Honors Data Structures

Lecture 18: Sorting I

3/30/22

Daniel Bauer

Sorting

- Input:

34	8	64	51	32	21

  - Array containing unordered `Comparables` (duplicates allowed).

- Output:

8	21	32	34	51	64

  - A sorted array containing the same items.

- Only comparisons between pairs of items allowed (**comparison based sorting**).

Sorting Applications

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
- Efficient search (binary search on sorted data).

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
- Efficient search (binary search on sorted data).
- Finding duplicates.

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
- Efficient search (binary search on sorted data).
- Finding duplicates.
- Greedy algorithms (explore k highest scoring paths first).

Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
- Efficient search (binary search on sorted data).
- Finding duplicates.
- Greedy algorithms (explore k highest scoring paths first).
- …

Sorting Overview

- We will discuss different sorting algorithms and compare their running time, required space, and stability.
  - Heap Sort
  - Selection Sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket Sort / Radix Sort (not comparison based)

Heap Sort

- First convert an unordered array into a heap in O(N) time.
- Then perform N `deleteMin` operations to retrieve the elements in sorted order.
  - each `deleteMin` is O(log N)

Heap Sort

- First convert an unordered array into a heap in O(N) time.

- Then perform N `deleteMin` operations to retrieve the elements in sorted order.

  - each `deleteMin` is O(log N)

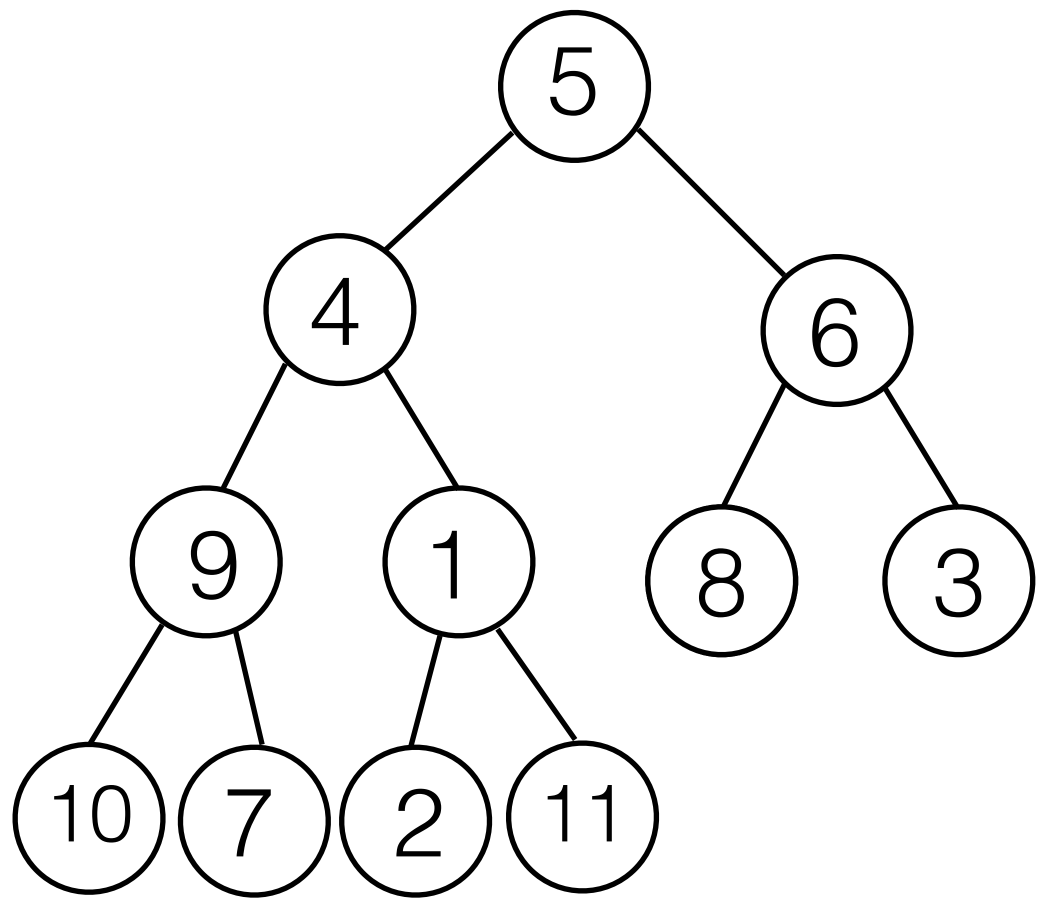
- Problem: This algorithm requires a second array to store the output: O(N) space!

- Idea: re-use the freed space after each `deleteMin`.

Heap Sort Example

5	4	6	9	1	8	3	10	7	2	11

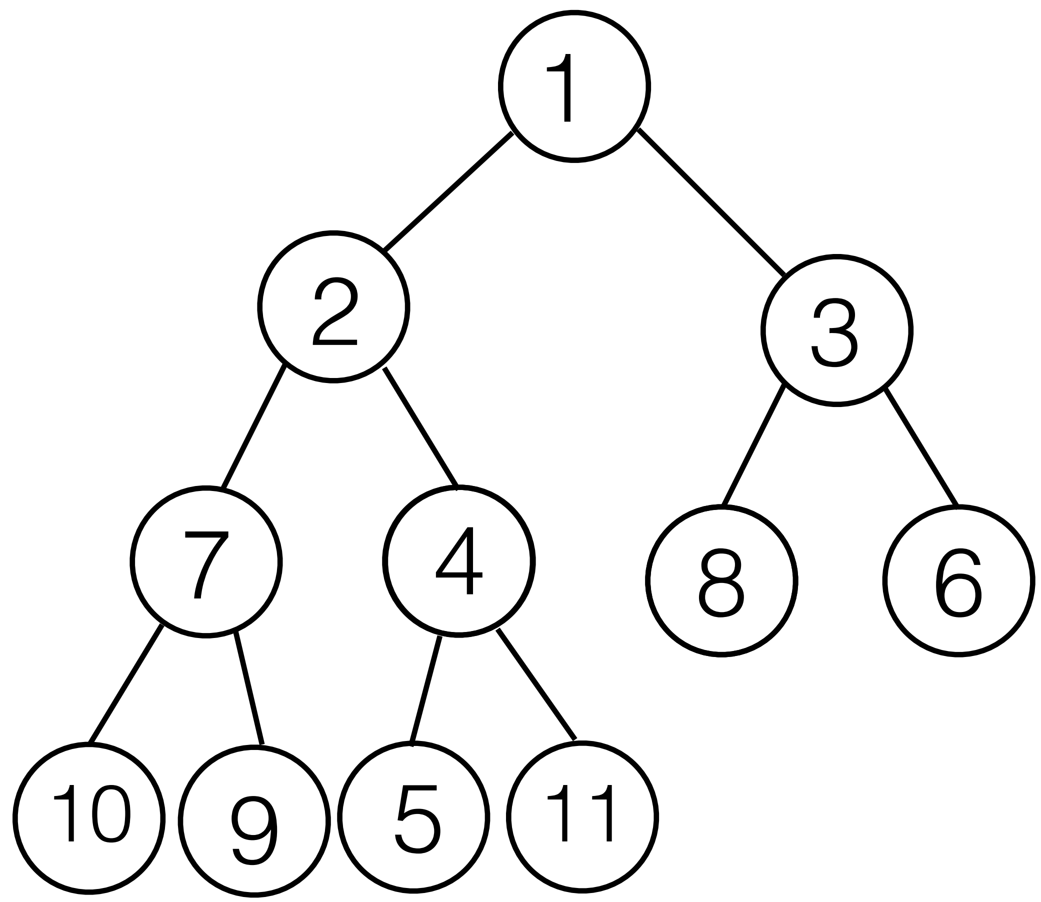
Heap Sort Example



	5	4	6	9	1	8	3	10	7	2	11

Heap Sort Example

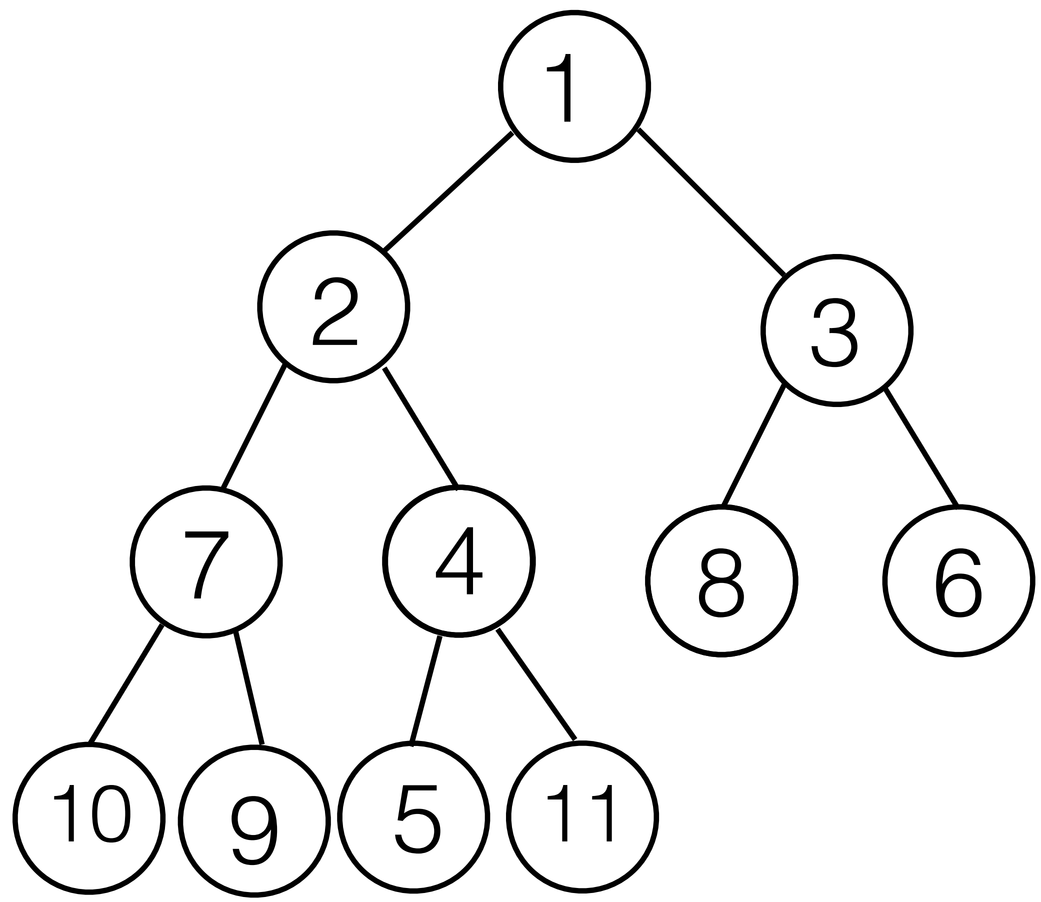
Build heap in O(N) time



1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

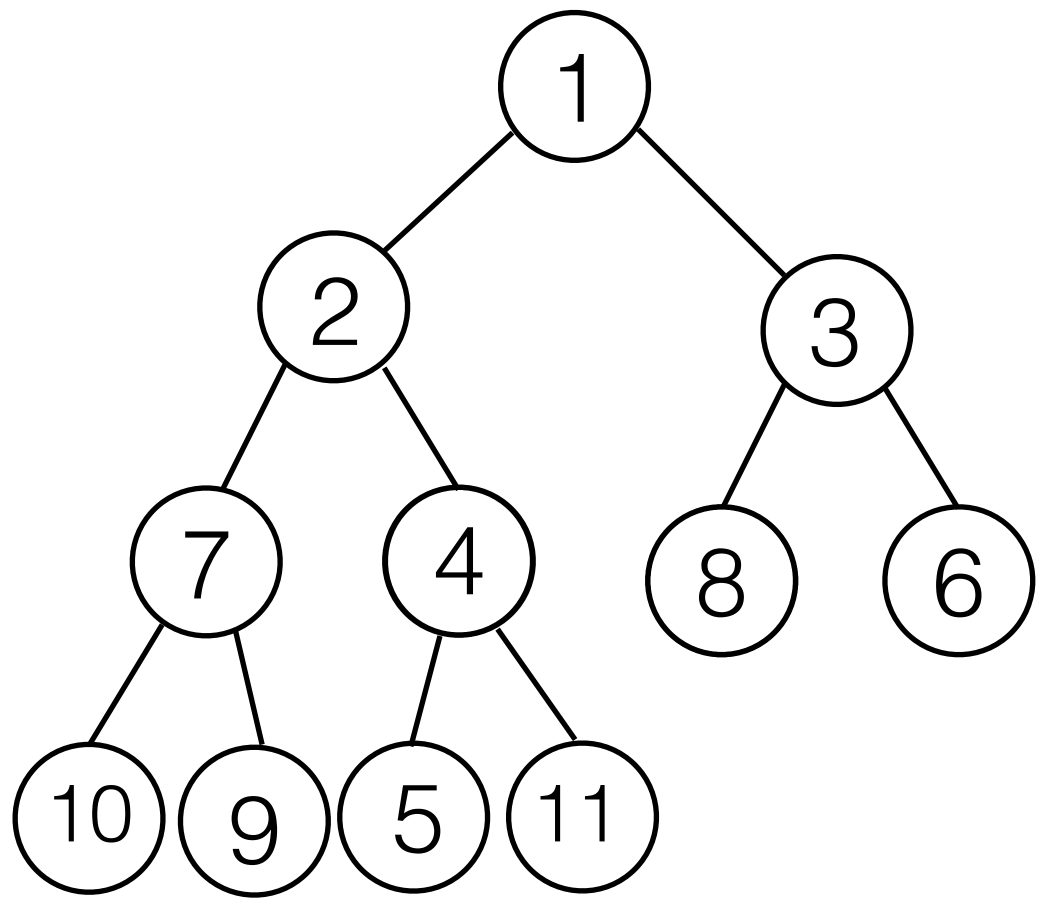
Build heap in O(N) time



	1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

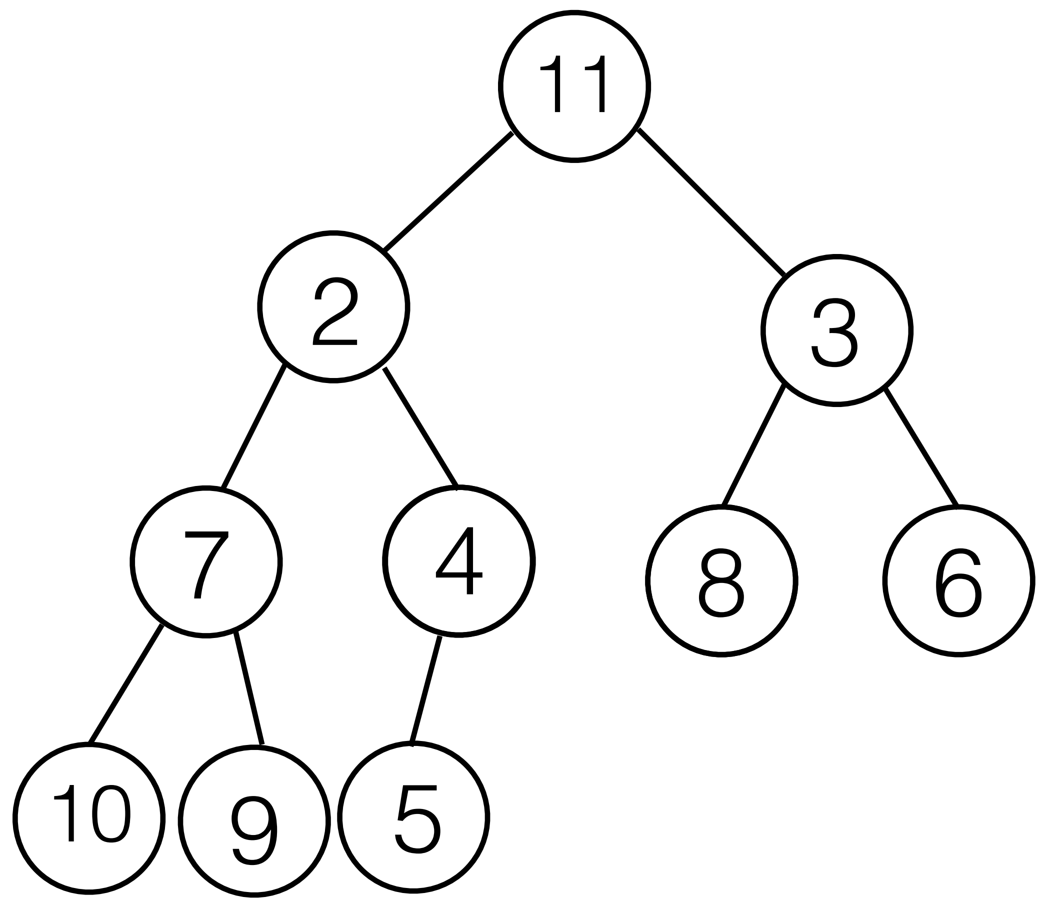
deleteMin, write min element into empty cell



	1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

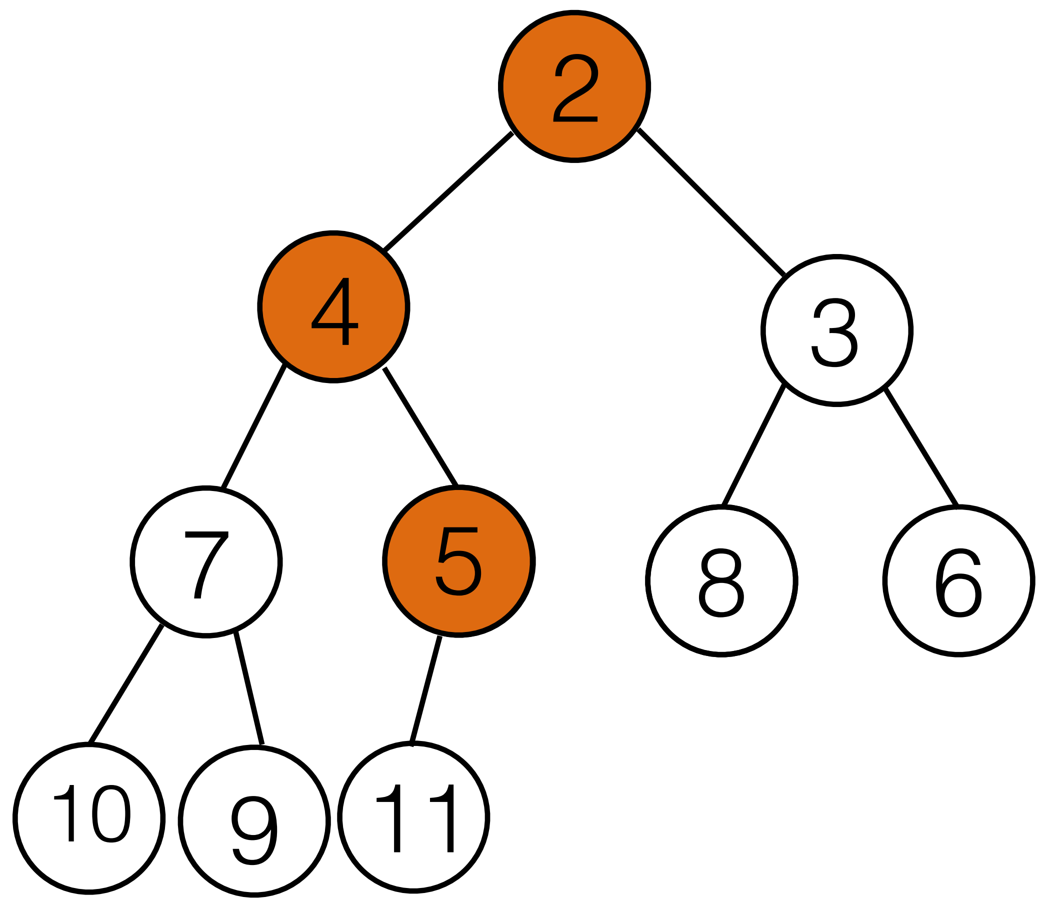
deleteMin, write min element into empty cell



	11	2	3	7	4	8	6	10	9	5	1

Heap Sort Example

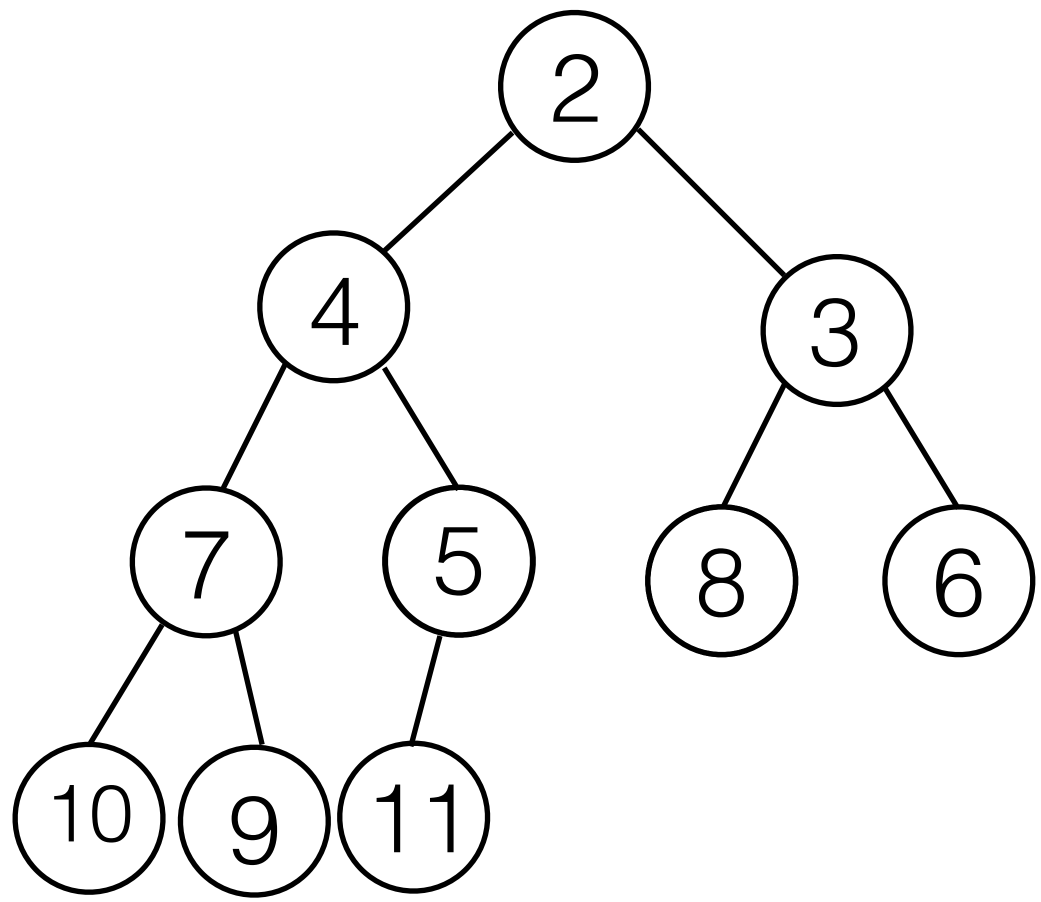
Percolate down



	2	4	3	7	5	8	6	10	9	11	1

Heap Sort Example

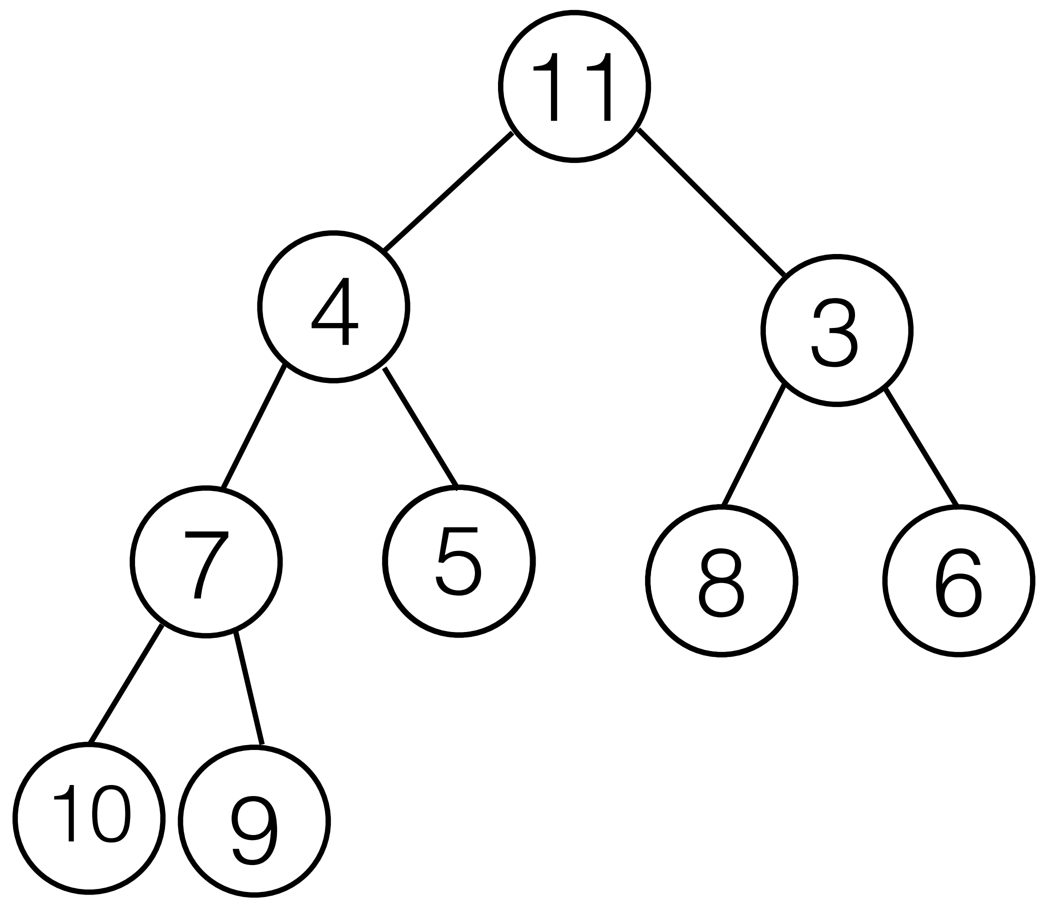
deleteMin, write min element into empty cell



	2	4	3	7	5	8	6	10	9	11	1

Heap Sort Example

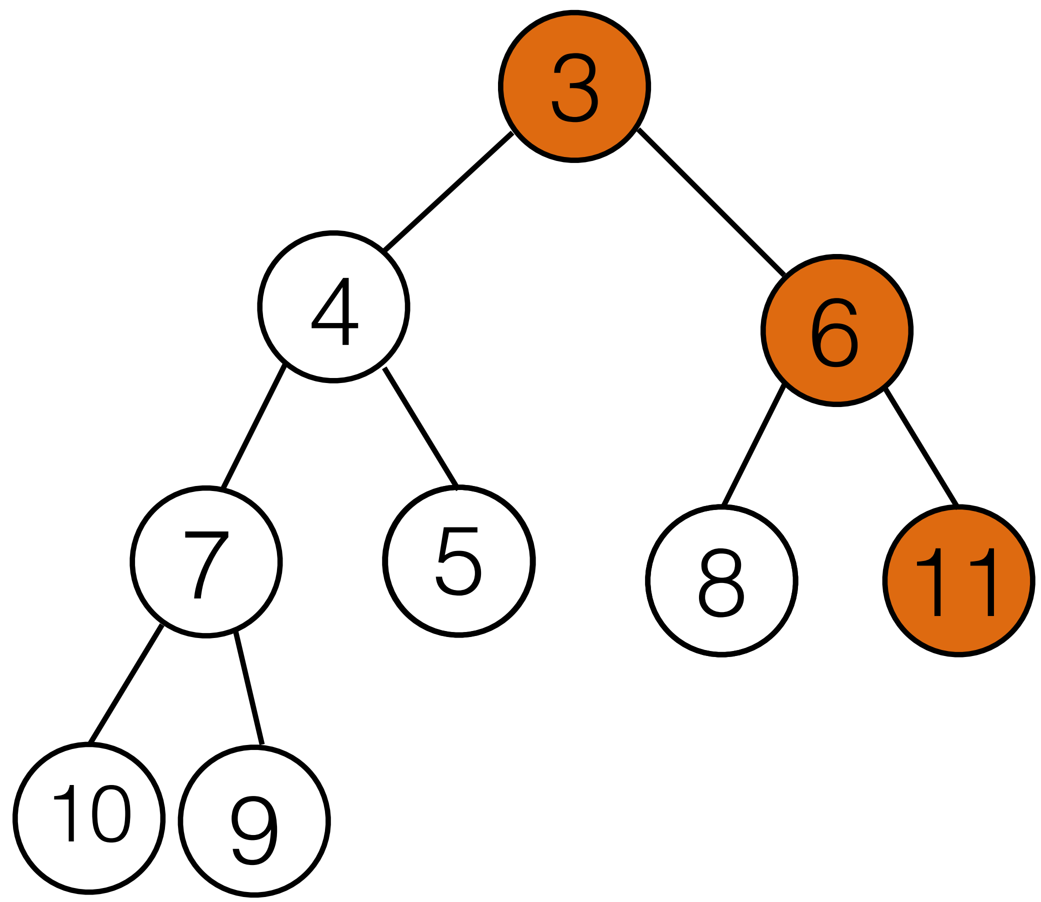
deleteMin, write min element into empty cell



	11	4	3	7	5	8	6	10	9	2	1

Heap Sort Example

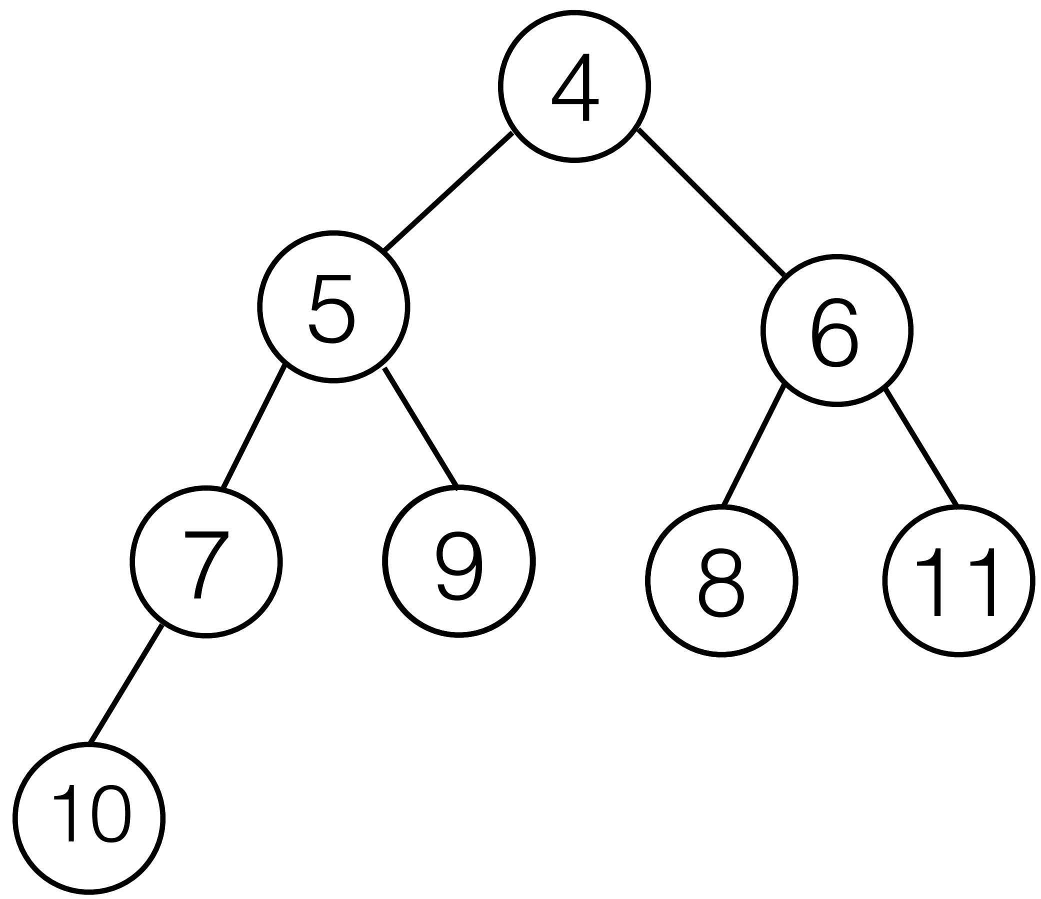
Percolate down



	3	4	6	7	5	8	11	10	9	2	1

Heap Sort Example

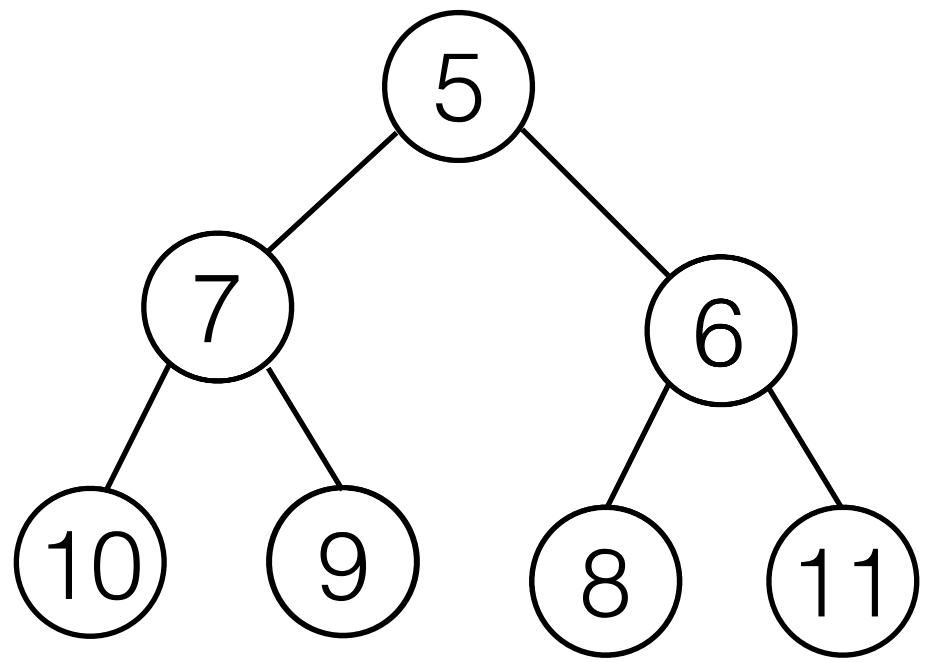
deleteMin, write min element into empty cell



	4	5	6	7	9	8	11	10	3	2	1

Heap Sort Example

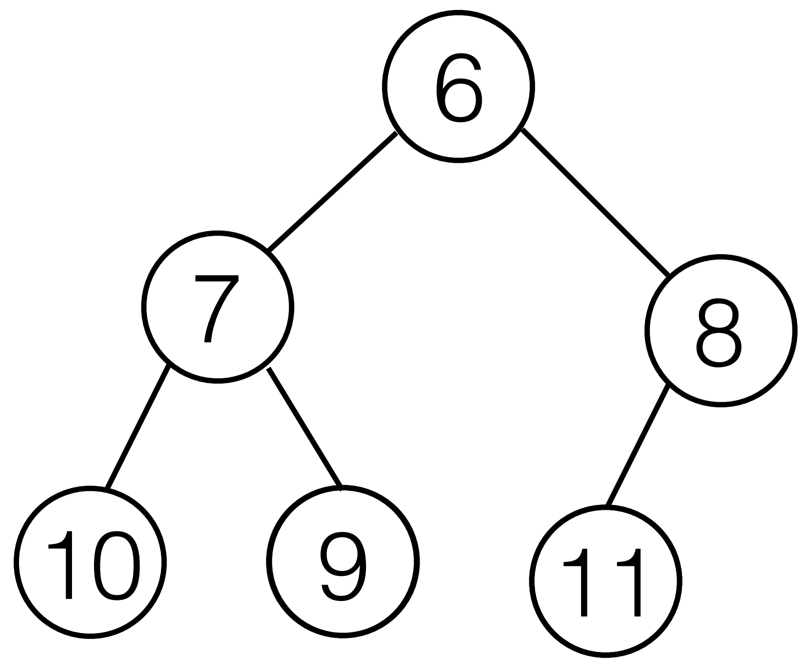
deleteMin, write min element into empty cell



	5	7	6	10	9	8	11	4	3	2	1

Heap Sort Example

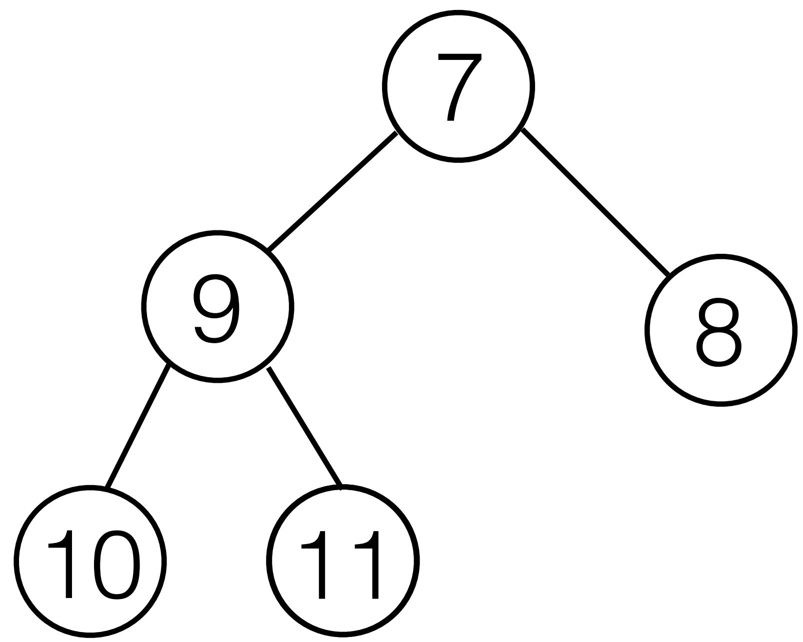
deleteMin, write min element into empty cell



	6	7	8	10	9	11	5	4	3	2	1

Heap Sort Example

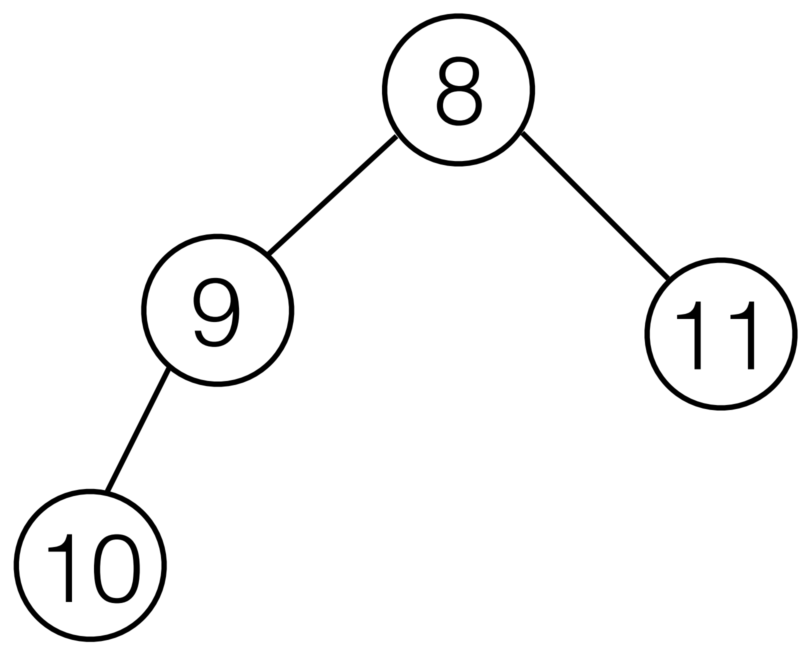
deleteMin, write min element into empty cell



	7	9	8	10	11	6	5	4	3	2	1

Heap Sort Example

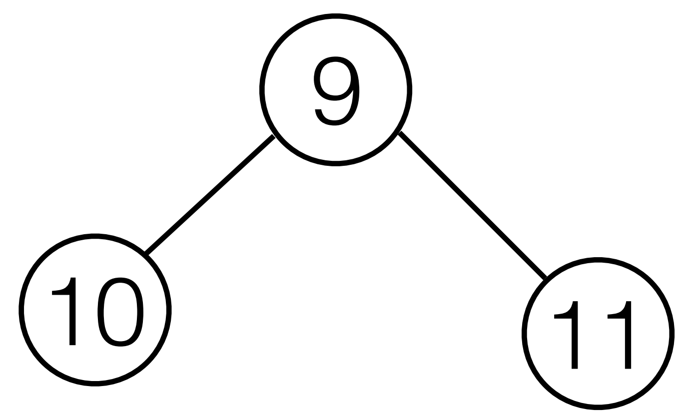
deleteMin, write min element into empty cell



	8	9	11	10	7	6	5	4	3	2	1

Heap Sort Example

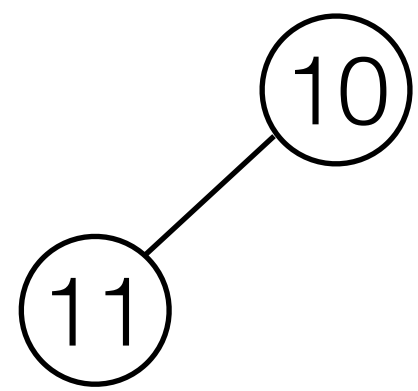
deleteMin, write min element into empty cell



	9	10	11	8	7	6	5	4	3	2	1

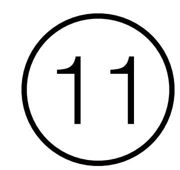
Heap Sort Example

deleteMin, write min element into empty cell



	10	11	9	8	7	6	5	4	3	2	1

Heap Sort Example



- Can use a max-heap if we want the output in increasing order.

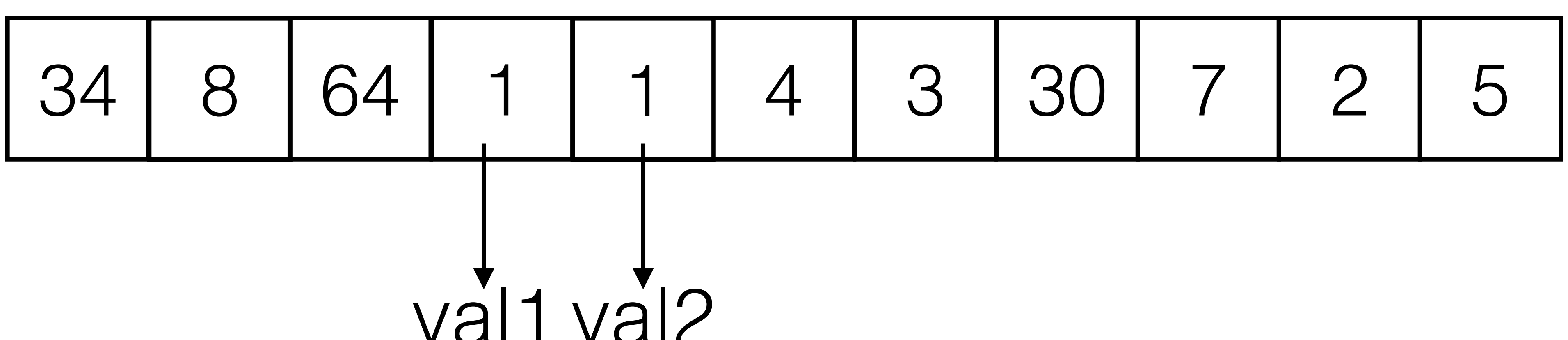
	11	10	9	8	7	6	5	4	3	2	1

Analyzing Sorting Algorithms

- For each sorting algorithm we will compare:
  - Runtime (best / worst case)
  - Space requirements:
    - Some sorting algorithms / implementations require extra space in addition to the input.
  - Stability

Sorting Stability

- Assume we can distinguish between duplicate items in the input (example: store pairs but compare only on first element).
- A sorting algorithm is stable if the *relative order* of duplicate items in the input is preserved.



Sorting Stability

- Assume we can distinguish between duplicate items in the input (example: store pairs but compare only on first element).
- A sorting algorithm is stable if the *relative order* of duplicate items in the input is preserved.

1	1	2	3	4	5	7	8	30	34	64

val1 val2

Sorting Stability

- Assume we can distinguish between duplicate items in the input (example: store pairs but compare only on first element).
- A sorting algorithm is stable if the *relative order* of duplicate items in the input is preserved.

not stable if this can happen

1	1	2	3	4	5	7	8	30	34	64

val2 val1

Sorting Stability

- Assume we can distinguish between duplicate items in the input (example: store pairs but compare only on first element).

- A sorting algorithm is stable if the *relative order* of duplicate items in the input is preserved.

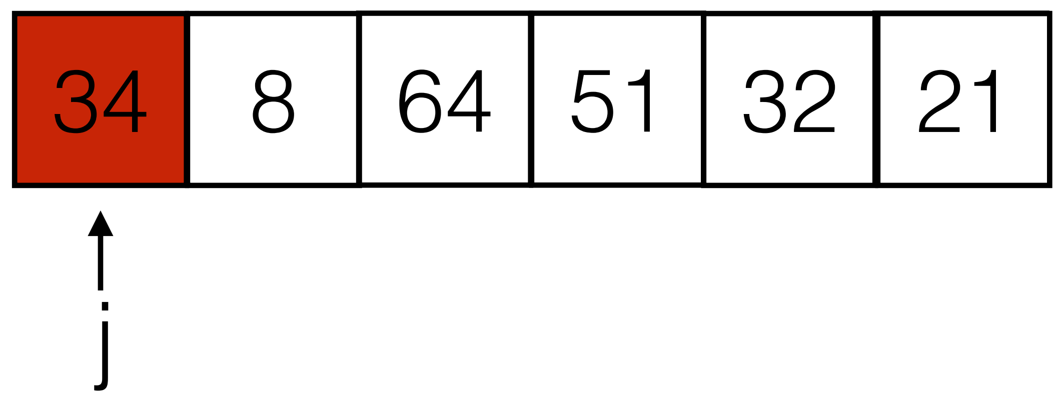
not stable if this can happen

1	1	2	3	4	5	7	8	30	34	64

val2 val1

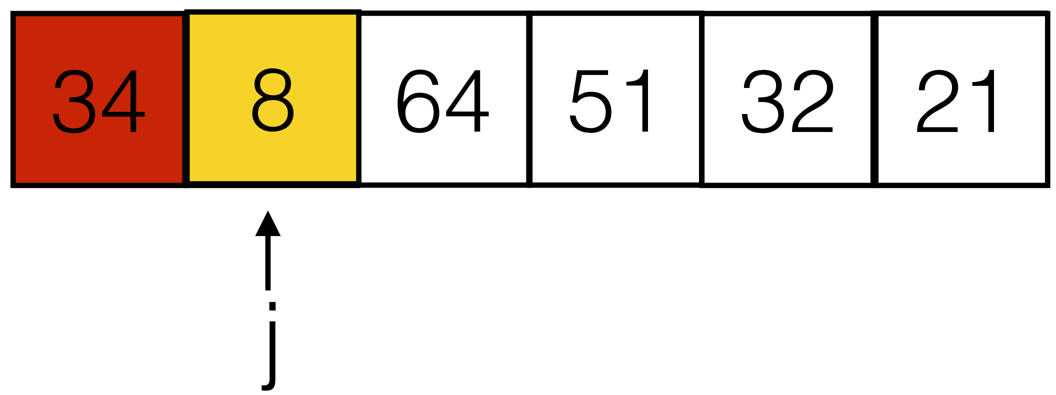
Heap Sort is not stable.

Selection Sort



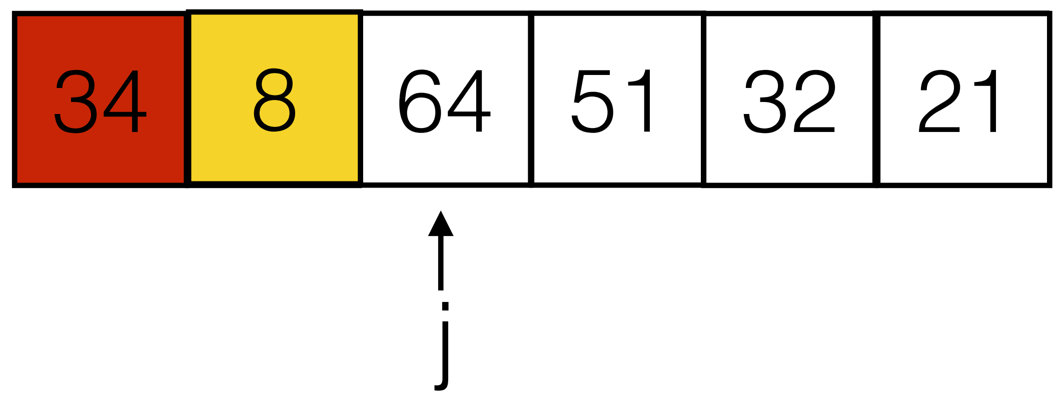
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



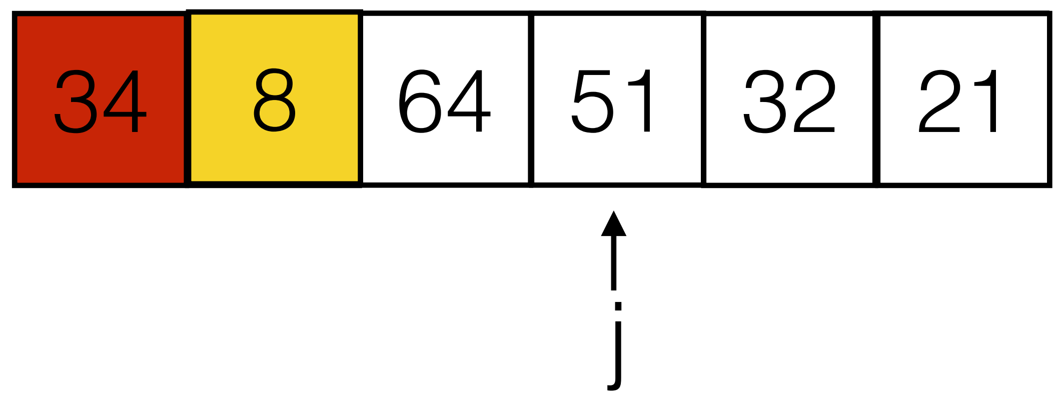
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



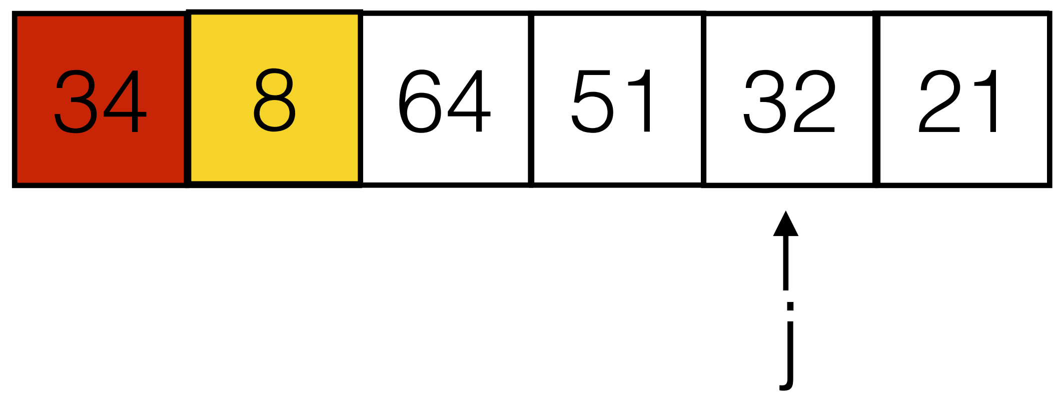
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



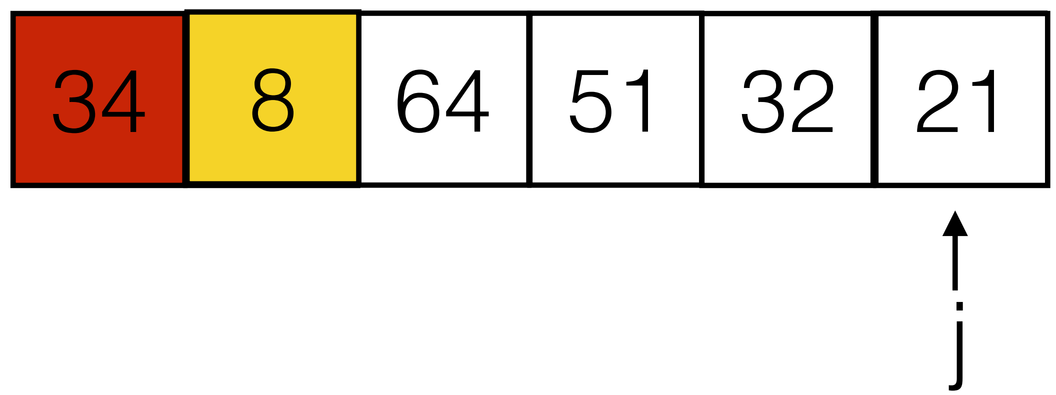
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



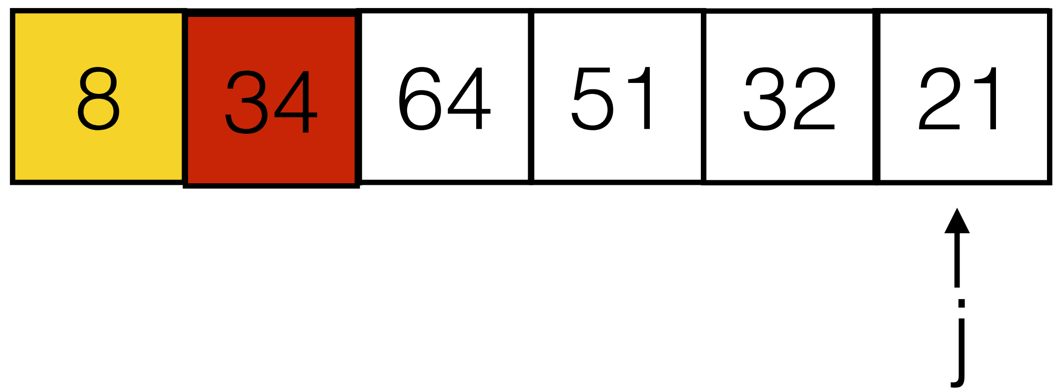
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



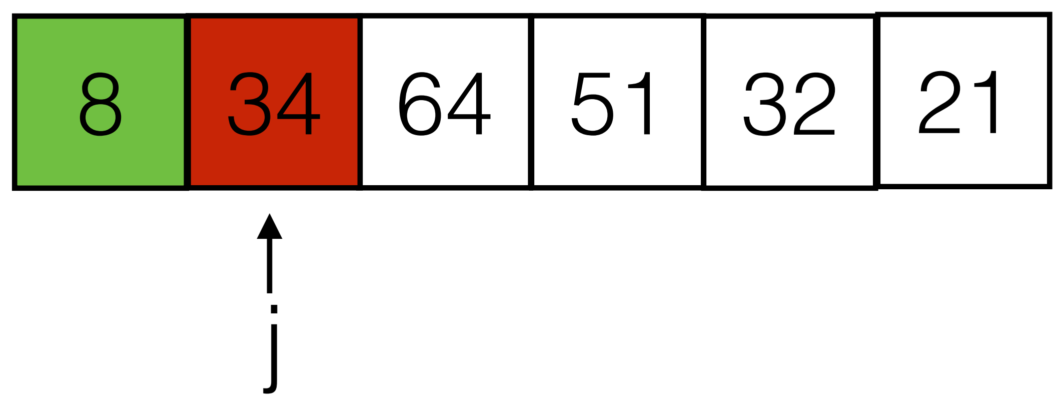
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



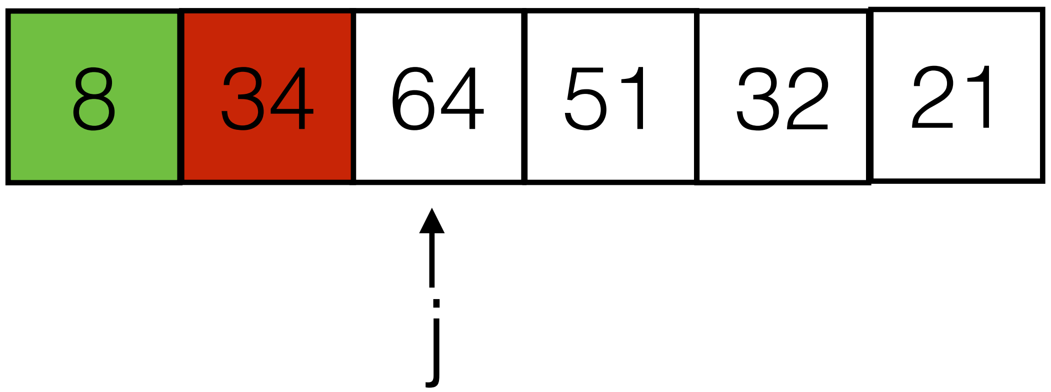
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



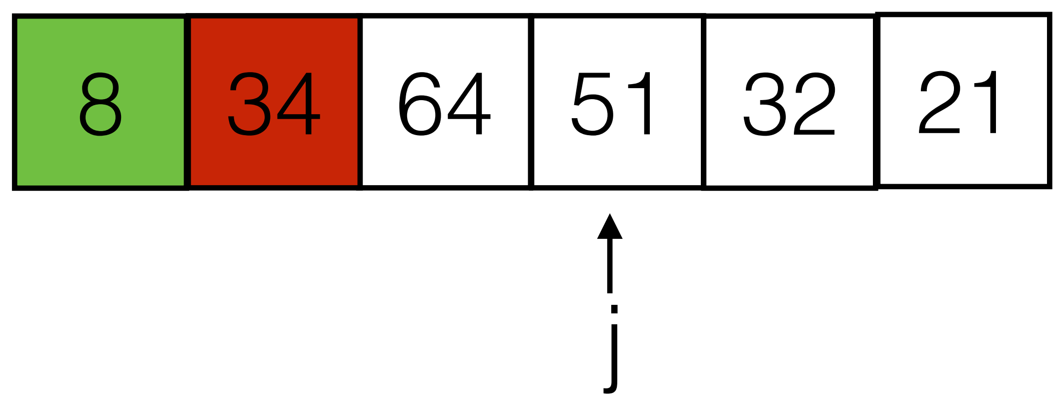
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



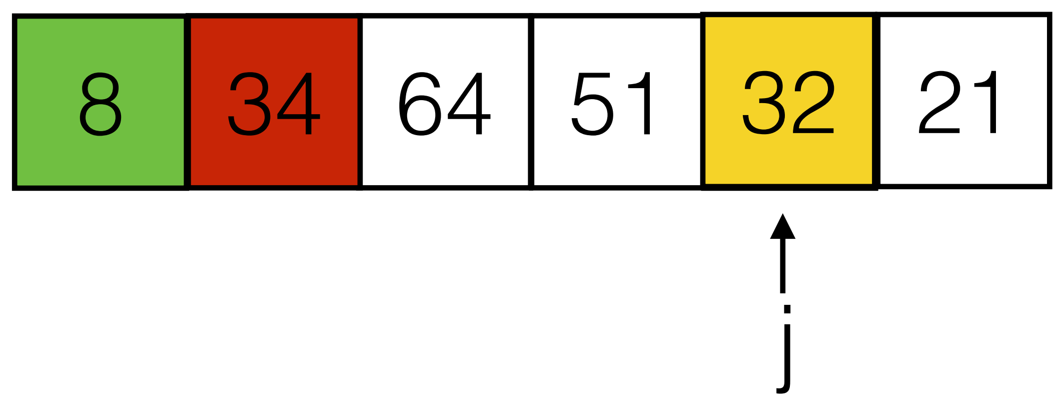
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



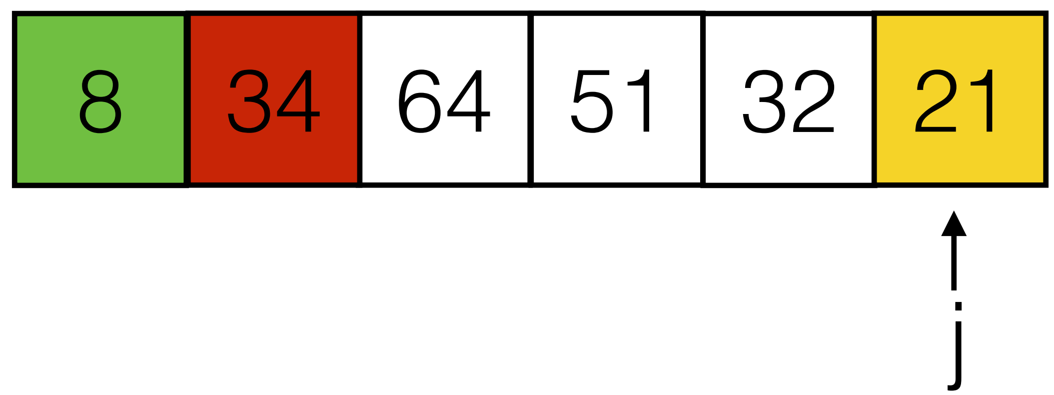
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



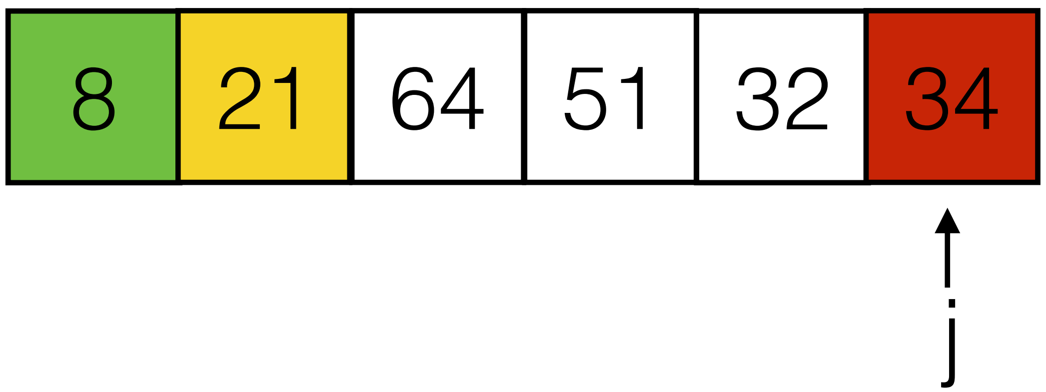
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



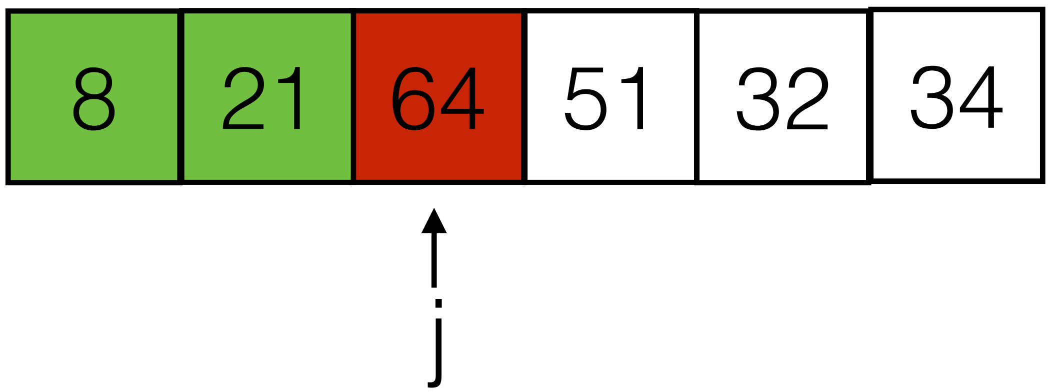
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



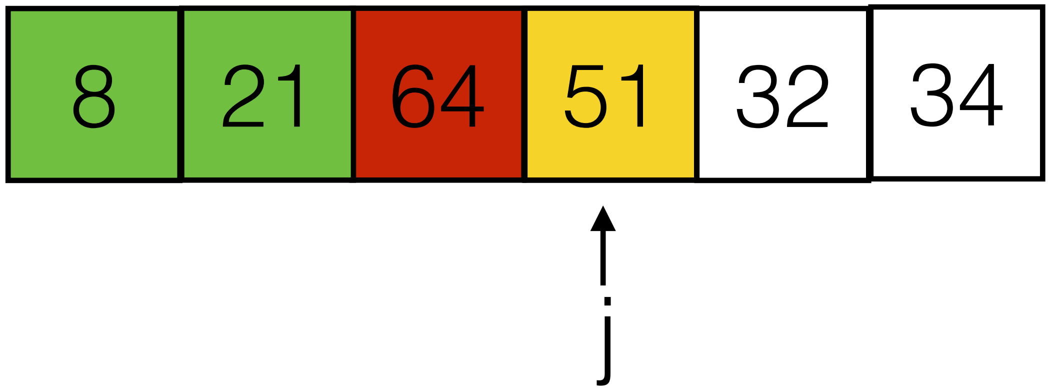
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



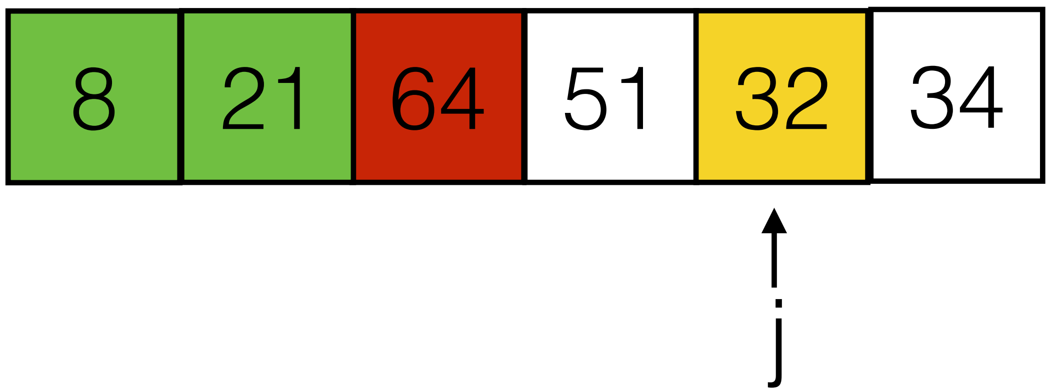
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



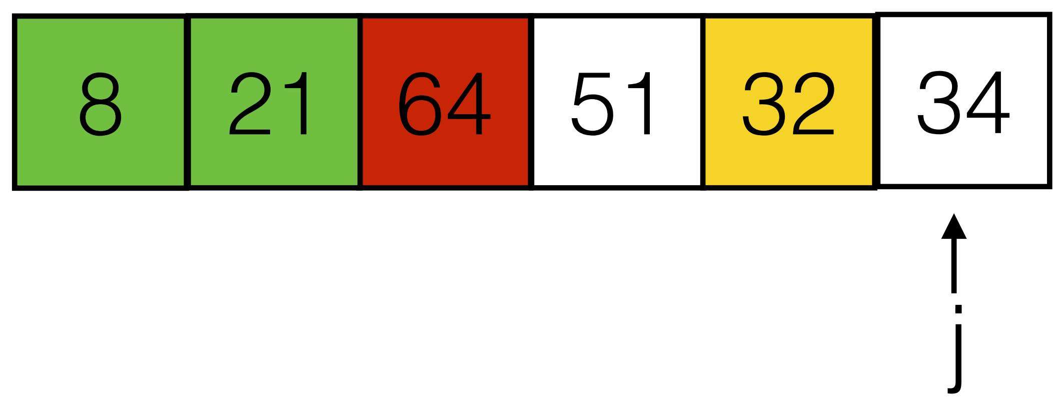
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



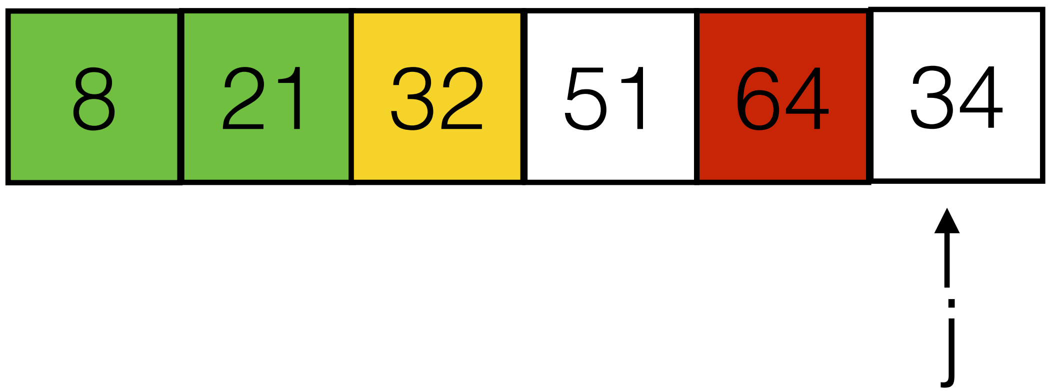
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



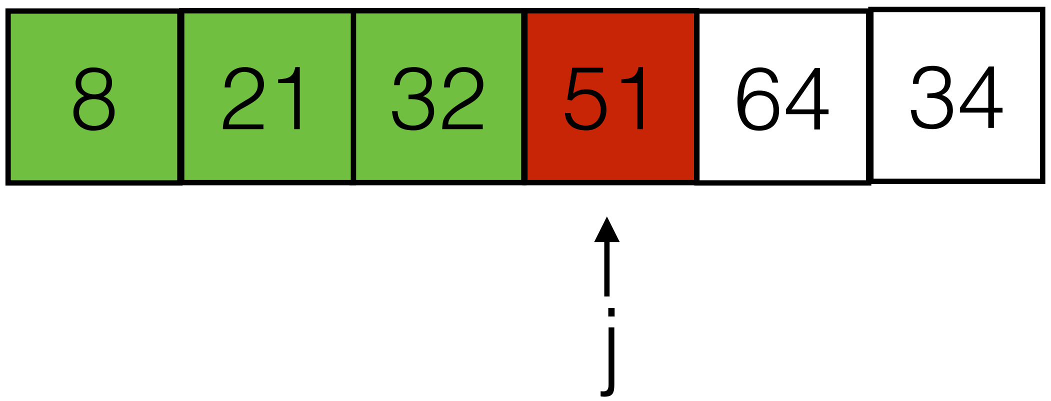
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



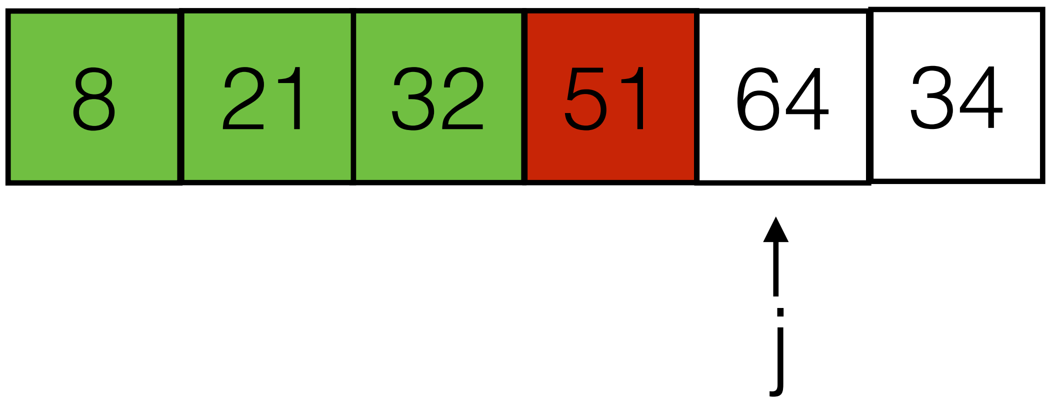
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



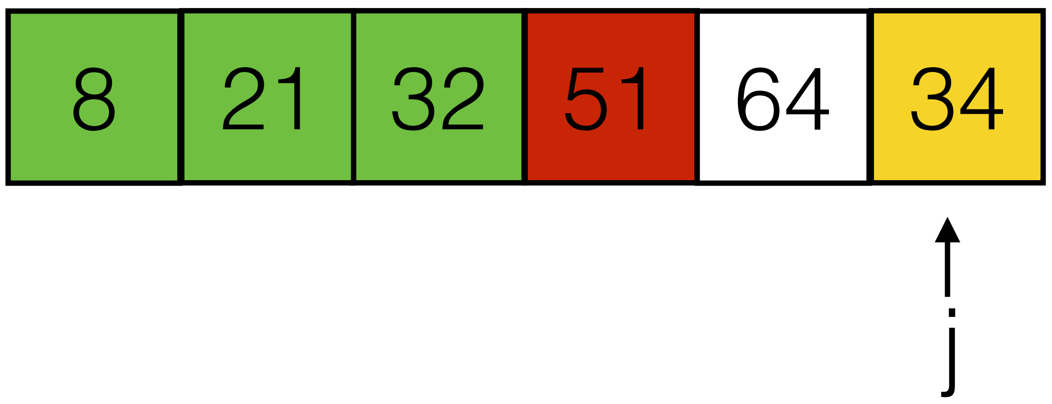
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



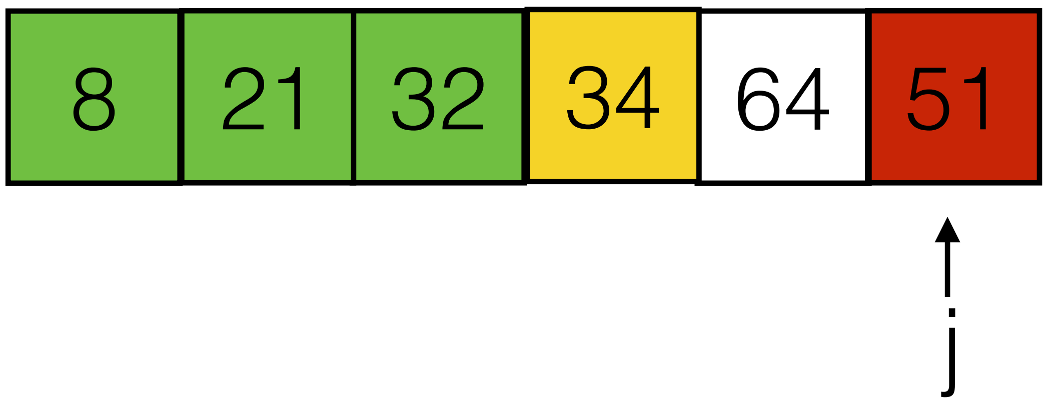
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



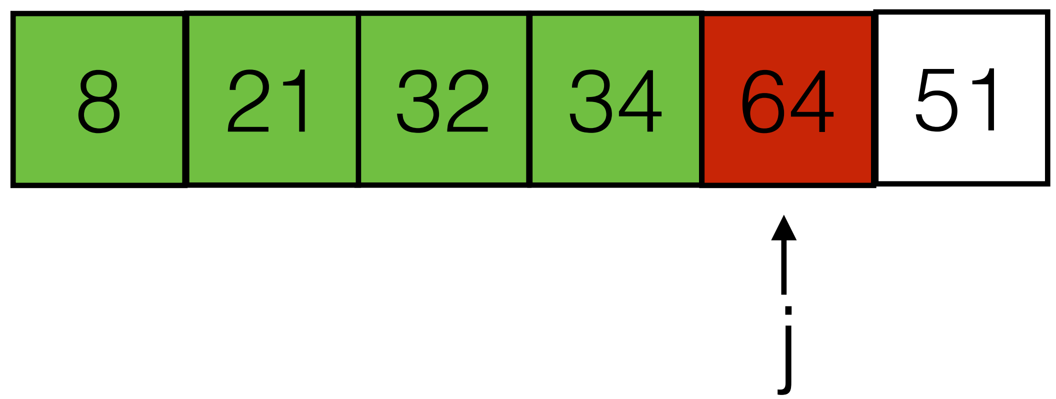
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



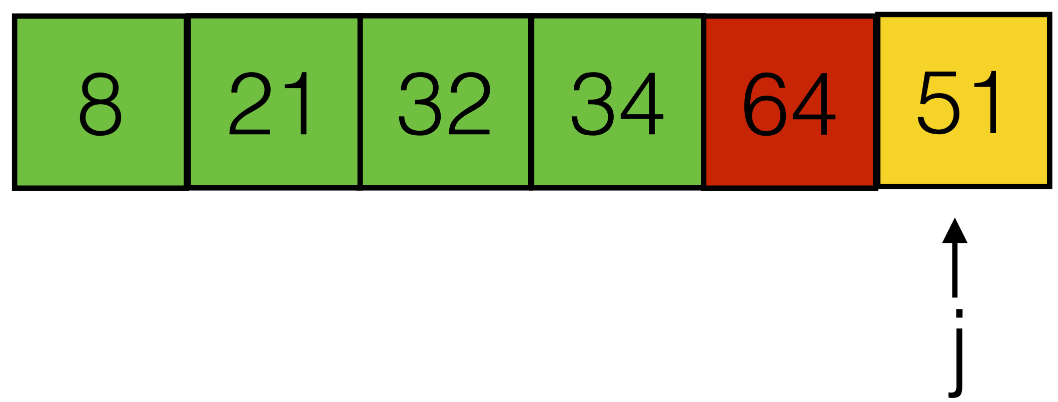
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



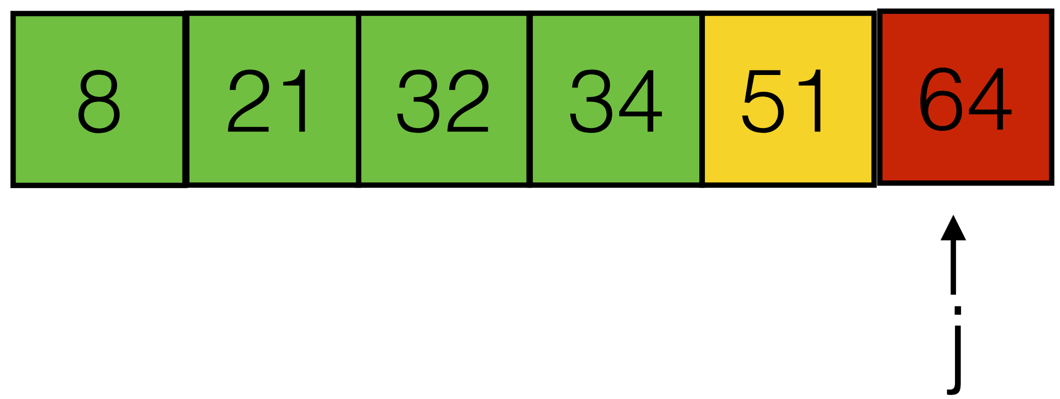
- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort



- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

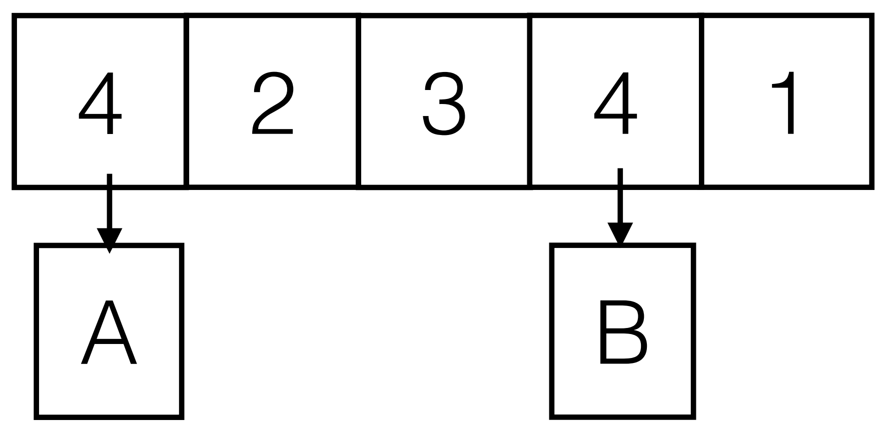
Selection Sort



- Perform N passes through the array, p=0…N-1
  - Assume array[0..p-1] is already sorted.
  - Find the minimum element in the unsorted partition.
    - Swap the element at position p with the minimum.

Selection Sort is **Not** Stable

- Try sorting the following example using Selection Sort



Insertion Sort

34	8	64	51	32	21

p=1

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	64	51	32	21

p=1

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	64	51	32	21

p=2

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	64	51	32	21

p=3

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	51	64	32	21

p=3

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	51	64	32	21

p=4

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	34	51	32	64	21

p=4

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

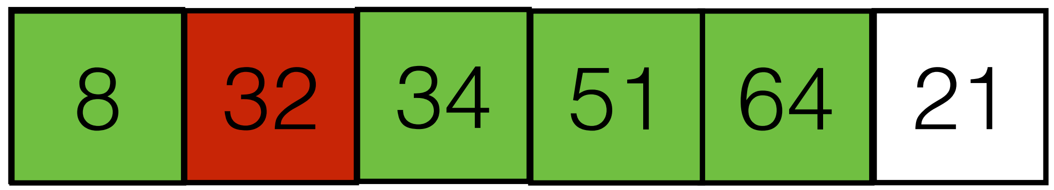
Insertion Sort

8	34	32	51	64	21

p=4

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort



p=4

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort

8	32	34	51	64	21

p=5

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

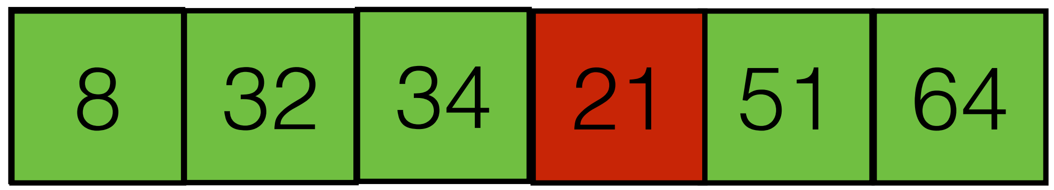
Insertion Sort

8	32	34	51	21	64

p=5

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort



p=5

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

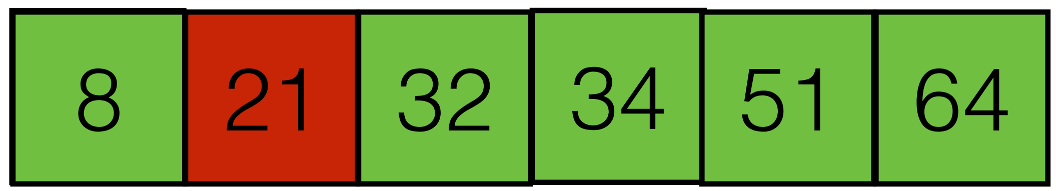
Insertion Sort

8	32	21	34	51	64

p=5

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion Sort



p=5

- Perform N passes through the array. p=1…N-1
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x with its left neighbor until x is in the correct position.

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:			
• Time			
Best case			
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)		
• Time			
Best case			
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	
• Time			
Best case			
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case			
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)		
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	$O(1)$	$O(1)$	$O(1)$
• Time			
Best case	$O(N)$	$O(N^2)$	
Worst case			
• Stable?			

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)	$O(N^2)$	O(N log N)
Worst case			

- Stable?

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	$O(1)$	$O(1)$	$O(1)$
• Time			
Best case	$O(N)$	$O(N^2)$	$O(N \log N)$
Worst case	$O(N^2)$		

- Stable?

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)	O(N^2)	O(N log N)
Worst case	O(N^2)	O(N^2)	

- Stable?

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)	$O(N^2)$	O(N log N)
Worst case	$O(N^2)$	$O(N^2)$	O(N log N)

• Stable?

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)	O(N^2)	O(N log N)
Worst case	O(N^2)	O(N^2)	O(N log N)
• Stable?	Yes		

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	O(1)	O(1)	O(1)
• Time			
Best case	O(N)	O(N^2)	O(N log N)
Worst case	O(N^2)	O(N^2)	O(N log N)
• Stable?	Yes	**NO**	

Insertion and Selection Sort

	Insertion Sort	Selection Sort	HeapSort
• Space:	$O(1)$	$O(1)$	$O(1)$
• Time			
Best case	$O(N)$	$O(N^2)$	$O(N \log N)$
Worst case	$O(N^2)$	$O(N^2)$	$O(N \log N)$
• Stable?	Yes	**NO**	**NO**

Merge Sort

- A classic *divide-and-conquer* algorithm.
- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2	51	32	21	1

Merge Sort

- A classic *divide-and-conquer* algorithm.
- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2

51	32	21	1

Merge Sort

- A classic *divide-and-conquer* algorithm.
- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

2	8	34	64

1	21	32	51

Merge Sort

- A classic *divide-and-conquer* algorithm.
- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

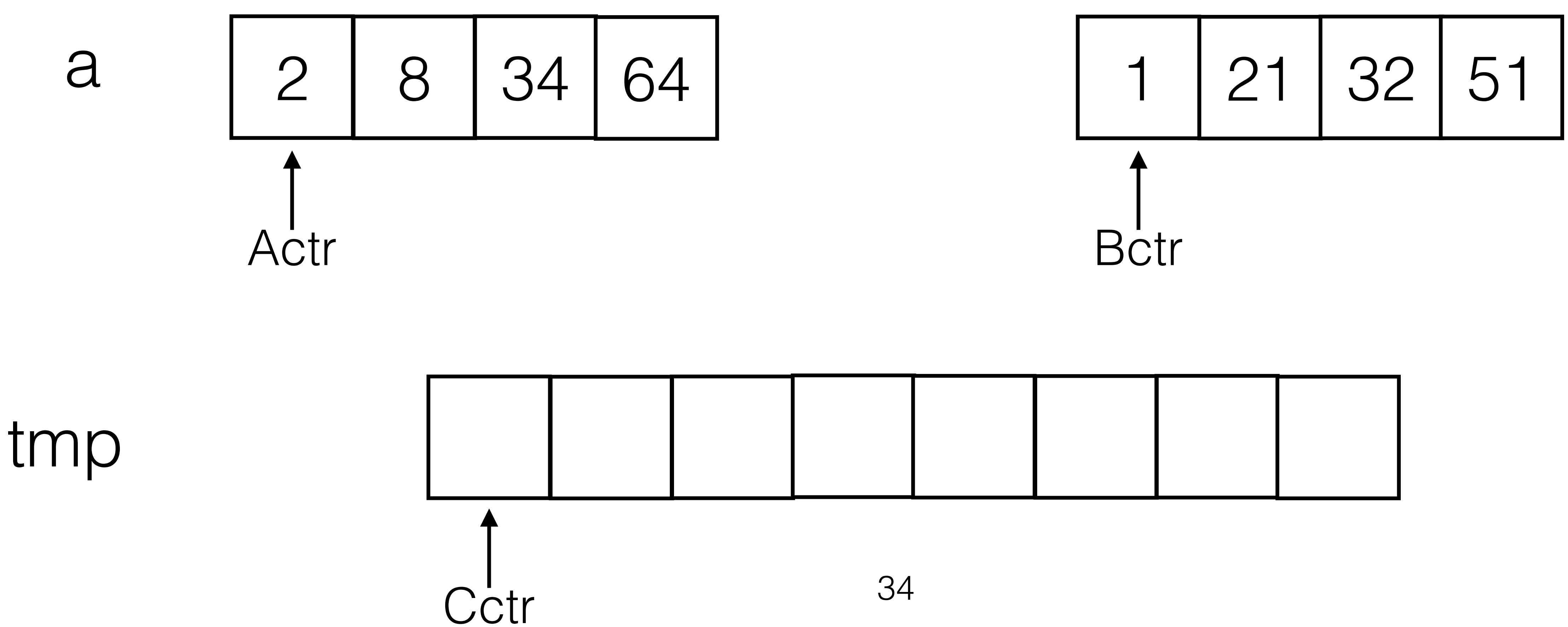
2	8	34	64

1	21	32	51

1	2	8	21	32	34	51	64

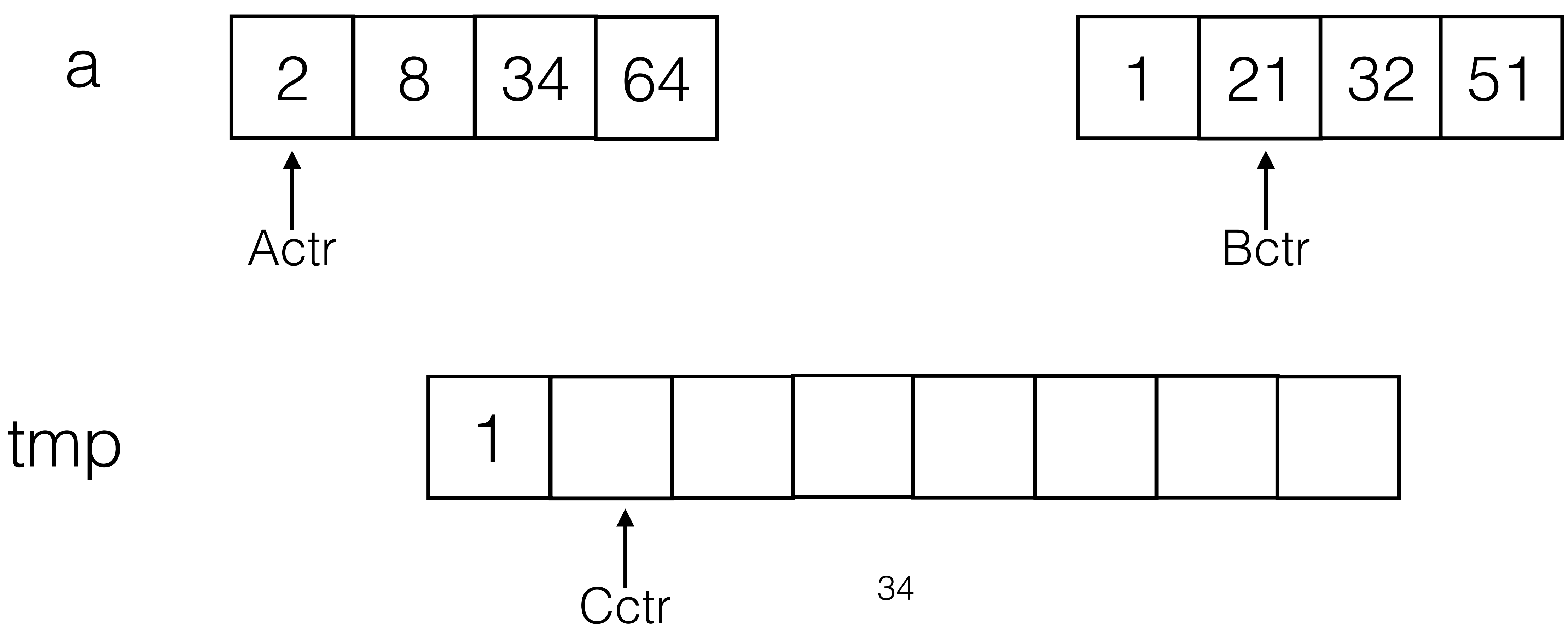
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
    - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
    - Otherwise, copy a[Bctr] to the output and advance Bctr.



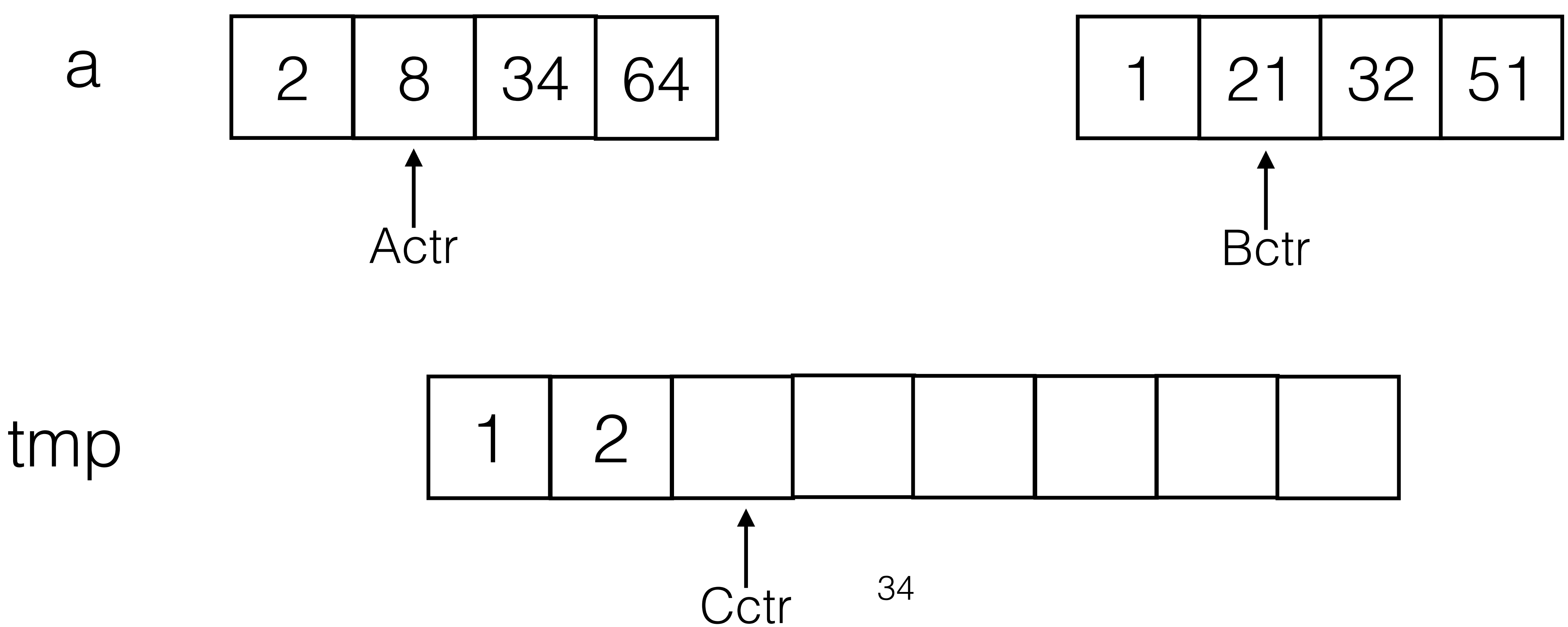
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



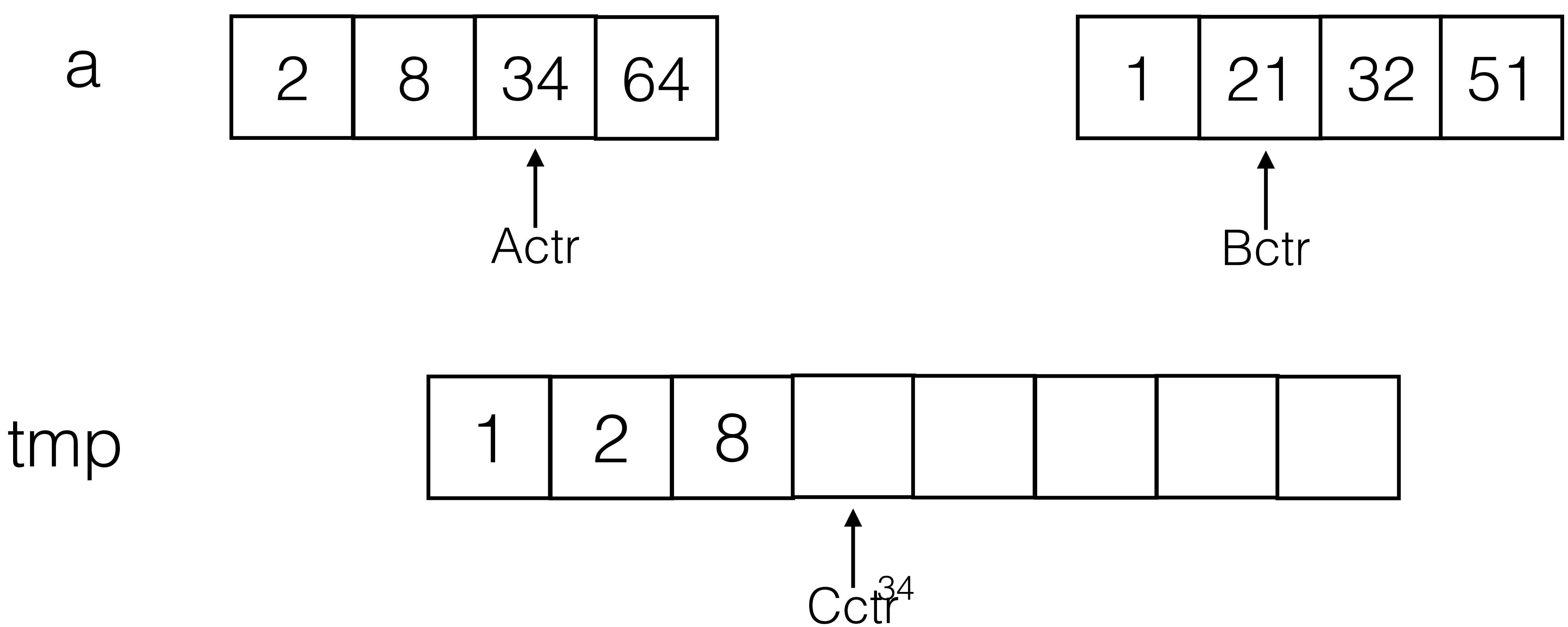
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



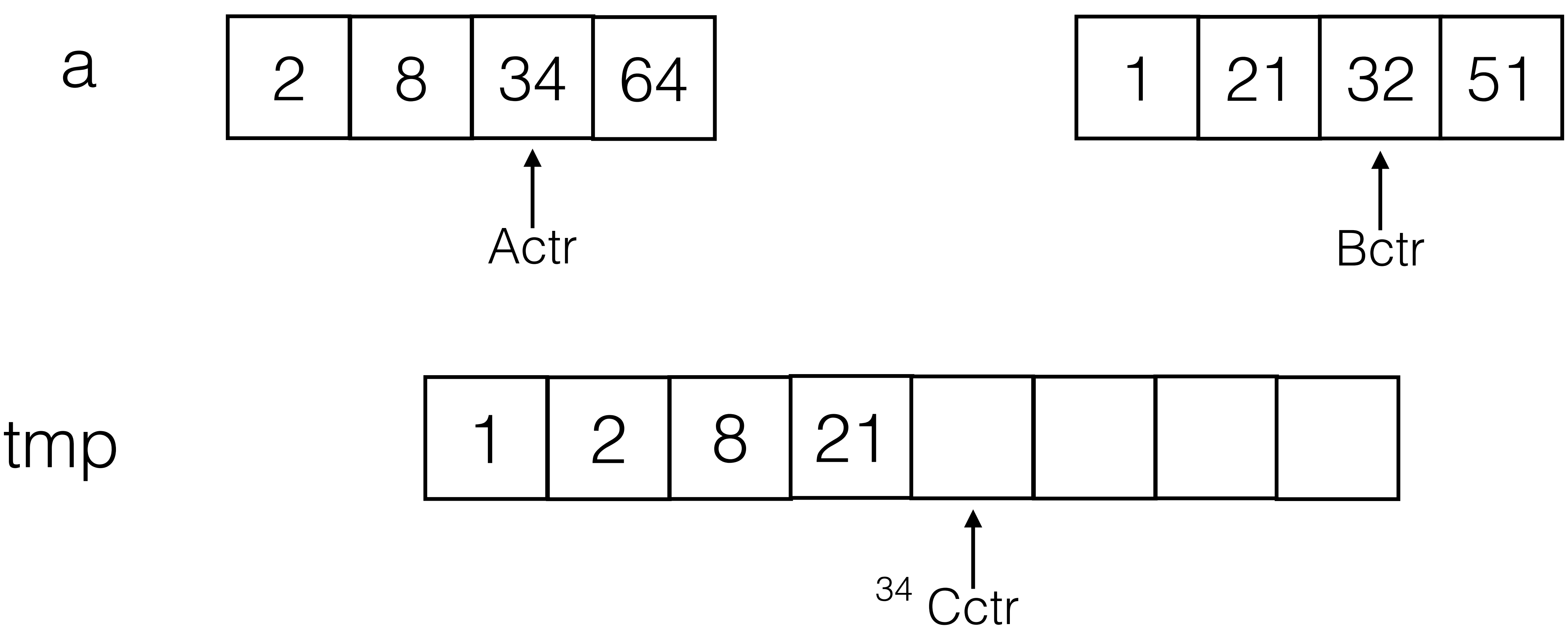
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



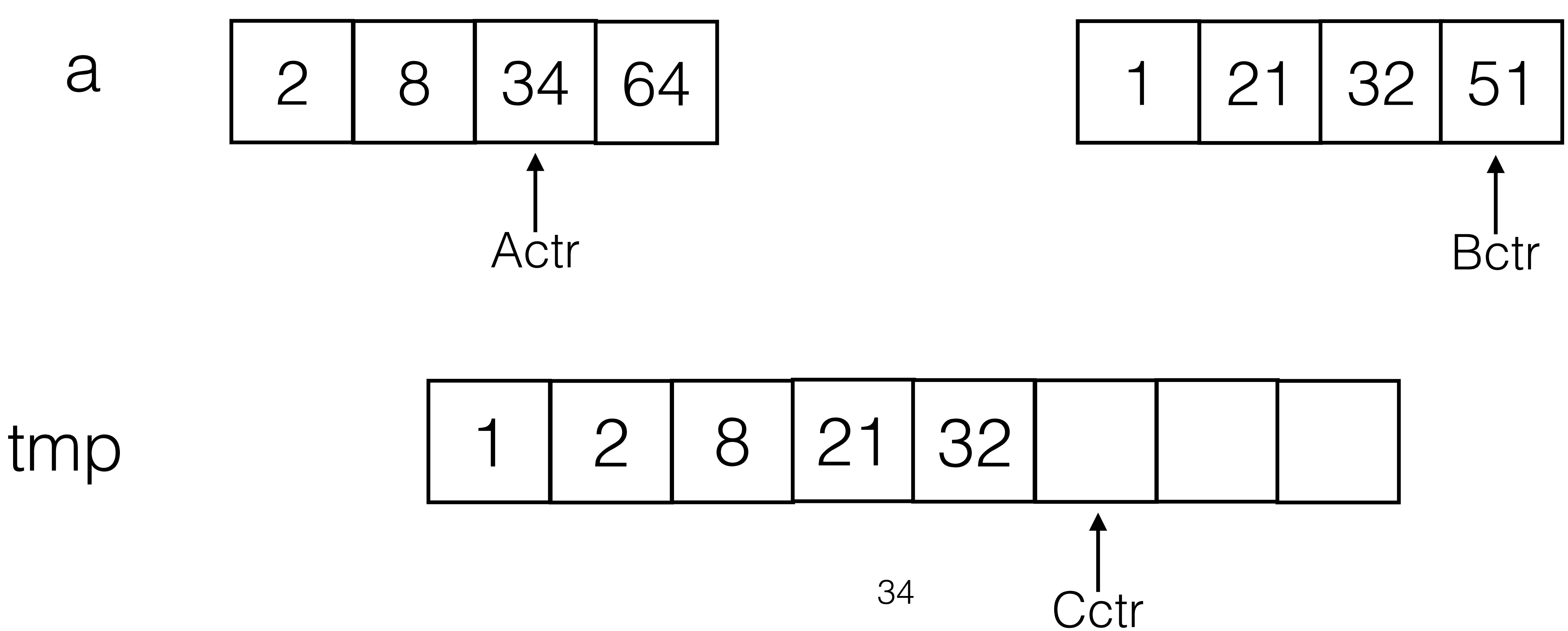
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



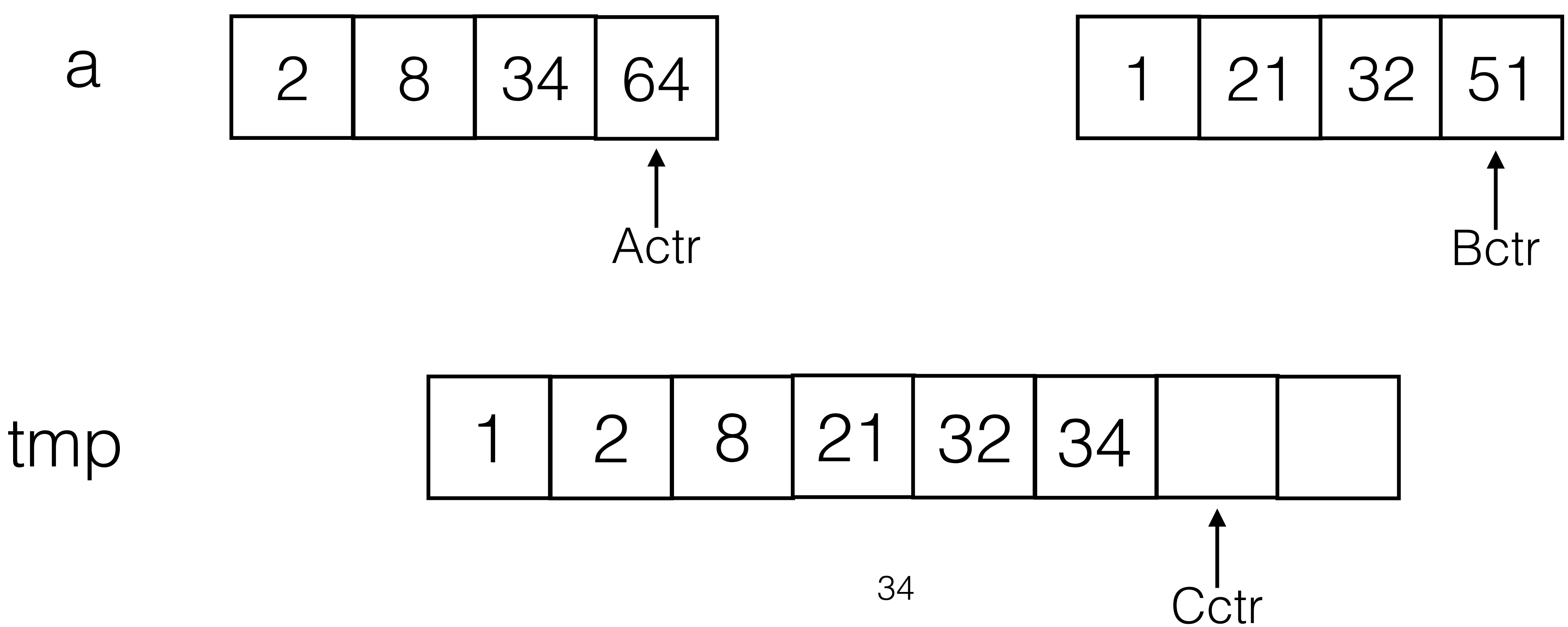
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
    - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
    - Otherwise, copy a[Bctr] to the output and advance Bctr.



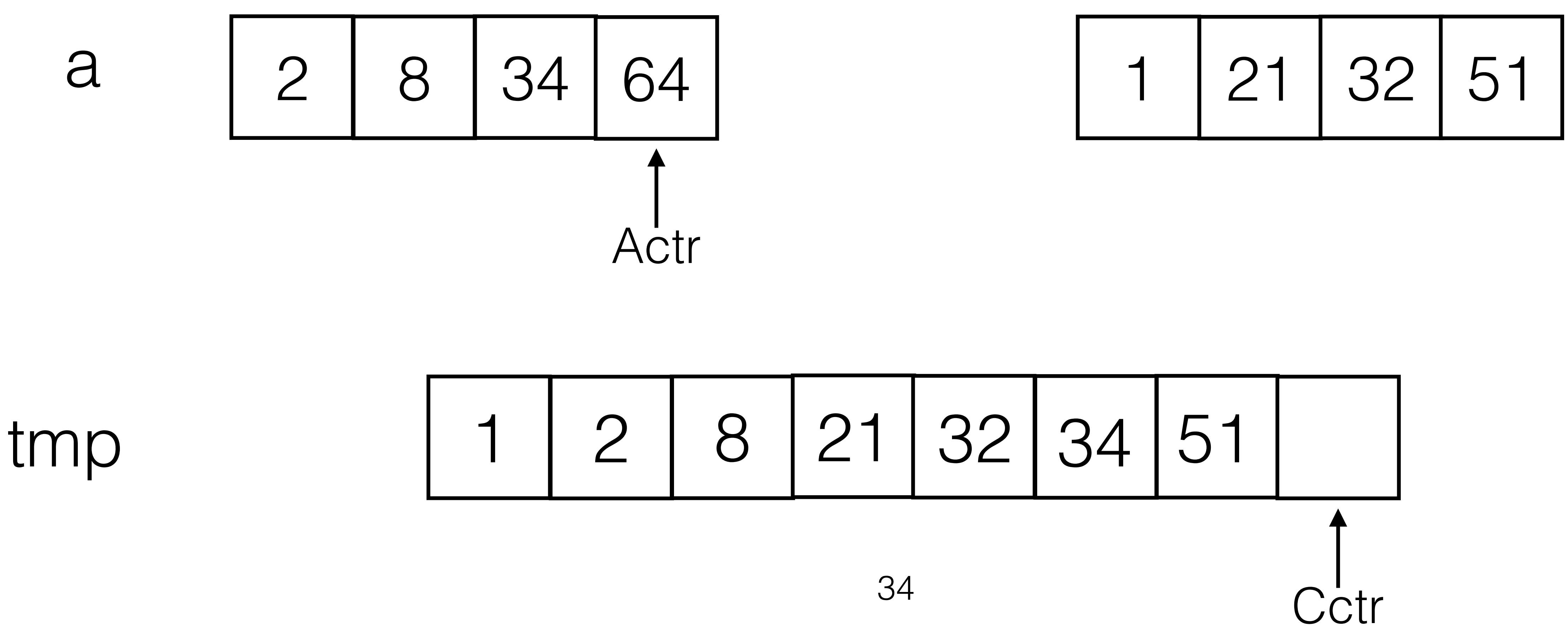
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.



Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If a[Actr] < a[Bctr], copy a[Actr] to tmp and advance Actr.
  - Otherwise, copy a[Bctr] to the output and advance Bctr.

a

2	8	34	64

1	21	32	51

tmp

1	2	8	21	32	34	51	64

Merging Sorted Sublists

```
private static <T extends Comparable<T>>
void merge( T[] a, T[] tmpArray, int aCtr, int bCtr, int rightEnd ) {
    int leftEnd = bCtr - 1;
    int tmpPos = aCtr;
    int numElements = rightEnd - aCtr + 1;

    // Main loop
    while( aCtr <= leftEnd && bCtr <= rightEnd )
        if( a[ aCtr ].compareTo( a[ bCtr ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ aCtr++ ];
        else
            tmpArray[ tmpPos++ ] = a[ bCtr++ ];

    while( aCtr <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ aCtr++ ];

    while( bCtr <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ bCtr++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2	51	32	21	1

Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2

51	32	21	1

Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2	51	32	21	1

Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2	51	32	21	1
8	34	2	64	32	51	1	21

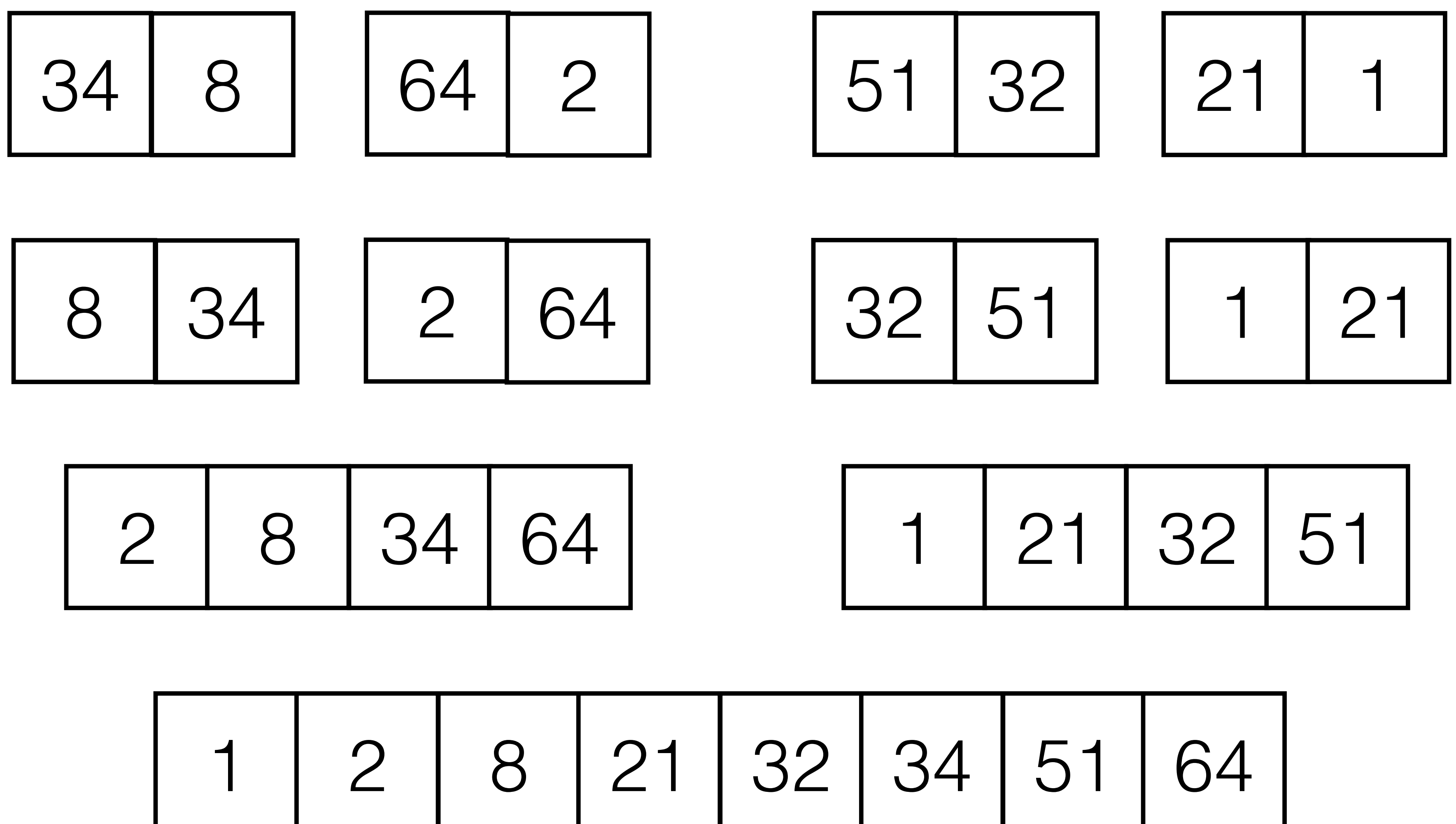
Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.

34	8	64	2	51	32	21	1
8	34	2	64	32	51	1	21
2	8	34	64	1	21	32	51

Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.



Merge Sort - Implementation

```
private static <T extends Comparable<T>>
void mergeSort( T[] a, T[] tmpArray, int left, int right )

    if( left < right ) {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }

}
```

Merge Sort Running Time

- This running time analysis is typical for divide and conquer algorithms.

- Merge sort is a recursive algorithm. The running time analysis should be similar to what we have seen for other algorithms of this type (e.g. binary search)

- Base case: N=1 (sort a 1-element list). T(1) = 1

- Recurrence: T(N) = 2 T(N/2) + N

Merge the two halfs

Recursively sort each half

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N$$

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

$$= N \cdot T(1) + logN \cdot N$$

Merge Sort Running Time

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

$$= N \cdot T(1) + logN \cdot N$$

$$= N + N \cdot \log N = \Theta(N \log N)$$

Merge Sort Properties

- Worst case running time: $\Theta(N \log N)$
- Is MergeSort stable?
- Space requirement?

Merge Sort Properties

- Worst case running time: $\Theta(N \log N)$
- Is MergeSort stable?
  Yes. Merging preservers order of elements.
- Space requirement?

Merge Sort Properties

- Worst case running time: $\Theta(N \log N)$

- Is MergeSort stable?

  Yes. Merging preservers order of elements.

- Space requirement?

  Need a temporary array. O(N)

Quick Sort

- Another divide-and-conquer algorithm.
  - Pick any **pivot** element v.
  - Partition the array into elements
    - x ≤ v and x ≥ v.
  - Recursively sort the partitions, then concatenate them.

34	8	64	2	51	32	21	1

Quick Sort

- Another divide-and-conquer algorithm.
- Pick any **pivot** element v.
- Partition the array into elements
  - $x \leq v$ and $x \geq v$.
- Recursively sort the partitions, then concatenate them.

34	8	64	2	51	32	21	1

21

v

Quick Sort

- Another divide-and-conquer algorithm.
- Pick any **pivot** element v.
- Partition the array into elements
  - x ≤ v and x ≥ v.
- Recursively sort the partitions, then concatenate them.

34	8	64	2	51	32	21	1

8	2	1

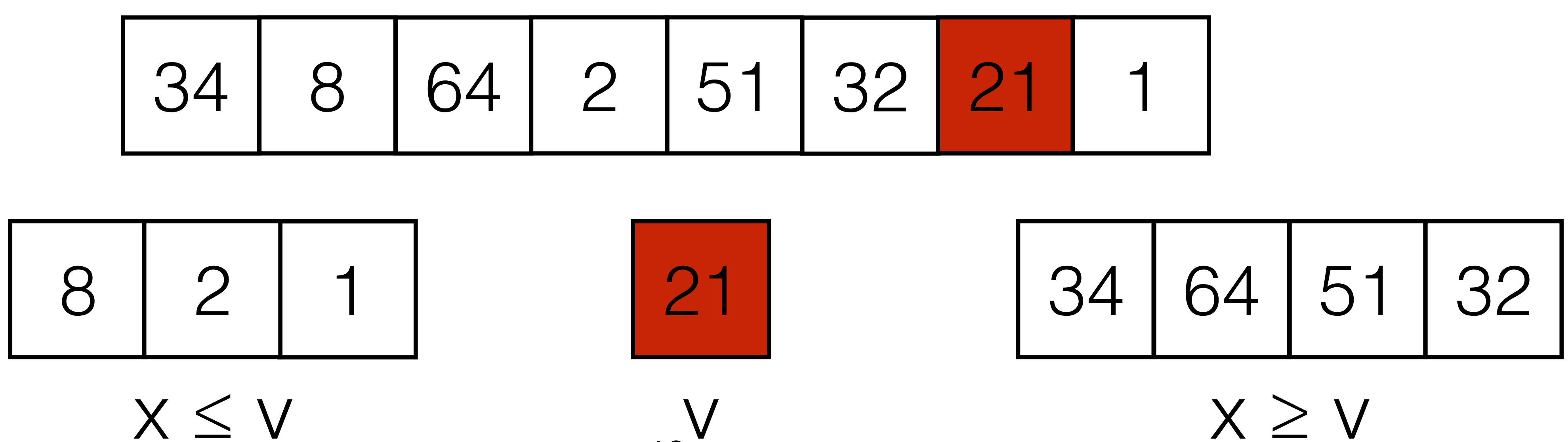
x ≤ v

21

v

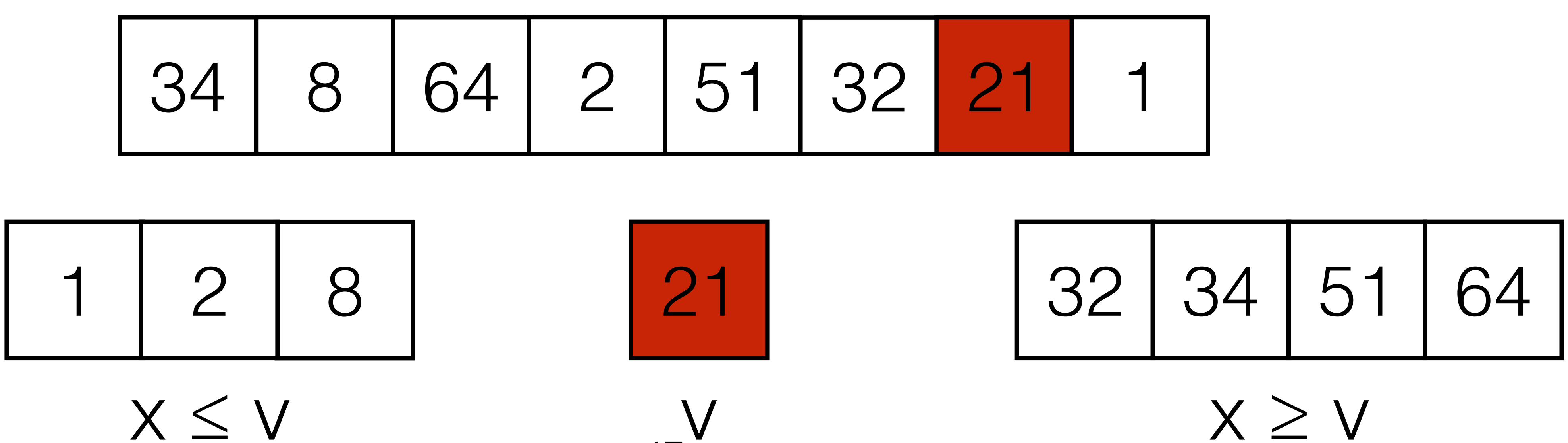
Quick Sort

- Another divide-and-conquer algorithm.
  - Pick any **pivot** element v.
  - Partition the array into elements
    - x ≤ v and x ≥ v.
  - Recursively sort the partitions, then concatenate them.



Quick Sort

- Another divide-and-conquer algorithm.
- Pick any **pivot** element v.
- Partition the array into elements
  - x ≤ v and x ≥ v.
- Recursively sort the partitions, then concatenate them.



Quick Sort

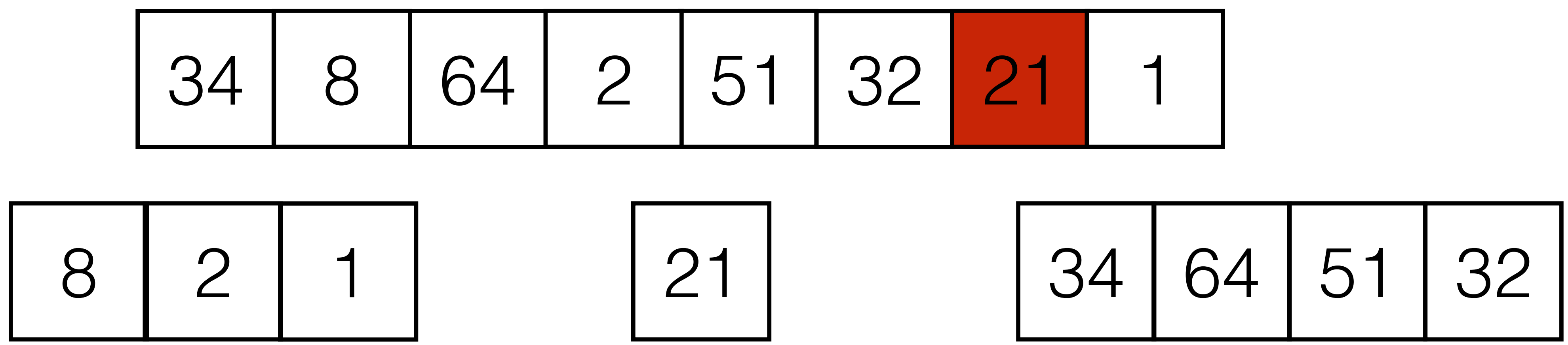
- Another divide-and-conquer algorithm.
- Pick any **pivot** element v.
- Partition the array into elements
  - x ≤ v and x ≥ v.
- Recursively sort the partitions, then concatenate them.

34	8	64	2	51	32	21	1

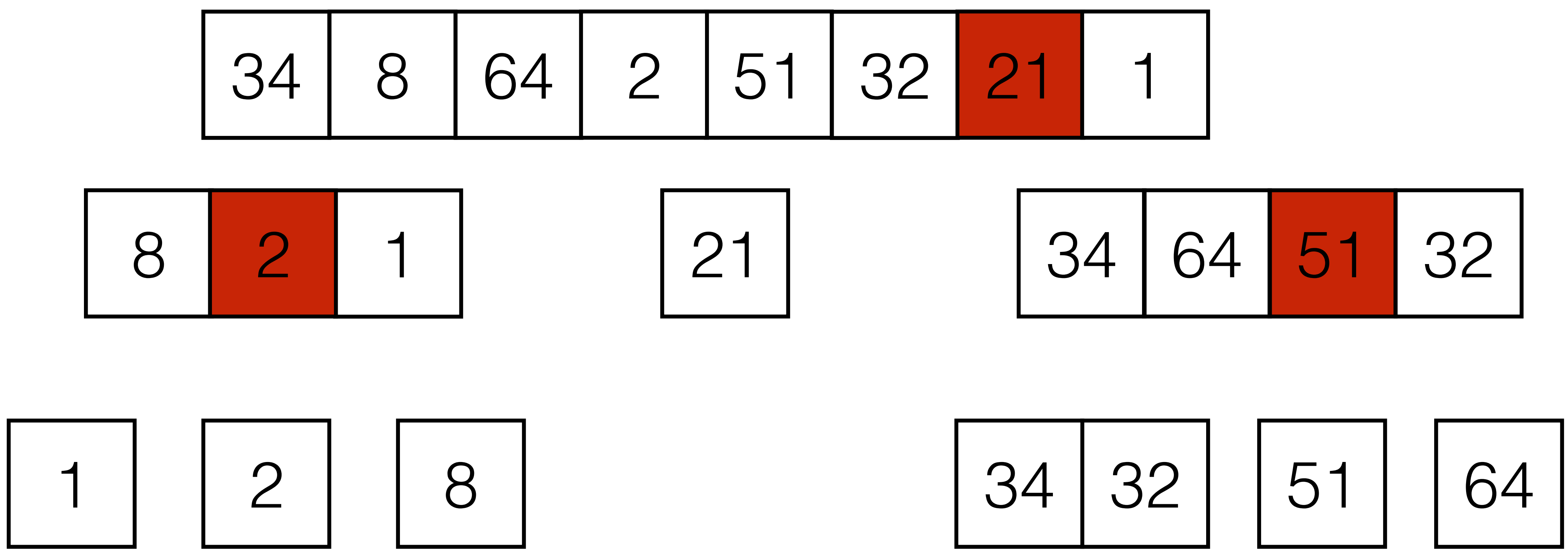
1	2	8	21	32	34	51	64

x ≤ v   v   x ≥ v

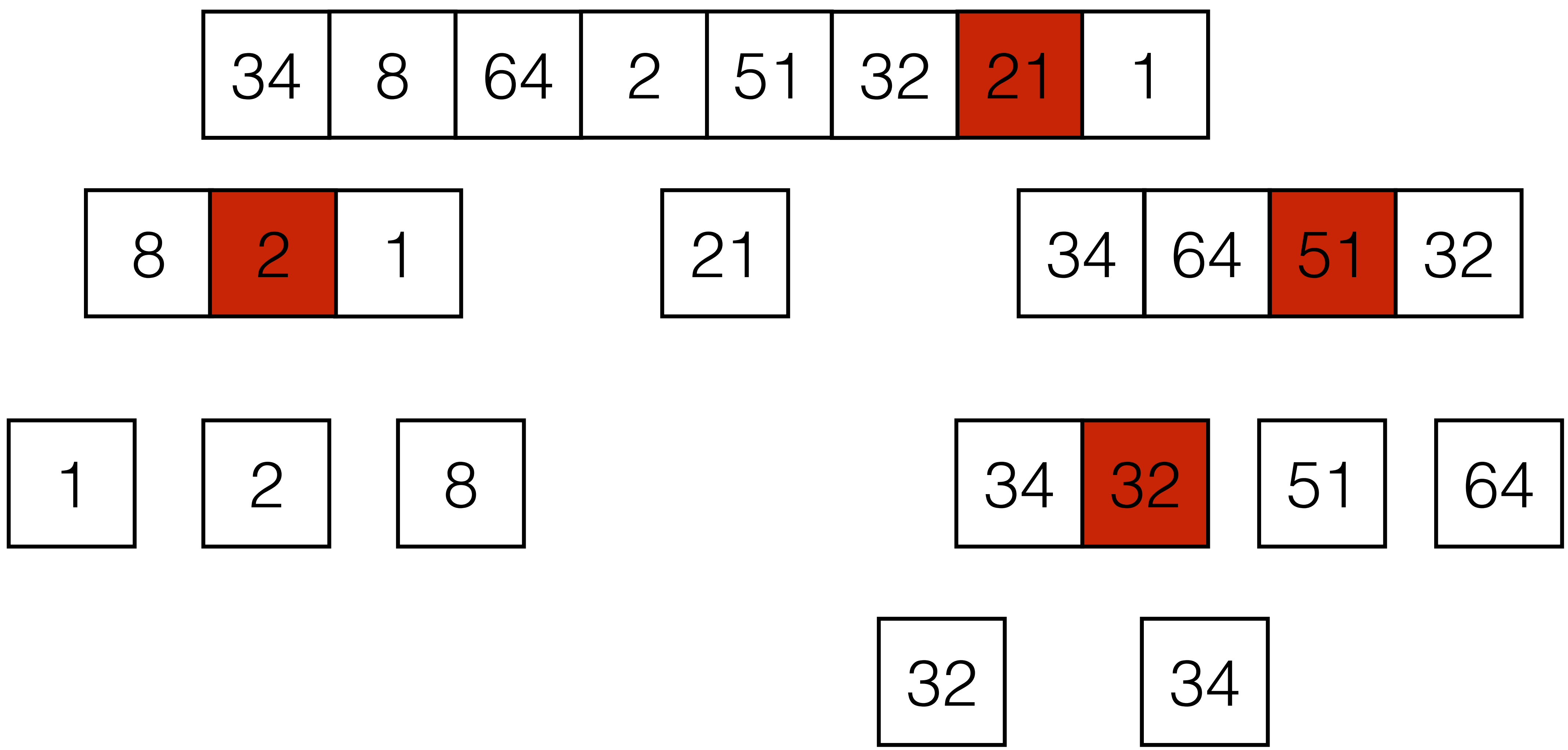
Quick Sort



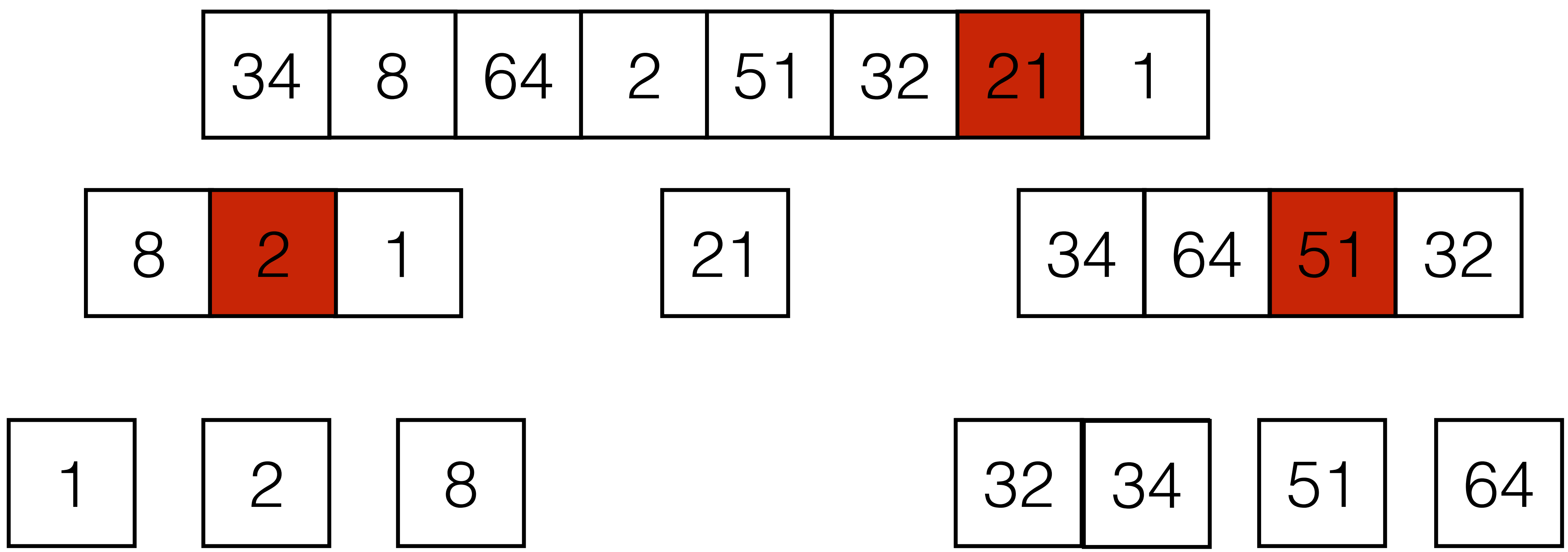
Quick Sort



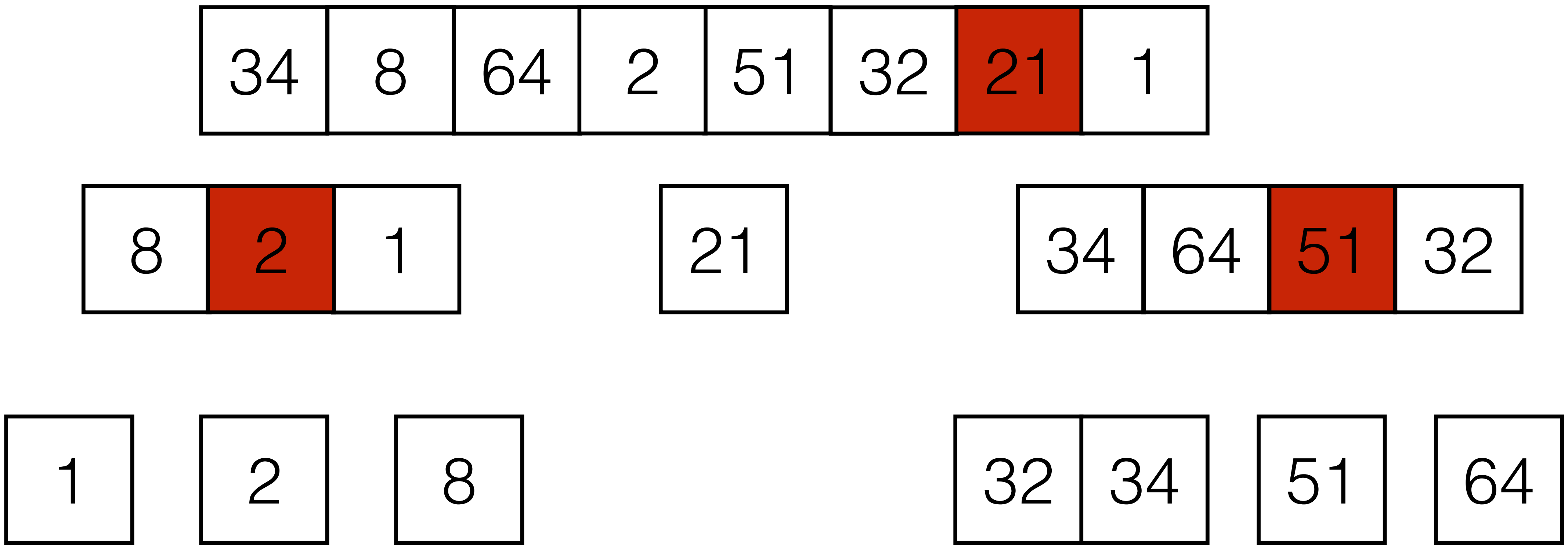
Quick Sort



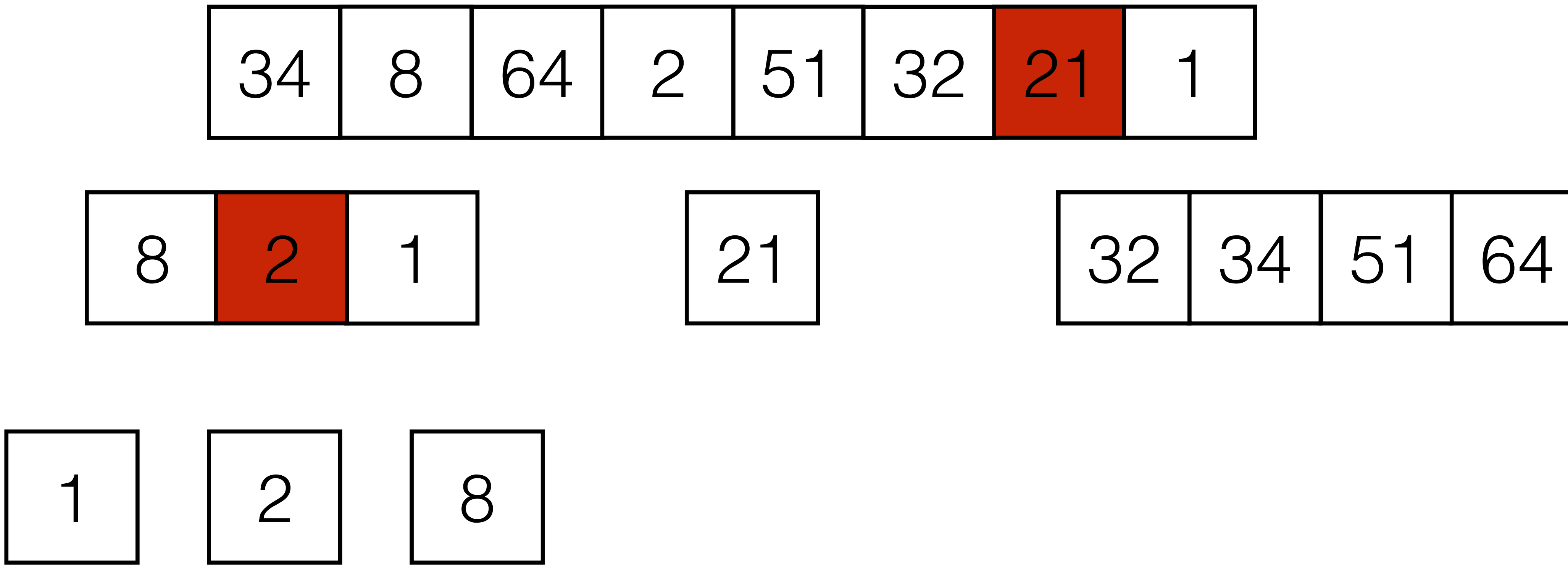
Quick Sort



Quick Sort



Quick Sort



Quick Sort

34	8	64	2	51	32	21	1

1	2	8

21

32	34	51	64

Quick Sort

34	8	64	2	51	32	21	1

1	2	8

21

32	34	51	64

Quick Sort

1	2	8	21	32	34	51	64

Quick Sort

1	2	8	21	32	34	51	64

- How do we partition the array efficiently (in place)?
- How do we pick a pivot element?
  - Running time performance on quick sort depends on our choice.
  - Bad choice leads to $\Theta(N^2)$ running time.

Partitioning the Array

- We don’t want to use any extra space. Need to partition the array in place.

- Use swaps to push all elements $x \leq v$ to the left and elements $x \geq v$ to the right.

34	8	64	2	51	32	21	1

Partitioning the Array

- We don’t want to use any extra space. Need to partition the array in place.

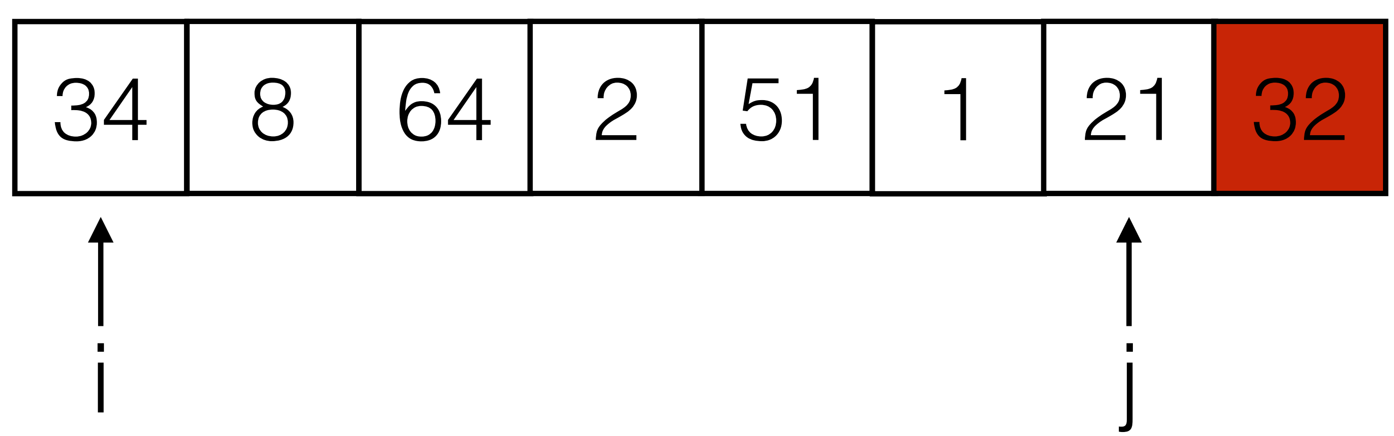
- Use swaps to push all elements $x \leq v$ to the left and elements $x \geq v$ to the right.

34	8	64	2	51	1	21	32

Move the pivot to the end.

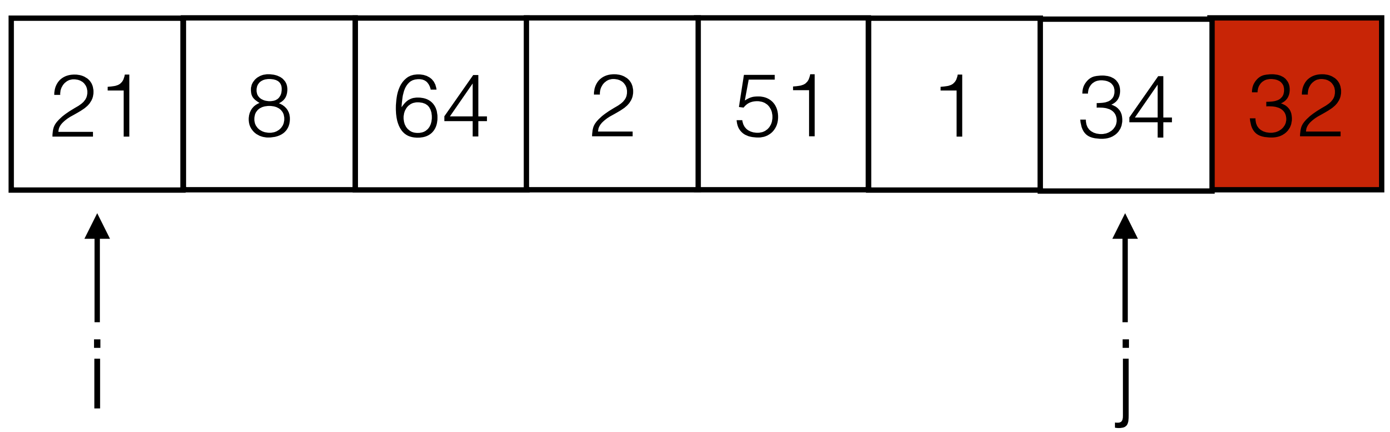
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



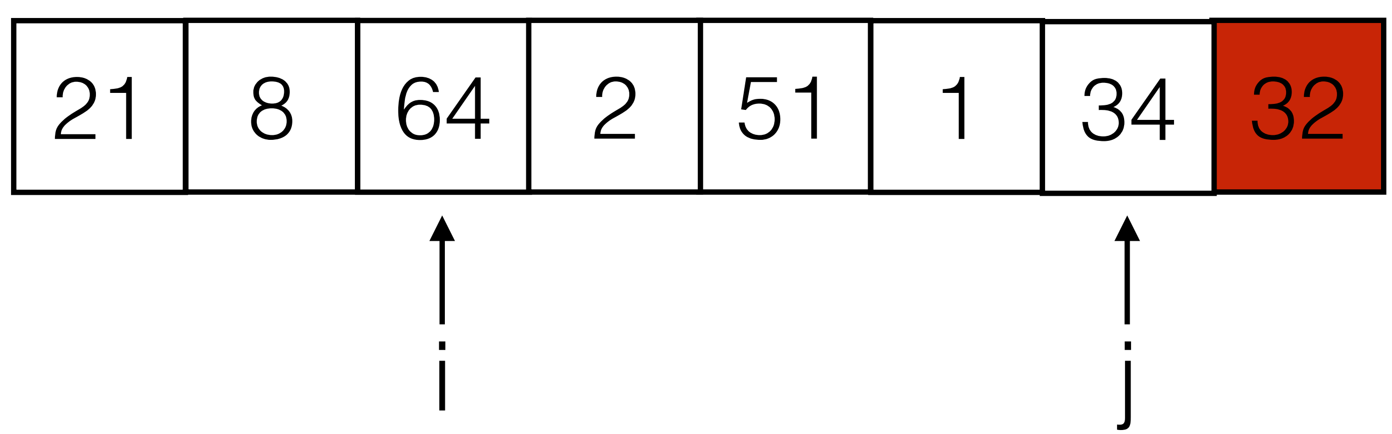
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



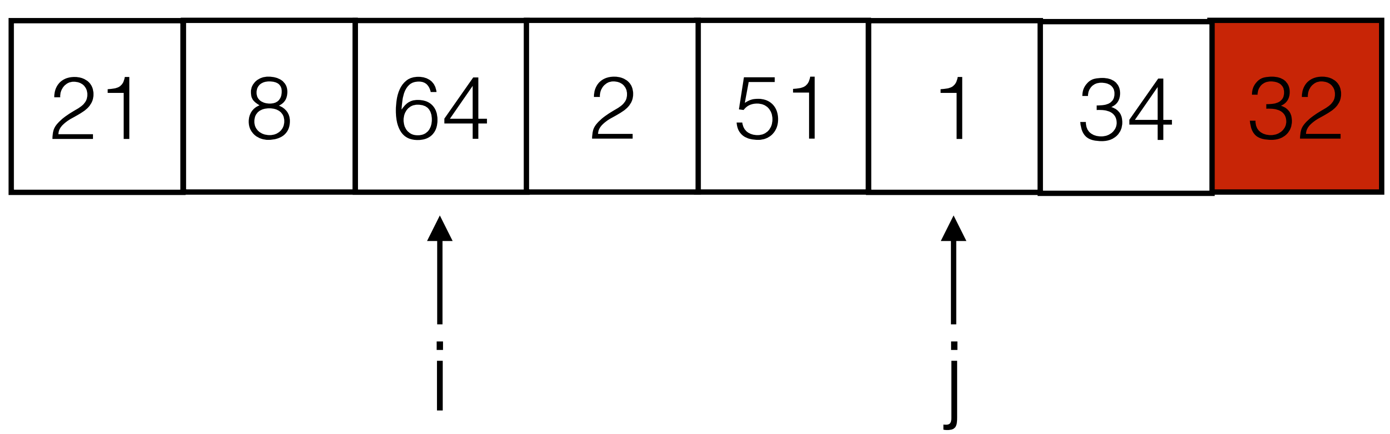
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



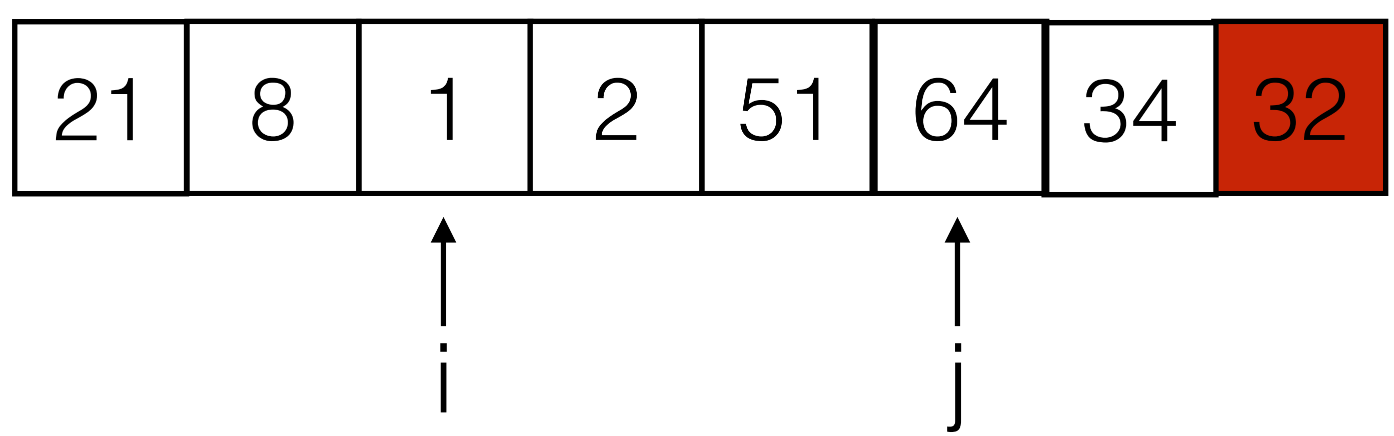
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



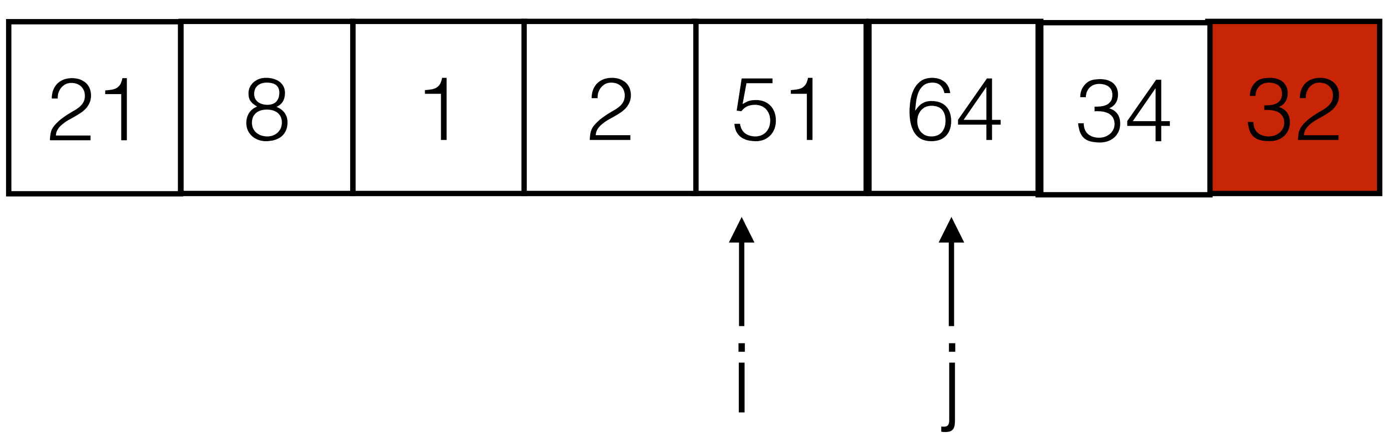
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



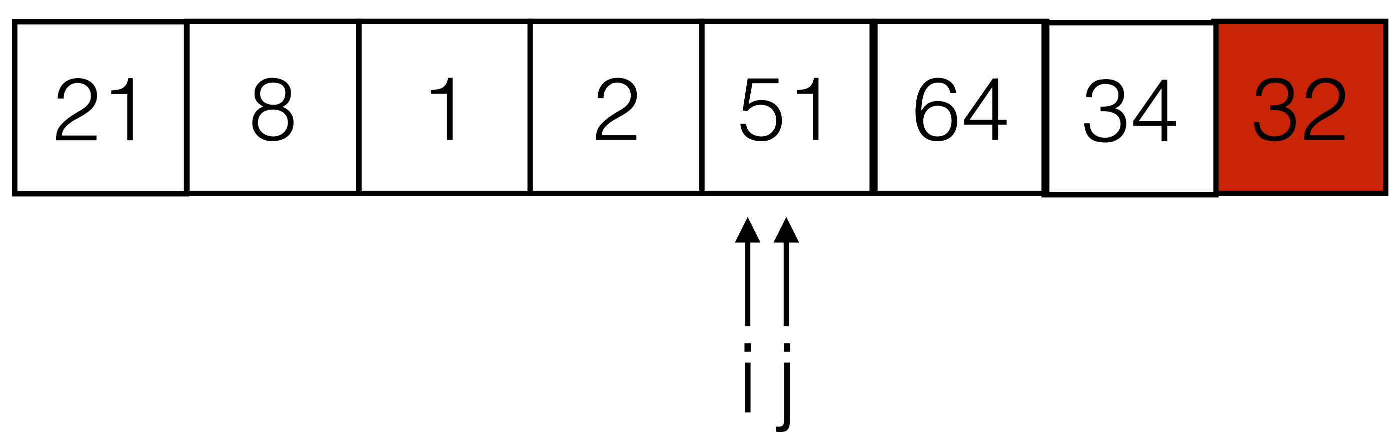
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



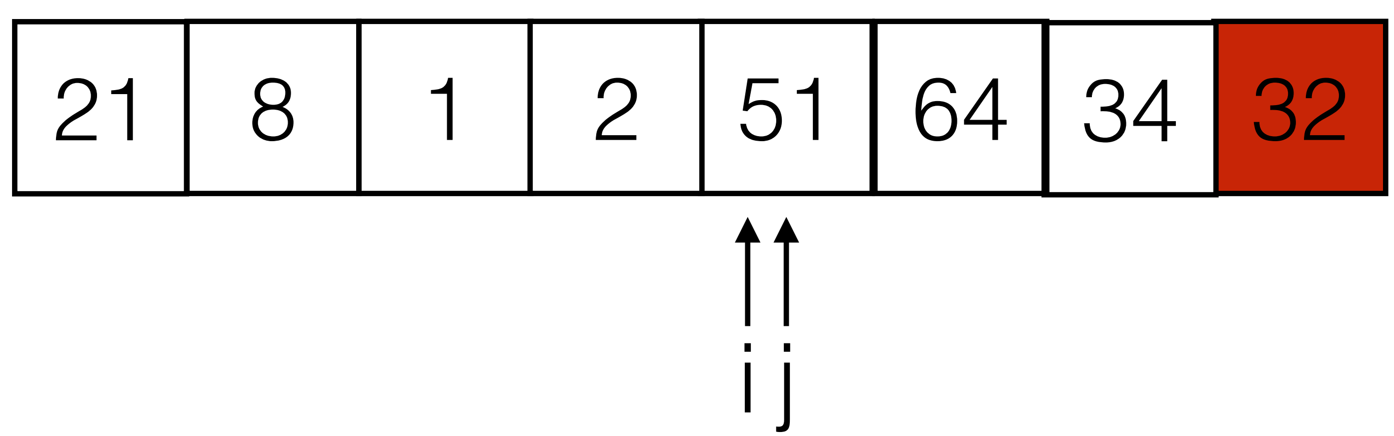
Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].



Partitioning the Array

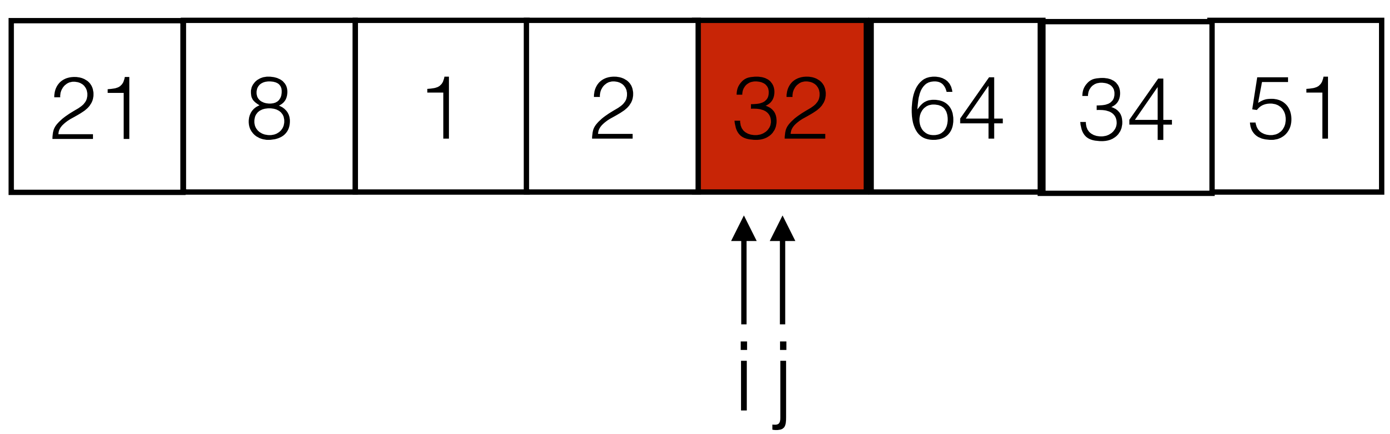
- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].
- Swap array[i] with v.



- i points to a value greater than the pivot.

Partitioning the Array

- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].
- Swap array[i] with v.



- i points to a value greater than the pivot.

Partitioning the Array

```
public static void quicksort(Integer[] a, int left, int right) {

    if (right > left) {
        int v = find_pivot_index(a, left, right);
        int i = left;   int j = right-1;

        // move pivot to the end
        Integer tmp = a[v]; a[v] = a[right]; a[right] = tmp;

        while (true) { // partition
            while (a[++i] < a[v]) {};
            while (a[—j] > a[v]r) {};
            if (i >= j) break;
            tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }

        // move pivot back
        tmp = a[i]; a[i] = a[right]; a[right] = tmp;
        //recursively sort both partitions
        quicksort(a,left, i-1);  quicksort(a,i+1, right);
    }
}
```

Partitioning the Array

```
public static void quicksort(Integer[] a, int left, int right) {

    if (right > left) {
        int v = find_pivot_index(a, left, right);
        int i = left;   int j = right-1;

        // move pivot to the end
        Integer tmp = a[v]; a[v] = a[right]; a[right] = tmp;

        while (true) { // partition
            while (a[++i] < a[v]) {};
            while (a[—j] > a[v]r) {};
            if (i >= j) break;
            tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }

        // move pivot back
        tmp = a[i]; a[i] = a[right]; a[right] = tmp;
        //recursively sort both partitions
        quicksort(a,left, i-1);  quicksort(a,i+1, right);
    }
}
```

O(N)

Quick Sort: Worst Case

- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!

34	8	64	2	51	32	21	1

Quick Sort: Worst Case

- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!

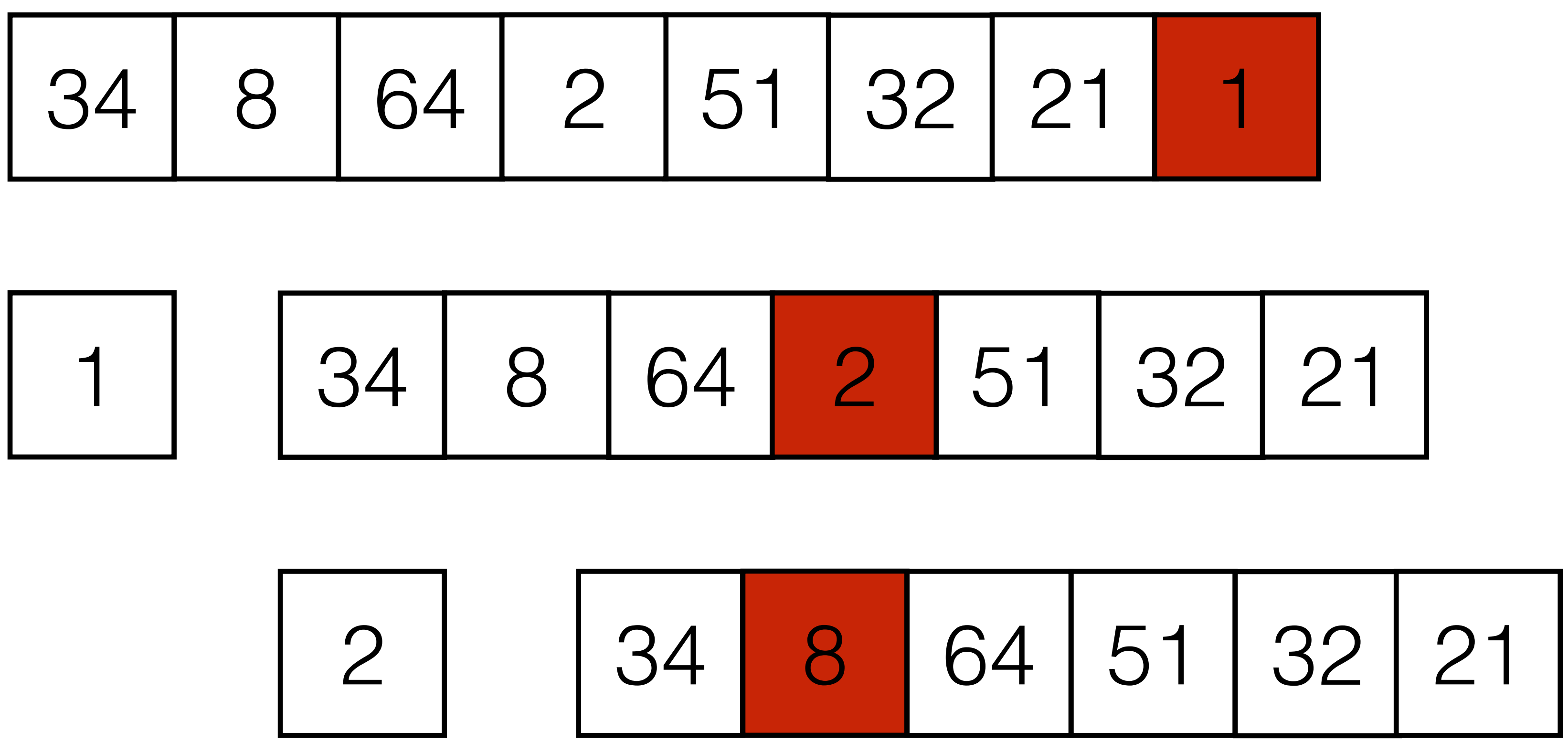
34	8	64	2	51	32	21	1

1		34	8	64	2	51	32	21

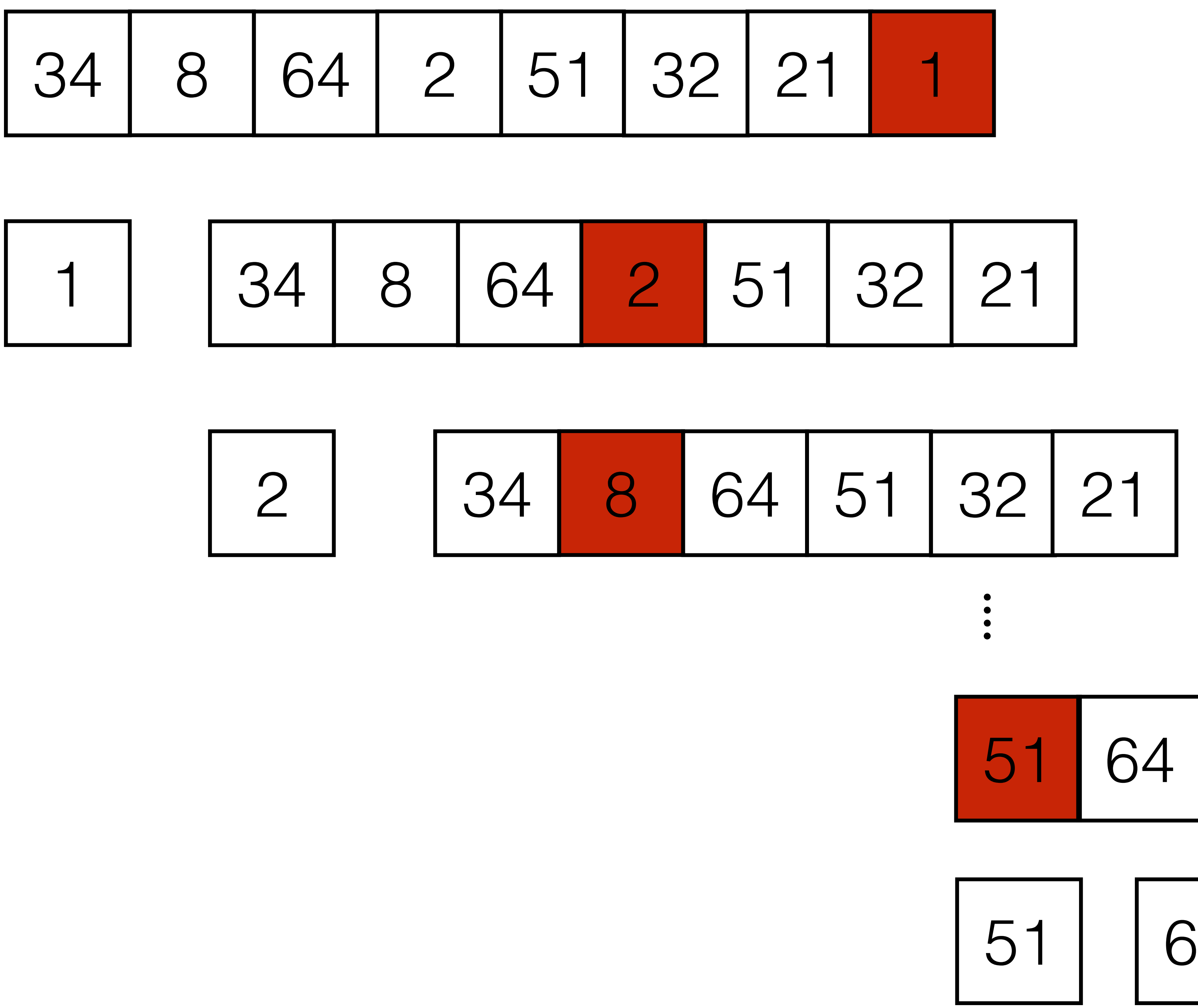
.
.
.

Quick Sort: Worst Case

- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!

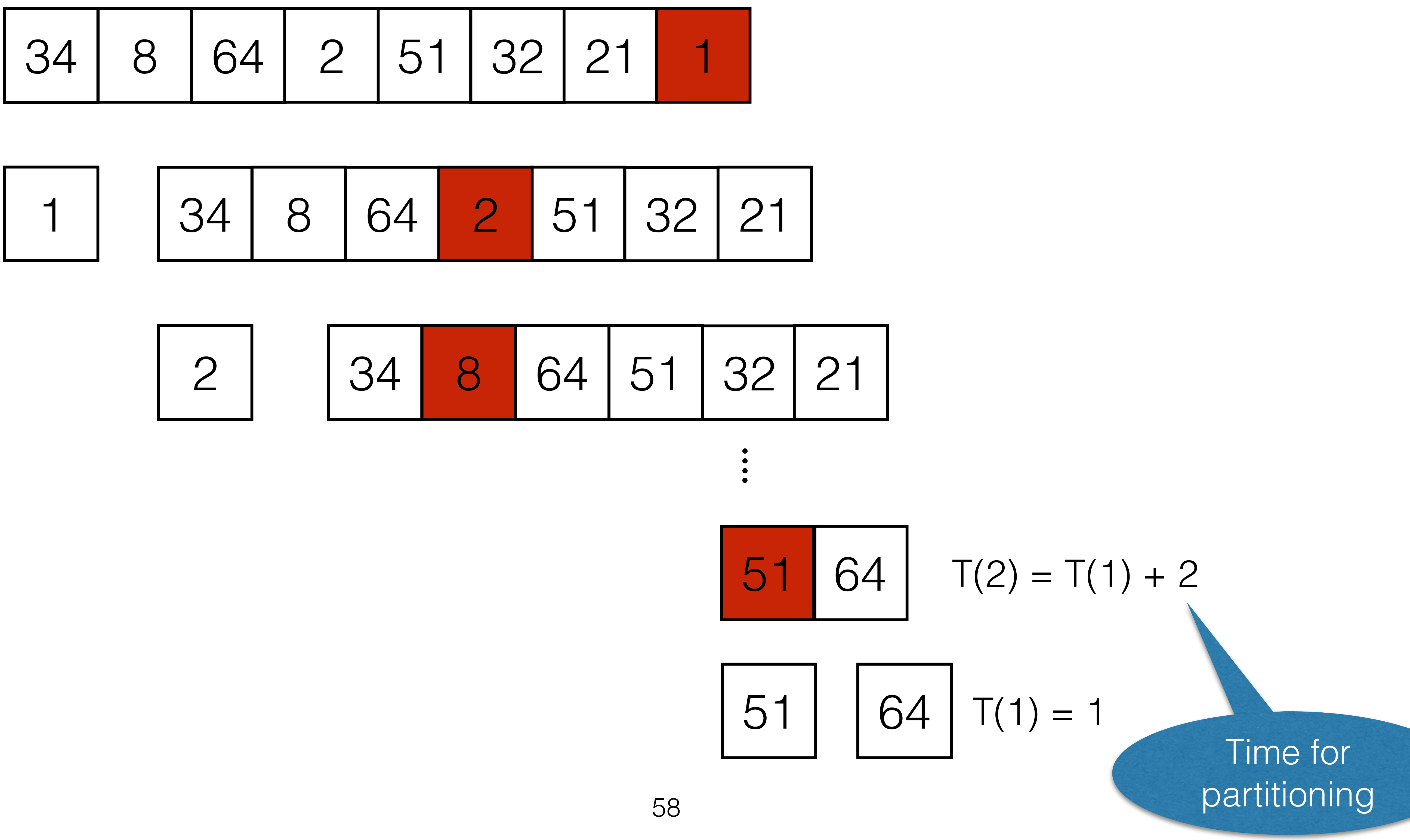


Quick Sort: Worst Case

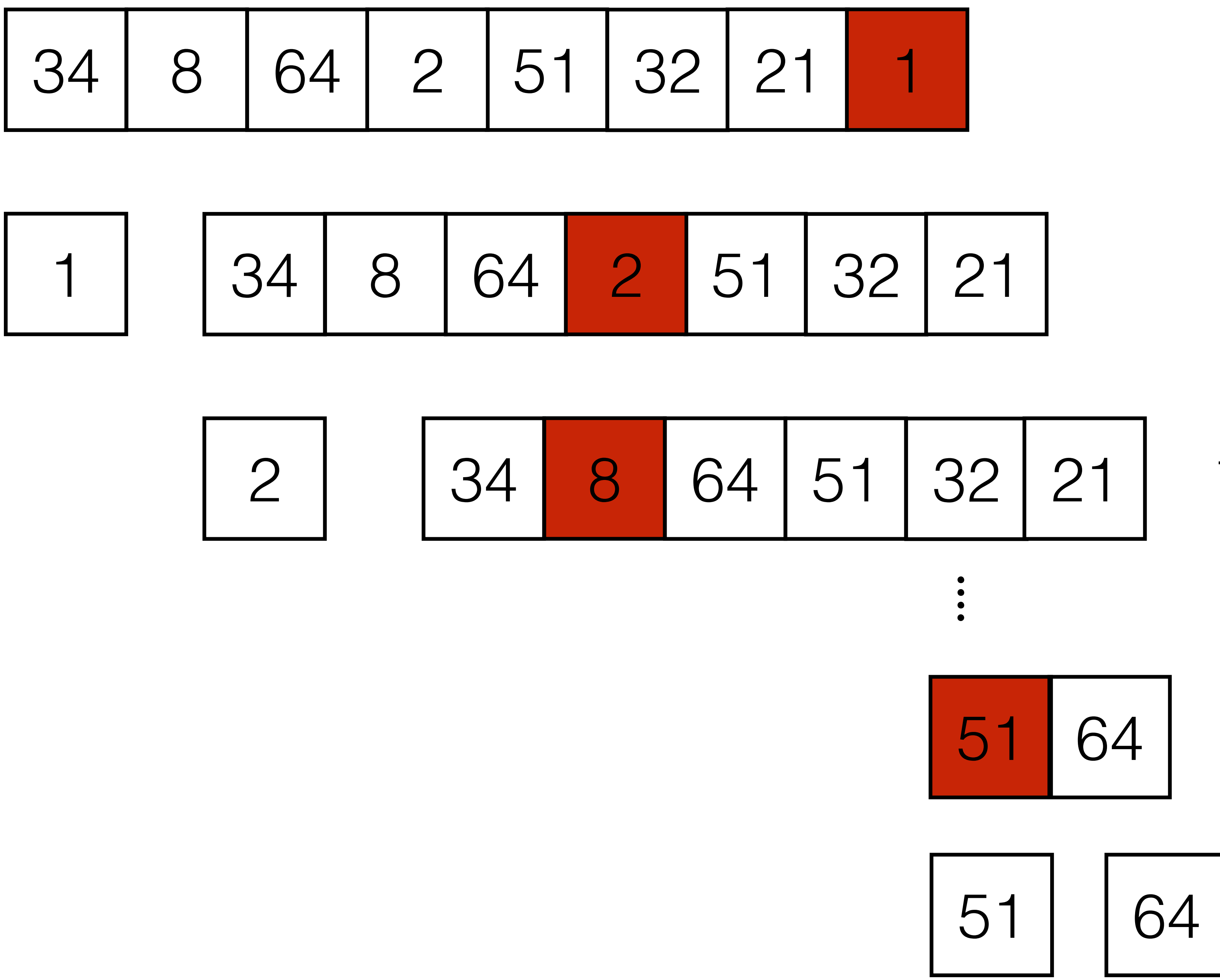


T(1) = 1

Quick Sort: Worst Case



Quick Sort: Worst Case



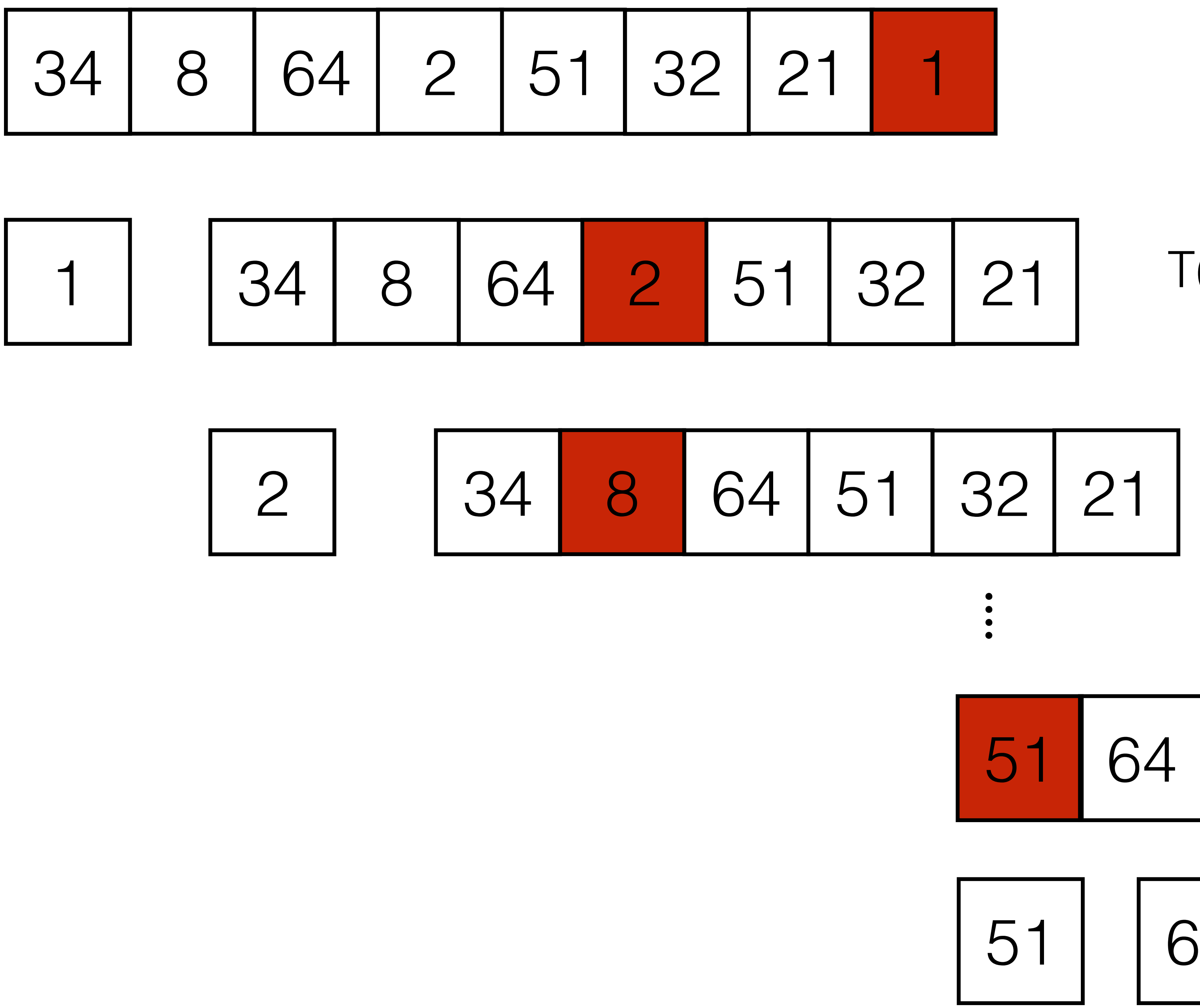
$T(N-2) = T(N-3) + (N-2)$

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

$T(1) = 1$

Time for partitioning

Quick Sort: Worst Case



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

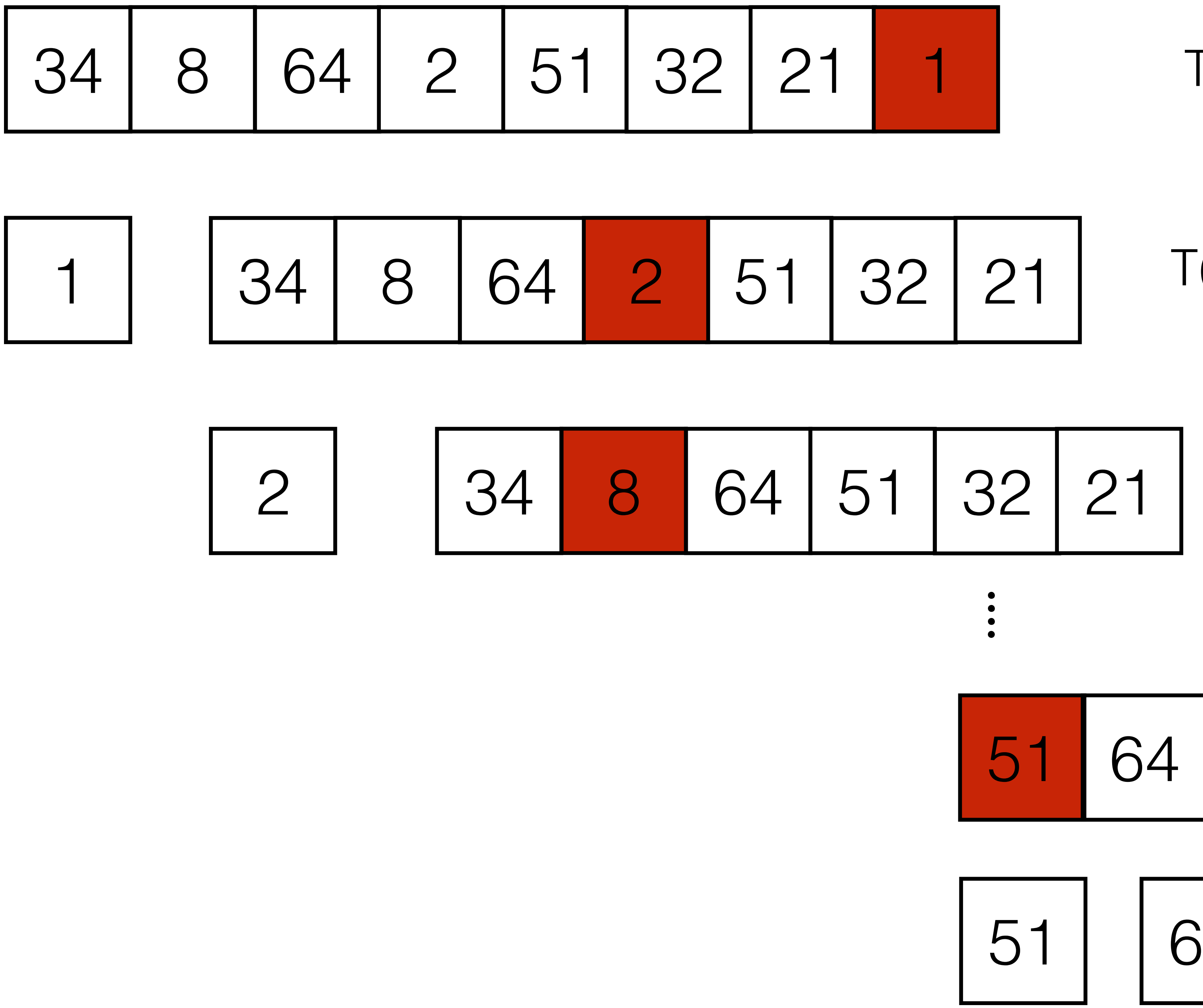
$T(N-2) = T(N-3) + (N-2)$

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

$T(1) = 1$

Time for partitioning

Quick Sort: Worst Case



$T(N) = T(N-1) + N$

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

$T(N-2) = T(N-3) + (N-2)$

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

$T(1) = 1$

Time for partitioning

Quick Sort: Worst Case

$$T(N) = T(N-1) + N$$

Quick Sort: Worst Case

$$
\begin{aligned}
T(N) &= T(N-1) + N \\
&= T(N-2) + (N-1) + N
\end{aligned}
$$

Quick Sort: Worst Case

$$
\begin{aligned}
T(N) &= T(N-1) + N \\
&= T(N-2) + (N-1) + N \\
&= T(N-k) + (N-(k-1)) + \cdots + (N-1) + N \\
&\ \ \vdots \\
&= T(1) + 2 + 3 + \cdots + (N-1) + N
\end{aligned}
$$

Quick Sort: Worst Case

$$T(N) = T(N-1) + N$$

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

$$= T(N-k) + (N-(k-1)) + \cdots + (N-1) + N$$

$$\vdots$$

$$= T(1) + 2 + 3 + \cdots + (N-1) + N$$

$$= 1 + \sum_{i=2}^{N} i = \sum_{i=1}^{N} i$$

Quick Sort: Worst Case

$$T(N) = T(N-1) + N$$

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

$$= T(N-k) + (N-(k-1)) + \cdots + (N-1) + N$$

$$\vdots$$

$$= T(1) + 2 + 3 + \cdots + (N-1) + N$$

$$= 1 + \sum_{i=2}^{N} i = \sum_{i=1}^{N} i$$

$$= N\,\frac{N+1}{2} = \Theta(N^2)$$

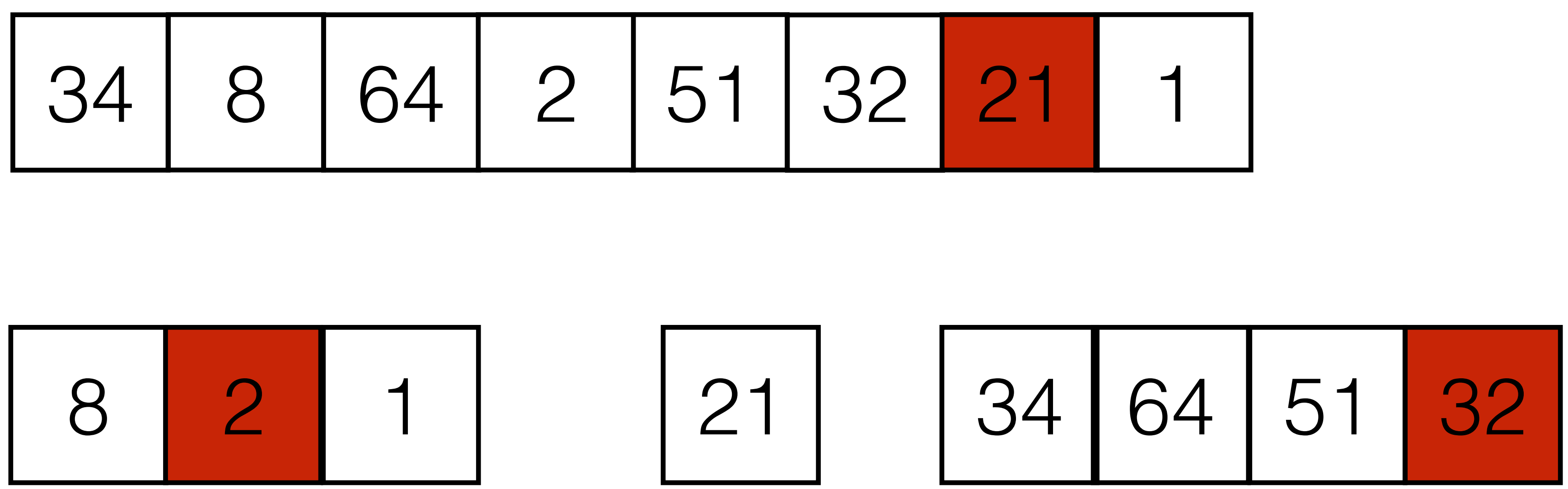
Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

34	8	64	2	51	32	21	1

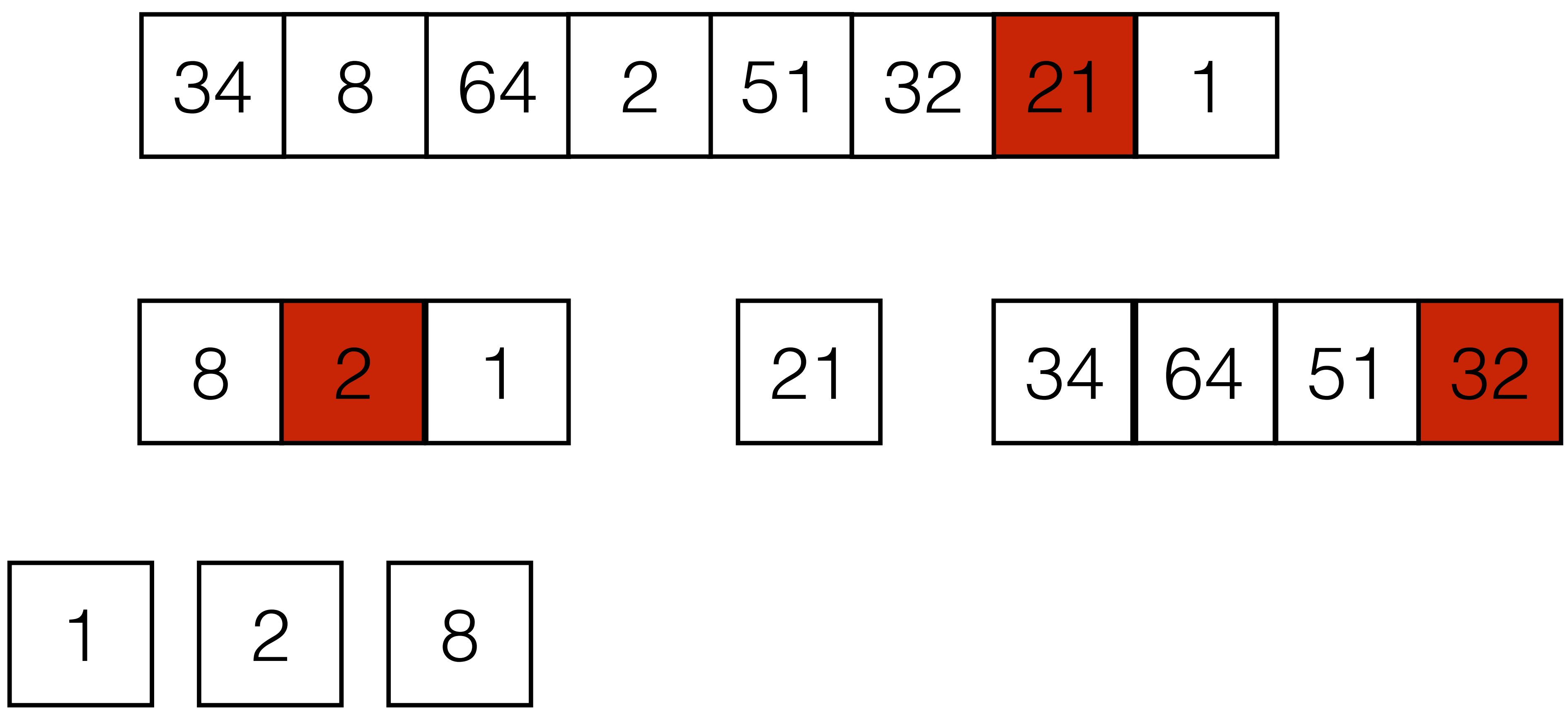
Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.



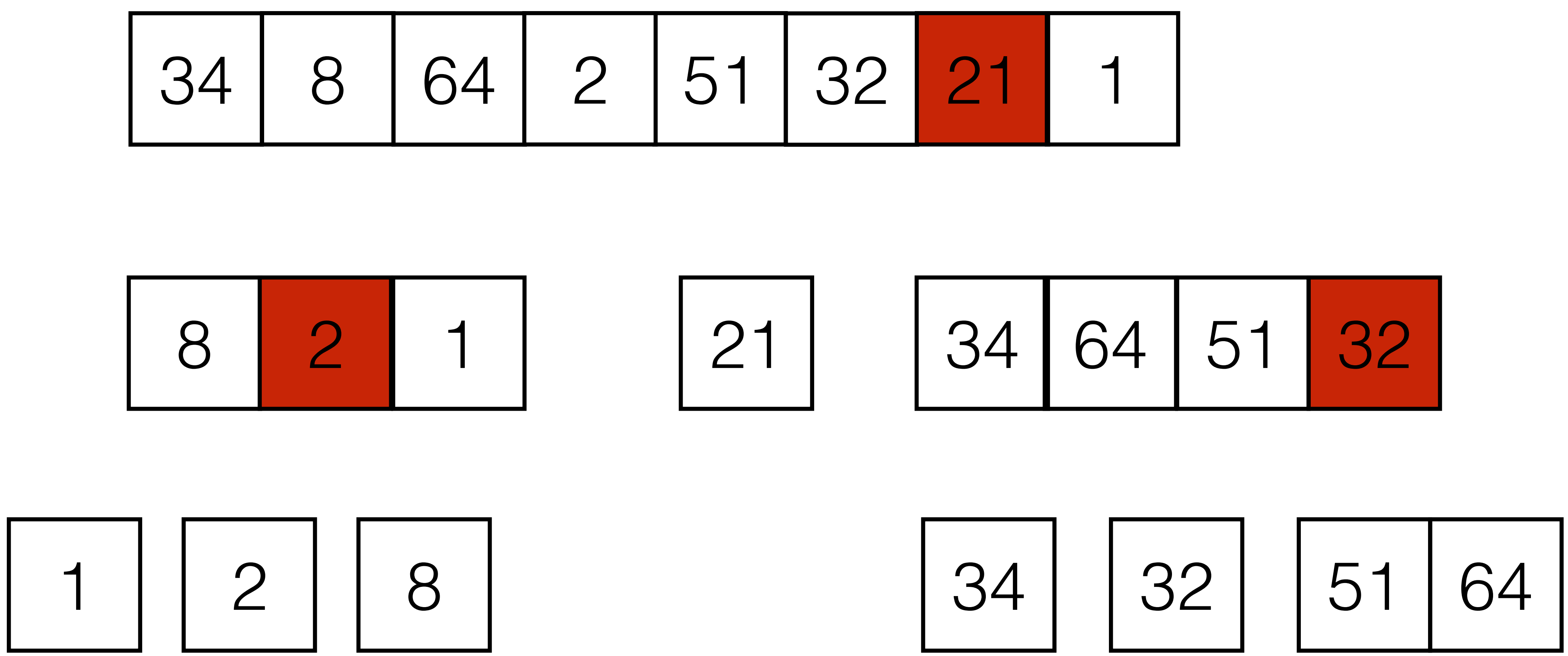
Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.



Quick Sort: Best Case

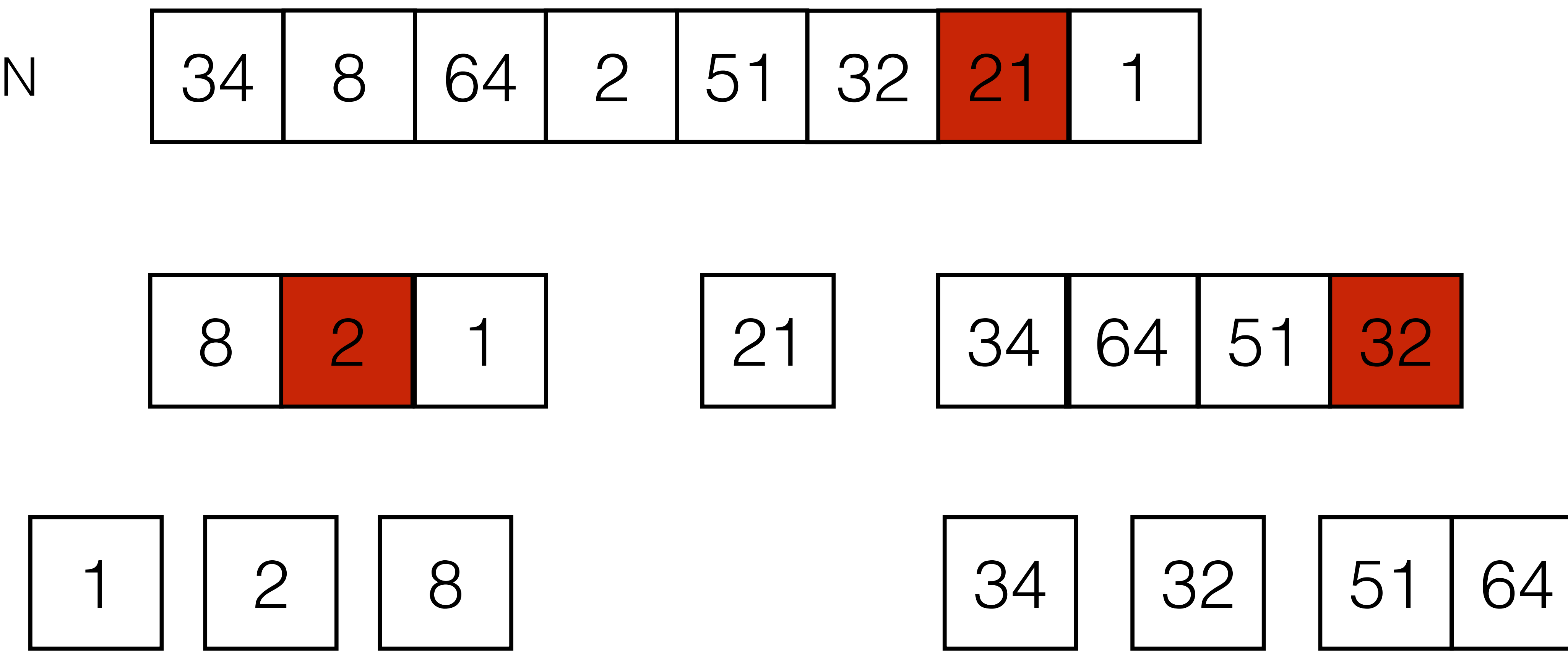
- Best case: Pivot is always the median element.
  Both partitions have about the same size.



Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

$T(N) = 2\,T(N/2) + N$



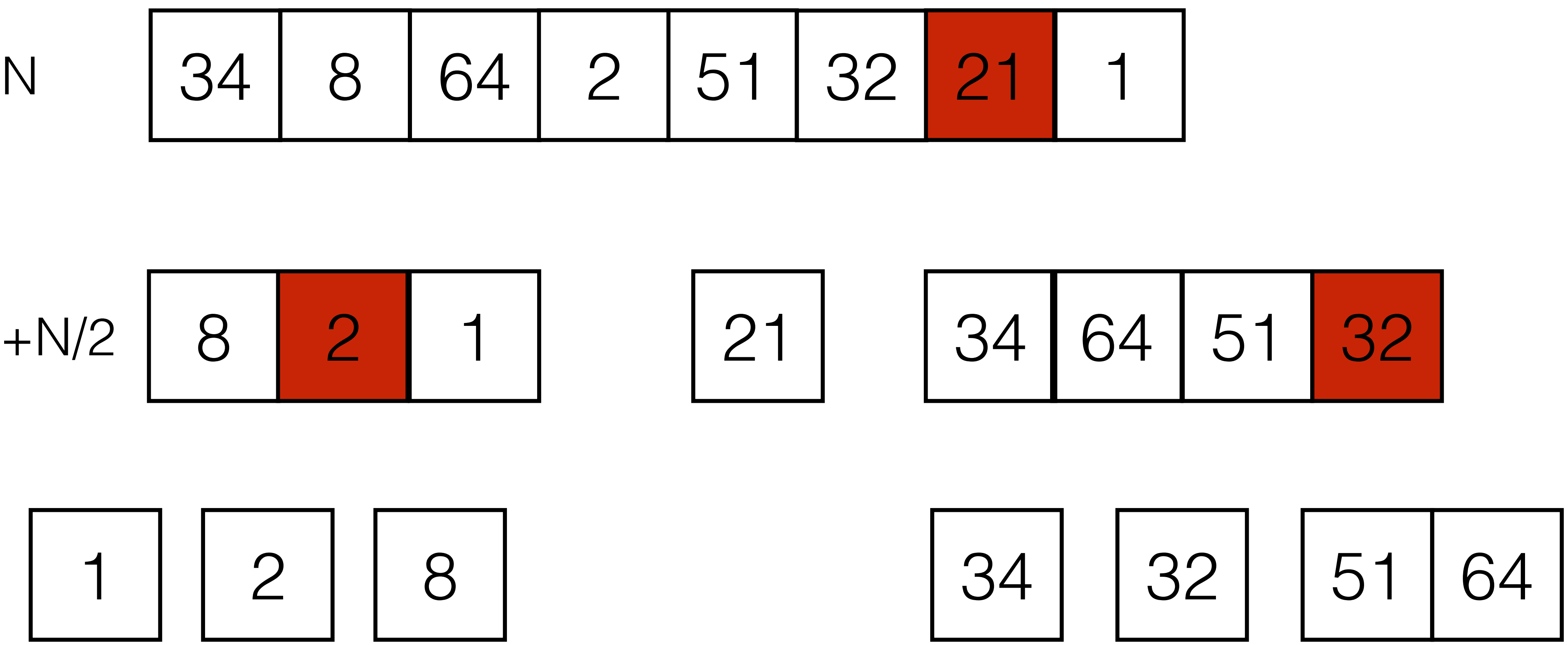
(we ignore the pivot element, so this overestimates the running time slightly)

Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

$T(N) = 2\,T(N/2) + N$

$T(N/2) = 2\,T(N/4) + N/2$



(we ignore the pivot element, so this overestimates the running time slightly)

Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

$T(N) = 2\,T(N/2) + N$

34	8	64	2	51	32	21	1

$T(N/2) = 2\,T(N/4) + N/2$

8	2	1

21

34	64	51	32

⋮

$T(1) = 1$

1	2	8	34	32	51	64

(we ignore the pivot element, so this overestimates the running time slightly)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

(note that this is the same analysis as for Merge Sort)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

(note that this is the same analysis as for Merge Sort)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

(note that this is the same analysis as for Merge Sort)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

(note that this is the same analysis as for Merge Sort)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

$$= N \cdot T(1) + logN \cdot N$$

(note that this is the same analysis as for Merge Sort)

Quick Sort: Best Case

$$T(N) = 2 \cdot T(\frac{N}{2}) + N$$

$$= 2 \cdot (2 \cdot T(\frac{N}{4}) + \frac{N}{2}) + N \qquad = 4 \cdot T(\frac{N}{4}) + N + N$$

$$= 2^k \cdot T(\frac{N}{2^k}) + k \cdot N$$

assume $k = \log N$

$$= N \cdot T(1) + logN \cdot N$$

$$= N + N \cdot \log N = \Theta(N \log N)$$

(note that this is the same analysis as for Merge Sort)

Choosing the Pivot

Choosing the Pivot

- Ideally we want to choose the median in each partition, but we don't know where it is!

Choosing the Pivot

- Ideally we want to choose the median in each partition, but we don’t know where it is!

- Computing the pivot should be a constant time operation.

Choosing the Pivot

- Ideally we want to choose the median in each partition, but we don't know where it is!
- Computing the pivot should be a constant time operation.
- Choosing the element at the beginning/end/middle is a terrible idea!
  Better: Choose a random element.

Choosing the Pivot

- Ideally we want to choose the median in each partition, but we don't know where it is!
- Computing the pivot should be a constant time operation.
- Choosing the element at the beginning/end/middle is a terrible idea!
  Better: Choose a random element.
- Good approximation for median: *"Median-of-three"*

Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].

34	8	64	2	51	32	21	1

Choosing a Pivot: Median of Three

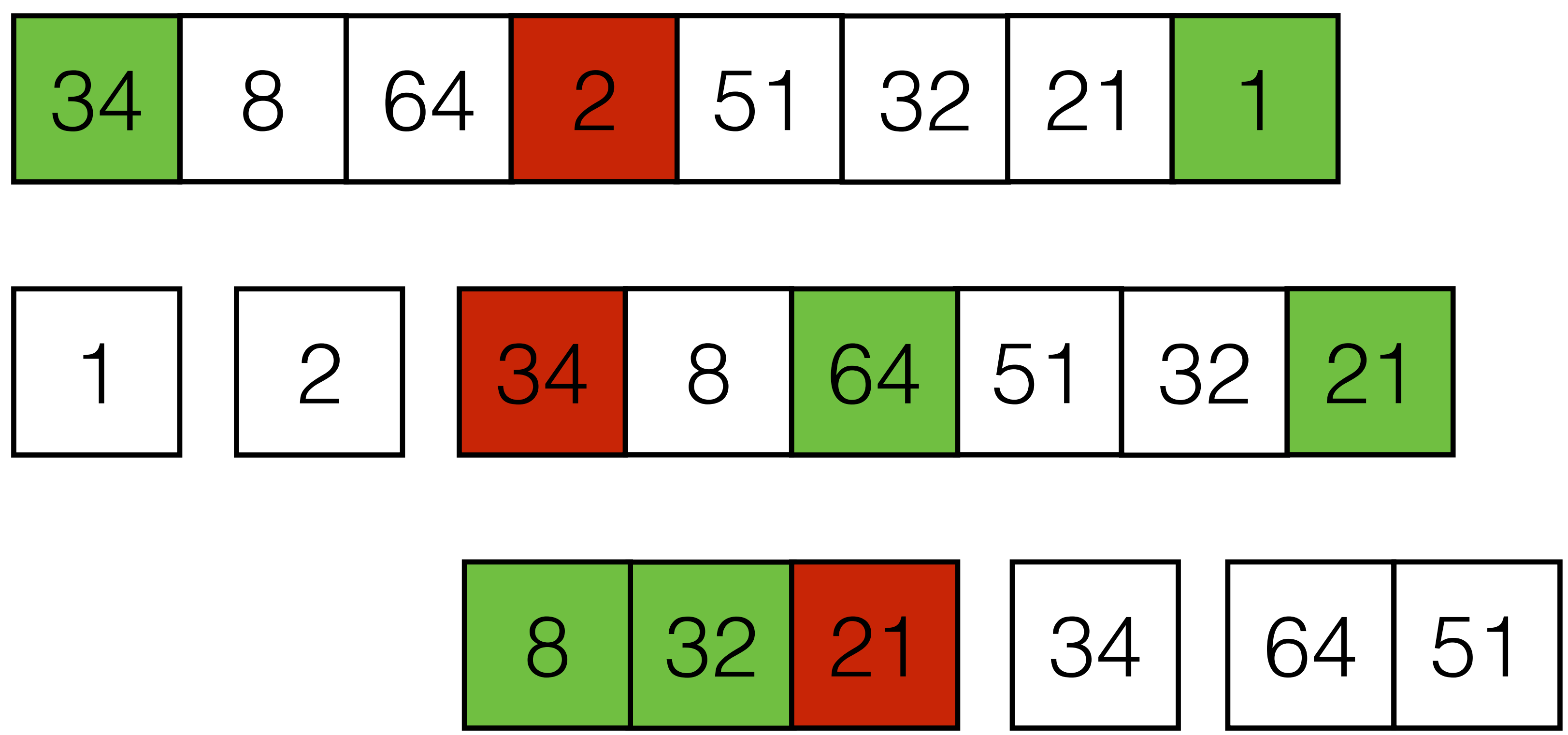
Choose the median of array[0], array[n]m and array[n/2].

34	8	64	2	51	32	21	1

1	2	34	8	64	51	32	21		

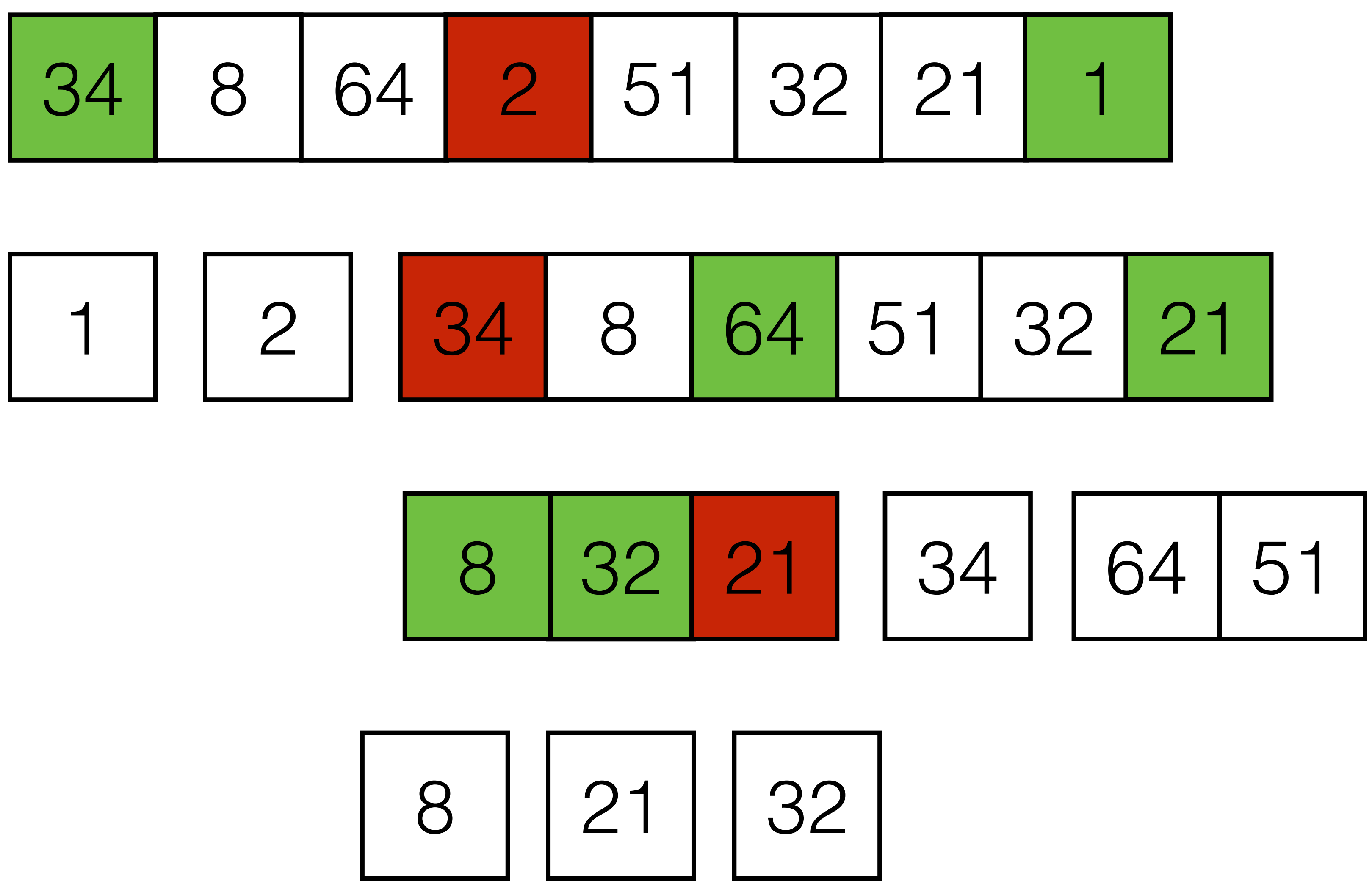
Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].



Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].



Median of Three

```
public static int find_pivot_index(Integer[] a, int left, int right) {
    int center = ( left + right ) / 2;
    Integer tmp;
    if (a[center] < a[left]) {
        tmp = a[center]; a[center] = a[left]; a[left] = tmp;}
    if (a[right] < a[left]) {
        tmp = a[right]; a[right] = a[left]; a[left] = tmp;}
    if (a[right] < a[center]) {
        tmp = a[right]; a[right] = a[center]; a[center] = tmp;}
    return center;
}
```

Analyzing Quick Sort

- Worst case running time: $\Theta(N^2)$
- Best and average case (random pivot): $\Theta(N \log N)$
- Is QuickSort stable?
- Space requirement?

Analyzing Quick Sort

- Worst case running time: $\Theta(N^2)$
- Best and average case (random pivot): $\Theta(N \log N)$
- Is QuickSort stable?

  No. Partitioning can change order of elements.
- Space requirement?

Analyzing Quick Sort

- Worst case running time: $\Theta(N^2)$

- Best and average case (random pivot): $\Theta(N \log N)$

- Is QuickSort stable?

  No. Partitioning can change order of elements.

- Space requirement?

  In-place O(1), but the method activation stack grows with the running time. O(N)

Comparison-Based Sorting Algorithms

	T_{Worst}	T_{Best}	T_{Avg}	Space	Stable?
Insertion Sort	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N^2)$	$O(1)$	✓
Heap Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(1)$	✗
Merge Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(N)$	✓
Quick Sort	$\Theta(N^2)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(1)$	✗

gray entries: not shown in class

Comparison-Based Sorting Algorithms

	T_{Worst}	T_{Best}	T_{Avg}	Space	Stable?
Insertion Sort	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N^2)$	$O(1)$	✓
Heap Sort	Θ				✗
Merge Sort	Θ				✓
Quick Sort	$\Theta(N^2)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(1)$	✗

$\Omega(N \log N)$ worst case lower bound on comparison based general sorting!
Can we do better if we make some assumptions?

gray entries: not shown in class