

Recursion

Reduce and Conquer

Divide (Evenly) and Conquer

Sorting

Recursive algorithms - Divide and Conquer

- General idea:
 - **Divide** a large problem into **smaller** ones
 - Solve each **smaller one** *recursively*
 - **Combine** the solutions of smaller ones to form a solution for the original problem
- Examples:
 - Tower of Hanoi (Reduced and Conquer)
 - Fibonacci sequence (Reduced and Conquer, Dynamic Programming)
 - Euclidean's Algorithm (GCD) (Reduced and Conquer)
 - Merge Sort, Quick Sort (Divide and Conquer)

The Recursive Factorial Function

- The factorial function:

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1 \text{ if } n > 0$$

$$n! = 1 \text{ if } n = 0$$

- Can compute factorial of n if the factorial of $(n-1)$ is known:

$$n! = n * (n-1) !$$

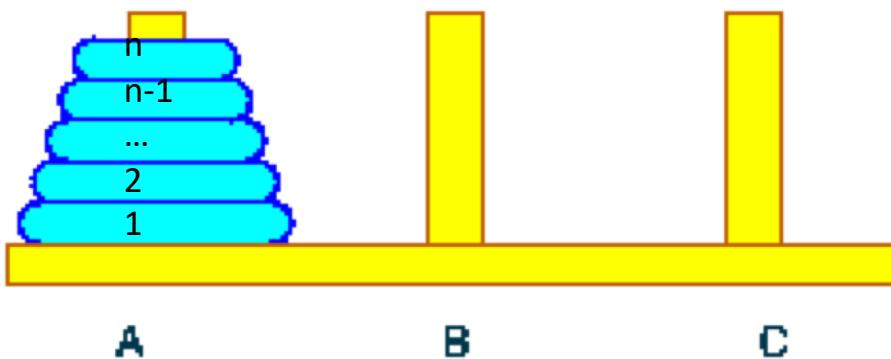
- $n = 0$ is the base case

The Recursive Factorial Function

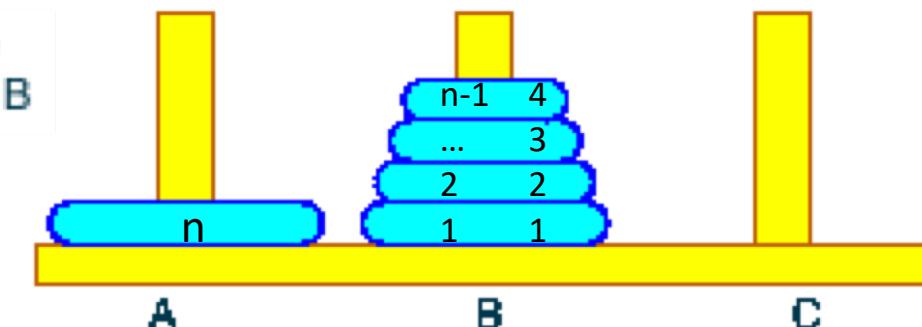
```
def factorial (num):  
{  
    if (num > 0)  
        return num * factorial(num - 1)  
    else  
        return 1  
}
```

- To move n discs from peg A to peg C:

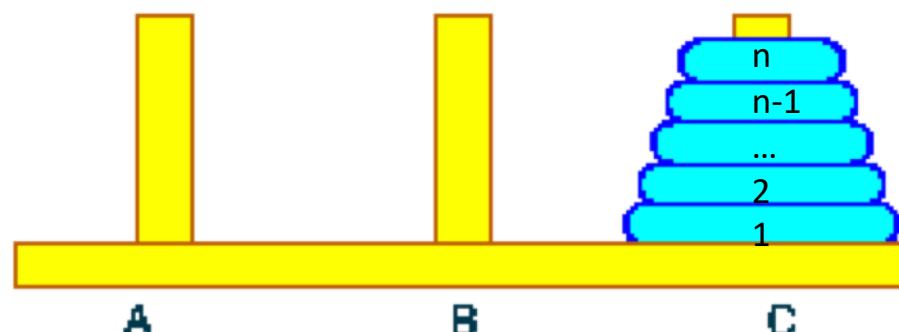
- move $n-1$ discs from A to B. This leaves disc n alone on peg A
- move disc n from A to C
- move $n-1$ discs from B to C so they sit on disc n



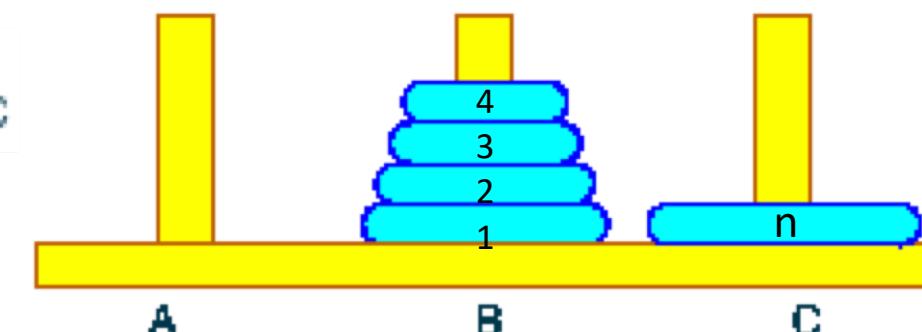
Recursively move
four rings from A to B

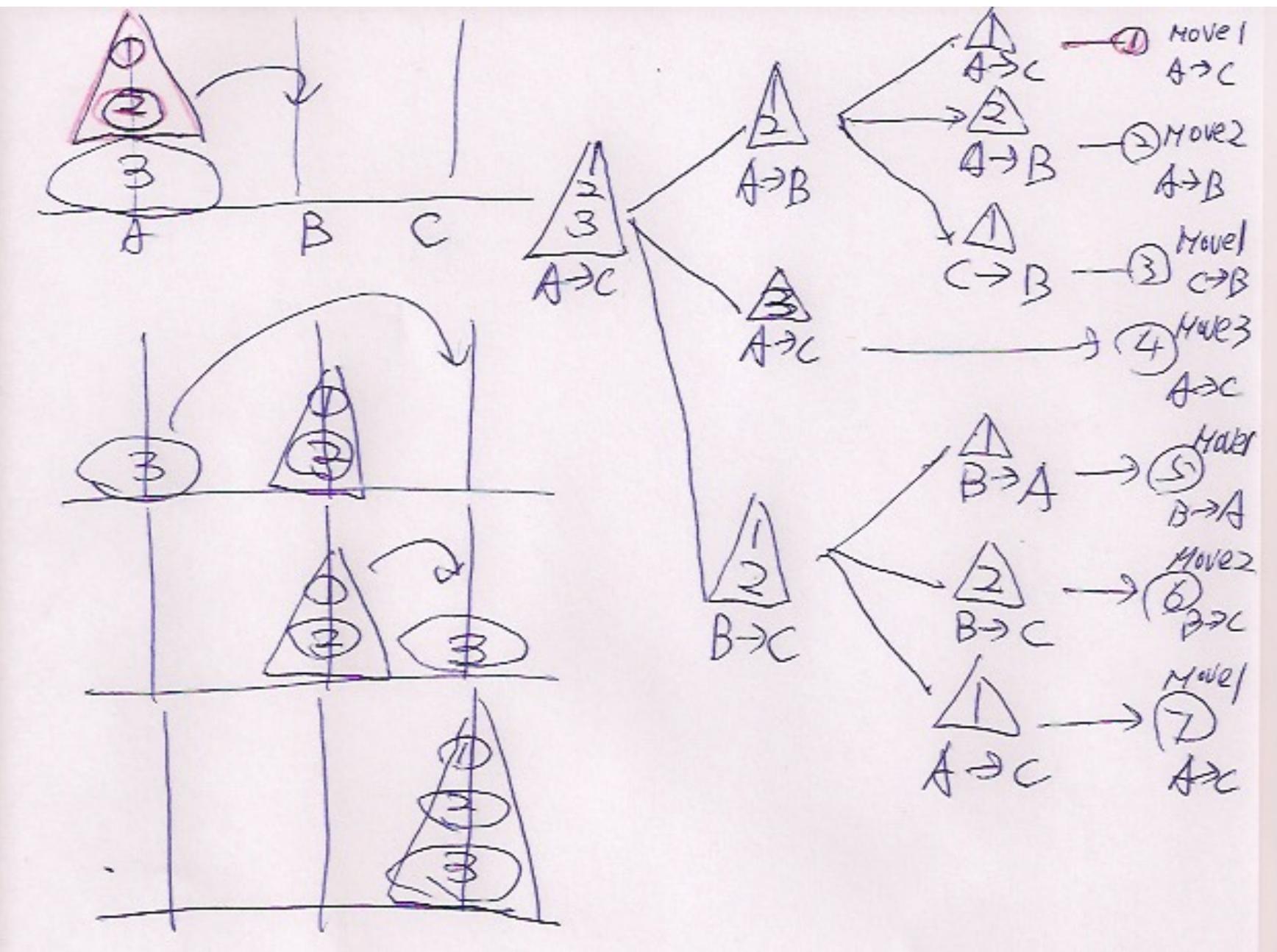


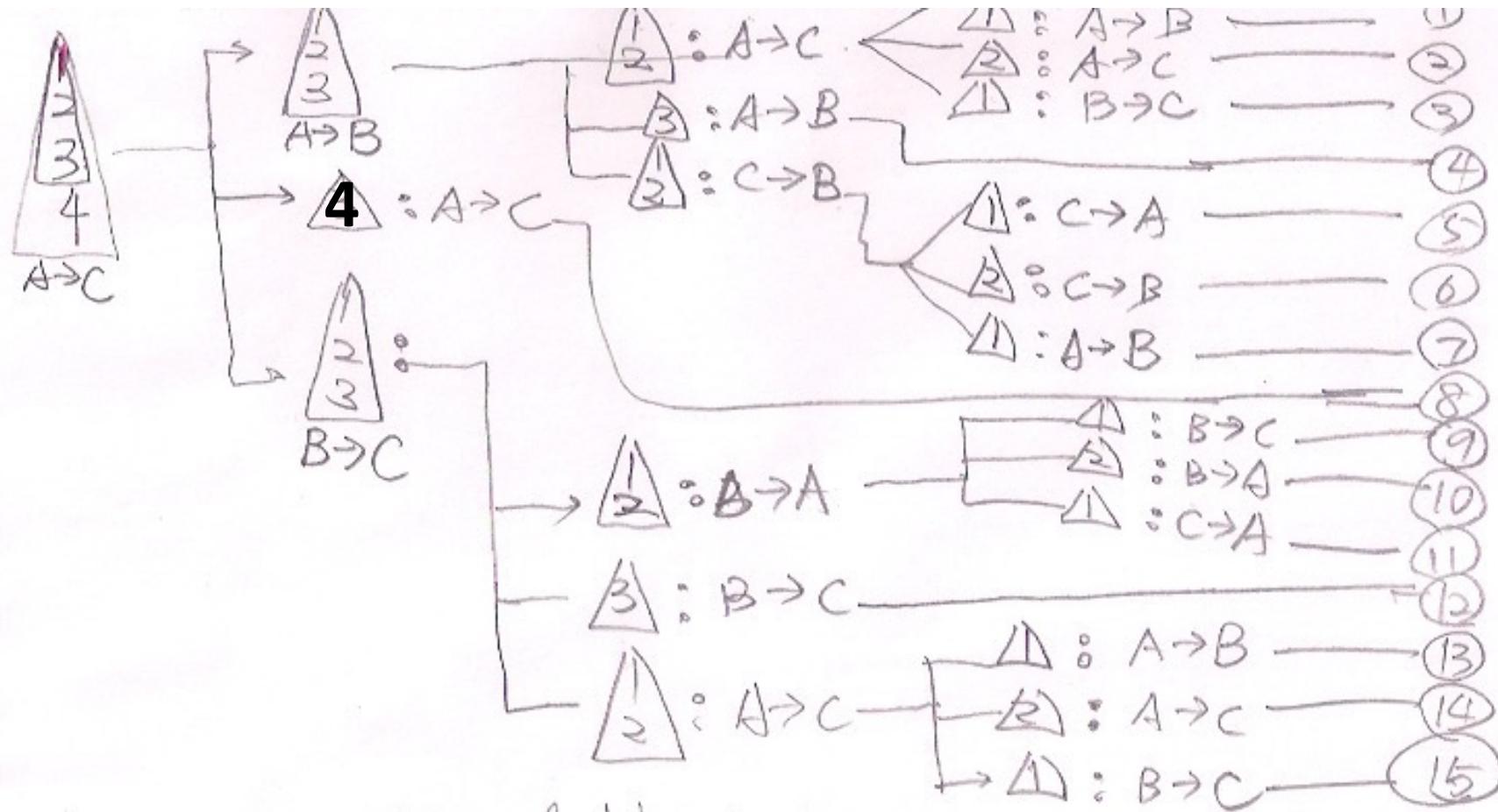
move one ring



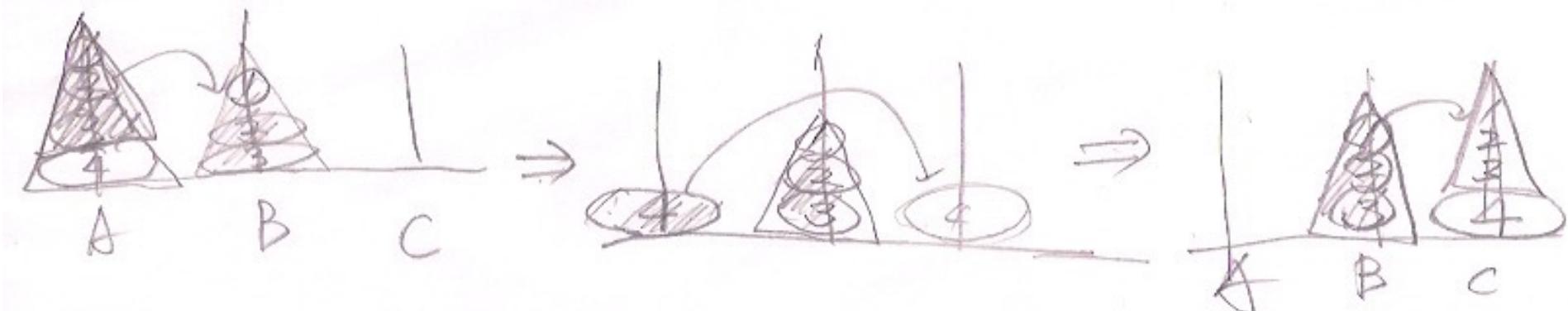
Recursively move
four rings from B to C







Tower of Hanoi



Recursive Algorithms for Tower of Hanoi

```
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):  
    if n == 0:  
        return  
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)  
    print ("Move disk",n,"from",from_rod,"to",to_rod )  
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
```

```
TowerOfHanoi(6, "A","B","C")
```

$$T(n) = \begin{cases} 0 & \text{if } n = 0; \\ 2T(n-1) + 1 & \text{if } n > 1. \end{cases}$$

Recurrence Function

$$T(n) = \begin{cases} 0 & \text{if } n = 0; \\ 2T(n-1) + 1 & \text{if } n > 1. \end{cases}$$

$$\begin{aligned} T(n) &= 2 * T(n-1) + 1 \\ &= 2 * (2 * T(n-2) + 1) + 1 = 4 * T(n-2) + 3 = 2^2 * T(n-2) + 2^2 - 1 \\ &= 4 * (2 * T(n-3) + 1) + 3 = 8 * T(n-3) + 7 = 2^3 * T(n-3) + 2^3 - 1 \\ &= 8 * (2 * T(n-4) + 1) + 7 = 16 * T(n-4) + 15 = 2^4 * T(n-4) + 2^4 - 1 \\ &\dots &= \dots &= \dots \\ &= &= &= 2^n * T(0) + 2^n - 1 \\ &= 2^n - 1 \end{aligned}$$

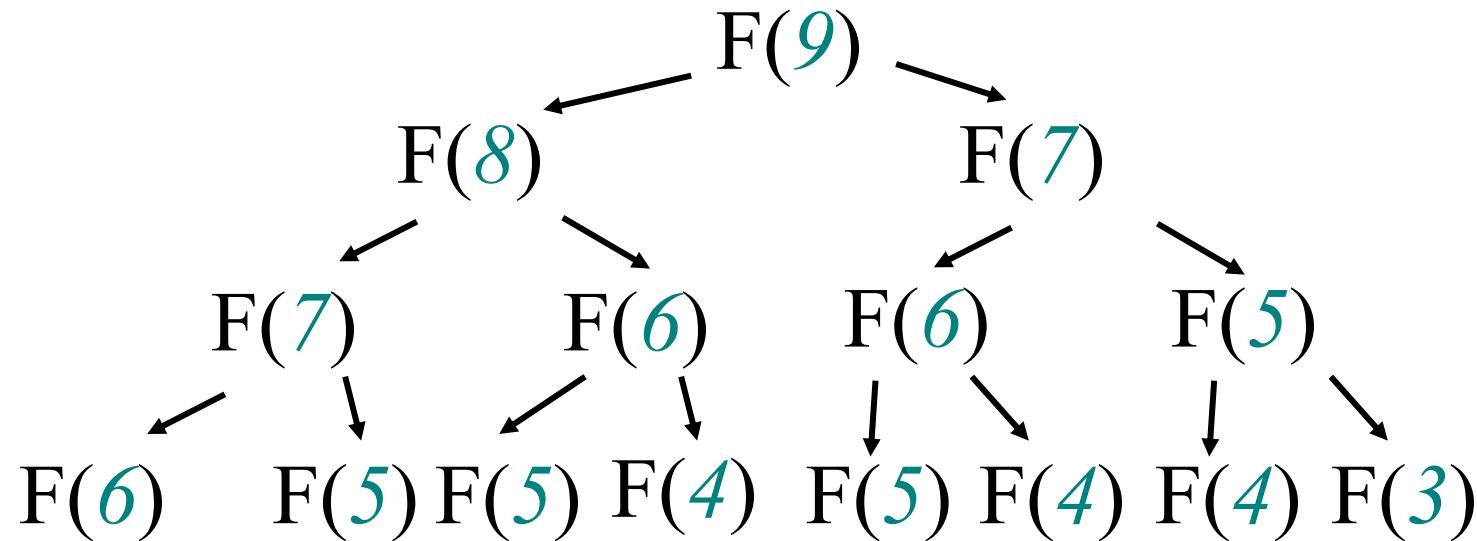
Fibonacci numbers

An recursive algorithm

fib(n) // recursive

$$T(n) = T(n-1) + T(n-2) + 1$$

$$T(n) = \Theta(F_n) = \Theta(\phi^n)$$



Fibonacci numbers

An **iterative** algorithm

`fib(n): // iterative`

Time complexity? $T(n)$

The Recursive gcd Function

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:

$\text{gcd}(x, y) = y$ if y divides x evenly

$\text{gcd}(x, y) = \text{gcd}(y, x \% y)$ otherwise

- $\text{gcd}(x, y) = y$ is the base case

Iteration vs. Recursion

```
def gcd(x, y):  
    while y != 0:  
        (x, y) = (y, x % y)  
    return x
```

```
def gcd_recursive(a, b):  
    if b == 0:  
        return a  
    else:  
        return gcd_recursive(b, a % b)
```

Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
 - + Models certain algorithms most accurately
 - + Results in shorter, simpler functions
 - May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
 - + Executes more efficiently than recursion
 - Often is harder to code or understand

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3 5 7 8 9 12 15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9



Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3 5 7 8 9 12 15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9



Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9



Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, done!
3. else if less than wanted, search right half
4. else search left half

Example: Find 9



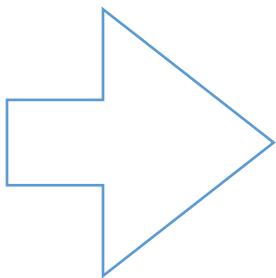
Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, done!
3. else if less than wanted, search right half
4. else search left half

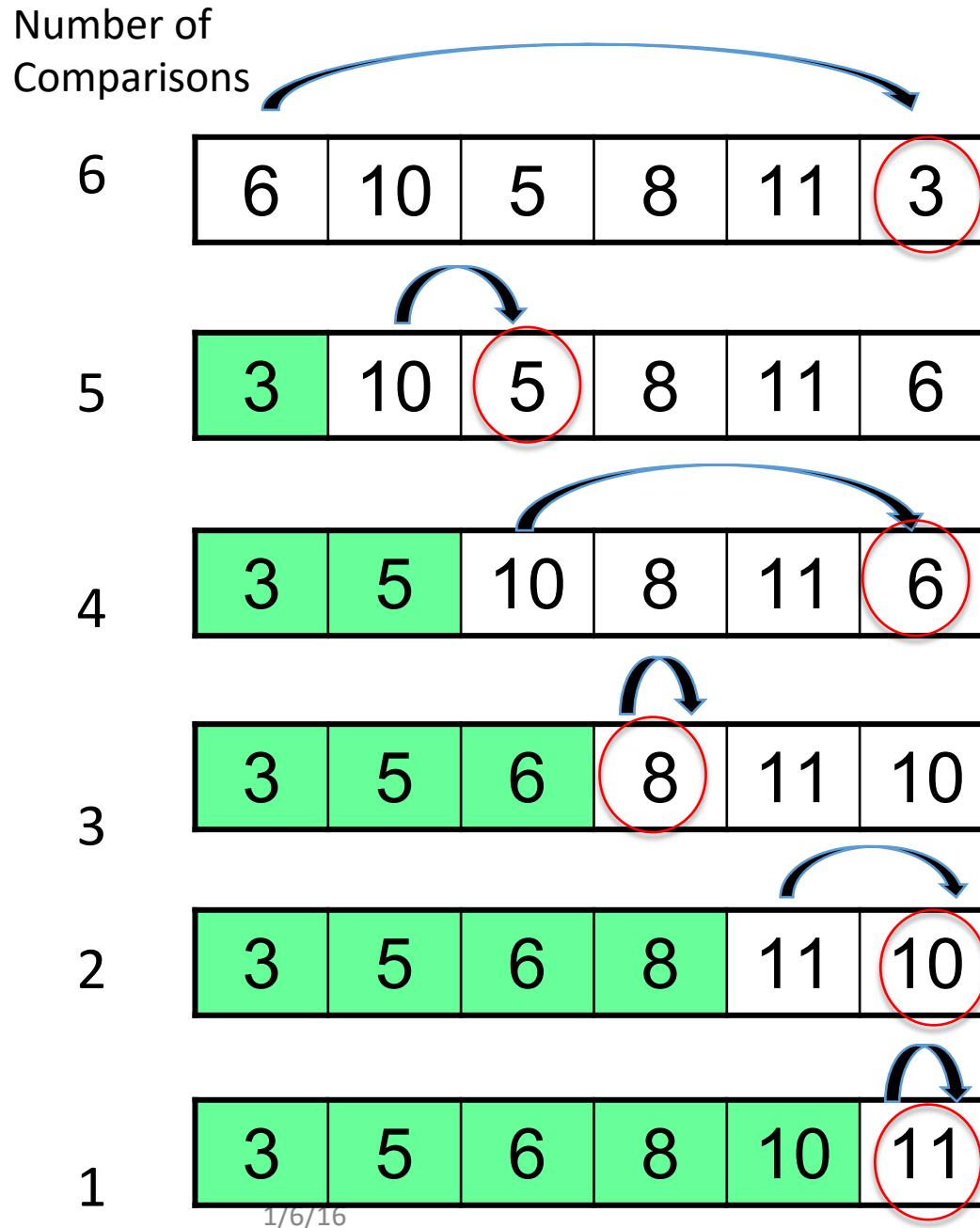
$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Sorting Algorithms



Sorting Algorithms

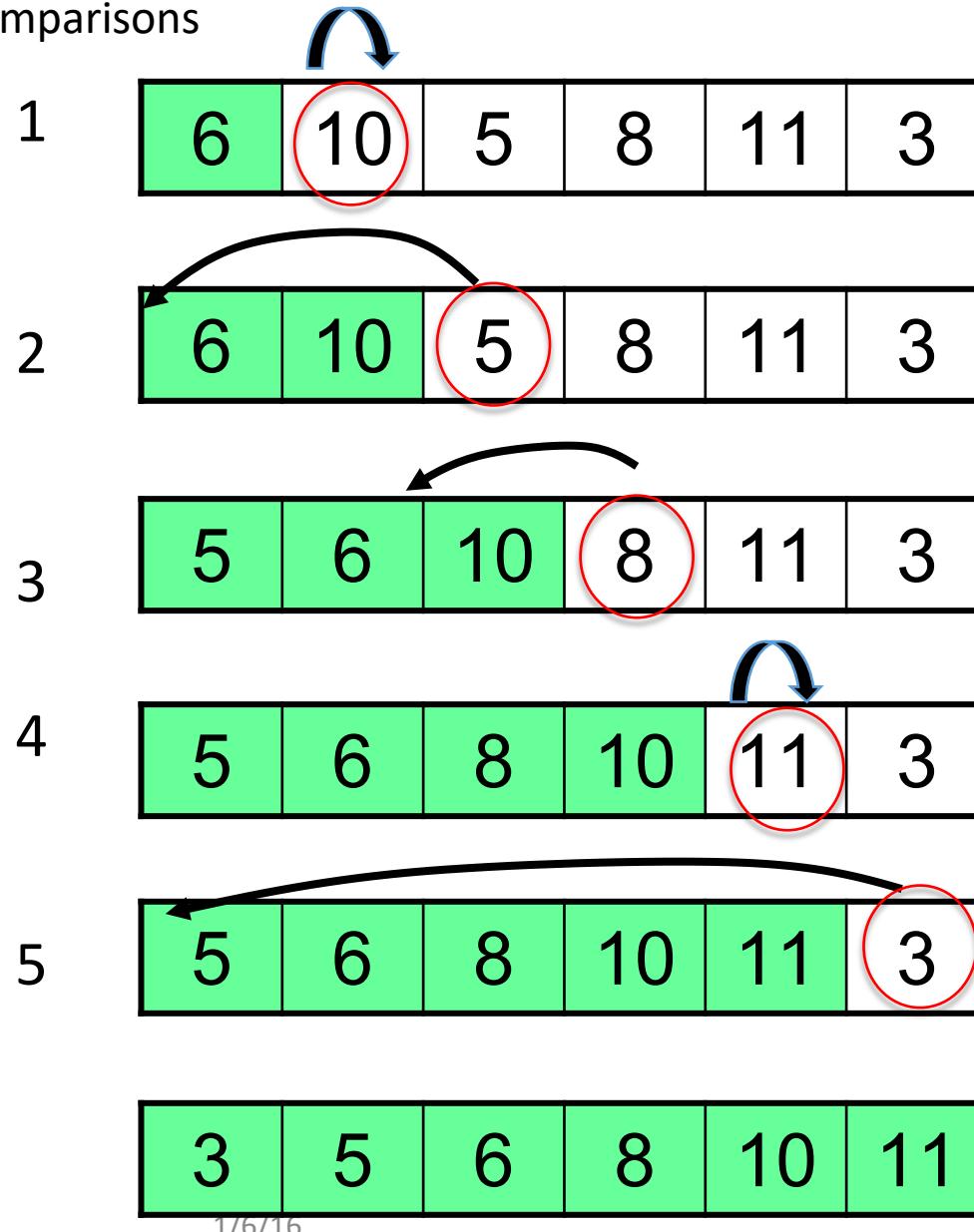
- Selection and Insertion Sort:
 - Incremental approach (Reduced and Conquer)
 - Simple but slow
- Merge and Quick Sort
 - Divide and Conquer



Selection Sort (Incremental)

- Find smallest item, move to 1st location (n comparisons).
- Find next smallest item, move to 2nd location ($n-1$ comparisons).
- ...
- Repeat until the final location is reached.
- Time complexity: $n+(n-1)+(n-2)+\dots+1 = (n^2+n)/2 = O(n^2)$

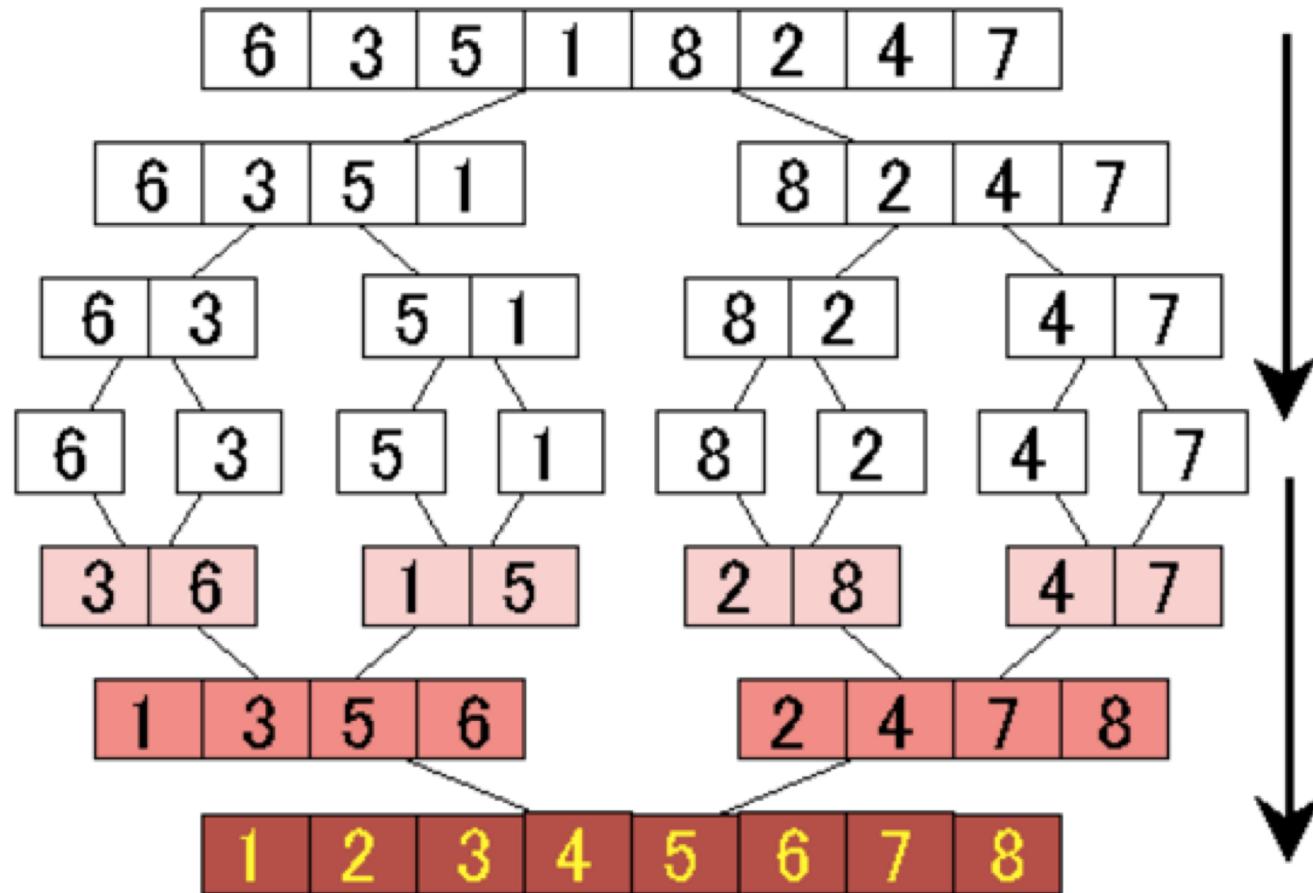
Number of Comparisons



Insertion Sort (Incremental)

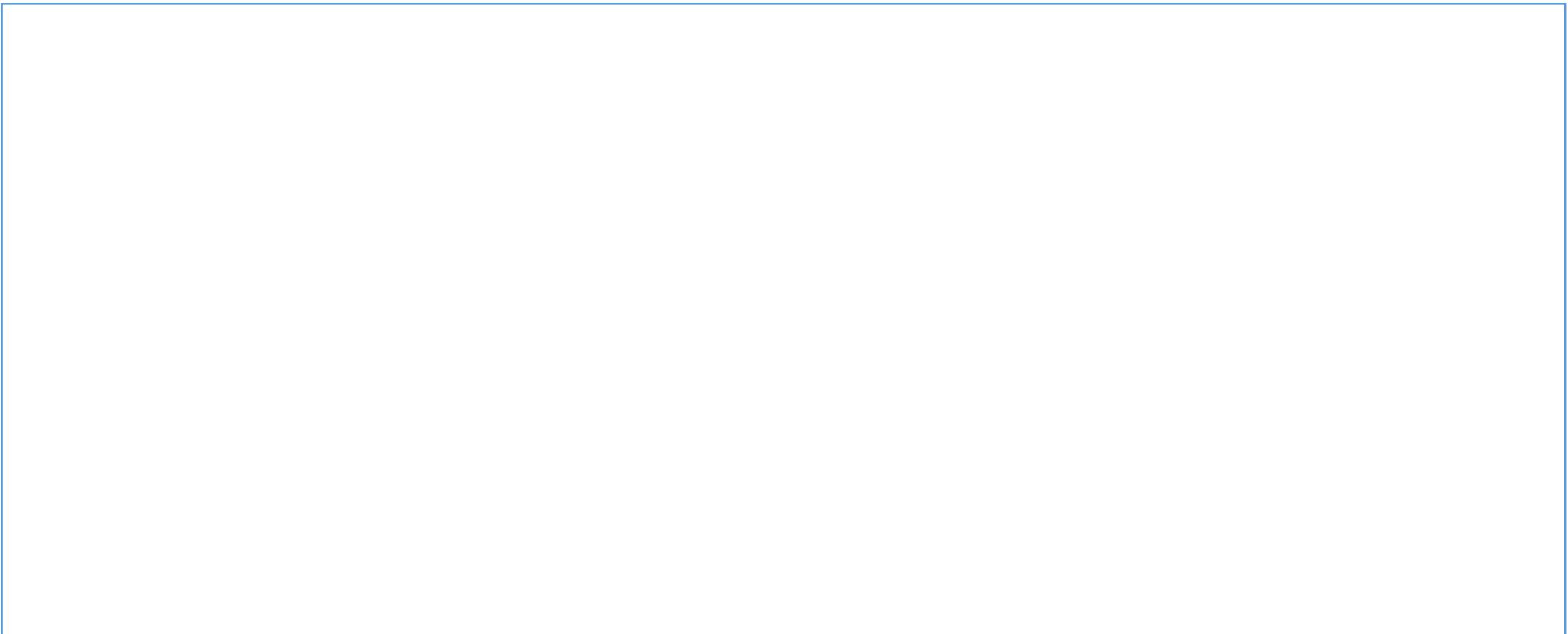
- Begin with 1 sorted item.
- Insert 2nd item properly onto the sorted list
 - 2 sorted items
- Insert 3rd item properly onto the sorted list
 - 3 sorted item
- ...
- Repeat until we have all (n) sorted items.
- Time complexity: $1+2+\dots+(n-1) = (n^2-n)/2 = O(n^2)$

Merge Sort



- Divide-and-conquer
- Recursively sort $\frac{1}{2}$ the set
- Then merge them together: $O(n)$
- $O(n \lg n)$ for both best, average, and worst cases
- Can be made to be in-place

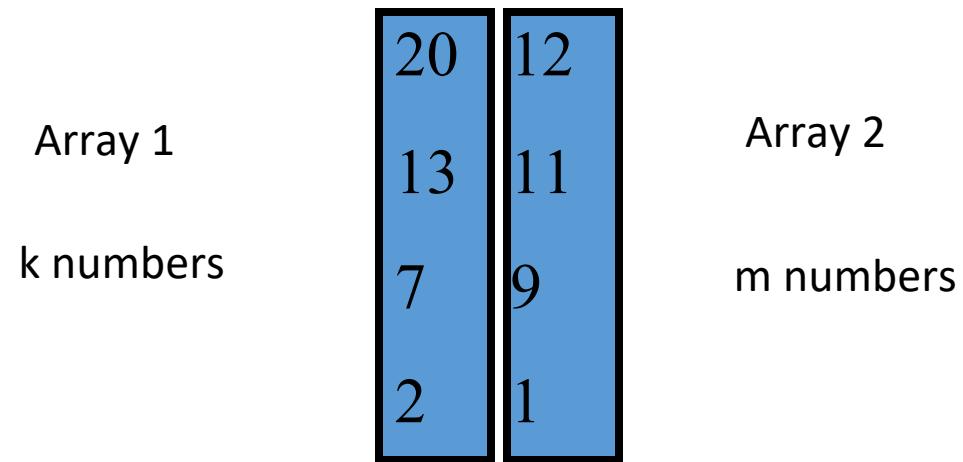
Codes: Merge Sort



Key subroutine: MERGE O(n)



Merging two sorted arrays



Time: $O(k + m)$.

Merging two sorted arrays

20 12

13 11

7 9

2 1

Merging two sorted arrays

20 12

13 11

7 9

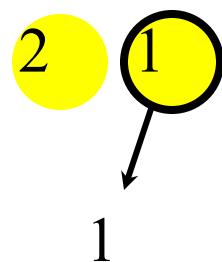
2 1

Merging two sorted arrays

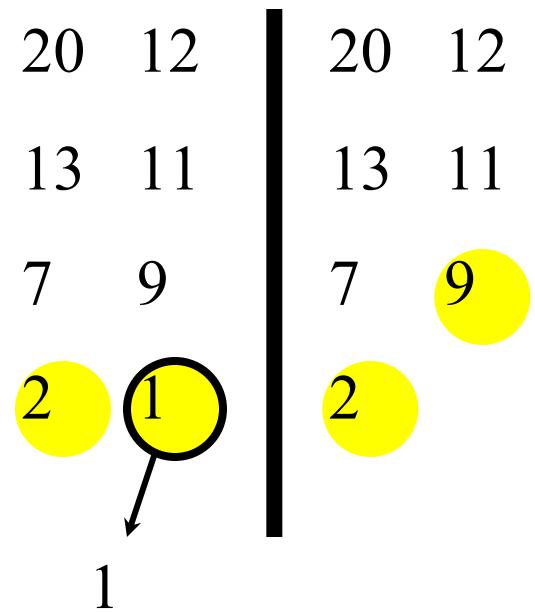
20 12

13 11

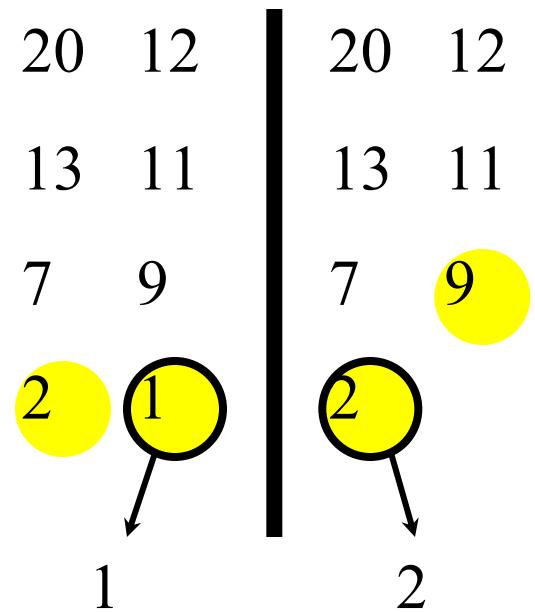
7 9



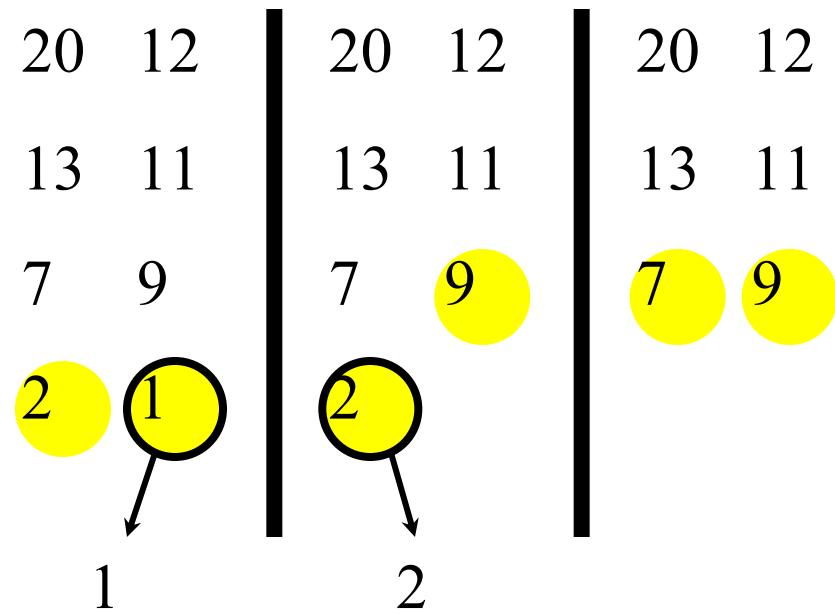
Merging two sorted arrays



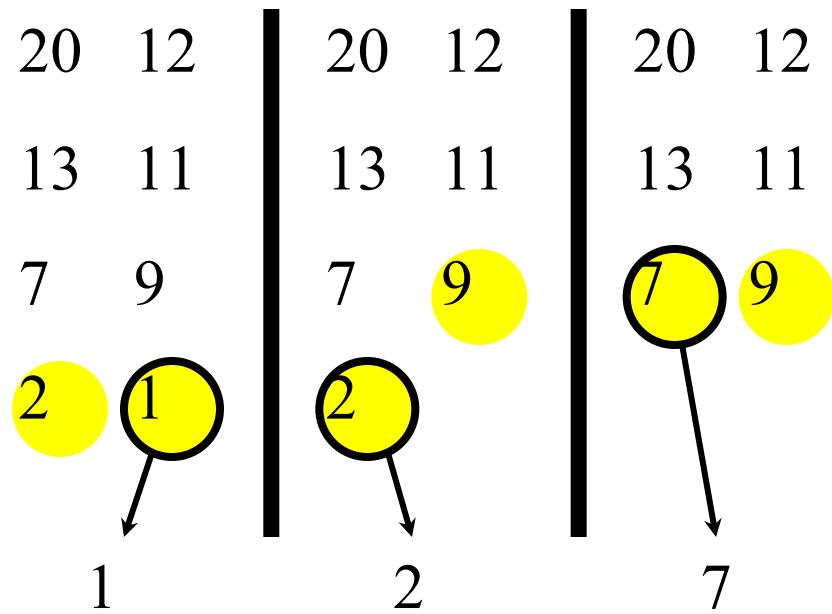
Merging two sorted arrays



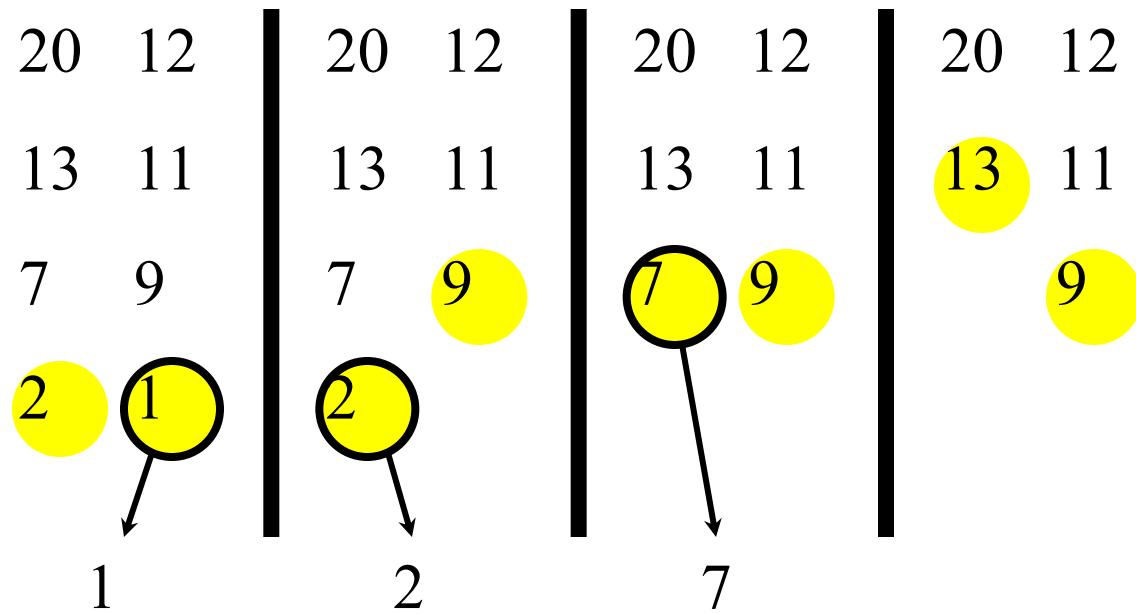
Merging two sorted arrays



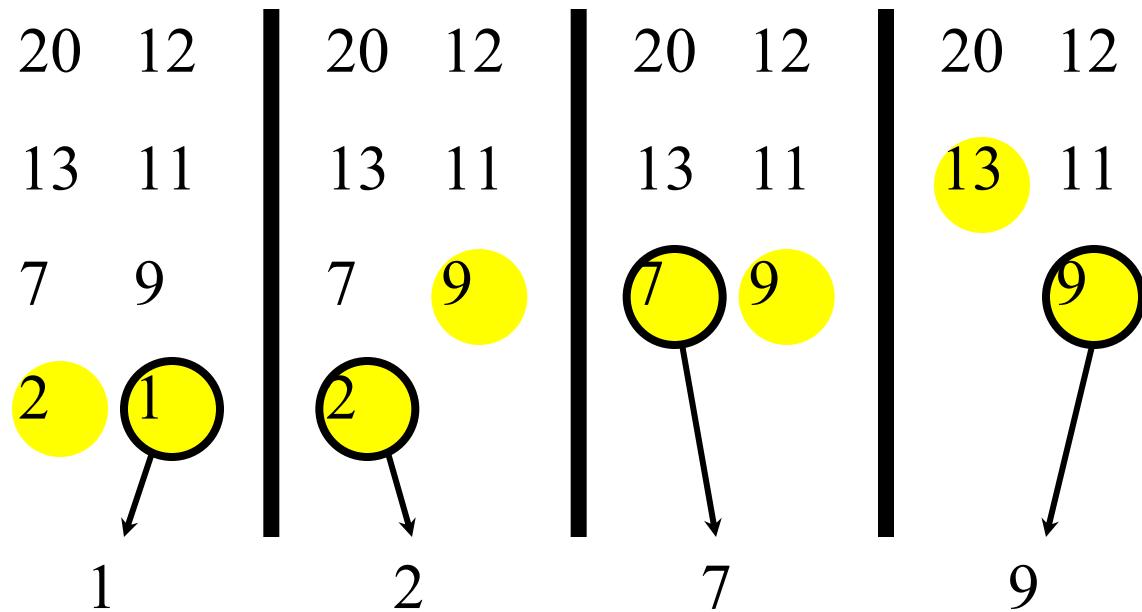
Merging two sorted arrays



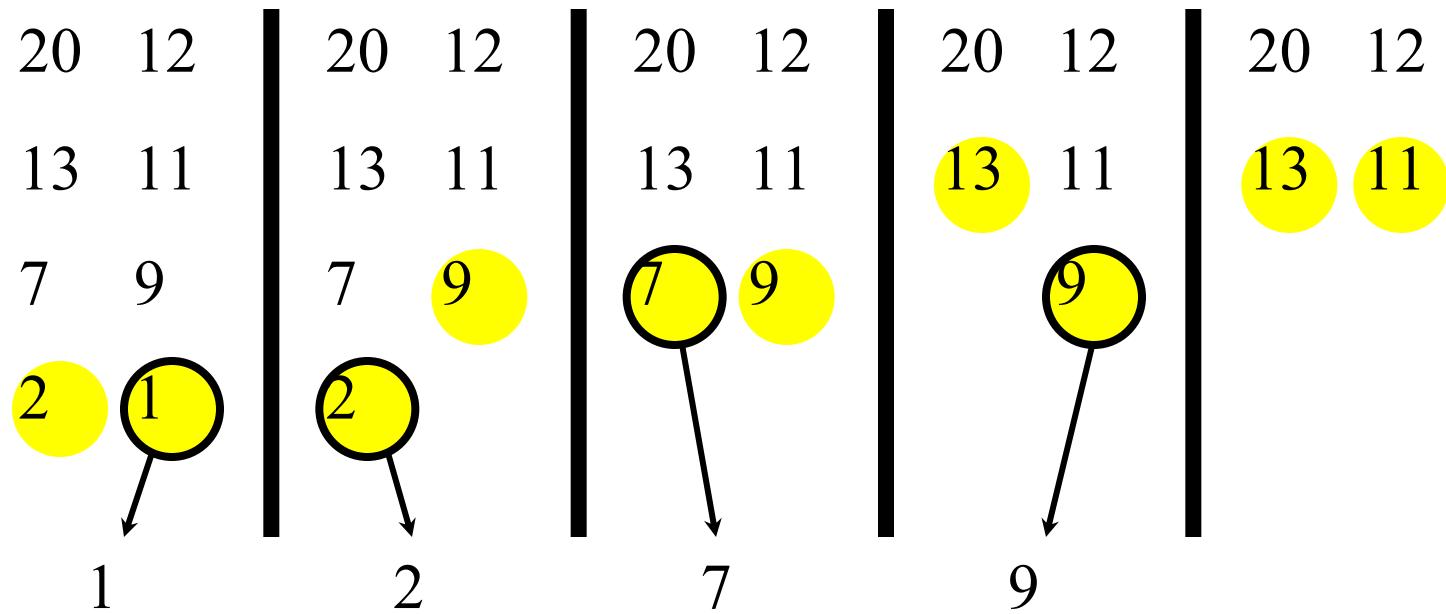
Merging two sorted arrays



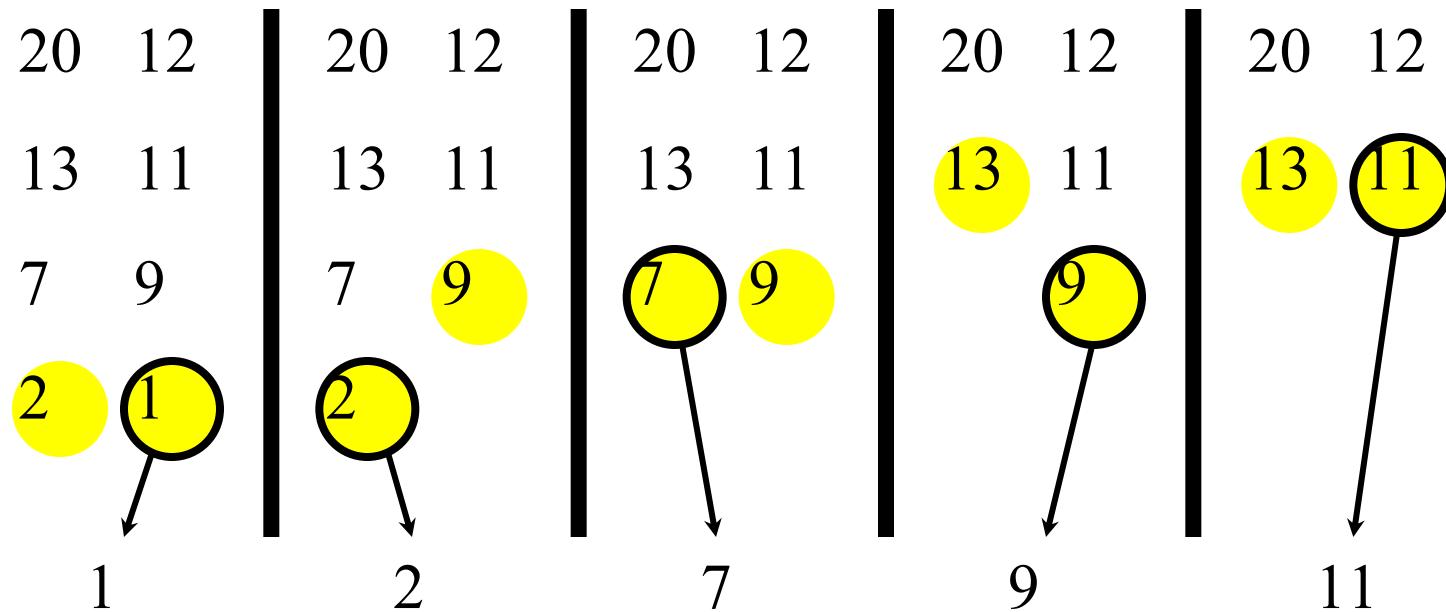
Merging two sorted arrays



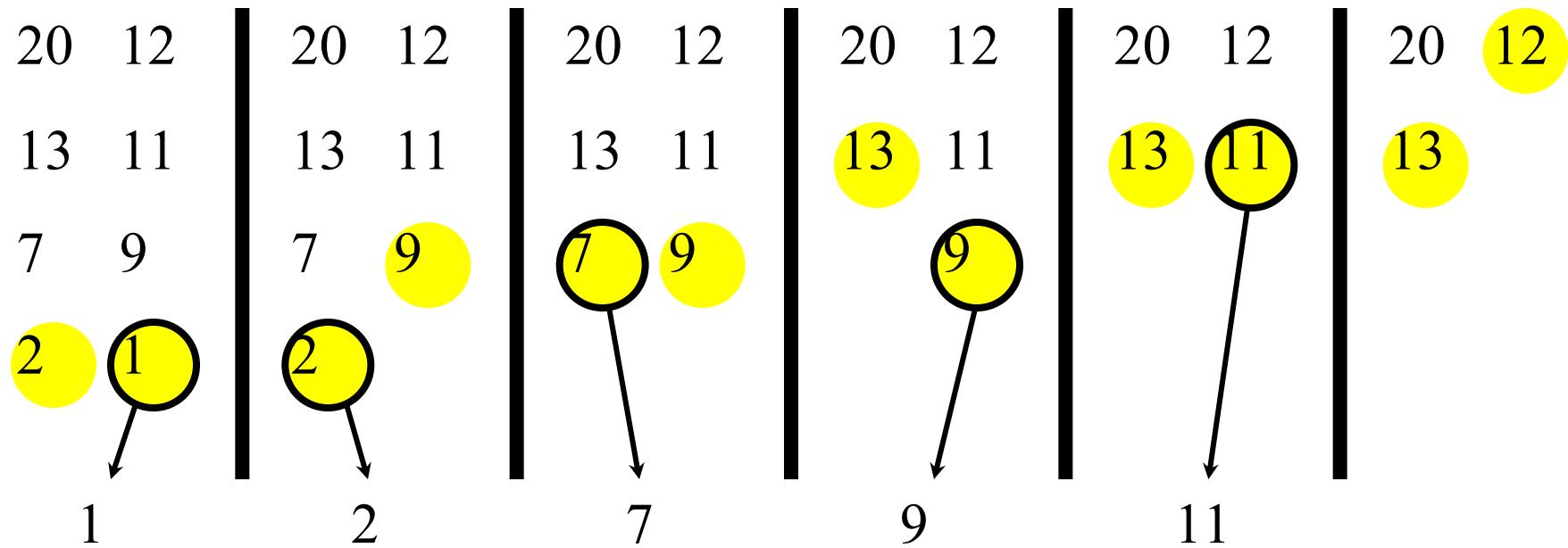
Merging two sorted arrays



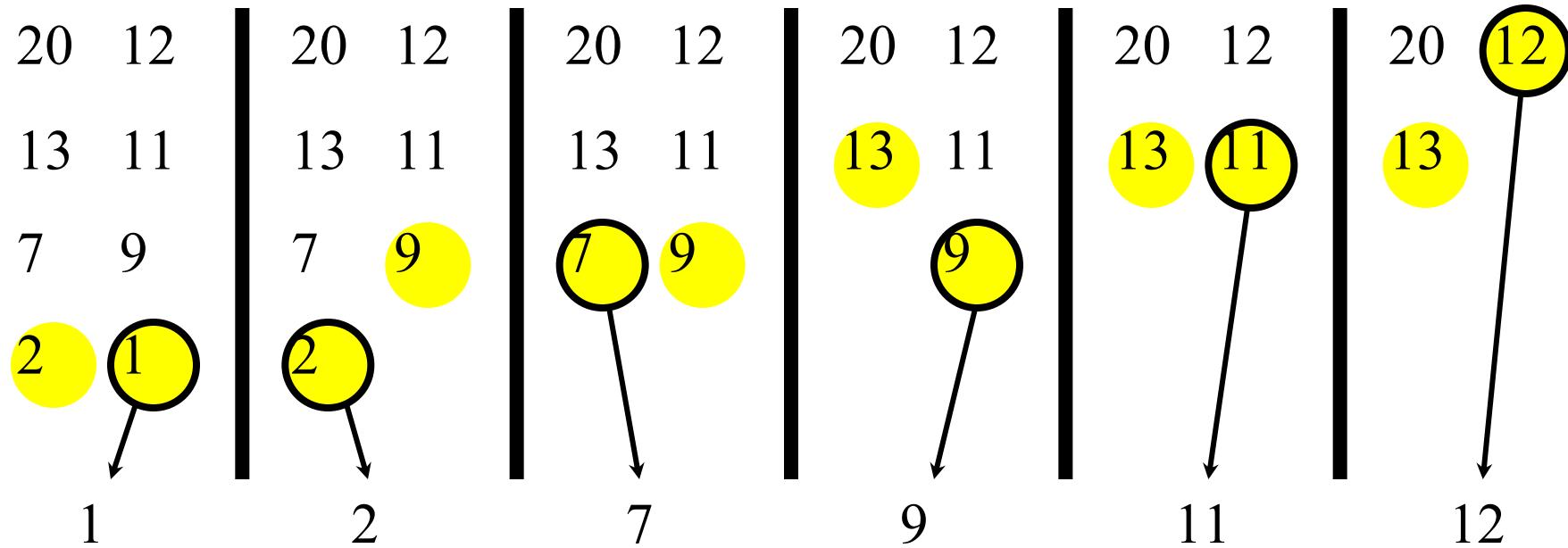
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Analyzing merge sort

```
Def mergesort(data, first, last):  
  
    if last > first:  
        mid = (first+last)//2 ;  
        mergesort(data, first, mid )  
        mergesort(data[], mid+1,last)  
        merge ( data, first, last )
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Recurrence for Merge Sort

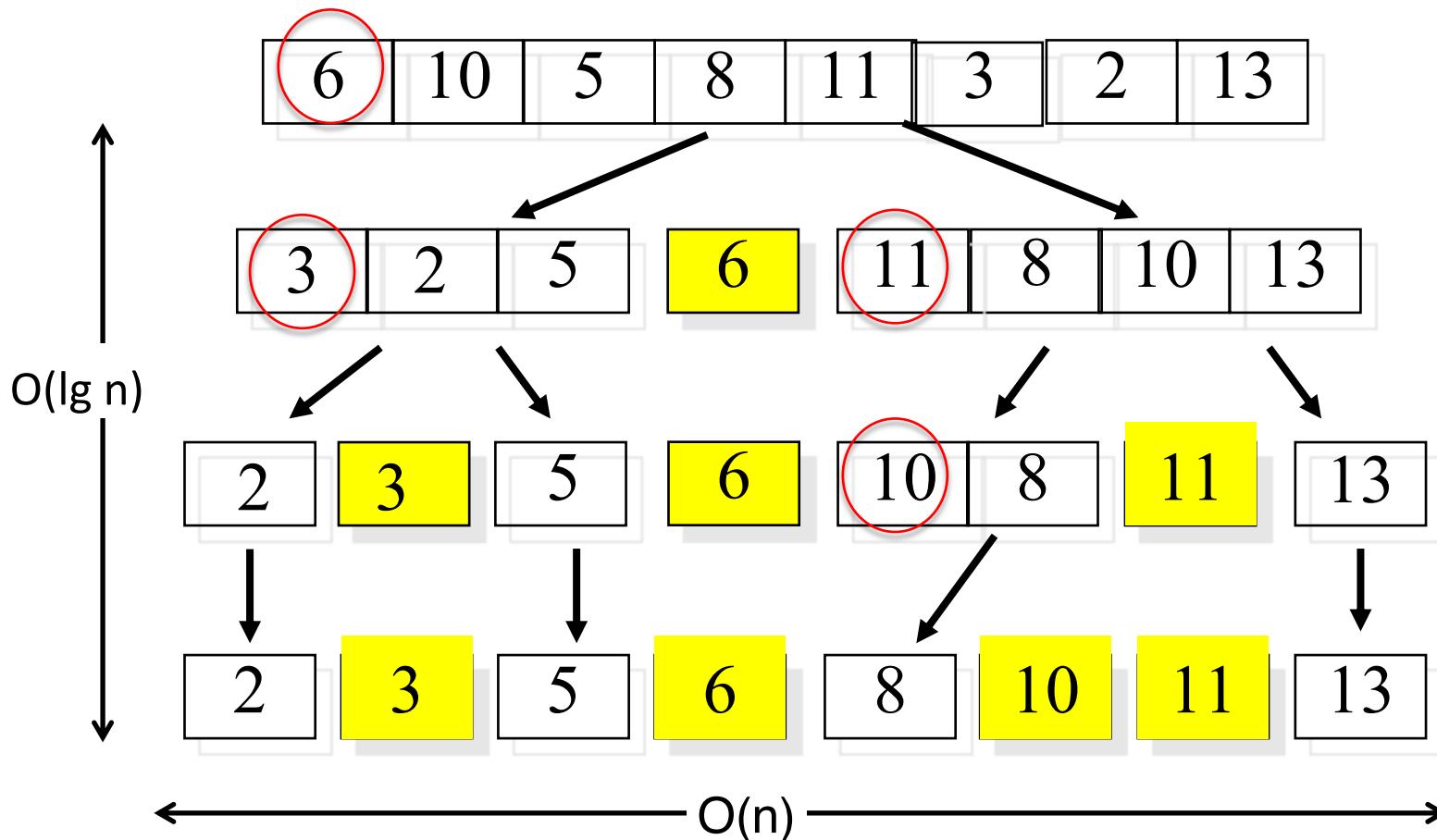
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

$$\begin{aligned} T(n) &= 2 T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n = 4 T(n/4) + 2*n \\ &= 4 (2T(n/8) + n/4) + 2*n = 8 T(n/8) + 3*n \\ &= \dots = 2^k T(n/2^k) + k*n = \\ &= \dots = n * T(1) + k*n, \text{ where } k = \log n \\ &= n + (\log_2 n) * n = O(n \log n) \end{aligned}$$

Quicksort

- Another divide and conquer sorting algorithm
 - like merge sort
- The worst-case and average-case running time
- Learn new algorithm analysis tricks

Quicksort (Divide & Conquer)

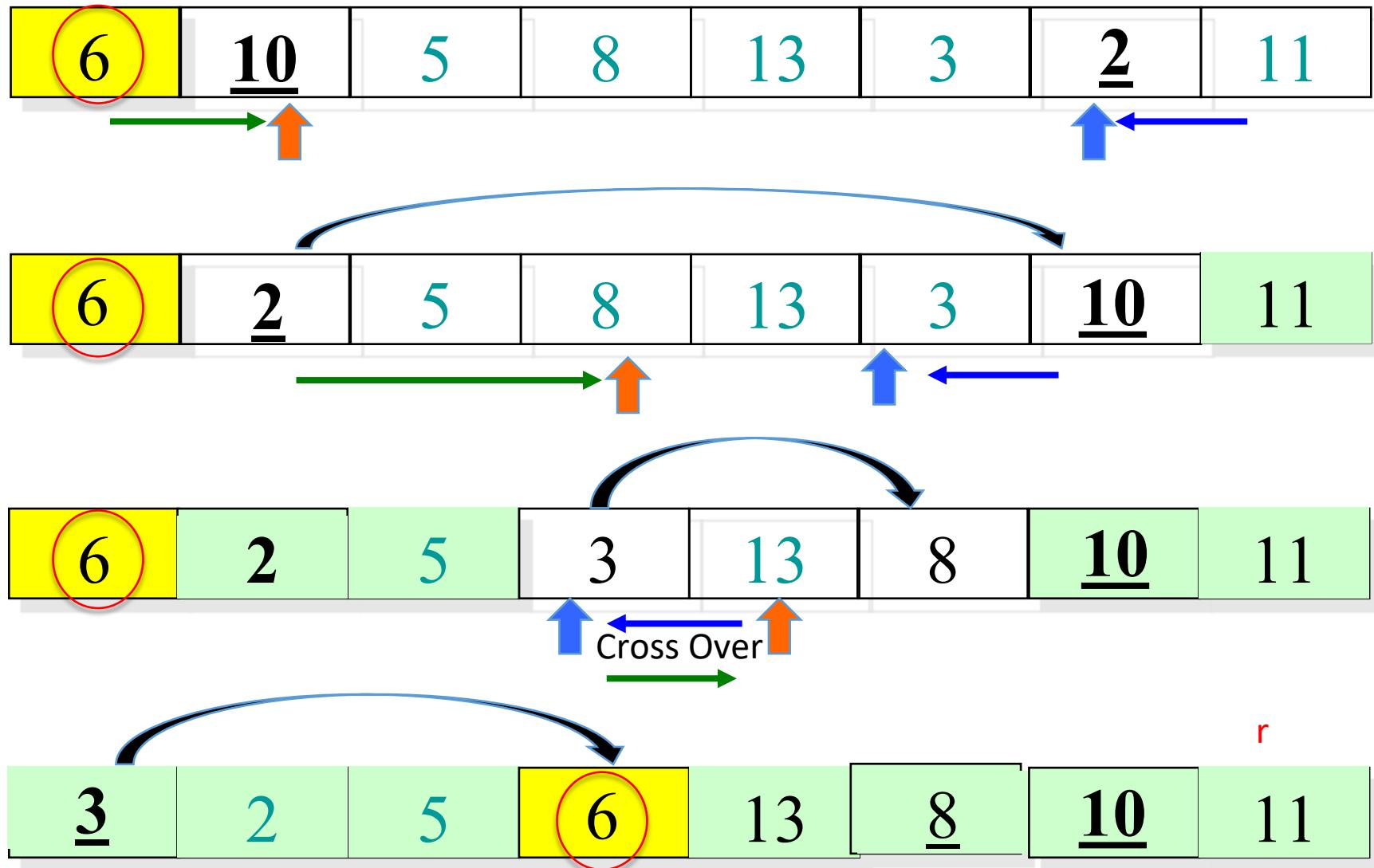


- Pick a pivot element
- Partition items to 2 groups
 - Items $<$ pivot on the left
 - Items $>$ pivot on the right
- Sort the left and the right partitions recursively
 - ...
- Done when problem trivial
- Time complexity: $O(n \lg n)$

Pseudocode for Quicksort

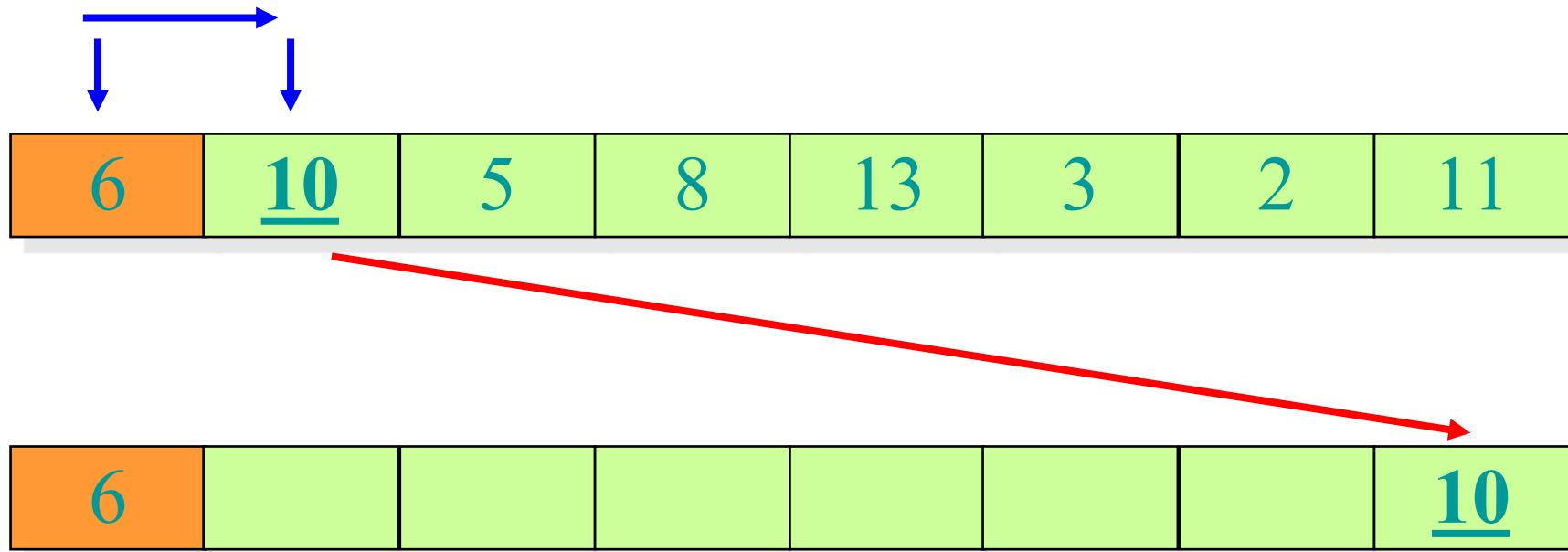


Key Step of Quick Sort: Partitioning O(n)



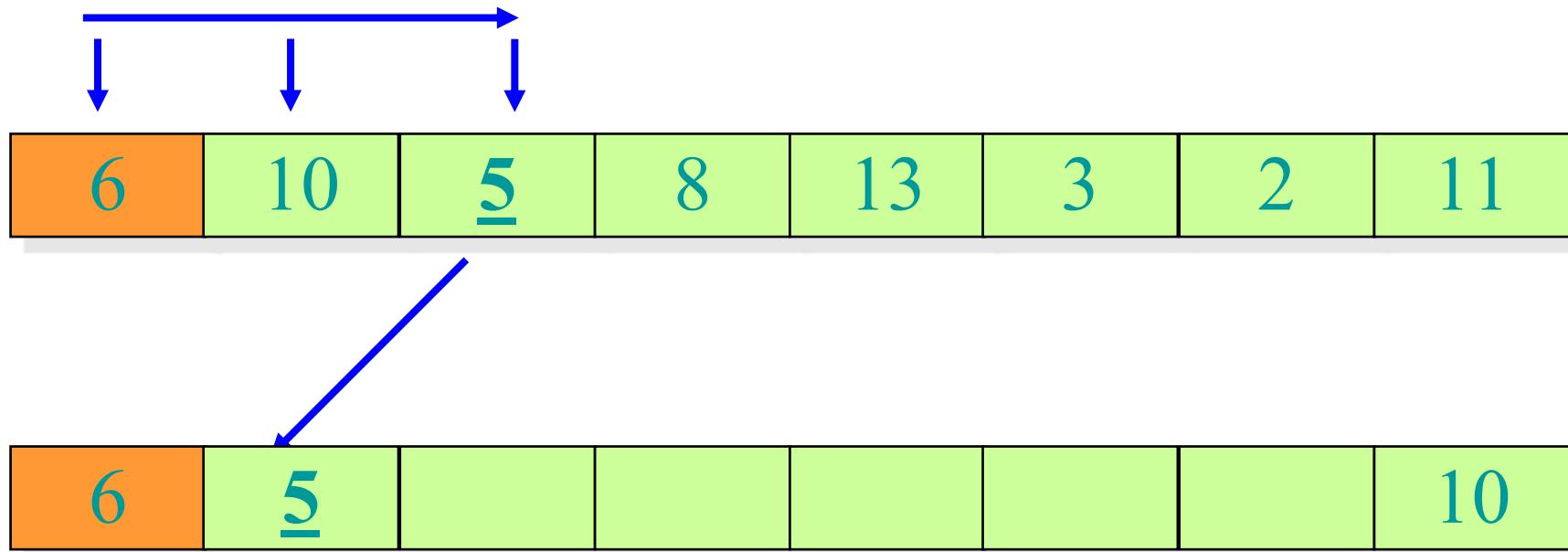
Idea of partition

- If we are allowed to use a second array, it would be easy



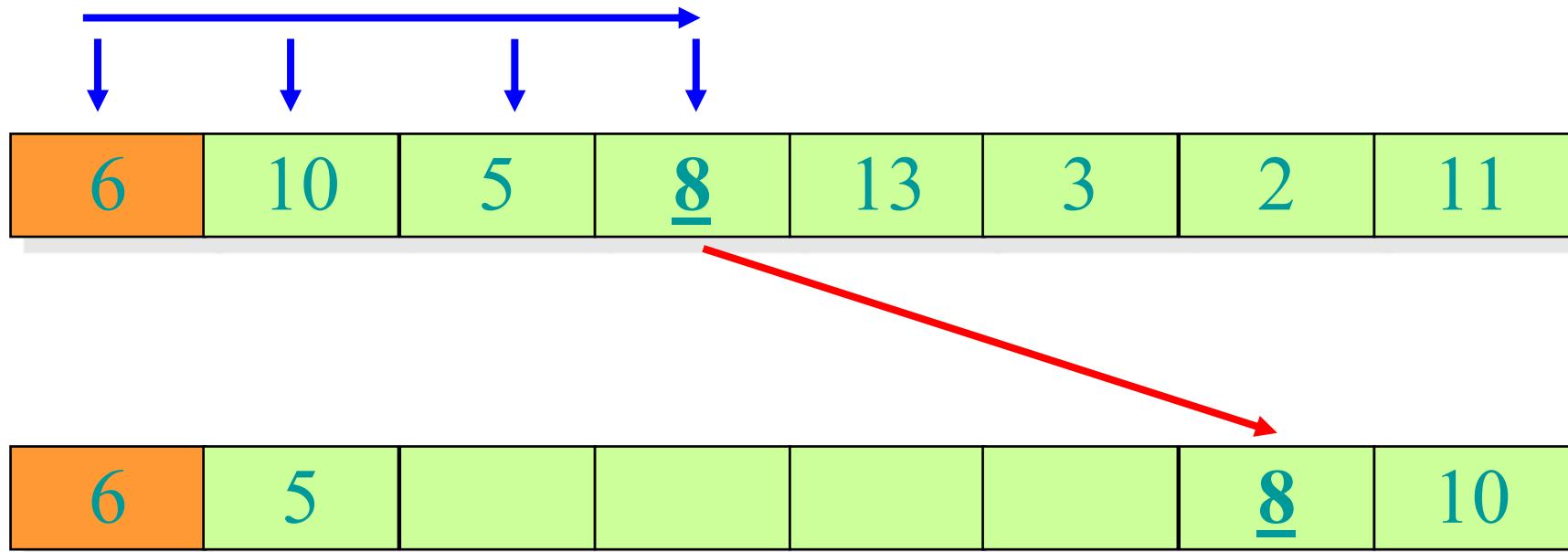
Idea of partition

- If we are allowed to use a second array, it would be easy



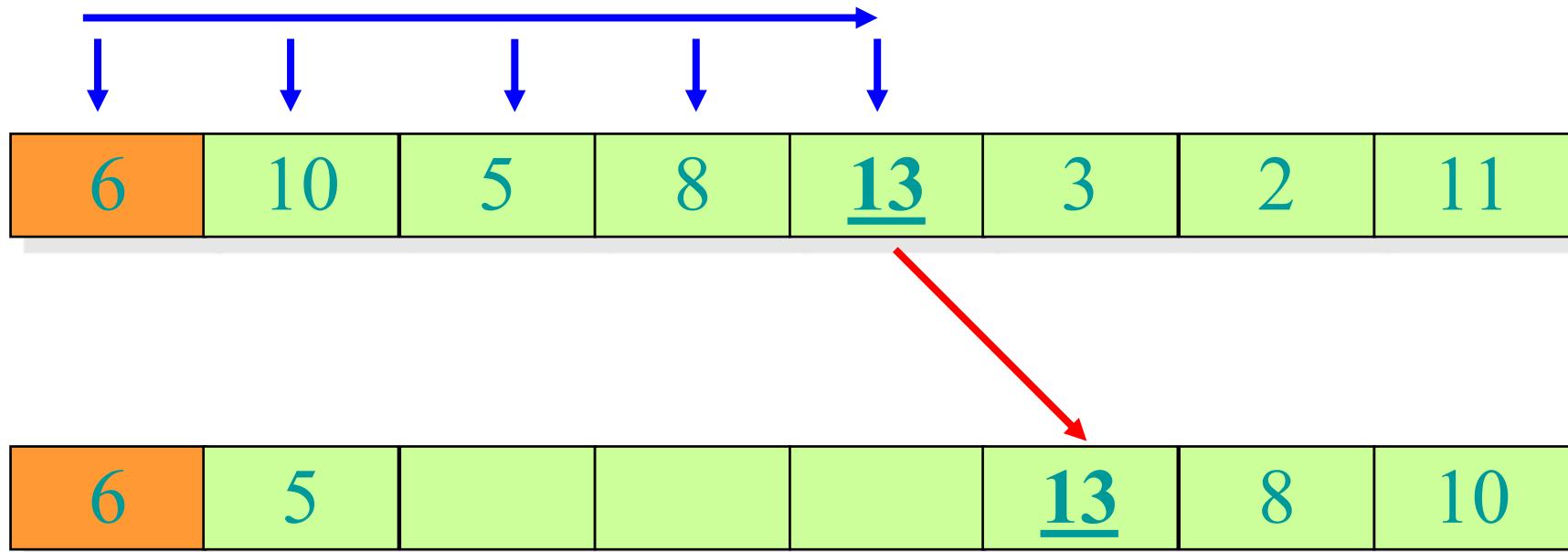
Idea of partition

- If we are allowed to use a second array, it would be easy



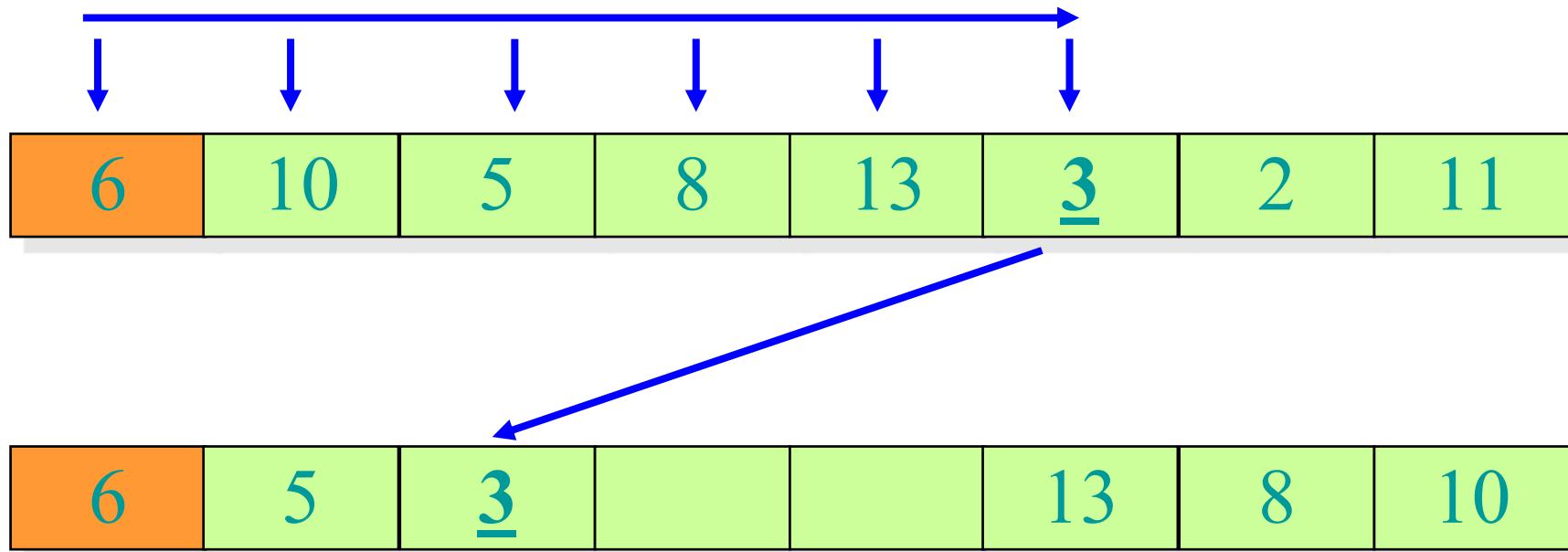
Idea of partition

- If we are allowed to use a second array, it would be easy



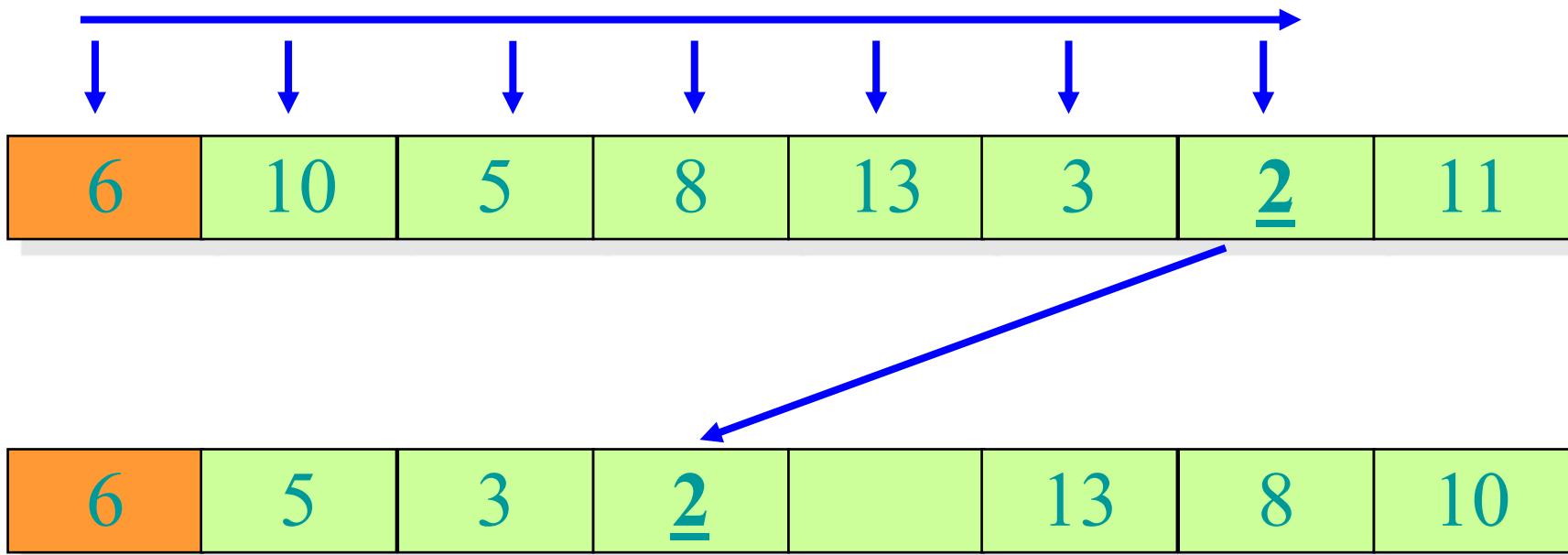
Idea of partition

- If we are allowed to use a second array, it would be easy



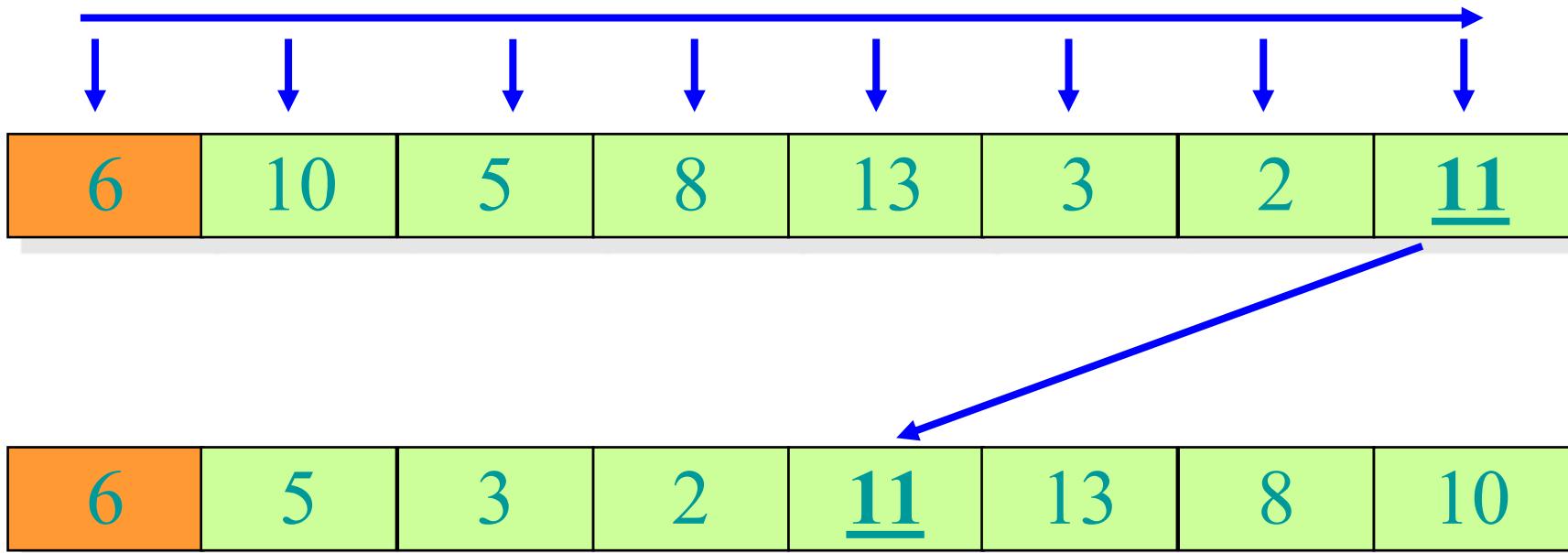
Idea of partition

- If we are allowed to use a second array, it would be easy



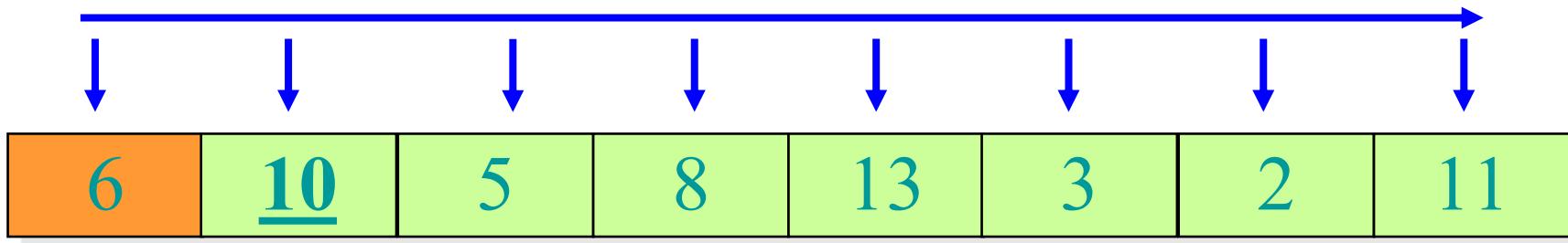
Idea of partition

- If we are allowed to use a second array, it would be easy



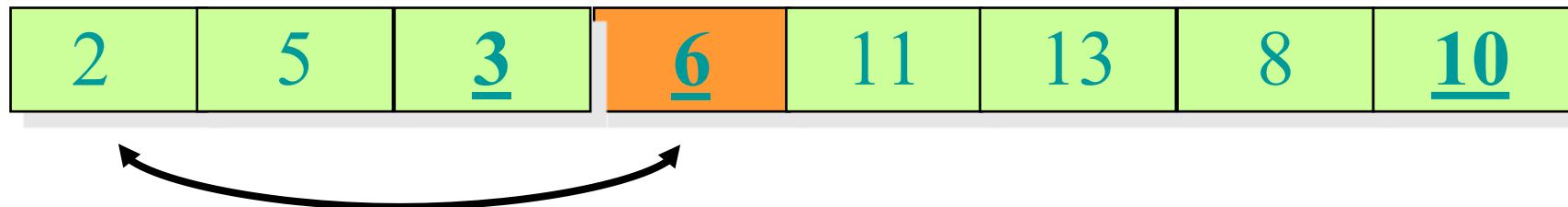
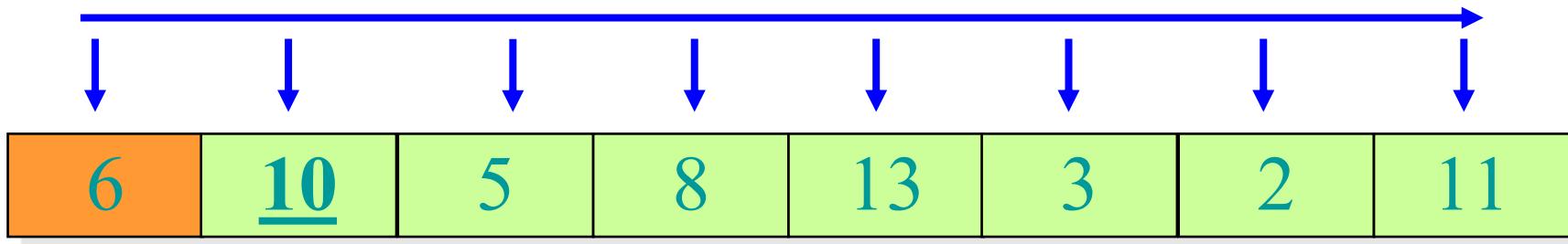
Idea of partition

- If we are allowed to use a second array, it would be easy

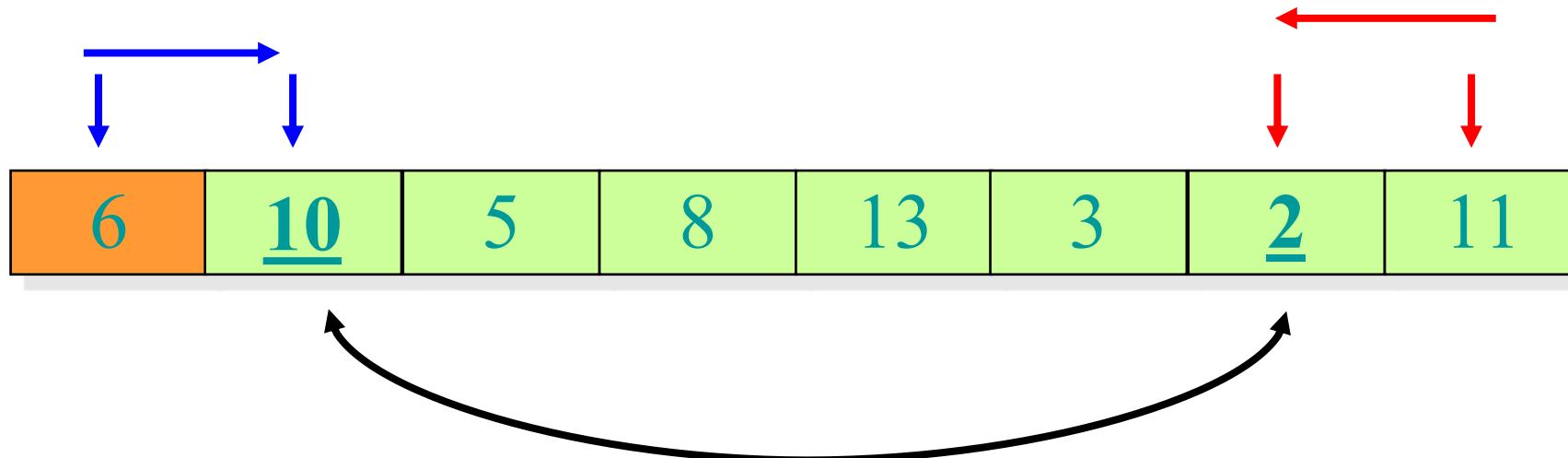


Idea of partition

- If we are allowed to use a second array, it would be easy

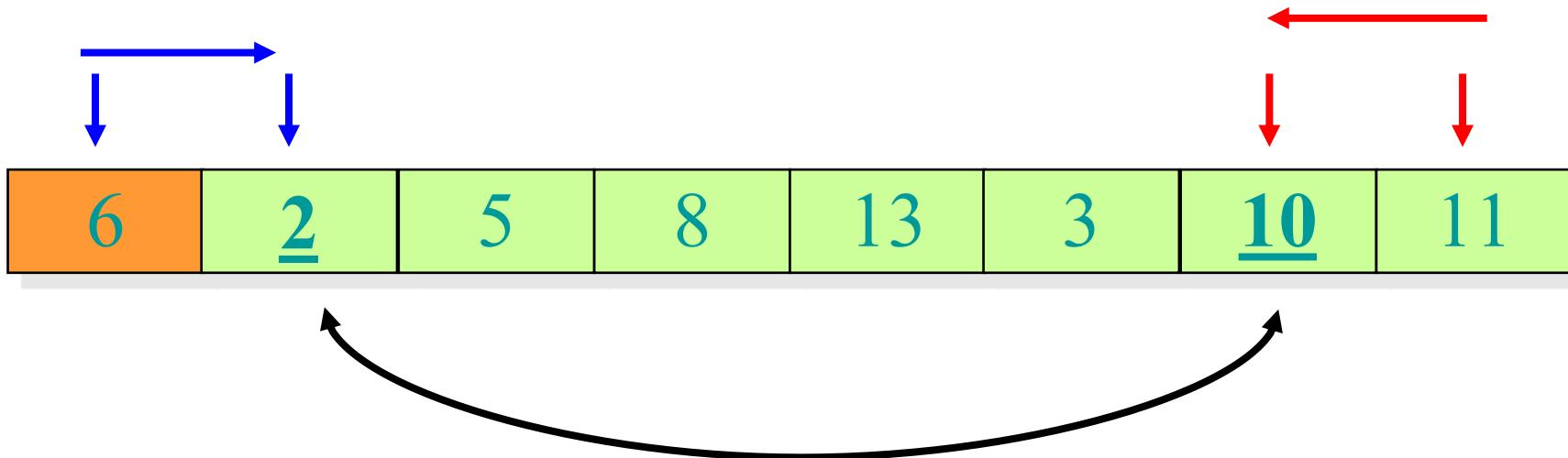


Idea of **in-place** partition



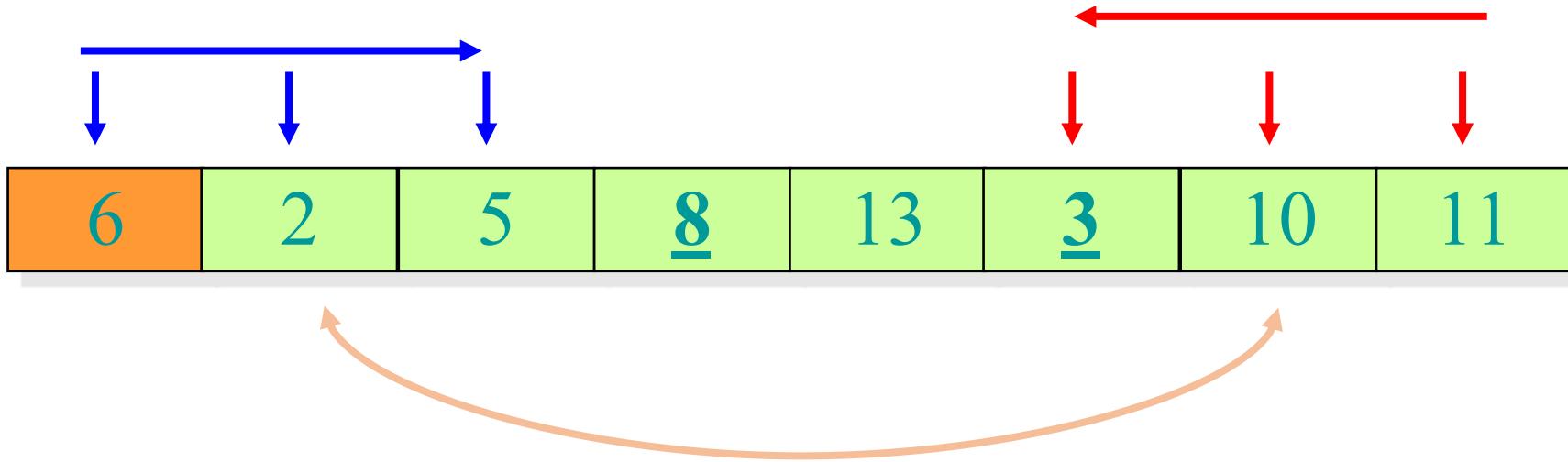
- Keep two iterators: one from head, one from tail
- Blue pointer stops at a number > 6 (pivot)
- Red pointer stops at a number < 6 (pivot)
- Swap both numbers
 - Stop when BLUE and RED pointers cross over

Idea of **in-place** partition



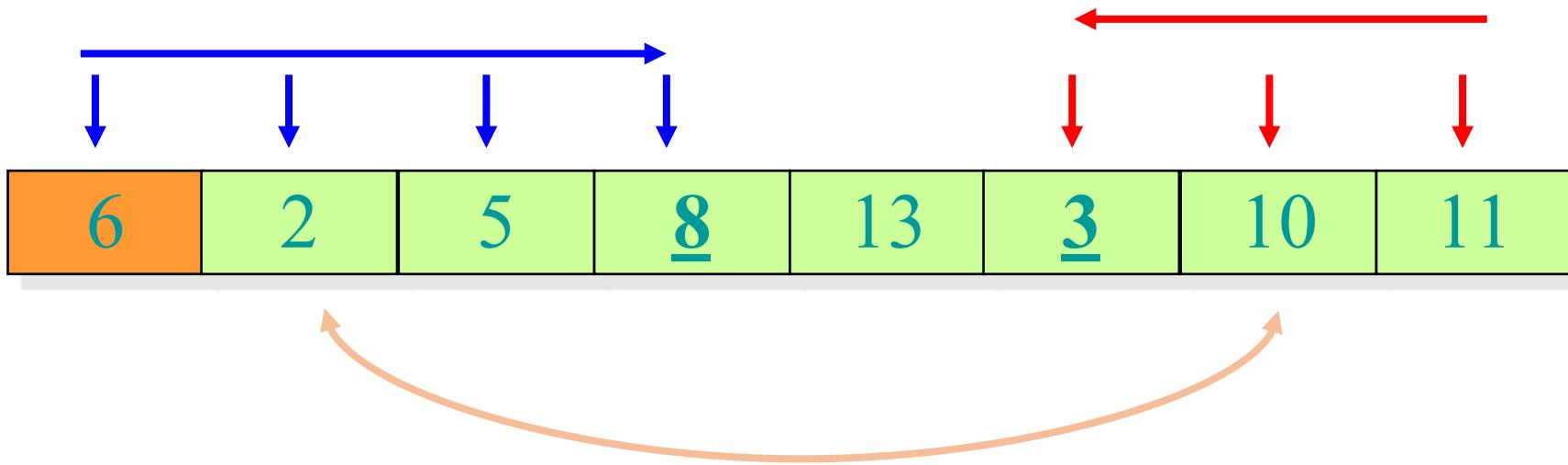
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



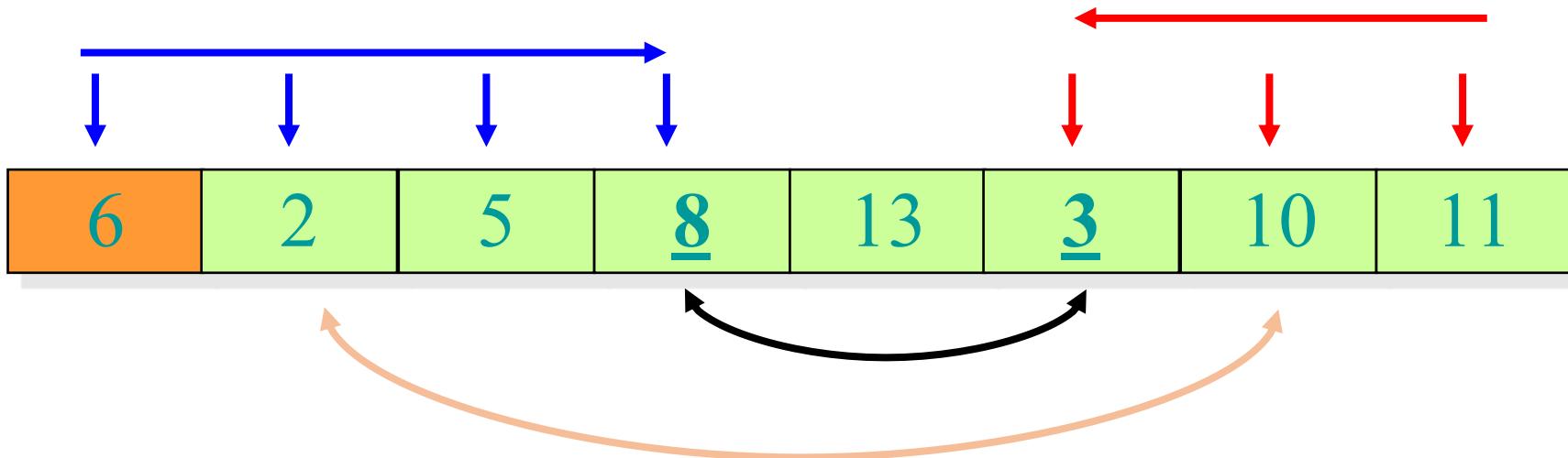
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



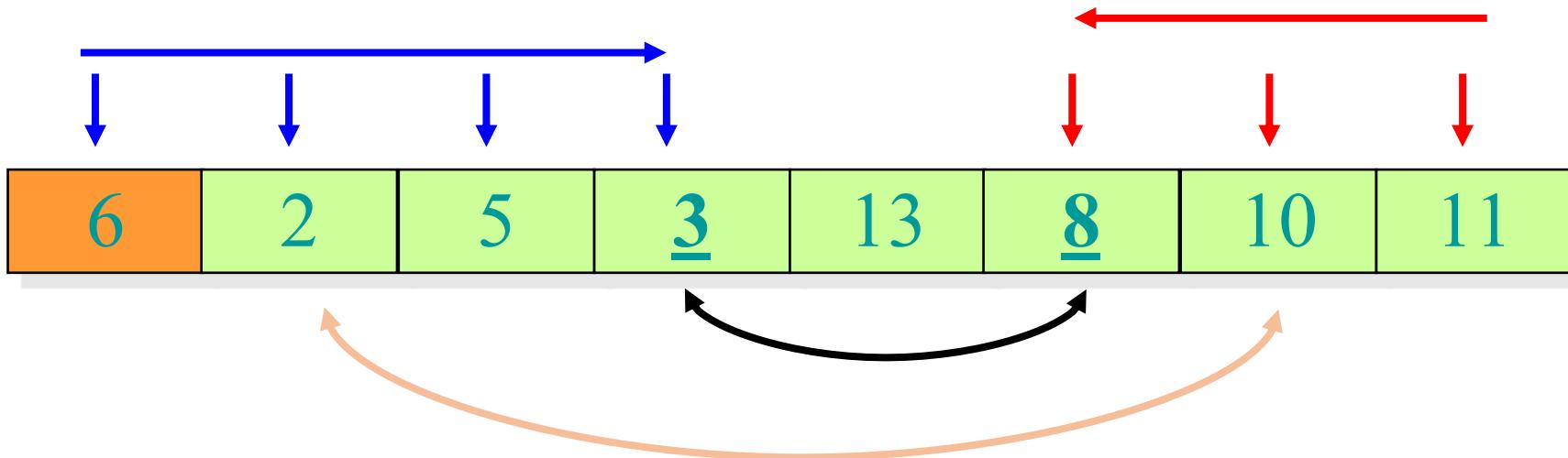
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



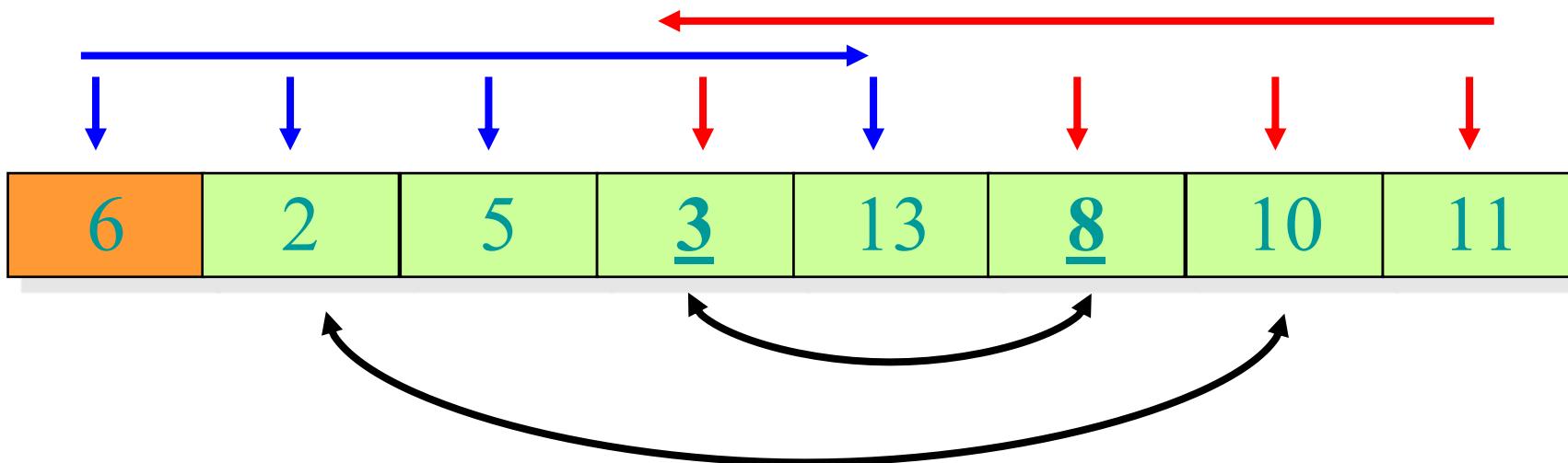
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



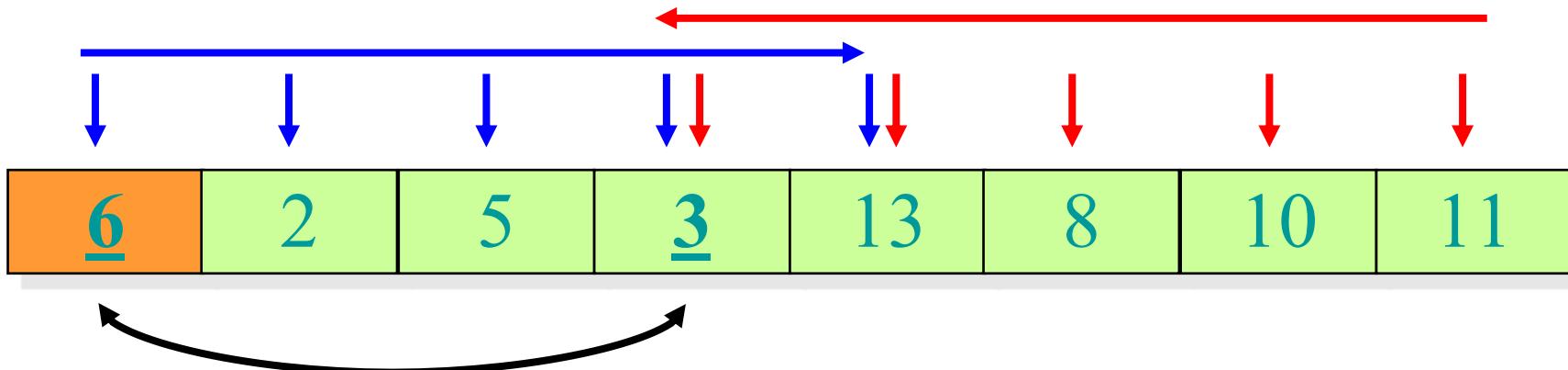
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



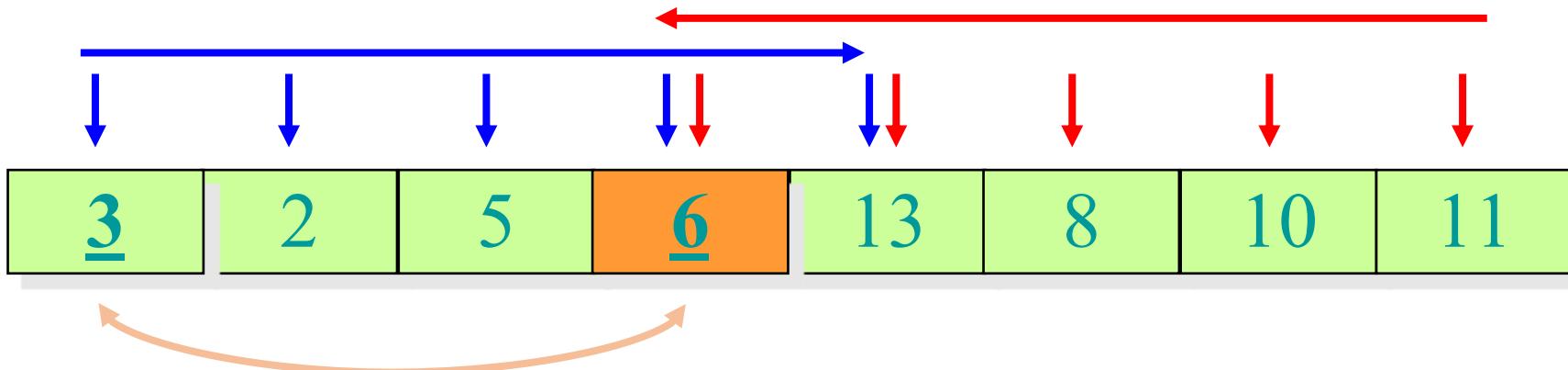
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Idea of **in-place** partition



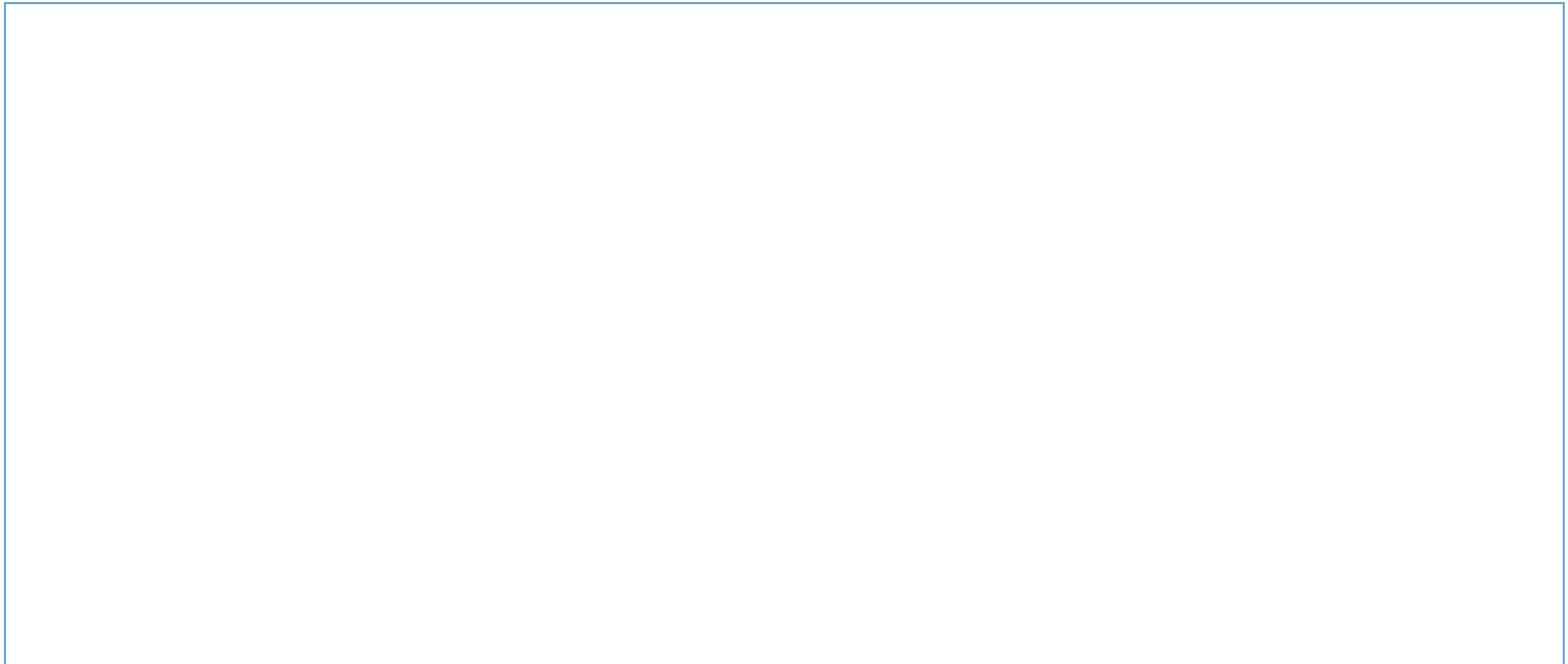
- Keep two iterators: one from head, one from tail
 - Stop when they cross over

Partition In Words

- Partition(A, p, r):
 - Select an element to act as the “pivot” (*which?*)
 - Grow two regions, $A[p..i]$ and $A[j..r]$
 - All elements in $A[p..i] \leq \text{pivot}$
 - All elements in $A[j..r] \geq \text{pivot}$
 - Increment i until $A[i] > \text{pivot}$
 - Decrement j until $A[j] < \text{pivot}$
 - Swap $A[i]$ and $A[j]$
 - Repeat until $i \geq j$
 - Swap $A[j]$ and $A[p]$
 - Return j

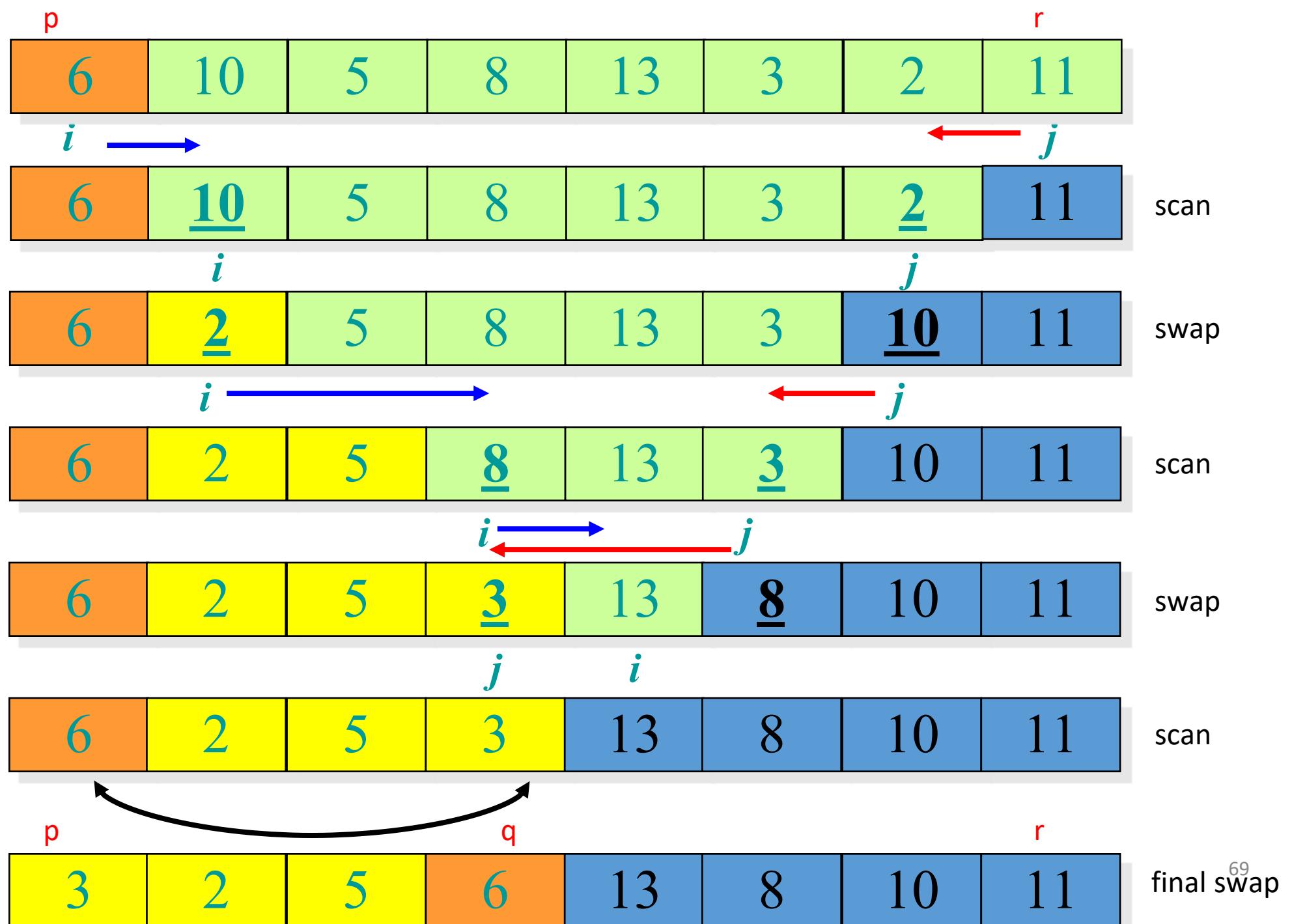
*Note: different from book's partition() ,
which uses two iterators that both move forward.*

Partition Code

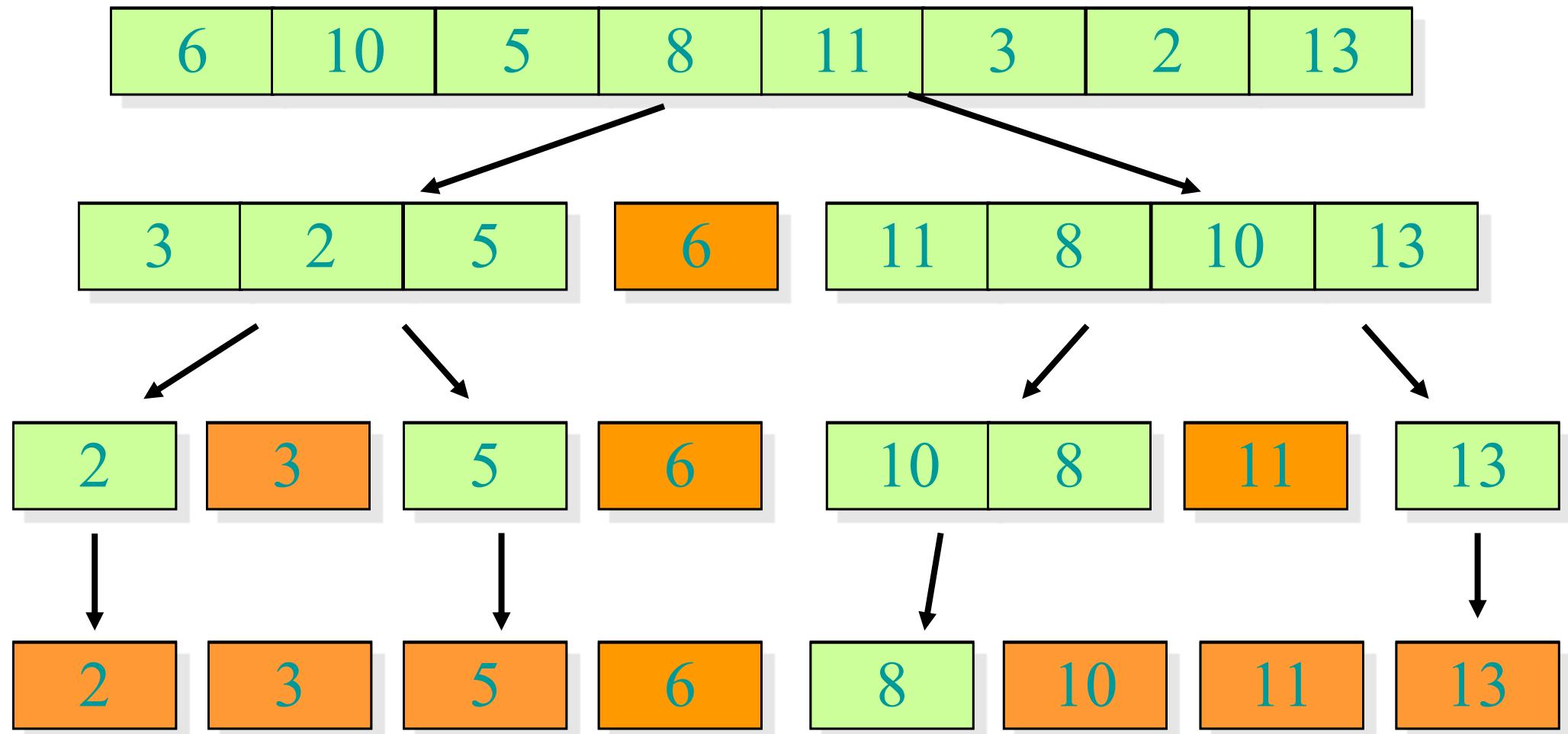


Running time: $\Theta(n)$ time

$$x = 6$$



Quick sort example



Quicksort Runtimes

- Best-case runtime $T_{\text{best}}(n) \in \Theta(n \log n)$
- Worst-case runtime $T_{\text{worst}}(n) \in \Theta(n^2)$
- Worse than mergesort? Why is it called quicksort then?
- Its average runtime $T_{\text{avg}}(n) \in \Theta(n \log n)$
 - That is, the expected runtime of **randomized quicksort** is $\Theta(n \log n)$

Time Complexity

Algorithm	Time Complexity
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Quick Sort	$O(n \log_2 n)$

For 10 billion random numbers (10^{10}),
Quick Sort is about 300,000,000 times faster!!!



Ratio	
n	$\log_2 n$
10^1	3.3
10^2	6.6
10^3	10
10^4	13
10^5	17
10^6	20
10^7	23
10^8	26
10^9	30
10^{10}	33

Time Complexity

	Best-Case	Average-Case	Worst-Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$

Other Issues

	Space Complexity	Stable
Selection Sort	O(1)	NO
Insertion Sort	O(1)	YES
Quick Sort	O(1)	NO

Stable: the relative order of elements with the same value is maintained.

Quiz

Which algorithm is the best for sorting data files frequently in the flash memory?

(Hints: Flash memory has limitation on the number of writes to it.)



Homework

1. Which algorithm is the best for sorting a big data file where most of data are in the sorted order already? (Hint: Best Case)
2. How do we avoid the worst-case behavior of the Quicksort? (Hints: selection of keys for partitioning.)
3. How many sorting algorithms are out there? (Hints: Google Search)

Unstable Sorting Algorithms

Please give a counter-example to show that they are not stable sorting algorithms.

- Heap Sort
- Selection sort
- Quick Sort

- Heap Sort: Consider array 20a 20b 19 (already in max-heap format) → 19 20b 20a.
- Selection sort: 4a 2 3 4b 1 → 1 2 3 4b 4a
- Quick Sort: 4a 2 3 4b 1 (use 3 as pivot) → 1 2 3 4b 4a

In place algorithm

Which of the following sorting algorithms is NOT in-place?

- Quick sort
- Merge Sort
- Insertion Sort
- Selection Sort
- Heap sort