Recursion

- A function which calls itself (directly or indirectly) is called a **recursive function**.
- Recursion can be used to do something repeatedly (similar to loops).
- For many problems, it is much easier to use recursion than loops to solve the problems.
- This is especially true for many problems which can be defined recursively.

Example: Factorial

- The factorial is defined in this way:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \cdots \times n & n > 0 \end{cases}$$

  This can be computed by a loop.

- We can also define the factorial as:

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

  So, if we know how to compute the factorial of $n - 1$, we know how to compute the factorial of n.

Example: Factorial

```
int factorial(int n)  // assumes n >= 0
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

To see how the computation is done, trace `factorial(3)`:

```
factorial(3) = 3 * factorial(2)
             = 3 * (2 * factorial(1))
             = 3 * (2 * (1 * factorial(0)))
             = 3 * (2 * (1 * 1))
```

Recursion vs. Iteration

- In iteration, we start from the "bottom" and iterate up to build the result.
- In recursion, we start from the "top" (what we are interested in at the end), and reduce the problem until we reach the bottom.
- Note that in recursion we must have a "bottom" (known as the **base case**) which stops the recursion. Otherwise, we will have infinite recursion.
- The computations are done in reverse order as each invocation of the recursive function exits (in this case).

Example: List Reversal

- We can take advantage of the "reversal" property in some applications.
- Example: we wish to print out the elements in a linked list in reverse order.

```
void printList(Element *e)
{
  if (e) {   // if NULL do nothing
    printList(e->next);
    cout << e->data << endl;
  }
}
```

- Putting the print statement before the recursive call would print the list elements in order.

Recursion: Local Variables and Parameters

- When a "normal" function is called, it has its own local variables and parameters.
- This is the same for a recursive function: each time the function is called it gets its own local variables and parameters.
- Changing local variables in one invocation does not affect the values in another invocation.
- Recursive invocations communicate to each other by return value or reference parameters (or global/static variables. . . not recommended).

Writing Recursive Functions

- Identify easy or "small" inputs such that the computation is trivial. e.g. 0! is easy, printing an empty list is easy.
- For other inputs, try to see if you can break up the problem into smaller problem(s) of the same type.

Tower of Hanoi

- Three pegs, n disks of different sizes.
- Originally all n disks are on peg 1, with the disks sorted by size (largest at the bottom).
- Rule: move one disk at a time, never putting a larger disk on top of a smaller disk.
- Goal: move all the disks to peg 3.

Recursive Solution

- If we want to move 1 disk from peg a to peg b, it is easy (just do it).
- If we want to move n disks from peg a to peg b, what we need to do is:
  - Move the top $n-1$ disks to a temporary peg $c \neq a, b$.
  - Move the largest disk from a to b.
  - Move the top $n-1$ disks from c to b.
- "Moving the top $n-1$ disks" is a smaller problem of the same type.

Tower of Hanoi

```
void hanoi(int n, int from, int to, int temp)
{
  if (n == 1)
    cout << from << " -> " << to << endl;
  else {
    hanoi(n-1, from, temp, to);
    cout << from << " -> " << to << endl;
    hanoi(n-1, temp, to, from);
  }
}
```

Start with

```
hanoi(n, 1, 3, 2);
```

Tic Tac Toe: Perfect Player

- Write one routine to decide for the best move for player 1, and another routine for player 2.
- In player 1:
  - Try all possible locations. If it is a legal move, make the move.
  - Call the other player and see what his/her best move is.
  - Choose the move that makes the other player's best move the worst.
  - Remember to "undo" a move before trying the next one (if pass by reference).
  - Base case: when the game is finished (win/tie/loss).
- The code for player 2 is the same as that for player 1 (except what is best for player 2 is different).
- You can write both functions into one (with a parameter on whose turn it is).

Tic Tac Toe: Perfect Player

- Basically, the algorithm looks at all possible sequences of moves.
- So if there is a sequence that forces a win, it will find it.
- Failing that, if there is a sequence that forces a tie, it will find it.
- There are at most $9! = 362,880$ sequences of moves in this game, so we can look at all possibilities.
- You can write similar routines for other games (e.g. chess), but there are many more possibilities and it may be too slow.
- Lots of tricks are used to speed up such searches (take courses on AI and algorithms if interested).