Problem 1.
Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance? Explain.

Problem 2.
The maximum-spanning tree problem is defined similarly to the minimum-spanning tree problem except that we want to maximize the sum of the weights of the edges of the tree. Give an algorithm that finds the maximum-spanning tree of an undirected graph $G = (V, E, W)$ where edges have weights represented in the weight matrix $W$. What is the running time of the algorithm?

Problem 3.
You are given a directed graph with positive weights on its edges. Show how you can solve the minimum cost directed cycle problem in that graph in time faster than $o(n^4)$ (this is a little-oh) in the worst-case. A directed cycle is a directed path that starts and ends in the same vertex and has at least one edge. A minimum cost directed cycle is one for which the sum of the weights of the edges of the cycle is minimal.

Problem 4.
How would you detect the existence of a negative-weight cycle in Floyd-Warshall?

Problem 5.
You are given an acyclic connected and undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cycle (note that the bound involves only $n$ but not $m$)?

Problem 6.
You are given an undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cycle (note that the bound involves only $n$ but not $m$)? Assume an adjacency list representation of the graph.

Problem 7.
We have an undirected weighed graph whose edges have weights that are only 10 or 20. Can you find the minimum spanning tree of this graph in $O(n + m)$ time? Explain.

Problem 8.
You are given a graph $G = (V, E)$. We want to determine $\forall u, v$ whether there exists a path from $u$ to $v$. Give three different algorithms that solve this problem and their worst case running time in terms of $n$ and $m$.

Problem 9.
Consider inserting keys 1, 3, 5, 6, 12, 23, 10, 17, 34 into a hash table of length $N = 11$ using open addressing with the primary hash function $h'(k) = k \mod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k, i) = (h'(k) + i) \mod N$, (ii) quadratic probing with hash function $h(k, i) = (h'(k) + i^2) \mod N$. 


Solution Outline

Problem 1.
There is no reason to analyze its worst-case performance. Randomized quick-sort for example has the same worst-case performance as ordinary quick-sort and randomization does not change this. If the choices of the random number generator lead to the worst-case, that’s it, we get worst-case performance. The introduction of randomness, however, makes this highly unlikely. Much more interesting is an average-case analysis that analyzes the performance that "most likely" will be observed.

Problem 2.
If the graph is $G = (V, E, W)$ consider graph $G' = (V, E, -W)$ i.e. we negate the weights of all the edges in $G'$. A maximum spanning tree $T$ in $G$ is a minimum spanning tree in $G'$. Note that the conversion of $W$ into $-W$ takes time $O(m)$. If $G$ is represented by an adjacency-matrix we find all the edges and negate the corresponding weights in $O(n^2)$ time. If $G$ is represented by an adjacency-list we go through all the edges in time $O(m)$ and negate the corresponding weights.

Whatever the cost is $O(m \lg m)$, the running time of Kruskal’s algorithm will contribute more.

The maximum-spanning tree problem is defined similarly to the minimum-spanning tree problem.

Problem 3.
Set $a_{ii} = \infty$. Run Floyd-Warshall. Then $a_{ii}^n$ will give the length of the smallest cycle from $i$ to $i$. Find the minimum among all $a_{ii}^n$ for all $i$.

Problem 4.
At every iteration inspect the diagonal elements. Initially, they are all zero. If a value along the diagonal changes it can only grow smaller, i.e. negative. A negative $a_{ii}^k < 0$ show the existence of a cycle with end-points $i$ whose all intermediate vertices are $k$ or less.

Problem 5.
Yes! $G$ is a tree and thus $m = n - 1$. Therefore depth-first-search solves this problem in $O(n + m)$ time which is $O(n)$ as well.

Problem 6.
Yes! Run depth first search by also using two counters for each connected component discovered during the top-level call. One counts the vertices discovered and the other the edges discovered. If edges discovered is equal or more than vertices discovered, we have a cycle by the simple observation that in a tree edges are one less than vertices, and if in a connected component searched through depth-first-search we have discovered a number of edges equal to or more than the number of vertices, a cycle should exist there.

Note that in this algorithm the running time is $\sum_i O(n_i + m_i)$, where $n_i$ is the number of vertices in the $i$-th component searched and $m_i$ the number of edges searched in that component. Naturally $\sum i = n$. Given that the search process stops as soon as $m_i \geq n_i$, we have $m_i \leq n_i$. Therefore $\sum i m_i = O(n)$. Combining the two we obtain that $\sum_i O(n_i + m_i) = O(n)$ as well.

Problem 7.
Yes! We only need to maintain a heap whose operations are $O(1)$ rather than the traditional $O(\log n)$. Note that in Prim’s algorithm the key in a heap is the edge weight. We can only have two values 10 or 20. Therefore we can implement a heap with an array of size two that stores two (doubly) linked lists. We attach to element 0 a linked list of all edges of weight 10, and to element 1 a linked list of all edges of weight 20. We only insert or delete from the head of the list to maintain $O(1)$ time operations.

Insert inserts the weight of edge $(u, v)$ into the Heap as the weight of $v$. We insert either to the linked list attached to element 0 or to the one attached to element 1. Either time is constant. At the same time we point $point[v]$ to this location (we can also choose to record in $(u, v)$ information that this edge is related to $v$).

ExtractMin takes $O(1)$ time. Check list 0 first. If list 0 is not empty, extract its element pointed by head. If list 0 is empty go to list 1 and extract an element through the head pointer. If both lists 0 and 1 are empty return "underflow".

Decrease takes constant time because we use doubly linked lists. We might move an element (edge) from one list to the other. Note that in accordance to Prim’s algorithm for every vertex in an array of size $n$ we store a pointer $point[v]$ that points to the position of the vertex in the linked list/heap. Therefore DecreaseMin($v$) (lines 10-12), first locates the list attached to element 1 and if the value needs to be decreased (i.e. from 20 to 10) then the element pointed to by $point[v]$ is deleted in constant time, a new element is inserted in the linked list of element 0 and $point[v]$ points to this new location (and also the new location points to $v$).
Problem 8.
Improvis. No solutions for this problem.

Problem 9.
coll@a,b,c means a,b,c keys caused collision in the indicated slot.

<table>
<thead>
<tr>
<th>Slot</th>
<th>Linear Probing</th>
<th>Quadratic Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 :</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1 :</td>
<td>1 coll@12,23,34</td>
<td>1 coll@12,23,34</td>
</tr>
<tr>
<td>2 :</td>
<td>12 coll@23,34</td>
<td>12 coll@23,34</td>
</tr>
<tr>
<td>3 :</td>
<td>3 coll@23,34</td>
<td>3</td>
</tr>
<tr>
<td>4 :</td>
<td>23 coll@34</td>
<td>34</td>
</tr>
<tr>
<td>5 :</td>
<td>5 coll@34</td>
<td>5 coll@23,34</td>
</tr>
<tr>
<td>6 :</td>
<td>6 coll@17,34</td>
<td>6 coll@17,34</td>
</tr>
<tr>
<td>7 :</td>
<td>17 coll@34</td>
<td>17</td>
</tr>
<tr>
<td>8 :</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>9 :</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 :</td>
<td>10</td>
<td>23 coll@10,34</td>
</tr>
</tbody>
</table>

In the Linear probing case you can observe the long cluster that causes problems to 34. Observe that for quadratic probing the collisions are fewer (by one).