0. [1 point, only if all parts are completed]

(a) Submit your homework to Gradescope.
(b) Student id is the same as on T-square: it’s the one with alphabet + digits, NOT the 9 digit number from student card.
(c) Pages for each question are separated correctly.
(d) Words on the scan are clearly readable.

Updates to this homework since version 0 are:

• Question 1, changed in v1 to explicitly specify that you only need to find the length of the subsequence, not the subsequence itself.

1. (5 points) Longest alternating subsequence. Recall the longest increasing subsequence problem. Now instead we want to find the maximum length subsequence that alternates increasing/decreasing, starting with increasing. That is, we want to find a sequence of indices \(i_1 < i_2 < \ldots < i_k\) such that

- For each \(1 \leq j < k\) with \(j\) odd, \(A[i_j] < A[i_{j+1}]\), and
- For each \(1 \leq j < k\) with \(j\) even, \(A[i_j] > A[i_{j+1}]\).

(a) (3 points) Given an \(O(n^2)\) time algorithm for finding the length of the longest alternating subsequence using dynamic programming.

SOLUTION:
States: two states per index,
• LongestUp\([i]\) longest subsequence ending at \(i\) where the last entry is bigger than the preceding.
• LongestDown\([i]\) longest subsequence ending at \(i\) where the last entry is smaller than the preceding.

Base case: LongestUp\([i]\), LongestDown\([i]\) ≥ 1.

Transition:
\[
\text{LongestUp}[i] = \max_{j < i, A[j] < A[i]} \text{LongestDown}[j] + 1
\]
\[
\text{LongestDown}[i] = \max_{j < i, A[j] > A[i]} \text{LongestUp}[j] + 1
\]

These values can also be viewed as breaking down the states by the parity of subsequence lengths. LongestDown\([i]\) is the length of the longest odd lengthed alternating subsequence ending at \(i\), and LongestUp\([i]\) is the length of the longest even lengthed alternating subsequence ending at \(i\).

SOLUTION:
It’s also possible to do this by having 1 state per index.
We can directly transit on LongestDown\([i]\) by doing
\[
\text{LongestDown}[i] = \max_{j < k < i, A[j] < A[k], A[k] > A[i]} \text{LongestDown}[j] + 2,
\]
but this takes \(O(n^3)\) time, and one also need to take the even total length one into account by also outputting the solution
\[
\]

SOLUTION:
As it turns out, greedy even works, and runs in \(O(n)\) time!
Consider maintaining an alternating subsequence.
If the length is even, and we have an element that’s less than the last one, we append it to the sequence.
Otherwise, we have an element that’s bigger or equal. In that case we simply replace the last element with it. This works because the even length means we have \(A[i_j] > A[i_{j-1}]\), and increasing the value of \(A[i_j]\) would still meet this condition.

SOLUTION:
Another way to view the greedy solution is that it essentially goes through the array \(A\) and removes all ‘non-alternating triples’ \(i\) such that
\[
A[i - 1] \leq A[i] \leq A[i + 1]
\]
or
\[ A[i - 1] \geq A[i] \geq A[i + 1]. \]

**SOLUTION:**
Due to the above solution, it’s also possible to prove the following seemingly incorrect dynamic program that just stores the values of \( LAS[i] \) is correct:
State: \( LAS[i] \) is the max length of an alternating subsequence ending at \( i \).
Base case: \( LAS[i] \geq 1 \).
Transition:
\[
\]

**GRADING:**
1 point for justification of algorithm.
−0.5 point for an \( O(n^3) \) time algorithm.
2 points for a correct algorithm (any of the ones listed above), even if the justification are iffy.

(b) (2 points) Show that the problem of finding the longest alternating subsequence can be formulated and solved as a shortest path on \( 2n \) vertices.

**SOLUTION:**
We can create two vertices per index, corresponding to the two states from the first solution to part a).
Then for any \( j < i \) with \( A[j] < A[i] \), we add an edge from \( LongestDown[j] \) to \( LongestUp[i] \) with weight \(-1\), and for any \( j < i \) with \( A[j] > A[i] \), we add an edge from \( LongestUp[j] \) to \( LongestDown[i] \) with weight \(-1\).
We also add a super source with edges of weight 0 to all vertices, and a super sink where all vertices have edges of weight 0 going to.
Then the shortest path from the super source to super sink gives the negation of the length of the longest alternating sequence.

**GRADING:**
The requirement of exactly \( 2n \) vertices was not necessary: allow \( O(1) \) discrepancies in vertex counts.

2. (4 points) Minimum Spanning Tree
Consider the graph below.
(a) (2 points) Give a cut that shows edge $e = \{B, D\}$ is in an MST. Be sure to specify the cut via a set of vertices.

**SOLUTION:**

As $\{B, D\}$ is the minimum edge in the graph, any valid cut for this edge is a correct answer.

(b) (2 points) Show using the cut rule that if we increase the weights of some edges in $G$ to form $H$, then an edge $e$ in the minimum spanning tree in $G$ is in some minimum spanning tree of $H$ if its weights were not adjusted.

**SOLUTION:**

The proof is by cut property. If edge $e$ was placed in the original MST, there must exist some cut $C$ for which $e$ had the minimum weight. When some of the non-tree edges are modified, $e$ must still be the lowest weight edge across the cut.

3. (5 points) Bellman-Ford algorithm for finding shortest paths.

In class, we discussed the following variant of the Bellman-Ford algorithm, whose pseudocode is below.

```plaintext
While true {
    For all edges $e = u \rightarrow v$ {
        $dist[v] = \min(dist[v], dist[u] + l(u \rightarrow v))$;
    }
    if no vertex’s $dist$ value is changed, stop
}
```

The text book’s version (in Section 4.6.) instead fixes the iteration count at $\Theta(n)$. We also showed in class that if we do indeed get to such an iteration count, there must be a negative cycle.

In this question, we will discuss how many times the outer loop of Bellman-Ford algorithm actually needs to run. For such a discussion, it is important to fix an ordering by which we consider the edges, that is we view the edges as a fixed list, $e_1 \ldots e_m$.

(a) (1 point) For any $n$, exhibit a graph and an ordering of the edges where the Bellman-Ford algorithm takes more than $n/2$ steps.
SOLUTION:

Consider the following directed graph with 4 vertices (n = 4), starting node as 1 and distances to all the other nodes as infinity:

With the following edge order, it will take 3 (greater than 2 = n/2) iterations before the algorithm converges:

(3) → (4)
(2) → (3)
(1) → (2)

GRADING:
There is some ambiguity about whether this needs to work for every n (above some threshold), or just some particular n.
The more restrictive version of ‘for some n’ is considered acceptable.
Providing a graph with a negative cycle is also acceptable, since this version didn’t specify positive weighted edges.

(b) (2 points) Show that for any graph G without negative cycles and any s-t pair, there exists an ordering of edges so that dist[t] (as defined in the pseudocode above) becomes the value of the true s to t distance in 1 step.  

SOLUTION: 
For any s-t pair, consider the shortest path from s to t and process the edges in that order. That way, in the first iteration itself, dist[t] will be the ”true” s-t distance.

(c) (2 points) Consider the k × k square grid with two-way edges of positive lengths. We can label each vertex by its row/column number, with the top left corner being (1, 1), which is also our source vertex.
Then each edge is a 4 tuple, (r_1 ,c_1 ) → (r_2 ,c_2 ), and we can produce edge ordering by sorting lexicographically via these tuples. For example, on a 2 × 2 grid, this would order the edges
as:

\[(1, 1) \rightarrow (1, 2)\]
\[(1, 1) \rightarrow (2, 1)\]
\[(1, 2) \rightarrow (1, 1)\]
\[(1, 2) \rightarrow (2, 2)\]
\[(2, 1) \rightarrow (1, 1)\]
\[(2, 1) \rightarrow (2, 2)\]
\[(2, 2) \rightarrow (1, 2)\]
\[(2, 2) \rightarrow (2, 1)\]

Show that for any \(k\), there exists a set of positive weights for these edges of the \(k \times k\) square grid so that the Bellman-Ford algorithm with \((1, 1)\) as the starting point takes at least \(k/10\) iterations (the 10 was chosen as an attempt to remove the need of getting constructions with good constants).

**SOLUTION:**
Consider a graph where all the vertical edges have length 0, but all horizontal edges have length \(\infty\) (which if we want actual numbers, can be set to \(10k^2\)) except for the bottom ones in odd indexed columns, and top ones in even indexed columns having weight 1. Specifically, the edges with weight 0 are

\[(k, 1) \rightarrow (k, 2)\]
\[(1, 2) \rightarrow (1, 3)\]
\[(k, 3) \rightarrow (k, 4)\]
\[(1, 4) \rightarrow (1, 5)\]
\[\ldots\]

have weight 0.
Then the shortest path consists of moving downwards, rightwards, upwards, rightwards, downwards, and etc.
Each sequence of upwards propagation takes \(k\) iterations of the Bellman-Ford algorithm because the edges are updated in reverse ordering.
As there are at least \(k/2\) of these upward segments, the total number of iterations required is at least \(k^2/2\).

**GRADING:**
edge weights of 0 and \(\infty\) are acceptable.
4. (6 points) Coloring a graph

The minimum coloring problem asks to label the vertices with the fewest number of distinct colors so that no edge has two endpoints having the same color.

(a) (2 points) Consider the following greedy algorithm: find a maximum independent set, color it with color 1, remove it from the graph, and repeat (using colors \( \geq 2 \)). Show that this algorithm is suboptimal by providing a graph of size at most 5 where it does not produce the optimal coloring.

**SOLUTION:**

\[
\begin{array}{c}
A \\
B \\
D \\
E \\
F \\
\end{array}
\]

One possible max independent set is \(A\) and \(F\). Use one color for this. We now have \(B, D,\) and \(E\) left. This is a complete graph, so we’d need 3 colors, totaling in 4 colors. However, this graph can be colored in 3 colors, if \(A\) and \(E\) are colored, then \(B\) and \(F\) are colored, then finally \(D\).

(b) (3 points) Give an \(O(4^n)\) time algorithm for finding the fewest number of colors needed to color a graph. One way that this can be done is to define a state to be the minimum number of colors needed to color a subset of the vertices, and the transition can then be an entire subset of vertices that can be colored using the same color.

**SOLUTION:**

We define a subproblem \(C(S)\) as the minimum number of colors needed, where \(S\) is a subset of \(V\). We construct a table \(C\) and enumerate all \(2^n\) independent subsets of \(V\) in increasing order of size.

The base case is \(C(\emptyset) = 0\).

At each subset \(S_i\) where \(1 \leq i \leq 2^n\),

\[
C(S) = \min_{T \subseteq S} \{C(S \setminus T)\} + 1,
\]

where \(T\) is a non-empty independent subset of \(S\). This iterates over all \(2^n\) subsets of \(V\), and on each iteration, we select minimum over all \(2^n\) possible independent subsets of \(S\). So the upper bound is \(O(4^n)\).

(c) (1 points): Improve the running time to \(O(3^n)\).
SOLUTION:
The transition cost from Part (b) can be expressed as follows:

\[
\sum_{k=0}^{n} \binom{n}{k} 2^k = \sum_{k=0}^{n} \binom{n}{k} 2^k 1^{n-k} = (2 + 1)^n
\]  
(1)