# Recursion

## 1.1 Factorial

Definition of the factorial: \(0! = 1\) and \(n! = 1 \times 2 \times 3 \times \cdots \times n = \prod_{k=1}^{n} k\) for \(n > 0\).

**Definition 1.1.** The factorial \(n!\) of a natural number \(n \in \{0, 1, 2, 3, \ldots\}\) is defined by

\[
    n! = \begin{cases} 
    1 & \{n = 0\} \\
    n \cdot (n - 1)! & \{n > 0\}
    \end{cases}
\]

## 1.2 Growth of the factorial and Stirling’s formula

The factorial grows very rapidly — it is a superexponential function: for every fixed \(c > 1\), there exists \(n_0(c)\) s.t.

\[
    c^n < n! \quad \left\{ n = n_0(c), n_0(c) + 1, \ldots \right\} \quad (1.1)
\]

For example, \(n_0(c) = 25\) works for \(c = 10\): \(25! = 15511210043330985984000000 > 10^{25}\).

**Theorem 1.1 (Stirling’s approximation).** For all \(n = 1, 2, \ldots\) there exists \(0 < \theta_n < 1\) s.t.

\[
    n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \times \exp\left(\frac{\theta_n}{12n}\right).
\]

So, \(n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n\). In fact, Stirling’s formula provides a tight lower bound (hence (1.1) is correct with \(n_0(c) = \lceil ce \rceil\)).
Exercise 1.1. Use Stirling’s formula to characterize the asymptotic growth of the double factorial:

\[(2k + 1)!! = 1 \cdot 3 \cdot 5 \cdot 7 \cdot \cdots (2k + 1) \quad (2k)!! = 2 \cdot 4 \cdot 6 \cdot \cdots (2k) \quad \{k = 0, 1, 2, \ldots \}\]

Hint: write \((2k)!! = (k!) \prod_{i=1}^{k} 2, (2k + 1)!! = \frac{(2k+1)!}{k! \prod_{i=1}^{k} 2},\) and plug in Stirling’s formula.

1.3 Call stack

Definition 1.1 translates directly into a recursive algorithm:

```c
int fact(int n) {  
    if (n==0) return 1;  
    else return n*fact(n-1);  
}
```

One can verify that the algorithm satisfies the minimal requirements for a correct recursion: (1) there is a base case, and (2) every recursive call takes us “closer” to a base case. Consequently, the algorithm terminates after a finite number of recursive calls.

During the execution, the program needs to recover the context (e.g., parameter \(n\)) after returning from a recursive call. It is impossible to predict the depth of the recursion at compile-time. Memory for local variables and parameters needs to be allocated at runtime. Typically, the run-time environment uses a call stack. Whenever a procedure is activated, a block is allocated on the call stack to store the parameters, local variables, and the return address.

In the machine code, \(SP\) denotes the stack pointer. An activation block uses two memory cells in this example: \(n\) is stored at \(SP - 1\), and the return address is stored at \(SP - 2\). \(r_0\) and \(r_1\) are CPU registers. \(r_0\) is used to store the return value.

```
int fact(int n)  
{  
    if (n==0) return 1;  
    else return n*fact(n-1);  
}
```

imaginary assembly code for `fact`

M1 load r1, [SP - 1] // (r1 stores n)
M2 jmp nonzero r1, M6
M3 load r0, 1 // (value returned in r0)
M4 sub SP, 2 // (resetting the call stack)
M5 jmp [SP]
M6 sub r1, 1 // (r1 = n - 1)
M7 store [M11], [SP] // (return address)
M8 add SP, 2 // (frame allocation)
M9 store r1, [SP - 1] // (argument)
M10 jmp [M11] // (executing the procedure call)
M11 mul r0, [SP - 1] // (value returned in r0)
M12 sub SP, 2 // (resetting the stack)
M13 jmp [SP]
Variables. variable = abstraction of a memory location [John von Neumann]
nom + address (lvalue) + value (rvalue) + type + visibility
Local variable (including function parameters): lvalue is relative with respect to the stack frame ⇒ every activated copy of the procedure has its own address space.

1.4 Fibonacci numbers
Definition 1.2. The Fibonacci numbers \( F(n) \) are defined for \( n = 0, 1, 2, \ldots \) by:
\[
F(0) = 0; \quad F(1) = 1; \quad F(n) = F(n-1) + F(n-2) \quad \{ n > 1 \} \tag{1.2}
\]
Binet’s formula. The roots of the homogeneous recurrence equation \( x^n = x^{n-1} + x^{n-2} \) are \( \phi = \frac{1+\sqrt{5}}{2} = 1.618 \cdots \) et \( \phi_- = 1 - \phi = \frac{1-\sqrt{5}}{2} = -0.618 \cdots \). We can find the specific solution by checking \( F(0) \) and \( F(1) \):
\[
F(0) = 0 = a\phi^0 + b\phi^0; \quad F(1) = 1 = a\phi^1 + b\phi_1.
\]
Hence \( a = -b = 5^{-1/2} \), and
\[
F(n) = \frac{\phi^n - \phi_-^n}{\sqrt{5}}. \tag{1.3}
\]

In other words, the Fibonacci numbers grow exponentially. Since \( F(n) \) is integer and \( \left| \frac{\phi^n}{\sqrt{5}} \right| < \frac{1}{2} \) for all \( n \geq 0 \), \( F(n) \) must equal \( \frac{\phi^n}{\sqrt{5}} \), rounded to the nearest integer:
\[
F(n) = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor.
\]

Exercise 1.2. Characterize the growth of the Pell numbers \( P(n) \) for \( n = 0, 1, 2, 3, \ldots \):
\[
P(0) = 0; P(1) = 1; P(n) = 2P(n-1) + P(n-2) \quad \{ n > 1 \}.
\]

1.5 Euclid’s algorithm
Euclid’s algorithm finds the greatest common divisor (gcd) between two positive integers.

```c
int gcd(int a, int b)
{    assert (b<=a && b>=0);
    if (b==0) return a;
    else return gcd(b, a%b);
}
```

The next theorem, along with Eq. (1.3) show that the maximum recursion depth is logarithmic in \( b \).

Theorem 1.2. Let \( n \) be the largest integer for which \( F(n) \leq b < F(n+1) \) when invoking Euclid’s algorithm. The algorithm terminates after at most \( (n-1) \) recursive calls.
Sometimes, recursion gives a very simple solution for complicated problems.

**Theorem 1.3.**

Definition $A$ disks from peg $i$ to peg $j$ using the intermediate peg $k$.

**Rule 1.** Only one disk may be moved at a time. A move consists of taking the top disk from one peg, and sliding it onto another peg, on top of other disks that may be already there.

**Rule 2.** A disk can be moved to an unoccupied peg, or on top of a larger disk.

A solution can be formulated in terms of a recursive procedure $H(AO\Pi(i \bowtie k \bowtie j, n))$ that moves the top $n$ disks from peg $i$ to peg $j$ using the intermediate peg $k$.

**Theorem 1.3.** $H(AO\Pi)$ moves $n$ disks from one peg to another using $2^n - 1$ moves.

**Proof.** The proof is by induction in $n$. Let $D(n)$ be the number of moves (Line $H3$).

**Base case:** the theorem is correct for $n = 0$ since $D(0) = 0 = 2^0 - 1$.

**Induction hypothesis:** assume that the theorem holds for some $n \geq 0$.

**Induction case:** the algorithm has two recursive calls and one $MOVE(i \rightarrow j)$ moves the top disk from peg $i$ to peg $j$. There are three pegs, and the disks are originally ordered in descending order, with the smallest on top.

In the game **Towers of Hanoi**, a stack of disks with different diameters $(1, 2, \ldots, n)$ has to be moved from one peg onto another, obeying Rules 1 and 2 below. Operation $MOVE(i \rightarrow j)$ moves the top disk from peg $i$ to peg $j$. There are three pegs, and the disks are originally ordered in descending order, with the smallest on top.

**Exercise 1.3.** Prove by induction that disk $m$ is moved $2^{m-1}$ times. (Note that the result implies Theorem 1.3 since $2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$.)

**Exercise 1.4.** Show by induction that $\sum_{k=0}^{n} F(k) = F(n+2) - 1$ and that $\sum_{k=0}^{n} kF(k) = nF(n+2) - F(n+3) + 2$.

**Hint:** show the equalities for the base cases (there are two!) and proceed with the induction using the definition 1.2.