Dynamic Programming

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

Dynamic Programming is a technique that you can use to construct your own algorithms to solve problems. The idea is to set up a recursive solution and then add *memoization*, which means storing each solution you've calculated along the way and reusing them any time the same recursive call arises.

You might think "why does *dynamic* programming mean *recursion with memory*"? It's a good question. It turns out the term was invented to sound cool so that the developer, Richard Bellman, could get more research funding, and doesn't really mean anything! (Who wouldn't want to fund *dynamic programming*? Sounds coooooool.)

In any case, a dynamic programming algorithm needs to be set up this way:

- The problem is broken down into a series of *subproblems*, which are often smaller versions of the main problem.
- The *subproblems* are related recursively (that is, you determine a way to calculate each subproblem in terms of other smaller subproblems), with an ordering that will ultimately end up on a *base case*.
- The solution to each subproblem is stored in a *memo* once it is calculated, usually a dictionary, so that if that subproblem is encountered again in a future recursive call, the solution can be recalled rather than recalculated.

When conceptualized this way, we can calculate the big-Oh of a dynamic programming algorithm by simply doing O(number of subproblems) * O(work per subproblem).

Example: Fibonacci Numbers

It's probably well known to you by now that an iterative Fibonacci implementation is much faster than a recursive one. What if we want a recursive one anyway?

```python
# Not memoized
def fib(x):
    if x < 2:
        return 1
    else:
        return fib(x-2) + fib(x-1)

print(fib(35))
```

```python
# Memoized
def fib(x, fibs):
    if x not in fibs:
        fibs[x] = fib(x-2, fibs) + fib(x-1, fibs)
    return fibs[x]

fibs = {0:1, 1:1}
print(fib(35, fibs))
```

If you think about it for a moment, you'll probably be able to convince yourself that the solution on the right is just as fast as the iterative implementation. (Or, at least, has the same big-Oh). Each number is only calculated once; if it has been seen before, it's simply accessed. So our big-Oh here would be O(n) * O(1), since there are n subproblems (n total Fibonacci numbers) and the work to compute each one is a simple addition.

Set Up Your Code

For this problem set, no matter how far you get, you'll only turn in one file. (You can resubmit as you finish each part!) You'll get two command line inputs – a number (don't forget to convert it to an int) and the name of a .txt file. The number will tell you which problem you should run. The text file will contain your test cases for that problem. Solve each test case and print the output.

If your code doesn't have a solution to the problem identified by the command line number, just print "unsolved".

Level 1: GREEN

So: how do you use dynamic programming to solve a problem? The key is coming up with the right subproblems – as few as you can manage, related recursively, leading to a base case. Let's do a couple of examples.

1) **Candy Prices.** The candy store Mad Maggie's is run by a well-known irrational shopkeeper. Whereas most shop keepers would say something like "5 pieces for $1, 12 pieces for $2, 40 pieces for $5" – a sliding scale where larger payments get more value – Maggie doesn't do it that way. She prices things pretty randomly, and her prices change every day. For example, one day she has this price structure:

   $1 – 1 piece
   $2 – 10 pieces
   $3 – 13 pieces
   $4 – 18 pieces
   $5 – 20 pieces
   $6 – 31 pieces
   $7 – 32 pieces

   The question is – what's the largest amount of candy you can get for $7?

   You might try looking at each ratio and starting with whatever option gives the greatest ratio. In this case, that's 31 pieces for $6; slightly over 5 pieces per dollar. So, you get one of those. Now you have $1 left, and get 1 additional piece, for a total of 32. That's the same as you could get by just buying the $7 option. Can you do better?

   It turns out yes, though this is not obvious. You can get more than 32 candies for $7. Do you see how?

   If you think of this like a standard BFS, you can see that you'll be searching for a long time. But let's approach this as a dynamic programmer. What are the subproblems? How are they recursively related?

   You should be able to find an algorithm for this that is $O(n^2)$, with n subproblems and O(n) work for each one. Make sure you check it with someone before you code it!

   Your test cases are in a file called "01_candy_prices.txt". Each row represents Maggie's prices on a certain day. You can assume that the first number is the number of candies for $1, the next number is $2, etc. For the sake of simplicity, assume that the largest dollar value represented is the same as the number of dollars you have with you on that day – eg, in the example above, the numbers go up to $7 and we also walked into the store with $7. (Alternately, you can imagine that on every day Maggie has infinitely many payment options, but the only ones listed are the ones you could afford!) Your job is to read in this file and return your maximum number of candies each day.

   To check yourself, the first three answers should be 594 candies, 1199 candies, and 1799 candies. All of the tests should run in less than 15 seconds total.

2) **Candy Jar Game.** Mad Maggie has another game she likes to play. You pay a flat fee (so, we aren't worried about price here) and you're given a row of jars. Each jar is labelled with a number, which could be either positive or negative. You can claim any jars you like and leave any jars you like on the shelf, and you get the total amount of candy you've claimed, ignoring the values left behind. So far, this is an easy question – just take all the positive numbers and leave the negatives. But there's a twist. If you like, you can claim any jar and a jar next to it as a pair together; each time you do this, you get the *product* of the two numbers on the candy jars.

   For example, let's say this was the row of jars today:

   1 1 9 9 2 -5 -5

   You could individually take all the positive jars for a total of 22 candies. Or, you could claim the 9s together as a pair, for 81, and the -5s together as a pair, for *positive* 25. Then, take the 1s and 2s individually, for a total of 110.

   Note that when a jar is claimed, it leaves an empty space. If this was Maggie's Candy Jar Game for the day:

   10 5 10

   …then you couldn't claim the 5, then claim the 10s as a pair and multiply them. If you claim the 5, the game looks like this:

   10 __ 10

   …and you would have to claim the 10s individually. So, your best outcome this time is 60 – claim a 10 with the 5 to get 50, then add the other 10.

   As a hint about subproblems this time, note that, one way or another, you'll need to decide what to do with the first pin. What are your choices? What's left after you make them?

   You should be able to get an algorithm for this problem in O(n) time, with n subproblems and O(1) work for each.

   Your test cases are in a file called "02_candy_jar_game.txt". The formatting should be self-explanatory. For each test case, print the best outcome you can get from that game.

   To check yourself, the first three outcomes should be 6561, 58993, and 196765 (that's a lot of candy). You should be able to solve all test cases in less than 15 seconds total.

Level 2: BLUE

Finding good subproblems can be challenging and there's no definite rule (like "there must always be n subproblems!" – definitely not true.) On some questions, n-squared subproblems is the best you can do. Sometimes your subproblems aren't even the same question as the original question – they may be more specific, or more general. The key is that once you've answered all your subproblems, you can answer the original question (even if that takes a bit more work).

Here's one problem that needs more than n subproblems:

Largest Common Subsequence. Given two sequences of integers, find the length of the longest common subsequence.

Note that a subsequence is any sequence of integers found left-to-right within the original sequence; they don't need to be adjacent. In other words, [1, 4] is a subsequence of [0, 1, 2, 3, 4, 5] but [4, 1] isn't.

So, as an example, if we were given the two lists [1, 2, 8, 4, 3] and [6, 2, 8, 7, 4], the largest common subsequence has a length of 3. Specifically, it would be [2, 8, 4].

What are our subproblems here? Naturally, we'd want the largest common subsequence of something smaller; that feels recursive. We could make a smaller problem by, say, starting at an index that isn't 0 on either the first subsequence or the second. Or both. We restrict the values to the values from that index to the end, making a smaller sequence to look through. You can probably start to see some recursive relations here.

But can we cut it down any more than that? Nope. There's no way to be especially clever here – we'll need to have the largest common sequence starting at any index of the first sequence *and* any index of the second. In other words, we have n*m subproblems here (if n is the length of the first sequence and m the length of the second).

The good news is that the work per subproblem is O(1). Given starting indices, we only need answer a few questions:

- Is either starting index outside the bounds of its sequence? If so, the LCS would obviously be of length 0.
- Is the number at each starting index the same? If so, we'll want to include that number in our LCS. So, our LCS length is 1 + the LCS length starting at the next index in each substring.
- Is the number at each starting index different? Well, then, we can't claim that match as part of our LCS. But each number might match itself later on in the other sequence. So we'll have to compute two possibilities (increase each index by 1 but not the other) and take the max.

This code implements the LCS length algorithm outlined above:

```
def lcs(s1, s2, i1, i2, memo):
    if (i1, i2) in memo:
        return memo[(i1, i2)]
    if i1 == len(s1) or i2 == len(s2):
        memo[(i1, i2)] = 0
    elif s1[i1] == s2[i2]:
        memo[(i1, i2)] = 1 + lcs(s1, s2, i1+1, i2+1, memo)
    else:
        memo[(i1, i2)] = max(lcs(s1, s2, i1+1, i2, memo), lcs(s1, s2, i1, i2+1, memo))
    return memo[(i1, i2)]
```

Talk through this with another student or myself to make sure that it makes sense!

Then begin your problems on the next page.

3) **Largest Common Subsequence Part 2.** The algorithm above just returns the *length* of the largest common subsequence. Copy it and modify it so it instead returns *the entire subsequence*, as a list.

   Your test cases are in a file called "03_lcs.txt". Each line contains two sequences, so we can't just put spaces between everything. Within each sequence, the values are separated by "," characters. There is a single space between the two sequences. For each pair, print a list of the largest common subsequence.

   To check yourself, the first three should be:
   [-49, -3, 14, 67, -50, -76, 12, 66]
   [-30, 106, -156, -95, -195, -82, 100, 97, -173, -118, 112, 7, -105, -8, 34, -163, -91, 10, 111, 32, 68]
   [45, -54, 16, 18, -236, 135, -42, 122, -237, 252, 41, 174, 201, 221, 76, 287, -181, 145, -59, 167, 270]

   You should be able to solve all test cases in less than 30 seconds total.

4) **Largest Increasing Subsequence.** You'll be given a sequence of integers. You should return the longest subsequence that is strictly increasing (that is, where each number in the subsequence is strictly larger than the previous).

   This is a straightforward question that is surprisingly tricky to solve! I'm tempted to put some hints into this document, but I think I'd rather give you a chance to think about it. Please let me know if you'd like a hint!

   You should be able to find a solution with O(n) subproblems, each of which takes O(n) to solve.

   Your test cases are in a file called "04_lis.txt". Each line is a single sequence, separated by spaces. You need to print out the largest increasing subsequence. Where there are multiple subsequences of the largest length, you should return the leftmost one (ie, the one that starts on the earliest number). Your code will probably do this automatically, but it's worth running a test or two to double check.

   To check yourself, the first three answers should be:
   [-186, -181, -176, -169, -166, -153, -147, -136, -133, -123, -115, -96, -85, -84, -59, -51, -39, -35, 1, 5, 7, 30, 62, 143, 187]

   [-367, -267, -206, -200, -191, -173, -172, -117, -110, -87, -82, -53, 11, 42, 83, 91, 102, 118, 146, 163, 172, 174, 193, 212, 235, 295, 325, 341, 347, 349, 359, 370, 379, 387, 392]

   [-595, -519, -428, -427, -408, -341, -317, -249, -225, -224, -215, -168, -161, -146, -142, -41, 27, 31, 102, 130, 154, 166, 194, 233, 267, 276, 299, 329, 336, 343, 450, 486, 511, 514, 517, 523, 536, 546, 561, 580, 595]

   You should be able to solve all test cases in less than 15 seconds total.

Level 3: RED

Harder questions! Remember that at any point you can turn in the work you've done so far and get credit for it; you're allowed to resubmit your file as many times as you like.

5) **Well-Spaced Largest Decreasing Subsequence.** This is similar to the previous question, but you'll need to twist your thinking a bit. For one thing, this time I'd like the largest decreasing subsequence, not increasing. There's also a new restriction on your subsequence. In addition to the sequence, you'll get an integer, a group size. Imagine taking your original sequence and splitting it into groups of that size.

   For example, if you received [10, 107, 106, 4, 5, 207, 3, 2, 7, 105, 6, 104, 197, 100] and 4, your groups would look like this:

   10, 107, 106, 4
   5, 207, 3, 2
   7, 105, 6, 104
   197, 100

   I'd like you to return the largest decreasing subsequence where you choose exactly **one number** from **each group** for several **consecutive groups**. That is, you can't take more than one number from the same group, but also, your decreasing subsequence can't *skip* a group either.

   Without any restrictions, the largest decreasing subsequence from the above sequence would be [107, 106, 105, 104, 100]. With the restriction, the best we can do is [207, 105, 100]. Note that [107, 105, 100] would be an incorrect answer, since it skips the 5, 207, 3, 2 group.

   Your test cases are in a file called "05_well_spaced_lds.txt". On each line, there is a single number in parentheses at the beginning; this is your group size. The rest of the numbers on the line form your sequence.

   To check yourself, the first three should be:
   [120, 99, 47, 44, 7, -36, -71, -80, -85]
   [335, 306, 288, 159, 151, 131, 117, 102, -2, -126, -163, -277, -312, -360]
   [560, 554, 506, 481, 453, 384, 341, 340, 298, 251, 206, 197, 104, -53, -81, -89, -164, -309, -497, -595]

   You should be able to solve all test cases in less than 15 seconds total.

6) **Best Parenthesization.** You'll get a sequence of integers. Your job is to leave that sequence in order, but to place parentheses and + or * symbols so that you get the largest possible result from executing the resulting math problem.

For example, given this:

1 2 3

Your best result would be this:

(1 + 2) * 3 = 9

Or, given this:

-5 -5 1 -5 -5

Your best result would be this:

-5 * -5 * (1 + -5 * -5) = 650

A particularly interesting example is this:

-4 -3 -5

Because your best would be this:

(-4 + -3) * -5 = 35

…and this has interesting consequences for your choice of subproblems. Note that your subproblem can't just be "the largest possible result from parenthesizing any substring", since -7 is NOT the largest number you can get from -4 and -3. So: what can you do here to address this situation?

This is quite a challenging problem; feel free to ask for a hint.

Your test cases are in a file called "06_parenthesization.txt". Each line is a sequence of numbers separated by spaces. Your job is to output the largest result from parenthesizing each line, as well as the actual parenthesization that achieves that result. There are multiple different rules you might use to write the parentheses into your final string; any valid rule is fine.

To check yourself, the first three should be:
-3 * 2 * -2 = 12
(1 + 4) * -3 * -3 = 45
(1 + 4) * (0 + 1 + 5) = 30

(And once again, if you have additional parentheses that I don't, it's totally ok as long as the parenthesization is valid.)

You should be able to solve all test cases in less than 15 seconds total.

Level 4: BLACK

Harder still! Very tricky twists on a couple of the earlier problems.

7) **Harder Parenthesization.** Same as the above problem, but decimal values are allowed and so are the operations of subtraction and division.

   Your test cases are in "07_harder_parenthesization.txt". For your testing, the first three results are:
   -2.18 * -2.27 * 1.87 = 9.253882000000003
   -2.18 / ((-0.16 / -3.61) * -0.15)) = 327.90833333333336
   (-1.26 - 4.35) * 4.75 * 3.89 * -2.43 = 251.89082324999998

   If you're identical except for the last digit, it's a floating point rounding error and you shouldn't worry about it.

   You should be able to solve all test cases in less than 15 seconds total.

8) **Harder Candy Jar Game.** Same game as #2, except that when you remove a jar, it does *not* leave a space. The jars on either side can be claimed as a pair, now. That is, if you have this game:

   10 5 10

   …the ideal solution is to take the 5, leaving:

   10 10

   …then take the 10s as a pair, multiplying to 100, for a final total of 105.

   For this one, I'd like you to print not only the final result but also every state along the way; show the complete Candy Jar Game being played, start to finish, and then the final score.

   Your test cases are in "08_harder_candy_jar_game.txt", formatted the same as question 2. For your testing, the first three results are 205, 1489, and 5045. You'll need to print out the states along the way; I am not including that here for you to check, but if you hit the right final results, it's correct.

   The runtime might be a little longer on this one, but you're still aiming for less than a minute total.

Specification

Submit a single Python script to the link on the course website. I will run all 8 problems, and you'll get credit based on which ones are correct.

As a reminder, you'll receive two command line inputs – a number (don't forget to convert it to an int) and the name of a .txt file. The number will tell you which problem you should run. The text file will contain your test cases for that problem. Solve each test case and print the output. If you haven't implemented a problem, just print "unsolved".

Make sure that **you have no extra print statements.** There should be exactly 1 line of output per run (except for #8). There are a lot of test cases on this assignment, so extra print statements make it difficult to grade. Be sure you don't have any!

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- For GREEN credit: questions 1-2 are correct.
- For BLUE credit: questions 1-4 are correct.
- For RED credit: questions 1-6 are correct.
- For BLACK credit: questions 1-8 are correct.
- Your runtime is within the time limits given on this sheet.
- **You have no extra print statements.**