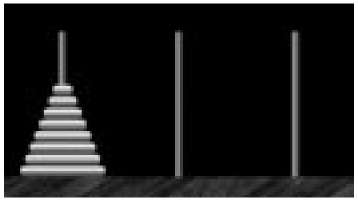
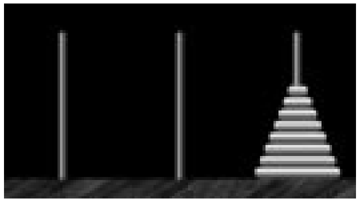
Lecture P6: Recursion



Start



Goal

Overview

What is recursion?

- When one function calls ITSELF directly or indirectly.

Why learn recursion?

- New mode of thinking.
- Powerful programming tool to solve a problem by breaking it up into one (or more) smaller problems of similar structure.
- Many computations are naturally self-referential.
  - a Unix directory contains files and other directories
  - Euclid's gcd algorithm
  - linked lists

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 sum(n) = 0 + 1 + 2 + . . . + 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.

A Simple Example

Goal: function to compute sum(n) = $\underbrace{0 + 1 + 2 + \ldots + n-1}_{\text{sum(n-1)}}$ + 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 (0 == n)
    return 0;
  return n + sum(n-1);
}
```

base case

reduction step

A Simple Example

Goal: function to compute sum(n) = 0 + 1 + 2 + . . . + 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 + . . . + n = n(n+1) / 2

better sum

```
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.

mystery1(n)

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

Is n even?

no base case

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)
     return;
  else if (n % 2 == 0)
     mystery2(n/2);
  else
     mystery2(3*n + 1);
}
```

base case

reduction step

anti-reduction step

Greatest Common Divisor

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

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

base case

reduction step

converges to base case

Euclid (300 BC)

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

$1440 = 2^5 \times 3^2 \times 5^1$

$408 = 2^3 \times 3^1 \times 17^1$

Greatest Common Divisor

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

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

base case

reduction step

converges to base case

gcd(m, n)

```
int gcd(int m, int n) {
  if (0 == n)
    return m;
  else
    return gcd(n, m % n);
}
```

base case

reduction step

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)

43	1
21	11
10	011
5	1011
2	01011
1	101011
0	

Check: $43 = \mathbf{1} \times 2^5 + \mathbf{0} \times 1^4 + \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0$
$= 32 + 0 + 8 + 0 + 2 + 1$

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’.

convert( )

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

odd; 0 if N is even

function calls

```
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")
```

Unix

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

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 $b \leq 10$.

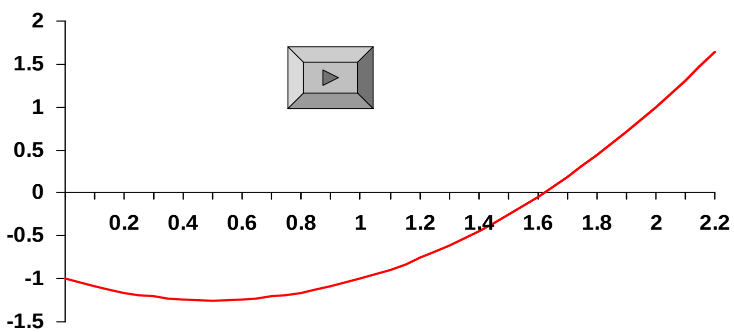
- Exercise: extend to handle hexadecimal (base 16).

Root Finding

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

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


Root Finding

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

- Fundamental problem in mathematics, engineering.
  - to find minimum of a (differentiable) function, need to identify where derivative is zero.
- Faster methods if function is sufficiently smooth.
  - Newton's method.
  - Steepest descent.

Possible Pitfalls With Recursion

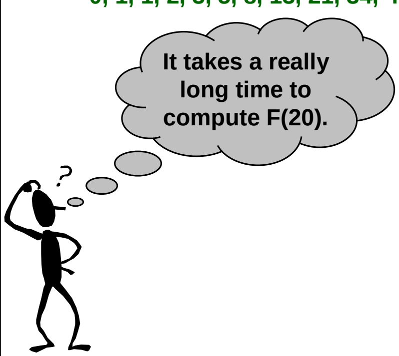
Is recursion fast?

- Yes. We produced remarkably efficient program for exponentiation.
- No. Can easily write remarkably inefficient programs.

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


Possible Pitfalls With Recursion

F(8) is recomputed 2 times.

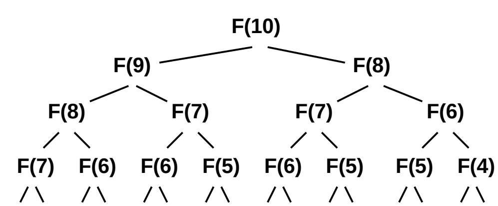
F(7) is recomputed 3 times.

F(6) is recomputed 5 times.

F(5) is recomputed 8 times.

...

F(1) is recomputed 12,555 times.



Requires F(n) recursive calls to compute F(n).

bad Fibonacci function

```
int F(int n) {
  if (0 == n || 1 == n)
    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 (0 == n || 1 == n)
    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).

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.

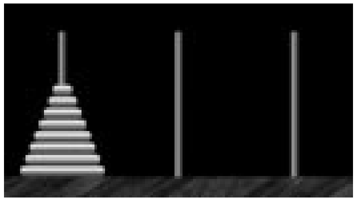
Should I use iteration or recursion?

- Consider ease of implementation.
- Consider time/space efficiency.

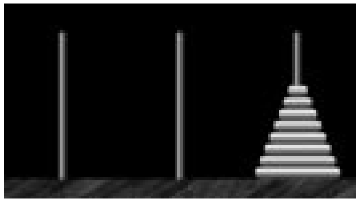
Towers of Hanoi

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.



Start

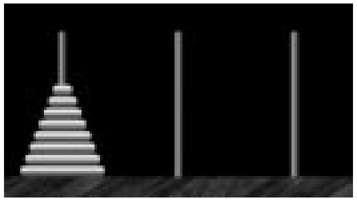


Goal

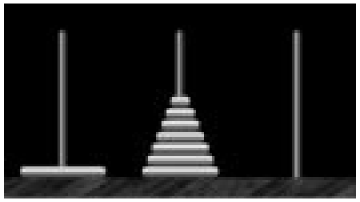
Towers of Hanoi demo

Edouard Lucas (1883)

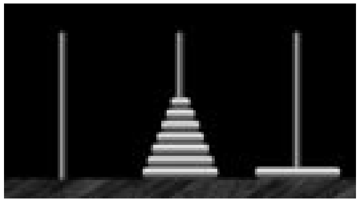
Towers of Hanoi: Recursive Solution



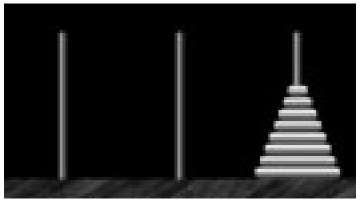
Move N-1 discs 1 peg to right.



Move largest disc 1 peg to left.



Move N-1 discs 1 peg to right.



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;
}
```

Solve 4 disc problem

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’;
}
```


Towers of Hanoi: Recursive Solution

hanoi.c

```
int main(void) {
  HanoiLeft(4);
  return 0;
}
```

Solve 4 disc problem

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 B.
Move disc 2 from C to B.
Move disc 1 from B to B.
Move disc 4 from A to C.
Move disc 1 from B to C.
Move disc 2 from B to C.
Move disc 1 from C to C.
Move disc 3 from B to C.
Move disc 1 from C to B.
Move disc 2 from C 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.**