CS 161 **Problem Set 6**
Spring 2023 **Due:** Wednesday, May 24 at 1pm on Gradescope.

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

Please list all members of your homework group and all other students in the class you have collaborated with on this homework, in accordance with the Collaboration Policy, at the beginning of each homework submission.

Exercises

We recommend you do the exercises on your own before collaborating with your group.

---

1. **(8 pt.)** Let A be an array of length n containing real numbers. A *longest increasing subsequence* (LIS) of A is a sequence of indices $0 \leq i_1 < i_2 < \dots i_\ell < n$ such that $A[i_1] < A[i_2] < \cdots < A[i_\ell]$ and ℓ is as large as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then one LIS is $i_1 = 1, i_2 = 3, i_3 = 4, i_4 = 6$, which corresponds to the increasing values $3, 5, 6, 8$. (Notice that a LIS doesn't need to be unique).

   In the following parts, we'll walk through the recipe that we saw in class for coming up with DP algorithms, in order to develop an $O(n^2)$-time algorithm for finding an LIS.

   (a) **(2 pt.) (Identify optimal sub-structure and a recursive relationship).** We'll come up with the sub-problems and recursive relationship for you, although you will have to justify it. Let $D[i]$ be the length of the longest increasing subsequence of $[A[0], \dots, A[i]]$ that ends on $A[i]$. Explain why

   $$D[i] = \max\left(\{D[k] + 1 \,:\, 0 \leq k < i, A[k] < A[i]\} \cup \{1\}\right).$$

   That is, $D[i]$ is the maximum of "1" and the values $D[k] + 1$, where k ranges over all values between 0 and $i - 1$ so that $A[k] < A[i]$.

   [**We are expecting:** *A short informal explanation (a paragraph or so). You do not need to write a formal proof.*]

   (b) **(3 pt.) (Develop a DP algorithm to find the value of the optimal solution)** Use the relationship above to design a dynamic programming algorithm that returns the length of the longest increasing subsequence. Your algorithm should run in time $O(n^2)$ and should fill in the array D defined above.

   [**We are expecting:** *Just pseudocode. No justification is required.*]

(c) **(3 pt.) (Adapt your DP algorithm to return the optimal solution)** Adapt your algorithm above to return the values of the LIS instead of its length. Your algorithm should run in time $O(n^2)$.

Note that you should return the values of the LIS: for example, for the array $A = [6, 3, 2, 5, 6, 4, 8]$ given at the beginning of this question, if your LIS is $[1, 3, 4, 6]$, you should return the corresponding values $[3, 5, 6, 8]$.

[**We are expecting:** *Pseudocode* ***AND*** *a short English explanation of what your algorithm is doing. You do not need to justify correctness or runtime.*]

Note: Actually, there is an $O(n \log(n))$-time algorithm to find an LIS, which is faster than the DP solution in this exercise!

Problems

2. **(7 pt.)** (**Improving on Divide-and-Conquer**) Consider the following problem, MinElementSum.

    MinElementSum(n, S): Let S be a set of positive integers, and let n be a non-negative integer. Find the minimal number of elements of S needed to write n as a sum of elements of S. (Note: it's okay to use an element of S more than once, but this counts towards the number of elements). If there is no way to write n as a sum of elements of S, return `None`.

    For example, if $S = \{1, 4, 7\}$ and $n = 10$, then we can write $n = 1 + 1 + 1 + 7$ and that uses four elements of S. The solution to the problem would be "4." On the other hand if $S = \{4, 7\}$ and $n = 10$, then the solution to the problem would be "`None`," because there is no way to make 10 out of 4 and 7.

    Your friend has devised a divide-and-conquer algorithm to solve MinElementSum. Their pseudocode is below.

```
def minElementSum(n, S):
     if n == 0:
         return 0
     if n < min(S):
         return None
     candidates = []
     for s in S:
         cand = minElementSum( n-s, S )
         if cand is not None:
              candidates.append( cand + 1 )
     if len(candidates) == 0:
         return None
     return min(candidates)
```

    Your friend's algorithm correctly solves MinElementSum. Before you start doing the problems on the next page, it would be a good idea to walk through the algorithm and to understand what this algorithm is doing and why it works.

[Questions on next page]

(a) **(1 pt.)** Argue that for $S = \{1, 2\}$, your friend's algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$).

[**Hint:** *You may use the fact (stated in class) that the Fibonacci numbers $F(n)$ satisfy $F(n) = 2^{\Omega(n)}$.*]

[**We are expecting:**

- *A recurrence relation that the running time of your friend's algorithm satisfies when $S = \{1, 2\}$.*
- *A convincing argument that the closed form for this expression is $2^{\Omega(n)}$. You do not need to write a formal proof.*

]

(b) **(3 pt.)** Turn your friend's algorithm into a top-down dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[**Hint:** *Add an array to the pseudocode above to prevent it from solving the same subproblem repeatedly.*]

[**We are expecting:**

- *Pseudocode* ***AND*** *a short English description of the idea of your algorithm.*
- *An informal justification of the running time.*

]

(c) **(3 pt.)** Turn your friend's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[**Hint:** *Fill in the array you used in part (b) iteratively, from the bottom up.*]

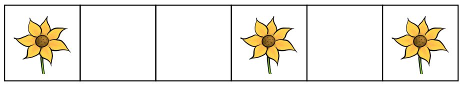
[**We are expecting:**

- *Pseudocode* ***AND*** *a short English description of the idea of your algorithm.*
- *An informal justification of the running time.*

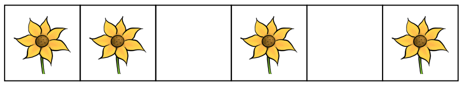
]

3. **(12 pt.)** **[Spring flowers!]** It's spring! You are planting your flower garden, which has n spots arranged in a line. Different spots in the garden will result in different quality flowers, measured by fragrance: suppose that the location i will result in flowers of frangrance $F[i]$, where $F[i]$ is a positive integer. Further, you cannot plant two plants directly next to each other, because they will compete for resources and wilt. Your goal is to create the most fragrant flower garden possible (summed up over all of the flowers in the garden).

   For example, if the input was $F = [21, 4, 6, 20, 2, 5]$, then you should plant flowers in the pattern



   and you would obtain fragrance $21 + 20 + 5 = 46$. You would **not** be allowed to plant flowers in the pattern



   because there are two flowers next to each other.

   You will first design a dynamic programming algorithm which runs in time $O(n)$, takes as input the array F, and returns the maximum fragrance possible given F. Do this by answering the two parts below.

   (a) **(3 pt.)** What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship satisfied between the sub-problems?

   [**We are expecting:**
   - *A clear description of your sub-problems.*
   - *A recursive relationship that they satisfy, along with any necessary base case(s).*
   - *An informal justification that the recursive relationship is correct.*

   ]

   (b) **(3 pt.)** Write pseudocode for your algorithm. Your algorithm should take as input the array F, and return a single number which is the maximum amount of fragrance possible. Your algorithm does not need to output the optimal way to plant the flowers. [**We are expecting:** *Clear pseudocode. You do not need to justify that your algorithm is correct, but correctness should follow from your reasoning in part (a).*]

   (c) **(6 pt.)** You're pretty happy with your approach from the previous parts, but you realized that you forgot about the cost of planting the flowers! Now there's also a length-n array C, so that $C[i]$ is a positive integer representing the cost of planting a flower in position i. You have a total budget of B. Design a DP algorithm that runs in time $O(nB)$, takes as input F, C, and B, and returns the maximum fragrance possible, given the additional constraint that you can't exceed your budget B: that is,

$$\sum_{i=1}^{n} C[i] \cdot \mathbf{1}[\text{there is a flower in position } i] \leq B.$$

Your algorithm does not need to return how to plant the flowers.

Note that you are still not allowed to plant two flowers next to each other.

[**Hint:** *Consider filling in a two-dimensional array.*]

[**We are expecting:** *A clear description of your sub-problems and the recursive relationship that they satisfy, pseudocode, and a short justification for the runtime.*]

4. **(6 pt.) [Fish fish eat eat fish.]** Plucky the Pedantic Penguin enjoys fish, and wants to catch as many fish as possible from two nearby Lakes named A and B. They have discovered that on some days the fish supply is better in Lake A, and some days the fish supply is better in Lake B. Plucky has access to two tables A and B, where $A[i]$ is the number of fish they can catch in Lake A on day i, and $B[i]$ is the number of fish they can catch in Lake B on day i, for $i = 0, \ldots, n-1$.

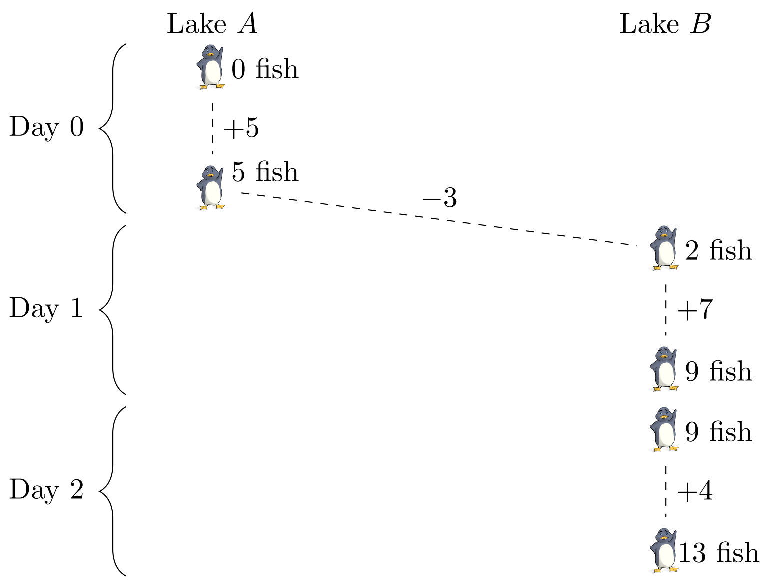
   If Plucky is at Lake A on day i and wants to be at Lake B on day $i+1$, they may pay L fish to a polar bear who can take them from Lake A to Lake B overnight; the same is true if they want to go from Lake B back to Lake A. The polar bear does not accept credit, so **Plucky must pay *before* they travel**. (And if they cannot pay, they cannot travel).

   Assume that when day 0 begins, Plucky is at Lake A, and they have zero fish. Also assume that $A[i]$ and $B[i]$ are positive integers for $i = 0, 1, \ldots, n-1$ and that L is also a positive integer.

   For example, suppose that $n = 3$, $L = 3$, and that A and B are given by

   $$A = [5, 2, 3] \qquad B = [2, 7, 4].$$

   Then Plucky might do:



   So Plucky's total fish at the end of day $n-1 = 2$ is 13.

   In this question, you will design an $O(n)$-time dynamic programming algorithm that finds the maximum number of fish that Plucky can have at the end of day $n-1$. Do this by answering the two parts below.

   (a) **(3 pt.)** What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between the sub-problems?

   [**We are expecting:**

   - *A clear description of your sub-problems.*

- *A recursive relationship that they satisfy, along with a base case.*
- *An informal justification that the recursive relationship is correct.*

]

(b) **(3 pt.)** Design a dynamic programming algorithm that takes as input A, B, L and n, and in time $O(n)$ returns the maximum number of fish that Plucky can have at the end of day $n-1$.

[**We are expecting:** *Clear pseudocode, and a justification of why it runs in time $O(n)$. You do not need to justify the correctness of your pseudocode, but it should follow from your reasoning in part (a).*]

5. **(4 pt.) [EthiCS: Vaccine Distribution]**

   In all of the problems above (and in general in the sorts of DP problems we solve on pedagogical CS161 HW assignments), we have a very tidy set of options and constraints. In the flower-bed problem, we want to maximize "fragrance" subject to not having two flowers next to each other; in Plucky's fishing problem, we want to maximize fish subject to Plucky being able to pay the polar bear. However, the real world is not so tidy. For example, imagine it is early 2021, and you are deciding how to allocate a limited number of COVID-19 vaccines. At a first pass, it might look a bit like our optimization problems with options and constraints. We'd like to optimize public health, by vaccinating some people, given a limited amount of vaccine and with the additional constraint of being equitable.

   We can see that our goals and constraints are not as easy to mathematically formulate as with our tidy story problems above, but in order to come up with a good policy, we need to try! How might you define the goal of "maximizing population health" and the constraint of "equitable access" for vaccine distribution? What other factors do you need to consider that aren't necessarily represented in the problem formulation of optimization and constraints?

   Of course, there is not a right answer to this question! What we are looking for is a serious attempt to think about how you might model this, and what the issues might be.

   [**We are expecting:** *(1) A paragraph or so describing what might contribute to formal definitions of "population health" and "equitable access," and (2) 2-3 sentences describing at least two other factors you might need to consider about vaccine distribution when designing a policy.*]

   (Optional Reading: Nature and NIH have articles about research that tested different vaccine distribution strategies)

6. **[Please Go Fill Out This Group-Work Survey] (1 pt.)**

   In collaboration with Professor Shima Salehi of the Stanford Ed School and her team, we have created a short google form on your experience with group-work in this class, which is linked below. SUNet IDs will be collected, but we will only use hashes of IDs for the study, so the responses will be effectively anonymous. The form is very short, and will take ≈ 1 minute to fill out: `https://forms.gle/qW9vRAyFGHQvg1UD9`.

   We really appreciate all your feedback and help with making CS161 the best possible learning experience it can be! If you would like to give more general feedback on the class as a whole, please check out the many possible options highlighted in the linked Ed post[1].

   Have you submitted your answers to the google form above?

   [**We are expecting:** *Yes*]

[1] `https://edstem.org/us/courses/38246/discussion/2970522`