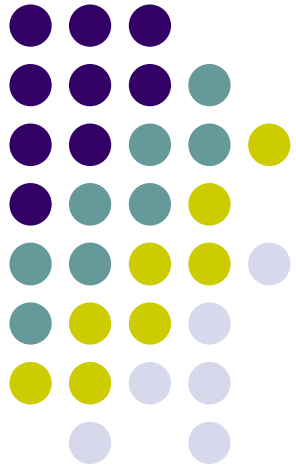
Mastering Data Structures and Algorithms: A Practical Approach

Recursion

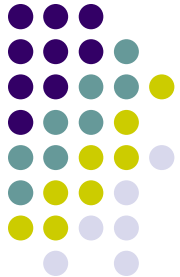


By Dr. Juan C. Gomez

Fall 2018

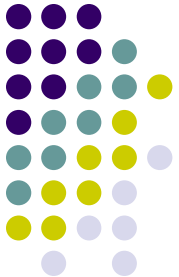
UC Santa Cruz Extension in Silicon Valley

Overview



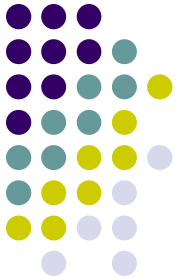
- Concept
- Approaching Recursive Problems
- Recursion vs Iterative Solutions
- Types of Recursion
- To Recurse or not to Recurse: there lies the question
- Problem with Recursion
- Structure of Recursion
- Hidden Cost of Recursion: Memory
- Four Basic Rules of Recursion
- Recursion and Induction
- Simple Examples

Overview



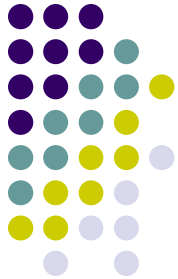
- Performance analysis of recursive functions.
- Examples

Concept



- Recursion:
  - A function defined in terms of itself is said to be recursive.
  - Valid for mathematical functions and programming functions.

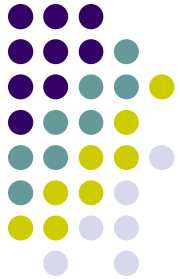
Approaching Recursive Problems



Bottom-Up Approach

The bottom-up approach is often the most intuitive. We start with knowing how to solve the problem for a simple case, like a list with only one element. Then we figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can *build* the solution for one case off of the previous case (or multiple previous cases).

Approaching Recursive Problems



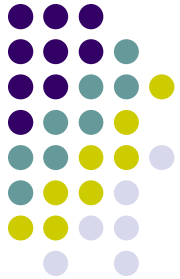
Top-Down Approach

The top-down approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem.

In these problems, we think about how we can divide the problem for case N into subproblems.

Be careful of overlap between the cases.

Approaching Recursive Problems



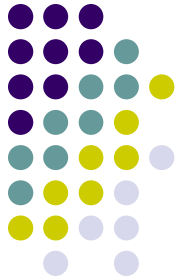
Half-and-Half Approach

In addition to top-down and bottom-up approaches, it's often effective to divide the data set in half.

For example, binary search works with a "half-and-half" approach. When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.

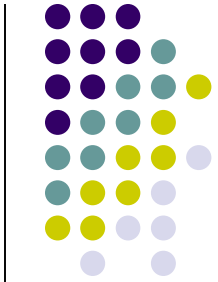
Merge sort is also a "half-and-half" approach. We sort each half of the array and then merge together the sorted halves.

Recursive vs Iterative Solutions



- Iterative solutions are usually faster and more efficient.
  - Recursion involves the overhead of function call implementation and returned values
  - Recursion entails implicit memory allocation in the stack.
- Recursion is more intuitive. But…deep recursive functions may run out of memory!
- Some data structures like trees are very simple to traverse using recursion.
- Recursive functions are usually shorter and simpler to understand.

Types of Recursion

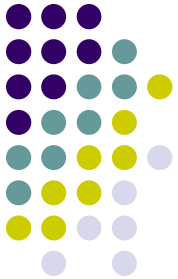


- Simple Recursion
  - Linked list traversal
- Multiple Recursion
  - Tree traversal
- Indirect Recursion
  - F() => G() => F()
- Tail Recursion
  - Can be easily transformed into iterative function

To Recurse or not To Recurse: there lies the question

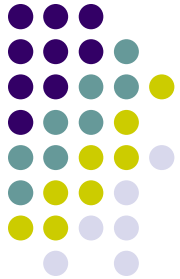
- Are you solving the same smaller problem within various recursive calls to solve a larger problem?
  - In other words: do subproblems overlap?
- Is N very large? Are you recursing over N entries or log(N) entries (i.e. balanced trees)?

Problems with Recursion



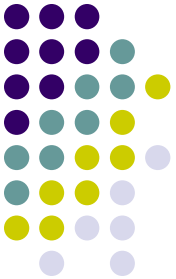
- Stack overflow on deep recursion.
- Inefficiencies steaming from nested function calls.
- Hidden or implicit memory allocation.
- Some times used without need: i.e. Tail recursion.
- Infinite loops when base case not implemented correctly.

Structure of Recursion



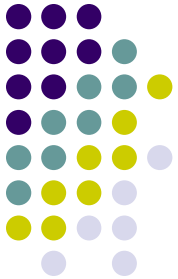
- Base case(s): when problem is of small enough size, a predefined solution is returned.
- Recursive calls with problems of increasingly small size.
- Progress or convergence to the base case(s).

Hidden Cost of Recursion



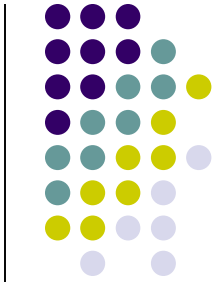
- Nested function calls, imply accumulated activation records or frames in the stack.
- What looks like O(1) memory is in fact O(n) or O(log(n)) in many cases.

Basic Rules of Recursion



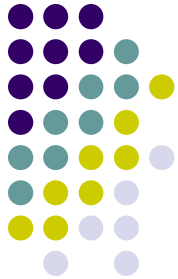
- Base Case(s):
  - There must be at least one case that can be solved without recursion.
- Progress:
  - Recursion is always to a case that makes progress to a base case.

Basic Rules of Recursion



- Design Rule:
  - All recursive calls work.
- Compound Interest Rule:
  - do not duplicate work.

Recursion and Induction



Proof (*by induction on the number of digits in n*).

First, if *n* has one digit, then the program is trivially correct, since it merely makes a call to `printDigit`. Assume then that `printOut` works for all numbers of *k* or fewer digits. A number of *k* + 1 digits is expressed by its first *k* digits followed by its least significant digit. But the number formed by the first *k* digits is exactly $\lfloor n/10 \rfloor$, which, by the inductive hypothesis, is correctly printed, and the last digit is *n* mod 10, so the program prints out any (*k* + 1)-digit number correctly. Thus, by induction, all numbers are correctly printed.

Simple Examples

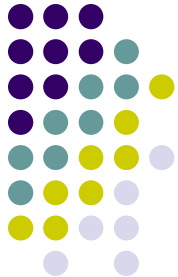


Figure 1.4 Recursive routine to print an integer

```
1     public static void printOut( int n )  /* Print nonnegative n*/
2     {
3         if( n >= 10 )
4             printOut( n / 10 );
5         printDigit( n % 10 );
6     }
```

Simple Examples

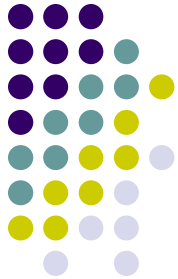
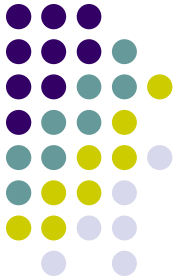


Figure 1.3 A nonterminating recursive method

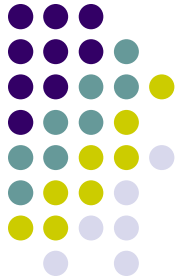
```
1     public static int bad( int n )
2     {
3         if( n == 0 )
4             return 0;
5         else
6             return bad(  n / 3 + 1 ) + n - 1;
7     }
```

Simple Examples



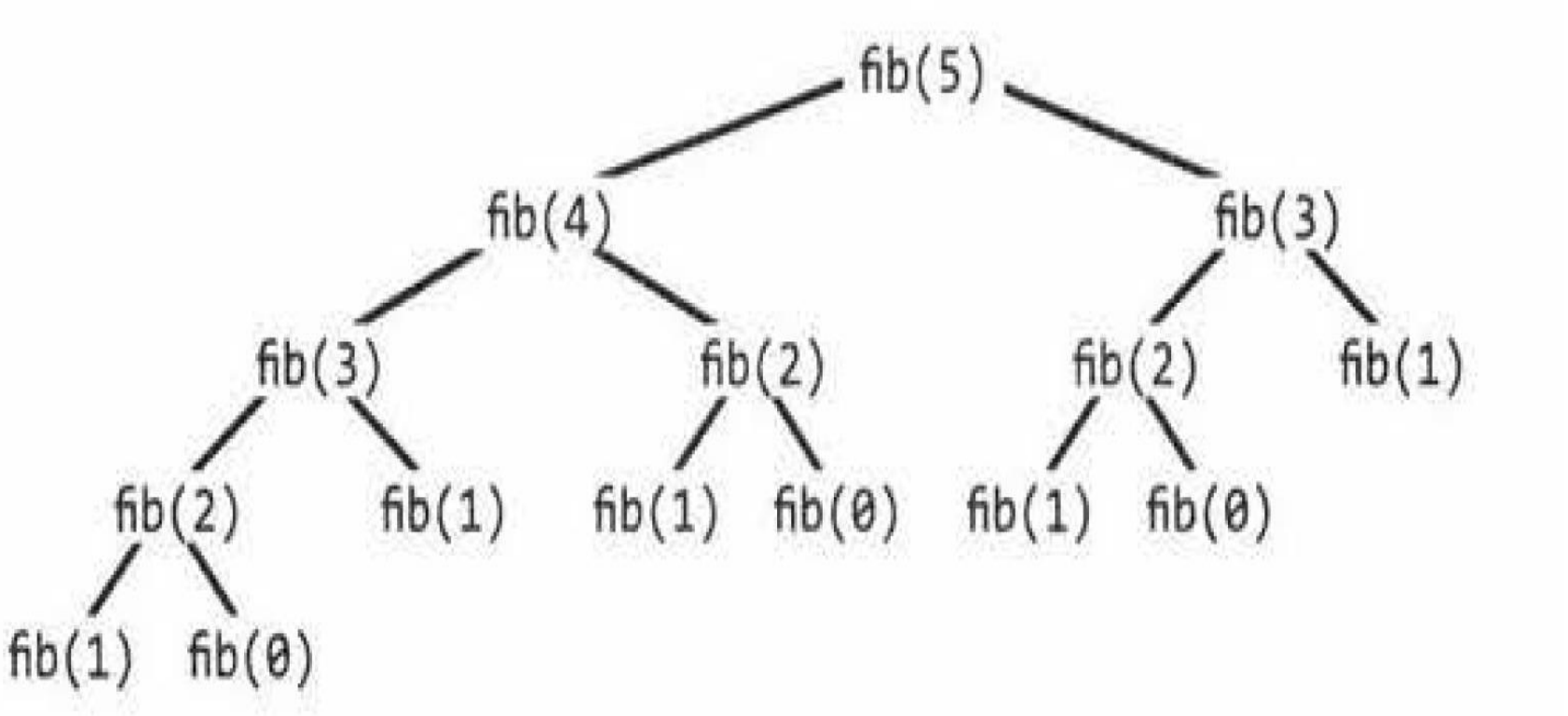
```
public static int rank(int key, int[] a)
{  return rank(key, a, 0, a.length - 1);  }

public static int rank(int key, int[] a, int lo, int hi)
{  // Index of key in a[], if present, is not smaller than lo
   //                                 and not larger than hi.
   if (lo > hi) return -1;
   int mid = lo + (hi - lo) / 2;
   if      (key < a[mid]) return rank(key, a, lo, mid - 1);
   else if (key > a[mid]) return rank(key, a, mid + 1, hi);
   else                   return mid;
}
```

Simple Examples

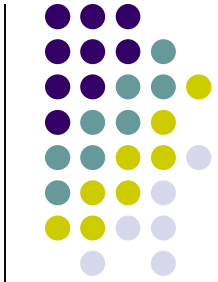


```
1  int fibonacci(int i) {
2     if (i == 0) return 0;
3     if (i == 1) return 1;
4     return fibonacci(i - 1) + fibonacci(i - 2);
5  }
```

Simple Examples

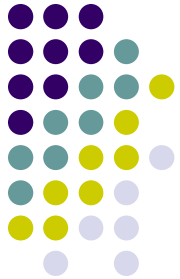


Performance Analysis of Recursive Functions



- Examples:
  - T(n)=T(n-1) + c
  - T(n)=T(n/2) + n

Examples

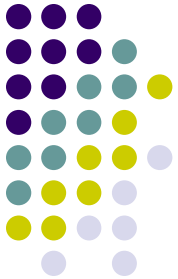


- Sit the Antisocial Recursive
- Given a string and a dictionary of valid words, find if the string is a sequence of valid words in the dictionary.
- Given an integer number as a string, and the operators +,-,/,* find if you may create an operation interspersing the operators in the number that yields a target number.

Examples

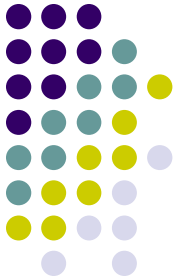
8.1 **Triple Step:** A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

Examples



8.2 Robot in a Grid: Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

Examples

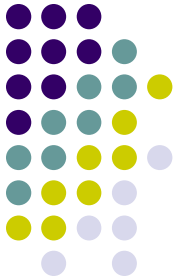


8.3 **Magic Index:** A magic index in an array `A[0...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.

FOLLOW UP

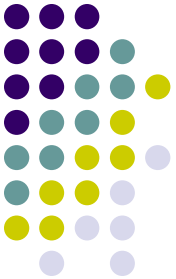
What if the values are not distinct?

Examples



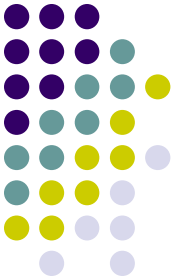
8.4 Power Set: Write a method to return all subsets of a set.

Examples



8.5 **Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator. You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

Examples



8.6 **Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

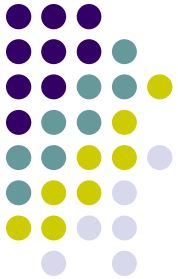
(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto another tower.

(3) A disk cannot be placed on top of a smaller disk.

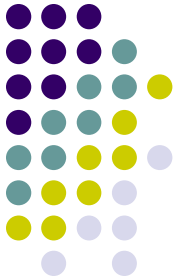
Write a program to move the disks from the first tower to the last using stacks.

Examples



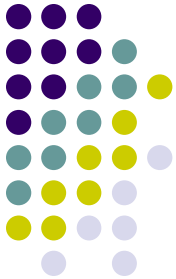
8.7 Permutations without Dups: Write a method to compute all permutations of a string of unique characters.

Examples



8.8 Permutations with Dups: Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

Examples



Parens: Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

Output: ((())), (()()), (())(), ()(()), ()()()

Examples

Paint Fill: Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

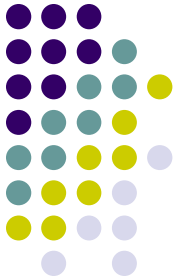
Examples

8.11 **Coins:** Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

Examples

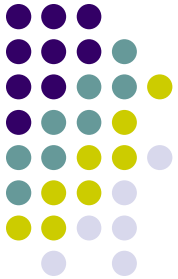
8.12 Eight Queens: Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

Examples



8.13 Stack of Boxes: You have a stack of n boxes, with widths w_i, heights h_i, and depths d_i. The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

Examples



8.14 Boolean Evaluation: Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`.

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```