PROBLEM SET 5

Zhiqiang Xie 77892769

Problem 1: Save the Bridges

1. **Python**

```python
def MinCost(G, plans):
    '''
    G = (V, E), where V is the set of all houses and E is the set of all bridges.
    '''
    # Since the GatesTown is a connected city, we can always find the MST
    minTree = MST(G)
    max_stone = max([e.weight for e in minTree.edges])
    min_cost = float("Inf")
    best_plan = -1
    for i in range(k):
        power, cost = plans[i]
        if power >= max_stone and cost < min_cost:
            min_cost, best_plan = cost, i
    # No plan will keep the entire city connected if best_plan == -1
    return best_plan
```

2. Firstly, let's do some preprocessing on the information.

   Sort the plans by their cost c_i, and then we delete all plans with higher cost but less power by traversing them all. Now we get a new plan list in sorted (both cost and power) order:

   $C_{new} = \{cn_1, cn_2, \ldots, cn_g\}, P_{new} = \{pn_1, pn_2, \ldots, pn_g\}$,

   where cn is the cost and pn is the power, g is the number of plans now with $g \le k$.

   Sort the bridges by their number of stones $s_{u,v}$, and then we get a sorted array $S = s_1, s_2, \ldots, s_m$, where m is the number of bridges.

   This preprocessing costs $O(m \log m + k \log k)$.

   Python

```
DUS = DisjointUnionSet(V) # V is the set of all houses
plan_results = [0]*g # g is the number of plans
j = 0
for i in range(g):
    # S is the sorted array of bridges, P is the sorted array of powers
    while S[j] <= P[i]:
        DUS.union(S[j].u, S[j].v) # u,v are the endpoints
        j += 1
    # Here I assume this copy operation costs constant time
    # Though it's actually not, and it's more a technique issue
    plan_results[i] = DUS.deepcopy()

for c in costs_max: # cost_max is the array of budget
    target = C.BinarySearch(c) # C is the sorted array of costs of plans
    result = plan_results[target]
    connected = result.countSet(u)
    # count the number of elements in the corresponding set
```

The time complexity of process above should be:

$T(n) = O(n) + O(g) + O(m) * O(\log n) + O(n) * O(\log q) \in O(m \log n + n \log q)$

Here we have $m \geq n, k \geq q$, and $k \geq n$ or $k < n$. Thus $O(n \log q) \in O(k \log k)$ or $O(n \log q) \in O(m \log m)$.

Therefore, we have the total time complexity $T(n) \in O(m \log m + k \log k)$

Problem 2: Parity Problems

1. Start from an arbitrary node v_0 in the graph and run DFS, set $h(v_0) = 0$ and then for any new node we visited, assign $h(v_j) = \neg h(v_i)$ if $w(v_i, v_j)$ is even, assign $h(v_j) = h(v_i)$ if $w(v_i, v_j)$ is odd.
2. I'd like to transform this problem to 2-coloring or bipartite graph problem:
   - Merge two nodes connected by an odd weighted edge to be a super-node iteratively. Finally we'll have a graph with add edges to be even-weighted.
   - Now our task is actually to color the nodes in two colors to make any two adjecent nodes have different color, the color here is corresponding to the $h(v_i)$.
   - Equivalently, now we want to prove whether this graph is a bipartite graph.
   - And we already know that a graph is bipartite if and only if it has no odd length cycles. Here odd length cycles are equivalent to cycles with an odd number of even-weight edges in this problem.
3. Since we have the function h exists if the undirected graph G doesn't contain any cycles with an odd number of even-weight edges.
   - We can firstly run the edge reweighting $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ to get a new graph with all edges in odd weights. This step costs $O(|E|)$ time.
   - Then apply the black magic algorithm to get the all-pairs shortest paths in $O(|V|^2)$ time.
   - Since the edge reweighting doesn't change the shortest paths, the total time complexity

     $T(n) = O(|E|) + O(|V|^2) \in O(|V|^2)$

Problem 3: Largest Weight Cycle

1. Brute-force:
   - Try all pairs of nodes (u, v) and then calculate the weights of the cycles they formed.
   - Pick the largest one to be the answer.
   - $T(n) = n^2 * O(n) \in O(n^3)$
2. & 3.

   A direct bottom-top strategy:

   - Set four attributes for every nodes $(l_1(v_i), l_2(v_i), p_1(v_i), p_2(v_i))$, initialize them to be zero.
   - Pick a leaf node v and then select the edge (v, u).

     $l = l_1(v) + w(v, u)$

     - If $l > l_1(u)$, $l_1(u), l_2(u) = l, l_1(u)$ and $p_1(u), p_2(u) = v, p_1(u)$
     - If $l > l_2(u)$, $l_2(u) = l$ and $p_2(u) = v$
   - Remove the node v and the edge (v, u), the graph remains to be a tree.
   - Repeat picking leaf nodes until all edges are removed.

   Finally we have a list of nodes with their $(l_1(v_i), l_2(v_i), p_1(v_i), p_2(v_i))$, the final largest weight $w = max([l_1(v_i), l_2(v_i)])$

   And once we found the target node, we know the node is on the longest path of the weighted tree, which leads the largest weight cycle.

   Besides, we could trace back from the $p_1(v), p_2(v)$ to the two endpoints of the longest path, where we add a zero-weighted edge between them to form the largest weight cycle.

   Further more, if all the weights of the edges of the tree are negative, the procedrue outputs a sigle node which leads a self-loop. We should treat this case specially if the self-loop is illegal:

   - Traverse all the edges and pick the one with largest weight.

   The overall time complexity depends on the implementation of picking leaf nodes:

   - Search from root to leaf every time, which leads

     $T(n) = O(n) + n * O(n) + O(n) \in O(n^2)$
   - Search following u to a leaf node, which means that we visit a node k times, where k is the number of it's child. Since every nodes except root have a parent, thus we have

     $T(n) = O(n) + O(2n) + O(n) + O(n) \in O(n)$