

ALG 09

Radix sort

Counting sort

Overview of sorts asymptotic complexities

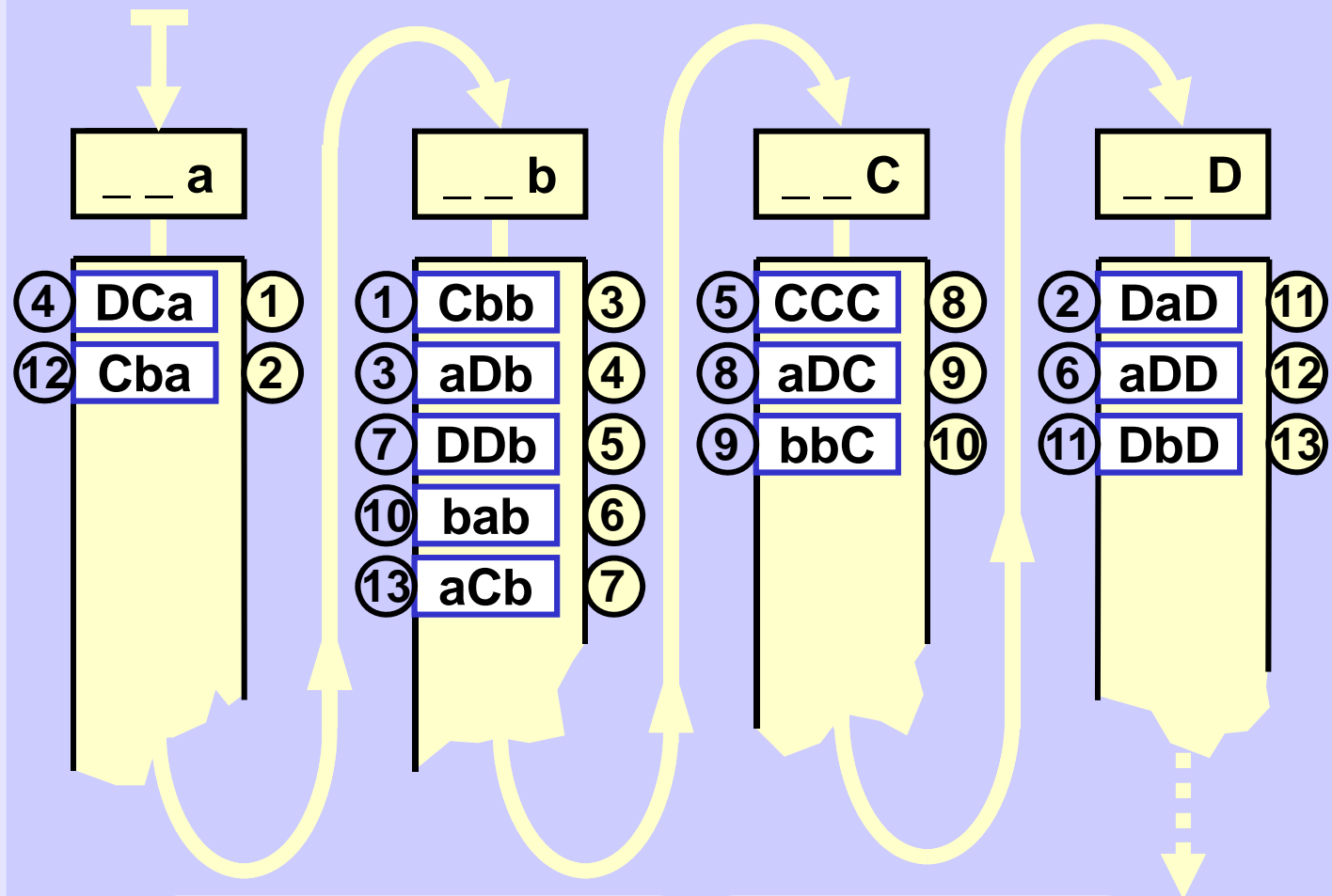
Sorting experiment

Radix sort

Unsorted

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

Sort by the 3rd symbol



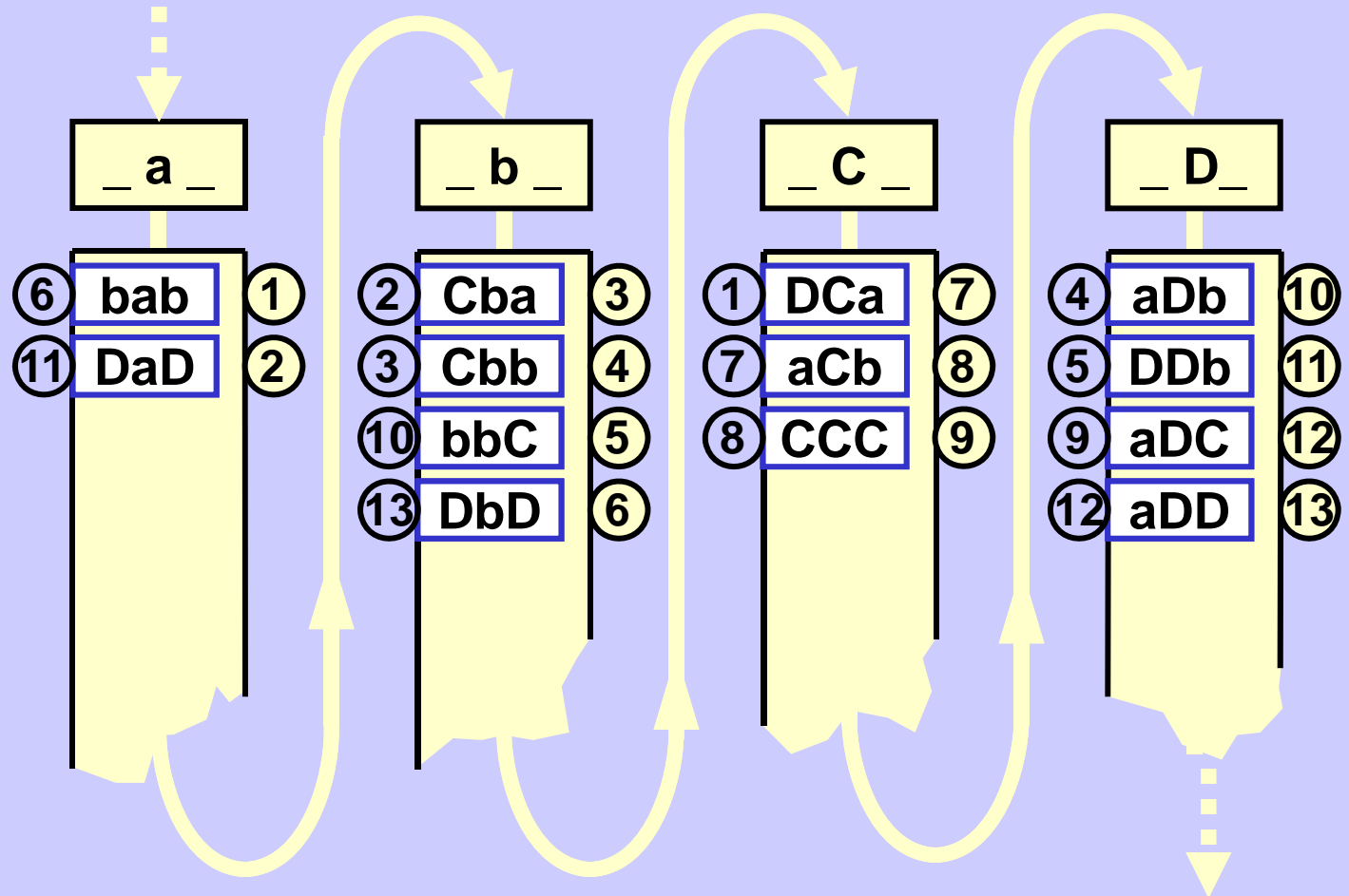
Previous order ① — New order ①

Radix sort

Sorted from
3rd symbol

- | | |
|---|-----|
| ① | DCa |
| ② | Cba |
| ③ | Cbb |
| ④ | aDb |
| ⑤ | DDb |
| ⑥ | bab |
| ⑦ | aCb |
| ⑧ | CCC |
| ⑨ | aDC |
| ⑩ | bbC |
| ⑪ | DaD |
| ⑫ | aDD |
| ⑬ | DbD |

Sort by the 2nd symbol

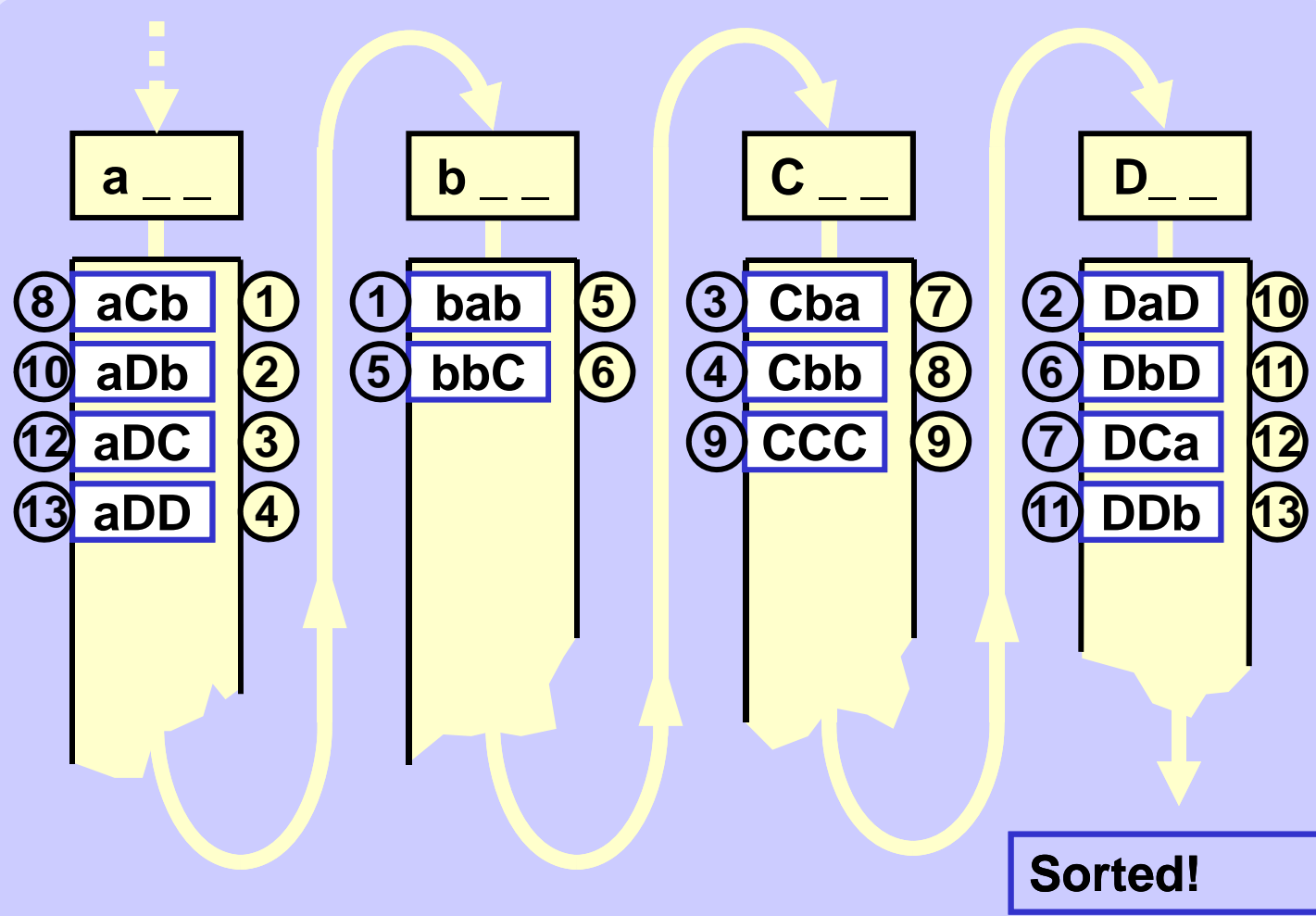


Radix sort

Sorted form
2nd symbol

①	bab
②	DaD
③	Cba
④	Cbb
⑤	bbC
⑥	DbD
⑦	DCa
⑧	aCb
⑨	CCC
⑩	aDb
⑪	DDb
⑫	aDC
⑬	aDD

Sort by the 1st symbol

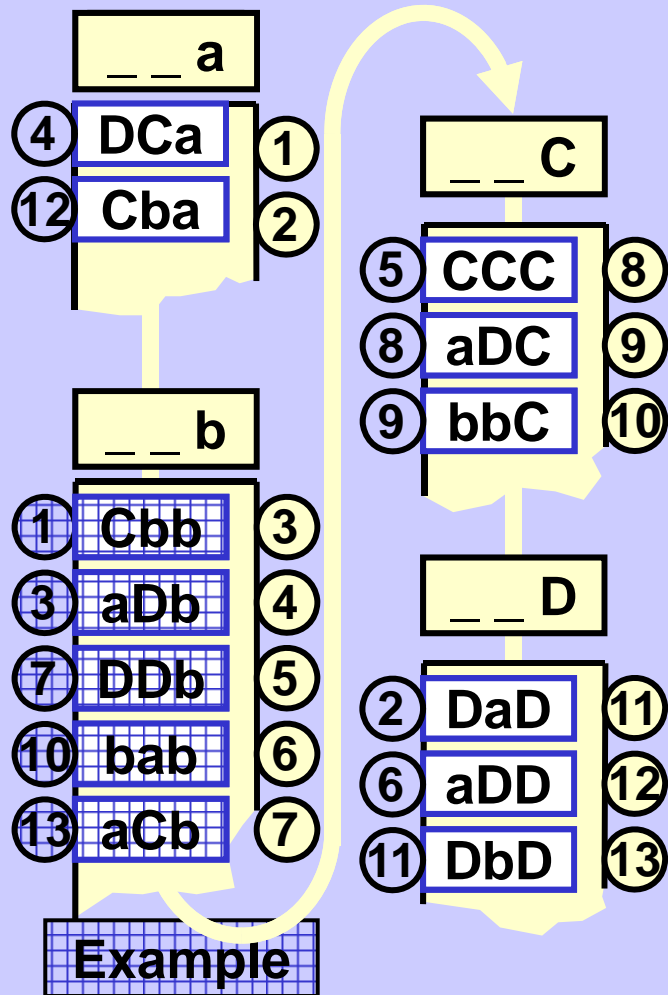


Radix sort Implementation

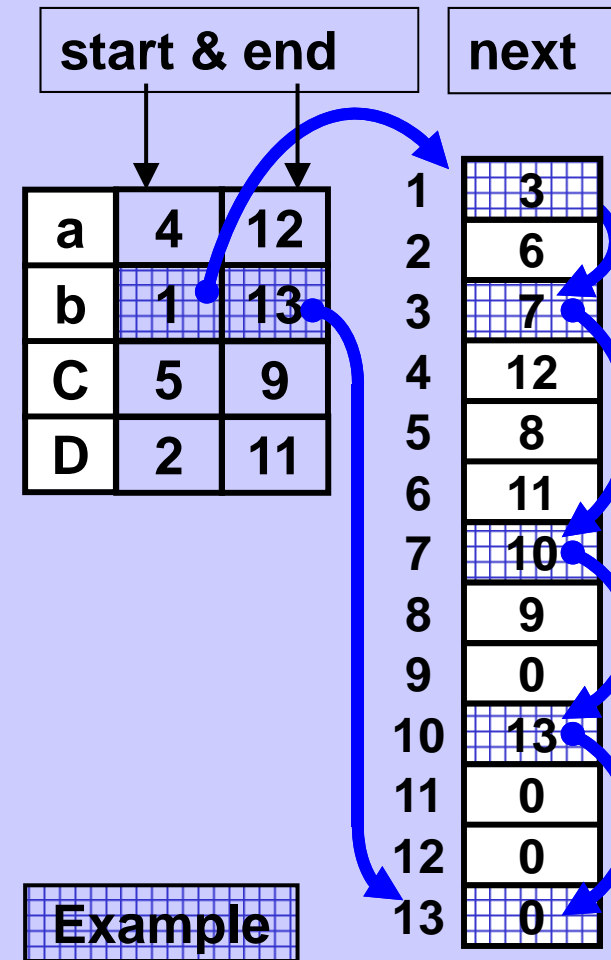
Unsorted

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

Sorted by the 3rd symbol



Auxiliary index arrays register modified order



Radix sort Implementation

Unsorted

①	Cbb
②	DaD
③	aDb
④	DCa
⑤	CCC
⑥	aDD
⑦	DDb
⑧	aDC
⑨	bbC
⑩	bab
⑪	DbD
⑫	Cba
⑬	aCb

One array for all lists

3
6
7
12
8
11
10
9
0
13
0
0
0

Example:
The list
in the
"b" slot

	s	e
a	4	12
b	1	13
C	5	9
D	2	11

Array of pointers
to the start and to
the end of the list
for each symbol

Here, both arrays
reflect the status
after sorting by
the 3rd character.



Radix sort can be performed
without moving the original data.

It suffices just to manipulate the
pointer arrays which contain
all information about the
current progress of the sort.

Radix sort Implementation

Unsorted

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

After: sorted by the 2nd symbol

9
0
7
13
0
0
8
6
11
2
0
1
5

	s	e
a	10	2
b	12	11
C	4	5
D	3	6

After: sorted by the 1st symbol = all sorted

5
11
8
7
0
0
0
6
0
9
4
1
3

	s	e
a	13	6
b	10	9
C	12	5
D	2	7

Example:
The list
in the
"b" slot

Radix sort Implementation

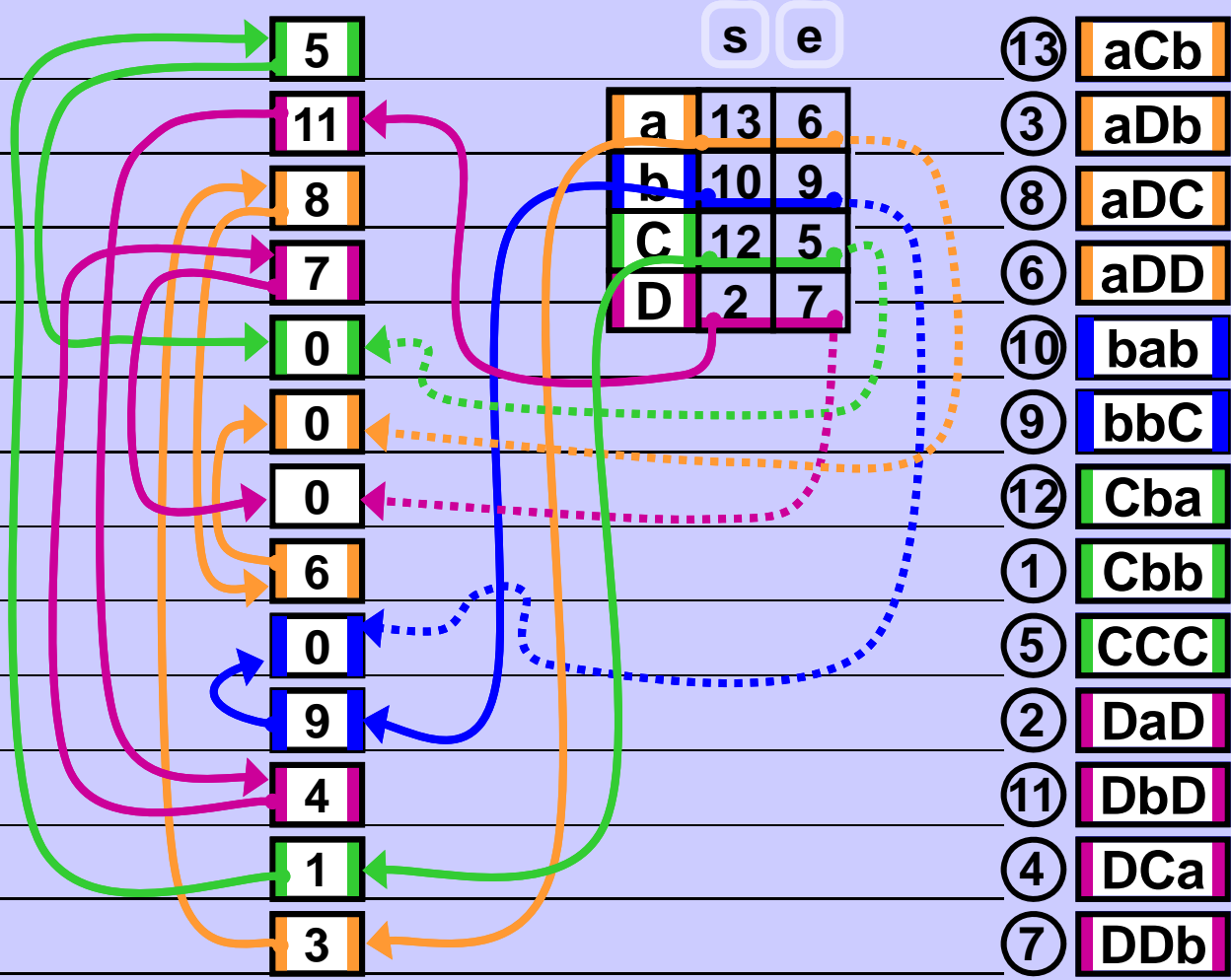
Unsorted

- ① Cbb
- ② DaD
- ③ aDb
- ④ DCa
- ⑤ CCC
- ⑥ aDD
- ⑦ DDb
- ⑧ aDC
- ⑨ bbC
- ⑩ bab
- ⑪ DbD
- ⑫ Cba
- ⑬ aCb

After: sorted by the 1st symbol = all sorted

Just print the data in the order given by the lists:

a → b → C → D →



From sorted by 2nd symbol to sorted by 1st symbol

Arrays specify order after sorted by the 2nd symbol.

	s	e
a	10	2
b	12	11
C	4	5
D	3	6

1	Cbb	9	4
2	DaD	0	2
3	aDb	7	10
4	DCa	13	7
5	CCC	0	9
6	aDD	0	13
7	DDb	8	11
8	aDC	6	12
9	bbC	11	5
10	bab	2	1
11	DbD	0	6
12	Cba	1	3
13	aCb	5	8

d



	s1	e1
a	0	0
b	0	0
C	0	0
D	0	0

Arrays will specify order after sorted by the 1st symbol.

0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

Update arrays s, e, d:
Fill temporary arrays s1, e1, d1 and copy their contents back to s, e, d.

Implementation will not copy anything, it will only swap the pointers to the original and temporary arrays.

From sorted by 2nd symbol to sorted by 1st symbol

Sorted by the 2nd symbol

	s	e
a	10	2
b	12	11
C	4	5
D	3	6

①	Cbb	9	④
②	DaD	0	②
③	aDb	7	⑩
④	DCa	13	⑦
⑤	CCC	0	⑨
⑥	aDD	0	⑬
⑦	DDb	8	⑪
⑧	aDC	6	⑫
⑨	bbC	11	⑤
⑩	bab	2	①
⑪	DbD	0	⑥
⑫	Cba	1	③
⑬	aCb	5	⑧

d

	s1	e1
a	0	0
b	0	0
C	0	0
D	0	0

0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

	s1	e1
a	0	0
b	10	10
C	0	0
D	0	0

0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

	s1	e1
a	0	0
b	10	10
C	0	0
D	2	2

0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

	s1	e1
a	0	0
b	10	10
C	12	12
D	2	2

0
0
0
0
0
0
0
0
0
0
0
0
0
0

d1

	s1	e1
a	0	0
b	10	10
C	12	1
D	2	2

0
0
0
0
0
0
0
0
0
0
0
0
1
0

d1

	s1	e1
a	0	0
b	10	9
C	12	1
D	2	2

0
0
0
0
0
0
0
0
0
0
0
0
0
1
0

d1

	s1	e1
a	0	0
b	10	9
C	12	1
D	2	11

0
11
0
0
0
0
0
0
0
0
0
0
0
1
0

d1

From sorted by 2nd symbol to sorted by 1st symbol

Sorted by the 2nd symbol

	s	e
a	10	2
b	12	11
C	4	5
D	3	6

1	Cbb	9	4
2	DaD	0	2
3	aDb	7	10
4	DCa	13	7
5	CCC	0	9
6	aDD	0	13
7	DDb	8	11
8	aDC	6	12
9	bbC	11	5
10	bab	2	1
11	DbD	0	6
12	Cba	1	3
13	aCb	5	8

d

	s1	e1	s1	e1	s1	e1	s1	e1	s1	e1	s1	e1
a	0	0	13	13	13	13	13	3	13	3	13	8
b	10	9	10	9	10	9	10	9	10	9	10	9
C	12	1	12	1	12	5	12	5	12	5	12	5
D	2	4	2	4	2	4	2	4	2	7	2	7

0	0	5	5	5	5	5	5
11	11	11	11	11	11	11	11
0	0	0	0	0	0	8	8
0	0	0	0	0	7	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
9	9	9	9	9	9	9	9
4	4	4	4	4	4	4	4
1	1	1	1	1	1	1	1
0	0	0	0	3	3	3	3

d1

d1

d1

d1

d1

d1

d1

Finished

Radix sort Implementation

```
def radixSort(A):  
    alphabetsize = 128      # 2^16 in unicode  
    S = [0] * alphabetsize  # all arrays  
    E = [0] * alphabetsize  
    D = [0] * alphabetsize  
    S1 = [0] * alphabetsize  
    E1 = [0] * alphabetsize  
    D1 = [0] * alphabetsize  
  
    radixInit(A, S, E, D)   # 1st pass with last char  
  
    for p in range(len(A[0])-2, -1, -1):  
        radixStep(A, p, S, E, D, S1, E1, D1)  
        S, S1 = S1, S      # just swap arrays  
        E, E1 = E1, E      # ditto  
        D, D1 = D1, D      # ditto  
  
    radixOutput(A, S, E, D) # print sorted A
```

Radix sort Implementation

```

def radixInit(A, S, E, D):
    pos = len(A[0]) - 1          # last char in string
    for i in range(len(S)):
        S[i], E[i] = -1, -1
    for i in range(len(A)):
        c = ord(A[i][pos])      # char to index
        if S[c] == -1:
            S[c], E[c] = i, i   # start new list
        else:
            D[E[c]], E[c] = i, i # extend existing list

```

Add spaces to the shorter strings to make all strings of the same length.

**Caution: The arrays in the code are indexed from 0,
The arrays in the code are indexed from 1.**

Radix sort Implementation

```

def radixStep(A, pos, S, E, D, S1, E1, D1):
    for i in range(len(S)):
        S1[i], E1[i] = -1, -1    # init arrays

    for i in range(len(S)):
        if S[i] != -1:          # unempty old list
            j = S[i]            # traverse the list
            while True:
                c = ord(A[j][pos])    # list index
                if S1[c] == -1:
                    S1[c], E1[c] = j, j # start new list
                else:                # extend existing list
                    D1[E1[c]], E1[c] = j, j
                if j == E[i]:
                    break
            j = D[j]              # next string index

```

Radix sort

Resume

d symbols d loops

loop $\Theta(n)$ operations

total $\Theta(d \cdot n)$ operations

$d \ll n \Rightarrow$ $\Theta(n)$ operations

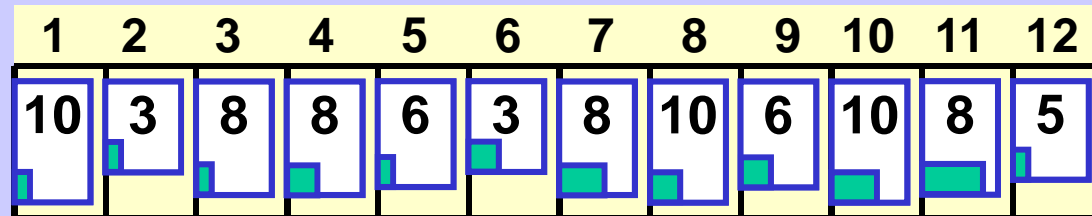
Radix sort does not change the order of equal values.

Asymptotic complexity of Radix sort is $\Theta(d \cdot n)$

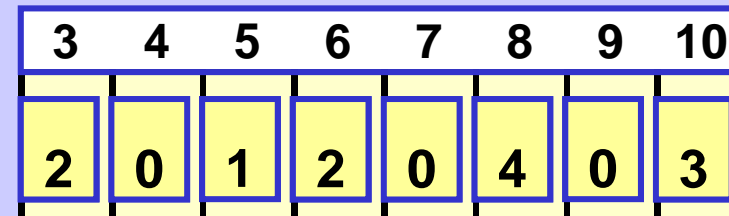
It is a stable sort.

Counting sort

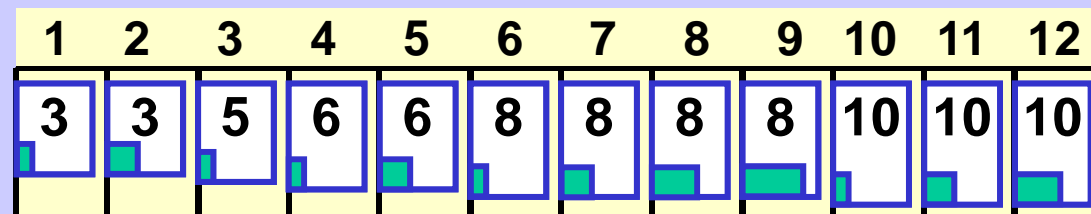
Input
input.length == N



Frequency
frequency.length == k
 $k = \max(\text{input}) - \min(\text{input}) + 1$



Output
output.length == N



Counting sort

Step 1

Reset frequency array

3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0

One pass through the input array

----->

10	3	8	8	6	3	8	10	6	10	8	5
----	---	---	---	---	---	---	----	---	----	---	---

Fill the frequency array

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

```
for x in inputArr:  
    freq[x] += 1
```

Counting sort

Step 2

One pass

low = 3

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

high = 10

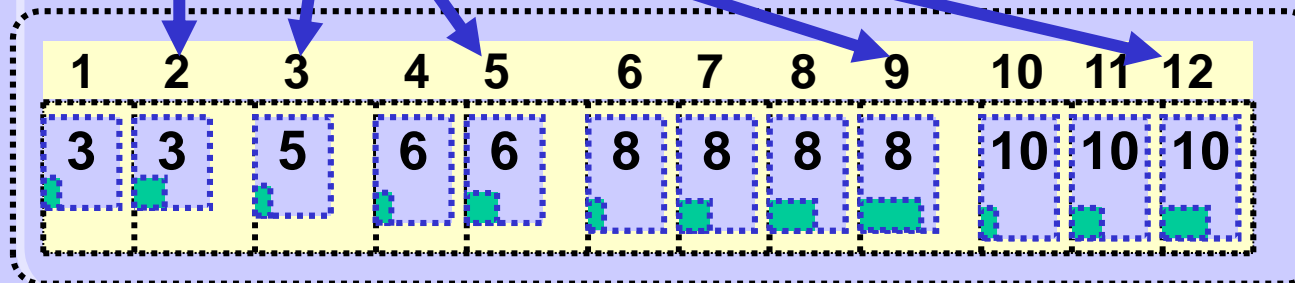
Frequency array
changes its
meaning

Update the frequency array

```
for i in range(low+1, high+1):
    freq[i] += freq[i-1]
```

3	4	5	6	7	8	9	10
2	2	3	5	5	9	9	12

Elem `freq[j]` denotes position
of the last elem with value `j`
in the output array



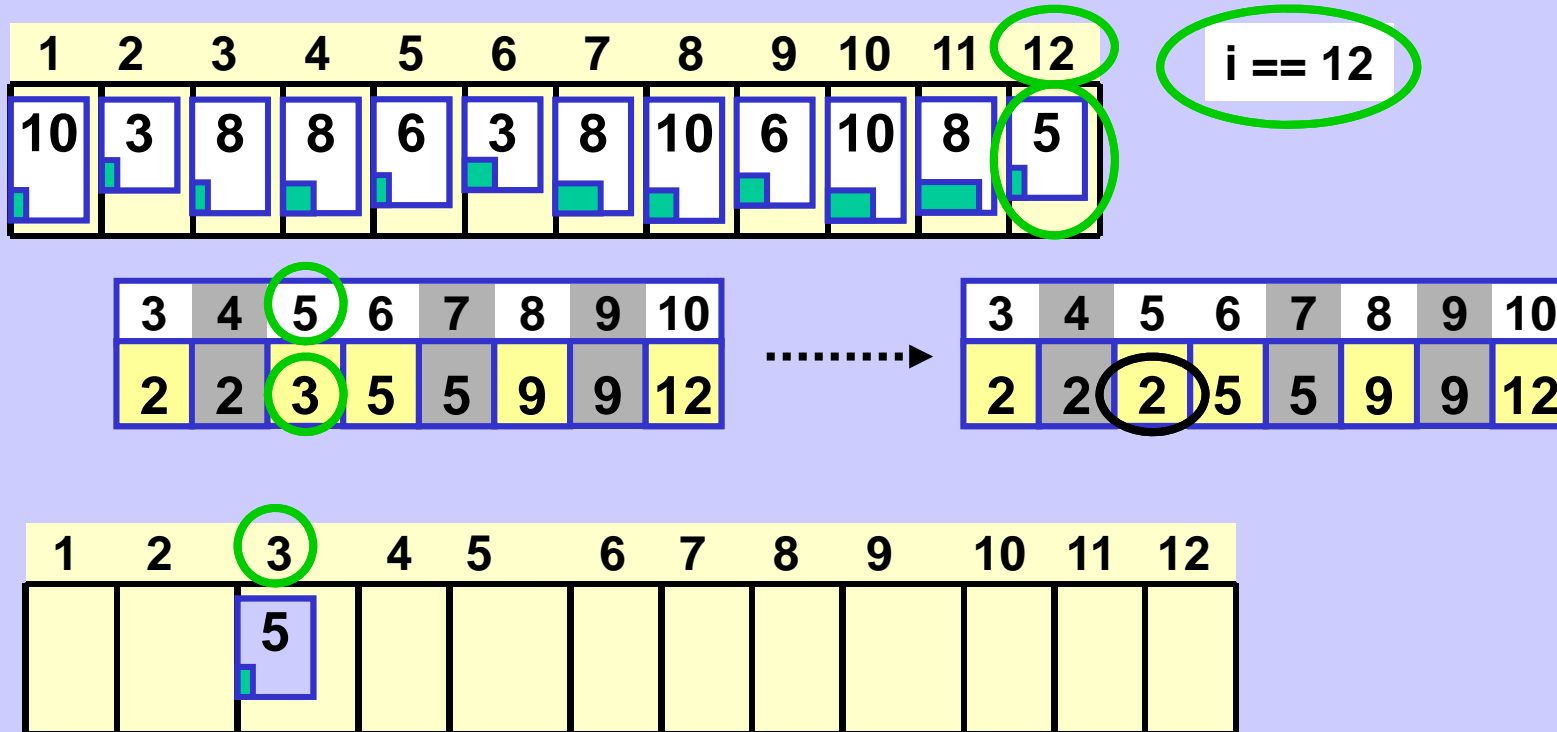
Counting sort

Step 3

`i == N`

```

for i in range(N, 0, -1):
    output[freq[input[i]]] = input[i]
    freq[input[i]] -= 1
  
```



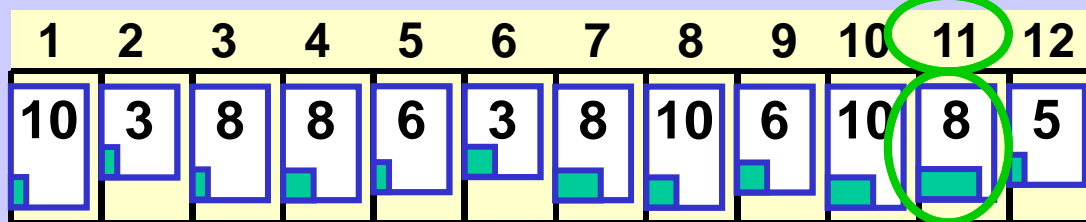
Counting sort

Step 3

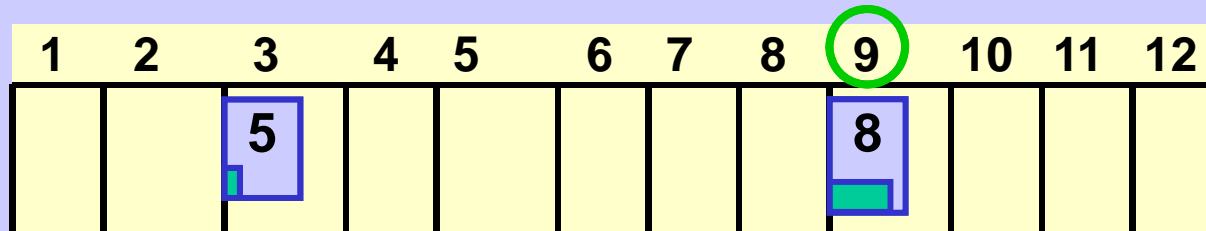
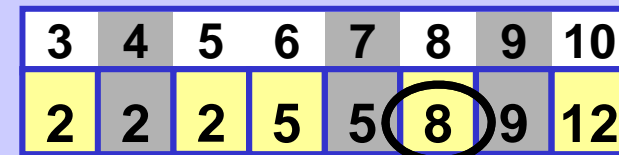
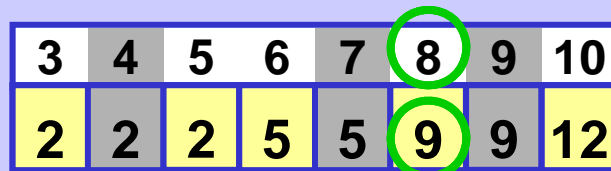
`i == N-1`

```

for i in range(N, 0, -1):
    output[freq[input[i]]] = input[i]
    freq[input[i]] -= 1
  
```



`i == 11`



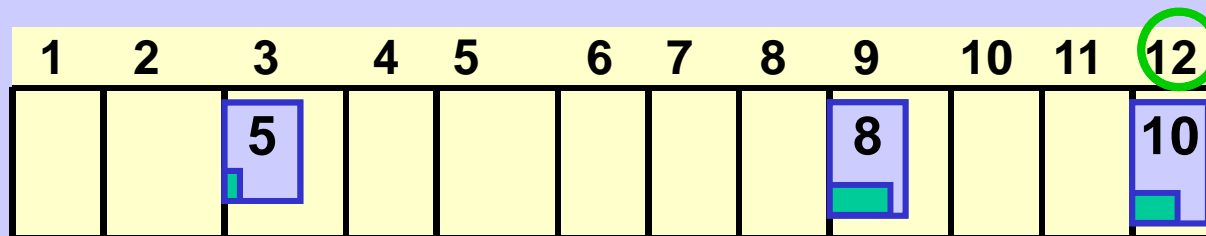
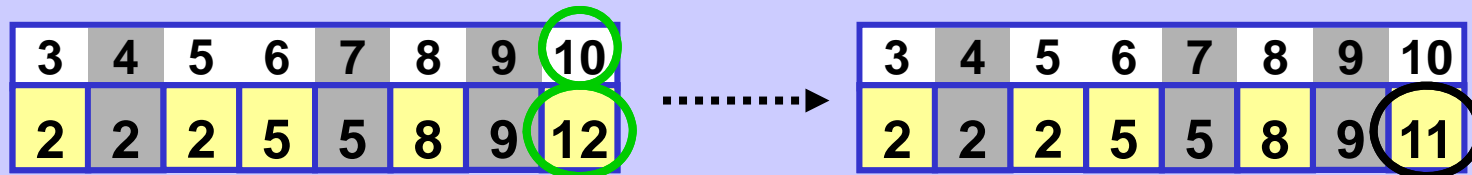
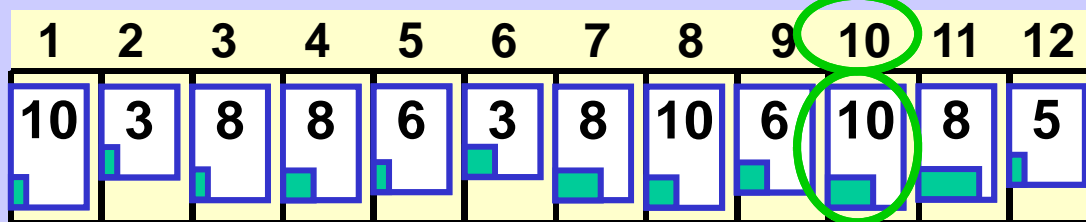
Counting sort

Step 3

$i == N-2$

```

for i in range(N, 0, -1):
    output[freq[input[i]]] = input[i]
    freq[input[i]] -= 1
  
```



etc...

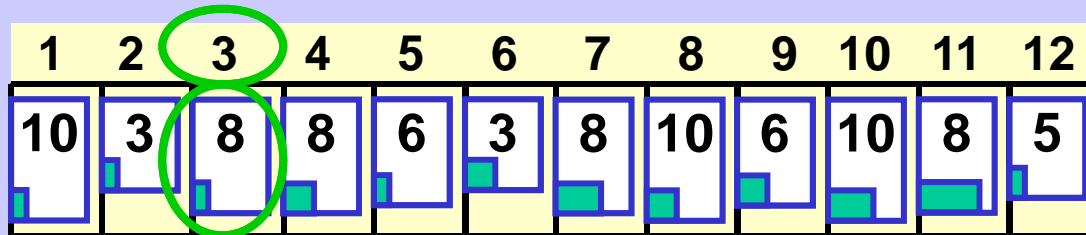
Counting sort

Step 3

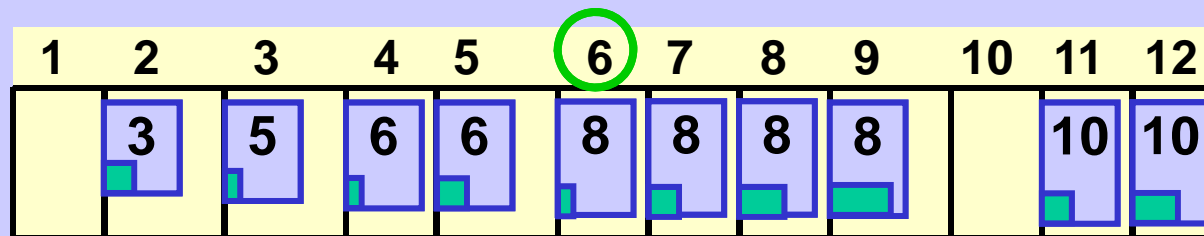
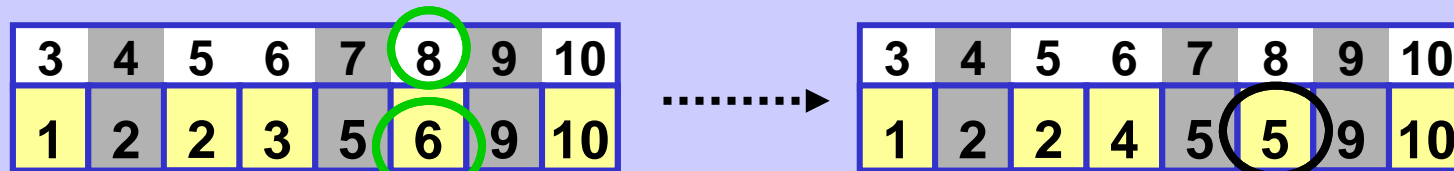
`i == 3`

```

for i in range(N, 0, -1):
    output[freq[input[i]]] = input[i]
    freq[input[i]] -= 1
  
```



`i == 3`



etc...

Sorts complexities overview

Array size n	Worst case	Best case	Average, "expected" case	Stable
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	No
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Yes
Bubble sort *)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Yes
Quick sort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	No **)
Merge sort	$\Theta(n \cdot \log(n))$	$O(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Yes
Heap sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	No
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Yes
Counting sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Yes

*) Not recommended to use

**) Stable slow versions exist

Small sorting experiment

Environment

Intel(R) 1.8 GHz, Microsoft Windows XP SP3, jdk 1.6.0_16.

Organization

**Explored the sorts which compare the elements value (double).
Each datasets of particular datasizes used in all sorts.
Arrays values generated by the pseudorandom generator.
The results are the averages of repeated runs.**

Conclusion

**There is no particular sort method which would be
optimal in all circumstances.
The performance is influenced by the data size and by the degree of
the original organisation (partial order) of the data.**

Small sorting experiment

Array size	% sorted	Time in milliseconds if not indicated otherwise					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	0.0005	★ 0.0002	0.0005	0.0004	0.0009	0.0005
10	90%	0.0004	★ 0.0001	0.0004	0.0004	0.0007	0.0005
100	0%	0.028	0.016	0.043	0.081	0.014	★ 0.011
100	90%	0.026	★ 0.003	0.030	0.010	0.011	0.011
1 000	0%	2.36	1.30	4.45	★ 0.12	0.19	0.17
1 000	90%	2.31	0.18	2.86	0.16	★ 0.15	0.16
10 000	0%	228	130	450	★ 1.57	2.40	2.31
10 000	90%	229	17.5	285	1.93	★ 1.68	2.11
100 000	0%	22 900	12 800	45 000	★ 18.7	31.4	31.4
100 000	90%	22 900	1 760	28 500	27.4	★ 24.6	25.5
1 000 000	0%	38 min	22 min	75 min	★ 237	385	570
1 000 000	90%	38 min	2.9 min	47.5 min	336	★ 301	381

Degree of order. The array is first sorted and then x% of randomly chosen elements randomly change value.

Small sorting experiment

Array size	% sorted	Ratio of slowdown (>1) compared to Quick sort					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	1.3	★ 0.7	1.4	1	✗ 2.5	1.4
10	90%	1	★ 0.26	0.96	1	✗ 1.8	1.3
100	0%	3.4	✗ 1.8	5.4	★ 1	1.75	1.35
100	90%	2.46	★ 0.28	2.9	1	✗ 1.07	1.07
1 000	0%	20	✗ 11	37.5	★ 1	1.65	1.4
1 000	90%	15	✗ 1.2	18.5	1	★ 0.95	1.03
10 000	0%	146	✗ 83	287	★ 1	1.53	1.48
10 000	90%	118	✗ 9.1	148	1	★ 0.87	1.09
100 000	0%	1 220	✗ 686	2 410	★ 1	1.7	1.7
100 000	90%	837	✗ 64.1	1 040	1	★ 0.9	0.93
1 000 000	0%	9 960	✗ 5 400	19 000	★ 1	1.6	2.41
1 000 000	90%	6 820	521	8 480	1	★ 0.9	1.14

Fastest



Slowest



Stable



Selection and Bubble sort do not compete.

Small sorting experiment

Array size	% sorted	Ratio of slowdown (> 1) when comparing the sort speeds of unsorted and partially sorted data.					
		Sort <input type="checkbox"/>					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	1	1	1	1	1	1
10	90%	0.8	0.5	0.8	1	0.8	1
100	0%	1	1	1	1	1	1
100	90%	0.9	0.2	0.68	1.27	0.78	1
1 000	0%	1	1	1	1	1	1
1 000	90%	0.98	0.14	0.64	1.31	0.75	0.95
10 000	0%	1	1	1	1	1	1
10 000	90%	1.0	0.14	0.63	1.23	0.7	0.91
100 000	0%	1	1	1	1	1	1
100 000	90%	1.0	0.14	0.63	1.46	0.78	0.81
1 000 000	0%	1	1	1	1	1	1
1 000 000	90%	1.0	0.14	0.63	1.42	0.78	0.67

Stable