CS 430 – Fall 2024
Introduction To Algorithms
Homework #6

1a) When the table is not full, the cost of insertion is 1.

When the table is full, we need to expand the table. The cost of this expansion involves:

Allocating a new table of size 3 times the current table size.

Copying all elements from the old table to the new one.

Inserting the new element.

Analyzing the Cost of the ith Insertion:

If the current size of the table is s, and it is full when we attempt to insert the ith element, the cost of expansion will include copying all s elements from the old table to the new one. Therefore, the cost is s + 1 (where s accounts for copying elements, and 1 is for inserting the new element).

If the table is not full, the cost of the ith insertion is simply 1.

1b) We assign an amortized cost to each insertion operation, which is greater than or equal to the actual cost. Any excess is saved to pay for future operations (i.e., when the table needs to be expanded).

We first assign an amortized cost of 3 units to every insertion.

When an insertion occurs and the table does not need to be resized, the actual cost is 1 unit. The extra 2 units of amortized cost are saved as credits.

When the table is resized, the credits accumulated from previous operations are used to pay for the cost of copying elements into the new table.

Suppose the table has a size s and becomes full. To insert the next element, we need to allocate a new table of size 3s and copy all s elements from the old table to the new one.

The cost of copying s elements is s units. However, each of these s elements contributed 2 units of credit when they were inserted, totaling 2s units of credit. These credits saved from each insertion are sufficient to cover the cost of copying when a resize occurs.

Thus, the amortized cost per insertion is O(1).

2) The assumption that the cost of flipping bits might remain logarithmic does not hold in this case. In fact, a sequence of operations could be constructed that causes every bit in the k-bit counter to flip frequently. Specifically, if we have a series of events consisting of incrementing the counter when it is at all 1s and then decrementing when the counter is at all 0s, this would result in a costly pattern. Each of these operations would require flipping all k bits, and if there are n such operations, the total cost would be $\Theta(nk)$.

3)

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]
        self.rank = [1] * (n + 1)
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1

def canAssignDorms(n, same_requests, diff_requests):
    uf = UnionFind(n)
    # same dormitory requests
    for s, t in same_requests:
        uf.union(s, t)
    # different dormitory requests
    for u, v in diff_requests:
        if uf.find(u) == uf.find(v):
            return "Not Possible"
    return "Possible"
```

Explanation:
We use the Union-Find data structure to represent the connected components (groups of students who must be assigned to the same dorm).
Initialize an array parent to manage the Union-Find operations
Process Same Dormitory Requests:

For each request (si,ti), perform a union(si, ti) operation. This ensures that students si and ti are in the same dorm.

Check Different Dormitory Requests:

For each request (ui,vi), check if ui and vi are in the same connected component using the find operation:

If find(ui) == find(vi), it is impossible to satisfy the constraint of assigning ui and vi to different dorms, so return "Not Possible".

If find(ui) != find(vi), then it is possible.

4) In any simple, undirected graph with n ≥ 2 vertices, there exist at least two vertices that have the same degree.

Proof:
Consider a graph G = (V, E) where ∣V∣ = n ≥ 2. In this graph:
Since the graph is undirected and simple:

The degree of each vertex v must be an integer between 0 and n − 1.

A person cannot be their own friend (no self-loops), and the relationships are bidirectional (friendship is mutual).

There are n vertices, and each vertex's degree is an integer between 0 and n − 1. This gives nn possible degree values in the range [0, n − 1].
However, not all of these degree values can occur simultaneously:

If one vertex has degree 0, it implies that there cannot be a vertex with degree n − 1, because the vertex with degree n − 1 would need to be connected to every other vertex, including the vertex with degree 0, which contradicts the definition.

Thus, there are only n − 1 distinct degree values that can be valid. By the pigeonhole principle, since we have n vertices but only n − 1 possible degrees, at least two vertices must share the same degree.

5) We can represent the problem of checking the internal consistency of the oral history data using a directed graph.
Vertices:

- Each person Pi is represented as a vertex in the graph.

Edges:

- For each "Pi died before Pj was born," create a directed edge from Pi to Pj.
- For each "Pi and Pj were both alive at the same time," introduce a constraint that Pi and Pj must have overlapping lifespans. This translates to the requirement that there should be no directed path from Pi to Pj or from Pj to Pi.

Algorithm:
Constructing the Graph:

Initialize an empty directed graph G with n vertices, one for each person Pi.
For each "Pi died before Pj was born" add a directed edge from Pi to Pj.

Detecting Cycles in the Graph:

Use a cycle detection algorithm, like depth-first search to check if there is a cycle in the directed graph G.
If there is a cycle, then the facts are inconsistent. Output the subset of facts that form this cycle as evidence of inconsistency.
If there is no cycle, the facts related to the "died before" relationships are consistent.

Checking Overlapping Lifespan Constraints:

For each pair (Pi, Pj) that they were both alive at the same time, check:

If there is a directed path from Pi to Pj or from Pj to Pi, then the constraint that they were alive simultaneously is violated, and the facts are inconsistent.
If no such directed path exists, the overlap constraint is satisfied.

If any inconsistency is found (either a cycle in the graph or a violation of the overlap constraint), output "Inconsistent" along with the subset of facts causing the inconsistency.
If no inconsistencies are found, output "Consistent."

7) FIND-SET is called 2*|E| times
UNION is called |V| - K times