Recursion

Recall factorial: \( n! \triangleq n \times (n - 1) \times \ldots \times 2 \times 1 \)

Written more formally:
\[
n! = \begin{cases} 
1 & \text{if } n = 0 \\
(n - 1)! & \text{if } n > 0 
\end{cases}
\]

This is a recursive definition — \( n! \) is defined in terms of \( (n - 1)! \)

In programming we say that a function (subroutine) is recursive if, when called, it may be called again before it returns.

```cpp
int factorial(int n) {
    int result = 1;
    if (n > 0) {
        result = n * factorial(n-1);
    }
    return result;
}
```

Note: if `foo` calls `bar` and `bar` calls `foo` then they’re both recursive.

Stack Frame/Invocation Record

When a function calls another function, the system must save:
- local variables,
- registers,
- instruction to return to — called the invocation record.

This information is needed in LIFO order, so it’s stored on a stack (in most programming languages).

Stack frame — the state of the stack of invocation records at a particular time. (Note: sometimes the location of the top of the stack is called the stack frame or stack frame pointer.)

Aside: Trees and Graphs

A graph is a set of vertices, \( V \), and edges, \( E \), which are pairs of vertices (i.e., \( e = (v_1, v_2) \)).

Two vertices are adjacent if there is an edge connecting them.

Two vertices are connected if there is a sequence of edges leading from one to the other.

A graph is connected if every vertex is connected to every other one.

A cycle is a sequence of edges leading from a vertex back to itself in which no edge appears more than once.

A tree is a connected graph with no cycles.
**Call Trees**
Illustrate the execution of an algorithm by a tree:

- each vertex represents an invocation of a function,
- each edge represents a function (the parent) calling another (the child),
- the root of the tree is the starting point of the algorithm (e.g., main) — it has no parent.

siblings are vertices with the same parent
leaf vertices have no children.

The number of vertices on the longest path from the root to a leaf is the **height** of the tree.

The depth of a vertex is the number of branches on a path from the root to the vertex.

---

**Factorial Call Tree**

```
      n!
     /   \
   (n-1)!
  /     \  
(n-2)!
  .      .
  .      .
  .      .
  1!
  0!
```

---

**Fibonacci Numbers**

\[
F(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F(n-1) + F(n-2) & \text{if } n > 1 
\end{cases}
\]

---

**Towers of Hanoi**

- 3 pegs, \( n \) disks, all different sizes
- Start with all disks on peg #1, ordered so that smaller disks are on top.
- Goal is to move all \( n \) disks to peg #2, subject to:
  - Move one disk at a time.
  - A larger disk can never be on top of a smaller disk.

If we can move the top \( n - 1 \) disks to the spare peg (#3) then we can simply move the largest disk to #2 and then move the other disks back on top of it.
Algorithm Hanoi \( n, \text{source}, \text{dest}, \text{spare} \)

\[
\text{if } n > 0 \text{ then} \\
\quad \text{Move top } n - 1 \text{ disks from source to spare using dest} \\
\quad \text{Move disk } n \text{ from source to dest} \\
\quad \text{Move top } n - 1 \text{ disks from spare to dest using source} \\
\text{end if}
\]

Call tree

Recursion Principles

We need two things:

1) Base case — a simple instance of the problem that we know how to solve without recursion (e.g., \(0!\), 1 disk Towers of Hanoi).

2) Recursive step — a means of solving a given instance of the problem by reducing it to one or more more simple instances.

- Important that each recursive step only uses more simple instances.
- A variant expression is a natural number expression that is smaller in recursive calls.
- If there is a variant expression the recursion cannot be infinite.

\(- variant \text{ expression} = 0\) is the base case.

See: ListR.h.

Tail Recursion

A recursive function in which the last thing it does is a recursive call to itself is tail recursive.

Tail recursion can be converted to iteration by re-assigning the values of local variables and using a loop.

```
int factorial(int n) {
    int result = 1;
    for (int i = n; i > 0; i--) {
        result *= i;
    }
    return result;
}
```

Algorithm Maze \((\text{start, end})\)

\[
\text{mark start as seen} \\
\text{done } = (\text{start } == \text{end}) \\
\text{if } \neg \text{done } \land \text{forward is accessible and unseen} \text{ then} \\
\quad \text{Move forward} \\
\quad \text{done } = \text{Maze(forward, end)} \\
\text{if } \neg \text{done} \text{ then} \\
\quad \text{Move backward} \\
\text{end if} \\
\text{if } \neg \text{done } \land \text{left is accessible and unseen} \text{ then} \\
\quad \text{Move left} \\
\quad \text{done } = \text{Maze(left, end)} \\
\text{if } \neg \text{done} \text{ then} \\
\quad \text{Move right} \\
\text{end if} \\
\text{end if}
\]

Backtracking
if \( \neg \text{done} \land \text{right is accessible and unseen} \) then
  Move right
  \( \text{done} = \text{Maze(left, end)} \)
  if \( \neg \text{done} \) then
    Move left
  end if
end if
return \text{done}