Dynamic Programming 2

Nov 21.2015

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. We're going to be working on the 0-1 Knapsack and Edit-Distance challenges today.

A. Dynamic Programming Recap(Skip if you came to last session) – 10 points

What is Dynamic Programming?

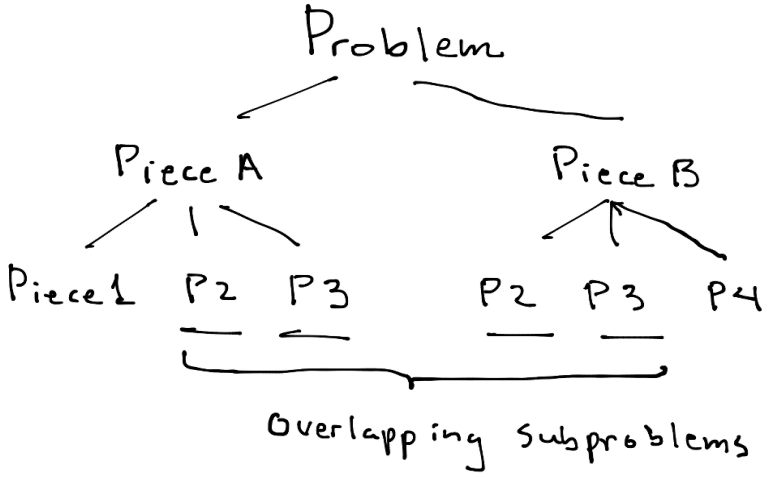
> Write down "1+1+1+1+1+1+1+1 =" on a sheet of paper*
> "What's that equal to?"
> *counting* "Eight!"
> *writes down another "1+" on the left*
> "What about that?"
> *quickly* "Nine!"
> "How'd you know it was nine so fast?"
> "You just added one more"
> "So you didn't need to recount because you remembered there were eight!
> *Dynamic Programming* is just a fancy way to say 'remembering stuff to save time later'" – *From Quora.com, how to explain dynamic programming to a child.*

> "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)."

Please read the very short overview here:

http://www.algorithmist.com/index.php/Dynamic_Programming

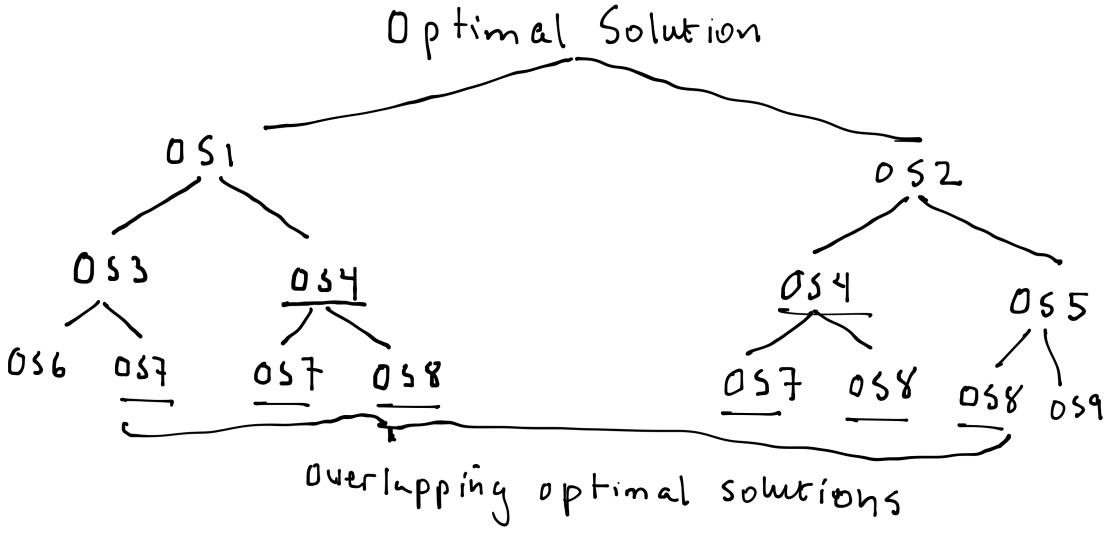
Overlapping Subproblems Illustration:



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(primarily a table).

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

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

Congratulations! Give yourself 10 points.

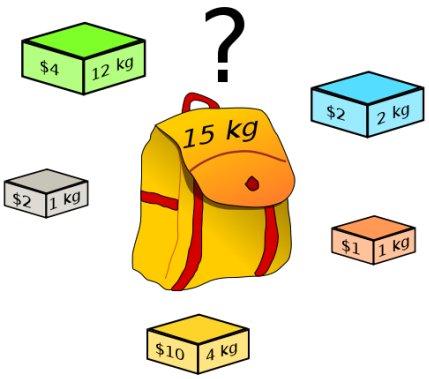
B. 0-1 Knapsack Problem – Function spec – 10 points

Here's the challenge statement:

> *Suppose you have a knapsack(which is an English word for backpack) which can carry a maximum weight of W kilograms.*
>
> *Suppose there are N items. Each item has a weight, w_i in kgs and value, v_i in dollars.*

i_i Item Number	w_i Weight	v_i Value
i_1	w_1 =1kg	v_1 =$1
i_2	w_2 =3kg	v_2 =$4
i_3	w_3 =4kg	v_3 =$5
i_4	w_4 =5kg	v_4 =$7

> *The "0-1" prefix in "0-1 Knapsack Problem" means that we can only pick a maximum of 1 of each item to put into the knapsack. So our choices are to either pick zero or one of each item from 1 to N. Hence the name "0-1 Knapsack".*
>
> *Restriction: The weights must be integer values, otherwise an easy DP solutions isn't possible.*



So, for the set of items $\{i_1,...,i_n\}$ with corresponding weights, $\{w_1, \ldots, w_n\}$, and values, $\{v_1,..,v_n\}$, what's the largest value that can be placed into a knapsack with a maximum weight capacity of W? Each item can only be picked once, and items can't be split into pieces.

1. Express this problem in terms of a function call in your programming language that returns the answer. i.e.

function signature:

? solveKnapsack(?, ?, ?)

*Note in C or C++ it may necessary to add extra parameters and a different function signature.

C. 0-1 Knapsack Problem – Table Construction – 20 points

We going to be using a "bottoms-up" dynamic programming approach to solve this problem. We are going to use a table to do this.

	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}	Weight capacity
Possible items									
$\underline{i_1}$					Each Cell = $ Max possible knapsack value				
$i_1, \underline{i_2}$					per weight capacity and per possible items.				
$i_1, i_2, \underline{i_3}$									
$i_1, i_2, i_3, \underline{i_4}$									

Underline=item under consideration

The x-axis is the weight capacity for different backpacks going all the way from a backpack that holds 0_{kg}(nothing) to a backpack that can hold 7_{kgs}. The y-axis represents possible items that could be added into the backpack, going from the option of holding only 1 instance of i_1 all the way up to optionally holding single instances of i_1,i_2,i_3 and/or i_4. So the final answer will be in the right-most, bottom-most cell that matches the knapsack with largest capacity and the item set with the most choices.

The values and weights we'll use are:

i_i Item Number	w_i Weight	v_i Value
i_1	w_1 =1kg	v_1 =$1
i_2	w_2 =3kg	v_2 =$4
i_3	w_3 =4kg	v_3 =$5
i_4	w_4 =5kg	v_4 =$7

How can we fill in the first column,0_{kg} ? Well, with a knapsack with 0_{kg} capacity, we can't fit any items in at all for any of the rows, representing the sets of possible items. So go ahead and put in $0 for each of rows in the 0_{kg} column.

	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}	Weight capacity
Possible items									
$\underline{i_1}$	$0								
$i_1,\underline{i_2}$	$0								
$i_1,i_2,\underline{i_3}$	$0								
$i_1,i_2,i_3,\underline{i_4}$	$0								

Underline=item under consideration

Now, let's change the table a bit so we can see the item values and weights on the y-axis.

Weight capacity ->	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}
Possible items								
item weight,value								
$\underline{i_1}$ = 1_{kg},$1	$0							
i_1, $\underline{i_2}$ = 3_{kg},$4	$0							
i_1,i_2, $\underline{i_3}$ = 4_{kg},$5	$0							
i_1,i_2,i_3, $\underline{i_4}$ = 5_{kg},$7	$0							

Underline=item under consideration

*So for the first row, we can **only** have a single instance of item 1. Remember that only one of each item can be picked. So no matter how big the backpack is, we can only put in one instance of item 1. So the first row will look like this:*

	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}
$\underline{i_1} = 1_{kg}, \$1$	$0	$1	$1	$1	$1	$1	$1	$1

Underline=item under consideration

Note that each cell is the optimal solution for a backpack of the weight capacity in the the x axis and the possible item set in the y-axis. So that means each cell contains the maximum possible value for the item set for its current position. We are going to be re-using the optimal solutions found for smaller knapsacks and smaller possible sets to find the optimal solutions for bigger knapsacks and bigger possible sets! These solutions, of course, are all going to be stored in our two dimensional table above.

Now, let's work on the second row.

Weight capacity ->	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}
Possible items								
item weight,value								
$\underline{i_1} = 1_{kg}, \$1$	$0	$1	$1	$1	$1	$1	$1	$1
i_1, $\underline{i_2} = 3_{kg}, \$4$	$0	$1	$1	?				

Underline=item under consideration

Ok, so we've reached a point where's it's not clear what to do. Here's where we define a rule. So, the idea is this:

If the backpack's weight capacity is more than the item under consideration's weight, then there are two choices. First, (a) we can choose not to put in the new item under consideration. Or, (b) we can make space for the new item under consideration by taking out other items and putting it the new item.

So more formally:

```
let Wj = Weight capacity at column j.
let wi = weight for item at row i.

let   spaceAvailableForNewItem = Wj ≥ wi
if    spaceAvailableForNewItem
      a.    don't put in new item
                        or
      b.    step b1 :make knapsack lighter +
            step b2 :new item's value.

else
      value = Table(row above, current column)
```

Let's figure out how to use our table to find out what the value would be if we chose option a. Well, we can see from our table, that the row above at the exact same column, tells us exactly what the maximum value is without the new item. So:

```
if spaceAvailableForNewItem → a. don't put in new item
                              Cell=Table(row above,column)
```

For step b_1, it's more complicated. We need to make knapsack lighter for the new item by taking out its weight. But actually our table can do this. How??? We can tell by looking at a backpack in our table with a lighter weight capacity. How much lighter? Exactly the new item's weight lighter. So for step b_1, we can find the optimal solution for that lighter weight by looking at the row above, and go back the new item's weight, w_i , columns. Hence, b_1 = Table(row above, column - w_i).

For step b_2, we simply represent that as the value of the new item. So, $b_2 = v_i$.

So now we can say:

```
let (i,j) = (current row, current column)
let spaceAvailableForNewItem = Wj ≥ wi
if spaceAvailableForNewItem →
     Cell = maximum[  Table(row above, column)
                           or
                           Table(row above, column – wi) +  vi ]
                           (step b1)              +     (step b2)
else
     Cell = Table(row above, column)
```

So using this principle, we can fill in the rest of row 2.

	Weight capacity ->	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}
Possible items	item weight,value								
	$\underline{i_1}$ = 1_{kg},$1	$0	$1	$1	$1	$1	$1	$1	$1
i_1,	$\underline{i_2}$ = 3_{kg},$4	$0	$1	$1	$4	$5	$5	$5	$5

Underline=item under consideration

a. Fill out the rest of the table here:

	Weight capacity ->	0_{kg}	1_{kg}	2_{kg}	3_{kg}	4_{kg}	5_{kg}	6_{kg}	7_{kg}
Possible items	item weight,value								
	$\underline{i_1}$ = 1_{kg},$1	$0	$1	$1	$1	$1	$1	$1	$1
i_1,	$\underline{i_2}$ = 3_{kg},$4	$0	$1	$1	$4	$5	$5	$5	$5
i_1,i_2,	$\underline{i_3}$ = 4_{kg},$5	$0							
i_1,i_2,i_3,	$\underline{i_4}$ = 5_{kg},$7	$0							

Underline=item under consideration

If you get stuck, please watch Tushar Roy's video for a solution to this problem. It's at:

https://www.youtube.com/watch?v=8LusJS5-AGo

D. 0-1 Knapsack Problem – Recurrence 10 , Code 20, Tests 15 points

Find the recurrence formula for this dynamic programming problem and make sure you understand it.

Implement a solution in code with your language for this problem. Start by initializing an array with N items rows, and W weight capacity+1 columns(because the first column will hold 0_{kg}). Then fill the first column with $0.

Next, go row by row, applying the rules you learned above.

After writing the code. Write a unit test that takes in the parameters for our exercise and makes sure the ending answer is 9. Add in a couple extra unit-tests for boundary conditions.

E. Knapsack Bonus – Return items selected - 30 points

Watch the end of Tushar Roy's video and this video to see how to calculate the actual items picked:

https://www.youtube.com/watch?v=qOUsP4eoYls

F. Minimum Edit Distance – Function Spec – 10 points

The edit distance problem is defined as:

> *Given a string S, say "abcd", and a string T, say "abf". What are the minimum number of edits(replacing, deleting, or adding a character) necessary to get from string S to T?*
>
> *So, for this example it would take 2 edits.*
> *"abcd" $\rightarrow_{\text{delete c}}$ "abd" $\rightarrow_{\text{replace d with f}}$ "abf"*

Write a function signature for your edit distance implementation that

takes in a source string and target string and spits out the minimum edit distance necessary to transform the source into the target string.

F. Minimum Edit Distance – Table Creation – 30 points

We're going to be using a 2D table again to construct our answer. Let our Source = "abcdef" and Target = "azced".

	∅	a	ab	abc	abcd	abcde	abcdef ← Source string
Target string							
∅							
a							
az							
azc							
azce							
azced							

∅ means Empty String

The x-axis holds longer and longer versions of our source string. The y-axis holds longer and longer versions of our target string. Any one table cell, represents the minimum number of edits to get from the string in the column to the string in the row.

We can easily deduce what to put in our first row and column, which is just the minimum number of edits to get from the empty string to the source string or the target strings respectively.

	∅	a	ab	abc	abcd	abcde	abcdef ← Source string
Target string							
∅	0	1	2	3	4	5	6
a	1						
az	2						
azc	3						
azce	4						
azced	5						

∅ means Empty String

Now for our second row, from: *Source → Target* *Edits*

a → a	*0*
ab → $_{\text{del b}}$ *a*	*1*
abc → $_{\text{del bc}}$ *a*	*2*
abcd → $_{\text{del bcd}}$ *a*	*3*
abcde → $_{\text{del bcde}}$ *a*	*4*

Let's intuitively continue filling out another cell.

Source → Target *Edits*

ab → $_{\text{b=z}}$ *az* *1*

Target string	∅	a	ab	abc	abcd	abcde	abcdef ← Source string
∅	0	1	2	3	4	5	6
a	1	0	1[d]	2[u]	3	4	5
az	2	1	1[l]	?			
azc	3						
azce	4						
azced	5						

∅ means Empty String

Now what about abc → az … Now it's getting more complicated. Fortunately there's a deduction we can make. Specifically that going from abc → az can go through 3 previous possible paths:

1. Diagonal cell, symbol [d]: *abc →* $_{\text{refactor}}$*(ab)c*
(ab)c → $_{\text{ab→a}}$*(a)c* *1 edit*
(a)c → $_{\text{c=z}}$*(a)z* *1 edit* *total edits = 2*

2. Row above, symbol [u]: *abc →* $_{\text{refactor}}$*(abc)*
(abc) → $_{\text{abc→a}}$*(a)c* *2 edits*
(a)c → $_{\text{c=z}}$*(a)z* *1 edit* *total edits = 3*

3. Row to left, symbol [l]: *abc →* $_{\text{refactor}}$*(ab)c*
(ab)c → $_{\text{ab→az}}$*(az)c* *1 edit*
(az)c → $_{\text{del c}}$ *(az)* *1 edit* *total edits = 2*

To calculate the new cell value simply take the minimum of options 1, 2, or 3.

These options can be calculated mentally by simply checking whether the last character is different, and if so, take the minimum of the 3 adjacent cells and add one. If the last character is the same, then just take the value from the upper-left diagonal.

a. Complete the rest of the chart below using this technique.

	∅	a	ab	abc	abcd	abcde	abcdef ← Source string
Target string							
∅	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
az	2	1	1				
azc	3						
azce	4						
azced	5						

∅ means Empty String

After filling out the table watch Tushar Roy's video at:

https://www.youtube.com/watch?v=We3YDTzNXEk

The formula can be specified as:

```
let (i,j) = (row,column)
let isLastCharacterSame = source[i] == target[j]

if isLastCharacterSame

      T[i][j] = T[i-1][j-1] // upper-left diagonal value

else

      T[i][j] = min(      T[i-1][j],   // row above
                          T[i-1][j-1], // upper-left diagonal
```

```
T[i]  [j-1]) // column to left
```

G. Minimum Edit Distance – Implement the code 30 points

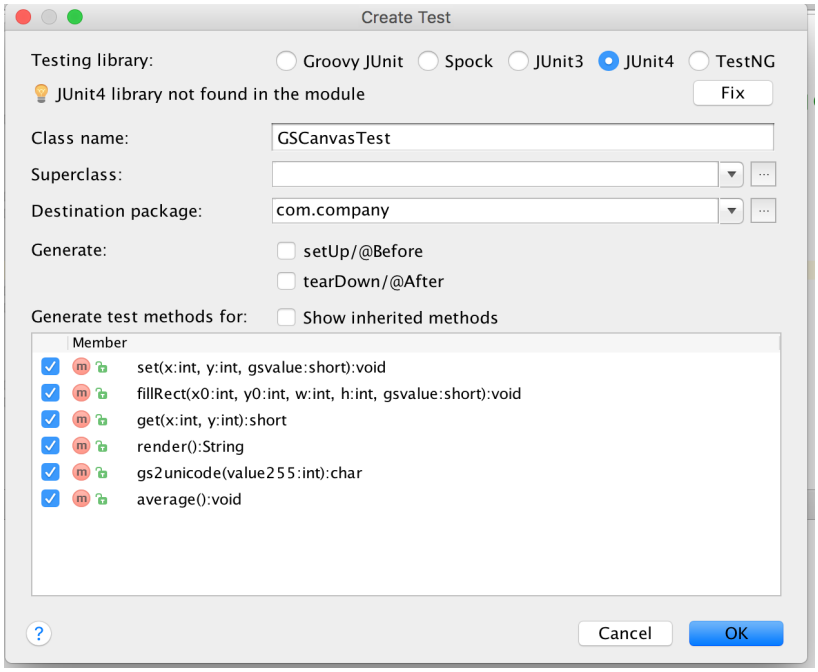
Here's the pseudo-code link:
http://www.algorithmist.com/index.php/Edit_Distance

H. Minimum Edit Distance Bonus – Specify the edits 30 points

Research this.

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/