Review of Recursion

Courtesy of Dr. Chuck Cusack

CSCE310: Data Structures and Algorithms
www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444
Recursion

- A subroutine/function that calls itself is called *recursive*
- Warning: A subroutine/function that simply calls itself would result in infinite recursion
- Thus, when using recursion, one must ensure that, at some point, the subroutine/function terminates without calling itself

Two (2) fundamental rules of recursion:

1. **Base case(s)**, solved without recursion.
   A.k.a Stopping case, Terminating condition.

2. **Making progress**. Recursive calls must always be to a case that makes progress toward a base case.
   A.k.a inductive case(s) always progress toward a base case

Two other rules (later)
RecursiveFactorial(n)
    IF (n <= 1) THEN RETURN 1
    RETURN n * Factorial(n-1)

What ensures that the function RecursiveFactorial terminates?
Where is Recursion Seen/Used?

In problems that can be solved by combining solutions of smaller instances of the given problem.

Toy problems: Russian Matryoshka (nested dolls), Tower of Hanoi.

CS examples: binary search, mergesort, $n!$, Fibonacci numbers, divide-&-conquer algorithms.
Iteration vs. Recursion: examples

Iterative count down:

CountDown (n)
WHILE i > 0
    DO Print (n)
        n <-- (n - 1)

Recursive count down: Incorrect

CountDown (n)
    Print(n)
    CountDown(n-1)

What is wrong?

Recursive count down: Fixed

CountDown (n)
    IF n > 0
        THEN Print(n)
            CountDown(n-1)
Recursion and Memory

• Each function call generates a function instance
• An instance of a function contains
  – memory for each parameter (input)
  – memory for each local variable
  – memory for the return value
This chunk of memory is referred to as activation record
• A recursive function that calls itself \( n \) times must allocate \( n \) activation records
• Usually, an iterative implementation will require on the order of one activation record, plus a constant amount of space
• Space: reason recursion for avoiding recursion
• Recursion simplifies code readability (good compilers remove recursion whenever possible)
Example:

- RecursiveFactorial(4)

```
return value
```

Activation record for factorial(4)

- RecursiveFactorial(4): call to

RecursiveFactorial(3)

```
return value
```

Activation record for factorial(3)

```
return value
```

Activation record for factorial(4)
• **RecursiveFactorial(4): call to RecursiveFactorial(2)**

```
n  n  n  n
2  3  4
```

return value  return value  return value  return value

Activation record for factorial(2)  Activation record for factorial(3)  Activation record for factorial(4)

• **RecursiveFactorial(4): call to RecursiveFactorial(1)**

```
n  n  n  n
1  2  3  4
```

return value  return value  return value  return value

Activation record for factorial(1)  Activation record for factorial(2)  Activation record for factorial(3)  Activation record for factorial(4)
• RecursiveFactorial(4):
  RecursiveFactorial(1) is the base case, so it returns ‘1’.

  Activation record for factorial(1)

  Activation record for factorial(2)

  Activation record for factorial(3)

  Activation record for factorial(4)
• RecursiveFactorial(4):
  RecursiveFactorial(2) now returns ‘2’.

  n
  return value
  2
  Activation record for factorial(2)

  n
  return value
  3
  Activation record for factorial(3)

  n
  return value
  4
  Activation record for factorial(4)
• RecursiveFactorial(4):
  RecursiveFactorial(3) now returns ‘6’.

\[
\begin{array}{c|c}
 n & 3 \\
\hline
 \text{return value} & 6 \\
\end{array}
\]

  Activation record for factorial(3)

\[
\begin{array}{c|c}
 n & 4 \\
\hline
 \text{return value} & \\
\end{array}
\]

  Activation record for factorial(4)

• RecursiveFactorial(4): returns ‘24’. This was the original function call, so the execution is finished.

\[
\begin{array}{c|c}
 n & 4 \\
\hline
 \text{return value} & 24 \\
\end{array}
\]

  Activation record for factorial(4)
The Run-Time Stack

- In order to support recursive function calls, the run-time system treats memory as a stack of activation records

- `$\text{RecursiveFactorial}(n)$` requires the allocation of $n$ activation records on the stack
  
  ```
  \text{RecursiveFactorial}(n)
  \text{IF (n} =< 1) \text{THEN 1}
  \text{RETURN n} \ast \text{RecursiveFactorial}(n-1)
  ```

- What if we have infinite recursion:
  
  ```
  \text{InfiniteRecursion (n)}
  \text{IF (n}=0) \text{THEN 1}
  \text{ELSE InfiniteRecursion(n)}
  ```

  The value of $n$ never reaches zero, so the function is called, and records are pushed onto the stack, until the system runs out of memory.

- Even if our recursion is not infinite, system may run out of memory: the recursion may be too deep, and since memory is finite
Recursion vs. iteration

Recursive functions can be translated to functions that use loops.

- **Recursive:**
  
  \[
  \text{RecursiveFactorial}(n) = \begin{cases} 
  1 & \text{IF } (n < 1) \\
  n \times \text{RecursiveFactorial}(n-1) & \text{IF } (n \geq 1)
  \end{cases}
  \]

- **Iterative:**
  
  \[
  \text{IterativeFactorial}(n) = \begin{cases} 
  1 & \text{WHILE } (n \geq 1) \\
  \text{result} \leftarrow \text{result} \times n & \text{result} \leftarrow 1
  \end{cases}
  \]

RETURN result
Memory Usage?

For input $n$:

**Recursive implementation** needs to allocate $2n$ integers

**Iterative implementation** needs only 3
Recursion: Disadvantages

Each time one function calls another, the computer’s operating system must:

- Record how to re-start the calling function later on,
- Pass the parameters from the calling function to the called function (often by pushing the parameters onto a stack controlled by the system)
- Set up space for the called function’s local variables
- Record where the calling function’s local variables are stored

Doing all this requires time and memory.
Common recursion errors

- Forgetting or having incomplete base cases

  \begin{verbatim}
  Sum1toN(N)
  IF (N == 0)
    THEN RETURN 0
  ELSE RETURN (N + Sum1toN(N-1))
  \end{verbatim}

- Getting things backwards

  \begin{verbatim}
  CountHow(n)
  IF n > 0
    THEN Print(n)
    CountHow(n-1)
  CountGuess(n)
  IF n > 0
    THEN CountGuess(n-1)
    Print(n)
  \end{verbatim}
Two (2) fundamental rules of recursion:

1. Base case(s)
2. Making progress

Two other rules:

1. **Design rule.** Assume all recursive calls work.
   Don’t do the bookkeeping yourself.

2. **Compound interest rule.**
   Don’t duplicate work by solving the same instance of a problem in separate recursive calls.
**Recursion:** Conclusion

- A recursive function is one that invokes another instance of itself
- Recursion is an alternative to iteration
- Recursion often provides a more elegant solution than iteration
- Each instance of a function has its own set of local variables and parameters
- Recursive solutions are often less efficient, in terms of time and space, than an iterative solution.
- Some problems are difficult to solve without recursion. Particularly when the problem is recursive in its definition, example Tower of Hanoi.
Exercise 11.4-2 (not tested)

Print-node (node)

IF (left-child[\text{node}] = \text{nil}) AND (right-sibling[\text{node}] = \text{nil})

THEN print(key[\text{node}])

ELSE IF left-child[\text{node}] = \text{nil}

THEN print-node (left-child[\text{node}])

ELSE IF right-sibling[\text{node}] = \text{nil}

THEN print-node (right-sibling[\text{node}])

ELSE print-node (right-child[\text{node}])

print-node (left-child[\text{node}])

print-node (right-sibling[\text{node}])
Common functions

Floors and ceilings:

- The floor of $x$: $\lfloor x \rfloor$
- The ceiling of $x$: $\lceil x \rceil$
- $\forall n \in \mathbb{N}, \lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

Logarithms:

- Binary logarithm: $\lg n = \log_2 n$
- Natural logarithm: $\ln n = \log_e n$
- $\log_a n = \frac{\ln n}{\ln a}$ and $\log n = \frac{\ln n}{\ln 10}$
- $\ln e = 1$, $\ln 1 = 0$, $\log_k 1 = 0$, $\log_k k = 1$
- Exponentiation: $\lg^k n = (\lg n)^k$
- Composition: $\lg \lg n = \lg(\lg n)$
- $a = b^{\log_b a}$
- $\log_b a^n = n \log_b a$
- $a^{\log_b n} = n^{\log_b a}$

Sum of finite series:

- Arithmetic: $S_n = \frac{n(t_1 + t_n)}{2}$
- Geometric: $S_n = t_1 \frac{r^n - 1}{r - 1} = \frac{t_n r - t_1}{r - 1}$