Lecture P6: Recursion

Overview

What is recursion?
- When one function calls ITSELF directly or indirectly.

Why learn recursion?
- New mode of thinking.
- Powerful programming tool.
- Many computations are naturally self-referential.
  - a Unix directory contains files and other directories
  - Euclid’s gcd algorithm
  - linked lists and trees
  - GNU = GNU’s Not Emacs

Overview

How does recursion work?

How does a function call work?
- A function lives in a local environment:
  - values of local variables
  - which statement the computer is currently executing

- When f() calls g(), the system
  - saves local environment of f
  - sets value of parameters in g
  - jumps to first instruction of g, and executes that function
  - returns from g, passing return value to f
  - restores local environment of f
  - resumes execution in f just after the function call to g

Implementing Functions

How does the compiler implement functions?

Return from functions in last-in first-out (LIFO) order.
- FUNCTION CALL: push local environment onto stack.
- RETURN: pop from stack and restore local environment.
A Simple Example

Goal: function to compute \( \text{sum}(n) = 0 + 1 + 2 + \ldots + n-1 + n \).

- Simple ITERATIVE solution.

\[
\text{iterative sum 1}
\]
```c
int sum(int n) {
    int i, s = 0;
    for (i = 0; i <= n; i++)
        s += i;
    return s;
}
```

Note that changing the variable \( n \) in \text{sum} does not change the value in the calling function.

\[
\text{iterative sum 2}
\]
```c
int sum(int n) {
    int s = n;
    while (n > 0) {
        n--;
        s += n;
    }
    return s;
}
```

A Simple Example

Goal: function to compute \( \text{sum}(n) = 0 + 1 + 2 + \ldots + n-1 + n \).

- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.

This is just a stupid example to illustrate recursion.
- Don’t even need iteration, let alone recursion.
- \( 0 + 1 + 2 + \ldots + n = n(n+1)/2 \)

\[
\text{better sum}
\]
```c
int sum(int n) {
    return (n * (n+1)) / 2;
}
```

A Bad Recursive Function

BASE CASE is special input for which the answer is trivial.
- Won’t "bottom-out" of recursion without a base case.
- Analog of infinite loops with for and while loops.

\[
\text{mystery1(n)}
\]
```c
void mystery1(int n) {
    if (n == 0)
        return 0;
    else
        return n + sum(n-1);
}
```
A Bad Recursive Function

BASE CASE is special input for which the answer is trivial.

REDUCTION STEP makes input converge to base case.
- Unknown whether program terminates for all positive integers n.
- Stay tuned for Halting Problem in Lecture T4.

```c
void mystery2(int n) {
    printf("%d\n", n);
    if (n <= 1)
        return;
    else if (n % 2 == 0)
        mystery2(n/2);
    else
        mystery2(3*n + 1);
}
```

Greatest Common Divisor

Find largest integer d that evenly divides into m and n.

\[
gcd(m, n) = \begin{cases} 
m & \text{if } n = 0 \\
gcd(n, m \mod n) & \text{otherwise} \\
\end{cases}
\]

Euclid (300 BC)

\[
gcd(1440, 408) = gcd(408, 216) = gcd(192, 24) = gcd(24, 0) = 24.
\]

\[
1440 = 2^5 \times 3^2 \times 5^1 \\
408 = 2^3 \times 3^1 \times 17^1
\]

Number Conversion

To print binary representation of integer N:
- Stop if N = 0.
- Write ‘1’ if N is odd; ‘0’ if n is even.
- Move pencil one position to left.
- Print binary representation of \( N / 2 \).

(integer division)

Check: 43 = \( 32 + 0 + 8 + 0 + 2 + 1 \)

Easiest way to compute by hand.
- Corresponds directly with a recursive program.
Recursive Number Conversion

Computer naturally prints from left to right.
- So we need to first convert \( N / 2 \).
- Then write '0' or '1'.

\[
N = 2 \times (N / 2) + (N \% 2)
\]

```c
void convert(int N) {
    if (N == 0)
        return;
    convert(N / 2);
    printf("%d", N % 2);
}
```

**proof of correctness:**

1 if \( N \) is odd; 0 if \( N \) is even

```c
convert(43)
convert(21)
convert(10)
convert(5)
convert(2)
convert(1)
convert(0)
printf("1")
printf("0")
printf("1")
printf("0")
printf("1")
printf("1")
```

Convert to any base \( b \leq 10 \).

Exercise: extend to handle hexadecimal (base 16).

Possible Pitfalls With Recursion

Is recursion fast?

Fibonacci numbers:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... 

\[
F_n = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F_{n-1} + F_{n-2} & \text{otherwise}
\end{cases}
\]

It takes a really long time to compute \( F(40) \).

```c
int F(int n) {
    if (n == 0 || n == 1)
        return n;
    else
        return F(n-1) + F(n-2);
}
```

bad Fibonacci function

331,160,281 function calls for \( F(40) \).
Possible Pitfalls With Recursion

Recursion can take a long time if it needs to repeatedly recompute intermediate results.

- DYNAMIC PROGRAMMING solution: save away intermediate results in a table.

```c
int knownF[1000] = {0};
int F(int n) {
    if (knownF[n] != 0)
        return knownF[n];
    else if (n == 0 || n == 1)
        return n;
    knownF[n] = F(n-1) + F(n-2);
    return knownF[n];
}
```

Fibonacci using dynamic programming

Stores \(n\)th Fibonacci number in \(n\)th element.

Uses only 2n recursive calls to compute \(F(n)\).

Recursion vs. Iteration

Fact 1. Any recursive function can be written with iteration.
   - Compiler implements recursion with stack.
   - Can avoid recursion by explicitly maintaining a stack.

Fact 2. Any iterative function can be written with recursion.

Should I use iteration or recursion?
   - Ease and clarity of implementation.
   - Time/space efficiency.

Towers of Hanoi

Move all the discs from the leftmost peg to the rightmost one.

- Only one disc may be moved at a time.
- A disc can be placed either on empty peg or on top of a larger disc.

\[ \text{Start} \quad \text{Goal} \]

Edouard Lucas (1883)

Towers of Hanoi: Recursive Solution

Move N-1 smallest discs to pole B.
Move largest disc to pole C.
Move N-1 smallest discs to pole C.
Towers of Hanoi: Recursive Solution

```c
#include <stdio.h>

void hanoi(int n, char from, char to) {
    char temp;
    if (n == 0)
        return;
    temp = getOtherPeg(from, to);
    hanoi(n-1, from, temp);
    printf("Move disc %d from %c to %c.\n", n, from, to);
    hanoi(n-1, temp, to);
}

int main(void) {
    hanoi(4, 'A', 'C');
    return 0;
}
```

Unix

```bash
% gcc hanoi.c
% a.out
Move disc 1 from A to B.
Move disc 2 from A to C.
Move disc 1 from B to C.
Move disc 3 from A to B.
Move disc 1 from C to A.
Move disc 2 from C to B.
Move disc 1 from A to B.
Move disc 4 from A to C.
Move disc 1 from B to C.
Move disc 2 from B to A.
Move disc 1 from C to A.
Move disc 3 from B to C.
Move disc 1 from A to B.
Move disc 2 from A to C.
Move disc 1 from B to C.
```

Towers of Hanoi

Is world going to end (according to legend)?
- Monks have to solve problem with \( N = 40 \) discs.
- Computer algorithm should help.

Better understanding of recursive algorithm supplies non-recursive solution!
- Alternate between two moves:
- See Sedgewick 5.2.

Summary

How does recursion work?
- Just like any other function call.

How does a function call work?
- Save away local environment using a stack.

Trace the executing of a recursive program.
- Use pictures.

Write simple recursive programs.
- Base case.
- Reduction step.
Root Finding

Given a function, find a root, i.e., a value $x$ such that $f(x) = 0$.

1. $f(x) = x^2 - x - 1$
2. $\phi = \frac{1 + \sqrt{5}}{2} = 1.61803...$ is a root.

Assume $f$ is continuous and $l, r$ satisfy $f(l) < 0.0$ and $f(r) > 0.0$.

Root Finding

Reduction step:

- Maintain interval $[l, r]$ such that $f(l) < 0, f(r) > 0$.
- Compute midpoint $m = (l + r) / 2$.
- If $f(m) < 0$ then run algorithm recursively on interval is $[m, r]$.
- If $f(m) > 0$ then run algorithm recursively on interval is $[l, m]$.

Progress achieved at each step:
- Size of interval is cut in half.

Base case (when to stop):
- Ideally when $(0.0 == f(m))$, but this may never happen!
  - root may be irrational
  - machine precision issues
- Stop when $(x - 1)$ is sufficiently small.
  - guarantees $m$ is sufficiently close to root

Root Finding

Given a function, find a root, i.e., a value $x$ such that $f(x) = 0$.

```c
#define EPSILON 0.000001

double f (double x) {
    return x*x - x - 1;
}

double bisect (double left, double right) {
    double mid = (left + right) / 2;
    if (right - left < EPSILON || 0.0 == f(mid))
        return mid;
    if (f(mid) < 0.0)
        return bisect(mid, right);
    return bisect(left, mid);
}
```