

Around the year 1900 the illustration of the "nurse" appeared on Droste's cocoa tins.

This is most probably invented by the commercial artist Jan (Johannes) Musset, who had been inspired by a pastel of the Swiss painter Jean Etienne Liotard, *La serveuse de chocolat*, also known as *La belle chocolatière*.

The illustration indicated the wholesome effect of chocolate milk and became inextricably bound with the name Droste.

- Wikipedia Commons

recursion a method of defining functions in which the function being defined is applied within its own definition

$$factorial(n) = \begin{cases} 1 & n = 0 \\ n \cdot factorial(n-1) & n > 0 \end{cases}$$

$$\begin{aligned} factorial(5) &= 5 * factorial(4) \\ &= 5 * (4 * factorial(3)) \\ &= 5 * (4 * (3 * factorial(2))) \\ &= 5 * (4 * (3 * (2 * factorial(1)))) \\ &= 5 * (4 * (3 * (2 * 1))) \\ &= 120 \end{aligned}$$

$$fibonacci(n) = \begin{cases} 1 & n = 0, 1 \\ fibonacci(n-1) + fibonacci(n-2) & n > 1 \end{cases}$$

$$fibonacci(4) = fibonacci(3) + fibonacci(2)$$

$$= fibonacci(2) + fibonacci(1) + \\ fibonacci(1) + fibonacci(0)$$

$$= fibonacci(1) + fibonacci(0) + \\ fibonacci(1) + fibonacci(1) + fibonacci(0)$$

$$= 1 + 1 + 1 + 1 + 1$$

$$= 5$$

Every recursive algorithm must possess:

- a base case in which no recursion occurs
- a recursive case

There must be a logical guarantee that the base case is eventually reached, otherwise the recursion will not cease and we will have an *infinite recursive descent*.

Recursive algorithms may compute a value, or not.

To express recursive algorithms, we need to extend the pseudo-code notation to incorporate the notion of an interface to an algorithm:

algorithm <name> **takes** <list of inputs>

For example:

```
algorithm XtoN takes number X, number N

# Computes the value of X^N.
# Pre:  X, N are integers, N >= 0.
#
    number XtoN      # result

    . . .
    display XtoN     # report result
    halt
```

We must also be able to express the invocation of an algorithm:

`<name> (<list of input values to algorithm>)`

For example:

```
algorithm fiboN takes number N

# Computes the value of the N-th Fibonacci number.
# Pre: N is a non-negative integer.
#
    if N < 2                                # base case
        display 1
    endif

    display fiboN(N-1) + fiboN(N-2) # recursive case
halt
```

Very large integers are (somewhat) easier to read if they are not simply printed as a sequence of digits:

12345678901234567890 vs 12,345,678,901,234,567,890

How can we do this efficiently? The basic difficulty is that printing proceeds from left to right, and the number of digits that should precede the left-most comma depends on the total number of digits in the number.

Here's an idea; let N be the integer to be printed, then:

if N has no more than 3 digits, just print it normally

otherwise

print all but the last 3 digits

print a comma followed by the last 3 digits

```
algorithm printWithCommas takes number N

# Prints N with usual comma-separation.
# Pre:  N is an integer.
#

    if N < 0          # handle negative sign, if necessary
        display '-'
        N := -N
    endif

    if ( N < 1000 ) # base case
        display N
    else
        printWithCommas( N / 1000) # integer division!
        display ','
        display N % 1000 with 0's for padding
    endif

    halt
```


It is a mathematical theorem that any recursive algorithm can be expressed without recursion by using iteration, and perhaps some auxiliary storage.

The transformation from recursion to iteration may be simple or very difficult.

```
algorithm facN takes number N
```

```
# Computes the value of N!.
# Pre: N is a non-negative integer.
#
if N < 2 # base case
    display 1
endif

display N * facN(N-1) # recursive call
halt
```

```
algorithm facN takes number N
```

```
# Computes the value of N!.
# Pre: N is a non-negative integer.
#
number Fac # result

Fac := 1
while N > 0
    Fac := N * Fac
    N := N - 1
enwhile

display Fac
halt
```

(pure) tail recursion

there is a single recursive call, and when it returns there are no subsequent computations in the caller

```
algorithm GCD takes number M, number N
# Computes the largest integer that divides both M and N.
# Pre:  M,N are a non-negative integers, not both 0.
# Credit:  Euclid
#
if N = 0                                # base case
    display M
endif
display GCD(N, M % N)                    # recursive case
halt
```

Tail-recursive algorithms are particularly easy to transform into an iterative form.

"near" tail recursion

there is a single recursive call, and when it returns there are only trivial subsequent computations in the caller; often called *augmenting recursion*

```
algorithm facN takes number N
# Computes the value of N!.
# Pre: N is a non-negative integer.
#
  if N < 2                # base case
    display 1
  endif

  display N * facN(N-1) # recursive case
halt
```

"Near" tail-recursive algorithms are often easy to transform into an iterative form.

Given a $K \times K$ chessboard, find a way to place K queens on the board so that no queen can attack another queen.

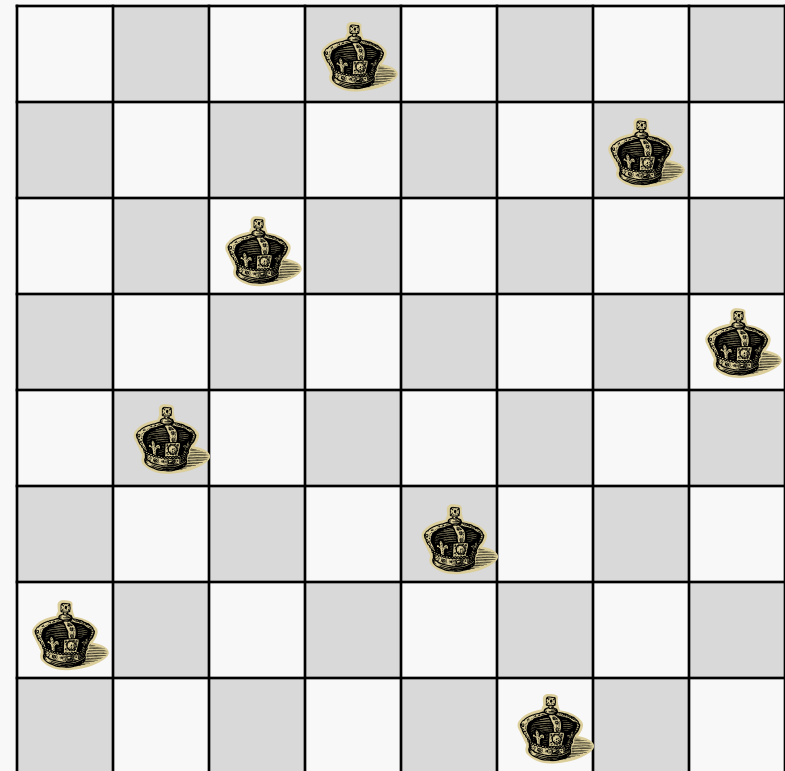
A queen can move an arbitrary number of squares vertically, horizontally or diagonally.

It's immediately clear that there must be one queen in every row and one queen in every column.

Here is one solution:

There are over 4 billion different ways to drop 8 queens onto an 8×8 board.

It's known that there are exactly 92 distinct solutions to the problem.




Let's consider a variant on a 4x4 board... how to start?



Let's flag squares that are under attack with Xs, since we cannot put a queen there.

Let's process the board row by row, from the top down.

Let's start by putting a queen in the first square in row 1:

| | | | |
|---|---|---|---|
|  | X | X | X |
| X | X | | |
| X | | X | |
| X | | | X |



Now for row 2... we have two choices, let's try the first one:

| | | | |
|---|---|--|---|
|  | X | X | X |
| X | X |  | X |
| X | X | X | X |
| X | | X | X |

Oops... now all the squares in row 3 are under attack, so this cannot lead to a solution...




What to try next?

Let's *backtrack*... take back the last move and try a different one:

| | | | |
|---|---|---|---|
|  | X | X | X |
| X | X | X |  |
| X | | X | X |
| X | X | | X |


OK, now we have possibilities... let's fill the free square in row 3:

Rats! Now there are no free squares left in row 4.



| | | | |
|---|---|---|---|
|  | X | X | X |
| X | X | X |  |
| X |  | X | X |
| X | X | X | X |

We can backtrack again, but that means we must now remove the 2nd and 3rd queens, since we've already tried all the possibilities for the 2nd one, and then we must consider a different spot for the 1st one...




So, we'll try the 1st queen in column 2:

| | | | |
|---|---|---|---|
| X |  | X | X |
| X | X | X | |
| | X | | X |
| | X | | |





That leaves just one place for a queen in row 2:

| | | | |
|---|---|---|---|
| X |  | X | X |
| X | X | X |  |
| | X | X | X |
| | X | | X |

And, that leaves just one place for a queen in row 3:

| | | | |
|---|---|---|---|
| X |  | X | X |
| X | X | X |  |
|  | X | X | X |
| X | X | | X |

And, that leaves just one place for a queen in row 4:

| | | | |
|---|---|--|---|
| X |  | X | X |
| X | X | X |  |
|  | X | X | X |
| X | X |  | X |

And, we have a solution... now can we deduce an algorithm?

Let's suppose we have some way to represent a board configuration (size, location of queens, number of queens, etc.)

K Queens Algorithm

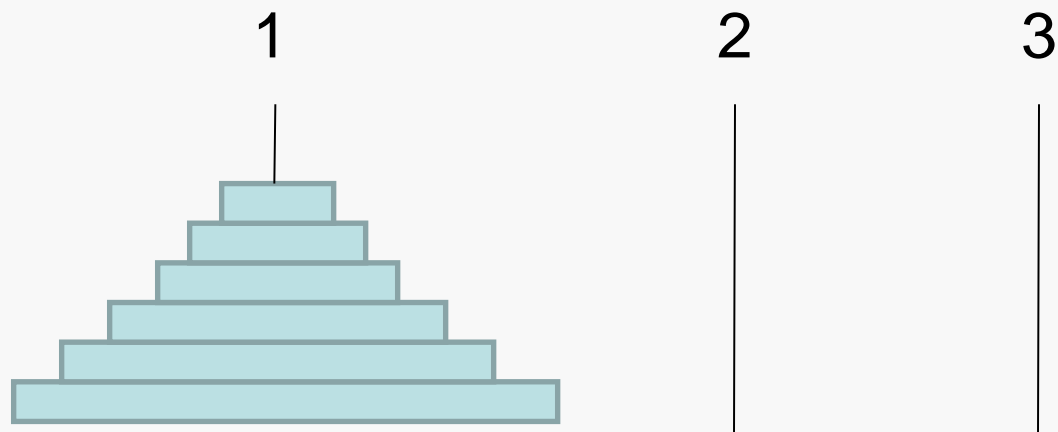
```
Try_config takes configuration C, number m
  if C contains K queens
    display C
    halt
  endif

  for each square in row m of C
    if square is free
      place a queen in square
      Try_config(C, m + 1)           # leads to soln?
      remove queen from square     # no, backtrack
    endif
```

Move one disk at a time

No disk can sit on a smaller disk

Get all disks from pole 1 to pole 3



Algorithm idea:

Move top $n-1$ disks to pole 2

Move bottom disk to pole 3

Move disks from pole 2 to pole 3

How many times must a disk be moved from one pole to another to solve the problem?

Call this $hanoi(n)$ where n is the number of disks; then from the preceding slide we have:

$$hanoi(n) = \begin{cases} 1 & n = 1 \\ 2hanoi(n-1) + 1 & n > 1 \end{cases}$$

Hmm... recursion again.

This is an example of a *recurrence relation* (as are *factorial* and *fibonacci* seen earlier).

Now this does indicate that adding one more disk causes the number of disk moves to more or less double.

But, we'd really like to have a *closed-form* (non-recursive) formula for $hanoi(n)$ since that might be faster to evaluate.

Here it is: $hanoi(n) = 2^n - 1$

For more information on useful techniques for solving recurrence relations, take Math 3134 or CS 4104.