Q1:

[TRUE/FALSE] FALSE

Suppose $f(n) = f\left(\frac{n}{2}\right)$+56, then $f(n) = \Theta(n)$

[TRUE/FALSE] TRUE

Maximum value of an s-t flow could be less than the capacity of a given s-t cut in a flow network.

[TRUE/FALSE] FALSE

For edge any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer k>1, then that edge will no longer be part of the minimum cut.

[TRUE/FALSE] FALSE

Any problem that can be solved using dynamic programming has a polynomial worst case time complexity with respect to its input size.

[TRUE/FALSE] FALSE

In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems

[TRUE/FALSE] FALSE

In the divide and conquer algorithm to compute the closest pair among a given set of points on the plane, if the sorted order of the points on both X and Y axis are given as an added input, then the running time of the algorithm improves to O(n).

[TRUE/FALSE] TRUE

If f is a max s-t flow of a flow network G with source s and sink t, then the value of the min s-t cut in the residual graph G_f is 0.

[TRUE/FALSE] TRUE

Bellman-Ford algorithm can solve the shortest path problem in graphs with negative cost edges in polynomial time.

[TRUE/FALSE] TRUE

Given a directed graph G=(V,E) and the edge costs, if every edge has a cost of either 1 or -1 then we can determine if it has a negative cost cycle in O(|V|^3) time..

[TRUE/FALSE] TRUE

The space efficient version of the solution to the sequence alignment problem (discussed in class), was a divide and conquer based solution where the divide step was performed using dynamic programming.

Q2:

The problem can be formulated as a dynamic programming problem in many different ways (These are the correct solutions to the three most common formulations used by students in the exam):

1) OPT[v] will denote the minimum number of coins required to make change for value "v".
   The recursive formula would be:
   $$OPT[T] = \min_{1 \le i \le n} \{OPT[T - X_i] + 1\}$$
   The boundary values will be as follows:
   $$OPT[0] = 0$$
   $$OPT[v] = \infty \; if \; v < 0$$
   We fill in the *OPT[]* array in the following order:

```
for (int i = 1; i <= v; i++) {
   int min = infinity;
   for (int x : {X1, X2, … , Xn}) {
      if (OPT[i – x] + 1 > min)
         min = OPT[i – x] + 1;
   }
   OPT[i] = min;
}
```

   At the end if $OPT[v] \le k$ we return success, otherwise we return failure.

2) $OPT[k, v]$ denotes the minimum number of coins required to make change for v using at most k coins: Then:
   $$OPT[k, v] = \min\left\{OPT[k-1, v], \min_{1 \le i \le n} \{OPT[k-1, v - X_i] + 1\}\right\}$$
   Here the boundary values will be:
   $$OPT[k, 0] = 0$$
   $$OPT[0, v] = \infty \; if \; v \neq 0$$
   $$OPT[k, v] = \infty \; if \; v < 0$$
   We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
   for (int j = 1; j <= k; j++) {
      int min = OPT[j-1, i];
      for (int x : {X1, X2, … , Xn}) {
         if (OPT[j-1, i – x] + 1 > min)
            min = OPT[j-1, i – x] + 1;
      }
      OPT[j, i] = min;
   }
```

}

Like the previous formulation, if $OPT[k, v] \leq k$ we return success, otherwise we return failure.

3) $OPT[k, v]$ is a binary value denoting whether it's possible to make change for v using at most k coins. Then:

$$OPT[k, v] = OPT[k-1, v] \vee \left\{ \bigvee_{1 \leq i \leq n} OPT[k-1, v - X_i] \right\}$$

The boundary values will be:

$$OPT[k, 0] = true$$
$$OPT[0, v] = false$$
$$OPT[k, v] = \ false \ if \ v < 0$$

We should fill the $OPT[]$ array in the following order:

```
for (int i = 1; i <= v; i++) {
   for (int j = 1; j <= k; j++) {
      OPT[i, j] = OPT[j-1, i];
     for (int x : {X1, X2, ... , Xn}) {
         if (OPT[j-1, i – x]) OPT[j, i] = true;
      }
   }
}
```

At the end, we only need to return $OPT[k, v]$ as the result.

Q3:

This problem can be mapped to a network flow problem. First assign a node s_i for each student and node n_i for each night. We also add two nodes S and T . Now we need to assign the edges and capacities. We create an edge between S and each student node with capacity 1 . We also create an edge between each night node and T with capacity 1. Then we need to connect student node s_i with the night node n_j if s_i is capable to cook on night n_i. The capacity for this edge is also 1. Now we only need to run ford-Fulkerson algorithm to find the maximum flow and see if this flow is equal to n or not.

Q4:

Part a) We present a few proofs of the claim. The first two are perhaps the most straightforward but the third leads naturally to an algorithm for part b.

Proof 1: Since A is a finite array, it has a minimum. Let j denote an index where A is minimum. If j is not in the boundary (that is j is neither 1 nor n), then j is a local minimum as well. If j is 1, then since A[1]>=A[2], A[2] is the minimum values in the array and 2 is a local minimum. Likewise, if j is n, then since A[n]>=A[n-1], A[n-1] is the minimum value in A and thus n-1 is a local minimum. Thus in every case, we may conclude that there is a local minimum.

Proof 2: Assume that A does not have a local minimum. Since A[1]>=A[2], this implies that A[2]>A[3] (otherwise, 2 would be a local minimum). Likewise A[2]>A[3] implies that A[3]>A[4] and so on. In particular A[n-1]>A[n], contradicting the fact that A[n]>=A[n-1]. Thus our assumption is incorrect.

There is also an analogous proof that goes from right to left.

Proof 3. We prove the claim by induction. If n=3, then 2 is a local minimum. Consider A[1...n] with n>3, A[1]>=A[2] and A[n]>=A[n-1]. As the induction hypothesis, assume that all arrays B[1....k] with 2<k<n, B[1]>=B[2] and B[k]>=B[k-1] have a local minimum. Let j=floor(n/2). If A[j]<=A[j+1], then A[1....j+1] has a local minimum (by the induction hypothesis) and hence A[1....n] has a local minimum. Else (that is, A[j]>A[j+1]), then A[j....n] has a local minimum (by the induction hypothesis) and hence A[1....n] has a local minimum.

Part b) A divide and conquer solution for part b follows immediately from the third proof for part a. If n=3, then 2 is a local minimum. If n>3, set j=floor(n/2). If A[j]<=A[j+1], then search for a local minimum in A[1....j+1]. Else, search for a local minimum in A[j....n]. Let T(n) denote the number of pairwise comparisons performed by our recursive algorithm for finding a local minimum in A[1...n]. Then T(n)<=T(ceil(n/2))+1 which implies that T(n) = O(log(n)).

Q5:

(a) The number of unique ways are shown as follows:

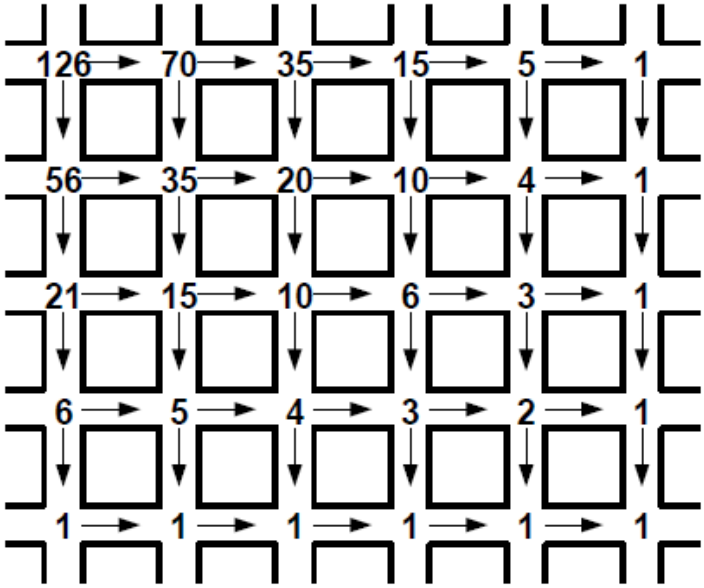

Figure A.

Answer: 126

(b) Define OPT(i,j) as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :
OPT(i,j) = OPT(i+1, j) + OPT(i,j+1) , for i<5, and j<4
Boundary condition: OPT(5,j) = 1; OPT(i,4) = 1

Alternative solution:
If you define OPT(i,j) as the number of unique ways from intersection S: (0,0) to intersection (i,j), you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

(c) With dead ends, the numerical results of the number of unique ways are shown as follows:

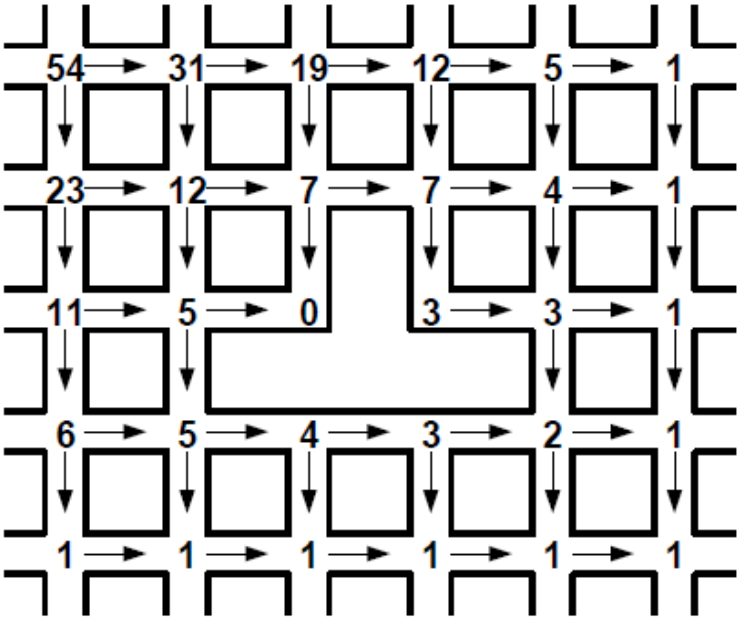

Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t, and add edges to t from each node in S.
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t.
4. Find the min q-t cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes O(|S|) times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S, but min-cut only works when S is a single node. We fix this by creating t directly connected from S, but we want to make sure the min-cut's cut-set doesn't include any edge connecting t, which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S.

Q5:

(a) The number of unique ways are shown as follows:

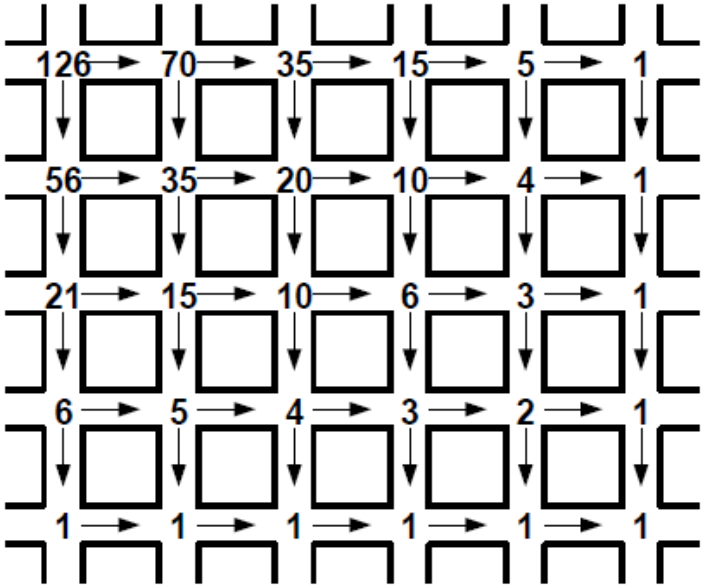

Figure A.

Answer: 126

(b) Define OPT(i,j) as the number of unique ways from intersection (i,j) to E: (5,4). The recursive relation is :
OPT(i,j) = OPT(i+1, j) + OPT(i,j+1) , for i<5, and j<4
Boundary condition: OPT(5,j) = 1; OPT(i,4) = 1

Alternative solution:
If you define OPT(i,j) as the number of unique ways from intersection S: (0,0) to intersection (i,j), you can also get the correct answer, but the recursive relation and boundary conditions should correspondingly changes.

(c) With dead ends, the numerical results of the number of unique ways are shown as follows:

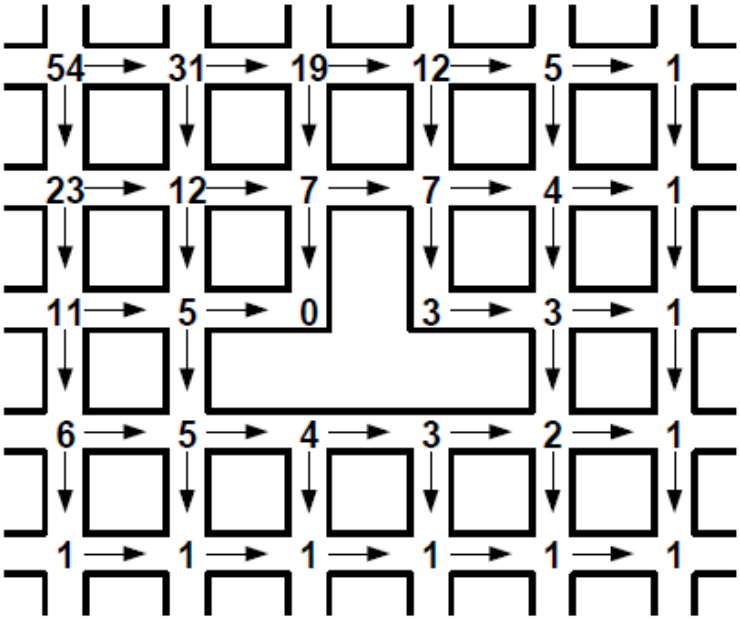

Figure B.

Answer: 54

Q6:

Algorithm:

1. Add node t, and add edges to t from each node in S.
2. Set the capacity of each of these new edges as infinity.
3. Set spammer node q as the source node. Then the communication network G becomes a larger network G' with source q and sink t.
4. Find the min q-t cut on G' by running a polynomial time max-flow algorithm.
5. Install a spam filter on each edge in the min-cut's cut-set.

Complexity:

Since the construction of the new edges takes O(|S|) times, together with the polynomial time of the min-cut algorithm, the entire algorithm takes polynomial time.

Justification:

We're simply trying to separate q from S, but min-cut only works when S is a single node. We fix this by creating t directly connected from S, but we want to make sure the min-cut's cut-setdoesn't include any edge connecting t, which is why we put the capacities arbitrarily large on these edges. Min-cut would then find the min-cost way to separate q from S.