Exercises

Please do the exercises on your own.

1. (8 pt.) In this exercise we’ll practice designing and analyzing dynamic programming algorithms. Let $A$ be an array of length $n$ containing real numbers. A longest increasing subsequence (LIS) of $A$ is a sequence $0 \leq i_0 < i_1 < \cdots < i_{\ell-1} < n$ so that $A[i_0] < A[i_1] < \cdots < A[i_{\ell-1}]$, so that $\ell$ is as large as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then a LIS is $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 6$ corresponding to the subsequence 3, 5, 6, 8. (Notice that a longest increasing subsequence 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 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 $[A[0], \ldots, A[i]]$ that ends on $A[i]$. Explain why

$$D[i] = \max \{ D[k] + 1 : 0 \leq k < i, A[k] < A[i] \} \cup \{ 1 \}. \quad [\text{We are expecting: A short informal explanation (a paragraph or so). It might be good practice to write a formal proof, but this is not required for credit.}]$$

(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 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: Pseudocode. No justification is required.]

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

[We are expecting: Pseudocode AND a short English explanation of what your algorithm is doing. You do not need to justify that it is correct.]

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!
2. (6 pt.) In this exercise we’ll practice designing and analyzing greedy algorithms. We’ll look at a continuous variant of the knapsack problem that we saw in class. You have a knapsack with a capacity of $Q$ ounces and there are $n$ items; the difference between this exercise and the version that we saw in class is that you can take a fractional amount of each item. For example, perhaps one item is 3.6 ounces of brightly colored sand; you can choose to take 2.5235 ounces of sand for your knapsack if that’s how much you want.

Each item $i$ has a value per ounce $v_i > 0$ (measured in units of dollars per ounce) and a quantity $q_i > 0$ (measured in ounces). There are $q_i$ ounces of item $i$ available to you, and for any real number $x \in [0, q_i]$, the total value that you derive from $x$ ounces of item $i$ is $x \cdot v_i$.

Your goal is to choose an amount $x_i \geq 0$ to take for each item $i$ in order to maximize the value $\sum_i x_i v_i$ that you receive while satisfying:

1. you don’t overfill the knapsack (that is, $\sum_i x_i \leq Q$), and
2. you don’t take more of an item than is available (that is, $0 \leq x_i \leq q_i$ for all $i$).

Assume that $\sum_i q_i \geq Q$, so there always is some way to fill the knapsack.

(a) (0 pt.) Suppose that you already have partially filled your knapsack, and there is some amount of each item left. What item should you take next, and how much?

[We are expecting: Nothing, this part is worth zero points, but it’s a good thing to think about before you go on to the next part.]

(b) (3 pt.) Design a greedy algorithm which takes as input $Q$ along with the tuples $(i, v_i, q_i)$ for $i = 0, \ldots, n - 1$, and outputs tuples $(i, x_i)$ so that (1) and (2) hold and $\sum_i x_i v_i$ is as large as possible. Your algorithm should take time $O(n \log n)$.

Note: If you have a list of tuples $(a_i, b_i, c_i)$, it is perfectly acceptable to say something like “Sort the list by $c_i$” in your pseudocode.

[We are expecting:  
- Pseudocode AND an English explanation of what it is doing.  
- A justification of the running time.]

(c) (3 pt.) Fill in the inductive step below to prove that your algorithm is correct.

- Inductive hypothesis: After making the $t$'th greedy choice, there is an optimal solution that extends the solution that the algorithm has constructed so far.
- Base case: Any optimal solution extends the empty solution, so the inductive hypothesis holds for $t = 0$.
- Inductive step: (you fill in)
- Conclusion: At the end of the algorithm, the algorithm returns a set $S^*$ of tuples $(i, x_i)$ so that $\sum_i x_i = Q$. Thus, there is no solution extending $S^*$ other than $S^*$ itself. Thus, the inductive hypothesis implies that $S^*$ is optimal.

[We are expecting: A proof of the inductive step: assuming the inductive hypothesis holds for $t - 1$, prove that it holds for $t$.]
Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own before collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

3. (7 pt.) [Duck Dance-off.] Two dancing duck troupes are having a dance-off. The rules are as follows. There is a dance floor, which is laid out as a row of \( n \) squares, where \( n \) is an even number. Each square has a score (a positive number), which is given by an array \( D \) of length \( n \). Each duck receives the score of the square it dances in, and the score for the whole team is the sum of the scores of each dancer in that team.

The two dancing duck troupes take turns adding dancers to the dance floor; but the rules are that a new dancer can only join next to an existing dancer, or next to the edge of the dance floor. The two troupes are colored green and white, and green goes first.

For example, the following would be a legal dance-off, with a dance-floor-array \( D = [5, 7, 3, 4, 4, 6] \).

\[
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 1} & & & & & \\
\end{array}
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 2} & & & & & \\
\end{array}
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 3} & & & & & \\
\end{array}
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 4} & & & & & \\
\end{array}
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 5} & & & & & \\
\end{array}
\begin{array}{cccccc}
5 & 7 & 3 & 4 & 4 & 6 \\
\text{Round 6} & & & & & \\
\end{array}
\]

At the end of this dance-off, the green ducks have a score of \( 5 + 4 + 6 = 15 \), while the white ducks have a score of \( 7 + 3 + 4 = 14 \), so the green ducks win. Notice that in the above example, the ducks may not have been using the optimal strategy.

For the questions below, “green ducks” refers to the dance troupe that goes first in this dance-off.

In this problem, you will design an algorithm to compute the best score that the green ducks can obtain, assuming that the white ducks are playing optimally. Your algorithm should run in time \( O(n^2) \).

(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?

<table>
<thead>
<tr>
<th>We are expecting:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• A clear description of your sub-problems.</td>
</tr>
<tr>
<td>• A recursive relationship that they satisfy, along with a base case.</td>
</tr>
<tr>
<td>• An informal justification that the recursive relationship is correct.</td>
</tr>
</tbody>
</table>
(b) (4 pt.) Write pseudocode for your algorithm. Your algorithm should take as input the array $D$, and return a single number which is the best score the green team can achieve. Your algorithm does not need to output the optimal strategy. It should run in time $O(n^2)$.

[We are expecting: Pseudocode AND a clear English description. You do not need to justify that your algorithm is correct, but correctness should follow from your reasoning in part (a).]

4. (8 pt.) [k-well-connected graphs.] Let $G = (V, E)$ be an undirected, unweighted graph with $n$ vertices and $m$ edges. For a subset $S \subseteq V$, define the subgraph induced by $S$ to be the graph $G' = (S, E')$, where $E' \subseteq E$, and an edge $\{u, v\} \in E$ is included in $E'$ if and only if $u \in S$ and $v \in S$.

For any $k < n$, say that a graph $G$ is $k$-well-connected if every vertex has degree at least $k$.

For example, in the graph $G$ below, the subgraph $G'$ induced by $S = \{a, b, c, d\}$ is shown on the right. $G'$ is 3-well-connected, since every vertex in $G'$ has degree at least 3. However, $G$ is not 3-well-connected since vertex $e$ has degree 2.

![Subgraph Example](image)

$G = (V, E) \quad G' = (S, E')$, for $S = \{a, b, c, d\}$

(a) (4 pt.) Design a greedy algorithm to find a set $S \subseteq V$ of maximum size so that the subgraph $G' = (S, E')$ induced by $S$ is $k$-well-connected. In the example above, if $k = 3$, your algorithm should return $\{a, b, c, d\}$, and if $k = 4$ your algorithm should return the empty set.

You may assume that your representation of a graph supports the following operations:

- $\text{degree}(v)$: return the degree of a vertex in time $O(1)$
- $\text{remove}(v)$: remove a vertex and all edges connected to that vertex from the graph, in time $O(\text{degree}(v))$.

Your algorithm should run in time $O(n^2)$.

[We are expecting:

- Pseudocode AND an English description of what your algorithm is doing.
]

(b) (4 pt.) Prove that your algorithm is correct.

[We are expecting: A formal proof. If you use induction, be sure to clearly state your inductive hypothesis, base case, inductive step, and conclusion.]
5. (NOT REQUIRED, WORTH ONE BONUS pt.) [Another activity selection algorithm?]
Consider the following alternative greedy algorithm for activity selection. The idea is that at each step, we greedily add a valid activity with the fewest conflicts with other valid activities. (An activity is *valid* if it doesn’t conflict with an already selected activity).

For example, if the activities looked like:

```
0   3   3   3   3   3   1   1   2
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
```

then the number of conflicts to begin with are:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a_1</td>
<td>a_2</td>
<td>a_3</td>
<td>a_4</td>
<td>a_5</td>
<td>a_6</td>
<td>a_7</td>
<td>a_8</td>
</tr>
</tbody>
</table>
```

The algorithm (breaking ties arbitrarily) could choose \( a_1 \), then \( a_6 \), then \( a_7 \), then \( a_2 \).

Is this algorithm correct?

[**We are expecting:** To get the bonus point, give either a counterexample or a formal proof of correctness.]

- **If you give a proof by induction, make sure to clearly state your inductive hypothesis, base case, inductive step and conclusion. (Note, in this case you should show that the algorithm is correct no matter how it breaks ties).**
- **If you give a counterexample, it should be a drawing like the one above; you can either draw it by hand or use your favorite software. You should also explain what this algorithm does on your counter-example and why it is not optimal. (Note, in this case it is okay to give an example where there is some way of breaking ties so that the algorithm messes up).**

**Feedback**

There’s no “correct” answer here, and it is completely anonymous.

6. (1 pt.) Please fill out the following poll which asks about your comfort with topics in the class so far.

https://forms.gle/fw3RuRUoCJMrcysdA

Did you fill out the poll?

[**We are expecting:** The answer “yes.”]
Yet another problem

This problem set is long enough, but it’s always good to get more practice with dynamic programming! If you want more DP practice, you can try the problem below. It’s not worth any points (not even extra credit), and we won’t grade it, but it might be fun to try.

7. (0 pt.) [Fish fish eat eat fish.] Plucky the Pedantic Penguin enjoys fish, and 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 Plucky cannot pay, Plucky 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

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

Then Plucky might do:

<table>
<thead>
<tr>
<th>Day 0</th>
<th>Lake A</th>
<th>Lake B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day 0</td>
<td>0 fish</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+5</td>
<td>-3</td>
</tr>
<tr>
<td>Day 1</td>
<td>5 fish</td>
<td>2 fish</td>
</tr>
<tr>
<td></td>
<td>+7</td>
<td>+4</td>
</tr>
<tr>
<td>Day 2</td>
<td></td>
<td>13 fish</td>
</tr>
</tbody>
</table>

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.

(Questions on next page)
(a) (0 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) (0 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: Pseudocode AND a short English description of what it does and why it works, and a justification of why it runs in time \( O(n) \).]