# Quiz 2 Solutions

**Problem 1. True or False** [35 points] (7 parts)

Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

**(a)** To sort $n$ integers in the range from $1$ to $n^2$, a good implementation of radix sort is asymptotically faster in the worst case than any comparison sort.

**True** **False**

*Explain:*

**Solution: True.** Split each number into two digits, each between $0$ and $n$. Then make two passes of counting sort, each taking $\theta(n)$ time. Any comparison sort will take $\Omega(n \log n)$ time.

**(b)** Call an ordered pair $(x_1, y_1)$ of numbers ***lexically less than*** an ordered pair $(x_2, y_2)$ if either (i) $x_1 < x_2$ or (ii) $x_1 = x_2$ and $y_1 < y_2$. Then a set of ordered pairs can be sorted lexically by two passes of a sorting algorithm that only compares individual numbers.

**True** **False**

*Explain:*

**Solution: True.** The idea is similar to RADIX-SORT though it is not exactly the same. First we sort on the $y_i$ digit of each pair, $(x_i, y_i)$. Next we sort on the $x_i$ digit using a stable sort.

**(c)** Any in-place sorting algorithm can be used as the auxiliary sort in radix sort.

**True False**

*Explain:*

**Solution: False.** The auxiliary algorithm has to be *stable*. In-place means the algorithm only uses constant additional memory.

**(d)** Every directed acyclic graph has only one topological ordering of its vertices.

**True False**

*Explain:*

**Solution: False.** Counterexample: the graph {(1,2), (1,3)} can be topologically sorted either [1,2,3] or [1,3,2].

**(e)** If the priority queue in Dijkstra's algorithm is implemented using a sorted linked list (where the list is kept sorted after each priority queue operation), then Dijkstra's algorithm would run in $O(E \lg V + V \lg V)$ time.

**True False**

*Explain:*

**Solution: False.** The array can take $\Theta(V)$ time to insert a node, and there are $V$ nodes to insert, for a running time of $\Omega(V^2)$.

In addition, the $\Theta(E)$ calls to decrease-key can each take $\Theta(V)$ time for a total running time of $\Theta((V + E)V)$.

**(f)** In searching for a shortest path from vertex $s$ to vertex $t$ in a graph, two-way breadth-first search never visits more nodes than a normal one-way breadth-first search.

**True** **False**

*Explain:*

**Solution:** **False.** Take a graph which looks like many spokes coming out of a center. The starting vertex is on the outside of one of the spokes, and the ending vertex is in the center.

Now a one-way BFS will only travel in one spoke, but a 2-way BFS will travel out all of the spokes.

**(g)** Every sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case to sort $n$ elements.

**True** **False**

*Explain:*

**Solution:** **False.** Counting sort, radix sort, and bucket sort all sort in fewer than $O(n \lg n)$ comparisons.

The lower bound only applies to comparison sorts.

**Problem 2. Linear Dijkstra?** [20 points] (2 parts)

**(a)** Professor Devamaine has just invented an exciting optimization for Dijkstra's algorithm that runs in $O(V + E)$ time for undirected graphs with edge weights of just $0$ and $1$.

Show the professor that the same time bound can be achieved simply by modifying the graph and then running breadth-first search as a black box.

**Solution:** We modify the graph as follows:

1. Run DFS on the graph to discover all connected components, only following 0-weight edges. This takes $O(V + E)$ time.
2. Collapse each connected component into a single supernode. This takes $O(V)$ time.
3. Put back 1-weight edges into the supernode graph. There is an edge between two supernodes if there is an edge with weight 1 between any pair of nodes that belong to each of the supernodes. This takes $O(E)$ time.
4. Run BFS on the new supernode graph, to get the shortest paths from the source supernode (the supernode that contains the source node as given to Dijkstra.) BFS can be run because all edges have an equal weight of 1.

**(b)** After hearing of his colleague's embarrassment, Professor Demaidas invents another modification to Dijkstra's algorithm that runs in $O(V + E)$ time for undirected graphs with edge weights of just 1 and 2.

Show the professor that the same time bound can again be achieved by modifying the graph and then running breadth-first search as a black box.

**Solution:** We modify the graph as follows:

1. For every 2-weight edge, insert a new node and split the edge into two 1-weight edges (with a new node inbetween). This takes $O(E)$ time.
2. Run BFS on the modified graph beginning with the source vertex as given to Dijkstra and determine shortest paths to each of the reachable vertices. BFS can be run since all the edges have equal weights. This takes $O(V + E)$ time. There are at most $E$ new vertices and $E$ new edges compared to the original graph.

**Problem 3. Bottleneck Path** [20 points] (2 parts)

**(a)** In the ***bottleneck-path problem***, you are given a graph $G$ with edge weights, two vertices $s$ and $t$, and a particular weight $W$; your goal is to find a path from $s$ to $t$ in which every edge has at least weight $W$. Describe an efficient algorithm to solve this problem. Your algorithm should work even if the edge weights are negative and/or the particular weight $W$ is negative.

**Solution:** Run BFS, ignoring any edges of weight less than $W$. This will take $O(V + E)$ time.

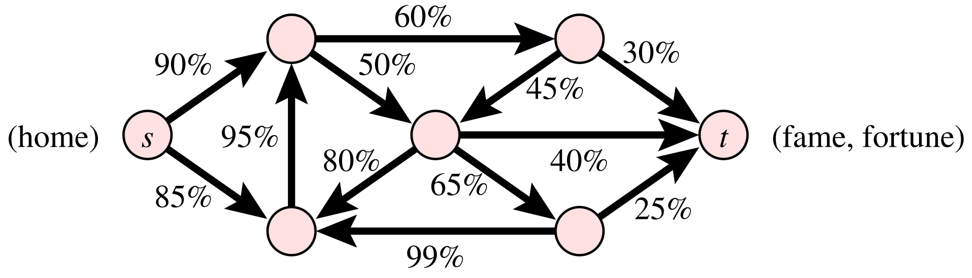
**(b)** In the ***maximum-bottleneck-path problem***, you are given a graph $G$ with edge weights, and two vertices $s$ and $t$; your goal is to find a path from $s$ to $t$ whose minimum edge weight is maximized. Describe an efficient algorithm to solve this problem that uses an efficient algorithm from Part (a) as a subroutine. You may assume an efficient algorithm for Part (a) exists, and use it as a black box.

**Solution:** This can be done in $O((V + E) \lg E)$ time using binary search on the edge weights. Since there are $|E|$ edges, there are at most $|E|$ unique weights. The desired weight must be some edge weight so we binary search for the largest weight of $w_{min}$ that maintains a path from $s$ to $t$. Sorting the edges take $O(E \lg E)$ time, and binary search takes $O(\lg E)$ time, with each iteration requiring $O(V + E)$ time to give $O((V + E) \lg E)$.

**Problem 4. Reality** [20 points]

You've just agreed to star in the new hit reality show, *Whose Geek Is It Anyway?* At the outset, you're given a map of an island of puzzles, which is a directed graph marked with a start vertex $s$ and a goal vertex $t$. Traversing each edge $e$ requires solving a puzzle, which you believe you can solve with probability $p(e)$. Describe how to modify the graph so that Dijkstra's algorithm will find a path from $s$ to $t$ that has the maximum probability of winning. (Assume your abilities to solve different puzzles are independent events.)

A sample input:



**Solution:** Let $e_1, e_2, \ldots, e_k$ be a list of edges forming a path. The probability of succeeding by folling this path is $p(e_1) * p(e_2) * \cdots * p(e_k)$. We want a path that maximizes this probability.

Dijkstra uses an incompatible definition for the weight of a path: $w(e_1) + w(e_2) + \cdots + w(e_k)$, and computes the minimum-weight path.

To compensate for the differences, we (1) use logs to turn products into sums, and (2) negate all weights so that the max path becomes the minimum cost path.

Specifically, we use $w(e) = -\log p(e)$.

Note that $0 \leq p(e) \leq 1$, so $-\infty \leq \log p(e) \leq 0$, so our formula gives us the non-negative edge-weights needed for Dijkstra's algorithm.

**Problem 5. Negative-Weight Cycles** [25 points]

If a directed graph $G = (V, E)$ contains a negative-weight cycle, shortest paths to some vertices will not exist, but there may still be shortest paths to other vertices. Assume that every vertex $v$ is reachable from the source vertex $s$ in $G$. Give an efficient algorithm that labels each vertex $v$ with the shortest-path distance from $s$ to $v$, or with $-\infty$ if no shortest path exists. (In other words, compute $\delta(s, v)$ for all $v$.) You can invoke all algorithms covered in lectures or recitations.

For reference, below is the pseudocode for Bellman Ford adapted from CLRS, which returns False if there are reachable negative weight cycles and True otherwise:

```
def bellman_ford(V, E, w, s):
    initialize_single_source(V, E, s)
    for i in range(|V|-1):
        for (u, v) in E:
            relax(u, v, w)
    for (u, v) in E:
        if d[v] > d[u] + w(u, v):
            return False
    return True
```

**Solution:** There are two natural $O(VE)$-time algorithms for this problem. In general, the idea is to run Bellman-Ford to compute $d[v] = \delta(s, v)$ when $\delta(s, v) > -\infty$. Then, we need to find all vertices $v$ reachable from a negative-weight cycle and set $d[v] = -\infty$.

1. After running Bellman-Ford, find all vertices with relaxable outgoing edges.

   Then run a DFS (or BFS) from each such vertex $v$, marking them with $d[v] = -\infty$. This runs in $O(VE)$ time ($O(E)$ per DFS). For practical efficiency, each successive DFS can backtrack when it hits a vertex visited by a previous DFS.

2. Just run Bellman-Ford again, but instead of relaxing relaxable edges, set $d$'s to $-\infty$.

SCRATCH PAPER

SCRATCH PAPER