Sorting

Outline

- Sorting
  - Stable Sort
  - Comparison Based
    - Lower bound $\Omega(n\log n)$
  - Non-comparison Based
- Internal Sorting
- External Sorting (Read 7.10)
  - Limited memory

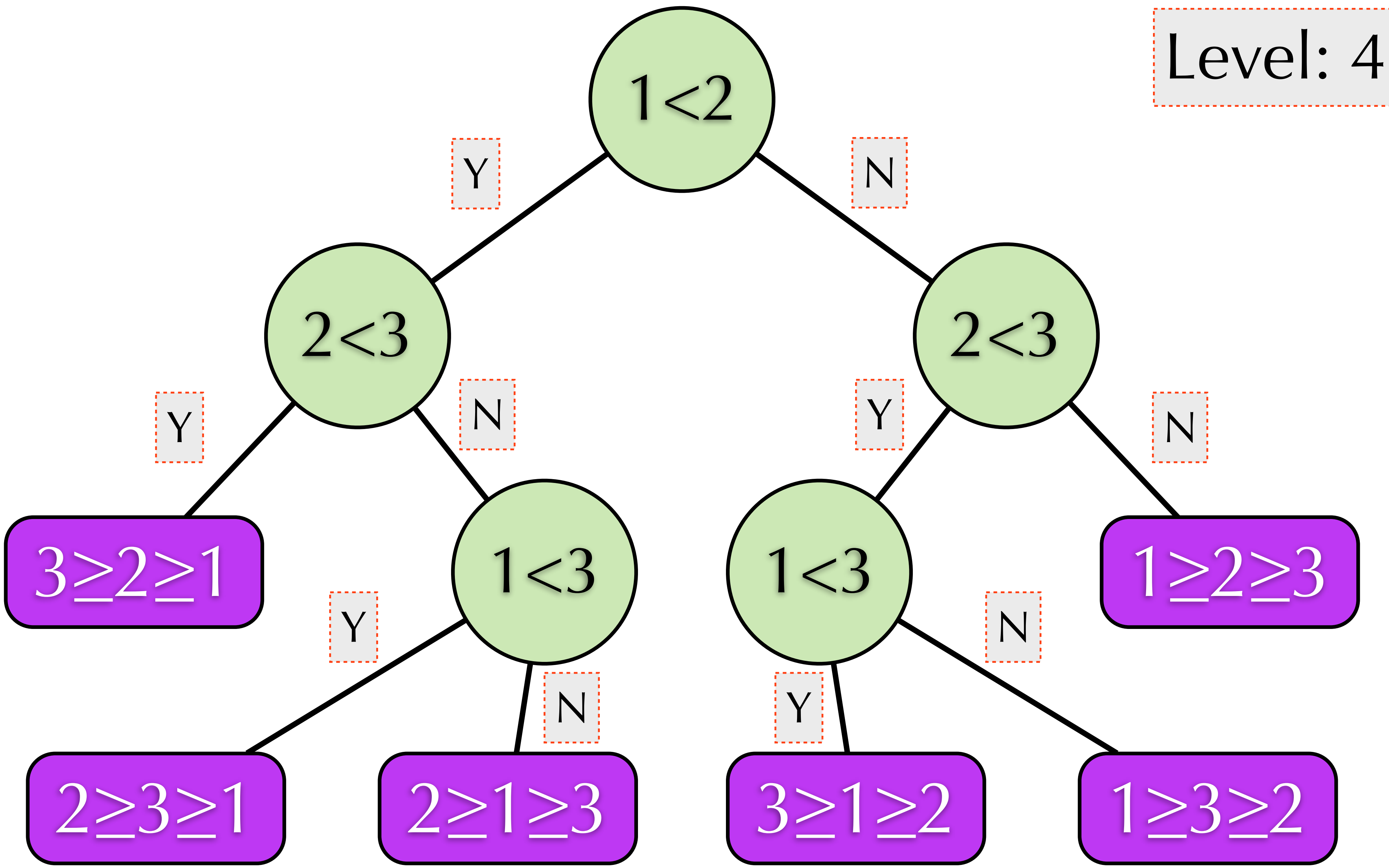
Sorting algorithm

- Make the sequence elements in a particular order. For ex: ascending order
- A sorting algorithm is stable: If two elements have the same key, then order of their appearance remain the same after sorting.
- Comparison based sorting algorithm need $\Omega(n\log n)$ time.

Sorting Lower Bound

- All possible permutations: n!
- Comparison result: at most 3 (<, =, >)
- A comparison-based sorting algorithm can represented as a decision tree.
- The minimum level of tree of n! leaves is $\Omega(\log(n!))=\Omega(n\log n)$
  - $\log(n!)=\log 1+\log 2+\ldots+\log n$
    $\geq\log(n/2)+\log(1+(n/2))+\ldots+\log n$
    $\geq(n/2)(\log n-\log 2)=\Omega(n\log n)$

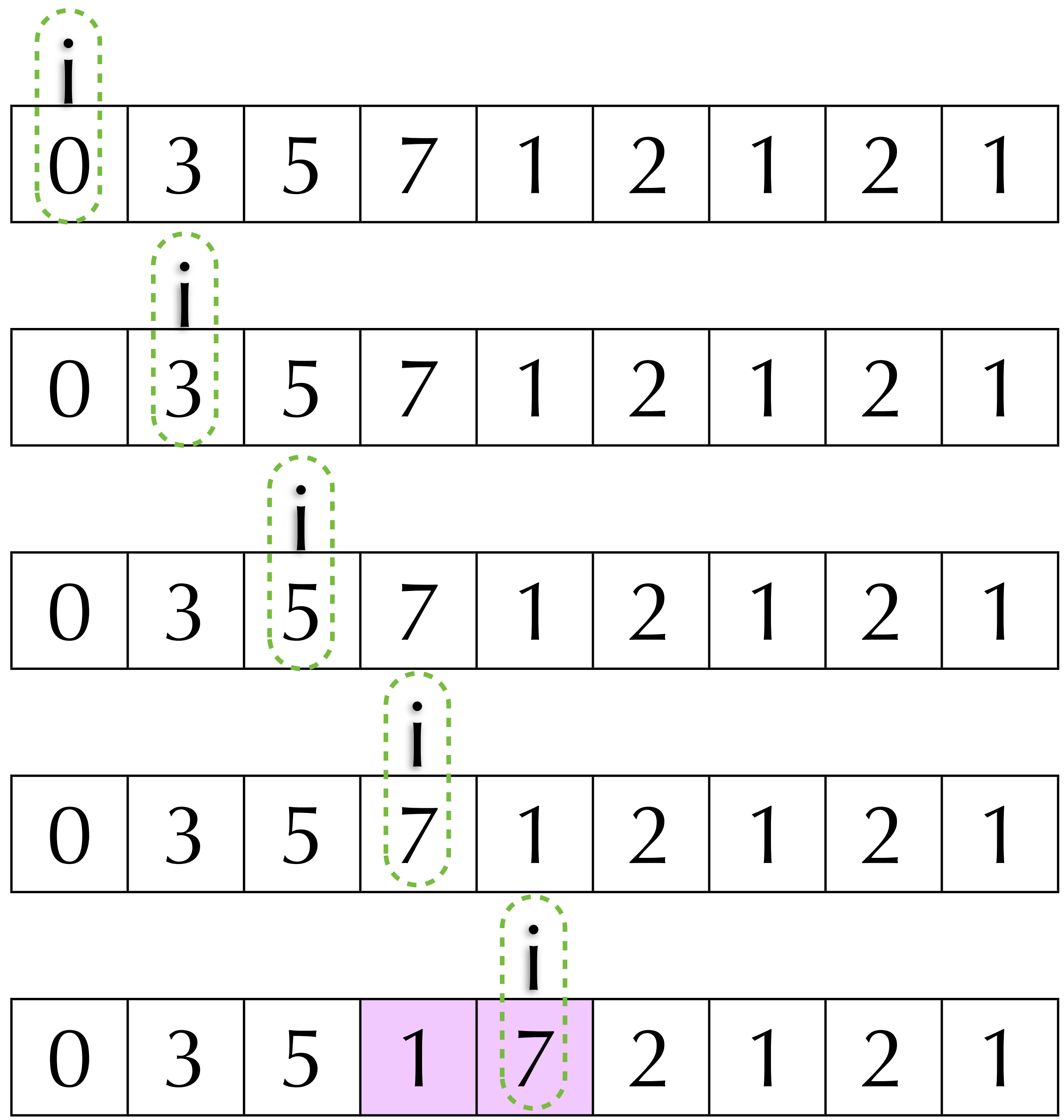
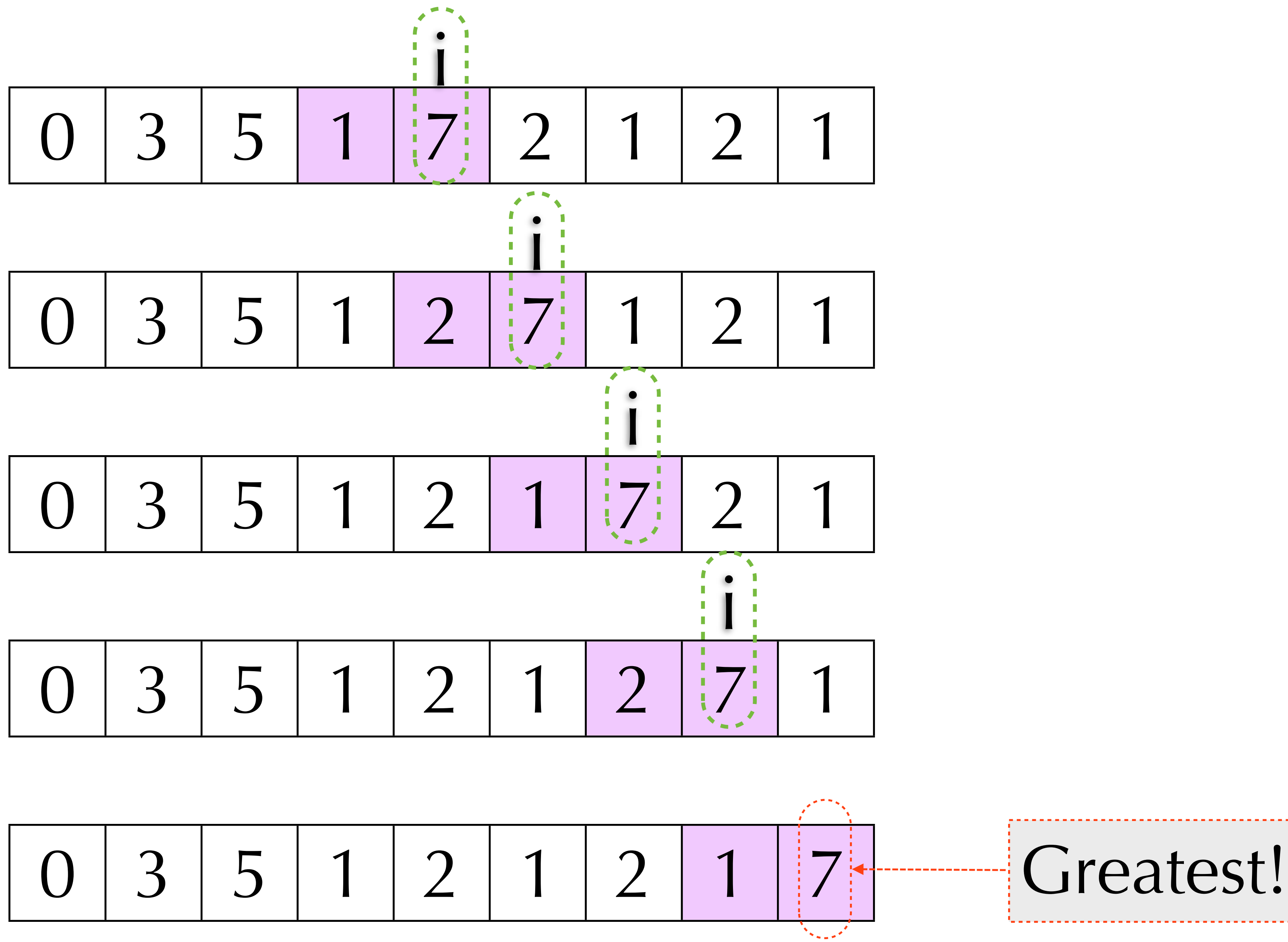
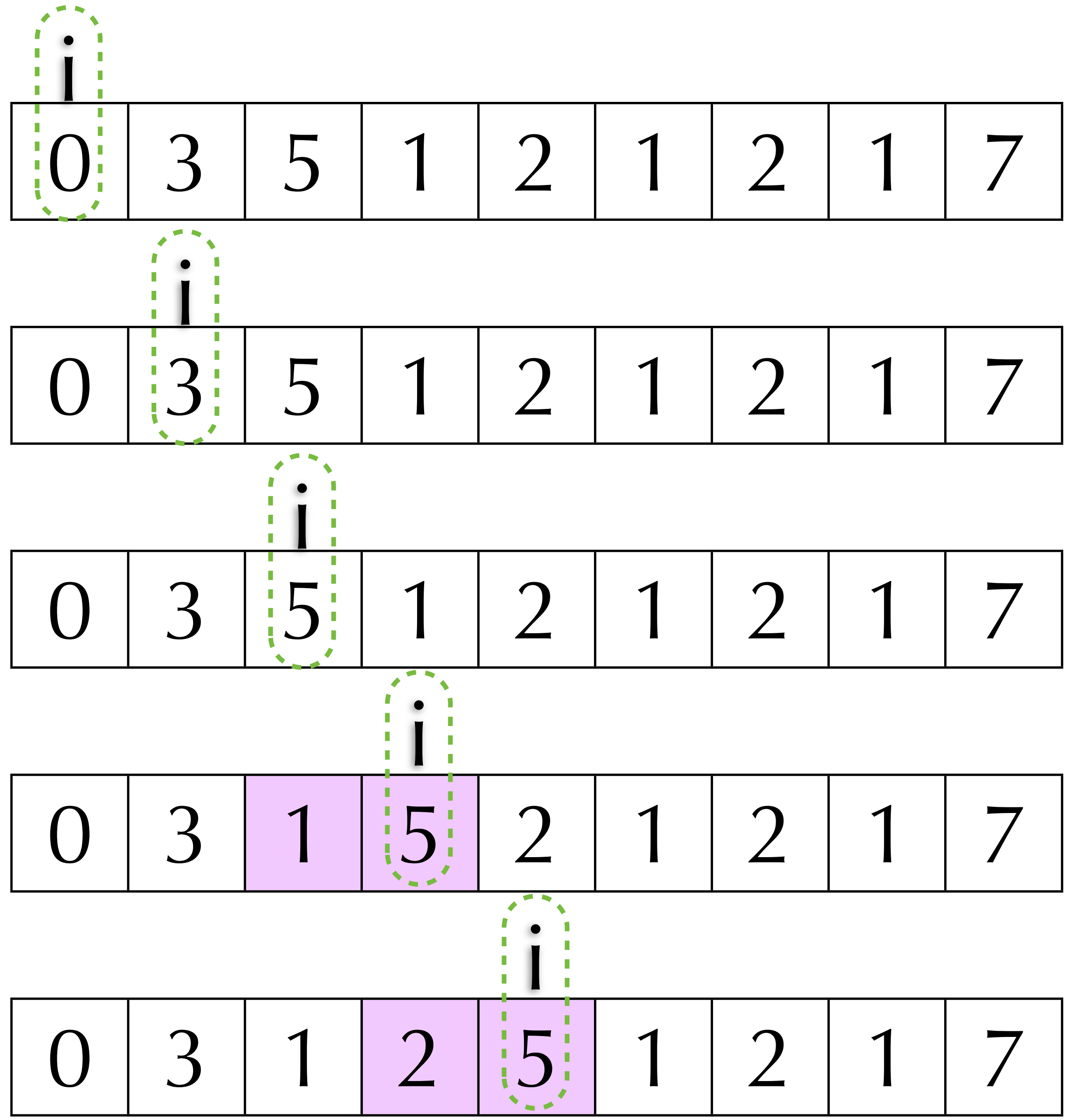
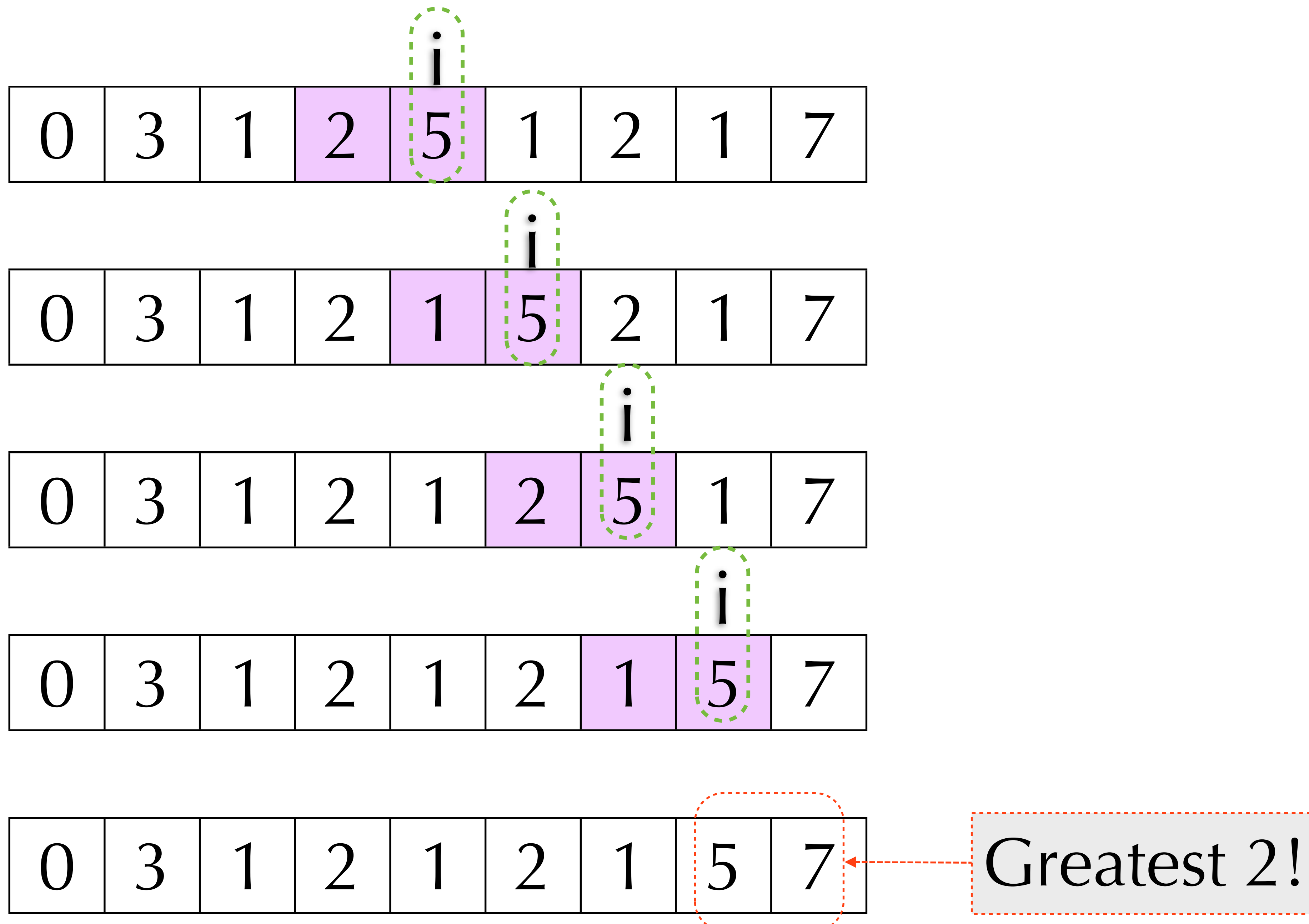
Sorting Lower Bound



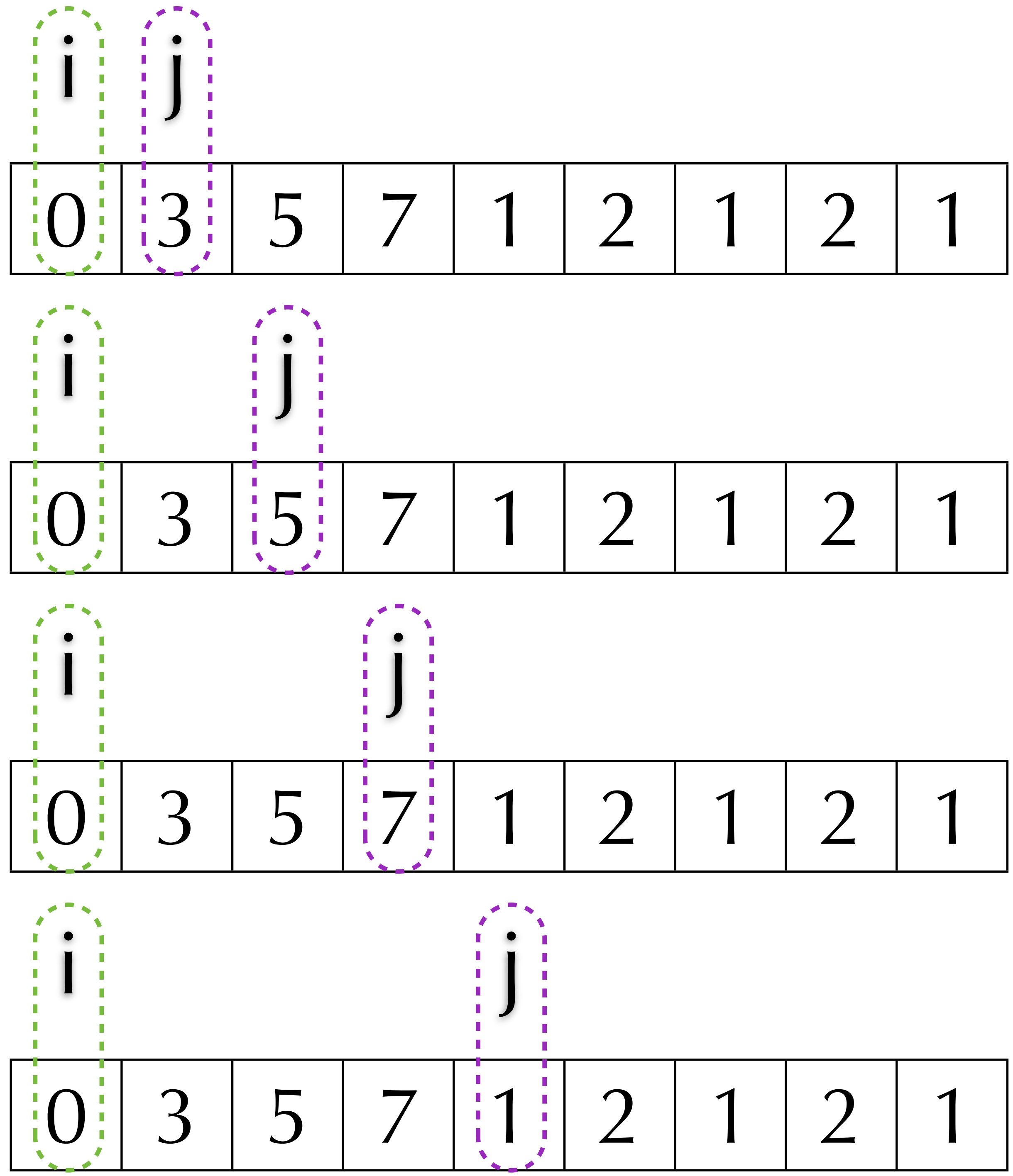
Internal Sorting

- Bubble Sort $O(n^2)$ Stable
- Selection Sort $O(n^2)$ Unstable
- Insertion Sort $O(n^2)$ Stable
- Quick Sort $O(n^2)$ Unstable
- Merge Sort $O(n\log n)$ Stable
- Heap Sort $O(n\log n)$ Unstable
- Counting Sort $O(n+m)$ Stable
- Radix Sort $O(d(n+m))$ Stable

Bubble Sort

- Basic operation: Compare two adjacent elements. If they are not in correct order, then swap them.
- ```
  For iter = 1 to n do
     For i = 1 to n-1 do
        if A[i]>A[i+1] then
           SWAP(A[i],A[i+1])
  ```
- After k iterations, the k greatest elements is stored in the k rightmost slots.
  ```

Bubble Sort


i
0 3 5 7 1 2 1 2 1
i
0 3 5 7 1 2 1 2 1
i
0 3 5 7 1 2 1 2 1
i
0 3 5 7 1 2 1 2 1
i
0 3 5 1 7 2 1 2 1

Bubble Sort


i
0 3 5 1 7 2 1 2 1
i
0 3 5 1 2 7 1 2 1
i
0 3 5 1 2 1 7 2 1
i
0 3 5 1 2 1 2 7 1
0 3 5 1 2 1 2 1 7
Greatest!

Bubble Sort


i
0 3 5 1 2 1 2 1 7
i
0 3 5 1 2 1 2 1 7
i
0 3 5 1 2 1 2 1 7
i
0 3 1 5 2 1 2 1 7
i
0 3 1 2 5 1 2 1 7

Bubble Sort


i
0 3 1 2 5 1 2 1 7
i
0 3 1 2 1 5 2 1 7
i
0 3 1 2 1 2 5 1 7
i
0 3 1 2 1 2 1 5 7
0 3 1 2 1 2 1 5 7
Greatest 2!

Selection Sort

- Put all elements into a priority queue.
- Repeating extract the minimum
- A possible implementation:
  - ```
    For i = 1 to n–1 do
        For j = i+1 to n do
            if A[i]>A[j] then
                SWAP(A[i],A[j])
    ```
- After k iterations, the k smallest elements is stored in the k leftmost slots.
    ```

Selection Sort



Selection Sort

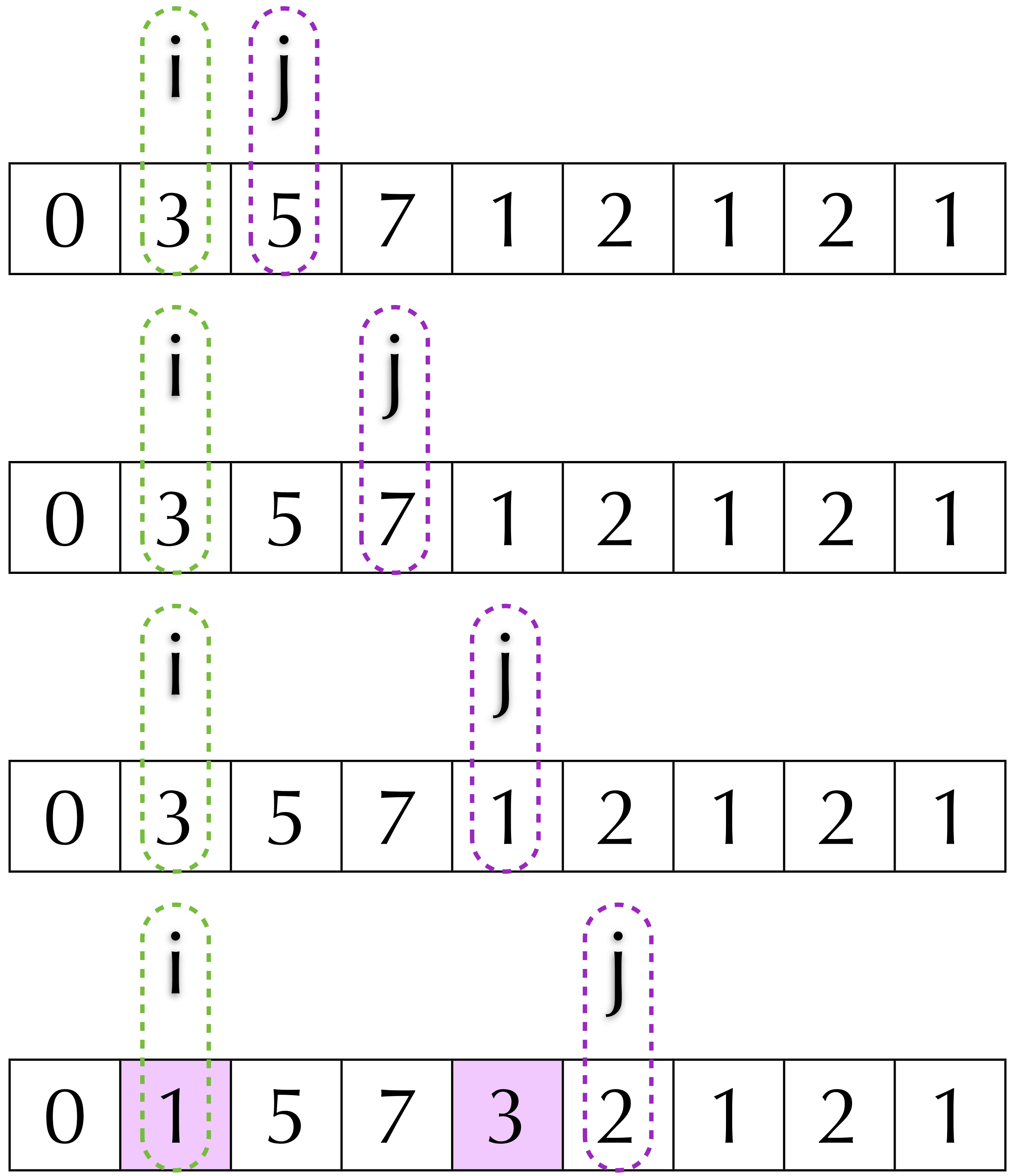
i					j			
0	3	5	7	1	2	1	2	1

i						j		
0	3	5	7	1	2	1	2	1

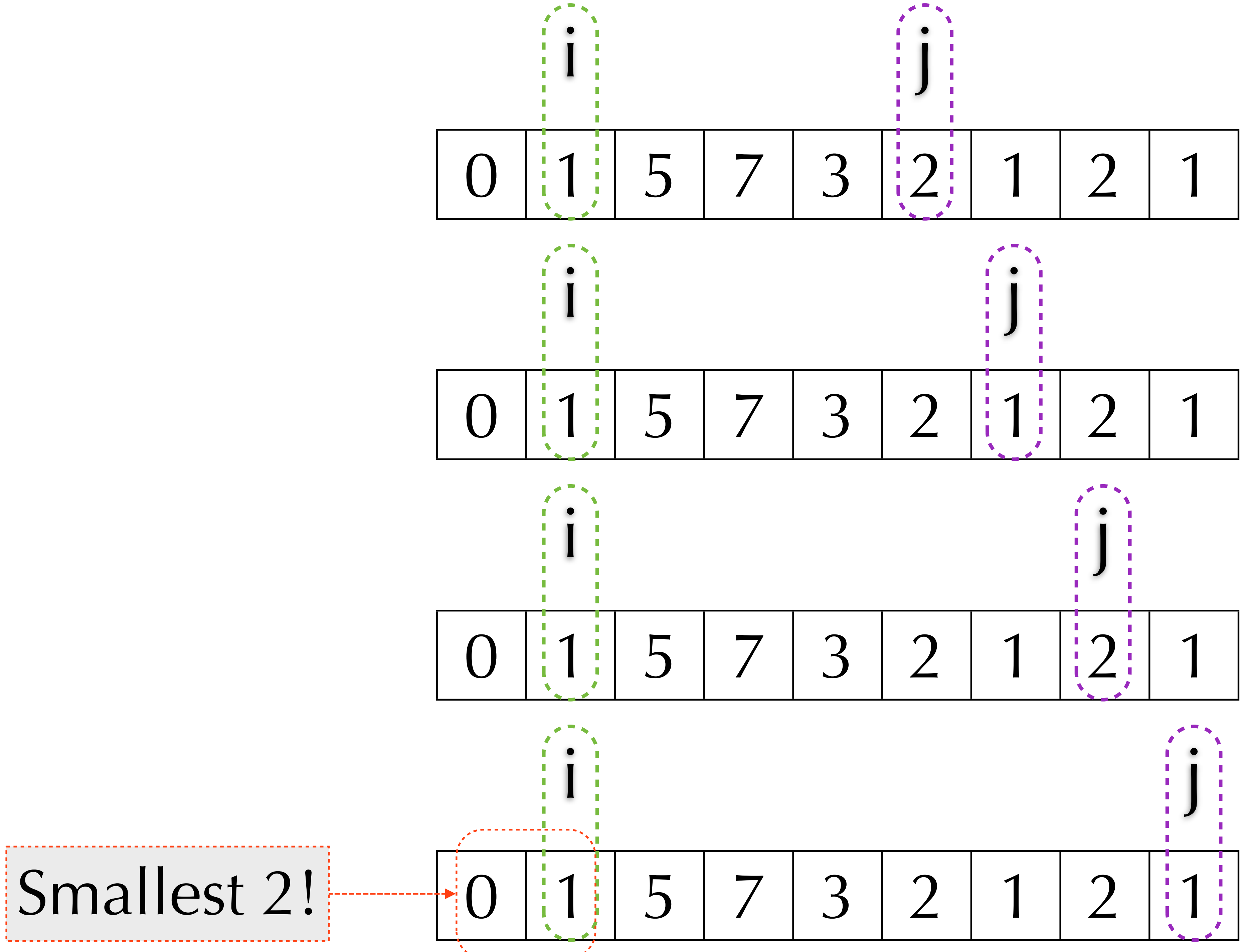
i							j	
0	3	5	7	1	2	1	2	1

i								j
0	3	5	7	1	2	1	2	1

Selection Sort

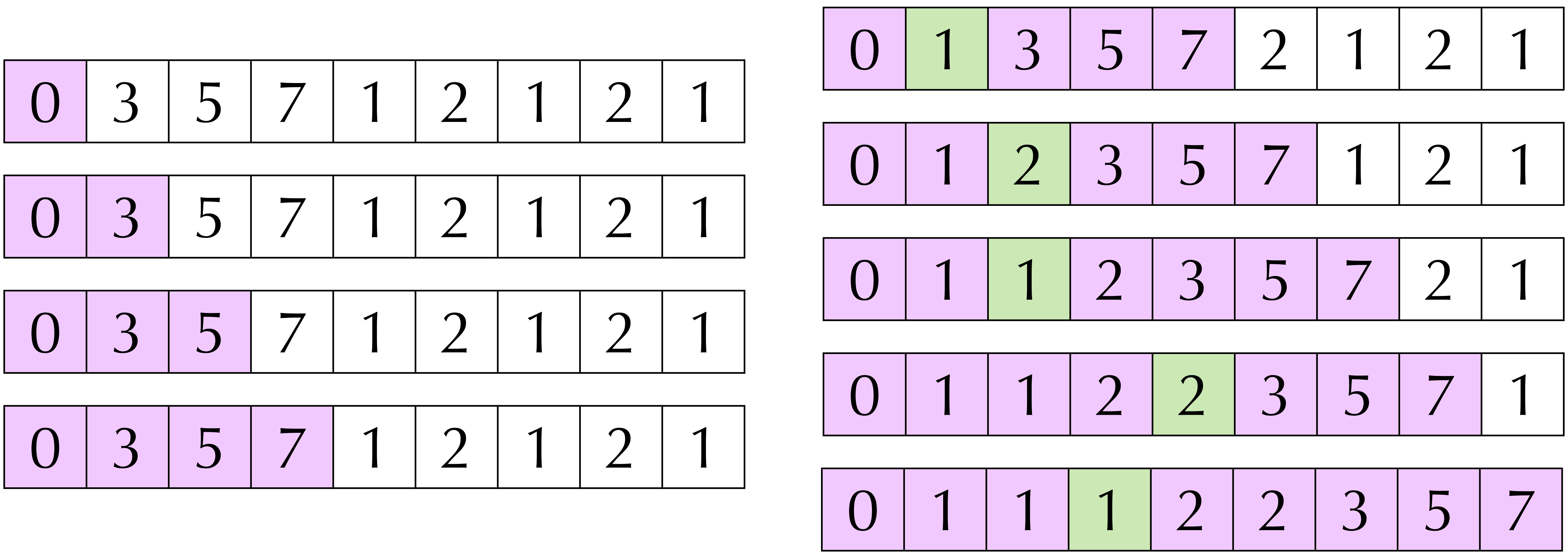


Selection Sort



Insertion Sort

- In each iteration, insert a new arrival element into a sorted list.



Homework 5.1

- a) Why bubble sort is stable?
- b) Why selection sort can be unstable?
- c) Why insertion sort is stable
- d) Among bubble sort, selection sort (the implementation on the slides) and insertion sort, which is the fastest in practice?

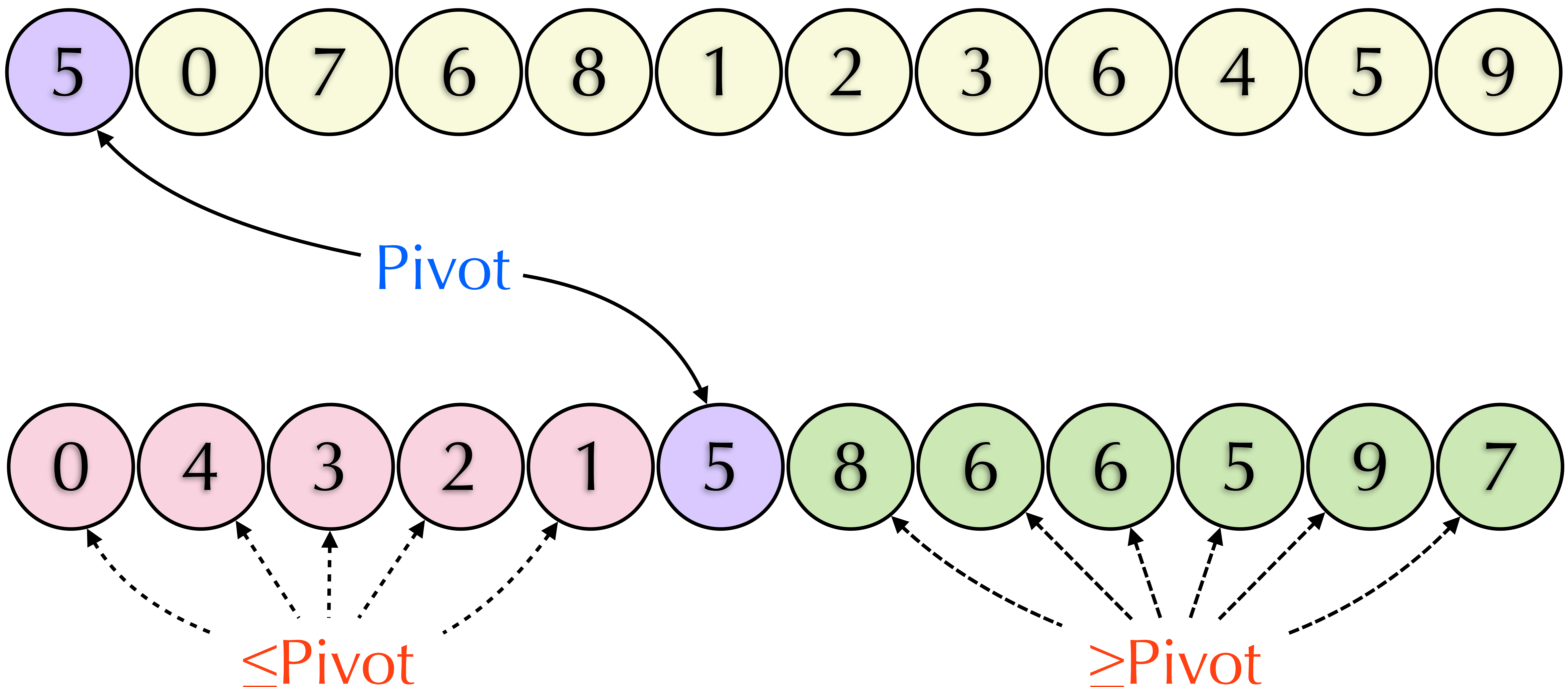
Quick Sort

- A recursive sorting algorithm
- Considered as the fastest internal sorting in practical use.
  - Need optimization!
  - Randomized version.
- Worst case: $O(n^2)$
- Average case: $O(n\log n)$
  - Assuming the input is uniformly randomly sampled.

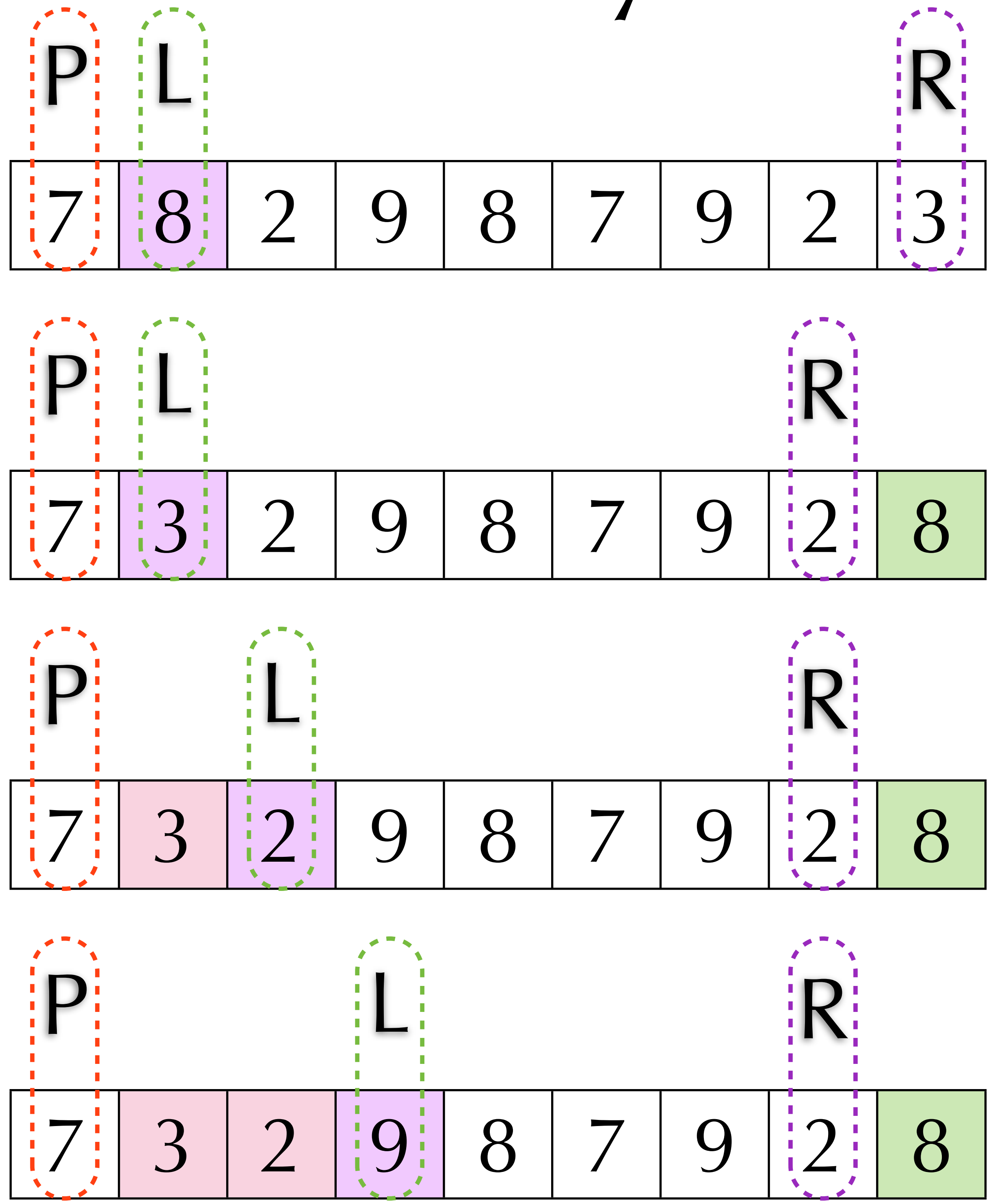
Quick Sort

- Partition: Return m such that
  - For $i<m$, $A[i] \leq A[m]$.
  - For $i>m$, $A[i] \geq A[m]$.
- Quick sort A[1..n]
  - If $n \leq 1$, then we're done.
    m=partition(A)
    quicksort(A[1..m–1])
    quicksort(A[m+1..n])

Partition by Pivot



Partition by Pivot

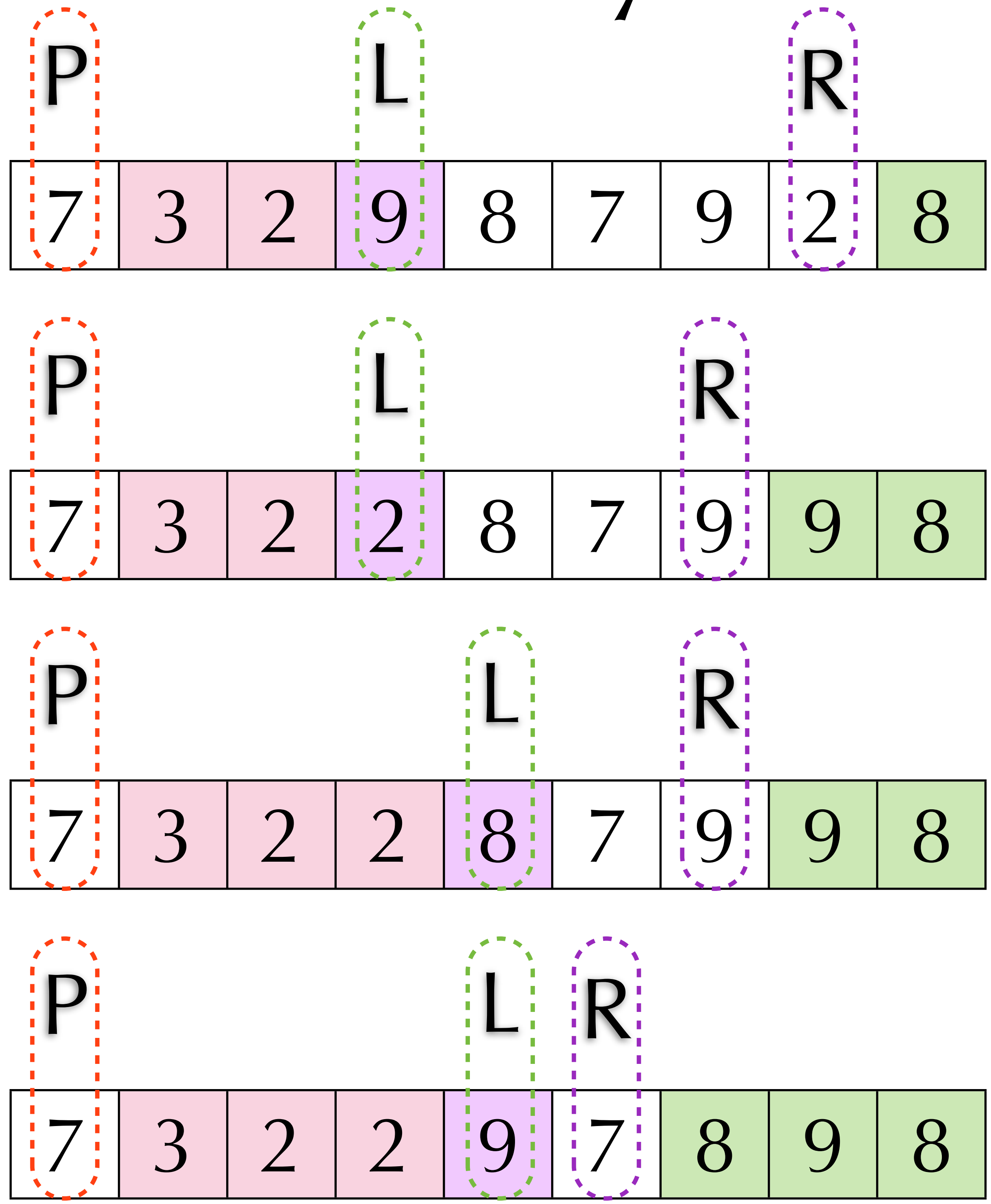


If A[P]<A[L]:
swap(A[L],A[R])
R=R–1

If A[P]≥A[L]:
L=L+1

If A[P]≥A[L]:
L=L+1

Partition by Pivot

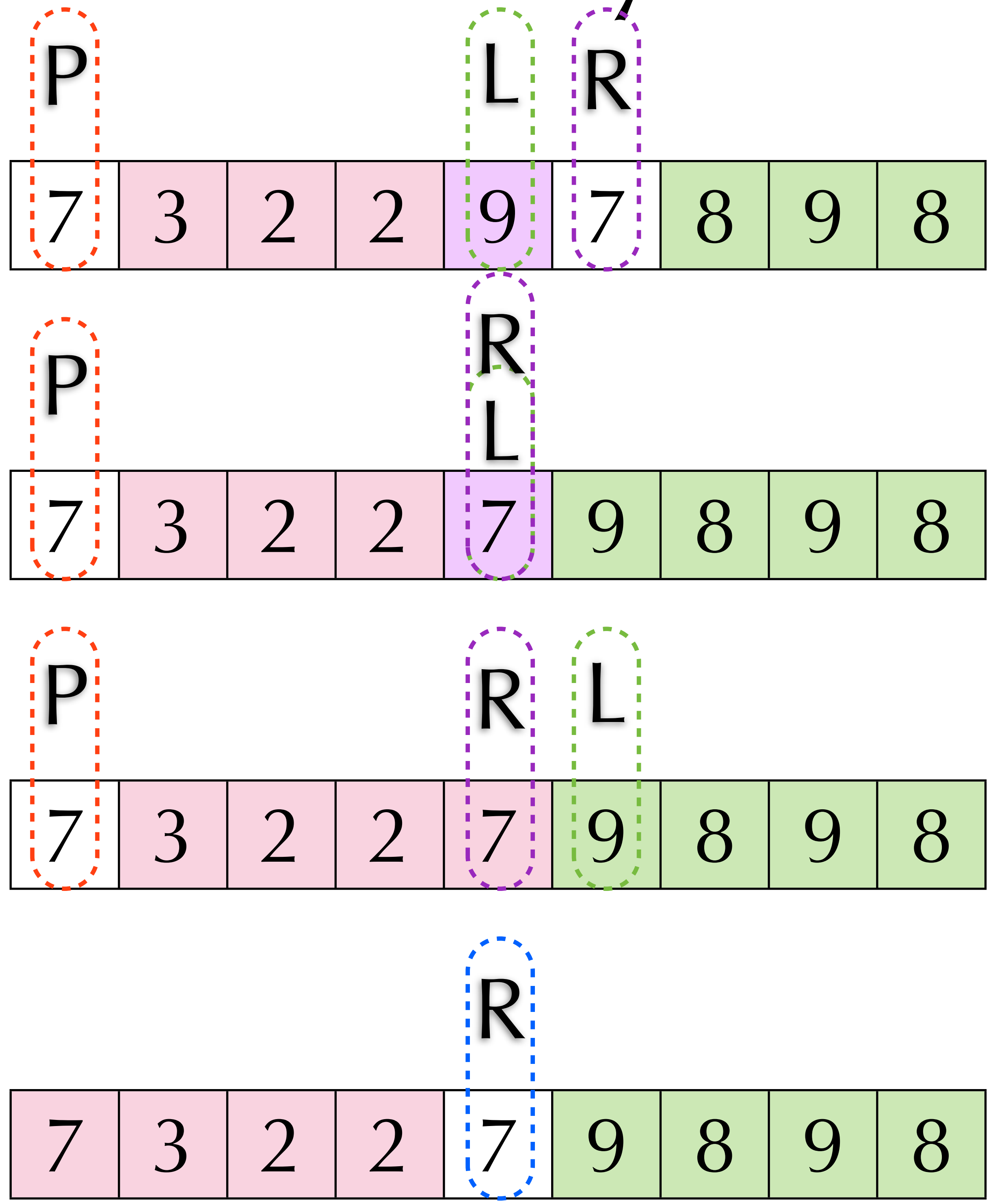


If $A[P]<A[L]$:
swap(A[L],A[R])
R=R–1

If $A[P] \geq A[L]$:
L=L+1

If $A[P]<A[L]$:
swap(A[L],A[R])
R=R–1

Partition by Pivot

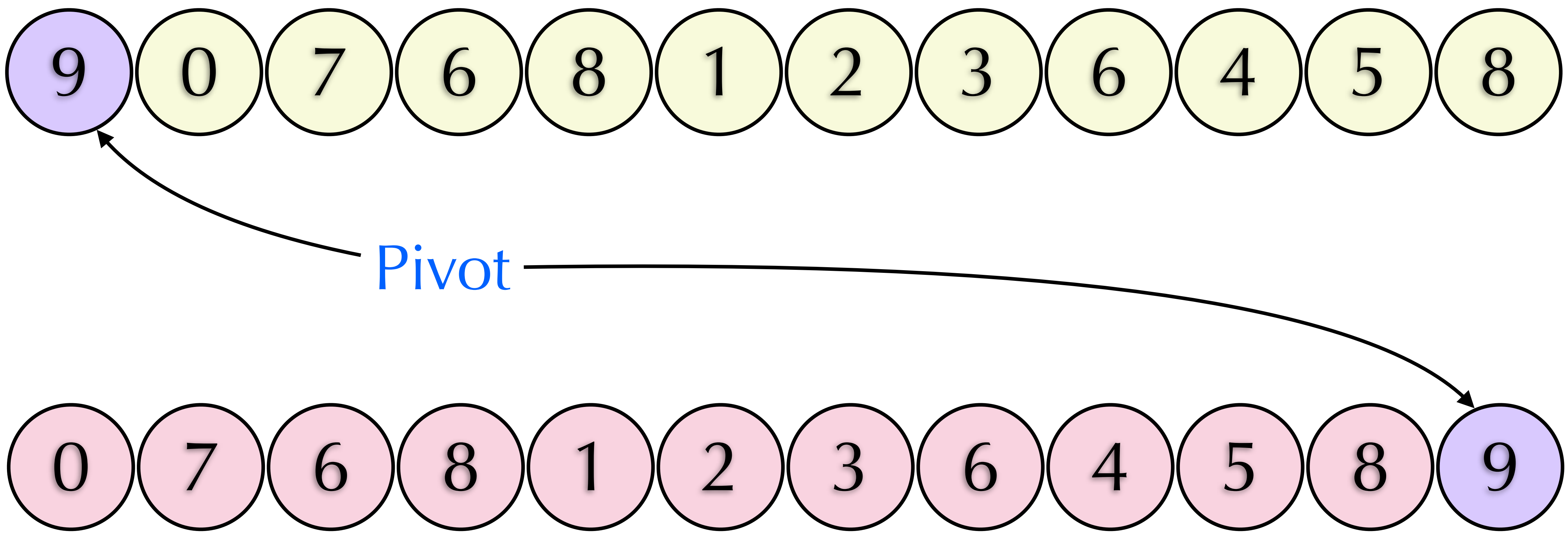


If A[P]<A[L]:
swap(A[L],A[R])
R=R–1

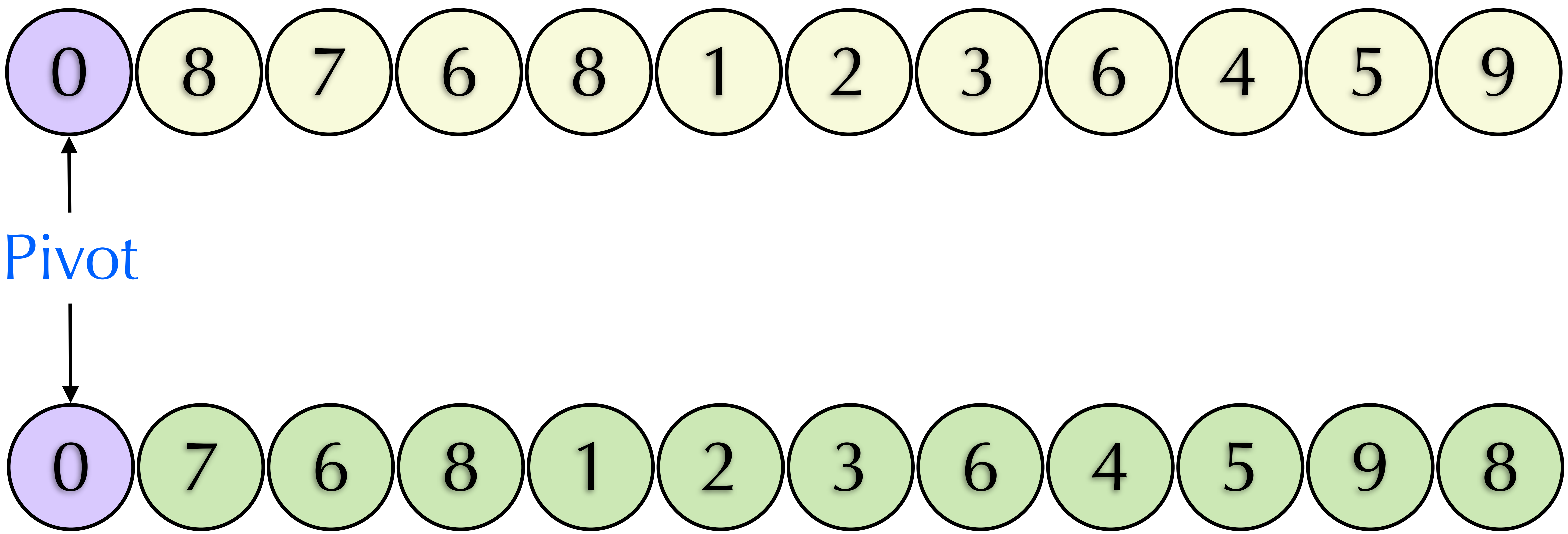
If A[P]≥A[L]:
L=L+1

If R<L:
swap(A[P],A[R])
return R

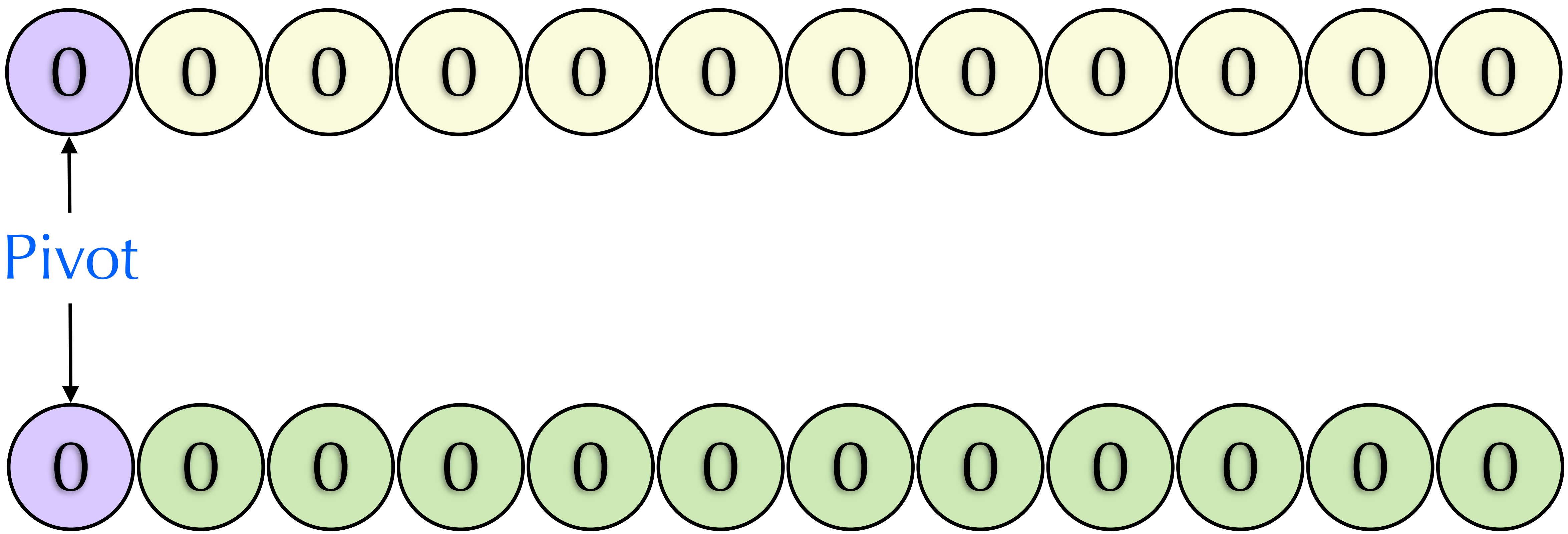
Partition: Worst Case 1



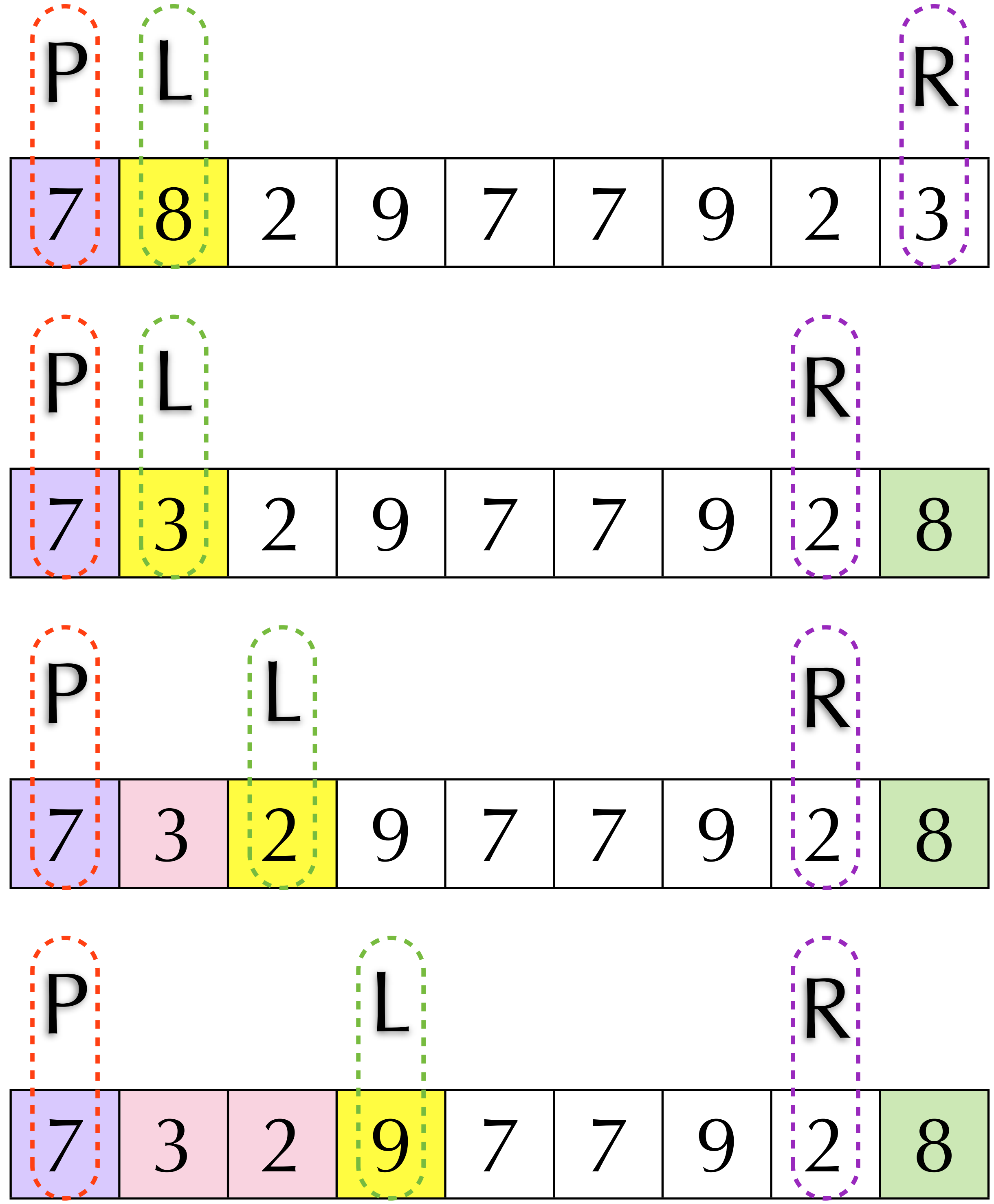
Partition: Worst Case 2



Partition: Worst Case 3



Modified Partition

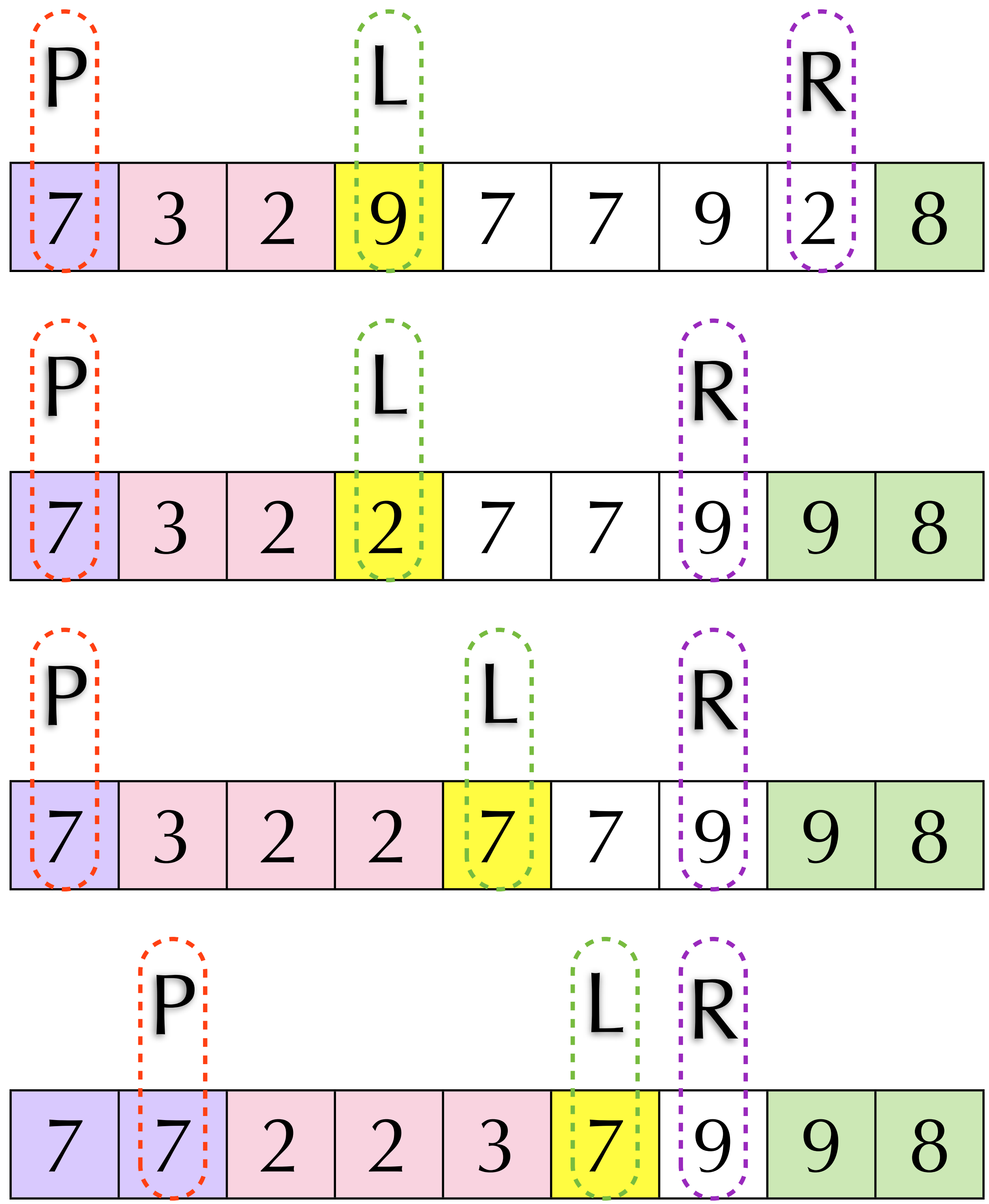


If A[P]<A[L]:
swap(A[L],A[R])
R=R–1

If A[P]>A[L]:
L=L+1

If A[P]>A[L]:
L=L+1

Modified Partition

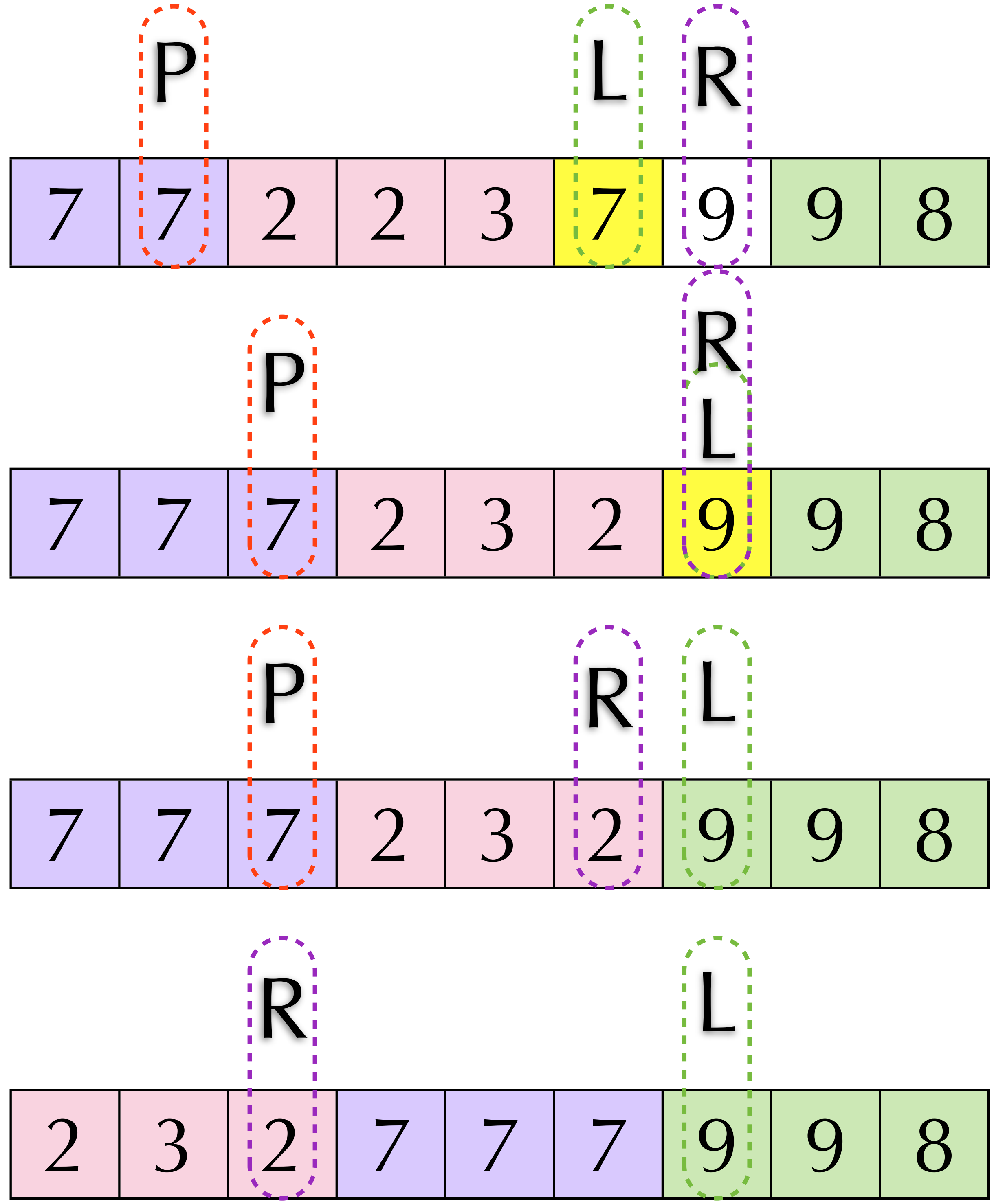


If A[P]<A[L]:
swap(A[L],A[R])
R=R–1

If A[P]>A[L]:
L=L+1

If A[P]=A[L]:
P=P+1
swap(A[P],A[L])
L=L+1

Modified Partition



If A[P]=A[L]:
P=P+1
swap(A[P],A[L])
L=L+1

If A[P]<A[L]:
swap(A[L],A[R])
R=R–1

If R<L:
while(P>0)
  swap(A[P],A[R])
  P=P–1, R=R–1

qsort(A[1..R])
qsort(A[L..n])

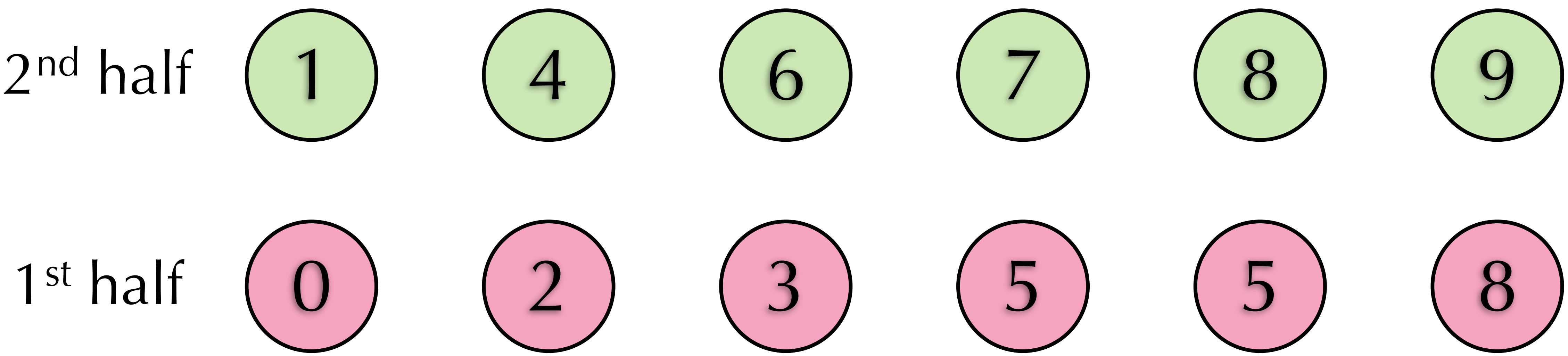
Merge Sort

- A recursive sorting algorithm
- Can be an external sorting in practice.
- Stable!
- Worst case: O(nlogn)
  - NOT average!
- Weakness: need $\Omega(n)$ extra space!

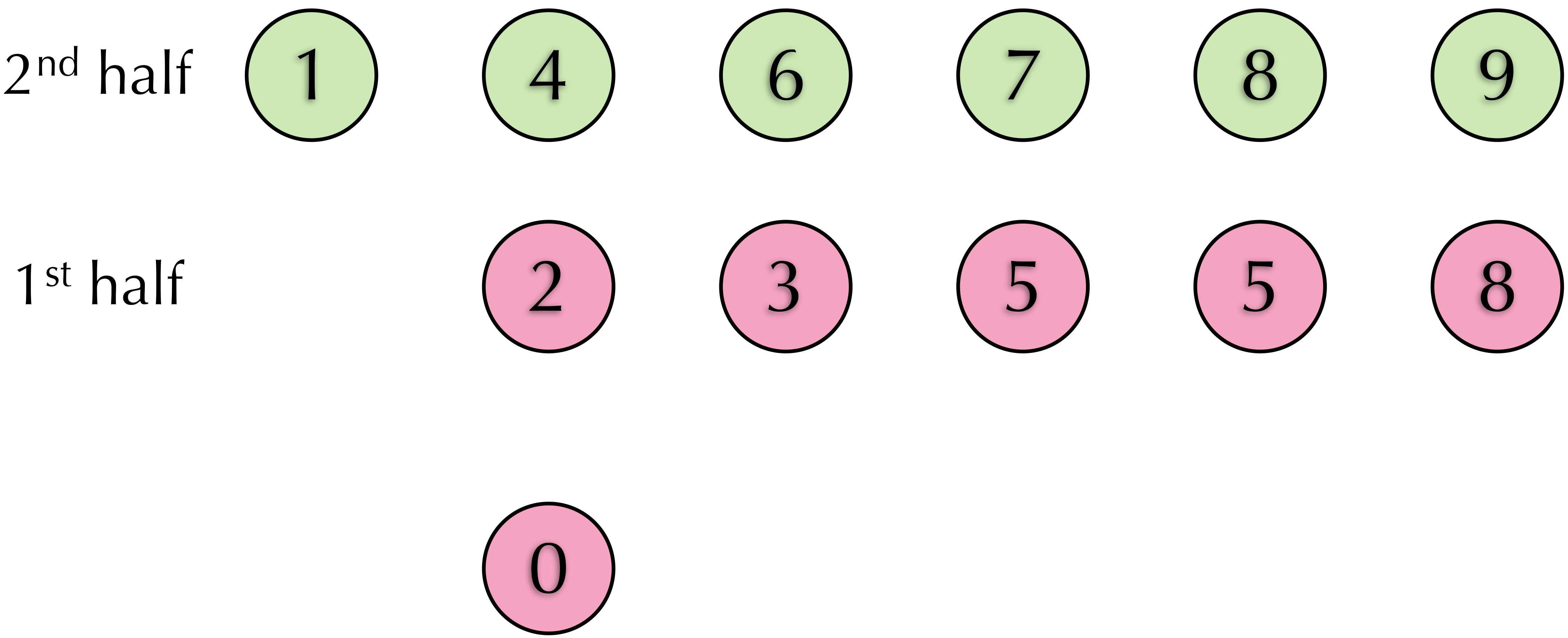
Merge Sort

- Key: merge two sorted arrays into one
- Merge sort A[1..n]
  - If n≤1, then we're done.
    m=n/2
    mergesort(A[1..m])
    mergesort(A[m+1..n])
    B=merge(A[1..m],A[m+1..n])
    For i = 1 to n
        A[i]=B[i]

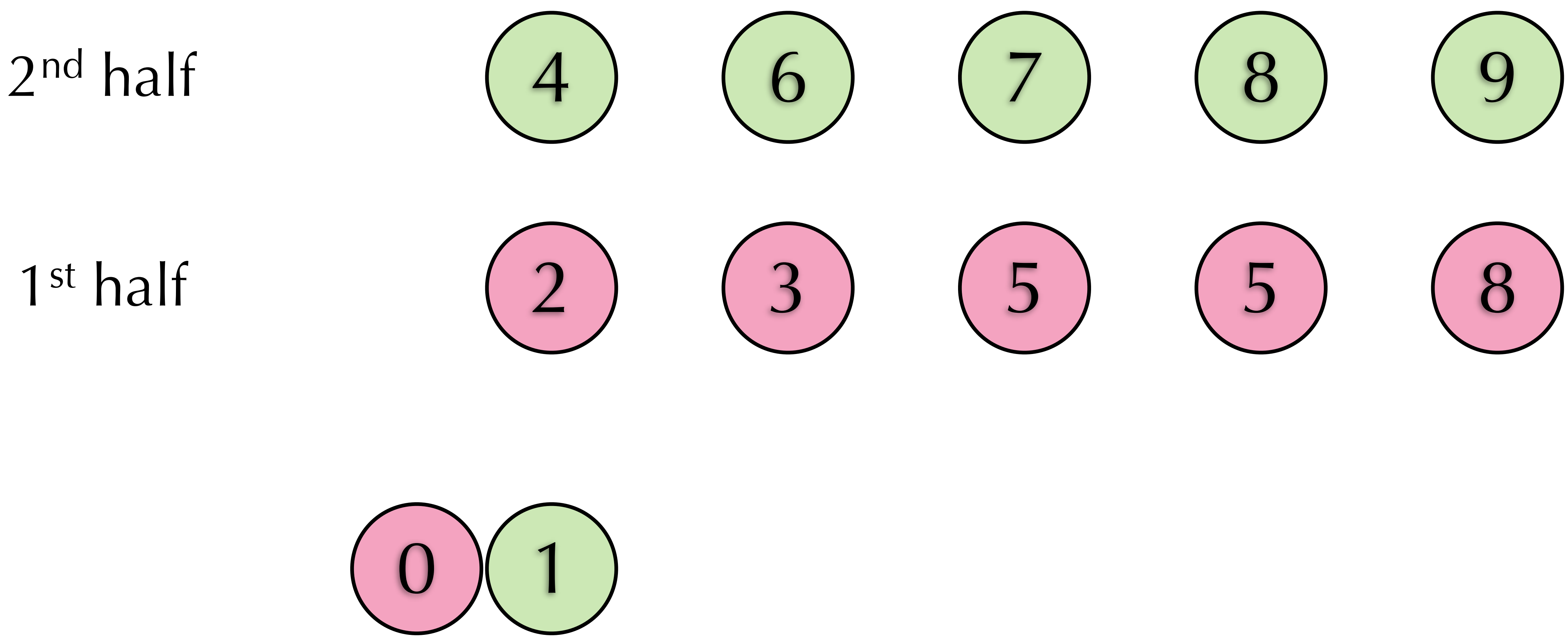
Merge



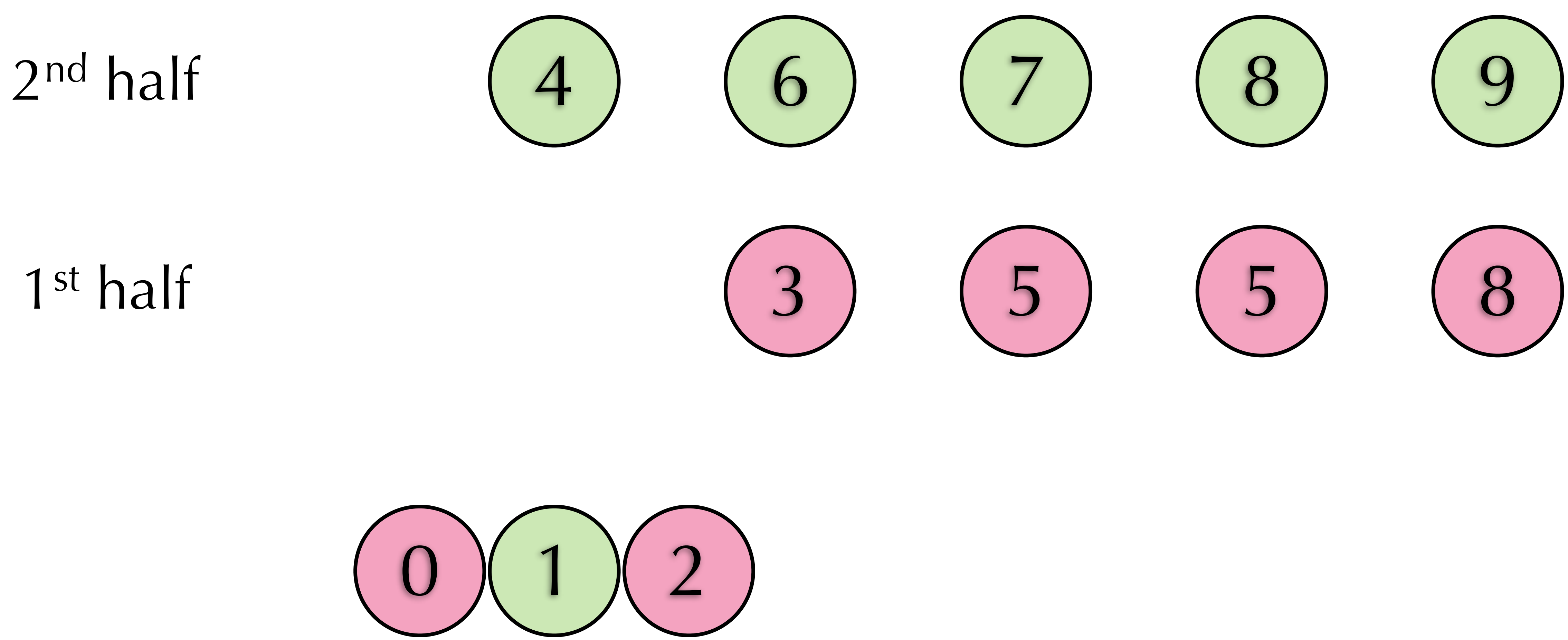
Merge



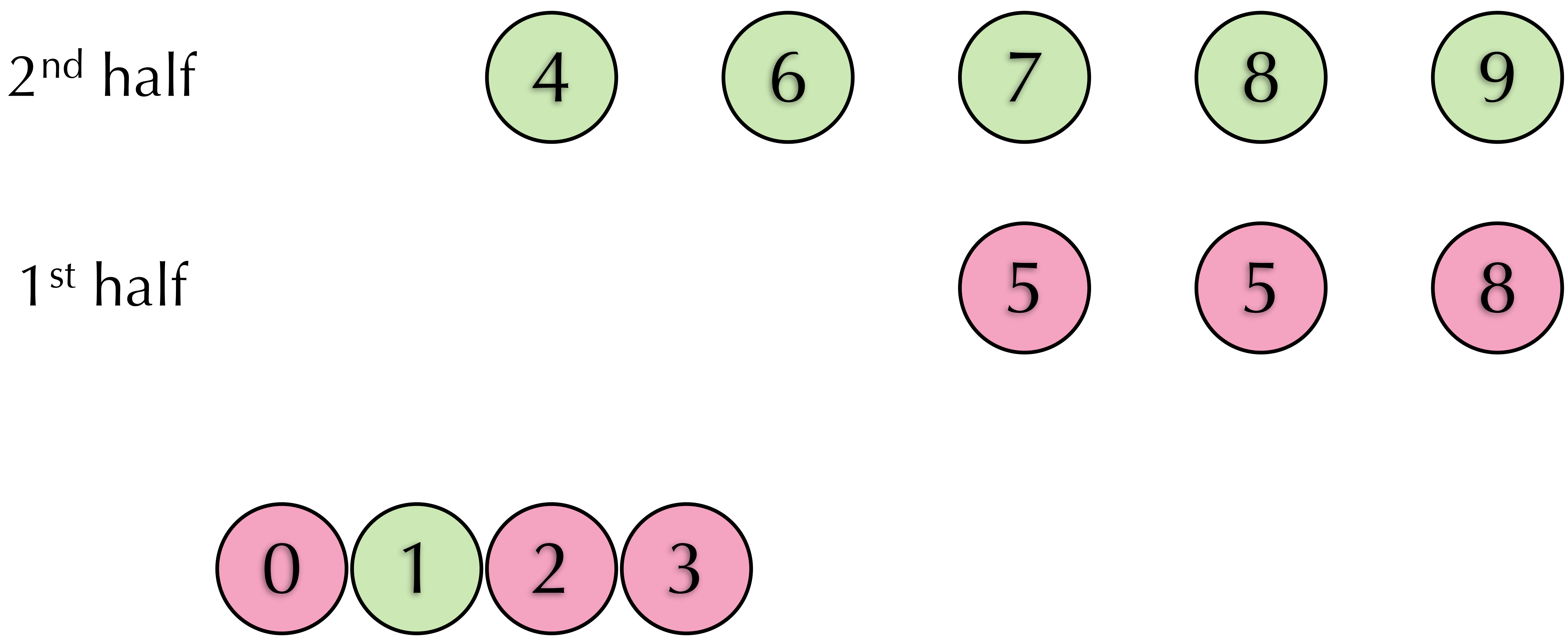
Merge



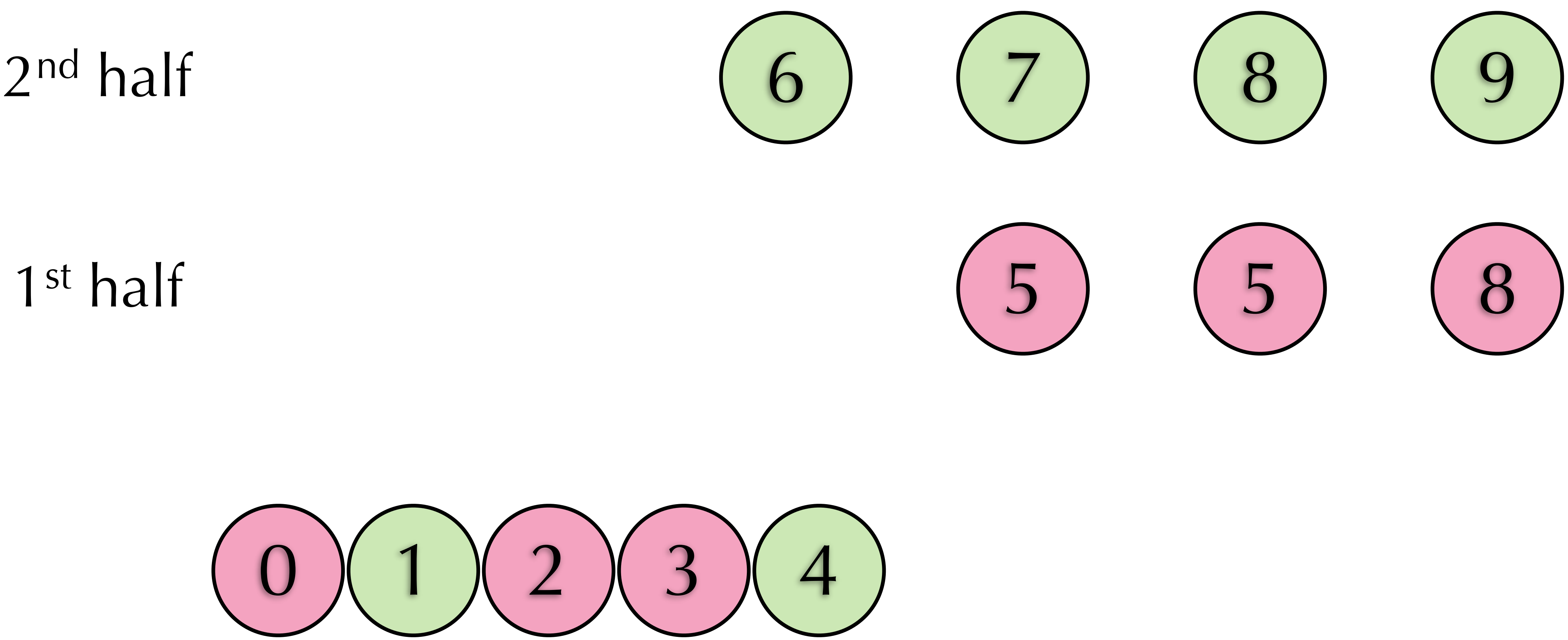
Merge



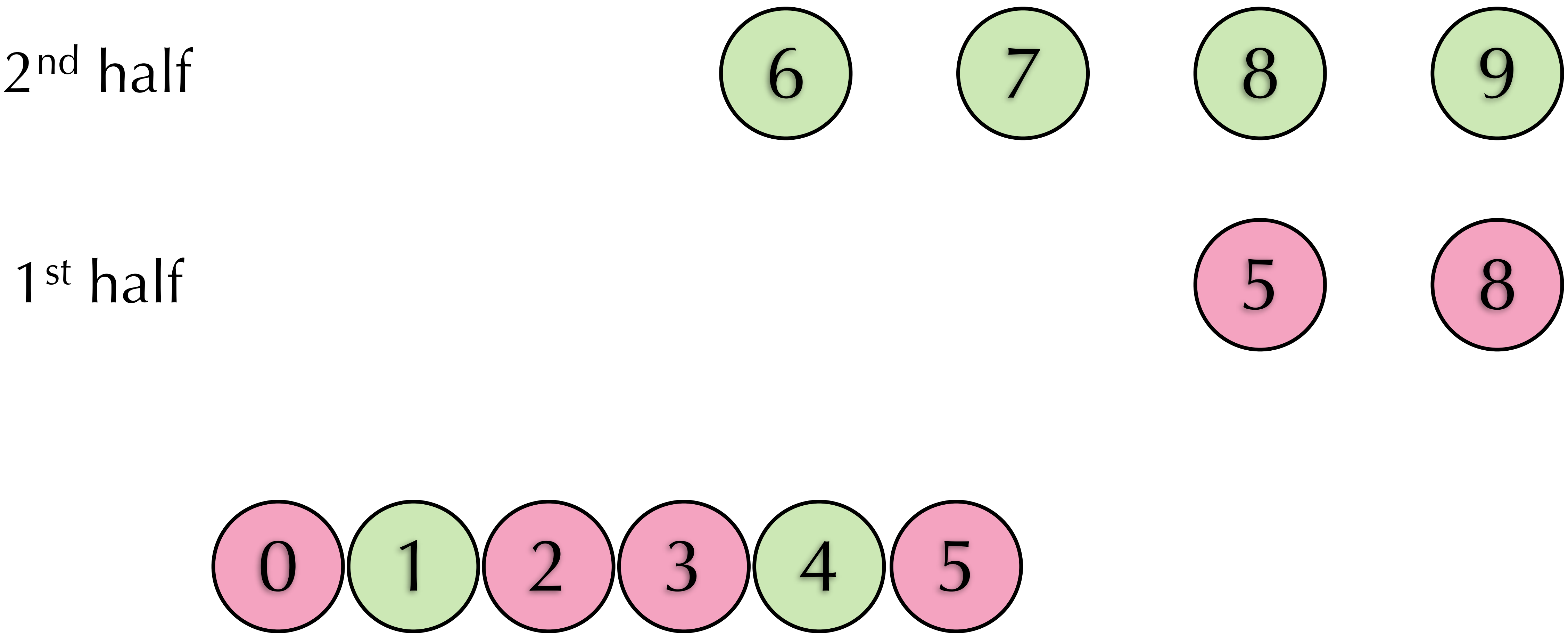
Merge



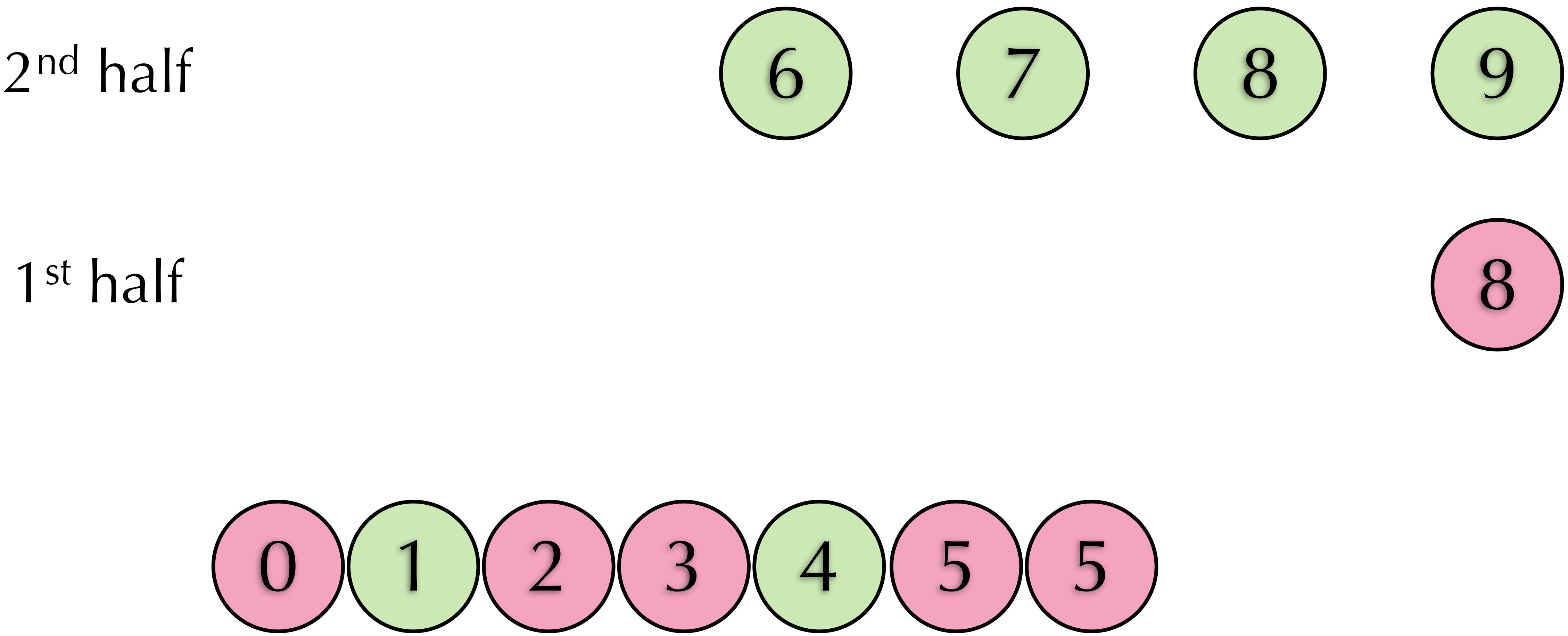
Merge



Merge



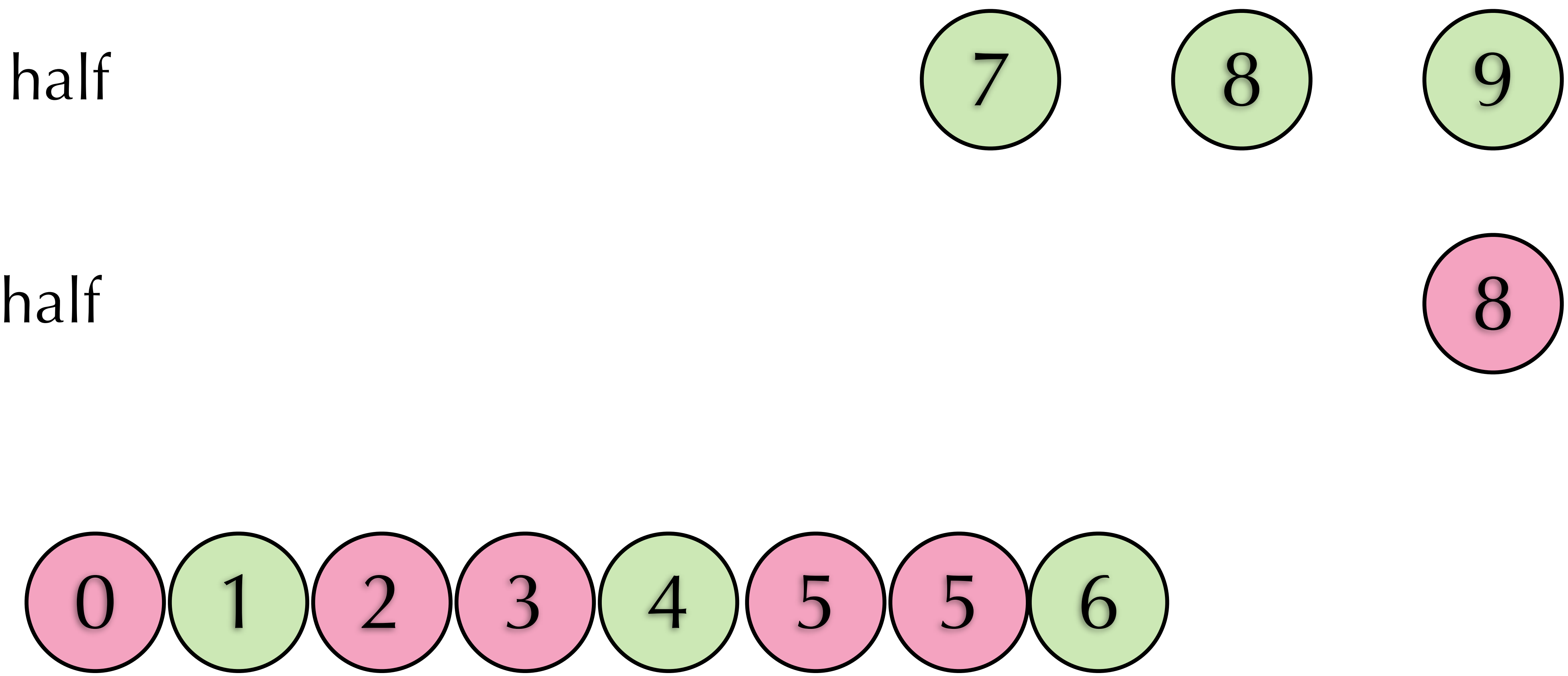
Merge



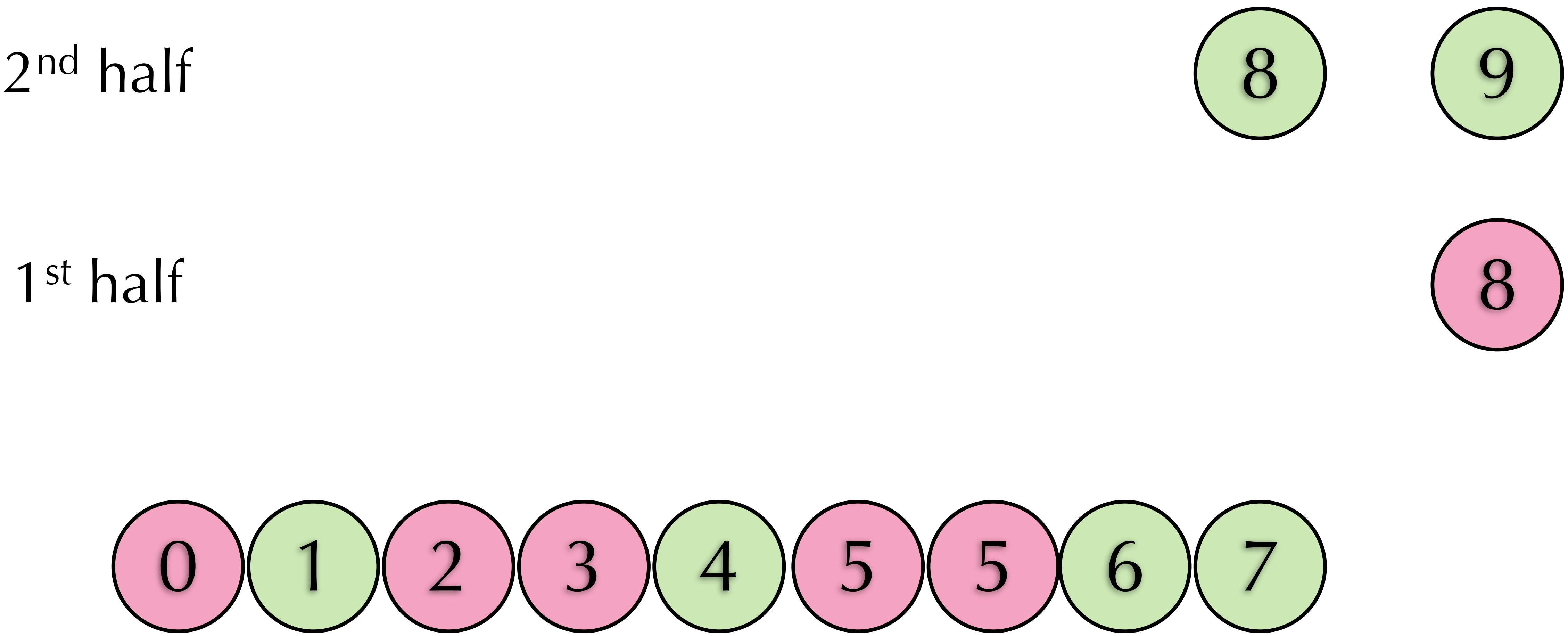
Merge

2nd half

1st half



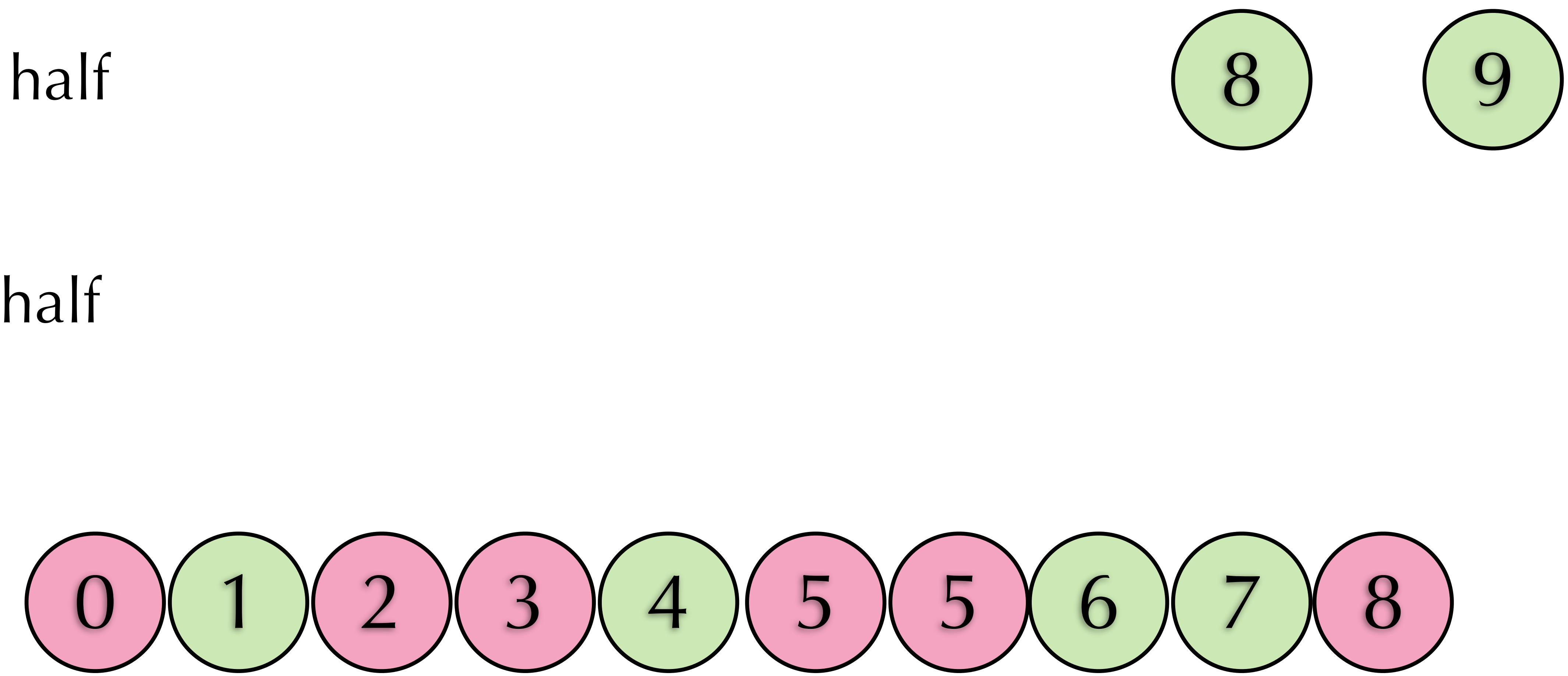
Merge



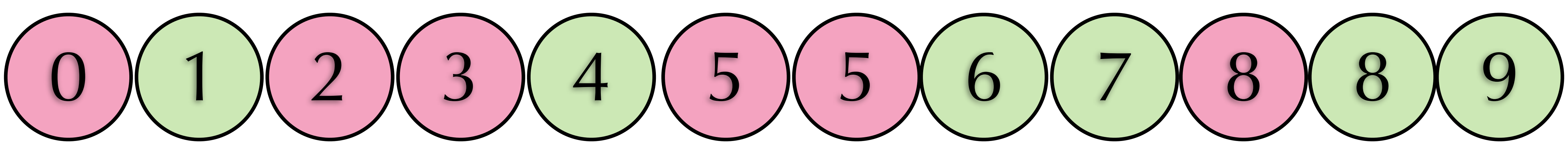
Merge

2nd half

1st half



Merge



Heap Sort

- Use a binary-heap-based priority queue to implement selection sort.
- Construction the initial priority queue: O(nlogn)
- ExtractMin: O(logn)
- Time complexity: O(nlogn)+nO(logn)=O(nlogn)
- Only need constant extra space!

Homework 5.2

- a) What is randomized quicksort? When is it faster than quicksort?
- b) Explain why heapsort is unstable.
- c) Explain why mergesort is slower than quicksort?
- d) Implement heapsort and mergesort, then compare their performance.
- e) How to use `qsort` in stdlib.h?

Counting Sort

- Works only if there are not many kinds of keys. (Non-comparison based sorting)
- Suppose there are m kinds of keys.
  - Keys: $k_1 < k_2 < \ldots < k_m$
  - Prepare m queues $Q_1, \ldots, Q_m$.
  - For each object o of key k_i: enqueue o into Q_i.
  - For i=1 to n: Repeat dequeueing Q_i until empty. (Output object when it is dequeued)
- Stable!

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
							7		

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
							7	8	

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	9
								8	

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	9
							7	8	

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	9
							7	8	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2					7	8	9
		2					7	8	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

Queue Array

0	1	2	3	4	5	6	7	8	9
		2	3				7	8	9
		2					7	8	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

2	2	3	7	7	8	8	9	9

Queue Array

0	1	2	3	4	5	6	7	8	9

Counting Sort

- Queue array takes a lot extra space.
- Let $c_{count}[i]$ be the numbers of elements whose key is k_i.
- Let $c_{umulated\_count}[i]=\Sigma_{i' \leq i} c_{count}[i']$.
- For j=n downto 1
  If A[j]'s key=k_i
    $B[c_{umulated\_count}[i]]=A[j]$
    $c_{umulated\_count}[i]=c_{umulated\_count}[i]-1$

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	2	3	3	3	3	5	7	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

		3						

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	2	3	3	3	3	5	7	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

	2	3						

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	2	2	3	3	3	5	7	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

	2	3						9

cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	5	7	9

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

	2	3		7				9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	5	7	8

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

	2	3		7		8		9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	4	7	8

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

	2	3		7		8	9	9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	4	6	8

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

2	2	3		7		8	9	9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	4	6	7

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

2	2	3		7	8	8	9	9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	0	2	3	3	3	4	6	7

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

2	2	3	7	7	8	8	9	9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	0	2	3	3	3	4	5	7

Counting Sort

A

7	8	2	9	8	7	9	2	3

B

2	2	3	7	7	8	8	9	9

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	0	2	3	3	3	3	5	7

Counting Sort

A

7	8	2	9	8	7	9	2	3

Count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	0	0	0	2	2	2

B

2	2	3	7	7	8	8	9	9

Cumulated_count

0	1	2	3	4	5	6	7	8	9
0	0	0	2	3	3	3	3	5	7

Radix Sort

MSD: Most Significant Digit

LSD: Least Significant Digit

- Suppose all keys are d digits number based on m.
- We can sort n numbers by d stable sorts
  - Sort them according to LSD
  - Sort them according to 2nd-LSD.
  - …
  - Sort them according to 2nd-MSD.
  - Sort them according to MSD.
- Note: Counting sort is stable.

Radix Sort

A

17	28	32	29	28	17	19	52	43

Queue Array

0	1	2	3	4	5	6	7	8	9

Radix Sort

LSD: Least Significant Digit

A

17	28	32	29	28	17	19	52	43

Queue Array

0	1	2	3	4	5	6	7	8	9
		32	43				17	28	29
		52					17	28	19

Radix Sort

A

17	28	32	29	28	17	19	52	43

32	52	43	17	17	28	28	29	19

Queue Array

0	1	2	3	4	5	6	7	8	9

Radix Sort

MSD: Most Significant Digit

A

17	28	32	29	28	17	19	52	43

32	52	43	17	17	28	28	29	19

Queue Array

0	1	2	3	4	5	6	7	8	9
	17	28	32	43	52				
	17	28							
	19	29							

Radix Sort

A

17	28	32	29	28	17	19	52	43

32	52	43	17	17	28	28	29	19

17	17	19	28	28	29	32	43	52

Queue Array

0	1	2	3	4	5	6	7	8	9

Homework 5.3

- a) Can we replace the queue array in counting sort by a stack array? If yes, how should we modify the algorithm?
- b) Give an algorithm to sort n integers $a_1,\ldots,a_n \in [0,n^5)$ in $O(n)$ time.
- c) Integral types in C all have a fixed number of bits. Therefore, we can sort integers by radix sort in O(n) time. Will this method faster than quick sort on your computer? Give your answer and explain why.