

Iterative Search and Sort

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Binary Search
- Bubble Sort
- Counting Sort
- Radix Sort

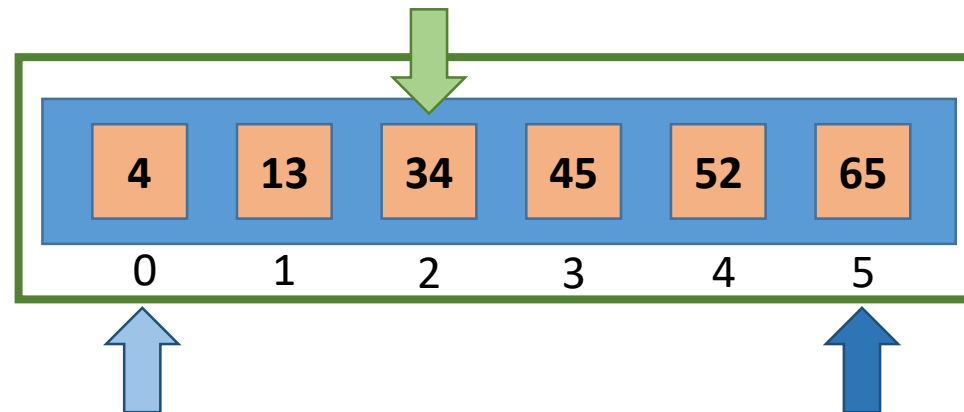
Binary Search

- Takes a “divide and conquer” approach.
- Begins searching in the middle of the array.
 - If the middle element is not what we are looking for, we then split the array in half:
 - If the value sought is greater* than the middle element, we will then check the middle element of the second half of the array.
 - If the value sought is less* than the middle element, we will then check the middle element of the first half of the array.
 - The process begins again with the new half.

*- The array must be in some order! (Alphabetically, numerically, etc.)

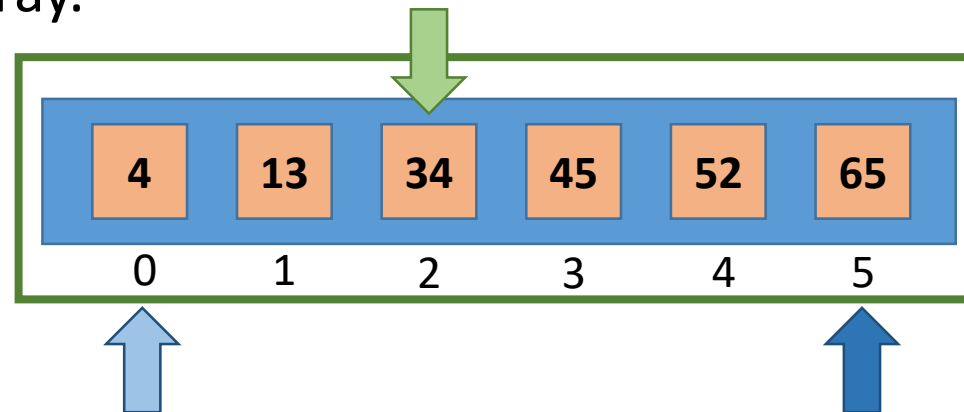
Binary Search (Searching for 45)

- In the first step of the algorithm:
 - The lower boundary is index 0.
 - The upper boundary is the last index.
 - To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: $(0 + 5)/2 = 2.5 \rightarrow 2$



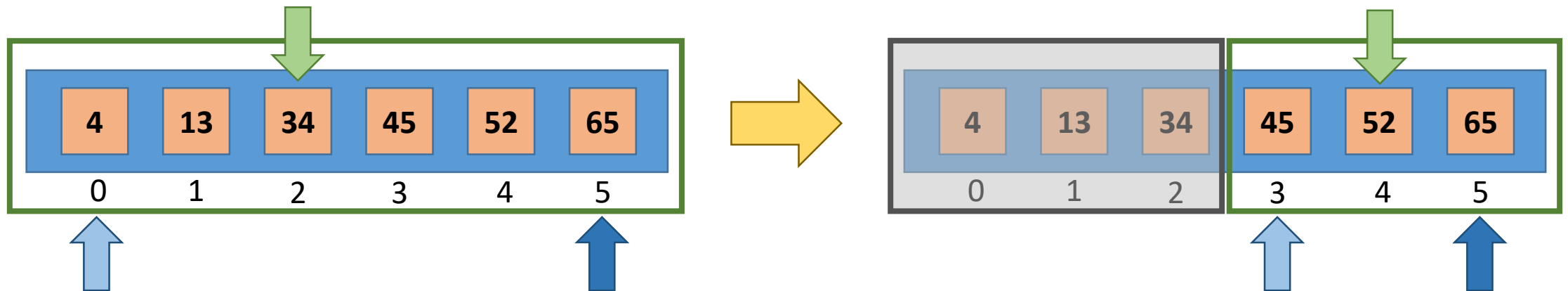
Binary Search (Searching for 45)

- Next, we do one of three things:
 - If the value we are seeking (45) is at this middle index, we are done searching.
 - **If the value we are seeking is greater than this value, then we will search the upper half of this array.**
 - If the value we are seeking is less than this value, then we will search the lower half of this array.



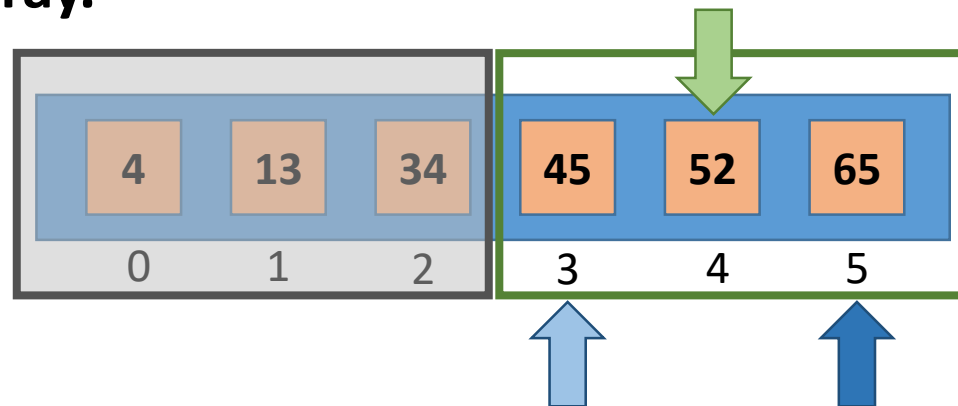
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
 - Lower Boundary: Middle Index + 1 $\rightarrow 2 + 1 = 3$
 - Upper Boundary: Does not change.
 - Middle Index: (Lower + Upper) / 2 $\rightarrow (3 + 5) / 2 = 4$



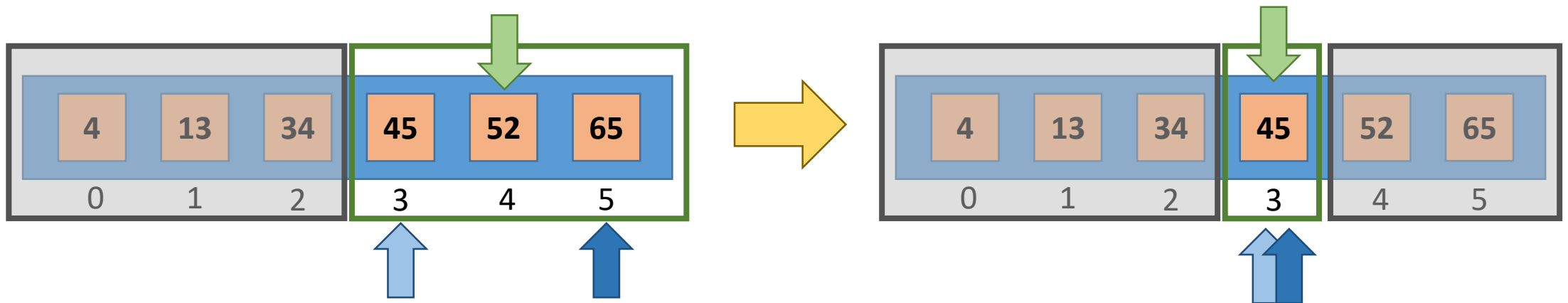
Binary Search (Searching for 45)

- We start the process over:
 - If the value we are seeking (45) is at this middle index, we are done searching.
 - If the value we are seeking is greater than this value, then we will search the upper half of this array.
 - **If the value we are seeking is less than this value, then we will search the lower half of this array.**



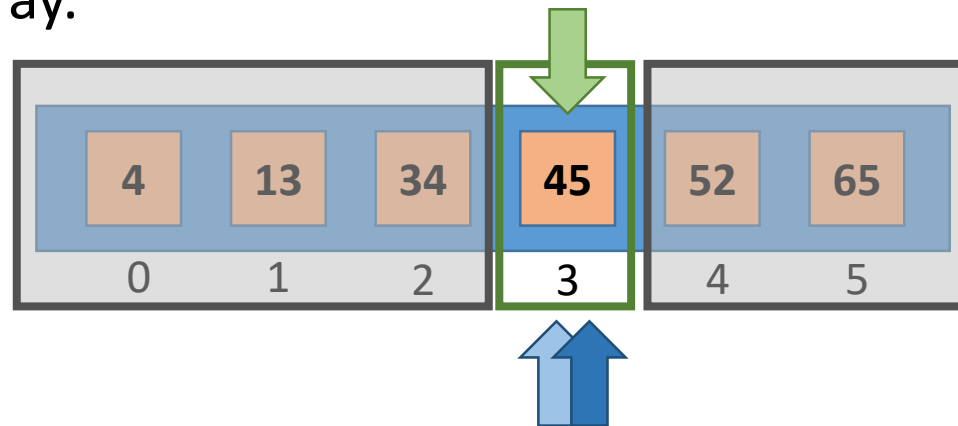
Binary Search (Searching for 45)

- We must now calculate the boundaries and middle index of the half we will search next.
 - Lower Boundary: Does not change.
 - Upper Boundary: Middle Index - 1 $\rightarrow 4 - 1 = 3$
 - Middle Index: (Lower + Upper) / 2 $\rightarrow (3 + 3) / 2 = 3$



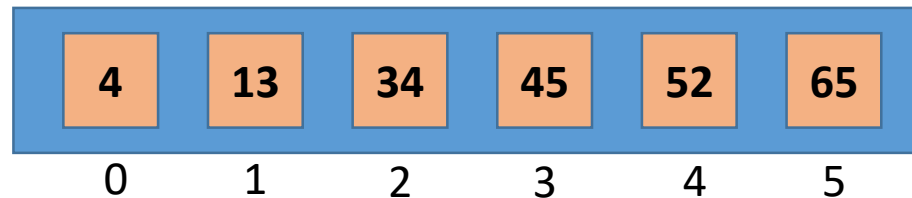
Binary Search (Searching for 45)

- We start the process over:
 - **If the value we are seeking (45) is at this index, we are done searching.**
 - If the value we are seeking is greater than this value, then we will search the upper half of this array.
 - If the value we are seeking is less than this value, then we will search the lower half of this array.



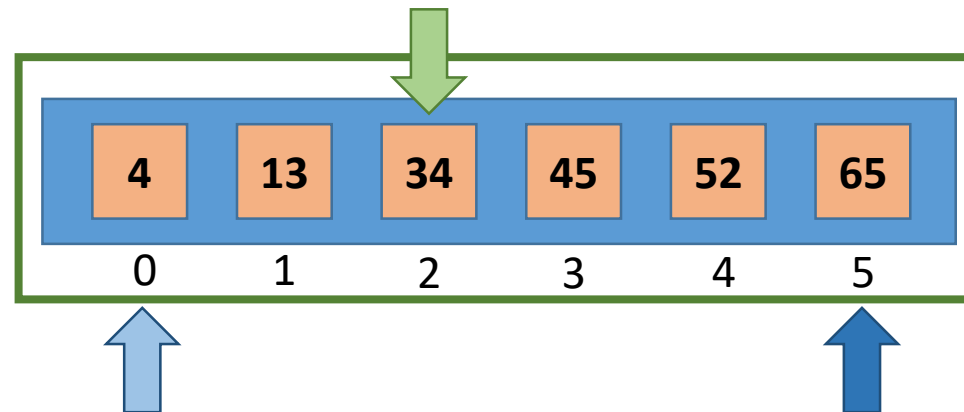
Binary Search

- What happens when the value we are looking for isn't in the array?
 - How does the algorithm know when to stop halving the array?
 - **When the upper boundary index is less than the lower boundary index.**



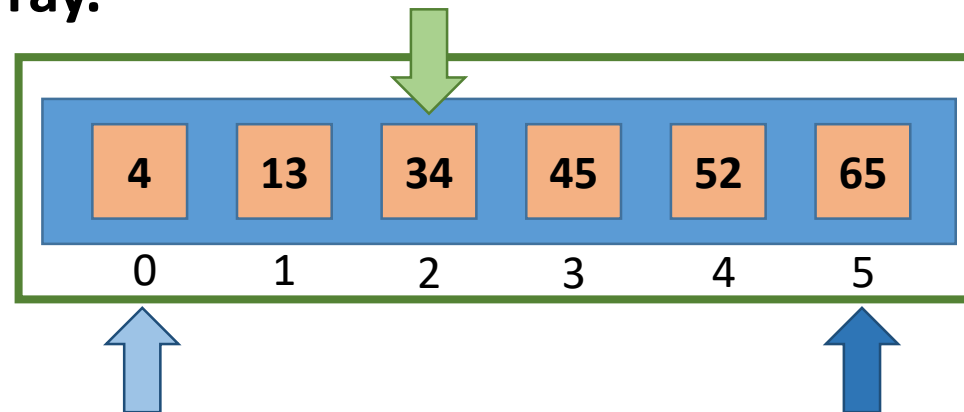
Binary Search (Searching for 12)

- The lower boundary is index 0.
- The upper boundary is the last index.
- To find the middle index: add the lower boundary index and the upper boundary index then divide by 2: $(0 + 5)/2 = 2.5 \rightarrow 2$



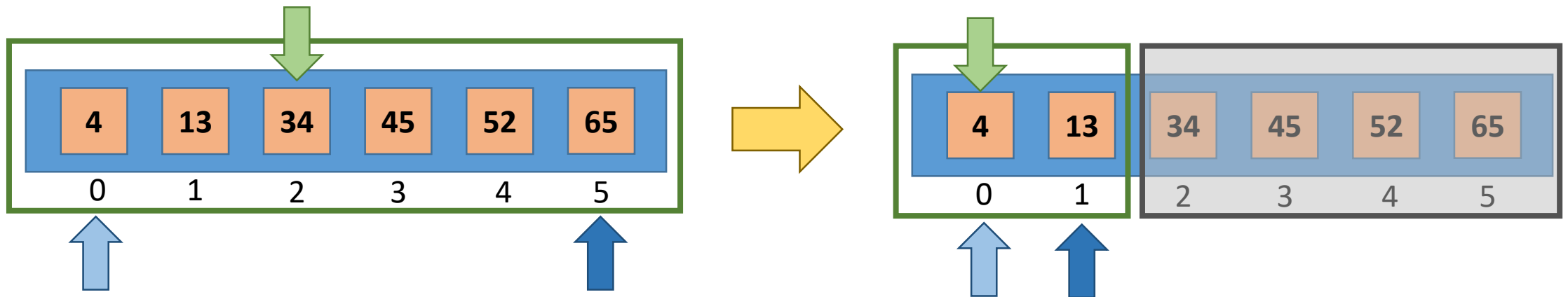
Binary Search (Searching for 12)

- Next, we do one of three things:
 - If the value we are seeking (12) is at this middle index, we are done searching.
 - If the value we are seeking is greater than this value, then we will search the upper half of this array.
 - **If the value we are seeking is less than this value, then we will search the lower half of this array.**



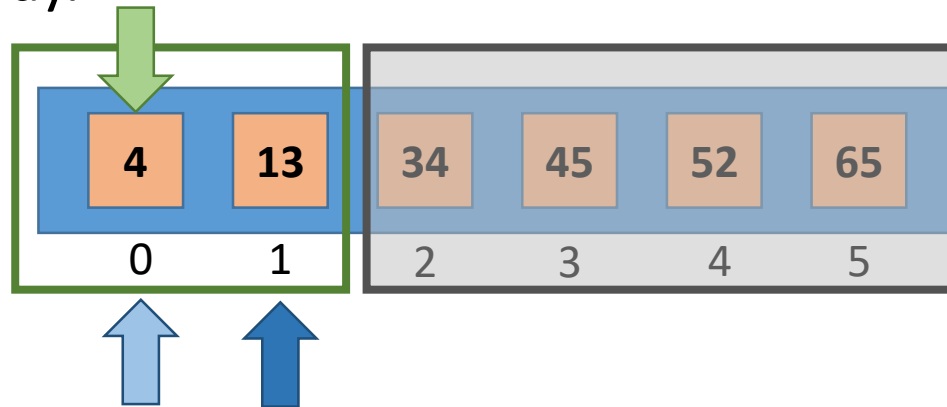
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
 - Lower Boundary: Does not change.
 - Upper Boundary: Middle Index $- 1 \rightarrow 2 - 1 = 1$
 - Middle Index: $(\text{Lower} + \text{Upper}) / 2 \rightarrow (0 + 1) / 2 = 0.5 \rightarrow 0$



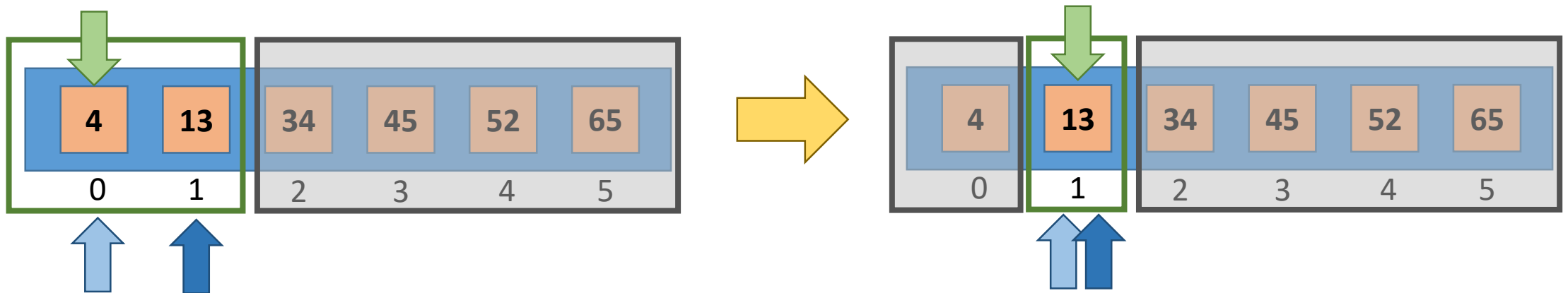
Binary Search (Searching for 12)

- We start the process over:
 - If the value we are seeking (12) is at this index, we are done searching.
 - **If the value we are seeking is greater than this value, then we will search the upper half of this array.**
 - If the value we are seeking is less than this value, then we will search the lower half of this array.



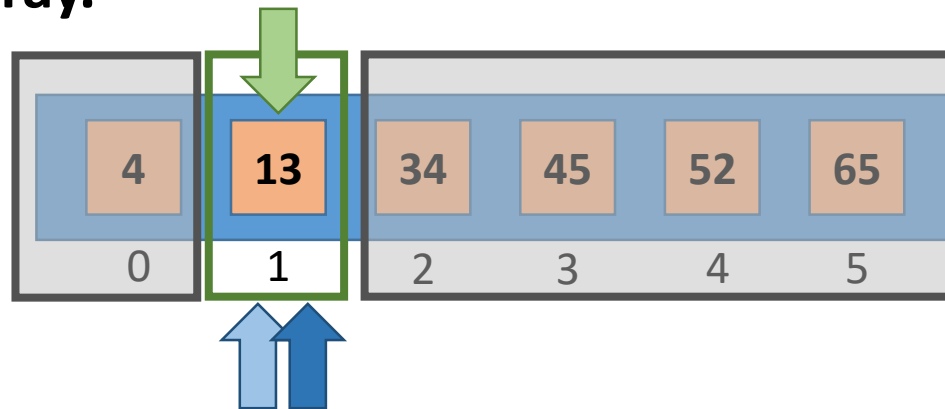
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
 - Lower Boundary: Middle Index + 1 $\rightarrow 0 + 1 = 1$
 - Upper Boundary: Does not change.
 - Middle Index: (Lower + Upper) / 2 $\rightarrow (1 + 1) / 2 = 1$



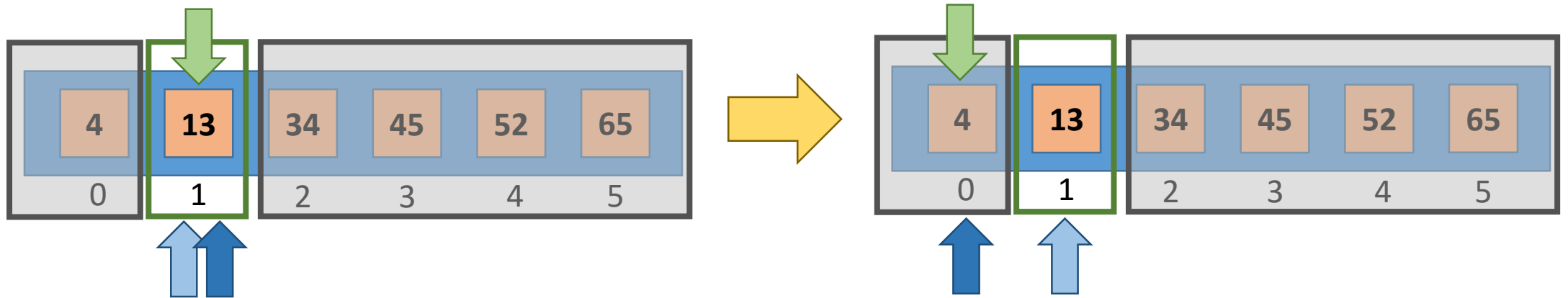
Binary Search (Searching for 12)

- We start the process over:
 - If the value we are seeking (12) is at this index, we are done searching.
 - If the value we are seeking is greater than this value, then we will search the upper half of this array.
 - **If the value we are seeking is less than this value, then we will search the lower half of this array.**



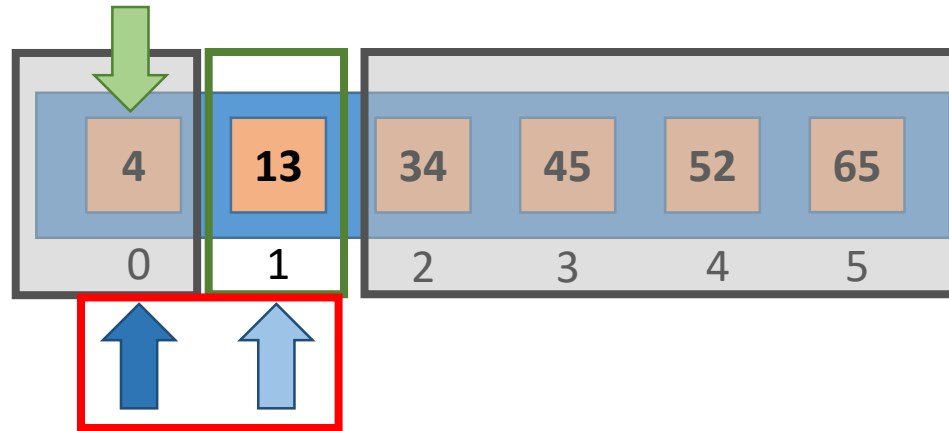
Binary Search (Searching for 12)

- We must now calculate the boundaries and middle index of the half we will search next.
 - Lower Boundary: Does not change.
 - Upper Boundary: Middle Index $- 1 \rightarrow 1 - 1 = 0$
 - Middle Index: $(\text{Lower} + \text{Upper}) / 2 \rightarrow (1 + 0) / 2 = 0.5 \rightarrow 0$



Binary Search (Searching for 12)

- When the upper boundary is less than the lower boundary, the algorithm will "give up."



Binary Search (Pseudocode)

Set *low* boundary to the first index (0)

Set *high* boundary to the last index (length-1)

while the *high* boundary \geq *low* boundary :

 Calculate the middle index, m $(high + low) / 2$

 If the element $a[m]$ is what you are seeking :

 Return m

 If the element you are seeking is $> a[m]$:

 Calculate the new *low* boundary $(m + 1)$

 If the element you are seeking is $< a[m]$:

 Calculate the new *high* boundary $(m - 1)$

Binary Search (C++ Function)

```
int binarySearch(int a[], int length, int searchValue) {  
    int lowBoundary = 0;  
    int highBoundary = length - 1;  
    while(highBoundary >= lowBoundary) {  
        int middle = (highBoundary + lowBoundary) / 2;  
        if(searchValue == a[middle]) {  
            return middle;  
        }  
        else if(searchValue > a[middle]) {  
            lowBoundary = middle + 1;  
        }  
        else {  
            highBoundary = middle - 1;  
        }  
    }  
    return -1;  
}
```

Indicates the value was not found.

The return statement in the loop would never return -1

Binary Search

- The elements **must** be sorted (alphabetically, numerically, some order) or a binary search will not work.
- Best case scenario: The information sought is the middle element.
- Worst case scenario: You check (at most) half of the elements in an array.

Bubble Sort Algorithm

- The Bubble Sort is a **comparative** sorting algorithm where neighboring pairs of values in an array are compared and swapped so that they are in the correct order.
- The algorithm iterates through the entire array, performing any swaps when necessary.
 - It repeats this process n-number of times, where n is the length of the array.

Bubble Sort Algorithm

- After the first iteration, the largest number in the array will have been moved (repeatedly swapped) into the last index of the array.
 - Assuming we are sorting in ascending order (smallest to largest)
- After the next iteration, the next largest number will have been moved to the second from last index of the array.
 - The next iteration will move the next largest number to the third from last index, and so on.
- Doing this process n-number of times guarantees the array will have all elements in the correct order.

Bubble Sort Algorithm

- Here is a link to a video with a visualization of the Bubble Sort sorting a short array of numbers.

https://www.youtube.com/watch?v=xli_FI7CuzA

- The pseudocode at the end of the video is a little different from the pseudocode on the next slide (and the following C++ algorithm).

Bubble Sort (Pseudocode)

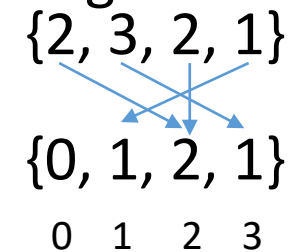
```
For  $i$  in indexes 0 through length-1 of array  $a$  :  
    For  $j$  in indexes 1 through length-1 :  
        If  $a[j-1] > a[j]$  :  
            Swap  $a[j-1]$  and  $a[j]$ 
```

Bubble Sort (C++ Function)

```
void bubbleSort(int a[], int length) {  
    for(int i = 0; i < length; i++) {  
        for(int j = 1; j < length; j++) {  
            if(a[j-1] > a[j]) {  
                int temp = a[j-1];  
                a[j-1] = a[j];  
                a[j] = temp;  
            }  
        }  
    }  
}
```

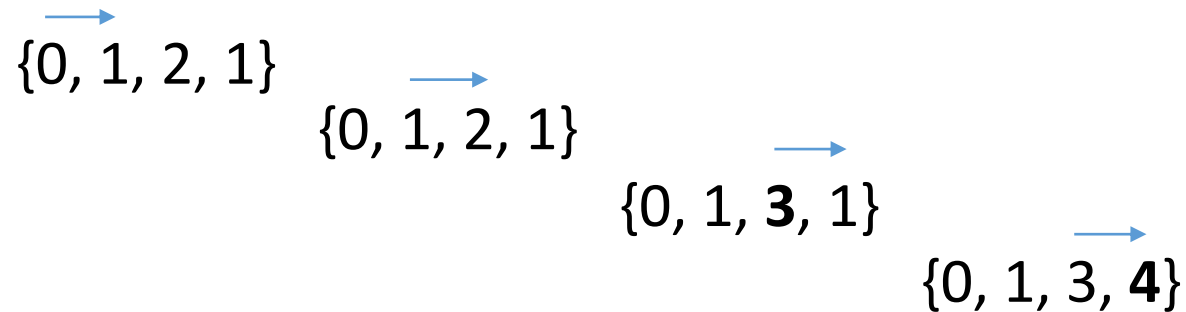
Counting Sort

- The Counting Sort is a **non-comparative** sorting algorithm that uses a separate “counting” array for determining how many times an integer appears in an unsorted array.
- The counting array uses its own **indexes** to hold the totals of that **corresponding value** in the unsorted array.
 - For example, for the unsorted array {2, 3, 2, 1} a Counting Sort’s counting array would contain {0, 1, 2, 1}
 - Zero 0’s, One 1, Two 2’s, One 3



Counting Sort

- The values in the counting array are summed linearly.
 - The counting array previously shown would become {0, 1, 3, 4}



Counting Sort

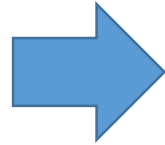
- The counting array (c) is then used to determine the placement of each unsorted element.
 - The result is a second, sorted array (r).
 - **This algorithm does not change the original array (a).**
- Each value is retrieved from the original array.
 - 1 is subtracted from corresponding index of the counting array.
 - If 2 is retrieved from the original array, 1 is subtracted from index 2 of the counting array.
- The new value at the corresponding index is the index where the value is placed in the resulting array.

Counting Sort

$a = \{\mathbf{2}, 3, 2, 1\}$

$c = \{0, 1, 3, 4\}$

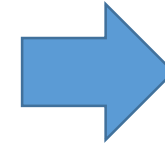
$r = \{0, 0, 0, 0\}$



$a = \{\mathbf{2}, 3, 2, 1\}$

$c = \{0, 1, \mathbf{2}, 4\}$

$r = \{0, 0, 0, 0\}$



$a = \{\mathbf{2}, 3, 2, 1\}$

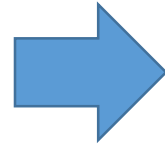
$c = \{0, 1, \mathbf{2}, 4\}$

$r = \{0, 0, \mathbf{2}, 0\}$

$a = \{2, \mathbf{3}, 2, 1\}$

$c = \{0, 1, 2, 4\}$

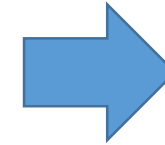
$r = \{0, 0, 2, 0\}$



$a = \{2, \mathbf{3}, 2, 1\}$

$c = \{0, 1, 2, \mathbf{3}\}$

$r = \{0, 0, 2, 0\}$



$a = \{2, \mathbf{3}, 2, 1\}$

$c = \{0, 1, 2, \mathbf{3}\}$

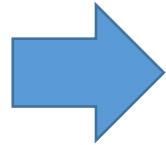
$r = \{0, 0, 2, \mathbf{3}\}$

Counting Sort

$a = \{2, 3, \mathbf{2}, 1\}$

$c = \{0, 1, 2, 3\}$

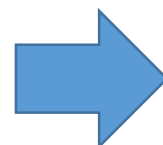
$r = \{0, 0, 2, 3\}$



$a = \{2, 3, \mathbf{2}, 1\}$

$c = \{0, 1, \mathbf{1}, 3\}$

$r = \{0, 0, 2, 3\}$



$a = \{2, 3, \mathbf{2}, 1\}$

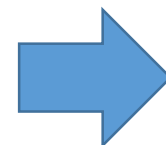
$c = \{0, 1, \mathbf{1}, 3\}$

$r = \{0, \mathbf{2}, 2, 3\}$

$a = \{2, 3, 2, \mathbf{1}\}$

$c = \{0, 1, 1, 3\}$

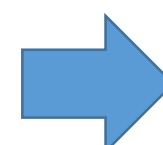
$r = \{0, 2, 2, 3\}$



$a = \{2, 3, 2, \mathbf{1}\}$

$c = \{0, \mathbf{0}, 1, 3\}$

$r = \{0, 2, 2, 3\}$



$a = \{2, 3, 2, \mathbf{1}\}$

$c = \{0, \mathbf{0}, 1, 3\}$

$r = \{\mathbf{1}, 2, 2, 3\}$

Counting Sort (Pseudocode)

Determine maximum integer value, max , in array a

Create an array, c , of $max + 1$ length

Put a 0 at every index of c

For each element, x , in a :

 Add 1 to $c[x]$

For i in indexes 1 through length-1 of c :

 Add $c[i-1]$ to $c[i]$

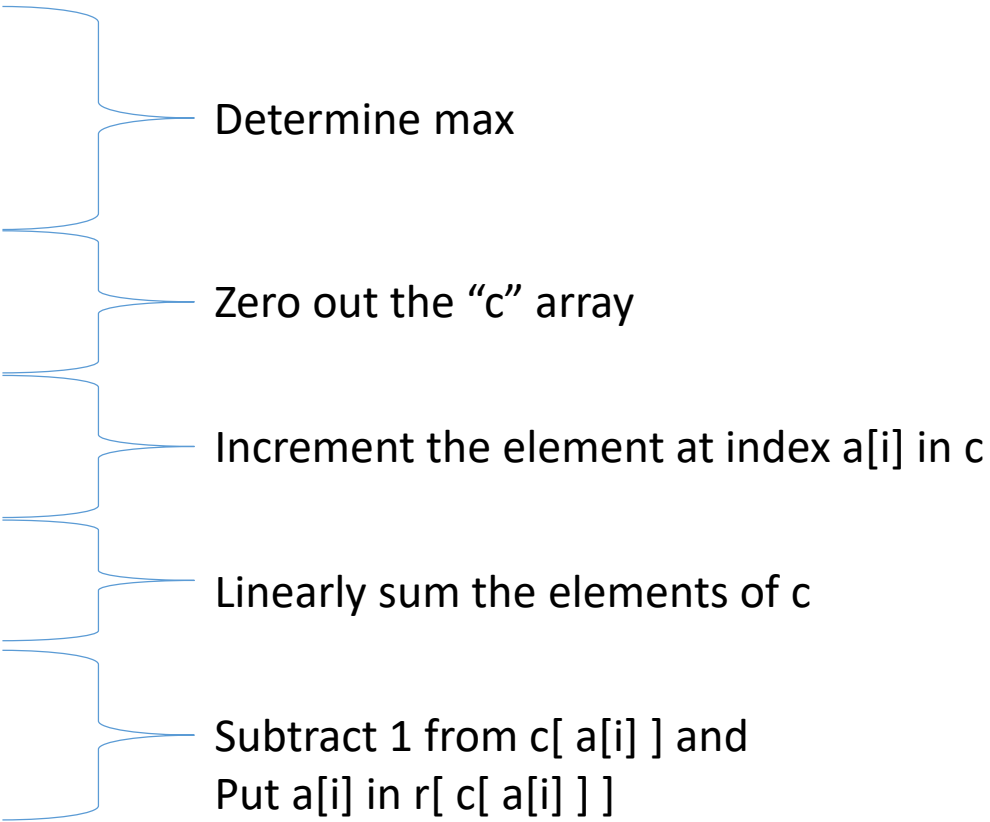
For each element, x , in a :

 Subtract 1 from $c[x]$

 Assign x to $r[c[x]]$

Counting Sort (C++ Function)

```
void countingSort(int a[], int r[], int length) {  
    int max = a[0];  
    for(int i = 1; i < length; i++) {  
        if(a[i] > max) {  
            max = a[i];  
        }  
    }  
    int c[max + 1];  
    for(int i = 0; i < max+1; i++) {  
        c[i] = 0;  
    }  
    for(int i = 0; i < length; i++) {  
        int value = a[i];  
        c[value] += 1;  
    }  
    for(int i = 1; i < max + 1; i++) {  
        c[i] += c[i-1];  
    }  
    for(int i = 0; i < length; i++) {  
        int temp = a[i];  
        c[temp] -= 1;  
        r[c[temp]] = temp;  
    }  
}
```



- Determine max
- Zero out the “c” array
- Increment the element at index a[i] in c
- Linearly sum the elements of c
- Subtract 1 from c[a[i]] and Put a[i] in r[c[a[i]]]

Counting Sort Time Complexity

- $O(n + k)$
 - Where k is the length of the counting array.
- $\Omega(n + k)$
- $\Theta(n + k)$
- Counting sort always performs in linear time.
 - However, **it can only sort non-negative integers**.
 - It cannot sort arrays of floating point numbers or arrays with negative integers.

Counting Sort Space Complexity

- $O(k)$
- Sometimes it is inefficient with the auxiliary space.
 - Array to sort: {4, 10, 3}
 - Counting array: {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1}
- Sometimes it is efficient with the auxiliary space.
 - Array to sort: {1, 2, 0, 1, 2, 0, 1, 2, 2, 0, 1}
 - Counting array: {3, 4, 4}

Radix Sort

- The Radix Sort is another non-comparative sorting algorithm that is closely related to the Counting Sort algorithm.
- The Radix Sort sorts an array of numbers, going digit-by-digit of each value, starting with the least-significant digit to the most-significant digit.
 - The algorithm uses a Counting Sort to perform the actual sorting.

Radix Sort

$a = \{45, 32, 7, 19\}$

$c = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$



$a = \{\underline{4}5, \underline{3}5, \underline{7}, \underline{1}9\}$

$c = \{0, 0, 0, 0, 0, 2, 0, 1, 0, 1\}$



$c = \{0, 0, 0, 0, 0, 2, 2, 3, 3, 4\}$



$a = \{35, 45, 7, 19\}$

$a = \{35, 45, 7, 19\}$

$c = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$



$a = \{\underline{3}5, \underline{4}5, \underline{0}7, \underline{1}9\}$

$c = \{1, 1, 0, 1, 1, 0, 0, 1, 0, 1\}$



$c = \{1, 2, 2, 3, 4, 4, 4, 5, 6, 7\}$



$a = \{7, 19, 35, 45\}$

Radix Sort (C++ Function)

```
void radixSort(int a[], int length) {  
    int max = getMax(a, length);  
    for (int i = 1; max/i > 0; i *= 10) {  
        countingSort(a, length, i);  
    }  
}
```

```
int getMax(int a[], int length) {  
    int max = a[0];  
    for (int i = 1; i < length; i++) {  
        if (a[i] > max) {  
            max = a[i];  
        }  
    }  
    return max;  
}
```

```
void countingSort(int a[], int length, int i) {  
    int temp[length];  
    int digitCount[10] = {0};  
  
    for (int j = 0; j < length; j++) {  
        digitCount[(a[j] / i) % 10]++;  
    }  
    for (int j = 1; j < 10; j++) {  
        digitCount[j] += digitCount[j - 1];  
    }  
  
    for (int j = length - 1; j >= 0; j--) {  
        int index = digitCount[(a[j] / i) % 10] - 1;  
        temp[index] = a[j];  
        digitCount[(a[j] / i) % 10]--;  
    }  
    for (int j = 0; j < length; j++) {  
        a[j] = temp[j];  
    }  
}
```

Radix Sort (C++ Function)

```
void radixSort(int a[], int length) {  
    int max = getMax(a, length);  
    for (int i = 1; max/i > 0; i *= 10) {  
        countingSort(a, length, i);  
    }  
}
```

Find the largest number; This will determine how many times the below loop repeats

Iterates from 1, to 10, to 100, and so on... based on the largest number

Passes the array, its length, and the current position to sort to a Counting Sort algorithm

Radix Sort (C++ Function)

```
void countingSort(int a[], int length, int i) {  
    int temp[length];  
    int digitCount[10] = {0};  
  
    for (int j = 0; j < length; j++) {  
        digitCount[(a[j] / i) % 10]++;  
    }  
    for (int j = 1; j < 10; j++) {  
        digitCount[j] += digitCount[j - 1];  
    }  
  
    for (int j = length - 1; j >= 0; j--) {  
        int index = digitCount[(a[j] / i) % 10] - 1;  
        temp[index] = a[j];  
        digitCount[(a[j] / i) % 10]--;  
    }  
  
    for (int j = 0; j < length; j++) {  
        a[j] = temp[j];  
    }  
}
```

Counting array with a length of 10 (indexes 0-9)

Finds the digit at the i position of each number in a, and adds one at that index in the counting array

Linearly sums the values in the counting array

Subtracts 1 from the value in the counting array, for each value in a (based on the current digit/position it is sorting for). Puts the at the calculated index in a temporary array. Decrements the index by one.

Copies all values from the temporary array to the actual array (replacing the original ordering)

Radix Sort Complexities

- Time
 - $O(n * k)$
 - Where k is the number of digits in the largest value.
 - $\Omega(n * k)$
 - $\Theta(n * k)$
- Space
 - $O(n + k)$
- Radix sort always performs in linear time.
 - Like the Counting Sort, **it can only sort non-negative integers.**
 - It cannot sort arrays of floating point numbers or arrays with negative integers.