CHAPTER 6

Sorting and Selection

Algorithm Bubble Sort

This algorithm compares consecutive elements, bubbling the largest elements to the end of the array.

```
bubbleSort(a) {
    n = a.last
    for i = n downto 1 {
        for j = 1 to i-1 {
            if a[j] > a[j+1] {
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp
            }
        }
    }
}
```

NOTE: This is a “traditional” version of BubbleSort. A revised version uses a boolean variable to track whether or not at least one swap has occurred in a pass through the array, thereby eliminating unnecessary passes through the array.

Algorithm Selection Sort

This algorithm sorts the array *a* by first finding the minimum value in the array and swaps it with the first element in the array; then finds the next smallest and swaps it with the second element in the array; and so on.

```
selectionSort(a) {
    n = a.last
    for i = 1 to n-1 {
        minPos = i
        for j = i+1 to n {
            if a[j] < a[minPos]
                minPos = j
        }
        if minPos != i {
            temp = a[minPos]
            a[minPos] = a[i]
            a[i] = temp
        }
    }
}
```

Algorithm 6.1.2 Insertion Sort

This algorithm sorts the array a by first inserting $a[2]$ into the sorted subarray $a[1]$; next inserting $a[3]$ into the sorted subarray $a[1]$, $a[2]$; and so on; and finally inserting $a[n]$ into the sorted subarray $a[1], \ldots, a[n - 1]$.

```
insertionSort(a) {
   n = a.last
   for i = 2 to n {
      val = a[i]            // save a[i] so it can be inserted
      j = i - 1             // into the correct place
      while (j >= 1 && val < a[j]) {
         a[j + 1] = a[j]    // move larger elements down by
         j = j - 1          // one to make room for val
      }
      a[j + 1] = val        // insert val in the correct place
   }
}
```

Algorithm Merge Sort

This algorithm uses a divide and conquer technique. The array is split in half, the halves are recursively sorted and then merged together.

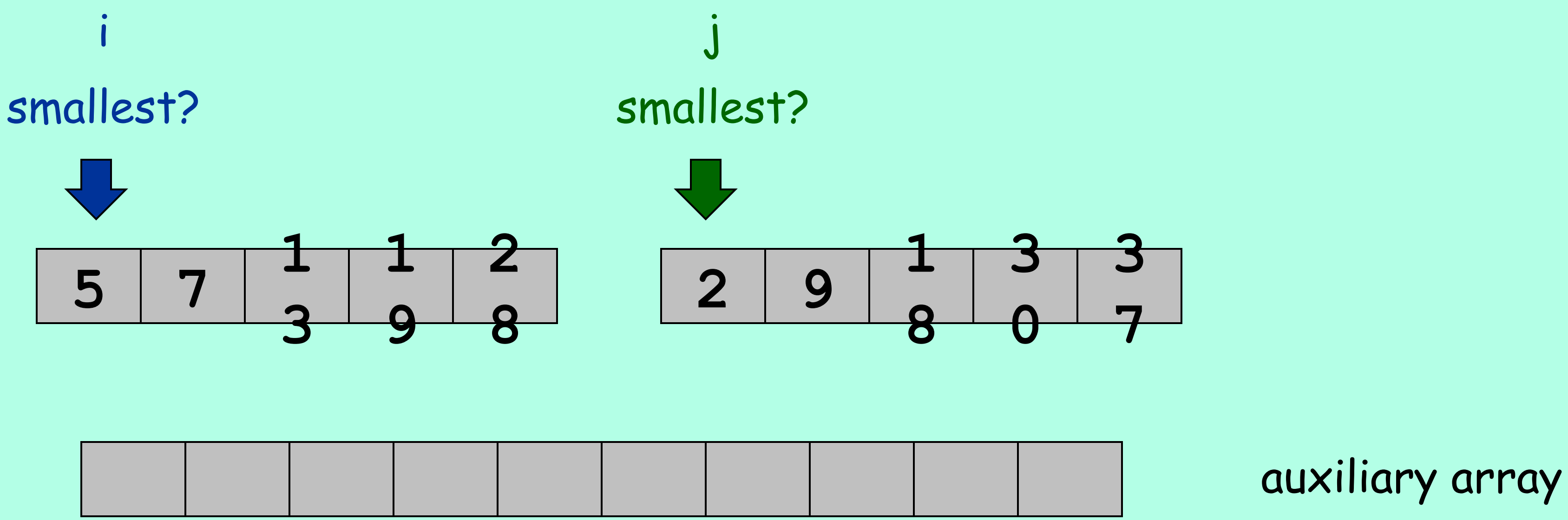
```
mergeSort(a, i, j)
  if (i == j)
    return
  m = (i + j) / 2
  mergeSort(a, i, m)
  mergeSort(a, m+1, j)
  merge(a, i, m, j)

merge(a, i, m, j)
  // merge two sorted subarrays into one
  // see next slides
```

Merging

Merge.

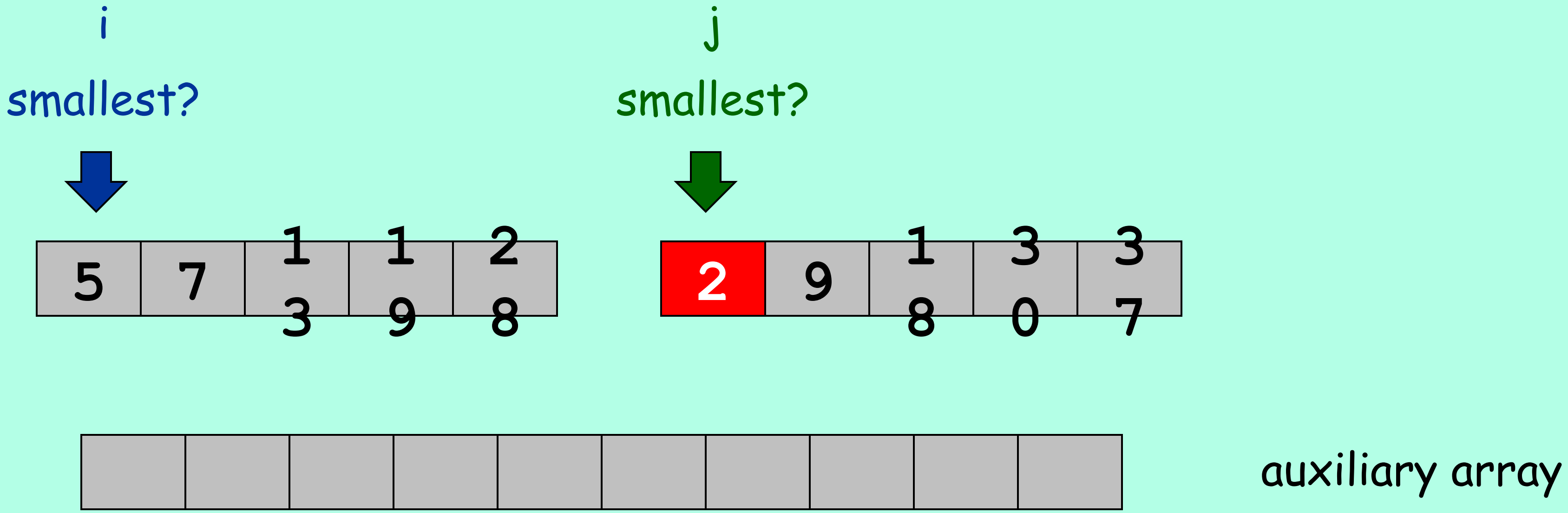
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

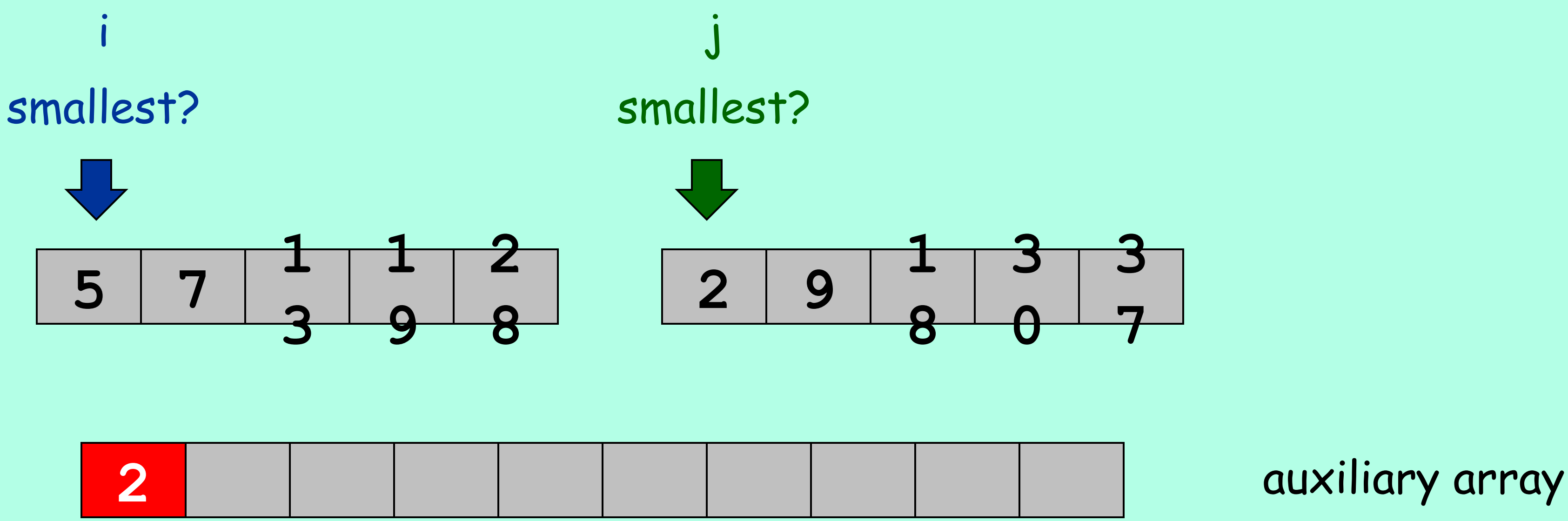
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

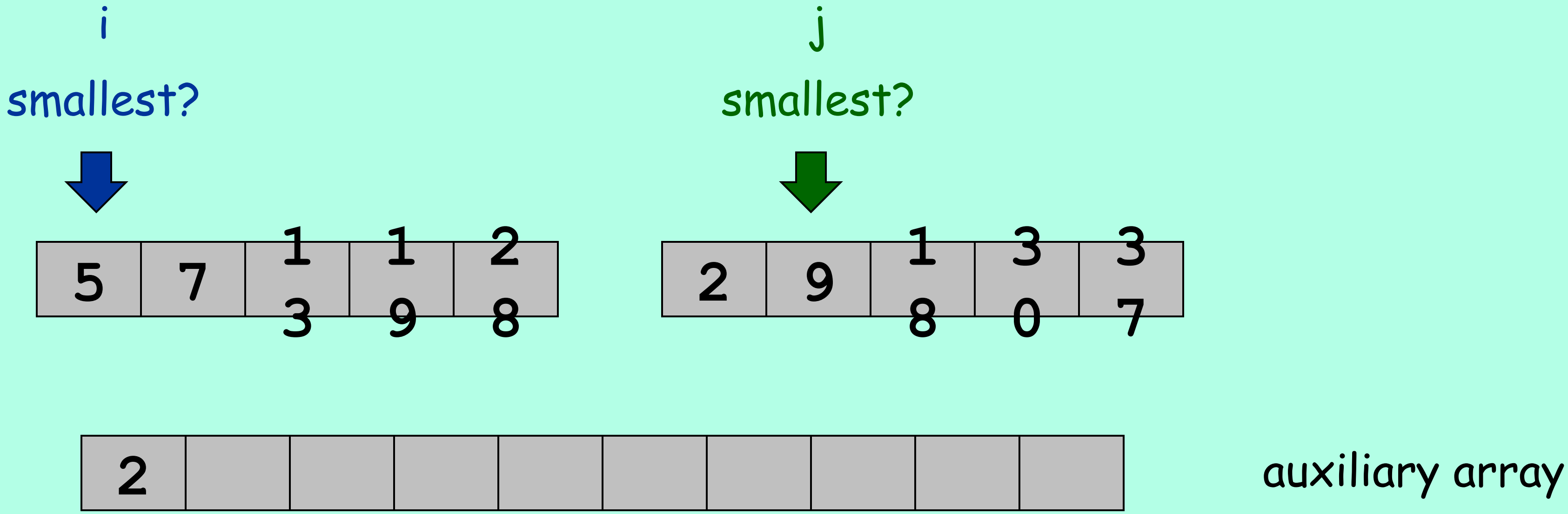
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

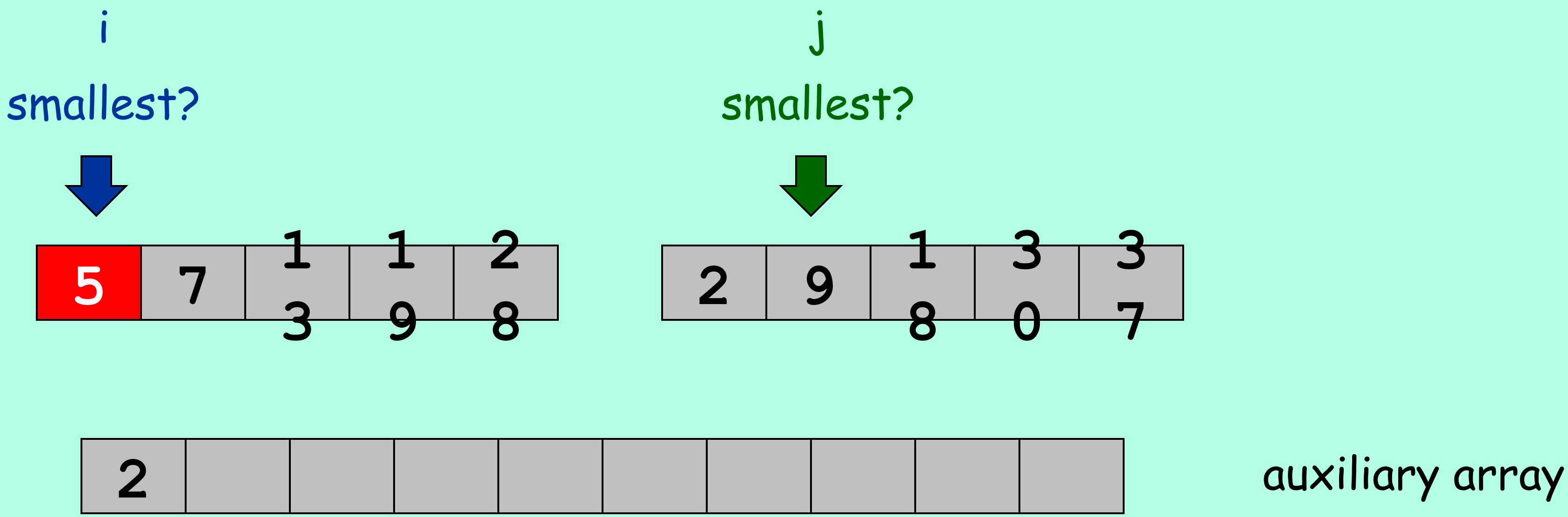
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

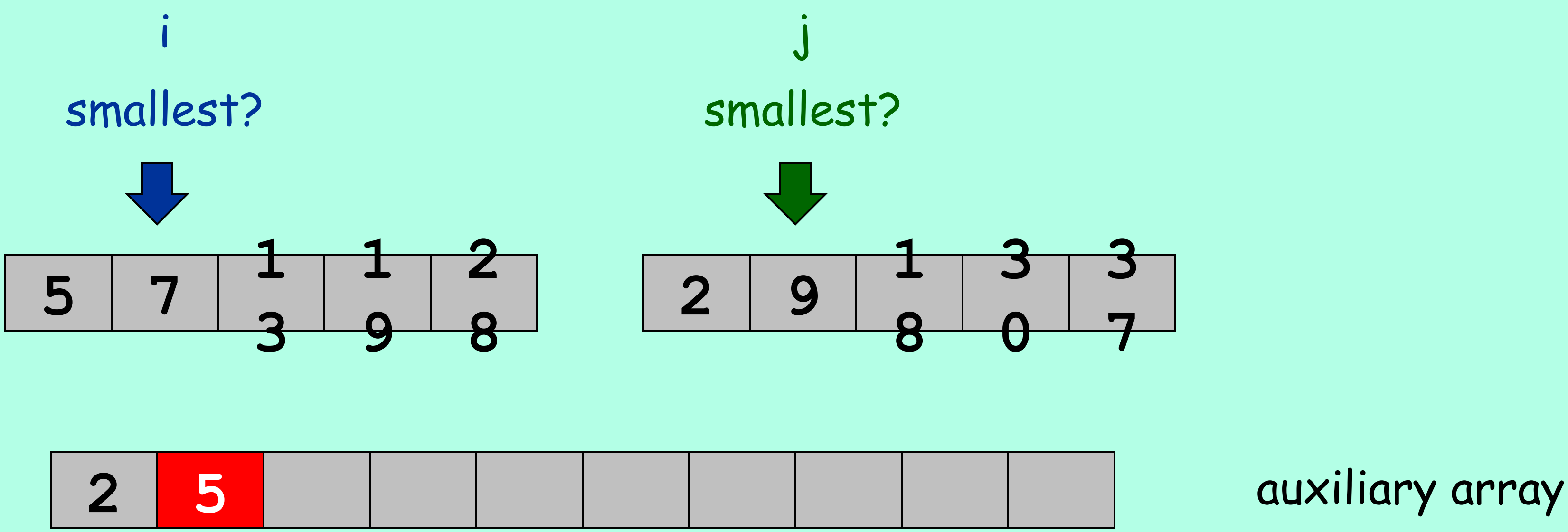
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

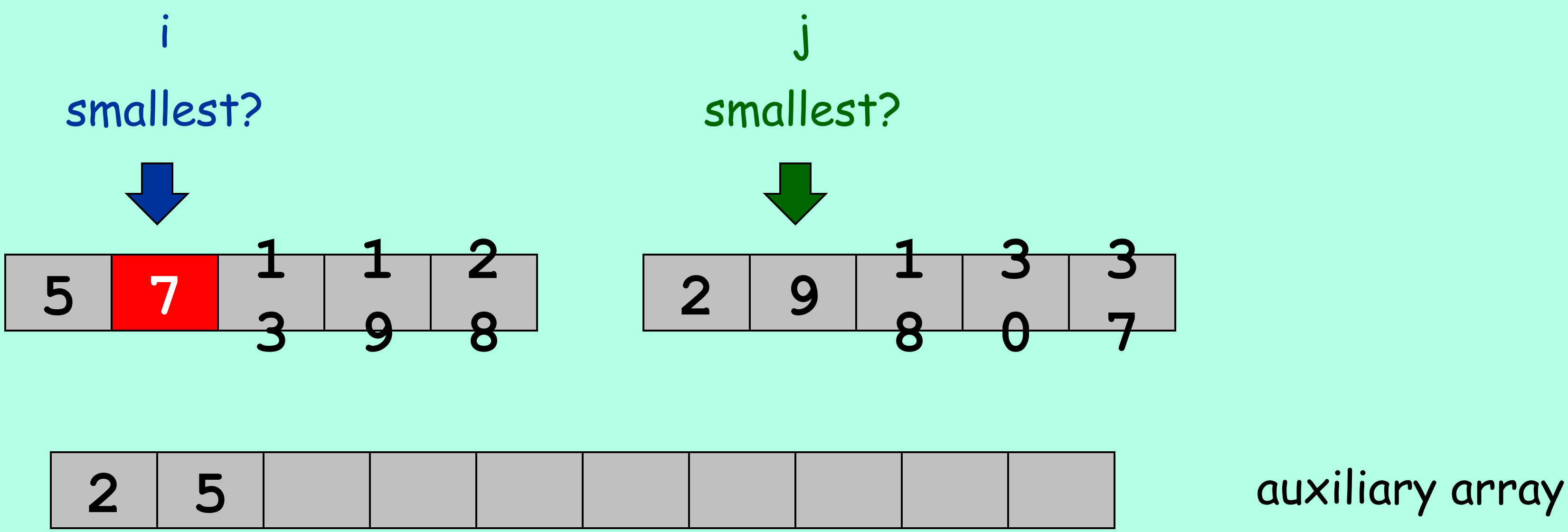
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

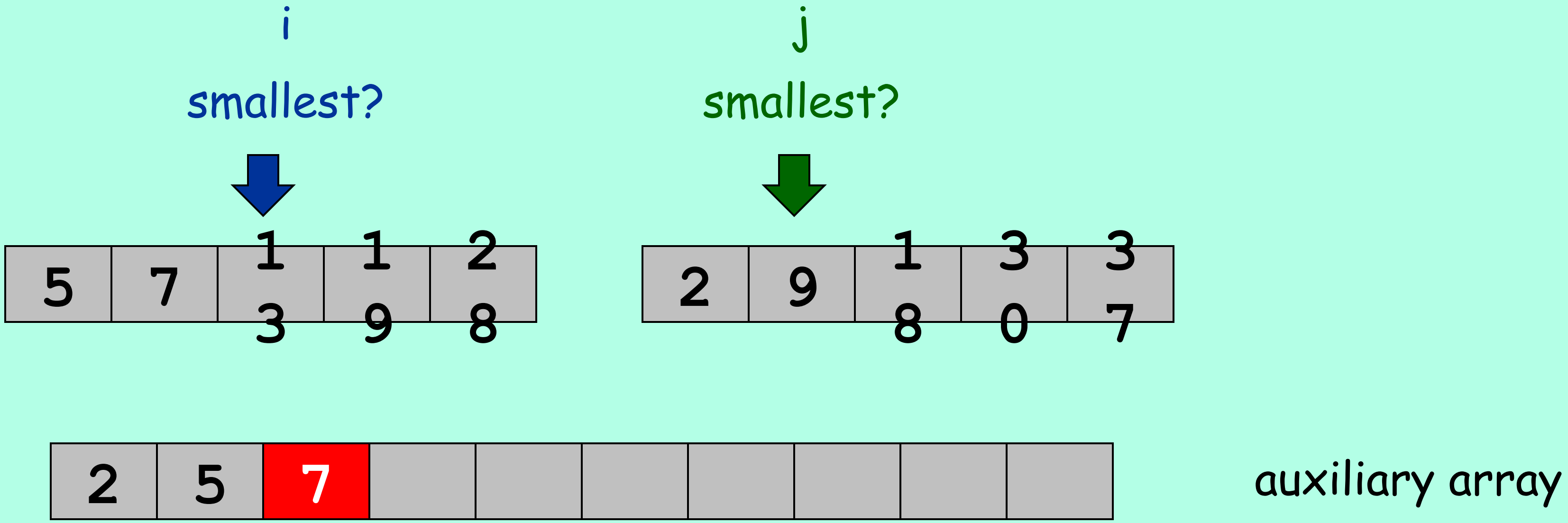
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

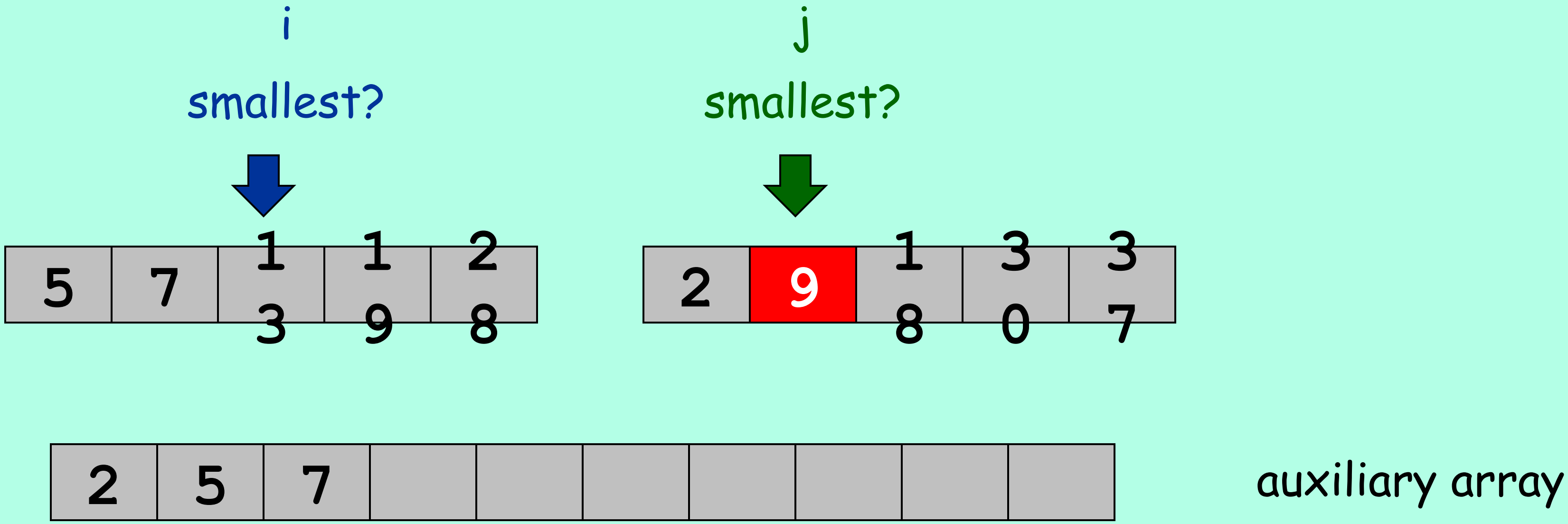
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

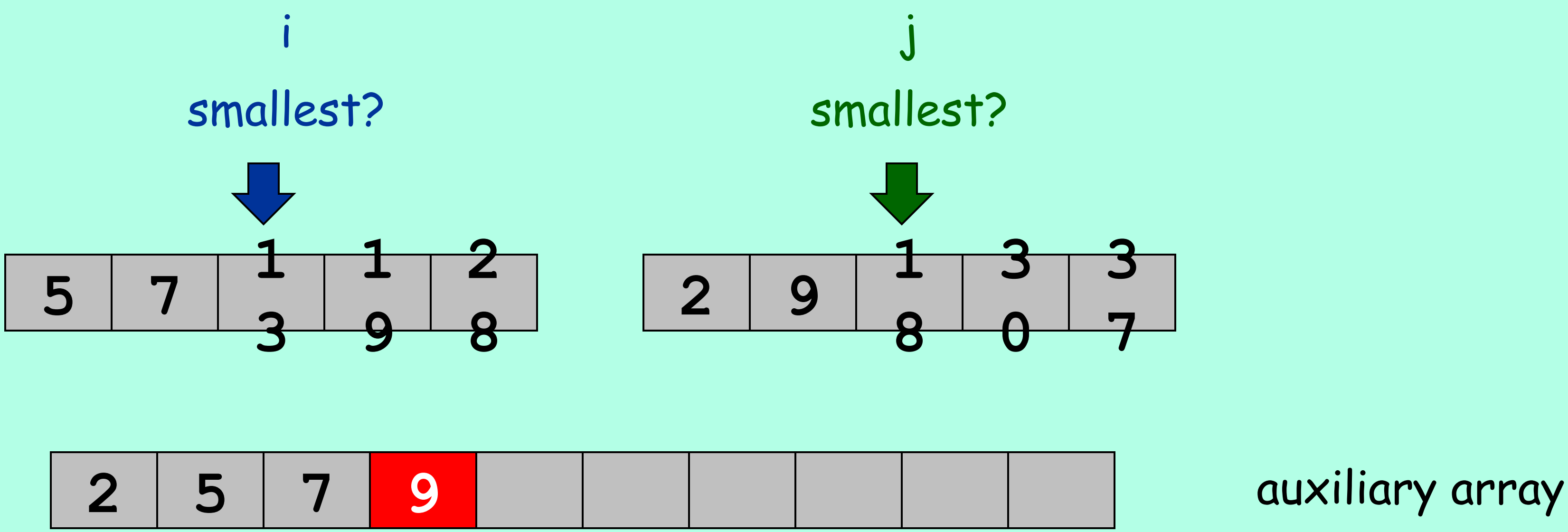
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

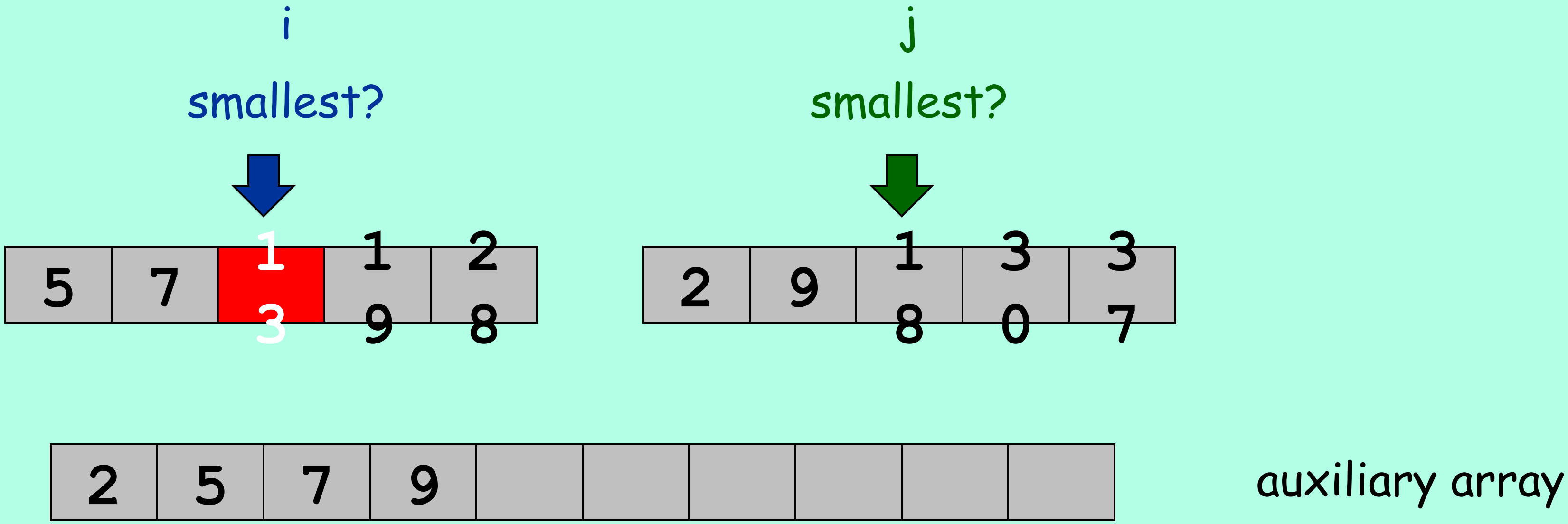
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

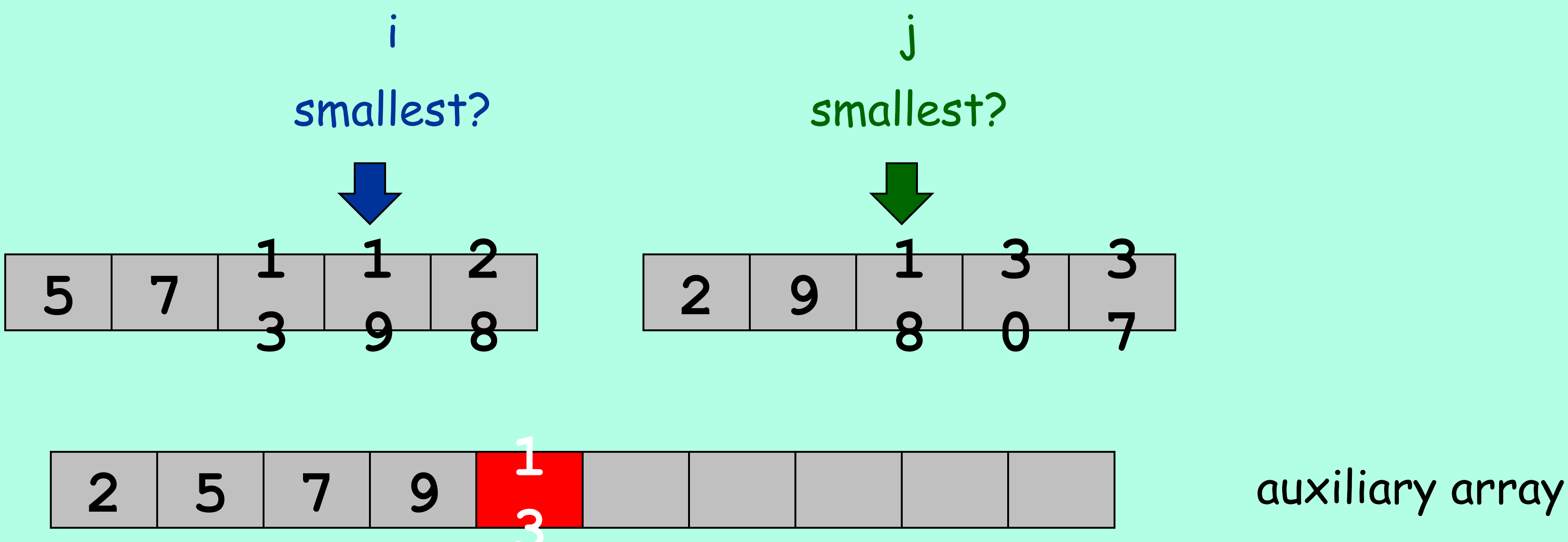
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

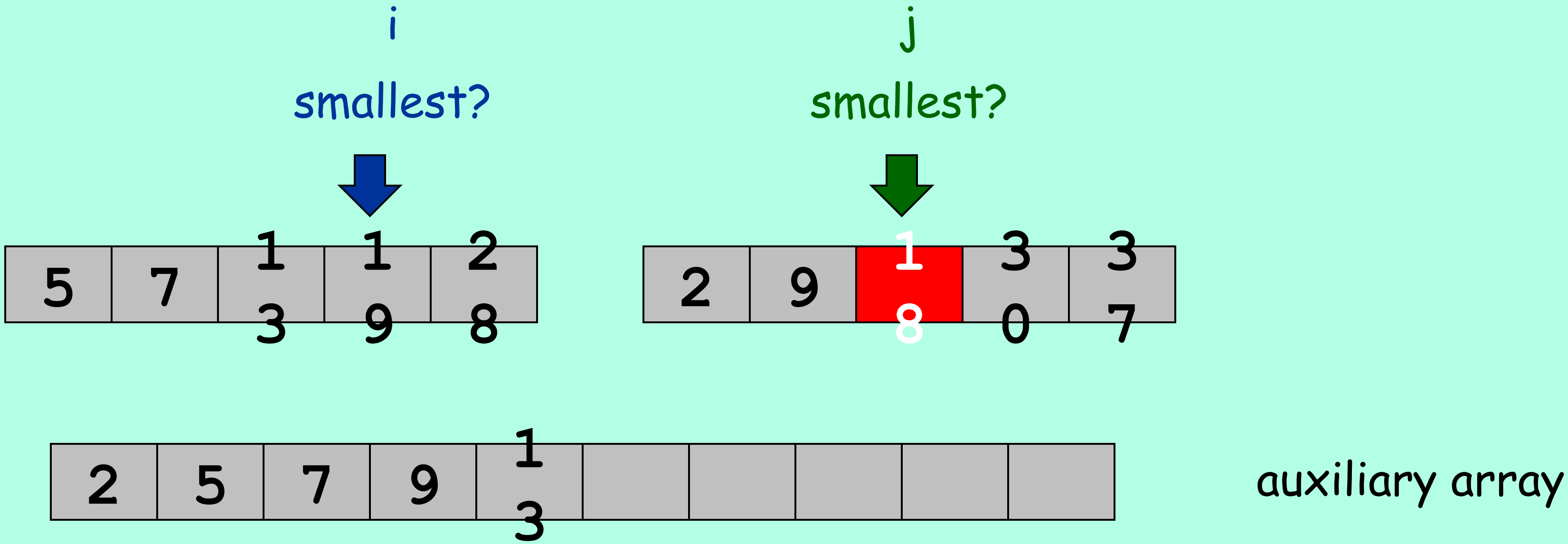
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

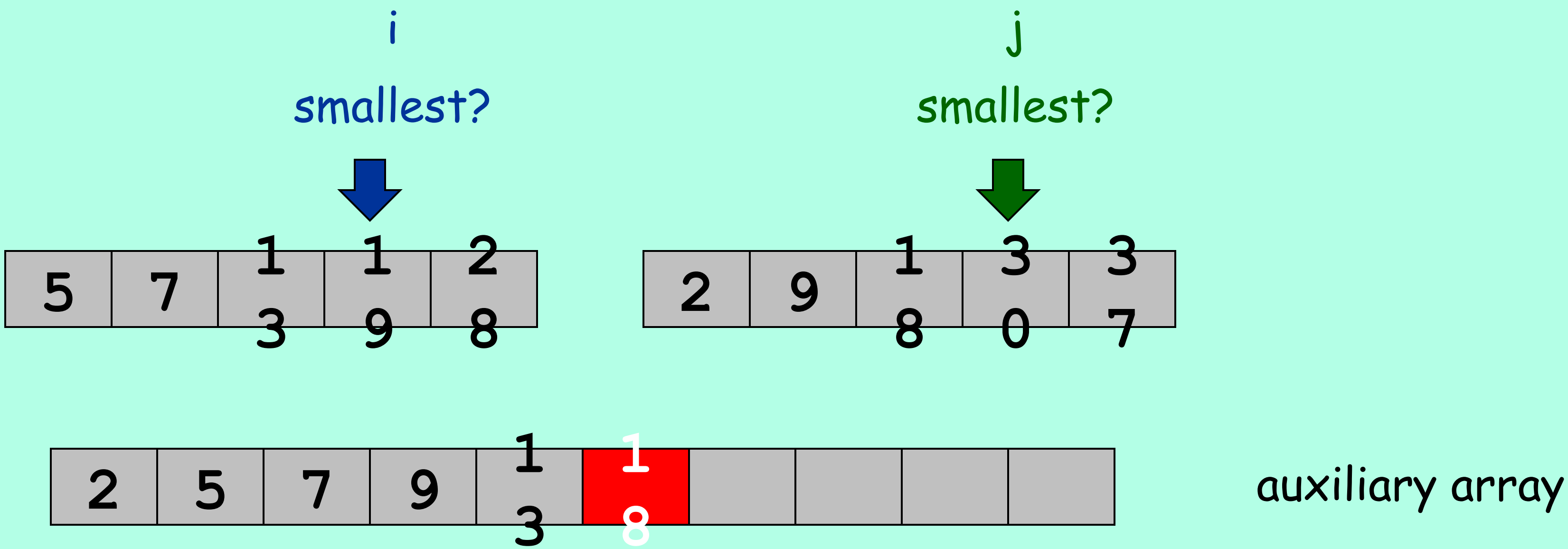
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

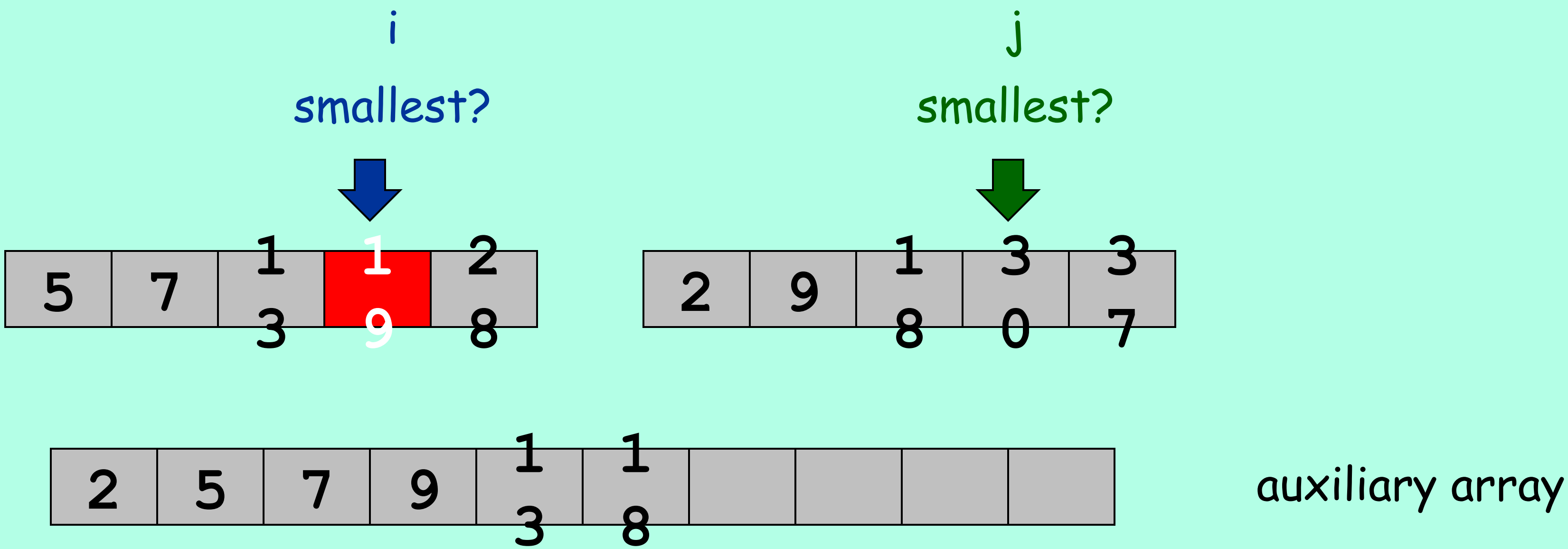
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

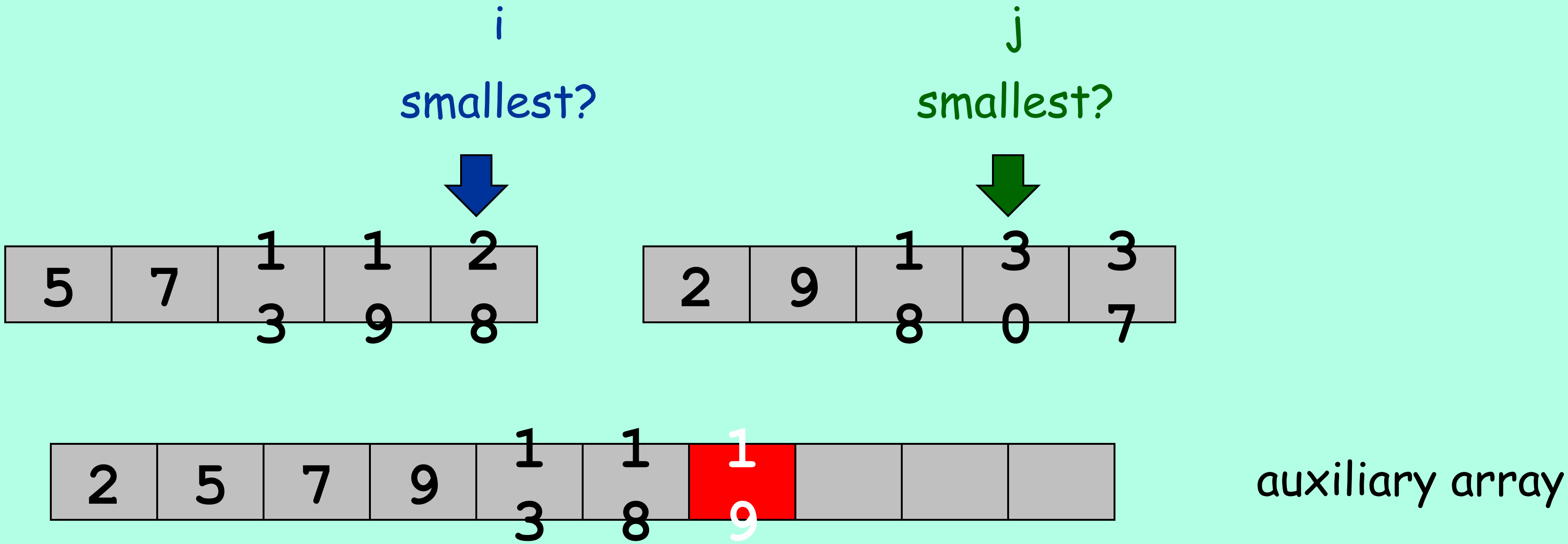
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

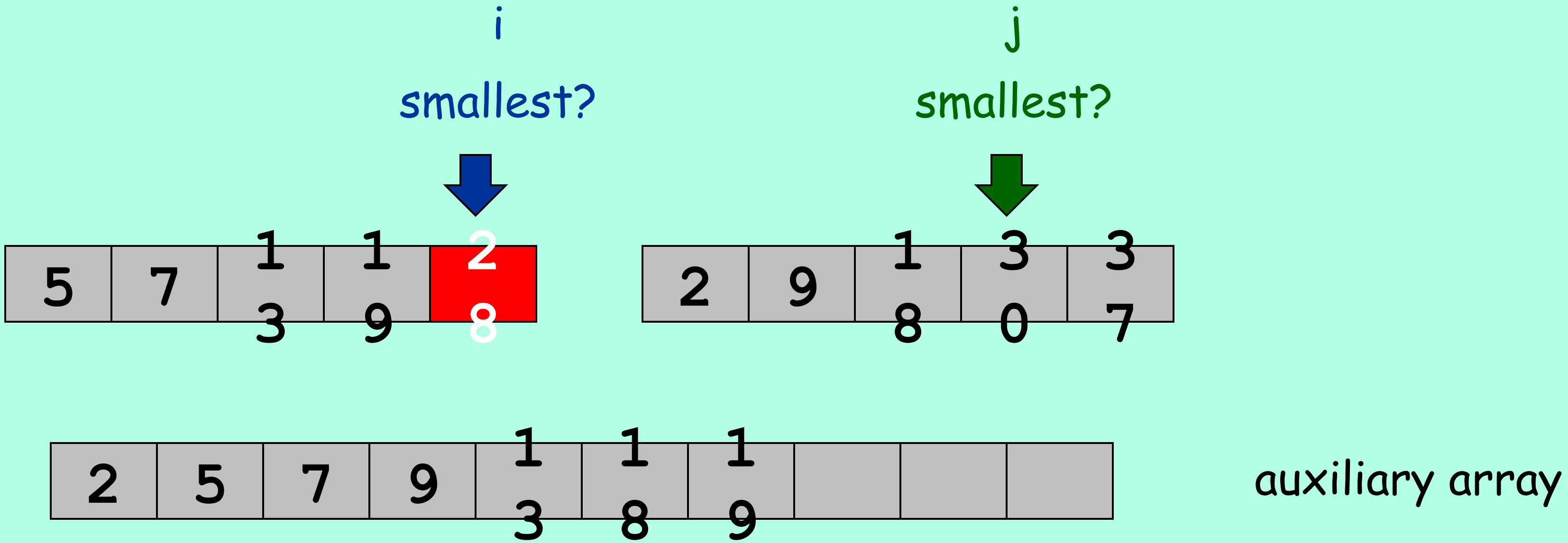
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

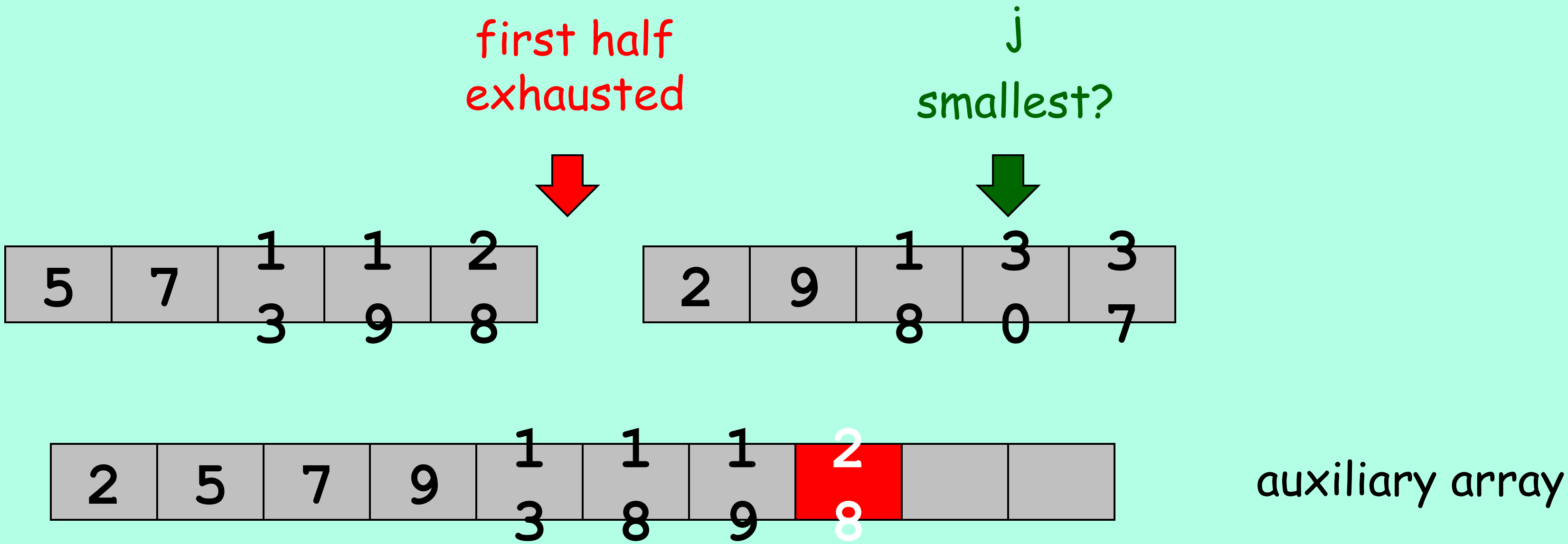
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

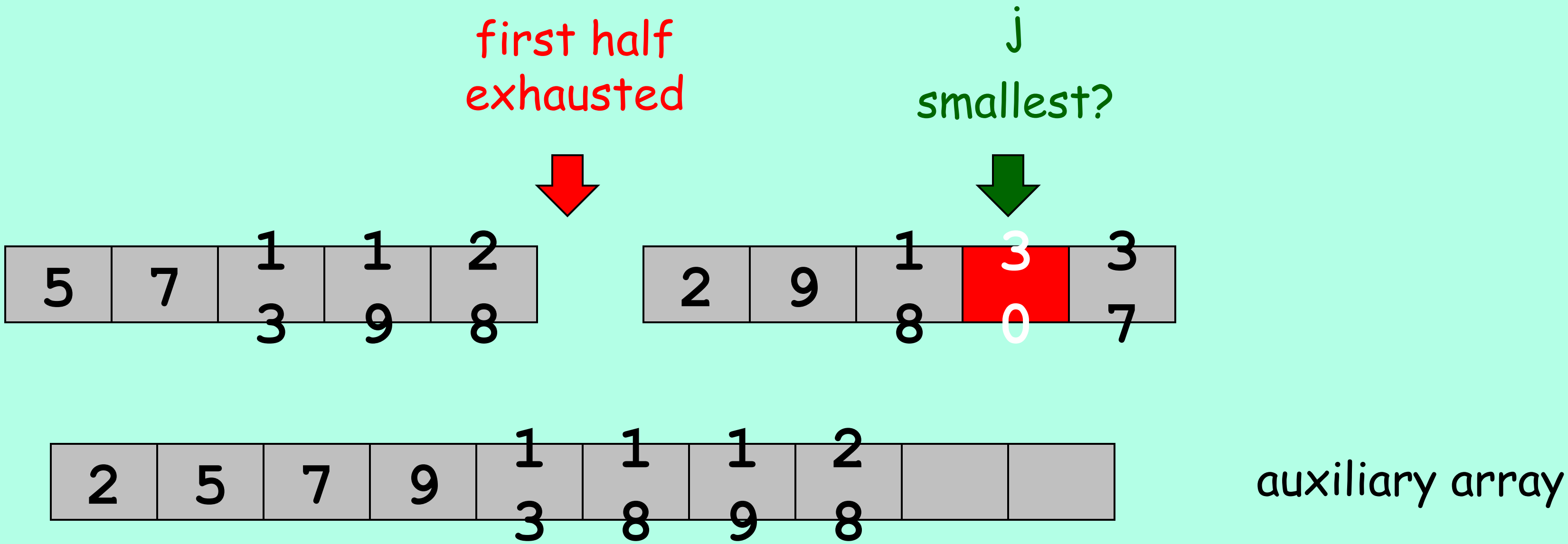
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

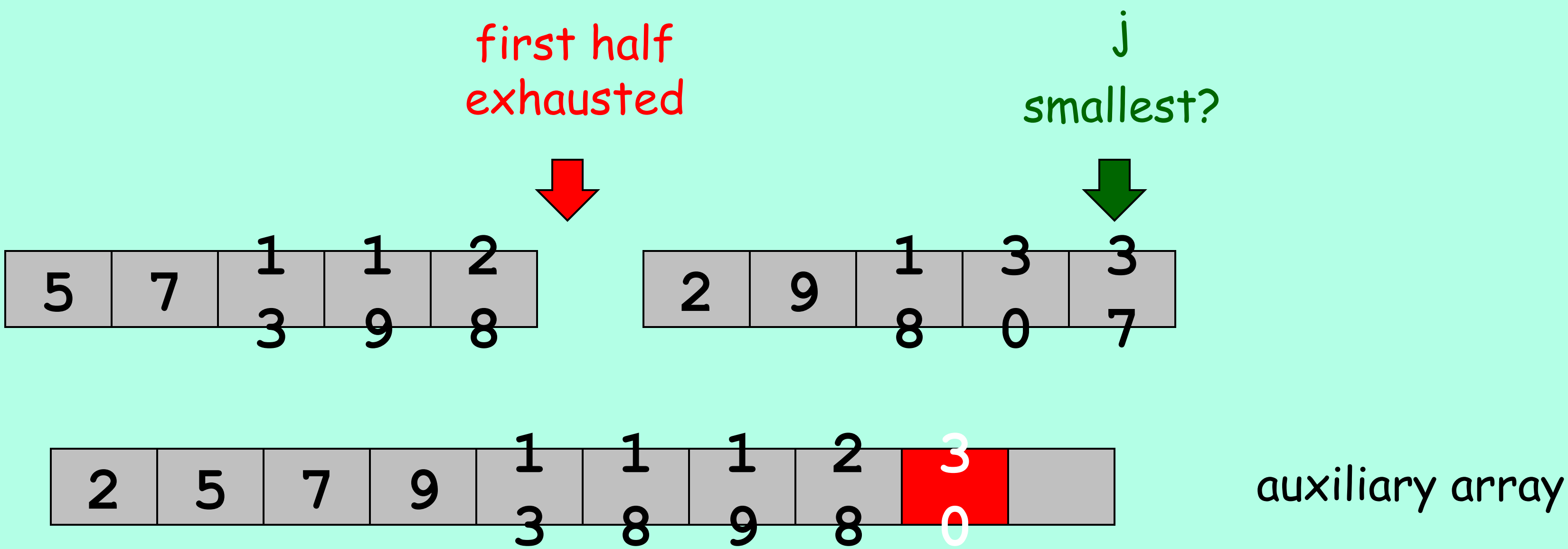
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

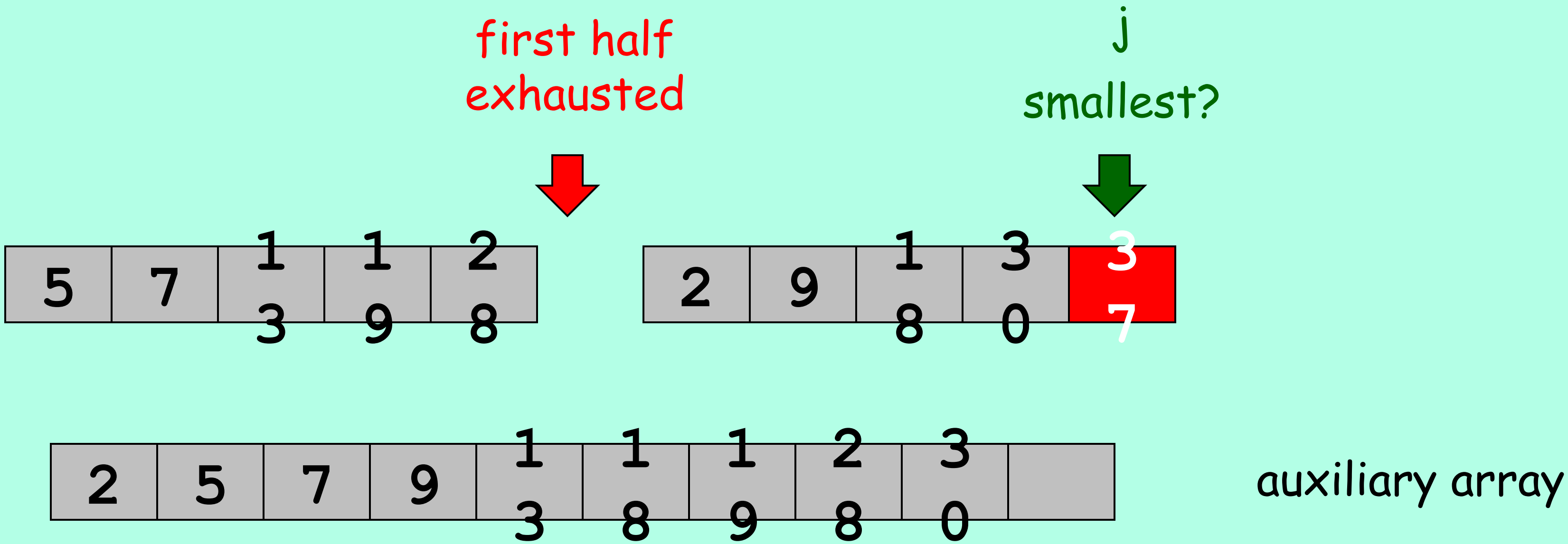
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

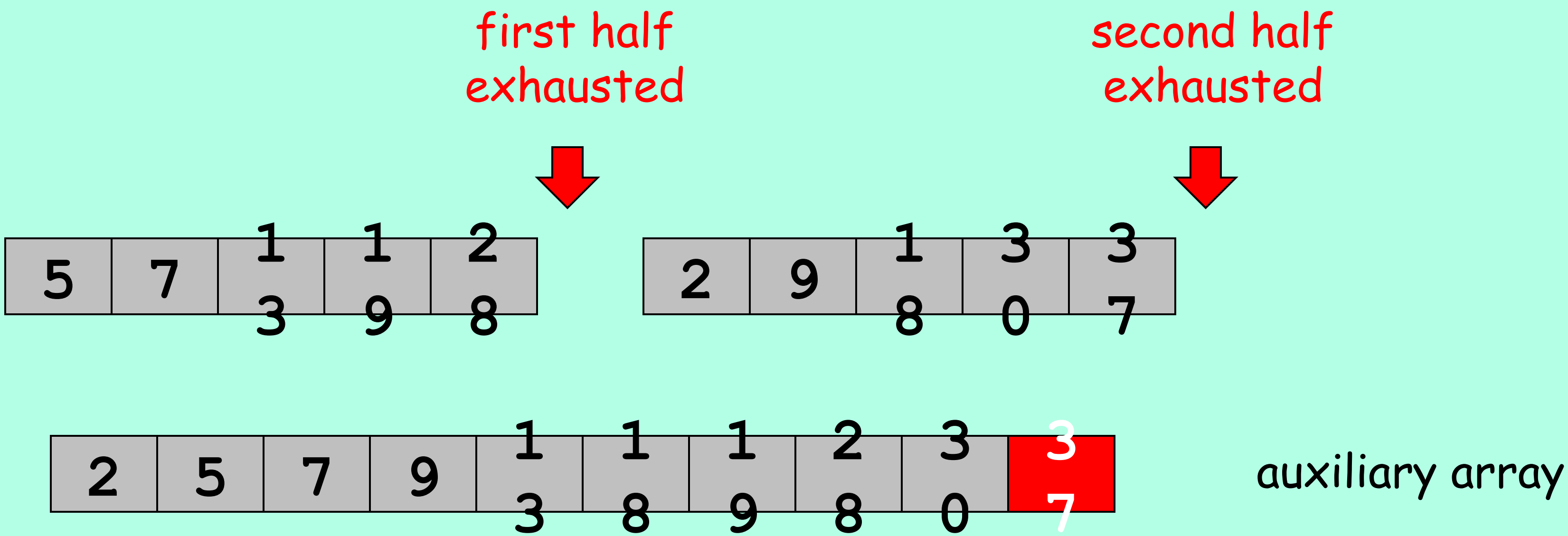
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

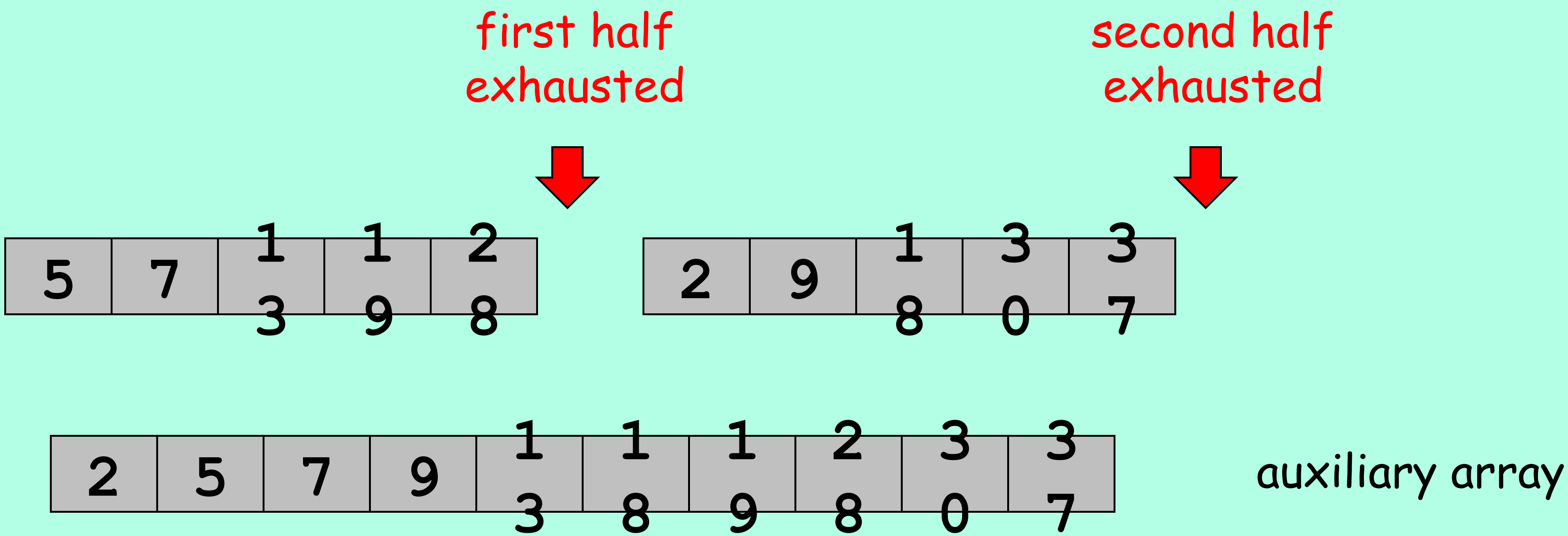
- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Merge.

- Each half is in sorted order.
- Use indices i and j to step through the two halves.
- Compare the two elements.
- Insert smallest of two elements into next position of the auxiliary array.
- Repeat until done.



Merging

Determine the running time of the merge operation:

Given two sorted arrays, each with $n/2$ elements, how many comparisons and copies are made?

There is one comparison for each element that is copied to the auxiliary array.

Each of the n elements are copied one at a time to the auxiliary array.

Total running time is $\Theta(n)$.

5	7	13	19	28

2	9	18	30	37

2	5	7	9	13	18	19	28	30	37

auxiliary array

```
merge(a, left, middle, right)
  // create temporary array b of size right - left + 1
  i = left
  j = middle+1
  k = 1
  while (i <= middle && j <= right) {
    if (a[i] < a[j]) {
      b[k] = a[i]
      i = i + 1
    }
    else {
      b[k] = a[j]
      j = j + 1
    }
    k = k + 1
  }
  while (i <= middle) {
    b[k] = a[i]
    k = k + 1
    i = i + 1
  }

  // continued next slide
```

```
// continued from previous slide

    while (j <= right) {
    b[k] = a[j]
      k = k + 1
      j = j + 1
    }
    x = left
    for (t = 1 to right - left + 1) {
      a[x] = b[t]
      x = x + 1
    }
}
```

Algorithm 6.2.4 Quicksort

This algorithm sorts the array $a[i], \ldots, a[j]$ using a divide and conquer approach. The elements to the left of the partition index p are less than a[p] and the elements to the right of p are greater than p.

```
Input Parameters: a,i,j
Output Parameters: i

quickSort(a,i,j) {
   if (i < j) {
      p = partition(a,i,j)     // partitions smaller
elements
                            // left of a[p], larger to
                            // right of a[p]
      quickSort(a,i,p - 1)
      quickSort(a,p + 1,j)
   }
}
```

Algorithm Heapsort

This algorithm sorts the array *a* using a priority queue (heap). The maximum is always at the top of the heap. The largest and last element of the heap are swapped, siftdown is called to preserve the heap property with a decremented heapsize.

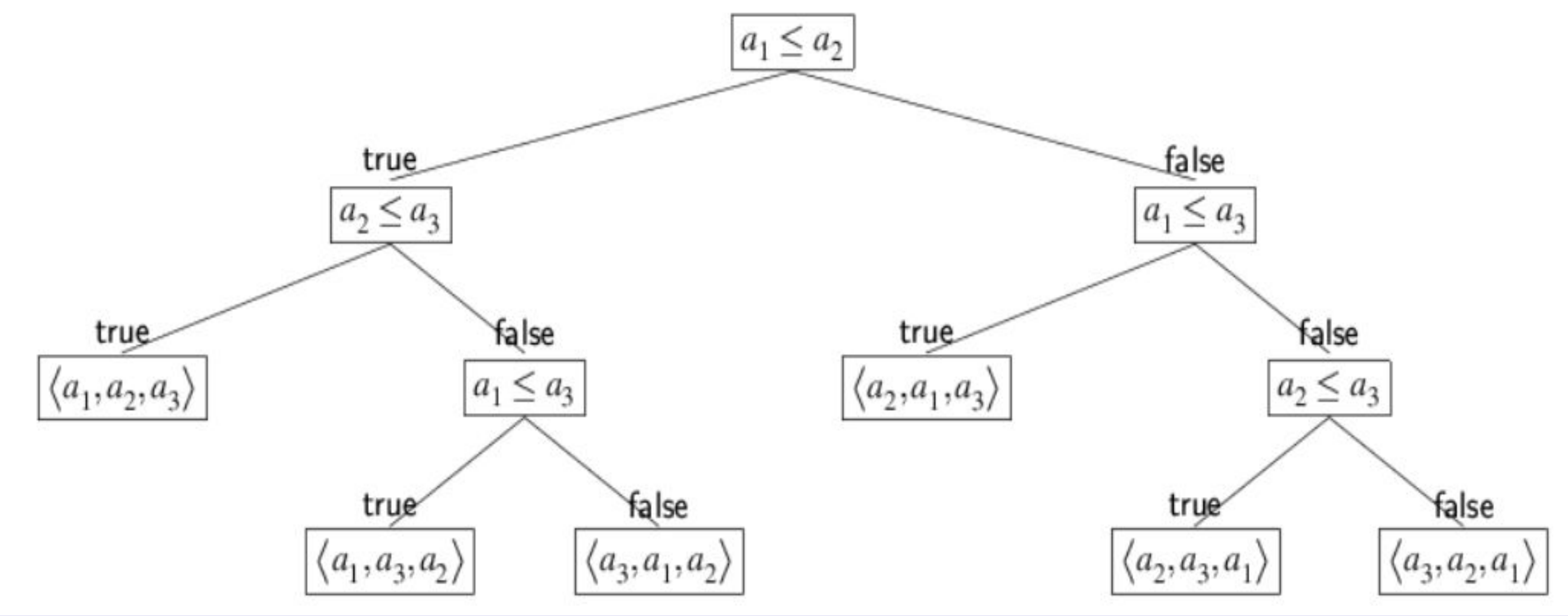
```
heapSort(){
    n = a.last
    heapify(a,n)      // linear time!!
    for  i = n  downto 2 {
        swap(a[1], a[i])
        siftdown(a,1,i-1)
    }
}
```

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

A Lower Bound for the Sorting Problem

Theorem Any comparison-based sorting algorithm has worst case time $\Omega(n \lg n)$.

Conclusion: Any comparison-based sorting algorithm must take time **<u>at least</u>** $n \lg n$. There is no hope of finding any faster comparison-based algorithm.



The decision tree for comparing and ordering an array of n elements has $n!$ leaves.

In the worst case, the number of comparisons $\geq$ height of tree

$$\geq \lg(n!)$$
$$= \Omega(n \lg n)$$

Algorithm 6.4.2 Counting Sort

This algorithm sorts an array $a[1], \ldots, a[n]$ of integers.

Assumption: Each integer is in the range 0 to m, inclusive; usually m is a fairly small value.

It operates by **counting** how many occurrences there are of each integer in the range 0 to m. Next, the array c is used to determine how many values in the array are less than or equal to each integer in the range 0 to m.

```
c[k] = the number of values less than or equal to k
       in the array a
```

```
countingSort(a,m) {
    for k = 0 to m
        c[k] = 0
    n = a.last

    for i = 1 to n
        c[a[i]] = c[a[i]] + 1        // how many of each
value

    for k = 1 to m
        c[k] = c[k] + c[k - 1]      // how many ≤ k

    // sort a with the result in b
    for i = n downto 1 {
        b[c[a[i]]] = a[i]
        c[a[i]] = c[a[i]] - 1
    }

    // copy b back to a
    for i = 1 to n
        a[i] = b[i]
}
```

Counting Sort

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

Data in the range 0..6
m = 6

	0	1	2	3	4	5	6
c	**0**	**0**	**0**	**0**	**0**	**0**	**0**

```
for i = 1 to n
  c[a[i]] = c[a[i]] + 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	0	1	2	3	4	5	6
c	**0**	**0**	**0**	**0**	**1**	**0**	**0**

```
for i = 1 to n
  c[a[i]] = c[a[i]] + 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	0	1	2	3	4	5	6
c	**0**	**0**	**0**	**1**	**1**	**0**	**0**

```
for i = 1 to n
  c[a[i]] = c[a[i]] + 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	0	1	2	3	4	5	6
c	**1**	**0**	**0**	**1**	**1**	**0**	**0**

```
for i = 1 to n
    c[a[i]] = c[a[i]] + 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	0	1	2	3	4	5	6
c	**2**	**1**	**1**	**3**	**2**	**0**	**1**

```
for k = 1 to m
    c[k] = c[k] + c[k - 1]
```

a

1	2	3	4	5	6	7	8	9	10
4	3	0	3	6	2	0	1	3	4

k

c

0	1	2	3	4	5	6
2	1	1	3	2	0	1

```
for k = 1 to m
  c[k] = c[k] + c[k - 1]
```

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

k

	0	1	2	3	4	5	6
c	**2**	**3**	**1**	**3**	**2**	**0**	**1**

```
for k = 1 to m
  c[k] = c[k] + c[k - 1]
```

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

k

	0	1	2	3	4	5	6
c	2	3	4	3	2	0	1

```
for k = 1 to m
    c[k] = c[k] + c[k - 1]
```

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

k

	0	1	2	3	4	5	6
c	2	3	4	7	2	0	1

```
for k = 1 to m
  c[k] = c[k] + c[k - 1]
```

a

1	2	3	4	5	6	7	8	9	10
4	3	0	3	6	2	0	1	3	4

k

c

0	1	2	3	4	5	6
2	3	4	7	9	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	1	2	3	4	5	6	7	8	9	10
b										

	0	1	2	3	4	5	6
c	**2**	**3**	**4**	**7**	**9**	**9**	**10**

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b									4	

	0	1	2	3	4	5	6
c	2	3	4	7	8	9	10

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b									4	

	0	1	2	3	4	5	6
c	2	3	4	7	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b							3		4	

	0	1	2	3	4	5	6
c	2	3	4	6	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b							3		4	

	0	1	2	3	4	5	6
c	2	3	4	6	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b			1				3		4	

	0	1	2	3	4	5	6
c	2	2	4	6	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b			1				3		4	

	0	1	2	3	4	5	6
c	2	2	4	6	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

a

1	2	3	4	5	6	7	8	9	10
4	3	0	3	6	2	0	1	3	4

b

1	2	3	4	5	6	7	8	9	10
	0	1				3		4	

c

0	1	2	3	4	5	6
1	2	4	6	8	9	10

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	1	2	3	4	5	6	7	8	9	10
b		**0**	**1**				**3**		**4**	

	0	1	2	3	4	5	6
c	**1**	**2**	**4**	**6**	**8**	**9**	**10**

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b		0	1	2			3		4	

	0	1	2	3	4	5	6
c	1	2	3	6	8	9	10

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b		0	1	2			3		4	

	0	1	2	3	4	5	6
c	1	2	3	6	8	9	10

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b		0	1	2			3		4	6

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

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b		0	1	2			3		4	6

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

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b		0	1	2		3	3		4	6

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

```
for i = n downto 1
   b[c[a[i]]] = a[i]
   c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	1	2	3	4	5	6	7	8	9	10
b		**0**	**1**	**2**		**3**	**3**		**4**	**6**

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

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b	0	0	1	2		3	3		4	6

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

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b	0	0	1	2		3	3		4	6

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

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b	0	0	1	2	3	3	3		4	6

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

```
for i = n downto 1
    b[c[a[i]]] = a[i]
    c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	1	2	3	4	5	6	7	8	9	10
b	**0**	**0**	**1**	**2**	**3**	**3**	**3**		**4**	**6**

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

```
for i = n downto 1
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
```

i

	1	2	3	4	5	6	7	8	9	10
a	**4**	**3**	**0**	**3**	**6**	**2**	**0**	**1**	**3**	**4**

	1	2	3	4	5	6	7	8	9	10
b	**0**	**0**	**1**	**2**	**3**	**3**	**3**	**4**	**4**	**6**

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

```
for i = 1 to n
  a[i] = b[i]
```

i

	1	2	3	4	5	6	7	8	9	10
a	4	3	0	3	6	2	0	1	3	4

	1	2	3	4	5	6	7	8	9	10
b	0	0	1	2	3	3	3	4	4	6

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

```
for i = 1 to n
  a[i] = b[i]
```

i

a

1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	3	3	4	4	6

b

1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	3	3	4	4	6

c

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

```
countingSort(a,m) {
    for k = 0 to m
        c[k] = 0
    n = a.last

    for i = 1 to n
        c[a[i]] = c[a[i]] + 1        // how many of each
value

    for k = 1 to m
        c[k] = c[k] + c[k - 1]       // how many ≤ k

    // sort a with the result in b
    for i = n downto 1 {
        b[c[a[i]]] = a[i]
        c[a[i]] = c[a[i]] - 1
    }

    // copy b back to a
    for i = 1 to n
        a[i] = b[i]
}
```


$\theta(\mathbf{n+m}) = \theta(\mathbf{n})$ if m is some “small” constant

Running time?

What's going on? Linear time?

This is not a comparison-based sort. It never compares two elements in the array (e.g., `if  a[i] < a[j]`).

Algorithm 6.4.4 Radix Sort

This algorithm sorts the array $a[1], \ldots , a[n]$ of integers.

Assumption: Each integer has k digits.

It sorts the integers by digit, working from the LEAST significant digit to the most significant digit.

```
radixSort(a,k)
{
    // k is the number of digits in each element of  a
   for i = 0 to k-1
      countingSort(a,9) on digit in 10^i place
}
```

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329

457

657

839

436

720

355

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329

457

657

839

436

720

355

sort by 10^0 = ones digit

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329		720
457		355
657		436
839	→	457
436		657
720		329
355		839

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329	→	720
457		355
657		436
839		457
436		657
720		329
355		839

sort by 10^1 = tens digit

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

sort by 10^2 = hundreds digit

```
radixSort(a,k) {
    for i = 0 to k-1
        countingSort(a,9) on digit in 10^i place
}
```

k = 3

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm 6.4.4 Radix Sort

This algorithm sorts the array $a[1], \ldots, a[n]$ of integers.

Assumption: Each integer has k digits.

It sorts the integers by digit, working from the LEAST significant digit to the most significant digit.

```
radixSort(a,k)
{
     // k is the number of digits in each element of  a
    for i = 0 to k-1
       countingSort(a,9) on digit in 10^i place
}
```

Determine the running time of `radixSort`.

$\theta(\mathbf{kn}) = \theta(\mathbf{n})$ if k is some "small" constant