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 $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Solution: The runtime recurrence is $T(n) = T(n-1) + T(n-2)$.

We can lower bound it by, $T(n) \geq 2T(n-2)$, so we know $T(n) = \Omega(2^{n/2})$. We can also upper-bound it by $T(n) \leq 2T(n-1)$, which gives $T(n) = O(2^n)$.

Hence, $T(n) = 2^{\Theta(n)}$. However, we can actually compute a more precise runtime! Since we know the runtime is exponential with respect to n, we can write the runtime in the form $T(n) = \Theta(a^n)$. Then, plugging this into the recurrence, we have

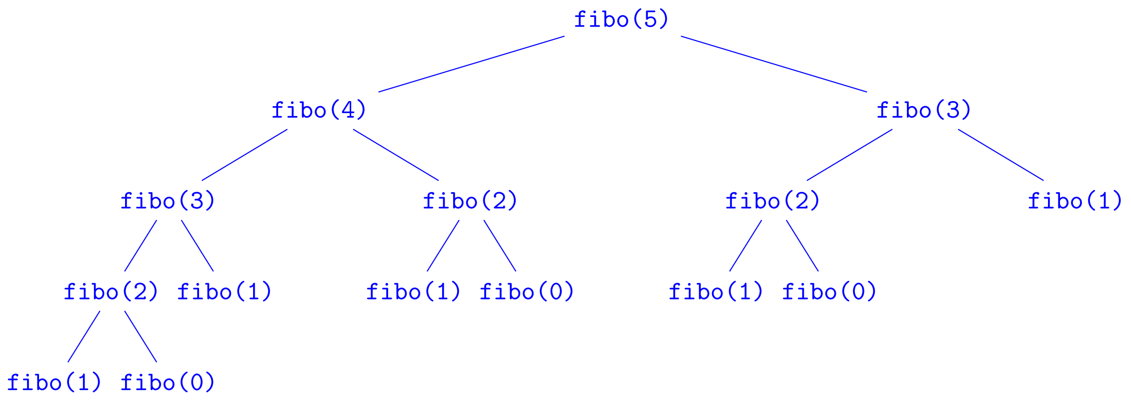
$$a^n = a^{n-1} + a^{n-2}$$
$$a^2 = a + 1$$
$$a^2 - a - 1 = 0$$

where we can divide by a^{n-2} since $a \neq 0$. By the quadratic formula, we get that $a = \frac{1\pm\sqrt{5}}{2}$. Since a must be positive, we conclude that $a = \frac{1+\sqrt{5}}{2}$ and thus

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

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)?

Solution:



In the recurrence tree, we notice that we end up recomputing some of same values multiple. For instance, we end up computing F_1 5 times! To reduce the number 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. To implement this, fill out the blank lines in the code below:

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

    def fibo(n):
        # base case

        if __________:

            return __________

        # if we've already computed fibo(n) before, we can reuse it via
            stored_fibos!

        if ____________________:

            return ____________________

        # 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!

        ______________________________

        return ____________________
```

Solution:

```
def optimized_fibo(n):
    stored_fibos = [-1 for _ in range(n+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?

Solution: Observe that for each $k \leq n$, the kth Fibonacci number will computed exactly once. All the other times `fib(k)` will return the stored value. Furthermore, `fib(k)` is called at most twice: by `fib(k+1)` and `fib(k+2)`.

Hence the runtime is $O(n)$.

Congratulations, you've just implemented your first **dynamic programming (DP)** algorithm! This is essentially all that DP is: recursion plus storing stuff (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.

Referencing your previous top-down approach, fill in the blank lines in the code below to complete the bottom-up implementation of the Fibonacci DP algorithm:

```
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):

        ________________________________________

    # what element in stored_fibos represents the nth fibonacci number?

    answer = ____________________
    return answer
```

Solution:

```
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
```

Yay! You've now learned how to implement both types of DP algorithms.

(Challenge) What is the space complexity of your algorithm? Can you modify it to only use O(1) extra space?

```
def fibo_dp_bottom_up(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
```

Solution: The space complexity of our algorithm was $O(n)$, as we had to create a $n+1$ length array to store all the $F_0, F_1, F_2, \ldots, F_n$ fibonacci numbers.

Note that for any given fibonacci number, it only depends on the previous two fibonacci numbers. So we can actually modify our algorithm to store the last two fibonacci numbers computed at any given point of time (instead of all the fibonacci numbers seen so far). This is implemented below:

```
def fibo_dp_bottom_up(n):
    # define base cases here
    if n <= 1:
        cur_fib = n

    prev_2_fib = 0
    prev_1_fib = 1

    # build your subproblems "up" from your smaller subproblems
    for i in range(2, n+1):
        cur_fib = prev_2_fib + prev_1_fib

        # update prev_2_fib and prev_1_fib for i+1
        prev_2_fib = prev_1_fib
        prev_1_fib = cur_fib

    # where are you storing the nth fibonacci number?
    answer = cur_fib
    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.

Solution:

(a) For the first algorithm, a simple input where this fails is $[2, 1, 1, 2]$. Here, the best solution is to plant trees in spots 1 and 4. For the second algorithm, a simple input where this fails is $[2, 3, 2]$. Here, the greedy algorithm plants a tree in spot 2, but the best solution is to plant a tree in spots 1 and 3.

(b) $f(1) = x_1$, $f(2) = \max\{x_1, x_2\}$

(c) If we don't plant a tree in spot i, then the best way to plant trees in spots 1 to i is the same as the best way to plant trees in spots 1 to $i-1$. Then, $f(i) = f(i-1)$.

(d) If we plant a tree in spot i, then we get x_i apples from it. However, we cannot plant a tree in spot $i-1$, so we are only allowed to place trees in spots 1 to $i-2$. In turn, in this case we can pick the best way to plant trees in spots 1 to $i-2$ and then add a tree at i to this solution to get the best way to plant trees in spots 1 to i. So we get $f(i) = f(i-2) + x_i$.

(e) Initialize a length n array, where the ith entry of the array will store $f(i)$. Fill in $f(1)$, and then use the formula $f(i) = \max\{f(i-1), x_i + f(i-2)\}$ to fill out the rest of the table in order. Then, return $f(n)$ from the table.

3 Longest Common Subsequence

In lecture, we covered the longest increasing subsequence problem (LIS). Now, let us consider the longest common subsequence problem (LCS), which is a bit more involved. Given two arrays A and B of integers, you want to determine the length of their longest common subsequence. If they do not share any common elements, return 0.

For example, given $A = [1, 2, 3, 4, 5]$ and $B = [1, 3, 5, 7]$, their longest common subsequence is $[1, 3, 5]$ with length 3.

We will design an algorithm that solves this problem in $O(nm)$ time, where n is the length of A and m is the length of B.

(a) Define your subproblem.

Hint: looking at the Edit Distance subproblem may be helpful.

(b) Write your recurrence relation.

(c) In what order do we solve the subproblems?

(d) What is the runtime of this dynamic programming algorithm?

(e) What is the space complexity of your DP algorithm? Is it possible to optimize it?

Solution:

Algorithm Description: Let $L[i][j]$ be the length of the LCS between the first i characters of A and j characters of B. (i.e between $A[0:i]$ and $B[0:j]$). Then the final answer will be $L[n][m]$. The recurrence is given as follows:

$$L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if } A[i-1] = B[j-1] \\ \max\{L[i-1][j], L[j-1][i]\} & \text{otherwise} \end{cases} \tag{1}$$

Correctness: The correctness follows by induction. Observe that if the last characters of A and B match, then any longest common subsequence should have the same last character. Otherwise, at least one of $A[i-1]$ and $B[j-1]$ will not be in the solution, so we take the maximum over the possibilities.

Runtime: Our DP has $(n+1)(m+1)$ states and calculating each state takes $O(1)$. Hence, the algorithm runs in $O(nm)$.

Space: Naively, we can just store all the states, which would take up $O(nm)$ space. However, we can note that to compute any $L[i][j]$, we only need the subproblems involving $i-1$, $j-1$, and i, j. Thus, we simply need to store the last 2 "rows" or "columns" of the DP table, yielding a space complexity of $O(2\min\{n,m\}) = O(\min\{n,m\})$.

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.

Solution:

(a) For $0 \leq w \leq W$, define

$$f(w) = \text{the minimum number of coins needed to make a change for } w.$$

It satisfies

$$f(w) = \min \begin{cases} 0 & \text{if } w = 0, \\ 1 + \min_{j\,:\, v_j \leq w} f(w - v_j) \\ \infty \end{cases}$$

The answer is $f(W)$, where ∞ means impossible. It takes $O(nW)$ time.

(b) For $0 \leq i \leq n$ and $0 \leq w \leq W$, define

$$f(i, w) = \text{the minimum number of coins (among the first } i \text{ coins) needed}$$
$$\text{to make a change for } w\text{, having one coin per denomination.}$$

It satisfies

$$f(i,w) = \min \begin{cases} 0 & \text{if } i = 0 \text{ and } w = 0, \\ f(i-1,w) & \text{if } i > 0, \\ 1 + f(i-1, w - v_i) & \text{if } i > 0 \text{ and } v_i \le w, \\ \infty & \end{cases}$$

The answer to the problem is $f(n, W)$. It takes $O(nW)$ time.