COMP90038 Algorithms and Complexity

Recursion

Michael Kirley

Lecture 6

Semester 1, 2016

Recursion

We have already seen a few examples of recursion. Now let us cover recursion in some more detail.

Recursion is a very natural approach when we have to process data structures that are defined recursively, such as trees.

But there are also examples of, say, array-processing algorithms that are naturally recursive.

Next week we shall express depth-first graph traversal recursively (that's the natural way), and later we'll meet other examples of recursion.

Example: Factorial Numbers

To compute $n!$, we can use recursion (left algorithm) or iteration (right algorithm):

```
function $\textsc{Fac}(n)$
    if $n = 0$ then
        return 1
    return $\textsc{Fac}(n - 1) * n$
```

```
function $\textsc{Fac}(n)$
    $result \leftarrow 1$
    while $n > 0$ do
        $result \leftarrow result * n$
        $n \leftarrow n - 1$
    return $result$
```

Here the iterative solution would normally be preferred.

Example: Fibonacci Numbers

To generate the *n*th number of sequence 1 1 2 3 5 8 13 21 34 55 ...

```
function $\textsc{Fib}(n)$
    if $n = 0$ then
        return 1
    if $n = 1$ then
        return 1
    return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$
```

This algorithm follows the definition of Fibonacci numbers closely, so it is very easy to understand.

Unfortunately it performs a lot of redundant computation. Why?

In fact, this algorithm is exponential in n.

Fibonacci Again

The following solution makes use of the fact that we only ever need to remember the two latest numbers in the sequence (recursive version on the left, iterative on the right).

```
function FIB($n, a, b$)
    if $n = 0$ then
        return $a$
    return FIB($n - 1, a + b, a$)
```

Initial call: FIB$(n, 1, 0)$

```
function FIB($n$)
    $a \leftarrow 1$
    $b \leftarrow 0$
    while $n > 0$ do
        $a \leftarrow a + b$
        $b \leftarrow a - b$
        $n \leftarrow n - 1$
    return $a$
```

In either case, we have a simple, linear-time solution.

(There is a cleverer, still recursive, way which is $O(\log n)$.)

The Tower of Hanoi

Assume *n* disks need to be moved from peg *init* to peg *fin*.

Of course we can only move some top-most disk to the top of some pile. Nevertheless the solutions is conveniently expressed recursively, saying how to solve the task for *n* desks, assuming we first solve the problem for $n - 1$:

```
function $\textsc{Hanoi}(n, init, aux, fin)$
    if $n > 0$ then
        $\textsc{Hanoi}(n-1, init, fin, aux)$
        Move one disk from $init$ to $fin$
        $\textsc{Hanoi}(n-1, aux, init, fin)$
```

This is an elegant formulation which relieves of us the headache of deciding which peg to target in the first move.

A Challenge: The Coin Change Problem

There are six different kinds of Australian coin.

In cents, their values are: 5, 10, 20, 50, 100, 200.

In how many different ways can I produce a handful of coins adding up to \$4?

This is not an easy problem!

A key to solving it is to find a way of breaking it down to simpler sub-problems.

Recursion as a Problem Solving Technique

The number of ways to make \$4 is:

the number of ways to make \$2
plus
the number of ways to make \$4 using 5, 10, 20, 50, 100 only

The reason is that a given \$4 handful either makes use of a \$2 coin (200) or it does not.

Now we have two sub-problems to solve, but each is smaller than the original problem!

The Recursive Case

The number of ways to make \$4 is 2728:

the number of ways to make \$2: 293
plus
the number of ways to make \$4 using 5, 10, 20, 50, 100 only: 2435

We will express this as an algorithm shortly.

But what are the base cases?

Usually that's the easy part, but here they are harder to determine than what the recursive case looks like!

The Base Cases

If we are asked to produce coins for the amount 0 then we can do that, in exactly one way: by using none of the allowed coins. At the end of a chain of recursive calls, this situation indicates that we found a successful combination of coins.

If we are asked to produce a negative amount then we must renege—it can be done in zero ways.

If we are asked to produce a positive amount, but none of the six kinds of coins are allowed to be used, then again that is not possible—it can be done in zero ways.

The Recursive Algorithm

```
function $\textsc{Ways}(amount, denominations)$
    if $amount = 0$ then
        return 1
    if $amount < 0$ then
        return 0
    if $denominations = \emptyset$ then
        return 0
    $d \leftarrow selectFrom(denominations)$
    return $\textsc{Ways}(amount - d, denominations) +$
            $\textsc{Ways}(amount, denominations \setminus \{d\})$
```

Initial call: $\textsc{Ways}(amount, \{5, 10, 20, 50, 100, 200\})$.

The Recursive Solution and Complexity

Although our recursive algorithm is short and elegant, it is not the most efficient way of solving the problem.

Its running time grows exponentially as you grow the input amount.

More efficient solutions can be developed using memoing or dynamic programming—more about that later.

Coming Soon to a Theatre Near You

Graphs, trees, graph traversal and allied algorithms.