1 Format
You will have 75 minutes to complete the exam. The exam will have true/false questions, multiple choice, example/counterexample problems, run-this-algorithm problems, and problem set style present-and-prove problems.

Whenever you are asked to give an algorithm for a problem, you are expected to analyze its correctness and running time. More efficient correct solutions receive more credit. Less efficient but correct solutions would receive partial credit. We understand and expect that proofs of correctness will be more concise on exams than they would be on homework.

The exam is long, so be sure to work efficiently. Do not be discouraged, however, if you do not finish.

2 Topics Covered
This is a pretty comprehensive list of topics we have gone over this first half of the semester, but anything said in lecture through 2/13 inclusive is fair game for the exam.

2.1 Math/Fundamentals
- **Induction.** If \( P(n) \) is a statement ("\( 2n \) is even"), \( P(1) \) is true, and \( \forall n, P(n) \rightarrow P(n + 1) \), then \( \forall n, P(n) \) is true.
- **Big-O Notation.** \( o, O, \omega, \Omega, \Theta \) and identifying which of these relations hold for two functions.
- **Recurrence Relations.** Solve simple ones by finding a pattern and proving them with induction. More complicated recurrences can be solved using the Master Theorem (must memorize).
- **Integer Multiplication.** Perform 3 multiplications on \( n/2 \) digit numbers, and then do some additions.
- **Fast Powering.** Use repeated squaring to find an \( n \)th power in \( O(\log n) \) time.
- **Merge Sort.** Sort a list of \( n \) numbers in \( O(n \log n) \) time. Implement recursively or iteratively.

2.2 Graph Search
- **Representation.** Adjacency list versus adjacency matrix
- **Depth First Search (DFS).** Uses a stack to process nodes, Pre and post order numbers, labels for the edges (tree, forward, back, cross).
  **Important Applications.** Detecting cycles, topological sorting, finding strongly connected components (relate to 2SAT).
- **Breadth First Search (BFS).** Uses a queue to process nodes, can be used to find the shortest path when all edge weights are 1.
• **Dijkstra’s Algorithm.** Single source shortest path for non-negative edge weights. Uses a heap or priority queue to process nodes. Does not work when there are negative edge weights (why?).

• **Heaps.** Binary heap implementation, operations: **DELE**EMIN, **IN**SERT, **DECRE**ASE**KEY**, how they are used in Dijkstra’s algorithm.

• **Bellman-Ford Algorithm.** Single source shortest path for general edge weights, edge relaxation procedure (referred to as **update** in the lecture notes), detecting negative cycles.

• **Floyd-Warshall Algorithm.** All pairs shortest paths via dynamic programming, detecting negative cycles.

• **Shortest Path in DAG.** Can be done in linear time via dynamic programming regardless of edge weights.

### 3 Practice Problems

**Problem 1.**

Answer True or False for the following:

(a) T F $4\sqrt{n} = o(2^n)$

(b) T F $\log_5(n) = \Omega(\log_3(n))$

(c) T F $2^{\sqrt{\log n}} = \omega(n)$

(d) T F $\log(n)\log(n) = o(n))$

(e) T F Suppose $T(n) = 2T(n/b) + n$. Then, as $b \rightarrow \infty$, we eventually get $T(n) = \Theta(1)$.

(f) T F If $T(n) = T(\sqrt{n}) + 1$, then $T(n) = o(\log \log n)$.

#### Solution

(a) [T] Let’s take the limit:

$$\lim_{n \rightarrow \infty} \frac{4\sqrt{n}}{2^n} = \lim_{n \rightarrow \infty} \frac{2^{2\sqrt{n}}}{2^n} = \lim_{n \rightarrow \infty} 2^{2\sqrt{n} - n} = 0$$

The limit is 0 and therefore the little $o$ relationship holds.

(b) [T] Using our change of base formulas, we have:

$$\log_5 n = \frac{\log_3 n}{\log_3 5} = \log_5 3 \cdot \log_3 n \geq 0.1 \log_3 n$$

and therefore the $\Omega$ relationship is true.
(c) Take the limit:

$$\lim_{n \to \infty} \frac{2^{\log n}}{n} = \lim_{n \to \infty} \frac{2^{\log n}}{2^{\log n}} = 0$$

Therefore, the $\omega$ relationship is false and the relationship should be $o$.

(d) Substitute in $n = 2^k$. Then we have $\log(2^k)^{\log(2^k)} = k^k$ and $2^k = \omega(2^k)$, so the stated relationship does not hold.

(e) As $b \to \infty$, it is true that $T(n)$ becomes smaller because you are breaking the problem into more and more sub-parts. However, no matter how big $b$ is, you always need to pay the fixed cost of $n$ and so asymptotically, $T(n)$ will never be constant. You can also see this through the Master Theorem.

(f) Perform a similar substitution as the one we did on the problem set. Let $S(n) = T(2^n)$. We get the recurrence: $S(n) = S(n/4)+1$ which solves to $S(n) = \log(n)$ and thus $T(n) = \log \log(n)$. Therefore, the correct relationship should be $\Theta$ and not $\omega$.

Problem 2.

Given a directed acyclic graph $G = (V, E)$ and two nodes $u, v \in V$, calculate the number of distinct paths from $u$ to $v$. (Two paths are considered distinct if they have at least 1 vertex not in common)

Solution

Since $G$ is a DAG, we can perform a topological sort of its vertices in $O(n+m)$ time using DFS. Suppose our topological sort comes out to be $u, w_1, w_2, \ldots, w_k, v$. Note that if $v$ appears before $u$, then we should return 0 immediately since there would be no way to get from $u$ to $v$.

Now, we define a recursive function $f(i)$ to be the number of distinct paths from $u$ to $w_i$ and let $u = w_0$. We are interested in the value $f(v)$.

Base Case: if $i = 0$, then $f(i) = 1$. There is only 1 way to get from $u$ to $u$, which is to stay at $u$.

Recursive Case: if $i \neq 0$, then

$$f(i) = \sum_{w_j:(w_j,w_i) \in E} f(j)$$

The number of ways to get to $w_i$ is the sum of the number of ways to get to each of $w_i$’s predecessors that have an edge connecting to $w_i$. Note that $0 \leq j < i$ and since the graph is topologically sorted, we guarantee that if there’s an edge $w_j \to w_i$, then $j < i$.

Run-Time: First, the topological sort takes $O(n+m)$ time by DFS. Next, there can be up to $n$ values of $f(i)$ to compute, and for each vertex $w_j$, the it takes $O(1) + O(\text{indegree}(v))$ time to compute $f(w_j)$ so the total run-time is $O(n+m)$. The sum of the indegrees of each vertex in a directed graph is the number of edges in the graph. Note that we can pre-compute $G^R$ in order to be able to access indegrees rather than just outdegrees in $O(1)$ time.

Problem 3.

A directed graph $G = (V, E)$ is called semiconnected if for each pair of distinct vertices $u, v \in V$,
there is either a path from $u$ to $v$ or a path from $v$ to $u$. Find an algorithm to determine whether a directed graph is semiconnected. *Hint: Look at the SCC graph.*

**Solution**

Let $G'$ be the DAG on the strongly connected components of $G$. Number the nodes of $G'$ in some topological order. We claim that $G$ is semiconnected iff there is always a path from the $i$-th node to the $j$-th node of $G'$ if $i < j$.

Suppose that there is always a path from the $i$-th node to the $j$-th node of $G'$ if $i < j$. Then, for any two vertices $u$ and $v$ in $G$, they either belong to the same SCC (in which case there is a path from $u$ to $v$ and a path from $v$ to $u$), or they belong to two different SCC’s, $c_u$ and $c_v$. Without loss of generality, suppose $c_u$ occurs earlier than $c_v$ with respect to the topological order. Then, by hypothesis, there is a path from $c_u$ to $c_v$, so there is a path from $u$ to $v$ as desired.

Now suppose that for some topological sort of $G'$, there is no path from the $i$-th node (with respect to this sort) to the $j$-th node, where $i < j$. Then, there can be no path from the $j$-th node to the $i$-th node since $i < j$ and the nodes are topologically sorted. Hence, if $u$ is a vertex lying in the $i$-th SCC and $v$ a vertex lying in the $j$-th SCC, there are no paths from $u$ to $v$ or from $v$ to $u$, so $G$ is not semiconnected.

This observation allows us to devise an algorithm to determine if $G$ is semiconnected: we first compute the strongly connected components of $G$ and construct the graph $G'$ in time $O(n + m)$, where $n = |V|$ and $m = |E|$. Then, using DFS from any vertex in $G'$, we can topologically sort $G'$ in $O(n + m)$ time. Now, we check if there is a path from the $i$-th SCC to the $j$-th SCC if $i < j$. Note that this is the case iff there is an edge in $G'$ from the $i$-th SCC to the $i + 1$-th SCC. Hence, we can just scan through the topological sort in $O(n)$ time to see if the $i$-th SCC has an edge to the $i + 1$-th SCC. The total running time of the algorithm is $O(n + m)$.

**Problem 4.**

Suppose you are given an adjacency-list representation of an $n$-vertex graph undirected $G$ with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the $K$ closest vertices to some vertex $v$ in $O(K \log K)$ time.

*Warning:* Your solution’s run-time must not involve $N$ in any way!

**Solution**

We use a modified version of Dijkstra’s algorithm. Notice that in Dijkstra’s, every time we pop a node off the heap, we are saying that we have finalized its shortest distance from the source. Therefore, if we run Dijkstra’s and stop once we pop off $K$ nodes from the heap, we will have found the $K$ closest nodes to the heap.

We start Dijkstra’s algorithm with an empty heap and insert $v$ first with priority 0. We run the algorithm until we have popped off $K$ nodes and then stop. The $K$ nodes popped off are precisely the $K$ closest nodes to $v$. The correctness of this algorithm falls from the correctness of Dijkstra’s. So why is the run-time necessarily $O(K \log K)$?

The run-time is bounded by the number of deleteMin and the number of insert/decreaseKey operations to the heap. When we pop off a node, we add at most 5 other nodes because the degree of the graph is bounded by 5. Therefore, over the course of $K$ deleteMin’s, we will insert/decreaseKey
at most 5\(K\) times. Therefore, the run-time is:

\[ O(5K \cdot \text{insert/decreaseKey} + K \cdot \text{deleteMin}) \]

which becomes \(O(K \log K)\) if we use a binary heap.

**Problem 5.**

We are given a directed graph \(G = (V, E)\) on which each edge \((u, v) \in E\) has an associated value \(r(u, v)\), which is a real number in the range \(0 \leq r(u, v) \leq 1\) that represents the reliability of a communication channel from vertex \(u\) to vertex \(v\). We interpret \(r(u, v)\) as the probability that the channel from \(u\) to \(v\) will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

**Solution**

The reliability of any path is the product of the probabilities of not failure over each segment of the path. Therefore, this problem is very similar to the shortest path problem except we are trying to maximize the product instead of minimizing the sum. There are two ways we can approach this problem:

1. **Modify the probabilities**

   For each edge \((u, v) \in E\), we replace \(r(u, v)\) with \(-\log r(u, v)\). If \(r(u, v) = 0\), then let \(\log r(u, v) = \infty\). The resulting numbers will be between 0 and \(\infty\). This works because by taking the log of a product, we convert multiplication to addition. Then, taking the negative of that quantity makes minimizing the sum of the logs equivalent to maximizing the product of the \(r\)'s. We have that

   \[
   \max \left( \prod r(u, v) \right) = \min \left( -\log \prod r(u, v) \right) = \min \left( -\sum \log r(u, v) \right)
   \]

   where each sum or product is taken over \((u, v) \in \) path. Therefore, by minimizing the right hand side, we maximize the left side. We can perform a standard Dijkstra’s Algorithm after modifying the weights in this manner because the \(-\log r\) (where \(0 \leq r \leq 1\)) is always non-negative.

2. **Modify the algorithm**

   - We can replace the min-heap in Dijkstra’s Algorithm with a max-heap because now we are trying to maximize some quantity. Whenever we pop a node off the max-heap, we know we have the longest path to that node because everything else in the heap has a smaller reliability, so it would not be possible to get a better reliability by going via a node with worse reliability.

   - Instead of checking \(\text{dist}[w] > \text{dist}[v] + w(v, w)\) for some edge \((v, w)\), we replace the + with a \(\times\) and the < with a >. The resulting expression is: \(\text{dist}[w] < \text{dist}[v] \cdot w(v, w)\). We update the distance of some node \(w\) a larger value can be achieved by \(\text{dist}[v] \cdot w(v, w)\). In other words, going through \(v\) to get to \(w\) gets us a larger total distance (i.e. greater reliability).

The correctness of either solutions falls immediately from that of Dijkstra’s. Their run-time is therefore also the same as that of Dijkstra’s.