# Quiz 2 Solutions

**Problem 1. True or False** [14 points] (14 parts)

For each of the following questions, circle either T (True) or F (False).

**(a)** **T F** Heapsort is not a stable sorting algorithm.

**Solution:** True

**(b)** **T F** Finding the minimum element of a binary min heap takes logarithmic time.

**Solution:** False

**(c)** **T F** Any type of sorting algorithm can be used to sort the digits in one phase of radix sort.

**Solution:** False

**(d)** **T F** Given a matrix representation of a graph with V vertices, we can run depth-first search in O(V) time.

**Solution:** False

**(e)** **T F** DFS finds the longest paths from start vertex s to each vertex v in the graph.

**Solution:** False

**(f)** **T F** In a graph with negative weight cycles, one such cycle can be found in $O(VE)$ time where $V$ is the number of vertices and $E$ is the number of edges in the graph.

**Solution:** True

**(g)** **T F** One always obtains the shortest path to each vertex, i.e., one using the minimum number of edges, using breadth-first search.

**Solution:** True

**(h)** **T** **F** A directed graph with a negative weight cycle has a topological ordering.

**Solution:** False

**(i)** **T** **F** If one can reach every vertex from a start vertex in a directed graph, then the graph is strongly connected.

**Solution:** False

**(j)** **T** **F** Topological sort can be performed using one breadth-first search procedure on the graph.

**Solution:** False

**(k)** **T** **F** Finding the strongly connected components in a graph requires $\Omega(VE)$ time where $V$ is the number of vertices and $E$ is the number of edges.

**Solution:** False

**(l)** **T** **F** If one transforms edge weights $w(u, v)$ to $w'(u, v) = w(u, v) - \lambda(u) + \lambda(v)$ for arbitrary $\lambda()$, then Dijkstra on the modified graph will return shortest paths in the original graph.

**Solution:** False

**(m)** **T** **F** The only operations required by Dijkstra on the priority queue data structure are EXTRACT_MIN, INSERT and DECREASE_KEY.

**Solution:** True

**(n)** **T** **F** Dijkstra assumes the triangle inequality on edge weights, that is, for each triple of edges $(u, v)$, $(u, a)$ and $(a, v)$, we have $w(u, v) >= w(u, a) + w(a, v)$.

**Solution:** False

**Problem 2. Merging Multiple Lists** [15 points]

You are given $K$ sorted lists of numbers. Each list has length $N/K$, so the total length of all of the lists is $N$. Describe an algorithm using heaps to merge these $K$ lists of $N/K$ numbers into a single sorted list of $N$ numbers. Your algorithm should run in $O(N \log K)$ time, although you will receive partial credit for less efficient algorithms.

**Solution:** Clearly $K \leq N$ as otherwise each list would contain $< 1$ numbers.

**Algorithm:** The pseudocode of the proposed algorithm is the following:

1.Create a min-heap $\mathcal{H}$ as follows

- the elements stored in $\mathcal{H}$ are the $K$ sorted lists given in the input;
- the key of each element of $\mathcal{H}$ is the smallest number in the corresponding list (note that finding this is an $O(1)$ operation since every list in the input is ordered);

2.create an empty output array $F[0:N]$; initialize an index $i = 0$;

3.while $\mathcal{H}$ is non-empty

(a)let $L$ be the list at the root of the heap;

(b)extract the minimum number $x$ from $L$; $F[i] \leftarrow x$; $i \leftarrow i + 1$;

(c)if $L$ is empty, call extract_min($\mathcal{H}$) to extract $L$ from $\mathcal{H}$;

(d)else call Heapify($\mathcal{H}$, 1) to fix the min heap property at the root of the heap (which may have been violated after extracting $x$ from $L$ since the key of $L$ was updated).

**Correctness:** The correctness of the algorithm follows from the fact that every time a number $x$ is added to the output list of numbers $F$, $x$ is the minimum among all uninserted numbers. Indeed, every list inside the heap is keyed on its minimum number; hence the minimum number in the list at the root of the heap is the minimum number among all lists' minimum numbers, and therefore the overall minimum number of all uninserted numbers.

**Running Time:** The running time of the algorithm is $O(K + N \log K)$. Indeed, it takes $O(K)$ to build the heap since $\mathcal{H}$ contains $K$ elements and reading the key out of each element is an $O(1)$ operation. Then for each number added to the output array $F$ we need to perform either an `extract_min` operation or a `heapify` operation on $\mathcal{H}$. These operations take $O(\log K)$ time, since at all times in the execution of the algorithm $\mathcal{H}$ has size $O(K)$. The total number of numbers added to the output array are $N$, hence we need to call these operations $O(N)$ times. Hence, the cost of the second phase of the algorithm is $O(N \log K)$, and the overall running time of the algorithm is $O(K + N \log K) = O(N \log K)$, using also that $K \leq N$.

**Problem 3. Assigning Directions** [15 points]

Consider a mixed unweighted graph $G = (V, E)$ with both directed and undirected edges. Assume that initially there are no cycles in $G$ which use only directed edges. Give an algorithm to assign direction to each of the undirected edges so that the completely directed graph so obtained has no cycles. Analyze the asymptotic complexity of the algorithm.

**Solution:** The set of edges consists of directed and undirected edges: $E = E_{\text{dir}} \cup E_{\text{undir}}$. We first find a topological ordering $(v_1, \ldots, v_{|V|})$ of vertices in the restricted graph $(V, E_{\text{dir}})$. We know that the ordering exists, because $G'$ is directed and acyclic. Every directed edge in $E_{\text{dir}}$ goes "from left to right" in the ordering, that is, it goes from some $v_i$ to some $v_j$ such that $i < j$. Then the algorithm assigns a direction to every edge $(v_p, v_q)$ in $E_{\text{undir}}$ so that it goes from $v_{\min\{p,q\}}$ to $v_{\max\{p,q\}}$, i.e., also from left to right in the ordering. The resulting graph is directed and clearly has no cycles.

The running time of the algorithm is $O(|V| + |E|)$. A topological ordering can be found in linear time, and directions can be assigned in linear time as well.

**Problem 4. Modified Dijkstra** [20 points] (2 parts) Modify Dijsktra's algorithm to find, out of all shortest paths, the one with fewest number of edges from a start vertex $s$ to all other vertices.

**(a)** [10 points] Propose an augmented data-structure for each node for solving this problem.

**Solution:** Augment each node `u` with value `l[u]`, which represents the current least number of edges in the current shortest path from the source to `u`. While relaxing an edge, if a path of a smaller length or the same length but with fewer edges is found, `l[u]` is updated to the number of edges in that path. During initialization, `l[source]` is set to 0, while other values can be set arbitrary.

**(b)** [10 points] Modify the RELAX function with your augmentation.

```
RELAX(u,v,w)

   if (_____________________________________) // relax condition

         then d[v] = d[u] + w(u,v)

                   ___________________________________


                   parent[v] = u
```

**Solution:**

```
RELAX(u,v,w)
   if (d[u]+w(u,v)<d[v] or (d[u]+w(u,v)==d[v] and l[u]+1<l[v]))
         then d[v] = d[u] + w(u,v)
              l[v] = l[u] + 1
              parent[v] = u
```

**Problem 5. Road Network** [15 points]

Consider a road network modelled as a weighted undirected graph $G$ with positive edge weights where edges represent roads connecting cities in $G$. However some roads are known to be very rough, and while traversing from city $s$ to $t$ we never want to take a route that takes more than a single rough road. Assume a boolean attribute $r[e]$ for each edge $e$ which indicates if $e$ is rough or not. Give an efficient algorithm to compute the shortest distance between two cities $s$ and $t$ that doesn't traverse more than a single rough road. (Hint: Transform G and use a standard shortest path algorithm as a black-box.)

**Solution:** Transform the graph $G$ to get the graph $G' = (V', E')$ in the following way:

For every vertex $v \in V(G)$, create two vertices $v_r$ and $v_s$ in $G'$. For every smooth edge $(u, v) \in E(G)$, create edges $(u_r, v_r)$ and $(u_s, v_s)$ in $G'$. For every rough edge $(u, v) \in E(G)$, create edges $(u_s, v_r)$ and $(v_s, u_r)$ in $G'$. Now we can run Dijkstra's shortest path algorithm to compute the shortest paths from $s_s$ to $t_s$ and from $s_s$ to $t_r$ and select the minimum of the two. Creating $G'$ from $G$ takes $O(E + V)$ time and has $2V$ vertices and $2E$ edges. Running Dijkstra takes $O(2E + 2V \log 2V)$, i.e. $O(E + V \log V)$ time using fibonacci heaps. Alternatively, we can also add $0$ weight edges from $u_s$ to $u_r$ for all $u \in V(G)$ and run Dijsktra once from $s_s$ to $t_r$ which still has the same asymptotic runtime.

The rough vertices $v_r \in V(G')$ model the scenario when one rough edge has been traversed during the path. From construction we can guarantee that any path in $G'$ can have atmost one rough edge. As soon as a rough edge is traversed, from the smooth vertices region we reach the rough vertices region and now there are no outgoing rough edges from this region, only smooth edges can be traversed from this point onwards.

**Problem 6. Dynamic Programming** [20 points] (2 parts)

Describe an algorithm, using dynamic programming, to find the number of binary strings of length N which do not contain any two consecutive 1's. For example for N=2, the algorithm should return 3 as the possible binary strings are 00, 01 and 10. You will receive greater credit for a more efficient algorithm.

**(a)** [10 points] State the set of subproblems that you will use to solve this problem and the corresponding recurrence relation to compute the solution.

**Solution:** Let $a[i]$ be the number of binary strings of length $i$ which do not contain any two consecutive 1's and which end in 0. Similarly, let $b[i]$ be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

$$a[i] = a[i-1] + b[i-1]$$
$$b[i] = a[i-1]$$

The base cases of our recurrence are $a[1] = b[1] = 1$. The total number of strings of length $i$ is just $a[i] + b[i]$.

**(b)** [10 points] Write iterative (non-recursive) pseudo-code to compute the solution. Analyze the running time of your algorithm.

**Solution:**

```
(a,b) = (1,1)
for i in range(1,N):
    (a,b) = (a+b,a)
print a+b
```

This algorithm involves a loop with $N-1$ iterations and a total of $N-1$ additions. If we assume that each addition takes constant time, then our algorithm is this $O(N)$. Note, however, that the values $a[i]$ and $b[i]$ grow exponentially. This means that the final answer will have $O(N)$ digits. Thus, each addition requires $O(N)$ time. With $O(N)$ additions, this makes our algorithm $O(N^2)$.