Matrix sizes You can only multiply $r_1 \times c_1$ and $r_2 \times c_2$ dimensional matrices if $c_1 = r_2$, and the result is a $r_1 \times c_2$ dimensional matrix. Say that you are given a list of pairs of integers representing dimensions of matrix variables $M_1,...M_n$, where $M_i$ is a $r_i \times c_i$ dimensional matrix. You want to find a list of all possible dimensions $(r,c)$ of products of sequences of matrices from among $M_1...M_n$. Give an efficient algorithm for this problem. (10 points correct algorithm, 10 points efficiency).

Consider a graph $G$ where the nodes $V$ are all sizes of rows or columns for an input matrix, and where the edges $E = \{(r_i,c_i)\mid i = 1...n\}$, i.e., for each input matrix we add an edge from its number of rows to its number of columns. We claim that there is a sequence of input matrices whose product is an $r \times c$ matrix if and only if there is a path of length at least 1 in $G$ from $r$ to $c$. First, assume we can multiply $M_i_1M_i_2...M_i_k$ to get an $r \times c$ matrix $M$. Then the number of rows of the product $r$ must be $r_i_1$, the number of columns $c$ must be $c_i_k$ and each intermediate multiplication must have $c_i_j = r_i_{j+1}$ and there is an edge from each $r_i_j$ to $c_i_j$ by definition of $G$. Thus there is a path $r = r_i_1 \leftarrow c_i_1 = r_i_2 \leftarrow c_i_2... \leftarrow c_i_k = c$ from $r$ to $c$ in $G$ of the same length $k \geq 1$. Conversely, if there is a path from $r$ to $c$ of length $k \geq 1$ in $G$, $r = v_0 \leftarrow v_1 \leftarrow v_2... \leftarrow v_{k-1} \leftarrow c = v_k$, for each edge $v_j \leftarrow v_{j+1}$, there is an $v_j \times v_{j+1}$ input matrix $M_j$. Then we can multiply $M_0...M_{k-1}$ to get a $r \times c$ matrix.

Our algorithm is then to construct the adjacency list for $G$ (first, find $V$ by sorting all dimensions in $O(n \log n)$ time, rename vertices by their position in the array, and then for each matrix, insert an edge in constant time.) Then for each node $r$ we perform a DFS of $G$ from $r$ and record a list of all reachable nodes $c \neq r$, adding $r \times c$ to the list of possible matrix product dimensions. By the above claim, since the path to reach $c$ must be at least length one, this produces exactly the matrix dimensions of the form $r \times c$ with $c \neq r$. We store the reachable pairs in a matrix of Booleans, with the possible dimensions $r$ and $c$ as the indices. Finally, we need to check for which values of $r$ there is a $r \times r$ product. This is a bit subtle, since in the DFS, $r$ is always reachable from $r$, but there isn’t always such a product (because our characterization involved non-zero length paths.). But if there is a path of length 1 or higher from $r$ to $r$, it is either a self-loop or goes to some $c \neq r$ with $c \in N(r)$. By going through the adjacency list for $r$, we can see if $r$ has a self-loop, and check
for each candidate c whether r is listed as reachable from c (via a constant
time look-up in the matrix).

The total time then is $O(n \log n)$ to create $G$, $O(|V|(|V| + |E|))$ to do
$DFS$ from each node r, and $O(|V| + |E|)$ to do the final search for $r \times r$
matrices. Since $|V| \leq 2n$, and $|E| \leq n$, this gives a total time of at most
$O(n^2)$. (Notes: it is possible to have $\Omega(n^2)$ possible dimensions, e.g., if the
matrices are $1 \times 2, 2 \times 3, \ldots, n−1 \times n$. So any algorithm will be $\Omega(n^2)$ in the
worst-case. It is also easy to see that we can go in the reverse direction:
take any instance of the problem of which pairs of nodes are reachable from
which in a directed graph, and make an instance of the matrix product
problem where for each edge $(u_i, U − j)$ we create an $i \times j$ matrix. So
in some sense, this problem is just the transitive closure problem in thin
disguise.)

**Flight scheduling** You are devising a flight scheduler for a travel agency. The
scheduler will get a list of available flights, and the customer’s origin and
destination. For each flight, it is given the cities and times of departure
and arrival. The scheduler should output a list of flights that will take
the customer from her origin to her destination that arrives as early as
possible, subject to giving her at least 15 minutes for each connection.

Give a formal specification for this problem (Instance, Solution Space,
Constraints, Objective). (5 points)

- **Instance**: A set of $n$ cities, a set of $m$ flights $f$, each having an
  origin $O_f$ which is a city, a destination $D_f$ which is a city, a time of
  departure $d_f$ and a time of arrival $a_f$ with $d_f < a_f$. We are also
given two cities $s$ and $t$ (the customer’s origin and destination).
- **Solution Space**: A list of flights $f_1, \ldots, f_k$.
- **Constraints**: $O_{f_i} = s, D_{f_k} = t, D(f_i) = O(f_{i+1})$ for $1 \leq i \leq k − 1,$
  and $a_i + 15 < d_{i+1}$ for each $1 \leq i \leq k − 1$. (The sequence of flights
  must be a path from $s$ to $t$, and each flight must arrive at least 15
  minutes before the next departs.)
- **Objective**: Minimize $a_k$. (Arrival time at the customers destination)

b. Give an efficient algorithm to solve the problem.

We use a Dijkstra-like approach. We will fill in an array $T(c)$ which gives
the minimum time the customer could get to city $c$; we do this in order of
increasing minimum times, just like Dijkstra computes the distances to all
nodes in order of increasing distance. However, it won’t be efficient to just
use Dijkstra’s algorithm, although it is possible to solve the problem that
way. Say we have computed $T(c)$ for all the cities in a set $C$ with $s \in C$
and know that all cities outside $C$ require more time to get to than those
inside $C$. The next reachable city will be the destination of the flight from
C to outside C with smallest arrival time \( a_f \), given that the departure time satisfies \( d_f > T(O_f) + 15 \). (i.e., given that the passenger could make it to the airport the flight leaves from with at least 15 minutes for her connection.) We will also keep for each city the flight \( F(c) \) that arrives at the city at time \( T(c) \).

It is actually quite simple to implement this. Sort all the flights by arrival times. Initialize \( T(s) = 0 \). Run through the flights from smallest arrival time to largest. For each flight \( f \) check whether \( T(D_f) \) is already defined; if so, it must be smaller than \( a_f \), so there is no reason for the passenger to take this flight, since it is possible to reach the same city earlier. Also check that \( T(O_f) \) is defined and that \( d_f > T(O_f) + 15 \). If not, then the passenger cannot make the flight. Otherwise, set \( T(D_f) = a_f \) and set \( F(D_f) = f \). When we have defined \( T(t) \), we can stop, and find the sequence of flights that give this arrival time by running back using the \( F(c)'s \), i.e., \( f_k = F(t), f_{k-1} = F(O_{f_k}), \ldots \) The time is dominated by sorting the flights, so the algorithm is \( O(m \log m) \).

To prove the above algorithm is correct, we show the following by induction on the number of flights processed by the algorithm. Assume without loss of generality that no two flights have the exact same arrival time (if not, then break ties arbitrarily).

Claim: Immediately after we have processed flight \( f \), for each city \( c \), \( T(c) \) is the minimum time at which the passenger could arrive at \( c \) if this time is \( \leq a_f \) and is undefined otherwise.

Proof: Assume the claim holds for all earlier flights. Assume \( T(c) \) is defined. Either it was defined by some flight \( f' \) arriving before \( f \) does, and so by induction is correct and at most \( a_{f'} < a_f \), or it became defined in the step for \( f \). In the latter case, \( c = D_f, T(O_f) \) was already defined before this step, \( T(c) \) was not, \( d_f > T(O_f) + 15 \), and we set \( T(c) = a_f \). From the induction hypothesis for the flight \( f' \) arriving just before \( f \), we know that the minimum time the customer could arrive at \( c \) is greater than \( a_{f'} \) and hence at least \( a_f \). On the other hand, the customer could reach \( O_f \) by \( T(O_f) \) and then take flight \( f \) to \( c \) and reach \( c \) at \( a_f \). Thus, the minimum time to reach \( c \) is \( a_f = T(c) \).

If \( T(c) \) is undefined, and \( c \neq D_f \), then it was undefined at the flight \( f' \) immediately preceding \( f \) and hence has minimum arrival time \( > a_f \), and so \( \geq a_f \). So unless the passenger arrives via \( f \), the passenger will not be able to arrive at \( c \) until strictly after \( a_f \). But then \( c = D_f \), and since we did not define \( T(c) \) in this step, the minimum time to reach \( O_f \) is at least \( d_f - 15 \), so the passenger cannot make the flight \( f \). (Note: here we are using \( d_f < a_f \) implicitly.) Thus, in both cases, the claim remains true after we process \( f \).

It follows from the claim that as soon as \( T(t) \) is defined, we have correctly
computes the minimum time to reach $t$.

**Palindromic path** You are given a directed graph $G$ where every edge $e$ has a label, $l(e) \in \Sigma$, where $\Sigma$ is a finite set of symbols, and two nodes $s$ and $t$. You want to determine whether there is a path $e_1 \ldots e_k$ (not necessarily simple) from $s$ to $t$ so that the labels of the edges form a palindrome, i.e., $l(e_1) \ldots l(e_k) = l(e_k) \ldots l(e_1)$. (10 points correct algorithm and correctness proof, 10 points efficiency)

Let $G'$ be a new directed graph whose nodes are ordered pairs of nodes of $V$. Intuitively, each node in $G'$ represents the endpoints of potential palindromic path in $G$. An edge connects $(r', s')$ to $(r, s)$ in $G'$ if $(r, r'), (s', s) \in E$ and $l((r, r')) = l((s', s)) = \sigma$ for some symbol $\sigma$.

Let $S'$ be the subset of nodes $(s, s)$ for $s \in V$ and $(s, s') \in E$. These nodes represent paths in $G$ that are palindromes trivially. From these we build longer palindromic paths by moving through $G'$ as the following claim shows.

**Lemma:** There is a palindromic path from $r$ to $s$ in $G$ if and only if there is a path in $G'$ from a node in $S'$ to node $(r, s)$.

**Proof:** We first show that if there is a palindromic path from $r$ to $s$ in $G$, then there is a path from a node in $S'$ to $(r, s)$ in $G'$. Now consider a palindromic path $r, u_1, \ldots u_{k-1}, s$ from $r$ to $s$ and let $\omega_1 \cdots \omega_k$ be the labels of the edges used. Since the labels form a palindrome $\omega_1 = \omega_k$, $\omega_2 = \omega_{k-1}$, ... . If $k$ is even, the middle labels are $\omega_{k/2} = \omega_{k/2+1}$, and the node is $u_{k/2}$. Then $(u_{k/2}, u_{k/2}) \in S$, and the path $(u_{k/2}, u_{k/2}), (u_{k/2} - 1, u_{k/2} + 1), \ldots (u_1, u_{k-1}), (r, s)$ is in $E'$, so there is a path from a node in $S'$ to $(r, s)$ in $G'$. If $k$ is odd, the middle nodes are $u_{(k-1)/2}$ and $u_{(k+1)/2}$, and the pair $(u_{(k-1)/2}, u_{(k+1)/2}) \in S$ because they are adjacent in $G$. Then there is a path in $E'$ from this node to $(r, s)$, namely $(u_{k-1/2}, u_{(k+1)/2}), (u_{(k-3)/2}, u_{(k+3)/2}), \ldots (u_1, u_{k-1}), (r, s)$. Conversely, if there is a path $(u_0, v_0), (u_1, v_1), \ldots (u_k, v_k)$ from some $(u_0, v_0) \in S$, then if $u_0 = v_0$, the path $r, v_k, v_{k-1}, \ldots v_1, u_0, u_1, \ldots u_k, s$ is a palindromic path from $r$ to $s$ (since the edges $(v_i, v_{i-1})$ and $(u_{i-1}, u_i)$ exist and have the same label), and if $(u_0, v_0)$ is an edge, the path $r, v_k, v_{k-1}, \ldots v_1, v_0, u_0, u_1, \ldots u_k, s$ is a palindromic path from $r$ to $s$ (since the edges $(v_i, v_{i-1})$ and $(u_{i-1}, u_i)$ exist and have the same label). \( \square \)

To check whether such a path exists in $G$, we can add a new node $A$ and put in edges connecting $A$ to each $(u, v) \in S$. Then there is a path from $A$ to $(r, s)$ if and only if there is a path from some node in $S$ to $(r, s)$ if and only if there is a palindromic path from $r$ to $s$. We can run DFS on this new graph once from $A$ to get the entire list of all such pairs $(r, s)$, and check whether $(r, s)$ is on this list.

We construct the adjacency list representation of $G'$ as follows. For each pair of nodes we insert a node to $G'$. Then for each pair of edges, if the
labels are the same, we insert the corresponding edge in $E'$. Finally, we add the new node $A$, and for each node and edge we add one edge out of $A$ to the corresponding node in $S$. This takes time $O(|V|^2 + |E|^2)$ and creates a graph with $O(|V|^2)$ nodes and $O(|E|^2)$ edges. Running DFS from $A$ on this graph then takes time $O(|V'| + |E'|) = O(|V|^2 + |E|^2) = O(|E|^2)$ since we can remove all nodes of degree 0 in the original graph before this process begins.

**Implementation problem: Popular Websites** A web-company wants a data structure that will display webpages by popularity, displaying the most popular. The input will be a sequence of IP addresses. Intermittently, the data structure will need to display the top $k$ most frequently visited sites (where $k$ is given by the user). The data structure should update its list after every new site hit. So the data structure needs to store a list of webpages, ordered by $hit(site)$, the number of hits on the site. It needs to update this list each time a new hit is made, i.e., Update(site) adds site to the list with 1 hit, if it isn’t already there, or increments $hit(site)$ if it is. Top($k$) needs to report the top $k$ most popular sites. Describe and implement at least two data structures for this problem. Compare their performances on test data generated as follows:

Test distribution: A sequence of 1,000,000 random web addresses, each with probability 1/4 of being of the form: random 3 letter word.cs.edu probability 1/4 of the form: free.random 3 letter word.com and probability 1/2 of the form: random 2 letter word.random 2 letter word.com, org, edu. All words are lower case and have only standard letters. After every 1000 sites, perform Top($k$) for $k = 2^i$, $i$ uniformly chosen from 0 to 10.

Discuss any conclusions or issues that arose.