EECS2101A **Assignment 2 Solutions** **Fall 2023**

Instructor: Andy Mirzaian

Our solution directory (also the package name) is **A2sol**. The file you are reading is **a2sol.pdf** in that directory. The java codes are best viewed by Eclipse.

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

Java Code: [Coins.java]

Design Idea:

The code and its comments describe the underlying idea. Carefully read the cautionary notes near the top of the code. The recursive helper method ***enumerate(rem, maxIndx, more)*** is the heart of the algorithm. We still have **rem** (remainder) amount to convert to coins. The restriction is that we are not allowed to use any coin of larger index indicated by **maxIndx**. We also need to add **more** to the current coin count at index maxIndx. We now divide the enumeration into two categories: those that use at least one more coin of type maxIndx, versus those that do not. These are handled by two recursive calls. The boundary conditions correspond to the cases **rem** $\leq$ 0.

Note: our main method tested only the Canadian coin system. The reader is welcome to test our program with any other coin system too.

Test I/O results on the console:

```
Console
<terminated> Coins [Java Application] C:\Program Files\Java\jre1.8.0_
Enter an amount in cents:
32
This amount can be changed in the following ways:
        1)      1 quarter, 1 nickel, 2 pennies
        2)      1 quarter, 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
```

Time Complexity:

Let N be the number of different coin types. In the Canadian system $N = 4$. Let $E(m)$ denote the number of ways to make change for m units of currency. The running time of our algorithm is in the order of output size $\Theta(N * E(m))$. **This is the extent of analysis I was expecting from you.**

More detailed analysis appears below. Here is an experimental sample for the Canadian system:

m	17	32	1000	2000	3000	4000	5000
$E(m)$	6	18	142,511	1,103,021	3,681,531	8,678,041	16,892,551

Claim: $E(m) = \Theta(m^3)$ for the Canadian coin system.

Proof: To prove this, we follow the recursive structure of our algorithm and set up a recurrence relation and then solve it. More specifically, let us define

$E_i(m) =$ the number of ways we can make change for m cents without using any coin higher indexed than i (where $i \in \{0,1,2,3\}$ indicates pennies to quarters).

We see that $E(m) = E_3(m)$. So we need to asymptotically bound the latter.

Based on the recursive structure of our algorithm, we can express the following recurrence relation:

$$E_i(m) = \begin{cases} E_i(m - denom[i]) + E_{i-1}(m) & if\ m > 0\ \ and\ \ i \geq 0 \\ 1 & if\ m = 0\ \ and\ \ i \geq 0 \\ 0 & if\ m < 0\ \ or\ \ i < 0 \end{cases}$$

We solve this recurrence for each value of index i in increasing order (and arbitrary m). We already know the base cases. So, let's look at the recurrence (the first line).

For $m \geq 0$ we have:

$$E_0(m) = E_0(m-1) = E_0(m-2) = \cdots = E_0\,(m-k) = \cdots = E_0(0) = 1.$$

$$E_1(m) = E_1(m-5) + E_0(m) = E_1(m-5) + 1 = E_1(m - 2*5) + 2 = E_1(m - 3*5) + 3$$

$$= \cdots = E_1(m-5k) + k = \cdots = \underbrace{E_1\big(m - 5(\lfloor m/5 \rfloor + 1)\big)}_{=0} + \lfloor m/5 \rfloor + 1 = \lfloor m/5 \rfloor + 1\,.$$

$$E_2(m) = E_2(m-10) + E_1(m) = E_2(m-10) + \lfloor m/5 \rfloor + 1 = \cdots \quad \text{(after expansion)}$$

$$= \left(1 + \left\lfloor \frac{m}{10} \right\rfloor\right)\left(1 + \left\lfloor \frac{m}{10} + \frac{1}{2} \right\rfloor\right) \quad = \quad \left(1 + \frac{m}{10}\right)^2 \pm O(m)\ .$$

$$E_3(m) = E_3(m-25) + E_2(m) = E_3(m-25) + \left(1 + \frac{m}{10}\right)^2 \pm O(m) = \ \cdots \text{(after expansion)}$$

$$= \frac{1}{7500}\ m^3 \ \pm\ O(m^2) \quad = \quad \Theta(m^3). \qquad \blacksquare$$

Problem 2 (40 points): A Walk on the Hypercube:
Java Codes: [Hypercube.java] **and** [Queue.java]

Design Idea: The code and its comments describe the underlying idea. The recursive helper method rWalk and the iterative method iterativeWalk are the key methods of the implementation. We implemented our own version of FIFO Queue for use in the itarative walk.

Time Complexity: Running time of our recursive algorithm is $\Theta(n2^n)$, since $\Theta(2^n)$ corners are generated and each corner requires $\Theta(n)$ time to be reported. That dominates the time complexity of the entire recursive algorithm. The running time of our iterative algorithm is also $\Theta(n2^n)$. The outer loop iterates $\Theta(n)$ times. In iteration d of the outer loop, size of the queue is $\Theta(2^d)$ which doubles during the inner loop. The inner loop takes $\Theta(n2^d)$ time (each iteration of it has to instantiate a new corner). Therefore, the total time spent over all iterations is $\Theta\left(\sum_{d=0}^{n-1} n2^d\right) = \Theta\left(n \sum_{d=0}^{n-1} 2^d\right) = \Theta(n2^n)$, which dominates rest of the algorithm.

Note: Wikipedia Gray Code describes bit-manipulation based algorithms (bit shifting, arithmetic exponentiation, etc.). Such an approach without detailed and convincing time complexity analysis will get less than full credit.

Space Complexity: The memory space used by the iterative method is $\Theta(n2^n)$, since all corners are stored in the queue and each takes $\Theta(n)$ space. That's really bad!

Exercise: How much space does the recursive algorithm use, including the recursion stack (i.e., the recursion depth is n)?

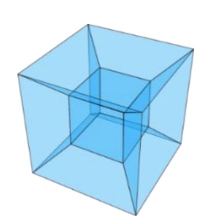
Test I/O results on the console: Only results for n = -6, 3, 4 are shown below.

```
Console
<terminated> Hypercube (1) [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe
Enter a dimension n:
-6
        java.lang.IllegalArgumentException: Dimension must be a positive integer
```

```
Console
<terminated> Hypercube (1) [Java Applicatior
Enter a dimension n:
3
Recursive walk of the 3D hypercube:
        000
        100
        110
        010
        011
        111
        101
        001
Iterative walk of the 3D hypercube:
        000
        001
        011
        010
        110
        111
        101
        100
```

```
Console
<terminated> Hypercube (1) [Java Applicatior
Enter a dimension n:
4
Recursive walk of the 4D hypercube:
        0000
        1000
        1100
        0100
        0110
        1110
        1010
        0010
        0011
        1011
        1111
        0111
        0101
        1101
        1001
        0001
```

```
Iterative walk of the 4D hypercube:
        0000
        0001
        0011
        0010
        0110
        0111
        0101
        0100
        1100
        1101
        1111
        1110
        1010
        1011
        1001
        1000
```



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

Java Code: [AugmentedStack.java]

Design Idea:
We let the *AugmentedStack s* be a pair of ordinary stacks (*eStk*, *minStk*), where *eStk* contains the elements of *s*, and *minStk* contains its *one-sided minima*. An element *e* in *eStk* is a one-sided minimum if its value is minimum among elements from *e*'s position to bottom of *eStk*. In this way, the minimum value is always at the top of *minStk*. See the illustrative example below.

eStk:	20	49	35	28	15	16	23	10	88	7	9	7	10	12	TOP
minStk:	20				15			10		7		7			TOP

To push an element *e* on top of *s*, we push *e* onto *eStk*, and if minStk is empty or *e* $\le$ top element of minStk, then we also push *e* on *minStk*. The *pop* operation on s is done by popping from *eStk*, and if that value is $\le$ top element of minStk, we also *pop* from *minStk*. The *getMin* operation on s simply returns the top element of minStk. The operations pop and getMin return null if s is empty. The operations top and isEmpty on s are also obvious.

Time Complexity: Each of the AugmentedStack methods **push(), pop(), getMin(), top(), isEmpty()** take O(1) worst-case time (assuming the generic compareTo() method takes O(1) time).

Test I/O results on the console:

```
Console
<terminated> AugmentedStack [Java Application] C:\Program Files\Java\jre1.8.0
The following elements will be pushed onto the AugmentedStack:
20    49    35    28    15    16    23    10    88    7    9    7    10    12

The results of getMin() after each push:
20    20    20    20    15    15    15    10    10    7    7    7    7    7

The results of repeated getMin(), pop():
        getMin   pop
        7        12
        7        10
        7        7
        7        9
        7        7
        10       88
        10       10
        15       23
        15       16
        15       15
        20       28
        20       35
        20       49
        20       20
        null     null
```