CS2040C Data Structures and Algorithms

Which one is your favorite character in Harry Potter?

- Harry Potter
- Ron Weasley
- Hermione Granger
- The Sorting Hat
- Bilbo Baggins

Today

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Before We Start

- Arithmetic progression
- Given a number n, what is

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots. + 1$$

- In Big O Notation?

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots. + 1$$
$$= n\ (n+1)/2$$
$$= O(n^2)$$

Sorting Problem Definition

- **Input:** an array `A[1..n]` of elements
- **Output:** array `B[1..n]` that is a permutation of `A`
  - such that:

$$B[1] \leq B[2] \leq \ldots \leq B[n]$$

- E.g.

$$A=[9,3,6,6,6,4] \rightarrow [3,4,6,6,6,9]$$

Let’s Try: BogoSort

```
BogoSort(A[1..n])
  Repeat:
    Choose a random permutation of the
  array A.
    If A is sorted, return A.
```

- What is the expected running time?

Let’s Try: QuantumBogoSort

```
BogoSort(A[1..n])
  Repeat:
    Choose a random permutation of the
  array A.
    If A is sorted, return A
        else destroy the universe
```

- What is the expected running time?
- (Remember QuantumBogoSort when you learn about non-deterministic Turing Machines.)

Today

- Sorting algorithms
  - **BubbleSort**
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

BubbleSort (Version 1)

```
BubbleSort(A, n)
  Repeat n times:
    for j ← 1 to n - 1
      if A[j] > A[j+1] then swap(A[j], A[j+1])
```

`if A[j] > A[j+1] then swap`

BubbleSort Example:

- Given:

8	2	4	9	3	6

- Repeat the first time

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	4	8	3	9	6
2	4	8	3	6	9

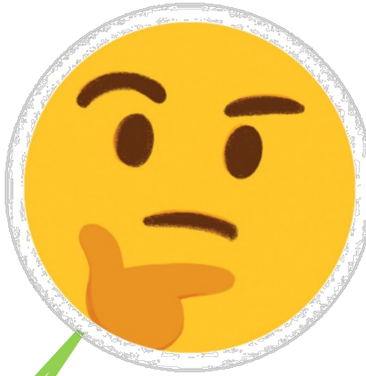
`if A[j] > A[j+1] then swap`

BubbleSort Example:

- From last iteration :

2	4	8	3	6	9

- Repeat the second time



2	4	8	3	6	9
2	4	8	3	6	9
2	4	8	3	6	9
2	4	3	8	6	9
2	4	3	6	8	9
2	4	3	6	8	9

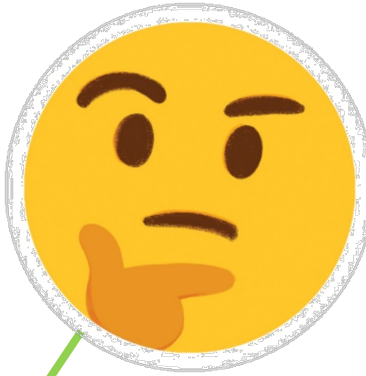
```
if A[j] > A[j+1] then swap
```

BubbleSort Example:

- From last iteration :

2	4	8	3	6	9

- For the second time, I can stop here!



2	4	8	3	6	9
2	4	8	3	6	9
2	4	8	3	6	9
2	4	3	8	6	9
2	4	3	6	8	9

`if A[j] > A[j+1] then swap`

BubbleSort Example:

- After one iteration, the last item is “fixed”

2	4	8	3	6	9

- After two iterations, the last two items are “fixed”

2	4	3	6	8	9

- After `i` iterations, the last `i` items are “fixed”!

BubbleSort (Version

I don’t have to go all the way to the **end** of the array

```
BubbleSort(A, n)
  Repeat n times:
    for j ← 1 to n - 1
      if A[j] > A[j+1] then swap(A[j], A[j+1])
```

BubbleSort (Version 2)

```
BubbleSort(A, n)
  for i ←1 to n - 1
   for j ←1 to n - i
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

What is the time complexity of BubbleSort?

```
BubbleSort(A, n)
  for i ←1 to n - 1
   for j ←1 to n - i
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

i	j = #inner loop iteration
1	n – 1
2	n – 2
3	n – 3
…	…
n – 1	1

Total running time = 1 + 2 + 3 + … + (n-1)
$= n\,(n\text{-}1)/2$
$= O(n^2)$

```
if A[j] > A[j+1] then swap
```

BubbleSort Example:

- From last (second) iteration:

2	3	4	6	8	9

- After the third iteration

2	3	4	6	8	9

- And how many more iterations do I have to go?

BubbleSort (Version 3)

```
BubbleSort(A, n)
  repeat until no more swapping
   for j ← 1 to n - 1
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

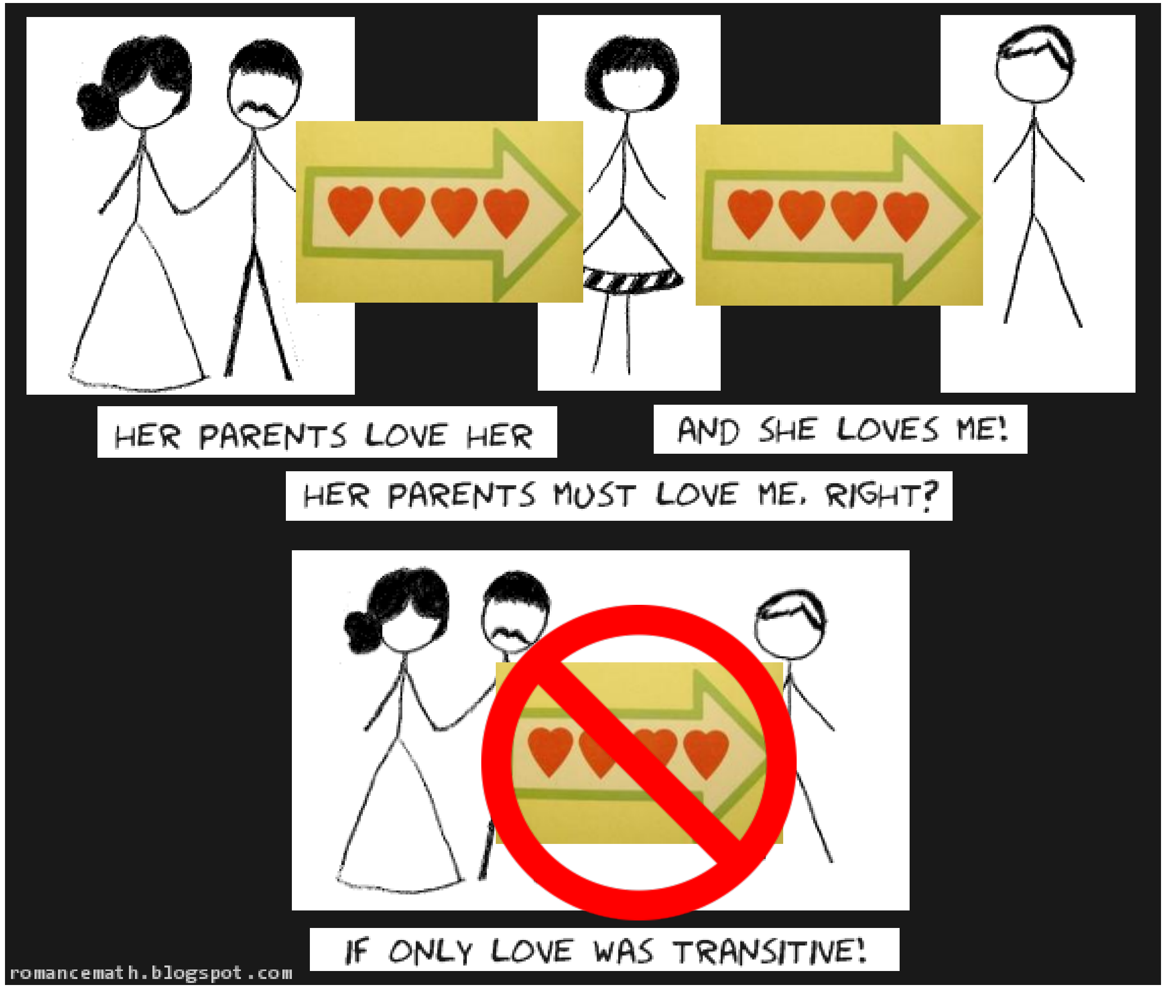
- Running time?
  - Best-case: $O(n)$
  - Worst-case: $O(n^2)$

Is BS only limited to sorting numbers?

```
BubbleSort(A, n)
  repeat until no more swapping
   for j ← 1 to n - 1
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

- You can use this to sort any elements with a total ordering that is ***transitive***

Non-transitive relationship



Sorting ANY type of elements

```
BubbleSort(A, n)
  repeat until no more swapping
   for j ←1 to n - 1
    if A[j].compareTo(A[j+1])==1 then swap(A[j], A[j+1])
```

- Such That:

```
x.compareTo(y)  :
  -1:    if (x<y)
   0:    if (x == y)
   1:    if (x>y)
```

Or you can just overload the “>” operator

```
BubbleSort(A, n)
  repeat until no more swapping
   for j ← 1 to n - 1
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

Let’s Say We Want to Sort the class FOOD

```
class Food {
private:
  string _name;
  int _cal;
public:
  Food() { _name = ""; _cal = 0; };
  Food(string, int);

  bool operator>(const Food&);

  friend ostream &operator<<(ostream&, const Food&);
};
```

return True or False when we ask if food1>food2

Usage

```
bool Food:: operator>(const Food& f) {
  return _cal > f._cal;
}

int main() {
    Food dish1("Chicken", 100);
    Food dish2("Rice", 400);
   if ( dish1 > dish2 ) . . .
```

```cpp
Food food1("Salad", 100);
Food food2("French Fries", 10000);

cout << "Among" << endl;
cout << food1;
cout << "and" << endl;
cout << food2;
cout << "The food with more calories is" <<
endl;
cout << (food1 > food2 ? food1 : food2);
```

```
Among
Salad with 100 calories
and
French Fries with 10000 calories
The food with more calories is
French Fries with 10000 calories
Press any key to continue . . .
```

BubbleSort C++ Code for Every Class

```
template<class TypeT>
void bubble(TypeT a[], int n) {
  int i, j;
  for (i = 0; i < n - 2; i++)
    for (j = 0; j < n-i-1; j++)
      if (a[i]>a[i+1])
        swap(a[i],a[i+1]);
}
```

* In C++, we assume the array indices are from 0 to n-1

Today

- Sorting algorithms
  - BubbleSort
  - **<u>SelectionSort</u>**
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

SelectionSort

```
SelectionSort(A, n)
   for j ← 1 to n - 1:
     find index k s.t. A[k] is the smallest in A[j..n]
     swap(A[j], A[k])
```

SelectionSort Example:

- j = 1, find the smallest in A[1..n] and swap it into A[1]

- j = 2, find the smallest in A[2..n] and swap it into A[2]

- j = 3, find the smallest in A[3..n] and swap it into A[3]

SelectionSort Example:

- Given:

8	2	4	9	3	6

- j = 1, A[2]=2 is the smallest in A[1..n]

2	8	4	9	3	6

- j = 2, A[5]=3 is the smallest in A[2..n]

2	3	4	9	8	6

- j = 3, A[3]=4 is the smallest in A[3..n]

2	3	4	9	8	6

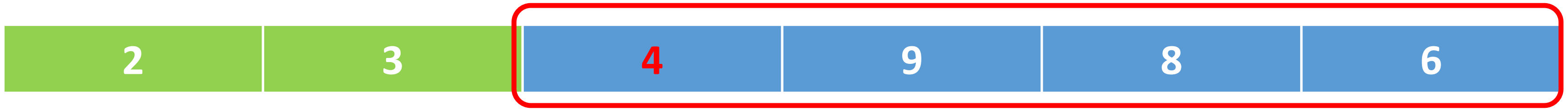
Loop Invariant:

- After j iterations, the subarray A[1..j] is sorted

- j = 1, A[2]=2 is the smallest in A[1..n]

- j = 2, A[5]=3 is the smallest in A[2..n]

- j = 3, A[3]=4 is the smallest in A[3..n]



SelectionSort Time Complexity?

```
SelectionSort(A, n)
   for j ← 1 to n - 1:
      find index k s.t. A[k] is the smallest in A[j..n]
      swap(A[j], A[k])
```

- Only loop n times, so $O(n)$?
- But how long does it take to search for the minimum in `A[j..n]`?
  - $O(n - j)$
- Total complexity:
  - $O(n - 1) + O(n - 2) + O(n - 3) + \ldots + O(1) = O(n^2)$

Today

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - **InsertionSort**
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Loop Invariant:

- **After j iterations, the subarray A[1..j] is sorted**
- What is another way to maintain the loop invariant if I do not want to look for the minimum of the rest of the array?

1	6	8	3	9	7

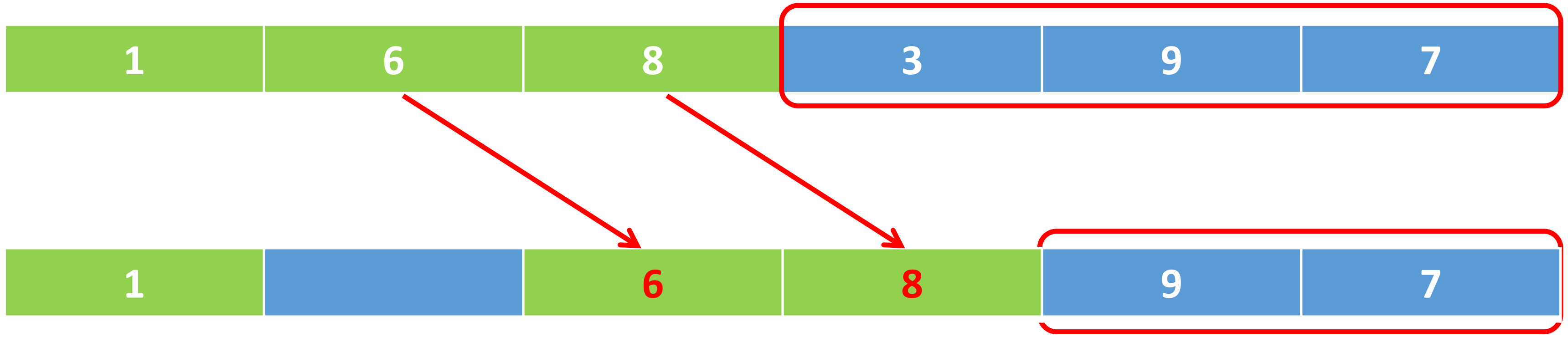
- For the k^{th} iteration, I know that the left part is sorted, how to I merge the *next* number into the green part?
- **Insert** the item into the "right" position in the sorted array!

1	3	6	8	9	7

- So we don't need to find the minimum like SelectionSort, so $O(n)$?

But When you “Insert”

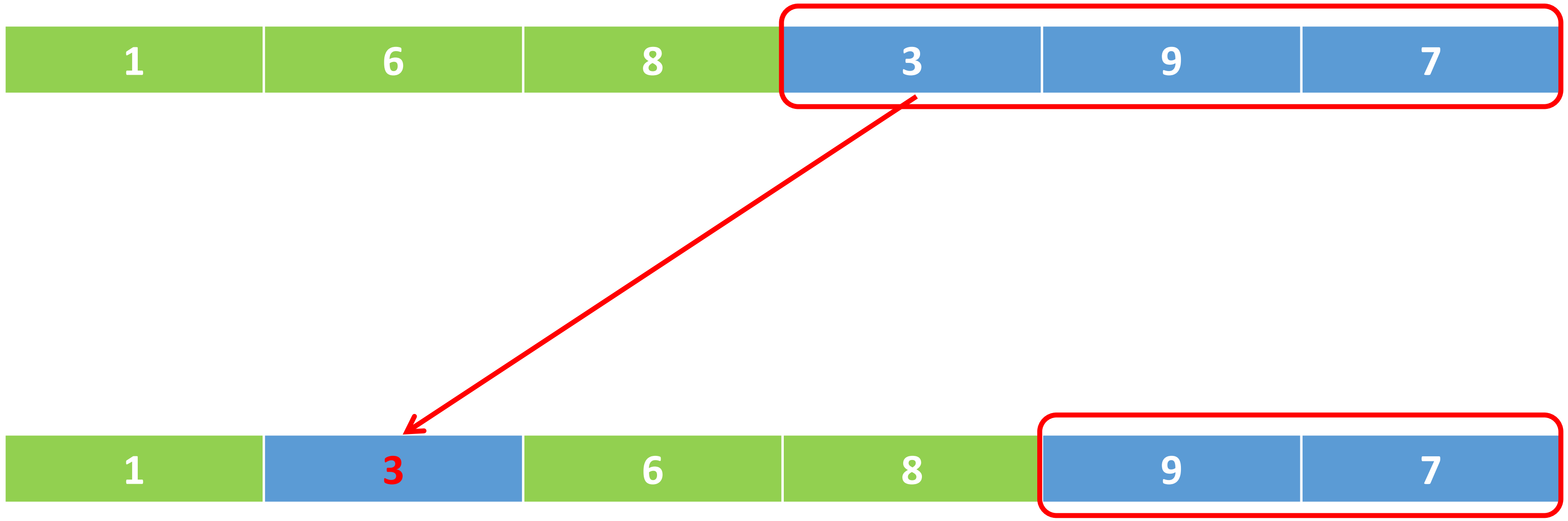
- You need to “shift” some elements to the right



- What will be the worst case scenario?
  - Or how many numbers do we shift for every iteration?
- Or what is the best scenario?

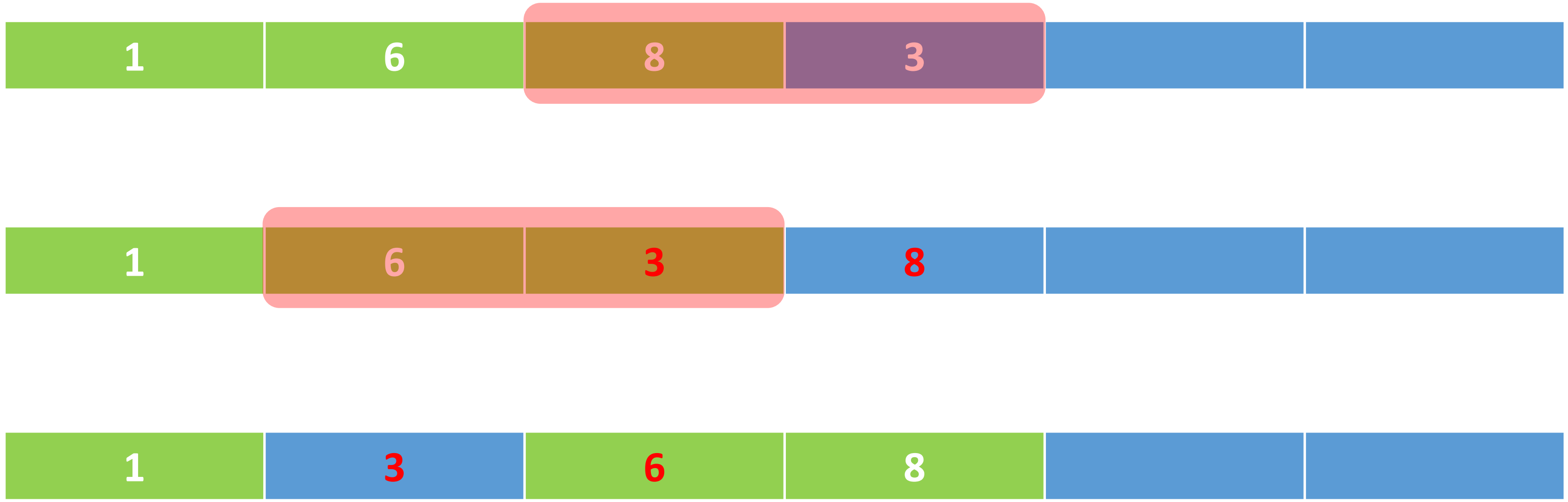
InsertionSort

```
InsertionSort(A, n)
    for  j ← 2 to n
      key ← A[j]
      Insert key into the sorted array A[1..j-1]
```



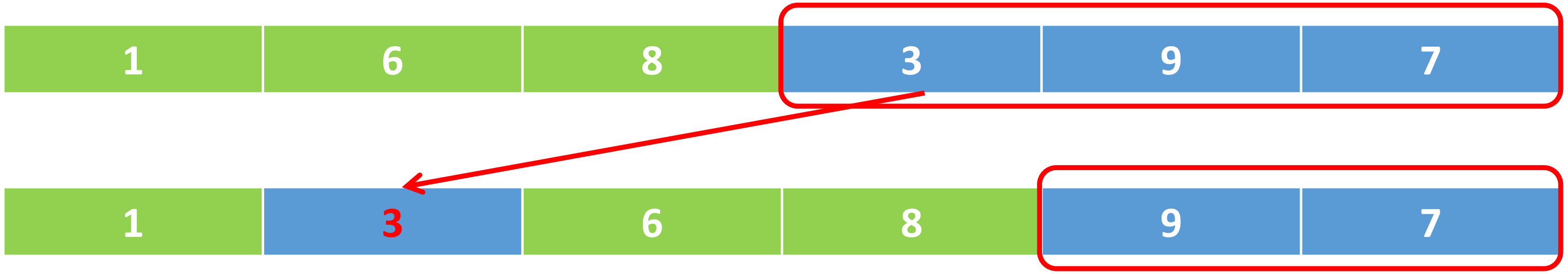
We can do “inverse” BubbleSort!

```
InsertionSort(A, n)
    for  j ← 2 to n
      key ← A[j]
      Insert key into the sorted array A[1..j-1]
```

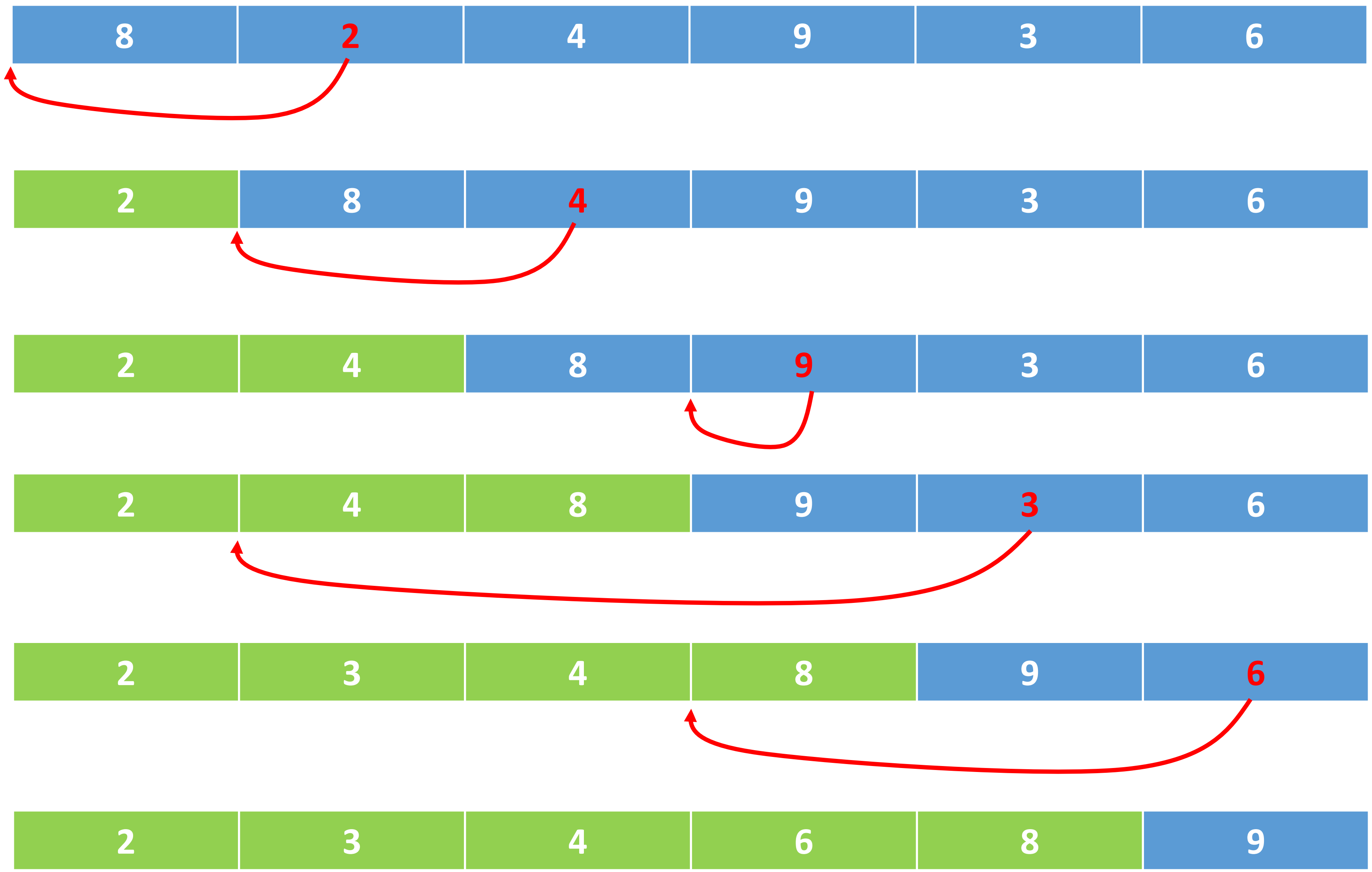


InsertionSort

```
InsertionSort(A, n)
    for  j ← 2 to n
     key ← A[j]
     i ← j-1
     while (i > 0) and (A[i] > key)
          A[i+1] ← A[i]
          i ← i-1
     A[i+1] ← key
```



InsertionSort Example



InsertionSort Time Complexity

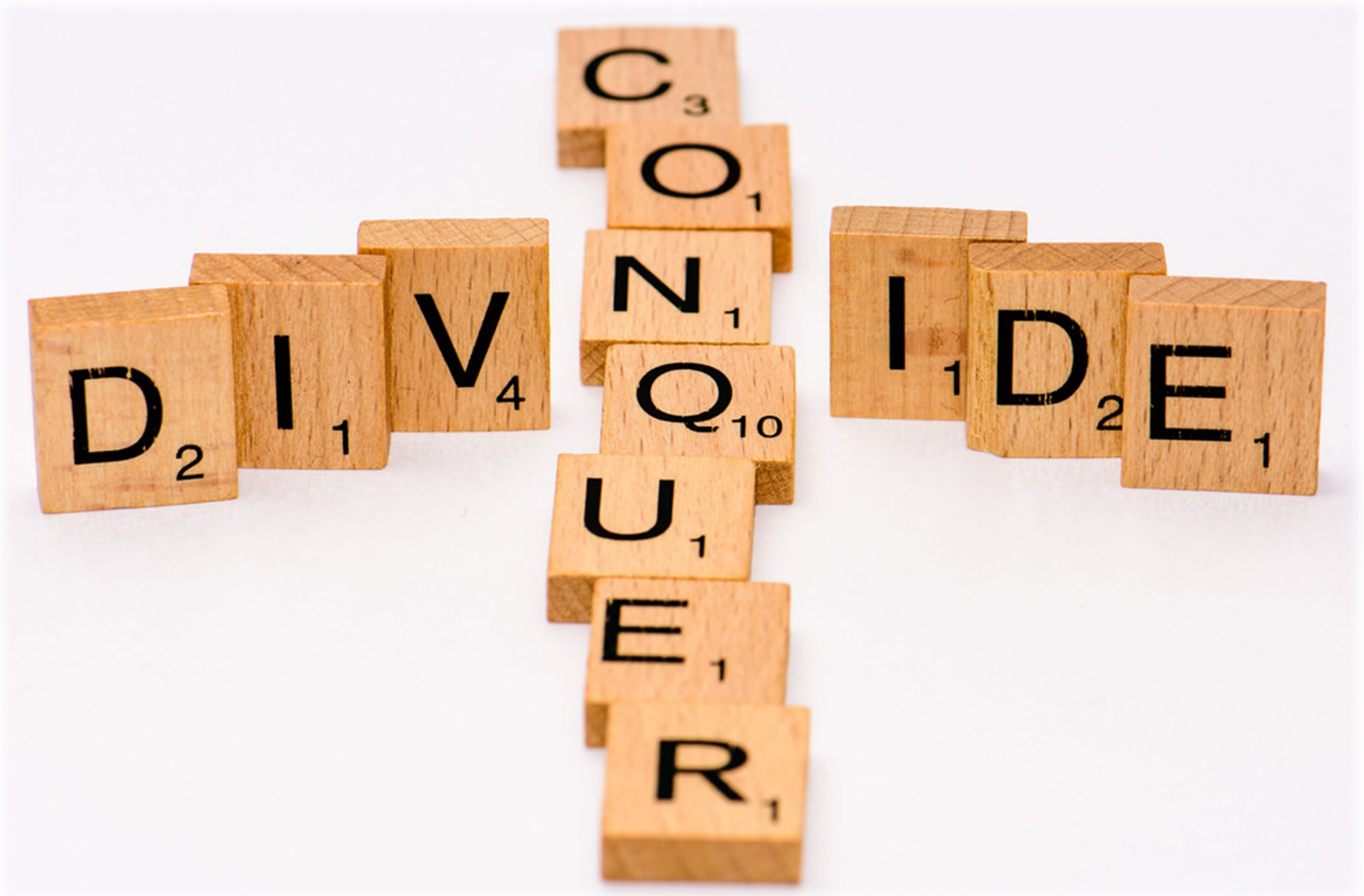
```
InsertionSort(A, n)
    for  j ← 2 to n
      key ← A[j]
      i ← j-1
      while (i > 0) and (A[i] > key)
           A[i+1] ← A[i]
           i ← i-1
      A[i+1] ← key
```

- Worst-case: $O(n^2)$
- What is the best-case scenario?

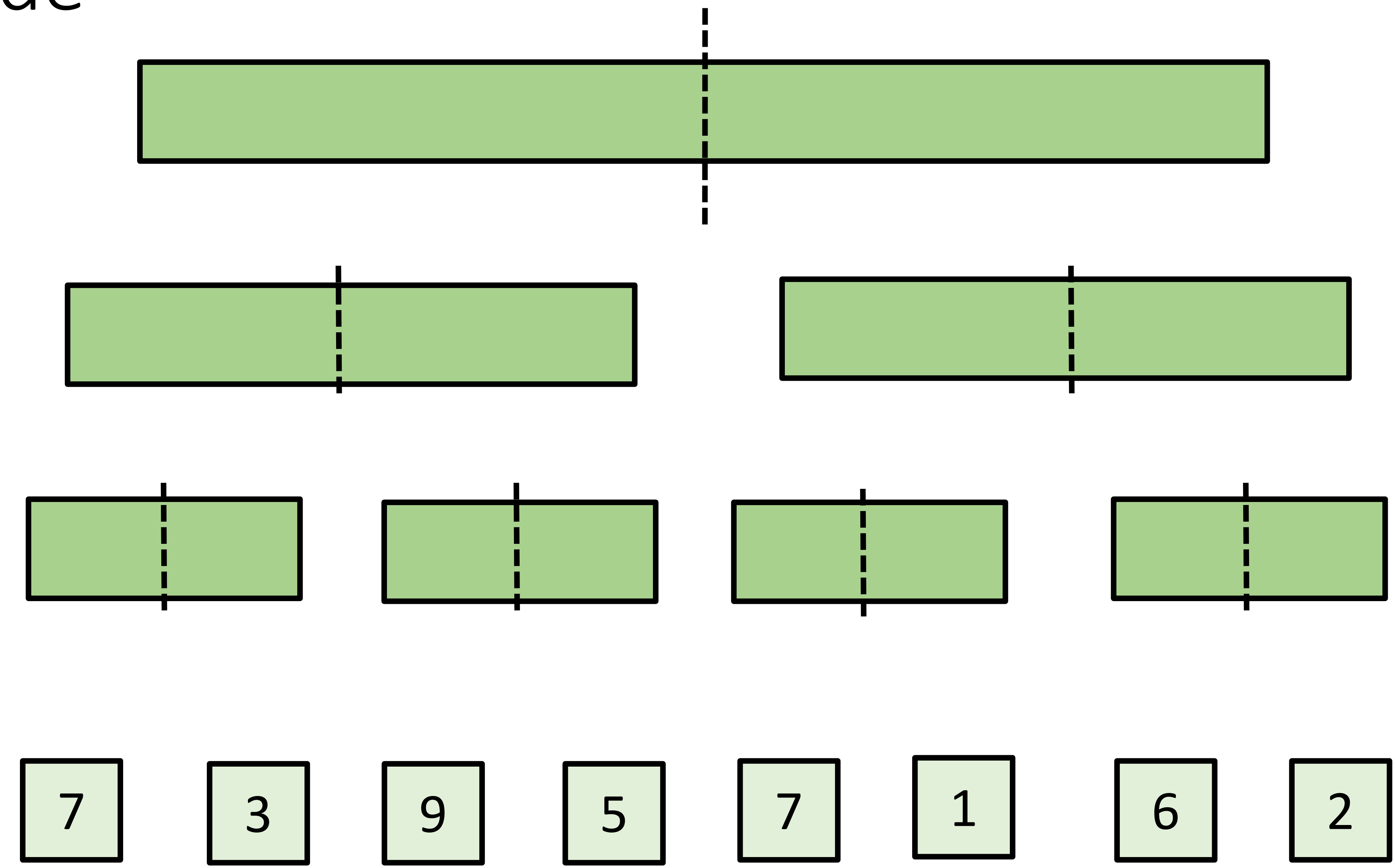
Today

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

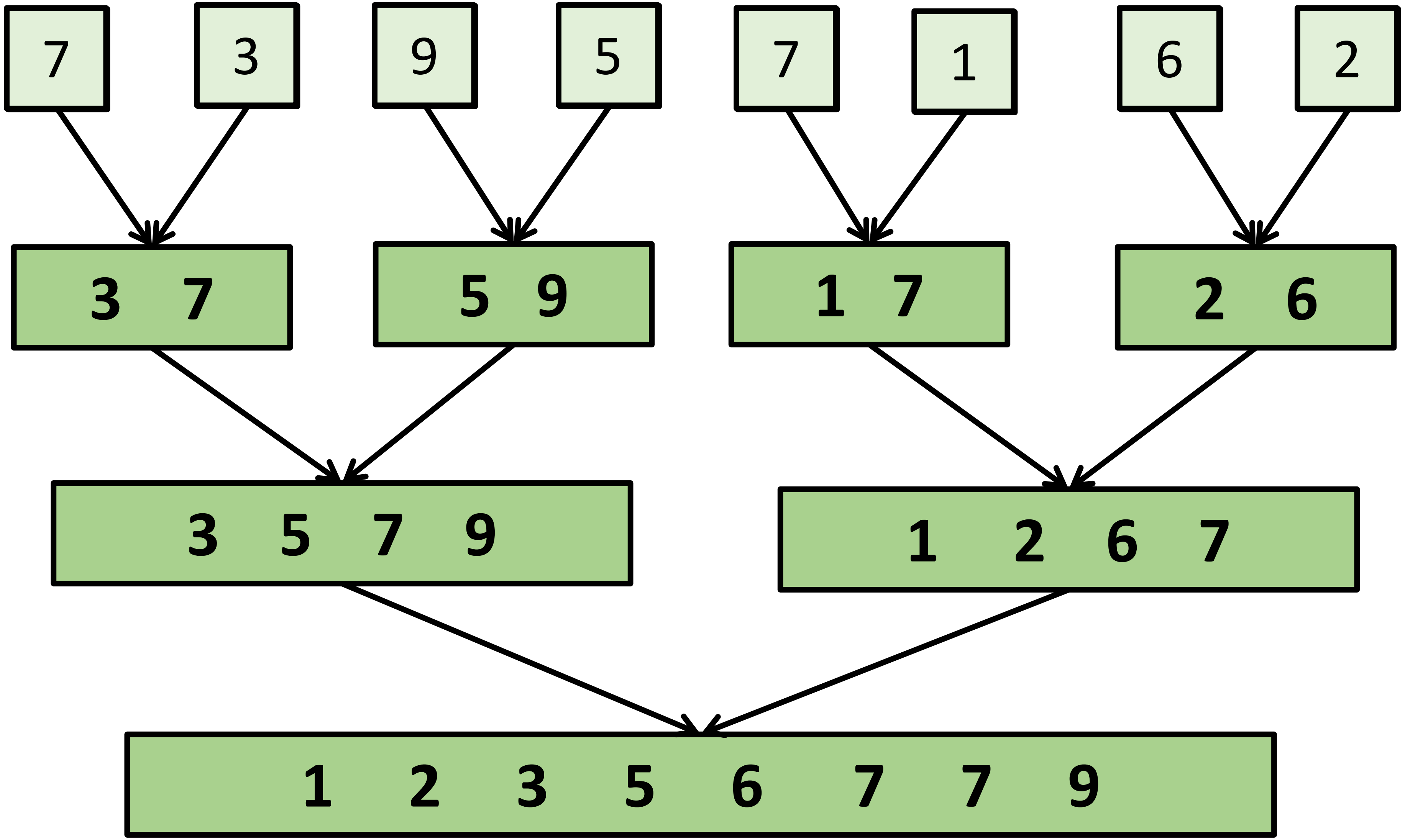
Idea

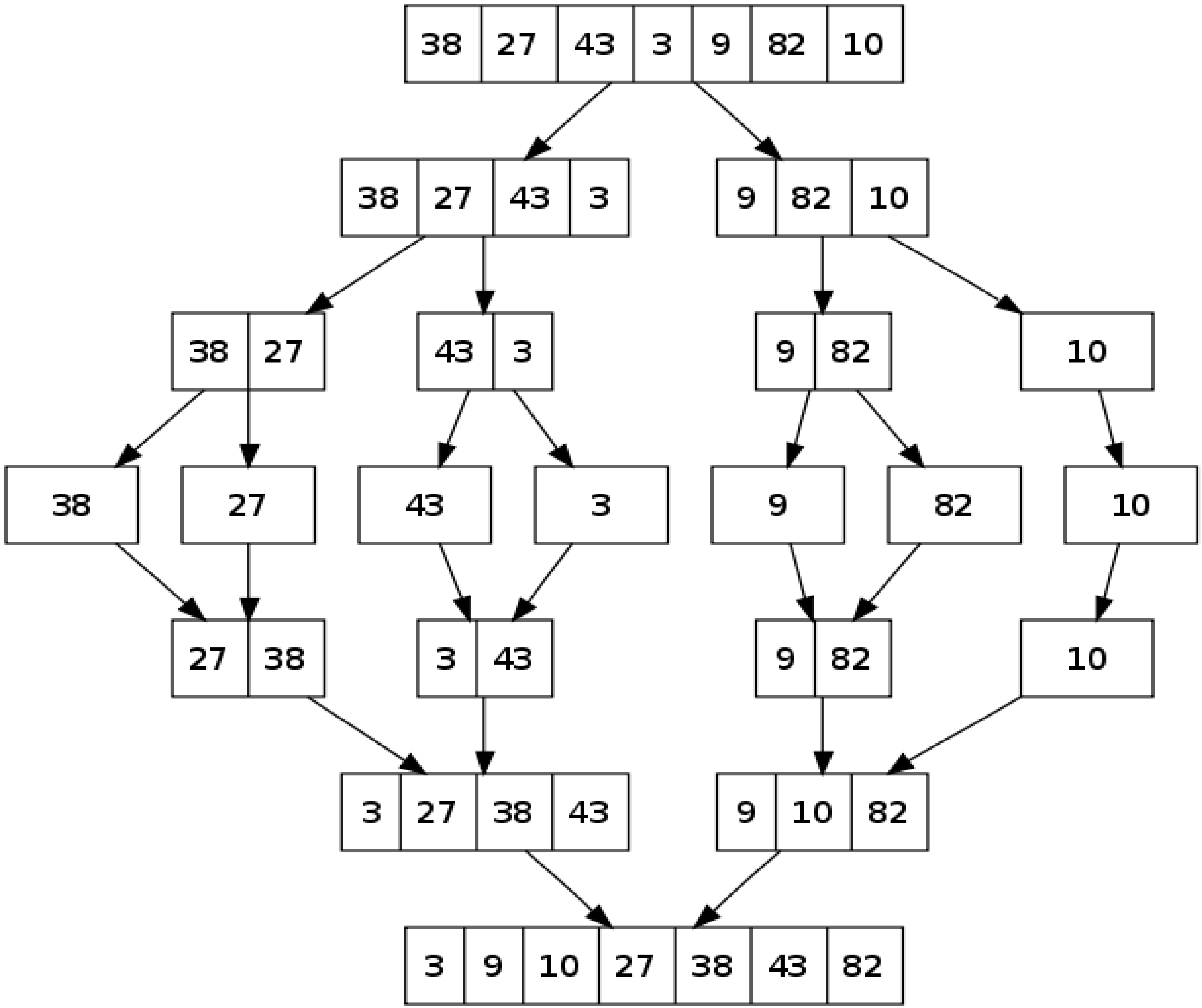


Divide



Merge




38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

MergeSort

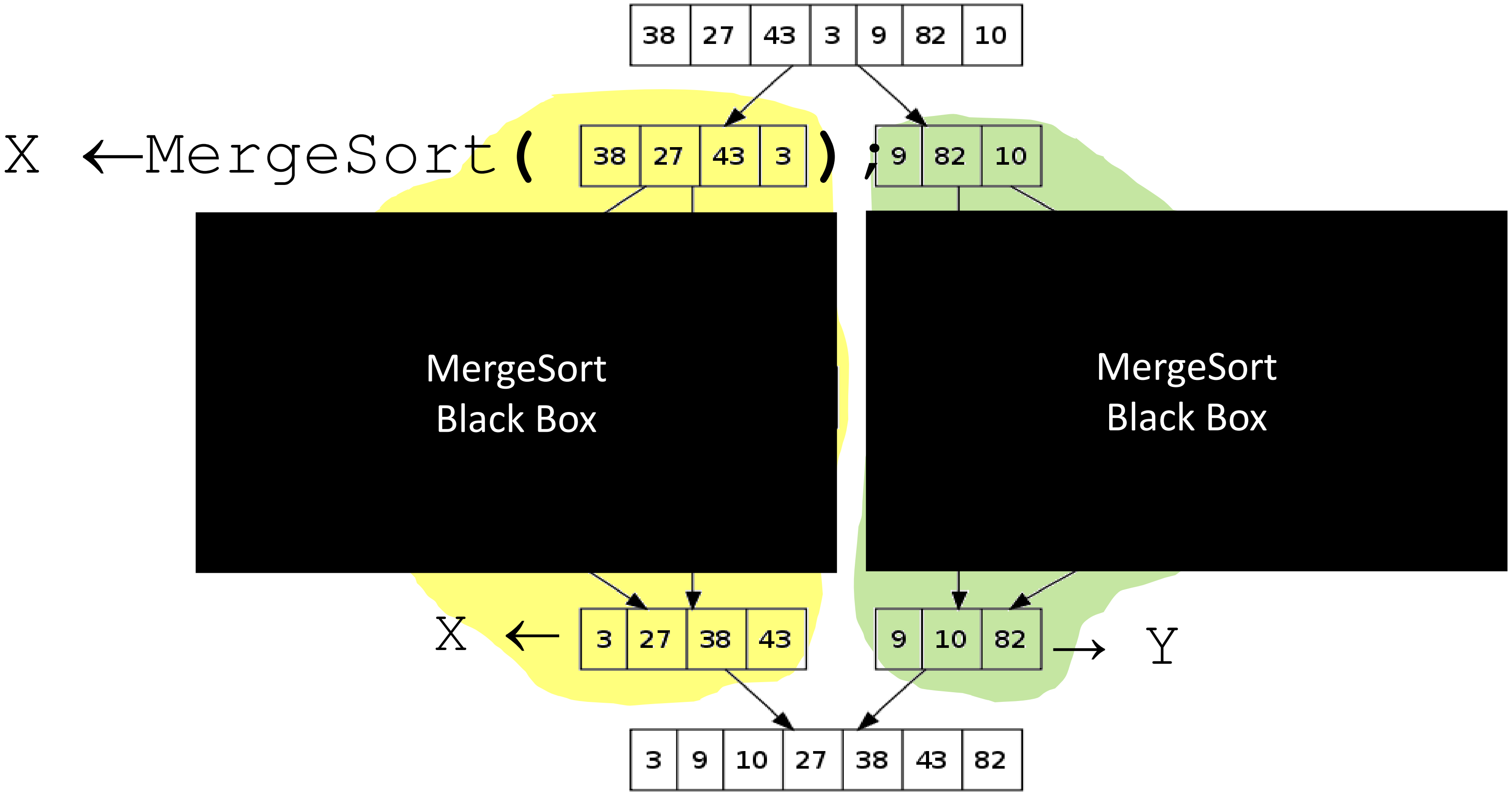
```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
     return Merge (X,Y, n/2);
```

Divide

Merge

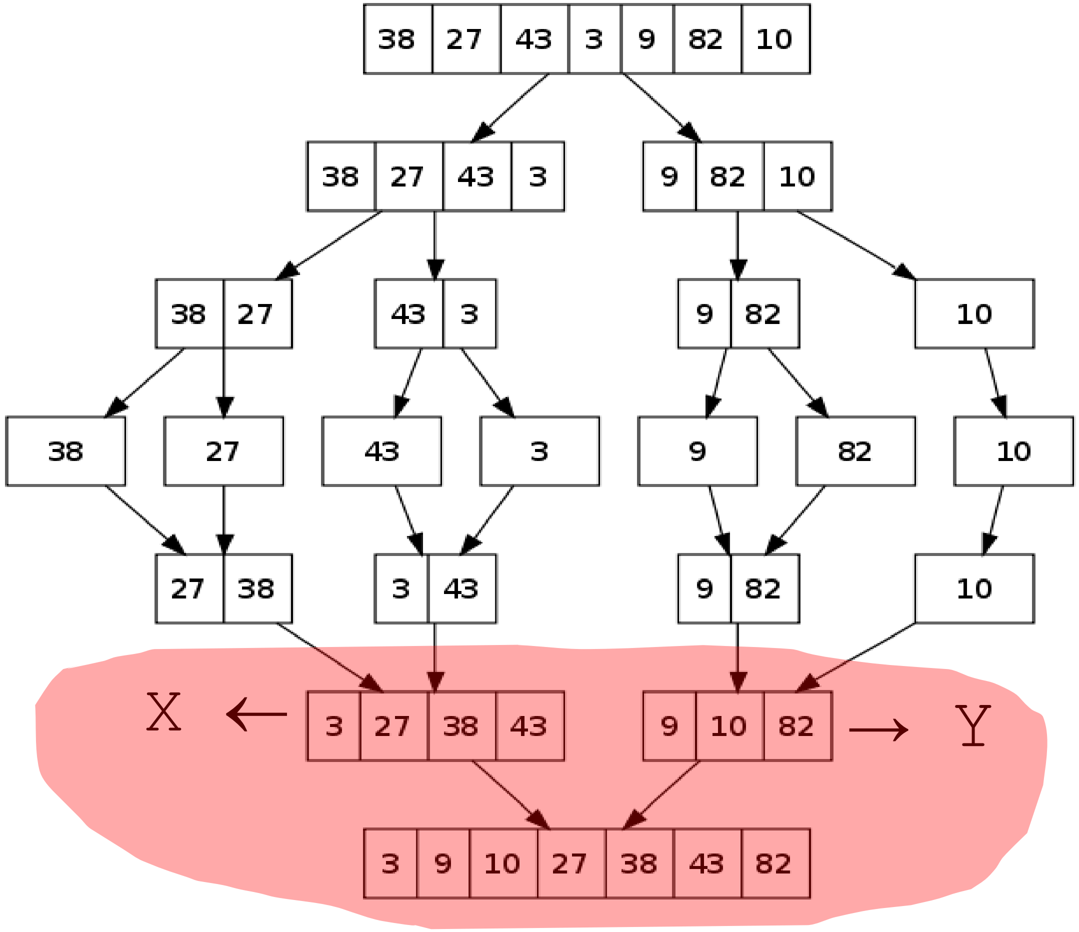
Divide

```
X ←MergeSort(A[1..n/2],  n/2);
Y ←MergeSort(A[n/2+1, n],  n/2);
```



Merge

```
Merge (X,Y, n/2);
```



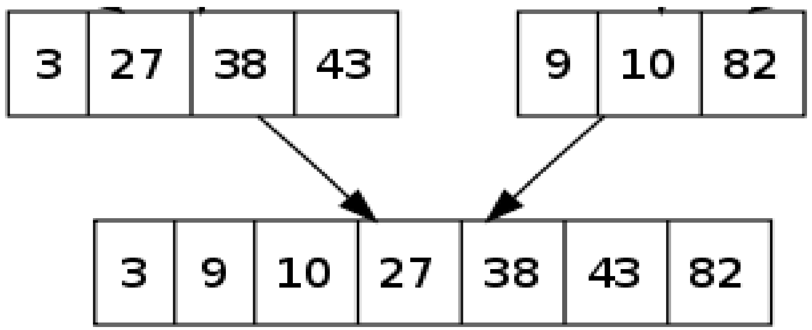
How to do Merge?

```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
     return Merge (X,Y, n/2);
```

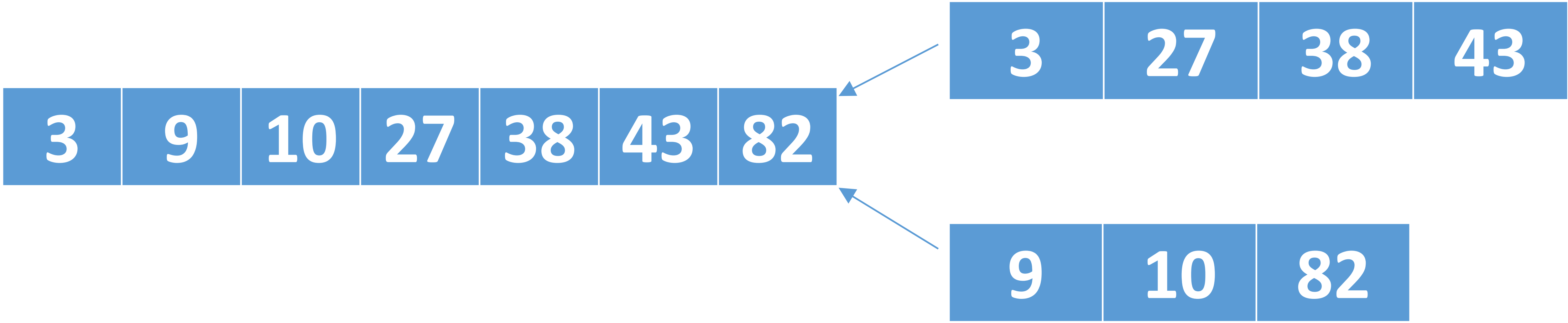
Divide

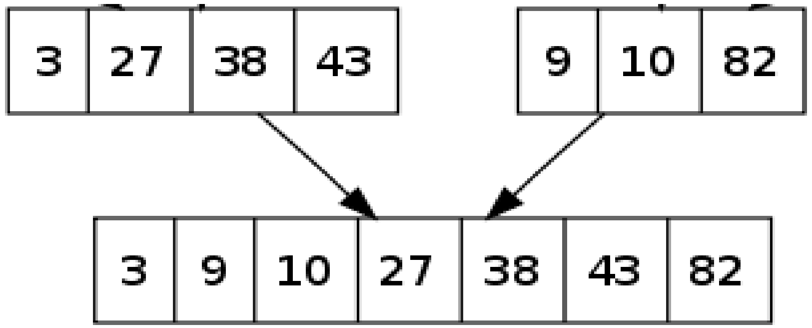
Merge

Merge



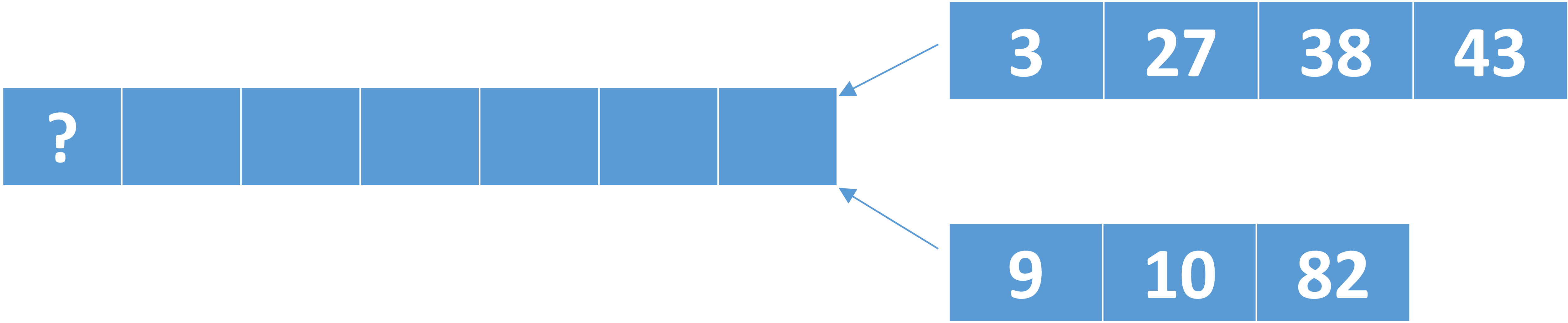
- Given two sorted lists, how to merge into one?
- Image there are two queues? How to merge into one?



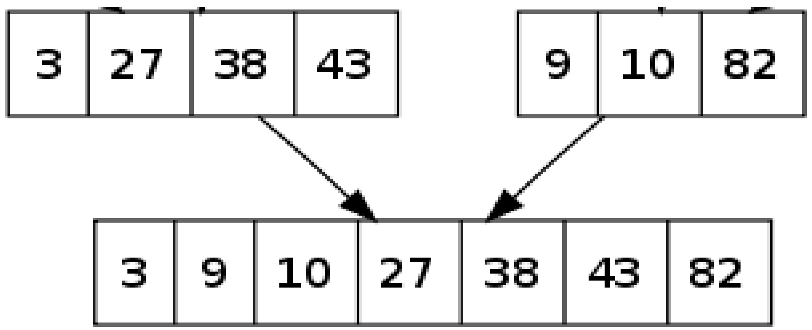


Merge

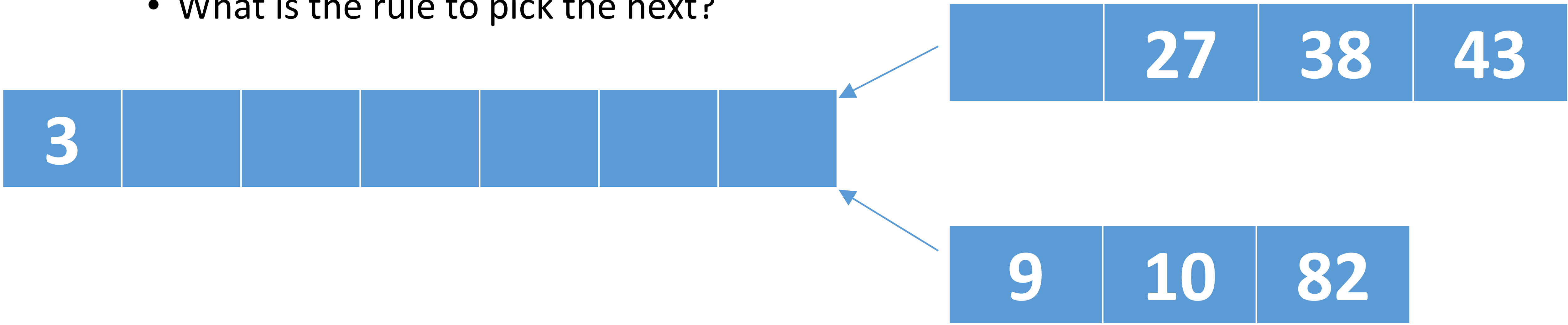
- Given two sorted lists, how to merge into one?
- Image there are two queues? How to merge into one?
  - Which one should be the first in the array?



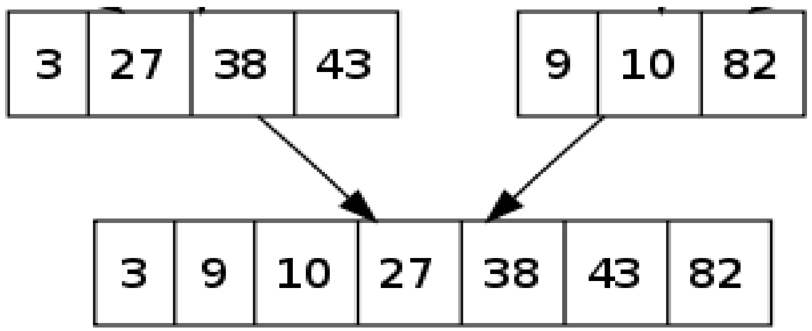
Merge



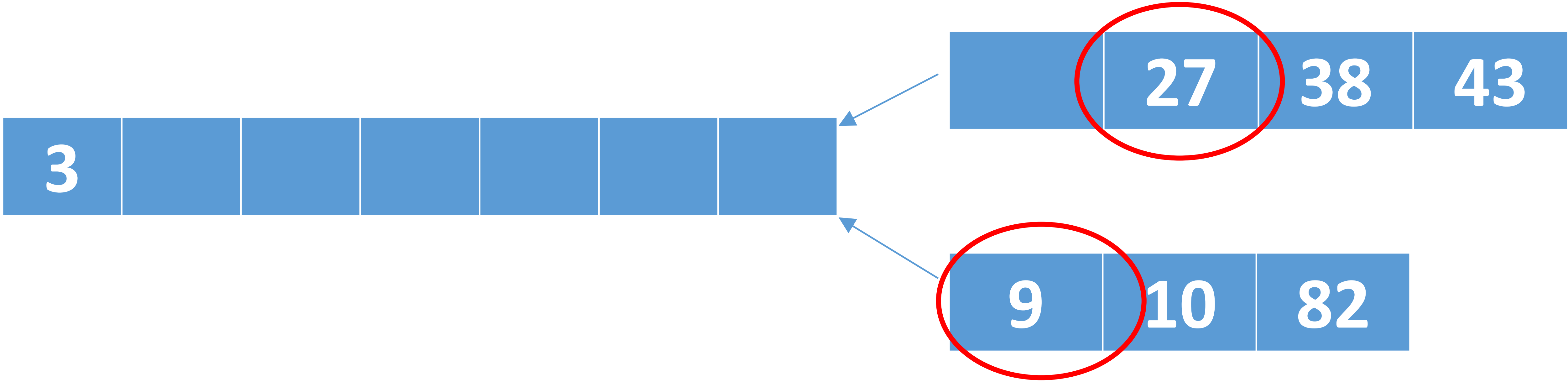
- Given two sorted lists, how to merge into one?
- Image there are two queues? How to merge into one?
  - Which one should be the first in the array?
- Then?
  - What is the rule to pick the next?



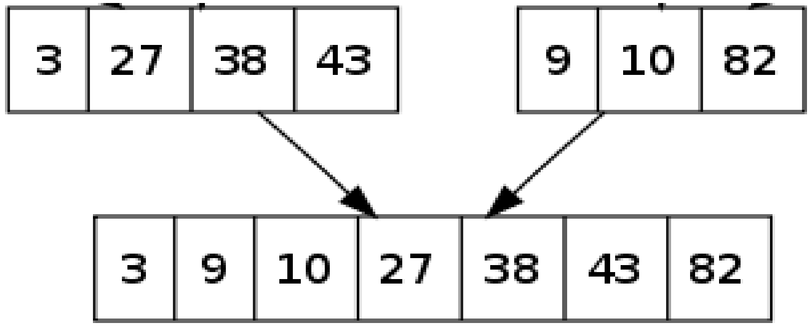
Merge



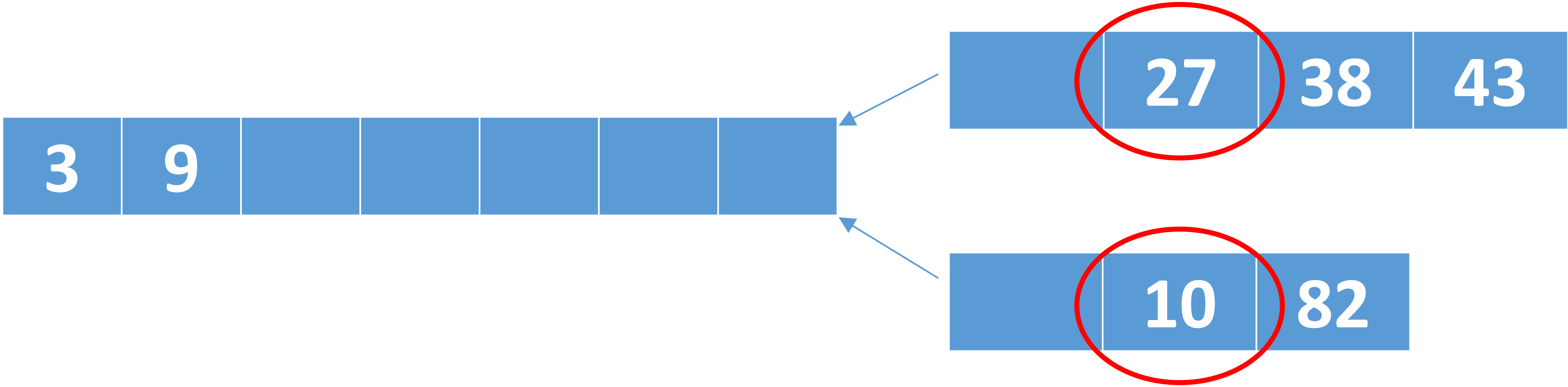
- Given two sorted lists, how to merge into one?
- Compare the two "heads" of the remaining queues
  - Pick the smaller one



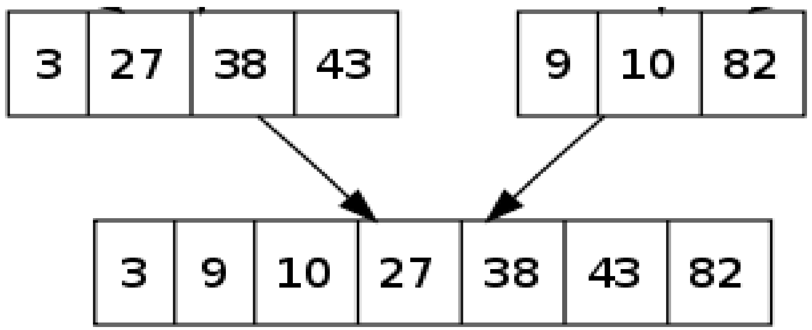
Merge



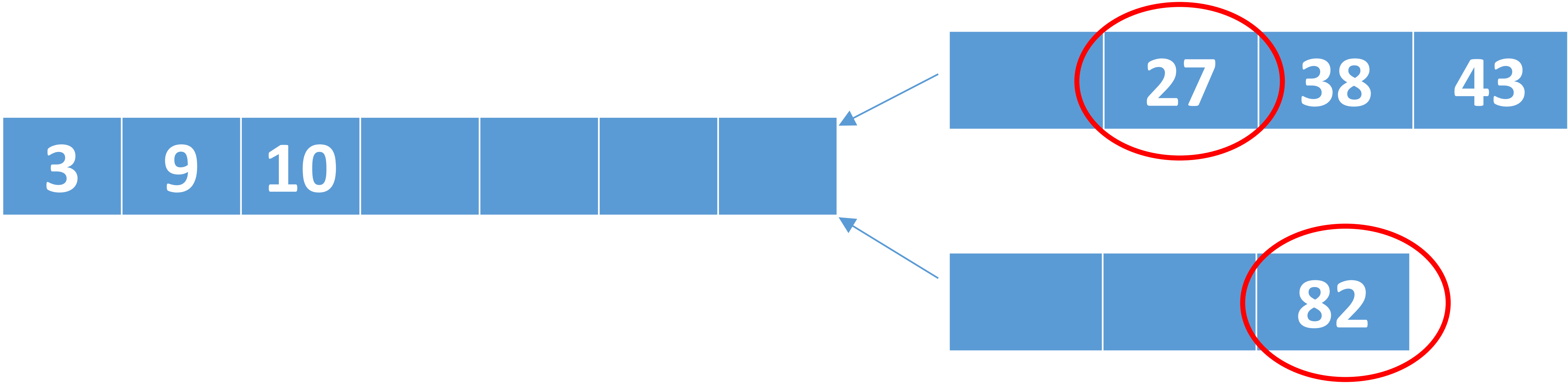
- Given two sorted lists, how to merge into one?
- Compare the two “heads” of the remaining queues
  - Pick the smaller one



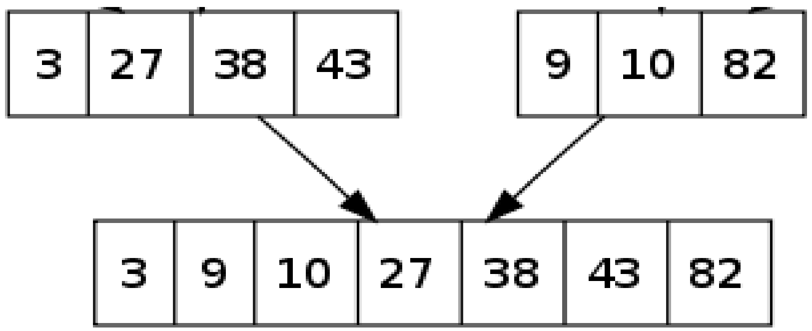
Merge



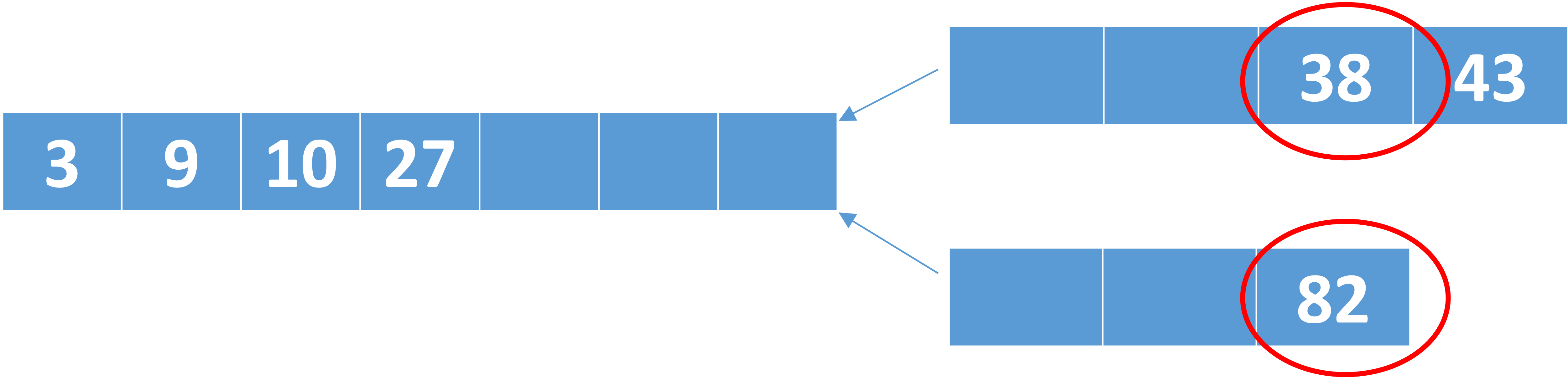
- Given two sorted lists, how to merge into one?
- Compare the two “heads” of the remaining queues
  - Pick the smaller one



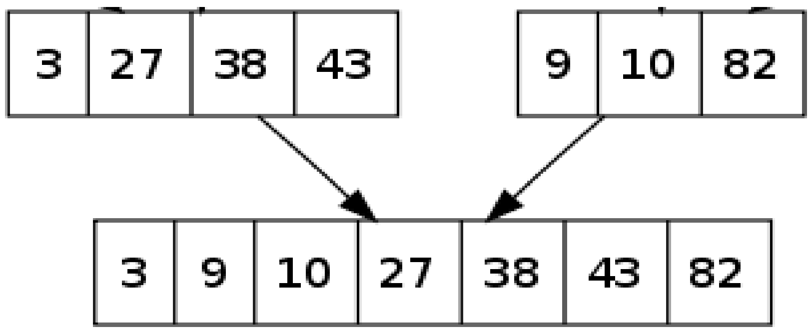
Merge



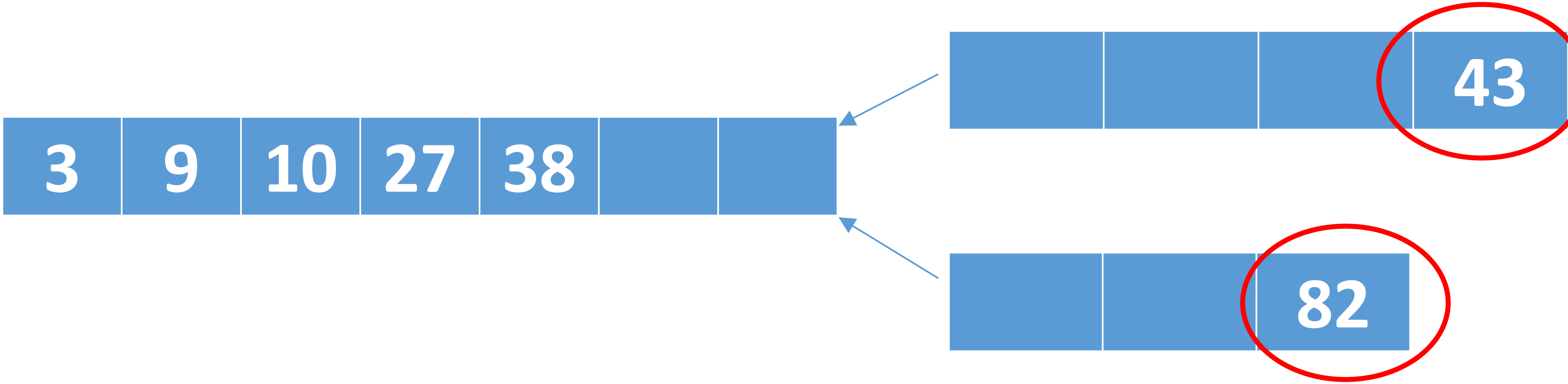
- Given two sorted lists, how to merge into one?
- Compare the two “heads” of the remaining queues
  - Pick the smaller one



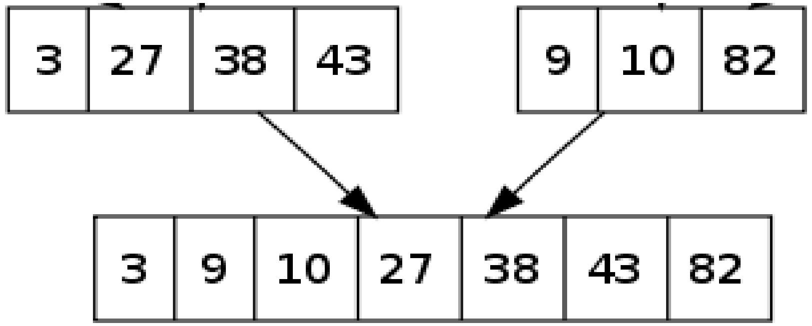
Merge



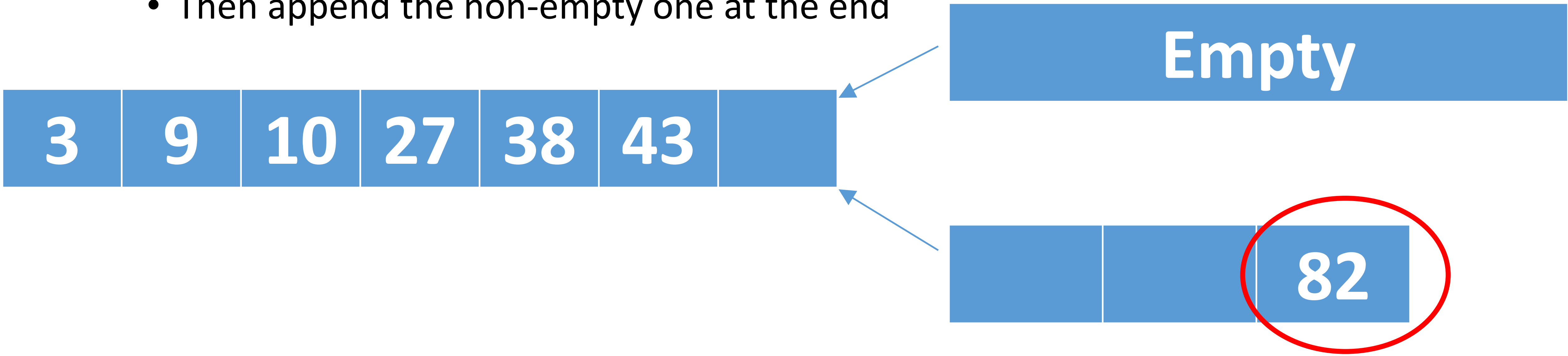
- Given two sorted lists, how to merge into one?
- Compare the two “heads” of the remaining queues
  - Pick the smaller one
- Until?



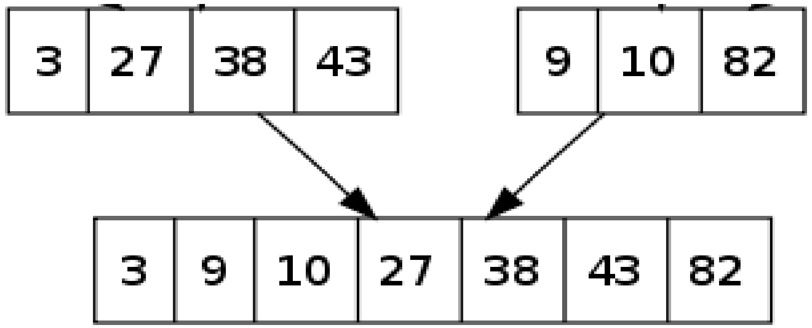
Merge



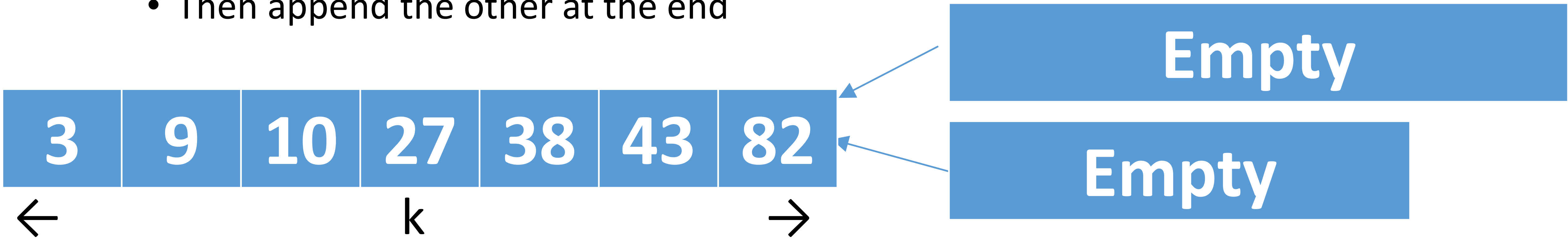
- Given two sorted lists, how to merge into one?
- Compare the two “heads” of the remaining queues
  - Pick the smaller one
- Until either one of the two queues is empty
  - Then append the non-empty one at the end



Merge



- Given two sorted lists, how to merge into one?
- Compare the two "heads" of the remaining queues
  - Pick the smaller one
- Until either one of the two queues is empty
  - Then append the other at the end



- Running time merging two queues into one array with k elements?
  - O(k) because need constant time to move one element from the 2 queues

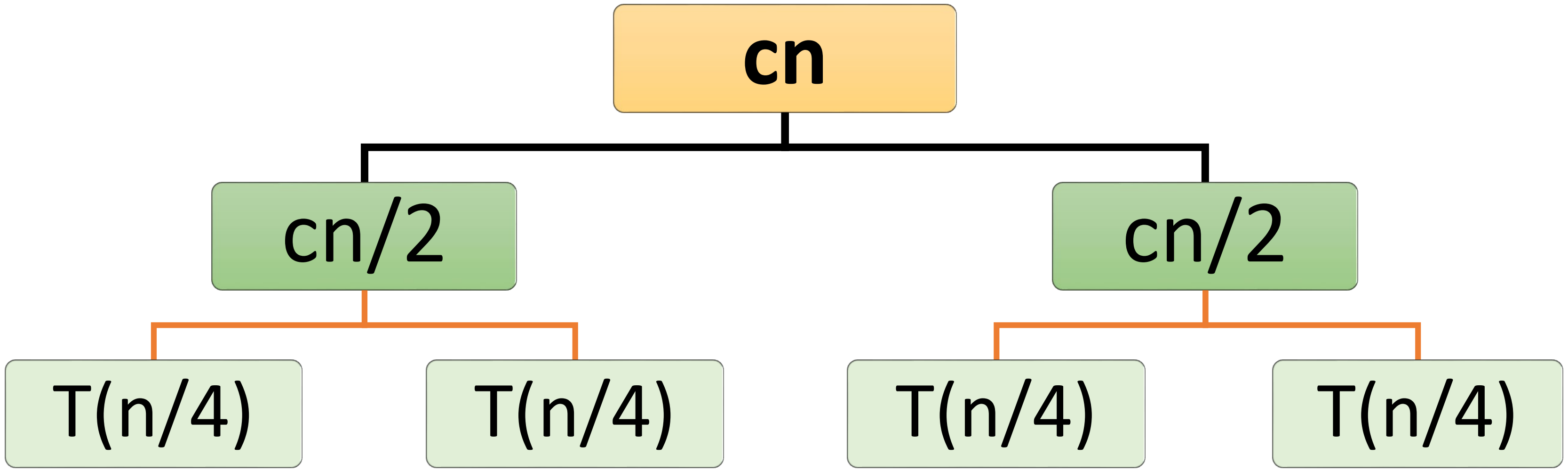
Time Complexity?

```
MergeSort(A, n)
   if (n=1) then return;
   else:
        X ←MergeSort(A[1..n/2],  n/2);   ←——  T(n/2)
        Y ←MergeSort(A[n/2+1, n],  n/2); ←——  T(n/2)
    return Merge (X,Y, n/2);   ←————————————  O(n)
```

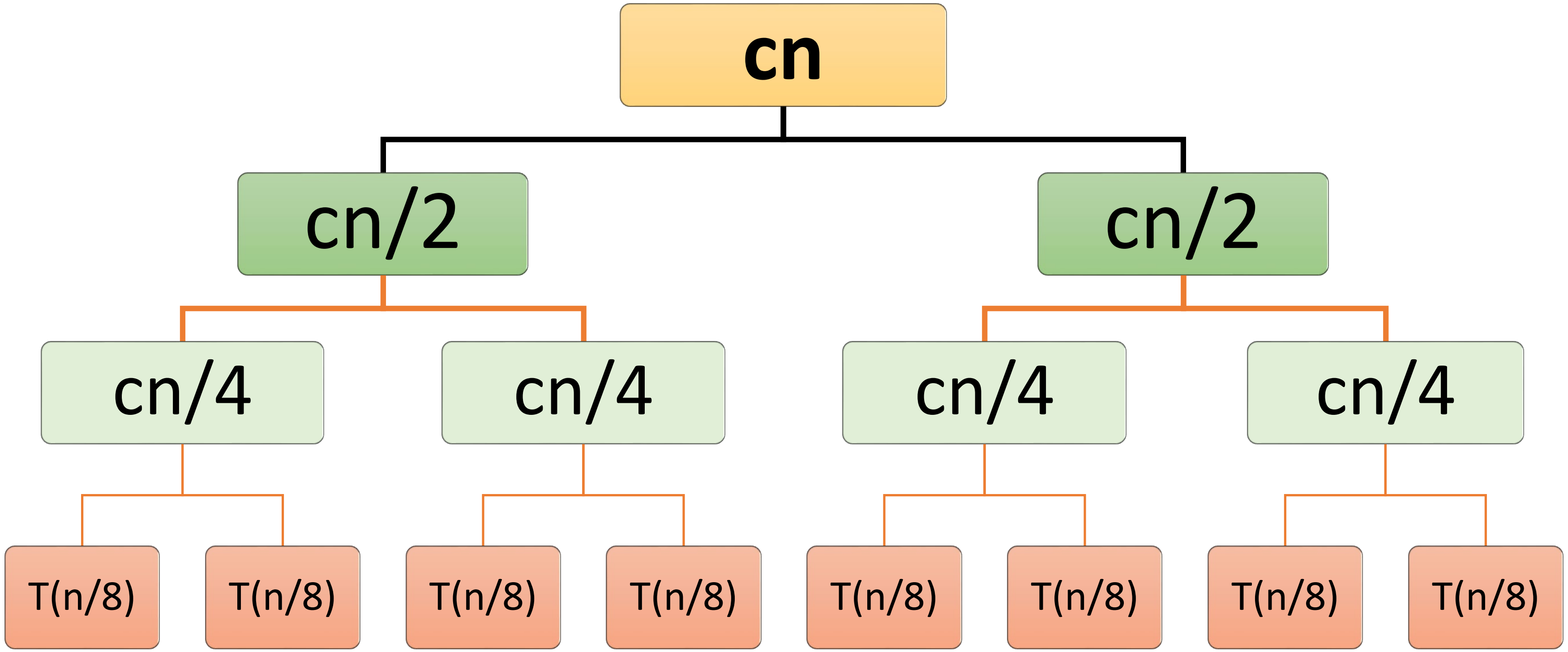
- $T(n) = T(n/2) + T(n/2) + c\ n$

  $= 2\ T(n/2) + c\ n$

$$T(n) = 2\,T(n/2) + c\,n$$

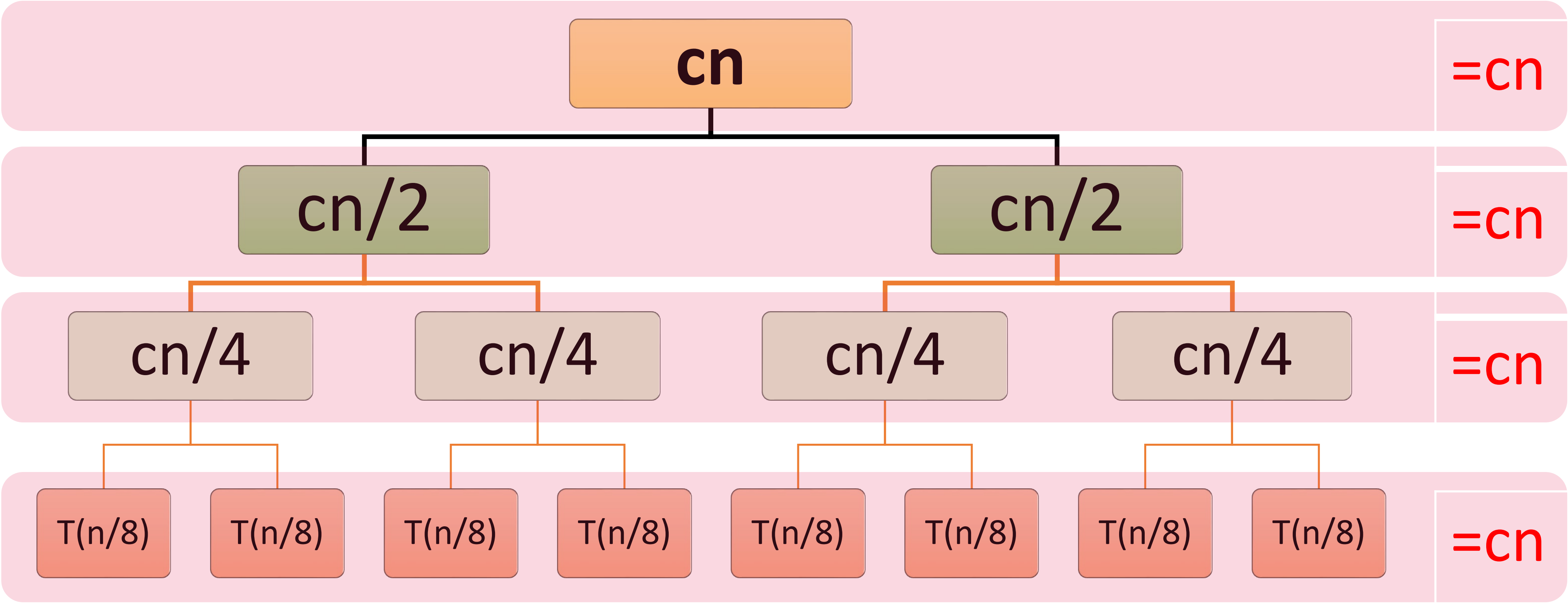


$$T(n) = 2\,T(n/2) + c\,n$$



$$T(n) = 2\,T(n/2) + c\,n$$

- How many levels ?



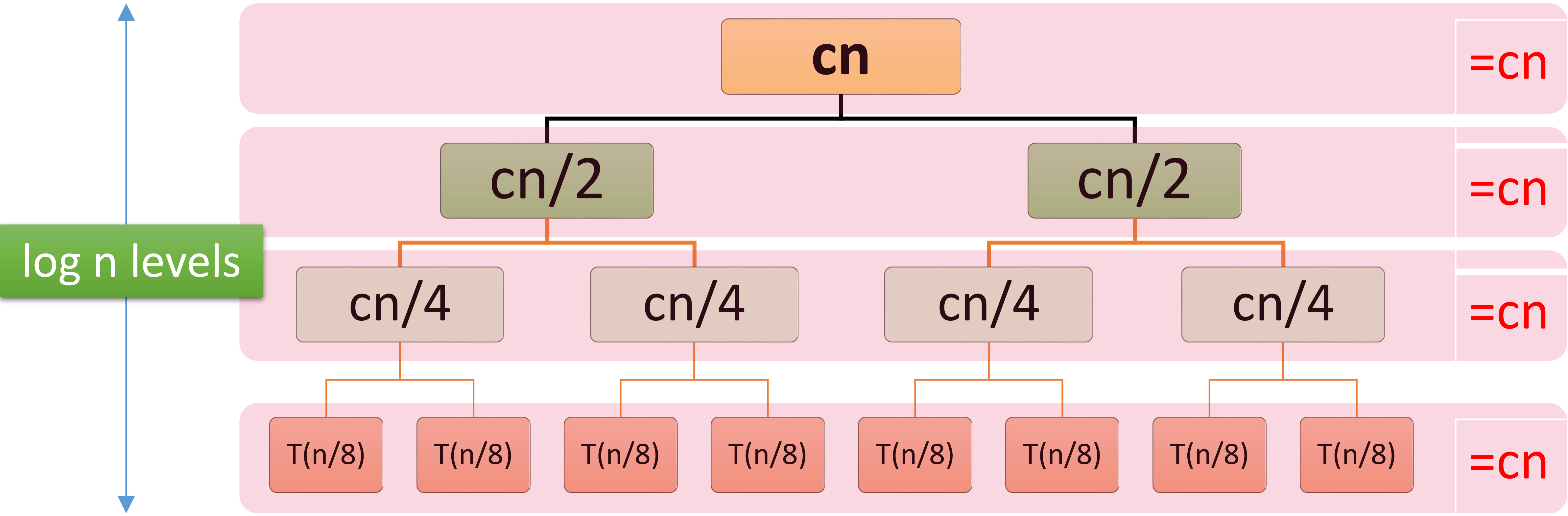
How many Levels?

- Each extra level we can handle a double number of elements
- $n = 2^h$
- Therefore $h = \log n$

Level	Number
0	1
1	2
2	4
3	8
4	16
$\dots$	$\dots$
h	2^h

$T(n) = cn \times \log n = O(n \log n)$

- How many levels ?



Another way

- $T(n) = 2\ T(n/2) + cn$

$$= 2\ (\ 2\ \mathrm{T}(n/4) + c\ (n/2)\ )\ + cn$$
$$= 4\ T(n/4) + 2\ c\ (n/2) + cn$$
$$= 4\ T(n/4) + 2\ cn$$
$$= 8\ T(n/8) + 3\ cn$$
$$= 16\ T(n/16) + 4\ cn$$
$$= 2^k\ T(n/2^k) + k\ cn$$
$$\ldots$$
$$= n\ T(1) + cn \log n$$
$$= \mathrm{O}(n \log n)$$

Finally $n/2^k = 1$

$k = \log_2 n$ (and $2^k = n$)

MergeSort Time Complexity: O(n log n)

```
MergeSort(A, n)
   if (n=1) then return;
   else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```

Sorting so far

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort

- Worst-case
  $O(n^2)$
  $O(n^2)$
  $O(n^2)$
  $O(n \log n)$

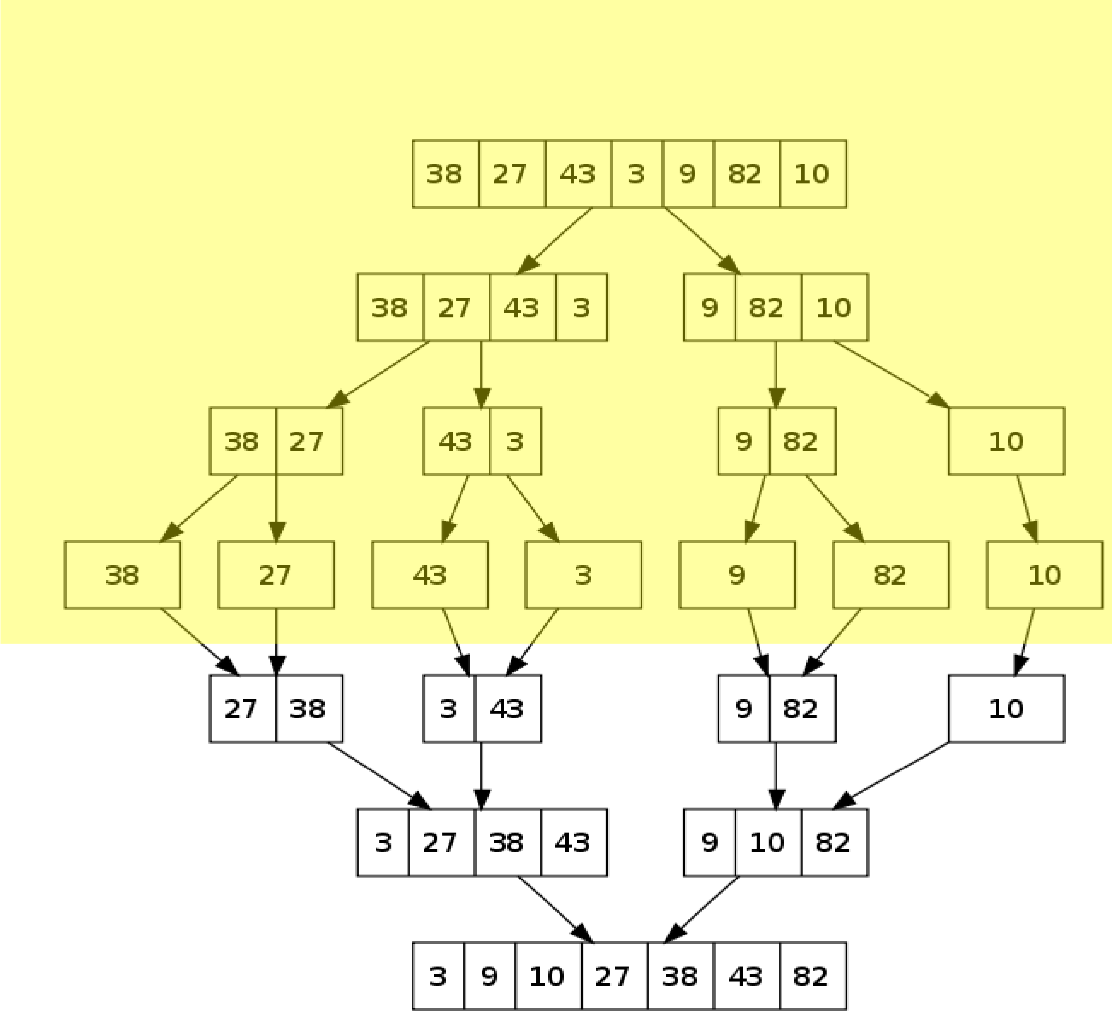
- Will there be a reason why we want to use InsertionSort rather than MergeSort?

MergeSort vs InsertionSort

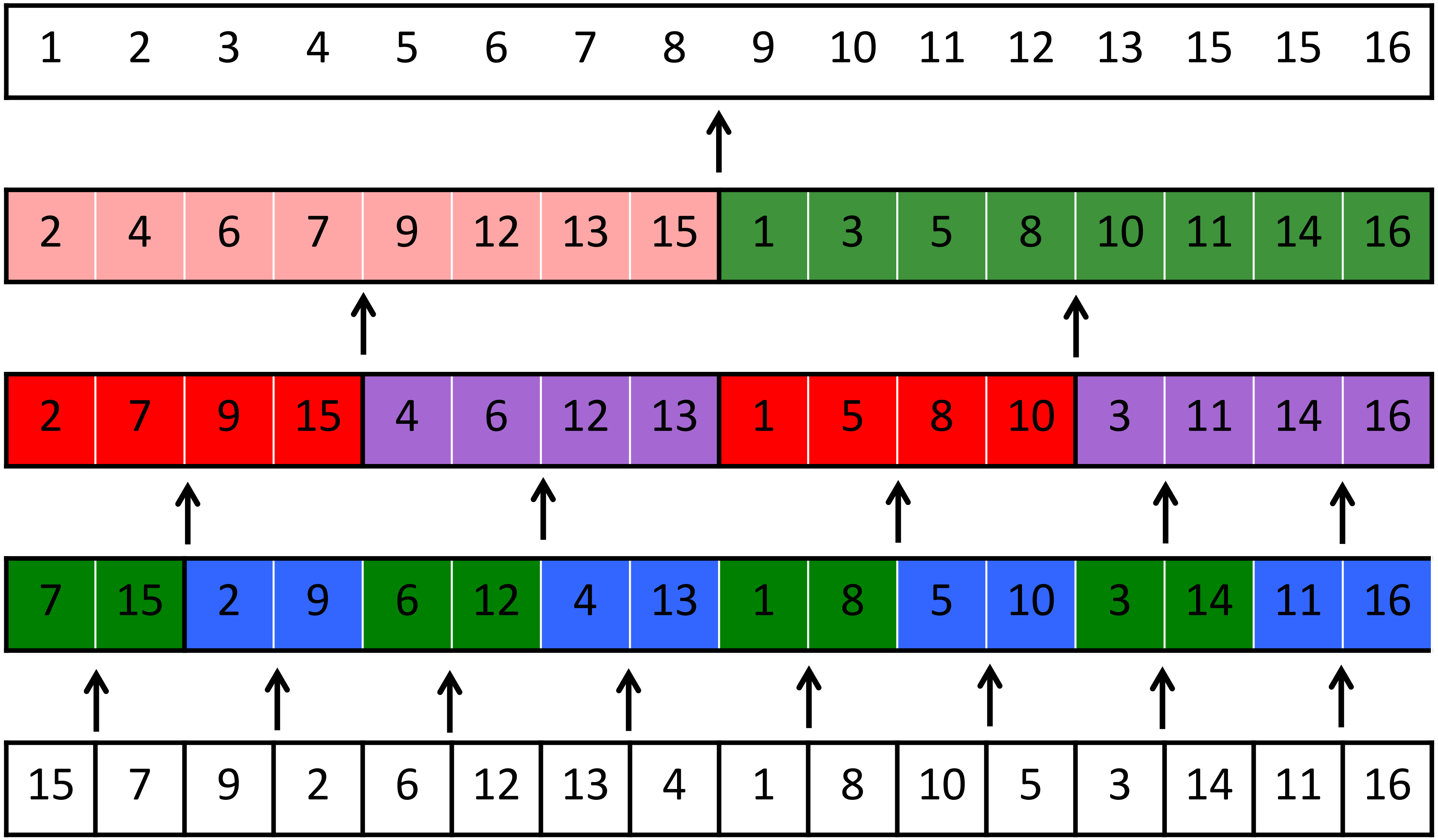
- InsertionSort is faster when
  - When the array is almost sorted
  - When the array is small
- Because MergeSort needs
  - Caching performance, branch prediction, etc.
- In Practice:
  - Inside MergeSort, use InsertionSort when $n < 1000$ instead

Space Time Analysis?

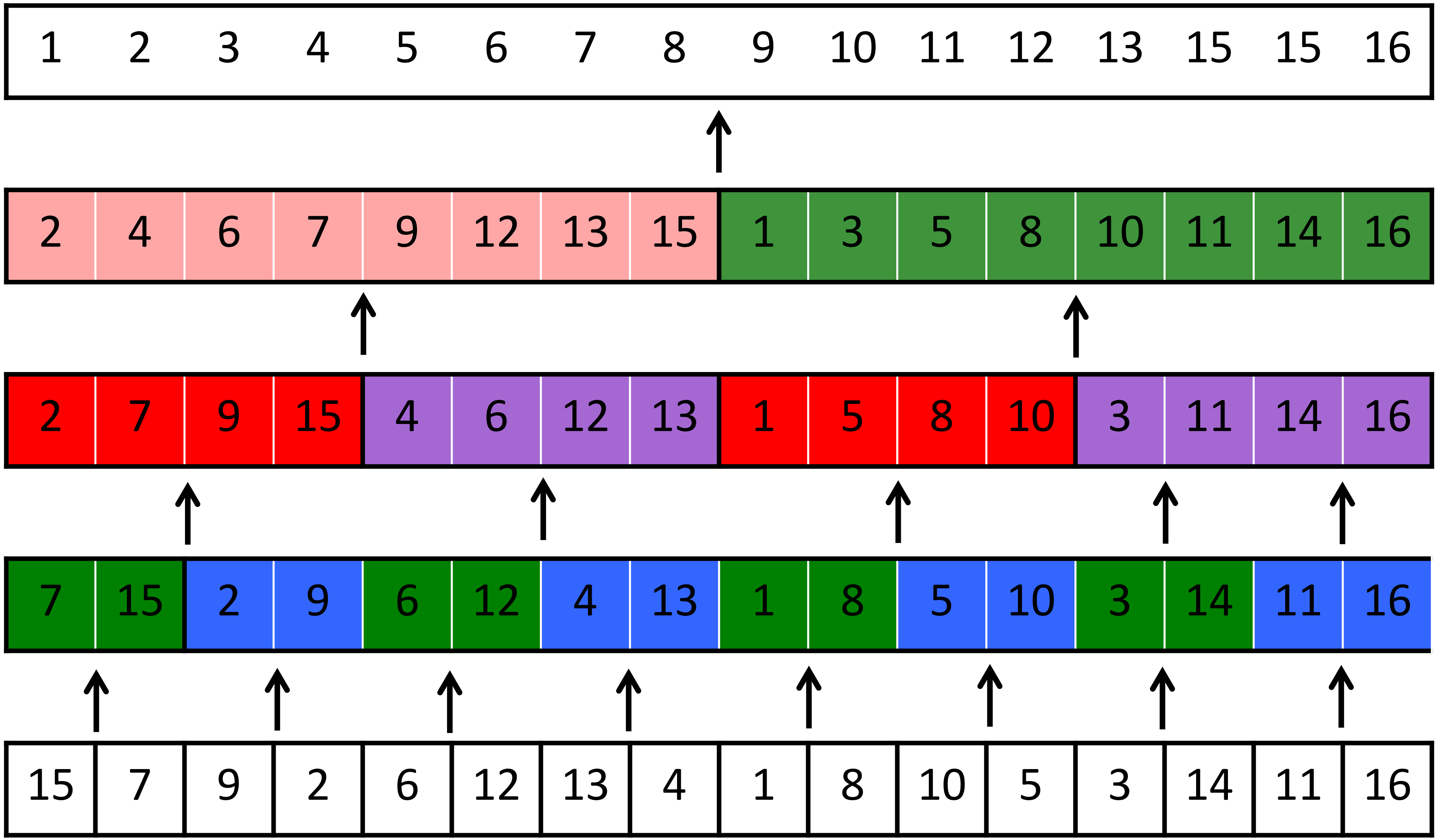
- How about space usage for MergeSort?
- We noticed that we don't need extra space for BubbleSort, InsertionSort and SelectionSort
- If all these stay in the memory, how much space do we need?
  - O (n log n)   (!!!)
- Can we do better?



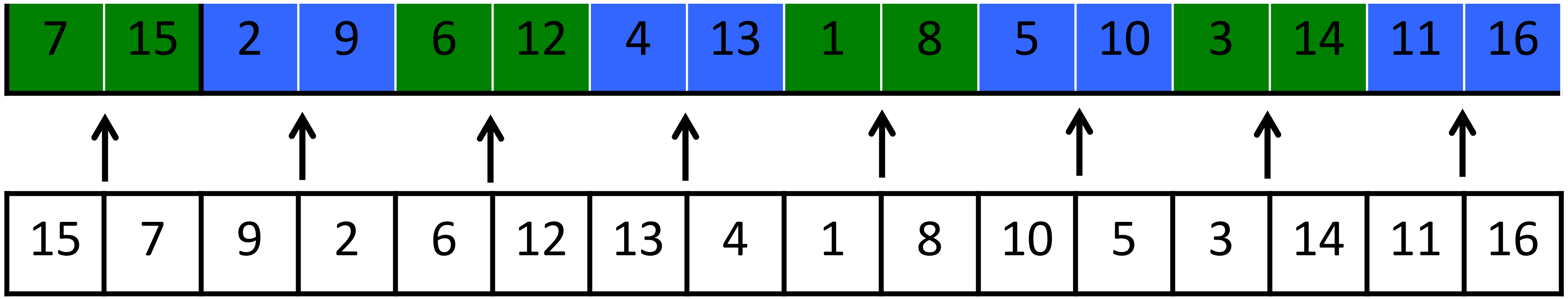
We Thought We need to keep all of these



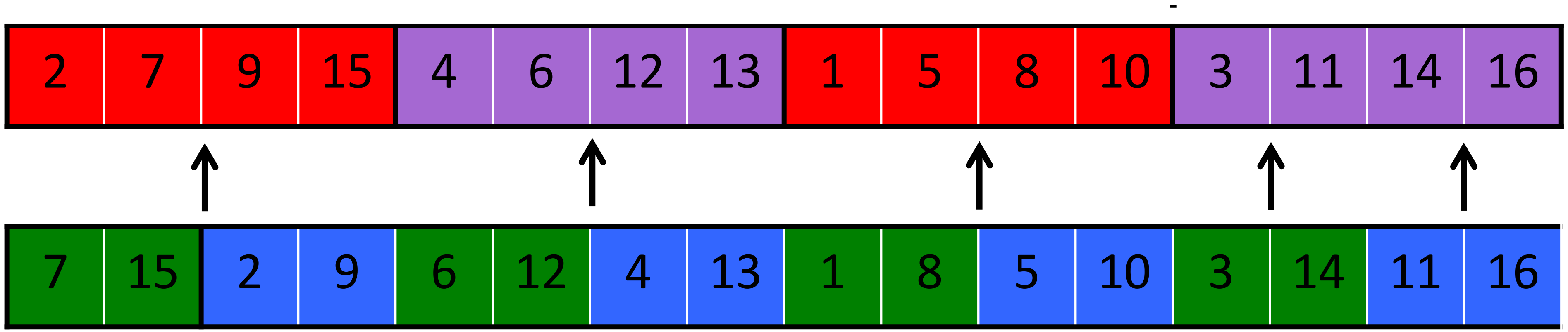
Actually, we only need these TWO AT A TIME



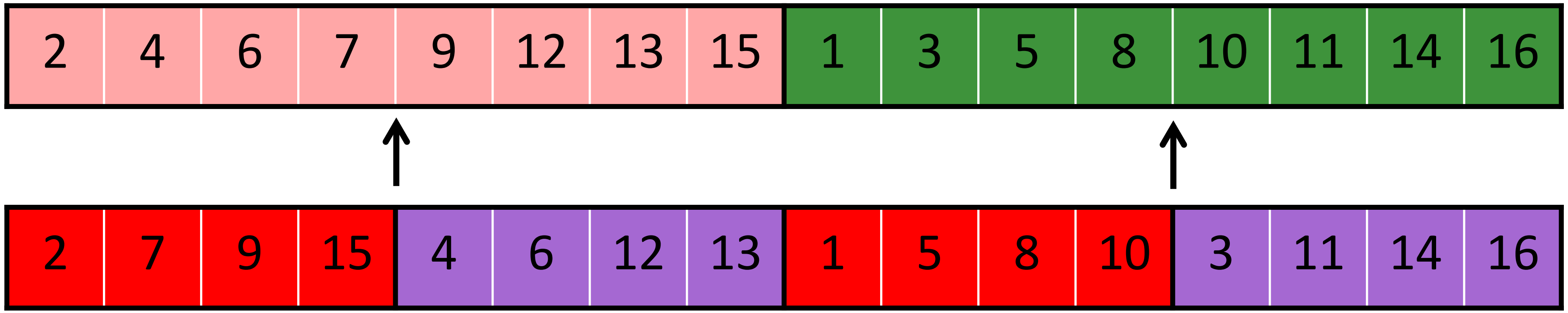
Actually, we only need these TWO AT A TIME



Actually, we only need these TWO AT A TIME



Actually, we only need these TWO AT A TIME



Actually, we only need these TWO AT A TIME

1	2	3	4	5	6	7	8	9	10	11	12	13	15	15	16

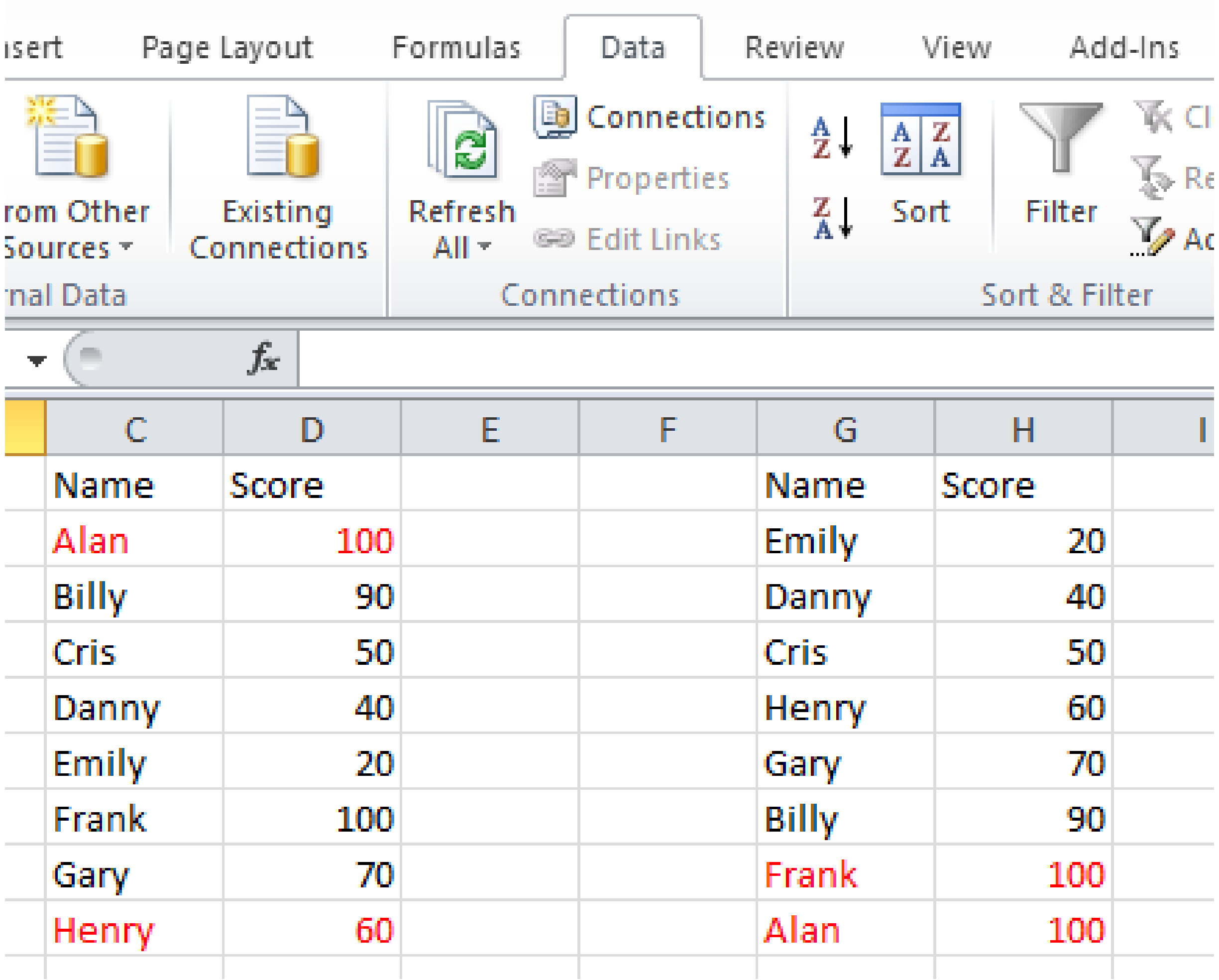
2	4	6	7	9	12	13	15	1	3	5	8	10	11	14	16

Can we do EVEN better?

Properties of Sorting Algorithms

Property of Sorting Algorithms: **Stability**

- When you sort elements that allow duplicates
- Will the “original” order be preserved?
- E.g. sorting the records on the right in Excel by scores, will the order of “Alan” and “Henry” always be preserved?



C	D	E	F	G	H
Name	Score			Name	Score
Alan	100			Emily	20
Billy	90			Danny	40
Cris	50			Cris	50
Danny	40			Henry	60
Emily	20			Gary	70
Frank	100			Billy	90
Gary	70			Frank	100
Henry	60			Alan	100

Stable Sort

Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

STABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Data	a	b	g	h	**C**	**D**	j	k	l	m

Unstable Sort

Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

⇩ UNSTABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Data	a	b	g	h	**D**	**C**	j	k	l	m

Which ones are stable?

- BubbleSort
- InsertionSort
- SelectionSort
- MergeSort

BubbleSort (Stable)

```
BubbleSort(A, n)
  repeat until no more swapping
   for j ←1 to n - 1
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

InsertionSort (Stable)

```
InsertionSort(A, n)
    for  j ← 2 to n
     key ← A[j]
     i ← j-1
     while (i > 0) and (A[i] > key)
          A[i+1] ← A[i]
          i ← i-1
     A[i+1] ← key
```

SelectionSort (Unstable)

```
SelectionSort(A, n)
   for j ← 1 to n - 1:
     find index k s.t. A[k] is the smallest in A[j..n]
     swap(A[j], A[k])
```

- Thank of a case that is not stable

MergeSort (Stable)

```
MergeSort(A, n)
   if (n=1) then return;
   else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```

- Challenge: How to we make sure that it is stable?

Summary

Name	Best Case	Average Case	Worst Case	Memory	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No*
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n \log n$	$n \log n$	$n \log n$	N	Yes

*Stable with O(n) extra space