Sorting

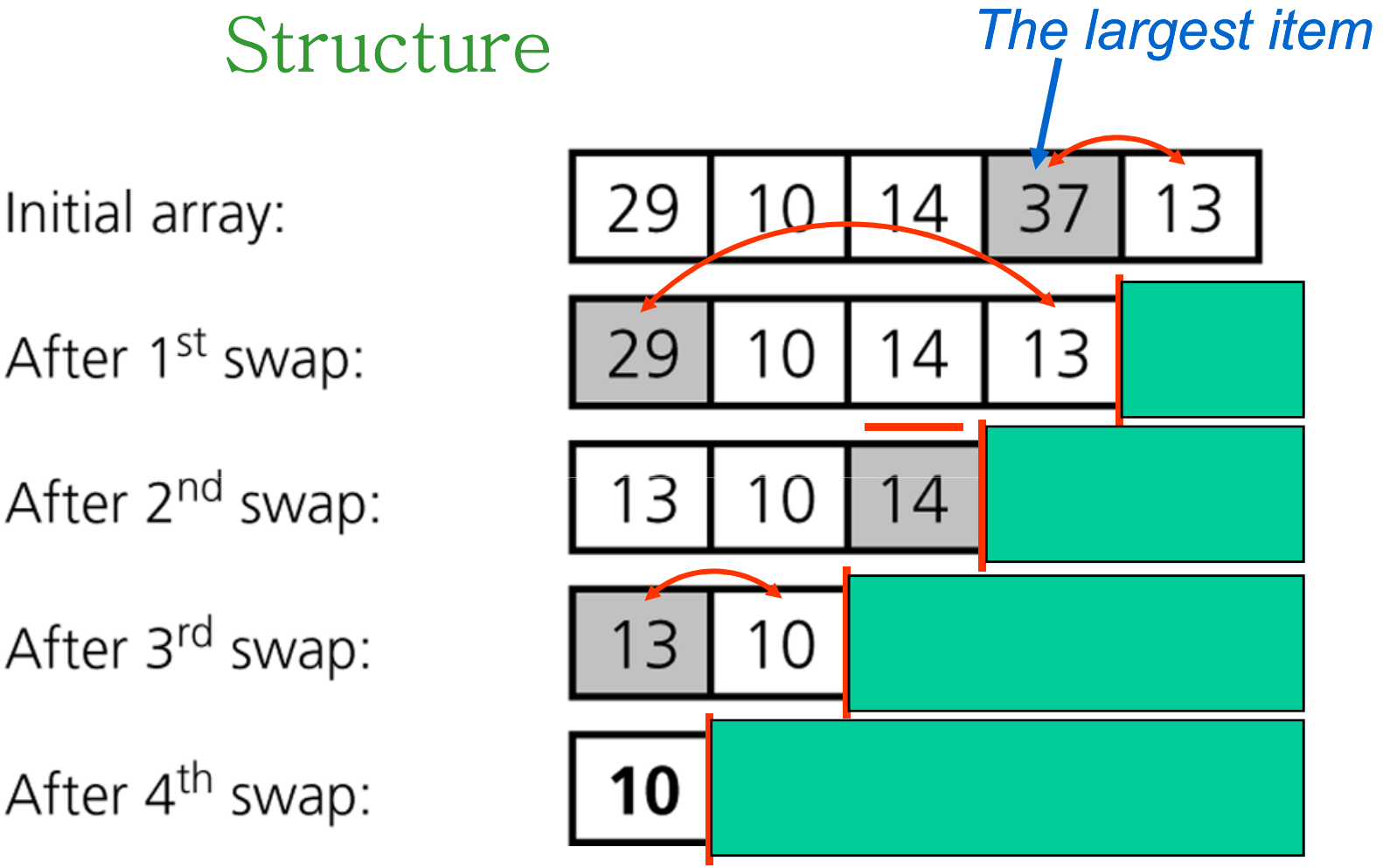
Sorting Algorithms

- Between $O(n^2)$ and $O(n\log n)$
- For special input, $O(n)$ sorting is possible
  - E.g., input – integer between $-O(n)$ and $O(n)$

Selection Sort

- For each loop
  - Find max
  - Swap max and rightmost element
  - Exclude rightmost element
- Loop until one element left

Finding the Recursive Structure



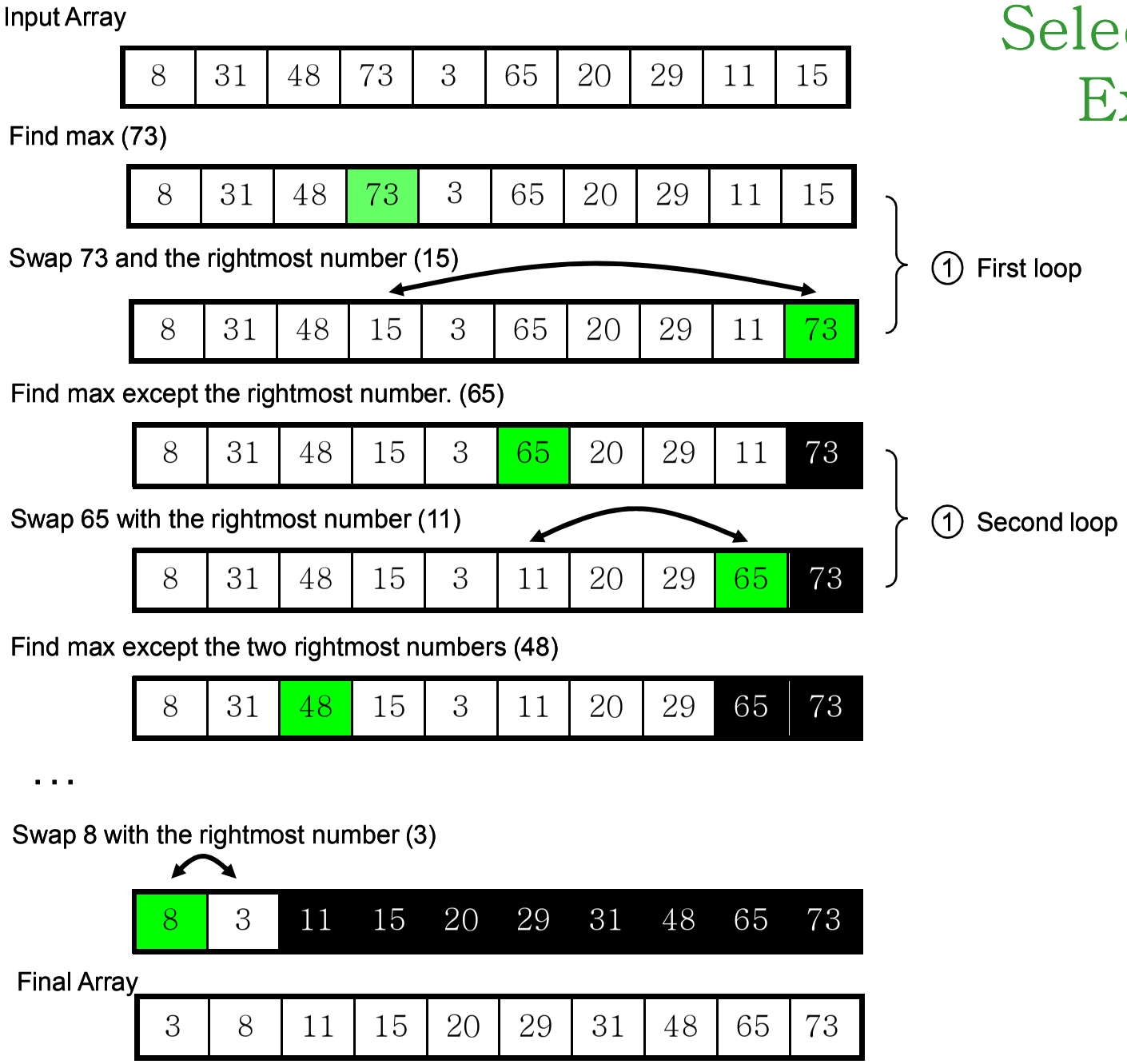
✓ Running time: $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = O(n^2)$ ← Worst case, Average case

```
selectionSort(A[], n)      ▷ Sort array A[1 ... n]
{
        for last ← n downto 2 {                                  -------- ①
                Find max A[k] among A[1 ... last];               -------- ②
                A[k] ↔ A[last];  ▷ Swap A[k] and A[last]         ----- ③
        }
}
```

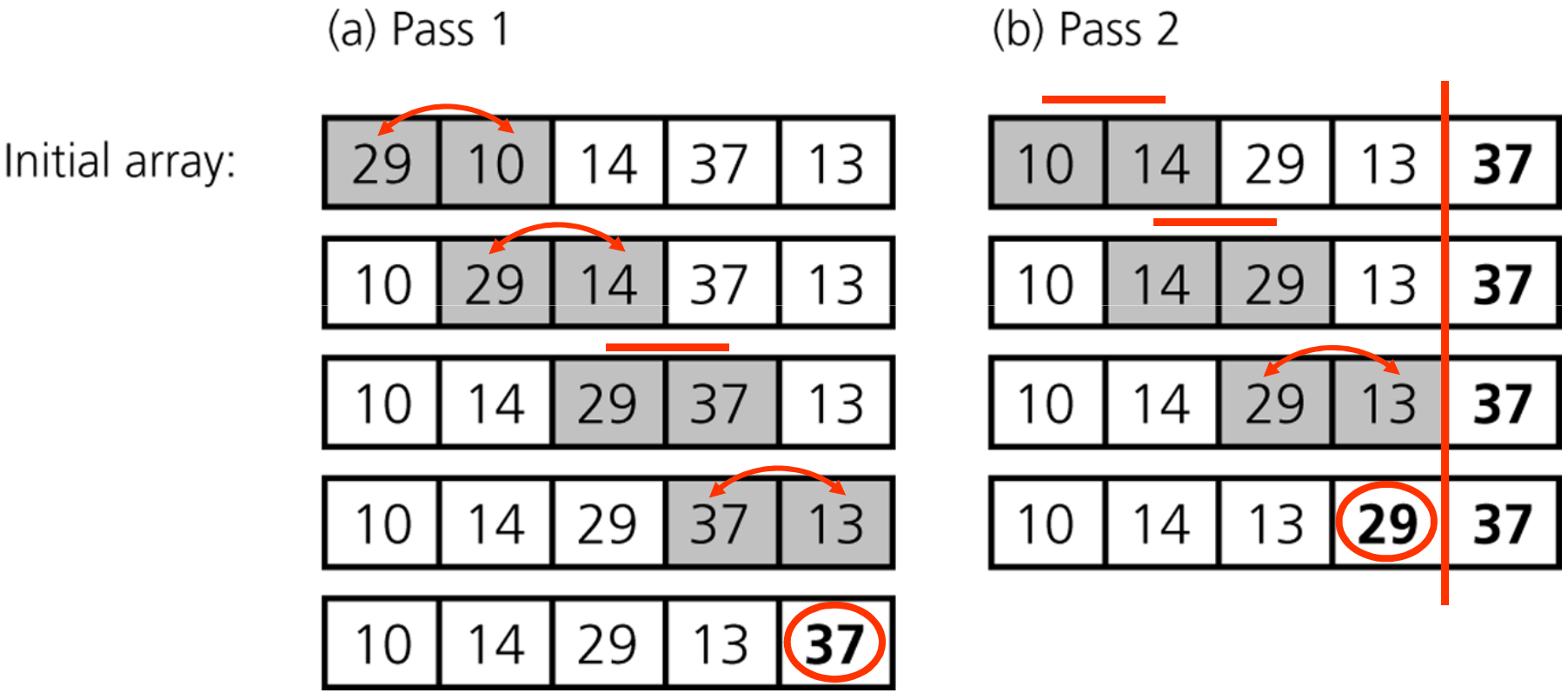
✓ Running time:
- — **for** loop in ① : n-1 times
- — # of comparisons to find max in ② : n-1, n-2, …, 2, 1
- — swap in ③ : constant time

✓ $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = O(n^2)$

Selection Sort Example



Bubble Sort



✓ Running time: $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = O(n^2)$ ← Worst case / Average case

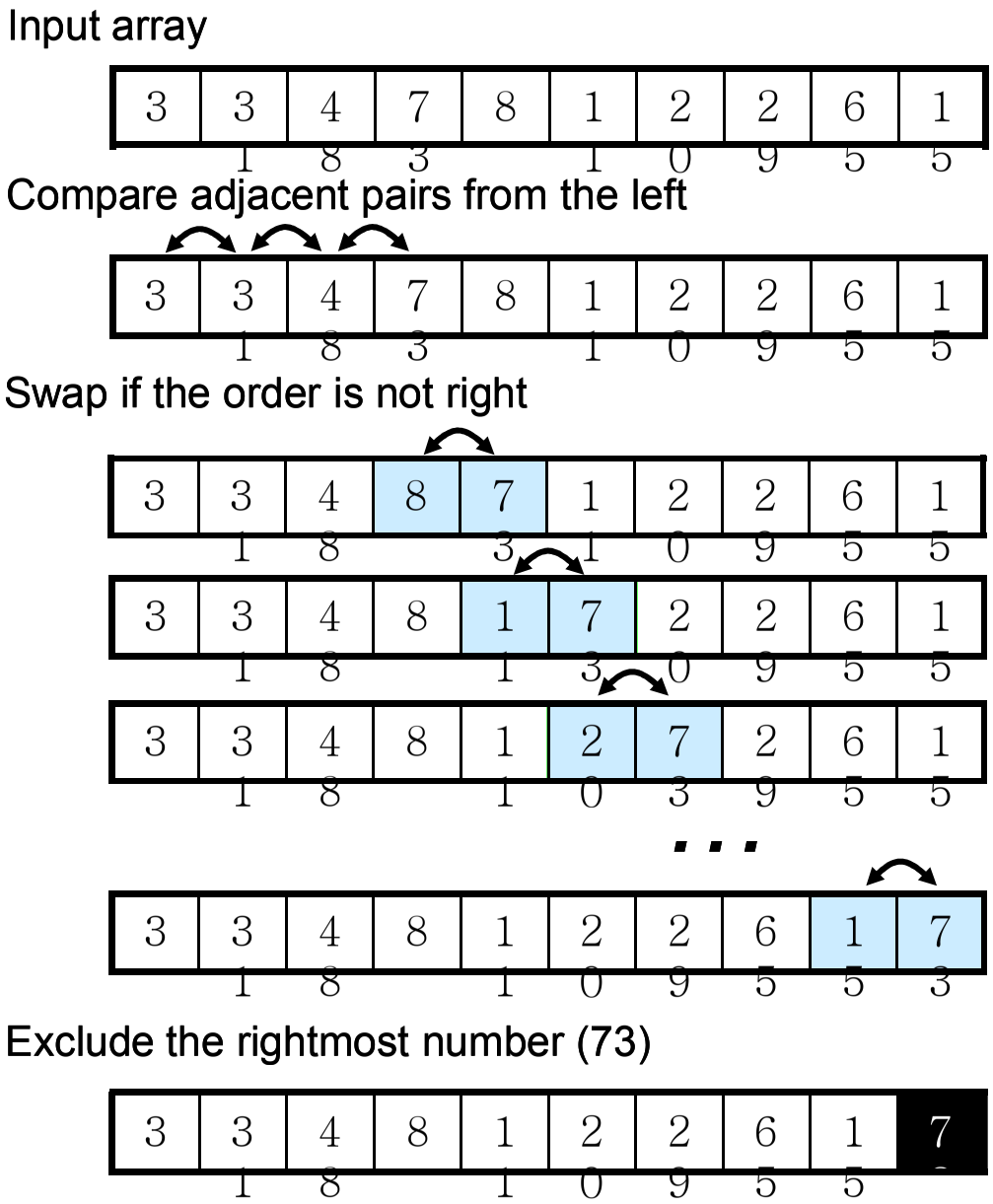
```
bubbleSort(A[], n)    ▷ Sort A[1 ... n]
{
        for last ← n downto 2        ------------------ ①
                for i ← 1 to last-1   ------------------ ②
                        if (A[i] > A[i+ 1]) then A[i] ↔ A[i+ 1]; ▷
    Swap-- ③
}
```

✓ Running time:

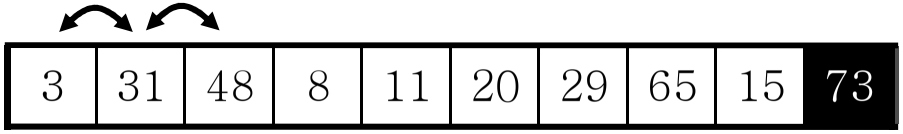
- ① : **for** loop – n-1 times
- ② : **for** loop – n-1, n-2, …, 2, 1 times
- ③ : constant time

✓ $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = O(n^2)$

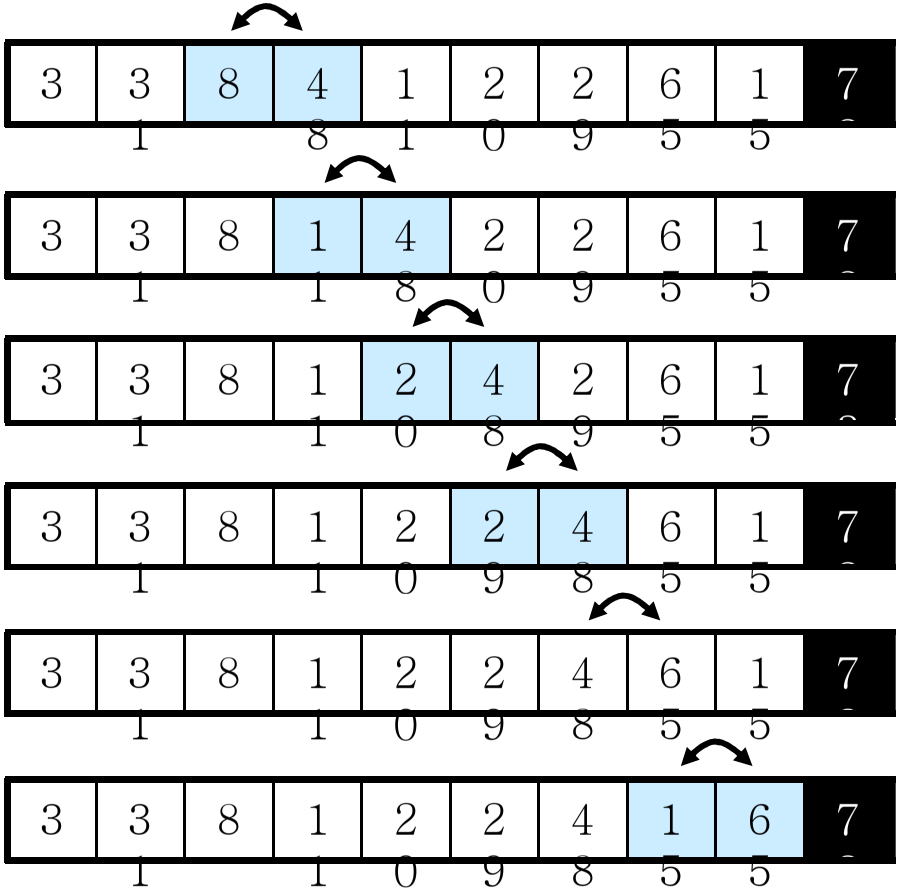
Bubble Sort Example



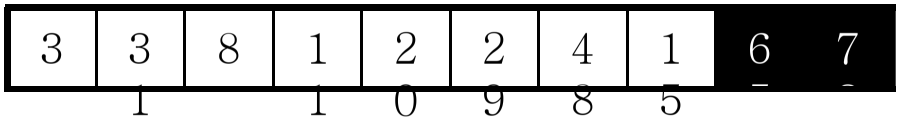
Starting from the left, compare adjacent pairs



Swap if the pair is not orderly

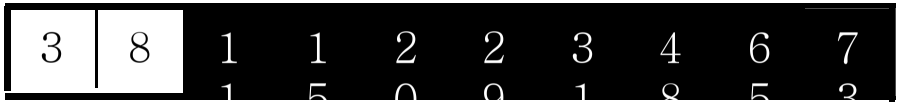


Exclure the rightmost value (65)

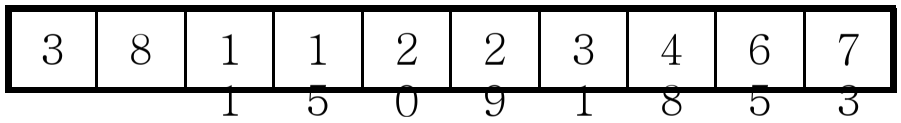


Keep excluding the values by repeating the loop

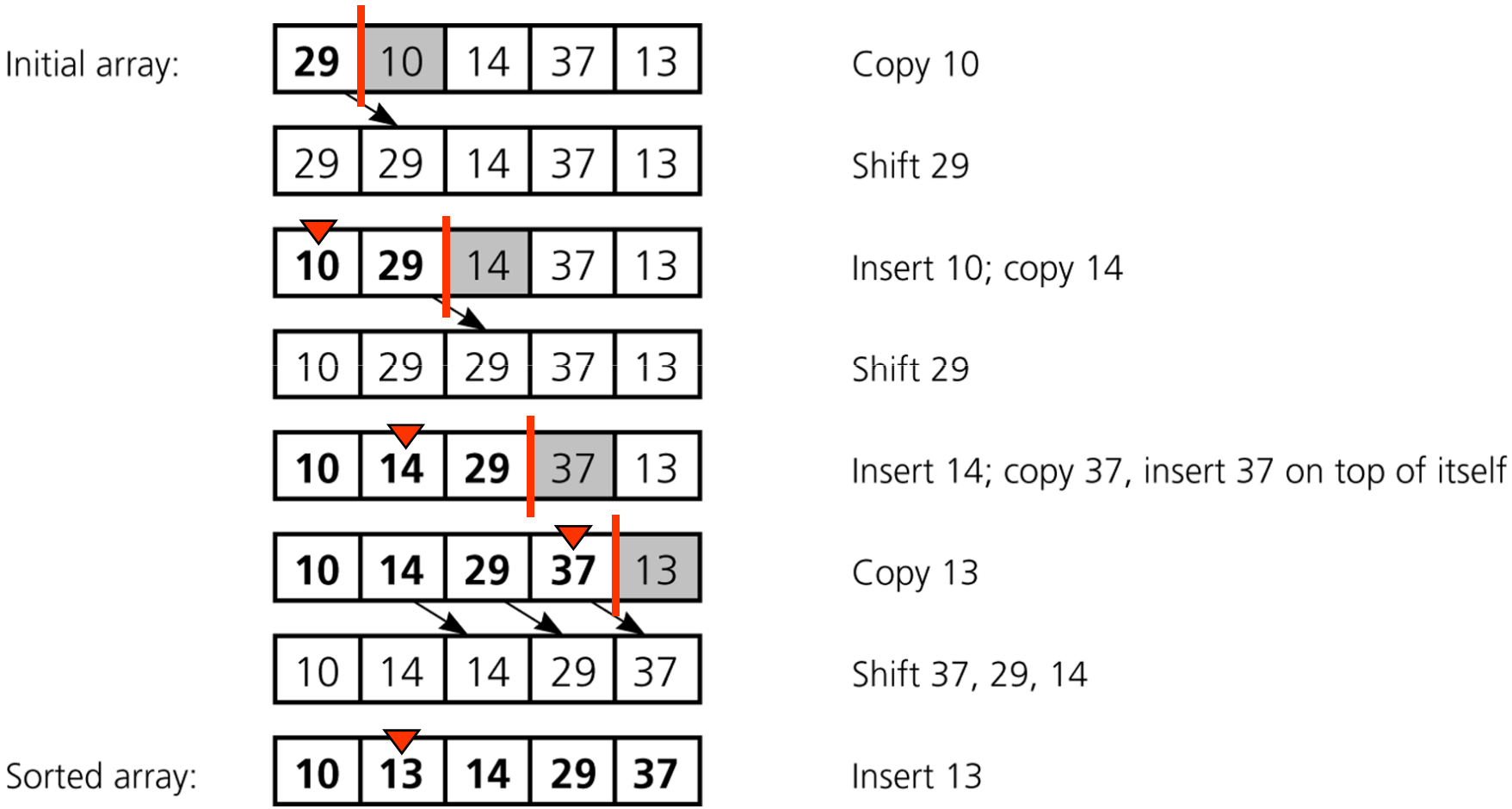
. . .



Process the final two elements



Insertion Sort



✓ Running time: $O(n^2)$ ← Worst case: $1+2+\cdots+(n-2)+(n-1)$

← Average case: $\frac{1}{2}\,(1+2+\cdots+(n-2)+(n-1))$

```
insertionSort(A[], n)        ▷ Sort A[1 ... n]
{
        for i ← 2 to n      ---------------------------- ①
                Insert A[i] into the right position in A[1 ... i]; ---- ②
}
```

✓ Running time:

— ① : **for** loop – n-1 times

— ② : i-1 times comparison in the worst case

✓ Worst case: $1+2+\cdots+(n-2)+(n-1) = O(n^2)$

✓ Average case: $\frac{1}{2}(1+2+\cdots+(n-2)+(n-1)) = O(n^2)$

Inductive Verification of Insertion Sort

- A[1]
  - Sorted
- If A[1 … k] sorted

  ➔ By the insertion in ②, A[1 … k+ 1] are sorted

Mergesort

```
mergeSort(A[ ], p, r)
▷ Sort A[p ... r]
{
        if (p < r) then {
                q ← (p+ q)/2;   ------------  ①   ▷ Midpoint of p, q
                mergeSort(A, p, q);  --------  ②   ▷ Sort the left
                mergeSort(A, q+ 1, r); ------  ③   ▷ Sort the right
                merge(A, p, q, r);   ---------  ④   ▷ Merge
        }
}

merge(A[ ], p, q, r)
{
        Two sorted array A[p ... q] and A[q+ 1 ... r]
        Merge them into one array A[p ... r]
}
```

Mergesort Example

Input array given

31	3	65	73	8	11	20	29	48	15

Partition the array

31	3	65	73	8	11	20	29	48	15

— (1)

Sort them independently

3	8	31	65	73	11	15	20	29	48

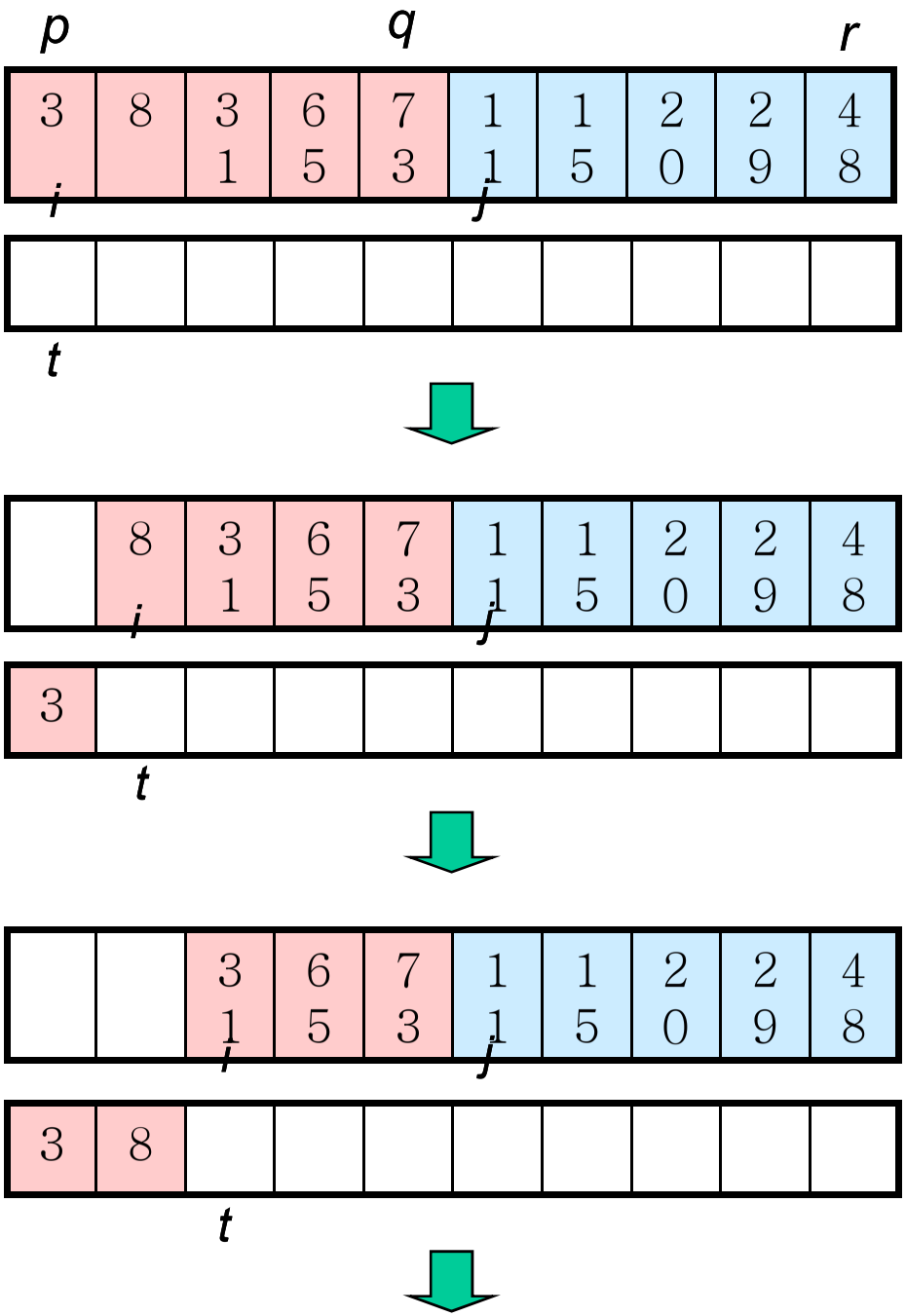
— (2) (3)

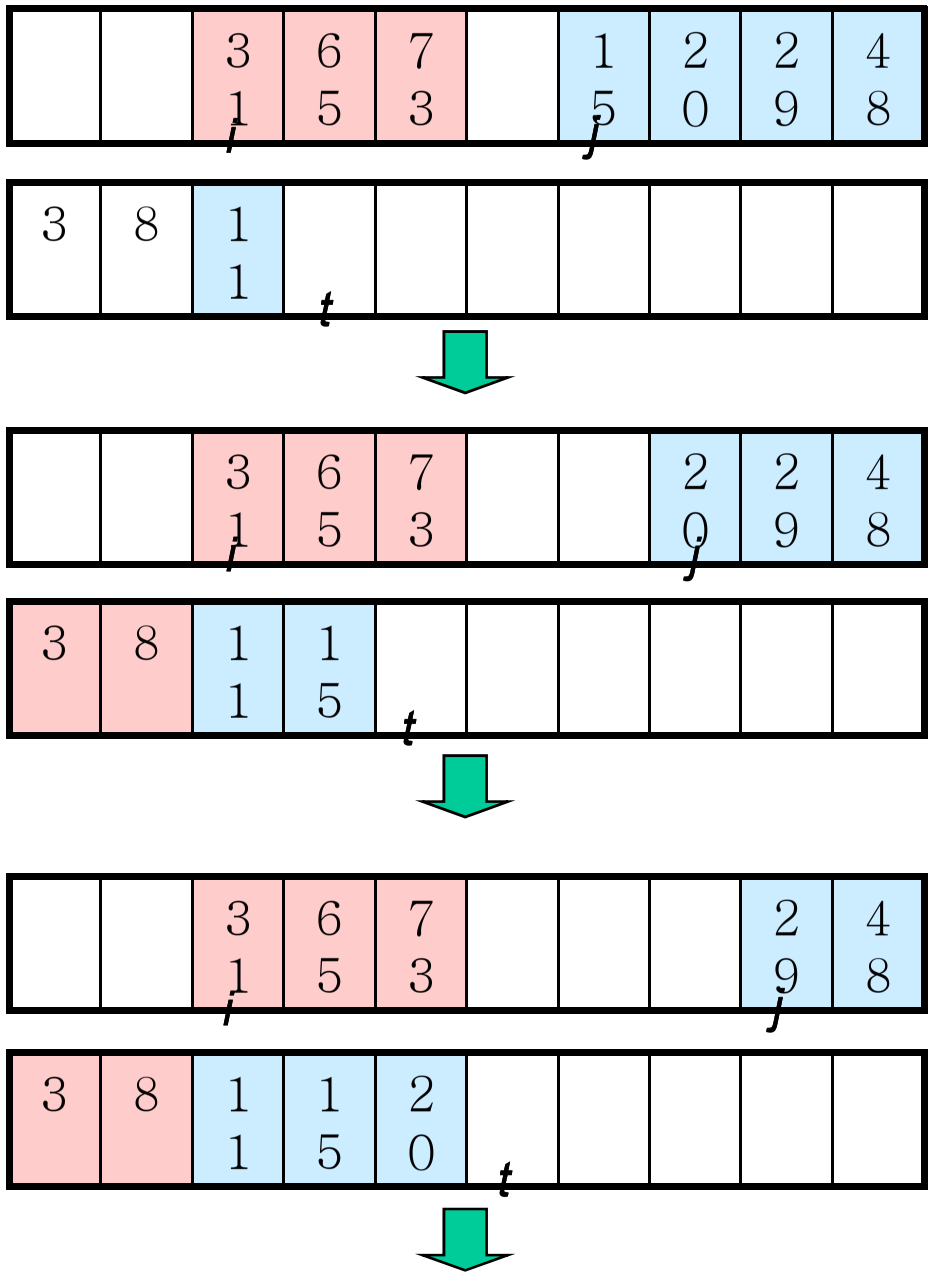
Merge

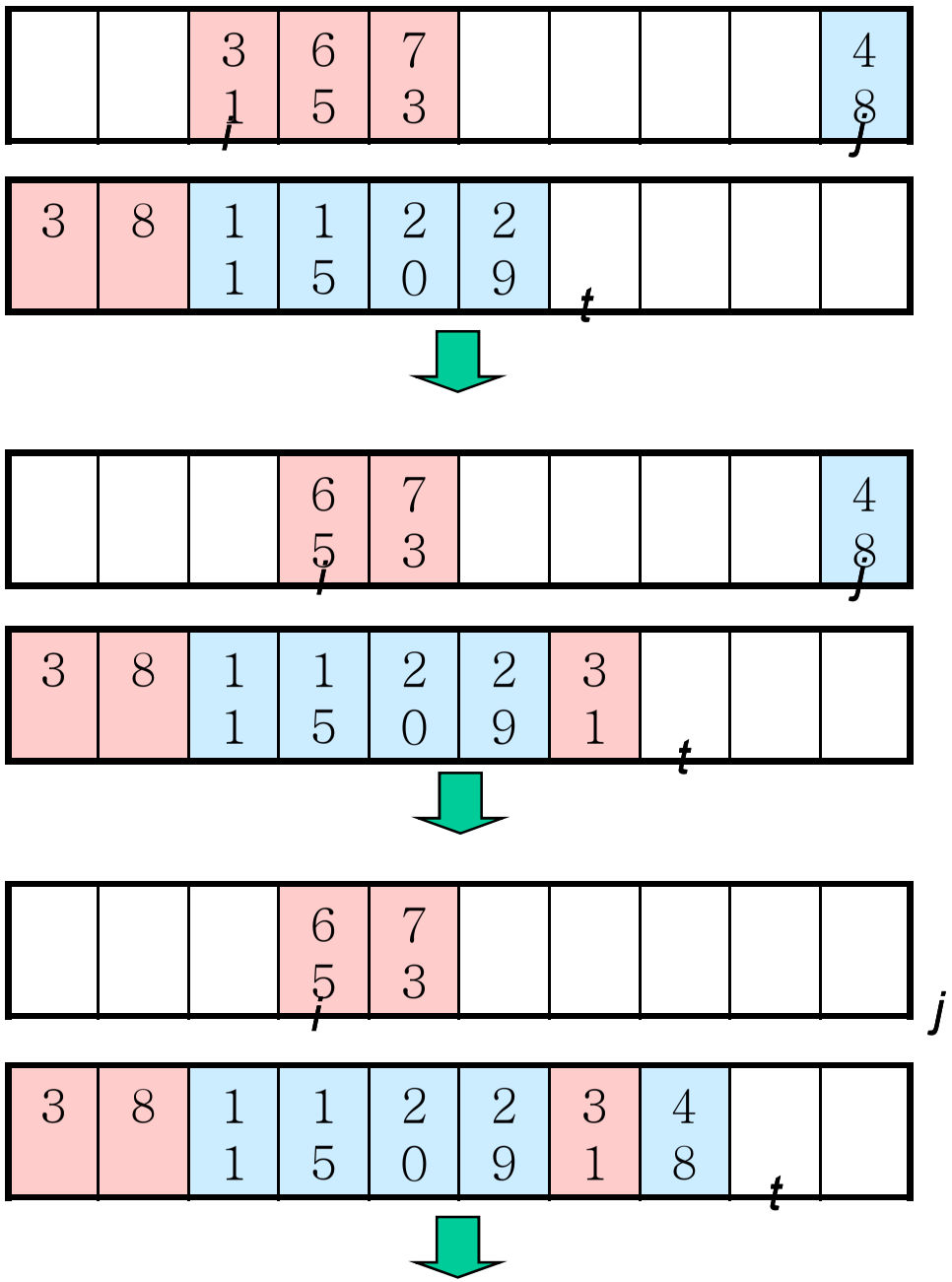
3	8	11	15	20	29	31	48	65	73

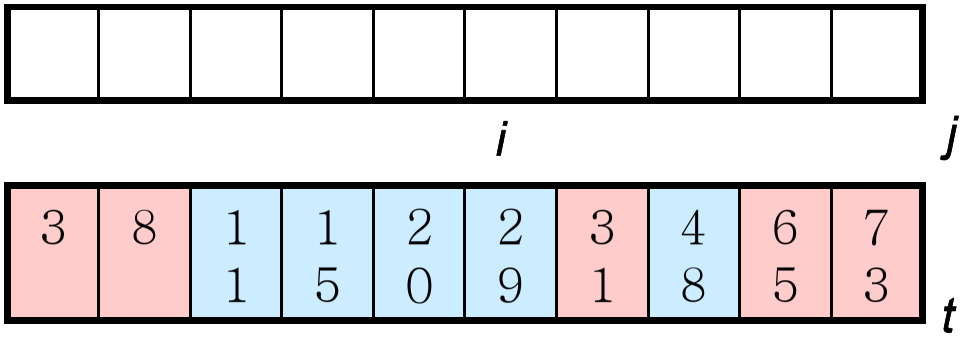
— (4)

Merge Example

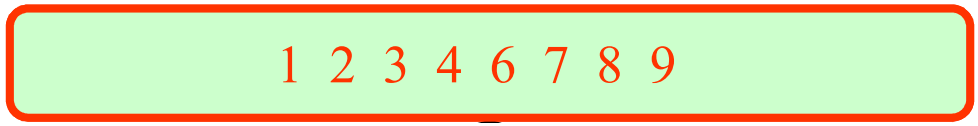



31 65 73 15 20 29 48
i
j
38 11
t
31 65 73 20 29 48
i
j
38 11 15
t
31 65 73 29 48
i
j
38 11 15 20
t


31 65 73 48
i
j
38 11 15 20 29
t
65 73 48
i
j
38 11 15 20 29 31
t
65 73
i
j
38 11 15 20 29 31 48
t


i
j
3
8
11
15
20
29
31
48
65
73
t

Animation (Mergesort)



✓ Running time: $O(n \log n)$

Quicksort

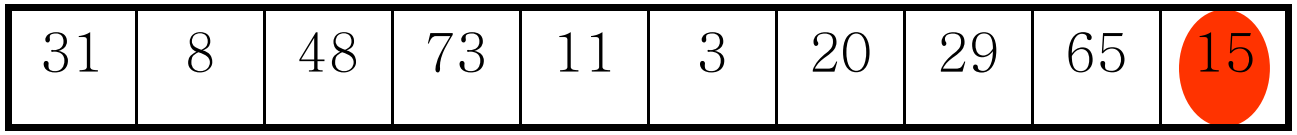
```
quickSort(A[], p, r)    ▷ Sort A[p ... r]
{
        if (p < r) then {
           q = partition(A, p, r);    ▷ Partition
           quickSort(A, p, q-1);      ▷ Sort the left
           quickSort(A, q+ 1, r);     ▷ Sort the right
        }
}


partition(A[], p, r)
{
        Arrange the elements in A[p ... r] according to A[r]
        Return the position of A[r];
}
```

Animation (Quicksort)

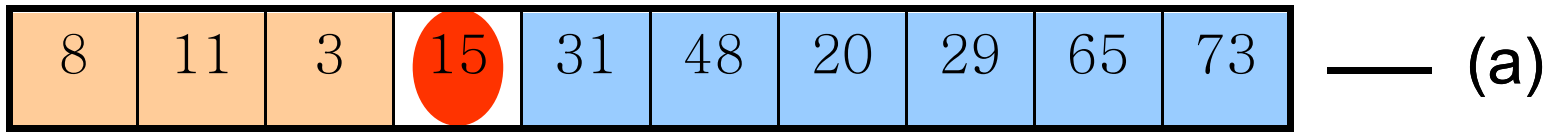
- ✓ Average running time: $O(n\log n)$
- ✓ Worst running time: $O(n^2)$

Quicksort Example

In the input array, set the first value as a pivot

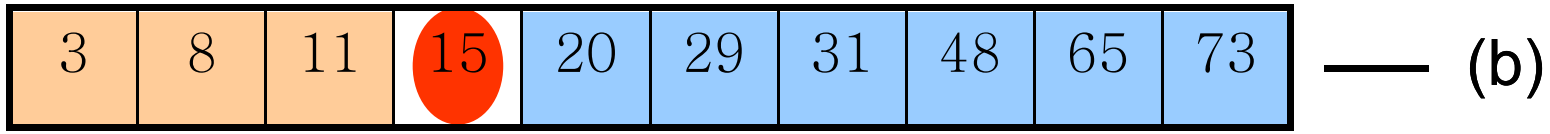


Partition the input array so that smaller elements on the left, and bigger elements on the right



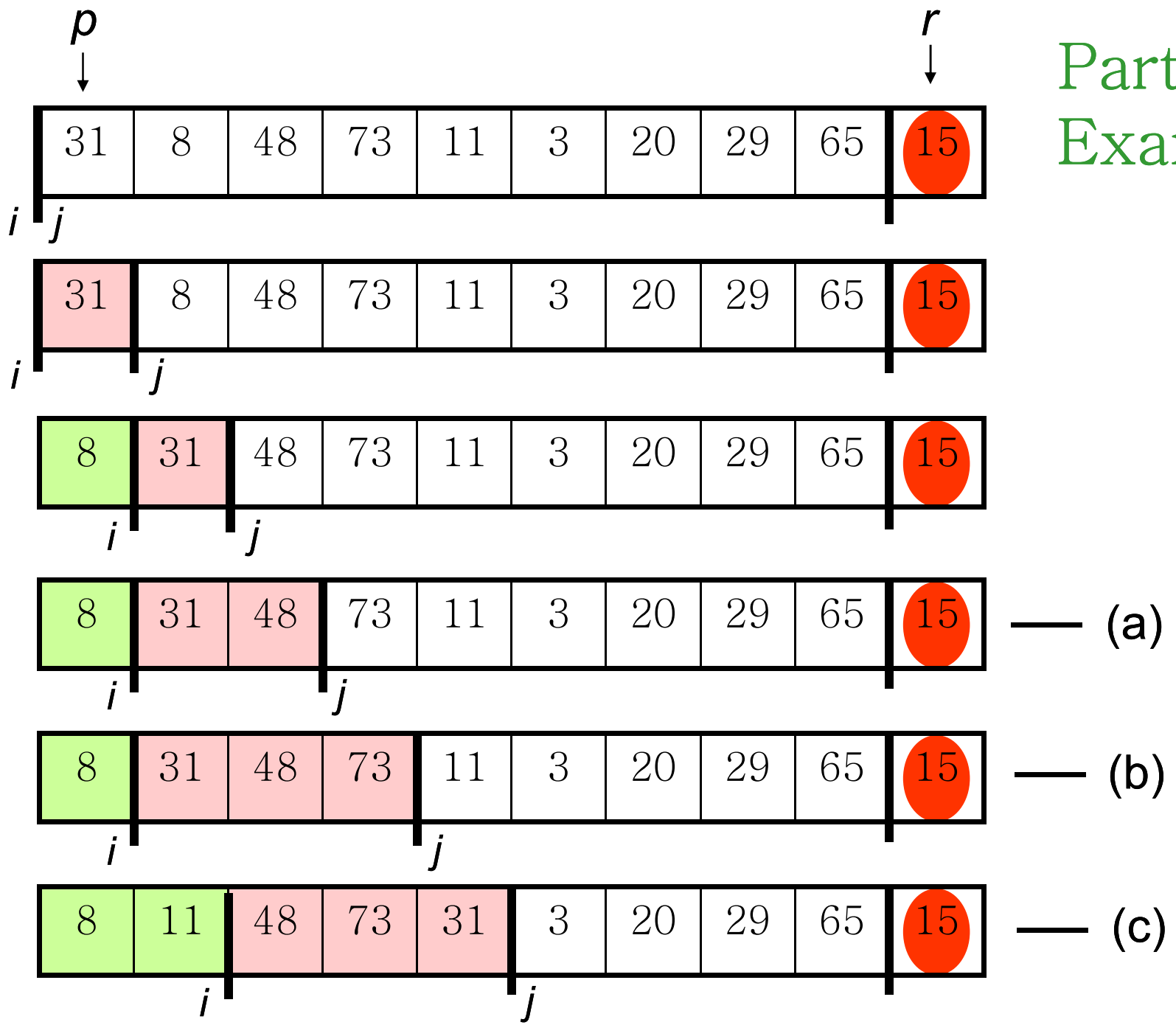
— (a)

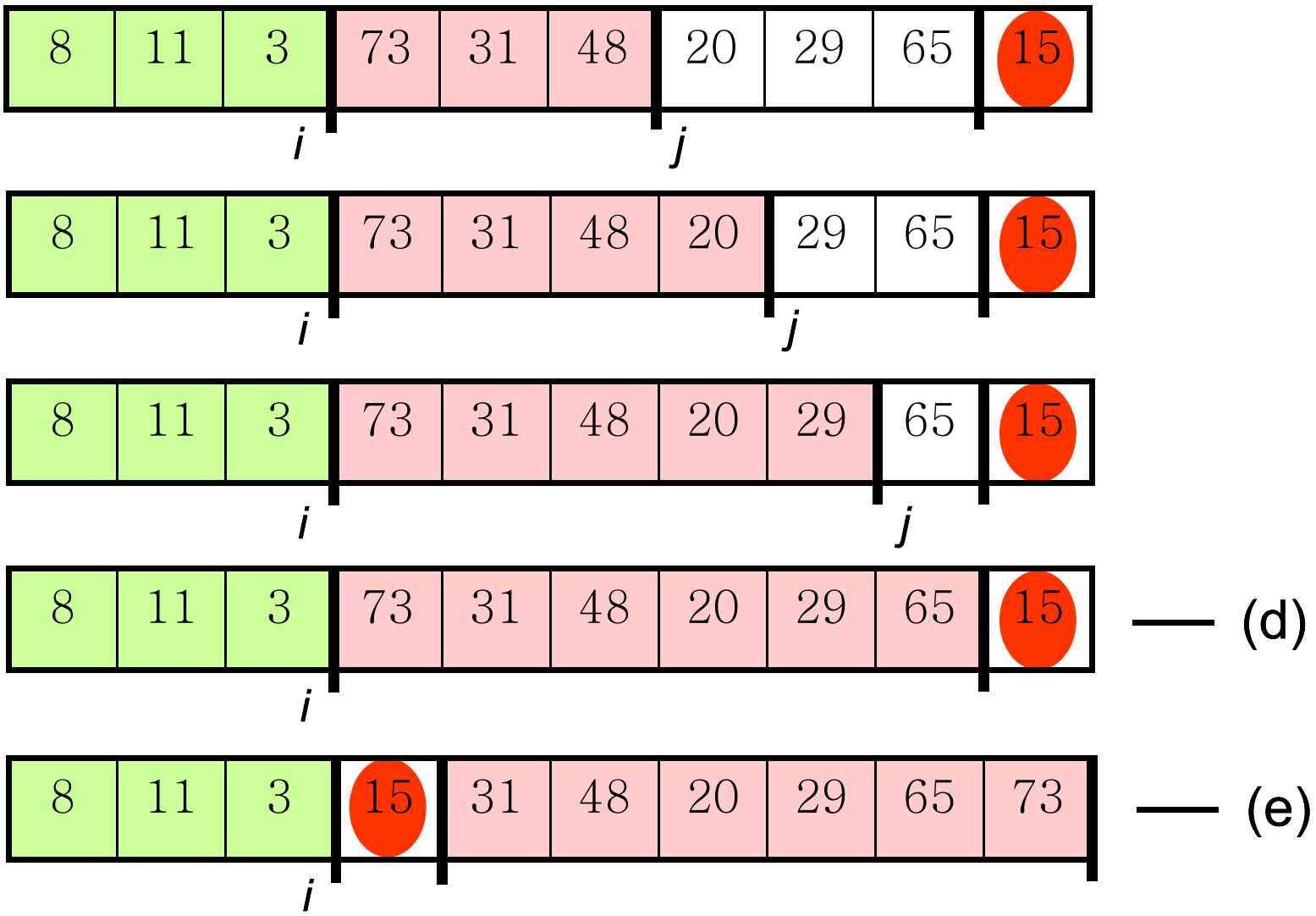
Sort the left and the right independently



— (b)

Partition Example




8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
i
j
8 11 3 73 31 48 20 29 65 15
i
(d)
8 11 3 15 31 48 20 29 65 73
i
(e)

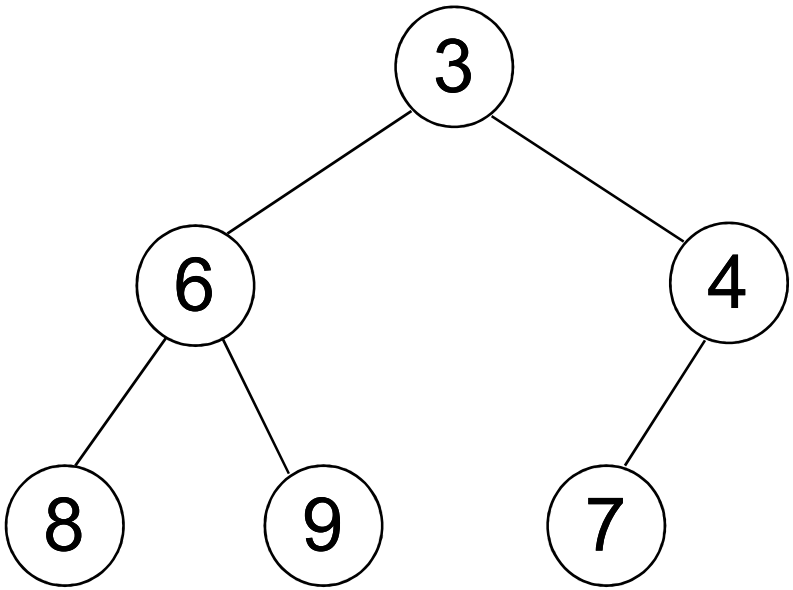
Heapsort

- Heap
  - Complete binary tree that satisfies the following
    - Each node's value is not bigger than its children
- Heapsort
  - Convert the input array to a Heap, and remove top element from the Heap one by one

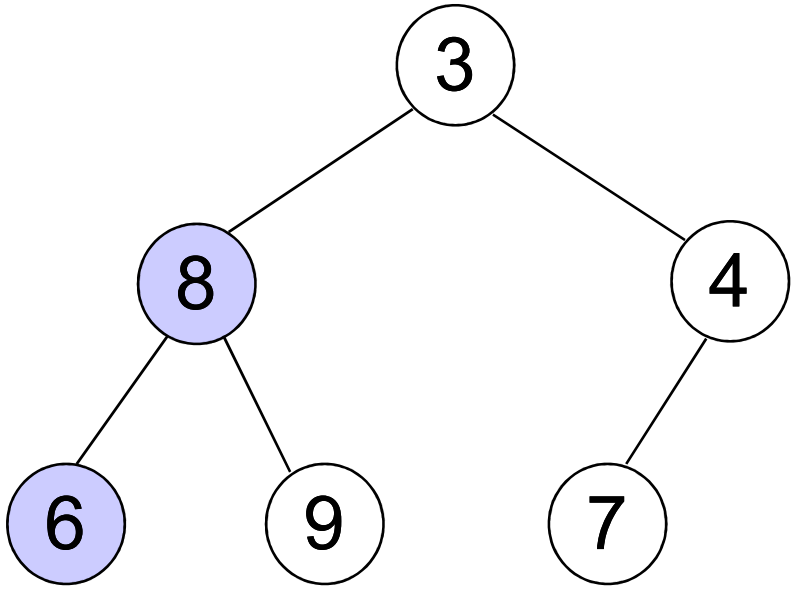
```
heapSort(A[ ], n)
{
        buildHeap(A, n);                ▷ Make Heap
        for i ← n downto 2 {
                A[1] ↔ A[i];            ▷ Swap
                heapify(A, 1, i-1);
        }
}
```

✓$O(n \log n)$ time in the worst case

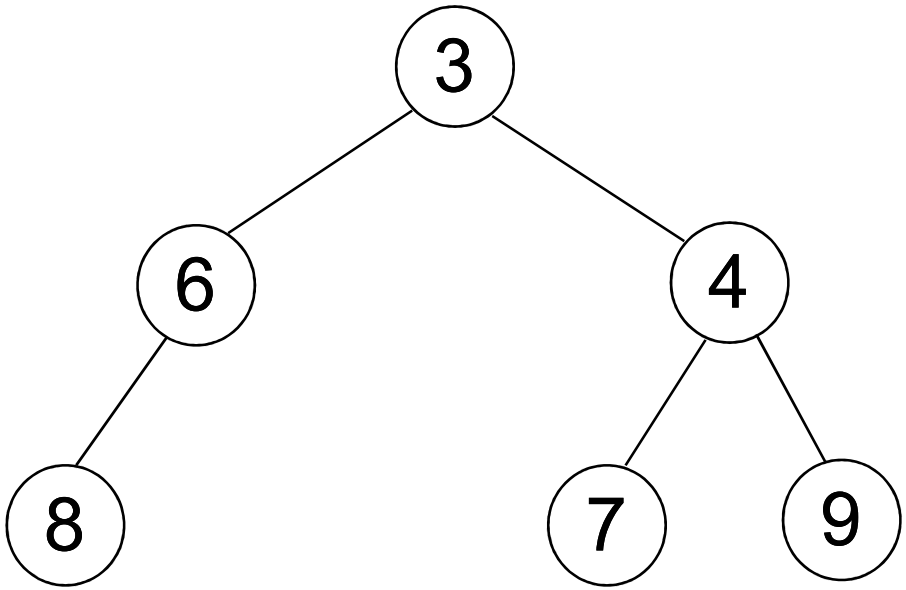
Heap



Heap

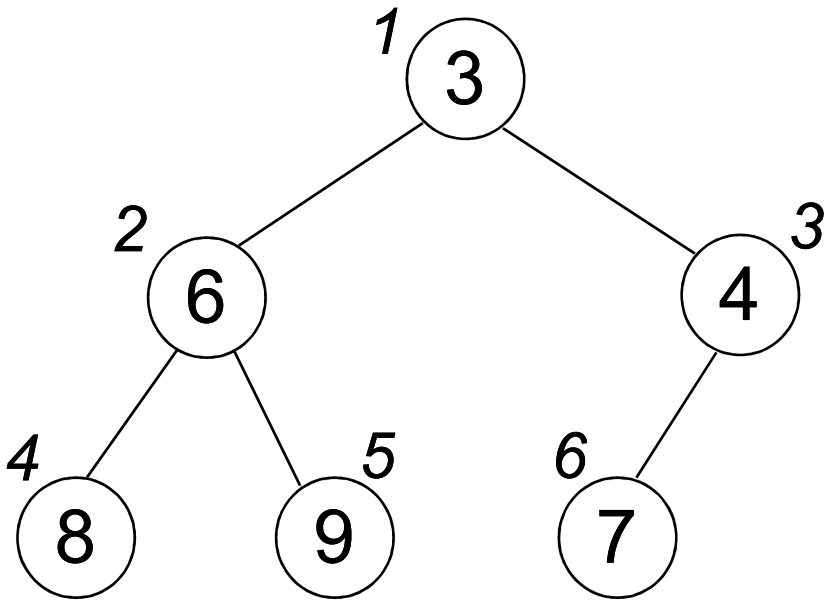


Not a Heap



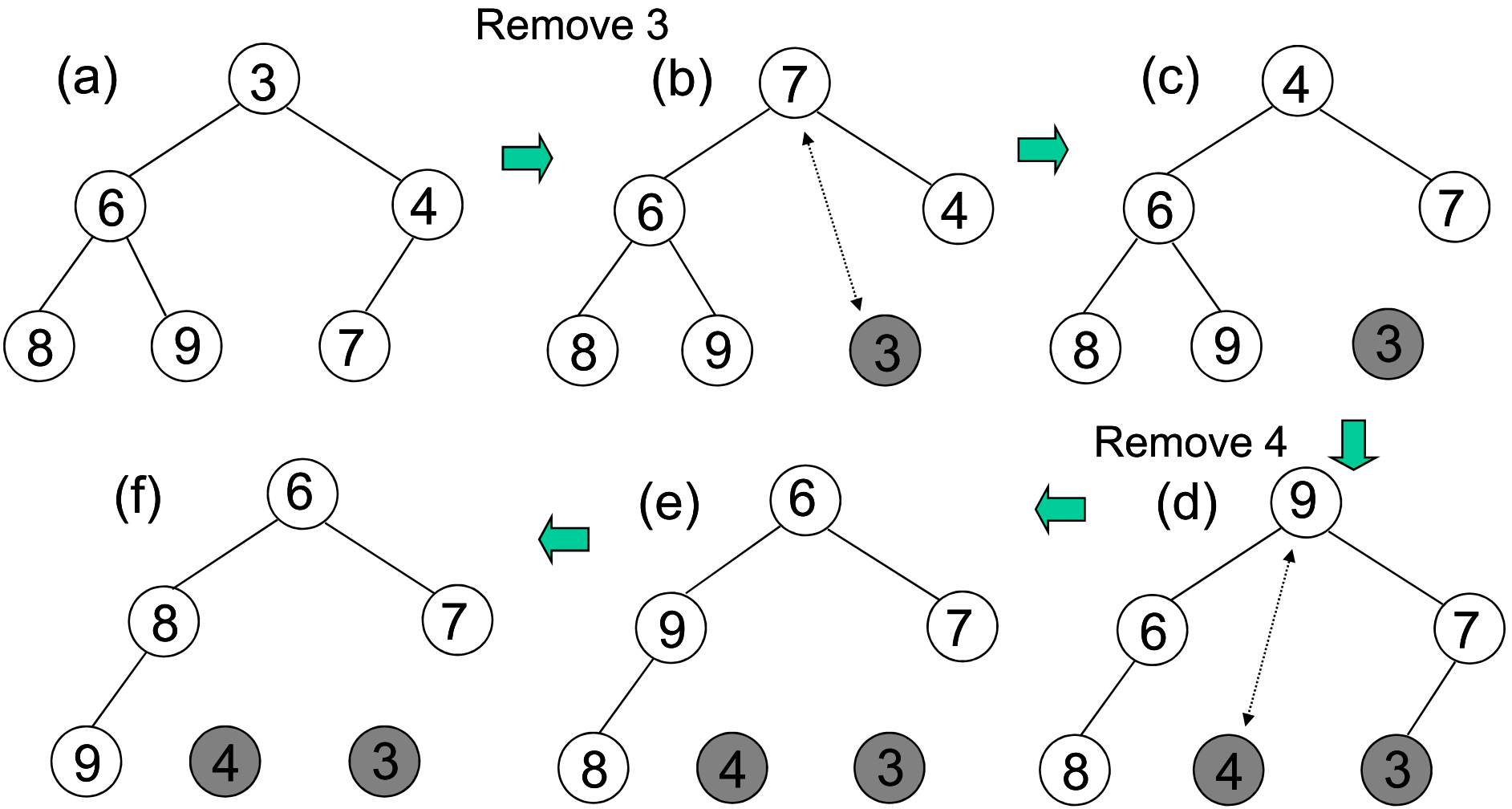
Not a Heap

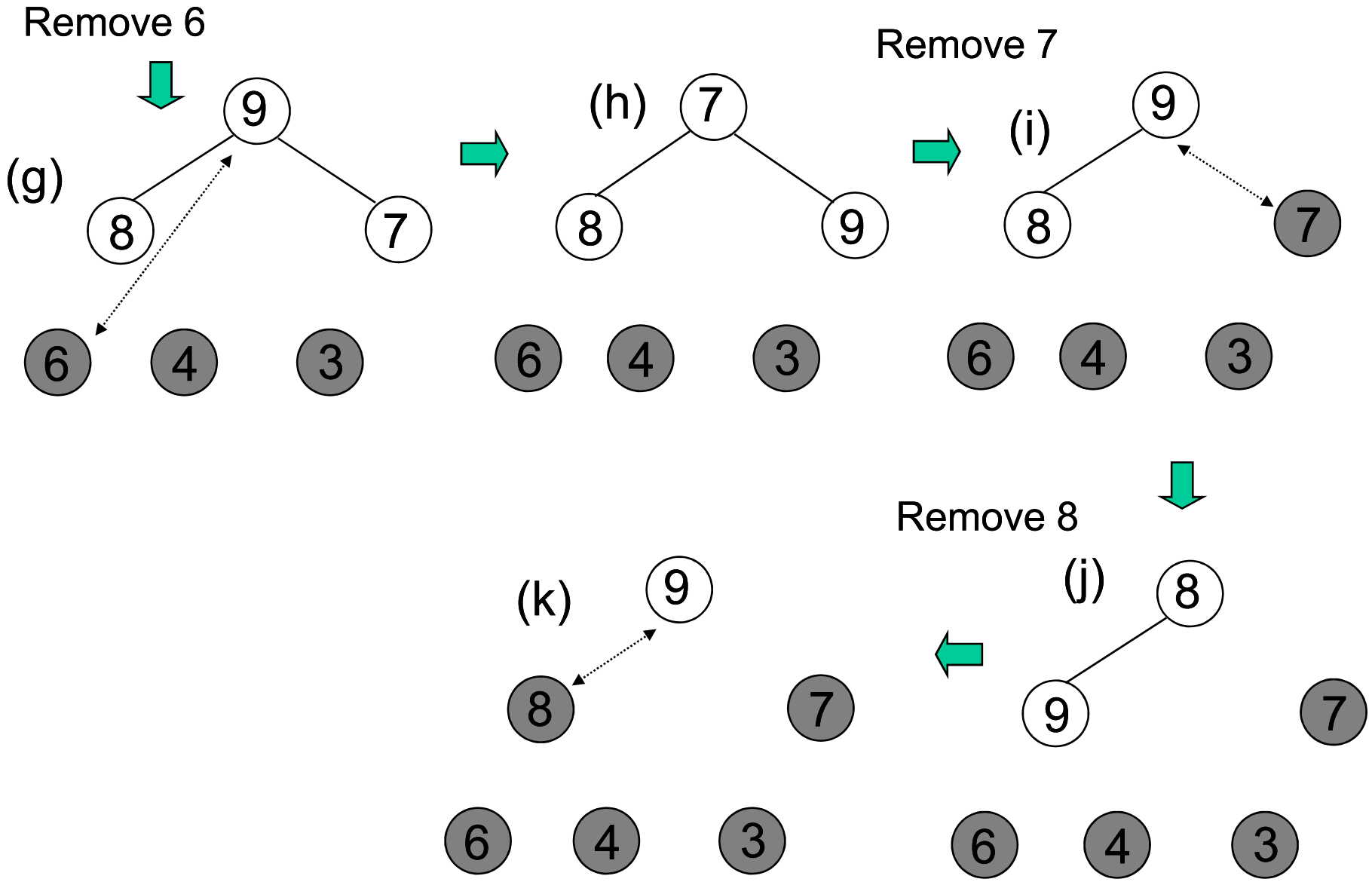
Heap can be represented in an array



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

Heapsort Example




Remove 6
(g)
9
8
7
6
4
3
(h)
7
8
9
6
4
3
Remove 7
(i)
9
8
7
6
4
3
Remove 8
(j)
8
9
7
6
4
3
(k)
9
8
7
6
4
3

$O(n)$ Sort

- Lower bound of sorting algorithms is $\Omega(n \log n)$
- If the elements satisfy a certain condition, $O(n)$
  - Counting Sort
    - Elements magnitude is between $-O(n) \sim O(n)$
  - Radix Sort
    - Elements has the number of digits less than k (k is constant)

Counting Sort

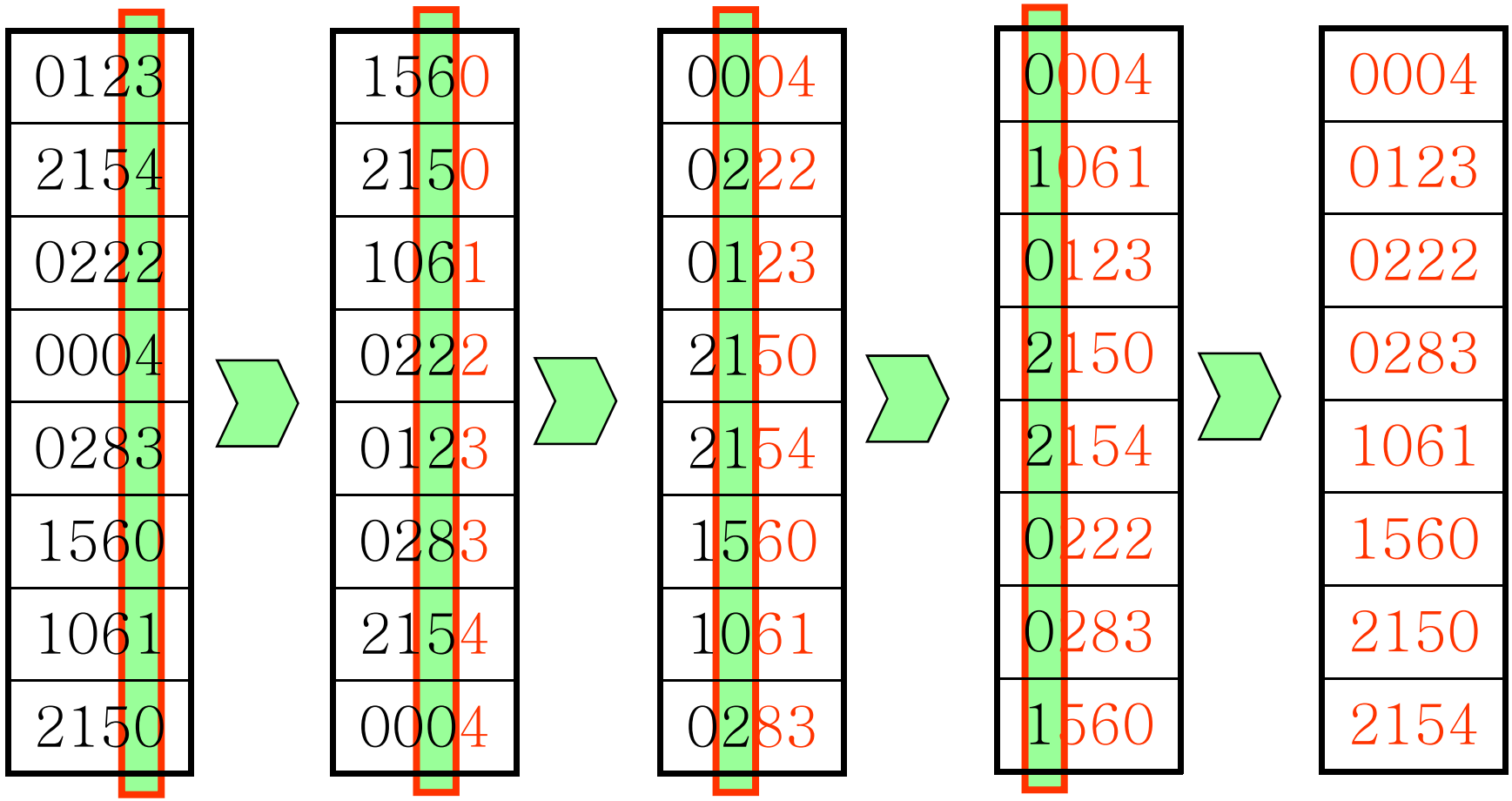
```
countingSort(A[ ], n)   ▷ simple version
{       ▷ A[ ]: Input Array, n: Input Size
        for i = 1 to k
                C[i] ← 0;
        for j = 1 to n
                C[A[j]]++;
        ▷ C[i] here : total number of elements with i value
        for i = 1 to k
                print C[i]  i's;        ▷ print i for C[i] times
}
```

Radix Sort

```
radixSort(A[ ], d)
{
        for j = d downto 1 {
                Do a stable sort on A[ ] by j^th digit;
        }
}
```

✓ Stable sort

— The order of items with a same value does not change after the sorting



✓ Running time: $O(n)$ ← d: a constant

Efficiency

	Worst Case	Average Case
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$n\log n$	$n\log n$
Quicksort	n^2	$n\log n$
Counting Sort	n	n
Radix Sort	n	n
Heapsort	$n\log n$	$n\log n$