RECURSION

WEEK 2 PRESENTATION 3

MATHEMATICAL INDUCTION

- Thankfully, maths come to the rescue to solve these parenthesis; not maths itself, but a mathematical method which is very closely related to what we'll do.
- This closely related mathematical method is induction, a very clever way of proving theorems.
- More in discrete maths

MATHEMATICAL INDUCTION

The sum of the n first odd integers is equal to n^2.

Proof outline

- - Obvious for 1
  - If it's true for n, it's true for n+1
- Therefore it's true for any integer value.

MATHEMATICAL INDUCTION

The sum of the n first odd integers is equal to n2.

Proof

It is obvious for 0 and 1 (Base case).

Suppose it is true for n. Then,

$1 + 3 + \ldots + (2n - 1) = n^2$.

We will try to show that if the statement is true for n, then it follows from that it is true for n+1

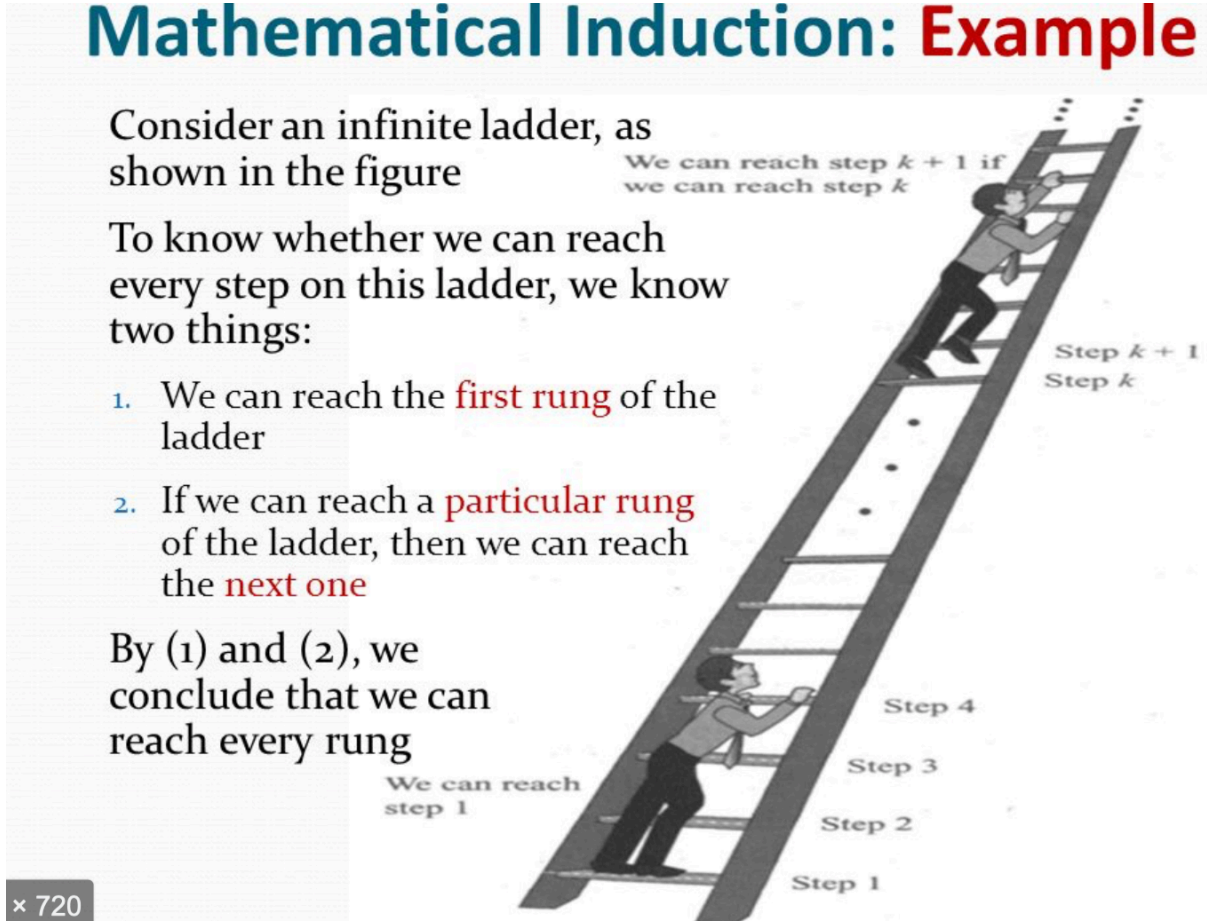
$1 + 3 + \ldots + (2n - 1) + (2n + 1)$ Next odd integer after the nth one

Rearranging gives: $n^2 + (2n+1) = (n+1)^2$

QED

HOW **MATHEMATICAL INDUCTION** WORKS

- Base case
- Link between *n* and *n* + *1* (the truth of n+1 follows from the truth of n)

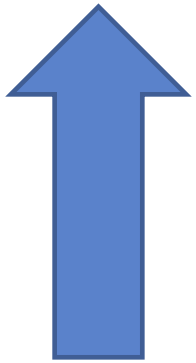


RECURSION

- Is a related (although not identical) approach in programming
- Is also based on a link between n and n+ 1 and a trivial case, but it works in reverse order

MATHEMATICAL INDUCTION

- Link between n and n+1



- Base case

Mathematical induction goes from the trivial case towards infinity.

RECURSION

- Link between n and n+1

- Base case

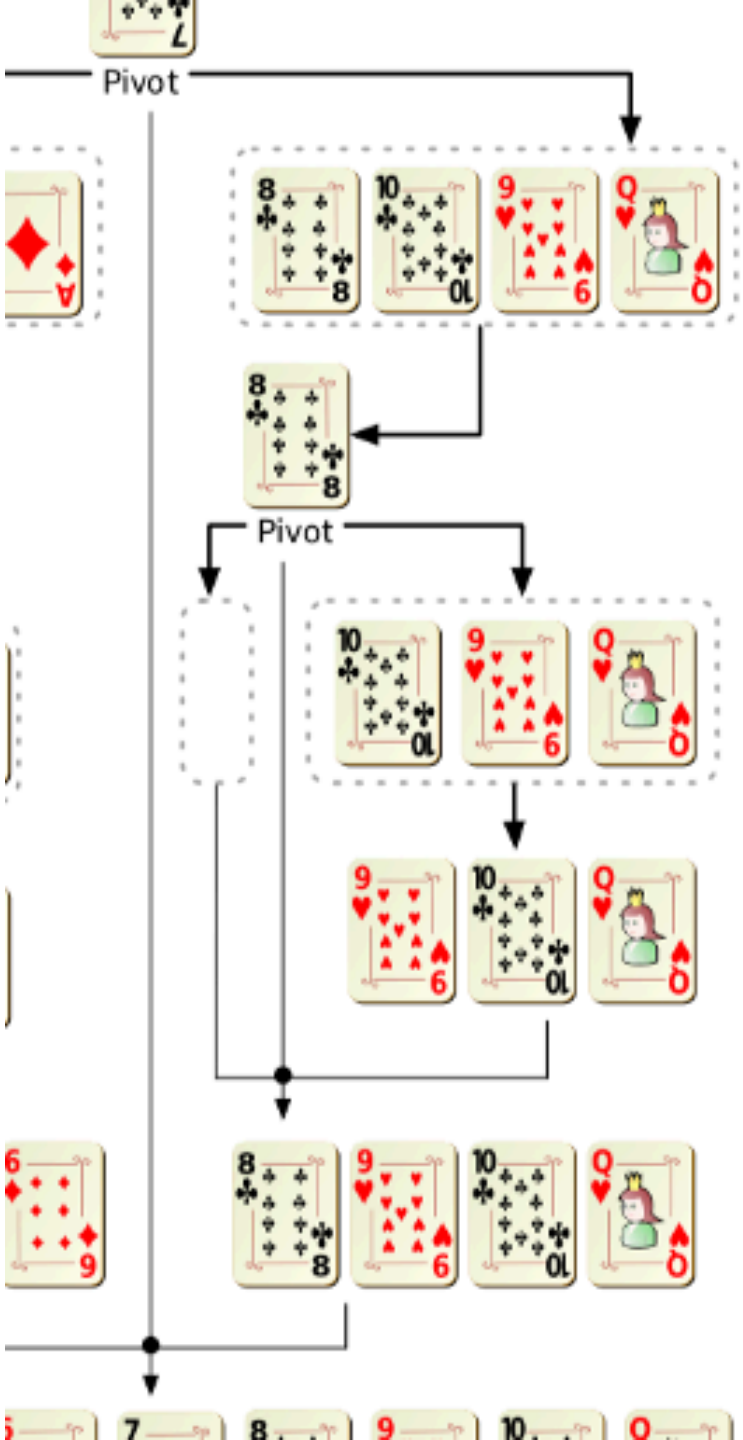
Recursion, which we'll use for the Quick-Sort, works by identifying a trivial case, then assuming that we can solve a problem at level n-1 and expressing the solution to the problem at level n as a function of the n-1 level solution. Once the trivial case is found it works the solution back to level n.

RECURSION

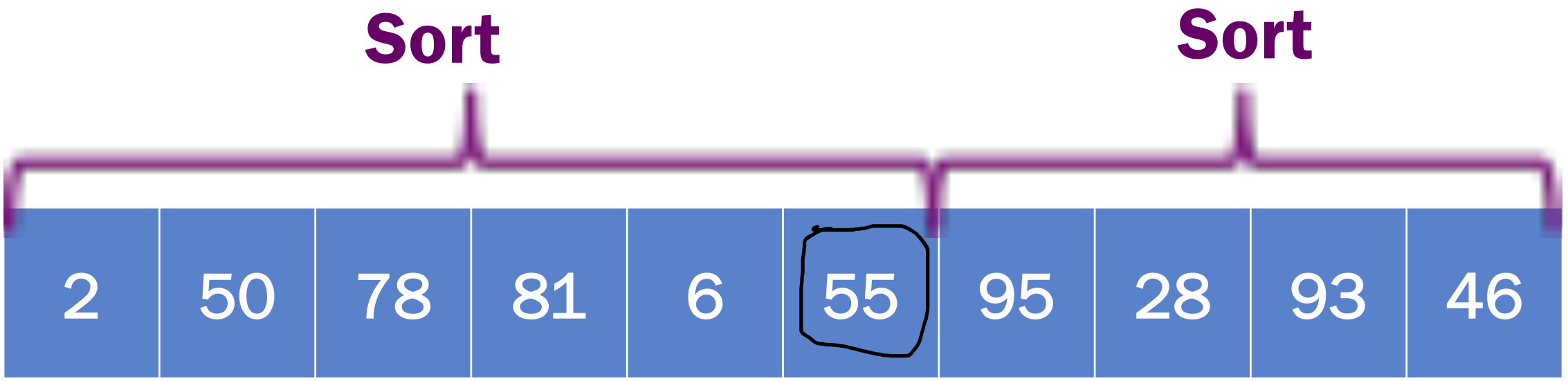
DEF: A FUNCTION CONTAINS A CALL TO ITSELF (WITH OTHER PARAMETERS).

Recursion is characterized by functions that contain calls to themselves, with different parameters corresponding to a smaller problem.

- Of course at some point the function must reach a very easy case and no longer call itself! There will necessarily be a condition in the function to stop the recursion.
- We also call this the base case.
- With cards we said this was the case with *less than* 4 cards.



With a recursive function you need to define a base case...




```
static void quickSort(int[] arr, int first, int last) {
```

BASE CASE: arrays with 1 value are easy to sort

RECURSIVE QUICKSORT IMPLEMENTATION

- If there is 0 or 1 value in the array, nothing to do. If there are two values in the array we just check that they are in order and swap them if not.

```
static void quickSort(int[] arr, int first, int last) {
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first) {
        case 1: // base case
            if (arr[last] < arr[first]) {
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
```

RECURSIVE QUICKSORT IMPLEMENTATION

In the general case, we place the pivot, then call the function again for sorting the two subsets of the original array.

And THAT'S ALL!

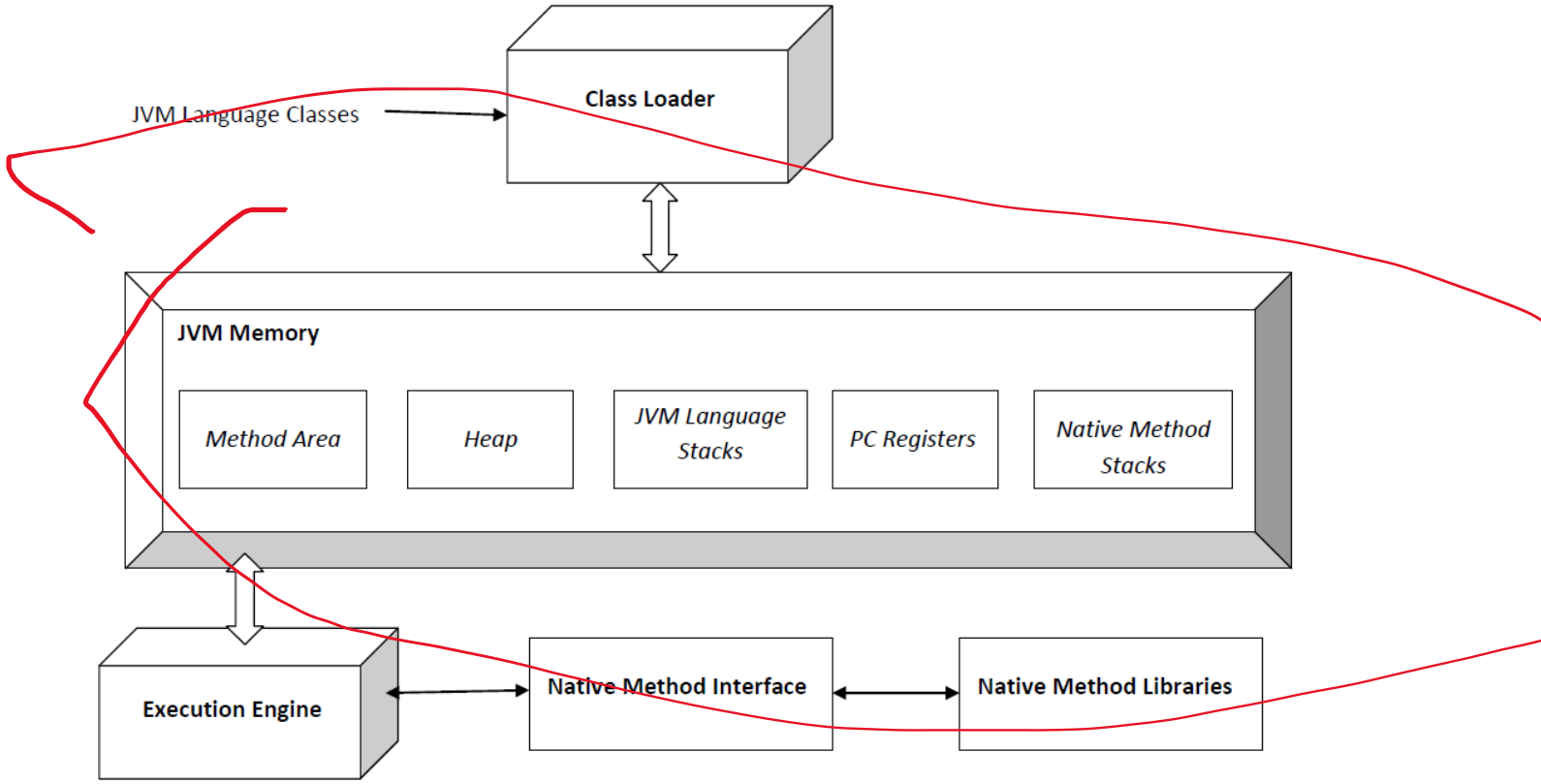
```
static void quickSort(int[] arr,
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first)
        case 1: // base case
            if (arr[last] < arr[
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
        default:
            pivotPos = placePivot(arr, first, last);
            quickSort(arr, first, pivotPos-1); // sort left
            quickSort(arr, pivotPos+1, last); // sort right
            break;
            }
    }
}
```

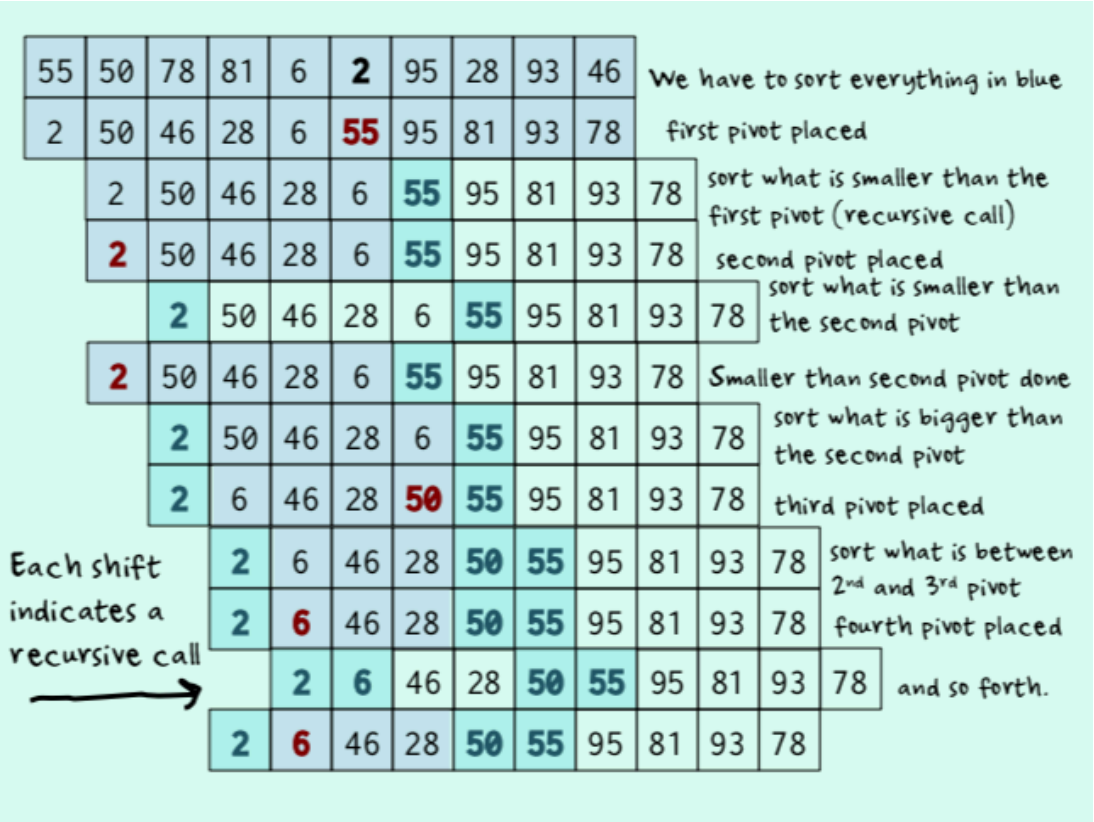
The algorithm can be improved (clever choice of pivot, sorting what is smaller first, etc ...).

WHY DOES IT WORK?

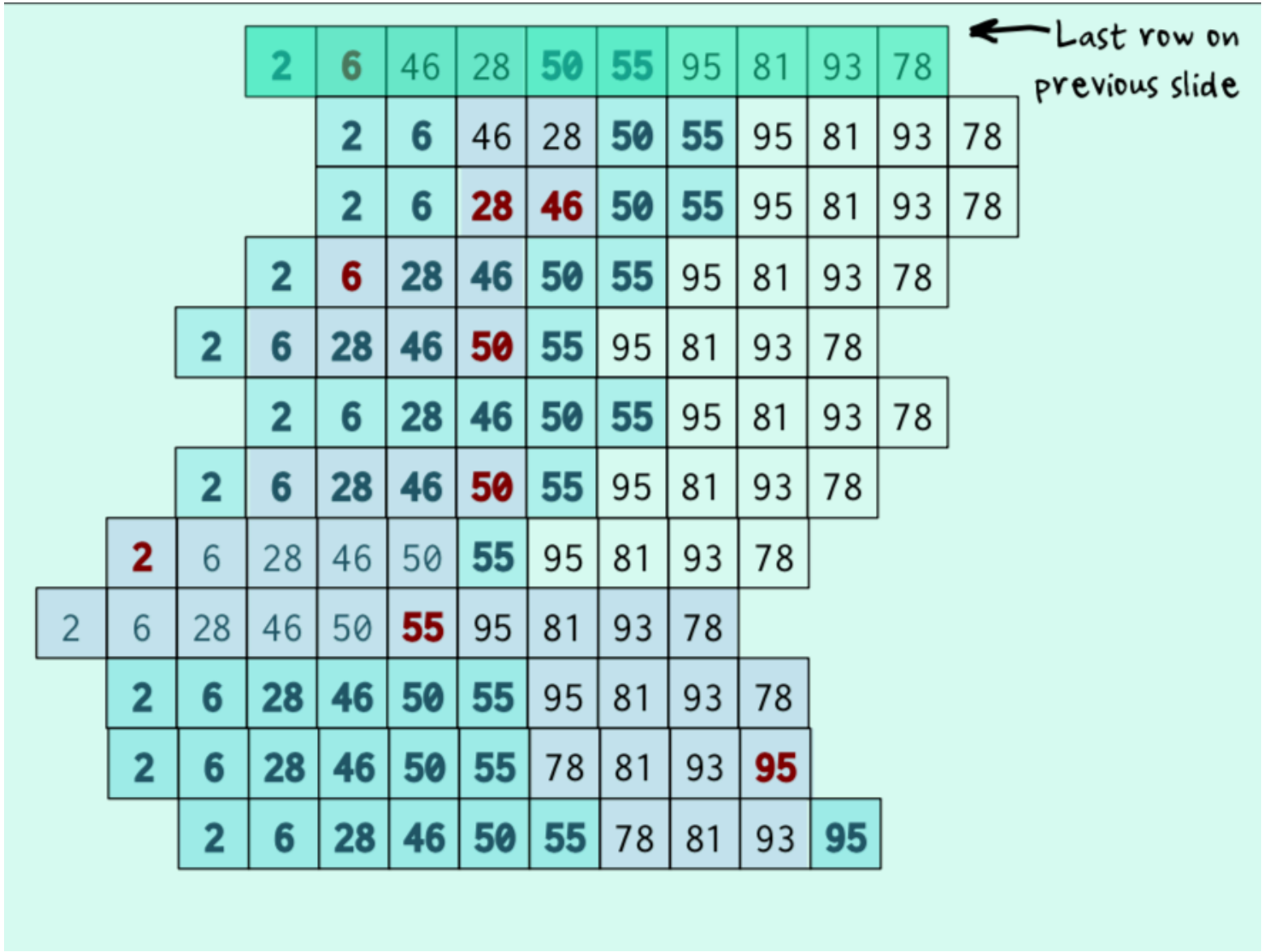
The stack!

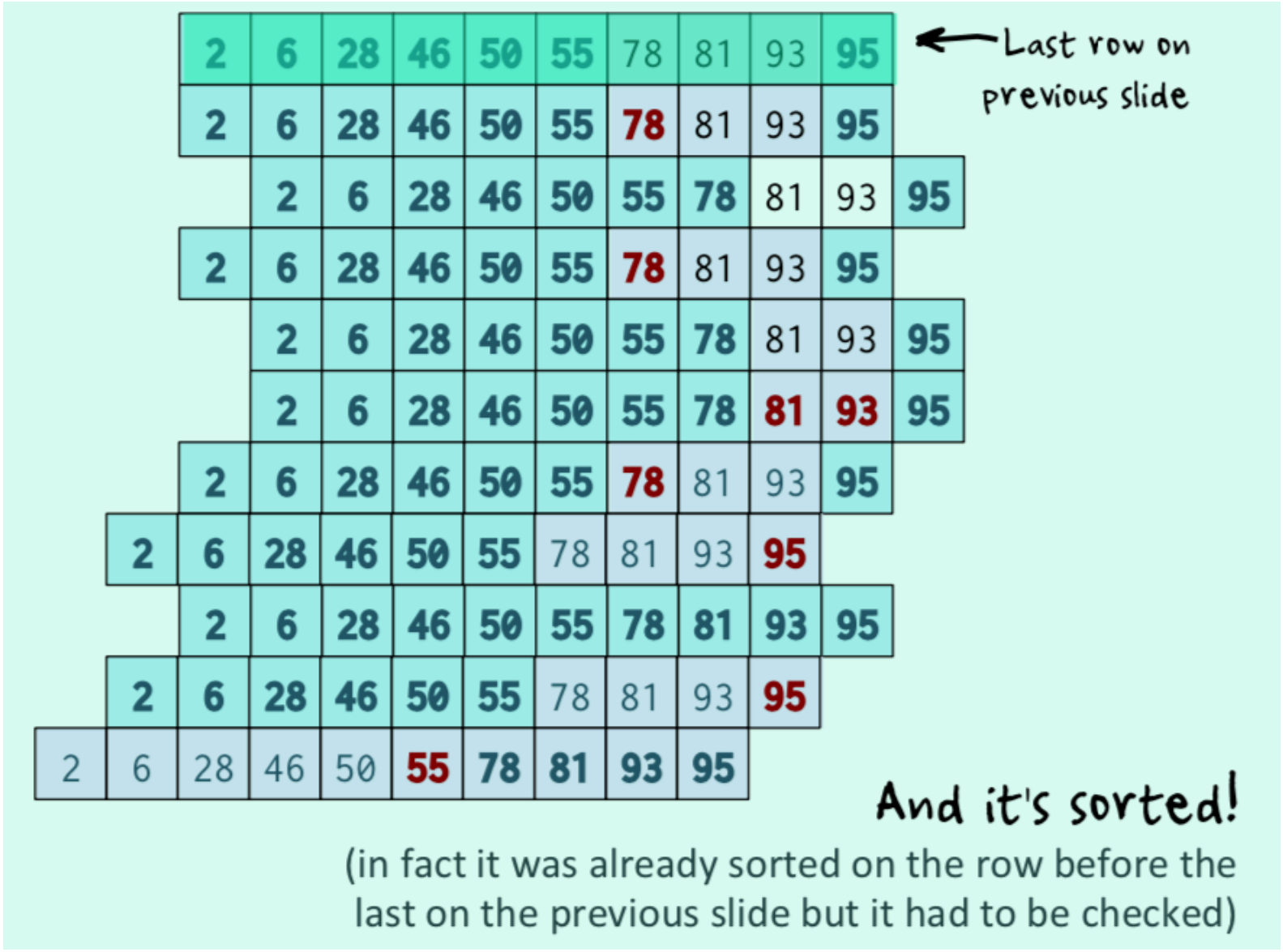
Because of the stack mechanism used by computers when they call functions, operations that have to be performed accumulate in the stack, and are popped out of the stack when done. It's the stack that keeps track of everything. What occurs is usually fairly complicated, but the program that you write is simple.






```java
static void quickSort(int[] arr, int first, int last) {
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first) {
        case 1: // base case
            if (arr[last] < arr[first]) {
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
        default:
            pivotPos = placePivot(arr, first, last);
            quickSort(arr, first, pivotPos-1); // sort left
            quickSort(arr, pivotPos+1, last); // sort right
            break;
            }
    }
}
```


Last row on previous slide
2 6 46 28 50 55 95 81 93 78
2 6 46 28 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95


2 6 28 46 50 55 78 81 93 95
Last row on previous slide
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
And it's sorted!
(in fact it was already sorted on the row before the last on the previous slide but it had to be checked)

RECURSIVE FACTORIAL FUNCTION

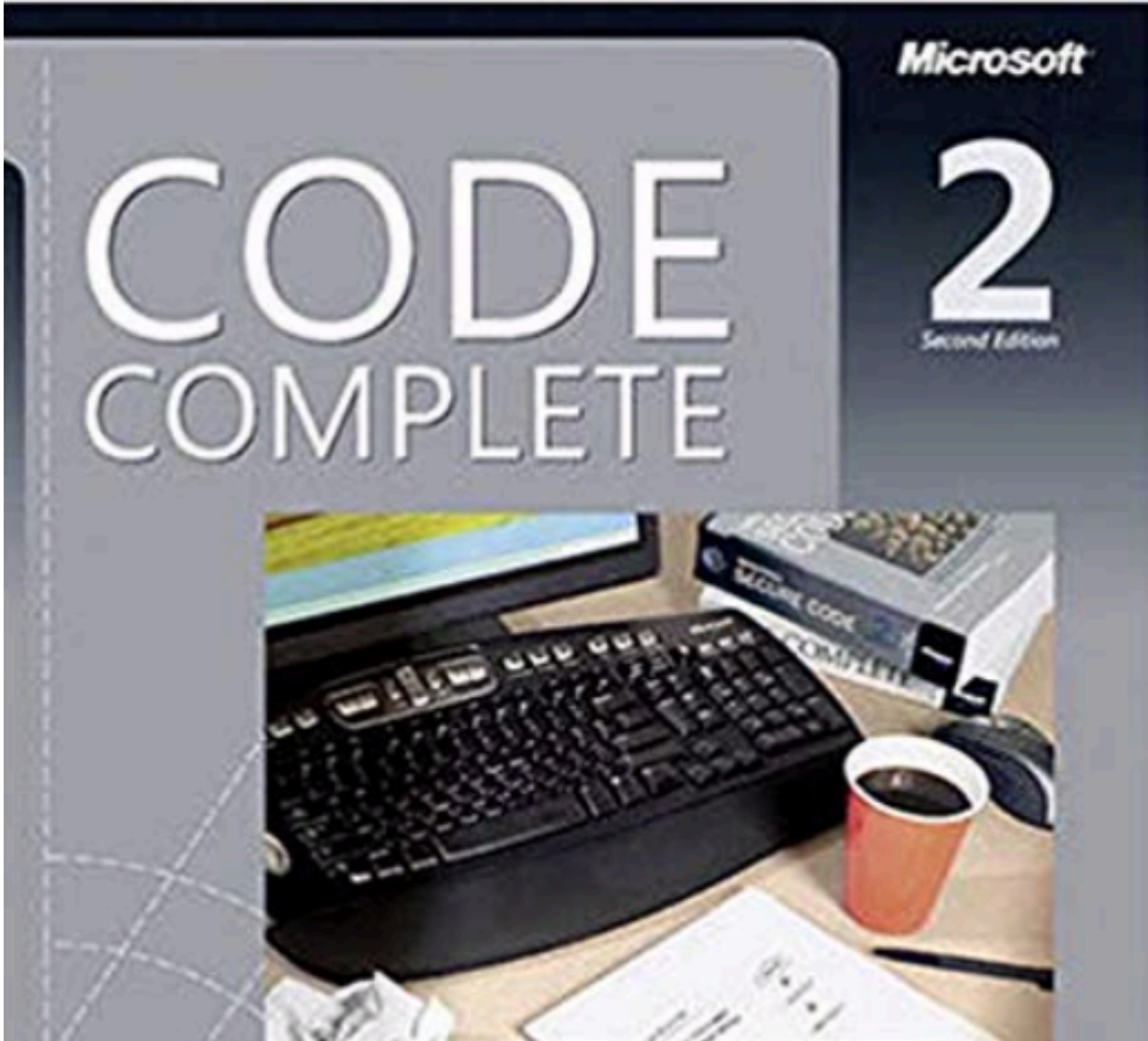
"One problem with computer-science textbooks is that they present silly examples of recursion. The typical examples are computing a factorial or computing a Fibonacci sequence. Recursion is a powerful tool, and it's really dumb to use it in either of those cases. If a programmer who worked for me used recursion to compute a factorial, I'd hire someone else. . . . In addition to being slow and making the use of run-time memory unpredictable, the recursive version of [a factorial-computing] routine is harder to understand than the iterative version . . . ."

--- Code Complete Steve Carrol

```
long fact(int n) {
    if (n == 0) {
        return (long) 1; // base case
    } else {
        return n * fact(n - 1); // recursive call
    }
}
```

SOFTWARE DESIGN PRACTICES

- Code Complete
- Often recommended book about software practice
- Bit old but probably a good source to understand software design practices



```java
long fact(int n) {
    if (n == 0) {
        return (long) 1; // base case
    } else {
        return n * fact(n - 1); // recursive call
    }
}
```

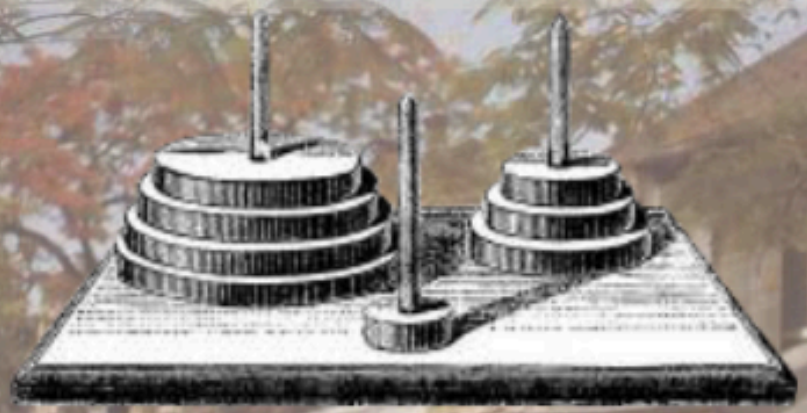
```java
int factorial(int n) {
    int f = 1;
    for(int i=1;i<=n;i++) {
        f *= i;
    }
    return f;
}
```

FACTORIAL FUNCTION

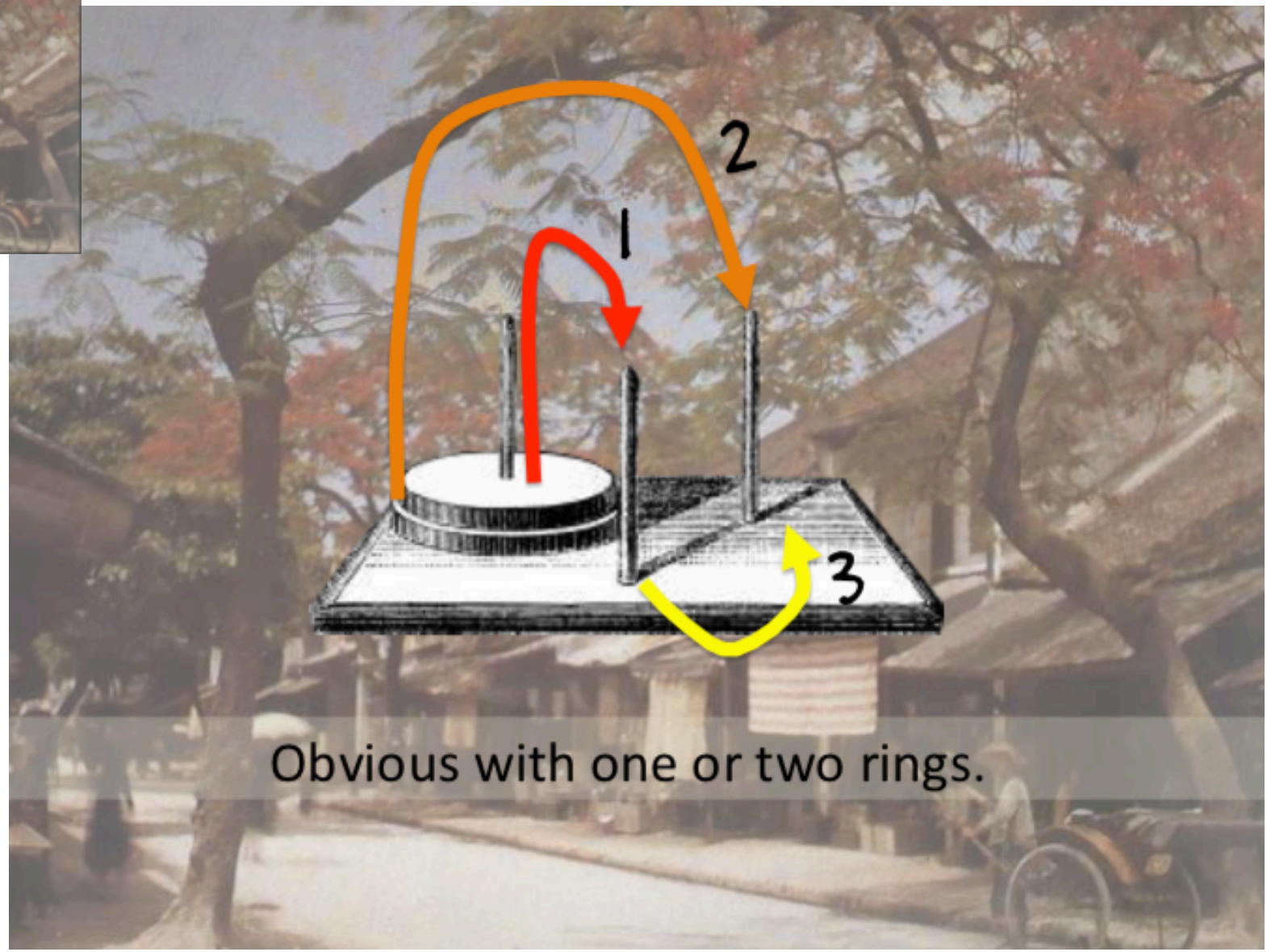
RECURSIVE VS NON RECURSIVE

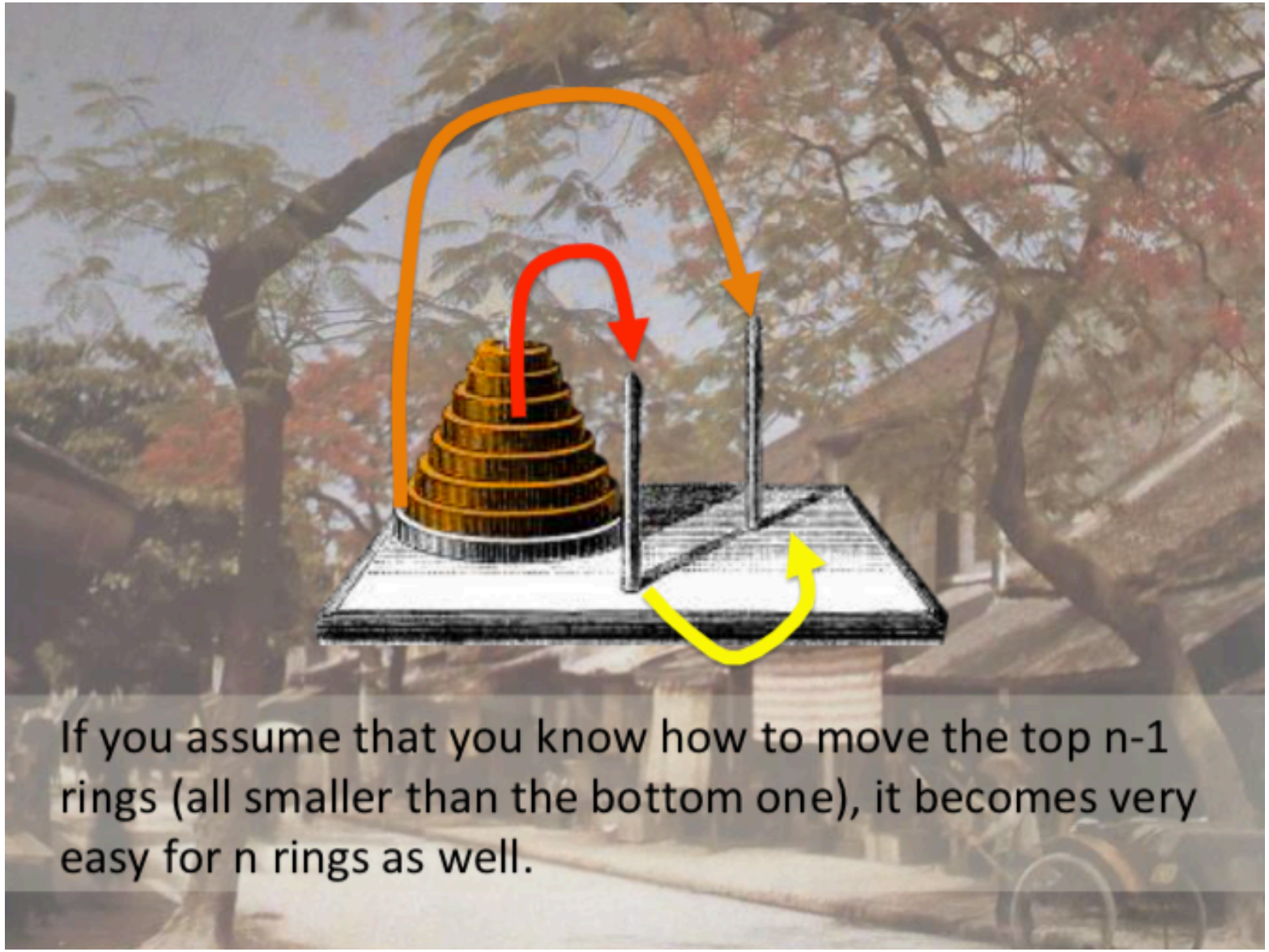
- Unless you have already pre-computed a number of factorial values and don't need to go all the way down to 1, a loop is not more complicated and is more efficient (doing things in the stack has a cost)

Another famous (but good) recursion example popular with textbooks is the towers of Hanoi problem, a puzzle invented by a French mathematician in the late 19th century.



The goal is to move a stack of discs or rings of decreasing radius from one peg to another, using an intermediary third peg, never stacking a bigger ring over a smaller one.


2
1
3
Obvious with one or two rings.


If you assume that you know how to move the top n-1 rings (all smaller than the bottom one), it becomes very easy for n rings as well.

RECURSIVE ALGORITHM FOR TOWERS OF HANOI PROBLEM

```
move_tower(tower_size, from_peg, to_peg, using_peg)
    if tower_size is 1
        move ring from from_peg to to_peg
    else if tower_size is 2
        move top ring from from_peg to using_peg
        move bottom ring from from_peg to to_peg
        move top_ring from using_peg to to_peg
    else
        move_tower(tower_size -1, from_peg, using_peg, to_peg)
        move bottom ring from from_peg to to_peg
        move_tower(tower_size - 1, using_peg, to_peg, from_peg)
```