1 (30 pts) Amortization

(a) Red and Blue Nodes.
A splay tree is being used to store \( n + m \) items; \( n \) of them are red and \( m \) of them are blue.

Using the standard splay tree potential function (with appropriately chosen weights) prove the following two statements:

- The amortized number of splay steps (as defined in lecture) done when a red item is accessed is at most \( 4 + 3 \log_2 n \).
- The amortized number of splay steps (as defined in lecture) done when a blue item is accessed is at most \( 4 + 3 \log_2 m \).

**Solution:** Let the weight of each red node be \( 1/n \), and the weight of each blue node be \( 1/m \). The total weight of the tree is 2. (Of course it will also work out the same if we give the red nodes weights of \( 1/2n \) and the blue nodes weights of \( 1/2m \).)

Now by the access lemma, the amortized cost of splaying a red node \( x \) is at most:

\[
3(r(root) - r(x)) + 1 = 3(1 - r(x)) + 1 = 4 - 3r(x)
\]

\[
= 4 + 3(-r(x))
\]

But \( r(x) \geq \log(1/n) = -\log n \), therefore \( -r(x) \leq \log n \). Thus we conclude that the amortized cost of splaying is at most \( 4 + 3 \log n \). The proof for the blue nodes is identical.

This is curious because if there are very few red items, it seems to become very efficient to access them. However the algorithm itself does not even know which items are red and which are blue! Can you explain this?

**Solution:** There are many ways to look at this. Say that the nodes are accessed randomly (all with equal probability). Because the bound is amortized, we can simply move the cost of the red accesses to the blue accesses.

On the other hand, suppose that most of the accesses are to the red nodes. Then because there are few red nodes there must be many times when a red node is accessed, then soon thereafter accessed again. The second access is cheap because the node will still be near the root of the tree.

(b) A Deep Splay.
i. You have a splay tree of 1,000,000 nodes. Each node has a weight of 1. Using these weights, the usual sizes and ranks of all nodes are computed. You splay a node \( x \) that is at depth 1000 in the tree. Consider the 500 splay steps that are done. Call a splay step “scary” if the rank of \( x \) does not change during the step. What is the minimum number of scary splay steps that can be occur?

**Solution:** The rank of the root of the tree is 19. The initial rank of \( x \) is at least 0. The final rank of \( x \) is 19. For each splay step the rank of \( x \) either increases or remains the same. Therefore among the 500 splay steps, the rank can increase a maximum of 19 times. Therefore for the remaining 481 splay steps the rank cannot increase. Thus there are at least 481 scary steps.

ii. Continuing with the scenario in part i., you correctly reason that the potential ought to decrease. Find the largest value of \( k \) you can such that it’s guaranteed that the potential will decrease by at least \( k \).

**Solution:** By the access lemma, the amortized number of splay steps done in this scenario is at most \( 3 \times (19 - 0) + 1 = 58 \). And we know that the amortized number of splay steps is the change in potential plus the actual cost (which is 500). So we have:

\[
\Delta \Phi + 500 = \text{amort # splay steps} \leq 58.
\]

Therefore \( \Delta \Phi \leq -442 \).
(c) Chopping in a Graph.

Let \( G_0 \) be the complete undirected graph of four vertices. Now, starting with \( G_0 \) the graph is updated \( n \) times. \( G_{i+1} \) is obtained from \( G_i \) by one of the following operations:

- **link\((u, v)\):** Add edge \((u, v)\) to the graph. \( u \) and \( v \) are distinct and not already an edge in the graph. Cost = 1.
- **chop\((u)\):** Vertex \( u \) (of degree \( d(u) \)) is deleted from the graph, and replaced by a collection of new vertices \( x_1, x_2, \ldots, x_{d(u)} \) connected in a cycle. The edges that used to be connected to \( u \) are each connected to a distinct one of these \( x_i \)s. After this step, each \( x_i \) is of degree three. (See the figure below.) The cost of this operation is \( d(u) \). (You can think of this as what happens when you chop a corner off of a polyhedron.)

Find a constant \( c \) (the smallest possible) and show that any sequence of links and chops has a total cost \( \leq cn \).

Structure your proof as follows: Define a potential function. Show that with this potential the amortized cost of chop() and link() are at most \( c \). Show that the initial potential is less than the final potential. (It may be useful to view the potential function as placing tokens on nodes of the graph, as in the analysis of splay trees.)

**Solution:** We'll choose \( c = 3 \). Let \( d(v, i) \) denote the degree of a vertex \( v \) in graph \( G_i \).

\[
\Phi(G_i) = \sum_{v \in G_i} (d(v, i) - 3)
\]

**Analysis of link:** The link operation costs one and increases the degrees of two vertices.

\[
\text{(amortized cost)} = \text{(cost of the operation)} + \text{(change in potential)} = 1 + 2 = 3
\]

**Analysis of chop:** Let \( k \) be the degree of the node being chopped.
(amortized cost) = (cost of the operation) + (change in potential)

(amideized cost) = (cost of the operation) + (new potential) − (old potential)

= \( k + 0 - (k - 3) \)

= 3

Converting total amortized cost to total actual cost: Recall the equation from lecture:

(total actual cost) = (total amortized cost) + (initial potential) − (final potential)

≤ \( 3n + 0 - (\text{final potential}) \)

≤ 3n

The last step follows from the fact that all degrees are always at least three, and thus all the potentials are always non-negative.

2. (25 pts) Flow Problems

(a) Carpools.

Say there are \( m \) days, and \( S_i \) is the set of people that carpool together on day \( i \). For each set \( S_i \), one of the people \( j \in S_i \) must be chosen to be the driver that day. Since people would rather not drive, they want the work of driving to be divided as fairly as possible. Your task in this problem is to give an algorithm to do this efficiently.

The fairness criterion is the following: Say that person \( p \) is in some \( k \) of the sets, which have sizes \( n_1, n_2, \ldots, n_k \), respectively. Person \( p \) should really have to drive \( \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_k} \) times, because this is the amount of resource that this person effectively uses. Of course this number may not be an integer, so let’s round it up to an integer. This quantity \( \lceil \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_k} \rceil \) is called their fair cost. A fair solution is a schedule for who drives on each day, such that each person drives no more than their fair cost.

For example, say that on day 1, Aram and Bob carpool together, and on day 2, Aram, Celia, and Dorothy carpool together. Aram’s fair cost would be \( \lceil 1/2 + 1/3 \rceil = 1 \). So Aram driving both days would not be fair. Any solution except that one is fair.

Give a polynomial-time algorithm that, given \( S_1, S_2, \ldots, S_m \), computes a fair solution. This will also show that there always exists a fair solution.

Solution: Consider the following flow-graph. We create a source node \( S \) and connect it by edges of capacity 1 to \( m \) nodes \( c_1, \ldots, c_m \), one for each committee. We then make one node for each person \( p_1, \ldots, p_n \), and connect \( c_i \) to \( p_j \) by an edge of capacity 1 if person \( j \) was in committee \( i \). (i.e., \( c_i \) is connected to all people in the committee \( C_i \)). Finally, we have a sink node \( T \), and connect each person to the sink. The capacity of the edge connecting some person \( p \) to the sink is equal to \( \lceil \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_k} \rceil \), where...
\(a_1, \ldots, a_k\) are defined as above (the sizes of the committees that \(p\) is in). [Equivalently, you could connect the source to the people and the sink to the committees.]

There are now two steps to the proof. Step 1 is to show that the max flow in this graph is \(m\). It is easy to see the flow can’t be any larger, since you could cut the source-committee edges. The fact that there really is a flow of size \(m\) comes from the observation that if we send one unit of flow from \(S\) into each of the \(c_i\), and then split \(c_i\)’s flow evenly among each of its outgoing edges, then there is enough capacity to get all that flow into the sink \(T\) by how we defined the capacities of the edges into \(T\). Alternatively, a more complicated way to do this is to show that the min cut has capacity \(m\). It helps for this if you make the \(c_i\) to \(p_j\) edges have a large capacity (say, \(> m\)), because then the min cut can’t contain any \(c_i \rightarrow p_j\) edges. (Note that this doesn’t change the solution because the flow out of any \(c_i\) is still at most 1). The cut that includes all \(S\) to \(c_i\) edges has capacity \(m\), and you can then show that any cut that includes a combination of \(S\) to \(c_i\) and \(c_i\) to \(T\) edges must have capacity \(\geq m\) (this argument is like the Hall’s theorem proof from recitation.

Step 2 is to notice that since all capacities are integers, if we run Ford-Fulkerson, we will find a max flow in which all flows are integral. Even without doing anything fancy (like Edmonds-Karp), Ford-Fulkerson will run in polynomial time since the max flow is just \(m\) (so it will make at most \(m\) iterations). This flow will match each committee to exactly one person, and by definition of our capacities, will be a fair solution.

(b) Dominos on a Mutilated Chessboard.  
You’re given an \(n \times m\) grid of squares, some of which are marked “forbidden”. The goal is to place as many \(2 \times 1\) dominos on the chessboard as possible. Each domino must cover two adjacent non-forbidden squares. Give an \(O(n^2m^2)\) algorithm to compute this.

Solution: First color the squares like a chessboard, so each black square is surrounded by up to four white squares, one in each of the four compass directions.

Now construct a flow graph as follows. There is a node for each non-forbidden square. There are also two special nodes \(s\) and \(t\). Add an edge from \(s\) to each non-forbidden black square. Now add an edge from each non-forbidden black square to its neighboring non-forbidden white squares. Finally add edges from each non-forbidden white square to \(t\). The edges all have capacity 1 in the specified direction.

The maximum flow in this graph from \(s\) to \(t\) is the answer we are searching for. It’s easy to verify that the flow can be turned into a domino tiling with the same number of dominos, and any domino tiling can be turned into a flow of value equal to the number of dominos.

Notice that this construction depends on the fact that the graph is bipartite. And this in turn is a consequence of the fact that you can color the squares in chessboard fashion.

Ford-Fulkerson will take \(O(n^2m^2)\). Let \(N = n \times m\), the area of the board. The bipartite flow graph has at most \(N + 2\) vertices (some of the squares are forbidden) and \(O(N)\) edges. Each pass through it (DFS or BFS) to find an augmenting path takes \(O(N)\). This only needs to be repeated \(O(N)\) times until the max flow is found. So the whole thing is \(O(N^2)\).
3. (20 pts) **Long Path or Large Independent Set**

Here we will see an application of DFS to approximating one of two problems in an undirected graph \( G(V,E) \) – longest path and shortest independent set. (An independent set is a set of vertices \( U \subseteq V \) with no edge in \( E \) connecting any two vertices in \( U \).)

(a) Prove that if \((u,v) \in E\), then in any DFS tree of \( G \), either \( u \) is an ancestor of \( v \) or vice versa.

**Solution:** All we have to do is show that there are no cross edges in the DFS, because all of the other types of edges (tree, forward edges, and back edges) connect two nodes, one of which is an ancestor of the other.

In the DFS of any undirected graph, there can be no cross edges. A cross edge \((u,v)\) connects two nodes which are roots of disjoint subtrees of nodes. Say the \( u \) is searched first. But before the DFS of \( u \) is completed, it should follow the edge \((u,v)\), and thus \( v \) will be a descendant of \( u \).

(b) Give an \( O(n+m) \)-time algorithm which either computes a simple path of length \( \lceil \sqrt{n} \rceil \) or an independent set of cardinality \( \lceil \sqrt{n} \rceil \).

**Solution:** For simplicity let's use \( z \) to denote \( \lceil \sqrt{n} \rceil \).

Run a DFS on \( G \). This will produce a forest of DFS trees. Label each node with its distance from the root in its DFS tree (just using the tree edges). The roots have distance 0. The children of the roots have distance 1, etc. This partitions all the vertices of the graph into a sequence of levels \( L_0, L_1, \ldots, L_{k-1} \) – The vertices of \( L_i \) are those at distance \( i \) from their root. Let \( k \) be the number of levels.

Observation 1: There exists a simple path of length \( k \). This is because any node in level \( k-1 \) has a parent in level \( k-2 \), etc. This lets us construct a simple path of \( k \) nodes.

Observation 2: Each level is an independent set. This follows immediately from part (a), because an edge inside a level would represent a cross edge, and there are no cross edges.

Let \( w \) the the size of the largest level. We know that \( k \cdot w \geq n \), because the levels partition the \( n \) nodes. And this inequality implies that either \( k \geq z \) or \( w \geq z \). (If both of these were false then \( k \cdot w \) would be less than \( n \).)

So if \( k \geq z \), then there is a simple path of length at least \( z \). And if \( w \geq z \) then there is an independent set (the biggest level) of size at least \( z \).

**Context:** Both longest path and independent set are NP-hard problems, and it is even NP-hard to approximate them within any \( n^{1-\epsilon} \) factor, for any constant \( \epsilon > 0 \). Nonetheless, this exercise shows that for any graph you can approximate (at least) one of these problems to within an \( O(n^{1/2}) \) factor, in linear time.

B2. (Bonus) **Counting Planar Spanning Trees**

You're given an undirected connected graph \( G = (V,E) \) where \( V = \{0,1,\ldots,n-1\} \). The vertices are placed in counter-clockwise order around a unit circle with vertex \( k \) placed at the point \((\cos(2\pi k/n), \sin(2\pi k/n))\).
Give an $O(n^3)$ algorithm to count the number of planar spanning trees of $G$. That is, a tree satisfies the requirements if no two of its edges cross in the interior of the unit circle, all of its edges are in $E$, and the tree connects all the vertices of $G$.

So, for example, if $G$ is the complete graph on four vertices, then there are 12 planar spanning trees that satisfy the conditions.
4. Computing an Ear Decomposition

An undirected graph of at least 3 vertices is bi-connected if there are two vertex-disjoint paths between every pair of vertices in it. For example, a cycle is bi-connected.

A useful algorithmic tool for bi-connected graphs is the ear decomposition. Given a graph $G = (V, E)$ with $n$ vertices and $m$ edges, the ear composition ED is a sequence $[P_1, P_2, \ldots, P_k]$. Here $P_1$ is a simple cycle in $G$, and $P_i$ $(i > 1)$ is a simple path in $G$. Every edge of $G$ occurs exactly once in the cycle and paths of the ED. The two end points of path $P_i$ $(i > 1)$ must have already appeared in a path or cycle occurring prior to it in the sequence. The intermediate vertices of the path (the ones that are not endpoints of the path) must not have already appeared. Every bi-connected graph of at least three vertices has an ED.

The following figure shows a graph and an ear decomposition of it. The cycle $P_1$ is shown in blue, the first path $P_2$ is shown in red, the second path $P_3$ is shown in green, and the last path (just an edge) $P_4$ is shown in purple.

The goal of this assignment is to write a fast program to compute an ear decomposition of a graph.

Input:

The first line of input consists of two numbers $n$ and $m$, the number of vertices and edges of a graph. $3 \leq n \leq 50000, n \leq m \leq 500000$. Each of the following $m$ lines contains one edge of the graph, represented by two vertex numbers. (The vertices are numbered $0, 1, \ldots, n - 1$.) The graph is guaranteed to be bi-connected, and there are no self-loops or multi-edges. The time limit will be 5 seconds.

Output:

The first line of output is $k$, the number of parts in the ear decomposition sequence. The next line contains the cycle $P_1$ of the ED. It begins with the number $\ell$ of edges on the cycle, and continues with $2\ell$ vertex numbers. For example if the line is:

```
3 0 4 4 5 5 0
```

This means that the cycle has three edges, and they are $(0, 4), (4, 5)$, and $(5, 0)$. The order of the edges and the order of the pair of vertex numbers within the edge does not matter.

The following $k - 1$ lines represent $P_2, P_3, \ldots, P_k$ in analogous fashion.
Examples:

The following input is for the graph illustrated above. The input is shown on the left. On the right is the ear decomposition that corresponds to the one in the figure.

<table>
<thead>
<tr>
<th>Input</th>
<th>Ear Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 9</td>
<td>4</td>
</tr>
<tr>
<td>0 1</td>
<td>3 0 4 4 5 5 0</td>
</tr>
<tr>
<td>1 2</td>
<td>3 4 3 3 1 1 0</td>
</tr>
<tr>
<td>2 3</td>
<td>2 1 2 2 0</td>
</tr>
<tr>
<td>3 4</td>
<td>1 2 3</td>
</tr>
<tr>
<td>4 5</td>
<td></td>
</tr>
<tr>
<td>5 0</td>
<td></td>
</tr>
<tr>
<td>4 0</td>
<td></td>
</tr>
<tr>
<td>3 1</td>
<td></td>
</tr>
<tr>
<td>0 2</td>
<td></td>
</tr>
</tbody>
</table>

Here are two more input/output pairs on the left and right:

<table>
<thead>
<tr>
<th>Input</th>
<th>Ear Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5</td>
<td>2</td>
</tr>
<tr>
<td>0 1</td>
<td>3 0 3 3 1 1 0</td>
</tr>
<tr>
<td>1 2</td>
<td>2 1 2 2 3</td>
</tr>
<tr>
<td>2 3</td>
<td></td>
</tr>
<tr>
<td>3 1</td>
<td></td>
</tr>
<tr>
<td>3 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Ear Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 17</td>
<td>7</td>
</tr>
<tr>
<td>0 1</td>
<td>8 0 8 8 10 10 7 2 2 6 6 3 3 1 1 0</td>
</tr>
<tr>
<td>1 2</td>
<td>1 7 8</td>
</tr>
<tr>
<td>2 3</td>
<td>2 7 9 9 8</td>
</tr>
<tr>
<td>3 4</td>
<td>1 3 2</td>
</tr>
<tr>
<td>4 5</td>
<td>1 1 2</td>
</tr>
<tr>
<td>3 6</td>
<td>3 3 5 5 4 4 2</td>
</tr>
<tr>
<td>2 7</td>
<td>1 4 3</td>
</tr>
<tr>
<td>7 8</td>
<td></td>
</tr>
<tr>
<td>8 9</td>
<td></td>
</tr>
<tr>
<td>8 10</td>
<td></td>
</tr>
<tr>
<td>5 3</td>
<td></td>
</tr>
<tr>
<td>4 2</td>
<td></td>
</tr>
<tr>
<td>3 1</td>
<td></td>
</tr>
<tr>
<td>6 2</td>
<td></td>
</tr>
<tr>
<td>9 7</td>
<td></td>
</tr>
<tr>
<td>10 7</td>
<td></td>
</tr>
<tr>
<td>8 0</td>
<td></td>
</tr>
</tbody>
</table>
Solution:

Perform a DFS on the graph, creating a DFS tree. The choice of root does not matter. Each node $u$ has a depth $d(u)$ in the DFS tree, where nodes farther from the root have higher depth. Define the depth of a back edge $e = (u,v)$ as $d(u)$, where $u$ is the upper endpoint of $e$. From this point on every time I mention a back edge $(u,v)$, $u$ is the upper vertex.

The algorithm is to process all back edges in the graph in increasing order of depth. If two back edges have the same depth, it does not matter which one is processed first. For each back edge, add its induced cycle, minus all edges that have already been included in some ear, as a new ear in the ear decomposition.

It will turn out that for each induced cycle, the edges on it that have already been included in some ear form a contiguous section of the cycle. This will guarantee that each iteration adds a valid path.

So why does this work?

Note that the first back edge processed will create the starting cycle, as no edges in the graph have yet been used.

Otherwise, suppose we are processing a back edge $e = (u,v)$. We must show now that at least one edge in the induced cycle of $e$ has already been added in the ear decomposition.

Let $c$ be the child of $u$ on the path from $u$ to $v$. $(u,c)$ is an edge in the induced cycle of $e$. Assume for the sake of contradiction that $(u,c)$ is not yet part of a ear. All back edges with depth strictly less than $d(u)$ have already been processed. Thus there is no back edge in the subtree rooted at $c$ with depth less than $d(u)$, so $u$ is an articulation point, a contradiction! Thus $(u,c)$ have already been added as part of a previous ear.

So how can we find all the edges that are part of the ear including $e$? Since $(u,c)$ is already part of a ear, we can simply walk from $v$ towards $u$, adding all edges seen, stopping when we first see an edge that is part of a previous ear. It is obvious at this point that if we performed this walk for all previously processed back edges, all edges in the induced cycle of $e$ that are already part of a ear form a contiguous segment. Thus we can stop the walk as soon as we see an edge that is already used.

One final detail is that we don’t want to naively sort our back edges, as then our algorithm will no longer be linear time. But since all depths are integers in the range $[0, N]$, we can use counting sort instead. In other words, we can add back edges to “buckets” based on the depth of their upper vertex, then loop through all buckets in increasing order.
Code:

```cpp
//Andy Yang
#include <bits/stdc++.h>
using namespace std;

vector<int> edges[1000001]; // adj list
vector<pair<int, int>> b[1000001]; // buckets for back edges
bool used[1000001]; // used[i]: whether the edge (i, p[i]) is assigned to a ear
bool visited[1000001]; // whether a node has been visited in the DFS
int depth[1000001]; // depth of each node
int p[1000001]; // parent of each node
vector<int> ans[1000001]; // final answer: array of ears

/*
Finds and orders all back edges,
and calculates information about each node in the DFS tree.
*/
void dfs(int u, int d) { // u = current node, d = depth(u)
    visited[u] = true;
    depth[u] = d;
    for (int v : edges[u]) { // loop over neighbors
        if (v == p[u]) continue; // ignore edge to parent
        if (visited[v] && depth[v] < depth[u]) // back edge
            b[depth[v]].push_back({ v, u });
        else if (!visited[v]) { // tree edge
            p[v] = u;
            dfs(v, d + 1);
        }
    }
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; ++i) {
        int u, v;
        scanf("%d%d", &u, &v);
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    p[0] = -1;
    dfs(0, 0);
    for (int d = 0; d < n; d++) {
        for (pair<int, int> e : b[d]) { // process a back edge
            vector<int> ear;
        }
    }
}
```
int v = e.first;
int u = e.second;
ear.push_back(v);
ear.push_back(u);
while (u != v && !used[u]) { // walk upwards from v
    used[u] = true;
    u = p[u];
ear.push_back(u);
}
}

// output
int earcnt = m - n + 1;
printf("%d
", earcnt);
for(int i = 0; i < earcnt; i++) {
    printf("%d ", ans[i].size() - 1);
    for (int j = 0; j < ans[i].size() - 1; j++) {
        printf("%d %d ", ans[i][j], ans[i][j + 1]);
    }
    printf("\n");
}