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 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.

2 Maximum Subarray Sum Revisited

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \le j} \sum_{k=i}^{j} A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$.

In discussion 2, we saw how to find the maximum subarray sum in $O(n \log n)$ time using divide and conquer. This problem can actually be solved in linear time. Describe a $O(n)$-time algorithm and prove its correctness.

Hint: Use dynamic programming.

Solution:

Algorithm Description:

- Subproblem Definition: $dp[i]$ denotes the maximum sum subarray that ends at index i (non-inclusive of the empty subarray).
- Base Case(s): $dp[0] = A[0]$
- Recurrence Relation: we compute the dp values in increasing order of i (for $i = 1 \dots n-1$) with the recurrence,
$$dp[i] = \max(0, dp[i-1]) + A[i]$$
- Final Answer: for the final answer we output the largest $dp[i]$ or 0 if it is negative.

Here is pseudocode illustrating this algorithm:

```
def max_subarray_sum(A):
    n = len(A)
    dp = [0 for i in range(n)]

    # base case
    dp[0] = A[0]

    # using dynamic programming to compute the rest of dp array
    for i in range(1, n):
        dp[i] = max(0, dp[i-1]) + A[i]

    return max(max(dp), 0)
```

Note that this solution uses $O(n)$ space, as we need to store the entire dp array. However, when computing a given $dp[i]$, we only use the previous $dp[i-1]$, so we can optimize the space complexity of this solution by just storing a "running sum", as follows:

```
def max_subarray_sum(A):
    n = len(A)

    # base case
    max_sub_sum = running_sum = A[0]

    # using dynamic programming to compute the rest of dp array
    for i in range(1, n):
        running_sum = max(0, running_sum) + A[i]
        max_sub_sum = max(max_sub_sum, running_sum)

    return max(max_sub_sum, 0)
```

This solution uses $O(1)$ space since it only has to keep track of three integers n, `max_sub_sum`, and `running_sum`! (Note that we do not count the space complexity of input A when determining this)

Also, if you've done a lot of leetcode before, you may recognize this algorithm as **Kadane's algorithm**.

Proof of Correctness: We use induction to show that $dp[i]$ is correct for all $i \geq 0$.

- Base Case ($i = 1$): when there's just a single element, the only subarray option is $A[0]$.
- Inductive Hypothesis ($i = k$): assume that $dp[k]$ is the correct maximum sum subarray that ends at index k.
- Inductive Step ($i = k \implies i = k + 1$): when finding $dp[k + 1]$, we either want to use $dp[k]$ or ignore it (and start a new subarray sum). Thus, we break it down into two cases:
  - When $dp[k] \geq 0$, it's optimal to use it and simply add $A[k + 1]$ to try to potentially maximize $dp[k + 1]$.
  - When $dp[k] < 0$, it's optimal to forget about it and just start a new subarray sum at $k+1$. This is because when $dp[k] < 0$, we have that $dp[k] + A[k + 1] < 0 + A[k + 1] = A[k + 1]$.

  The two cases are summarized below:

$$dp[k+1] = \begin{cases} dp[k] + A[k] & dp[k] \geq 0 \\ A[k] & dp[k] < 0 \end{cases}$$
$$= \max(dp[k], 0) + A[k]$$

  which is exactly our recurrence relation.

Runtime Analysis: Since we perform $O(n)$ iterations of constant-time operations, the overall runtime is $O(n)$

3 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

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

It satisfies

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

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

4 Copper Pipes

Bubbles has a copper pipe of length n inches and an array of nonnegative integers that contains prices of all pieces of size at most n. He wants to find the maximum value he can make by cutting up the pipe and selling the pieces. For example, if length of the pipe is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Give a dynamic programming algorithm so Bubbles can find the maximum obtainable value given any pipe length and set of prices. Clearly describe your algorithm and analyze its runtime (proof of correctness not required).

Solution:

Main idea: We create a recursive formula, where for each subproblem of length k we choose the cut-length i such that $Price(i) + Value(k-i)$ is maximized. Here $Price(i)$ is the price of selling the full pipe of length i and $Value(k-i)$ is the amount obtained after optimally cutting the pipe of length $k-i$.

```
def cutPipe(price, n):
    val = [0 for x in range(n+1)]
    val[0] = 0

    # Build the table val[] in bottom up manner and return
    # the last entry from the table
    for i from 1 to n:
        max_val = 0
        for j from 0 to i-1:
             max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    return val[n]
```

Proof: An inductive proof on the length of the pipe will show that our solution is correct. We let $cutPipe(n)$ represent the optimal solution for a pipe of length n. Base case: If the pipe is length 1, $cutPipe(1) = Price(1) = Val(1)$. Inductive: Assume the optimal price is found for all pipes of length less than or equal to k. If the first cut the algorithm makes x_1 is not optimal, then there is an x_1' such that $Val((k+1)-x_1) + Price(x_1) < Val((k+1)-x_1') + Price(x_1')$. By the induction hypothesis, this implies that $cutPipe((k+1)-x_1) + Price(x_1) < cutPipe((k+1)-x_1') + Price(x_1')$. So the algorithm must have chosen x_1' instead of x_1, by construction (a contradiction). Therefore, by induction $cutPipe(n) = Val(n)$ for all $n > 0$.

Run-time: The algorithm contains two nested for-loops resulting in a run-time of $O(n^2)$.