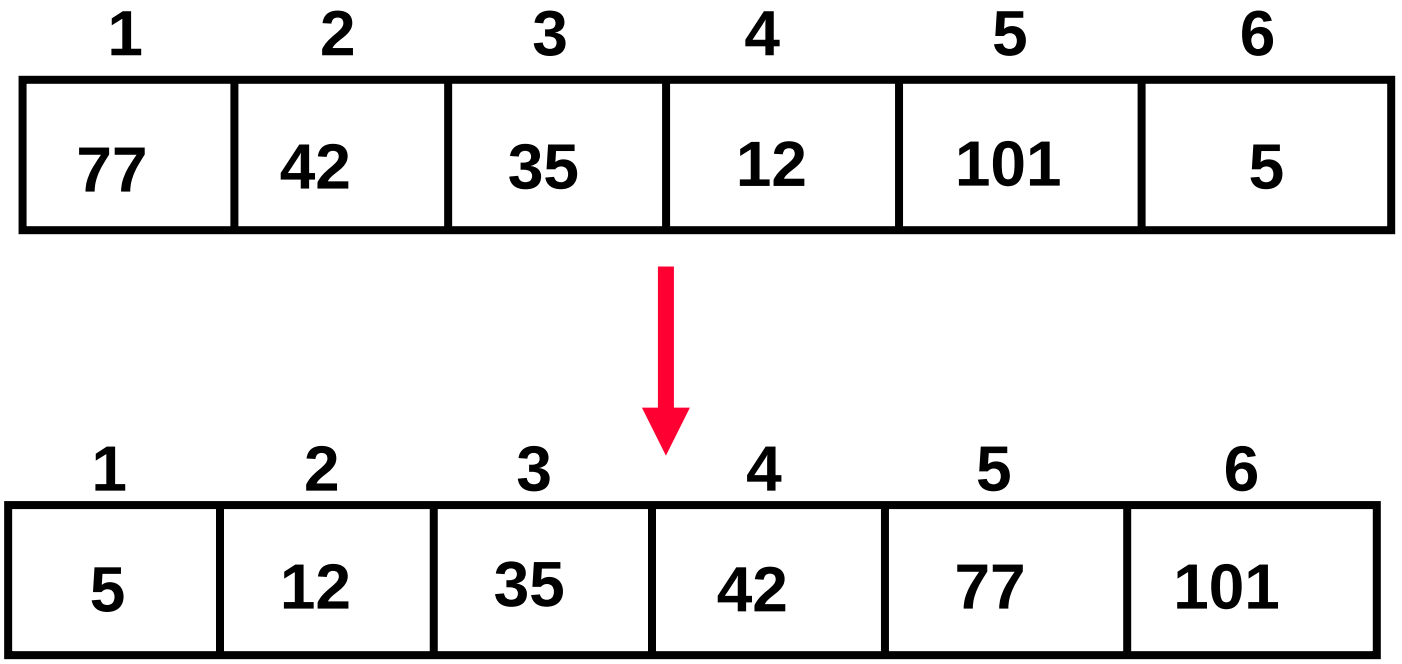
OOP345 – Sorting And Searching Using C++

Sukhbir Tatla

sukhbir.tatla@senecacollege.ca

Sorting

- **Sorting takes an unordered collection and makes it an ordered one.**



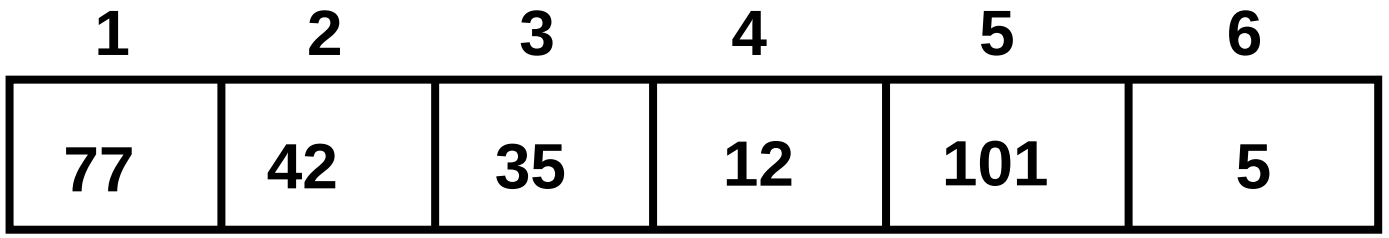
Bubble Sort

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

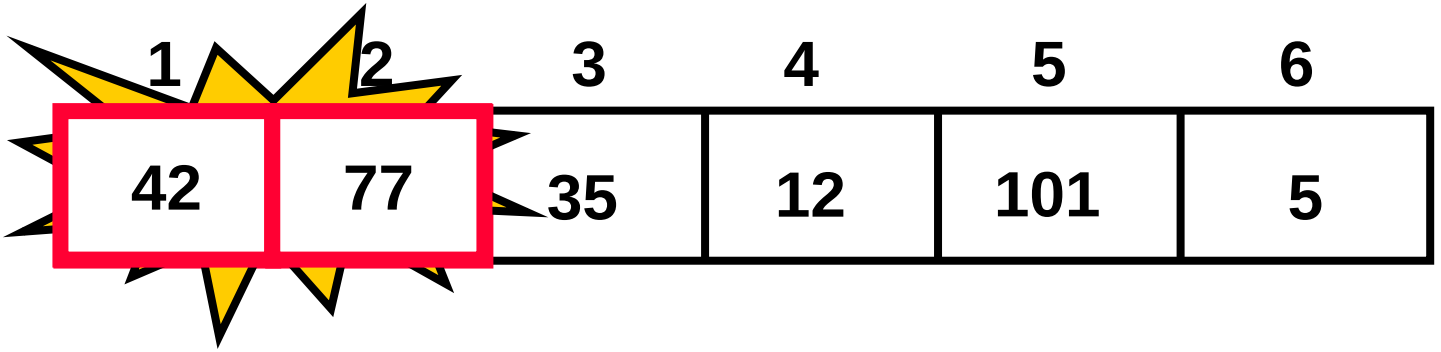
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



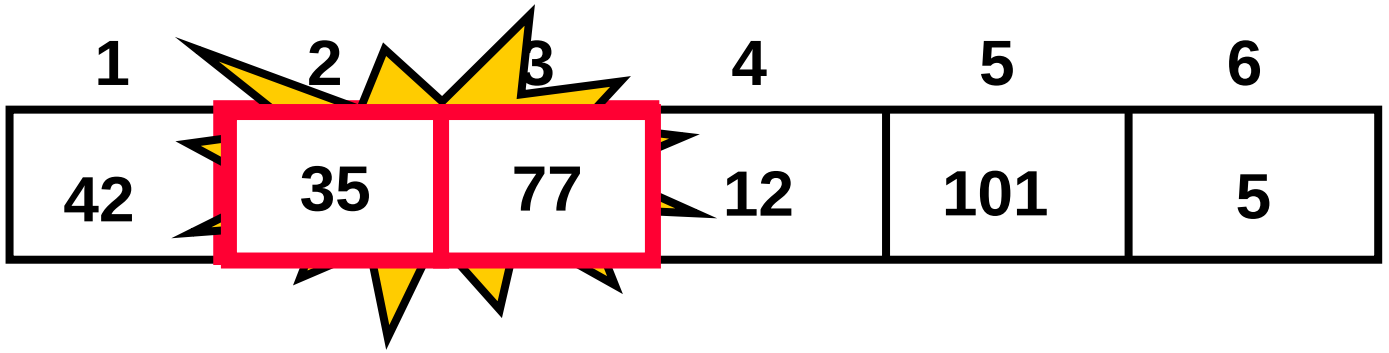
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



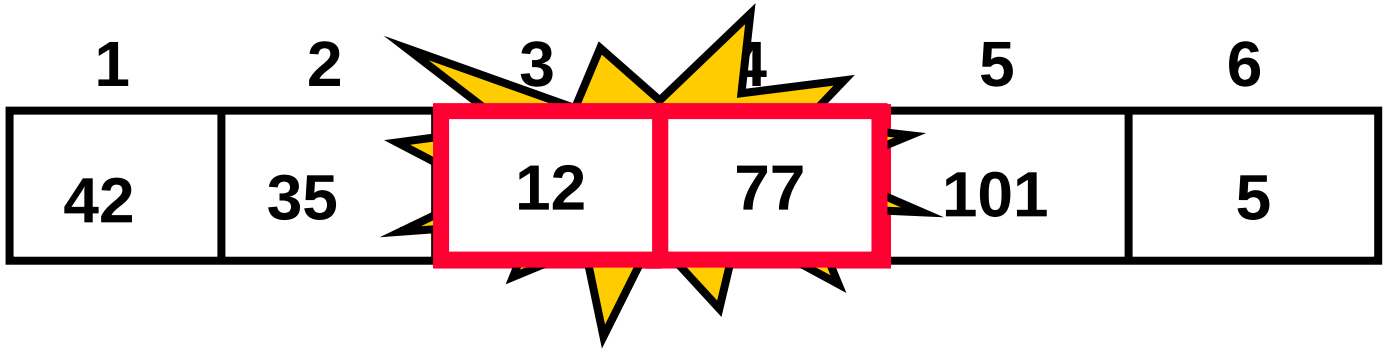
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



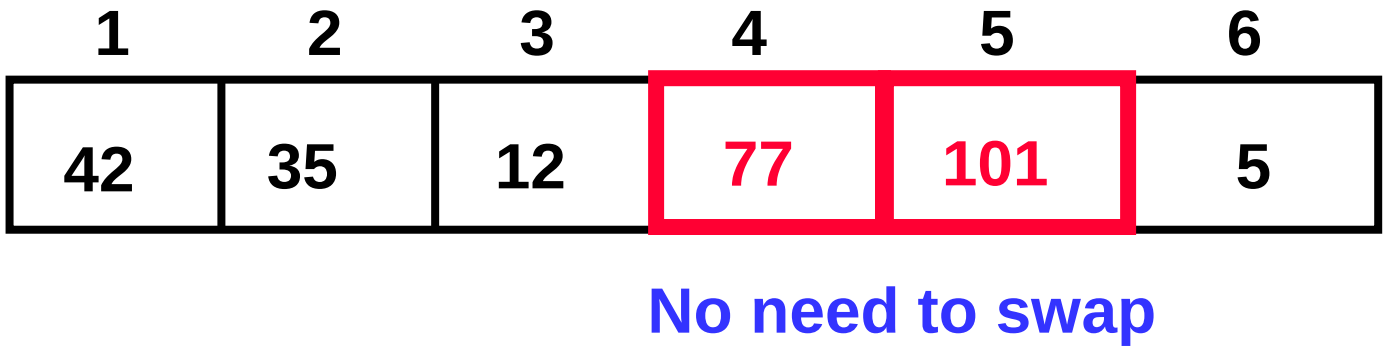
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



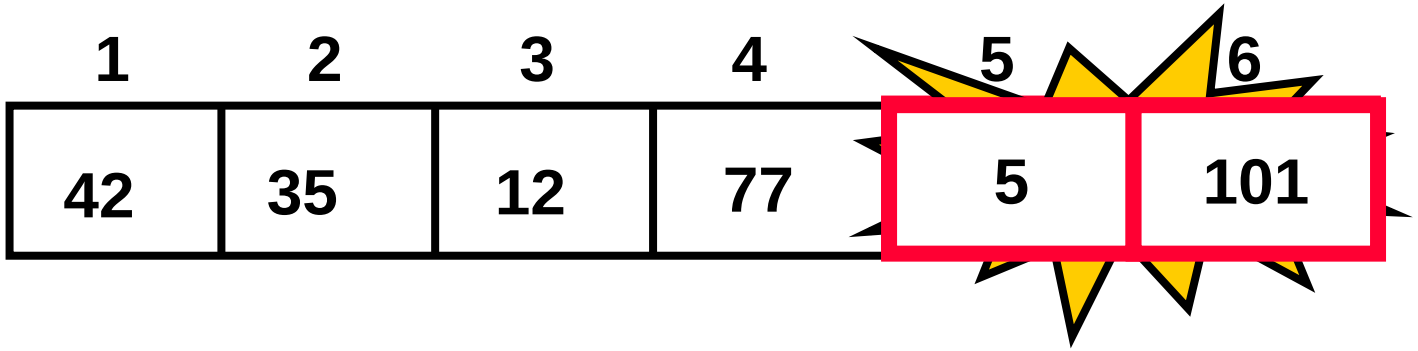
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



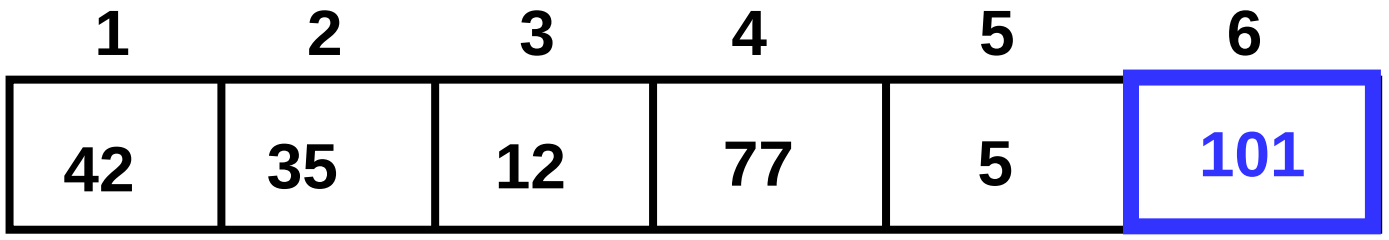
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

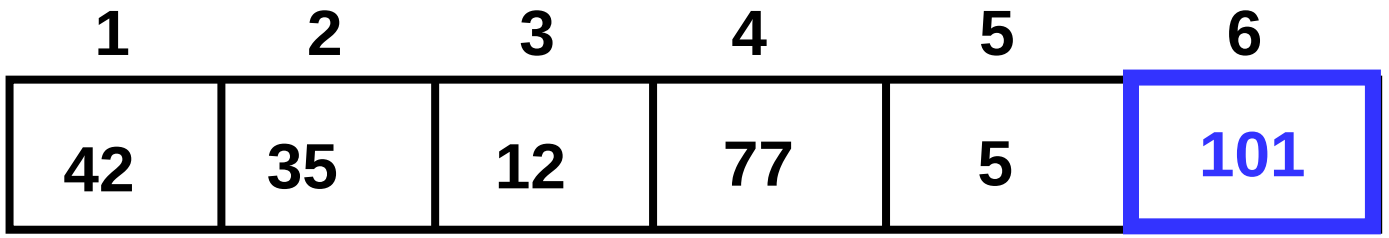
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



Largest value correctly placed

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

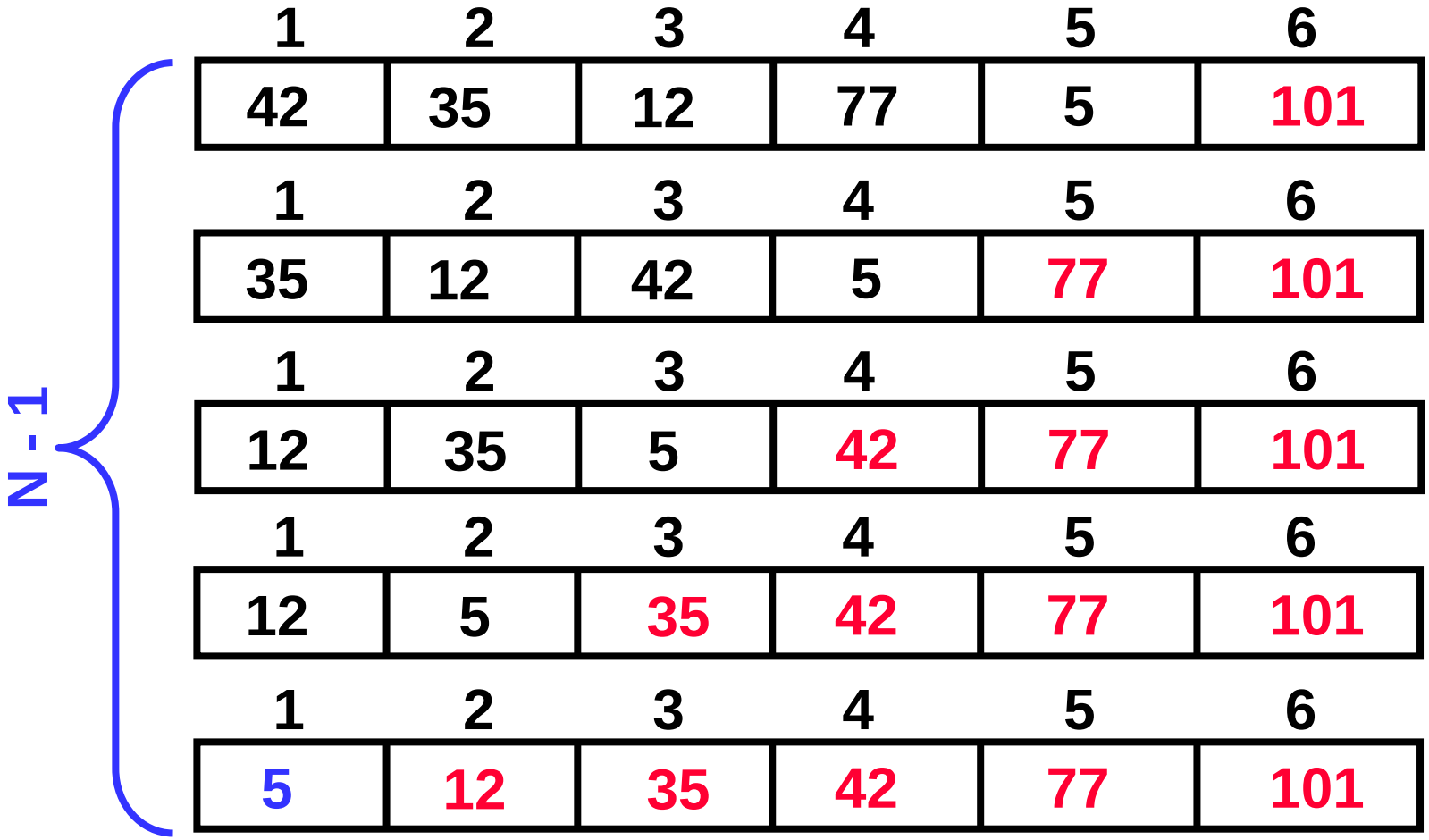


Largest value correctly placed

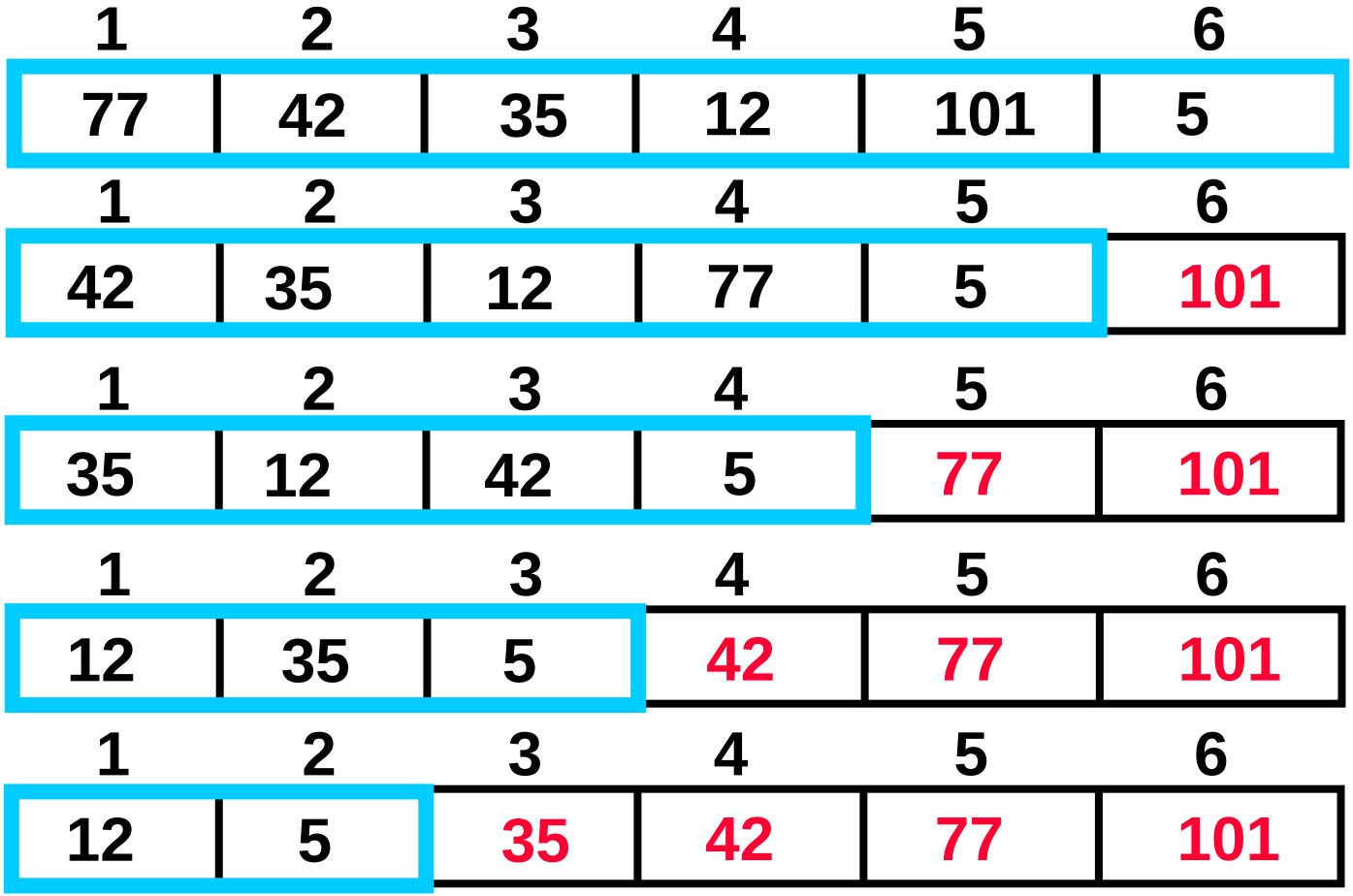
Repeat "Bubble Up" How Many Times?

- **If we have N elements…**
- **And if each time we bubble an element, we place it in its correct location…**
- **Then we repeat the "bubble up" process N – 1 times.**
- **This guarantees we'll correctly place all N elements.**

“Bubbling” All the Elements



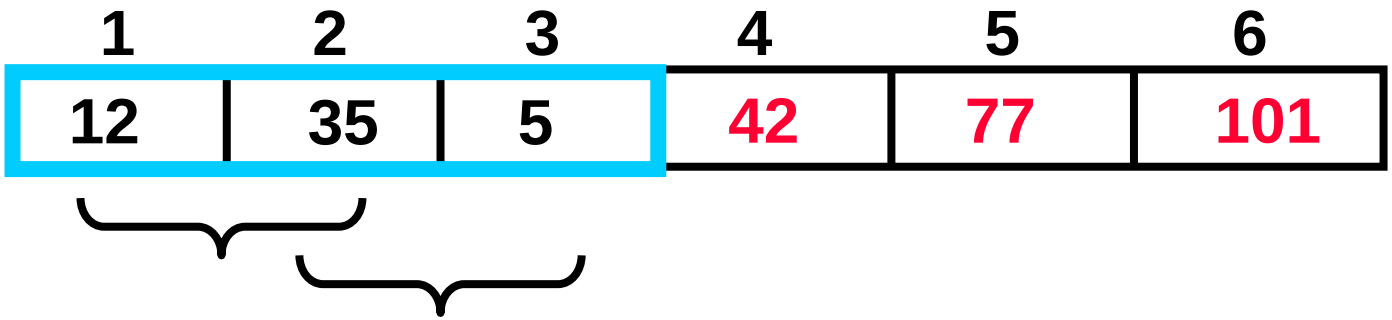
Reducing the Number of Comparisons



Reducing the Number of Comparisons

- On the Nth "bubble up", we only need to do MAX-N comparisons.

- For example:
  - This is the 4th "bubble up"
  - MAX is 6
  - Thus we have 2 comparisons to do



Bubble Sort Implementation

Here is an ascending-order implementation of the Bubble Sort algorithm for integer arrays:

```
void BubbleSort(int List[] , int Size) {
    int tempInt;   // temp variable for swapping list
    for (int Stop = Size - 1; Stop > 0; Stop--) {
        // make a pass
        for (int Check = 0; Check < Stop; Check++) {
                // compare elements
                if (List[Check] > List[Check + 1]) {
                // swap if in the wrong order
                tempInt           = List[Check];
                List[Check]       = List[Check + 1];
                List[Check + 1]   = tempInt;
            }
        }
    }
}
```

Summary

- "Bubble Up" algorithm will move largest value to its correct location (to the right)
- Repeat "Bubble Up" until all elements are correctly placed:
  - Maximum of N-1 times
  - Can finish early if no swapping occurs
- We reduce the number of elements we compare each time one is correctly placed

Selection Sort

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the largest element from the unsorted sublist and swap it with the element at the end of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of *n* elements requires *n-1* passes to completely rearrange the data.

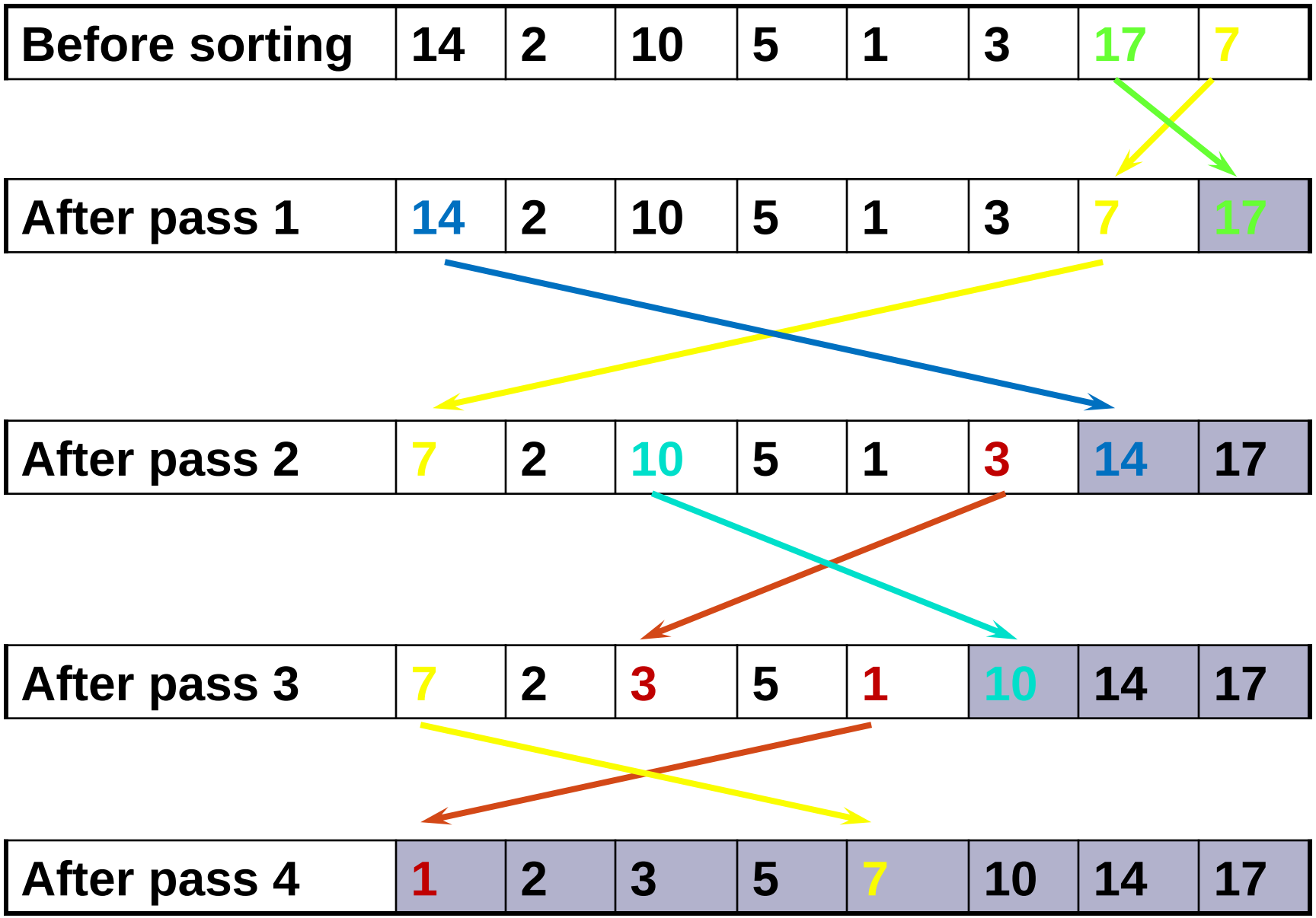
Selection Sort

- Algorithm
  1. Define the entire array as the unsorted portion of the array
  2. While the unsorted portion of the array has more than one element:

     ⇒ Find its largest element.

     ⇒ Swap with last element (assuming their values are different).

     ⇒ Reduce the size of the unsorted portion of the array by 1.


Before sorting
14
2
10
5
1
3
17
7
After pass 1
14
2
10
5
1
3
7
17
After pass 2
7
2
10
5
1
3
14
17
After pass 3
7
2
3
5
1
10
14
17
After pass 4
1
2
3
5
7
10
14
17

Selection Sort Implementation

```
void SelectionSort(int List[], int Size) {
   int Begin, SmallSoFar, Check;
      void Swap(int& Elem1, int& Elem2);
   for (Begin = 0; Begin < Size - 1; Begin++) {
       SmallSoFar = Begin;  // set head of tail
      // scan current tail
       for (Check = Begin + 1; Check < Size;
Check++) {
          if (List[Check] < List[SmallSoFar])
             SmallSoFar = Check;
       }
     // put smallest elem at front of current tail
       Swap(List[Begin], List[SmallSoFar]); }
}

void Swap(int& Elem1, int& Elem2) {
   int tempInt;
   tempInt = Elem1;
   Elem1   = Elem2;
   Elem2   = tempInt;       }
```

```
// Sort array of strings in ascending order
void select(char data[][string_size], // in/output:
  array
              int size){  // input: array size
  char temp[string_size];          // for swap
  int max_index;  // index of max value
  for (int rightmost=size-1; rightmost>0; rightmost--){
      // find the largest item
      max_index = 0;
      for(int current=1; current<=rightmost; current++)
                  if (strcmp(data[current],
  data[max_index]) >0)
                max_index = current;
      // swap with last item if necessary
      if(strcmp(data[max_index], data[rightmost])>0){
          strcpy(temp,data[max_index]);    // swap
          strcpy(data[max_index],data[rightmost]);
          strcpy(data[rightmost], temp);
          for (int index=0; index< size; index++)
              cout << data[index] << " ";
          cout<<endl;

      }      }      }
```

```
// Sort an array of strings in ascending order
void bubble(char data[][string_size], // in/output:array
            int size){          // input: array size
char temp[string_size];         // for swap
   for(int outer=size-1 ;  outer>0; outer--){
        for (int inner=0; inner < outer; inner++) {
            // traverse the nested loops
            if ( strcmp(data[inner], data[inner+1])>0 )
            {
                 // swap current element with next
                 // if the current element is greater
                 strcpy(temp, data[inner]);
                 strcpy(data[inner], data[inner+1]);
                 strcpy(data[inner+1], temp);
                 for (int index=0; index< size; index++)
                      cout << data[index] << " ";
                 cout<<endl;
            }
        }  // inner for loop
    }  // outer for loop       }
```

Practice

- REALITY CHECK (Week 9)

- Questions #1 (Word problem)
- Questions #2 (Word Problem)

Searching: Introduction

- Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding the value's location.

- Two popular search algorithms are the simple *linear search* and the faster but more complex *binary search*.

Searching: Algorithm

- Looking up a phone number, accessing a website and checking a word's definition in a dictionary all involve searching through large amounts of data.
- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does, in fact, exist.
- The major difference is the amount of *effort* they require to complete the search.
- One way to describe this *effort* is with Big O notation.
  - For searching and sorting algorithms, this is particularly dependent on the number of data elements.

Linear/Sequential Search

An Example

- **Imagine we have a database of students**
- **I want to know if Bob Smith is in my class.**

- **I want to search my database and have the module print out "IN THE CLASS" or "NOT IN THE CLASS."**

Another Situation

- **I have the same student database.**
- **I need to view Alice Jones' grades and update them.**

- **Again, I need to search the database.**
- **This time, I need Alice's information returned to me so I can view and modify them (i.e. need access).**

Searching: Linear Search

Function Template linearSearch

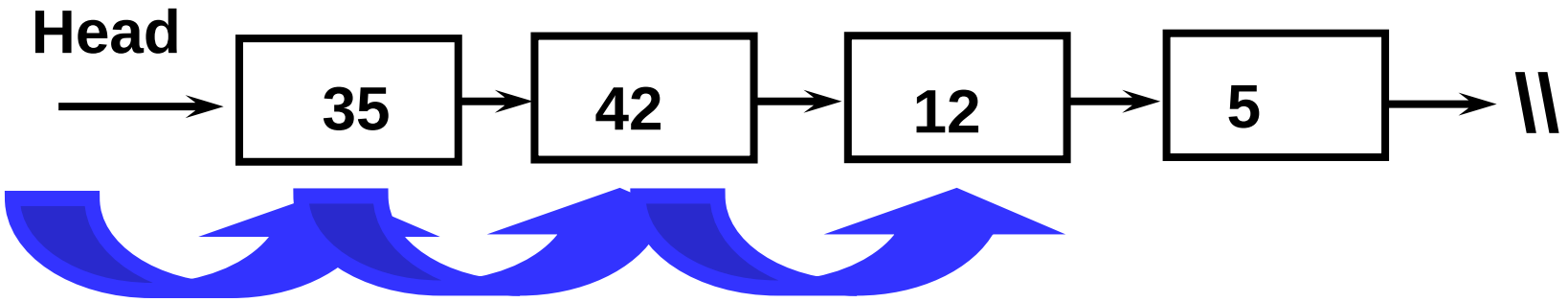
- Function template linearSearch compares each element of an array with a *search key*.
- Because the array is not in any particular order, it's just as likely that the search key will be found in the first element as the last.
- On average, therefore, the program must compare the search key with *half* of the array's elements.
- To determine that a value is *not* in the array, the program must compare the search key to *every* array element.
- Linear search works well for *small* or *unsorted* arrays.
- However, for large arrays, linear searching is inefficient.
- If the array is *sorted* (e.g., its elements are in ascending order), you can use the high-speed binary search technique.

A Linear Search

- A linear search traverses the collection until:
  - The desired element is found
  - Or the collection is exhausted
- If the collection is ordered, I might not have to look at all elements
  - I can stop looking when I know the element cannot be in the collection.

Searching in an Unordered Collection

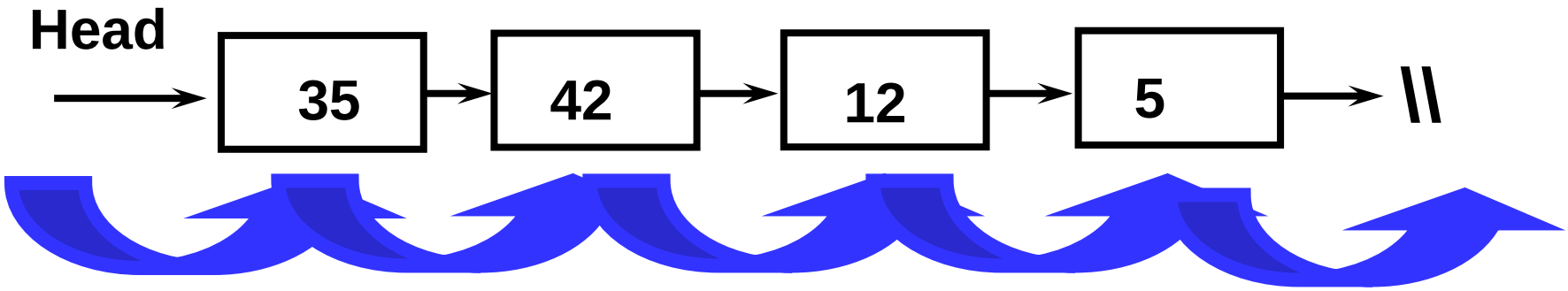
- **Let's determine if the value 12 is in the collection:**



12 Found!

Searching in an Unordered Collection

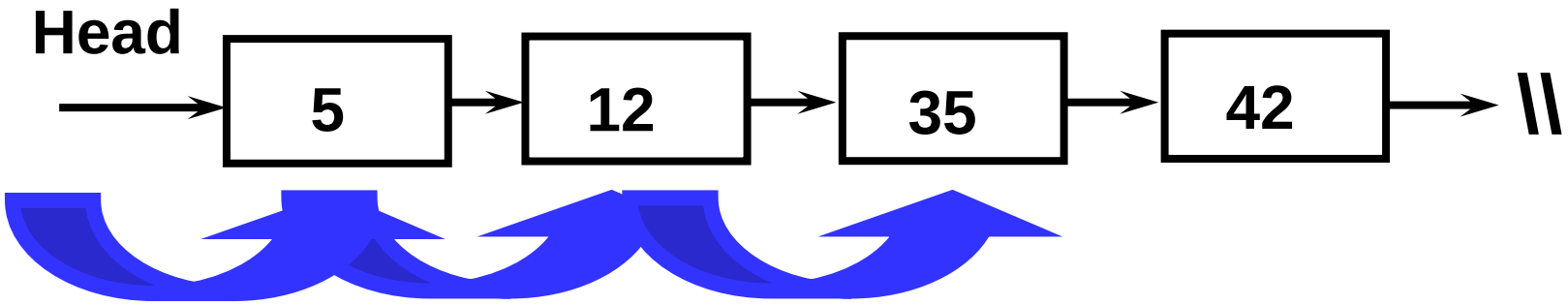
- **Let's determine if the value 13 is in the collection:**



13 Not Found!

Searching in an Ordered Collection

- **Let's determine if the value 13 is in the collection:**



13 Not Found!

```
1  // Fig. 20.2: LinearSearch.cpp
2  // Linear search of an array.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  // compare key to every element of array until location is
8  // found or until end of array is reached; return location of
9  // element if key is found or -1 if key is not found
10 template < typename T, size_t size >
11 int linearSearch( const array< T, size > &items, const T& key )
12 {
13    for ( size_t i = 0; i < items.size(); ++i )
14       if ( key == items[ i ] ) // if found,
15          return i; // return location of key
16
17    return -1; // key not found
18 } // end function linearSearch
19
20 int main()
21 {
22    const size_t arraySize = 100; // size of array
23    array< int, arraySize > arrayToSearch; // create array
24
```

Fig. 20.2 | Linear search of an array. (Part 1 of 3.)

```
25      for ( size_t i = 0; i < arrayToSearch.size(); ++i )
26         arrayToSearch[ i ] = 2 * i; // create some data
27
28      cout << "Enter integer search key: ";
29      int searchKey; // value to locate
30      cin >> searchKey;
31
32      // attempt to locate searchKey in arrayToSearch
33      int element = linearSearch( arrayToSearch, searchKey );
34
35      // display results
36      if ( element != -1 )
37         cout << "Found value in element " << element << endl;
38      else
39         cout << "Value not found" << endl;
40   } // end main
```

```
Enter integer search key: 36
Found value in element 18
```

Fig. 20.2 | Linear search of an array. (Part 2 of 3.)

```
Enter integer search key: 37
Value not found
```

Fig. 20.2 | Linear search of an array. (Part 3 of 3.)

Binary Search

Binary Search

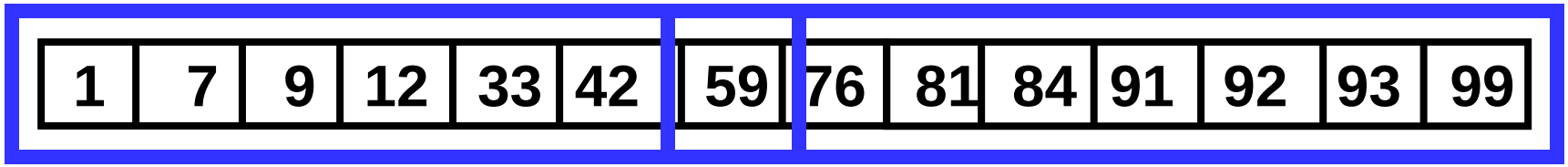
- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the `array` first be sorted.
- This is only worthwhile when the vector, once sorted, will be searched a great many times—or when the searching application has *stringent* performance requirements.
- The first iteration of this algorithm tests the *middle* `array` element.
- If this matches the search key, the algorithm ends.

Binary Search

- Assuming the array is sorted in *ascending* order, then if the search key is *less* than the middle element, the search key cannot match any element in the array's second half so the algorithm continues with only the first *half* (i.e., the first element up to, but *not* including, the middle element).
- If the search key is *greater* than the middle element, the search key cannot match any element in the array's first half so the algorithm continues with only the second *half* (i.e., the element *after* the middle element through the last element).
- Each iteration tests the *middle value* of the array's remaining elements.
- If the element does not match the search key, the algorithm eliminates half of the remaining elements.
- The algorithm ends either by finding an element that matches the search key or by reducing the sub-array to zero size.

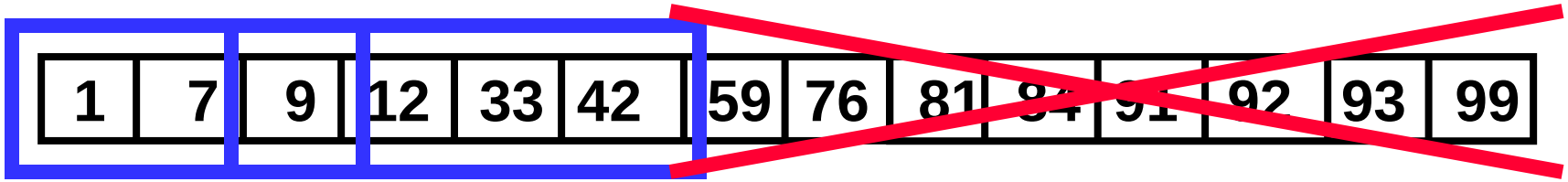
The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



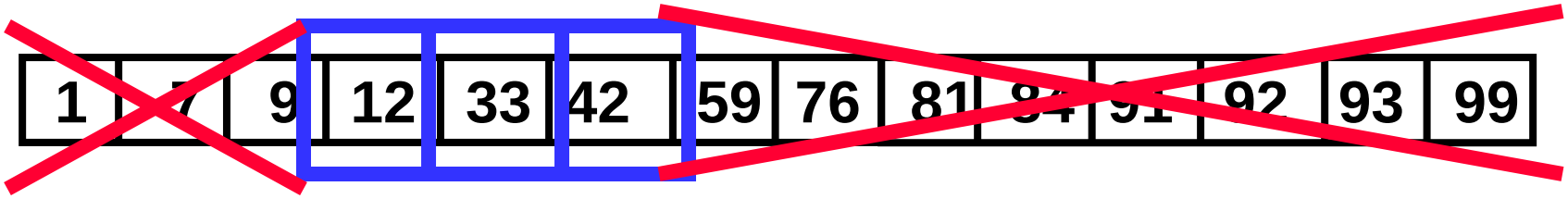
The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



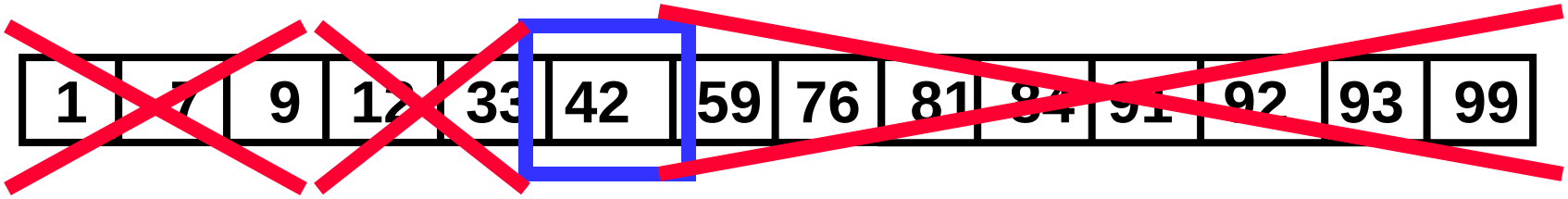
The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



The Binary Search Algorithm

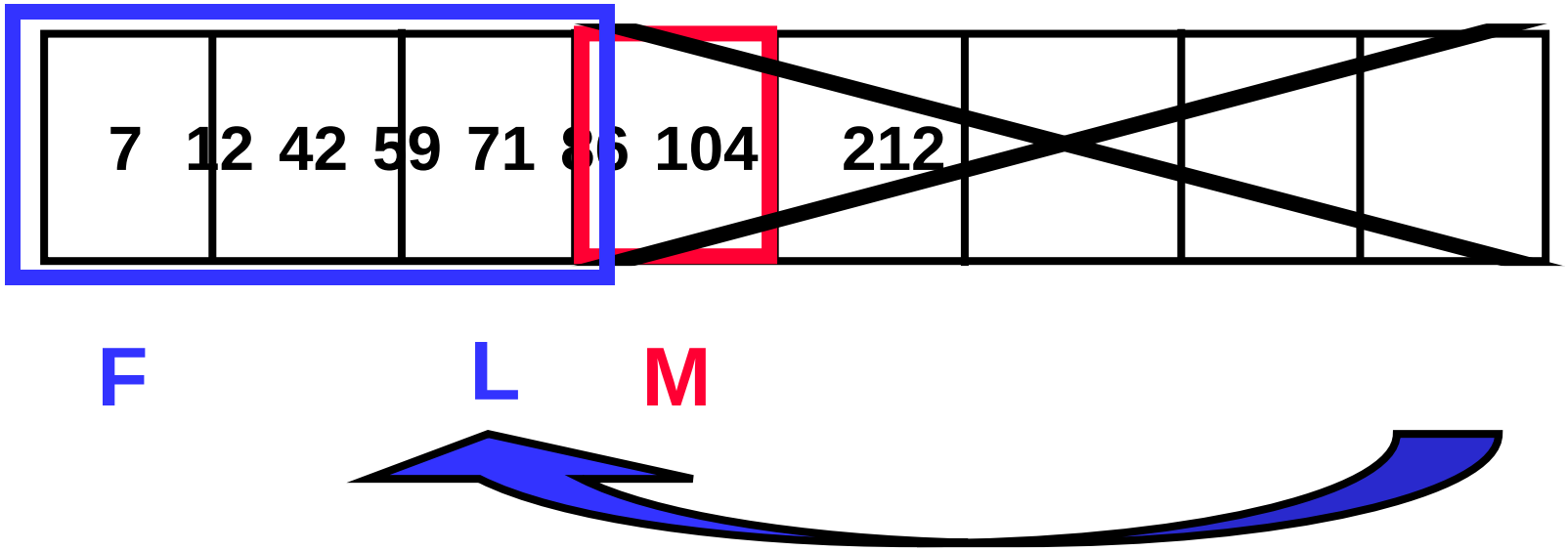
- Return found or not found (true or false), so it should be a function.
- We'll use recursion
- We must pass the entire array into the module each pass, so set bounds (search space) via index bounds (parameters)
  - We'll need a first and last

The Binary Search Algorithm

```
calculate middle position
if (first and last have "crossed") then
   DO NOT FOUND WORK
elseif (element at middle = to_find)
  then
   DO FOUND WORK
elseif to_find < element at middle then
   Look to the left
else
   Look to the right
```

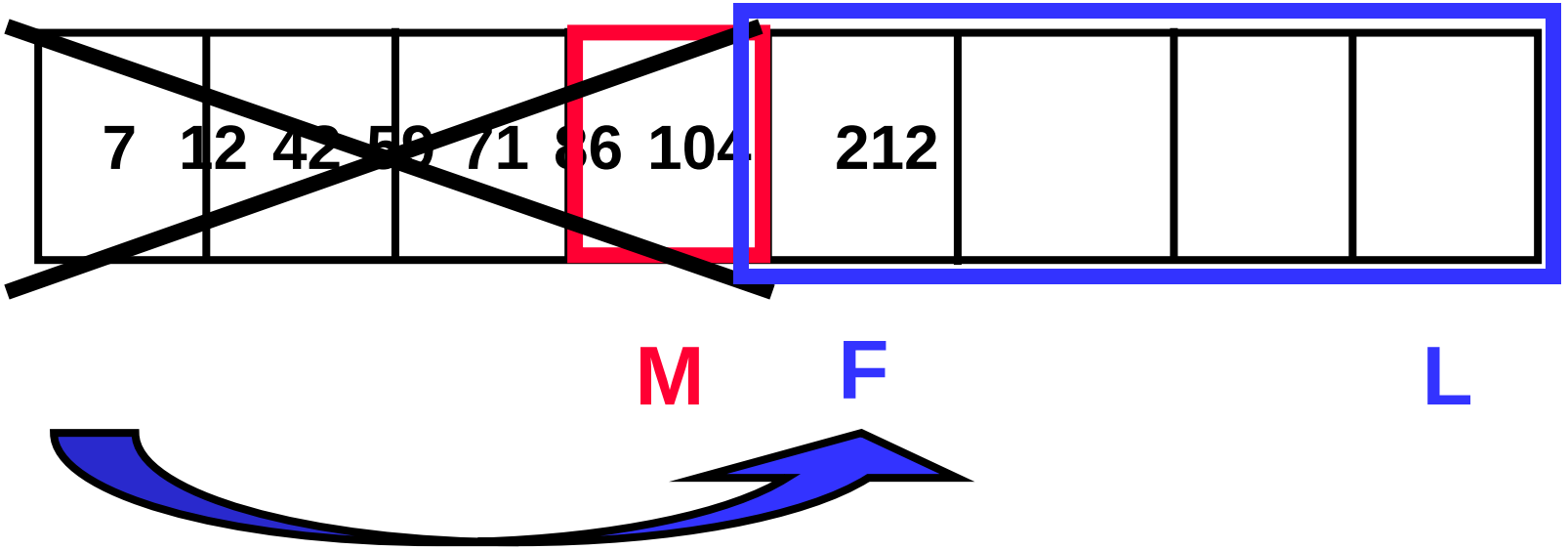
Looking Left

- Use indices "first" and "last" to keep track of where we are looking
- Move left by setting last = middle – 1

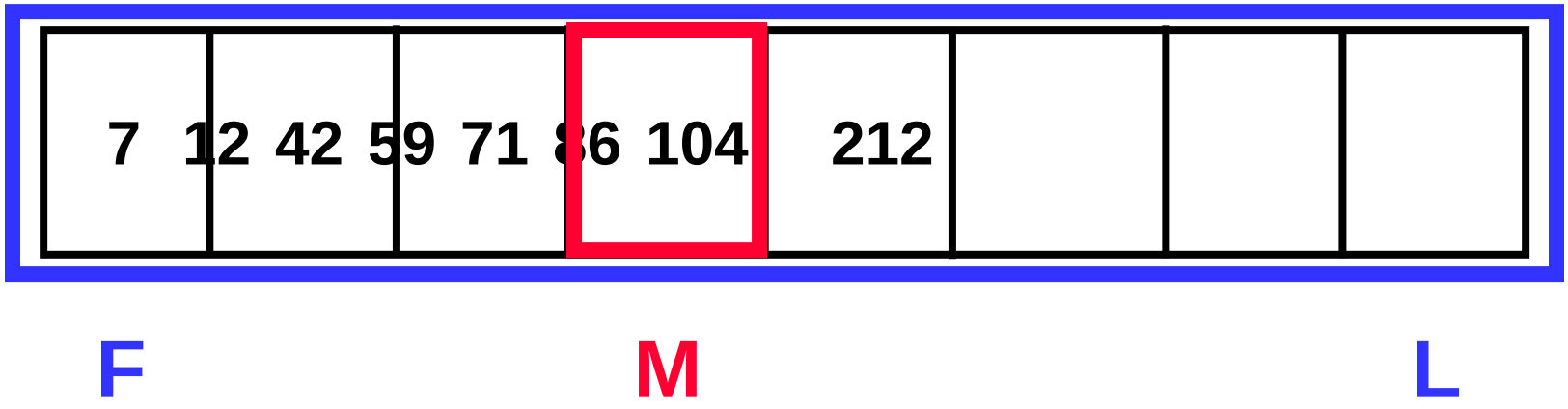


Looking Right

- Use indices "first" and "last" to keep track of where we are looking
- Move right by setting first = middle + 1

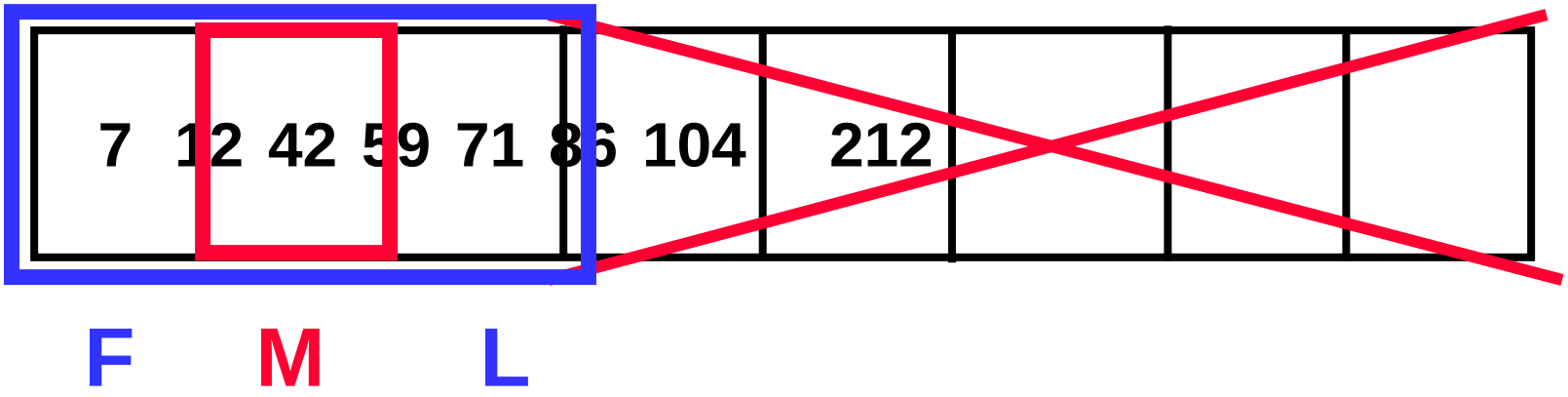


Binary Search Example – Found



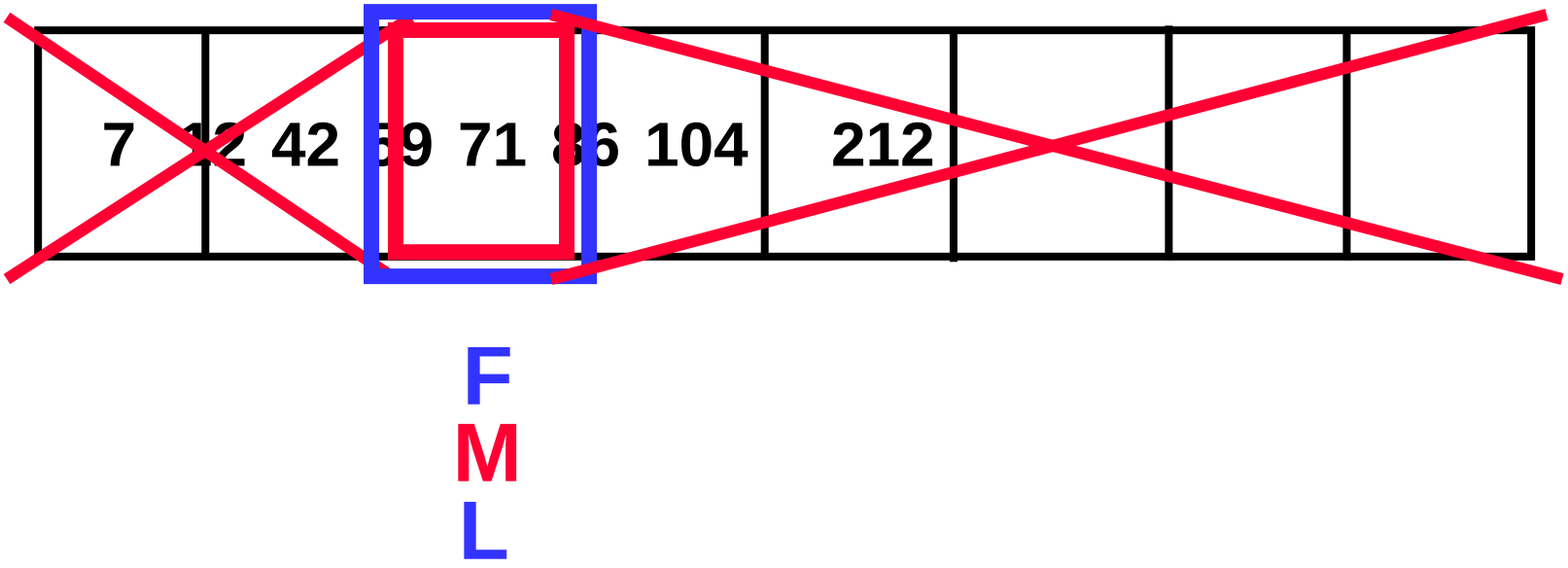
Looking for 42

Binary Search Example – Found



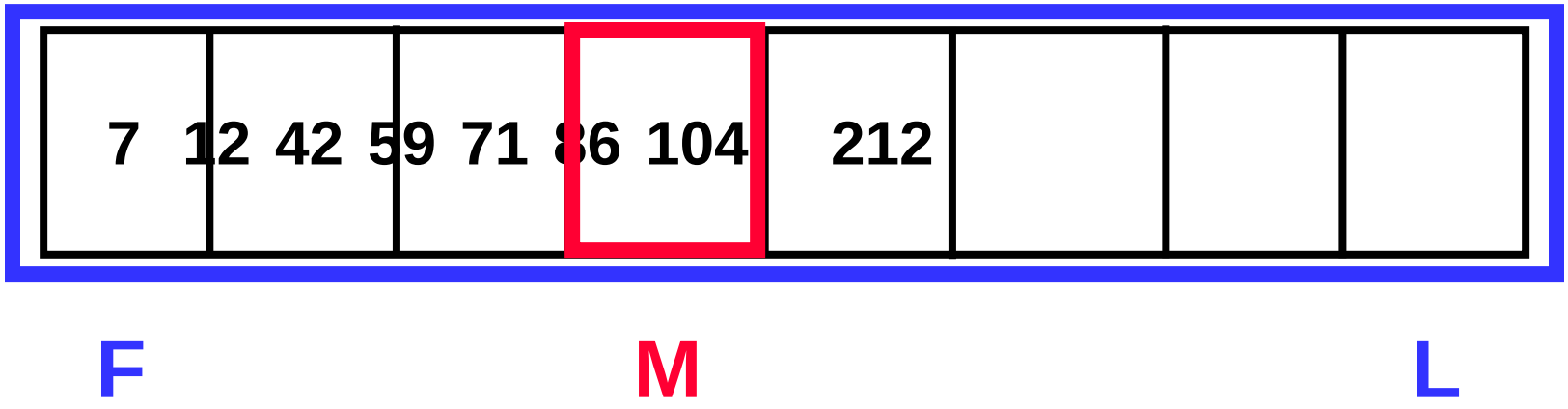
Looking for 42

Binary Search Example – Found



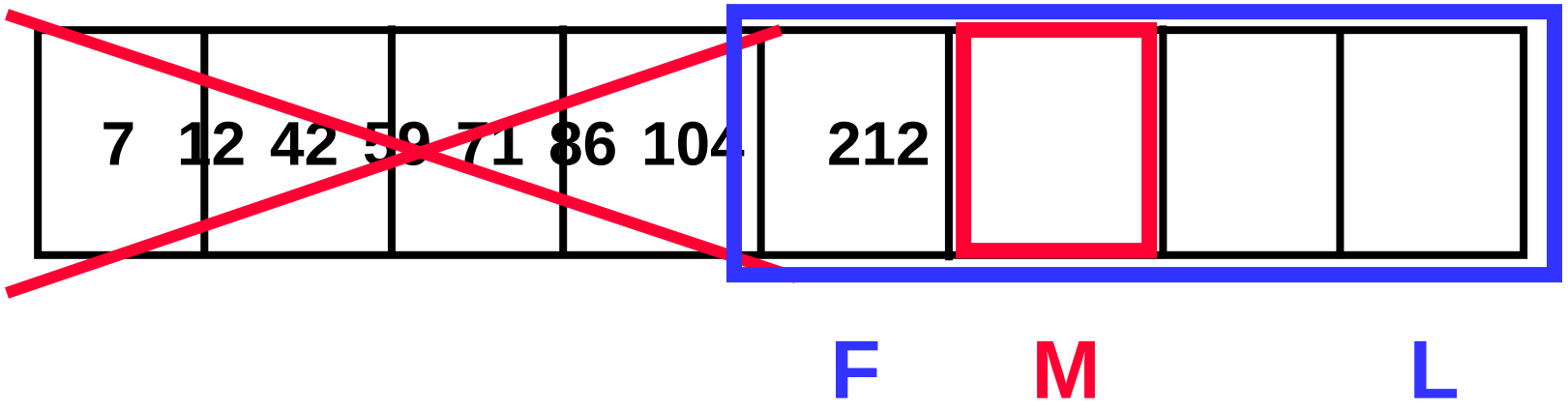
42 found – in 3 comparisons

Binary Search Example – Not Found



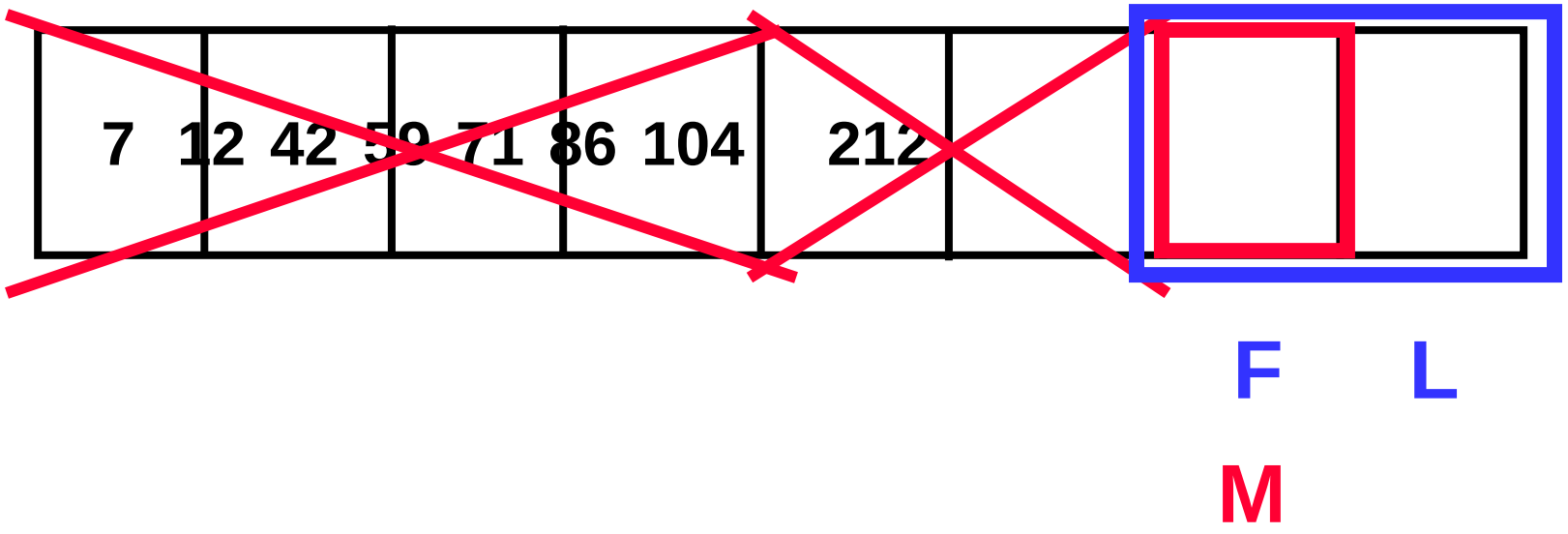
Looking for 89

Binary Search Example – Not Found



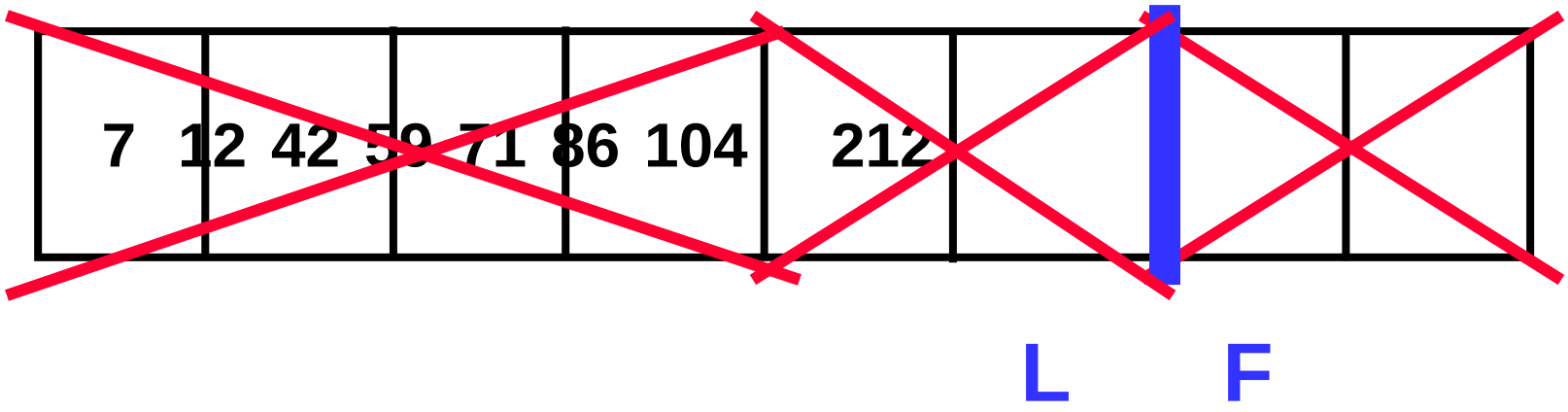
Looking for 89

Binary Search Example – Not Found



Looking for 89

Binary Search Example – Not Found



89 not found – 3 comparisons

```
Function Find returnsa boolean (A isoftype in
  ArrayType, first, last, to_find isoftype in num)
  // contract (Pre, Post, Purpose) here
  middle isoftype num
  middle <- (first + last) div 2
  if (first > last) then
    Find returns false
  elseif (A[middle] = to_find) then
    Find returns true
  elseif (to_find < A[middle]) then
    Find returns Find(A, first, middle-1, to_find)
  else
    Find returns Find(A, middle+1, last, to_find)
endfunction
```

Binary Search Implementation

```
void search(const int a[ ], size_t first, size_t size, int target, bool& found,
    size_t& location)        {
    size_t middle;
    if(size == 0) found = false;
    else {
            middle = first + size/2;
            if(target == a[middle])    {
                    location = middle;
                    found = true;          }
            else if (target < a[middle])
                  // target is less than middle, so search subarray before middle
                    search(a, first, size/2, target, found, location);
            else
                 // target is greater than middle, so search subarray after middle
                    search(a, middle+1, (size-1)/2, target, found, location);
    }
}
```

Summary

- Binary search reduces the work by half at each comparison
- With the array, we need to keep track of which "part" is currently "visible"
- We know the element isn't in the array when our first and last indices "cross"

Practice

- REALITY CHECK (Week 9)

- Questions #3 (Word problem)
- Questions #4 (Word Problem)

THANK YOU!

Any Questions Please?