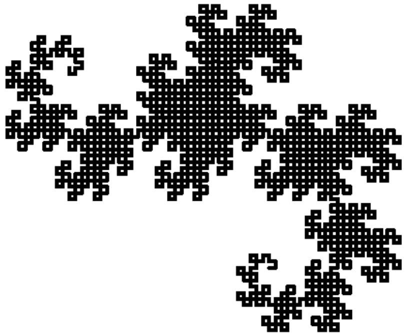
Lecture P6: Recursion



Overview

What is recursion?

- When one function calls ITSELF directly or indirectly.

Why learn recursion?

- 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
  - 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
- Any function call (call function g from f) requires system to:
  - save the local environment of f
  - set the value of parameters in g
  - jump to the first instruction of g, and execute that function
  - return from g, passing return value to f
  - restore the local environment of f
  - resume execution in f just after the function call (return address)

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 0 + 1 + 2 + . . . + 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--)
    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 0 + 1 + 2 + . . . + 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

A Simple Example

Goal: function to compute 0 + 1 + 2 + . . . + 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.

- The program will not "bottom-out" of recursion without a base case.
- Analog of infinite loops with for and while loops.

mystery(n)

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

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.

mystery(n)

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

base case

reduction step

anti-reduction step

Exponentiation

Goal: function to compute X^n, for positive integers x, n.

- Simple ITERATIVE solution.

iterative power function

```
int power(int x, int n) {
  int prod = 1;
  while (n--)
    prod *= x;
  return prod;
}
```

Check if (n == 0), then decrement n.

Number Conversion

To convert an integer N to binary:

- Stop if N = 0.
- Write “1” if N is odd; “0” if n is even.
- Move pencil one position to left.
- Convert N / 2 to binary. (integer division)

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

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

Easiest way to convert to binary by hand.

- Corresponds directly with a recursive program.

Recursive Number Conversion

Computer naturally prints from left to right.

- So we need to convert N / 2.
- Then write “0” or “1”.

convert

```
void convert(int N) {
  if (N == 0) return;
  convert(N / 2);
  printf(“%1d”, N % 2);
}
```

is 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
101011
```

Indentation level pairs statements belonging to same “invocation”

Recursive Number Conversion

Computer naturally prints from left to right.

- So we need to convert N / 2.
- Then write “0” or “1”.

convert

```
void convert(int N) {
  if (N == 0) return;
  convert(N / 2);
  printf(“%1d”, N % 2);
}
```

1 if N is odd; 0 if N is even

Proof of correctness:

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

Convert to any base $b \leq 10$.

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

Exponentiation

Goal: function to compute X^n, for positive integers x, n.

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

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{otherwise} \end{cases}$$

base case

reduction step

converges to base case

recursive power function

```
int power(int x, int n) {
  if (n == 0) return 1;
  return x * power(x, n-1);
}
```

base case

reduction step

Exponentiation

Goal: function to compute X^n, for positive integers x, n.

- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.
- Both require n multiplications, but can do with n/2 + 1 if n is even.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \cdot x^{n/2} & \text{if } n \text{ is even} \end{cases}$$

1 multiplication

$$17^{46} = 17^{23} \times 17^{23}$$

23 multiplications using previous algorithm

already computed

Exponentiation

Goal: function to compute X^n, for positive integers x, n.

- Simple ITERATIVE solution.
- Can also express using SELF-REFERENCE.
- Both require n multiplications, but can do with n/2 + 1 if n is even.
- Only **$2 \log_2 n$** multiplications needed with divide-and-conquer!

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \cdot x^{n/2} & \text{if } n \text{ is even} \\ x \cdot x^{(n-1)/2} \cdot x^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

n decreases by factor of two after at most 2 multiplications

improved recursive power function

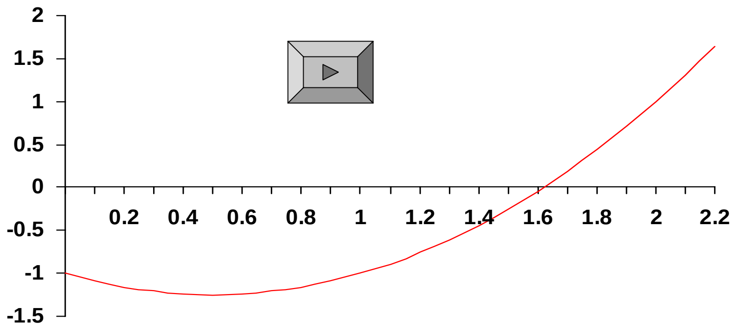
```
int power(int x, int n) {
  int t;
  if (n == 0) return 1;
  t = power(x, n/2);
  if (n % 2 == 0) return t * t;
  else return x * t * t;
}
```

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 you know l, r, such that 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 `f(m) == 0.0`, 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 || f(mid) == 0.0)
      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.
- Other methods.
  - Newton's method.
  - Steepest descent.

Traveling Salesperson Problem

Given N points, find a shortest tour connection them.

- Brute force approach is to try all N! possible permutations.
- If cities named a, b, c, then 6 possible permutations are:
  abc, acb, bac, bca, cab, cba.
- Not easy to do without recursion.

Key idea: permutations of abcde look like:

- End with a preceded by one of 4! permutations of bcde.
- End with b preceded by one of 4! permutations of acde.
- End with c preceded by one of 4! permutations of abde.
- End with d preceded by one of 4! permutations of abce.
- End with e preceded by one of 4! permutations of abcd.

Reduces enumerating permutations of N elements to enumerating permutations of N-1 elements.

Traveling Salesperson Problem

Recursive solution for trying all permutations:

- Use array a to store current permutation in a[1], …, a[N]
- N denotes number of cities whose position has not been determined.

Enumerating all Permutations

```
void visit(int N) {
  int i;
  if (N == 1) {
    checklength();
    return;
  }
  for (i = 1; i <= N; i++) {
    swap(i, N);
    visit(N-1);
    swap(N, i);
  }
}
```

base case

Swap cities i and N.

Decide position of remaining N-1 cities.

Restore order.

Traveling Salesperson Problem

Recursive solution for finding best TSP tour.

- Takes N! steps.
- No computer can run this for large value of N.
- For N = 100, 100! > 10^{150}.

Is there an efficient way to do this computation?

Possible Pitfalls With Recursion

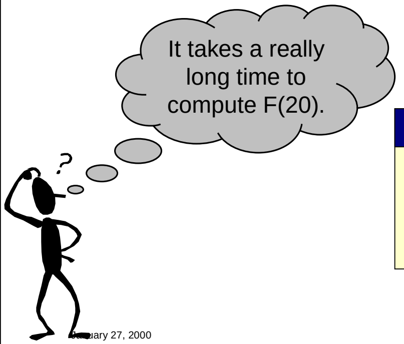
Is recursion fast?

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

Possible Pitfalls With Recursion

F(8) is recomputed 2 times.

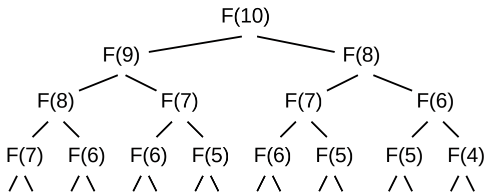
F(7) is recomputed 3 times.

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 (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 function using dynamic programming

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

`knownF` is an array that stores i^{th} Fibonacci number in i^{th} element. We assume `knownF` is initialized to 0.

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.

- LISP programming language has only recursion.

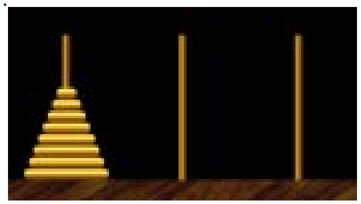
Should I use iteration or recursion?

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

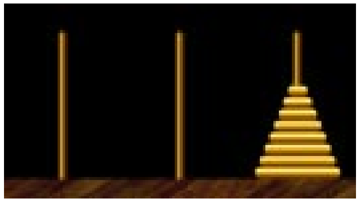
Towers of Hanoi

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 an empty peg or on top of a larger disc.
- Legend: world will end when monks accomplish this task with 40 golden discs on 3 diamond pegs.



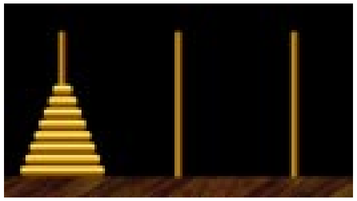
Start



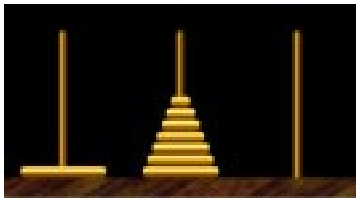
End

Towers of Hanoi demo

Towers of Hanoi: Recursive Solution

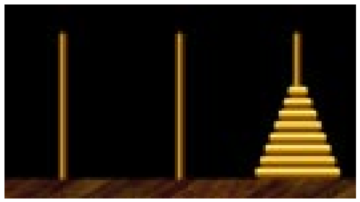


Move N-1 discs 1 peg to right.



Move largest disc 1 peg to left.

Move N-1 discs 1 peg to right.



Towers of Hanoi: Recursive Solution

HanoiLeft(N)

```
void HanoiLeft(int N) {
  if (N == 0) return;
  HanoiRight(N-1);
  ShiftLeft(N);
  HanoiRight(N-1);
}
```

Move top N-1 discs one peg to right.

Move disc N one peg to left.

mutually recursive functions

HanoiRight(N)

```
void HanoiRight(int N) {
  if (N == 0) return;
  HanoiLeft(N-1);
  ShiftRight(N);
  HanoiLeft(N-1);
}
```


Towers of Hanoi: Recursive Solution

ShiftLeft(N) and ShiftRight(N)

```
void ShiftLeft(int N) {
  printf("Shift disc %d one "
         "peg to left.\n", N);
}

void ShiftRight(int N) {
  printf("Shift disc %d one "
         "peg to right.\n", N);
}
```

Print directions to screen.

main

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

Solve 4 disc problem.

Towers of Hanoi: Recursive Solution

Unix

```
% gcc hanoi.c
% a.out

Move disc 1 one peg to right.
Move disc 2 one peg to left.
Move disc 1 one peg to right.
Move disc 3 one peg to right.
Move disc 1 one peg to right.
Move disc 2 one peg to left.
Move disc 1 one peg to right.
Move disc 4 one peg to left.
Move disc 1 one peg to right.
Move disc 2 one peg to left.
Move disc 1 one peg to right.
Move disc 3 one peg to right.
Move disc 1 one peg to right.
Move disc 2 one peg to left.
```

Towers of Hanoi

Is world going to end (according to legend)?

- Monks have to solve problem with N = 40 discs.
- Computer algorithm should help.
  - not really - takes 2^N - 1 steps
  - assuming rate of 1 disc per second, will take 348 centuries

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.