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.

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?

2 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 ℓ 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-ℓ nonbacktracking walks in G between every pair of vertices.

(a) Prove that $A_G^\ell[i,j] = \#$ of length-ℓ walks from i 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 $\mathrm{NB}^{(\ell)}$ be the matrix such that $\mathrm{NB}^{(\ell)}[i,j]$ contains the number of length-ℓ non-backtracking walks between i and j. Prove that $\mathrm{NB}^{(\ell)}$ satisfies the following recurrence relationship.

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

(c) Given T as input, give an $O(Tn^\omega)$-time dynamic programming-based algorithm to output $\mathrm{NB}^{(T)}$ where n^ω 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 $\mathrm{NB}^{(T)}$.

3 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

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

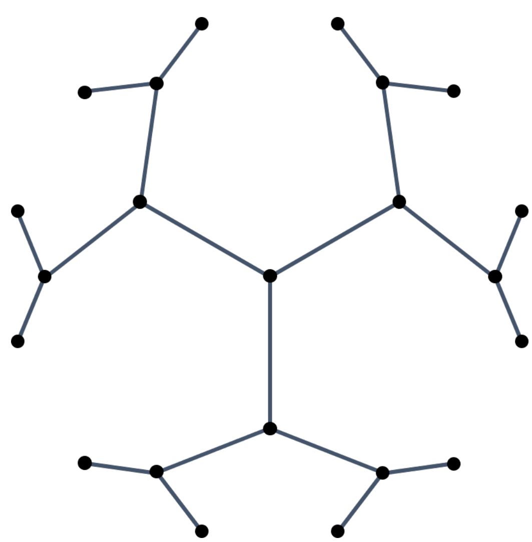

Figure 1: Finite piece of 3-regular infinite tree

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-ℓ first visit walks from u to v. Write a recurrence for $F(\ell)$ and consequently give a dynamic programming algorithm that takes in ℓ 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 $u \to v$ 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-ℓ first visit walks from u to u. Give an $O(\ell^2)$-time algorithm that takes in ℓ as input and computes $G(\ell)$.
Hint: You may want to use the algorithm from part (a).

(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 ℓ 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 (b).

4 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.

5 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 $O(m^2 n^2)$.

(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 $g(s, i) \coloneqq \sum_{t=1}^{s} f(t, i)$.

6 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 trees root if you do a find.*