Dynamic Programming

Oct 31.2015

Please pair with someone, collaborate and share unit-test cases. Each section lists points earned(100 means superb). For how to implement unit-tests see Appendix.

A. Slow and Fast Fibonacci – 25 points

Background: The Fibonacci series is defined as:

$$f(x)=\begin{cases} f(x-1)+f(x-2) & for\ x\geq 1 \\ 1 & for\ x=1 \\ 0 & for\ x=0 \end{cases}$$

The first 5 numbers in the series would be:

```
f(0)=0
f(1)=1
f(2)=1
f(3)=2
f(4)=3
```

a. Write a very simple recursive function and set of unit tests that calculates the n^{th} fibonacci number without using a lookup table. Try seeing how many seconds it takes to calculate f(40). (10 points)

Fibonacci.java

```
public class Fibonacci {
    public static long fibonacci(int n) {

    }
}
```

FibonacciTest.java

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class FibonacciTest {
    @Test
    public void testFibonacci() throws Exception {
    }
}
```

b. Now write a version that uses a lookup table to store the result once it's been calculated. Copy the unit tests used for fibonacci for fastFibonacci. (15 points)

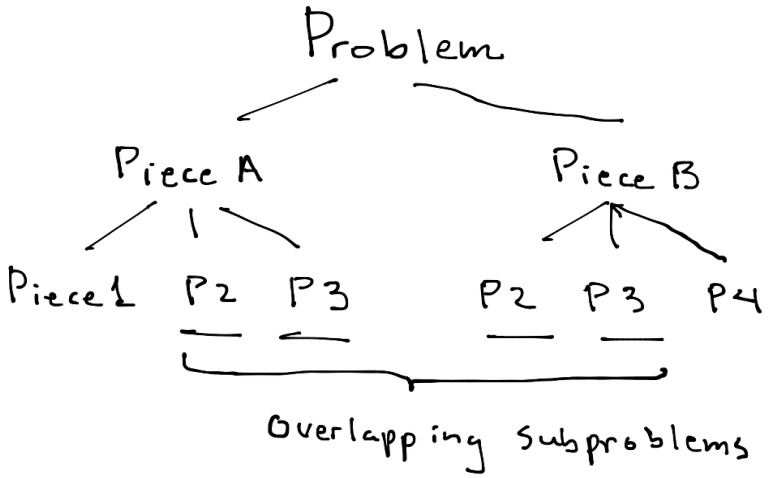
```
public class Fibonacci {
    ...
    static final int LOOKUP_TABLE_SIZE = 50;
    static long sLookupTable[] = new long[LOOKUP_TABLE_SIZE];
    public static long fastFibonacci(int n) {
        ...
        return answer;
    }
}
```

B. Dynamic Programming Background

See: https://en.wikipedia.org/wiki/Dynamic_programming

"In mathematics, management science, economics, computer science,and bioinformatics, dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems[1] and optimal substructure (described below). When applicable, the method takes far less time than other methods that don't take advantage of the subproblem overlap (like depth-first search)."

Overlapping Subproblems Illustration:

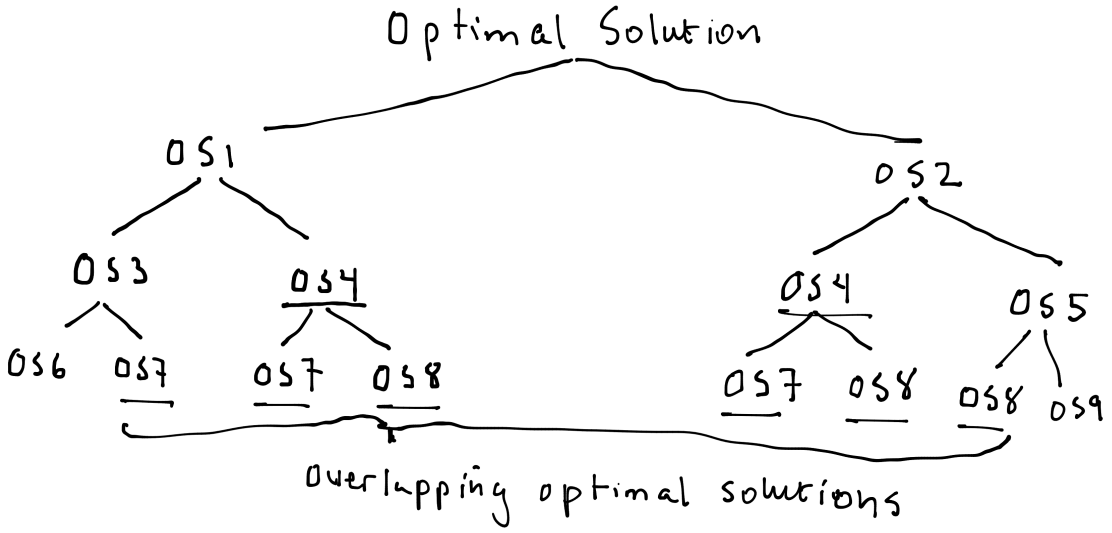


Since these problems overlap, as seen above, they are candidates for

being stored in a lookup table(aka as memo table).

Optimal Substructure(aka Principle of Optimality):

"A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solutions to its sub-problems"



So, the key to dynamic programming is storing overlapping optimal solutions in a lookup structure such as an array or dictionary.

Please watch the following video on YouTube (8 minutes):

> Principle of Optimality - Dynamic Programming
> by CSBreakdown
> https://www.youtube.com/watch?v=_zE5z-KZGRw

C. Understanding Dynamic Programming Tables – Filling out table 30pts

Let's examine the solution to the coin changing problem:

Given a set of coins with different denominations, find the total number of ways change could be given for a total amount of money.

So the function would be

```
int waysToMakeChange(int[] coins,int total) {
      …
      return numberOfWays;
}
```

Ex: *waysToMakeChange({1}, 5)* *would return 1*
waysToMakeChange({1,2},3) *would return 2*
waysToMakeChange({1,2},4) *would return 3*

Solving this using a recursive search would require exponential time. Dynamic programming can solve this problem with a dramatic improvement in efficiency with a complexity of only:

*O(numberOfCoins*total)*

Unfortunately, understanding the solution can be difficult at first. Here's the classical solution written out, and we'll go over it step by step:
http://algorithms.tutorialhorizon.com/dynamic-programming-coin-change-problem/

Please watch the video by Tushar Roy – called:

Coin Changing Number of ways to get total dynamic programming
https://www.youtube.com/watch?v=_fgjrs570YE

Create a solution matrix. (solution[coins+1][amount+1]).

Base Cases:

- if amount=0 then just return empty set to make the change, so 1 way to make the change.
- if no coins given, 0 ways to change the amount.

Rest of the cases:

- For every coin we have an option to include it in solution or exclude it.
- check if the coin value is less than or equal to the amount needed, if yes then we will find ways by including that coin and excluding that coin.

1. **Include the coin**: reduce the amount by coin value and use the sub problem solution (amount-v[i]).
2. **Exclude the coin**: solution for the same amount without considering that coin.

- If coin value is greater than the amount then we can't consider that coin, so solution will be without considering that coin.

Equation:

solution[coins+1][amount+1]

	=	0	if i=0
solution[i][j]	=	1	if j=0
	=	solution[i — 1][j] + solution[i][j — v[i —1]]	if(coin[i]<=j)
	=	solution[i —1][j];	if(coin[i]>j)

Example:

Amount = 5, coins[] = { 1,2,3 }

Coins \ Amount	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5

Classical dynamic programming problems are often expressed in terms of a generating function that creates a table. These functions can be pretty challenging to understand at first. It is very helpful to work through what the table means. Specifically what's represented by:

a. each column.
b. each row.
c. each cell – most important.

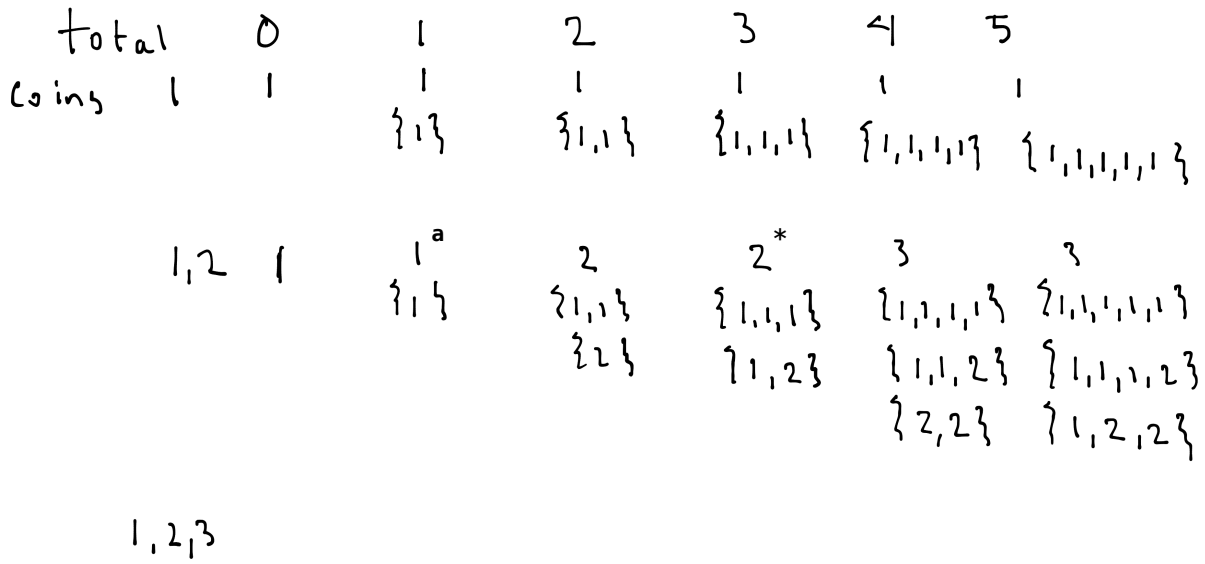
In our problem:

Each column = amount of money in cents.

Each row =the coin set, for example coins=1 means the set {1}
coins = 2 means the set {1,2}
coins = 3 means the set {1,2,3}

Each cell= the total number of ways to make change for amount given in the column, using the set of coins in the row.

a. Write a table by hand for the coins={1,2,3} and the total=5. In each cell specifically write each way to make change. See the example below for the first two rows.



1,2,3

* Notice how in the row for coins 1,2 and column for the amount 3, I took the value from row above at the same column, specifically {1,1,1}, and then I took the value at the same row, but 2 columns to the left(position [a]), specifically {1}, then I simply append a 2 coin to the end, so it became {1,2}.

This is how the generating function works...

solution[i][j] = solution from row above (which is solution[i-1][j])

and

solution from coin value columns to the left (which is solution[i][j – coinValue]) with coinValue appended to the list

C. Implement the Coin Changing Solution – Code 60 pts

a. Use the algorithm presented in http://algorithms.tutorialhorizon.com/dynamic-programming-coin-change-problem/ to implement your own solution for the coin changing problem using Dynamic programming.

D. If you get really stuck I've placed my IntelliJ project for reference up at:

a. Find the github repo at:
https://github.com/JonathanRitchey03/DynamicProgramming

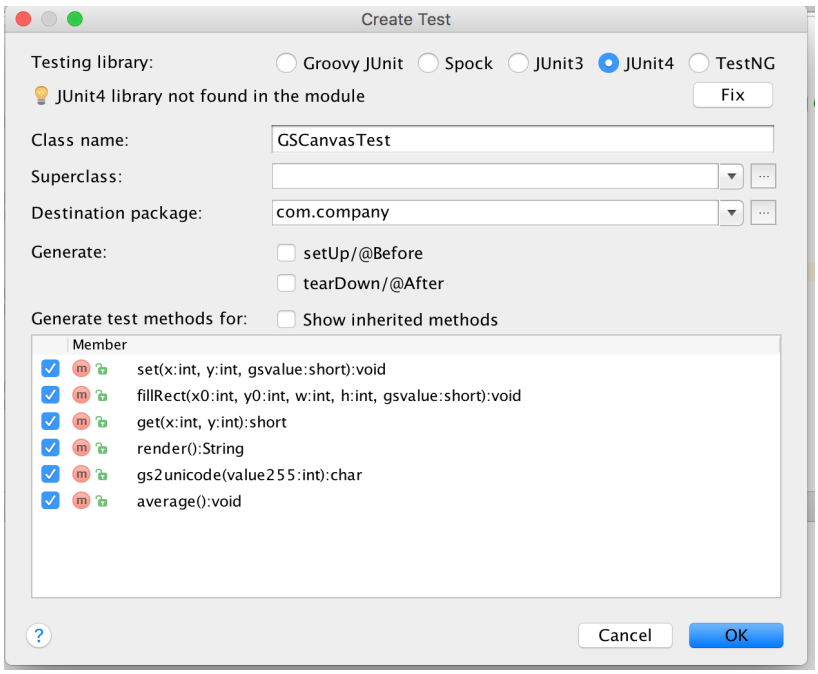
Please let me know if I can help...

E. Bonus Challenge 1 – Optimize the code so that it only needs to store two rows instead of coinArray.length rows – Code 10 pts

F. Bonus Challenge 2 – Add an option that actually prints out all the different ways to make change as part of the answer – Code 40 pts

Appendix – How to add Unit Tests to IntelliJ

1. In IntelliJ select the file you want to add tests to and press Command-Shift-T. (⌘ + shift + T)
2. Select "Create New Test".
3. On the pop that comes up, select JUnit 4. Now click on the "Fix" button so it includes the JUnit 4 package.
4. Click on all the methods. (see screenshot)



Here are some example unit tests:

```
@Test
public void testingCrunchifyAddition() {
assertEquals("Here is test for Addition Result: ", 30, addition(27, 3));
}

@Test
public void testingHelloWorld() {
assertEquals("Here is test for Hello World String: ", "Hello + World", helloWorld());
}
```

Appendix – How to add Unit Tests in PyCharm

https://confluence.jetbrains.com/display/PYH/Creating+and+running+a+Python+unit+test

1. In IntelliJ select the file you want to add tests to and press Command-Shift-T. (⌘ + shift + T)
2. Click on the check mark on all the method then press "OK". See screenshot:

Here are some example test cases:

```
def test_get_team_and_score_from_string(self):

    self.assertEqual(rank_teams.get_team_and_score_from_string("My Team 5"), ("My Team", '5'))
    self.assertEqual(rank_teams.get_team_and_score_from_string("My Team 5        "), ("My Team", '5'))

def test_update_rank_dict_for_team_by_points(self):

    rank_dict = {"teamA": 1}
    rank_teams.update_rank_dict_for_team_by_points(rank_dict, "teamA", 1)
    self.assertEquals(rank_dict, {"teamA": 2})
    rank_teams.update_rank_dict_for_team_by_points(rank_dict, "teamA", 0)
    self.assertEquals(rank_dict, {"teamA": 2})
    rank_teams.update_rank_dict_for_team_by_points(rank_dict, "teamB", 0)
    self.assertEquals(rank_dict, {"teamA": 2, "teamB": 0})
    rank_teams.update_rank_dict_for_team_by_points(rank_dict, "teamB", 3)
    self.assertEquals(rank_dict, {"teamA": 2, "teamB": 3})
```

Appendix – How to add Unit Tests in Visual C#

Check online. One link I found was:

http://www.codeproject.com/Articles/391465/Creating-Unit-tests-for-your-csharp-code

Appendix – How to add Unit Tests in JavaScript

If using WebStorm, JetBrains has support for this.

http://www.codeproject.com/Articles/391465/Creating-Unit-tests-for-your-csharp-code

Appendix – Dynamic Programming Resources

Tutorials

http://algorithms.tutorialhorizon.com/category/dynamic-programming/

TopCoder - https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/

CodeChef - https://www.codechef.com/wiki/tutorial-dynamic-programming

Videos

Tushar Roy - https://www.youtube.com/user/tusharroy2525

CSBreakdown - https://www.youtube.com/user/krispykarim
Goto Playlists – Select Dynamic Programming

Books

There are no easy books on Dynamic programming; however, I did find one interview book, with a collection of problems in it.

A Collection of Dynamic Programming Interview Questions Solved in C++ by Gulli

Problem Solutions

Geeks for Geeks - http://www.geeksforgeeks.org/tag/dynamic-programming/