COMP 2402

Sorting

part 2

Alina Shaikhet

Carleton University

Sorting

- Comparison-based Sorting:
  - **MergeSort**
  - **QuickSort**
  - **HeapSort**

$\boldsymbol{O(n \log n)}$ **asymptotically optimal**

worst-case

expected

worst-case

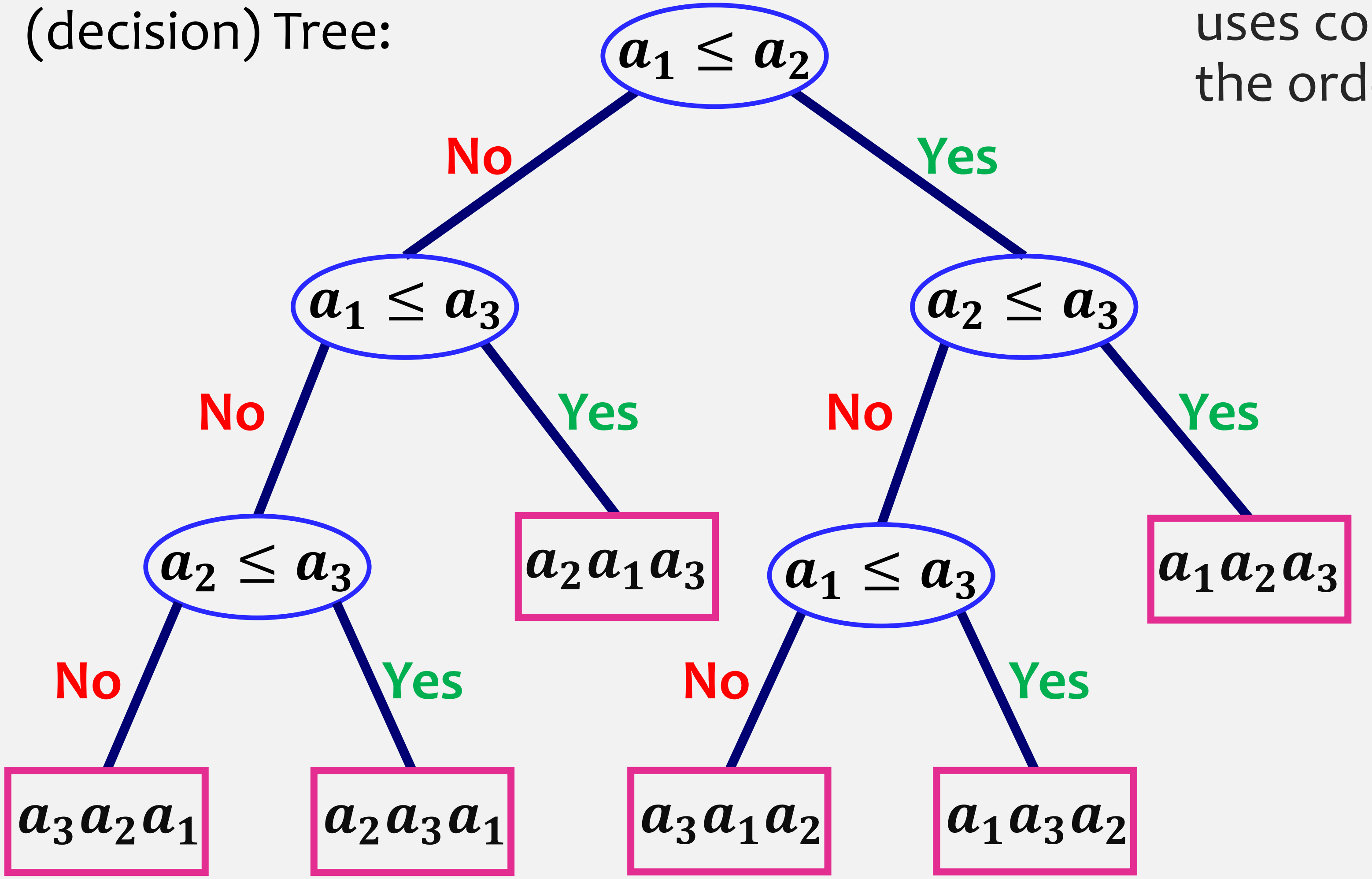
- Comparison-based algorithms can be used to sort any **array** of **comparable** items.
- Is there a faster (maybe $\boldsymbol{O(n)}$ time) sorting algorithm (for general elements)?

No / Yes

- All branching in the comparison-based algorithms is based on the results of comparisons of the form $\boldsymbol{a[i] < a[j]}$
- Every comparison-based sorting algorithm takes $\boldsymbol{\Omega(n \log n)}$ time for some input.

Comparison-based Sorting $\Omega(n \log n)$

Comparison (decision) Tree:



Comparison based sorting algorithm uses comparison operators to find the order between two numbers.

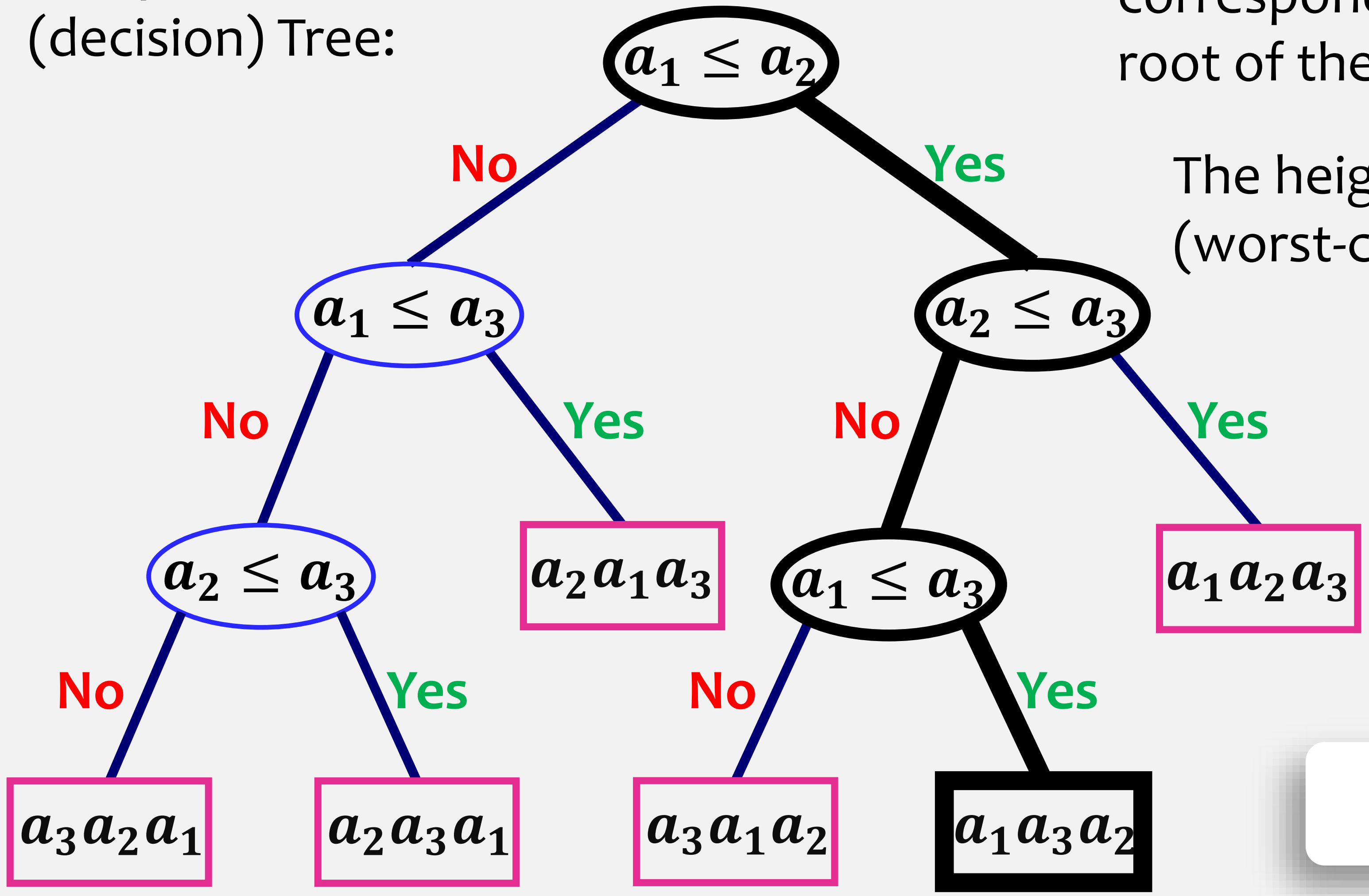
Input: a_1, a_2, a_3

Output:
reordering a'_1, a'_2, a'_3 of the input such that

$$a'_1 \leq a'_2 \leq a'_3$$

Comparison-based Sorting $\Omega(n \log n)$

Comparison (decision) Tree:



The execution of the sorting algorithm corresponds to **tracing a path** from the root of the decision tree to a leaf.

The height of the tree is equal to the (worst-case) number of comparisons

Leaf: sorted sequence.

Input: $a_1, a_2, \dots, a_n$

# of leaves: $n!$

max height: $\log(n!)$

$$\log(n!) \ge cn \log n, \; c > 0$$

Comparison-based Sorting $\Omega(n \log n)$

Every comparison tree that sorts any input of length $\boldsymbol{n}$ has **height** at least $\dfrac{\boldsymbol{n}}{\mathbf{2}} \log_2 \dfrac{\boldsymbol{n}}{\mathbf{2}}$

- The **height** of a binary tree with $\boldsymbol{m}$ leaves is at least $\mathbf{log_2}\, \boldsymbol{m}$
- The **height** of a binary tree with $\boldsymbol{n}!$ leaves is at least $\mathbf{log_2}(\boldsymbol{n}!)$

$$\begin{aligned}
\log_2 n! &= \log_2(n) + \log_2(n-1) + \cdots + \log_2(1) \\
&\geq \log_2(n) + \cdots + \log_2(n/2) \\
&\geq \log_2(n/2) + \cdots + \log_2(n/2) \\
&= (n/2)\log_2(n/2)
\end{aligned}$$

- Lower bound can be improved to $\boldsymbol{n} \ln \boldsymbol{n} - \boldsymbol{O}(\boldsymbol{n})$.

$$\mathbf{log}(\boldsymbol{n}!) \geq \boldsymbol{cn} \,\mathbf{log}\, \boldsymbol{n},\ \ \boldsymbol{c} > \mathbf{0}$$

Theorems 11.5 and 11.6

For every deterministic comparison-based sorting algorithm $\boldsymbol{A}$ and any integer $\boldsymbol{n} \geq \mathbf{1}$, there exists an input array $\boldsymbol{a}$ of length $\boldsymbol{n}$ such that $\boldsymbol{A}$ requires $\boldsymbol{\Omega}(\boldsymbol{n} \log \boldsymbol{n})$ comparisons to sort $\boldsymbol{a}$.

For every comparison-based sorting algorithm $\boldsymbol{A}$, the expected number of comparisons done by $\boldsymbol{A}$ when sorting a random permutation of $\{\mathbf{1}, \dots, \boldsymbol{n}\}$ is $\boldsymbol{\Omega}(\boldsymbol{n} \log \boldsymbol{n})$.

Summary

- **MergeSort** – $n \log n$ comparisons
- **QuickSort** – $1.38n \log n$ comparisons
- **HeapSort** – $2n \log n$ comparisons

- Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time.
- **MergeSort**, **QuickSort**, and **HeapSort** are **optimal** comparison-based sorting algorithms.

NOT comparison-based sorting

a:

3	1	3	1	0	3	1	3	1	0	3

c:

0	1	2	3
2	4	0	5

counters

count the number of occurrences of i in a and store this in $c[i]$

a:

$$\underbrace{0\quad 0}_{2}\quad \underbrace{1\quad 1\quad 1\quad 1}_{4}\ \underbrace{\ }_{0}\ \underbrace{3\quad 3\quad 3\quad 3\quad 3}_{5}$$

NOT comparison-based sorting

- **CountingSort** ← stable
- **RadixSort**

Suppose we have an input array $\boldsymbol{a}$ consisting of $\boldsymbol{n}$ integers, each in the range $\{\mathbf{0}, \ldots, \boldsymbol{k}-\mathbf{1}\}$.

Specialized for sorting "**small**" integers.

They use (parts of) the elements in $\boldsymbol{a}$ as indices into an array.

They can sort faster than comparison-based algorithms – faster than $\boldsymbol{\Omega}(\boldsymbol{n} \log \boldsymbol{n})$.

not much less

CountingSort is very efficient for sorting an array of integers when $\mathbf{n} \; \neg\ll \; \underbrace{\boldsymbol{k}-\mathbf{1}}_{\text{maximum value in the array}}$

length of the array

RadixSort uses several passes of **CountingSort** to allow for a much greater range of maximum values.

CountingSort

```java
int[] countingSort(int[] a, int k) {
  int c[] = new int[k];
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  int b[] = new int[a.length];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  return b;
}
```

a:	3	1	3	1	0	3	1	3	1	0	3

c:	2	4	0	5
	0	1	2	3

- Suppose we have an input array $\boldsymbol{a}$ consisting of $\boldsymbol{n}$ integers, each in the range $\{\mathbf{0}, \dots, \boldsymbol{k}-\mathbf{1}\}$.
- For each $\boldsymbol{i} \in \{\mathbf{0}, \dots, \boldsymbol{k}-\mathbf{1}\}$, count the number of occurrences of $\boldsymbol{i}$ in $\boldsymbol{a}$ and store this in $\boldsymbol{c}[\boldsymbol{i}]$.
- Compute a **running-sum** of the counters so that $\boldsymbol{c}[\boldsymbol{i}]$ becomes the number of elements in $\boldsymbol{a}$ that are less than or equal to $\boldsymbol{i}$.
- Scan $\boldsymbol{a}$ backwards to place its elements, in order, into an output array $\boldsymbol{b}$. When scanning, the element $\boldsymbol{a}[\boldsymbol{i}]$ is placed at location $\boldsymbol{b}\big[\boldsymbol{c}[\boldsymbol{a}[\boldsymbol{i}]]-\mathbf{1}\big]$ and the value $\boldsymbol{c}[\boldsymbol{a}[\boldsymbol{i}]]$ is decremented.

stable

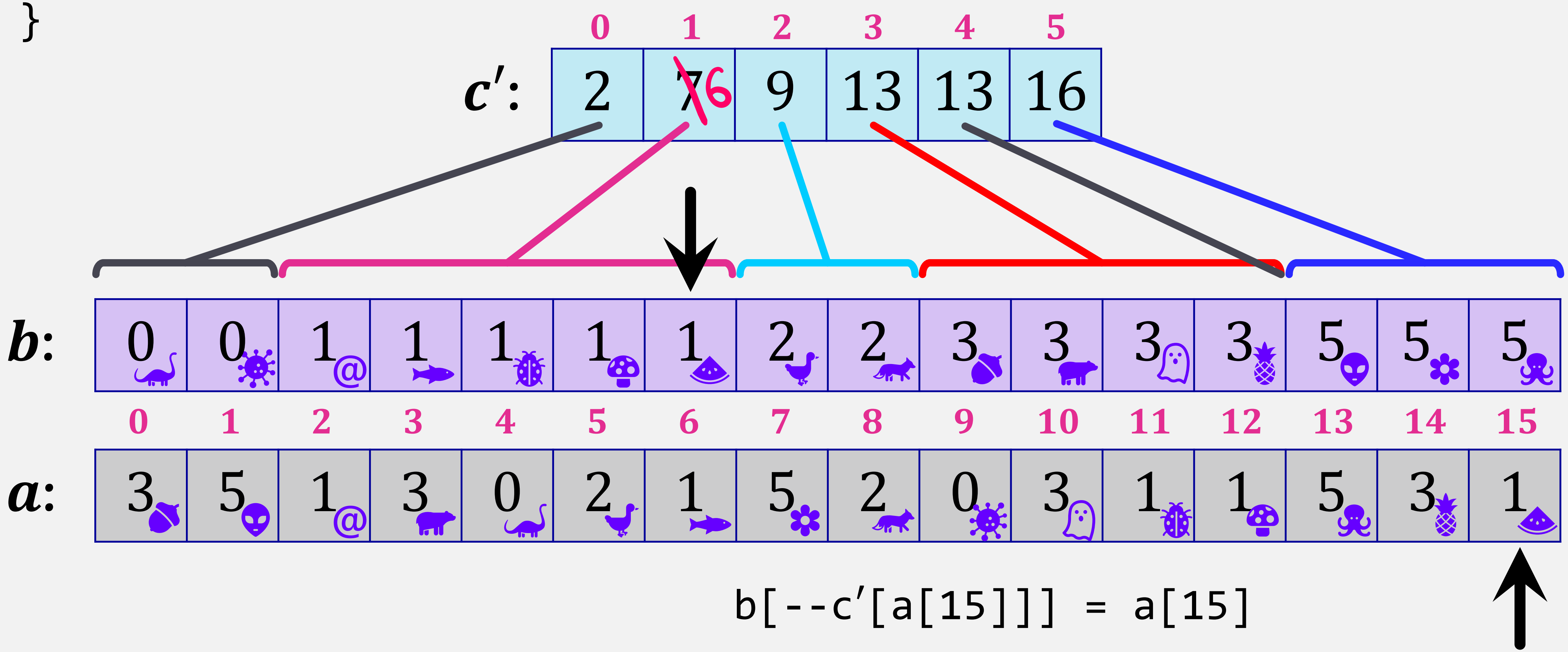
	0	1	2	3	4	5
c':	2	7	9	13	13	16

there are 16 items in a

	0	1	2	3	4	5
c:	2	5	2	4	0	3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b:	0	0	1	1	1	1	1	2	2	3	3	3	3	5	5	5
a:	3	5	1	3	0	2	1	5	2	0	3	1	1	5	3	1

```
for (int i = a.length - 1; i >= 0; i--){
    b[--c′[a[i]]] = a[i];
}
```

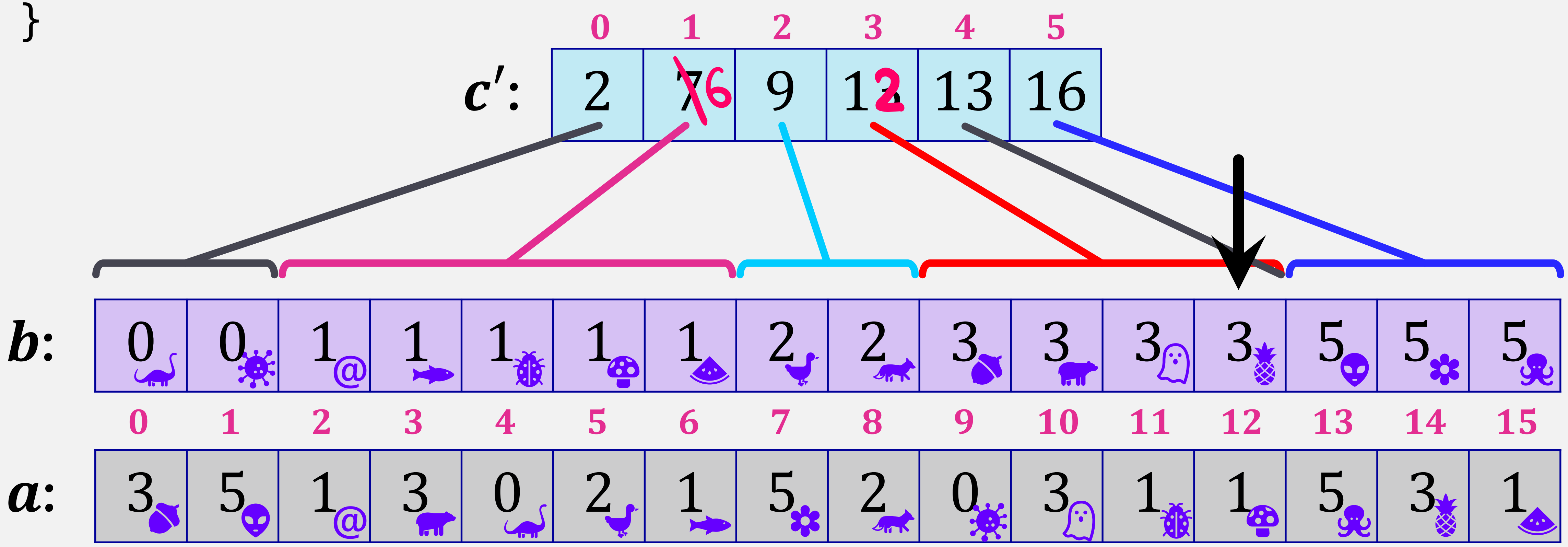


b[--c′[a[15]]] = a[15]

b[--c′[1]] = 1

b[6] = 1

```
for (int i = a.length - 1; i >= 0; i--){
    b[--c′[a[i]]] = a[i];
}
```

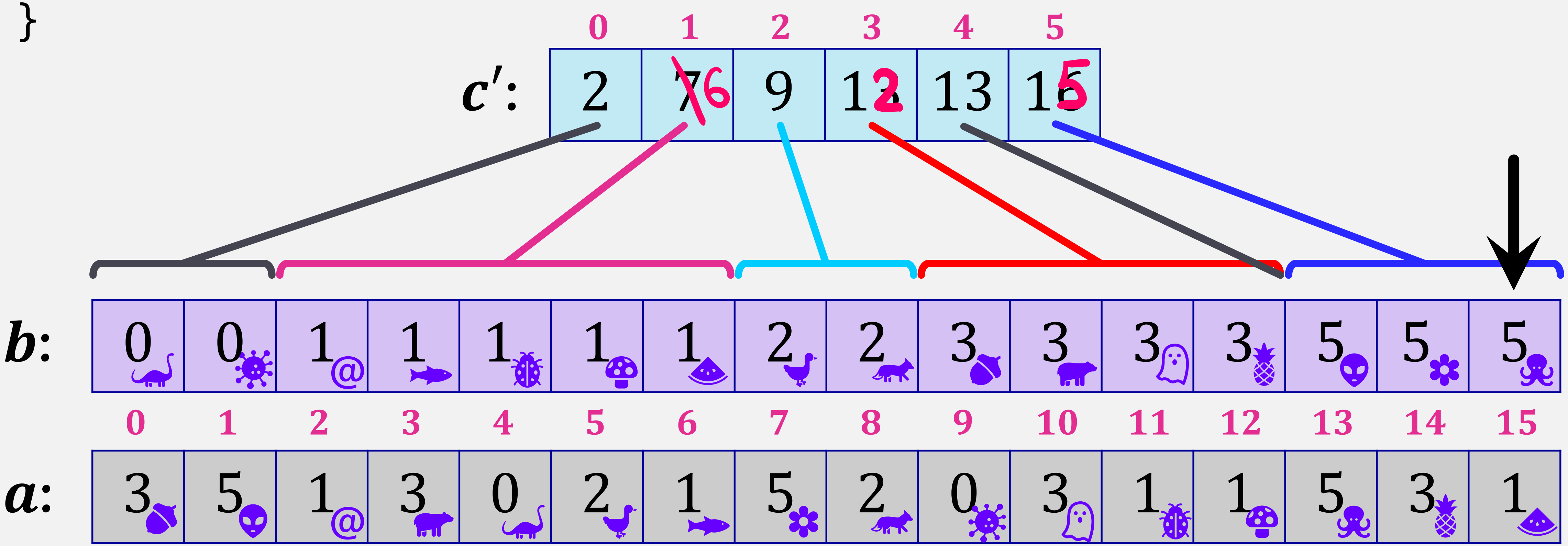


b[--c′[a[14]]] = a[14]

b[--c′[3]] = 3

b[12] = 3

```
for (int i = a.length - 1; i >= 0; i--){
    b[--c′[a[i]]] = a[i];
}
```

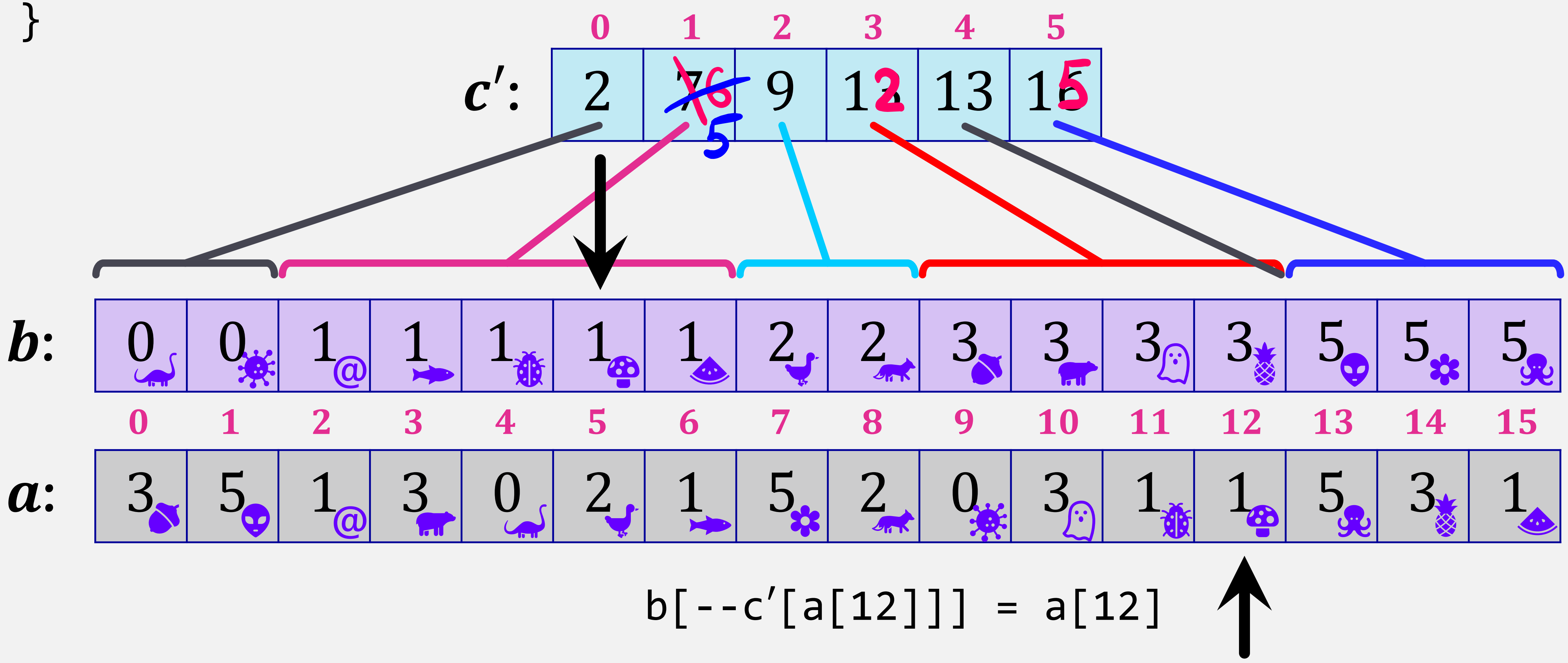


b[--c′[a[13]]] = a[13]

b[--c′[5]] = 5

b[15] = 5

```
for (int i = a.length - 1; i >= 0; i--){
    b[--c′[a[i]]] = a[i];
}
```

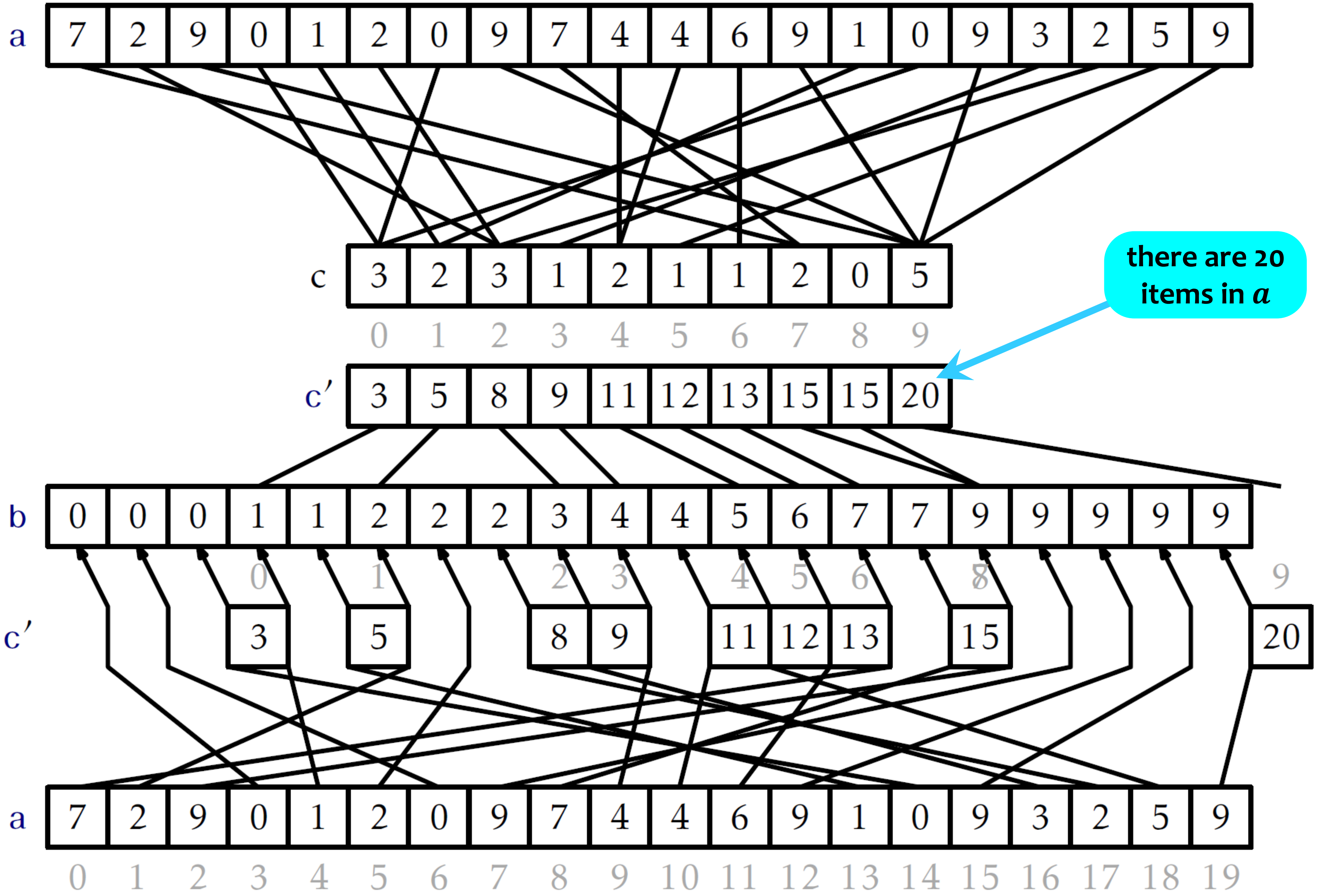



```
b[--c′[a[12]]] = a[12]

b[--c′[1]] = 1

b[5] = 1
```

ods textbook



Alina Shaikhet – COMP 2402 – Carleton University

CountingSort

```
int[] countingSort(int[] a, int k) {
  int c[] = new int[k];
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  int b[] = new int[a.length];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  return b;
}
```

executes n times

executes k times

executes n times

$O(n+k)$

Theorem 11.7

The **countingSort**$(\boldsymbol{a}, \boldsymbol{k})$ method can sort an array $\boldsymbol{a}$ containing $\boldsymbol{n}$ integers in the set $\{\mathbf{0}, \ldots, \boldsymbol{k} - \mathbf{1}\}$ in $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{k})$ time.

The **CountingSort** algorithm is **stable;** it preserves the relative order of equal elements: If two elements $\boldsymbol{a}[\boldsymbol{i}]$ and $\boldsymbol{a}[\boldsymbol{j}]$ have the same value, and $\boldsymbol{i} < \boldsymbol{j}$ then $\boldsymbol{a}[\boldsymbol{i}]$ will appear before $\boldsymbol{a}[\boldsymbol{j}]$ in $\boldsymbol{b}$.

RadixSort

not much less

CountingSort is very efficient for sorting an array of integers when $\mathbf{n} \neg\ll \underbrace{k-1}_{\text{maximum value in the array}}$

length of the array

RadixSort uses several passes of **CountingSort**
to allow for a much greater range of maximum values.

RadixSort sorts integers one **digit** at a time:

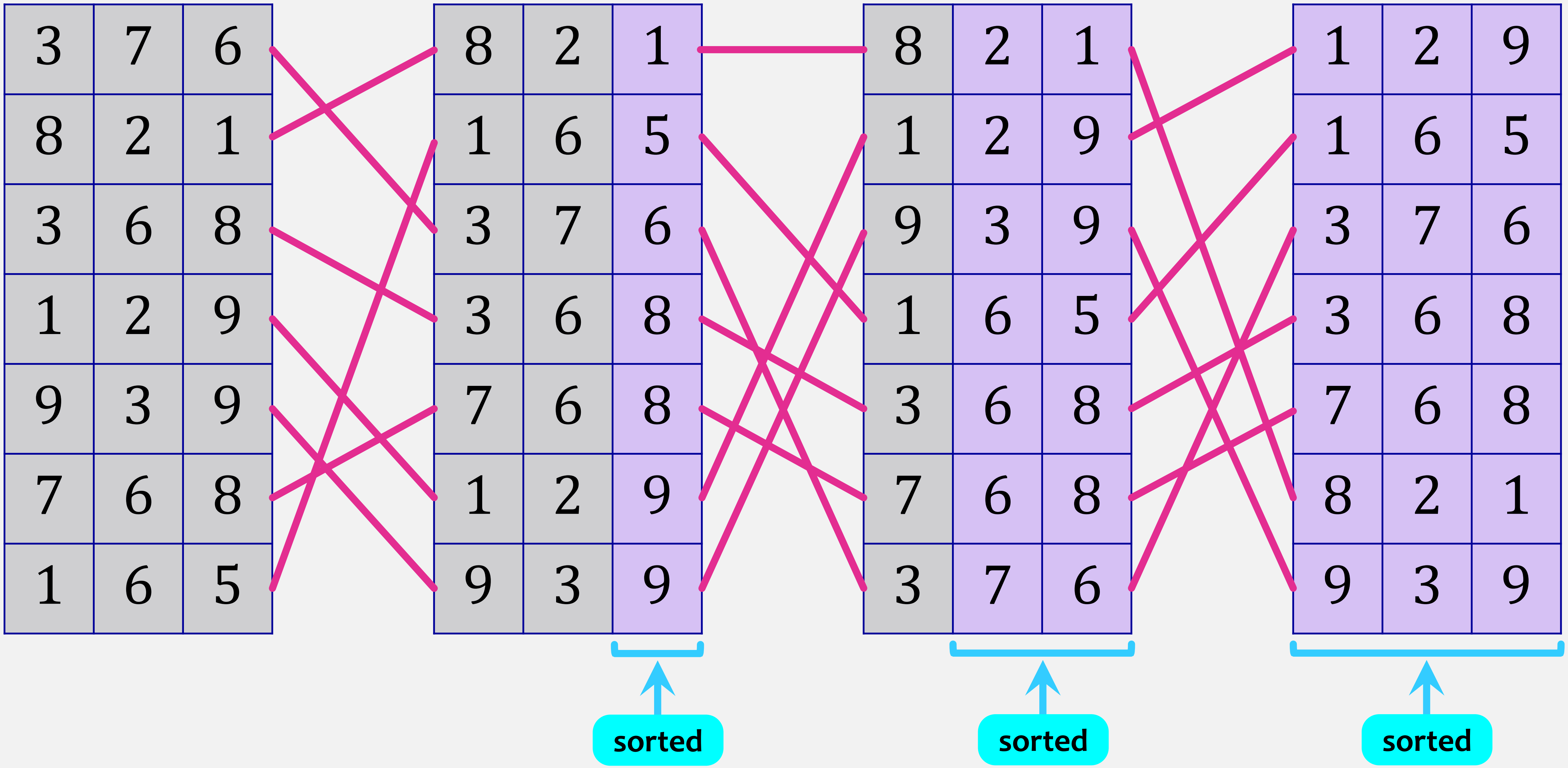
- integers have w bits
- **digit** has d bits
- uses w/d passes of **CountingSort**

We assume that d divides w, otherwise we can always increase w.

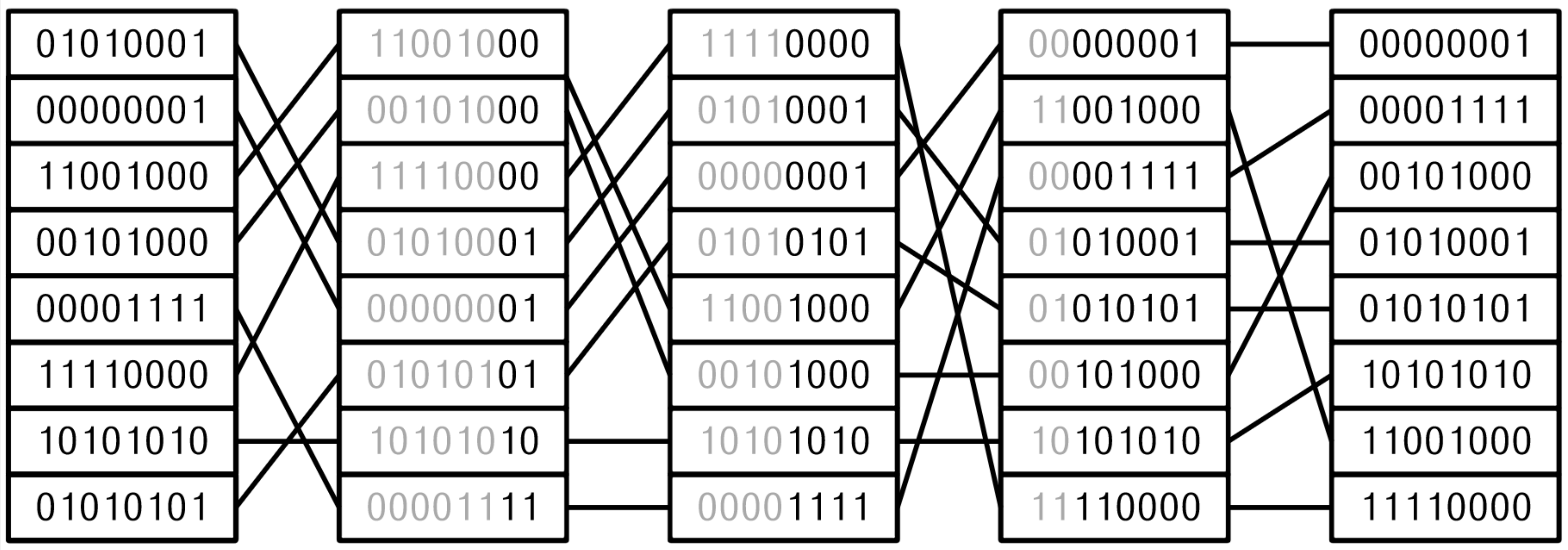
RadixSort starts by sorting the integers by their least-significant digit (d bits), then their next significant digit, and so on until, in the last pass, the integers are sorted by their most significant digit (d bits).

RadixSort

The algorithm sorts correctly because CountingSort is a **stable** sorting algorithm



RadixSort



ods textbook

Theorem 11.8

RadixSort performs w/d passes of **CountingSort**.
Each pass requires $O(n + 2^d)$ time.

For any integer $d > 0$, the **RadixSort** algorithm can sort an array a containing n w-bit integers in $O\left(\left(\frac{w}{d}\right)(n + 2^d)\right)$ time.

Take $d = \lceil \log_2 n \rceil$

The **RadixSort** algorithm can sort an array a containing n integers in the range $\{0, \ldots, n^c - 1\}$ in $O(cn)$ time.

Summary

MergeSort

QuickSort

HeapSort

- can each sort an array of length $\boldsymbol{n}$ in $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ time.
- they work for any comparable data type.
- **QuickSort** and **HeapSort** are in-place but do more comparisons.
- **MergeSort** requires an auxiliary array.

RadixSort

- can sort an array $\boldsymbol{a}$ of $\boldsymbol{n}$ integers in the range $\{\mathbf{0}, \dots, \boldsymbol{n}^{\boldsymbol{c}} - \mathbf{1}\}$ in $\boldsymbol{O}(\boldsymbol{cn})$ time (and does no comparisons).