CS2800 Design and Analysis of Algorithms
Problem Set #9 April 17, 2023

1. A *metapalindrome* is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For instance, the string BOBSMAMASEESAUKULELE can be decomposed as (BOB)(S)(MAM)(ASEESA)(UKU)(L)(ELE) where the lengths form the sequence (3,1,3,6,3,1,3). The length of the metapalindrome is the number of parts to which the string has been decomposed. Design an efficient algorithm to find the shortest metapalindrome of a given input string.

   Assume that you have access to a function IsPalindrome that checks if a given string is a palindrome.

   Solution: The idea is to guess the position k such that $A[1,k]$ and $A[n-k+1,n]$ are palindromes themselves, and $A[k+1,n-k]$ can be decomposed to a metapalindrome. Let $\text{len}(i)$ denote the length of the shortest metapalindrome of a string $A[i,i+1,\ldots,n-i+1]$. Either $A[i,i+1,\ldots,n-i+1]$ is already a palindrome, in which case $\text{len}(i)=1$. Otherwise, $\text{len}(i)$ is the minimum over all k, where $i<k\le n/2$, of the number $2+\text{len}(k)$ whenever IsPalindrome$(A[i,k-1])$ and IsPalindrome$(A[n-i-k,n-i+1])$ is true.

   Keep track of the value of k where the minimum is obtained to obtain the decomposition of the string to a metapalindrome.

2. Design an efficient algorithm to compute the longest contiguous subarray that appears both in forward and backward in a string $S[1,2,\ldots,n]$ such that the occurrences do not overlap. For instance, in REDIVIDE, the string EDIV appears in both directions, but the letter V overlaps. The correct value for this string is 3, corresponding to the string EDI.

   Solution: Let $\text{subarray}(i,j)$ denote the maximum length contiguous subarray in $A[i,i+1,\ldots,j]$ where the subarrays in both directions start in i and j. We are interested in $\max_{i,j}\{\text{subarray}(i,j)\}$. We can write a recurrence for $\text{subarray}(i,j)$ as follows:

   $$\text{subarray}(i,j) = (1+\text{subarray}(i+1,j-1))\cdot[A[i]=A[j]]$$

   The base cases are as follows: $\text{subarray}(i,i+1)=1$ if $A[i]=A[i+1]$, $\text{subarray}(i,i)=0$, and $\text{subarray}(i,j)=0$ when $i>j$.

3. Suppose you are given a set L of n line segments such that each line segment has one endpoint on the line $y=0$ and another on the line $y=1$. Assume that all the $2n$ endpoints are distinct.

   (a) Design an efficient algorithm to find the largest subset $L'\subseteq L$ such that no line segments in L' intersect.

Hint: Reduce this problem to finding the longest increasing subsequence of a sequence.

> **Solution:** Let (u_i, v_i) be the end-points of the line segments and suppose that $u_1 < u_2 < \cdots < u_n$. Now, two segments (u_k, v_k) and (u_l, v_l) where $k < l$ intersect iff $v_l < v_k$ (since the u_is are in the increasing order). Thus, a set of line segments are non-intersecting iff the corresponding sequence of v_is are increasing. Thus the problem reduces to finding the length of the longest increasing subsequence.

(b) Design an efficient algorithm to find the largest subset $L' \subseteq L$ such that every pair of line segments in L' intersect.

Hint: Reduce this problem to finding the longest common subsequence of two sequences.

> **Solution:** We can reduce this to finding the longest common subsequence of two sequences as follows: Let a character C_i correspond to the line segment (u_i, v_i). First sort u_is in the increasing order and write the sequence S corresponding to this order. Now, sort v_is in the decreasing order and let the sequence of characters be S'. Prove that the the cardinality of the set L' is the longest common subsequence between S and S'.

4. Let $T(V, E, w)$ be an edge-weighted tree where $w : E \longrightarrow \mathbb{R}^+$. A matching is a subset $E' \subseteq E$ of edges such that no two edges in E' share an end-point. Give a linear-time algorithm to compute the maximum-weight matching in a tree.

> **Solution:** Let $MWM(v)$ denote the weight of the maximum-weight matching in the subtree rooted at v, and let $MWM^+(v)$ and $MWM^-(v)$ denote the maximum-weight matching in the subtree rooted at v that includes an edge incident on v with one of its children, and excludes all edges incident on v with its children respectively. Like we did in class, $MWM(v) = \max\{MWM^+(v), MWM^-(v)\}$. Let C_v denote the children of v in the tree. Then we can write
>
> $$MWM^+(v) = \max_{z \in C_v} \left\{ w(v, z) + MWM^-(z) + \sum_{z' \in C_v - \{z\}} MWM(z') \right\},$$
>
> $$MWM^-(v) = \sum_{z \in C_v} MWM(z)$$

5. Let $p(u, v)$ denote the vertex occuring just before v in the shortest path from u to v in a graph. Describe an efficient algorithm to construct the matrix of predecessors $p(u, v)$ by suitably modifying the Floyd-Warshall algorithm.

Solution: Just like how the matrix $d(u, v, k)$ stores the length of the shortest path from u to v that pass through vertices numbered between 1 and k, we have a matrix $p(u, v, k)$ that stores the predecessor pointer from v of the shortest path from u to v that pass only through vertices numbered 1 to k. We can define it recursively as follows.

$$p(u, v, k) = \begin{cases} p(u, v, k-1) & \text{if } d(u, v, k-1) < d(u, k, k-1) + d(k, v, k-1), \\ p(k, v, k-1) & \text{otherwise} \end{cases}$$

This can now be suitably added into the Floyd-Warshall algorithm, and $p(u, v)$ is the same as $p(u, v, n)$.

6. Suppose that we have an edge-weighted graph $G(V, E, w)$ where the edge-weights could potentially be negative. Furthermore, the graph could contain negative weight cycles. Give an efficient algorithm that obtains the APSP $d(u, v)$ that satisfies the following conditions.

   - If v is not reachable from u, $d(u, v)$ should be ∞.
   - If v is reachable from u via a walk that contains a negative weight cycle, then $d(u, v)$ should be set to $-\infty$.
   - Otherwise, $d(u, v)$ should be the actual shortest distance from u to v in G.

Solution: After running the Floyd-Warshall algorithm on G, observe that if $d(u, u) < 0$, then u is in a negative weight cycle. Now, we can compute $\text{reach}_G(u)$ in G and $\text{reach}_{\text{rev}(G)}(u)$ in $\text{rev}(G)$ for every $u \in V$ (this will take $O(V^3)$ time). For every $v \in \text{reach}(u)$ and $z \in \text{reach}_{\text{rev}(G)}(u)$, $d(u, z) = -\infty$.

7. Suppose that $G(V, E, w)$ is a weighted digraph that do not contain negative weight cycles. We will design a slightly different $O(V^3)$-time algorithm to find APSP.

   (a) Let $v \in V$ be an arbitrary vertex. Design an $O(V^2)$-time algorithm to create a new graph $G'(V \setminus \{v\}, E', w')$ such that all the shortest path distances in G' is same as in G.

   Solution: For each pair (u, x) such that $(u, v), (v, x) \in E$, add an edge $(u, x) \in E'$ with edge weight $w'(u, x) = \min\{w(u, x), w(u, v) + w(v, x)\}$. All the other edges in E are present in E' with the same weight. Notice that no extra paths are created in G'. Every shortest path that does not pass through v has the same weight as before. If a shortest path in G passes through v, then we have the corresponding path that short-circuits v with the corresponding sum of edge weights in G'.

   (b) Suppose that we have computed APSP in G'. Design an $O(V^2)$-time algorithm to compute the shortest paths from v to all the other vertices, and from all the other vertices to v.

Solution: Let d' be the matrix of shortest paths in G'. For any vertex $x \in G$, we can write

$$d(x, v) = \min_{(u,v)\in E} \{d'(x, u) + w(u, v)\} .$$

Similarly, we have

$$d(v, x) = \min_{(v,u)\in E} \{w(v, u) + d'(x, u)\} .$$

(c) Describe how Parts (a) and (b) can be combined to obtain an $O(V^3)$-time algorithm for APSP.

Solution: Convert the two solution ideas above into an algorithm. Also work out how you will additionally add the matrix of predecessors to reconstruct the shortest paths.

8. Suppose you have a currency exchange where there are n different currencies and for each pair of currencies (i, j) the value $E(i, j)$ gives the exchange rate for converting 1 unit of currency i to currency j. An *arbitrage cycle* is a sequence of conversions that you can do such that starting with 1 unit of currency i, you end up with >1 unit of i. For instance, if the currencies are \$, ¥ and € such that the exchange rates are as follows 1\$= 150¥, 1¥=0.0068€, and 1€=1.1\$, then we can convert 1\$ to ¥ to € and then back to \$ to obtain 1.122\$. Give an efficient algorithm to check if the given input of currency exchanges contain an arbitrage cycle.

Solution: Create a graph $G(V, E, w)$ where the vertices are the currencies, and the edge (u, v) for currencies u and v has a weight equal to the exchange rate for converting 1 unit of currency u to currency v. An arbitrage cycle is a cycles $u = u_0, u_1, u_2, \ldots, u_k = u$ such that $\prod_{i=0}^{k-1} w(u_i, u_{i+1}) > 1$. In other words, $\sum_{i=0}^{k-1} \log\left(\frac{1}{w(u_i,u_{i+1})}\right) < 0$. Obtain a new graph $G'(V, E, w')$ where $w'(u, v) = -\log w(u, v)$ and run the algorithm to find the negative weight cycles.