Lab 06: Dynamic Programming

Contents

Objectives

By the end of this lab you will understand:

- How greedy algorithms work
- How dynamic programming works
- How to solve problems recursively

By the end of this lab you will be able to:

- Solve different problems in different ways
- Integrate optimizations into your code

1 Making Change

In mathematics and computer science (and maybe life in general!), it is crucial to solve big problems by breaking them down into smaller problems. Once we solve the smaller problems, if we then put the smaller solutions together correctly we can solve the big problem. We have already seen examples of this in the first half of the term, in terms of combining different functions to solve big tasks, but as we start tackling bigger and bigger problems in the second half of the term this way of decomposing problems and then combining their solutions will become absolutely essential.

Today's lab will be devoted mostly to a single task: when faced with an amount to give back in change to someone, how do we do so using the smallest possible number of coins?[1] This is an ideal problem to demonstrate an important technique called *dynamic programming* (DP), which allows us to compute the optimal solutions to problems that it would otherwise take a very long time to try to solve. In other words, it really relies on this ability to break the problem down into smaller problems and then reconstruct the big solution based on the smaller ones.

Task 1. Come over and sit where you have a view of the whiteboard. We're going to do things a bit differently today!

[1] `https://en.wikipedia.org/wiki/Change-making_problem`

2 Greedy Algorithms

A *greedy* algorithm[2] acts, as the name suggest, to solve a given problem in the "greediest" manner possible. What does it mean for an algorithm to be greedy? Essentially, at every step along the way, it attempts to solve the problem as quickly as possible. So, in making change, it will start with the biggest possible coin less than the amount needed, and subtract that. This is probably the easiest way to break the problem down into smaller problems, where the smaller problem can be thought of as the amount of change that is left after taking away the biggest coin.

For example, a greedy algorithm to make change for 17p would first identify the coin with the highest value that is less than or equal to the amount of change; in this case it would be the 10p coin. There would then be 7p left in terms of change to make, at which point the algorithm would pick the 5p coin. There would then be 2p left to make, at which point the algorithm would pick the 2p coin and be done. The final output would be the number of coins needed to make change (3) and the list of coins (10p, 5p, and 2p).

Task 2. Write a function `greedy_change` in a file `make_change.py` that, given a list of coin denominations `denoms` and an amount `amt` that we need to give in change, outputs a list of coin denominations representing the change that should be given. Your code should act as greedily as possible by always picking the biggest possible value first, and should iterate until you have all the coins needed to make change. You are welcome to assume `denoms` is already sorted from high to low (so the highest denomination is first and the lowest is last).

Test out this function by running it on "normal" coin denominations, like we have in the UK, and any amounts you want. For example, try making change for 14p, 28p, 83p, and 89p. Now, run it using some "weird" denominations, like if the 20p coin were replaced with a 7p coin, and test it again on the same amounts.

3 Dynamic Programming

Greedy algorithms have the advantage that they run quickly and are relatively easy to write, but they are not guaranteed to produce the optimal solution, and in some cases might be quite far off. To make change for 14p using the "weird" denomination of a 7p coin, for example, the minimum number of coins to give out is two, as you can give back two 7p coins. Because the greedy algorithm will immediately go for the biggest denomination, however, it gives out three coins instead (a 10p coin and two 2p coins).

In contrast, dynamic programming (DP)[3] is designed to guarantee the optimal solution every time. It will feel very challenging when you first start working with it, so please don't panic when this happens! Instead, spend a lot more time than you might otherwise do working through examples by hand, and looking around on the Internet for practice problems and solutions.

Briefly, DP works as follows: first, the problem is broken down into smaller problems according to a *recurrence relation* that specifies how each larger problem depends on a smaller one. In making change, for example, we can write $J(D, x) = \min_{d \in D}\{J(D \setminus d, x), 1 + J(D, x - d)\}$, where $J(D, x)$ is the minimum number of coins needed to make change for an amount x using the set of coins D.

In other words, at each step we consider each coin denomination $d \in D$ and imagine either not using that value to give change (which explains the $J(D \setminus d, x)$ term, representing the case where we ignore the coin d so still need to make change for the full amount x) or using it to give change (which explains the $1 + J(D, x - d)$ term, representing the case where we use d and now need to make change only for the amount $x - d$). These are both subproblems, and the relation expresses how their solutions can be combined to give a solution to the bigger problem. We are then searching for the values of d that minimize this overall. The *base case* or *boundary condition* is when we want to make change for the amount 0 or have no coins, meaning $J(D, 0) = 0$ and $J(\emptyset, x) = \infty$ (where as a reminder $\emptyset$ is the empty set).

This recurrence relation specifies how we can solve the bigger problem by piecing together solution to the smaller problems. We start with the easiest problem, which is making change for 0, where we already know the solution (i.e., the smallest number of coins) is 0. We then move on to making change for 1, where the solution (assuming there is a 1 coin denomination) is 1. We then go through every amount of change to make, at every step considering the solutions to the problems that came before it and the ways we can fold in new coin denominations.

[2] `https://en.wikipedia.org/wiki/Greedy_algorithm`
[3] `https://en.wikipedia.org/wiki/Dynamic_programming`

Every amount we consider can thus be thought of as a column in a table, where the number of rows in the table is the number of possible coin denominations (plus one, to reflect the case of having no coins). At the i-th column and in row j, we're storing the minimum number of coins needed to make change for an amount i, assuming that the j smallest coin denominations are used. When we move to the next columns, we look at the minimum values in earlier rows and columns according to the recurrence relation.

For example, to fill in the row $\{1, 2, 5\}$ in column 5, we consider two things: how we've already made change for 5p with just the 1p and 2p coins, and how it would work to use the 5p coin to make change. For the first one, we look at the entries in column 5 for earlier rows; for now, it is not so hard to work out by hand that the minimum number of coins is 3 (two 2p coins and one 1p coin). For the second one, we look at column 0 (since the recurrence relation says to look at column $5 - 5$) and consider the minimum number there; here it is defined by our base cases to be 0. We then add 1 to it, according to the recurrence relation, and see which is less: the values in the earlier rows (which we decided was 3), or 1 plus the minimum value in column 0 (which we decided was 0). Since 1 is less than 3, we store 1 in this cell of the table.

We continue filling out this table until we get to the amount for which we actually want to make change. At this point, the minimum number of coins for this amount of change is simply the cell in the bottom right corner of the table. To find not only the minimum number of coins but also the actual coins themselves, we just need to modify the table to store the coins too.

Task 3. Take the pseudocode we worked through on the board and write it as a function `dynamic_change` in the same file `make_change.py` that has the same inputs and outputs as `greedy_change` but uses dynamic programming. In particular, your function should output not only the minimum number of coins but also a list of the coins that are used.

Again, test out this function by running it on both normal and weird coin denominations, and on various amounts. For comparison, run and print it next to the outputs of the greedy algorithm.

4 Recursion

Dynamic programming exemplifies well the benefits of decomposing a problem into smaller subproblems. This overall paradigm is known as *recursion*, which is a general and powerful way of reducing a problem to simpler versions of the same problem. Like DP, solving a problem via recursion requires doing two things: establishing a base case, and establishing a recurrence relation that dictates how solutions to larger problems depend on the solutions to smaller problems. Recursive functions can then be split into these two parts: if the input fits the base case, then the function returns the result for that. If instead it is a larger problem, the function implements the rules to take the result of running the function on a smaller problem and use it to solve the larger problem.

For example, let's consider the Fibonacci sequence. This is a sequence of numbers that occurs frequently in the natural world, and looks like $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$. The first two values are defined as $F_0 = 0$ and $F_1 = 1$, and then each subsequent value is defined by the recurrence relation $F_n = F_{n-2} + F_{n-1}$.

If we wanted to write a recursive function `fibonacci` to, on input n, compute the Fibonacci number F_n, we need to do the two things described above. Establishing a base case is easy, as it is defined for us by the sequence. Thus, if we are given $n = 0$ we return 0, and if we are given $n = 1$ we return 1. Next, to establish the rules, we again just follow the relation that defines the sequence and output the sum of the two previous numbers. This means the code for this function would look like:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

In particular, this function calls itself (in what is known as a *recursive call*), but it crucially uses smaller inputs ($n - 2$ and $n - 1$) when doing so. If it instead called itself on the original input n, it would run forever.

Task 4. Write and test a recursive function `r_len` in a file `recursion.py` to compute the length of a list. Before you even start coding, be sure to establish the base case and the recurrence relation.

5 Putting It All Together

In our dynamic programming solution for making change, we went to all the trouble of solving the smaller problems and storing the solutions in the table, but then we threw the table away at the end of the function call. This means that if we computed the minimum number of coins needed to make change for 89p, and then computed it for 14p, we'd redo the exact same computation twice.

The process of remembering the results of previous function calls is called *memoization*,[4] and allows us to store the table. This means we end up using more space in the computer's memory, but can potentially run the function much faster as we call it more and more.

Task 5. Write in the same file `make_change.py` a function `memo_dynamic_change` that alters the function `dynamic_change` to operate in a memoized fashion by taking as input and producing as output the table used to store solutions. The code can then run almost exactly as in `dynamic_change`, but should take the existing solutions into account and produce new columns in the table only as necessary.

Again, test out this function by running it on both normal and weird coin denominations, and on various amounts. For comparison, you may find it useful to add code into both this function and `dynamic_change` that counts the number of times through the loop, and in your tests to compute the change needed for higher coin denominations before doing smaller ones. Let's say you computed the change needed for 89p and then for 14p. How many times would you go through the loop in this function, compared to in the non-memoized one?

6 Handing In

As always, feel free to continue trying out the various concepts we've covered in this lab. Once you are done, add the files into a `lab06` directory; this should mean adding:

- `make_change.py`
- `recursion.py`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.

[4] `https://en.wikipedia.org/wiki/Memoization`