

# Quicksort

# Outline

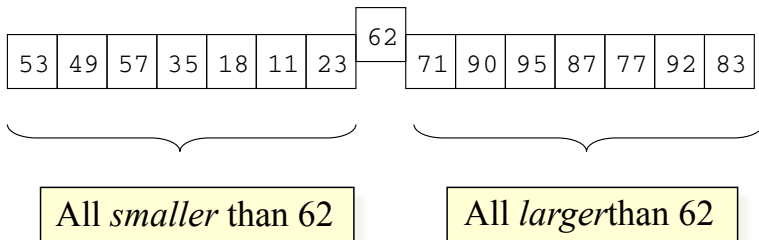
- ▶ What is quicksort?
- ▶ How is it implemented in Java?
- ▶ How well does it perform?

# A recursive sorting algorithm

- ▶ Quicksort is a *recursive* sorting algorithm.
- ▶ Suppose you had to sort the following array with 16 elements:

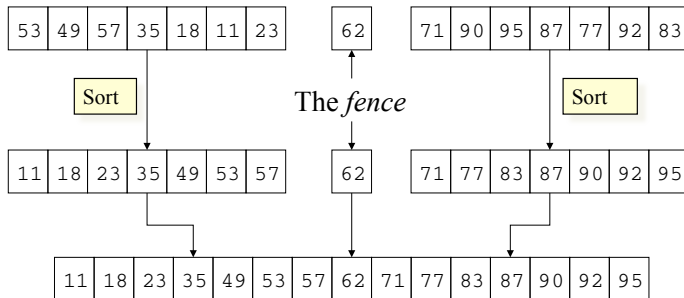
53	49	57	35	18	11	23	62	71	90	95	87	77	92	83
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- ▶ Just before starting, you notice that the array has a very special structure:



# Divide and conquer

- ▶ This means we can now divide the problem into two smaller problems:



- ▶ And the two “half-size” problems take much less than half the time.

## Sorting sub-arrays

- ▶ What if the array is not in this nice form?
- ▶ Then we *put* it into this nice form.

First we choose an element to be the “fence”, and then we adjust the array so that the fence is in the correct position, everything to the left of the fence is smaller than it, and everything to the right of the fence is larger than it.

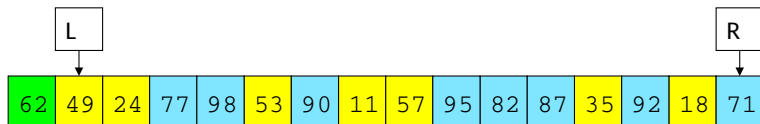
62	49	24	77	98	53	90	11	57	95	82	87	35	92	18	71
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



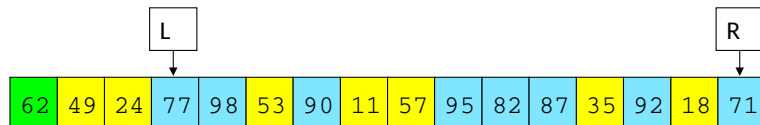
Choose the fence (we explain how later)

## Sorting sub-arrays

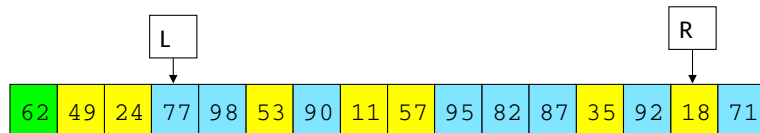
We find out-of-place elements:



Now increase L until reaching an element *bigger* than the fence:

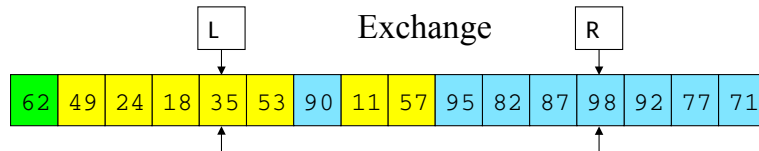
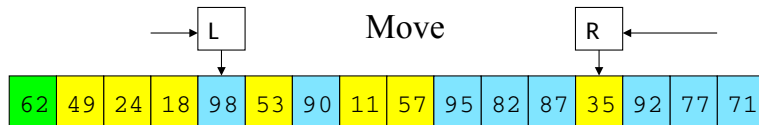
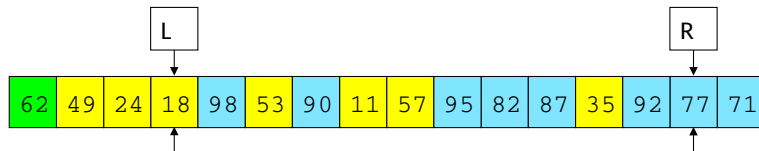


And decrease R until reaching an element *smaller* than the fence:



## Sorting sub-arrays

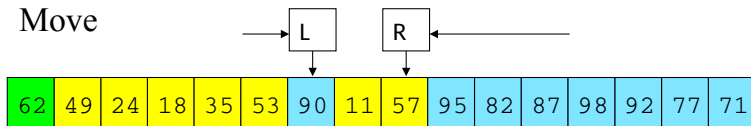
Then swap them over:



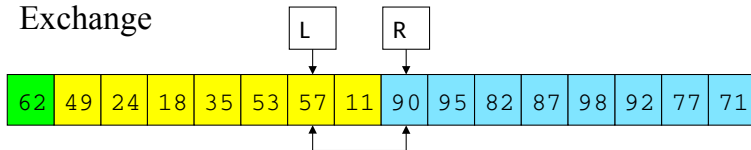
## Sorting sub-arrays

And repeat ...

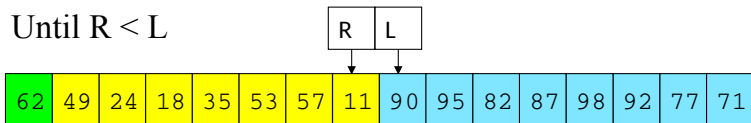
Move



Exchange



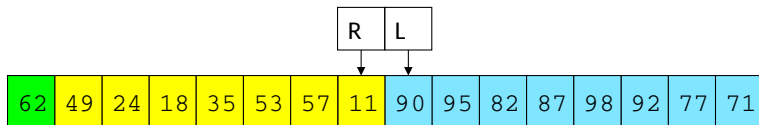
Until  $R < L$



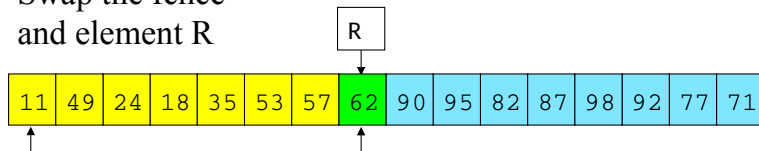


# Terminating

When  $R < L$ , then every element to the right of  $L$  is larger than the fence, and every element (except the fence) to the left of  $L$  is less than the fence.



Swap the fence  
and element R



# Quicksort java code

```
public void partition(int[] a) {  
    int fence = a[0];  
    int left = 1;  
    int right = a.length-1;  
    while (right >= left) {  
        while (left <= right && a[left] <= fence)  
            left++;  
        while (right >= left && a[right] >= fence)  
            right--;  
        if (right > left) {  
            int swap = a[left];  
            a[left] = a[right];  
            a[right] = swap;  
        }  
    }  
    a[0] = a[right];  
    a[right] = fence;  
}
```

## Quicksort java code

```
public void partition(int[] a) {
```

```
    int fence = a[0];  
    int left = 1;  
    int right = a.length-1;
```

```
    while (right >= left) {
```

```
        while (left <= right && a[left] <= fence)  
            left++;  
        while (right >= left && a[right] >= fence)  
            right--;
```

```
        if (right > left) {  
            int swap = a[left];  
            a[left] = a[right];  
            a[right] = swap;  
        }
```

```
    }  
    a[0] = a[right];  
    a[right] = fence;
```

```
}
```

The first element in the array is chosen as the fence and left and right are set up appropriately

These loops increase left and decrease right until they are on swappable elements or meet in the middle

# Quicksort java code

```
public void partition(int[] a) {
```

```
    int fence = a[0];
```

```
    int left = 1;
```

```
    int right = a.length-1;
```

```
    while (right >= left) {
```

```
        while (left <= right && a[left] <= fence)
```

```
            left++;
```

```
        while (right >= left && a[right] >= fence)
```

```
            right--;
```

```
        if (right > left) {
```

```
            int swap = a[left];
```

```
            a[left] = a[right];
```

```
            a[right] = swap;
```

```
        }
```

```
    }
```

```
    a[0] = a[right];
```

```
    a[right] = fence;
```

```
}
```

This swaps the two elements if the two indices have not met in the middle

And when they do meet, the method finishes off by swapping the fence into the correct position

# QuickSort

- ▶ **QuickSort** is a recursive sorting method defined as follows:
- ▶ To sort an array:
  - ▶ Partition the array around some fence
  - ▶ **QuickSort** the elements of array before the fence position
  - ▶ **QuickSort** the elements of array after the fence position
- ▶ This is a recursive definition because we have *used* **QuickSort** in the *definition* of **QuickSort**
- ▶ We have only given the recursive part of the definition – what is the base case?
  - ▶ This is easy – if the array has 0 or 1 elements to be sorted, then we do not need to do anything!

# In-place sorting

**QuickSort** sorts elements *in place*.

```
private static int partition(int[] a, int start, int finish)
{
    int fence = a[start];
    int left = start+1;
    int right = finish;

    // omitted code is identical to before

    a[start] = a[right];
    a[right] = fence;

    return right;
}
```

We just treat the elements  
a[start]..a[finish] as the array to  
be partitioned, and ignore the rest

The method returns the correct  
position of the fence after it has  
been located

## QuickSort Java code

The actual quickSort code is as follows:

```
private static void quickSort(int[] a, int start, int finish)
{
    if (finish-start > 0) {
        int fence_position = partition(a,start,finish);
        quickSort(a,start,fence_position-1);
        quickSort(a,fence_position+1,finish);
    }
}
```

Partition

Sort left

Sort right

```
public static void quickSort(int[] a) {
    quickSort(a,0,a.length-1);
}
```

The public method  
should be as simple  
to use as possible

## QuickSort Java code

- ▶ What happens when we call `quickSort(a)` on an array `a`?
- ▶ The call `quickSort(a)`:

a

62	49	24	77	98	53	90	11	57	95	82	87	35	92	18	71
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

First calls `partition(a, 0, 15)`, which changes `a` to

11	49	24	18	35	53	57	62	90	95	82	87	98	92	77	71
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

and returns 7 as the fence position

`quickSort(a, 0, 6)`  
sorts the left half

`quickSort(a, 8, 15)`  
sorts the right half

11	49	24	18	35	53	57	62	90	95	82	87	98	92	77	71
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



## Comments on QuickSort

- ▶ Works well when the fence position ends up being roughly half way through the array, as the two sub-problems are then about equal in size
  - ▶ If our fence were *exactly* in the middle every time, then it would take  $\log_2 n$  recursive calls to sort the array.
- ▶ Our choice of fence in this code – position 0 – means that the worst case for QuickSort is when the array is *already ordered*.
  - ▶ If the array is already ordered, and we choose position 0 for our fence –  
then the items to the *left* of the fence will be no items at all, and the items to the *right* of the fence will be *all* the other items.
  - ▶ And when we do a recursive call on the “right-hand” portion, we have only reduced its size by 1;
  - ▶ So the number of calls will be proportional to  $n$ .

## Comments on QuickSort

- ▶ It turns out that in the *worst* case, QuickSort has complexity  $O(n^2)$ , like insertion sort.
- ▶ But if the fence-post we choose is near the middle of the array values, then the *average* number of comparisons QuickSort makes is  $n \log_2 n$ .
- ▶ So, in practice, it pays to put in a little effort to ensure our fence-post value *is* towards the middle of the values.
- ▶ Some possible ways of doing so:
  - ▶ Pick the fence randomly
  - ▶ Randomly permute the array before sorting with QuickSort
  - ▶ Sample a small portion of the array, and choose the median as fence  
e.g. take the median of the first, last and mid-point values.

# Java library

- ▶ You should never need to write QuickSort yourself in Java because the Java library includes a number of optimized methods for sorting.
  - ▶ For example: `Arrays.sort` and `Collections.sort(list)`
- ▶ We will discuss the Java Collections API later.

# Summary

- ▶ The Quicksort algorithm is efficient,  $O(n \log n)$  – *on average* – because it breaks a large list into smaller ones and sorts those.
- ▶ However, Quicksort is not always the best solution. For example, its worst case performance is  $O(n^2)$  when the list is already sorted.