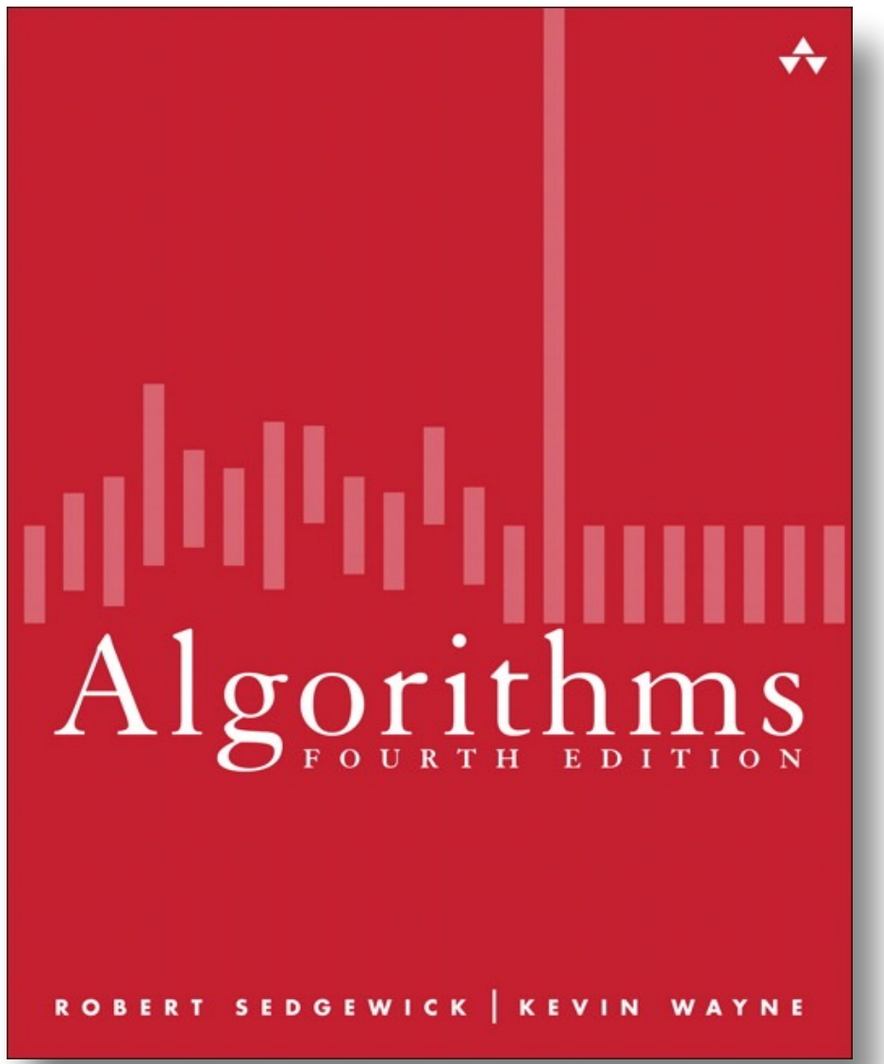
Data Structures I
Lecture 4

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a sumHelper method
- reverse (Modify the existing method)
  - **add** a reverseHelper method



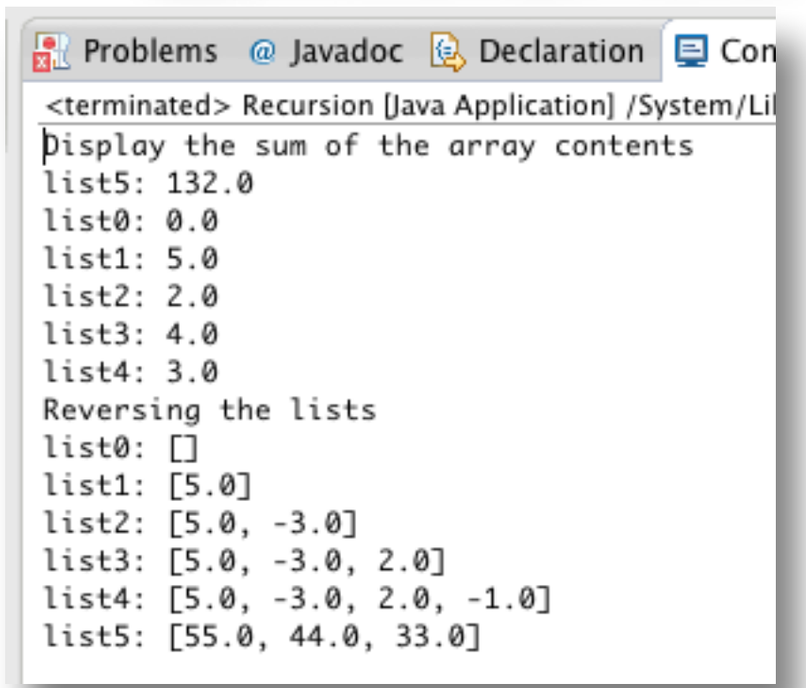


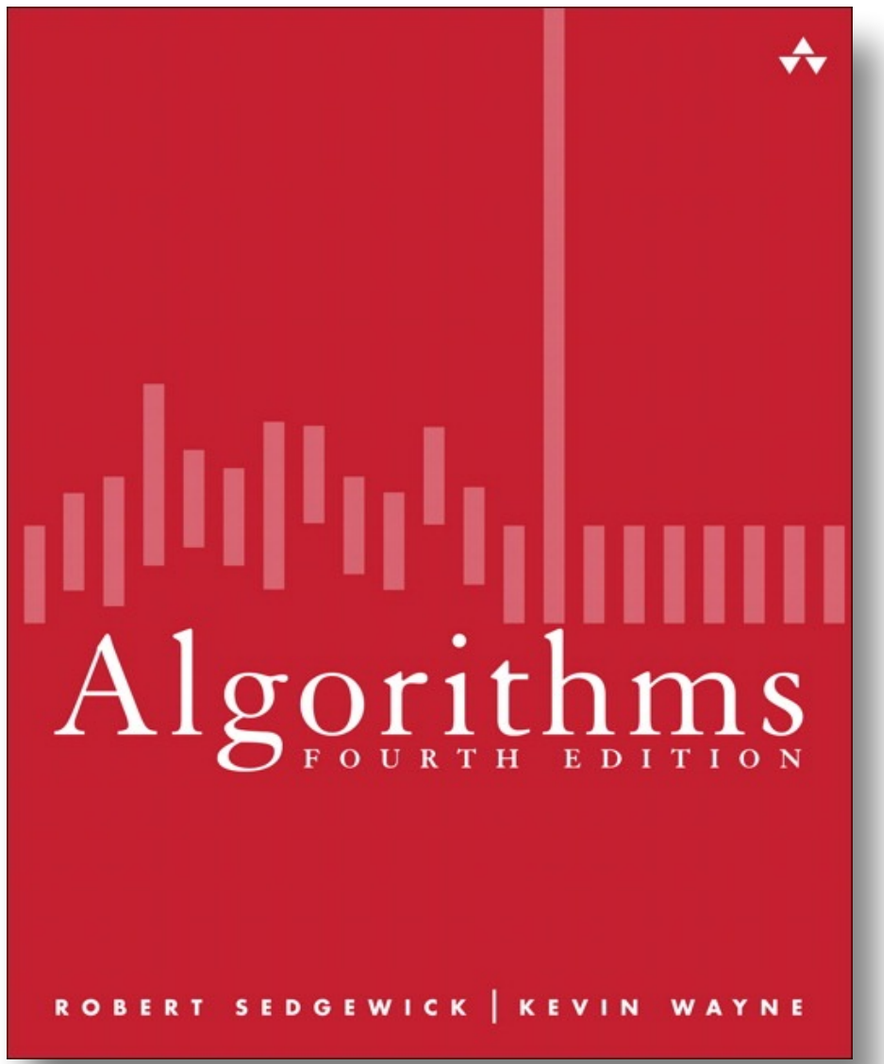
Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a sumHelper method
- reverse (Modify the existing method)
  - **add** a reverseHelper method





Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

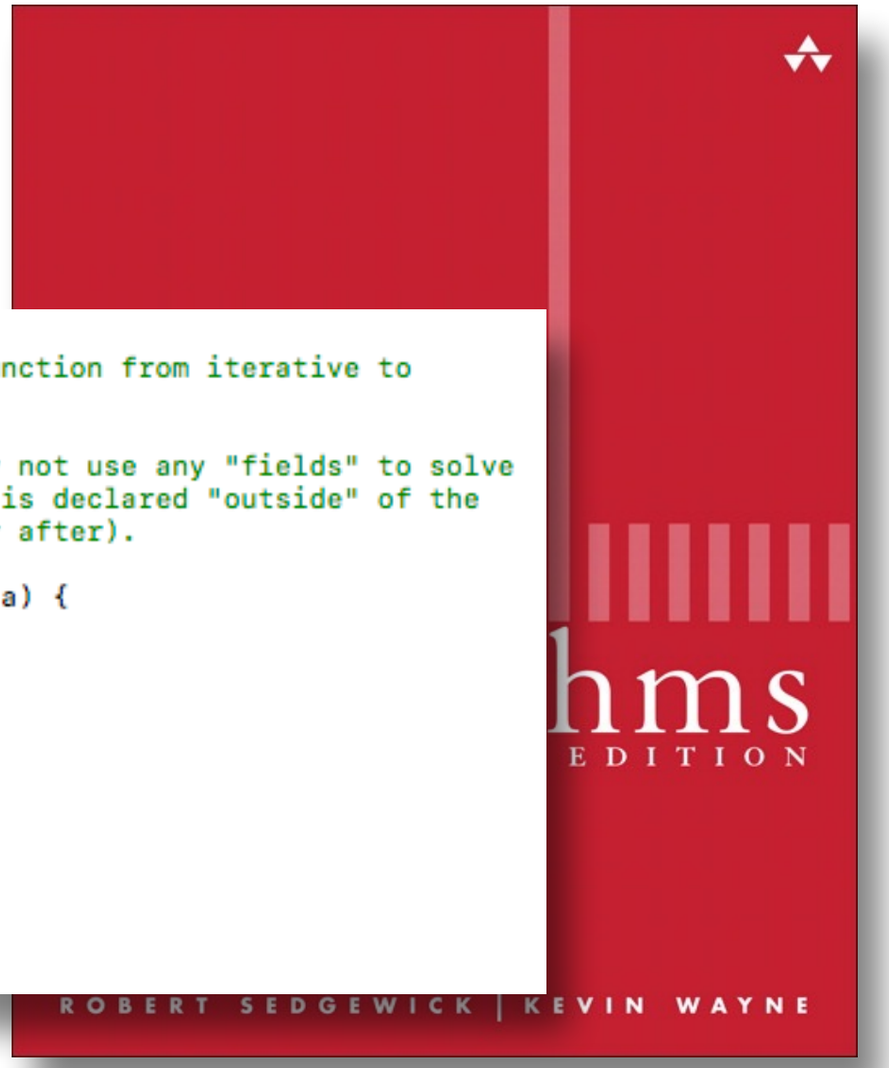
- sum (Modify the existing method)
  - **add** a su
- reverse (Mo
  - **add** a re

```
/**
 * PROBLEM 1: Translate the following sum function from iterative to
 * recursive.
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */
public static double sumIterative (double[] a) {
    double result = 0.0;
    int i = 0;
    while (i < a.length) {
        result = result + a[i];
        i = i + 1;
    }
    return result;
}
public static double sum (double[] a) {
    return 0; // TODO
}
```

Problems @ Ja

<terminated> Recur

```
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the li
list0: []
list1: [5.0]
list2: [5.0, -3.0]
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```



Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion java application to provide valid logic

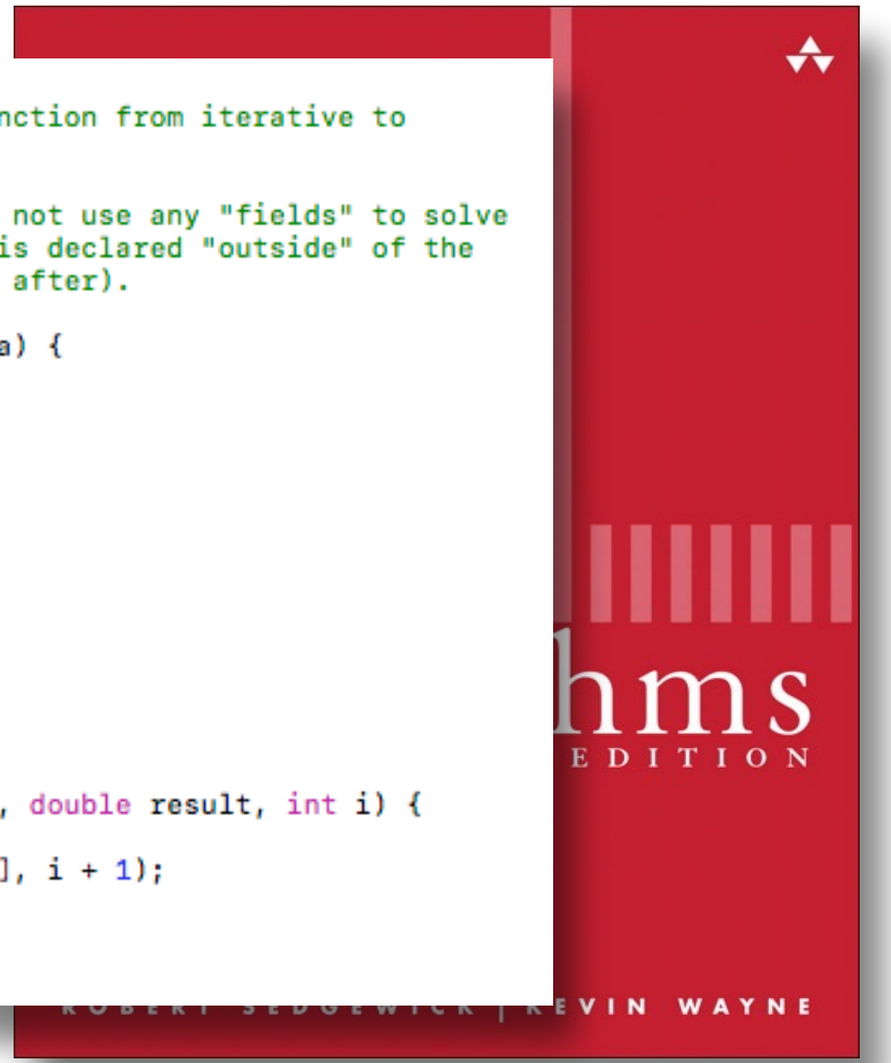
- sum (Modify
  - **add** a su
- reverse (Mod
  - **add** a re

```
/**
 * PROBLEM 1: Translate the following sum function from iterative to
 * recursive.
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */
public static double sumIterative (double[] a) {
    double result = 0.0;
    int i = 0;
    while (i < a.length) {
        result = result + a[i];
        i = i + 1;
    }
    return result;
}

// Example 0
public static double sum (double[] a) {
    return sum0Helper (a, 0.0, 0);
}
private static double sum0Helper (double[] a, double result, int i) {
    if (i < a.length) {
        result = sum0Helper (a, result + a[i], i + 1);
    }
    return result;
}
```

Problems @ Ja

<terminated> Recur

```
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the li
list0: []
list1: [5.0]
list2: [5.0, -3.0]
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```

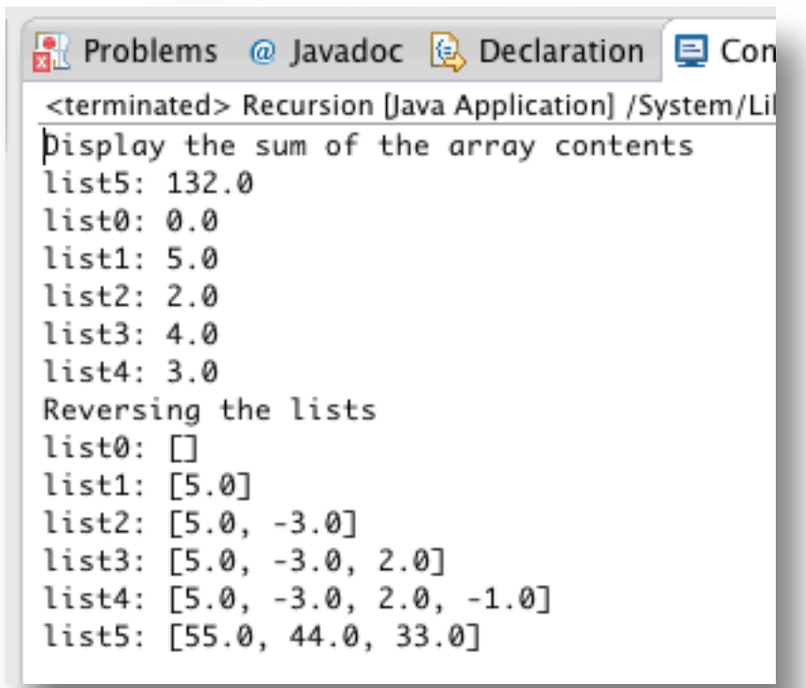


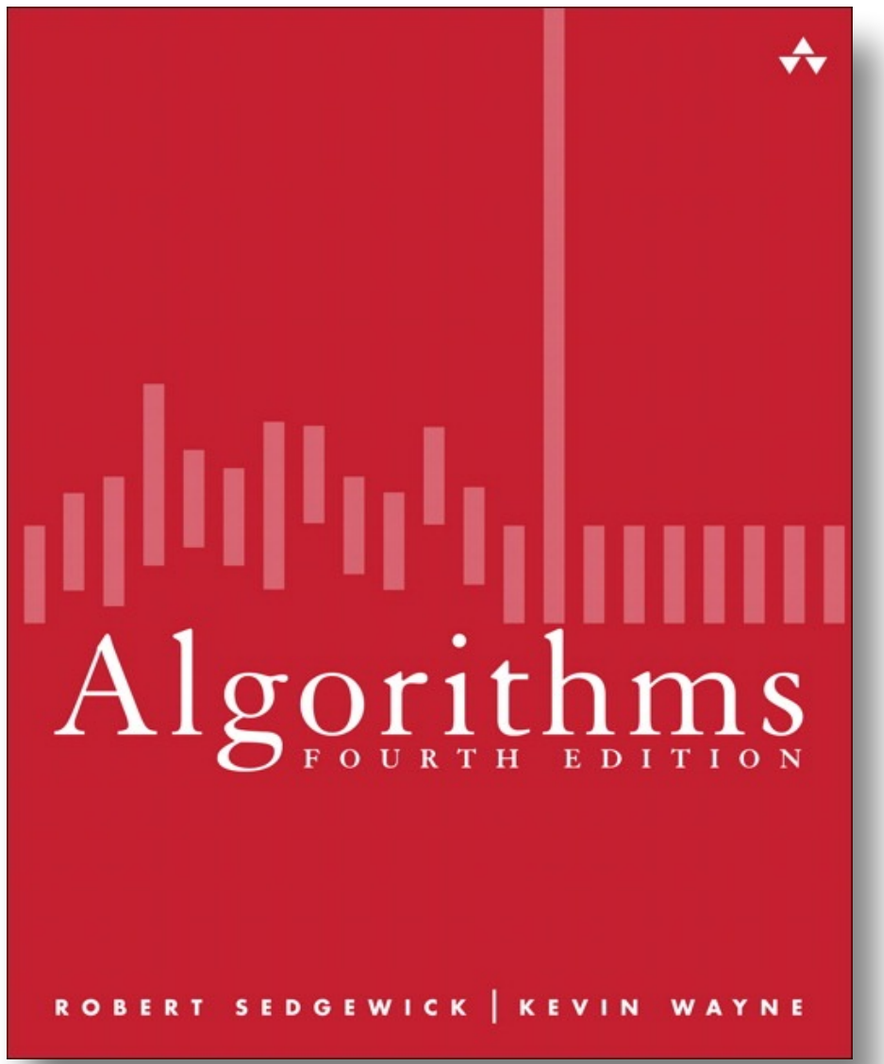
Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a sumHelper method
- reverse (Modify the existing method)
  - **add** a reverseHelper method





Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a su
- reverse (Mo
  - **add** a re

```
/**
 * PROBLEM 2: Do the same translation for this in-place reverse function
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */
public static void reverseIterative (double[] a) {
    int hi = a.length - 1;
    int lo = 0;
    while (lo < hi) {
        double loVal = a[lo];
        double hiVal = a[hi];
        a[hi] = loVal;
        a[lo] = hiVal;
        lo = lo + 1;
        hi = hi - 1;
    }
}
public static void reverse (double[] a) {
    // TODO
}
```

```
Problems @ Ja
<terminated> Recu
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the l
list0: []
list1: [5.0]
list2: [5.0, -3.0]
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursi
provide valid logic

- sum (Modify
  - **add** a su
- reverse (Moc
  - **add** a re

```
/**
 * PROBLEM 2: Do the same translation for this in-place reverse function
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */

public static void reverseIterative (double[] a) {
    int hi = a.length - 1;
    int lo = 0;
    while (lo < hi) {
        double loVal = a[lo];
        double hiVal = a[hi];
        a[hi] = loVal;
        a[lo] = hiVal;
        lo = lo + 1;
        hi = hi - 1;
    }
}

public static void reverse (double[] a) {
    reverseHelper (a, 0, a.length - 1);
}
private static void reverseHelper (double[] a, int lo, int hi) {
    if (lo < hi) {
        double loVal = a[lo];
        double hiVal = a[hi];
        a[lo] = hiVal;
        a[hi] = loVal;
        reverseHelper (a, lo + 1, hi - 1);
    }
}
```

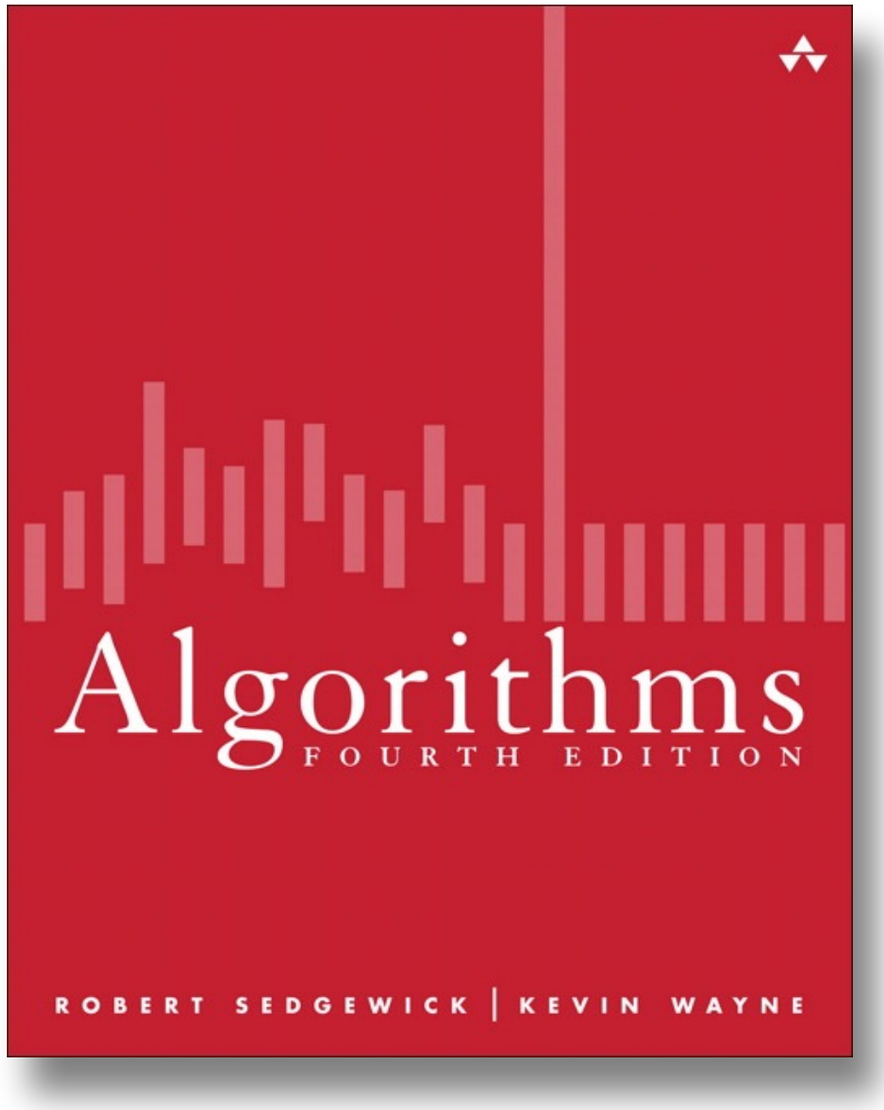
Problems @ Ja

<terminated> Recur

```
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the li
list0: []
list1: [5.0]
list2: [5.0, -3.
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```

Assignment due next Monday at 11:59 PM

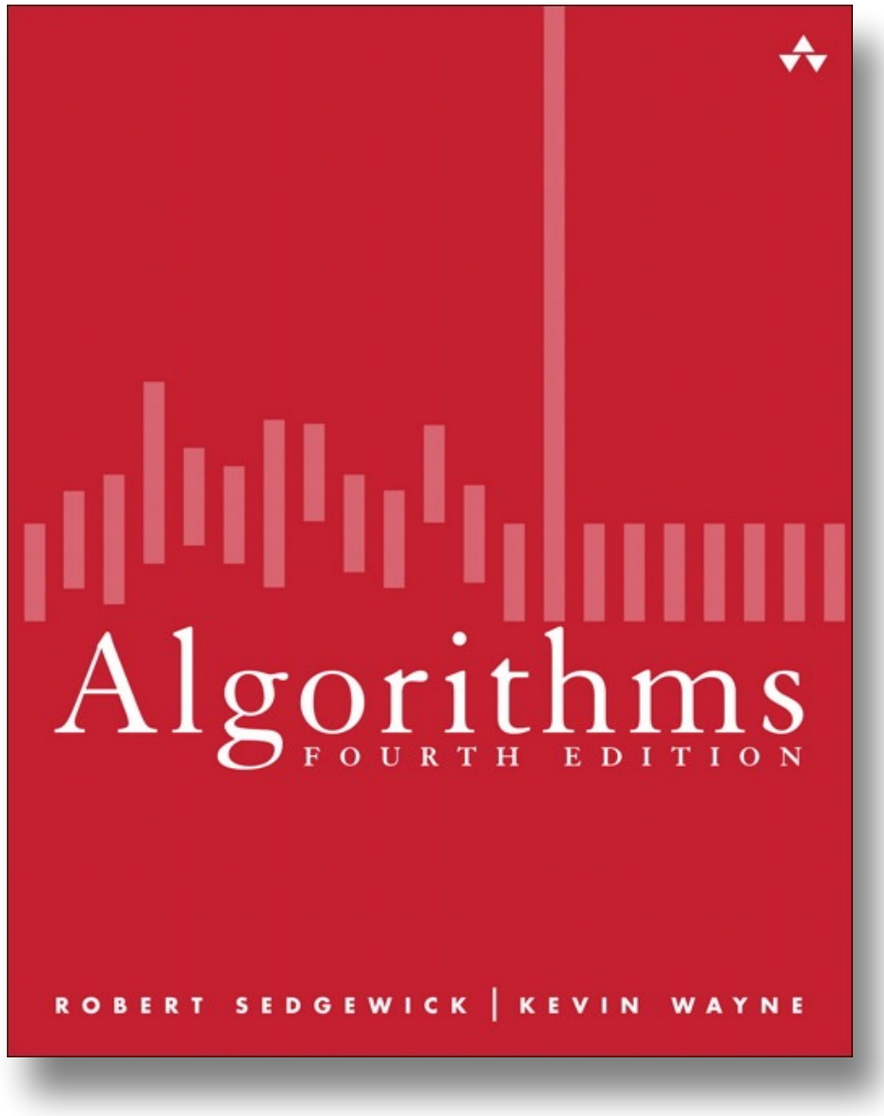
Lecture Overview



2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

Lecture Overview



2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

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

Sorting problem

Our primary concern is algorithms for rearranging arrays of items where each item contains a key. The objective is to rearrange the items such that their keys are in ascending order. In Java, the abstract notion of a key is captured in a built-in mechanism the ***Comparable*** interface. With but a few exceptions, our sort code refers to the data only through two operations: the method less() that compares objects and the method exch() that exchanges them.

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

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

Sorting problem

Sorting cost model. When studying sorting algorithms, we count compares and exchanges. For algorithms that do not use exchanges, we count array accesses.

Extra memory. The sorting algorithms we consider divide into two basic types: those that sort in place (no extra memory except perhaps for a small function-call stack or a constant number of instance variables), and those that need enough extra memory to hold another copy of the array to be sorted.

Types of data. Our sort code is effective for any type of data that implements Java's ***Comparable*** interface. This means that there is a method compareTo() for which v.compareTo(w) returns an integer that is negative, zero, or positive when v < w, v = w, or v > w, respectively. The method must implement a total order:

- Reflexive: for all v, v = v.
- Antisymmetric: for all v and w, if (v < w) then (w > v); and if (v = w) then (w = v).
- Transitive: for all v, w, and x, if (v ≤ w) and (w ≤ x), then v ≤ x..

In addition, v.compareTo(w) must throw an exception if v and w are of incompatible types or if either is null. Date.java illustrates how to implement the Comparable interface for a user-defined type.

Sorting problem

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
```

Sorting problem

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]
```

Sorting problem

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
```

Sorting problem

Goal. Sort any type of data (for which sorting is well defined).

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

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

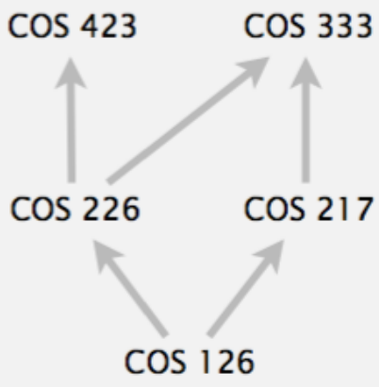
Ex.

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

No transitivity. Rock-paper-scissors.

No totality. PU course prerequisites.

violates transitivity



violates totality

Sorting problem

Goal. Sort any type of data (for which sorting is well defined).

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 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.

Sorting problem

client

```
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]);
   }
}
```

data-type implementation

```
public class String
implements Comparable<String>
{
   ...
   public int compareTo(String 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 String data type

Sorting problem

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

- Defines 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.

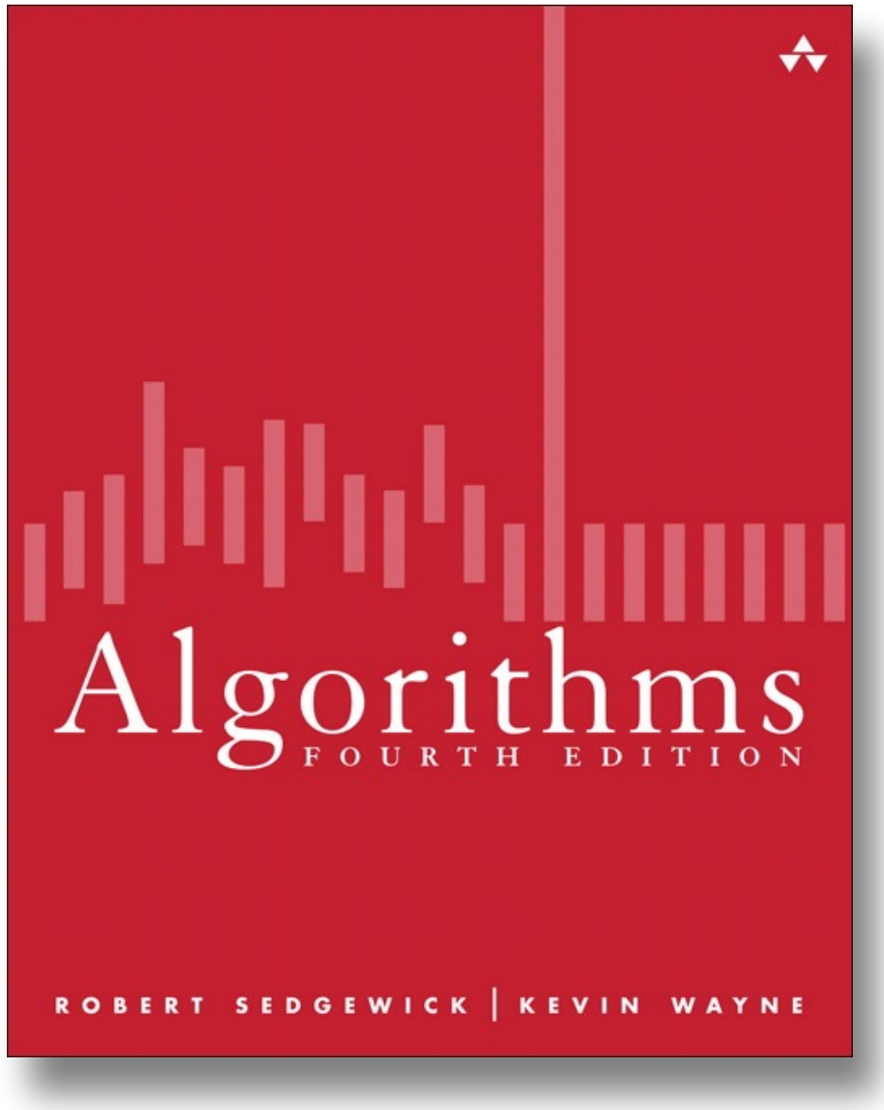
Sorting problem

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

Lecture Overview



2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

Sorting problem

Selection sort. One of the simplest sorting algorithms works as follows: First, find the smallest item in the array, and exchange it with the first entry. Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted. This method is called selection sort because it works by repeatedly selecting the smallest remaining item. Selection.java is an implementation of this method.

Selection sort uses ~N2/2 compares and N exchanges to sort an array of length N.

		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)

Selection Sort Demo

Selection sort demo

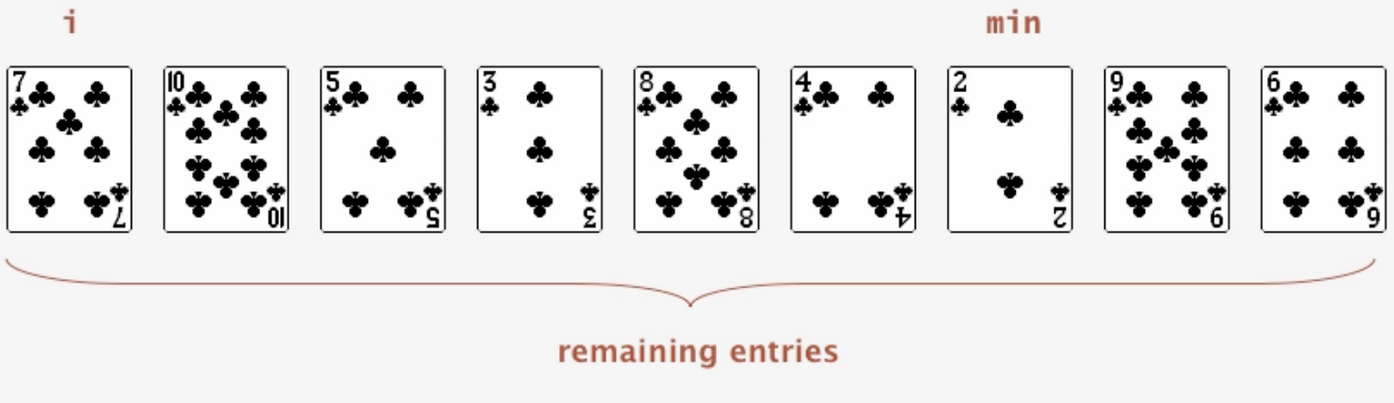
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.

initial

Selection Sort Demo

Selection sort demo

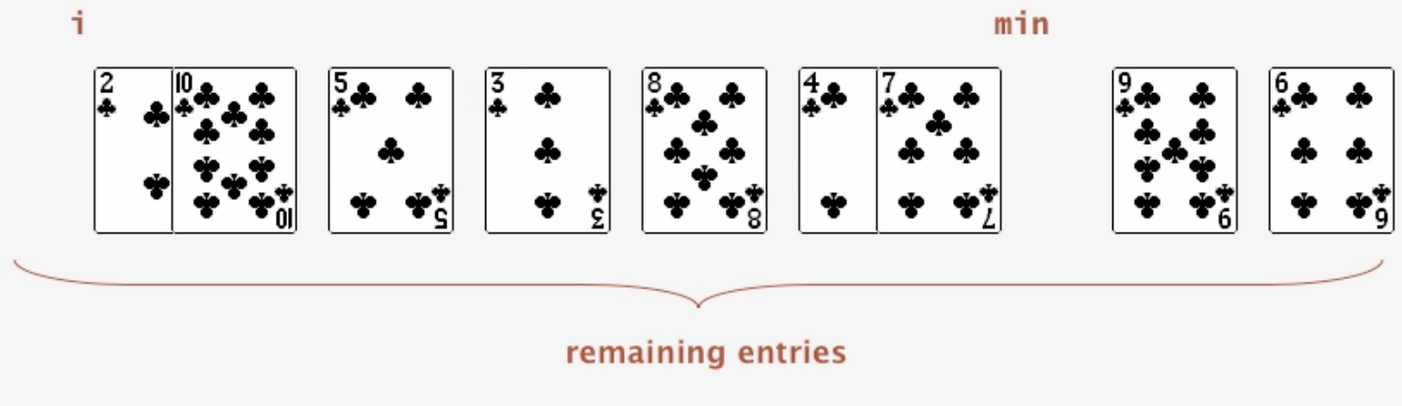
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

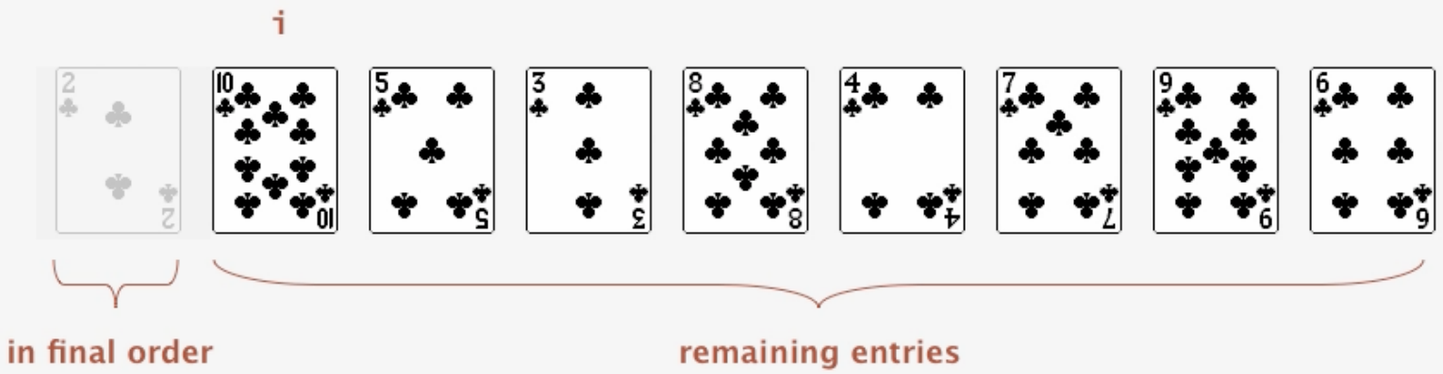
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

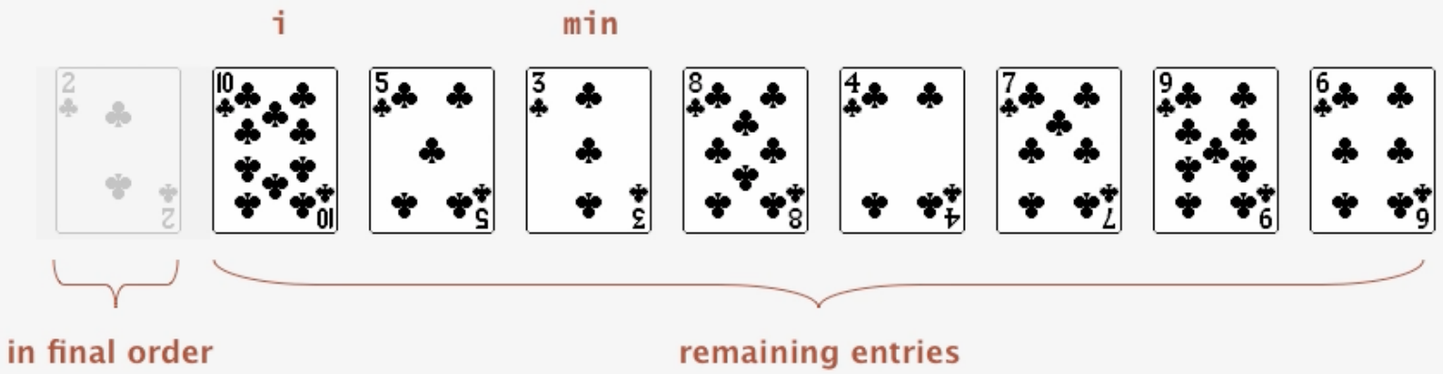
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

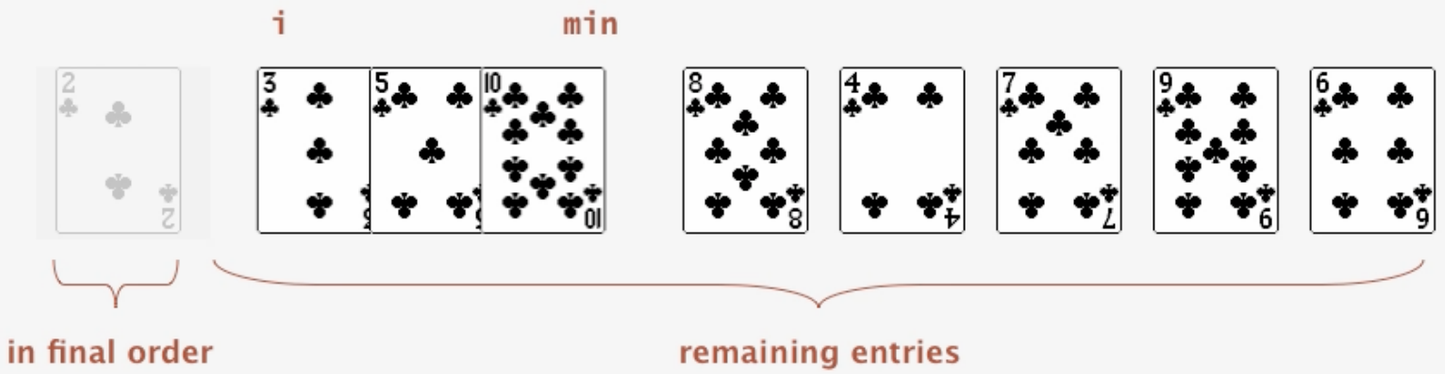
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

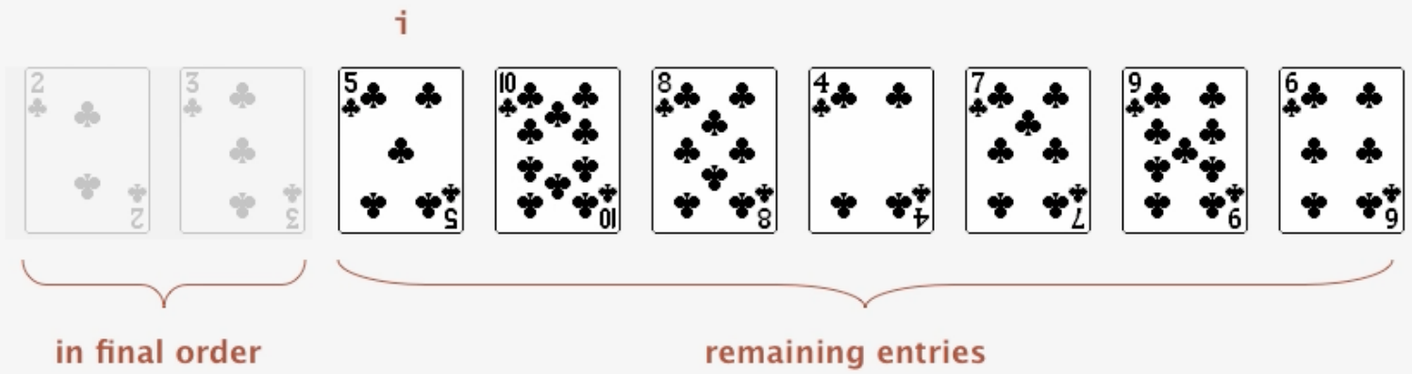
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

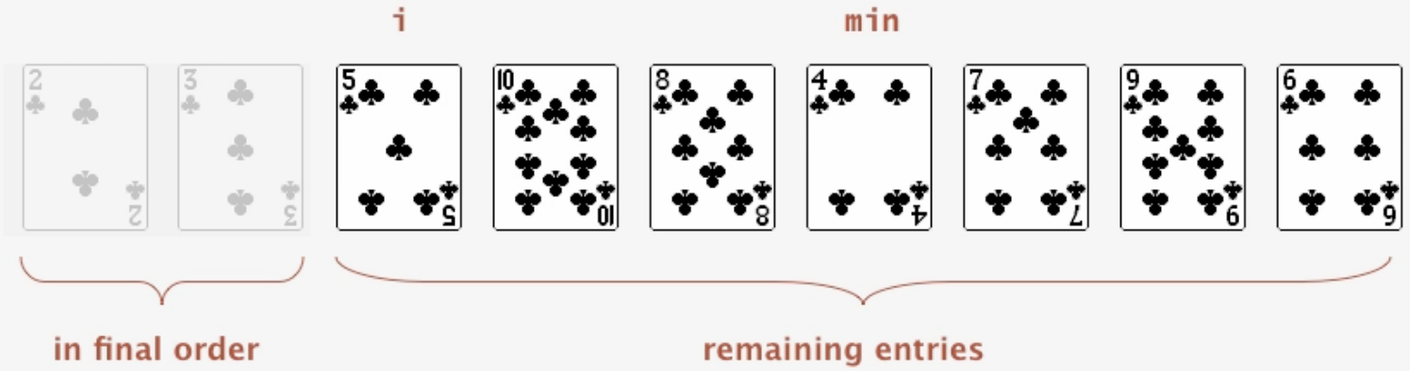
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

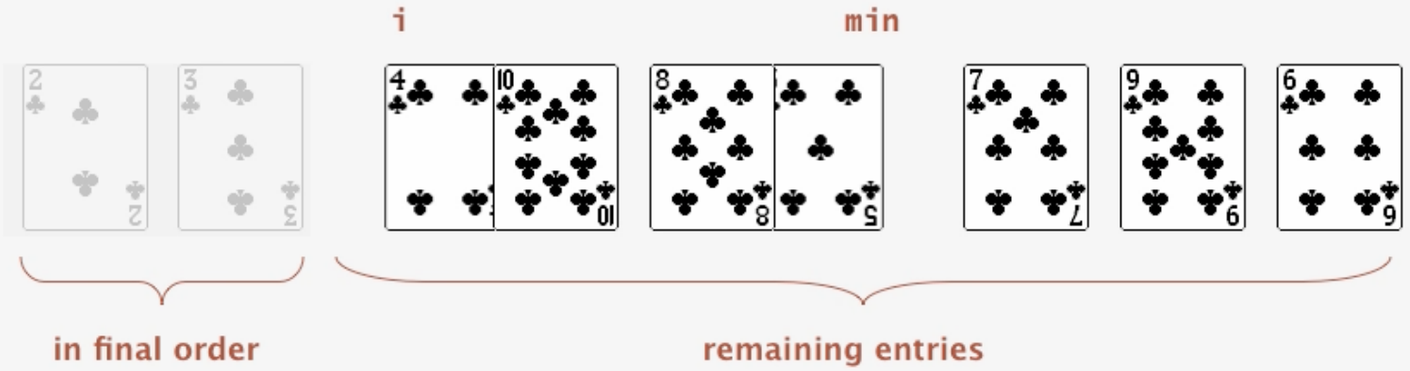
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

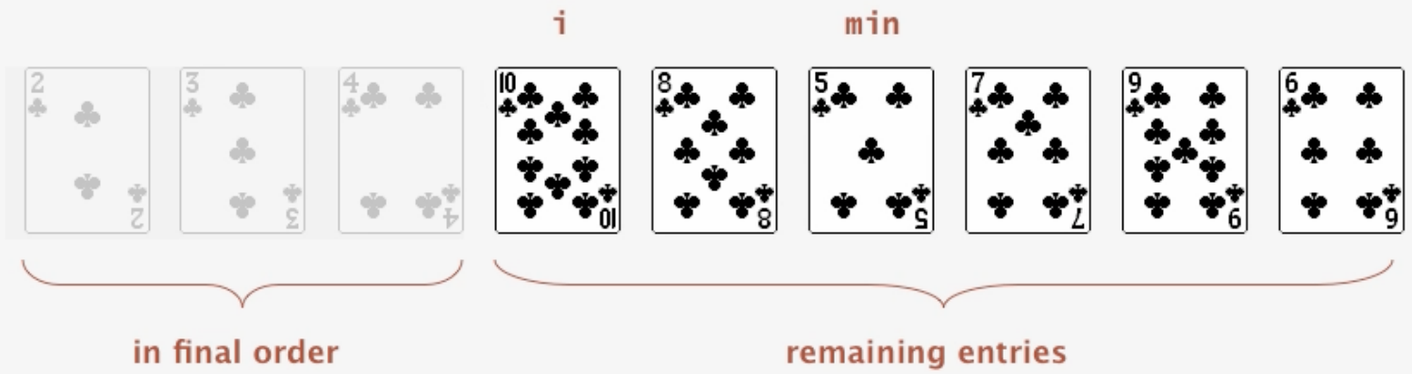
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

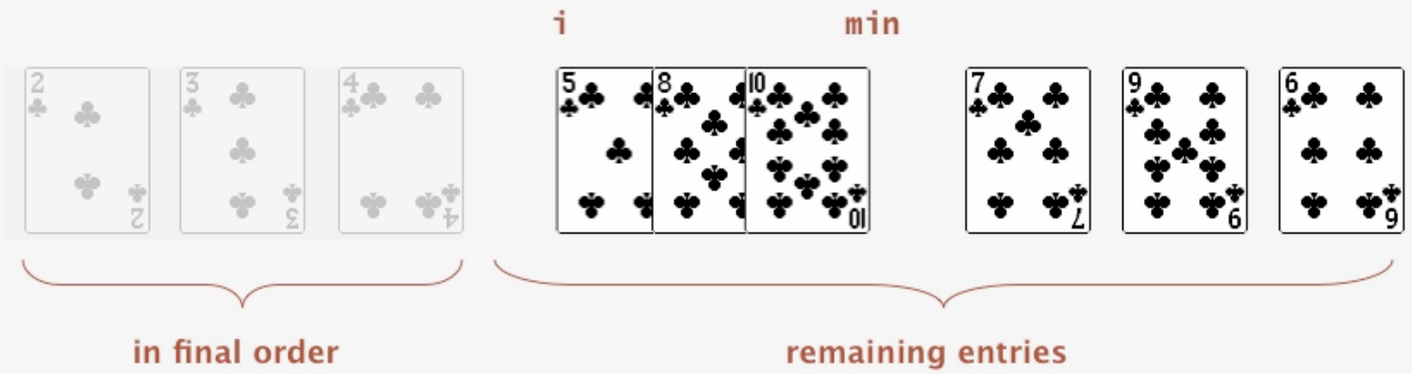
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

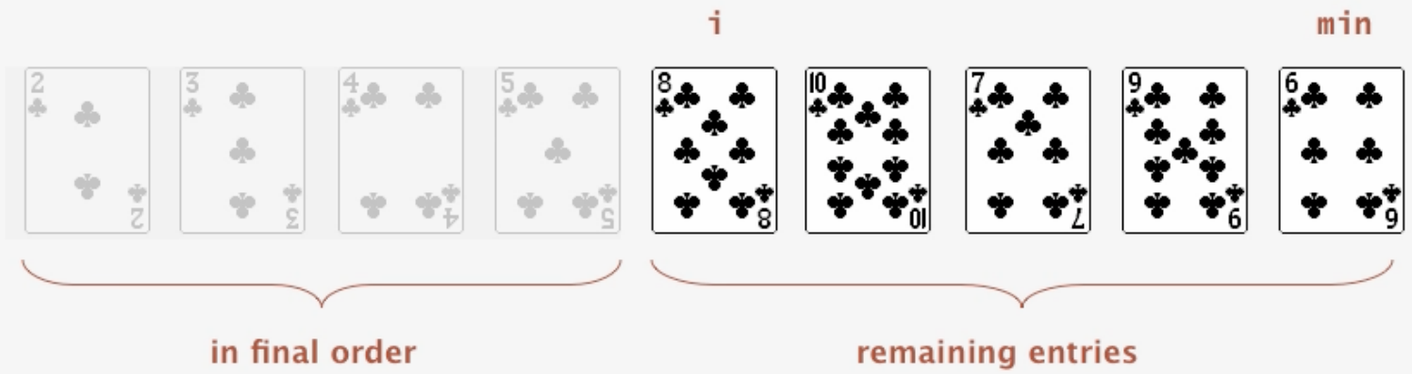
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

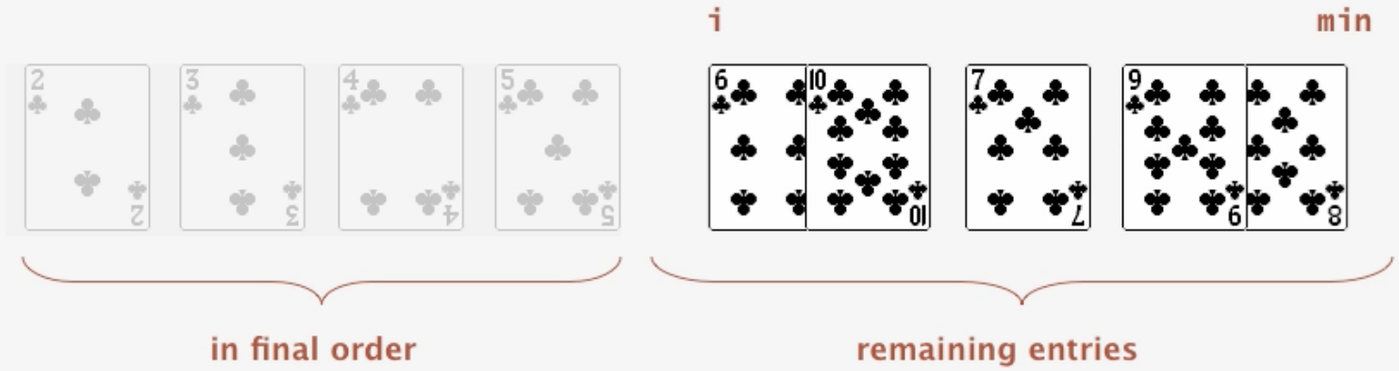
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

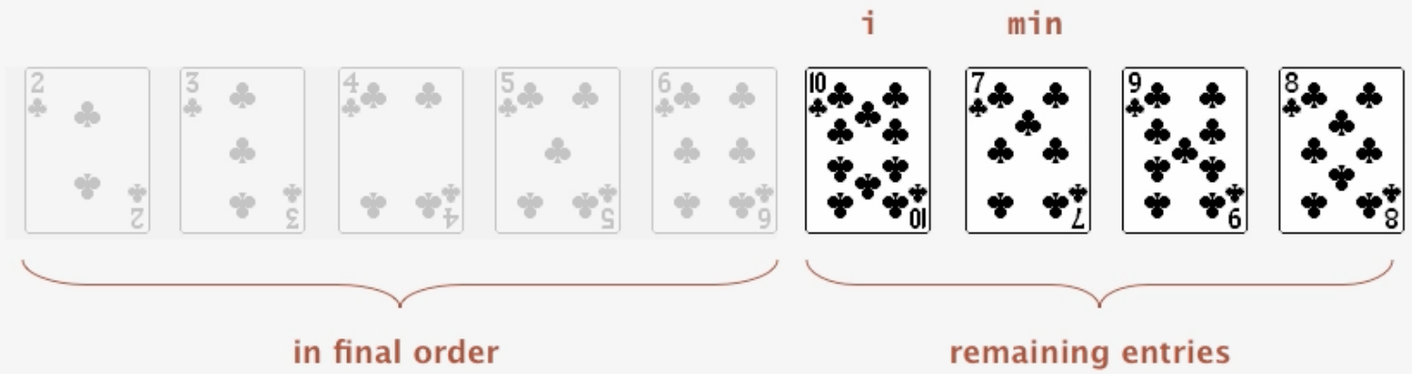
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

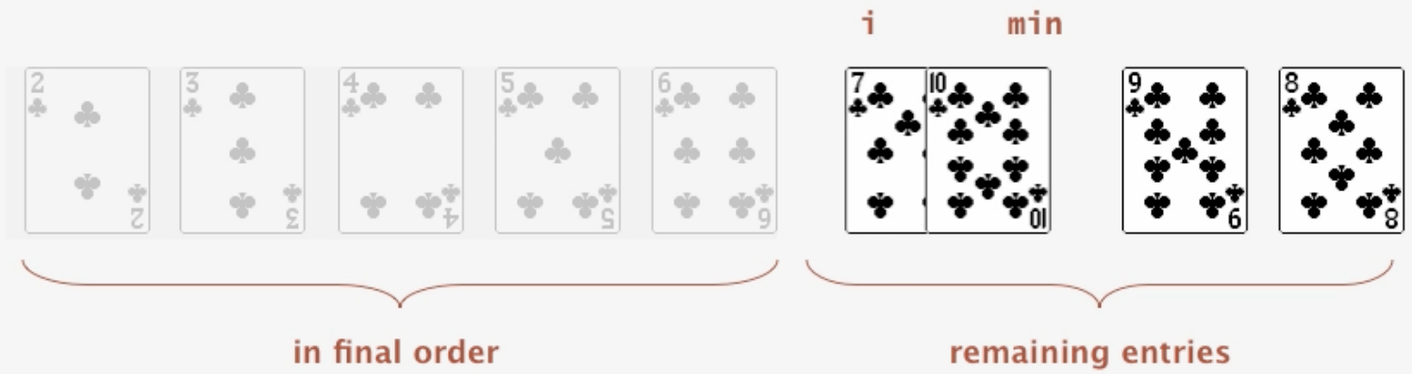
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

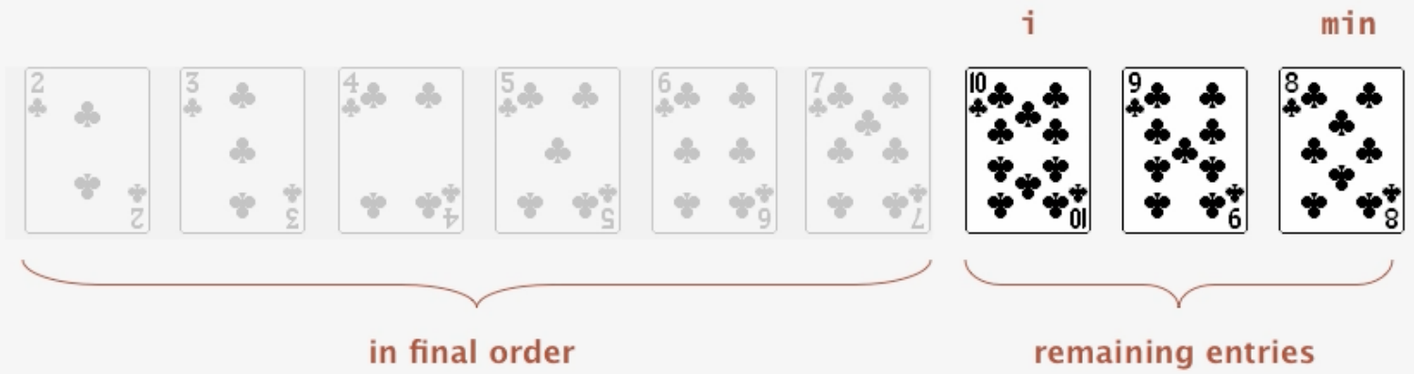
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

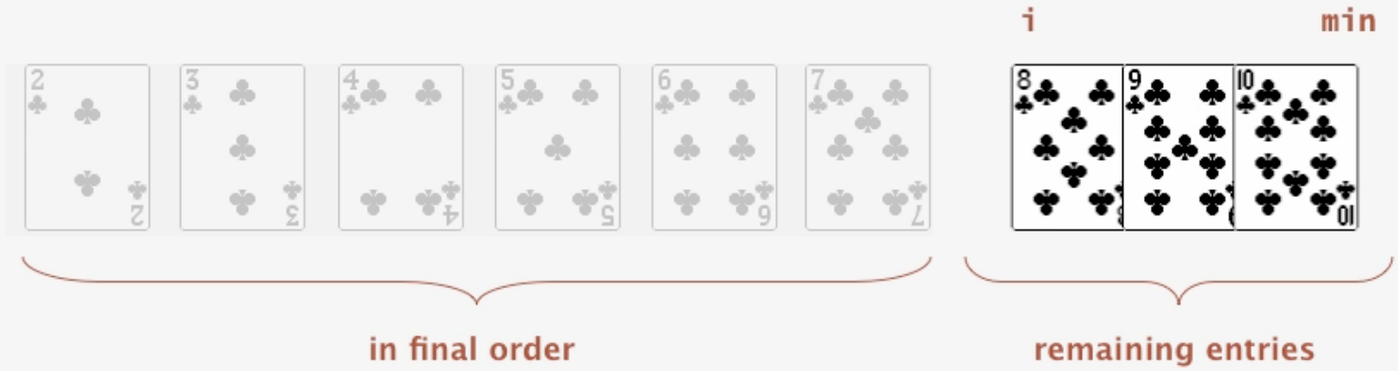
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

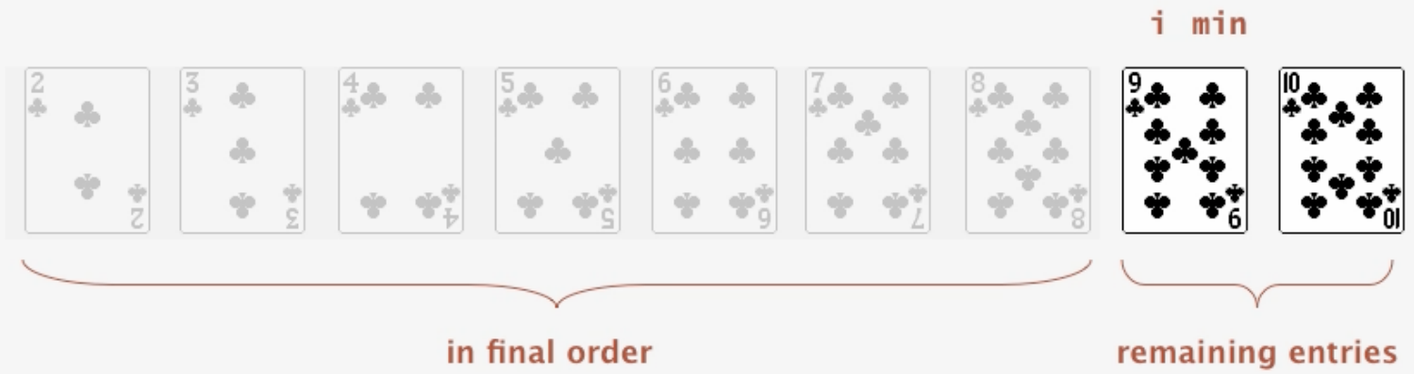
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

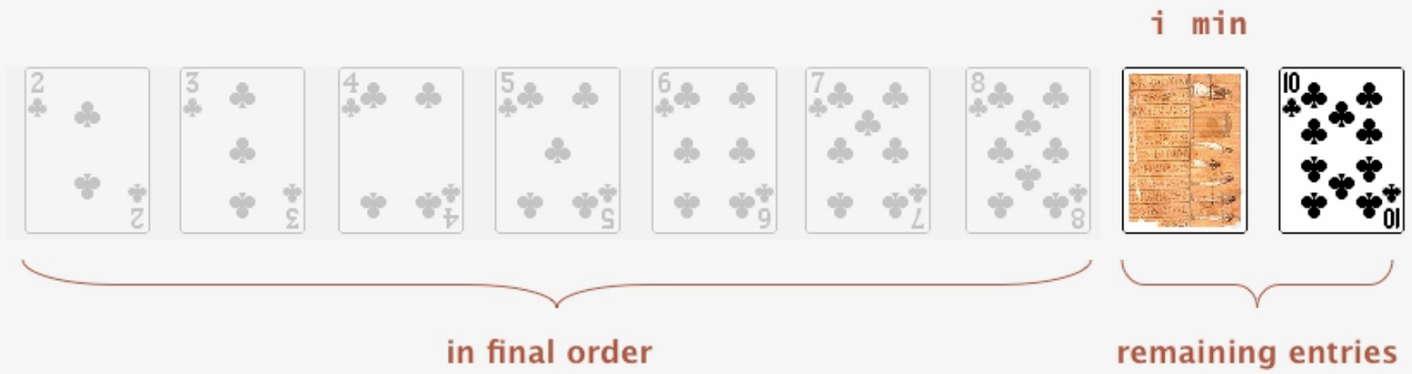
- In iteration `i`, find index `min` of smallest remaining entry.
- Swap `a[i]` and `a[min]`.



Selection Sort Demo

Selection sort demo

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



Selection Sort Demo

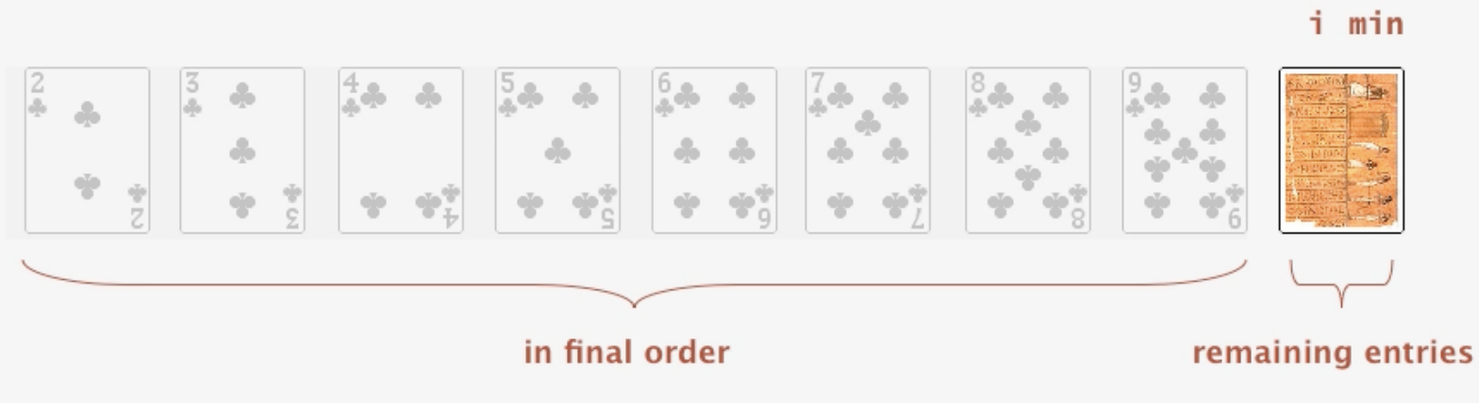
Selection sort demo

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

Selection Sort Demo

Selection sort demo

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



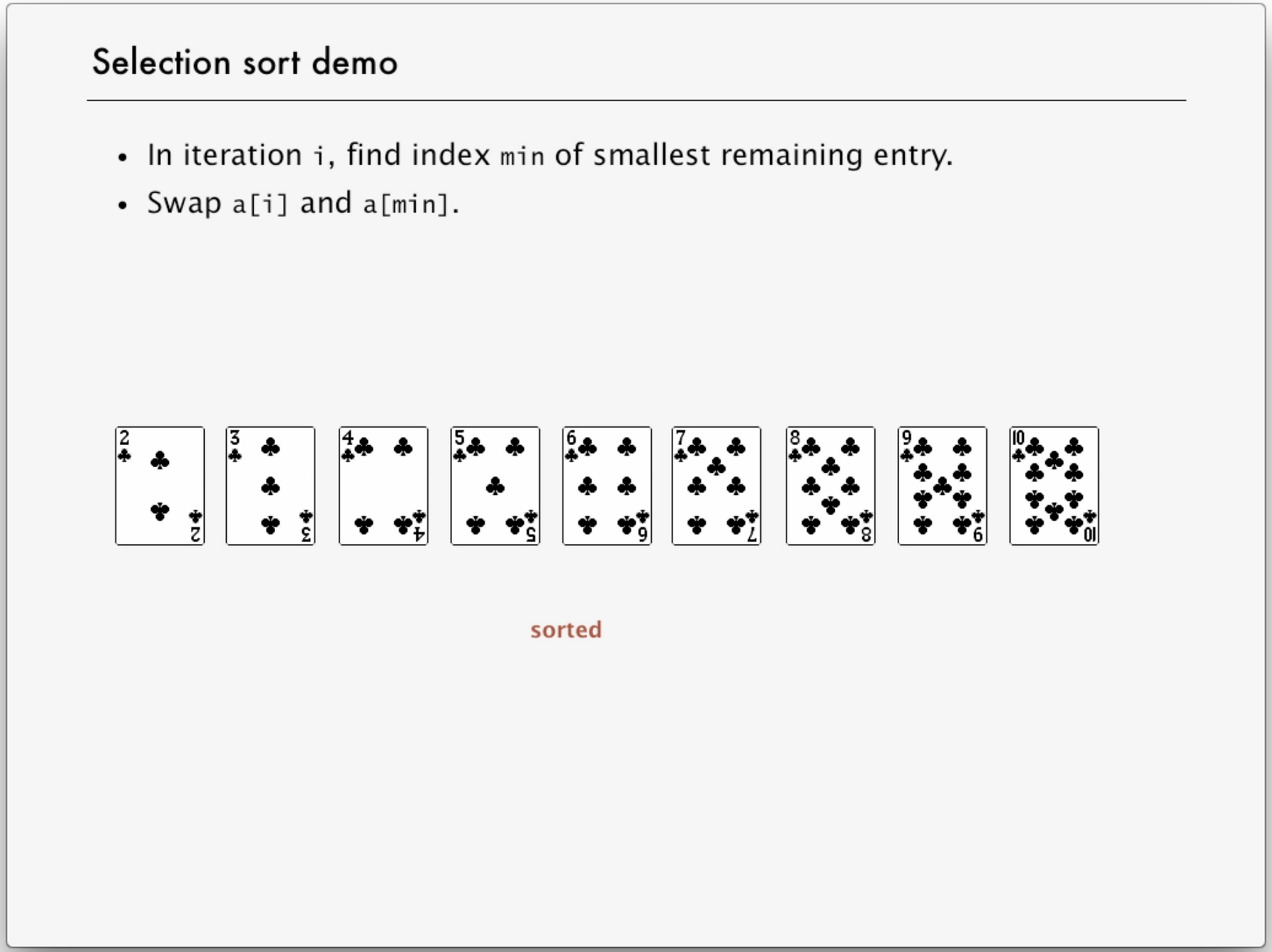
Selection Sort Demo

Selection sort demo

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

in final order

Selection Sort Demo

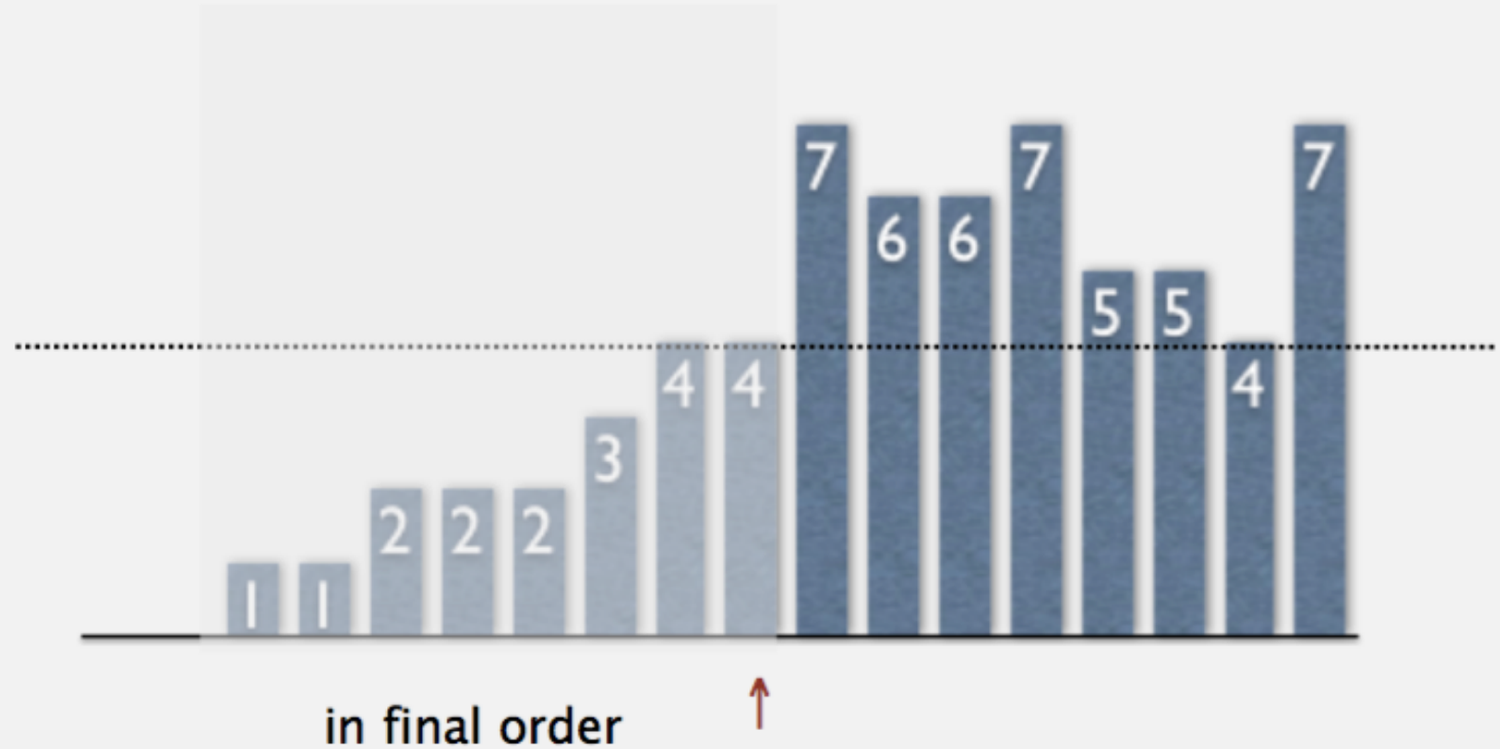


Selection Sort

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

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;
}
```

Selection Sort

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;
```

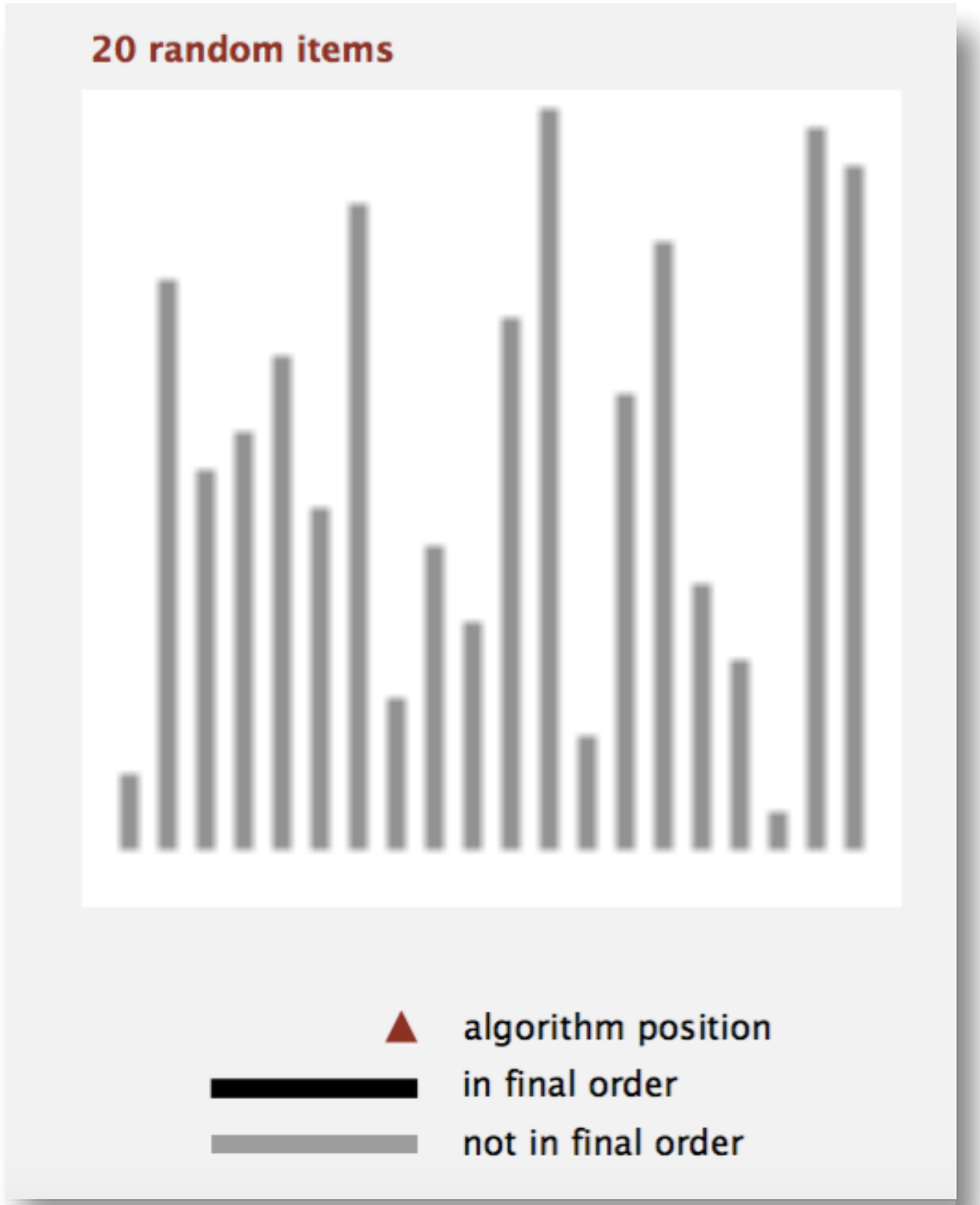
- Exchange into position.

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

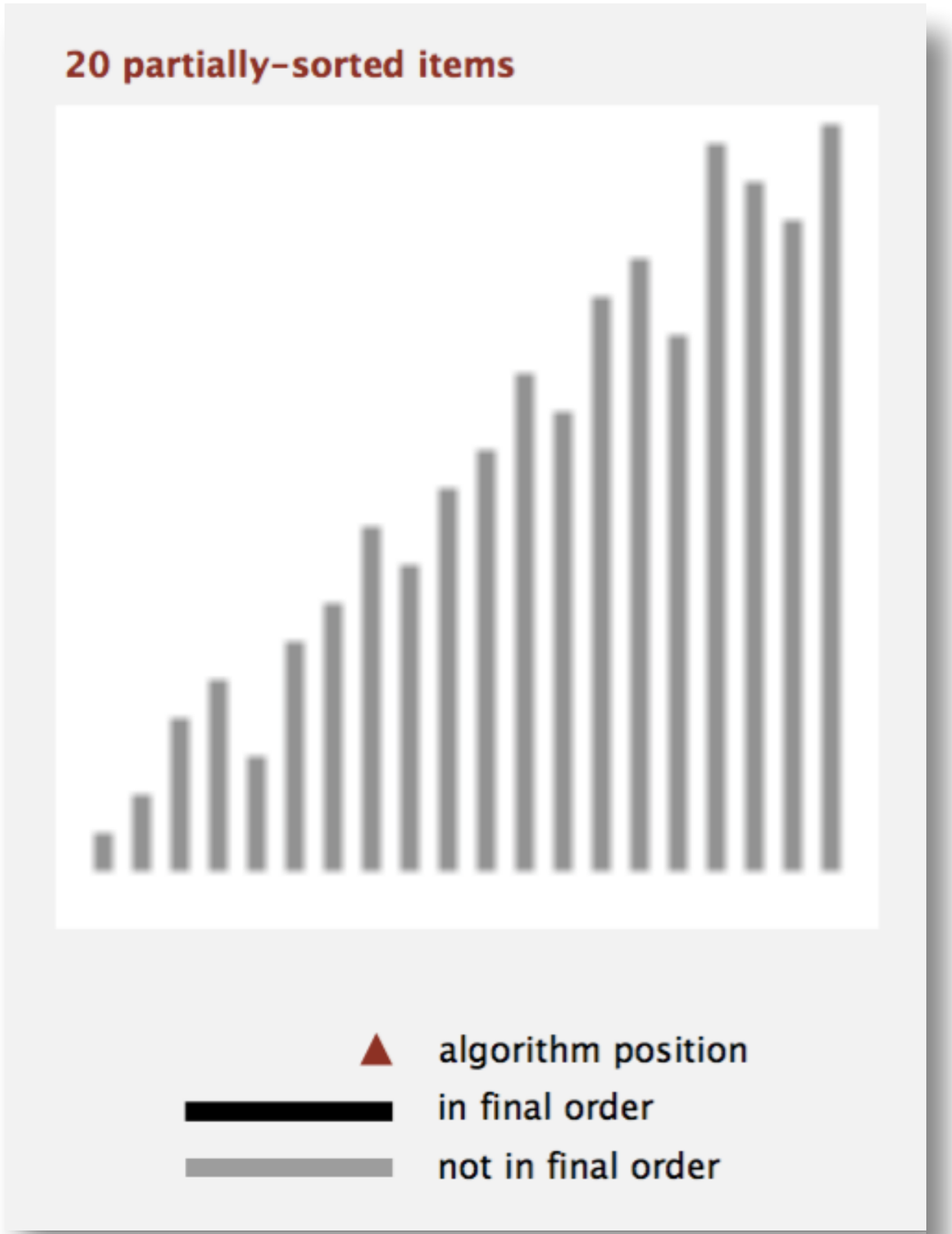
Selection Sort

```
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 */  }
}
```

Selection Sort



Selection Sort



Selection Sort

Proposition. Selection sort uses $(N-1)+(N-2)+\ ...\ +1+0 \sim N^2/2$ 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]

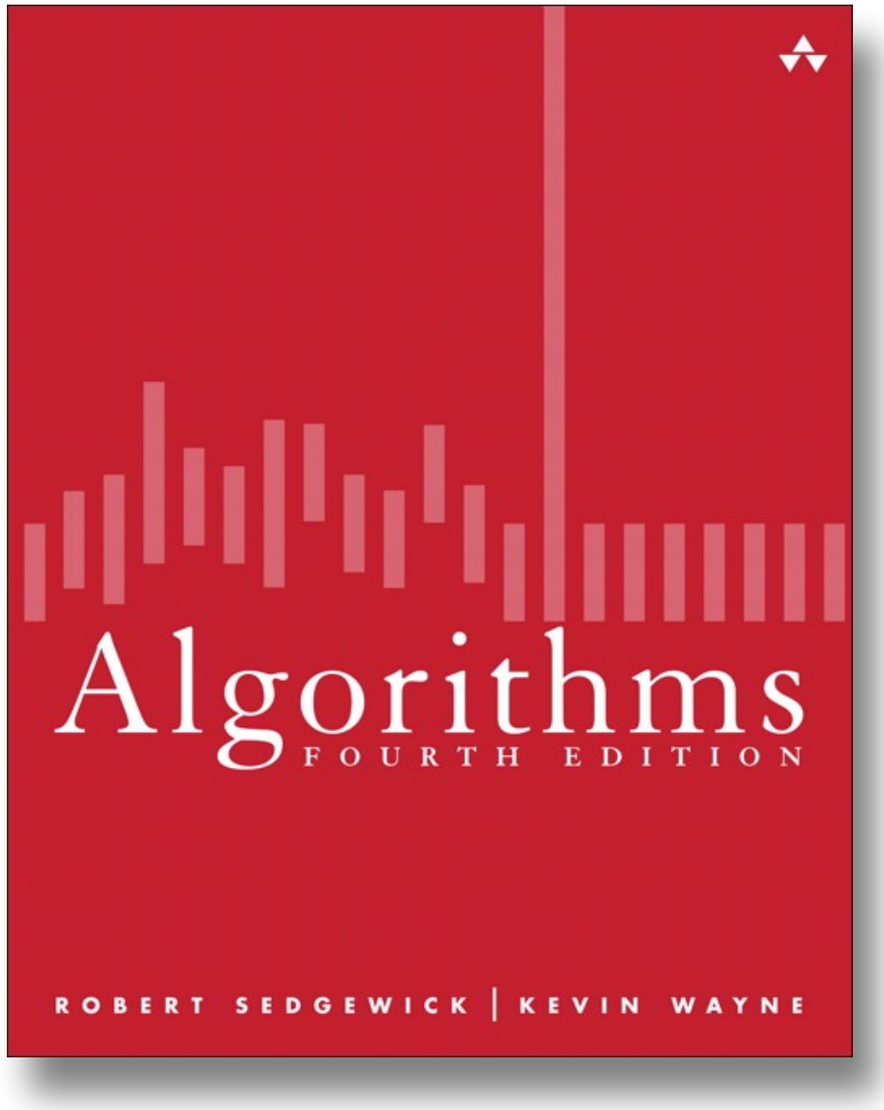
entries in gray are in final position

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

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

Data movement is minimal. Linear number of exchanges.

Lecture Overview

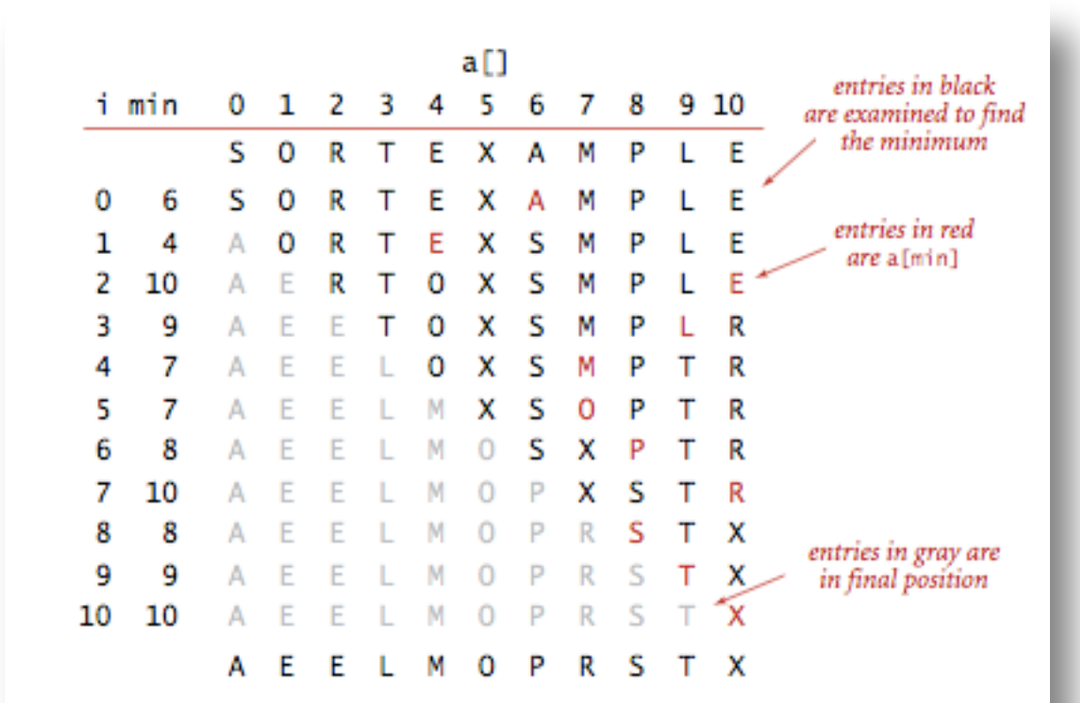


2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

Insertion Sort

Insertion sort. The algorithm that people often use to sort bridge hands is to consider the cards one at a time, inserting each into its proper place among those already considered (keeping them sorted). In a computer implementation, we need to make space for the current item by moving larger items one position to the right, before inserting the current item into the vacated position. Insertion.java is an implementation of this method, which is called insertion sort.

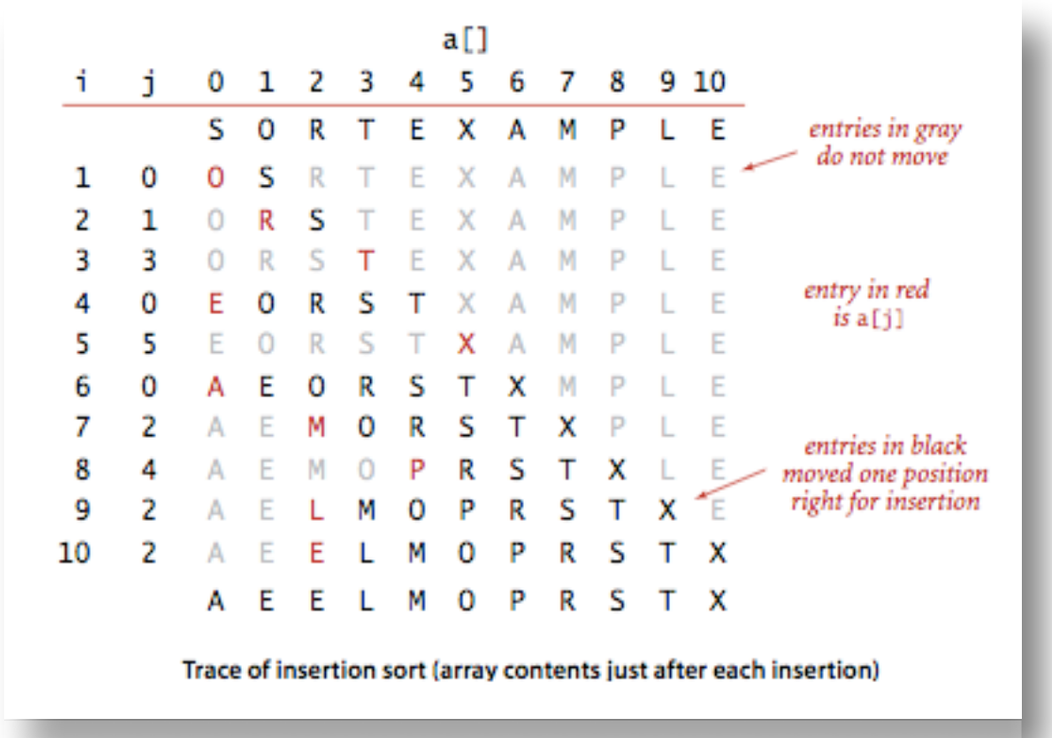


Trace of selection sort (array contents just after each exchange)

Insertion Sort

For randomly ordered arrays of length N with with distinct keys, insertion sort uses ~N2/4 compares and ~N2/4 exchanges on the average. The worst case is ~ N2/2 compares and ~ N2/2 exchanges and the best case is N-1 compares and 0 exchanges.

Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. An inversion is a pair of keys that are out of order in the array. For instance, E X A M P L E has 11 inversions: E-A, X-A, X-M, X-P, X-L, X-E, M-L, M-E, P-L, P-E, and L-E. If the number of inversions in an array is less than a constant multiple of the array size, we say that the array is partially sorted.

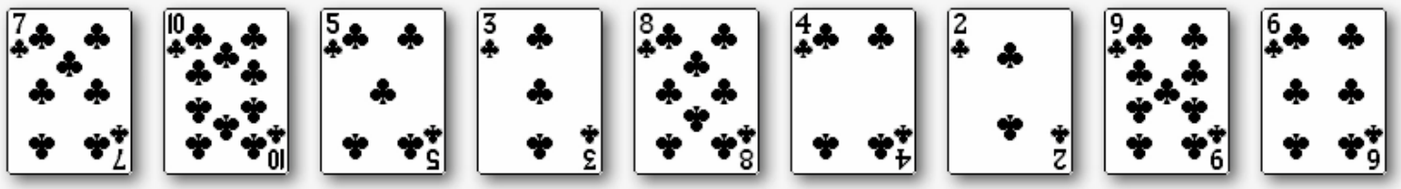


Trace of insertion sort (array contents just after each insertion)

Insertion Sort Demo

Insertion sort demo

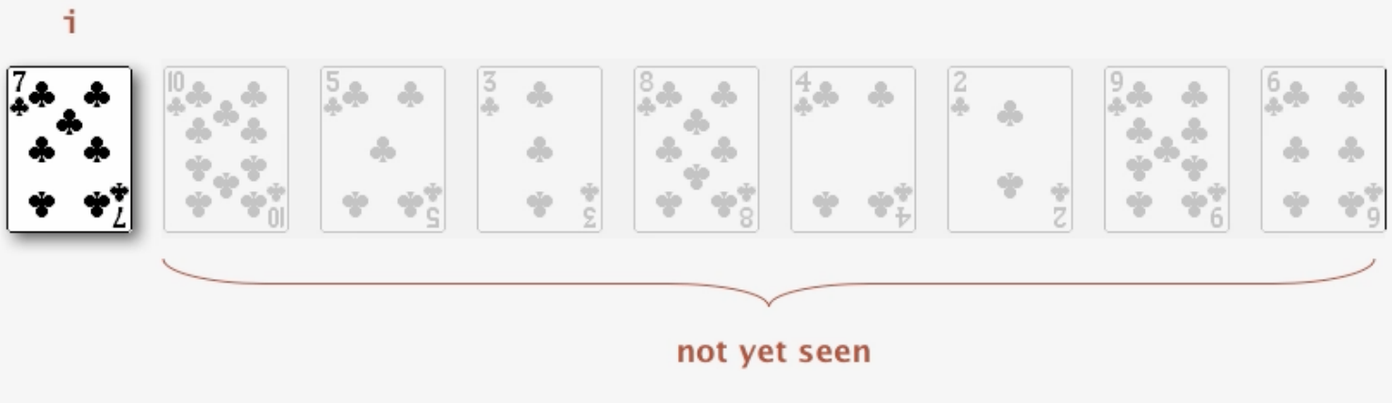
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

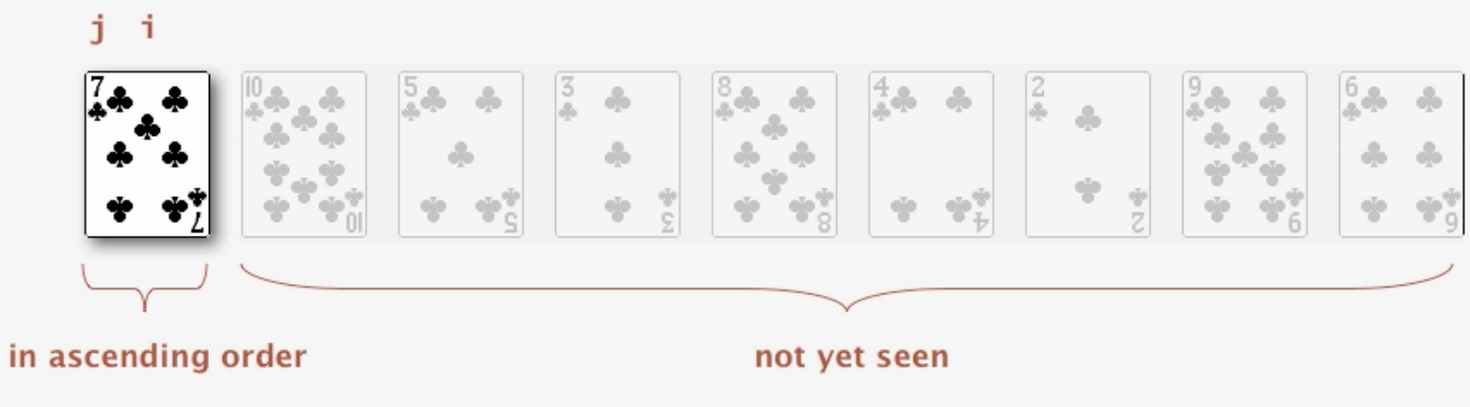
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

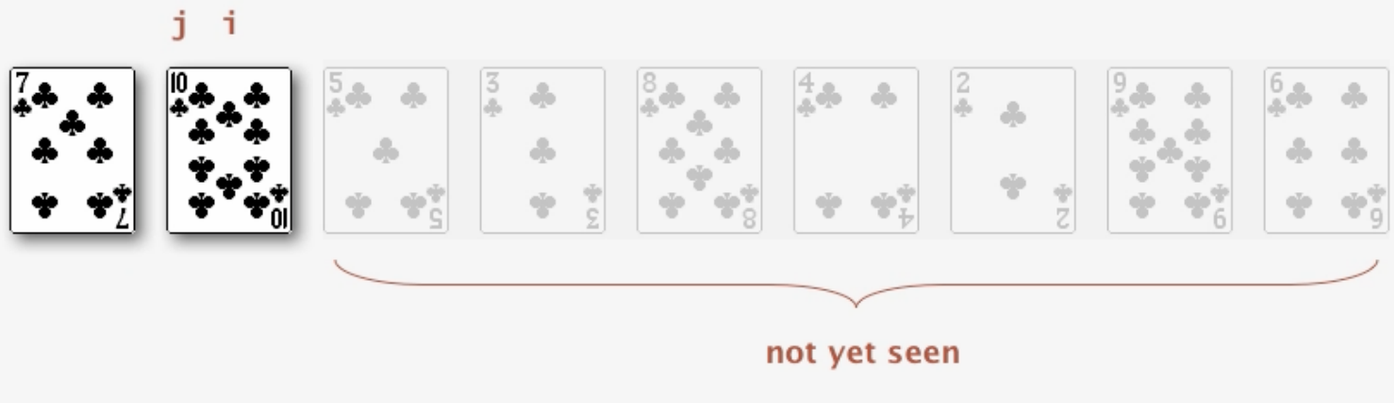
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

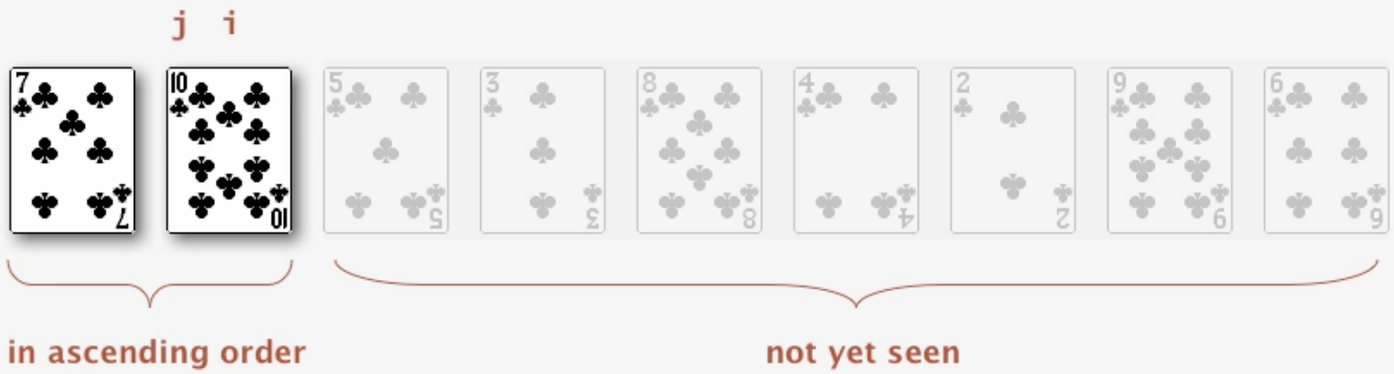
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

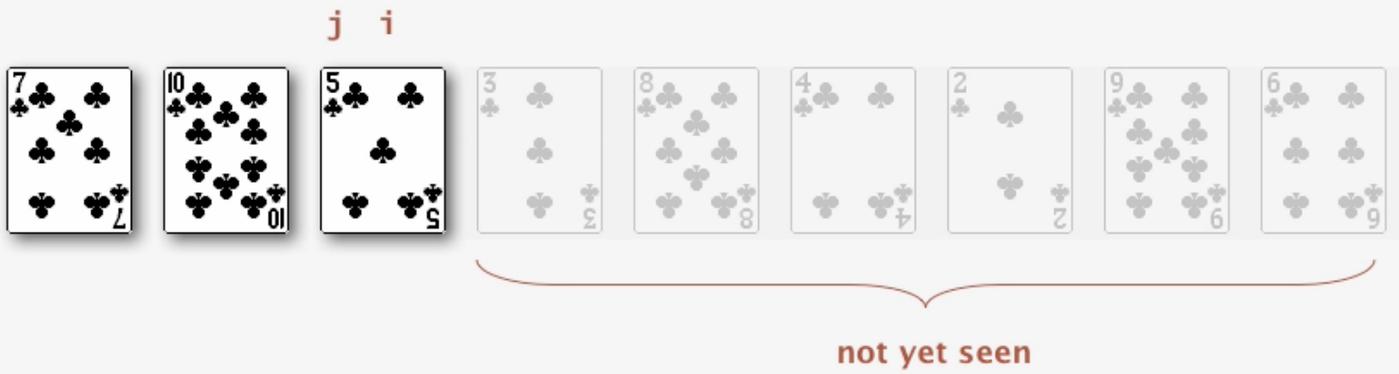
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

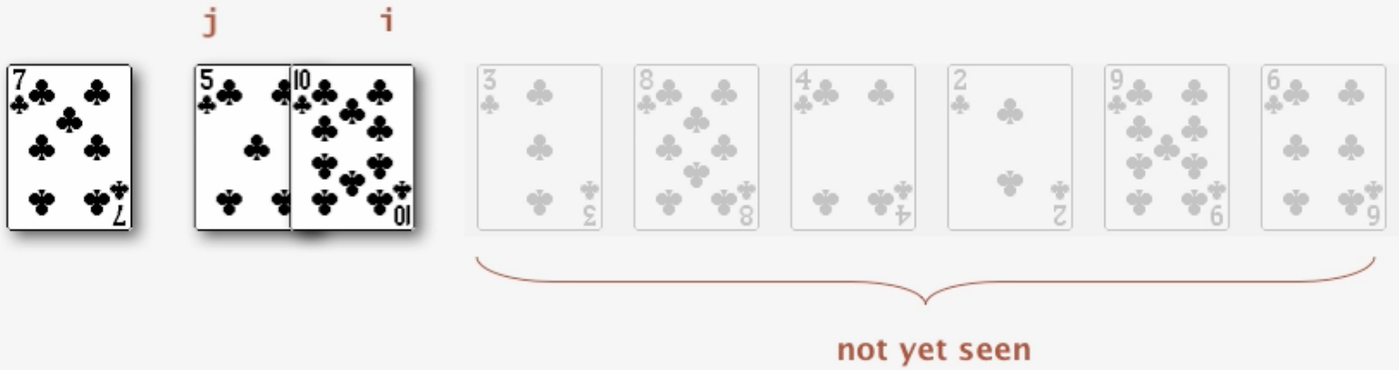
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

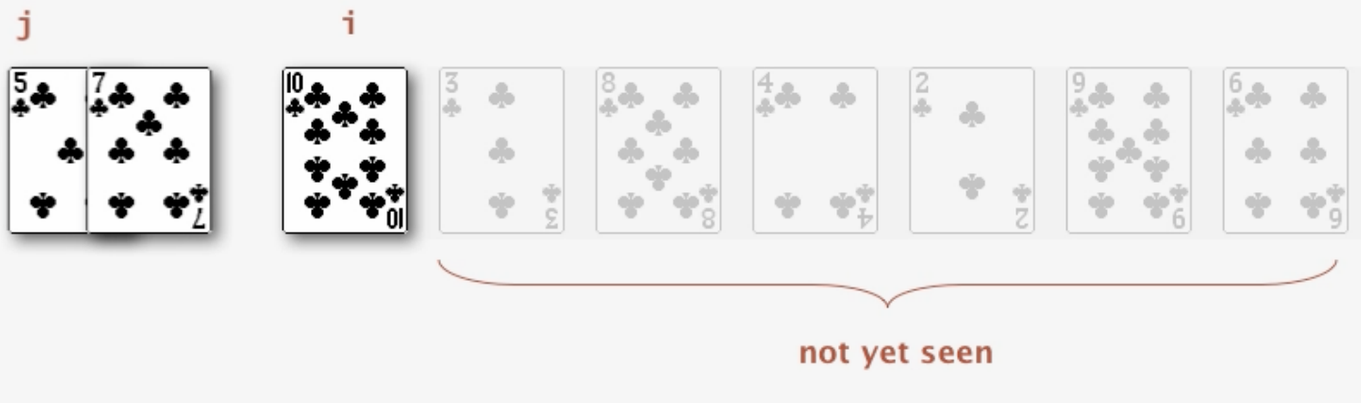
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

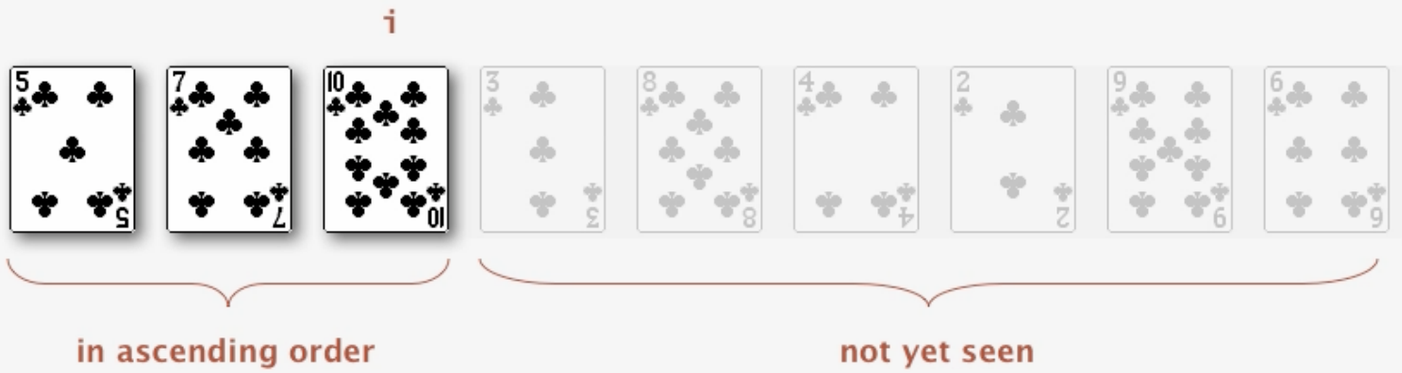
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

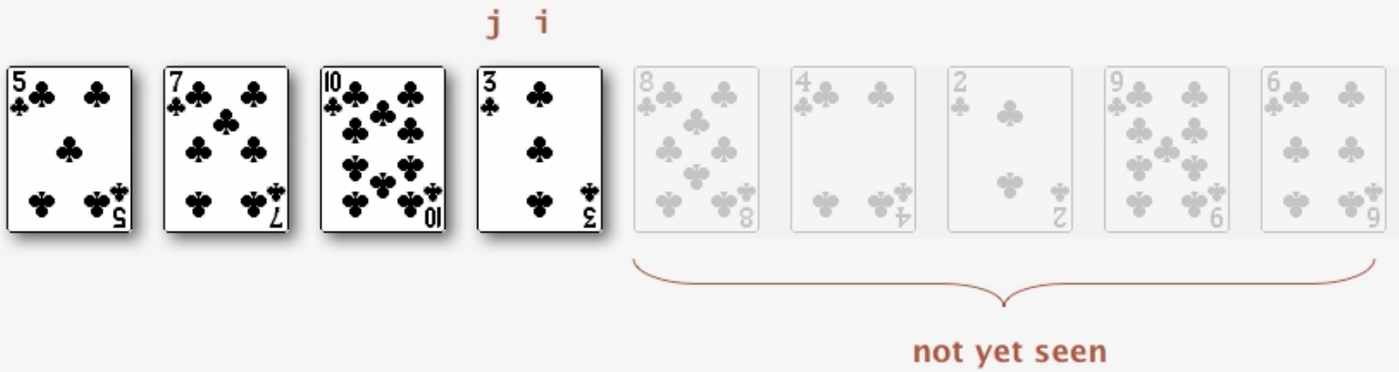
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

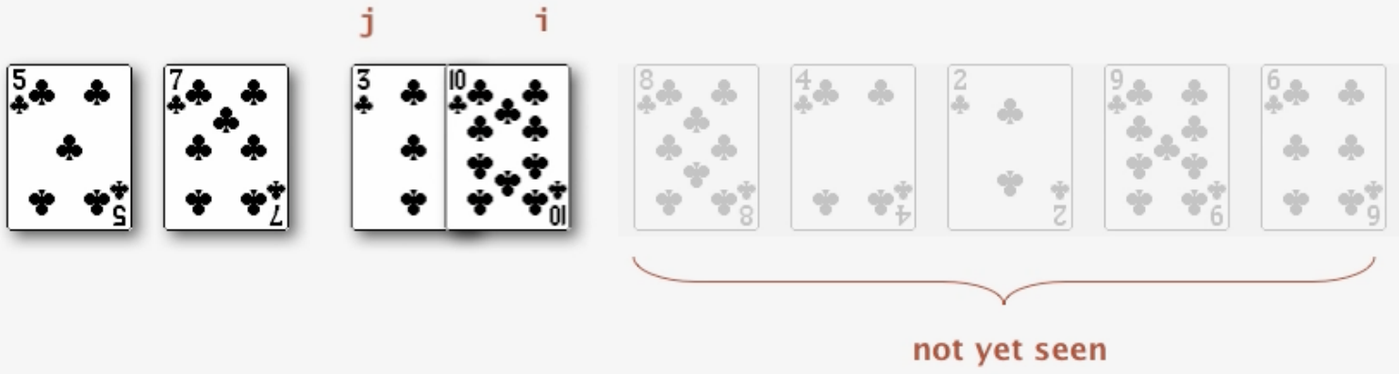
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

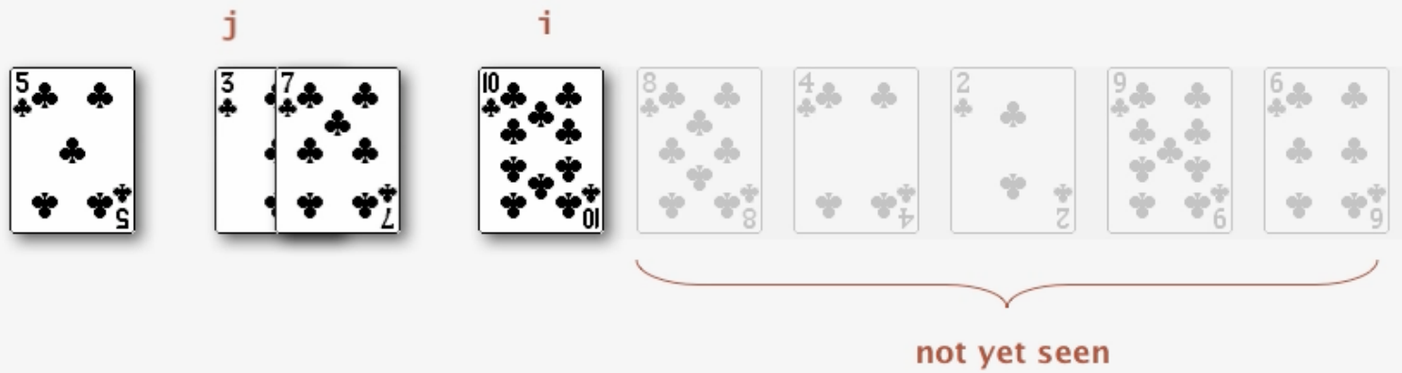
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

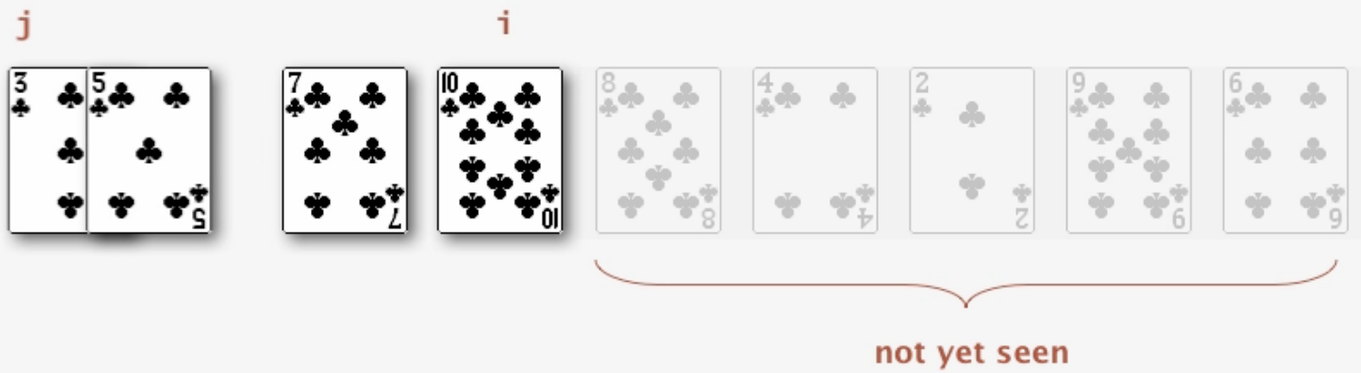
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

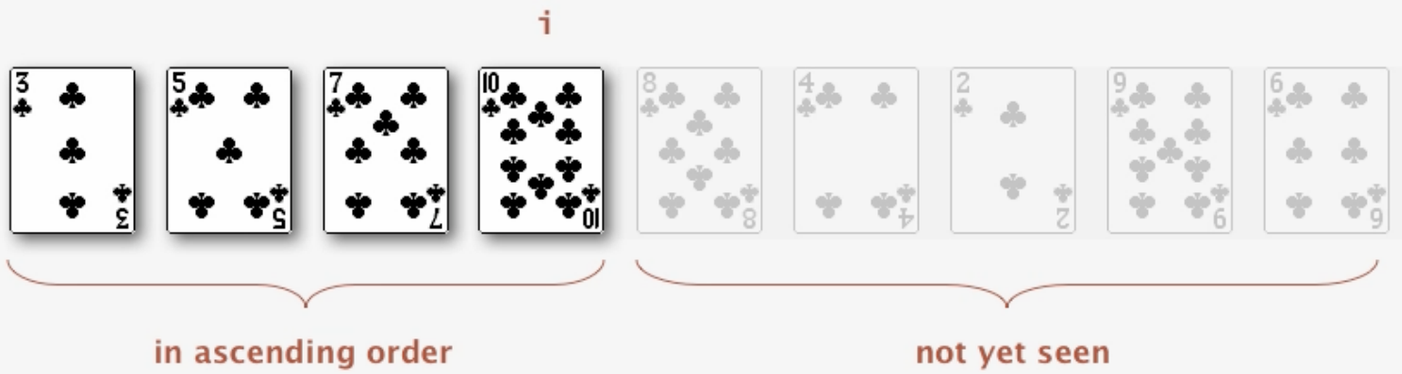
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

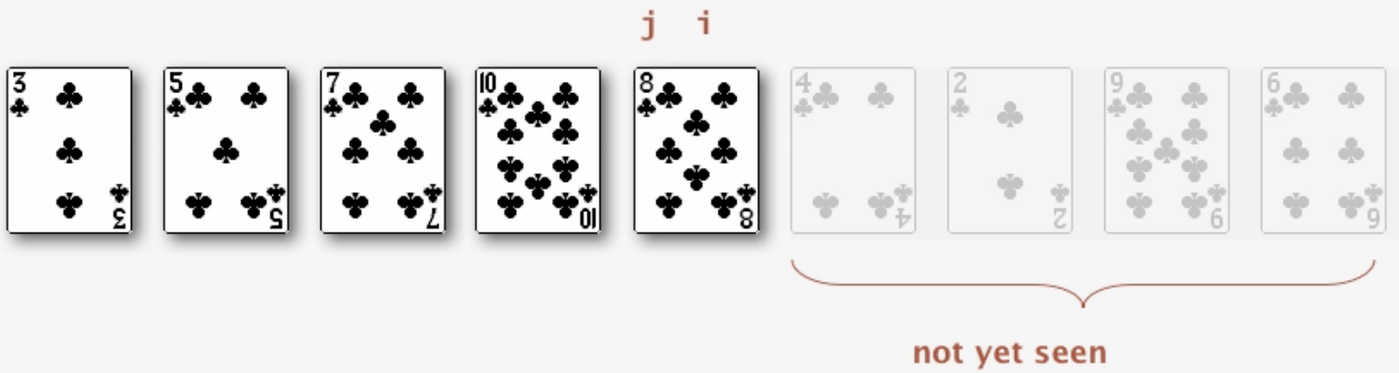
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

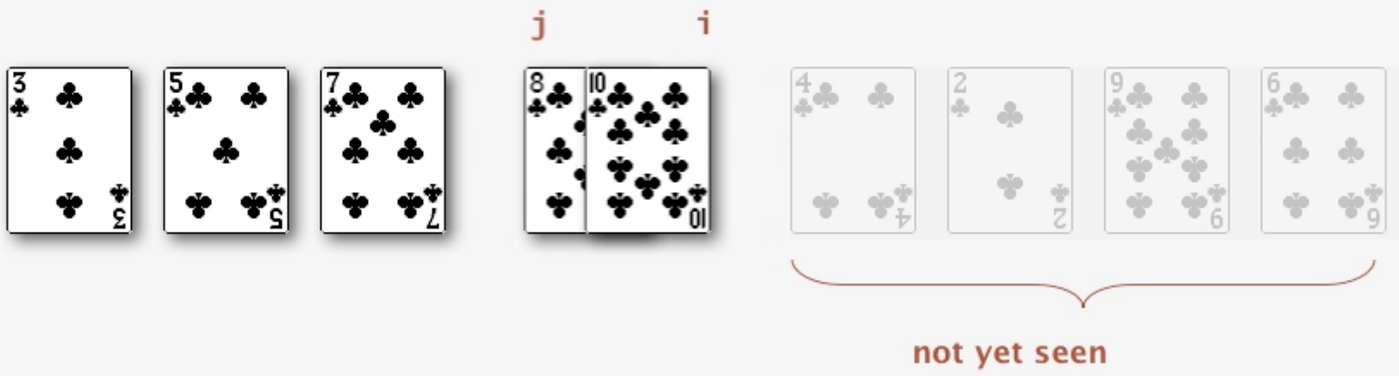
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

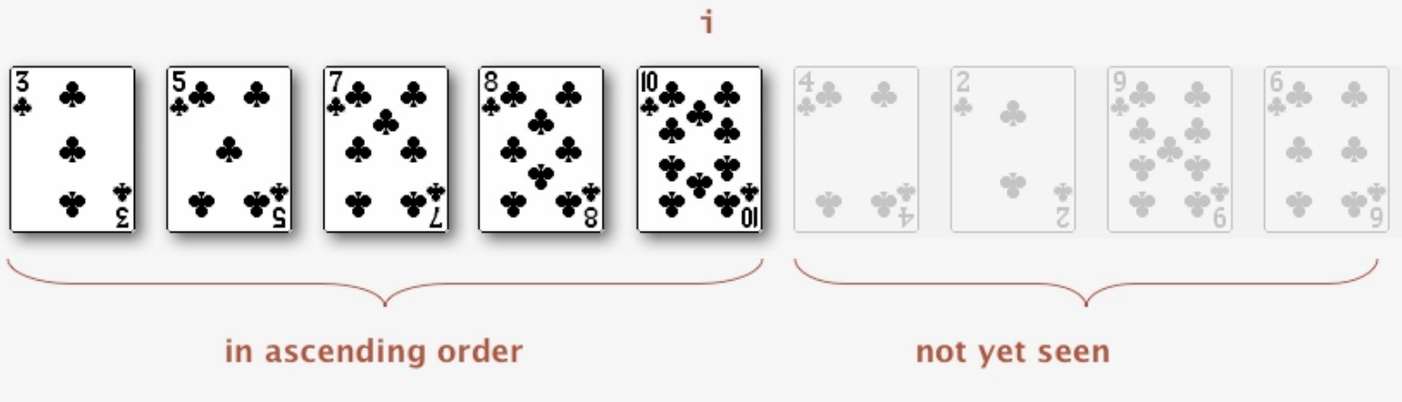
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

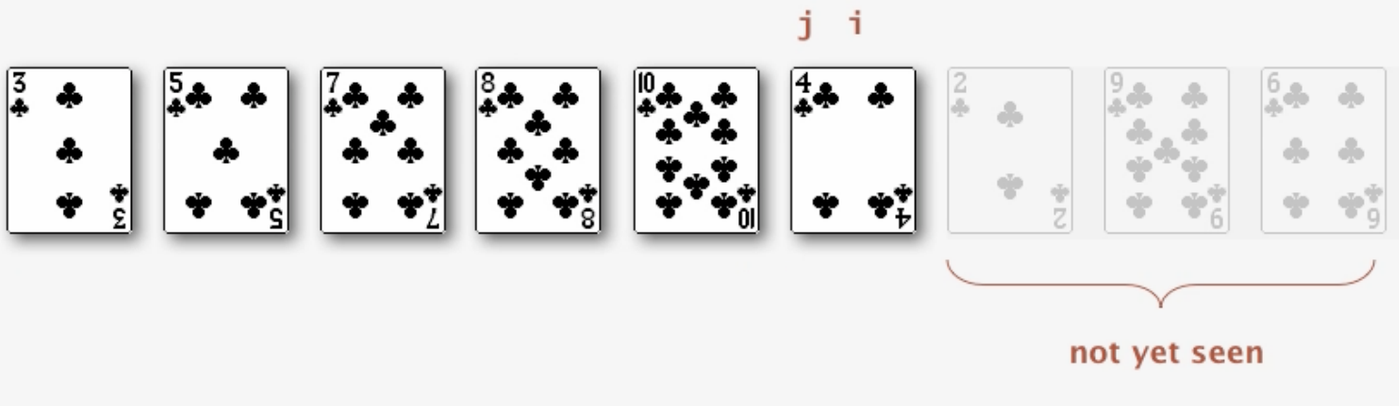
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

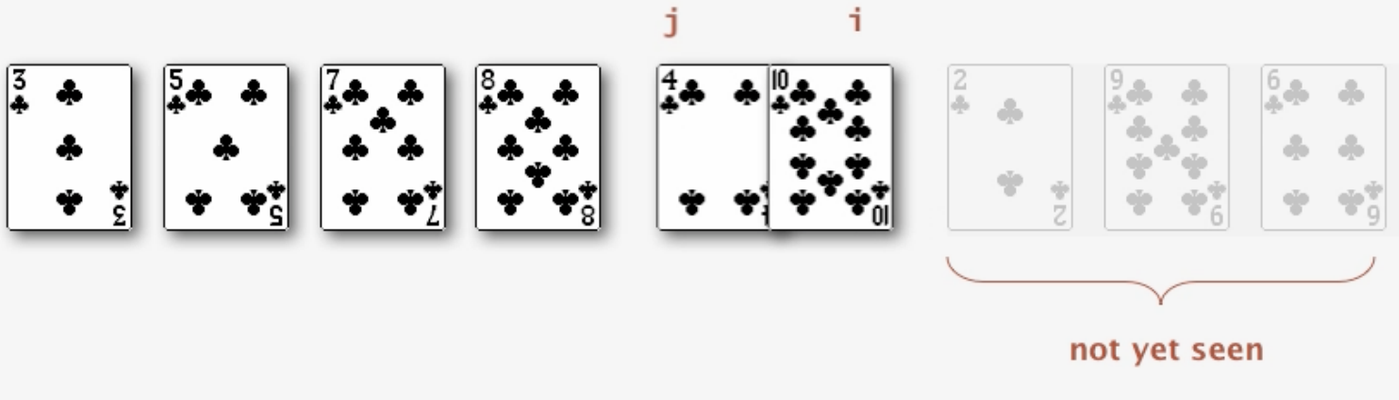
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

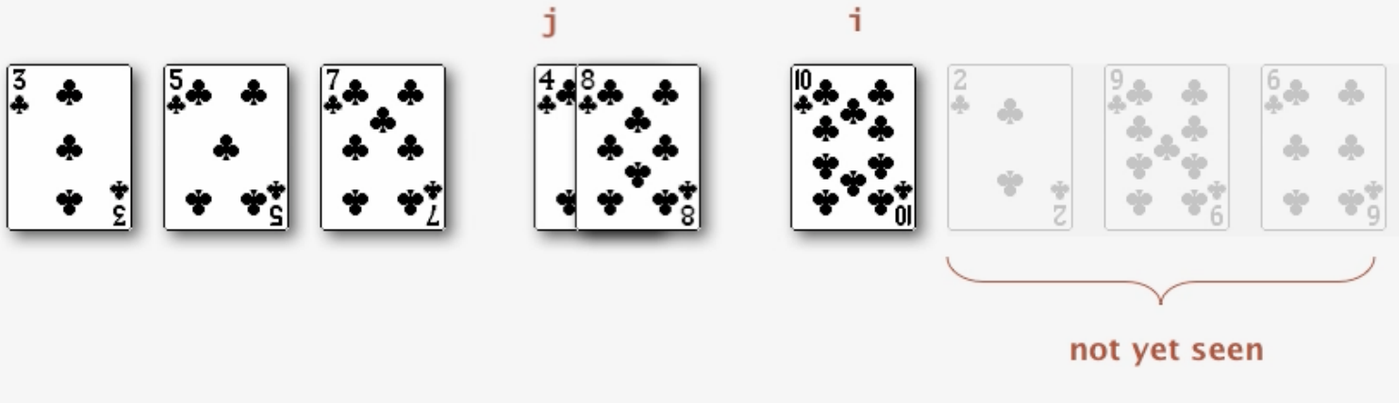
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

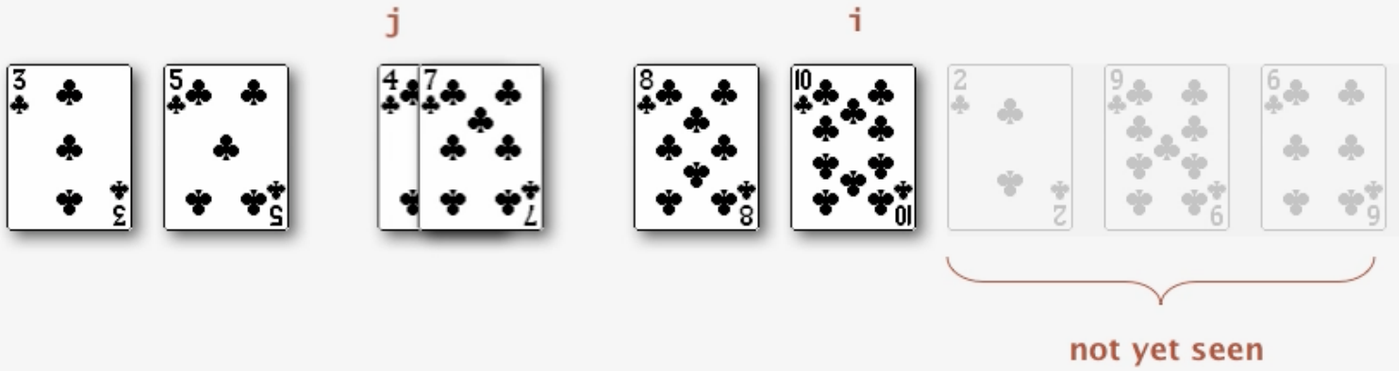
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

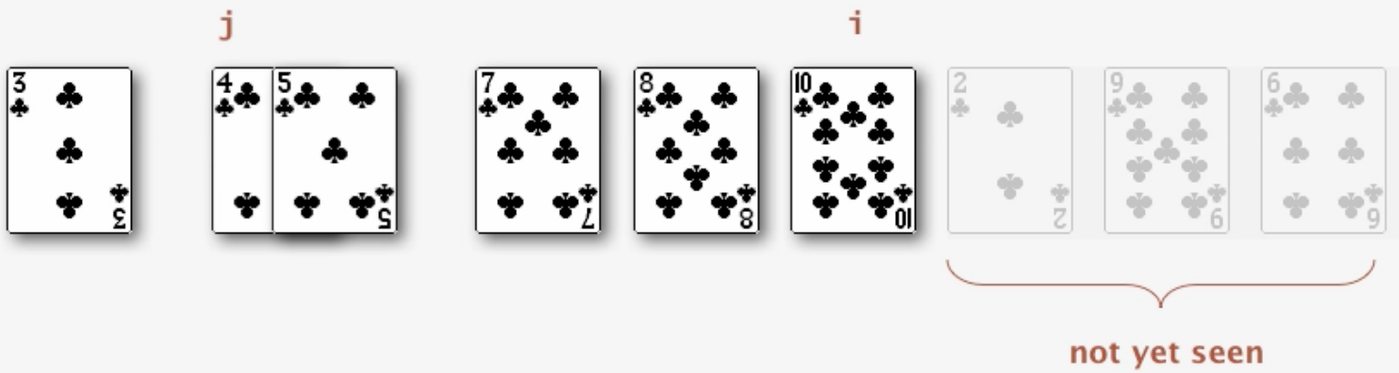
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

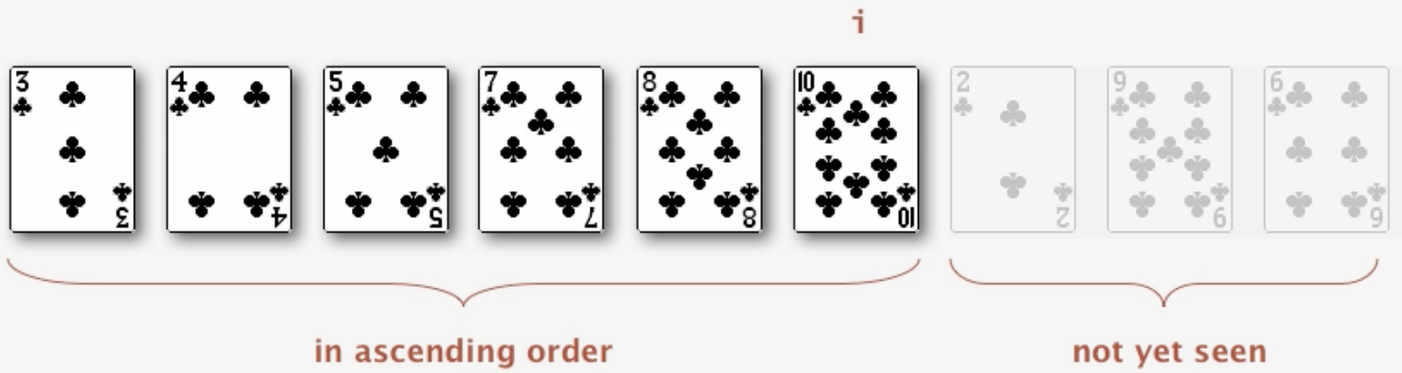
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

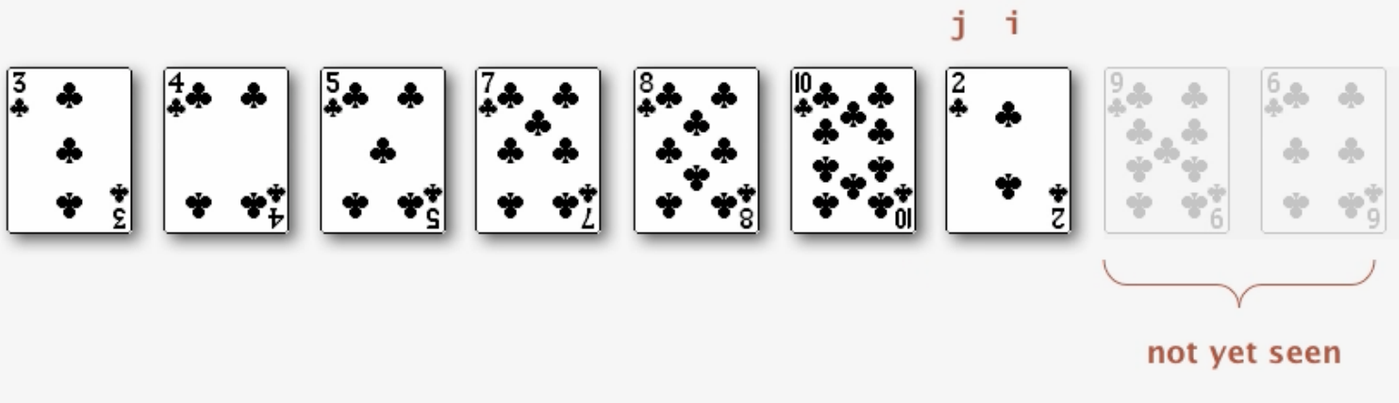
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

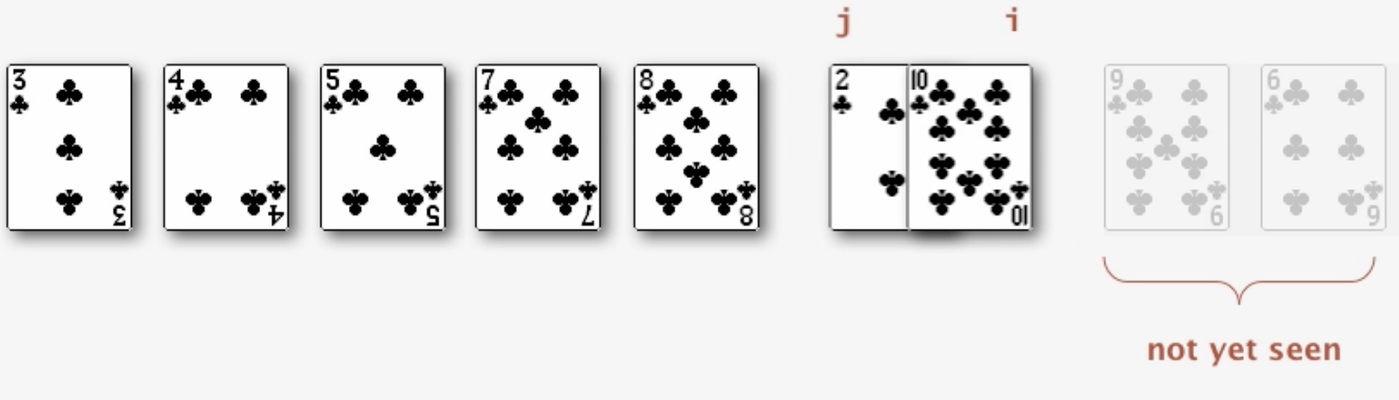
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

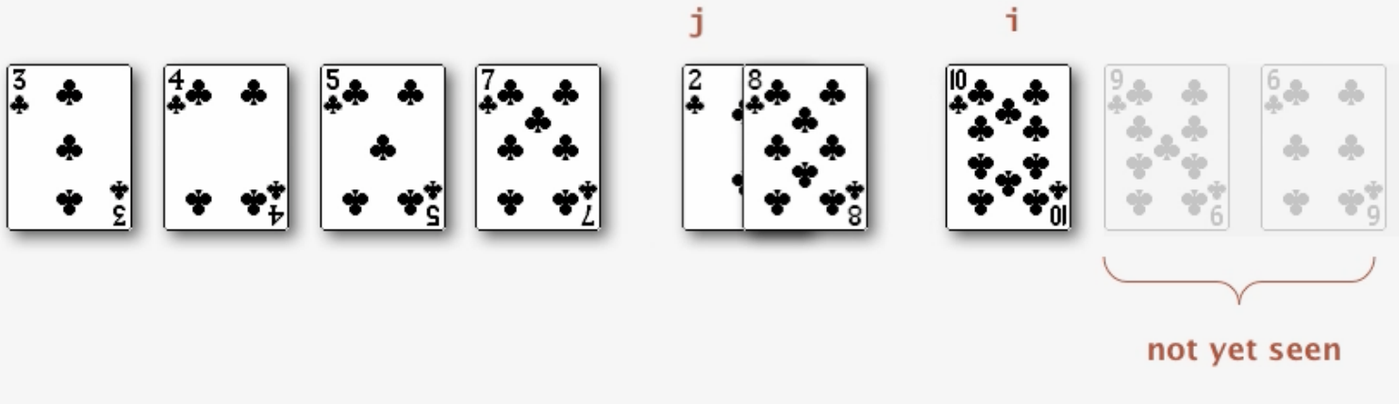
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

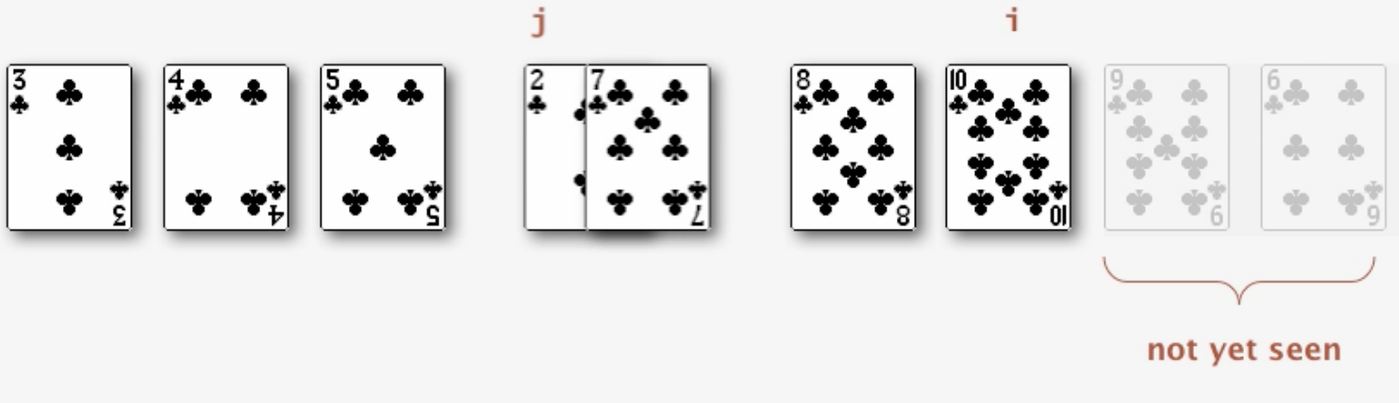
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

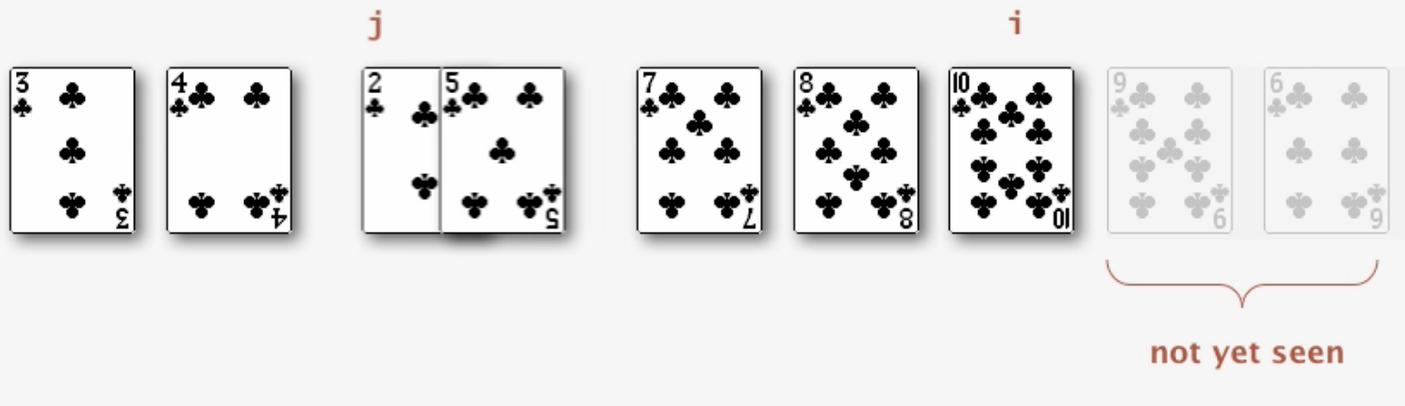
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

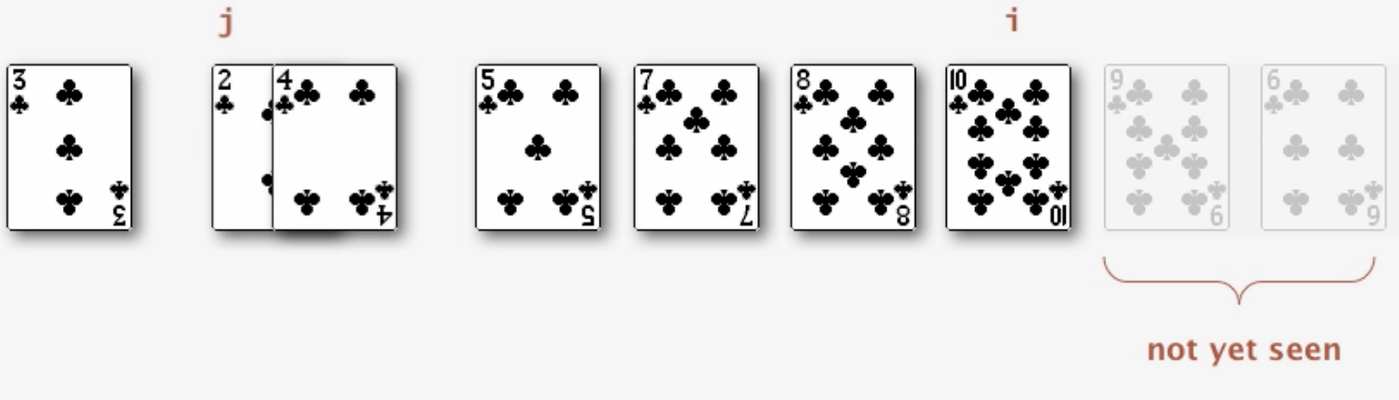
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

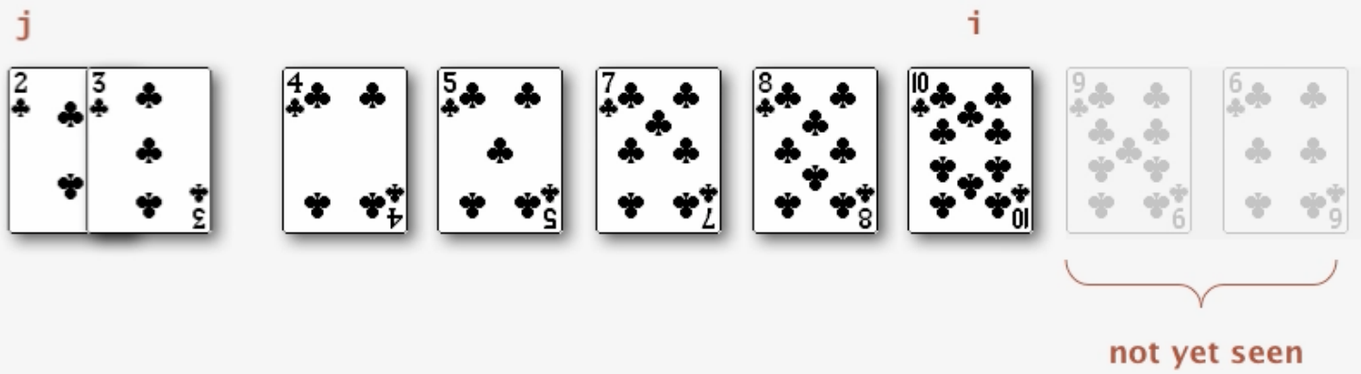
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

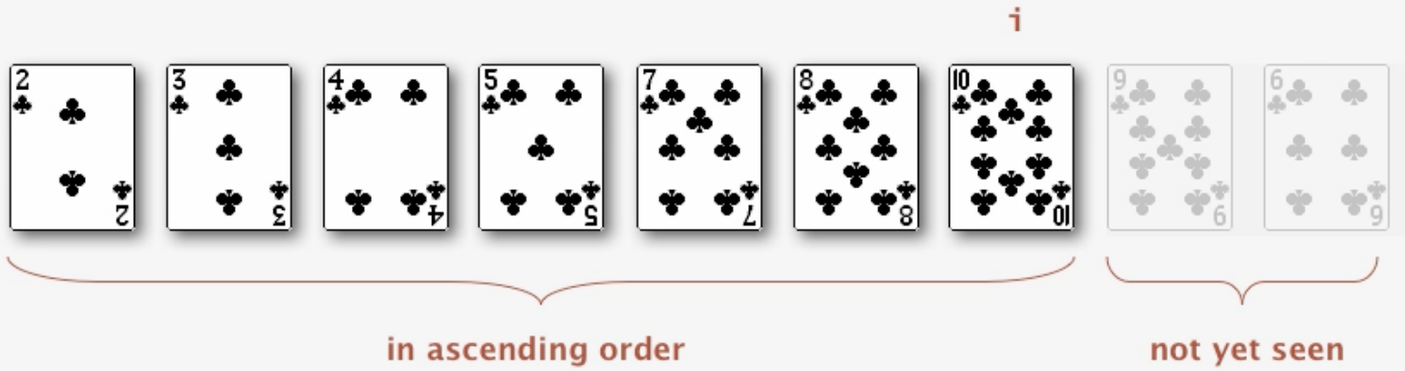
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

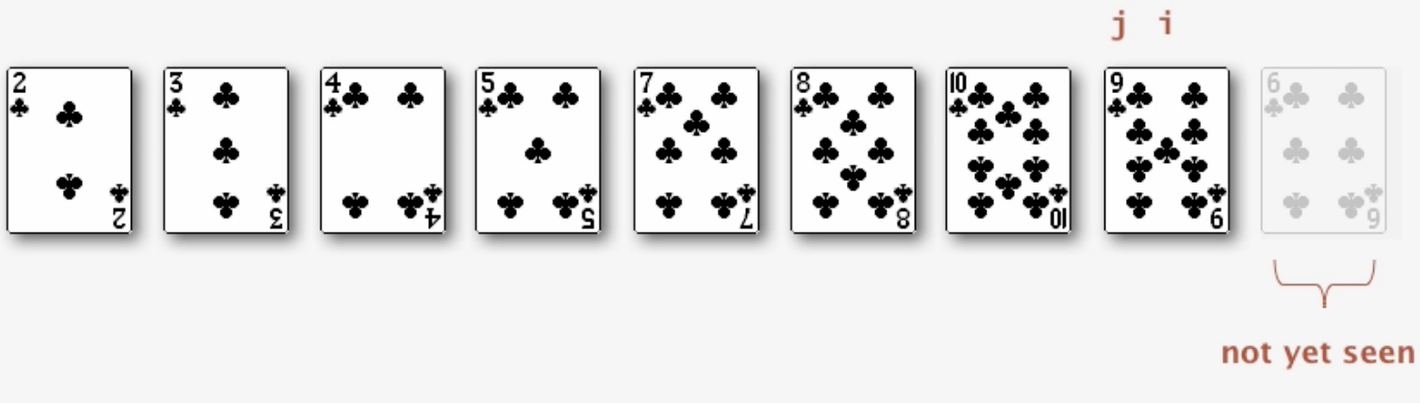
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

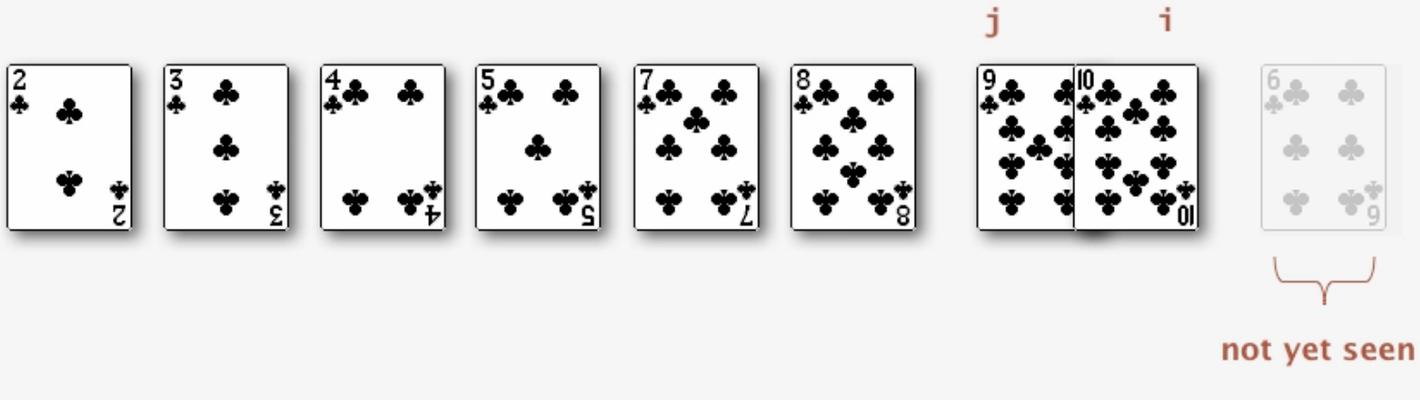
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

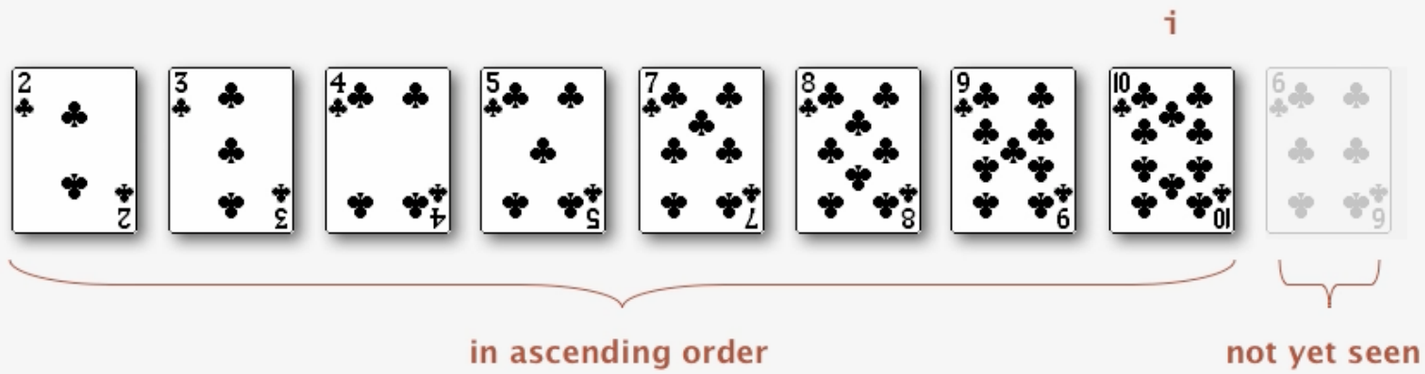
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

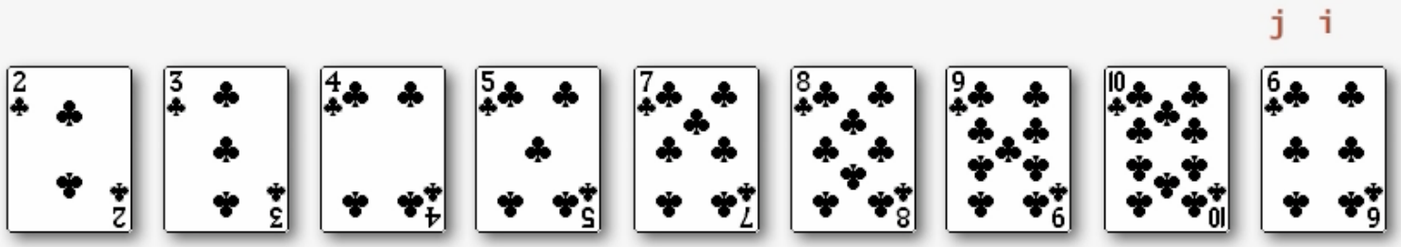
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

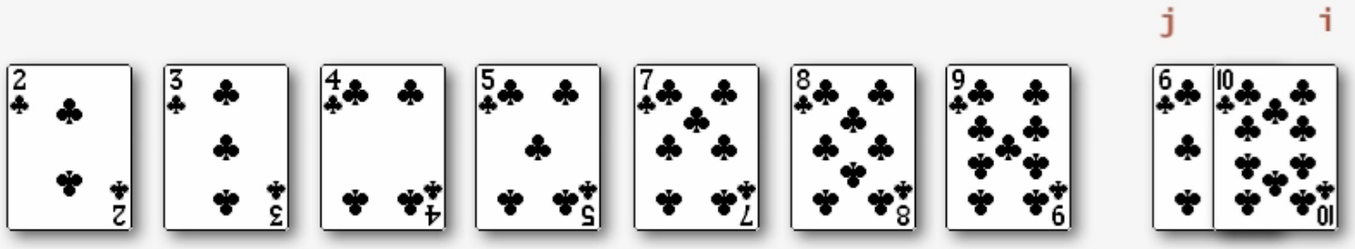
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

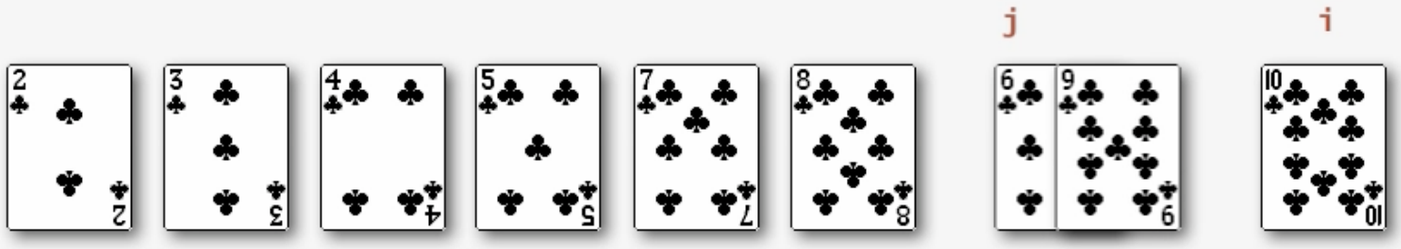
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

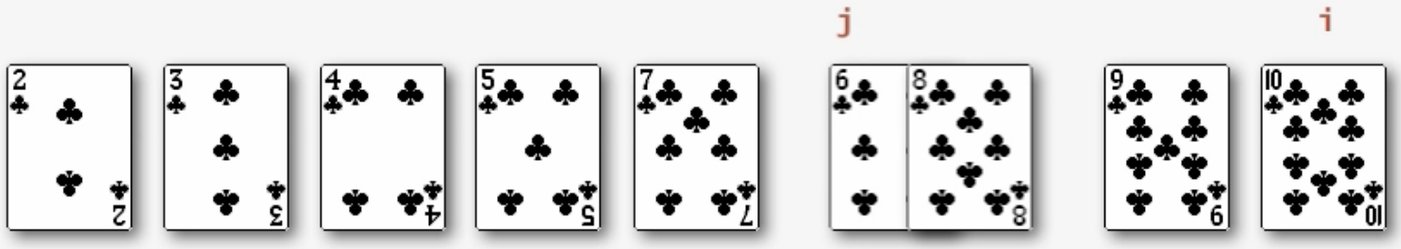
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

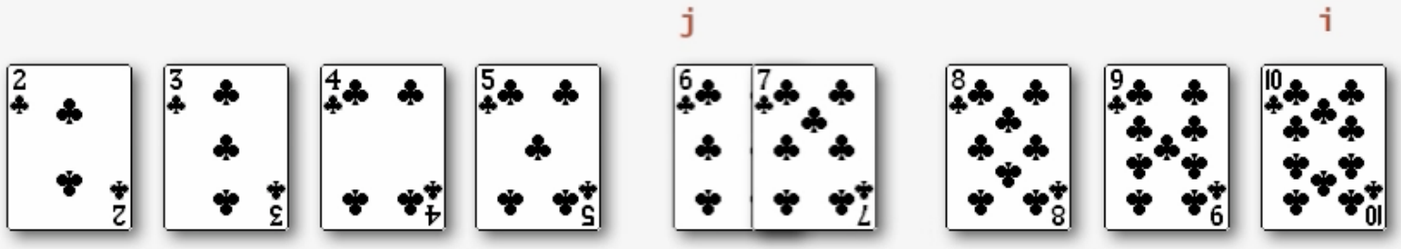
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

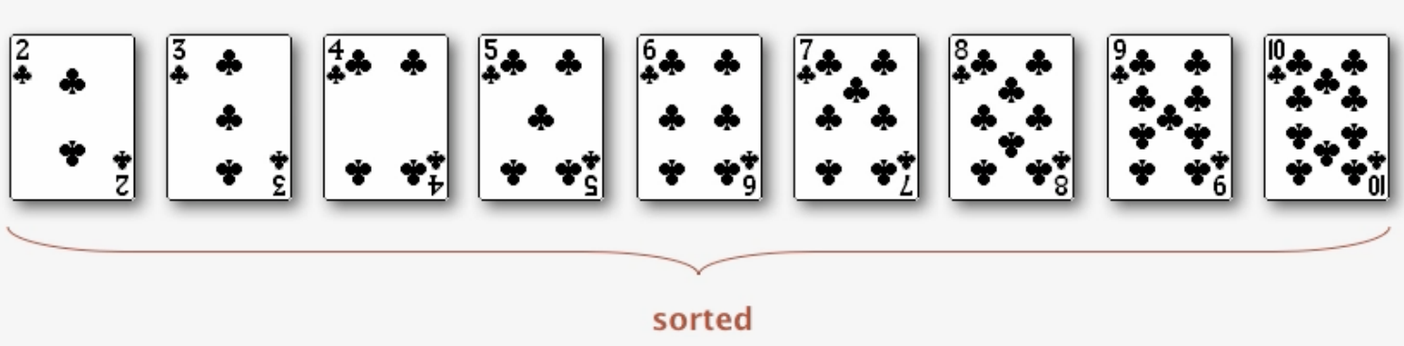
- In iteration `i`, swap `a[i]` with each larger entry to its left.



Insertion Sort Demo

Insertion sort demo

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

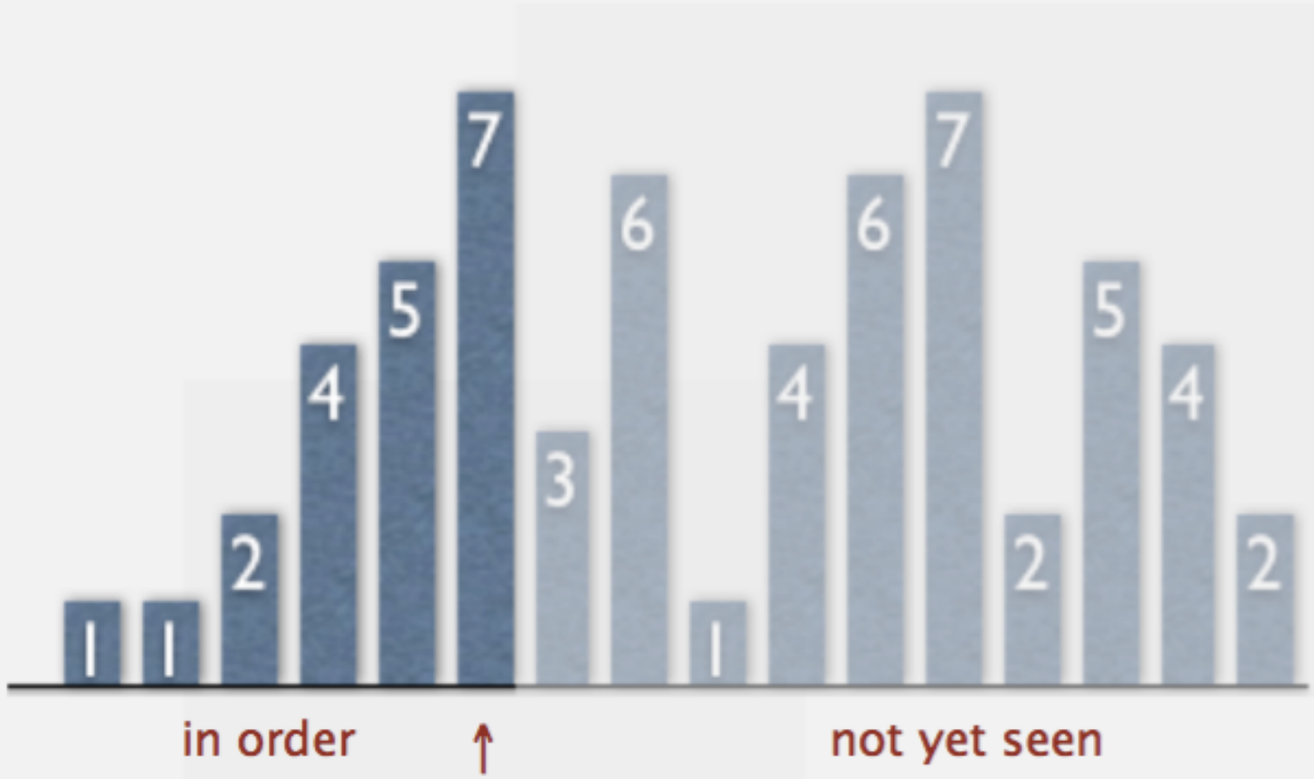


Insertion Sort

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

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

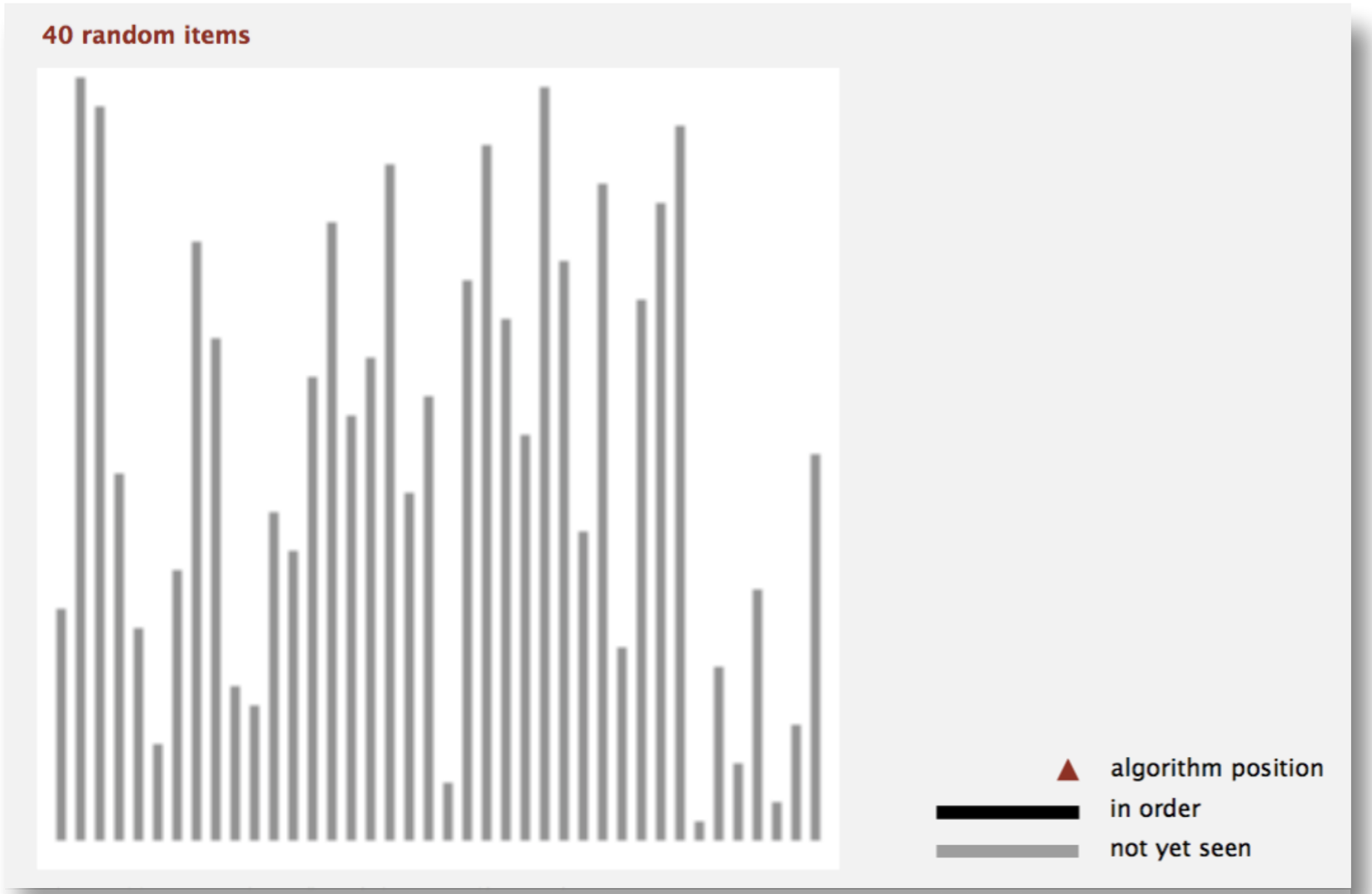
- Moving from right to left, exchange `a[i]` with each larger entry to its left.

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

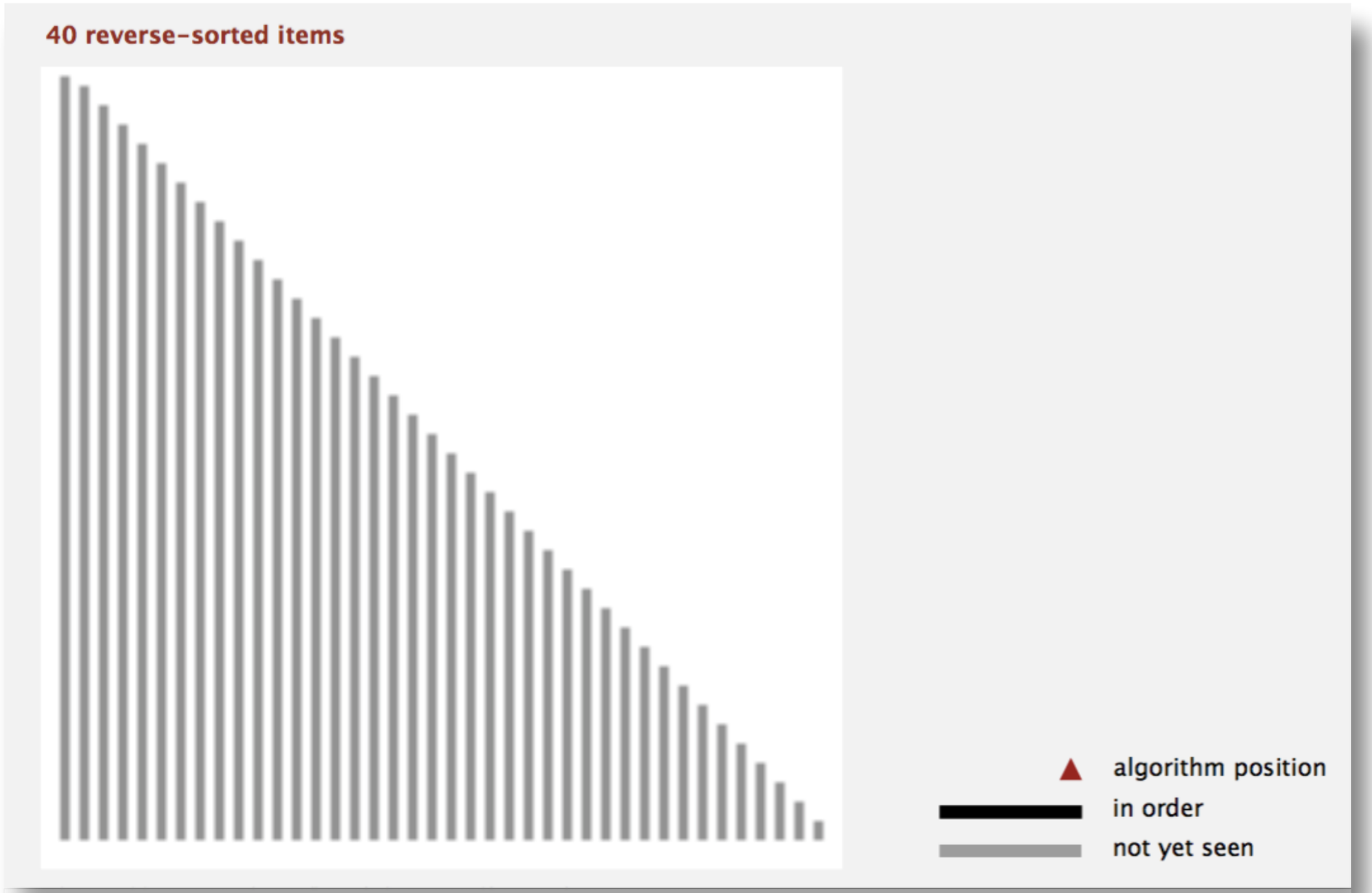
Insertion Sort

```
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 */  }
}
```

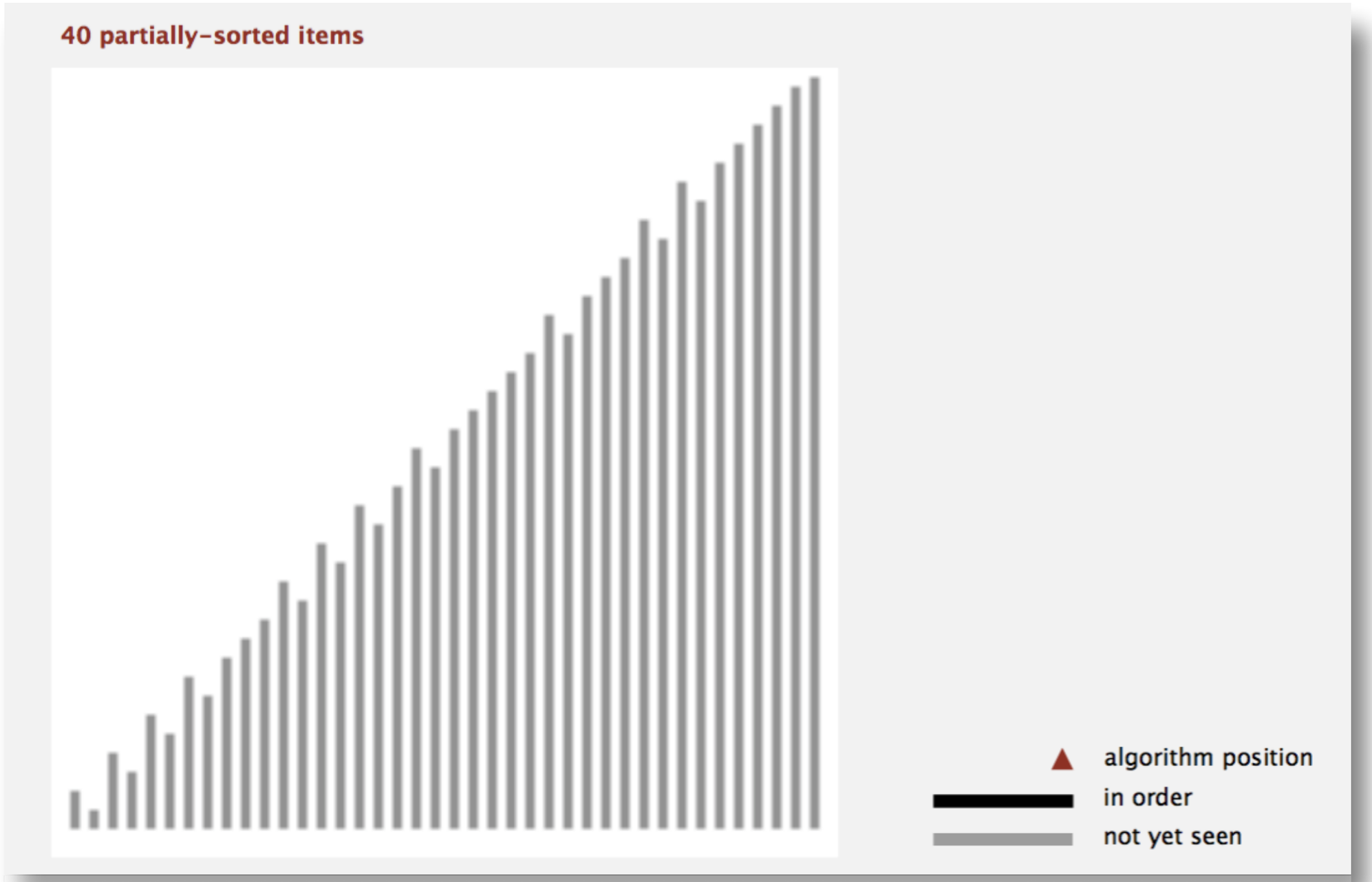
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

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

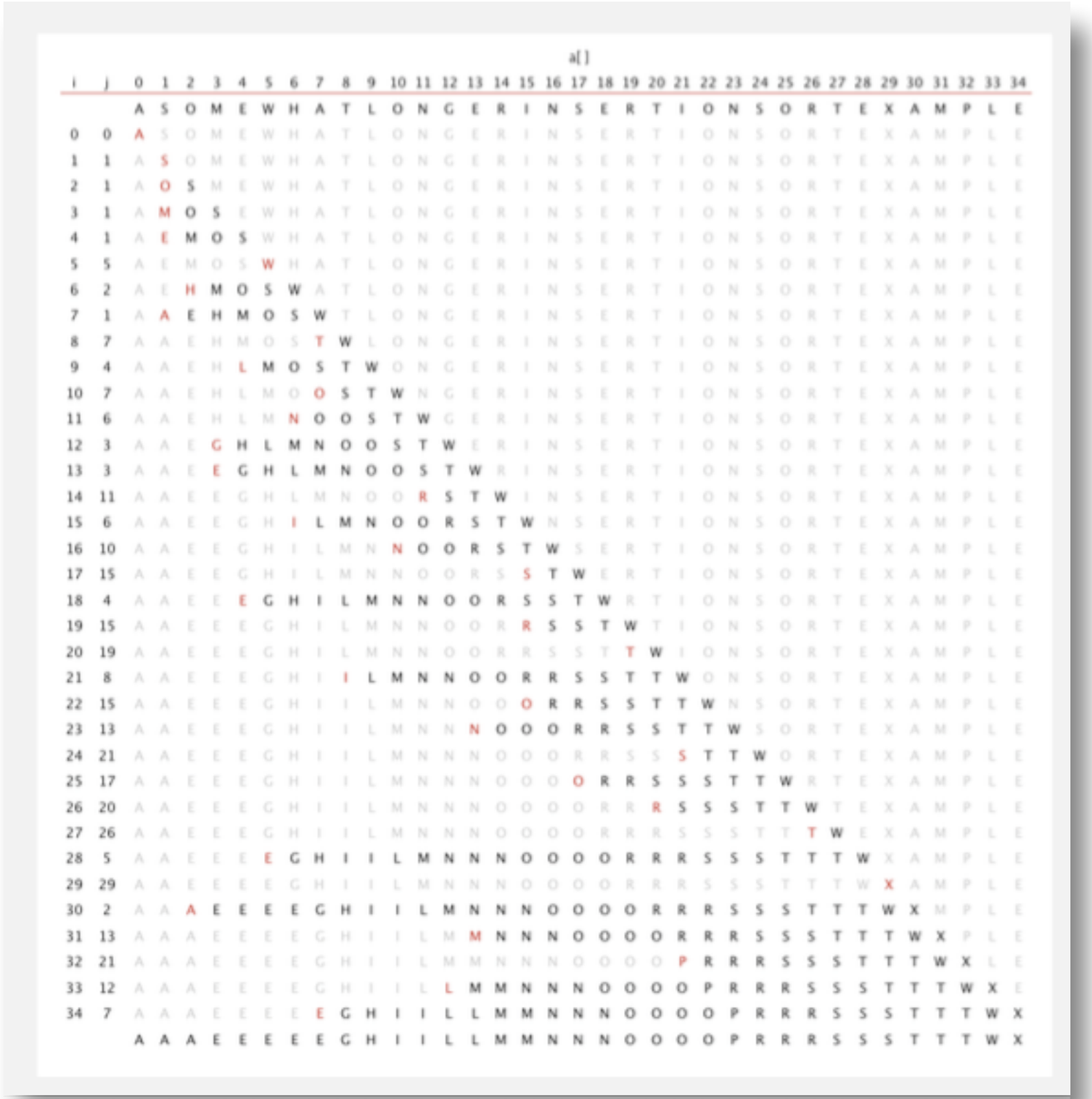
entries in gray do not move

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



		a[ ]																																
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
		A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
0	0	A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
1	1	A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
2	1	A	O	S	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
3	1	A	M	O	S	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
4	1	A	E	M	O	S	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
5	5	A	E	M	O	S	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
6	2	A	E	H	M	O	S	W	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
7	1	A	A	E	H	M	O	S	W	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
8	7	A	A	E	H	M	O	S	T	W	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
9	4	A	A	E	H	L	M	O	S	T	W	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
10	7	A	A	E	H	L	M	O	O	S	T	W	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
11	6	A	A	E	H	L	M	N	O	O	S	T	W	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
12	3	A	A	E	G	H	L	M	N	O	O	S	T	W	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
13	3	A	A	E	E	G	H	L	M	N	O	O	S	T	W	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
14	11	A	A	E	E	G	H	L	M	N	O	O	R	S	T	W	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
15	6	A	A	E	E	G	H	I	L	M	N	O	O	R	S	T	W	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
16	10	A	A	E	E	G	H	I	L	M	N	N	O	O	R	S	T	W	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
17	15	A	A	E	E	G	H	I	L	M	N	N	O	O	R	S	S	T	W	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
18	4	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	S	S	T	W	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
19	15	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	R	S	S	T	W	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
20	19	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	R	S	S	T	T	W	I	O	N	S	O	R	T	E	X	A	M	P	L	E
21	8	A	A	E	E	E	G	H	I	I	L	M	N	N	O	O	R	R	S	S	T	T	W	O	N	S	O	R	T	E	X	A	M	P	L	E
22	15	A	A	E	E	E	G	H	I	I	L	M	N	N	O	O	O	R	R	S	S	T	T	W	N	S	O	R	T	E	X	A	M	P	L	E
23	13	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	R	R	S	S	T	T	W	S	O	R	T	E	X	A	M	P	L	E
24	21	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	R	R	S	S	S	T	T	W	O	R	T	E	X	A	M	P	L	E
25	17	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	S	S	S	T	T	W	R	T	E	X	A	M	P	L	E
26	20	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	W	T	E	X	A	M	P	L	E
27	26	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	E	X	A	M	P	L	E
28	5	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	A	M	P	L	E
29	29	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	A	M	P	L	E
30	2	A	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	M	P	L	E
31	13	A	A	A	E	E	E	E	G	H	I	I	L	M	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	P	L	E
32	21	A	A	A	E	E	E	E	G	H	I	I	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X	L	E
33	12	A	A	A	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X	E
34	7	A	A	A	E	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X
		A	A	A	E	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X

Insertion Sort

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 F E A

Insertion Sort

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 $\leq cN$.

- 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

Half exchanges. Shift items over (instead of exchanging).

- Eliminates unnecessary data movement.
- No longer uses only `less()` and `exch()` to access data.

A C H H I M N N P Q X Y K B I N A R Y

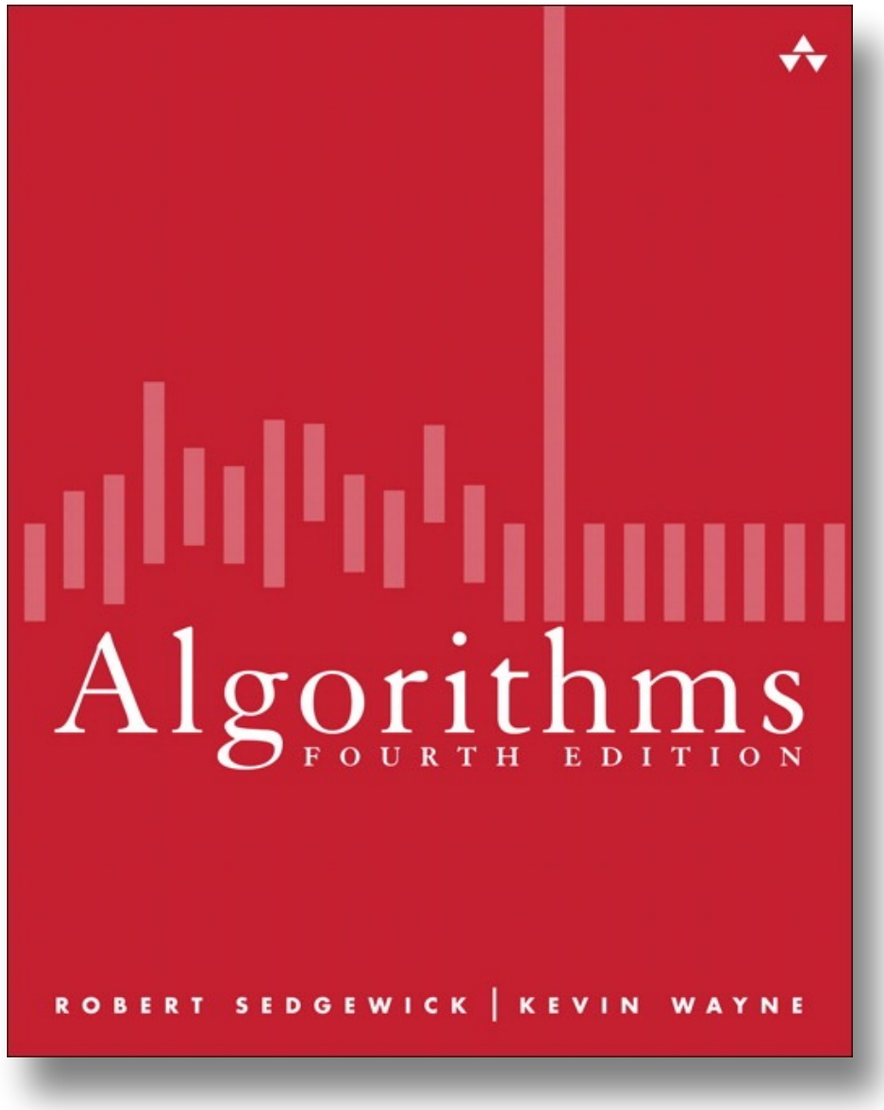
Binary insertion sort. Use binary search to find insertion point.

- Number of compares $\sim N \lg N$.
- But still a quadratic number of array accesses.

A C H H I M N N P Q X Y K B I N A R Y

binary search for first key > K

Lecture Overview

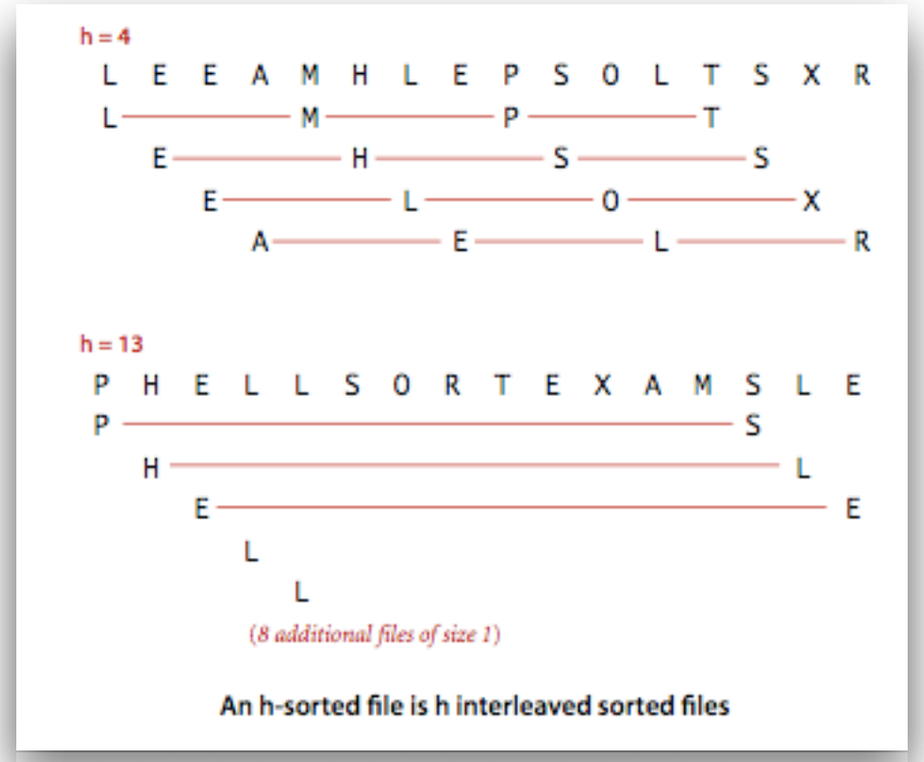


2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

Shellsort

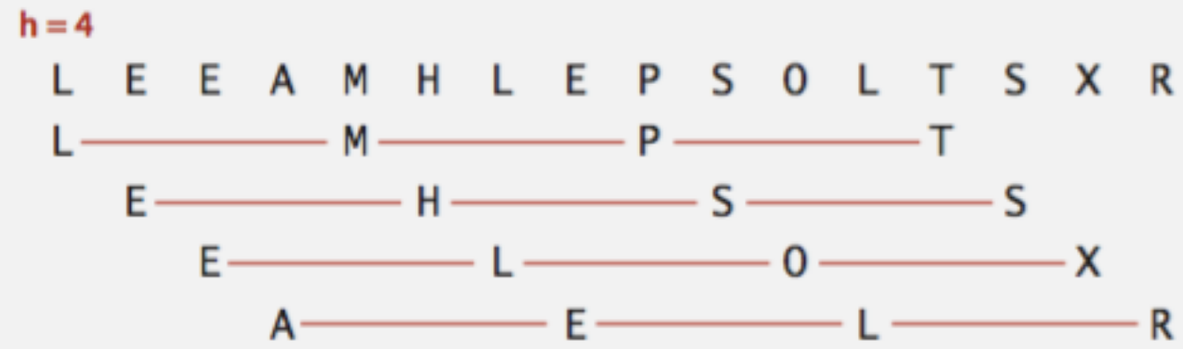
Shellsort is a simple extension of insertion sort that gains speed by allowing exchanges of entries that are far apart, to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort. The idea is to rearrange the array to give it the property that taking every hth entry (starting anywhere) yields a sorted sequence. Such an array is said to be h-sorted.



Shellsort

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

an h-sorted array is h interleaved sorted subsequences

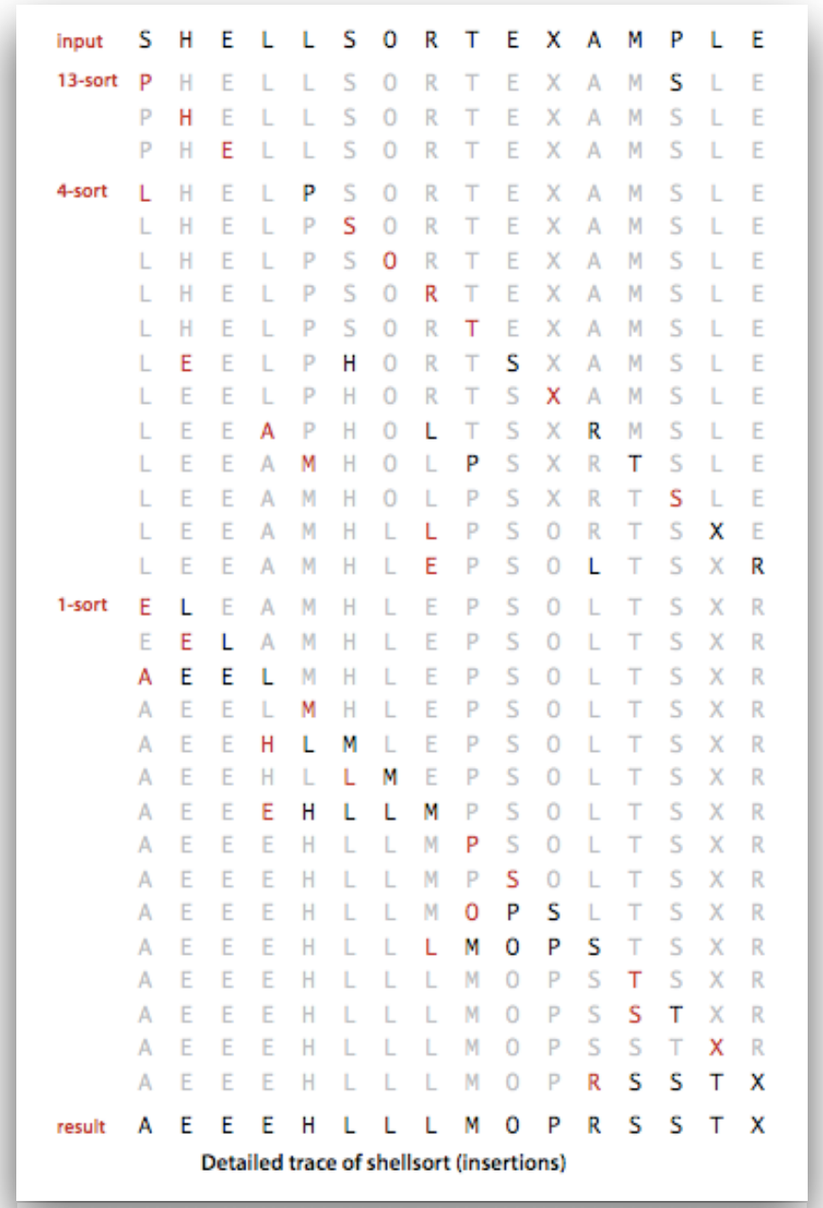


Shellsort. [Shell 1959] h-sort array for decreasing sequence of values of h.

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

Shellsort

By h-sorting for some large values of h, we can move entries in the array long distances and thus make it easier to h-sort for smaller values of h. Using such a procedure for any increment sequence of values of h that ends in 1 will produce a sorted array: that is shellsort. Shell.java is an implementation of this method.



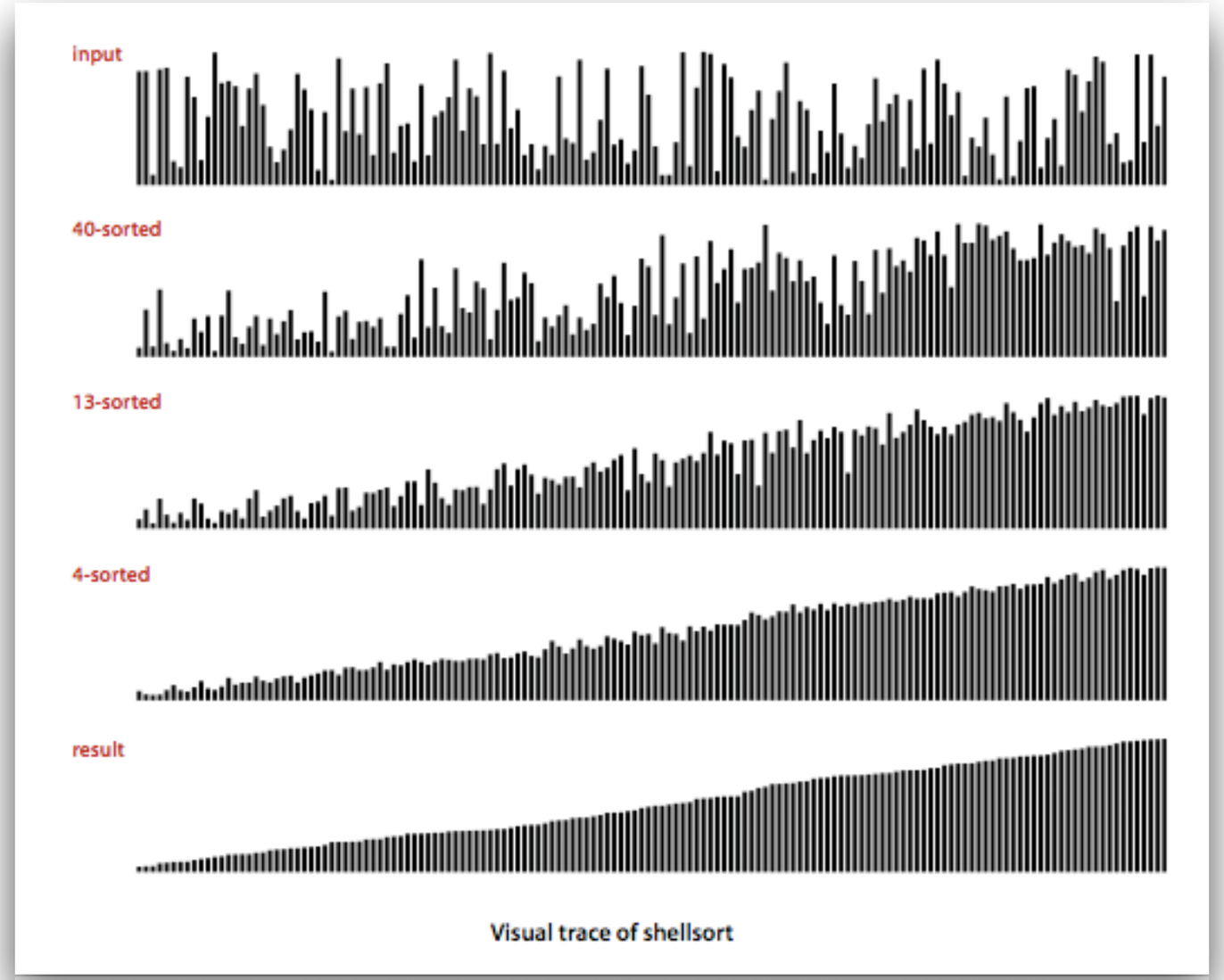
input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T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

Detailed trace of shellsort (insertions)

Shellsort

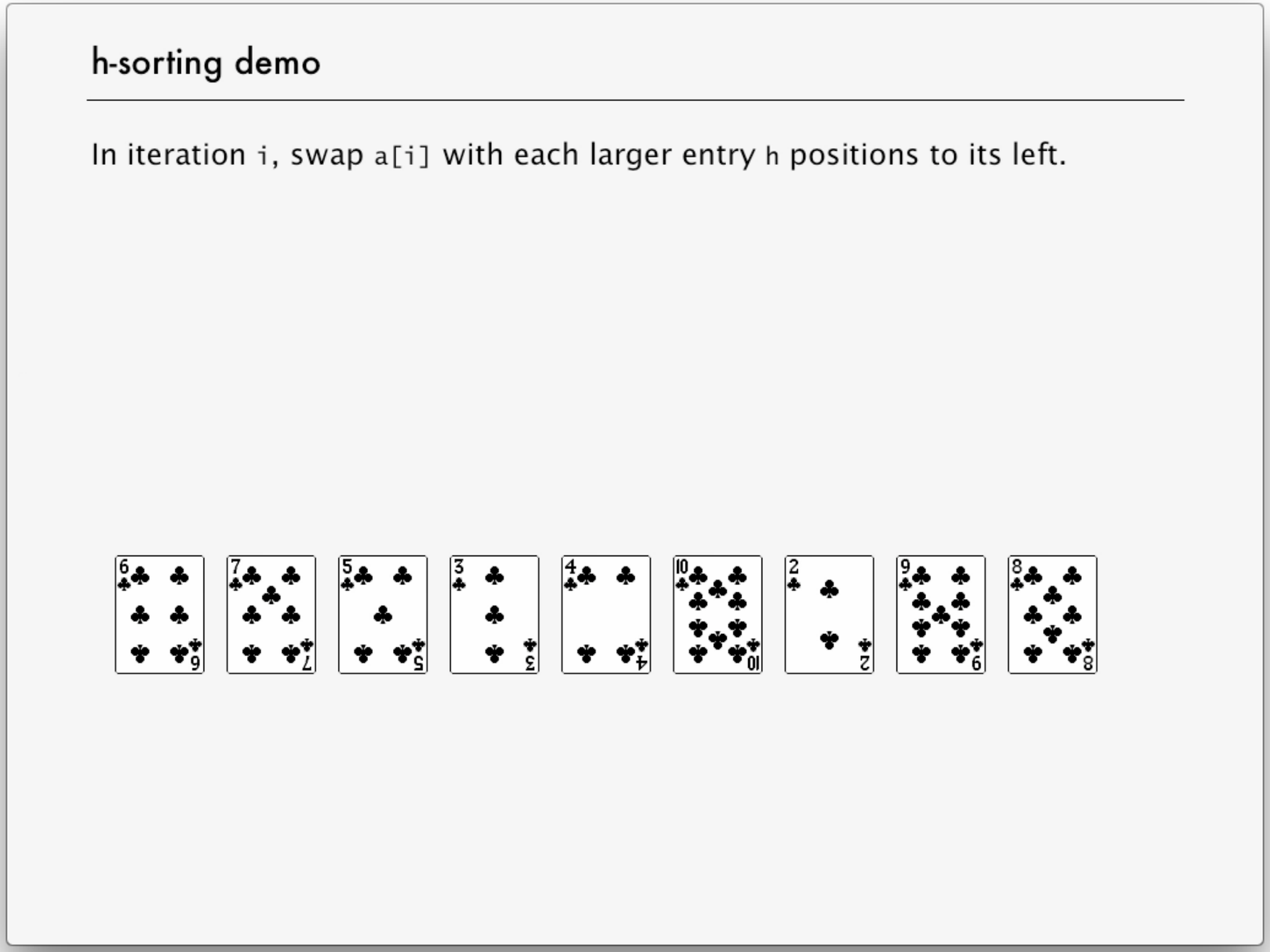
The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, ... is bounded by a small multiple of N times the number of increments used.

The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, ... is O(N3/2).

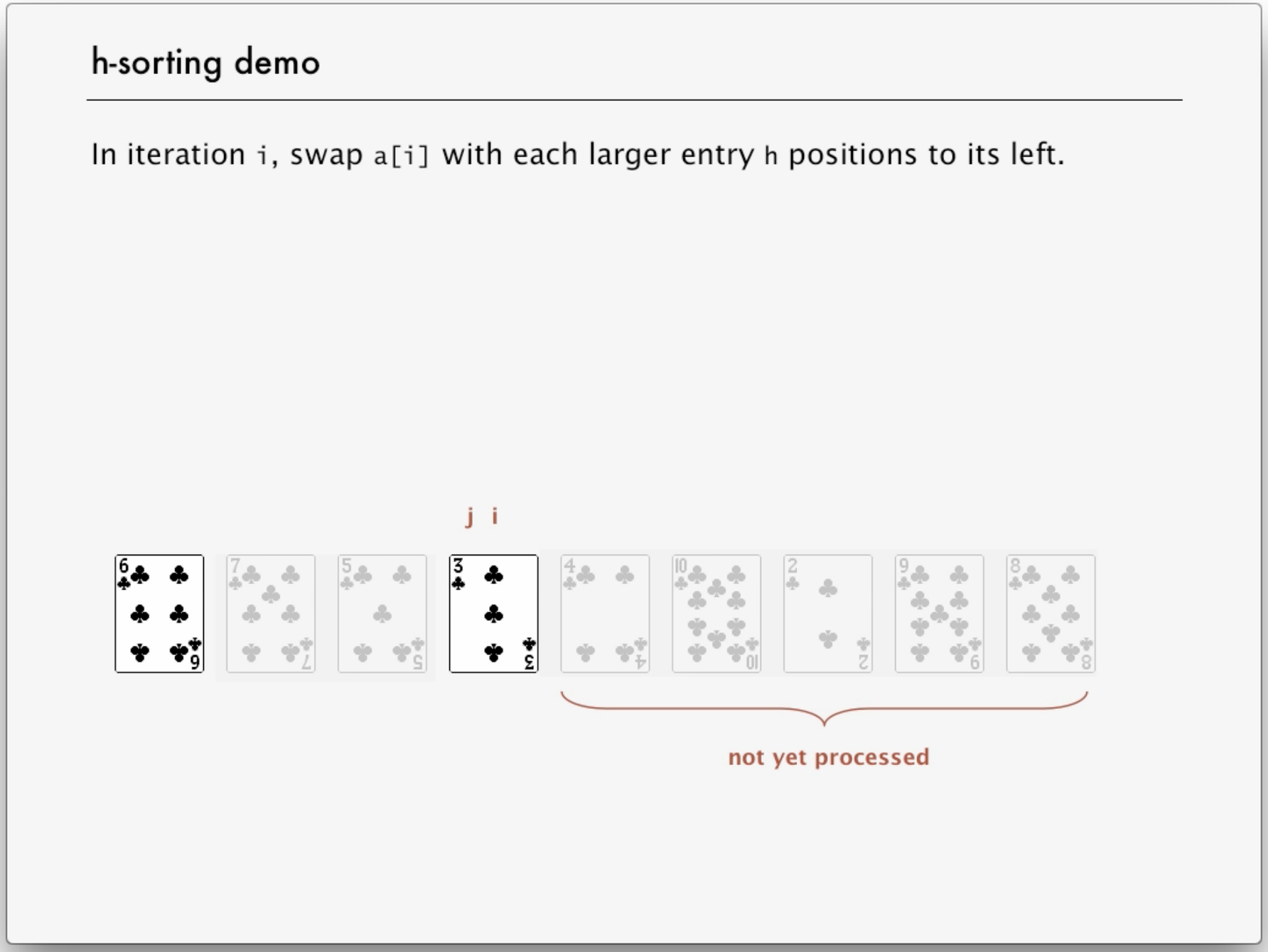


Visual trace of shellsort

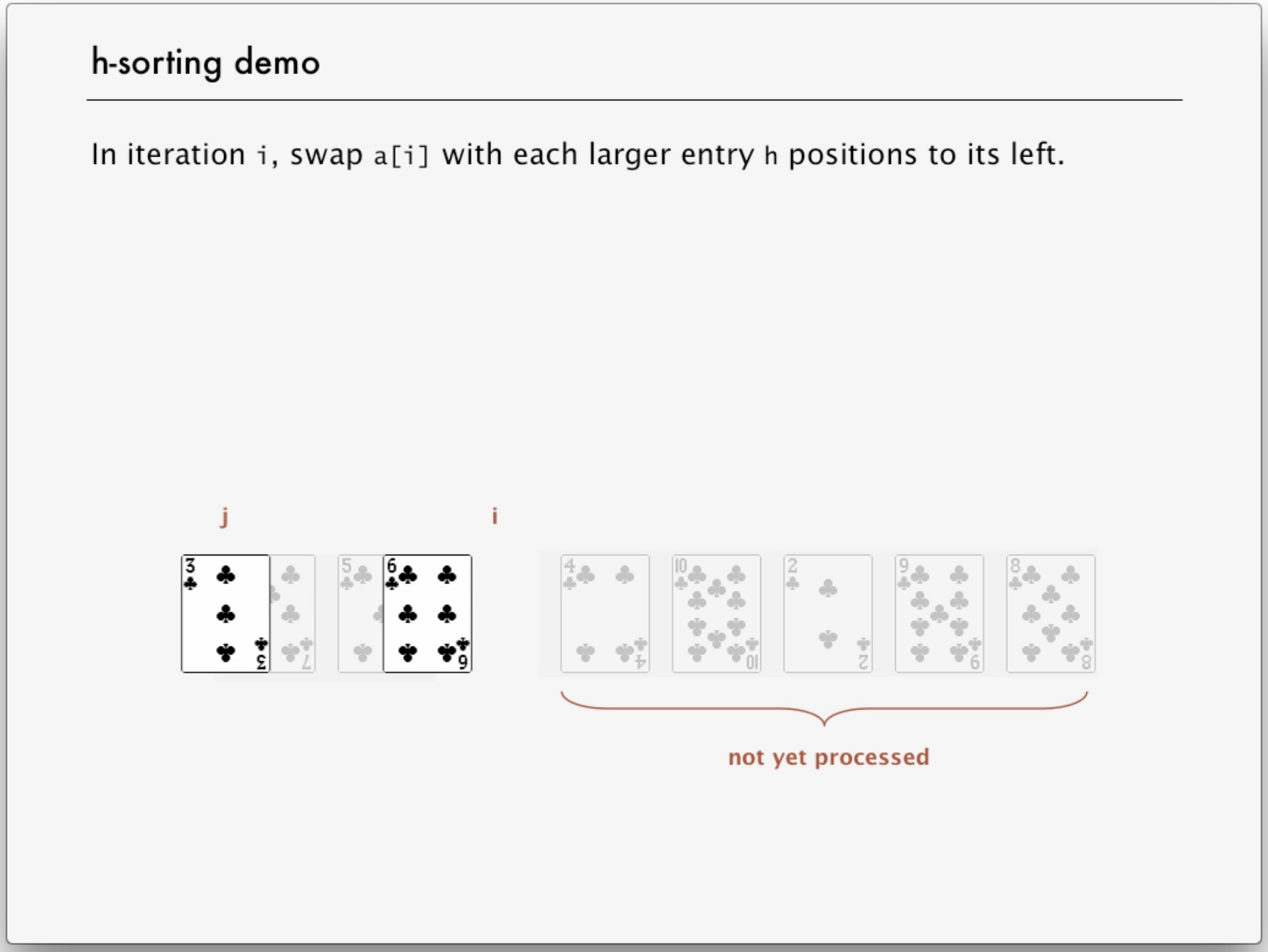
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.

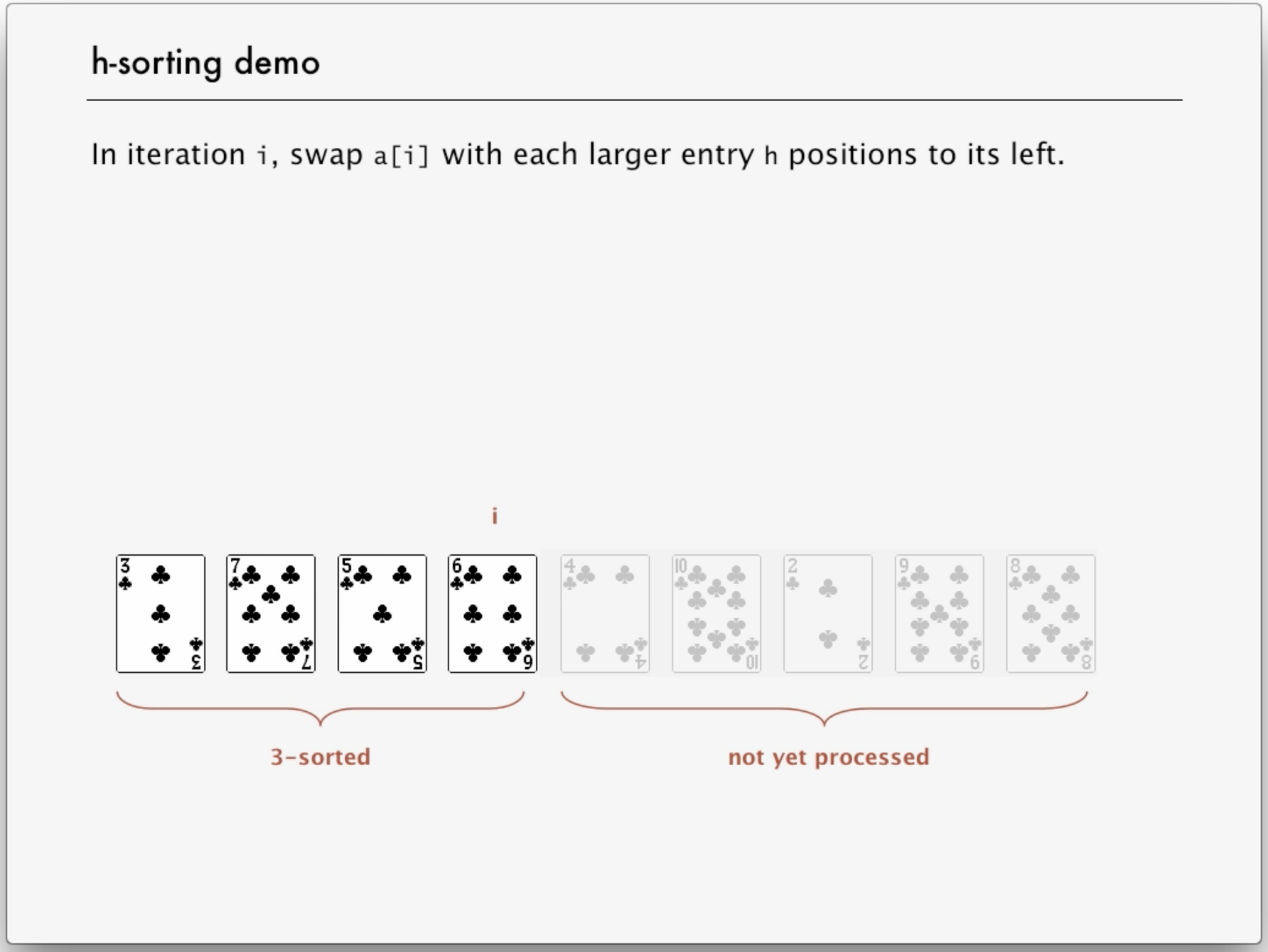
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i
not yet processed

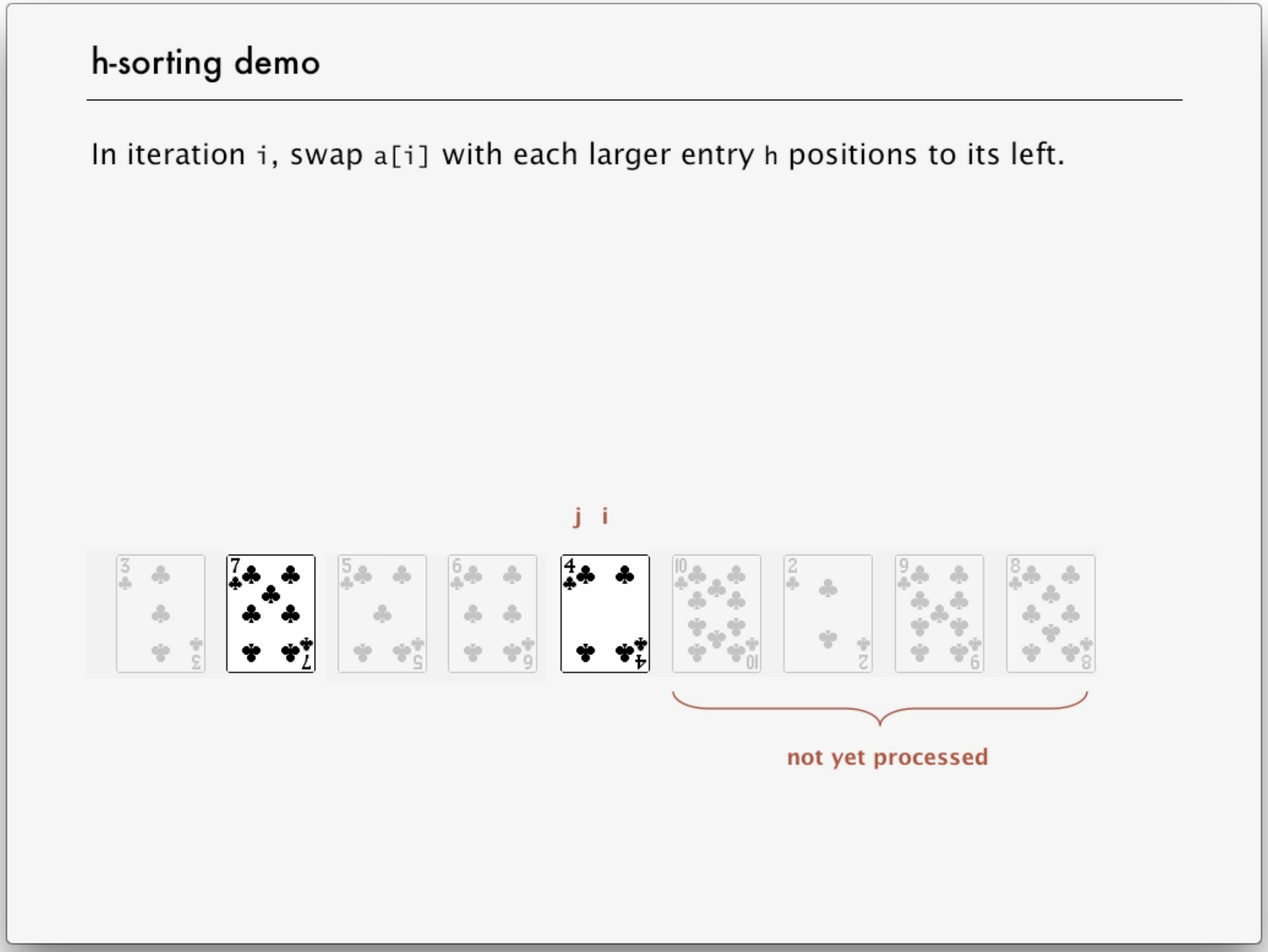
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j
i
not yet processed

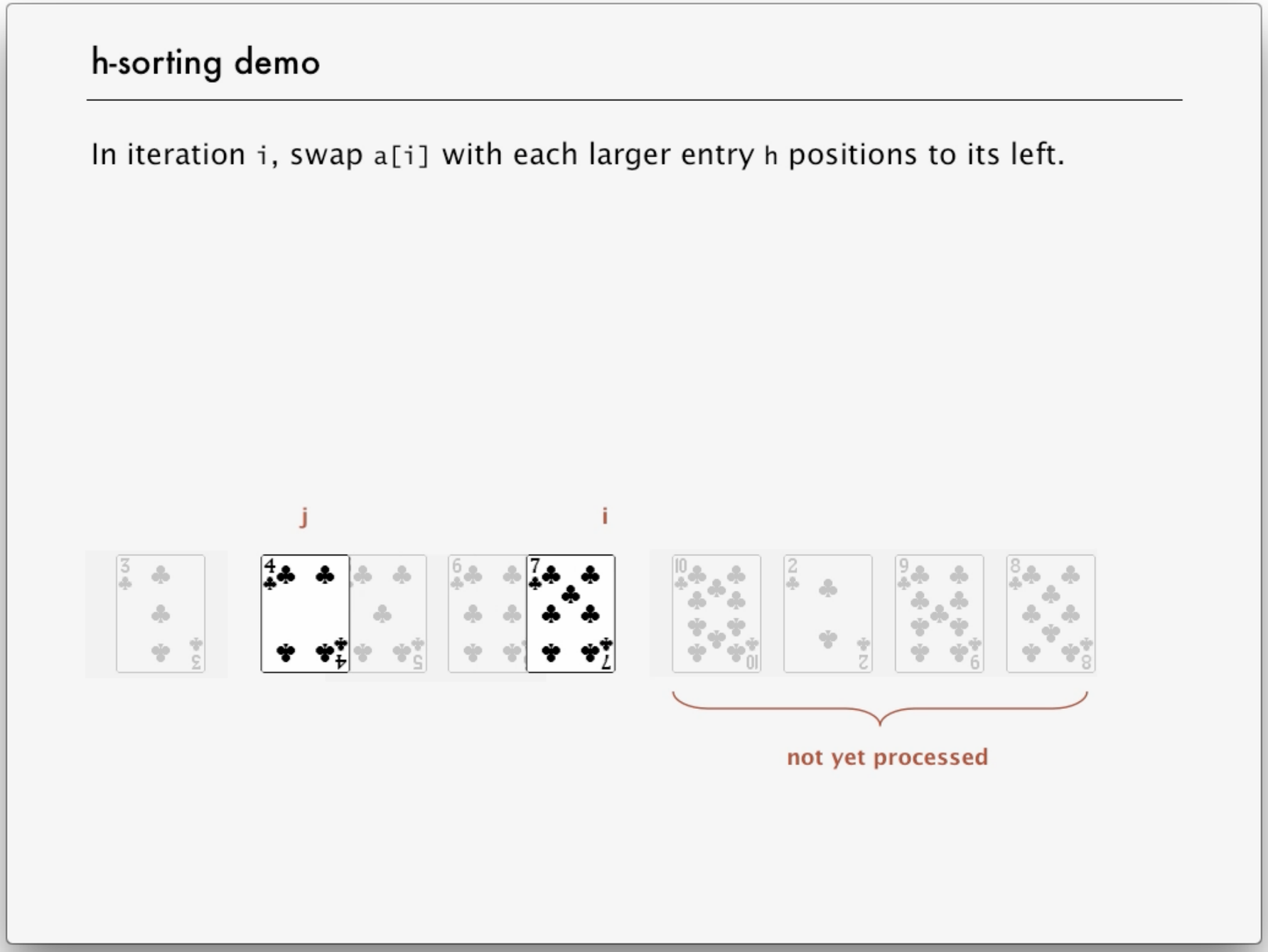
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

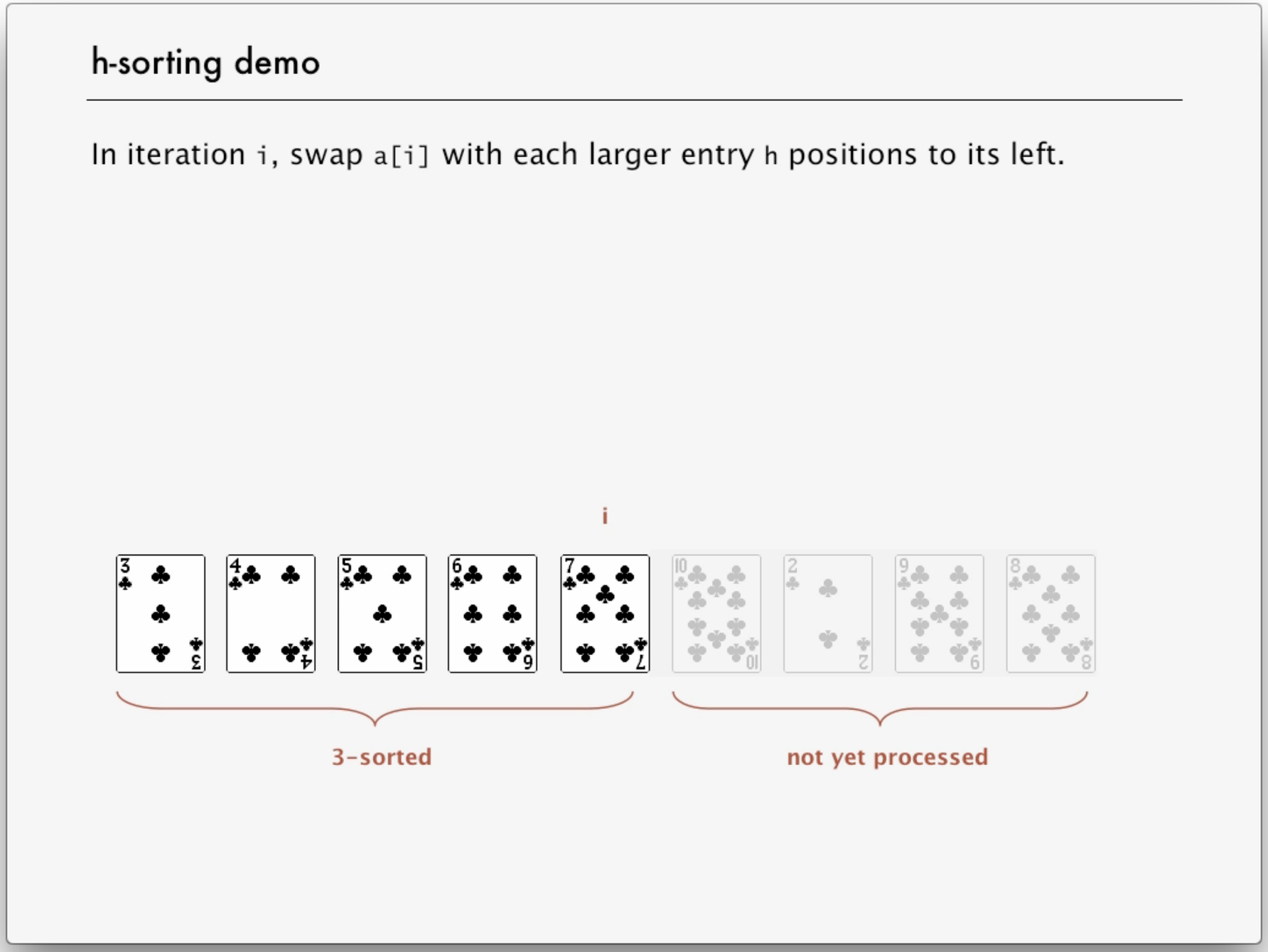
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i
not yet processed

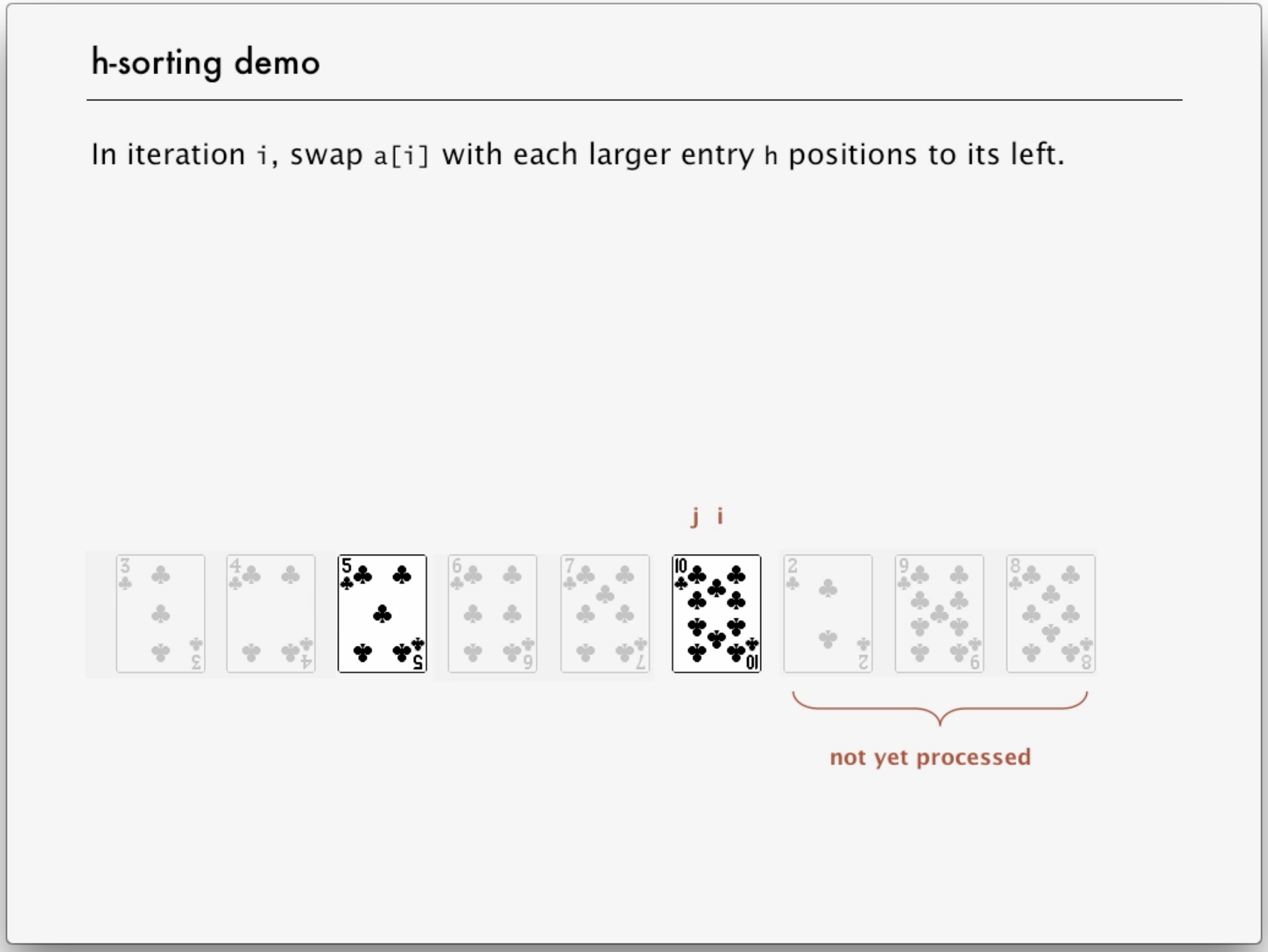
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j
i
not yet processed

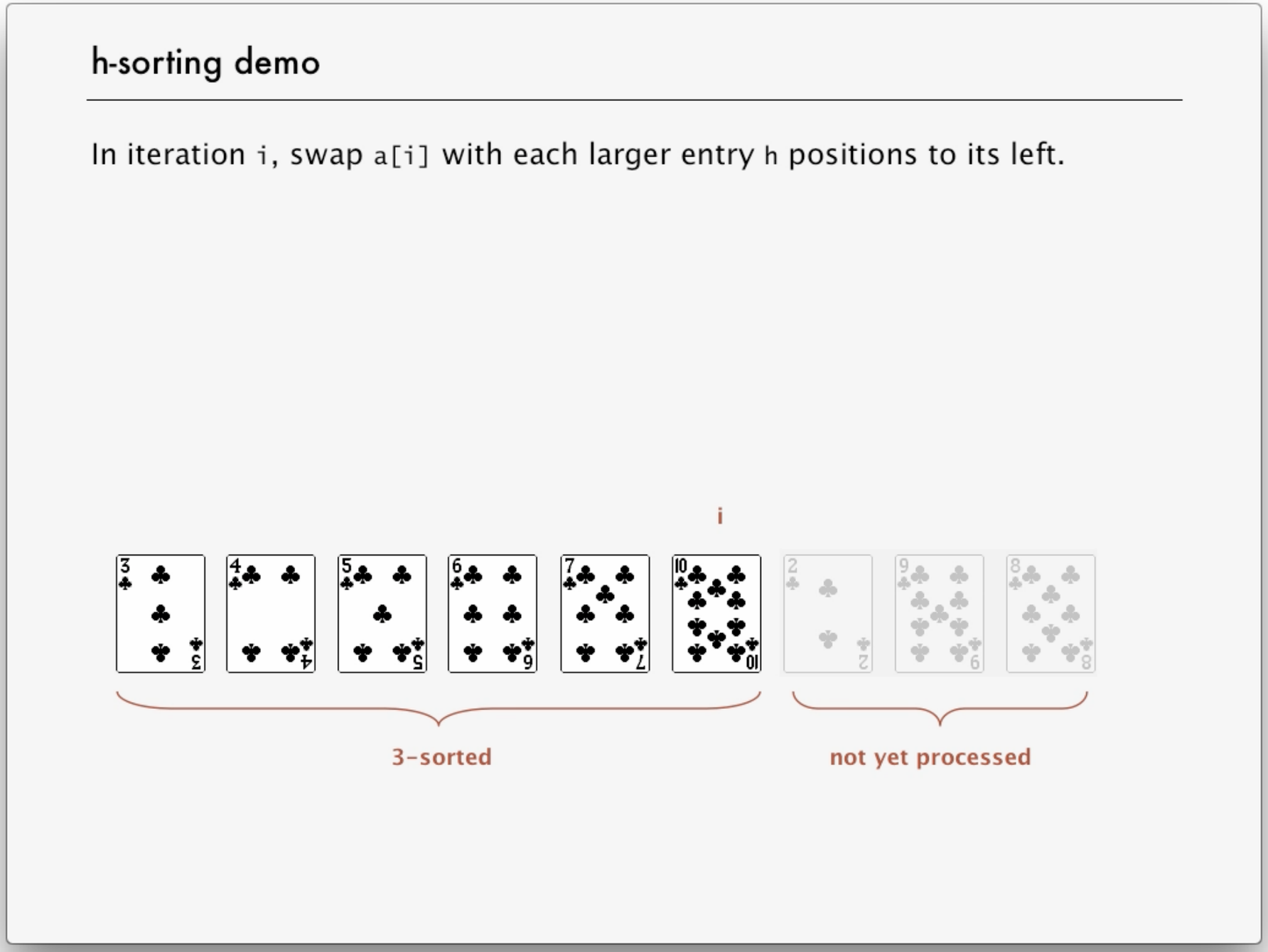
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

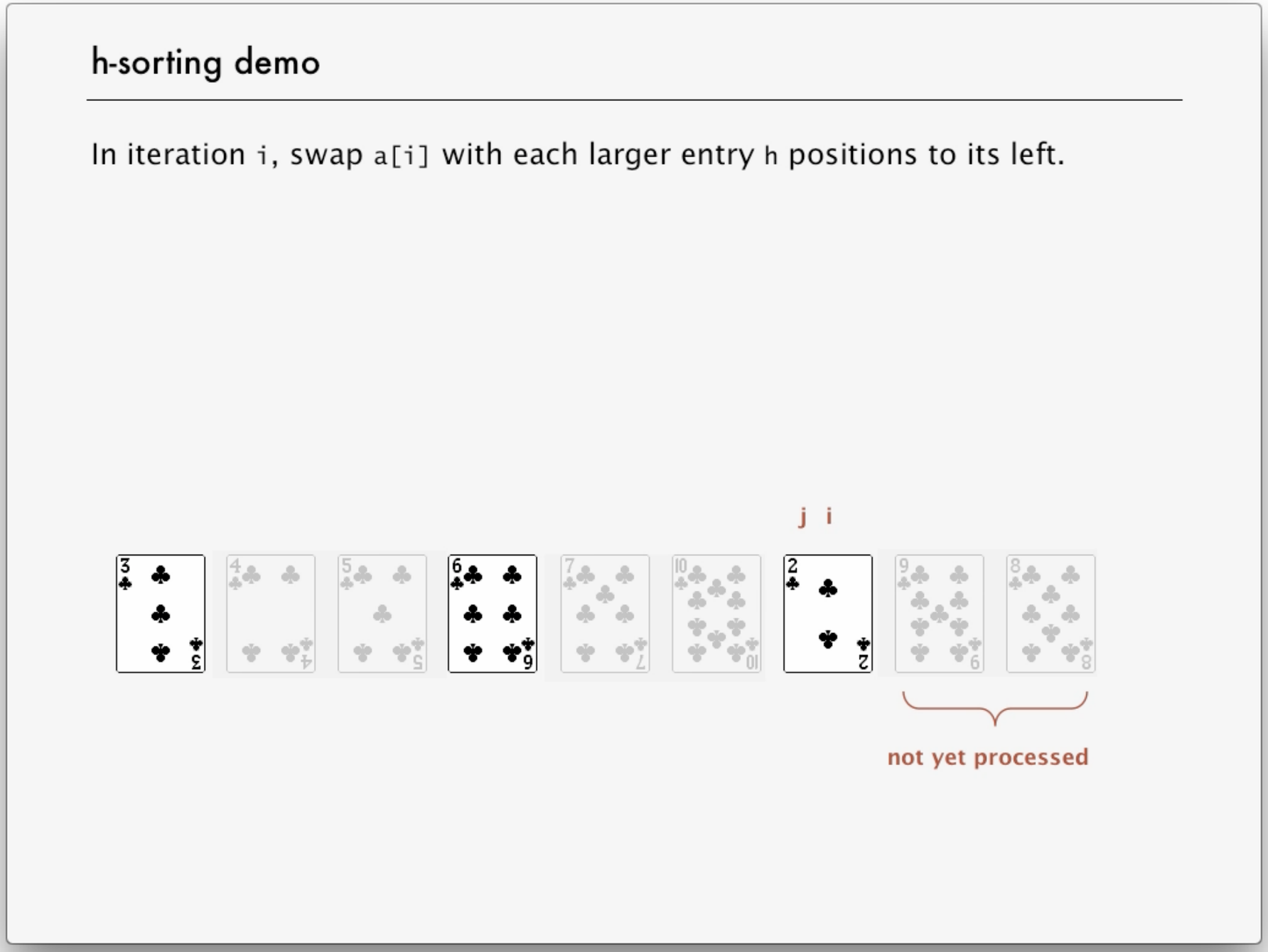
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i
not yet processed

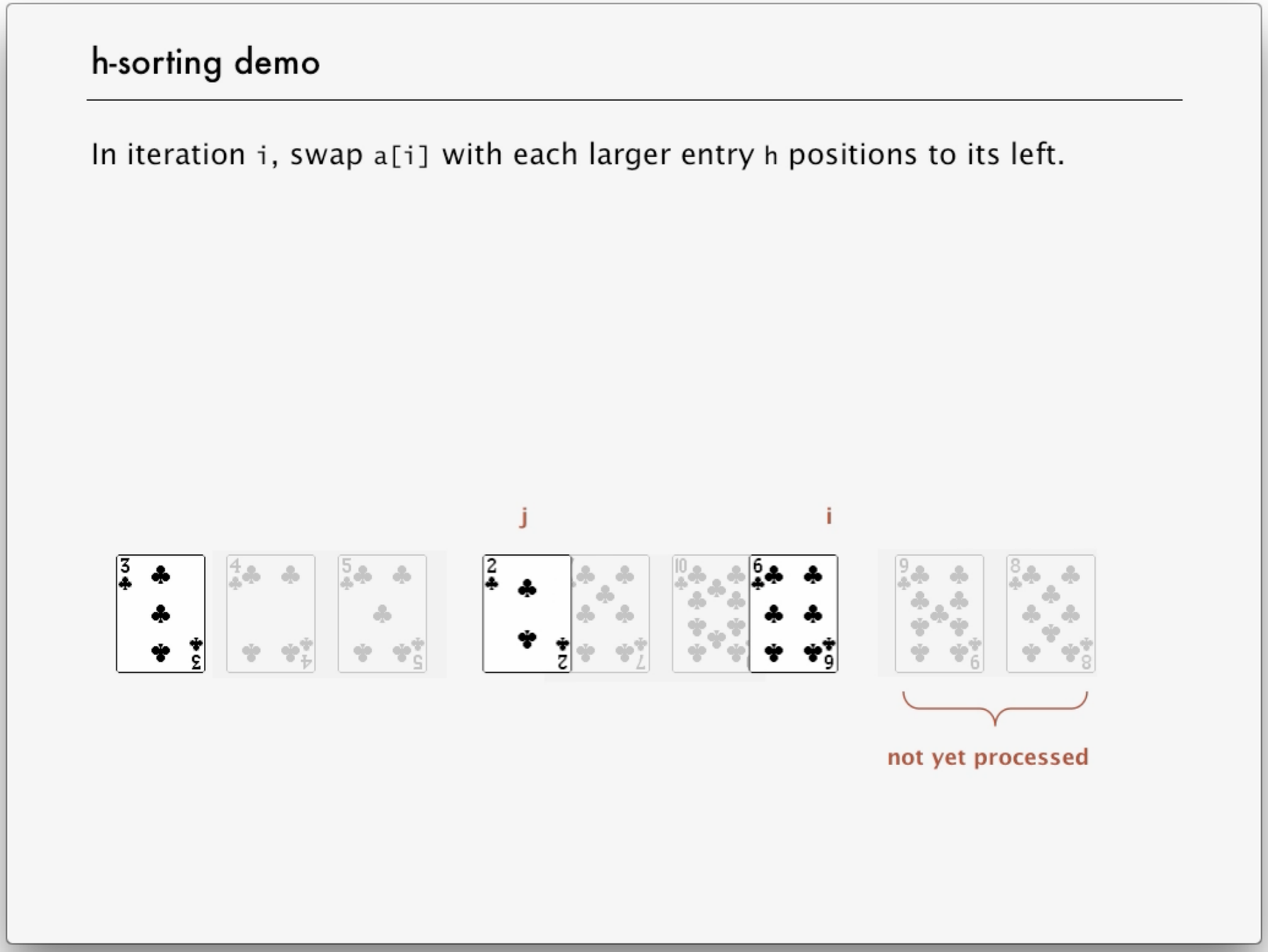
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

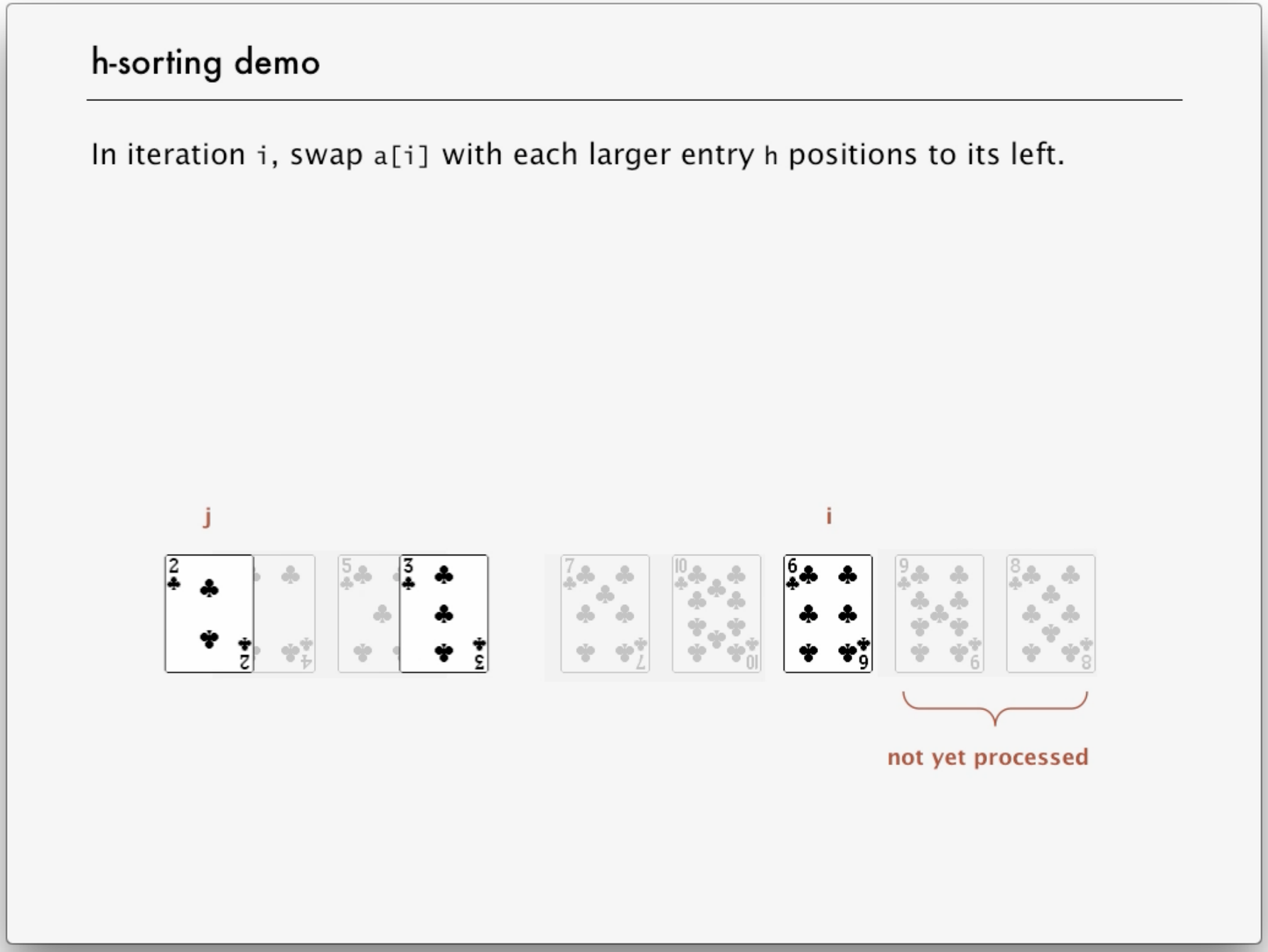
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i
not yet processed

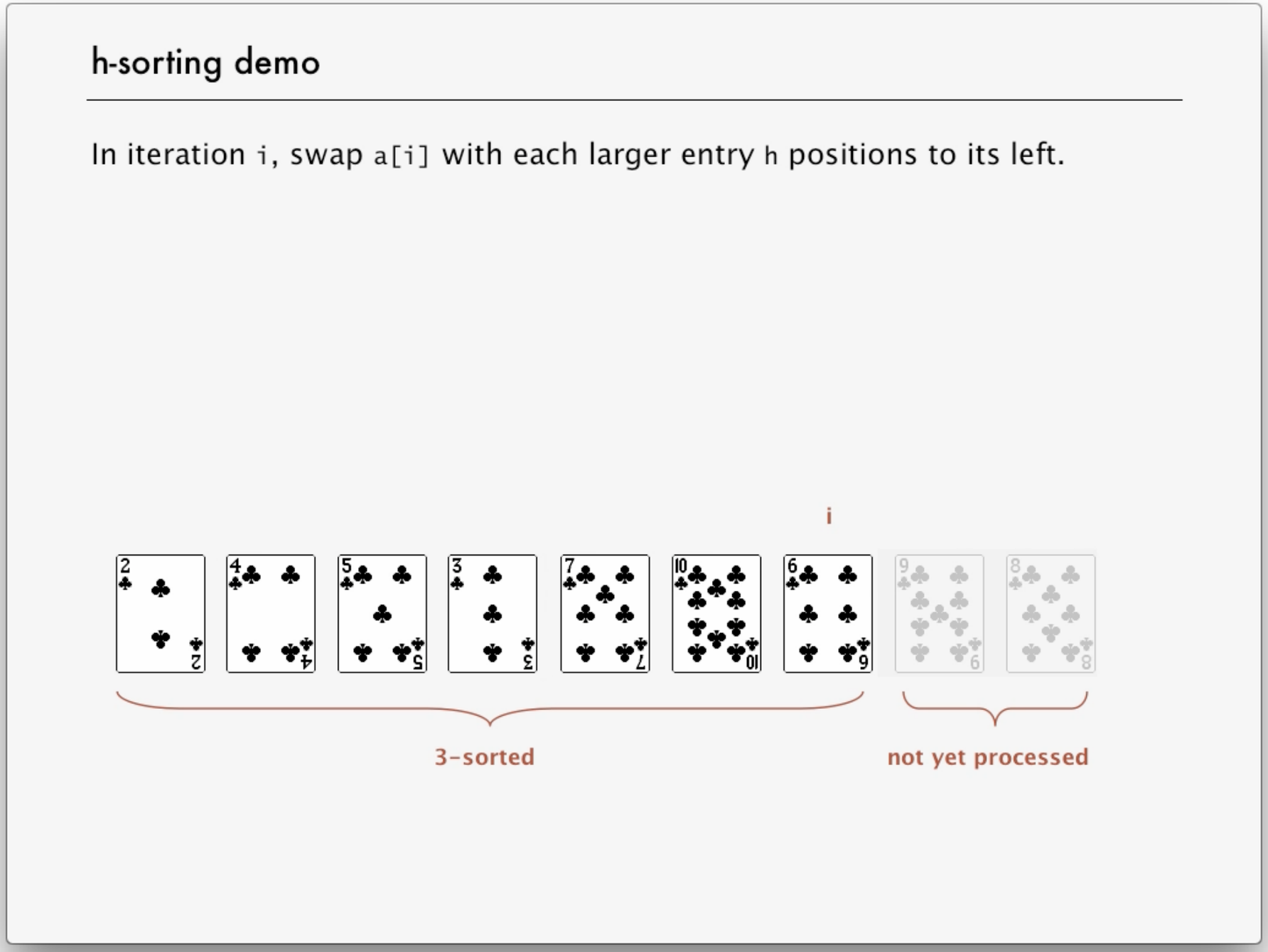
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j
i
not yet processed

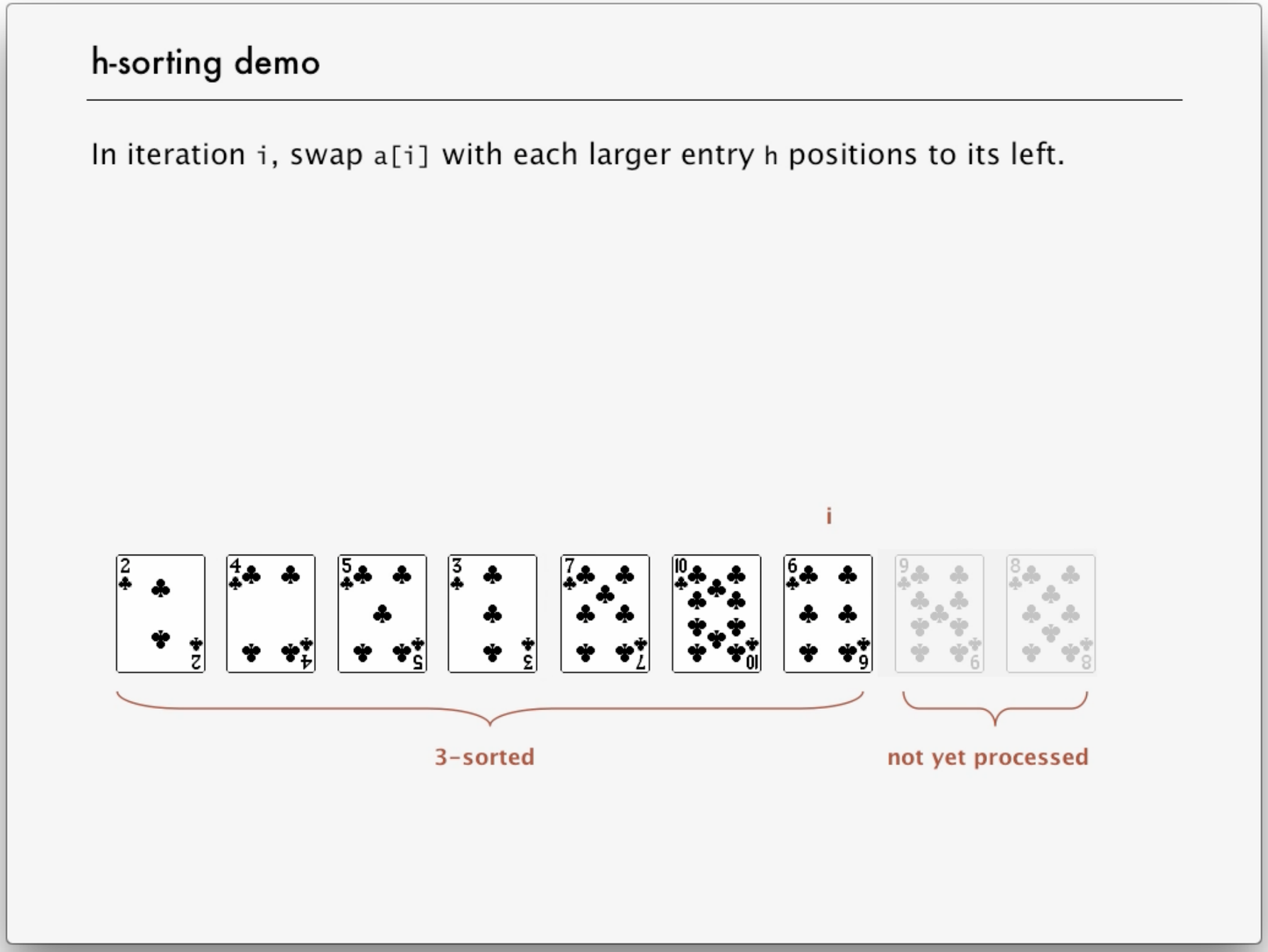
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j
i
not yet processed

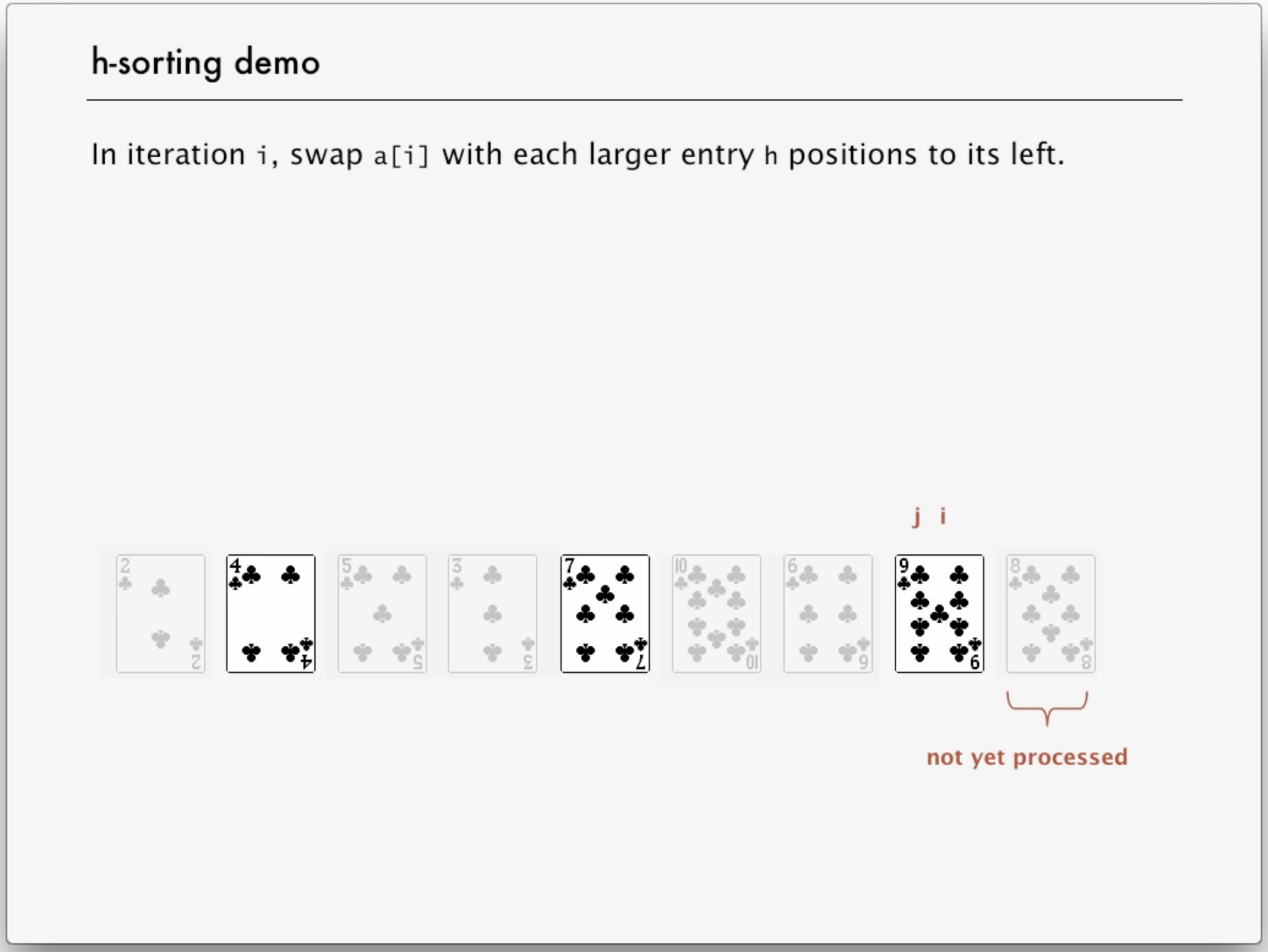
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

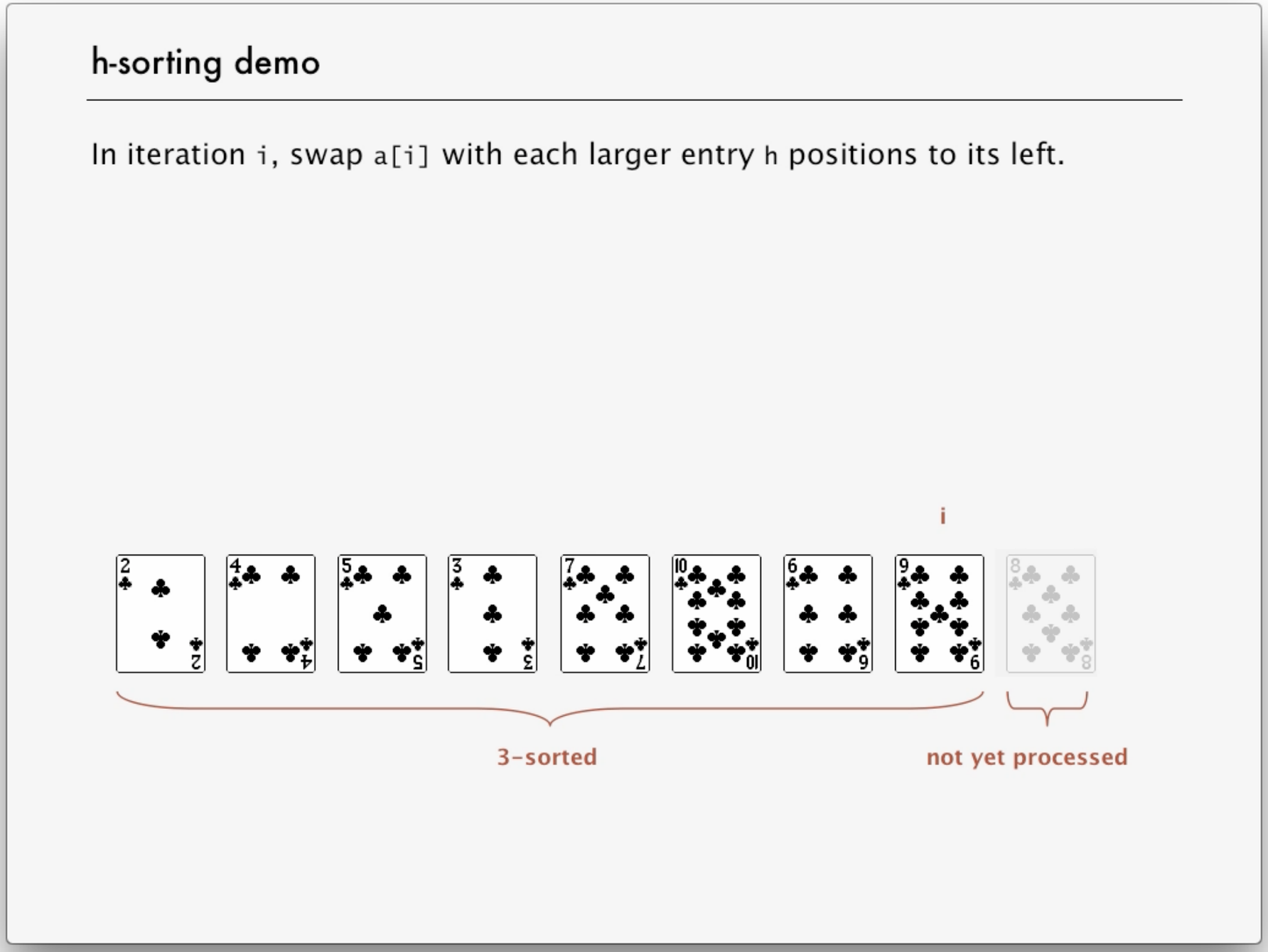
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

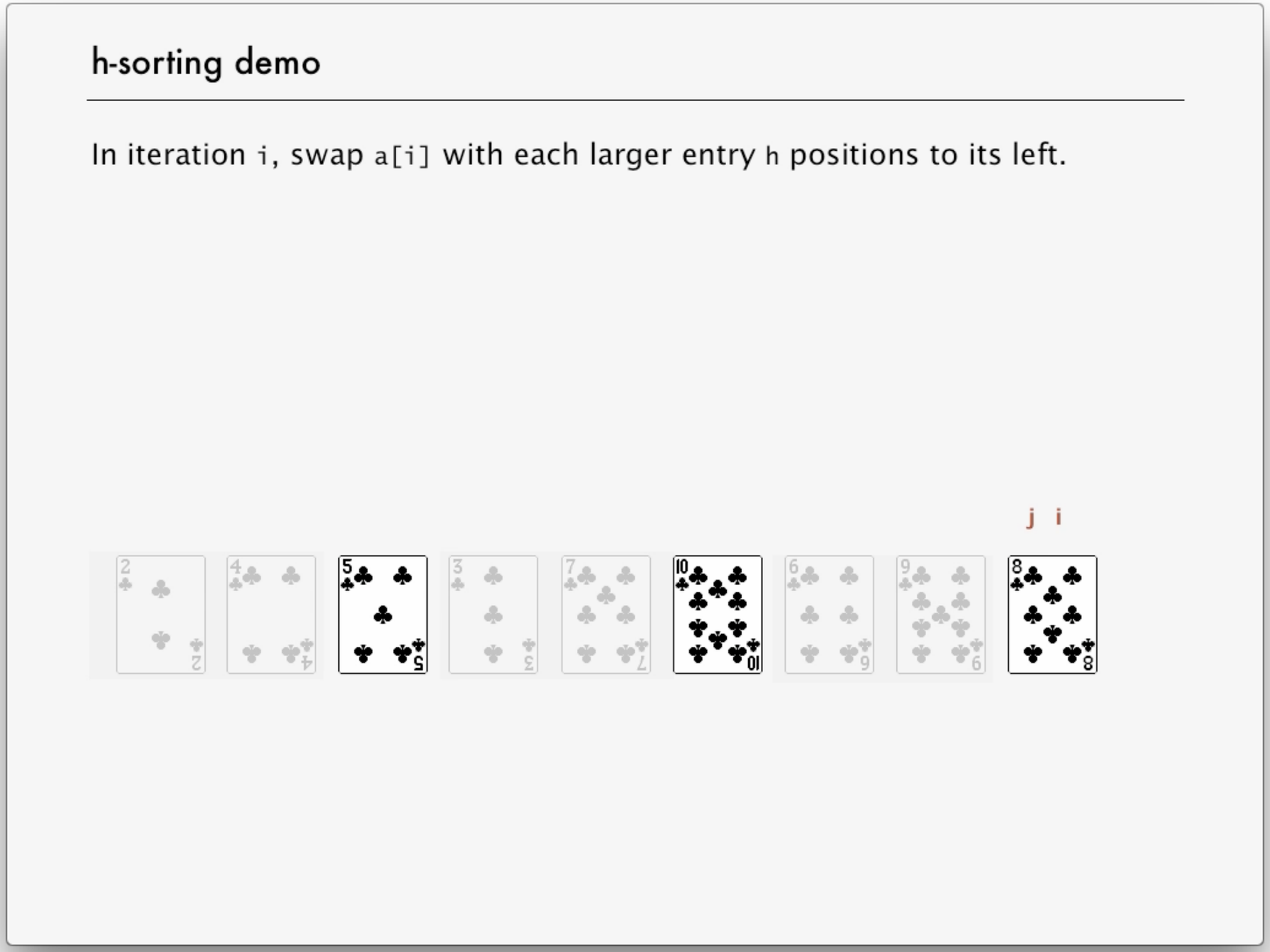
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i
not yet processed

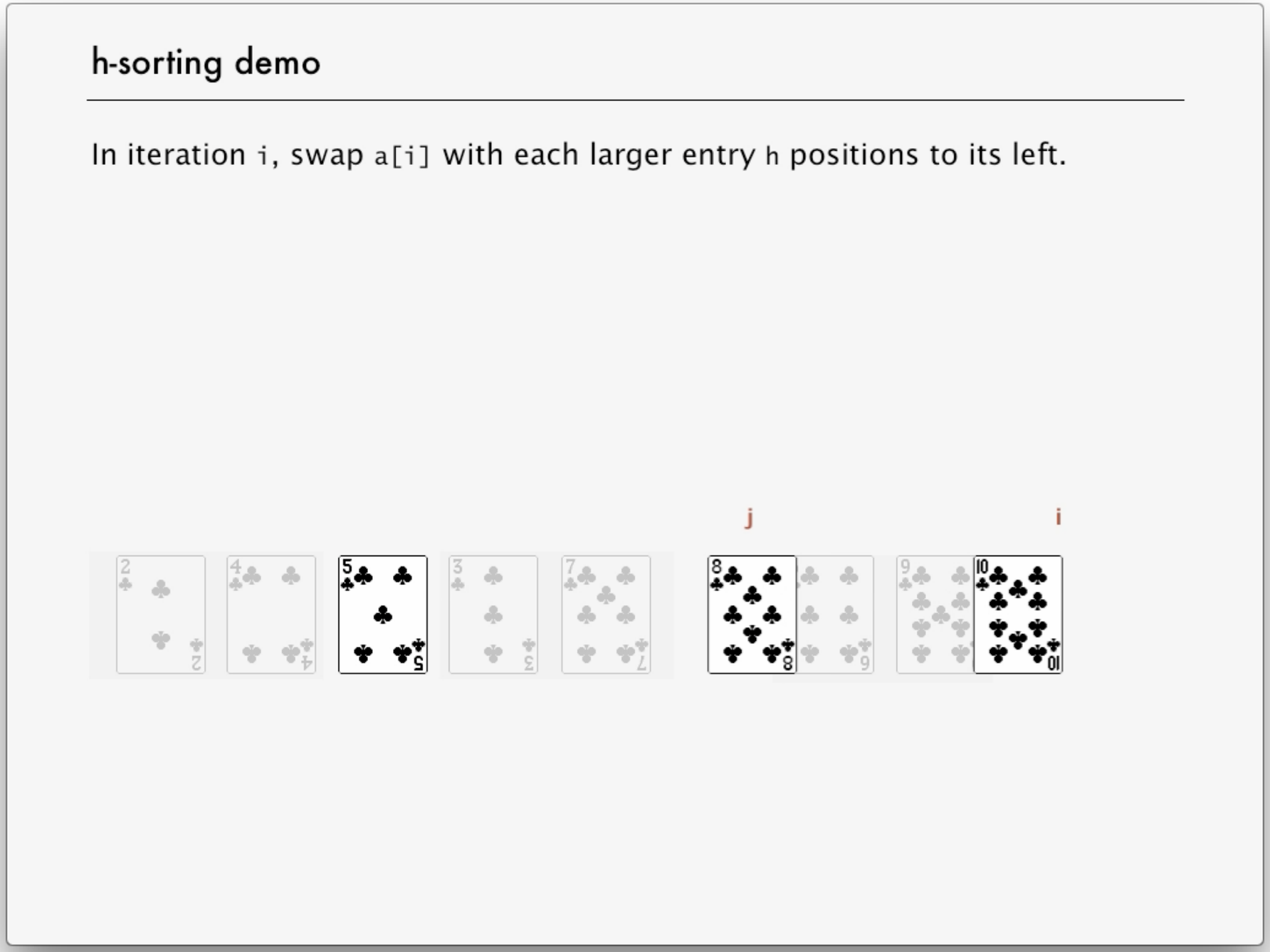
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
i
3-sorted
not yet processed

h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j i

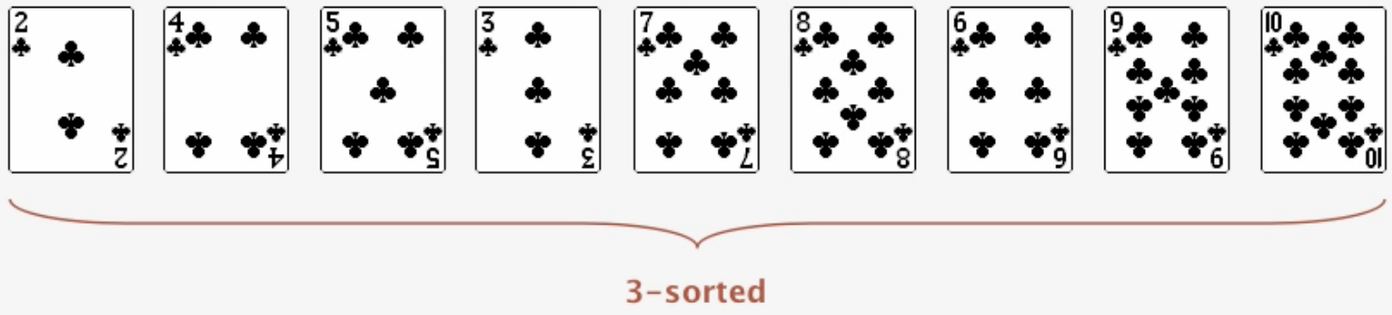
h-Sort Demo


h-sorting demo
In iteration i, swap a[i] with each larger entry h positions to its left.
j
i

h-Sort Demo

h-sorting demo

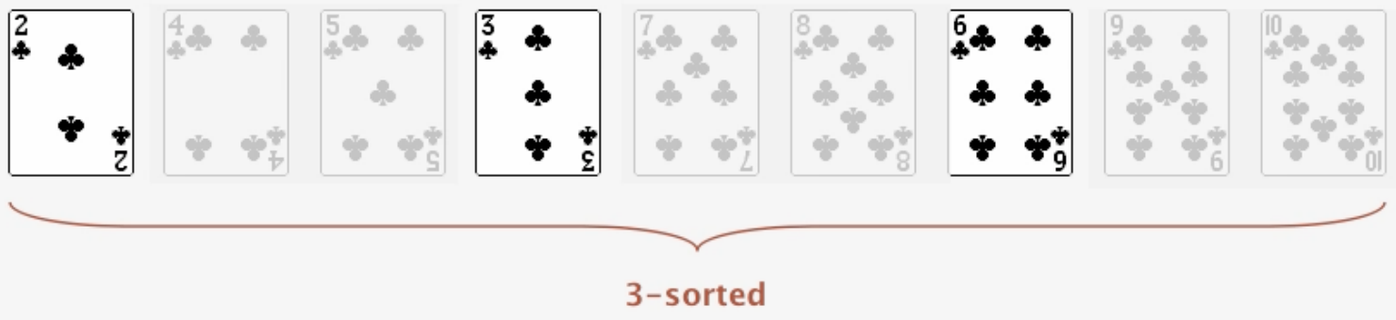
An array is `h`-sorted if `a[i-h] <= a[i]` for each `i`.



h-Sort Demo

h-sorting demo

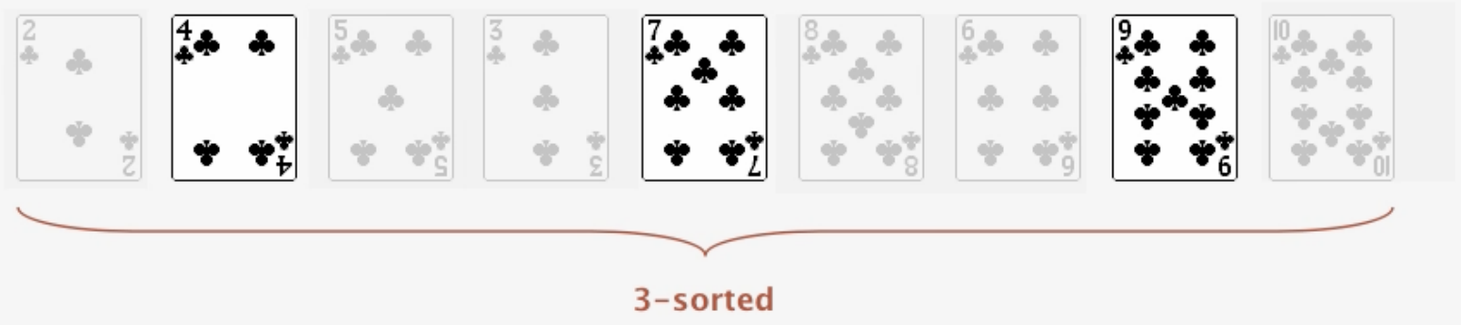
An array is `h`-sorted if `a[i-h] <= a[i]` for each `i`.



h-Sort Demo

h-sorting demo

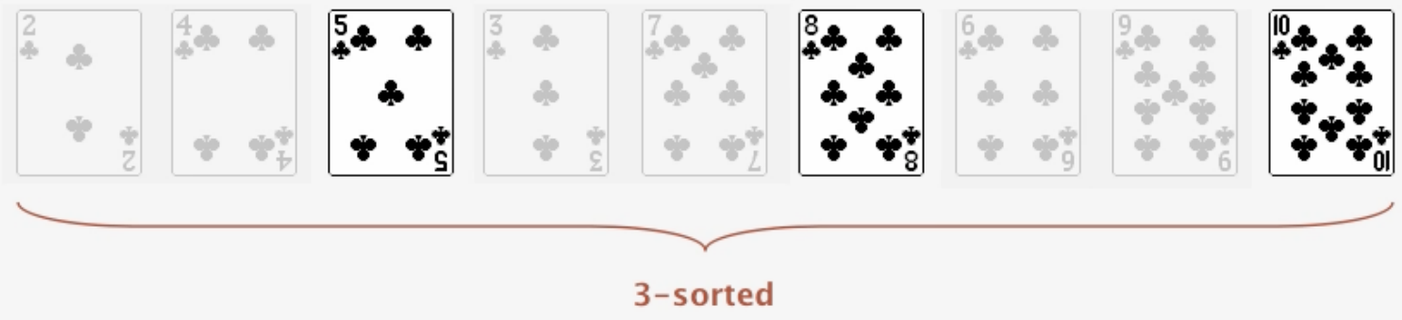
An array is `h`-sorted if `a[i-h] <= a[i]` for each `i`.



h-Sort Demo

h-sorting demo

An array is `h`-sorted if `a[i-h] <= a[i]` for each `i`.



Shellsort

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 ⇒ small subarray.
- Small increments ⇒ nearly in order. [stay tuned]

Shellsort


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 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

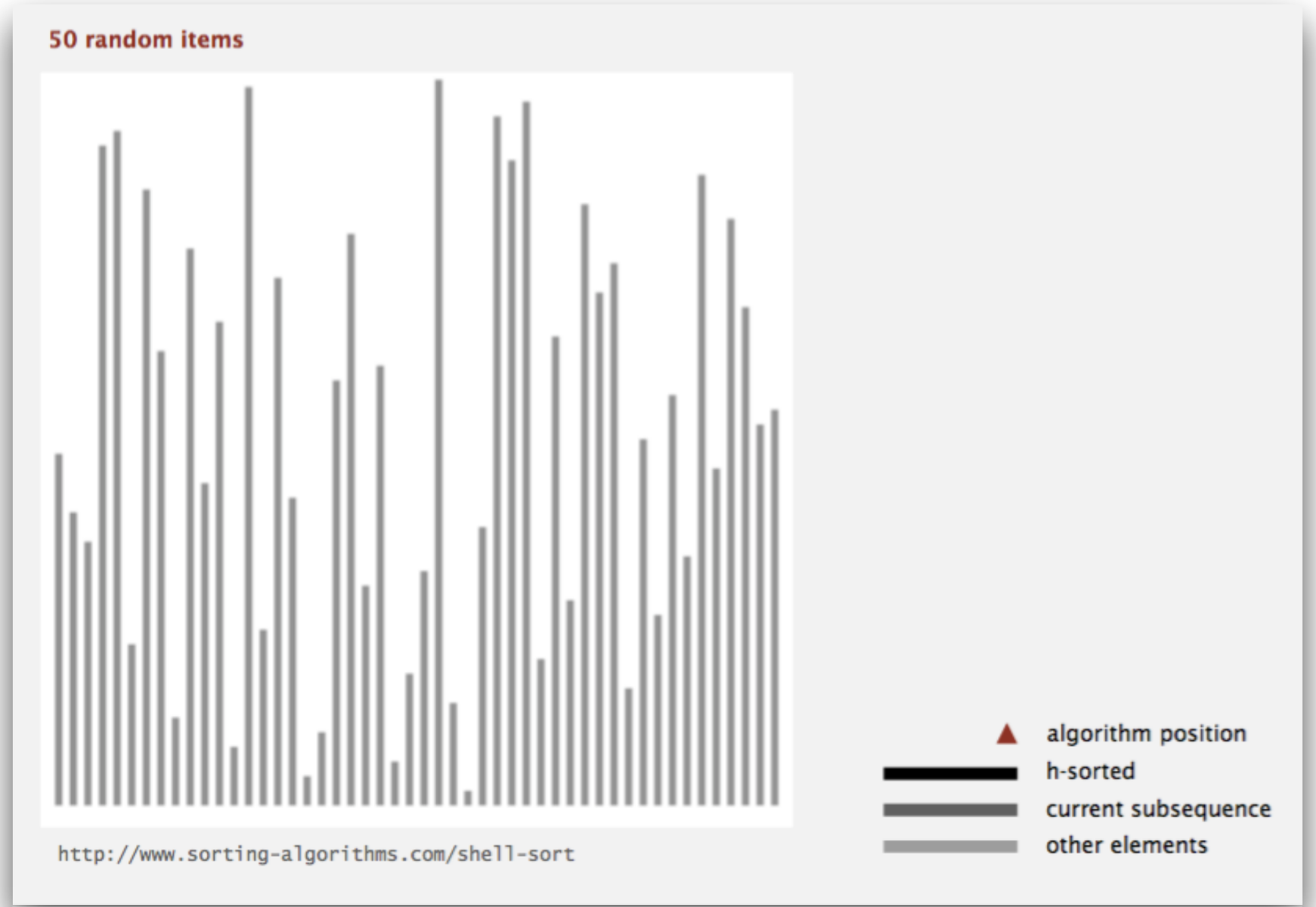
```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

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

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

3x+1 increment sequence

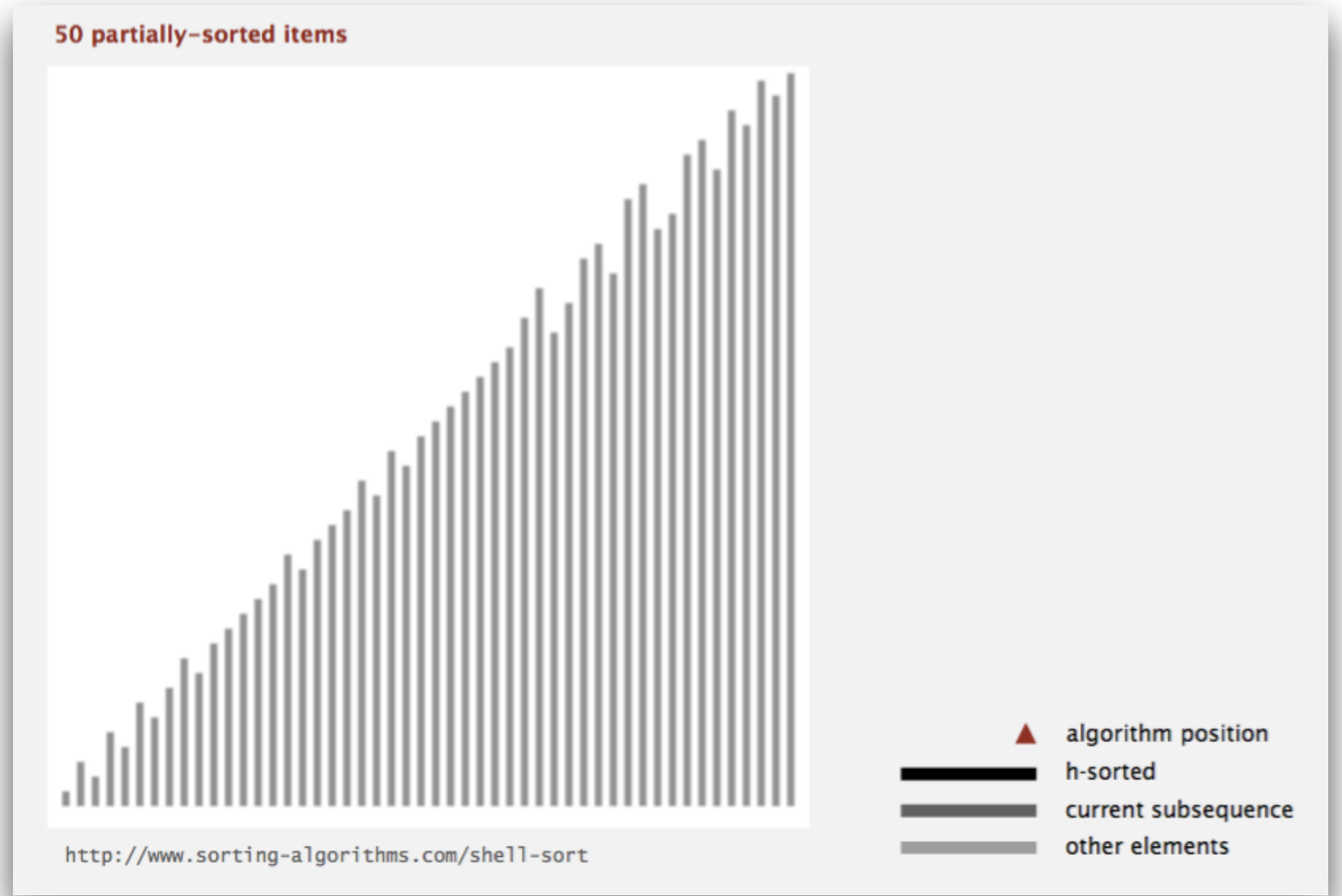
insertion sort

move to next increment

Shellsort


50 random items
algorithm position
h-sorted
current subsequence
other elements
http://www.sorting-algorithms.com/shell-sort

Shellsort


50 partially-sorted items
http://www.sorting-algorithms.com/shell-sort
algorithm position
h-sorted
current subsequence
other elements

Shellsort

Powers of two. 1, 2, 4, 8, 16, 32, ...
No.

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

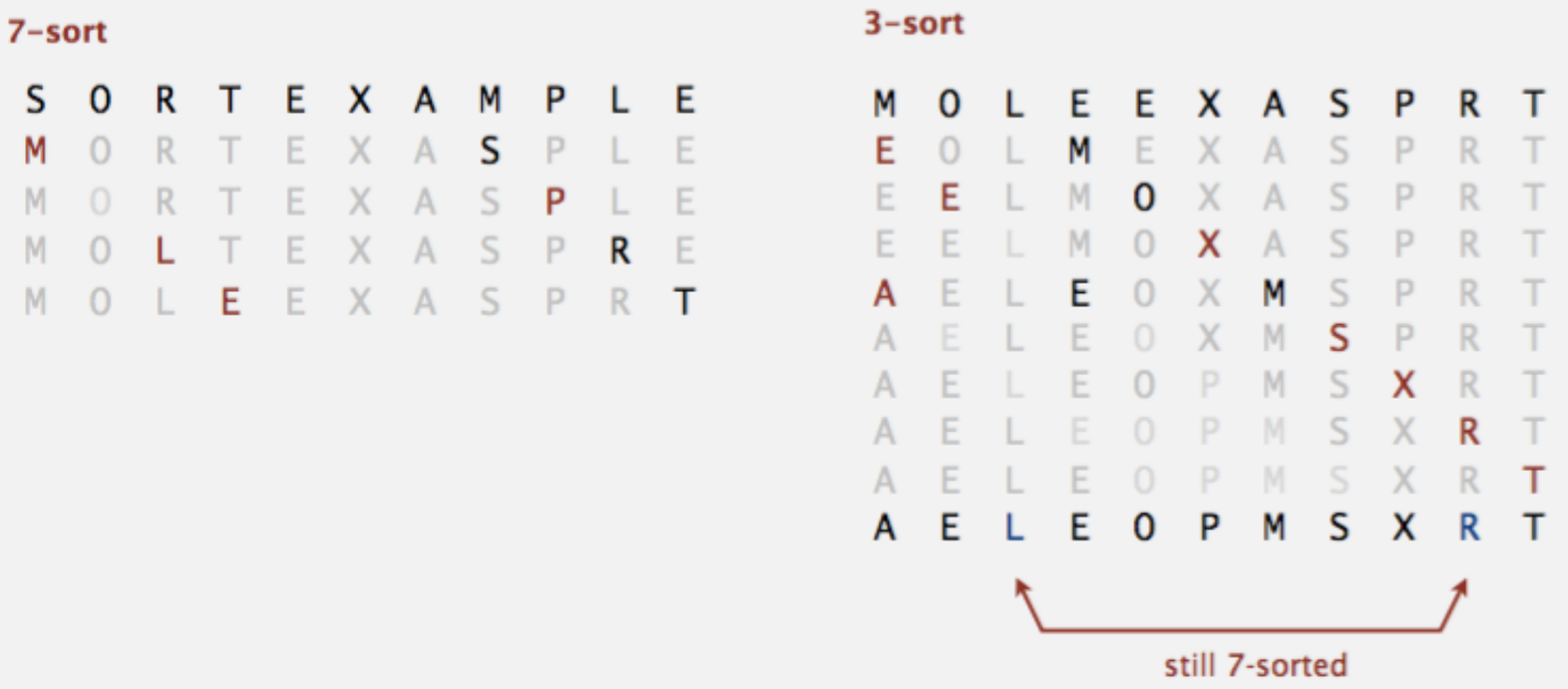
→ 3x + 1. 1, 4, 13, 40, 121, 364, ...
OK. Easy to compute.

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...
Good. Tough to beat in empirical studies.

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

Shellsort

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



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

Shellsort

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

Property. The expected number of compares to shellsort a randomly-ordered array using 3x+1 increments is....

N	compares	2.5 N ln N	0.25 N ln ² N	N 1.3
5,000	93K	106K	91K	64K
10,000	209K	230K	213K	158K
20,000	467K	495K	490K	390K
40,000	1022K	1059K	1122K	960K
80,000	2266K	2258K	2549K	2366K

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

Shellsort

Example of simple idea leading to substantial performance gains.

Useful in practice.

R, bzip2, /linux/kernel/groups.c

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

uClibc

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.

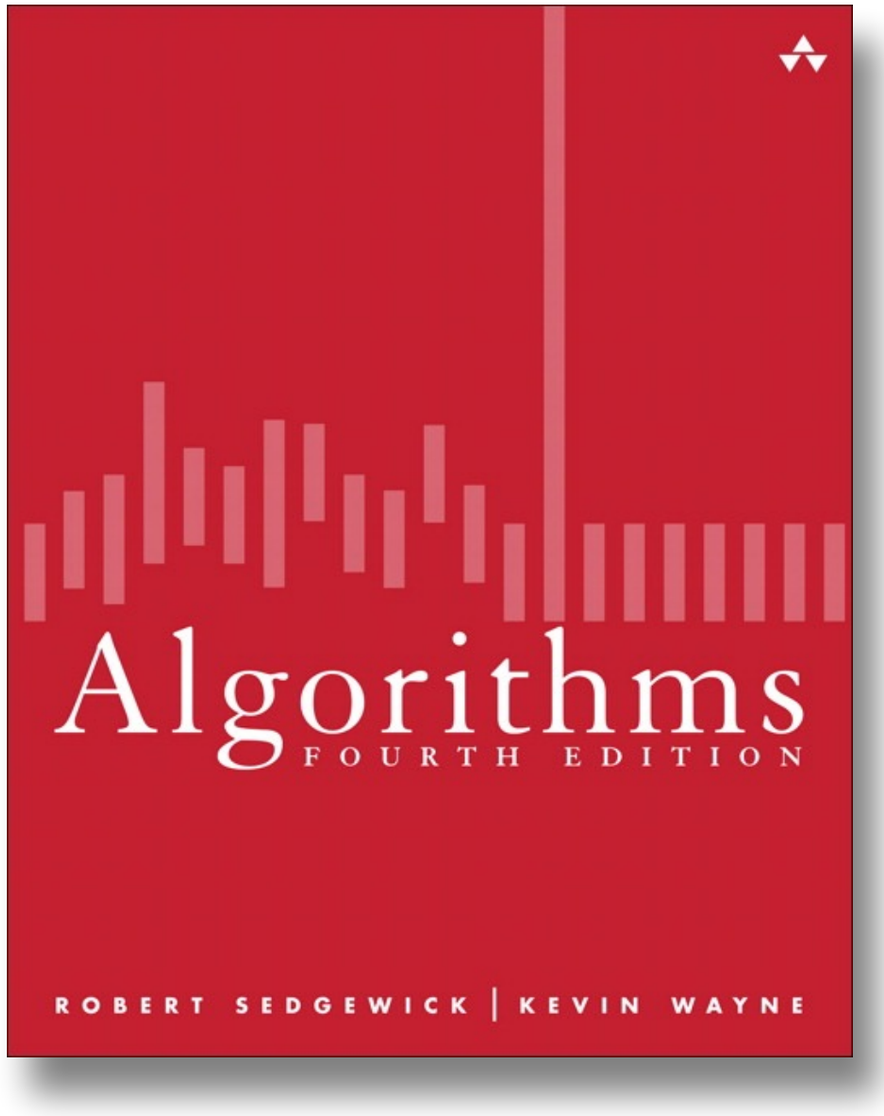
Shellsort

Today. Elementary sorting algorithms.

algorithm	best	average	worst
selection sort	N^2	N^2	N^2
insertion sort	N	N^2	N^2
Shellsort (3x+1)	$N \log N$	?	$N^{3/2}$
goal	N	$N \log N$	$N \log N$

order of growth of running time to sort an array of N items

Lecture Overview



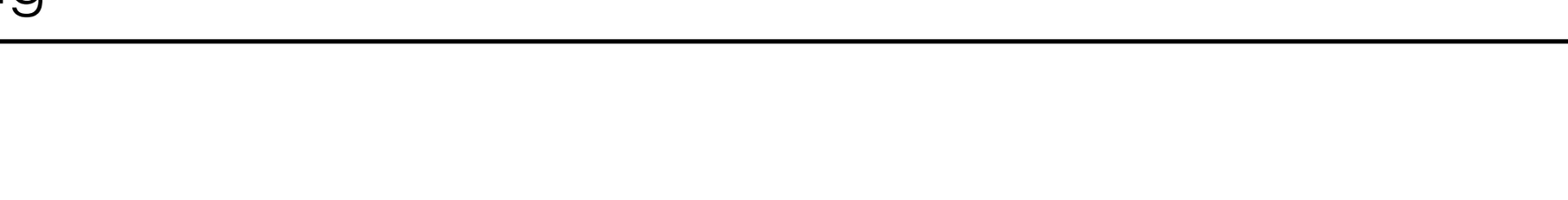
2.1 Elementary Sorts

- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Shuffling

Shuffling

Goal. Rearrange array so that result is a uniformly random permutation.

all permutations
equally likely



Shuffling

Goal. Rearrange array so that result is a uniformly random permutation.

all permutations
equally likely

Shuffling

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling columns in a spreadsheet

0.8003 0.9706 0.9157 0.9649 0.1576 0.4854 0.1419 0.4218 0.9572

Shuffling

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling columns in a spreadsheet

0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Shuffling

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling columns in a spreadsheet

0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Proposition. Shuffle sort produces a uniformly random permutation.

assuming real numbers uniformly at random (and no ties)

Shuffling

Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser in Windows 7.

Solution? Implement shuffle sort by making comparator always return a random answer.

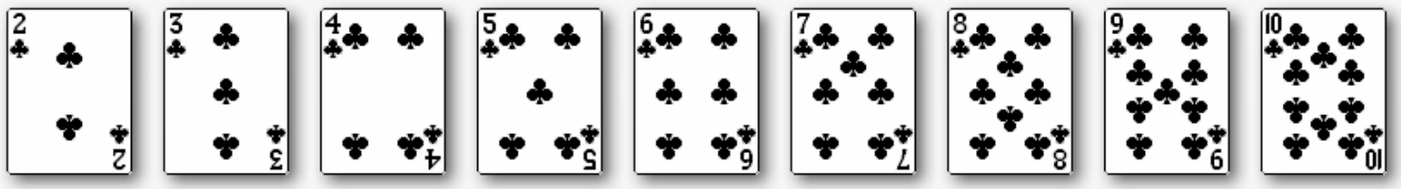
```
public int compareTo(Browser that)
{
   double r = Math.random();
   if (r < 0.5) return -1;
   if (r > 0.5) return +1;
   return 0;
}
```

browser comparator
(should implement a total order)

Knuth Shuffle Demo

Knuth shuffle

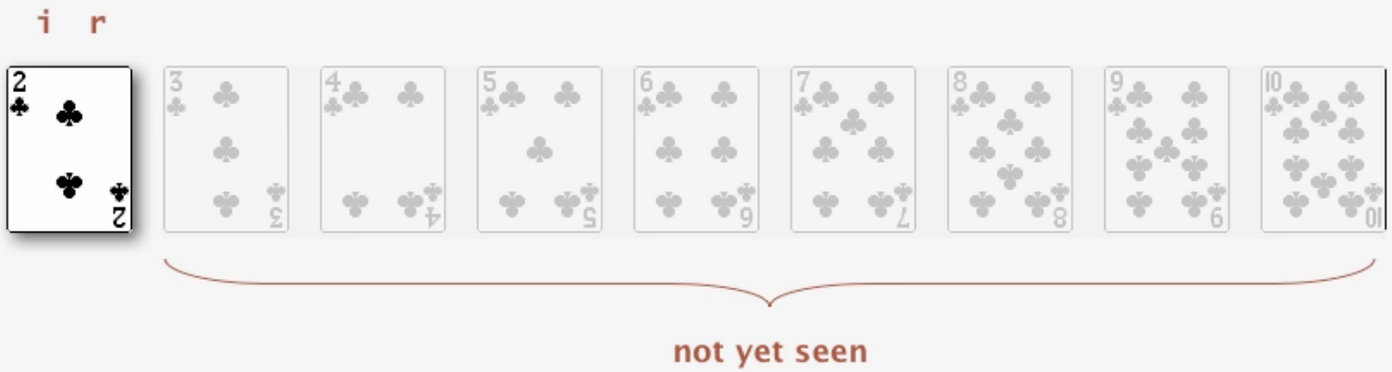
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

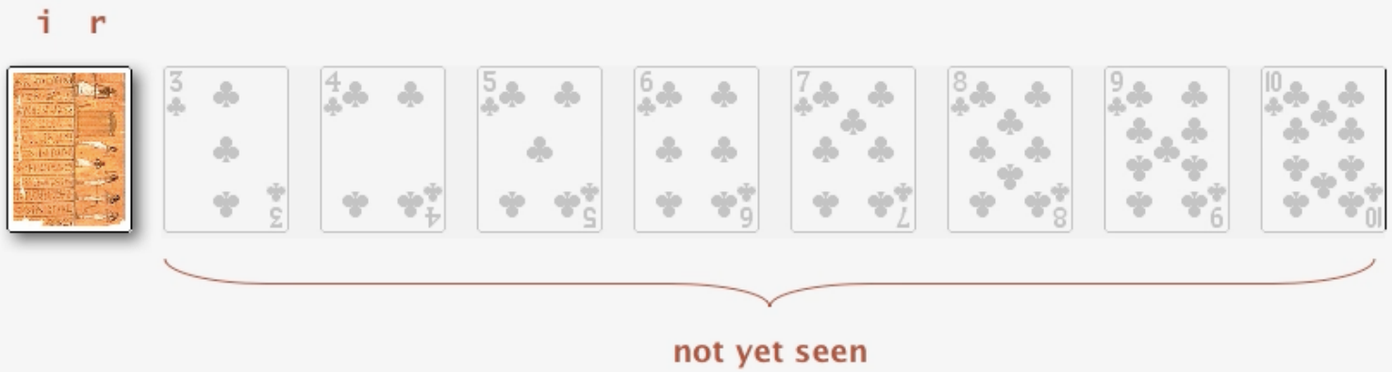
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

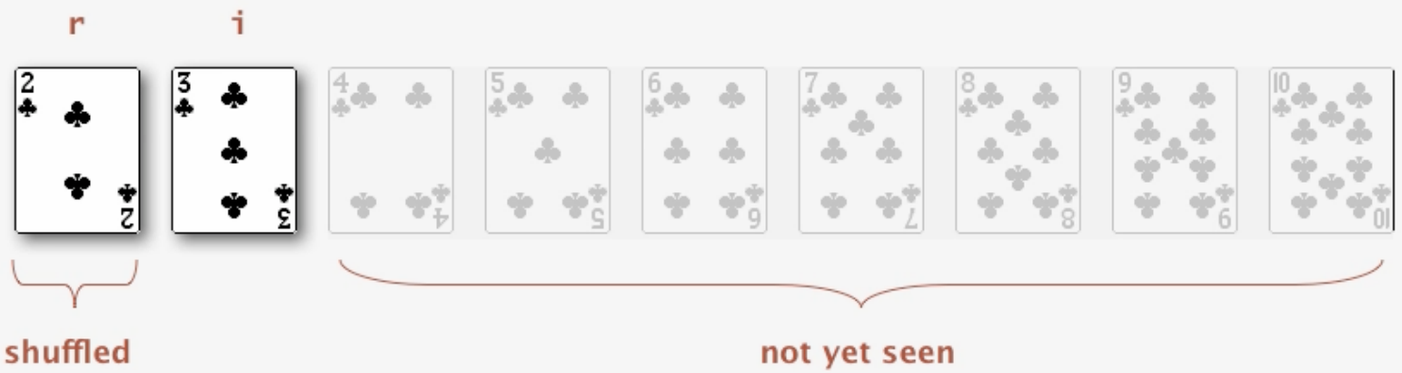
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.

Knuth Shuffle Demo

Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

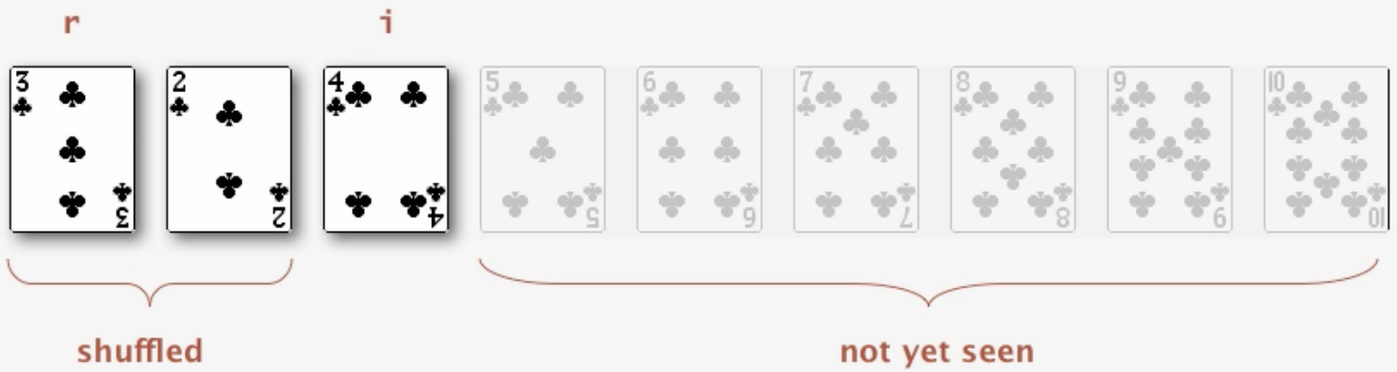
Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.

Knuth Shuffle Demo

Knuth shuffle

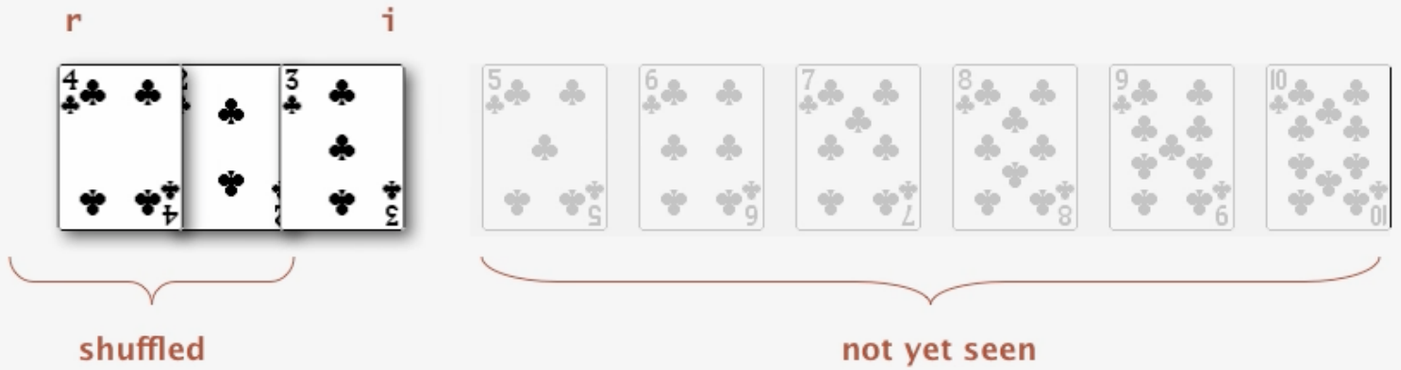
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

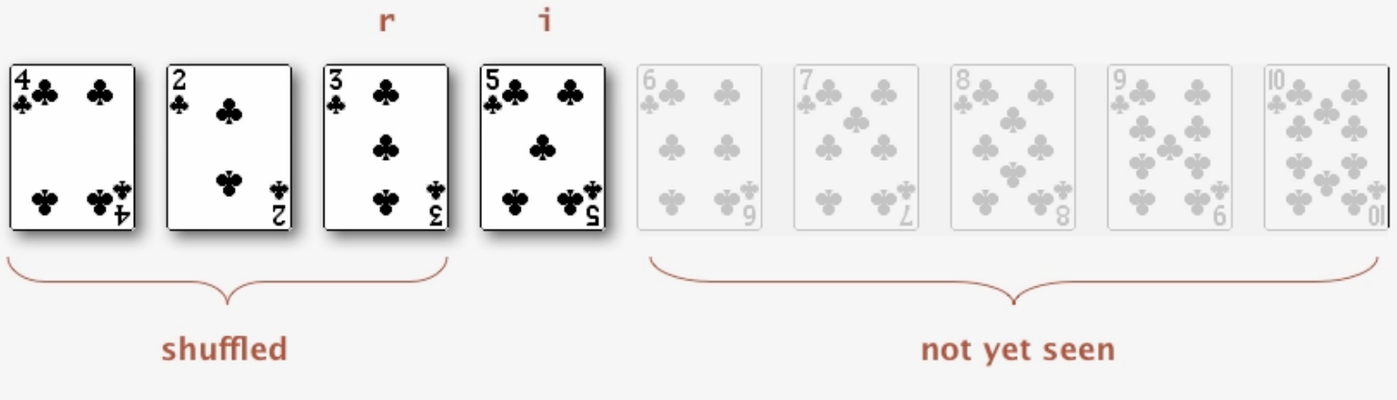
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

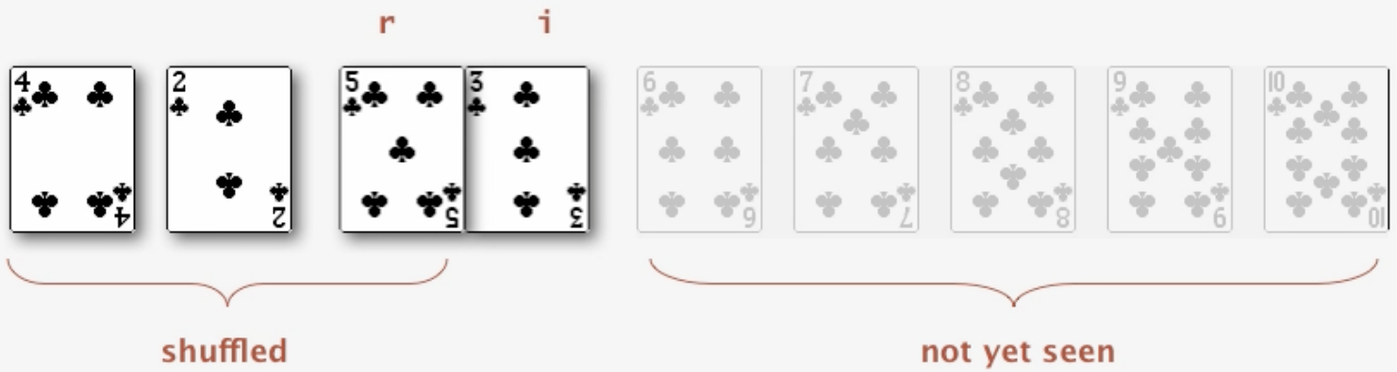
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

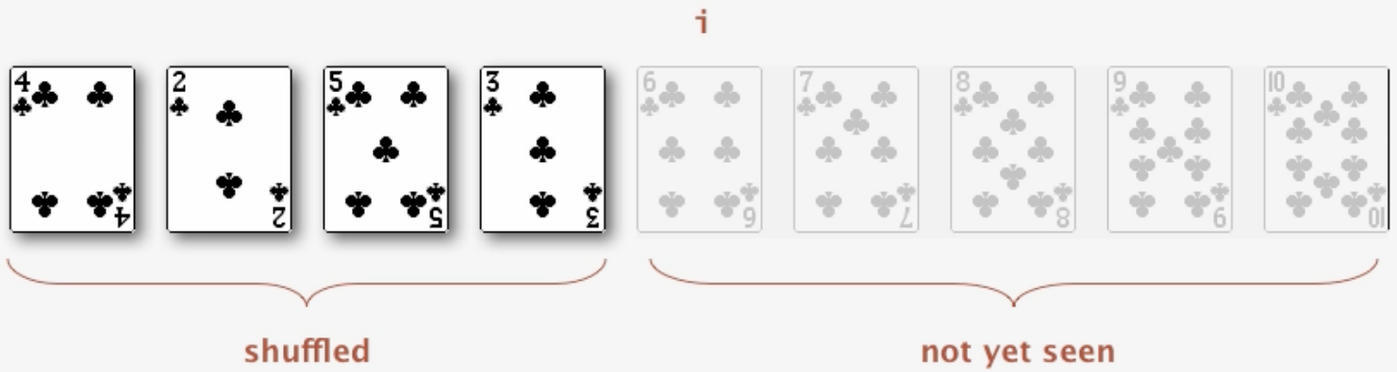
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

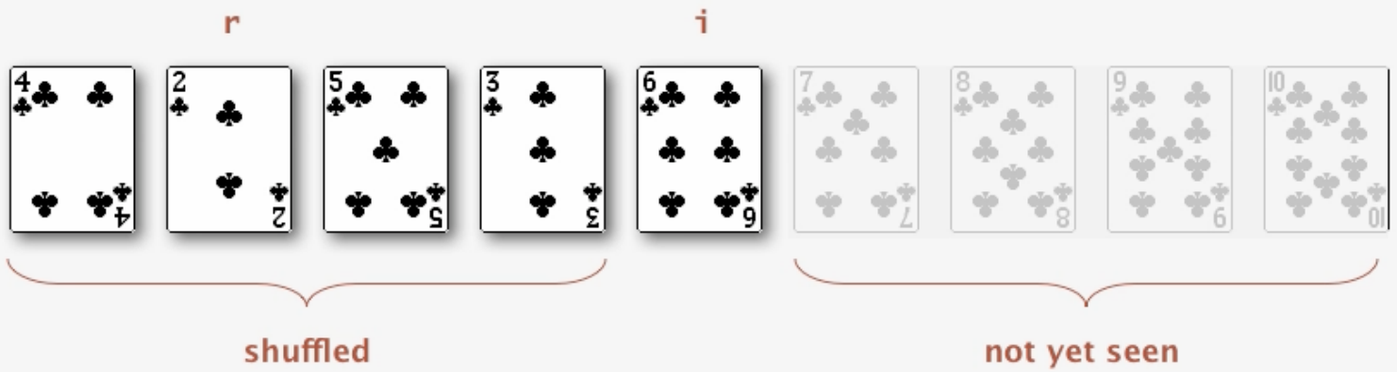
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

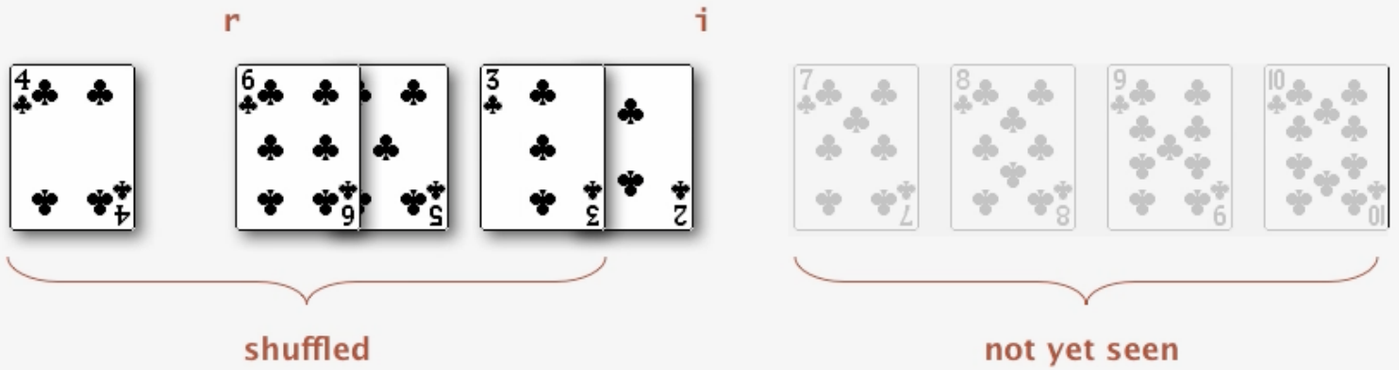
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

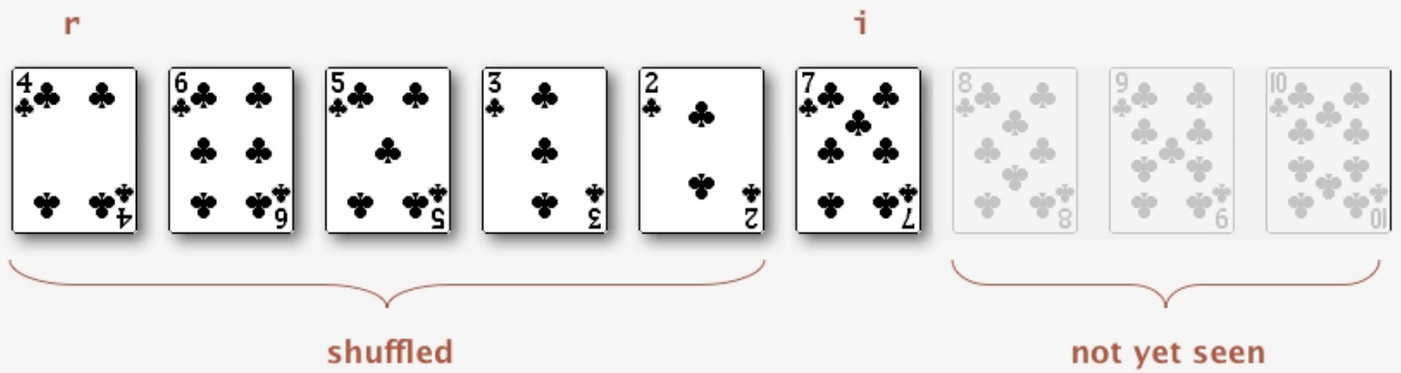
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

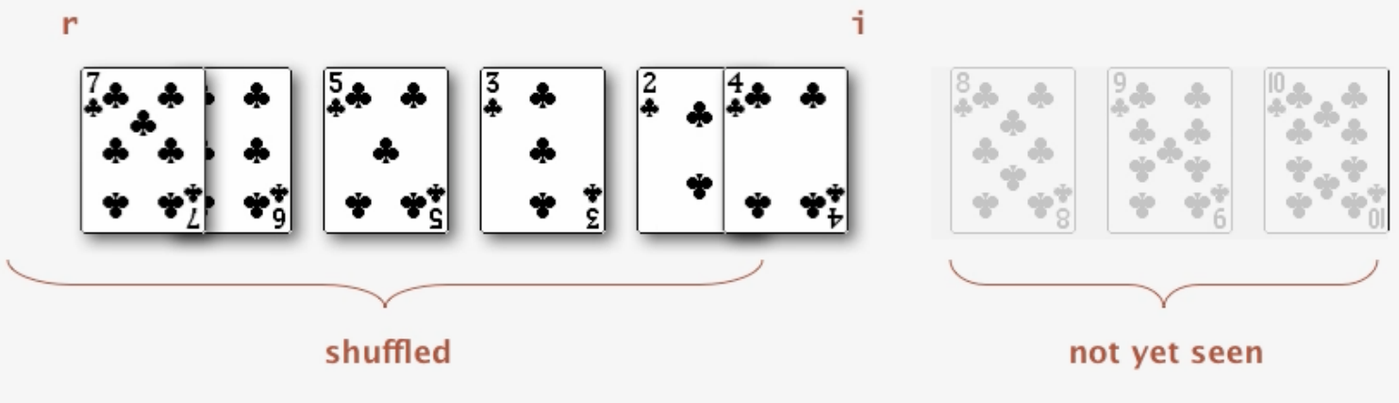
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

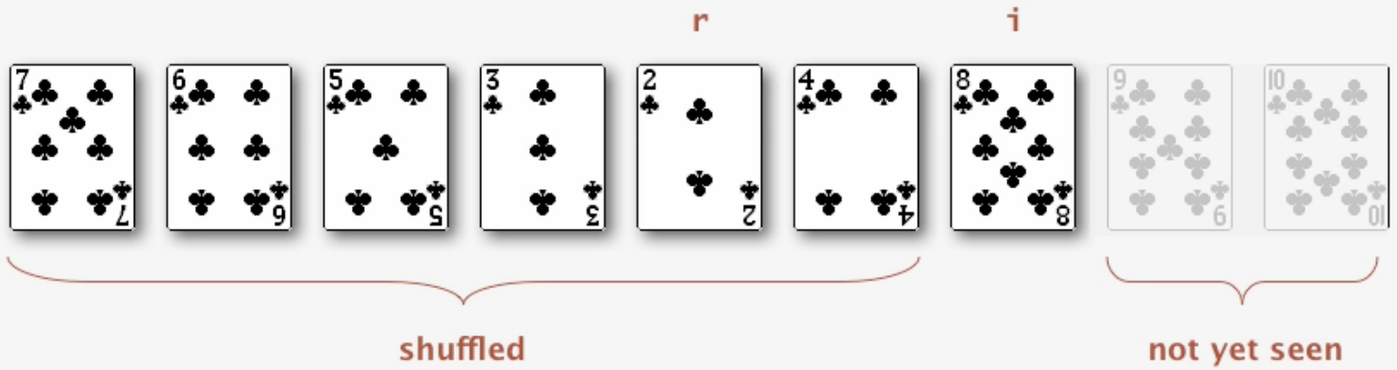
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

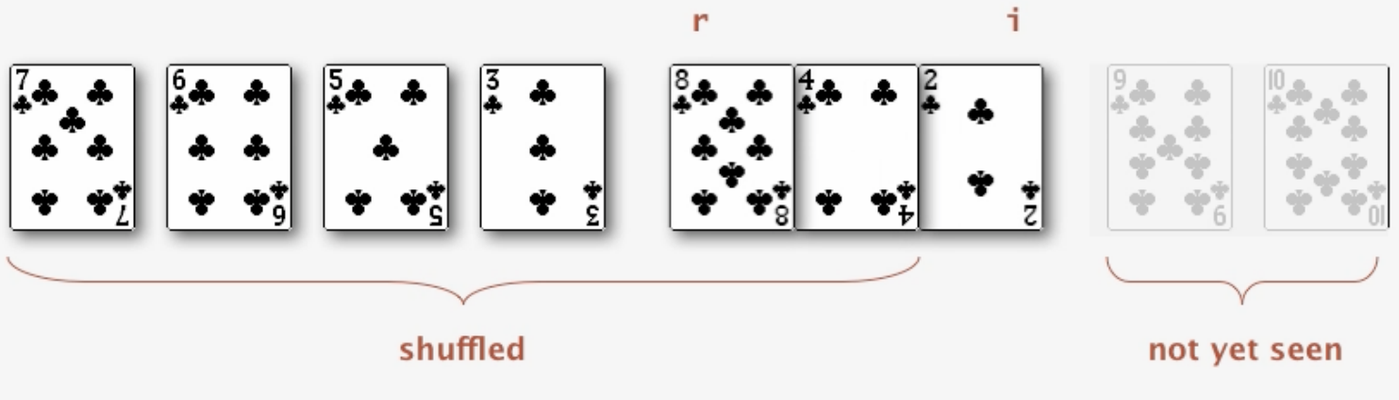
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

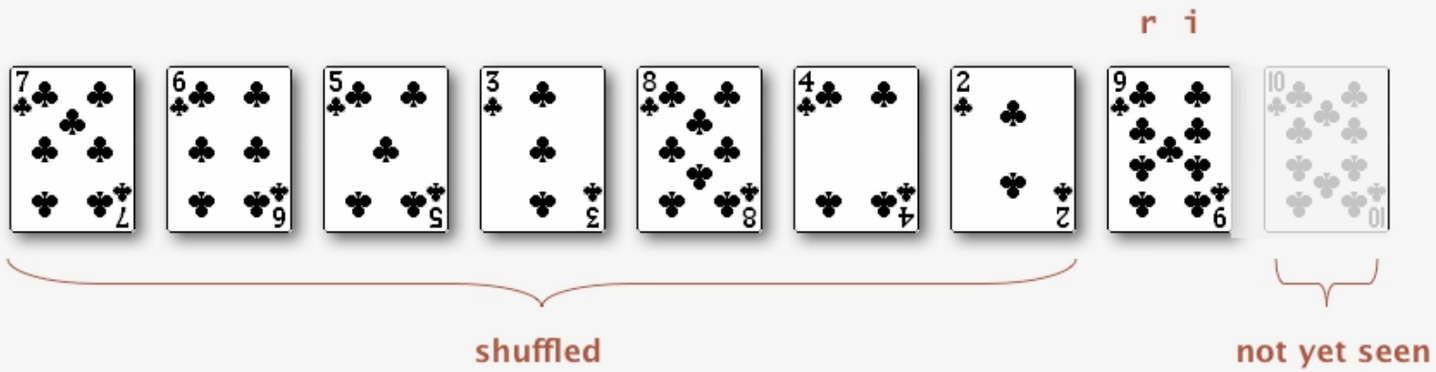
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

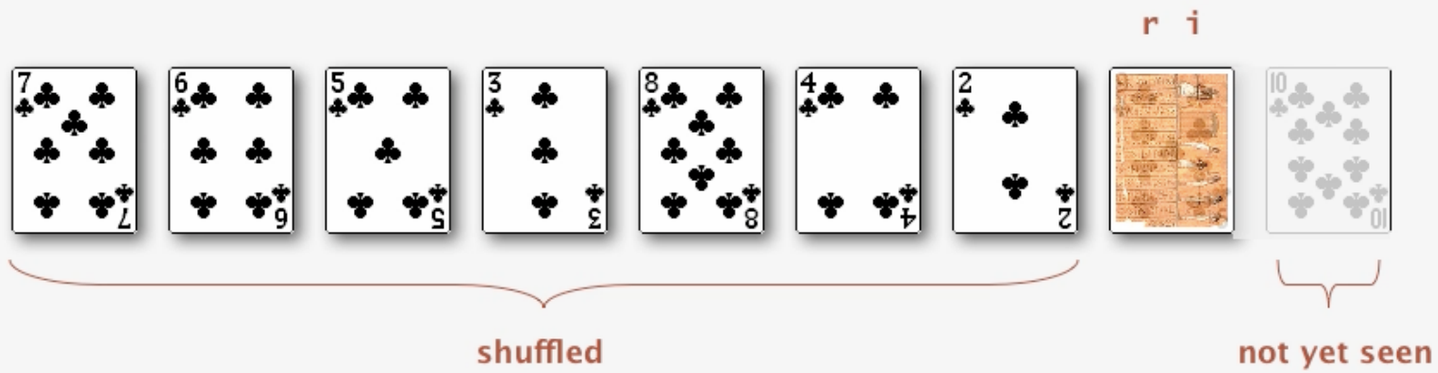
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

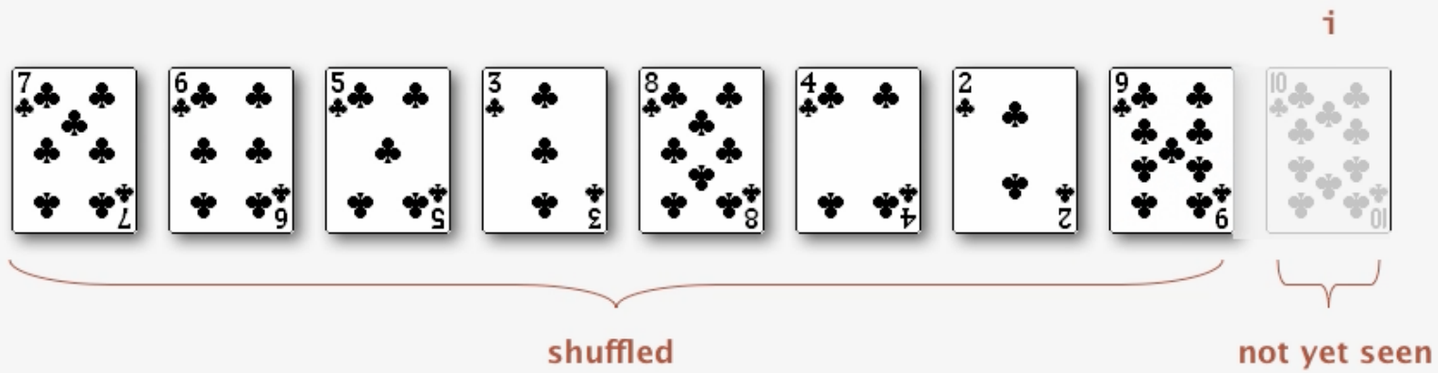
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

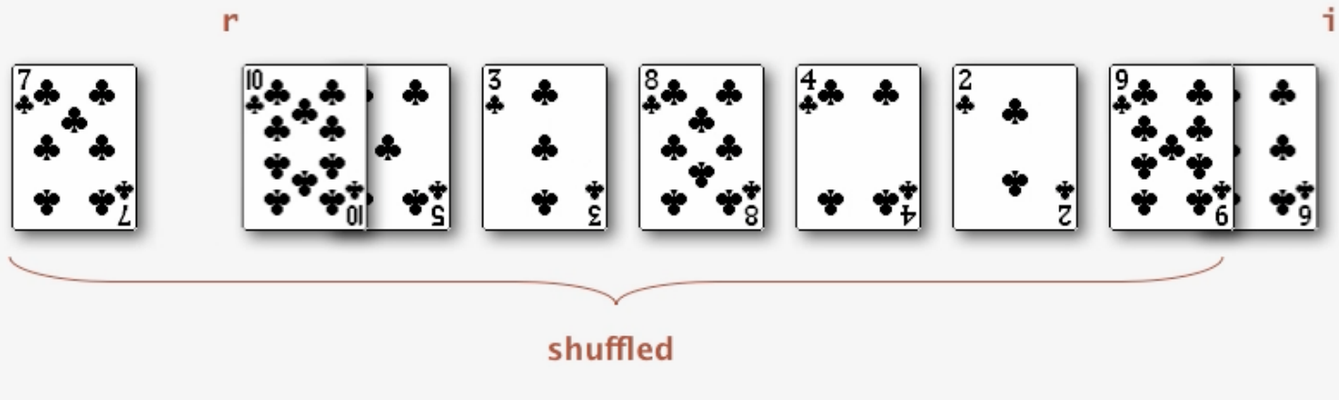
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

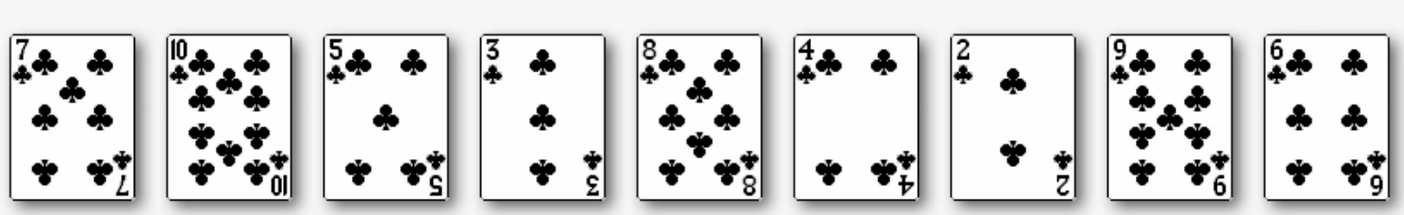
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



Knuth Shuffle Demo

Knuth shuffle

- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



shuffled

Midterm Exam

Midterm next classroom session from 5:45 to 8:00.

You may bring the following to the exam:

- Textbook (open book).
- Laptop
- Class notes

You may ***NOT*** use the internet for any question on the exam. Those in violation will receive a zero for the exam.

Focus

- Autoboxing
- Stacks, Queues, Linked Lists
- Java Control statements (loops)
- Understand the methods you had for homework