Project #2
CS3410- Dr Gallagher
10/11/09
Nathan Flack, Caleb Mays, Alex Laird

# Making Change

## Algorithm Analysis

By comparison, the dynamic, bottom-up implementation performed the fastest on all accounts, as shown in Figure 1. Even at zero and one, the overhead for the method call was the smallest since there was only one call. We convinced the iterative, bottom-up implementation to run quickly until throwing an OutOfMemoryError at a value of 1,470,723. Even at this value, the method ran in 0.1 seconds.

Recursive with memoization was second best, as shown in Figure 1. This became especially evident once the values reached the thousands. Recursive memoization ran in a reasonable amount of time for any value until the stack grew too large and a StackOverflowError was thrown. On the box the tests were run on, recursive using memoization could make it to a value of 21,305 before overflowing the stack after 0.007 seconds.

Recursive without memoization came in dead last, as shown in Figures 1, unable to even make it to a value of fifty in a reasonable amount of time. For values higher than twenty-six, this implementation takes a matter of minutes, as shown in Figure 2. It is horribly inefficient, as expected since, by nature, it has to operate in $O(2^n)$ time.

For the iterative solution, zero and one take 3,000 ns. For both of the recursive solutions, zero and one take 4,000 ns. The additional 1,000 ns is likely caused by the overhead produced by the recursive calls.

Interestingly, all three methods seemed to have a jump in performance time around values of 200, and recursive using memoization hit another jump around 400. Our assumption was that some values simply had to do more lookups than other values, even than higher values.

All three versions of the project compiled and ran with the desired output, giving the best solution each time.