Sorting Key-Value Pairs

By Yufei Tao's Teaching Team

- We have learned how to sort a set of *n* integers in
  - $O(n \log n)$ time;
  - $O(n + U)$ time if every integer is from [1, *U*].

- In practice, sorting is often used to sort a set of "records" by their "keys" (e.g., sorting student records by student ID). In this tutorial, we will discuss how the algorithms we have learned can be adapted for this purpose.

The Problem of Sorting Key-Value Pairs

- Define a ***record*** to be a pair (k, v) where *k* is an integer referred to as the record's ***key***, and *v* is another integer referred to as the record's ***value***.

- **Input:** A set *S* of *n* records given in an array *A*, where *A*[*i*] stores the *i*-th pair of *S*.
  - Note: Some records may have the same key.
- **Output:** An array where the *n* records are arranged in non-ascending order of their keys.

Example

- Input:

$$S = \{\{9, v_1\}, \{7, v_2\}, \{2, v_3\}, \{6, v_4\}, \{2, v_5\}, \{7, v_6\}, \{1, v_7\}, \{2, v_8\}\}$$

- Initially we have the following array

Input Array

k_1	v_1	k_2	v_2	k_3	v_3	k_4	v_4	k_5	v_5	k_6	v_6	k_7	v_7	k_8	v_8
9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

- Rearrange the records so that their keys are non-descending:

Sorted Array

1	v_7	2	v_3	2	v_5	2	v_8	6	v_4	7	v_2	7	v_6	9	v_1

Adapting Comparison-Based Algorithms

- We have discussed 3 comparison-based sorting algorithms: selection sort, merge sort, and quick sort.
- Our discussions have assumed that the elements to be sorted are distinct. This assumption allows that every comparison has a clear "winner".
- We no longer have this property here as records can have the same key.
- It is possible to look at each individual algorithm and work out the necessary adaptation tailored for that algorithm.
- However, there is a "black-box" adaptation that works on all comparison-based sorting algorithms. This is the **composite-key** approach.

- Suppose that we have two records with the same key, e.g., (k, v_1) and (k, v_2). The two records' relative ordering in the output array is unimportant. We can utilize the property to create a "new key" for each record, ensuring that all the records' new keys are distinct.

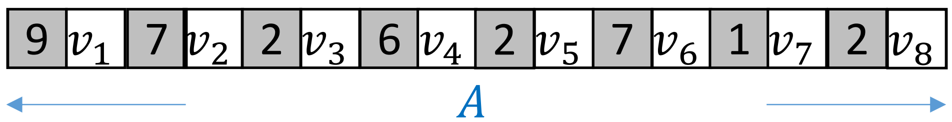
- Specifically, for each record (k, v), we will assign it a distinct ID, after which the record becomes (k, id, v).
  - We will treat (k, id) as the record's new key.

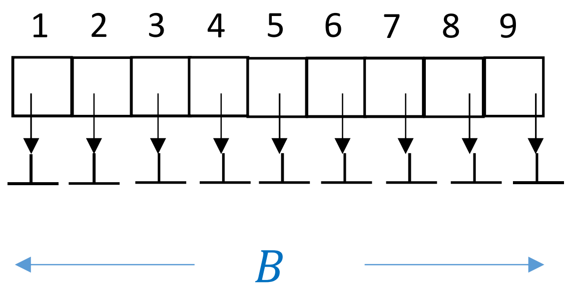
- Whenever an algorithm needs to compare two records (k_1, id_1, v_1) and (k_2, id_2, v_2), the comparison is resolved as follows:
  - If $k_1 < k_2$, then the first record is "smaller";
  - If $k_1 = k_2$ and $id_1 < id_2$, then the first record is "smaller";
  - Otherwise, the second record is "smaller".

- We can now apply merge sort to perform the sorting in $O(n \log n)$ time.

Adapting Counting Sort

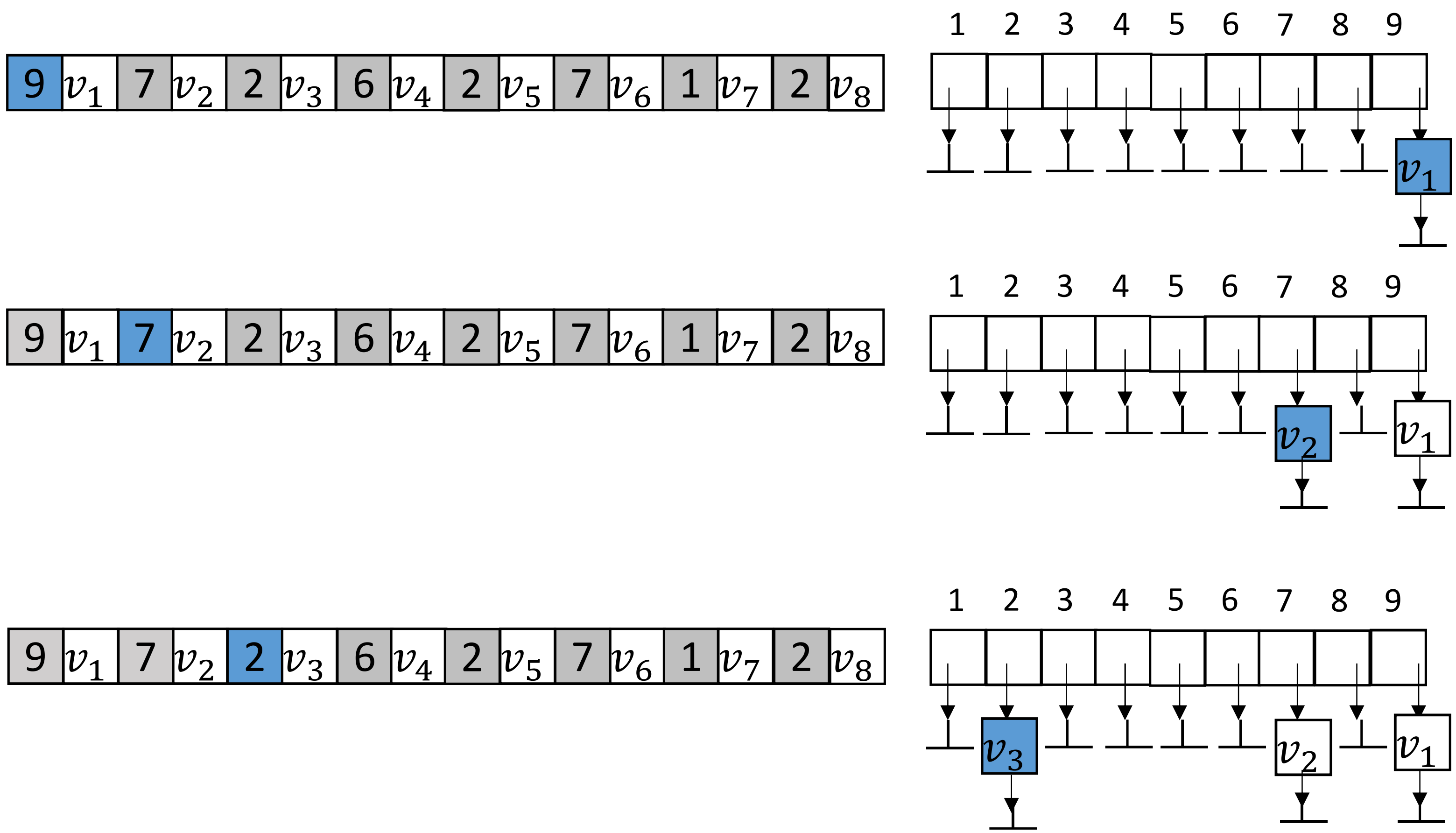
Counting Sort (Linked List Ver.)



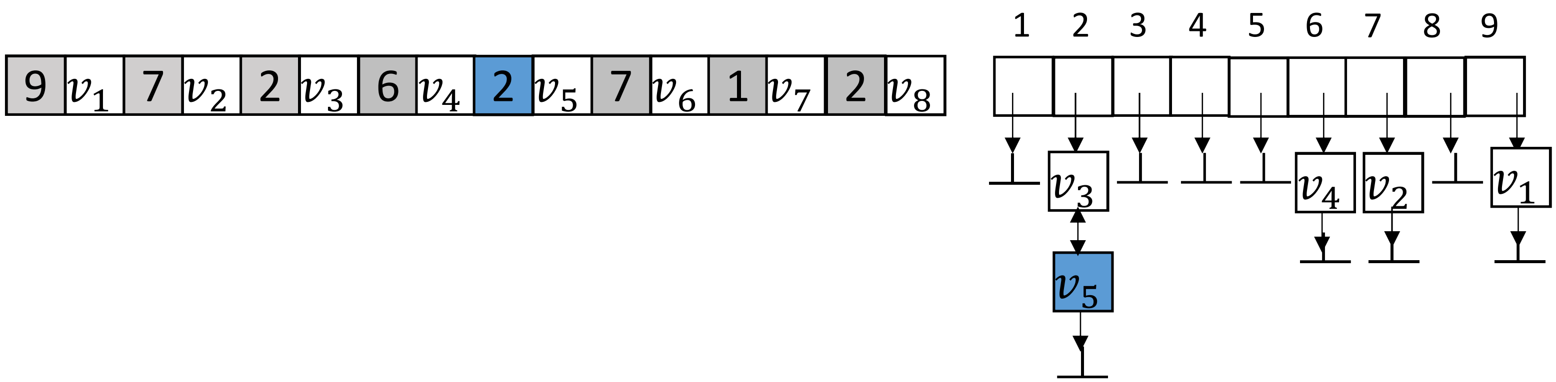


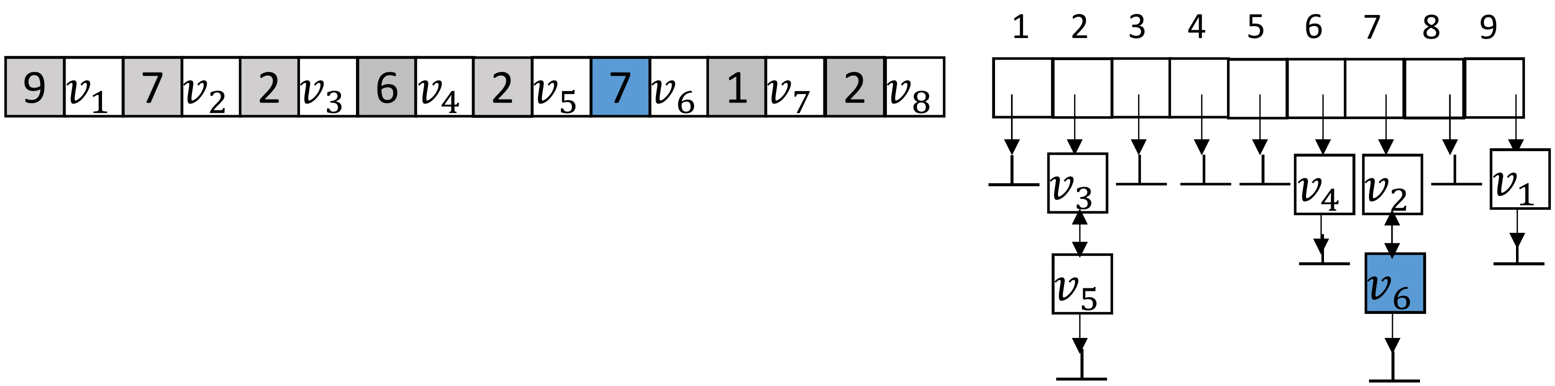
$\perp$: A null pointer

Counting Sort (Linked List Ver.)

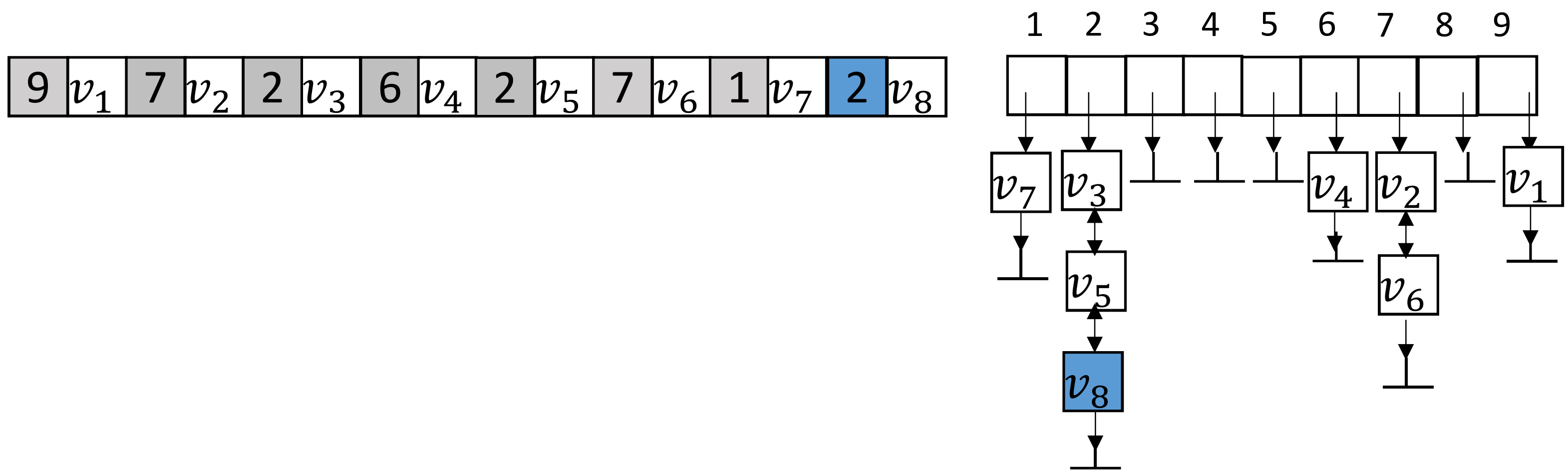


Counting Sort (Linked List Ver.)

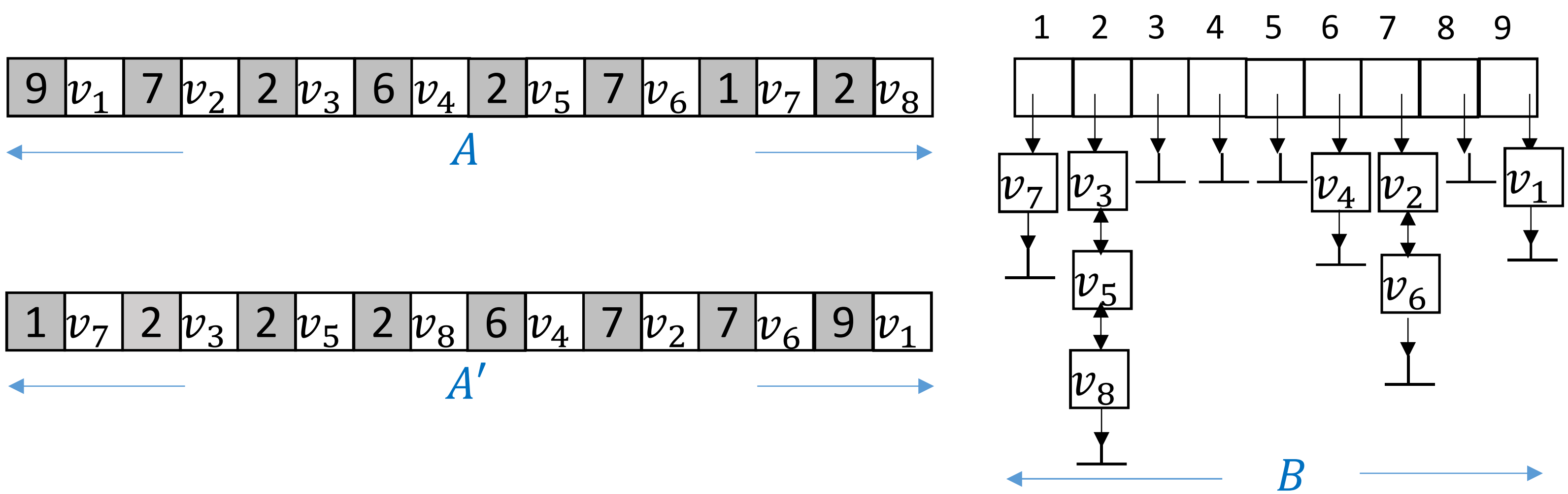




Counting Sort (Linked List Ver.)



Counting Sort (Linked List Ver.)



How do we produce the sorted array A'?

Scan array B. For each cell referencing a non-empty linked list, enumerate all the pairs therein.

Overall time complexity: $O(n + U)$

Next, we will give another version of
counting sort that does not use linked lists.

Counting Sort (2nd Ver.)

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

A

We first compute an array B to store the number of occurrences of each key.

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

B

The next slide will explain how to do so.

Counting Sort (2nd Ver.)

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

...

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

...

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

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

Counting Sort (2nd Ver.)

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

A

Then we will change *B* from

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

To array *C*

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

where $C[k] = \sum_{i=1}^{k} B[i]$ for every $k \in [1, n]$.
The next slide will explain how to do so in *O*(*U*) time.

Counting Sort (2nd Ver.)

For each $i \in [2, U]$, set $C[i] \leftarrow B[i] + C[i-1]$.

$i = 2$

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

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

$i = 3$

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

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

$i = 4$

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

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

……

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

Counting Sort (2nd Ver.)

A:

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

C:

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

Our goal is to produce

A':

1	v_7	2	v_3	2	v_5	2	v_8	6	v_4	7	v_2	7	v_6	9	v_1

We will scan *A* backward and keep an invariant:

> If (k, v) is the rightmost pair among all the pairs with key *k* in the non-scanned part of *A*, the position of (k, v) in A' is $C[k]$.

Counting Sort (2nd Ver.)

Scan array A from right to left.

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

A

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

C

Insert the pair to A' and update B.

1		2		3		4		5		6		7		8	
						2	v_8								

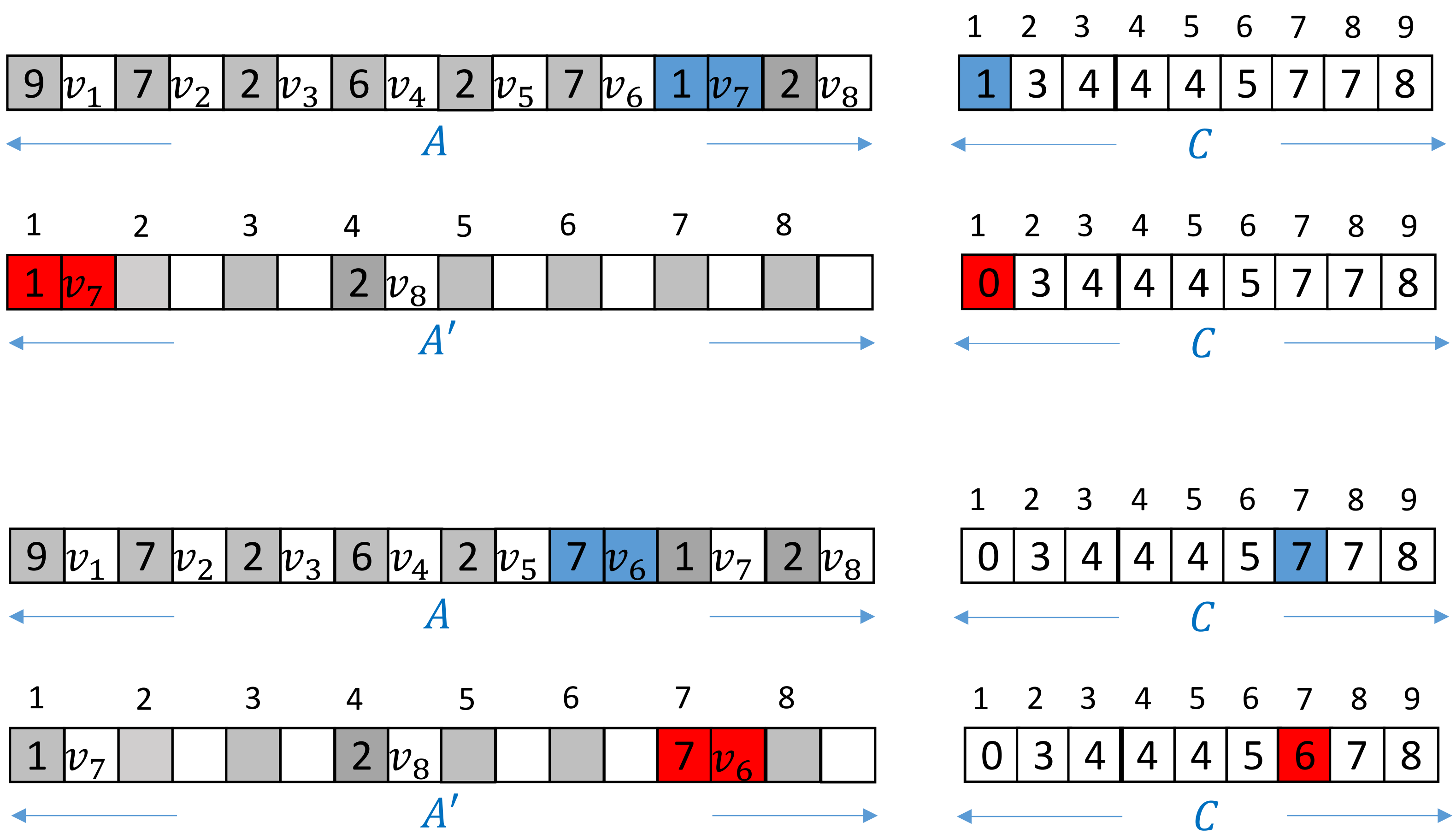
A'

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

C

If (k, v) is the rightmost pair among all the pairs with key k in the non-scanned part of A, the position of (k, v) in A' is $C[k]$.

Counting Sort (2nd Ver.)



Counting Sort (2nd Ver.)

A

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

C

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

A'

1		2		3		4		5		6		7		8	
1	v_7			2	v_5	2	v_8					7	v_6		

C

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

A

9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

C

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

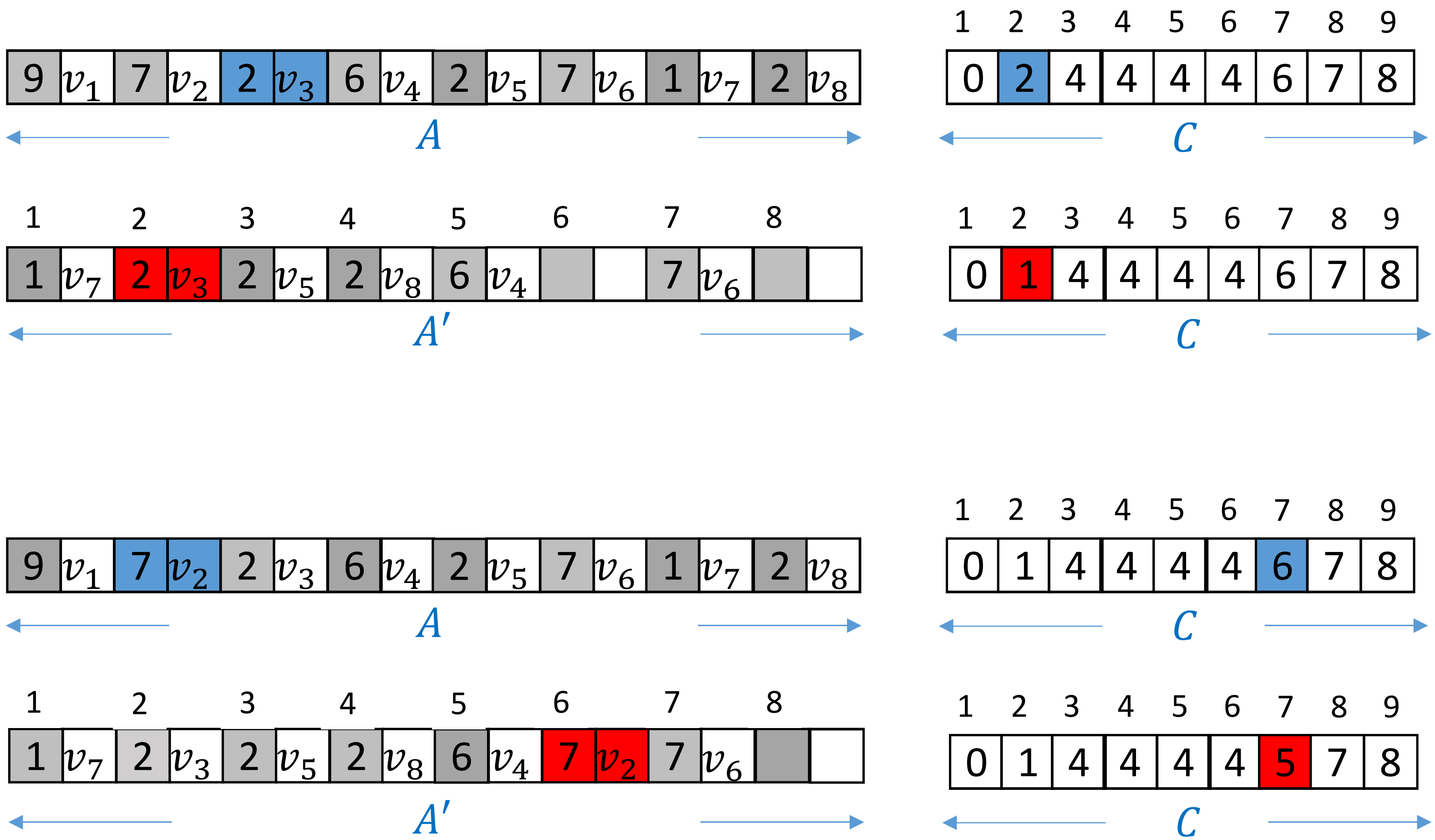
A'

1		2		3		4		5		6		7		8	
1	v_7			2	v_5	2	v_8	6	v_4			7	v_6		

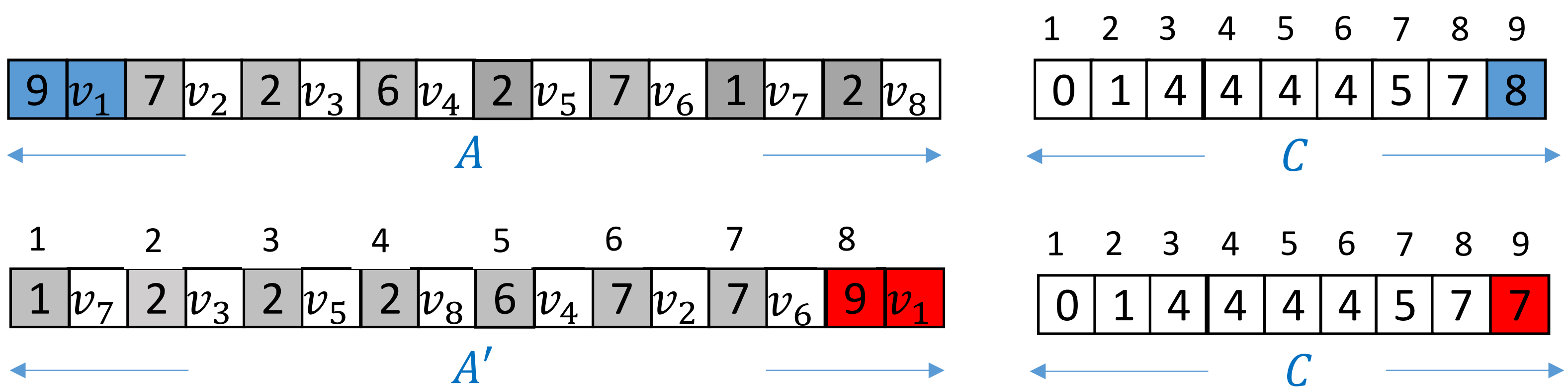
C

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

Counting Sort (2nd Ver.)



Counting Sort (2nd Ver.)



Overall time complexity: $O(n + U)$