Data Structures in Java

Lecture 15: Sorting II

11/11/2015

Daniel Bauer

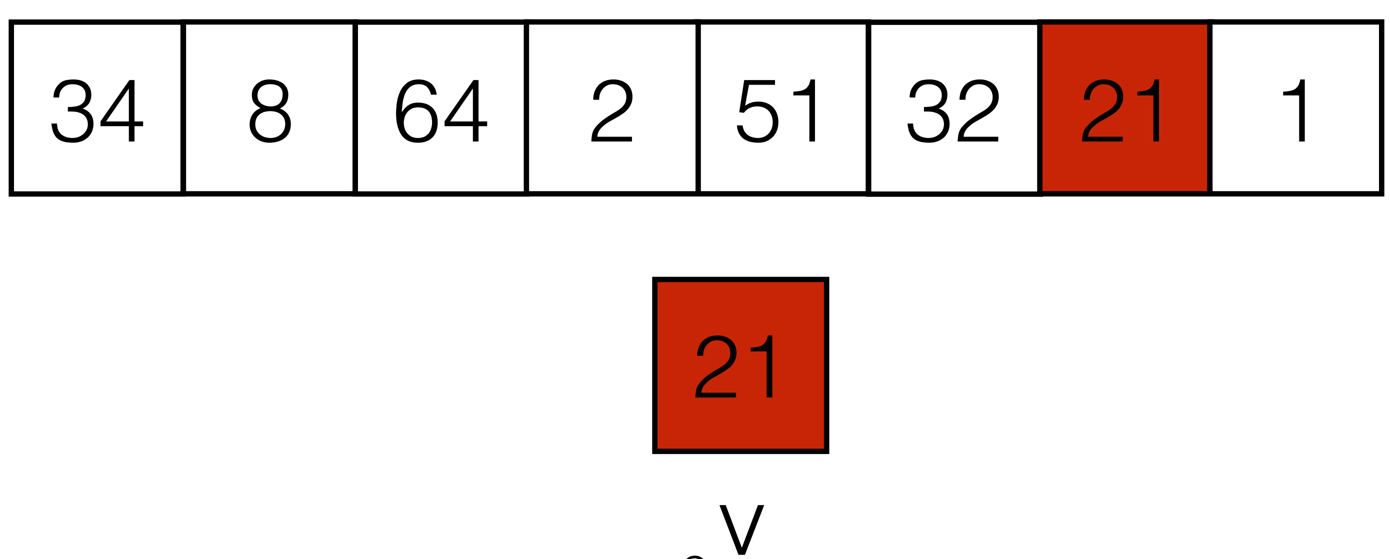
Quick Sort

- 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

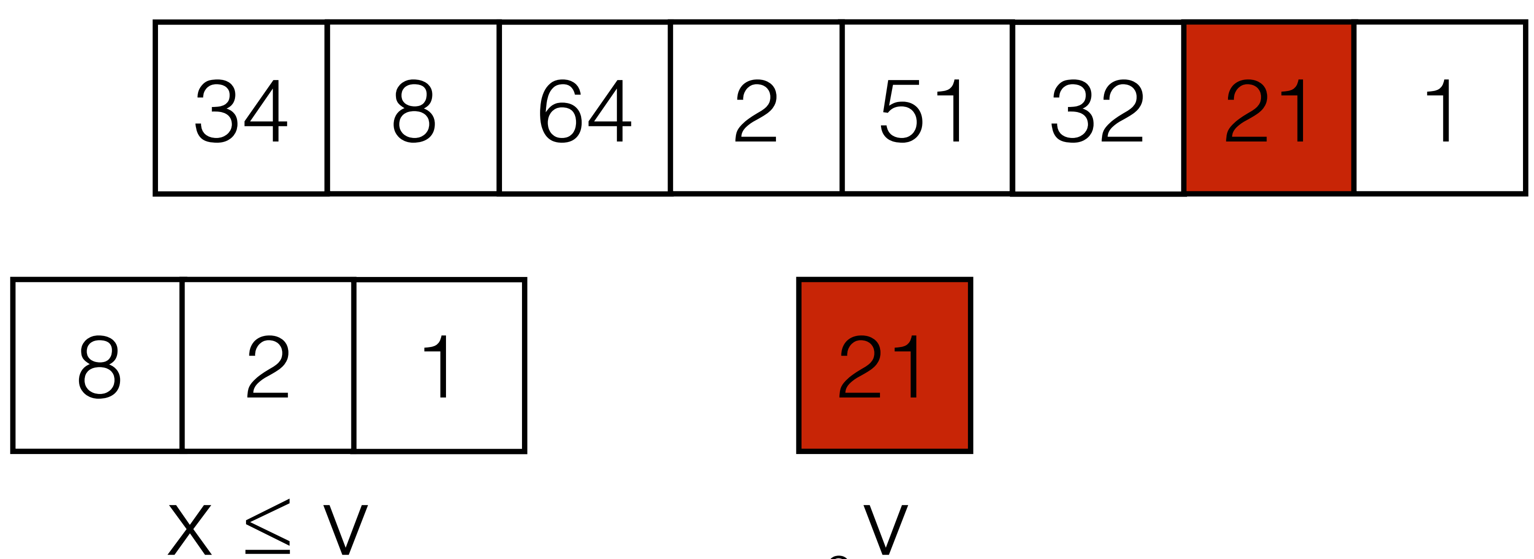
Quick Sort

- 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.



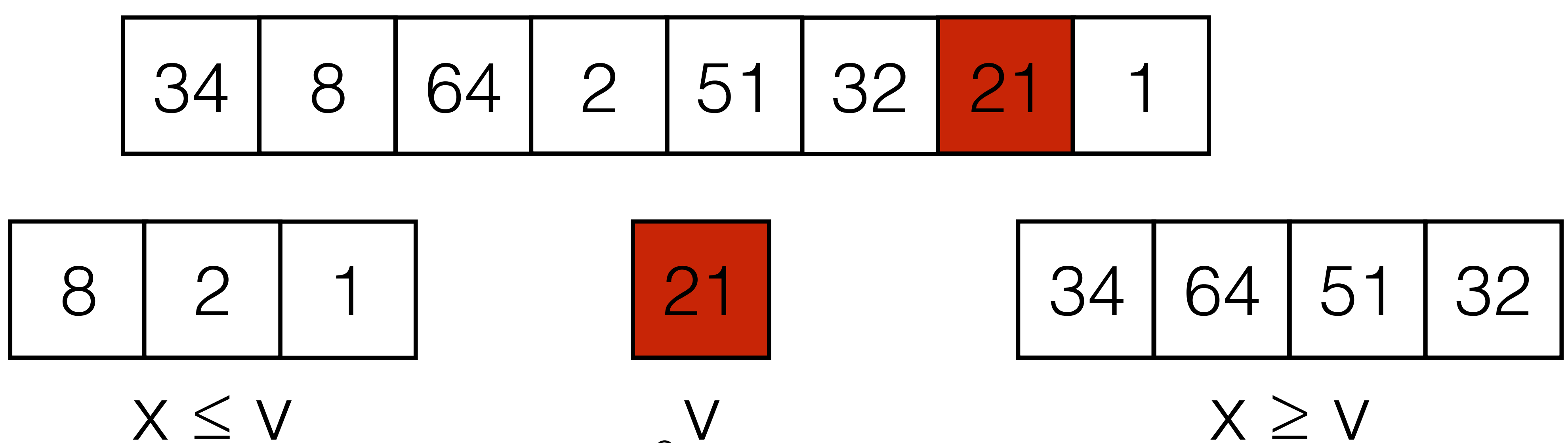
Quick Sort

- 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.



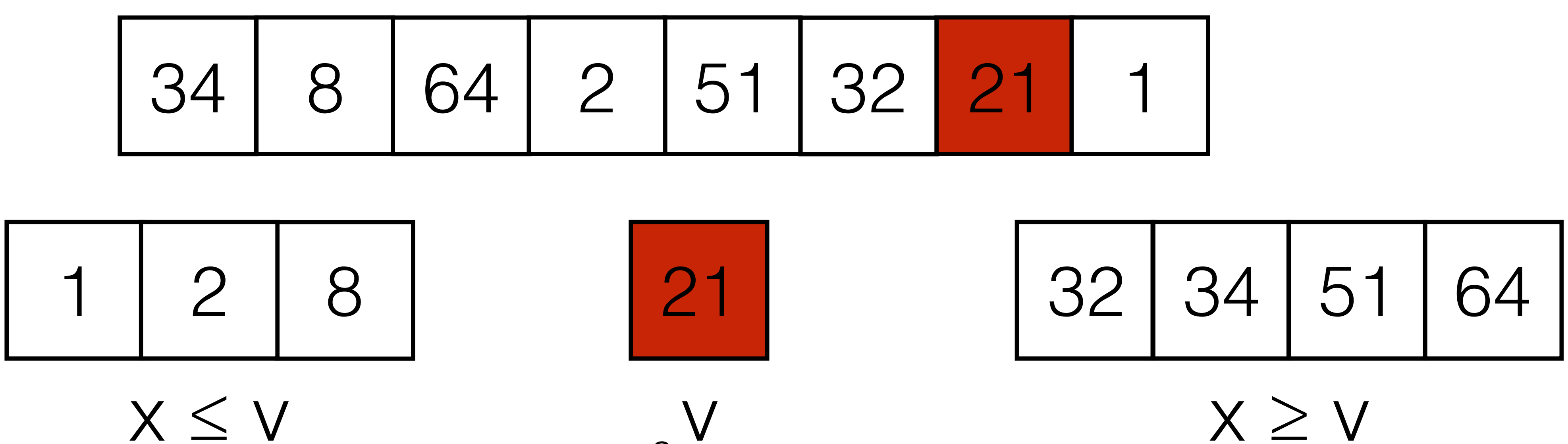
Quick Sort

- 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.



Quick Sort

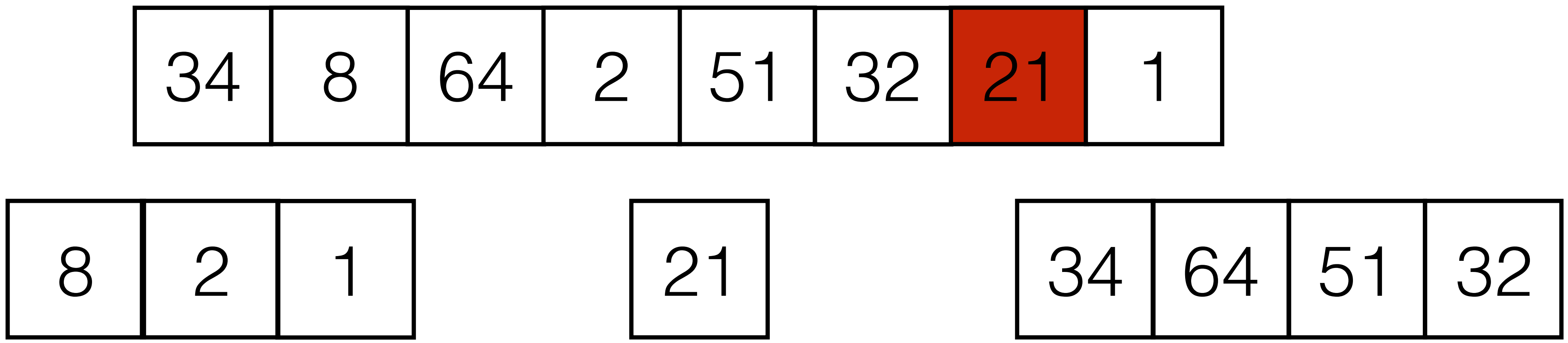
- Another divide-and-conquer algorithm.
- Pick any **pivot** element v.
- Partition the array into elements
    - x ≤ v and x ≥ v.
- Recursively sort the partitions, then concatenate them.

34	8	64	2	51	32	21	1

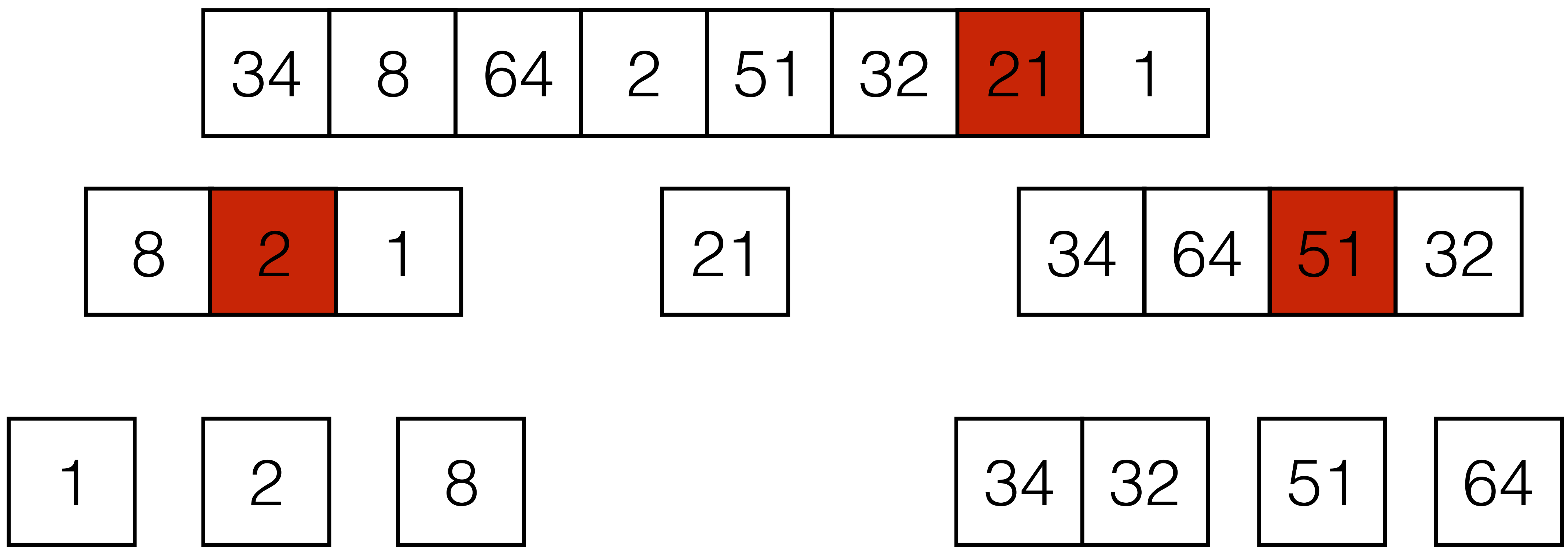
1	2	8	21	32	34	51	64

x ≤ v v x ≥ v

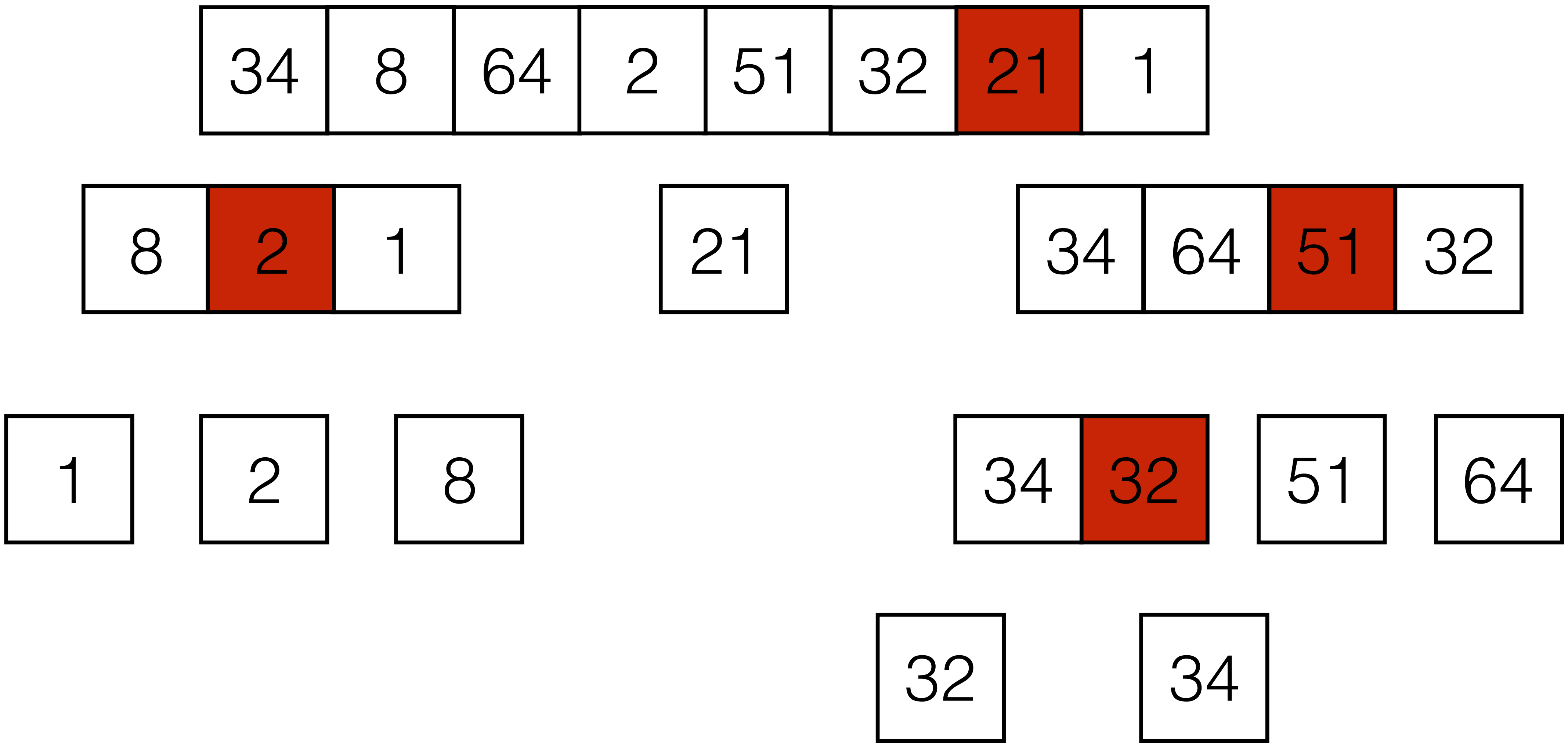
Quick Sort



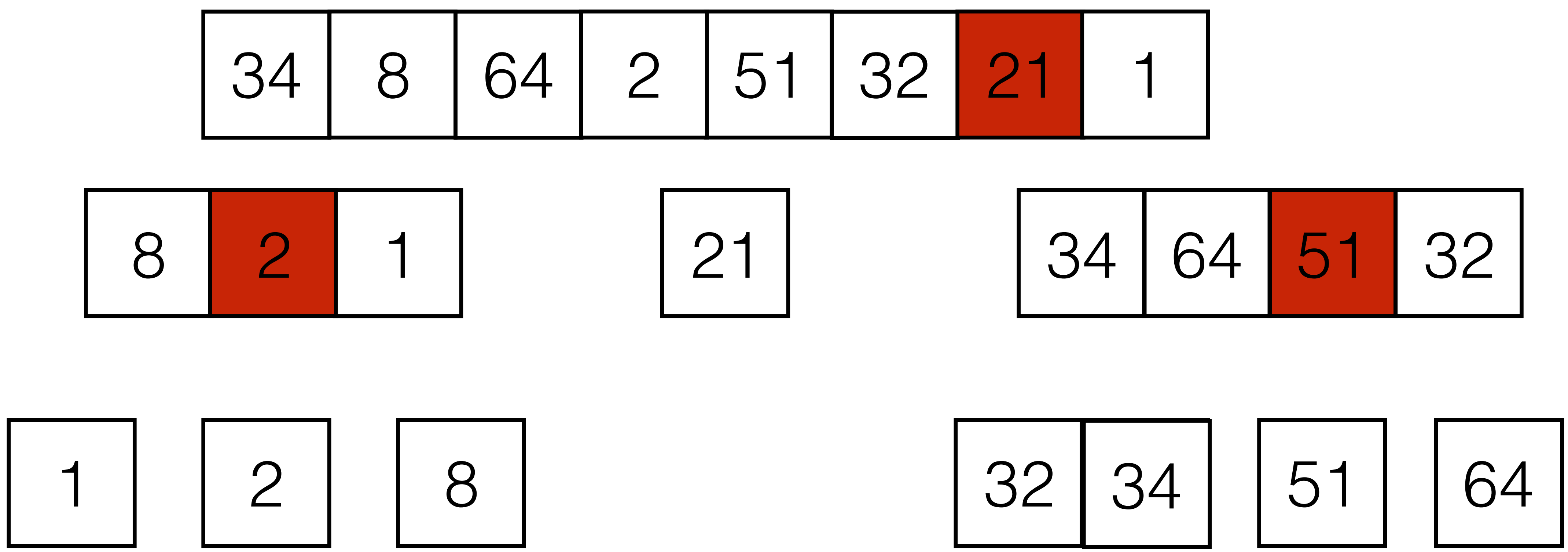
Quick Sort



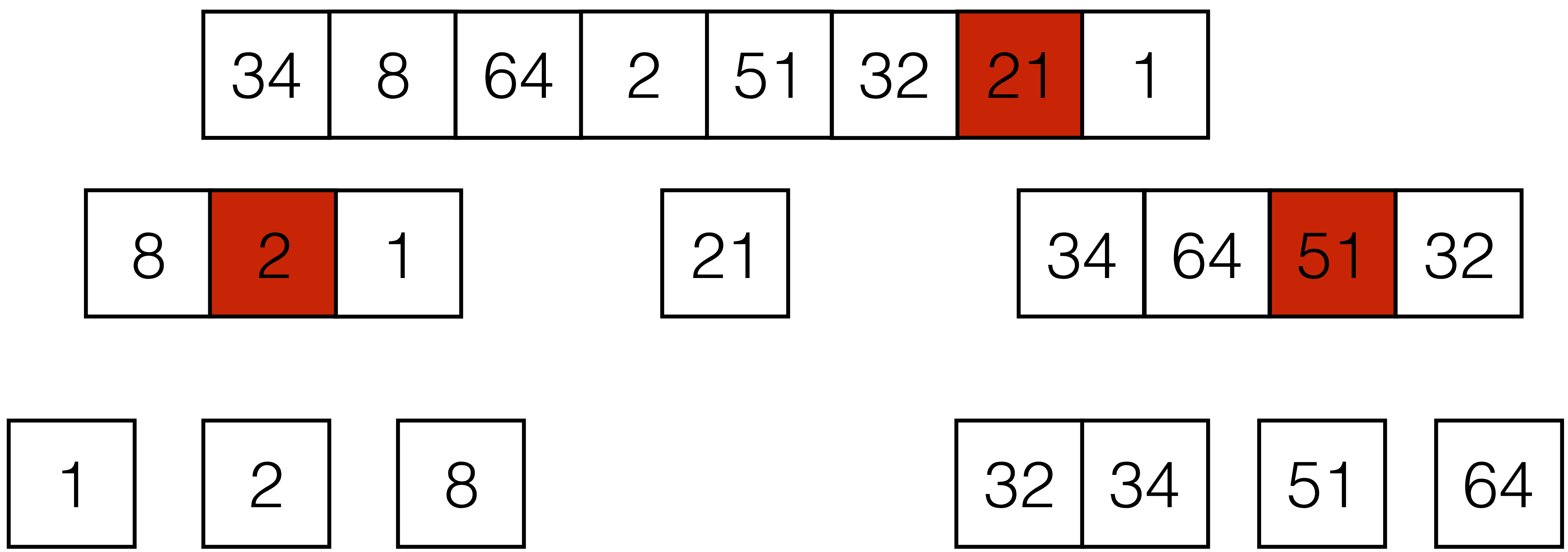
Quick Sort



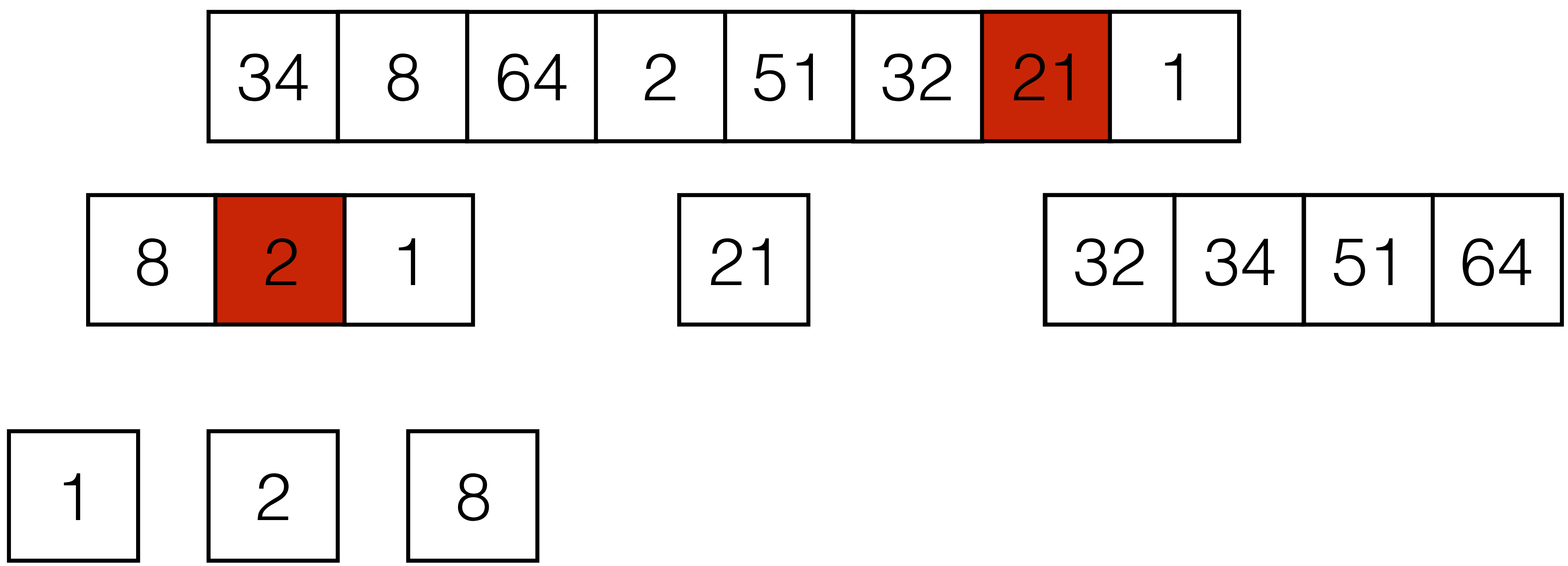
Quick Sort



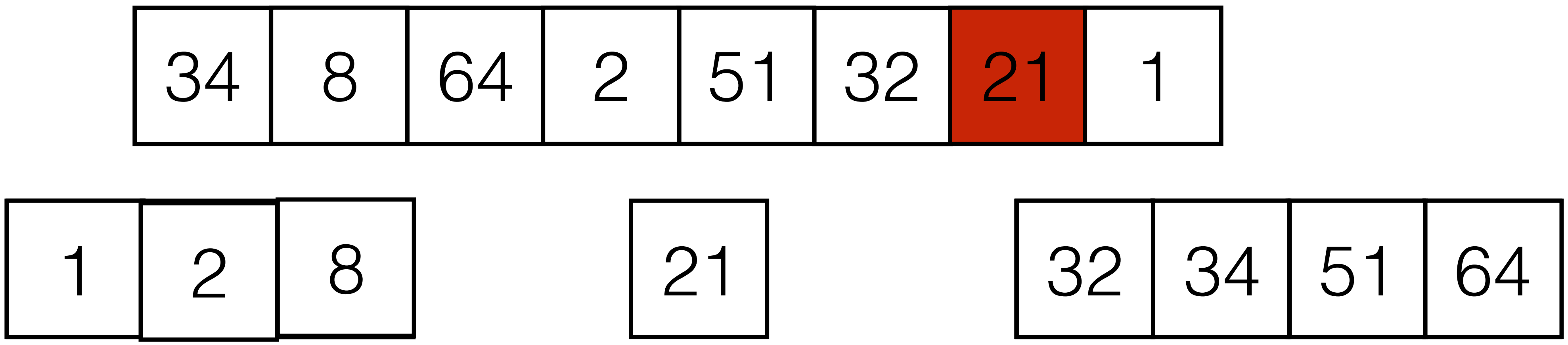
Quick Sort



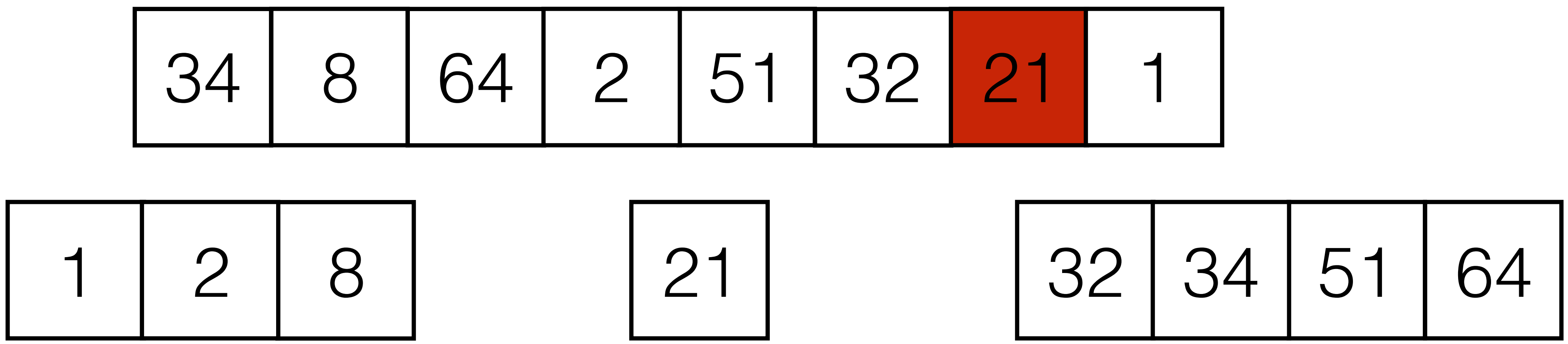
Quick Sort



Quick Sort



Quick Sort



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 ≤ v to the left and elements x ≥ 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.

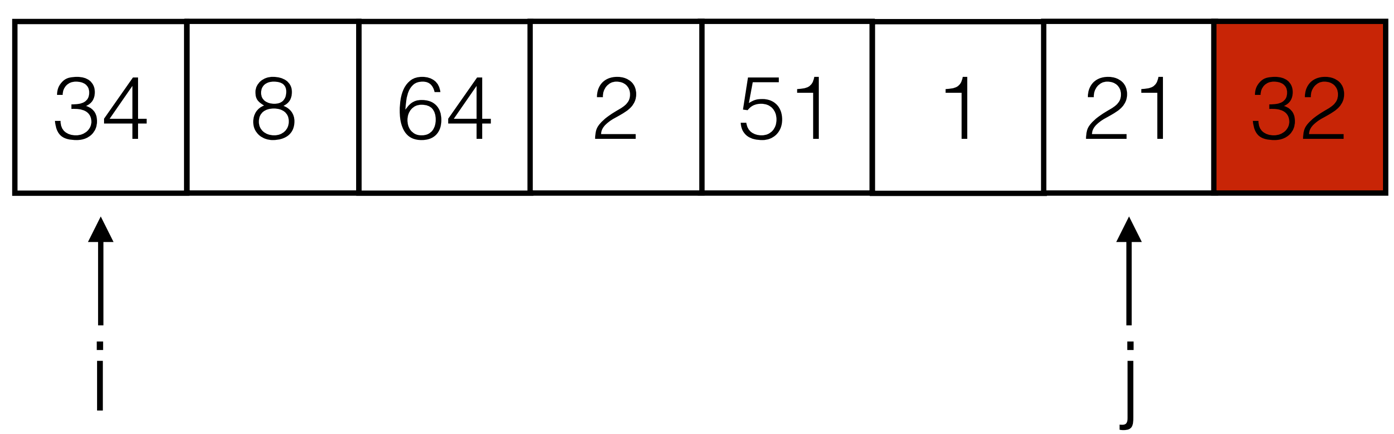
- Use swaps to push all elements $x \leq v$ to the left and elements $x \geq v$ to the right.

34	8	64	2	51	1	21	32

Move the pivot to the end.

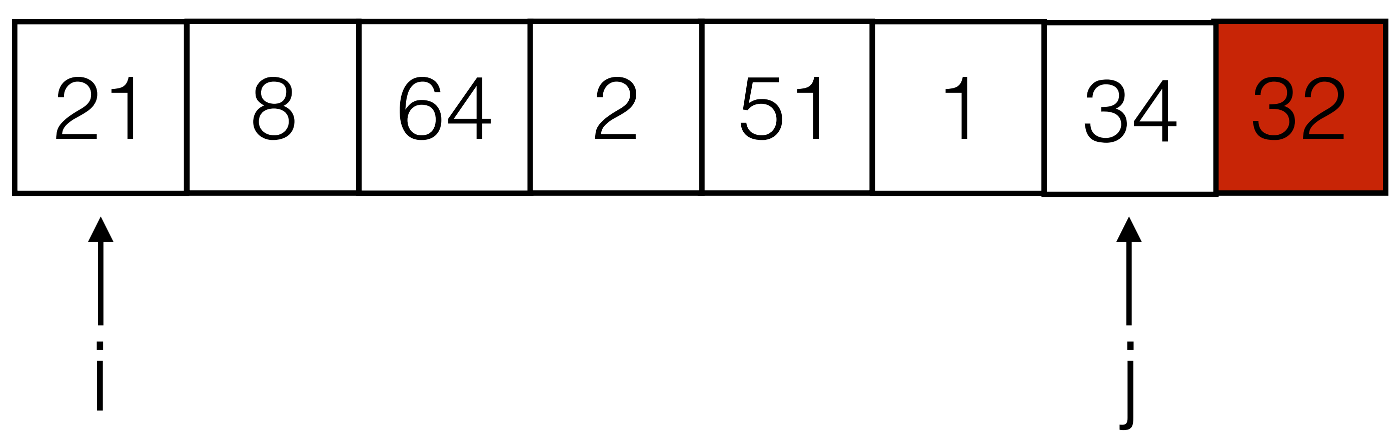
Partitioning the Array

- 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].



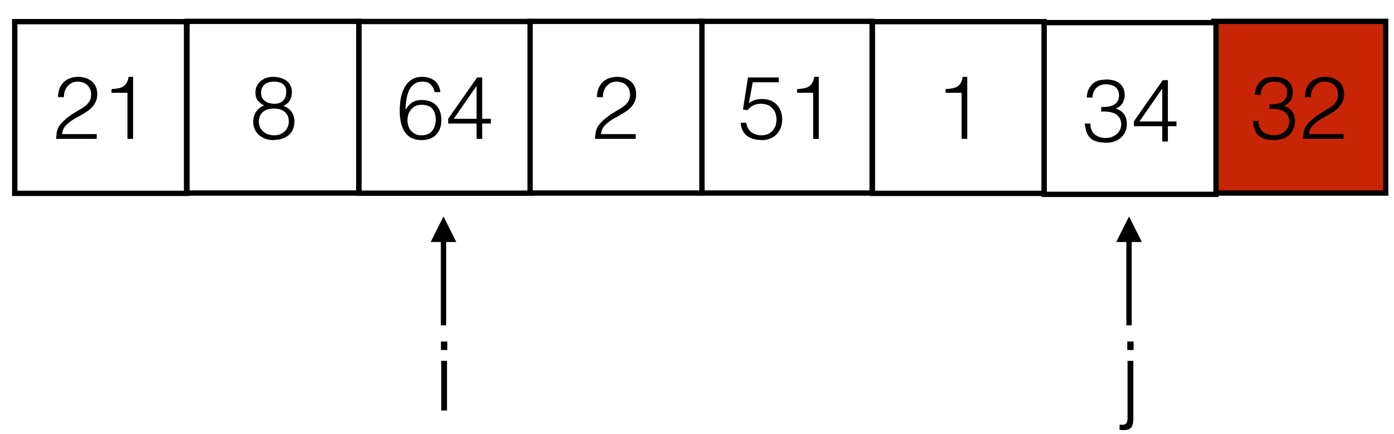
Partitioning the Array

- 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].



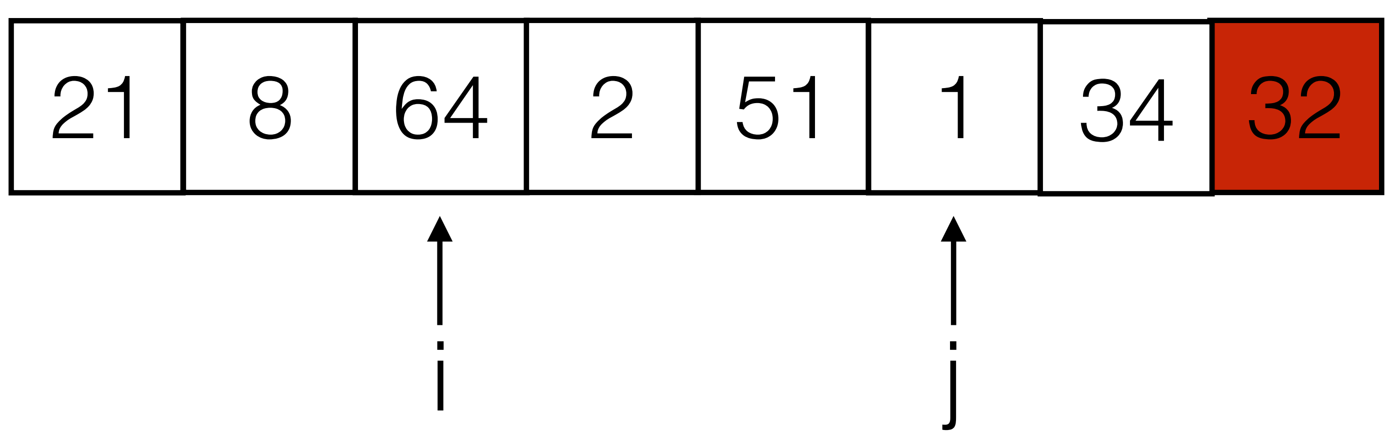
Partitioning the Array

- 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].



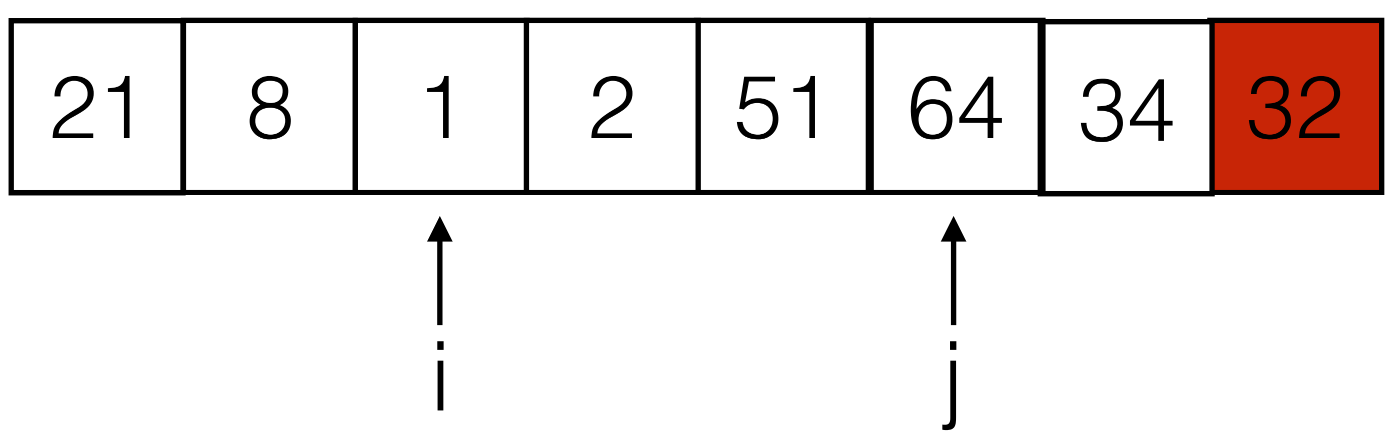
Partitioning the Array

- 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].



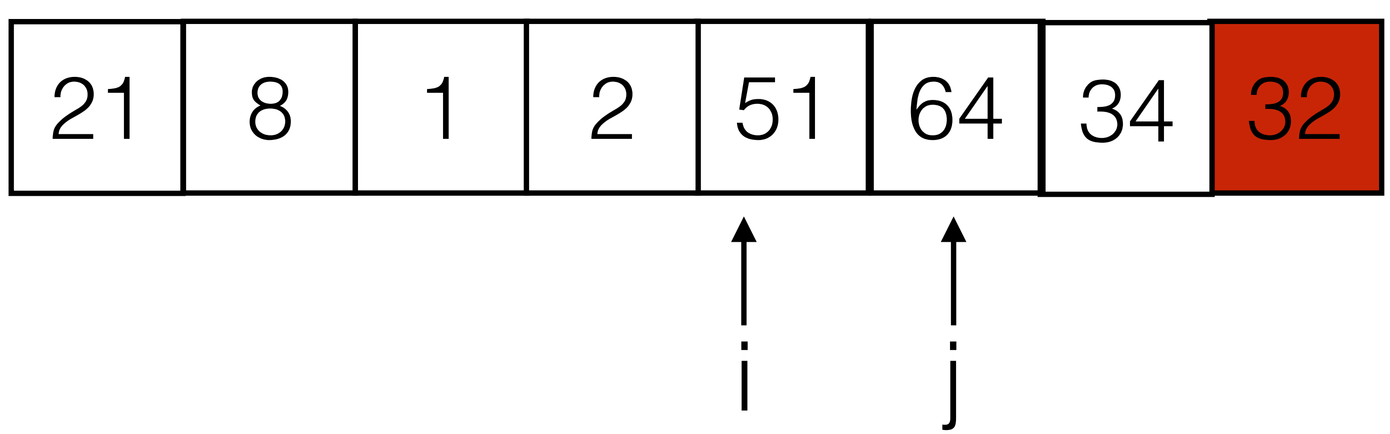
Partitioning the Array

- 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].



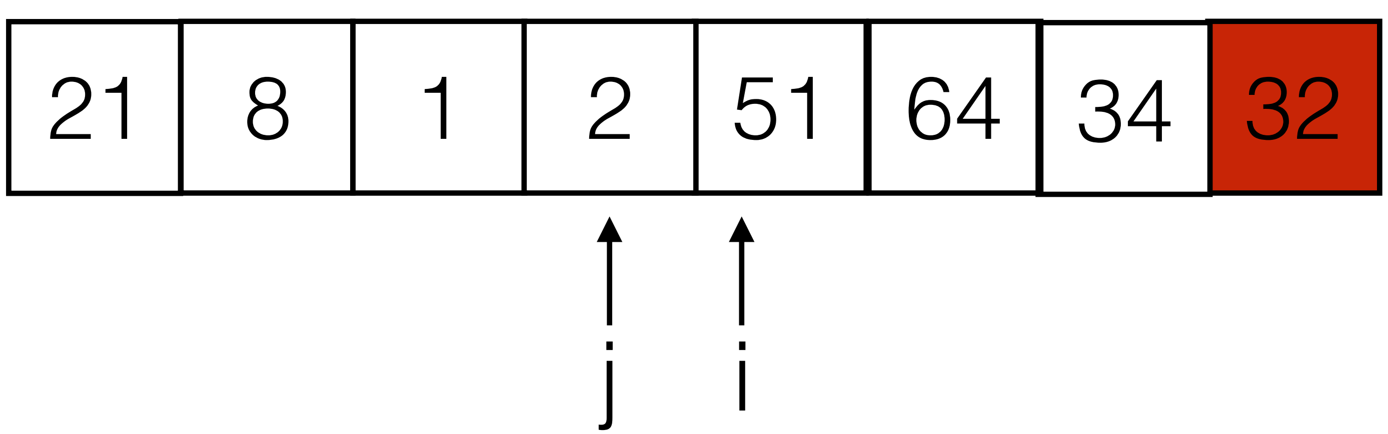
Partitioning the Array

- 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].



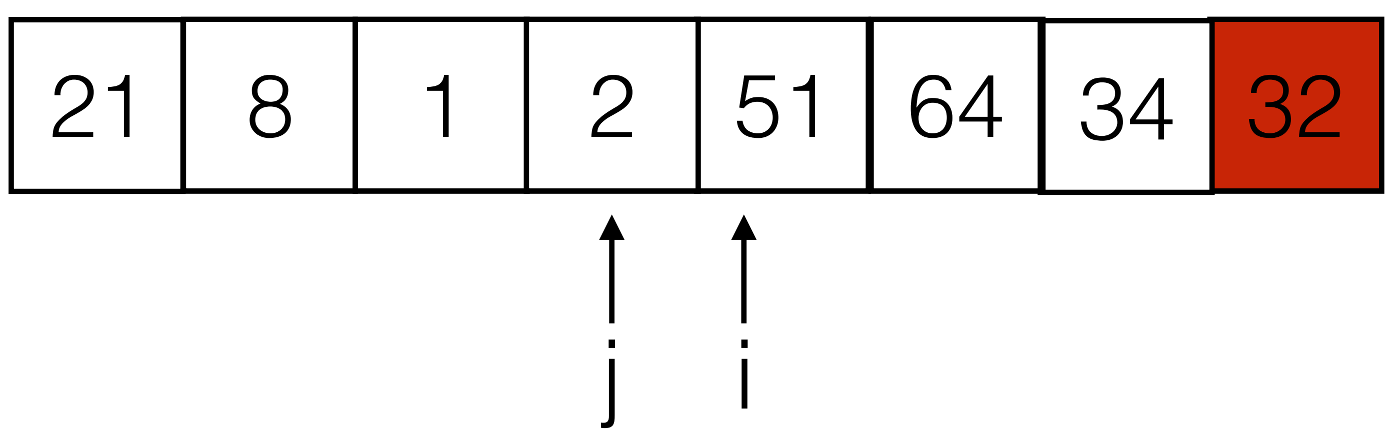
Partitioning the Array

- 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

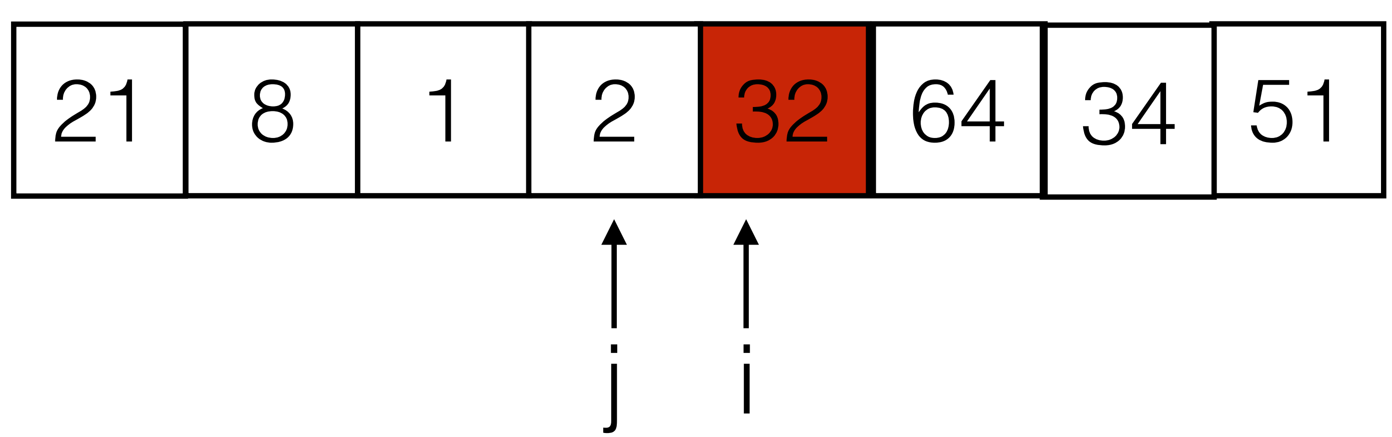
- While True:
  - Move i right until we find an element array[i] ≥ v
  - Move j left until we find an element array[j] ≤ v.
  - if i ≥ j break
  - Swap array[i] and array[j].
- Swap array[i] with v.



- 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 = 0;   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] < v) {};
            while (a[++j] > v) {};
            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 = 0;   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] < v) {};
            while (a[++j] > v) {};
            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

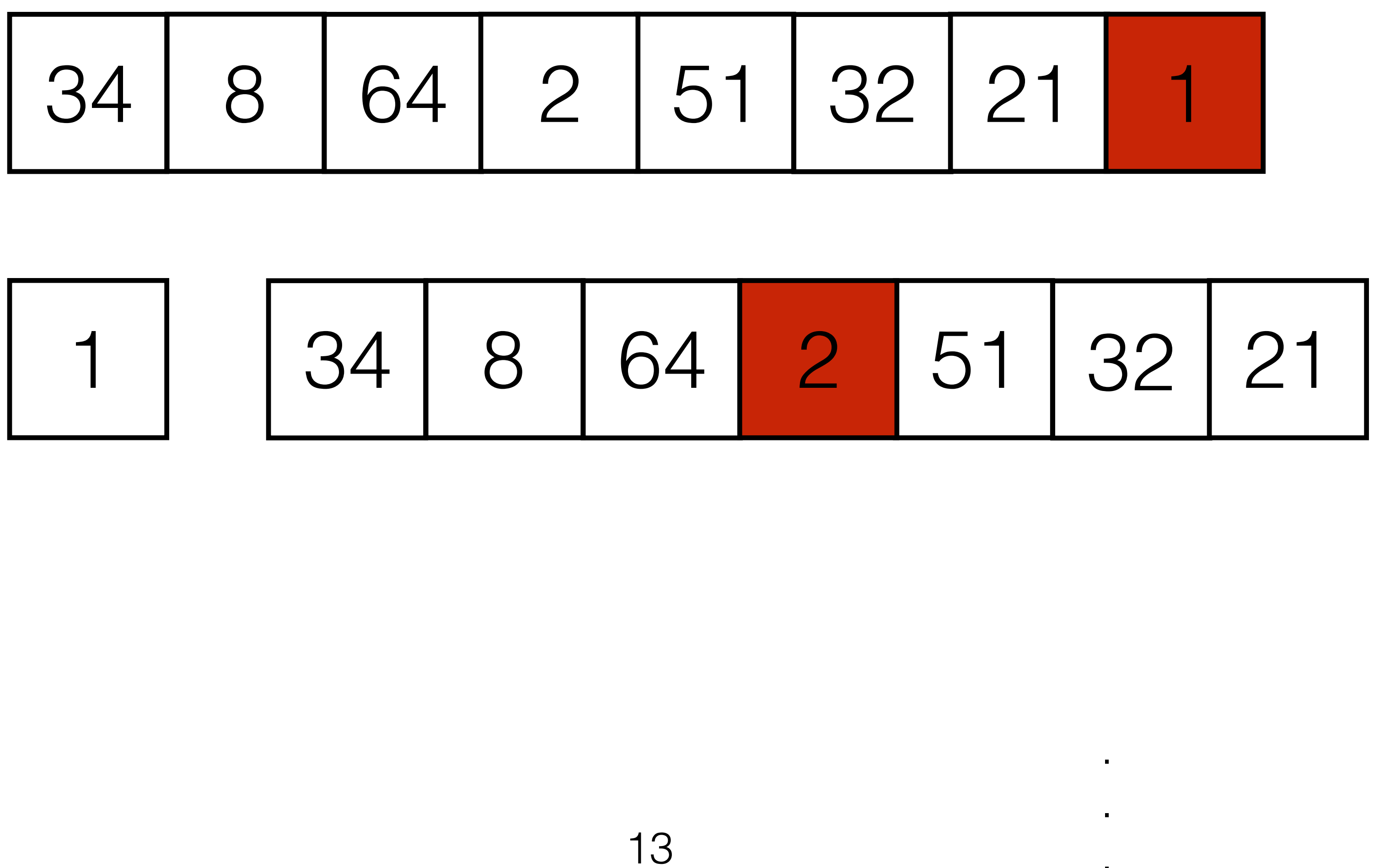
- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!

34	8	64	2	51	32	21	1

.
.
.

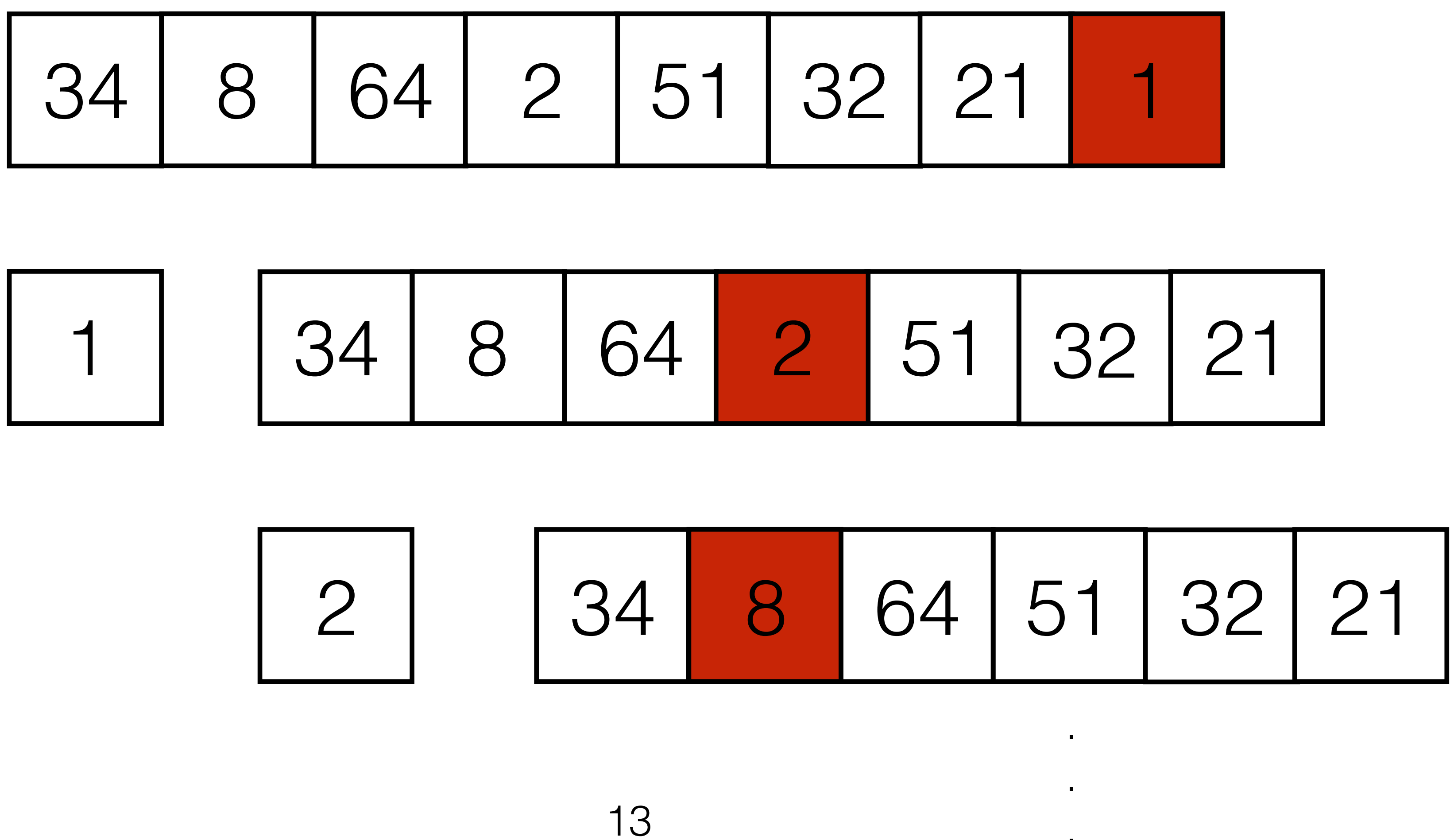
Quick Sort: Worst Case

- 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!

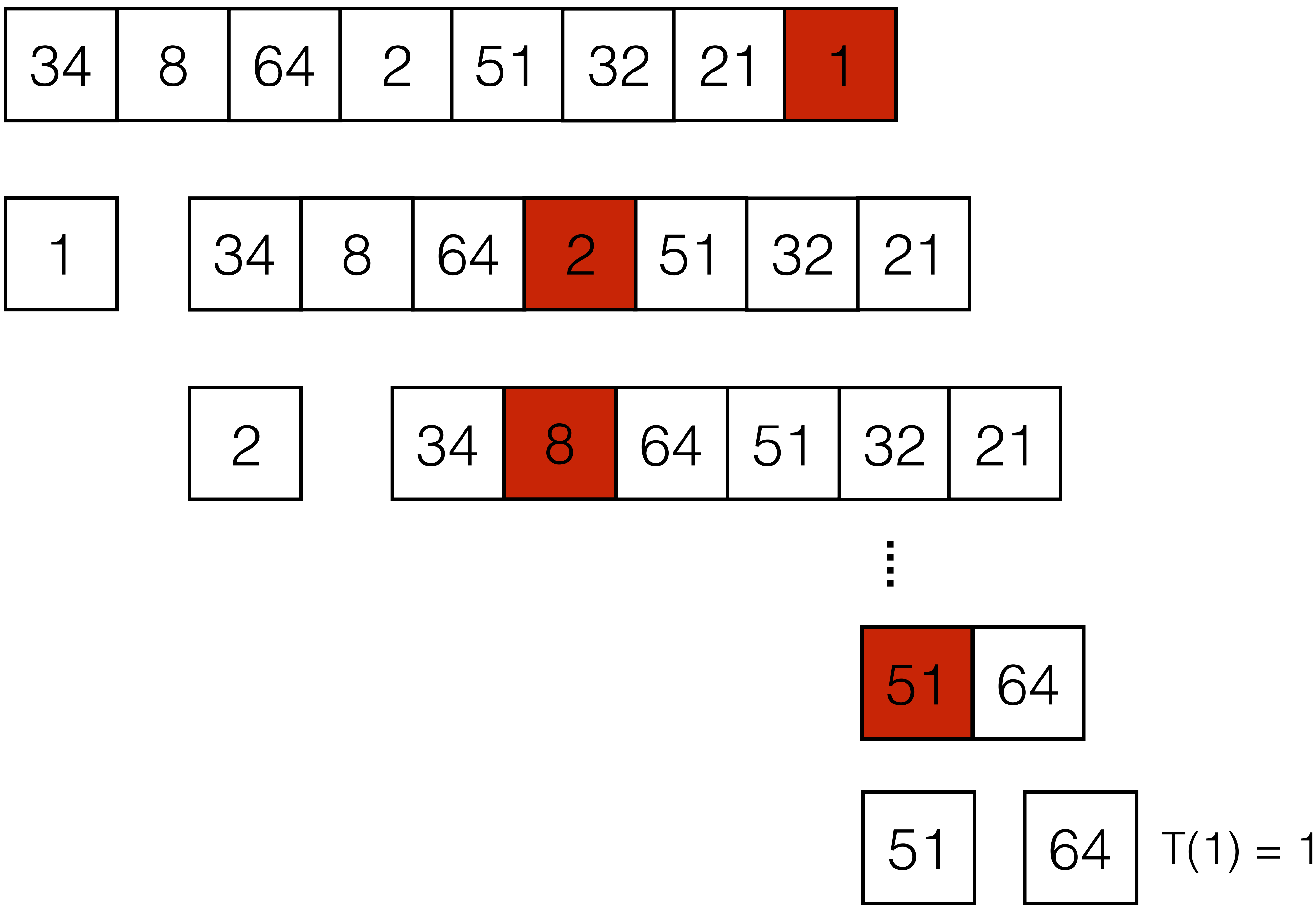


Quick Sort: Worst Case

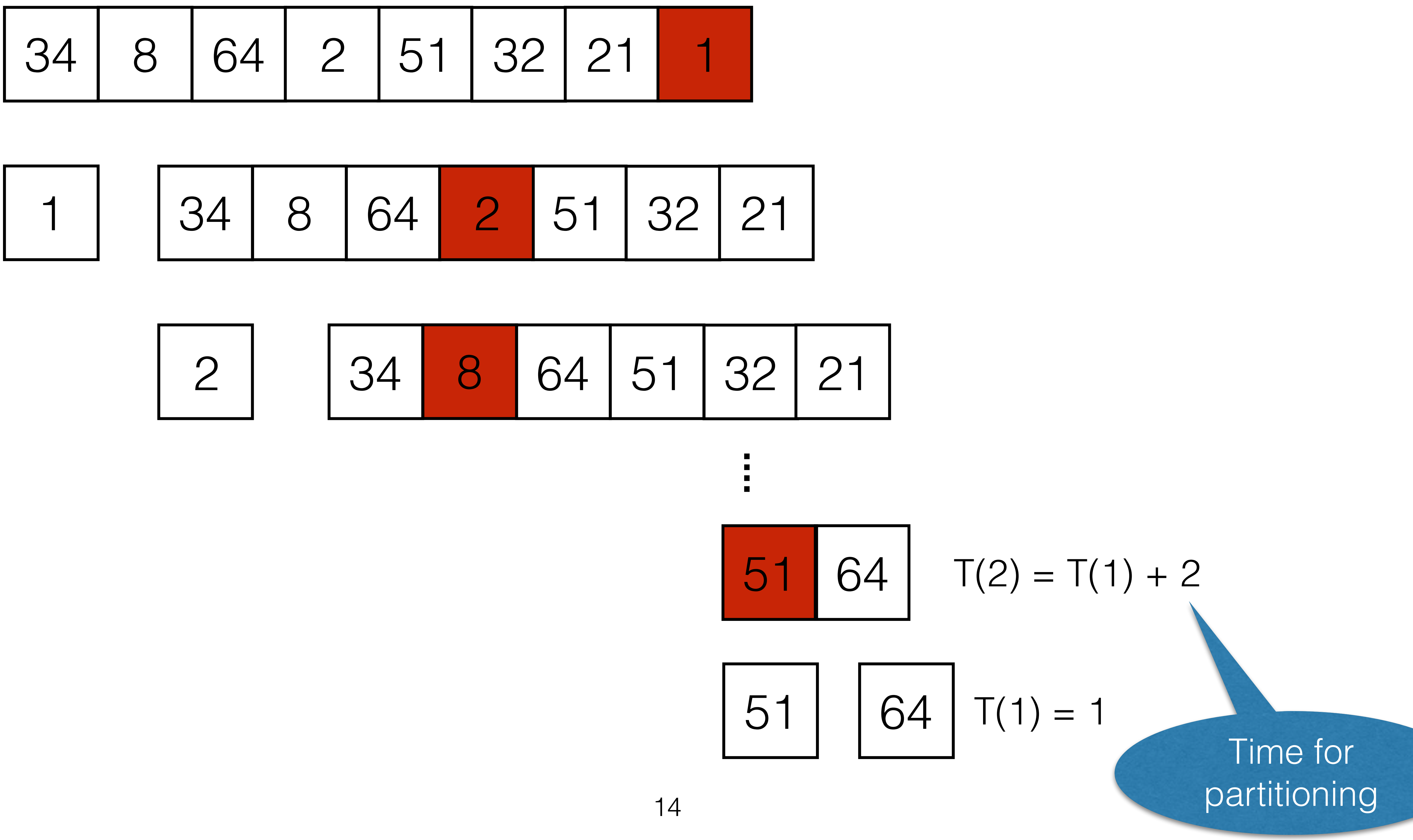
- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!



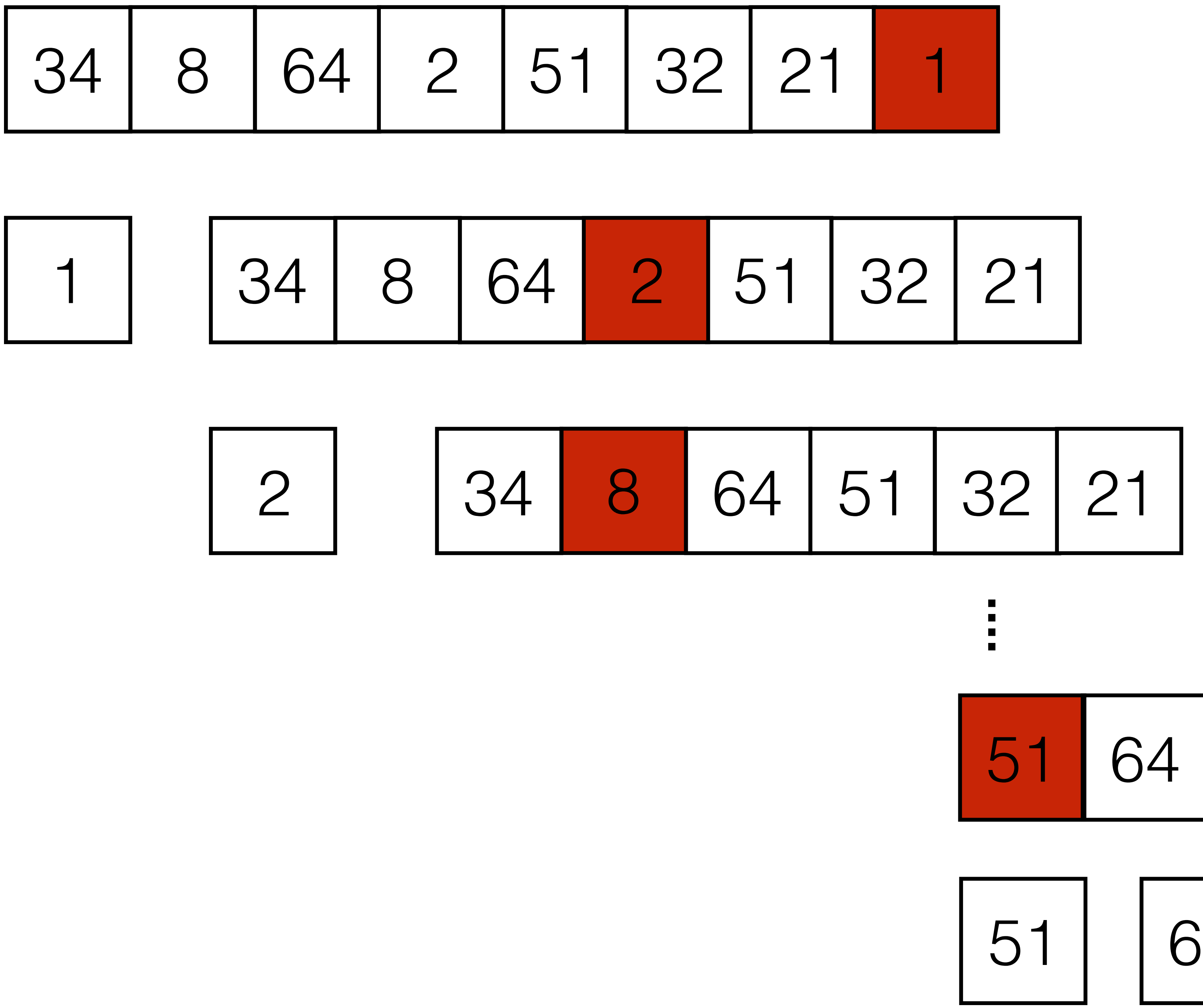
Quick Sort: Worst Case



Quick Sort: Worst Case



Quick Sort: Worst Case



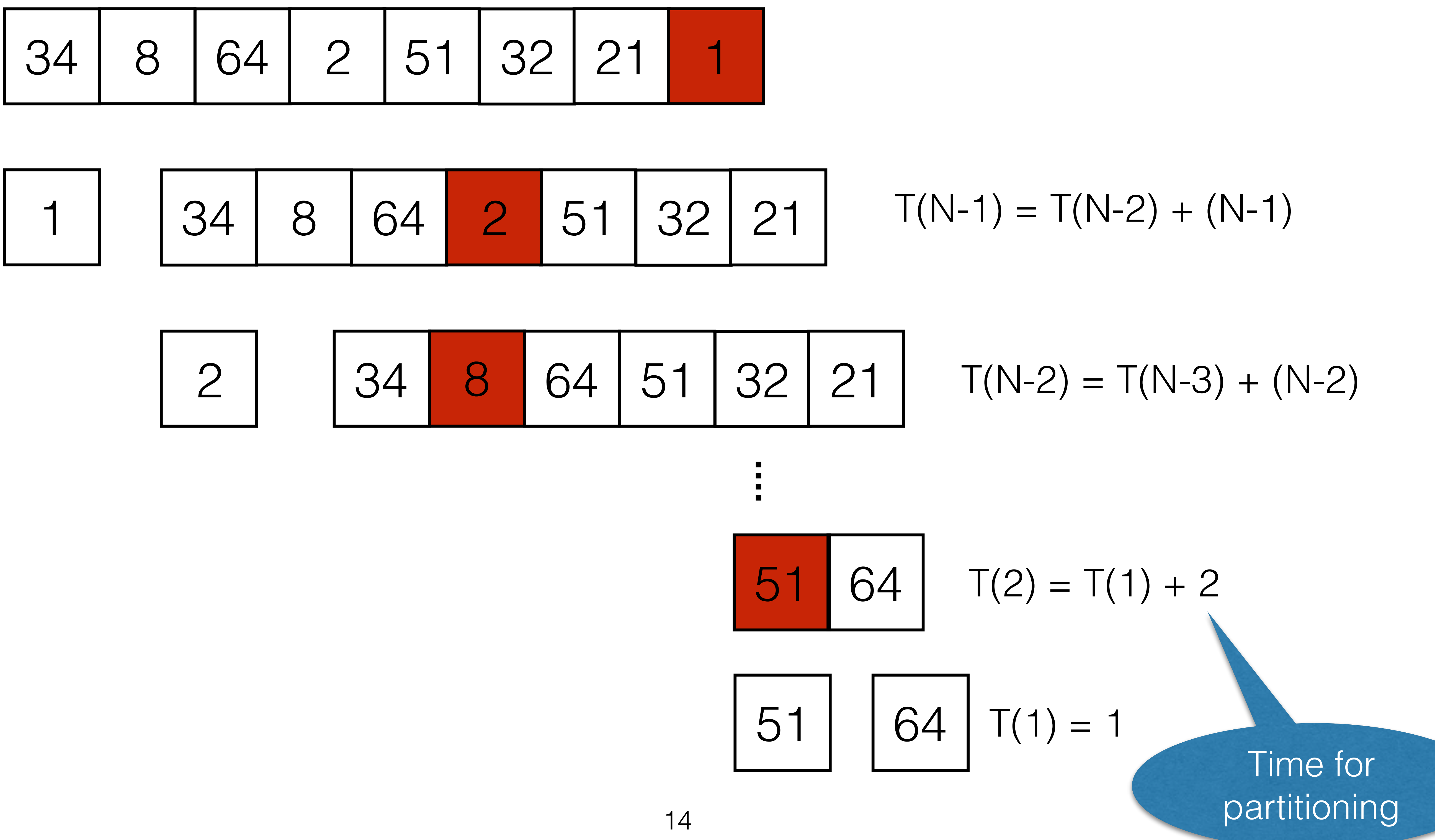
$T(N-2) = T(N-3) + (N-2)$

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

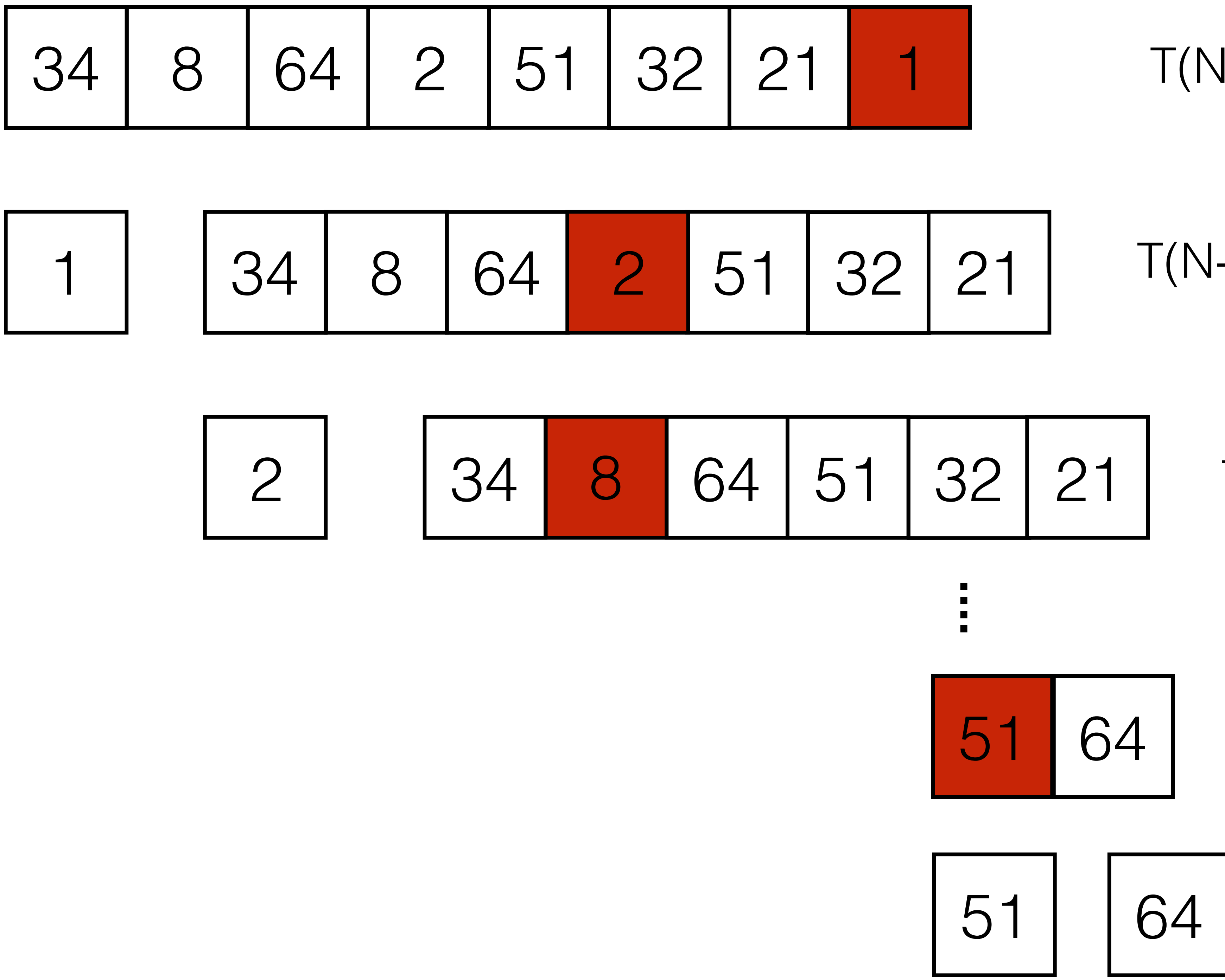
$T(1) = 1$

Time for partitioning

Quick Sort: Worst Case



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

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

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

Quick Sort: Worst Case

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

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

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

$$\vdots$$

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

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

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

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

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

$$\vdots$$

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

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

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

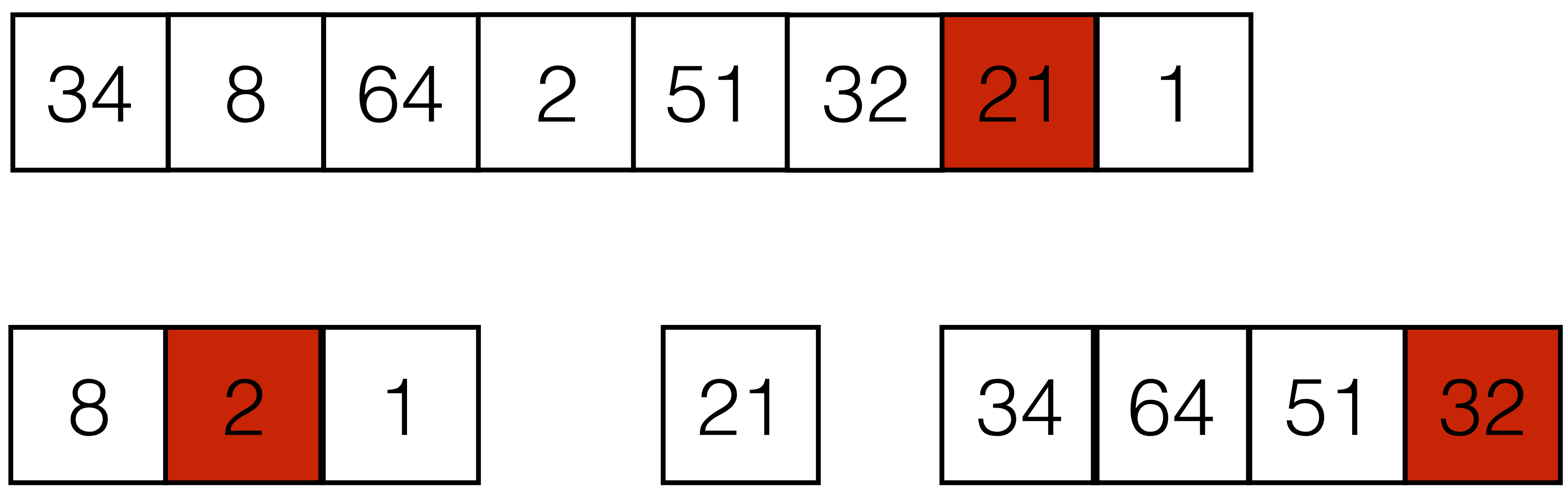
Quick Sort: Best Case

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

34	8	64	2	51	32	21	1

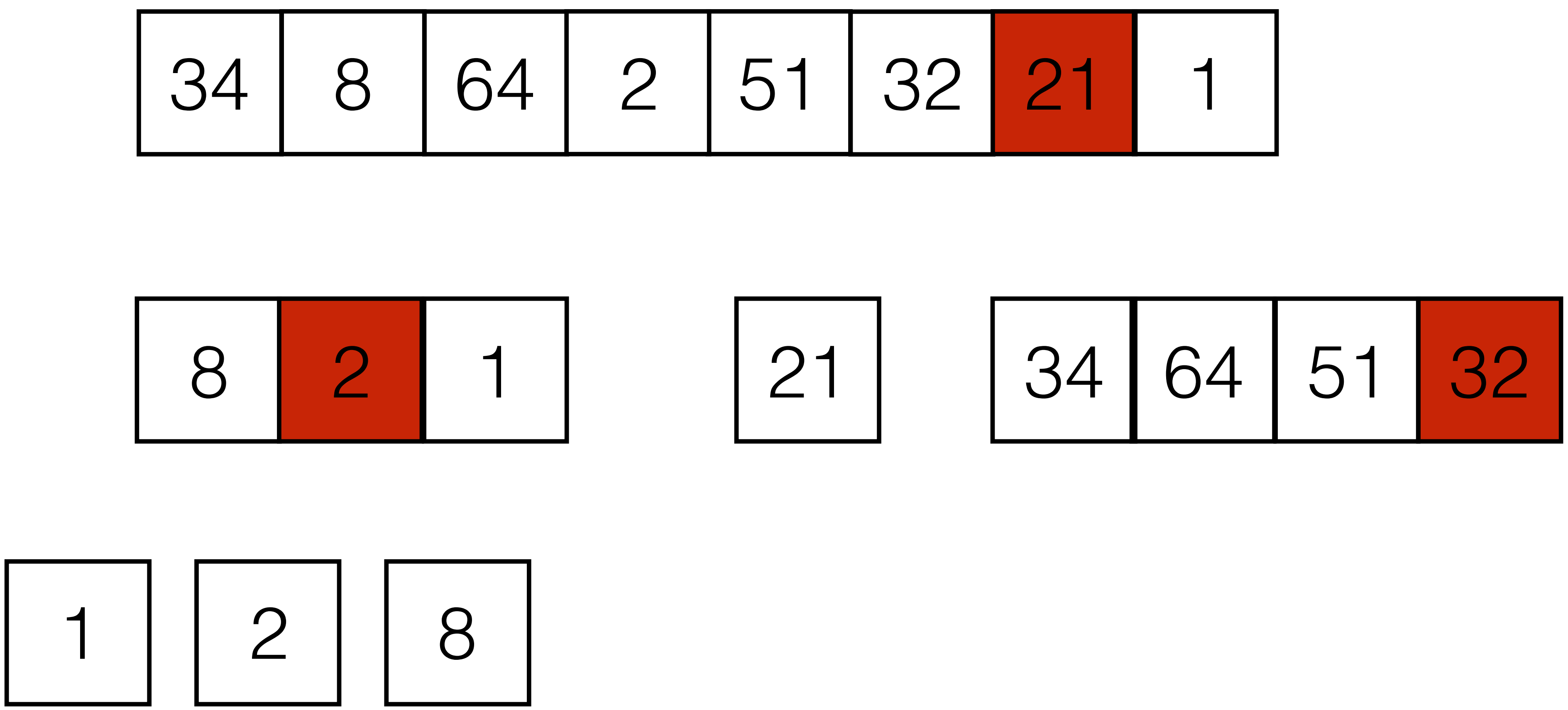
Quick Sort: Best Case

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



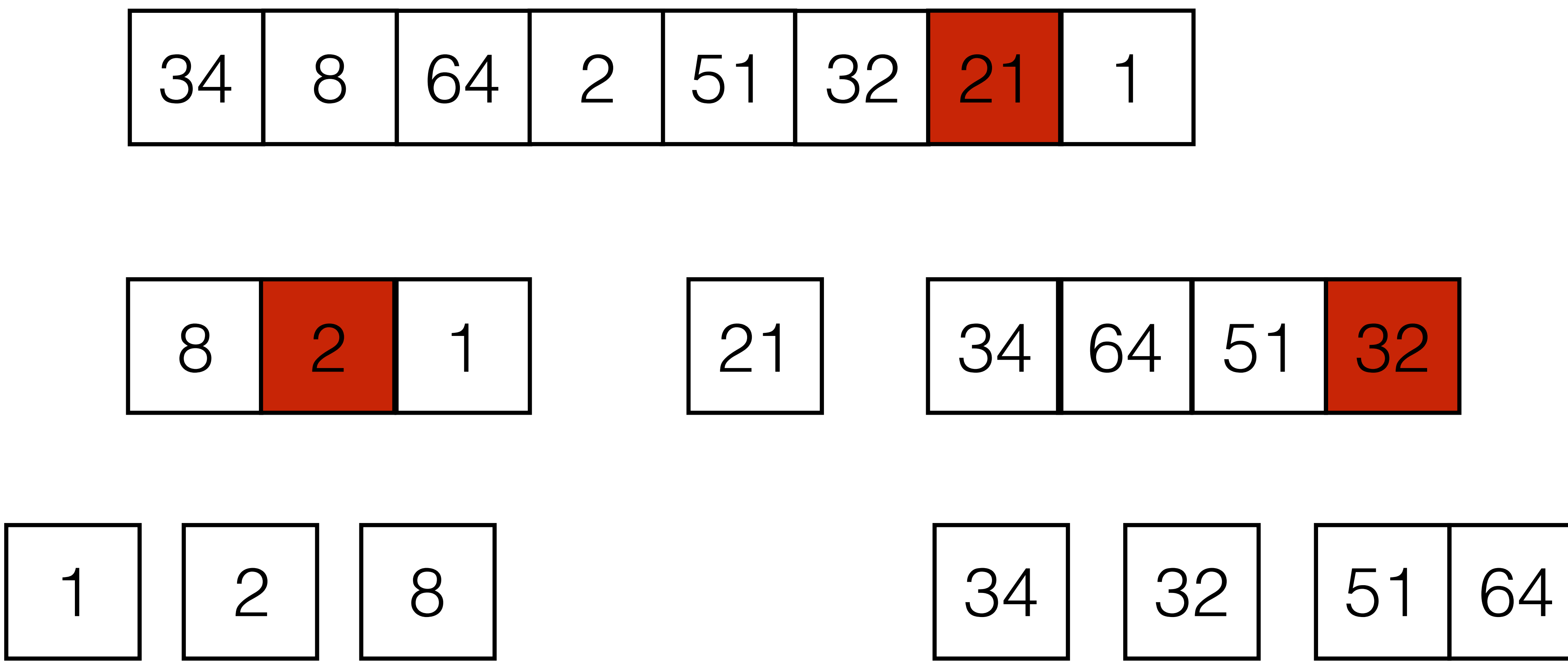
Quick Sort: Best Case

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



Quick Sort: Best Case

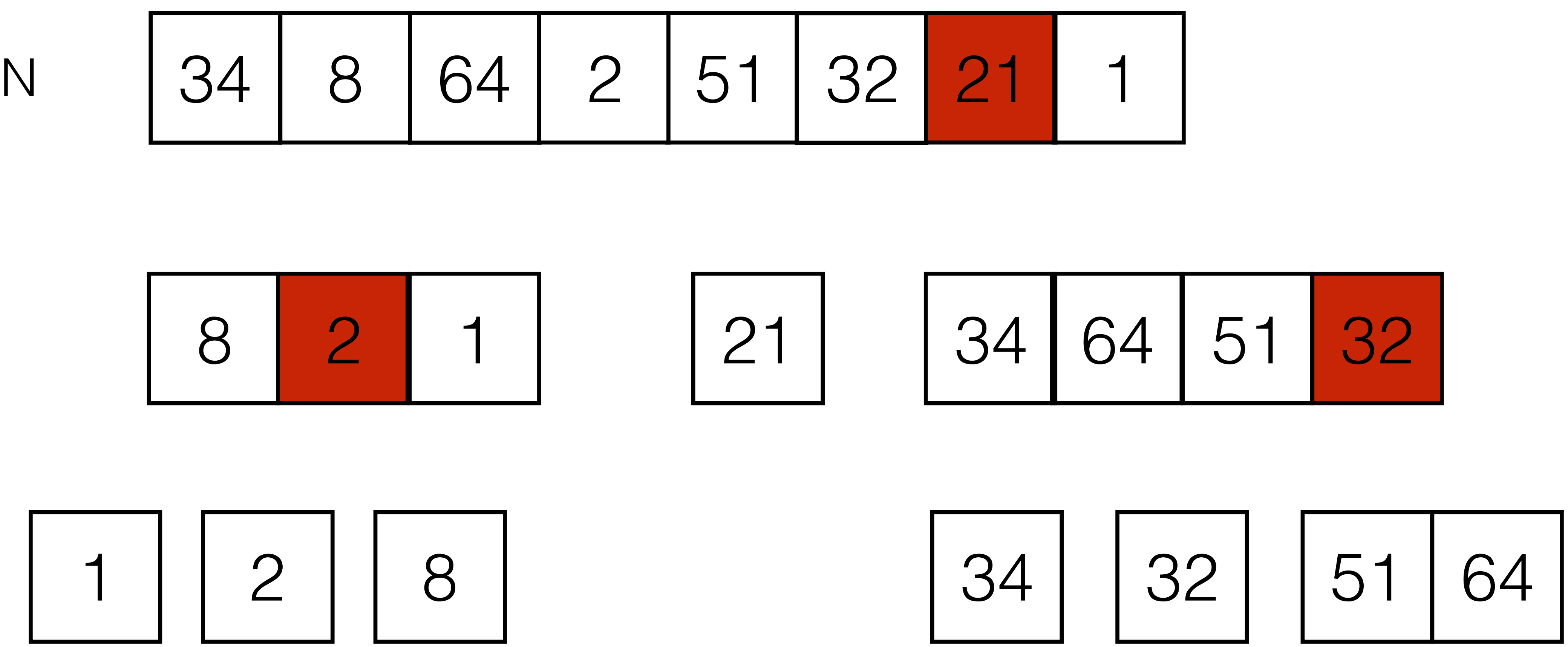
- Best case: Pivot is always the median element.
  Both partitions have about the same size.



Quick Sort: Best Case

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

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

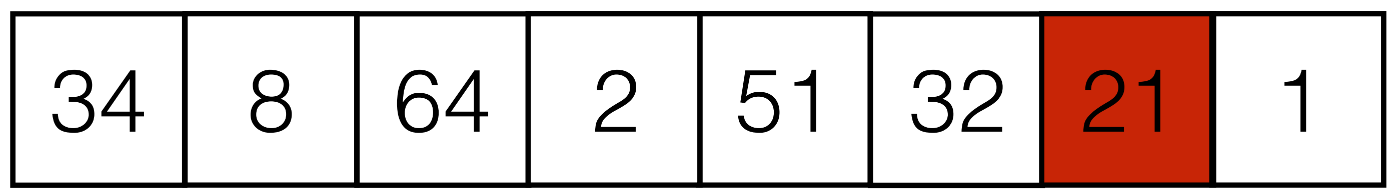


(we ignore the pivot element, so this overestimates the running time slightly)

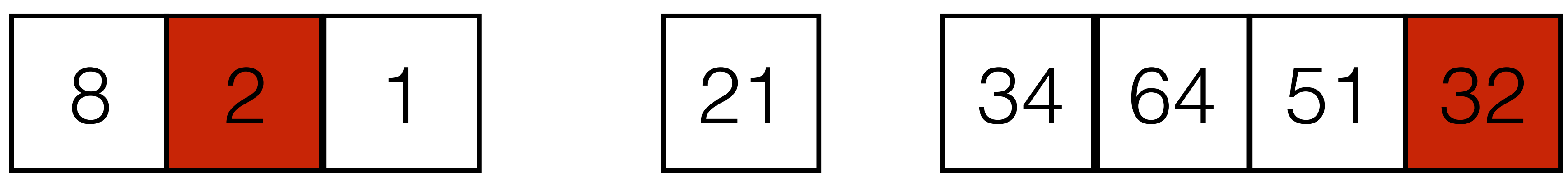
Quick Sort: Best Case

- 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

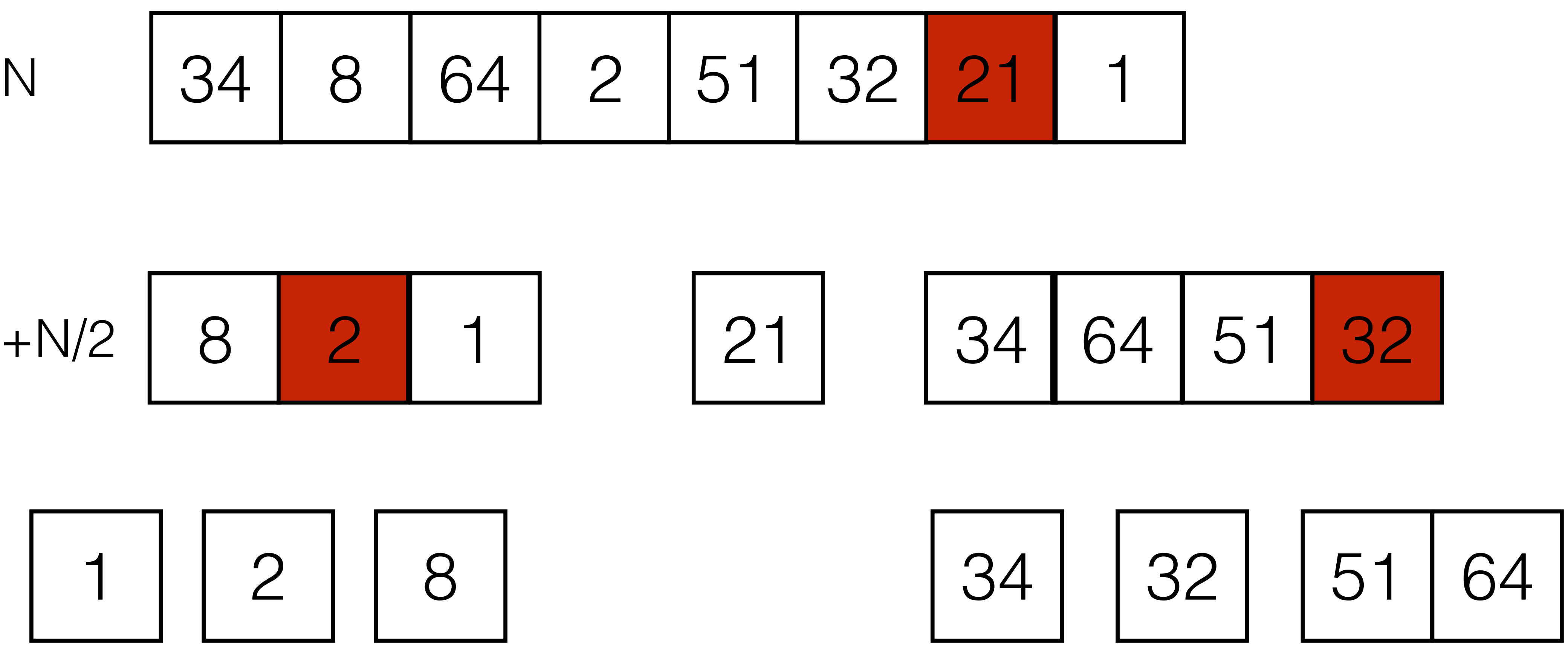
- Best case: Pivot is always the median element.
  Both partitions have about the same size.

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

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

⋮

$T(1) = 1$



(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

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

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

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

(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”*

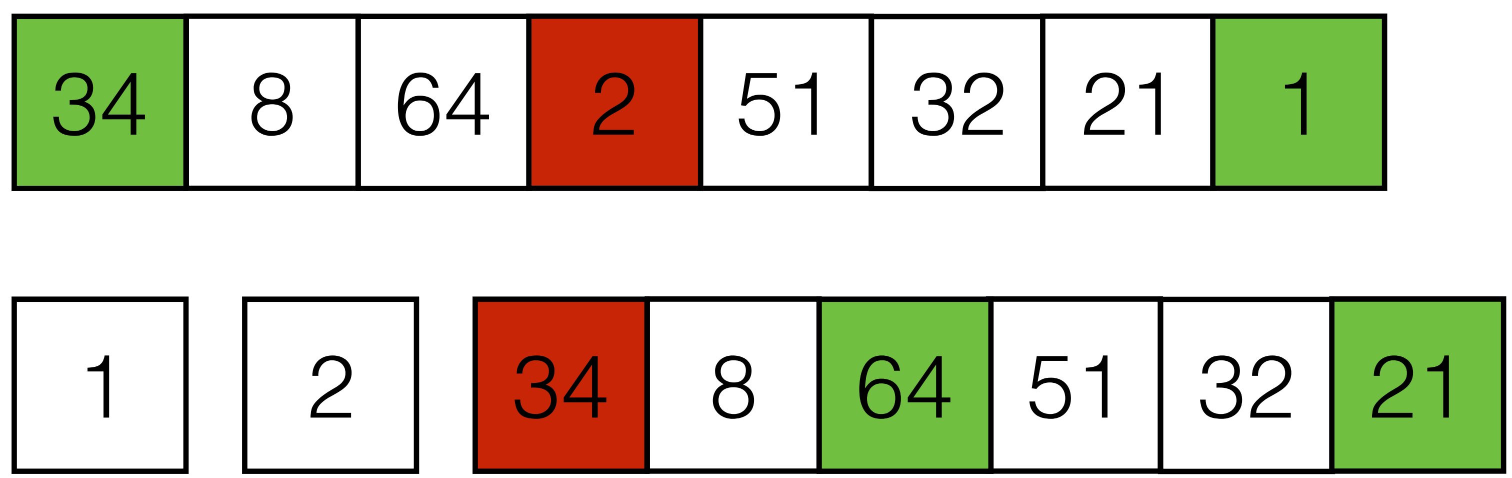
Choosing a Pivot: Median of Three

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

34	8	64	2	51	32	21	1

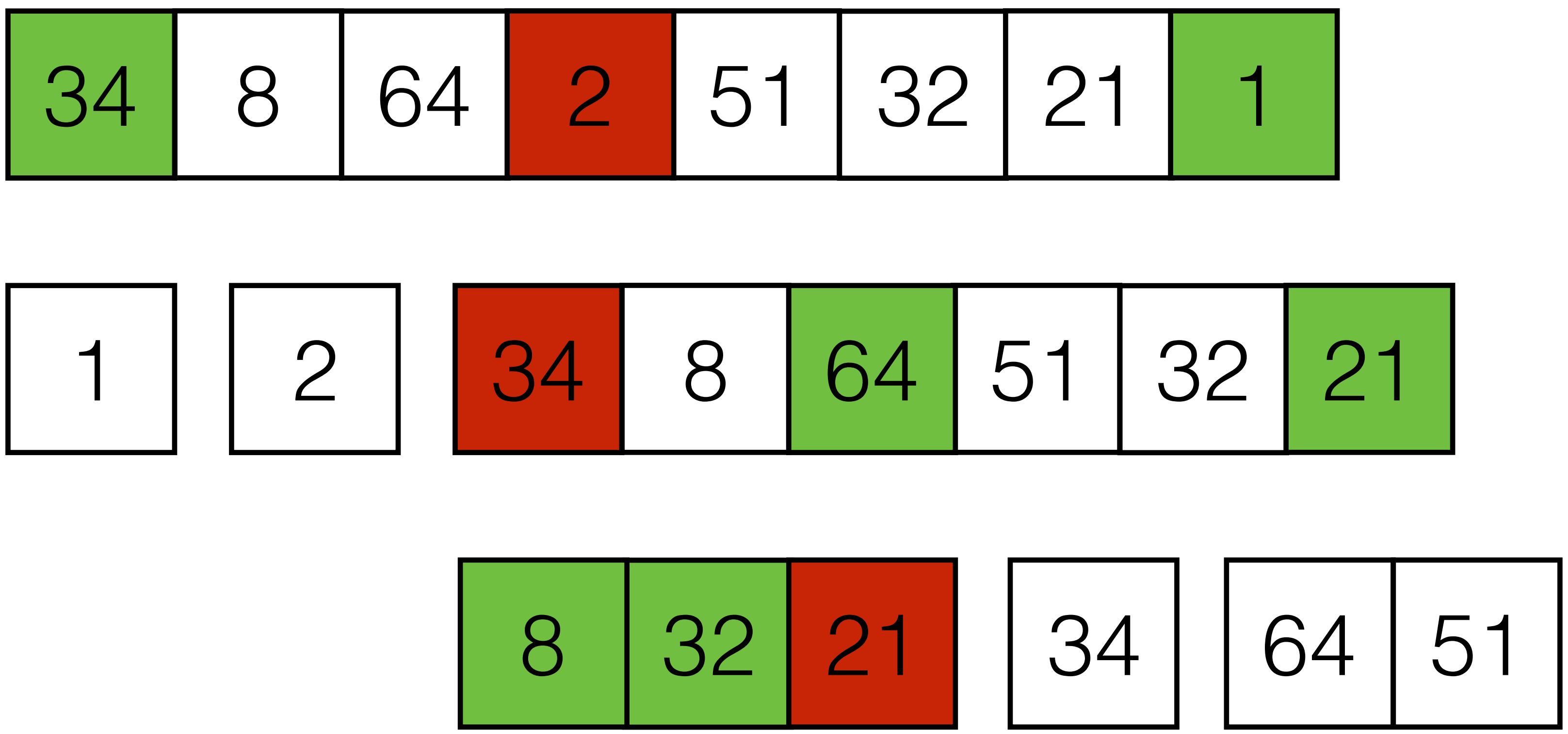
Choosing a Pivot: Median of Three

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



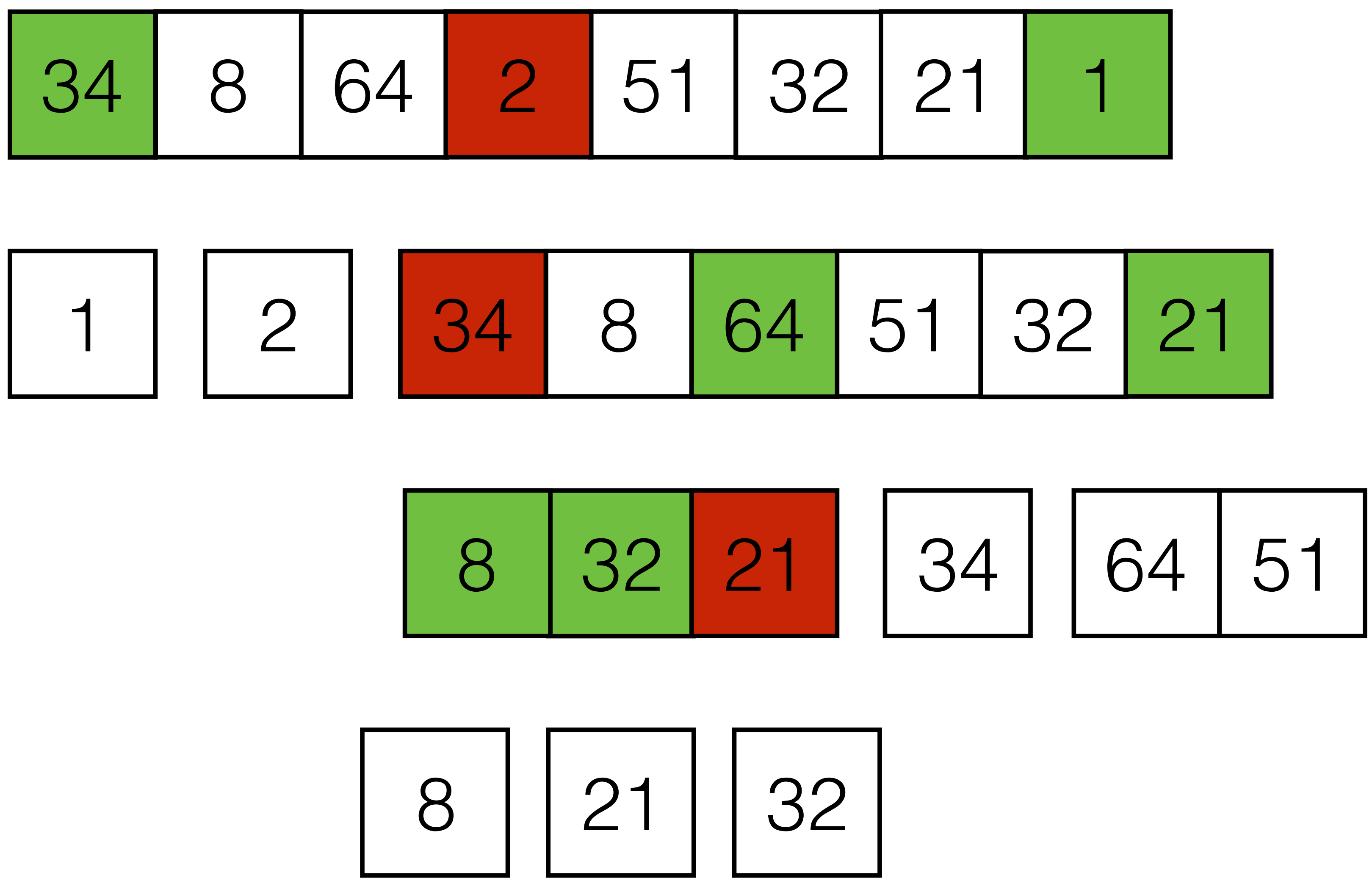
Choosing a Pivot: Median of Three

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

```java
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.
  (but can make 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.
  (but can make QuickSort stable).

- 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	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N^2)$	$O(1)$	✓
Shell Sort	$\Theta(N^{3/2})^*$	$\Theta(N)$	$\Theta(N^{3/2})^*$	$O(1)$	✗
Heap Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(1)$	✗
Merge Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(N)$	✓
Quick Sort	$\Theta(N^2)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(N)$	✗

*depends on increment sequence

gray entries: not shown in class

Comparison-Based Sorting Algorithms

$\Omega(N \log N)$ worst case lower bound on comparison based general sorting!

Can we do better if we make some assumptions?

				ce	Stable?
Insertion Sor				)	✓
Shell Sort				)	✗
Heap Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(1)$	✗
Merge Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(N)$	✓
Quick Sort	$\Theta(N^2)$	$\Theta(NlogN)$	$\Theta(NlogN)$	$O(N)$	✗

*depends on increment sequence

gray entries: not shown in class

Bucket Sort

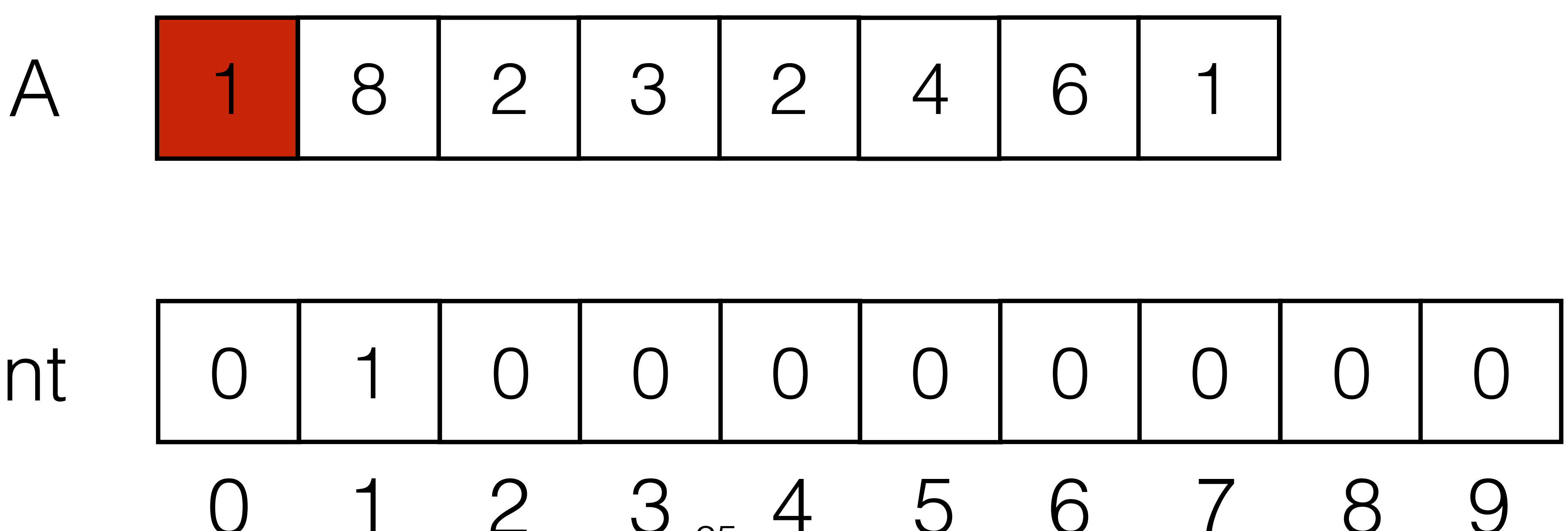
- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].

A	1	8	2	3	2	4	6	1

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

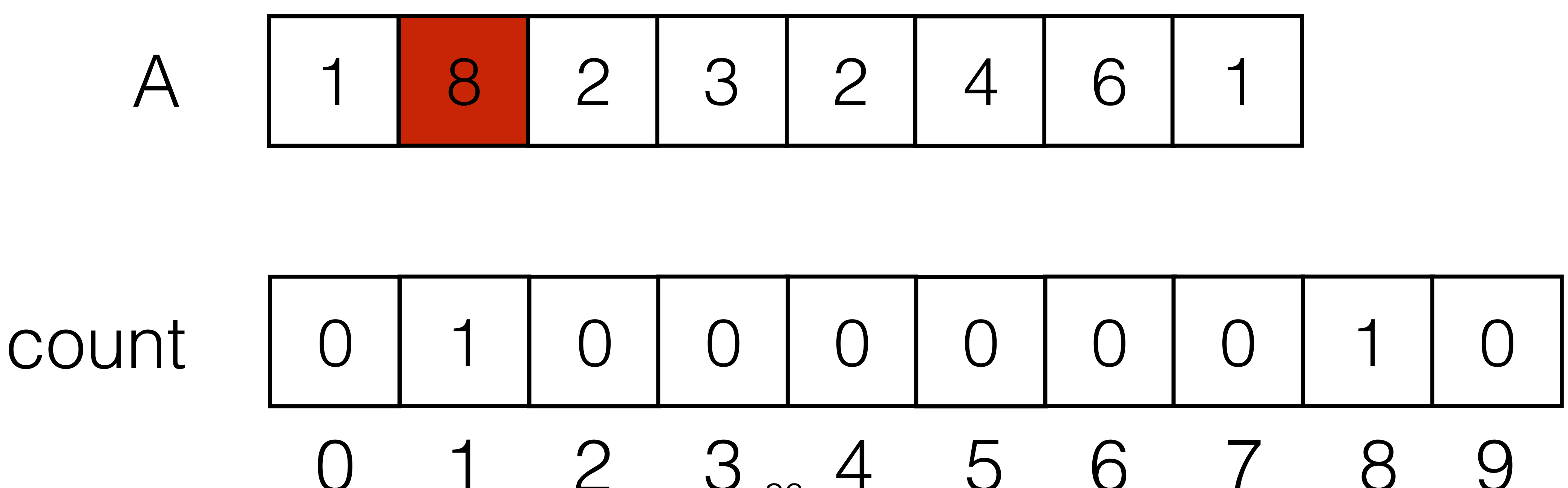
Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].



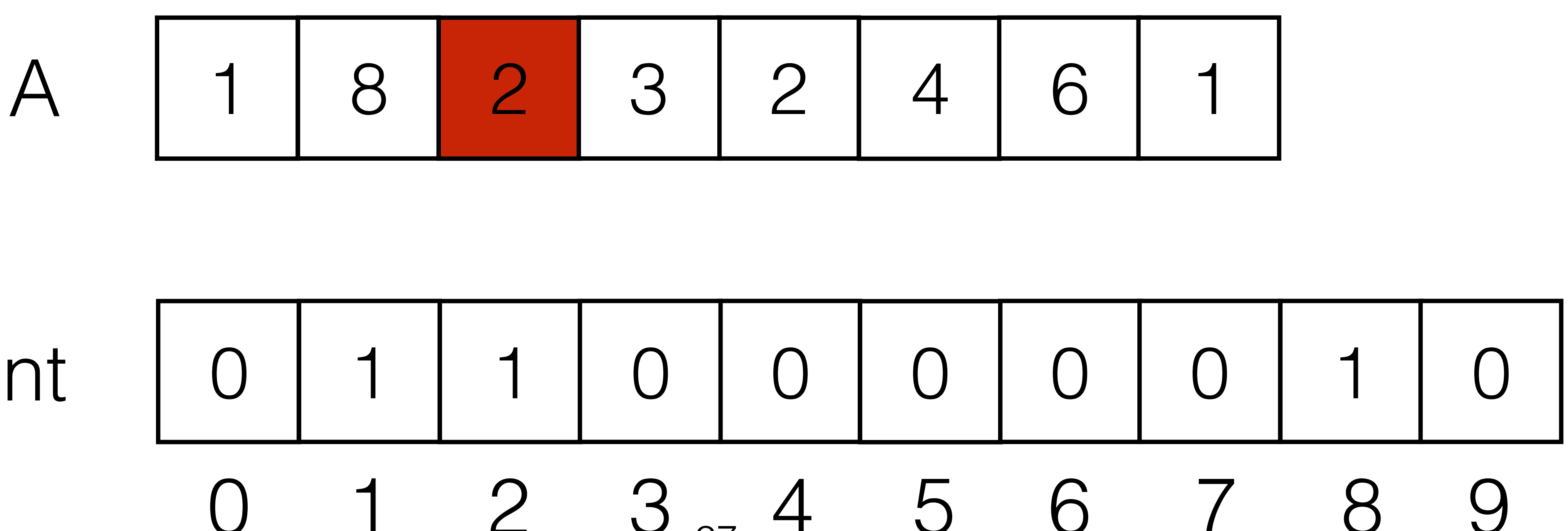
Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].



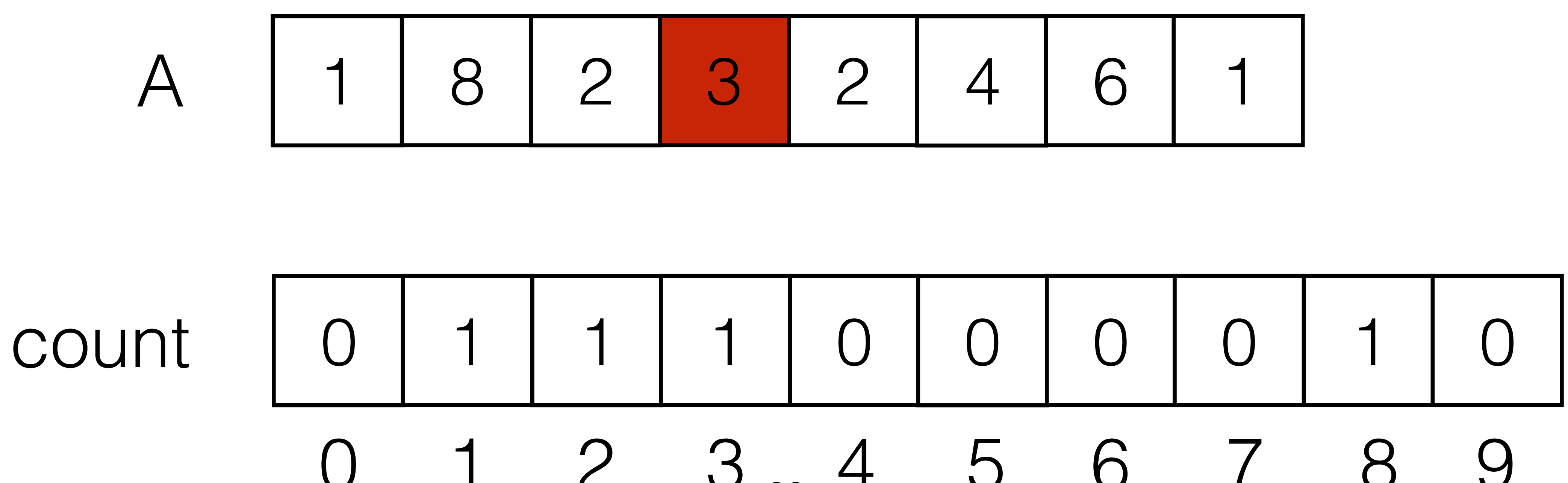
Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].



Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].



Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].

A	1	8	2	3	2	4	6	1

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

Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].

A								
	1	8	2	3	2	4	6	1

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

Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].

A	1	8	2	3	2	4	6	1

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

Bucket Sort

- Assume we know there are M possible values.
- Keep an array **`count`** of length M.
- Scan through the input array A and for each i increment **`count`**$[A_i]$.

A	1	8	2	3	2	4	6	1

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

Bucket Sort

- Assume we know there are *M* possible values.
- Keep an array **`count`** of length *M*.
- Scan through the input array *A* and for each *i* increment **`count`**[A_i].

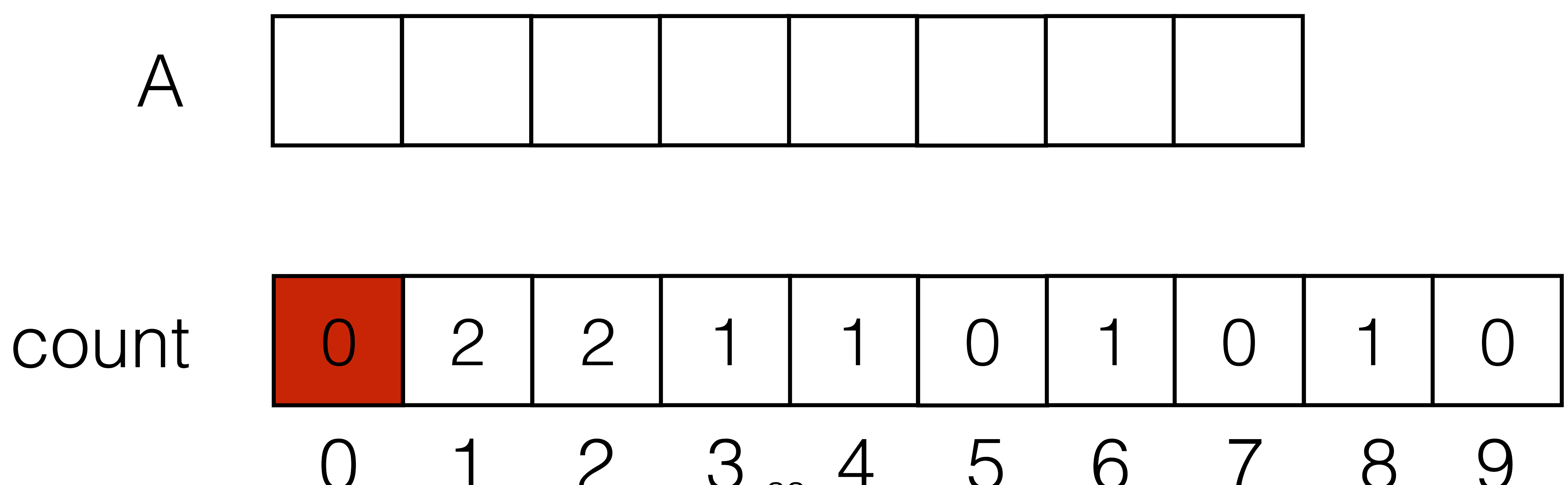
O(N)

A								
	1	8	2	3	2	4	6	1

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

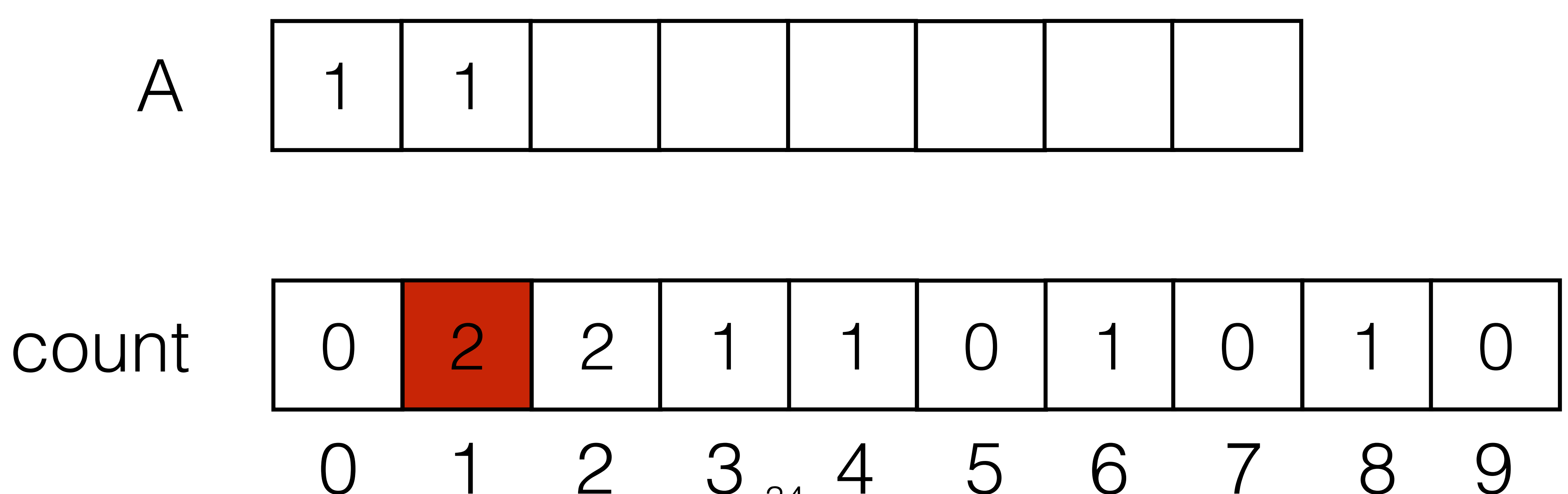
Bucket Sort

- Then iterate through **`count`.** For each *i* write **`count`**[*i*] copies of *i* to *A*.



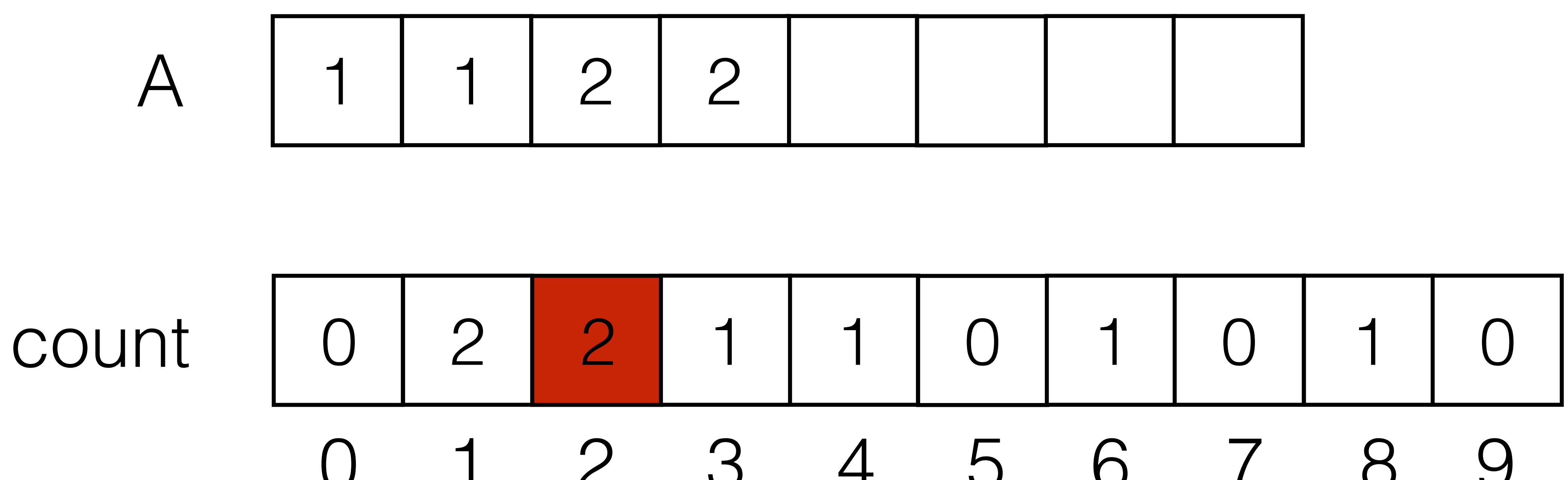
Bucket Sort

- Then iterate through `count`. For each *i* write `count`[*i*] copies of *i* to *A*.



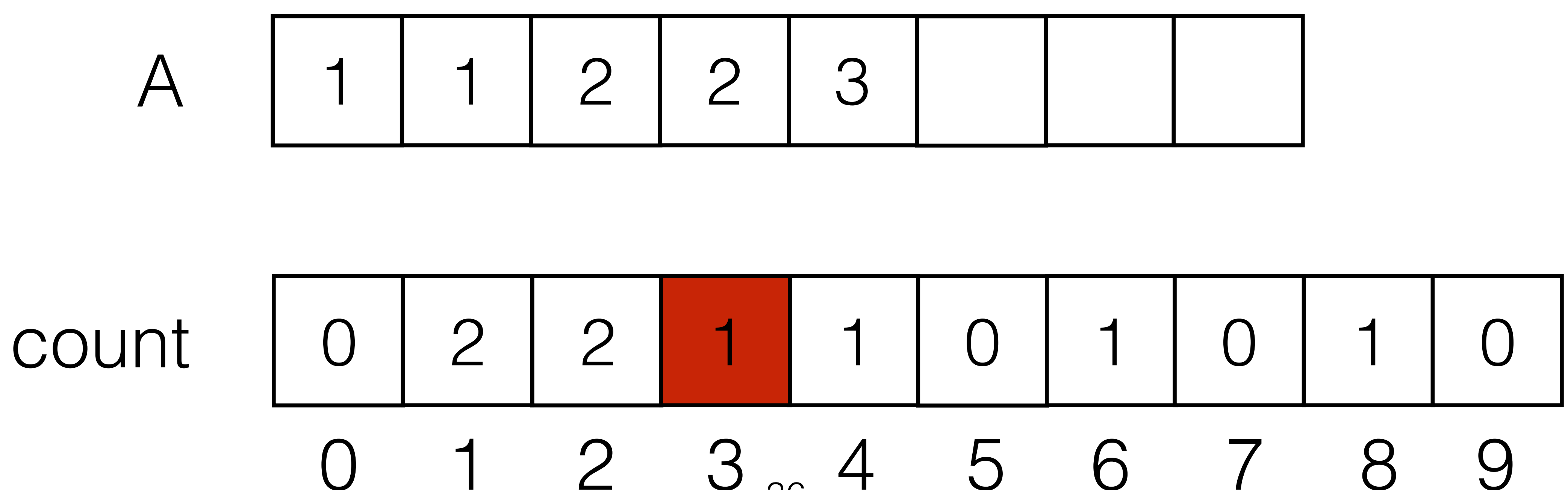
Bucket Sort

- Then iterate through **`count`**. For each *i* write **`count`**[*i*] copies of *i* to *A*.



Bucket Sort

- Then iterate through **`count`**. For each *i* write **`count`**[*i*] copies of *i* to *A*.



Bucket Sort

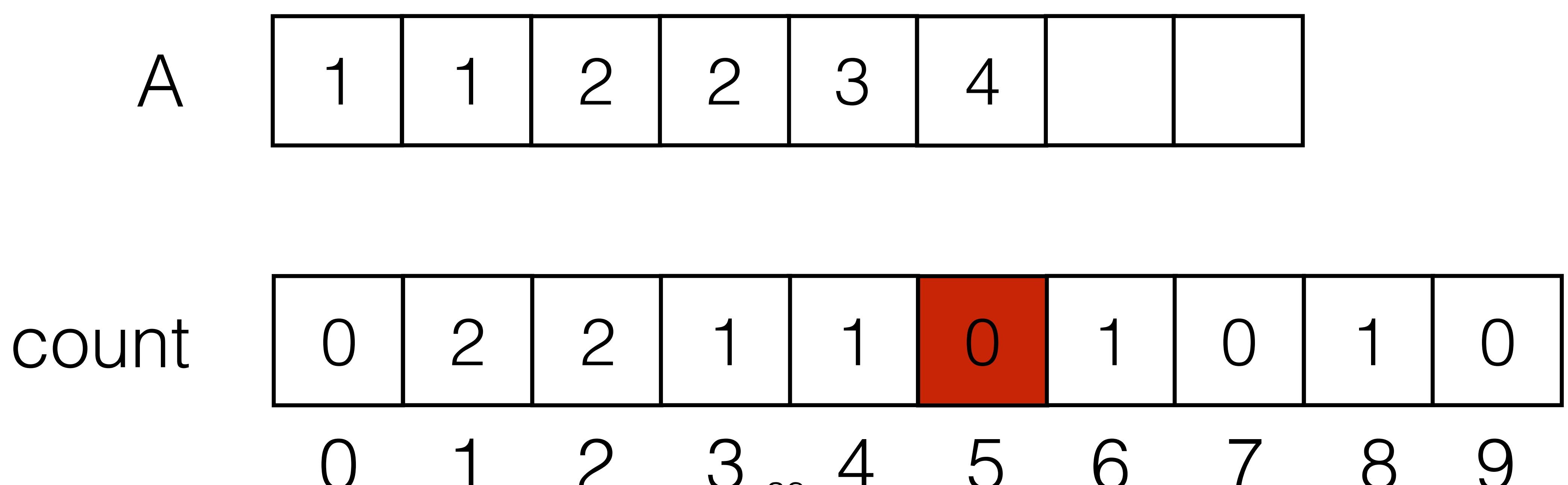
- Then iterate through `count`. For each *i* write `count`[*i*] copies of *i* to *A*.

A	1	1	2	2	3	4		

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

Bucket Sort

- Then iterate through **`count`**. For each *i* write **`count`**[*i*] copies of *i* to *A*.



Bucket Sort

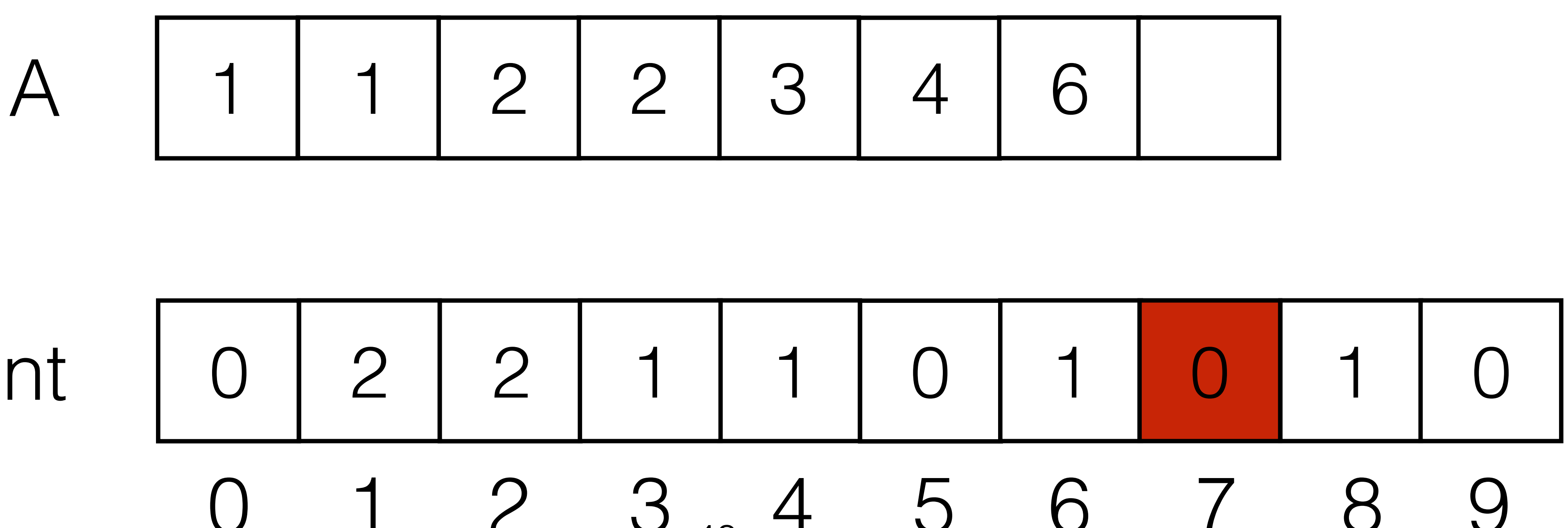
- Then iterate through **`count`**. For each *i* write **`count`**[*i*] copies of *i* to *A*.

A	1	1	2	2	3	4	6	

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

Bucket Sort

- Then iterate through `count`. For each *i* write `count`[*i*] copies of *i* to *A*.



Bucket Sort

- Then iterate through **`count.`** For each *i* write **`count`**[*i*] copies of *i* to *A*.

A									
	1	1	2	2	3	4	6	8	

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

Bucket Sort

- Then iterate through **`count`.** For each *i* write **`count`**[*i*] copies of *i* to *A*.

A								
	1	1	2	2	3	4	6	8

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

Bucket Sort

- Then iterate through `count`. For each *i* write `count`[*i*] copies of *i* to *A*.

O(M)

A	1	1	2	2	3	4	6	8

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

Bucket Sort

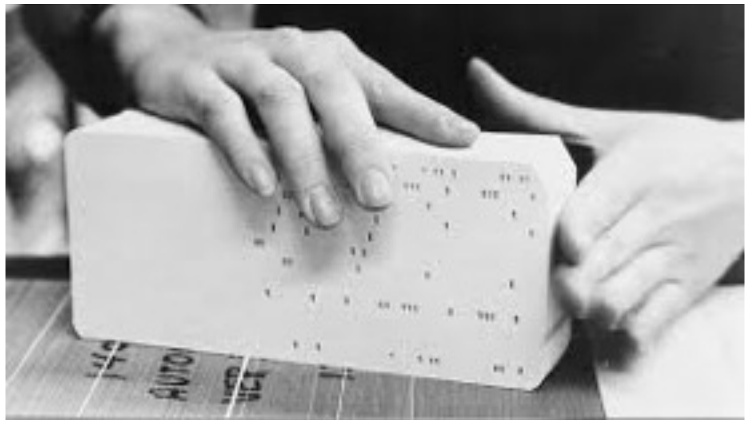
- Then iterate through `count`. For each *i* write `count`[*i*] copies of *i* to *A*.

O(M)

Total time for Bucket Sort:O(N +M)

A	1	1	2	2	3	4	6	8

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



Radix Sort

- Generalization of Bucket sort for Large M.
- Assume M contains all base b numbers up to b^p-1 (e.g. all base-10 integers up to 10^3)
- Do p passes over the data, using Bucket Sort for each digit.
- Bucket sort is stable!

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	
3	
4	064
5	
6	
7	
8	
9	

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	
3	
4	064
5	
6	
7	
8	008
9	

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	
3	
4	064
5	
6	216
7	
8	008
9	

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	512
3	
4	064
5	
6	216
7	
8	008
9	

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	512
3	
4	064
5	
6	216
7	027
8	008
9	

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	
1	
2	512
3	
4	064
5	
6	216
7	027
8	008
9	729

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	000
1	
2	512
3	
4	064
5	
6	216
7	027
8	008
9	729

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	000
1	001
2	512
3	
4	064
5	
6	216
7	027
8	008
9	729

- Bucket sort according to least significant digit.

Radix Sort

06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	000
1	001
2	512
3	003
4	064
5	
6	216
7	027
8	008
9	729

- Bucket sort according to least significant digit.

Radix Sort

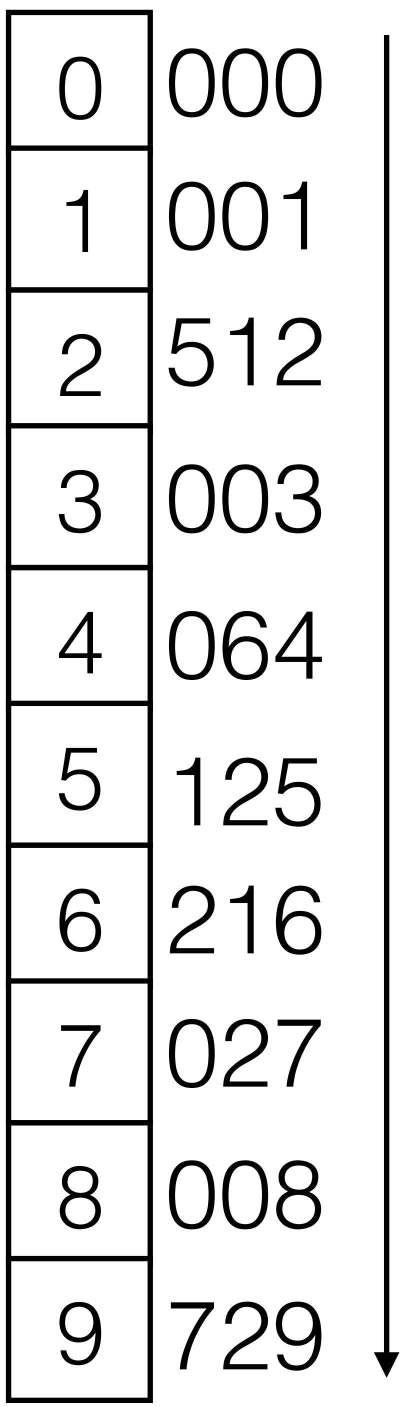
06**4** 00**8** 21**6** 51**2** 02**7** 72**9** 00**0** 00**1** 34**3** 12**5**

0	000
1	001
2	512
3	003
4	064
5	125
6	216
7	027
8	008
9	729

- Bucket sort according to least significant digit.

Radix Sort

000 001 512 343 064 125 216 027 008 729



- read off new sequence

Radix Sort

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

0	000
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	512
2	
3	
4	
5	
6	
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	512
2	
3	
4	343
5	
6	
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	512
2	
3	
4	343
5	
6	064
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	512
2	125
3	
4	343
5	
6	064
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000 001
1	512 216
2	125
3	
4	343
5	
6	064
7	
8	
9	

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000	001
1	512	216
2	125	027
3		
4	343	
5		
6	064	
7		
8		
9		

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000	001	008
1	512	216	
2	125	027	
3			
4	343		
5			
6	064		
7			
8			
9			

- Bucket sort according to second-least significant digit.

Radix Sort

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

0	000	001	008
1	512	216	
2	125	027	729
3			
4	343		
5			
6	064		
7			
8			
9			

- Bucket sort according to second-least significant digit.

Radix Sort

000 001 008 512 216 125 027 729 343 064

0	000	001	008
1	512	216	
2	125	027	729
3			
4	343		
5			
6	064		
7			
8			
9			

- read off new sequence

Radix Sort

000 **0**01 **0**08 **5**12 **2**16 **1**25 **0**27 **7**29 **3**43 **0**64

0	000 001 008 027 064
1	125
2	216
3	343
4	
5	512
6	
7	729
8	
9	

- Bucket sort according to third-least significant digit.

Radix Sort

000 001 008 027 064 125 216 343 512 729

0	000 001 008 027 064
1	125
2	216
3	343
4	
5	512
6	
7	729
8	
9	

- read off new sequence
- Sorted!

Radix Sort

000 001 008 027 064 125 216 343 512 729

0	000 001 008 027 064
1	125
2	216
3	343
4	
5	512
6	
7	729
8	
9	

- read off new sequence
- Sorted!

Each Bucket Sort: O(N+b)
There are p Bucket Sorts,
so total time for Radix sort: O(p (N+b))

Sorting Strings with Radix Sort