CS161 Summer 2022 Problem Session 5

Problem 1: Let Me Count The Ways

This is a classic problem that ultimately involves a form of dynamic programming that can be done by hand!

Suppose you are in the top left cell of a grid of cells with r rows and c columns. Your goal is to get to the bottom right cell, but you can only make moves 1 cell down or 1 cell to the right. (Not both at once – diagonal moves are not allowed.) You wonder how many *different* ways there are for you to do this.

(a) Show that the answer for a 3×3 grid is 6. (You can find all of the ways by hand.)

(b) Drawing paths by hand is not going to scale to larger grids. Consider dividing the problem up into subproblems: **for each cell, how many ways are there to get to that cell?** Let's try that approach for a 3×3 grid. The value for the top left cell is 1 – since we start there, there is trivially one way to "get" there. What are the other two values in the top row, and the other two values in the left column?

(c) Now consider the cell that is diagonally just below and to the right of the starting cell. How many ways are there to get there?

Then fill in these values for all of the cells in the 3×3 grid – again enumerating the possible paths by hand, as needed to get the answers.

(d) Inspecting these values, can you see a pattern? Specifically, how you can get a value using only some other nearby values?

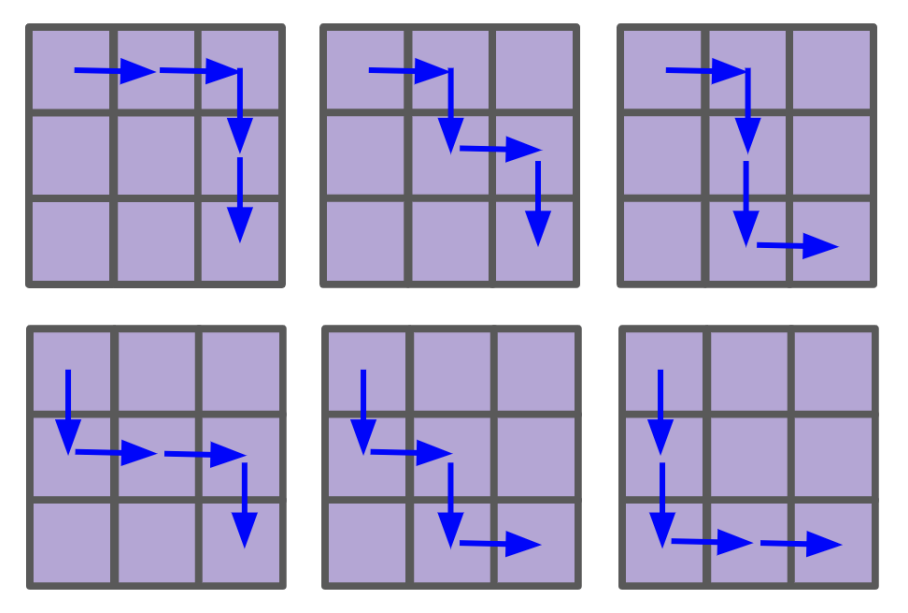
(e) With the answer to (d) in mind, is there an efficient order in which to fill in the cells of the grid? (Here, by efficient, I mean that any values that you need for each new calculation are present at the time that you need them.)

(f) As an extension, let's introduce a complication. How would you adapt your method if there were some impassable cells in the grid – i.e., cells that cannot not be entered? (It is guaranteed that the upper left cell is not of this type.)

(g) Here is a different extension just for fun – it has nothing to do with DP. Let's go back to the original version of the problem, without the impassable cells. Suppose that someone has found a valid path in the grid. Can you see a way to find another valid path that does not repeat any of their moves? That is, it is OK to enter a cell that they did, but your path cannot duplicate any consecutive pair of cells that they used.

Solutions to Problem 1

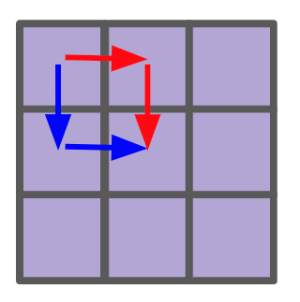
(a) The six paths are as follows:



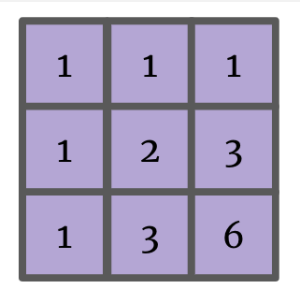
(b) The only way to get to a cell in the top row is to head directly right from the starting cell. Once we make even one move down, we can never return to the top row.

By a similar argument, there is only one way to get to each cell of the left column.

(c) There are two ways to get to this cell:



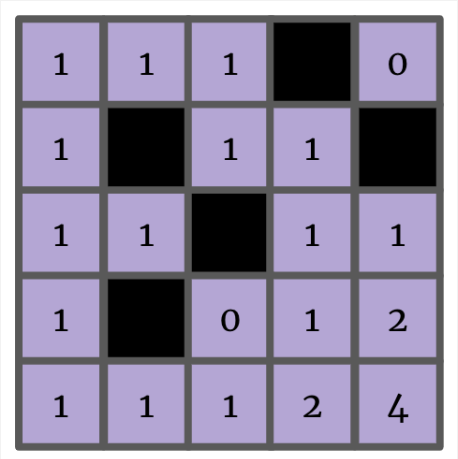
The number of ways to get to each cell are:



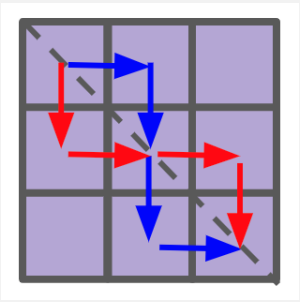
(d) We can see that – except for the upper left value – each value is the sum of the value immediately above it (if any) and the value immediately to the left of it (if any). This makes sense – any solution that reaches a cell has to do so either from above or from the left, and there can be no overlap between these two types of solution.

(e) Because each value depends only on the values above it and to the left of it, one valid strategy is to fill in all the values in the top row (from left to right), then all the values in the next row (from left to right), and so on. It also works to fill in all the values in the left column (from top to bottom), then all the values in the next column (from top to bottom), and so on. This ensures that we always have the value(s) we need.

(f) We can treat impassable cells as having values of 0 and being unenterable from the left or from above. Then we can use the same strategy as before, and it works. As an example:



(g) One simple (but perhaps hard-to-spot) solution is to take exactly the same path, but mirrored across the top-left-to-bottom-right diagonal:



Problem 2: Bellman-Ford

Sisi the systems gator is back with another distributed computing problem! Suppose we want to solve Problem 1 – the version with impassable cells – for an extremely large $n \times n$ grid. Suppose that this grid is divided up evenly into $\frac{n}{k}$ by $\frac{n}{k}$ square chunks, each of which is in the memory of one of k^2 machines (a machine knows only about its chunk and where it is in the larger grid). Also suppose that machines can send messages, much as in the model of Homework 2, Problem 5, but with some important differences:

- There is no central machine. Any machine can communicate with any other.
- Now each round consists of a computation phase followed by a communication phase.
- Here it is OK for a machine to send or receive more data (say, $O(\frac{n}{k})$ values) in each round of communication.

(We are also not worrying about the number of bits per value here. Assume that we're taking all the values modulo some prime p, so they can't get bigger than $p - 1$.)

Suppose we want to find the correct value for the lower right cell of the grid. Can you see a way to solve the problem in only $O(k)$ rounds of communication?

Problem 3: Costly Edits

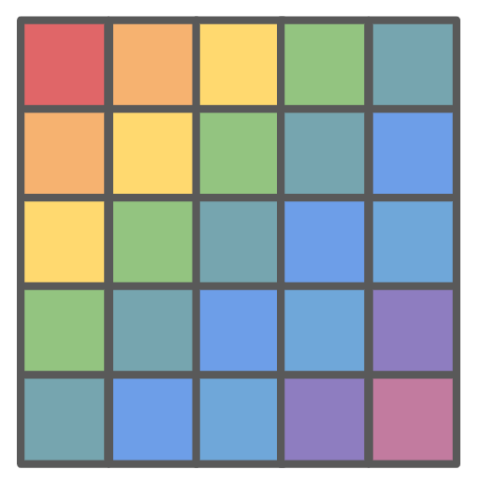
In lecture I mentioned that different kinds of DNA mutations might be substantially more or less likely. (For example, insertion or deletion of a single base pair is likely to screw up any coding region so badly that such a mutation would be heavily selected against, and would be very rare. On the other hand, substitutions might not be so bad, but even then, certain kinds of substitution might be more or less likely than others.)

Let's extend this idea to the edit distance algorithm from Lecture 11. Suppose that the insertion, deletion, and substitution operations now all have their own (positive) costs c_i, c_d, c_s, and we still want the minimum total cost. How would you modify our DP algorithm to take this into account?

Solution to Problem 2

Consider the grid of chunks. We can only start processing a chunk once we know the values in the rightmost column of its left neighbor (if any), and the values in the bottom row of its upper neighbor (if any). But then we can complete that chunk, and hand off its rightmost column to a right neighbor (if any), and its bottom row to a bottom neighbor (if any).

That is, in the first round, only the machine with the top left chunk of the grid does any computation. Then it sends info to its right and bottom neighbors. In the second round, those two do their computations, and so on. So each round uses a new "diagonal band" of machines:



We can see that there are no more than $O(k)$ diagonals (each has to originate in a different place in the leftmost column or bottom row) – in fact, it's exactly $2k - 1$. Therefore we only need that many rounds of communication.

Can we make this more efficient? It seems like a waste to have most machines idle most of the time. If each machine somehow has access to the entire dataset, then we could perform the same algorithm with only k machines total instead of k^2, although it would take the same number of rounds. There isn't really a way around the latter that I know of, because of how each chunk depends on the chunks above and to the left of it.

Solution to Problem 3

This turns out to be a pretty simple change! Here is the algorithm as given on Homework 5:

Base cases:

`edit_distance`$(\ell_1, \ell_2) = 0$
`edit_distance`$(\ell_1, k) = \ell_2 - k$, for $k < \ell_2$ // consumed w_1, insert at end to finish w_2
`edit_distance`$(k, \ell_2) = \ell_1 - k$, for $k < \ell_1$ // matched w_2, delete remainder of w_1

Recursion:

`edit_distance`$(i, j) = \min($
`edit_distance`$(i+1, j) + 1$, // deletion
`edit_distance`$(i, j+1) + 1$, // insertion
`edit_distance`$(i+1, j+1) + (1$ if $w_1[i] \neq w_2[j]$ else $0))$ // substitution or nothing

Then `edit_distance`$(0, 0)$ gives the edit distance between the two words.

All we have to do are change the costs from 1 to c_i, c_d, c_s as appropriate. The easiest part to miss is that the base cases also need to be modified:

This turns out to be a pretty simple change! Here is the algorithm as given on Homework 5:

Base cases:

`edit_distance`$(\ell_1, \ell_2) = 0$
`edit_distance`$(\ell_1, k) =$ **c_i**$(\ell_2 - k)$, for $k < \ell_2$
`edit_distance`$(k, \ell_2) =$ **c_d**$(\ell_1 - k)$, for $k < \ell_1$

Recursion:

`edit_distance`$(i, j) = \min($
`edit_distance`$(i+1, j) +$ **c_d**, // deletion
`edit_distance`$(i, j+1) +$ **c_i**, // insertion
`edit_distance`$(i+1, j+1) + ($**c_s** if $w_1[i] \neq w_2[j]$ else $0))$ // substitution or nothing

Problem 4: Bellman-Ford

Recall the question from Pre-Homework 4 about Waverly moving through a grid of cells like the one in Problem 1 above – for simplicity we'll say it's k rows by k columns, with k being large (so you don't need to think about edge cases). Waverly is trying to get from the top left cell to the bottom right cell, and moving into a cell incurs a cost (which might be negative). She wants to minimize the total cost.

Let's introduce an important difference from the Pre-Homework version: now Waverly can move in *any* direction, not just down and to the right. Because of this, we will have to add a guarantee that there are no negative cycles in the graph (though there may still be negative values).

Waverly decides to use the Bellman-Ford algorithm to compute the lowest possible cost. However, she thinks it goes on for too many rounds. She chooses to run the algorithm for only $2k$ rounds, rather than the usual k^2 rounds (here this is because k^2 is the number of vertices). Construct a grid for which Waverly's abridged version of Bellman-Ford will get the wrong answer.

Problem 5: Equal Sum

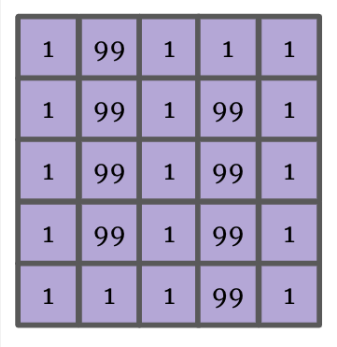
Here's a problem that even Indy's hash table methods can't handle! Given a list of n integers, determine whether it is possible to divide them into two groups such that each group has the same sum, or say that it is impossible. (You don't need to find the groups themselves.)

For this method to be tractable via dynamic programming, the integers need to be in a bounded range – here we will say that range is $[-100, 100]$.

This is challenging! As a hint, you can construct and populate a 2-dimensional table, where one of the dimensions is of size $n + 1$ (with the first column being the "start", before any elements have been looked at), and the other dimension keeps track of possible cumulative sums "so far". Or, you can find a way to do it without using a 2D table, and instead using, e.g., sets to keep track of possible subtotals you've been able to make so far.

Solution to Problem 4

A version of Bellman-Ford that runs for only $2k$ rounds could fail to find a low-cost but snaking path like the path of all 1s here, which is longer than $2k$ steps. (Suppose that the edge order happens to have all of the edges in that path reversed. Then only one "step" of progress is made each round, as in our pathological example in class.)



Solution to Problem 5

One thing we can notice right away is that if the sum of the values is odd, there is no way to divide them into two groups of equal sum, since one group's total would have to be odd and the other group's total would have to be even. So that's a quick check to do early on. It also helps to find the overall sum, because then that tells us the target sum for each group: exactly half the overall sum. Call that target t.

Therefore the problem is to select some subset of values that adds up to exactly t. We only really need to think about one of the two subsets: if the values we select add up to t, then so will the values left behind. But there are 2^n possible ways to put values into our subset (or not), since each value can be either chosen or not. We can't explicitly enumerate all of them.

Instead, let's proceed through the list in some order – say, the order in which the values were given to us. For each value, we decide whether to use it or not use it. As we do this, we update a branching set of possibilities that starts off looking like a tree, but eventually starts finding multiple paths to the same subtotal. The power of DP is that from there on out, we can ignore the multiple ways of getting somewhere and only ask where we can go from there.

Here is one somewhat hacky example using the Python "set" structure, and the list $[2, 3, 5, -1, 3]$. Notice that the "set" here is holding all the possible subtotals we have been able to make so far, not the set of values we will pick.

- The sum of the values is 12, so we are looking for a target of 6.
- We start with a set of subtotals with just 0. (The total is 0 since we have not added anything to our subset yet.)
- We consider whether to use 2 or not. If we do, our subtotal becomes 2. If not, it stays at 0. So now our set of subtotals is: {0, 2}.
- Now we consider whether to use 3 or not. Our set tells us that we have two subtotals so far to consider:
  - If we are at 0: we can either use the 3 and get 3, or not use the 3 and stay at 0.
  - If we are at 2: we can either use the 3 and get 5, or not use the 3 and stay at 2.

  So our new set of subtotals is: {0, 2, 3, 5}.
- Now we consider whether to use 5 or not:
  - 0: goes to 5 or 0
  - 2: goes to 7 or 2
  - 3: goes to 8 or 3
  - 5: goes to 10 or 5

  Up until now, it looked like we were going to explore the entire exponentially branching tree, but here we have our first example of two ways to reach the same subtotal (5). But now we stop exploring those branches individually – they are essentially merged into one. Our new set of subtotals is: {0, 2, 3, 5, 7, 8, 10}.
- Now we consider whether to use -1 or not. While doing this, we actually find that we can get to 6 (by adding -1 to 7), so we stop and declare victory!

However, if we had gone through this entire process and never found our target, we would have declared that the case was impossible.

```
def solve(ls):
    sm = sum(ls)
    if sm % 2 == 1:
        return "IMPOSSIBLE"
    target = sm // 2
    s = set([0]) # these sets will hold the subtotals we can make
    for v in ls:
        new_s = set()
        for w in s:
            new_s.add(w)
            new_s.add(w+v)
        if target in new_s:
            return "POSSIBLE"
        s = new_s
    return "IMPOSSIBLE"

assert solve([1, 2, 3, 4, 5]) == "IMPOSSIBLE"
assert solve([0]) == "POSSIBLE"
assert solve([-1, 1]) == "POSSIBLE"
assert solve([1, 2, 4, 8, 16, 32, 64, 128]) == "IMPOSSIBLE"
assert solve([1, 2, 4, 8, 16, 32, 64, 127]) == "POSSIBLE"
assert solve([1, 2, 4, 8, 16, 32, 64, 99]) == "POSSIBLE"
assert solve([-10000000, 4999999, -10000000, 1, 25000000]) == "POSSIBLE"
```

Problem 6: Do You Even Lift?

The standard set of weight plates in most gyms – at least the ones that use pounds – is: 2.5, 5, 10, 25, 35, 45.[1] When doing an exercise like a bench press or a squat, these weights are loaded onto a 45 pound bar, symmetrically – that is, if you put a 35 on one side, you had better also put a 35 on the other side. For example, one way to get a total of 320 is to put three 45s and one 2.5 on each side.

Brutus is preparing for a bench press and wants the total weight (bar + weight plates) to be exactly n pounds. (Brutus chooses some n that can actually be produced with the available weights; assume there are infinitely many copies of each weight plate available.[2]) He has asked Waverly to help load up the bar. However, Waverly wants to find a solution that involves as few weight plates as possible. (Technically, as few *pairs* of plates as possible, since they have to be loaded symmetrically!)

Waverly devises the following algorithm: start with the empty bar, and keep adding the heaviest pair of plates possible that doesn't cause the total weight so far to exceed n. E.g., if Brutus asked for 320 lbs. (which is just a warm-up for him!), Waverly's strategy would involve adding three sets of 45s, and then one set of 2.5s to cover the remaining 5 pounds.

Like many greedy strategies, this method *almost* works. Can you find the smallest value of n for which it fails – i.e., it uses more plates than it needs to for the target weight? This would be cumbersome to do by hand, so I recommend writing a program to simulate this greedy strategy, writing a correct dynamic programming algorithm, and going through all possible values comparing the results of the two methods until you find a difference.

[1] There are 100 lb. plates, but these are for vanity leg presses where the sled moves 1 inch. Non-powerlifter gyms have the good sense not to offer them; it is very easy to hurt oneself with a 100 lb. iron disc!

[2] In my experience, this is a bad assumption. In the small gym I used to use in the Before Times, it was not uncommon for most of the 45s to end up in a corner because someone was doing deadlifts.

Solution to Problem 6

At first, this problem might look somewhat like the knapsack problem, but there are important differences: knapsack is trying to maximize a value while staying under a weight limit, whereas we are trying to minimize a *number of items* while hitting a *specific* value.

In fact, this is exactly a different classic problem: the "change-counting" problem, which asks how to give a particular amount of change using the fewest coins possible. One way to solve this is via the following recurrence:

- As one base case, if our current value of n is less than 0, we must have overloaded the bar and this is not a solution.
- As another base case, if our current value of n is exactly 0, then we have no more weights to add, so we return 0.
- Otherwise, we try using one pair of each of the six possible types of weight plate, in turn. Note that these are all *separate* situations – what if we did this vs. that.
  - We consider adding two 45s: we solve the subproblem for $n - 2(45)$, then add 2 to the result to represent the two plates we just added.
  - ...
  - We consider adding two 2.5s: we solve the subproblem for $n - 2(2.5)$, then add 2 to the result to represent the two plates we just added.

  And then we take the minimum of all the values returned in this way.

Because we are always adding only 2 plates at a time, another way to solve the problem would be to use BFS on a graph with vertices representing $0, 5, 10, ..., n$ remaining pounds, where there is a directed edge from one vertex to another if the first is exactly $2k$ larger than the second, for some k in the set $\{2.5, ..., 45\}$ of allowable plates.

Here is the code I wrote to check the answer, which turns out to be $\boxed{165}$. (This is 120 without the bar.) In this situation, the greedy strategy tries to use two 45s, leaving it with 30 lbs, then has to use two 10s, leaving it with 10 lbs., then has to use two 5s, for a total of 6 plates. But the DP solution correctly uses two 35s and two 25s, for a total of 4 plates.

```
import sys

WEIGHTS = [90, 70, 50, 20, 10, 5] # pairs of 45s, 35s, ...

def solve_greedy(n):
    plates_used = 0
    n -= 45 # weight of the bar
    for i in range(len(WEIGHTS)):
        while n >= WEIGHTS[i]:
            n -= WEIGHTS[i]
            plates_used += 2
    return plates_used

INF = 99999999
def solve_dp(n):
    memo = {} # I'm being lazy again and doing top-down DP
    def helper(n):
        if n in memo:
            return memo[n]
        if n < 0:
            return INF # hackily indicate "this is not a valid solution"
        elif n == 0:
            return 0
        best = INF
        for i in range(len(WEIGHTS)):
            best = min(best, helper(n-WEIGHTS[i]) + 2)
        memo[n] = best
        return best
    return helper(n-45)

n = 45
while True:
    g = solve_greedy(n)
    d = solve_dp(n)
    if g != d:
        print("Weight {}: greedy used {} plates, dp used {}".format(n, g, d))
        #sys.exit(0)
    n += 5
```