SPRING 2023

Information Technology Research

Yi Han

Department of Information Management
National Sun Yat-sen University

Lecture slides are based on the supplemental materials of the textbook: https://algs4.cs.princeton.edu

http://algs4.cs.princeton.edu

2.1 Elementary Sorts

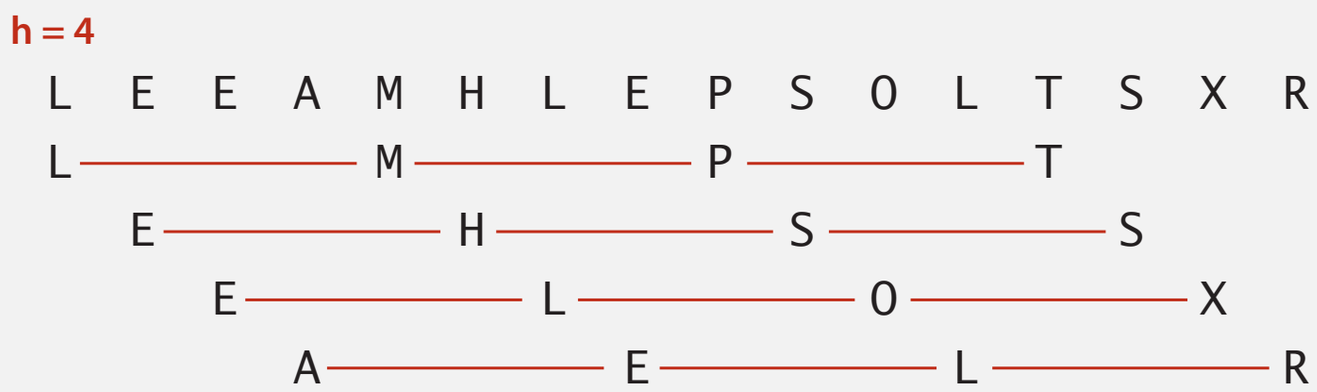
- *rules of the game*
- *selection sort*
- *insertion sort*
- *shellsort*
- *shuffling*

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

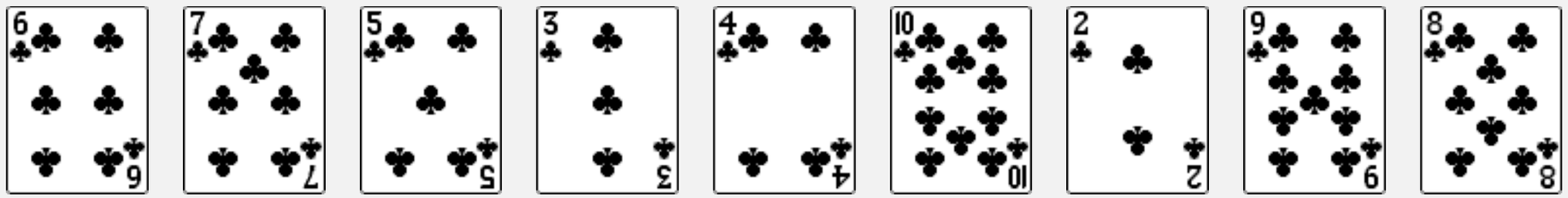


Shellsort. [Shell 1959] h-sort array for decreasing sequence of values of h.

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

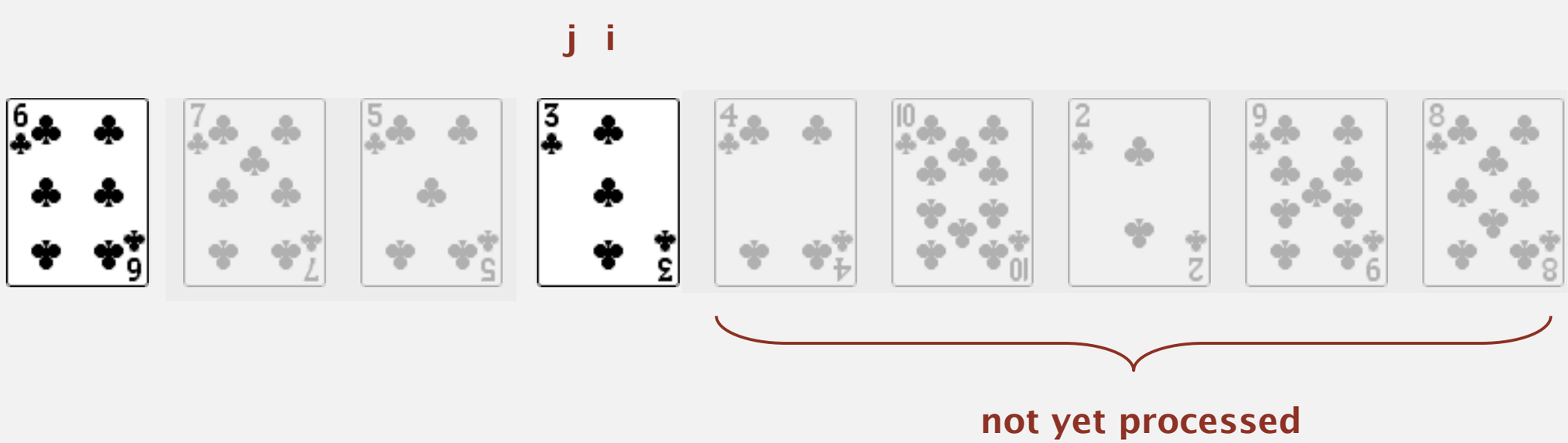
h-sorting demo

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



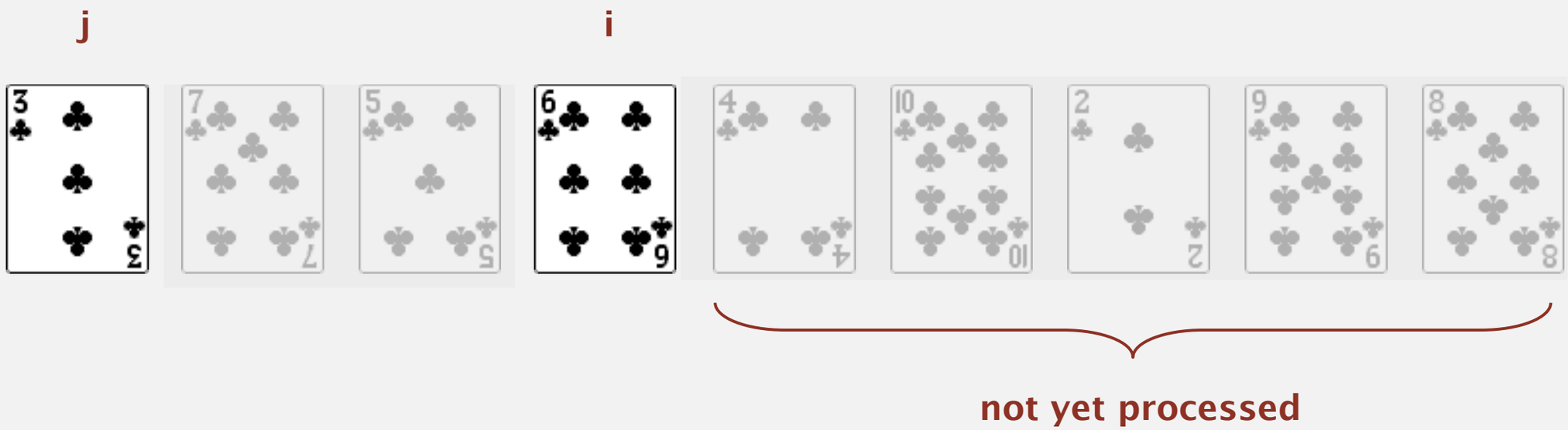
h-sorting demo

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



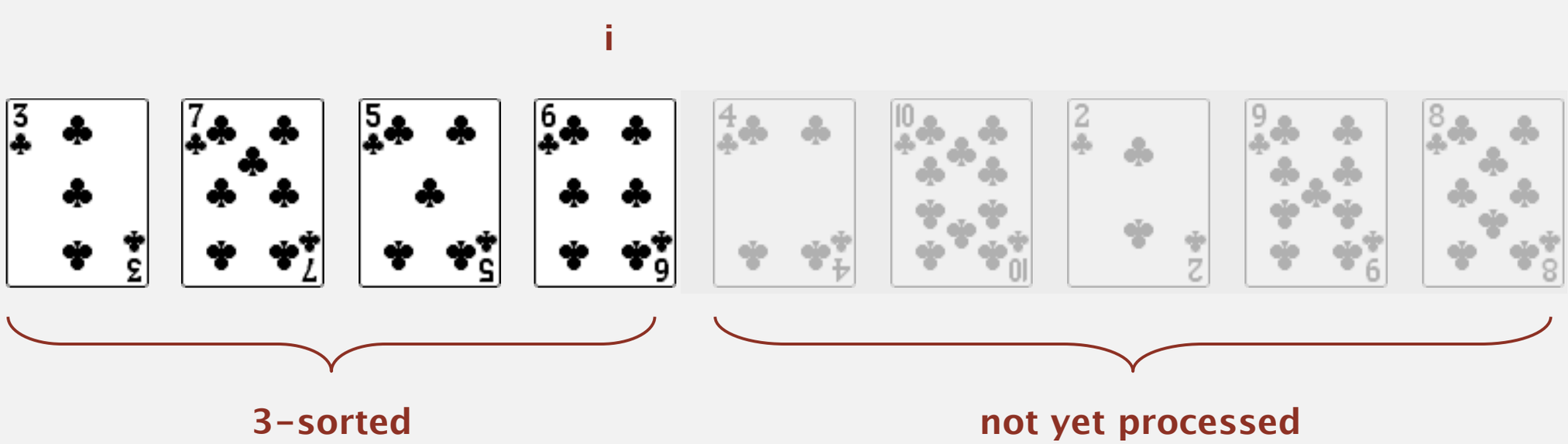
h-sorting demo

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



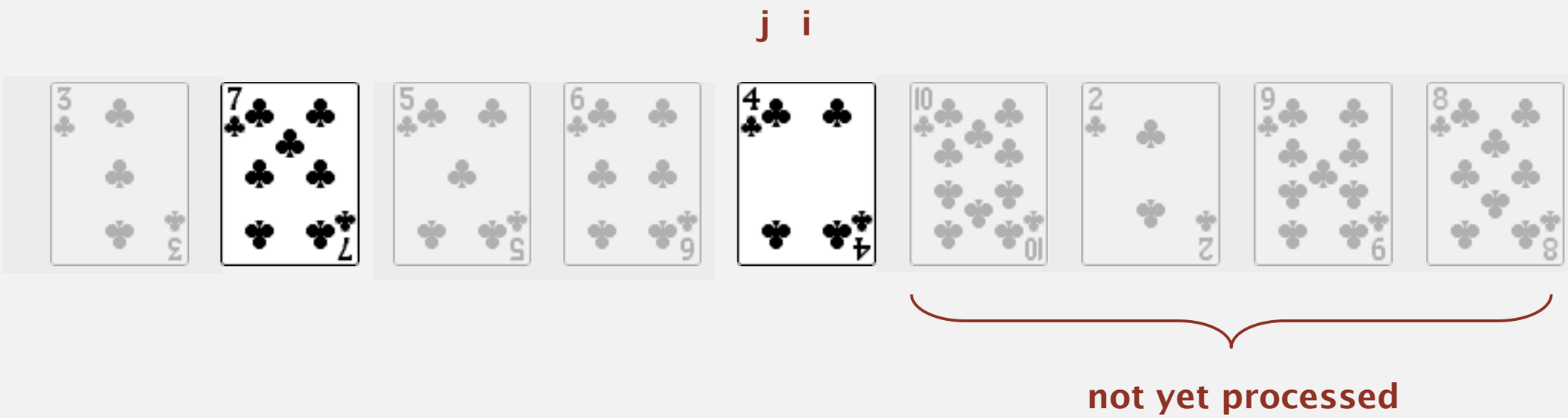
h-sorting demo

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



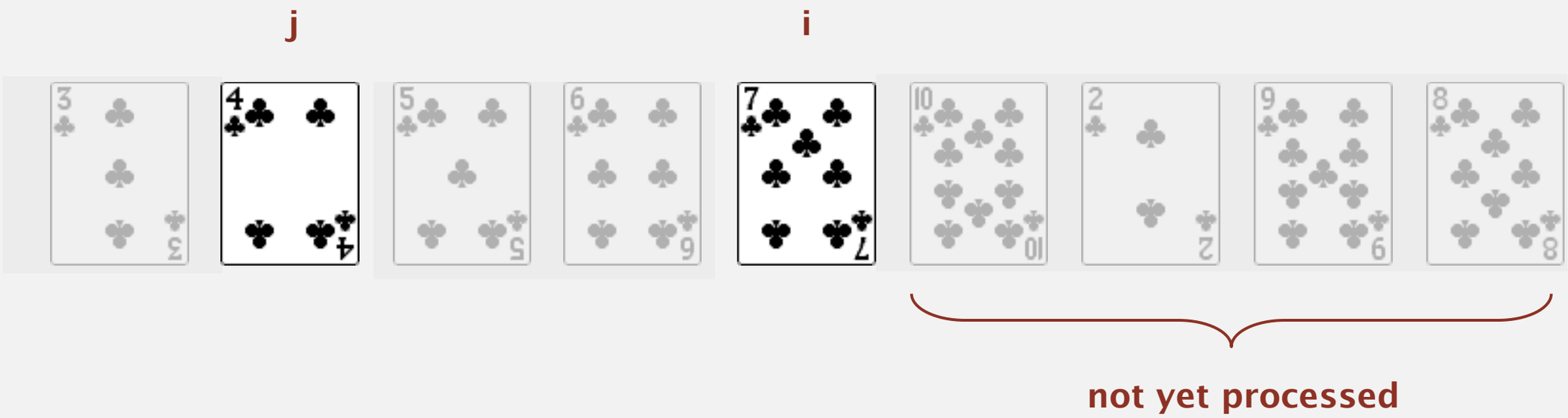
h-sorting demo

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



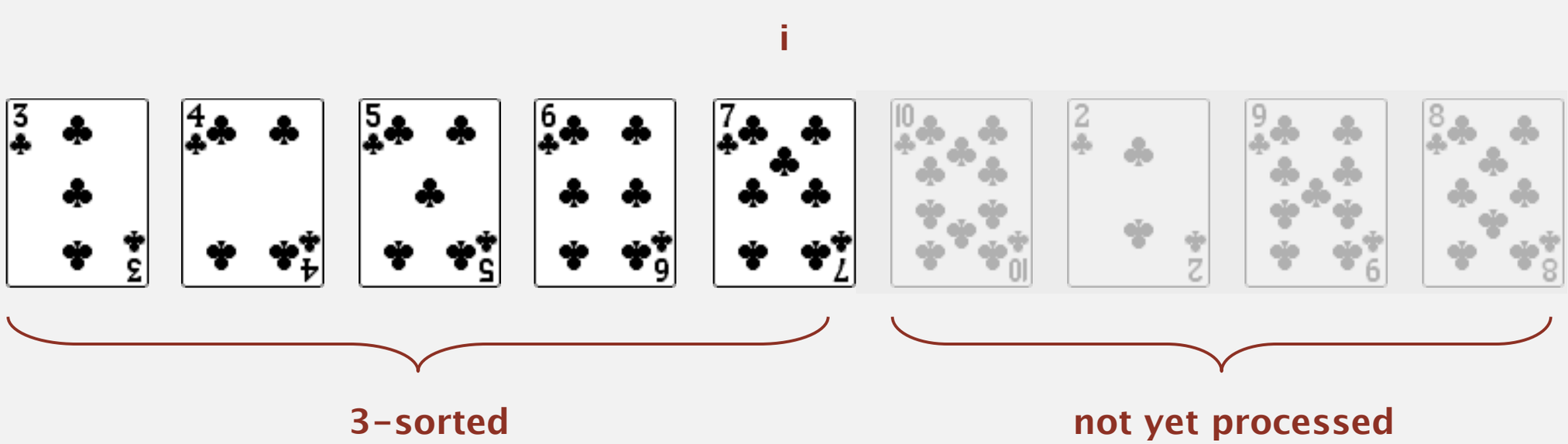
h-sorting demo

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



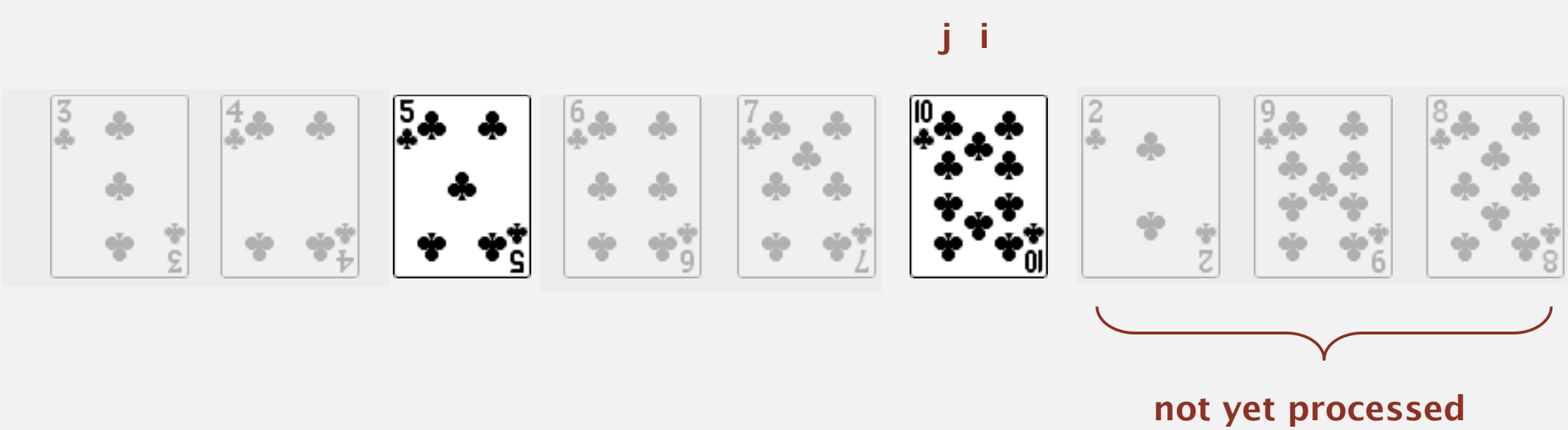
h-sorting demo

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



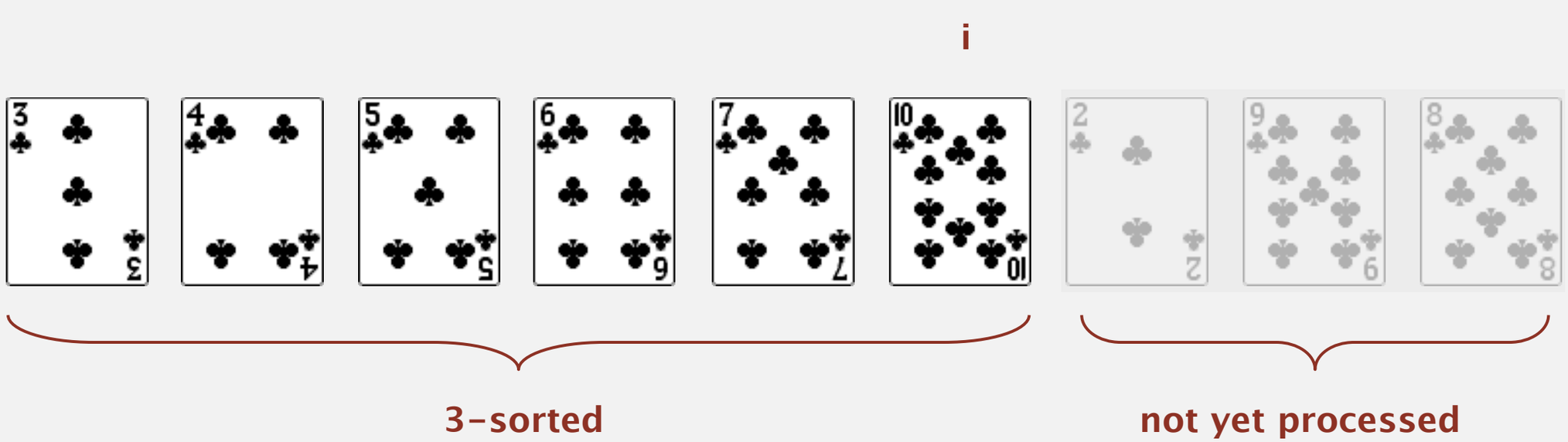
h-sorting demo

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



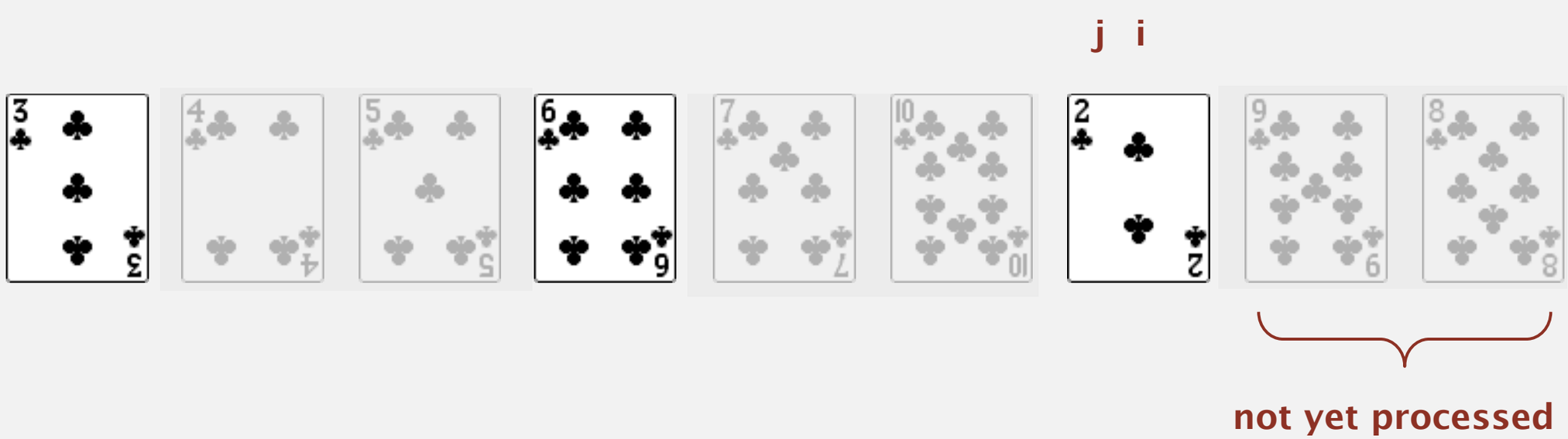
h-sorting demo

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



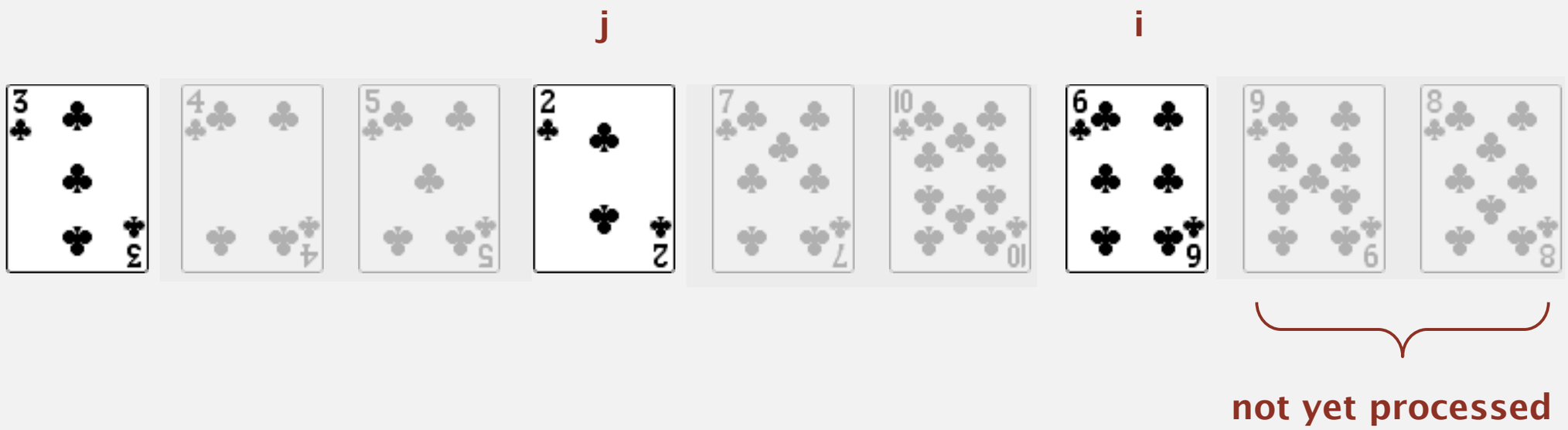
h-sorting demo

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



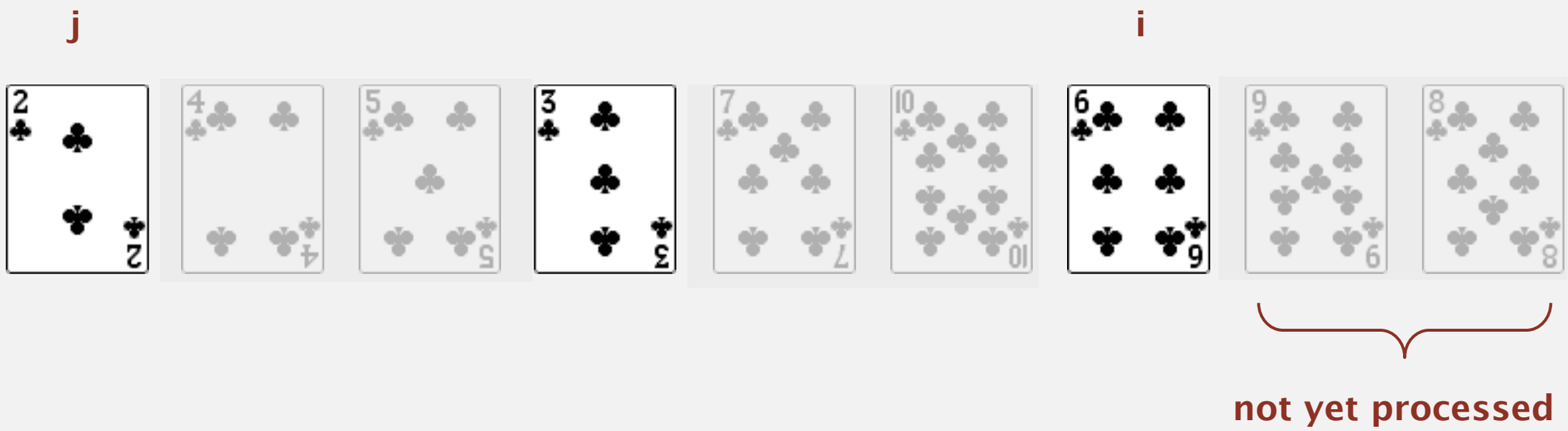
h-sorting demo

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



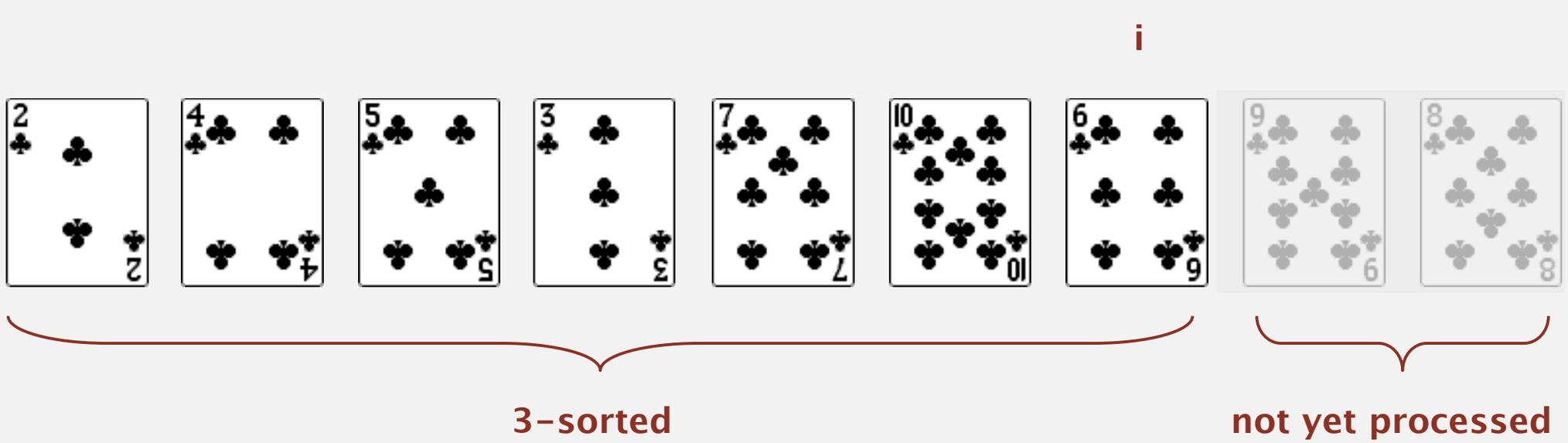
h-sorting demo

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



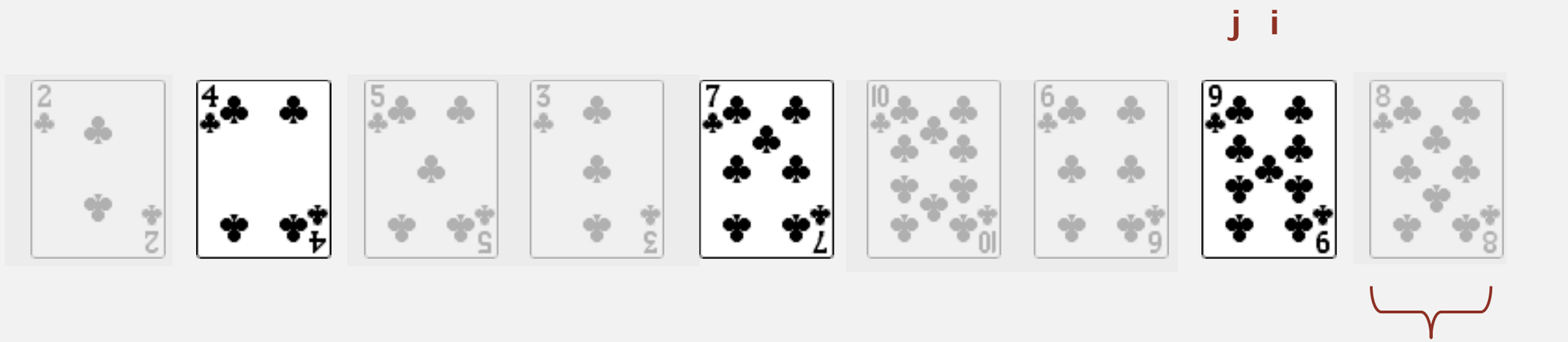
h-sorting demo

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



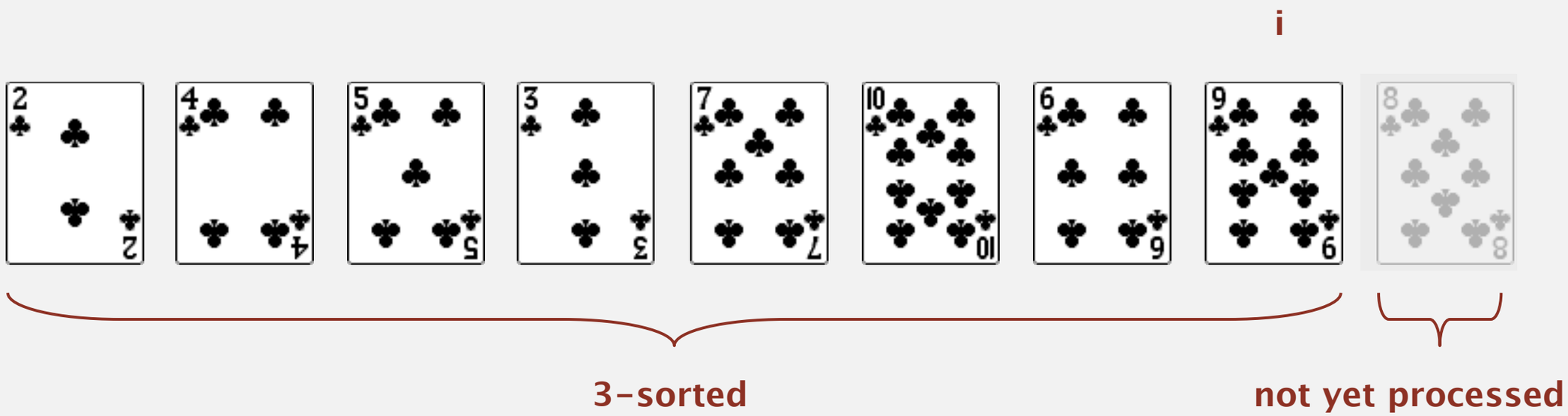
h-sorting demo

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



h-sorting demo

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



h-sorting demo

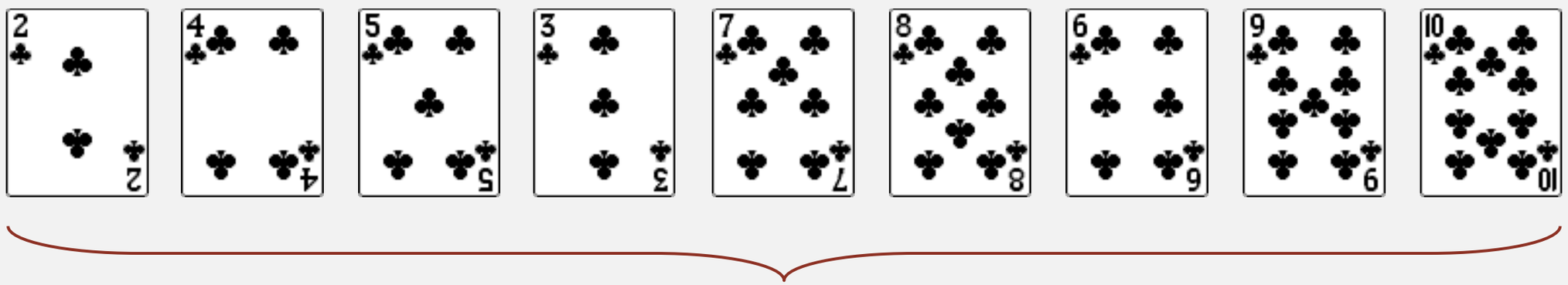
In iteration `i`, swap `a[i]` with each larger entry `h` positions to its left.

h-sorting demo

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

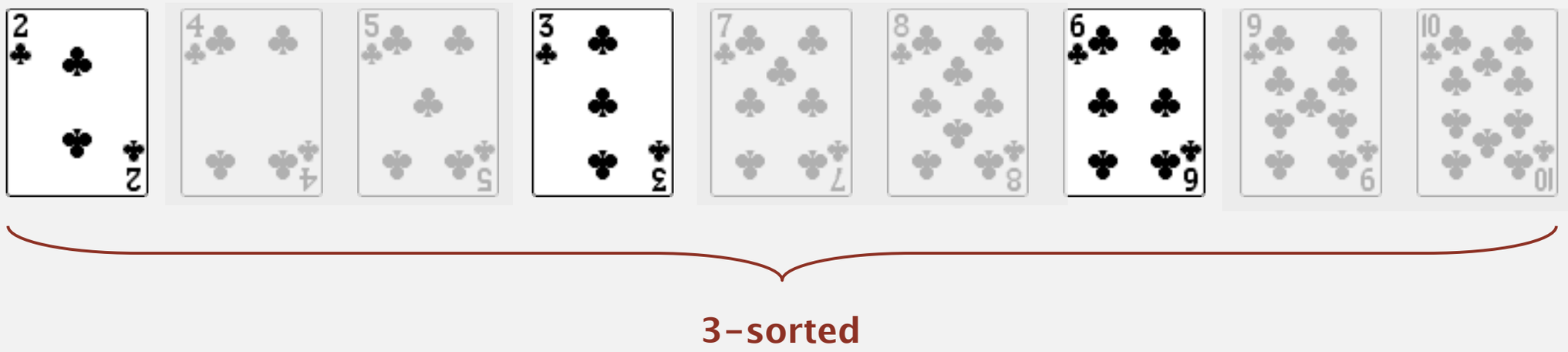
h-sorting demo

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



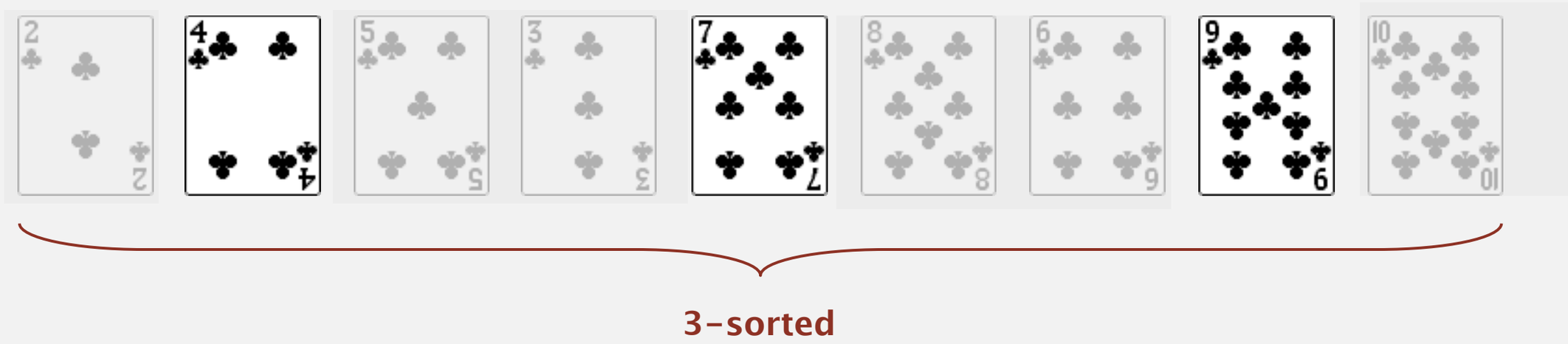
h-sorting demo

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



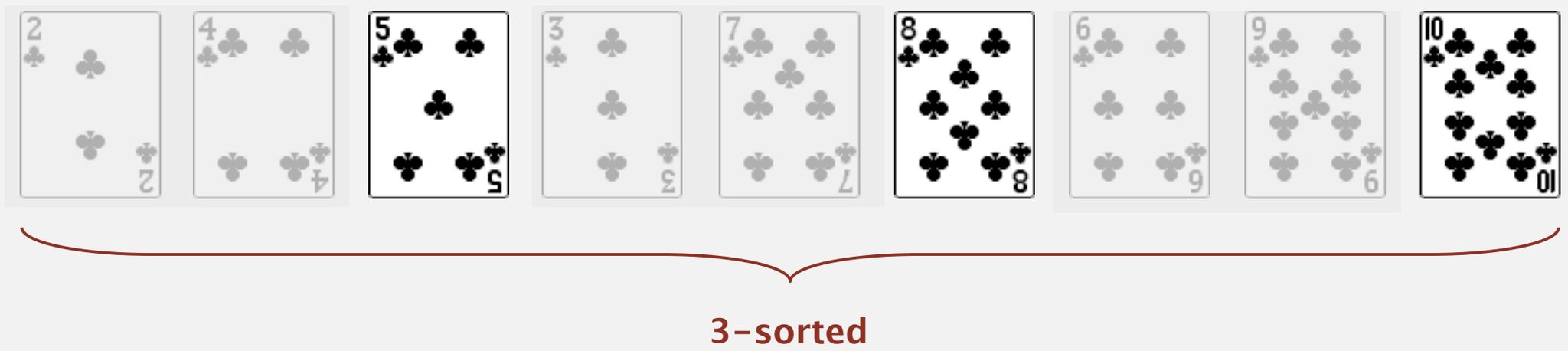
h-sorting demo

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



h-sorting demo

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



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]

h-sort: how would you implement it from 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 */  }
}
```

What to change here?
For h-sorted.
4 mins.

h-sort: how would you implement it from insertion sort?

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

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

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

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 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: 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, ...

      What to write here? 5 mins.
      Can use exch and less methods.

   }

   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

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, ...

      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 */ }
}
```

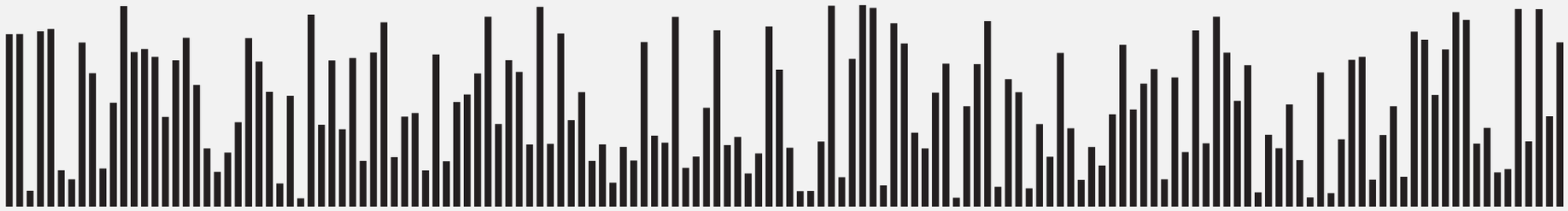
3x+1 increment sequence

insertion sort

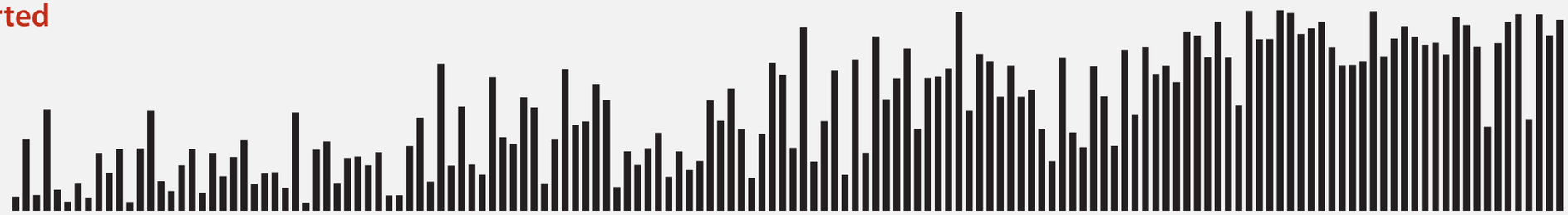
move to next increment

Shellsort: visual trace

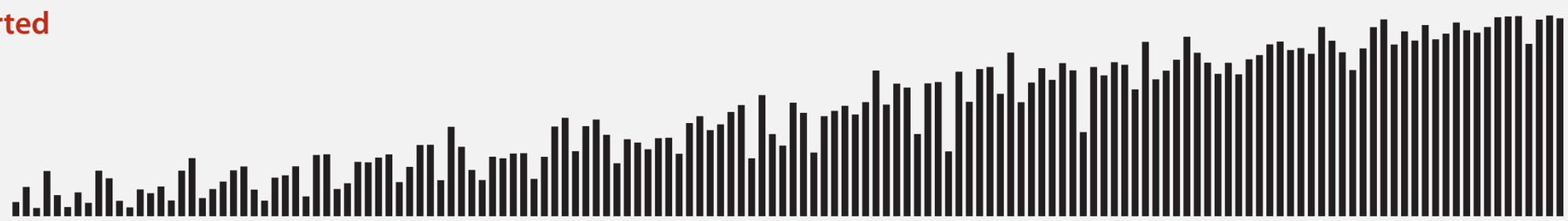
input



40-sorted



13-sorted

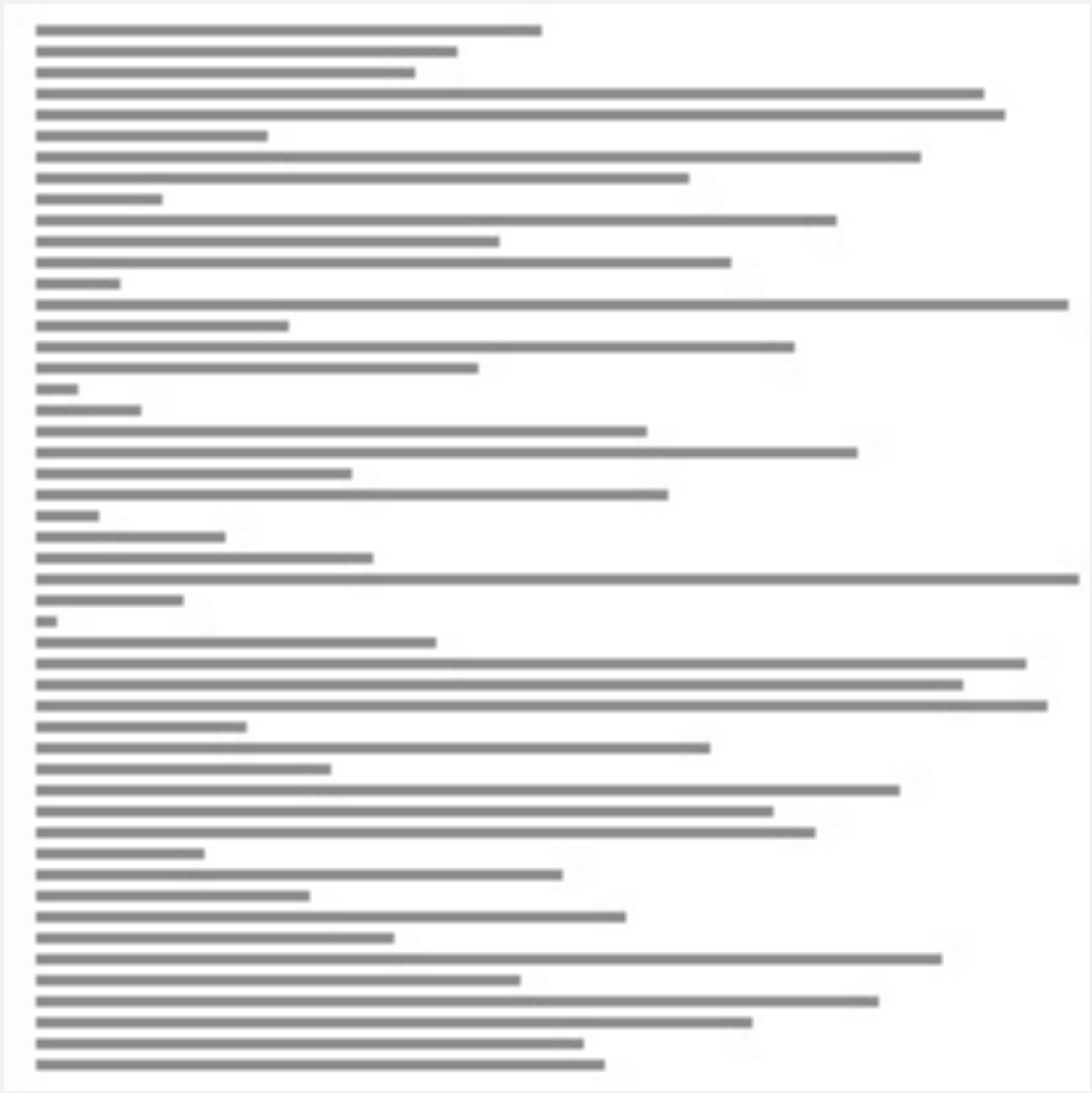


4-sorted

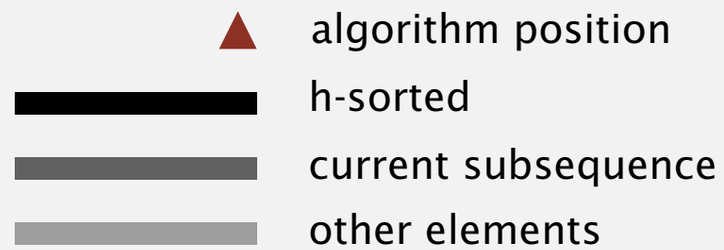
result

Shellsort: animation

50 random items

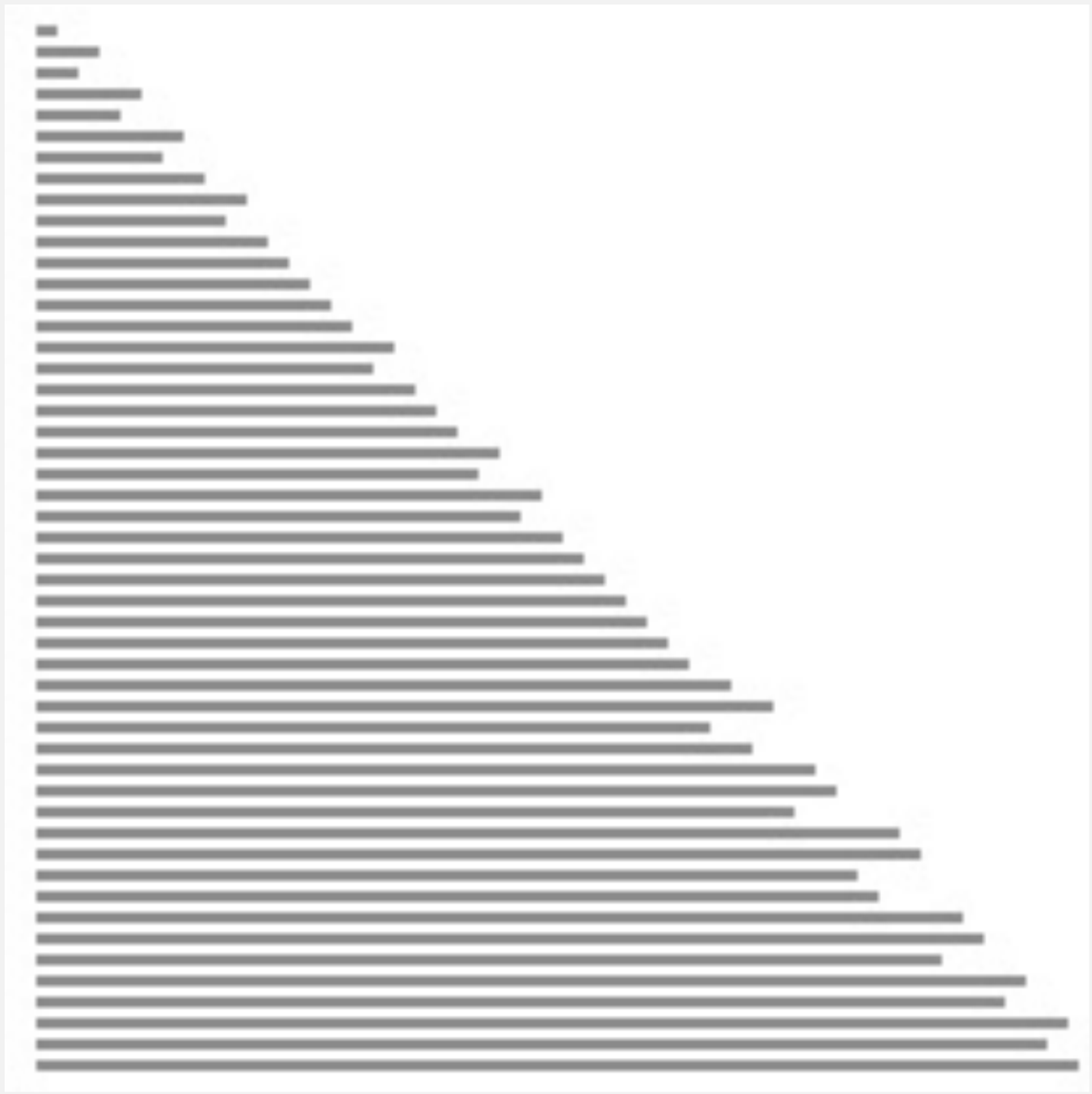


http://www.sorting-algorithms.com/shell-sort

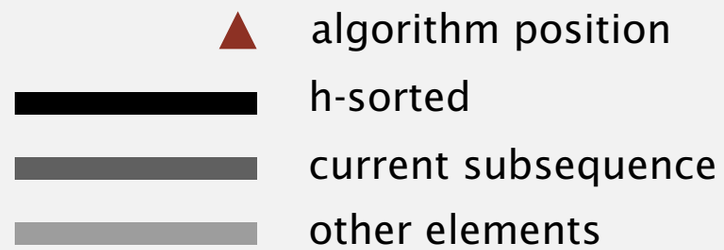


Shellsort: animation

50 partially-sorted items



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



Shellsort: analysis

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 (!)

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.

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

http://algs4.cs.princeton.edu

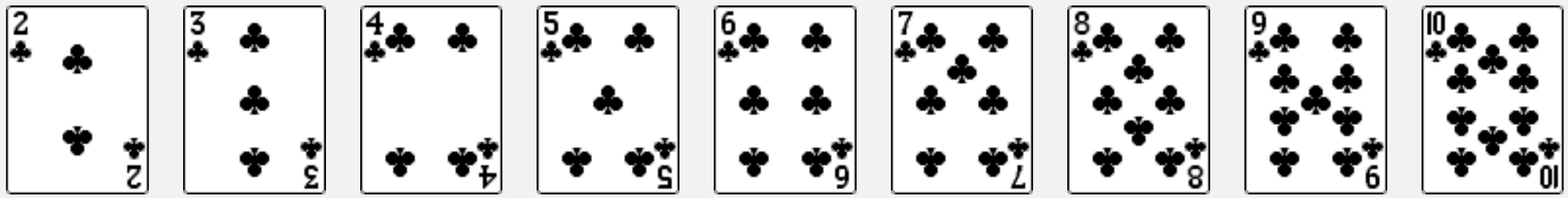
2.1 Elementary Sorts

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

How to shuffle an array

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

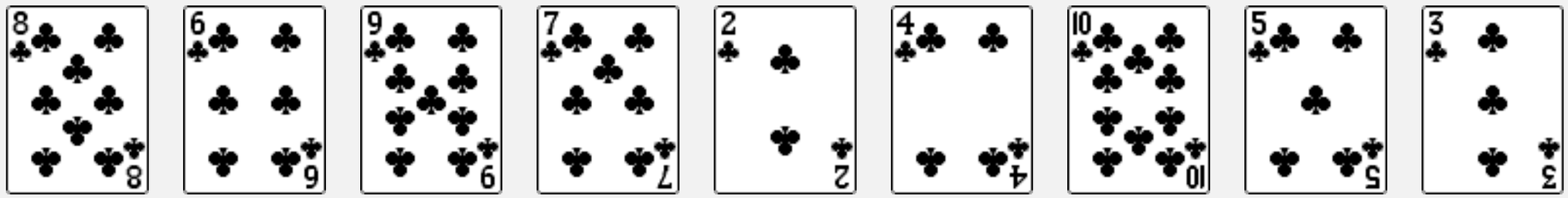
all permutations
equally likely



How to shuffle an array

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

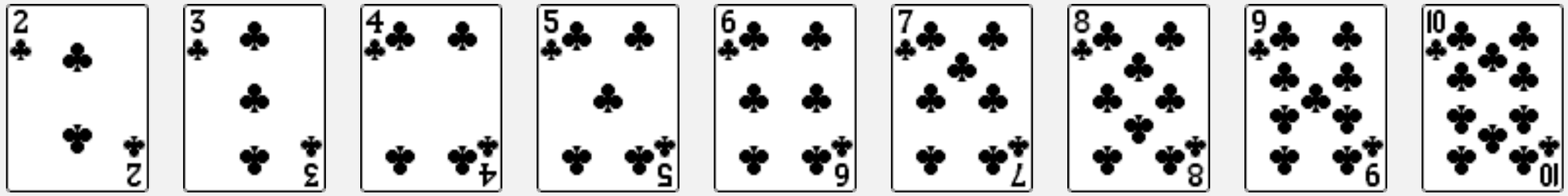
all permutations
equally likely



Shuffle sort

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

useful for shuffling columns in a spreadsheet

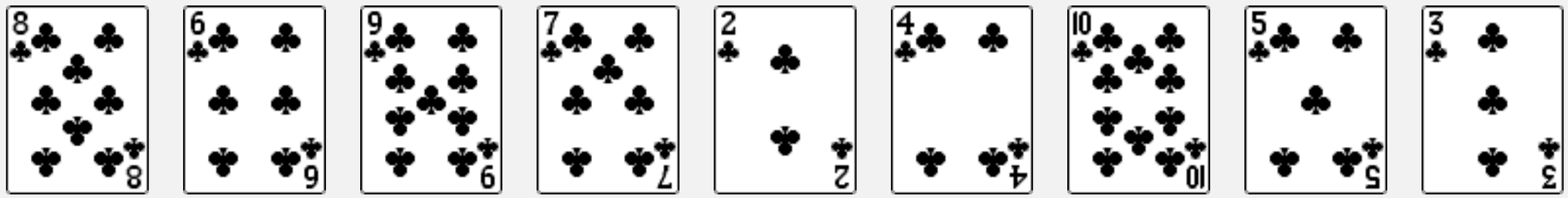


0.8003 0.9706 0.9157 0.9649 0.1576 0.4854 0.1419 0.4218 0.9572

Shuffle sort

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

useful for shuffling columns in a spreadsheet

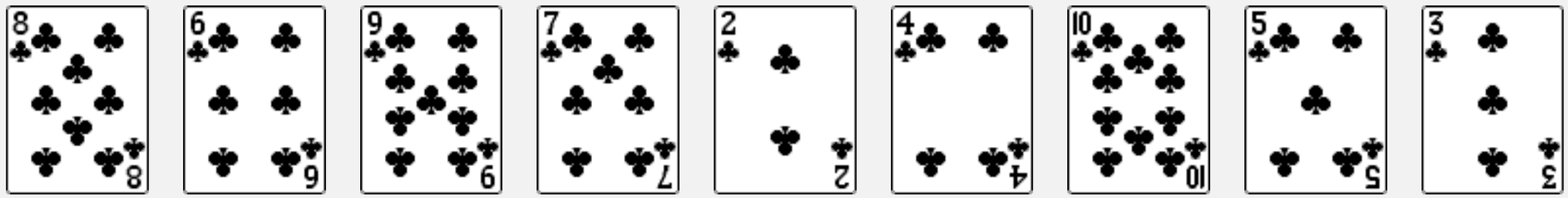


0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Shuffle sort

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

useful for shuffling columns in a spreadsheet



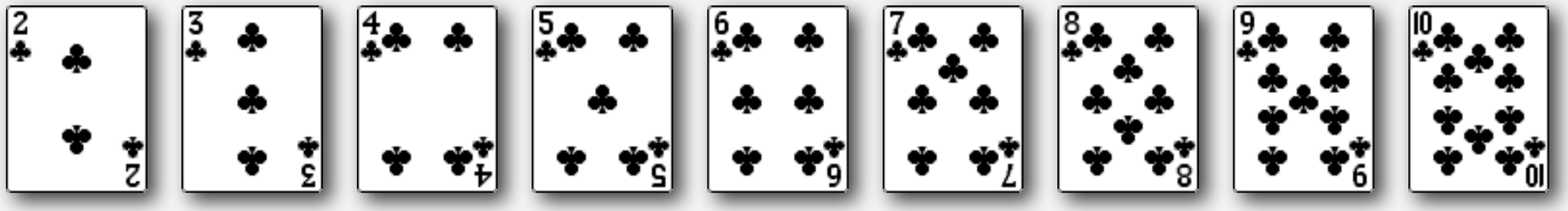
0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Proposition. Shuffle sort produces a uniformly random permutation.

assuming real numbers uniformly at random (and no ties)

Knuth shuffle

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



Knuth shuffle

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

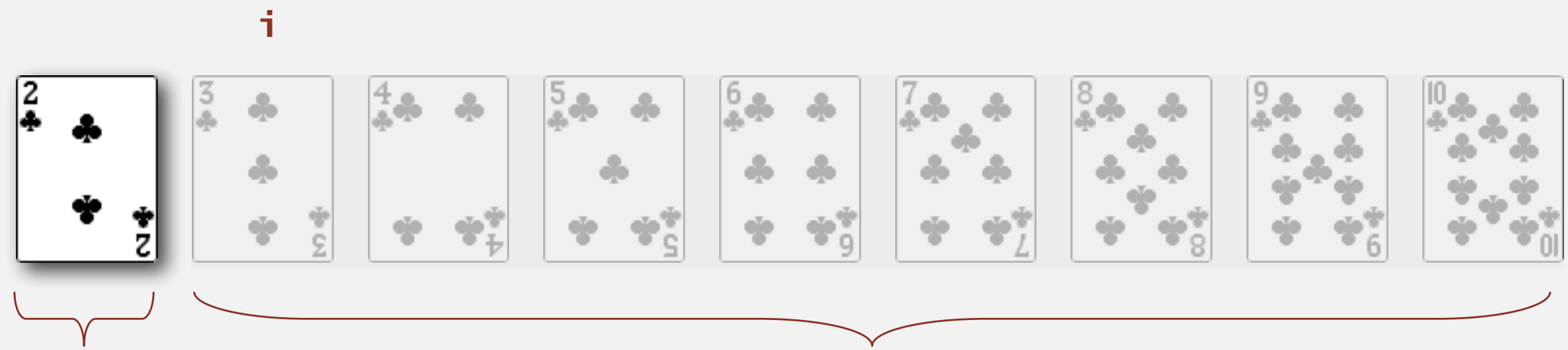
Knuth shuffle

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

not yet seen

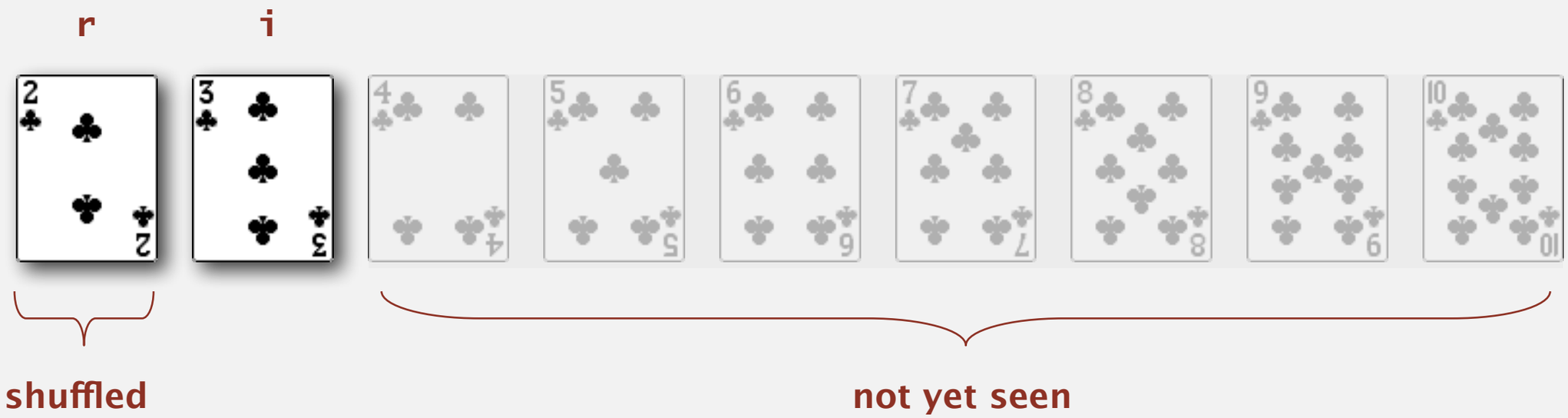
Knuth shuffle

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



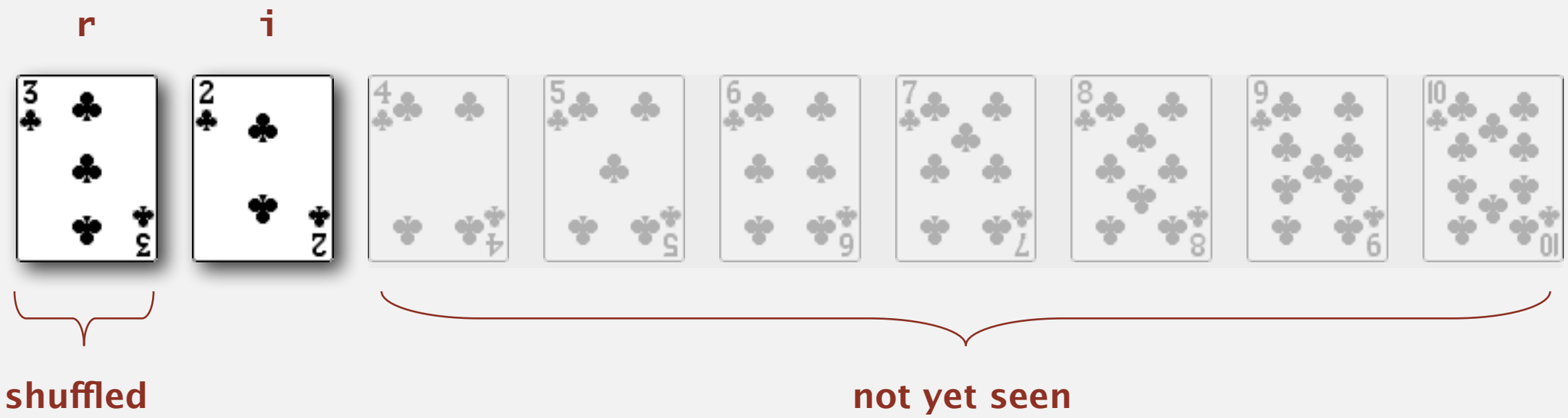
Knuth shuffle

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



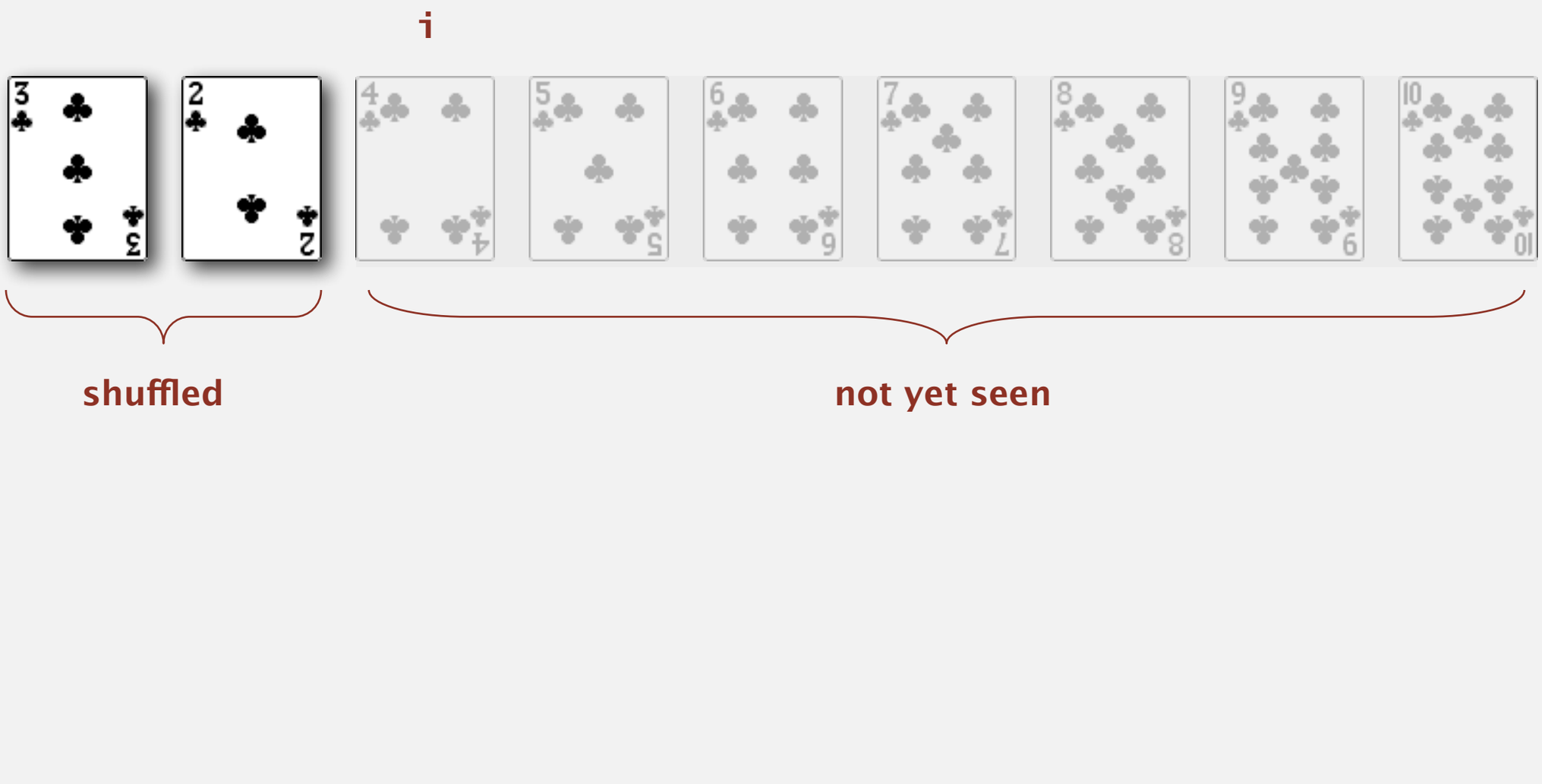
Knuth shuffle

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



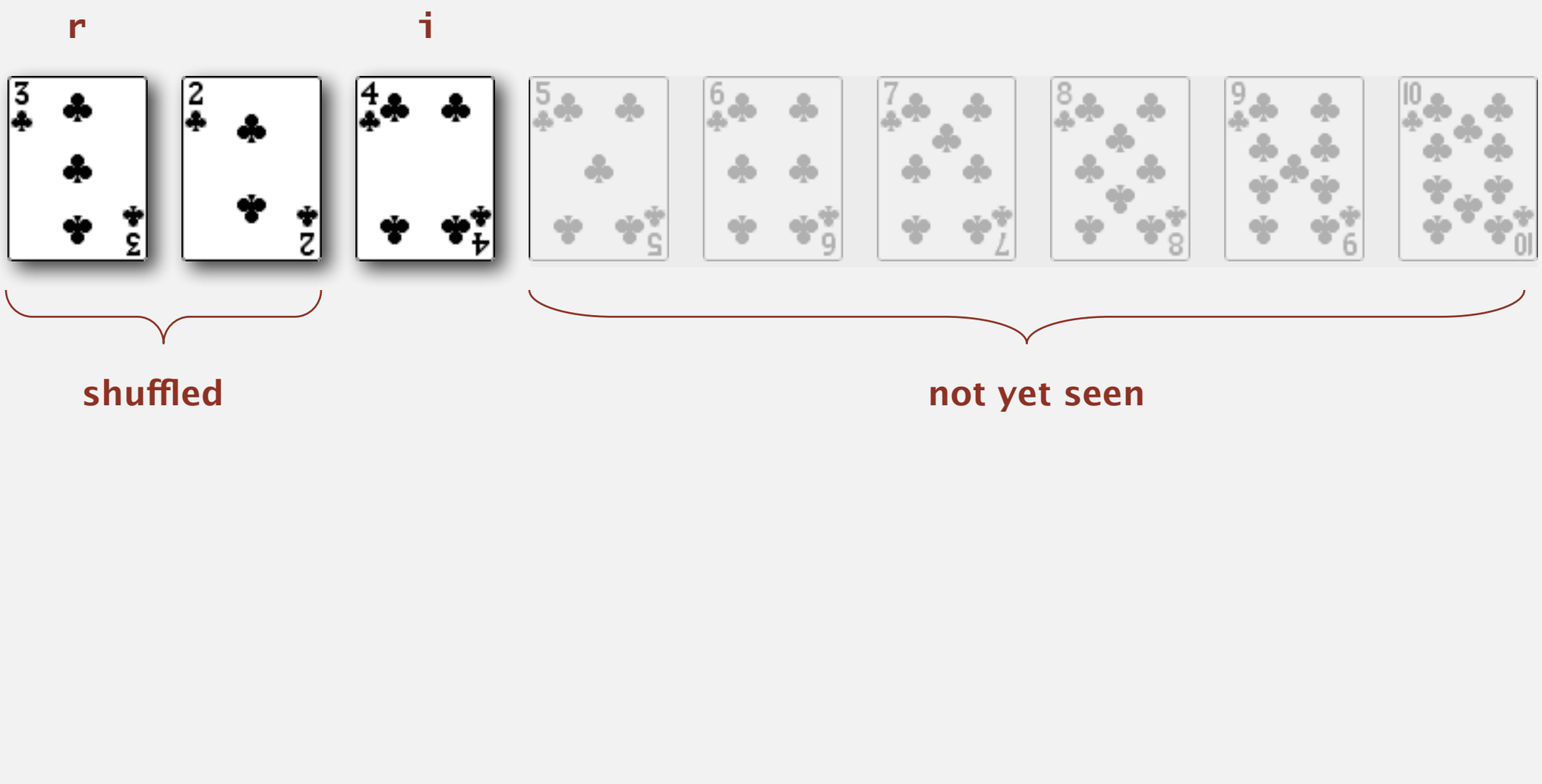
Knuth shuffle

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



Knuth shuffle

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



Knuth shuffle

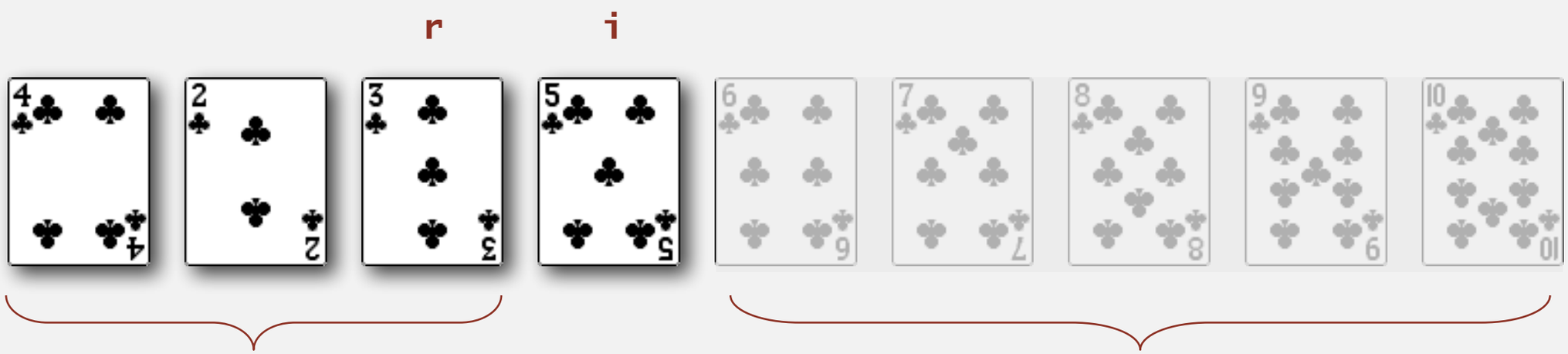
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.

Knuth shuffle

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

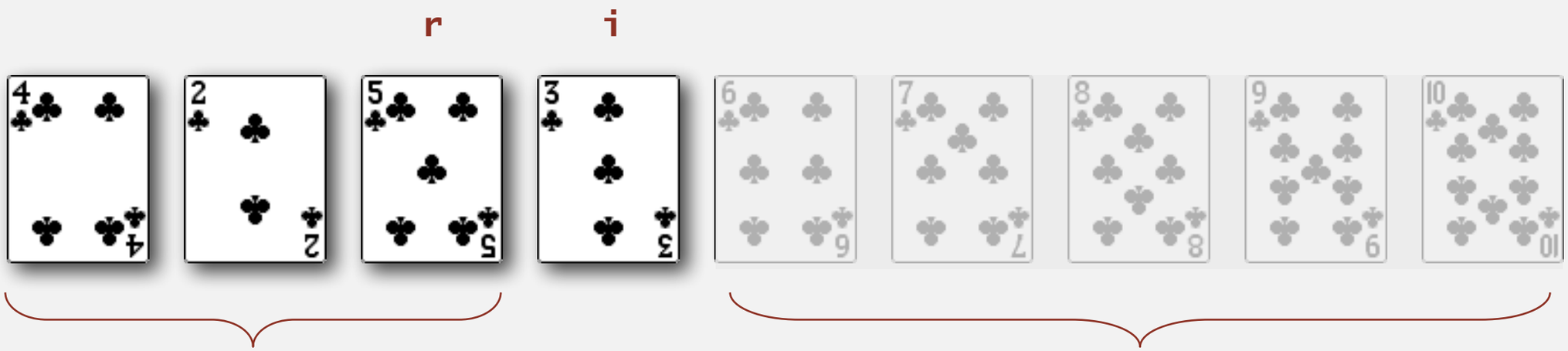
Knuth shuffle

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



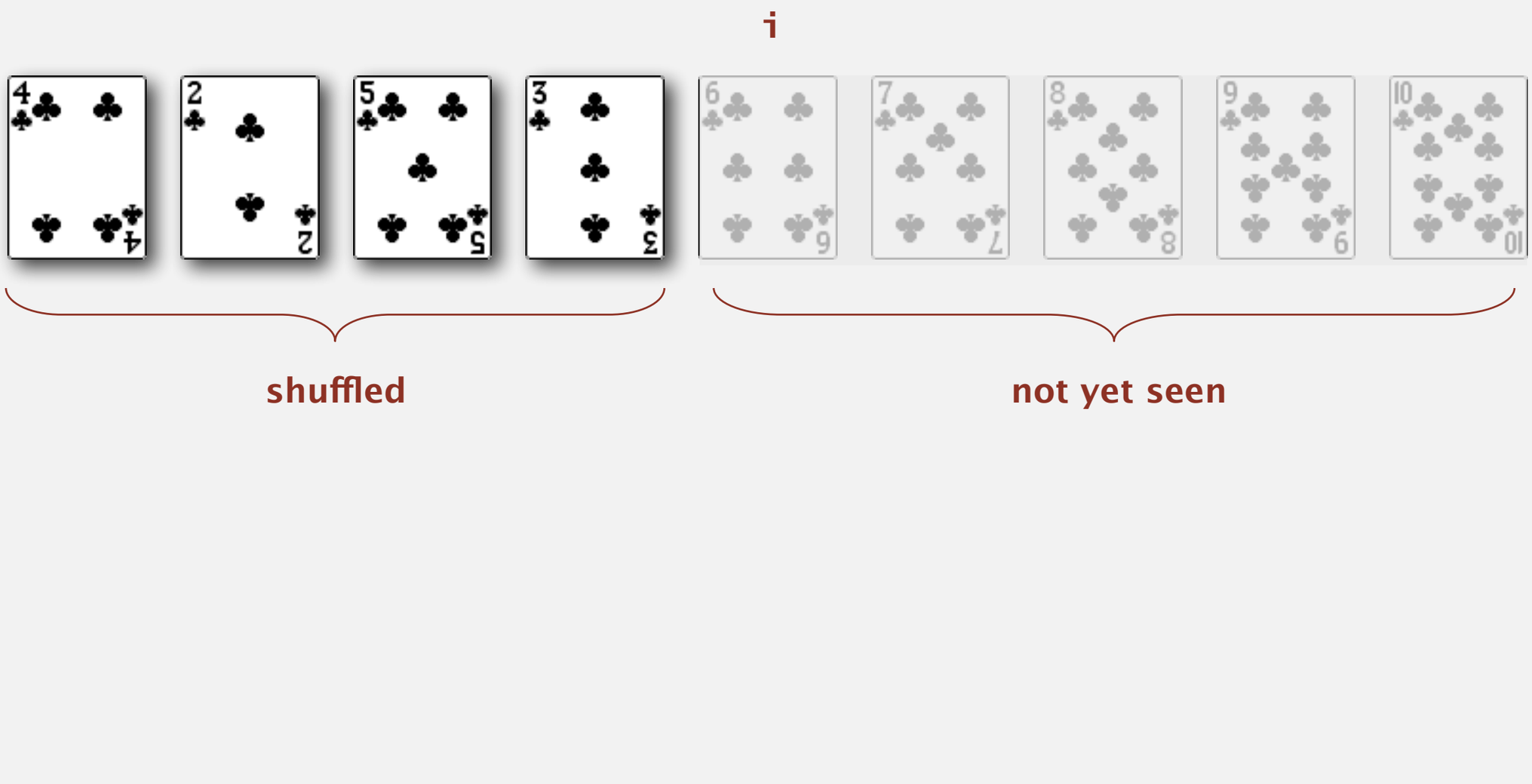
Knuth shuffle

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



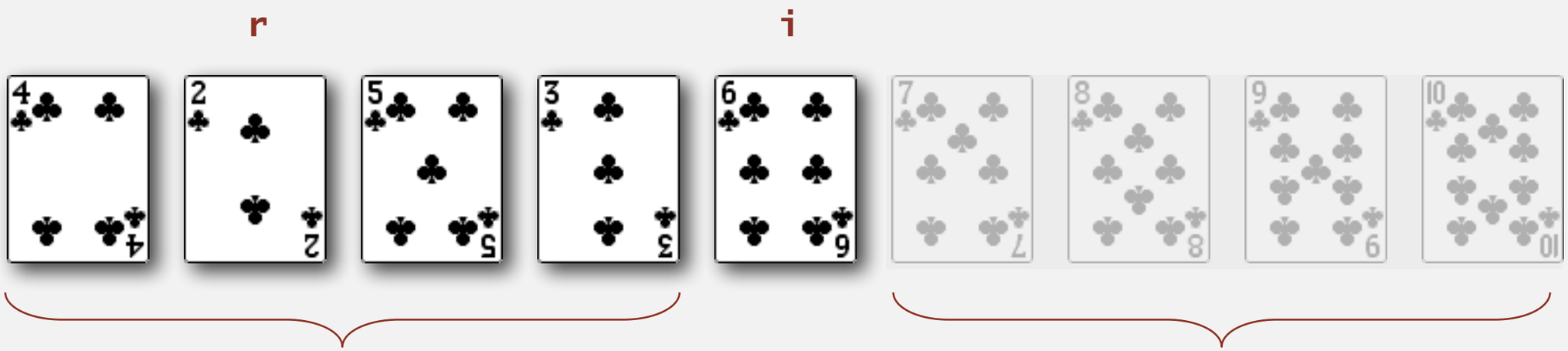
Knuth shuffle

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



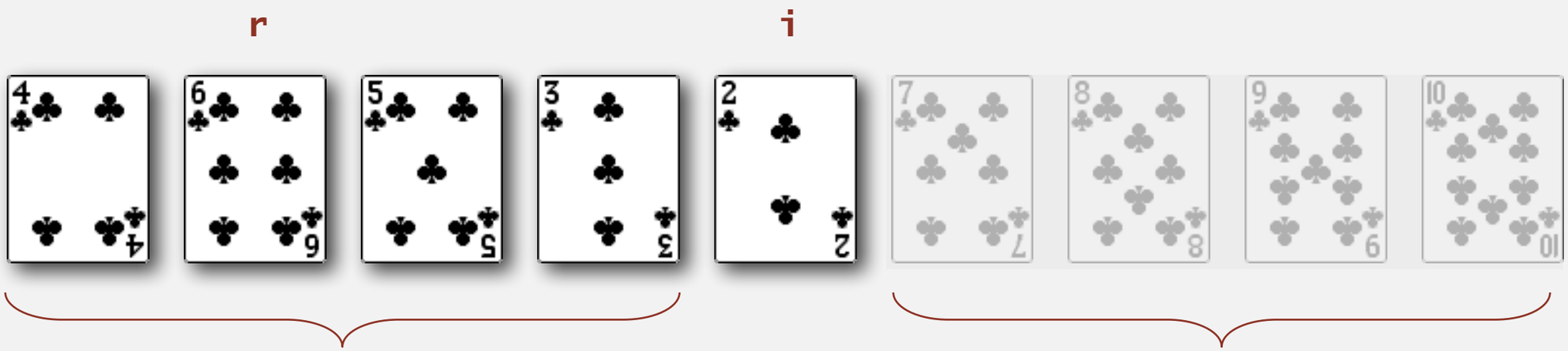
Knuth shuffle

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



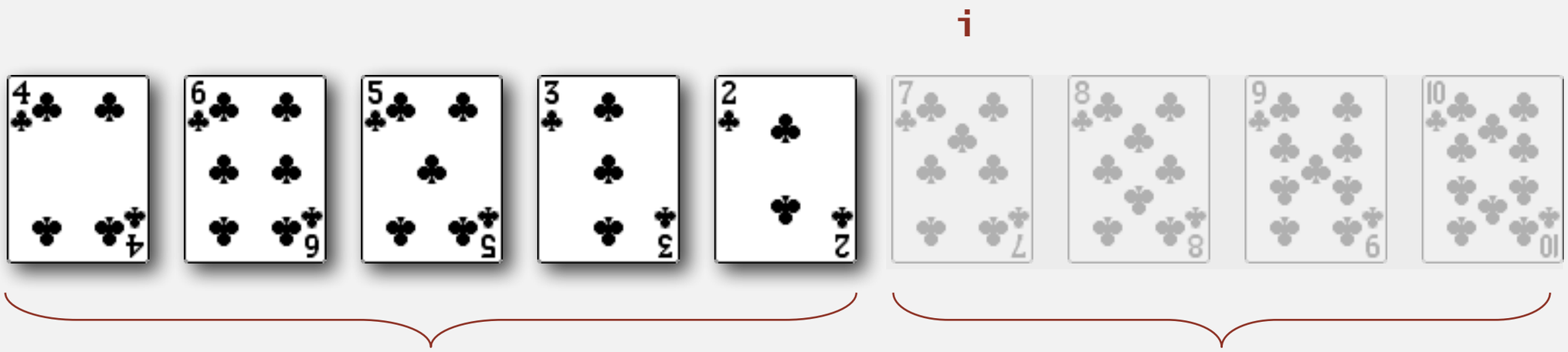
Knuth shuffle

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



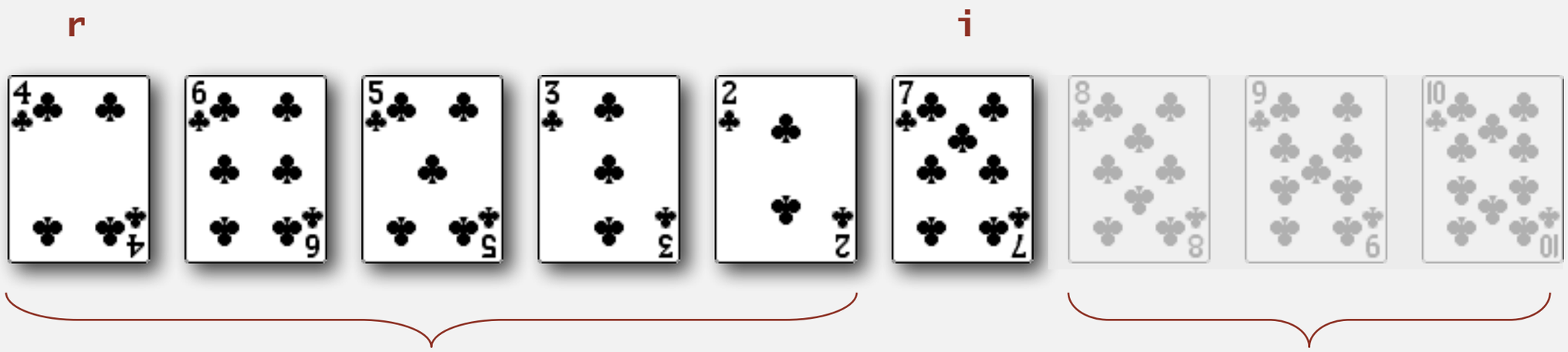
Knuth shuffle

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



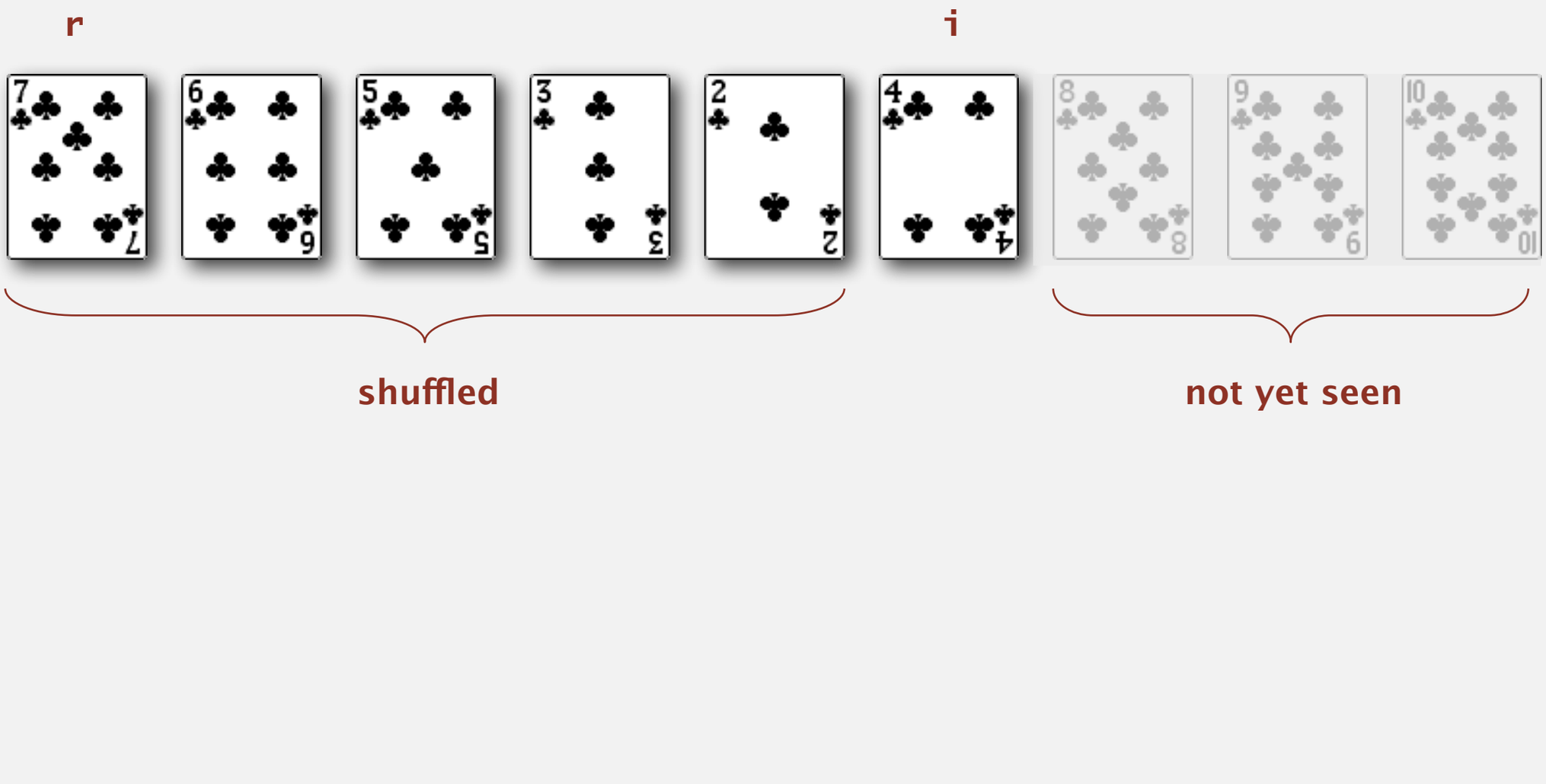
Knuth shuffle

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



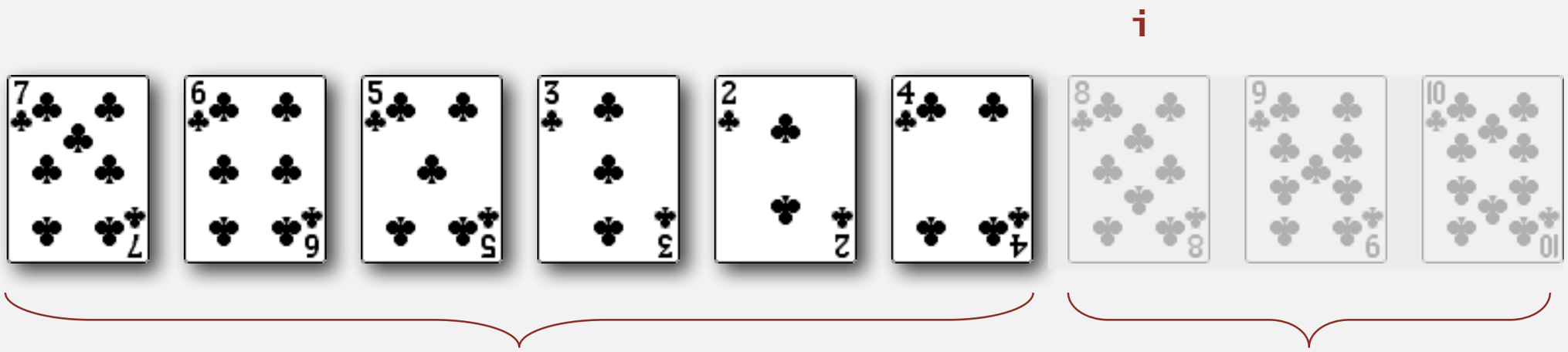
Knuth shuffle

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



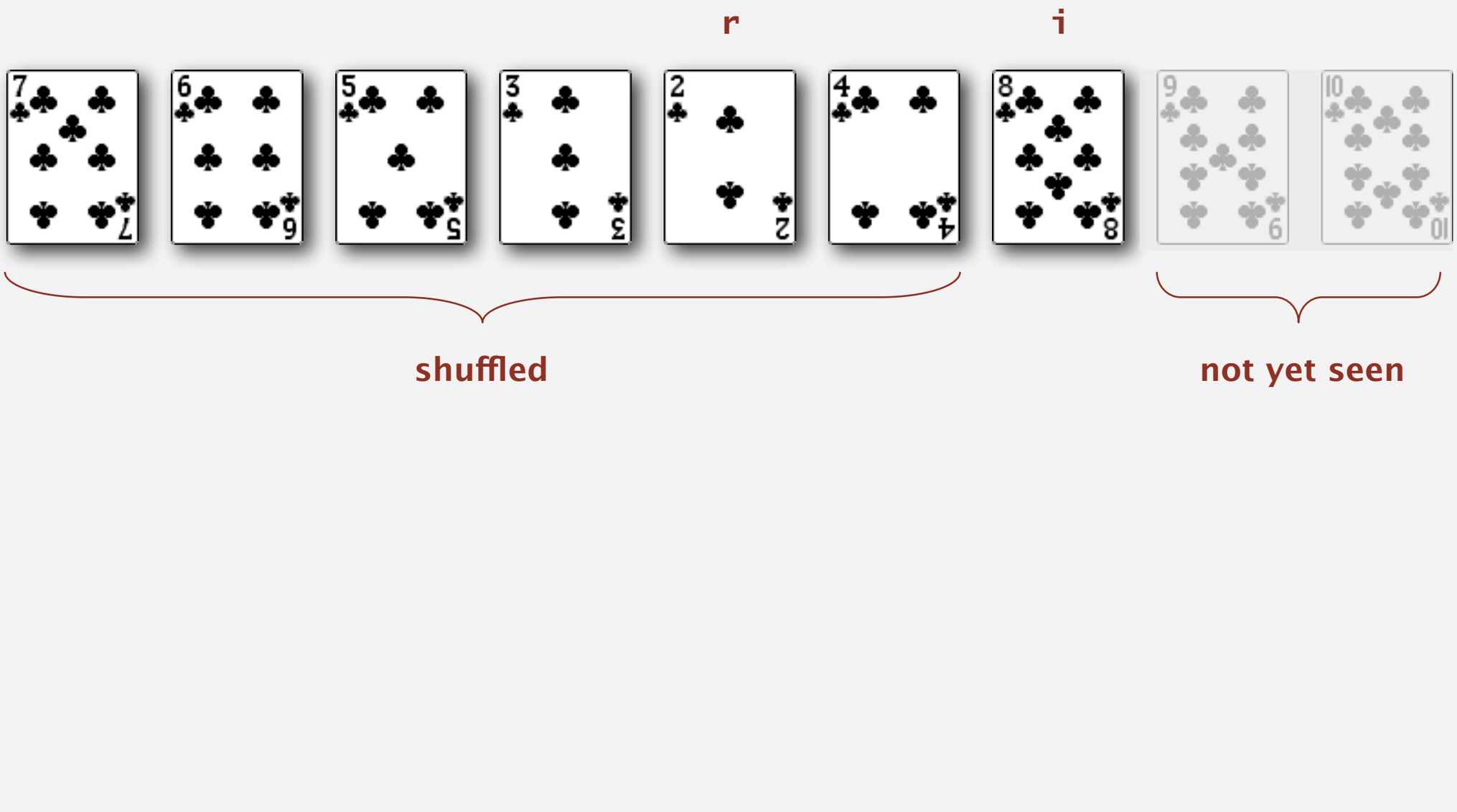
Knuth shuffle

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



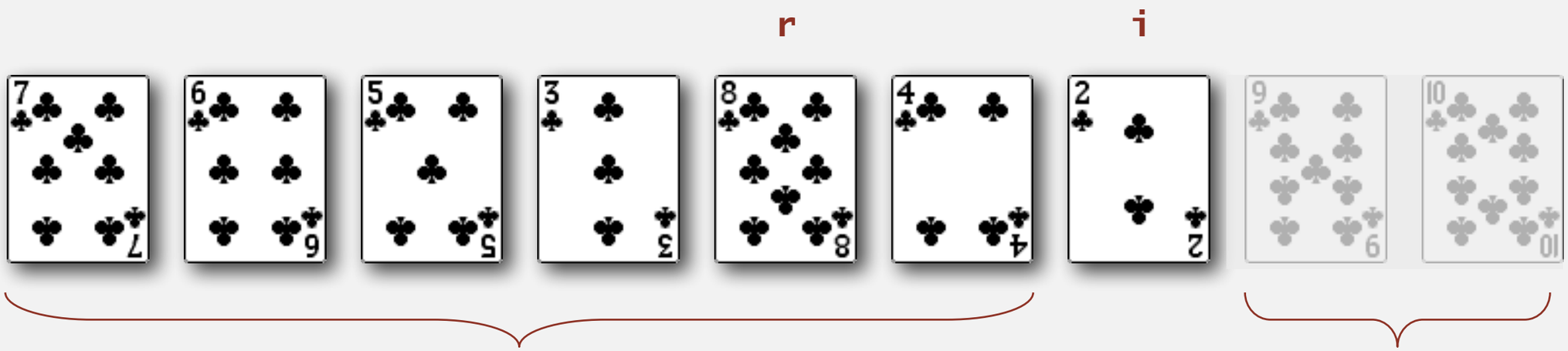
Knuth shuffle

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



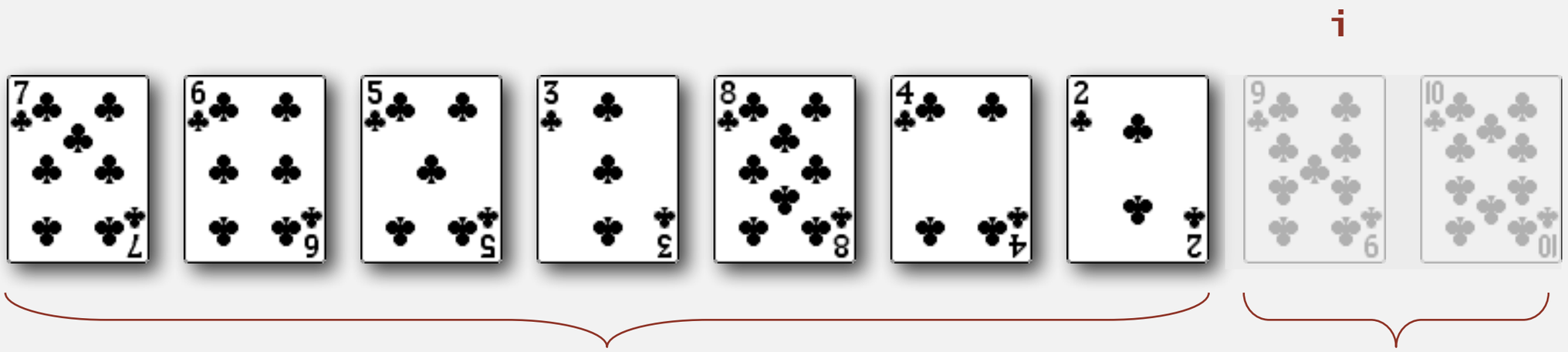
Knuth shuffle

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



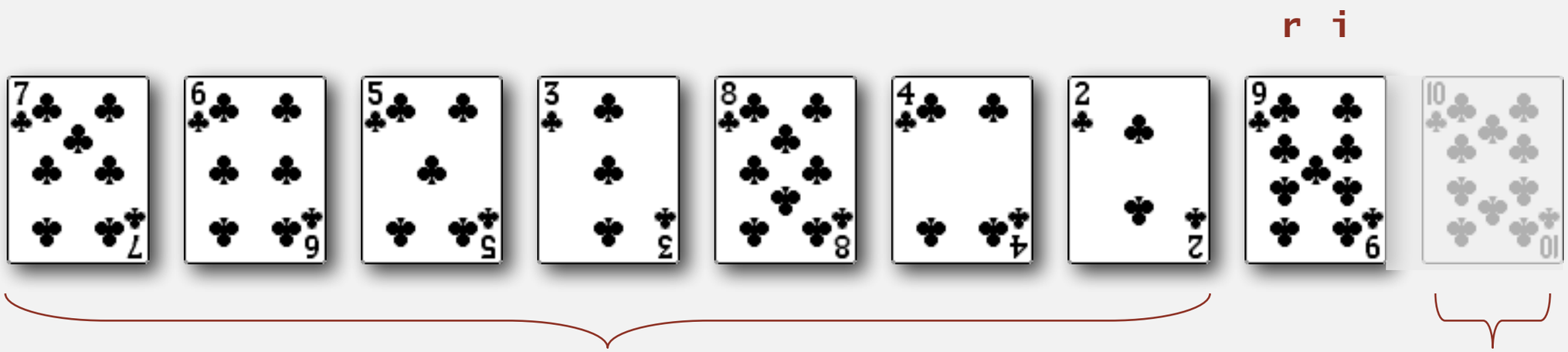
Knuth shuffle

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



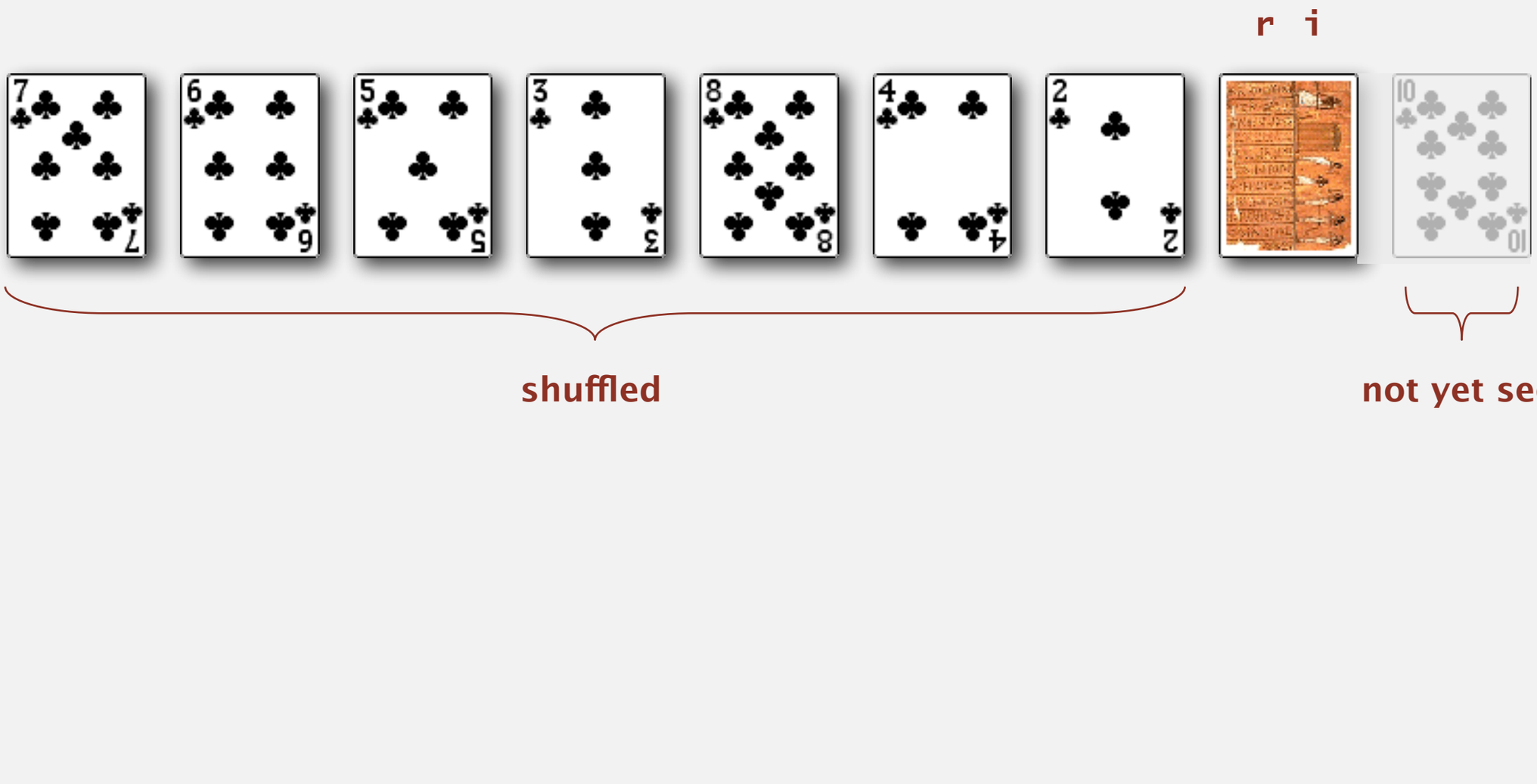
Knuth shuffle

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



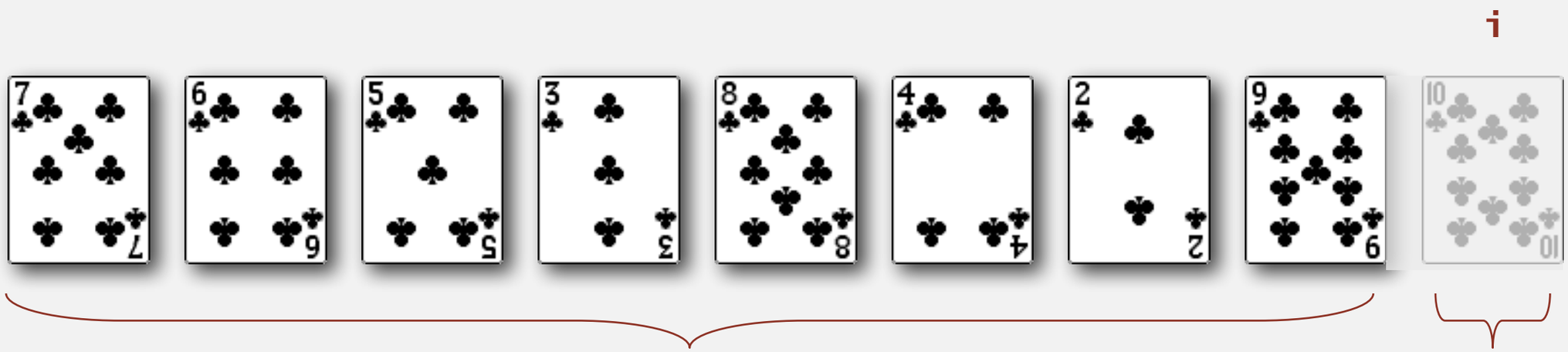
Knuth shuffle

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



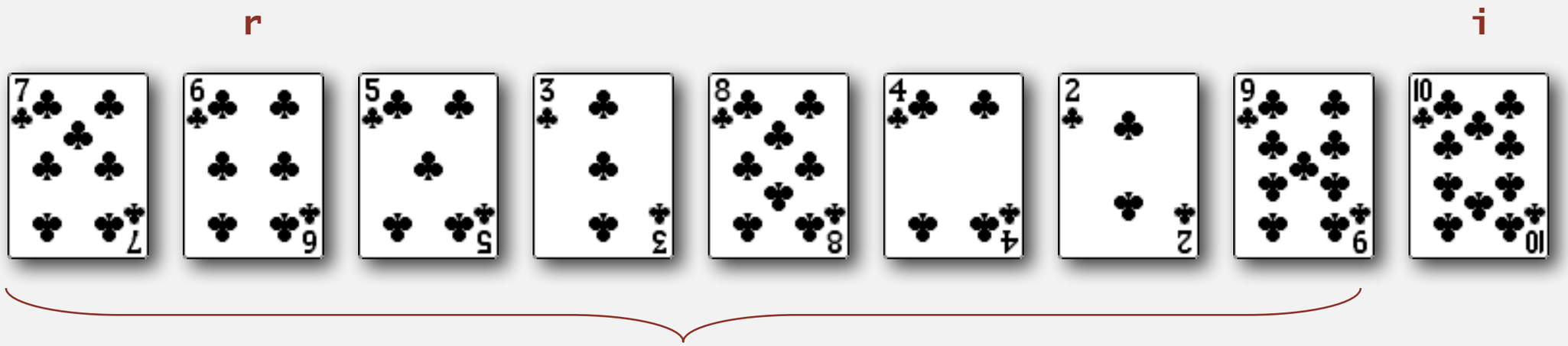
Knuth shuffle

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



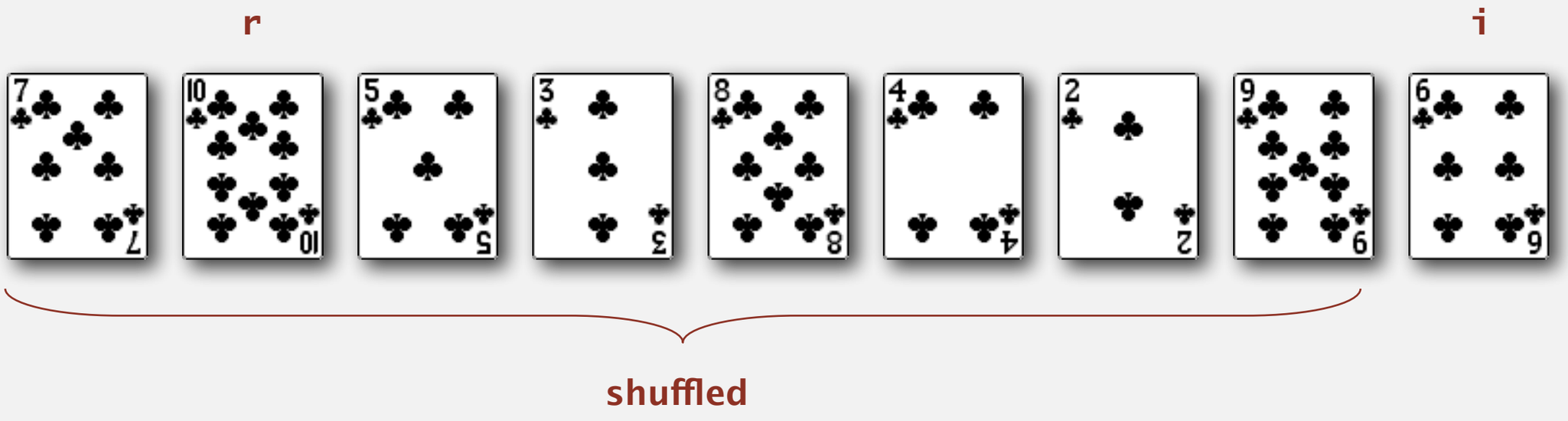
Knuth shuffle

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



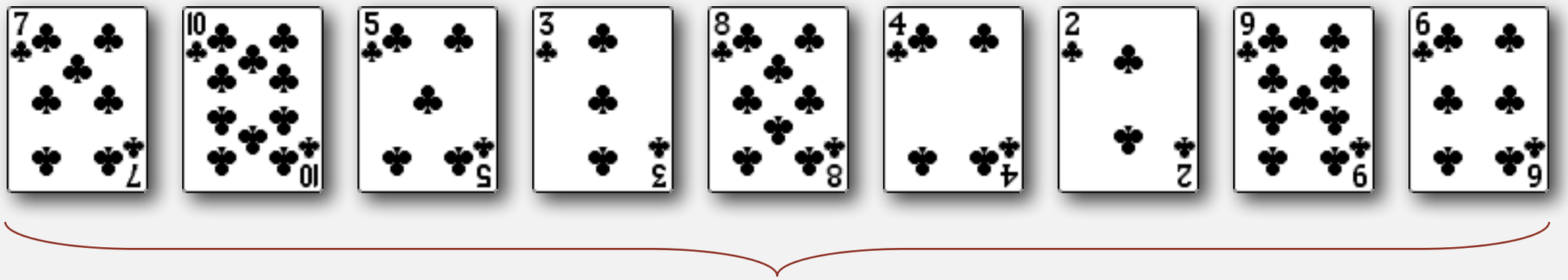
Knuth shuffle

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



Knuth shuffle

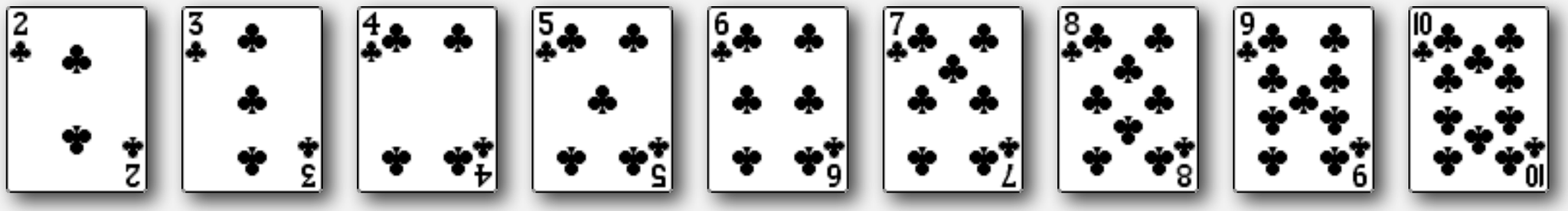
- In iteration `i`, pick integer `r` between `0` and `i` uniformly at random.
- Swap `a[i]` and `a[r]`.



shuffled

Knuth shuffle

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



Proposition. [Fisher-Yates 1938] Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time.

assuming integers uniformly at random

Knuth shuffle

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

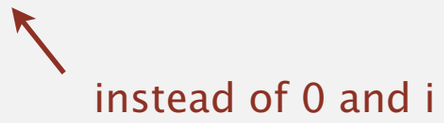
common bug: between 0 and N – 1
correct variant: between i and N – 1

```
public class StdRandom
{
   ...
   public static void shuffle(Object[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int r = StdRandom.uniform(i + 1);
         exch(a, i, r);
      }
   }
}
```

between 0 and i, not including I

Broken Knuth shuffle

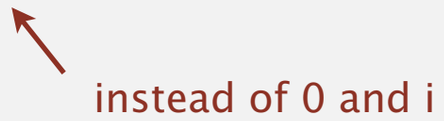
Q. What happens if integer is chosen between `0` and `N-1` ?



Broken Knuth shuffle

Q. What happens if integer is chosen between `0` and `N-1` ?

A. Not uniformly random!



permutation	Knuth shuffle	broken shuffle
A B C	1/6	4/27
A C B	1/6	5/27
B A C	1/6	5/27
B C A	1/6	5/27
C A B	1/6	4/27
C B A	1/6	4/27

probability of each result when shuffling { A, B, C }

https://www.i-programmer.info/programming/theory/2744-how-not-to-shuffle-the-kunth-fisher-yates-algorithm.html

War story (online poker)

Texas hold'em poker. Software must shuffle electronic cards.

How We Learned to Cheat at Online Poker: A Study in Software Security

https://www.developer.com/guides/how-we-learned-to-cheat-at-online-poker-a-study-in-software-security/

War story (online poker)

Shuffling algorithm in FAQ at www.planetpoker.com

```
for i := 1 to 52 do begin
   r := random(51) + 1;
   swap := card[r];
   card[r] := card[i];
   card[i] := swap;
end;
```

Any problems with this shuffling algorithm?

War story (online poker)

Shuffling algorithm in FAQ at www.planetpoker.com

```
for i := 1 to 52 do begin
   r := random(51) + 1;
   swap := card[r];
   card[r] := card[i];
   card[i] := swap;
end;
```

between 1 and 51

Bug 1. Random number `r` never 52 $\Rightarrow$ 52nd card can't end up in 52nd place.

Bug 2. Shuffle not uniform (should be between 1 and i).

Seed = milliseconds since midnight $\Rightarrow$ 86.4 million shuffles.

Exploit. After seeing 5 cards and synchronizing with server clock, can determine all future cards in real time.

> *" The generation of random numbers is too important to be left to chance. "*
>
> *— Robert R. Coveyou*

War story (online poker)

Best practices for shuffling (if your business depends on it).

- Use a hardware random-number generator that has passed both the FIPS 140-2 and the NIST statistical test suites.
- Continuously monitor statistic properties: hardware random-number generators are fragile and fail silently.
- Use an unbiased shuffling algorithm.

RANDOM.ORG

Bottom line. Shuffling a deck of cards is hard!