This review section covers material up to and including dynamic programming (Quiz 2).

Problem 1. True or False
Answer True or False to the following questions:

(a) Every non-empty DAG has at least one source.
(b) Suppose we have a graph where each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in this graph.
(c) A dynamic programming problem where we need to solve $O(n^2)$ subproblems will take $O(n^2)$ time.

Solution

(a) True. Start at an arbitrary vertex $v$, and repeatedly go backwards in the graph. If there is no source, you can keep going forever. But that means some vertex will be visited twice, and in particular, you’ll have found a cycle in the graph - so it couldn’t have been a DAG in the first place.

(b) (Berkeley CS 170) False. Consider the following graph. The MST has weight 6, and can be any of $b - a - c - d - e - f$, $b - a - c - d - f - e$, $a - b - c - d - f - e$, or $a - b - c - d - e - f$. 

(c) False. Each subproblem may take more than $O(1)$ time to compute.

Problem 2. Big O Only
Is it possible to find two functions $f(n), g(n)$ such that $f(n) = O(g(n))$, but $f(n) \neq o(g(n)), f(n) \neq \Omega(g(n))$? If so, give an example. If not, give a proof.

Solution
The challenge here is to come up with 2 functions such that big O is the only relationship that holds between them. An example of such a pair of functions is $f(n) = n^{\sin n}$ and $g(n) = n$. It is clear that $f(n) \leq 1 \cdot g(n)$. It is also clear that $f(n) \neq o(g(n))$ because the limit of the ratio does not exist. We also have that $f(n) \neq \Omega(g(n))$ because $f(n)$ can be arbitrarily small when $\sin n = -1$ for example.

Problem 3. Cyclic Sorted Array
You’re given an array $A$ of $n$ distinct numbers that is sorted up to a cyclic shift, i.e. $A = (a_1, a_2, \ldots, a_n)$
where there exists a $k$ such that $a_k < a_{k+1} < \ldots < a_n < a_1 < \ldots < a_{k-1}$, but you don’t know what $k$ is. Give an efficient algorithm to find the largest number in this array.

Solution
Here’s a recursive solution based on binary search that runs in $O(\log n)$ time.

First, we check that the array isn’t cycled at all if $a_n > a_0$, then we can return the last element (which will be the greatest), $a_n$. Otherwise, $a_0 > a_n$, and start at the middle element $a_m$ of the entire array. If $a_m > a_0$, then move to the right half of the array and recurse. Otherwise, if $a_m < a_0$, move to the left half of the array and recurse.

Problem 4. Counting Permutations
How many permutations of $\{1, 2, \ldots, N\}$ have exactly $K$ inversions? An inversion is a pair of numbers $i, j$ in the permutation such that $i > j$ but $i$ comes before $j$. For example, for $N = 3$ and $K = 1$ the answer is 2: $\{2, 1, 3\}$ and $\{1, 3, 2\}$.

Solution
We use dynamic programming on $N$ and $K$, and at each step, we decide where the smallest number goes.

- **Definition**: Define $X[i, j]$ to be the number of permutations of $\{1, 2, 3, \ldots, i\}$ that have exactly $j$ inversions. Our desired answer is $X[N, K]$.

- **Recurrence**: Consider the element $i$. Given any permutation of $1, 2, \ldots, i - 1$, we can insert $i$ into any of the $i$ slots (i.e. before 1, between 1 and 2, between 2 and 3... after $i - 1$). Inserting all the way at the beginning causes $i - 1$ inversions, while inserting all the way at the end incurs 0 inversions. Therefore, we have:

$$X[i, j] = \begin{cases} 0 & j < 0 \\ 1 & i = 0, j = 0 \\ \sum_{\ell=0}^{i-1} X[i-1, j-\ell] & \text{otherwise} \end{cases}$$

- **Analysis**: The run-time is $O(N^2K)$ because there are $O(NK)$ states of the DP and each takes $O(N)$ time to compute. The space complexity is naively $O(NK)$, but since we are only interested in the count of the number of permutations and not the permutations themselves, we can get away with $O(K)$ space by only storing each row $X[\cdot, K]$ of the table at a time.

Problem 5. Dijkstra’s Variations
In lecture, we showed that Dijkstra’s algorithm may not work correctly if negative edge weights are present. In this problem. Let $G = (V, E)$ be a weighted, directed graph with no negative-weight cycles. For each of the following property of $G$, give an algorithm that finds the length the shortest path from $s$ to all other vertices in $V$. In both cases, your algorithm should have the same run-time as that of Dijkstra’s.

(a) $G$ only has one negative edge. Give an algorithm to find length of the shortest path from $s$ to all other vertices in $V$.

(b) The edges leaving from the source $s$ may have negative weights, but all other edges in $E$ have non-negative weights,
Solution

(a) Let’s say the negative-weight edge is \((u, v)\). First, remove the edge and run Dijkstra from \(s\). Then, check if \(d_s[u] + w(u, v) < d_s[v]\). If not, then we’re done because it implies that the shortest path tree does not use this negative edge.

If yes, then run Dijkstra from \(v\), with the negative-weight edge still removed. Let the distances obtained by that be \(d_v[\cdot]\). Then, for any node \(t\), its shortest distance from \(s\) will be \(\min(d_s[t], d_s[u] + w(u, v) + d_v[t])\). This works because the shortest path to any node is either that where you use the negative edge or you do not. If you do use the negative edge, then the shortest path consists of the shortest path to \(u\), using \((u, v)\) and then the shortest path from \(v\) to \(t\).

(b) The claim is that running Dijkstra’s as usual will work. Let’s compare the execution of Dijkstra’s algorithm on \(G\) to the execution on a graph \(G'\) where we’ve added the same amount \(m\) to the weights of all edges coming out of the source so as to make them non-negative.

On the one hand, since all edge weights in \(G'\) are non-negative, we know that Dijkstra’s algorithm will find the correct shortest-path distances in \(G'\).

On the other hand, observe that the flow of Dijkstra’s algorithm is controlled by two things: the comparisons \(dist[w] > dist[v] + length[v, w]\), and the element with lowest priority in the heap.

Observe that at any step in Dijkstra’s algorithm, in the comparison \(dist[w] > dist[v] + length[v, w]\) we’re actually comparing the lengths of two paths from \(s\) \(w\). Whenever \(w \neq s\), both of these paths contain exactly one edge coming out of \(s\). So the truth value of the comparison will be the same in \(G\) and \(G'\) because the difference is \(m\) on both sides of the comparison. Moreover, once we pop \(s\) from the queue, the queue for \(G'\) will be a copy of the queue for \(G\), but with all priorities shifted up by \(m\). This preserves the minimal element.

It follows that Dijkstra’s algorithm is the same on \(G\) and \(G'\), up to the priorities of all vertices \(v \neq s\) shifted by \(m\). Thus, Dijkstra’s algorithm on \(G\) will find the shortest paths in \(G'\). But the shortest paths in \(G'\) to all \(v \neq s\) are the same as those in \(G\) by an argument similar to the above: if we have a path \(p: s \rightarrow v\) which is shortest in \(G'\), this means that \(l'(p) \leq l'(q)\) for any \(q: s \rightarrow v\), where \(l'\) is distance calculated in \(G'\), and \(l\) is distance calculated in \(G\). But then, \(l(p) = l'(p) - m \leq l'(q) - m = l(q)\) for all such \(q\) in \(G\), hence \(p\) is a shortest path in \(G\).

Finally, it remains to show that we find the right distance to \(s\) itself in \(G\) - which is clear because we assumed \(G\) has no negative cycles, so the true distance is 0, and this is what Dijkstra’s algorithm gives.

Problem 6. Largest Submatrix of 1’s

You are given a matrix \(M\) with \(m\) rows and \(n\) columns consisting of only 0’s and 1’s. Give an efficient algorithm for find the maximum size square sub-matrix consisting of only 1’s.

Solution

We will solve this via dynamic programming:

- **Definition**: Let \(S[i, j]\) be the size of largest square of 1’s submatrix containing \(M[i, j]\) in the
right-most and bottom-most corner. Note that if $M[i, j] = 0$, then $S[i, j] = 0$. The desired answer is $\max_{i,j} S[i, j]$.

- **Recurrence**: We can then copy the first row and column from $M$ to $S$ because at most it will be a square of size $1 \times 1$. This is equivalent to saying that if $i = 1$ or $j = 1$, then we have that $S[i, j] = M[i, j]$. For the recursive case, we can then fill $S$ according to

$$S[i, j] = \begin{cases} 
\min (S[i, j-1], S[i-1, j], S[i-1, j-1]) + 1 : M[i, j] = 1 \\
0 : M[i, j] = 0 
\end{cases}$$

It is pretty easy to see why this is correct after drawing an example.

- **Analysis**: This will take $O(mn)$ time and $O(m+n)$ space since we only need to keep the last two diagonals.

**Problem 7. Minimum Bottleneck Spanning Tree**

Let $G = (V, E)$ be an undirected graph with weights $w_e$ on the edges. A **minimum bottleneck spanning tree** of $G$ is a spanning tree such that the weight of the heaviest edge is as small as possible. Note that for minimum bottleneck spanning trees, we do not care about the sum of the weights of the chosen edges—only the maximum edge.

(a) Show that any minimum spanning tree is necessarily a minimum bottleneck spanning tree

(b) Give an example of a graph and a minimum bottleneck spanning tree that is not a minimum spanning tree.

**Solution**

(a) Assume that there is an MST $T$ of a graph $G$. Let $T'$ be an MBST of $G$. Define $b(T)$ and $b(T')$ to be the weight of the bottleneck edge in $T$ and $T'$. Let $e$ be the bottleneck edge of $T$. Consider the cut in the tree defined by $e$. By the cut property every other edge crossing the cut necessarily has weight at least that of $e$. Now look at the edges of $T'$ which cross the cut. By the above statement they must too all have weight at least that of $c(e)$. But then if $b(T)$ is the bottleneck cost of a tree $T$, $b(T) = c(e)$. Thus, we know that $b(T) \leq$ the cost of any edge of $T'$ crossing the cut $\leq b(T')$. Therefore, we have shown that $b(T) \leq b(T')$. We also know that by definition of minimum bottleneck spanning tree that $b(T') \leq b(T)$. Therefore, $b(T') = b(T)$ and $T$, a MST, is also an MBST.

\[ \leq \text{cost of any edge of } T_0 \text{ crossing the cut} \leq b(T_0) : b(T) \leq b(T_0), \text{ and so since } T_0 \text{ has minimum bottleneck cost among trees } (b(T_0)) \leq b(T)), b(T) = b(T_0) - T \text{ is an MBST.} \]

(b) Consider the following graph:
Notice that the minimum spanning tree has weight 4 and has bottle-neck edge 2. However, \( d - a - b - c \) is a minimum bottle neck spanning tree because it has heaviest edge 2 and it is impossible to find any other spanning trees that have a lighter bottle neck edge.

Problem 8. Well-Nested Brackets
There are four types of brackets: (, ), <, and >. We define what it means for a string made up of these four characters to be well-nested in the following way:

(a) The empty string is well-nested.
(b) If A is well-nested, then so are \(<A>\) and \(A\).
(c) If S, T are both well-nested, then so is their concatenation ST.

For example, (), <>, (()), ()<>, and ()<> are all well-nested. Meanwhile, (, <, ), )(, (<)>, and (<>) are not well-nested.

Devise an algorithm that takes as input a string \(s\) of length \(n\) made up of these four types of characters. The output should be the length of the shortest well-nested string that contains \(s\) as a subsequence (not sub-string). For example, if ()<> is the input, then the answer is 6; a shortest string containing \(s\) as a subsequence is ()><.

Solution
This problem can be solved with dynamic programming.

- **Definition:** Let \(s[i, j]\) be the substring of \(s\) from the \(i\)th to the \((j-1)\)th character, inclusive. Then, we define the sub problem \(f(i, j)\) to be the length of the shortest well-nested string containing \(s[i, j]\). Our desired output is \(f(0, n)\).

- **Recurrence:** The base case is where the length of the substring is 1 or 0. If \(i = j\), the substring has length 0, and \(f(i, j) = 0\).
If \(i = j + 1\), the substring has length 1, and we must add one character to close it, so \(f(i, j) = 2\).
To compute \(f(i, j)\) recursively, we first note that if \(s[i]\) is a closing bracket, ) or >, we must insert an matching opening bracket to achieve well-nestedness. Otherwise, the opening bracket \(s[i]\) must be matched by some closing bracket \(s[k]\) where \(i < k < j\), or by adding a closing bracket. Let the matching closing bracket of of \(s[i]\) be denoted by \(s[i]'\).
The analysis above give the following recurrence:

\[
f(i, j) = \begin{cases} 
2 + f(i + 1, j) & s[i] = ), > \\
\min(2 + f(i + 1, j), \min_{s[k]=s[i]}(f(i + 1, k) + f(k + 1, j))) & s[i] = (, < 
\end{cases}
\]

- **Analysis** The runtime is \(O(n^3)\) since there are \(O(n^2)\) values to calculate, each of which takes \(O(n)\) time (from scanning the substring for the matching bracket of \(s[i]\)). The space of the algorithm is \(O(n^2)\) since the array is \(n + 1\) by \(n + 1\).

**Problem 9. Efficient Waiters**

A restaurant has \(n\) tables and 2 waiters. Table \(i\) is located at coordinate \((x_i, y_i)\) in the restaurant for \(i = 1, 2, \ldots, n\). At the beginning, one waiter is at table 1 and the other waiter is at table 2. We will then receive a sequence of \(m\) queries. Each query is simply a number in \(\{1, 2, 3, \ldots, n\}\) representing that the customer at that table needs some kind of service. Hence, one of the two waiters needs to be service that table. The waiters are complaining of walking too much and so the restaurant has hired you to help them be more efficient.

Suppose an omniscient being has told you in advance the order in which the tables would need servicing. Let \(t_1, t_2, \ldots, t_m\) be the sequence of queries (i.e. tables that need servicing). The goal is to minimize the total walking distance of the two waiters, where the distance between two tables is the standard Euclidean distance. Assume that there is enough time between the queries for whichever of the 2 waiters is chosen to walk to the table and service it.

(a) Consider the following greedy algorithm. In order to service table \(t_i\), we will have whichever waiter is closer to service that table. Give an example to show that this greedy algorithm is not optimal.

(b) Give a \(O(n^2m)\) time and \(O(n^2)\) space solution that finds the value of the shortest total walking distance.

**Solution**

(a) The intuition behind why the greedy algorithm is incorrect is that it might be better to move a waiter that is further away to service a table because the next query is close to where that waiter was originally. Here’s a simple example:

Let \((x_1, y_1) = (0, 0), (x_2, y_2) = (5, 0)\) and \((x_3, y_3) = (2, 0)\). Recall that waiter 1 starts at table 1 \((0, 0)\) and waiter 2 starts at table 2 \((5, 0)\). Suppose \(m = 2\) and there are 2 queries: \(t_1 = 3\) and \(t_2 = 1\).

The greedy algorithm would move waiter 1 to table 3 first to service that query. This has a distance of 2. Then, when table 1 is queried, the greedy algorithm will move waiter 1 back to table 1, costing a distance of 2. The total distance moved of the two waiters is 4. A more optimal solution would have been to just move waiter 2 to table 3 at the beginning. Then, when table 1 needs servicing, nothing needs to be moved at all!

(b) We will use dynamic programming:

- **Definition:** Let \(f(k, a, b)\) be the smallest possible walking distance for servicing requests
$t_k, t_{k+1}, \ldots, t_m$ given that waiter 1 is at table $a$ and waiter 2 is at table $b$. The final answer is $f(1, 1, 2)$.

- **Recurrence:** When we receive request $t_k$, we must choose one of two things to do: move waiter 1 there or waiter 2 there. Once that has happened, we have incurred some cost, but then we are in another state of the same problem. Let $d(i,j)$ be the Euclidean distance between $(x_i, y_i)$ and $(x_j, y_j)$.

\[
f(k; a, b) = \begin{cases} 
0 & k > m \\
\min \left[ f(k+1, t_k, b) + d(a, t_k), f(k+1, a, t_k) + d(b, t_k) \right] & k \leq m
\end{cases}
\]

- There are $O(n^2m)$ states to the DP and each takes $O(1)$ time to compute. Therefore, the run-time is $O(n^2m)$. The space complexity is $O(n^2)$ because we each $f(k, \cdot, \cdot)$ only depends on $f(k-1, \cdot, \cdot)$ and so we only need to store one row at a time.

**Problem 10. Significant Inversions (Kleinberg-Tardos 5.2)**

Call a significant inversion in an array $a$ a pair $i < j$ such that $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions in an array.

**Solution**

This is very similar to GOBOSORT’s inversion counting. We can use the same mergesort-like algorithm, only at each recursive level we perform two merges instead of one. Suppose we have the left half $\ell$ and right half $r$ at a current recursive step. Then let $r'$ be the same list as $r$ except with each element doubled; merging $\ell$ and $r'$ will allow us to count the number of significant inversions (see pset1 solutions for details). Then we merge $\ell$ and $r$ and continue as usual.

This uses the same recurrence relation as GOBOSORT and mergesort, because two merges still takes $O(n)$ time. Thus this algorithm takes $O(n \log n)$ time.