>>

Recursion

∧ >>

❖ Recursion

Recursion is a powerful problem-solving strategy

- employing a variant of divide-and-conquer
- leading to simple, elegant solutions

It is related to *induction* in mathematics, which has

- a base case (a problem instance where the solution is trivial)
- an inductive step (build solution from a simpler version of the problem)

❖ Example #1: factorial

A simple example: computing factorial (**n!**)

- base case: **n** is 1 ⇒ **n**! is 1
- for larger values:
  - I can't solve the whole problem directly
  - but I do know the value of **n**
  - I could compute (**n**-1)! (easier than **n**! ?)
- multiply **n** by (**n**-1)!, giving **n**!

E.g.

```
factorial(3) = 3 * factorial(2)
             = 3 * (2 * factorial(1))
             = 3 * (2 * 1) = 6
```

<< ^ >>

❖ ... Example #1: factorial

Expressing this as a C function:

```
int factorial(int n) {
   if (n == 1)    // base case
      return 1;
   else           // recursive case
      return n * factorial(n-1);
}
```

compared to iterative version

```
int factorial(int n) {
   int fac = 1;
   for (int i = 1; i <= n; i++)
      fac = fac * i;
   return fac;
}
```

<< ∧ >>

❖ Example #2: Summing values in a list

Another simple example: summing integer values in a list

- base case: empty list ⇒ sum is zero
- for larger lists:
  - I can't solve the whole problem directly
  - but I do know the first value in the list
  - sum the rest of the list (smaller than whole list, easier?)
- add first value to sum-of-rest, giving sum of whole

E.g.

```
sum [1,2,3] = 1 + sum [2,3]
            = 1 + (2 + sum [3])
            = 1 + (2 + (3 + sum []))
            = 1 + (2 + (3 + 0)) = 6
```

❖ ... Example #2: Summing values in a list

Expressing previous method as an (abstract) function

```
int sum(List L) {
   if (empty(L))
      return 0;
   else {
      int first, sumRest;
      first = head(L);
      sumRest = sum(tail(L));
      return first + sumRest;
   }
}
```

❖ ... Example #2: Summing values in a list

And then expressing using typical list data structure:

```
struct Node { int val; struct Node *next; };

int sum(struct Node *L) {
   if (L == NULL)
      return 0;
   else {
      int first, sumRest;
      first = L->val;
      sumRest = sum(L->next);
      return first + sumRest;
   }
}

or

int sum(struct Node *L) {
   if (L == NULL)
      return 0;
   else
      return L->val + sum(L->next);
}
```

<< ^ >>

❖ How it works

Recursion is a function calling itself

Won't the system get confused?

No, because each call to the function is a separate instance

- each function call creates a new mini-environment
- this holds all of the data needed by the function

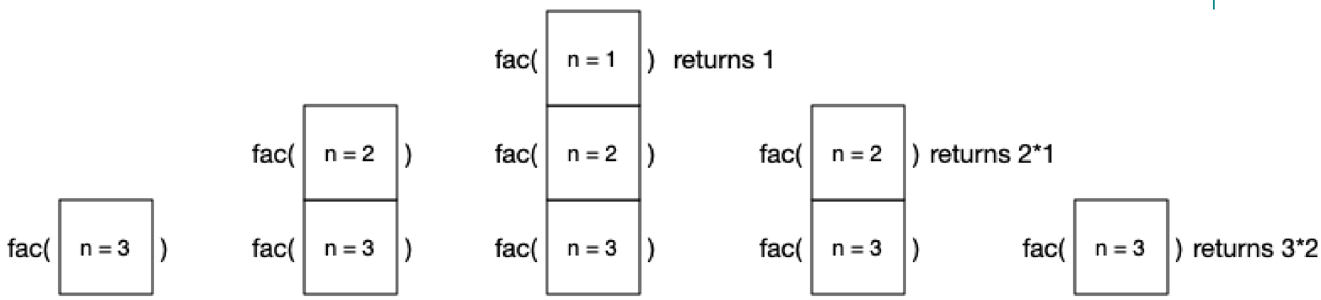
The "mini-environments" are called stack frames

- they are created as part of the function call
- they are removed when the function **`return`**s

<< ^ >>

❖ ... How it works

How the memory state changes during execution



State of stack during recursive evaluation of fac(3)

fac(3) calls fac(2) calls fac(1) returns 1 returns 2*1 returns 3*2 returns 6

<< ^ >>

❖ Using Recursion

While it is useful to know how it works ...

Sometimes it is confusing to think about stacks, etc.

When designing (or reading) recursive functions

- return to recursion basics
- identify the base case
- see how the problem can be reduced
- see how results can be built from base + recursive case

COMP2521 has many examples of recursively-defined algorithms

<< ^

❖ Postscript

Generally, recursive solutions are efficient (except stack space)

Sometimes, they can be very inefficient, e.g.

```
// returns n'th fibonacci number
int fibonacci(int n) {
   if (n == 1)
      return 1;
   else if (n == 2)
      return 1;
   else
      return fibonacci(n-1) + fibonacci(n-2);
}
```

Trace the recursive calls for **`fibonacci(5)`** to see the problem

Produced: 2 Jun 2020