COMP 1020 - Analyzing algorithms

UNIT 11

Searching Algorithms

- Previously we had seen two different searching algorithms: linear search and binary search.
- We know that the two algorithms have different performance (speed). How could we compare them?
  - We could implement them and time them, but this is dependent on the implementation details, computer system we are using, and test data.
  - Or we could analyze the algorithms.

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

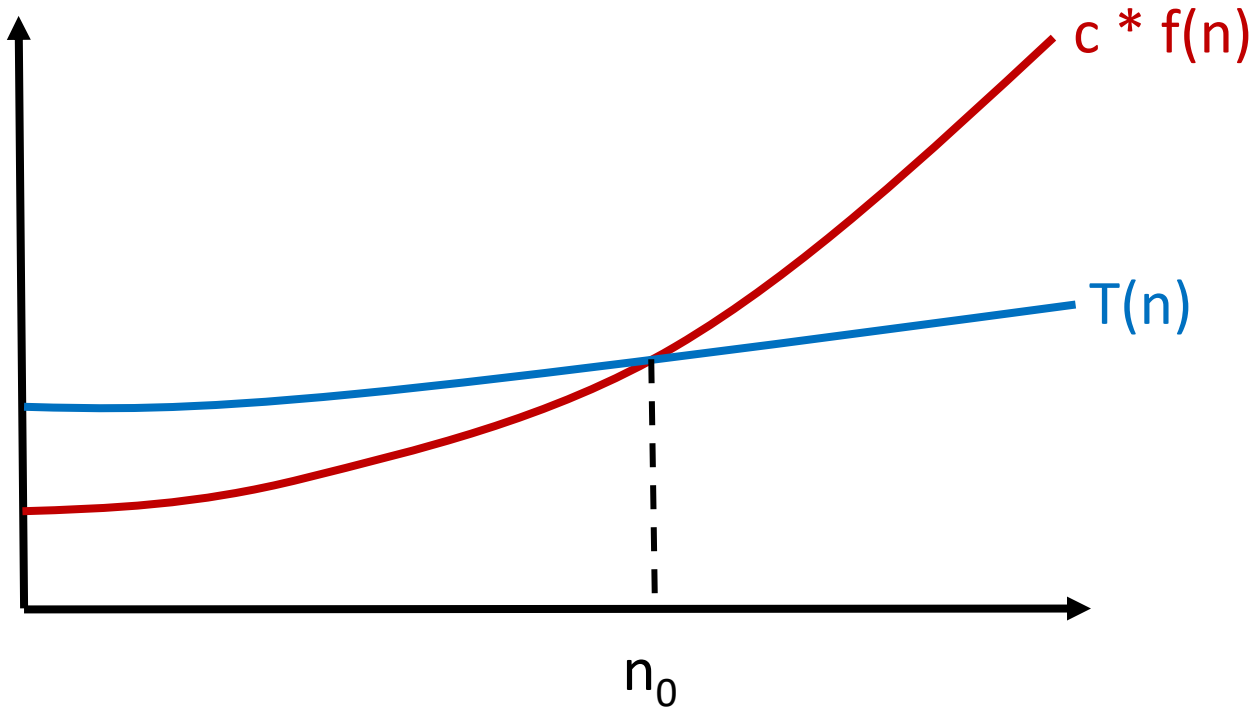
The big-O notation → you can read it as:
the number of operations (or runtime) is in the order of n, or log n, or any function of n

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))

Important 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

Important 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!

Important info: typical bounds

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

Important 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)

Important 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!

What about sorting algorithms?

- We can perform the same kind of analysis on our sorting algorithms.

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

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 - 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

Simple sorting algorithms

- All of our simple sorting algorithms, like insertion, selection, and bubble, are $O(n^2)$.
  - That means that even though some may be faster than others when used on the same data, they are all theoretically equally slow.
  - When the lists get very long (many millions or billions of items), they all become impractical or impossible to use.

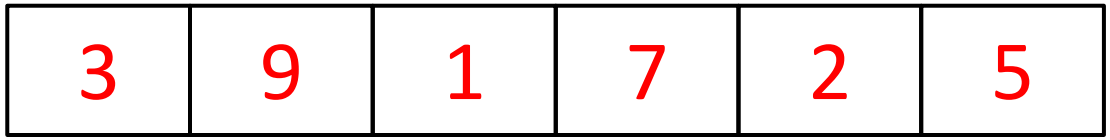
Better sorting algorithms

- To improve on the simple sorts, we need to get rid of at least one of the nested loops through (approximately) all items.
  - Because it's those loops that make them $O(n^2)$.
- We replace the loop with the same kind of "divide and conquer" technique that we used to make the binary search faster than the linear search.

Merge sort

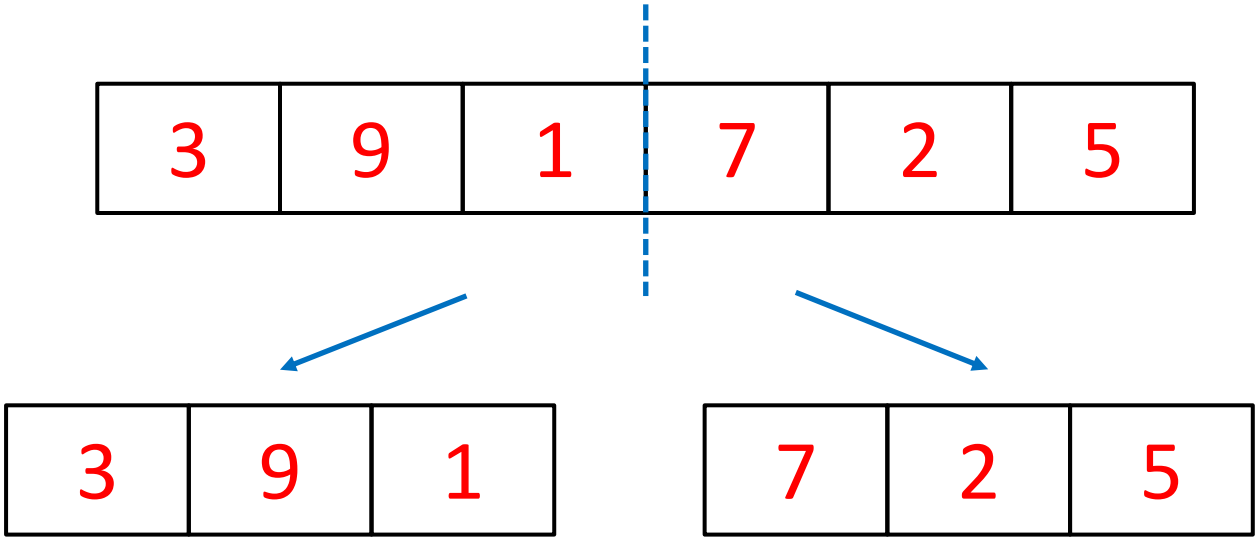
- The basic merge sort algorithm is:
  - Split the array into two small arrays (half each)
  - Sort the two halves (using 2 merge sorts)
    - Recursion! x2!
  - Merge the two sorted halves into a sorted array after the 2 merge sorts have returned

Merge sort example

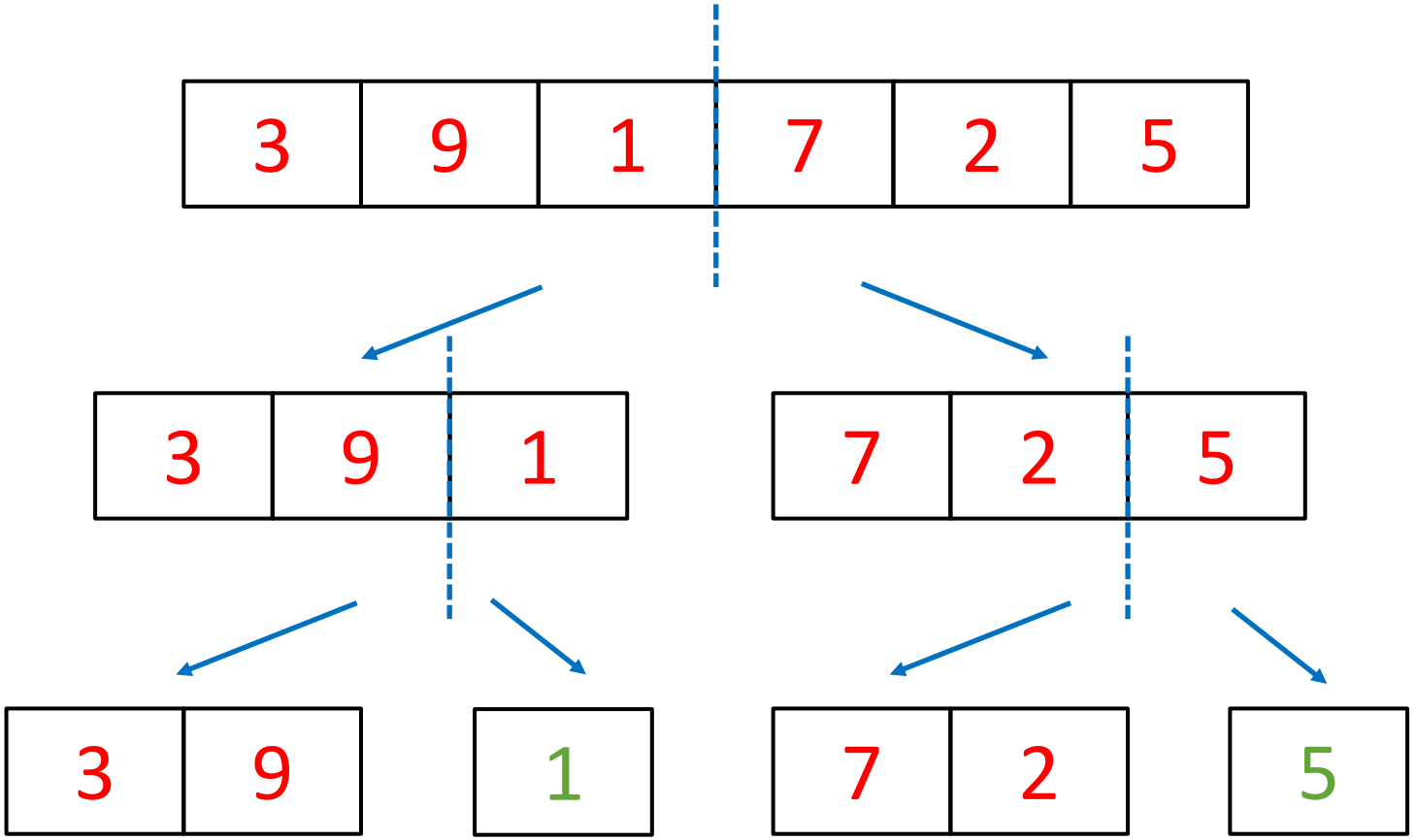


First, the recursive calls will divide the arrays in half until we get to the base case, which is when we get an array with only one element → nothing to sort (an array of one element is sorted by default)

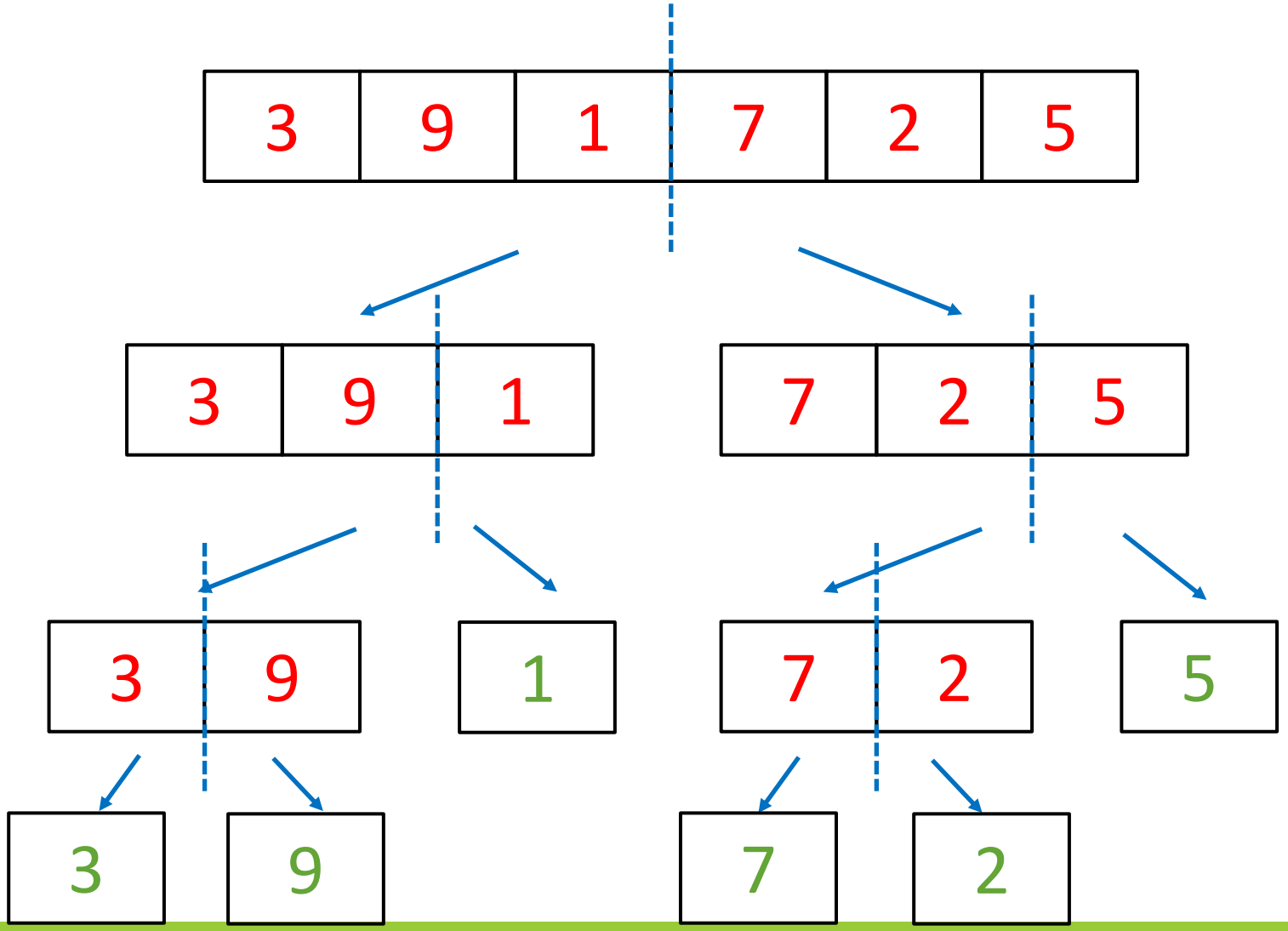
Merge sort example



Merge sort example

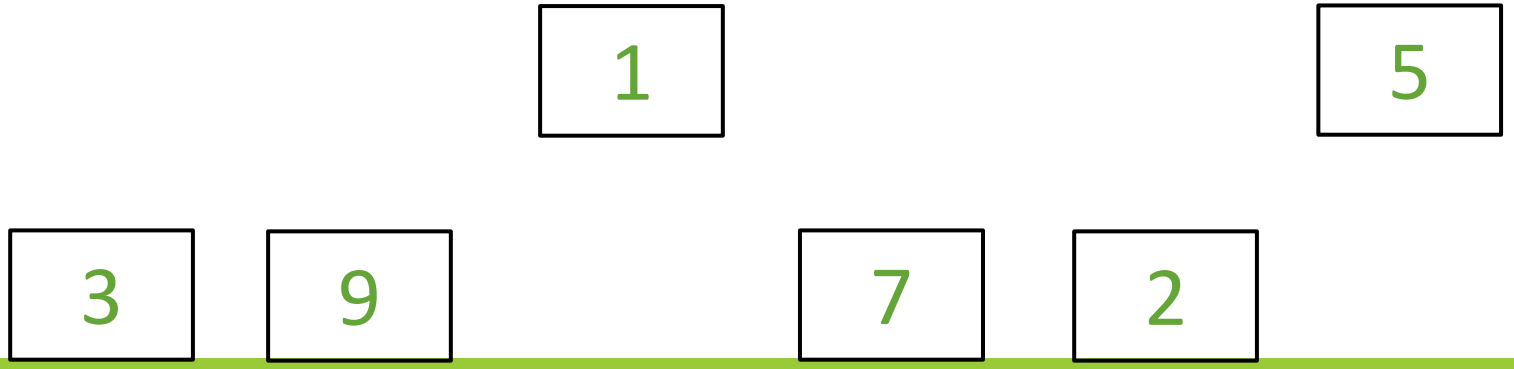


Merge sort example

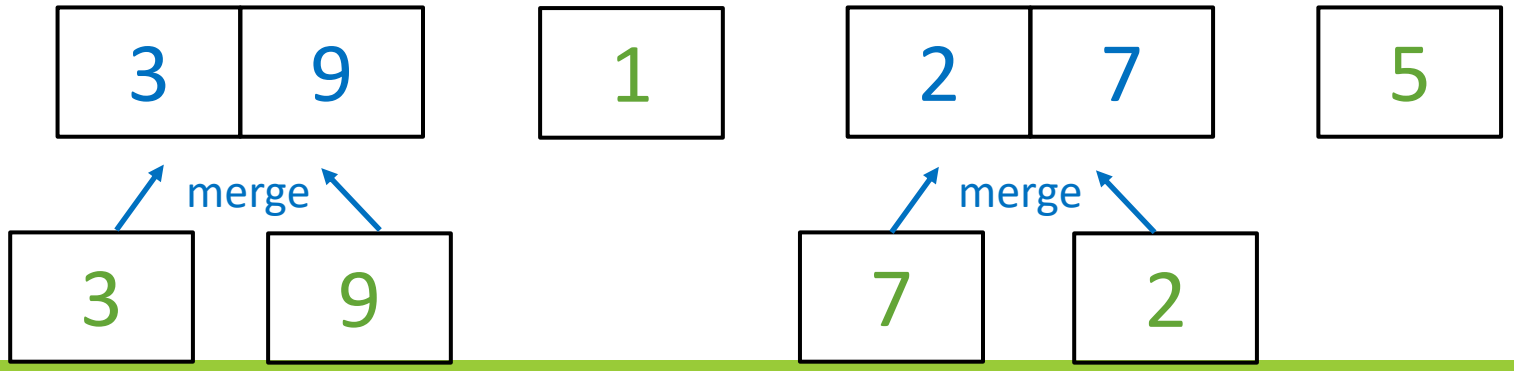


Merge sort example

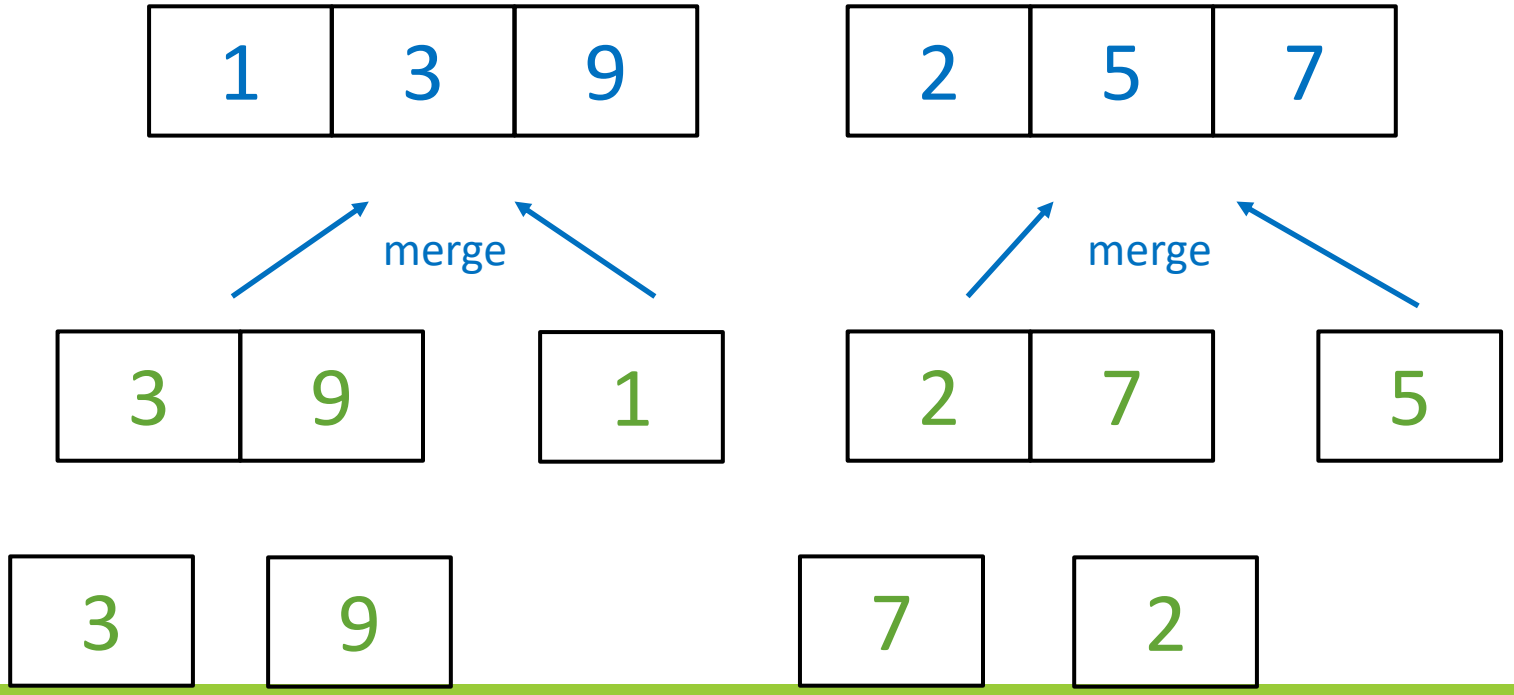
Now, when the recursive calls return, we have to merge the two halves together, in the right order, to create an array that is sorted!



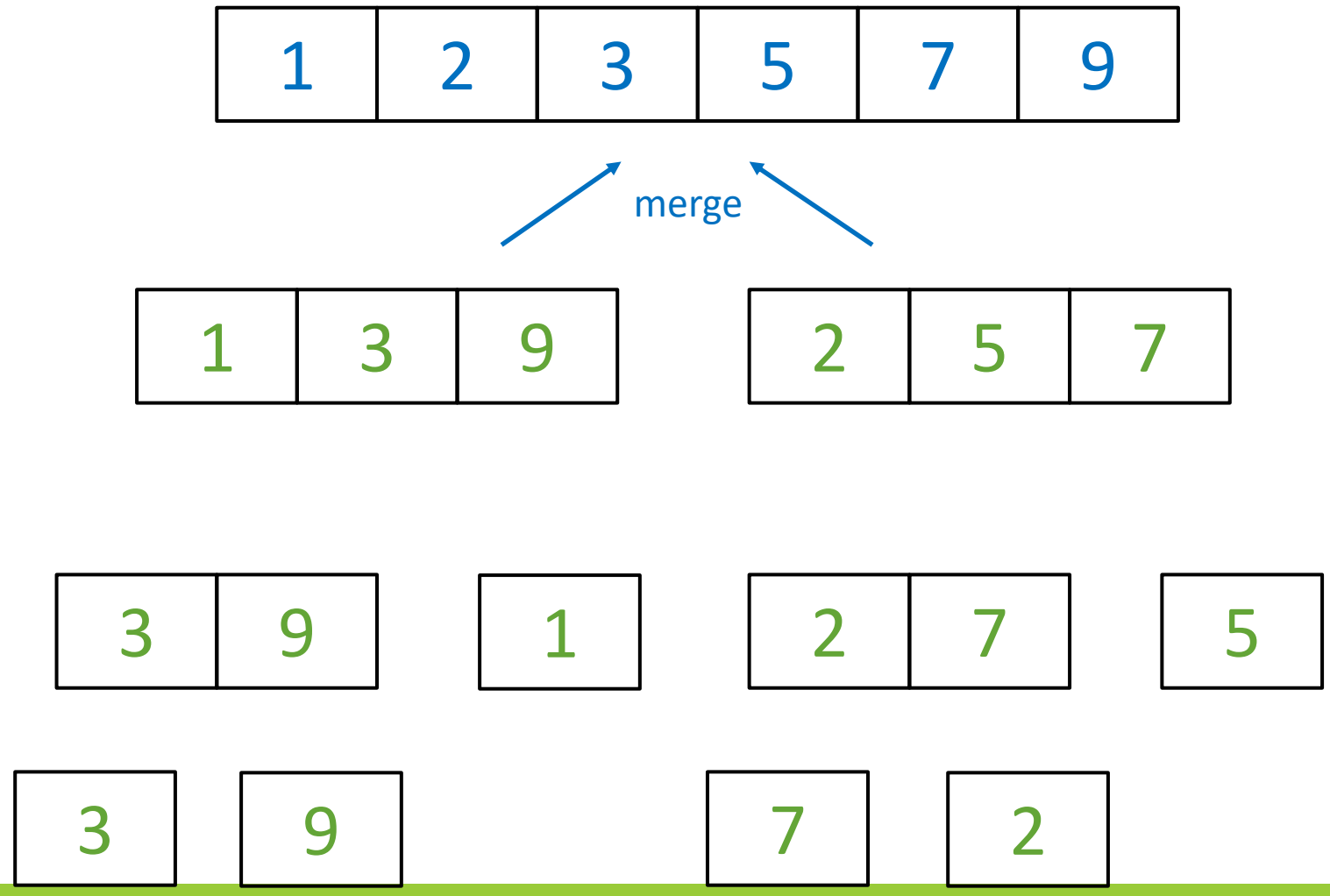
Merge sort example



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?

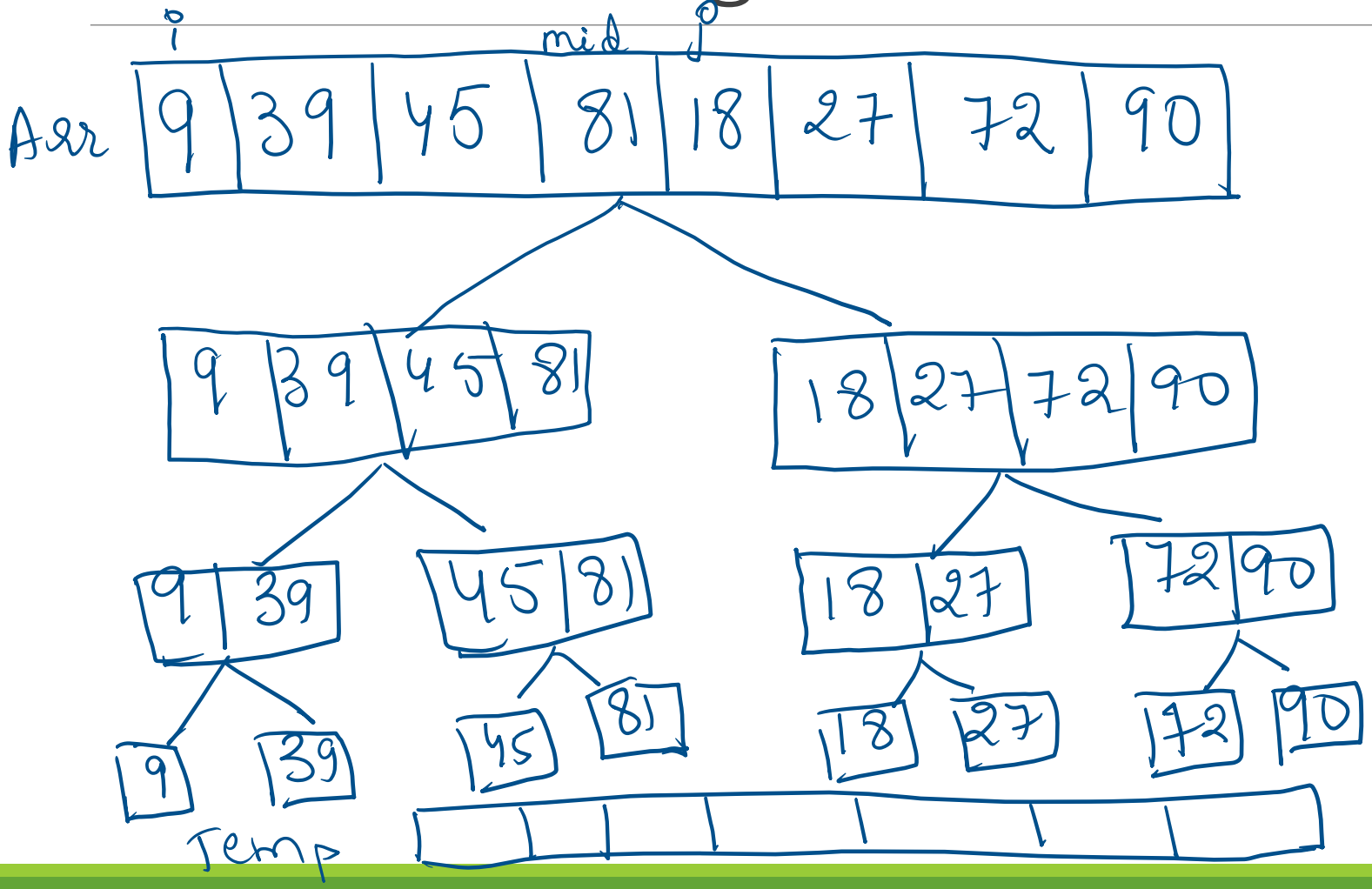
Merge sort analysis

- 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 MergeSort.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



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!

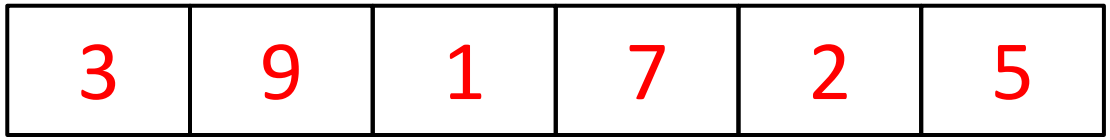
Merge sort

- 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

Quicksort

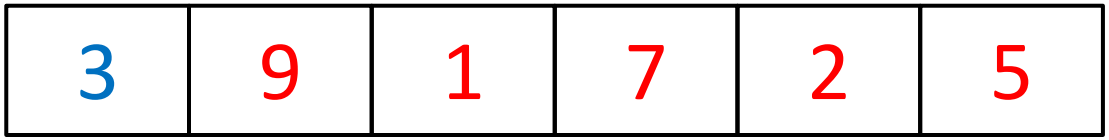
- The basic quick sort algorithm is:
  - Choose a “pivot” value
  - Partition step: place everything that is smaller (resp. bigger) than the pivot on the left (resp. right) side of the array (not necessarily in order at this point in time)
  - Place the pivot in the middle of the left part and right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

Quick sort example



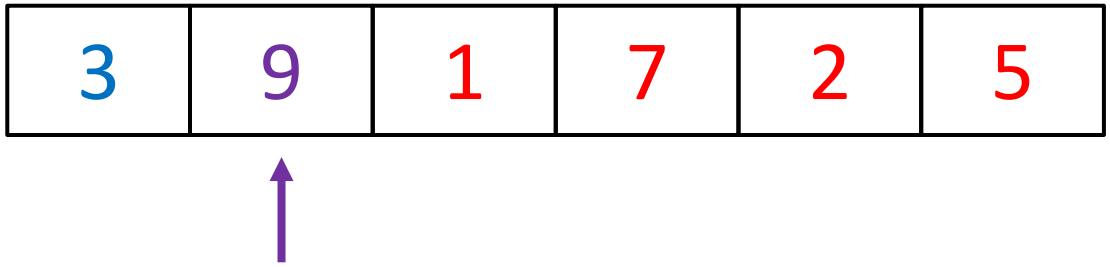
Choose a pivot first. There are many ways you can choose the pivot: the easiest option is to take the first element in the array (note that this easiest option can fail terribly if the array is already sorted).

Quick sort example



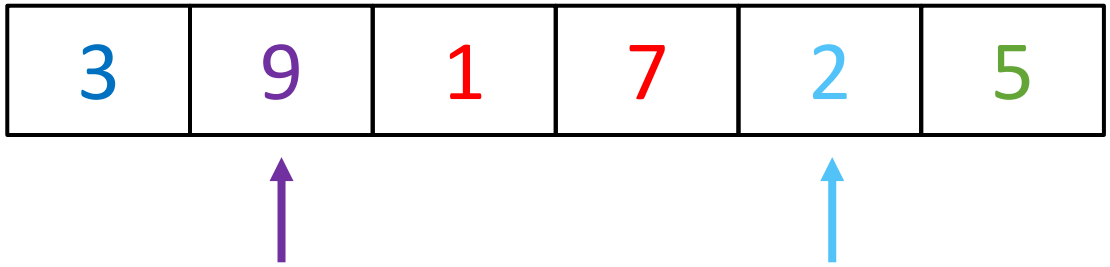
Choose a pivot first. There are many ways you can choose the pivot: the easiest option is to take the first element in the array (note that this easiest option can fail terribly if the array is already sorted).

Quick sort example



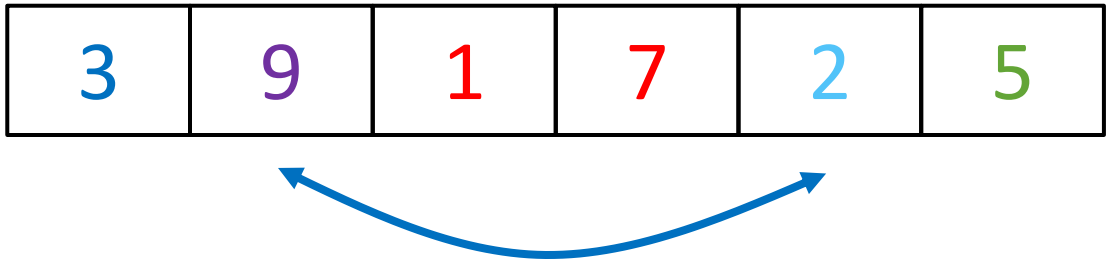
After the pivot, search from the left of the array for the first value that is bigger than the pivot.

Quick sort example



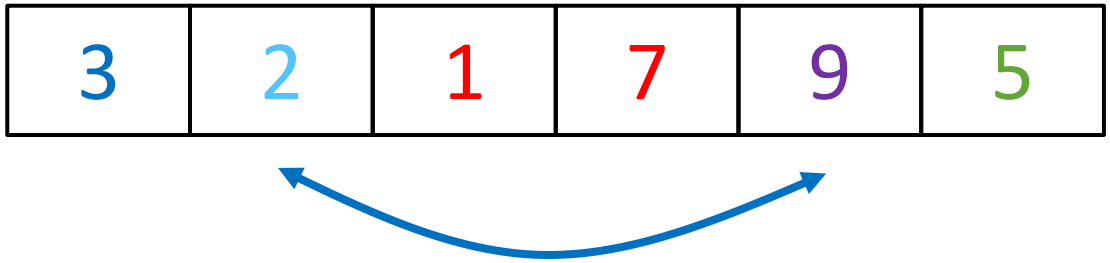
Similarly, search from the right side of the array for the first value that is smaller than the pivot.

Quick sort example



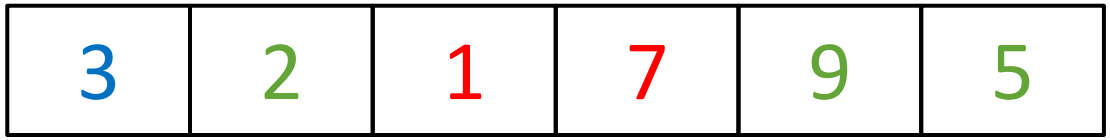
Swap them.

Quick sort example



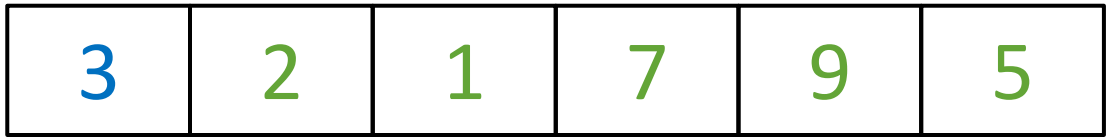
Swap them.

Quick sort example



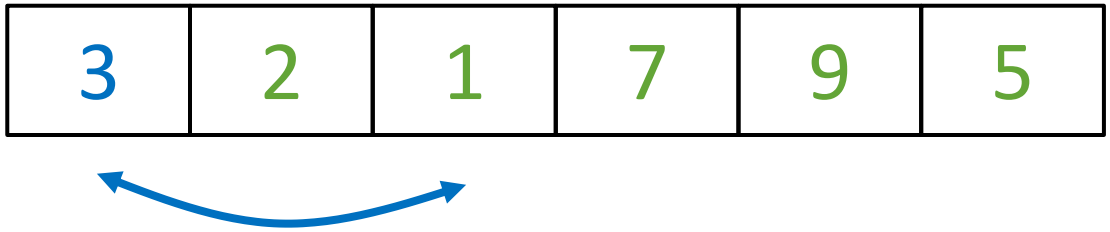
Keep doing that until you meet in the middle.

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.

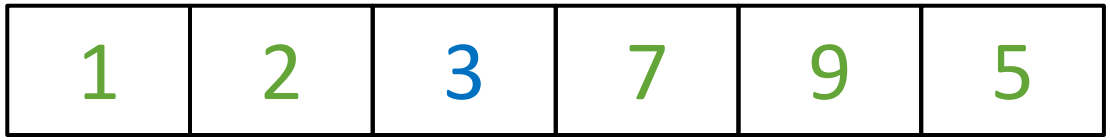
Quick sort example



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).

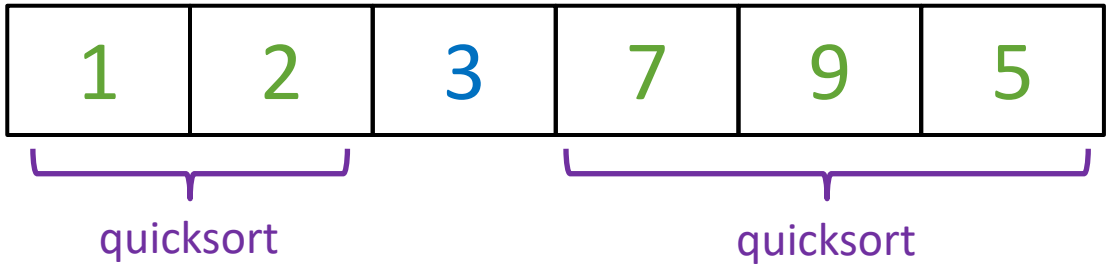
Quick sort example



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

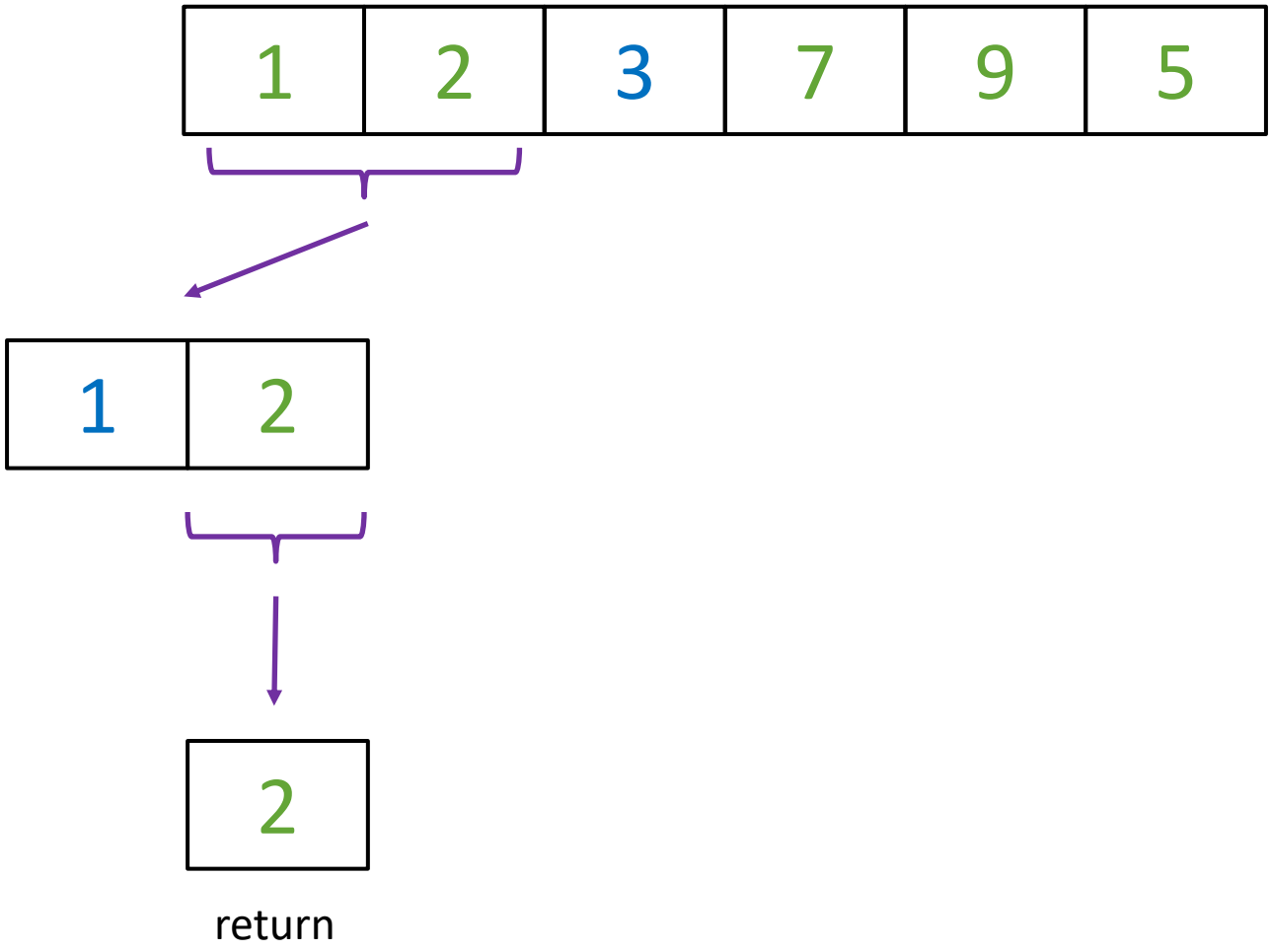
Last step is to move the pivot to the middle (at the end of the half that is smaller).

Quick sort example

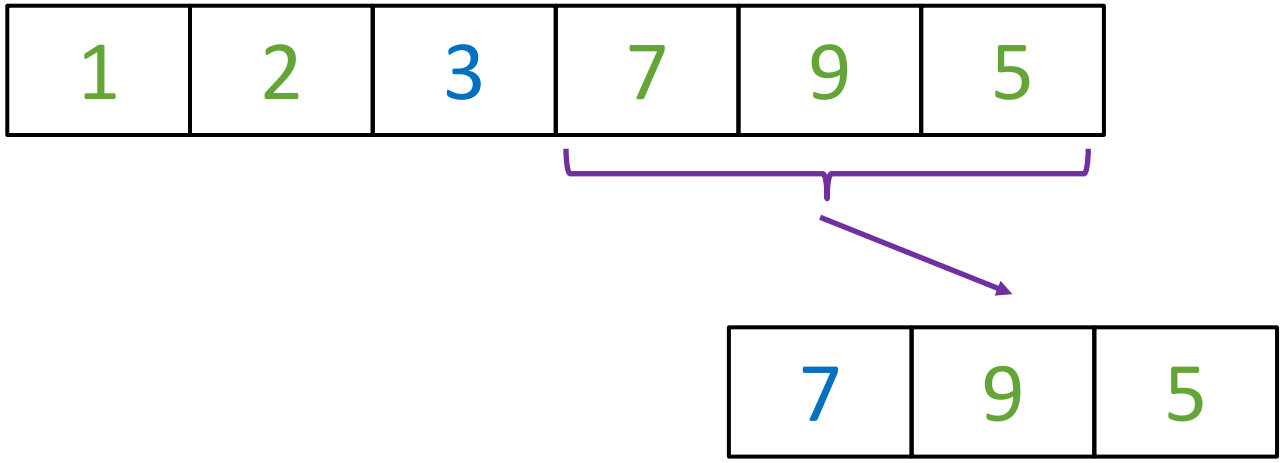


Now we make 2 recursive calls to sort both sides of the pivot (excluding the pivot, because it is now in its final spot).

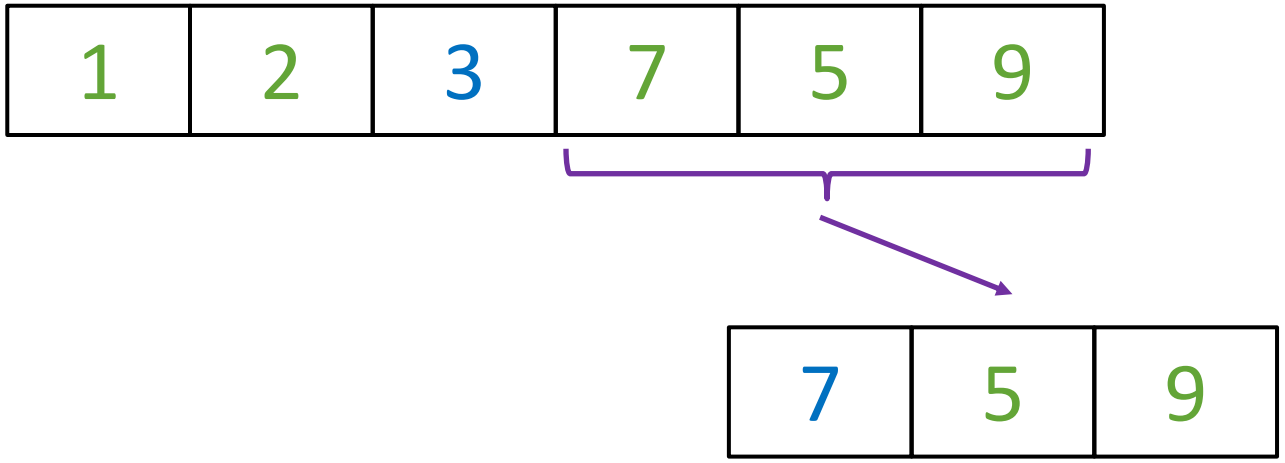
Quick sort example



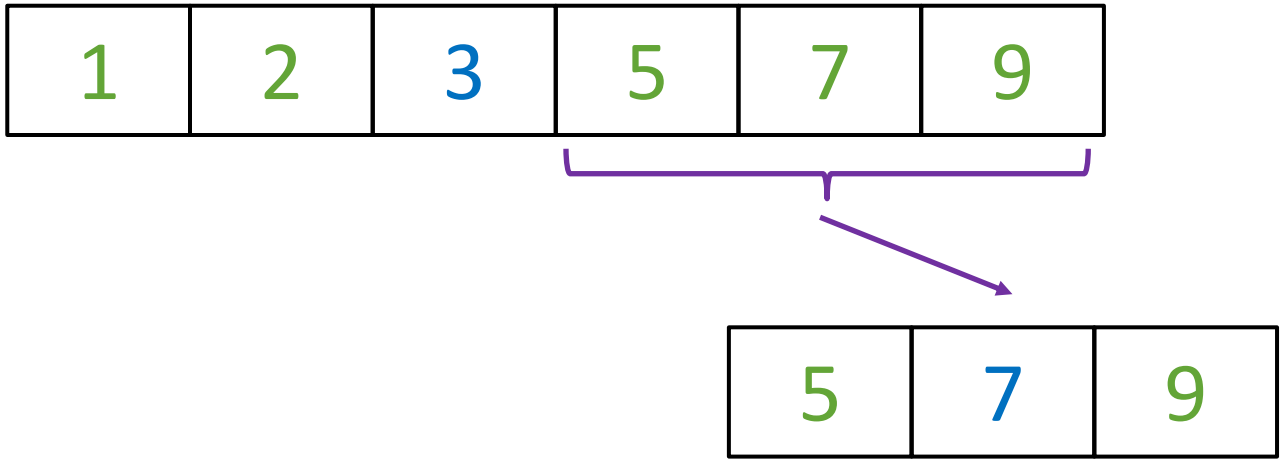
Quick sort example



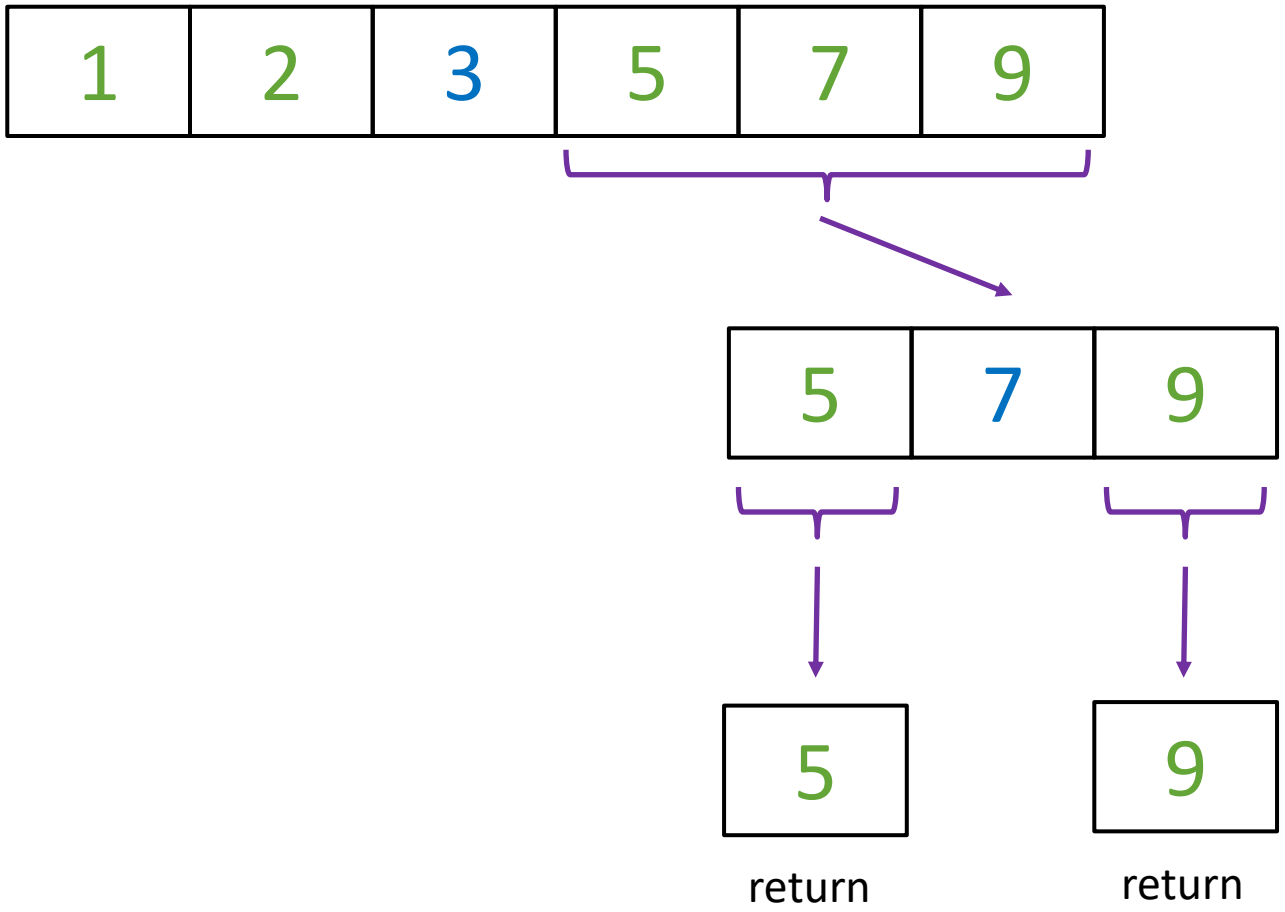
Quick sort example



Quick sort example



Quick sort example



Quick sort example

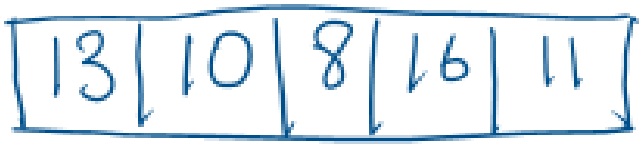
1	2	3	5	7	9

Done!

Quick Sort with pivot as last element

Psuedo-code:

```
Partition(arr,begin,end)
        pivot = arr[end];
        i=begin
        for j=begin to end-1
          if arr[j]<=pivot
                swap arr[i] and arr[j]
                i++
        swap arr[i] and arr[end]
        return i
```



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) on average

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…

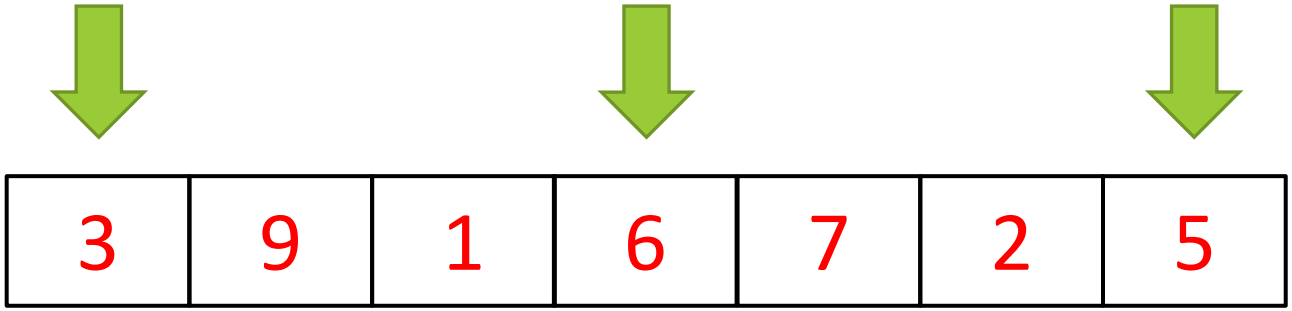
However…

- 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

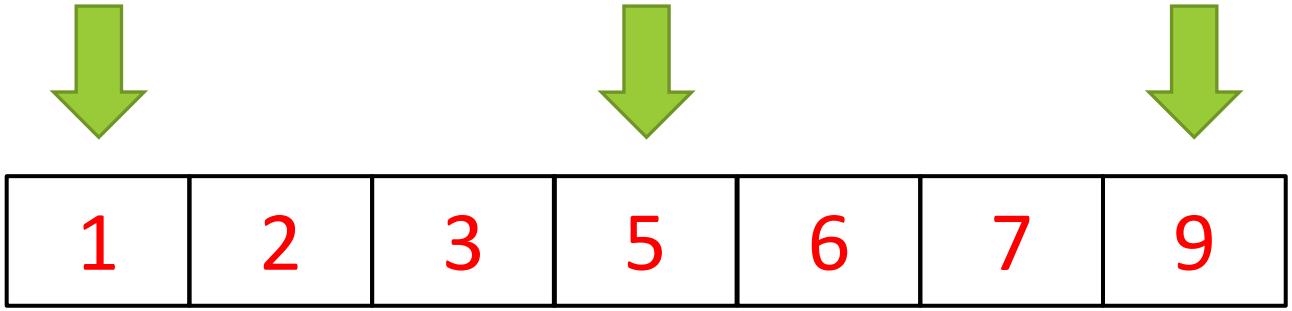
- Pick the median of three values (beginning, middle, end) for the pivot



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 QuickSort.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