CS5800 Algorithms	*Out: October 15, 2019*
Solutions for Problem Set 5	
Name: Instructor's solutions	*Due: October 22, 2019*

Preamble

Every problem in this problem set is a dynamic programming problem. You first find an algorithm and write-up the solution using the template from the recitations (reproduced below). Each problem also comes with a programming component that is a matter of simple implementation once you have the algorithm. Here's the link for the Hackerrank contest: `www.hackerrank.com/cs5800-fall-2019-programming-assignment-5`.

We give a description of the problem you need to solve. The programming component may have a *slightly* different setup to make test cases and so on. Make sure you write-up a solution for the problem given here, not for variant in the programming component. Look at the challenges on the contest for details about input/output formats.

Dynamic Programming Solution Guidelines

Dynamic programming can be very tricky and this template will help guide you through solving new problems. Get in the habit of going through the list and filling everything out step by step. We will grade harshly on missing items. And if there's no english description of the two items mentioned below then the solution will get an **AUTOMATIC 0** on homeworks and exams with no exceptions.

1. (**AUTOMATIC 0 IF MISSING**) English description of the
   (a) problem you are actually solving with your recursion.
   (b) logic behind your recursion.
2. The actual recursion. And don't forget the base cases!
3. Brief description of how an iterative algorithm would loop through the memoization array and fill the values (make sure your order makes sense, and that values are already filled when you call them). Pseudocode isn't required, just a couple sentences.
4. The number of subproblems you have to solve.
5. How much time each subproblem takes to solve (usually constant or linear).
6. Final running time.
7. Final value we have to return.

Problem 1 (Adding of Fibonacci's) 10+10

For a given number N, you need to find the sum of all even Fibonacci numbers that are less than N. For this, assume that the Fibonacci numbers start with 1 and 2.

Note: The programming component asks this for T such numbers, but in here you only need to write the solution for when you are given a single number N.

Solution:

There are at least three approaches to solve this. Each approach is very similar in how the DP table is constructed and they all take similar (total) times to fill up the table. However, they differ in how the final output is extracted from the table. The approaches are as follows.

1. We could either compute all the Fibonacci numbers, keep them in a table, then add up the even numbers until we see the value N.
2. We can only keep a table of the even Fibonacci numbers (either by computing those directly or by computing the Fibonacci numbers), then add up the table until we see the value N.
3. We can keep track of just the partial sums of even Fibonacci numbers. That is, the sum of even Fibonacci Numbers until i. Then we output the partial sum upto the value N.

Approach 1: Table full of Fibonacci numbers

- **Logic:** Use the recursive formula for the Fibonacci numbers to compute the whole series of numbers and store them in a table.
- **The Bellman Equation:** $f_k = f_{k-1} + f_{k-2}$. The base cases would be $f_1 = 1, f_2 = 2$.
- **Iterative algorithm:** Fill in $f_1 = 1, f_2 = 2$ and then loop over k and fill in $f_k = f_{k-1} + f_{k-2}$.
- **Number of subproblems:** We need to store until $N = f_k$. So the number of subproblems (size of the table) will be k. In terms of N, it is known that $k \approx \log N$.
- **Time per subproblem:** For each recursive call, we only need to lookup table values/make other $O(1)$.
- **Total time for memoization:** $O(k) = O(\log N)$.
- **Final output:** You loop through the table while the value in the table is less than N and add up all the *even* values.

Approach 2: Table full of Even Fibonacci numbers

- **Logic:** Use the recursive formula for the Fibonacci numbers (or Even Fibonacci numbers) to store only the even Fibonacci numbers in a table.
- **The Bellman Equation:** This stays the same $f_k = f_{k-1} + f_{k-2}$. The base cases would be $f_1 = 1, f_2 = 2$.[1]

[1] If you want to only compute the Even Fibonacci numbers, the recurrence for that would be $E_k = 4 \cdot E_{k-1} + E_{k-2}$. And it is easy to see that $E_k = F_{3k-1}$.

- **Iterative algorithm:** Fill in $f_1 = 1, f_2 = 2$ and then loop over k and add to the table if f_k is even. [2]
- **Number of subproblems:** $O(k)$ where $f_k = N$ irrespective of whether we use Fibonacci or Even Fibonacci recurrence. In terms of N, it is known that $k \approx \log N$.
- **Time per subproblem:** $O(1)$.
- **Total time for memoization:** $O(k) = O(\log N)$.
- **Final output:** You loop through the table while the value in the table is less than N and add up *all* the values. Earlier we only added up the even values, but now all values are even.

Approach 3: Table full of Sums of Even Fibonacci numbers

- **Logic:** We have been looping through a table to add up values until some point. This is wasteful in the sense that if you want to add up until 8 and until 34, you'll add up until 8 in both cases. We could directly store the sums until n and that would suffice. We again use the recursive formula for the Fibonacci numbers (or Even Fibonacci numbers) to compute the even Fibonacci numbers and store their sums in a table.
- **The Bellman Equation:** This stays the same $f_k = f_{k-1} + f_{k-2}$ or $E_k = 4 \cdot E_{k-1} + E_{k-2}$. The base cases would be $f_1 = 1, f_2 = 2$ or $E_1 = 1, E_2 = 8$.
- **Iterative algorithm:** This changes, the first value filled in table T will be $(2, 2)$ to indicate that the sum until 2 is 2; the next value would be $(8, 10)$ indicating that sum until 8 is 10. For later values we find the next even Fibonacci number E_k and add $(E_k, T[k-1] + E_k)$ to the table.
- **Number of subproblems:** This also doesn't change. $O(k)$ where $f_k = N$ irrespective of whether we use Fibonacci or Even Fibonacci recurrence. In terms of N, it is known that $k \approx \log N$.
- **Time per subproblem:** $O(1)$.
- **Total time for memoization:** $O(k) = O(\log N)$.
- **Final output:** Perform binary search (we can do this since the table is sorted by construction) and find the largest value $E_k \leq N$ in the table and output the corresponding sum. Note that if we perform a binary search, we only spend $O(\log k)$ time to find the sum as opposed to spending $O(k)$ to loop through linearly and search.

Clearly the approaches get more sophisticated, although even the first one will get through the test cases in the programming part. The most optimal way (asymptotically), would be the third approach and from a programming point of view, it would also help to use the recurrence for just the even Fibonacci numbers. ■

[2] Note that if we use the recurrence for even Fibonacci numbers, we would always add the new computed number into the table but we would do only a third of the work as $E_k = f_{3k-1}$. This doesn't affect anything asymptotically but it would help during programming.

Problem 2 (TriPaSu) 25+10

You are given a triangle of non-negative numbers with i numbers at row i. You want to find the maximum path sum starting at the top of the triangle and moving to any of the numbers on the bottom row. At each step you can go to any of the (up to) two neighboring numbers in the row below. See this link for an example: `https://projecteuler.net/problem=67`.[3]

Solution:

- **Logic:** If we know what the best possible sum is until every value in the second last row, then we can easily find the best sum on the last row. For each entry in the last row, choose the max of its two "parents", the two entries that can step from to reach the entry on the last row. To do this, we store the maximum sum possible upto each number in the triangle given.
- **The Bellman Equation:** Let the given array be A, let the numbers of rows be n. We store the best sums upto index (i, j) in the $OPT(i, j)$.

$$OPT(i,j) = \begin{cases} A[i][j] + max(OPT(i+1,j), OPT(i+1,j+1)) & \text{if } i < n \\ A[i][j] & \text{if } i = n \end{cases}$$

- **Iterative algorithm:** This sum is calculated row by row, from bottom to the top. We start at the bottom row, and reduce the first index value and calculate the sums for the row above using the sums already calculated.
- **Number of subproblems:** $O(n^2)$. This is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$.
- **Time per subproblem:** $O(1)$. This is just making a 2 recursive calls and 1 comparison and 1 addition.
- **Total runtime time:** $O(n^2)$
- **Final output:** $OPT(1, 1)$

■

Problem 3 (Coin Change) 30+15

Given a denomination of positive coins $c_1, c_2, \ldots, c_m$ and a value n as input how many ways can you make change for n. For example, with coins being $\{1, 2, 3\}$ the number of ways to get the value 4 is 4 as follows: $\{1, 1, 1, 1\}, \{1, 1, 2\}, \{2, 2\}, \{3, 1\}$. With coins being $\{2, 5, 3, 6\}$ the number of ways to get the value 10 is 5 as follows: $\{2, 2, 2, 2, 2\}, \{2, 2, 3, 3\}, \{2, 2, 6\}, \{2, 3, 5\}, \{5, 5\}$.

Solution:

- **Logic:** Let us arbitrarily arrange the coins as $c_1, c_2, \ldots, c_m$. Now for any number i we can say that the number of ways to get to i using the coins from c_1 to c_j will need to count the ways that use no c_j coin and will also need to count the ways to use at least one c_j coin. These are separate events but they also account for all possibilities. Thus, they become our subproblems. If c_j is not used, then we still need to make i but now we only use c_1 to c_{j-1}. On the other hand, if c_j is used at least once, then we only need to make $i - c_j$ and we still have all of c_1 to c_j to use (because c_j may be used more than once).

[3]The challenge in the contest is based on this but the triangle seems to be messed up on the Hackerrank challenge.

- **The Bellman Equation:** Let $OPT(i, j)$ denote the number of ways to make the value i using the coins from c_1 to c_j.

$$OPT(i,j) = \begin{cases} 0 & \text{if } i < 0 \text{ or } j = 0 \\ 1 & \text{if } i = 0 \\ OPT(i, j-1) + OPT(i - c_j, j) & \text{otherwise} \end{cases}$$

- **Iterative algorithm:** The iterative algorithm would build up the OPT table from bottom up. It would first fill in the values when $i = 0$ and when $j = 0$. From there, it would iterate over all available coins (upto j) and see how many ways can higher up i can be filled.
- **Number of subproblems:** The index i varies from 0 to n and the index j varies from 0 to m. So a total of $O(nm)$ problems need to be solved.
- **Time per subproblem:** Each recursive call is just an addition. So $O(1)$.
- **Total time for memoization:** $O(nm)$.
- **Final output:** $OPT(n, m)$.

■