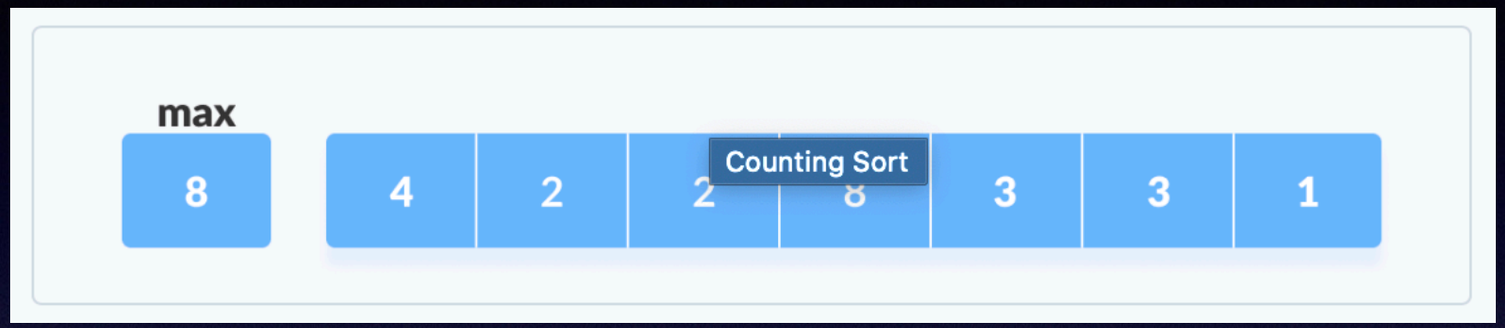
Data Structures and Algorithms

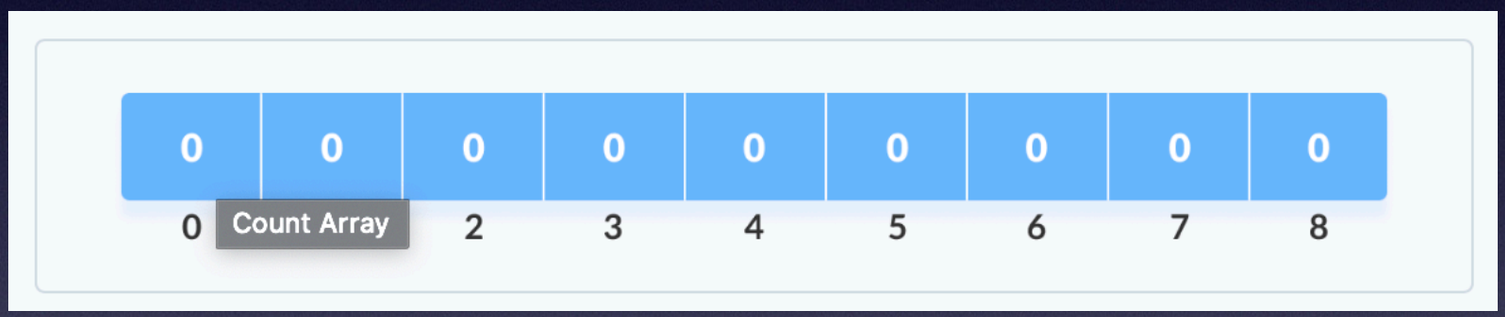
Counting Sort / Radix Sort / Sort on Strings

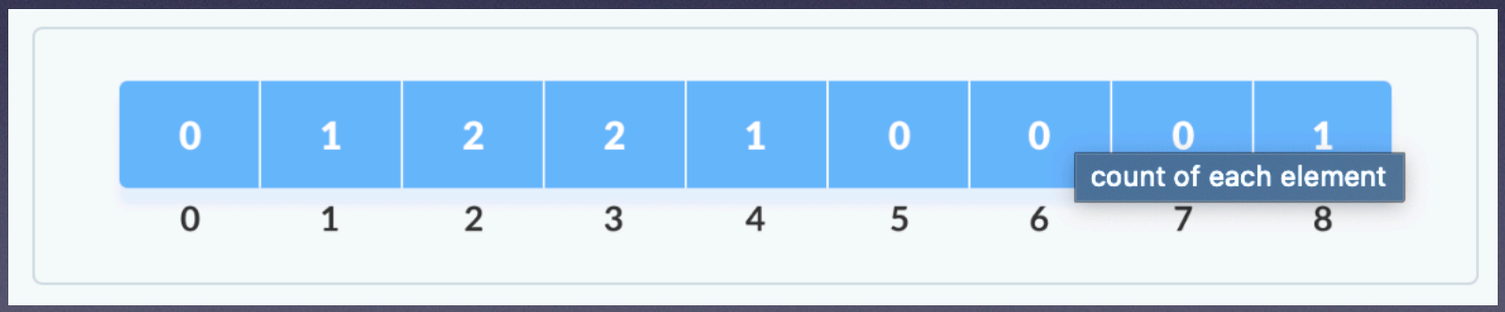
Counting Sort

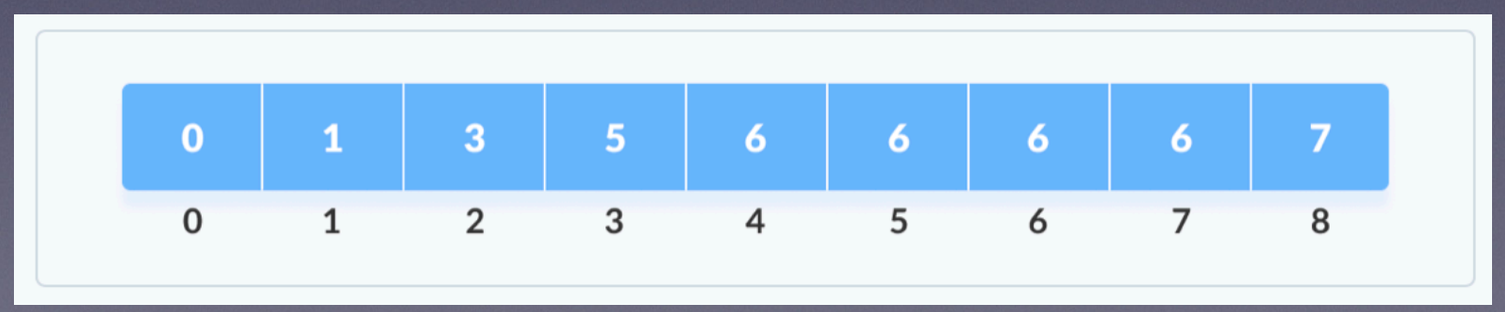
- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.
- The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Counting Sort

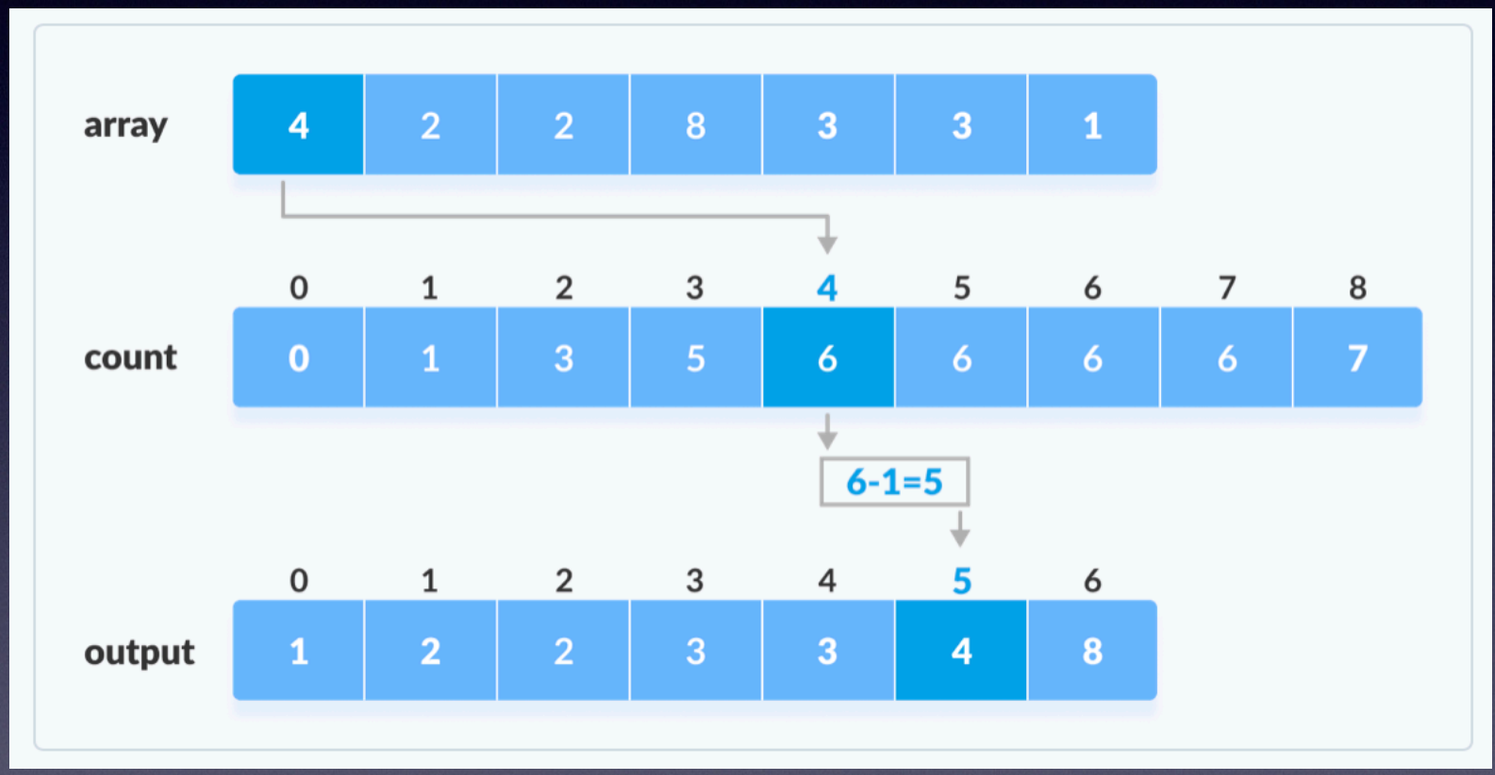








Counting Sort - II



Counting Sort Complexity

Time Complexities:

There are mainly four main loops. (Finding the greatest value can be done outside the function.)

for-loop	time of counting
1st	O(max)
2nd	O(size)
3rd	O(max)
4th	O(size)

Overall complexity = `O(max)+O(size)+O(max)+O(size)` = `O(max+size)`

- **Worst Case Complexity:** `O(n+k)`
- **Best Case Complexity:** `O(n+k)`
- **Average Case Complexity:** `O(n+k)`

In all the above cases, the complexity is same because no matter how the elements are placed in the array, the algorithm goes through `n+k` times.

Space Complexity:

The space complexity of Counting Sort is `O(max)`. Larger the range of elements, larger is the space complexity.

Radix Sort

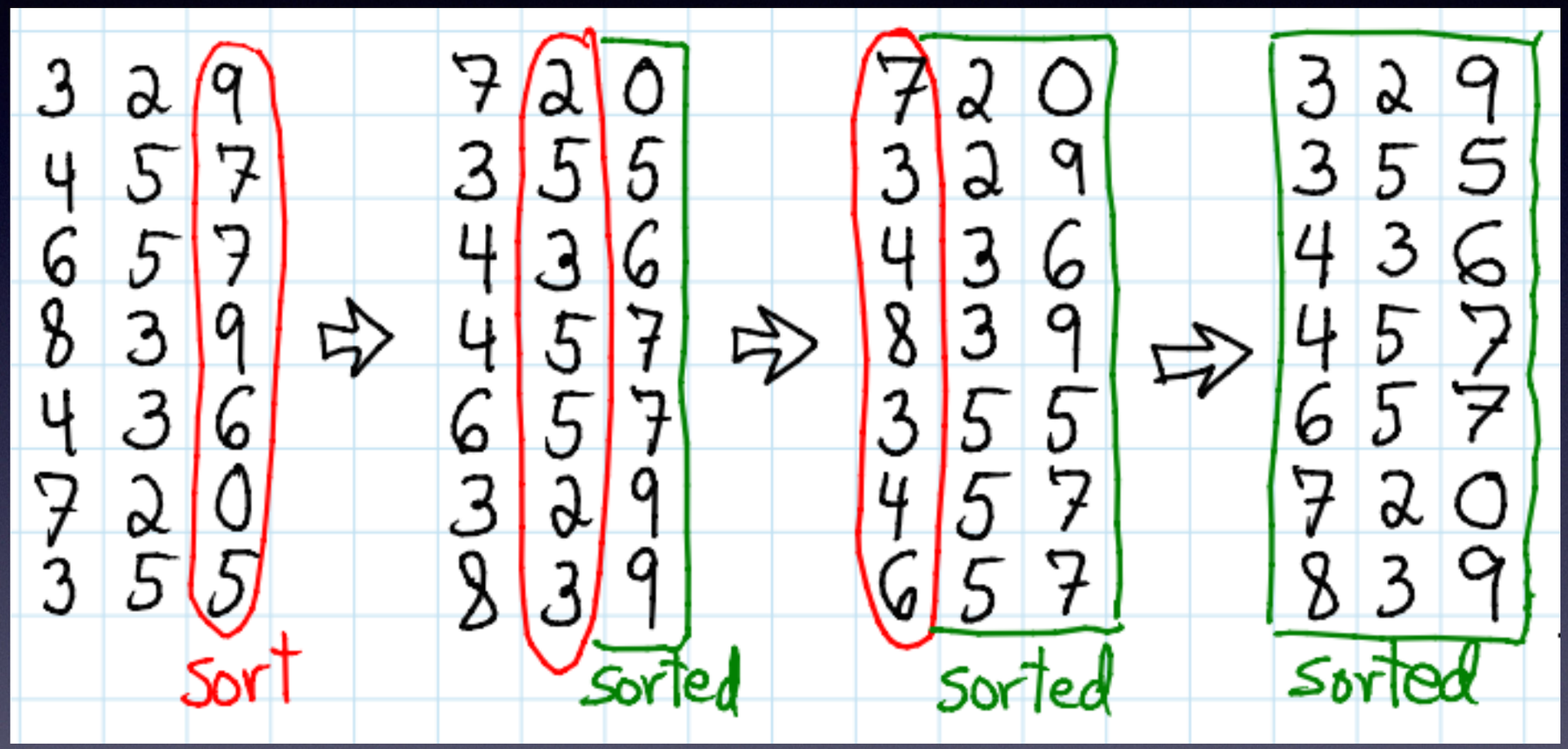
- Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same **place value**. Then, sort the elements according to their increasing/decreasing order.

Radix Sort - II

Initial Array

`[121, 432, 564, 23, 1, 45, 788]`

sorting the integers according to units, tens and hundreds place digits


3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
sort
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
sorted
7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7
sorted
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9
sorted

Radix Sort Complexity

```
O(d(n+k))
```

```
O(nlog n)
```

If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space.

This makes radix sort space inefficient. This is the reason why this sort is not used in software libraries.

Radix Sort Complexity

Radix sort takes in a list of n integers which are in base b (the radix) and so each number has at most d digits where $d = \lfloor (\log_b(k) + 1) \rfloor$ and k is the largest number in the list. For example, three digits are needed to represent decimal 104 (in base 10). It is important that radix sort can work with any base since the running time of the algorithm, $O(d(n + b))$, depends on the base it uses. The algorithm runs in linear time when b and n are of the same size magnitude, so knowing n, b can be manipulated to optimize the running time of the algorithm.

Radix sort is a **stable sort**, which means it preserves the relative order of elements that have the same key value. This is very important.

Why Radix Sort

Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

What if the elements are in range from 1 to n^2?

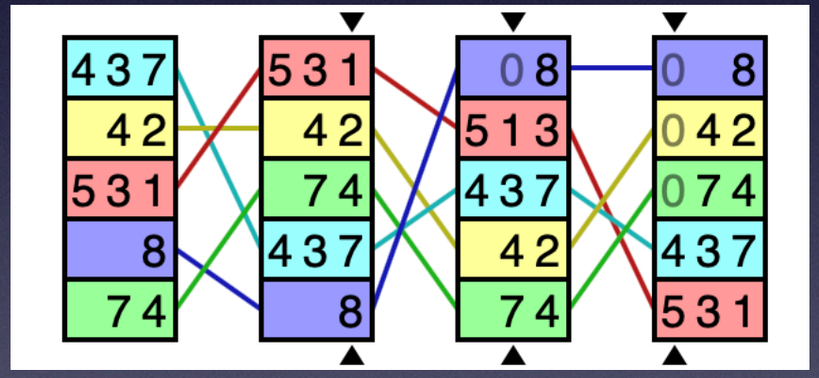
We can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

Radix Sort Variants

- Least Significant Digit Radix Sort
- Most Significant Digit Radix Sort

LSD Radix Sort

The LSD variant of radix sort performs a stable counting sort on the list for each digit, starting from the least significant (right-most) digit. It runs in $O(wn)$ time where n is the input size and w is the word size (the number of digits in the largest number for the given radix).

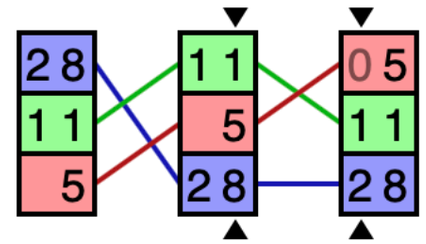


Complexity

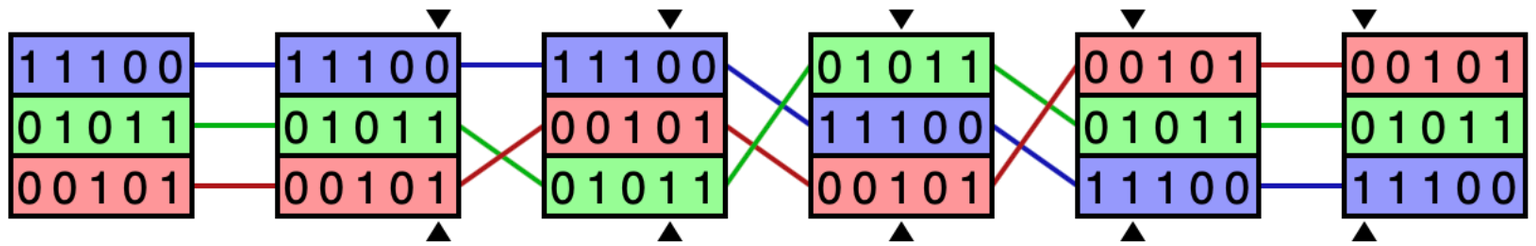
Time			Space
Worst case	Best case	Average case	Worst case
$O(wn)$	$O(wn)$	$O(wn)$	$O(n + r)$ auxiliary

Where:

- n = input size
- w = word size
- r = radix



Sorting [28,11,5] where r=10 and w=2



Sorting [28,11,5] where r=2 and w=5

When it's fast

Since comparison sorts cannot perform better than $O(n \log n)$, LSD radix sort is considered one of the best alternatives provided the word size w is expected to be less than $\log n$.

It does however have limitations on the type of keys that can be sorted in that they need to have some way of being split up (ie. the radix), so it's typically only used for string (where $r = 255$ for ASCII characters) and integer keys.

Sort on Strings

Key-indexed counting

Task: sort an array `a[]` of N integers between 0 and R-1

Plan: produce sorted result in array `temp[]`

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates
3. Access cumulates using key as index to find record positions.
4. Copy back into original array

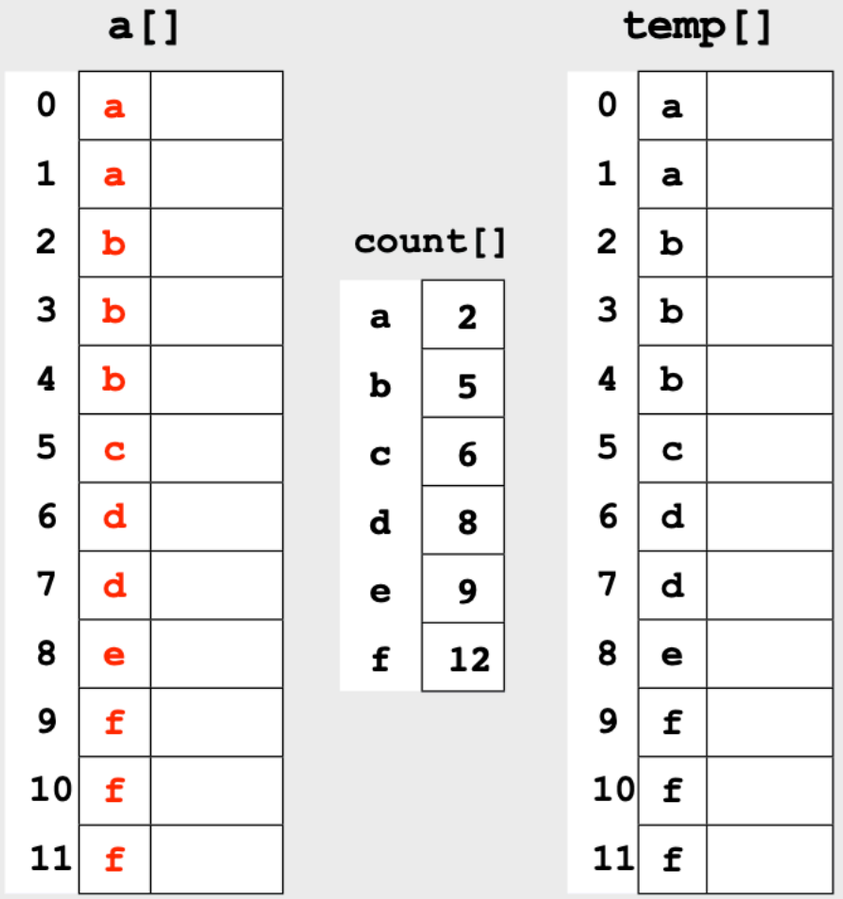
```
int N = a.length;
int[] count = new int[R];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int k = 1; k < 256; k++)
   count[k] += count[k-1];

for (int i = 0; i < N; i++)
   temp[count[a[i]++]] = a[i]

for (int i = 0; i < N; i++)
   a[i] = temp[i];
```

count frequencies

compute cumulates

move records

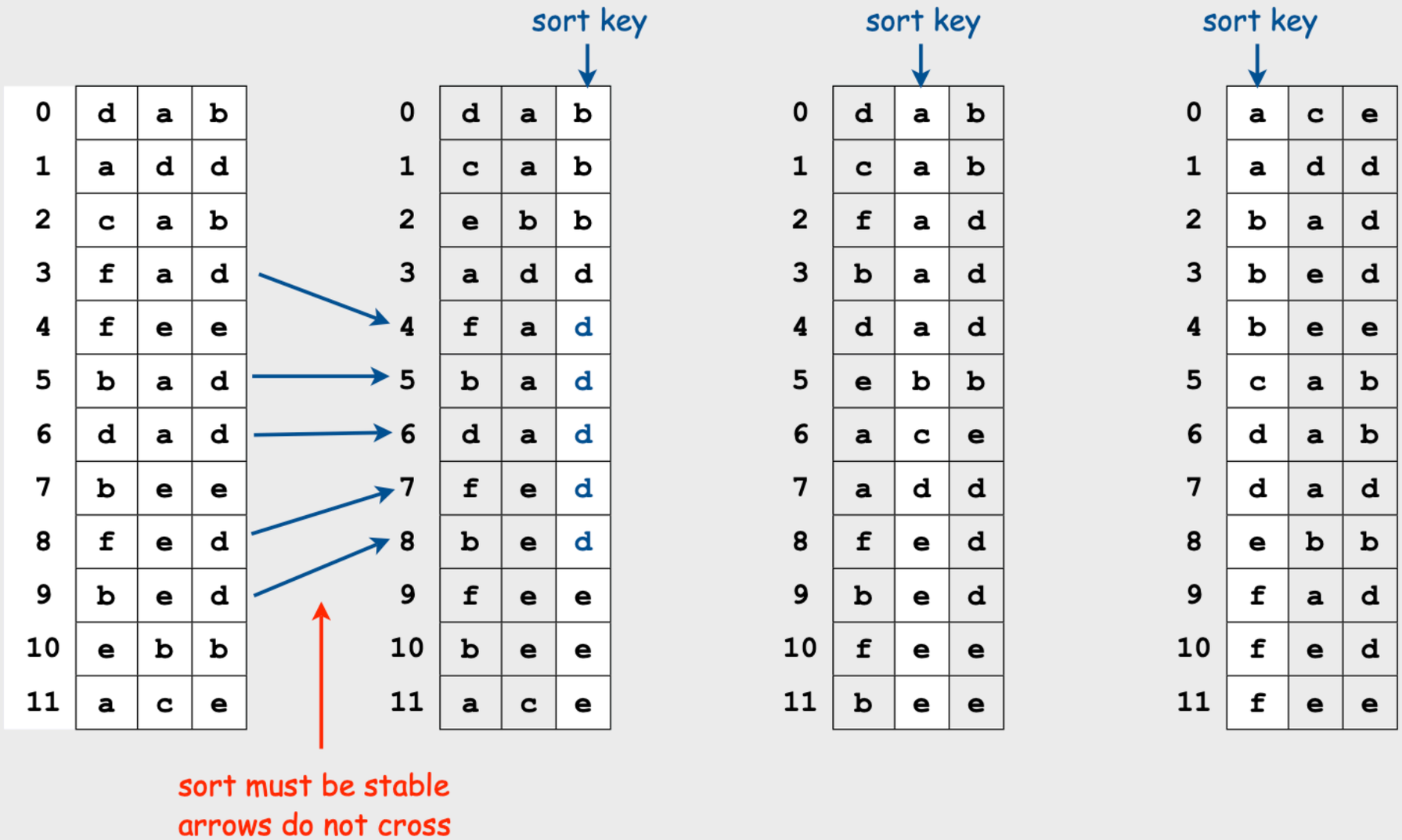
copy back



Least-significant-digit-first radix sort

LSD radix sort.

- Consider characters d from right to left
- Stably sort using dth character as the key via key-indexed counting.



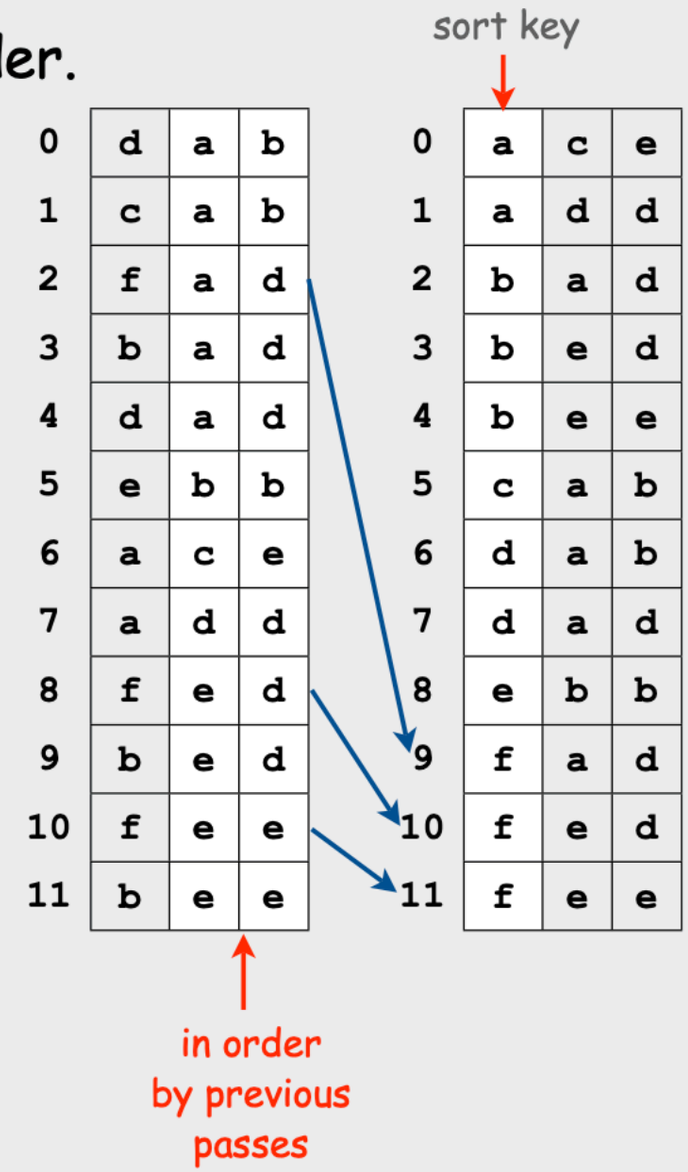
LSD radix sort: Why does it work?

Pf 1. [thinking about the past]

- If two strings differ on first character, key-indexed sort puts them in proper relative order.
- If two strings agree on first character, stability keeps them in proper relative order.

Pf 2. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.



LSD radix sort implementation

Use k-indexed counting on characters, moving right to left

```
public static void lsd(String[] a)
{
   int N = a.length;
   int W = a[0].length;
   for (int d = W-1; d >= 0; d--)
   {
      int[] count = new int[R];
      for (int i = 0; i < N; i++)
         count[a[i].charAt(d) + 1]++;
      for (int k = 1; k < 256; k++)
         count[k] += count[k-1];
      for (int i = 0; i < N; i++)
         temp[count[a[i].charAt(d)]++] = a[i];
      for (int i = 0; i < N; i++)
         a[i] = temp[i];
   }
}
```

key-indexed counting

count frequencies

compute cumulates

move records

copy back

Assumes fixed-length keys (length = W)

Sorting Challenge

Problem: sort a huge commercial database on a fixed-length key field

Ex: account number, date, SS number

	B14-99-8765		
	756-12-AD46		
	CX6-92-0112		
	332-WX-9877		
	375-99-QWAX		
	CV2-59-0221		
	[illegible]7-SS-032[illegible]		
	KJ-[illegible]2388		
	715-YT-013C		
	MJ0-PP-983F		
	908-KK-33TY		
	BBN-63-23RE		
	48G-BM-912D		
	982-ER-9P1B		
	WBL-37-PB81		
	810-F4-J87Q		
	LE9-N8-XX76		
	908-KK-33TY		
	B14-99-8765		
	CX6-92-0112		
	CV2-59-0221		
	332-WX-23SQ		
	332-6A-9877		

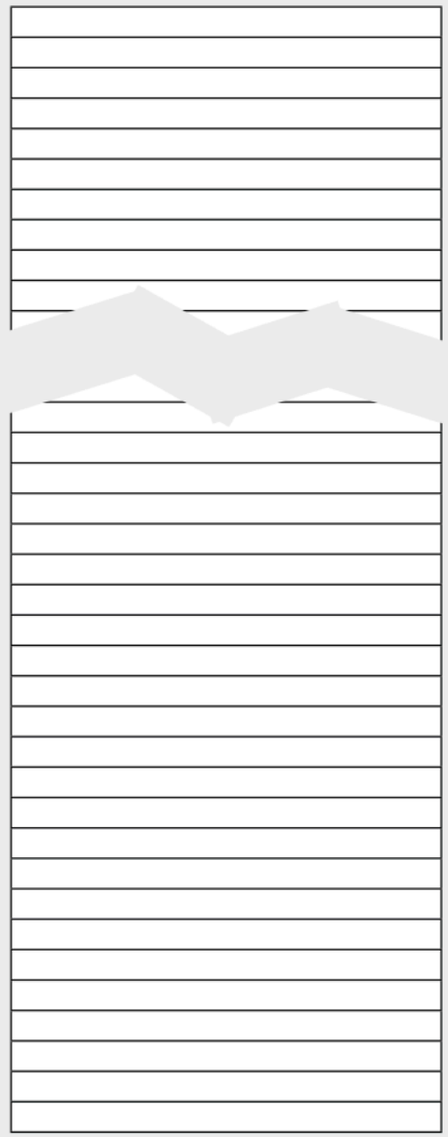
Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort

Sorting Challenge

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router



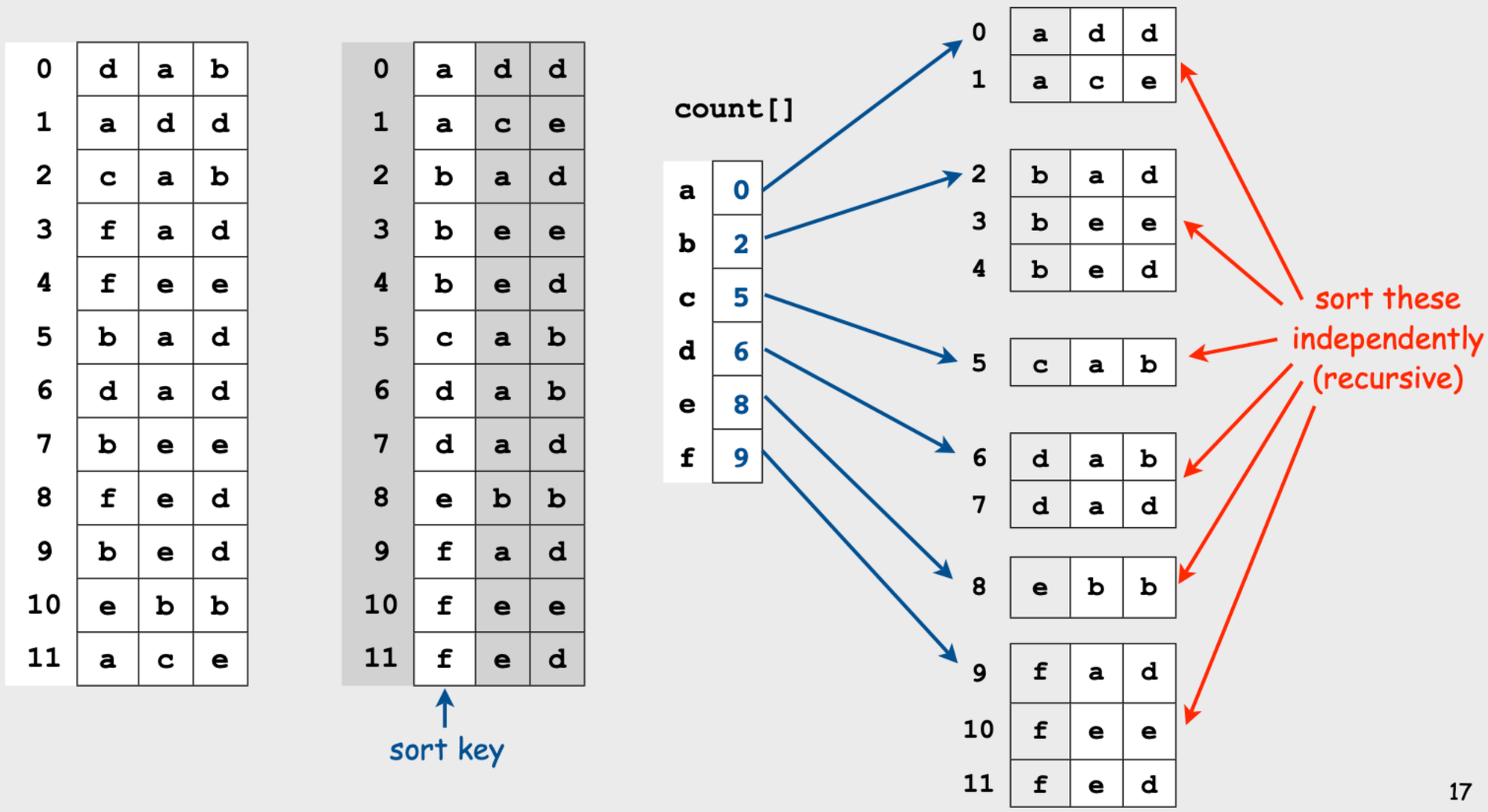
Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort

MSD Radix Sort

Most-significant-digit-first radix sort.

- Partition file into R pieces according to first character (use key-indexed counting)
- Recursively sort all strings that start with each character (key-indexed counts delineate files to sort)



MSD radix sort implementation

Use key-indexed counting on first character, recursively sort subfiles

```
public static void msd(String[] a)
{  msd(a, 0, a.length, 0);  }

private static void msd(String[] a, int lo, int hi, int d)
{
   if (hi <= lo + 1) return;
   int[] count = new int[256+1];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   for (int i = 0; i < N; i++)
      temp[count[a[i].charAt(d)]++] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = temp[i];
   for (int i = 0; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

key-indexed counting →

← count frequencies

← compute cumulates

← move records

← copy back

MSD radix sort: potential for disastrous performance

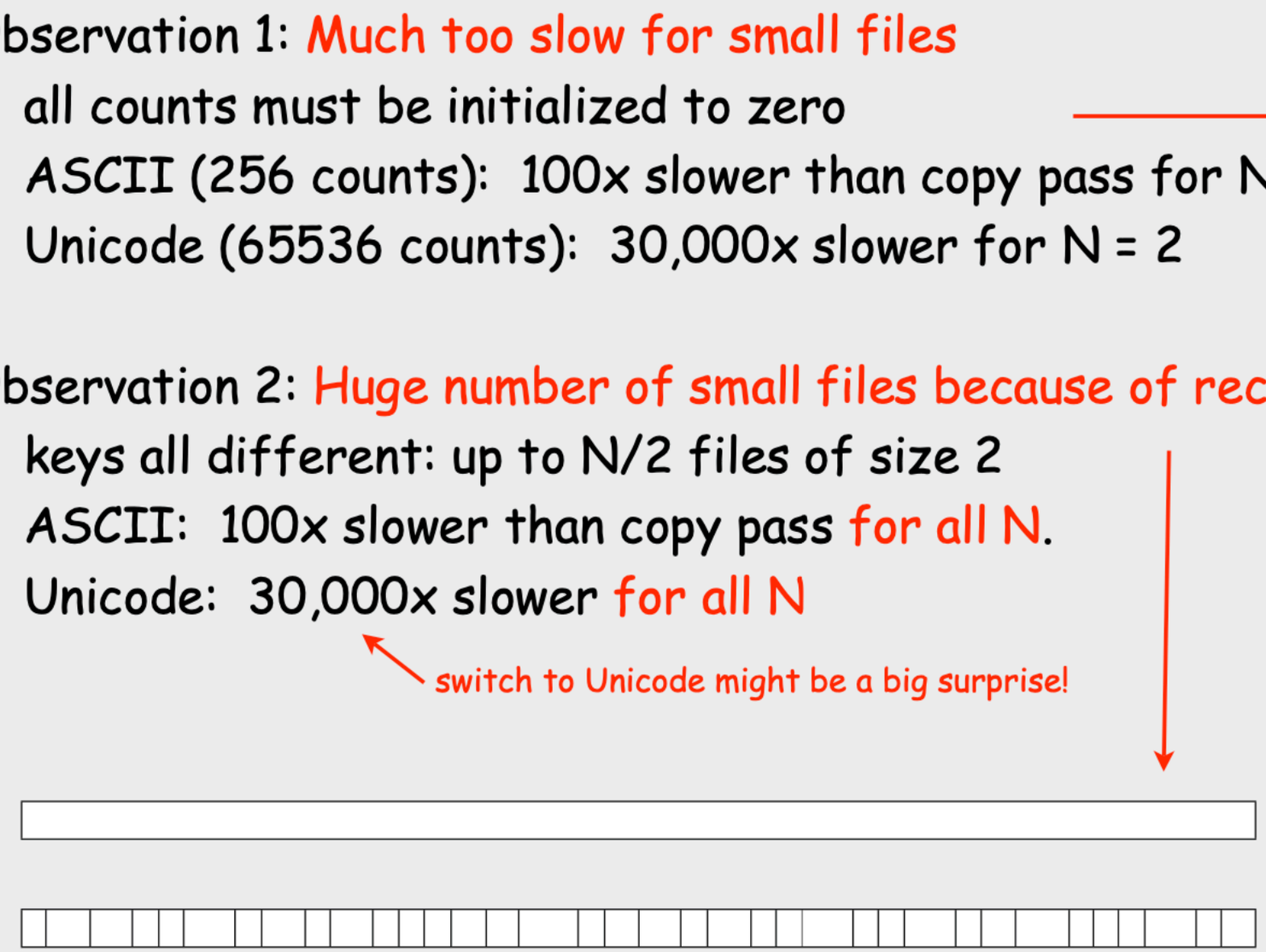
Observation 1: Much too slow for small files

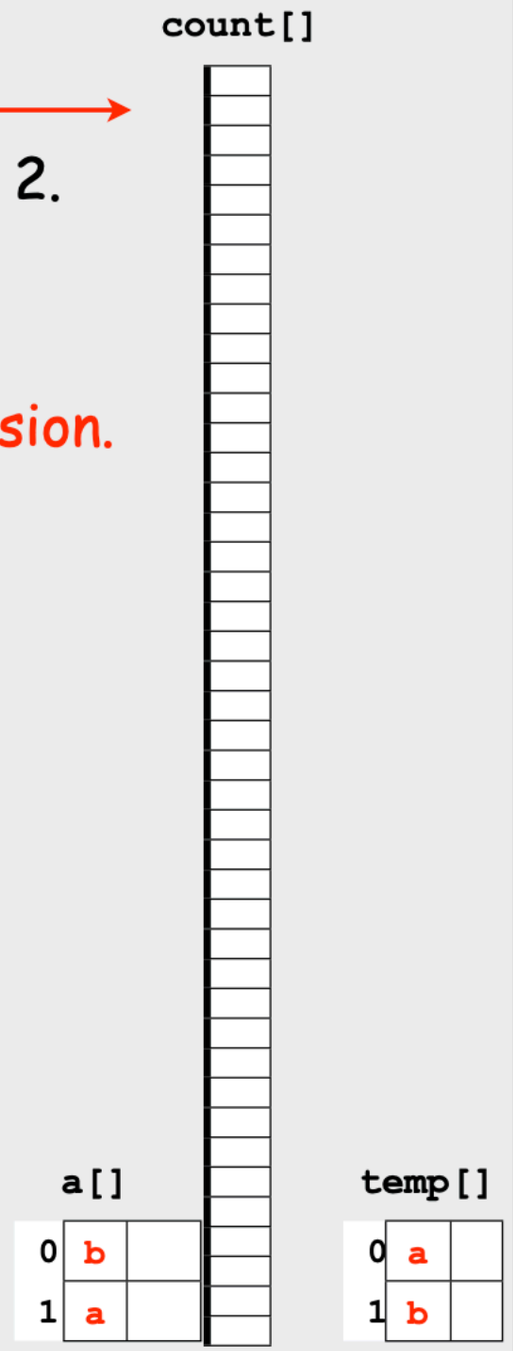
- all counts must be initialized to zero
- ASCII (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 30,000x slower for N = 2

Observation 2: Huge number of small files because of recursion.

- keys all different: up to N/2 files of size 2
- ASCII: 100x slower than copy pass for all N.
- Unicode: 30,000x slower for all N

switch to Unicode might be a big surprise!





Solution. Switch to insertion sort for small N.

MSD radix sort bonuses

Bonus 1: May not have to examine all of the keys.

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d

← 19/24 ≈ 80% of the characters examined

Bonus 2: Works for variable-length keys (`String` values)

0	a	c	e	t	o	n	e	\0	
1	a	d	d	i	t	i	o	n	\0	
2	b	a	d	g	e	\0			
3	b	e	d	a	z	z	l	e	d	\0
4	b	e	e	h	i	v	e	\0	
5	c	a	b	i	n	e	t	r	y	\0
6	d	a	b	b	l	e	\0		
7	d	a	d	\0					

← 19/64 ≈ 30% of the characters examined

Implication: sublinear sorts (!)

MSD string sort implementation

Use key-indexed counting on first character, recursively sort subfiles

```
public static void msd(String[] a)
{  msd(a. 0. a.length, 0);

private static void msd(String[] a, int l, int r, int d)
{
   if (r <= l + 1) return;
   int[] count = new int[256];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   for (int i = 0; i < N; i++)
      temp[count[a[i].charAt(d)]++] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = temp[i];
   for (int i = 1; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

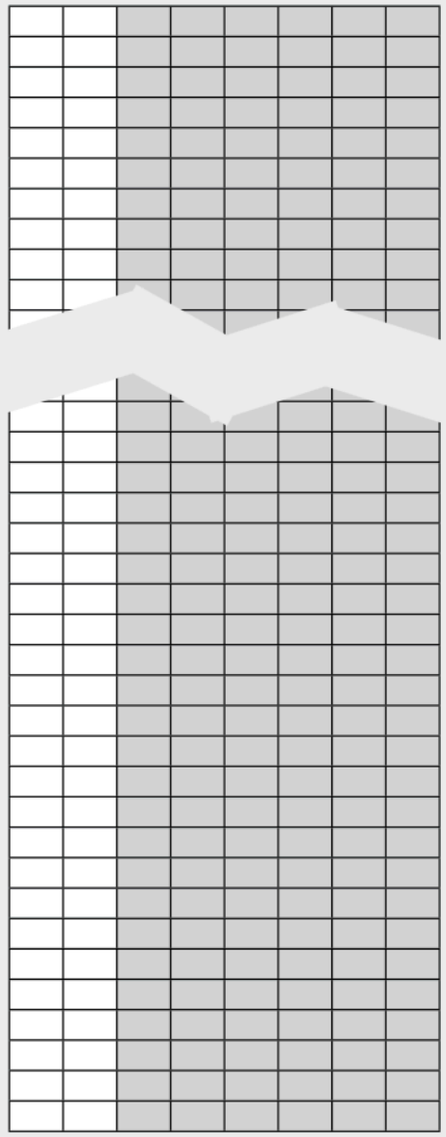
key-indexed counting

don't sort strings that start with '\0' (end of string char)

Sorting Challenge (revisited)

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router



Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. ✓ LSD radix sort on MSDs

2^{16} = 65536 counters
divide each word into 16-bit "chars"
sort on leading 32 bits in 2 passes
finish with insertion sort
examines only ~25% of the data

MSD radix sort versus quicksort for strings

Disadvantages of MSD radix sort.

- Accesses memory "randomly" (cache inefficient)
- Inner loop has a lot of instructions.
- Extra space for counters.
- Extra space for temp (or complicated inplace key-indexed counting).

Disadvantage of quicksort.

- N lg N, not linear.
- Has to rescan long keys for compares
- [but stay tuned]

Divide-and-Conquer Problems

MergeSort