COMP 1020 - A01, D01 Searching and sorting algorithms

FALL 2020

What’s coming up in our last 2 weeks

- This week:
  - Searching and sorting algorithms
- Next week:
  - finishing up algorithms, review for the final exam

What is searching?

- Searching: Looking through an array/ArrayList/linked list/any list for a particular item (the “key” value)

- There are two fundamental algorithms:

  - The linear search (we’ve done this many times)

  - The binary search

Linear search

- The linear search
  - Searches from beginning to end
  - It has to look at each item one after the other until the key is found
  - Does not require the list to be sorted

Binary search

- The binary search
  - Divides the list in half repeatedly
  - Fast
  - Requires the list to be sorted
  - Requires fast random access to the list

Binary search

- The binary search
  - Divides the list in half repeatedly
  - Fast
  - Requires the list to be sorted
  - Requires fast random access to the list

The file SearchSort.java contains implementations of all the algorithms we will see this week.

Linear search: code

- A basic linear search:

```
int linearSearch(int[] list, int key){
    /* Search for key within the list. If found,
    * return its position (index), if not
    * return -1. */
    for(int index=0; index < list.length; index++)
        if(list[index] == key)
            return index;
    return -1;
}//linearSearch
```

Linear search analysis

- A linear search takes linear time to run
  - i.e. it will do a number of operations that is linear in comparison with the size of the input array/list
    - because it always goes through all the positions in the array/list until it finds a match

Binary search

- Imagine searching, by hand, a list of 30,000 U of M student names for “Zach Williams” using a linear search!
  - “Aaron Adams”? No.
  - “Aivee Albert”? No.
  - ……

Binary search

- If the list is sorted there’s a much faster way: a binary search
    - This is actually what you do naturally when you search in a dictionary (a real dictionary, you know, with pages)!

Binary search

- If the list is sorted there's a much faster way: a binary search
  - This is actually what you do naturally when you search in a dictionary (a real dictionary, you know, with pages)!
- Basic idea: At every point in the search, keep track of the section of the array, from list[lo] to list[hi], where the key might be
- Initially lo=0, hi=size-1 (the whole array)

Binary search

- One step of a binary search:
    - Trying to find key in the portion of the list from list[lo] to list[hi]

Binary search

- One step of a binary search:
  - Trying to find key in the portion of the list from list[lo] to list[hi]
  - Pick the middle element in that range
    - list[mid] where mid = (lo+hi)/2

Binary search

- One step of a binary search:
  - Trying to find key in the portion of the list from list[lo] to list[hi]
  - Pick the middle element in that range
    - list[mid] where mid = (lo+hi)/2
  - If list[mid]==key, you're done, of course

Binary search

- One step of a binary search:
  - Trying to find key in the portion of the list from list[lo] to list[hi]
  - Pick the middle element in that range
    - list[mid] where mid = (lo+hi)/2
  - If list[mid]==key, you're done, of course
  - If list[mid]>key, then
    - Key can only be from list[lo] to list[mid-1]
      - Everything above list[mid] must be too big, too
    - Change hi = mid-1

Binary search

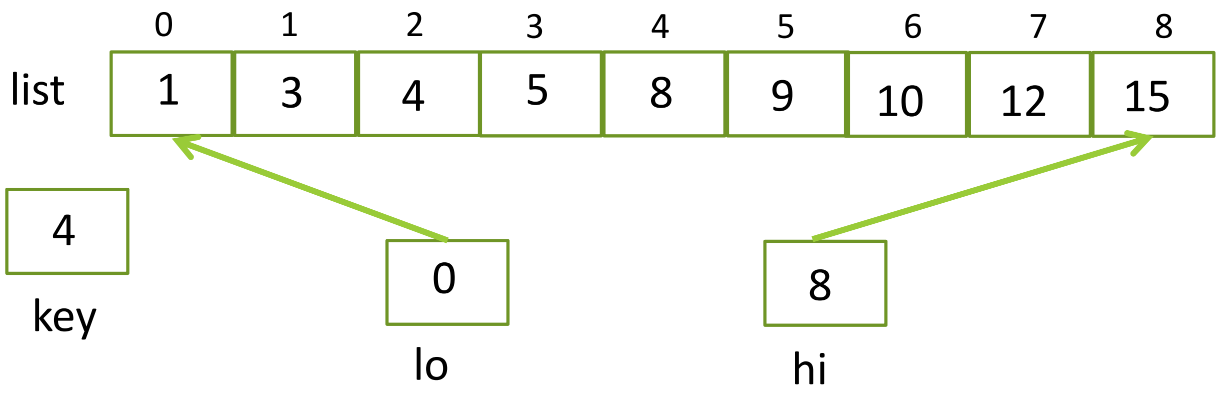
- One step of a binary search:
    - Trying to find key in the portion of the list from list[lo] to list[hi]
    - Pick the middle element in that range
        - list[mid] where mid = (lo+hi)/2
    - If list[mid]==key, you're done, of course
    - If list[mid]>key, then
        - Key can only be from list[lo] to list[mid-1]
            - Everything above list[mid] must be too big, too
        - Change hi = mid-1
    - Similarly if list[mid]<key change lo to mid+1

Binary search

- One step of a binary search:
  - Trying to find key in the portion of the list from list[lo] to list[hi]
  - Pick the middle element in that range
    - list[mid] where mid = (lo+hi)/2
  - If list[mid]==key, you're done, of course
  - If list[mid]>key, then
    - Key can only be from list[lo] to list[mid-1]
      - Everything above list[mid] must be too big, too
    - Change hi = mid-1
  - Similarly if list[mid]<key change lo to mid+1
- Keep going until you find key, or run out of places to look (when lo>hi)

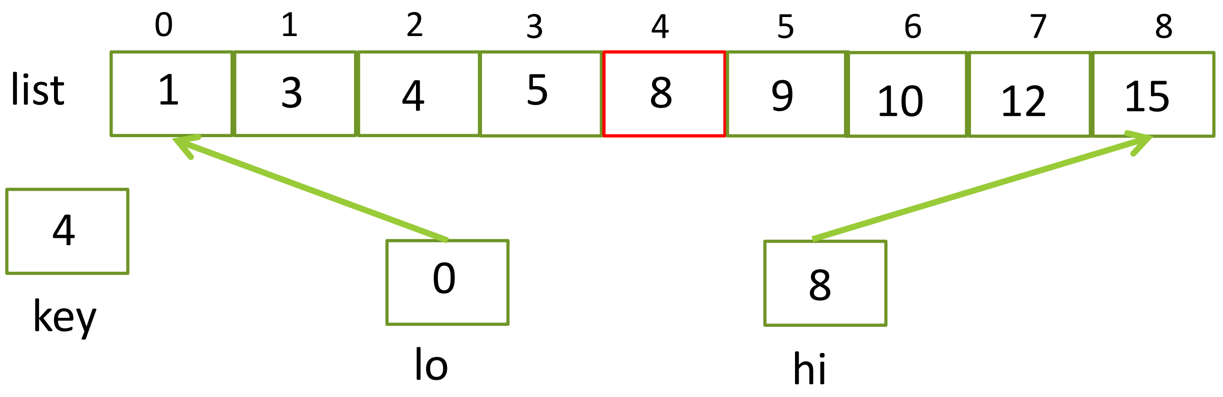
Binary search - Example

- Search this array:



Binary search - Example

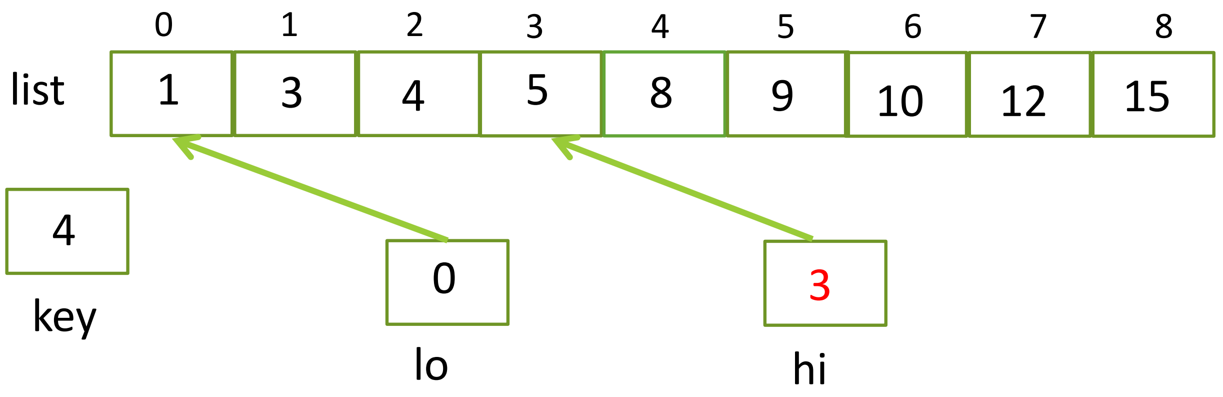
- Search this array:



- Find the middle: mid = (lo+hi)/2 = (0+8)/2 = 4
- Check list[4], which is 8

Binary search - Example

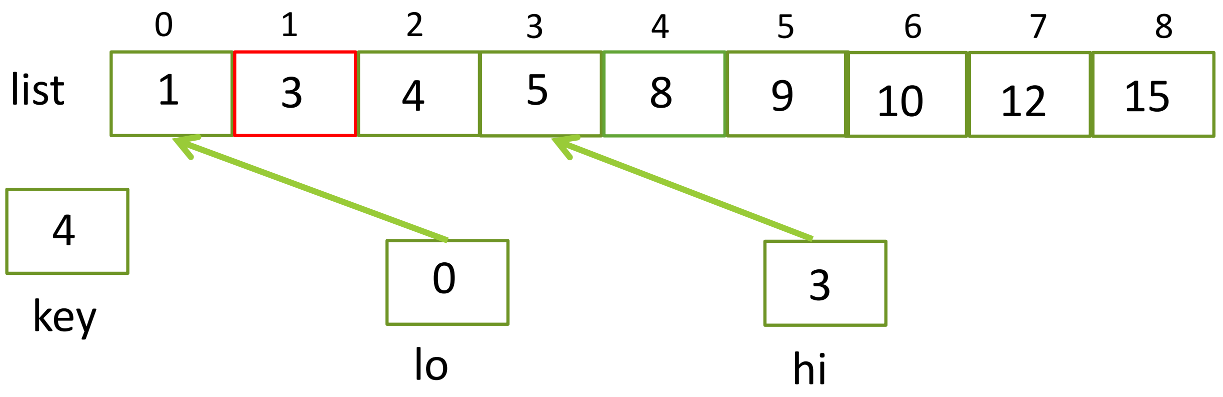
- Search this array:



- Find the middle: mid = (lo+hi)/2 = (0+8)/2 = 4
- Check list[4], which is 8
- It’s too big – change hi to mid-1 = 3

Binary search - Example

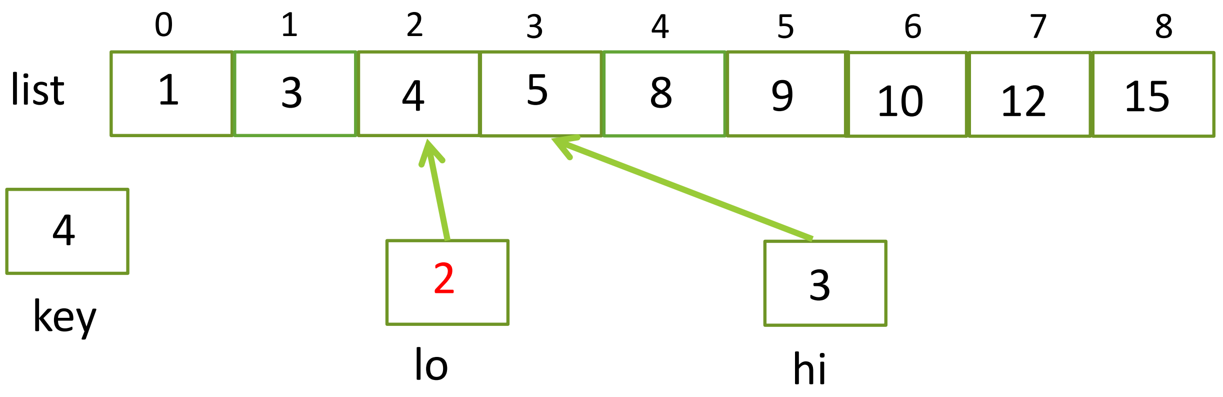
- Search this array:



- Find the middle: mid = (lo+hi)/2 = (0+3)/2 = 1
- Check list[1], which is 3

Binary search - Example

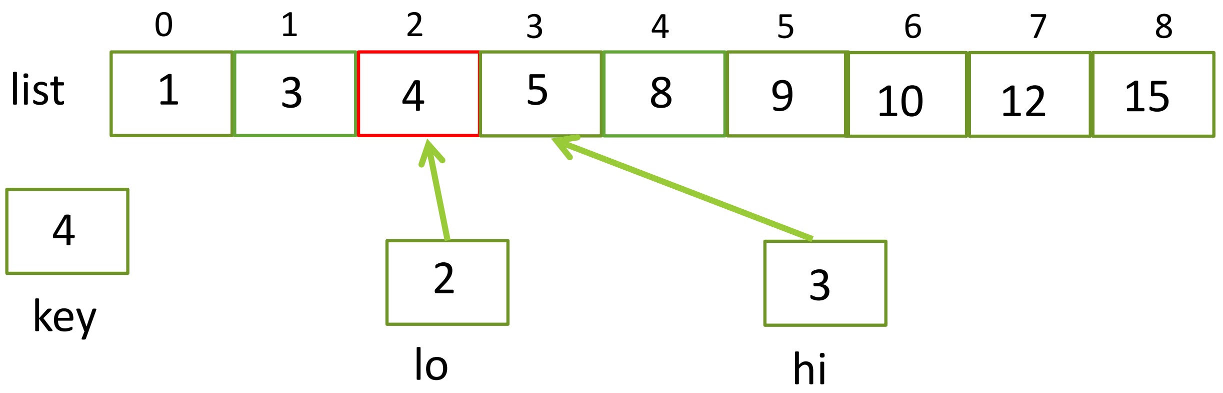
- Search this array:



- Find the middle: mid = (lo+hi)/2 = (0+3)/2 = 1
- Check list[1], which is 3
- It’s too small – change lo to mid+1 = 2

Binary search - Example

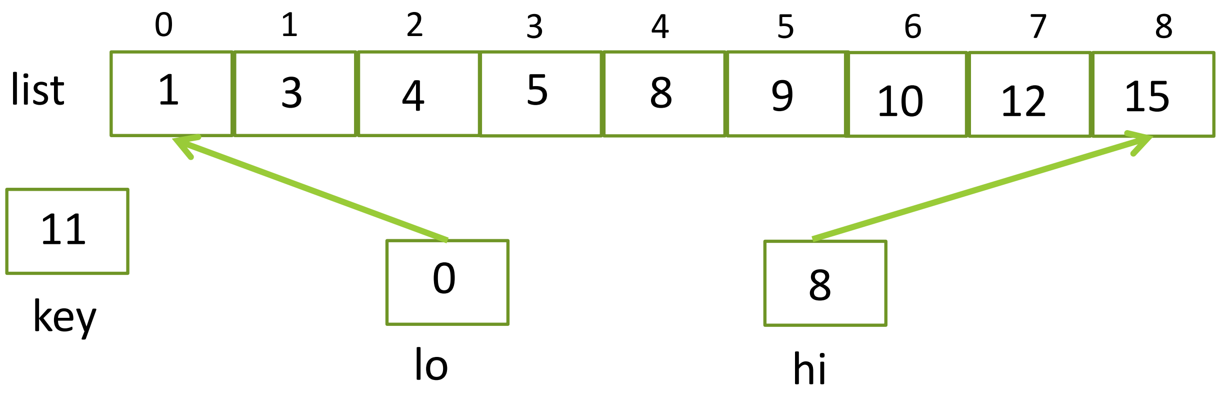
- Search this array:



- Find the middle: mid = (lo+hi)/2 = (2+3)/2 = 2
- Check list[2], which is 4
- Found!

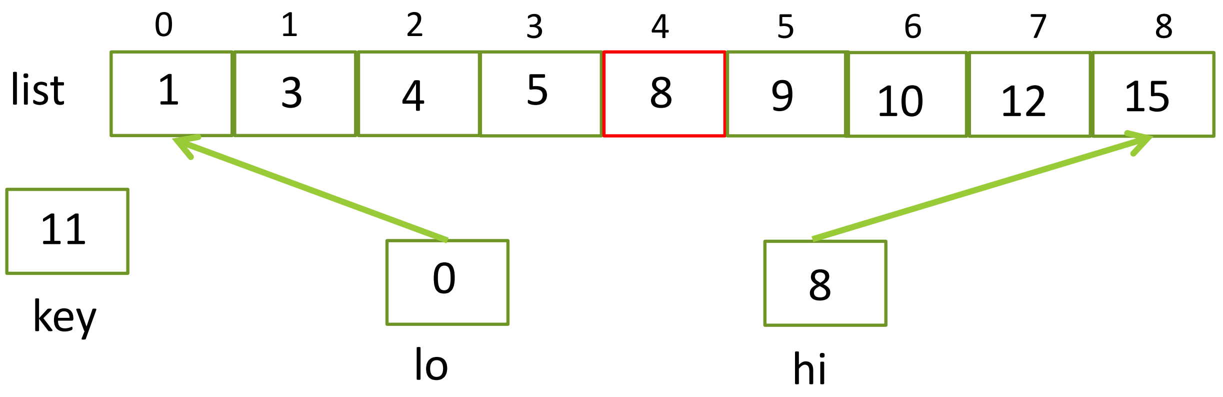
Binary search - Example

- What if the key isn't there?



Binary search - Example

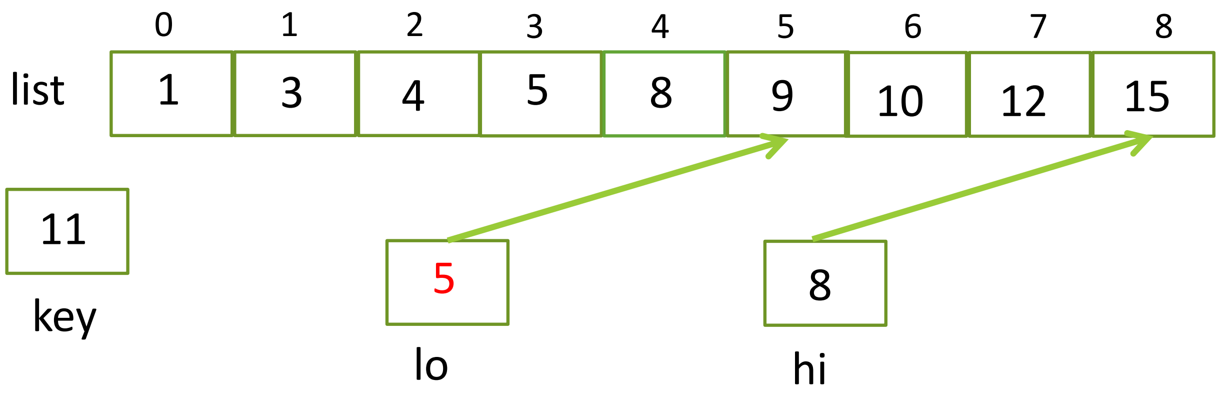
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (0+8)/2 = 4
- Check list[4], which is 8

Binary search - Example

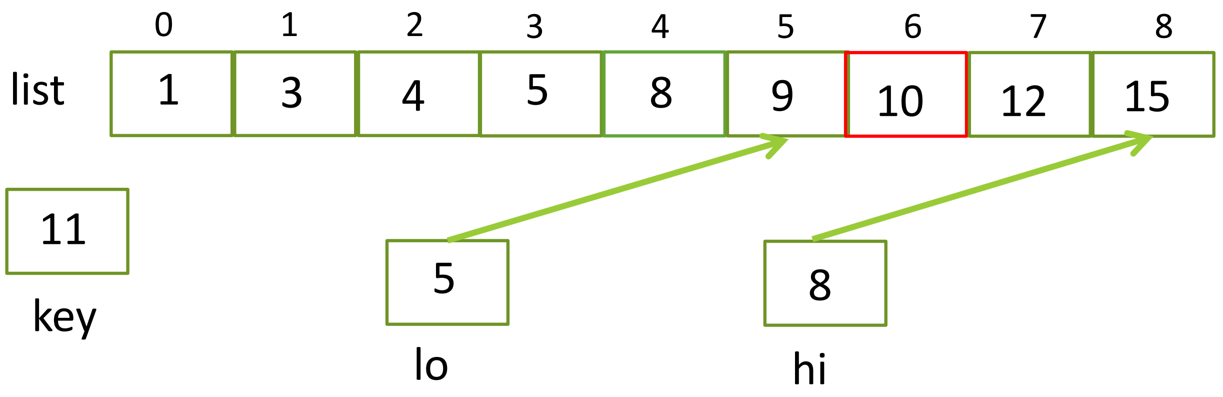
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (0+8)/2 = 4
- Check list[4], which is 8
- It's too small - change lo to mid+1 = 5

Binary search - Example

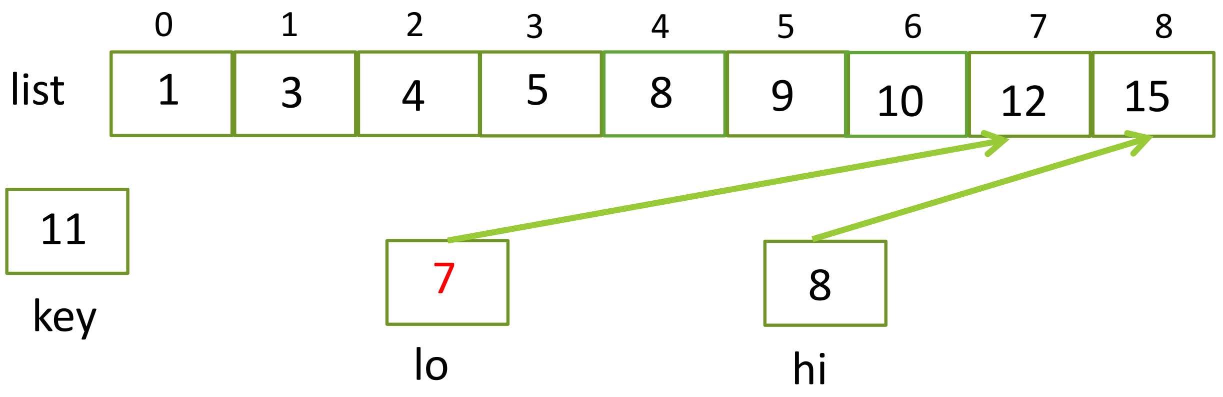
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (5+8)/2 = 6
- Check list[6], which is 10

Binary search - Example

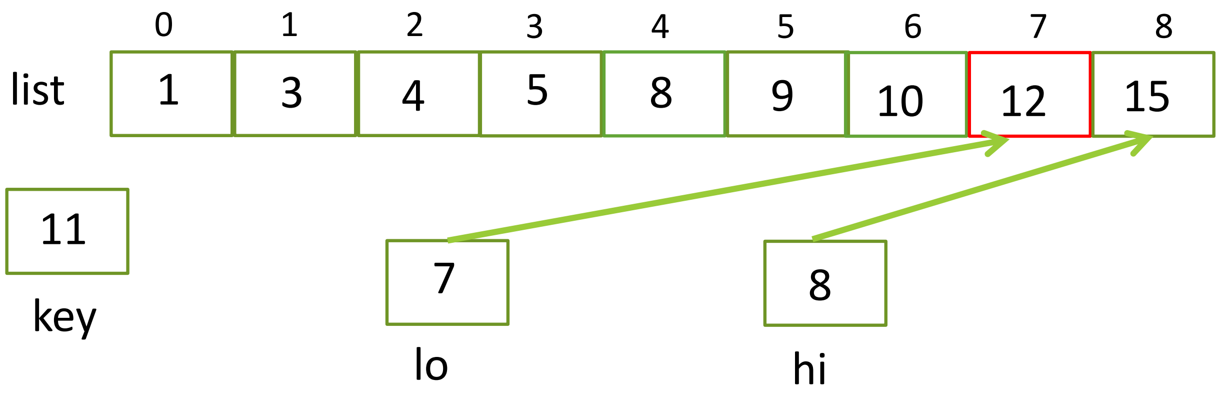
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (5+8)/2 = 6
- Check list[6], which is 10
- It's too small - change lo to mid+1 = 7

Binary search - Example

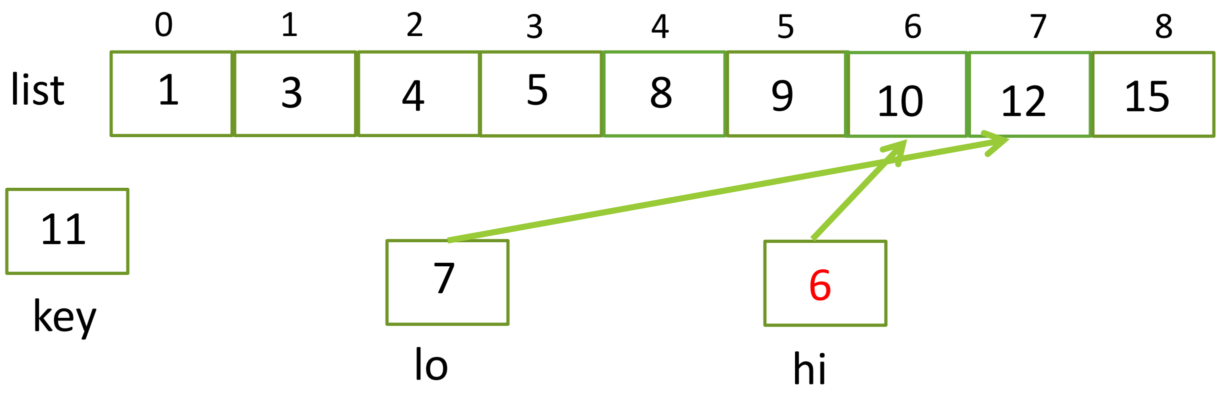
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (7+8)/2 = 7
- Check list[7], which is 12

Binary search - Example

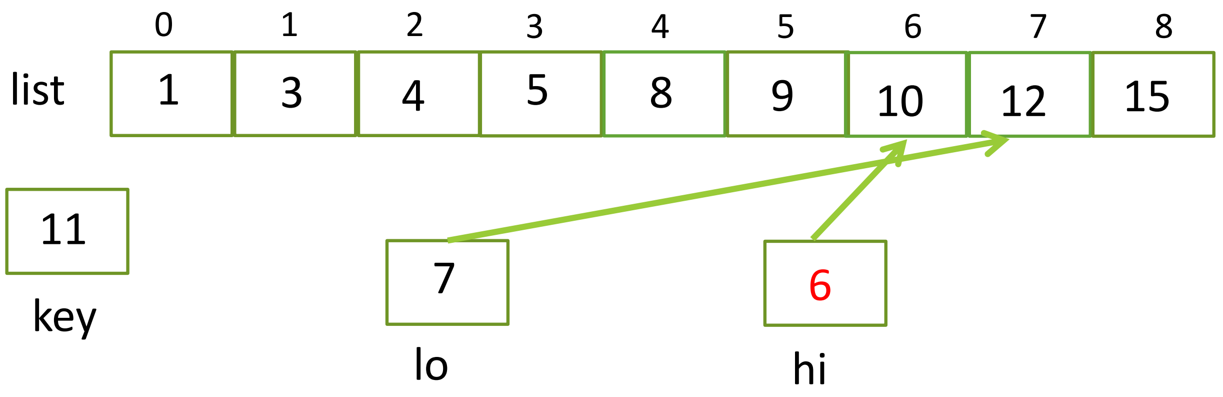
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (7+8)/2 = 7
- Check list[7], which is 12
- It's too big - change hi to mid-1 = 6

Binary search - Example

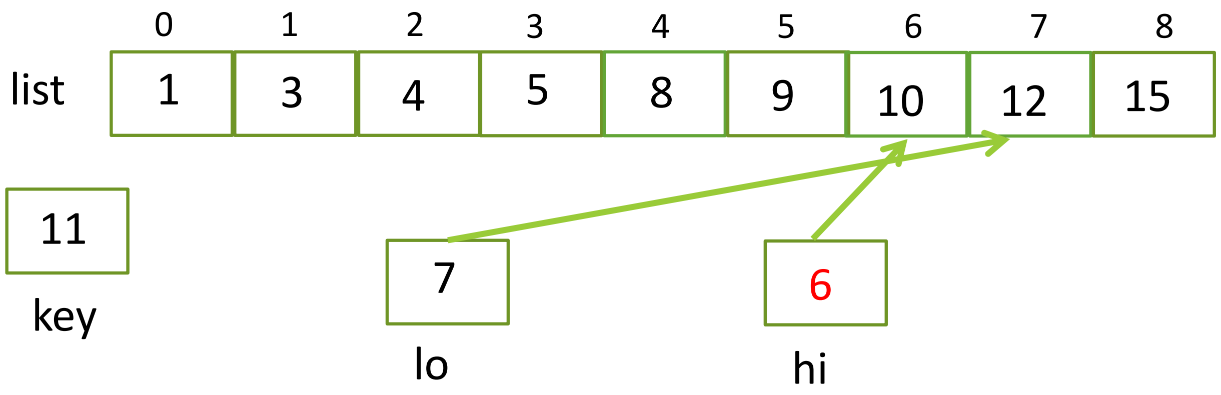
- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (7+8)/2 = 7
- Check list[7], which is 12
- It's too big - change hi to mid-1 = 6
  - Now hi < lo! → impossible to continue!
  - we know now that the key is not in the table

Binary search - Example

- What if the key isn't there?



- Find the middle: mid = (lo+hi)/2 = (7+8)/2 = 7
- Check list[7], which is 12
- It's too big - change hi to mid-1 = 6
  - Now hi < lo! → impossible to continue!
  - we know now that the key is not in the table

Let's write the code for this!

Binary search code (iterative)

```
public static int binarySearch(int[] list, int key){
    int lo=0;
    int hi=list.length-1;
    int mid;
    while(lo<=hi){
        mid=(lo+hi)/2;
        if(list[mid]==key)
            return mid;
        else if(list[mid]<key)
            lo=mid+1;
        else
            hi=mid-1;
    }//while
    return -1;
}//binarySearch
```

Binary search (recursive)

- Consider a method

```
public static int binSearch(int[] data, int key){
//Returns the index of key in data, or -1 if not there
```

- Can this be written recursively, with those parameters?

Binary search (recursive)

- Consider a method

```
public static int binSearch(int[] data, int key){
//Returns the index of key in data, or -1 if not there
```

- Can this be written recursively, with those parameters?
  - No, not quite
  - We would have to use it to search only the first half, or last half, of the array
  - As it is, it can't do that
  - This method signature can only search an entire array

Binary search (recursive)

- But a more general version could be recursive:

```
public static int binSearch(int[] data, int lo, int hi, int key)
{
    //Search the portion from data[lo] to data[hi] only.
    //Return the index of key in this range, or -1.
```

Let’s write the code for this!

Binary search code (recursive)

```
private static int binSearch(int[] data, int lo, int hi, int key){
    if(hi<lo) //There must be an easy non-recursive case
        return -1;
    else {
        int mid = (lo+hi)/2;
        if(data[mid]==key)
            return mid;
        else if(data[mid]<key)
            return binSearch(data,mid+1,hi,key); //Search top half
        else
            return binSearch(data,lo,mid-1,key); //Search bottom half
} }
public static int binSearch(int[] data, int key) {  //interface for the user
    return binSearch(data,0,data.length-1,key);
}
```

Relative speed comparison

- The difference between the two algorithms is huge!

- If you double the list size:
  - The linear search doubles the iterations
  - The binary search adds only 1 iteration!

List Size	Linear iterations	Binary iterations
10	10	4
20	20	5
1000	1000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30
	(max)	(max)

Analyzing speed

- To analyse the speed of an algorithm
    - Count the number of steps or operations needed
    - Look at it as a function of the size of the problem
    - You can look at either the average number of steps, or the maximum number of steps
- For any kind of search, the size of the problem (size of the input) is the size of the list/array – call it n

Analyzing speed

- For a linear search:
  - The loop will be executed an average of n/2 times, or a maximum of n times
  - The number of operations (time) needed for each iteration will be some small constant amount – call it c
  - So the total average number of operations (time) is t(n) = c*n/2, and the maximum time is t(n) = c*n

Analyzing speed

- For a binary search:
  - each iteration will take some small constant amount of time c
  - Each iteration will cut the size of the list in half
    - The maximum number of iterations is related to the maximum number of times you can cut n in half
  - That's $\lceil \log_2 n \rceil$ (ceiling of $\log_2 n$)
    - For n=15 it will search 15, 7, 3, and 1 elements ($\lceil \log_2 15 \rceil = 4$ iterations max)

Analyzing speed

- For a binary search:
  - The average number of operations (the average case) is exactly 1 less than this (it takes some mathematics to prove that - out of the scope of this course)
  - So the average time is t(n) = $c * ((\log_2 n) - 1)$, and the maximum time is t(n) = $c * \log_2 n$

Comparing algorithms

- In summary:
  - Linear search: t(n) = c*n/2 or c*n
  - Binary search: t(n) = c*$\log_2 n$ - 1 or c*$\log_2 n$
  - Constants don't really matter, so you can ignore c's, or ½ or 2 or -1

Comparing algorithms

- The really important thing about an algorithm is: as n grows, how does the number of operations (time) grow?
  - That's determined only by how n (size of input) affects the number of operations

Comparing algorithms

- The really important thing about an algorithm is: as n grows, how does the number of operations (time) grow?
    - That's determined only by how n (size of input) affects the number of operations

- The linear search is "O(n)"
    - If n doubles, the time doubles

- The binary search is "O(log n)"
    - If n doubles, the time goes up by a small increment

Comparing algorithms

- The really important thing about an algorithm is: as n grows, how does the number of operations (time) grow?
  - That's determined only by how n (size of input) affects the number of operations

- The linear search is "O(n)"
  - If n doubles, the time doubles

- The binary search is "O(log n)"
  - If n doubles, the time goes up by a small increment

The big-O notation → you can read it as:
the number of operations (or runtime) is in the order of n, or log n, or any function of n

Extra info: Big-O notation

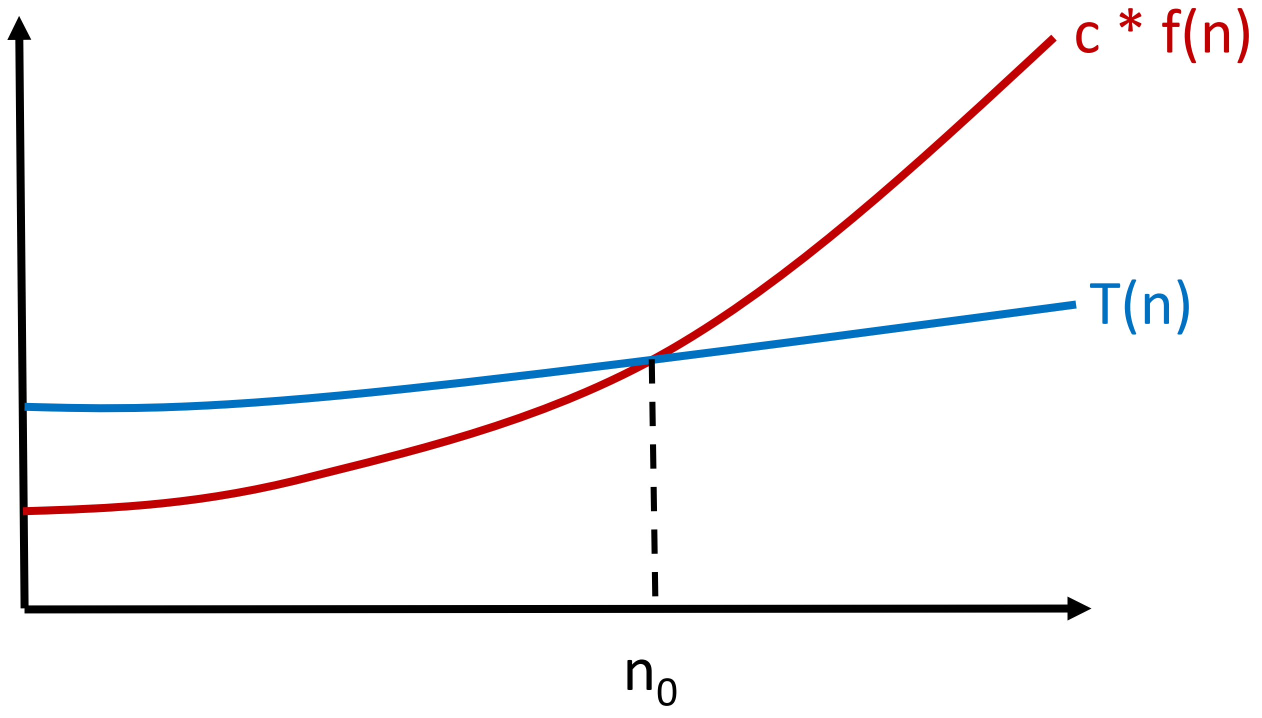
- Let n be the size of the input

- Let $T(n)$ be the function that defines the number of operations (or the space) required by the algorithm on the input n

- $T(n) = O(f(n))$ if for positive constants c and n_0 , $T(n) <= c * f(n)$ when $n >= n_0$

Extra info: Big-O notation

- $T(n) = O(f(n))$ if for positive constants c and n_0 , $T(n) <= c * f(n)$ when $n >= n_0$



Extra info: Big-O notation

- So the goal is to find a function of n, that will be an upper bound on the number of operations that our algorithm uses

  - Let such a function be f(n)

  - Then we can say that the number of operations required by our algorithm is in O(f(n))

Extra info: typical bounds

f(n)	Nb operations
c	constant
logn	logarithmic
n	linear
nlogn	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Extra info: typical bounds

f(n)	Nb operations
c	constant
logn	logarithmic
n	linear
nlogn	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Good

OK

!Danger zone!

Extra info: typical bounds

- You can get a quick estimate of the order by asking "If I double n, what will happen?"

Extra info: typical bounds

- You can get a quick estimate of the order by asking "If I double n, what will happen?"
  - O(log n) – very fast – doubling n adds c to time
  - O(n) – linear – double n and double the time
  - O(n log n) – not much slower than O(n)
  - O(n^2) – slow – double n, 4 times the time
  - O(n^3) – slower – double n, 8 times the time

  - All of the above are "polynomial" time, which is usually considered "computable" (always depends on input size)

Extra info: typical bounds

- You can get a quick estimate of the order by asking "If I double n, what will happen?“
  - O(2^n) – or any constant power n – exponential – double n, square the time!
    - grows more quickly than any polynomial – considered “not computable”, except for very small inputs
  - O(n!) – factorial – nightmare! Don’t even try to run this!

When to use a binary search

- The binary search needs a sorted list
  - You can keep the list in order as you build it
  - Or take an existing list and sort it

When to use a binary search

- The binary search needs a sorted list
    - You can keep the list in order as you build it
    - Or take an existing list and sort it

- Sorting a list (or keeping a list sorted) is even slower than a linear search…

    - So why bother?

When to use a binary search

- Use a binary search if:
  - You happen to have a sorted list already
  - You plan to do a LOT of searching
    - But the list doesn’t change much, so keeping it sorted is easy
  - You have lots of time to sort (overnight, maybe), but when they happen, the searches need to be FAST!

Ordered insertion?

- If we know we might need a sorted array, why not try to keep it ordered at all times, after each insertion?
  - The idea is: always keep it sorted as you're creating it

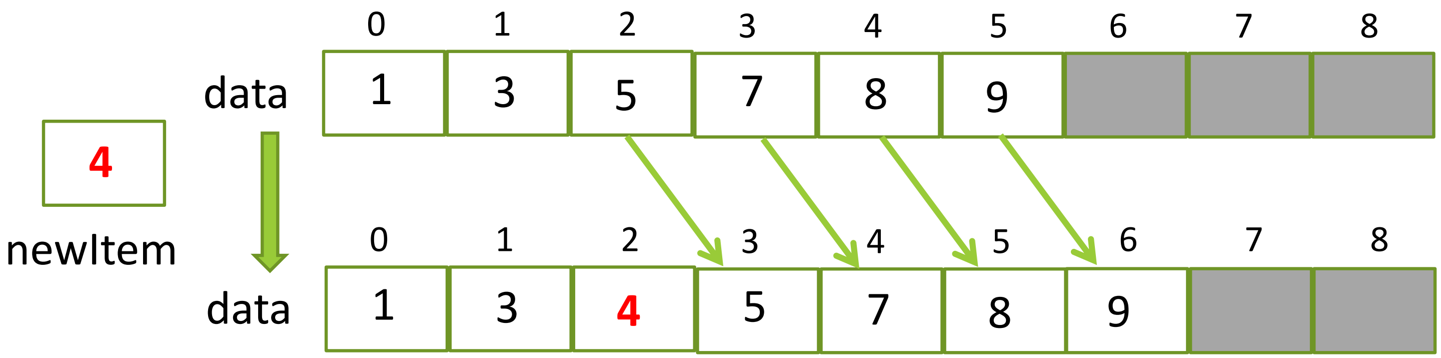
- We have seen this already in previous weeks, but let's look at it one more time

Ordered Insertion – on arrays

- When adding a new element to a (partially-filled) array
  - the old way (adding it to the end) won't work:
    data[numItems++] = newItem;

Ordered Insertion – on arrays

- When adding a new element to a (partially-filled) array
  - the old way (adding it to the end) won't work:
    data[numItems++] = newItem;
- We must now insert it into the proper spot to keep the array sorted (an "ordered insert" – slower, harder):



Ordered Insertion – code

- Assume data[0]..data[n-1] are there, and in order
- Insert newItem so that data[0]..data[n] are in order

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1; //Must start at the high end!
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {  //process larger ones
            data[index+1]=data[index]; //move them up 1 spot
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem; //index is a smaller item (or
}//ordInsert                     // -1). newItem goes next to it.
```

Ordered Insertion – example

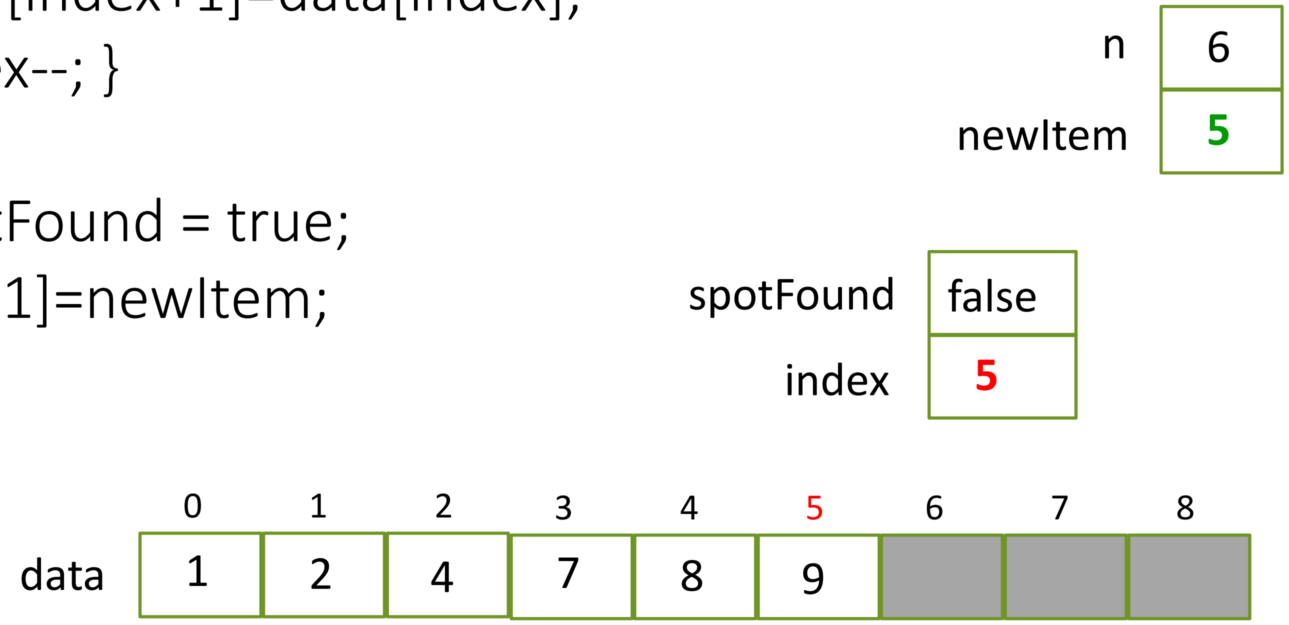
```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```

n	6
newItem	**5**

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

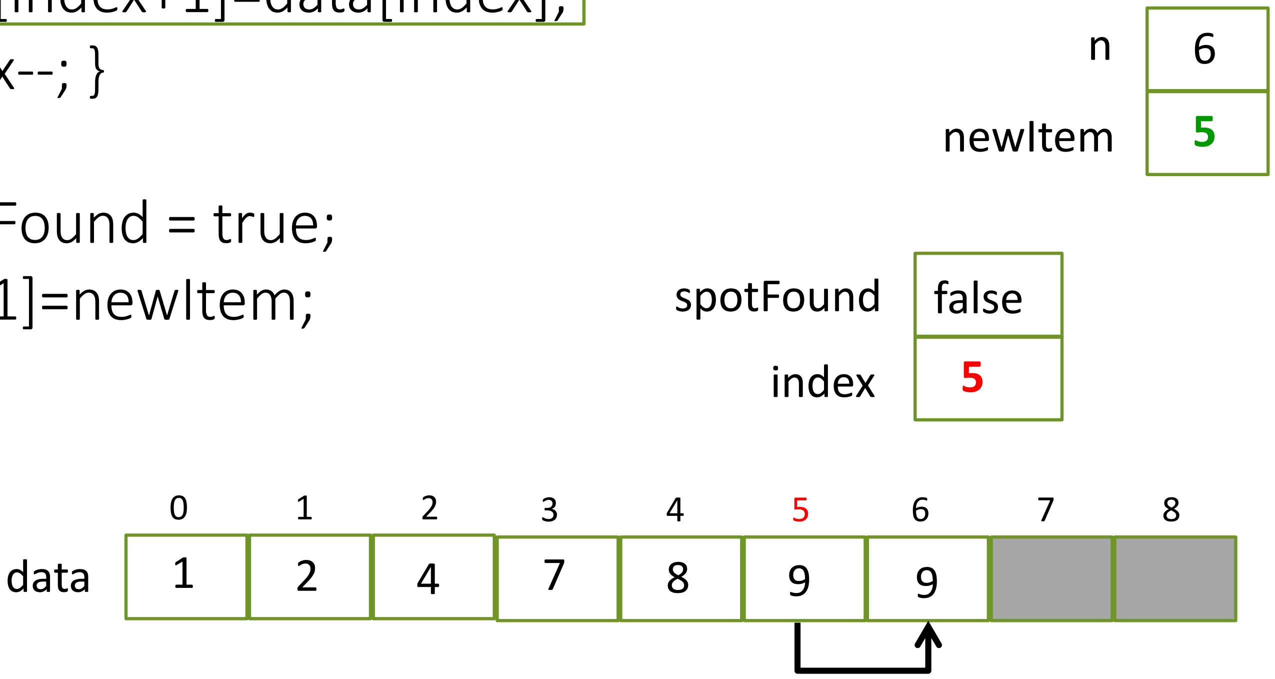
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



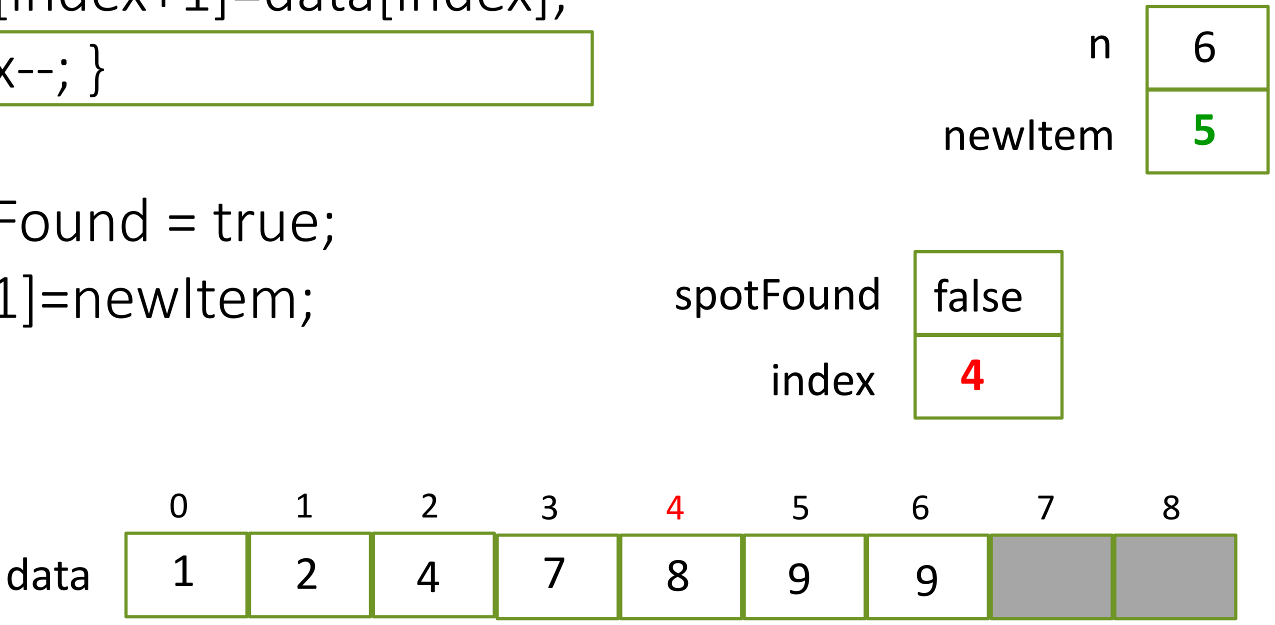
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



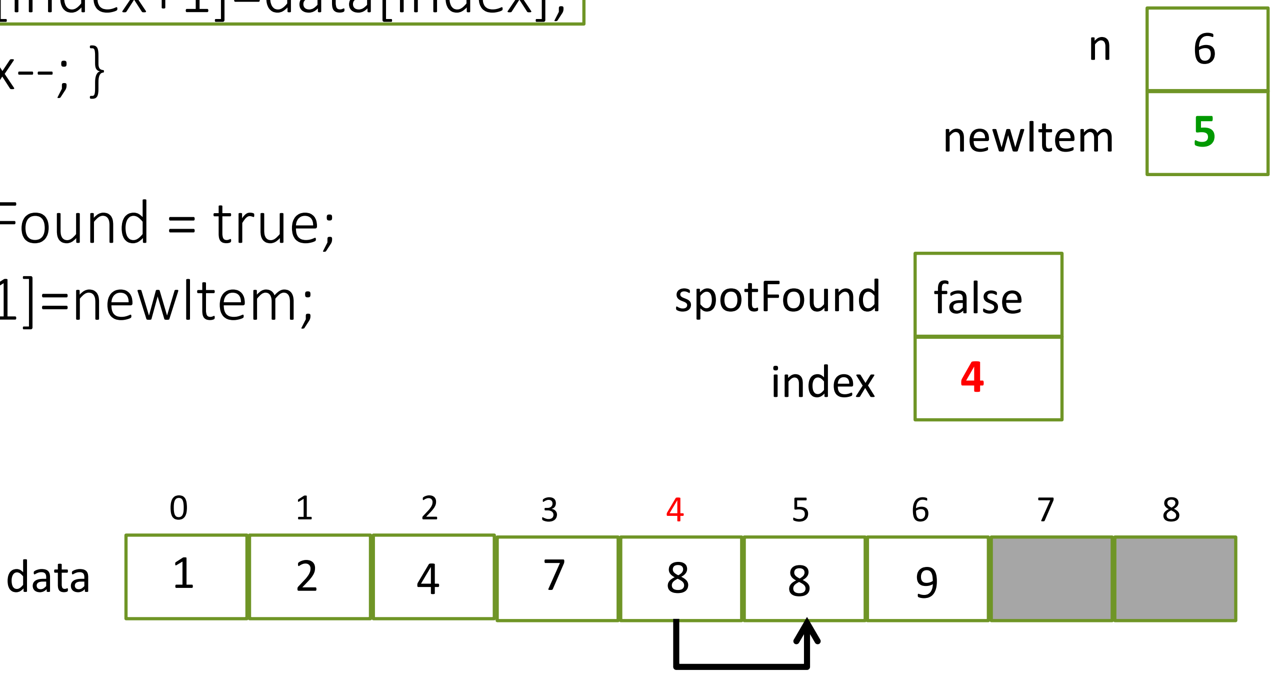
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



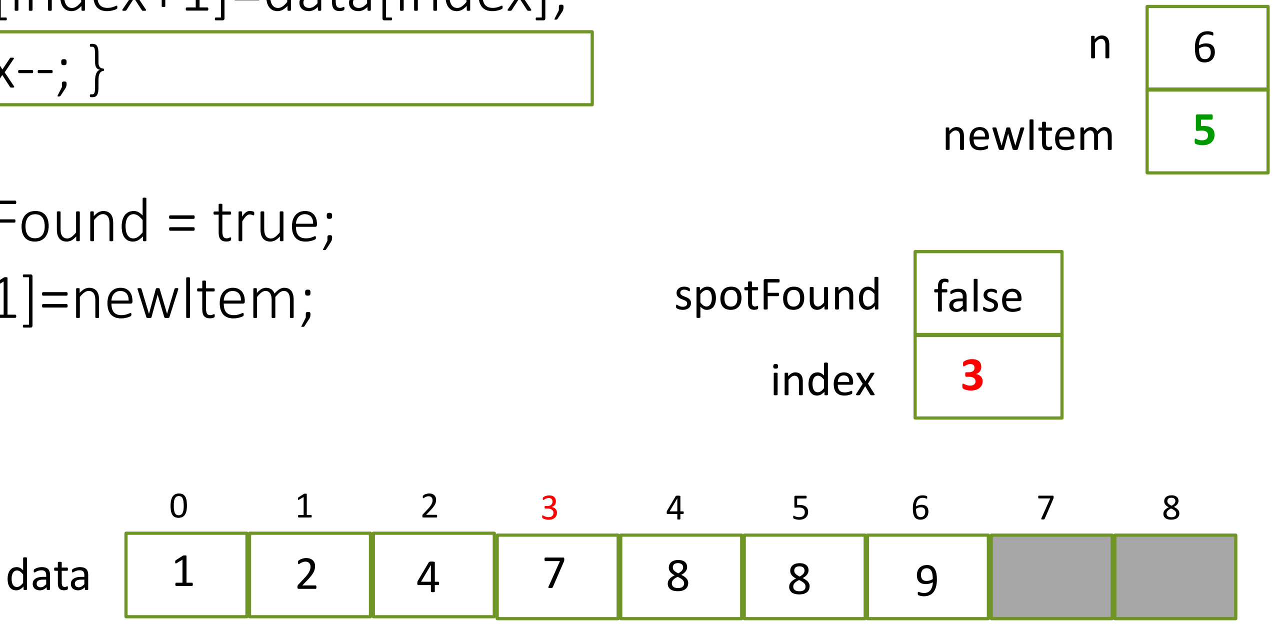
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



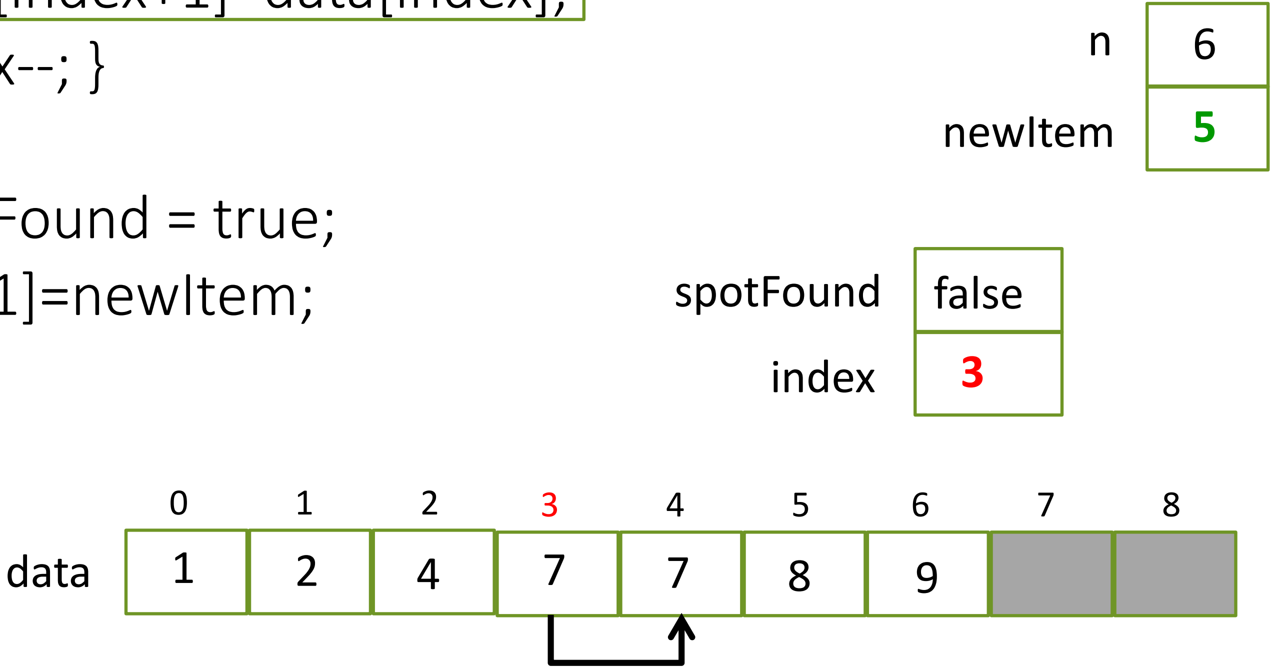
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



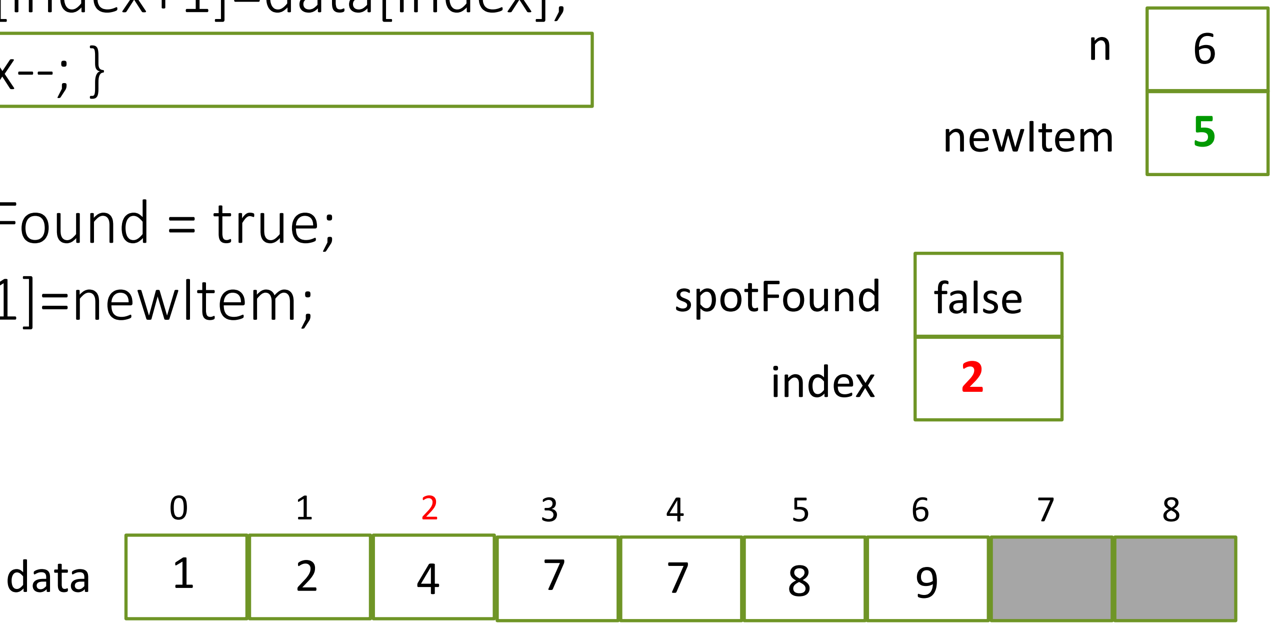
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



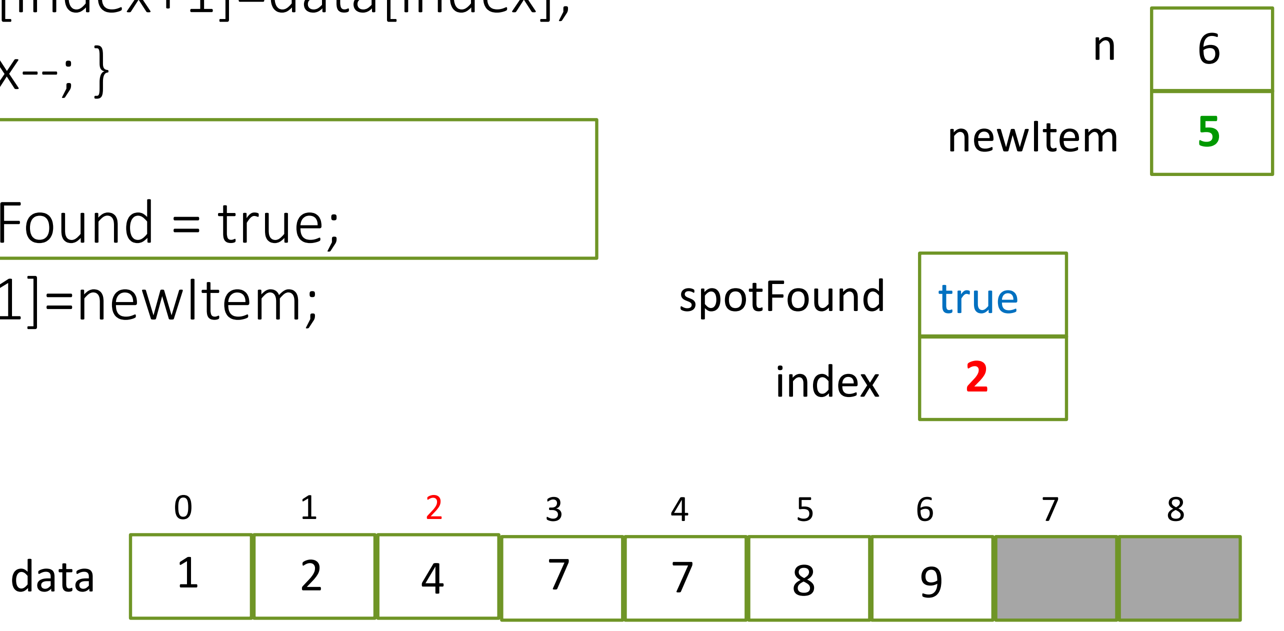
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



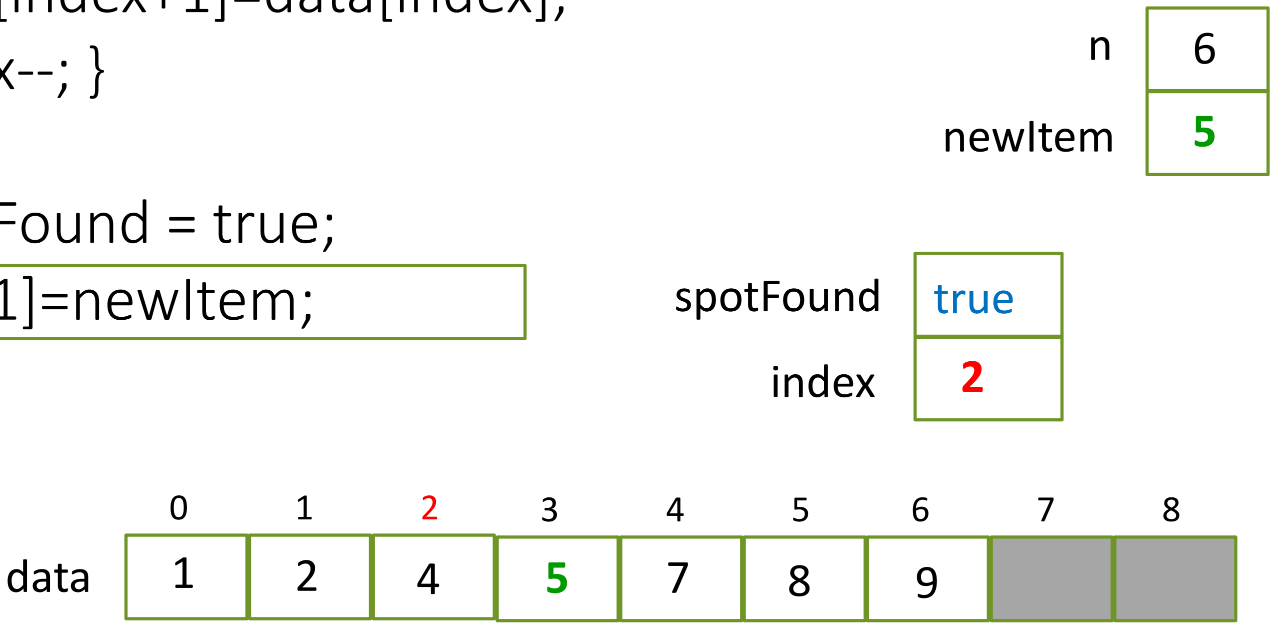
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



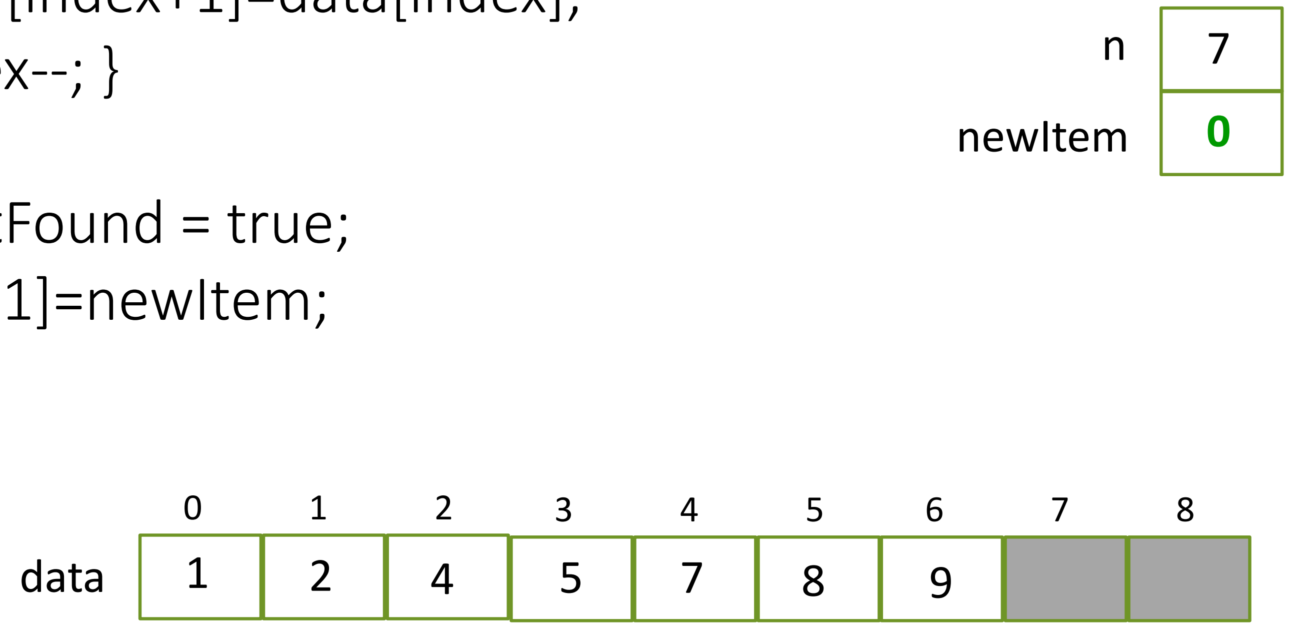
Ordered Insertion – example

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



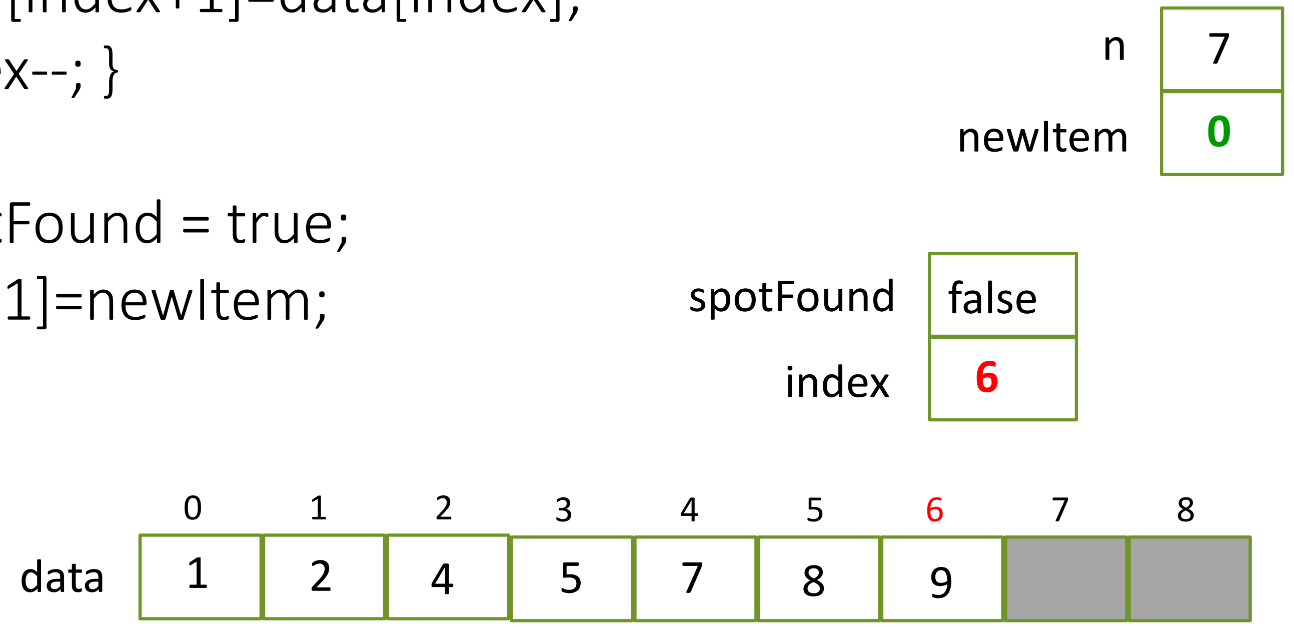
Ordered Insertion – now insert 0

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



Ordered Insertion – now insert 0

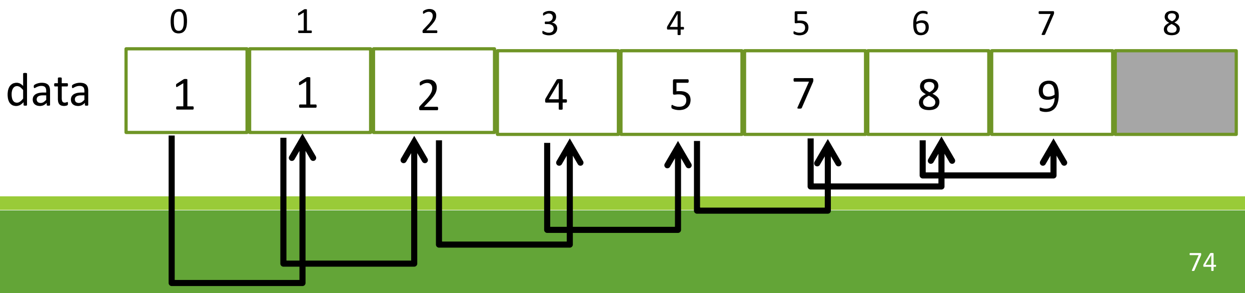
```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```



Ordered Insertion – now insert 0

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```

n	7
newItem	0
spotFound	false
index	-1



Ordered Insertion – now insert 0

```
public static void ordInsert(int n, int[] data, int newItem){
    int index = n-1;
    boolean spotFound = false;
    while(index>=0 && !spotFound)
        if(data[index] > newItem) {
            data[index+1]=data[index];
            index--; }
        else
            spotFound = true;
    data[index+1]=newItem;
}
```

n	7
newItem	**0**

spotFound	false
index	**-1**

	0	1	2	3	4	5	6	7	8
data	**0**	1	2	4	5	7	8	9	

Ordered insert: analysis

- How fast is an ordered insert?

Ordered insert: analysis

- How fast is an ordered insert?
- There's only one loop
- It goes through the array one element at a time
- It might go through all of them (maximum n)
- On average, it will go through half of them

- This is O(n). It's the same as a linear search.
  - It *is* a linear search, really
  - But you're moving elements as you search

Sorting an array?

- Now, let’s see how we can sort an existing array
- There exists many different algorithms, and most of them have the advantage of sorting the array in-place, i.e. they don’t require any additional space!
- We will briefly see some of them (not all of them)

Simple but slow sorting algos

- Insertion sort
  - The best one to use by default
- Selection sort
  - Also usable
  - Good when moving the data around is expensive
    - It does the fewest data movements (but more comparisons)
- Bubble sort (we will not study this one)
  - Very simple to code
  - It's the same "order" as the others, but much slower
  - For more info: https://www.geeksforgeeks.org/bubble-sort/
- All of the above are in $O(n^2)$

More advanced sorting algos

- Merge sort
- Quicksort
  - The best of these, on average
  - But it dies horribly if the list is already sorted!
- Shell sort (we will not study this one)
  - generalization of insertion sort which allows exchanges of elements that are far apart
- Heap sort (very cool, but out of the scope of this class)

- All of these are O(n log n) – more or less

- We'll look at some of these later

Let’s see some sorting algos

- We’ll start by taking a look at the simpler algorithms
  - Insertion sort
  - Selection sort

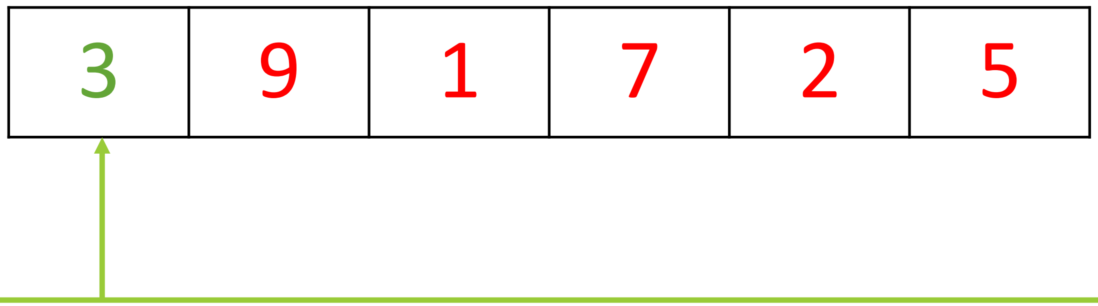
- The main idea of these in-place sorting algorithms is to separate the array into two parts: a sorted part (generally at the beginning) and an unsorted part (generally at the end) and gradually increase the size of the sorted part until everything is sorted

Insertion sort

- If you just want one simple sorting algorithm, this should be the one
- It is relatively easy to implement
- It runs reasonably fast

Insertion sort example

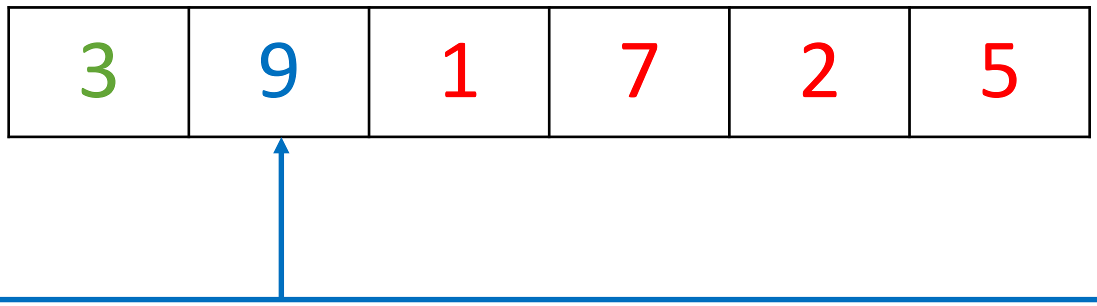
- The idea of insertion sort is, at each step:
    - Insert the first element of the unsorted part in the correct position of the sorted part



When you start the insertion sort, initially the sorted part has only one element, the first one.

Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Then, at each step you choose the first element of the unsorted part, and put it in the correct place!

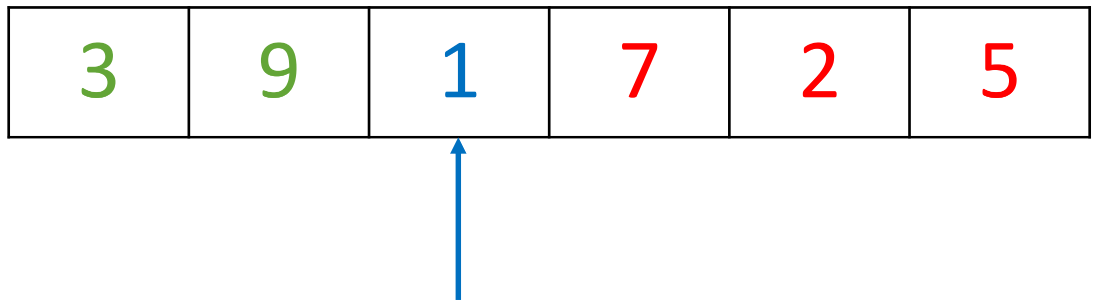
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part

3	9	1	7	2	5

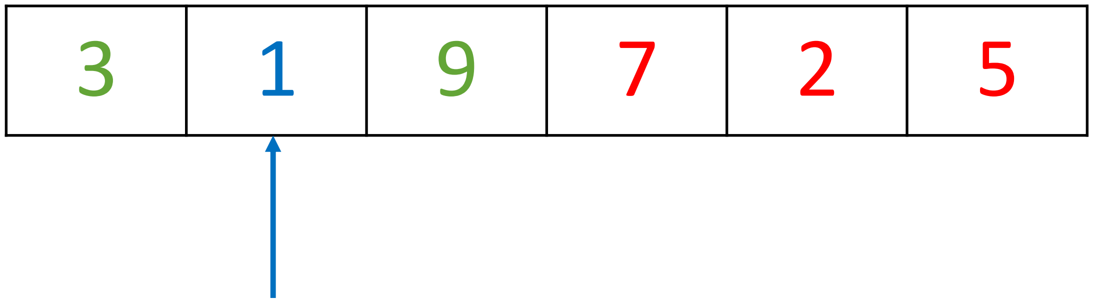
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



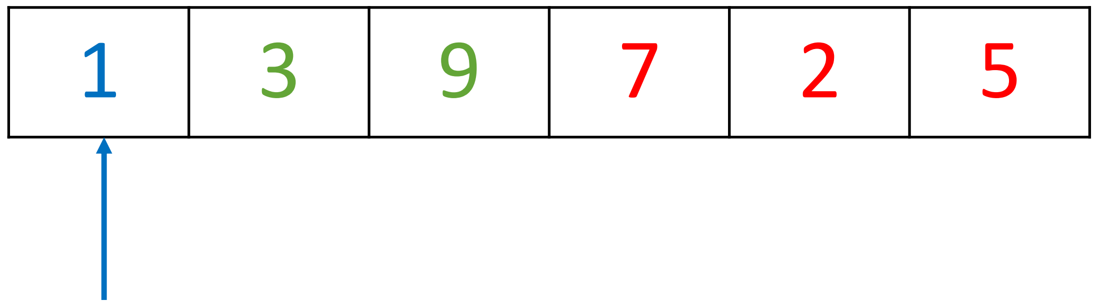
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



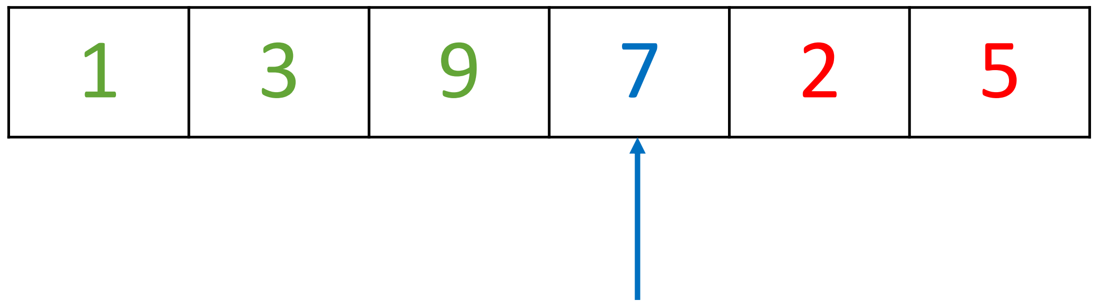
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part

1	3	9	7	2	5

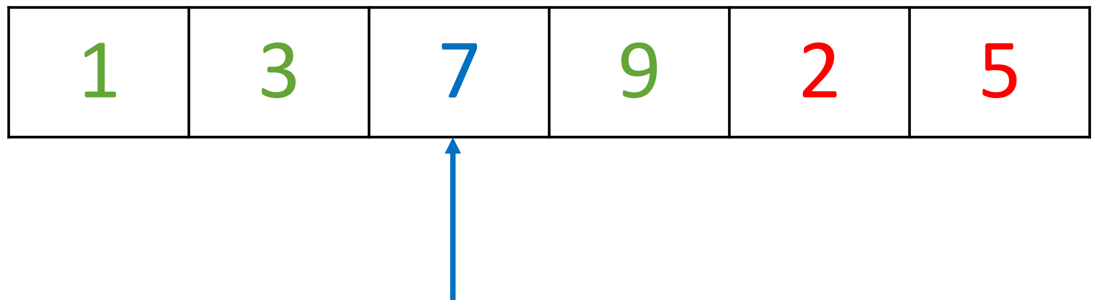
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



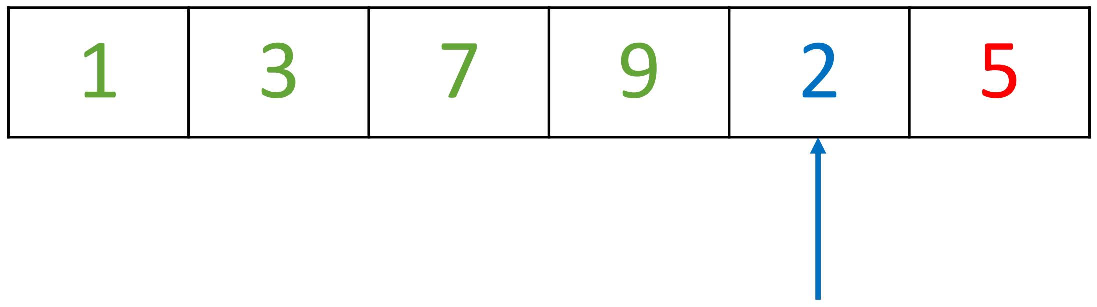
Insertion sort example

- The idea of insertion sort is, at each step:
    - Insert the first element of the unsorted part in the correct position of the sorted part

1	3	7	9	2	5

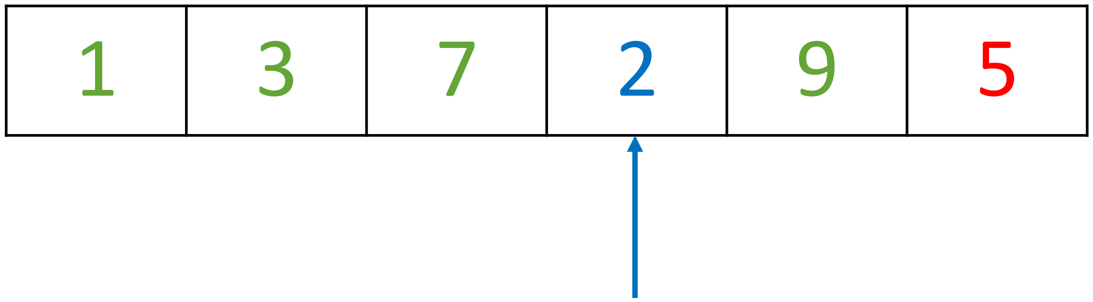
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



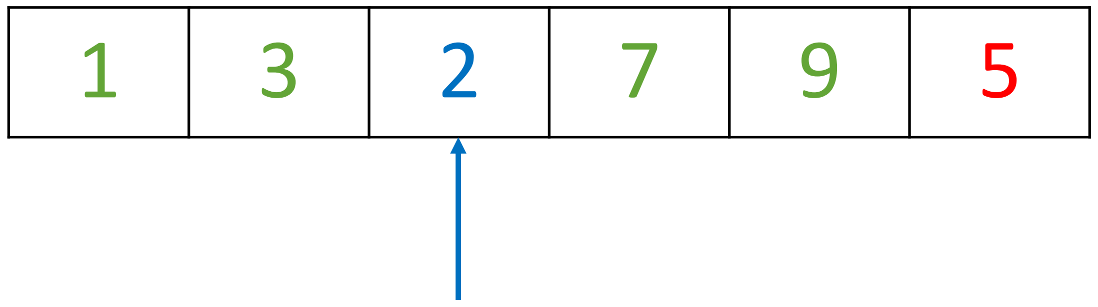
Insertion sort example

- The idea of insertion sort is, at each step:
    - Insert the first element of the unsorted part in the correct position of the sorted part



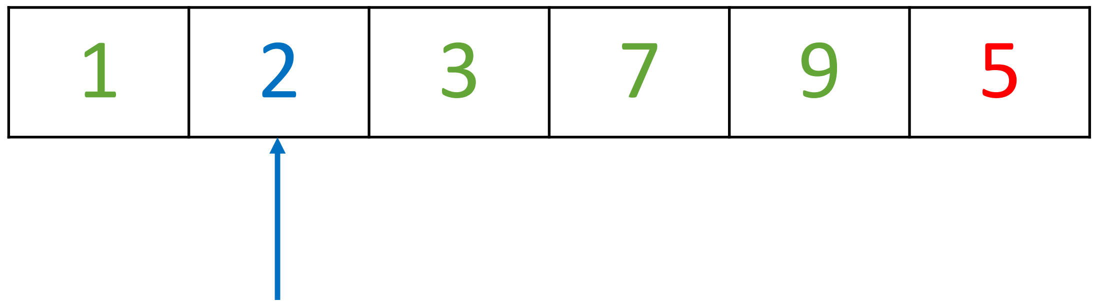
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



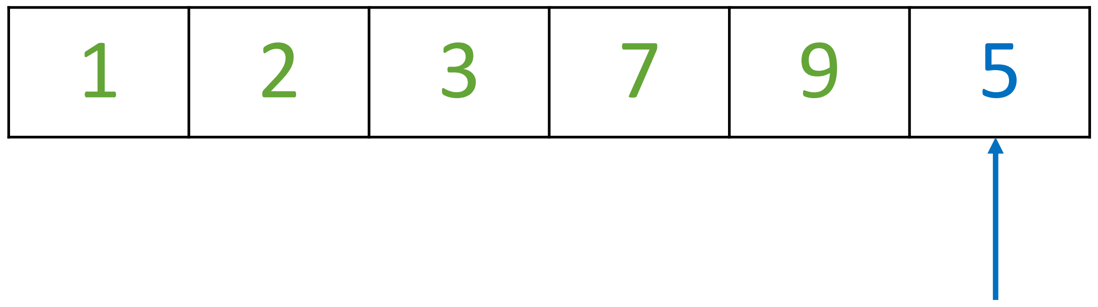
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part

1	2	3	7	9	5

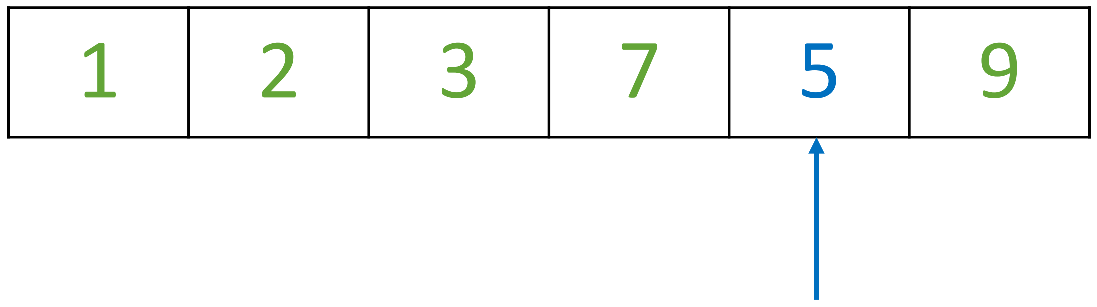
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



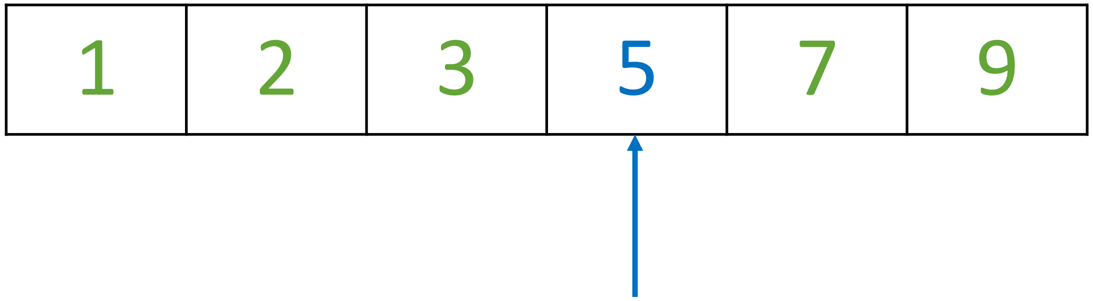
Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Insertion sort example

- The idea of insertion sort is, at each step:
  - Insert the first element of the unsorted part in the correct position of the sorted part



Insertion sort example

- The idea of insertion sort is, at each step:
    - Insert the first element of the unsorted part in the correct position of the sorted part

1	2	3	5	7	9

Done!

Insertion sort

- Basic concept:
  - To sort a[0] to a[n-1], simply use an ordered insert to gradually re-build the list in sorted order:
    - The single-element list a[0] to a[0] is sorted (obviously, a single element is ordered)
    - Insert a[1] to make a[0] to a[1] sorted.
    - Insert a[2] to make a[0] to a[2] sorted.
    - …
    - Insert a[n-1] to make a[0] to a[n-1] sorted. Done!

Insertion sort

- Using our previous ordInsert method (yes, the same one!), the code is simply:

```
for(int k=1; k < n; k++)
    ordInsert(k, a, a[k]);    //Insert a[k] to make
                              // a[0]..a[k] sorted
```

size of the array before insertion (in this case, size of the sorted part)

array

element to insert

Insertion sort - rough analysis

- The insertion sort is:
  ```
  for(int k=1; k < n; k++)
      ordInsert(k, a, a[k]);
  ```
- It contains one simple loop that always runs n times
- Inside that loop it does an ordered insertion, which is O(n) – it does n steps
  - Actually, it does 1, 2, 3, 4, ..., n steps
  - But that's, on average, n/2, which is O(n) anyway
- So we do n steps, n times: this is $O(n^2)$

Selection sort

- Selection sort is another sort that is easy to implement

- It is a little bit slower than insertion sort in practice, even though the maximum number of operations (worst-case scenario) is the same

Selection sort example

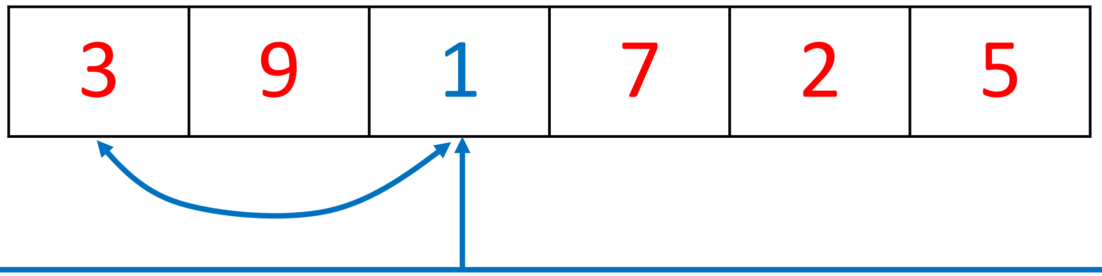
- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

3	9	1	7	2	5

When you start the selection sort, initially the sorted part is empty, everything is unsorted!

Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)



You must find the smallest element of the unsorted part, and swap it with the first element of the unsorted part!

Selection sort example

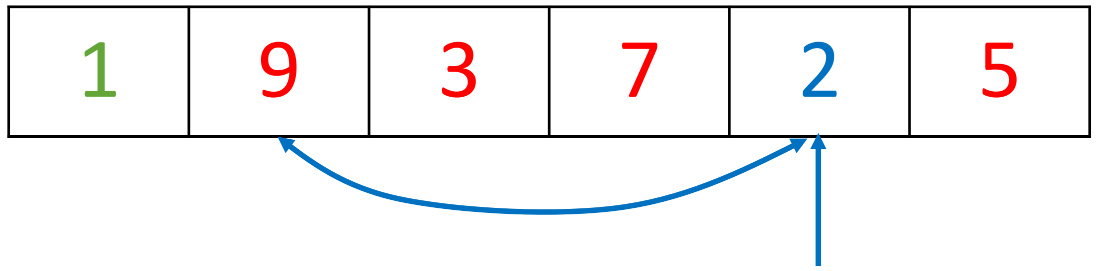
- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

1	9	3	7	2	5

After the swap, this increases the size of the sorted part by 1!

Selection sort example

- The idea of selection sort is, at each step:
    - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)



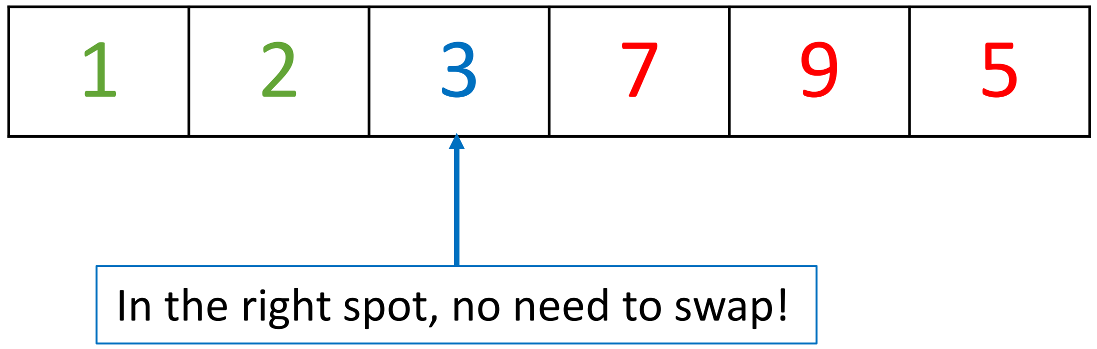
Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

1	2	3	7	9	5

Selection sort example

- The idea of selection sort is, at each step:
    - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)



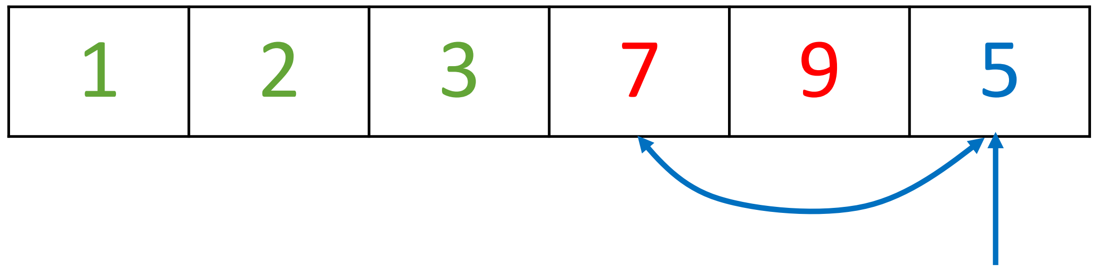
Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

1	2	3	7	9	5

Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)



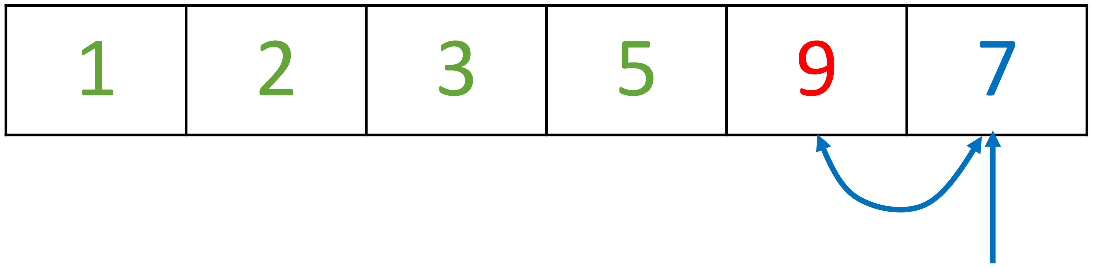
Selection sort example

- The idea of selection sort is, at each step:
    - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

1	2	3	5	9	7

Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)



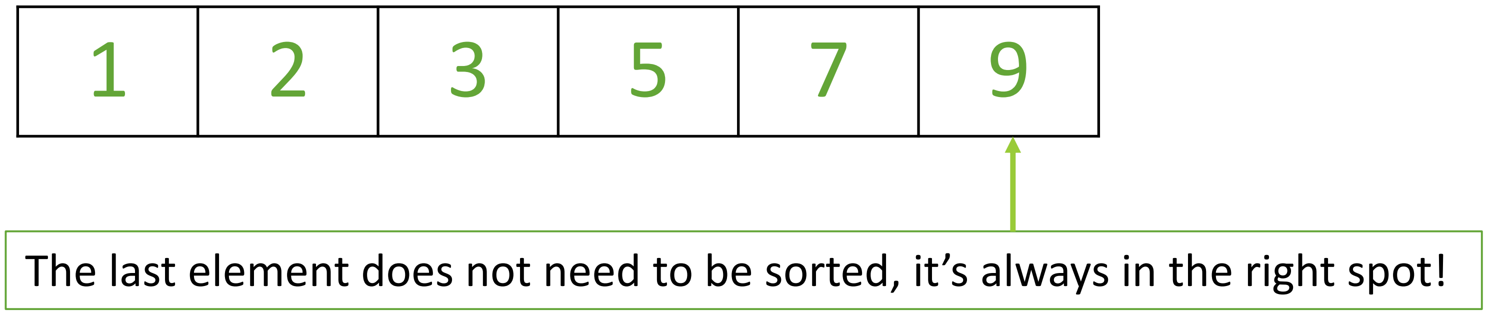
Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)

1	2	3	5	7	9

Selection sort example

- The idea of selection sort is, at each step:
  - Select the smallest element from the unsorted part, and swap it with the element that is after the end of the sorted part (i.e. the first element of the unsorted part)


1
2
3
5
7
9
The last element does not need to be sorted, it’s always in the right spot!

Selection sort

- Another simple sorting idea (again, we're sorting a[0] to a[n-1]):
  - Search a[0]..a[n-1] for the smallest element
    - Swap it into position a[0]
  - Search a[1]..a[n-1] for the next smallest one
    - Swap it into position a[1]
  - Search a[2]..a[n-1] for the next smallest one
    - Swap it into position a[2]
  - …
  - Search a[n-2]..a[n-1] for the next smallest one
    - Swap it into position a[n-2]
  - No need to search a[n-1]..a[n-1] – that's only one left! Done.

Selection sort

- We clearly need a loop:

```
for(int k=0; k<=n-2; k++) {
        //Find the smallest number from a[k] to a[n-1]
        int min = ??;   //The smallest number
        int where = ??; //it was found in a[where]
        ..
        //Swap a[k] and min, which was found in a[where]
        ..
}//for
```

Selection sort

- We clearly need a loop:

```
for(int k=0; k<=n-2; k++) {
    //Find the smallest number from a[k] to a[n-1]
    int min = ??;   //The smallest number
    int where = ??; //it was found in a[where]
    ..
    //Swap a[k] and min, which was found in a[where]
    a[where] = a[k];
    a[k] = min;
}//for
```

Selection sort

- We clearly need another loop:

```
for(int k=0; k<=n-2; k++) {
        //Find the smallest number from a[k] to a[n-1]
        int min = a[k];   //The smallest number
        int where = k; //it was found in a[where]
        for (int i = k+1; i < n; i++) {
            if (a[i] < min) {  //new min!
                    min = a[i];  where = i;
            }
        }
        //Swap a[k] and min, which was found in a[where]
        a[where] = a[k];
        a[k] = min;
}//for
```

Selection sort

- Make it a method:

```
public static void selectionSort(int[] a){
    for(int k=0; k<=a.length-2; k++) {
        //Find the smallest number from a[k] to a[n-1]
        int min = a[k];   //The smallest number
        int where = k; //it was found in a[where]
        for (int i = k+1; i < a.length; i++) {
            if (a[i] < min) {  //new min!
                min = a[i];  where = i;
            }
        }
        //Swap a[k] and min, which was found in a[where]
        a[where] = a[k];
        a[k] = min;
    }//for
}//selectionSort
```

Selection sort - rough analysis

- The selection sort, stripped down, is:

```
for(int k=0; k<=a.length-2; k++) {
        ...small bit of work (initializing min)...
        for(int i=k+1; i<=a.length-1; i++)
        ...small bit of work (updating min if necessary)...
      ...small bit of work (swapping)...
}
```

Selection sort - rough analysis

- The selection sort, stripped down, is:

```
for(int k=0; k<=a.length-2; k++) {
      ...small bit of work (initializing min)...
      for(int i=k+1; i<=a.length-1; i++)
      ...small bit of work (updating min if necessary)...
    ...small bit of work (swapping)...
}
```

- There are two nested loops, both of which are O(n) (a.length is n here)
  - Again, the inner one is only ½ of n, on average, but that's still O(n). Ignore constants like ½ .
- So this is $O(n^2)$, too

Merge sort

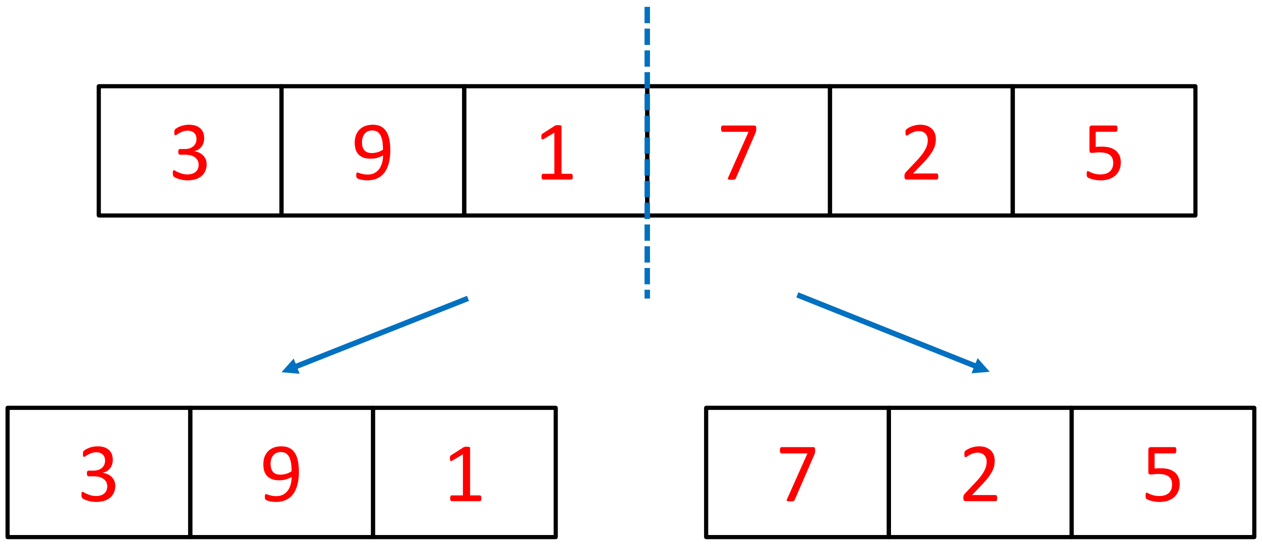
- The basic merge sort algorithm is:
  - Split the array into two small arrays (half each)
  - Sort the two halves (using 2 merge sorts)
    - Recursion! x2!
  - Merge the two sorted halves into a sorted array after the 2 merge sorts have returned

Merge sort example

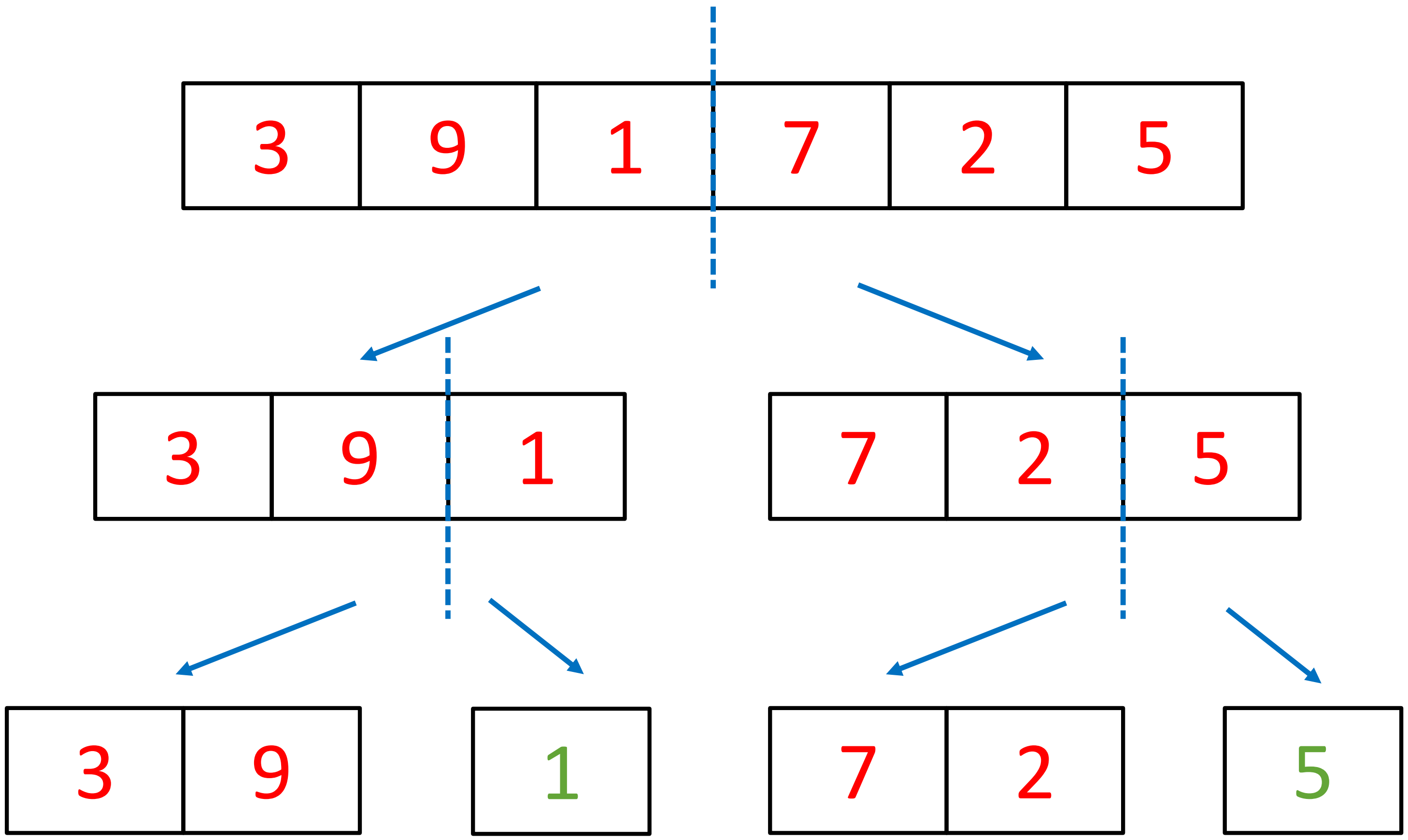
3	9	1	7	2	5

First, the recursive calls will divide the arrays in half until we get to the base case, which is when we get an array with only one element → nothing to sort (an array of one element is sorted by default)

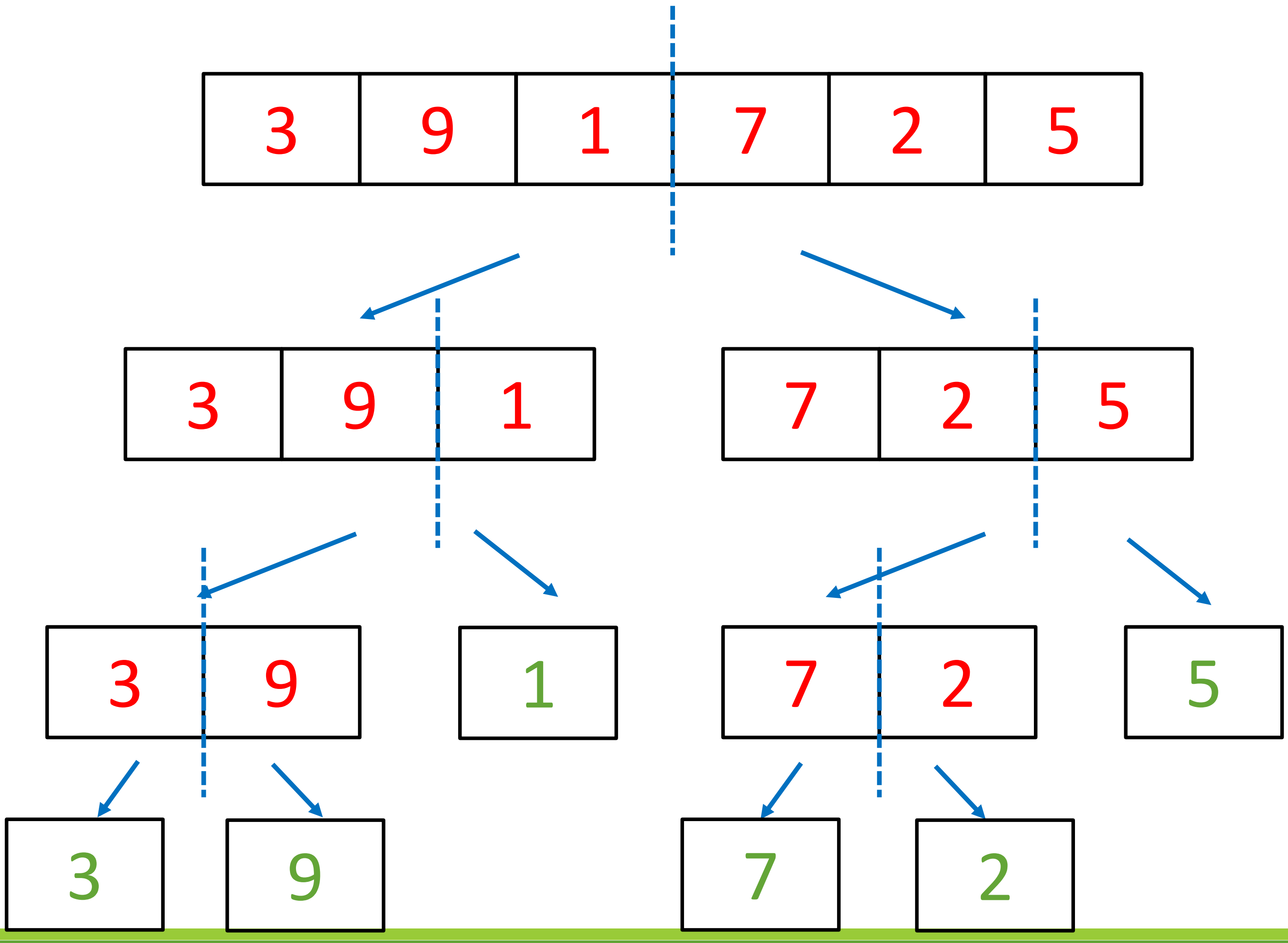
Merge sort example



Merge sort example

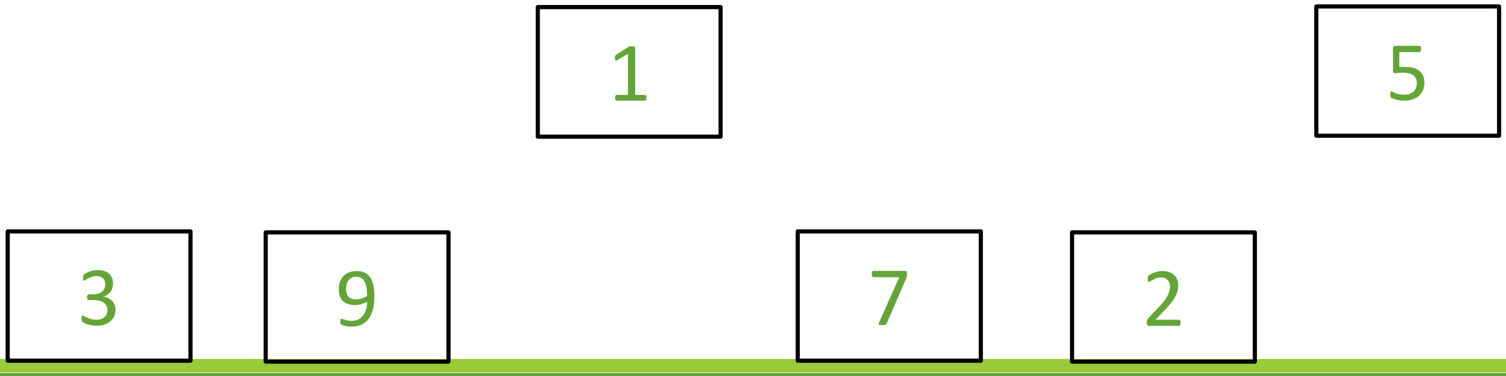


Merge sort example

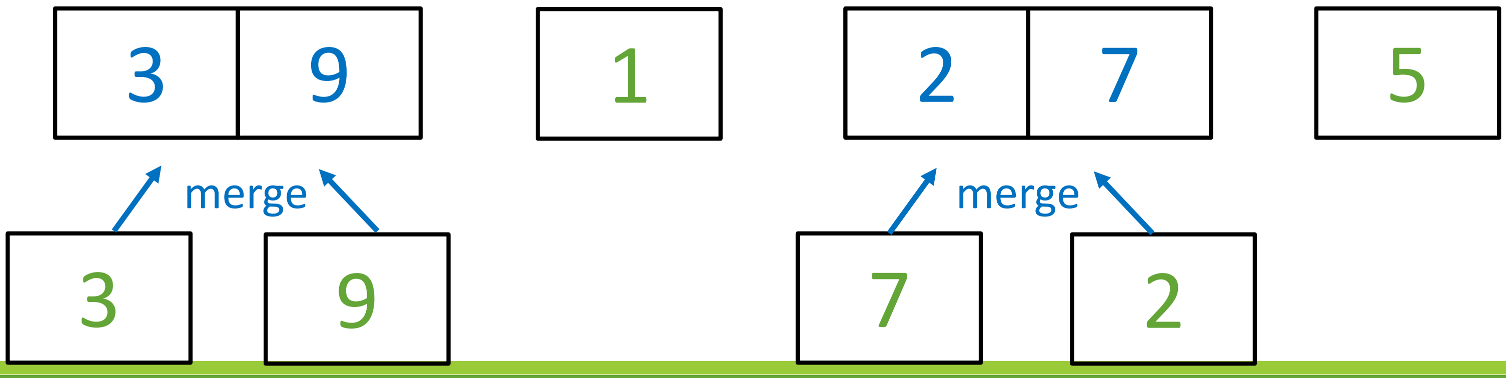


Merge sort example

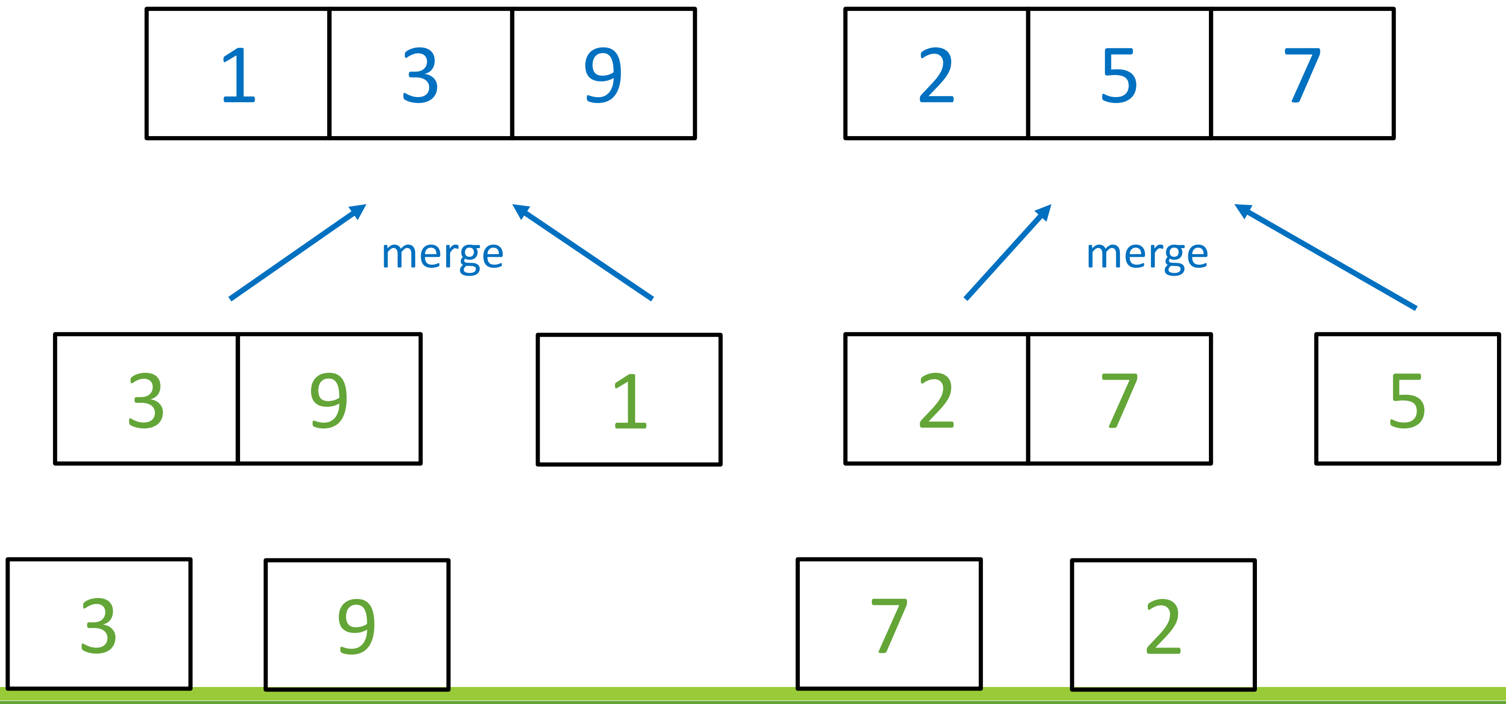
Now, when the recursive calls return, we have to merge the two halves together, in the right order, to create an array that is sorted!



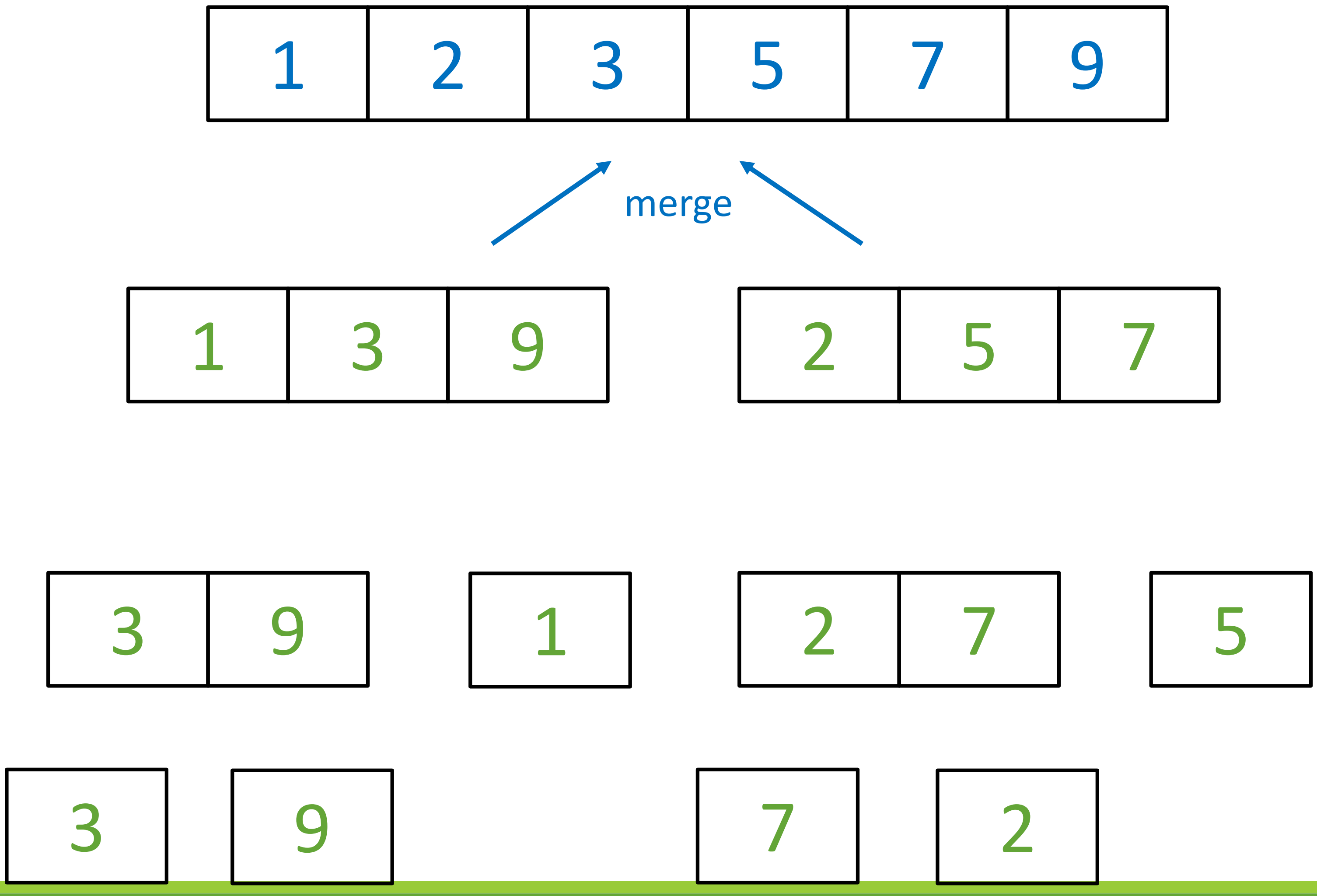
Merge sort example



Merge sort example



Merge sort example



Merge sort example

1	2	3	5	7	9

Done!

Merge sort analysis

- This is a fast sort
    - In the order of what, exactly?

Merge sort analysis

- This is a fast sort
  - In the order of what, exactly?

- The arrays are divided in half a total of logn times (we have seen this before → binary search!)

- On each "level" (you can see the levels in the previous example; there are logn levels), we have to merge O(n) elements

- Result: merge sort is in O(nlogn)

Merge sort

- You can see an efficient implementation of merge sort in SearchSort.java
    - You don't have to learn the actual code by heart; just focus on understanding the general idea
    - The general idea is explained in the next slides

Merge sort

- What is the general idea?
  - We will have a recursive mergeSort method
  - Base case: only 1 element (nothing to do!)
  - Otherwise:
    - Split the array into 2 halves, use 2 recursive calls to mergeSort on both halves
    - Merge the two resulting halves together, in order! That’s it!

Merge sort

- The merging part is the "trickiest" bit. The idea is:
  - Look at the first element in each of the two halves
  - Remove the smaller one and append it to the answer
  - Repeat until one list is empty
  - Append the remaining elements from the other one

Quick sort

- The basic quick sort algorithm is:
  - Choose a “pivot” value
  - Partition step: place everything that is smaller (resp. bigger) than the pivot on the left (resp. right) side of the array (not necessarily in order at this point in time)
  - Place the pivot in the middle of the left part and right part
  - Call quickSort on the left and right parts

Quick sort example

3	9	1	7	2	5

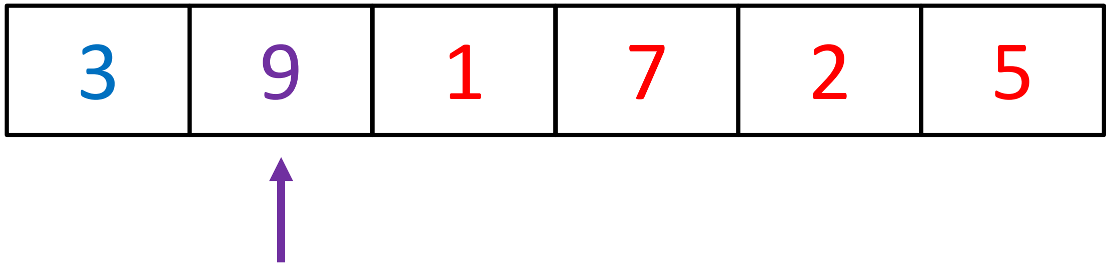
Choose a pivot first. There are many ways you can choose the pivot: the easiest option is to take the first element in the array (note that this easiest option can fail terribly if the array is already sorted).

Quick sort example

3	9	1	7	2	5

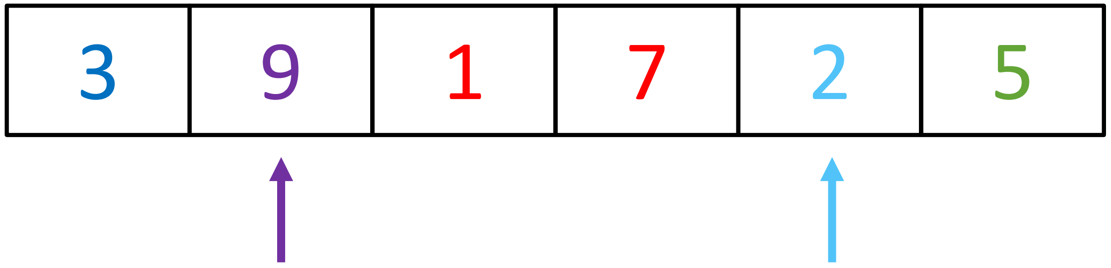
Choose a pivot first. There are many ways you can choose the pivot: the easiest option is to take the first element in the array (note that this easiest option can fail terribly if the array is already sorted).

Quick sort example



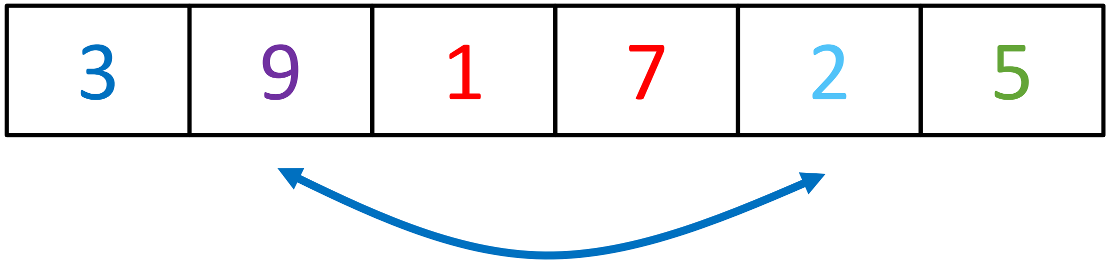
After the pivot, search from the left of the array for the first value that is bigger than the pivot.

Quick sort example



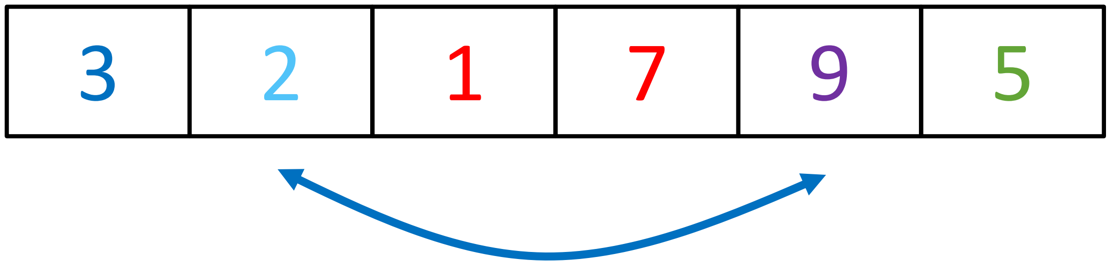
Similarly, search from the right side of the array for the first value that is smaller than the pivot.

Quick sort example



Swap them.

Quick sort example



Swap them.

Quick sort example

3	2	1	7	9	5

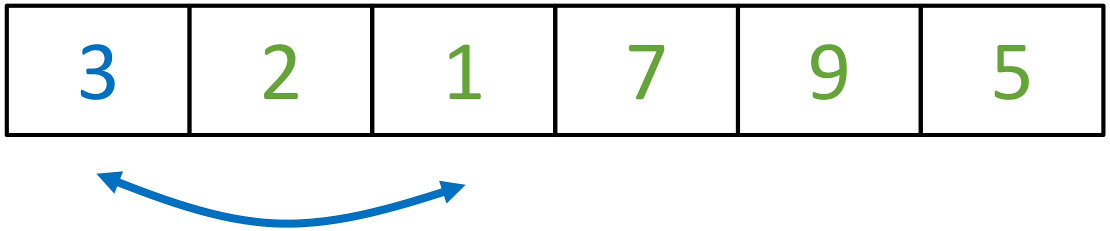
Keep doing that until you meet in the middle.

Quick sort example

3	2	1	7	9	5

Keep doing that until you meet in the middle.
In this example, there is nothing else to swap, except the pivot itself.

Quick sort example



Keep doing that until you meet in the middle.
In this example, there is nothing else to swap, except the pivot itself.

Last step is to move the pivot to the middle (at the end of the half that is smaller).

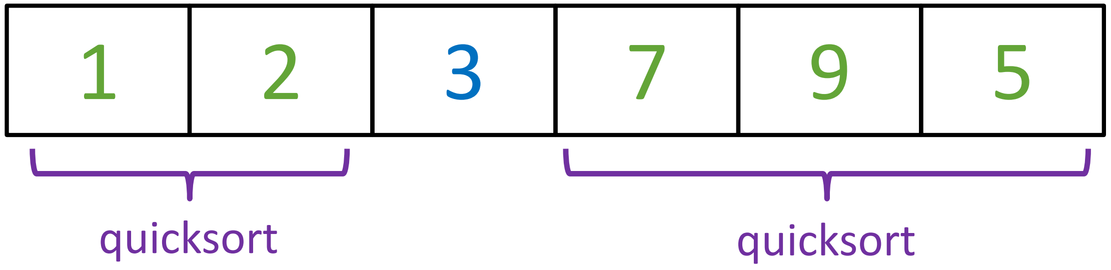
Quick sort example

1	2	3	7	9	5

Keep doing that until you meet in the middle.
In this example, there is nothing else to swap, except the pivot itself.

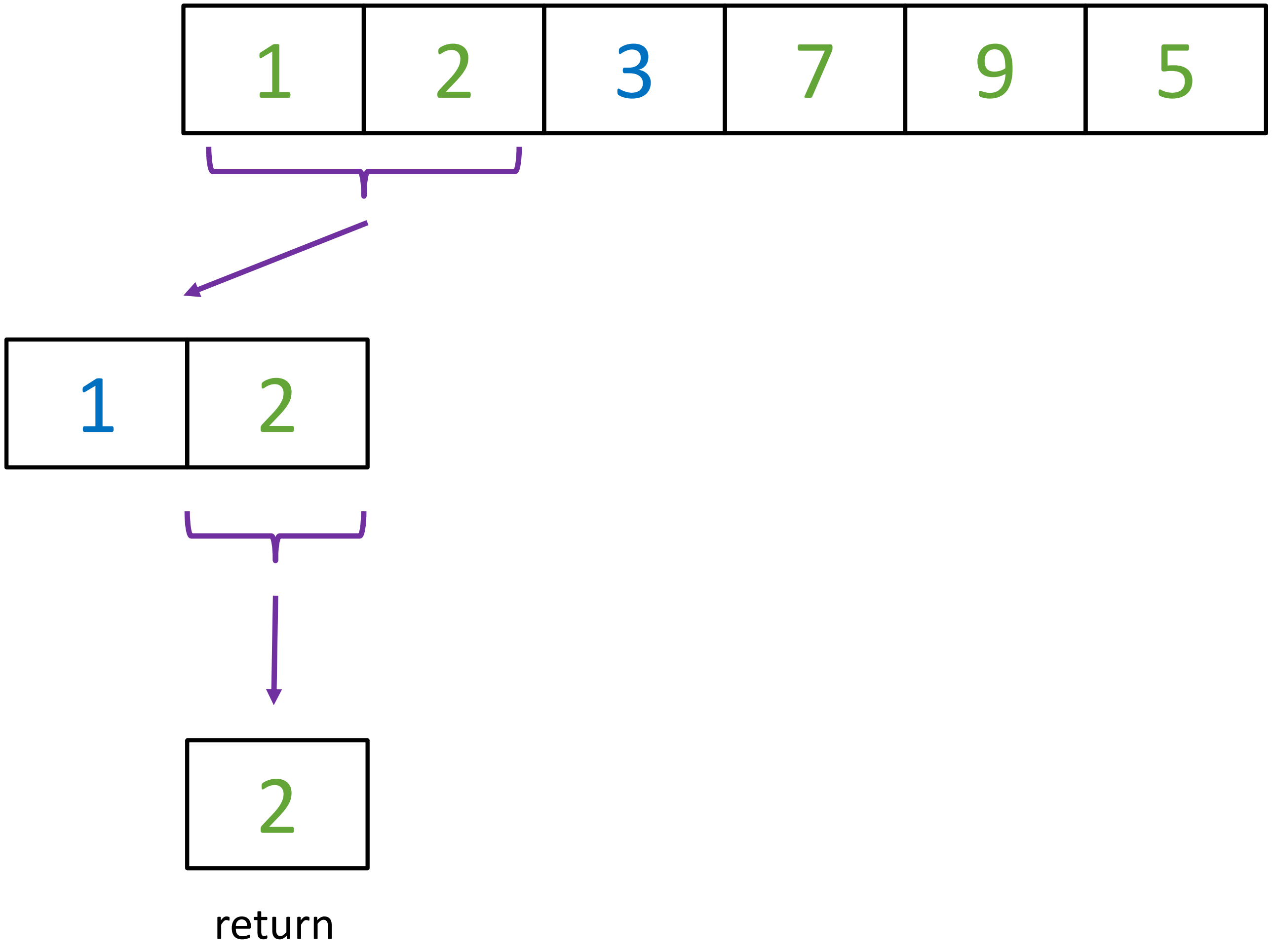
Last step is to move the pivot to the middle (at the end of the half that is smaller).

Quick sort example

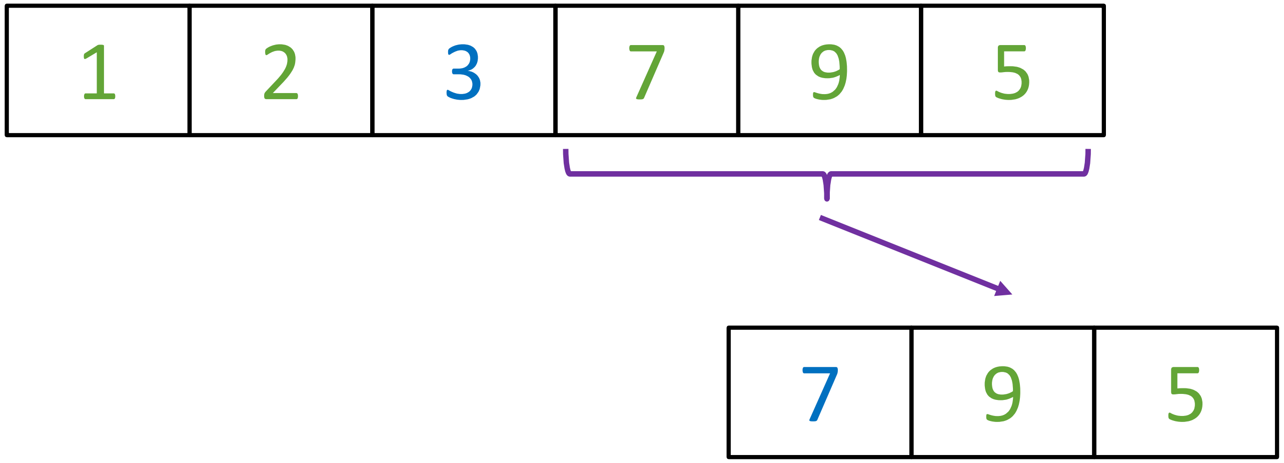


Now we make 2 recursive calls to sort both sides of the pivot (excluding the pivot, because it is now in its final spot).

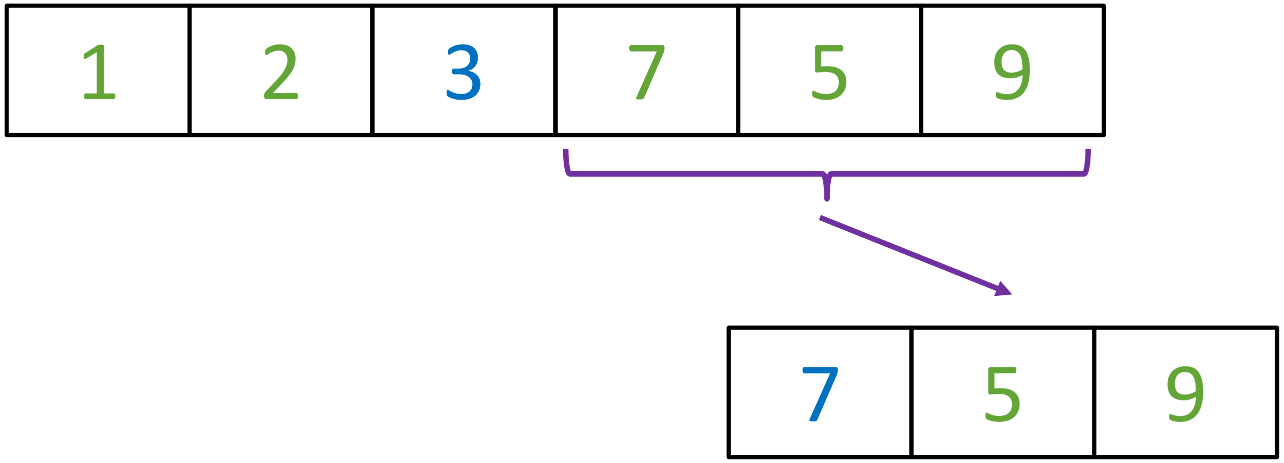
Quick sort example



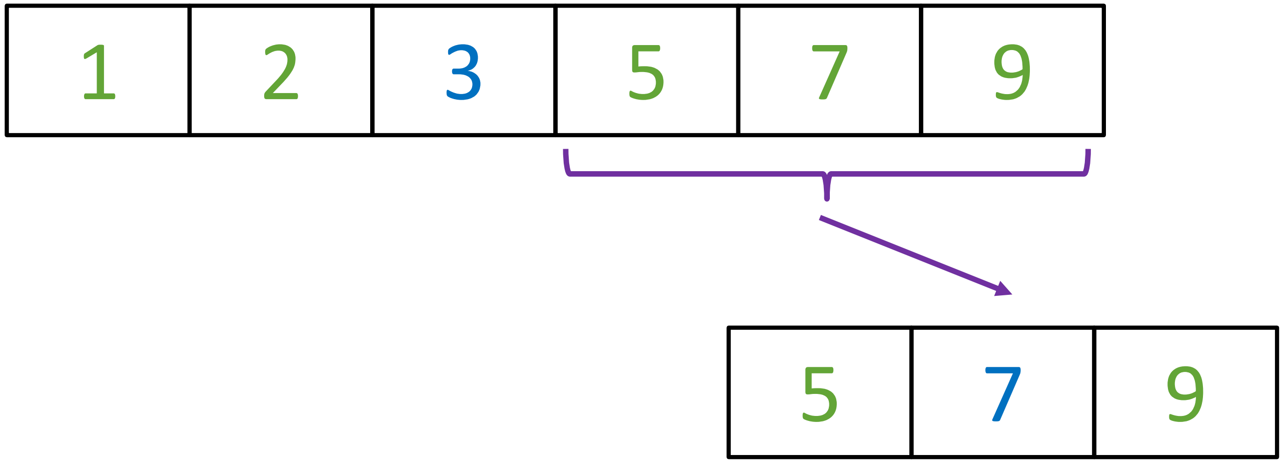
Quick sort example



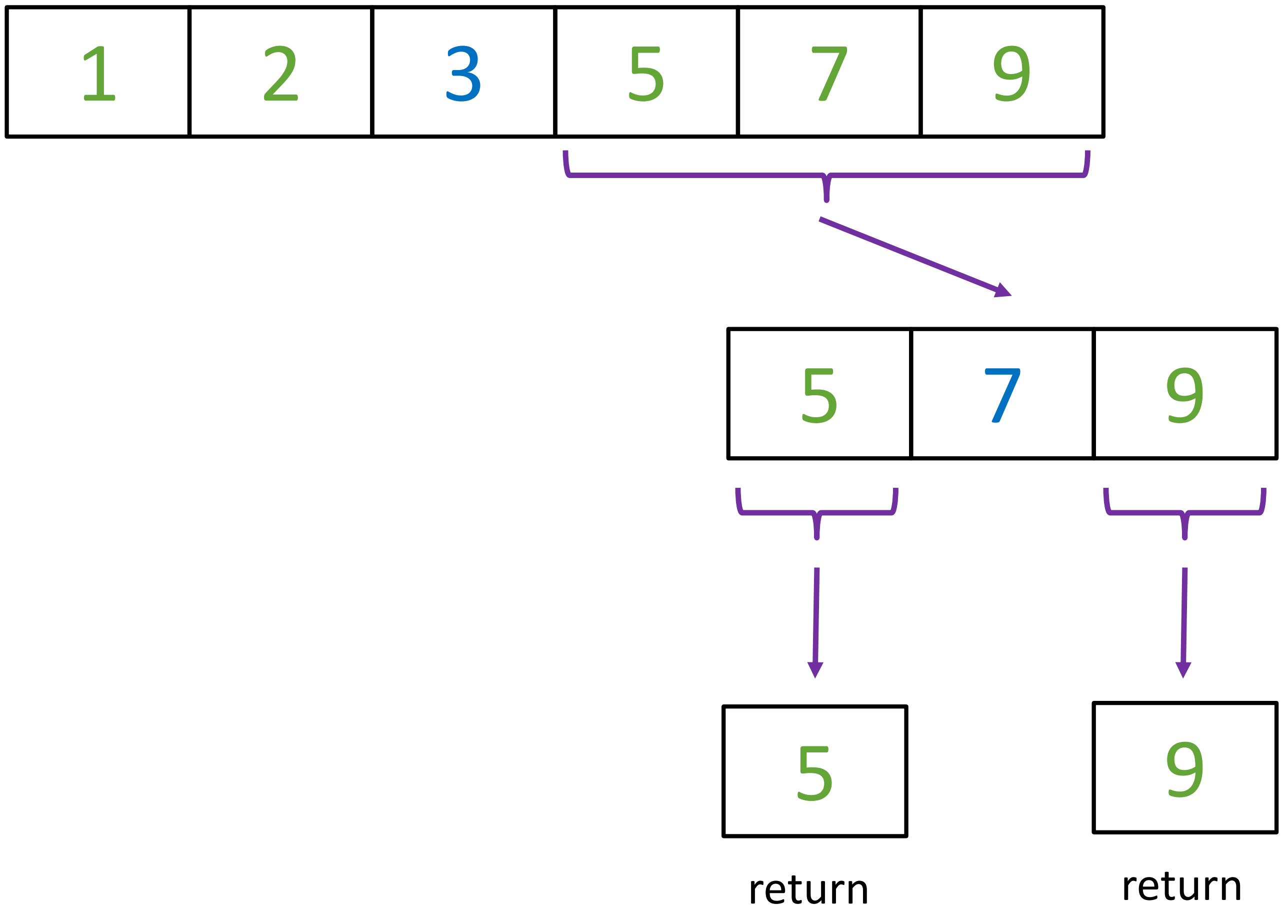
Quick sort example



Quick sort example



Quick sort example



Quick sort example

1	2	3	5	7	9

Done!

Quick sort analysis

- This is also a fast sort, in practice
  - In the order of what, exactly?

Quick sort analysis

- This is also a fast sort, in practice
  - In the order of what, exactly?

- *If* we are lucky with the choices of the pivots, the arrays are divided in half roughly logn times

- On each "level" (you can see the levels in the previous example), we have to partition O(n) elements

- Result: quick sort is in O(nlogn) <u>on average</u>

However…

- If we make bad choices for the pivot, the arrays are not divided in half and we lose the efficiency of quick sort

However…

- If we make bad choices for the pivot, the arrays are not divided in half and we lose the efficiency of quick sort

- Worst-case scenario: choose the first value in the array for the pivot, but the array is already sorted…

However…

- If we make bad choices for the pivot, the arrays are not divided in half and we lose the efficiency of quick sort

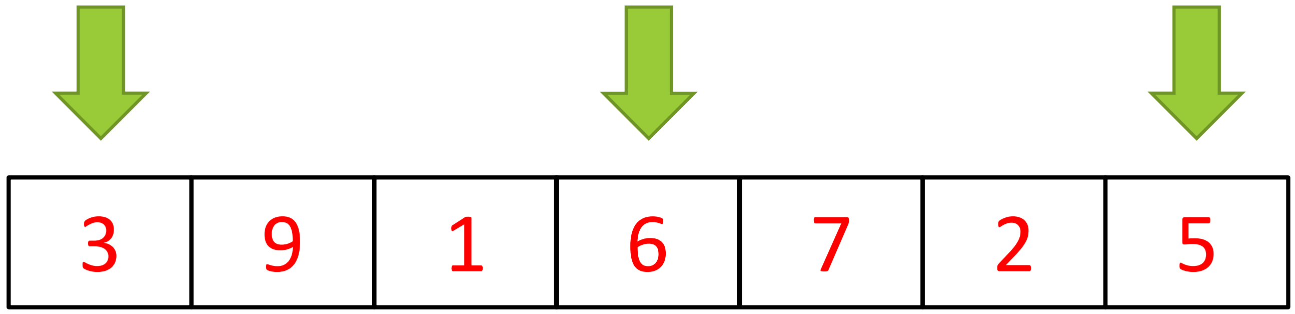
- Worst-case scenario: choose the first value in the array for the pivot, but the array is already sorted…

- Result: we remove only 1 cell at a time, and we have to call quick sort O(n) times (instead of logn)

  - quick sort is in $O(n^2)$ in the worst case

Choosing a better pivot

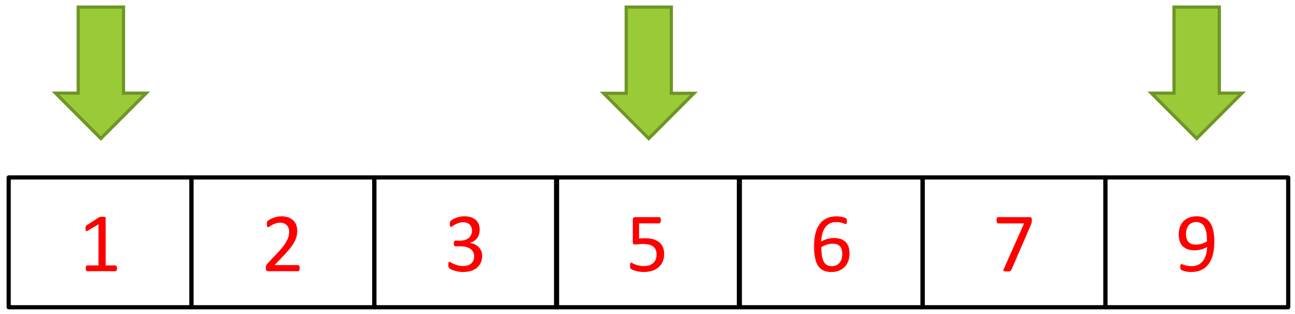
- To avoid this pitfall, there is a simple strategy that allows to avoid this worst case
- Pick the median of three values (beginning, middle, end) for the pivot



Median of (3, 6, 5) = 5

Choosing a better pivot

- Choosing the median works well even when the array is already sorted (or almost sorted)!



Median of (1, 5, 9) = 5

Quick sort implementation

- See SearchSort.java for a simple implementation (without the median)

- Once again, no need to understand and learn all the code, you just need to understand the general idea.

What to remember

- How to roughly analyze the speed of an algorithm
- The names of all the algorithms we have seen and what they are
- The general idea for each of them
- You should be able to easily implement all the algorithms, except the merge sort and quick sort