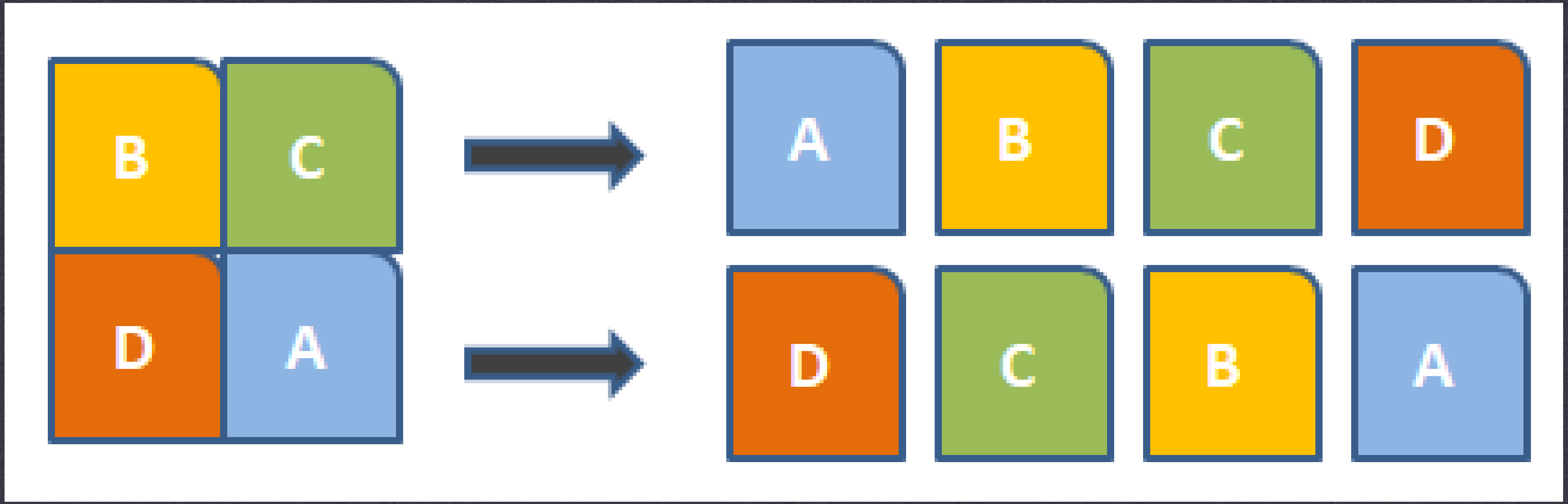
Sorting



Sorting

- One of the most fundamental problems in CS
  - Still many questions open!
- Given a list of objects we want to put **in order**
  - Alphabetical, word length, by score in the exam, sickness level,…
  - We assume we have a comparison
- How fast can we do it?

Speed is not the only concern

- How much extra memory do we use?
- Can we handle repeated numbers?
- Is information destroyed?
- Easy to implement?
- What computation model?
- …

Algorithm 1: selection sort

- Most intuitive algorithm
- Look for smallest value
  - Place it in first position
- Look for second smallest value
  - Place it second
- Etc

Example

9	5	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Look for smallest value

Example

9	5	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Look for smallest value

Example

2	5	10	8	12	11	14	9	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Place first place, now look for second smallest

Example

2	5	10	8	12	11	14	9	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Already in second place we do nothing

Example

2	5	8	10	12	11	14	9	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Third one to third position

Example

2	5	8	9	12	11	14	10	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- And so on…

Runtime

- Two nested loops
  - Quadratic runtime!
- Let's prove it!
  - Outer loop n times
  - Inner loop n-j
  - Constant number of operations inside

TOTAL?

Big O bound

- Two nested loops
  - Quadratic runtime!
- Let's prove it!
  - Outer loop **at most** n times
  - Inner loop n-j **(at most n times)**
  - Constant number of operations inside

$O(N^2)$

Algorithm 2: insertionSort

9	5	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Assume the first i elements have been sorted
- Insert the i+1 in its proper place
- Start with i=1 and stop when i=n

Algorithm 2: insertionSort

9	5	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- The first position is always sorted with itself (progress!)

Algorithm 2: insertionSort

9	5	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Second position is smaller: we swap

Algorithm 2: insertionSort

5	9	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Second position is smaller: we swap

Algorithm 2: insertionSort

5	9	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Third position is largest: nothing to do

Algorithm 2: insertionSort

5	9	10	8	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Fourth position is smaller than third: we swap with previous position

Algorithm 2: insertionSort

5	9	8	10	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- 8 is still too big, we need another swap

Algorithm 2: insertionSort

5	8	9	10	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- 8 is still finally in place. Let's look for next number

Algorithm 2: insertionSort

5	8	9	10	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Already sorted, nothing to do

Algorithm 2: insertionSort

5	8	9	10	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

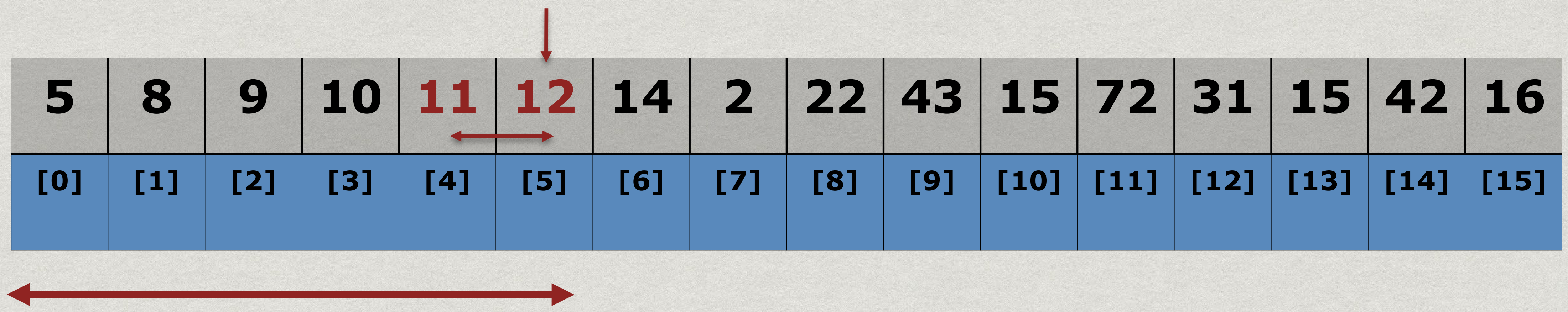
- Next position almost sorted

Algorithm 2: insertionSort

5	8	9	10	12	11	14	2	22	43	15	72	31	15	42	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

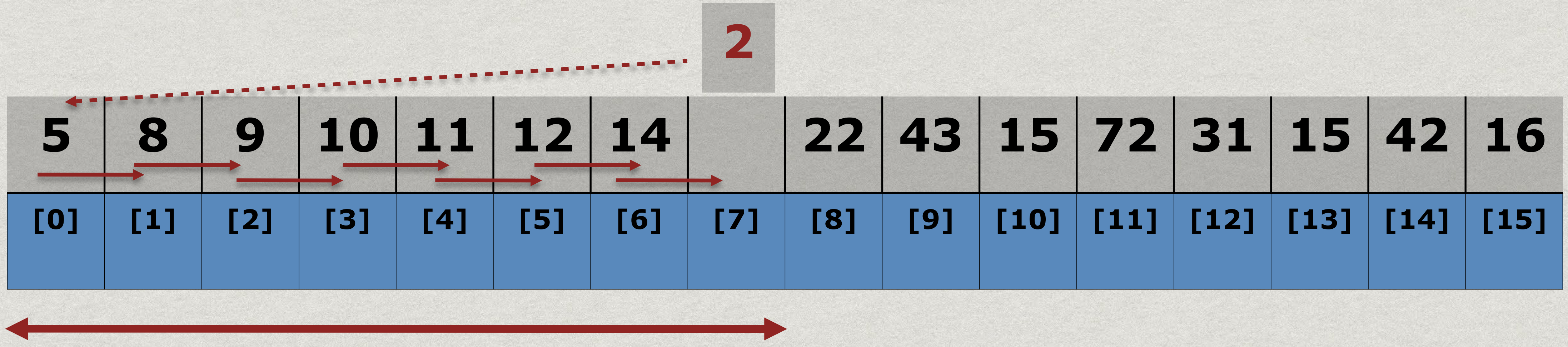
- Swap with previous number

Algorithm 2: insertionSort



- Done! Is it always this easy?

Algorithm 2: insertionSort



- No! The j-th position may travel j-1 positions **in worst case**

Code?

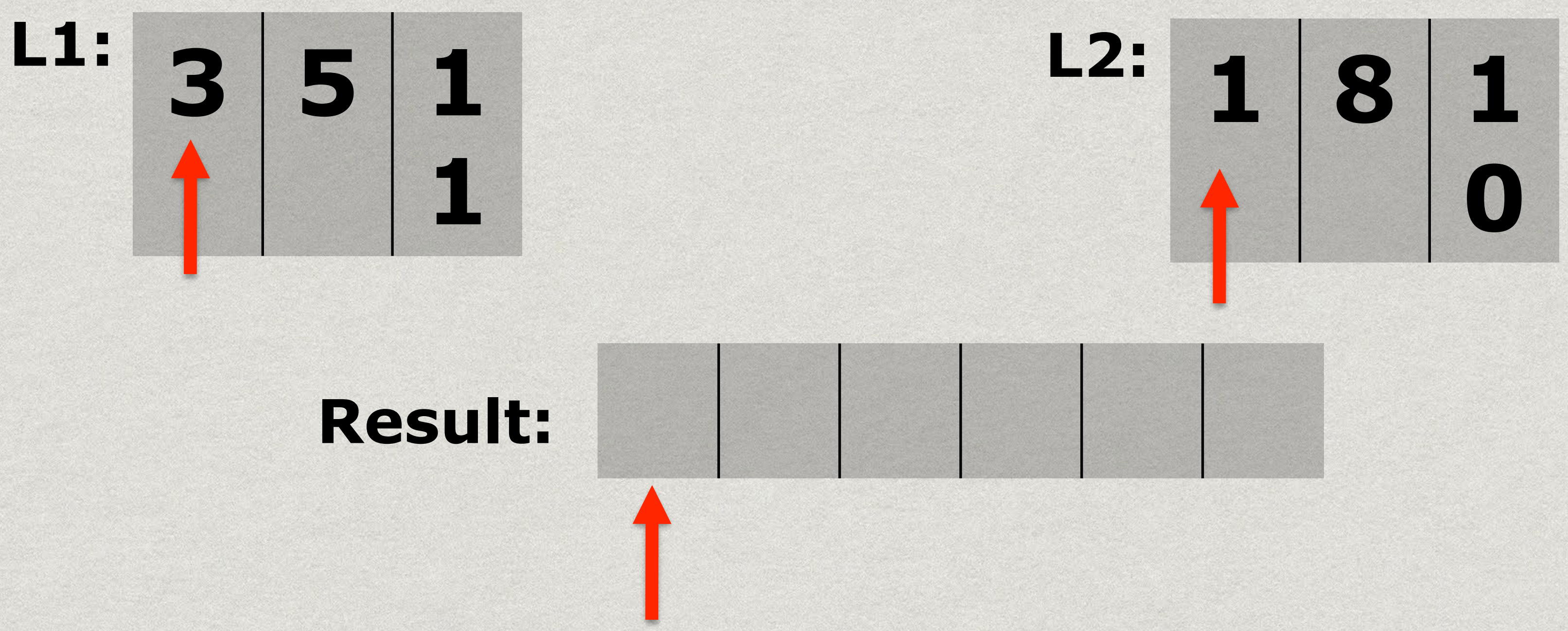
```
VOID INSERTIONSORT(INT ARR[], INT N)
{
    INT I, KEY, J;
    FOR (I = 1; I < N; I++) {
        KEY = ARR[I];
        J = I - 1;

        /* MOVE ELEMENTS OF ARR[0..I-1], THAT ARE
          GREATER THAN KEY, TO ONE POSITION AHEAD
          OF THEIR CURRENT POSITION */
        WHILE (J >= 0 && ARR[J] > KEY) {
            ARR[J + 1] = ARR[J];
            J = J - 1;
        }
        ARR[J + 1] = KEY;
    }
}
```

Introducing Mergesort

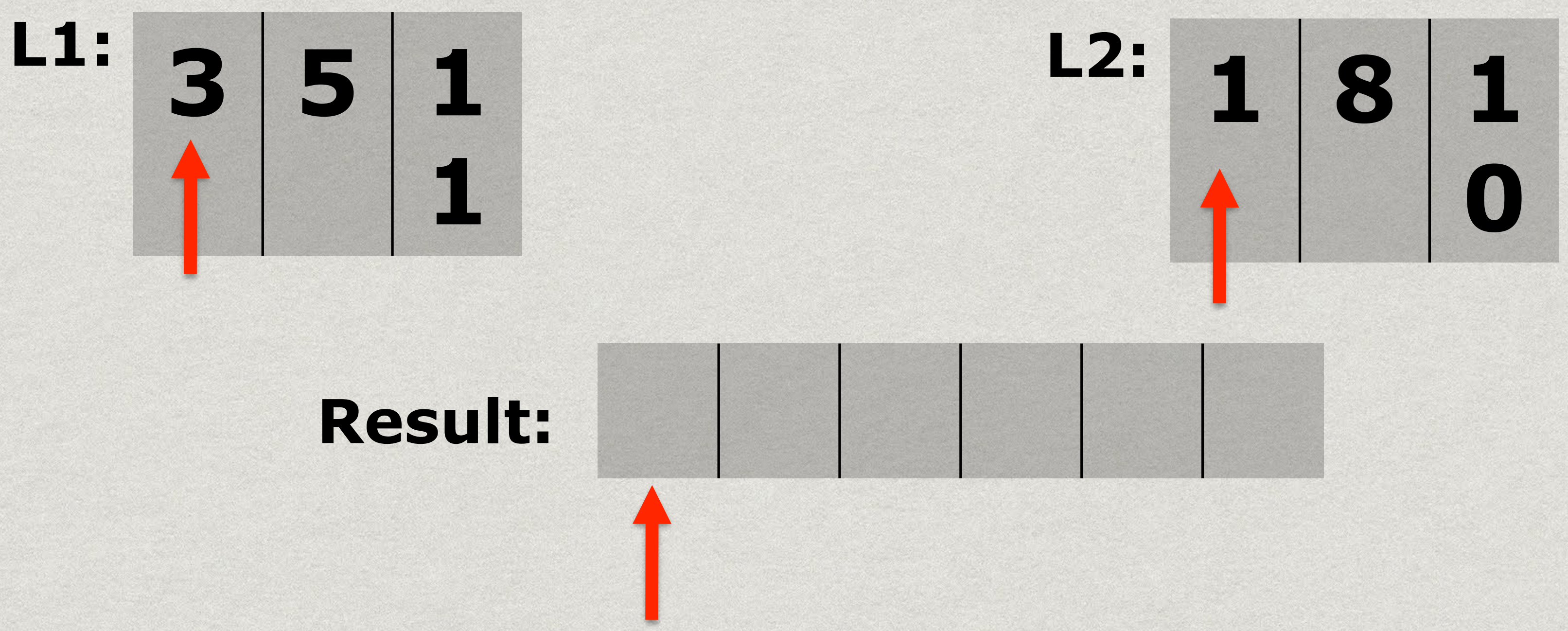
- Partition the array of n elements into two arrays of size n/2
  - Recursively sort them
  - Merge the two solutions into one

Combining two sorted lists



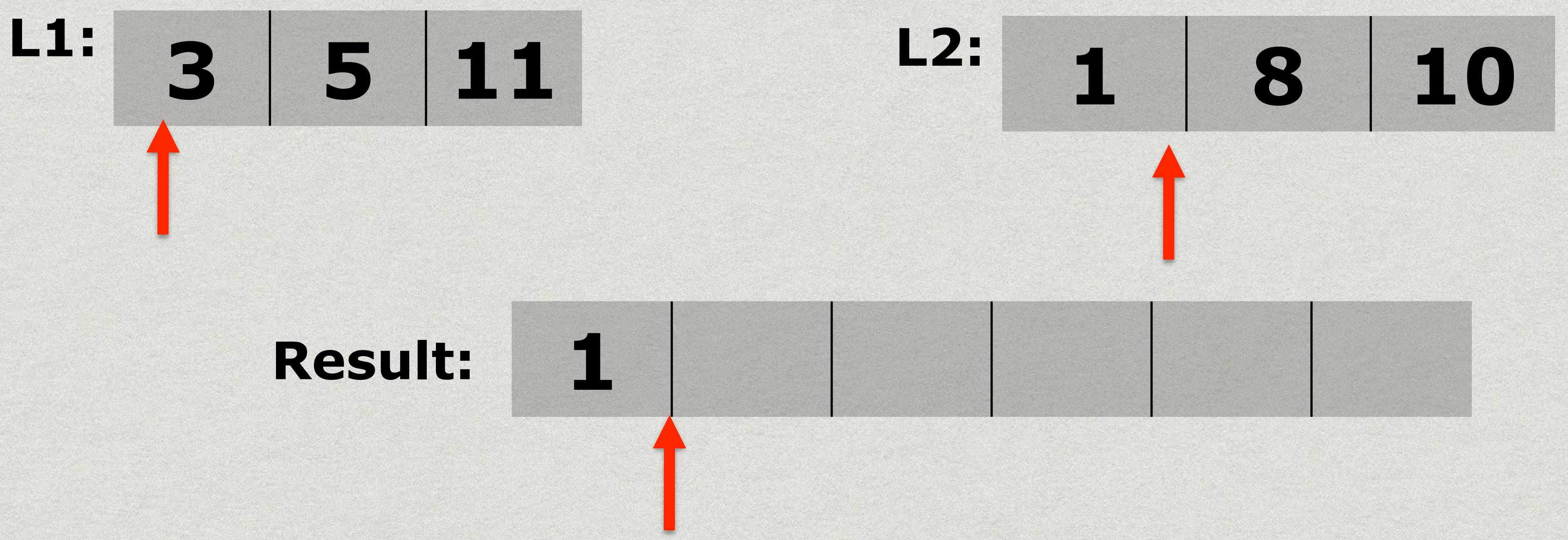
- Keep 3 pointers
  - Current positions in the three arrays
  - Start at rightmost positions

Combining two sorted lists



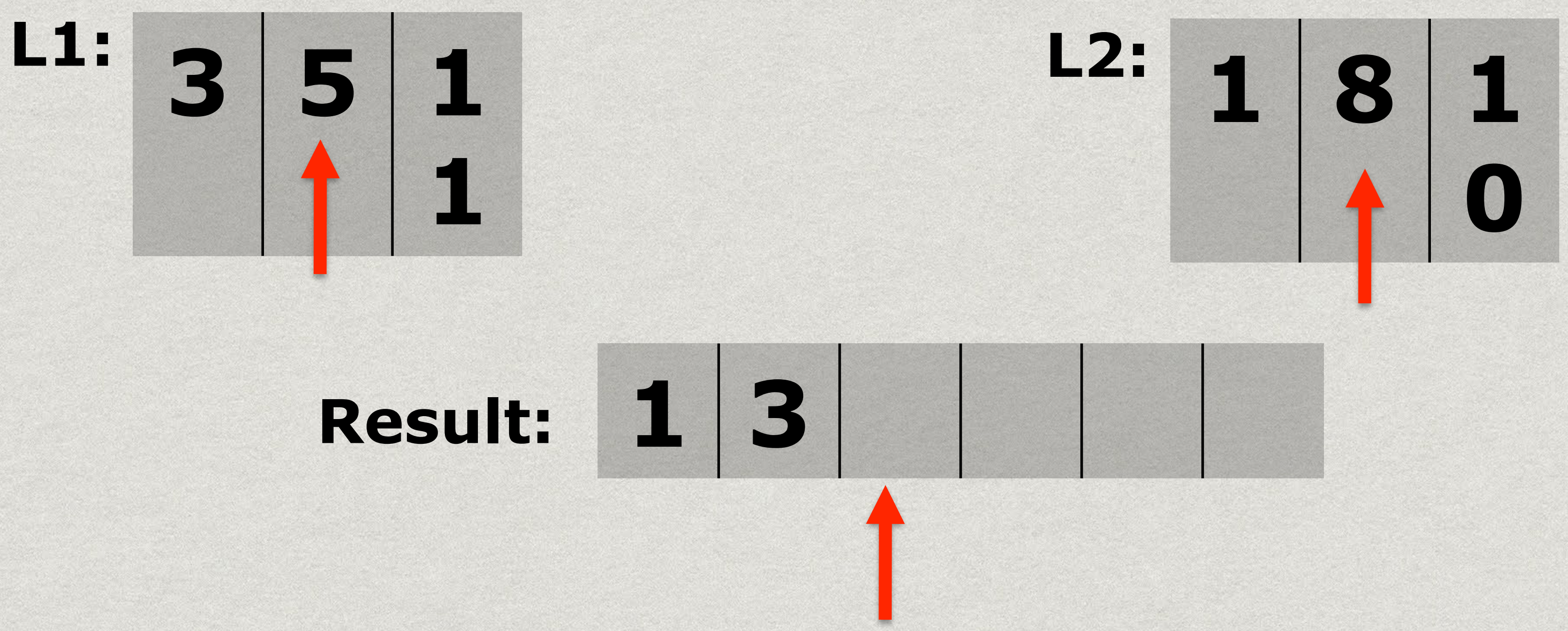
- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



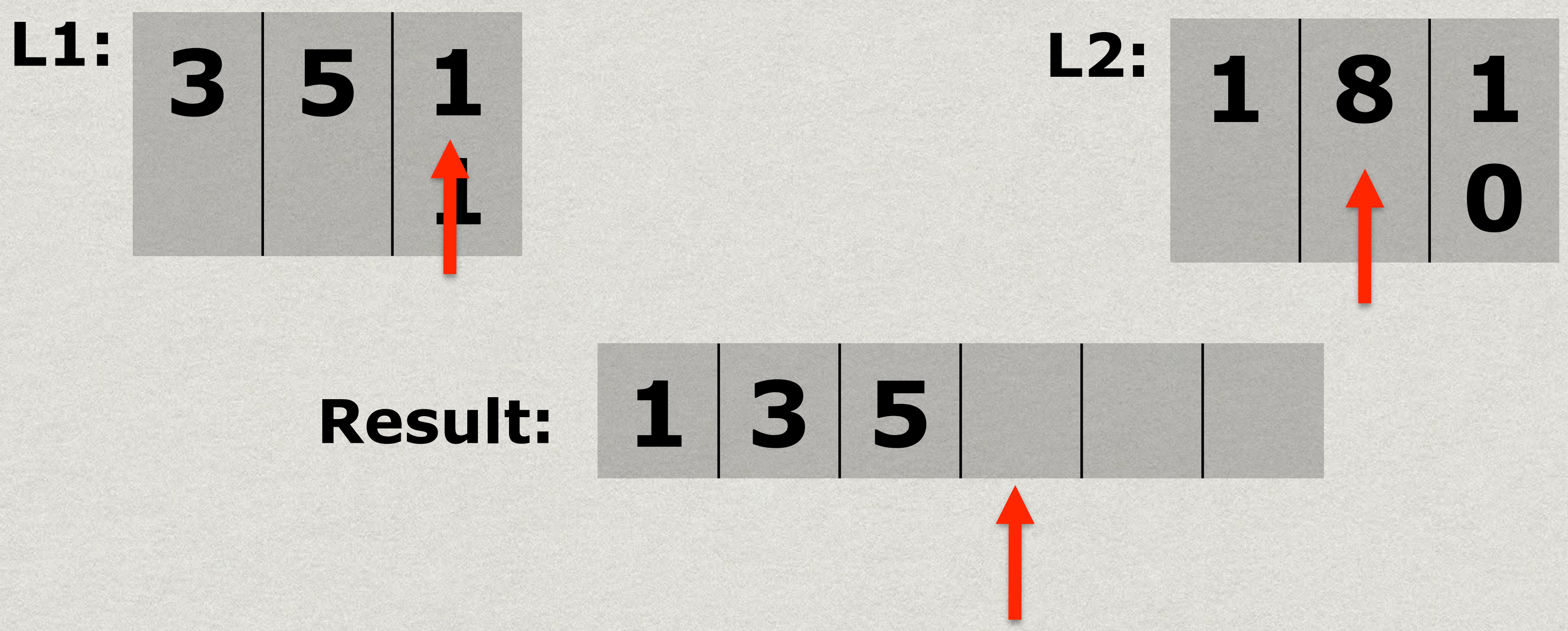
- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



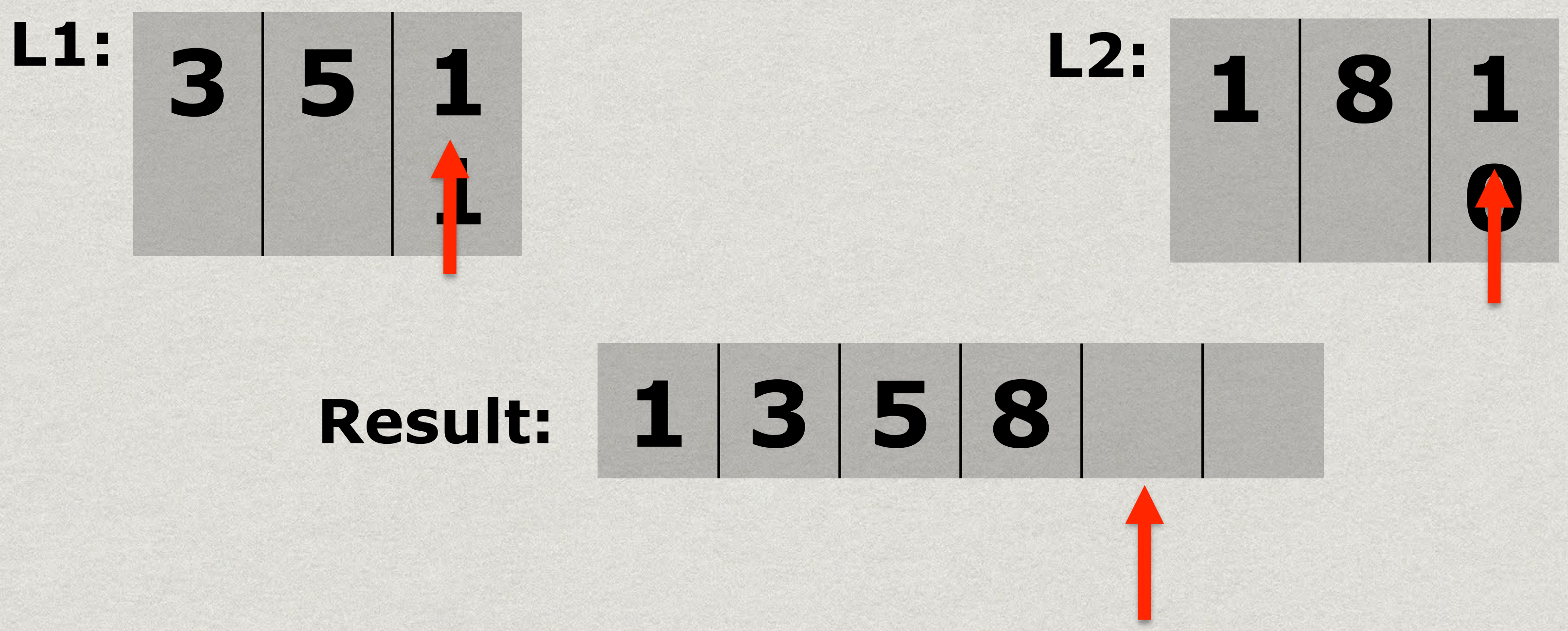
- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



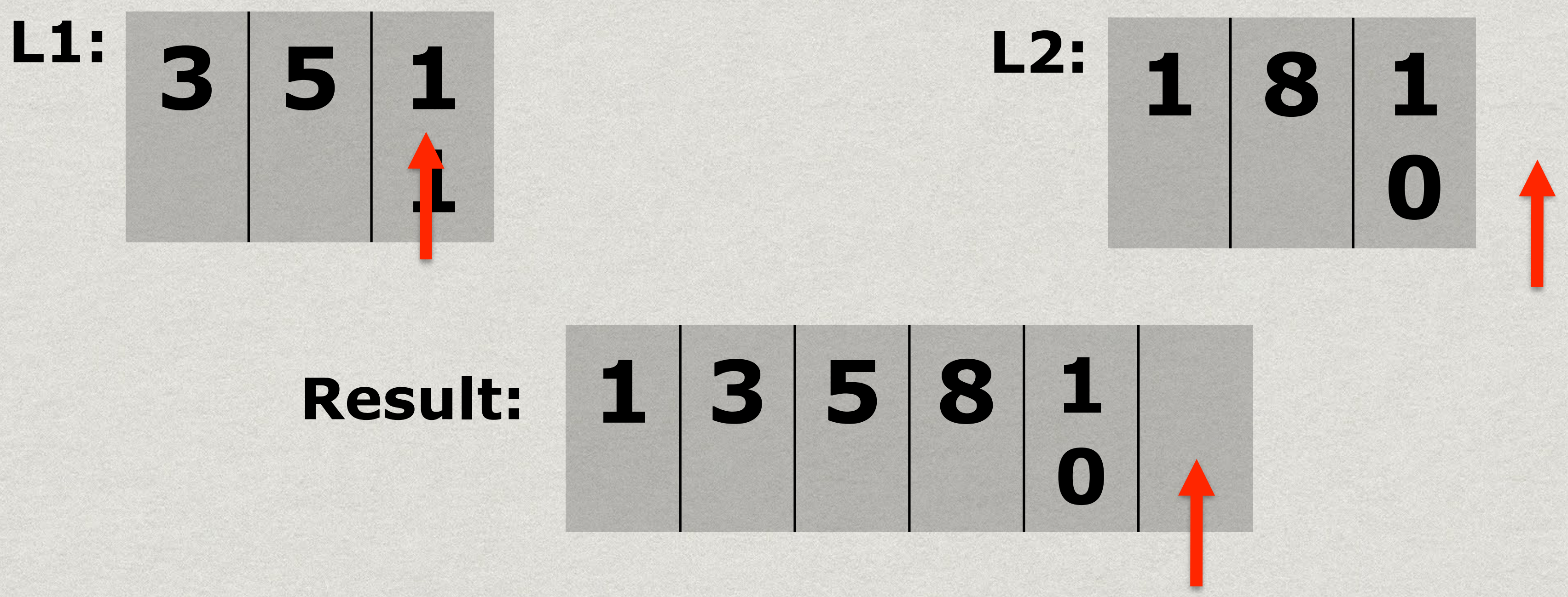
- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



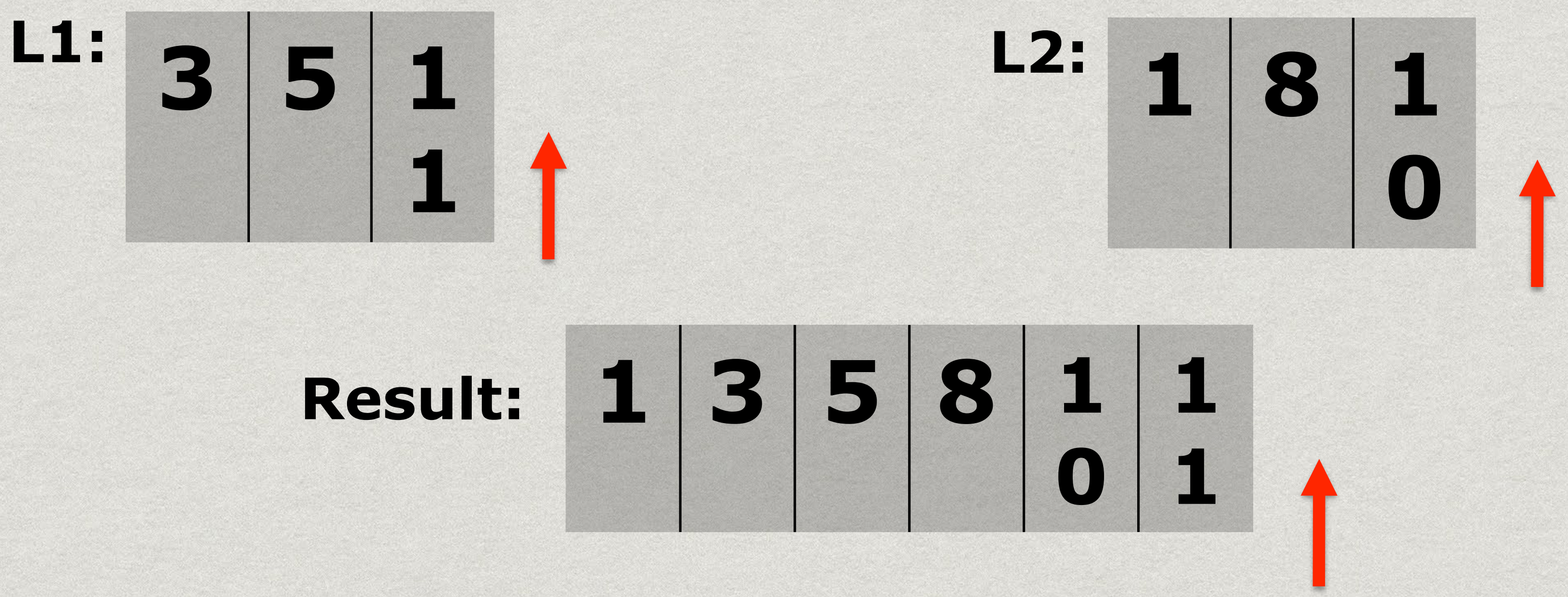
- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Combining two sorted lists



- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest

Merge Sort Full Example

99	6	86	15	58	35	86	4	0

Merge Sort Full Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

Merge Sort Full Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

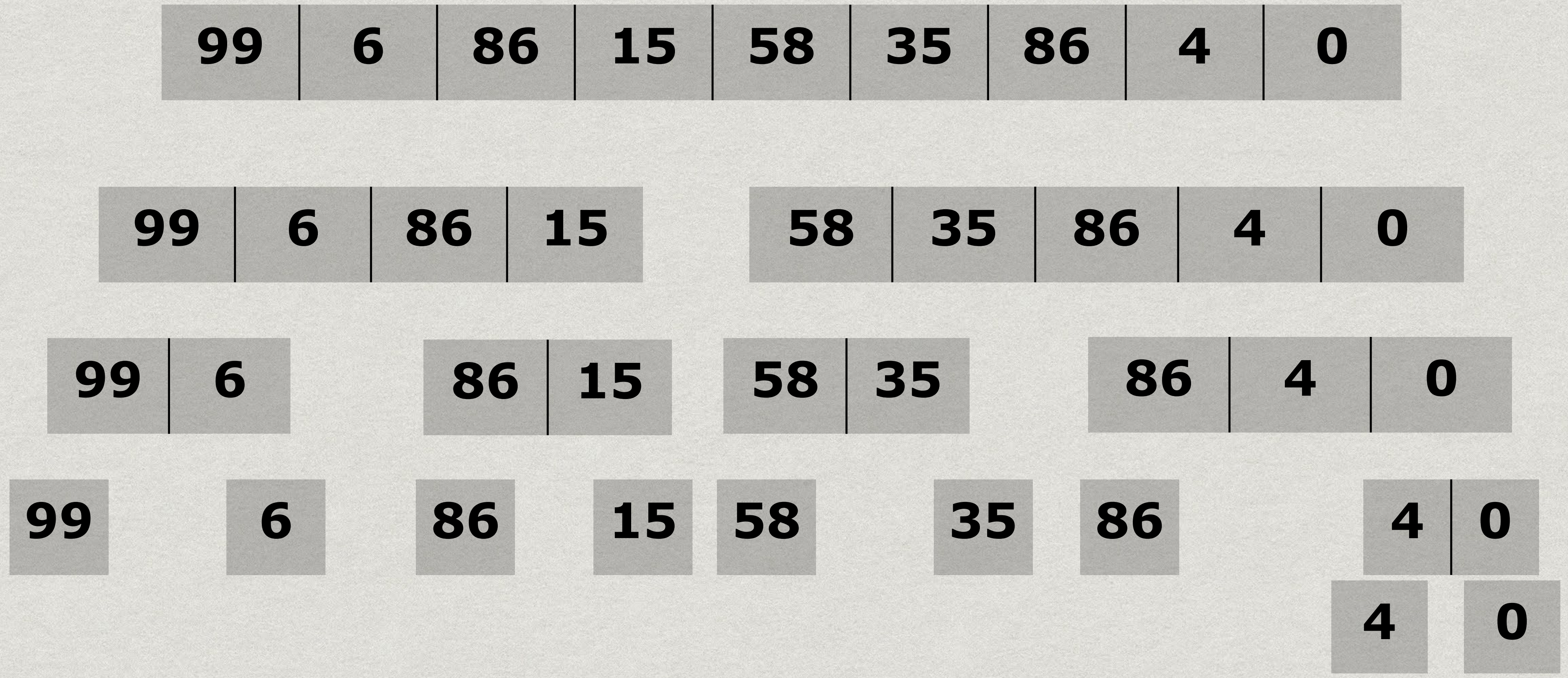
99	6

86	15

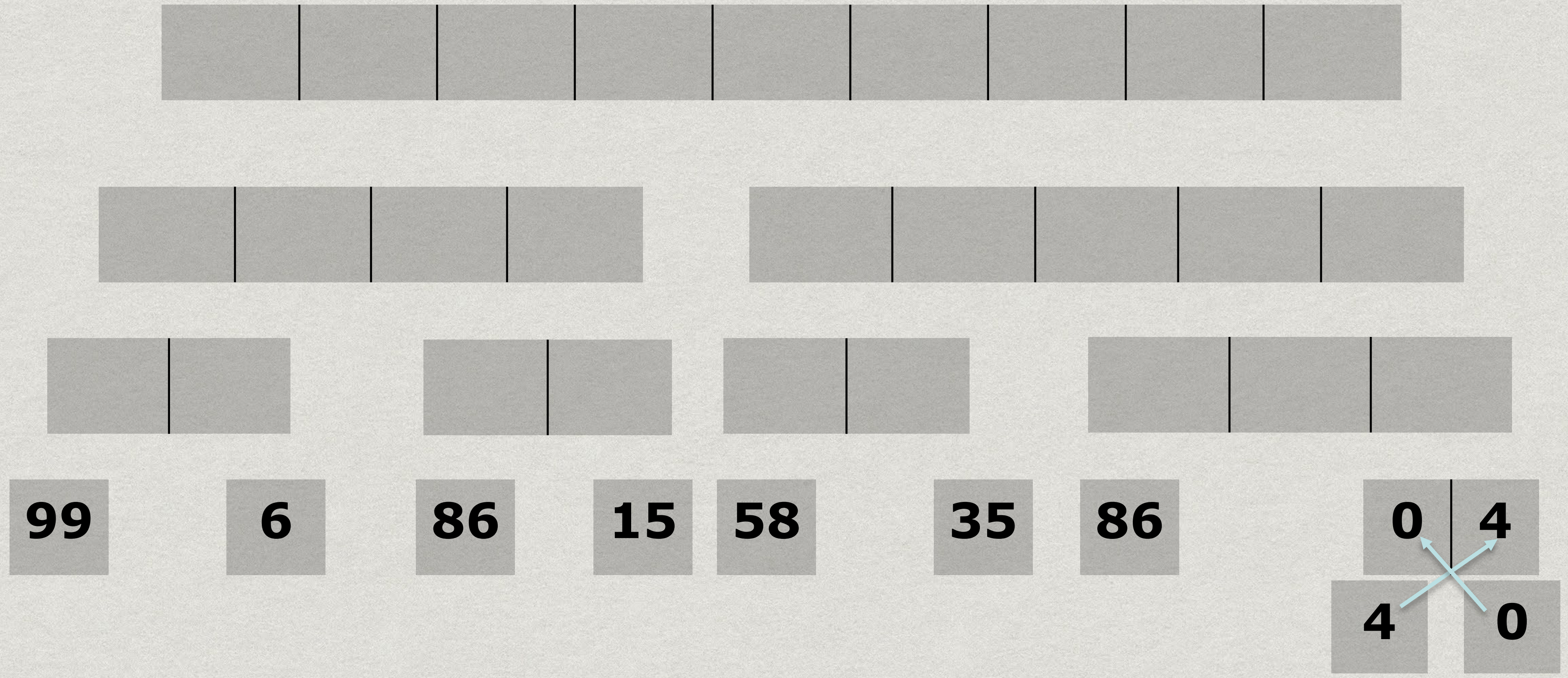
58	35

86	4	0

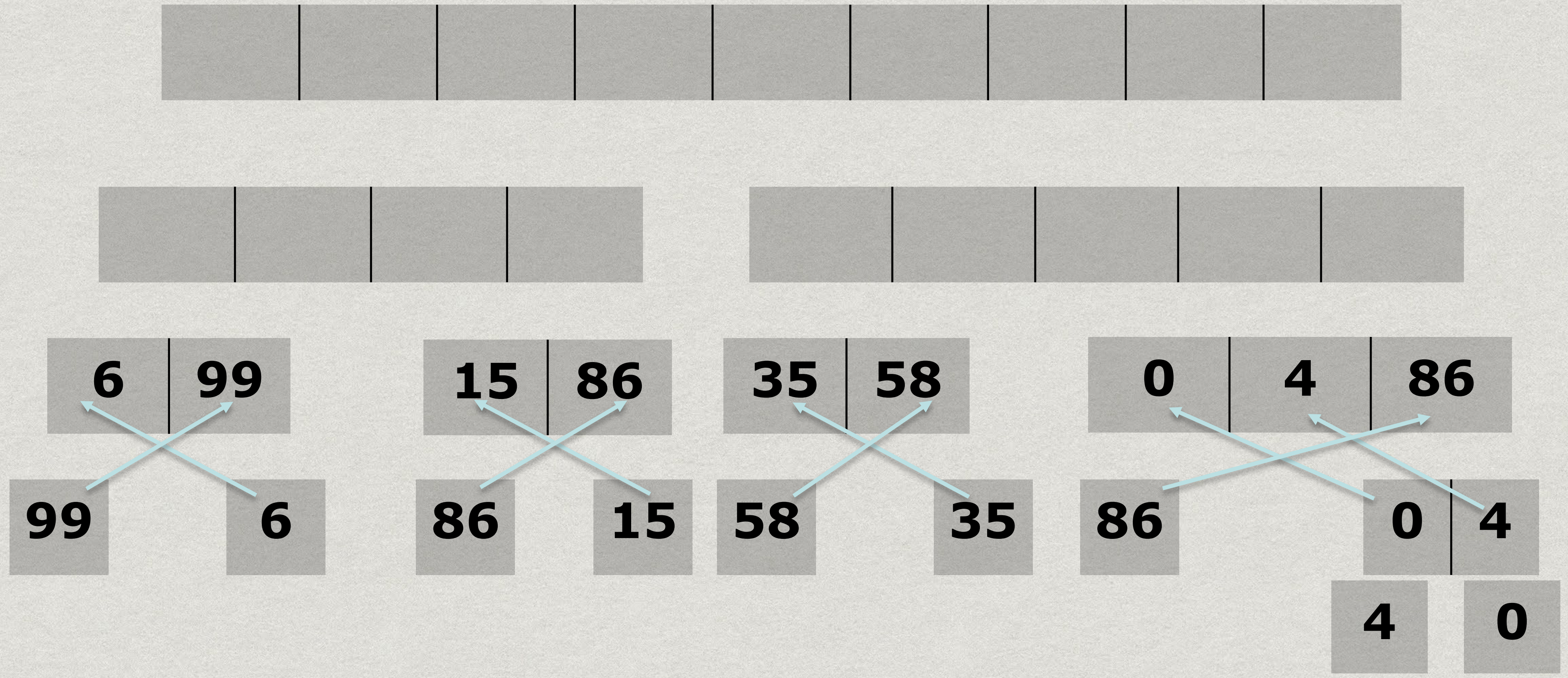
Merge Sort Full Example



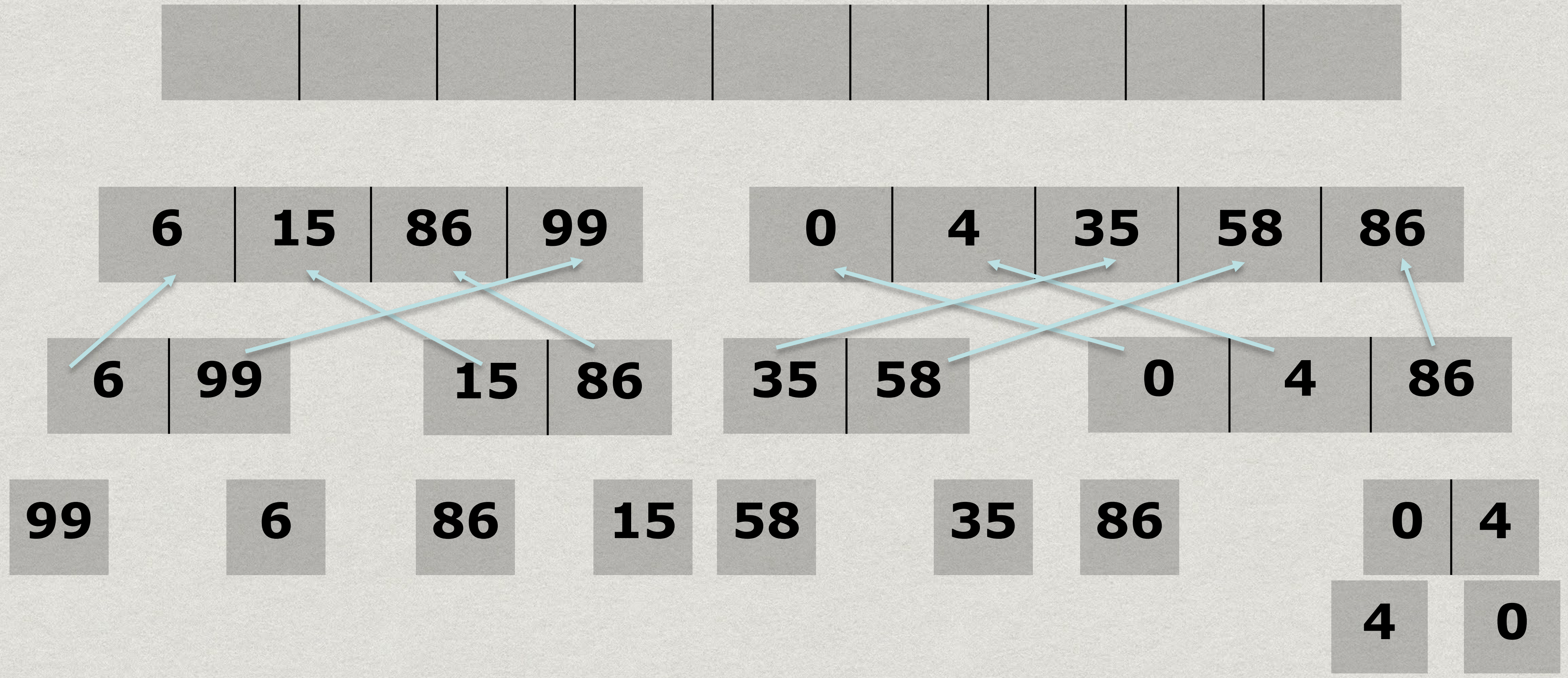
Building pieces upwards



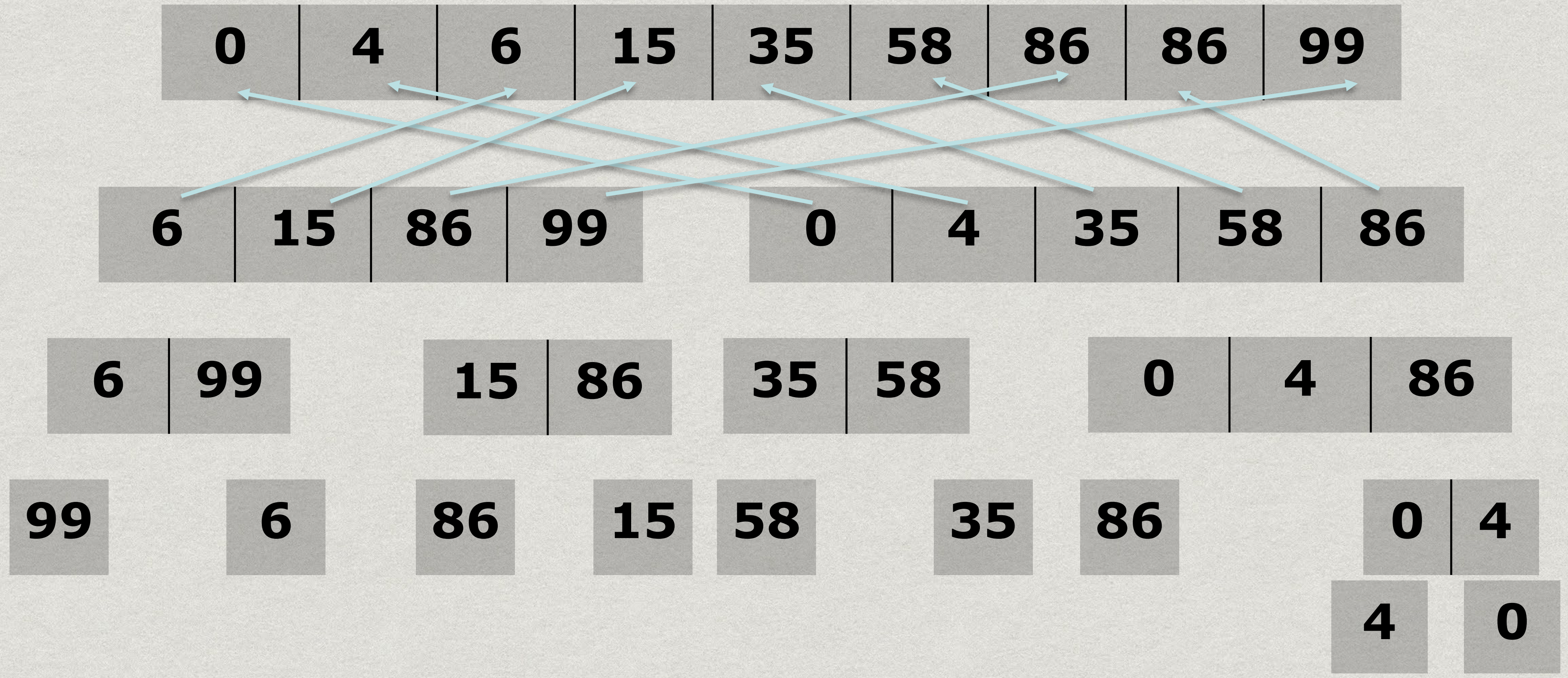
Building pieces upwards



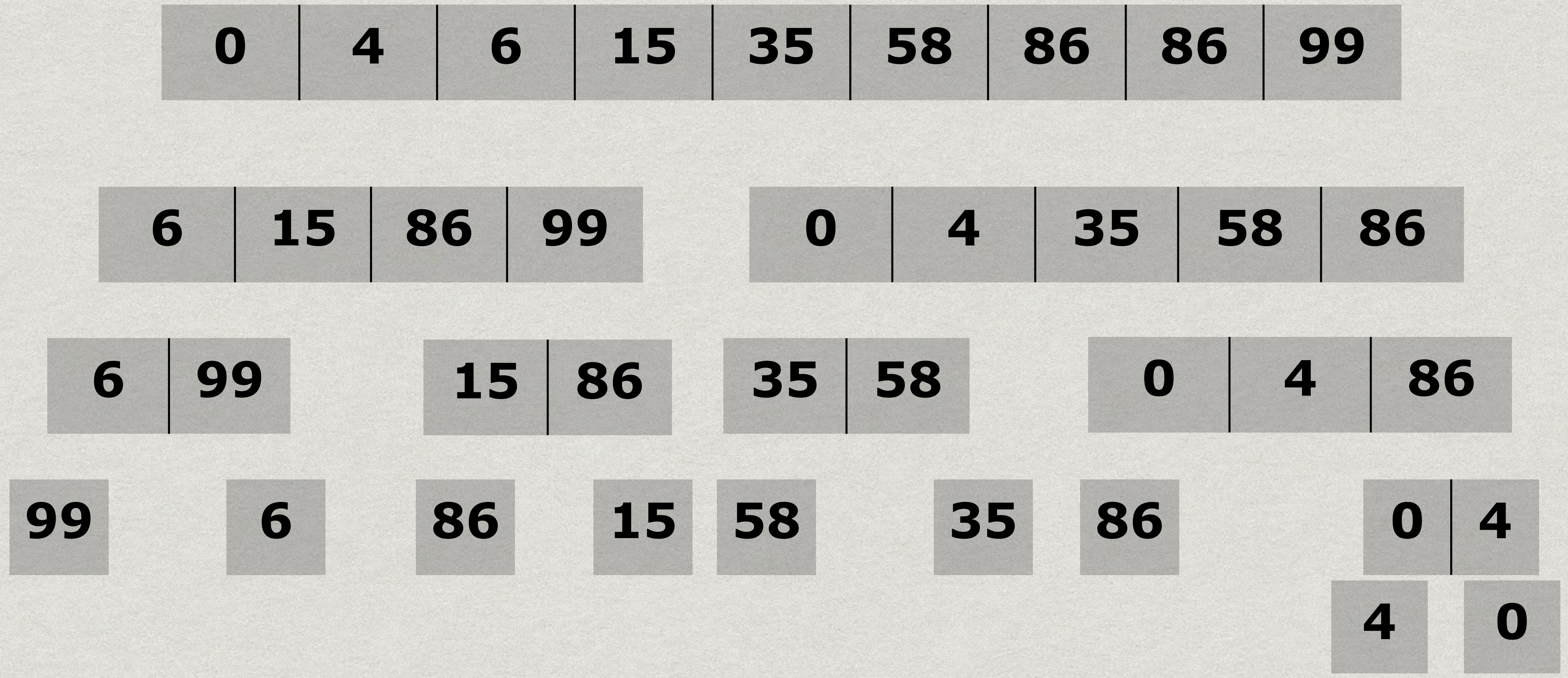
Building pieces upwards



Building pieces upwards

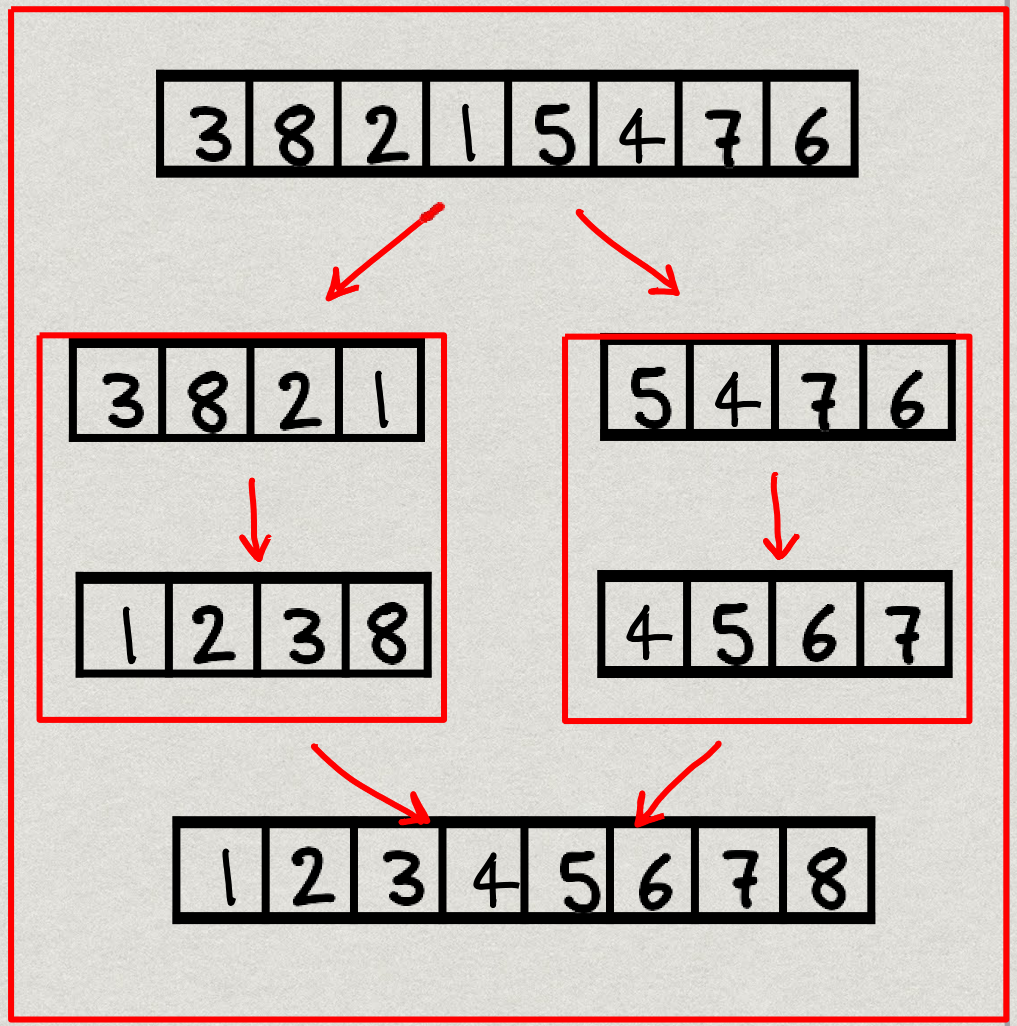


Building pieces upwards



Runtime?

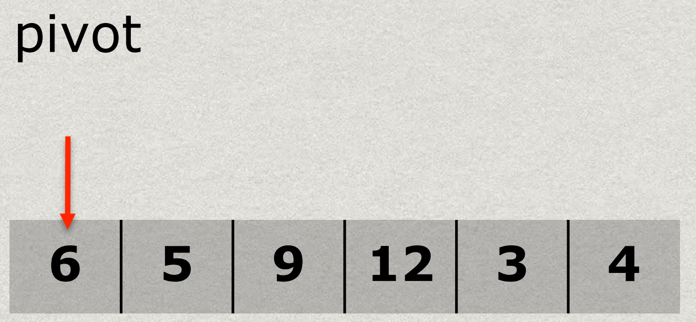
Mergesort time for n elements:



Quicksort

- Very similar in than MergeSort
  - **Divide and Conquer** strategy
  - Worse from a theoretical standpoint
  - Faster in practice
  - Does not need extra space

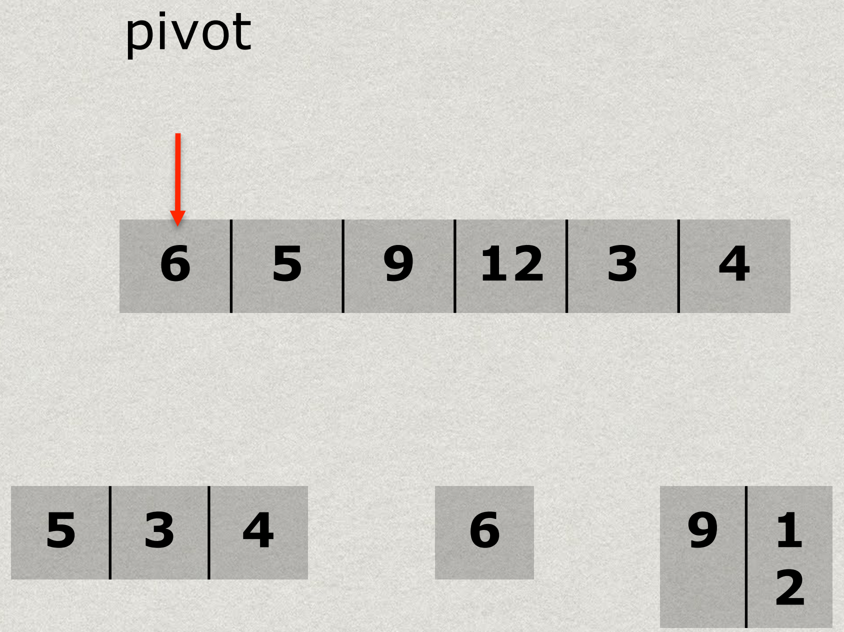
Quicksort



- Pick an element of the array (the **pivot**)
    - Split array into smaller and larger than pivot

NAIVE CHOICE:
A[0]

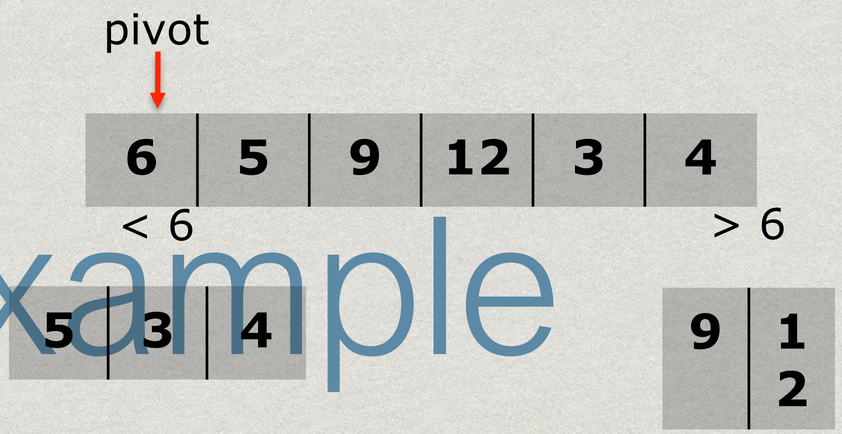
Quicksort



- Pick an element of the array (the **pivot**)
  - Split array into smaller and larger than pivot
  - Recursively sort both arrays

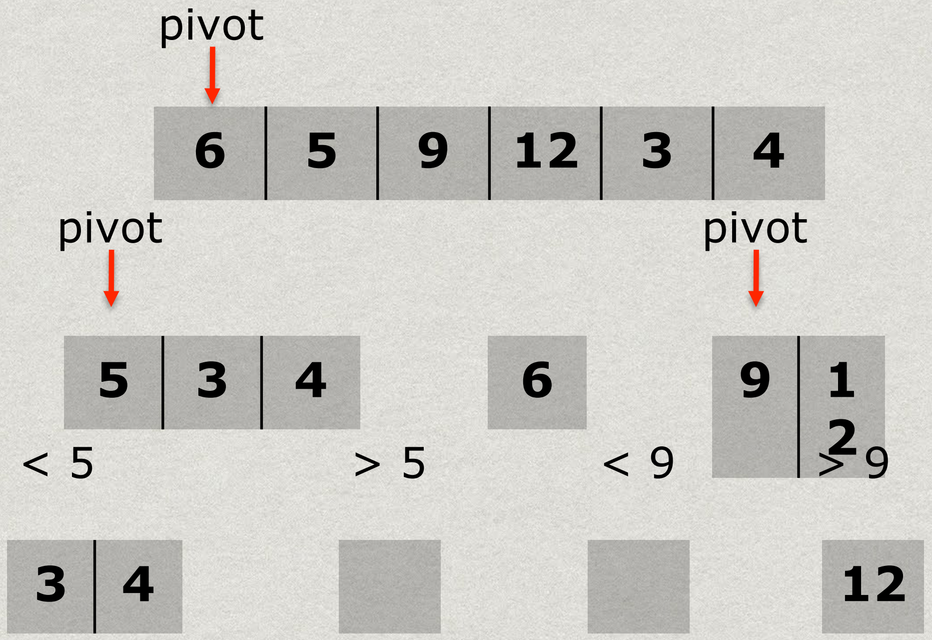
NAIVE CHOICE: A[0]

Execution example



- Pick an element of the array (the **pivot**)
  - Split array into smaller and larger than pivot
  - Recursively sort both arrays

Execution example

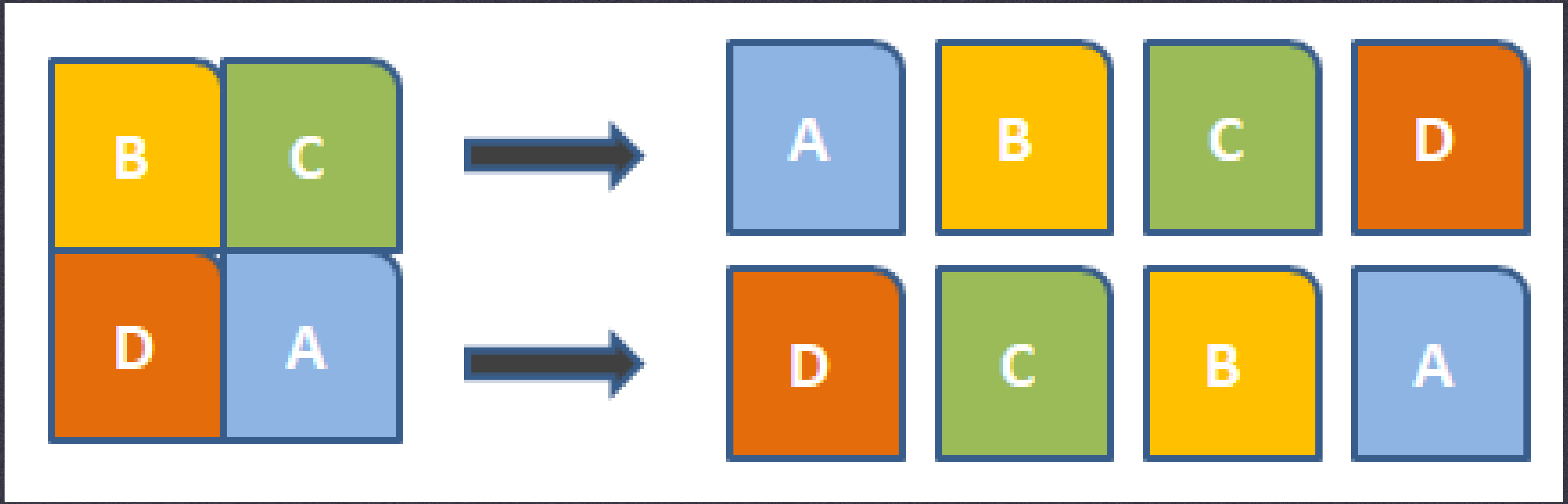


- Pick an element of the array (the **pivot**)
  - Split array into smaller and larger than pivot
  - Recursively sort both arrays

Runtime?

- Worst case?
  - Already sorted input!
  - $O(n^2)$ runtime
- Easy solution?
  - Pick a pivot **at random**
  - Still $O(n^2)$ worst case runtime

Part 2: Sorting in Linear Time



CountingSort

- Let’s spice things up
- Can we do it in a different way?

  Not based on usual comparison

- Assume input has limited range

  $0<A[i]<k$ for all values of I

- Can you make an algorithm that uses this property?

Countingsort: phase 1

9	5	10	8	3	6	5	2	5	2	1	3	5	2	6	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	0	0	0	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

10 IN THIS EXAMPLE

- Scan array, find largest value k
- Make array of size k+1, all entries zero
- Scan array again, each time increasing count

Countingsort: phase 1

9	5	10	8	3	6	5	2	5	2	1	3	5	2	6	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

10 IN THIS EXAMPLE

- Scan array, find largest value k
- Make array of size k+1, all entries zero
- Scan array again, each time increasing count

Countingsort: phase 2

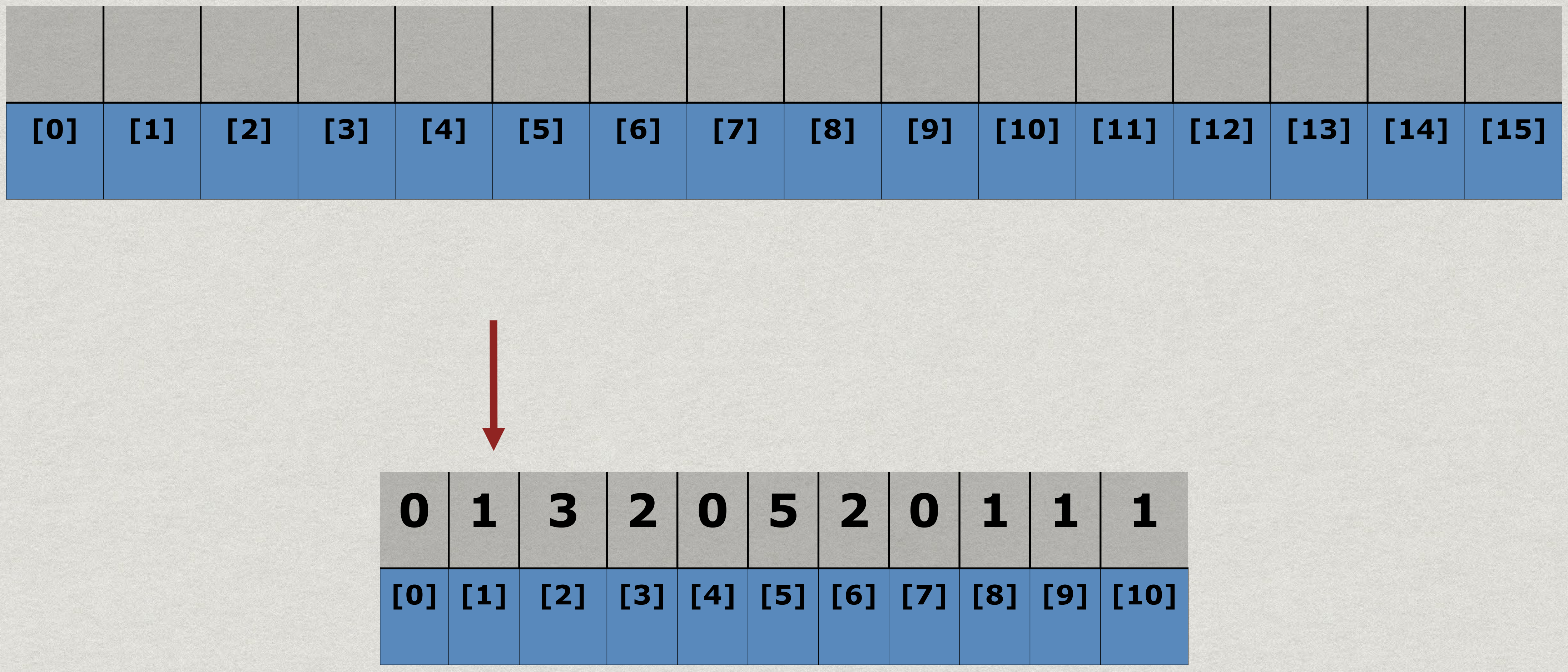
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

NOTHING TO DO

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

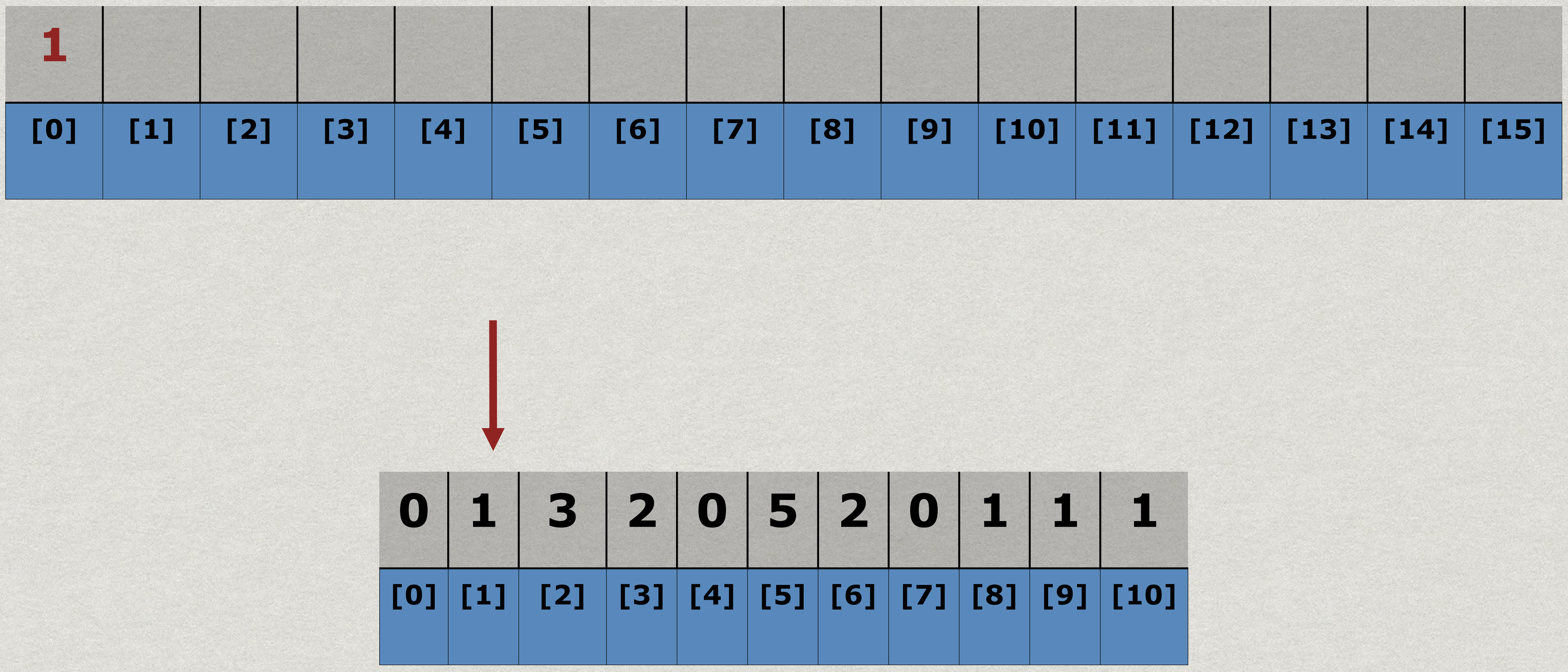
- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2



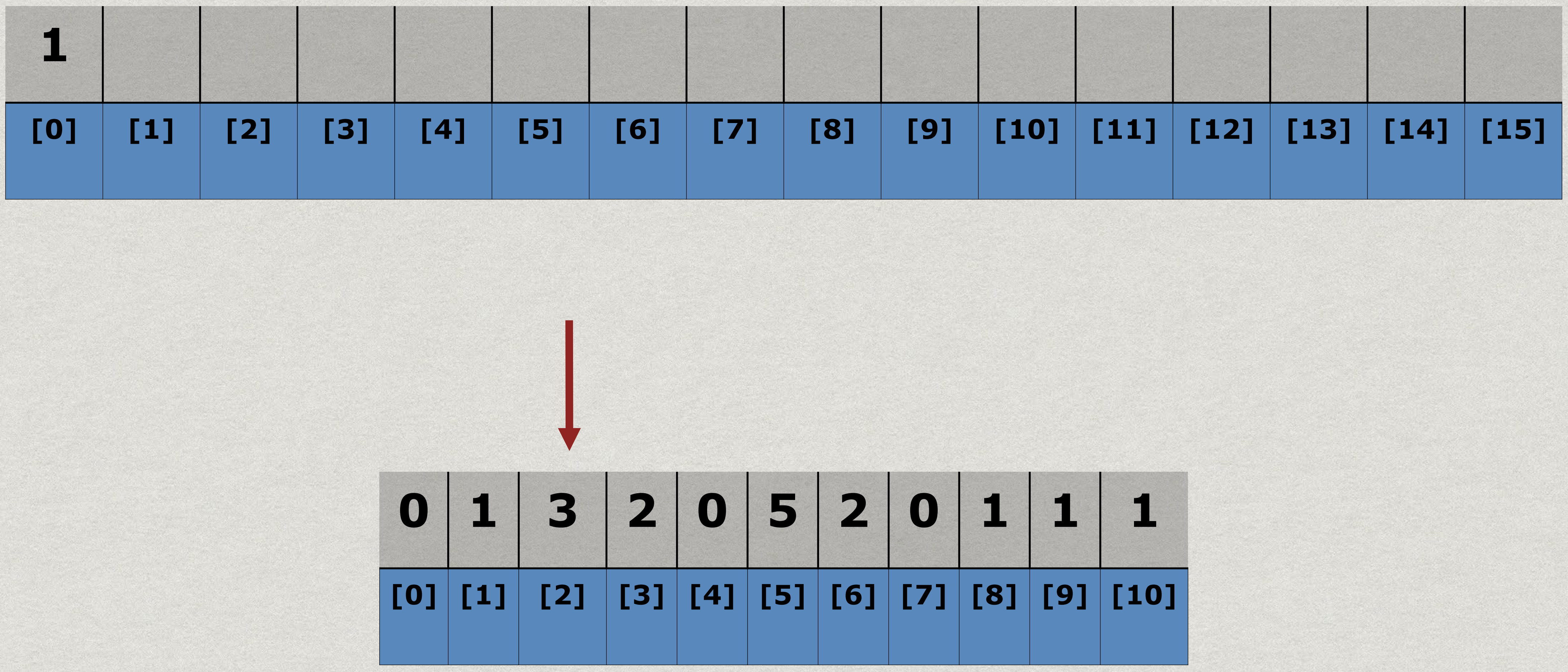
- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2



- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2



- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2												
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2												
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
1	2	2	2	3	3										

↓

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	1	3	2	0	5	2	0	1	1	1

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2	3	3										
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

↓

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2	3	3										
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2	3	3	5	5	5	5	5					
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

↑

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2	3	3	5	5	5	5	5	6	6			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Countingsort: phase 2

1	2	2	2	3	3	5	5	5	5	5	6	6			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

ETC

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution

Runtime?

Phase 1:

Scan input array, find max O(N)

Create array of size k O(1)

Make all entries zero O(K)

Scan input array, increase count at each step O(N)

Runtime?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
1	2	2	2	3	3	5	5	5	5	5	6	6			

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	1	3	2	0	5	2	0	1	1	1

Phase 2:

For all entries of count array K ITERATIONS

TOTAL TIME O(N+K)

n: number of elements

1073	284	5	8261	2714	382

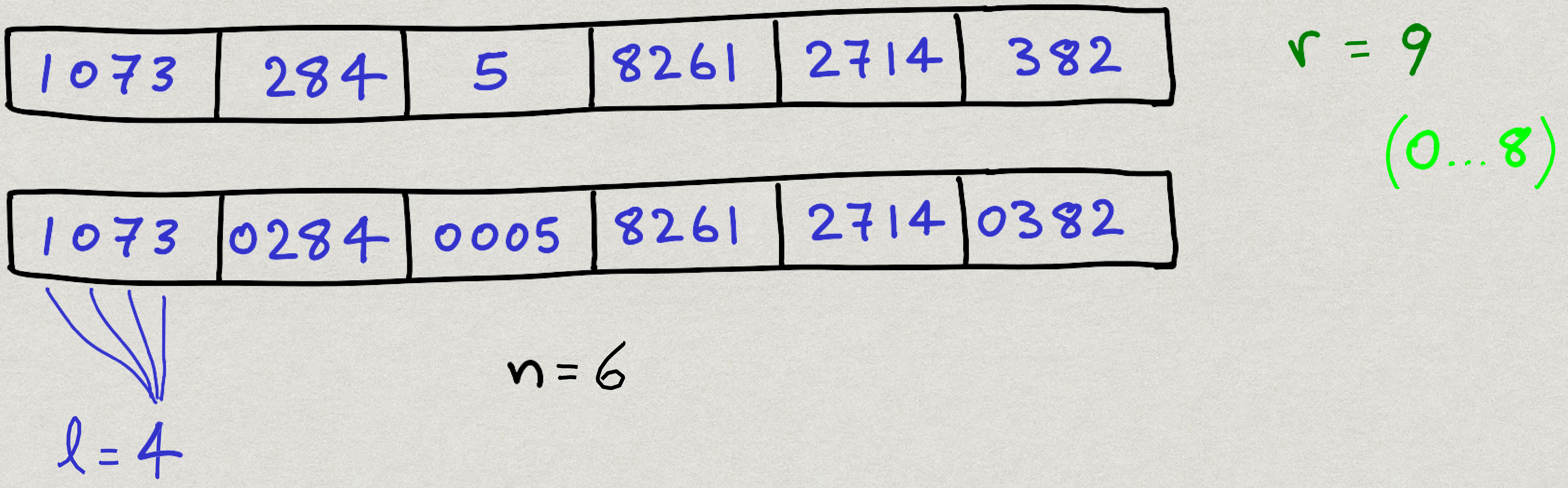
$n = 6$

RADIX SORT

n : number of elements

l : (max) length of each element

r : radix (#symbols available at each digit) e.g., binary, decimal, hex



RADIX SORT

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

$n = 7$

$\ell = 3$

$r = 10$

RADIX SORT

uses the least significant digit.
↓
usually

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

RADIX SORT

uses the <u>least</u> significant digit.

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

RADIX SORT

uses the **least** significant digit.

don't cross the streams

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

RADIX SORT

uses the least significant digit.

use stable counting sort → $\Theta(n+r)$

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

WAIT A SECOND: WAS THIS STABLE?

9	5	10	8	3	6	5	2	5	2	1	3	5	2	6	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

0	1	3	2	0	5	2	0	1	1	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

RADIX SORT

uses the **least** significant digit.

iteration 2

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

RADIX SORT

uses the least significant digit.

iteration 3

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

RADIX SORT

uses the <u>least</u> significant digit.

Time = ?

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

RADIX SORT

uses the <u>least</u> significant digit.

$\Theta(\ell \cdot (n+r))$

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