Practice Quiz 2

This take-home quiz contains 5 problems worth 25 points each, for a total of 125 points. Each problem should be answered on a separate sheet (or sheets) of 3-hole punched paper.

Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time, and your TA. **Your exam is due between 9:00 and 11:00 A.M. on Friday, April 29, 2005.** Late exams will not be accepted unless you obtain a Dean's Excuse or make prior arrangements with your recitation instructor. You must hand in your own exam in person.

The quiz should take you about 10 hours to do, but you have four days in which to do it. Plan your time wisely. Do not overwork, and get enough sleep. Ample partial credit will be given for good solutions, especially if they are well written. Of course, the better your asymptotic bounds, the higher your score. Bonus points will be given for exceptionally efficient or elegant solutions.

Write-ups: Each problem should be answered on a separate sheet (or sheets, stapled separately for each problem) of 3-hole punched paper. Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time, and your TA. Your solution to 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 running time your algorithm achieves.

Write up your solutions cleanly and concisely to maximize the chance that we understand them. Be explicit about running time and algorithms. For example, don't just say you *sort* n numbers, state that you are using heapsort, which sorts the n numbers in $O(n \lg n)$ time in the worst case. When describing an algorithm, give an English description of the main idea of the algorithm. Use pseudocode only if necessary to clarify your solution. Give examples, and draw figures. Provide succinct 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.

Part of the goal of this exam is to test 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, etc.: If you think that you've found a bug, send email to `6.046 course staff.` 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. If you did not receive an email last Friday reminding you about Quiz 2, then you are not on the class email list and you should let your recitation instructor know immediately.

Policy on academic honesty: This quiz is “limited open book.” You may use your course notes, the CLRS textbook, basic reference materials such as dictionaries, and any of the handouts posted on the course web page, but *no other sources whatsoever may be consulted*. For example, you may not use notes or solutions from other times that this course or other related courses have been taught, or materials on the World-Wide Web. (These materials will not help you, but you may not use them anyhow.) Of prime importance, you may not communicate with any person except members of the 6.046 staff about any aspect of the exam until after noon on Friday, April 29, even if you have already handed in your exam.

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, send email to `6.046 course staff`.

Survey: Attached to this exam is a survey on your experiences with the exam, especially as they relate to academic honesty. Please detach the survey, fill it out, and hand it in when you hand in your exam. Responses to the survey will be anonymous. No attempt will be made to associate a response with a person. This information will be used to gauge the usefulness of the exam, and summary statistics will be provided to the class.

PLEASE REREAD THESE INSTRUCTIONS ONCE A DAY DURING THE EXAM.
GOOD LUCK, AND HAVE FUN!

Problem 1. ***Static Graph Representation***

Let $G = (V, E)$ be a sparse undirected graph, where $V = \{1, 2, \ldots, n\}$. For a vertex $v \in V$ and for $i = 1, 2, \ldots, \text{out-degree}(v)$, define v's ***ith neighbor*** to be the ith smallest vertex u such that $(v, u) \in E$, that is, if you sort the vertices adjacent to v, then u is the ith smallest.

Construct a representation of the graph G to support the following queries:

- DEGREE(v): returns the degree of vertex v.
- LINKED(u, v): output TRUE if an edge connects vertices u and v, and FALSE otherwise.
- NEIGHBOR(v, i): returns v's ith neighbor.

Your data structure should use asymptotically as little space as possible, and the operations should run asymptotically as fast as possible, but space is more important than time. Analyze your data structure in terms of both space and time.

Problem 2. ***Video Game Design***

Professor Cloud has been consulting in the design of the most anticipated game of the year: ***Take-home Fantasy***. One of the levels in the game is a maze that players must navigate through multiple rooms from an entrance to an exit. Each room can be empty, contain a monster, or contain a life potion. As the player wanders through the maze, points are added or subtracted from her ***life points*** L. Drinking a life potion increases L, but battling a monster decreases L. If L drops to 0 or below, the player dies.

As shown in Figure 1, the maze can be represented as a digraph $G = (V, E)$, where vertices correspond to rooms and edges correspond to (one-way) corridors running from room to room. A vertex-weight function $f : V \to \mathbb{Z}$ represents the room contents:

- If $f(v) = 0$, the room is empty.
- If $f(v) > 0$, the room contains a life potion. Every time the player enters the room, her life points L increase by $f(v)$.
- If $f(v) < 0$, the room contains a monster. Every time the player enters the room, her life points L drop by $|f(v)|$, killing her if L becomes nonpositive.

The ***entrance*** to the maze is a designated room $s \in V$, and the ***exit*** is another room $t \in V$. Assume that a path exists from s to every vertex $v \in V$, and that a path exists from every vertex $v \in V$ to t. The player starts at the entrance with $L = L_0$ life points, i.e. L_0 is the value of f for the entrance. The figure shows $L_0 = 1$.

Professor Cloud has designed a program to put monsters and life potions randomly into the maze, but some mazes may be impossible to safely navigate from entrance to exit unless the player enters with a sufficient number $L_0 > 0$ of life points. A path from s to t is "safe" if the player stays alive along the way, i.e., her life-points never become non-positive. Define a maze to be ***r-admissible*** if a safe path through the maze exists when the player begins with $L_0 = r$.

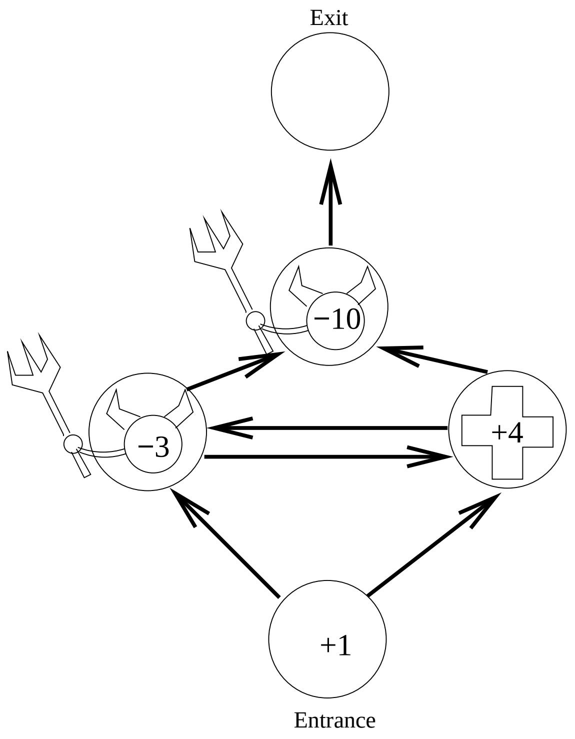

Figure 1: An example of a 1-admissible maze.

Help the professor by designing an efficient algorithm to determine the minimum value r so that a given maze is r-admissible, or determine that no such r exists. (For partial credit, solve the problem determining whether a maze is r-admissible for a given r.)

Problem 3. ***Image Filtering***

Two-dimensional filtering is a common operation in vision and image processing. An image is represented as an $n \times n$ matrix of real values. As shown in Figure 2, the idea is to pass a $k \times k$ window across the matrix, and for each of the possible placements of the window, the filter computes the "product" of all the values in the window. The "product" is not typically ordinary multiplication, however. For this problem, we shall assume it is an associative and commutative binary operation $\otimes$ with identity element e, that is, $x \otimes e = e \otimes x = x$. For example, the product could be $+$ with identity element 0, $\times$ with 1, $\min$ with ∞, etc. Importantly, you may not assume that $\otimes$ has an inverse operation, such as $-$ for $+$.

To be precise, given an $n \times n$ image

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{pmatrix},$$

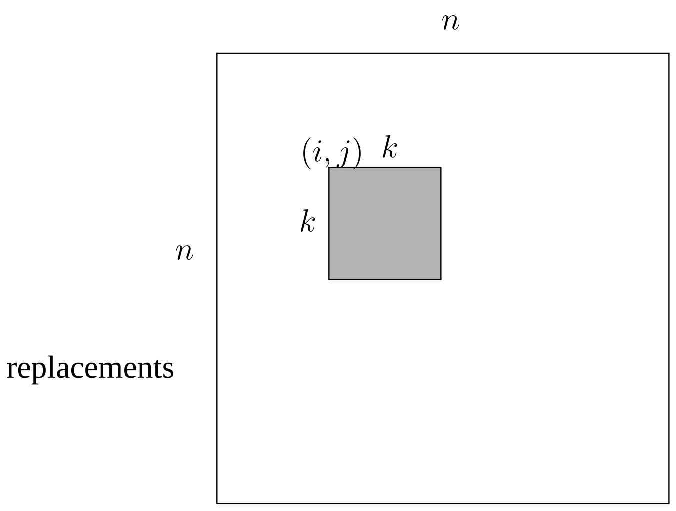

Figure 2: The output element b_{ij} is the "product" of all the a's in the shaded square.

the $(\boldsymbol{k} \times \boldsymbol{k})$-***filtered*** image is the $n \times n$ matrix

$$B = \begin{pmatrix} b_{00} & b_{01} & \cdots & b_{0(n-1)} \\ b_{10} & b_{11} & \cdots & b_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & b_{(n-1)(n-1)} \end{pmatrix},$$

where for $i, j = 0, 1, \ldots, n-1$,

$$b_{ij} = \bigotimes_{x=i}^{i+k-1} \bigotimes_{j=y}^{j+k-1} a_{xy} .$$

(For convenience, if $x \geq n$ or $y \geq n$, we assume that $a_{xy} = e$.)

Give an efficient algorithm to compute the $(k \times k)$-filter of an input matrix A. While analyzing your algorithm, do not treat k as a constant. That is, express your running time in terms of n and k. (For partial credit, solve the problem in one dimension.)

Problem 4. ***ViTo Design***

You are designing a new-and-improved digital video recorder, called ViTo. In the ViTo software, a television show i is represented as a triple: a ***channel number*** c_i, a ***start time*** s_i , and an ***end time*** e_i. The ViTo owner inputs a list of n shows to watch and for each show $i = 1, 2, \ldots, n$, assigns it a ***pleasure rating*** r_i. Since shows may overlap, and the ViTo can record only one show at a time, the ViTo should record the subset of the shows that maximize the aggregate "pleasure." Since the owner receives no pleasure from watching only part of a show, the ViTo never records partial shows. Design an efficient algorithm for the ViTo to select the best subset of shows to record.

Problem 5. *Growing a Graph*

We wish to build a data structure that supports a dynamically growing directed graph $G = (V, E)$. Initially, we have $V = \{1, 2, \ldots, n\}$ and $E = \emptyset$. The user grows the graph with the following operation:

- INSERT-EDGE(u, v): Insert a directed edge from vertex u to vertex v, that is, $E \leftarrow E \cup \{(u, v)\}$.

In addition, at any time the user can query the graph for whether two vertices are connected:

- CHECK-PATH(u, v): Return TRUE if a directed path from vertex u to vertex v exists; otherwise, return FALSE.

The user grows the graph until it is fully connected. Since the number of edges increases monotonically and the user never inserts the same edge twice, the total number of INSERT-EDGE operations is exactly $n(n-1)$. During the time that the graph is growing, the user performs m CHECK-PATH operations which are intermixed with the $n(n-1)$ INSERT-EDGE's. Design a data structure that can efficiently support any such sequence of operations.

Quiz 2 Solutions

Problem 1. ***Static Graph Representation***

Let $G = (V, E)$ be a sparse undirected graph, where $V = \{1, 2, \ldots, n\}$. For a vertex $v \in V$ and for $i = 1, 2, \ldots, \text{out-degree}(v)$, define v's ***ith neighbor*** to be the ith smallest vertex u such that $(v, u) \in E$, that is, if you sort the vertices adjacent to v, then u is the ith smallest.

Construct a representation of the graph G to support the following queries:

- DEGREE(v): returns the degree of vertex v.
- LINKED(u, v): output TRUE if an edge connects vertices u and v, and FALSE otherwise.
- NEIGHBOR(v, i): returns v's ith neighbor.

Your data structure should use asymptotically as little space as possible, and the operations should run asymptotically as fast as possible, but space is more important than time. Analyze your data structure in terms of both space and time.

Solution: We give a solution that uses $\Theta(E)$ space for the data structure and takes $\Theta(1)$ time for each of the three operations. Create a hash table that contains key (u, v) if $(u, v) \in E$ or key (u, i) if vertex u has an i-th neighbor. It is possible that for some u and $v = i$, vertex u has both an adjacent vertex v and an i-th neighbor. This is handled by storing satellite data with each record of the hash table. For the record with key (u, i) in the hash table, if u has a neighbor $v = i$, then indicate so using a bit in the record; if u has an i-th neighbor, then store the corresponding neighbor vertex index in the record. Also, for every vertex u that is connected to some other vertex, store its degree in the record for key $(u, 1)$.

Thus, for each vertex u in the first coordinate of the key, the hash table has at most $\text{degree}(u) + \text{degree}(u) = 2\,\text{degree}(u)$ entries. The total number of entries is thus at most $\sum_{u=1}^{n} 2\,\text{degree}(u) = 4|E|$. By using *perfect hashing*, and choosing a suitable hash function through a small number of random samplings (during data structure construction), we can make the lookup time $\Theta(1)$ and space requirement linear in the number of entries stored, i.e., $\Theta(E)$ (see CLRS, page 249, Corollary 11.12). We can use the same family of hash functions as in CLRS by converting each 2-tuple (u, v) into a distinct number $(u - 1)n + v$ in the range of $[1 \ldots n^2]$. The total space requirement of the degree array and hash table is thus $\Theta(V + E)$.

Then, DEGREE(v) looks up the key$(v, 1)$ in the hash table. If found, it returns the degree value stored in the record. Otherwise, it returns 0, since v is not adjacent to any vertex. This takes $\Theta(1)$ time. LINKED(u, v) looks up the key (u, v) in the hash table and returns TRUE if it exists and the associated bit in the record is set. This is $\Theta(1)$ time. NEIGHBOR(v, i) looks up the key (v, i) in the hash table – if it exists and a neighbor vertex is stored in the record, it returns its index. This is also $\Theta(1)$ time.

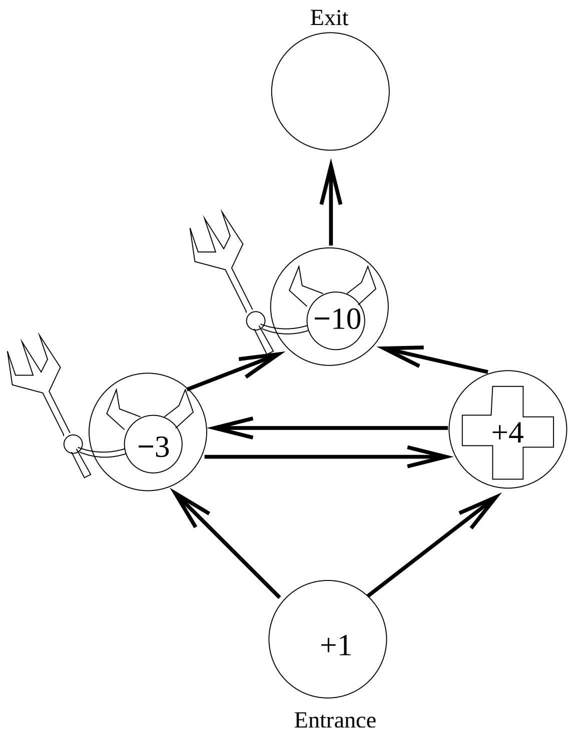

Figure 1: An example of a 1-admissible maze.

Problem 2. ***Video Game Design***

Professor Cloud has been consulting in the design of the most anticipated game of the year: ***Take-home Fantasy***. One of the levels in the game is a maze that players must navigate through multiple rooms from an entrance to an exit. Each room can be empty, contain a monster, or contain a life potion. As the player wanders through the maze, points are added or subtracted from her ***life points*** L. Drinking a life potion increases L, but battling a monster decreases L. If L drops to 0 or below, the player dies.

As shown in Figure 1, the maze can be represented as a digraph $G = (V, E)$, where vertices correspond to rooms and edges correspond to (one-way) corridors running from room to room. A vertex-weight function $f : V \to \mathbb{Z}$ represents the room contents:

- If $f(v) = 0$, the room is empty.
- If $f(v) > 0$, the room contains a life potion. Every time the player enters the room, her life points L increase by $f(v)$.
- If $f(v) < 0$, the room contains a monster. Every time the player enters the room, her life points L drop by $|f(v)|$, killing her if L becomes nonpositive.

The ***entrance*** to the maze is a designated room $s \in V$, and the ***exit*** is another room $t \in V$. Assume that a path exists from s to every vertex $v \in V$, and that a path exists from every vertex $v \in V$ to t.

The player starts at the entrance with $L = L_0$ life points, i.e. L_0 is the value of f for the entrance. The figure shows $L_0 = 1$.

Professor Cloud has designed a program to put monsters and life potions randomly into the maze, but some mazes may be impossible to safely navigate from entrance to exit unless the player enters with a sufficient number $L_0 > 0$ of life points. A path from s to t is "safe" if the player stays alive along the way, i.e., her life-points never become non-positive. Define a maze to be ***r-admissible*** if a safe path through the maze exists when the player begins with $L_0 = r$.

Help the professor by designing an efficient algorithm to determine the minimum value r so that a given maze is r-admissible, or determine that no such r exists. (For partial credit, solve the problem determining whether a maze is r-admissible for a given r.)

Solution:

We give an algorithm with running time $O(VE \lg r)$, where r is the minimum life needed at entry for the maze to be admissible. The problem is solved in two parts.

- An algorithm to determine if a given maze is admissible for a given r: This is done by using a modified version of Bellman-Ford. For every node u, this algorithm computes the maximum number of points $q[u]$ that the player can have on reaching u. Since the player will die if she reaches u with negative points, the value of $q[u]$ is either $-\infty$ (denoting that the player cannot reach u) or positive. Thus if $q[t]$ is positive (t is the exit node), then the graph is r-admissible.
- We know that the minimum r cannot be less than 1. Thus we use a combination of exponential and binary search to find the minimum value of r. We use the modified Bellman-Ford $\log r$ times to find the minimum r.

The running time of the algorithm is $O(VE \log r)$, where r is the minimum r, where the maze is r-admissible.

Determining Admissibility for a given r We use a modified version of Bellman-Ford algorithm. Given an r, for every node u we find the maximum (positive) number of points $q[u]$ the player can have when she reaches u. If $q[t]$ is positive, then the graph is r-admissible.

For each vertex $u \in V$, we maintain $p[u]$ which is a lower bound on $q[u]$. We initialize all the $p[u]$'s to $-\infty$, except the entrance, which is initialized to r. As we run the Bellman-Ford Algorithm and relax edges, the value of $p[u]$ increases until it converges to $q[u]$ (if there are no positive weight cycles). The important point to note is that reaching a node with negative points is as good as not reaching it at all. Thus, we modify $p[u]$ only it becomes positive, otherwise $p[u]$ remains $-\infty$. We change the relaxation routine to incorporate this as follows.

```
V-RELAX$(u, v)$
1  if $(u, v) \in E$
2      then if $((p[v] < p[u] + f[v])$ and $(p[u] + f[v] > 0))$
3      then $p[v] \leftarrow p[u] + f[v]$
4  $\pi[v] \leftarrow u$
```

After all the edges have been relaxed V times, if there are no negative weight cycles, all $p[u]$'s will have converged to the corresponding $q[u]$'s (the maximmum number of points you can have on reaching vertex u). If $q[t]$ is positive at this point, then the player can reach there with positive life points and thus the graph is r-admissible. If $p[t]$ is not positive, however, we relax all the edges one more time (just like Bellman-Ford). If $p[u]$ of any node changes, we have found a positive weight cycle which is reachable from s starting with r points. Thus the player can go around the cycle enough times to collect all the necessary points to reach t and thus the graph is r-admissible. If we dont fi nd a reachable positive weight cycle and $p[t]$ is $-\infty$, then the graph is not r admissible. The correctness of the algorithm follows from the correctness of Bellman-Ford, and the running time is $O(VE)$.

Finding the minimum r for which the graph is r-admissible Given the above sub-routine, we now fi nd the minimum r. We fi rst check if the graph is 1-admissible. If it is, we return 1 as the answer. If it is not, then we check if it is 2-admissible and then 4-admissible and so on. Thus on the ith step we check if the graph is 2^{i-1}-admissible. Eventually, we fi nd k such that the graph is not 2^{k-1}-admissible, but it is 2^k-admissible. Thus the minimum value of r lies between these two values. Then we binary search between $r = 2^{k-1}$ and $r = 2^k$ to fi nd the right value of r.

Analysis: The number of iterations is $k + O(\lg r) = O(\lg r)$, since $k = \lfloor \lg r \rfloor$. Thus you have to run Bellman-Ford $O(\lg r)$ times, and the total running time is $O(VE \lg r)$.

Alternate Solutions Some people visited nodes in DFS or BFS order starting from the exit, relaxing edges to fi nd the minimum number of points needed to get from any node u to the exit. The problem with this approach is that in the presence of positive weight cycles, the algorithm runs for $O(M(V + E))$ time, where M is the total sum of all monster points. This number can be big even if the real r is small. Some people did the same thing, except with Bellman-Ford instead of search, which gives a running time of $O(MVE)$. There were a couple of other clever solutions which ran in $O(V^2E$ time.

Problem 3. *Image Filtering*

Two-dimensional fi ltering is a common operation in vision and image processing. An image is represented as an $n \times n$ matrix of real values. As shown in Figure 2, the idea is to pass a $k \times k$ window across the matrix, and for each of the possible placements of the window, the fi lter computes the "product" of all the values in the window. The "product" is not typically ordinary

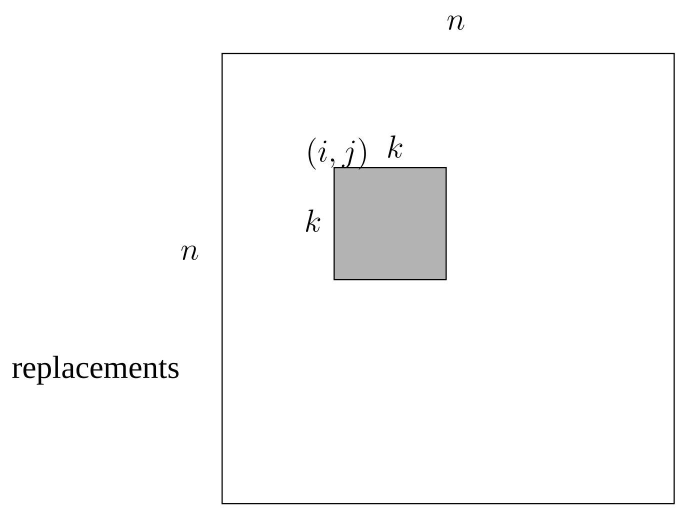

Figure 2: The output element b_{ij} is the "product" of all the a's in the shaded square.

multiplication, however. For this problem, we shall assume it is an associative and commutative binary operation $\otimes$ with identity element e, that is, $x \otimes e = e \otimes x = x$. For example, the product could be $+$ with identity element 0, $\times$ with 1, $\min$ with ∞, etc. Importantly, you may not assume that $\otimes$ has an inverse operation, such as $-$ for $+$.

To be precise, given an $n \times n$ image

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{pmatrix},$$

the ***(k × k)*-filtered** image is the $n \times n$ matrix

$$B = \begin{pmatrix} b_{00} & b_{01} & \cdots & b_{0(n-1)} \\ b_{10} & b_{11} & \cdots & b_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & b_{(n-1)(n-1)} \end{pmatrix},$$

where for $i, j = 0, 1, \ldots, n-1$,

$$b_{ij} = \bigotimes_{x=i}^{i+k-1} \bigotimes_{j=y}^{j+k-1} a_{xy} .$$

(For convenience, if $x \geq n$ or $y \geq n$, we assume that $a_{xy} = e$.)

Give an efficient algorithm to compute the $(k \times k)$-filter of an input matrix A. While analyzing your algorithm, do not treat k as a constant. That is, express your running time in terms of n and k. (For partial credit, solve the problem in one dimension.)

Solution:

We can solve the two-dimensional filtering problem in $\Theta(n^2)$ time by first reducing the problem to two one-dimensional filtering problems and then showing how a one-dimensional filter on n elements can be solved in $\Theta(n)$ time. We assume that $k \leq n$, since filtered values for $k > n$ are the same as for $k = n$. The $\Theta(n^2)$-time algorithm is optimal, since there are n^2 values to compute.

Define the intermediate matrix C by

$$C = \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0(n-1)} \\ c_{10} & c_{11} & \cdots & c_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(n-1)0} & c_{(n-1)1} & \cdots & c_{(n-1)(n-1)} \end{pmatrix} ,$$

where for $i, j = 0, 1, \ldots, n-1$,

$$c_{ij} = \bigotimes_{y=j}^{j+k-1} a_{iy} ,$$

that is, C is the one-dimensional k-filter on each row of A. We have

$$\begin{aligned} b_{ij} &= \bigotimes_{x=i}^{i+k-1} \bigotimes_{y=j}^{j+k-1} a_{xy} \\ &= \bigotimes_{x=i}^{i+k-1} c_{xj} , \end{aligned}$$

and thus B is just the one-dimensional k-filter on each column of C.

It remains to devise an efficient method to compute one-dimensional k-filters. The naive algorithm takes $\Theta(kn)$ time to solve the one-dimensional problem for an array of length n. Using this one-dimensional algorithm to solve the two-dimensional problem costs $\Theta(kn^2)$ to compute C from A and another $\Theta(kn^2)$ to compute B from C, resulting in $\Theta(kn^2)$ overall. Many students found a way to compute the one-dimensional problem in $\Theta(n \lg k)$, resulting in a two-dimensional solution of $\Theta(n^2 \lg k)$. In fact, as some students discovered, the one-dimensional problem can be solved in $\Theta(n)$ time, leading to a two-dimensional solution of $\Theta(n^2)$.

The $\Theta(n)$-time solution for the one-dimensional problem works as follows. Let the input array be $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ and the k-filtered output array be $B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$, where

$$b_i = \bigotimes_{x=i}^{i+k-1} a_x .$$

Assume without loss of generality that n is evenly divisible by k, since otherwise, we can pad the end of a with identity elements e to make n a multiple of k without more than doubling n.

The idea is to divide the arrays into blocks of k elements. Observe that any window of k elements starting at a given location i consists of the product of a suffix of one block and a prefix of the next block. Thus, we compute prefixes and suffixes of each block as follows. For $i = 0, 1, \ldots, n-1$, define

$$f_i = \begin{cases} e & \text{if } i \bmod k = 0, \\ f_{i-1} \otimes a_{i-1} & \text{otherwise;} \end{cases}$$

and for $i = n-1, n-2, \ldots, 0$, define

$$g_i = \begin{cases} a_i & \text{if } (i+1) \bmod k = 0, \\ a_i \otimes f_{i+1} & \text{otherwise.} \end{cases}$$

These two arrays can be computed in $\Theta(n)$ time, and then we obtain the output array by computing, for $i = 0, 1, \ldots, n-1$,

$$b_i = g_i \otimes f_{i+k} \ ,$$

which also takes $\Theta(n)$ time.

As an example, consider a one-dimensional 4-filter:

$$\begin{array}{lclcl}
b_0 & = & (a_0 \otimes a_1 \otimes a_2 \otimes a_3) & = & g_0 \otimes f_4 \\
b_1 & = & (a_1 \otimes a_2 \otimes a_3) \otimes (a_4) & = & g_1 \otimes f_5 \\
b_2 & = & (a_2 \otimes a_3) \otimes (a_4 \otimes a_5) & = & g_2 \otimes f_6 \\
b_3 & = & (a_3) \otimes (a_4 \otimes a_5 \otimes a_6) & = & g_3 \otimes f_7 \\
b_4 & = & (a_4 \otimes a_5 \otimes a_6 \otimes a_7) & = & g_4 \otimes f_8 \\
b_5 & = & (a_5 \otimes a_6 \otimes a_7) \otimes (a_8) & = & g_5 \otimes f_9 \\
b_6 & = & (a_6 \otimes a_7) \otimes (a_8 \otimes a_9) & = & g_6 \otimes f_{10} \\
\vdots & & & &
\end{array}$$

Problem 4. ***ViTo Design***

You are designing a new-and-improved digital video recorder, called ViTo. In the ViTo software, a television show i is represented as a triple: a ***channel number*** c_i, a ***start time*** s_i , and an ***end time*** e_i. The ViTo owner inputs a list of n shows to watch and for each show $i = 1, 2, \ldots, n$, assigns it a ***pleasure rating*** r_i. Since shows may overlap, and the ViTo can record only one show at a time, the ViTo should record the subset of the shows that maximize the aggregate "pleasure." Since the owner receives no pleasure from watching only part of a show, the ViTo never records partial shows. Design an efficient algorithm for the ViTo to select the best subset of shows to record.

Solution: Assume ViTo has enough harddisk space to record any subset of the programs. We use a dynamic programming approach to the problem.

First let's show the optimal substructure. Let $show_i$ denote the triple (c_i, s_i, e_i), $shows$ denote the whole list of shows $show_1, ..., show_n$, and $shows(t)$ denote the subset of shows $show_j$ such that $e_j < t$, i.e. all the shows with ending times before time t. Consider an optimal solution

$show_{i_1}, show_{i_2}, ..., show_{i_k}$. Then $show_{i_1}, show_{i_2}, ..., show_{i_{k-1}}$ must be an optimal solution to the subproblem $shows(s_{i_k})$ since if not, we could cut and paste a better solution to this subproblem, append $show_{i_k}$ to it, and get a better solution than the optimal one.

Sort the shows by ending time, so $e_i \leq e_j$ for $i < j$ and if $e_i = e_j$ then $s_i < s_j$. (if two or more shows start and end at the same time, then we can just keep the one with the maximum pleasure r_i, breaking ties arbitrarily). This can be done with counting sort in linear time: with n shows, there are only a maximum of $2n$ possible starting/ending times, and there are only 24 hours in a day, i.e. a limited (constant) amount of time, therefore the range of possible times is $O(n)$. Relabel the shows so that $show_1, ..., show_n$ are in sorted order. Let $shows^i$ be the list of shows up to and including the i-th show, i.e. $shows^i = show_1, show_2, ..., show_i$.

Note that $shows(t)$, the subset of shows $show_j$ such that $e_j < t$, is equal to $show^{k(t)}$ for some $k(t)$. The aggregate pleasure $p(i)$ of an optimal solution for shows in $show^i$ is:

$$p(i) = \begin{cases} 0 & \text{if } i < 1; \\ \max\{p(k(s_i)) + r_i, p(i-1)\} & \text{otherwise.} \end{cases}$$

The optimal solution is:

$$record(i) = \begin{cases} \{\} & \text{if } i < 1; \\ record(k(s_i)) \cup \{show_i\} & \text{if } p(k(s_i)) + r_i > p(i-1); \\ record(i-1) & \text{otherwise.} \end{cases}$$

Running Time = time to sort the shows + time to fi nd $p(n)$ = $O(n)$.

Problem 5. *Growing a Graph*

We wish to build a data structure that supports a dynamically growing directed graph $G = (V, E)$. Initially, we have $V = \{1, 2, \ldots, n\}$ and $E = \emptyset$. The user grows the graph with the following operation:

- INSERT-EDGE(u, v): Insert a directed edge from vertex u to vertex v, that is, $E \leftarrow E \cup \{(u, v)\}$.

In addition, at any time the user can query the graph for whether two vertices are connected:

- CHECK-PATH(u, v): Return TRUE if a directed path from vertex u to vertex v exists; otherwise, return FALSE.

The user grows the graph until it is fully connected. Since the number of edges increases monotonically and the user never inserts the same edge twice, the total number of INSERT-EDGE operations is exactly $n(n-1)$. During the time that the graph is growing, the user performs m CHECK-PATH operations which are intermixed with the $n(n-1)$ INSERT-EDGE's. Design a data structure that can effi ciently support any such sequence of operations.

Solution: To solve this problem, we keep an $n \times n$ ***transitive-closure*** matrix T that keeps track of whether there exists a directed path between each pair of vertices. We give an algorithm such that each CHECK-PATH operation takes $O(1)$ time, and a sequence of $n(n-1)$ INSERT-EDGE operations take a total of $O(n^3)$ time in the worst case. Combining these bounds, any sequence of m CHECK-PATH and $n(n-1)$ INSERT-EDGE operations takes a total of $O(n^3 + m)$ time. We later improve the data structure to deal with the case in which m is small, to get a total time of $O(\min\{n^3 + m, n^2 m\})$.

Our data structure maintains a transitive-closure matrix $T = (t_{uv})$ such that

$$t_{uv} = \begin{cases} 1 & : \text{ if there exists a directed path from } u \text{ to } v \text{ in } G \,, \\ 0 & : \text{ otherwise} \,. \end{cases}$$

The matrix T is similarly to an adjacency matrix, except that instead of keeping track of the existence of edges $u \to v$, it keeps track of paths $u \leadsto v$. Note that the 1's in the u-th row correspond to all the vertices that u can reach, and the 1's in the u-th column correspond to all the vertices that can reach u. We initialize $t_{uu} = 1$ because there is a directed path (of no edges) from a vertex to itself.

Given T, the implementation of CHECK-PATH(u, v) is straightforward: just query the value of t_{uv}. This query can be performed in constant time, so CHECK-PATH runs in constant time. Pseudocode for CHECK-PATH is given below:

```
CHECK-PATH(u, v)
1  if $t_{uv} = 1$
2      then return TRUE
3      else return FALSE
```

The tricky part of the data structure is maintaining the matrix T on an INSERT-EDGE(u, v). When the edge (u, v) is added, we check each vertex x. If x can reach u, and x cannot already reach v, then we update the matrix to indicate that u can reach all the vertices that v can reach (in addition to the vertices that it could reach before). In other words, let R_w be the set of vertices that the vertex w can reach (i.e., the set of indices of 1's in the w-th row in T). Then when adding (u, v), we iterate over all $x \in V$. For each x such that $u \in R_x$ and $v \notin R_x$, we set $R_x \leftarrow R_x \cup R_v$. Pseudocode for INSERT-EDGE is given below:

```
INSERT-EDGE(u, v)
1  for $x \leftarrow 1$ to $n$
2      do if $t_{xu} = 1$ and $t_{xv} = 0$    ▷ $x$ can reach $u$ but not $v$
3            then for $y \leftarrow 1$ to $n$
4                     do $t_{xy} \leftarrow \max\{t_{xy}, t_{vy}\}$    ▷ If $v \leadsto y$, add $x \leadsto y$ to $T$
```

Correctness. The following theorem proves that our algorithm is correct.

Theorem 1*The* INSERT-EDGE *operation maintains the invariant that $t_{xy} = 1$ iff there exists a directed path from x to y in G.*

Proof. We prove by induction on INSERT-EDGE operations. That is, we assume that the transitive-closure matrix is correct up to (before) a particular INSERT-EDGE(u, v) operation, and then we show that it is correct after that operation. We do not have to prove anything for CHECK-PATH as that operation does not modify the matrix.

First, suppose that $x \rightsquigarrow y$ before the edge (u, v) is added. Then $t_{xy} = 1$ before the INSERT-EDGE operation. The only place t_{xy} can be updated is in line 4, and if so, it keeps its value of 1. This behavior is correct because adding edges cannot destroy a path.

Suppose that $x \not\rightsquigarrow y$ before the edge (u, v) is added, but $x \rightsquigarrow y$ after the edge is added. Therefore, it must be the case the path from x to y uses the edge (u, v). Therefore, we have $x \rightsquigarrow u$ and $v \rightsquigarrow y$ before the INSERT-EDGE(u, v), so by assumption $t_{xu} = 1$ and $t_{vy} = 1$. Furthermore, it must also be true that $x \not\rightsquigarrow v$ before the addition of (u, v), or we would violate the assumption that $x \not\rightsquigarrow y$. Thus, we reach line 4, and $t_{xy} \leftarrow t_{vy} = 1$.

The last case to consider is the one in which $x \not\rightsquigarrow y$ after the operation. We need to make sure that we have $t_{xy} = 0$ in this case. If there is no path, then $t_{xy} = 0$ before the addition of (u, v). Moreover, there is no path that uses (u, v), so either $t_{xu} = 0$ or $t_{vy} = 0$. If $t_{xu} = 0$, we don't enter the loop in line 2, so the update in line 4 is not performed. If $t_{xu} = 1$, then $t_{vy} = 0$, and line 4 sets the value of $t_{xy} \leftarrow 0$. □

Analysis. Now let us examine the runtime of our algorithm. Each CHECK-PATH operation is just a table lookup, which takes $O(1)$ time. The analysis of INSERT-EDGE is slightly more complicated. We can trivially bound the worst-case cost of INSERT-EDGE to $O(n^2)$ because we have nested for loops, each iterating over n items and doing constant work in line 4. We can show a tighter bound on a sequence of $n(n-1)$ INSERT-EDGE operations using aggregate analysis. Each time INSERT-EDGE runs, the outer loop (line 1) executes, performing the constant work from line 2 on n items. Thus, the contribution of the outer loop totals to $O(n^3)$. The inner loop (line 3) executes only when $t_{xv} = 0$, and when it fi nishes, $t_{xv} = 1$. Thus, for a particular vertex x, the inner loop can be executed at most n times (actually, $n - 1$, as we begin with $t_{xx} = 1$). Since there are n vertices, the inner loop can run at most n^2 times in total for a total $O(n^3)$ work in the worst case. Thus, the total runtime for $n(n-1)$ INSERT-EDGEs and m CHECK-PATHs is $O(n^3 + m)$.

Slight improvements. There is another data structure with $O(1)$ cost for each INSERT-EDGE but $O(n^2)$ for each CHECK-PATH. To implement this data structure, we can use an adjacency list: we keep an array $A[1..n]$ of size n indexed by vertex and keep a (linked) list of all the outgoing edges from the corresponding vertex. To perform an INSERT-EDGE(u, v), simply insert v at the front of $A[u]$ in $O(1)$ time. (Note that edges are inserted only once, so we do not have to worry about v being present in the list already.) To perform CHECK-PATH(u, v), we run some sort of search, let's say breadth-fi rst search, starting at vertex u. If v is encountered at any point along the search, return TRUE. If not, return FALSE. Correctness of this algorithm should be somewhat obvious. BFS takes $O(V + E) = O(n^2)$ time. Thus, the total runtime of the sequence of operations is $O(n^2 + n^2 m)$.

This data structure is probably worse than the one given above. It seems safe to assume that $m \gg n$ as you probably query each vertex at least once. Assuming that $m \gg n^2$ is also reasonable. If you do not want make these assumptions, and you know m ahead of time, you can choose the appropriate data structure.

It turns out that we can also combine both data structures to achieve the better of the two bounds even if m is not known ahead of time. To do this, we use the adjacency-list data structure until there have been n queries. Once we reach the n-th query (CHECK-PATH), we construct the transitive-closure matrix and then use the matrix for all subsequent operations. Construction of the matrix takes $O(n^3)$ time by simply running BFS from each vertex u and marking each reachable vertex v by $t_{uv} \leftarrow 1$. Thus, if $m \leq n$, we use only the adjacently list, to get a total runtime of $O(n^2 m)$. If $m \geq n$, we fi rst use the adjacency list for a total of $O(n^3)$ work, then we transform to the transitive-closure matrix in $O(n^3)$ time, then we use the matrix for all subsequent operations, which comes to a total of $O(n^3 + m)$. Thus, this data structure achieves a runtime of $O(\min\{n^3 + m, n^2 m\})$ in the worst case.