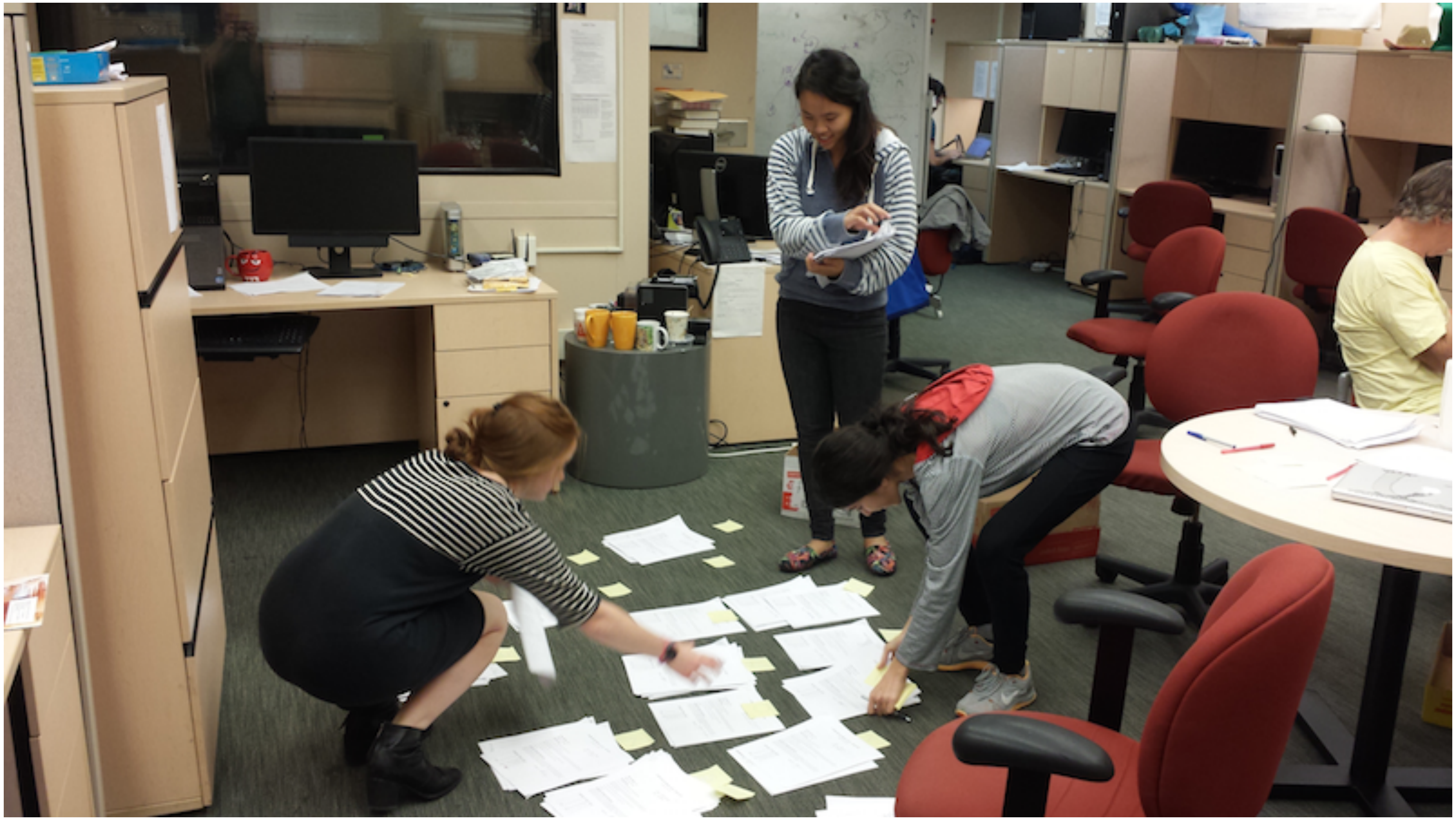
Data Structures in Java

Lecture 14: Sorting I

11/9/2015

Daniel Bauer

Sorting Midterm Exams



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).

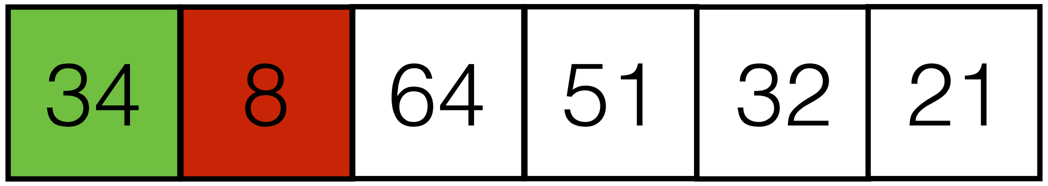
Sorting Applications

- 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.
  - Insertion sort
  - Shell sort
  - Heap sort
  - Merge sort
  - Quick sort
  - Radix Sort

Insertion Sort

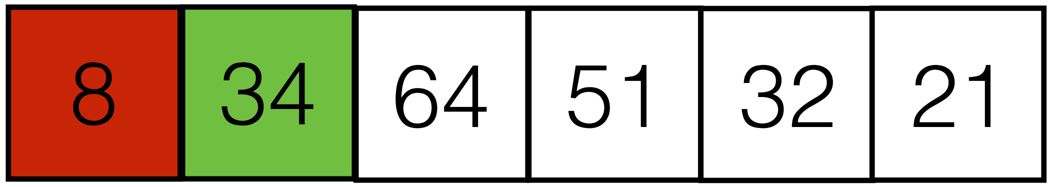


34	8	64	51	32	21

p=1

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

Insertion Sort

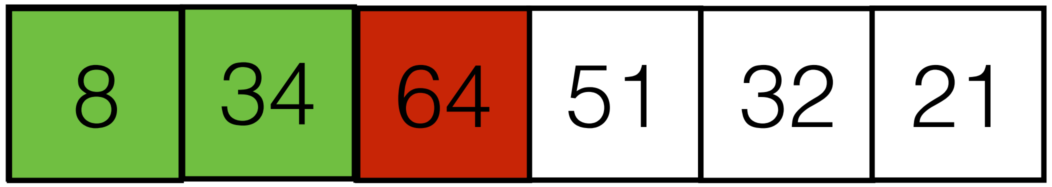


8	34	64	51	32	21

p=1

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

Insertion Sort

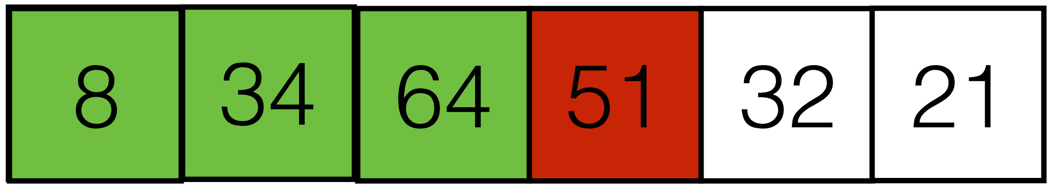


8	34	64	51	32	21

p=2

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

Insertion Sort

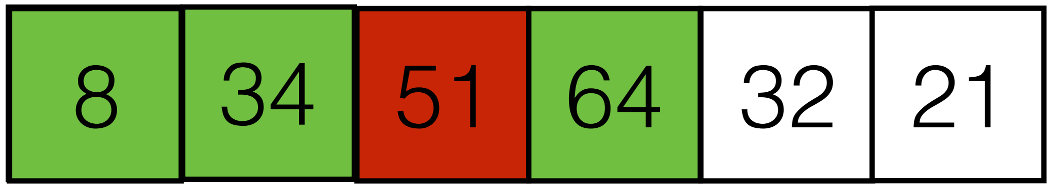


🟩 8	🟩 34	🟩 64	🟥 51	32	21

p=3

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

Insertion Sort

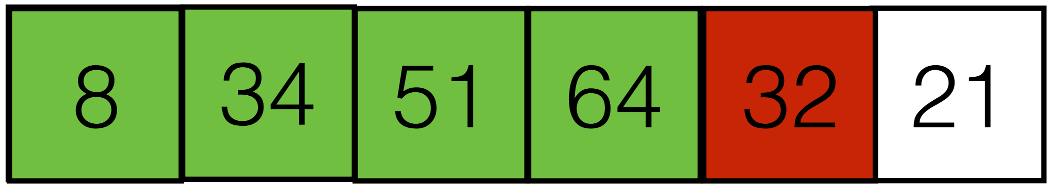


8	34	51	64	32	21

p=3

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

Insertion Sort

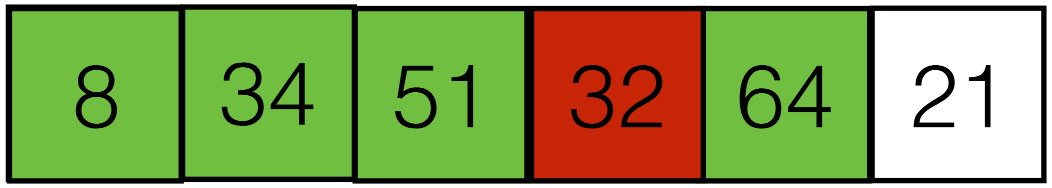


8	34	51	64	32	21

p=4

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

Insertion Sort

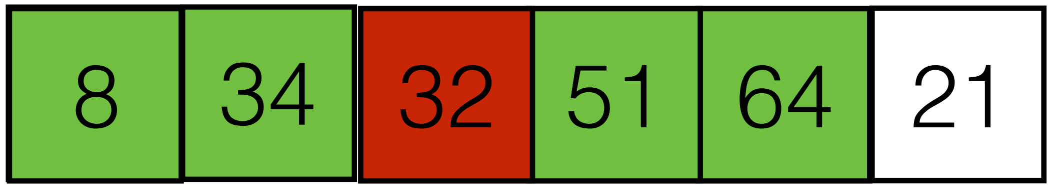


8	34	51	32	64	21

p=4

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

Insertion Sort

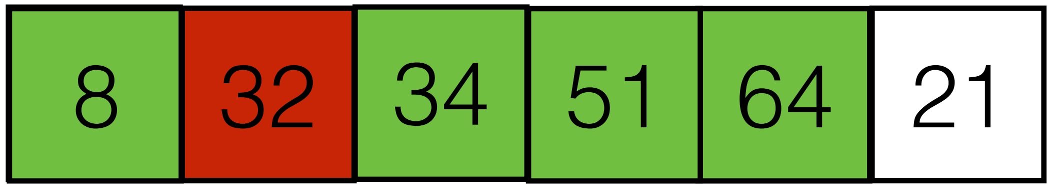


8	34	32	51	64	21

p=4

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

Insertion Sort

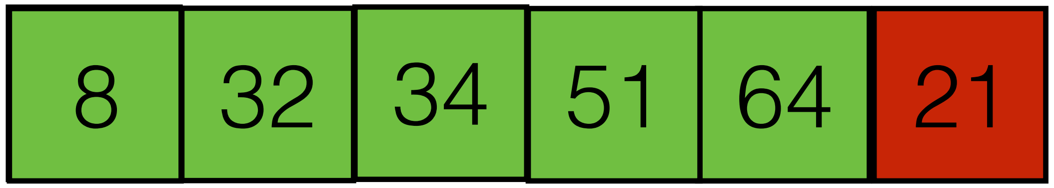


8	32	34	51	64	21

p=4

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

Insertion Sort

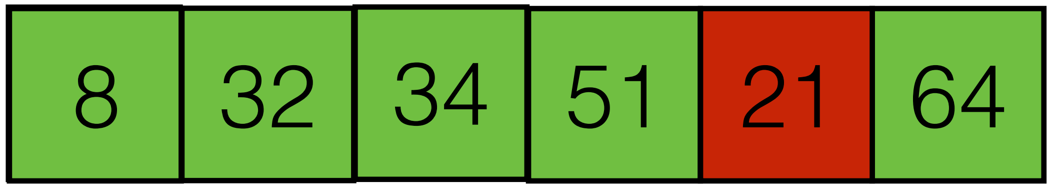


8	32	34	51	64	21

p=5

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

Insertion Sort

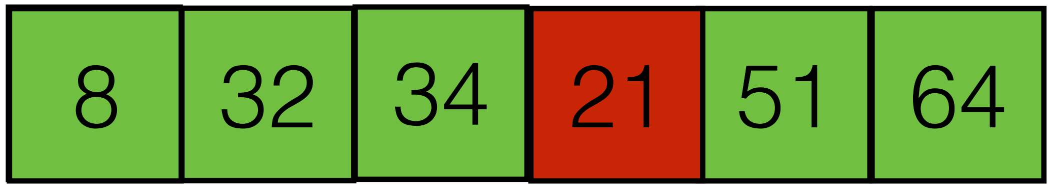


8	32	34	51	21	64

p=5

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

Insertion Sort

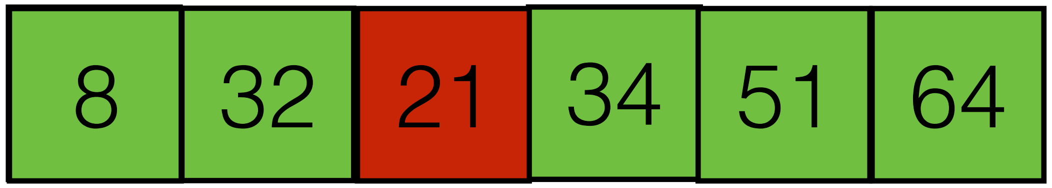


🟩 8	🟩 32	🟩 34	🟥 21	🟩 51	🟩 64

p=5

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

Insertion Sort

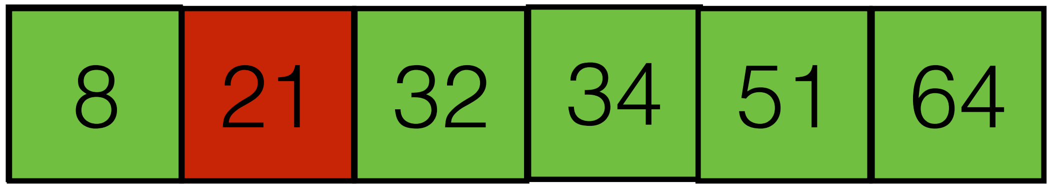


8 32 21 34 51 64

p=5

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

Insertion Sort



8	21	32	34	51	64

p=5

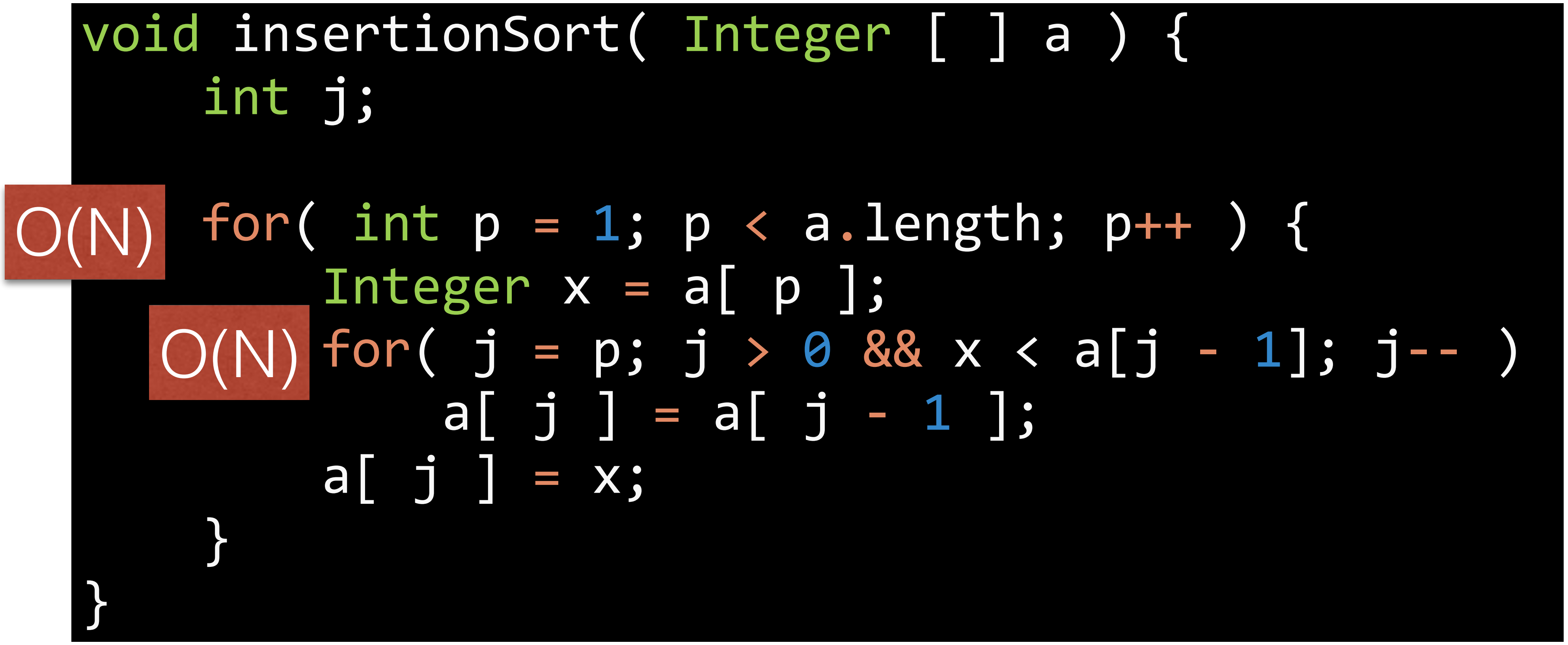
- Perform p=1…N-1 passes through the array.
  - Assume array[0..p-1] is already sorted.
  - Take the element x at position p.
    - Repeatedly swap x its left neighbor until it is in the correct position.

Insertion Sort

```
void insertionSort( Integer [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[j - 1]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Insertion Sort



```
void insertionSort( Integer [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[j - 1]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Total: $O(N^2)$

Insertion Sort

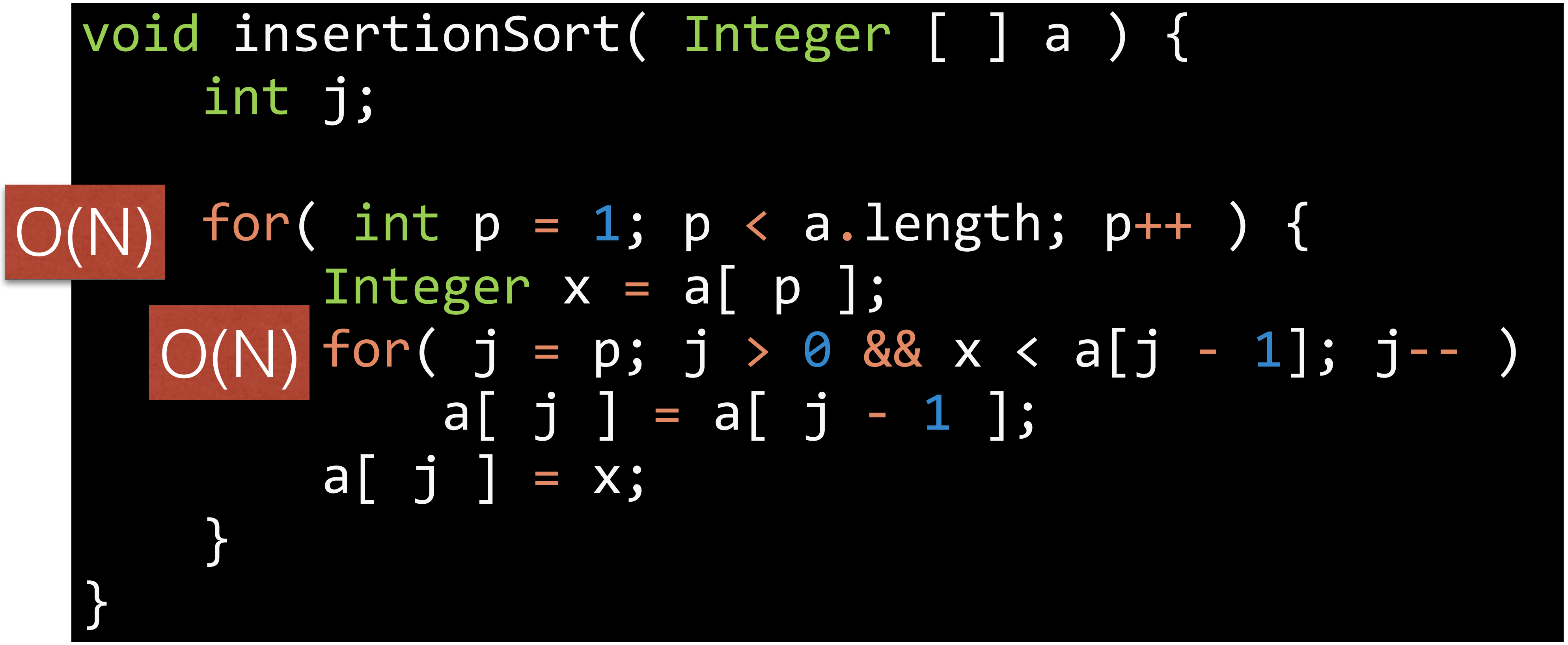
```
     void insertionSort( Integer [ ] a ) {
         int j;

O(N)     for( int p = 1; p < a.length; p++ ) {
             Integer x = a[ p ];
    O(N)     for( j = p; j > 0 && x < a[j - 1]; j-- )
                 a[ j ] = a[ j - 1 ];
             a[ j ] = x;
         }
     }
```

Total: O(N^2)

Best case input (sorted): O(N)

Insertion Sort



```
void insertionSort( Integer [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[j - 1]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Total: $O(N^2)$

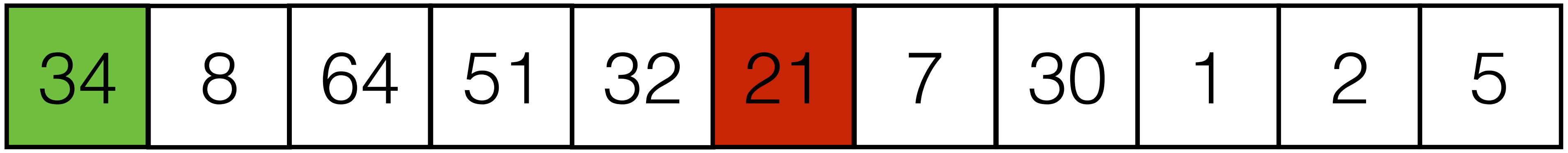
Best case input (sorted): $O(N)$

Worst case input (sorted in reverse order):

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

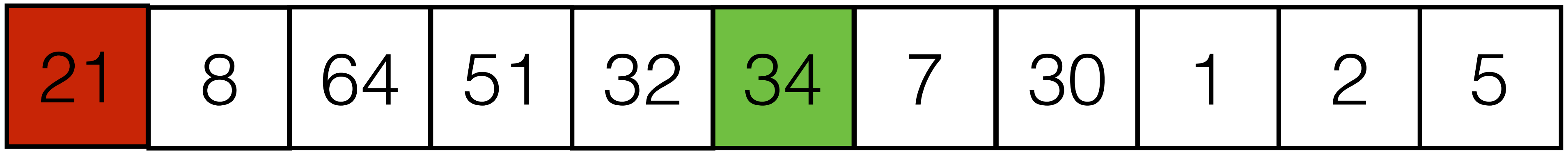


34	8	64	51	32	21	7	30	1	2	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps h_1, h_2, … ,h_t



21	8	64	51	32	34	7	30	1	2	5

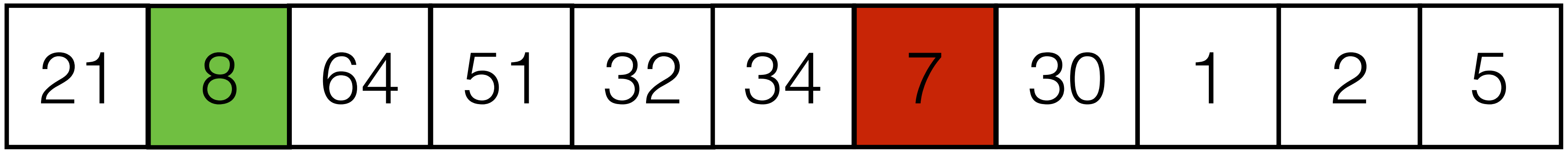
h_3 = 5 h_2 = 3 h_1 = 1

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

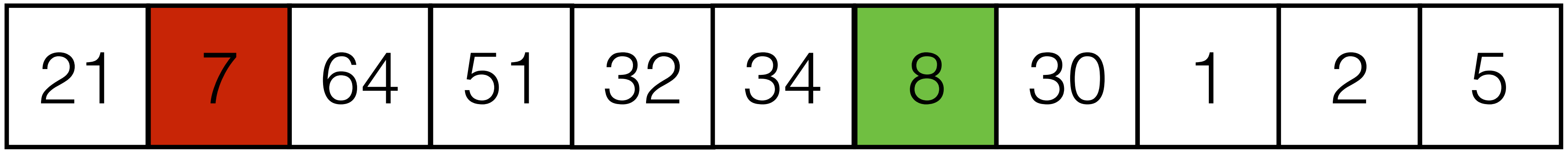


21	8	64	51	32	34	7	30	1	2	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$

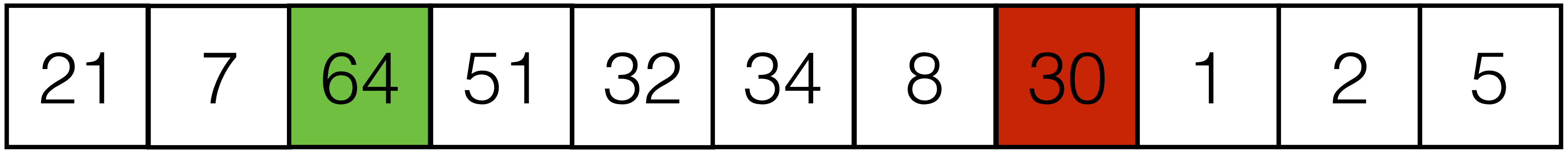


21	7	64	51	32	34	8	30	1	2	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

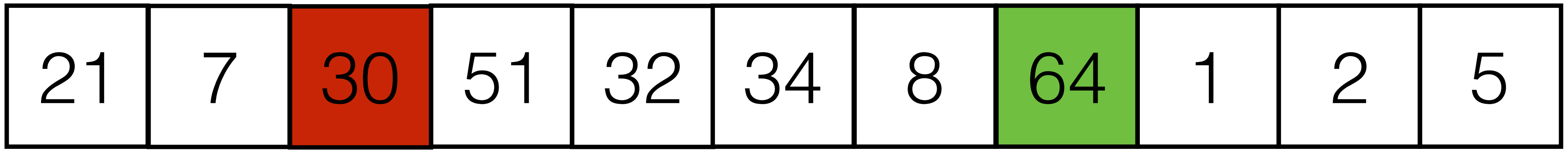


21	7	64	51	32	34	8	30	1	2	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \dots, h_t$



21	7	30	51	32	34	8	64	1	2	5

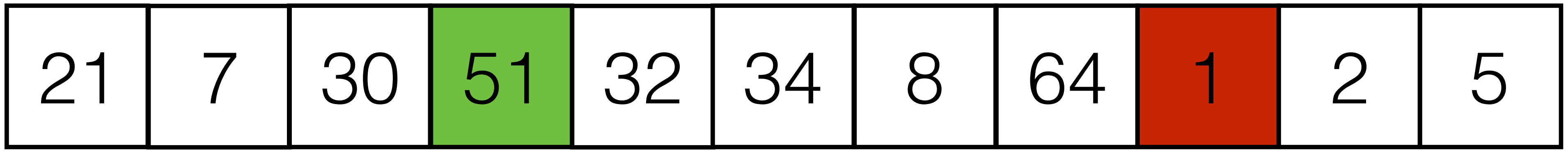
$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

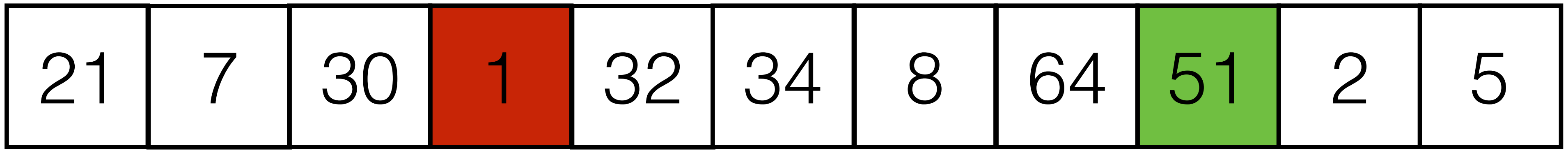


21	7	30	51	32	34	8	64	1	2	5

$\mathbf{h_3 = 5}$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

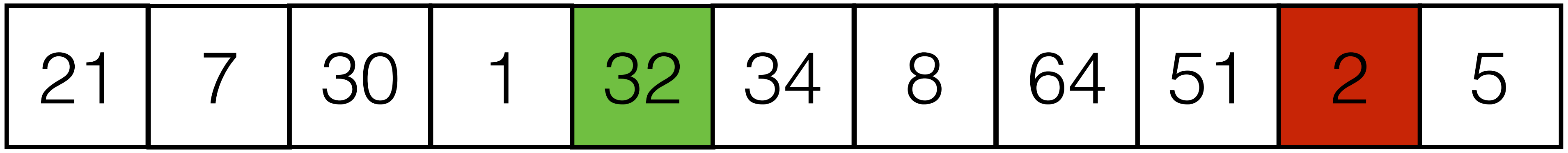


21	7	30	1	32	34	8	64	51	2	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



21	7	30	1	32	34	8	64	51	2	5

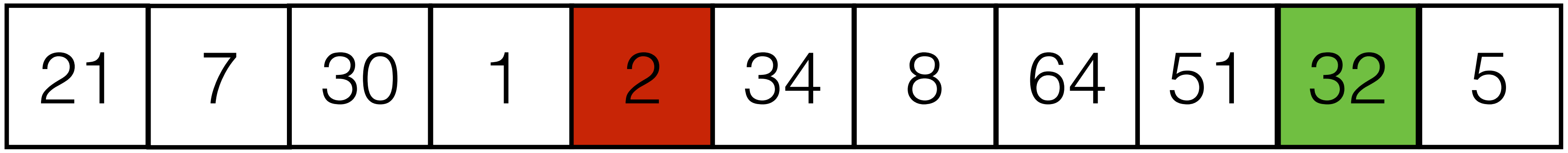
$\mathbf{h_3 = 5}$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



21	7	30	1	2	34	8	64	51	32	5

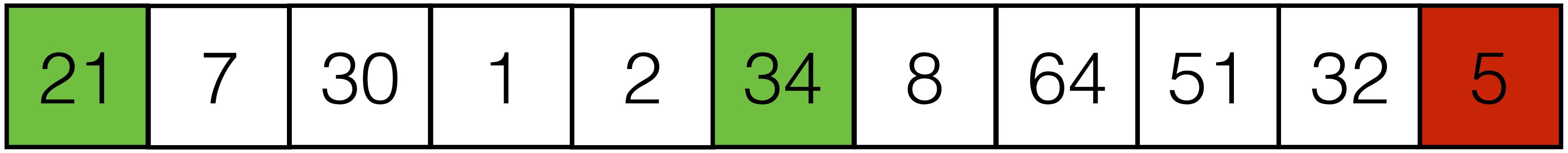
$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

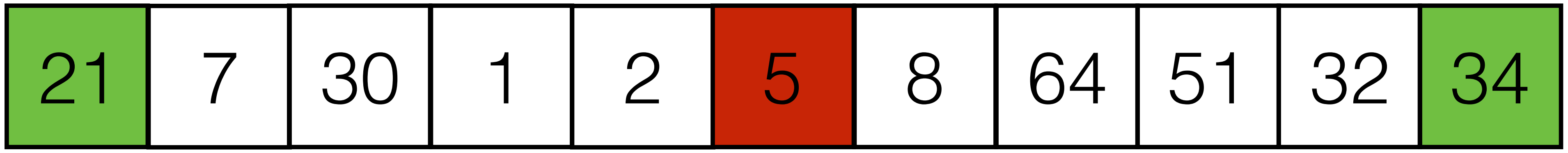


21	7	30	1	2	34	8	64	51	32	5

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

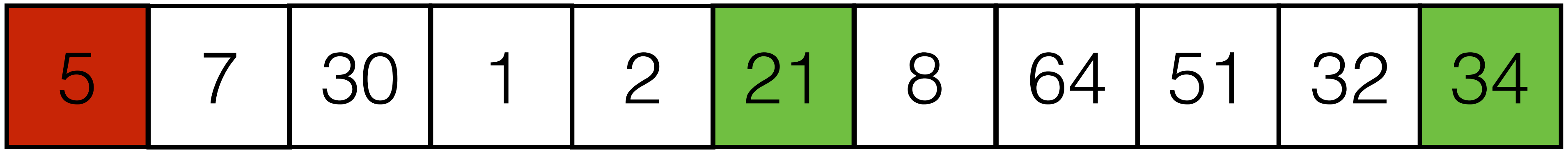


21	7	30	1	2	5	8	64	51	32	34

$h_3 = 5$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$

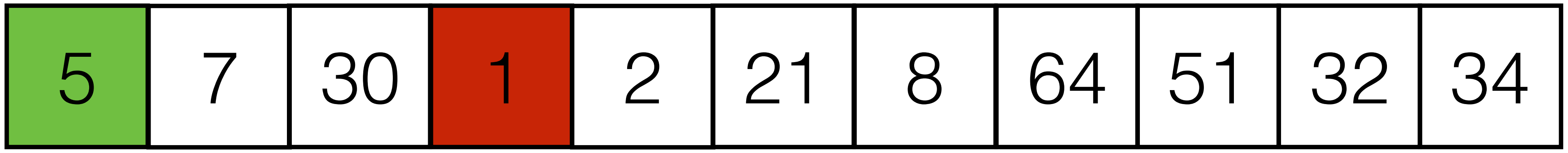


5	7	30	1	2	21	8	64	51	32	34

$\mathbf{h_3 = 5}$ $h_2 = 3$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$



5	7	30	1	2	21	8	64	51	32	34

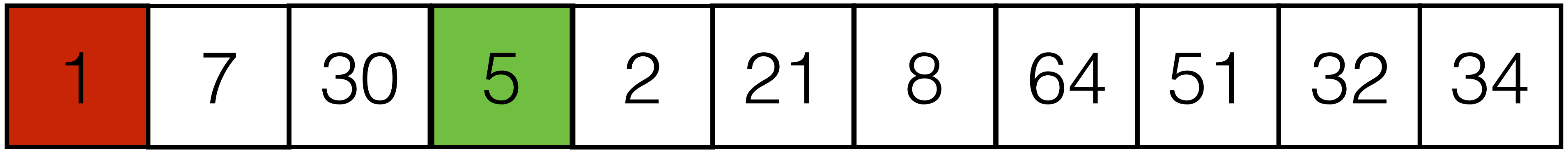
$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

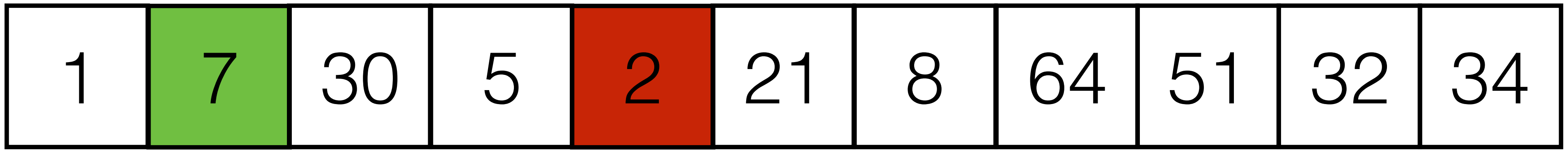


1	7	30	5	2	21	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \ldots, h_t$



1	7	30	5	2	21	8	64	51	32	34

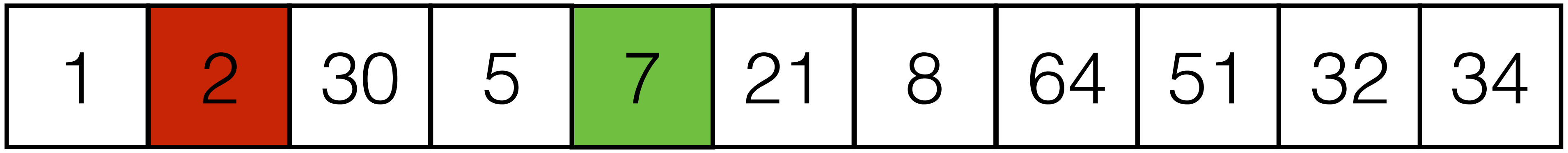
$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

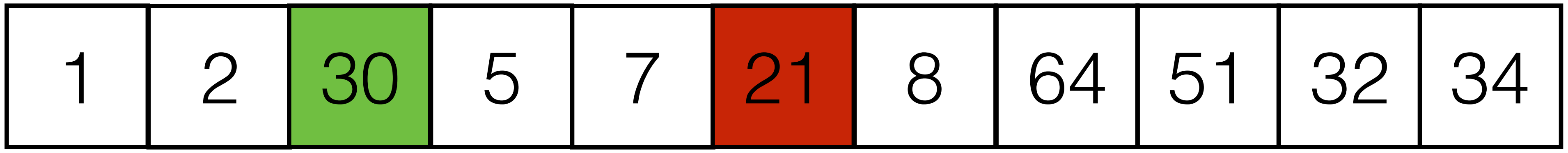


1	2	30	5	7	21	8	64	51	32	34

$h_3 = 5$ $\mathbf{h_2 = 3}$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

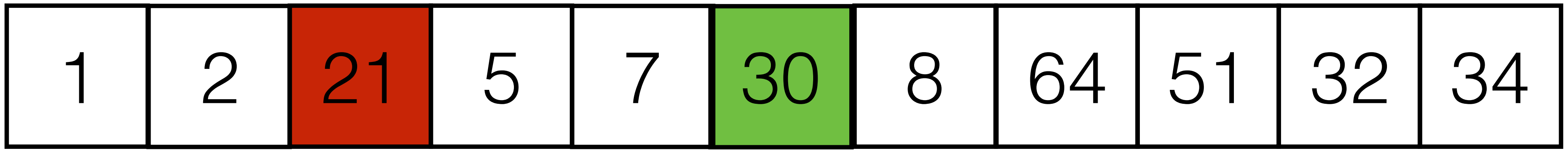


1	2	30	5	7	21	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$

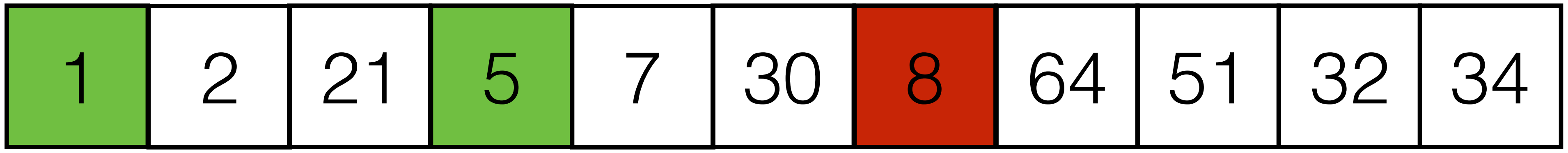


1	2	🟥 21	5	7	🟩 30	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$

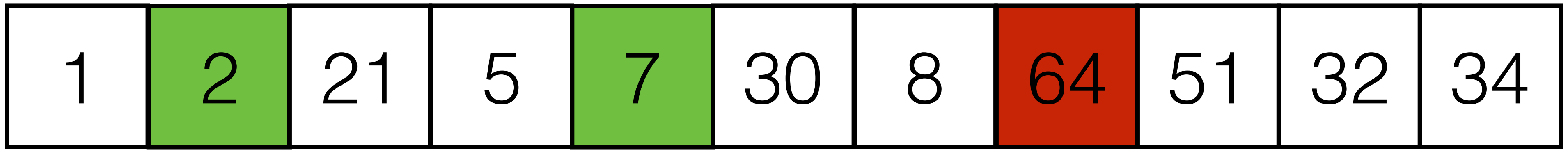


1	2	21	5	7	30	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$

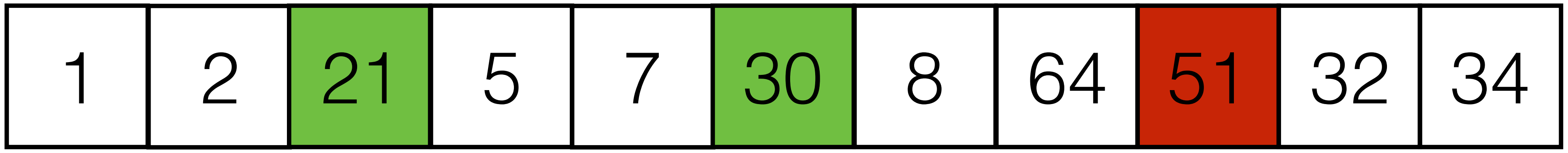


1	2	21	5	7	30	8	64	51	32	34

$h_3 = 5$ $\mathbf{h_2 = 3}$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

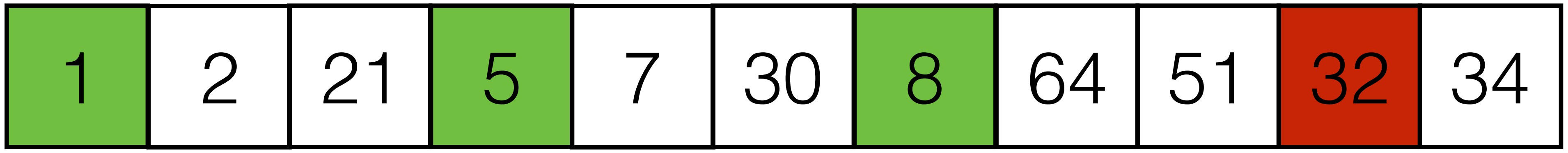


1	2	21	5	7	30	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \ldots, h_t$

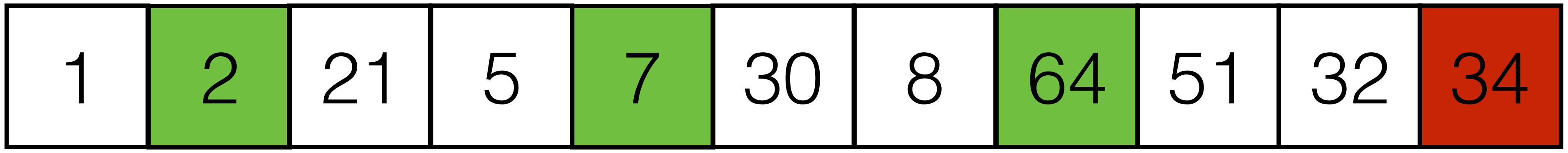


1	2	21	5	7	30	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \ldots, h_t$

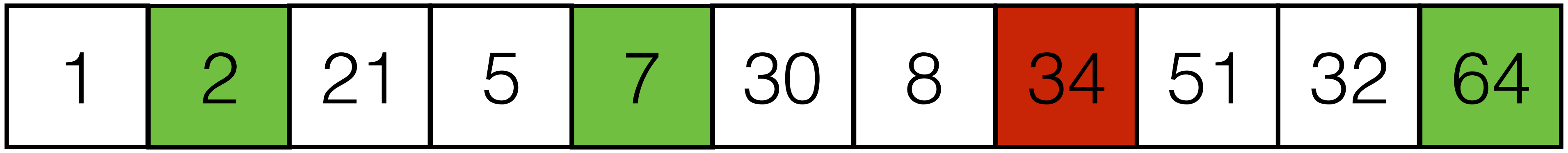


1	2	21	5	7	30	8	64	51	32	34

$h_3 = 5$ **$h_2 = 3$** $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \ldots, h_t$



1	🟩 2	21	5	🟩 7	30	8	🟥 34	51	32	🟩 64

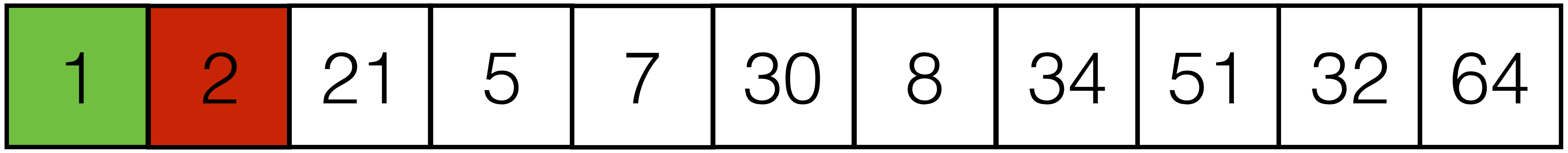
$h_3 = 5$ $\mathbf{h_2 = 3}$ $h_1 = 1$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- "increment sequence" of steps $h_1, h_2, \ldots, h_t$

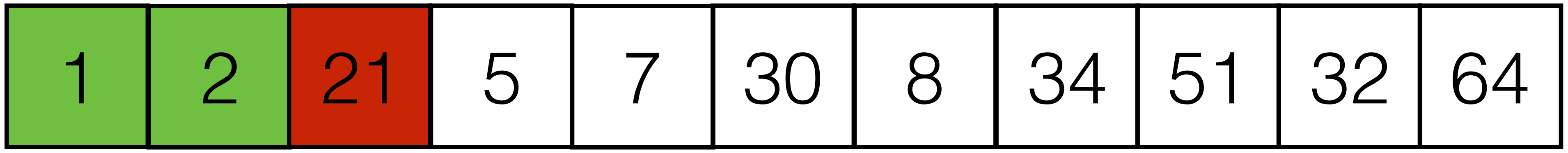


🟩 1	🟥 2	21	5	7	30	8	34	51	32	64

$h_3 = 5$ $h_2 = 3$ $\mathbf{h_1 = 1}$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



1	2	21	5	7	30	8	34	51	32	64

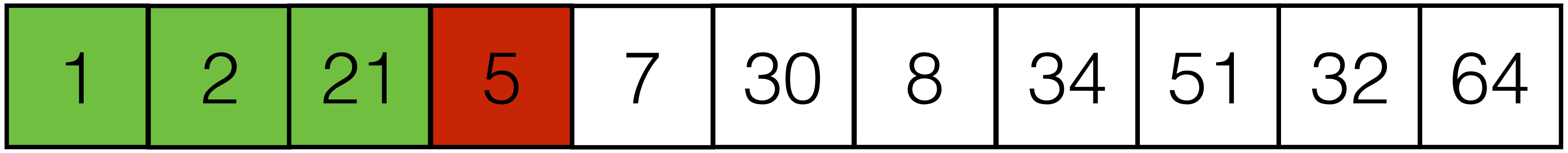
$h_3 = 5$ $h_2 = 3$ $\mathbf{h_1 = 1}$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps h_1, h_2, … ,h_t

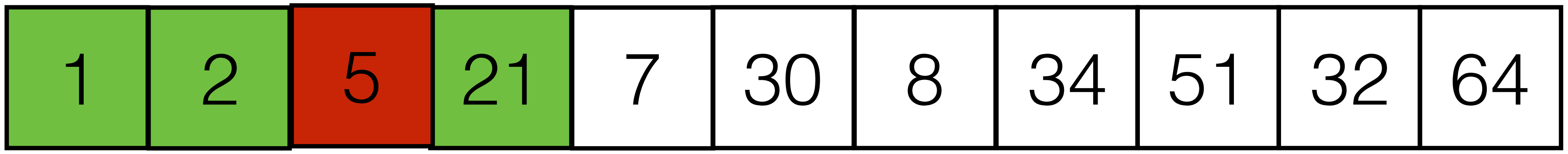


1	2	21	5	7	30	8	34	51	32	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \ldots, h_t$



1	2	5	21	7	30	8	34	51	32	64

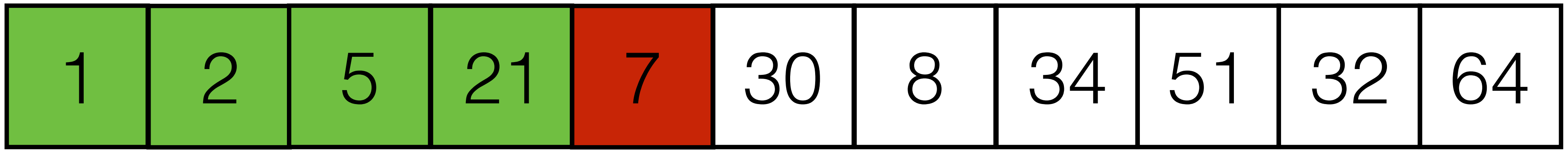
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

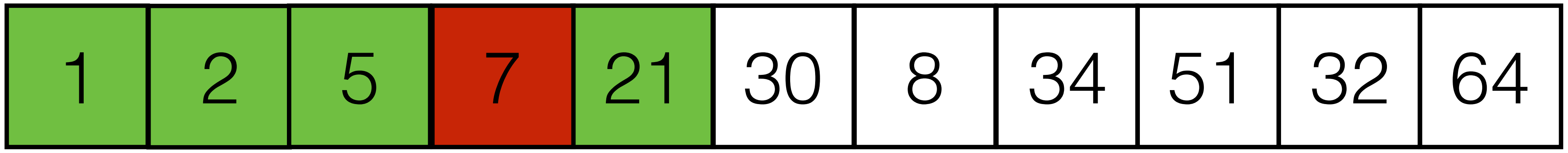


1	2	5	21	7	30	8	34	51	32	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- "increment sequence" of steps $h_1, h_2, \dots, h_t$



🟩 1	🟩 2	🟩 5	🟥 7	🟩 21	30	8	34	51	32	64

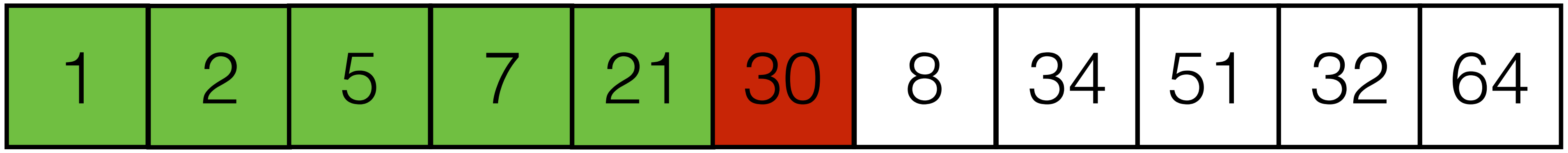
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps h_1, h_2, … ,h_t

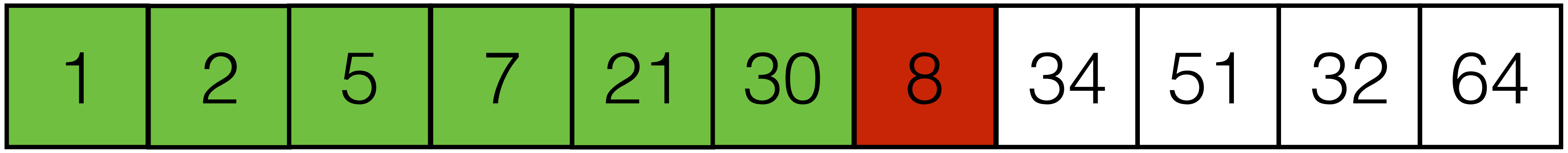


1	2	5	7	21	30	8	34	51	32	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$



1	2	5	7	21	30	8	34	51	32	64

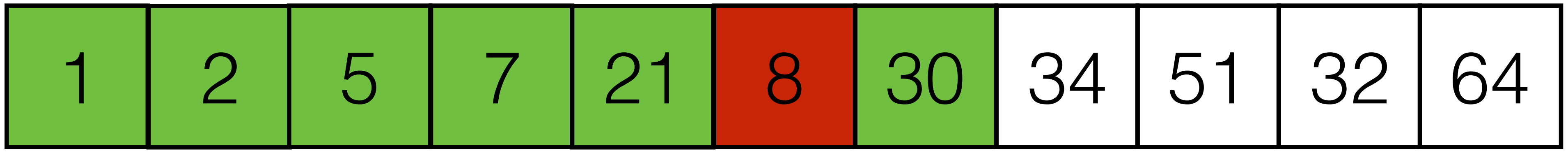
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



1	2	5	7	21	8	30	34	51	32	64

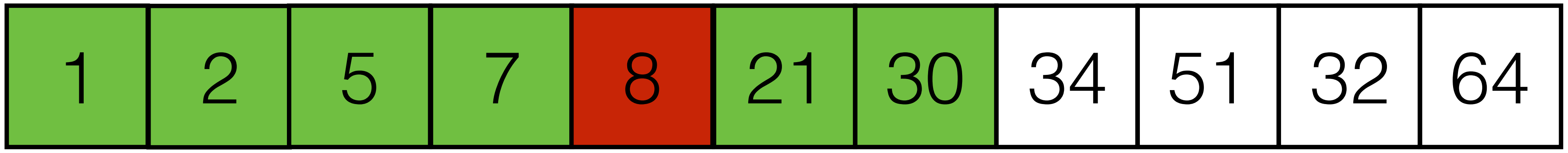
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



1	2	5	7	8	21	30	34	51	32	64

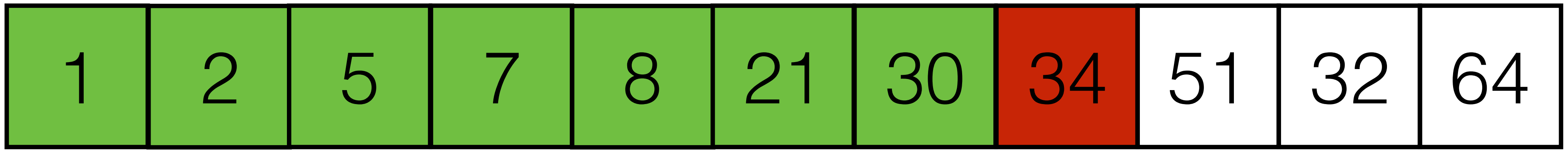
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



1	2	5	7	8	21	30	34	51	32	64

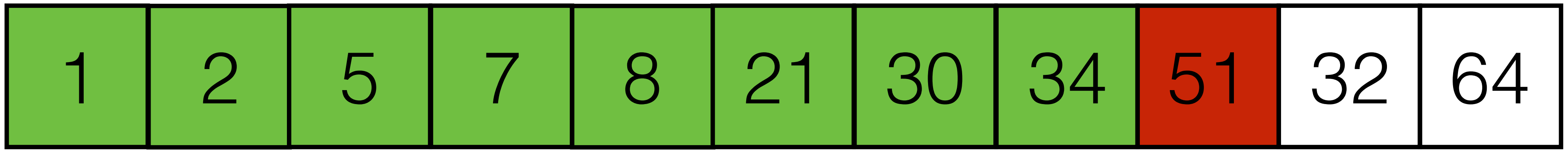
$h_3 = 5$ $h_2 = 3$ $\mathbf{h_1 = 1}$

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.

- "increment sequence" of steps $h_1, h_2, \ldots, h_t$

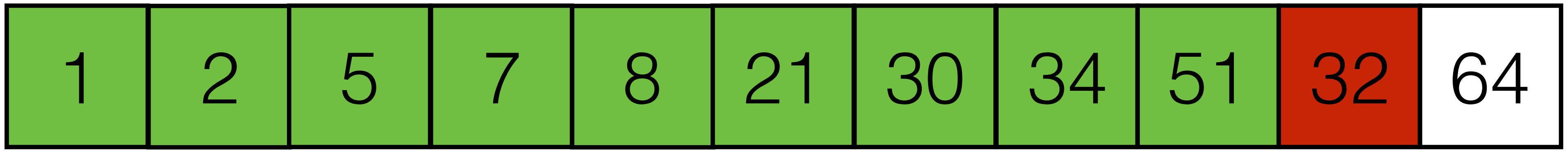


1	2	5	7	8	21	30	34	51	32	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

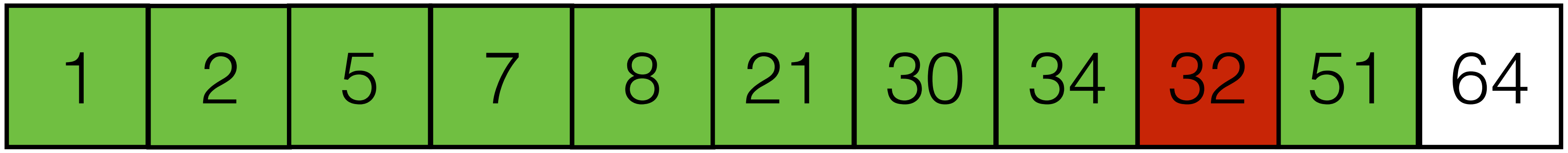
- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$



$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

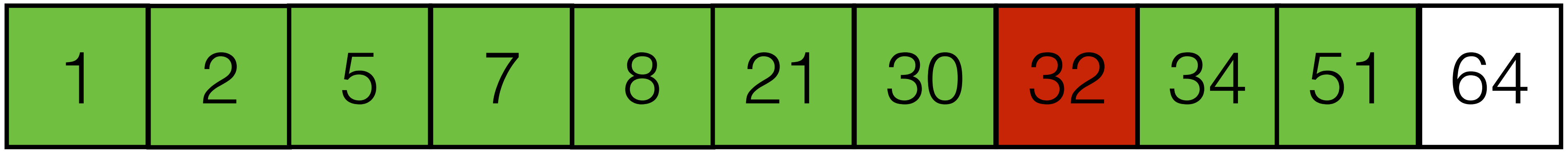


1	2	5	7	8	21	30	34	32	51	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

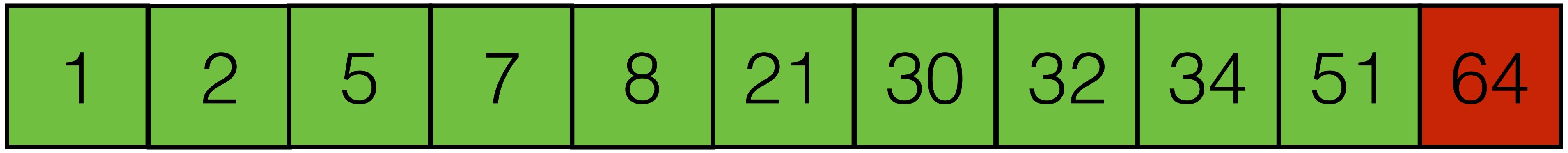


1	2	5	7	8	21	30	32	34	51	64

$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase k makes sure that all items space h_k apart are sorted.
- “increment sequence” of steps $h_1, h_2, \dots, h_t$



1	2	5	7	8	21	30	32	34	51	64

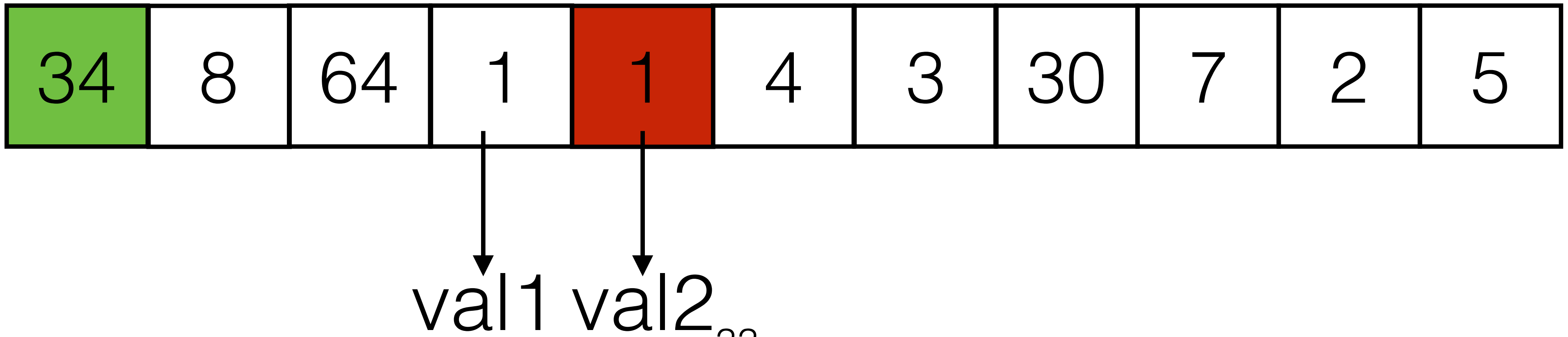
$h_3 = 5$ $h_2 = 3$ **$h_1 = 1$**

Shell Sort

- The running time analysis for shell sort is complex and depends on the specific increment sequence.
- With Hibbard's sequence (1,3,7,15,…,2^k-1) worst case running time is $\Theta(N^{3/2})$

Sorting Stability

- Assume we put key/value pairs sorted by keys into the array.
  - Shell Sort is *unstable*: keys will be sorted, but values for the same key may be in different order than in the input.

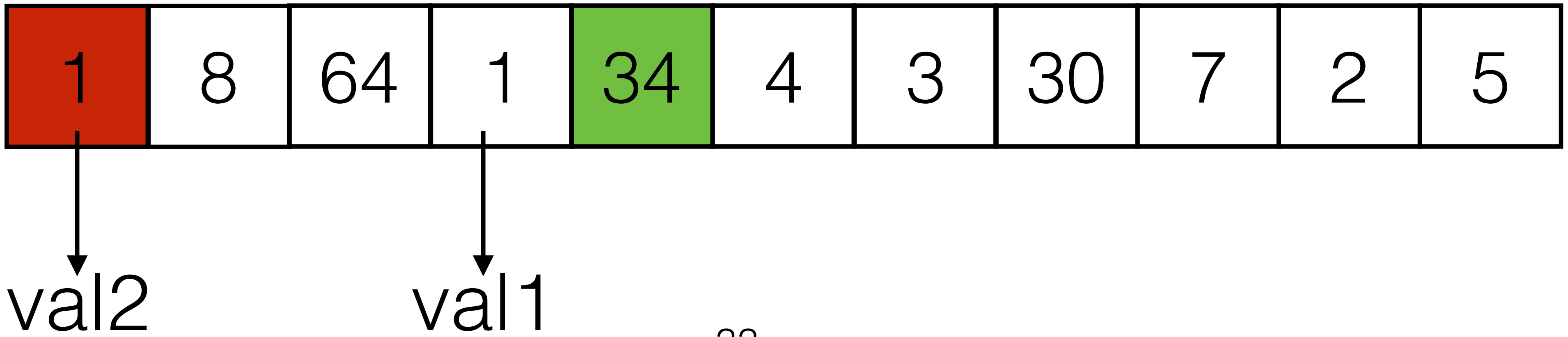


34	8	64	1	1	4	3	30	7	2	5

val1 val2

Sorting Stability

- Assume we put key/value pairs sorted by keys into the array.
  - Shell Sort is *unstable*: keys will be sorted, but values for the same key may be in different order than in the input.



1	8	64	1	34	4	3	30	7	2	5

val2

val1

Space Requirements

- Both Insertion Sort and Shell Sort operate in place.
- Only a small amount of memory required to store a temporary value for swaps.
- Space requirement: O(1)

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)

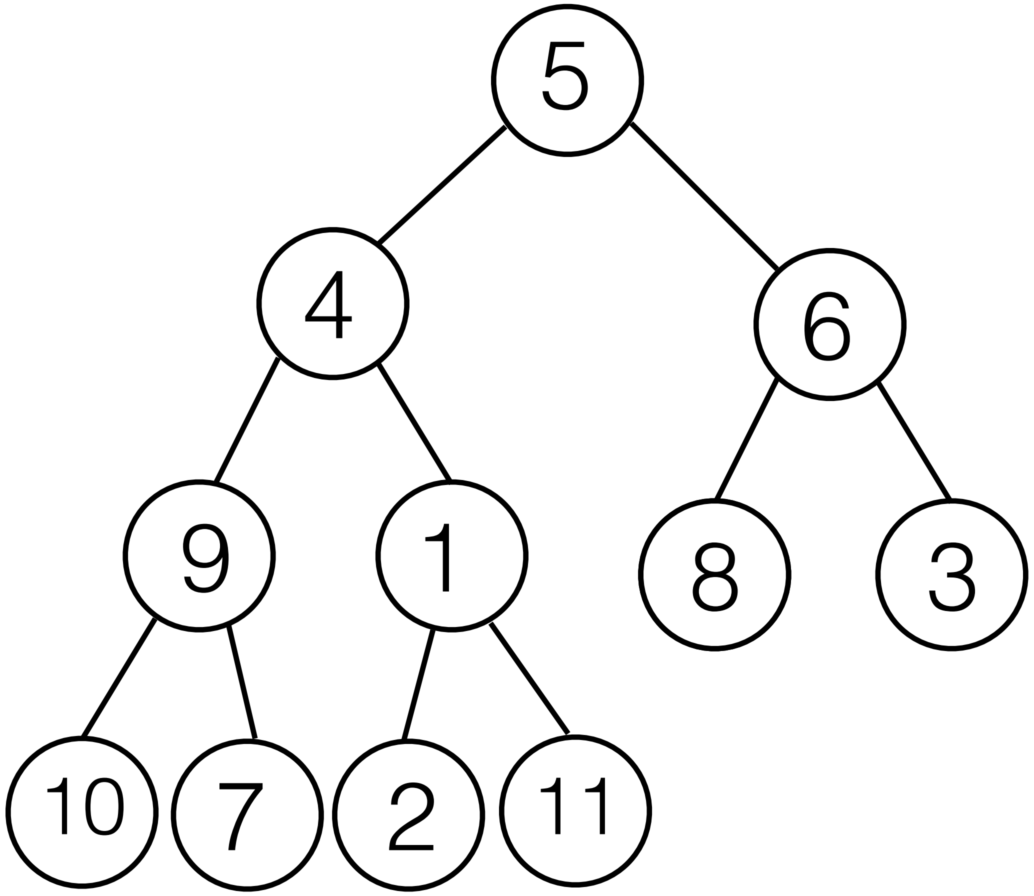
Heap Sort

- 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

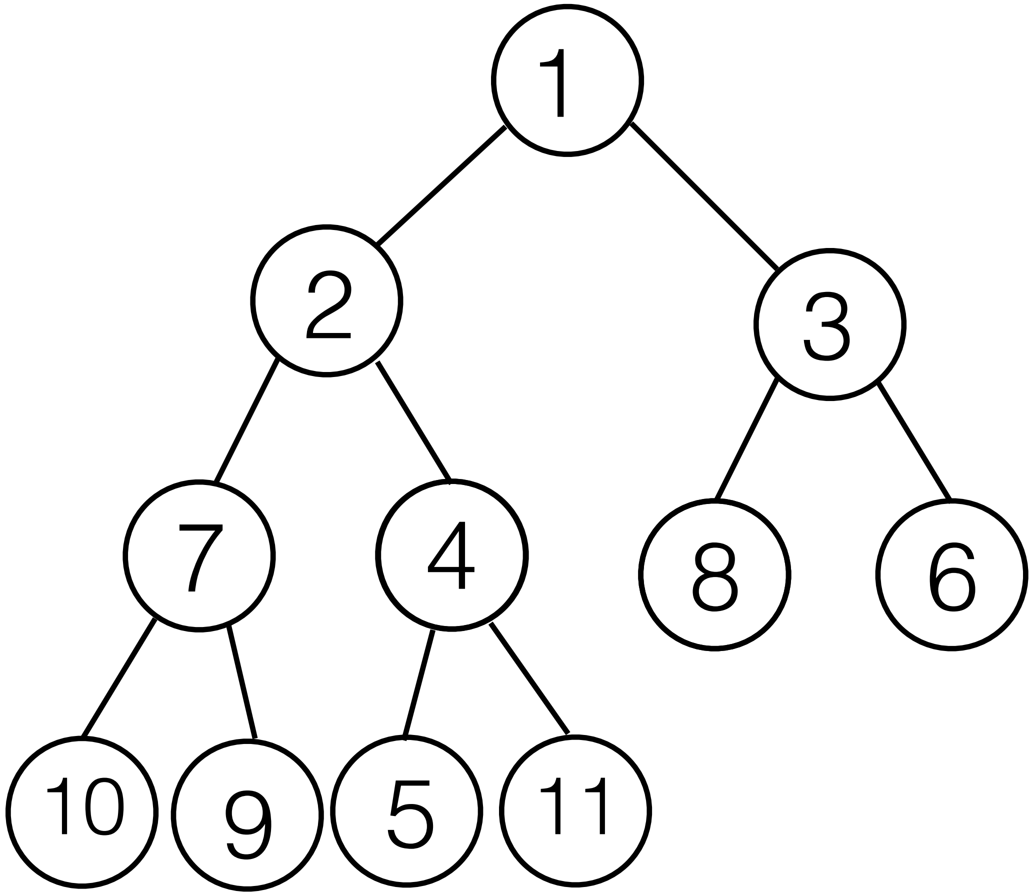
Heap Sort Example



	5	4	6	9	1	8	3	10	7	2	11

Heap Sort Example

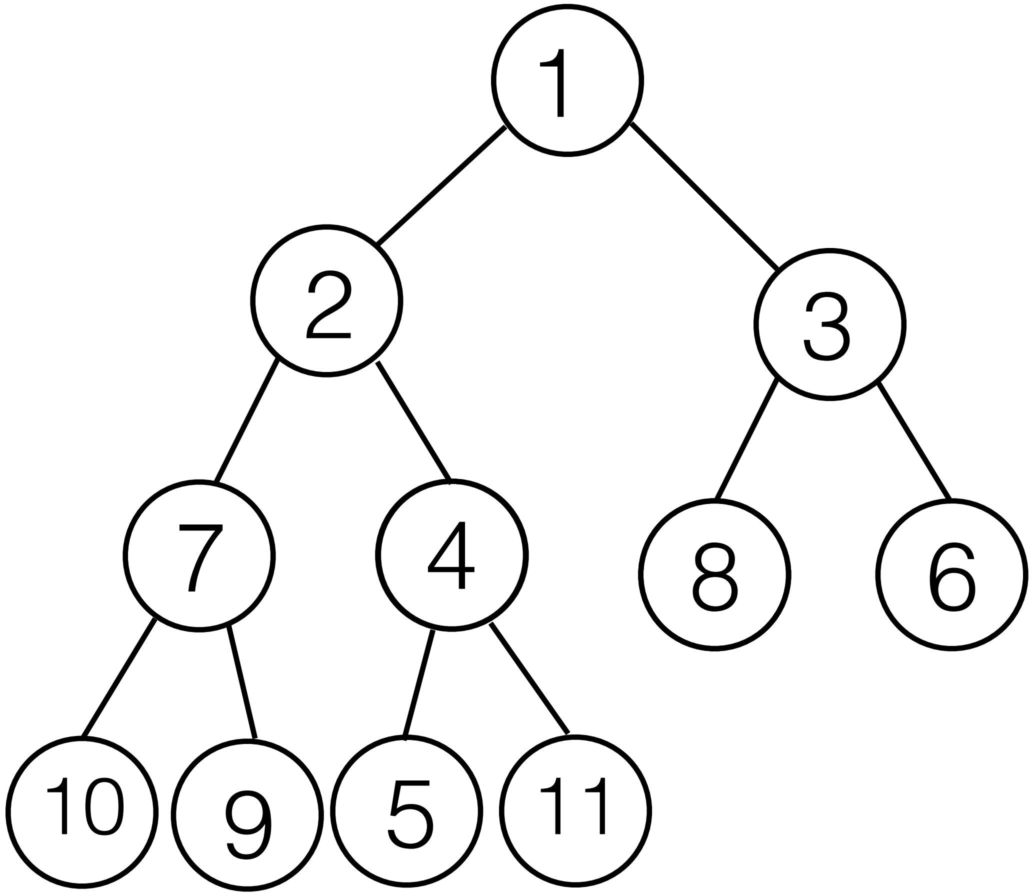
Build heap in O(N) time



1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

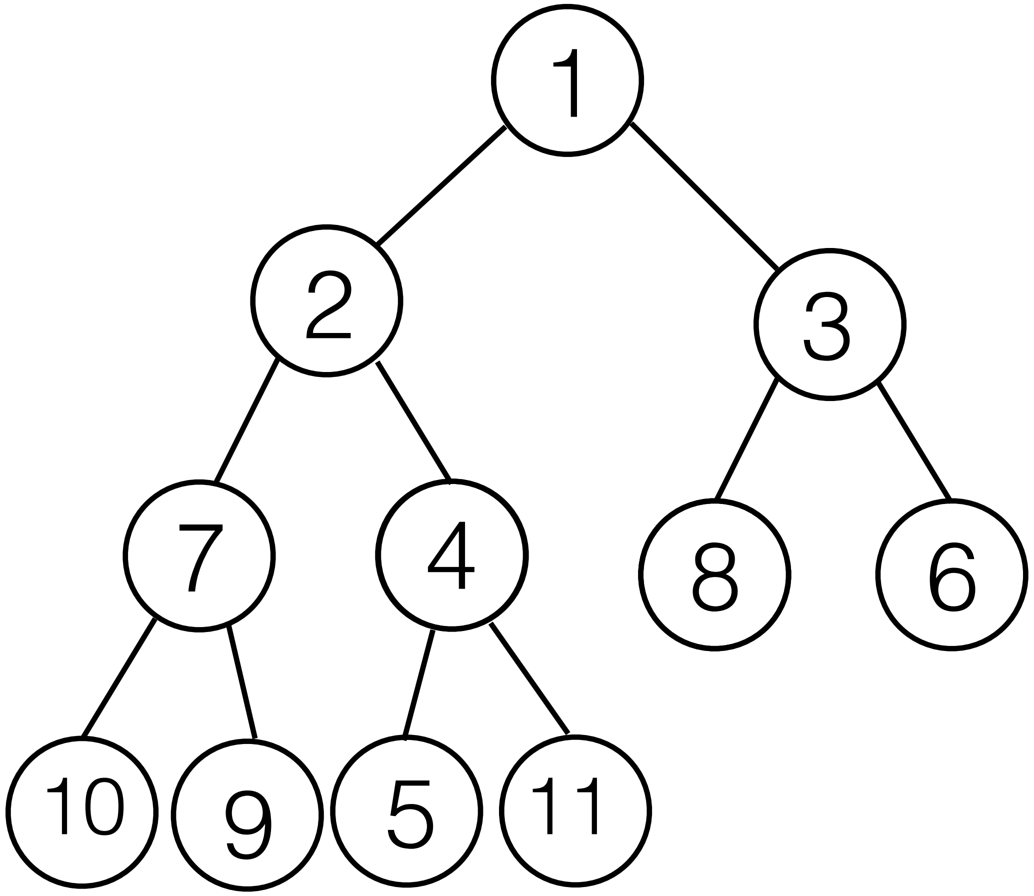
Build heap in O(N) time



	1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

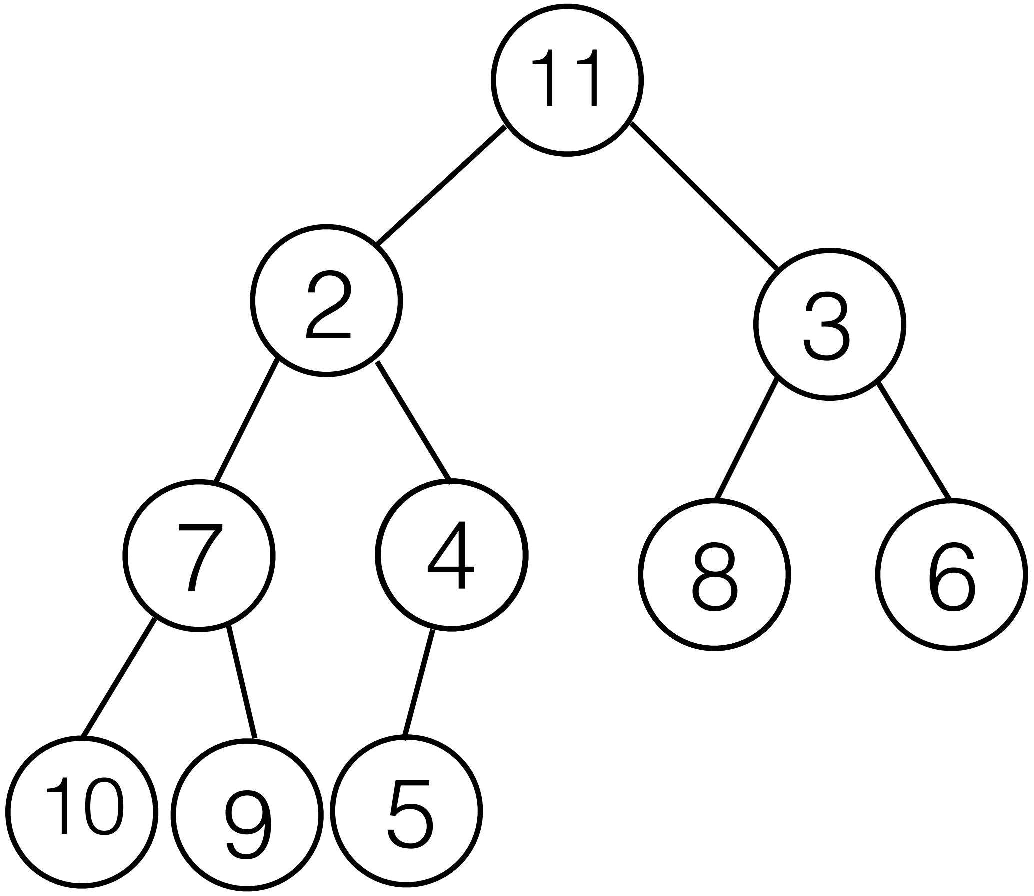
deleteMin, write min element into empty cell



	1	2	3	7	4	8	6	10	9	5	11

Heap Sort Example

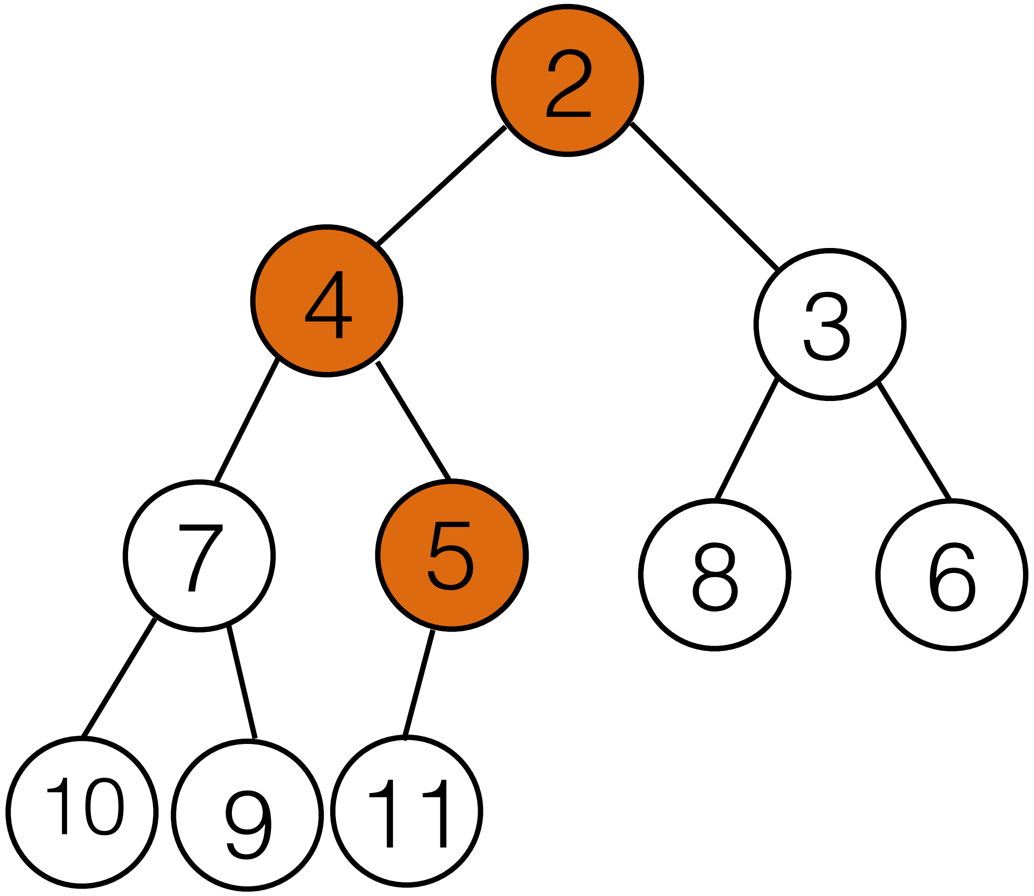
deleteMin, write min element into empty cell



	11	2	3	7	4	8	6	10	9	5	1

Heap Sort Example

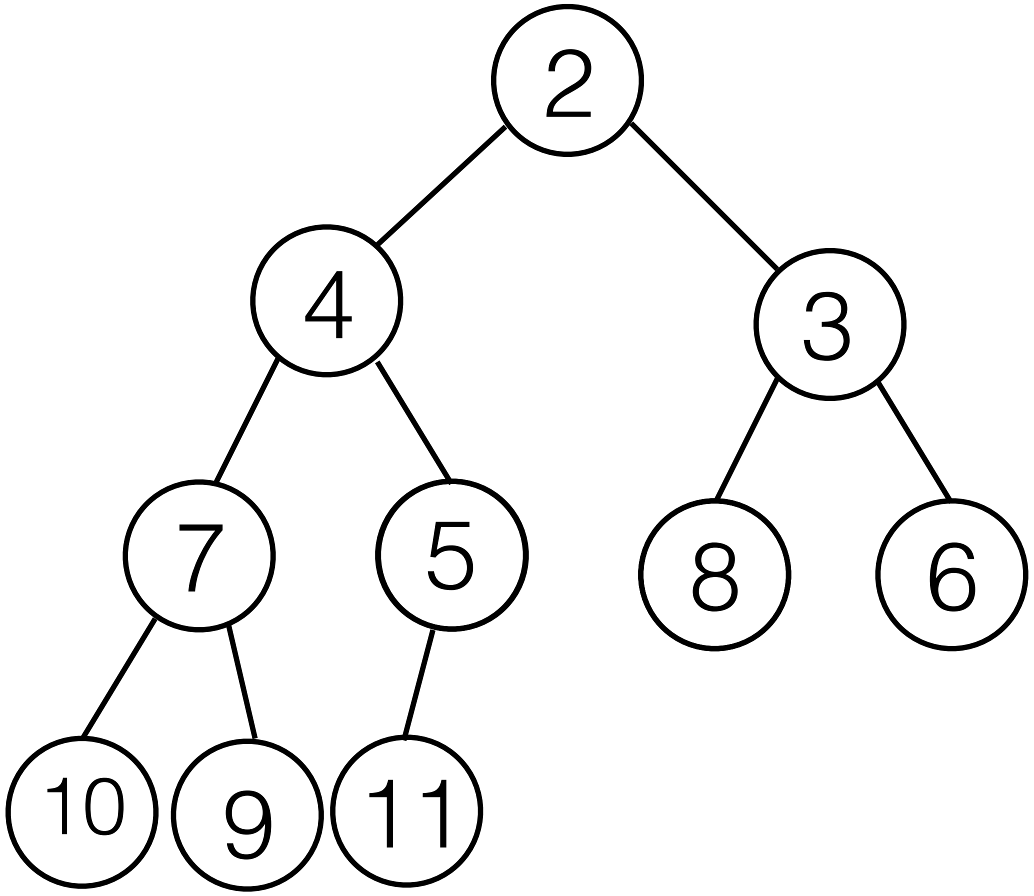
Percolate down



	2	4	3	7	5	8	6	10	9	11	1

Heap Sort Example

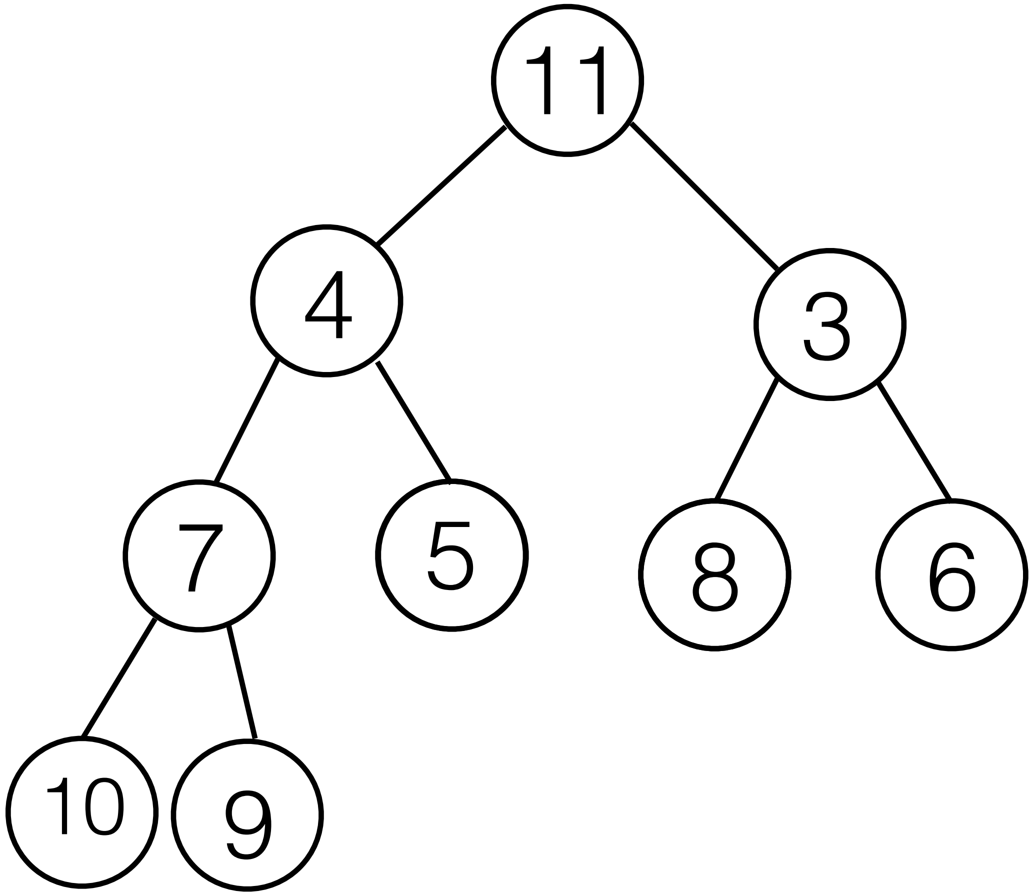
deleteMin, write min element into empty cell



	2	4	3	7	5	8	6	10	9	11	1

Heap Sort Example

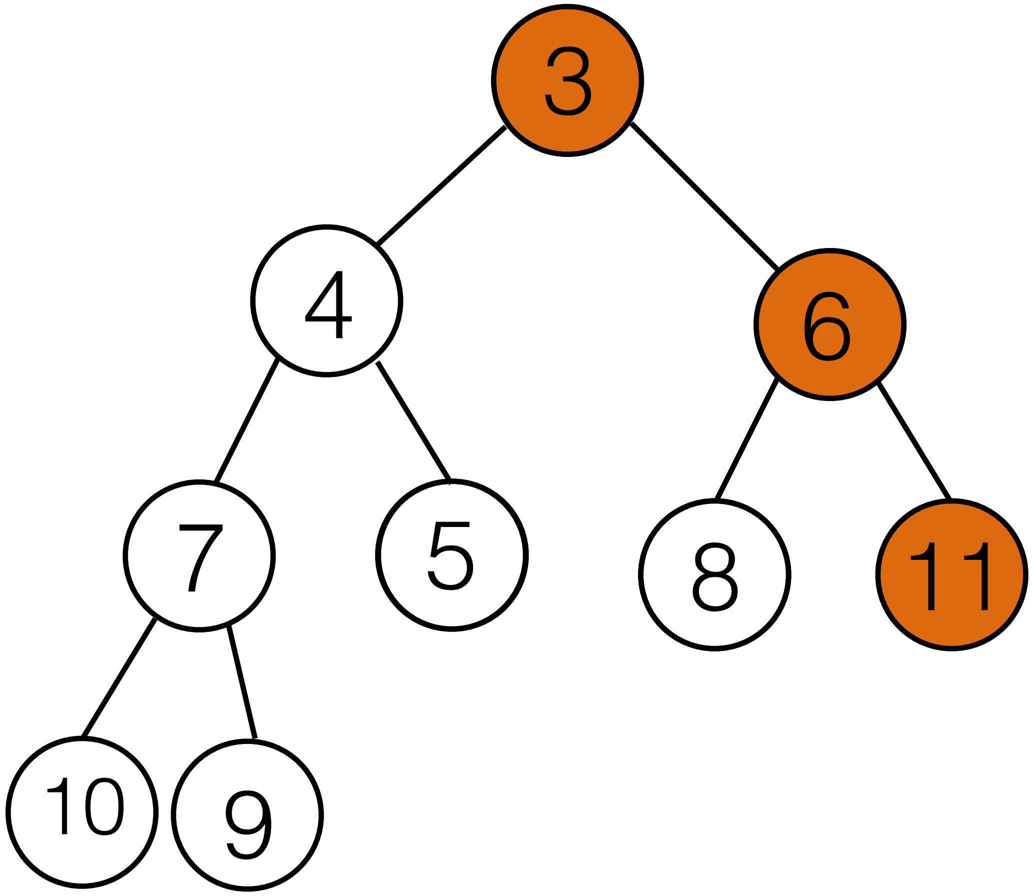
deleteMin, write min element into empty cell



	11	4	3	7	5	8	6	10	9	2	1

Heap Sort Example

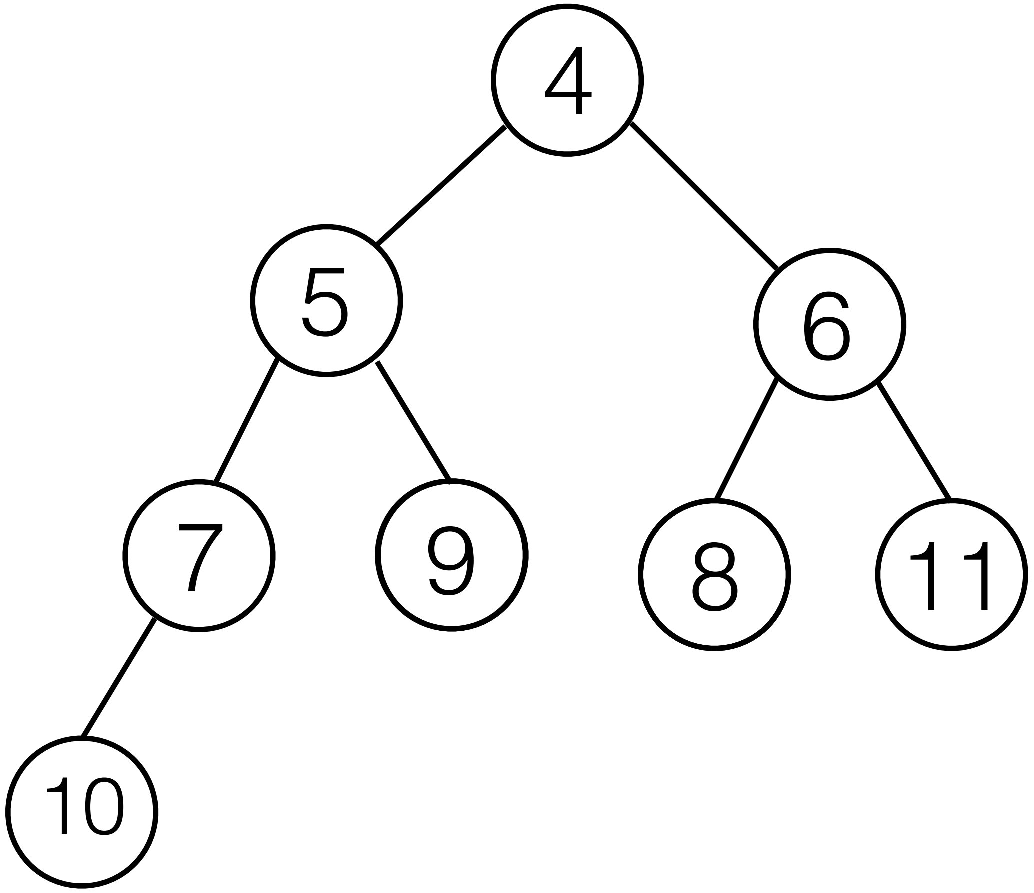
Percolate down



	3	4	6	7	5	8	11	10	9	2	1

Heap Sort Example

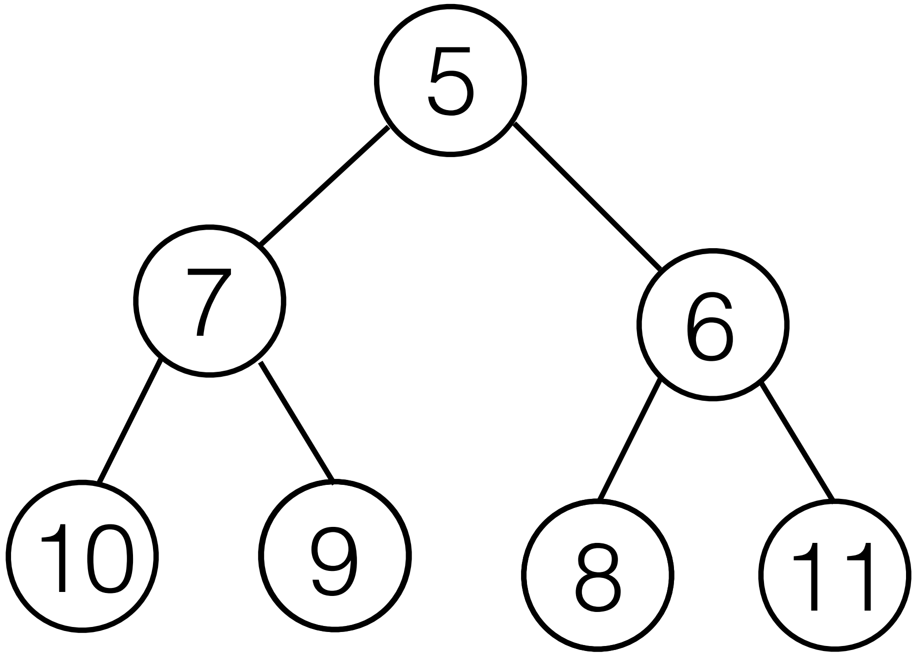
deleteMin, write min element into empty cell



	4	5	6	7	9	8	11	10	3	2	1

Heap Sort Example

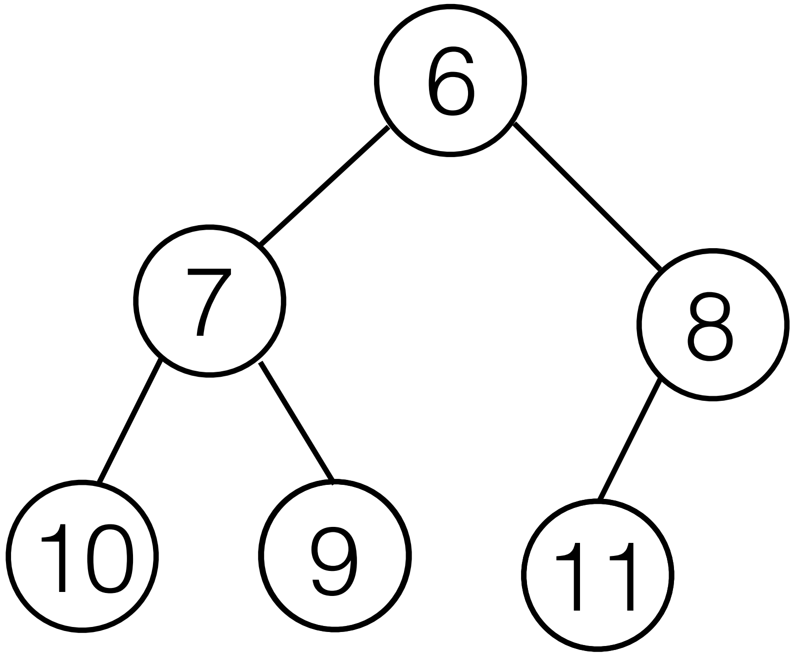
deleteMin, write min element into empty cell



	5	7	6	10	9	8	11	4	3	2	1

Heap Sort Example

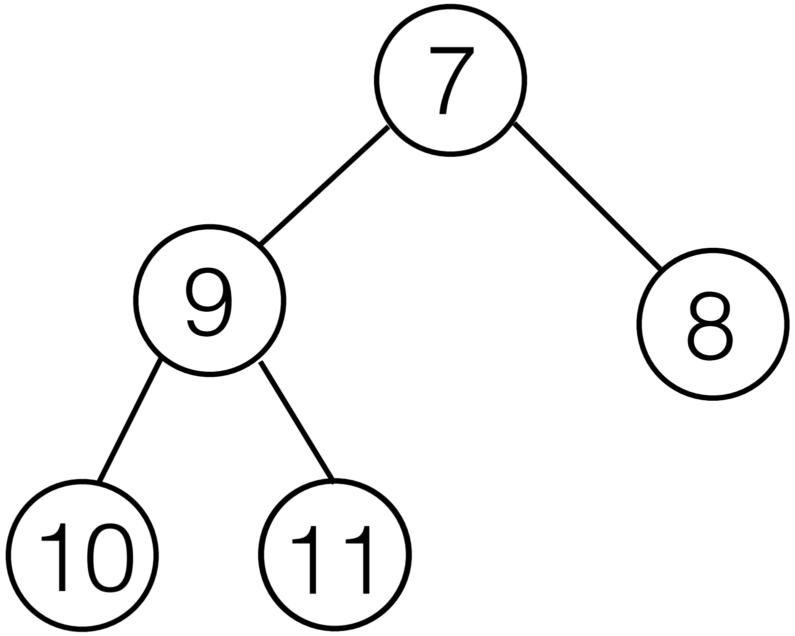
deleteMin, write min element into empty cell



	6	7	8	10	9	11	5	4	3	2	1

Heap Sort Example

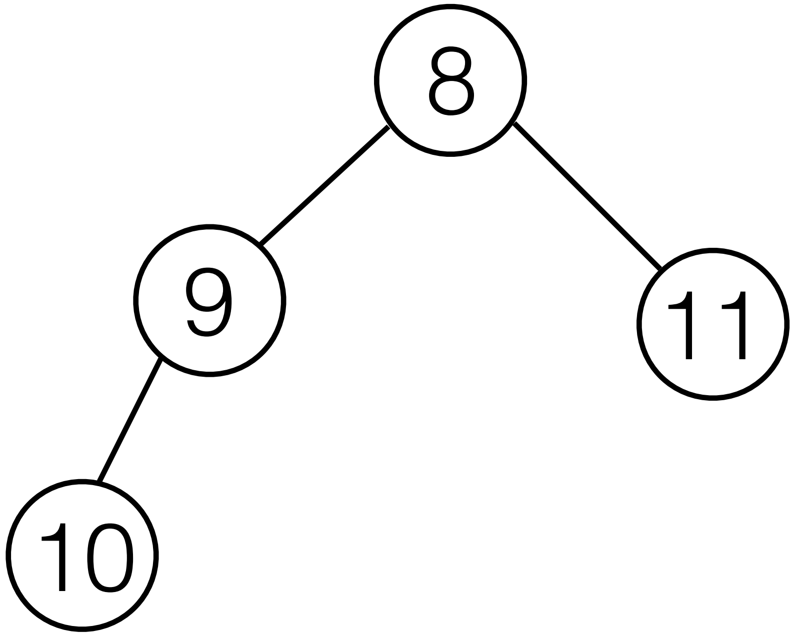
deleteMin, write min element into empty cell



	7	9	8	10	11	6	5	4	3	2	1

Heap Sort Example

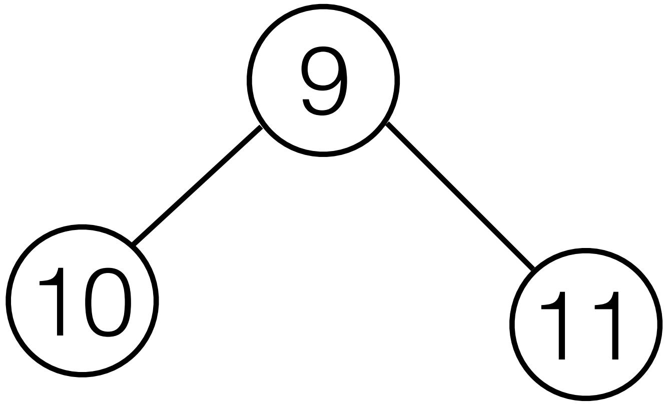
deleteMin, write min element into empty cell

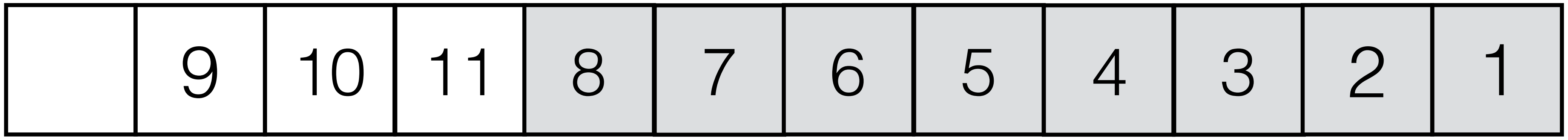


	8	9	11	10	7	6	5	4	3	2	1

Heap Sort Example

deleteMin, write min element into empty cell

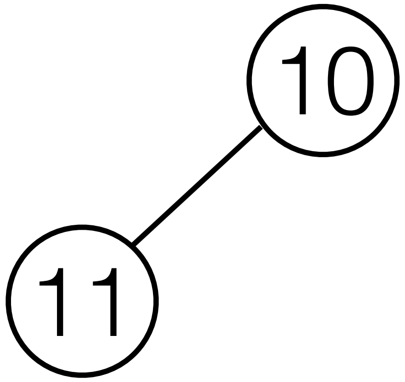




	9	10	11	8	7	6	5	4	3	2	1

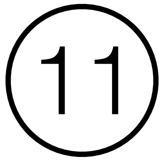
Heap Sort Example

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

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.

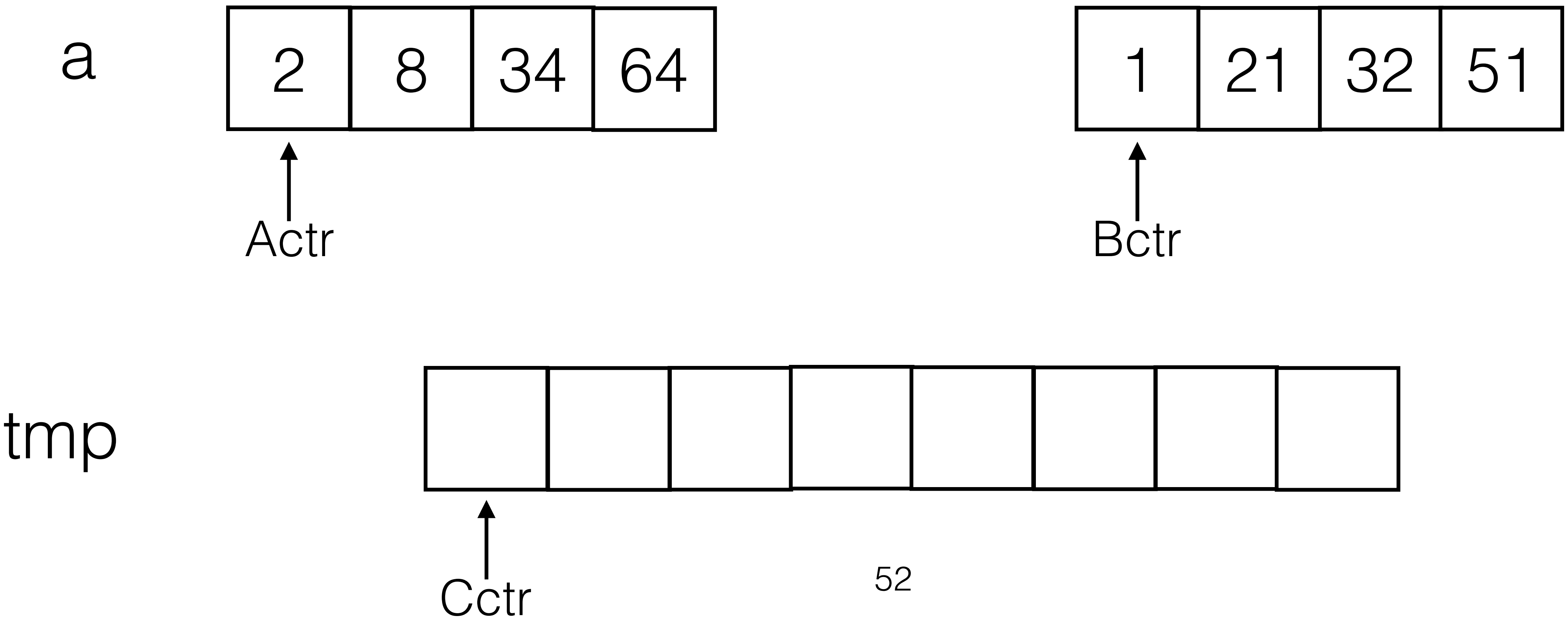
2	8	34	64

1	21	32	51

1	2	8	21	32	34	51	64

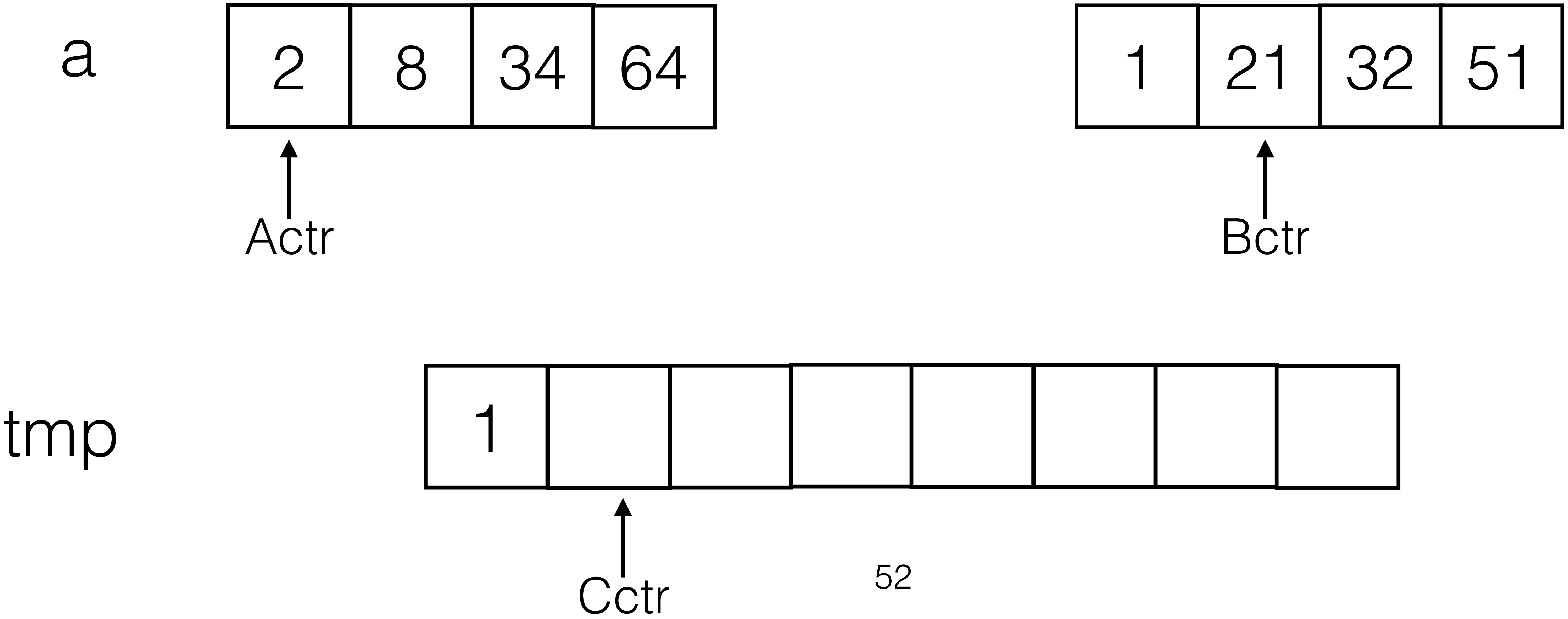
Merging Sorted Sublists

- 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.



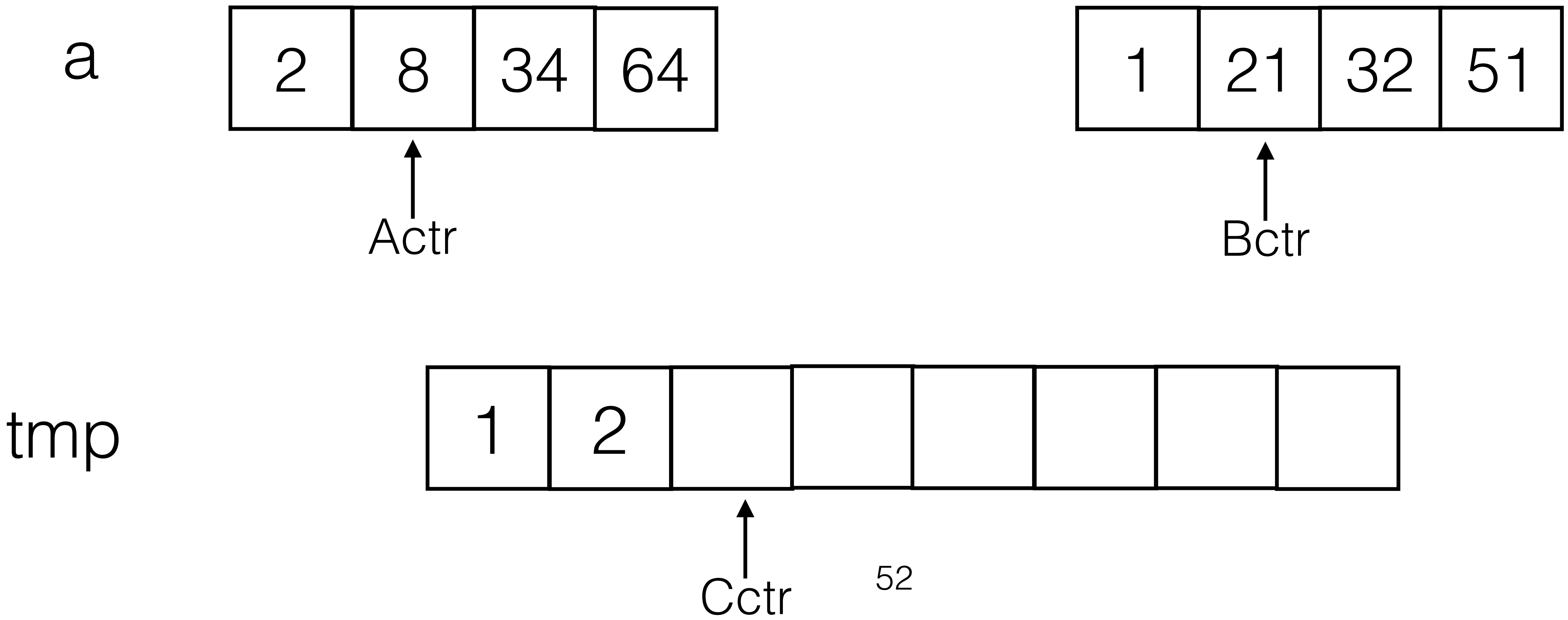
Merging Sorted Sublists

- 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.



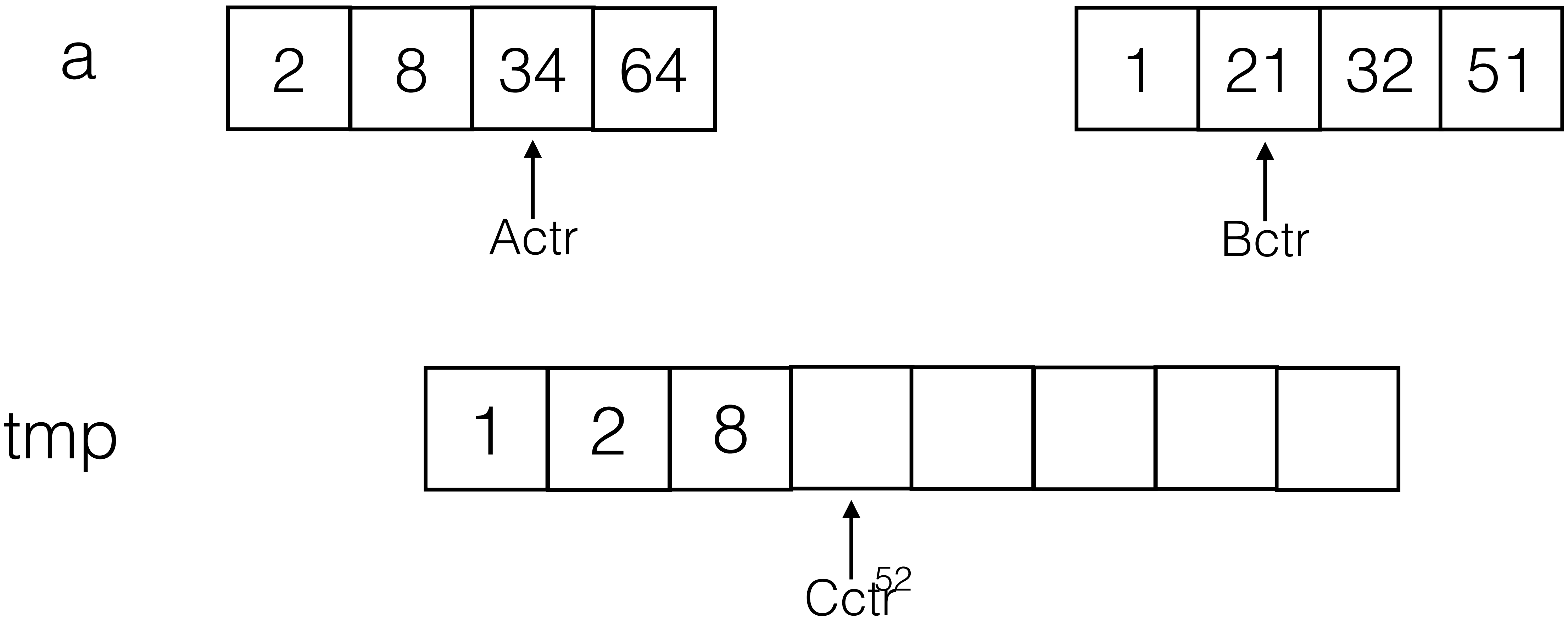
Merging Sorted Sublists

- 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.

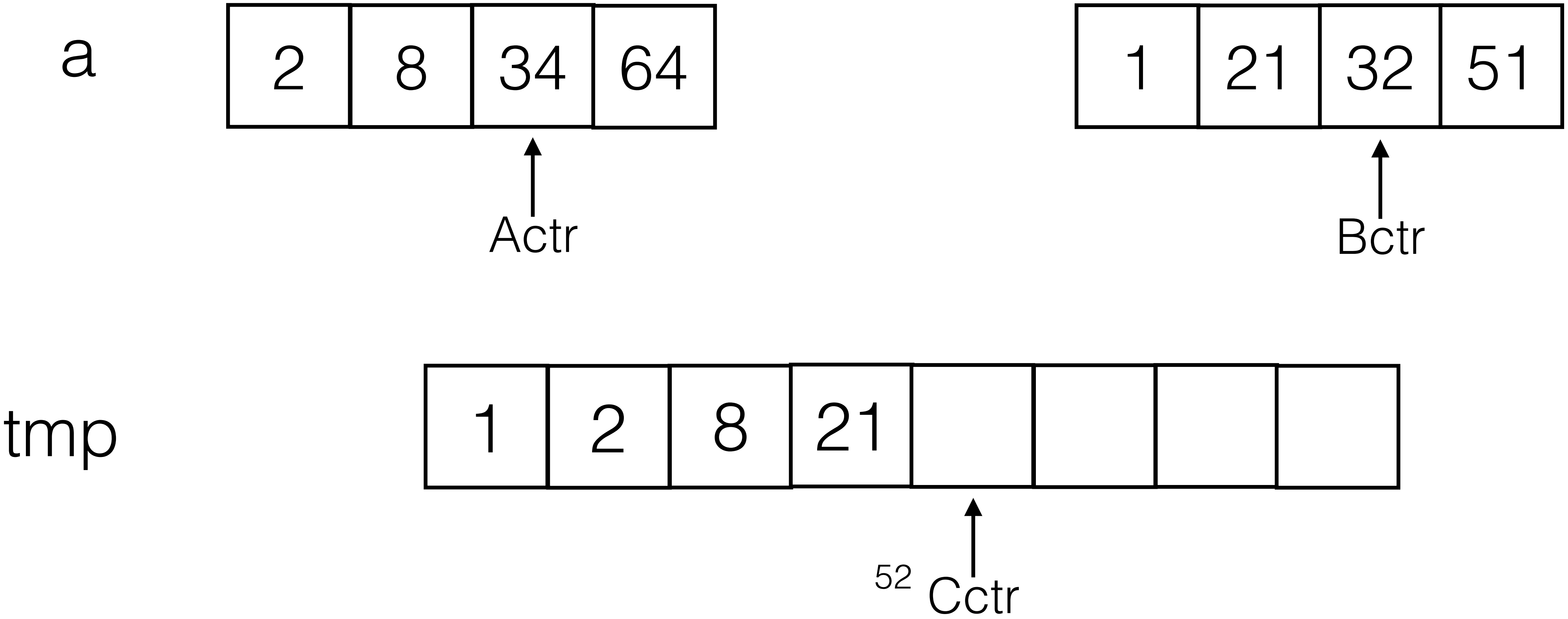


a: 2 | 8 | 34 | 64 … 1 | 21 | 32 | 51

tmp: 1 | 2 | 8 | | | | |

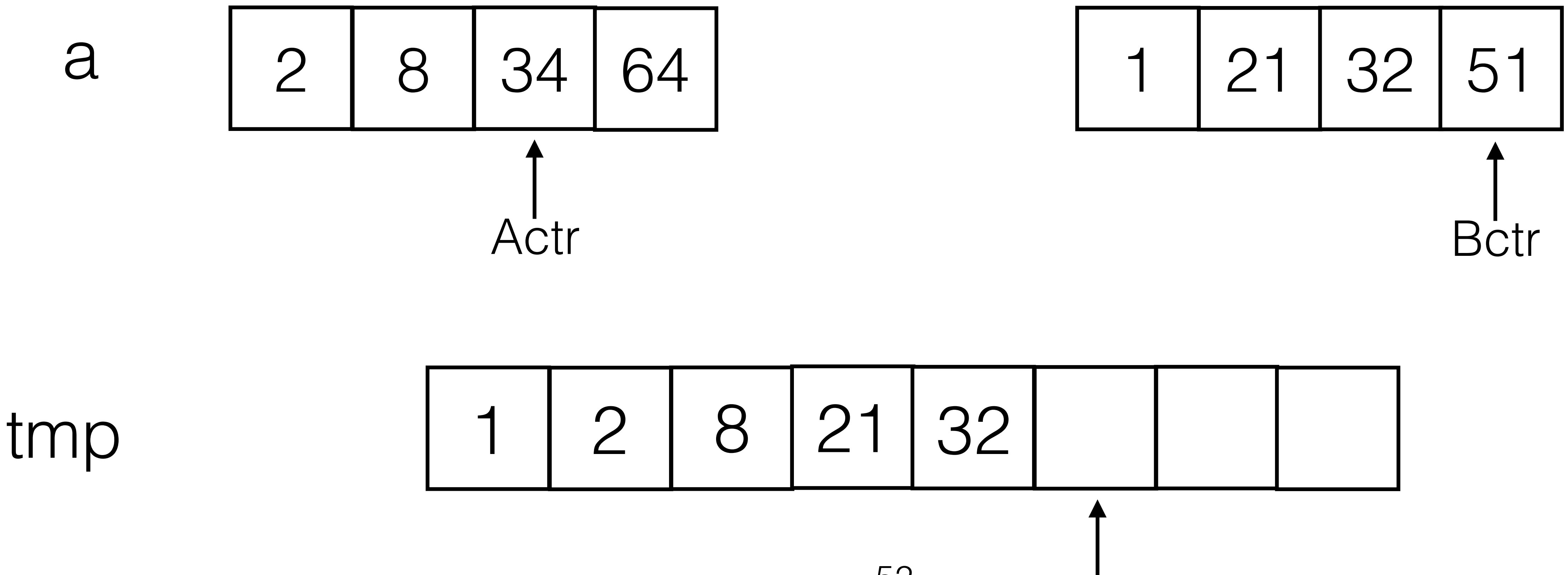
Merging Sorted Sublists

- 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.



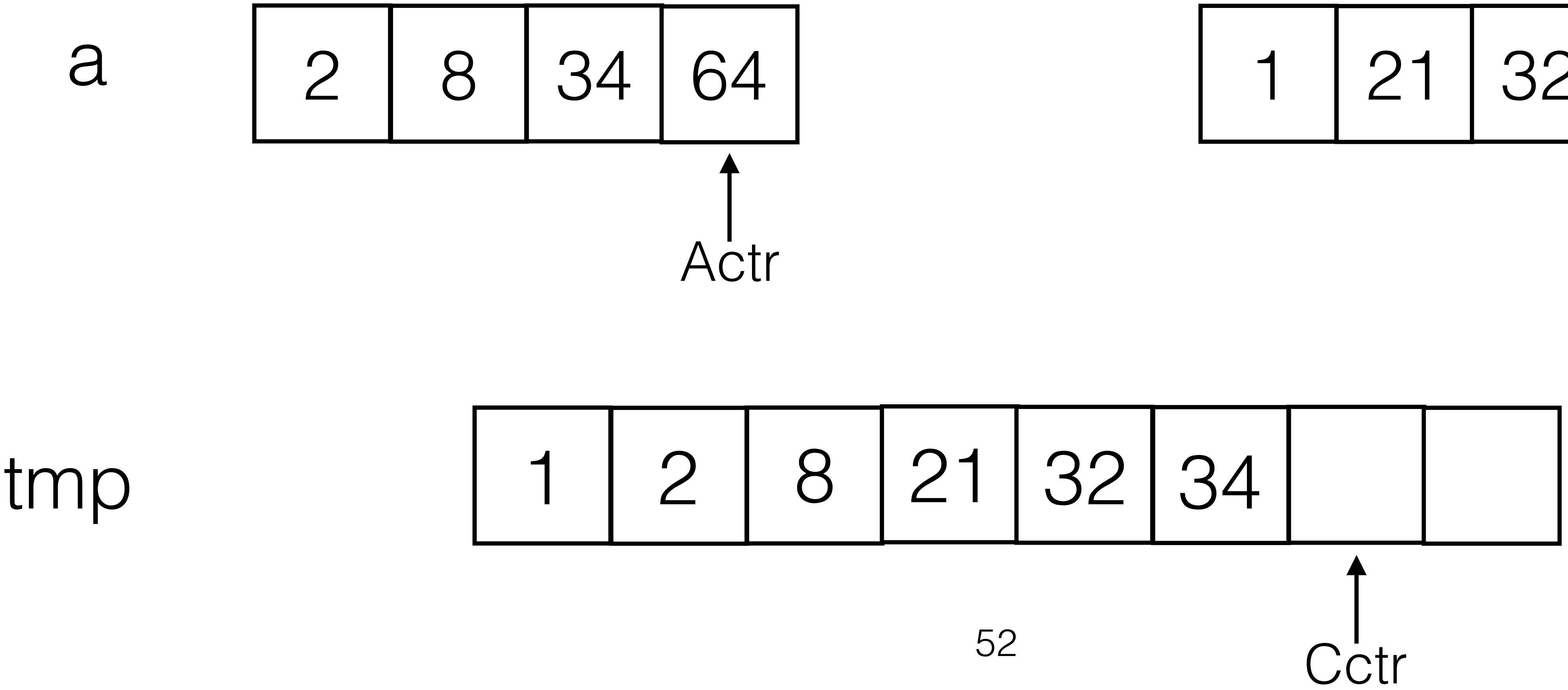
Merging Sorted Sublists

- 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.



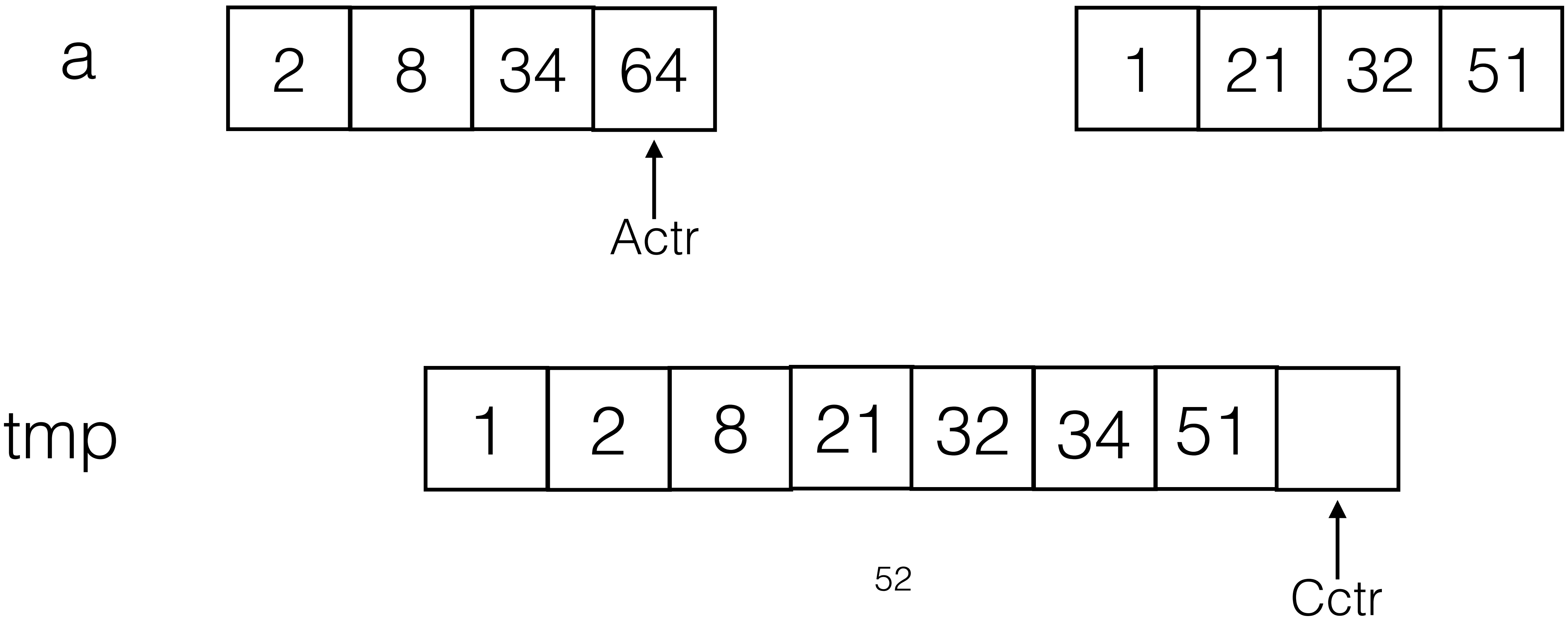
Merging Sorted Sublists

- 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.



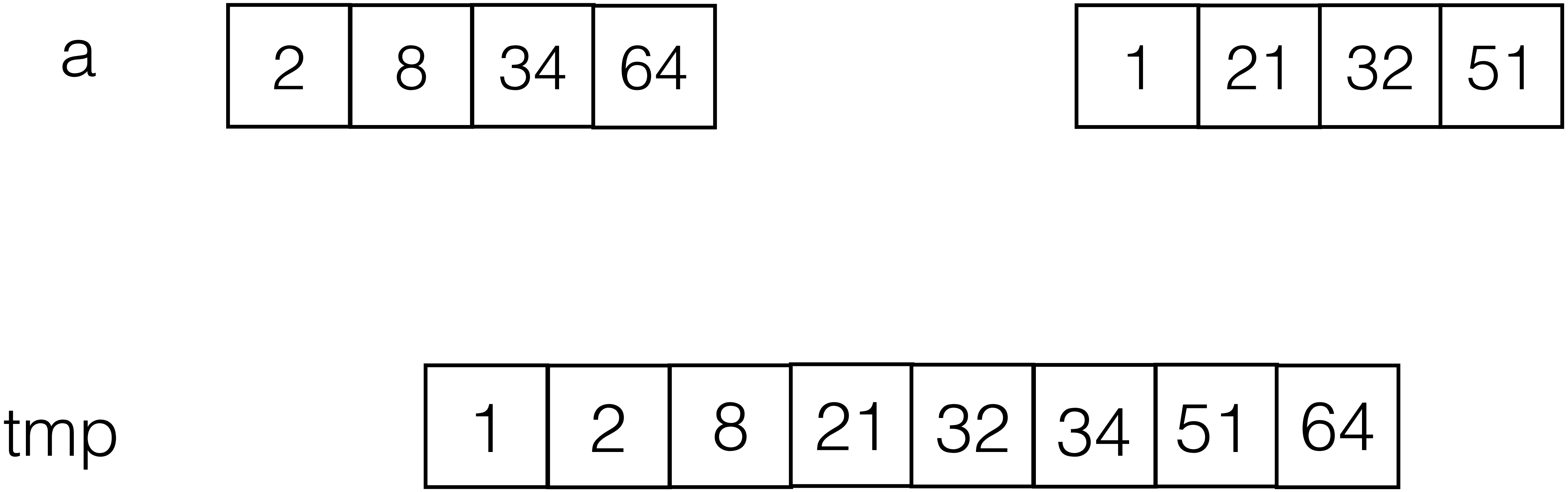
Merging Sorted Sublists

- 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

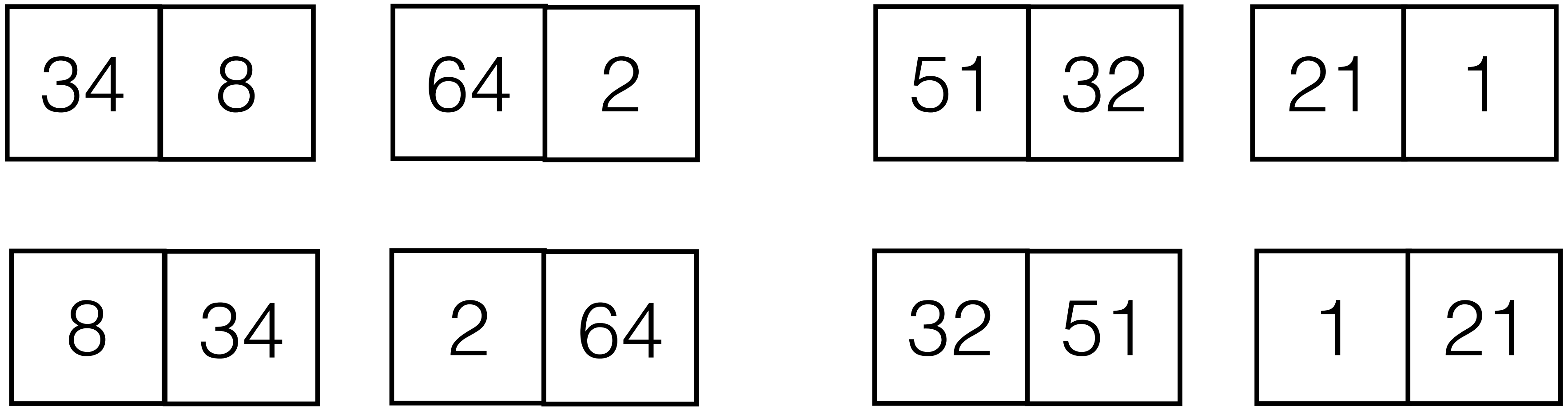
64	2

51	32

21	1

Merge Sort

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

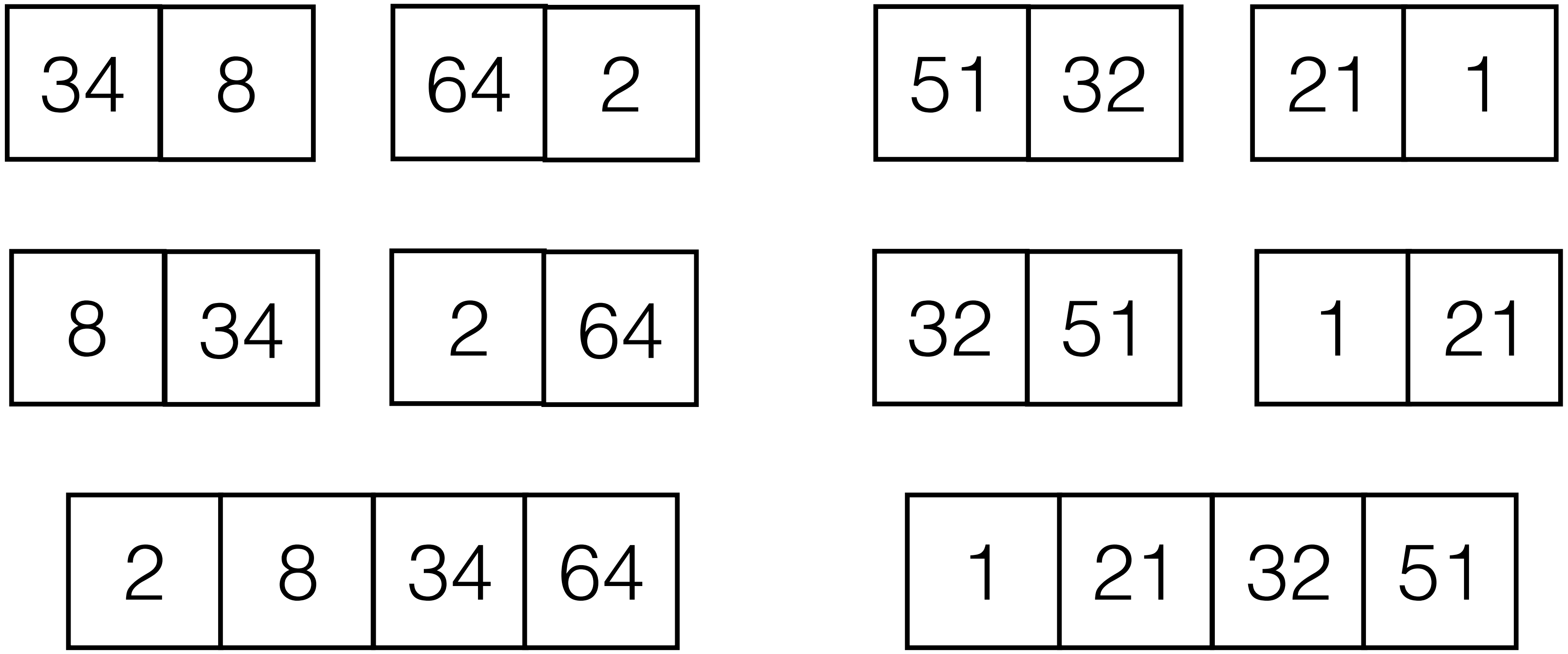


34 8 64 2 51 32 21 1

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

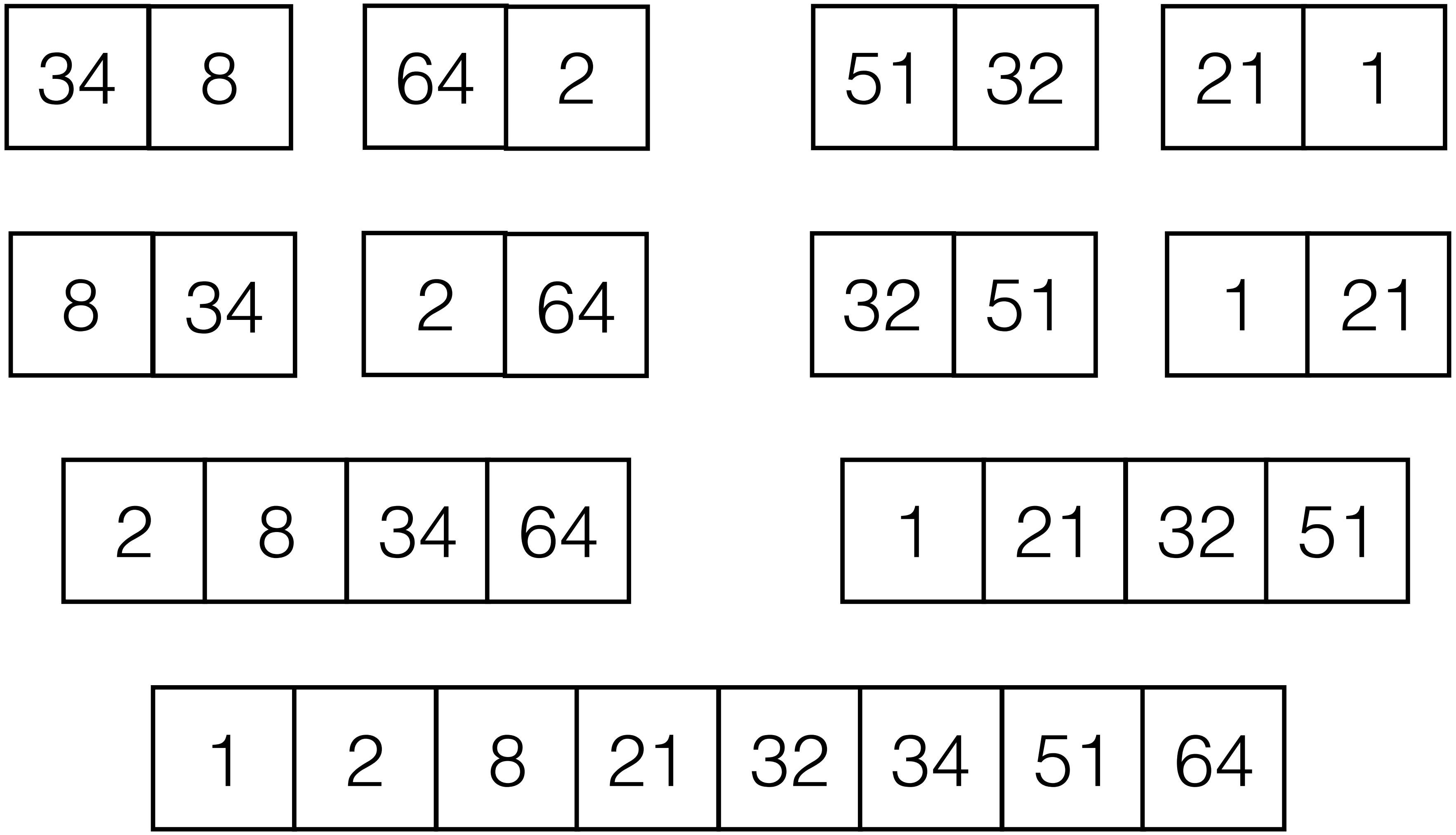
8 34 2 64 32 51 1 21

2 8 34 64 1 21 32 51

Merge Sort

- Split the array in half, recursively sort each half.

- Merge the two sorted lists.



34 8 | 64 2 | 51 32 | 21 1

8 34 | 2 64 | 32 51 | 1 21

2 8 34 64 | 1 21 32 51

1 2 8 21 32 34 51 64

Merge Sort - Implementation

```java
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: $T(N) = 2\,T(N/2) + N$

Recursively sort each half

Merge the two halfs

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)