http://algs4.cs.princeton.edu

2.1 ELEMENTARY SORTS

BASED ON SLIDES FROM ROBERT SEDGEWICK AND KEVIN WAYNE

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

Sorting problem

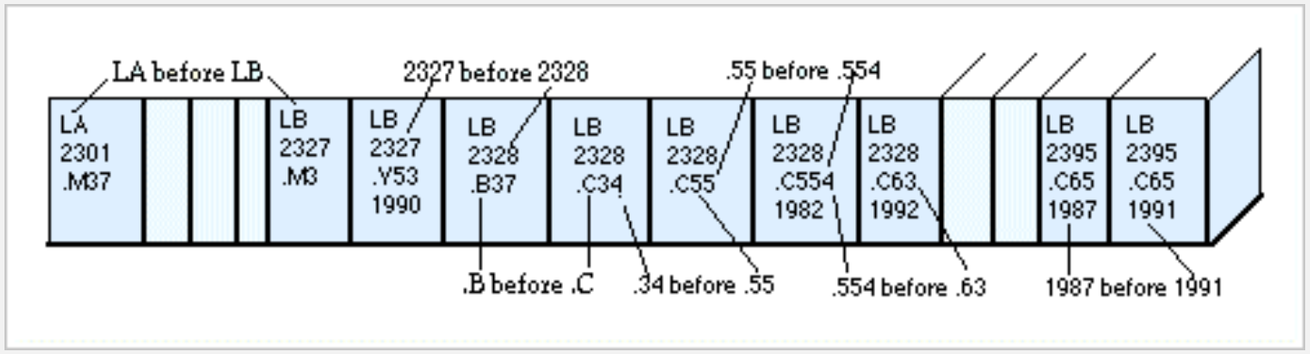
Ex. Student records in a university.

	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.

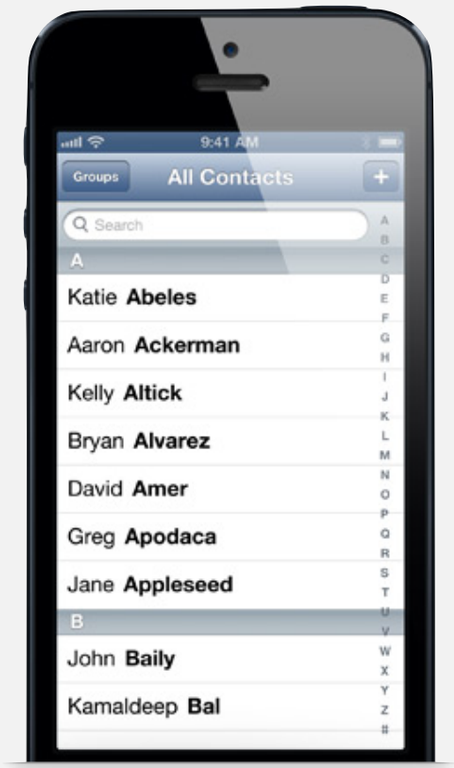
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Sorting applications

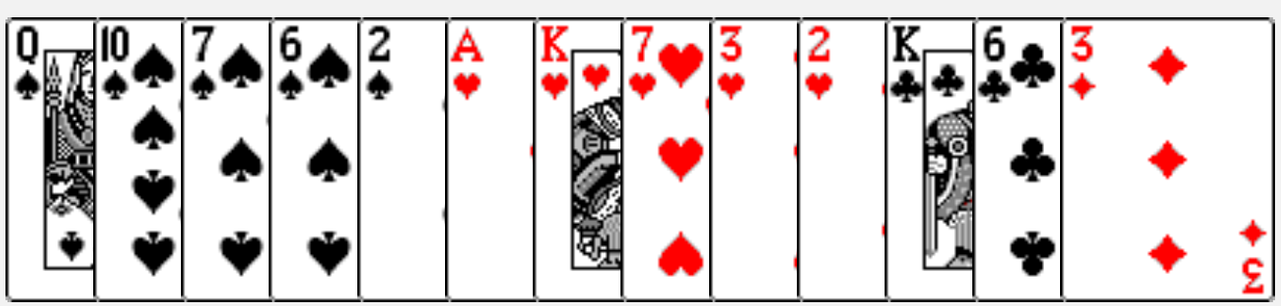


Library of Congress numbers

FedEx packages



contacts



playing cards

Hogwarts houses

http://algs4.cs.princeton.edu

2.1 Elementary Sorts

- selection sort
- insertion sort
- shellsort

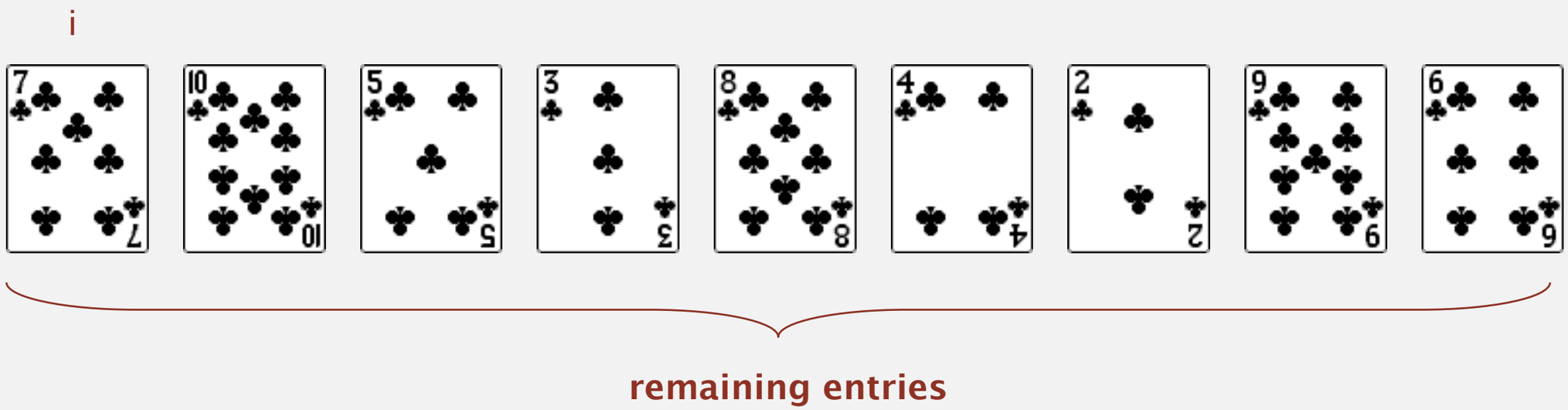
Selection sort demo

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

initial

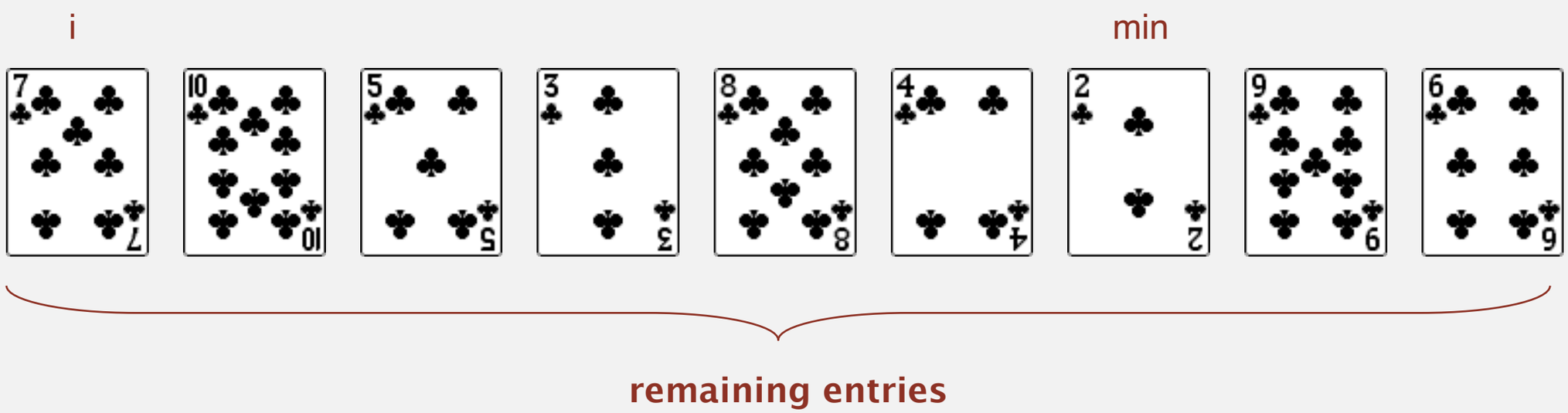
Selection sort demo

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



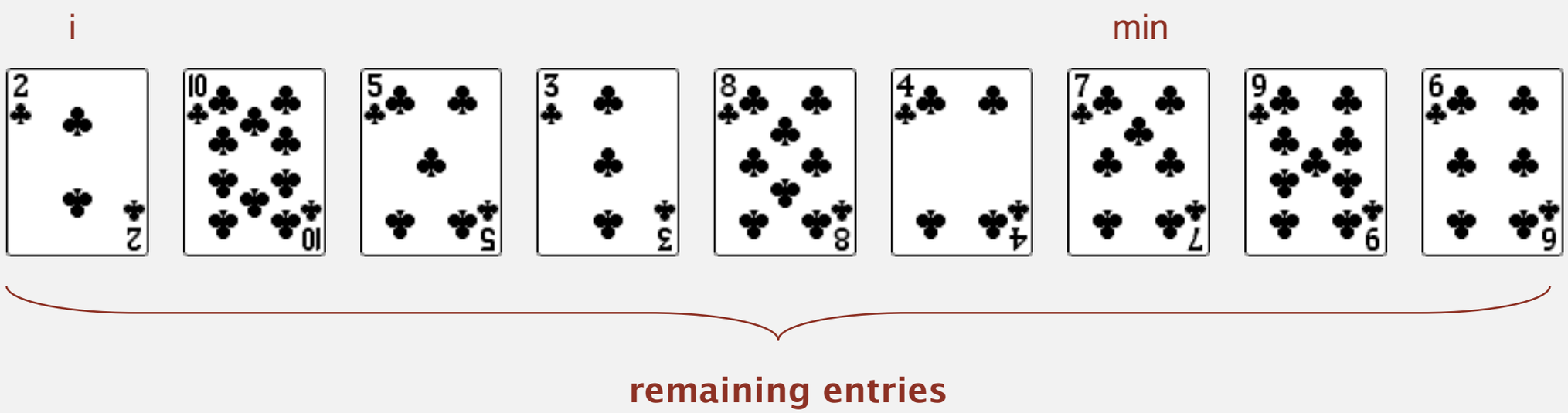
Selection sort demo

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



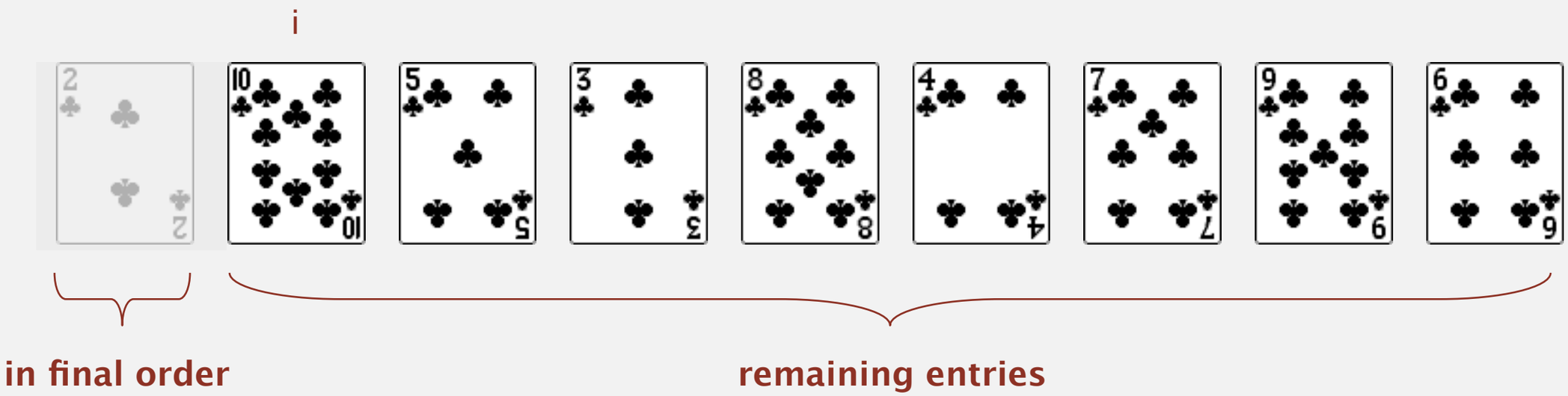
Selection sort demo

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



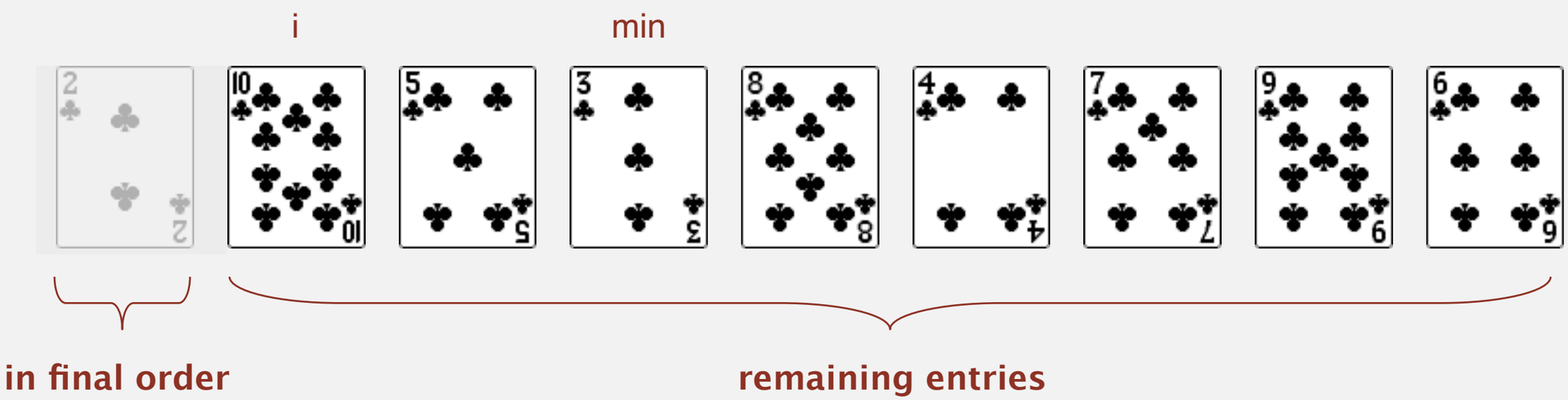
Selection sort demo

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



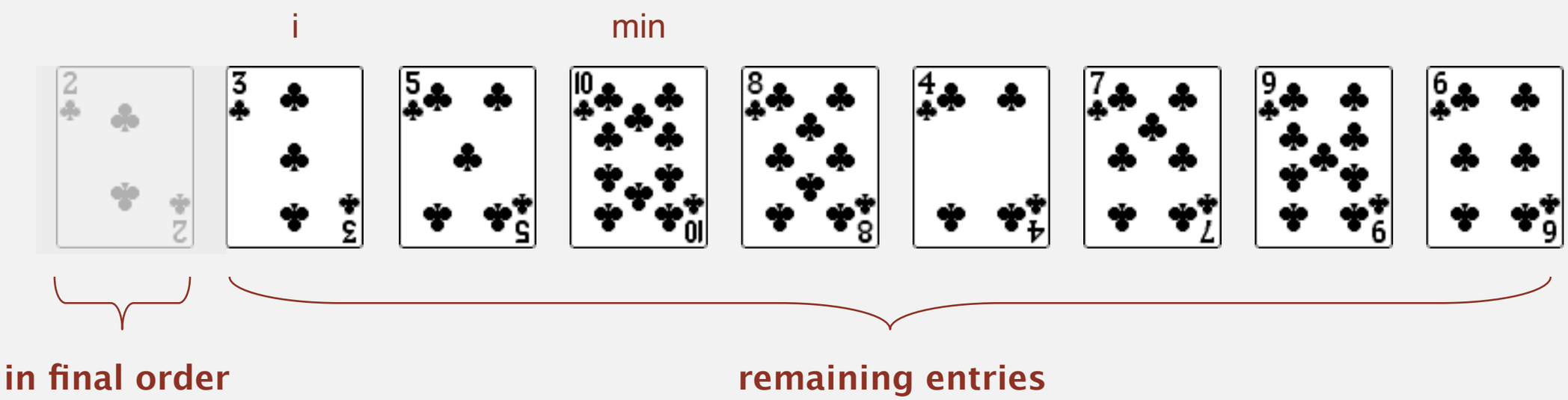
Selection sort demo

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



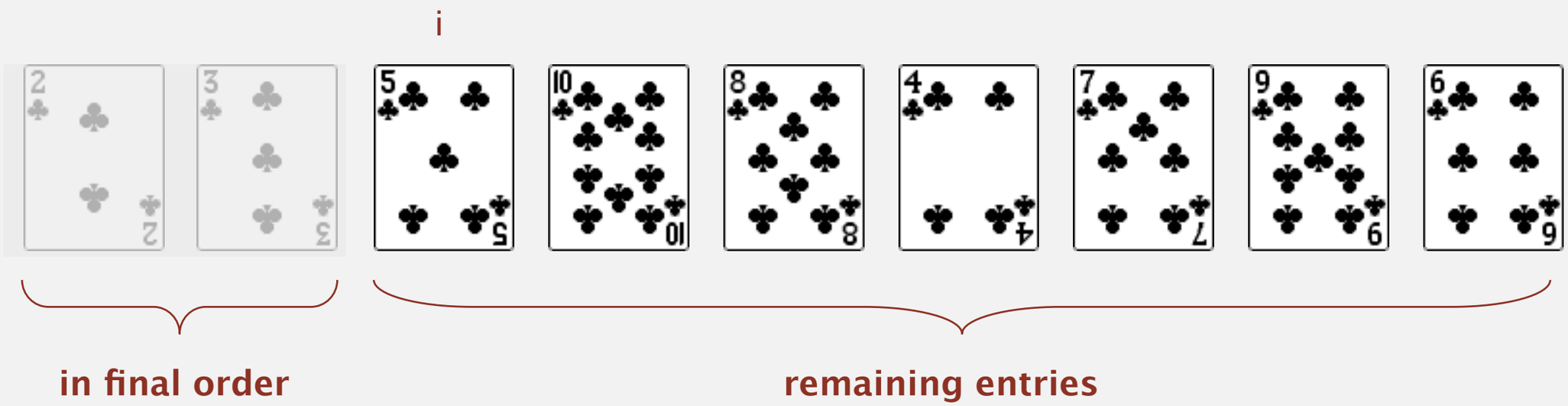
Selection sort demo

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



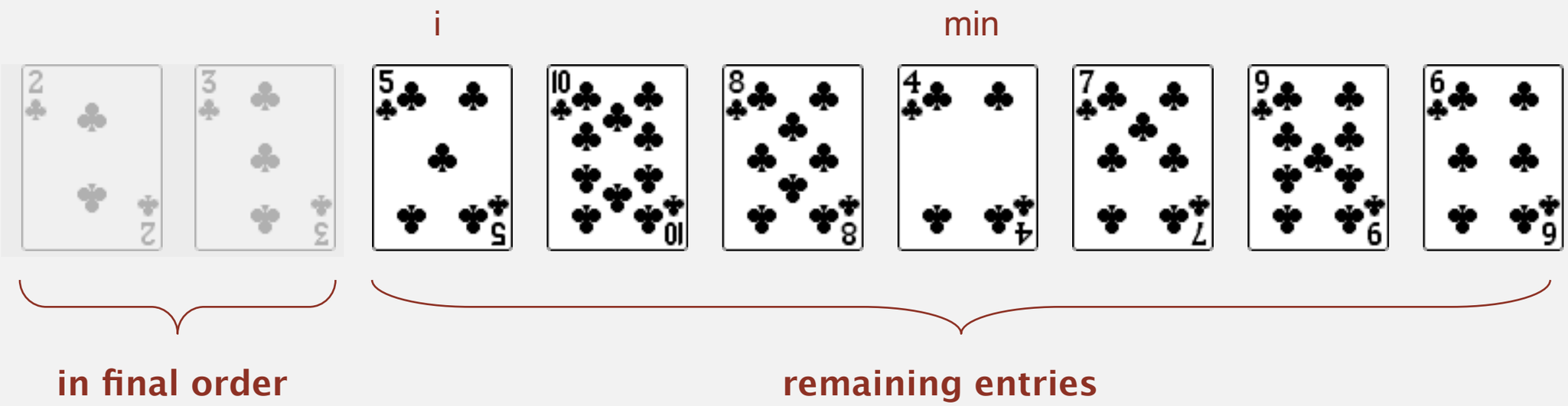
Selection sort demo

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



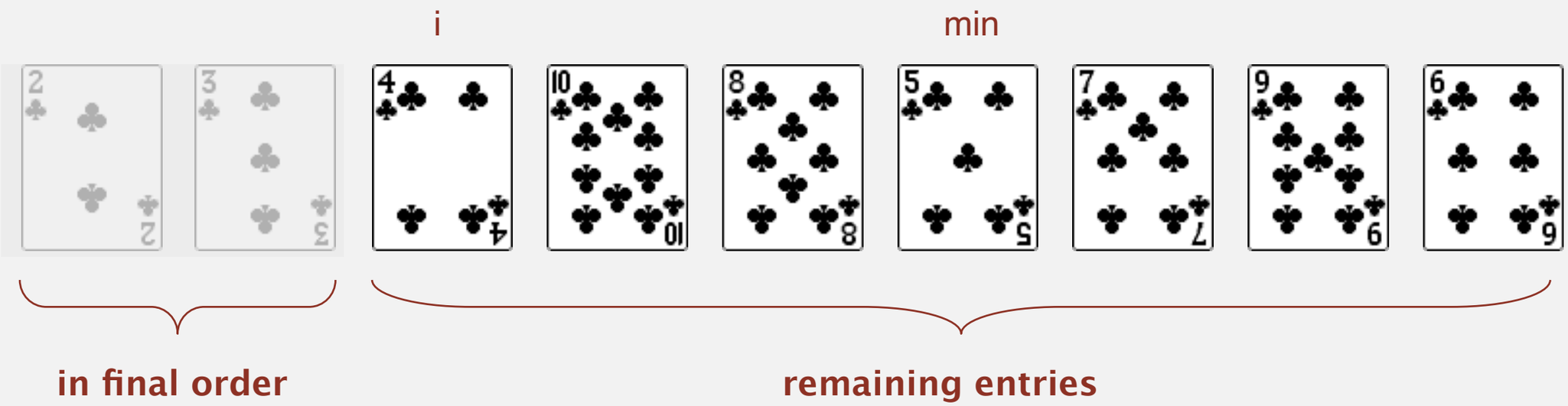
Selection sort demo

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



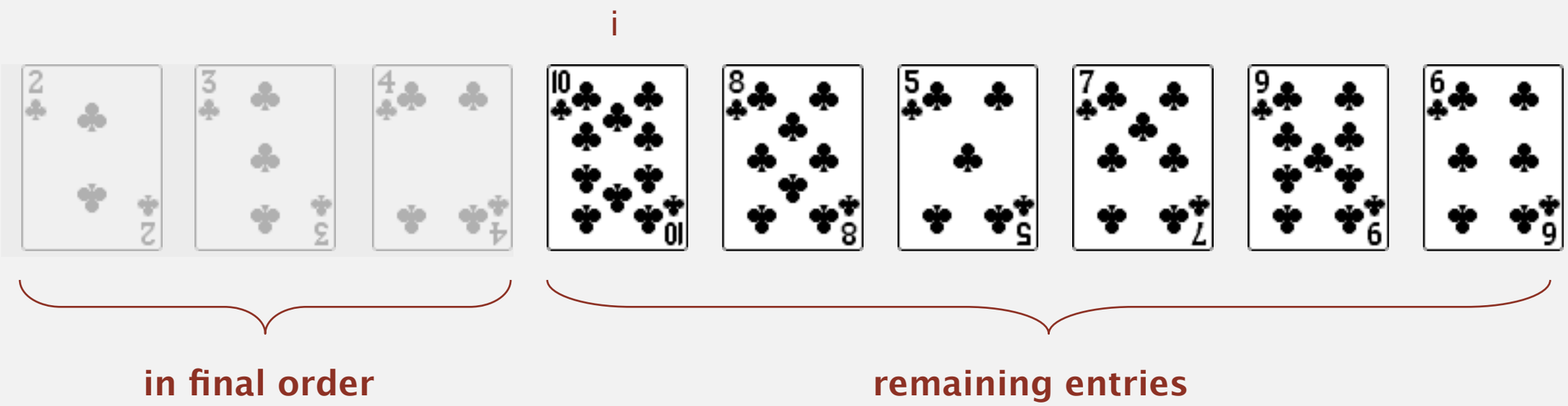
Selection sort demo

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



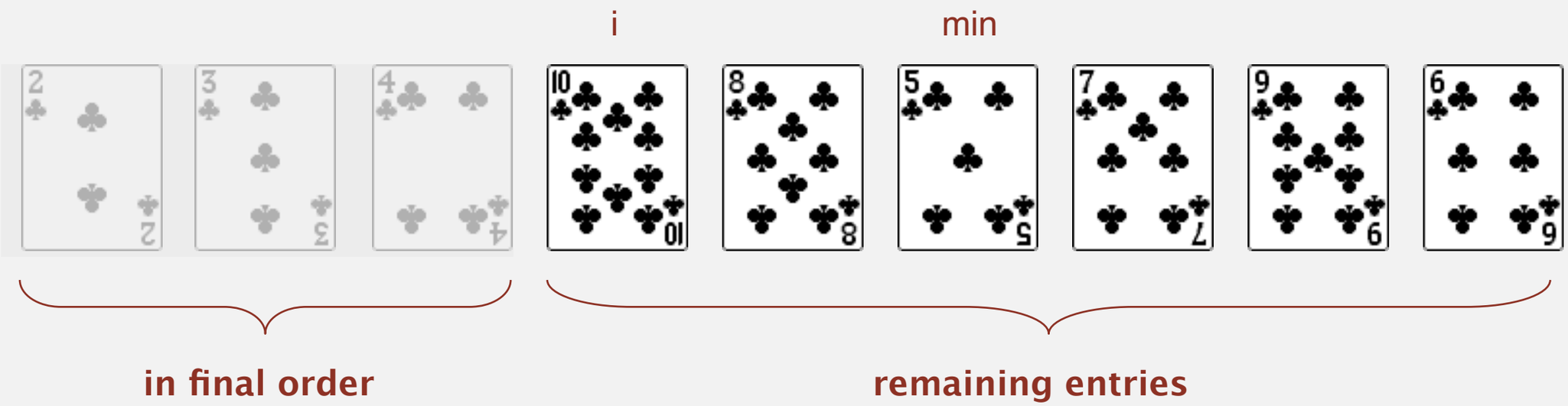
Selection sort demo

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



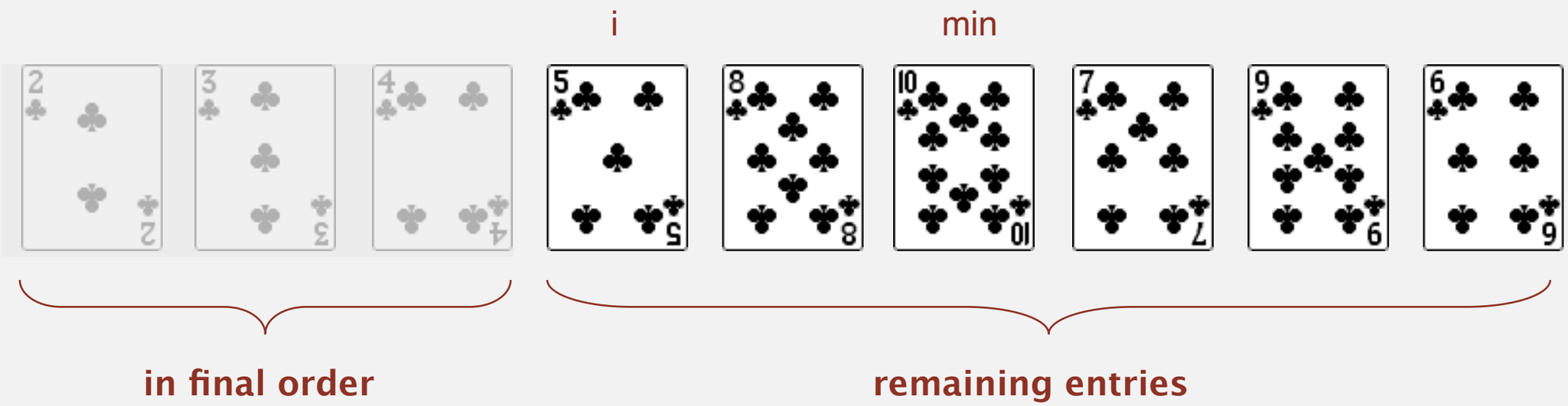
Selection sort demo

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



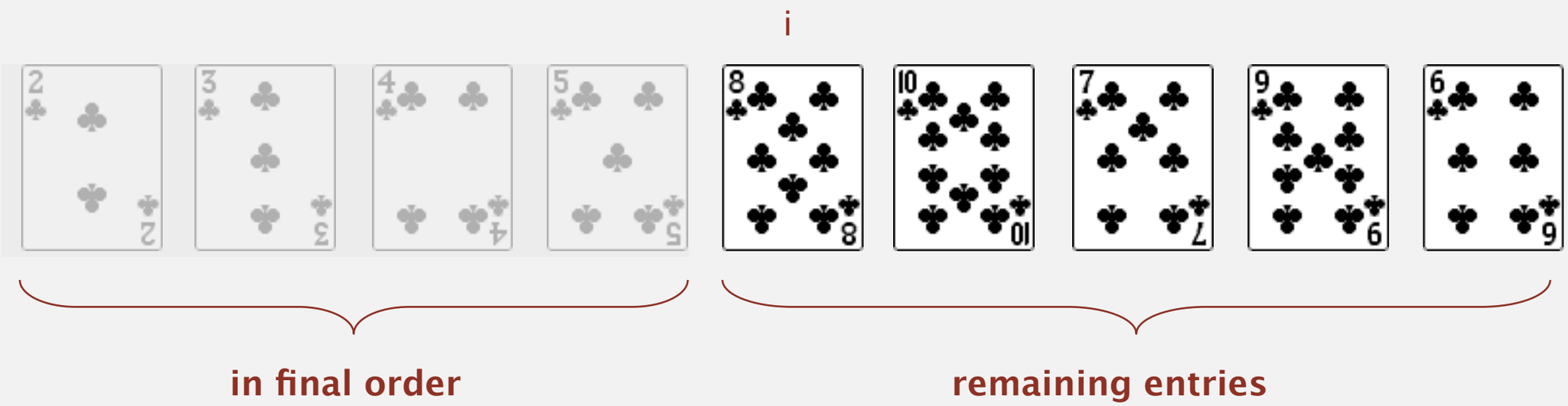
Selection sort demo

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



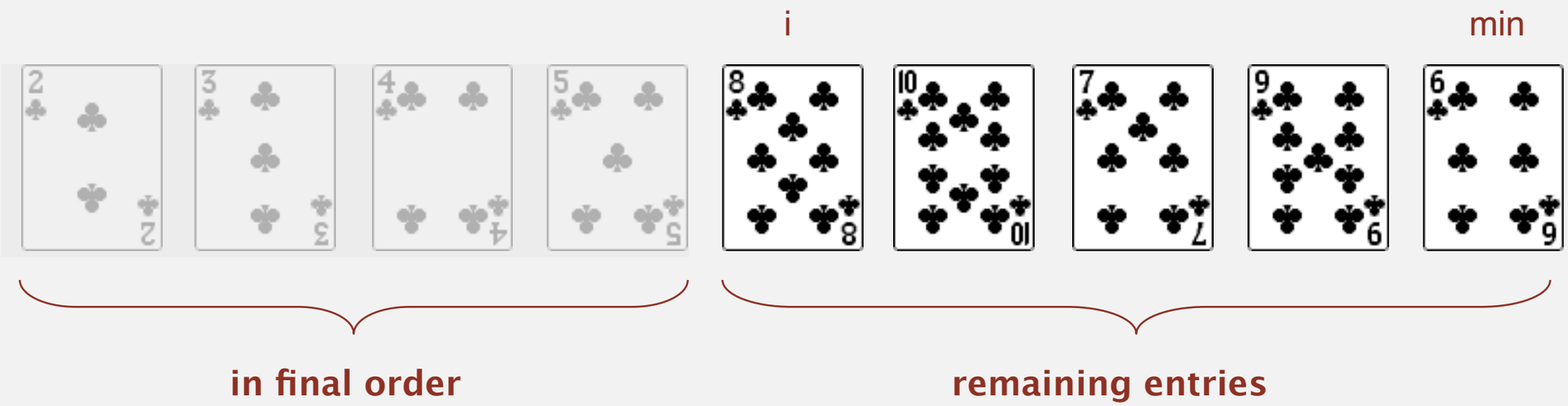
Selection sort demo

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



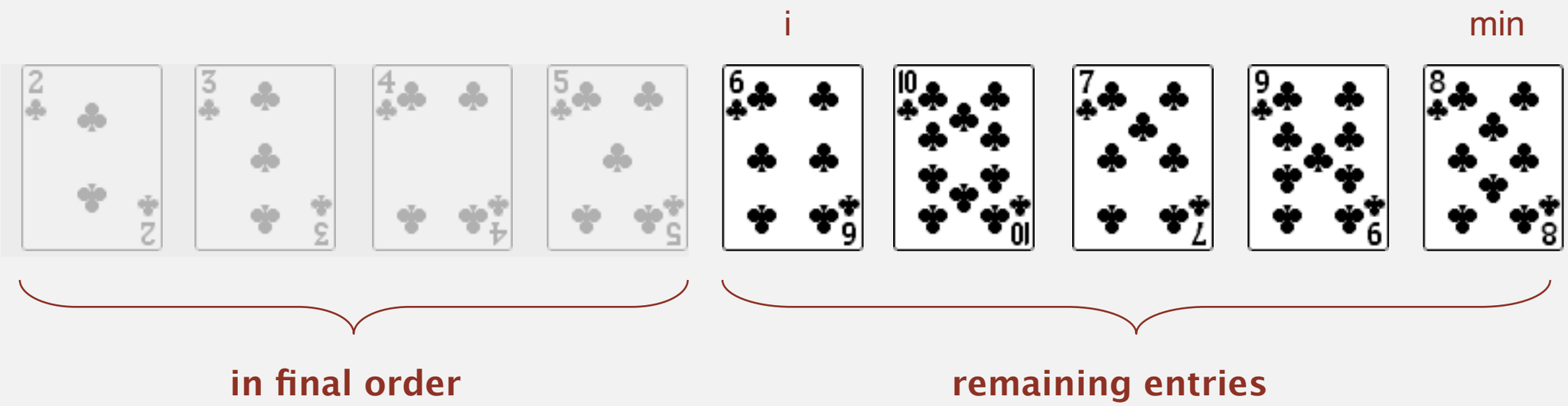
Selection sort demo

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



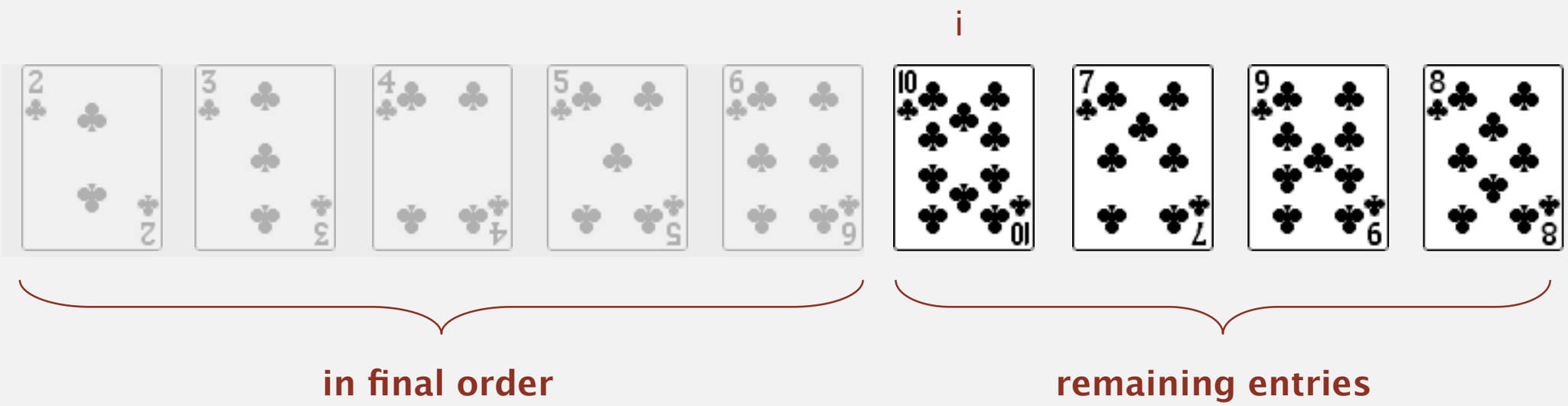
Selection sort demo

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



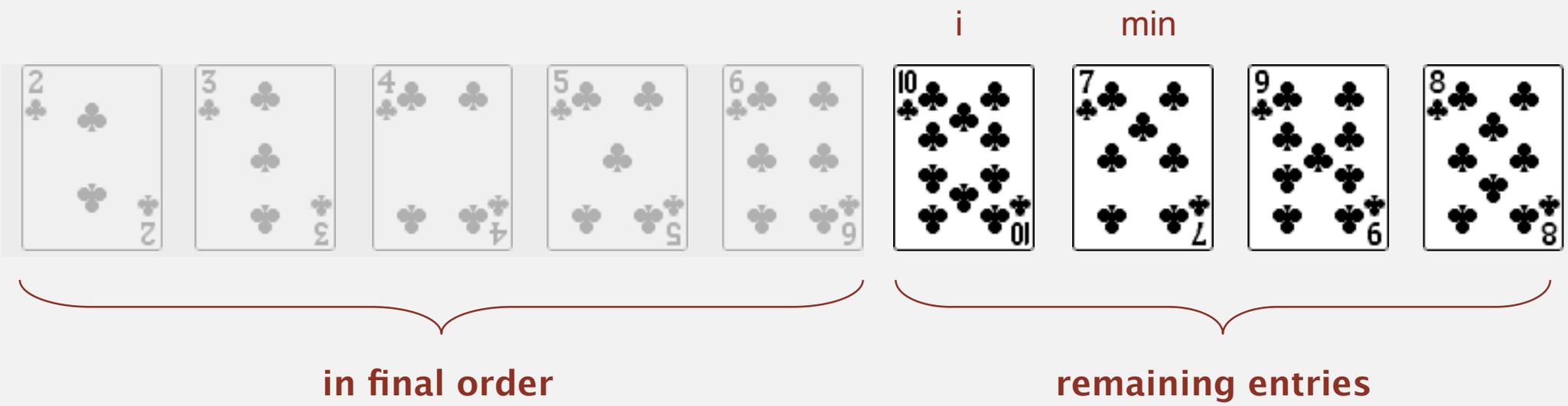
Selection sort demo

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



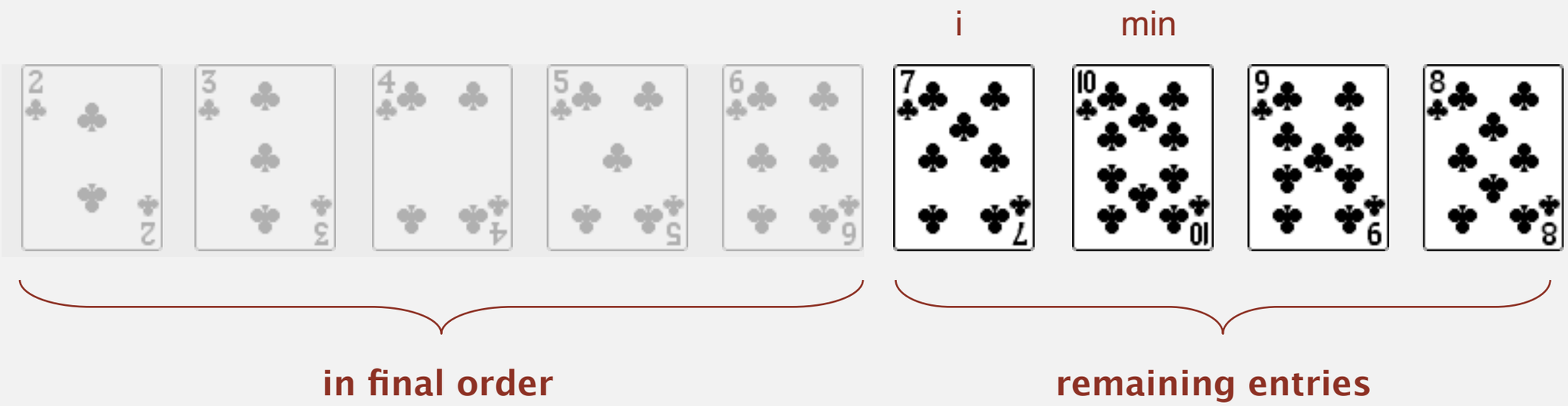
Selection sort demo

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



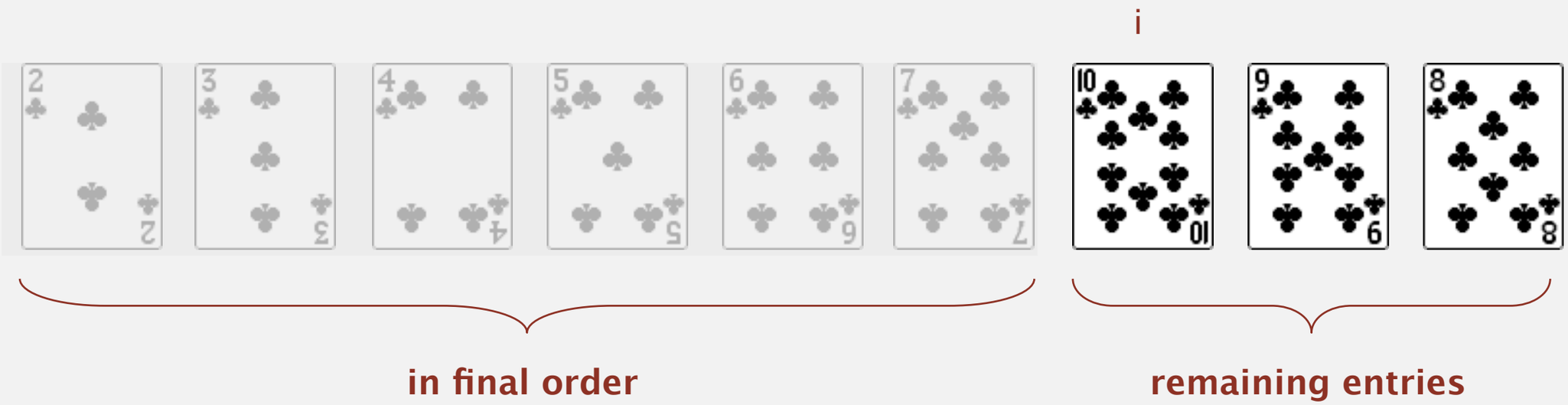
Selection sort demo

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



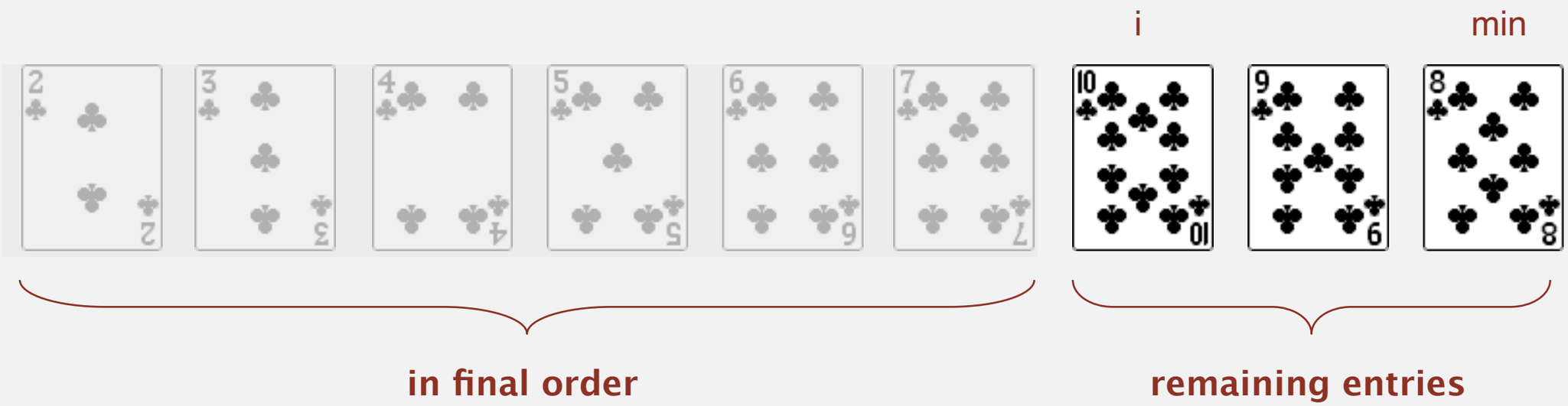
Selection sort demo

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



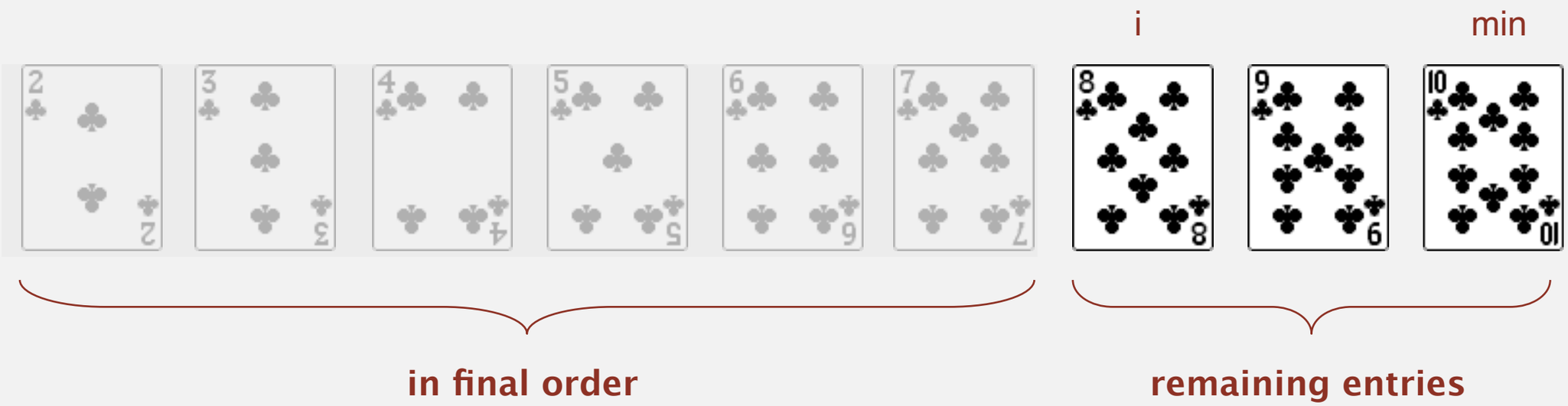
Selection sort demo

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



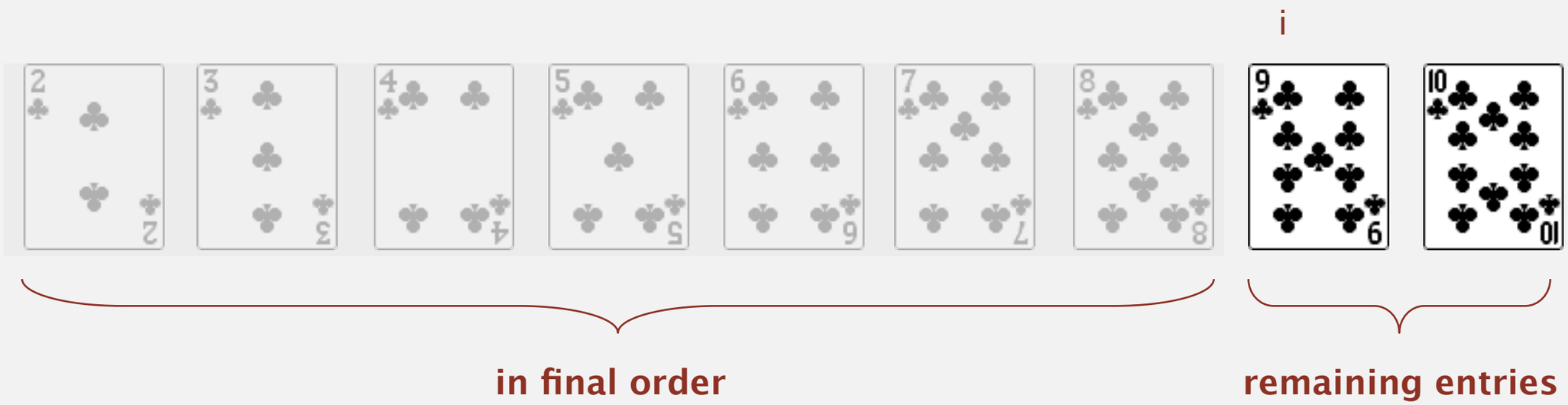
Selection sort demo

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



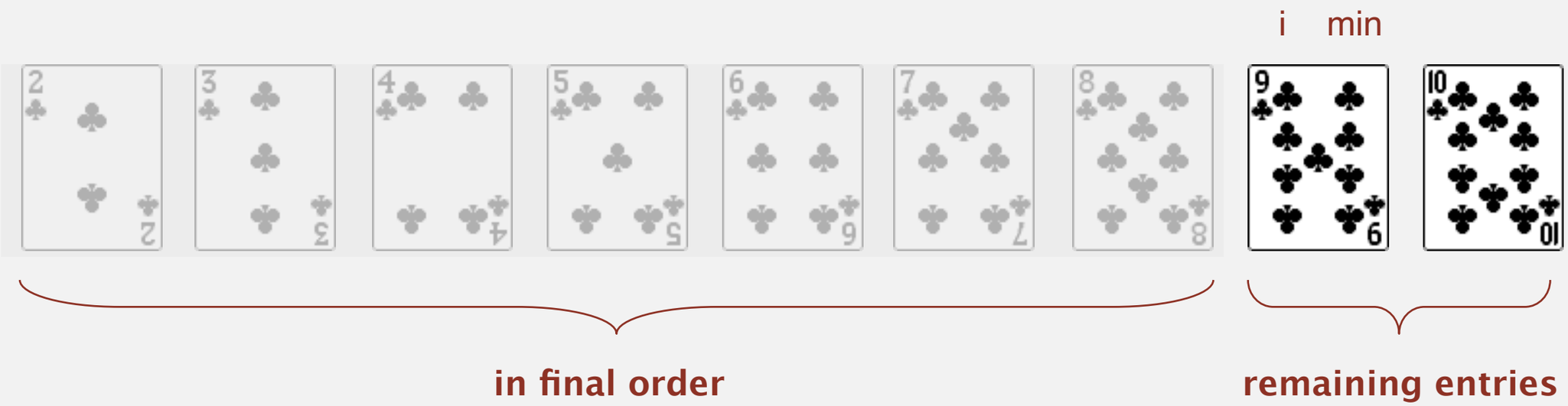
Selection sort demo

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



Selection sort demo

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



Selection sort demo

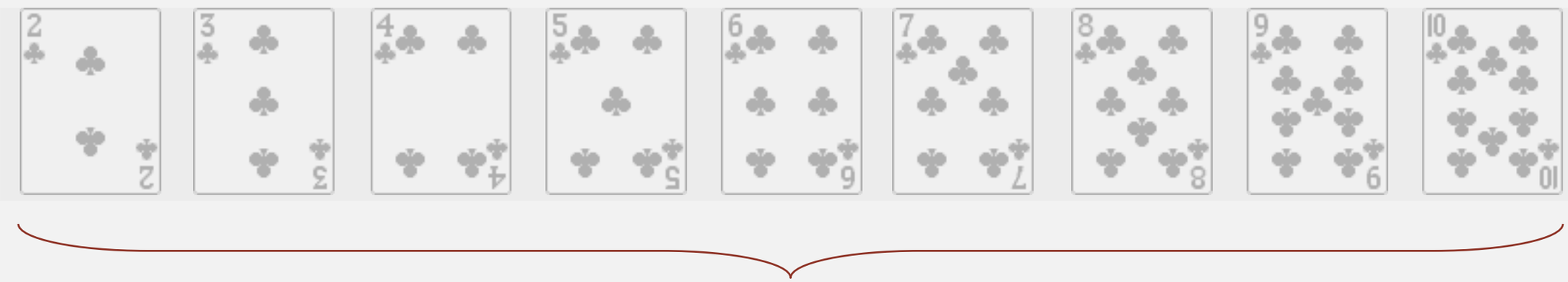
- In iteration i, find index min of smallest remaining entry.
- Swap a[i] and a[min].

Selection sort demo

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

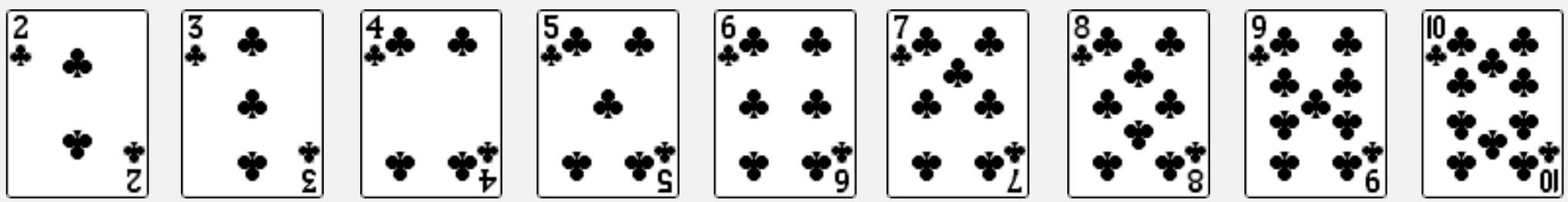
Selection sort demo

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



Selection sort demo

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



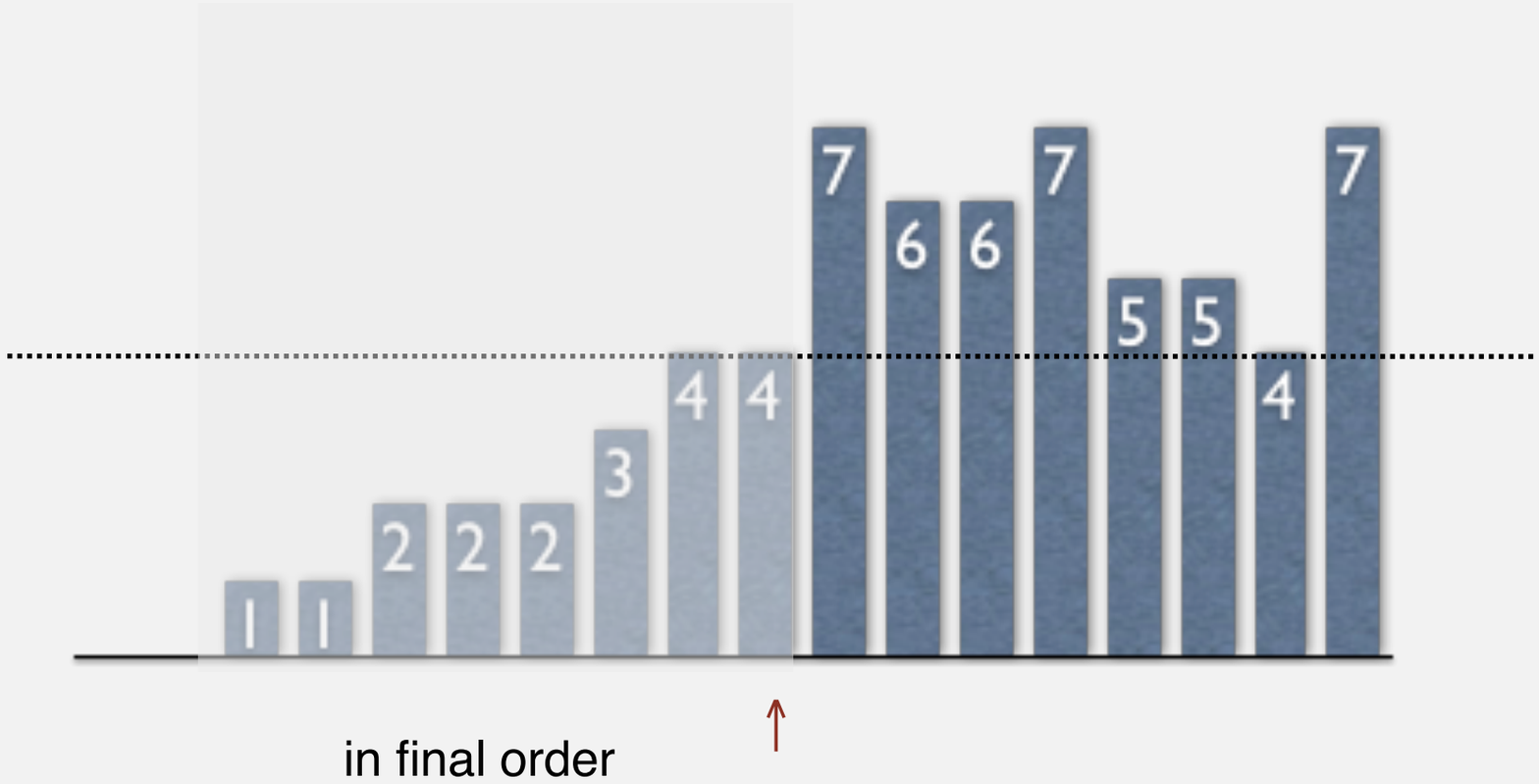
sorted

Selection sort

Algorithm. ↑ scans from left to right.

Invariants.

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



Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is item v less than w ?

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

Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is item v less than w ?

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

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

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

Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Identify index of minimum entry on right.

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

Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

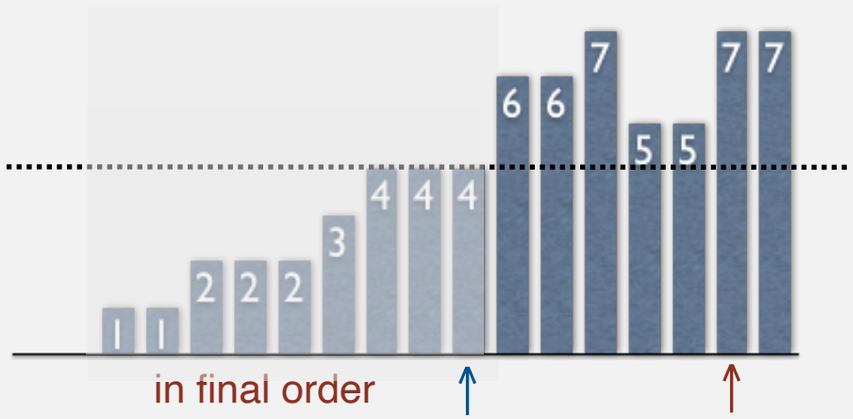
```
i++;
```

- Identify index of minimum entry on right.

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

- Exchange into position.

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

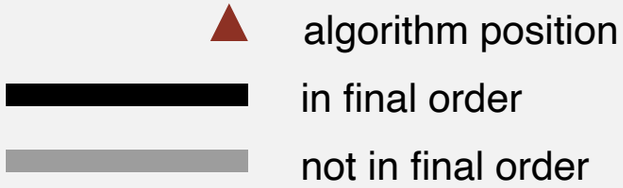


Selection sort: Java implementation

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

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

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

Selection sort: animations

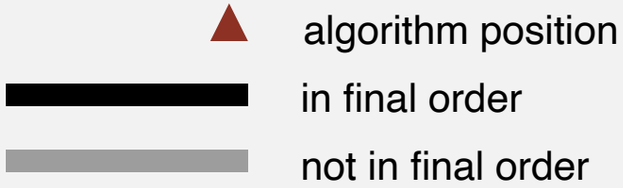
20 random items



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

Selection sort: animations

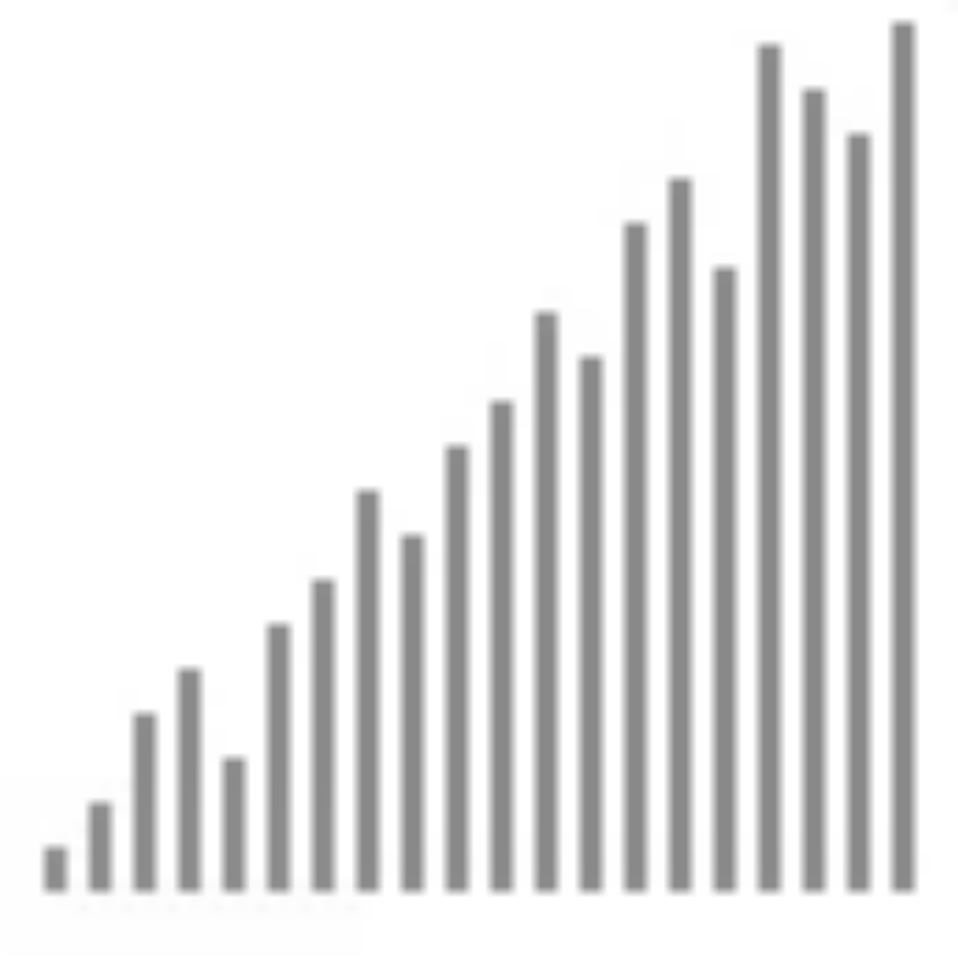
20 random items

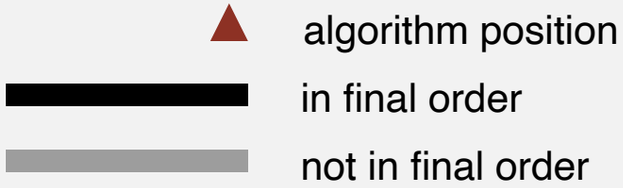


http://www.sorting-algorithms.com/selection-sort

Selection sort: animations

20 partially-sorted items

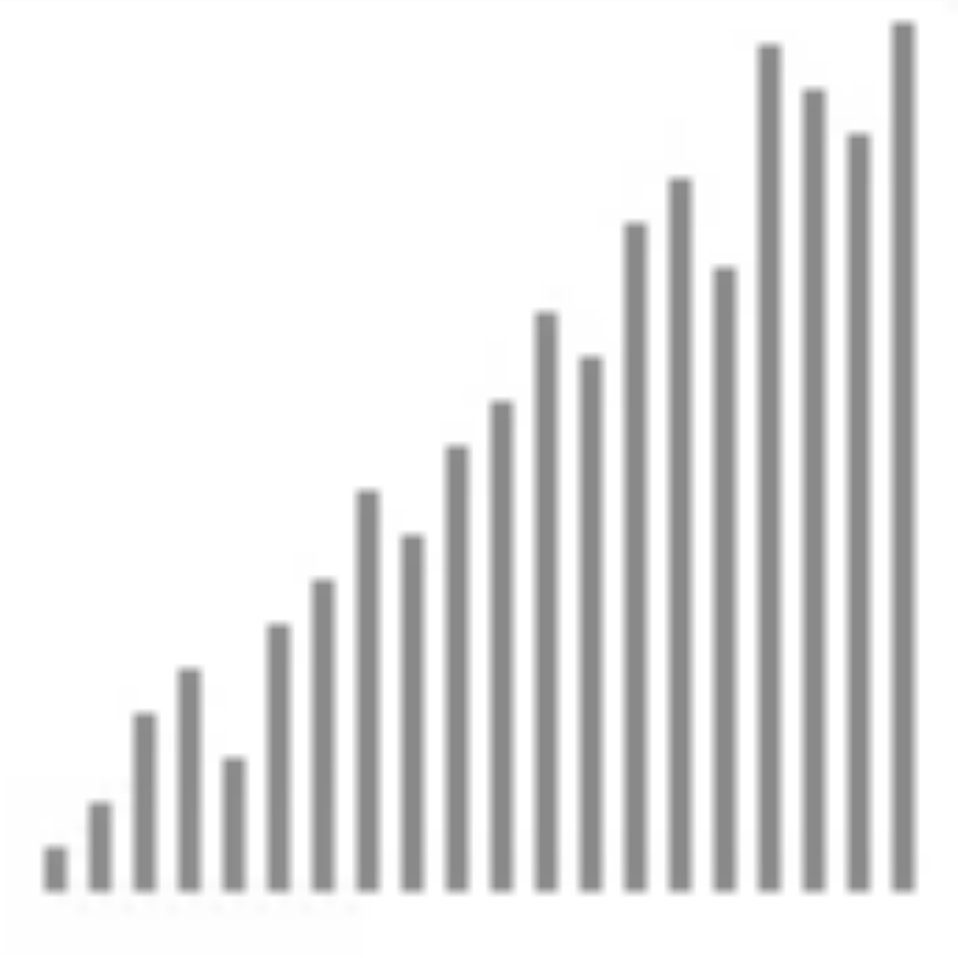


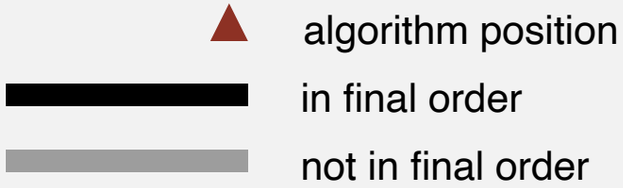


http://www.sorting-algorithms.com/selection-sort

Selection sort: animations

20 partially-sorted items





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

Selection sort: mathematical analysis

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

		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: mathematical analysis

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

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

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

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

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

Selection sort: mathematical analysis

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

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

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

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

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

Data movement is minimal. Linear number of exchanges.

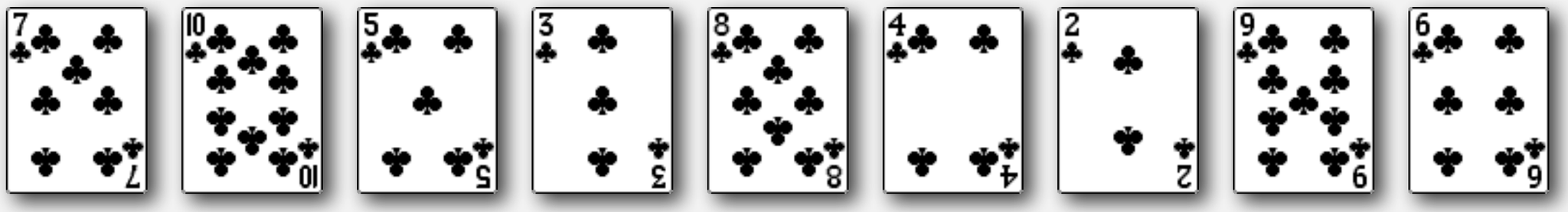
http://algs4.cs.princeton.edu

2.1 Elementary Sorts

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

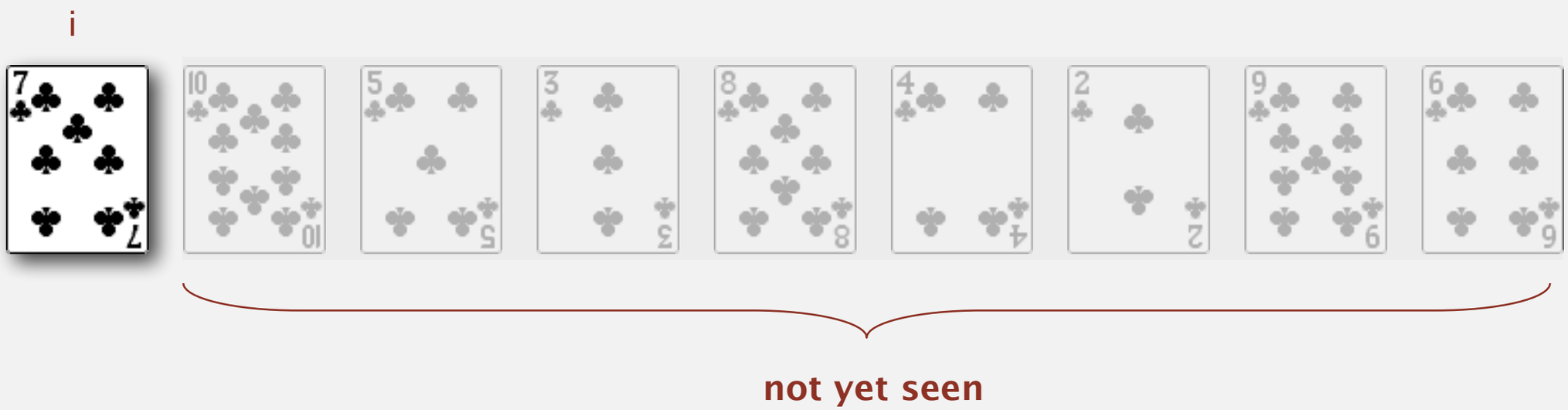
Insertion sort demo

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



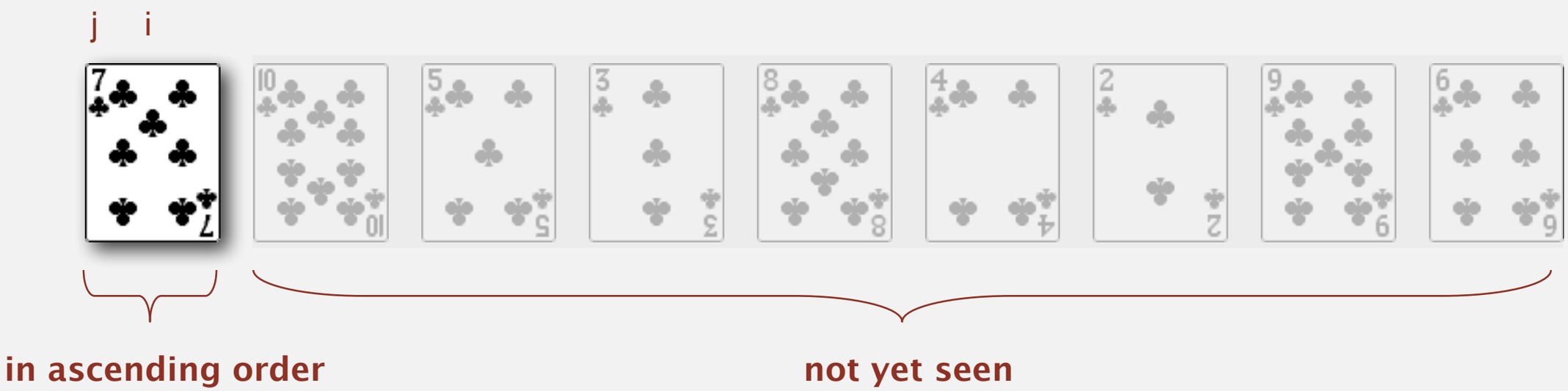
Insertion sort demo

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



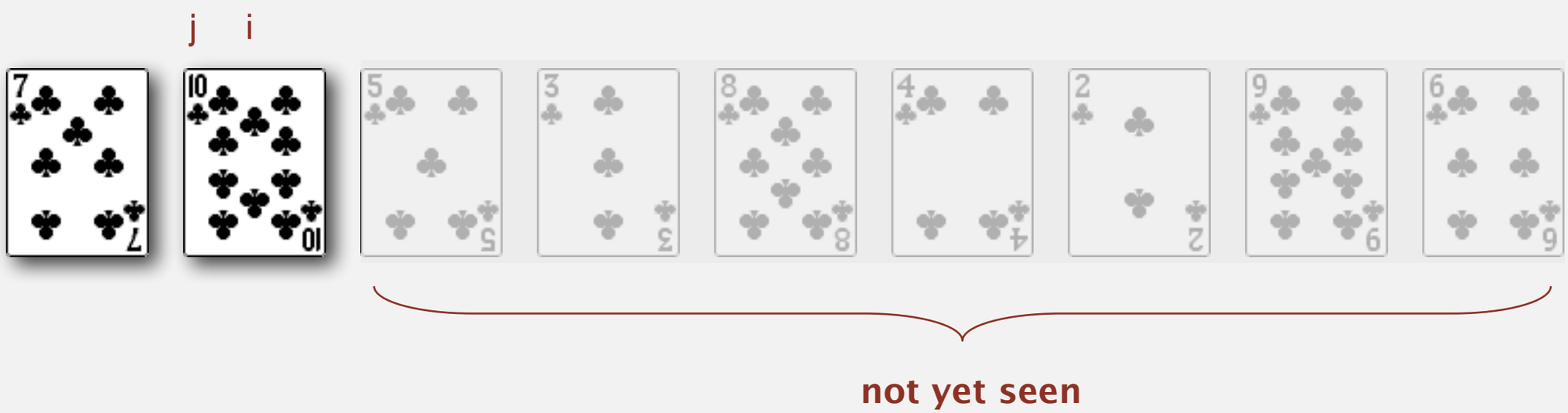
Insertion sort demo

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



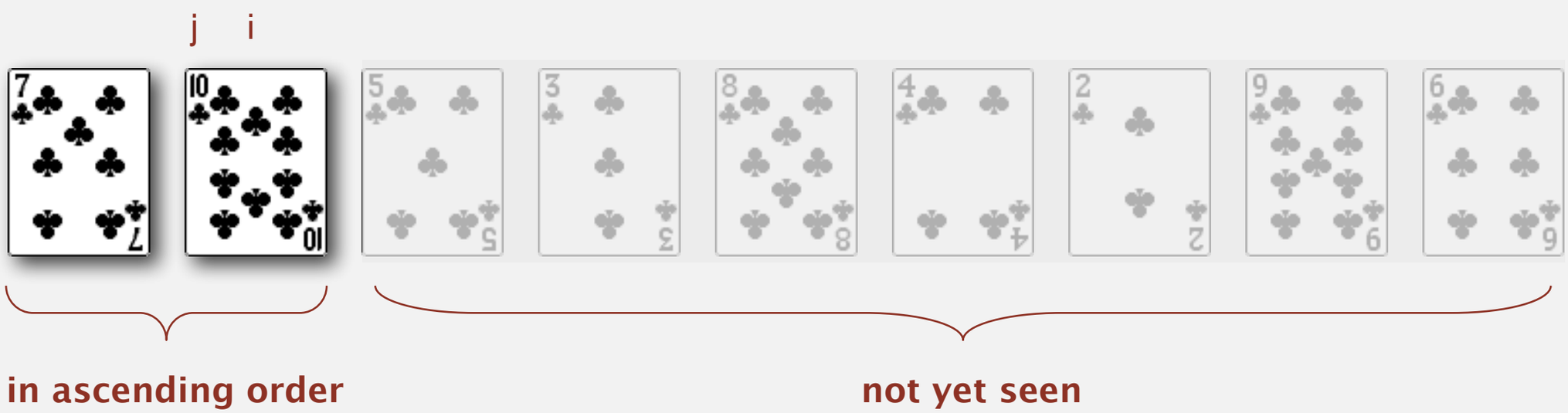
Insertion sort demo

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



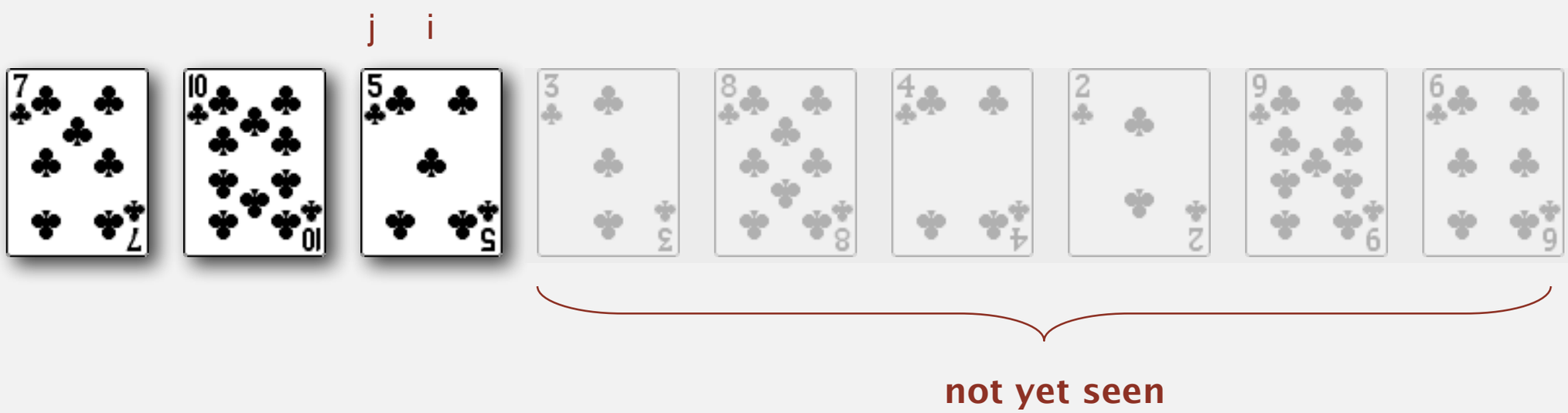
Insertion sort demo

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



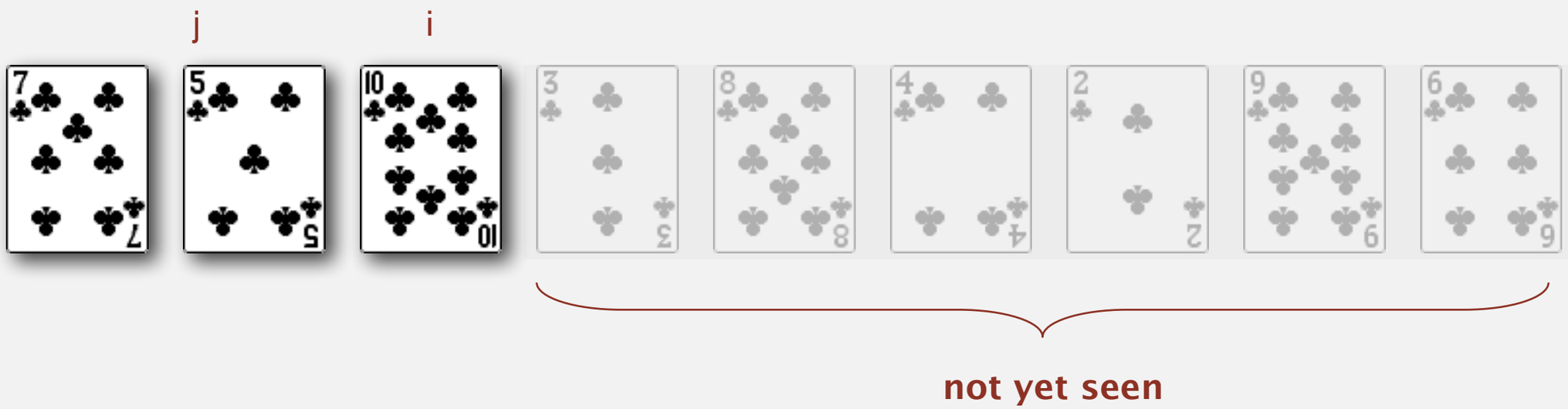
Insertion sort demo

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



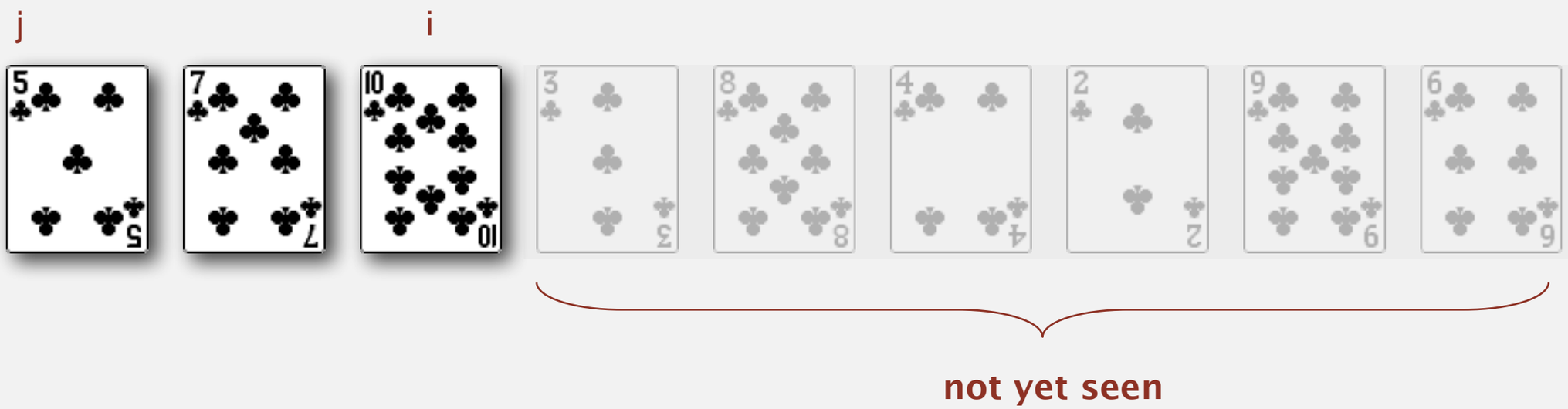
Insertion sort demo

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



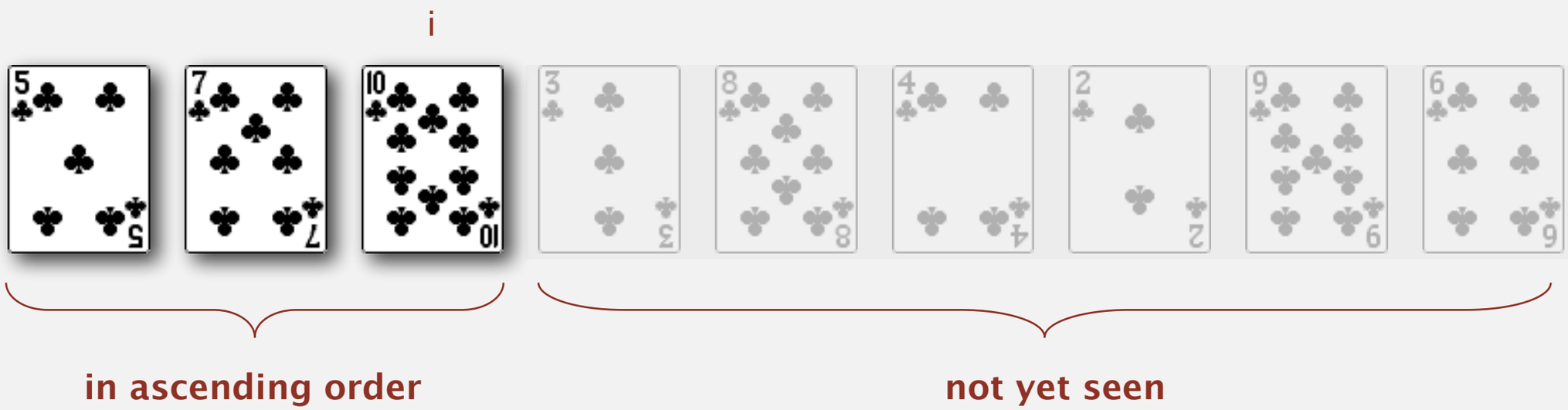
Insertion sort demo

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



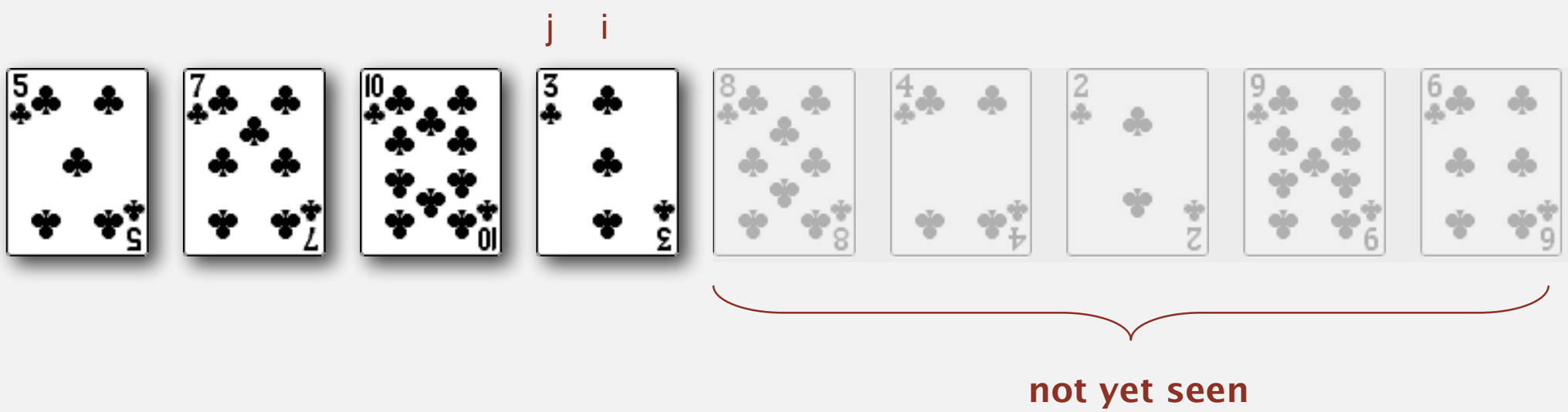
Insertion sort demo

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



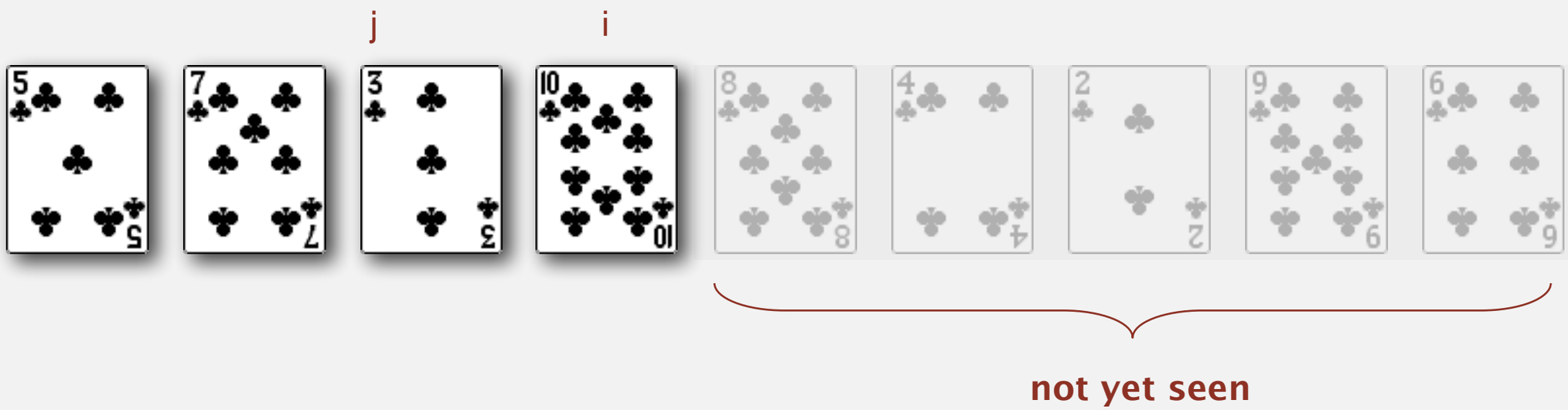
Insertion sort demo

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



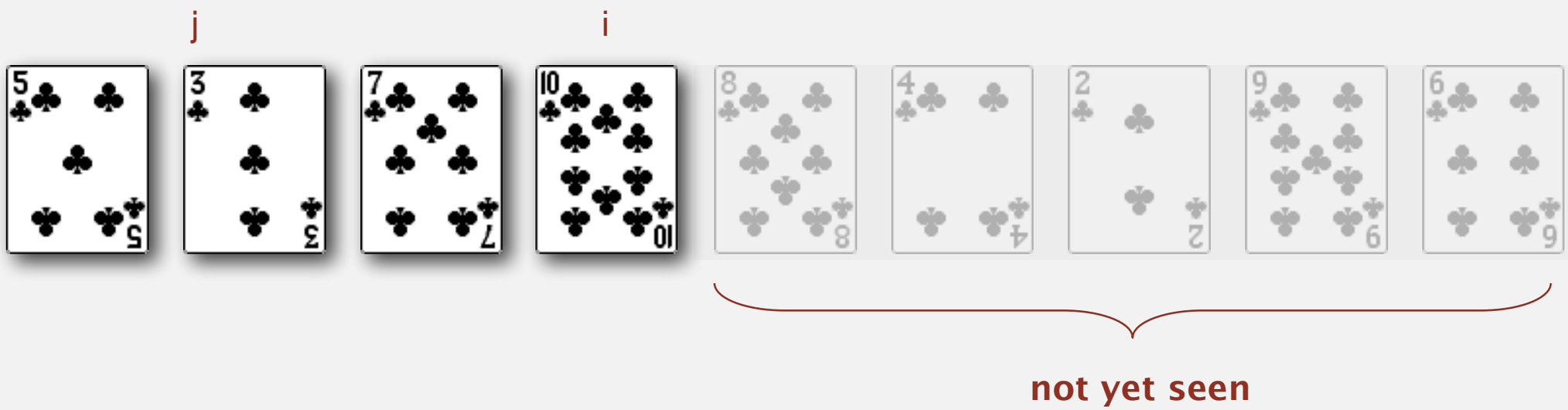
Insertion sort demo

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



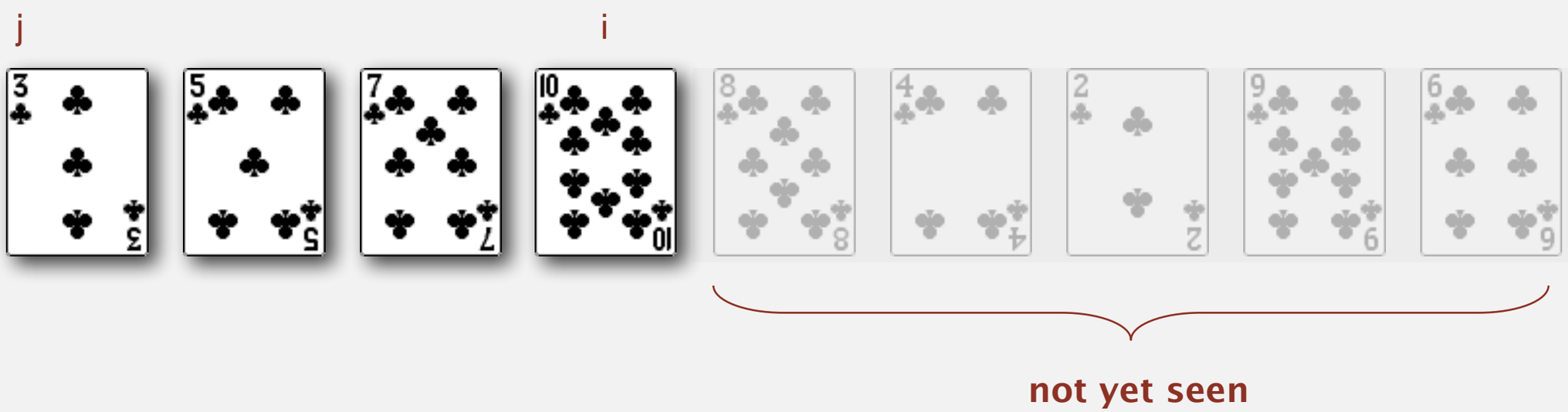
Insertion sort demo

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



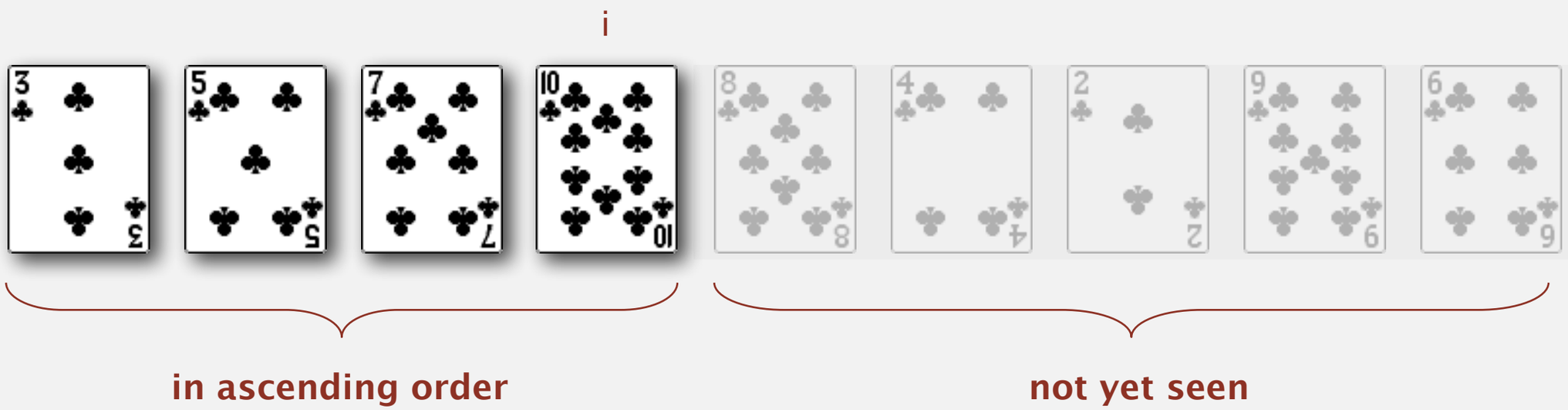
Insertion sort demo

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



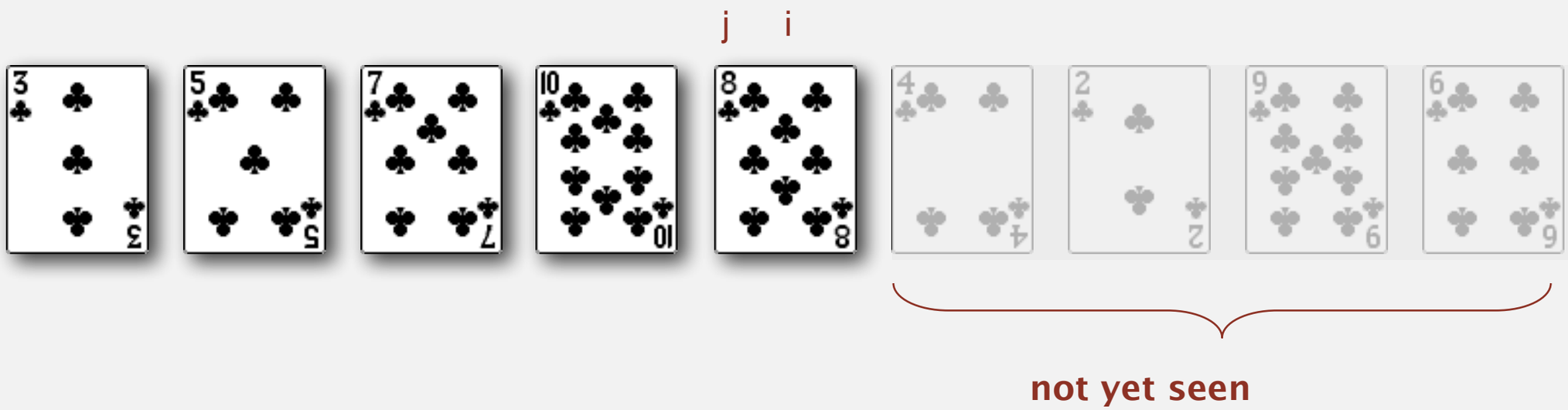
Insertion sort demo

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



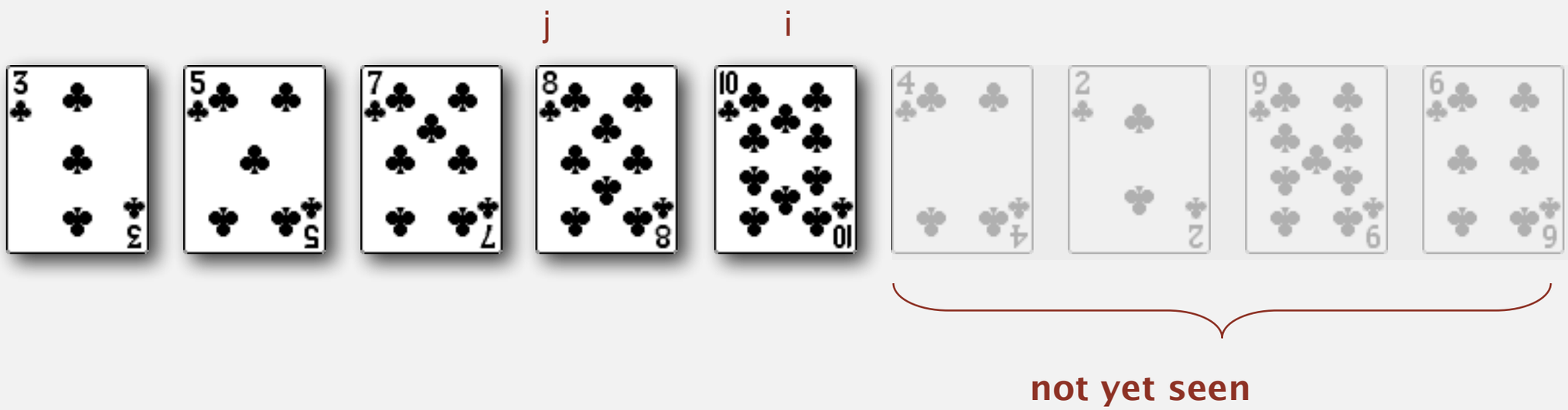
Insertion sort demo

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



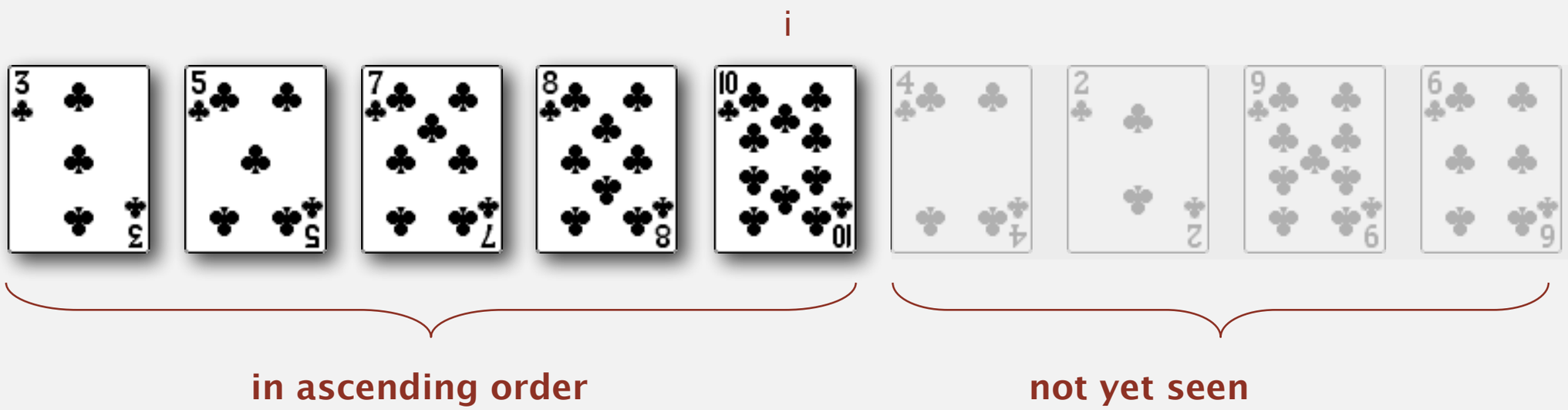
Insertion sort demo

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



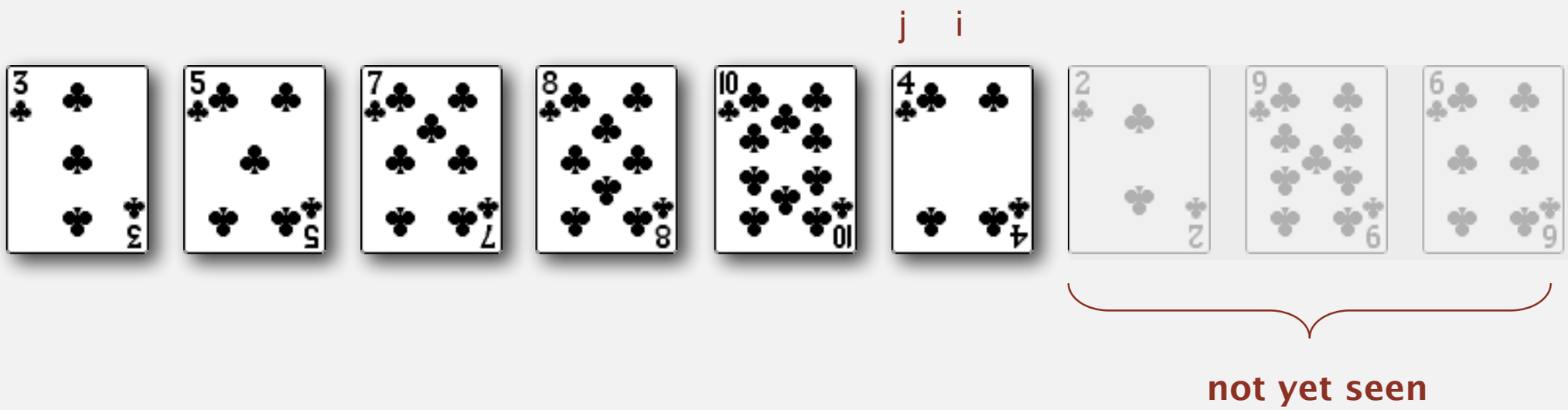
Insertion sort demo

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



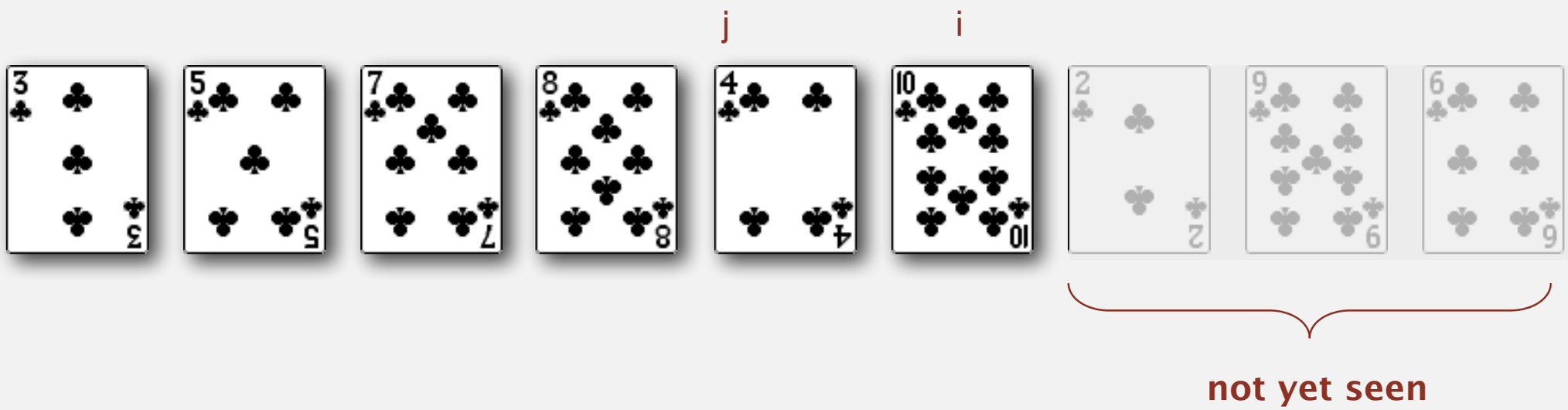
Insertion sort demo

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



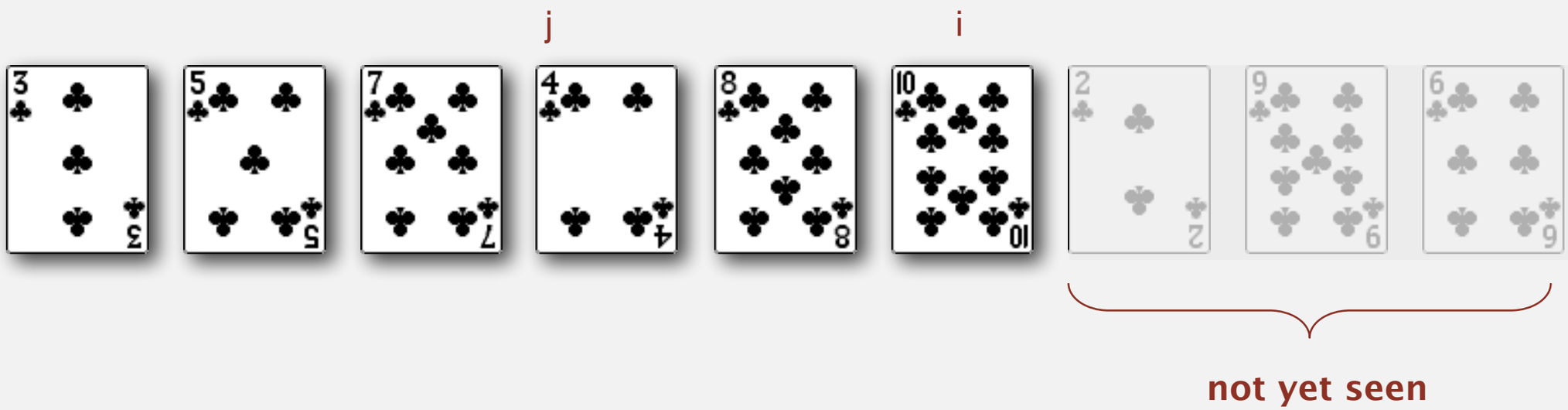
Insertion sort demo

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



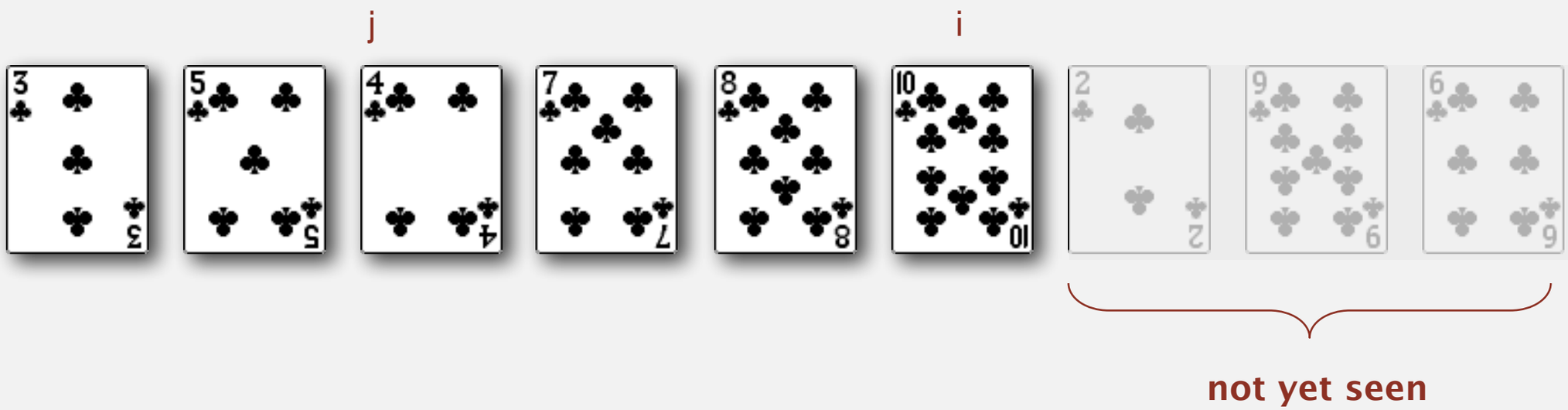
Insertion sort demo

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



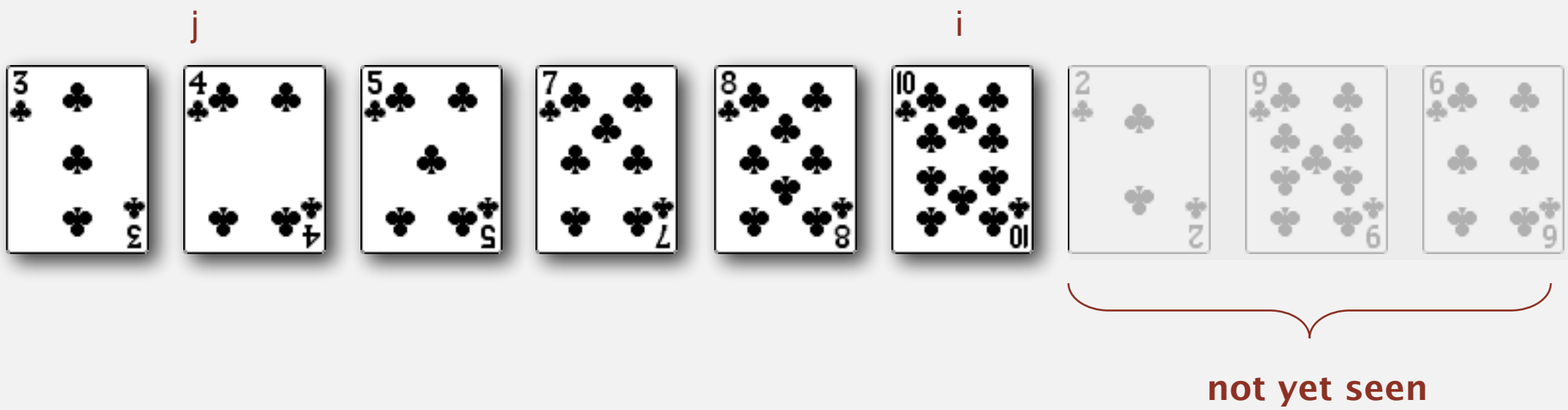
Insertion sort demo

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



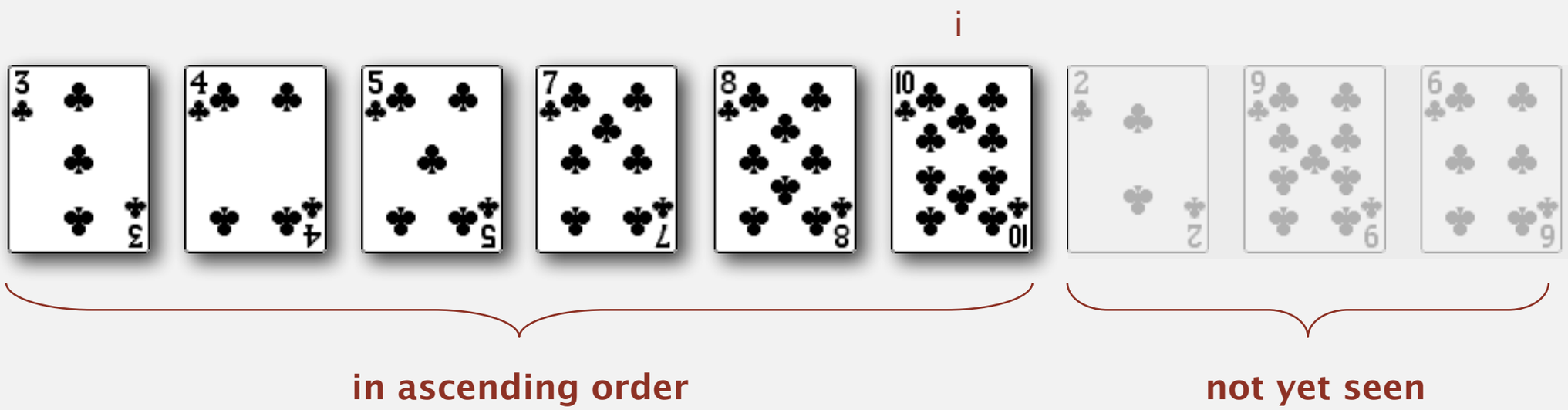
Insertion sort demo

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



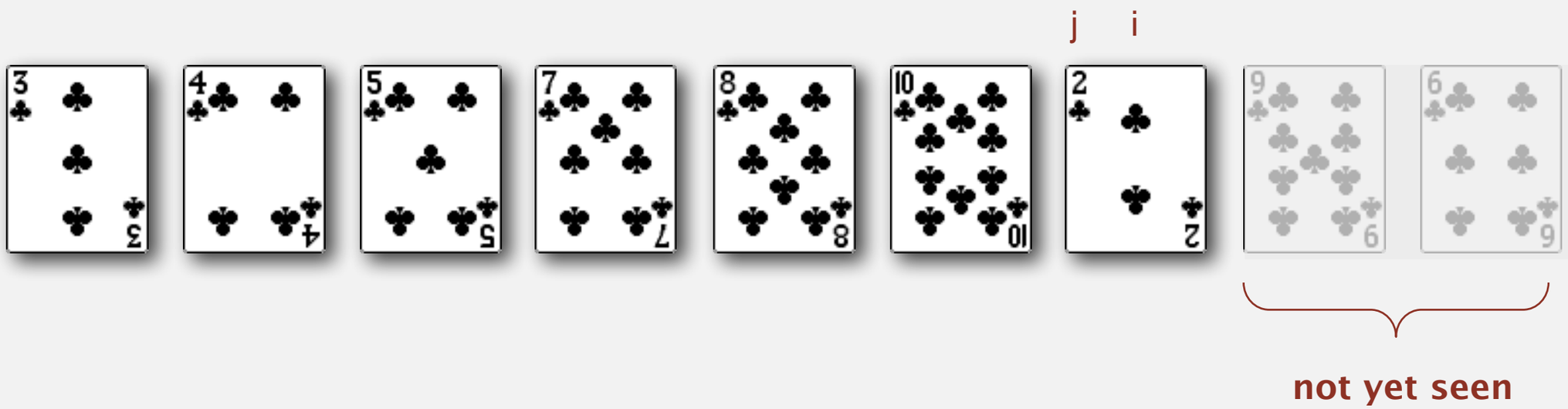
Insertion sort demo

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



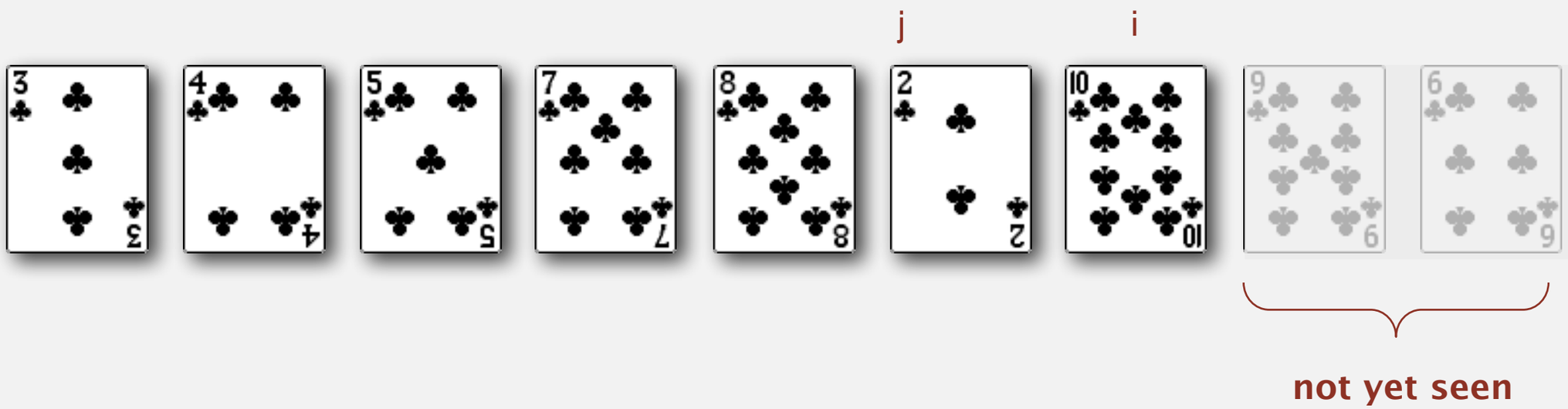
Insertion sort demo

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



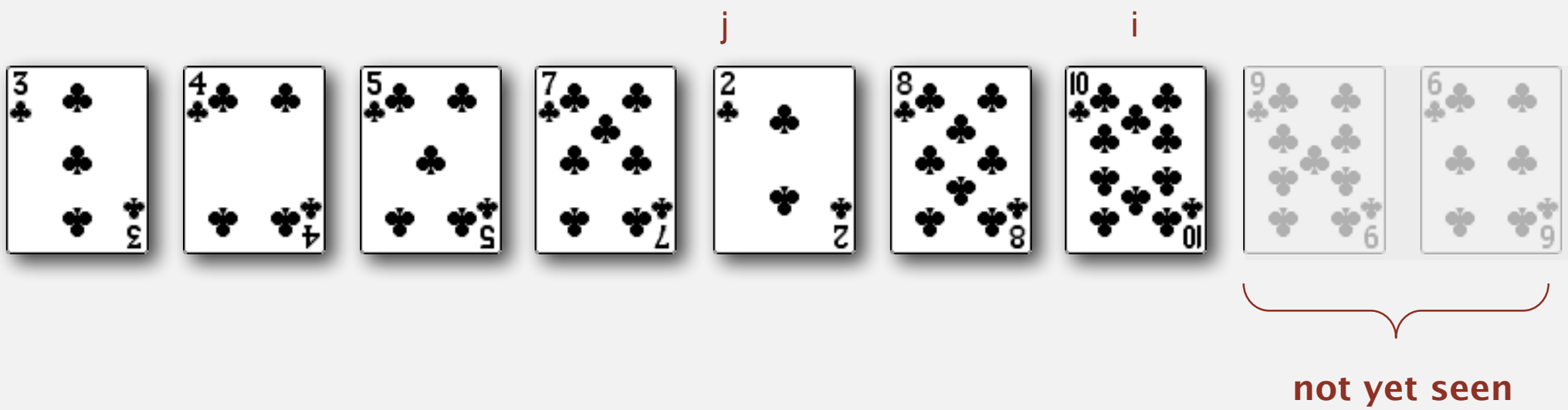
Insertion sort demo

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



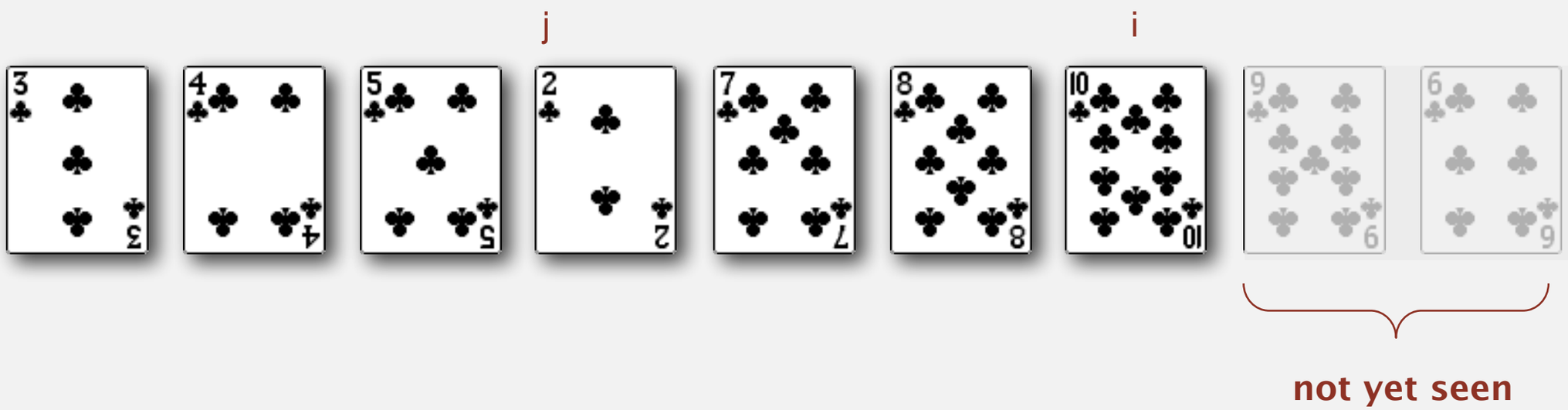
Insertion sort demo

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



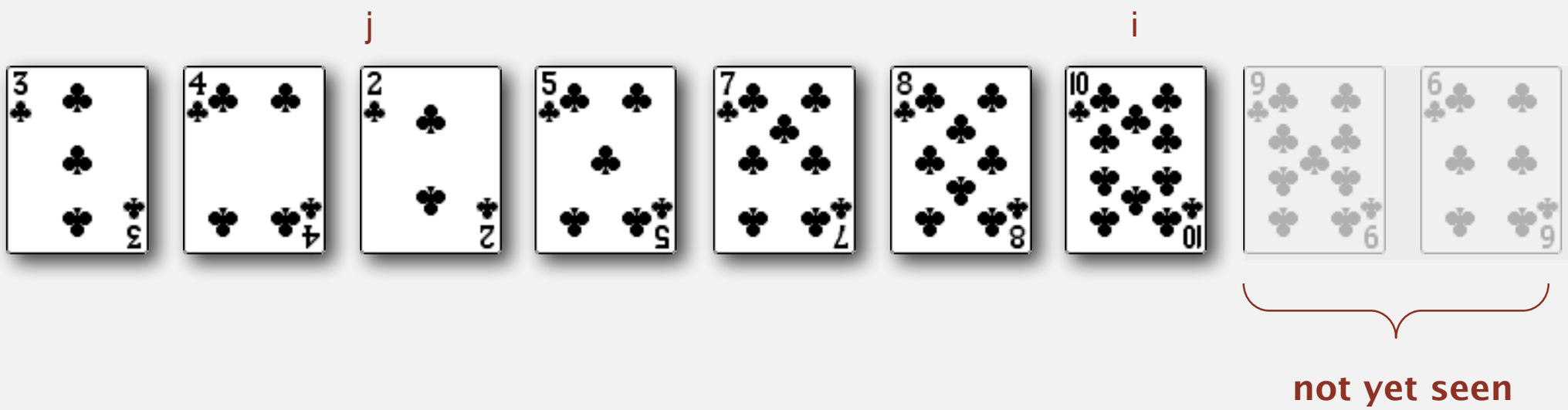
Insertion sort demo

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



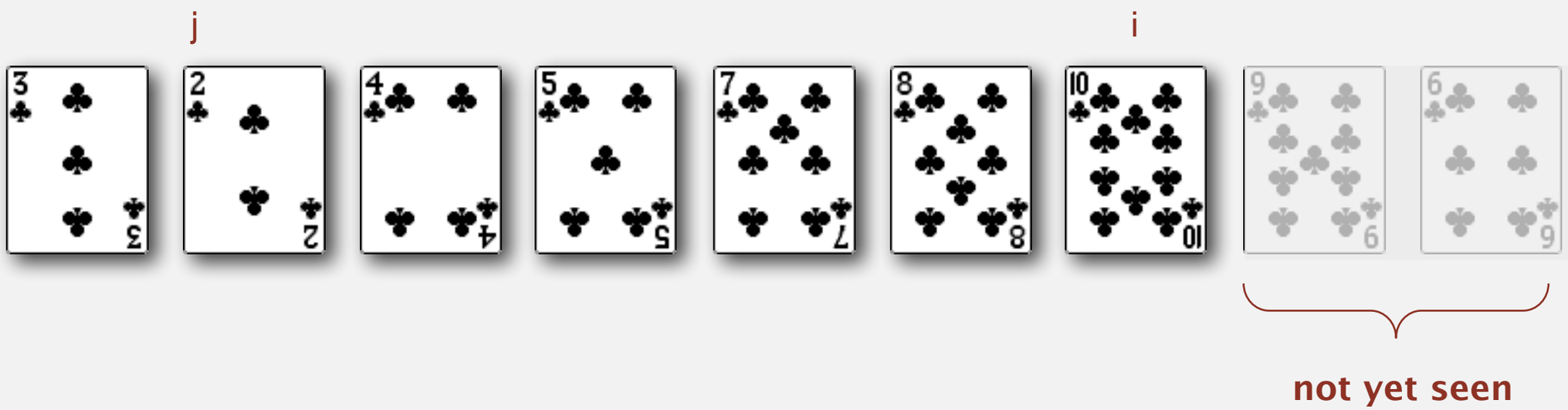
Insertion sort demo

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



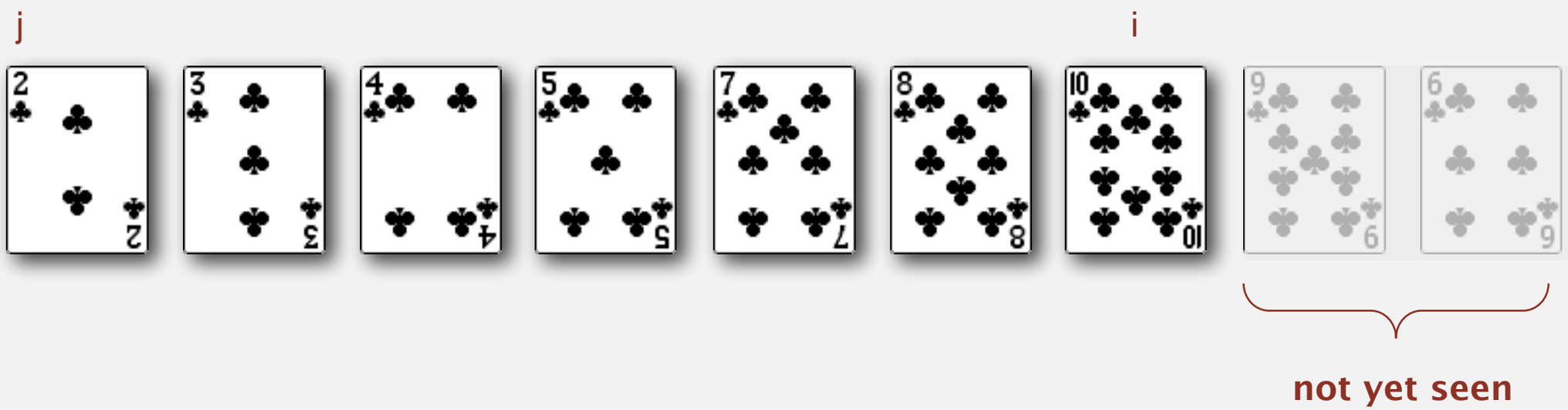
Insertion sort demo

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



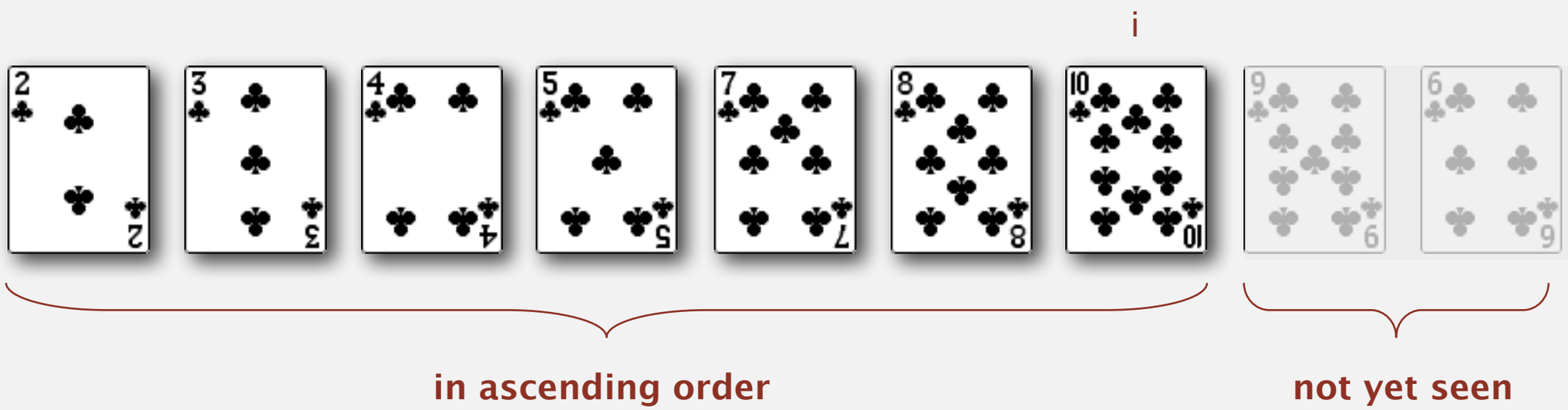
Insertion sort demo

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



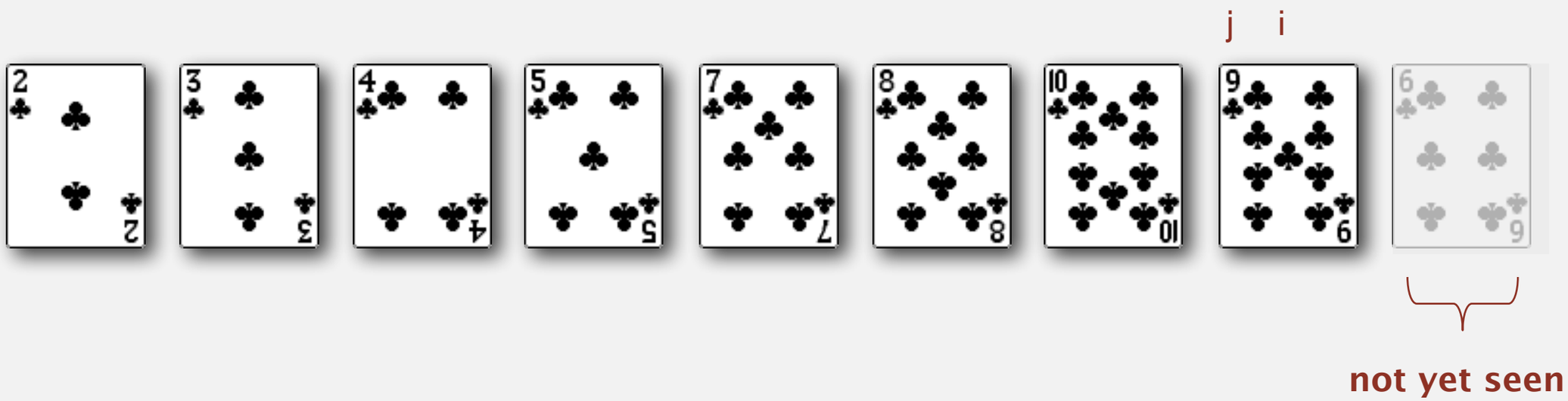
Insertion sort demo

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



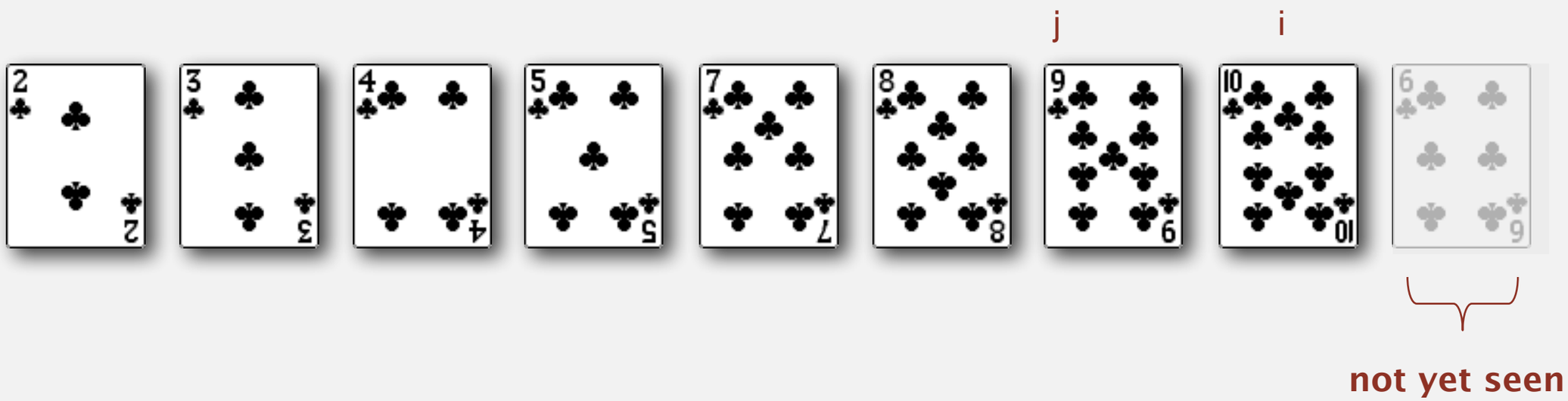
Insertion sort demo

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



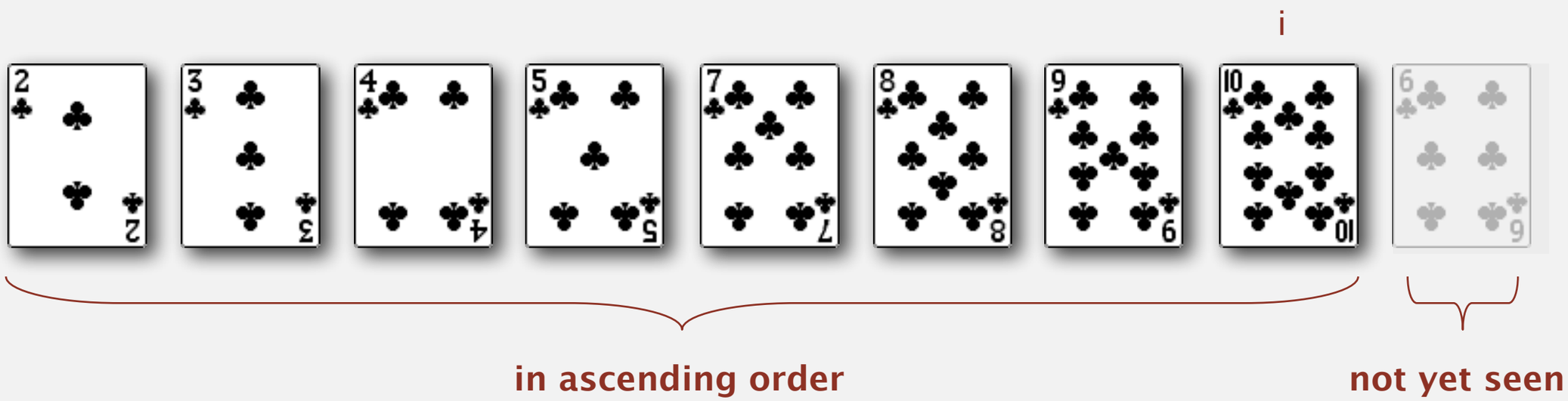
Insertion sort demo

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



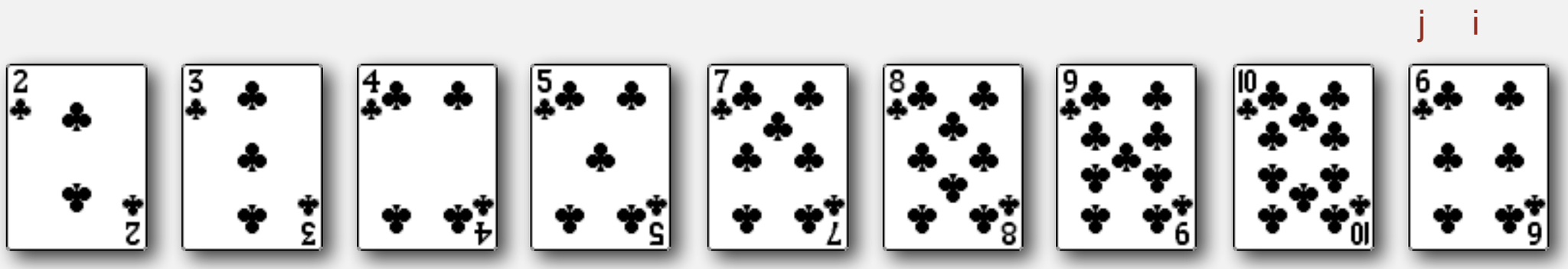
Insertion sort demo

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



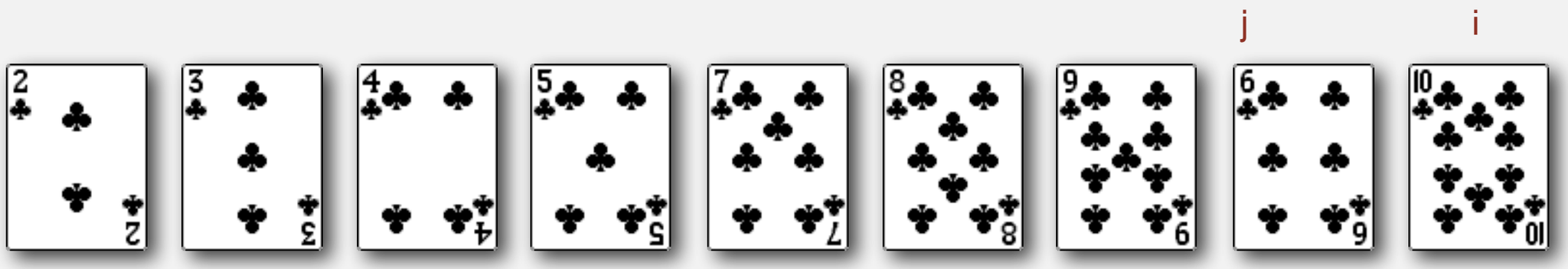
Insertion sort demo

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



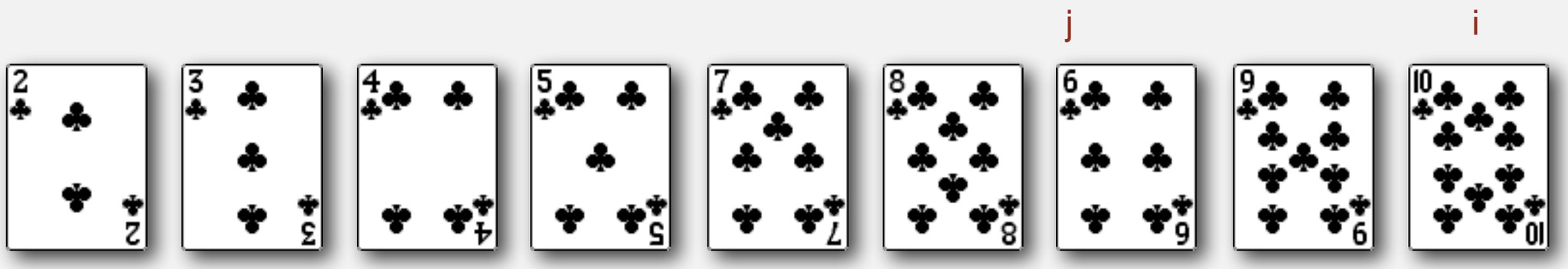
Insertion sort demo

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



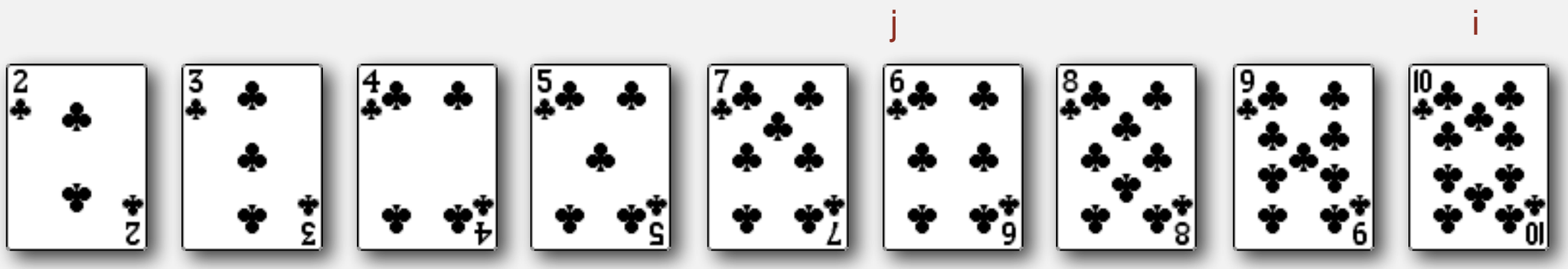
Insertion sort demo

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



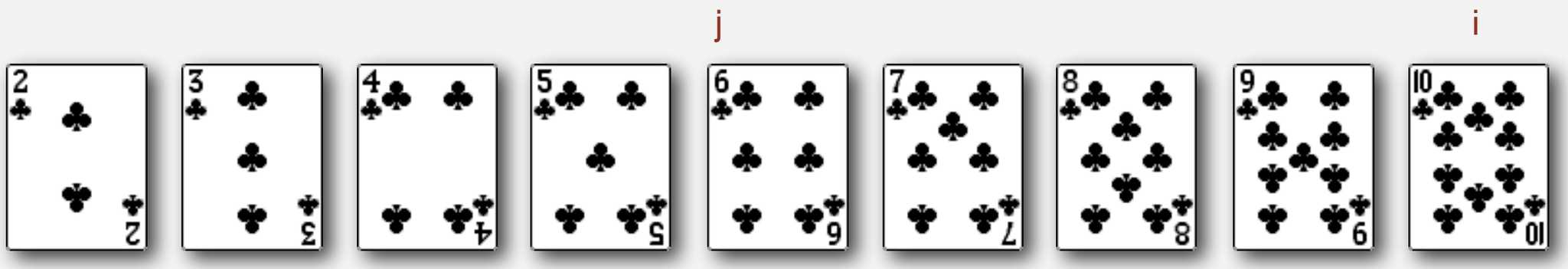
Insertion sort demo

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



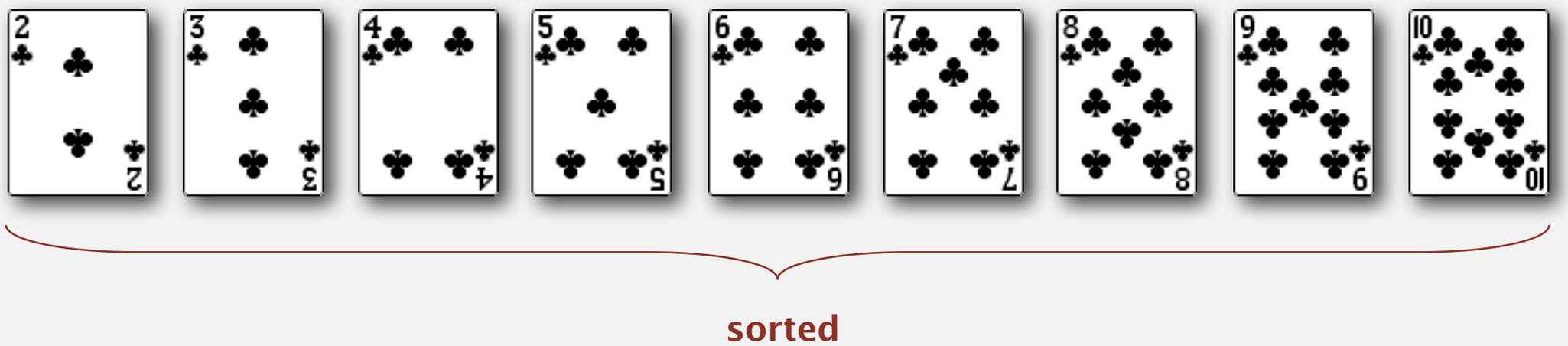
Insertion sort demo

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



Insertion sort demo

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



Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

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

Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

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

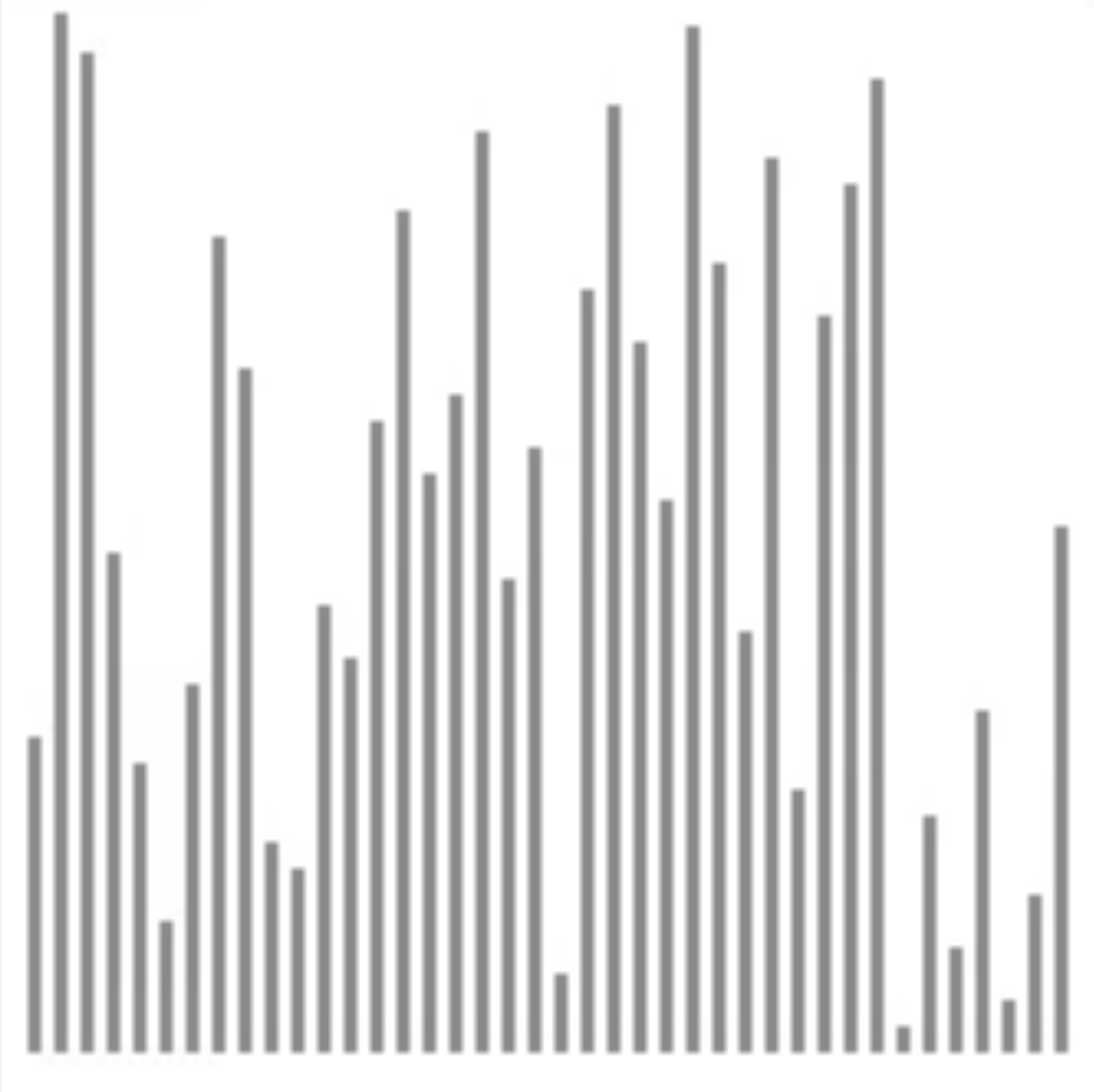
```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
      exch(a, j, j-1);
   else break;
```

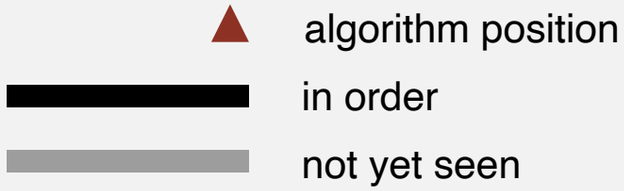
Insertion sort: Java implementation

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

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

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

Insertion sort: animation

40 random items

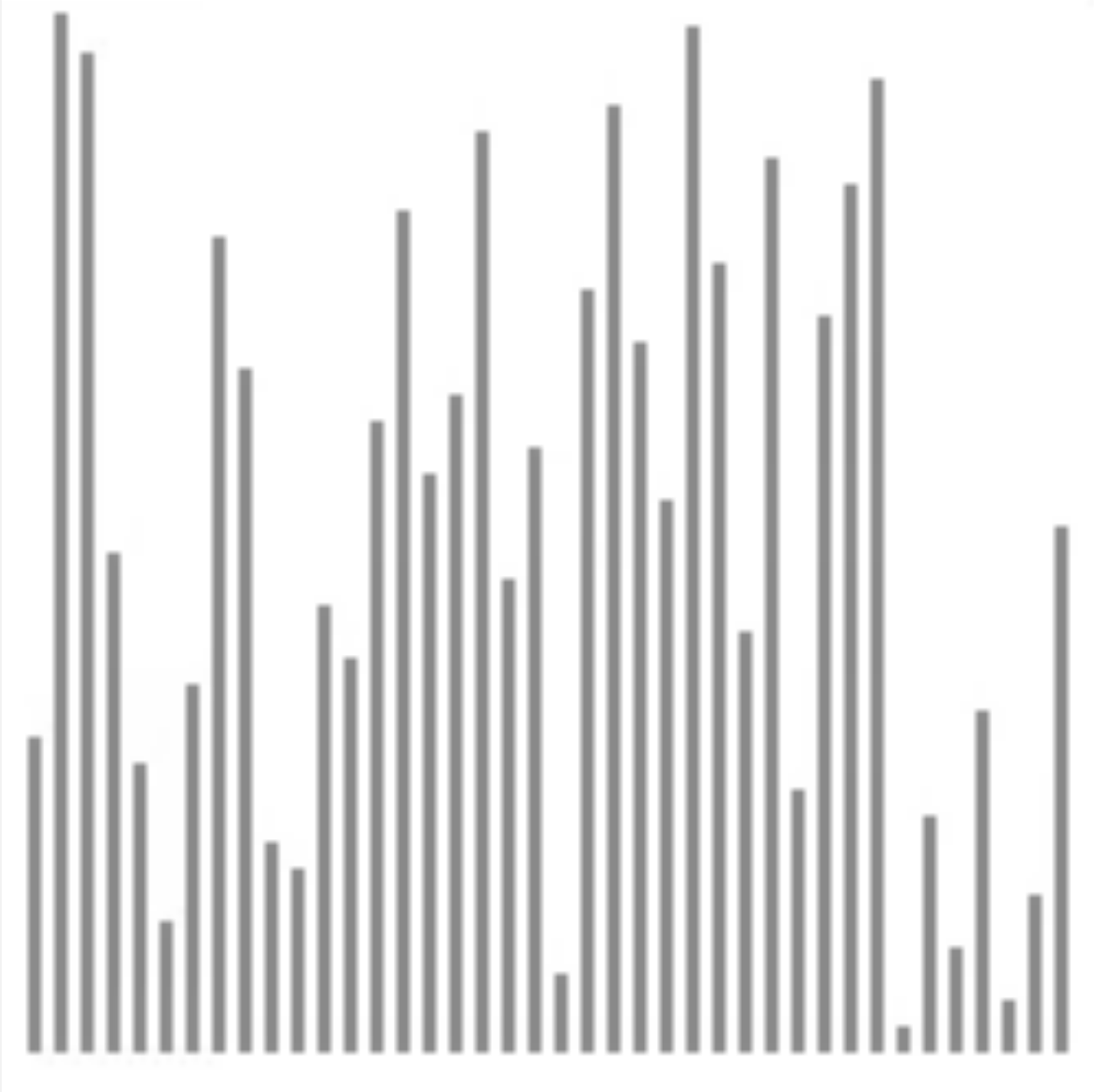


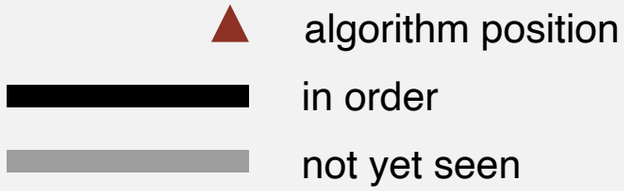


http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 random items

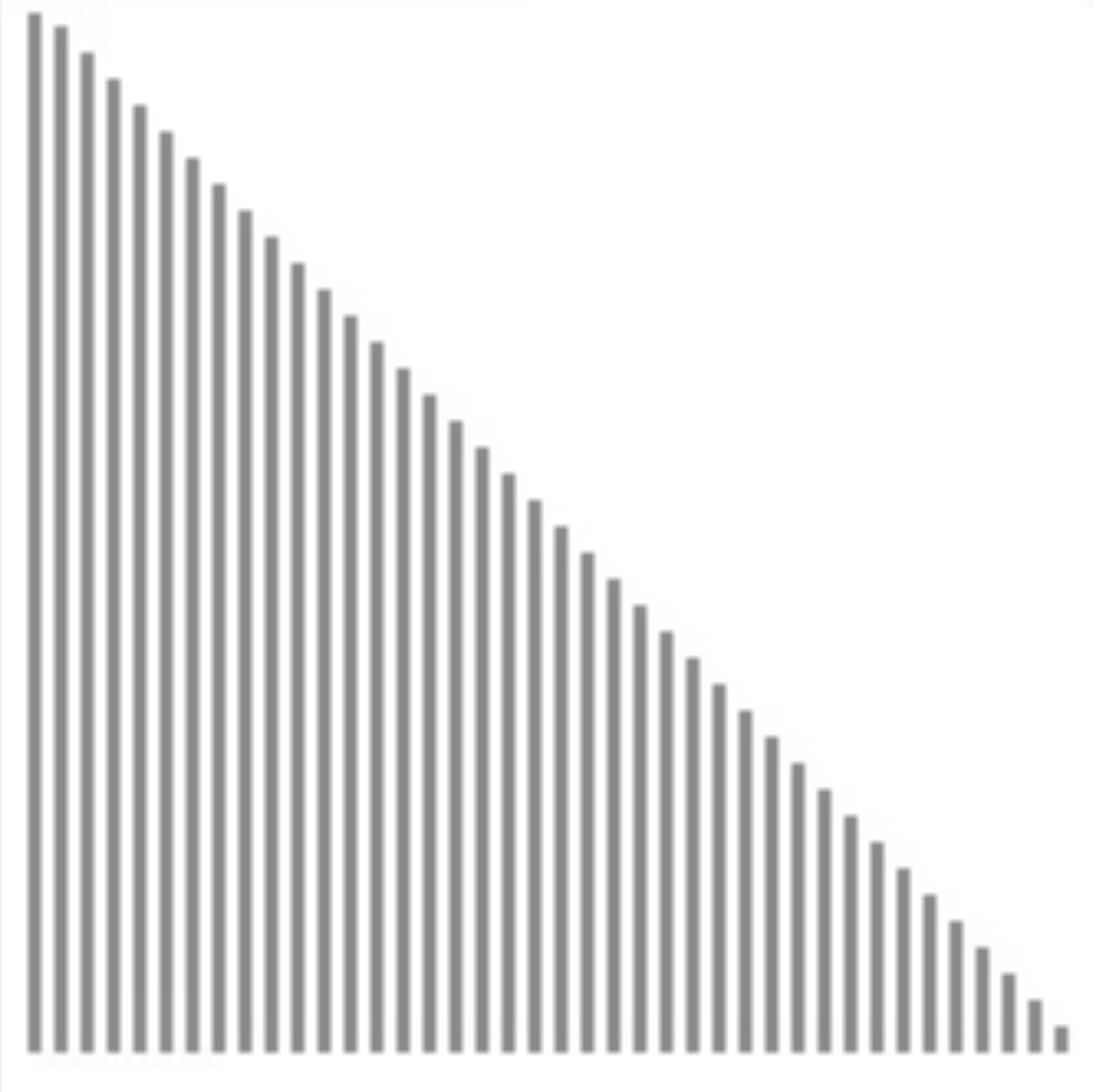


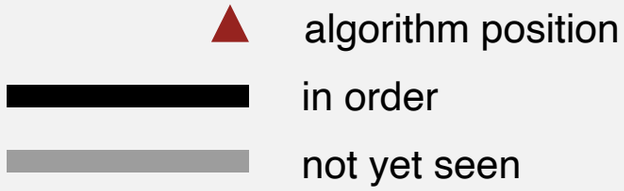


http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 reverse-sorted items

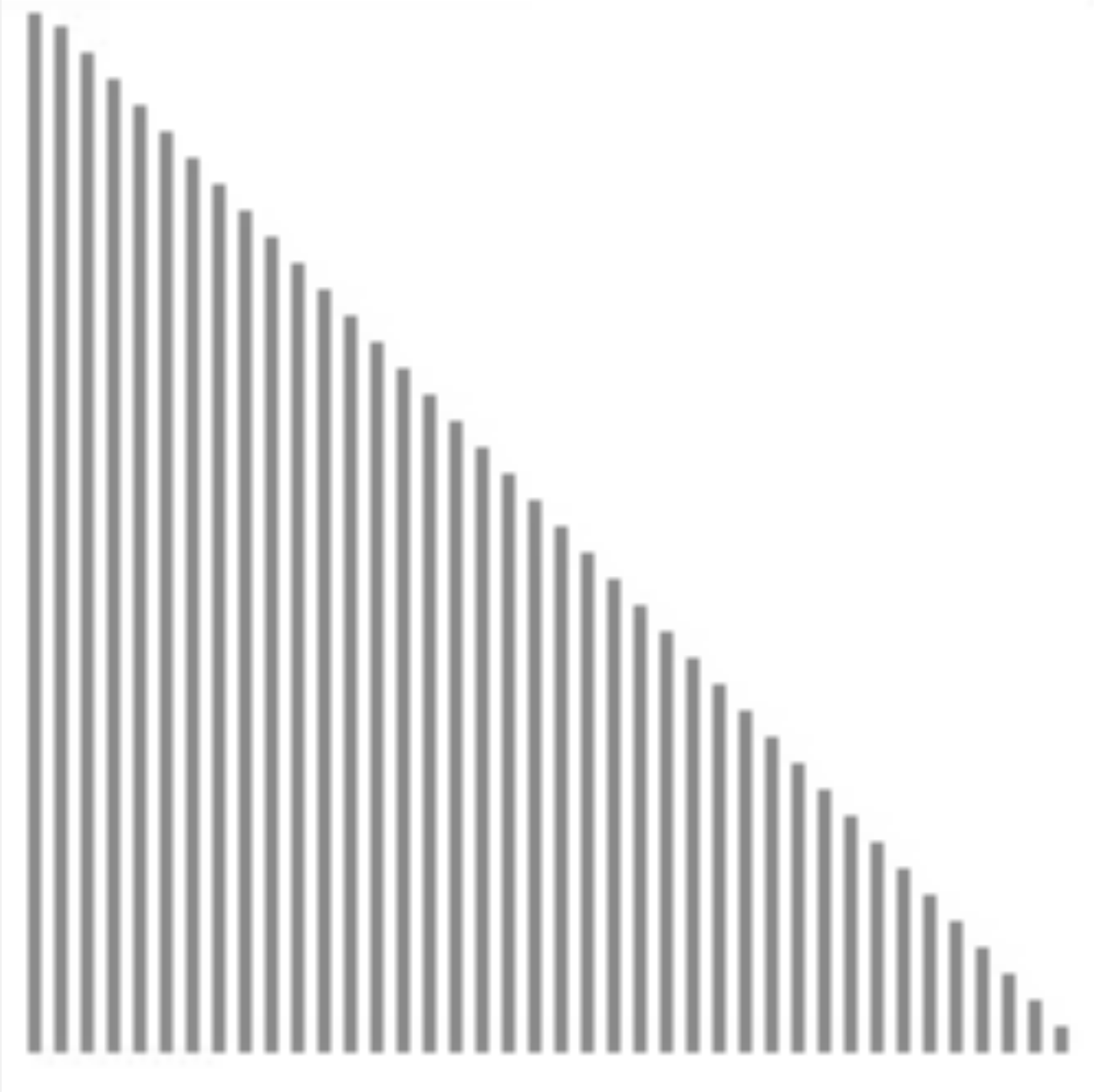


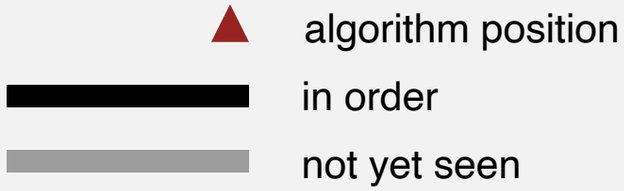


http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 reverse-sorted items

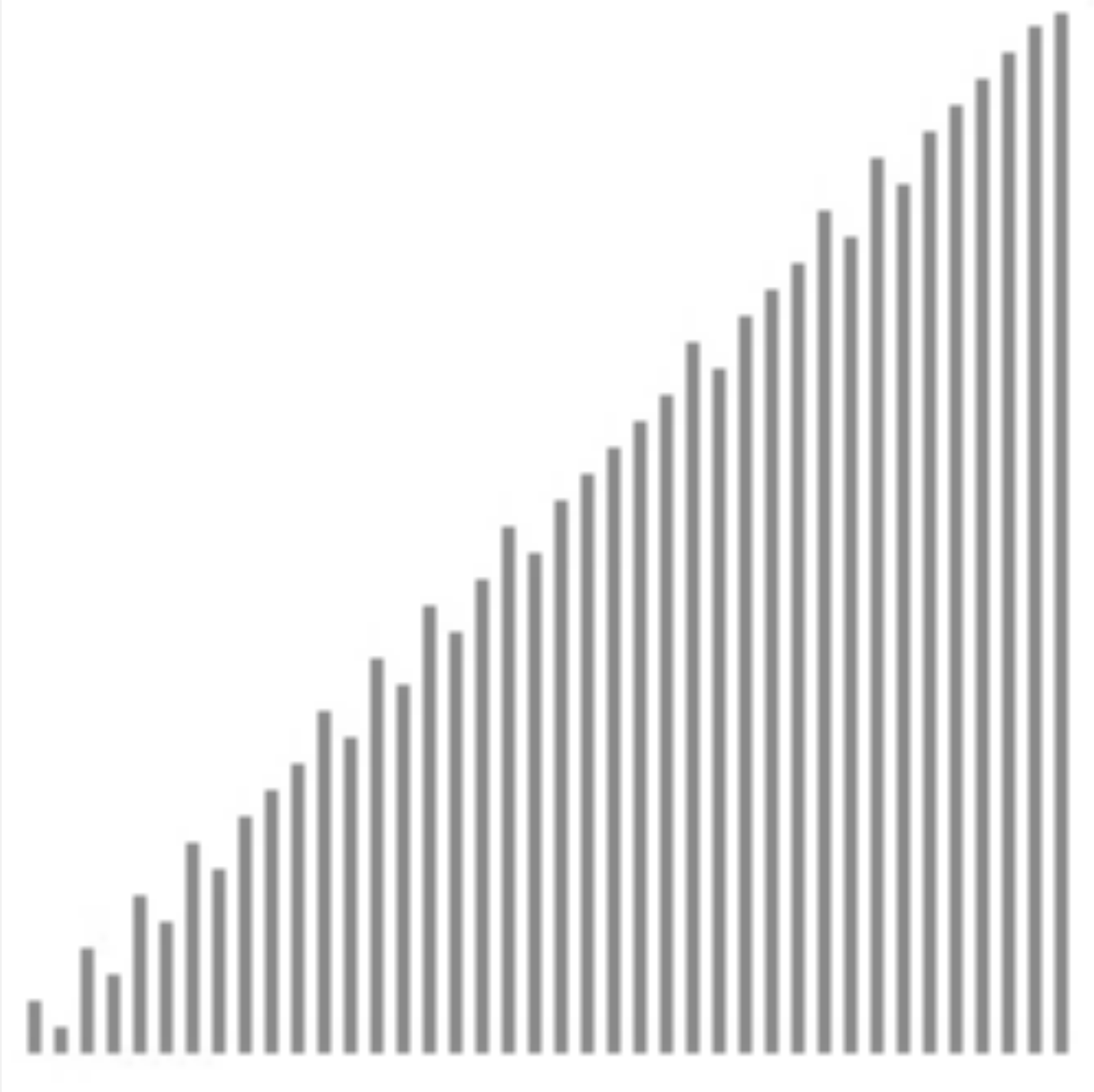


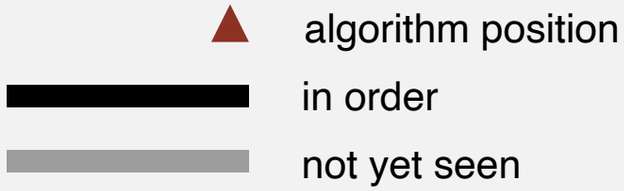


http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 partially-sorted items

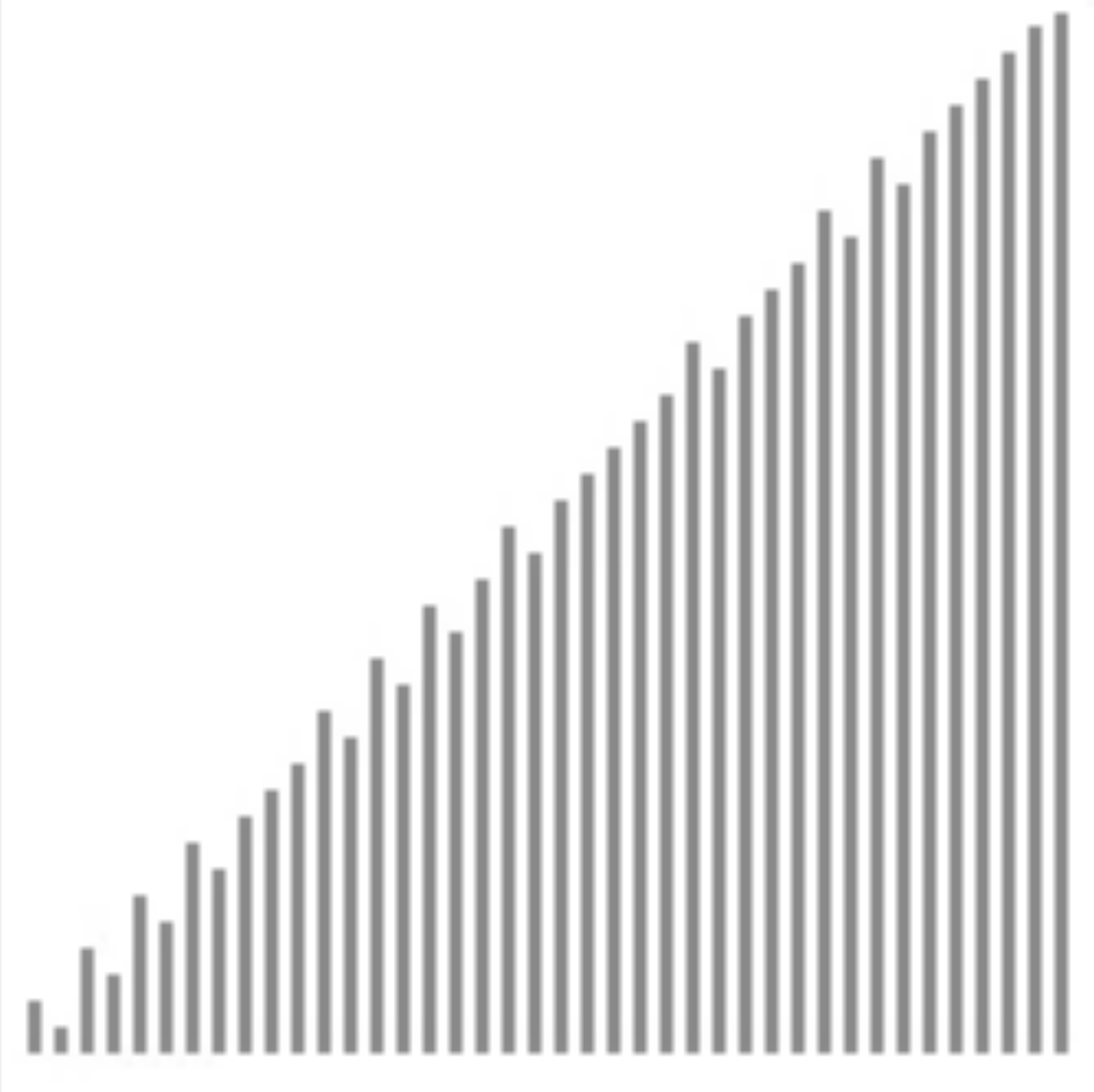


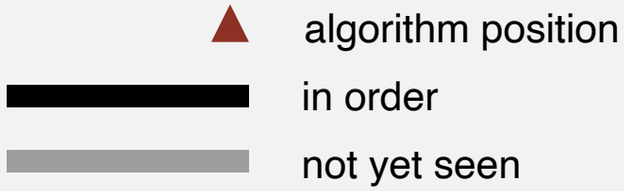


http://www.sorting-algorithms.com/insertion-sort

Insertion sort: animation

40 partially-sorted items





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

Insertion sort: mathematical analysis

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

Pf. Expect each entry to move halfway back.

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

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

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

Insertion sort: 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: 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

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

http://algs4.cs.princeton.edu

2.1 Elementary Sorts

- selection sort
- insertion sort
- shellsort

Shellsort overview

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

an h-sorted array is h interleaved sorted subsequences

h = 4

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

Shellsort overview

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

an h-sorted array is h interleaved sorted subsequences

h = 4

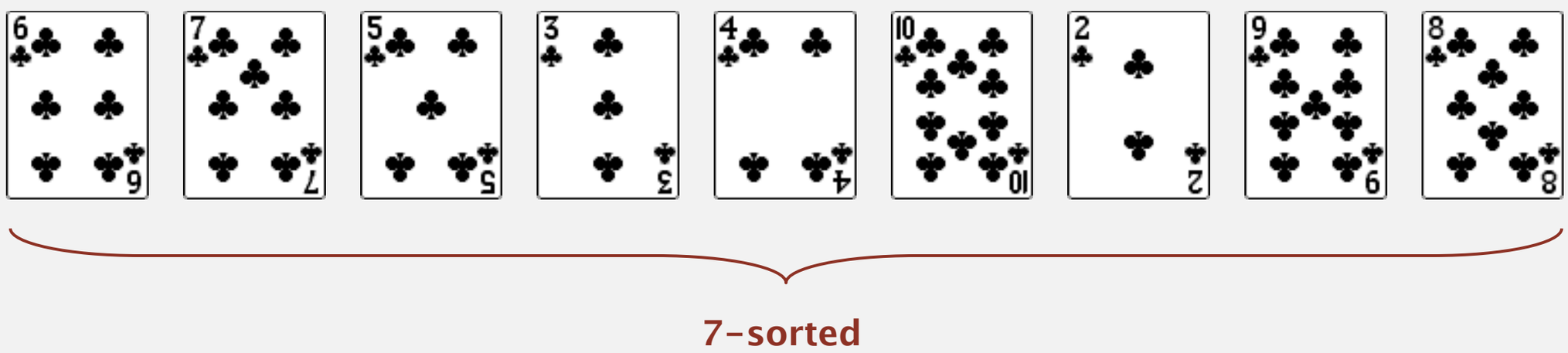
L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
L				M				P				T			
	E				H				S				S		
		E				L				O				X	
			A				E				L				R

Shellsort. [Shell 1959] h-sort 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

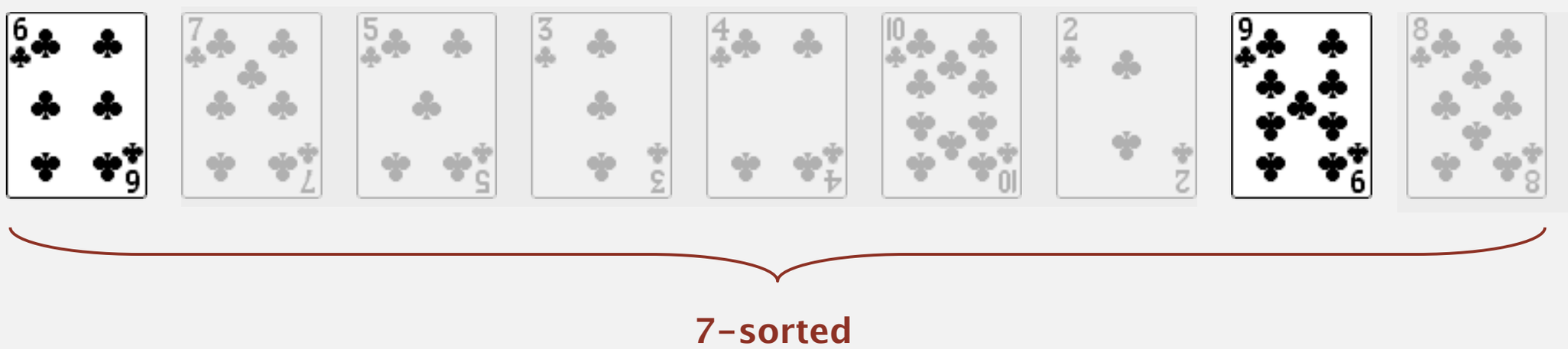
h-sorting demo

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



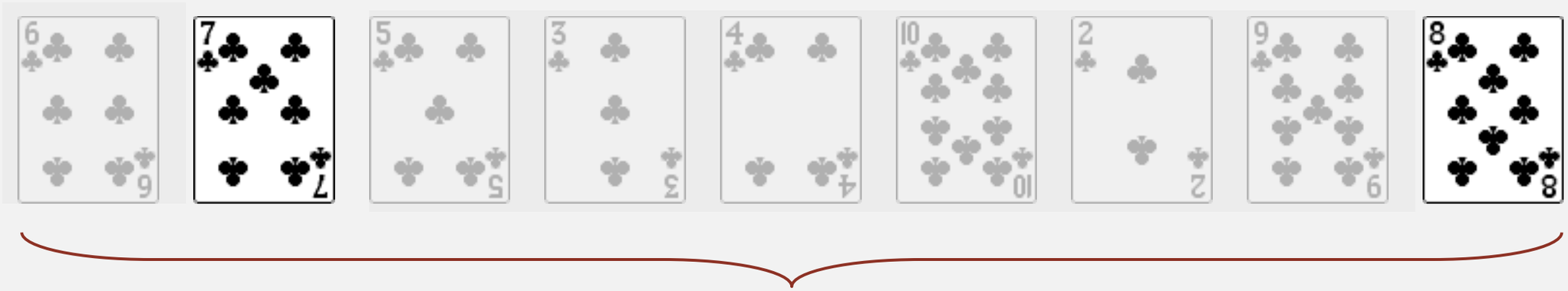
h-sorting demo

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.



7-sorted

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

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

h-sorting

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

3-sorting an array

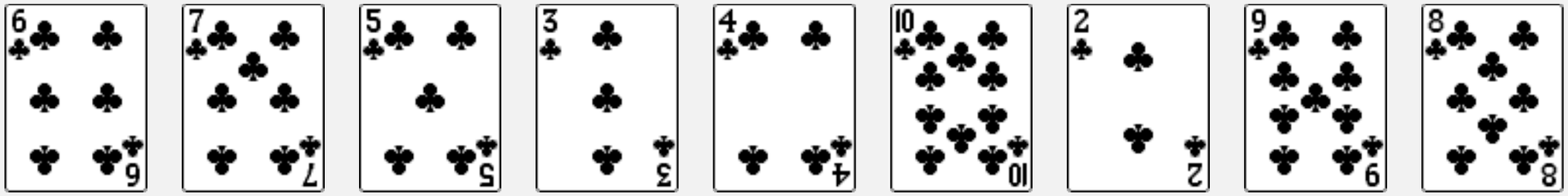
M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

Why insertion sort?

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

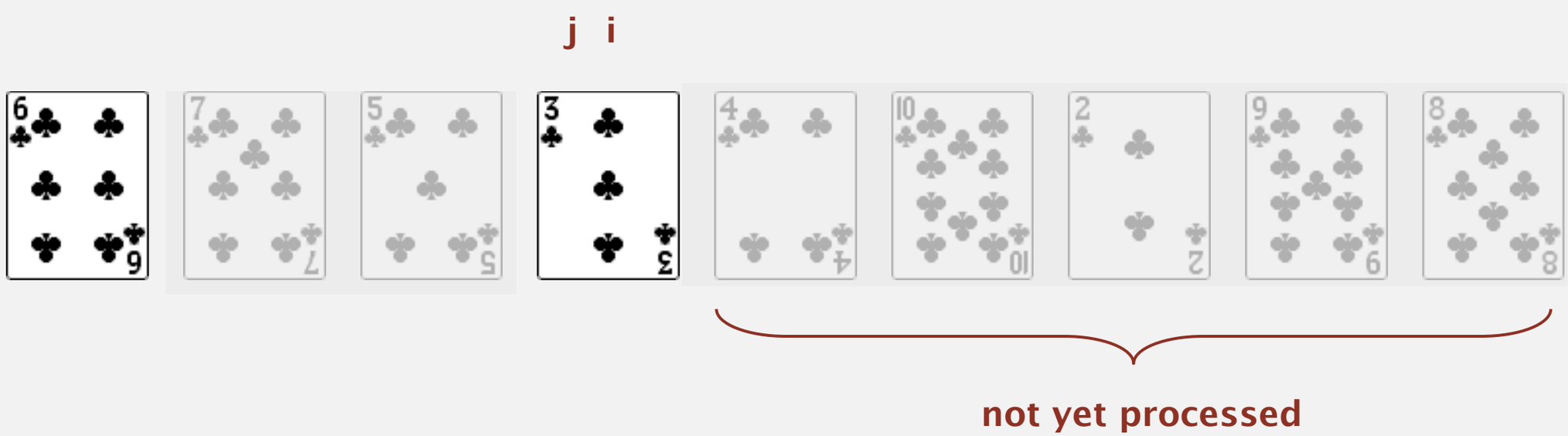
h-sorting demo

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



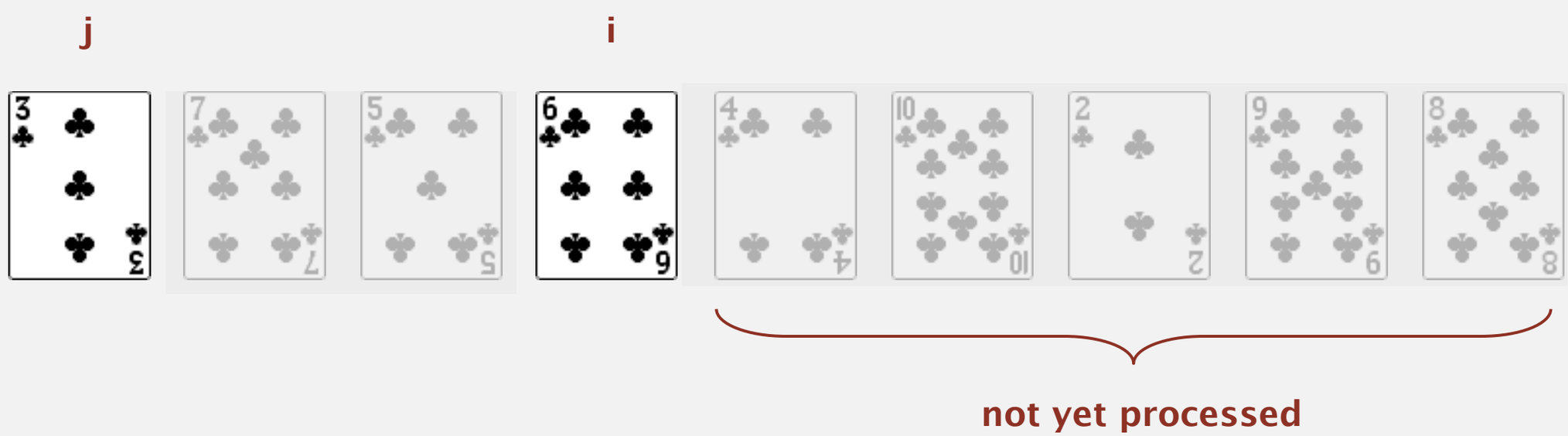
h-sorting demo

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



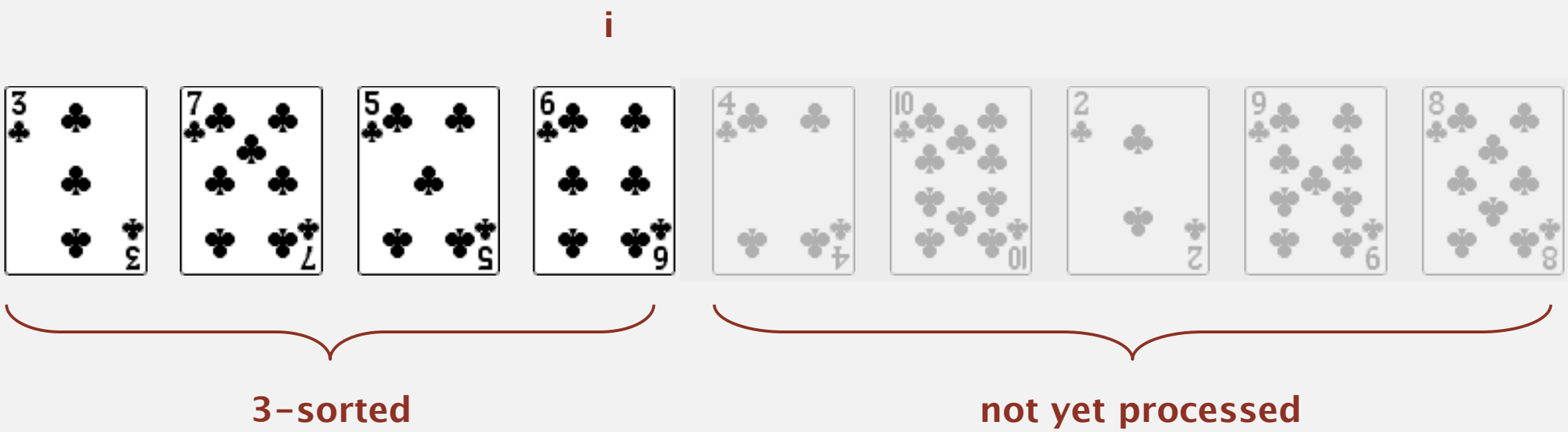
h-sorting demo

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



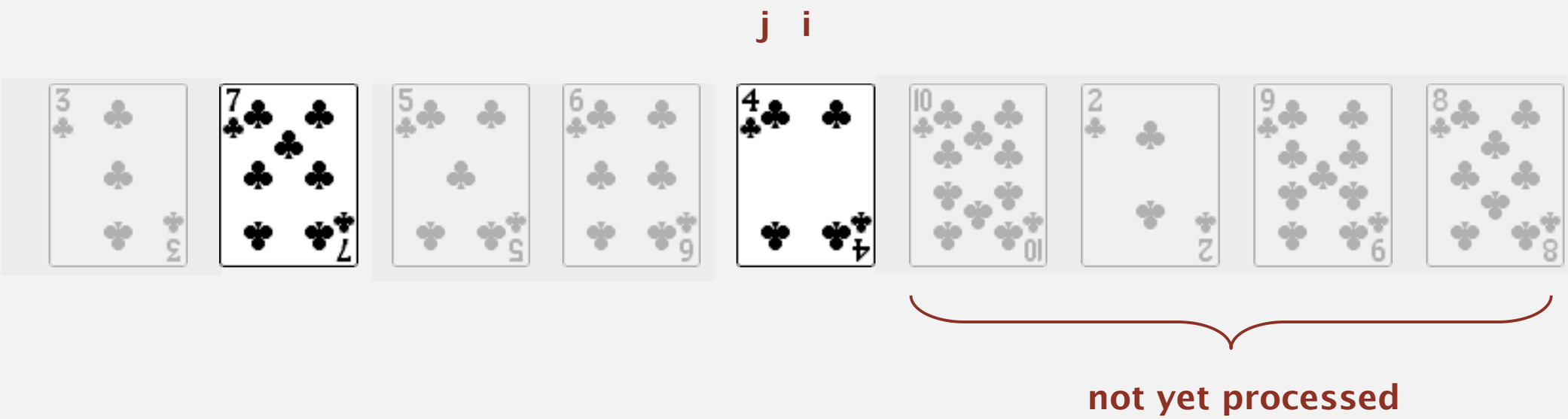
h-sorting demo

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



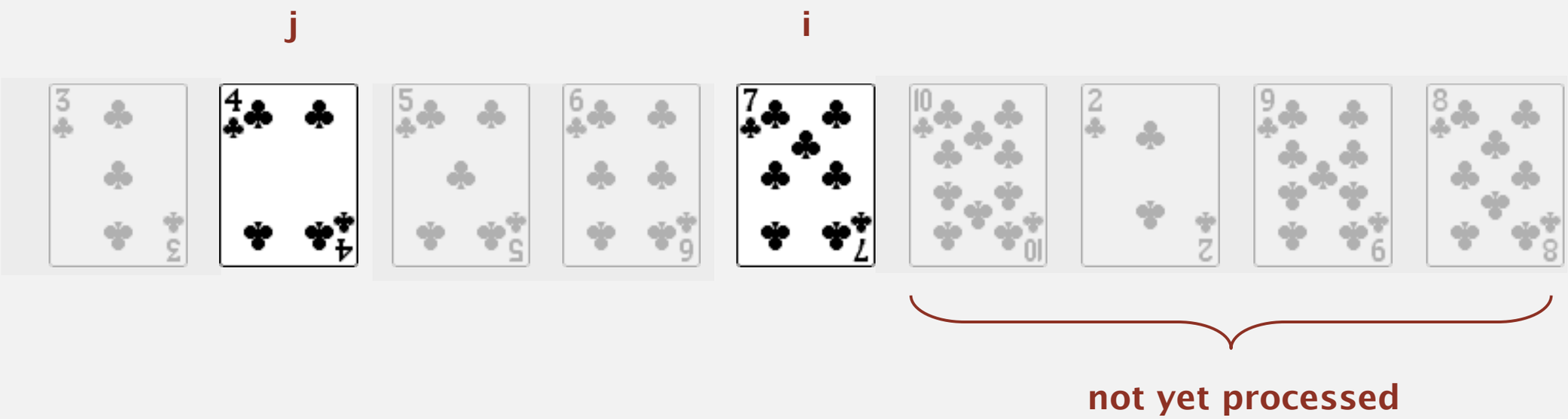
h-sorting demo

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



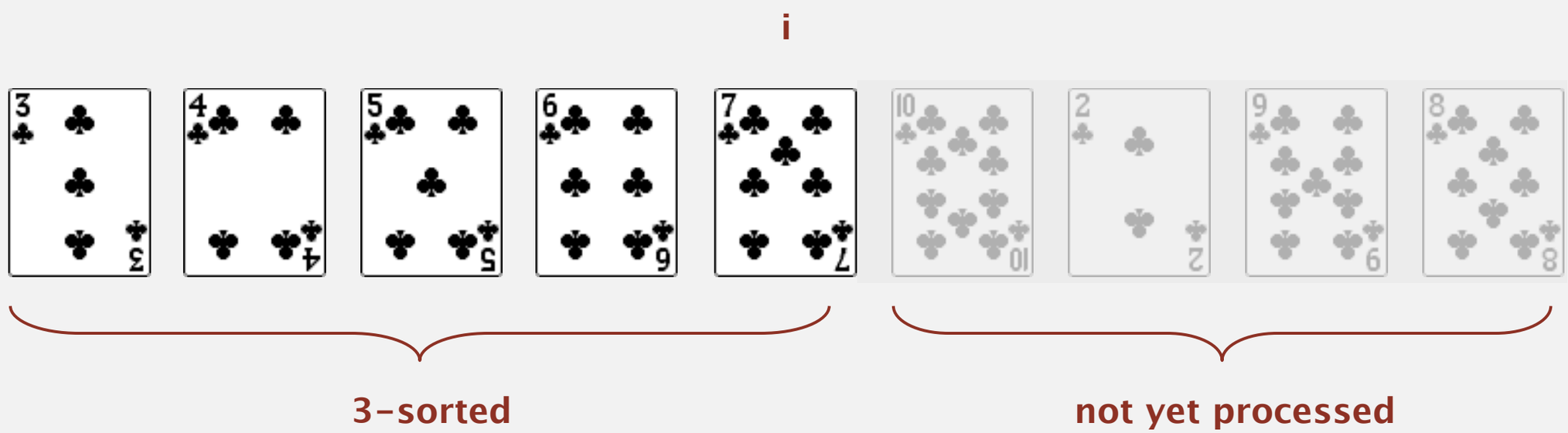
h-sorting demo

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



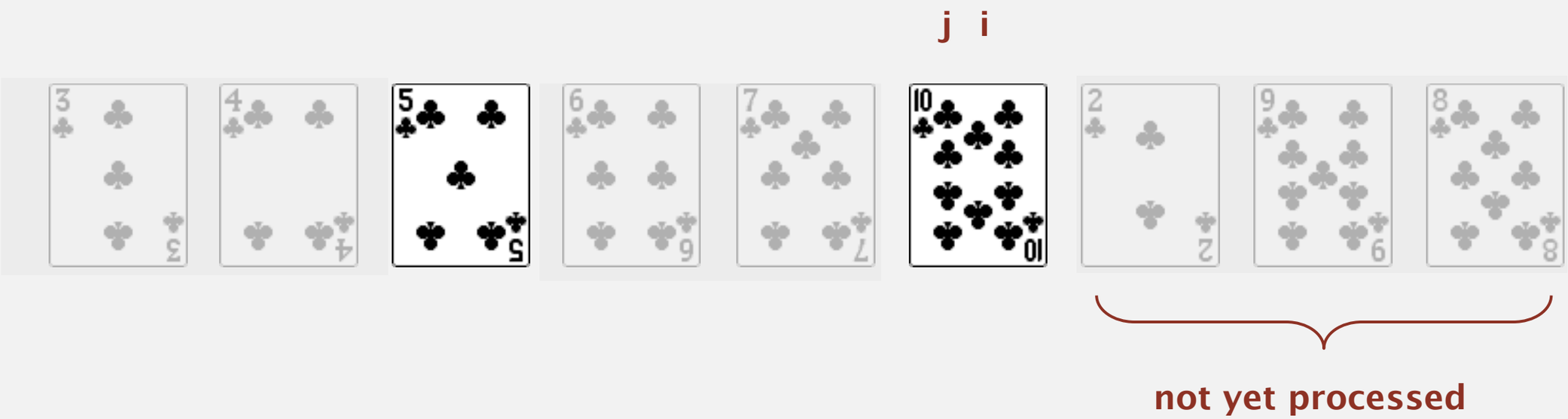
h-sorting demo

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



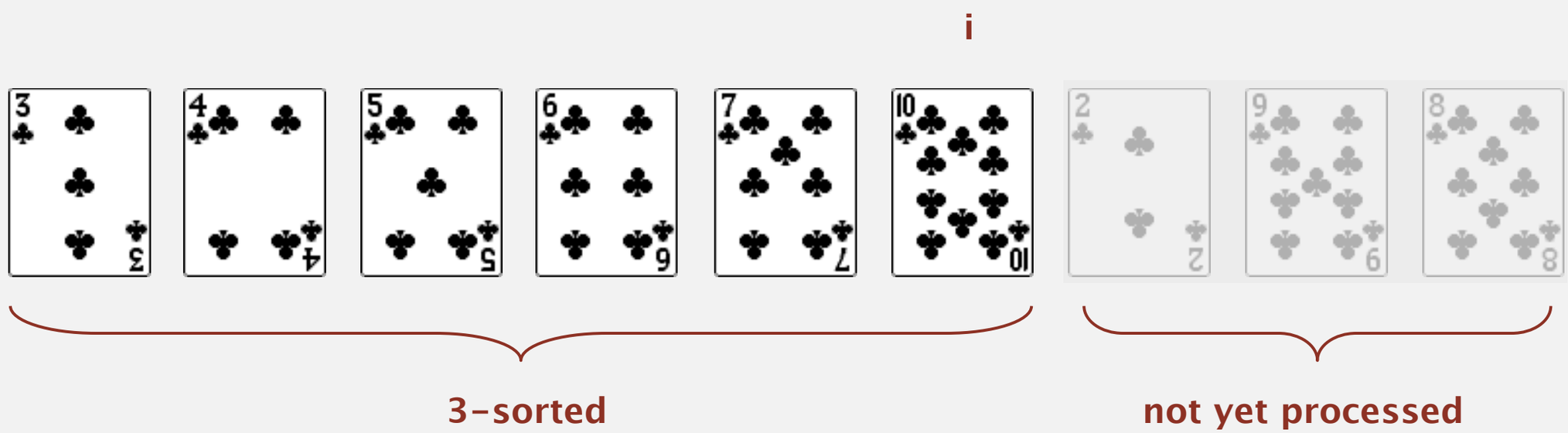
h-sorting demo

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



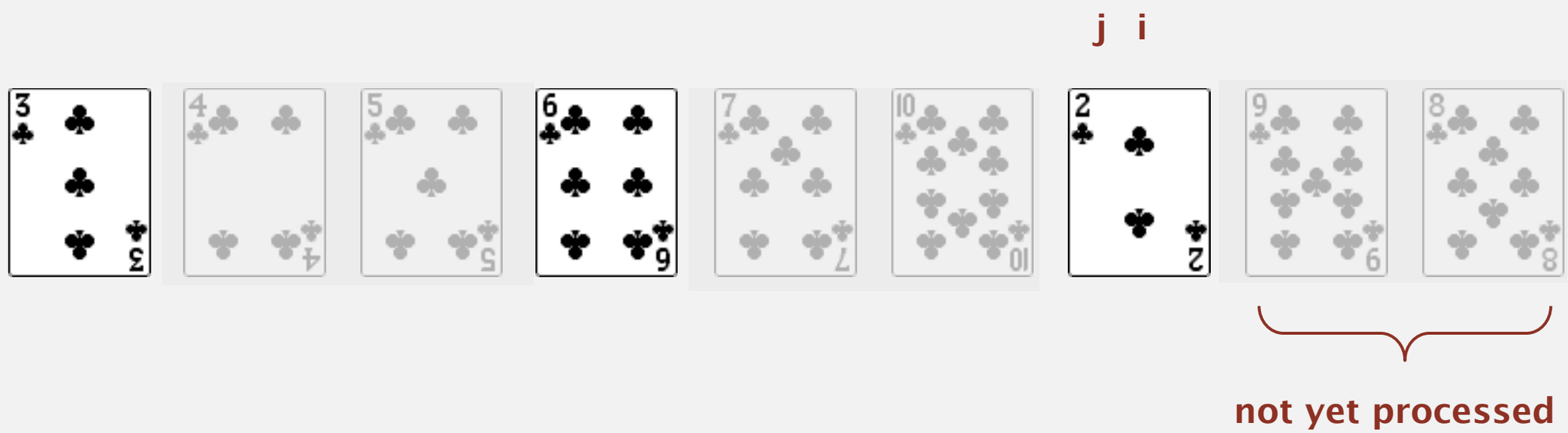
h-sorting demo

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



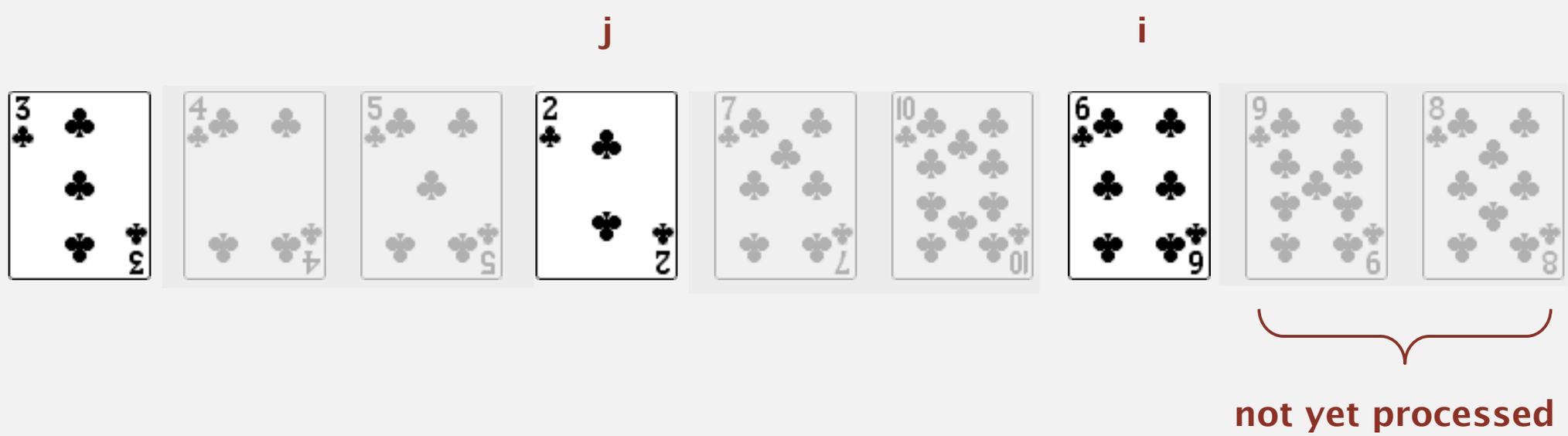
h-sorting demo

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



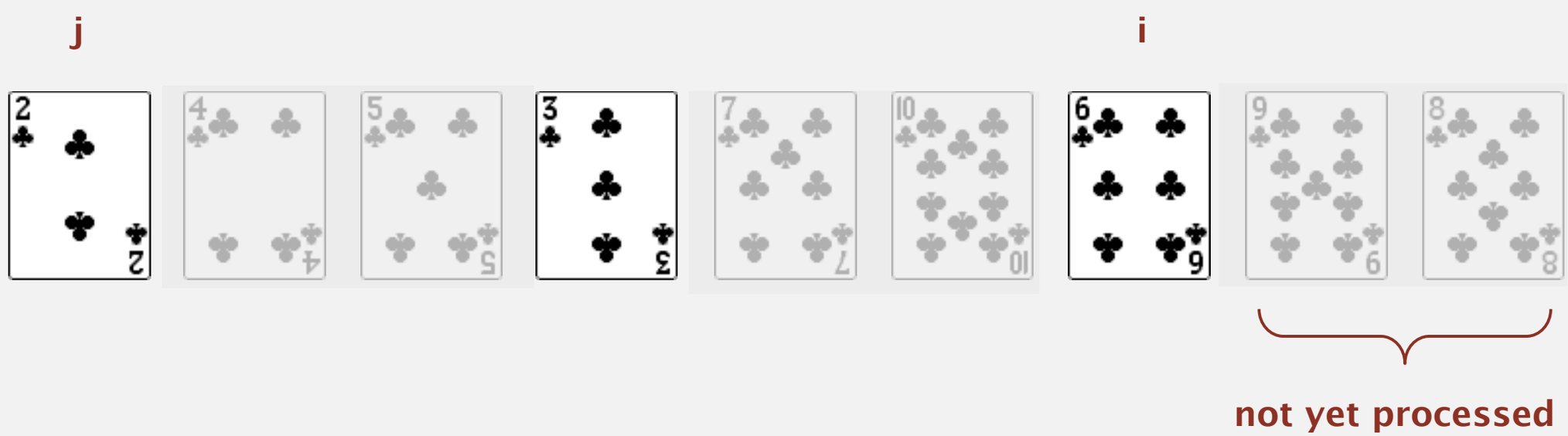
h-sorting demo

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



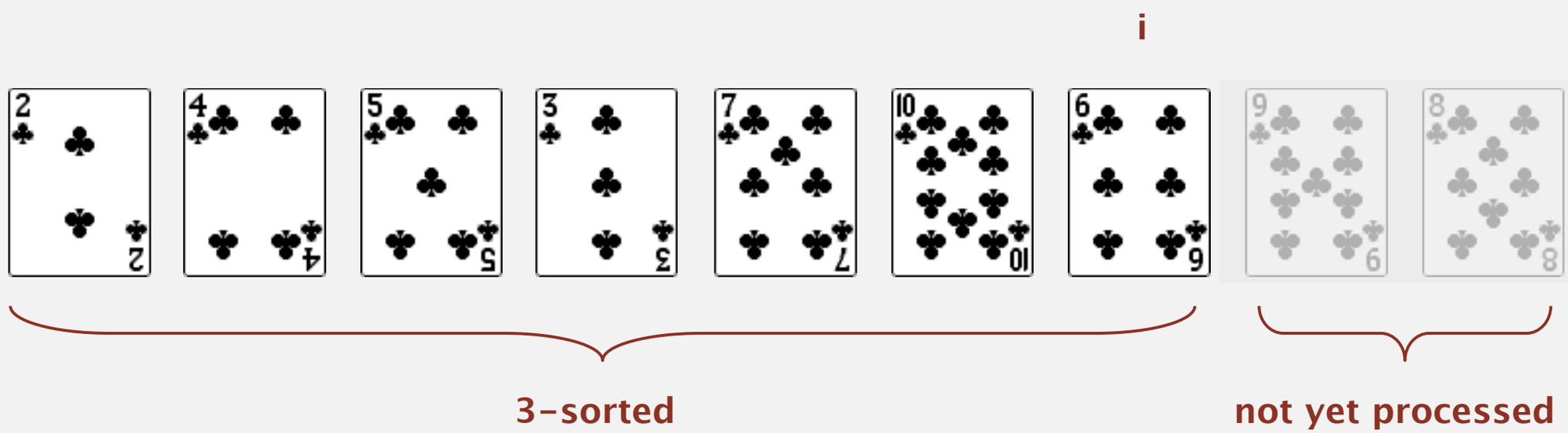
h-sorting demo

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



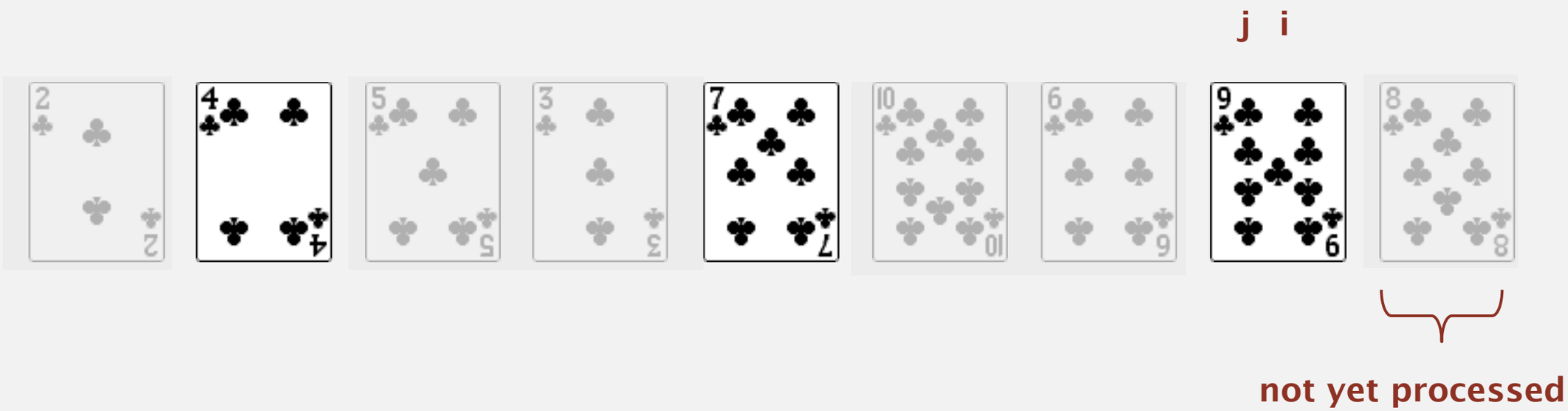
h-sorting demo

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



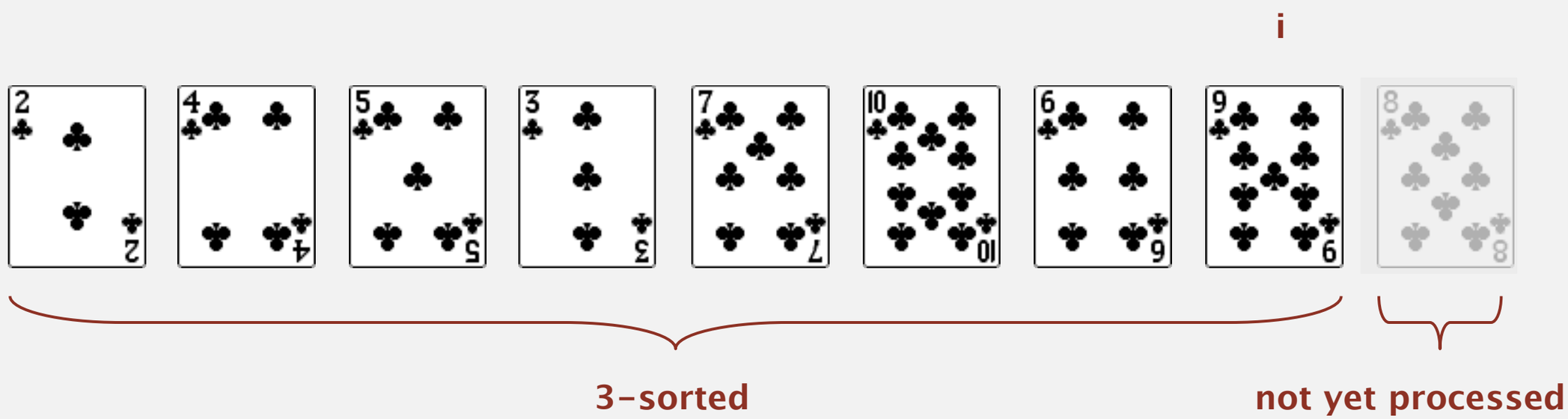
h-sorting demo

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

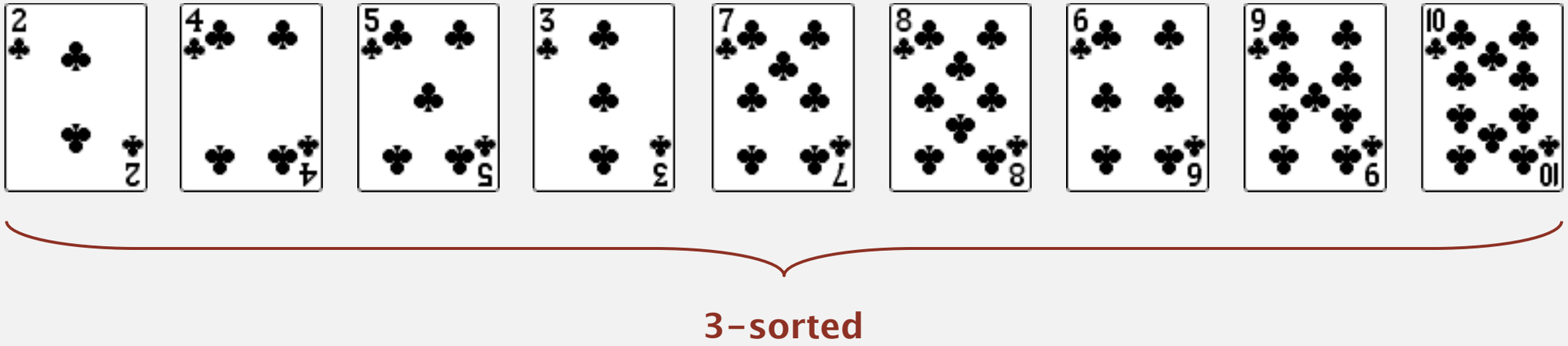
In iteration i, swap a[i] with each larger entry h positions to its left.

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.

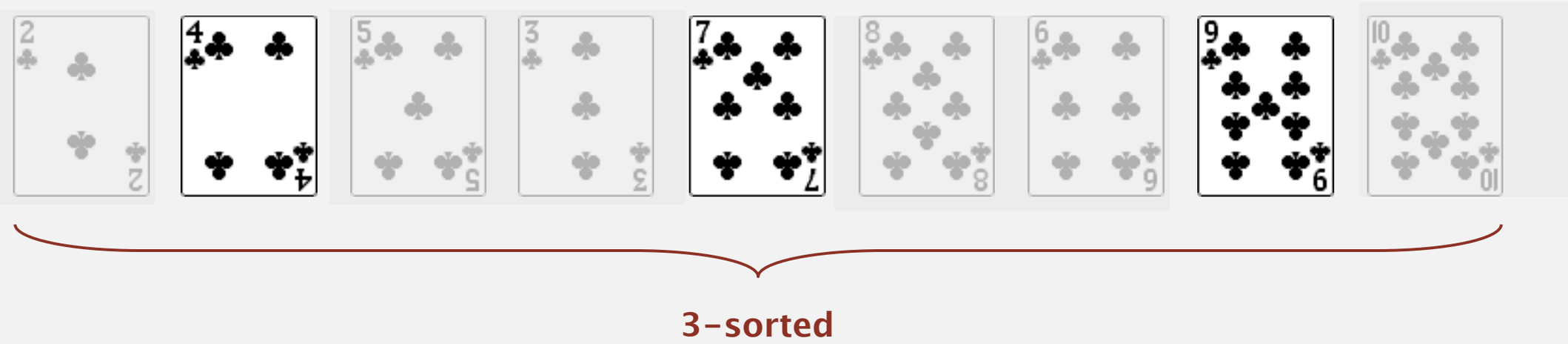


h-sorting demo

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

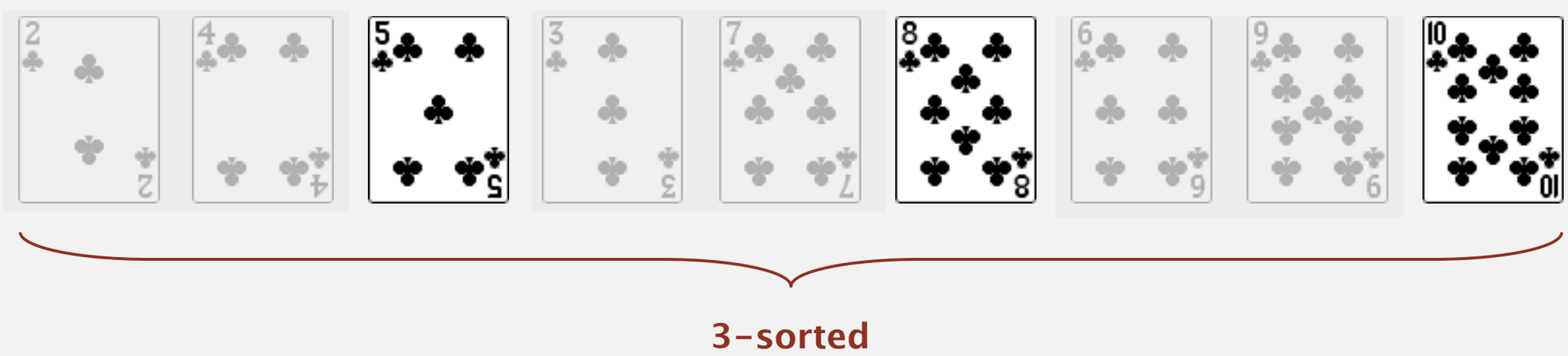
h-sorting demo

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



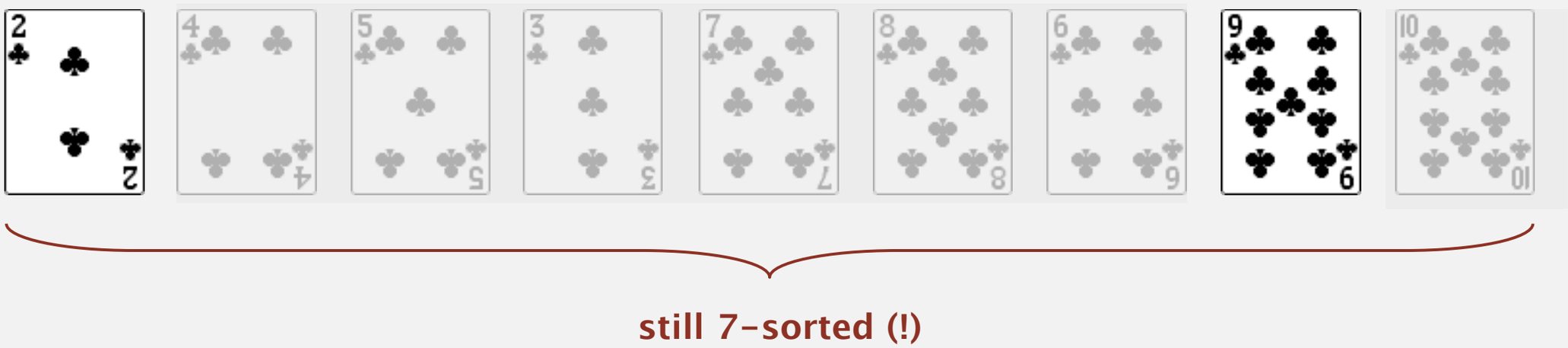
h-sorting demo

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



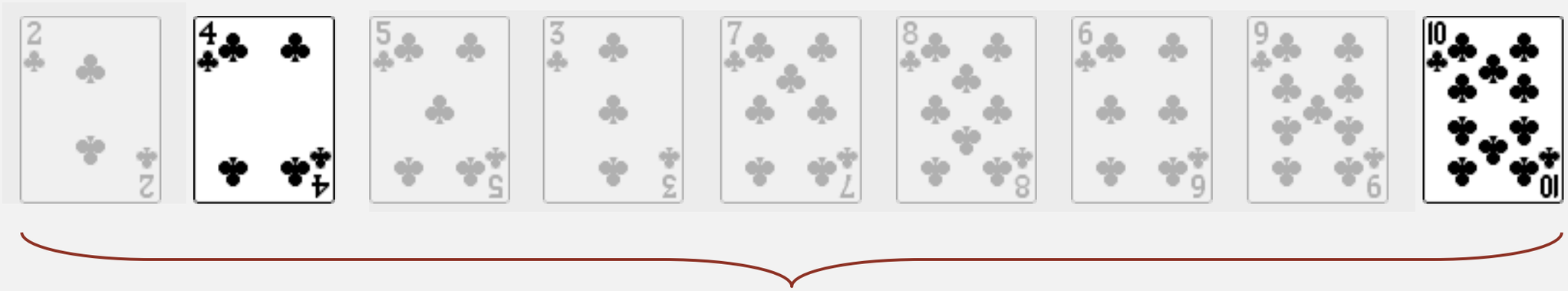
h-sorting demo

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.



still 7-sorted (!)

Shellsort example: increments 7, 3, 1

input

S O R T E X A M P L E

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

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

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

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: Java implementation

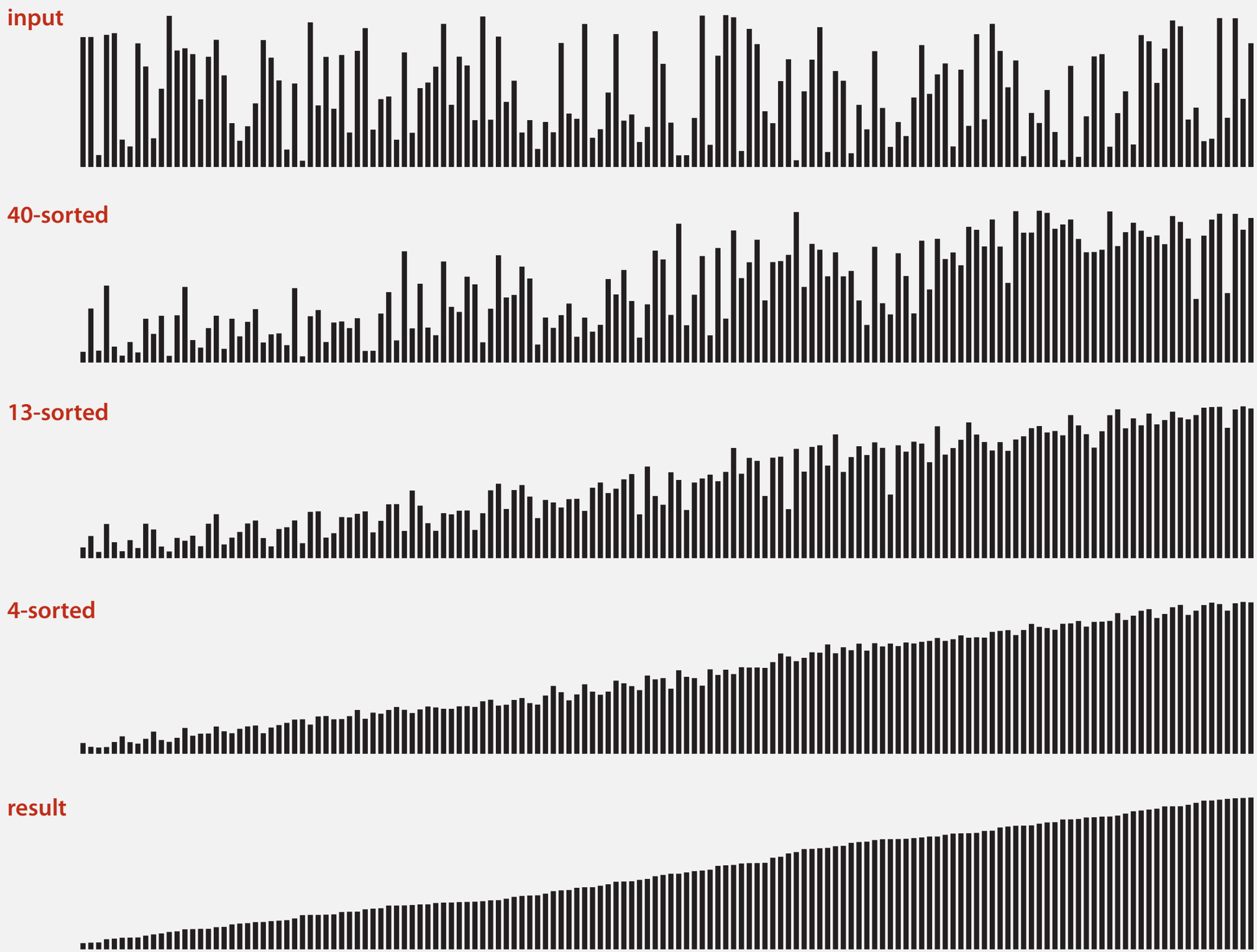
```
public class Shell {
  public static void sort(Comparable[] a) {
    int N = a.length;

    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

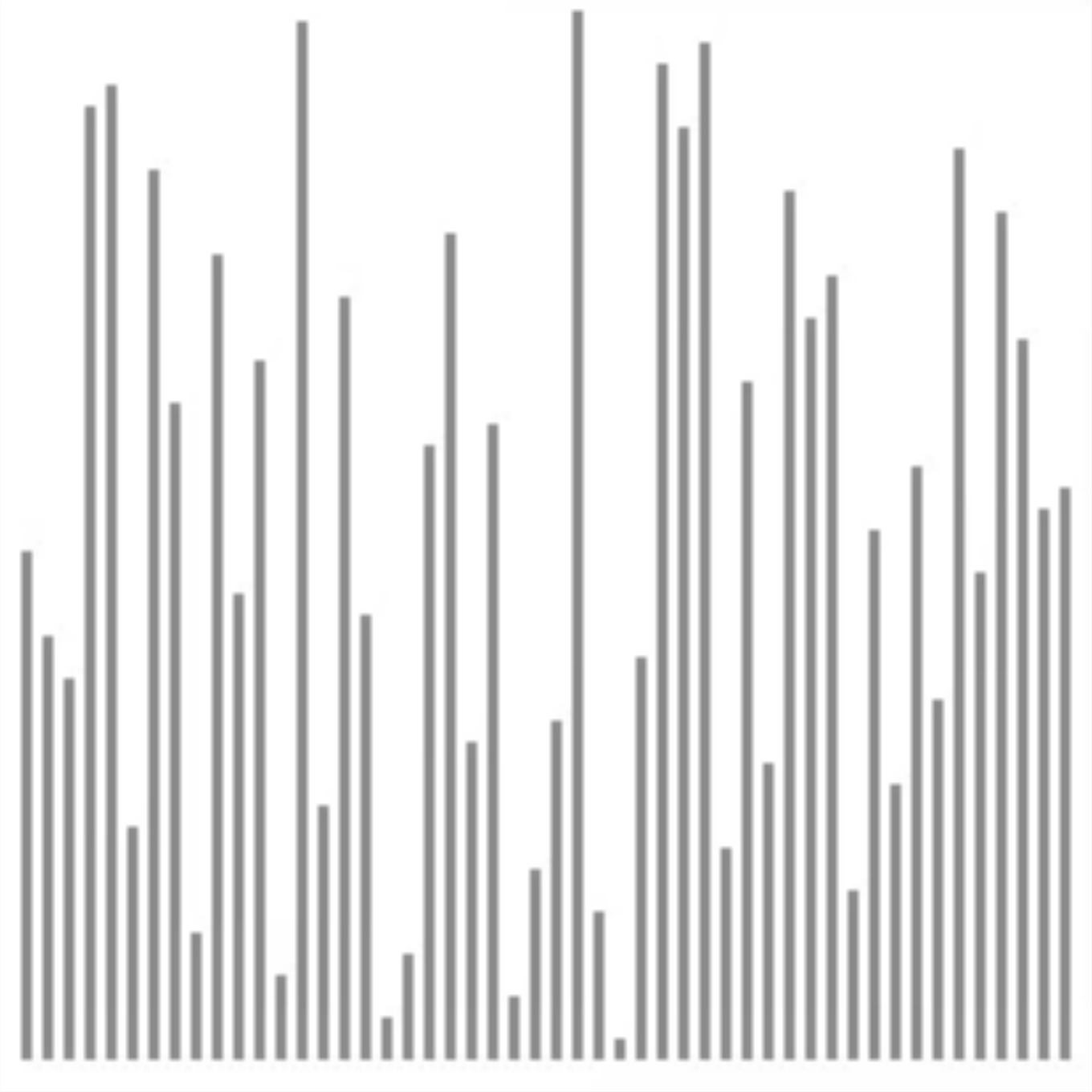
move to next increment

Shellsort: visual trace

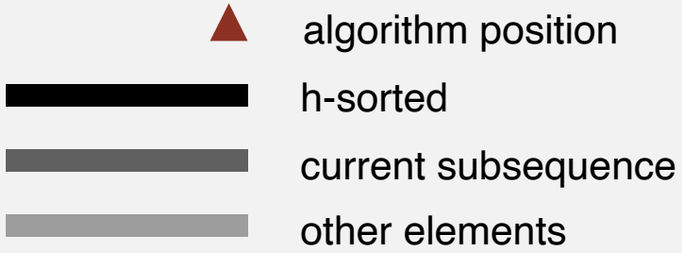


Shellsort: animation

50 random items

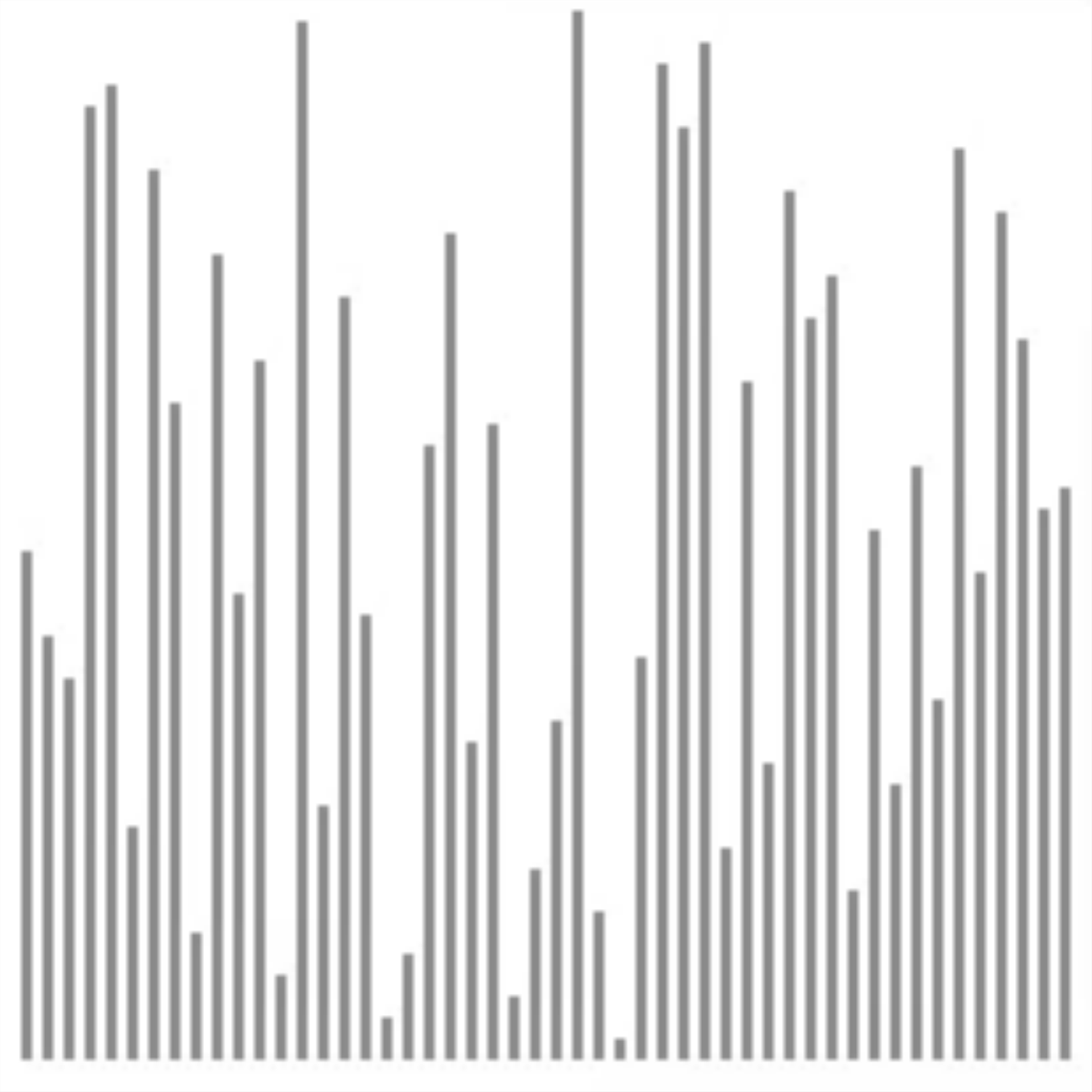


http://www.sorting-algorithms.com/shell-sort

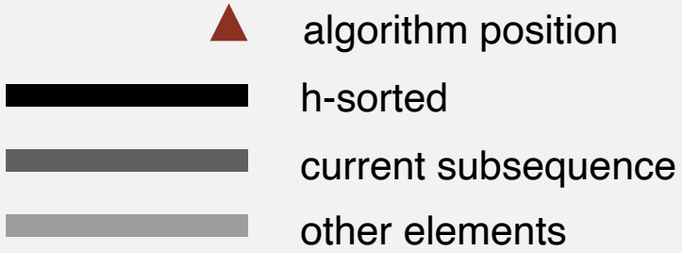


Shellsort: animation

50 random items

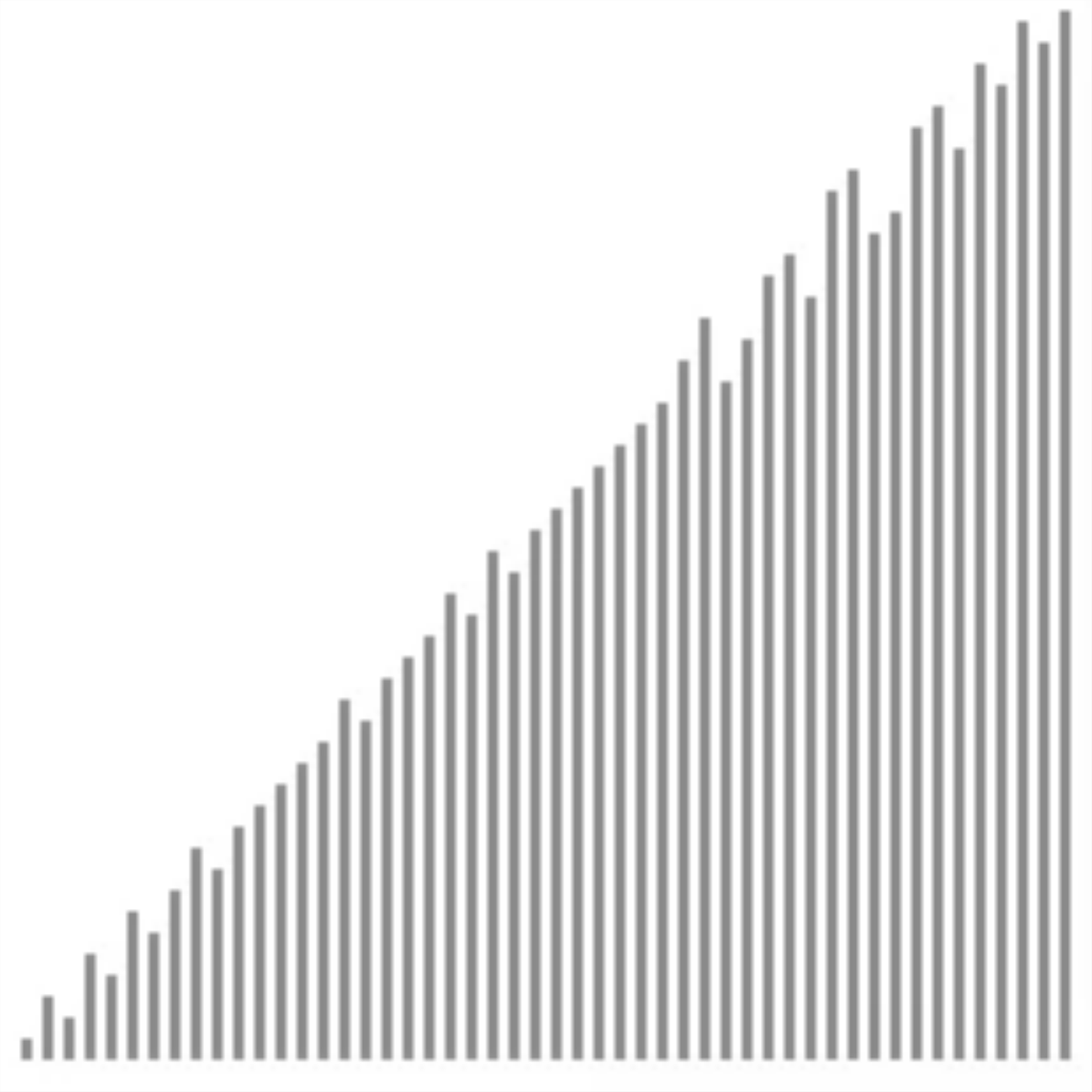


http://www.sorting-algorithms.com/shell-sort

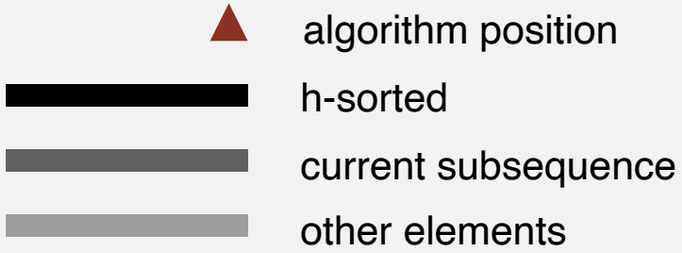


Shellsort: animation

50 partially-sorted items

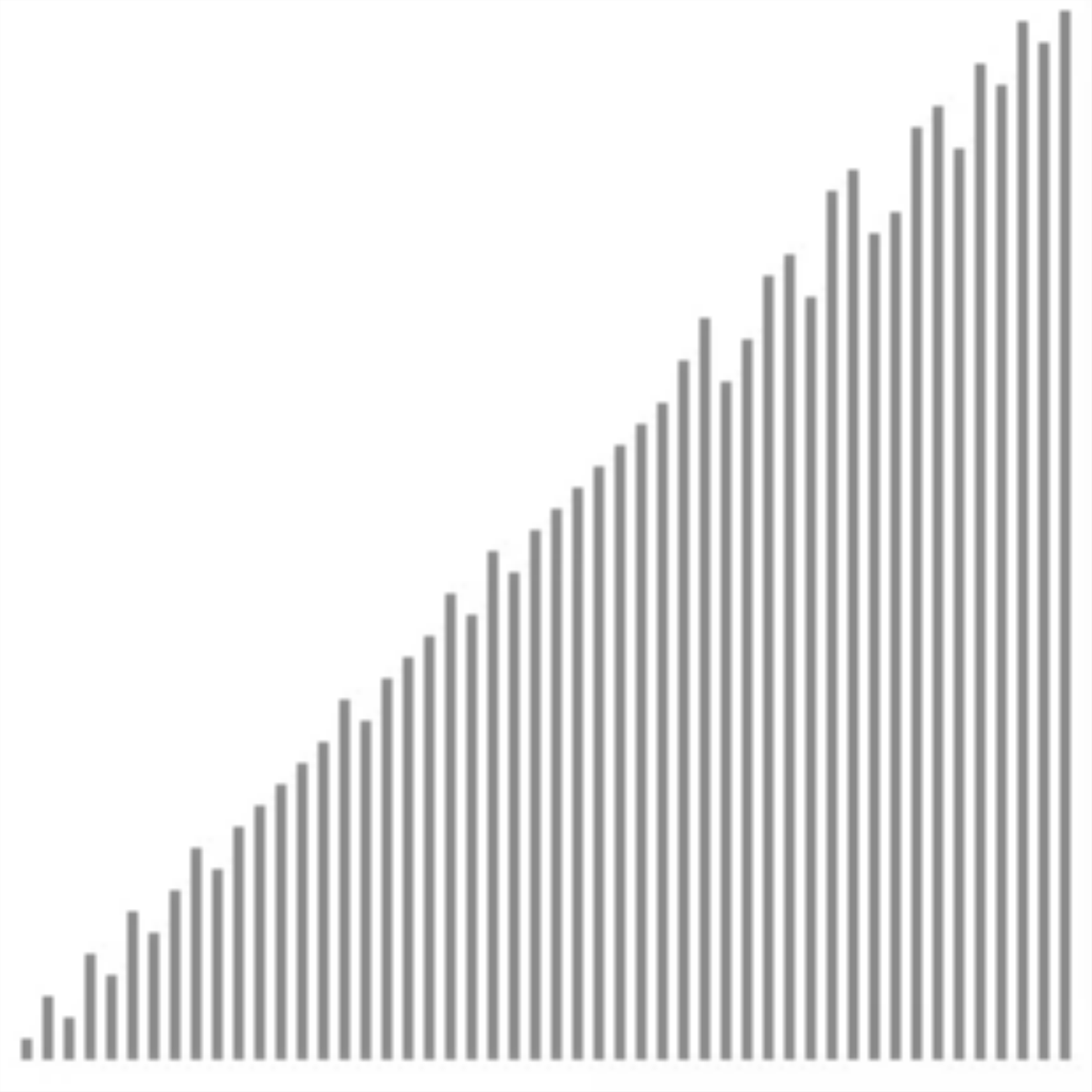


http://www.sorting-algorithms.com/shell-sort

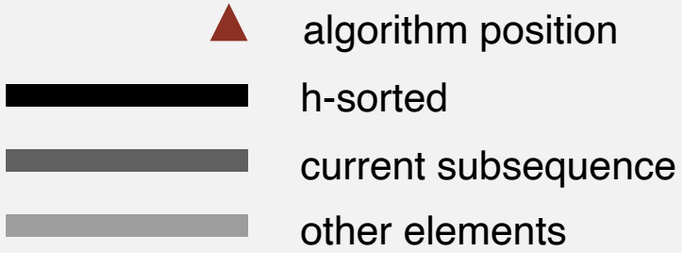


Shellsort: animation

50 partially-sorted items



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



Shellsort: which increment sequence to use?

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

No.

Shellsort: which increment sequence to use?

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

No.

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

Maybe.

Shellsort: which increment sequence to use?

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

No.

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

Maybe.

→ 3x + 1. 1, 4, 13, 40, 121, 364, …

OK. Easy to compute.

Shellsort: which increment sequence to use?

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

No.

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

Maybe.

→ 3x + 1. 1, 4, 13, 40, 121, 364, …

OK. Easy to compute.

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: intuition

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

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
A E L E O P M S X R T

still 7-sorted

Shellsort: intuition

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

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
A E L E O P M S X R T

still 7-sorted

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

Shellsort: analysis

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

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

Shellsort: analysis

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

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

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

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.

Why are we interested in 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

Why are we interested in 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?

Why are we interested in 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.

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)	$N \log N$	?	$N^{3/2}$

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

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).

http://algs4.cs.princeton.edu

2.2 MERGESORT

- mergesort
- bottom-up mergesort
- stability

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [this lecture]

...

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [this lecture]

Quicksort. [next lecture]

http://algs4.cs.princeton.edu

2.2 Mergesort

- mergesort
- bottom-up mergesort
- stability

Mergesort

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

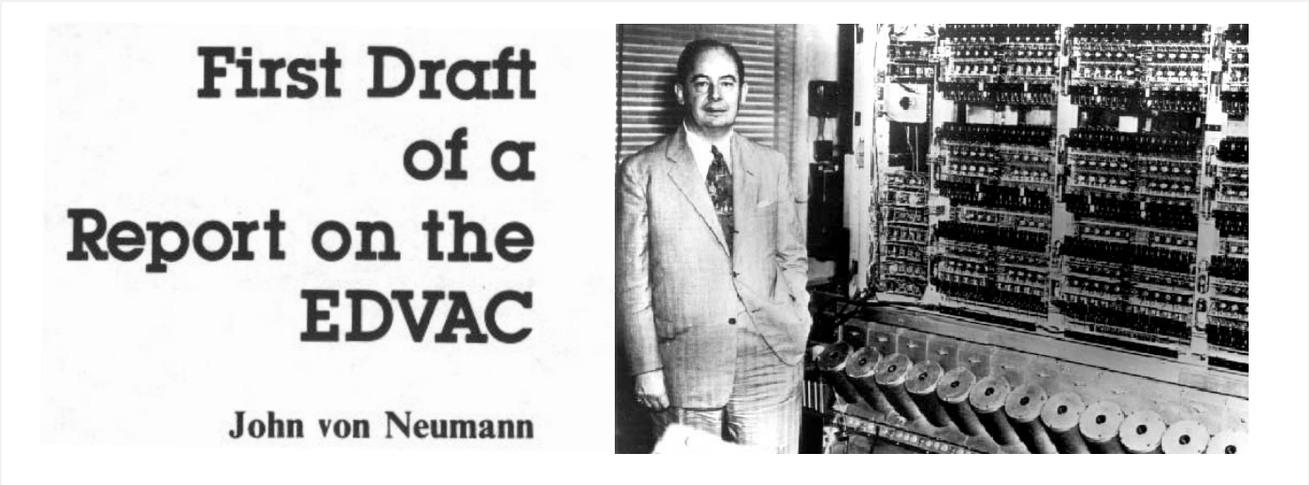
Mergesort

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

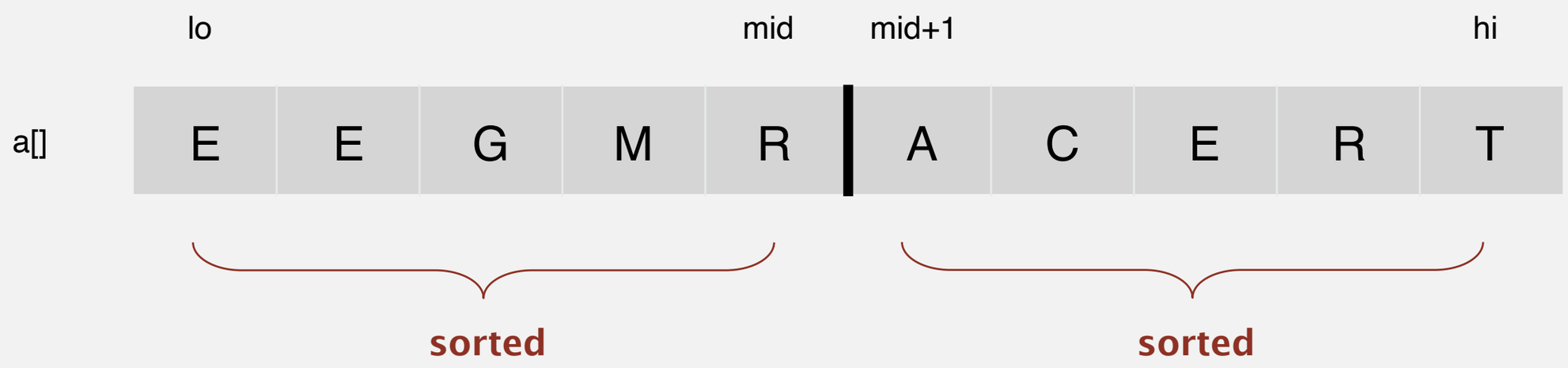
input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview



Merging demo

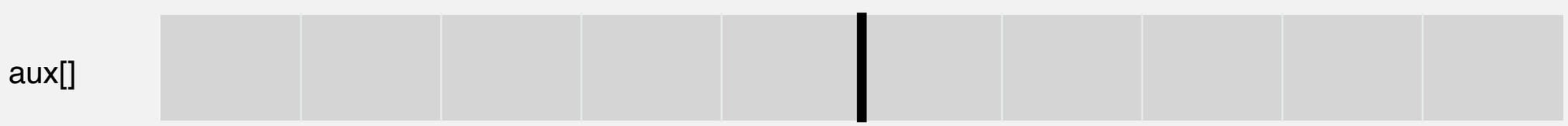
Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].



Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

copy to auxiliary array



Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

	lo				mid	mid+1				hi
a[]	E	E	G	M	R	A	C	E	R	T

copy to auxiliary array

aux[]	E	E	G	M	R	A	C	E	R	T

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	E	E	G	M	R	A	C	E	R	T

aux[]	E	E	G	M	R	A	C	E	R	T

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	E	E	G	M	R	A	C	E	R	T
	k									

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	E	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i					j				

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	E	G	M	R	A	C	E	R	T
	k									

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i						j			

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i							j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	G	M	R	A	C	E	R	T
		k								

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i							j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	M	R	A	C	E	R	T
			k							

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
	i							j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	M	R	A	C	E	R	T
			k							

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	M	R	A	C	E	R	T
			k							

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
		i						j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i					j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

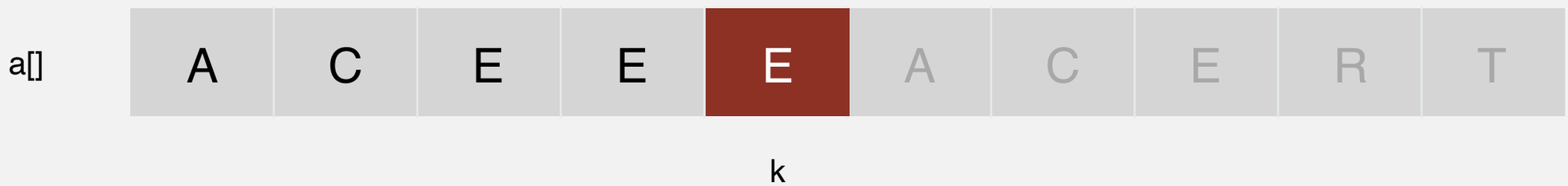
a[]	A	C	E	E	R	A	C	E	R	T
				k						

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i					j		

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].



compare minimum in each subarray

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	A	C	E	R	T
					k					

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	A	C	E	R	T
					k					

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	C	E	R	T
						k				

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
			i						j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	C	E	R	T
							k			

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	C	E	R	T
							k			

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	E	R	T
							k			

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
				i					j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	E	R	T
								k		

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
					i				j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	E	R	T
								k		

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
					i				j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
								k		

compare minimum in each subarray

aux[]	E	E	G	M	R	A	C	E	R	T
					i				j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i			j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
									k	

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i			j	

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

a[]	A	C	E	E	E	G	M	R	R	T
										k

one subarray exhausted, take from other

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].

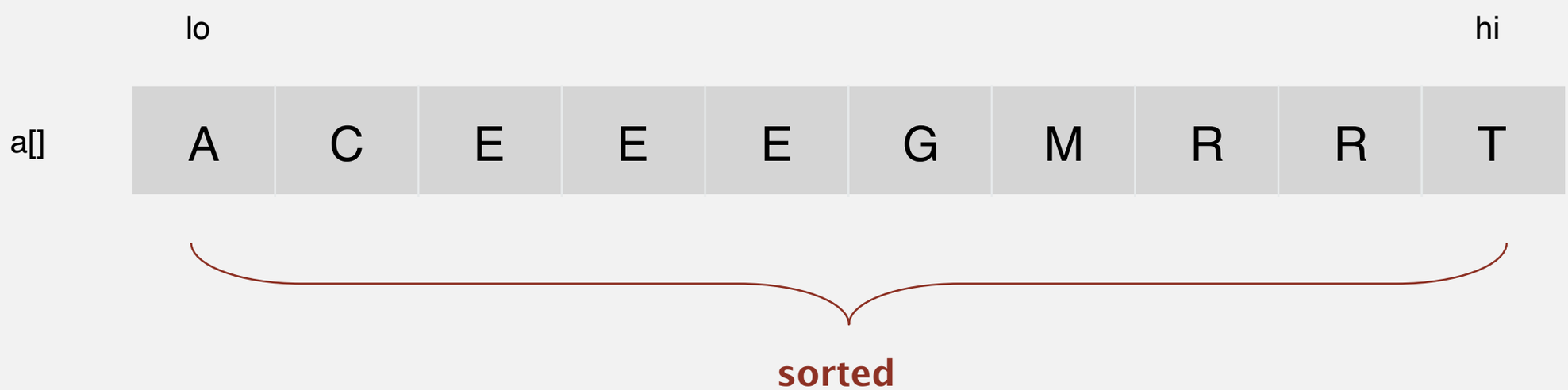
a[]	A	C	E	E	E	G	M	R	R	T
										k

both subarrays exhausted, done

aux[]	E	E	G	M	R	A	C	E	R	T
						i				j

Merging demo

Goal. Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].



Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input		E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
copy		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	A										0	6	E	E	G	M	R	A	C	E	R	T
	1	A	C									0	7	E	E	G	M	R		C	E	R	T
	2	A	C	E								1	7	E	E	G	M	R			E	R	T
	3	A	C	E	E							2	7		E	G	M	R			E	R	T
	4	A	C	E	E	E						2	8			G	M	R			E	R	T
	5	A	C	E	E	E	G					3	8			G	M	R				R	T
	6	A	C	E	E	E	G	M				4	8				M	R				R	T
	7	A	C	E	E	E	G	M	R			5	8					R				R	T
	8	A	C	E	E	E	G	M	R	R		5	9									R	T
	9	A	C	E	E	E	G	M	R	R	T	5	10										T
merged result		A	C	E	E	E	G	M	R	R	T												

Abstract in-place merge trace

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{

  for (int k = lo; k <= hi; k++)
    aux[k] = a[k];

  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++)
  {
    if      (i > mid)              a[k] = aux[j++];
    else if (j > hi)               a[k] = aux[i++];
    else if (less(aux[j], aux[i])) a[k] = aux[j++];
    else                           a[k] = aux[i++];
  }
}
```

copy

merge

Mergesort: Java implementation

```
public class Merge {
  private static void merge(...)
  {  /* as before */  }

  private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
  }

  public static void sort(Comparable[] a) {
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length - 1);
  }
}
```

Mergesort: trace

a[]

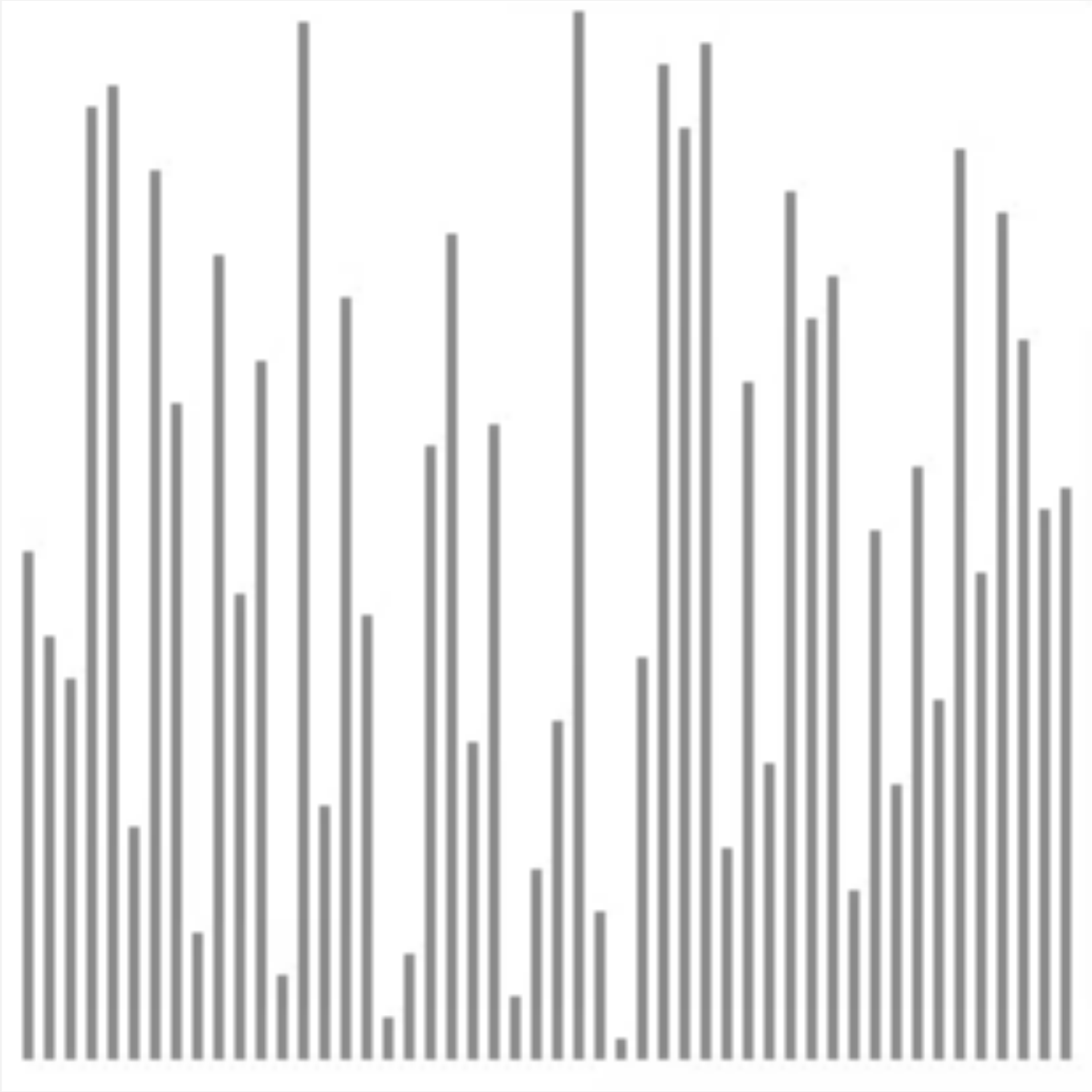
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

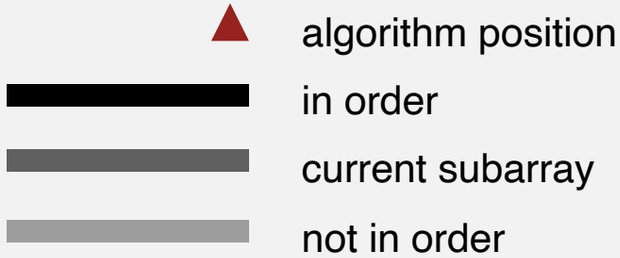
lo hi

result after recursive call

Mergesort: animation

50 random items

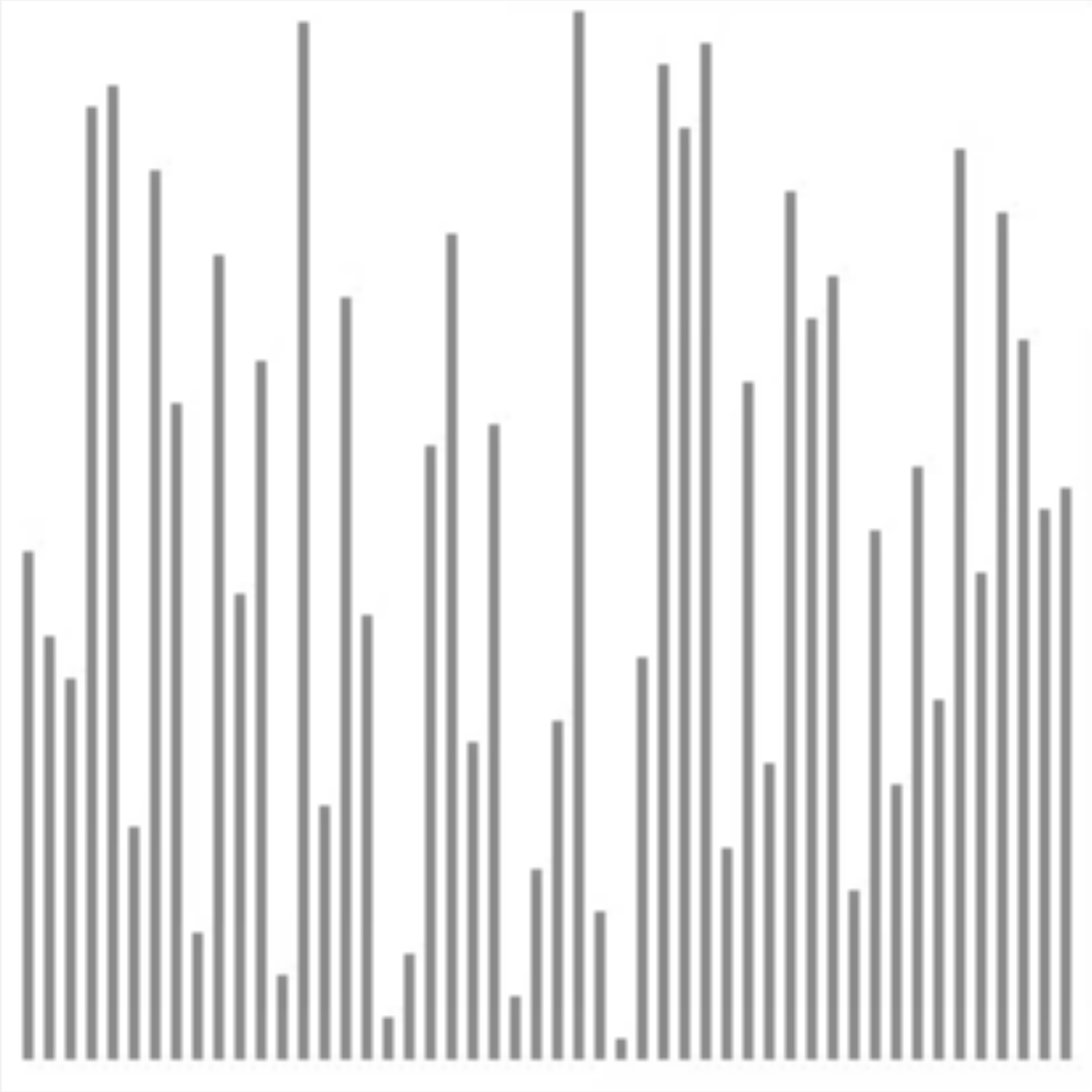


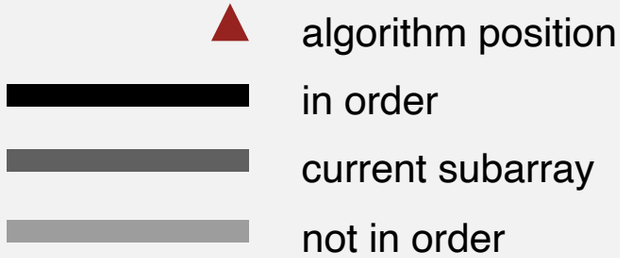


http://www.sorting-algorithms.com/merge-sort

Mergesort: animation

50 random items

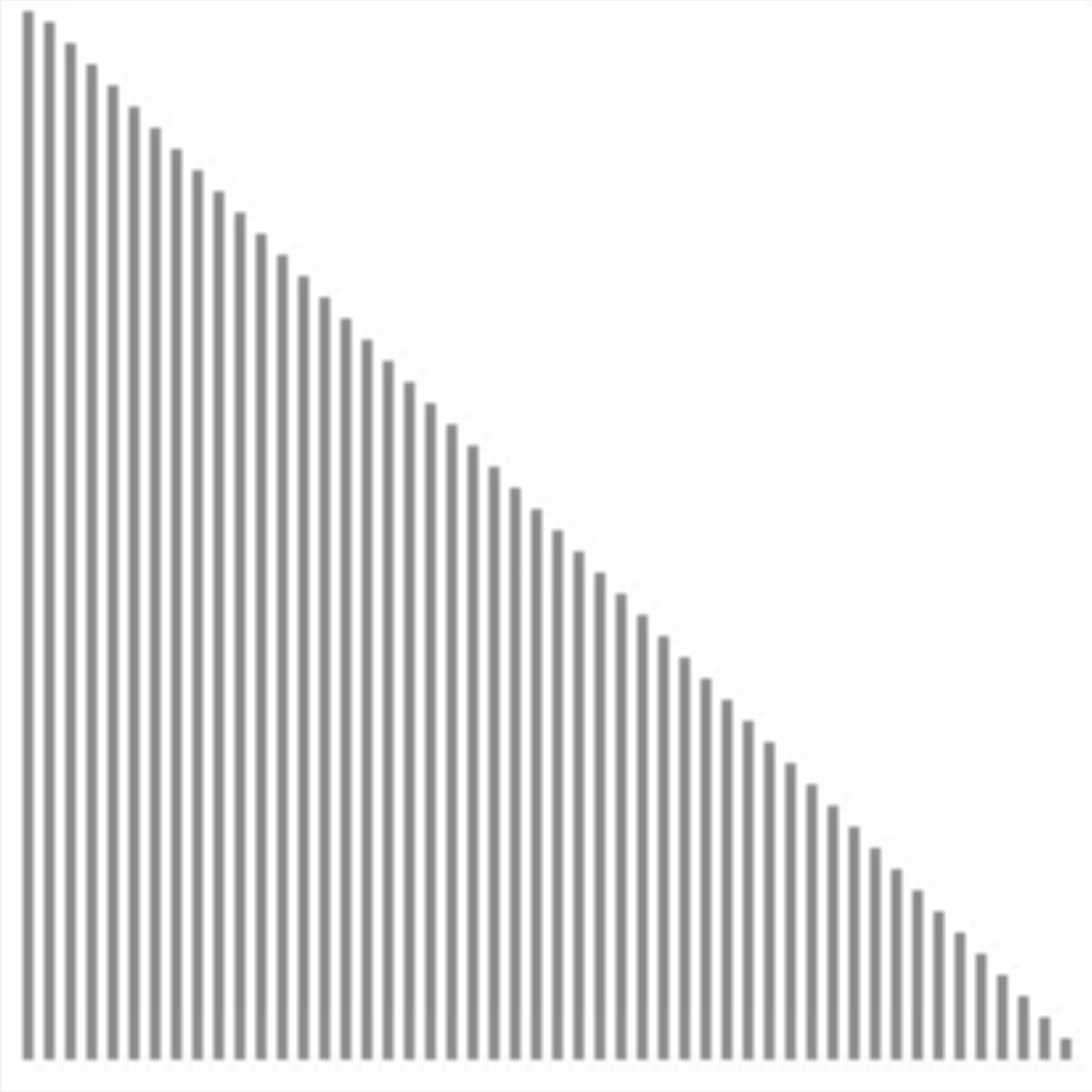


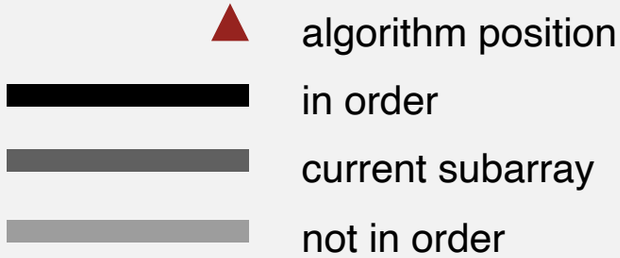


http://www.sorting-algorithms.com/merge-sort

Mergesort: animation

50 reverse-sorted items

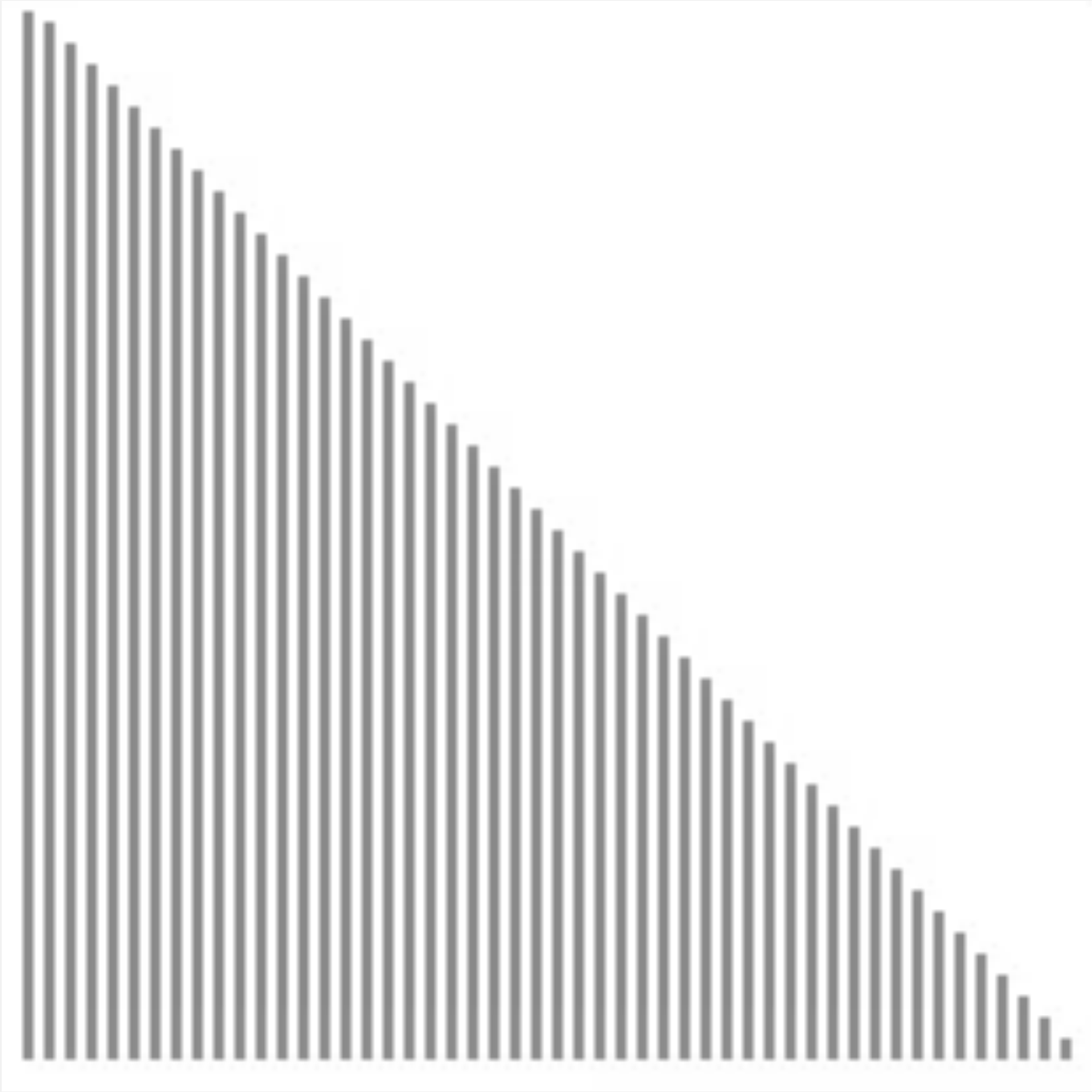


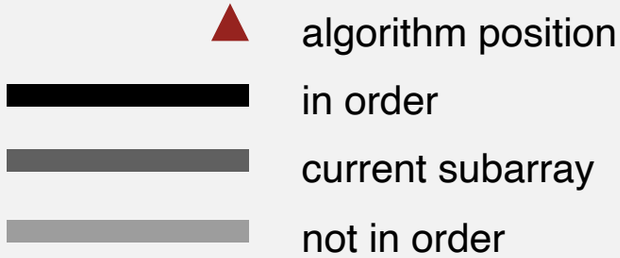


http://www.sorting-algorithms.com/merge-sort

Mergesort: animation

50 reverse-sorted items





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

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N}_{\text{merge}} \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N}_{\text{merge}} \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

We solve the recurrence when N is a power of 2:

result holds for all N
(analysis cleaner in this case)

$$D(N) = 2\,D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

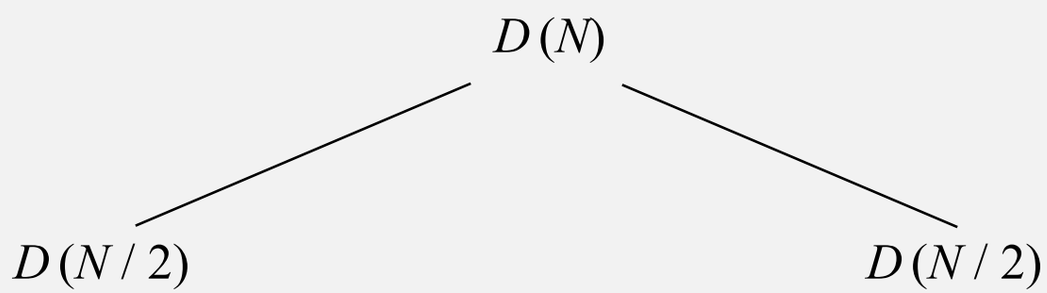
Pf 1. [assuming N is a power of 2]

$D(N)$

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

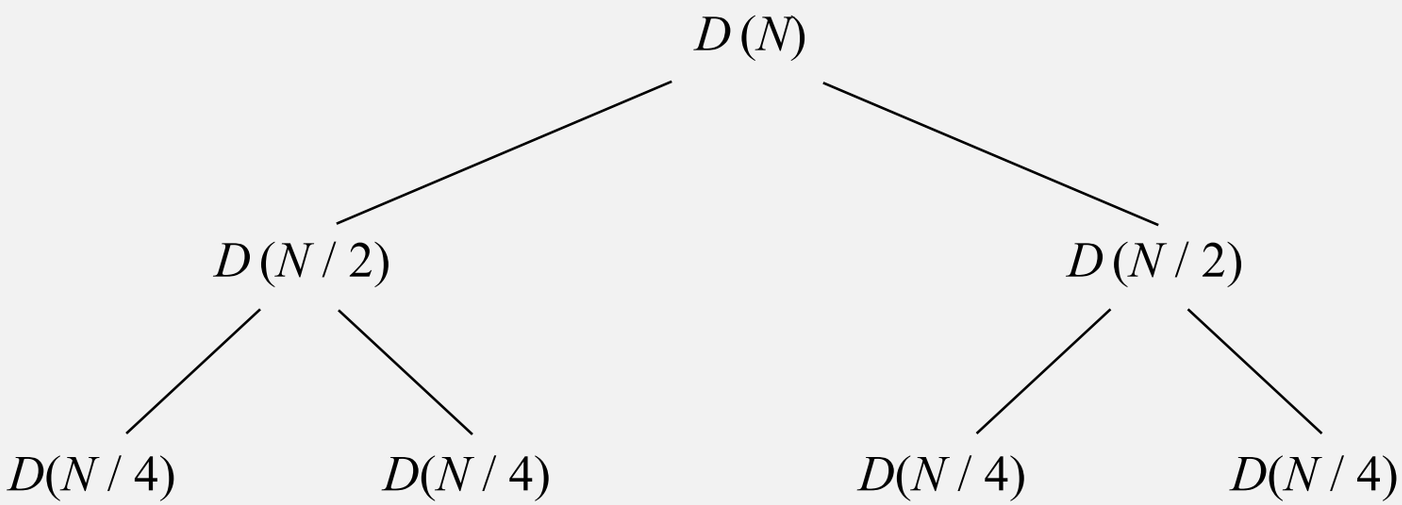
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

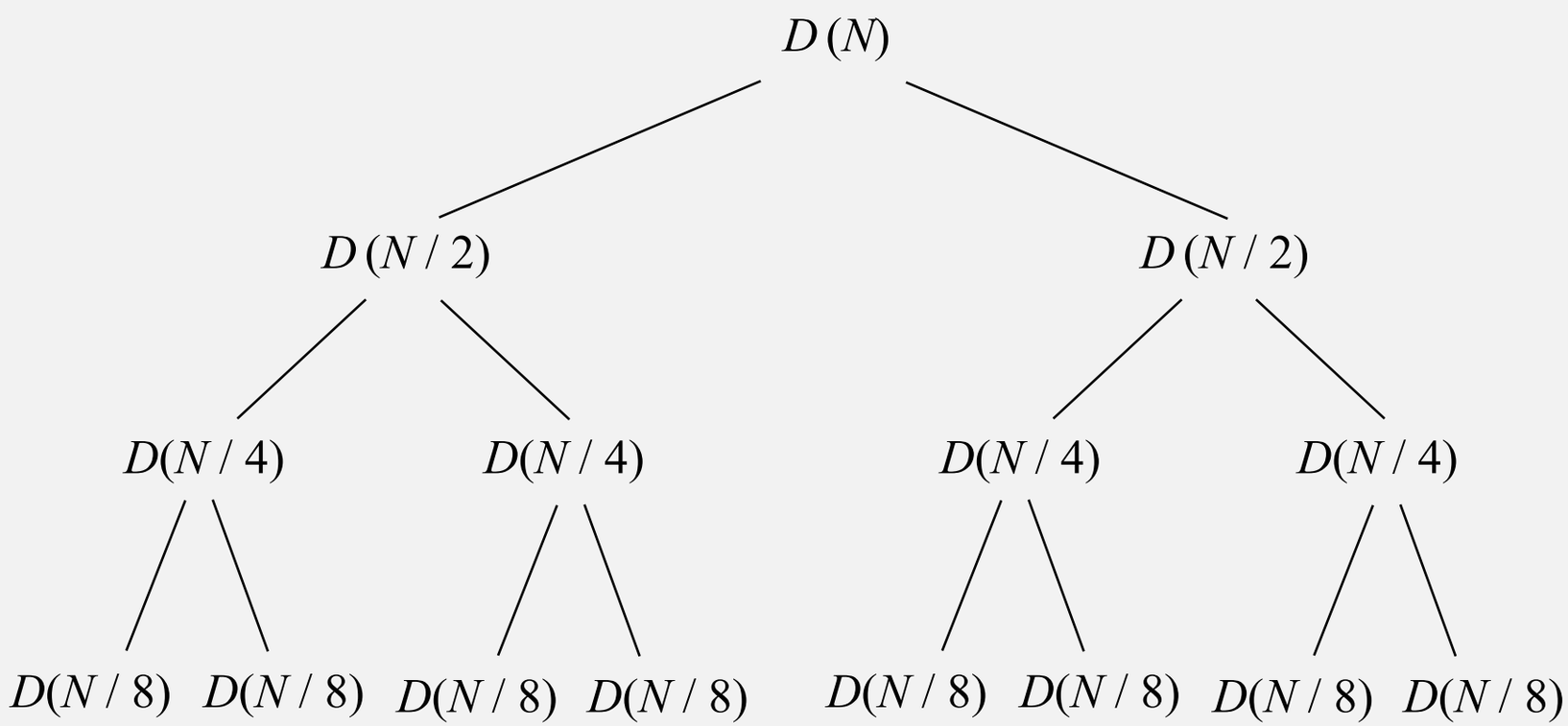
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

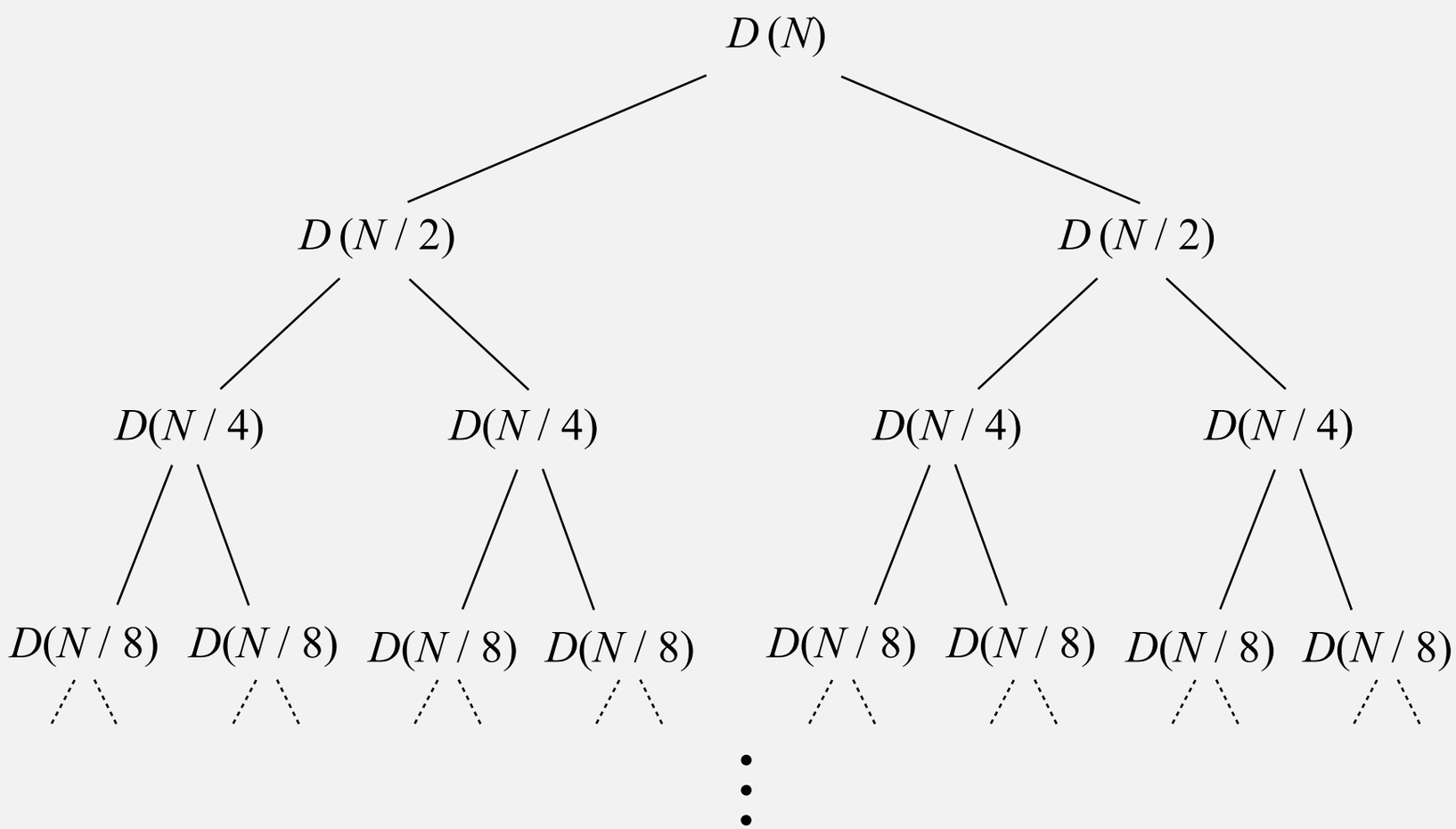
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

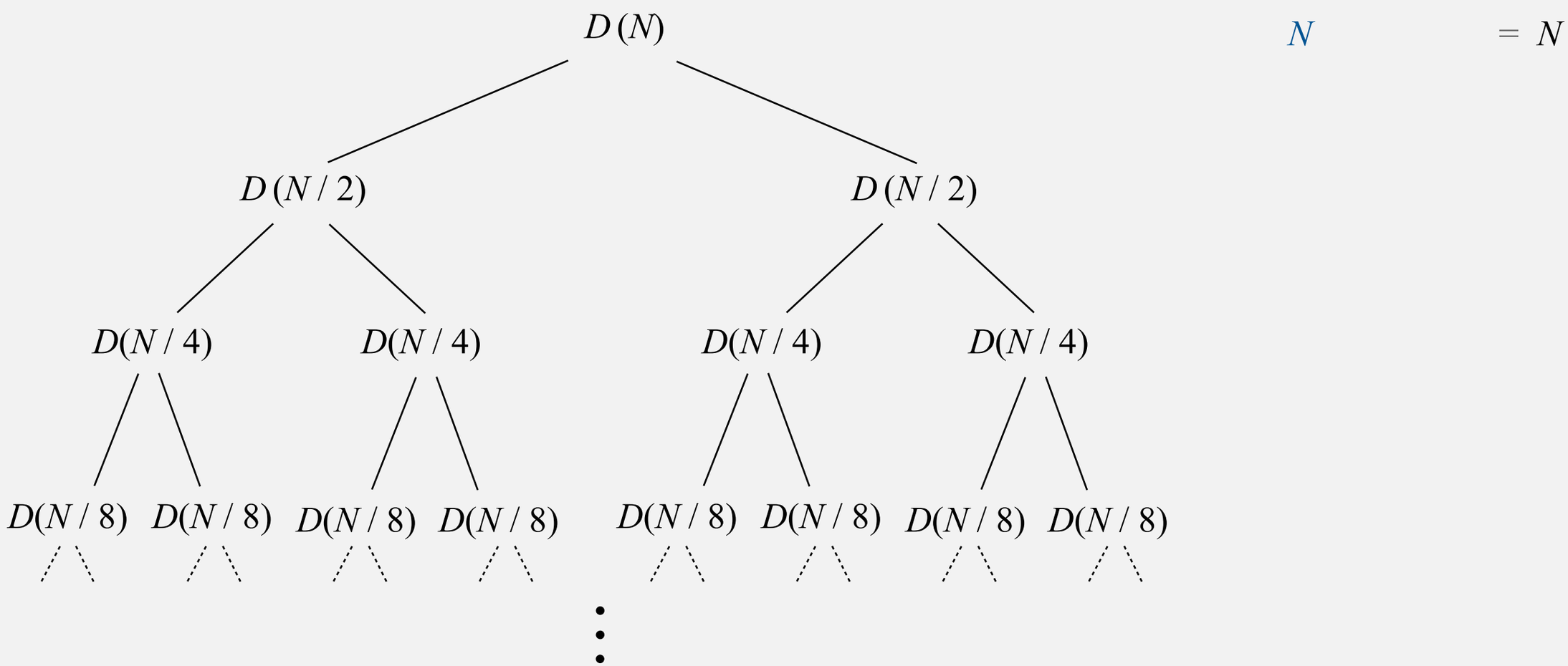
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

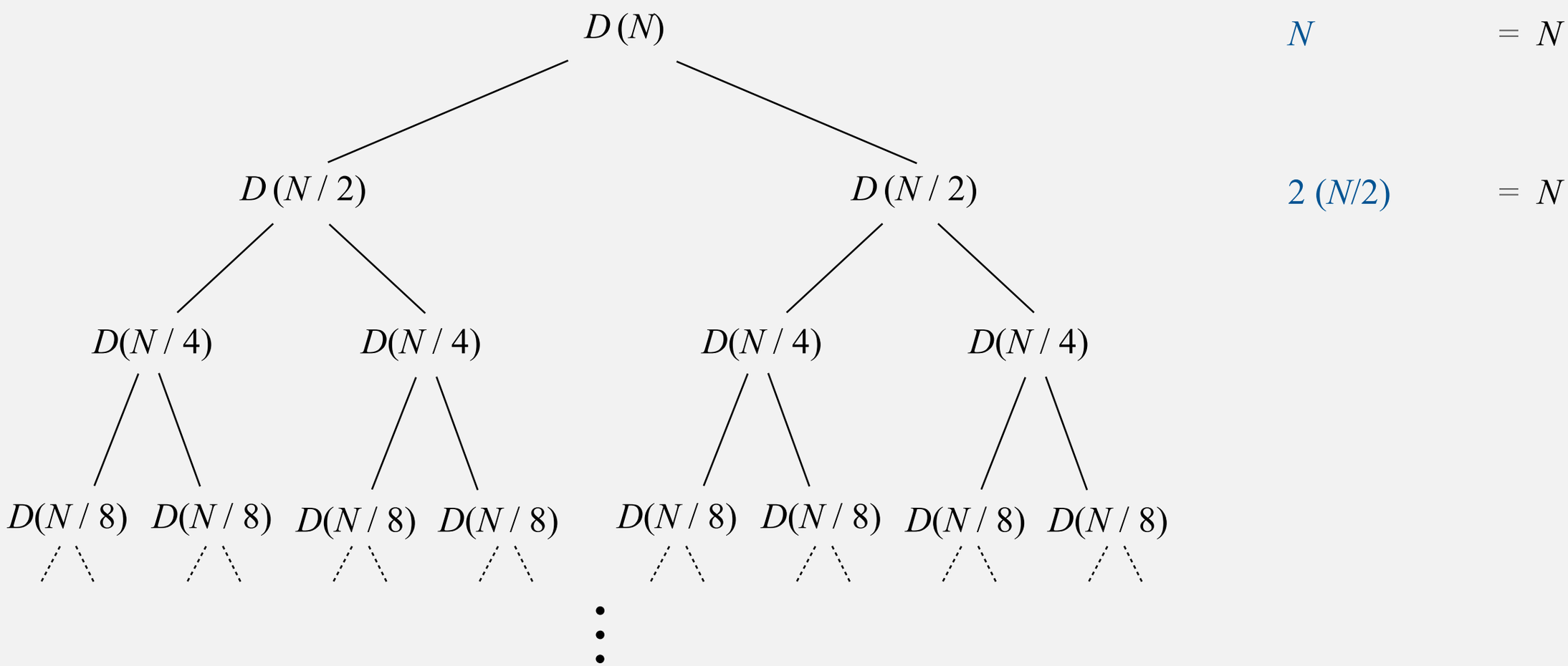
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

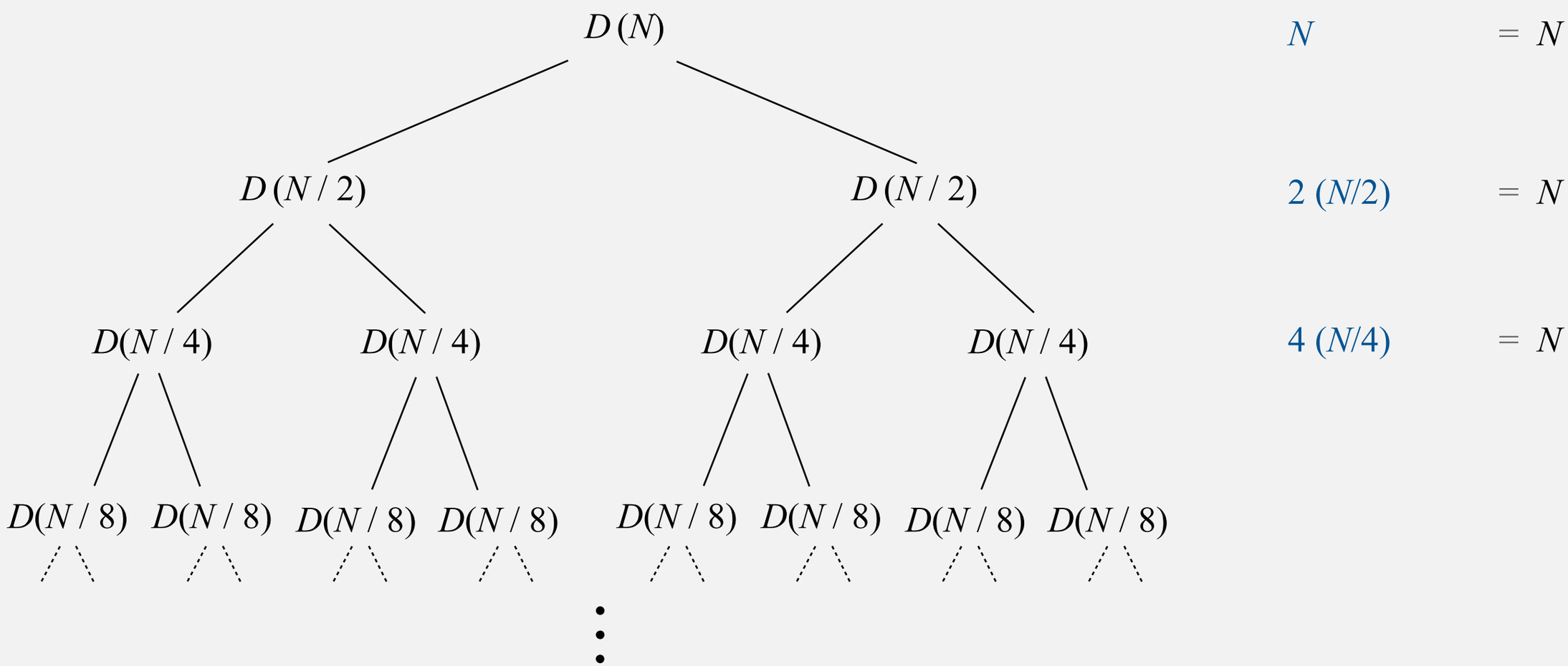
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

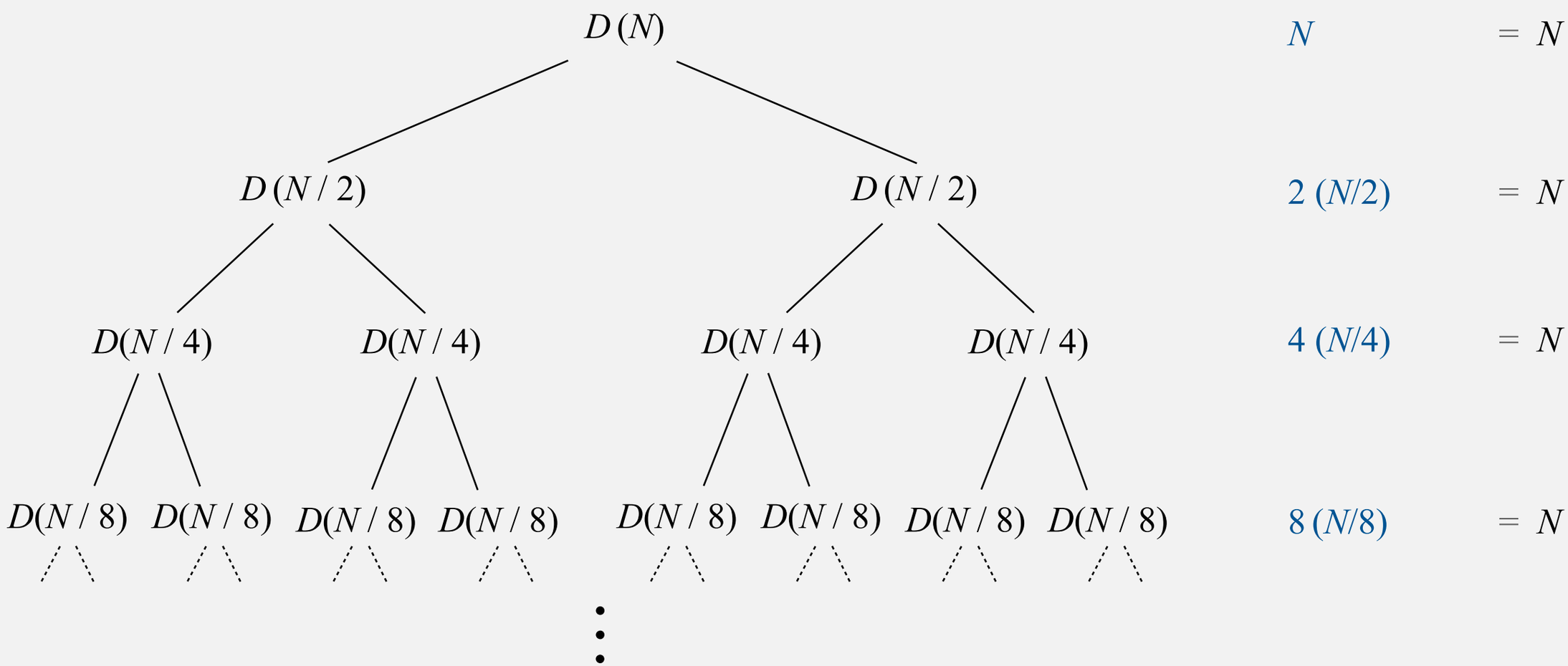
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

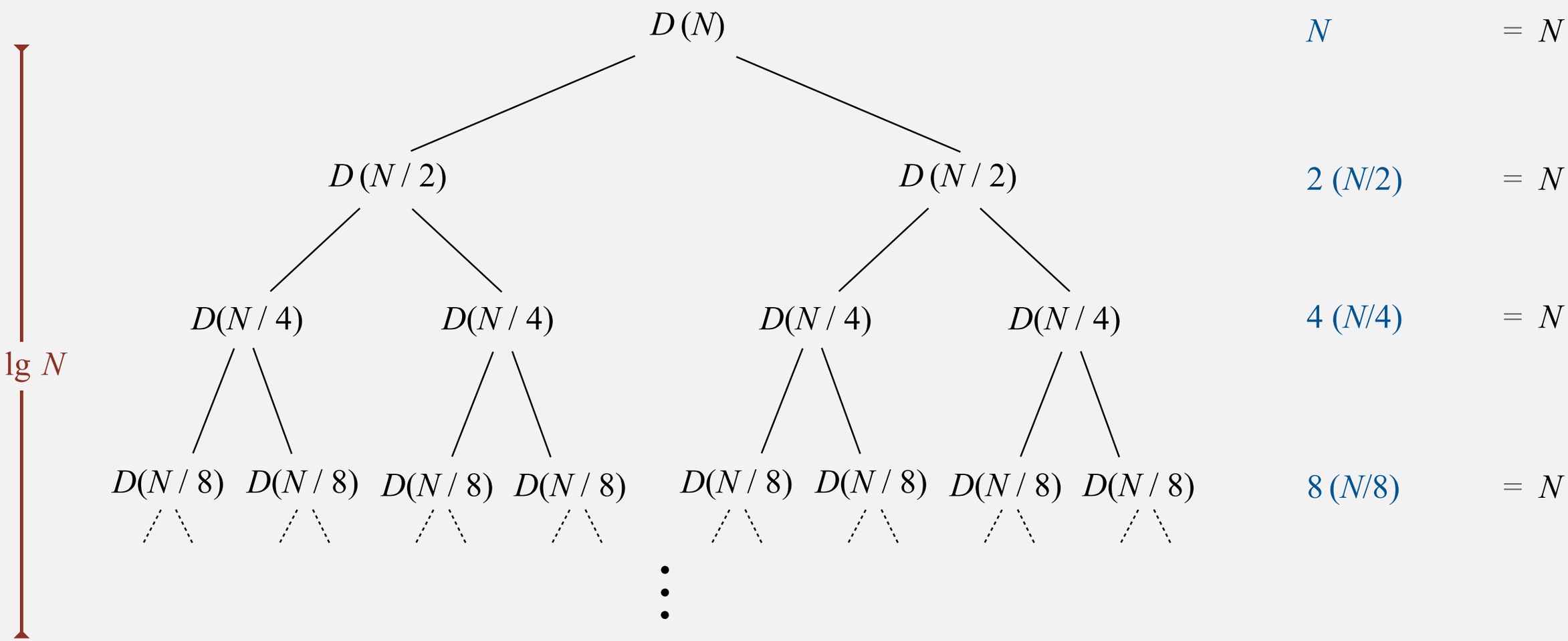
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

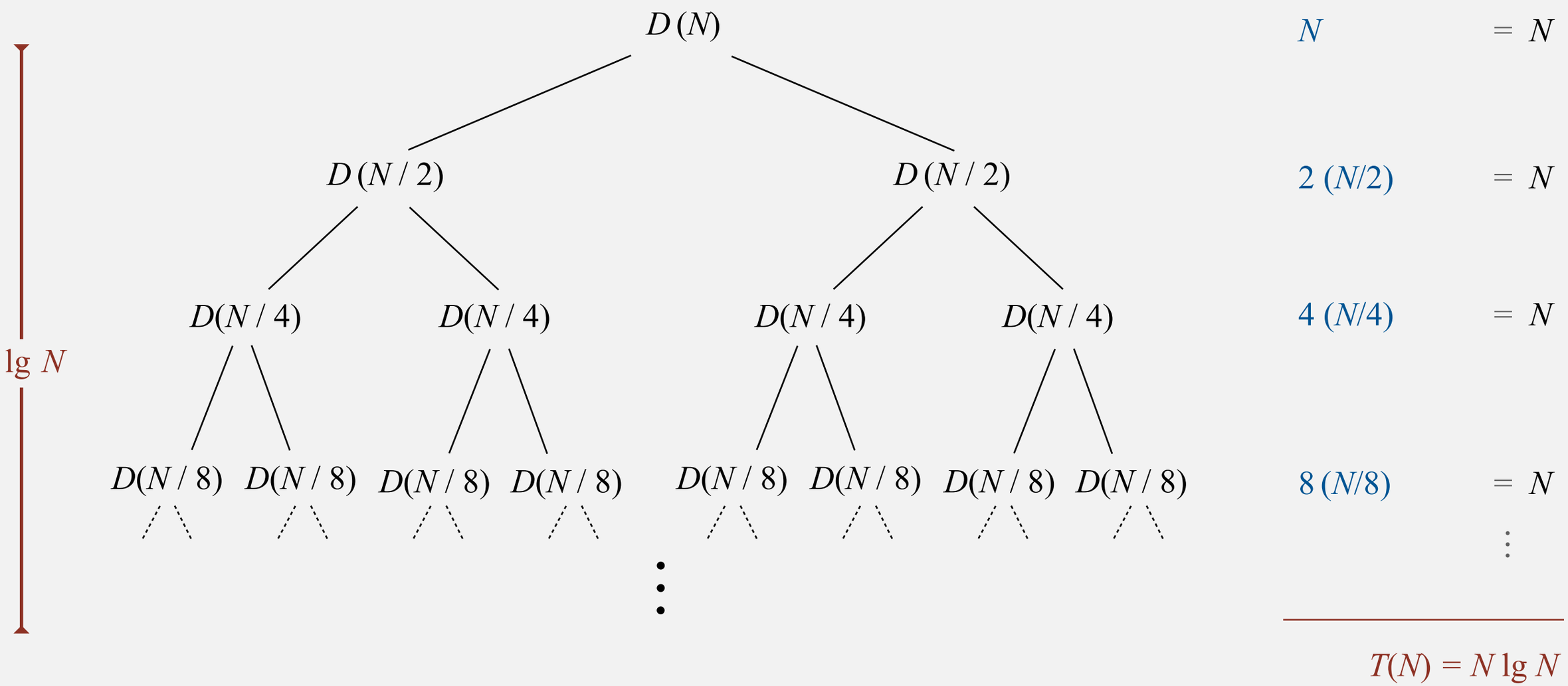
Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



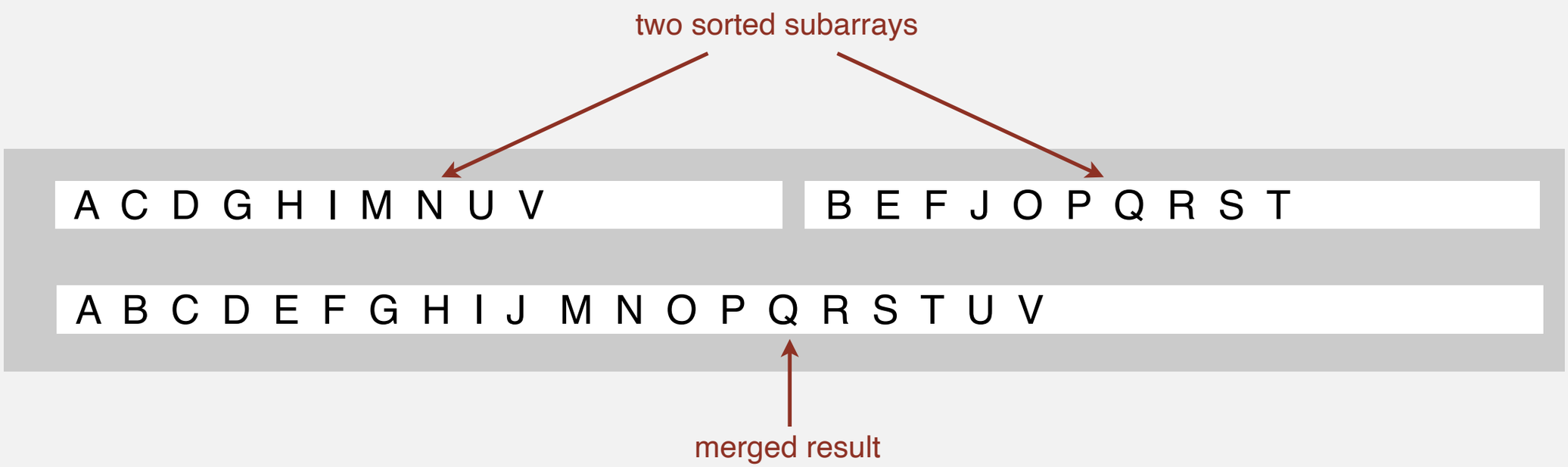
Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

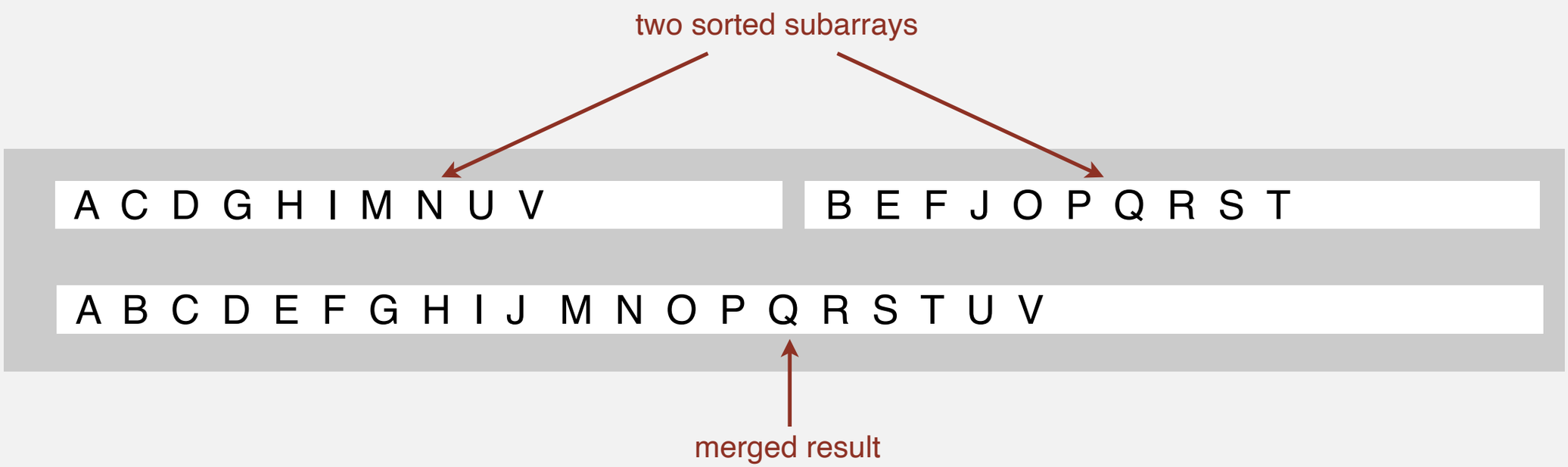
Pf. The array aux[] needs to be of length N for the last merge.



Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of length N for the last merge.

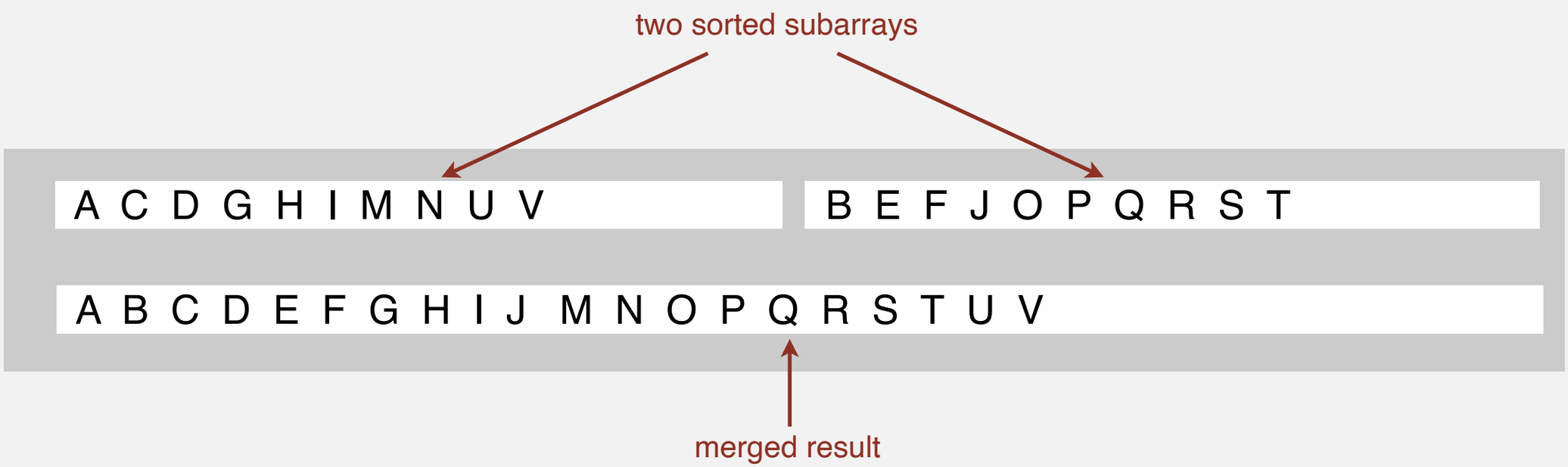


Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of length N for the last merge.



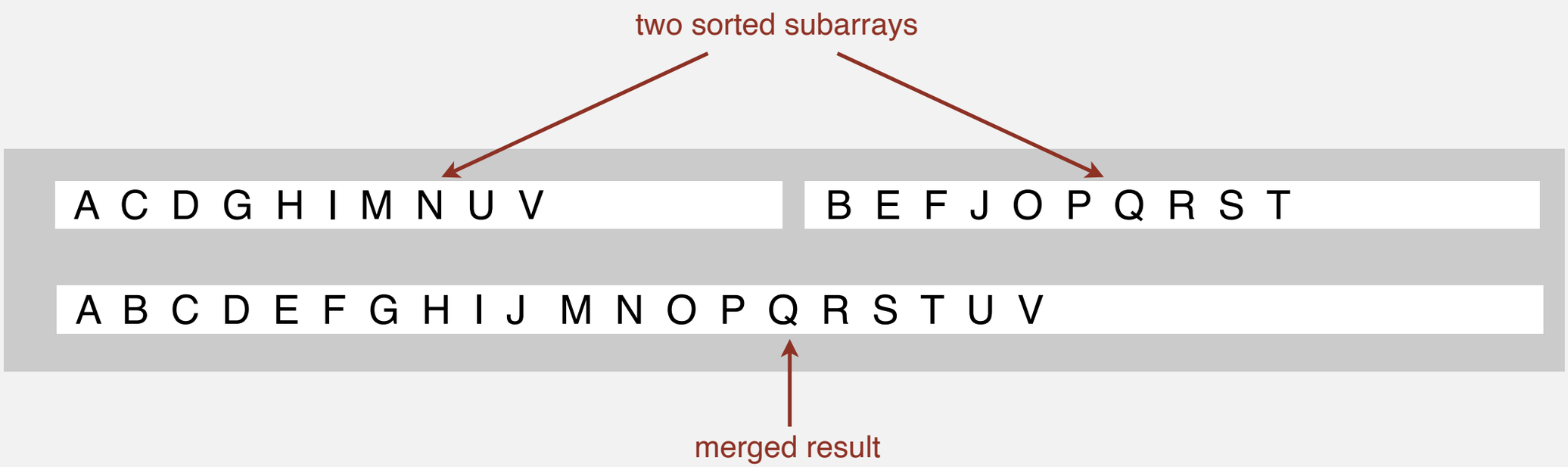
Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of length N for the last merge.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

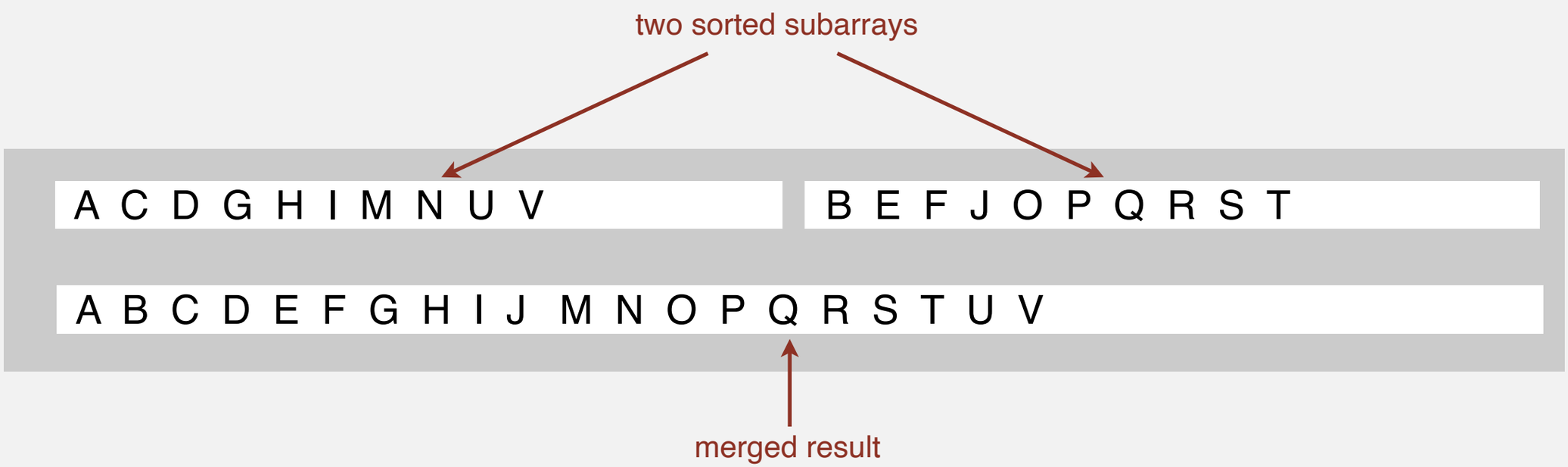
Ex. Insertion sort, selection sort, shellsort.

Challenge 1 (not hard). Use aux[] array of length $\sim \frac{1}{2} N$ instead of N.

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of length N for the last merge.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

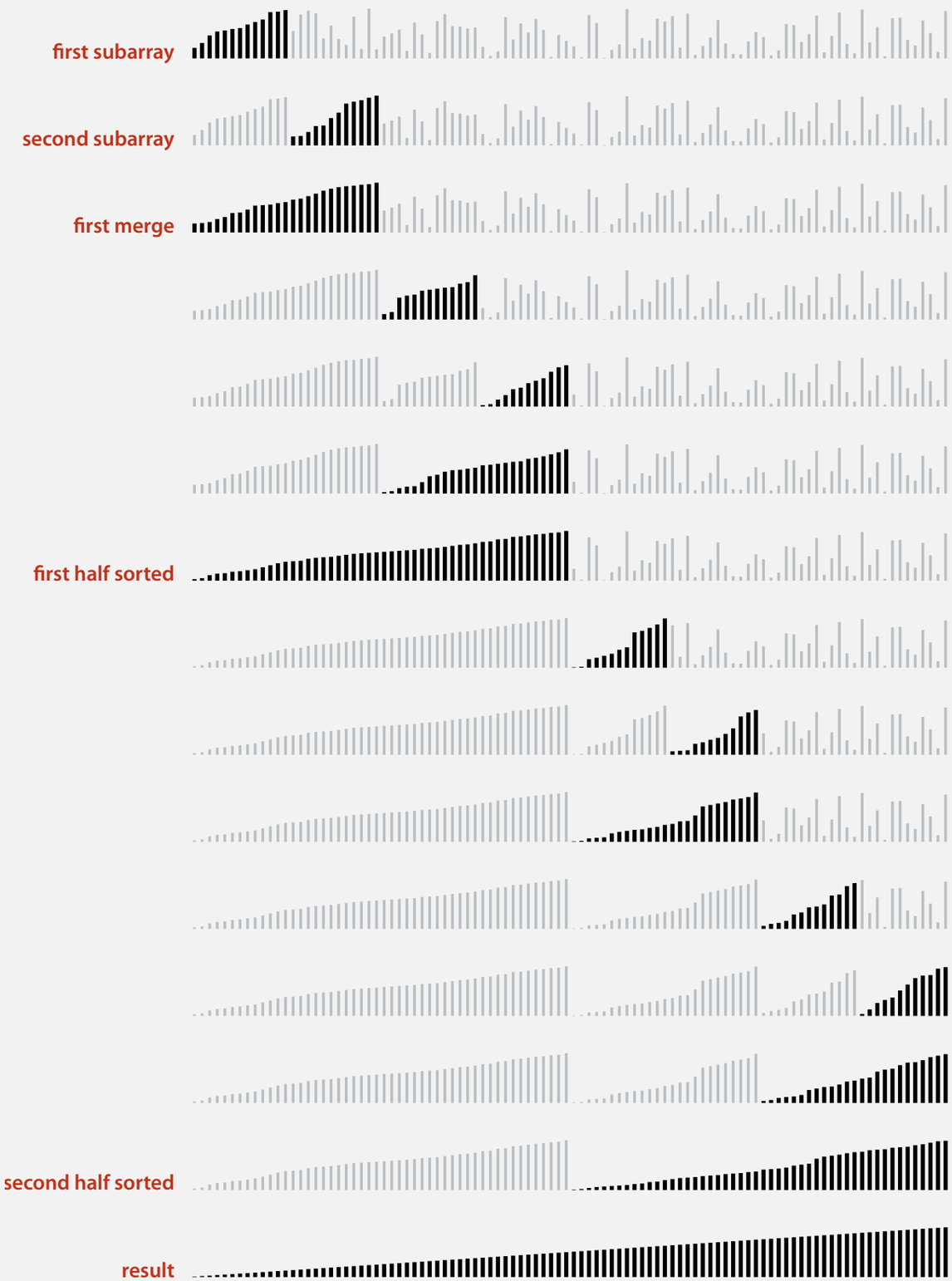
Challenge 1 (not hard). Use aux[] array of length $\sim \frac{1}{2} N$ instead of N.

Challenge 2 (very hard). In-place merge. [Kronrod 1969]

Mergesort with cutoff to insertion sort: visualization

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.



Mergesort: practical improvements

Stop if already sorted.

- Is largest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

A B C D E F G H I J

M N O P Q R S T U V

A B C D E F G H I J M N O P Q R S T U V

http://algs4.cs.princeton.edu

2.2 Mergesort

- mergesort
- bottom-up mergesort
- stability

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.

a[i]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.

a[i]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

a[i]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

a[i]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
`merge(a, aux, 8, 9, 11)`	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

a[i]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
`merge(a, aux, 8, 9, 11)`	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
`merge(a, aux, 0, 3, 7)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, aux, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

a[i]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
`merge(a, aux, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 4, 4, 5)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 6, 6, 7)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, aux, 8, 8, 9)`	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
`merge(a, aux, 10, 10, 11)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 12, 12, 13)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
`merge(a, aux, 14, 14, 15)`	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
`merge(a, aux, 0, 1, 3)`	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
`merge(a, aux, 4, 5, 7)`	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
`merge(a, aux, 8, 9, 11)`	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
`merge(a, aux, 12, 13, 15)`	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
`merge(a, aux, 0, 3, 7)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, aux, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
`merge(a, aux, 0, 7, 15)`	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom-up mergesort: Java implementation

```
public class MergeBU {
  private static void merge(...)
  {  /* as before */  }

  public static void sort(Comparable[] a) {
    int N = a.length;
    Comparable[] aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz)
      for (int lo = 0; lo < N-sz; lo += sz+sz)
        merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
  }
}
```

Bottom line. Simple and non-recursive version of mergesort.

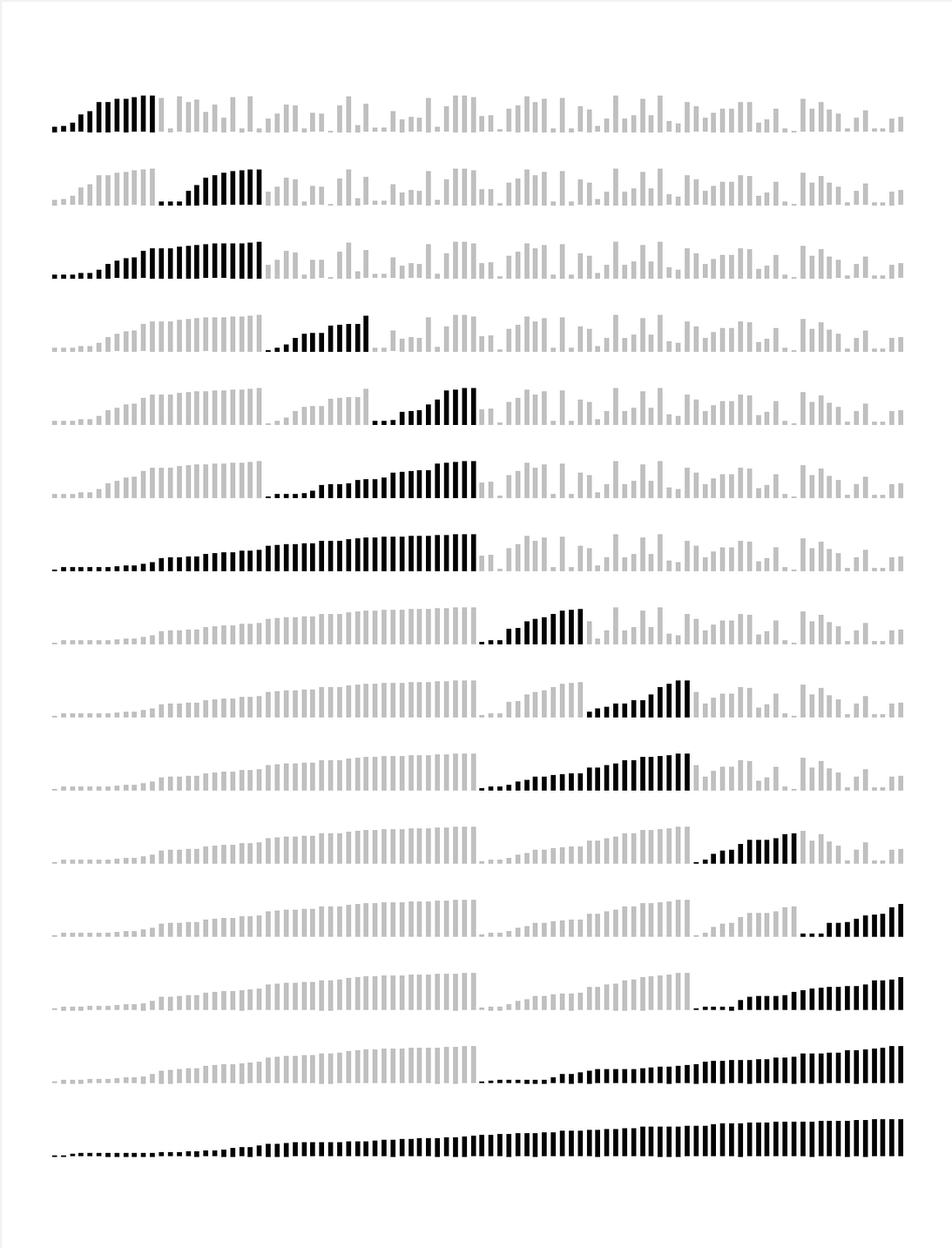
Bottom-up mergesort: Java implementation

```
public class MergeBU {
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a) {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

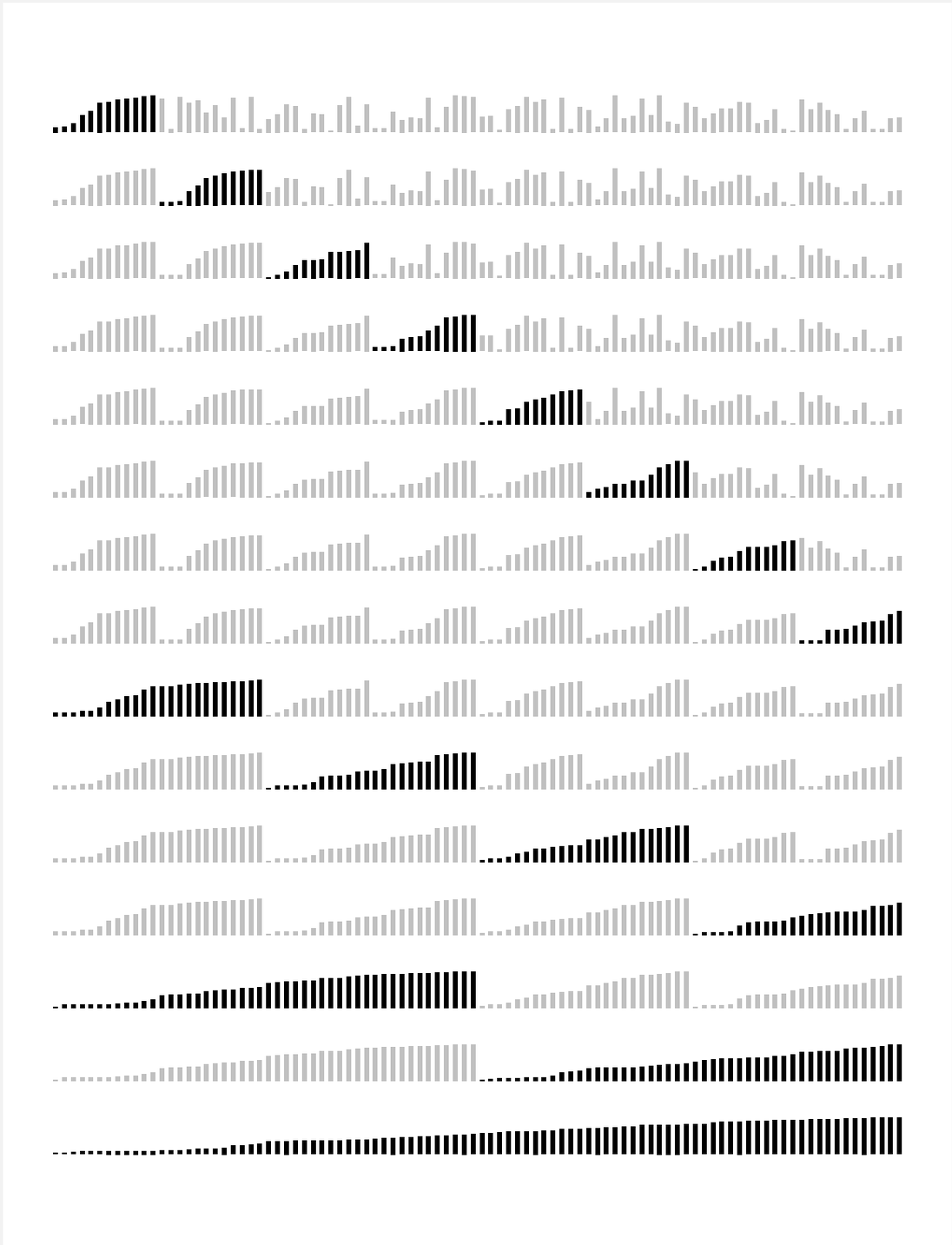
but about 10% slower than recursive,
top-down mergesort on typical systems

Bottom line. Simple and non-recursive version of mergesort.

Mergesort: visualizations



top-down mergesort (cutoff = 12)



bottom-up mergesort (cutoff = 12)

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

merge two runs

1	3	4	5	10	16	23	9	13	2	7	8	12	14

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14

merge two runs

1	3	4	5	10	16	23	9	13	2	7	8	12	14

Tradeoff. Fewer passes vs. extra compares per pass to identify runs.

Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Tim Peters

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

...

Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Tim Peters

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

...

Consequence. Linear time on many arrays with pre-existing order.

Timsort

Tim Peters

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

...

Consequence. Linear time on many arrays with pre-existing order.

Now widely used. Python, Java 7, GNU Octave, Android, ….

http://algs4.cs.princeton.edu

2.2 Mergesort

- mergesort
- bottom-up mergesort
- stability

Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, new Student.ByName());

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
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

Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, new Student.ByName());

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
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

Selection.sort(a, new Student.BySection());

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, new Student.ByName());

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
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

Selection.sort(a, new Student.BySection());

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Students in section 3 no longer sorted by name.

Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, new Student.ByName());

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
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

Selection.sort(a, new Student.BySection());

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

Stability

Q. Which sorts are stable?

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Stability

Q. Which sorts are stable?

A. Need to check algorithm (and implementation).

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Stability: insertion sort

Proposition. Insertion sort is stable.

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Stability: insertion sort

Proposition. Insertion sort is stable.

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Pf. Equal items never move past each other.

Stability: selection sort

Proposition. Selection sort is not stable.

Stability: selection sort

Proposition. Selection sort is not stable.

i	min	0	1	2
0	2	B_1	B_2	A
1	1	A	B_2	B_1
2	2	A	B_2	B_1
		A	B_2	B_1

Pf by counterexample. Long-distance exchange can move one equal item past another one.

Stability: shellsort

Proposition. Shellsort sort is not stable.

Stability: shellsort

Proposition. Shellsort sort is not stable.

h	0	1	2	3	4
	B_1	B_2	B_3	B_4	A_1
4	A_1	B_2	B_3	B_4	B_1
1	A_1	B_2	B_3	B_4	B_1
	A_1	B_2	B_3	B_4	B_1

Pf by counterexample. Long-distance exchanges.

Stability: mergesort

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.

0	1	2	3	4
A_1	A_2	A_3	B	D

5	6	7	8	9	10
A_4	A_5	C	E	F	G

Stability: mergesort

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.

0	1	2	3	4
A_1	A_2	A_3	B	D

5	6	7	8	9	10
A_4	A_5	C	E	F	G

Stability: mergesort

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.

Proposition. Merge operation is stable.

0	1	2	3	4
A_1	A_2	A_3	B	D

5	6	7	8	9	10
A_4	A_5	C	E	F	G

Pf. Takes from left subarray if equal keys.

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log_3 N$	?	$c N^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✔	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail

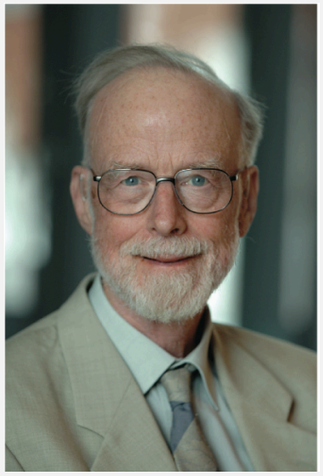
http://algs4.cs.princeton.edu

2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts

Tony Hoare

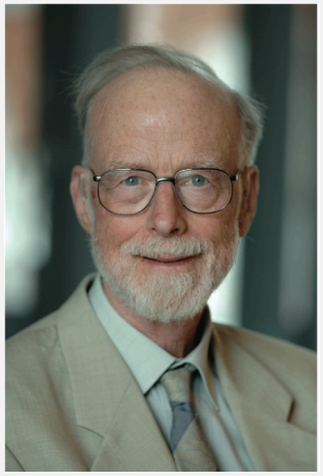
- Invented quicksort to translate Russian into English.



Tony Hoare
1980 Turing Award

Tony Hoare

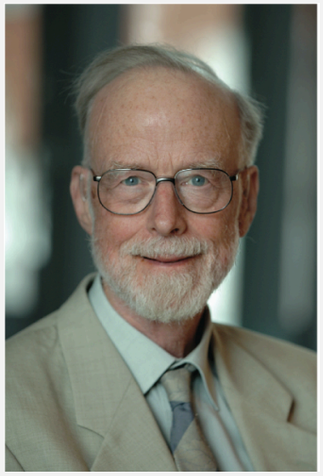
- Invented quicksort to translate Russian into English.
- [but couldn't explain his algorithm or implement it!]



Tony Hoare
1980 Turing Award

Tony Hoare

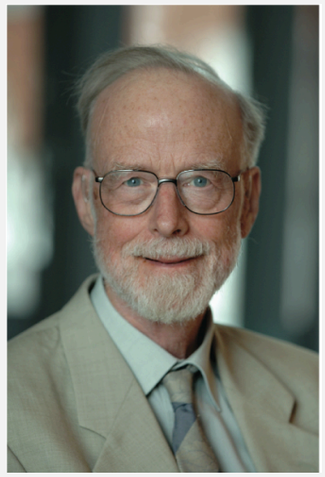
- Invented quicksort to translate Russian into English.
- [but couldn't explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).



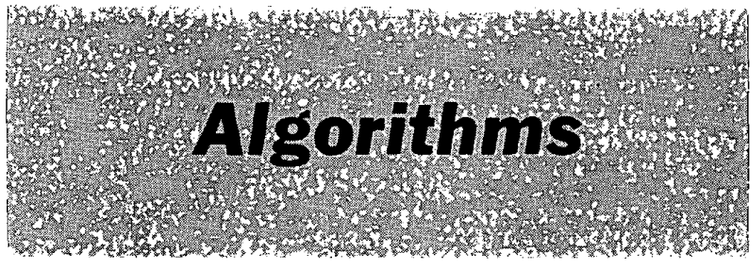
Tony Hoare
1980 Turing Award

Tony Hoare

- Invented quicksort to translate Russian into English.
- [but couldn't explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.



Tony Hoare
1980 Turing Award



ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  quicksort (A,M,N);  value M,N;
             array A;  integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin      integer I,J;
           if M < N then begin partition (A,M,N,I,J);
                                 quicksort (A,M,J);
                                 quicksort (A, I, N)
                           end
end        quicksort
```

Communications of the ACM (July 1961)

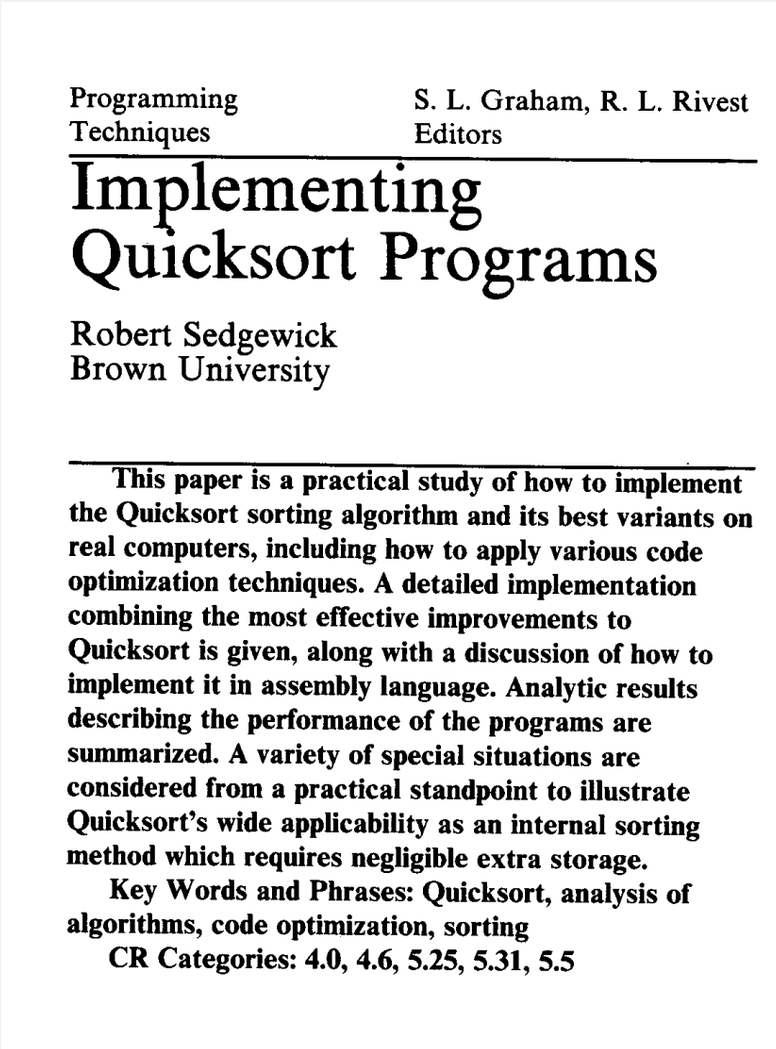
Bob Sedgewick

Bob Sedgewick

Bob Sedgewick

- Refined and popularized quicksort.

Bob Sedgewick



Programming Techniques

S. L. Graham, R. L. Rivest
Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Bob Sedgewick

- Refined and popularized quicksort.
- Analyzed quicksort.

Bob Sedgewick

Programming Techniques

S. L. Graham, R. L. Rivest
Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 19, 1976

Summary. The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

Quicksort

Basic plan.

input

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E

Quicksort

Basic plan.

- Shuffle the array.

shuffle

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E

Quicksort

Basic plan.

- Shuffle the array.

shuffle

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j

partition

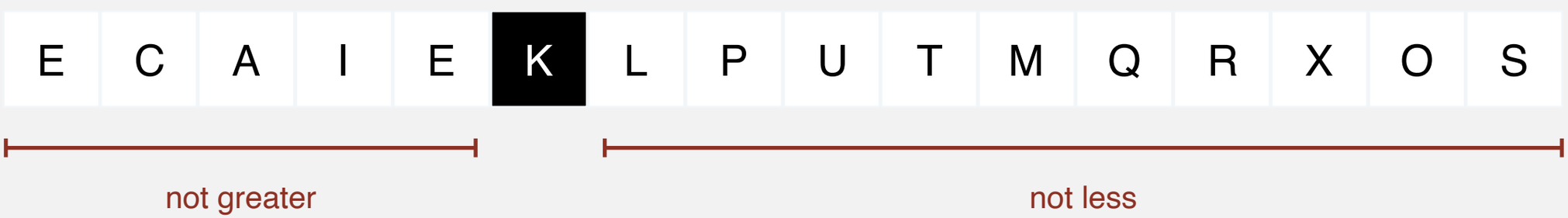
K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j

partition



Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each subarray recursively.

sort the left subarray

E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each subarray recursively.

sort the left subarray

A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each subarray recursively.

sort the right subarray

A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each subarray recursively.

sort the right subarray

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each subarray recursively.

sorted array

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Partition Key

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Partition Key

a[i] = R

a[lo] = K

a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = S

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = S

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = S

Needs to go on right side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

a[i] = R

a[lo] = K

a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Partition Key

a[i] = R

a[lo] = K

a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = O

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = O

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = O

Needs to go on right side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

a[i] = R a[lo] = K a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Partition Key

a[i] = R

a[lo] = K

a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = X

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = X

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K < a[j] = X

Needs to go on right side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

a[i] = R a[lo] = K a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Partition Key

a[i] = R a[lo] = K a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R a[lo] = K a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K > a[j] = C

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K > a[j] = C

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = R > a[lo] = K > a[j] = C

Needs to go on right side

Needs to go on left side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

a[i] = A a[lo] = K a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

Partition Key

a[i] = A a[lo] = K a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A a[lo] = K a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A a[lo] = K a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A < a[lo] = K a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A < a[lo] = K < a[j] = Q

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A < a[lo] = K < a[j] = Q

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = A < a[lo] = K < a[j] = Q

On correct side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

a[i] = T

a[lo] = K

a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

Partition Key

a[i] = T

a[lo] = K

a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T a[lo] = K a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T a[lo] = K a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K < a[j] = M

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K < a[j] = M

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K < a[j] = M

Needs to go on right side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

a[i] = T a[lo] = K a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

Partition Key

a[i] = T

a[lo] = K

a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T a[lo] = K a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T a[lo] = K a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K > a[j] = I

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K > a[j] = I

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = T > a[lo] = K > a[j] = I

Needs to go on right side

Needs to go on left side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

a[i] = E a[lo] = K a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

Partition Key

a[i] = E

a[lo] = K

a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E a[lo] = K a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E a[lo] = K a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E < a[lo] = K a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E < a[lo] = K < a[j] = U

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E < a[lo] = K < a[j] = U

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = E < a[lo] = K < a[j] = U

On correct side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

a[i] = L a[lo] = K a[j] = P

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

Partition Key

a[i] = L

a[lo] = K

a[j] = P

Quicksort partitioning demo

Select first element as partition key

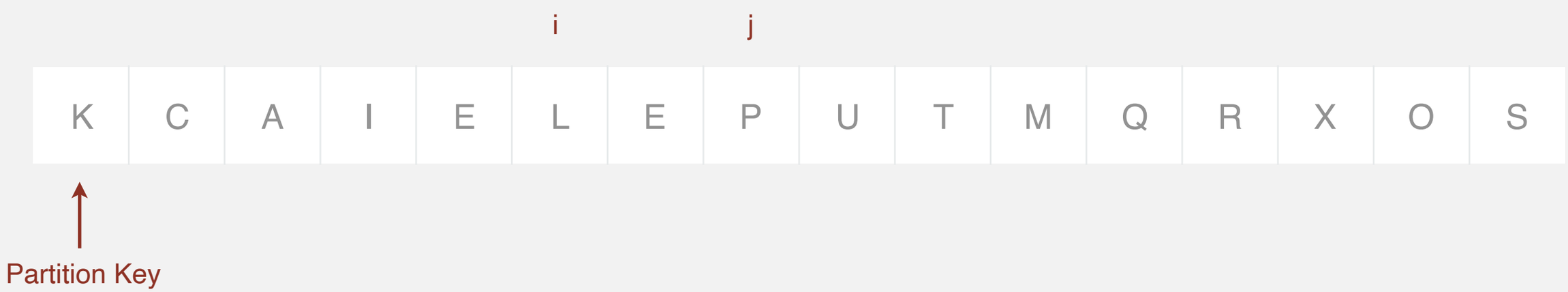
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = L a[lo] = K a[j] = P

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

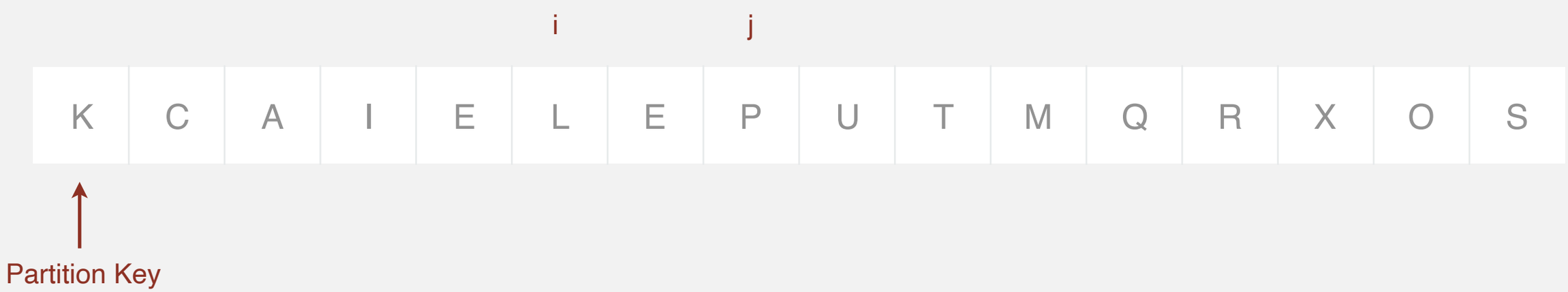


a[i] = L a[lo] = K a[j] = P

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

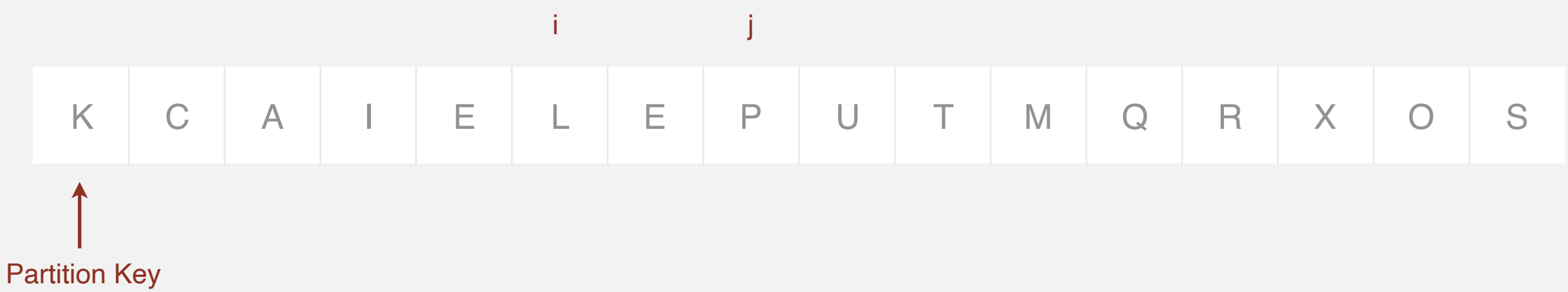


a[i] = L > a[lo] = K a[j] = P

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

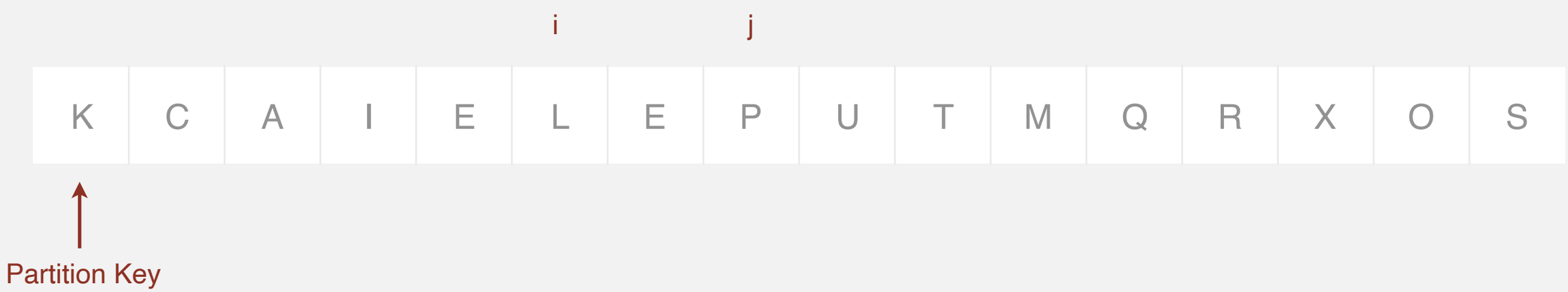


a[i] = L > a[lo] = K < a[j] = P

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



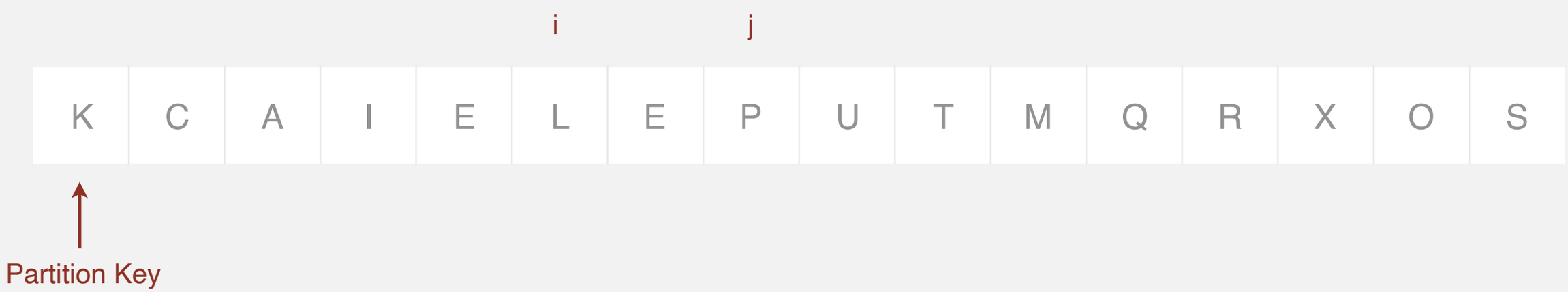
a[i] = L > a[lo] = K < a[j] = P

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



a[i] = L > a[lo] = K < a[j] = P

Needs to go on right side

On correct side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

a[i] = L

a[lo] = K

a[j] = E

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

Partition Key

a[i] = L a[lo] = K a[j] = E

Quicksort partitioning demo

Select first element as partition key

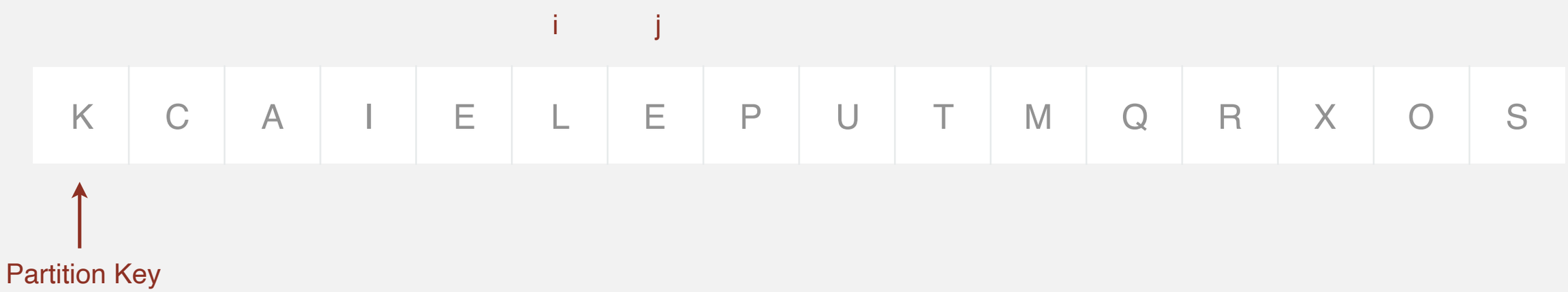
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

a[i] = L a[lo] = K a[j] = E

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

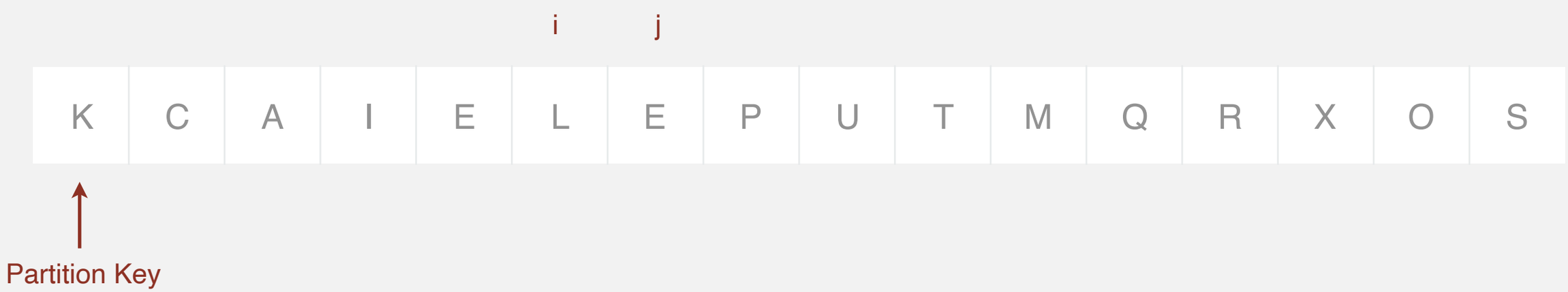


a[i] = L a[lo] = K a[j] = E

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

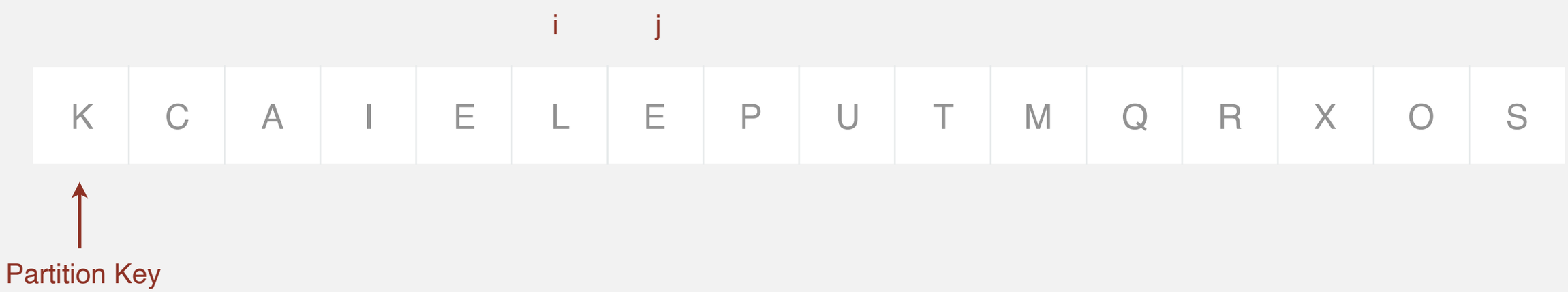


a[i] = L > a[lo] = K a[j] = E

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

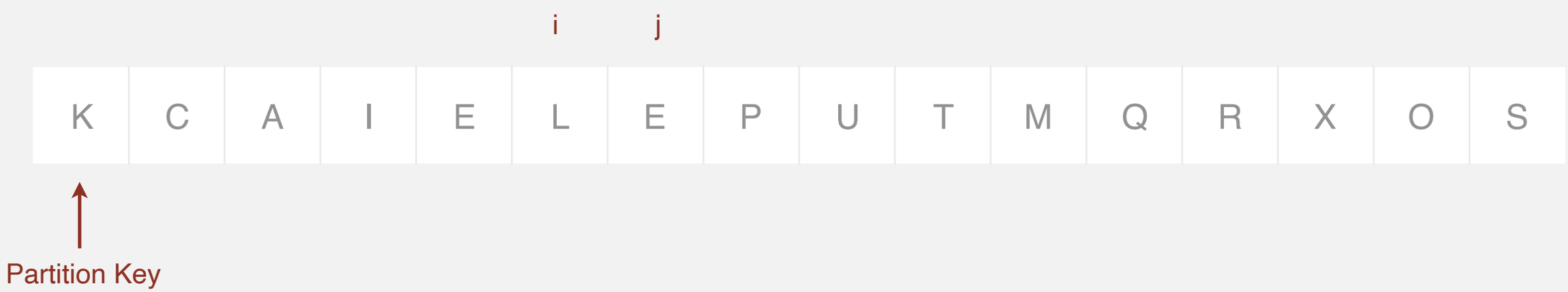


a[i] = L > a[lo] = K > a[j] = E

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



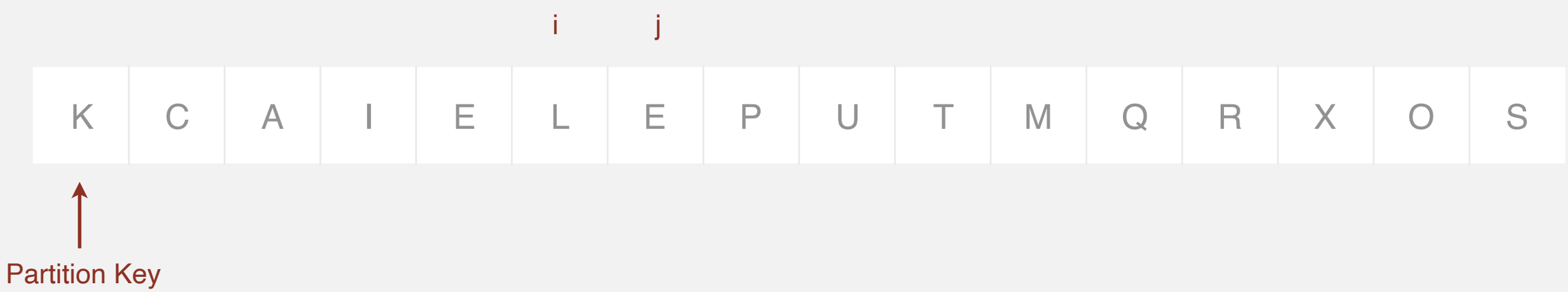
a[i] = L > a[lo] = K > a[j] = E

Needs to go on right side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



a[i] = L > a[lo] = K > a[j] = E

Needs to go on right side

Needs to go on left side

Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S

Quicksort partitioning demo

Select first element as partition key

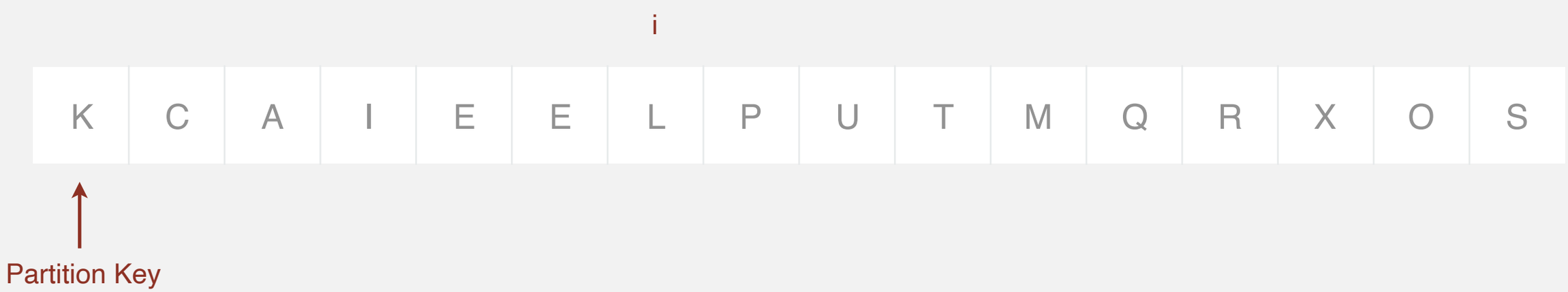
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S

Quicksort partitioning demo

Select first element as partition key

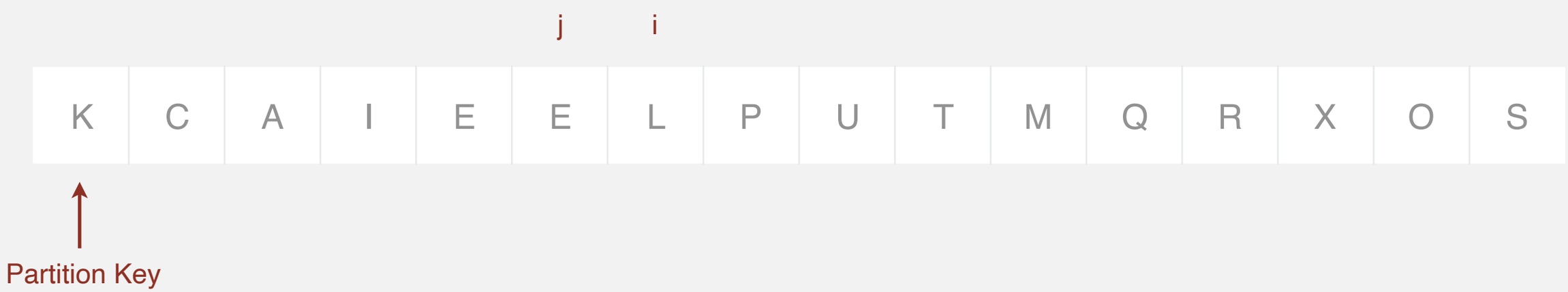
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



Quicksort partitioning demo

Select first element as partition key

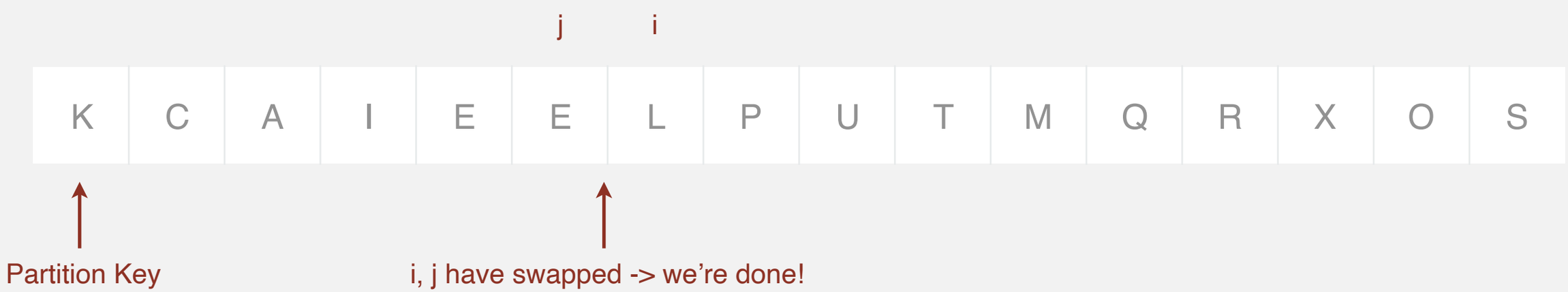
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



Quicksort partitioning demo

Select first element as partition key

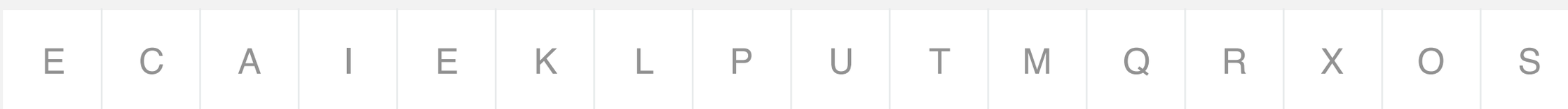
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



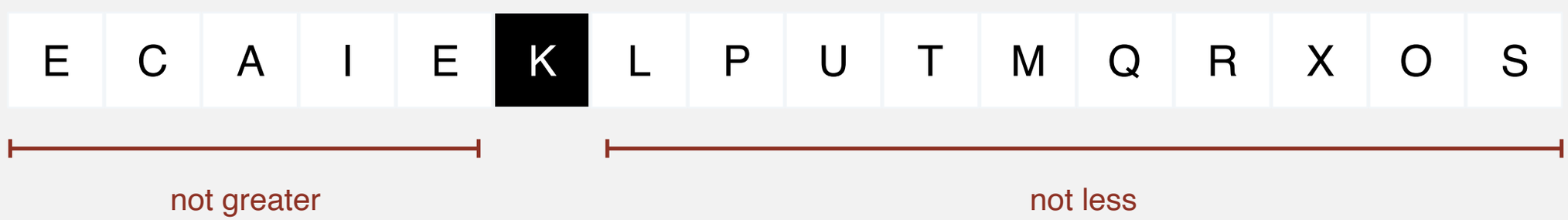
Quicksort partitioning demo

Select first element as partition key

- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



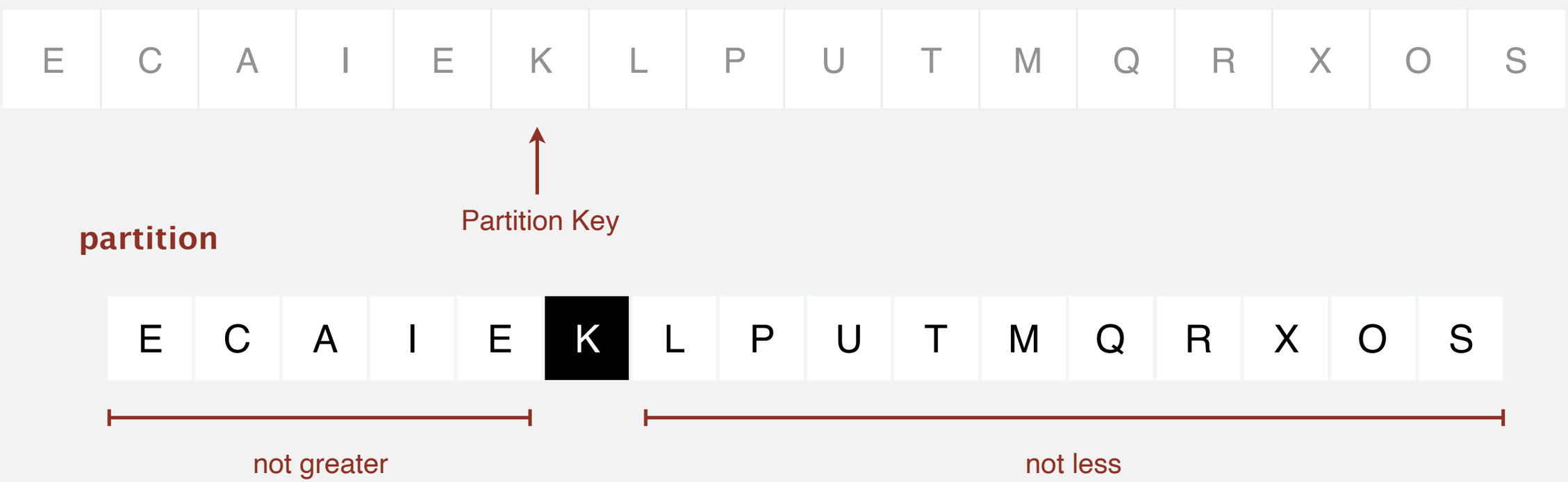
partition



Quicksort partitioning demo

Select first element as partition key

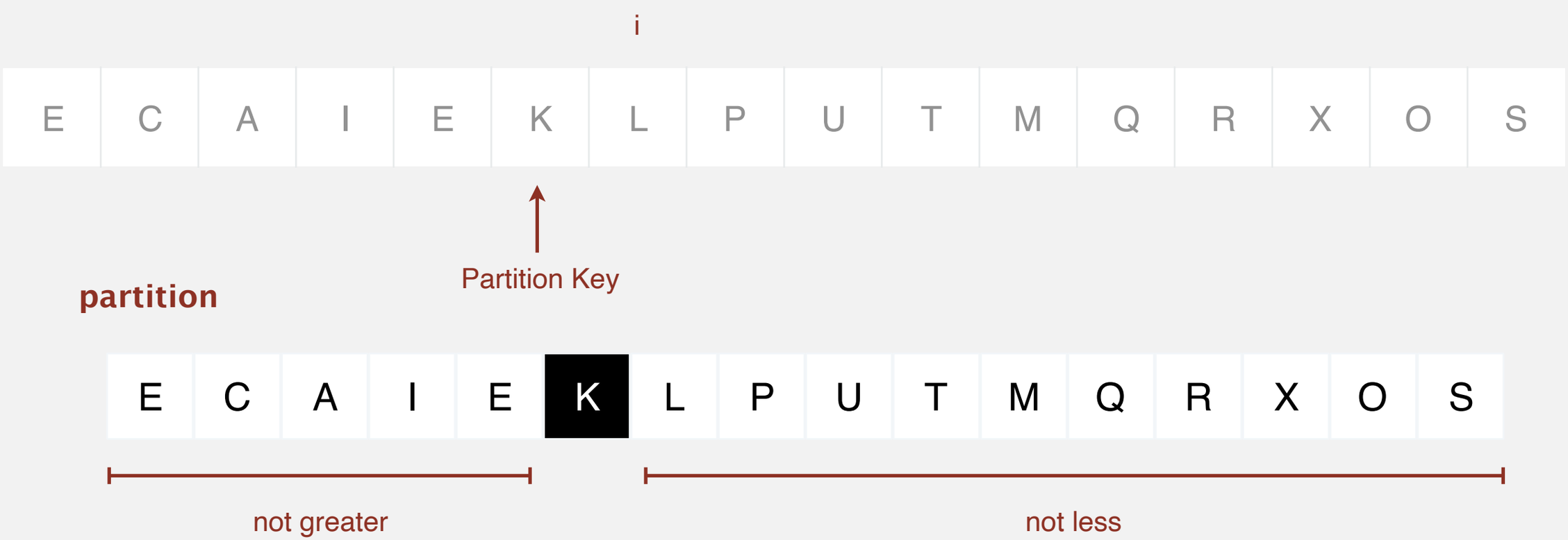
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



Quicksort partitioning demo

Select first element as partition key

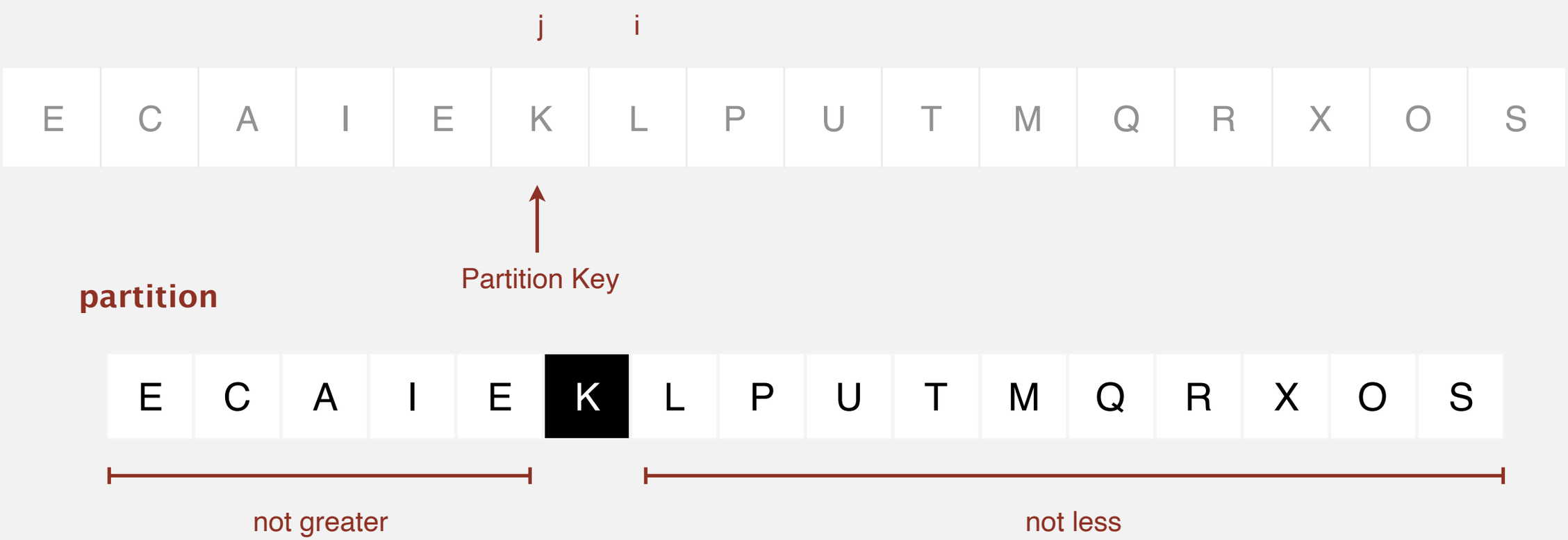
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].



Quicksort partitioning demo

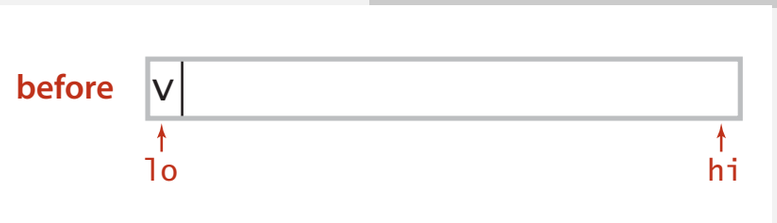
Select first element as partition key

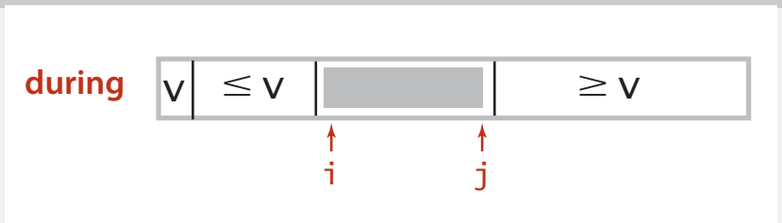
- Set i = lo + 1
- Set j = hi
- Scan i from left to right, until a[i] > a[lo]
- Scan j from right to left, until a[j] < a[lo]
- Swap a[i] and a[j] if a[i] > a[lo] and a[j] < a[0]
- When i and j cross, exchange a[lo] with a[j].

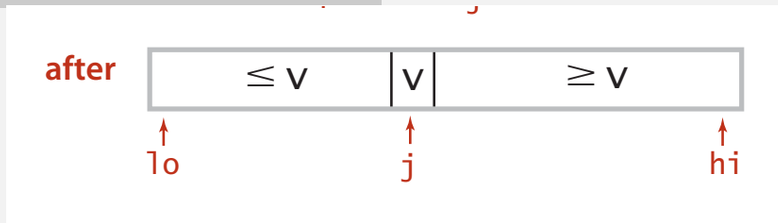


Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi) {
  int i = lo, j = hi+1;
  while (true) {
    while (less(a[++i], a[lo]))
      if (i == hi) break;                    find item on left to swap

    while (less(a[lo], a[--j]))
      if (j == lo) break;                    find item on right to swap

    if (i >= j) break;                       check if pointers cross
    exch(a, i, j);                           swap
  }

  exch(a, lo, j);                            swap with partitioning item
  return j;                                  return index of item now known to be in place
}
```







Quicksort: Java implementation

```
public class Quick {
  private static int partition(Comparable[] a, int lo, int hi)
  {  /* see previous slide */  }

  public static void sort(Comparable[] a) {
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
  }

  private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
  }
}
```

shuffle needed for performance guarantee (stay tuned)

Quicksort trace

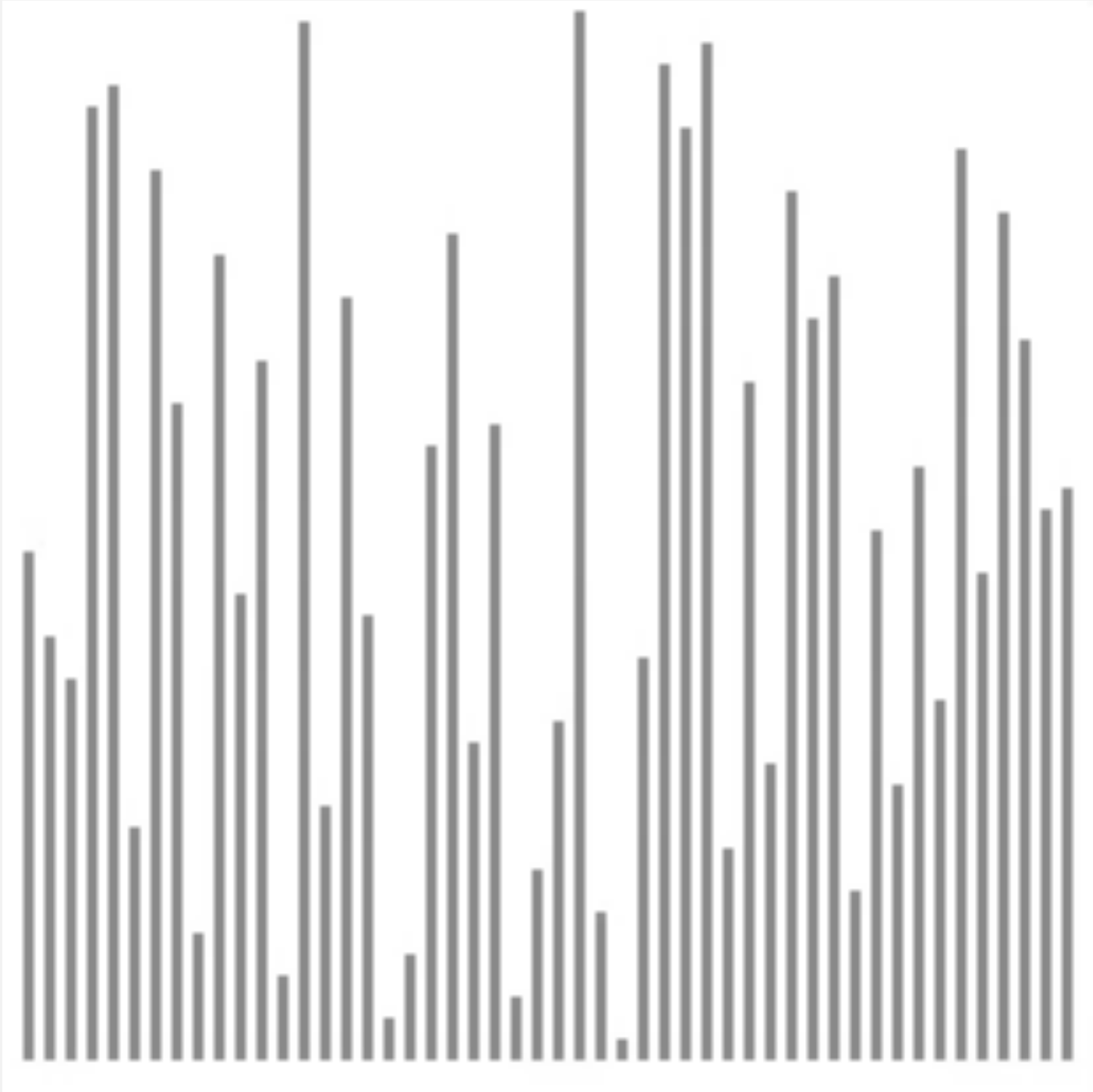
	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

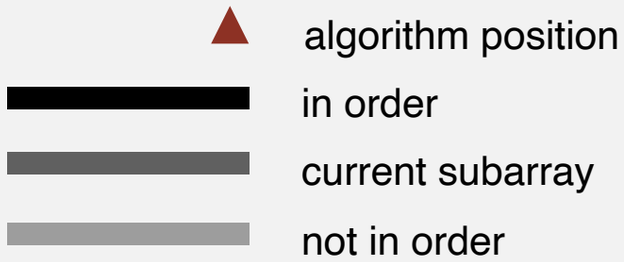
no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

Quicksort animation

50 random items

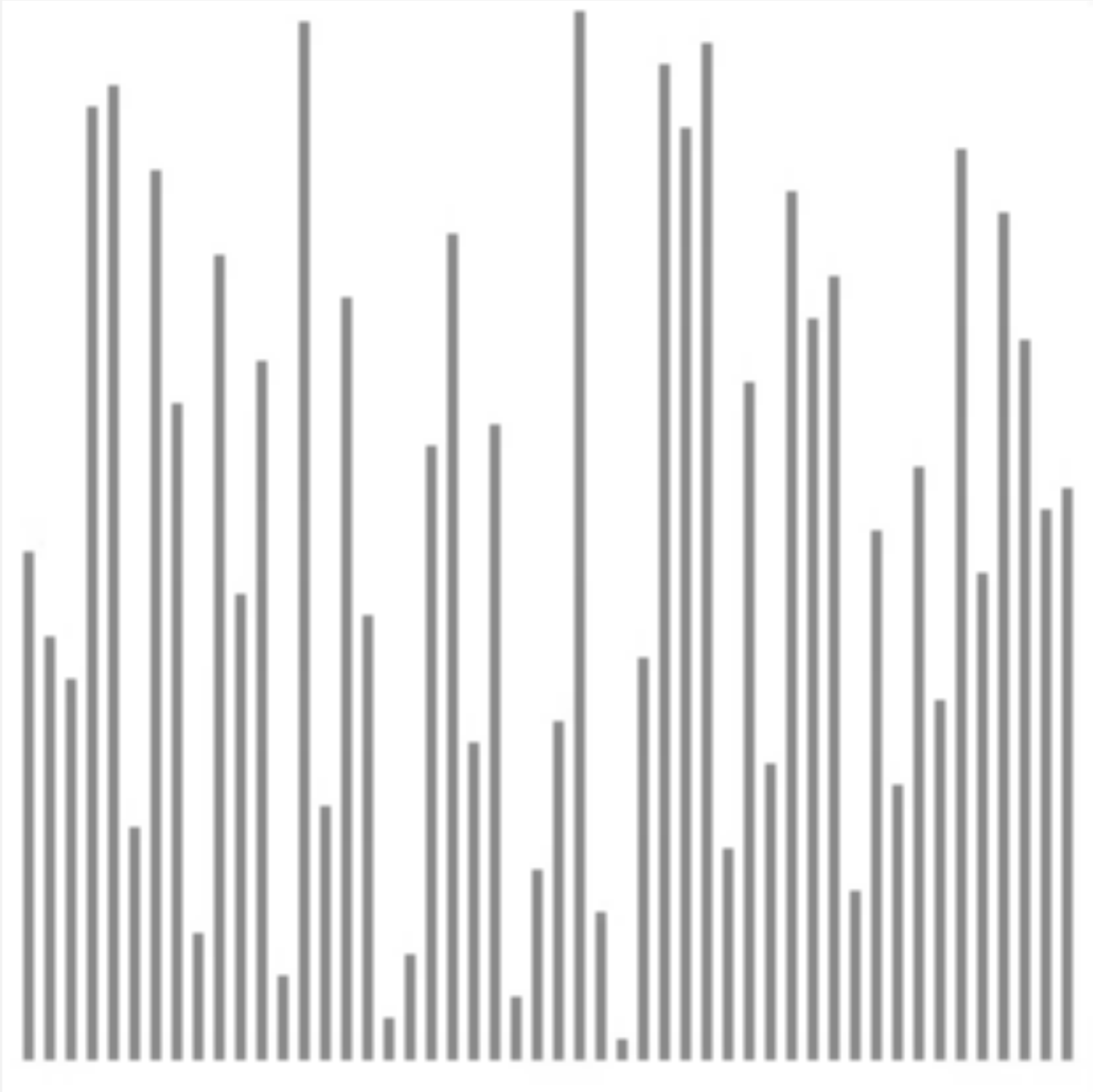


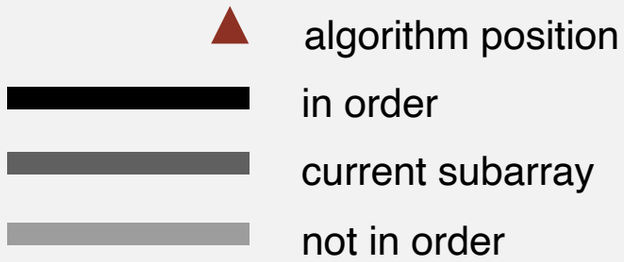


http://www.sorting-algorithms.com/quick-sort

Quicksort animation

50 random items





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

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key.

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key.

Preserving randomness. Shuffling is needed for performance guarantee.

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key.

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random partitioning item in each subarray.

Quicksort: empirical analysis (1961)

Running time estimates:

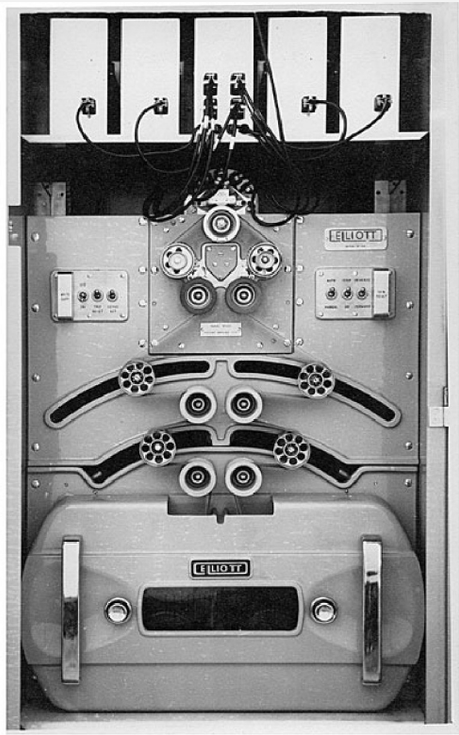
- Algol 60 implementation.
- National-Elliott 405 computer.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting N 6-word items with 1-word keys



Elliott 405 magnetic disc (16K words)

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: best-case analysis

Best case. Number of compares is $\sim N \lg N$.

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39\, N \lg N$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39\, N \lg N$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Best case. Number of compares is $\sim N \lg N$.

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39\, N \lg N$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Best case. Number of compares is $\sim N \lg N$.

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

[but more likely that lightning bolt strikes computer during execution]

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (requires using an explicit stack)

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (requires using an explicit stack)

Proposition. Quicksort is not stable.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (requires using an explicit stack)

Proposition. Quicksort is not stable.

Pf. [by counterexample]

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items

3-way quick sort.

- Partition array into three sections:
  - Less than partition key
  - Equal to partition key
  - Greater than partition key
- Only continue on left and right portions of partitions (less than, greater than)

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items

~ 12/7 N ln N compares (14% less)

~ 12/35 N ln N exchanges (3% more)

3-way quick sort.

- Partition array into three sections:
  - Less than partition key
  - Equal to partition key
  - Greater than partition key
- Only continue on left and right portions of partitions (less than, greater than)

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log_3 N$	?	$c\, N^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✔	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✔		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✔		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail