# CSC172 LAB 22

# CHANGE IS GOOD

## 1 Introduction

The labs in CSC172 will follow a pair programming paradigm. This requires every student to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is not optional in CSC172. It is a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

## 2 Making the Change

It's been a long hard semester, but a good one. Your participation in these labs have, no doubt, helped you learn a number of computer programming techniques. Moreover, your discussions with your lab partner and your direct, personal, face-to-face engagement with your peers and your TAs during these labs have honed your professional interaction skills and given you opportunity to practice teamwork, and strengthened your ability to articulate and discuss programming concepts. These secondary skills are valued by both employers and graduate schools. While the pair programming process can be challenging, your investment in it will reap benefits for you in diverse areas of endeavor for your whole life. Now, the end of the semester is upon us, and things must change. Change is important. We look all around us, we see change. The seasons change. As the semester draws to a close, we contemplate the change in our situations. Politicians on TV promise us change. Bums on the street ask us for change. The more things change, the more they stay the same. So, it is natural at this time of year, to seriously meditate on change in the universe, in nature, in society, and in ourselves. *What better way to do this than to write a computer program that calculates the minimum token count*

*required to make monetary change?*

So, our goal it to write a program that takes in an amount and a set of denominations in a money system and calculates the minimum set of monetary tokens required to issue that amount of change. In doing so, we will be able to explore the use of dynamic programming. The process of making change is simple, but the computation can be complex.

1. Let us begin this lab by defining a simple class with a main method. In order to work in integer values we will think of a ten dollar bill as a thousand pennies (1000), five dollars as 500, etc. Your main method should take a command line argument for the amount of change to make, but should include a default value if no amount is given. Consider the sample code, below.

```
int [] money = { 100000, 5000, 2000, 1000, 500, 100 , 25, 10 , 5, 1 };
int amount = 63;
if (args.length == 1) amount = Integer.parseInt(args[0]);
```

We want to define a method that returns an array of values. For instance, if we passed in 63 cents to our program, then we would want the return array to contain {25, 25, 10, 1, 1, 1}. Note that if we included a 21 cent piece then the minimum solution would be {21, 21, 21}. Our main method should be able to interact with this method as follows.

```
int [] change = makeChange(amount, coins);
System.out.print("Change for " + amount + " is  {");
for (int i = 0 ; i< change.length;i++)
   System.out.print(change[i] + ", ");
System.out.println("}");
```

In the end, your working program should operate roughly along the following lines.

```
$ java Change 5295
Change for 5295 is  {2000, 2000, 1000, 100, 100, 25, 25, 25, 10, 10, }
```

Of course, if we were to include a 21 cent piece we should get :

```
$ java Change
Change for 63 is  {21, 21, 21, }
```

Rather than the typical {25, 25, 10, 1, 1, 1}
Of course, writing the makeChange method is the hard part.

2. Writing a program to accomplish this presents a number of interesting design and implementation issues. At this point in the lab, you should take a few minutes and discuss and sketch solutions with your lab partner. What would you do if you had to solve the problem from this point forward without any guidance? How will the algorithm progress? How will you keep

track of the data as the algorithm progresses? How will you get the output in the form required? Together, write a short paragraph in English that sketches a possible solution before you start writing any code. *Include this paragraph in your lab write up*. The solution that you sketch may differ from what is presented in the remainder of this lab, but that is fine. There are a lot of equally valid strategies that will solve the problem.

3. Let's look at the basic backtracking algorithm. The basic idea is recursive. We are given an amount of money to make change for and a set of denominations. We should select every denomination in turn, subtract it's value from the amount and make a recursive call on the difference. If the recursive call returns the optimal change amount for the difference, then the difference plus the value of the current selection is the change we want. All we need to do is keep track of the amount of tokens, but this can be a little tricky. As is often the case with complex methods, we can write a "wrapper" method that sets up the call to the recursive method.

   So, write the first `int [] makeChange(int amount, int [] denomiations)` method. This method serves to set up and call the recursive backtracking method. In the method make a new array of integers called "counts" this array should be one longer than the denominations array. We will use it to count the number of each denomination the extra space can be used to keep track of the number of tokens. We will pass this array along the recursion in order to keep track of the optimal combinations. So our `makeChange` method should first declare and allocate the count array, then make a call on the (yet to be written) recursive method. After the recursion completes, the counts for the optimal set of change should be in the counts array. So, our method then needs to "unpack" the counts into the format we want for output. To do this, declare a second array of ints – the array we will return. The length of this array should be equal to the number of tokens (which is stored in the last location in the counts array). Loop through the counts array and place the appropriate values into the return array. For example, if the denominations array contained {25, 10, 5, 1}, then for the amount of 63 the counts array should contain {2, 1, 0, 3, 6}. We can allocate a return array of length 6. Using a double loop we can fill the return array with {25, 25, 10, 1, 1, 1} and return this array to the main method. Write this "helper" method.

4. Now, all that remains is the recursive backtracking method that actually explores the solution space. This method is called by the `makeChange` method written in the previous section. One way to design this is to have it return true if a solution is found and false if no solution is found. The actual solution is returned in the counts array.

   So, let's write `boolean recMakeChange(int amount, int [] denom, int [] counts)`. As The amount and denominations are as before. We will use the counts array to keep track of the solution candidates. If the amount is zero, then a solution has been found and the method returns true. If the amount is less than we have a bad attempt and the method should return false. Declare a boolean for the return value and initialize it to false.

   When the amount is a positive number, the method has to find the optimal solution. First, declare and allocate two integer arrays "temp" to hold sub-solutions and "best" to hold the best solution. These arrays should be the same length as the counts array that is passed in. Set the last value in the "best" array to the amount plus one. This ensures that any solution will be better than no solution at all.

   Loop through the denominations array. At the start of every loop set all the entries in the temp array to zero. Make a recursive call on the amount minus the denomination in question. If this returns true, then set the return boolean to true, add the denomination in question to the temp

array, and increase the size of the solution by one. Check the temp array against the current "best" solution. If the temp solution is better then, replace it with the temp solution. To do all this, consider the following code.

```
if (recMakeChange(amount - denom[i],denom,temp)) {
    if (temp[temp.length-1] < best[best.length-1]) {
         temp[i]++;
         temp[temp.length-1]++;
         for(int z = 0; z<best.length;z++)
          best[z] = temp[z];
    }
    rvalue = true;
}
```

When the loop completes (having tired all the denominations) check the return value. If it is true then you found at least one solution and it is stored in "best". Copy best into "counts" and return true. If you never found a solution, then return false.

5. At this point, you are all done - sort of. Test your program on a few values. You should be able to make change with this recursive program. However, you will find the solution abysmally slow. It's possible that you can make change for values under 20 cents in reasonable time, but your program may be so slow as to appear "stuck" when trying to make change for amounts above 50 cents. (This is where dynamic programming comes to the rescue.)

   The problem is that the program as written makes an enormous amount of redundant recursive calls. In order to investigate this declare a static class (ie "gobal") variable integer "calls" and initialize it to zero in your main method. Increment this variable at the start of recMakeChange. Print it out after the solution is printed in the main method. How fast are the number of recursive calls growing as a function of the amount you pass in

6. In order to speed the program up we need to keep track of partial solutions. We do this in a solutions table. Since our solution is an array, we need an array of arrays – a 2D array. Declare a static class (global) array. In the makeChange method allocate the first dimension of the array so that it can be indexed by an amount (table = new int[amount+1][]). Now add some code to the recMakeChange method to test and see if the amount has been previously calculated. If it has, then simply copy the solution into the counts array and return true. Consider the following code.

```
if (table[amount] != null) {
    for (int k = 0 ; k < table[amount].length;k++)
      counts[k]=table[amount][k];
    return true;
}
```

   Of course, in the case that we haven't previously calculated the result for this amount the

method proceeds as before. All we need to do is remember to save the solutions in the table when we do find one (at the very end of the recMakeChange method.

```
if (rvalue){
     table[amount] = new int[countarray.length];
     for(int z = 0; z<best.length;z++)
           table[amount][z] = counts[z] = best[z];
 }
```

Rerun your test with the above changes (the dynamic programming) and write a few sentances regarding the runtime comparison.

# 3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped” file that contains the following.) You will need to include the images of a screenshot of your maze.

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.). And a one sentence explaining the contents of all the other files you hand in.
2. Several JAVA source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file. It is expected that you will have a file for the test program class.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

# 4 Grading

Each section (1-6) accounts for 15% of the lab grade (total 90%)

(Correctness counts for 10%, proper style counts for 5%)

Attendance accounts for 5%

Working in pairs accounts for 5%