CS 161	**Problem Set 7**
Winter 2021	No Due Date: Optional Problem Set

Note: Starting from Problem 4 in this problem set are all DP problems. Since Exam 3 only covers up to lecture 12, we do not expect you to be able to answer general DP problems like the ones here in Exam 3.

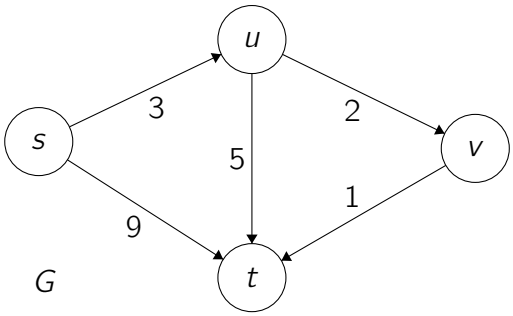
Exercises

1. In class, we saw pseudocode for Dijkstra's algorithm which returned shortest distances but not shortest paths. In this exercise we'll see how to adapt it to return shortest paths. One way to do that is shown in the pseudocode below:

```
Dijkstra_st_path(G, s, t):
    for all v in V, set d[v] = Infinity
    for all v in V, set p[v] = None
    // we will use the information p[v] to reconstruct the path at the end.
    d[s] = 0
    F = V
    D = []  // D is the list of "done" vertices
    while F isn't empty:
        x = a vertex v in F so that d[v] is minimal
        for y in x.outgoing_neighbors:
            d[y] = min( d[y], d[x] + weight(x,y) )
            if d[y] was changed in the previous line, set p[y] = x
        F.remove(x)
        D.add(x)

    // use the information in p to reconstruct the shortest path:
    path = [t]
    current = t
    while current != s:
        current = p[current]
        add current to the front of the path
    return path, d[t]
```

Step through `Dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below (on the next page) to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is. If it is helpful, the LaTeXcode for the table is reproduced at the end of the PSET.



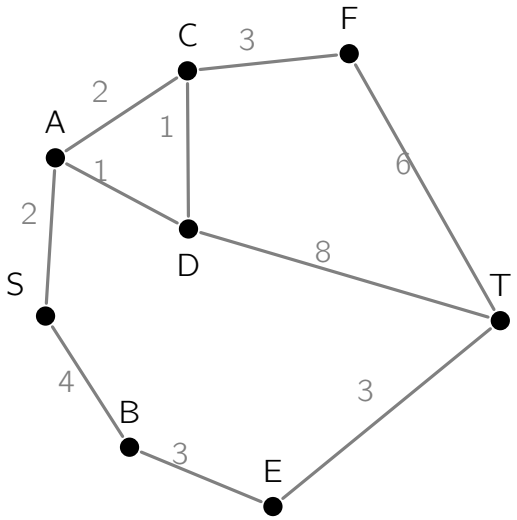
[**We are expecting:** The following things:

- The table below filled out
- The shortest path and its cost that the algorithm returns.

No justification is required.]

	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	9	None	s	None	s
Immediately after the second element of D is added, the state is:								
Immediately after the third element of D is added, the state is:								
Immediately after the fourth element of D is added, the state is:								

2. [A* Search Algorithm.]



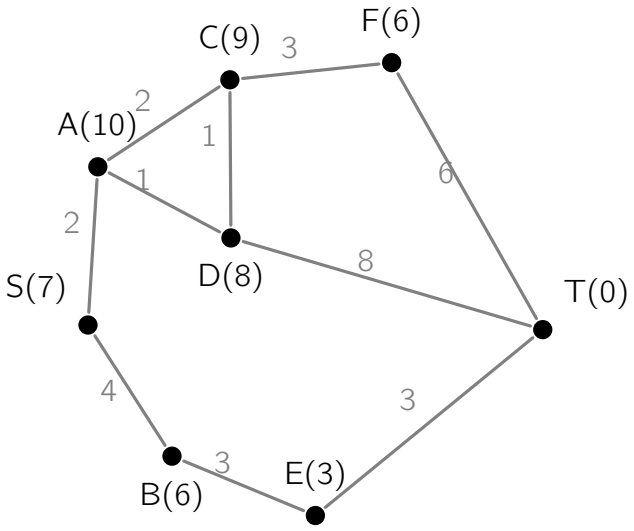
Consider the above undirected weighted graph $G = (V, E)$.

(a) Suppose we run Dijkstra's algorithm from class to find the minimum-weight path from S to T starting from node S. List the nodes visited by the Dijkstra's algorithm in the order of when they are first marked "sure" by the algorithm, breaking any tie by alphabetical order.

Note that the algorithm should stop when node T is marked "sure".

[**We are expecting:** A list of vertices from G that are visited by the Dijkstra's algorithm in the order of when they are first marked "sure".]

(b) In the above run of the Dijkstra's algorithm, we visited all nodes in the graph before finding out the shortest path between S and T. Suppose we want to make the algorithm even more efficient by visiting less nodes. Luckily, at each vertex v in G, we have a rough estimate $h(v)$ of how far it is from vertex v to vertex T. The value of $h(v)$ is denoted near each vertex in the following graph. Recall that in Dijkstra's algorithm, we use our estimate of the distance from S to v, denoted $d[v]$ to select the 'not-sure' node to visit in each iteration. Now with our rough estimates of the distances to T, we modify the algorithm such that we use $d^*[v] = d[v] + h[v]$ instead of $d[v]$ to choose the next 'not-sure' node. Rerun this modified Dijkstra's algorithm to search for shortest $S - T$ paths starting from node S, and give a list of vertices visited by the algorithm in the order of when they are first marked "sure".



[**We are expecting:** A list of vertices from G that are visited by the Dijkstra's algorithm in the order of when they are first marked "sure".]

Problems

---

3. **[Negative Dijkstra's Algorithm.]** It turns out Dijkstra's algorithm only works for graphs with nonnegative edge weights. In this question, we'll explore why this is the case. For this question assume that the graph is directed and that all edge weights are integers.

   (a) A "negative cycle" is a cycle where the sum of the edge weights is negative. Why can't we compute shortest paths in a graph with negative cycles?

   [**We are expecting:** An informal explanation.]

   (b) Even if a graph contains no negative cycles (but still contains negative edges) we might still be in trouble. Please draw a graph G, which contains both positive and negative edges but does not contain negative cycles, and specify some source $s \in V$ where `Dijkstra(G, s)` does not correctly compute the shortest paths from s.

   [**We are expecting:** A graph G and a brief explanation on the shortest path that is not correctly computed.]

   (c) Consider the algorithm `Negative-Dijkstra` for computing shortest paths through graphs with negative edge weights (but without negative cycles). This algorithm adds some number to all of the edge weights to make them all nonnegative, then runs `Dijkstra` on the resulting graph, and argues that the shortest paths in the new graph are the same as the shortest paths in the old graph.

```
Negative-Dijkstra(G, s):
    w* = minimum edge weight in G
    for v in V do
        for w in v.neighbors do
            w'(e) = w(e) - w*
    G' = G with weights w'
    T = Dijkstra(G', s) // run Dijkstra with w' to get a SSSP Tree
    update T to use weights w that corresponds to graph G
    return T
```

   (Note that an "SSSP tree", or a "single-source-shortest-path tree", is analogous to a breadth-first-search tree in that paths in the SSSP tree correspond to shortest paths in the graph. Here we assume that Dijkstra's algorithm has been modified to output such a tree.)

   Prove or disprove: Negative-Dijkstra *always* computes single-source shortest paths correctly in graphs with negative edge weights.

   [**We are expecting:** To prove the algorithm correct, show that for all $u \in V$,

a shortest path from s to u in the original graph lies in T. To disprove the algorithm, exhibit a graph with negative edges, but no negative cycles, where Negative-Dijkstra outputs the wrong "shortest" paths, and explain why the algorithm fails.]

4. **[Longest Increasing Subsequence.]** 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 $A[i_0] < A[i_1] < \cdots < A[i_{\ell-1}]$ **and** that ℓ 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.

   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!

   (a) **(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

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

   [**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) **(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:** Pseudocode of your algorithm. No justification is required.]

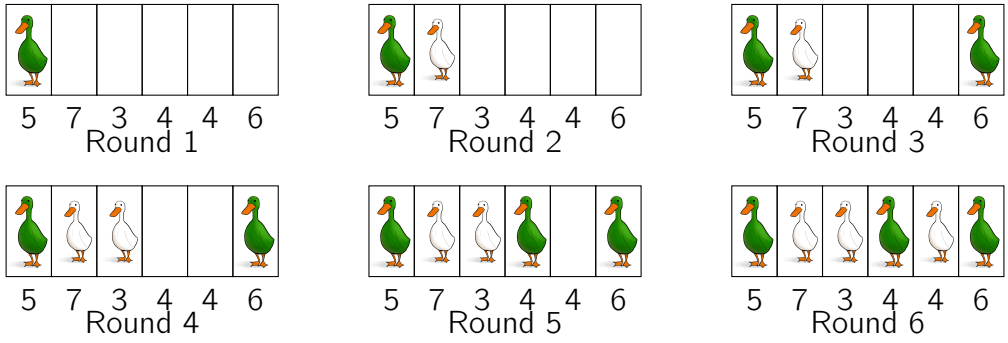
   (c) **(Adapt your DP algorithm to return the optimal solution)** Adapt your algorithm above to return an actual LIS instead of its length. Your algorithm should run in time $O(n^2)$.

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

5. [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 troupe 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 $D = [5, 7, 3, 4, 4, 6]$.



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) 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) 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).]

6. [MinElementSum.] 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 (possibly with repetitions). 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) Argue that for $S = \{1, 2\}$, your friend's algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$). You may assume that Fibonacci numbers grow exponentially, i.e., let $f(n)$ be a function that returns n-th Fibonacci number, you may assume that $f(n) = 2^{\Omega(n)}$.

[**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) 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 sub-problem repeatedly.

[**We are expecting:**

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

]

(c) 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.

]

7. [Fish fish eat eat fish.] Plucky the Pedantic Penguin enjoys fish, and it has 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 it can catch in Lake A on day i, and $B[i]$ is the number of fish it can catch in Lake B on day i, for $i = 0, \ldots, n-1$.

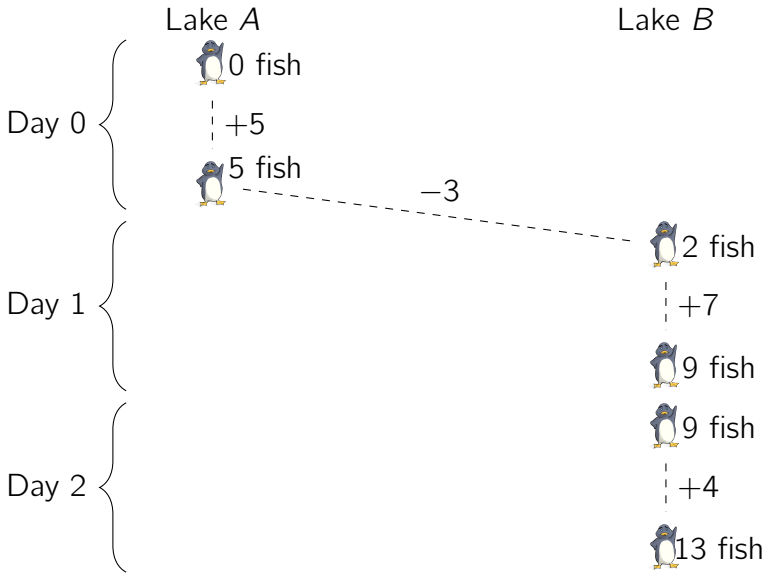
If Plucky is at Lake A on day i and wants to be at Lake B on day $i+1$, it may pay L fish to a polar bear who can take it from Lake A to Lake B overnight; the same is true if it wants to go from Lake B back to Lake A. The polar bear does not accept credit, so **Plucky must pay** *before* **it travels**.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

Assume that when day 0 begins, Plucky is at Lake A, and it has 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) 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) 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 $O(n)$.]