EECS2011: Fundamentals of Data Structures

Assignment 2

Name: Yang Wang

EECS ID: infi999

Student ID: 213894167

October 17, 2016

Problem 1 (30 points): Enumeration of Coin Change Making:

For this question, I used recursive method to solve the coin change problem.
In order to get the same combination sequence like instruction, I used money/25, money/10, money/5, money/1 to get the largest number of the 4 kinds of coins.
Then I use recursive method change() to find all combinations.

EX: I set money equal to 17 cents.
So the recursive start at 0 quarter, 1dimes, 3 nickels and 17 pennies. Then I fix the number of quarter dime and nickel, start to decrement penny from 17 to 0 to see if any of these combinations fit the input cents. After that I reset penny to 17 and reduce nickel by 1 then start the first step (decrement penny from 17 to 0). I used this way to go through all of combination and print out the result if this combination is fit the input money.
PS: <u>For my algorithm, it will return the correct result after going through all over coins' combination. But when you try to test some big numbers, it will lack of memory, because the recursive is too deep.</u> So I still have another algorithm, start recursive from quarter instead of penny. But that has a problem, the output is on the contrary. But I can still paste my algorithm below as support document.

```
public int ways(int money) {

    // For Quarter
    searchQuarter(money);
    return count;
}

public void searchQuarter(int money) {
    for (int i = 0; i < money/25 + 1; i++) {
        searchDim(money - i * 25, i + " quarter(s) ");
    }
}

public void searchDim (int money, String result) {
    for (int i = 0; i < money/10 + 1; i++) {
        searchNickel(money - i * 10, result + i + " dim(s) ");
    }
}

public void searchNickel (int money, String result) {
    for (int i = 0; i < money/5 + 1; i++) {
        searchPenny(money - i * 5, result + i + " nickel(s) ");
    }
}

public void searchPenny(int money, String result) {
    System.out.println("("+ ++count + ") " + result + " " + money + " penny(s)");
}
```

The output is below:

1.

```
Enter an amount in cents:
-2
The entered amount should be a positive integer.
```

2.

```
Enter an amount in cents:
17
This amount can be changed in the following ways:
        1) 1 dime, 1 nickel, 2 pennies
        2) 1 dime, 7 pennies
        3) 3 nickels, 2 pennies
        4) 2 nickels, 7 pennies
        5) 1 nickel, 12 pennies
        6) 17 pennies
```

3.

```
Enter an amount in cents:
11
This amount can be changed in the following ways:
        1) 1 dime, 1 penny
        2) 2 nickels, 1 penny
        3) 1 nickel, 6 pennies
        4) 11 pennies
```

Problem 2 (40 points): A Walk on the Hypercube:

For this problem, I am using recursion and iteration two ways to finish a walk on the hypercube. The path should go through all the corners.

In the `recursiveWalk()` method, I used bitwise operator(`<<`) to get `2^n` corners and save it into string. When it reaches to base I save "0" and "1" into string and use recursive to get `n-1`. Then I used a for loop to keep adding "0" and "1" into corner and the symmetry is very important.

EX:
When n = 3, there is 8 corner in string[] corners.

The string[] last is (00,01,11,10),

When i = 0, the string[] corners became(000, null, null, null, null, null, null, 110).

When i = 1, the string[] corners became(000, 001, null, null, null, null, 101, 100) and so on.

In the iterativeWalk() method, I tried a lot to find an equation.

For instance, **when n = 2, there would be four corners. If you write them down from 0 to 3 and convert them to 2 bits binary, it is 00, 01, 10, 11. And when you shift them to right by 1, the list became 00, 00, 01, 01. Now you xor between original binary and shift one, you will get the same corner and sequence for the walk (00^00=00, 01^00=01, 10^01=11, 11^01=10). And this equation corner = (i>>1)^i works for any n.** After that, I used num2Binary() method to help me convert the decimal corner to binary, this for loop execute once to get 1 bit binary and save the corner to queue.

The running time for recursiveWalk() is O(n).

The running time for iterativeWalk() is O(1).

The output is below:

1. n = 3

```
A Walk:
  000
  001
  011
  010
  110
  111
  101
  100
```

2. n = 4

A Walk:

0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

3. n = 5

A Walk:

00000
00001
00011
00010
00110
00111
00101
00100
01100
01101
01111
01110
01010
01011
01001
01000
11000
11001
11011
11010
11110
11111
11101
11100
10100
10101
10111
10110
10010
10011
10001
10000

Problem 3 (30 points): Augmented Stack with getMin:

For this problem, I build an ADT stack class with O(1) worst case time method push(S) pop() and getMin(S).

To make the getMin() method, I can pop every element out of the stack and compare, but in this way, the worst case time would not be constant. So in order to achieve the O(1) worst case time, I use two stacks, one for data and one for only minimum element. When the stackMin is empty, the element S would be pushed into both stack and stackMin. And if the next element is smaller or equal to minimum element, it will push to stackMin and the stackMin.peek() is always the minimum element. When pop something out, the element which got pop is the minimum element, it will pop from stackMin too.

The test case and output are below:

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AugmentedStack <Integer> stack1 = new AugmentedStack <Integer>();
        stack1.push(3);
        stack1.pop();
     // To test return null if it's empty in stack
    System.out.println("The top of stack is: " + stack1.getMin());
        stack1.push(3);
        stack1.push(5);
        stack1.push(4);
        stack1.push(2);
        stack1.push(1);
    System.out.println("The min of stack is: " + stack1.getMin());
    stack1.pop();
    System.out.println("The min of stack is: " + stack1.getMin());
    stack1.pop();
    System.out.println("The min of stack is: " + stack1.getMin());
    System.out.println("The top of stack is: " + stack1.top());}
/**
```

```
The top of stack is: null
The min of stack is: 1
The min of stack is: 2
The min of stack is: 3
The top of stack is: 4
```