15-451/651 Algorithms, Fall 2019

Homework #1
Due: September 12, 2019

This HW has three regular problems, and one programming problem. You may work in groups of (up to) three on these problems. But the write-up you submit must be written by you alone. Include the names of your group members on your submission.

Solutions to the three written problems should be submitted as a single PDF file using gradescope, with the answer to each problem starting on a new page. Unless otherwise stated, you should prove all of your answers. For example when presenting an algorithm you should prove correctness and the necessary running time bounds.

Submission instructions for the programming problem will be posted on Piazza.

0. Practice Exercise (do not turn in): Solving Recurrences

Give a tight asymptotic bound for the following recurrences. In each case explain the technique you use and why your answer is correct. For all these problems $T(1) = 1$. (Hint: In some cases it’s useful to write out the recursion tree.)

(a) $T(n) = 2T(\lfloor n/2 \rfloor) + 1$
(b) $T(n) = 3T(\lfloor n/2 \rfloor) + n \lg n$.
(c) $T(n) = 3T(\lfloor n/2 \rfloor) + n^3$
(d) $T(n) = T(\lfloor \sqrt{n} \rfloor) + 1$
(e) $T(n) = n^{2/3} T(\lfloor n^{1/3} \rfloor) + n$.

Solution :

(a) $T(n) \in \Theta(n)$
(b) $T(n) \in \Theta(n^{\log_2(3)}) = \Theta(n^{1.585})$
(c) $T(n) \in \Theta(n^3)$
(d) $T(n) \in \Theta(\lg \lg n)$
(e) $T(n) \in \Theta(n \lg \lg n)$
(25 pts) 1. **Finding the Longest Average-Edge-Length Cycle**

You’re given a directed graph $G = (V, E)$ where each edge $e \in E$ has a positive length $w(e)$. The lengths satisfy $1 \leq w(e) \leq B$. The number of vertices is $n$, and the number of edges is $m$. The average-edge-length of a cycle is the total length of the cycle divided by the number of edges on the cycle.

Give an algorithm to approximately compute the average-edge-length of the cycle with the greatest average-edge-length in $G$. The running time of your algorithm should be $O(nm \log(B))$. The approximation should be within 1% of the correct answer.

**Solution:**

Define

$$f(x) = \begin{cases} 
1 & \text{if there exists a cycle with average length greater than } x \\
0 & \text{otherwise}
\end{cases}$$

Note that for all $y < x$, $f(x) = 1 \implies f(y) = 1$ because if there is a cycle with average length greater than $x$, the average length of that same cycle is also greater than $y$. Thus there exists some $a \in [1, B]$ such that $f(x) = 1$ for all $x < a$, and $f(x) = 0$ for all $x \geq a$. By the definition of $f$, $a$ is the smallest value such that there does not exist a cycle with average length greater than $a$. Clearly $a$ equals the largest average length of a cycle, so our goal is to calculate $a$.

We can use binary search to find $a$. We maintain an interval $[l, r]$ where we know $a \in [l, r]$. Initially $[l, r] = [1, B]$. Then at every iteration, let $m = \left(\frac{l+r}{2}\right)$. If $f(m) = 0$, we know $a \in [l, m]$, and otherwise we know $a \in [m, r]$. The value of $r - l$ is reduced by a factor of 2 per iteration, so in $\log(100B) \in O(\log B)$ iterations, $r - l \leq 0.01$, at which point any value in $[l, r]$ will be within 0.01 and thus 1% of $a$ (as $a \geq 1$).

It remains to describe how to calculate $f(x)$. For a fixed $x$, obtain $G'$ from $G$ by negating all edge weights, then adding $x$ to each edge weight. We claim that a cycle in $G$ has average length greater than $x$ if and only if it is a negative cycle in $G'$.

To show this, fix a cycle $C$ in $G$, and let $A(C)$ be its average length. If $A(C) > x$, after negating the edge weights $A(C') < -x$, and then after adding $x$ to each edge, $A(C') < -x + x = 0$. So in $G'$, since $C$ has negative average weight, the sum of the edges on $C$ must be negative, and $C$ is a negative cycle. Similarly if $A(C) \leq x$, in $G'$ $C$ has nonnegative average length, so it is not a negative cycle.

So we can calculate $f(x)$ by first calculating $G'$ in $O(M)$ time, then using Bellman Ford to check for the existence of a negative cycle in $G'$ in $O(NM)$ time. Depending on how Bellman Ford is implemented, you may need to run it from a source vertex than can reach any other vertex. To handle this, add a new vertex $v$ to $G'$ with an edge of arbitrary weight to every other node. This does not create new cycles, since $v$ cannot be part of a cycle as it has no incoming edges. Then run Bellman Ford with source $v$. 


Since $f$ takes $O(NM)$ time and we need $O(\log B)$ calls to $f$, the total runtime of the algorithm is $O(NM \log B)$. 
2. Counting Subgraphs

Throughout this problem we will assume that $G = (V, E)$ is an undirected graph with $n$ vertices and $m$ edges. Let $A$ be its $n \times n$ adjacency matrix. So $A_{i,j} = A_{j,i}$ for all $i$ and $j$, and $A_{i,j} = 1$ if $\{i, j\} \in E$ and $A_{i,j} = 0$ if $\{i, j\} \not\in E$.

In an undirected graph, a $k$-clique is a set of $k$ distinct vertices such that there is an edge between every pair of these vertices. In class we gave an $O(n^{2.82})$ algorithm to determine if $G$ has a 3-clique.

(a) We also gave a formula for computing the number of 3-cliques. That is, the number of distinct subsets of three vertices that form a 3-clique.

$$\frac{1}{6} \sum_{A_{i,j} \neq 0} (A^2)_{i,j}$$

Prove this formula is correct.

(b) Give an $O(n^\omega)$-time algorithm for counting the number 4-cliques in $G$ for $\omega < 4$.

(c) Give an $O(n^\omega)$ time algorithm (for $\omega < 3$) to count the number of distinct 4-cycles in $G$. We can define a 4-cycle as a set of four edges such that those four edges involve four vertices, each one occurring in two of the edges.

So, in the complete graph of four vertices there are just three 4-cycles, with the following edge sets:

$$\{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{3, 4\}\}$$
$$\{\{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}$$
$$\{\{1, 2\}, \{1, 3\}, \{3, 4\}, \{2, 4\}\}$$

Solution:

(a) For every edge $(i, j)$ the term $(A^2)_{i,j}$ is the number of third vertices $k$ that form a triangle $(i, j, k)$. A given triangle $(a, b, c)$ is considered 6 times in this sum because it finds it for $(i, j) =$ each of $(a, b) \ (b, a) \ (a, c) \ (c, a) \ (b, c) \ (c, b)$.

(b) For each vertex $v$, construct the graph $G_v$, the subgraph of $G$ induced by the vertices which are neighbors of $v$. Now compute the number of triangles in $G_v$ using the algorithm in part a, Sum this over all vertices $v$. This will count each 4-clique 4 times, so you have to divide this sum by 4. The total work to construct all these subgraphs is easily done in $O(n^3)$ time, which is dominated by the other part.

(c) Compute $A^2$ as in part a. For each ordered pair of distinct vertices $(i, j)$ let $k = (A^2)_{i,j}$. Now sum together $\binom{k}{2}$ for all these $(i, j)$ pairs. This will count the number of 4-cycles, but each one will be counted 4 times. So divide by 4 to get the desired answer. Where does the 4 come from? Say the cycle is $(1, 2, 3, 4)$. Then it will be found when $(i, j) = (1, 3)(3, 1)(2, 4)(4, 2)$. 

4
3. **Summarize Data**

Let \( X = [x_1, x_2, \ldots, x_n] \) be a given sequence of real numbers. Also let \( C \) be a given constant. The goal in this problem is to approximate \( X \) with a piecewise constant function, in a specific way which is described below.

A sequence of numbers \( Y = [y_1, y_2, \ldots, y_n] \) can approximate \( X \) with the cost defined as follows:

\[
\text{Cost}(X, Y) = \sum_{i=1}^{n-1} C \delta(y_i, y_{i+1}) + \sum_{i=1}^{n} (x_i - y_i)^2
\]

Here \( \delta(y_i, y_{i+1}) \) is 0 if \( y_i = y_{i+1} \) and 1 otherwise. So the first summation above costs \( C \) each time the \( y \)'s change. The second term measures how closely the \( x \)'s and \( y \)'s match. The goal of this problem is to give an algorithm that computes the minimum over all possible \( Y \)'s of \( \text{Cost}(X, Y) \). Let’s call this quantity \( Q(X, C) \).

(a) Develop a dynamic programming algorithm to compute \( Q(X, C) \) in \( O(n^3) \) time.

(Make sure you state the meaning of the recurrence variable, state the recurrence, prove the recurrence is correct, and analyze the running time of the resulting algorithm.)

(b) Now improve the algorithm so that it runs in \( O(n^2) \) time.

Hint (for both parts): To do this it will be useful to make use of some of the *moments* of the sequence \( X \). The \( p \)th moment of \( X \), \( M_p \) is defined as follows:

\[
M_p = \sum_{i=1}^{n} x_i^p
\]

Note that \( M_0 = n \), and \( M_1 \) is just the sum of all the elements of \( X \).

We can also generalize this definition for a range, so \( M_p[j, k] = \sum_{i=j}^{k} x_i^p \).

For a range \([j, k]\) what value of \( y \) minimizes \( \sum_{i=j}^{k} (x_i - y)^2 \)? With this choice of \( y \) what is the value of this sum?

The moments that will be useful are the ones of order 0, 1 or 2. (I.e. \( p = 0, 1, \) or 2.)

**Solution:**

(a) First of all, observe that the optimal constant \( y \) value for a range would be the average of the \( x \) in that range. We can show this by taking the derivative of the second summation with respect to \( y \) and setting it equal to 0.

We want to do DP on incrementing ranges starting at 0. We start with \( DP[0] = 0 \) We want to calculate the range from \( x_0 \) to \( x_k \) for some value of \( k \leq n \), so we can assume we have all the optimal solutions calculated for the ranges \( x_0 \) to \( x_i \) for all \( i < k \). We iterate through all possible \( 0 < j < k \) which marks the rightmost place where the value
of $Y$ changes such that $y_{j-1} \neq y_j$. The optimal cost for each one of these iterations is the sum of

- The optimal cost for the range $x_0$ to $x_{j-1}$ which is $DP[j - 1]$
- The optimal cost of the right summation for a choice of constant $y$ in the range $x_j$ to $x_k$ which is the average of the $x_i$ in that range
- The additional cost of $C$ because the value of $Y$ changes

There is the additional case where $Y$ remains constant throughout the whole range of values in which case we also compute the average of the $x_i$ for the whole range $x_0$ to $x_k$ and don’t add the constant $C$. We take the minimum of all these values and store that into $DP[k]$. The final solution will be the last element of this array.

The size of the DP table is $n$, and on each step, we must iterate through all possible splitting points so $O(n)$, and we must also compute the average for the upper range, so $O(n)$. Multiplying these together gives us $O(n^3)$.

(b) We can remove the cost of computing the cost of the ranges when calculating the DP step in the algorithm by precomputing them. Notice that the right summation for a constant $y$ term becomes

$$\sum x_i^2 - 2x_iy + y^2 = M_2 - 2M_1y + ny^2$$

So if we have the moments precalculated, we can find the average $y = M_1/n$ in constant time along with the cost that average gives us. We can calculate all these moments using DP by using two arrays of size $n^2$ which stores the starting index and ending index of the range we want to find the moments of.

First, the moment of a single number is simply that number itself or squared. After that, we can add one element to it or add the square of one element to it in $O(1)$ time which expands the range covered. This means the total cost to precompute is $O(n^2)$ and this lets us remove the $O(n)$ operation of calculating the average during the other DP algo, so the overall algorithm is $O(n^2)$. 

6
4. **Programming: Cheapest Tree Separation**

There are \( N \) cities, numbered from 1 through \( N \), connected by \( N - 1 \) roads, forming a weighted tree. Countries \( A \) and \( B \) each occupy a set of cities (no city is occupied by both countries, and some cities may not be occupied at all).

To stop fighting between the two countries, you want to destroy roads such that no city occupied by country \( A \) is connected to a city in country \( B \). Destroying a road of length \( x \) costs \( x \) dollars. What is the minimum cost required? The time limit is 3 seconds.

**INPUT:** The first line contains \( N \) (\( 1 \leq N \leq 200000 \)), the number of cities. The next \( N - 1 \) lines contain three integers \( u, v, \ell \) (\( 1 \leq u, v \leq N, 1 \leq \ell \leq 10^9 \)), indicating that there is a road of length \( \ell \) between cities \( u \) and \( v \). The next line contains an integer \( h \) (\( 1 \leq h \leq N \)), the number of cities occupied by country \( A \), followed by \( h \) integers, describing the numbers of the cities occupied by country \( A \). The last line contains an integer \( k \) (\( 1 \leq k \leq N \)), the number of cities occupied by country \( B \), followed by \( k \) integers, describing the numbers of the cities occupied by country \( B \).

**OUTPUT:** Output a single integer, the minimum cost required to separate countries \( A \) and \( B \).

For example if the input is:

```
6
2 1 5
4 2 4
5 2 1
3 1 2
6 3 7
2 5 6
1 4
```

then the output is

```
3
```

**Solution:**

For brevity, we say that a node is labeled \( A \) if it is from country \( A \), and similar for \( B \). Observe that in the final cut of the tree into different components, each component contains at least one labeled node. This is because if a component does not contain a labeled node, we could get a smaller answer by leaving it connected to some other component.

Also clearly no component contains nodes of both labels \( A \) and \( B \). Thus we can think of each component as a component of label \( A \) or \( B \), depending on which label it contains.
Root the tree arbitrarily. For all nodes $u$, define $T_u$ as the subtree rooted at $u$. Define $DP[u][0]$ as the minimum cost of the problem restricted to $T_u$ (i.e. what is the minimum cost to remove edges in $T_u$ such that no node from country $A$ in $T_u$ can reach a node from country $B$ in $T_u$?), assuming $u$ must belong in a component of type $A$. Similarly define $DP[u][1]$ as the minimum cost for $T_u$, assuming $u$ must belong to a component of type $B$.

Then where $C_u$ denotes the children of $u$ and $w(u,v)$ denotes the weight of the edge connecting $u$ and $v$,

$$DP[u][0] = \sum_{v \in C_u} \min(DP[v][0], DP[v][1] + w(u,v))$$

$$DP[u][1] = \sum_{v \in C_u} \min(DP[v][1], DP[v][0] + w(u,v))$$

$DP[u][0] = \infty$ if $u$ is a node from country $B$, and similarly $DP[u][1] = \infty$ if $u$ is a node from country $A$.

Here for each child $v$, we are casing on which type of component $v$ belongs to, cutting the edge $(u,v)$ if necessary, and taking the case with less cost. Since the label of $u$'s component is fixed, the subproblems for each of the child subtrees of $u$ are completely independent.

The final answer will be $\min(DP[r][0], DP[r][1])$, where $r$ is the root of the tree that was chosen arbitrarily earlier.

A slightly different approach, which ends up with the exact same DP recurrence, is that the given problem is equivalent to trying to label all unlabeled nodes such that the total weight of edges connecting two nodes of different labels is minimized.

// Andy Yang
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
#define INF 1000000000000000LL
#define MAXN 200005
using namespace std;
l1 dp[MAXN][2];
vector<pair<int, int>> edges[MAXN];
bool a[MAXN];
bool b[MAXN];
void dfs(int node, int p){
   for(pii x : edges[node]){
      int v = x.first;
      int w = x.second;
      if (v == p) continue;
      dfs(v, node);
```c++
dp[node][0] += min(dp[v][0], dp[v][1] + w);
dp[node][1] += min(dp[v][1], dp[v][0] + w);
}
if (a[node]) dp[node][0] = INF;
if (b[node]) dp[node][1] = INF;
}
int main(){
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n - 1; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        edges[u].push_back({v, w});
        edges[v].push_back({u, w});
    }
    int q;
    scanf("%d", &q);
    for(int i = 0; i < q; i++) {
        int x;
        scanf("%d", &x);
        a[x] = true;
    }
    scanf("%d", &q);
    for(int i = 0; i < q; i++) {
        int x;
        scanf("%d", &x);
        b[x] = true;
    }
    dfs(1, -1);
    printf("%lld\n", min(dp[1][0], dp[1][1]));
}
```
B1 (Bonus) **Counting Hop-Paths**

Let \( A[1], A[2], \ldots, A[n] \) be an array of positive integers.

A valid path in the array starts at position 1, and hops to position \( 1 + A[1] \), then continues, where each subsequent hop from position \( j \) is to position \( j + A[j] \) or to position \( j + h \) (where \( h \) is the length of the previous hop). The valid path must end at position \( n \).

The goal is to compute, given the array \( A \), the number of distinct valid paths. Two valid paths are distinct if they visit a different set of cells. The running time of your algorithm should be \( O(n\sqrt{n}) \). As usual prove that your algorithm is correct and satisfies the running time bound.

**Solution**: Visit Andy at office hours for hints, or ask for the solution privately on Piazza.