1 Recursion

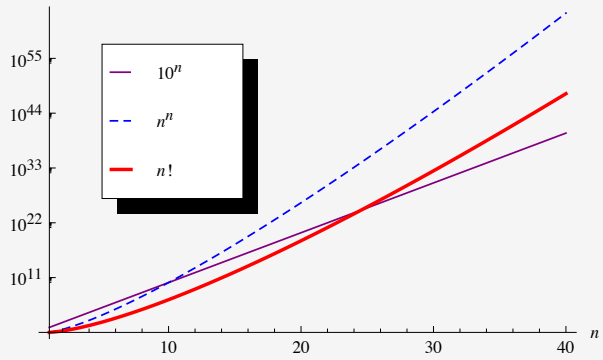
1.1 Factorial

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

Definition 1.1. *The* factorial $n!$ *of a natural number* $n \in \{0, 1, 2, 3 \dots\}$ *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.

$$c^n < n! \qquad \{n = n_0(c), n_0(c)+1, \dots\} \quad (1.1)$$

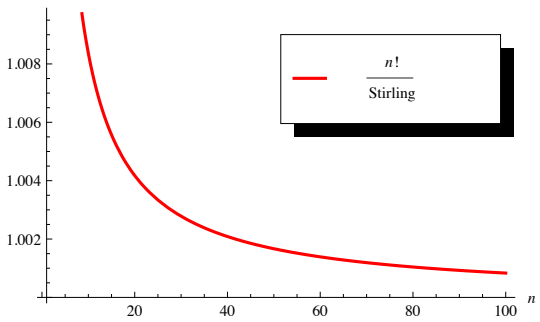
For example, $n_0(c) = 25$ works for $c = 10$: $25! = 15511210043330985984000000 > 10^{25}$.

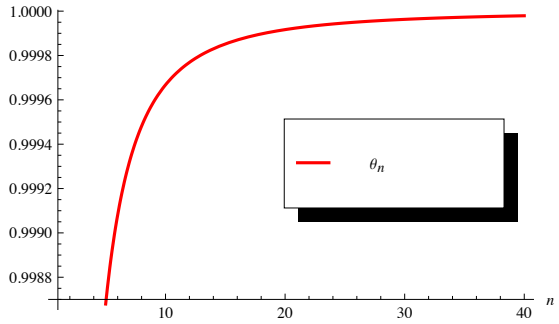
W(en)

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

$$n! = \underbrace{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}_{\textit{Stirling's formula}} \times \underbrace{\exp\left(\frac{\theta_n}{12n}\right)}_{\textit{error of order } 1/n}.$$

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*:*

$$(2k+1)!! = 1 \cdot 3 \cdot 5 \cdot 7 \cdots (2k+1) \quad (2k)!! = 2 \cdot 4 \cdot 6 \cdots (2k) \qquad \{k = 0, 1, 2, \dots\}$$

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

1.3 Call stack

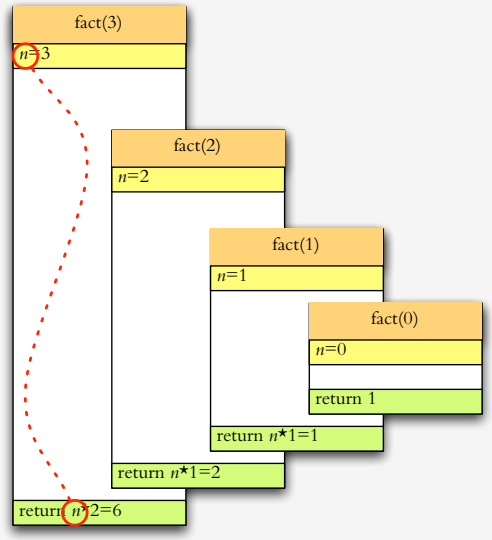
W(en)

Definition 1.1 translates directly into a **recursive algorithm**:

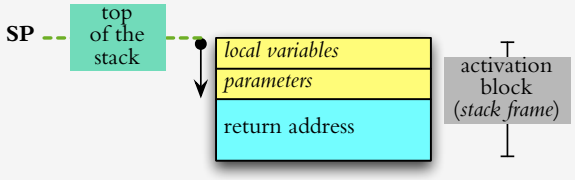
```
    FACT(n)                                   //(calculates n!)
F1  if n = 0 then return 1
F2  else return n × FACT(n − 1)               // recursive call
```

```
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 run-time. 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.




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

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

```
    imaginary assembly code for fact
M1  load r1, [SP − 1]                    // (r1 stores n)
M2  jumpnonzero r1, M6
M3  load r0, 1                           // (value returned in r0)
M4  sub SP, 2                            // (resetting the call stack)
M5  jump [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 jump M1                              // (executing the procedure call)
M11 mul r0, [SP − 1]                     // (value returned in r0)
M12 sub SP, 2                            // (resetting the stack)
M13 jump [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:* W(en)

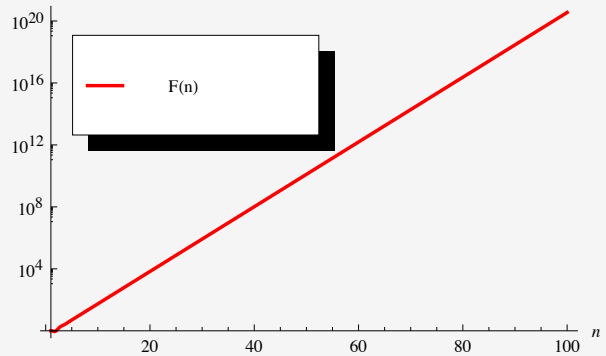
$$F(0) = 0; \quad F(1) = 1; \quad F(n) = F(n-1) + F(n-2) \qquad \{n > 1\} \tag{1.2}$$

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 $F(1)$:

$$F(0) = 0 = a\phi^0 + b\phi_-^0; \qquad F(1) = 1 = a\phi^1 + b\phi_-^1.$$

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 $\left|\phi_-^n/\sqrt{5}\right| < 1/2$ for all $n \geq 0$, $F(n)$ must equal $\phi^n/\sqrt{5}$, rounded to the nearest integer:

$$F(n) = \left\lfloor \phi^n/\sqrt{5} + 1/2 \right\rfloor.$$

Exercise 1.2. *Characterize the growth of the Pell numbers $P(n)$ for $n = 0, 1, 2, 3, \ldots$:*

$$P(0) = 0; P(1) = 1; P(n) = 2P(n-1) + P(n-2) \qquad \{n > 1\}.$$

1.5 Euclid's algorithm

W(en)

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

GCD(a, b) // $\{b \leq a\}$
E1 **if** $b = 0$ **then return** a
E2 **else return** GCD$\big(b, a \bmod b\big)$;

```
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.*

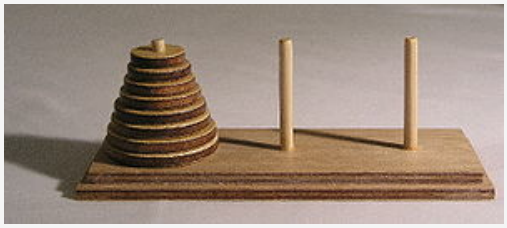
Proof. Define n_i for $i = 1, 2, \ldots$ as the index for the Fibonacci number at recursion depth i for which $F(n_i) \le a < F(n_i + 1)$ ($i = 1$ in the initial call). If $b < F(n_i)$, then $n_{i+1} \le n_i - 1$ in the next recursive call. If $b \ge F(n_i)$, then $a \bmod b \le a - b < F(n_i - 2)$. So, after two recursive calls, $a < F(n_i - 2)$ and, thus, $n_{i+2} \le n_i - 2$. Consequently, the recursion depth is bounded by $n_1 - 2$: with $n_i \le 3$, $0 \le b \le a < 3$ and the algorithm terminates after at most one more recursion. For the theorem's tighter bound, consider the initial value of b: $F(n_2) \le b < F(n_2 + 1)$ (since $a \leftarrow b$ at the first recursive call), and the algorithm terminates after at most $n_2 - 1$ recursions. ■

REMARK. The bound of Theorem 1.2 shows the worst-case: $a = F(n+1), b = F(n)$. with such a choice, $a \bmod b = F(n-1)$, and the algorithm recurs $n-2$ times to get $a = F(3) = 2, b = F(2) = 1$, and finishes with one more recursion where b becomes 0, and returns the answer $\gcd\big(F(n), F(n-1)\big) = 1$.

1.6 Towers of Hanoï

W(en)

Sometimes, recursion gives a very simple solution for complicated problems.



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$(i \to j)$ 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.

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 HANOI$(i \curvearrowright k \curvearrowright j, n)$ that moves the top n disks from peg i to peg j using the intermediate peg k.

```
   HANOI(i ↷ j ↷ k, n)
H1 if n ≠ 0 then
H2     HANOI(i ↷ j ↷ k, n − 1)
H3     MOVE(i → j)
H4     HANOI(k ↷ i ↷ j, n − 1)
```

Theorem 1.3. HANOI *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 $n \ge 0$.

Induction case: the algorithm has two recursive calls and one MOVE: $D(n+1) = 2D(n) + 1$. By the induction hypothesis, $D(n+1) = 2 \cdot \Big(2^n - 1\Big) - 1 = 2^{n+1} - 1$. Hence the theorem holds for $(n+1)$.

Consequently, $D(n) = 2^n - 1$ for all $n = 0, 1, 2, \ldots$. ■

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

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).