Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Dynamic Programming Introduction: Fibonacci Numbers

The Fibonacci sequence is defined by the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

with base cases $F_0 = 0$ and $F_1 = 1$. Back in CS 61A, we learned how to write a program to find the nth Fibonacci number, which would look something like:

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

However, this program is actually super slow! In the box below, show that calling `fibo(n)` takes $2^{\Theta(n)}$ time.

Challenge: show that the runtime is $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

If you didn't above, in the box below draw out the recurrence tree produced when calling `fibo(5)`. Do you notice any repeated computations (i.e. nodes)?

In the recurrence tree, we notice that we end up recomputing some values multiple times. For instance, we end up computing F_1 five separate times! To reduce the amount of recomputing we have to do, we can **store each fibonacci number in an array after computing it**. This way, we can simply index into that array when we need that value, rather than recomputing it every time we recurse, as seen in the code below:

```
def optimized_fibo(n):
    # an array we use to store already-computed fibonacci numbers
    stored_fibos = [-1 for _ in range(n+1)]
    stored_fibos[0] = 0
    stored_fibos[1] = 1

    def fibo(n):
        # base case
        if n <= 1:
            return n

        # if we've already computed fibo(n) before, we can reuse it via
            stored_fibos!
        if stored_fibos[n] != -1:
            return stored_fibos[n]

        # if we haven't already computed fibo(n), then we need to recurse as before:
        # make sure to store it in stored_fibos so that we can use it in the future!
        stored_fibos[n] = fibo(n-1) + fibo(n-2)
        return stored_fibos[n]
```

What is the runtime of this new algorithm? Briefly justify.

This is essentially all that DP is: recursion plus storing information (memoization) so that we don't have to fully solve any subproblems more than once.

Now, there are actually two ways to implement DP algorithms. The implementation that you've completed above uses a **top-down** approach, i.e. you start from the largest subproblem (top) and repeatedly recurse on smaller subproblems (going down). The other implementation method uses a **bottom-up** approach, which starts from the smallest subproblems (i.e. the base cases), and builds up larger subproblems in an iterative manner. Our Fibonacci code can be rewritten in a bottom-up fashion as well:


```python
def fibo_dp_bottom_up(n):
    stored_fibos = [-1 for _ in range(n+1)]

    # define base cases at the "bottom"
    stored_fibos[0] = 0
    stored_fibos[1] = 1

    # build your subproblems "up" from your smaller subproblems
    for i in range(2, n+1):
        stored_fibos[i] = stored_fibos[i-1] + stored_fibos[i-2]

    # what element in stored_fibos represents the nth fibonacci number?

    answer = stored_fibos[n]
    return answer
```

What is the space complexity of this algorithm?

One advantage to bottom-up approaches in dynamic programming is that, if at some point a subproblem is no longer needed, we can throw away this memory. Let's try it with the Fibonacci problem: can you modify it to only use $O(1)$ extra space?

```python
def fibo_dp_bottom_up_constant_space(n):
    # define base cases here

    if __________:

        ____________________

    ____________________

    ____________________

    # build your subproblems "up" from your smaller subproblems
    for i in range(2, n+1):

       ______________________________________

       ______________________________________

       ______________________________________

    # where are you storing the nth fibonacci number?

    answer = ____________________
    return answer
```

2 Planting Trees

This problem will guide you through the process of writing a dynamic programming algorithm.

You have a garden and want to plant some apple trees in your garden, so that they produce as many apples as possible. There are n adjacent spots numbered 1 to n in your garden where you can place a tree. Based on the quality of the soil in each spot, you know that if you plant a tree in the ith spot, it will produce exactly x_i apples. However, each tree needs space to grow, so if you place a tree in the ith spot, you can't place a tree in spots $i-1$ or $i+1$. What is the maximum number of apples you can produce in your garden?

(a) Give an example of an input for which:

- Starting from either the first or second spot and then picking every other spot (e.g. either planting the trees in spots $1, 3, 5\ldots$ or in spots $2, 4, 6\ldots$) does not produce an optimal solution.
- The following algorithm does not produce an optimal solution: While it is possible to plant another tree, plant a tree in the spot where we are allowed to plant a tree with the largest x_i value.

(b) To solve this problem, we'll think about solving the following, more general problem: "What is the maximum number of apples that can be produced using only spots 1 to i?". Let $f(i)$ denote the answer to this question for any i. Define $f(0) = 0$, as when we have no spots, we can't plant any trees. What is $f(1)$? What is $f(2)$?

(c) Suppose you know that the best way to plant trees using only spots 1 to i does not place a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$.

Hint: What spots are we left with? What is the best way to plant trees in these spots?

(d) Suppose you know that the best way to plant trees using only spots 1 to i places a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$.

(e) Describe a linear-time algorithm to compute the maximum number of apples you can produce.

Hint: Compute $f(i)$ for every i. You should be able to combine your results from the previous two parts to perform each computation in $O(1)$ time.

3 Bellman-Ford Formulations

Recall that Dijkstra's algorithm relies on the fact that we will visit vertices in increasing order of distance. So if the shortest path to a vertex u goes through v, we will visit v before u because $dist[v] < dist[u]$. This is not true when we have negative edges, since it can happen that even if v is visited before u, we could have $dist[v] > dist[u]$. In this problem, we will see how the Bellman-Ford algorithm avoids this issue through DP.

Let $G = (V, E)$ be a directed weighted graph with possibly negative weights, such that $|V| = n$ and $|E| = m$. We want to find the shortest path from a fixed source s to every vertex in V.

(a) Suppose G has no cycles of negative total weight (i.e. no negative cycles). Explain why for any pair of vertices u, v, there is a shortest path that has most $n - 1$ edges. *Hint: If the shortest path has more than $n - 1$ edges, what can we say about the number of vertices on the path?*

(b) In class, we defined a subproblem $d_i[v]$ to denote shortest distance from s to v using **at most i edges**. From this, we derived the RELIABLE-SSSP DP algorithm on $n - 1$ edges:

```
1: function RELIABLE-SSSP(Weighted graph G = (V, E), Vertex s)
2:     d_0[s] ← 0
3:     for v ≠ s in V do
4:         d_0[v] = +∞
5:     for i ∈ {1, ··· , n − 1} do
6:         for vertex v ∈ V do
7:             d_i[v] = min(d_{i−1}[v], min_{(u,v)∈E}(d_{i−1}[u] + w(u, v)))
```

Describe in words each part of the recursive relation.

(c) Suppose we replace the final line with $d[v] = \min(d[v], \min_{(u,v)\in E}(d[u] + w(u,v)))$. In other words, suppose we try to reduce the space complexity of RELIABLE-SSSP by a factor of n by overwriting $d[v]$ with a potentially new value. Explain why, at the end of this function, $d[v]$ will be exactly equal to $d_{n-1}[v]$ from the original RELIABLE-SSSP if G has no negative cycles.

Hint: Break this down into showing that $d[v] \leq d_{n-1}[v]$ and $d[v] \geq d_{n-1}[v]$. Consider the paths that correspond to $d[u]$ and $d[v]$ at various points of the for loop. Notice that whenever we update $d[v]$, the corresponding path that does so has a smaller total weight. What types of paths is $d[v]$ the minimum weight over?

(d) Now, suppose that we replace the inner for loop of RELIABLE-SSSP (the one over $v \in V$) with:

---

for edge $(u, v) \in E$ **do**
$d[v] = \min(d[v], d[u] + w(u, v))$

Use a similar argument as part (c) to explain why $d[v] = d_{n-1}[v]$ from RELIABLE-SSSP in this case as well assuming that G has no negative cycles.

(e) One crucial part of the proof in part (a) relies on the lack of negative total weighted cycles. How can we detect a negative cycle in a graph?

Hint: Think about how we defined $d_i[v]$. If there were a negative cycle, what would happen if we reached this cycle?

4 Change Making

You are given an unlimited supply of coins of denominations $v_1, \ldots, v_n \in N$ and a value $W \in N$. Your goal is to make change for W using the minimum number of coins, that is, find a smallest set of coins whose total value is W.

(a) Design a dynamic programming algorithm for solving the change making problem. What is its running time?

(b) You now have the additional constraint that there is only one coin per denomination. Does your previous algorithm still work? If not, design a new one.

5 Counting Strings

A string $x[1, \ldots, n]$ consisting of the letters a, b is called a *boring* string if the strings aba, bab do not appear as substring. Formally, $x[1, \ldots, n]$ is *boring* if none of the triples $\{x[1, 2, 3], x[2, 3, 4], \ldots, x[n-2, n-1, n]\}$ is an "aba" or "bab".

Devise a dynamic programming algorithm to compute the number of all possible *boring* strings of length n.

1. What are the subproblems? Give a succinct and precise definition for the subproblems.

2. How many subproblems do you have?

3. Write down the recurrence relation.

