Remind

	worst-case running times	on average
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$

A Lower Bound for Sorting

1. Rules for sorting.
2. The lower bound on comparison sorting.
3. Beating the lower bound with counting sort.
4. Radix sort.

Rules for sorting

"if this element's sort key is less than this other element's sort key, then do something, and otherwise either do something else or do nothing else."

Does a sorting algorithm use only this form?

No.

Rules for sorting

1) each sort key is either 1 or 2,

2) the elements consist of only sort keys.

In this simple situation, we can sort n elements in only $\Theta(n)$ time.

Rules for sorting

=>go through every element and count how many of them are 1s;
let’s say that k elements have the value 1.

=>go through the array, filling the value 1 into the first k positions and then filling the value 2 into the last n - k positions.

Rules for sorting

Procedure REALLY-SIMPLE-SORT(A, n)

Inputs:

- A: an array in which each element is either 1 or 2.
- n: the number of elements in A to sort.

Result: The elements of A are sorted into nondecreasing order.

1. Set k to 0.
2. For $i = 1$ to n:

   A. If $A[i] = 1$, then increment k.
3. For $i = 1$ to k:

   A. Set $A[i]$ to 1.
4. For $i = k + 1$ to n:

   A. Set $A[i]$ to 2.

The lower bound on comparison sorting

A comparison sort is any sorting algorithm that determines the sorted order ***only by comparing pairs of elements***.

The four sorting algorithms from the previous lecture are comparison sorts (but REALLY-SIMPLE-SORT is not).

The lower bound on comparison sorting

This is the lower bound:

- In the worst case, any comparison sorting algorithm for n elements requires $\Omega(n \lg n)$ comparisons between pairs of elements.

What is Ω-notation?

The lower bound on comparison sorting

We write: Ω-notation (It gives a lower bound)

We say: “for sufficiently large n, any comparison sorting algorithm requires at least (cnlg n) comparisons in the worst case, for some constant c”.

The lower bound on comparison sorting

1) Lower bound is saying something only about the worst case; the best case may be $\Theta(n)$ time.

In the worst case $\Omega(n \lg n)$ comparisons are necessary.

It is an **existential** lower bound.

The lower bound on comparison sorting

A **universal** lower bound => applies to all inputs.

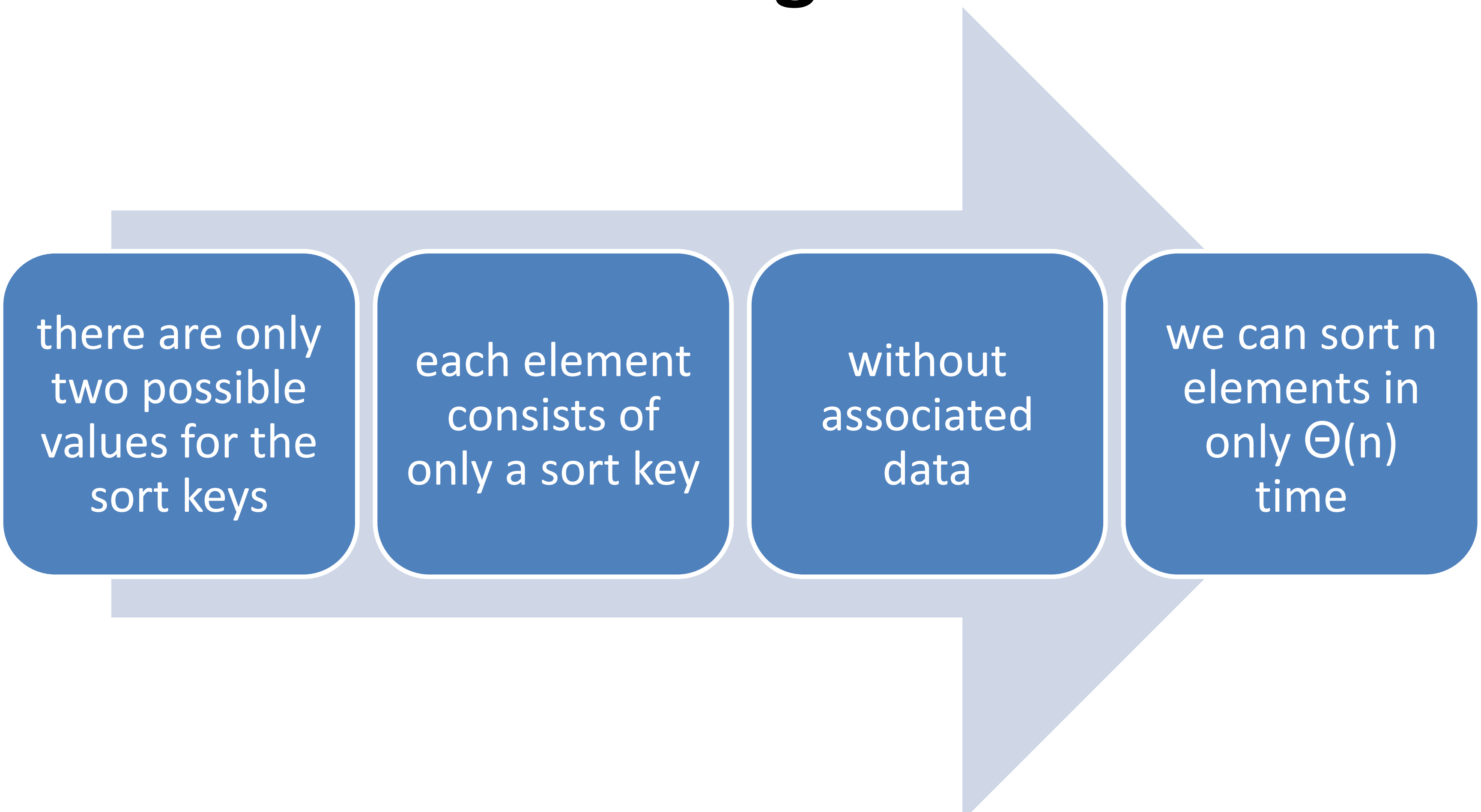
For sorting *the only universal lower bound* is $\Omega(n)$.

The lower bound on comparison sorting

2) The lower bound *<u>does not depend on</u>* the particular algorithm, as long as it's a comparison sorting algorithm.

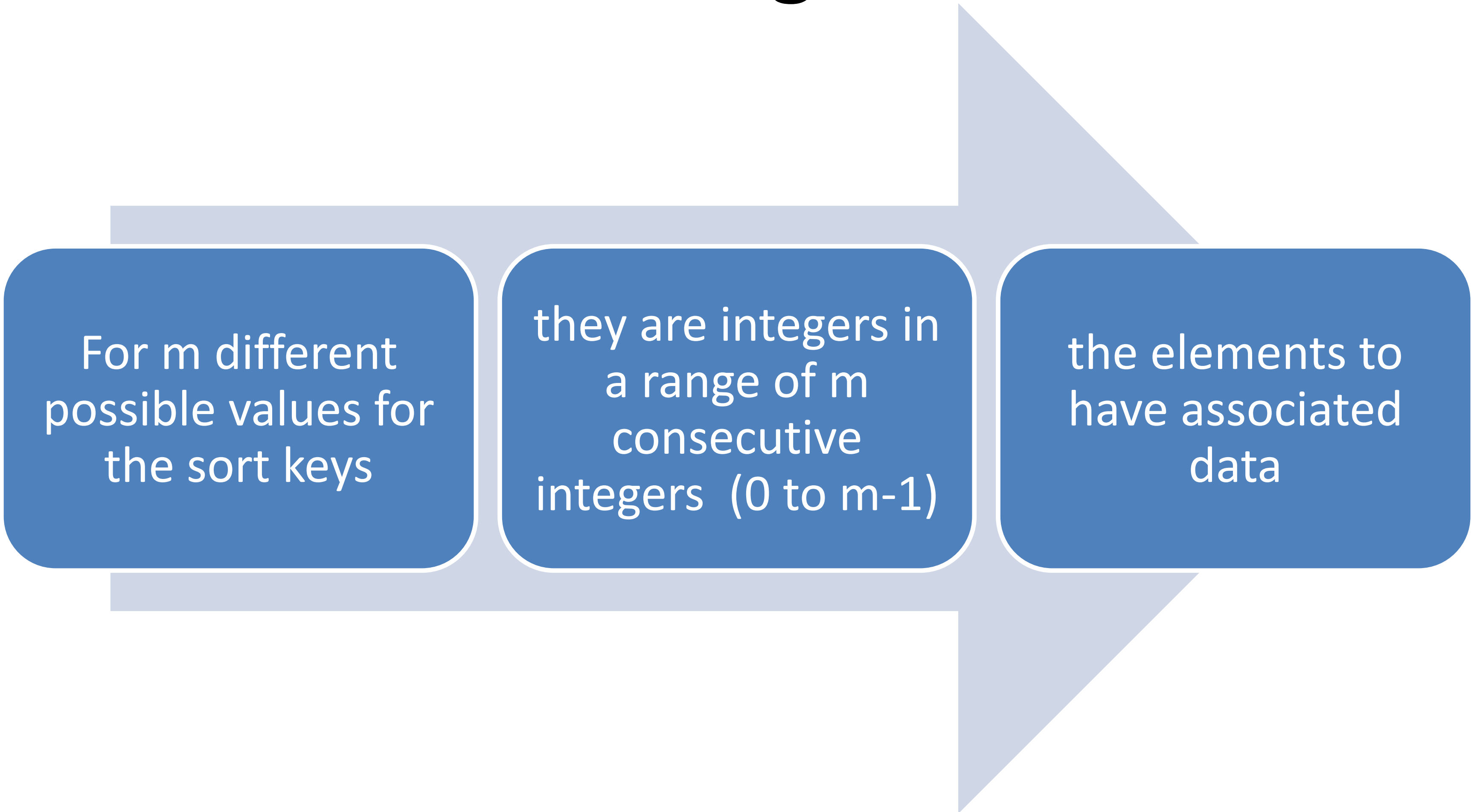
The lower bound applies to every comparison sorting algorithm, no matter how simple or complex.

Beating the lower bound with counting sort



Procedure REALLY-SIMPLE-SORT

Beating the lower bound with counting sort



Procedure COUNT-KEYS-EQUAL

Beating the lower bound with counting sort

Example. Let’s we know that the sort keys are integers in the range 0 to m-1.

And let’s we know:

- three elements have sort keys equal to 5;
- six elements have sort keys less than 5 (that is, in the range 0 to 4).

Then in the sorted array the elements with sort keys equal to 5 should occupy positions 7, 8, 9.

Beating the lower bound with counting sort

Generalize.

If k elements have sort keys equal to x and that l elements have sort keys ***less than x***, then the elements with sort keys equal to x should occupy positions $l+1$ through $l+k$ in the sorted array.

Beating the lower bound with counting sort

What should be done?

We want to compute, for each possible sort-key value,

1) how many elements have sort keys less than that value and

2) how many elements have sort keys equal to that value.

Beating the lower bound with counting sort

Procedure COUNT-KEYS-EQUAL(A, n, m)

Inputs:

- A: an array of integers in the range 0 to $m - 1$.
- n: the number of elements in A.
- m: defines the range of the values in A.

Output: An array *equal*$[0 \,.\,.\, m - 1]$ such that *equal*$[j]$ contains the number of elements of A that equal j, for $j = 0, 1, 2, \ldots, m - 1$.

1. Let *equal*$[0 \,.\,.\, m - 1]$ be a new array.
2. Set all values in *equal* to 0.
3. For $i = 1$ to n:
   A. Set *key* to $A[i]$.
   B. Increment *equal*[*key*].
4. Return the *equal* array.

Computing: how many elements have sort keys equal to that value.

Beating the lower bound with counting sort

Notice that COUNT-KEYS-EQUAL never compares sort keys with each other.

It uses sort keys only to index into the equal array.

Beating the lower bound with counting sort

Since the first loop (step 2) makes m iterations, the second loop (step 3) makes n iterations, and each iteration of each loop takes constant time, COUNT-KEYS-EQUAL takes $\Theta(m+n)$ time.

If m is a constant, then COUNT-KEYS-EQUAL takes $\Theta(n)$ time.

Beating the lower bound with counting sort

Procedure COUNT-KEYS-LESS(*equal*, m)

Inputs:

- *equal*: the array returned by COUNT-KEYS-EQUAL.
- m: defines the index range of *equal*: 0 to $m - 1$.

Output: An array $less[0..m-1]$ such that for $j = 0, 1, 2, \ldots, m-1$, $less[j]$ contains the sum $equal[0] + equal[1] + \cdots + equal[j-1]$.

1. Let $less[0..m-1]$ be a new array.
2. Set $less[0]$ to 0.
3. For $j = 1$ to $m - 1$:
   A. Set $less[j]$ to $less[j-1] + equal[j-1]$.
4. Return the *less* array.

Computing: how many elements have sort keys less than each value.

Beating the lower bound with counting sort

Example.

Suppose that $m = 7$, so that all sort keys are integers in the range 0 to 6.

Array A with $n = 10$ elements:

$A = (4; 1; 5; 0; 1; 6; 5; 1; 5; 3)$.

Beating the lower bound with counting sort

Then ***equal*** = (1; 3; 0; 1; 1; 3; 1)

A = (4; 1; 5; 0; 1; 6; 5; 1; 5; 3)

Because

- How many elements in the array A equal to 0? => 1 => then equal[0]=1
- How many elements in the array A equal to 1? => 3 => then equal[1]=3
- How many elements in the array A equal to 2? => 0 => then equal[2]=0

Beating the lower bound with counting sort

less = (0; 1; 4; 4; 5; 6; 9)***equal*** = (1; 3; 0; 1; 1; 3; 1)

Because

- less[0]= 0
- less[1]= equal [0] = 1
- less[2]= equal [0] + equal [1] =1 + 3 = 4
- less[3]= equal [0] + equal [1] + equal [2] = 1 + +3 + 0 = 4
- less[4]= equal [0] + equal [1] + equal [2] + +equal [3] = 1 + 3 + 0 + 1 = 5

Procedure REARRANGE(A, *less*, n, m)

Inputs:

- A: an array of integers in the range 0 to $m-1$.
- *less*: the array returned by COUNT-KEYS-LESS.
- n: the number of elements in A.
- m: defines the range of the values in A.

Output: An array B containing the elements of A, sorted.

1. Let $B[1\,..\,n]$ and $next[0\,..\,m-1]$ be new arrays.
2. For $j = 0$ to $m-1$:
   A. Set $next[j]$ to $less[j] + 1$.
3. For $i = 1$ to n:
   A. Set *key* to $A[i]$.
   B. Set *index* to $next[key]$.
   C. Set $B[index]$ to $A[i]$.
   D. Increment $next[key]$.
4. Return the B array.

Example.

	0	1	2	3	4	5	6
less	0	1	4	4	5	6	9
next	1	2	5	5	6	7	10

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

	0	1	2	3	4	5	6
next	1	2	5	5	7	7	10

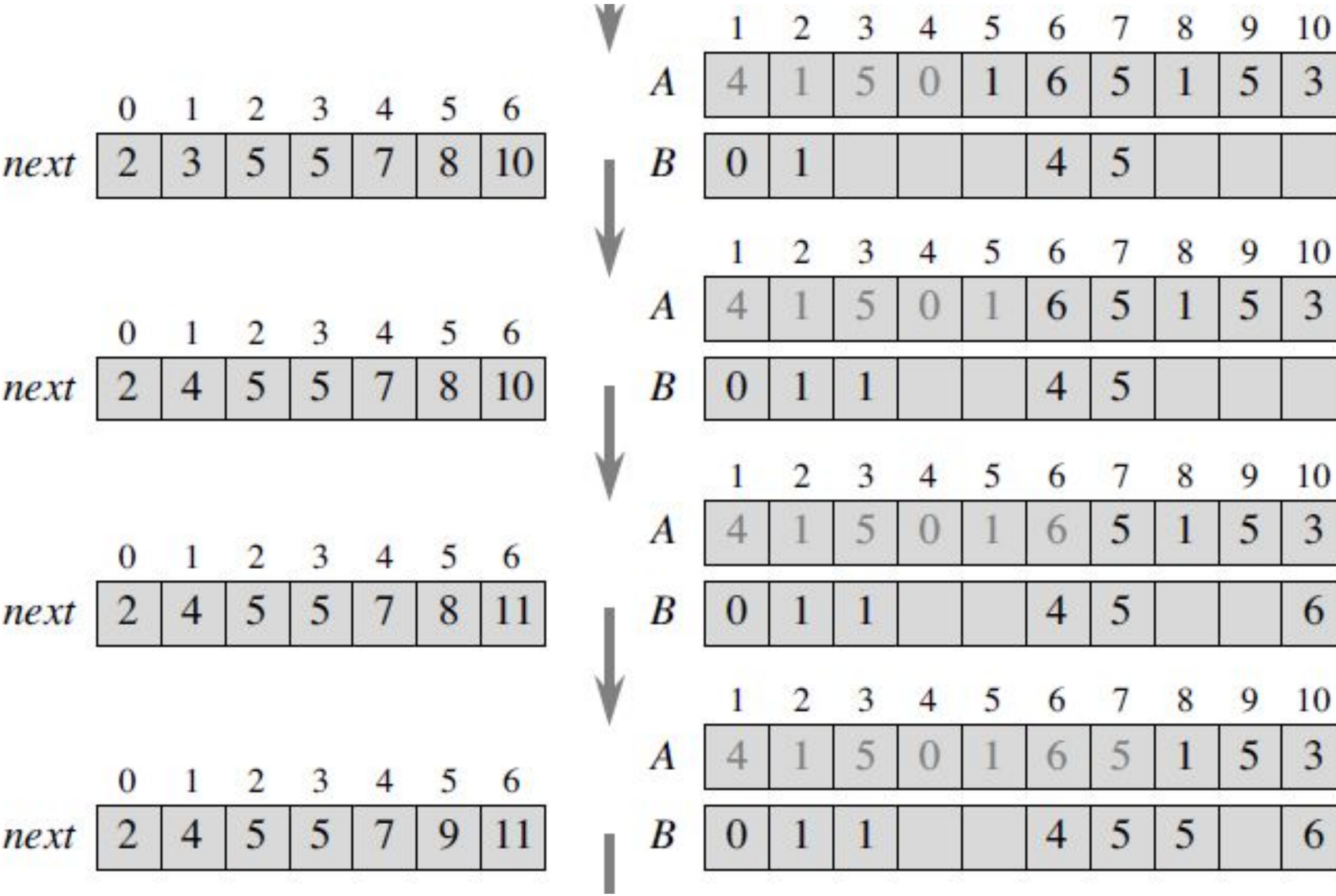
	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B						4				

	0	1	2	3	4	5	6
next	1	3	5	5	7	7	10

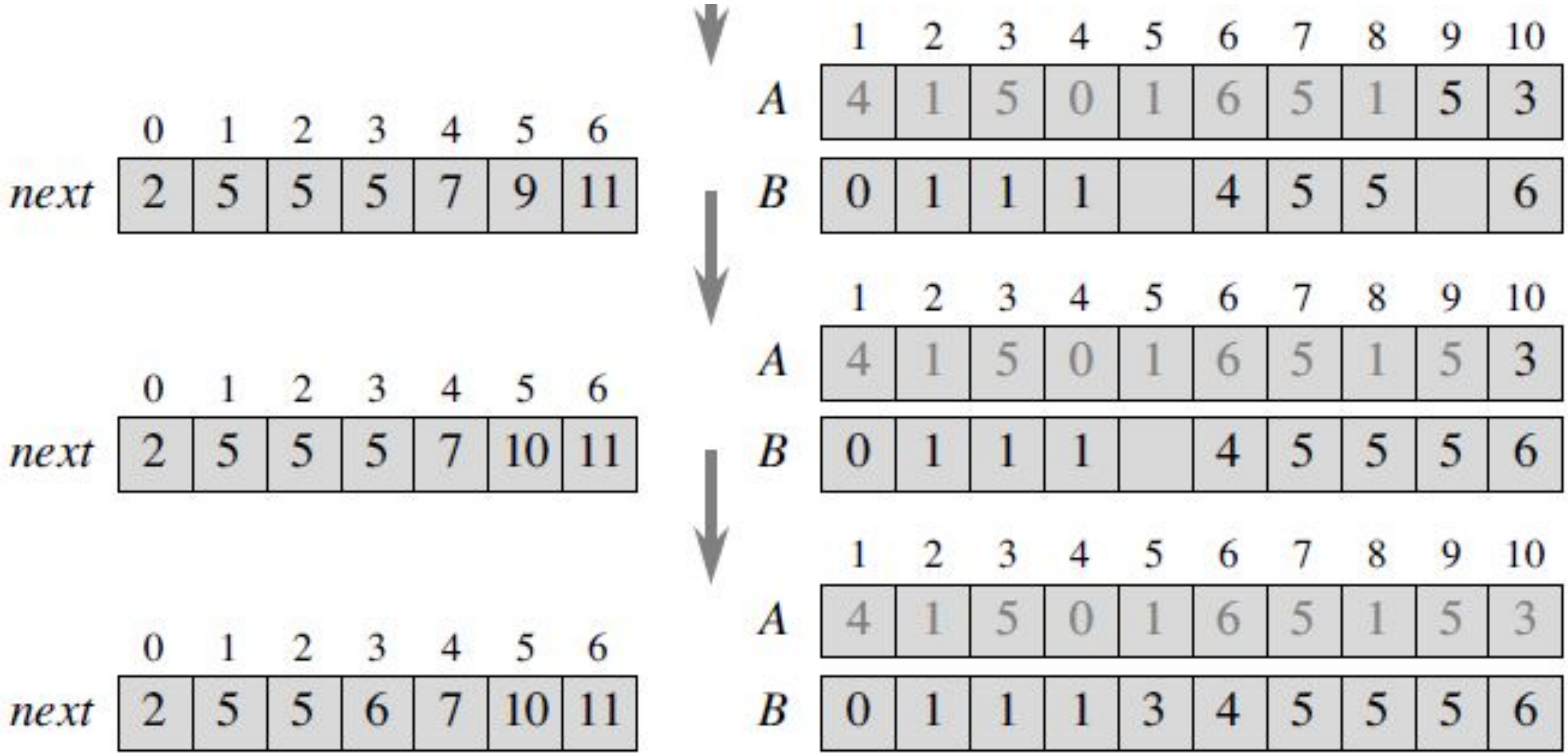
	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B		1				4				

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

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



1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
0 1 2 3 4 5 6
next 2 3 5 5 7 8 10
B 0 1 4 5
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
0 1 2 3 4 5 6
next 2 4 5 5 7 8 10
B 0 1 1 4 5
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
0 1 2 3 4 5 6
next 2 4 5 5 7 8 11
B 0 1 1 4 5 6
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
0 1 2 3 4 5 6
next 2 4 5 5 7 9 11
B 0 1 1 4 5 5 6



0 1 2 3 4 5 6
next 2 5 5 5 7 9 11
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 4 5 5 6
0 1 2 3 4 5 6
next 2 5 5 5 7 10 11
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 4 5 5 5 6
0 1 2 3 4 5 6
next 2 5 5 6 7 10 11
1 2 3 4 5 6 7 8 9 10
A 4 1 5 0 1 6 5 1 5 3
B 0 1 1 1 3 4 5 5 5 6

Procedure REARRANGE($A, less, n, m$)

Inputs:

- A: an array of integers in the range 0 to $m - 1$.
- *less*: the array returned by COUNT-KEYS-LESS.
- n: the number of elements in A.
- m: defines the range of the values in A.

Output: An array B containing the elements of A, sorted.

1. Let $B[1 \,..\, n]$ and $next[0 \,..\, m - 1]$ be new arrays.
2. For $j = 0$ to $m - 1$:
   A. Set $next[j]$ to $less[j] + 1$.
3. For $i = 1$ to n:
   A. Set *key* to $A[i]$.
   B. Set *index* to $next[key]$.
   C. Set $B[index]$ to $A[i]$.
   D. Increment $next[key]$.
4. Return the B array.

Beating the lower bound with counting sort

- The idea is that, as we go through the array A from start to end, next[j] gives the index in the array B of where the next element of A whose key is j should go.
- Recall from earlier that if *l* elements have sort keys less than *x*, then the *k* elements whose sort keys equal *x* should occupy positions *l+1* through *l+k*.

Beating the lower bound with counting sort

- The loop of step 2 sets up the array next so that, at first, next[j]= $l+1$, where $l= l+k$.
- The loop of step 3 goes through array A from start to end.

Beating the lower bound with counting sort

- For each element A[i], step 3A stores A[i] into key, step 3B computes index as the index in array B where A[i] should go, and step 3C moves A[i] into this position in B.
- Because the next element in array A that has the same sort key as A[i] (if there is one) should go into the next position of B, step 3D increments next[key].

Beating the lower bound with counting sort

How long does REARRANGE take?

- The loop of step 2 runs in $\Theta(m)$ time,
- and the loop of step 3 runs in $\Theta(n)$ time.
- Like COUNT-KEYSEQUAL, therefore, REARRANGE runs in $\Theta(m+n)$ time,
- which is $\Theta(n)$ if m is a constant.

Beating the lower bound with counting sort

Procedure COUNTING-SORT(A, n, m)

Inputs:

- A: an array of integers in the range 0 to $m - 1$.
- n: the number of elements in A.
- m: defines the range of the values in A.

Output: An array B containing the elements of A, sorted.

1. Call COUNT-KEYS-EQUAL(A, n, m), and assign its result to *equal*.
2. Call COUNT-KEYS-LESS(equal, m) and assign its result to *less*.
3. Call REARRANGE(A, less, n, m) and assign its result to B.
4. Return the B array.

Counting sort

Beating the lower bound with counting sort

The running times of

COUNT-KEYS-EQUAL $\Theta(m+n)$;

COUNTKEYS-LESS $\Theta(m)$;

REARRANGE $\Theta(m+n)$;

COUNTING-SORT runs in time $\Theta(m+n)$

or $\Theta(n)$ when m is a constant.

Beating the lower bound with counting sort

Counting sort beats the lower bound of $\Omega(n \lg n)$ for comparison sorting because it never compares sort keys against each other.

Instead, it uses sort keys to index into arrays, which it can do because the sort keys are small integers.

Beating the lower bound with counting sort

If the sort keys were real numbers with fractional parts, or they were character strings, then we **could not use counting sort**.

Beating the lower bound with counting sort

The running time is $\Theta(n)$ if m is a constant.

When would m be a constant?

One example would be if I were sorting exams by grade.

Beating the lower bound with counting sort

Sorting exams by grade.

The grades range from 0 to 10,

but the number of students varies.

Using counting sort to sort the exams of n students in $\Theta(n)$ time, since $m = 11$ (the range being sorted is 0 to $m-1$) is a constant.

Beating the lower bound with counting sort

Counting sort has another important property:

it is ***stable.***

The stable sort breaks ties between two elements with equal sort keys by placing first in the output array whichever element appears first in the input array.

Radix sort

Let’s you had to sort strings of characters of some fixed length.

For example, the confirmation code is XI7FS6.

=>36 values (26 letters plus 10 digits)

=>36^6 = 2,176,782,336 possible confirmation codes

Radix sort

36 characters => numeric from 0 to 35

The code for a digit   =>   the digit itself.

The codes for letters start at 10 for A and run through 35 for Z.

Radix sort

Simple example.

Confirmation code comprises two characters.

1) using the rightmost character as the sort key
2) using the leftmost character as the sort key

<F6; E5; R6; X6; X2; T5; F2; T3>

1) <X2; F2; T3; E5; T5; F6; R6; X6>
2) <E5; F2; F6; R6; T3; T5; X2; X6>

Radix sort

Simple example. BUT

1) using the leftmost character as the sort key
2) using the rightmost character as the sort key

<F6; E5; R6; X6; X2; T5; F2; T3>

1) <E5; F6; F2; R6; T5; T3; X6; X2>
2) <F2; X2; T3; E5; T5; F6; R6; X6>

It is incorrect. Why? => Using a stable sorting method

Radix sort

Example. <XI7FS6; PL4ZQ2; JI8FR9;XL8FQ6;PY2ZR5;KV7WS9; JL2ZV3; KI4WR2>

i	Resulting order
1	⟨PL4ZQ2, KI4WR2, JL2ZV3, PY2ZR5, XI7FS6, XL8FQ6, JI8FR9, KV7WS9⟩
2	⟨PL4ZQ2, XL8FQ6, KI4WR2, PY2ZR5, JI8FR9, XI7FS6, KV7WS9, JL2ZV3⟩
3	⟨XL8FQ6, JI8FR9, XI7FS6, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, JL2ZV3⟩
4	⟨PY2ZR5, JL2ZV3, KI4WR2, PL4ZQ2, XI7FS6, KV7WS9, XL8FQ6, JI8FR9⟩
5	⟨KI4WR2, XI7FS6, JI8FR9, JL2ZV3, PL4ZQ2, XL8FQ6, KV7WS9, PY2ZR5⟩
6	⟨JI8FR9, JL2ZV3, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, XI7FS6, XL8FQ6⟩

Radix sort

In the radix sort algorithm

the time to sort on **one** digit is <u>$\Theta(m+n)$.</u>

And the time to sort all **d** digits is <u>$\Theta(d(m+n))$.</u>