

Lecture P6: Recursion



Start



Goal

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 Unix



2

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

3

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.

4

A Simple Example

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

- Simple ITERATIVE solution.

iterative sum 1

```
int sum(int n) {
    int i, s = 0;
    for (i = 0; i <= n; i++)
        s += i;
    return s;
}
```

iterative sum 2

```
int sum(int n) {
    int s = n;
    while (n > 0) {
        n--;
        s += n;
    }
    return s;
}
```

Note that changing the variable n in `sum` does not change the value in the calling function.

5

A Simple Example

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

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

$\text{sum}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n + \text{sum}(n-1) & \text{otherwise} \end{cases}$

← base case
← reduction step
converges to base case

recursive sum

```
int sum(int n) {
    if (n == 0)
        return 0;
    return n + sum(n-1);
}
```

← base case

← reduction step

6

A Simple Example

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

- 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.
- $0 + 1 + 2 + \dots + n = n(n+1) / 2$

better sum

```
int sum(int n) {
    return (n * (n+1)) / 2;
}
```

7

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.

mystery1(n)

```
void mystery1(int n) {
    printf("%d\n", n);
    if (n % 2 == 0)
        mystery1(n/2);
    else
        mystery1(3*n + 1);
}
```

← no base case

Is n even? →

8

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.

```

mystery2(n)
void mystery2(int n) {
    printf("%d\n", n);
    if (n <= 1) ← base case
        return;
    else if (n % 2 == 0) ← reduction step
        mystery2(n/2);
    else ← anti-reduction step
        mystery2(3*n + 1);
}
    
```

9

Greatest Common Divisor

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

$$\text{gcd}(m, n) = \begin{cases} m & \text{if } n = 0 \\ \text{gcd}(n, m \% n) & \text{otherwise} \end{cases}$$

← **base case**
← **reduction step**

converges to
base case



Euclid (300 BC)

$$\begin{aligned} \text{gcd}(1440, 408) &= \text{gcd}(408, 216) \\ &= \text{gcd}(216, 192) \\ &= \text{gcd}(192, 24) \\ &= \text{gcd}(24, 0) \\ &= 24. \end{aligned}$$

$$\begin{aligned} 1440 &= 2^5 \times 3^2 \times 5^1 \\ 408 &= 2^3 \times 3^1 \times 17^1 \end{aligned}$$

10

Greatest Common Divisor

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

$$\text{gcd}(m, n) = \begin{cases} m & \text{if } n = 0 \\ \text{gcd}(n, m \% n) & \text{otherwise} \end{cases}$$

← **base case**
← **reduction step**

converges to
base case

```

gcd(m, n)
int gcd(int m, int n) {
    if (n == 0) ← base case
        return m;
    else ← reduction step
        return gcd(n, m % n);
}
    
```

11

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)

$$\begin{aligned} \text{Check: } 43 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 32 + 0 + 8 + 0 + 2 + 1 \end{aligned}$$

Easiest way to compute by hand.

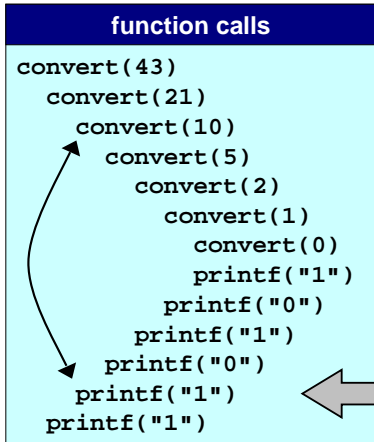
- Corresponds directly with a recursive program.

12

Recursive Number Conversion

Computer naturally prints from left to right.

- So we need to first convert N / 2.
- Then write '0' or '1'.



```

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

Unix

```

% gcc convert.c
% a.out
43
101011
  
```

odd; 0 if N is even

Indentation level pairs statements belonging to same "invocation"

Recursive Number Conversion

Computer naturally prints from left to right.

- So we need to first convert N / 2.
- Then write '0' or '1'.

Proof of correctness:

$$N = 2 * (N / 2) + (N \% 2)$$

```

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

1 if N is odd; 0 if N is even

Convert to any base b ≤ 10.

- Exercise: extend to handle hexadecimal (base 16).

Possible Pitfalls With Recursion

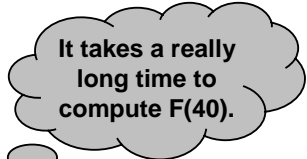
Is recursion fast?



Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

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



bad Fibonacci function

```

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

Possible Pitfalls With Recursion

F(39) is computed once.

F(38) is computed 2 times.

F(37) is computed 3 times.

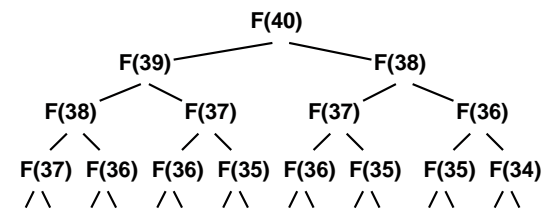
F(36) is computed 5 times.

F(35) is computed 8 times.

...

F(0) is computed 165,580,141 times.

331,160,281 function calls for F(40).



$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

$$= \left\lfloor \phi^n / \sqrt{5} \right\rfloor$$

$$\phi = 1.61803398875$$

bad Fibonacci function

```

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

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.

Fibonacci using dynamic programming

```
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];  
}
```

Stores i^{th} Fibonacci number in i^{th} element.

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

17

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.

18

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.



Start



Goal

Towers of Hanoi demo



Edouard Lucas (1883)

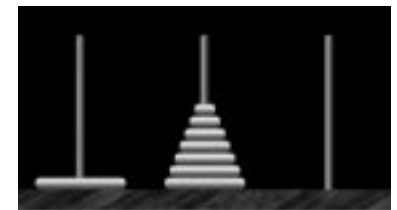
19

Towers of Hanoi: Recursive Solution

A B C



Move N-1 smallest discs to pole B.



Move largest disc to pole C.



Move N-1 smallest discs to pole C.

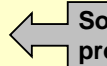
21

Towers of Hanoi: Recursive Solution

```
hanoi.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
% 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.
```



So pr

22

Towers of Hanoi: Recursive Solution

```
hanoi.c
char getOtherPeg(char x, char y) {
    if (x == 'A' && y == 'B') || (x == 'B' && y == 'A')
        return 'C';
    if (x == 'A' && y == 'C') || (x == 'C' && y == 'A')
        return 'B';
    return 'A';
}
```

23

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.

24

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.

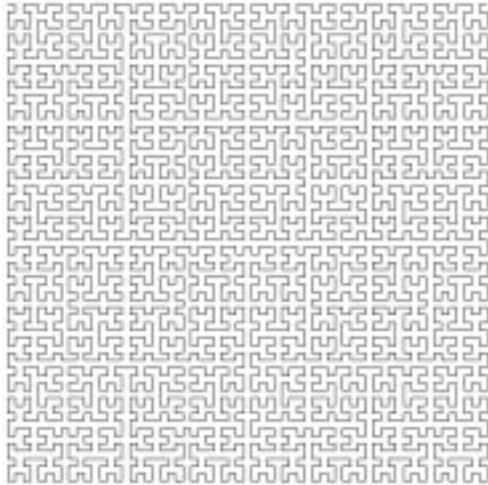
- Use pictures.

Write simple recursive programs.

- Base case.
- Reduction step.

25

Lecture P6: Supplemental Notes

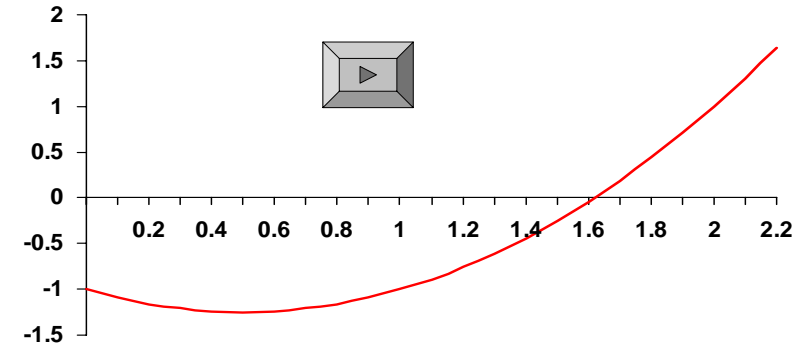


Root Finding

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

- $f(x) = x^2 - x - 1$
- $\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\dots$ 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 $(r - l)$ is sufficiently small.
 - guarantees m is sufficiently close to root

Root Finding

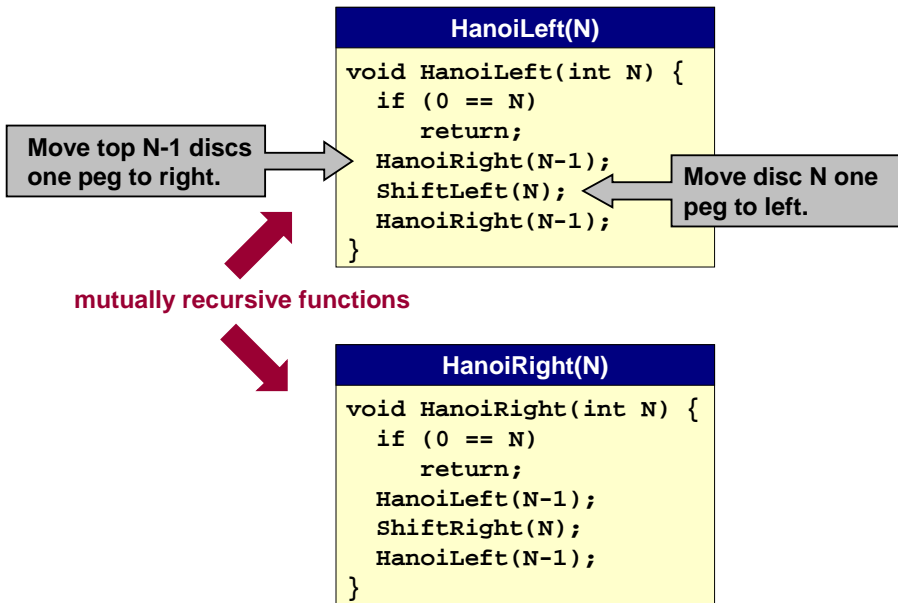
Given a function, find a root, i.e., a value x such that $f(x) = 0$.

```
recursive bisection function
#define EPSILON 0.000001

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

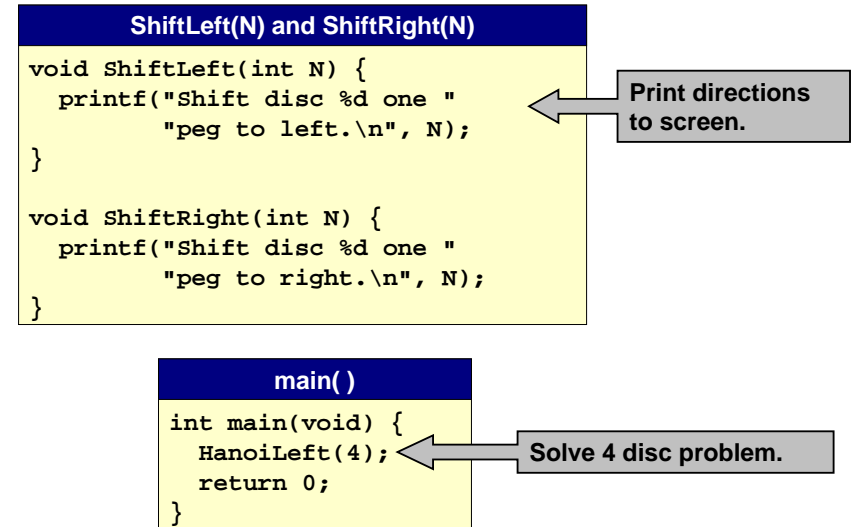
double bisect (double left, double right) {
    double mid = (left + right) / 2;
    if (right - left < EPSILON || 0.0 == f(mid))
        return mid;
    if (f(mid) < 0.0)
        return bisect(mid, right);
    return bisect(left, mid);
}
```

Towers of Hanoi: Alternate Recursive Solution



36

Towers of Hanoi: Alternate Recursive Solution



37