CPS 188

Computer Programming Fundamentals

Prof. Alex Ufkes

Topic 7.2: Algorithms using arrays

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Today

Array Examples

Searching

Sorting

And more!

Recall

Write a user-defined function that does the following:

Finds the largest element in an array and returns it.

```c
#include <stdio.h>

int find(int arr[], int size)
{
    int i, largest = arr[0];

    for (i = 1; i < size; i++)
    {
       if (arr[i] > largest)
          largest = arr[i];
    }

    return(largest);
}

int main (void)
{
   int nums[6] = {1, -2, 0, 4, -9, 3};
   printf(“largest: %d”, find(nums, 6));
   return (0);
}
```

Initialize largest to the first element

Compare each element to largest. If arr[i] is bigger than largest, set largest to arr[i]

Return largest

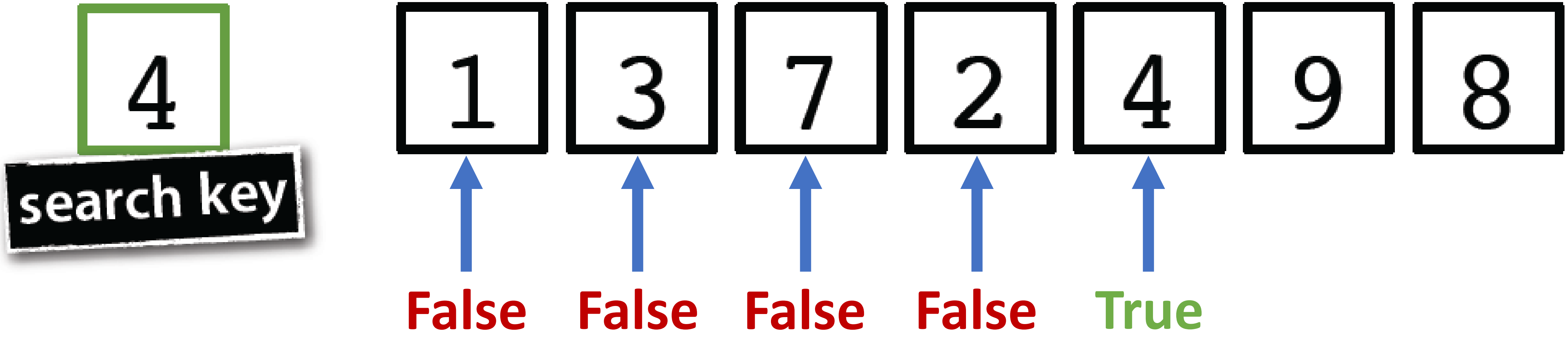
Linear Search

- When finding the maximum value in an array, we make a complete pass through the entire array.
- The same principle applies to finding the minimum value.
- There are many algorithms that rely on making a single, ***linear*** pass through an array…

Linear Search

- Have array, need <u>index</u> of target value.
- Used on ***unsorted*** arrays.
- Meaning? Elements in array are in no particular order.
- Far better options exist for sorted arrays.
- There are a few different flavors, though none do better than *linear time*.
- Linear time? One loop, roughly speaking.

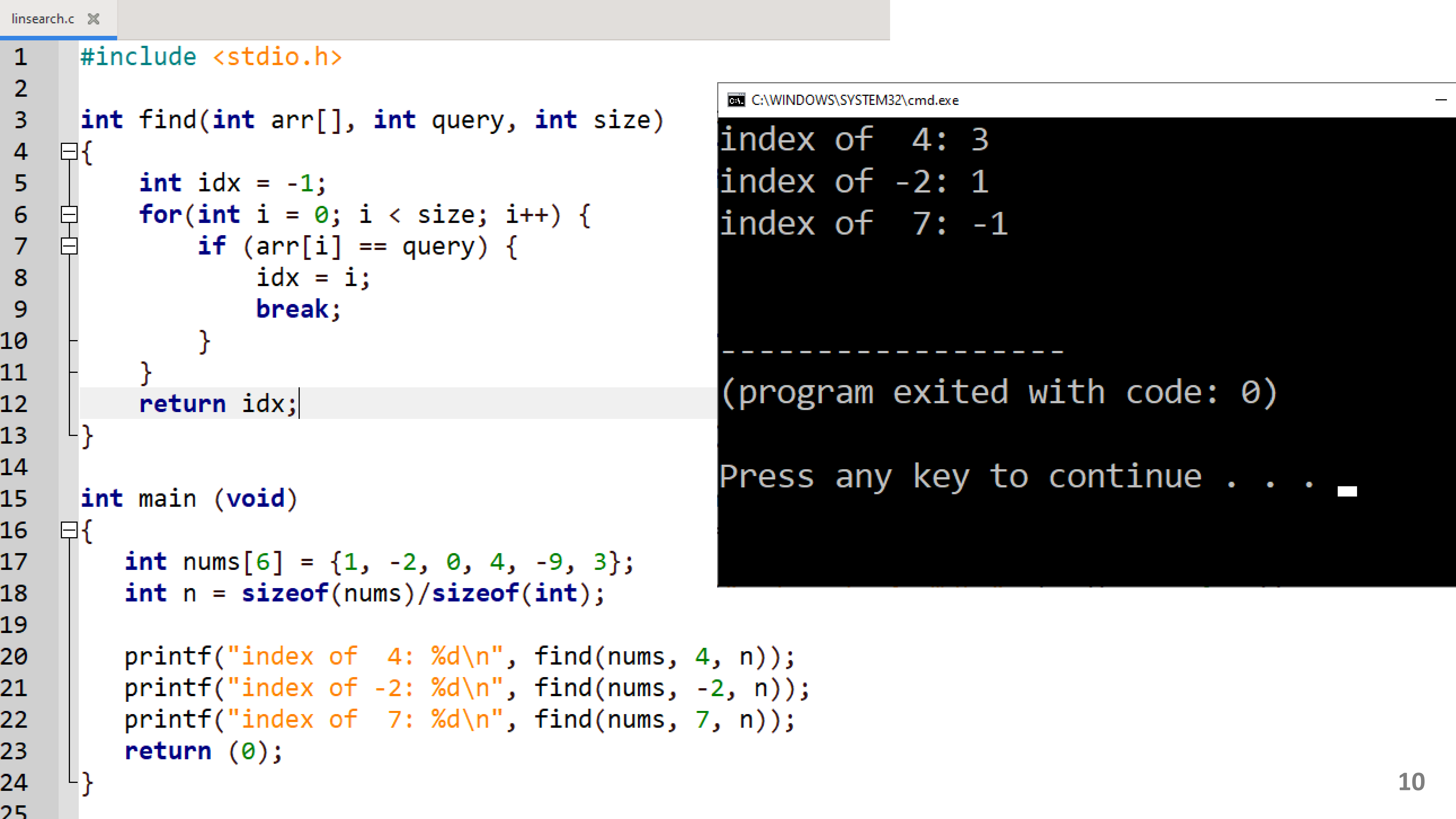
Linear Search: Check every element, one by one, until we find the item we're looking for or hit the end of the array.



`key == items[i]?`

```
int find(int arr[], int query, int size)
{
    int idx = -1;
    for(int i = 0; i < size; i++) {
        if (arr[i] == query) {
            idx = i;
            break;
        }
    }
    return idx;
}
```

- Use the value -1 as an error code
- Indicates that the element was not found.
- Why is this OK? Why not use error code 0?



```c
#include <stdio.h>

int find(int arr[], int query, int size)
{
    int idx = -1;
    for(int i = 0; i < size; i++) {
        if (arr[i] == query) {
            idx = i;
            break;
        }
    }
    return idx;
}

int main (void)
{
   int nums[6] = {1, -2, 0, 4, -9, 3};
   int n = sizeof(nums)/sizeof(int);

   printf("index of  4: %d\n", find(nums, 4, n));
   printf("index of -2: %d\n", find(nums, -2, n));
   printf("index of  7: %d\n", find(nums, 7, n));
   return (0);
}
```

```
index of  4: 3
index of -2: 1
index of  7: -1


------------------
(program exited with code: 0)

Press any key to continue . . .
```

For ***linear search***, it doesn’t matter what the order of the list is.

At worst, we’re required to check every single element.

If we know the array is sorted, is it possible to improve the efficiency of searching?

If you’re looking for a word in the dictionary (assume we’re living in the mid 90s), how do you do it?

Do you start with the first word on the first page and check every single word until you find it?

Aristotle
Bohr
Brahe
Cavendish
Copernicus
Curie

Einstein
Faraday
Galileo
Galton
Hooke
Laplace

Mendeleyev
Morley
Michelson
Newton
Pauling

Find ***Laplace*** in the list

Aristotle
Bohr
Brahe
Cavendish
Copernicus
Curie
Einstein
Faraday
Galileo
Galton
Hooke
Laplace
Mendeleyev
Morley
Michelson
Newton
Pauling

Check the element in the middle. If it’s less than the key, we can discard it, and ***everything that that comes before it***.

Galton
Hooke
Laplace

Mendeleyev
Morley
Michelson
Newton
Pauling

Check the element in the middle. If it’s greater than the key, we can discard it, and ***everything that that comes <u>after</u> it***.

Galton

Hooke

Laplace

Repeat until we land on the key, or the array is empty.

Laplace

Repeat until we land on the key, or the array is empty.

Binary Search

Three checks!

Binary Search: Progressively consider half the list until the search key is found or there are no more elements to consider.

Requires a sorted array!

Binary Search: Progressively consider half the array until the search key is found or there are no more elements to consider.

Requires a sorted array!

Binary VS Linear

Things to ponder:

- In the ***best*** case, both linear and binary searches find the key on the first comparison.
- In the ***worst*** case, linear search checks every element
- How many comparisons does binary search make in the worst case?
- Our intuition would say it’s fewer than linear search
- Linear search reduces search space by one value
- Binary search cuts the search space ***in half***

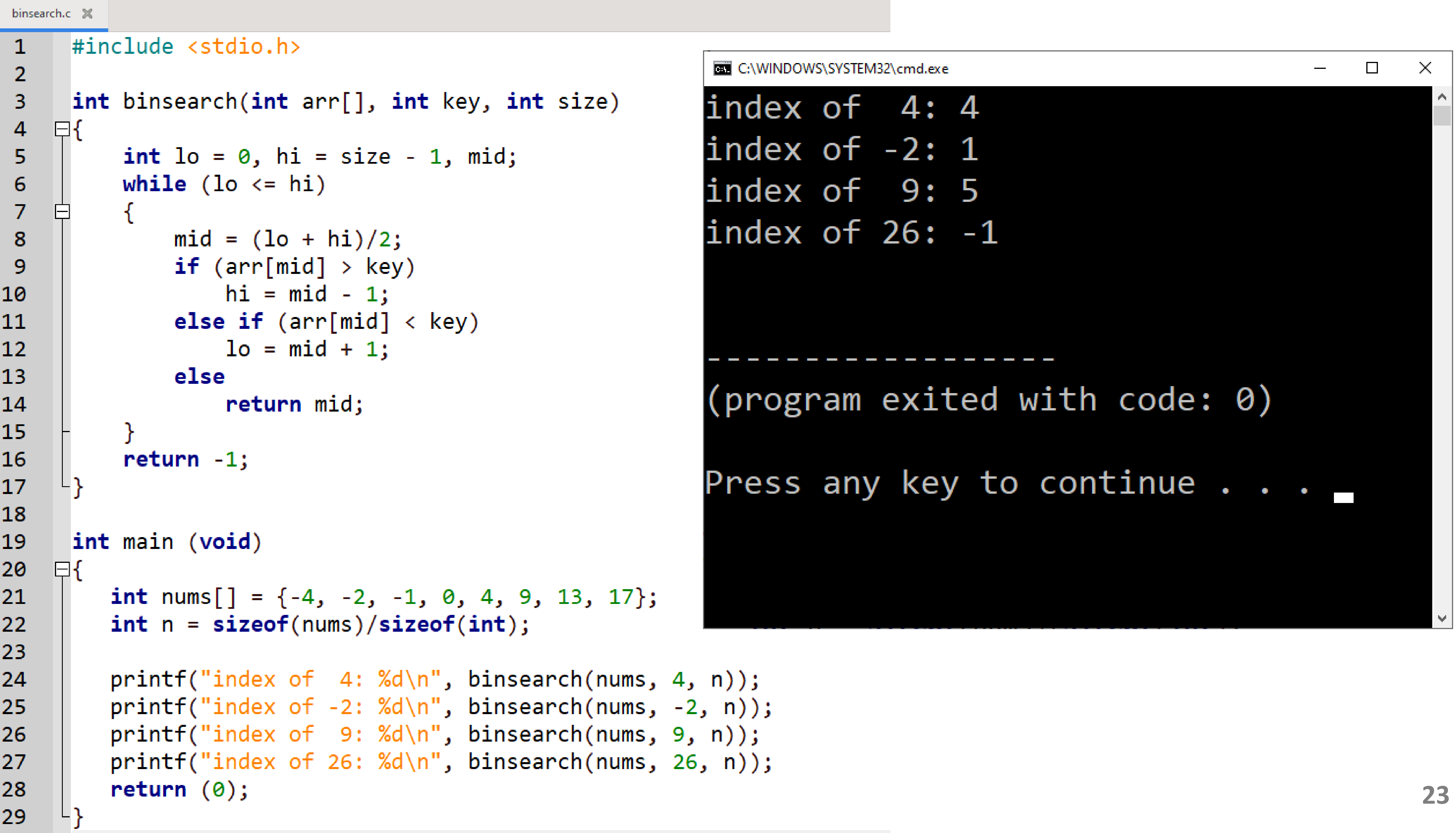
```
int binsearch(int arr[], int key, int size)
{
   int lo = 0, hi = size - 1, mid;
   while (lo <= hi)
   {
      mid = (lo + hi)/2;
      if (arr[mid] > key)
         hi = mid - 1;
      else if (arr[mid] < key)
         lo = mid + 1;
      else
         return mid;
   }
   return -1;
}
```

Compute **mid** index. Integer division ensures truncation to nearest whole number.

If the key is ***less than*** the element at the mid index, we move the high index down. Effectively discarding the upper half of the list.

If the key is ***greater than*** the element at the mid index, we move the low index up.

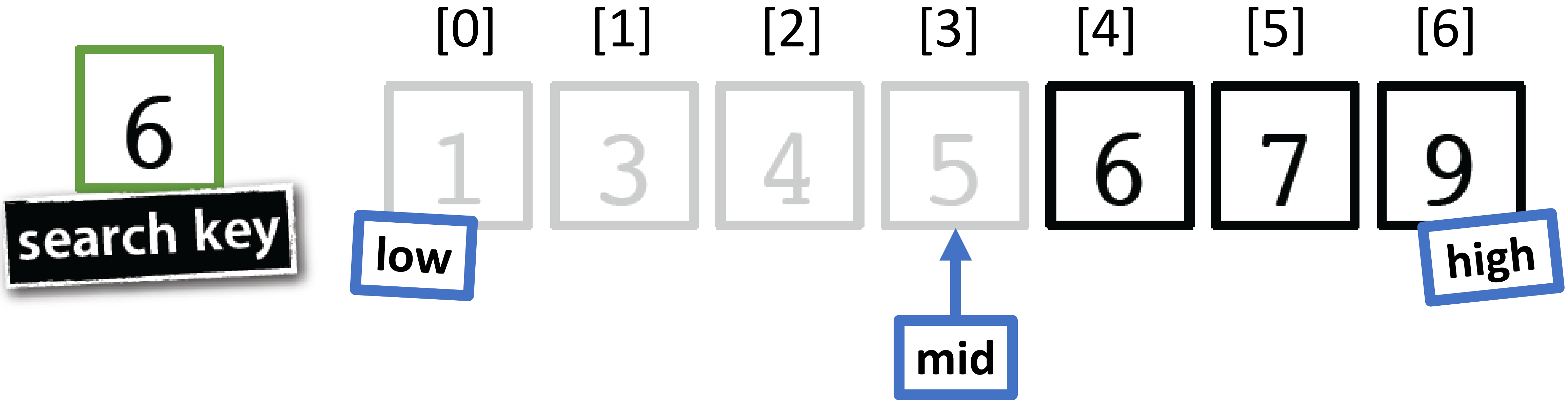
If neither is true, the key must equal the element at the mid index. Thus, we return **mid**.



```c
#include <stdio.h>

int binsearch(int arr[], int key, int size)
{
    int lo = 0, hi = size - 1, mid;
    while (lo <= hi)
    {
        mid = (lo + hi)/2;
        if (arr[mid] > key)
            hi = mid - 1;
        else if (arr[mid] < key)
            lo = mid + 1;
        else
            return mid;
    }
    return -1;
}

int main (void)
{
   int nums[] = {-4, -2, -1, 0, 4, 9, 13, 17};
   int n = sizeof(nums)/sizeof(int);

   printf("index of  4: %d\n", binsearch(nums, 4, n));
   printf("index of -2: %d\n", binsearch(nums, -2, n));
   printf("index of  9: %d\n", binsearch(nums, 9, n));
   printf("index of 26: %d\n", binsearch(nums, 26, n));
   return (0);
}
```

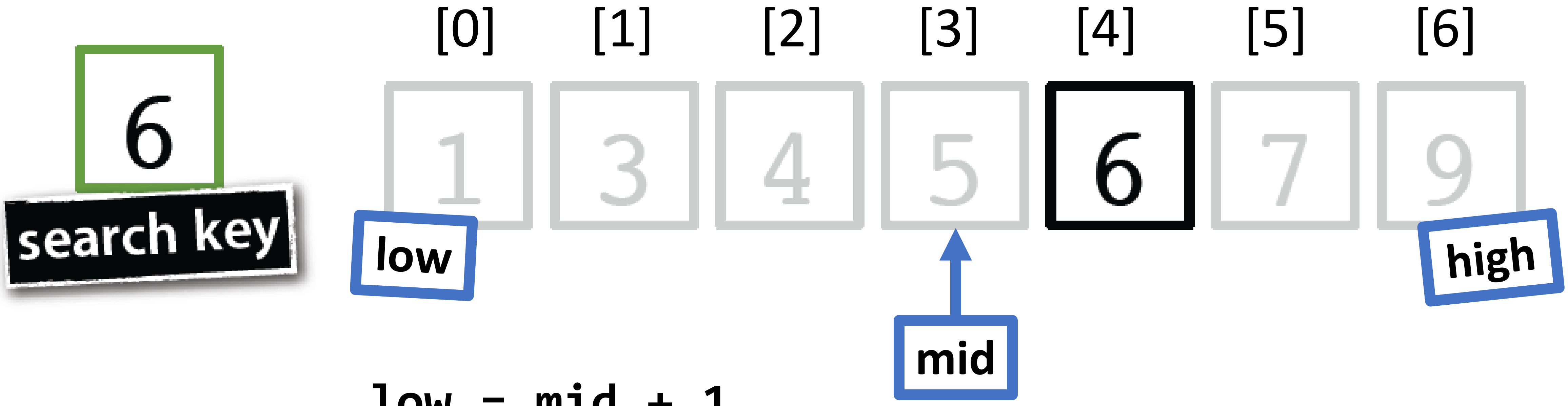
```
index of  4: 4
index of -2: 1
index of  9: 5
index of 26: -1


------------------
(program exited with code: 0)

Press any key to continue . . .
```

Binary Search



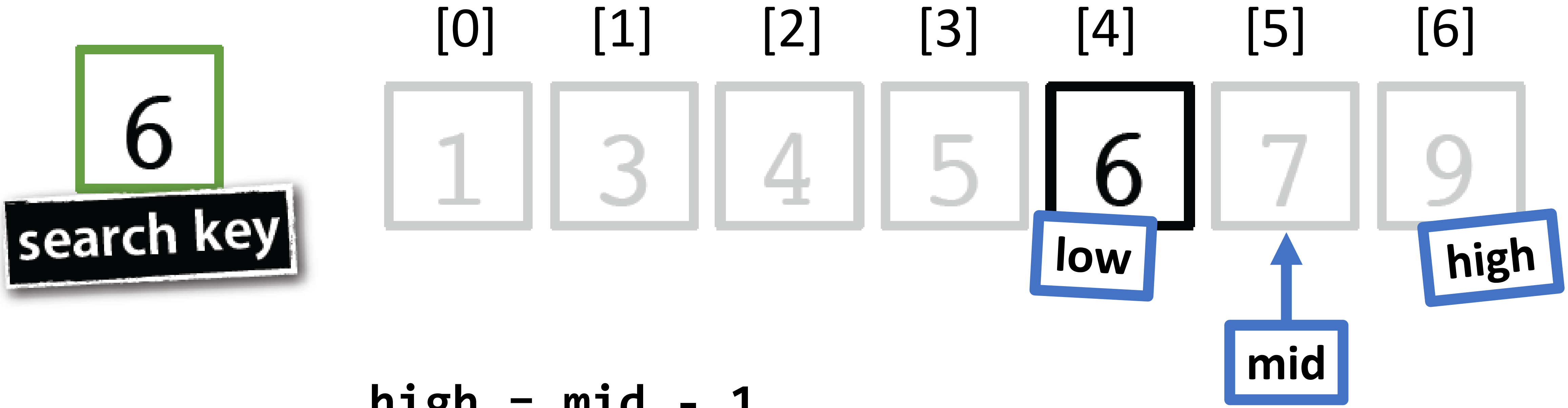
`mid = (0 + 6) // 2`

Binary Search



```
low = mid + 1
mid = (4 + 6) // 2
```

Binary Search



6

search key

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

low

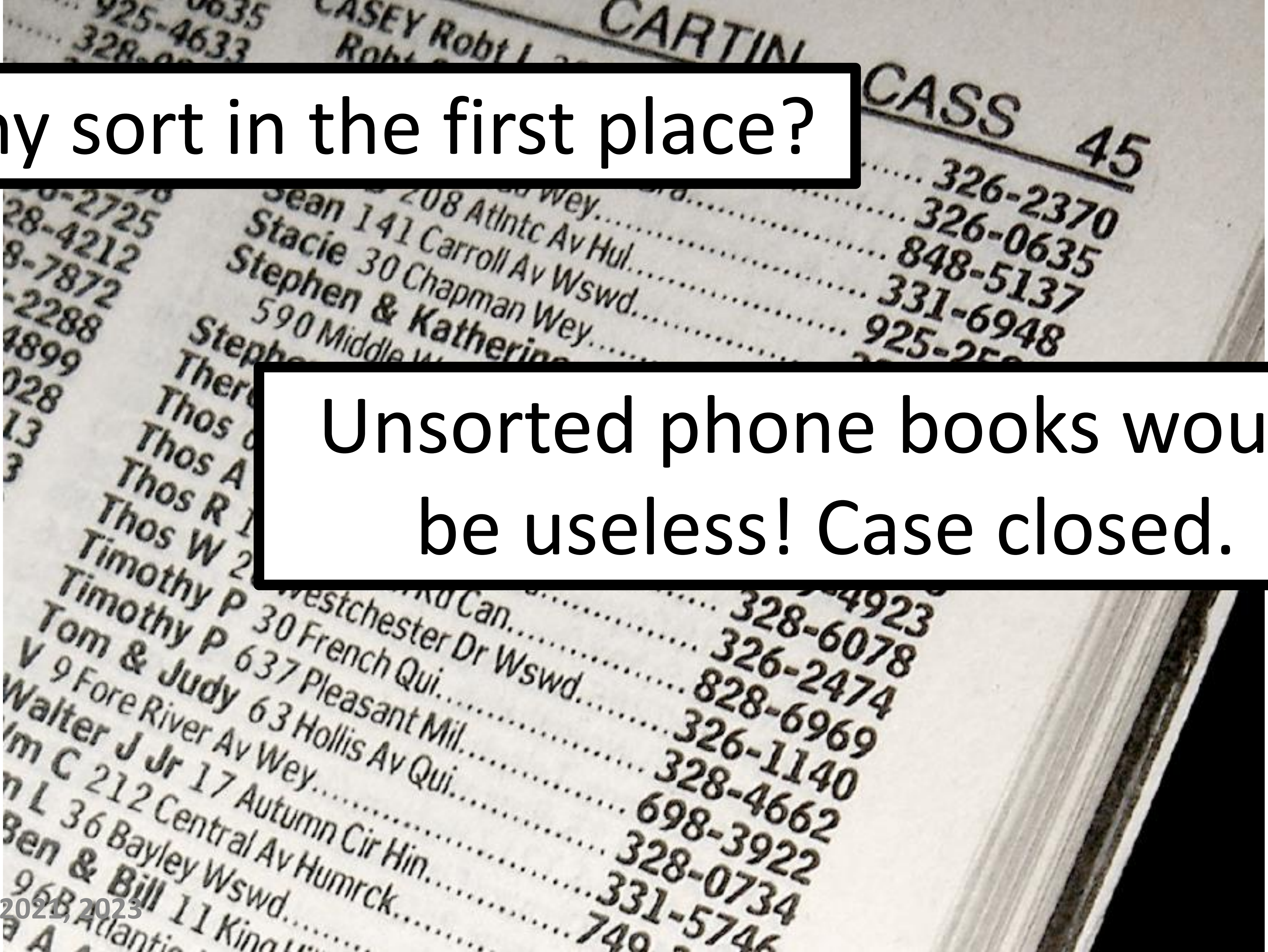
mid

high

```
high = mid - 1
mid = (4 + 4) // 2
```

```
key == list[mid]
```

Sorting



Why sort in the first place?

Unsorted phone books would be useless! Case closed.

Sorting is something we all do:

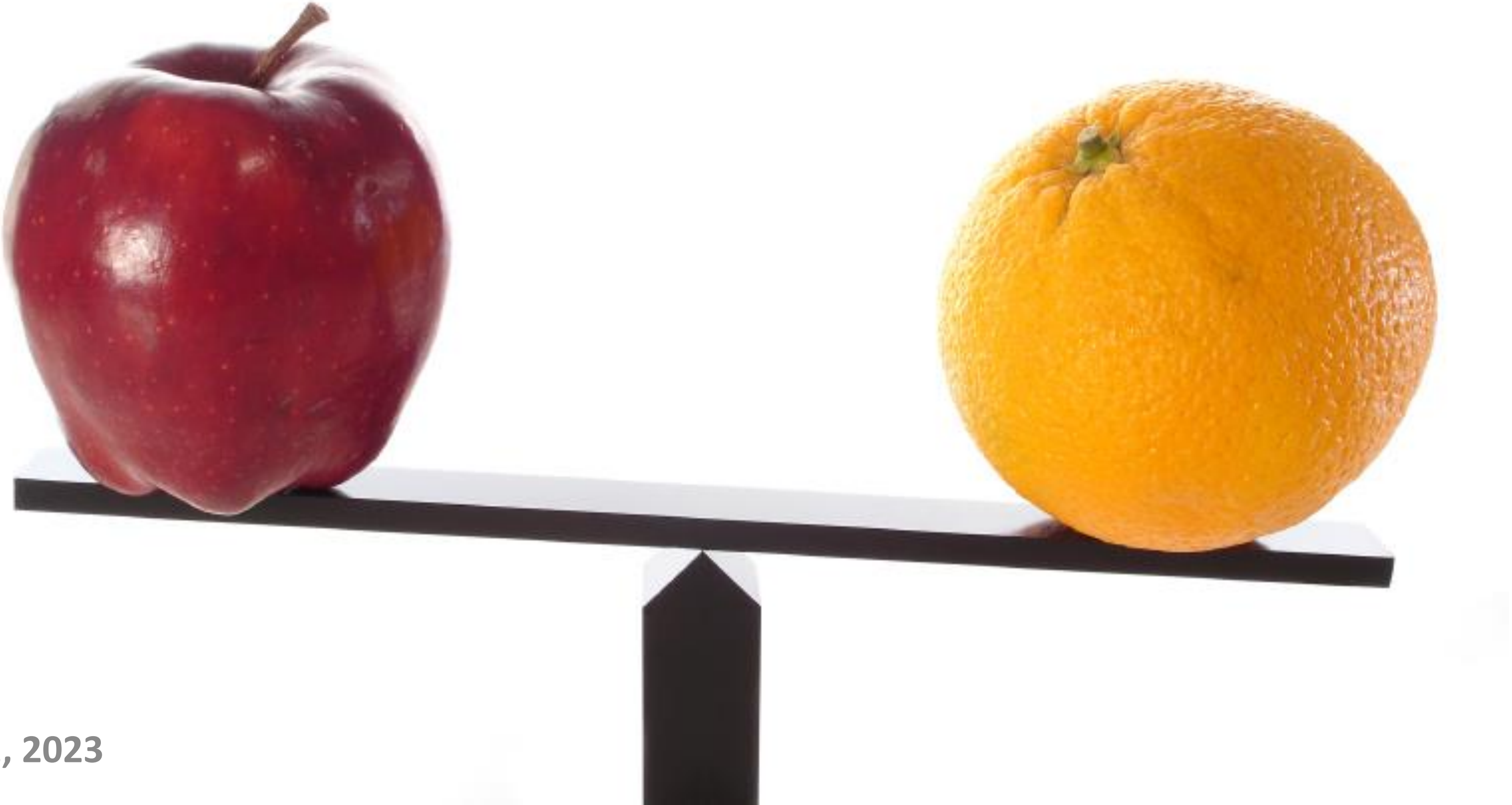
This person is clearly a monster. Not sorted alphabetically ***OR*** by point value.

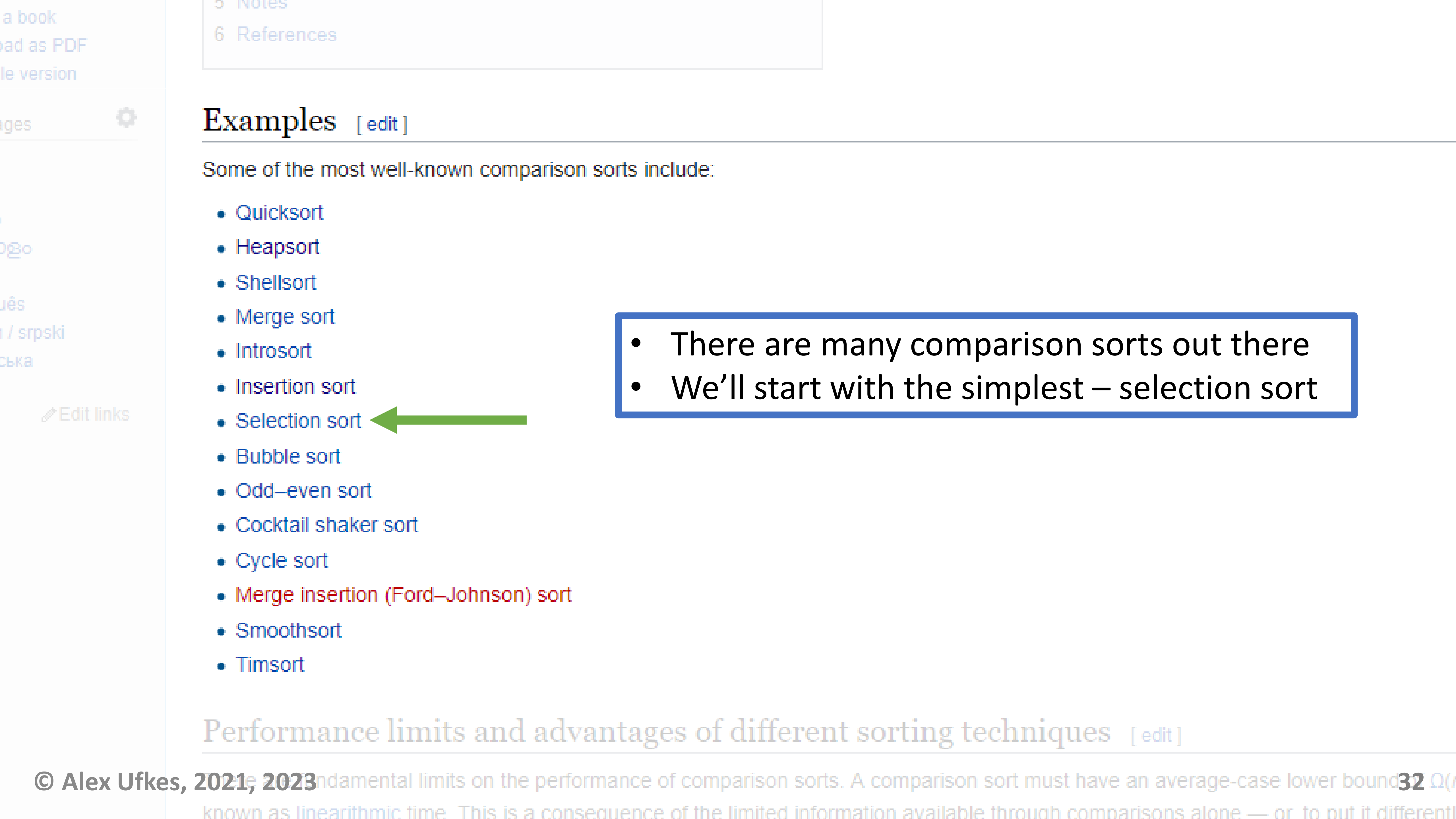
Sorting forms the foundation for faster algorithms

Very often:

Working on an *unsorted* array is **<u>more</u>** expensive than sorting the array **<u>first</u>** and **THEN** working on it.

Comparison Sorting





6 References

Examples [edit]

Some of the most well-known comparison sorts include:

- Quicksort
- Heapsort
- Shellsort
- Merge sort
- Introsort
- Insertion sort
- Selection sort
- Bubble sort
- Odd–even sort
- Cocktail shaker sort
- Cycle sort
- Merge insertion (Ford–Johnson) sort
- Smoothsort
- Timsort

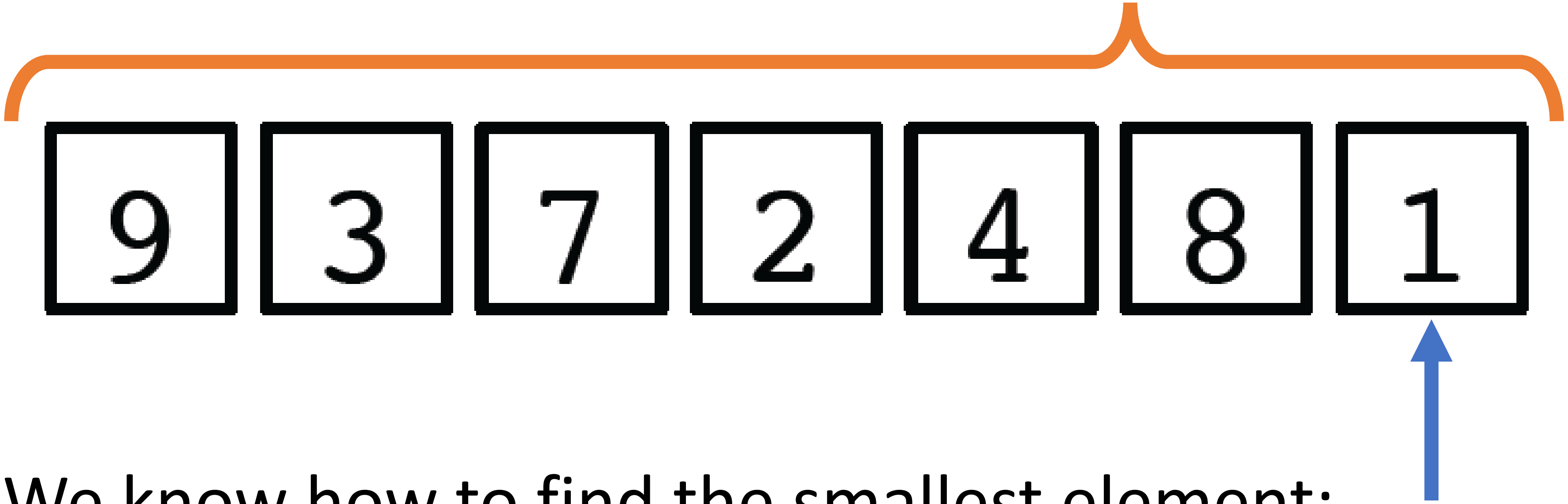
Performance limits and advantages of different sorting techniques [edit]

There are fundamental limits on the performance of comparison sorts. A comparison sort must have an average-case lower bound

known as linearithmic time. This is a consequence of the limited information available through comparisons alone — or, to put it different

- There are many comparison sorts out there
- We’ll start with the simplest – selection sort

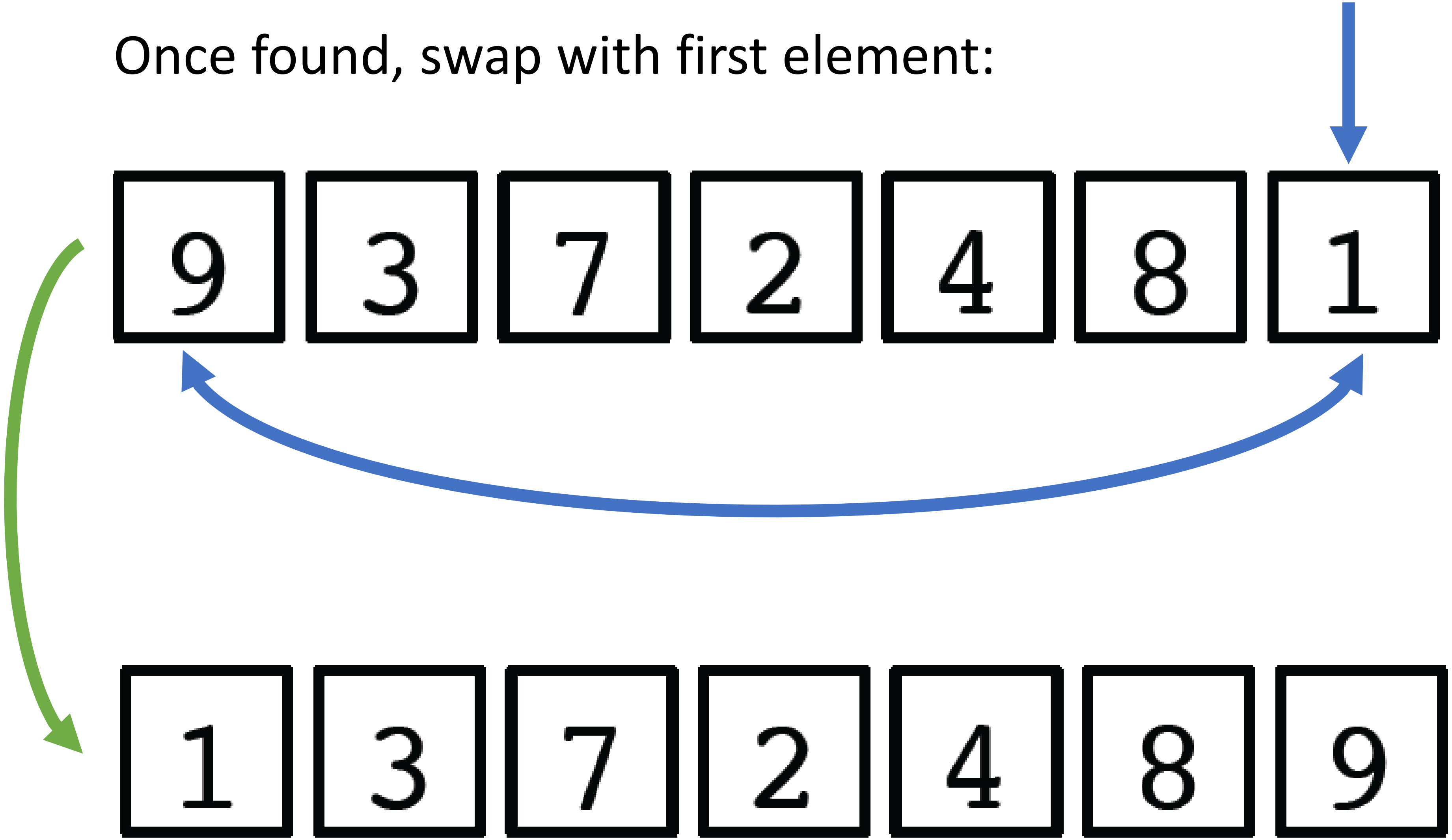
Selection Sort: Repeatedly find smallest element and move it to the front of the *unsorted* region



We know how to find the smallest element:

We did it earlier!

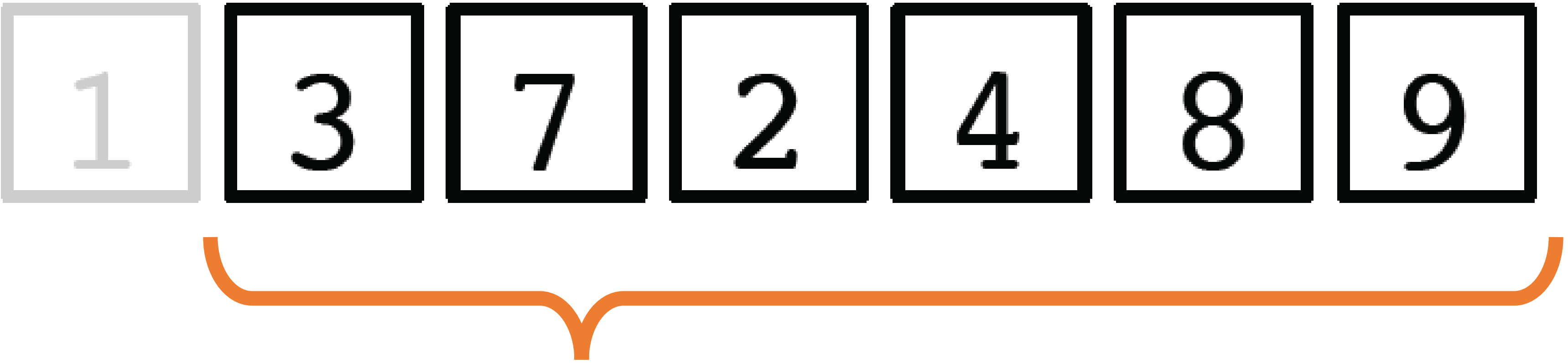
Once found, swap with first element:



9	3	7	2	4	8	1

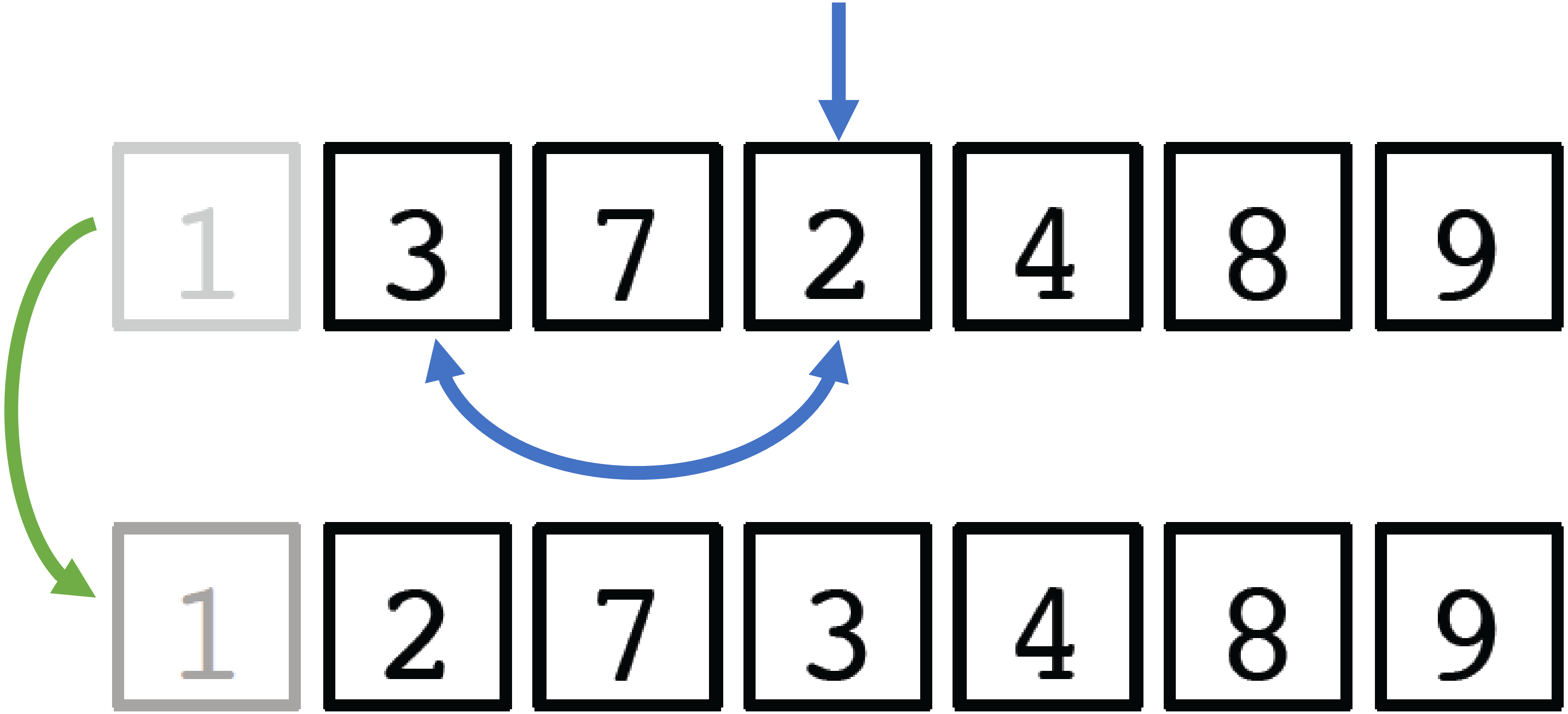
1	3	7	2	4	8	9

We can now be **certain** that the first element is in the correct location:

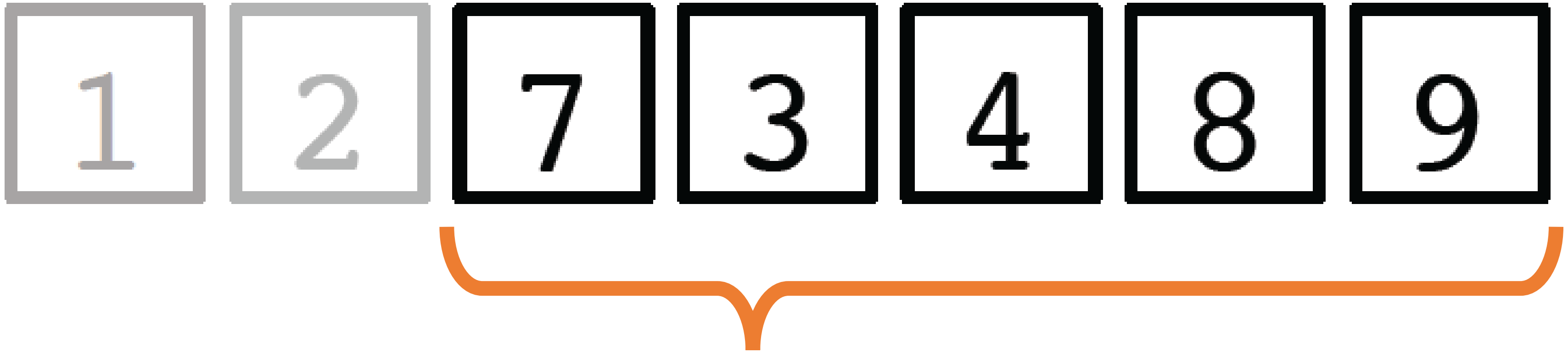


New unsorted region!

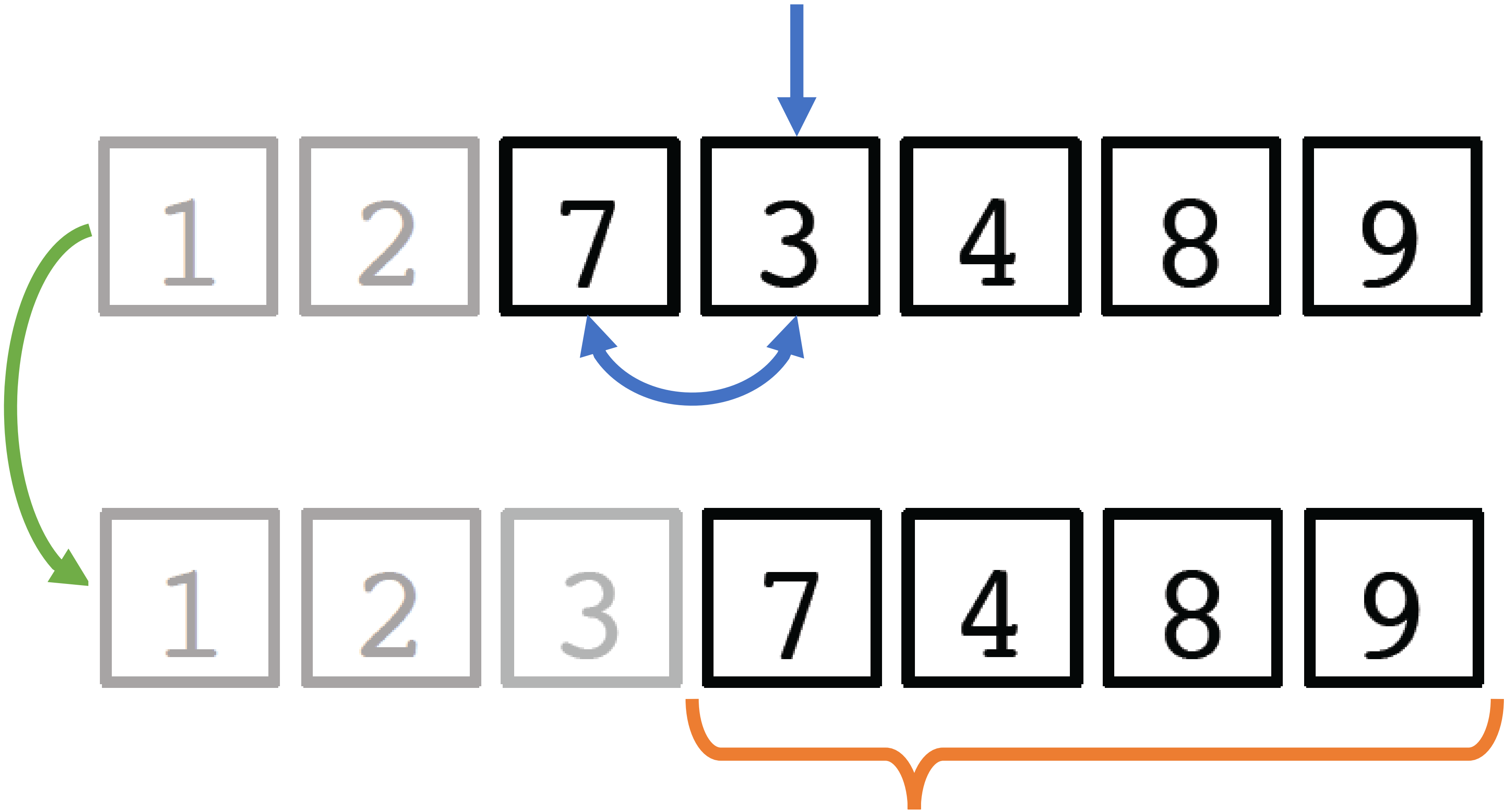
Once more, find the smallest element in the *unsorted* region, swap with element at the front of *unsorted* region:



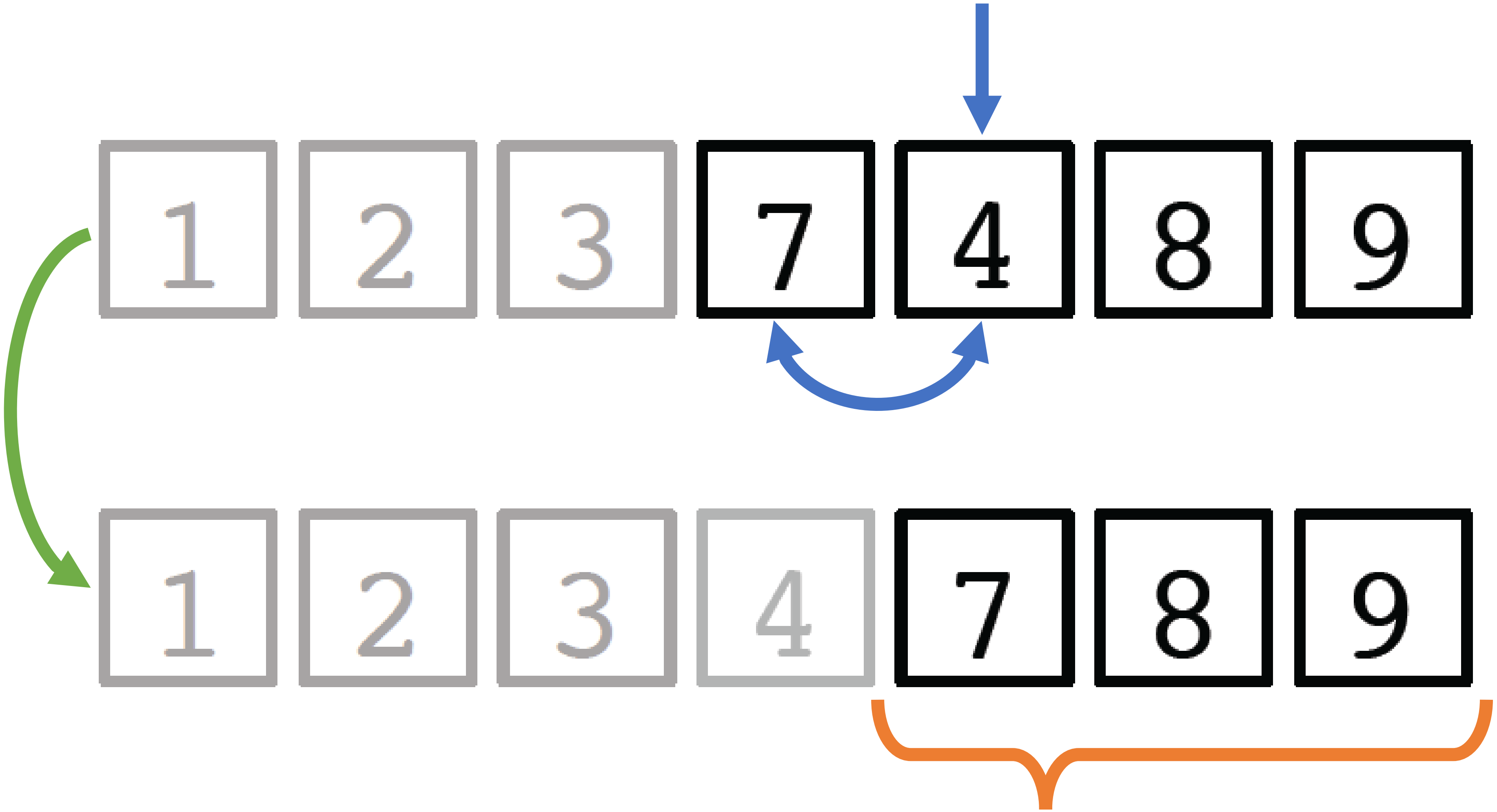
We can now be **certain** that the first TWO element are in the correct location:



New unsorted region!



New unsorted region!

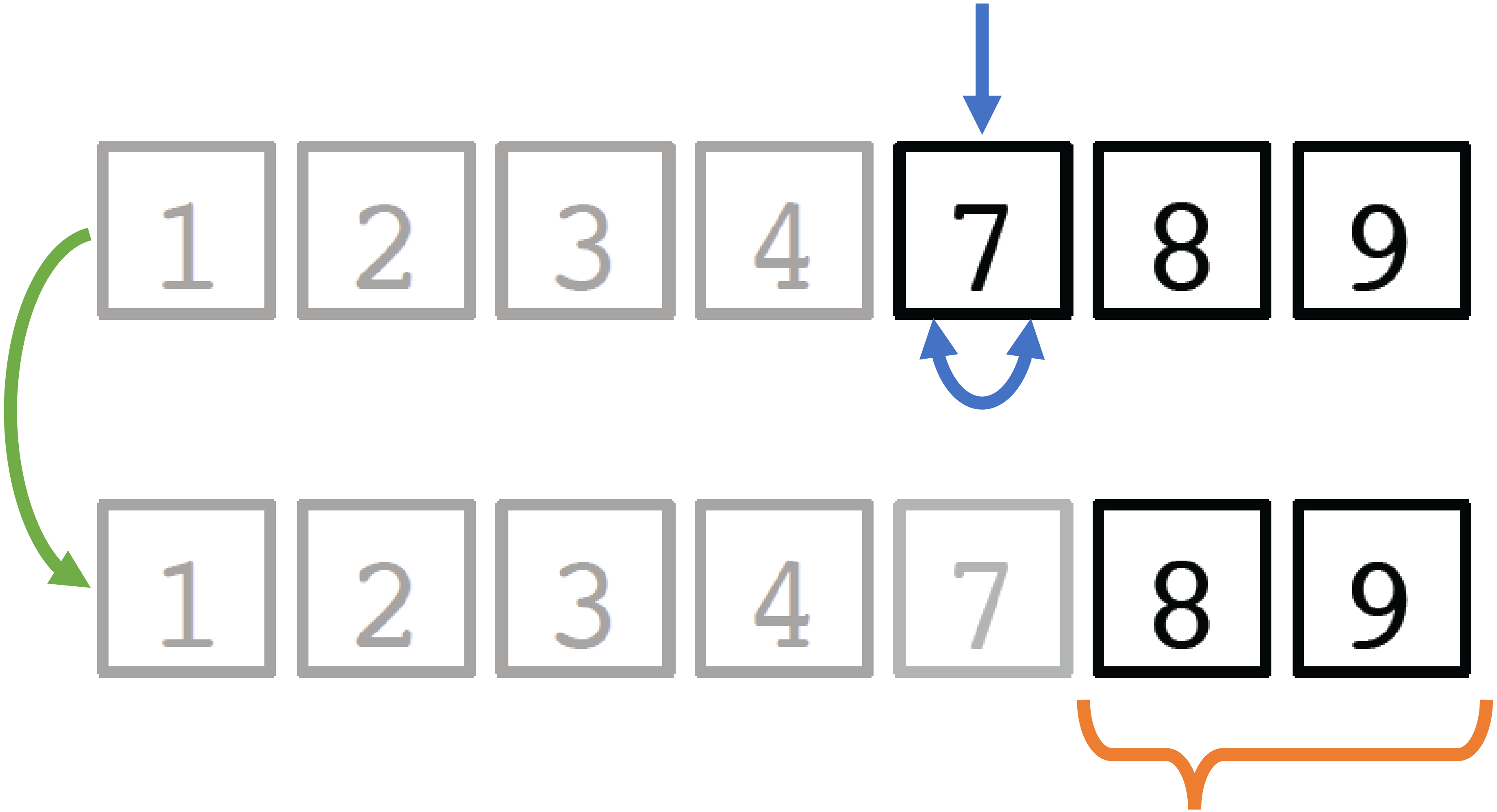


New unsorted region!

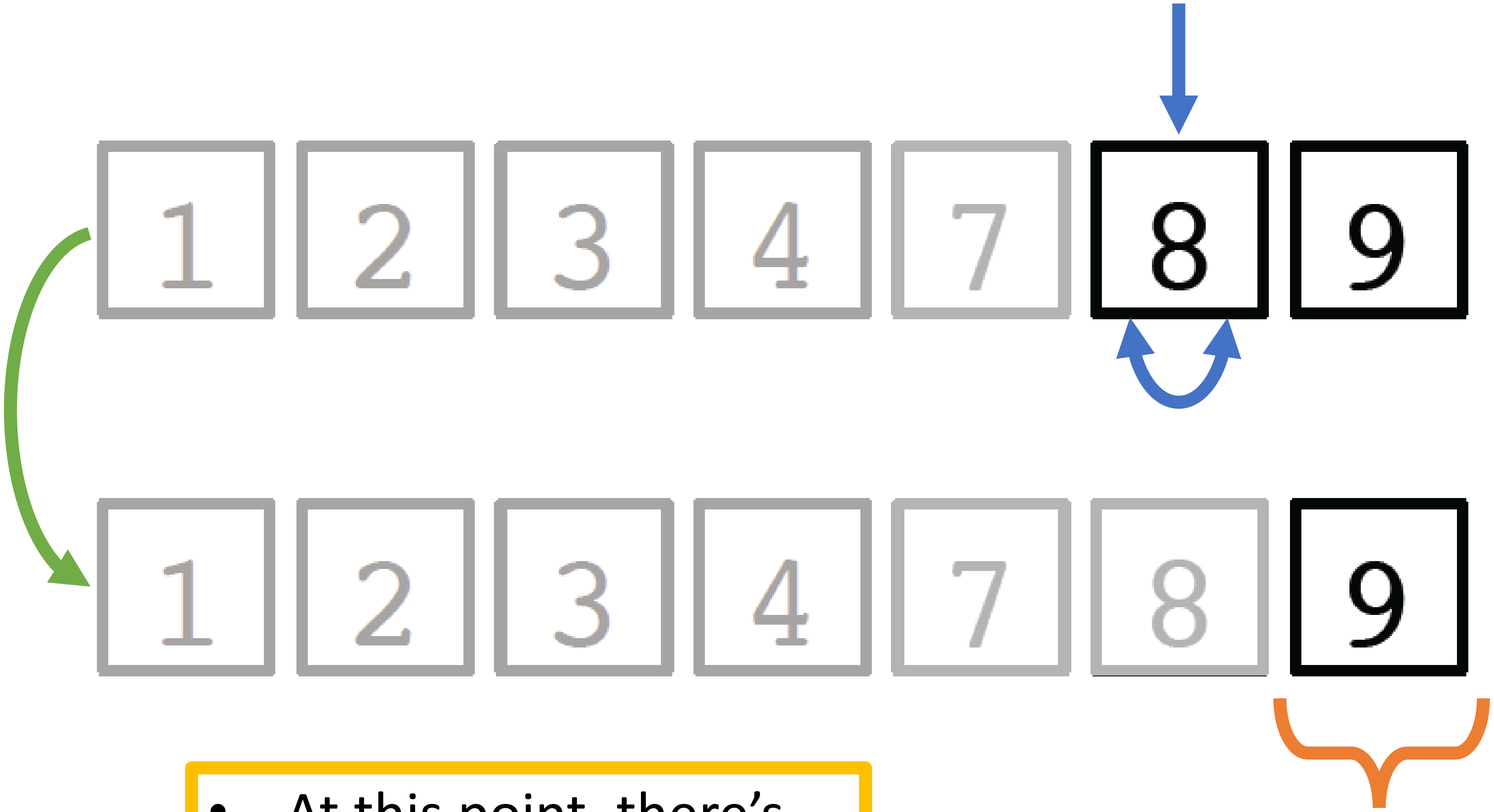
1	2	3	4	7	8	9

The array is sorted at this point, but the algorithm ***<u>doesn’t know that</u>***.

We continue this process until the entire array has been compared and swapped in this manner.



New unsorted region!



1	2	3	4	7	8	9
1	2	3	4	7	8	9

- At this point, there’s one element left.
- We know we’re done.

New unsorted region!

```
void sel_sort(int arr[], int size)
{
   for (int i = 0; i < size-1; i++)
   {
      int min_idx = i;
      for (int j = i + 1; j < size; j++)
      {
         if (arr[j] < arr[min_idx])
            min_idx = j;
      }
      swap(&arr[i], &arr[min_idx]);
   }
}
```

Iterate through every element except the last

Variable to store index of smallest element

Find the index of smallest element. Note relationship between `i` and `j`!

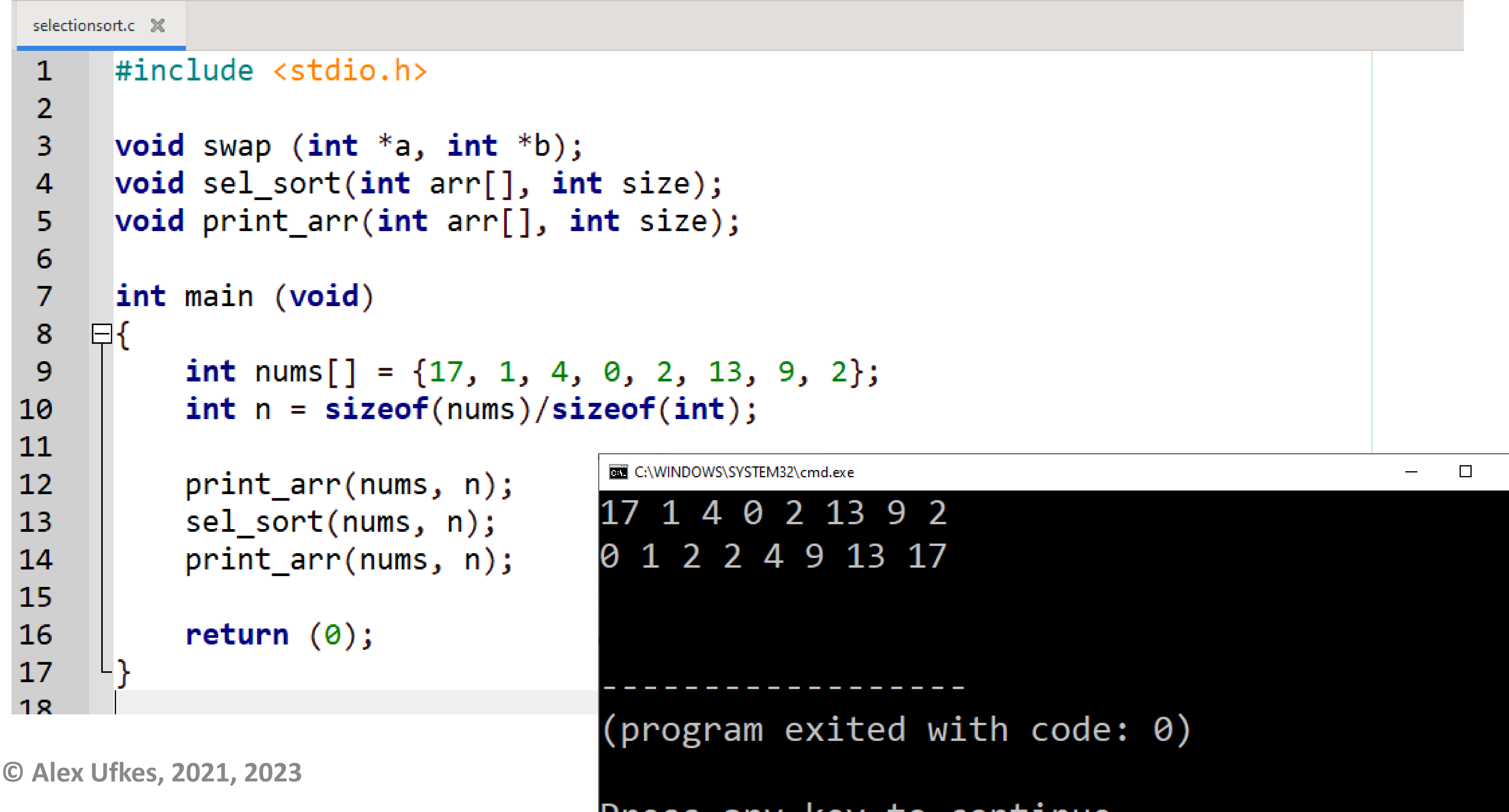
Once found, swap smallest element with front of *unsorted* region

```
void sel_sort(int arr[], int size)
{
   for (int i = 0; i < size-1; i++)
   {
      int min_idx = i;
      for (int j = i + 1; j < size; j++)
      {
         if (arr[j] < arr[min_idx])
            min_idx = j;
      }
      swap(&arr[i], &arr[min_idx]);
   }
}
```

```
void swap (int *a, int *b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
void swap (int *a, int *b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}

void sel_sort(int arr[], int size)
{
   for (int i = 0; i < size-1; i++)
   {
      int min_idx = i;
      for (int j = i + 1; j < size; j++)
      {
         if (arr[j] < arr[min_idx])
            min_idx = j;
      }
      swap(&arr[i], &arr[min_idx]);
   }
}
```

```
void print_arr(int arr[], int size)
{
   for (int i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main (void)
{
   int nums[] = {17,1,4,0,2,13,9,2};
   int n = sizeof(nums)/sizeof(int);

   print_arr(nums, n);
   sel_sort(nums, n);
   print_arr(nums, n);

   return (0);
}
```



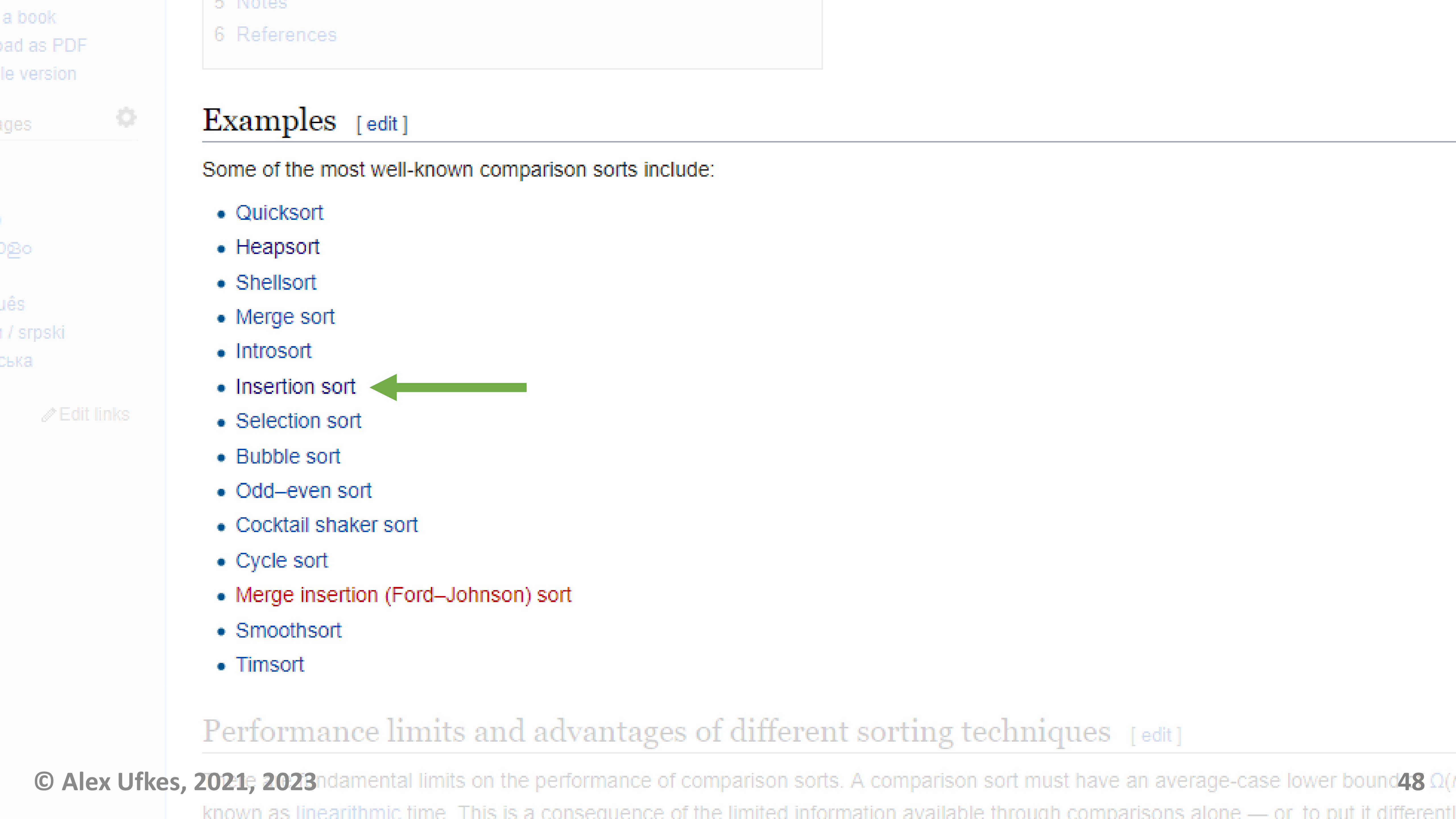
```c
#include <stdio.h>

void swap (int *a, int *b);
void sel_sort(int arr[], int size);
void print_arr(int arr[], int size);

int main (void)
{
    int nums[] = {17, 1, 4, 0, 2, 13, 9, 2};
    int n = sizeof(nums)/sizeof(int);

    print_arr(nums, n);
    sel_sort(nums, n);
    print_arr(nums, n);

    return (0);
}
```

```
17 1 4 0 2 13 9 2
0 1 2 2 4 9 13 17



------------------
(program exited with code: 0)

Press any key to continue
```

Selection Sort

It’s ***BAD***

- It’s good as *“my first sorting algorithm”*
- Bad for sorting in an efficient manner
- Performance is identical in best-case and worst-case scenarios.
- Even if the list is ***already sorted***, selection sort takes just as long to perform.



5 Notes
6 References

Examples [edit]

Some of the most well-known comparison sorts include:

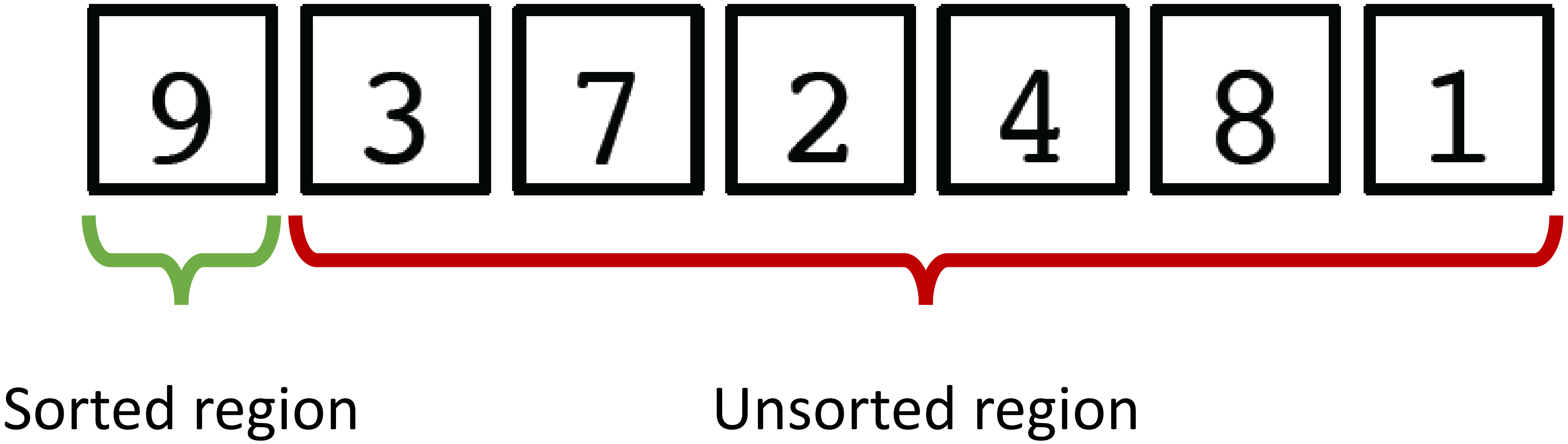
- Quicksort
- Heapsort
- Shellsort
- Merge sort
- Introsort
- Insertion sort ←
- Selection sort
- Bubble sort
- Odd–even sort
- Cocktail shaker sort
- Cycle sort
- Merge insertion (Ford–Johnson) sort
- Smoothsort
- Timsort

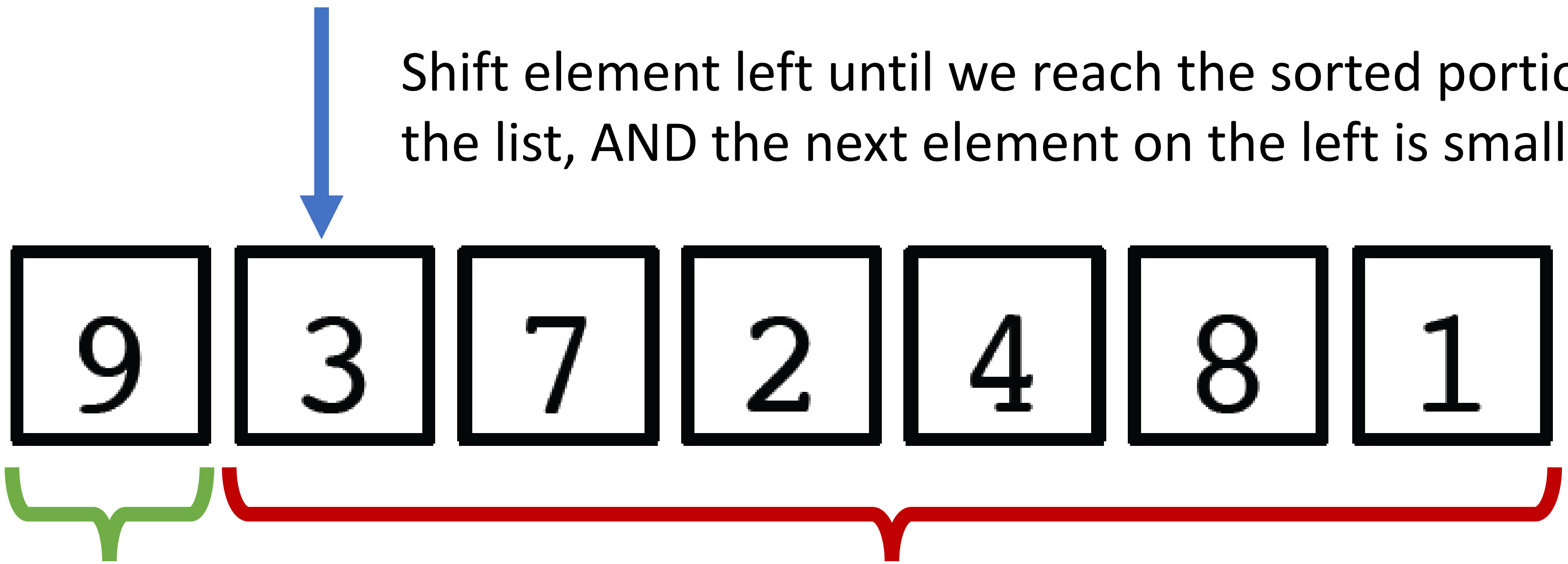
Performance limits and advantages of different sorting techniques [edit]

There are fundamental limits on the performance of comparison sorts. A comparison sort must have an average-case lower bound Ω(

known as linearithmic time. This is a consequence of the limited information available through comparisons alone — or, to put it different

Insertion Sort: Every iteration removes next element from the unsorted region and inserts it into the correct position within the sorted region.



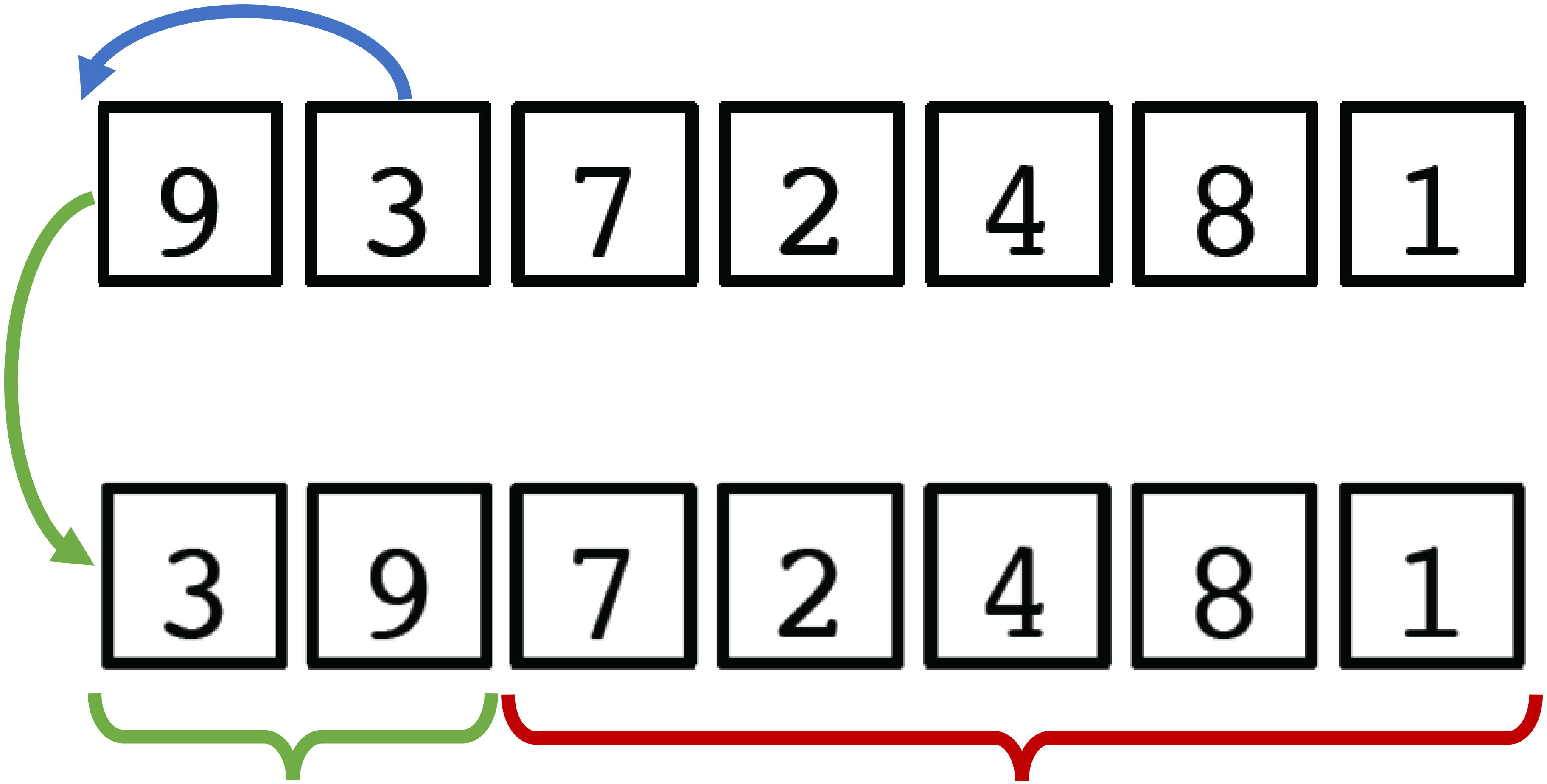


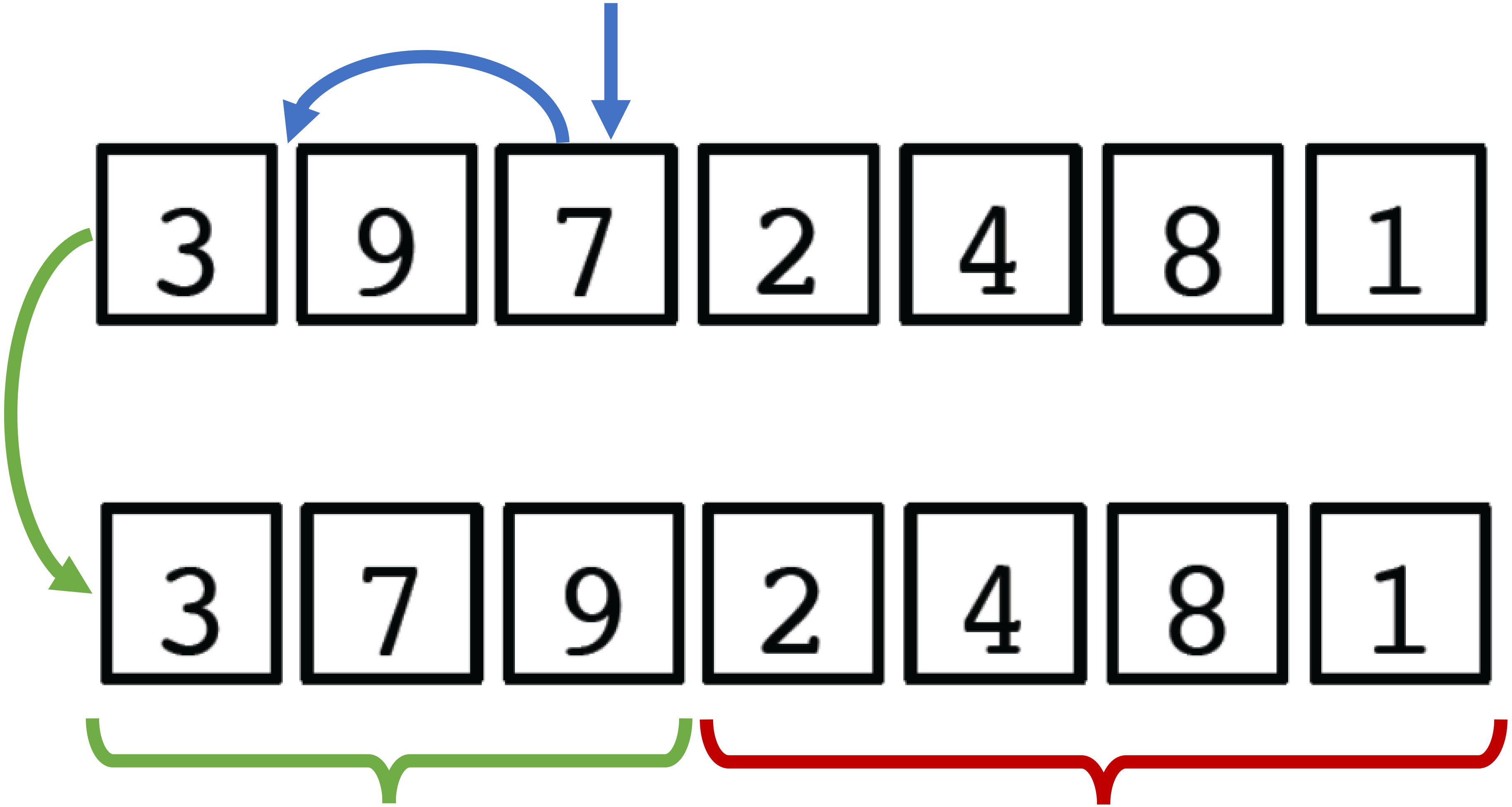
Shift element left until we reach the sorted portion of the list, AND the next element on the left is smaller.

Sorted region

Unsorted region

© Alex Ufkes, 2021, 2023

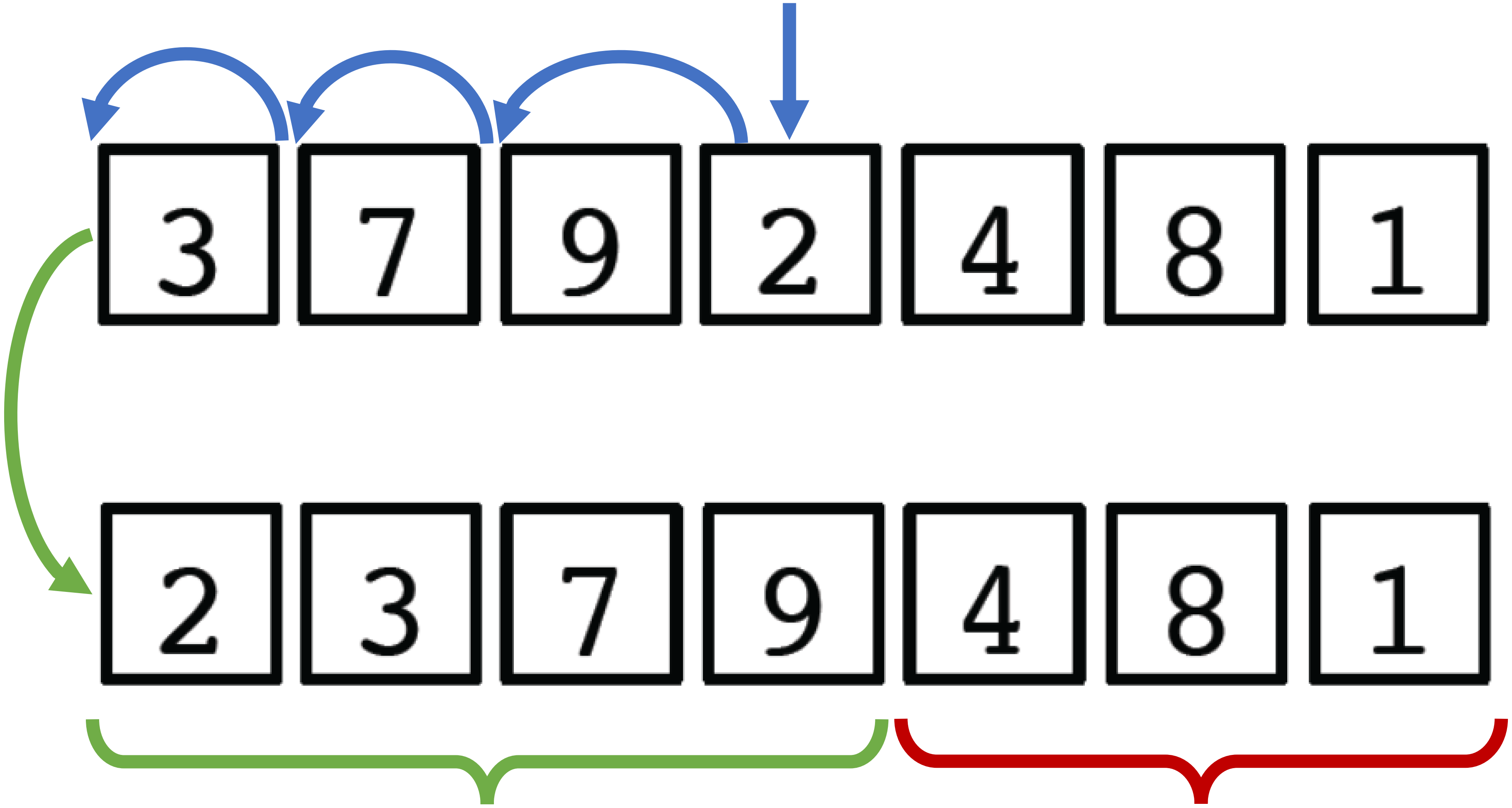




3	9	7	2	4	8	1
3	7	9	2	4	8	1

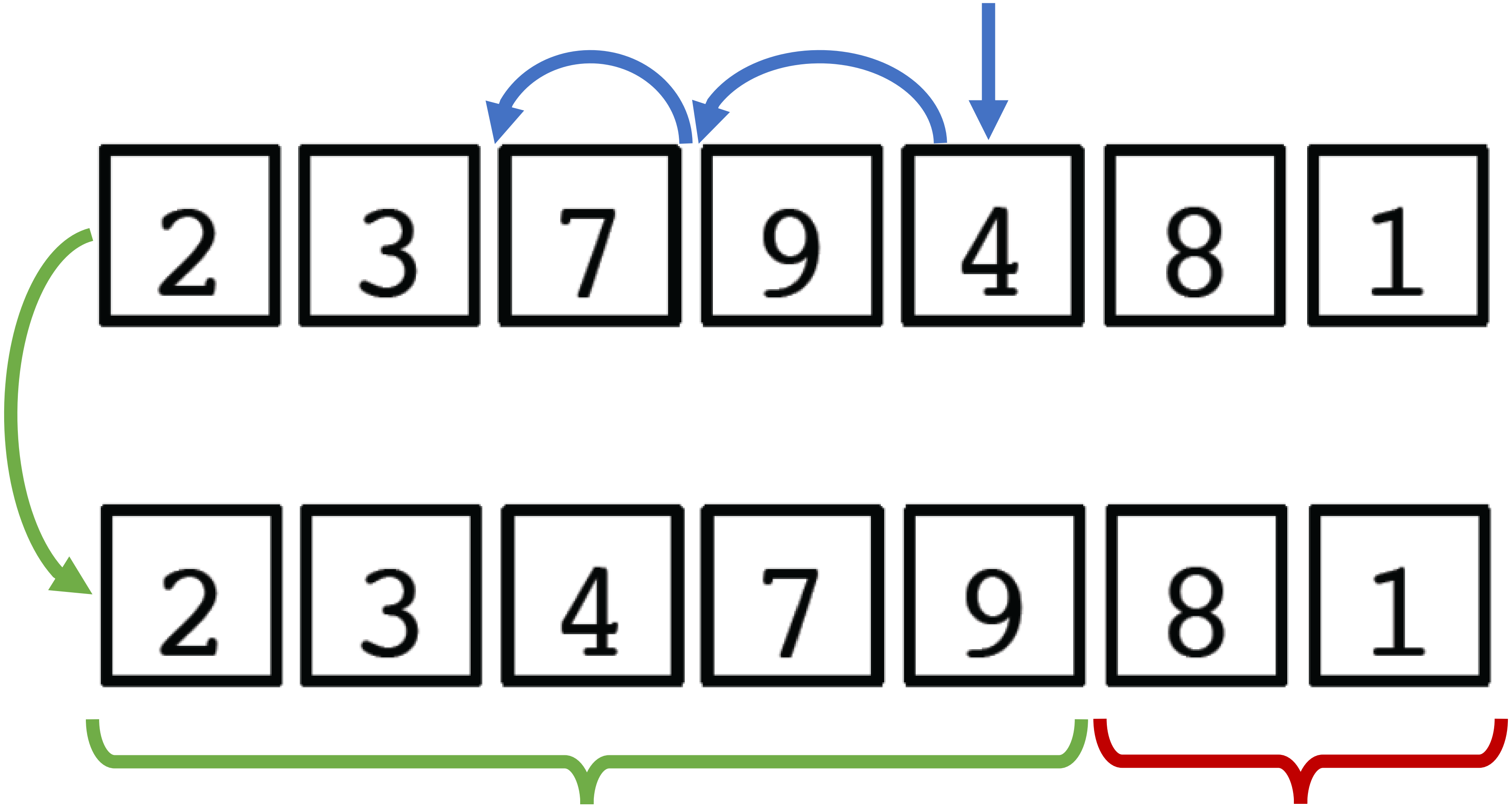
Sorted region

Unsorted region



Sorted region

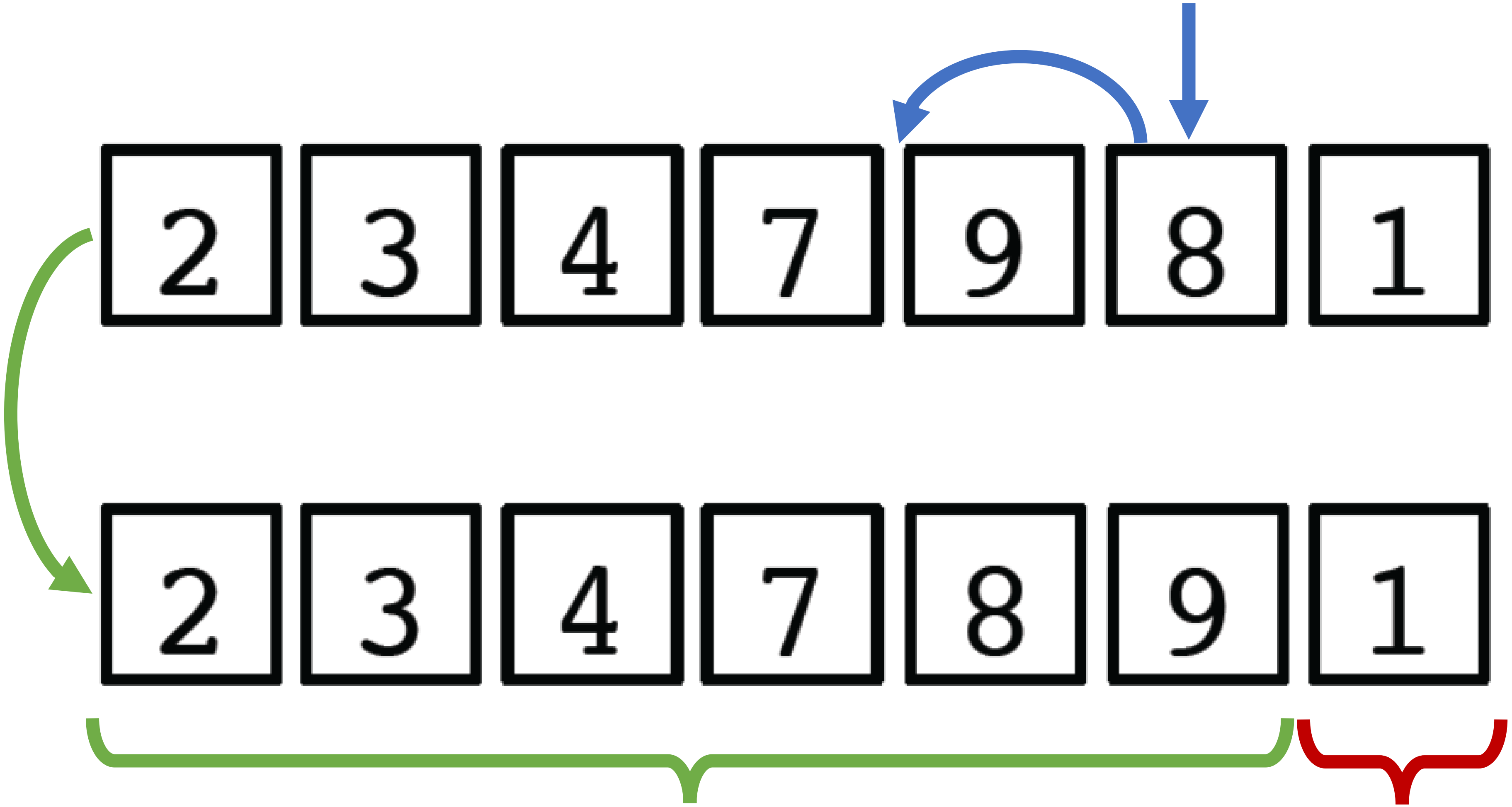
Unsorted region



2	3	7	9	4	8	1
2	3	4	7	9	8	1

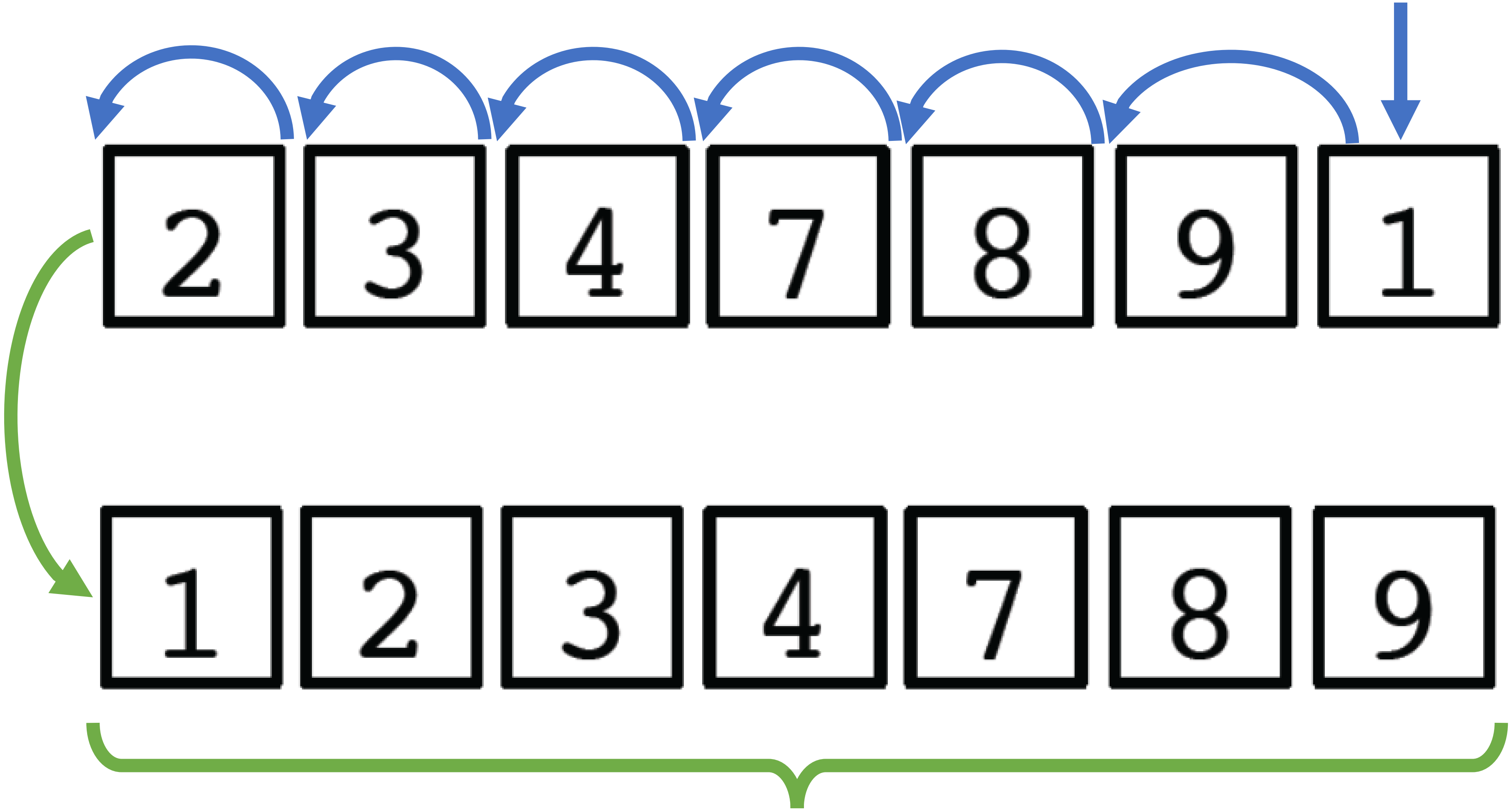
Sorted region

Unsorted region



Sorted region

Unsorted region



Sorted region

Done!

```
void ins_sort(int arr[], int size)
{
   for (int i = 1; i < size; i++)
   {
      int curr = arr[i], cid = i;
      while (cid > 0 && arr[cid-1] > curr)
      {
         arr[cid] = arr[cid-1];
         cid -= 1;
      }
      arr[cid] = curr;
   }
}
```

Iterate through every element except the *first*

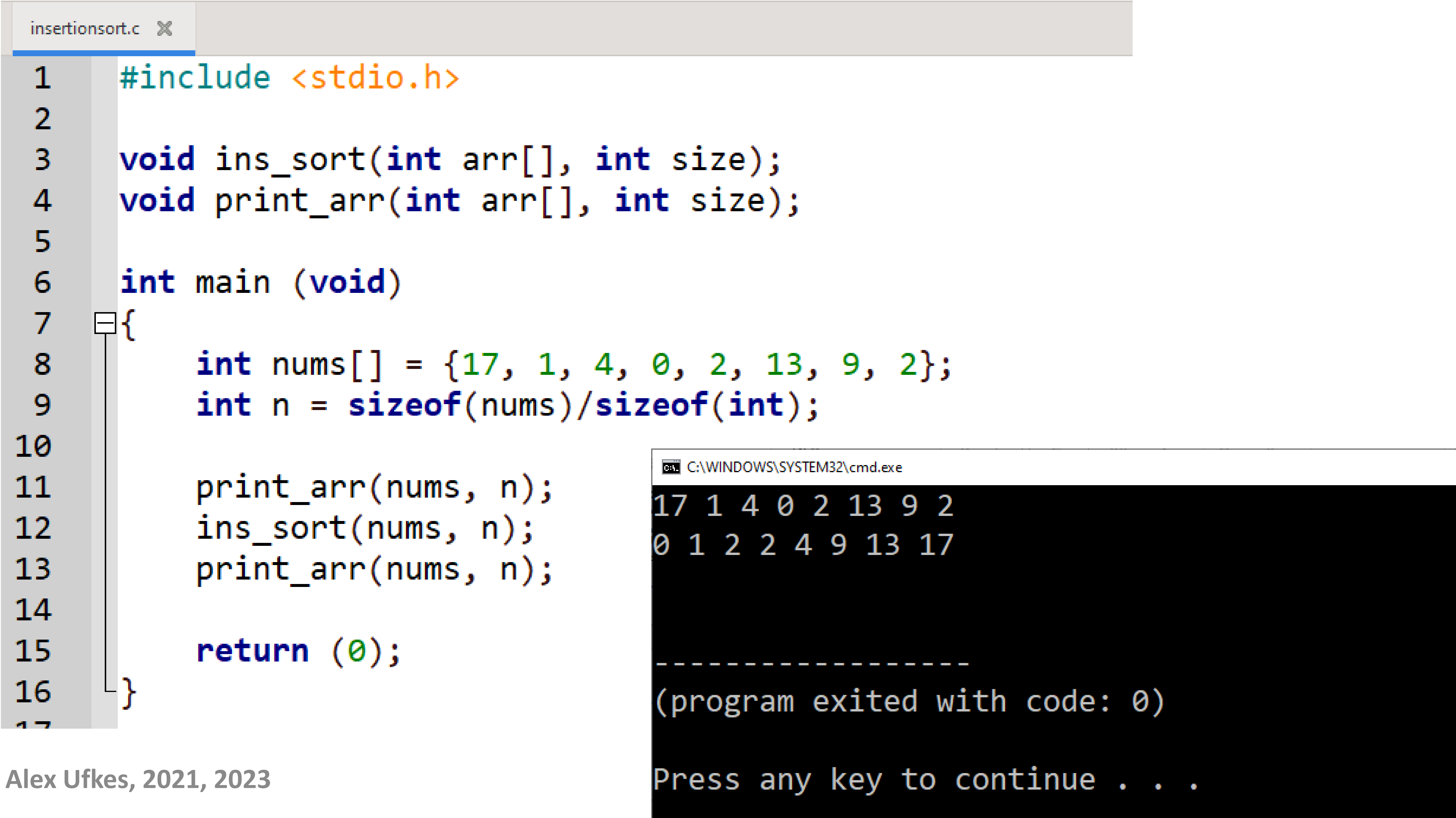
Variables to store current element and current index

Shift elements over until we hit the front OR an element smaller than **curr**

Once all larger elements are shifted, insert **curr** at index **cid**

```
void ins_sort(int arr[], int size)
{
  for (int i = 1; i < size; i++)
  {
    int curr = arr[i], cid = i;
    while (cid > 0 && arr[cid-1] > curr)
    {
      arr[cid] = arr[cid-1];
      cid -= 1;
    }
    arr[cid] = curr;
  }
}
```

```
void print_arr(int arr[], int size)
{
   for (int i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main (void)
{
   int nums[] = {17,1,4,0,2,13,9,2};
   int n = sizeof(nums)/sizeof(int);

   print_arr(nums, n);
   ins_sort(nums, n);
   print_arr(nums, n);

   return (0);
}
```



```c
#include <stdio.h>

void ins_sort(int arr[], int size);
void print_arr(int arr[], int size);

int main (void)
{
    int nums[] = {17, 1, 4, 0, 2, 13, 9, 2};
    int n = sizeof(nums)/sizeof(int);

    print_arr(nums, n);
    ins_sort(nums, n);
    print_arr(nums, n);

    return (0);
}
```

```
17 1 4 0 2 13 9 2
0 1 2 2 4 9 13 17



------------------
(program exited with code: 0)

Press any key to continue . . .
```

Selection VS Insertion

Insertion sort is far more powerful:

- It shifts elements only as far as they need to move.
- If the list is already sorted, no shifting is required!
- The efficiency of insertion sort depends on the initial *sorted-ness* of the list.
- In the worst case? It's just as bad as selection sort.
- In the best case? It's much, ***much*** better!
- Selection sort is ***always*** bad.

Questions?