ICS103 Programming in C

Lecture 11: Recursive Functions

Outline

- Introducing Recursive Functions
- Format of recursive Functions
- Tracing Recursive Functions
- Examples
- Tracing using Recursive Trees

Introducing Recursive Functions

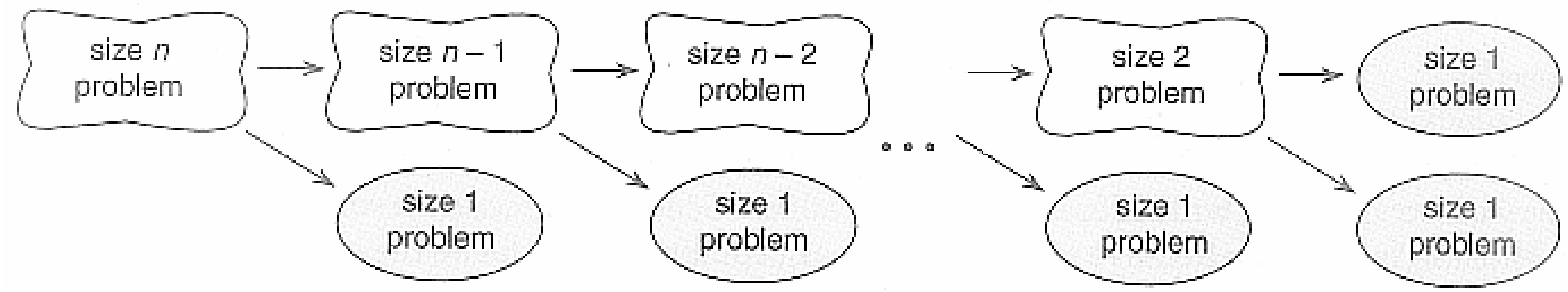
- We have seen so far that a function, such as *main*, can call another function to perform some computation.
- In C, a function can also call itself. Such types of functions are called recursive functions. A function, *f*, is also said to be recursive if it calls another function, ***g***, which in turn calls ***f***.
- Although it may sound strange for a function to call itself, it is in fact not so strange, as many mathematical functions are defined recursively.
  - For example, the factorial function is defined mathematically as:

$$n! = \begin{cases} 1, & n = 0 \\ n\,(n-1)!\,, & n>1 \end{cases}$$

- Although less efficient than iterative functions (using loops) due to overhead in function calls, in many cases, recursive functions provide a more natural and simple solutions.
- Thus, recursion is a powerful tool in problem solving and programming.

Introducing Recursive Functions ...

- Problems that can be solved using recursion have the following characteristics:
  - One or more **simple cases** of the problem have a direct and easy answer – also called base cases. Example: 0! = 1.
  - The other cases can be re-defined in terms of a similar but smaller problem - **recursive cases**. Example: n! = n (n-1)!
  - By applying this re-definition process, each time the recursive cases will **move closer** and eventually reach the base case. Example: n! → (n-1)! → (n-2)! → . . . 1!, 0!.
- The strategy in recursive solutions is called divide-and-conquer. The idea is to keep reducing the problem size until it reduces to the simple case which has an obvious solution.



Format of recursive Functions

- Recursive functions generally involve an if statement with the following form:

  if this is a simple case
    solve it
  else
    redefine the problem using recursion

- The if branch is the base case, while the else branch is the recursive case.
- The recursive step provides the repetition needed for the solution and the base step provides the termination
- **Note:** For the recursion to terminate, the recursive case must be moving closer to the base case with each recursive call.

Example 1: Recursive Factorial

- The following shows the recursive and iterative versions of the factorial function:

Recursive version

```
int factorial (int n)
{
   if (n == 0)
     return 1;
   else
     return  n * factorial (n-
1);
}
```

Recursive Call

Iterative version

```
int factorial (int n)
{
   int i, product=1;
   for (i=n; i>1; --i)
       product=product * i;

   return product;
}
```

The complete recursive multiply example

```
/* Computes the factorial of a number  */
#include <stdio.h>
int factorial(int n);

/* shows how to call a user-define function */
int main(void) {
  int num, fact;
  printf("Enter an integer between 0 and 7> ");
  scanf("%d", &num);
  if (num < 0) {
    printf("Factorial not defined for negative
     numbers\n");
  } else if (num <= 7) {
    fact = factorial(num);
    printf("The factorial of %d is %d\n", num, fact);
  } else {
    printf("Number out of range: %d\n", num);
  }

  system("pause");
  return (0);
}
```

```
/* Computes n! for n greater than or equal
      to zero */
int factorial (int n)
{
    if (n == 0) //base case
       return 1;
    else
      return  n * factorial (n-1); //recursive
     case
}
```

Tracing Recursive Functions

- Executing recursive algorithms goes through two phases:
  - Expansion in which the recursive step is applied until hitting the base step
  - "Substitution" in which the solution is constructed backwards starting with the base step

$$
\begin{aligned}
\text{factorial}(4) \quad &= 4 * \text{factorial}\ (3)\\
&= 4 * (3 * \text{factorial}\ (2))\\
&= 4 * (3 * (2 * \text{factorial}\ (1)))\\
&= 4 * (3 * (2 * (1 * \text{factorial}\ (0))))
\end{aligned}
$$

Expansion phase

$$
\begin{aligned}
&= 4 * (3 * (2 * (1 * 1)))\\
&= 4 * (3 * (2 * 1))\\
&= 4 * (3 * 2)\\
&= 4 * 6\\
&= 24
\end{aligned}
$$

Substitution phase

Example 2: Multiplication

- Suppose we wish to write a recursive function to multiply an integer m by another integer n using addition. [We can add, but we only know how to multiply by 1].
- The best way to go about this is to formulate the solution by identifying the base case and the recursive case.
- The base case is if n is 1. The answer is m.
- The recursive case is: m*n = m + m (n-1).

$$m*n \begin{cases} m, & n = 1 \\ m + m\,(n-1), & n>1 \end{cases}$$

Example 2: Multiplication …

```
#include <stdio.h>

int multiply(int m, int n);

int main(void) {
   int num1, num2;

   printf("Enter two integer numbers to multiply: ");
   scanf("%d%d", &num1, &num2);

   printf("%d x %d = %d\n", num1, num2, multiply(num1, num2));
   system("pause");
   return 0;
}

int multiply(int m, int n) {
     if (n == 1)
          return m;     /* simple case */
     else
          return m + multiply(m, n  - 1);  /* recursive step */
}
```

Example 2: Multiplication …

Expansion phase

multiply(5,4) = 5 + multiply(5, 3)
= 5 + (5 + multiply(5, 2))
= 5 + (5 + (5 + multiply(5, 1)))

= 5 + (5 + (5 + 5))
= 5 + (5 + 10)
= 5 + 15
= 20

Substitution phase

Example 3: Power function

- Suppose we wish to define our own power function that raise a double number to the power of a non-negative integer exponent. x^n , n>=0.
- The base case is if n is 0. The answer is 1.
- The recursive case is: $x^n = x * x^{n-1}$.

$$
\mathbf{x^n} \begin{cases} 1, & n = 0 \\ x * x^{n-1}, & n>0 \end{cases}
$$

Example 3: Power function …

```
#include <stdio.h>

double pow(double x, int n);

int main(void) {
    double x;
    int n;

    printf("Enter double x and integer n to find pow(x,n): ");
    scanf("%lf%d", &x, &n);

    printf("pow(%f, %d) = %f\n", x, n, pow(x, n));
    system("pause");
    return 0;
}

double pow(double x, int n) {
       if (n == 0)
            return 1;     /* simple case */
      else
            return x * pow(x, n  - 1);  /* recursive step */
}
```

Example 4: Fibonacci Function

- Suppose we wish to define a function to compute the nth term of the Fibonacci sequence.
- Fibonacci is a sequence of number that begins with the term 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms:
- Thus, the sequence is: 0, 1, 1,2,3,5,8,13,21,34 …
- Mathematically, the sequence can be defined as:

$$\textbf{fib(n)} \begin{cases} n, \quad n = 0, 1 \\ \text{fib(n-1) + fib(n-2)} \quad n>1 \end{cases}$$

Example 4: Fibonacci Function …

```
#include <stdio.h>

int fib(int n);

int main(void) {
    int n;

    printf("Enter an integer n to find the nth fibonacci term: ");
    scanf("%d", &n);

    printf("fibonacci(%d) = %d\n", n, fib(n));
    system("pause");
    return 0;
}

int fib(int n) {
      if (n == 0 || n== 1)
            return n;     /* simple case */
      else
            return fib(n-1) + fib(n-2);  /* recursive step */
}
```

Tracing using Recursive Tree

- Another way to trace a recursive function is by drawing its recursive tree.
- This is usually better if the recursive case involves more than one recursive calls.