CS 170 HW 7

Due 2020-3-9, at 10:00 pm

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write “none”.

2 DP solution writing guidelines

Try to follow the following 3-part template when writing your solutions.

- Define a function $f(\cdot)$ in words, including how many parameters are and what they mean, and tell us what inputs you feed into $f$ to get the answer to your problem.
- Write the “base cases” along with a recurrence relation for $f$.
- Prove that the recurrence correctly solves the problem.
- Analyze the runtime and space complexity of your final DP algorithm? Can the bottom-up approach to DP improve the space complexity?

3 No Backtracking

Let $G = (V, E)$ be a simple, undirected, and unweighted $n$-vertex graph, and let $A_G$ be its adjacency matrix, defined as follows:

$$A_G[i, j] = \begin{cases} 
1 & \text{if there is an edge between } i \text{ and } j \\
0 & \text{otherwise}
\end{cases}$$

We call a sequence of vertices $W = (u_0, u_1, \ldots, u_\ell)$ a walk if for every $i < \ell$, $\{u_i, u_{i+1}\}$ is an edge in $E$, and we call $\ell$ the length of $W$. Call a walk nonbacktracking if for every $i < \ell - 1$, $u_i \neq u_{i+2}$, i.e., the walk does not traverse the same edge twice in a row. In this problem, we will see a dynamic programming-based algorithm to compute the number of length-$\ell$ nonbacktracking walks in $G$ between every pair of vertices.

(a) Prove that $A_G^{\ell}[i, j] = \# \text{ of length-} \ell \text{ walks from } i \text{ to } j$.

(b) Let $I$ be the identity matrix (diagonal matrix of all-ones), $D_G$ be the degree matrix of $G$, i.e., the matrix defined as follows:

$$D_G[i, j] := \begin{cases} 
\text{degree}(i) & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}$$
and let $NB^{(\ell)}$ be the matrix such that $NB^{(\ell)}[i,j]$ contains the number of length-$\ell$ non-backtracking walks between $i$ and $j$. Prove that $NB^{(\ell)}$ satisfies the following recurrence relationship.

$$\begin{align*}
NB^{(1)} &= A_G \\
NB^{(2)} &= A_G^2 - D_G \\
NB^{(\ell)} &= NB^{(\ell-1)} \cdot A_G - NB^{(\ell-2)} \cdot (D_G - I).
\end{align*}$$

(c) Given $T$ as input, give an $O(Tn^{\omega})$-time dynamic programming-based algorithm to output $NB^{(T)}$ where $n^\omega$ is the time it takes to multiply two $n \times n$ matrices and $\omega \geq 2$.

(d) (Cool problem but worth no points) Given $T$, give a $O(n^3 \log T)$-time algorithm to output $NB^{(T)}$.

**Solution:**

(a) This can be proved by induction. Easy to see when $\ell = 1$. Suppose the statement is true for $\ell - 1$.

$$\begin{align*}
\text{# of length-$\ell$ } u \to v \text{ walks} &= \sum_{w \in V(G)} \text{# of length-$\ell - 1$ } u \to w \text{ walks} \cdot A_G[w,v] \\
&= \sum_{w \in V(G)} A_G^{\ell-1}[u,w] \cdot A_G[w,v] \\
&= A_G^{\ell-1} \cdot A_G[u,v] \\
&= A_G^{\ell}[u,v].
\end{align*}$$

(b) The case for $\ell = 1$ is immediate, and the case for $\ell = 2$ follows from the observation that all the walks which backtrack are recorded on the diagonal. Any length-$\ell$ nonbacktracking walk can be broken into 3 pieces (a) a nonbacktracking walk of length $\ell - 2$, followed by (b) a nonbacktracking step, followed by (c) another nonbacktracking step. On the other hand, $NB^{(\ell-1)} \cdot A_G$ records walks that are of the form (a) a nonbacktracking walk of length $\ell - 2$, followed by (b) a nonbacktracking step, followed by (c) any step. The walks of the second kind can be partitioned into length-$\ell$ nonbacktracking walks and walks that (a) take a length-$\ell - 2$ nonbacktracking walk, (b) take a nonbacktracking step, (c) backtrack along the step just taken. If the nonbacktracking walk from phase (a) ends at $u$, there are exactly $\text{degree}(u) - 1$ ways to perform phases (b) and (c), and thus these walks are recorded by the matrix $NB^{(\ell-2)} \cdot (D_G - I)$.

(c) For our DP algorithm, we store a length $T$ array of $n \times n$ matrices, where entry $i$ of this array is meant to contain $NB^{(i)}$. Computing $NB^{(i)}$ from the respective subproblems it breaks into by the recurrence in the previous part takes constant number of matrix multiplication and addition operations. Since there are $T$ subproblems, the computation takes $O(Tn^{\omega})$ time.

(d) For $\ell \geq 3$, observe that

$$\begin{bmatrix}
NB^{(\ell+1)} \\
NB^{(\ell)}
\end{bmatrix} = \begin{bmatrix}
NB^{(\ell)} & NB^{(\ell-1)}
\end{bmatrix} \cdot \begin{bmatrix}
A_G \\
-(D_G - I)
\end{bmatrix} = \begin{bmatrix}
I \\
0
\end{bmatrix}.$$
4 Walks in an infinite tree

Let $K_{d+1}$ be the undirected and unweighted complete graph on vertex set $\{0, \ldots, d\}$. Let $T_d$ be the undirected infinite tree with vertex and edge set

$$\begin{align*}
V_d &= \{W : W \text{ is a nonbacktracking walk starting at } 0 \text{ in } K_{d+1}\} \\
E_d &= \{\{W, W'\} : W' = (W, u) \text{ for some } u \in K_{d+1}\}.
\end{align*}$$

Let $u$ be an arbitrary vertex of $T_d$. In this problem, we will see a dynamic programming-based algorithm to compute the number of walks in $T_d$ from $u$ to $u$.

(a) Let $u$ and $v$ be two vertices in $T_d$ such that $\{u, v\}$ is an edge. Call a walk $u, w_1, \ldots, w_t, v$ from $u$ to $v$ in $T_d$ a first visit walk if $v \notin \{w_1, \ldots, w_t\}$, i.e., if $v$ is visited for the first time in the last step.

Let $F(\ell)$ be the number of length-$\ell$ first visit walks from $u$ to $v$. Write a recurrence for $F(\ell)$ and consequently give a dynamic programming algorithm that takes in $\ell$ as input and produces $F(\ell)$ as output. Your algorithm should run in $O(\ell^2)$ time.

*Hint: Suppose in the first step of a $u \rightarrow v$ first visit walk, $u$ steps to $v' \neq v$, the walk can be decomposed into 3 parts: (1) a single step from $u$ to $v'$, (2) a first visit walk from $v'$ to $u$, (3) a first visit walk from $u$ to $v$.*

(b) We call a walk $u, w_1, \ldots, w_t, u$ from $u$ to $u$ a first revisit walk if $u \notin \{w_1, \ldots, w_t\}$, i.e., if the only times $u$ is visited are at the start and the end. Let $G(\ell)$ be the number of length-$\ell$ first visit walks from $u$ to $u$. Give an $O(\ell^2)$-time algorithm that takes in $\ell$ as input and computes $G(\ell)$.

*Hint: You may want to use the algorithm from part (??).*

(c) Let $u$ be a vertex in $T_d$ and let $H(\ell)$ denote the number of walks from $u$ to $u$. Write a recurrence for $H(\ell)$ and consequently give a dynamic programming algorithm that takes
in $\ell$ as input and produces $H(\ell)$ as output. Your algorithm should run in $O(\ell^2)$ time. Your recurrence may also involve the function $G$ defined in part (??).

Solution:

(a) There is exactly one first-visit walk where the first step is to vertex $v$, and this first visit walk has length-1. Moreover it is the only length-1 first visit walk. So $F(1) = 1$. For any walk of length $\ell \geq 2$, we can assume that the first step was from $u$ to vertex $v' \neq v$; in particular, the walk can be broken up into 3 chunks, (a) the first step from $u$ to $v'$, (b) a length-$s$ $v' \rightarrow u$ first-visit walk, (c) a length $\ell - s - 1$ $u \rightarrow v$ first visit walk for any $s \leq \ell - 2$. For fixed $s$ there are $d - 1$ choices in (a), $F(s)$ choices in (b), and $F(\ell - s - 1)$ choices in (c), which leads to the recurrence

$$F(\ell) = \sum_{s=1}^{\ell-2} (d - 1) \cdot F(s) \cdot F(\ell - s - 1).$$

From the above recurrence, a dynamic programming-based algorithm to compute $F(i)$ takes $O(i)$ time to compute $F(i)$ from subproblems, and since there are $\ell$ subproblems, the runtime is bounded by $O(\ell^2)$.

(b) Any length-$\ell$ first revisit walk can be broken into (a) a single step from $u$ to $v$, followed by (b) a length-$\ell - 1$ $v \rightarrow u$ first visit walk. Since there are $d$ choices in (a) and $F(\ell - 1)$ choices in (b), this gives the formula

$$G(\ell) = d \cdot F(\ell - 1).$$

(c) First, note that the empty walk is a $u \rightarrow u$ walk, and hence $H(0) = 1$. For $\ell > 0$, a length-$\ell$ $u \rightarrow u$ walk can be decomposed into (a) a first revisit walk of length $s$ where $1 \leq s \leq \ell$, followed by (b) a length-$\ell - s$ $u \rightarrow u$ walk. This gives us the recurrence:

$$H(\ell) = \sum_{s=1}^{\ell} G(s) \cdot H(\ell - s).$$

Our algorithm to compute $H(\ell)$ first computes $G(1), G(2), \ldots, G(\ell)$ (which it can in $O(\ell^2)$ time). We then use the above recurrence for $H(\ell)$ to obtain a dynamic programming algorithm, which can compute $H(i)$ from its respective subproblems in $O(i)$ time. As a result, the runtime of the resulting DP algorithm can be bounded by $O(\ell^2)$.

5 GCD annihilation

Let $x_1, \ldots, x_n$ be a list of positive integers given to us as input. We repeat the following procedure until there are only two elements left in the list:

Choose an element $x_i$ in $\{x_2, \ldots, x_{n-1}\}$ and delete it from the list at a cost equal to the greatest common divisor of the undeleted left and right neighbors of $x_i$.

We wish to make our choices in the above procedure so that the total cost incurred is minimized. Give a poly($n$)-time dynamic programming-based algorithm that takes in the list
$x_1, \ldots, x_n$ as input and produces the value of the minimum possible cost as output. You may assume that we are given an $n \times n$ sized array where the $i, j$ entry contains the GCD of $x_i$ and $x_j$, i.e., you may assume you have constant time access to the GCDs.

**Solution:** Let $F(a, b)$ be the minimum cost incurred when the input is the subarray between indices $a$ and $b$. When $b = a + 1$, $F(a, b) = 0$. Suppose in performing the deletion on the $[a, b]$ subarray, element $s$ is the last element to be deleted, the total cost incurred is equal to $F(a, s) + F(s, b) + \gcd(x_a, x_b)$. This tells us that when $b > a + 1$,

$$F(a, b) = \min_{a + 1 \leq s \leq b - 1} F(a, s) + F(s, b) + \gcd(x_a, x_b)$$

Thus, if we turn the above recurrence to a DP algorithm, we get an $O(n^3)$ time algorithm since computing $F(a, b)$ from its subproblems takes up to $O(n)$ time and there are a total of $O(n^2)$ subproblems. The output of our algorithm is $F(1, n)$.

## 6 Counting Targets

We call a sequence of $n$ integers $x_1, \ldots, x_n$ valid if each $x_i$ is in $\{1, \ldots, m\}$.

(a) Give a dynamic programming-based algorithm that takes in $n, m$ and “target” $T$ as input and outputs the number of distinct valid sequences such that $x_1 + \cdots + x_n = T$. Your algorithm should run in time $O(m^2n^2)$.

(b) Give an algorithm for the problem in part (a) that runs in time $O(mn^2)$.

*Hint:* let $f(s, i)$ denotes the number of length-$i$ valid sequences with sum equal to $s$. Consider defining the function $g(s, i) := \sum_{t=1}^{s} f(t, i)$.

**Solution:**

(a) We use $f(i, s)$ to denote the number of sequences of length $i$ with sum $s$. $f(s, i)$ is 0 when $i > 0$ and $s \leq 0$, and $f(s, 1)$ is 1 if $1 \leq s \leq m$. Otherwise it satisfies the recurrence:

$$f(s, i) = \sum_{j=1}^{m} f(s - j, i - 1)$$

There are a total of $mn^2$ subproblems and it takes $O(m)$ time to compute $f(s, i)$ from its subproblems, which leads to an $O(m^2n^2)$ DP algorithm. Our algorithm outputs $f(T, n)$.

(b) We define $g(s, i)$ as follows:

$$g(s, i) = \sum_{j=1}^{s} f(j, i)$$

This is equal to

$$g(s, i) = f(s, i) + \sum_{j=1}^{s-1} f(j, i)$$

$$= \sum_{j=1}^{m} f(s - j, i - 1) + g(s - 1, i)$$

$$= g(s - 1, i - 1) - g(s - m - 1, i - 1) + g(s - 1, i).$$
Using this recurrence, there are still $mn^2$ subproblems, but it takes $O(1)$ time to compute $g(s, i)$ from its subproblems, and thus there is a $O(mn^2)$ time DP algorithm. We can then obtain $f(T, n)$ via $g(T, n) - g(T - 1, n)$.

7 Box Union

There are $n$ boxes labeled $1, \ldots, n$, and initially they are each in their own stack. You want to support two operations:

- **put($a, b$):** this puts the stack that $a$ is in on top of the stack that $b$ is in.
- **under($a$):** this returns the number of boxes under $a$ in its stack.

The amortized time per operation should be the same as the amortized time for find($\cdot$) and union($\cdot, \cdot$) operations in the union find data structure.

*Hint:* use “disjoint forest” and augment nodes to have an extra field $z$ stored. Make sure this field is something easily updateable during “union by rank” and “path compression”, yet useful enough to help you answer under($\cdot$) queries quickly. It may be useful to note that your algorithm for answering under queries gets to see the $z$ values of all nodes from the query node to its tree’s root if you do a find.

**Solution:** At any given time, let $u(s)$ denote the number of boxes under box $s$. In the disjoint forest union, let $z(s)$ denote the augmented field stored at node $s$. If $s$ is a root, we additionally store a parameter $size(s)$, which represents the total number of boxes in a given stack. At any given time, we will maintain the invariant that when $s$ is a root node, $z(s)$ is equal to $u(s)$, and otherwise $z(s)$ is equal to $u(s) - u(p(s))$ where $p(s)$ denotes the parent of $s$ in the disjoint forest data structure. To obtain the value of $u(s)$, we perform a find($s$) operation, and output the sum of $z(s')$ for $s'$ in the path between $s$ and the root $r$; this sum can be verified to equal $u(s)$. When the data structure is initialized, setting all $z(s)$ to 0, along with setting $size(s)$ to 1 maintains the invariant. Whenever a union operation is performed, one root $s$ is made a child of another root $s'$ — in this case, we

1. if $s'$ is in the “upper” stack, update $z(s')$ to $z(s') + size(s)$ and update $z(s)$ to $z(s) - z(s')$;
   otherwise if $s'$ is in the “lower” stack, keep $z(s')$ unchanged and update $z(s)$ to $z(s) + size(s') - z(s')$,
2. replace $size(s')$ with $size(s) + size(s')$.

Before a path compression operation is performed, we can compute the value of $u(s)$ for all $s$ whose parent is updated to root $r$ and replace each $z(s)$ with $u(s) - z(r)$. 