CS 124 Data Structures and Algorithms — Spring 2015
List of problems

1 Problem 1

Indicate for each pair of expressions (A, B) in the table below the relationship between A and B. Your answer should be in the form of a table with a "yes" or "no" written in each box. For example, if A is $O(B)$, then you should put a "yes" in the first box. If the base of a logarithm is not specified, you should assume it is base-2.

A	B	O	o	Ω	ω	Θ
$\log_2 n$	$\log_3 n$					
$\log\log n$	$\sqrt{\log n}$					
$2^{\log^7 n}$	n^7					
$n^2 2^n$	3^n					
$n!$	n^n					
$\log(n!)$	$\log(n^n)$					
$(n^2)!$	n^n					
$(n!)^2$	n^n					

Solution:

A	B	O	o	Ω	ω	Θ
$\log_2 n$	$\log_3 n$	y	n	y	n	y
$\log\log n$	$\sqrt{\log n}$	y	y	n	n	n
$2^{\log^7 n}$	n^7	n	n	y	y	n
$n^2 2^n$	3^n	y	y	n	n	n
$n!$	n^n	y	y	n	n	n
$\log(n!)$	$\log(n^n)$	y	n	y	n	y
$(n^2)!$	n^n	n	n	y	y	n
$(n!)^2$	n^n	n	n	y	y	n

To show $f(n) = O(g(n))$, it suffices (though is not equivalent) to show $\lim_{n\to\infty} f(n)/g(n) < \infty$. Similarly for $f(n) = \Omega(g(n))$ it suffices (though is not equivalent) to show $\lim_{n\to\infty} f(n)/g(n) > 0$.

First row: Note $\log_2 n = \frac{1}{\log_2 3} \cdot \log_3 n$. Thus their ratio is a constant (and is thus in the limit is both bounded away from 0 and away from ∞).

Second row: Perform a change of variables $m = \sqrt{\log n}$. Then we are comparing $\log(m^2) = 2\log m$ versus m. Then $\lim_{m\to\infty}(\log m)/m = 0$.

Third row: $2^{\log^7 n} = (2^{\log n})^{\log^6 n} = n^{\log^6 n}$. Thus the ratio of these two functions is $n^{(\log n)^6 - 7}$, which goes to ∞ as $n \to \infty$.

Fourth row: The ratio B/A is $(3/2)^n/n^2$. As we saw in class, exponential functions are always $\omega(\cdot)$ of any polynomial.

Fifth row: Clearly $n! \le n^n$, so $n! = O(n^n)$. How about $\Omega(\cdot)$?. Let's say n is even (the case of odd n is similar). Then

$$n! = \underbrace{1 \cdot 2 \cdots \frac{n}{2}}_{\le \left(\frac{n}{2}\right)^{\frac{n}{2}}} \cdot \underbrace{\left(\frac{n}{2}+1\right) \cdots n}_{\le n^{\frac{n}{2}}}$$

Thus $n! \le (n/2)^{n/2} \cdot n^{n/2} = n^n/2^{n/2}$. Thus the ratio B/A is at least $2^{n/2}$, which goes to ∞ as $n \to \infty$. It is also possible to look up things like Stirling's approximation and deduce the answer for this row based on that, but note that it is not necessary, as seen above (Stirling's approximation is much more precise than what's needed here).

Sixth row: Using the inequalities $n! \le n^n$ and $n! \ge (n/2)^{n/2}$, we have that $(1/2)n \log n - n/2 \le \log(n!) \le n \log n$. Thus $\log(n!) = \Theta(n \log n)$, which is the same as $\log(n^n)$.

Seventh row: $(n^2)! \ge (n^2/2)^{n^2/2} = n^{n^2}/2^{n^2/2}$. Thus the ratio A/B is $n^{n^2-n}/2^{n^2/2}$. To see that this goes to ∞, you could for example rewrite $n = 2^{\log n}$, so then $n^{n^2-n} = 2^{n^2 \log n - n \log n}$. Then $A/B = 2^{n^2 \log n - n^2/2 - n \log n}$, and we see that the $n^2 \log n$ term in the exponent asymptotically dominates the terms being subtracted off as $n \to \infty$, so $\lim_{n\to\infty} A/B = \infty$.

Eighth row: It turns out $A = \omega(B)$, but just using the approximation $n! \ge (n/2)^{n/2}$ wouldn't cut it here. That's because we would say $(n!)^2 \ge ((n/2)^{n/2})^2 = (n/2)^n$, but this is clearly not dominating n^n. Instead you could use the approximation (assuming n is divisible by 3; the other cases are handled similarly): $n! \ge (n/3)^{2n/3}$. This is because the last $(2/3)$rds of the numbers being multiplied to form $n!$ are at least $n/3$. Then $(n!)^2 \ge ((n/3)^{2n/3})^2 = (n/3)^{4n/3}$, which grows much faster than n^n.

2 Problem 2

For all of the problems below, when asked to give an example, you should give a function mapping positive integers to positive integers. (No cheating with 0's!)

- Show that if f is $o(g)$, then fh is $o(gh)$ for any function h.
- Give a proof or a counterexample: if f is not $O(g)$, then f is $\Omega(g)$.
- Find (with proof) a function f such that $f(2n)$ is $O(f(n))$.

- Find (with proof) a function f such that $f(n)$ is $o(f(2n))$.
- Show that for all $\epsilon > 0$, $\log n$ is $o(n^\epsilon)$.

Solution:

- If f is $o(g)$, then for any $c > 0$, $f(n) < cg(n)$ for large enough n. Then, $f(n)h(n) < cg(n)h(n)$ as well, so fh is $o(gh)$.
- Consider $f(n) = n^2$ if n is odd, and $n = 1$ if n is even. Then, f is not $O(n)$, but f is also not $\Omega(n)$.
- Note that $\log_2 2n = \log_2 n + 1$ is $O(\log_2 n)$ since $\log_2 n + 1 < 2\log_2 n$ for $n > 2$.
- Note that 2^n is $o(2^{2n})$, since for any constant c, $2^n < c2^{2n}$ for $n > \log_2 \frac{1}{c}$.
- We want to show that for any constant c, $\log n < cn^\epsilon$ for large enough n. This is equivalent to showing that $\lim_{n\to\infty} \frac{\log n}{n^\epsilon} = 0$. Using L'Hopitals rule,

$$\lim_{n\to\infty} \frac{\log n}{n^\epsilon} = \lim_{n\to\infty} \frac{1/n}{\epsilon n^{\epsilon-1}} = \frac{1}{\epsilon}\lim_{n\to\infty} n^{-\epsilon} = 0.$$

3 Problem 3

InsertionSort is a simple sorting algorithm that works as follows on input $A[0], \ldots, A[n-1]$:

```
InsertionSort(A):
  for i = 1 to n-1
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j - 1
```

Show that for any function $T = T(n)$ satisfying $T(n) = \Omega(n)$ and $T(n) = O(n^2)$, there exists an infinite sequence of inputs $\{A_n\}_{n=1}^\infty$ such that (1) A_n is an array of length n, and (2) if $f(n)$ denotes the running time of InsertionSort on A_n, then $f(n) = \Theta(T(n))$.

Solution: First, note that there exist constants c_1, N_1 such that $T(n) \le c_1 n$ for all $n \ge N_1$ and constants c_2, N_2 such that $T(n) \ge c_2 n^2$ for all $n \ge N_2$. Let $N = \max\{N_1, N_2\}$; then $c_1 n \le T(n) \le c_2 n^2$ for all $n \ge N$. Define $S(n) = T(n)/c_2$, so that $S(n) \le n^2$ for all $n \ge N$.

Our construction is as follows for $n \ge N$: let A_n be the array

$$\Big[r, r-1, r-2, \ldots, 1, r+1, r+2, \ldots, n\Big]$$

for $r = \lfloor\sqrt{S(n)}\rfloor$. Because $S(n) \le n^2$, this is a valid construction.

Now, we prove that this construction works. If we examine InsertionSort, we note that the runtime is $\Theta(n) + \Theta(\#$ of comparisons of adjacent elements). However, for each value of i, the number of comparisons is exactly one more than the number of swaps. Consequently, the runtime is also $\Theta(n) + \Theta(\#$ of swaps).

Consequently, it suffices to count the number of swaps in A_n.[1] Here, note that the only way for two elements i, j to switch order in the array is for them to swapped; meanwhile, any two elements i, j which are out of order will be swapped at most once, since once they are in the proper order, comparing them will never lead to a swap.

In our construction, for $i \leq r$, there are exactly $r - 1$ elements j such that i, j are out of order. Thus, our runtime is

$$\Theta(n + r(r-1)) = \Theta(r^2 + n) = \Theta(S(n) + n) = \Theta(T(n)).$$

4 Problem 4

Give asymptotic bounds for $T(n)$ in each of the following recurrences. Hint: You may have to change variables somehow in the last one.

- $T(n) = 2T(n/2) + n^3$.
- $T(n) = 5T(n/4) + n$.
- $T(n) = 9T(n/3) + n^2$.
- $T(n) = T(\sqrt{n}) + 1$.

Solution:

- Using the master theorem, $a = 2$, $b = 2$, $k = 3$, $b^k = 8 > a$, so $T(n) = \Theta(n^3)$.
- Using the master theorem, $a = 5$, $b = 4$, $k = 1$, $b^k = 4 < a$, so $T(n) = \Theta(n^{\log_4 5})$.
- Using the master theorem, $a = 9$, $b = 3$, $k = 2$, $b^k = 4 = a$, so $T(n) = \Theta(n^2 \log n)$.
- Set $n = 2^{2^m}$, so the recurrence is

$$T(2^{2^m}) = T(2^{2^m/2}) + 1 = T(2^{2^{m-1}}) + 1 = T(2^{2^{m-2}}) + 2 = \cdots = T(2) + m.$$

(To make this formal, use induction on m.) Since $m = \log_2 \log_2 n$, $T(n) = \Theta(\log_2 \log_2 n)$.

[1] Also known as the number of *inversions*.

5 Problem 5

It is known that every integer $n > 1$ can be uniquely factored as a product of primes. For example, $4 = 2 \times 2$, $6 = 2 \times 3$, and $90 = 2 \times 3 \times 3 \times 5$. Let $p(n)$ be the number of *distinct* prime divisors of n, so $p(6) = 2$ but $p(4) = 1$.

(a) Show that $p(n) = O(\log n)$.

(b) Show that $p(n) = O(\frac{\log n}{\log \log n})$.

(c) It is a fact, which you may assume without proof, that there are $\Theta(t/\log t)$ primes between 1 and t. Use this fact to show that it is *not* true that $p(n) = o(\frac{\log n}{\log \log n})$.

Solution:

(a) Let $p_1^{e_1} \cdots p_k^{e_k}$ be the prime factorization of n. Then,

$$n = p_1^{e_1} \cdots p_k^{e_k} \geq p_1 \cdots p_k \geq 2^k,$$

so $p(n) = k < \log_2 n$.

(b) We use the same idea as in part a, but a smarter bound. Namely, if $p_1 < p_2 < \cdots < p_k$, then $p_i \geq i + 1$ for all i, so

$$n = p_1^{e_1} \cdots p_k^{e_k} \geq p_1 \cdots p_k \geq (k+1)! \geq \left(\frac{k}{2}\right)^{k/2}.$$

Thus, for the function $f(x) = x^x$, we have shown that $f(p(n)/2) \leq n$. First, observe that $f(x)$ is a strictly increasing function for $x \geq 1$, since if $y > x \geq 1$ then $f(y)/f(x) = y^{y-x}(y/x)^x > (y/x)^x > 1$.

Now, since f is increasing, if we can show that $f(s) > n$ for some particular s, then this combined with $f(p(n)/2) \leq n$ would imply that $p(n)/2 \leq s$, i.e. $p(n) \leq 2s$.

Ultimately we want to show $p(n) = O(\log n/\log \log n)$, so we will choose $s = 2 \log n/ \log \log n$. Note

$$s^s > n \iff s \log s > \log n$$

since $\log(\cdot)$ is a strictly increasing function. Then we find for our choice of s,

$$\begin{aligned} s \log s &= 2 \cdot \frac{\log n}{\log \log n} \cdot (\log \log n + 1 - \log \log \log n) \\ &> 2 \cdot \frac{\log n}{\log \log n} \cdot (\frac{2}{3} \log \log n) \text{ (for } n \text{ sufficiently large since } \log \log \log n = o(\log \log n)) \\ &> \log n \end{aligned}$$

Thus $f(s) > n$ for n sufficiently large, implying $p(n) \leq \frac{4 \log n}{\log \log n}$ for n sufficiently large.

(c) First let us dissect what the problem is asking us to prove. We are asked to show that $p(n)$ is *not* $o(f(n))$ (where $f(n) = \log n / \log\log n$). What does it mean for $p(n)$ to be $o(f(n))$? It means that $\lim_{n\to\infty} p(n)/f(n) = 0$. Just rewriting this in terms of the definition of limit, this means

$$\forall \epsilon > 0\ \exists N\ \forall n \geq N \quad \frac{p(n)}{f(n)} < \epsilon$$

We are asked to show the negation of the above:

$$\neg\left(\forall \epsilon > 0\ \exists N\ \forall n \geq N \quad \frac{p(n)}{f(n)} < \epsilon\right) \tag{1}$$

In general, if $P(x)$ is a logical proposition that depends on x, then

$$\neg\forall x\ P(x) = \exists x\ \neg P(x)$$

and

$$\neg\exists x\ P(x) = \forall x\ \neg P(x)$$

Thus,

$$(1) = \exists \epsilon > 0\ \forall N\ \exists n \geq N \quad \frac{p(n)}{f(n)} \geq \epsilon.$$

In words, no matter how far out you go (i.e. no matter how big N is), there's always a number farther out (the number $n \geq N$) such that $p(n) \geq \epsilon \cdot f(n)$. Here $\epsilon > 0$ is some fixed constant that doesn't depend on n. In other words, we would like to show the existence of an infinite sequence $(a_t)_{t=1}^{\infty}$ of integers such that $p(a_t) \geq \epsilon \cdot f(a_t)$ for all t in the sequence.

Consider the sequence a_t, where a_t is the product of all the primes between 1 and t. Since there are at most $C\frac{t}{\log t}$ primes between 1 and t for some constant C, we have $a_t \leq t^{Ct/\log t}$. Also there are at least $c\frac{t}{\log t}$ primes between 1 and t for some other constant c. At least half these primes (the larger half) are all at least $(c/2)\frac{t}{\log t}$. Thus we also have

$$a_t \geq ((c/2)\frac{t}{\log t})^{(c/2)\frac{t}{\log t}}$$

implying

$$\begin{aligned}
\log a_t &\geq (c/2)\frac{t}{\log t} \cdot [\log t - \log((c/2)\log t)] \\
&\geq (c/2)\frac{t}{\log t} \cdot (1/2)\log t \text{ for } t \text{ large since } \log((c/2)\log t) = o(\log t) \\
&= (c/4)t
\end{aligned}$$

Hence, $(c/4)t \leq \log a_t \leq Ct$, We also have $p(a_t) \geq c\frac{t}{\log t}$. so

$$\frac{\log a_t}{\log\log a_t} \leq \frac{Ct}{\log((c/4)t)} \leq \alpha\frac{t}{\log t}$$

for some constant α. We conclude that

$$p(a_t) \geq c\frac{t}{\log t} \geq \frac{c\log a_t}{\alpha\log\log a_t}.$$

Hence, for any constant N, there exists $n > N$ such that $p(n) \geq \frac{c}{\alpha}\left(\frac{\log n}{\log\log n}\right)$, so $p(n)$ cannot be $o(\log n/\log\log n)$ (take ϵ above to be c/α).

Remark 1. Another approach is to let a_t be the product of the first t primes. This also works. One can use the fact given in the problem statement to show $(C_1 \cdot t\log t)^t \leq a_t \leq (C_2 \cdot t\log t)^t$ for some constants C_1, C_2 for t sufficiently large. We also of course have $p(a_t) = t$ (by definition of a_t). Then, for t sufficiently large

$$\frac{\log(a_t)}{\log\log(a_t)} \leq \frac{t\log(C_1 \cdot t\log t)}{\log(t\log(C_2 \cdot t\log t))} \leq \frac{t\log t + \log(C_1\log t)}{\log t} \leq \frac{(1+C_3)t\log t}{\log t} = (1+C_3)t$$

with the last inequality using that $\log(C_1\log t) = O(t\log t)$ (in fact it is even $o(\cdot)$). Thus the ratio $p(a_t)/f(a_t)$ is at least $\epsilon = 1/(C_3+1)$ for all sufficiently large t.

6 Programming Problem

Solve "Problem B - Implications" on the programming server; see the "Problem Sets" part of the course web page for the link. **Hint:** If A is the adjacency matrix of a graph, when is $(A^2)_{i,j}$ non-zero? How about $(A^3)_{i,j}$?

Solution: Let G be the graph of implications. Each vertex is a logical statement, and we have a directed edge (A, B) if statement A implies statement B. We are told in the problem statement that G is a DAG (directed acyclic graph). What we are asked to find is what is usually called the *transitive reduction* of G. That is, a graph H on the same vertex set such that (1) the number of edges of H is minimum, and (2) for all vertices i, j, we have that i has a path to j in H iff it also does in G. Let us define $TC(G)$, the *transitive closure* of G, to be the graph on the same vertex set such that (i, j) is in $TC(G)$ iff i has a path to j in G.

Thus what we are asked to find is a graph H with a minimum number of edges satisfying $TC(H) = TC(G)$. We will prove that there is a unique H satisfying that $TC(H) = TC(G)$ such that no subgraph of H also satisfies this. This is even stronger than saying there's a unique H with the minimum number of edges (if H has the right transitive closure and the minimum number of edges, then if a subgraph H' of H also has $TC(H') = TC(G)$ that would violate H having the minimum number of edges).

Theorem 1. *If G is a DAG, then there is a unique H with $TC(H) = TC(G)$ such that no subgraph of H also satisfies this.*

Proof. For the sake of contradiction, suppose there were two different graphs H, H' with this property. Since they are different and neither is a subgraph of the other, there is an edge $e = (x, y)$ in H which is not in H'. Then since they have the same transitive closure, H' must also have a path from x to y, and this path passes through at least one intermediate node z. Thus H' has paths both from x to z and z to y. Thus, again since they have the same transitive closure, H must have a path P_{xz} from x to z and P_{zy} from z to y. The edge e cannot appear in P_{xz}, since that would mean y has a path to z, which combined with the fact that P_{zy} is a path from z to y, means that H has a cycle (and we said G is acyclic). Similarly e cannot appear in P_{zy}. Thus there is a path from x to y, through z, which does not use the edge e at all. Therefore the subgraph of H with e removed has the same transitive closure, contradicting our assumption that no subgraph of H satisfies this. □

Because of the above theorem, it suffices to find a subgraph of G with the same transitive closure as G such that no subgraph of it also does. To do this, we remove every edge (u, v) from G for which there is already some other path from u to v. Once all these edges are removed, then by definition the resulting graph H has the same transitive closure as G and no subgraph also does, and thus it must be the minimizer by the above theorem. How might we go about quickly identifying such edges?

The key is that the "other path from u to v" mentioned above is a path of length at least 2. Thus we are trying to identify edges (u, v) for which there is also path of length at least 2 from u to v in G, and these are precisely the edges we should remove. Below we'll show two different approaches to identify such edges. Both are fast enough to get full credit.

Approach 1: Let A be the adjacency matrix of G. Then

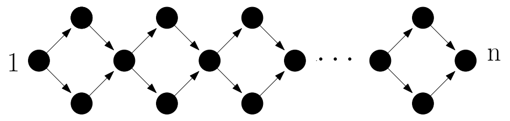
$$(A^2)_{u,v} = \sum_{w=1}^{n} A_{u,w} \cdot A_{w,v}$$

Each $A_{u,w} \cdot A_{w,v}$ product is 1 if the path $u \to w \to v$ exists in the graph and is 0 otherwise. Thus $(A^2)_{u,v}$ is the number of paths of length exactly 2 from u to v. Observing this, one might guess that $(A^k)_{u,v}$ is the number of paths of length exactly k from u to v, and indeed this is true and can be proven by induction on k.

Now, how do we check if there is a path of length at least 2 from one vertex u to another v? One way is to compute all the matrices $A^2, A^3, \ldots, A^{n-1}$ then check whether for any $2 \le k \le n-1$ we have $(A^k)_{u,v} > 0$. Note we only have to compute up to $k = n-1$ since if there is a path from one vertex to another, the path length must be at most $n-1$.

The above approach is correct, but it is too slow. What is its running time? Well, once we have A^{k-1} we can get A^k via one more matrix multiplication, namely $A^k = A \cdot A^{k-1}$. Thus we need to do $n-1$ matrix multiplications to get all the A^k matrices of interest. Each matrix multiplication takes n^3 arithmetic operations, so that seems like $O(n^4)$ overall. Then checking, for each edge, whether it has a non-zero entry in any A^k takes $O(n)$ time per edge, and thus $O(nm) = O(n^3)$ time overall. Thus the overall time seems to be $O(n^4)$, right? No! (And n^4 is too slow anyway.)

The problem is that the numbers that are being added and multiplied during each matrix multiplication can get quite big. For example, consider this graph:



How many paths are there from vertex 1 to n? At the beginning of each diamond, one can choose to cross the diamond either by going up or down, and thus the number of paths is 2^d where $d \approx n/3$ is the number of diamonds. (**Exercise:** Show that 2^n is also an upper bound on how many paths there can be between any two vertices.) In summary, the entries in A^k can get exponentially large, so they can take $\Theta(n)$ digits to write down. Thus multiplying them naively (without Karatsuba) would take $\Theta(n^2)$ time. Thus the running time of the above method is not $O(n^4)$. It is, however, $O(n^6)$.

Now, let's speed things up. First of all, let's prevent the numbers from getting too big! Rather than do matrix multiplication over the integers, let's do it over booleans. That is, usually we have

$$(A \times B)_{i,j} = \sum_k A_{i,k} \times B_{k,j}$$

but let's instead do Boolean matrix multiplication:

$$(A \times B)_{i,j} = \bigwedge_k A_{i,k} \vee B_{k,j}$$

where $\wedge$ is "logical and" and $\vee$ is "logical or". In this way, we only ever keep booleans around and don't deal with large numbers. Note now $(A^k)_{u,v}$ will not be the number of paths of length k, but rather will simply be "True" if there is at least one path of length k and "False" otherwise. This cuts our runtime down from $O(n^6)$ to $O(n^4)$.

The last modification to speed things up is the following. Consider modifying the graph by adding a self-loop from every vertex to itself (that is, adding the identity matrix to A). Then in this modified graph, when forming a path of length k, some of the k steps can choose to simply stay at the current vertex. Therefore $(I + A)^k_{u,v}$ will be "True" (using Boolean matrix multiplication) if there is a path of length *at most* k from u to v (as opposed to exactly k).

Therefore, the (u, v) entry of $A^2(I + A)^{n-1}$ tells us whether there is a path from u to v where we first take two steps, then afterward we take at most $n - 1$ more steps. $(I + A)^{n-1}$ can be computed using $O(\log n)$ matrix multiplications using repeated squaring (Lecture 2). Thus overall our running time is $O(n^3 \log n)$ to compute $A^2(I + A)^{n-1}$, then an additional $O(m) = O(n^2)$ to check, for each edge, whether or not the corresponding entry in this matrix is True or False.

Remark 2. There are matrix multiplication algorithms taking less than n^3 time; for example, Strassen's algorithm takes $O(n^{\log_2 7})$ time, giving an $O(n^{\log_2 7} \log n)$ time algorithm for this problem. Even better algorithms can perform matrix multiplication in $O(n^{2.3728639})$

time (François Le Gall. *Powers of Tensors and Fast Matrix Multiplication*, arXiv:10401.7714, 2014. http://arxiv.org/abs/1401.7714).

See the code below for a sample implementation:

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>

using std::cin;
using std::cout;
using std::istringstream;
using std::string;

// Matrix entry type. We really only need a bit per matrix entry; char is the
// smallest native type.
typedef char matrix_entry_t;

class matrix {
    private:
        int size_;
        // We store the matrix as one linear array instead of as a
        // two-dimensional array.  This guarantees that the matrix is stored in
        // one contiguous chunk of memory, giving better cache properties and
        // resulting in better performance.  The matrix is stored as:
        // row1 ... rowN, so the ij-th entry is stored at location i*size_+j.
        matrix_entry_t* matrix_;
    public:
        matrix(int size): size_(size) {
            matrix_ = (matrix_entry_t *) calloc(size_ * size_,
                    sizeof(matrix_entry_t));
            if (matrix_ == NULL) {
                std::cerr << "Out of memory.\n";
                exit(1);
            }
        }

        matrix(const matrix& A) {
            size_ = A.size_;
            matrix_ = (matrix_entry_t *) calloc(size_ * size_,
                    sizeof(matrix_entry_t));
            for (int i = 0; i < size_ * size_; ++i) {
                matrix_[i] = A.matrix_[i];
            }
        }

        matrix operator=(const matrix& A) {
            for (int i = 0; i < size_ * size_; ++i) {
                matrix_[i] = A.matrix_[i];
            }
            return *this;
    }

    ~matrix() {
        free(matrix_);
    }

    inline matrix_entry_t get(int i, int j) const {
        return matrix_[i * size_ + j];
    }

    inline matrix_entry_t set(int i, int j, matrix_entry_t value){
        return (matrix_[i * size_ + j] = value);
    }

    matrix operator+(const matrix &other) {
        matrix result(size_);
        for (int i = 0; i < size_ * size_; ++i) {
            result.matrix_[i] = matrix_[i] | other.matrix_[i];
        }
        return result;
    }

private:
    void multiply(const matrix &other, matrix* result) {
        for (int i = 0; i < size_; ++i) {
            for (int k = 0; k < size_; ++k) {
                for (int j = 0; j < size_; ++j) {
                    result->set(i, j,
                            result->get(i, j) |
                            (get(i, k) & other.get(k, j)));
                }
            }
        }
    }

public:
    matrix operator*(const matrix &other) {
        matrix result(size_);
        multiply(other, &result);
        return result;
    }

    /* Matrix exponentiation by repeated squaring. Running time is O(T log
     * N), where T is the time for matrix multiplication. Naive
     * exponentiation is O(T N).
     */
    matrix power(int n) {
        matrix result(size_), arg = *this;
        matrix temp(size_);
        for (int i = 0; i < size_; ++i) {
            result.set(i, i, 1);
            }

            while (n > 0)
            {
                if (n & 1) {
                    result.multiply(arg, &temp);
                    result = temp;
                }
                if (n > 1) {
                    arg.multiply(arg, &temp);
                    arg = temp;
                }
                n >>= 1;
            }
            return result;
        }
};


int main() {
    // Number of vertices
    int n;
    cin >> n;
    cin.ignore(1, '\n');

    // Adjacency matrix
    matrix adjacency(n);

    // Read in adjacency matrix
    for (int i = 0; i < n; ++i) {
        int vertex;
        string line;
        getline(cin, line);
        istringstream iss(line);
        iss.peek();
        while (!iss.eof()) {
            iss >> vertex;
            adjacency.set(i, vertex, 1);
        }
    }

    // Adjacency matrix plus identity
    matrix full_adjacency(n);
    for (int i = 0; i < n; ++i) {
        full_adjacency.set(i, i, 1);
    }
    full_adjacency = full_adjacency + adjacency;
    matrix final_result = adjacency * adjacency * full_adjacency.power(n - 1);

    // Remove all edges with nonzero entries in final_result,
    // and print the answer
    for (int i = 0; i < n; ++i) {
        int first = 1;
        for (int j = 0; j < n; ++j) {
            adjacency.set(i, j, adjacency.get(i, j) & ~final_result.get(i, j));
            if (adjacency.get(i, j)) {
                if (!first) {
                    cout << " ";
                }
                cout << j;
                first = 0;
            }
        }
        cout << "\n";
    }
    return 0;
}
```

Approach 2: We could, first, for each vertex v figure out what is reachable from v via a DFS (other than v itself). There are n DFS'S, each taking time $O(n+m)$, thus this would take $O(n^2+nm)$ time.

Then, we loop over every vertex u. For each edge (u,v) incident upon u, we "mark" all vertices w such that both (u,w) is an edge and w is reachable from v. Then after processing all edges incident upon u, we know that the marked w correspond exactly to the edges (u,w) such that u has a path of length at least 2 to w. Let d_u be the out-degree of u. Then the running time to process u is $O(d_u \cdot n)$. Thus to process all u takes time $O(n\sum_u d_u) = O(nm)$.

The overall running time is thus $O(n^2+nm)$ (which is always $O(n^3)$ since $m \le n^2$).

See the code below for a sample implementation:

```
#include <iostream>
#include <vector>
#include <sstream>
#include <string.h>
using namespace std;

// adjacency list representation
// adj[i] is a vector<int> of all vertices j such that (i,j) is an edge
vector< vector<int> > adj;

// adjacency matrix
char A[250][250];

// reachable[i][j] is 1 if i has a path to j (of length at least one),
// and is 0 otherwise
char reachable[250][250];

char visited[250];
void dfs(int v, int start) {
  if (v != start)
    reachable[start][v] = 1;
  visited[v] = 1;
  for (int i = 0; i < adj[v].size(); ++i)
    if (!visited[adj[v][i]])
      dfs(adj[v][i], start);
}

int main() {
  int n;
  cin >> n;
  cin.ignore(1, '\n');

  memset(A, 0, sizeof(A));
  memset(reachable, 0, sizeof(reachable));

  // Read input and fill in adjacency list and matrix
  for (int i = 0; i < n; ++i) {
    int vertex;
    string line;
    vector<int> V = vector<int>();
    getline(cin, line);
    istringstream iss(line);
    while (iss >> vertex) {
      V.push_back(vertex);
      A[i][vertex] = 1;
    }
    adj.push_back(V);
  }

  for (int i = 0; i < n; ++i) {
    // figure out what is reachable from i and put it in reachable[i]
    memset(visited, 0, sizeof(visited));
    dfs(i, i);
  }

  // the new adjacency list representation after removing edges
  vector< vector<int> > new_adj_list;
  for (int i = 0; i < n; ++i)
    new_adj_list.push_back(vector<int>());

  for (int i = 0; i < n; ++i) {
    // mark all vertices k such that i has a path of length at least 2 to k
    char mark[n];
    memset(mark, 0, sizeof(mark));
    for (int j = 0; j < adj[i].size(); ++j)
      for (int k = 0; k < n; ++k)
        if (reachable[adj[i][j]][k])
          mark[k] = 1;
    // copy over the adjacency list to new_adj_list, removing marked neighbors
    for (int j = 0; j < adj[i].size(); ++j)
      if (!mark[adj[i][j]])
        new_adj_list[i].push_back(adj[i][j]);
  }

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < new_adj_list[i].size(); ++j) {
      if (j)
        cout << "␣";
      cout << new_adj_list[i][j];
    }
    cout << endl;
  }

  return 0;
}
```