Lecture 6:
Recursive Algorithms

Aims:

- To consider the difference between recursive and iterative versions of the same algorithm; and
- To look at recursive algorithms on finite lists.

6.1. Recursion *vs.* Iteration

- We saw both an iterative and a recursive solution to the Towers of Hanoi problem. So what? Well, here's a comparison of iterative and recursive approaches.
  - Neither brings more computational power. Any algorithm that uses sequence, conditional and recursion can be reduced to one that uses sequence, conditional and at least one form of unbounded iteration. And vice versa: any algorithm that uses sequence, conditional and at least one of the forms of unbounded iteration can be reduced to one that uses sequence, conditional and recursion.

    What's more, nothing else is needed. Anything that can be computed can be computed using sequence, conditional and recursion. Anything that can be computed can be computed using sequence, conditional and at least one of the forms of unbounded iteration. Hence, sequence, conditional and recursion are, in some sense, a maximally powerful set of control flow constructs. Equally, sequence, conditional and at least one form of unbounded iteration also form a maximally powerful set of control flow constructs.
  - Recursive algorithms are often more elegant. For this reason, they are often easier to write (once you get used to writing them), especially when they access or modify recursively-defined data.
  - Recursive algorithms generally have greater time and space overheads. Whenever a procedure is called, overheads are incurred. Firstly, an amount of memory is needed. We need to store:
    - A *return address*. This is the place in the algorithm from where the procedure was called. When the procedure finishes, the program will resume execution from that point in the program.
    - Data used by the procedure. In CS2205, we don't need to go into the fine details of exactly what data will be stored. (Here, for the benefit of those of you who *are* interested is a list of what needs to be stored: the procedure's

local variables, the procedure's *pass-by-value* parameters, and pointers to the procedure's *pass-by-reference* parameters.)

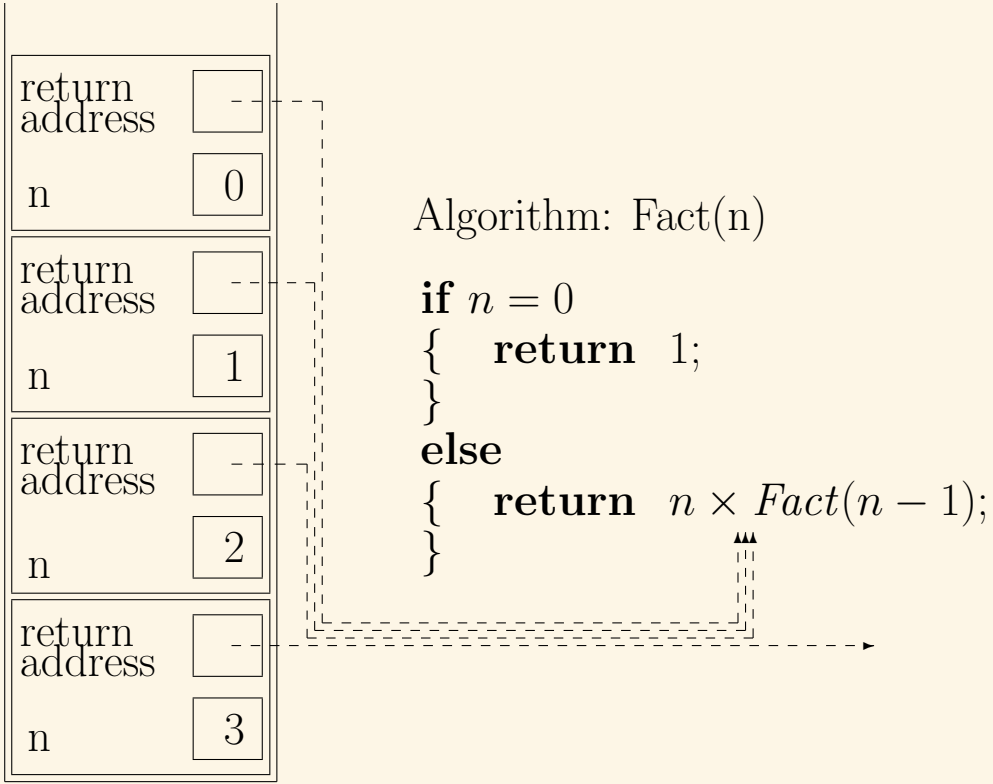
We collect all this together in a block of memory, which we call an *activation record*. We place activation records on a *stack*. (Why?)

This explains the memory overhead of using procedures in general, and recursion in particular.

There is also a time overhead. When a procedure is called, it takes time to create an activation record and put it on the stack. When a procedure finishes, it takes time to remove the activation record from the stack and resume execution from the return address stored in that activation record.

To obtain an appreciation for how much overhead there is, we will compare the execution of a recursive factorial algorithm with an iterative factorial algorithm.

Recursive Factorial



Iterative Factorial

Algorithm: FACT(n)

```
{   answer := 1;
    while n > 1
    {   answer := n × answer;
        n := n − 1;
    }
    return answer;
}
```

n	~~3~~ ~~2~~ 1
answer	~~1~~ ~~3~~ 6

6.2. Finite Lists

- A list is a finite sequence of zero, one or more values typically of the same type. So we might have a list of integers, a list of Booleans, a list of playing cards, a list of arrays of integers, a list of sets or even a list of lists.
- This is our notation for lists:

$$[1, 2, 3]$$

$$[4, 1, 7]$$

$$[\text{“hi”}, \text{“bye”}]$$

$$[[1, 2], [1, 3, 2], [4, 3]]$$

- A list might contain only one value:

$$[\text{“hi”}]$$

- Or a list might even contain no values:

$$[]$$

  in which case, we call it the *empty list*.

- Note that duplicates are allowed.

$$[1, 2, 3, 2]$$

- Note that order is important. So the next two are different lists:

$$[a, b, c]$$

$$[c, b, a]$$

- We will allow ourselves some obvious abbreviations, where necessary, e.g.:

$$[1 \ldots 25]$$

$$[2, 4, 6, \ldots, 100]$$

- That was an informal description of lists. Here's a recursive definition:

  Base case: The *empty list*, written [], is a list.

  Recursive case: If x is a value and L is a list of values, then cons(x, L), the result of adding x onto the front of L, is a list.

  Closure: Nothing else is a list

- So here are some lists, with the more conventional notation shown on the right:

$$\begin{array}{cc} [] & [] \\ \text{cons}(a, []) & [a] \\ \text{cons}(a, \text{cons}(b, [])) & [a, b] \\ \text{cons}(a, \text{cons}(b, \text{cons}(c, []))) & [a, b, c] \end{array}$$

- Let's assume we have available some primitive operations that we can perform on lists:
  - cons(x, L): as above (for sticking a value x on the front of list L)
  - isEmpty(L): returns **true** if L is the empty list, **false** otherwise, e.g.:

$$\text{isEmpty}([1, 2, 3]) = \textbf{false}$$

$$\text{isEmpty}([]) = \textbf{true}$$

  - head(L): returns the first element of L (assuming L is not empty), e.g.:

$$\text{head}([a, b, c]) = a$$

– tail(L): returns all but the first element of L (assuming L is not empty), e.g.:

$$\text{tail}([a,b,c]) = [b,c]$$
$$\text{tail}([a,b]) = [b]$$
$$\text{tail}([a]) = []$$

- So now let's write some recursive algorithms using just cons, head, tail and isEmpty. First, we'll show how to recursively compute the length of a list.

Algorithm: LENGTH(L)

```
{   if isEmpty(L)
    {   return 0;
    }
    else
    {   return 1 + LENGTH(tail(L));
    }
}
```

Note how this exploits the recursive structure of the data (as per the recursive definition of a list, given earlier).

- In the next example, we decide whether x is a member of L:

Algorithm: isMember(x, L)

Parameters: A value x and a list, L.

Returns: **true** if x is a member of L, **false** otherwise.

```
{   if isEmpty(L)
    {   return false;
    }
    else if x = head(L)
    {   return true;
    }
    else
    {   return isMember(x, tail(L));
    }
}
```

There are two base cases here.

- Finally, here's a more challenging example. The idea is to append two lists, in other words stick the second onto the first. For example, if we take $[a, b, c]$ and we append $[d, e]$, we get $[a, b, c, d, e]$.

Question: why can't we just do this: cons($[a, b, c], [d, e]$)?

Algorithm: APPEND(L_1, L_2)

```
{   if isEmpty(L1)
    {   return L2;
    }
    else
    {   return cons(head(L1), APPEND(tail(L1), L2));
    }
}
```

Here we had two lists. A temptation is to try to recurse on both of them. Sometimes this might be necessary. Here it wasn't. We exploited the recursive structure of L_1, leaving L_2 unchanged.

- Before leaving the subjects of lists, I should say a few words abut how lists are created, accessed and modified in conventional programming languages, such as Java. In such languages, lists are not defined in this beautifully simple recursive way. In CS2201, for example, lists are called 'sequences' and they offer the following operations (and many more besides): `first`, `last`, `insertFirst`, `insertBefore`, `atRank`, etc., etc. The Java library also has lists and they offer the following operations (and many more besides): `add`, `contains`, `get`, `size`, etc., etc. In languages such as Java, lists/sequences are usually implemented using node-and-pointer structures.

Acknowledgements

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.