Quiz 2 Solutions

This take-home quiz contains 5 problems worth 25 points each, for a total of 125 points. Your quiz solutions are due between 7pm and 8pm on Wednesday, November 18, 2009 across from 32-141. Late quizzes will not be accepted unless you obtain a Dean’s Excuse or make prior arrangements with the course staff. You must hand in your own quiz solutions in person.

Guide to this quiz: Every problem (except one part about NP-completeness) asks you to design an efficient algorithm for a given problem. Your goal is to find the most efficient algorithm possible. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best possible solution will receive full points if well written, but ample partial credit will be given for correct solutions, especially if they are well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

Policy on academic honesty: The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz during the exam period, even if you have already handed in your quiz solutions.

This take-home quiz is “limited open book.” You may use your course notes, the CLRS textbook, and any of the materials posted on the course web page, but no other sources whatsoever may be consulted. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. You may not use any materials on the World-Wide Web, including OCW. You probably won’t find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, please send email to 6046-staff@csail.mit.edu.

Write-ups: Answer each problem on a separate sheet (or set of stapled sheets) of paper. Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time and your TA’s name. Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.
Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm’s execution should not replace a succinct description of the algorithm itself.

Provide short and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don’t just say that you sort $n$ numbers, state that you are using \textsc{Merge-Sort}, which sorts the $n$ numbers in $O(n \lg n)$ time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs:** If you think that you’ve found a bug, send email to 6046-staff@csail.mit.edu. (If you did not find a bug, and ask us a question, our answer will likely be “state your assumptions”.) Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements.

**PLEASE REREAD THESE INSTRUCTIONS ONCE A DAY DURING THE EXAM.**

**GOOD LUCK, AND HAVE FUN!**
Problem 1. Stripper Display

Your company ShowOff has invented a new bioalgorithmic manufacturing process called Stripper that produces, at very low cost, a batch of \( n \) strips of organic LEDs, each consisting of an \( m \times 1 \) array of pixels. Unfortunately, the process is error-prone, so many of the LEDs are damaged. Thus the \( i \)th strip can be represented by an \( m \times 1 \) vector \( s_i \) of 0s (broken) and 1s (working). For example, one batch with \( n = 6 \) and \( m = 8 \) might look like this:

\[
\begin{align*}
s_1 &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\
s_2 &= \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \\
s_3 &= \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \\
s_4 &= \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \\
s_5 &= \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \\
s_6 &= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}
\end{align*}
\]

Unsatisfied by one-dimensional displays, you want to assemble these strips into an \( m \times n \) matrix, using any order of columns you want, to maximize the area of the biggest rectangle filled with 1s. (Then you will cut out that large working rectangle and sell it.) In the example above, the optimal solution is any matrix that places strips \( s_2, s_4, s_5 \) as consecutive columns, yielding an all-1s rectangle of area 21 (these three columns and rows 2–8).

Design the most efficient algorithm you can to find a permutation of columns \( s_1, s_2, \ldots, s_n \) to form an \( m \times n \) binary matrix with the largest possible rectangle of all 1s.

Solution:

Executive summary: The algorithm finds the max rectangle by sweeping through all rows and calculating the sizes of all possible rectangles one can have by aligning the bottom of the rectangles with the \( i \)th row. The basic idea is to traverse through each row \( i \), and maintain the “height” of each column, i.e., the number of ones in that column extending upwards from row \( i \). Given the heights of each column, one has to find a subset \( S \) of columns maximizing \( |S| \cdot \min(S) \). This subset can be found by trying values for \( \min(S) \) and counting how many columns have heights that are greater or equal.

Algorithm: The straightforward approach would be to first calculate the heights extending upward from each row for each strip, and store this information in a matrix \( H \) (i.e., \( H[i,j] = \text{height of the sequence of working LEDs extending upwards from the } i\text{th row of the } j\text{th strip} \)). Then, traverse through each row \( i = 1, 2, \ldots, m \), sort the columns in a given row \( i \) by their heights in descending order, and find the subset \( S \) of columns maximizing \( |S| \cdot \min(S) \). The running time of such algorithm would depend on the sorting algorithm used, because the sorting dominates the work done in each row.
The more efficient approach is to sort each row as we traverse the rows to count the heights. The key observation is that, while the sweep line advances to the next row, the height for a column is either incremented or reset to zero from one row to the next. This means that one can maintain the sorted order in linear time: place the values that become zero to the end, and the order for the rest is maintained naturally (because the relative order among those columns with nonzero heights are not changed).

We use an array $\text{count}$ to maintain the height of each column. If in the current row, the value in a column is 0 then its $\text{count}$ will be set to 0. Otherwise, its $\text{count}$ is incremented. We can use a linked list $\pi$ to maintain the sorted order of the columns. When a column’s height becomes zero, the entry corresponding to that column is removed from the list and appended to the end of the list. When a column’s height is incremented, do nothing (because all unremoved entries in the list are also incremented).

The pseudocode is shown below.

```plaintext
1 For all $i = 1, 2, \ldots, n$, set $\text{count}_i = 0$
2 Let $\pi$ be a linked list of length $n$ with entries initially set to $1, 2, \ldots, n$
3 For all $i = 1, 2, \ldots, n$, let $\text{pointer}_i$ be the pointer that keep track of the entry of value $i$ in $\pi$.
4 Set $A_{\text{max}} = 0$
5 for each row $i$ of the matrix in order from 1 to $m$
6   do
7     for each column $j$
8        do
9           if the entry $(i, j)$ of the matrix is 1
10              then
11                 $\text{count}_i = \text{count}_i + 1$
12           else
13              $\text{count}_i = 0$
14       Use $\text{pointer}_i$ to remove the entry of value $i$ from $\pi$ and append it to the end of $\pi$
15   done
16   done
17 done
18 for $i = 1, 2, \ldots, n$
19   do
20     Let $k = \pi_i$
21     Let $A = \text{count}_k \cdot i$
22     if $A > A_{\text{max}}$
23           then
24               $A_{\text{max}} = A$
25               $\pi_{\text{max}} = \pi$
26 Output $\pi_{\text{max}}$
```
Running time: The running time to maintain the linked list and update count is linear ($O(N)$). Therefore, the total running time is $O(MN)$.

Alternate solutions: We summarize a few approaches we have seen in the submissions. Some people chose to do the straightforward approach. In this case, depending on the sorting algorithm used, the running time is either $O(M(N + M))$ (e.g., counting sort or bucket sort) or $O(M(N \log N))$ (e.g., merge sort).

Another (slightly less efficient) approach taken by many people is to count the number of strips that have consecutive working LEDs from row $i$ to row $j$ (inclusive, where $1 \leq i \leq j \leq m$), and find the max rectangle by maximizing $|S| \cdot (j - i + 1)$, where $S$ is the subset of columns with consecutive working LEDs between those rows. Using dynamic programming, one can count the number of columns with consecutive working LEDs between rows $i$ and $j$ in $O(NM^2)$ — to determine whether a strip has working LEDs between rows $i$ and $j$, one can look at the solution to the subproblem for working LEDs between rows $i$ and $j - 1$, and whether the $j$th LED is working. After obtaining this information, maximizing $|S| \cdot (j - i + 1)$ takes either $O(M^2)$ or $O(NM^2)$, depending on how this information is stored. In either case, the total running time is just $O(NM^2)$.

Yet another approach with a similar idea taken by some people is as follows. Each row (fixed column order) is represented as a bit string: $R_i = b_1b_2b_3 \cdots b_n$, where $b_j$ in $R_i$ is 1 if the $i$th row in $j$th strip is working, and 0 if otherwise. We calculate the results of ANDing all possible subsets of consecutive rows. The final string from ANDing these rows gives us the columns that have consecutive working LEDs between these rows, indicated by the positions and number of 1’s in the string. We maximize $|S| \cdot (\# \text{ of 1’s in } \text{AND}(S))$, where $S$ is the subset of consecutive rows. Again, one can use dynamic programming to find the result of ANDing all possible subsets of consecutive rows, where the base case is all subsets of size one: $R_1, R_2, \ldots, R_m$, the subproblems are subsets of size $k = 1, 2, \ldots, m$, with choices of which row is the first row. This can be done in $O(NM^2)$ (number of subproblems = $M$, choices per subproblem = $M$, and each subproblem takes $O(N)$ to compute).

Some people tried to take the greedy approach, where they greedily choose which column to append to the set next. This approach does not work. What’s locally optimal here may not lead to a globally optimal choice.

Problem 2. Around the Country

Based on your 6.046 performance, you have won a prize sponsored by Algorithmic Airlines: a free ticket to travel around the continental United States. You begin your trip at Bangor, Maine, which is the easternmost point served by the airline. Then you must travel only from east to west, until you reach Eureka, California, which is the western-most point served by the airline. Finally, you must come back, only by west-to-east travel, until you return to Bangor. However, during your trip, no city may be visited more than once, except for Bangor which must be visited exactly twice (at the beginning and the end of the trip). For publicity reasons, you are not allowed to use any other airline or any other means of transportation.
You want to get the most out of your free ticket by visiting the most cities possible. Being Algorithmic Airlines, they have provided you with an electronic list of cities served by the airline and a list of direct flights between pairs of cities. Design the most efficient algorithm you can to find an itinerary which visits as many cities as possible while satisfying the above conditions.

Solution:

Executive summary: Dynamic programming. Let \( n \) be the number cities. We number the cities from 1 to \( n \) in order from east to west: 1 is the easternmost city and \( n \) is the western-most city. We have \( n^2 \) subproblems, named \( D[i, j] \) for \( 1 \leq i, j \leq n \): find the length of the optimal path that starts from city \( i \), travels west to visit city \( n \), and travels east to return to city \( j \). Clearly, \( D[1, 1] \) is the length of the optimal path we want to find. We spend \( O(n) \) time per subproblem, for a total running time of \( O(n^3) \).

Dynamic program: We write a recurrence for \( D[i, j] \) based on the following case analysis:

- If \( i = j = n \), then the obvious solution is 1. \( D[n, n] = 1 \).
- If \( 1 < i = j < n \), there is no such path since the any feasible path should not visit city \( i \) more than once. \( D[i, j] = -\infty \).
- If \( i < j \), consider an optimal path. Let \( i_1 \) be the first city after \( i \) in the path.
  
  Observe that, if we remove \( i \) from this path, the remaining should be an optimal solution for the problem “finding the optimal path starting from \( i_1 \), visiting \( n \) and returning to \( j \)”. Therefore, \( D[i, j] = D[i_1, j] + 1 \).
  
  Also, observe that there should be no \( k > i \) such that \((i, k) \in E \) and \( D[k, j] > D[i_1, j] \). Otherwise, we can find a path that is longer than the optimal path (by starting from \( i \), visiting \( k \) and then following the optimal path). Contradiction.
  
  Putting the observations together, \( D[i, j] \) can be computed as \( \max\{D[k, j] : i < k \leq n \text{ and } (i, k) \in E \} + 1 \).
- If \( j < i \), similar to the case \( i < j \), \( D[i, j] = \max\{D[i, k] : j < k \leq n \text{ and } (j, k) \in E \} + 1 \).
- If \( i = j = 1 \), similar to the case \( i < j \), \( D[1, 1] = \max\{D[k, 1] : 1 < k \leq n \text{ and } (1, k) \in E \} + 1 \).

Thus we obtain the following recurrence:

\[
D[i, j] = \begin{cases} 
1 & \text{for } i = n \text{ and } j = n, \\
-\infty & \text{for } 1 \leq i = j < n, \\
\max\{D[k, j] : i < k \leq n \text{ and } (i, k) \in E \} + 1 & \text{for } i < j, \\
\max\{D[i, k] : j < k \leq n \text{ and } (j, k) \in E \} + 1 & \text{for } i > j.
\end{cases}
\]

Implementing this recurrence with recursion and memoization, we have \( O(n^2) \) subproblems and spend \( O(n) \) time per subproblem, so the total running time is \( O(n^3) \). Alternatively, we can compute the table \( D[i, j] \) bottom-up using the following pseudocode:
Handout 21: Quiz 2 Solutions

```plaintext
1 \quad D[n, n] = 1
2 \quad \textbf{for} i = 1, 2, \ldots, n - 1
3 \quad \quad \textbf{do} \quad D[i, i] = -\infty
4 \quad \textbf{for} i = 1, 2, \ldots, n - 1
5 \quad \quad \textbf{do} \quad D[i, i] = -\infty
6 \quad \textbf{for} i = n, n - 1, \ldots, 2, 1
7 \quad \quad \textbf{do}
8 \quad \quad \quad \textbf{for} j = n, n - 1, \ldots, 2, 1
9 \quad \quad \quad \quad \textbf{if} i < j
10 \quad \quad \quad \quad \quad \textbf{then} \quad \text{Find } k > i \text{ such that } (i, k) \text{ is an edge and } D[k, j] \text{ is maximum.}
11 \quad \quad \quad \quad \quad \quad D[i, j] = D[k, j] + 1
12 \quad \quad \quad \quad \quad \textbf{if} i > j
13 \quad \quad \quad \quad \quad \textbf{then} \quad \text{Find } k > i \text{ such that } (i, k) \text{ is an edge and } D[k, j] \text{ is maximum.}
14 \quad \quad \quad \quad \quad \quad D[i, j] = D[i, k] + 1
```

**Finding an optimal path:** Once the table \( D \) is computed, we can find the actual optimal path by backtracking on table \( D \) to find out which subproblems were used to compute \( D[1, 1] \). Detailed pseudocode:

```plaintext
1 \quad \text{Let } forward \text{ be an empty linked list.} \quad \triangleright \text{ to record the forward path from 1 to } n
2 \quad \text{Let } backward \text{ be an empty linked list.} \quad \triangleright \text{ to record the backward path from } n \text{ to 1}
3 \quad \text{Set } i = 1 \text{ and } j = 1.
4 \quad \text{Append 1 to } forward.
5 \quad \text{Append 1 to } backward.
6 \quad \textbf{while } i \neq n \text{ or } j \neq n
7 \quad \quad \textbf{do}
8 \quad \quad \quad \textbf{if} i < j
9 \quad \quad \quad \quad \textbf{then} \quad \text{Find } k_i \text{ such that } k_i > i \text{ and } D[i, 1] = D[k_i, 1] + 1.
10 \quad \quad \quad \quad \text{Set } i = k_i.
11 \quad \quad \quad \quad \text{Append } k_i \text{ to the end of } forward.
12 \quad \quad \quad \textbf{if} j < i
13 \quad \quad \quad \quad \textbf{then} \quad \text{Find } k_j \text{ such that } D[n, k_j] = D[n, k_j] + 1.
14 \quad \quad \quad \quad \text{Set } j = k_j.
15 \quad \quad \quad \quad \text{Append } k_j \text{ to the head of } backward.
16 \quad \quad \text{Output the concatenation of } forward \text{ and } backward \text{ (with duplicate (at } n \text{) removed).}
```

**Optimization:** We can improve the running time bound to \( O(VE) \) using the handshaking lemma, because the running time is proportional to \( n \) times the sum of the degrees of the vertices.

**Alternate solutions:** A few people chose to reduce the problem to a max-cost flow problem, and solve it using linear programming. To do this, start by creating a flow network with a node for
each city. To make sure that you don’t visit a city multiple times, split every node into two nodes, and connect the two with a directed edge. Set all edge capacities and costs in the flow network to 1. Finally, create a source and a sink, and connect them to Bangor and Eureka (respectively) with an edge of capacity 2. To solve the original problem, we need to get 2 units of flow across the network, and do so with the maximum possible cost. We can do this by solving a linear program that maximizes
\[
\sum_{(u,v) \in E} f(u, v)
\]
while satisfying the ordinary flow constraints. Since we need our flow to be integral, a correct solution must show how a fractional solution obtained by solving the linear program can be turned into an integral solution. One way to show that is to note that we will only get a non-integral flow when the network branches with an equal number of edges on both sides of the branching. In that case, we can just round the flow on one side of the branching to get an integral flow. To obtain the optimal tour, note that the max-cost flow on the network will be forced to take two separate paths from Bangor to Eureka. Take one of the paths and reverse it, and the two paths will constitute the optimal tour. This algorithm is correct but not as fast as the dynamic program.

There is also the obvious, and exponential, solution obtained by enumerating all the possible tours from Bangor to Eureka and back, checking the validity of the tour, and comparing the lengths to come up with the longest one. Enumerating the paths might take \(O(4^n)\), \(O(2^n \lg n)\), or \(O(2^n)\), depending on how you do it, and evaluating the validity and length of the tour might take \(O(n)\) or \(O(n + |E|)\).

A few people attempted to solve the problem using DFS or BFS in different ways, while making the wrong assumption that the running time of these algorithms is linear in the size of the original graph. The running time of these algorithms is linear in the size of the search tree, which is exponential in the size of the original graph if we are to get a correct solution.

A few people made the mistake of trying to solve the problem by finding the longest path in the DAG constructed by concatenating the two DAGs corresponding to the eastbound and westbound flights, while making sure that no city is visited twice. This approach assumes that our problem has the same optimal substructure as this longest (or shortest) path problem, which is not the case. To see why this is the case, note that finding a longest path does not guarantee an optimal roundtrip, because we might settle for a shorter path on each way in order to maximize the total length or the round trip.

Another related and common mistake is to assume that our problem has the greedy-choice property (for some choice), and attempt to get the optimal tour in a greedy manner. Because our problem does not have this property, greedy algorithms will not be optimal, even if they produce a valid solution.

**Problem 3. TwitDate**

You have founded an algorithmic dating company TwitDate that suggests to participating Twitter users who they should ask out on a date. At the foundation of your plan is that personality is
fundamentally one-dimensional, that like personalities attract, and that people prefer to ask others out via word-of-mouth through public tweets (messages on Twitter). For example, Angelina Jolie (AngelinaJolie) follows Larry King (LarryKingHeart), who follows solidity28, who follows Brad Pitt (thepitts). So you might suggest to Brad Pitt that he ask Angelina Jolie out by tweeting that he fancies her, with the intent that solidity28 and then Larry King retweet the message, which then Angelina Jolie will see.

You have downloaded the database of all Twitter follow relations, and represented it as a directed graph $G = (V, E)$, with edges directed from follower to followee. You have also precomputed your patented one-dimensional personality measure $P(u)$ for each Twitter user $u$. Design the most efficient algorithm you can that, for every user $u$, finds a user $v$ reachable by a chain of follows $u \to \cdots \to v$ and, subject to this constraint, minimizes $|P(u) - P(v)|$.

**Solution: Reduction to graph problem:** Let $G = (V, E)$ be the “follow” graph. The vertex set $V$ represents the people, and for any two vertices $u, v \in V$, there is a directed edge $(u, v)$ if $u$ follows $v$ on Twitter. We say that a vertex $v$ is reachable from a vertex $u$ if there is a directed path from $u$ to $v$ in $G$.

**Executive summary:** The basic idea is to compute the transitive closure of the graph (i.e., for each vertex $v \in V$, compute the set reachable vertices from vertex $v$). Then, for each vertex $v$, scan through the set of reachable vertices from $v$ to find the best match for $v$. We can solve the problem in $O(\min(V^2 + VE, V^{2.376} \log V))$ by taking the best of two algorithms. The first algorithm uses BFS or DFS, runs in $O(V^2 + VE)$ time and is best if $E = \Omega(V^{1.376} \log V)$. The second algorithm uses Coppersmith–Winograd matrix multiplication, runs in $O(V^{2.376} \log V)$ time and is best if $E = \omega(V^{1.376} \log V)$.

**Algorithm 1:** The $O(V^2 + VE)$ algorithm works as follows. For each vertex $v \in V$, use BFS or DFS to find the set of vertices that are reachable from $v$. Then, we scan through the set of vertices that are reachable from $v$ to find the best match. The running time for each execution of BFS (or DFS) is $O(E)$. Because we have to make $V$ executions of BFS (or DFS), one for each vertex, the total running time to compute the transitive closure is $O(VE)$. Once the transitive closure is found, the running time to find the best match for each vertex $v$ is $O(V)$. Therefore, the total running time is $O(V^2 + VE)$.

**Algorithm 2:** The $O(V^{2.376} \log V)$ algorithm works as follows. Let $A$ be the adjacency matrix for $G$, i.e., $A_{uv} = 1$ if $(u, v) \in E$, and $A_{uv} = 0$ otherwise. Let $B = (A + I)^V$ where $I$ is the identity matrix. We define the positivity indicator of $B$ to be the following matrix:

$$C_{uv} = \begin{cases} 1 & \text{if } B_{uv} > 0, \\ 0 & \text{otherwise}. \end{cases}$$

As we will prove later, for any pair of vertices $u, v \in V$, $C_{uv} = 1$ if and only if $v$ is reachable from $u$ in $G$. Therefore, in order to find the best match for a vertex $u$, we only need to look at the row $u$ of $C$ and find $v$ such that $C_{uv} = 1$ and $|P(u) - P(v)|$ minimizes.

The detail of the algorithm is shown in the following pseudocode:
1. Let $A$ be the adjacency matrix for $G$.
2. Let $A[u, u] = 1$ for all $u \in V$.
3. $C = \text{COMPUTE}(A, |V|)$  
   \[\triangleright \text{Computing } B = A^{|V|}\]
4. for each $u \in V$
   5. do
   6. Best-Match$(u) = \arg\min_v \{ |P(v) - P(u)| : B[u, v] > 0 \}$

where the function $\text{COMPUTE}$ can be implemented as follows:

$\text{COMPUTE}(A, n)$
1. if $n = 1$
2. then
3. \quad return $A$
4. if $n = 1 \pmod 2$
5. then
6. \quad $D = \text{COMPUTE}(A, n - 1)$
7. \quad $C = D \times A$
8. \quad Replace all positive entries of $C$ by 1.
9. \quad return $C$
10. else
11. \quad $D = \text{COMPUTE}(A, n/2)$
12. \quad $C = D \times D$
13. \quad Replace all positive entries of $C$ by 1.
14. \quad return $C$

**Running time:** In the algorithm above, computing $C$ takes $O(\log V)$ matrix multiplications each of which can be done in $O(V^{2.376})$ time using Coppersmith–Winograd matrix multiplication algorithm. Therefore, computing $C$ takes $O(V^{2.376} \log V)$. Computing the best matches for all vertices from $C$ takes only $O(V^2)$. Therefore, the overall running time of this algorithm is $O(V^{2.376} \log V)$. Note that in this algorithm, we don’t compute the matrix $B = (A + I)^V$ directly because the entries in this matrix may have exponential values.

**Proof of correctness:** It is sufficient to prove the claim above that, for any pair of vertices $u, v \in V$, $C_{uv} = 1$ if $v$ is reachable from $u$ in $G$, and $C_{uv} = 0$ otherwise.

Consider the sequence of matrices $C^{(1)}, C^{(2)}, \ldots, C^{(V)}$ such that $C^{(i)}$ is the positivity indicator of the matrix $B^{(i)} = (A + I)^i$. (Clearly, by definition, $C^{(V)} = C$.) We will prove the following stronger claim (which implies the claim we want to prove):

**Claim:** For any $1 \leq i \leq V$, for any pair of vertices $u, v \in V$, $C^{(i)}_{uv} = 1$ if there is a path of length at most $i$ from $u$ to $v$ in $G$, and $C^{(i)}_{uv} = 0$ otherwise.

**Proof of claim:** By induction on $i$. 
• Case $i = 1$. $B^{(i)} = A + I$. Clearly, $B^{(i)}_{uv} > 0$ if and only if $u = v$ or $(u, v) \in E$. Therefore, there must be a path of length at most 1 from $u$ to $v$.

• Case $i > 1$. By the induction hypothesis, $C^{(i-1)}_{uv} = 1$ if and only if $v$ is reachable from $u$ in $G$ by a path of length at most $k - 1$.

Consider any pair of vertices $u, v \in V$ such that $C^{(k)}_{uv} = 1$, thus, $B^{(k)}_{uv} > 0$. Since $B^{(k)} = B^{(k-1)} \cdot (A + I)$,

$$B^{(k)}_{uv} = \sum_{w=1}^{n} B^{(k-1)}_{uw} \cdot (A + I)_{wv}$$

Therefore, there must exist some $w$ such that $B^{(k-1)}_{uw} > 0$ and $(A + I)_{wv} > 0$. This implies that $w$ is reachable from $u$ by a path of length at most $k - 1$ and $v$ is reachable from $w$ by a path of length at most 1. Therefore, $v$ is reachable from $u$ by a path of length at most $k$.

Conversely, consider any pair of vertices $u, v \in V$ such that $v$ is reachable from $u$ by a path of length at most $k$. Let $P = (u = w_1, w_2, \ldots, w_{p-1}, w_p = v)$ with $p \leq k$ be one such path. Clearly, $A_{w_{p-1}v} = 1$ by definition. Furthermore, since $w_{p-1}$ is reachable from $u$ by a path of length at most $k - 1$, $B^{(k-1)}_{uw_{p-1}} > 0$. Therefore,

$$B^{(k)}_{uv} = \sum_{w=1}^{n} B^{(k-1)}_{uw} \cdot (A + I)_{wv} > B^{(k-1)}_{uw_{p-1}w} \cdot (A + I)_{w_{p-1}v} > 0$$

Therefore, $C^{(k)}_{uv} = 1$.

Alternate solutions: Some students reduced this problem to the all-pair shortest-path problem. This approach is also correct because, for any pair of vertices $u, v \in V$, $v$ is only reachable from $u$ if and only if the shortest path from $u$ to $v$ is less than infinity. However, the running time is always worse (because some computation was wasted to compute the shortest path). In particular, the Floyd-Warshall algorithm can be used to achieve running time $O(V^3)$. Johnson’s algorithm is slightly better and achieves running time $O(V^2 \log V + VE)$.

Some students made the very nice observation that, by identifying and contracting strongly connected components (see CLRS), the problem can be reduced to finding transitive closure on an acyclic graph. However, this trick does not help improve the running time.

Problem 4. Rein à Trois

The American Kidney Association (a.k.a. AKA) has started a new service for finding matching pairs of kidney donors and kidney recipients. In this service, a pair of people (e.g., spouses, siblings, or other relatives) can register as a donor–recipient pair: the donor of the pair offers a kidney for donation to someone else, in exchange for the recipient of the pair receiving a kidney from someone. The donor would have given her/his kidney directly to the recipient in the pair, except that their blood types differ, making them incompatible for kidney transplant. The AKA stores a list of all donor–recipient pairs that have registered, along with the blood types of every person. (Each person can register for only one pair.)
You’re heading a new project called ThreeWay that looks for three pairs \(a, b, c\) where \(a\)’s donor is compatible with (has the same blood type as) \(b\)’s recipient, \(b\)’s donor is compatible with \(c\)’s recipient, and \(c\)’s donor is compatible with \(a\)’s recipient. In this case, \(a, b, c\) form a \textit{rein à trois}.

(a) Design the most efficient algorithm you can to find a rein à trois, or report that none exist.

**Solution:** Executive summary: After appropriate reduction, this problem is equivalent to finding a directed triangle in a directed graph \(G = (V, E)\), where \(|V| = b\) is the number of blood types and \(|E| = n\) is the number of donor–recipient pairs. (Note that \(b \leq n\).) We can solve this problem in \(O(\min\{nb, n + b^{2.376}\})\) time by taking the best of two algorithms. The \(O(nb)\) algorithm is an optimized brute force, and is best when \(n = O(b^{1.376})\) (relatively sparse graphs). The \(O(n + b^{2.376})\) algorithm is based on Coppersmith–Winograd matrix multiplication, and is best when \(n = \Omega(b^{1.376})\) (relatively dense graphs).

**Reduction:** We construct the directed graph \(G = (V, E)\) with a vertex for each blood type, and an edge \((u, v) \in E\) if there is a pair whose donor has blood type \(u\) and whose recipient has blood type \(v\). [We could also define it the other way around.] A rein à trois \(a, b, c\) then corresponds to the three edges in a directed triangle in \(G\). We can construct an adjacency-matrix representation of this graph in \(O(n + b^2)\) time (which is smaller than the target running time) by scanning through the list of pairs of people (with their blood types) and setting the corresponding entry in the matrix to 1. From this representation, we can compute the adjacency-list representation of the graph in \(O(b^2)\) additional time.

**Algorithm 1:** The \(O(VE) = O(nb)\) algorithm works as follows: for each edge \((u, v) \in E\), test whether \(u\) and \(v\) have a common neighbor, i.e., a vertex \(x\) such that \((v, x) \in E\) and \((x, u) \in E\). There are \(|E|\) edges to test, and each test can be performed in \(O(V)\) time by scanning \(v\)’s row and \(u\)’s column in the adjacency matrix in parallel, checking for a common 1 entry.

**Algorithm 2:** The \(O(n + b^{2.376})\) algorithm uses the matrix-multiplication view of transitive closure, introduced in Lecture 11. Specifically, the adjacency matrix \(A\) indicates whether it is possible to go from one vertex to another via a directed path of length 1. The ring-world product \(A \odot A\) indicates whether it is possible to go from one vertex to another via a directed path of length 2. And \(A \odot A \odot A\) indicates whether it is possible to go from one vertex to another via a directed path of length 3. Therefore, \(A \odot A \odot A\) has a 1 on its diagonal if and only if there is a directed triangle. The two \(b \times b\) products can be computed using Coppersmith–Winograd matrix multiplication (as mentioned in lecture), for a cost of twice \(O(b^{2.376})\). We can then scan the diagonal in \(O(b)\) time. If we find a 1 on the diagonal, that is, a vertex \(v\) on a directed triangle, we can find the corresponding triangle in \(O(n)\) time by scanning all edges and testing whether they form a triangle with \(v\). Otherwise, there are no directed triangles.
Alternate solutions: We received a wide range of correct solutions, from one of the two algorithms above to an $O(n^4)$ algorithm. Many students found one of the algorithms described above, but no one found both and combined the algorithms to improve the overall performance as a function of $n$ and $b$. (A couple of students did so for two other algorithms, though.) The obvious brute-force algorithm tries all triples of pairs and checks compatibility of each, which takes $O(n^3)$ time. An improvement is to first translate pairs into a graph of blood types and then test all triples of blood types for existence of a triangle of edges, which takes $O(n + b^3)$ time. Many students found the $O(V E)$ algorithm for triangle finding, but used it on the graph $G' = (V', E')$ whose vertices represent pairs and whose edges represent compatibilities among pairs. Unfortunately, in this graph, $|V'| = n$ and $|E'|$ can be $\Theta(n^2)$, leading to a worst-case running time of $\Theta(n^3)$.

A common mistake was to use depth-first or breadth-first search to find triangles, which is incorrect; as far as the field knows, there is no way to modify these algorithms to find triangles in linear time.

[State-of-the-art: The best known running time for triangle finding in a directed graph $G = (V, E)$ is $O(\min\{E^{1.408}, V^{2.376}\})$ time. The $O(V^{2.376})$ algorithm is the second algorithm described above. The $O(E^{1.408})$ algorithm is described in the following paper: http://www.cs.bris.ac.uk/~ian/graphs/part.pdf (Theorem 3.5). Obviously we did not expect anyone to find this algorithm. If you found the quiz problem interesting, though, you might take a look now.]

(b) Prove that it is NP-complete to decide whether there are $k$ disjoint instances of rein à trois, that is, $k$ rein-à-trois triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots, (a_k, b_k, c_k)$ involving $3k$ distinct pairs of people. In your NP-hardness reduction, you may reduce from any of the NP-complete problems listed in Lectures 15, 16, or 17 or in the textbook.

Solution: Executive summary: First we show that the problem is in NP using the obvious certificate. Then we prove that the problem is NP-hard by a polynomial-time reduction from Three-Dimensional Matching.

In NP: To prove that the problem is in NP, we give a polynomial-length certificate for YES instances verifiable by a polynomial-time algorithm. The certificate is simply the solution: a list of $k$ rein-à-trois triples. Given such a certificate, we simply need to check two properties. First, each triple must form an instance of rein à trois; blood-type compatibility can be checked in $O(1)$ time per triple. Second, the triples must be disjoint, which we can check in $O(n^2)$ time by naively trying all pairs or in $O(n \log n)$ time by sorting. (We cannot use the $O(n)$ hashing solution because the algorithm must be deterministic.)

NP-hardness reduction: Three-Dimensional Matching (3DM) is one of the problems listed as NP-hard in Lecture 15. We reduce 3DM to the rein-à-trois problem, which proves that the latter is at least as hard and therefore also NP-hard. We are given an instance of 3DM, which consists of three disjoint sets $X, Y, Z$, a set $T \subseteq X \times Y \times Z$.
of triples, and a positive integer $k$. We construct a rein-à-trois instance, which we represent as a positive integer $k'$ and a directed graph $G = (V, E)$, whose vertices represent blood types and whose edges represent donor–recipient pairs. Our goal is for there to be $k'$ disjoint instances of rein-à-trois (which correspond to $k'$ edge-disjoint triangles in $G$) if and only if there are $k$ disjoint triples in $T$.

The reduction replaces each element in $x \in X \cup Y \cup Z$ with two distinct vertices $x_1, x_2$ connected by a directed edge $(x_1, x_2)$. Then, for each triple $(x, y, z) \in T$, we create the gadget shown in Figure 1 (left), by adding three new vertices $xyz, yzx, zxy$ and connecting them to $x_1, x_2, y_1, y_2, z_1, z_2$ as shown. Finally, we set $k' = 3|T| + k$. Note that this reduction takes polynomial time.

**Claim 1:** If there are $k$ disjoint triples $S \subseteq T$, then $G$ has $k'$ edge-disjoint triangles in $G$. Namely, for each triple $(x, y, z) \in S$, we include the four triangles $(x_1, x_2, zxy), (y_1, y_2, xyz), (z_1, z_2, yzx), (zxy, xyz, yzx)$ (Figure 1, middle); and for each triple $(x, y, z) \in T \setminus S$, we include the three triangles $(x_2, zxy, xyz), (y_2, xyz, yzx), (z_2, yzx, zxy)$ (Figure 1, right). The number of triangles is clearly $3|T| + k$. Edge-disjointness of the triangles holds within each gadget, and across gadgets because each element $x \in X \cup Y \cup Z$ appears in only one triple in $S$ and hence the corresponding edge $(x_1, x_2)$ appears in only one triangle.

**Claim 2:** If there are $k'$ edge-disjoint triangles in $G$, then there are $k$ disjoint triples $S \subseteq T$. Here we use that the only four-triangle solution within a gadget is the one in Figure 1 (middle), which requires the use of the edges $(x_1, x_2), (y_1, y_2), (z_1, z_2)$, which means that no other triples with $x$, $y$, or $z$ can have a four-triangle solution. Thus we choose $S$ to consist of the triples having four rein-à-trois triangles in their corresponding gadget. The remaining (unchosen) triples might as well use the three-triangle solution in Figure 1 (middle) in their gadget, because three triangles is the best possible, and this particular three-triangle solution does not use up any of the shared edges $(x_1, x_2), (y_1, y_2), (z_1, z_2)$. Therefore we obtain $k' - 3|T| = k$ disjoint triples in $S$.

**Figure 1:** Left: Gadget for each triple $(x, y, z)$. The bold edges are shared among multiple gadgets. Middle: Four triangles corresponding to chosen triple. Right: Three triangles corresponding to unchosen triple.
Alternate solutions: Many tried to reduce from 3DM, but without any gadgets. The resulting “natural” reduction is incorrect. The few who got correct solutions generally solved a different problem, in which the triangles must be vertex-disjoint instead of edge-disjoint. From the rein-à-trois perspective, this corresponds to allowing a more flexible definition of compatibility. This is a different problem, but we graded as if it were the same. In this variation, correct reductions were obtained from 3SAT, Graph 3-Coloring, and Independent Set.

Problem 5. The Producer
You are the producer for a soon-to-be cult-classic movie, 6.046 Returns, and you need both actors and investors. You have constructed a list of all \( n \) available actors (mostly former 6.046 students), with actor \( i \in \{1, 2, \ldots, n\} \) charging \( a_i \) dollars. For funding, you have found \( m \) available investors. Investor \( j \) offers \( b_j \) dollars toward your budget, but only on the condition that actors \( L_j \subseteq \{1, 2, \ldots, n\} \) are included in the movie. (The offer is “all or nothing”: all actors in \( L_j \) must be chosen in order to receive any funding from investor \( j \), and then the funding is \( b_j \).) Your profit is the sum of the payments from investors minus the sum of the payments to actors.

Design the most efficient algorithm you can to maximize your profit. (You have no limits on the number of actors or the amount of funding.)

Solution:

Executive summary: We can represent this problem as a linear program, and use a linear program solver to solve it in polynomial time. The most straightforward way to solve this problem via linear programming is to formulate this problem as an integer linear program, then relax the integrality constraints on the variables to get a linear program. One can prove that the solution to this relaxed linear program may be transformed into a solution for the original problem. A viable alternative solution, which is faster than linear programming, is to reduce this problem to maximum flow on a particular graph.

Exponential-time solutions: The simple exponential-time solution to this problem is to try all possible combinations of investors, compute the total profit from those investors, and choose the combination that maximizes this total profit. A similar exponential-time solution may iterate over all possible combinations of actors.

Linear-programming solution: Our problem can be rewritten as the following integer linear programming (ILP) problem:

\[
\begin{align*}
\text{maximize} \quad & \sum_{j=1}^{m} b_j y_j - \sum_{i=1}^{n} a_i x_i \\
\text{subject to} \quad & x_i \geq y_j \quad \text{for every } 1 \leq i \leq n, 1 \leq j \leq m \text{ such that } i \in L_j \\
& x_i = 0 \text{ or } 1 \quad \text{for each } i = 1, 2, \ldots, n \\
& y_j = 0 \text{ or } 1 \quad \text{for each } j = 1, 2, \ldots, m
\end{align*}
\]
In this ILP, the variable \( x_i \) corresponds to the actor \( i \) and the variable \( y_j \) corresponds to investor \( j \). \( x_i = 1 \) represents the event that actor \( i \) is hired, and \( x_i = 0 \) means that actor \( i \) is not hired. \( y_j = 1 \) indicates that the offer from investor \( j \) is accepted, and \( y_j = 0 \) indicates that investor \( j \)’s offer is not accepted.

If we relax the requirement that the solution must be integral, we will get the following linear programming (LP) problem:

\[
\text{maximize } \sum_{j=1}^{m} b_j y_j - \sum_{i=1}^{n} a_i x_i \\
\text{subject to } x_i \geq y_j \quad \text{for every } 1 \leq i \leq n, 1 \leq j \leq m \text{ such that } i \in L_j \\
\quad x_i \leq 1 \quad \text{for each } i = 1, 2, \ldots, n \\
\quad y_j \leq 1 \quad \text{for each } j = 1, 2, \ldots, m \\
\quad x_i \geq 0 \quad \text{for each } i = 1, 2, \ldots, n \\
\quad y_j \geq 0 \quad \text{for each } j = 1, 2, \ldots, m
\]

Clearly the optimal objective value for this LP is as least as large the optimal objective value for the ILP. Therefore, if we can show that there is an optimal solution for the LP that is integral, then this solution must also be the optimal solution for the ILP.

Let \((\vec{x}^*, \vec{y}^*)\) be an optimal solution for the LP. There are two possible cases:

1. \(x^*\) is integral (i.e., \(x_i^*\) is either 0 or 1 for every \(i = \{1, 2, \ldots, n\}\)): Assume that \(\vec{y}\) is not integral. There must be some \(y_j^*\) such that \(0 < y_j^* < 1\). Clearly, by setting \(y_j^* = 1\) we will obtain a better solution and no constraint is violated. This gives us a contradiction, and therefore \(\vec{y}\) must also be integral.

2. \(x^*\) is not integral: Let \(x_{\min}^*\) be the minimum positive value among all \(x_i^*\)’s (i.e., \(x_{\min}^* = \min_{x_i^*>0} x_i^*\)). Since \(x^*\) is not integral, \(x_{\min}^*\) is not integral. Let \(S_{\min} = \{i : x_i^* = x_{\min}^*\}\). Clearly, \(S_{\min}\) is non-empty by definition. Let \(T_{\min} = \{j : y_j^* > 0 \text{ and } L_j \cap S_{\min} \neq \emptyset\}\). By the constraints in the LP, \(y_j < x_{\min}^*\) for all \(j \in T_{\min}\). Let \(a = \sum_{i \in S_{\min}} a_i\) and \(b = \sum_{j \in T_{\min}} b_j\). There are 3 possible cases:

   i. \(a > b\): By setting \(x_i^* = 0\) for all \(i \in S_{\min}\) and \(y_j^* = 0\) for all \(j \in T_{\min}\), the objective value will be increased by

   \[
   \sum_{i \in S_{\min}} x_i a_i - \sum_{j \in T_{\min}} y_j b_j \geq a \cdot x_{\min} - b \cdot x_{\min} > 0.
   \]

   Therefore, we can obtain a better solution and no constraint is violated, which contradicts the optimality of our original solution.

   iii. \(a < b\): Let \(x_1^*\) be the next smallest positive value in \(\vec{x}\) (i.e., \(x_1 = \min_{x_i^*>x_{\min}^*} x_i^*\)). Then by increasing \(x_i^*\) for all \(i \in S_{\min}\) and \(y_j^*\) for all \(j \in T_{\min}\) by \(x_1 - x_{\min}\), the objective value will be increased by

   \[
   (b - a)(x_1 - x_{\min}) > 0.
   \]
Therefore, the can obtain a better solution and no constraint is violated, which again contradicts the optimality of our original solution.

ii. \( a = b \): By setting \( x_i^* = 0 \) for all \( i \in S_{\text{min}} \) and \( y_j^* = 0 \) for all \( j \in T_{\text{min}} \) we will obtain another optimal solution for the LP with smaller number positive variables and no constraint is violated.

Therefore, \((x^*, y^*)\) is either integral or there is an algorithm to obtain another optimal solution from \((x^*, y^*)\) with smaller number of positive variables.

Our algorithm works as follows:

1. Set up a LP problem based on the input.
2. Solve the LP problem (using an efficient LP solver) to obtain a solution \((x^*, y^*)\).
3. \textbf{while} \((x^*, y^*)\) is not integral \textbf{do}
4. \hspace{1em} \( x_{\text{min}}^* = \min_{x_i^* > 0} x_i^* \)
5. \hspace{1em} \( S_{\text{min}} = \{ i : x_i^* = x_{\text{min}}^* \} \)
6. \hspace{1em} \( T_{\text{min}} = \{ j : y_j^* > 0 \text{ and } L_j \cap S_{\text{min}} \neq \emptyset \} \)
7. \hspace{1em} Set \( x_i^* = 0 \) for all \( i \in S_{\text{min}} \)
8. \hspace{1em} Set \( y_j^* = 0 \) for all \( j \in T_{\text{min}} \)
9. \textbf{Output} \((x^*, y^*)\).

**Maximum-flow solution:** A faster algorithm for solving this problem uses a reduction to maximum flow. One may represent this problem using a flow network containing one vertex \( y_j \) for each investor \( j \), one vertex \( x_i \) for each actor \( i \), one source vertex \( s \), and one sink vertex \( t \). Connect \( s \) to each \( x_i \) with an edge of capacity \( a_i \), and connect each \( y_j \) to \( t \) with an edge of capacity \( b_j \). Finally, for each actor \( i \) and each investor \( j \), if \( i \in L_j \) then connect \( x_i \) to \( y_j \) with an infinite capacity edge. Finding the maximum flow through this flow network will expose a minimum cut of this flow network.

We can find an \((S, T = V \setminus S)\) cut in this flow network that has finite capacity, so the minimum \((S, T)\) cut of this flow network must also have finite capacity. Observe that the finiteness of the minimum cut of this flow network must enforce the all-or-nothing constraint of the problem. Suppose some actor vertex \( x_i \) is in the \( S \) portion of the minimum cut, and suppose some actor vertex \( y_j \), where \( i \in L_j \), is in the \( T \) portion of this cut. Because \( i \in L_j \), by construction of this flow network, there must exist an edge of infinite capacity from \( x_i \) to \( y_j \). Therefore, there must exist an edge of infinite capacity from \( S \) to \( T \), and the total capacity of this cut must be infinite. This contradicts our assertion that the minimum cut must have finite capacity, so this scenario cannot happen. Therefore, if an investor vertex \( y_j \) is in the \( T \) portion of this cut, then all actor vertices \( x_i \) where \( i \in L_j \) must also be in the \( T \) portion of this cut.

We can derive the optimal profit for this problem from the size of the minimum cut in this flow network. Note that the total profit from the optimal selection of actors \( i \) and investors \( j \) is
The sum of all investor contributions is fixed for a given problem instance, so the capacity of the minimum cut must maximize the value of \( \sum_{j: y_j \in S} b_j - \sum_{i: x_i \in T} a_i \), which is exactly the equation for the profit of the original problem. Consequently, the minimum cut of this flow network maximizes the profit of the original problem, and because maximum flow is equivalent to minimum cut, finding the maximum flow of this flow network gives us an optimal solution to the original problem.

A similar solution to this problem swaps the positions of the actor and investor nodes in this flow network, and therefore chooses the actors and investors on the \( S \) side of a minimum cut.

**Alternate attempts:** Many people tried to solve this problem using dynamic programming. The difficulty with such a solution is that this problem does not exhibit optimal substructure, so any feasible dynamic programming solution would be equivalent to an exponential time algorithm. Similarly, greedy strategies to this problem do not work due to this problem’s lack of optimal substructure.