

Recursion

Often in solving a problem one breaks up the problem into subtasks. *Recursion* can be used if one of the subtasks is a *simpler version* of the original problem.

9.1 An Example

Suppose we are trying to sort a list of numbers. We could first determine the minimum element; and what remains to be done is to sort the remaining numbers. So the code might look something like this:

```
void sort(Collection &C) {
    min = C.getMinimum();
    cout << min;
    C.remove(min);
    sort(C);
}
```

Every recursive method needs a *stopping case*: otherwise we have an infinite loop or an error. In this case, we have a problem when C is empty. So one always checks to see if the problem is simple enough to solve directly.

```
void sort(Collection &C) {
    if( C.isEmpty() )
        return;
    ... // as before
```

Example. Printing out a decimal number. The idea is to extract one digit and then recursively print out the rest. It's hard to get the most significant digit, but one can obtain the least significant digit (the “ones” column): use `num % 10`. And then `num/10` is the “rest” of the number.

```
void print( int n ) {
    if( n>0 ) {
        print ( n/10 );
        cout << n%10 ;
    }
}
```

9.2 Tracing Code

It is important to be able to *trace* recursive calls: step through what is happening in the execution. Consider the following code:

```
void g( int n ) {  
    if( n==0 ) return;  
    g(n-1);  
    cout << n;  
}
```

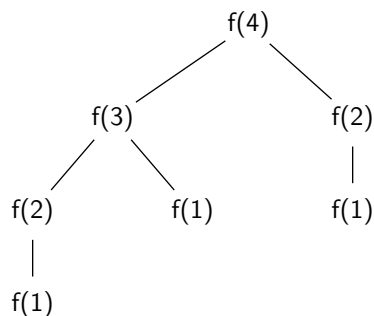
It is not hard to see that, for example, `g(3)` prints out the numbers from 3 down to 1. But, you have to be a bit more careful. The recursive call occurs before the value 3 is printed out. This means that the output is from smallest to biggest.

```
1  
2  
3
```

Here is some more code to trace:

```
void f( int n ) {  
    cout << n;  
    if(n>1)  
        f(n-1);  
    if(n>2)  
        f(n-2);  
}
```

If you call the method `f(4)`, it prints out 4 and then calls `f(3)` and `f(2)` in succession. The call to `f(3)` calls both `f(2)` and `f(1)`, and so on. One can draw a *recursion tree*: this looks like a family tree except that the children are the recursive calls.



Then one can work out that `f(1)` prints 1, that `f(2)` prints 21 and `f(3)` prints 3211. What does `f(4)` print out?

Exercise

Give recursive code so that `brackets(5)` prints out `(((((()))))`).

9.3 Methods that Return Values

Some recursive methods return values. For example, the sequence of Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, 21, ... is defined as follows: the first two numbers are 1 and 1, and then each next number is the sum of the two previous numbers. There is obvious recursive code for the Fibonacci numbers:

```
int fib(int n) {
    if( n<2 )
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

WARNING: Recursion is often easy to write (once you get used to it!). But occasionally it is very inefficient. For example, the code for `fib` above is terrible. (Try to calculate `fib(30)`.)

9.4 Application: The Queens problem

One can use recursion to solve the *Queens* problem. The old puzzle is to place 8 queens on a 8×8 chess/checkers board such that no pair of queens attack each other. That is, no pair of queens are in the same row, the same column, or the same diagonal.

The solution uses search and *backtracking*. We know we will have exactly one queen per row. The recursive method tries to place the queens one row at a time. The main method calls `place(0)`. Here is the code/pseudocode:

```
void place(int row) {
    if(row==8)
        celebrateAndStop();
    else {
        for( queen[row] = all possible vals ) {
            check if new queen legal;
            record columns and diagonals it attacks;
            // recurse
            place(row+1);
        }
    }
}
```

```
        // if reach here, have failed and need to backtrack
        erase columns and diagonals it attacks;
    }
}
```

9.5 Application: The Steps problem

One can use recursion to solve the Steps problem. In this problem, one can take steps of specified lengths and has to travel a certain distance exactly (for example, a cashier making change for a specific amount using coins of various denominations).

The code/pseudocode is as follows

```
bool canStep(int required)
{
    if( required==0 )
        return true;
    for( each allowed length )
        if( length<=required && canStep(required-length) )
            return true;
    //failing which
    return false;
}
```

The recursive boolean method takes as parameter the remaining distance required, and returns whether this is possible or not. If the remaining distance is 0, it returns true. Else it considers each possible first step in turn. If it is possible to get home after making that first step, it returns true; failing which it returns false. One can adapt this to actually count the minimum number of steps needed. See code below.

One can also use recursion to explore a maze or to draw a snowflake fractal.

Example Program: StepsByRecursion.cpp

```
// StepsByRecursion.cpp
// a recursive program to solve Steps problem
// wdg 2008
#import <iostream>
using namespace std;

static const int IMPOSSIBLE=-1;
```

```

// @pre: required>=0
// @returns: minimum number of steps; a value of -1 means impossible
int minSteps(int required, int allowed[], int numAllowed) {
    if(required==0)
        return 0;
    else {
        int min = required+1; //guaranteed to be too large
        // consider all valid first steps
        for(int poss=0; poss<numAllowed; poss++)
            if( allowed[poss]<=required ) {
                // call recursively to determine best way to do remainder
                int temp = minSteps( required-allowed[poss], allowed, numAllowed);
                if(temp<min && temp!=IMPOSSIBLE)
                    min=temp;
            }
        // return
        if(min==required+1)
            return IMPOSSIBLE;
        else
            return 1+min;
    }//else
}

int main( )
{
    int coins[] = {1,5,10};
    cout << "USCoins do 17 in " << minSteps(17,coins,3) << endl;
        // dime, nickel and 2 pennies
    int silly[] = {2,4,6};
    cout << "Impossible coins returns " << minSteps(17,silly,3) << endl;
        // odd is impossible
}

```