EECS 2011O: Fundamentals of Data Structures
Assignment 2.
a2sol.pdf
User: jeffkw
Date: February 14, 2020

Enumeration of Coin Change Making:

Task: Develop a program in a class named Coins that includes a method with the signature **ways(int money)** that uses recursion to enumerate the distinct ways in which the given amount of money in cents can be changed into quarters, dimes, nickels, and pennies. Test your program from the **main()** method for various amounts of change by prompting the user to enter an amount in cents.

Goal: In this problem, I am required to create a program that uses recursion to find all of the distinct ways to cut the given amount of money in cents into coins.

Outputs:

```
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
```

```
Enter an amount in cents:
25
This amount can be changed in the following ways:
        1) 1 Quater
        2) 2 Dimes, 1 Nickel
        3) 2 Dimes, 5 Pennies
        4) 1 Dime, 3 Nickels
        5) 1 Dime, 2 Nickels, 5 Pennies
        6) 1 Dime, 1 Nickel, 10 Pennies
        7) 1 Dime, 15 Pennies
        8) 5 Nickels
        9) 4 Nickels, 5 Pennies
        10) 3 Nickels, 10 Pennies
        11) 2 Nickels, 15 Pennies
        12) 1 Nickel, 20 Pennies
        13) 25 Pennies
```

```
Enter an amount in cents:
32
This amount can be changed in the following ways:
        1) 1 Quater, 1 Nickel, 2 Pennies
        2) 1 Quater, 7 Pennies
        3) 3 Dimes, 2 Pennies
        4) 2 Dimes, 2 Nickels, 2 Pennies
        5) 2 Dimes, 1 Nickel, 7 Pennies
        6) 2 Dimes, 12 Pennies
        7) 1 Dime, 4 Nickels, 2 Pennies
        8) 1 Dime, 3 Nickels, 7 Pennies
        9) 1 Dime, 2 Nickels, 12 Pennies
        10) 1 Dime, 1 Nickel, 17 Pennies
        11) 1 Dime, 22 Pennies
        12) 6 Nickels, 2 Pennies
        13) 5 Nickels, 7 Pennies
        14) 4 Nickels, 12 Pennies
        15) 3 Nickels, 17 Pennies
        16) 2 Nickels, 22 Pennies
        17) 1 Nickel, 27 Pennies
        18) 32 Pennies
```

Code:

Variables:

```
final int CoinCents = 4; // the number of cent coins
final int[] coin = new int[] { 1, 5, 10, 25 }; // the value of the cents
final int[] count = new int[CoinCents]; //storing the number of cents that could be broken down
final String[] Plural = new String[] { "Pennies", "Nickels", "Dimes", "Quaters" };// Plural names of Coins
final String[] Singular = new String[] { "Penny", "Nickel", "Dime", "Quater" };//Singular names of Coins
int list; //number of possibilities to break down cents into coins
StringBuilder sb; //buiding the output
```

Methods:

public Coins()
public String PrintOut(int currentunit, int list)
public void ways (int money)
public void Helperways(int currentbalance, int currentunit, int list)

Imports:

import java.util.*;

Constructor:

```
/************** PrintOut **************/
/* Reinitializing counter variables.
 * */
public Coins() {
    this.list = 1; //list always start at 1
    for (int i = 0; i < CoinCents; i++) {
        this.count[i] = 0; //reset the counter of each value to 0.
    }

}
```

In this part of the program, I used the constructor to initialize the counting of the list to 1 every single time the constructor is called. This part of the program is mainly used to reinitialize every variable that is not a constant.

Recursion:

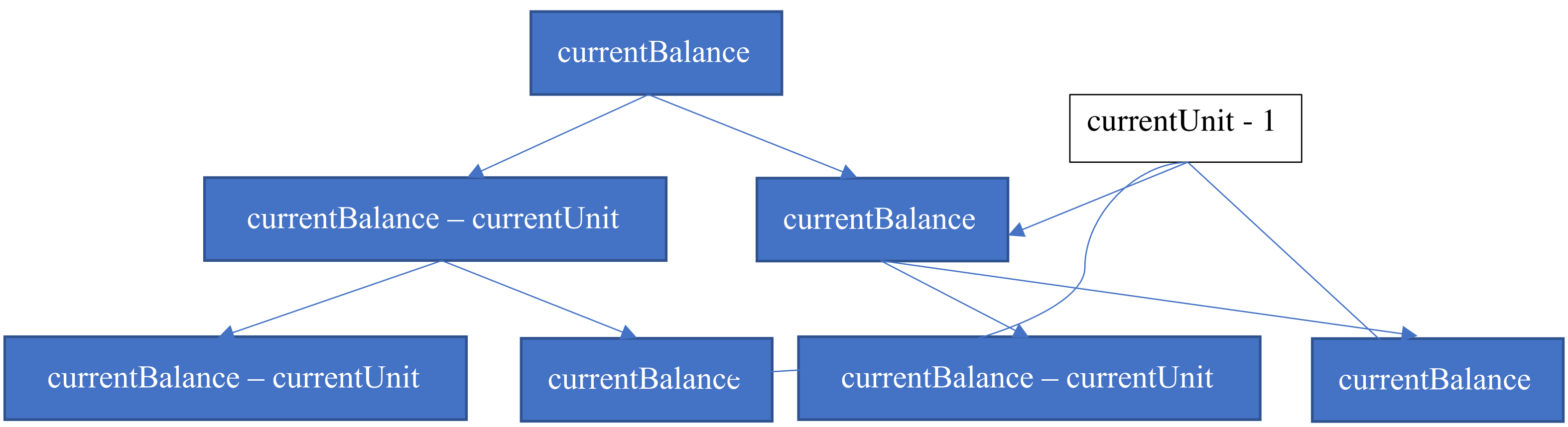
```
/*************Recursion*************/
/*
 * Recursive method that calls a helper method
 *
 * @param money
 *      cents inputed by user
 *
 * Pre-Condition: money has to be an integer
 * Post-Condition: money is positive
 */
public void ways(int money) {
    //if the money inputed is negative, throw an Exception.
    if (money <= 0) {
        throw new InvalidMoneyException("money needs to be positive");
    }
    System.out.println("This amount can be changed in the following ways:");
    Helperways(money, this.coin.length - 1, this.list);
}
```

The opening of the recursion making sure that the money inserted into the program is positive. This would then push the money of cents into the helper method.

Helper Recursion:

```java
/*************HelperMethod*************/
/*
 * Helper Recursive call
 * Pre-Condition: Money has to be an integer and positive
 * Post-Condition: organize multiple ways to group the number of cents into coins
 * @param currentbalance
 *      the balance of money when subtracted by the value of a coin
 * @param currentunit
 *      the current value of the coin
 * @param list
 *      The counter to determine how many times the recursion finds a possible solution.
 */
private void Helperways(int currentbalance, int currentunit, int list) {
    //return statement if the current unit reaches the smallest unit
    if (currentunit >= 0) {
        //print the possibility when the current balance is 0
        if (currentbalance == 0) {
            System.out.println(PrintOut(currentunit, this.list)); //print out result
            this.list++;
        } else {
            if (currentbalance >= coin[currentunit]) { //if the current balance is greater than the current unit coin
                this.count[currentunit]++; //increment that coin
                Helperways(currentbalance - coin[currentunit], currentunit, this.list); //recursively call the method with
                                                                                        //the difference of the current value and the current coin
                this.count[currentunit]--; //move back the count of the coin when finish the recursive call
            }
            //recursive call when the current balance is less than the current unit coin
            Helperways(currentbalance, currentunit - 1, this.list);
        }
    }
}
```

This part of the recursion would take the current balance and subtract it with a unit of cent coin, if the current balance is greater than the current unit. When this if statement runs, a counter for each coin would increment. When the current balance hits 0 or a unit less than the balance, the program would call the method again, but the current unit is smaller.



.
.
.
.

This type of algorithm would create branches to every possibility to organize the money into coins.

PrintOut:

```java
/************** PrintOut *************/
/* This prints out the results of the coins.
 * @param list
 *      the number of recursion it is on
 * @return String
 * 
 */
public String PrintOut(int list) {
    sb = new StringBuilder(); //creating a new StringBuilder everytime its called.
    System.out.print("  " + this.list + ") "); //indenting and getting the which recursion number its on
    for (int i = CoinCents - 1; i >= 0; i--) { //for loop descending so that it outputs the larger unit of coin first
        if (count[i] > 0) //if the count of coin is greater than 0, then output
            if (count[i] == 1) { //if the count of coin is 1, then output the singular word
                sb.append(this.count[i]);
                sb.append(" ");
                sb.append(Singular[i]);
                sb.append(", ");
            }
            else { //otherwise output the plural.
                sb.append(this.count[i]);
                sb.append(" ");
                sb.append(Plural[i]);
                sb.append(", ");
            }
    }
    sb.deleteCharAt(sb.length()-2); //Removing the comma of the last element in the string builder
    return sb.toString(); //returning the string of StringBuilder
}
```

This printout function prints out the total count for each coin with using StringBuilder. Using the append function to put in the number, that was incremented in the Helper recursion method, and then the name of the coin unit depending on the number that was incremented. If the count was 1, the name would be singular. If the count was greater than 1, the name would be plural.

A Walk on a HyperCube:

Task: Develop a program in a class named HyperCube which finds a pathway to every corner in the HyperCube. The unit hypercube in the n-dimensional space is the set of all points ($x1, x2, \ldots$, xn) in that space such that its coordinates are restricted to be $0 \le xi \le 1$ for $i = 1..n$. This hypercube has 2 n corners. Each coordinate of a corner is either 0 or 1. Each edge of the hypercube connects a pair of corners that differ in exactly one of their coordinates. A walk on the hypercube starts at some corner and walks along a sequence of edges of the hypercube so that it visits each corner exactly once. There are many such walks possible. The following figure shows an example for $n = 3$. The edges of the (hyper)cube along the walk are thickened.

Goal: In this program I need to create a program that will find every corner in the hypercube however, the way to find the corner by changing one of the coordinates thus making a neighbor of the corner.

Outputs:

```
Enter a dimension: 3
recursiveWalk:
A Walk:
 000
 100
 110
 010
 011
 111
 101
 001
```

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

```
Enter a dimension: 4
recursiveWalk:
A Walk:
 0000
 1000
 1100
 0100
 0110
 1110
 1010
 0010
 0011
 1011
 1111
 0111
 0101
 1101
 1001
 0001
```

```
iterativeWalk
A Walk:
 0000
 1000
 1100
 0100
 0110
 1110
 1010
 0010
 0011
 1011
 1111
 0111
 0101
 1101
 1001
 0001
```

```
Enter a dimension: 5
recursiveWalk:
A Walk:
 00000
 10000
 11000
 01000
 01100
 11100
 10100
 00100
 00110
 10110
 11110
 01110
 01010
 11010
 10010
 00010
 00011
 10011
 11011
 01011
 01111
 11111
 10111
 00111
 00101
 10101
 11101
 01101
 01001
 11001
 10001
 00001
```

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

Code:

Imports:

import java.util.Scanner
import java.util.ArrayDeque
import java.util.ArrayList

Classes:

public static class Corner
public static class HyperCube

Class:

Corner class: used to identify the corners of the hypercube but having the directions and the coordinates of the corners for the cube

HyperCube: used to find every possible pathway to find every single corner of the path by using recursion and iterations.

Corner Class Variable:

```
/*
 * Variables for the Corner class
 * coordinate for boolean to determine 0 or 1 (true = 1 & false = 0)
 * dimension is the dimension of the hypercube
 * visit is to determine if the corner has been read yet.
 */
boolean[] coordinate;
int dimension;
static ArrayList<String> visit;
```

Corner Class Methods:

public Corner(int dimension)
protected void move(int direct)
protected String Point()
protected Object clone(Corner corner)
protected boolean equals(Corner corner)

HyperCube Class Variable:

```
/*
 * HyperCube Class Variables
 * What dimension the user want the hypercube to be
 * origin is the starting point of the hypercube the coordinates are all false
 * visited is to check if the corner has been checked yet.
 */
int dimension;
Corner origin;
static ArrayList<String> visited;
```

HyperCube Class Methods:

public HyperCube(int dimension)
private void recursiveWalk()
private void HelperrecursiveWalk(Corner start)
private void printWalk(ArrayList<String> visited)
private ArrayDeque<String> iterativeWalk()
private void printWalk(ArrayDeque<String> visited)

Recursive Method:

```java
/*
 * HelperRecursiveWalk Method
 *
 * @param Corner start
 *      start is the variable that would be changing in this recursion is a parent
 * The method would move one of the coordinates of a corner and is a
 * child
 * A global ArrayList to keep track of which corner has been read
 * Recursively change the child node to a parent and move one of the coordinate
 * of the coordinate.
 *
 * This method is Depth First Search.
 */
private void HelperrecursiveWalk(Corner start) {
    if (!visited.contains(start.Point())) { // if the parent is in the ArrayList then recursion is done
        visited.add(start.Point()); // adding the parent to the ArrayList to make sure the method doesn't
                                    // StackOverFlow
        for (int i = 0; i < this.dimension; i++) { // for loop to go every neighbouring corner.
            Corner copy_start = (Corner) start.clone(start); // make a deep copy so the position of the parent
                                                             // corner doesn't change
            copy_start.move(i); // move the parent corner to a neighbouring corner/ to a child corner
            if (!visited.contains(copy_start.Point())) { // if the child is not in the ArrayList, Recursively call
                                                          // the method again.
                HelperrecursiveWalk(copy_start);
            }
        }
    }
}
```

This recursive method is a Depth First Search. It finds every combination in the given HyperCube. It goes through every possibility in the HyperCube, making the time Complexity very high. The best-case of this recursive Method is O(2^n). The worst-case O(n^(2^n)). To creating this method, I needed to constantly change the parent corner with a new one. So, every child is a parent to more children. This would create a tree that would infinitely branch out with no stop. To stop the recursive call, I just needed a list to store in the read parents. If every unique coordinate of corner is in the list, I could exit the recursive call and print out the possibility of a pathway. In this case I printed out the very first case thus always having the best case of O(2^n).

Worst-Case

Number of Nodes in 2 dimensional HyperCube:

$n = 1$ node = 3 $T(n) = 1 + 2^1$

$n = 2$ node = 7 $T(n) = 1 + 2^1 + 2^2$

$n = 3$ node = 15 $T(n) = 1 + 2^1 + 2^2 + 2^3$

$n = k$ $T(n) = 1 + 2^1 + 2^2 + 2^3 + \cdots + 2^k + \cdots + 2^{n-1} + 2^n$

$T(n) = \sum_{i=0}^{k} 2^i$

Number of Nodes in 3 dimensional HyperCube:

$n = 1$ node = 4 $T(n) = 1 + 3^1$

$n = 2$ node = 13 $T(n) = 1 + 3^1 + 3^2$

$n = 3$ node = 34 $T(n) = 1 + 3^1 + 3^2 + 3^3$

$n = k$ $T(n) = 1 + 3^1 + 3^2 + 3^3 + \cdots + 3^k + \cdots + 3^{n-1} + 3^n$

$T(n) = \sum_{i=0}^{k} 3^i$

Then,

$T(n) = \sum_{i=0}^{k} n^i$

$T(n) = \frac{1-n^{k+1}}{1-n}$ Geometric Series

The Height of the tree is 2^n

Therefore:

$O(n^{2^n})$

Iterative Method:

```
/*
 * iterativeWalk Method
 * @return ArrayDeque<String>
 *      return the ArrayDeque which is the pathway to go through every
 *      single point in the Cube.
 *
 * This method is a iterative method to the recursive method
 * Instead of a recursive call, it has a while loop with the same condition
 * in the recursive method. However in the for loop, the loop needs to break
 * so that the parent is able to change everytime the ArrayDeque doesn't contain
 * the child coordinate.
 *
 *
 */
private ArrayDeque<String> iterativeWalk() {
    Corner start = (Corner) this.origin.clone(this.origin); //starting point
    ArrayDeque<String> visited = new ArrayDeque<String>(); //storing elements that the program didn't read yet
    while (!visited.contains(start.Point())) { //keep looping until the parent starting is repeated
        visited.add(start.Point()); //push the parent into the ArrayDeque
        for (int i = 0; i < this.dimension; i++) { //for loop to go through every neighbour
            Corner copy_start = (Corner) start.clone(start); //keeping track where was parent corner
            copy_start.move(i); //moving making a child corner
            if (!visited.contains(copy_start.Point())) { //continue the for loop if the child is not in
                                                          //in the ArrayDeque
                start = (Corner) copy_start.clone(copy_start); //changing Child to Parent
                break; //break for loop
            }
        }
    }
    return visited;
}
```

This Iterative Method is identical to my recursive call. Instead of a recursive call I would have a while loop that would exit with the same condition of my recursive call base case. The main problem with this program is the change in child to parent. To solve this problem, I just needed to recreate a deep copy of the child equal that to the parent then break the for loop. Thus, the loop wouldn't continue to change the parent corner. The time complexity of this method is also O(n^(2^n)).

For loop: dependent on the dimension $O(n)$

While loop: dependent on how high the tree is, thus $O(2^n)$

If statement in for Loop: goes through once ever iteration of for loop $O(1)$

Conclusion:

$O(n^{2^n} + 1)$

$O(n^{2^n})$

Augmented Stack with getMin:

Task: Develop a program in a class named AugmentedStack which is an ADT that maintain a generic stack with comparable element type under the following operation, each requiring O(1) worst-case time:
s.push(x): Insert element x on top of s.
s.pop(): Remove and return the top element of s (if not empty). Return null if s is empty.
s.getMin(): Return the minimum element on s (if not empty). Return null if s is empty. (Upon return from this method, s should be in the same state as it was before this method was called. In particular, the minimum element is not removed from s.)

Also include the usual operations s.isEmpty() and s.top() with the same meaning as for conventional stacks that run in O(1) worst-case time.
Design and analyze a data structure that implements this ADT with the specified running times.

Goal: In this problem, I am required to create a program that is an ADT that has the same function as a stack with a getMin function.

Outputs:

```
//TEST_1//

Push: Hello
Push: 1
Push: 2
isEmpty? false
Top: 2
Stack: [Hello, 1, 2 ]
Pop: 2
New Stack: [Hello, 1 ]
getMin: 1
```

```
//TEST_2//

isEmpty? true
Top: null
getMin: null
Push: 500
Push: 356
Push: 1234567890
Stack: [500, 356, 1234567890 ]
isEmpty? false
Top: 1234567890
getMin: 356
```

```
//TEST_3//

isEmpty? true
Top: null
getMin: null
Push: MEME
Push: JOKE
Push: JEFF
Push: INTEGER
Stack: [MEME, JOKE, JEFF, INTEGER ]
isEmpty? false
Top: INTEGER
getMin: INTEGER
```

```
//TEST_4//

isEmpty? true
Top: null
getMin: null
Push: abc
Push: bcd
Push: abcd
Stack: [abc, bcd, abcd ]
isEmpty? false
Top: abcd
getMin: abc
```

```
//TEST_5//

Pop: abcd
Pop: bcd
Pop: abc
New Stack: []
isEmpty? true
Top: null
getMin: null
```

Code:

Variables:

```
    private Stack<S> smallest = new Stack<S>(); //a Stack to store in a current min value
    private Stack<S> stuff = new Stack<S>(); // putting in items into Stack
    static StringBuilder sb;
```

Having two stacks to store in the current min value and the items putting in.
The stack that get the smallest will push in new min values if the peek in the stack is bigger than the new value that is being pushed into the items stack. The other linked list is to act like a stack.

Methods:
public AugmentedStack()
public void push(S item)
public S pop()
public S getMin()
public Boolean isEmpty()
public S top()

Imports:
import java.util.Stack;

Constructor:

```
    /*************** Constructor ***************/
    public AugmentedStack() {
        sb = new StringBuilder();
    }
```

Initialize a new StringBuilder for output.

Push:

```java
public void push(S item) {
    stuff.push(item); // pushing item into stuff Stack
    if (smallest.isEmpty()) {
        smallest.push(item); // if the min Stack is empty, push in the new item
    } else {
        if (smallest.peek().compareTo(item) >= 0) { // if peek of the min Stack is greater than the item
            smallest.push(item); // push the item into the min Stack
        }
    }
}
```

Push in a new element into the stuff stack. If the stack that contains smallest values is empty, just push in the new element into the stack, otherwise, if the top of smallest stack is smaller than the item, push it onto the smallest.

Pop:

```java
/************** Pop **************/
/*
 * removing the top of the Stack
 *
 * @return null if element of Stack is empty or top unit of the Stack
 */
public S pop() {
    if (stuff.isEmpty()) {
        return null;
    } else {
        S min = stuff.pop();
        if (smallest.peek().equals(min)) { // if the peek of the min Stack is the popped Stack
            smallest.pop(); // remove the top of the min Stack as well
        }
        return min;
    }
}
```

If the stuff stack is empty, return null, otherwise pop the item in stuff, if the top of the smallest is equal to that item, pop it as well and return the item that was popped.

getMin:

```java
/************** getMin **************/
/*
 * @return the smallest value of the Stack if empty, return null
 */
public S getMin() {
    if (smallest.isEmpty())
        return null;
    return smallest.peek(); //return the peek of the smallest which is the min value in stuff
}
```

If ‘smallest’ is empty return null. Returns the peek of smallest, which is always going to be the smallest value in stuff since the ‘smallest’ only pushes elements into the stack if the peek of the current ‘smallest’ is greater than the item that was pushed into ‘stuff’.

isEmpty:

```java
/************** isEmpty **************/
/*
 * @return if Stack is empty true, else false
 */
public boolean isEmpty() {
    return stuff.isEmpty();
}
```

Returns if stack stuff is empty.

top:

```java
/************** top **************/
/*
 * @return the top of the Stack if empty, return null
 */
public S top() {
    if (stuff.isEmpty())
        return null;
    return stuff.peek();
}
```

If there are no elements in stuff, return null. Return the peek of stuff.