Sorting Algorithms II

Mark Matthews PhD

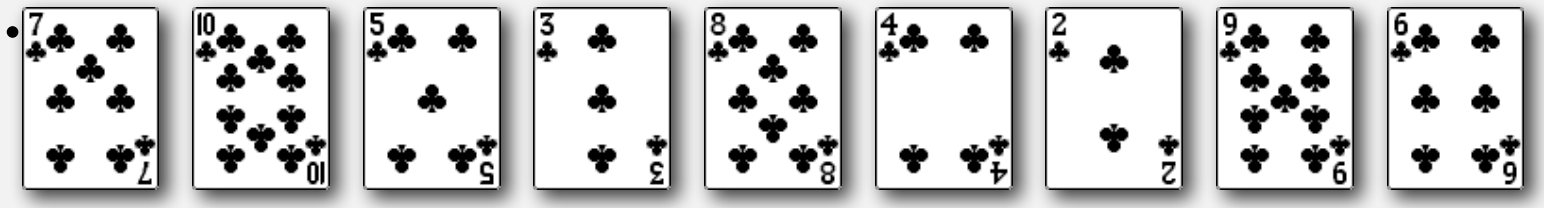
Summary: Elementary Sorts part II

- Insertion Sort -> enhanced
- Shuffle Sort
- Bogo Sort

Insertion Sort

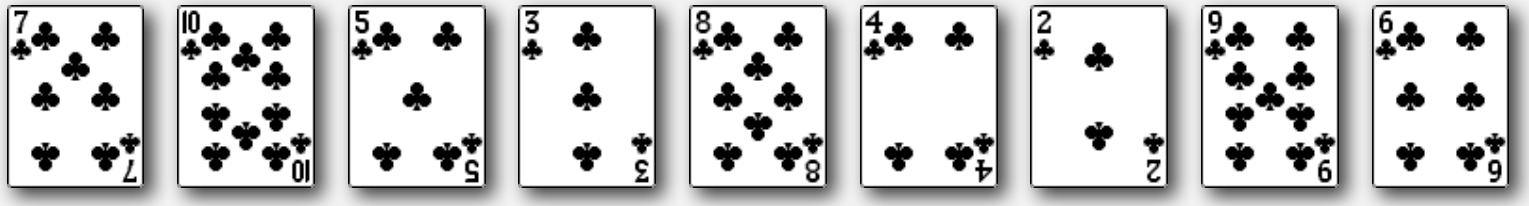
Insertion sort demo

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



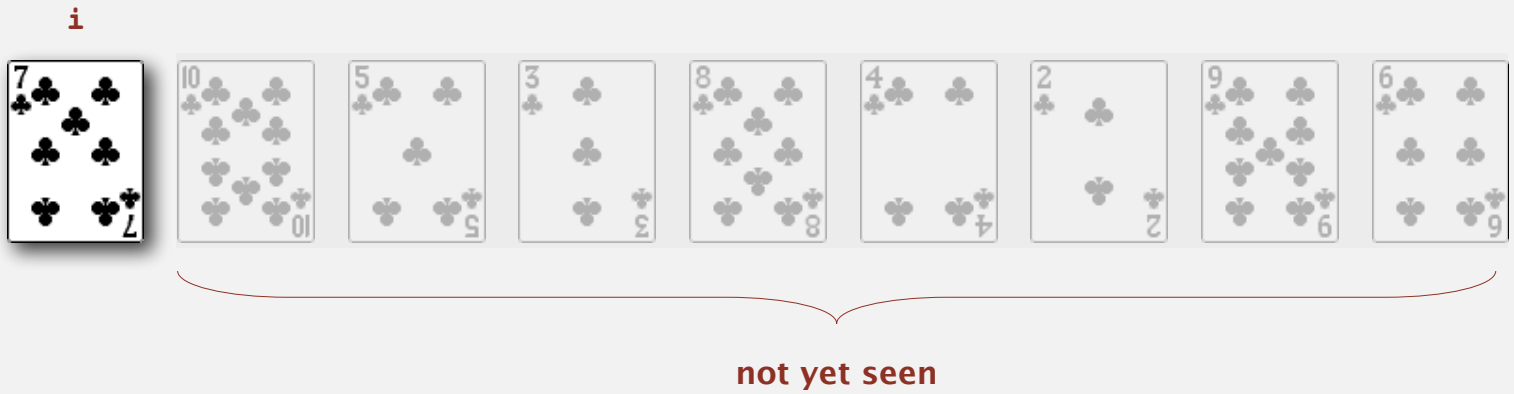
Insertion sort demo

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



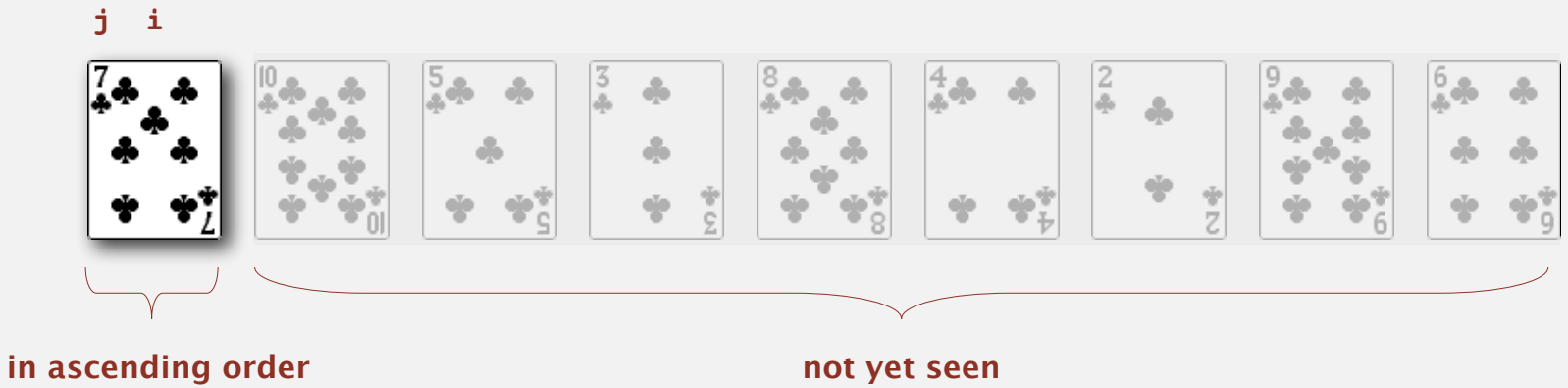
Insertion sort demo

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



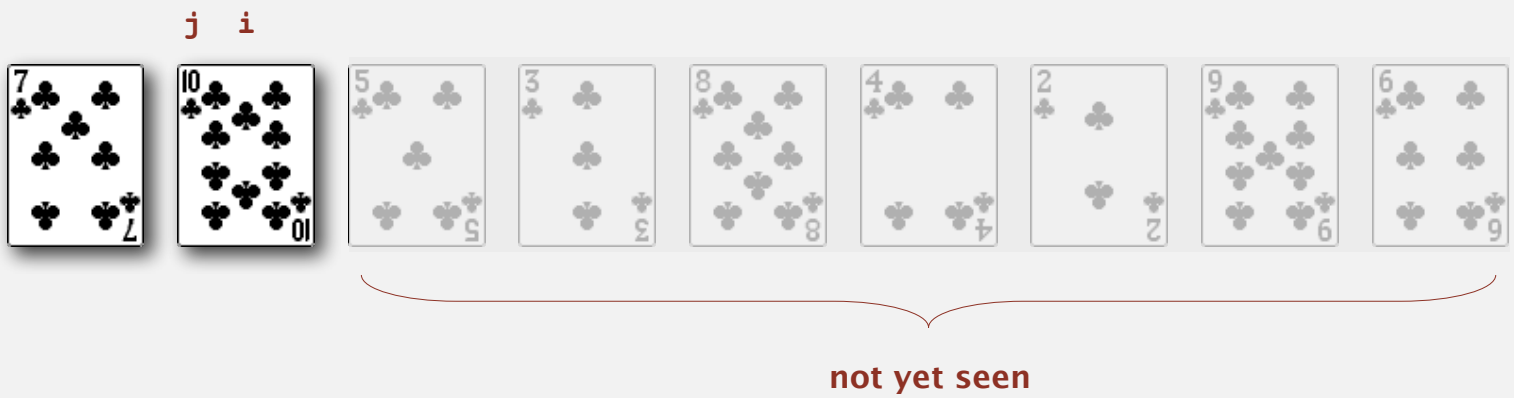
Insertion sort demo

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



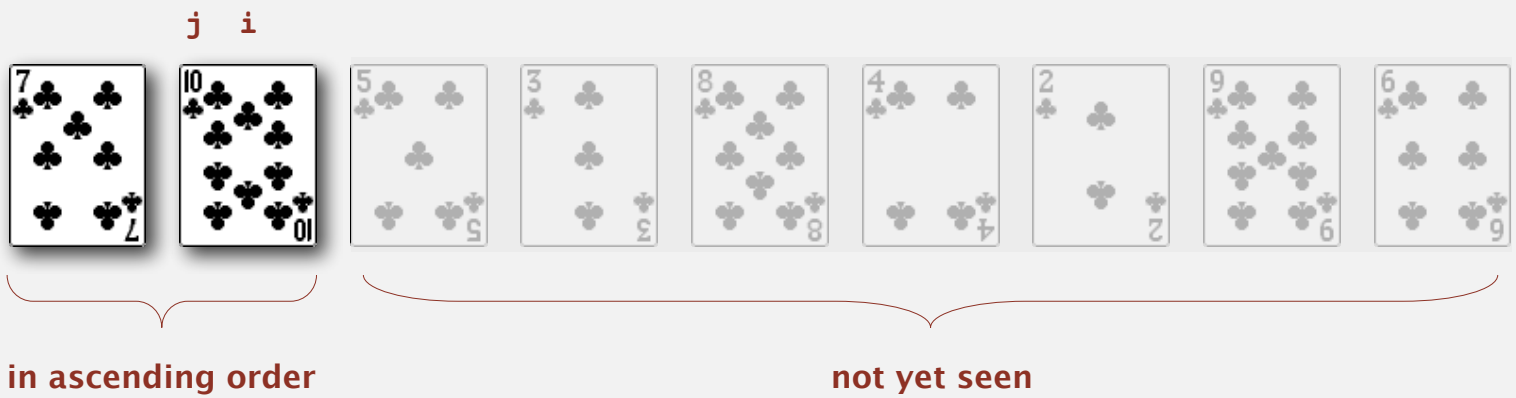
Insertion sort demo

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



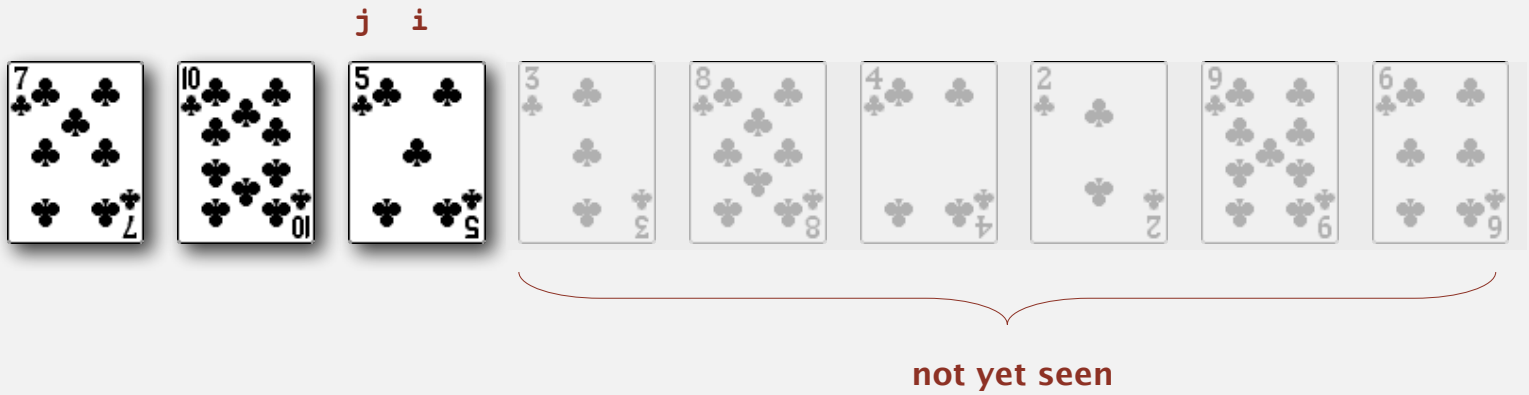
Insertion sort demo

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



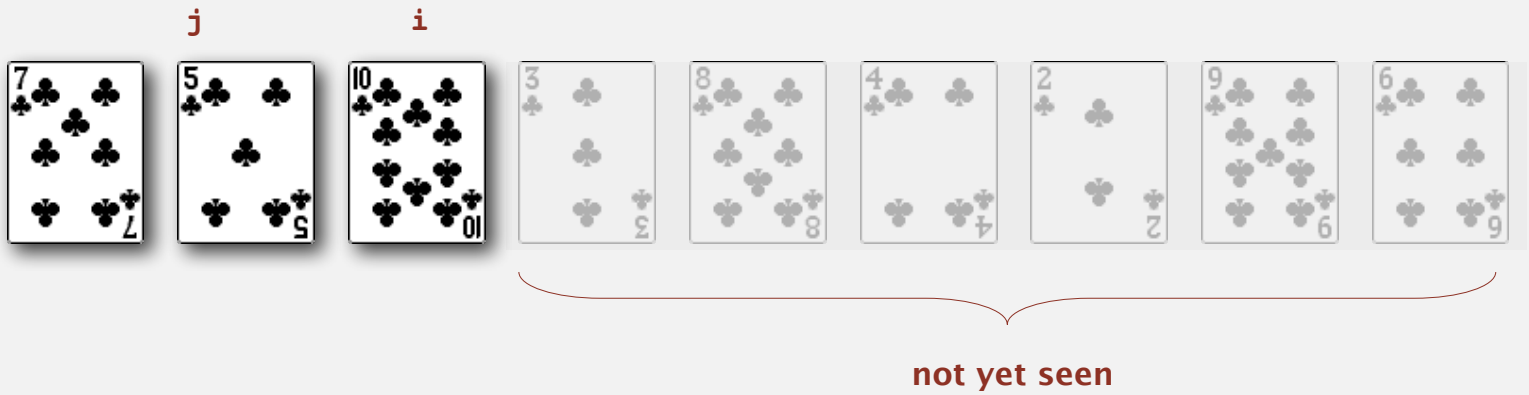
Insertion sort demo

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



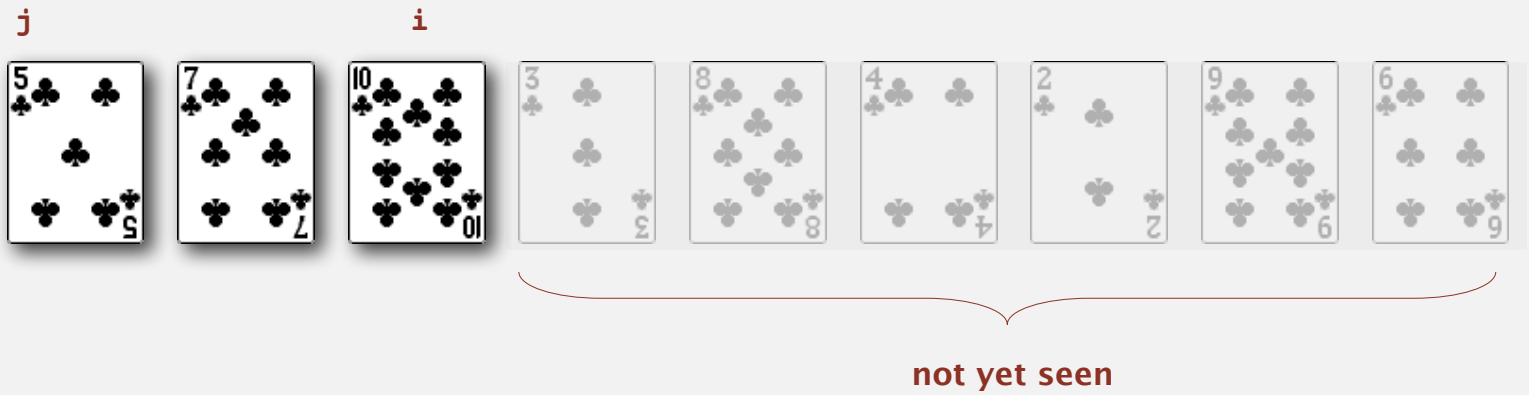
Insertion sort demo

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



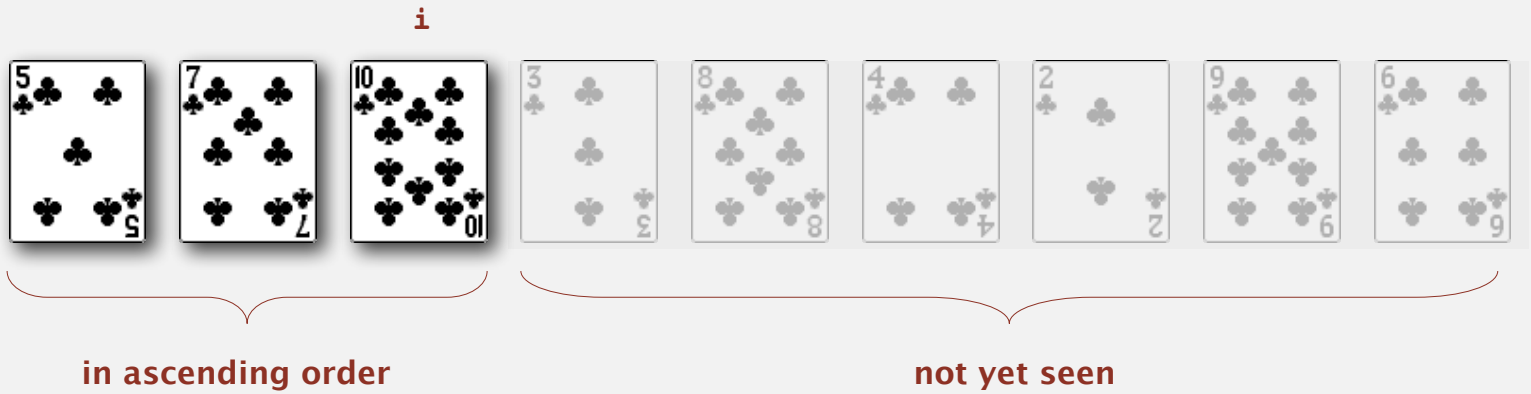
Insertion sort demo

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



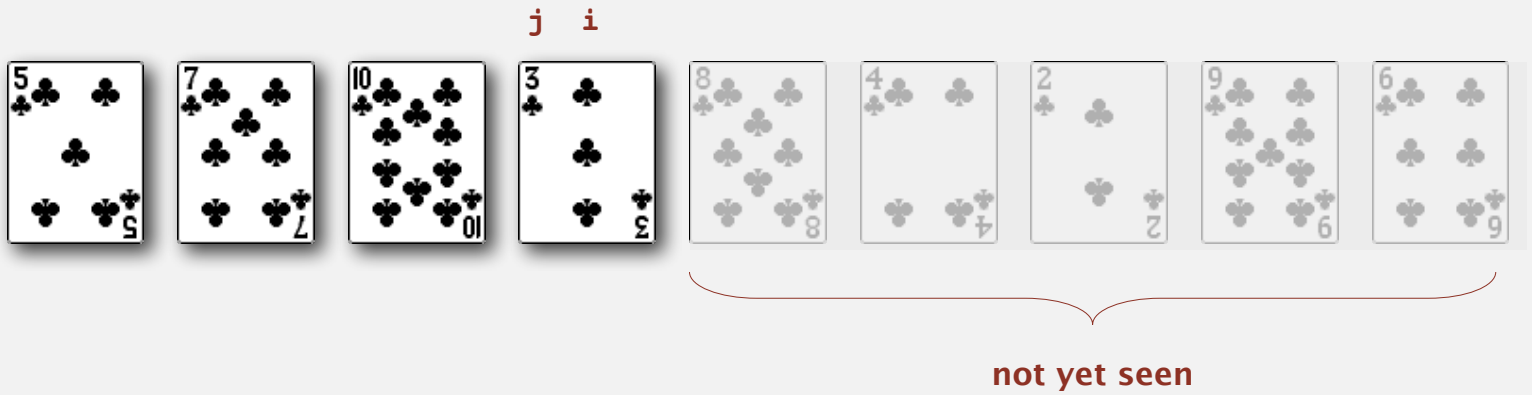
Insertion sort demo

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



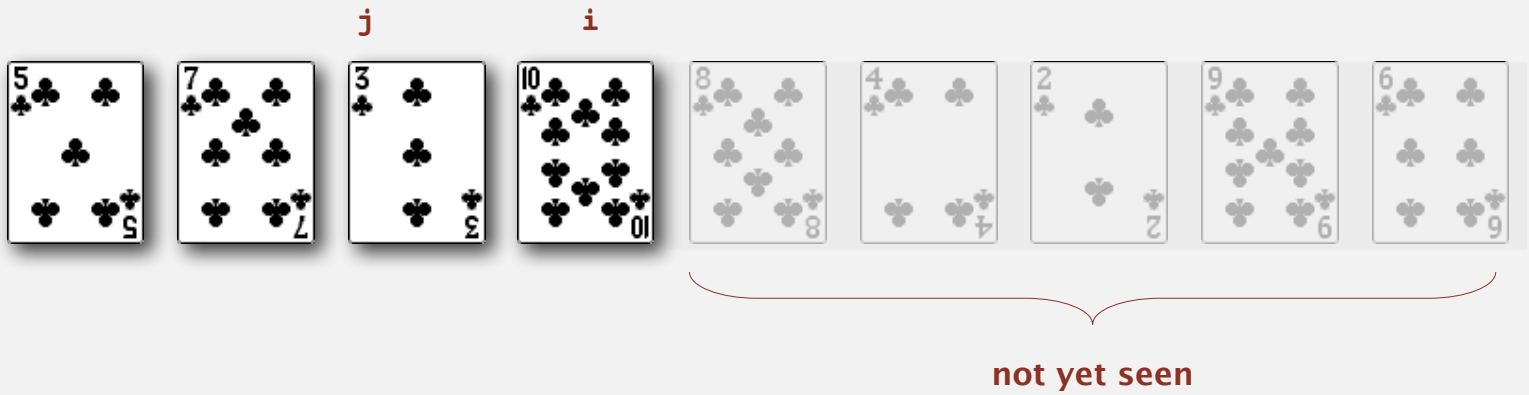
Insertion sort demo

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



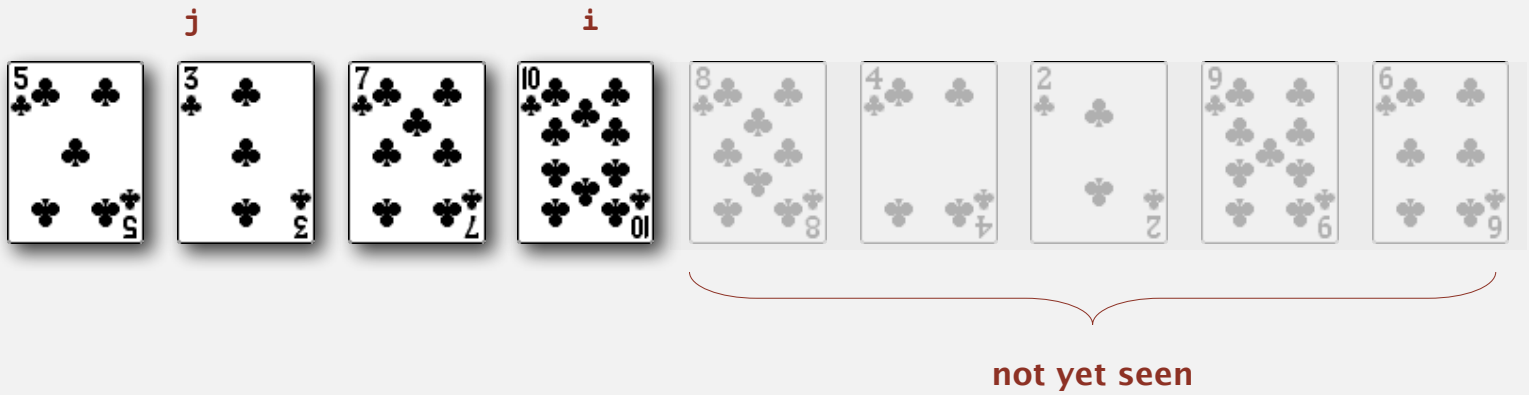
Insertion sort demo

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



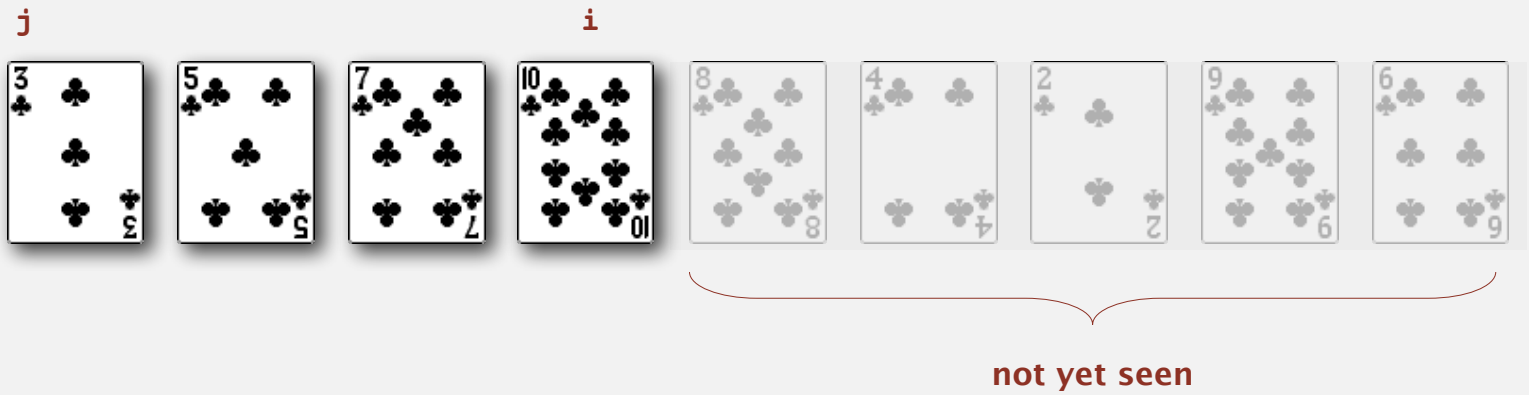
Insertion sort demo

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



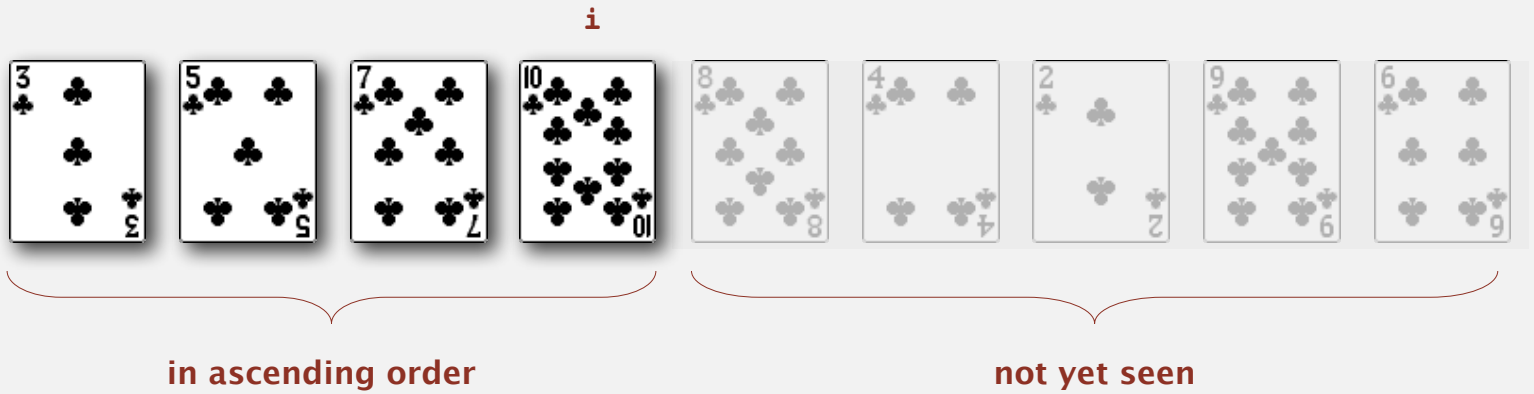
Insertion sort demo

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



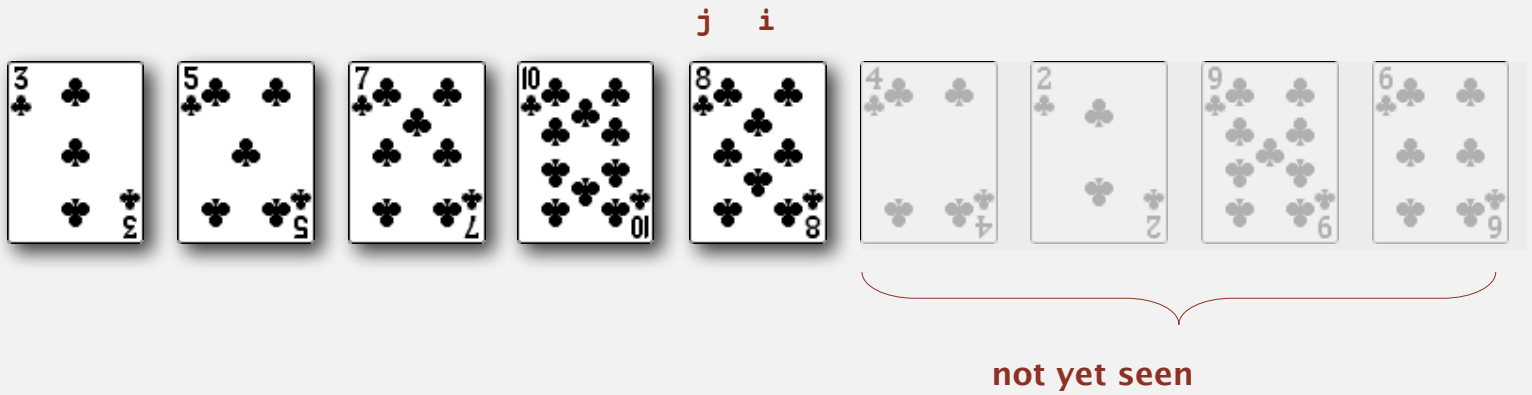
Insertion sort demo

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



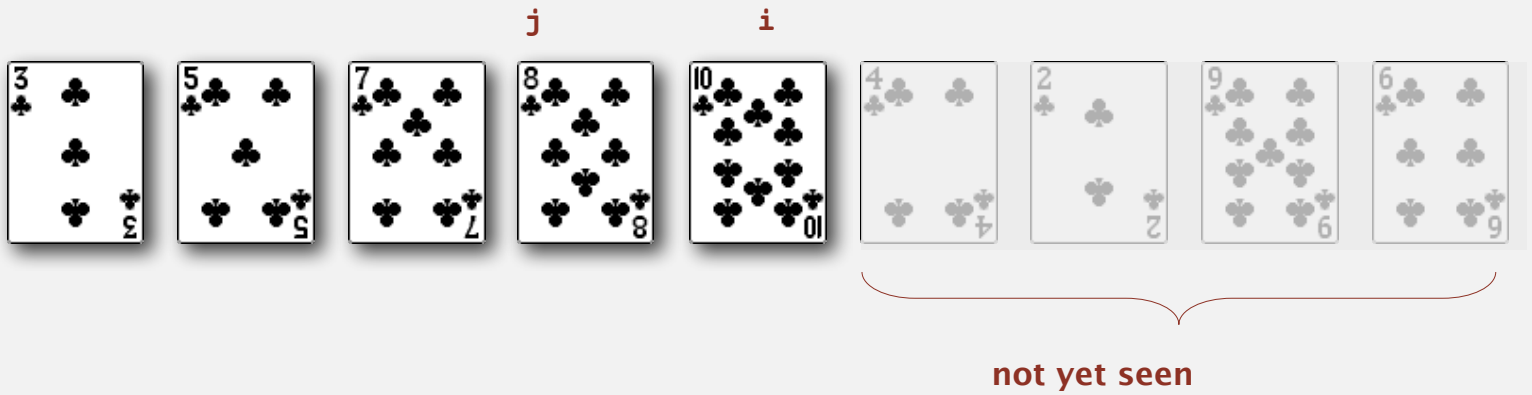
Insertion sort demo

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



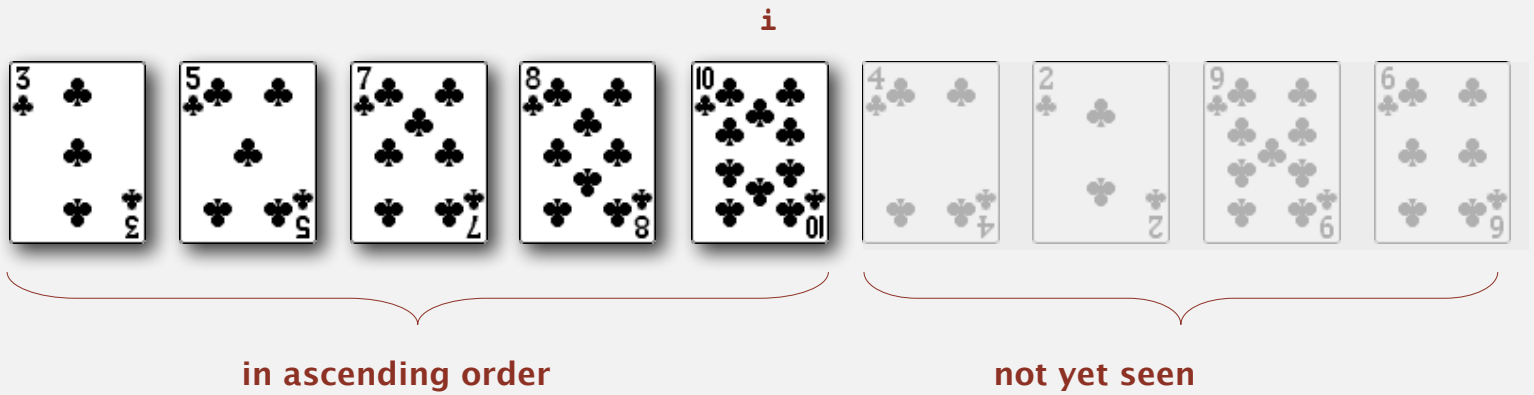
Insertion sort demo

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



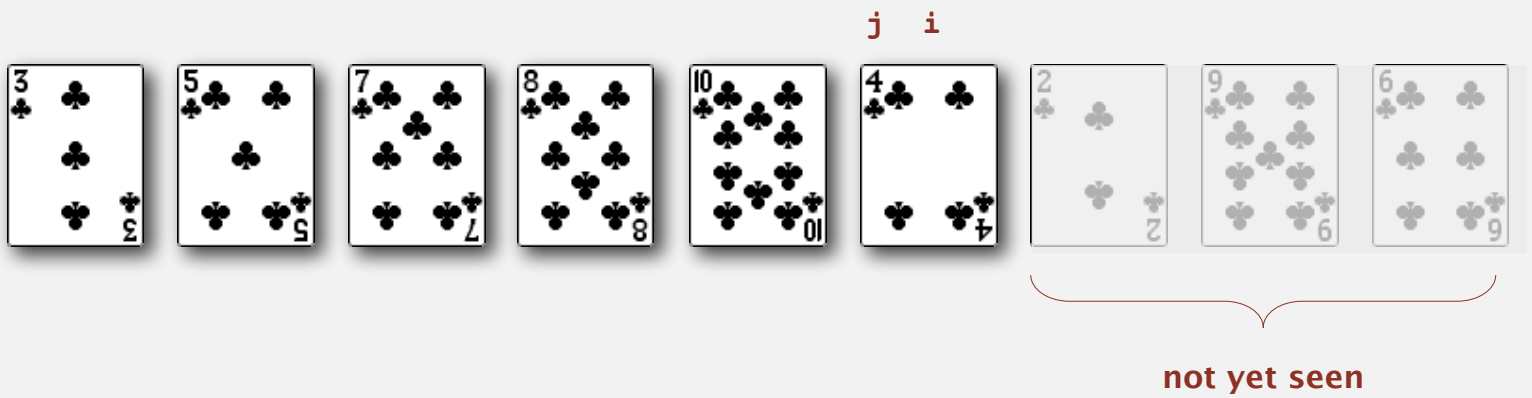
Insertion sort demo

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



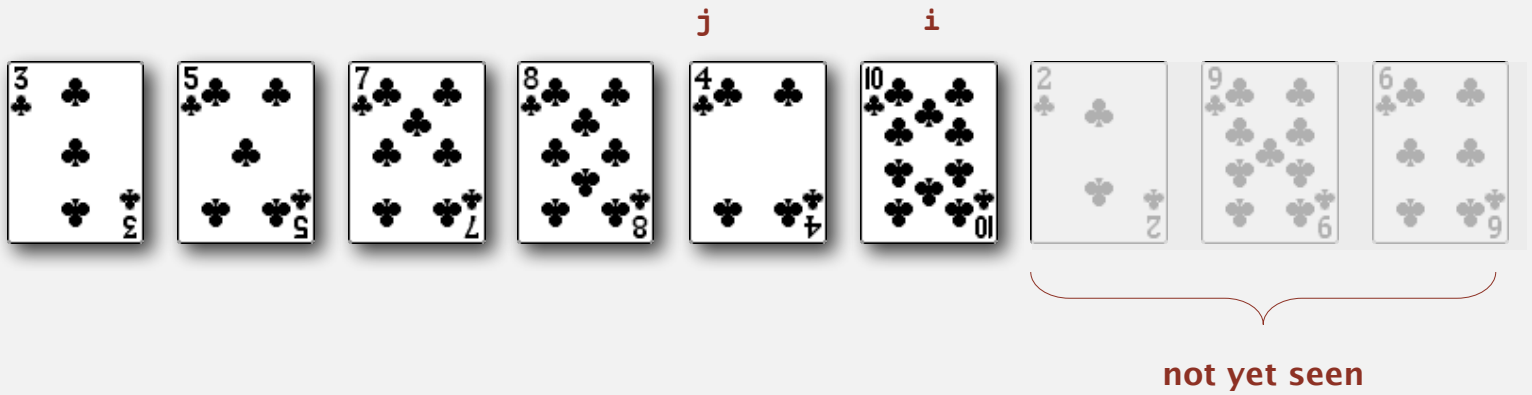
Insertion sort demo

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



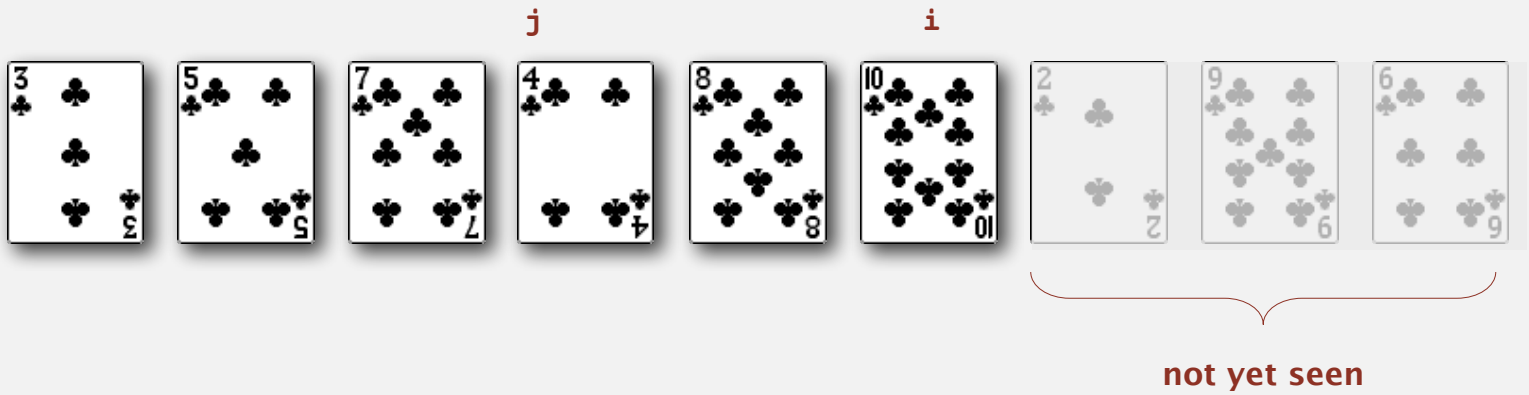
Insertion sort demo

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



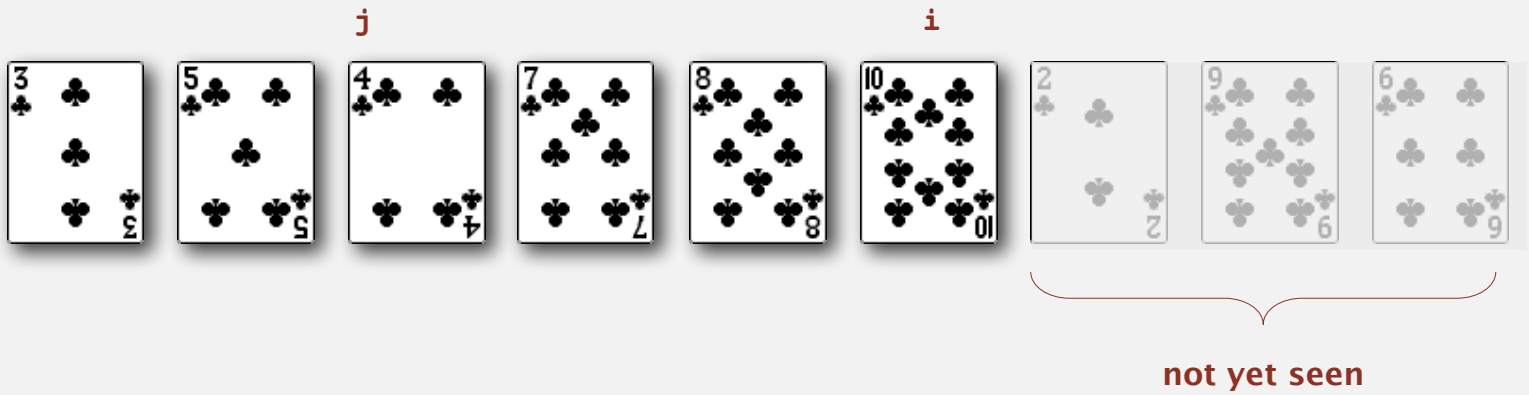
Insertion sort demo

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



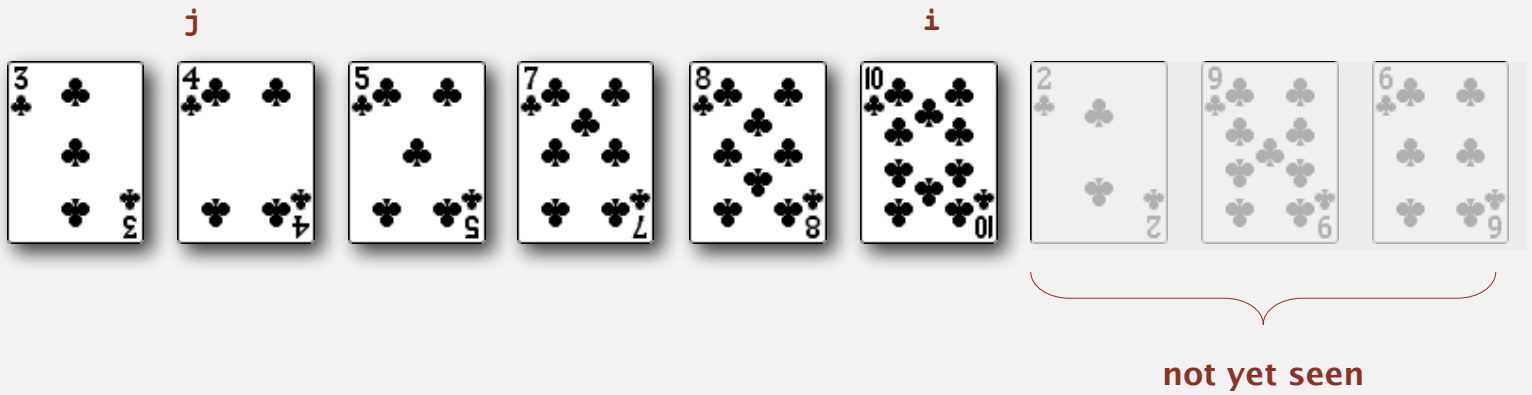
Insertion sort demo

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



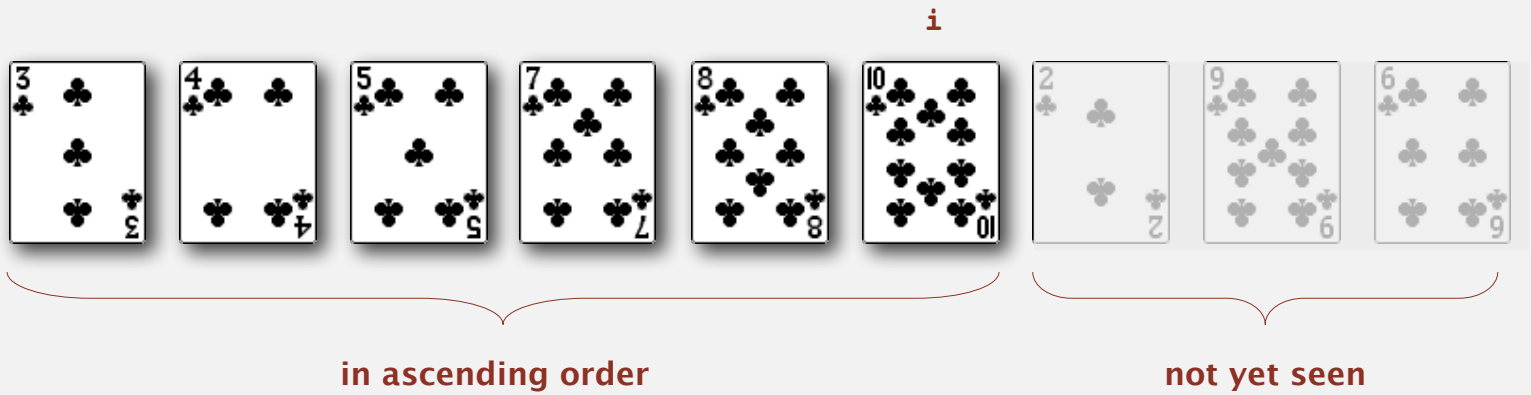
Insertion sort demo

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



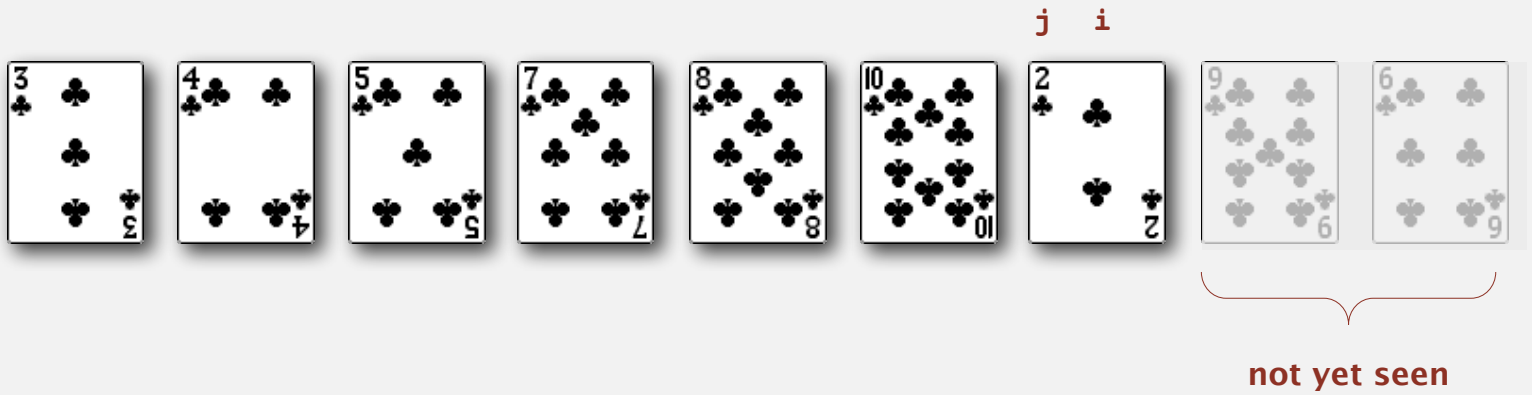
Insertion sort demo

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



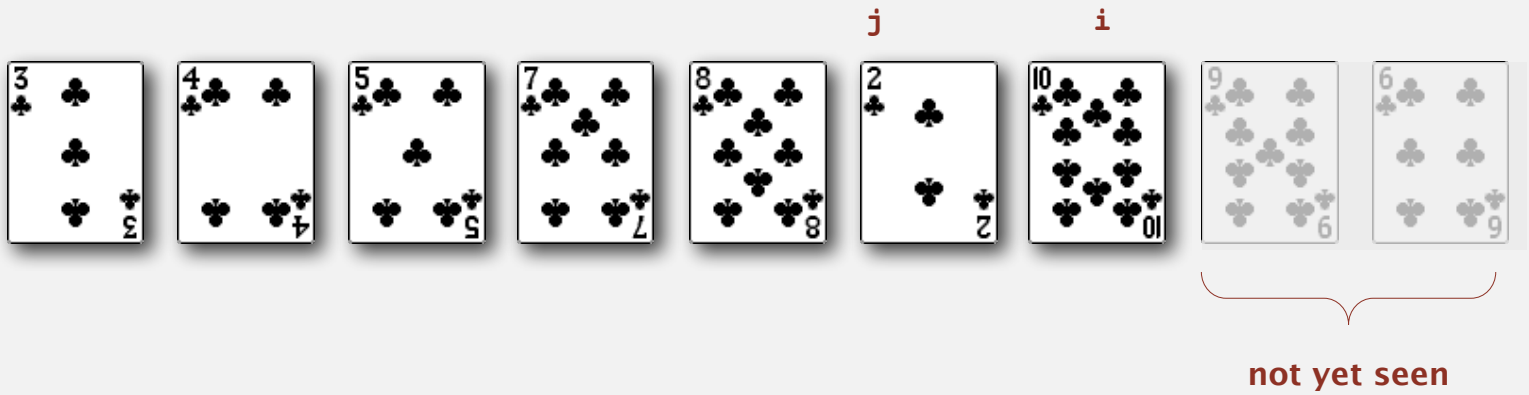
Insertion sort demo

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



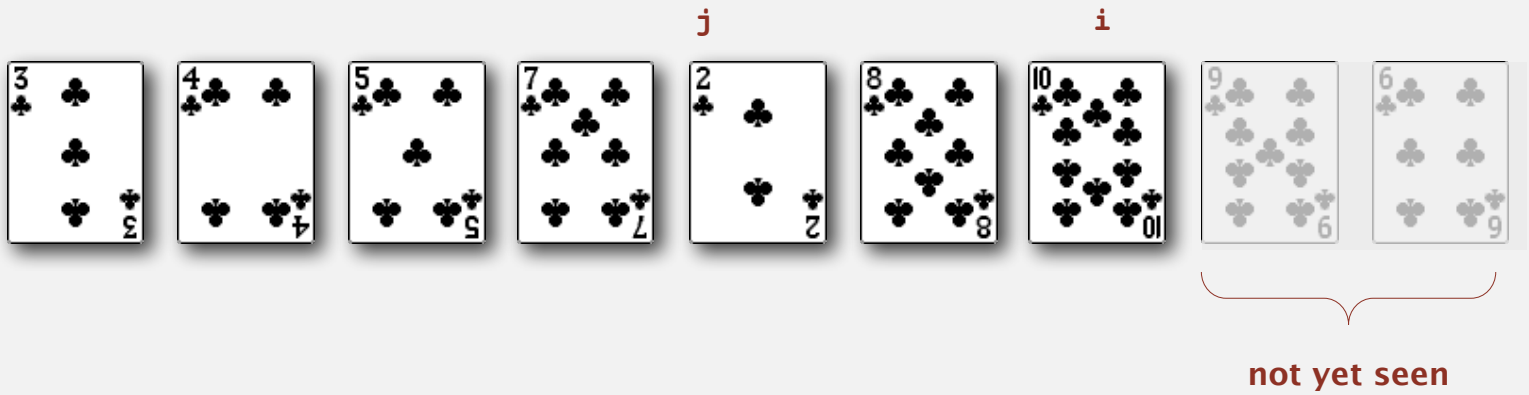
Insertion sort demo

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



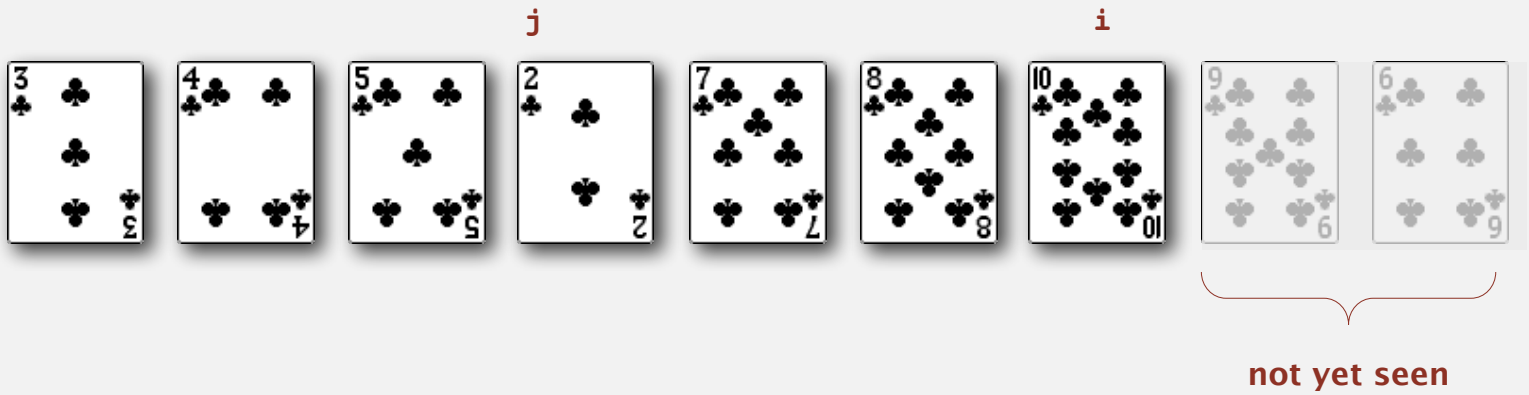
Insertion sort demo

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



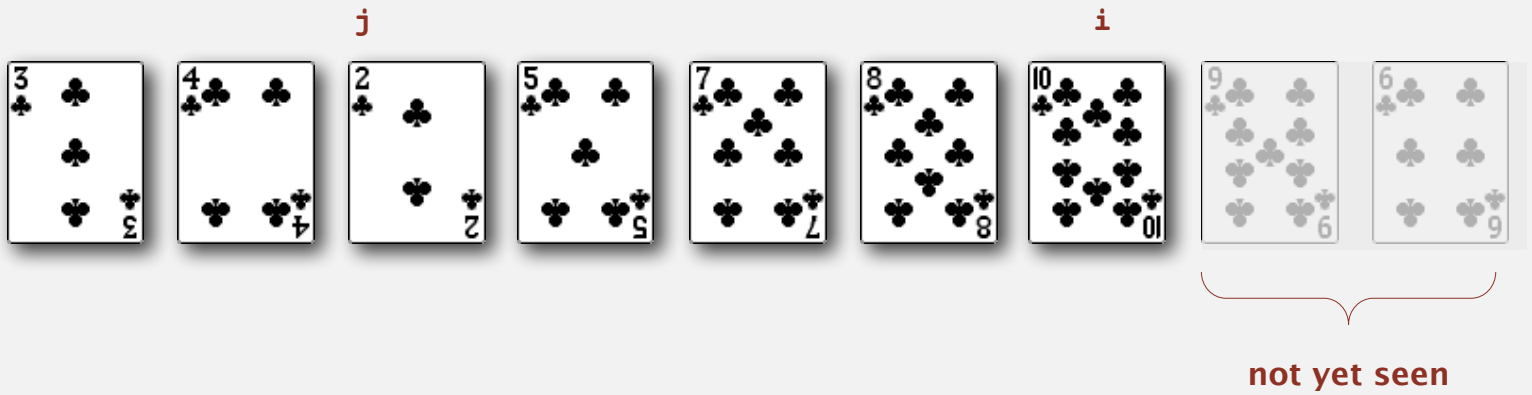
Insertion sort demo

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



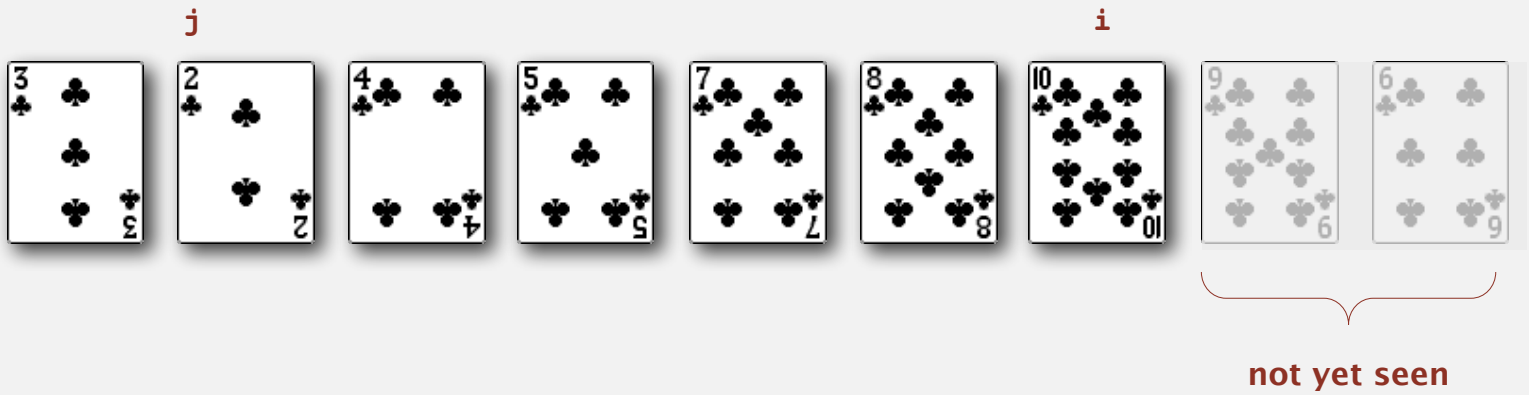
Insertion sort demo

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



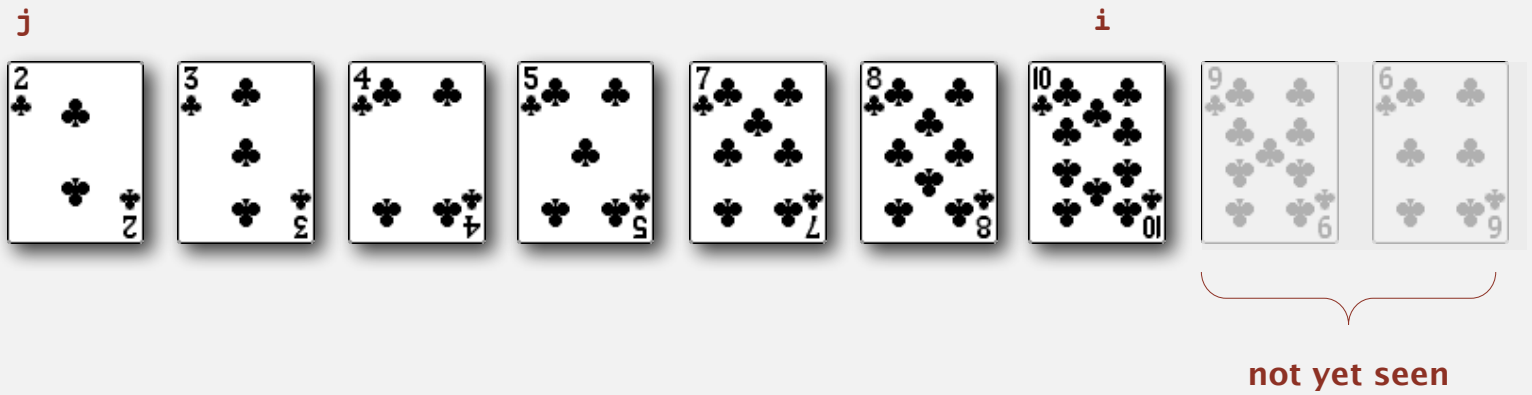
Insertion sort demo

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



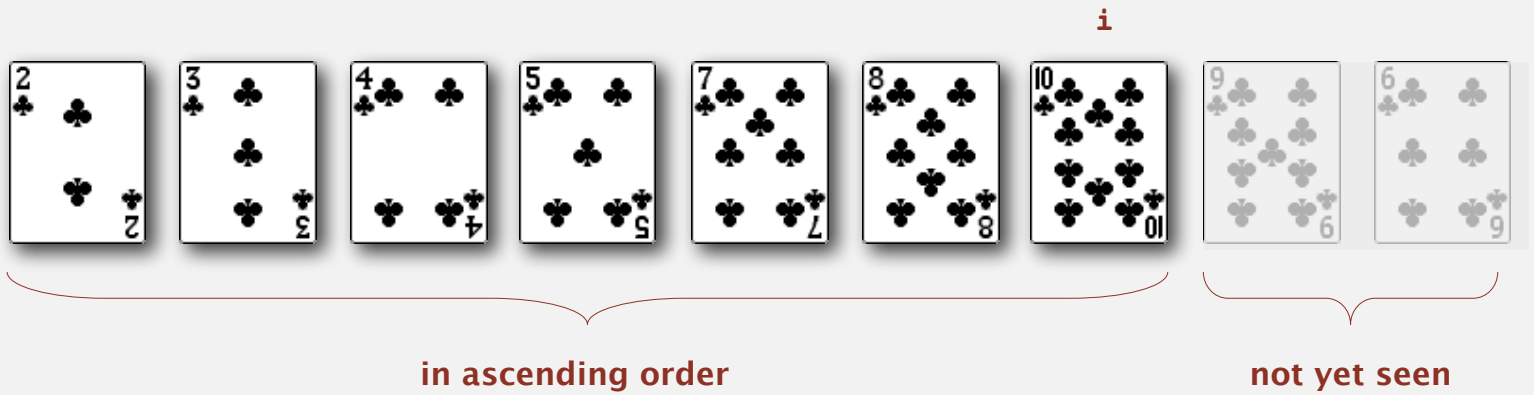
Insertion sort demo

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



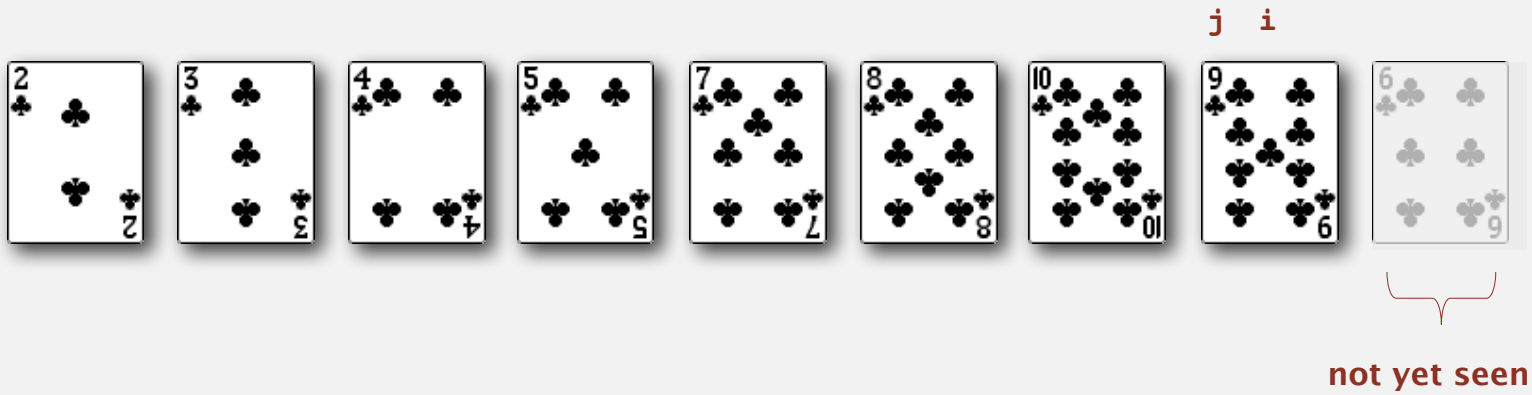
Insertion sort demo

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



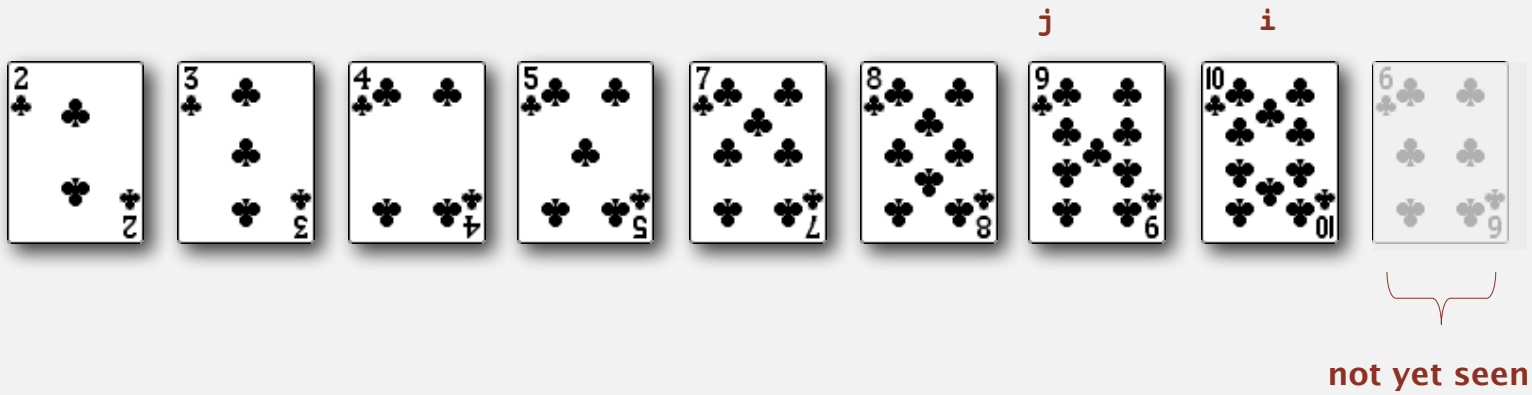
Insertion sort demo

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



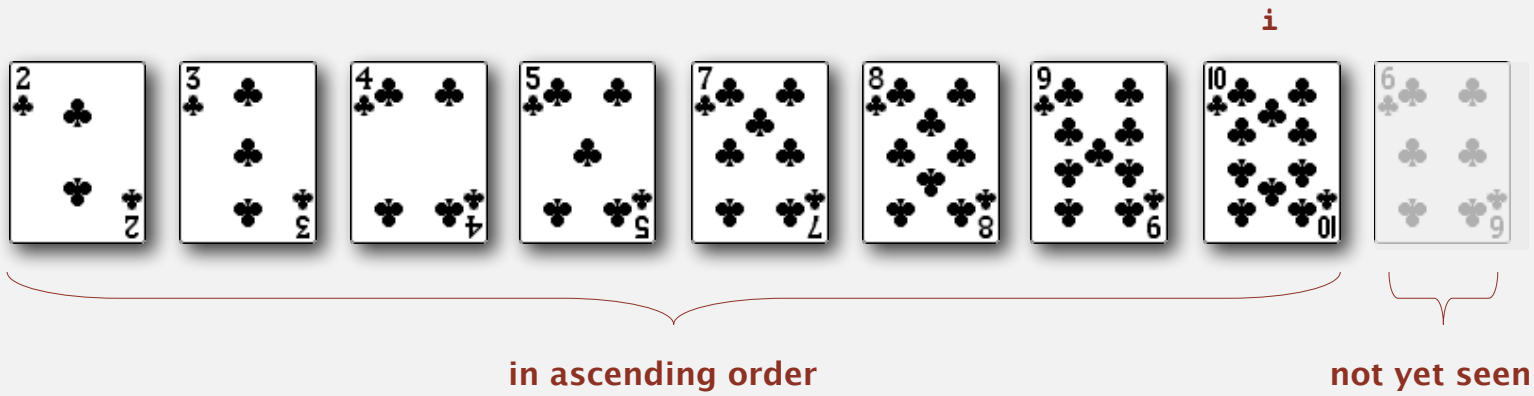
Insertion sort demo

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



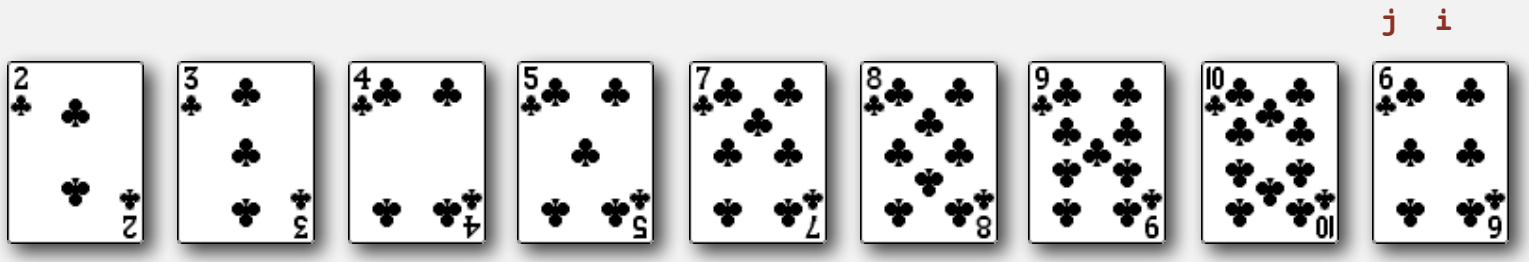
Insertion sort demo

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



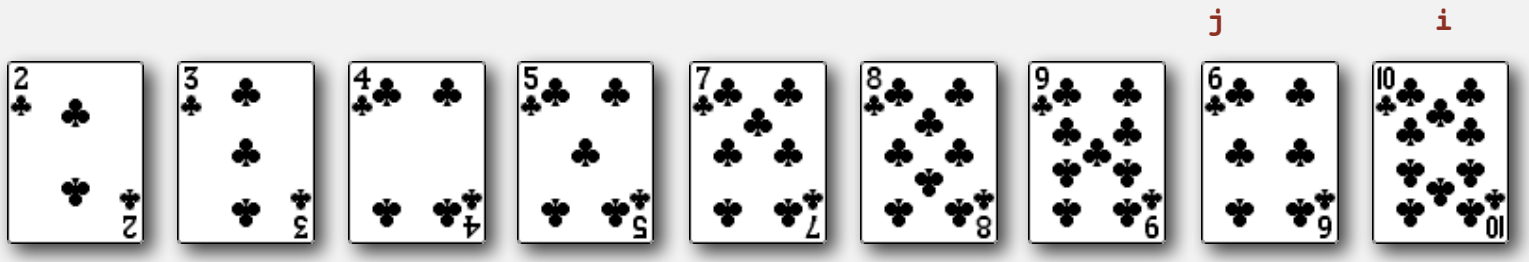
Insertion sort demo

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



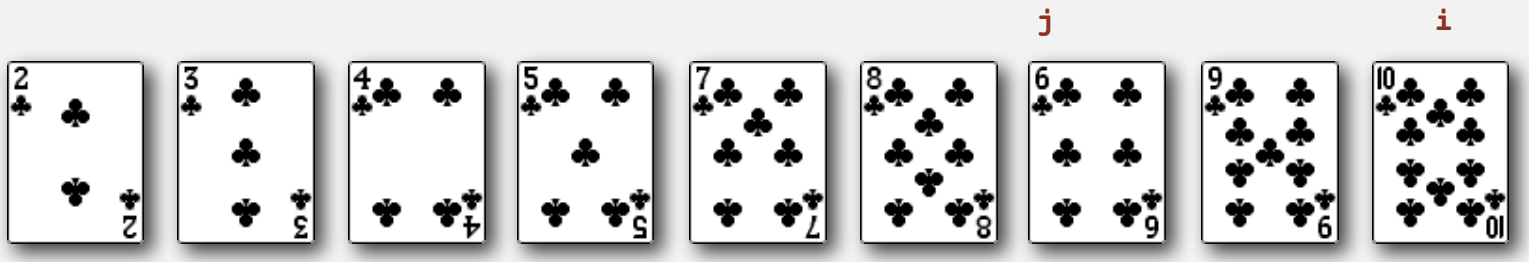
Insertion sort demo

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



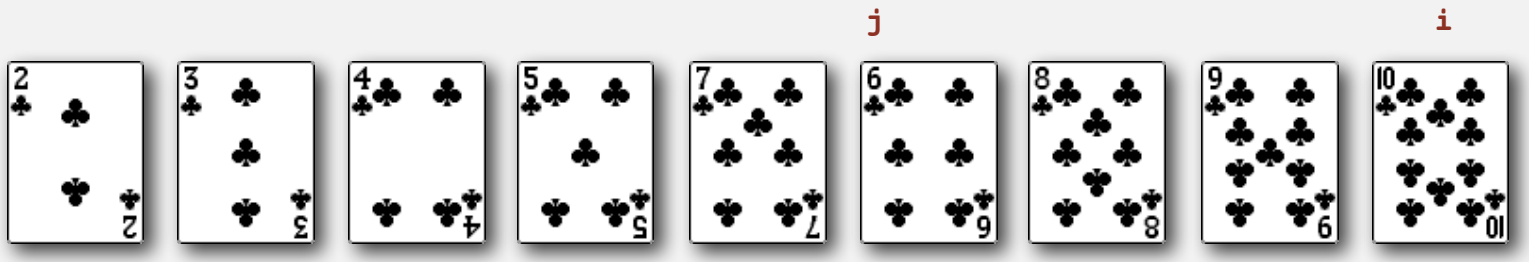
Insertion sort demo

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



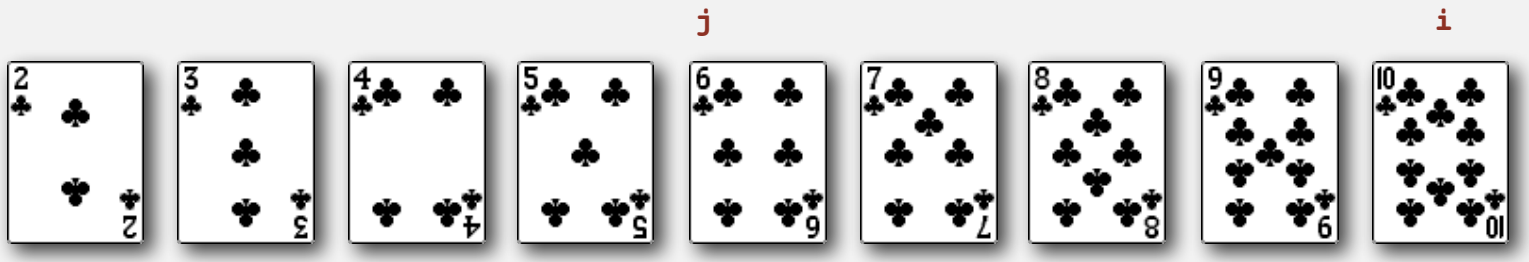
Insertion sort demo

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



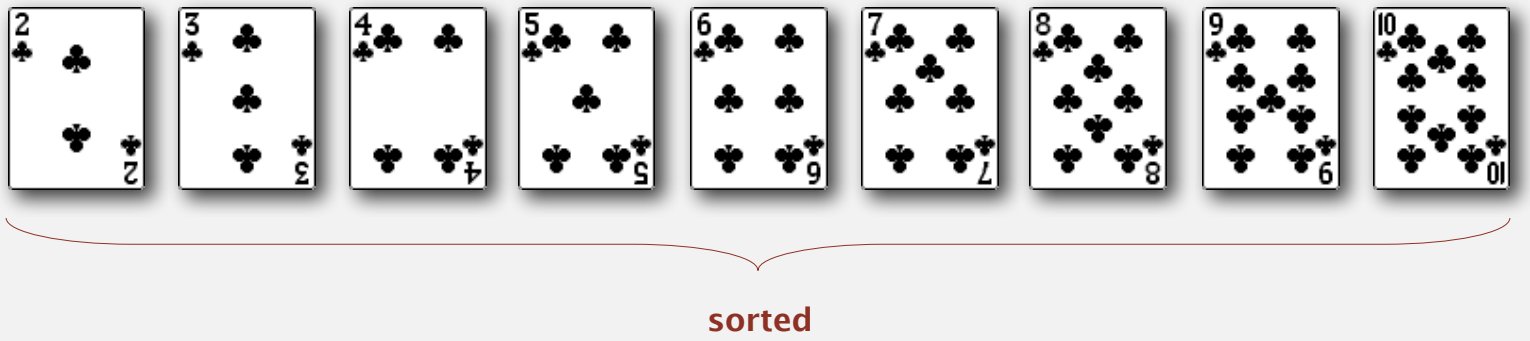
Insertion sort demo

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



Insertion sort demo

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



Insertion Sort Dancing

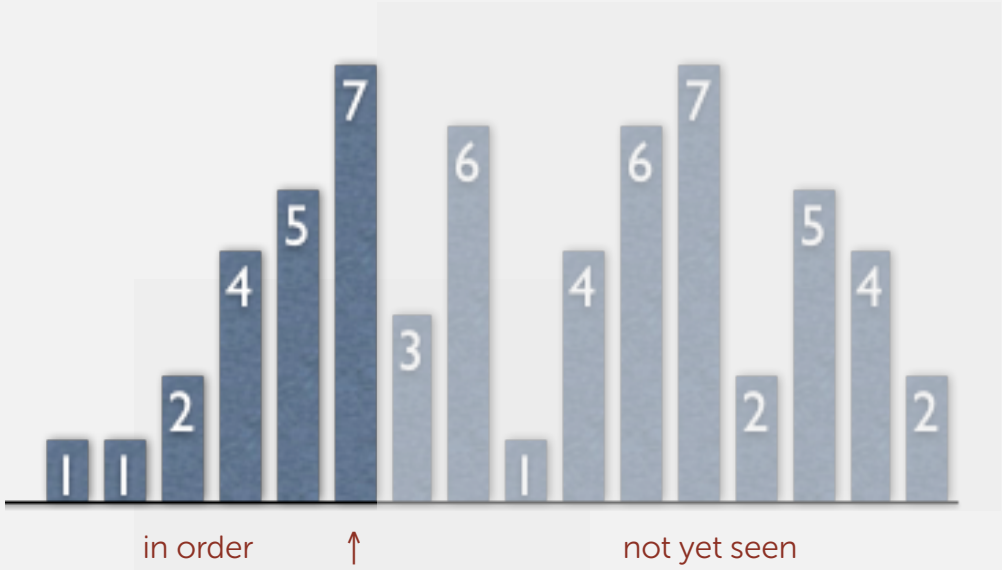
https://www.youtube.com/watch?v=ROalU379l3U

Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

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



Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

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

```
while (j > 0 && arr[j - 1] > valueToSort)
{
                arr[j] = arr[j - 1];
                j--;
        }
        arr[j] = valueToSort;
```

Insertion sort: Java implementation

```
private static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int valueToSort = arr[i];
            int j = i;
            while (j > 0 && arr[j - 1] > valueToSort)
{
                arr[j] = arr[j - 1];
                j--;
            }
            arr[j] = valueToSort;
        }
    }
```

Insertion sort: analysis

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

Pf. Expect each entry to move halfway back.

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

entries in gray do not move

entry in red is `a[j]`

entries in black moved one position right for insertion

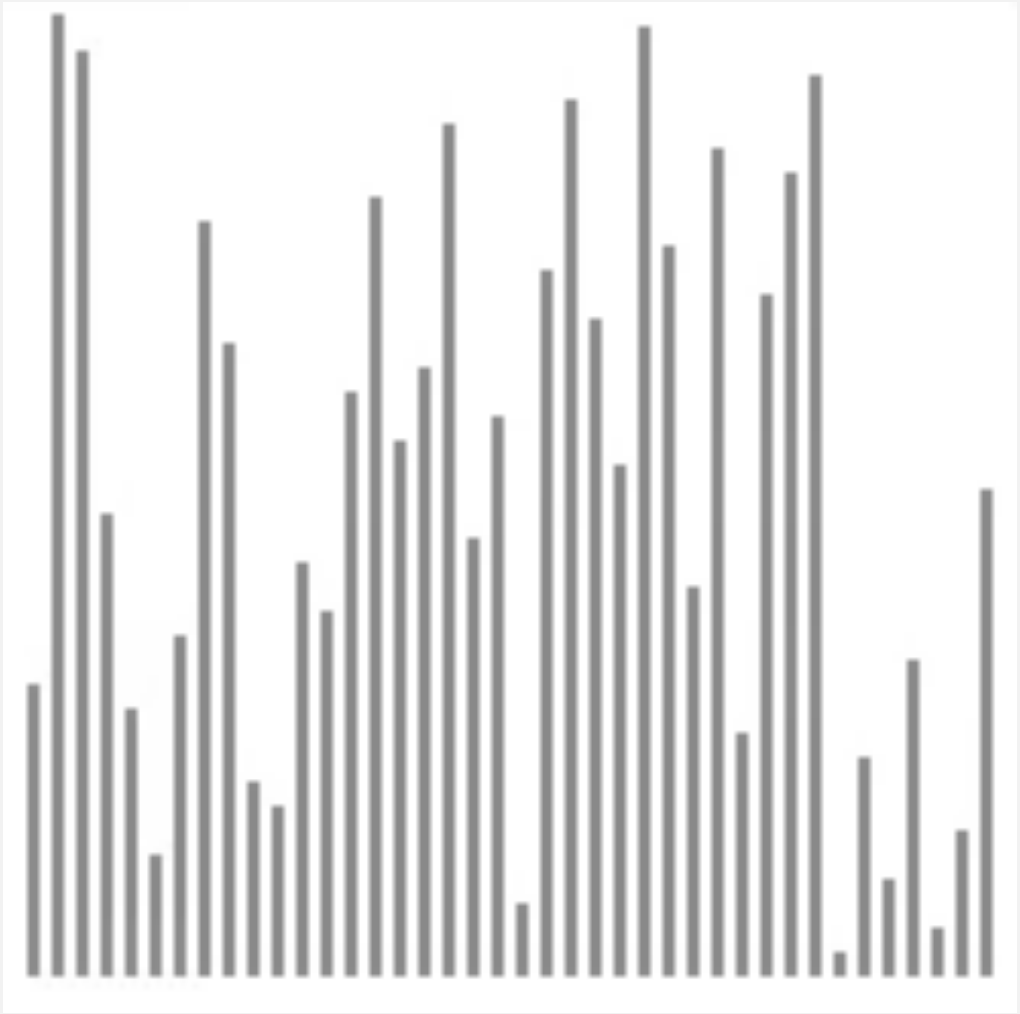
Trace of insertion sort (array contents just after each insertion)

Insertion sort: trace

		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: animation

40 random items



algorithm position

in order

not yet seen

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

Insertion sort: analysis

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

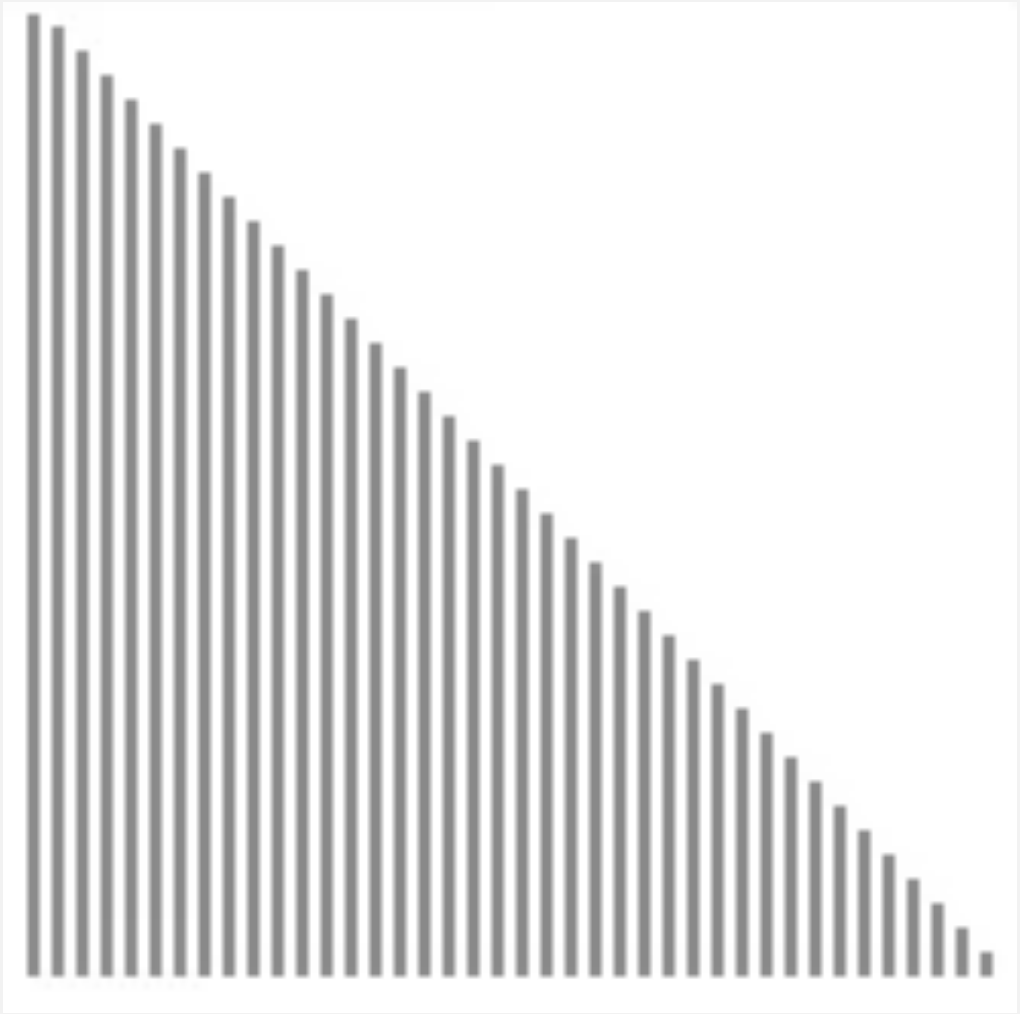
Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

X T S R P O M L F E A

First case much much faster than selection sort – linear vs. quadratic.
Second case slower than selection sort

Insertion sort: animation

40 reverse-sorted items



algorithm position

in order

not yet seen

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

Insertion sort: partially-sorted arrays

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

A E E L M O T R X P S

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

(6 inversions)

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

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

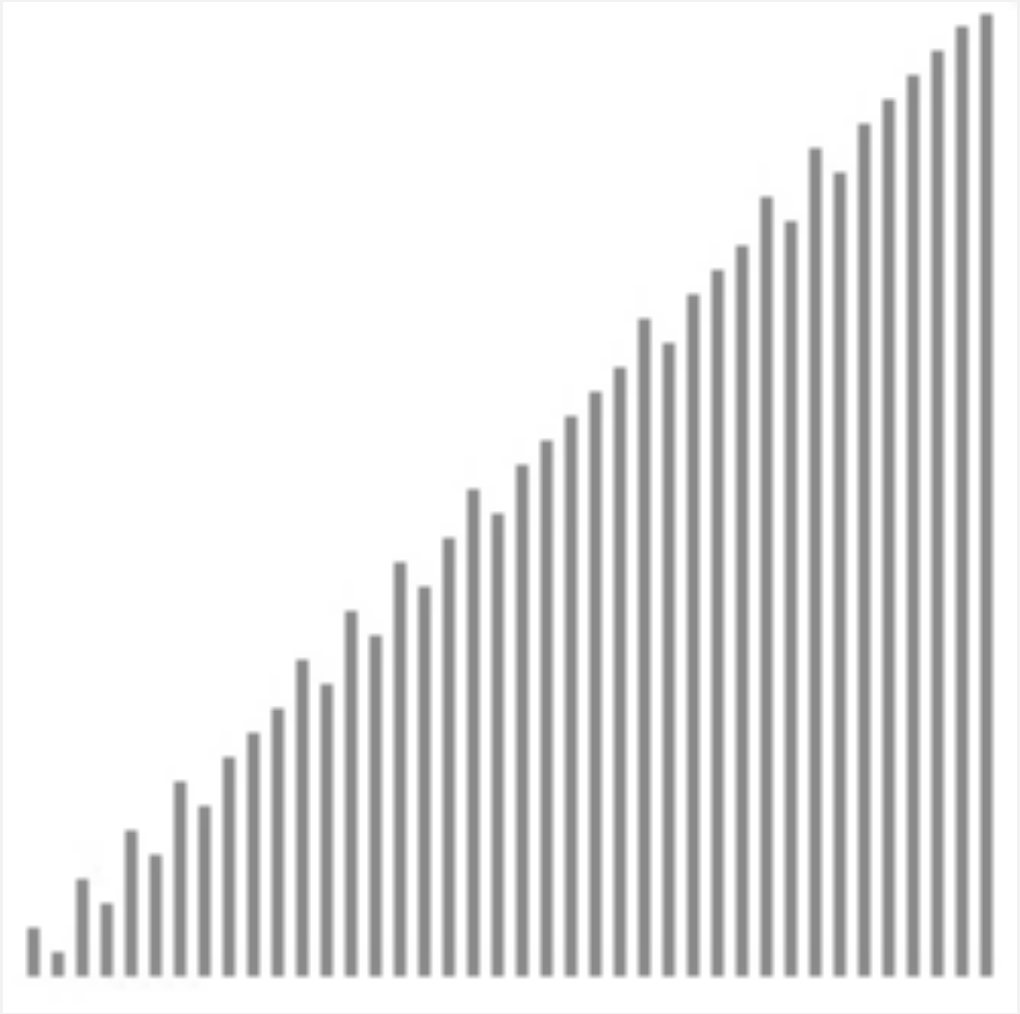
Proposition. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

number of compares = exchanges + (N – 1)

Insertion sort: animation

40 partially-sorted items



algorithm position

in order

not yet seen

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

Insertion sort: summary

Comparison sort?	Comparison
Time Complexity	$O(N^2)$
Space Complexity	In place
Internal or External?	Internal
Recursive / Non-recursive?	Non-recursive

algorithm	best	average	worst
selection sort	N^2	N^2	N^2
insertion sort	N	N^2	N^2

Shell Sort

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

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

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 example: increments 7, 3, 1

input

S O R T E X A M P L E

7-sort

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

3-sort

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

1-sort

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

result

A E E L M O P R S T X

Shellsort: which increment sequence to use?

Knuth Sequence

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

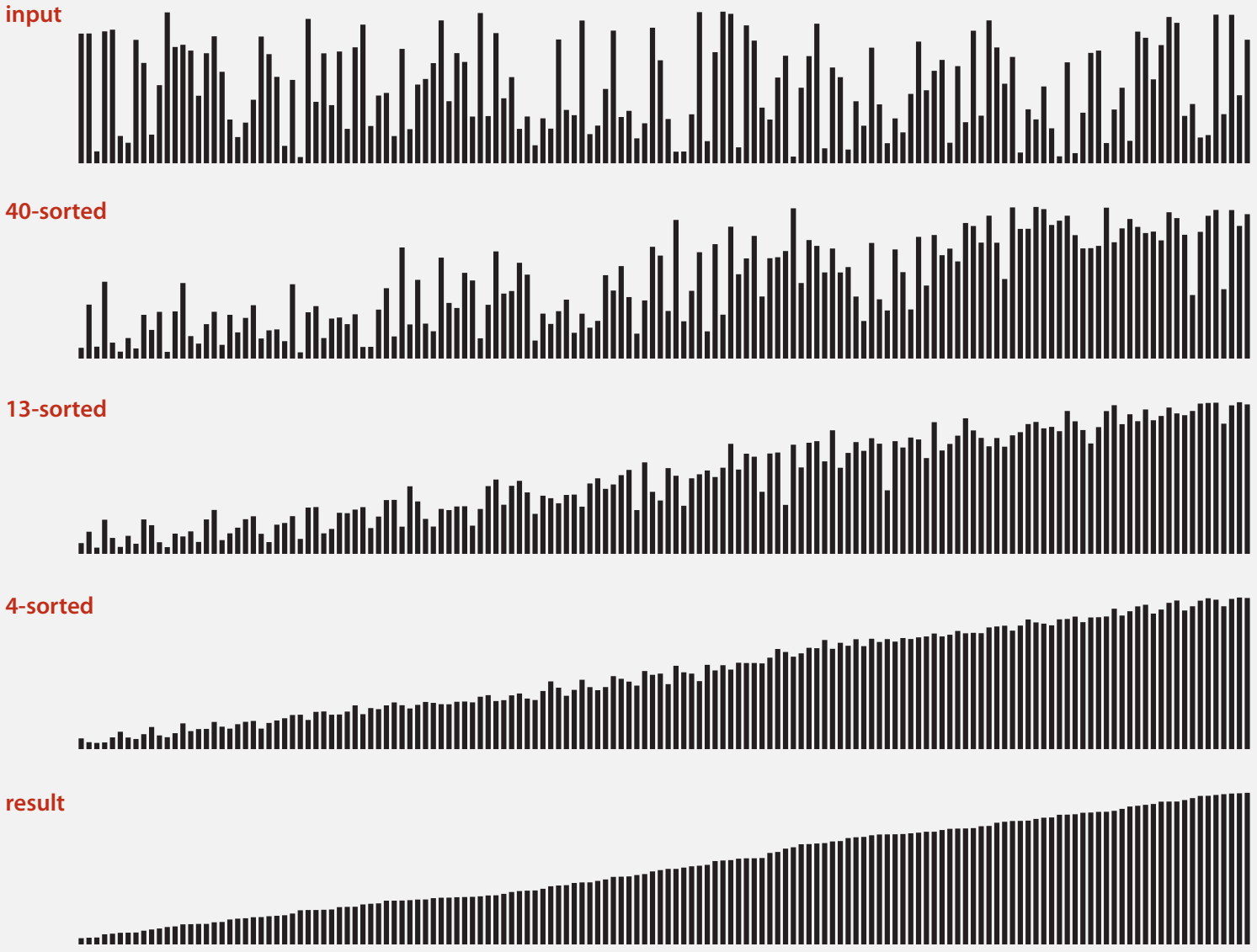
OK. Easy to compute.

Sedgewick Sequence

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

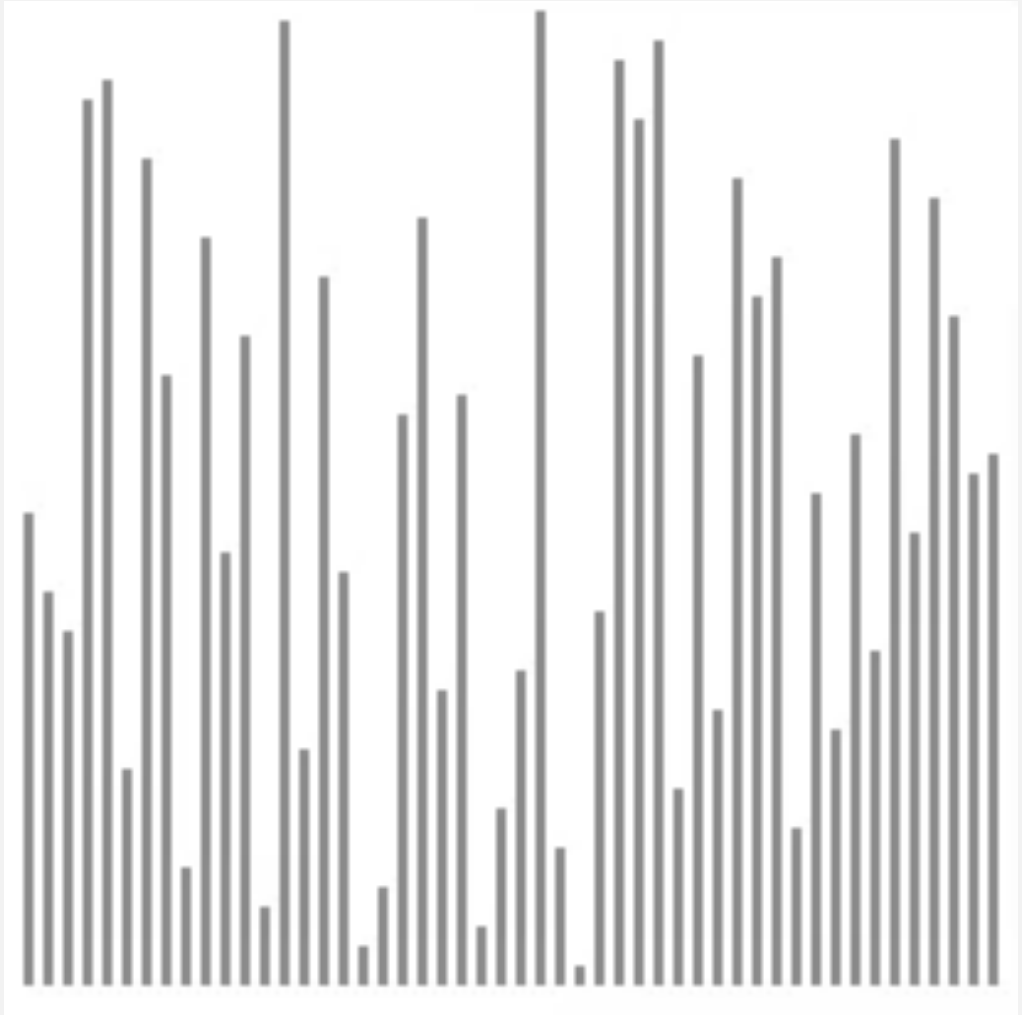
Good. Tough to beat in empirical studies.

Shellsort: visual trace

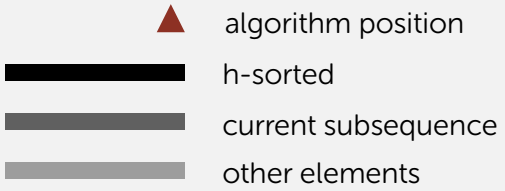


Shellsort: animation

50 random items

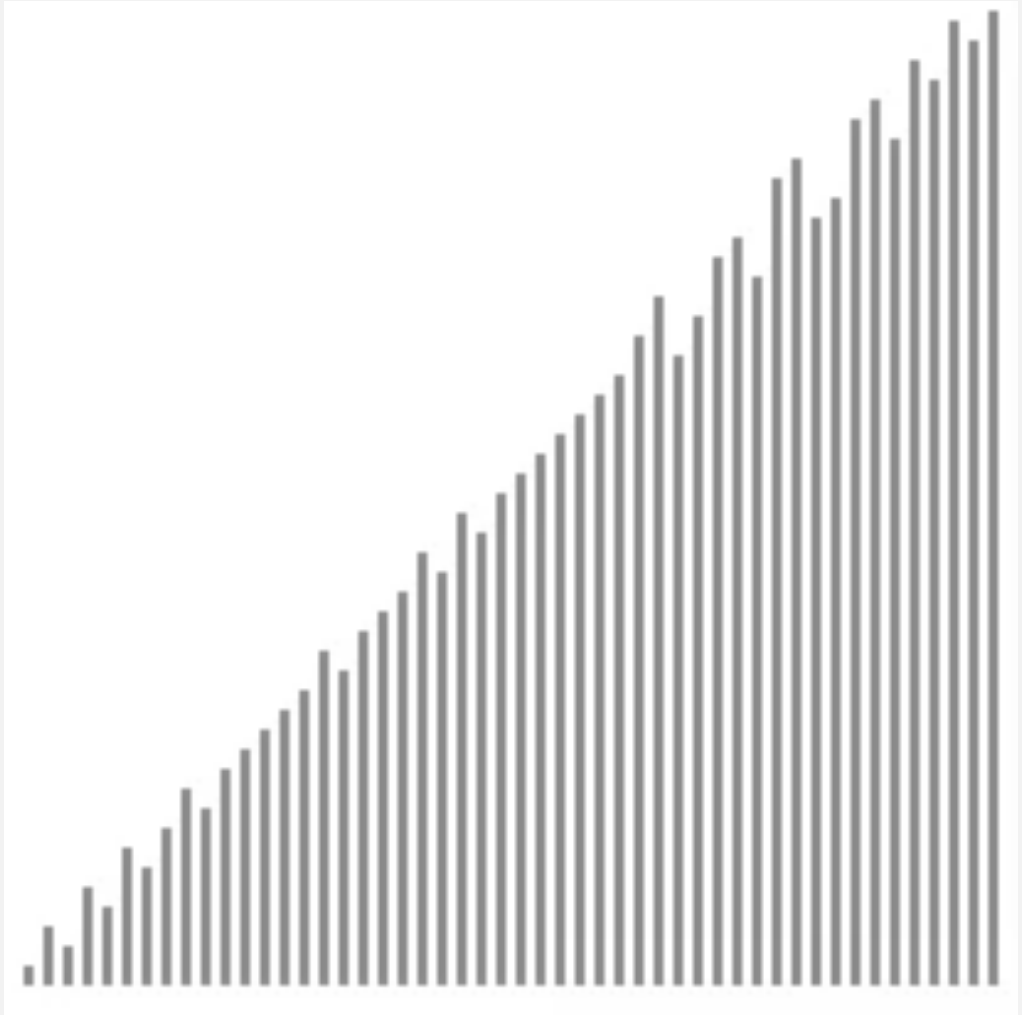


http://www.sorting-algorithms.com/shell-sort



Shellsort: animation

50 partially-sorted items



algorithm position

h-sorted

current subsequence

other elements

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.

Elementary sorts summary

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

Next week. $N \log N$ sorting algorithms (in worst case).

Shuffle Sorts

War story (online poker)

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

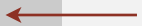
How We Learned to Cheat at Online Poker: A Study in Software Security

http://www.datamation.com/entdev/article.php/616221

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 $\Rightarrow$ 52^{nd} card can't end up in 52^{nd} place.

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

Bug 3. random() uses 32-bit seed $\Rightarrow$ 2^{32} possible shuffles.

Bug 4. Seed = milliseconds since midnight $\Rightarrow$ 86.4 million shuffles.

" The generation of random numbers is too important to be left to chance. "

— Robert R. Coveyou

Related Links

- HOLDEM POKER
- ONLINE CASH GAMES
- 365 CASINO
- CASINO DEPOSIT BONUS
- PLAY POKER ONLINE
- FREE MONEY ONLINE
- PLAY CASINO ONLINE
- FREE CASH
- ONLINE CASINO
- CASINO GAMES

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!

War story (Microsoft)

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

http://www.browserchoice.eu

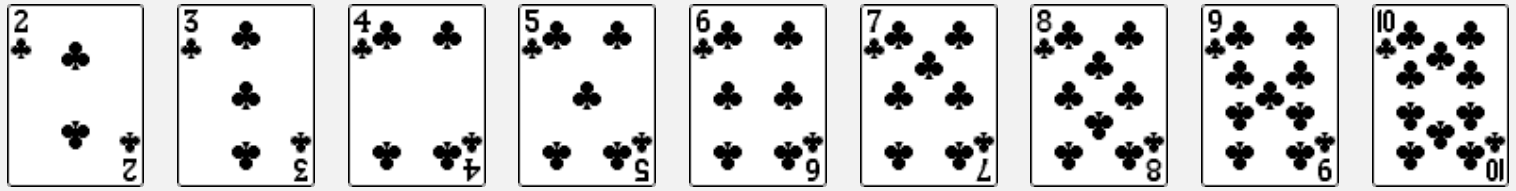
appeared last

50% of the time

How to shuffle an array

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

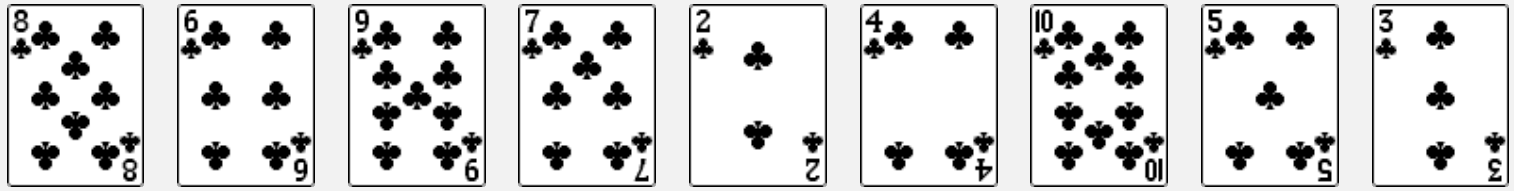
all permutations
equally likely



How to shuffle an array

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

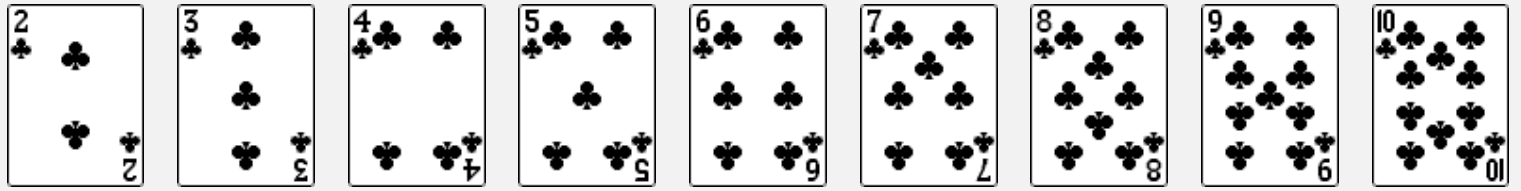
all permutations
equally likely



Shuffle sort

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

useful for shuffling columns in a spreadsheet

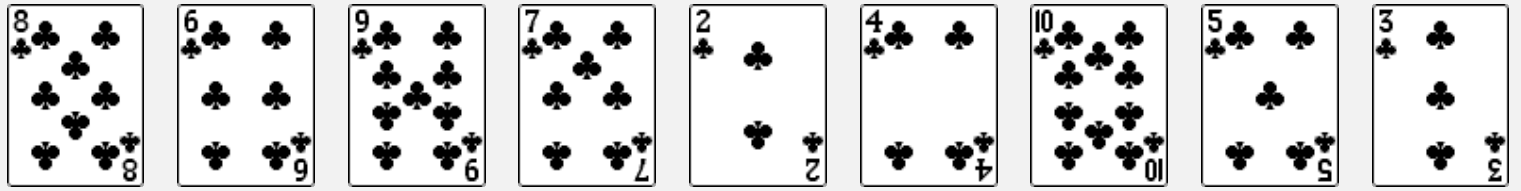


0.8003 0.9706 0.9157 0.9649 0.1576 0.4854 0.1419 0.4218 0.9572

Shuffle sort

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

useful for shuffling columns in a spreadsheet

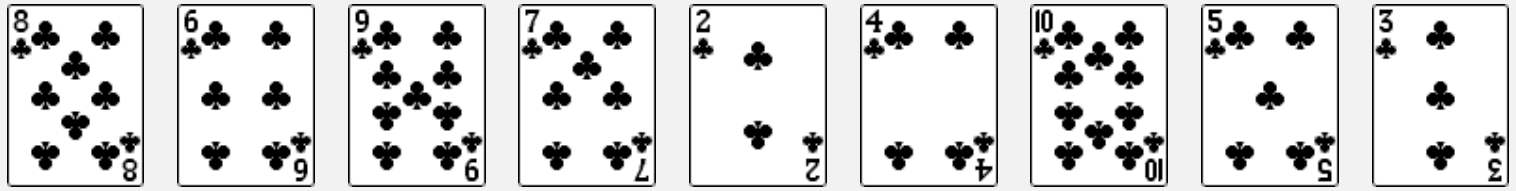


0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Shuffle sort

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

useful for shuffling columns in a spreadsheet



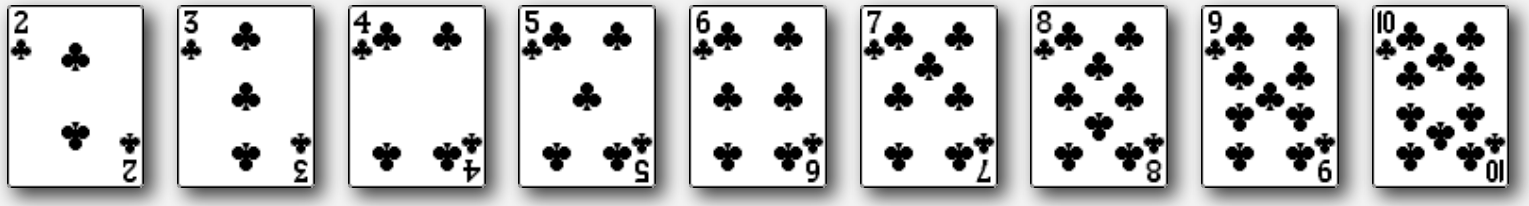
0.1419 0.1576 0.4218 0.4854 0.8003 0.9157 0.9572 0.9649 0.9706

Proposition. Shuffle sort produces a uniformly random permutation.

Drawback? We have to pay the performance cost of sorting.

Knuth shuffle

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

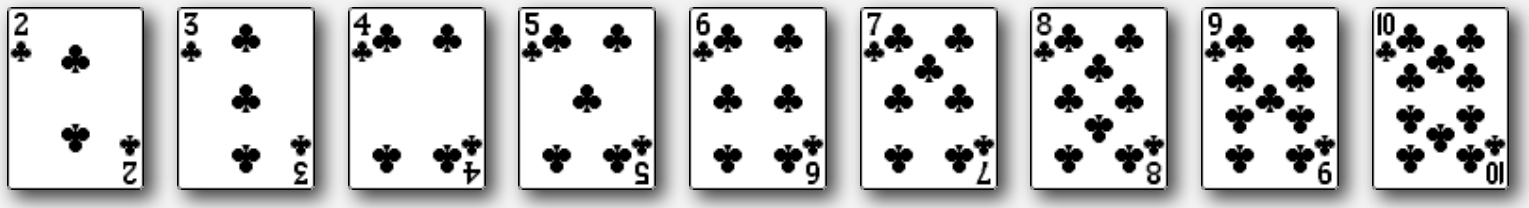


Proposition. [Fisher-Yates 1938] Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time.

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].



Knuth shuffle

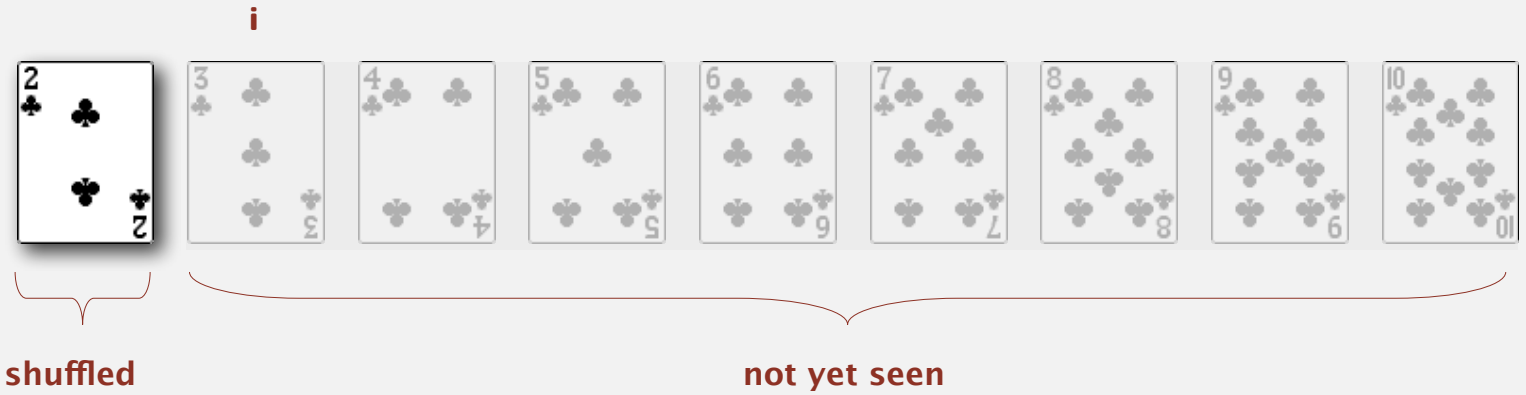
- In iteration i, pick integer r between 0 and i uniformly at random.
- Swap a[i] and a[r].

Knuth shuffle

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

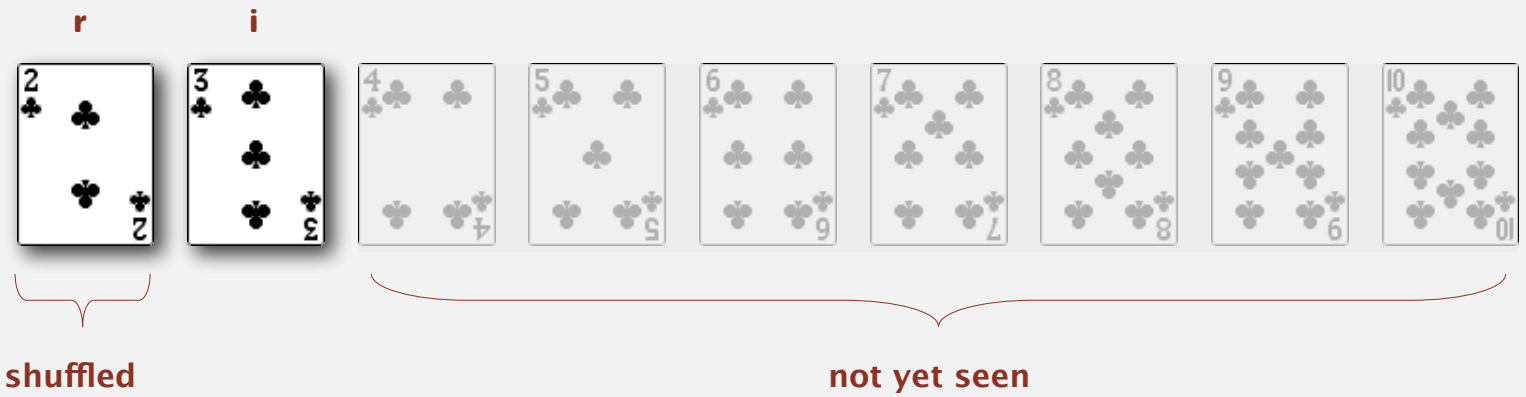
Knuth shuffle

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



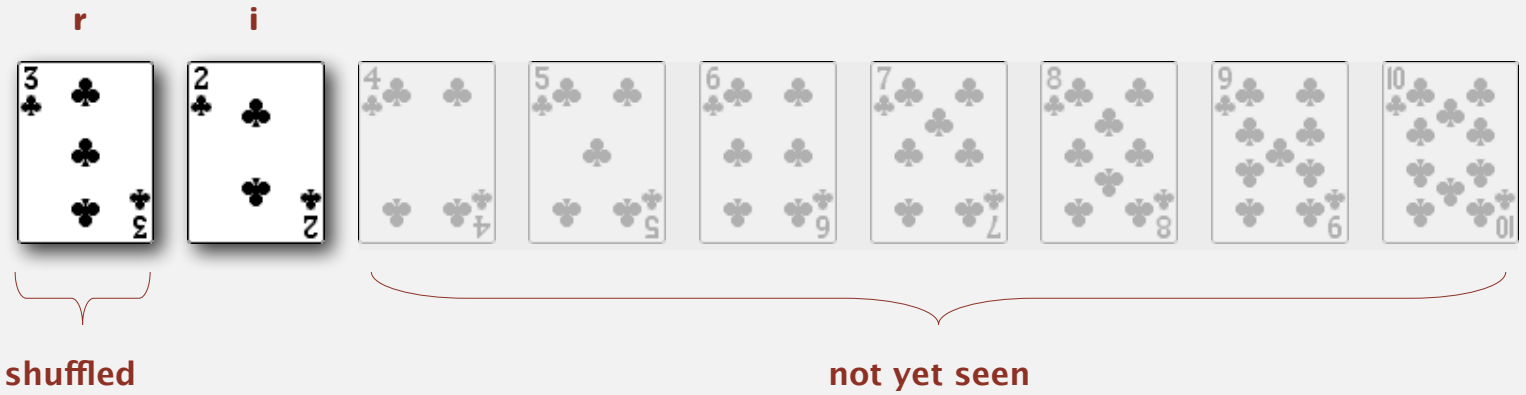
Knuth shuffle

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



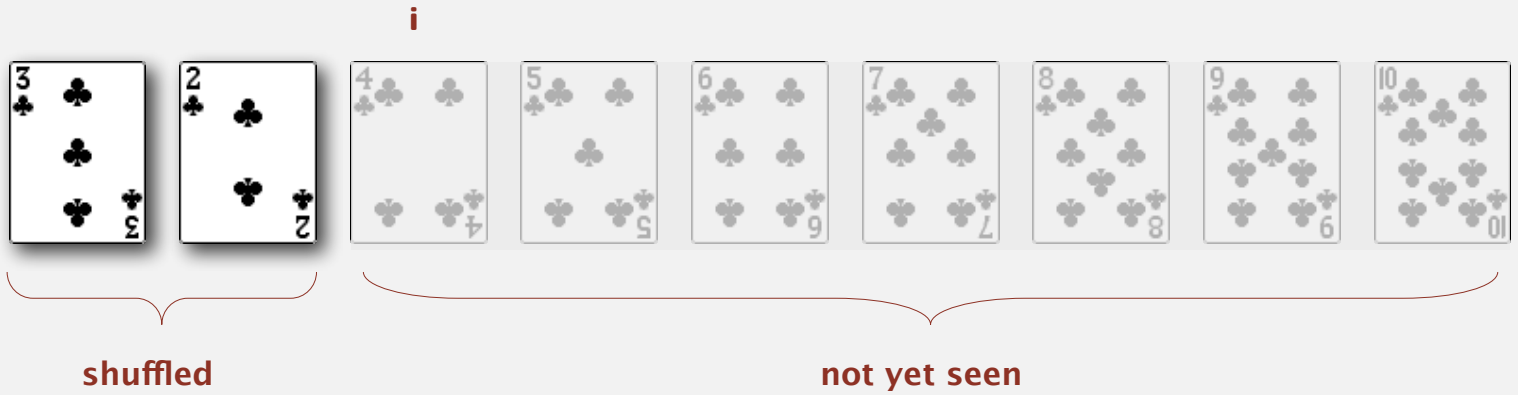
Knuth shuffle

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



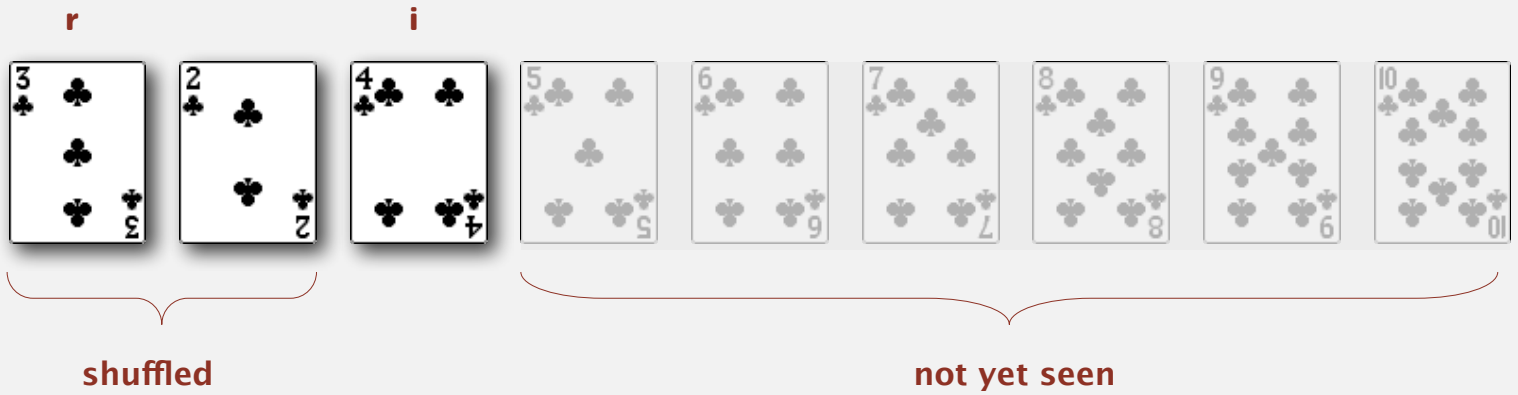
Knuth shuffle

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



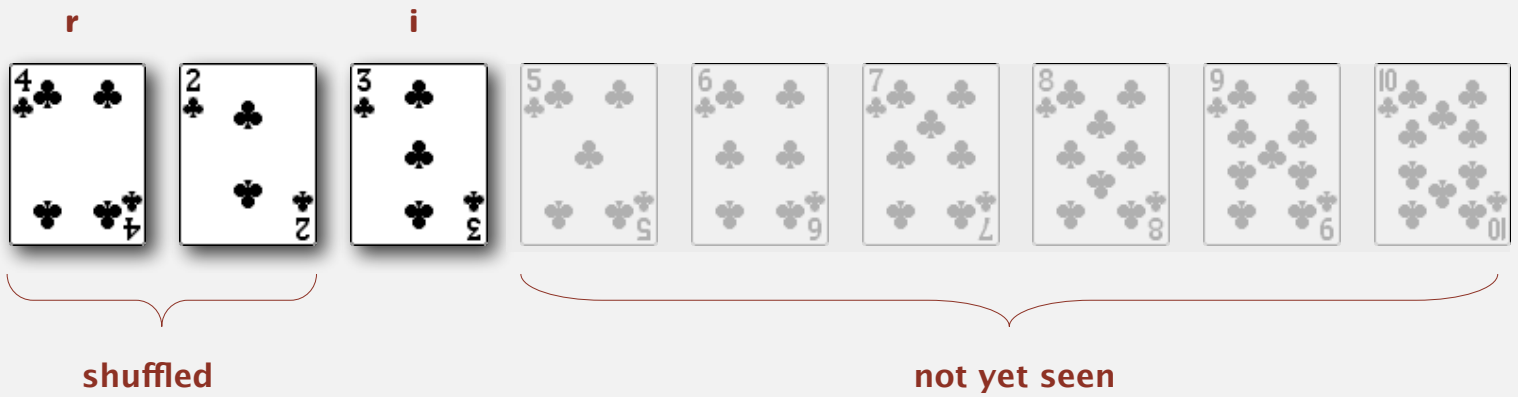
Knuth shuffle

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



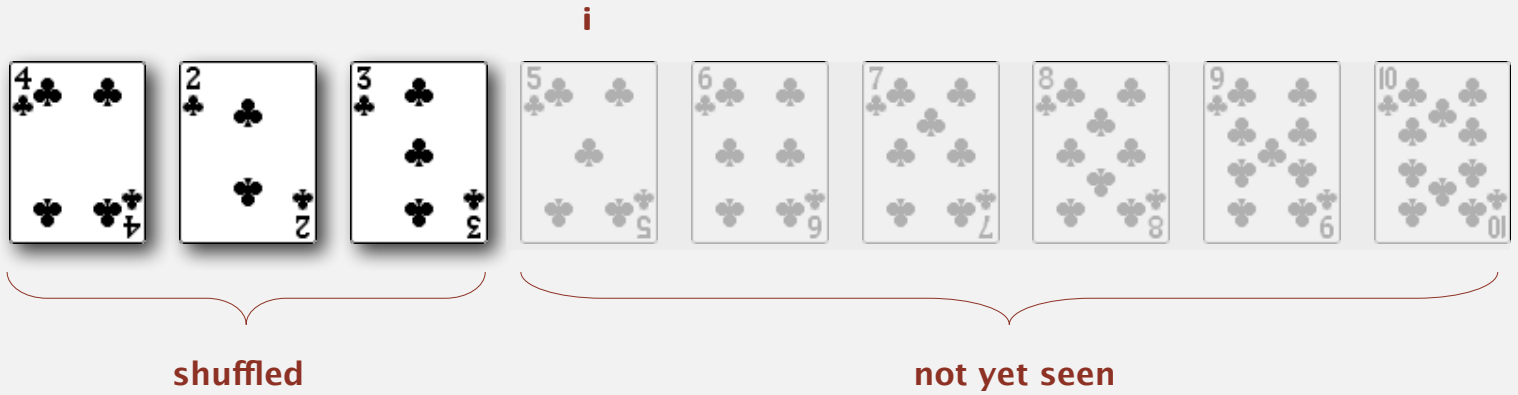
Knuth shuffle

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



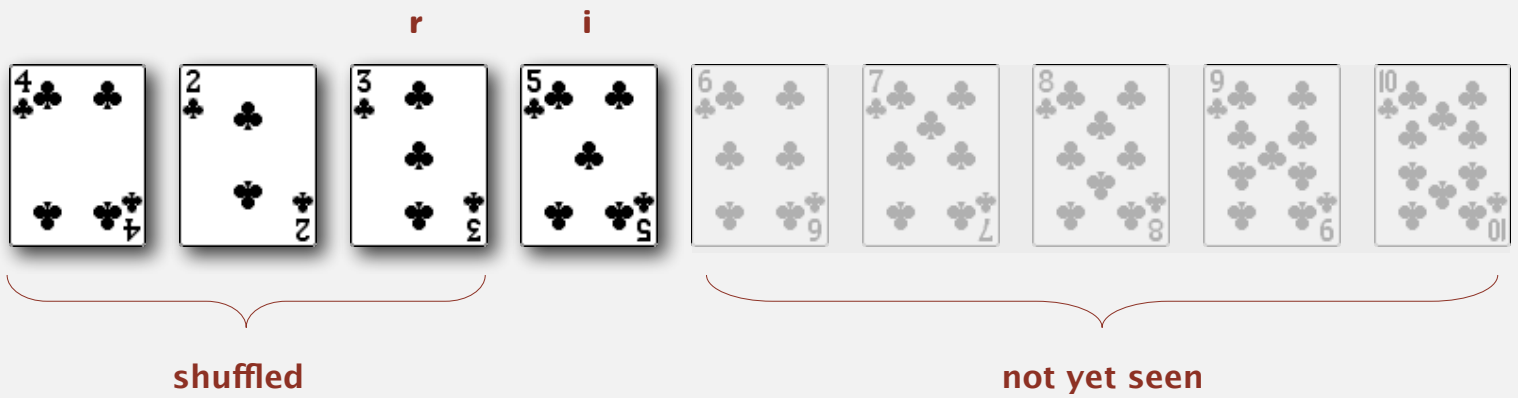
Knuth shuffle

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



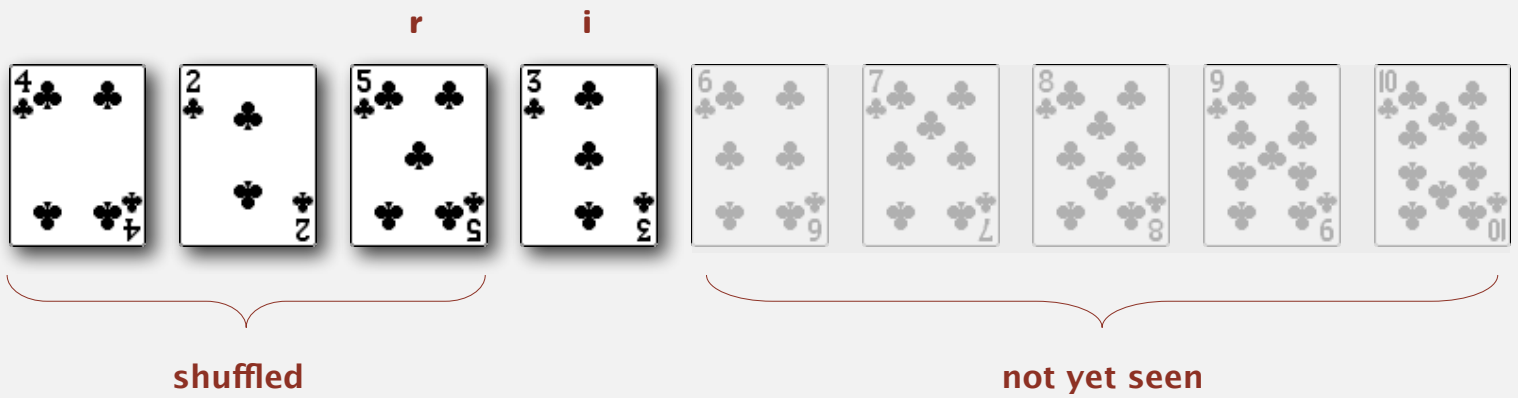
Knuth shuffle

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



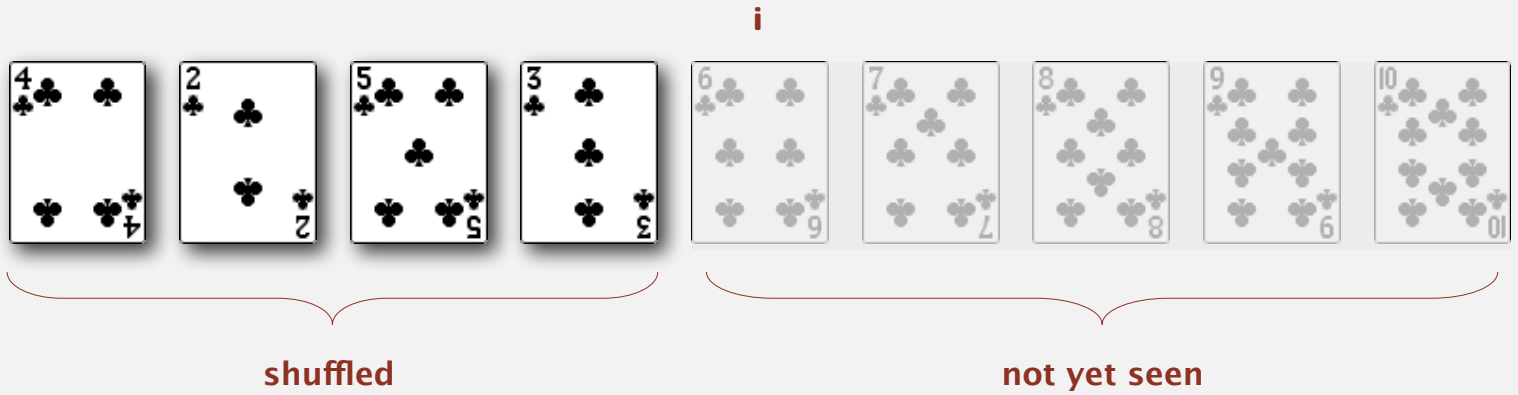
Knuth shuffle

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



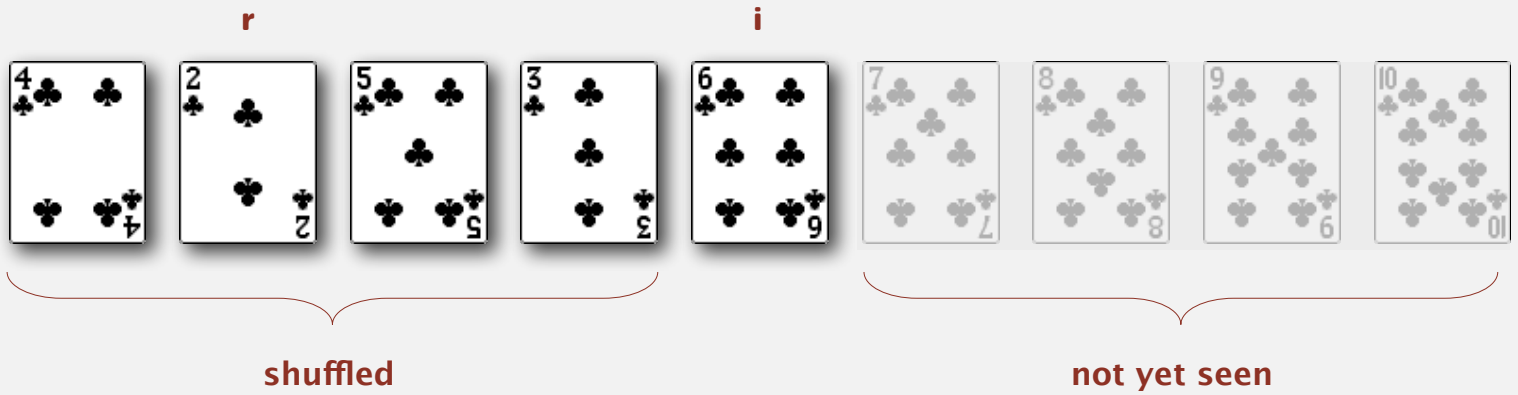
Knuth shuffle

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



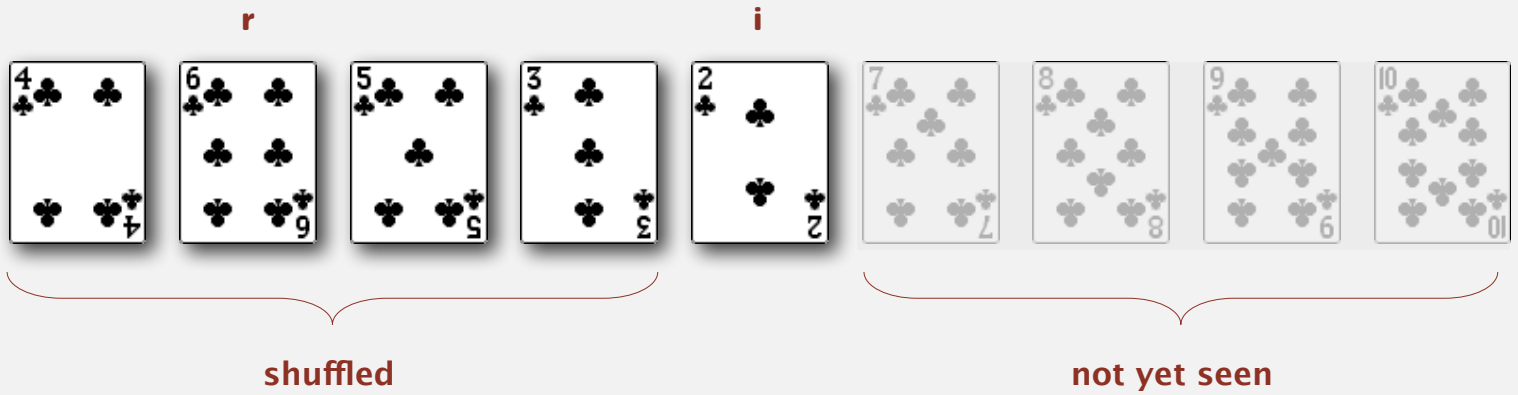
Knuth shuffle

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



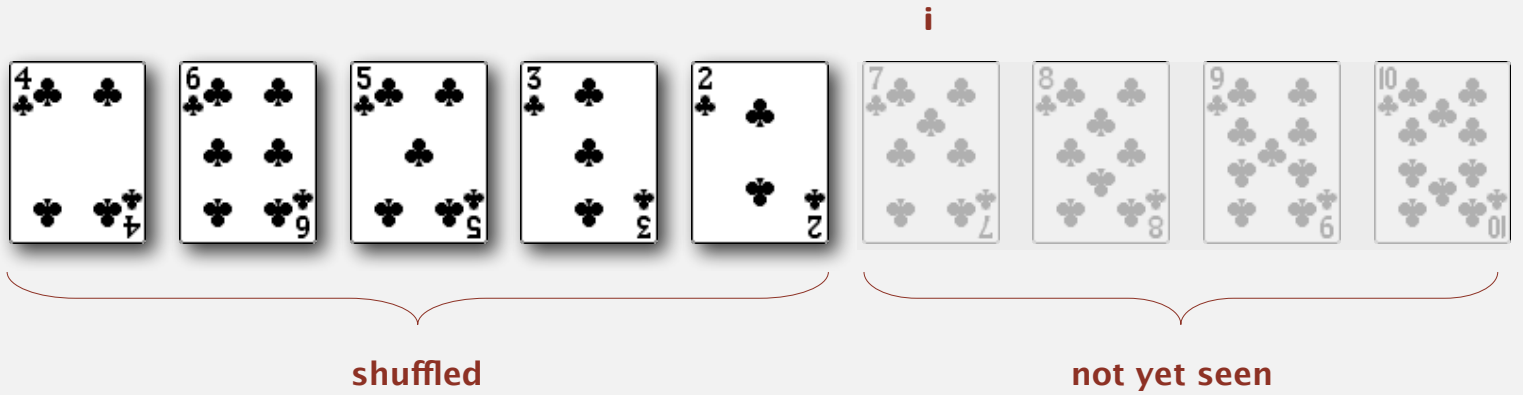
Knuth shuffle

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



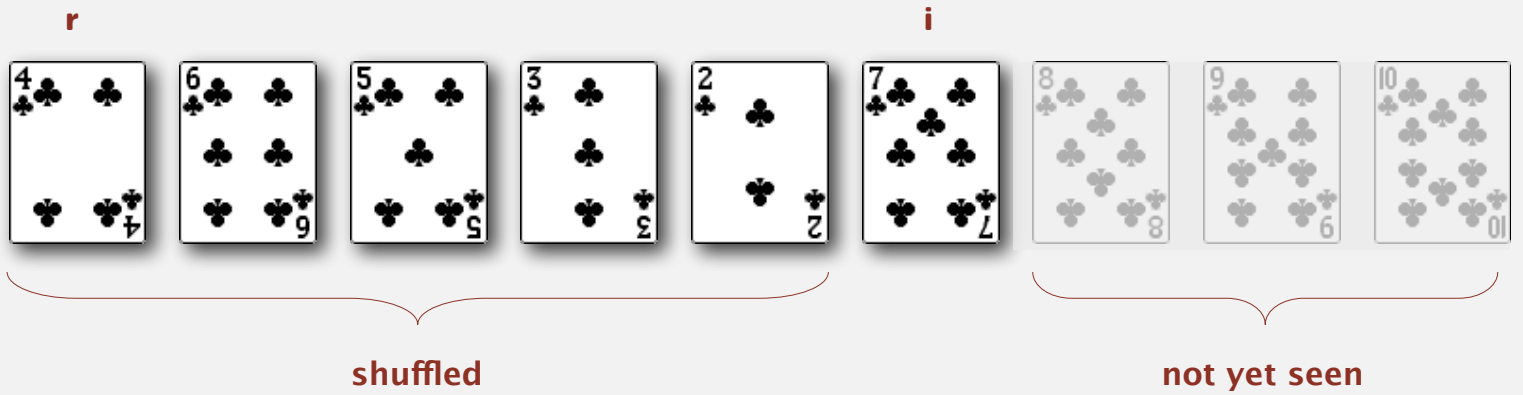
Knuth shuffle

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



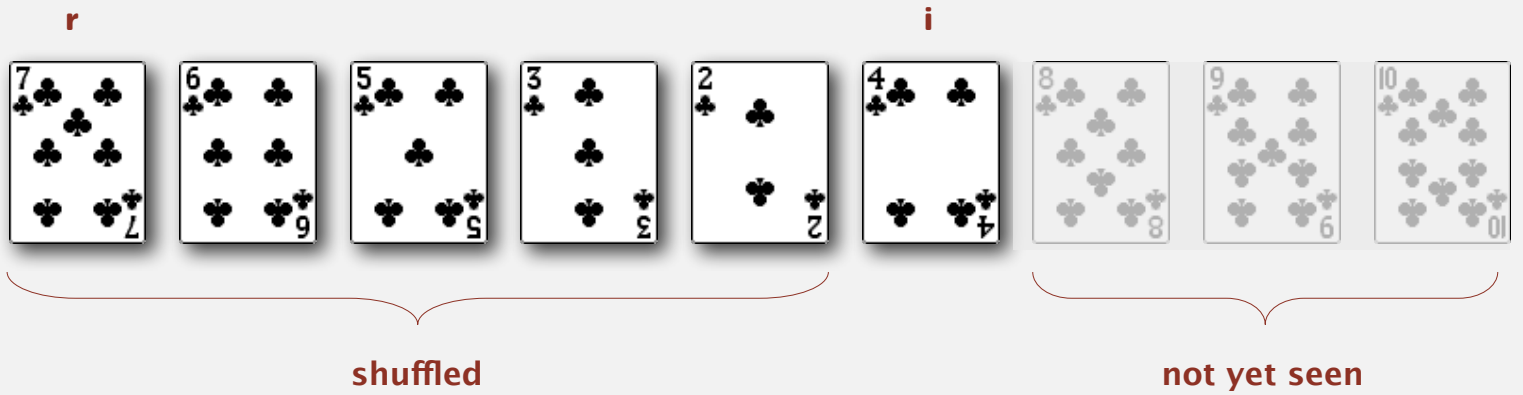
Knuth shuffle

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



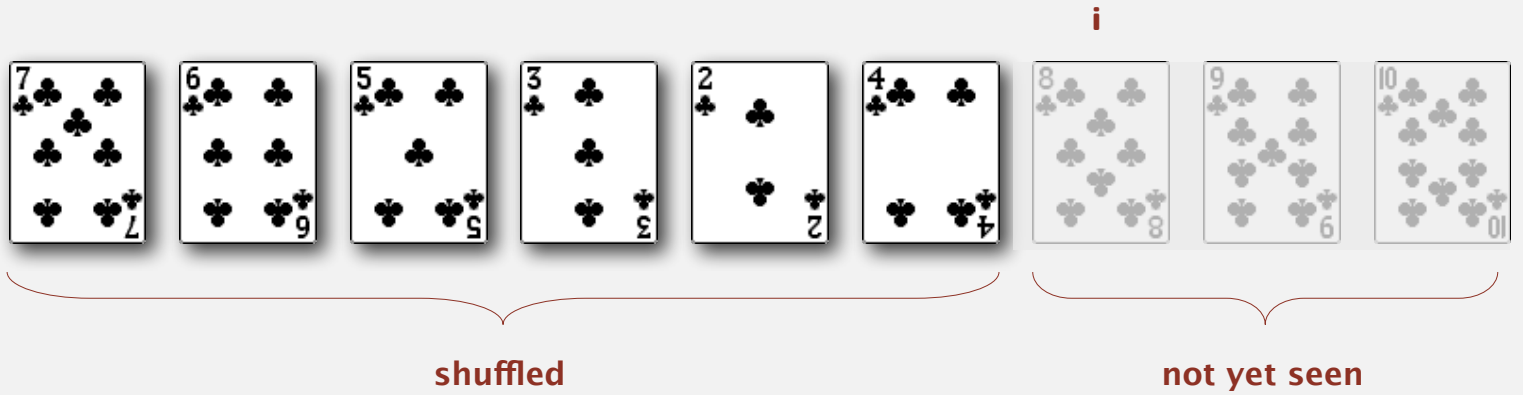
Knuth shuffle

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



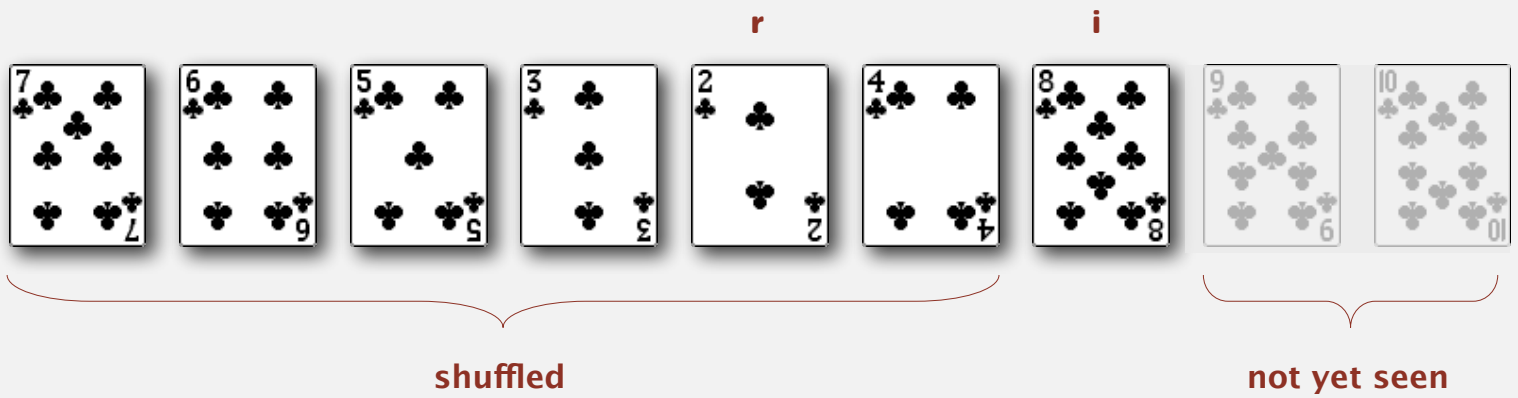
Knuth shuffle

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



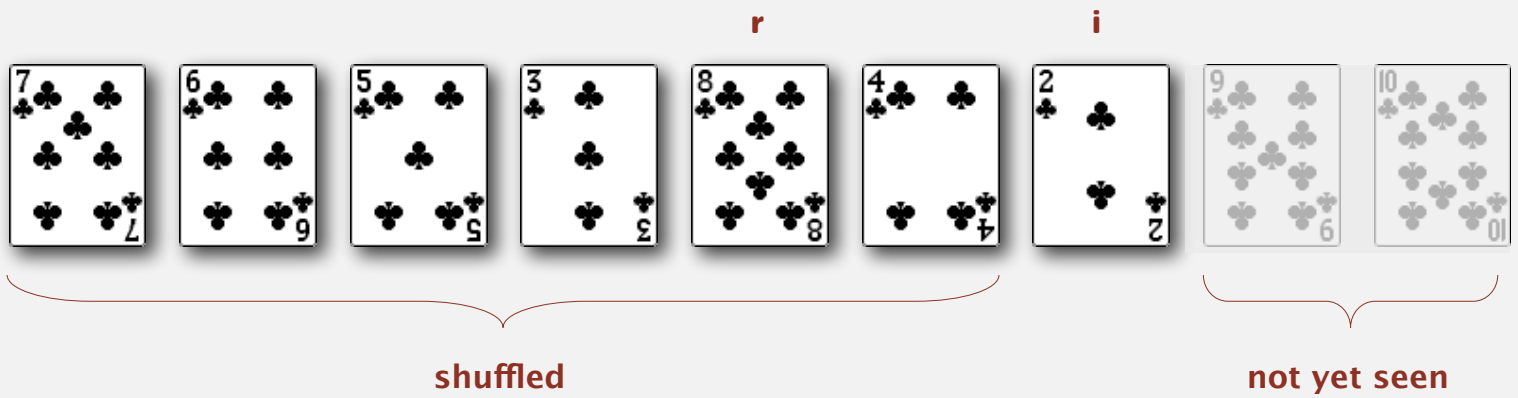
Knuth shuffle

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



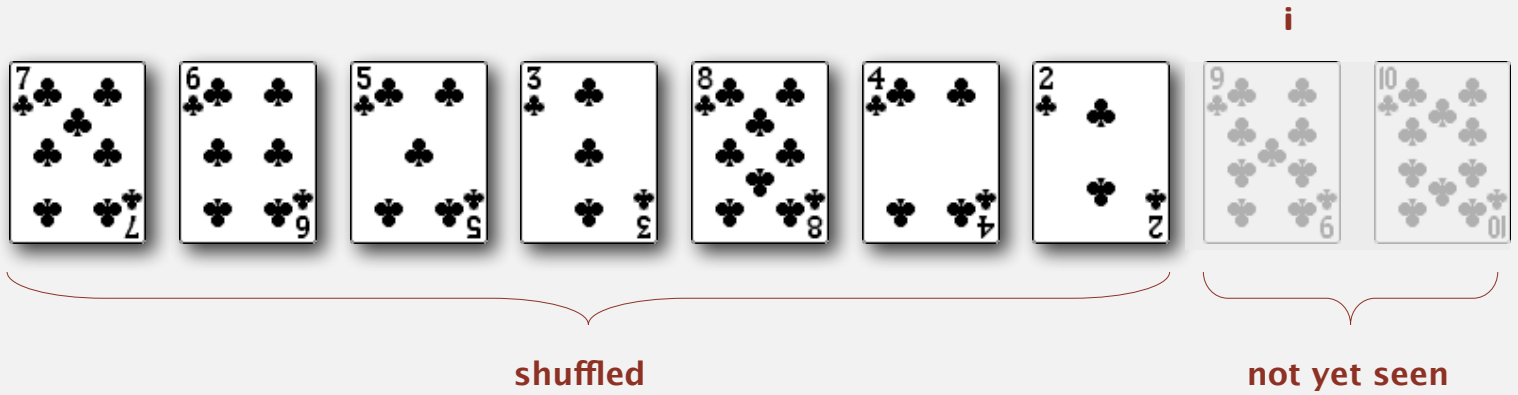
Knuth shuffle

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



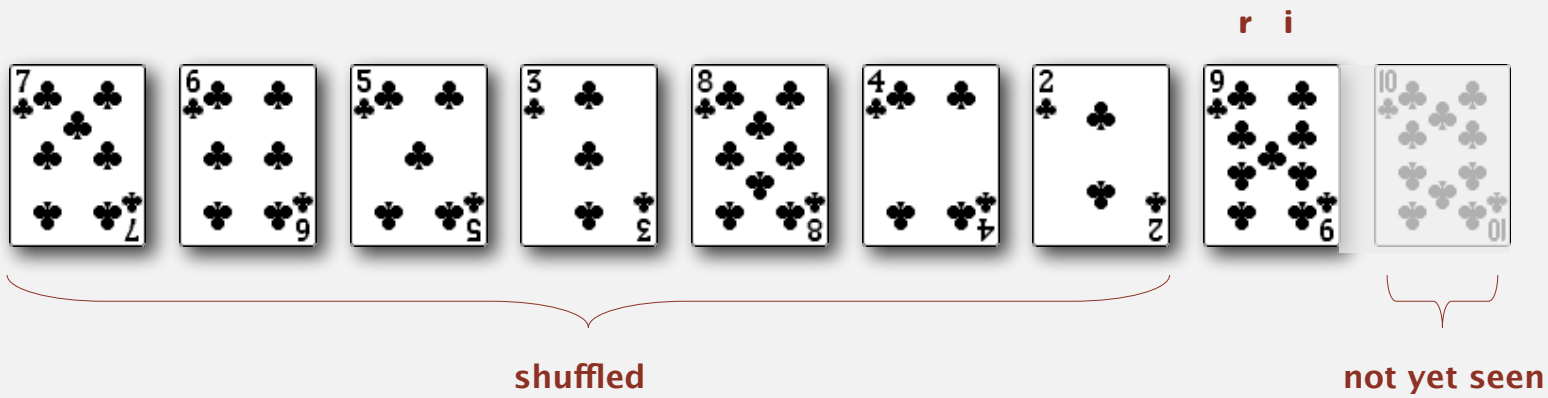
Knuth shuffle

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



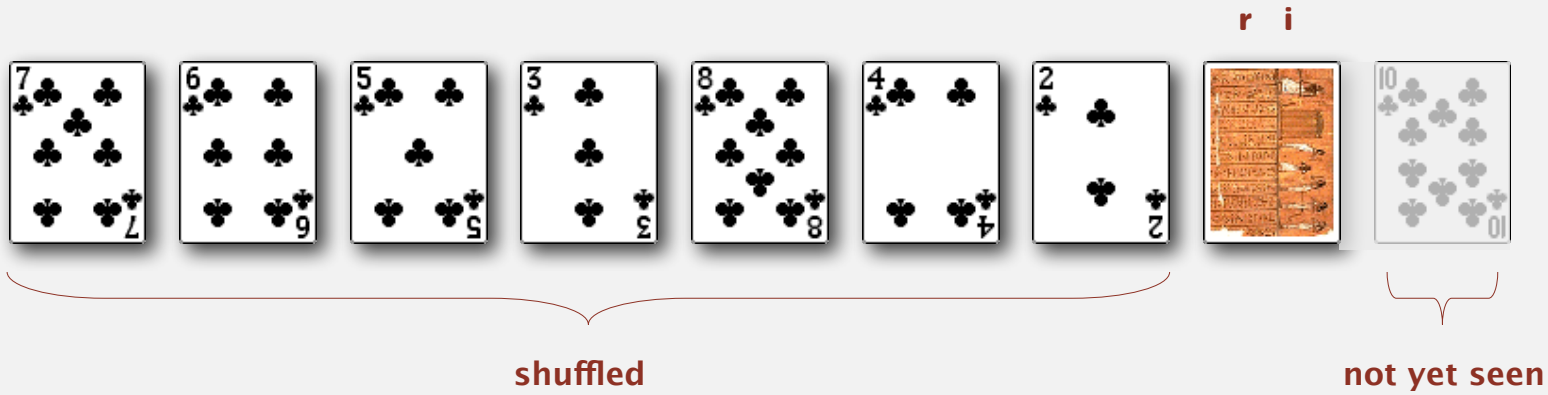
Knuth shuffle

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



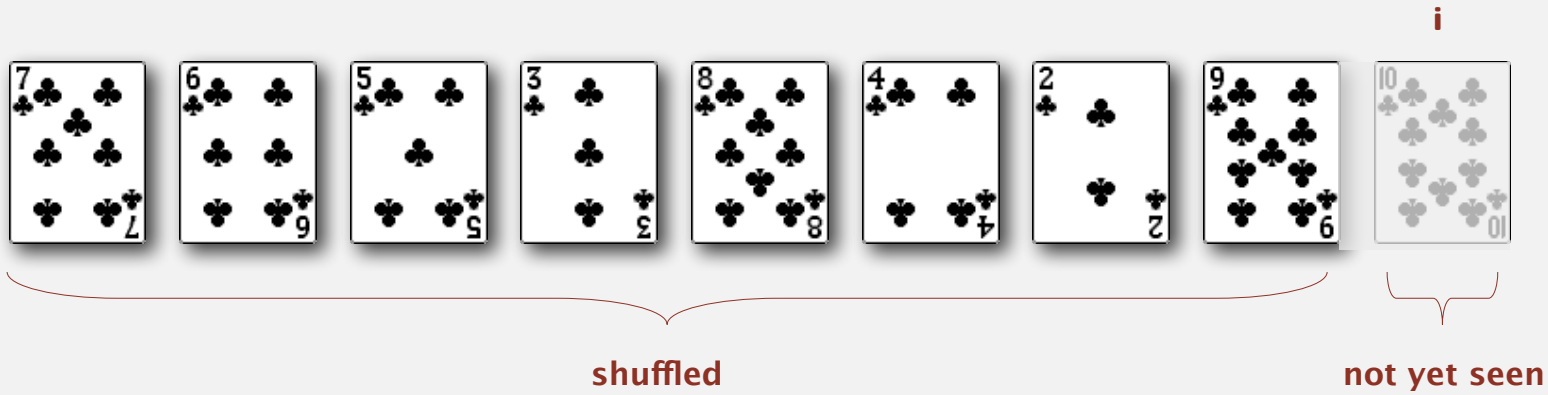
Knuth shuffle

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



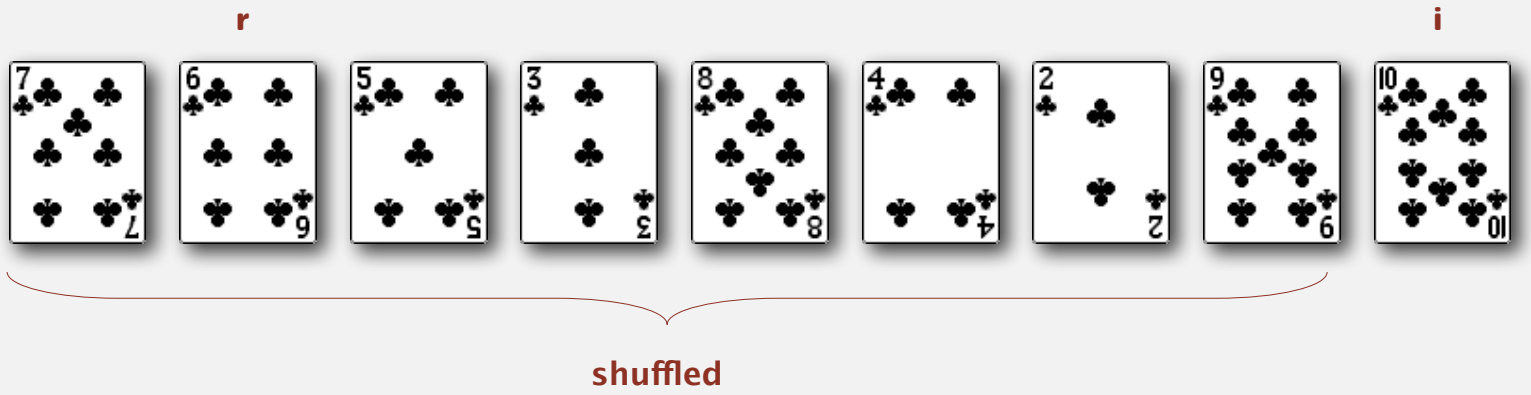
Knuth shuffle

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



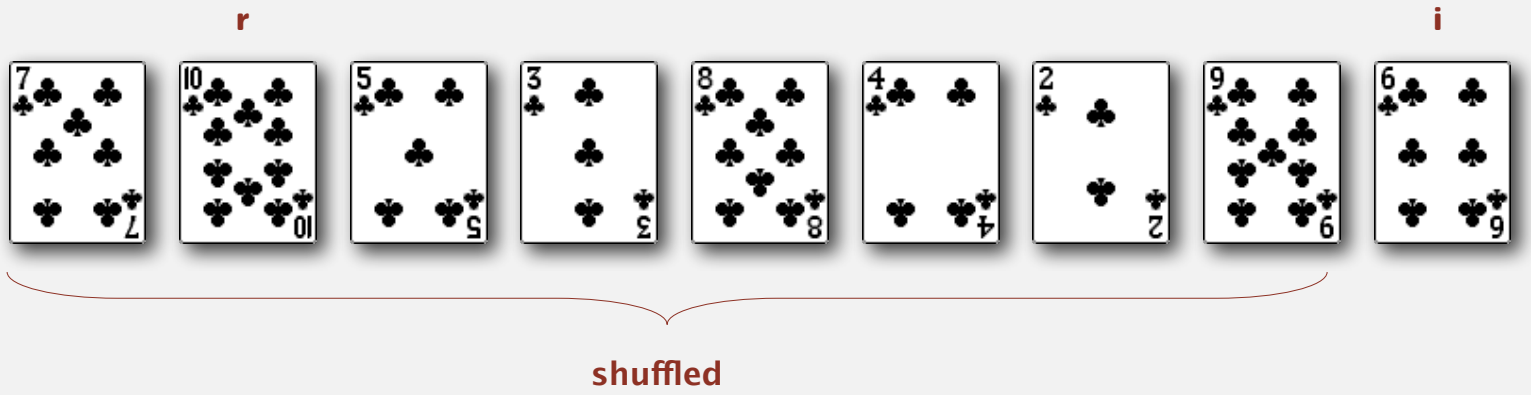
Knuth shuffle

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



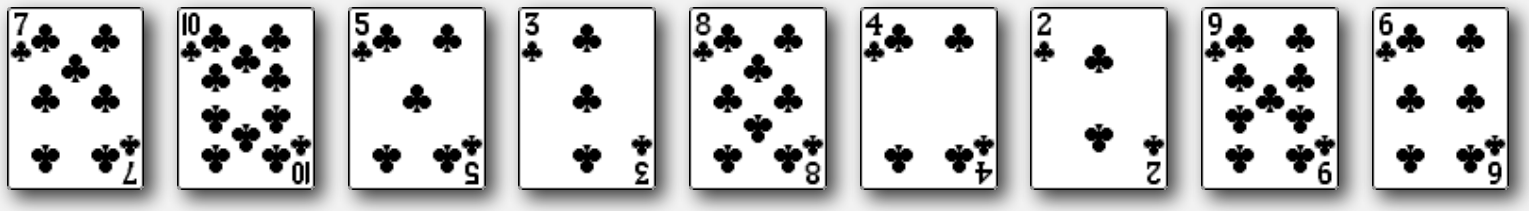
Knuth shuffle

- 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

Bogo Sort

The world's worst sorting algorithm?

BogoSort

bogo-sort: /boh'goh-sort'/ /n./ (var. 'stupid-sort') The archetypical perversely awful algorithm (as opposed to bubble sort, which is merely the generic *bad* algorithm).

Bogo-sort is equivalent to repeatedly throwing a deck of cards in the air, picking them up at random, and then testing whether they are in order. It serves as a sort of canonical example of awfulness.

Looking at a program and seeing a dumb algorithm, one might say "Oh, I see, this program uses bogo-sort." Compare bogus, brute force, Lasherism.

Source: The New Hacker's Dictionary https://mitpress.mit.edu/books/new-hackers-dictionary

BogoSort

```
public static void bogoSort(int[] nums) {
        while (!isSorted(nums)) {
            shuffle(nums);
        }
    }

    private static void shuffle(int[] nums) { // Knuth
Shuffle
        int n, tmp;
        for (int i = nums.length - 1; i > 0; i--) {
            n = r.nextInt(i + 1);
            tmp = nums[i];
            nums[i] = nums[n];
            nums[n] = tmp;
        }
    }
```

Time Complexity:

- Worst Case : O(infinity) (no upper bound)
- Average Case: O(n*n!)
- Best Case : O(n)(when array given is already sorted)

More

Hermann Gruber, Markus Holzer, and Oliver Ruepp, Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms, 2007.