Data Structures and Algorithms

Tutorial 6. Counting, radix, and bucket sorting

Today’s topic is covered in detail in

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to Algorithms.** The MIT Press 2009.

II Sorting and Order Statistics

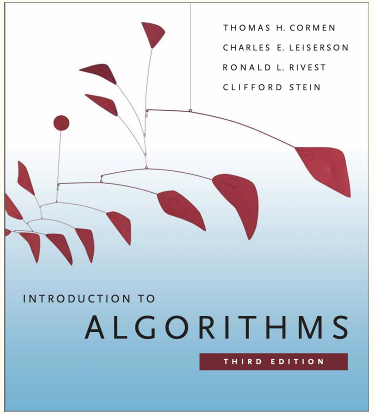
Introduction *147*

6 Heapsort ***151***
- 6.1 Heaps *151*
- 6.2 Maintaining the heap property *154*
- 6.3 Building a heap *156*
- 6.4 The heapsort algorithm *159*
- 6.5 Priority queues *162*

7 Quicksort ***170***
- 7.1 Description of quicksort *170*
- 7.2 Performance of quicksort *174*
- 7.3 A randomized version of quicksort *179*
- 7.4 Analysis of quicksort *180*

8 Sorting in Linear Time ***191***
- 8.1 Lower bounds for sorting *191*
- 8.2 Counting sort *194*
- 8.3 Radix sort *197*
- 8.4 Bucket sort *200*

9 Medians and Order Statistics ***213***
- 9.1 Minimum and maximum *214*
- 9.2 Selection in expected linear time *215*
- 9.3 Selection in worst-case linear time *220*



Objectives

- Counting sort
- Radix sort
- Bucket sort

Counting sort: naive approach

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

We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

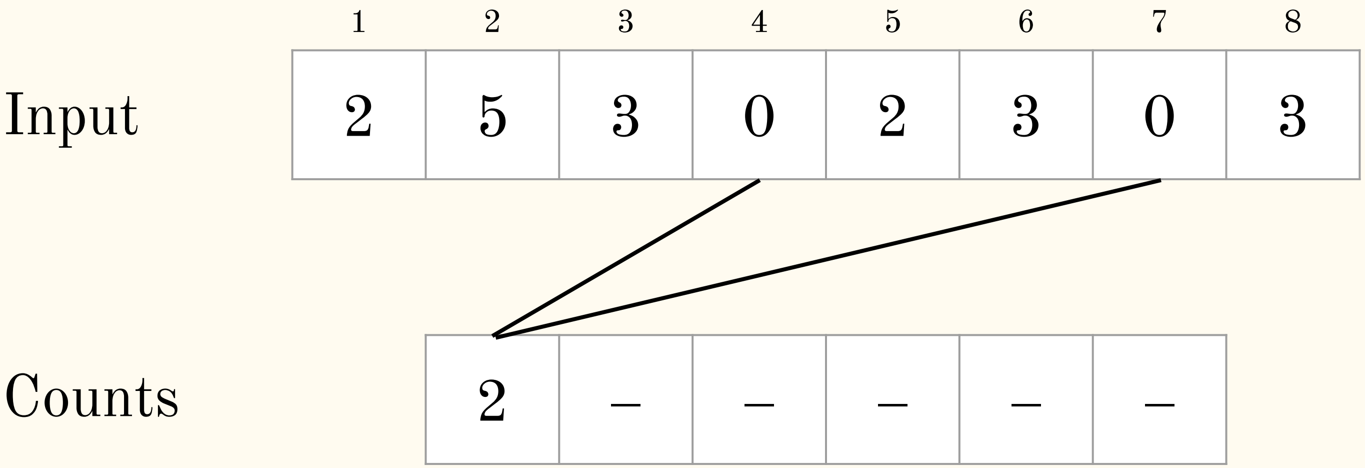
Counting sort: naive approach

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

Counts	–	–	–	–	–	–

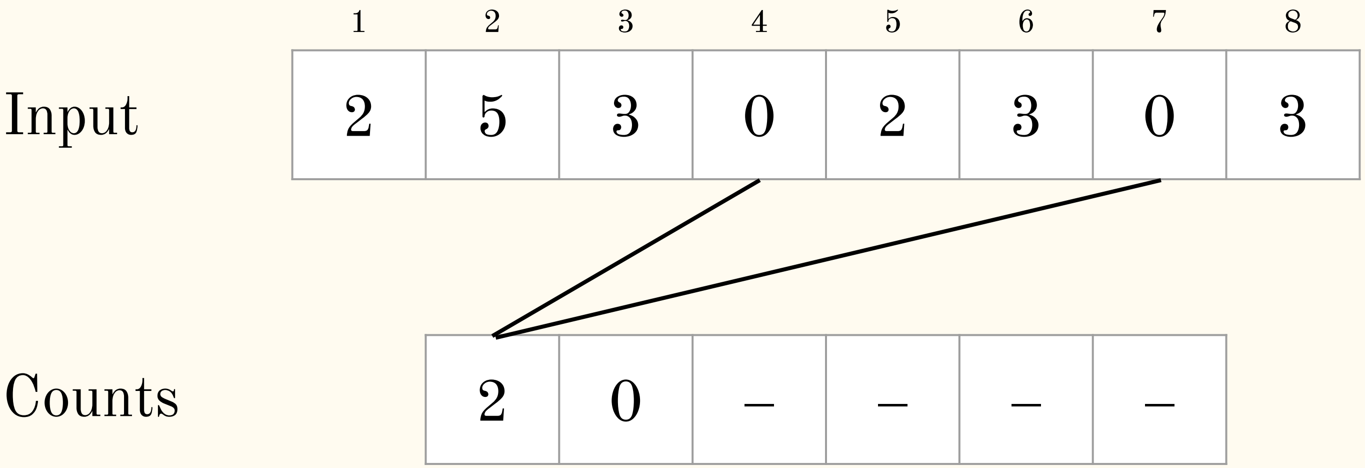
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach



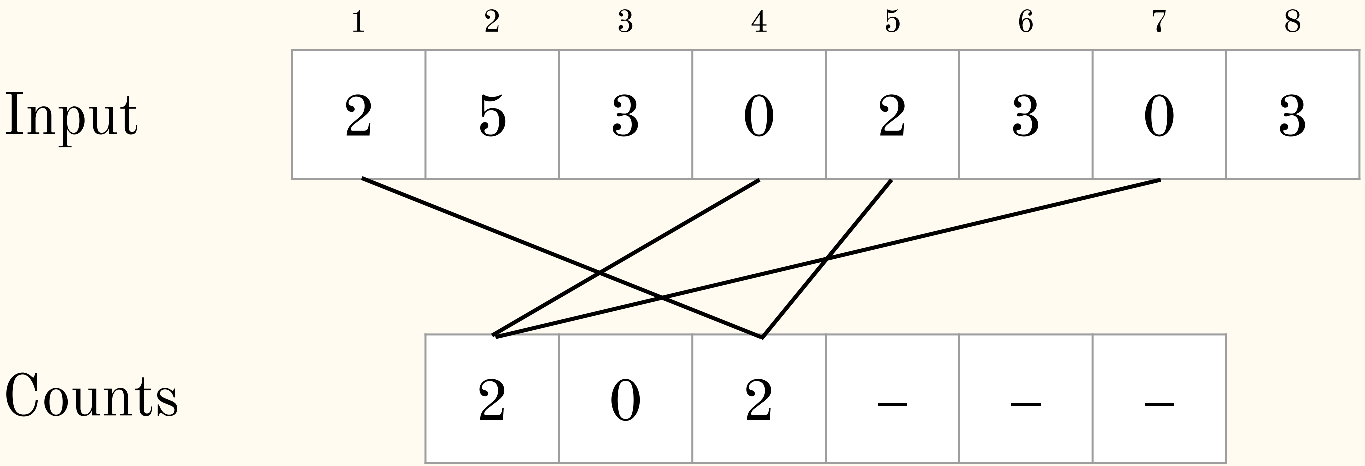
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach



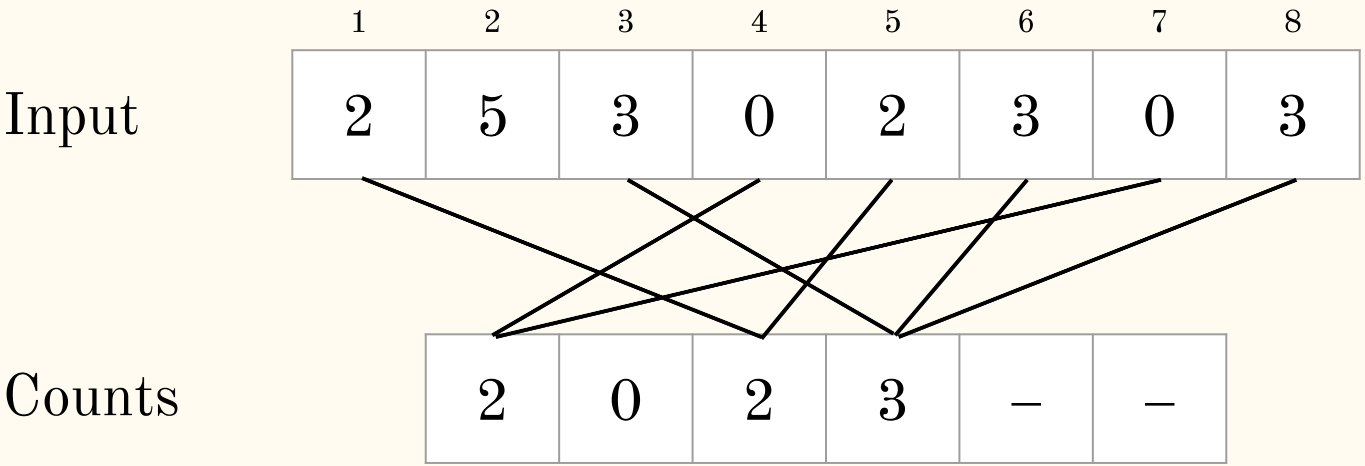
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach



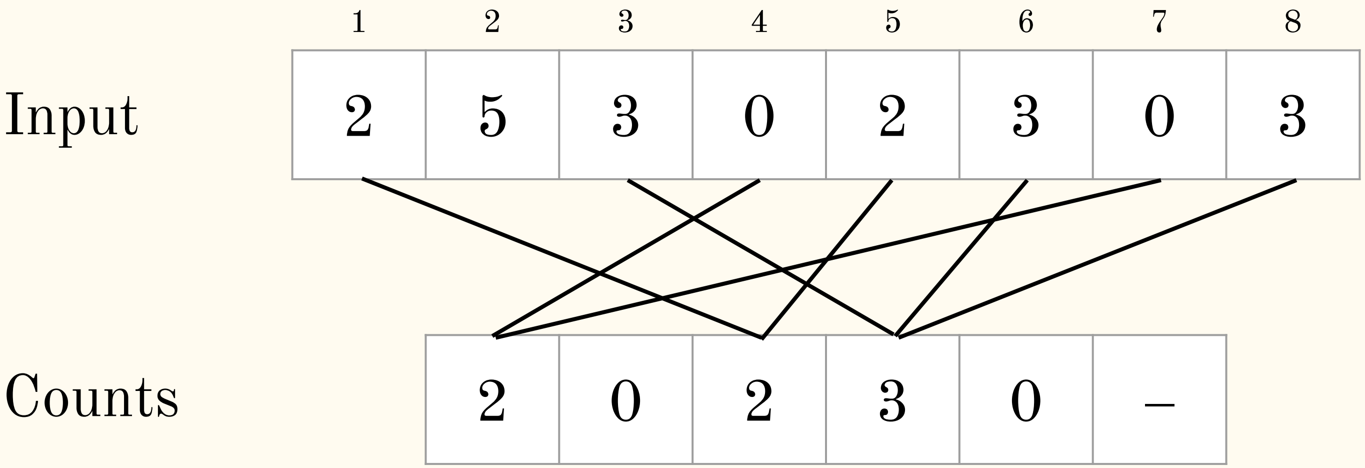
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach



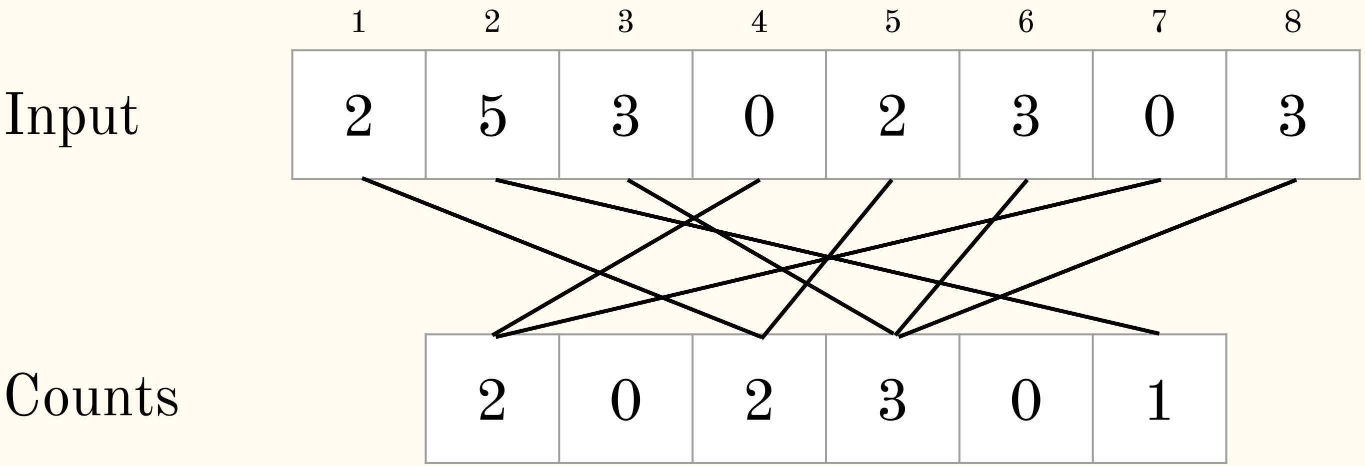
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach



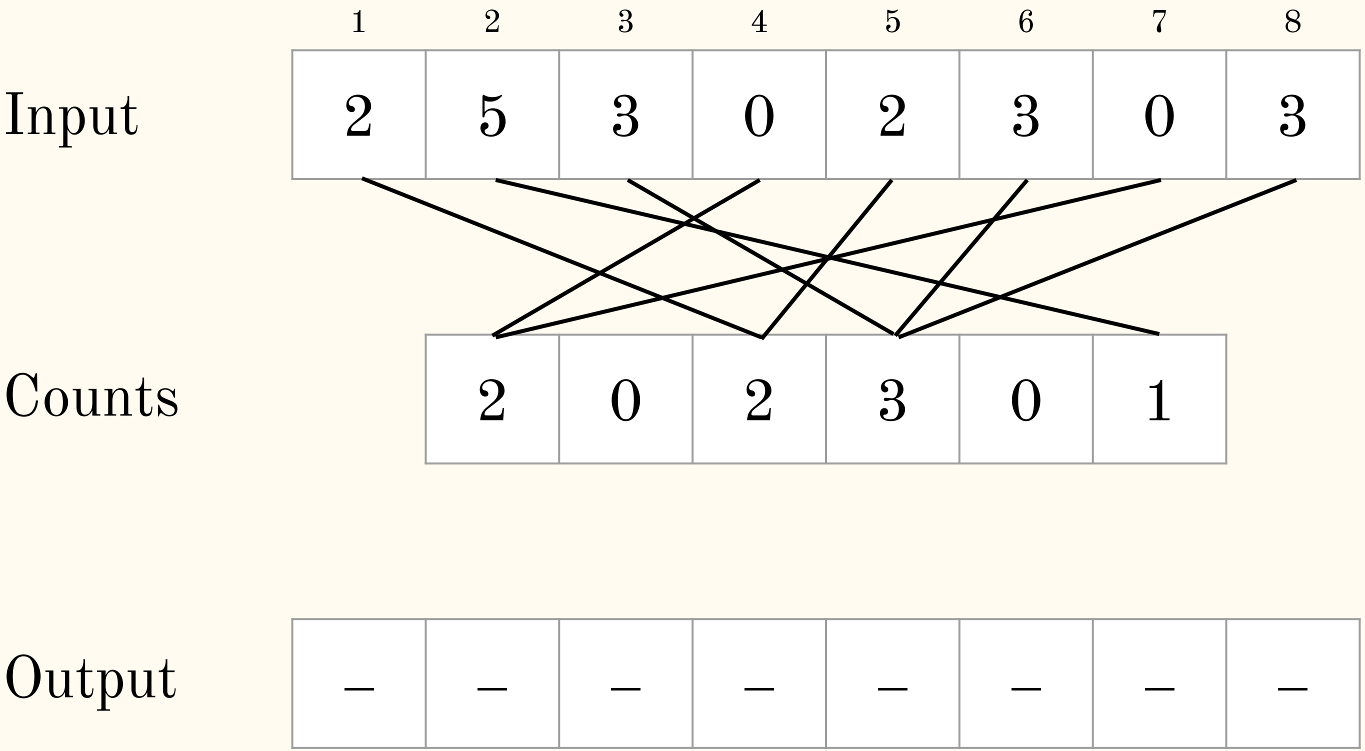
We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

Counting sort: naive approach

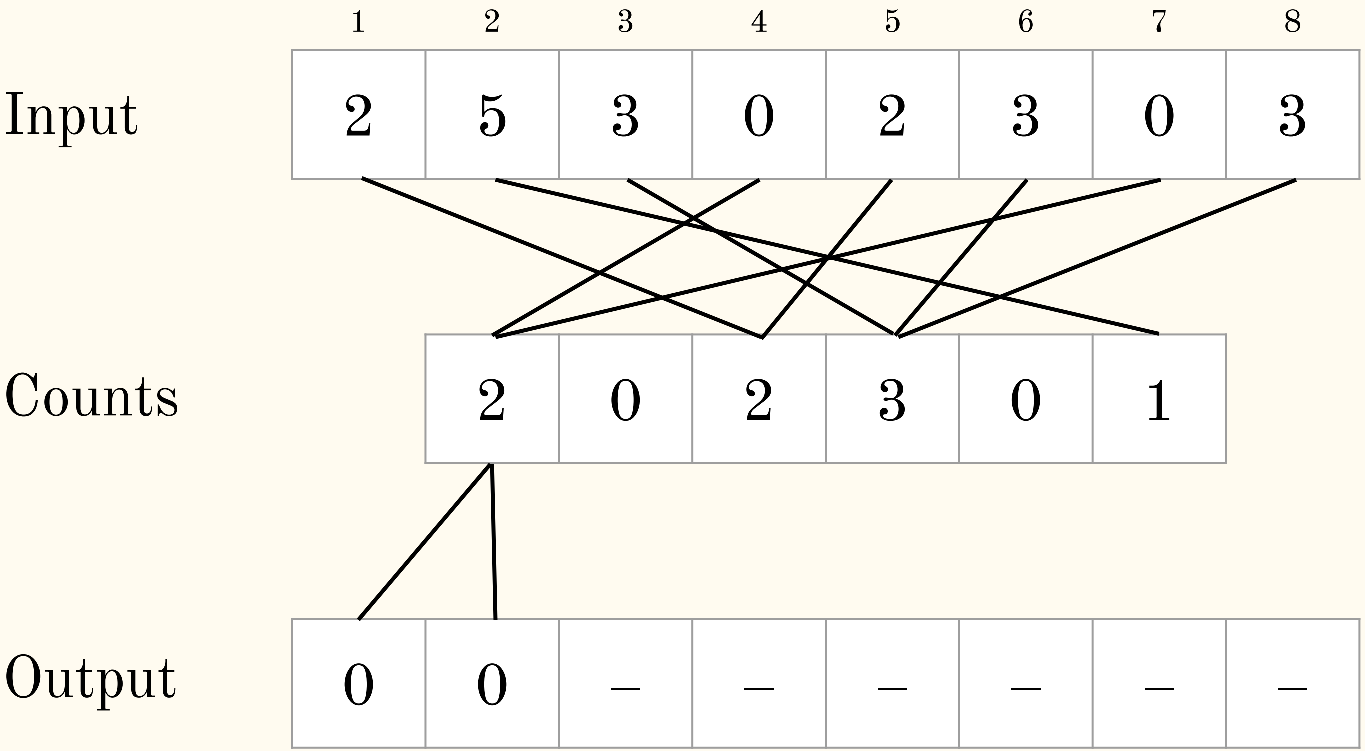


We want to sort an array of integers.
All values are in range from 0 to 5.
Idea: count how many 0s, 1s, 2s, 3s, 4s, and 5s we have!

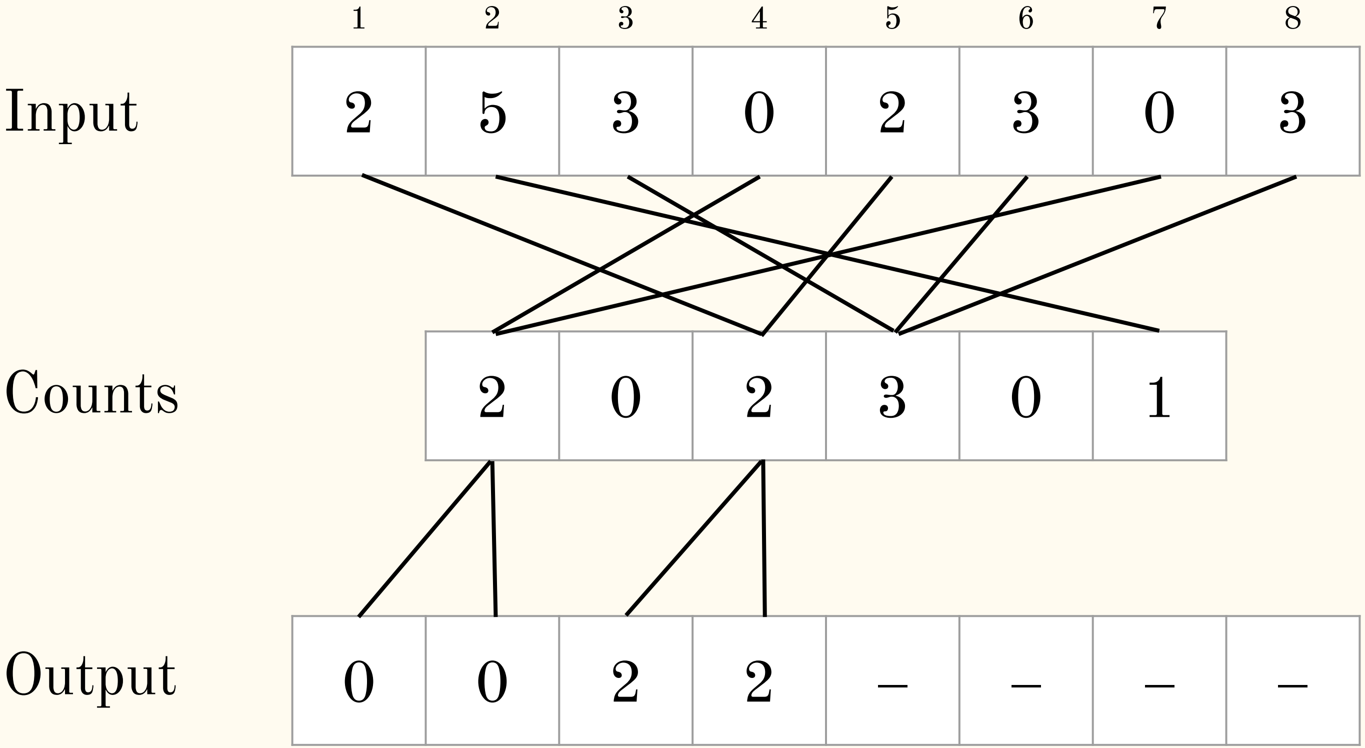
Counting sort: naive approach



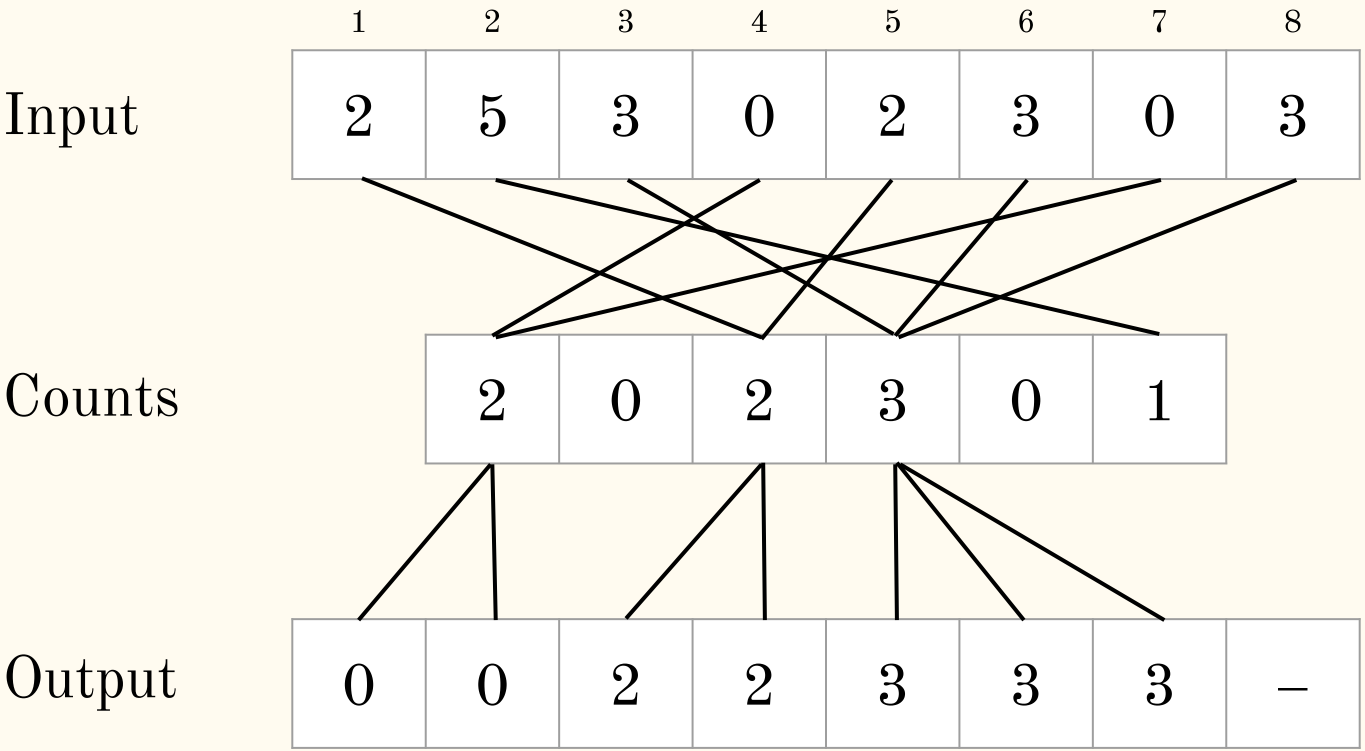
Counting sort: naive approach



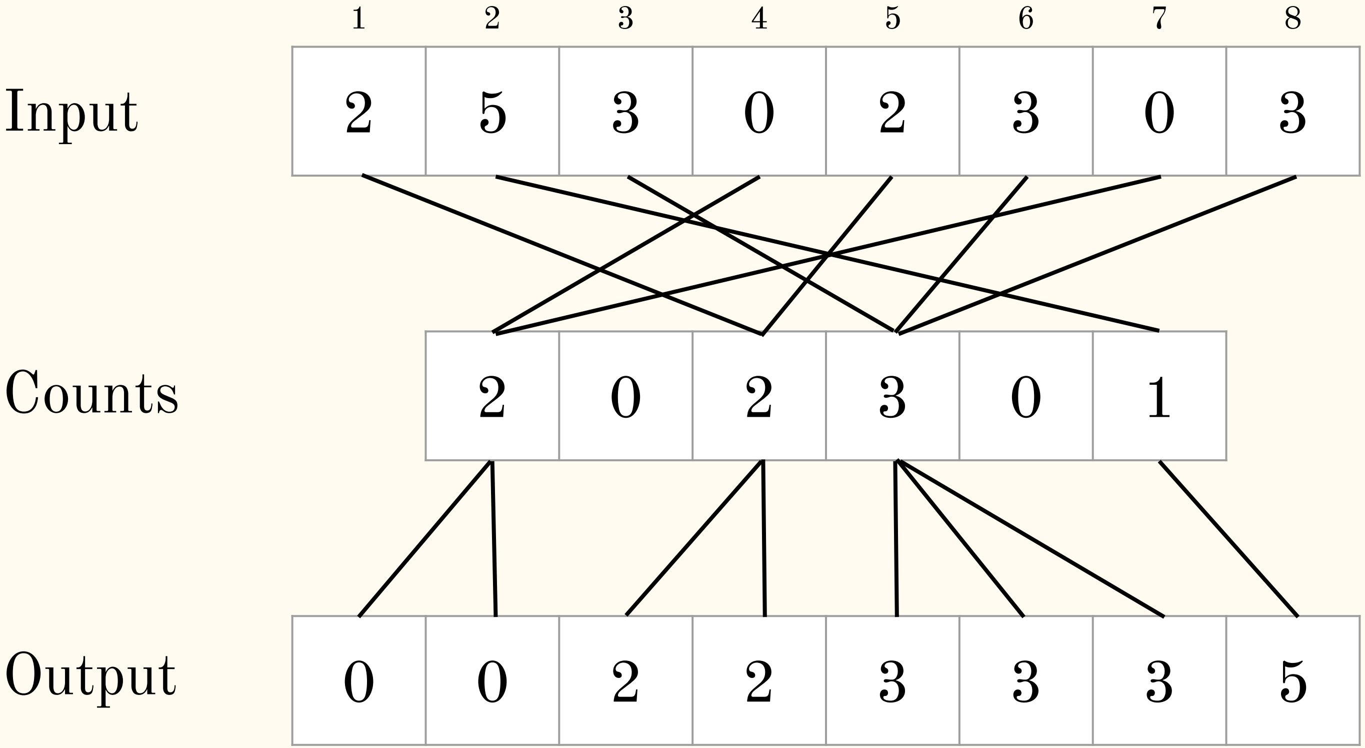
Counting sort: naive approach



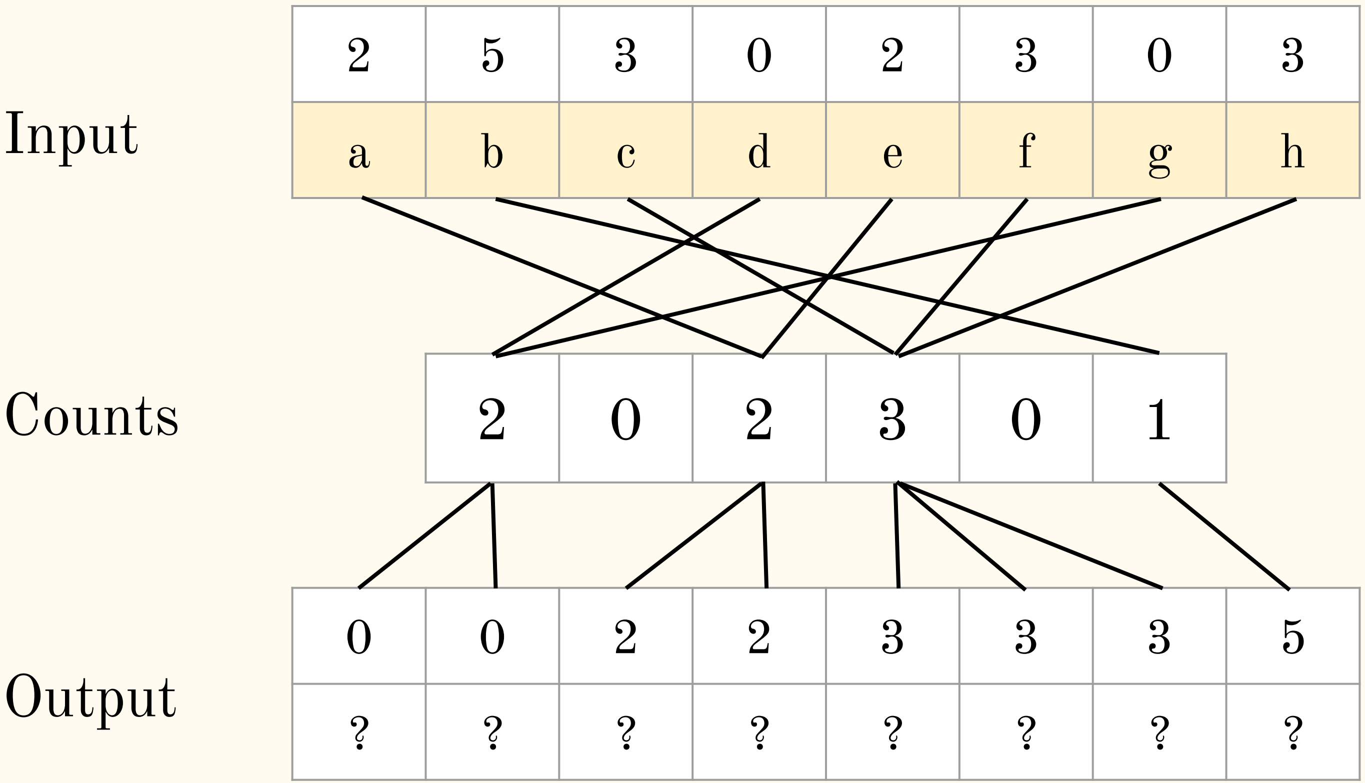
Counting sort: naive approach



Counting sort: naive approach



Counting sort: what about satellite data?



Counting sort: adjusted idea

Input

2	5	3	0	2	3	0	3
a	b	c	d	e	f	g	h

Count

2	0	2	3	0	1

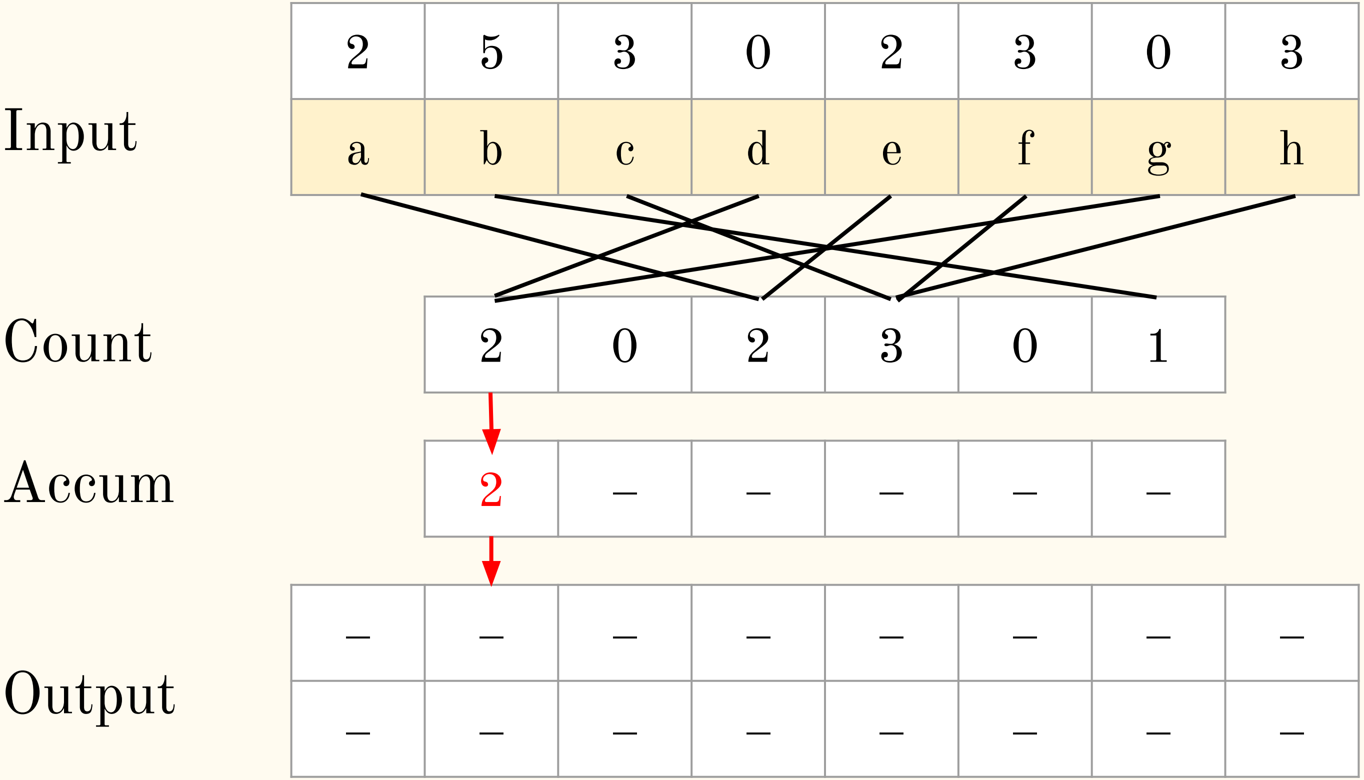
Accum

–	–	–	–	–	–

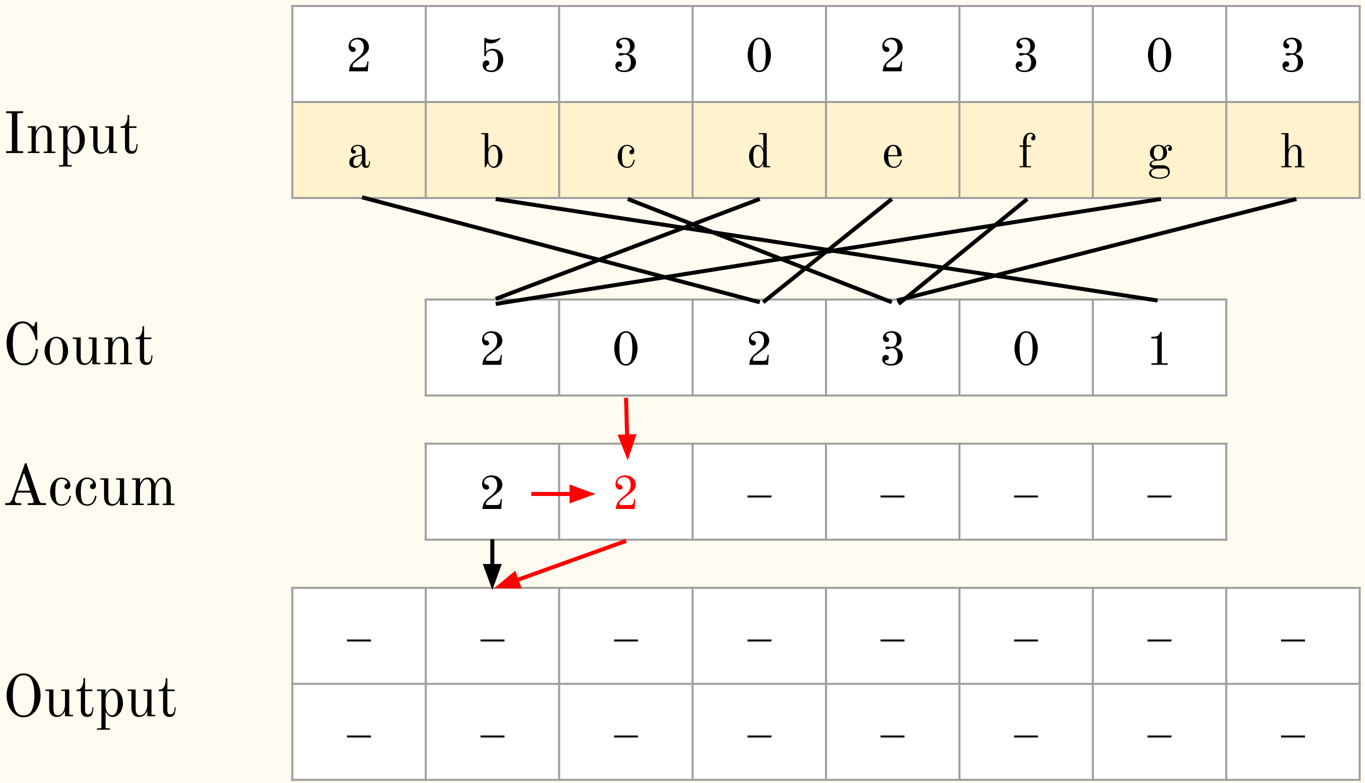
Output

–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–

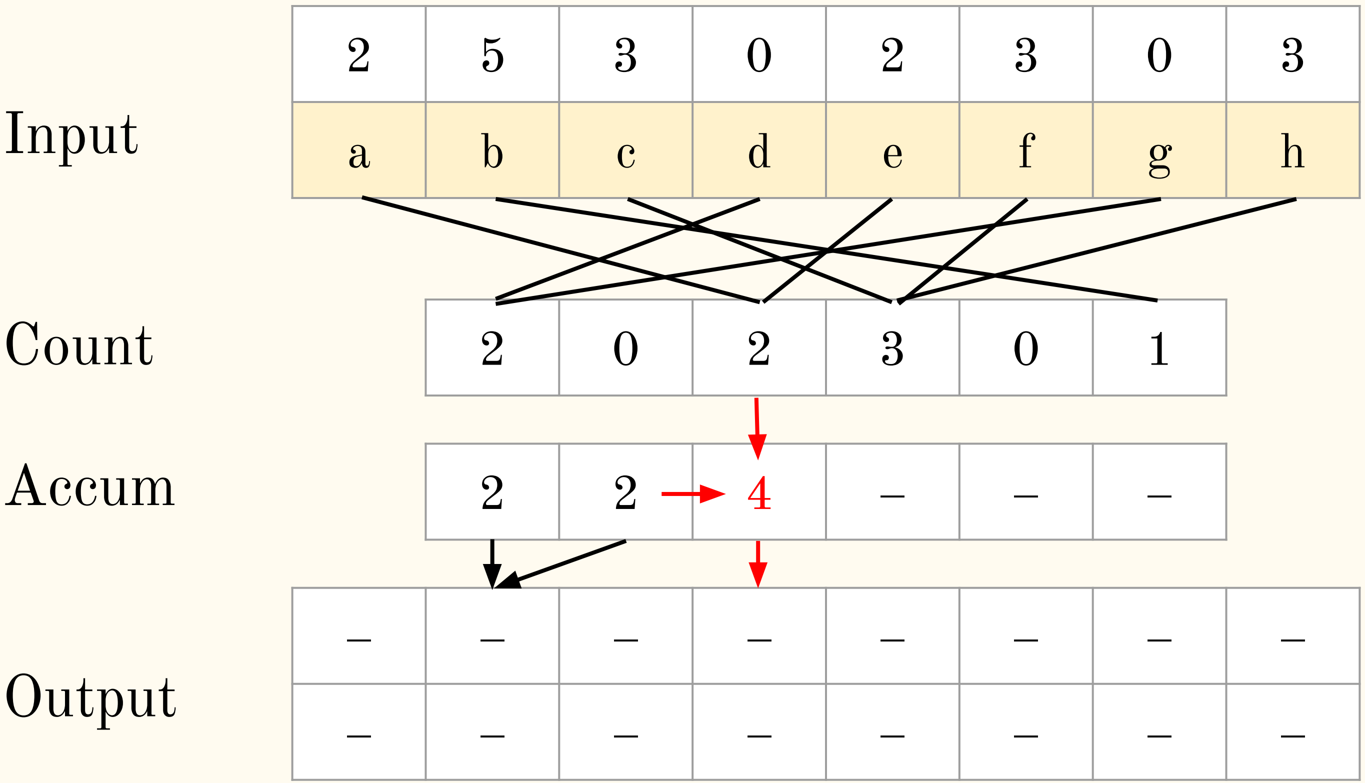
Counting sort: adjusted idea



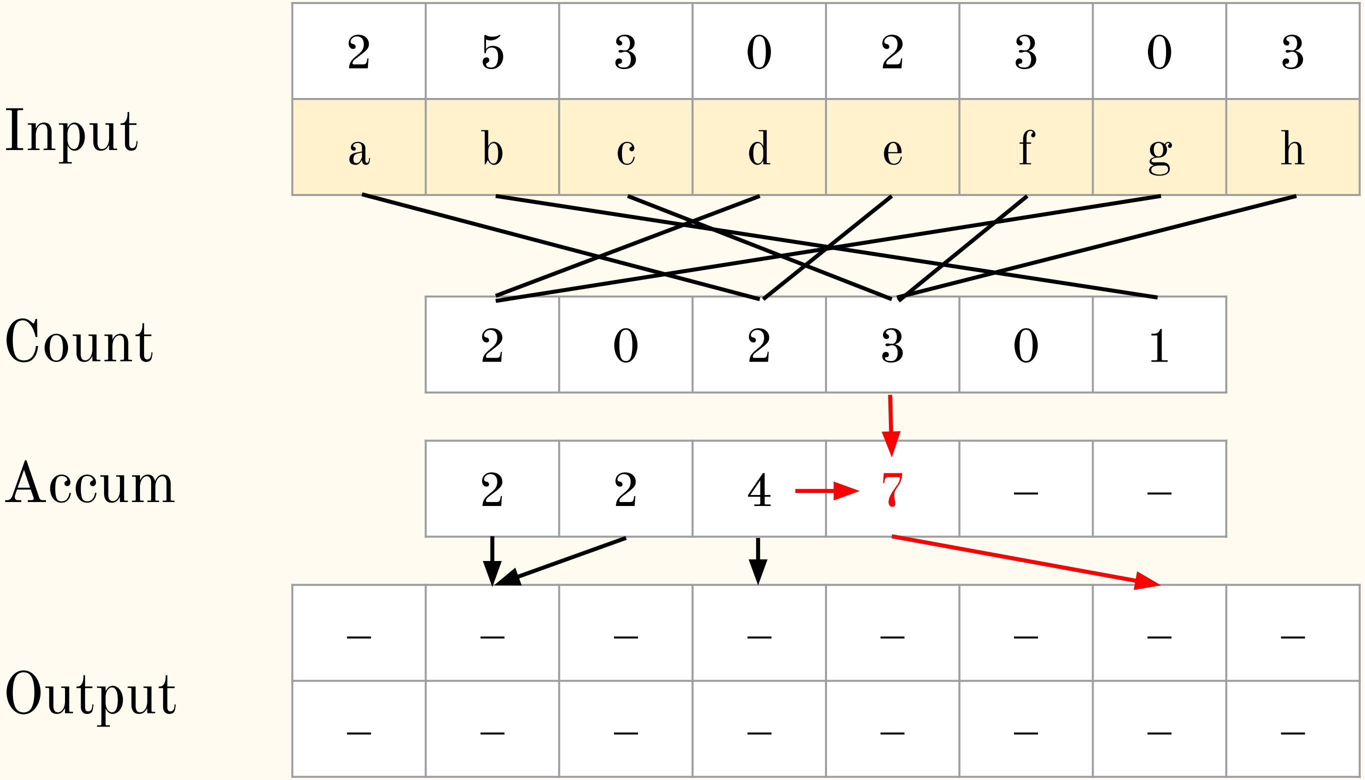
Counting sort: adjusted idea



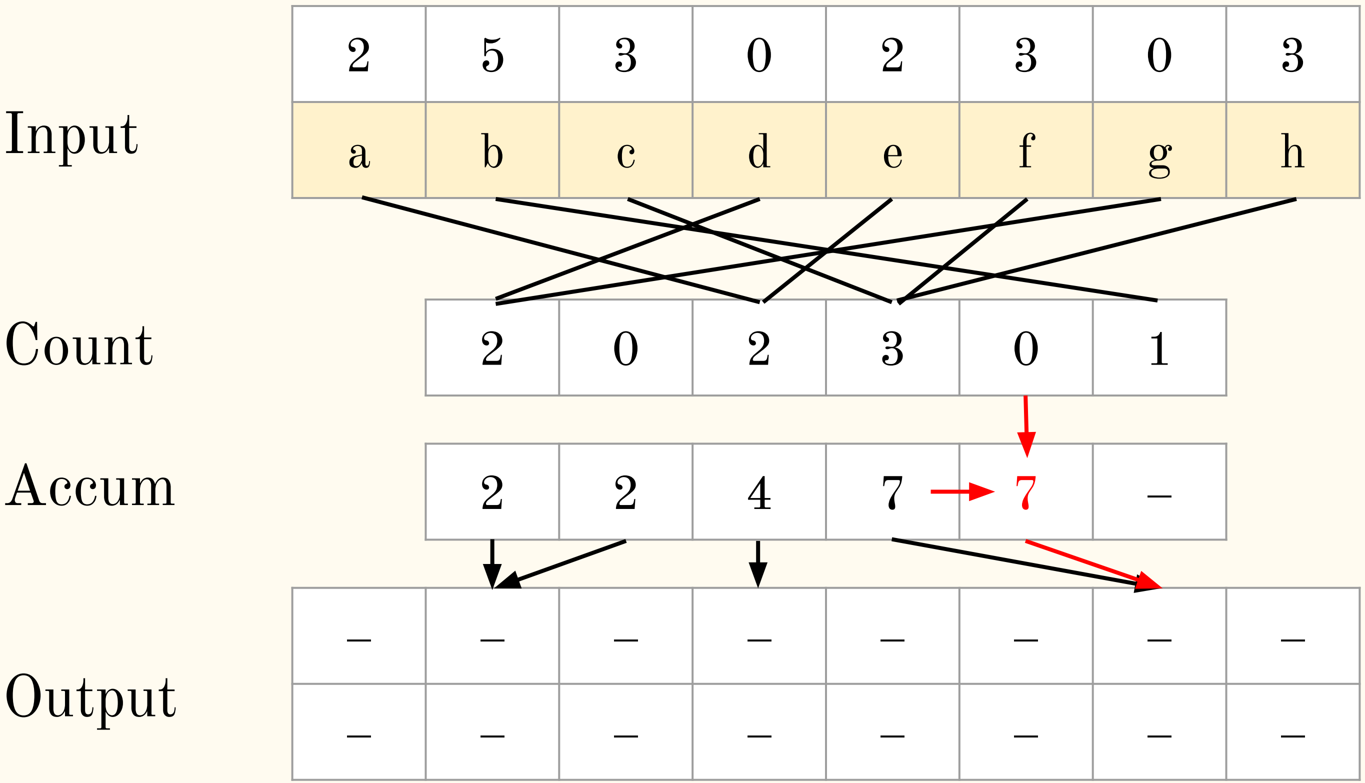
Counting sort: adjusted idea



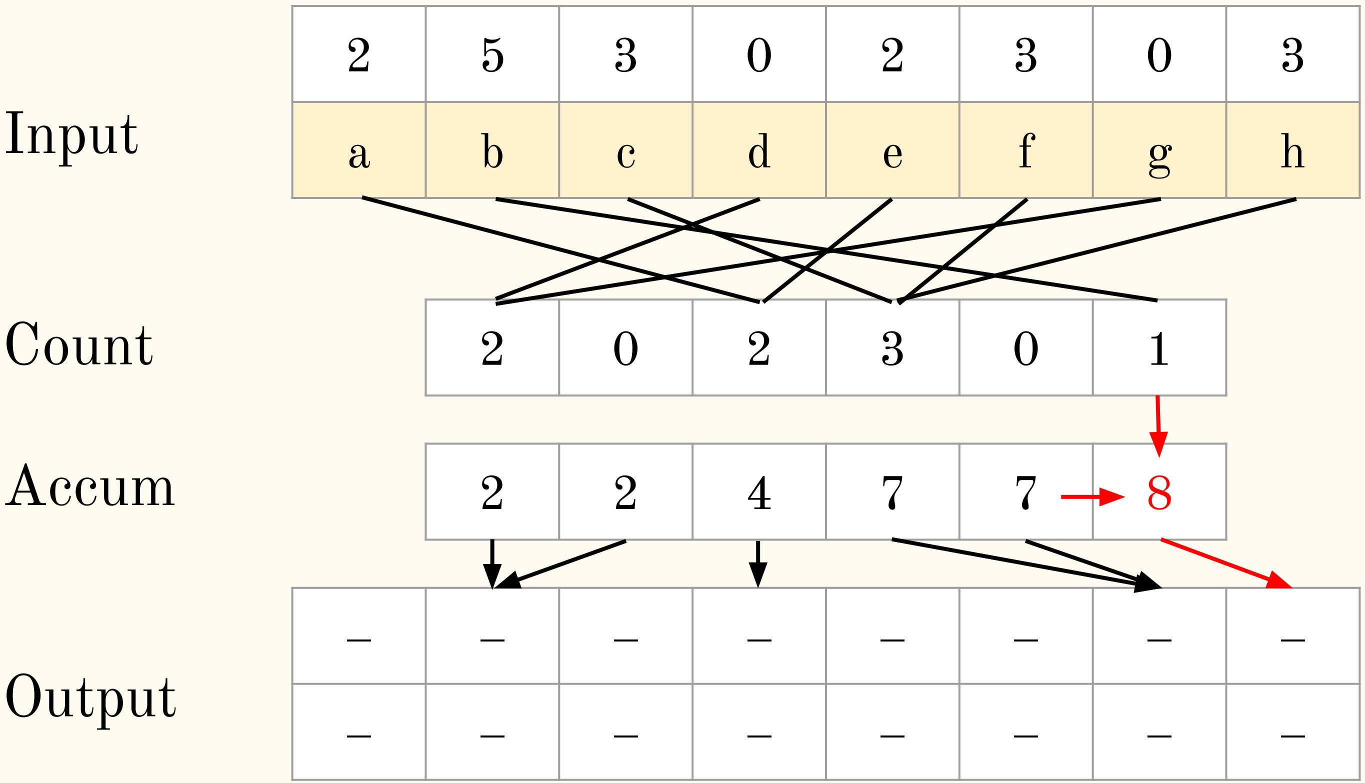
Counting sort: adjusted idea



Counting sort: adjusted idea



Counting sort: adjusted idea



Counting sort: adjusted idea

Input

2	5	3	0	2	3	0	3
a	b	c	d	e	f	g	h

Count

2	0	2	3	0	1
0	1	2	3	4	5

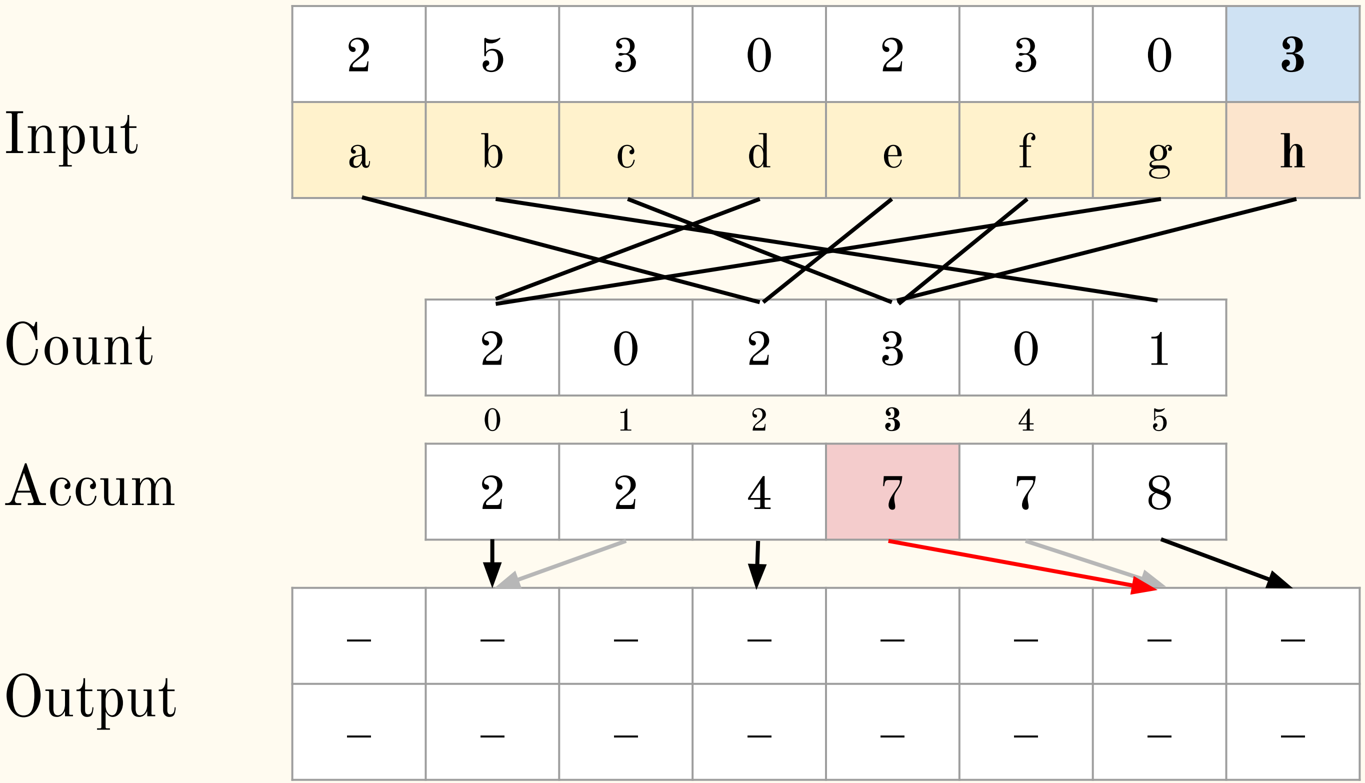
Accum

2	2	4	7	7	8

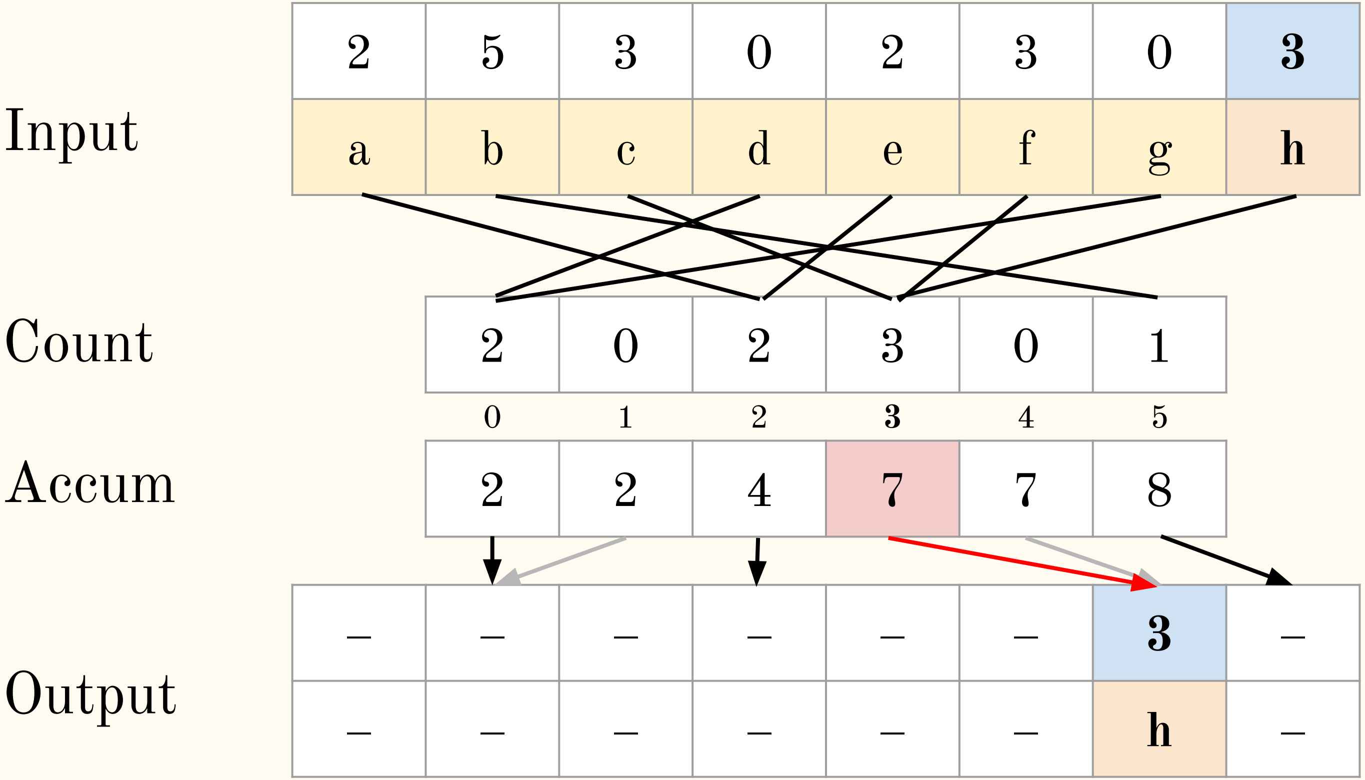
Output

–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–

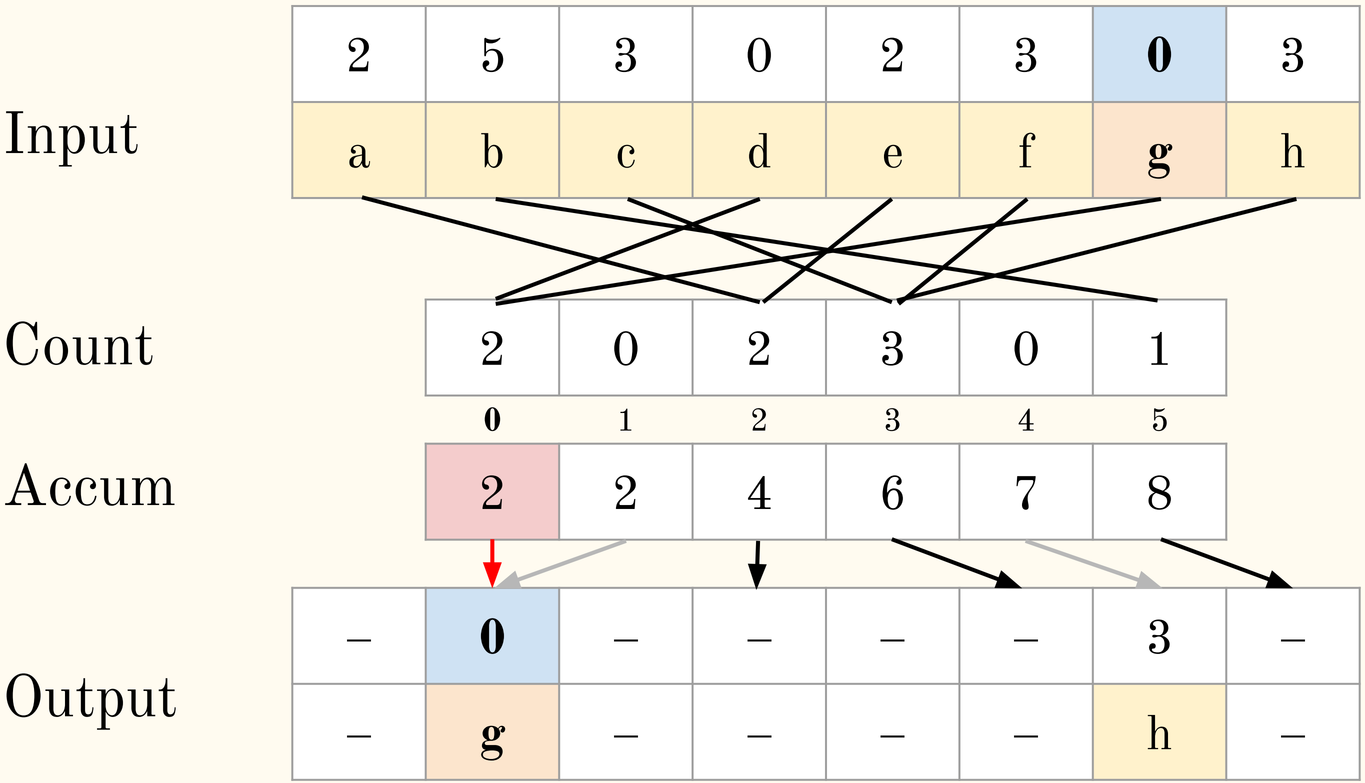
Counting sort: adjusted idea



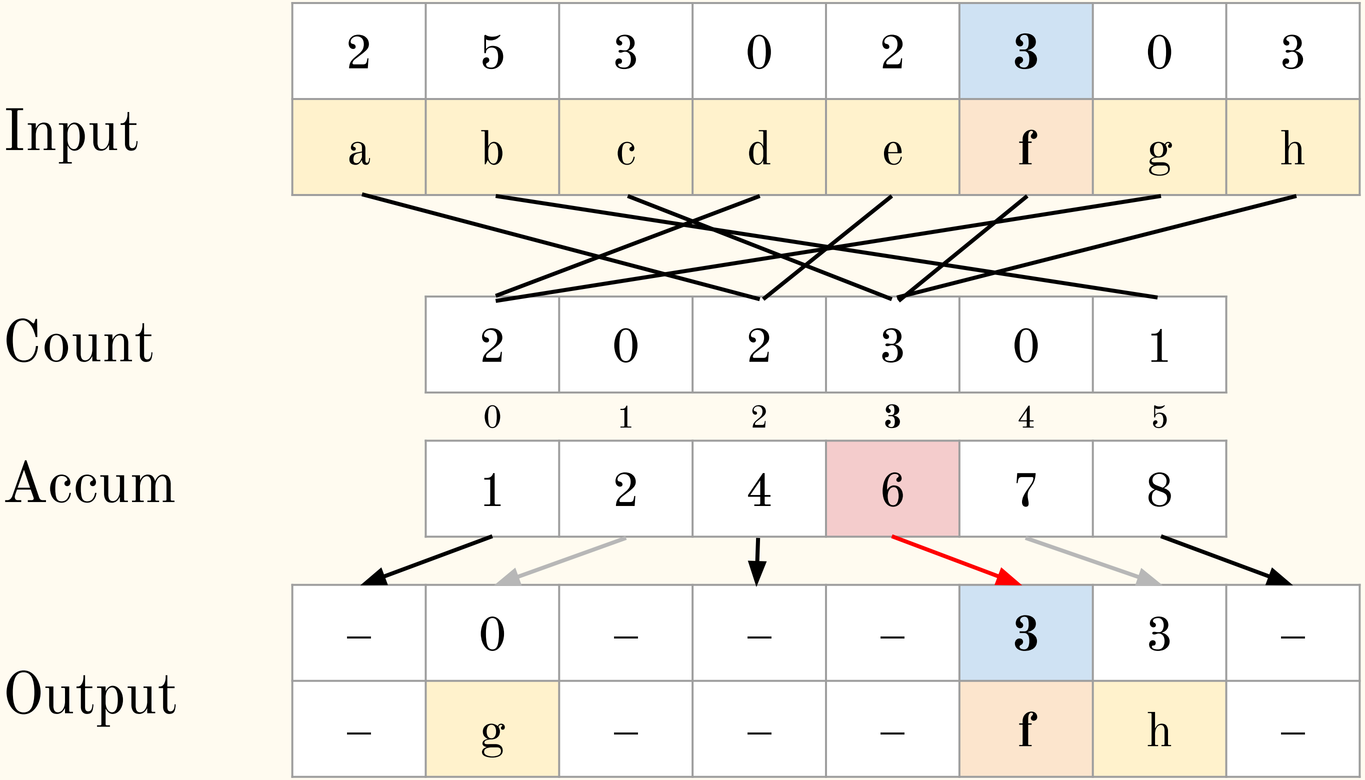
Counting sort: adjusted idea



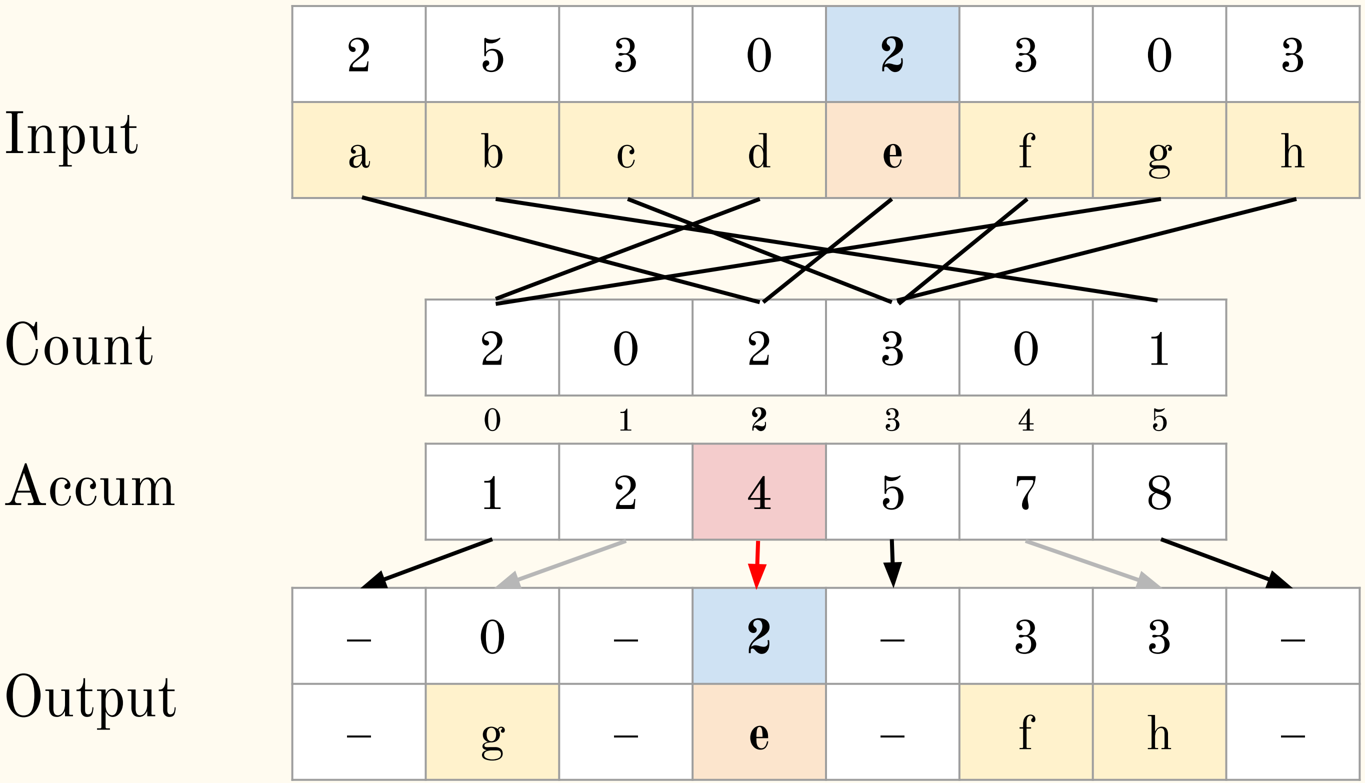
Counting sort: adjusted idea



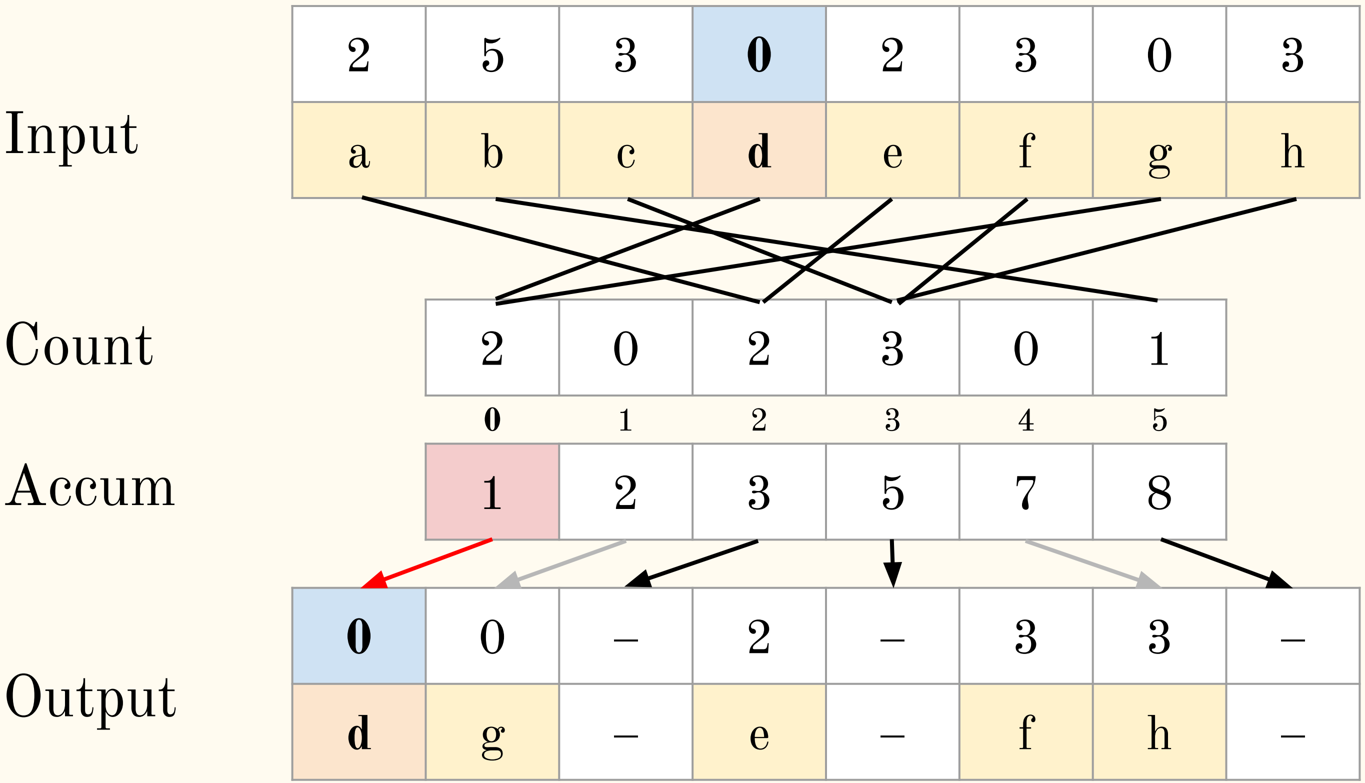
Counting sort: adjusted idea



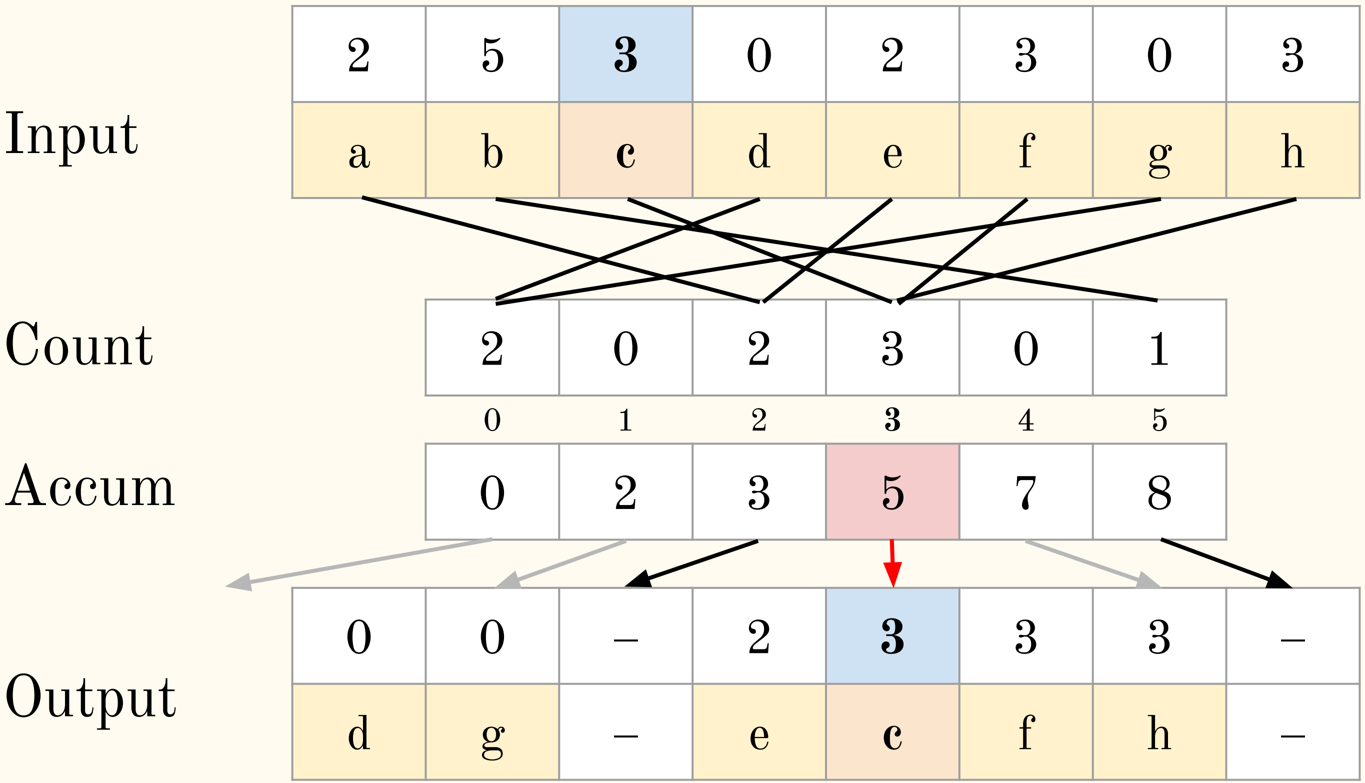
Counting sort: adjusted idea



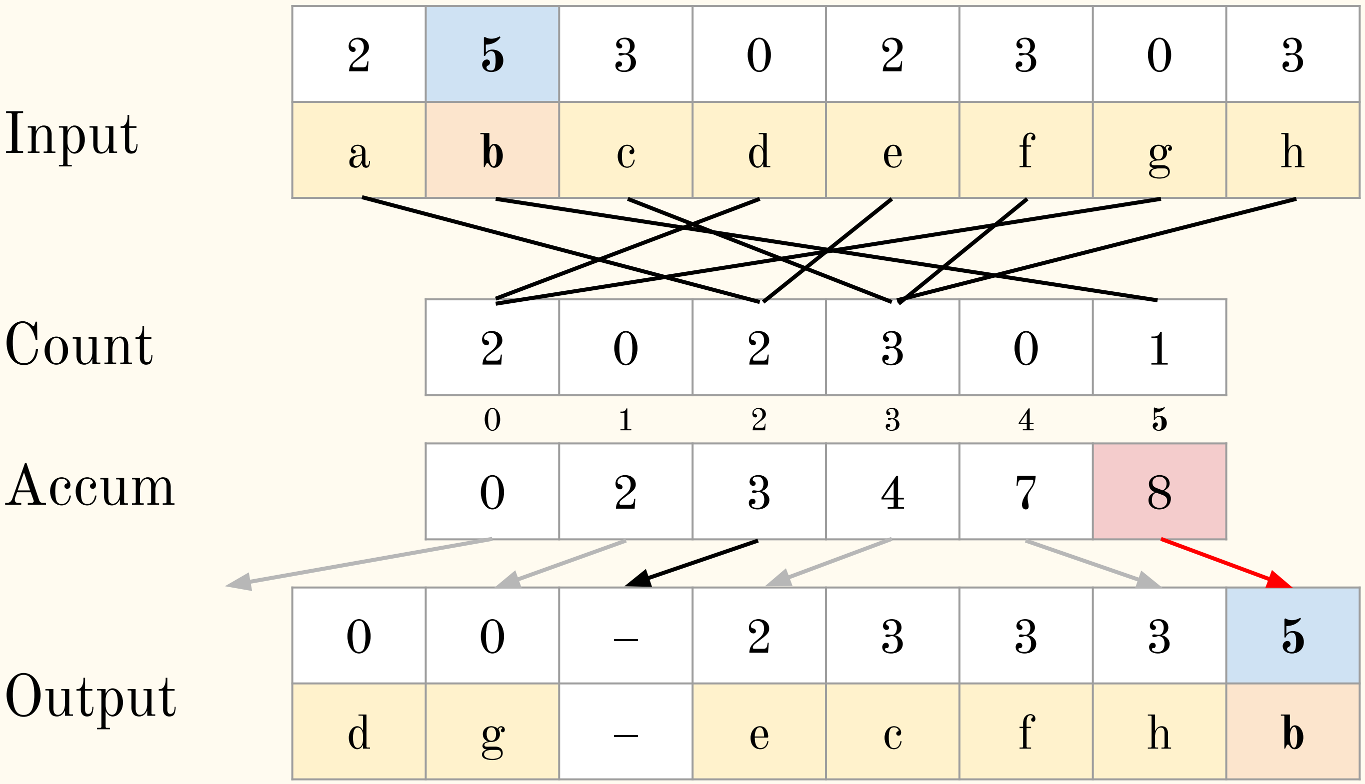
Counting sort: adjusted idea



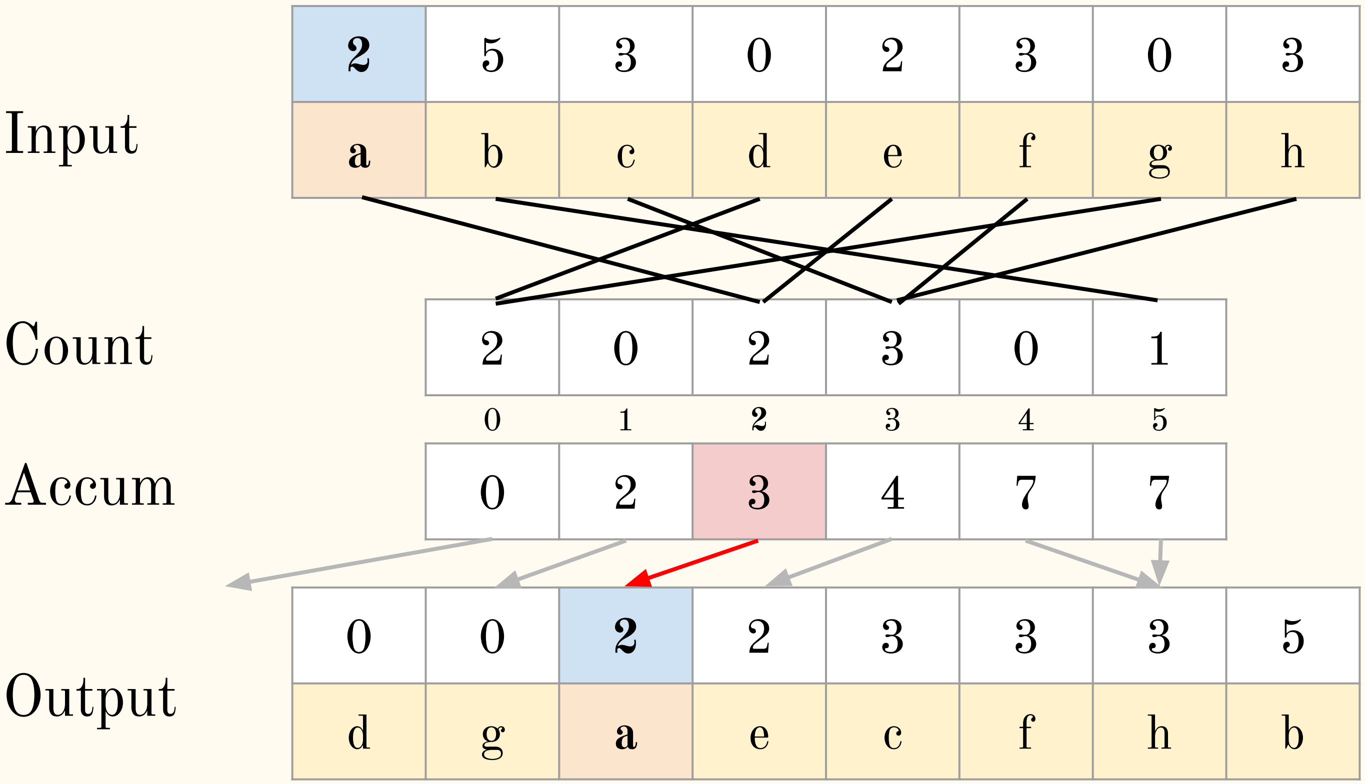
Counting sort: adjusted idea



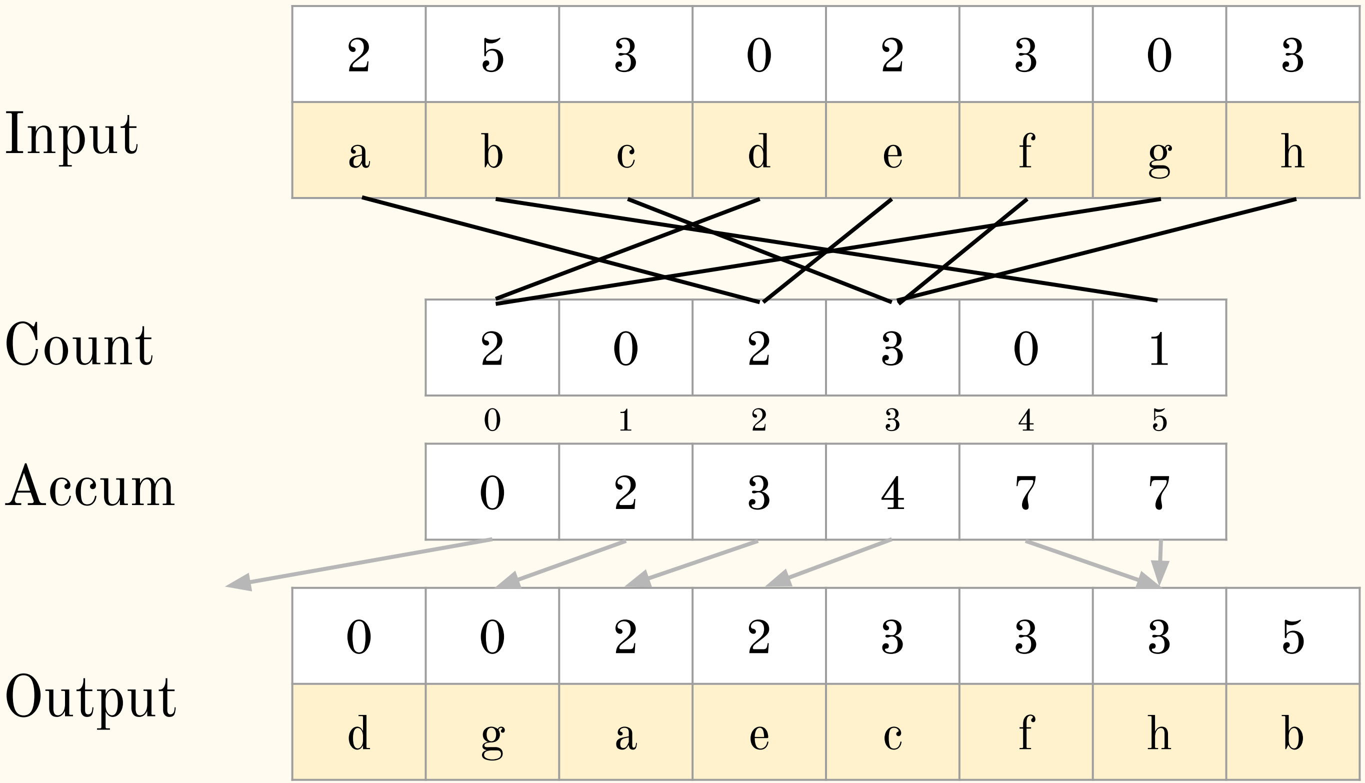
Counting sort: adjusted idea



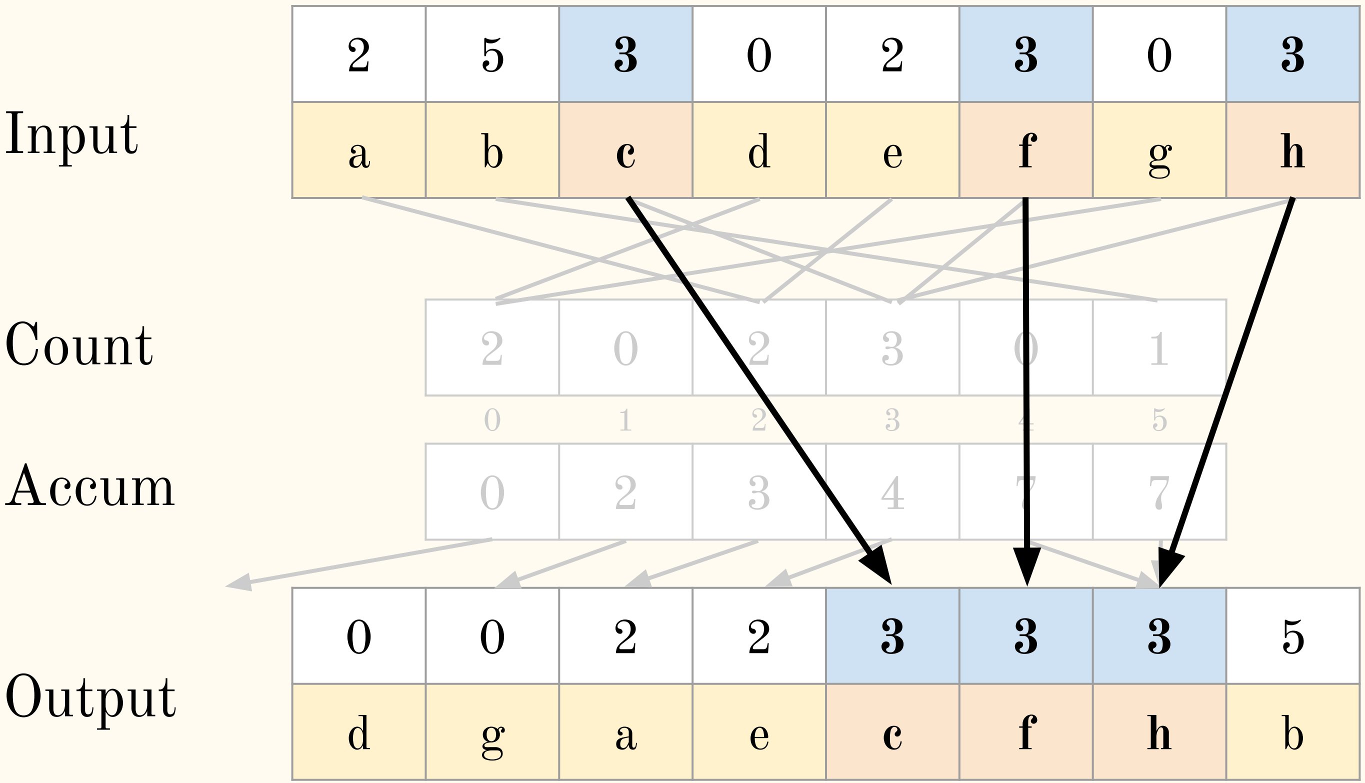
Counting sort: adjusted idea



Counting sort: adjusted idea



Counting sort is **stable**



Counting sort: algorithm

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Counting sort: algorithm

```
COUNTING-SORT(A, B, k)
    let C[0..k] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to i.
    for i = 1 to k
        C[i] = C[i] + C[i - 1]
    // C[i] now contains the number of elements less than or equal to i.
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Count elements

Counting sort: algorithm

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Count elements

Accumulate

Counting sort: algorithm

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Count elements

Accumulate

Populate output array

Counting sort: time complexity

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
        C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

O(k)

O(n)

O(k)

O(n)

O(n+k)

Counting sort: exercise

Exercise 6.1. Suppose that we were to rewrite the **for** loop in line 10 of the COUNTING- SORT as

`for j = 1 to A.length`

1. Is the algorithm still correct?
2. If yes, it is stable?

```
COUNTING-SORT(A, B, k)
1   let C[0..k] be a new array
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A.length
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number
7   for i = 1 to k
8       C[i] = C[i] + C[i - 1]
9   // C[i] now contains the number
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

Counting sort: exercise

Exercise 6.2. Describe an algorithm that, given **n** integers in the range from 0 to k, preprocesses its input and then answers **any** query about how many of the n integers fall into a range (a, b) in O(1) time. Your algorithm should use O(n + k) preprocessing time.

Radix sort: idea

Assumptions:

1. Input is a sequence of “numbers” (strings of digits)
2. Each number has d digits
3. Each digit can range from 0 to k

Radix sort: idea

Assumptions:

1. Input is a sequence of “numbers” (strings of digits)
2. Each number has d digits
3. Each digit can range from 0 to k

Radix sort idea:

1. Sort numbers by least significant digit first using a **stable** sort
2. Then sort again, by second least significant digit using a **stable** sort
3. ...
4. Sort by most significant digit using a **stable** sort
5. We have a sorted sequence of numbers!

Radix sort: example

329	457	657	839	436	720	355

Radix sort: example

329	457	657	839	436	720	355

32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**

Radix sort: example

329	457	657	839	436	720	355

32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**

72**0**	35**5**	43**6**	45**7**	65**7**	32**9**	83**9**

Radix sort: example

329	457	657	839	436	720	355

32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**

720	355	436	457	657	329	839

Radix sort: example

329	457	657	839	436	720	355

32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**

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

Radix sort: example

329	457	657	839	436	720	355
32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**
7**2**0	3**5**5	4**3**6	4**5**7	6**5**7	3**2**9	8**3**9
7**2**0	3**2**9	4**3**6	8**3**9	3**5**5	4**5**7	6**5**7

Radix sort: example

329	457	657	839	436	720	355
32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**
7**2**0	3**5**5	4**3**6	4**5**7	6**5**7	3**2**9	8**3**9
720	329	436	839	355	457	657

Radix sort: example

329	457	657	839	436	720	355
32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**
7**2**0	3**5**5	4**3**6	4**5**7	6**5**7	3**2**9	8**3**9
720	**3**29	**4**36	**8**39	**3**55	**4**57	**6**57

Radix sort: example

329	457	657	839	436	720	355
32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**
7**2**0	3**5**5	4**3**6	4**5**7	6**5**7	3**2**9	8**3**9
720	**3**29	**4**36	**8**39	**3**55	**4**57	**6**57
329	**3**55	**4**36	**4**57	**6**57	**7**20	**8**39

Radix sort: example

329	457	657	839	436	720	355
32**9**	35**5**	43**6**	45**7**	65**7**	72**0**	83**9**
7**2**0	3**5**5	4**3**6	4**5**7	6**5**7	3**2**9	8**3**9
720	**3**29	**4**36	**8**39	**3**55	**4**57	**6**57
329	355	436	457	657	720	839

Radix sort: time complexity

```
RADIX-SORT(A, d)
for i = 1 to d
        use a stable sort to sort array A on digit i
```

Radix sort: time complexity

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Assuming that sorting on digit **i** takes $\Theta(n+k)$, time complexity of radix sort is $\Theta(d(n+k))$.

Radix sort: exercise

Exercise **6.3.**

Sort n integers in the range from 0 to (n^3-1) in $O(n)$ time.

Attendance

https://baam.duckdns.org

Bucket sort: idea

Assumptions:

1. Input comes from a uniform distribution (e.g. over a real interval).
2. Input values can be easily truncated to discrete values (e.g. via floor/ceiling).
3. Truncated values fit in a small range from 0 to k.

Bucket sort: idea

Assumptions:

1. Input comes from a uniform distribution (e.g. over a real interval).
2. Input values can be easily truncated to discrete values (e.g. via floor/ceiling).
3. Truncated values fit in a small range from 0 to k.

Bucket sort idea:

1. Split input values into lists (**buckets**) by their truncated values.
2. Sort each bucket using comparison-based sorting algorithm.
3. Concatenate sorted buckets to produce final sorted result.

Bucket sort: example with real numbers in [0,1)

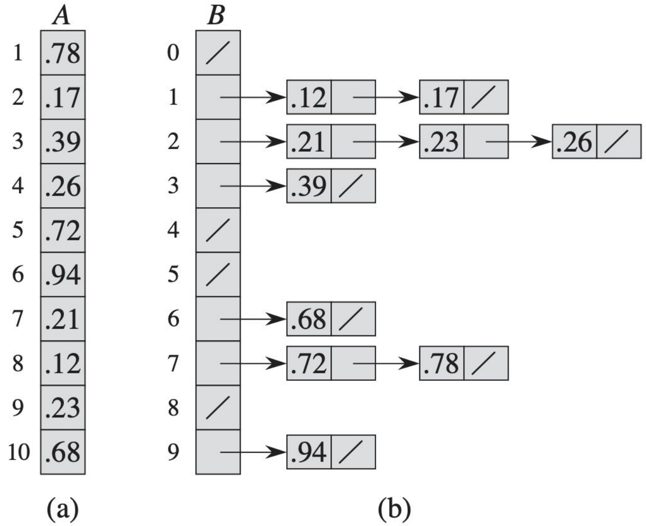


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1\,..\,10]$. **(b)** The array $B[0\,..\,9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket sort: algorithm for real numbers in [0,1)

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

O(n)

O(n)

O(?)

O(n)

Bucket sort: complexity analysis

Let $\mathbf{n_i}$ be a random variable denoting the number of elements placed in bucket $\mathbf{i}$.

Then overall running time of bucket sort is $$\boxed{T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)}$$

We then compute **expected** running time to be (see details in 8.4)

$$\boxed{\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)}$$

Thus, **average** running time of bucket sort is $\Theta(\mathrm{n})$.

Bucket sort: exercise

Exercise 6.4. Explain why the worst-case running time for bucket sort is $O(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

Summary

- Sorting algorithms do not have to rely (only) on comparison, if we know extra information about input

- We can achieve $\Theta(1)$ sorting for many situations:
  - **Counting sort** — for small integers (and enumerations)
  - **Radix sort** — for big integers (and sequences of enumerations)
  - **Bucket sort** — for uniformly distributed "continuous" data

Summary

- Sorting algorithms do not have to rely (only) on comparison, if we know extra information about input
- We can achieve $\Theta(1)$ sorting for many situations:
  - **Counting sort** — for small integers (and enumerations)
  - **Radix sort** — for big integers (and sequences of enumerations)
  - **Bucket sort** — for uniformly distributed "continuous" data

See you next week!