Lecture 5:
Recursive Definitions

Aims:

- To look at recursive definitions; and
- To look at the divide-and-conquer approach to problem-solving.

5.1 Recursion

- An algorithm can call itself. Here's one that takes in a non-negative integer, n.

Algorithm: SelfDestruct(n)
```
{   display"This machine will self-destruct in "n" minutes";
    SelfDestruct(n);
    display"I was only joking!";
}
```

If we execute this algorithm with, e.g., $n = 5$, it displays the first message ("This machine will self-destruct in 5 minutes") and then calls itself. This, in its turn, will display the first message (again), and then call itself (again). Indeed, this algorithm will run forever, repeatedly displaying the first message, never displaying the second message.

Of course, if you code this up in some programming language and run it on some computer, then after displaying the first message many times, it is likely to crash. This is because, each time the program calls itself, the machine must remember where to return to (see later). This uses up some memory space. But since the program keeps making calls, it uses ever more memory. Eventually, it will run out of memory and the program will crash.

- What we have here, of course, is an example of *recursion*: an algorithm (or procedure or program) calling itself. You can see that it is another way, like iteration, of getting the same text to be executed multiple times.

  But this is a degenerate example. It results in an infinite loop. To stop it running forever, we need a *base case*.

- We need to use a conditional that chooses between a (non-recursive) *base case* and the *recursive case*. We also need to make sure that the recursive case calls the algorithm on some data that gets closer to the base case:

Algorithm: SelfDestruct(n)
```
{   if n = 0
    {   display"BANG!!!";
    }
    else
    {   display"This machine will self-destruct in "n
            " minutes";
        SelfDestruct(n − 1);
        display"I was only joking!";
    }
}
```

Some students find writing recursive algorithms hard. Perhaps what makes them seem hard to write is actually how easy they are to write! They seem like a strange kind of magic.

To help you to overcome this, I want to show you that recursion is really natural. It crops up all over the place. Often you don't see it. We often define sets recursively. And we often carry out everyday tasks recursively. So before we look further at recursive algorithms, let's look at some of these naturally-occurring examples of recursion.

5.2 Recursive Definitions

- It's quite common for us to define a set recursively. As a first example, consider the set of royals. Defining *royal*:

  Base case: The monarch of a country.
  Recursive case: The child of a royal.
  Closure: No one else.

- Here's another example: the set of arithmetic expressions.

  Base case: Integers
  $$0, 1, 2, 3, \ldots$$
  Recursive case: If E_1 and E_2 are arithmetic expressions then
  $$(E_1 \textbf{ op } E_2)$$
  is an arithmetic expression, where **op** is one of
  $$+ \quad - \quad \times \quad \text{div} \quad \text{mod}$$
  Closure: Nothing else.

  Obviously, this is a simplified definition. It doesn't allow reals; it doesn't allow relations such as $=$, $\neq$, $<$, etc.; if treated literally, it forces us to write more parentheses than we'd be used to, etc. If we wanted to, we could extend it. We won't bother. The key point is that it is recursive.

5.2.1 Peano's Axiomatisation of the Natural Numbers

- Consider the natural numbers: $\mathbb{N}$, i.e. $\{0, 1, 2, 3, \ldots\}$.

  Perhaps surprisingly, this set can be defined recursively. Here is Peano's recursive definition of the natural numbers:

  Base case: 0 is a natural number;

  Recursive case: If x is a natural number, then succ(x), the successor of x, is a natural number.

  Closure: Nothing else is a natural number.

- From this definition, all the things on the left are natural numbers. On the right we show the more conventional (and shorter!) way of writing the same thing:

0	0
succ(0)	1
succ(succ(0))	2
succ(succ(succ(0)))	3
succ(succ(succ(succ(0))))	4
⋮	⋮

5.3 Recursion in Tasks

- Many human tasks can naturally be solved by taking a *divide-and-conquer* approach. We split the task into two or more simpler tasks. We perform the simpler tasks, and then combine the results.

  But how do we "perform the simpler tasks"? Well, if they're still quite complex tasks then we do them by divide-and-conquer too: we split them into subtasks, perform the subtasks and combine the results. We keep doing this until we reach tasks that are simple enough to be carried out without further decomposition.

  When a subtask is actually a simpler version of the original task, then we have recursion. Here's a simple example of this idea.

Problem 5.1.

Parameters: *A standard pack of 52 playing cards in no particular order.*

Returns: *The pack sorted into suits (♣ then ♡, then ♢, then ♠) and in order of face value (Ace to King) within suit.*

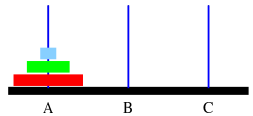
Algorithm: SortCards(S)

```
{   if S is small enough to easily sort in your hand
    {   Rearrange S into order;
    }
    else
    {   Split S into roughly equal halves, S1 and S2;
        SortCards(S1);
        SortCards(S2);
        Merge the sorted halves;
    }
}
```

Some of you may know that this is pretty close to a well-known recursive sorting algorithm called *merge sort*. Notice that, as an algorithm, it calls itself twice. (But so did our recursive definition of arithmetic expressions!)

- The *Towers of Hanoi* give us another task that can be naturally decomposed to give a recursive solution. You may know the task:

  > *In the great temple at Benares, there are three diamond pegs, A, B and C. 64 gold disks of varying diameters, each with a hole in its centre, have been piled onto peg A, in descending order of diameter. The other pegs are empty. We want to move the disks from A to B.*



  But there are three rules:

  1. We can move only one disk at a time.
  2. After a move, each disk must be on one of the pegs.
  3. A larger disk may never be placed on top of a smaller one.

  Legend has it that, when the task is complete, there will be a thunderclap and the world will end.

  This problem, as stated, has only one instance. So we instead solve a more general problem. There are still three pegs, but there are n disks, where n is a positive integer.

  How do we solve it? Here are some thoughts that lead towards a solution.

  – At some point, we will need to move the largest disk to B.
  – At that point, B will have to be empty.
  – But we can't move the largest disk until all the others have been moved somewhere.
  – It follows they'll have to be piled on C.

  That brings us close to a useful problem decomposition:

  – Our first job is to move all the other disks to C, perhaps using B.

– Then we can move the largest disk to B.

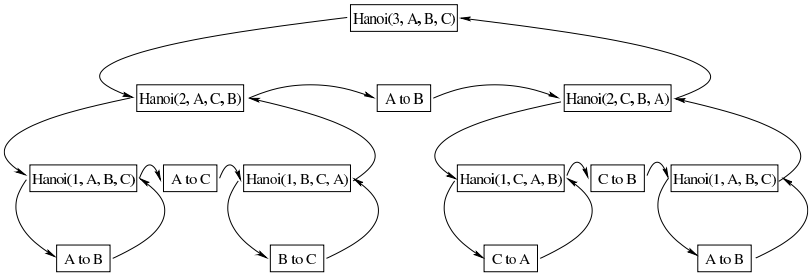
– And then all that's left to do is to move the other disks to B, perhaps using A. (The fact that, at this point, the largest disk will be on B doesn't matter. It can be safely ignored because we are allowed to put on top of it any disk we like.)

The last three bullet points (two of which are recursive) form the heart of the solution. We just add a base case and we're done.

Algorithm: HANOI(n, *source*, *dest*, *spare*)

```
{   if n = 1
    {   Move disk from source to dest;
    }
    else
    {   HANOI(n − 1, source, spare, dest);
        Move top disk from source to dest;
        HANOI(n − 1, spare, dest, source);
    }
}
```

Here's a trace of what happens when we run the algorithm for $n = 3$.



Summary

- We have seen that recursion occurs naturally. They key to writing recursive algorithms is to look for this natural recursion in the data and/or in the task — much as we have been doing with the examples above.
- The base case(s) take care of simple tasks on simple pieces of data. The recursive case(s) take care of more complex tasks on more complex data. One key idea is to make sure that the parameters to recursive calls are always simpler, e.g. smaller. That is what brings the process closer to the base case.

5.4 Iterative versus Recursive Solutions

- We have given a recursive solution to the Towers of Hanoi problem for n disks ($n > 0$). But an iterative solution, that uses **while** loops in place of recursion, is also possible.

  In this solution, pegs A, B and C are arranged in clockwise order. The solution is based on the following observations:

  – At any stage, the smallest disk can always be moved.

  – At any stage, there will be at most one other possible move, not involving the smallest disk.

  This is the Buneman-Levy iterative solution to the Towers of Hanoi problem:

```
while all disks are not on the destination peg
{   Move smallest disk clockwise;
    Make the only possible move that doesn't involve
        the smallest disk;
}
```

Acknowledgements

Explanation and diagrams for the Towers of Hanoi are based on [Har92]. Chapter 55 of [Dew93] contains the Buneman-Levy iterative solution to the Towers of Hanoi problem, along with another iterative solution, credited to T.R. Walsh.

References

[Dew93] A. K. Dewdney. *The (New) Turing Omnibus*. W.H. Freeman, 1993.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.