This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

There is an $n \times n$ grid of squares. Each square is either *special*, or has a positive integer *cost* assigned to it. No square on the border of the grid is special.

A set of squares S is said to be *good* if it does not contain any special squares and, starting from any special square, you cannot reach a square on the border of the grid by performing up, down, left and right moves without entering a cell belonging to S.

1.1 **[5 marks]** In the following diagram, squares are labelled 'X' if they are special, otherwise they are labelled by their cost. Identify a good set of squares with minimum total cost for this particular grid.

5	3	4	9
4	X	3	6
1	9	X	4
1	2	3	5

A good set of squares with minimum total cost is shown below, with the values of the cells in the set in bold.

5	**3**	4	9
4	X	**3**	6
1	9	X	**4**
1	**2**	**3**	5

1.2 **[15 marks]** Design an algorithm which receives an arbitrary $n \times n$ grid, runs in time polynomial in n and determines a good set of squares with minimum total cost.

Let $C(r, c)$ denote the cost of the square at row r and column c.

We construct a flow network as follows.

- Let (r, c) denote a vertex representing the square in the rth row and cth column.
- For each vertex (r, c), construct the edges:
  - (r, c) to $(r, c+1)$ with capacity $C(r, c+1)$.
  - (r, c) to $(r, c-1)$ with capacity $C(r, c-1)$.
  - (r, c) to $(r+1, c)$ with capacity $C(r+1, c)$.
  - (r, c) to $(r-1, c)$ with capacity $C(r-1, c)$.

  If the edge leads to a special square, or leads to a square outside the grid, do not construct the edge.
- Add a super source vertex s, which represents all special squares. Connect s to each special square with a capacity of ∞.
- Add a super sink vertex t, which represents the border of the grid. Connect all squares at the border of the grid to t with a capacity of ∞.

Now we can find the maximum flow of this flow network via the Edmonds-Karp algorithm. This is equal to the minimum cut, by the Max-Flow Min-Cut Theorem.

Furthermore, we can recover the edges of the minimum cut, and use this to determine S, as follows. Run a depth first search on the residual graph starting from the vertex s, only considering edges with positive residual capacity. Now for each original edge in the network, if it connects a visited vertex to an unvisited vertex (that is, it is an edge contributing to the capacity of the minimum cut), include the square corresponding to the unvisited vertex in S.

We claim that the set S is a good set with minimum total cost. Firstly, it is a good set, because any path from a special square to a border square must go through an edge in the minimum cut, which would enter a vertex in S. Also, it has minimum total cost, since for every good set S, there is a corresponding cut in the flow with the same cost, consisting of the edges which go from a square reachable from any special square to a cell in S.

The algorithm involves Edmonds-Karp, depth first search, then iterating through each edge. Letting $V = O(n^2)$ be the number of the vertices and $E = O(n^2)$ be the number of edges, we obtain a time complexity of $O(VE^2) + O(V) + O(E) = O(n^6)$, which is polynomial in n.

Question 2

There are k people living in a city, whose n suburbs and m roads can be represented by an unweighted directed graph.

Every person is either slow or fast. Every night,

- Each slow person can stay in the same suburb, or move along a road to an adjacent suburb.
- Each fast person can stay in the same suburb, or move to any other suburb in the entire city.

Over the last d days, you know how many people were located in each suburb. That is, for each day i and suburb j (where $1 \leq i \leq d$ and $1 \leq j \leq n$), you know $p_{i,j}$, the number of people located in suburb j on day i. You are guaranteed that $\sum_{j=1}^{n} p_{i,j} = k$ for all i.

2.1 **[10 marks]** Design an algorithm which runs in $O(d(n+m))$ time and determines whether there could possibly be at least one slow person.

Construct a flow network as follows:

- For each day i and suburb j, where $1 \leq i \leq d$ and $1 \leq j \leq n$, construct vertices $in_{i,j}$ and $out_{i,j}$.
- For each day i and suburb j, where $1 \leq i \leq d$ and $1 \leq j \leq n$, add an edge from the vertex $in_{i,j}$ to the vertex $out_{i,j}$ with capacity $p_{i,j}$.
- For each day i and suburb j, where $1 < i \leq d$ and $1 \leq j \leq n$, add an edge from the vertex $out_{i-1,j}$ to the vertex $in_{i,j}$ with capacity k.
- For each day i and suburb j, where $1 < i \leq d$ and $1 \leq j \leq n$, for each suburb j' adjacent to j, add an edge from the vertex $out_{i-1,j}$ to the vertex $in_{i,j'}$ with capacity k.
- Add a super sink s, with an edge from s to $in_{1,j}$ with capacity $p_{1,j}$ for all $1 \leq j \leq n$.
- Add a super sink t, with an edge from $out_{d,j}$ to t with capacity $p_{d,j}$ for all $1 \leq j \leq n$.

We claim that there could possibly be at least one slow person if and only if there exists a path of positive weight from s to t. This is because there is a bijection between paths of positive weight from s to t, which are of the form $s, in_{1,v_1}, out_{1,v_1}, in_{2,v_2}, \ldots, out_{d,v_d}, t$, and slow people who are at suburb v_1 on day 1, suburb v_2 on day 2, and so on, since $p_{i,v_i} \geq 1$ means there is at least one person in v_i on day i, and the fact that v_1 is the same as or adjacent to v_2, v_2 is the same as or adjacent to v_3 and so on means that a slow person could have made this journey.

Thus we should run a depth-first (or breadth-first) search from s, and if t is ever visited, return an answer of YES, otherwise return an answer of NO. There are $dn + 2 = O(dn)$ vertices and $dn + (d-1)n + (d-1)m + 2n = O(d(n+m))$ edges in the constructed flow network, giving a total time complexity of $O(d(n+m))$ since depth first search runs in linear time.

2.2 **[10 marks]** Design an algorithm which runs in time polynomial in n, m, and d, and determines the minimum possible number of fast people.

We first construct the same flow network as we did for 2.1. Now we run the Edmonds-Karp algorithm on this flow network to obtain the value of a maximum flow. Since there are $O(dn)$ vertices and $O(d(n+m))$ edges, the runtime of $O(VE^2)$ is indeed polynomial in n, m and d.

We claim that the value of a maximum flow, say $f_{\max}$ in the network is the maximum number of slow people, and so the minimum number of fast people is $k - f_{\max}$.

This is true because in any flow, the capacity of each edge from $out_{i-1,j}$ to the vertex $in_{i,k}$ can represent the number of slow people moving from city j to city k between day $i-1$ and day i in some valid allocation of slow and fast people to cities. The constraint that each slow person stays in the same suburb or moves to an adjacent one is enforced by how the edges between the main vertices are connected. The constraint that exactly $p_{i,j}$ people are in suburb j on day i is enforced by the vertex capacities which ensure that at most $p_{i,j}$ slow people are in suburb j on day i, and the fast people are able to occupy the remaining positions since they can be anywhere at any time. Thus the maximum flow represents the maximum number of slow people who could possibly be in the city.

Question 3

There n boys and n girls at a party. Whenever a song starts, they will form exactly n pairs to dance. No boy will dance with the same girl twice.

Some pairs of boys and girls like each other, and all other pairs of boys and girls dislike each other. Every boy will dance with at most k girls that he dislikes, and each girl will dance with at most k boys that she dislikes, where $k < n$.

As the DJ, your job is to determine the maximum number of songs you can play, such that it is possible for pairs to be formed for all songs according to the above requirements.

Design an algorithm which achieves this and runs in time polynomial in n:

3.1 [**10 marks**] for $k = 0$.

Label the boys $b_1, b_2, \cdots, b_n$, and the girls $g_1, g_2, \cdots, g_n$, and fix $x \le n$. Construct a flow network with:

- a vertex b_i for each boy, and a vertex g_j for each girl
- vertices s and t, the super source and super sink
- for each boy, an edge from s to b_i with capacity x
- for each girl, an edge from g_j to t with capacity x
- for each boy-girl pair who like each other, an edge from b_i to g_j with capacity 1.

The total capacity leaving s (and entering t) is nx, so the value of a maximum flow is at most nx. If x songs can be played, then we can construct a flow of value nx by flowing:

- all edges (s, b_i) with x units of flow
- all edges (g_j, t) with x units of flow
- for each pair (b_i, g_j) who dance together, the edge (b_i, g_j) with one unit of flow.

Proving the converse (that a flow of exactly nx allows x songs to be played) is more subtle.

Proof: Consider a maximum flow in the network. For each edge (b_i, g_j) which carries flow, we will record that boy i and girl j dance together for one song. If the maximum flow is nx, we will thus find x partners for each attendee, while enforcing that:

- no pair who dislike each other dances together, and
- no pair dances together more than once.

However, we have yet to confirm whether the nx boy-girl pairings can be grouped into x songs. The proof of this fact was not required to achieve full marks, but it is included here for completeness.

Construct a graph with $2n$ vertices corresponding to the boys and girls, with nx edges corresponding to the matched pairs. This graph is clearly bipartite, with parts $B = \{b_1, \ldots, b_n\}$ and $G = \{g_1, \ldots, g_n\}$, and x-regular (every vertex is incident to exactly x edges). Now, we use Hall's marriage theorem.

Definition: For a subset W of B, let $N(W) \subseteq G$ be the set of vertices adjacent to at least one vertex in W.

Theorem[a]: Suppose for all subsets W of B that $|W| \le |N(W)|$. Then the graph has a perfect matching, i.e. a matching of size n.

For a contradiction, suppose there is a set W of p boys to which only $q < p$ girls are adjacent. There are exactly px edges between W and $N(W)$, since each boy is matched with exactly x girls. However, each of these edges is also incident to exactly one of the q girls, and since $p > q$ it is impossible for all q of these girls to have fewer than x edges each. Therefore, we have the required contradiction, and it follows that a perfect matching exists (corresponding to the pairs who dance together in the first song). Furthermore, upon removing these edges, the remaining graph is $(x-1)$-regular, so the same property applies. We can thus make n pairs for each of the x songs, as required. $\square$

Thus, we have a test for whether x songs can be played or not. We can then apply this test for $x = 0, 1, 2, \ldots$ until we find the largest value of x for which it is possible, which is the answer.

We run the Edmonds-Karp algorithm up to $n+1$ times. In each iteration, the number of edges is $O(n^2)$ and the maximum flow is $O(n^2)$, so the total time complexity is $O(n^5)$. Note that this can be improved to $O(n^4 \log n)$ by binary searching for the largest allowable value of x. This is clearly polynomial in n - running max flow on $O(n)$ vertices, $O(n^2)$ edges at most n times.

[a] Note that this is a special case of the theorem; the full theorem applies to bipartite graphs where the parts are not necessarily equal in size.

3.2 [**10 marks**] for arbitrary k.

You may choose to skip 3.1, in which case we will mark your submission for 3.2 as if it was submitted for 3.1 also.

The fundamental structure is as above: for each x from 0 to n, run the Edmonds-Karp algorithm once to determine whether x songs can be played, and return the largest x for which it was possible. However, the graph construction is more intricate. For a particular value x, construct a flow graph with:

- vertices b_i, b_i^L and b_i^D for each boy
- vertices g_j, g_j^L and g_j^D for each girl
- vertices s and t, the super source and super sink
- for each boy:
  - an edge from s to b_i with capacity x
  - an edge from b_i to b_i^L with capacity x
  - an edge from b_i to b_i^D with capacity k
- for each girl:
  - an edge from g_j to t with capacity x
  - an edge from g_j^L to g_j with capacity x
  - an edge from g_j^D to g_j with capacity k
- for each boy-girl pair:
  - an edge from b_i^L to g_j^L with capacity 1 if they like each other, or
  - an edge from b_i^D to g_j^D with capacity 1 if they don't like each other.

As above, we will find a maximum flow in this graph, and record each boy-girl edge carrying flow as a pair who dance together. This guarantees that:

- no pair dances together more than once, since for each pair (i, j), either
  - they like each other, so $c(b_i^L, g_j^L) = 1$ and $c(b_i^D, g_j^D) = 0$, or
  - they don't like each other, so $c(b_i^L, g_j^L) = 0$ and $c(b_i^D, g_j^D) = 1$.
- each boy and each girl has exactly x partners
- of these partners, at most k are not liked.

The asymptotic time complexity is unaffected, since the number of edges is still $O(n^2)$.

Note that setting $k = 0$ in this graph recovers the construction from 3.1.

Question 4

You are given a flow network G with n vertices, including a source s and sink t, and m directed edges with integer capacities.

Your friend will ask you several queries of the form: "If an edge were to be added to G, going from vertex a to vertex b with capacity c, what would the maximum flow of the modified network be?" Note that each query considers adding an edge to the original flow network, so the modified network has $m + 1$ edges.

Before asking any queries, your friend gives you some time to prepare, which you can spend on precomputation.

Design an algorithm which performs any required precomputation and then answers each query in constant time, subject to the following restrictions:

4.1 **[5 marks]** $O(nm^2)$ precomputation; $b = t$ and $c = 1$ in all queries.

During the precomputation, first run the Edmonds-Karp algorithm to obtain a maximum flow in the original network. Let the value of a maximum flow be $f_{\max}$. Next, on the residual network, we run a depth first search starting from vertex s, only considering edges with positive capacity. Let S be the set of vertices visited during this depth first search, and set up a lookup table to be able to check whether a vertex is in S. The Edmonds-Karp algorithm takes $O(nm^2)$ time, the depth first search takes $O(n + m)$ time, and setting up the lookup table takes $O(n)$ time, so we have used $O(nm^2)$ time for precomputation.

When answering queries, we check if a is in S, which can be done in constant time with the lookup table. If a is in S, the answer to the query is $f_{\max} + 1$, otherwise the answer is $f_{\max}$. We now justify the correctness of this method, by considering how the Ford-Fulkerson algorithm may have proceeded in the modified network.

Firstly, if a is not in S, then Ford-Fulkerson's algorithm could have initially produced the same flow as was produced in the original network. After this, even with the addition of the new edge, there would be no augmenting path (since any augmenting path would have to leave S) and so Ford-Fulkerson's algorithm would terminate. Thus in this case, $f_{\max}$ is still the maximum flow.

Secondly, if a is in S, then Ford-Fulkerson's algorithm could have produced the same flow as was produced in the original network, before finally adding an augmenting path from s to b whose final edge is the new edge from a to b (which exists because a being in S means there is a path from s to a with positive integer capacities). This would add 1 to the value of the flow, giving a flow with value $f_{\max} + 1$. There would be no further augmenting paths, since they would have to leave S. Thus in this case, $f_{\max} + 1$ is the maximum flow.

4.2 **[6 marks]** $O(nm^2)$ precomputation; $c = 1$ in all queries.

During the precomputation, we first run the same steps as we did in 4.1 to obtain the value $f_{\max}$, the set S and the residual network. Now, on the residual network, we reverse the direction of every edge, then run a depth first search starting from vertex t. Let T be the set of vertices visited during this depth first search, and set up a lookup table so we can check in constant time whether a vertex is in T. This depth first search takes $O(n)$ time, keeping our precomputation time to $O(nm^2)$.

When answering queries, we check whether a is in S and whether b is in T in constant time. If a is in S and b is in T, the answer to the query is $f_{\max} + 1$, otherwise the answer is $f_{\max}$. The justification of correctness of this method, which we outline now, is very similar to that for 4.1.

If a is not in S or b is not in T, the new edge cannot result in a new augmenting path, since any augmenting path would have to leave S and enter T, so in this case $f_{\max}$ is indeed still the maximum flow. The existence and maximality of a flow with with value $f_{\max} + 1$ in the case that a is in S and b is in T is shown by considering an augmenting path from s to t via the edge from a to b.

4.3 [**7 marks**] $O(n^2m^2)$ precomputation; $b = t$ in all queries.

During the precomputation, we first run the same steps as we did in 4.1 to obtain the value $f_{\max}$, the set S and the residual network in $O(nm^2)$ time. Next, for each vertex u in the graph, we run the Edmonds-Karp algorithm to determine the maximum value of a flow from s to u in the residual network, which we denote as $f(u)$. There are n vertices to iterate over, each requiring $O(nm^2)$ time to run the Edmonds-Karp algorithm, so the total precomputation time is $O(n^2m^2)$.

The answer to each query is $f_{\max} + \min(c, f(a))$. This is because on the modified network, Ford-Fulkerson's algorithm first could have produced the same flow as was produced in the original network, before adding some sequence of augmenting paths from s to t whose final edge is the new edge from a to t (because S will remain disconnected from t except via a), effectively finding the maximum flow from s to t in the residual network combined with the new edge. This would terminate either when the new edge is fully saturated, at which point the total flow would be $f_{\max} + c$, or when there is no augmenting path from s to a, which would occur if a flow in the residual network of $f(a)$ was achieved, giving a total flow of $f_{\max} + f(a)$. This shows that the new value of the maximum flow is $f_{\max} + \min(c, f(a))$.

4.4 [**2 marks**] $O(n^2m^2)$ precomputation; queries unrestricted.

During the precomputation, we first run the same steps as we did in 4.2 to obtain the value $f_{\max}$, the sets S and T, and the residual network in $O(nm^2)$ time. We also run the same steps as we did in 4.3, obtaining the values $f(u)$ for each vertex u in the graph. Next, for each vertex u in the graph, we run the Edmonds-Karp algorithm to determine the maximum value of a flow from u to t in the residual network, which we denote as $g(u)$. There are n vertices to iterate over, each requiring $O(nm^2)$ time to run the Edmonds-Karp algorithm, so the total precomputation time is $O(n^2m^2)$.

The answer to each query is $f_{\max} + \min(c, f(a), g(b))$, for similar reasons to 4.3, which we outline now. On the modified network, Ford-Fulkerson's algorithm first could have produced the same flow as was produced in the original network, before adding some sequence of augmenting paths from s to t via the edge from a to b (because S will remain disconnected from T except via the edge from a to b). This would terminate either when the new edge is fully saturated, giving a total flow of $f_{\max} + c$, or when there is no augmenting path from s to a, giving a total flow of $f_{\max} + f(a)$, or when there is no augmenting path from b to t, giving a total flow of $f_{\max} + g(b)$. This shows that the new value of the maximum flow is $f_{\max} + \min(c, f(a), g(b))$.