CS14 Winter 2016 Lab 2
More Martian Money

January 22, 2016

DUE: Submit by iLearn by 11:59pm Sunday, January 31, 2016
LATE DEADLINE (50% penalty) 11:59pm Monday, February 1, 2016

1 Introduction

Chapter 2 in the text presents the idea of studying the time complexity of different algorithms for solving the same problem, and illustrates the concept using four algorthms for solving the maximum subsequence sum problem. In this assignment, you will be applying these complexity analysis concepts to the following three algorithms for converting money from Eath notation to Martian Square vector notation:

1. **Basic brute force algorithm** with *no improvements* described in Lab 1, section 3.
2. **Improved brute force algorithm** with *all three* of the improvements described in Lab 1, section 3, plus **Lagrange's four-square theorem**[1] which was mentioned in the Piazza discusssions for Lab 1. Lagrange's theorem states that *every* natural number can be written as the sum of *four* integer squares. Since some of those integers might be zero (and contribute nothing to the total), we can conclude that *the optimal Martian coin vector contains at most four coins.* Thus:
   - You can give up early on any partially-completed Martian coin vector that still has a non-zero remainder after including four or more coins, and
   - The highest-denomination coin in any optimal vector must be at least one quarter of the total transaction amount — or else four coins can't make up the total amount.
3. **Bottom-up caching algorithm**, which is described in Section 3 of this assignment.

2 Earth to Martian square currency conversion tables

In Lab assignment 1, you implemented an algorithm for converting from a *single amount* in Earth notation to an optimal vector in Martian Square vector notation. This time, your task is to produce a *conversion table* of a specified size, N. More specifically, each of your three algorithms starts with *one* input, N, representing the largest amount in the conversion table, from which it generates the following *vector-of-vectors*

```
vector <vector <int>> MU;
// MU[i] is a vector, giving an optimal Martian coin vector for total amount i
// MU[i][j] is an integer, giving the number of (j*j)-mu coins in that optimal vector
```

where `MU.size() == N+1`, and the vectors `MU[1]` through `MU[N]` contain the respective optimal Martian Square vector representations for all amounts between 1 and N.

[1] `https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem`

3 Bottom-up caching algorithm

Instead of converting a single currency transaction from Earth notation to Martian Square notation, let's consider the larger problem of building an entire conversion table covering *every* transaction amount startin from EA 1 and ending with EA N. If we build the table in "bottom-up" order, then we can use the known results for smaller transaction amounts to speed up the conversion process for the larger transaction amounts, which are handled later in the algorithm.

To see how these cached results for smaller transaction amounts is helpful, let us return to the example of converting EA 13 to Martian Square notation from Lab 1, Section 5. In this case, assume that *before* we start the calculation of the optimal vector for EA 13, we have *already found* optimal vectors for EA 1 through EA 12:

EA 1 = MU 1
EA 2 = MU 2
EA 3 = MU 3
EA 4 = MU 0,1
EA 5 = MU 1,1
EA 6 = MU 2,1
EA 7 = MU 3,1
EA 8 = MU 0,2
EA 9 = MU 0,0,1
EA 10 = MU 1,0,1
EA 11 = MU 2,0,1
EA 12 = MU 0,3

and stored them in the vectors `MU[1]` through `MU[12]`. Now consider the sequence of Martian vectors evaluated during the search for the optimal solution for EA 13, as shown in the verbose output from the basic algorithm:

```
EA 13 could be represented as:
MU 0,1,1
MU 4,0,1
MU 1,3
MU 5,2
MU 9,1
MU 13
A legal representation for this amount is:
MU 0,1,1
```

Notice that the brute-force algorithm evaluates a total of six distinct vectors in its search, out of which the *highest-denomination coin selected* was the 9μ coin for the first two output vectors, the 4μ coin for the next three output vectors, and the 1μ coin for the last output vector. Clearly, the case where the largest coin is 1μ can't be optimal because it would violate Lagrange's theorem by requiring more than four coins in the optimal vector. Thus, all we need to consider is starting with *one* 9μ or 4μ coin in order to reduce the remaining transaction amount to a value *for which we have already found the optimal Martian coin vector.*

In other words, to find the optimal Martian vector representation for EA 13, we have only two cases to consider:

- We start with a 9μ coin, so the remaining amount is EA 4. By assumption, we already know that the optimal Martian vector for EA 4 is MU 0, 1. Thus, by selecting a 9μ coin, the best Martian vector we can obtain will be MU 0, 1, 1 — which has a total of two coins.
- We start with a 4μ coin, so the remaining amount is EA 9. By assumption, we already know that the optimal Martian vector for EA 9 is MU 0, 0, 1. Thus, by selecting a 4μ coin, the best Martian we can obtain must be MU 0, 1, 1 — which again has a total of two coins.

The final result for EA 13 is obtained by choosing the alternative from the above cases that gives us the smallest total number of coins. In this case, we have a tie, since both lead to the same two-coin vector.

We note that the previous search for EA 13 can be reduced to a single case using the following observation. Unless the original transaction amount is a perfect square (which is not true for EA 13), then the optimal vector must include *at least* two coins. Thus, since the first case produces a solution with *exactly* two coins, it must be optimal. More generally,

- if the largest coin does not give us a one-coin solution, then the optimal vector must have at least two coins,
- if no coin whose value is at least half the total amount gives us a two-coin solution, then the optimal vector must have at least three coins, and
- if no coin whose value is at least one-third of the total amount gives us a three-coin solution, then the optimal vector must have four coins.

In order to implement this bottom-up caching algorithm efficiently, you should save the number of coins in the optimal Martian coin vector for each transaction amount, and store it in a separate vector

```
vector <int> MU_length;
```

where `MU_length[i]` is an integer, giving the total number of coins in the optimal Martian vector for transaction amount `i`. Given this information, you don't need to construct an actual vector until you have found the optimal one.

4 Estimating Program Complexity from Measurement Experiments

In lecture, we discussed how to estimate the running time for a program by estimating the total number of basic statements evaluated during program execution. In this assignment, you will be modifying the three algorithms to generate their own complexity estimates during execution, according to the following rule.

> We say that a program consumes one "tick" of execution time whenever it chooses *any specific value* for *one* coin denomination while searching for a possible Martian coin vector. For example, the brute-force algorithm consumes one "tick" for each choice of the number coins of the nth denomination, one "tick" for each choice of the number of coins of the $(n-1)$st denomination, and so on. Similarly, the bottom-up algorithm consumes one "tick" each time it chooses the denomination of the starting coin.

The easiest way to keep track of the total number of "ticks" is to add it as an extra call-by-reference parameter to your functions, say `TotalTicks`. The main program should initialize it to zero before making the first function call, and thereafter its value should get updated according the rule above.

To see how this tick-counting works, let us return to the example of converting EA 13 to a Martian coin vector. For convenience, Figure 1 shows how all the recursive calls in the basic brute-force method are related to each other. (This structure is called a *decision tree*. We will be spending a lot of time studying trees later in the course.)

At the start (top of diagram), the main program makes the first call to your recursive function at the point **A** where the 9μ coin is active and remaining transaction amount is EA 13. The function executing at point **A** may choose to include either zero or one 9μ coins in the vector. First, it chooses one 9μ coin (left branch), which costs one "tick" and triggers a recursive call to the point **B**, where the 4μ coin is active and the remaining transaction amount is EA 4. Next, the function executing at point **B** chooses one 4μ coin at a cost of one "tick" and triggers a recursive call to the point **C**, where the 1μ coin is active and the remaining transaction amount is EA 0. At this point, the function executing at point **C** chooses its only option of zero 1μ coins, which costs one "tick" and generates the first complete Martian vector MU 0,0,1 at a total cost so far of three "ticks".

After the recursive call to **C** has finished, execution at **B** resumes by choosing zero 4μ coins, which costs one "tick" and triggers a recursive call to **D** where the 1μ coin is active and the remaining transaction

amount is EA 4. Once again **D** chooses its only option of four 1μ coins, which costs one "tick" and generates the second complete Martian vector MU 4,0,1 — for a total cost so far of five "ticks".

At this point neither **D** nor **B** have any further choices available, so the execution of the function at point **A** resumes by choosing zero 9μ coins (right branch), which costs one more "tick" and triggers a recursive call to point **E**. Following a similar pattern, **E** calls **F** which generates the Martian vector MU 1,3 bringing the total number of "ticks" so far to eight. Next, we reach a total of ten "ticks" when **G** generates MU 5,2, then reach a total of twelve "ticks" when **H** generates MU 9,1, and finally fourteen "ticks" when **I** generates MU 13, which is the last of the six possible Martian vectors for this example.

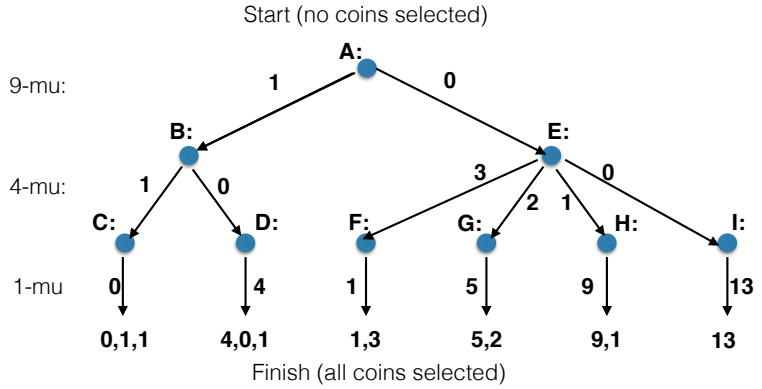

Figure 1: Structure of recursive calls for brute-force algorithm and EA 13.

Depending on how you implement the improved brute-force algorithm, **D** can skip the case of four 1μ coins, and **E** can skip the cases of three or two 4μ coins because the final result can't have fewer than the two coins in the best solution we already found. Also, **E** can skip the case of zero 4μ coins because the highest-denomination coin in the result would be smaller than one-quarter of the amount. And, finally, the function executing at **C** could declare victory (giving a total "tick" count of three) because its solution uses exactly two coins and no single-coin solution exists or **A** would never have made its recursive call to **B**.

5 Program Input and Output

Write a program that generates an Earth to Martian currency conversion table using one or more of the three algorithms described in Section 1.

Your program must accept the following *command-line arguments* given in the specified order:

1. The first parameter is a positive integer, say N, giving the largest amount included in your converion table. For example, if the first parameter is `100` then your program should generate a conversion table for all values from EA 1 to EA 100, inclusive.

2. At least one of the following *algorithm flags* presented in any order:

   - `-dumb` When present, indicates you should build a/the conversion table using the **basic brute force algorithm**.
   - `-smart` When present, indicates you should build a/the conversion table using the **improved brute force algorithm**.
   - `-cache` When present, indicates you should build a/the conversion talbe using the **bottom-up caching algorithm**.

   Calling the program without any algorithm flags, with multiple copies of the same algorithm flag, or with an unrecognized algorithm flag is an error, which should cause your program to terminate after printing an error message to the standard output.

3. The last parameter is the name of the specified output file with *no extension*, such as "`mmolle_prog2`".

Your program will use the specified output file name to generate *two* similarly-named text files with different extensions:

- The file name with extension "`.out`" will be used to print one copy of the complete Earth to Martian currency conversion table using the same format as Lab 1, so the beginning will look like this:

  ```
  EA 1 = MU 1
  EA 2 = MU 2
  EA 3 = MU 3
  EA 4 = MU 0,1
  ```

  Don't print multiple copies of the conversion table if the command line arguments include more than one algorithm flag. In that case, however, it would be helpful to check that the conversion tables generated by different algorithms are the same and print a warning message each time they produce different translations for the same amount.

- File name with extension "`.csv`" will be used to print the total "tick" data for each of the selected algorithms, shown as comma-separated vectors which can be read by a spreadsheet such as MS Excel, OpenOffice, Macintosh Numbers, etc. For each of the selected conversion algorithms, the second output file will include three lines of output in the following format:

  1. line one prints the *algorithm flag* that was used to generate this table.
  2. line two is a comma-separated, ordered list of the integers from 1 to the specified maximum, N.
  3. line three is a comma-separated list of *cumulative total "tick" counts* generated by this algorithm as it builds the conversion table.[2] The first number is the total "tick" count after the first entry `EA 1 = MU 1` has been added to the conversion table; the second number is the total "tick" count that results from adding *both* `EA 1 = MU 1` and `EA 2 = MU 2` to the conversion table; and so on until the Nth number is the total "tick" count for adding *all* N entries to the conversion table.

Note that these program specifications allow you to adjust the value of N according to the efficiency of each algorithm. Increasing the value of N makes the shape of the function $T(N)$ easier to determine. However, the inefficiencies of the basic brute-force algorithm may force you to use a smaller value of N to make sure the program can complete its task without runing out of resources.

6 Report

Use the data from the second output file to generate a written report showing the complexity of the three algorithms in "Big-Oh" notation. Your report should include at least one graph for each of the three functions, showing the reported total tick count and upper-bounding Big-Oh function, plus a couple of paragraphs explaining how you got these results. Please submit the report as a PDF document!

HINT: Recall the definition of "Big-Oh" functions that $T(N)$ is $O(f(N))$ if $T(N) \leq a \cdot f(N)$ for all $N \geq N_0$. Thus, suppose you create a graph showing the total tick count, $T(N)$, as a function of table size, N, i.e., $y = T(x)$. Now you wish to determine whether $T(N)$ can be classified as $O(N)$ say. In this case, simply add a series of curves to the same graph using different values for the constant a, such as: $y = x$, $y = 2 \cdot x$, ..., $y = 2^k \cdot x$. (In this example, the family of curves would be straight lines with different slopes.) Now look at how the original function $T(N)$ compares to the family of curves. If $T(N)$ is $O(N)$ then for large values of N its slope would obviously be less than some of the steeper curves in the family. However, if $T(N)$ is actually $\Theta(N^2)$, then its slope would keep getting steeper as N increases, so it eventually cuts across every curve of the family no matter what value you choose for the constant a.

[2]The kth value output by your program could be either the Kth *individual* "tick" count (representing the number of "ticks" required to generate the Kth individual entry in the conversion table), or the Kth *cumulative* "tick" count (representing the total number of "ticks" required to generate all of the first K entries in the conversion table). Although the interpretations of the two sets of numbers is slightly different, we will focus on the cumulative "tick" counts because a graph of these values will be much smoother than the individual "tick" counts and hence easier to interpret.