Lab02-MakingDynamicChange

Darwin Jacob Groskleg

January 30, 2019

Contents

1 src/main.cpp

```cpp
/* main.cpp
 * --------
 * CSCI 355  Algorithm Analysis
 * Lab 2     Making Dynamic Change
 *
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, January 29, 2019
 *
 * Purpose: Test the 3 different implementations for counting change.
 */
#include "change.hpp"

#include <iostream>
#include <vector>

using namespace std;

void test_part1();
void test_part2();
void test_part3();
void print(vector<int> Q);


int main() {
    test_part1();
    test_part2();
    test_part3();

    return 0;
}


// SLOW when big!
void test_part1() {
    cout << "\nTesting Part 1" << endl;

    cout << "For D={ 1, 5, 10, 25, 100, 200 }, Change for 1, 3 & 368 cents:\n";
    print( Part1::quantities_for(  1, { 1, 5, 10, 25, 100, 200 }) );
    print( Part1::quantities_for(  3, { 1, 5, 10, 25, 100, 200 }) );
    print( Part1::quantities_for(368, { 1, 5, 10, 25, 100, 200 }) );

    cout << "\nFor D={ 1, 5, 8, 10, 25 }\n";
    cout << "Coins for 16 cents: "
        << Part1::coins_for( 16, { 1, 5, 8, 10, 25 });
    print( Part1::quantities_for( 16, { 1, 5, 8, 10, 25 }) );

    cout << "Coins for 169 cents: "
        << Part1::coins_for( 169, { 1, 5, 8, 10, 25 });
    print( Part1::quantities_for( 169, { 1, 5, 8, 10, 25 }) );

    cout << '\n' << '\n';
}

// SLOW but better than part I
void test_part2() {
    cout << "Testing Part 2" << endl;

    cout << "For D={ 1, 5, 8, 10, 25 }\n";
    cout << "Coins for 16 cents: "
        << Part2::least_coins_for(16, { 1, 5, 8, 10, 25 })
        << " should be 2.\n";

    cout << "Coins for 169 cents: "
        << Part2::least_coins_for(169, { 1, 5, 8, 10, 25 })
        << " should be ?.\n";

    cout << '\n';
}


void test_part3() {
    cout << "Testing Part 3" << endl;

    cout << "For D={ 1, 5, 8, 10, 25 }\n";
    cout << "Coins for 16 cents: "
        << Part3::least_coins_for(16, { 1, 5, 8, 10, 25 })
        << " should be 2.\n";

    cout << "Coins for 169 cents: "
        << Part3::least_coins_for(169, { 1, 5, 8, 10, 25 })
        << " should be ?.\n";

    cout << '\n';
}


void print(vector<int> Q) {
    cout << "    Q={ ";
    for (auto& q : Q) cout << q << ' ';
    cout << '}'<< endl;
}
```

2 include/change.hpp

```cpp
/* change.hpp
 * ----------
 * CSCI 355  Algorithm Analysis
 * Lab 2     Making Dynamic Change
 *
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, January 29, 2019
 *
 * Purpose: interface for algorithms that convert units of money in cents
 *          to the appropriate amount of change in denominated coins.
 *
 * Definitions:     cents   = units of money
 *                  coins   = denominated money
 *                  pennies = a 1 cent denominated coin
 *                  change  = equivalent units of money in a set of coins
 */
#ifndef CHANGE_HPP_INCLUDED
#define CHANGE_HPP_INCLUDED
#include <vector>

/// Part I  The Greedy Approach
namespace Part1 {

/// quantities_for
///
/// Returns the quantities of each denomination specified that add up to the
/// given target amount in cents.
///
std::vector<int>
quantities_for(int cents, const std::vector<int> denominations);

/// coins_for
///
/// Returns the number of coins that could be given out in change for
/// a target amount of cents using specified denominations.
int coins_for(const int cents, std::vector<int> denominations);

}; // Part1
```

```cpp



/// Part II  The Recursive Approach
namespace Part2 {

/// least_coins_for
///
/// Returns the smallest number of coins that could be given out in change for
/// a target amount cents using specified denominations.
///
/// Same as `coins_for` Part I but is concerned with computing the smallest
/// number that is possible.
///
int least_coins_for(const int cents, std::vector<int> denominations);

}; // Part2



/// Part III  Dynamic Programming Approach
namespace Part3 {

/// least_coins_for
///
/// Returns the smallest number of coins that could be given out in change for
/// a target amount cents using specified denominations.
///
/// Same as `least_coins_for` in Part II.
///
int least_coins_for(const int cents, const std::vector<int> denominations);

}; // Part3


#endif // CHANGE_HPP_INCLUDED
```

3 src/change.cpp

```cpp
/* change.cpp
 * ----------
 * CSCI 355  Algorithm Analysis
 * Lab 2     Making Dynamic Change
 *
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, January 29, 2019
 *
 * Purpose: implementation of algorithms that convert units of money in cents
 *          to the appropriate amount of change in denominated coins.
 */
#include "change.hpp"

#include <algorithm>
#include <numeric>

/// Part I  quantities_for
///
///     The Greedy Approach
///
/// Analysis
///     Basic Operation: comparing the value of cents with each denomination.
///     Input Size:      n = number of denominations
///
///     T(n) = n        basic operation occurs for every denomination
///
std::vector<int>
Part1::quantities_for(int cents, const std::vector<int> denominations) {
    std::vector<int> change(denominations.size(), 0);

    for (int d=(int) denominations.size()-1; d>=0; --d) {
        if (denominations[d] <= cents) {
            change[d] = cents / denominations[d];
            cents %= denominations[d];
        }
    }

    return change;
}

/// Part II  least_coins_for
///
///     The Recursive Approach
///
/// Given Model Instructions
///    D = coin array
///    n = size of array
///    c = amount of change
///   int need(int D[], int n, int c)
///
/// NOTE ON TRACING EXERCISE
///     Tracing this took me FOREVER! But I still did it. I had no choice to
///     understand the series of decisions being made. Documented it in plain
///     english with the source below.
///
/// INTUITIVE IMPRESSION (pre-analysis)
///     It seems like the work of making copies of sub-arrays to pass to
///     recursive calls is roughly the same order of time compared as solving
///     each of the subproblems. Thus dividing and combining is just as
///     prominent of a part as 'conquering', putting this in 2 of the master
///     method.
///
///     Guess:  T(n) = O(nůlg n)
///
/// Analysis
///     Basic Operation: ??? (most common us copying arrays)
///     Input Size:      n = e of || m X d ||
///                          s.t. m=cents, d=number of denominations
///
///     Approach by first building a recurrence relation of the form
///         T(n) = aůT(n/b) + f(n).
///     Then use the master method.
///
///     a:    Worst case has 2 subproblems
///     b:    Worst case subproblem size is m+(d-1)
///     f(n): The cost of further dividing up the problem is the cost of copying
///           the sub-array of denominations. Thus:
///             f(n) = n-1 = O(n)
```

```
///
///     T(m+d) = 2ůT((m+d)/(m+d-1)) + O(n)
///
///     Master Method
///     1. Is f(n) = O( (m+d)^log( (m+d-1), 2-eps) ) where eps > 0?
///         So eps = 1?
///     ...this is where I was getting stuck with the math.
///
int Part2::least_coins_for(const int cents, std::vector<int> denominations) {
    using namespace Part2;

    // Base Cases
    if (cents == 0)                 return 0;
    if (denominations.size() == 1)  return cents;

    // Setup for Recursive Step
    //
    //  Make a copy of our denominations array without the last element.
    //  That is SD[m] from D[n] s.t. m = n-1
    //      T(n) = n - 1
    std::vector<int> sub_denom{denominations.begin(), denominations.end() - 1};

    // Recursive Step
    //
    // Model Instructions:
    //      if ( c < D[n-1] ) then
    //           need(D, n, c) = need(D, n-1, c)
    //      else
    //           need(D, n, c) = min ( need(D, n-1, c) ,
    //                             1 + need(D, n, c  D[n-1] )
    //
    // Does the number of cents fit in the current biggest denomination?
    if (cents < denominations.back())
        return least_coins_for(cents, sub_denom);

    // Otherwise, what is least coins in change:
    //  1. Change without the current largest denomination?
    //  2. Change using one coin of the current largest denomination?
    return std::min( least_coins_for(cents, sub_denom),
            1 + least_coins_for(cents - denominations.back(), denominations));
```

```
}





/// Part III  least_coins_for
///
///     Dynamic Programming Approach
///
/// Analysis
///     Basic Operation: comparing size of i (cents) to d (denominations)
///                      In step 3b: if (i >= d)
///     Input Size:      n = (number of cents, d size of denominations list)
///
///
///     T(n) = O(cents X d)
///
///
///     Since for each increasing number i towards cents we make a comparison
///     with each coin in the denomination list. We ignore accesses to the array
///     since they're constant time and happen at most as often than the
///     comparisons.
///
/// Algorithm Summary
///     This algorithm does the same set of decision making to find viable sets
///     of change as Part II by checking if it's better to use a smaller
///     denomination than required in order to use fewer coins. It performs
///     better than Part II by working up from a smaller target change amount
///     and saving computed change along the way for later reuse.
///
/// Final Note
///     This reminds me of using a sieve to find primes (Eratosthenes),
///     which I'm realising must also be dynamic programming.
///
int
Part3::least_coins_for(const int cents, const std::vector<int> denominations) {
    std::vector<int> T(cents+1);    // Step 1
    T.at(0) = 0;                    // Step 2

    // Step 3
    for (int i=1; i<=cents; i++) {
        // a. Initialize for a penny
        T.at(i) = i;

        // b. For each denominated coin
        for (const auto& coin : denominations) {
            if (i >= coin)
                T.at(i) = std::min(     T.at(i),
                                    1 + T.at(i - coin) );
        }
    }

    // Step 4: T[c] store the minimum coins for each value in cents
    return T[cents];
}



/// Part I  coins_for
///
/// Same order of time complexity as `Part1::quantities_for`.
///
int Part1::coins_for(const int cents, std::vector<int> denominations) {
    auto q = quantities_for(cents, denominations);
    return std::accumulate(q.begin(), q.end(), 0);
}
```