Midterm Exam

Name: _____

Perfect score: 100 points - Available points: 112.
No notes or books are allowed.

A. Specify whether the **entire** following statements are true or false and **argue why**:

- A dynamic programming algorithm is operating over input $x_1, \ldots, x_n$ and considers subproblems of the form $x_i, x_{i+1}, \ldots, x_j$, for the possible values of i and j where $1 \leq i \leq j \leq n$. Then the number of possible subproblems is linear, i.e., $O(n)$.

False. The number of possible subproblems is $O(n^2)$.

- Shortest paths do not make sense in graphs with negative weight cycles. Shortest paths, however, can contain zero weight and positive weight cycles.

False. Shortest paths do not contain positive weight cycles.

- The best running time of Dijkstra's algorithm is achieved for a Fibonacci heap implementation of the priority queue and corresponds to $O(|E|log|V| + |V|)$.

False. The running time of Dijkstra's algorithm for a Fibonacci heap implementation is $O(|V| \log |V| + |E|)$

- The Floyd-Warshall algorithm is a preferred solution for computing all-pair shortest paths on a dense graph versus calling the Bellman-Ford algorithm for every vertex.

True. The running time of Floyd-Warshall is $\Theta(|V|^3)$. Calling Bellman-Ford for every vertex of the graph would result in a $O(|V|^2 \cdot |E|^2)$ approach, which corresponds to $O(|V|^4)$ running time for dense graphs.

- Given value $h : V \rightarrow \mathbb{R}$ for every vertex of graph $G(V, E)$, update weights w according to: $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. The graph $\hat{G}$ with weights $\hat{w}$ is guaranteed not to have negative cycles.

False. There is no property for the value h provided that can guarantee that negative cycles are removed. For instance, if $h(u) = 0$ and the original graph has negative cycles, then the new graph $\hat{G}$ will also have negative cycles.

(20 points: 1 point for correct answer and 3 for correct reasoning per question)

B. Consider a set of n elements and that the optimal cover consists of k sets. What is the maximum number of sets that a greedy algorithm for set cover will return? (4 points)

The greedy algorithm will use at most $k \ln n$ sets.

Show that for any integer n that is a power of 2 it is possible to construct a set cover problem that has the following properties:

- The number of elements in the set are n.
- The optimal cover uses just two sets (i.e., $k = 2$).
- The greedy algorithm picks at least $logn$ sets.

[Hint: It might be helpful to think first how you can construct such a set cover problem for a specific value of n that is a power of 2, e.g., 32, and then generalize.] (12 points)

Since there is an optimal cover with just two sets, there must be two subsets s_A^* and s_B^* that together they cover all of the n elements. Lets consider the case that $|s_A^*| = |s_B^*| = \frac{n}{2}$ and there is no overlap between these two subsets.

A greedy algorithm will select at each iteration the subsets that have the maximum number of uncovered elements. Consider the case that there is a subset s_1 with the maximum number of elements so that $|s_1| = \frac{n}{2} + 1$ and $\frac{n}{4}$ elements of $|s_1|$ come from those of the subset $|s_A^*|$ and $\frac{n}{4} + 1$ elements come from those of the subset $|s_B^*|$. Then the greedy algorithm will select s_1 as its first choice. Then, each of the optimal subsets will have at most $\frac{n}{4}$ elements uncovered. Then consider that there is another subset s_2 with at least $\frac{n}{4} + 1$ elements, where $\frac{n}{8}$ of them come from the elements of s_B^* that were not covered by s_1, and the remaining $\frac{n}{8} + 1$ elements come from the subset s_A^* but were not covered by s_1. The greedy algorithm will then select s_2 as the second choice after s_1 because it still has more uncovered elements than the optimal subsets.

We can keep defining such subsets that the greedy algorithm will be selecting and which will always have more uncovered elements than the optimal subsets. These subsets will keep getting half in size at every iteration of the algorithm, so there will be $\log n$ of them.

You are given two strings of size m and n and you want to compute their longest common subsequence. How does the dynamic programming solution compute edit distances $E(i, j)$ for i-bit and j-bit prefixes of the two input strings given subproblems of smaller sizes? (4 points)

$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), diff(i, j) + E(i-1, j-1)\}$$

What is the running time of the dynamic programming solution for the longest common subsequence problem given the above two strings and why? (4 points)

The above procedure fills in a two-dimensional matrix row by row, and left to right within each row. The size of the matrix is $m \times n$, i.e., corresponding to the sizes of the two strings. Each entry takes constant time to fill in (it corresponds to the above computation), so the overall running time is just the size of the table, $O(mn)$.

C. You are given a set of programming jobs to run on a computer. Each job has its own memory requirements and a priority:

Type of Job	GB	Priority
1	6	30
2	3	14
3	4	16
4	2	9

Choose the set of jobs that can run in parallel without paging that maximizes the sum of priorities. The computer has 10 GB RAM. You can select any number of instances for each type of job. Use a dynamic programming approach and show clearly the intermediate steps of your computation. (8 points)

Knapsack with repetition.

	1	2	3	4	5	6	7	8	9	10
p(i)	-	9	14	18	23	30	32	39	44	48
job	-	4	2	4	4|2	1	1|3|2	1|4	1|2	1
goes to	-	-	-	2	3|2	-	1|3|4	2|6	3|6	4

How is it possible to identify (a) a sink and (b) a source strongly connected component in a DAG? (4 points)

The node that receives the highest post number in a DFS must lie in a source strongly connected component.

The node that receives the highest post number in a DFS on a graph G^R, with the edges reversed, must lie in a source strongly connected component.

There are multiple ways to correctly answer this question.

Provide an algorithm that computes the decomposition of a directed graph into its strong connected components. What is its running time? (4 points)

Compute the graph with the reversed edges G^R. Run DFS on G^R.
Run an undirected connected component algorithm on G, and during the DFS, process the vertices in decreasing order of their post number from step 1.

The running time is linear $O(|V| + |E|)$, because of the running time of a DFS.

What does the "optimal substructure" property of shortest paths specify? (4 points)

Given a weighted, directed graph $G(V, E)$ with weight function $w : E \rightarrow R$, let $p =< v_1, v_2, \ldots v_k >$ be a shortest path from vertex v_1 to vertex v_k and for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} =< v_i, v_{i+1}, \ldots v_j >$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is the shortest path from v_i to v_j.

D. 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 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. (8 points)

We can use any single source shortest path algorithm you have learned in the class. One of these algorithms is Bellman-Ford. But to apply such a procedure to this problem we need to first perform a domain conversion so that for each two nodes u and v, let $\bar{r}(u, v) = -\log r(u, v)$. By using $\bar{r}$ as the weignts for graph $\bar{G}$, we guarantee that all weights are positive. And by solving the shortest path problem between the given source node S and the destination node T for these transformed weights, we are going to find the most reliable path between S and T. The running time of this algorithm is actually the running time of Bellman-Ford algorithm, which is $O(|V||E|)$, in addition to the time needed for converting all weights which is $O(|E|)$. So the total running time is $O(|V||E|)$.

Show that Dijkstra's algorithm maintains the following invariant during each iteration: "The shortest path has already been computed for all the nodes that have been removed from the queue". (12 points)

We wish to show that in each iteration, $u.d = \delta(s, u)$ for every vertex added to set S.

We can use contradiction. First let $u.d \neq \delta(s, u)$ when it is added to set S.
Without loss of generality, let u be the first vertex added to S based on the distance value $u.d = \delta(s, u)$ at that time by examining a shortest path from s to u.
Assume for contradiction that $u \neq s$ and there is a shortest path p from s to u. Prior to adding u to S, path p connects a vertex in S, namely s, to a vertex in $V \in S$, namely u. And let us assume that there is another node y along p such that $y \in V - S$, and y is a predecessor of u along p.
Because y appears before u on a shortest path from s to u and all edge weights are non-negative, we have $\delta(s, y) \leq \delta(s, u)$, and thus $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$.
But because both vertices u and y were in $V - S$ when u was chosen, we have $u.d \leq y.d$. From these two findings to be satisfied, we can say that $y.d = \delta(s, y) = \delta(s, u) = u.d$
Consequently, $u.d = \delta(s, u)$.

What is the running time of Dijkstra's algorithm if the priority queue is implemented as a simple array? What is the running time of Dijkstra's algorithm if the priority queue is implemented as a binary heap? When is each of these implementations preferred over the other? (4 points)

The running time of Dijkstra's Algorithm if the underline data structure is an array will be $O(|V|^2)$.
While if we use binary heap for implementing the priority queue, Dijkstra's running time will be $O((|V| + |E|) \log |V|)$.
The array is simple for implementation purposes and the binary heap is more convenient to be used if we want to extract the smallest/largest elements in dynamic list. But in the case of implementing Dijkstra's algorithm we only care about efficiency. So if $|E| < |V|2/log|V|$ or $|E| = O(|V|2/log|V|)$ (i.e., sparse graph), then using binary heap for implementing the priority queue is preferred. Otherwise, if it is known that the graph is dense, i.e. $E = O(|V|^2)$, then using an array is more efficient.

E. What is an efficient approach for computing single-source shortest paths on directed acyclic graphs that may contain negative weight edges and what is the running time of this solution? (4 points)

The first step is to linearize the graph using topological sort in $O(m+n)$ time where m is the number of edges and n is the number of nodes. Given the topological sequence $\{v_1, \ldots, v_n\}$, it is easy to update the distance table. From v_1 to v_n, update the distance table d_j if $d_i + w_{i,j} < d_j$. This could be done in $O(m+n)$ time as each edge has been visited at most once. Hence the total running time is linear, i.e., $O(m+n)$.

Provide the Floyd-Warshall algorithm for solving all-pair shortest path problems. What is the running time of the approach? (8 points)

Let $f_{i,j,k}$ denote the distance of shortest path from node i to node j containing nodes belonging to the set $\{1, \ldots, k\}$. Then the recurrence equation is $f_{i,j,k} = \min\{f_{i,k,k-1} + f_{k,j,k-1}, f_{i,j,k-1}\}$. The initial condition is $f_{i,j,1} = w_{i,j}$ if $(i,j) \in E$ and $f_{i,j,1} = \infty$ otherwise. After $n-1$ iterations, the two dimensional table $\{f_{i,j,n}\}_{1 \le i,j \le n}$ corresponds to the pairwise shortest distances. Obviously, the time complexity is $\Theta(n^3)$ because there are totally $n-1$ iterations and n^2 entries to be filled in each iteration.

Dr. Profanious claims that Johnson's algorithm is not necessary and there is a simpler way to reweight edges in order to be able to use Dijkstra's algorithm for all-pair shortest path problems. Letting $w^* = min_{(u,v) \in E}\{w(u,v)\}$, just define $\hat{w}(u,v) = w(u,v) - w^*$ for all edges $(u,v) \in E$. This is guaranteed to make all edge weights positive so that Dijkstra's algorithm can now be used. Do you see anything wrong with Dr. Profanious' method of reweighting? Justify your answer. [12 points]

Dr. Profanious' method is wrong. Consider a graph G of 3 nodes A, B, C and 3 edges (A → B), (B → C), (A → C). The weight of them are $-2, 3, 2$ respectively. Clearly, the shortest path from A to C is $A \to B \to C$ of weight 1. However, after the reweighting, the shortest path change to $A \to C$ since the original path now is of weight 5 but the new shortest path is of weight 4. Therefore, Dr. Profanious' method is wrong.

The intuitive reason why the method is wrong is the following: if a shortest path contains many edges, then the reweighting would add more weight along this path than a more expensive path that has fewer edges.