Recursion(int day){return Recursion(day += 1);}

Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Outline

Implementation

"To create recursion, you must create recursion."

How to design a recursive algorithm

1. First write the base cases. Must always have some base cases, which can be solved without recursion.
2. Think about solving the problem by combining the results of one or more smaller, but similar, sub-problems. If the algorithm you write is correct, then certainly you can rely on it (recursively) to solve the smaller subproblems.
3. Making progress. For the cases that are to be solved recursively, the recursive call must always be to a case that makes a **fixed quantity** of progress toward a base case (not fixed proportion).
4. Compound interest guideline: Never duplicate work by solving the same instance of a problem in separate recursive calls.
5. Check progressively larger inputs to inductively validate that there is no infinite recursion
6. The secret to success is: Do not worry about how the recursive call solves the subproblem. Simply accept that it will solve it correctly, and use this result to in turn correctly solve the original problem.

Outline

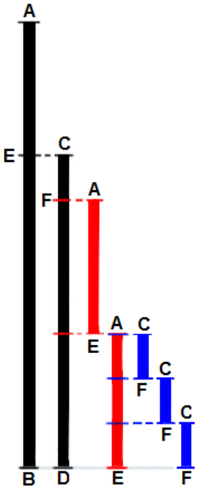
Example: Palindrome checking function

- Recursive case?
- Base case (which often results in termination)?
- Condition/test, which checks for the base?

Observe code to show recursion stack

Non-recursive GCD: Euclid's algorithm

Euclid's algorithm efficiently computes the greatest common divisor (GCD) of two numbers (AB and CD below), the largest number that divides both without leaving a remainder (CF).



Proceeding left to right:

Recursive GCD

Check out the code

Outline

Very simple word reversal

Use recursion to use the cin/cout buffer to reverse inputted text: see code

Outline

What is involved when making a function call?

Overhead:

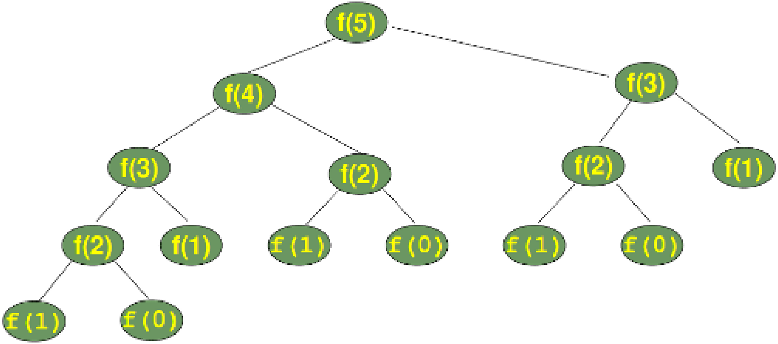
- save caller's state
- allocate stack space for arguments
- allocate stack space for local variables
- invoke routine at exit (return), release resources
- restore caller's "environment"
- resume execution of caller

Outline

Efficiency

Some recursive functions can be unnecessarily inefficient, and would be better as iterative functions, e.g., recursive Fibonacci:

fib(n) is 0 if n is zero
fib(n) is 1 if n is one
fib(n) is fib(n - 1) + fib(n - 2) otherwise



Observe code

Efficiency

Some recursive functions can be unnecessarily inefficient, and would be better as iterative functions, e.g., recursive Fibonacci:

fib(n) is 0 if n is zero
fib(n) is 1 if n is one
fib(n) is fib(n - 1) + fib(n - 2) otherwise

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1973
20	10946	10945	21891
25	121393	121392	242785
30	1346269	1346268	2692537

Observe code

Outline

Use a loop

If you can easily find a looping algorithm
(e.g., fibonacci with a loop below:)

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1973
20	19	57	21891
25	24	72	242785
30	29	87	2692537

Remember, recursion is just another way to loop

Outline

Tail recursion: just a glorified loop

- Tail call is a subroutine call performed as the final action of a procedure (recall order mattering example)
- Tail calls don't have to add new stack frame to the call stack.
- Tail recursion is a special case of recursion where the calling function does no more computation after making a recursive call.
- Tail-recursive functions are functions in which all recursive calls are tail calls and thus do not build up any deferred operations.
- Producing such code instead of a standard call sequence is called tail call elimination.
- Tail call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing more efficient structured programming.

Convert an iterative loop into a tail recursive

1. First identify those variables that exist outside the loop but are changing in the loop body; these variable will become formal parameters in the recursive function.
2. One then builds a function that has these "outside" variables as formal parameters, with default initial values
3. The original loop test becomes an if() test in the body of the new function
4. The if-true block becomes the recursive call.
5. Arguments to the recursive call encode the updates to the loop variables.
6. else block becomes the value the loop attempted to calculate
7. Conversion results in tail recursion

Examples in code

Language choice

gcc-c++ (aka g++) usually can do tail call optimization. Tail call elimination doesn't automatically happen in Java or Python, though it does reliably in functional languages like Lisp: Scheme, Clojure (upon specification), and Common Lisp, which primarily employ recursion.

Outline

Memoization and caching

Store the values which have already been computed, rather than re-compute them:

See code for Fibonacci

Outline

Convert recursion to loops

Conversion of recursive to loops is less systematic than converting a loop to tail recursive, requires more creativity, and isn't always easy for a human, though converting an already tail recursive algorithm is more straightforward:

```
void tail(int i){
  if(i > 0){
    cout << i << ' ' ;
        tail(i-1);
  }
{

void iterEquivalentOfTail(int i){
  for( ; i > 0; i--){
    cout << i << ' ';
  }
}
```

Outline

Implementing recursion with a stack

Often converting more inherently recursive algorithms to loops requires keeping a programmer-designed stack like would be done by the compiler in the first place. See example for factorial.