Basic Recurrences Lab

Lab4

Generated by Doxygen 1.8.7

Fri Sep 25 2015 19:11:07

Contents

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

2 Class Documentation

2.1 Recurrences Class Reference

Static Public Member Functions

- static boolean isPrime (int n)
- static int fibonacci (int n)
- static int power (int base, int exponent)
- static int pmod (int a, int modulus)
- static int gcd (int m, int n)
- static double sqrt (double x, double tolerance)
- static boolean summable (int target, int[] arrayOfIntegers)

2.1.1 Detailed Description

Instructions

Essentially, you will write a handful of static methods that implement some well-known algorithms over integers (and one over doubles) that have well-defined recursive representations.

Most of the methods that you will be required to write could be just as easily written using Java's iterative constructs. Again, that is not the purpose of this lab! At least one method, however, `summable` would be very difficult to write iteratively. As you work through these, ask yourself if you understand the difference between a solution that is recursively implemented, but is, essentially, *iterative* versus solutions that are fundamentally *recursive*, meaning that they rely upon recursively computed partial results to assemble their final answers—again, think about `summable`. You should also

ask yourselves how much easier many of these functions appear when written recursively—here, I'm thinking especially of the `gcd` and `fibonacci`.

- Naturally, your solutions will be examined by us, and non-recursive solutions will receive 0 points.
- You should **not** need to use any methods/classes that we have not either discussed in this class or in the previous class. In particular, you implementation of the `sqrt()` *may only use* the `Math.abs()` method from the `Math` package.

Author

UMD CS Department.

2.1.2 Member Function Documentation

2.1.2.1 static int fibonacci (int *n*) [static]

Recursively implements the classic Fibonacci function. Note, this implementation requires the fibonacci (0) = 0. And the first couple of Fibonacci numbers is given here:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
See if you can find the pattern. Hint: the next fibonacci is determined by summing previous fibonacci numbers (how many)? If you don't know what a fibonacci number is, try a Google search, or Wolfram Mathworld.

Parameters

n	

Returns

2.1.2.2 static int gcd (int *m,* int *n*) [static]

Uses the Euclidean algorithm to recursively compute the greatest common divisor of two positive integers, m and n. Note, in order to receive credit for this function, your implementation *must* use the `pmod` function implemented directly above. (Rather than give you this, you'll need to look it up!)

Parameters

m	
n	

Returns

2.1.2.3 static boolean isPrime (int *n*) [static]

Returns `true` iff `n` is a prime integer, which in this case means a positive integer $>$ 1 that is divisible only by itself and 1.

Parameters

n	

Returns

2.1.2.4 static int pmod (int *a*, int *modulus*) [static]

pmod stands for pseudo-mod. The real modulus function needs to deal with negative integers, but this one assumes positive integers only. The modulus operator, from our perspective, looks just like the % (remainder) operator in Java—except that you're implementing it recursively, and you will use it to implement the gcd function, which is defined immediately after this one.
For example: $\texttt{pmod}(4,2) = 0$, but $\texttt{pmod}(4,3) = 1$, etc.
You should look up "modulus" and/or review the definition of the "%" operator in Java for clarification.

Parameters

a	
modulus	

Returns

2.1.2.5 static int power (int *base*, int *exponent*) [static]

Implements the basic laws of positive exponents over the integers. Note: for this implementation, anything to the 0-power is 1.
For example: $0^0 = 1$. But $0^1 = 0$.

Parameters

base	
exponent	

Returns

2.1.2.6 static double sqrt (double *x*, double *tolerance*) [static]

Recursively compute (approximate) the square root of a positive double number by successive averaging. Hint: to do this, define several private helper methods:

- `closeEnough(double x, double guess, double tolerance)`: returns true when the absolute difference between the square of the guess and the original number is less than the tolerance. In other words, close enough is true when:
$$|guess^2 - x| < tolerance$$
- `refine(double x, double guess)` returns the average of the `guess + (x/guess)`. In other words, refine computes
$$\frac{guess + \frac{x}{guess}}{2}$$

- `sqrt_aux( double x, double guess, double tolerance )`: that recursively `refine`s the initial guess (which is 1.0), testing against the original number, `x`, until `closeEnough` is true. At this point, the refined value is returned.

Parameters

x	
tolerance	

Returns

2.1.2.7 static boolean summable (int *target,* int[ ] *arrayOfIntegers*) [static]

Uses "backtracking" (through recursion) to determine if the "target" (a positive integer) appears in or can be computed as the result of summing any combination of integers in the `arrayOfIntegers`.
Note: assume nothing about each integer in the array of integers except that each may be used only once.

For example: summable(10, {1,2,3,4,5}) is true, but summable(20, {1,2,3,4,5}) is not.

By way of a hint: you will most certainly want to define a recursive "auxiliary" method (private static) that manages the array indexing for you.

Parameters

target	
arrayOfIntegers	

Returns

The documentation for this class was generated from the following file:

- src/student_classes/Recurrences.java

Index