# Problem Set 9 Solutions

This problem set is due **at 11:59pm** on **Wednesday, May 3rd, 2017.**

# EXERCISES (NOT TO BE TURNED IN)

**Dynamic Programming**

- Do Exercise 15.1-4 in CLRS on page 370.
- Do Exercise 15.2-4 in CLRS on page 378.
- Do Exercise 15.4-5 in CLRS on page 397.

**Problem 9-1. Diamonds and Pearls** [50 points]
You are a diamond and pearl collector who travels on an $m \times n$ rectangular grid. You start at your home, which is the upper-left corner of the grid, and you can only travel one square to the right or one square downwards at each step. Each square of the grid has some number of diamonds, which you collect as you pass through the square, and the bottom-right corner has a pearl. You want to collect the pearl while maximizing the number of diamonds you collect along the way.

Formally, you are given an $m \times n$ 2D-array $A$, where each entry $A_{i,j}$ is a non-negative integer that represents the number of diamonds in square $(i, j)$, which is in row $i$ and column $j$. You start at the square $(1, 1)$, and you travel one square at a time to the right or downwards to reach the square $(m, n)$. Each time you travel through a square, you collect all the diamonds in that square. Your goal is to travel from the upper-left corner to the bottom-right corner while maximizing the number of diamonds you collect along the way.

**(a)** [15 points] Given $A$, compute the maximum number of diamonds you can collect as you travel from the upper-left to the bottom-right. Analyze the runtime of your solution.

**Solution:** This can be solved using dynamic programming. The subproblems are defined as
$C[i][j] :=$ the maximum number of diamonds you can collect in a path going from the square $(1, 1)$ to the square $(i, j)$.
and the update rule is

$$C[i][j] = \max(C[i-1][j], C[i][j-1]) + A_{i,j}$$

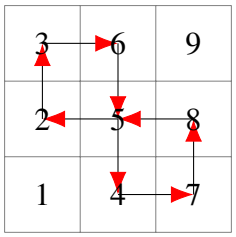
because the optimal way to get to $(i, j)$ must go through $(i-1, j)$ or $(i, j-1)$.
There are $O(mn)$ subproblems and each update takes $O(1)$ so the runtime is $O(mn)$.

Now, you want to collect the pearl and then deliver it back to your home. That is, you want to travel from the upper-left corner to the bottom-right corner while only moving to the right or downwards, and then you want to travel from the bottom-right corner back to the upper-left corner while only moving to the left or upwards. You still want to maximize the number of diamonds you collect along the way.

In particular, note that

1. You can (and should) collect diamonds on the way back as well
2. If you passed through a square before, there will be no more diamonds left in that square to collect if you pass through it a second time

For example, if you took the there-and-back-again path below, you would get a total of $35$ diamonds.




**(b)** [10 points] You come up with the following algorithm, which makes use of an algorithm $ALG$ that solves part (a).

1. Use $ALG$ to find the optimal path in $A$ to travel from the upper-left corner to the bottom-right corner. Call this path $P_1$.
2. Create a new 2D-array $A'$ that is equivalent to $A$, except with $0$'s in the squares along the path $P_1$.
3. Use $ALG$ to find an optimal path in $A'$ from the upper-left to the bottom-right (an optimal a path from the upper-left to the bottom-right is equivalent to an optimal path from the bottom-right to the upper-left). Call this path $P_2$.
4. Return $P_1 + P_2$.

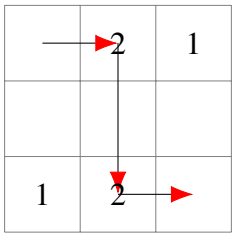
You may assume that $ALG$ works properly, whether or not you solved part (a).

Provide a counter-example showing that your new algorithm does not always result in the optimal there-and-back-again path.

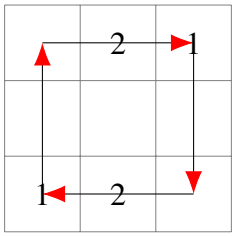
**Solution:**
Consider the following array, where the blank spaces correspond to squares with no diamonds.

| | | |
|---|---|---|
| | 2 | 1 |
| | | |
| 1 | 2 | |

In step 1, $ALG$ will find the path




and then the overall algorithm will collect one of the two 1's on the way back, giving a total of 5 diamonds. However, the optimal back-and-forth path is




and gives a total of 6 diamonds.

**(c)** [25 points] Given the matrix $A$, compute the maximum number of diamonds you can collect as you travel from the upper-left to the bottom-right then back to the upper-left. Analyze the runtime of your solution.
*Hint: Consider what happens if the path to the bottom-right and the path back to the upper-left cross. Can you simplify the paths so that they never cross?*
**Solution:**

The key observations are as follows.

1. Going from upper-left to bottom-right to upper-left is equivalent to having two paths going from upper-left to bottom-right
2. Paths that cross over each other can be replaced with paths that don't cross over each other
3. Paths should not intersect, except at the upper-left and bottom-right corners

In particular, the 3rd observation tell us that there exists some optimal solution such that the two paths only intersect at the upper-left and bottom-right corners, not that every optimal solution is of that formal. The observation does tell us that it is enough to find an optimal solution for the case of two non-intersecting paths, so we will only consider those.

Based on the above observations, we can then define the subproblem as
$C[i][j][r] :=$ the maximum number of coins you can get in two non-intersecting paths that both start from the square $(1, 1)$ and end up at the squares $(r, i)$ and $(r, j)$. These subproblems are only defined for $i < j$.

Our base case is technically just the value $C[1][1][1] = A_{1,1}$, and here $i = j$, but we can notice that some optimal path will definitely move the rightmost path to the right on the first step, so $C[1][2][1] = C[1][1][1] + A_{1,2}$. Similarly, we ultimately want to find the value $C[n][n][m]$, but we know that $C[n][n][m] = C[n-1][n][m] + A_{m,n}$ for the same reason.

The DP updates happen in three steps

1. Set $C[i][j][r] = C[i][j][r-1] + A_{r,i} + A_{r,j}$
2. Let $j$ range from 2 to $n$. For every fixed value of $j$, let $i$ range from $i$ to $j-1$. Set $C[i][j][r] = \max(C[i][j][r], C[i-1][j][r] + A_{r,i})$
3. Let $i$ range from 1 to $n-1$. For every fixed value of $i$, let $j$ range from $i+1$ to $n$. Set $C[i][j][r] = \max(C[i][j][r], C[i][j-1][r] + A_{r,j})$

These steps need to all be done separately, otherwise double-counting of certain squares might occur. For example, consider the update rule that updates everything at once

$$C[i][j][r] = \max(C[i][j][r-1] + A_{r,i} + A_{r,j}, C[i-1][j][r] + A_{r,i}, C[i][j-1][r] + A_{r,j})$$

It could add the value of certain squares multiple times. For example, $C[2][3][2]$ could have been computed as follows:

$$\begin{aligned} C[1][2][2] &= C[1][2][1] + A_{2,1} + A_{2,2} \\ C[1][3][2] &= C[1][2][2] + A_{2,3} \\ C[2][3][2] &= C[1][3][2] + A_{2,2} \end{aligned}$$

and here we see that $C[2][3][2]$ has already added the value $A_{2,2}$ twice. Thus, we have to do the DP updates in a smart way to prevent this type of double-counting from occurring.

The DP algorithm presented above with updates in 3 steps avoids this double-counting. In step 2, it fixes a value for where the rightmost path first enters the row, and it allows the leftmost path to move to the right, but not beyond the point where the rightmost path enters the row. After it has considered all of those possible movements for the leftmost path, it allows the rightmost path to move right beyond where it entered the row.

The solution given achieves the updates for each row in $O(n^2)$ time, and does updates for each of the $m$ rows, giving a total running time of $O(mn^2)$. There's no reason we have to choose to look at rows instead of columns, so we can actually achieve a runtime of $O(nm \min(n, m))$.

Noticing the double-counting flaw of the single-update DP approach is tricky, so any reasonable solution to address the double-counting is acceptable here (e.g. a naive

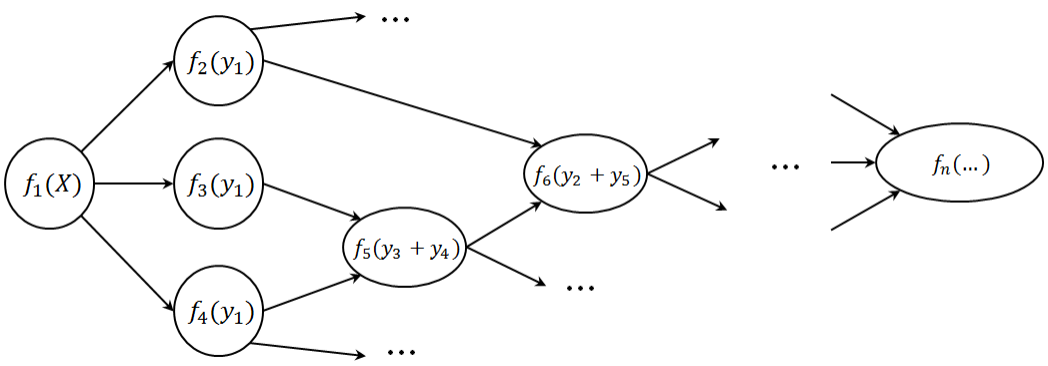
algorithm running in $O(mn^4)$ that correctly addresses the double-counting should not be penalized for having a slower runtime).

**Problem 9-2. Broken Backprop** [50 points]

A "computation graph" is a directed acyclic graph that defines a sequence of computations on a given input $X \in \mathbb{R}$. Each "node" in the graph is associated with some function $f_i : \mathbb{R} \to \mathbb{R}$. The input and output nodes of the graph are $f_1$ and $f_n$ respectively (where $n = |V|$). Let $x_i$ and $y_i$ represent the input and output respectively of $f_i$. Then,

$$x_i = \sum_{(j,i)\in E} y_j$$
$$y_i = f_i(x_i)$$

In other words, the input $x_i$ to each node is the sum of outputs $y_j$ from incident edges $(v_j, v_i) \in E$. Below is an example computation graph:



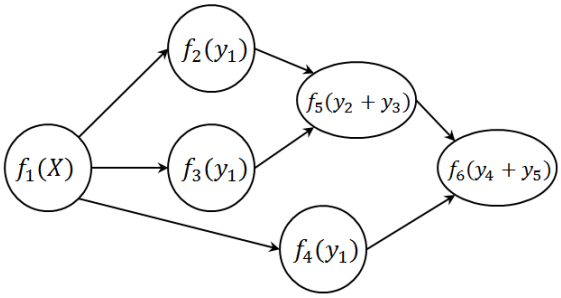

This type of abstraction comes up in many real-world applications, [1] and we are often interested in finding derivatives $\frac{\partial y_n}{\partial y_i}$. The equation for each derivative can be broken down as follows (using the chain rule):

$$\begin{aligned}\frac{\partial y_n}{\partial y_i} &= \sum_{(i,j)\in E} \frac{\partial x_j}{\partial y_i}\frac{\partial y_n}{\partial x_j} \\ &= \sum_{(i,j)\in E} \frac{\partial x_j}{\partial y_i}\frac{\partial y_j}{\partial x_j}\frac{\partial y_n}{\partial y_j} \\ &= \sum_{(i,j)\in E} \frac{\partial y_j}{\partial x_j}\frac{\partial y_n}{\partial y_j}\end{aligned}$$

where the last step follows from $\frac{\partial x_j}{\partial y_i} = \sum_{(k,j)\in E} \frac{\partial y_k}{\partial y_i} = 0 + \ldots \frac{\partial y_i}{\partial y_i} + 0 + \ldots = 1$.

[1] for example, "feed-forward neural networks" used in machine learning are a special type of computation graph

**(a)** [5 points] Warm Up: Derive $\frac{\partial y_6}{\partial y_1}$ for the following computation graph (evaluated on some arbitrary input $X$):




| | |
|---|---|
| $\partial y_1/\partial X$ | 2 |
| $\partial y_2/\partial x_2$ | 6 |
| $\partial y_3/\partial x_3$ | 4 |
| $\partial y_4/\partial x_4$ | 11 |
| $\partial y_5/\partial x_5$ | 3 |
| $\partial y_6/\partial x_6$ | 7 |

**Solution:**

$$\begin{aligned}\frac{\partial y_6}{\partial y_1} &= \frac{\partial y_2}{\partial x_2}\frac{\partial y_6}{\partial y_2} + \frac{\partial y_3}{\partial x_3}\frac{\partial y_6}{\partial y_3} + \frac{\partial y_4}{\partial x_4}\frac{\partial y_6}{\partial y_4} \\ &= 6 \cdot \frac{\partial y_5}{\partial x_5}\frac{\partial y_6}{\partial y_5} + 4 \cdot \frac{\partial y_5}{\partial x_5}\frac{\partial y_6}{\partial y_5} + 11 \cdot \frac{\partial y_6}{\partial x_6}\frac{\partial y_6}{\partial y_6} \\ &= 6 \cdot 3 \cdot \frac{\partial y_6}{\partial x_6}\frac{\partial y_6}{\partial y_6} + 4 \cdot 3 \cdot \frac{\partial y_6}{\partial y_6}\frac{\partial y_6}{\partial y_6} + 11 \cdot 7 \\ &= (18 \cdot 7 + 12 \cdot 7 + 11 \cdot 7) = 287\end{aligned}$$

**(b)** [20 points] Devise an algorithm to compute $\frac{\partial y_n}{\partial y_1}$ on *any* computation graph in $O(|E|)$ time. Assume you can compute $\frac{\partial y_i}{\partial x_i}$ for any $i$ in $O(1)$.

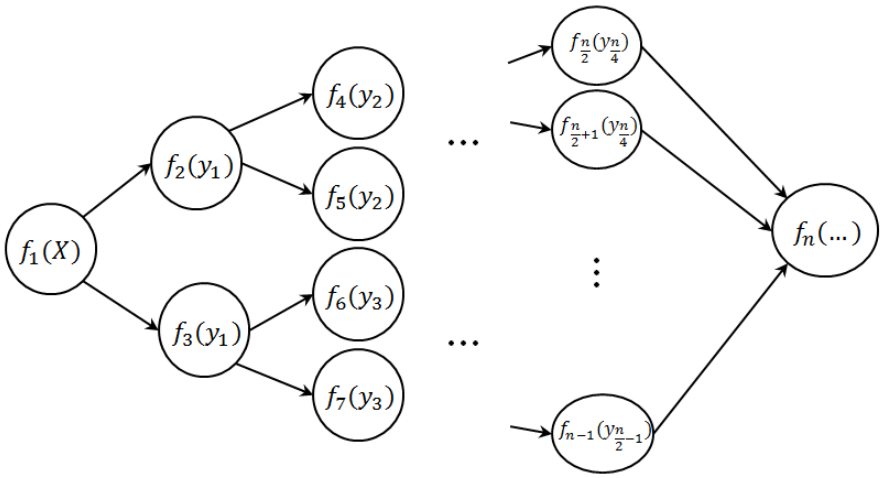
**Pre-Solution:**

The equation $\sum_{(i,j)\in E} \frac{\partial y_j}{\partial x_j}\frac{\partial y_n}{\partial y_j}$ gives us a recursive dependence between values $\frac{\partial y_n}{\partial y_i}$ and $\frac{\partial y_n}{\partial y_j}$ where $(i,j) \in E$. If we naively expand out the formula for each $\frac{\partial y_n}{\partial y_i}$, we may need to compute up to $n$ derivatives and multiply up to $n$ terms. However, since much of the computation is redundant (e.g. between computing the derivatives for two adjacent nodes), we can look into to memoizing our computation.

**Solution:** Let $DP[i] := \frac{\partial y_n}{\partial y_i}$, and as the notation implies, let us store all our values in an array. Then we can compute all $DP[i]$ using our recursive formula, so that $DP[i] = \sum_{(i,j)} \frac{\partial y_j}{\partial x_j} DP[j]$. Our base case is $DP[n] = \frac{\partial y_n}{\partial y_n} = 1$ and we populate our $DP$ array backwards. Our final answer is $DP[1]$ by definition.

For each $DP[i]$ we perform $d_i$ multiplications and $d_i$ derivative computations (both in $O(1)$) where $d_i$ is the out-degree of $v_i$. Therefore, our total runtime is $O(\sum_i d_i) = O(|E|)$.

Now, consider a computation graph where nodes $\{f_1, ..., f_{n-1}\}$ form form a directed binary tree, such that each $f_i$ has exactly one input and feeds into exactly two other functions. The remaining node $f_n$ takes input from all the leaves of the tree $\{f_{n/2}, ..., f_{n-1}\}$. Visually,




Suppose that some of the derivatives were computed incorrectly on a subset of the nodes. In particular, instead of $\frac{\partial y_i}{\partial x_i}$ we have $r \cdot \frac{\partial y_i}{\partial x_i}$ (for some constant $r > 1$) on certain nodes. Assume $\frac{\partial y_n}{\partial x_n}$ is *not* among these incorrect derivatives.

Let $\frac{\partial y_n}{\partial y_1}'$ represent the value computed by the chain rule expansion as in part (a) using these incorrect derivatives. Assume all derivatives $\frac{\partial y_i}{\partial x_i}$ are positive, hence $\frac{\partial y_n}{\partial y_1}'$ is always an overestimate of the true value of $\frac{\partial y_n}{\partial y_1}$.

**(c)** [25 points] Devise an algorithm determines the minimum value of $\frac{\partial y_n}{\partial y_1}'$ that can be attained by correcting at most $k$ derivatives. For full credit, your algorithm should run in $O(nk^2)$.

**Pre-Solution:**

With DP problems, a good idea for breaking down the original problem into suitable subproblems is looking at a subset of your input (in particular, subsets that look like a mini-version of your input). In the case of a tree, it is natural to consider subtrees rooted at child nodes as your subproblems. Our goal now is to flesh out a formal recurrence for $DP[i]$ (the solution of the subtree rooted at $v_i$) in terms of $DP[c_1]$ and $DP[c_2]$ (where $v_{c_1}$ and $v_{c_2}$ are the children of $v_i$).

A key question here is how to *distribute* our $k$ fixes in an optimal fashion. Since we have the option to fix a certain derivative (and thereby reduce the number of fixes available moving forward), it would be convenient to add another dimension to our $DP$ to encode the remaining number of fixes. In particular, we should refine our subproblem such that $DP[i][j]$ represents the solution of the subtree rooted at $v_i$ that uses *at most*

$k$ *fixes*. Armed with this intuition, we can now formally define our subproblems and recurrence.

**Solution:**
We define subproblems $DP[i][j]$ as the minimum value of $\frac{\partial y_n}{\partial x_i}'$ we can achieve by correcting at most $j$ derivatives in the subtree rooted at $v_i$.
First, we'll define a helper value

$$H[i][j] = \min_{0 \le q \le j} (DP[c_1][q] + DP[c_2][j-q])$$

where $c_1$ and $c_2$ are the indices of children of $v_i$ in the tree. This helper value represents the minimum value of $\frac{\partial y_n}{\partial y_i}'$ we can attain by optimally distributing the $j$ corrections between the two child subtrees.
Now to transform $\frac{\partial y_n}{\partial y_i}'$ into $\frac{\partial y_n}{\partial x_i}'$ (i.e. $DP[i][j]$) we must analyze two cases:
Case 1: $\frac{\partial y_i}{\partial x_i}$ is not a faulty derivative. Then we simply have

$$DP[i][j] = \frac{\partial y_i}{\partial x_i} H[i][j]$$

since we do not have to account for using up any fixes for this node. Note that multiplying $\frac{\partial y_n}{\partial y_i}$ by $\frac{\partial y_i}{\partial x_i}$ gives us $\frac{\partial y_n}{\partial x_i}$ as desired.
Case 2: $\frac{\partial y_i}{\partial x_i}$ is a faulty derivate. Now we have two options: (1) we fix $\frac{\partial y_i}{\partial x_i}$ and leave $j-1$ corrections for the children, or (2) we do not fix $\frac{\partial y_i}{\partial x_i}$ and leave $j$ corrections for the children (but incur a multiplicative penalty of $r$ for using our faulty derivative). Note, however, that the first option is always better than the second option since $r$ is a constant and the multiplicative penalty at the root will always be as great or greater than the multiplicative penalty at any of the nodes in the child subtrees (think of multiplying a sum of values vs. multiplying a single summand). Therefore, we have

$$\begin{aligned} DP[i][j] &= \frac{\partial y_i}{\partial x_i} \min(r \cdot H[i][j], H[i][j-1]) \\ &= \frac{\partial y_i}{\partial x_i} H[i][j-1] \end{aligned}$$

We can initialize our base case by setting $DP[n][j]$ for all $j$ to the true value of $\frac{\partial y_n}{\partial x_n}$ since $\frac{\partial y_n}{\partial x_n}$ is not among the incorrect derivatives.
Finally, we return $DP[1][k] \cdot \frac{1}{\frac{\partial y_1}{\partial x_1}}$ or alternatively, just return $H[1][k]$.
Runtime Analysis:
There are $O(nk)$ subproblems since $1 \le i \le n$ and $1 \le j \le k$. To check for the minimum of $k$ possible values naively, we have $O(k)$ time per subproblem (although this can be done faster with some pre-processing), giving us $O(nk^2)$ as desired.