EECS 2011O: Fundamentals of Data Structures

Assignment 2.

a2sol.pdf

User: jeffkw

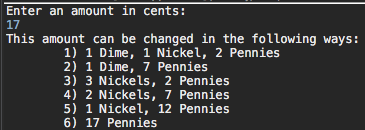
Date: February 14, 2020

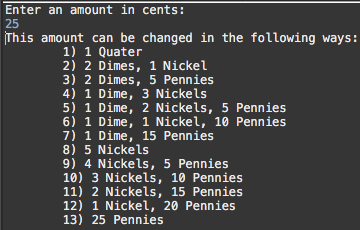
**Enumeration of Coin Change Making:**

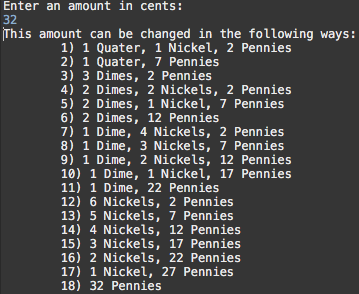
Task: Develop a program in a class named Coins that includes a method with the signature **ways(int money)** that uses recursion to enumerate the distinct ways in which the given amount of money in cents can be changed into quarters, dimes, nickels, and pennies. Test your program from the **main()**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

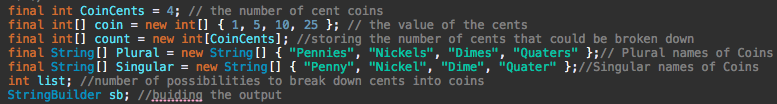






Code:

Variables:



Methods:

public Coins()

public String PrintOut(int currentunit, int list)

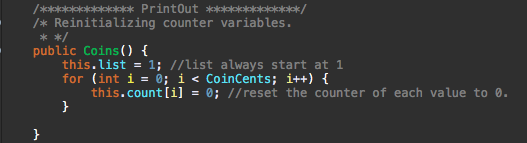
public void ways (int money)

public void Helperways(int currentbalance, int currentunit, int list)

Imports:

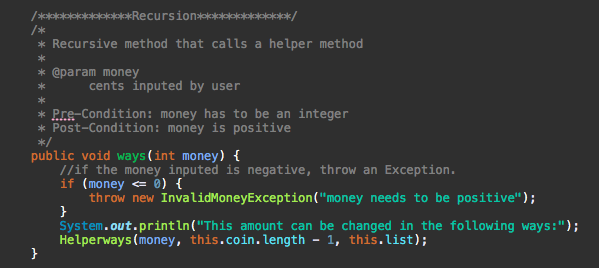
import java.util.\*;

Constructor:



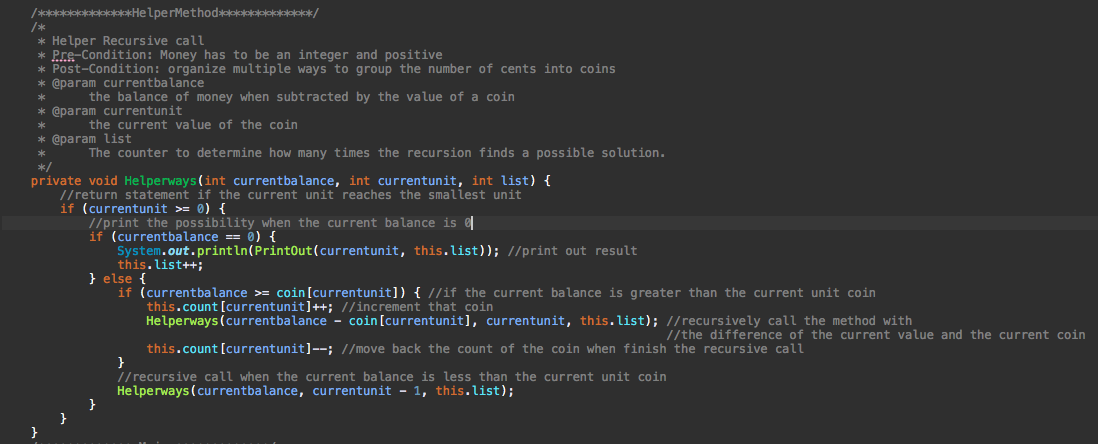
In this part of the program, I used the constructor to initialize the counting of the list to 1 every single time the constructor is called. This part of the program is mainly used to reinitialize every variable that is not a constant.

Recursion:



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:



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.

currentBalance

currentBalance – currentUnit

currentBalance

currentBalance – currentUnit

currentBalance – currentUnit

currentBalance

currentBalance

currentUnit - 1

.

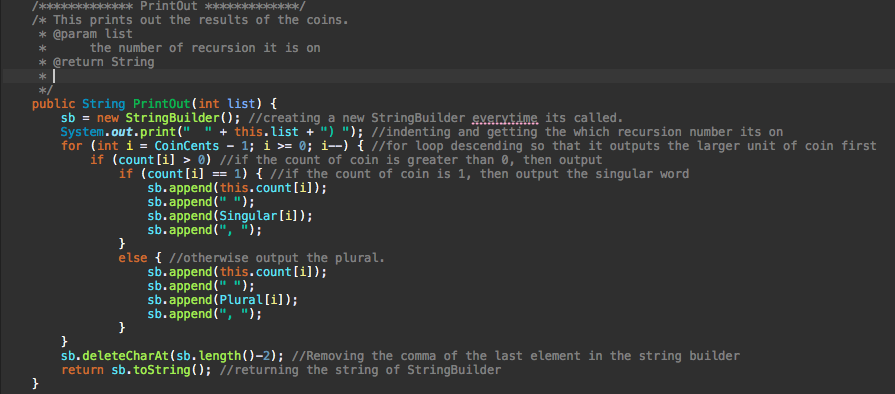
.

.

.

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

PrintOut:



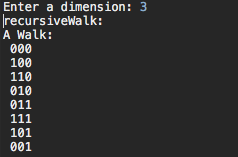
This printout function prints out the total count for each coin with using StringBuilder. Using the append function to put in the number, that was incremented in the Helper recursion method, and then the name of the coin unit depending on the number that was incremented. If the count was 1, the name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

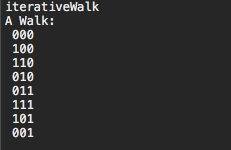
**A Walk on a HyperCube:**

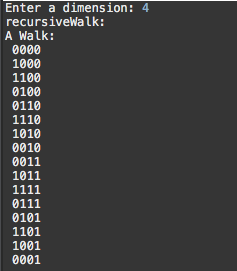
Task: Develop a program in a class named HyperCube which finds a pathway to every corner in the HyperCube. The unit hypercube in the 𝑛-dimensional space is the set of all points (𝑥1, 𝑥2, … , 𝑥𝑛) in that space such that its coordinates are restricted to be 0 ≤ 𝑥𝑖 ≤ 1 for 𝑖 = 1. . 𝑛. This hypercube has 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.

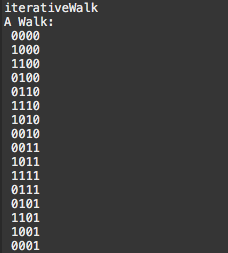
Goal: In this program I need to create a program that will find every corner in the hypercube however, the way to find the corner by changing one of the coordinates thus making a neighbor of the corner.

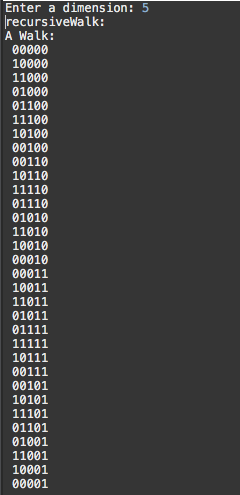
Outputs:

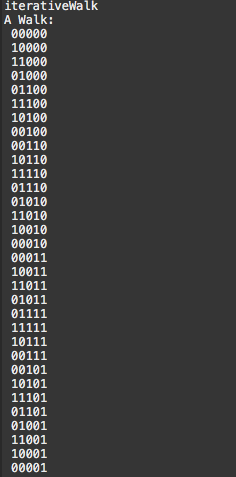












Code:

Imports:

import java.util.Scanner

import java.util.ArrayDeque

import java.util.ArrayList

Classes:

public static class Corner

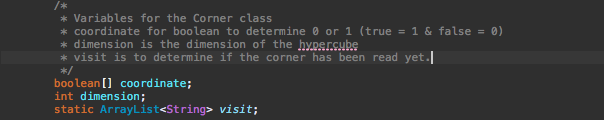
public static class HyperCube

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:



Corner Class Methods:

public Corner(int dimension)

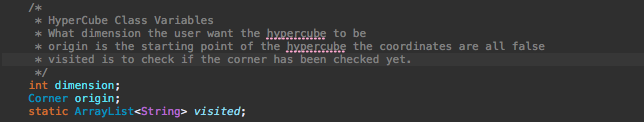
protected void move(int direct)

protected String Point()

protected Object clone(Corner corner)

protected boolean equals(Corner corner)

HyperCube Class Variable:



HyperCube Class Methods:

public HyperCube(int dimension)

private void recursiveWalk()

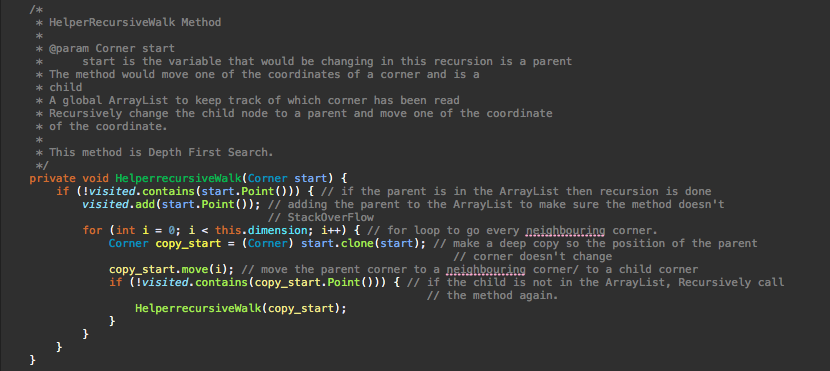
private void HelperrecursiveWalk(Corner start)

private void printWalk(ArrayList<String> visited)

private ArrayDeque<String> iterativeWalk()

private void printWalk(ArrayDeque<String> visited)

Recursive Method:



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:

node = 3

node = 7

node = 15

Number of Nodes in 3 dimensional HyperCube:

node = 4

node = 13

node = 34

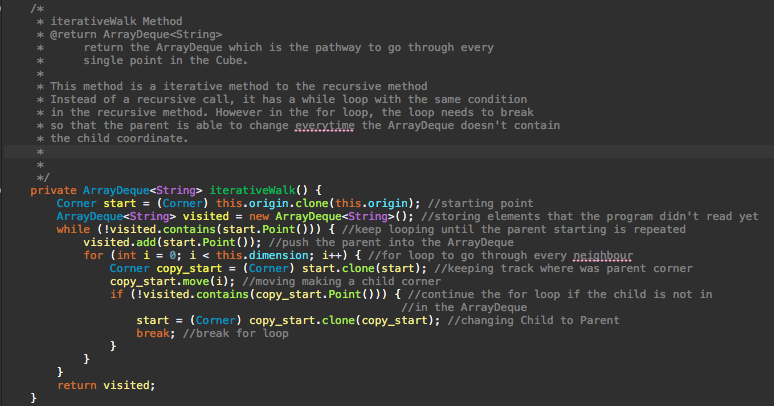
Then,

Geometric Series

The Height of the tree is

Therefore:

Iterative Method:



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

While loop: dependent on how high the tree is, thus

If statement in for Loop: goes through once ever iteration of for loop

Conclusion:

**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.

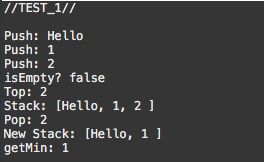
s.pop(): Remove and return the top element of s (if not empty). Return null if s is empty. s.getMin(): Return the minimum element on s (if not empty). Return null if s is empty. (Upon return from this method, s should be in the same state as it was before this method was called. In particular, the minimum element is not removed from s.)

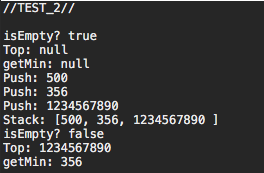
Also include the usual operations s.isEmpty() and s.top() with the same meaning as for conventional stacks that run in O(1) worst-case time.

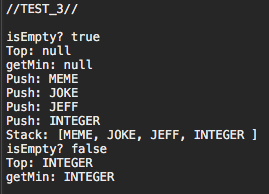
Design and analyze a data structure that implements this ADT with the specified running times.

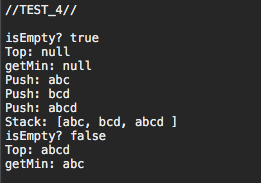
Goal: In this problem, I am required to create a program that is an ADT that has the same function as a stack with a getMin function.

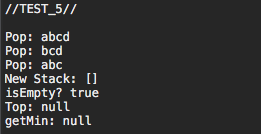
Outputs:











Code:

Variables:



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()

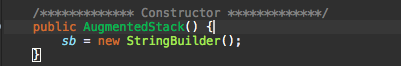
public Boolean isEmpty()

public S top()

Imports:

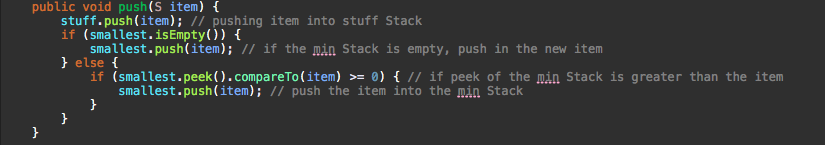
import java.util.Stack;

Constructor:



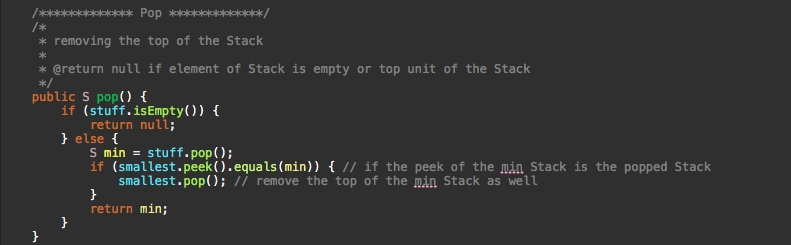
Initialize a new StringBuilder for output.

Push:



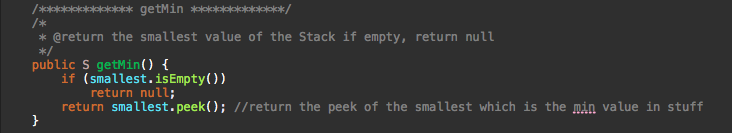
Push in a new element into the stuff stack. If the stack that contains smallest values is empty, just push in the new element into the stack, otherwise, if the top of smallest stack is smaller than the item, push it onto the smallest.

Pop:



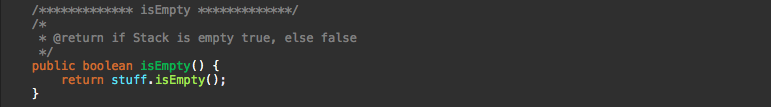
If the stuff stack is empty, return null, otherwise pop the item in stuff, if the top of the smallest is equal to that item, pop it as well and return the item that was popped.

getMin:



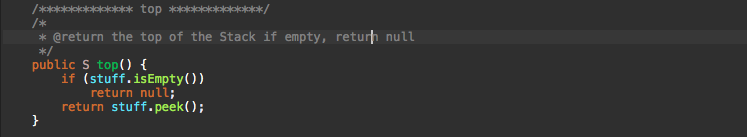
If ‘smallest’ is empty return null. Returns the peek of smallest, which is always going to be the smallest value in stuff since the ‘smallest’ only pushes elements into the stack if the peek of the current ‘smallest’ is greater than the item that was pushed into ‘stuff’.

isEmpty:



Returns if stack stuff is empty.

top:



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