Goals
Practice hash tables, induction, graphs.

Questions

1. **Hashing:** Quadratic probing is an alternative way to resolve collisions. It is similar to linear probing in the sense that if a collision occurs, another index will be probed, but if, say, the key is hashed to index \( i \), instead of probing \( i + 1 \mod m \), \( i + 2 \mod m \), \( i + 3 \mod m \) etc., the probing is done in quadratically larger jumps, so the probing chain in this case is \( i + 1 \mod m \), \( i + 4 \mod m \), \( i + 9 \mod m \) etc., and in general the \( k^{th} \) probe step is \( i + k^2 \mod m \).

   Given a hash table of size 11, the key data is the following identifiers for some inventory: A29, C42, E12, D31, F08 and G34, B10. The hash function is \( H(LN) = ((L-'A') + N) \mod 11 \), where \( L \) is the letter, \( N \) is the number and 'A' is the ASCII value of A. For example: \( H("C29") = (2 + 29) \mod 11 = 9 \) since 'C' - 'A' = 2. For your convenience — the multiples of 11 are 11, 22, 33, 44, 55 etc. (such that the two digits are the same).

   (a) Draw the final configuration of the table after all the elements are inserted in the following illustration using linear probing. Do not rehash.

   ![Linear Probing Table](image)

   (b) Do the same for quadratic probing. Again, do not rehash.

   ![Quadratic Probing Table](image)

2. **Graphs: K&T, Ch. 3 q.7:** Some friends of yours work on wireless networks, and they’re currently studying the properties of a network of \( n \) mobile devices. As the devices move around (actually, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the \( n \) devices, and there is an edge between device \( i \) and device \( j \) if the physical locations of \( i \) and \( j \) are no more than 500 meters apart. (If so, we say that \( i \) and \( j \) are “in range” of each other.) They’d like it to be the case that the network of devices is connected at all times, and so they’ve constrained the motion of the devices to satisfy the following property: at all times, each device \( i \) is within 500 meters of at least \( n/2 \) of the other devices. (We’ll assume \( n \) is an even number.) What they’d like to know is: Does this property by itself guarantee that the network will remain connected? Here’s a concrete way to formulate the question as a claim about graphs.

   **Claim:** Let \( G \) be a graph on \( n \) nodes, where \( n \) is an even number. If every node of \( G \) has degree at least \( n/2 \), then \( G \) is connected. Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

3. **Graphs: K&T, Ch. 3 q.10:** Suppose we are given an undirected graph \( G = (V, E) \), and we identify two nodes \( v \) and \( w \) in \( G \). Give an algorithm that computes the number of shortest \( v-w \) paths in \( G \). (The
algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with $n$ nodes and $m$ edges. **Hint:** Slightly modify the BFS algorithm to also account for the number of possible paths to each node, but do it in an efficient way that does not require you to add any (asymptotic) runtime.

4. **Directed graphs:** Trace Depth-First search (DFS) on the following graph, starting from $v_1$. Mark all vertices and edges in order of visitation. To make the search fully specified, if two or more options exist, break tie by alphabetical order. Mark by * the edges that participate in the DFS tree.

```
\text{v}_2
\text{v}_1
\text{v}_5
\text{v}_4
\text{v}_3
\text{v}_6
```

5. **DAG:** A student suggested to topologically sort a DAG $G = (V, E)$ by first running BFS from a vertex $s$ with an in-degree of 0, and then listing the vertices in the order of their distance from $s$. Show a simple example where this algorithm will not work. You may assume $G$ is indeed a DAG, so no wise-assery.

6. **Graphs:** K&T, Ch. 3 q.3: The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG. Here is the algorithm:

- Find a node $v$ with no incoming edges and order it first
- Delete $v$ from $G$
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$

But suppose that we are given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph $G$, it outputs one of two things: (a) a topological ordering, thus establishing that $G$ is a DAG; or (b) a cycle in $G$, thus establishing that $G$ is not a DAG. The running time of your algorithm should be $O(m + n)$ for a directed graph with $n$ nodes and $m$ edges.