Q11 [5 points] In HW23, you were asked to implement a data structure using the disjoint-set API to maintain an array of bits X[1 ... n] along with these operations.
(i) Init() : Initialization. All bits of X are initialized to 0.
(ii) Set(i) : Set X[i]=1. It is guaranteed that i <= n-1 so the rightmost bit of X[] is never set.
(iii) IsSet(i): return X[i]
(iv) NextUnset(i): return the next largest smallest j >= i s.t. X[j]=0. This will always return some index.
Which disjoint-set implementation will you choose, and why, if you require that Set() has ammortized complexity O(log n) and NextUnset() has worst-case complexity O(1)? Explain.
Ans: Set() is implemented using 1 Union() and NextUnset() is implemented using 1 Find().
We should use the shallow-tree+threading+union_by_rank implementation. The ammortized cost of Union() in this implementation is O(log n) and the cost of Find() is O(1).
Path-compression+union-by-rank is not suitable here since its Find() has O(log n) worst-case complexity.

Q12. [5 points] Suppose I am trying to find the shortest distances from a source vertex s on a graph G using the Bellman-Ford algorithm. While filling the memo OPT[i,v], where i is an integer and v is a vertex, in the increasing order of i (starting at i=0 and filling the table row-wise), I observe that for one particular k, OPT[k-1,u] = OPT[k,u] for all vertices u.
Suppose dist(s,u) denotes the shortest distance between s and u. Which of the following is true and why?
(a) OPT[k,u] < dist(s,u) (b) OPT[k,u] <= dist(s,u) (c) OPT[k,u] = dist(s,u)
(d) OPT[k,u] >= dist(s,u) (e) OPT[k,u] > dist(s,u) (f) None (answer depends on G)
Ans: The basic idea is the recurrence formula for OPT[j,.] depends only on OPT[j-1,.]:
OPT[j,u] = max_v { OPT[j-1,u], OPT[j-1,v] + w(v,u) }
So, the values for any j (i.e., the j-th row in the memo storing the OPT values) can be filled entirely from the values for j-1 (the (j-1)th row). Since the k-1 and k-th rows are identical, this would mean the (k+1)-th row would be the same as them, and so will be all the next rows, including the (n-1)-th row. The (n-1)th row stores the distances dist(s,u). So dist(s,u)=OPT[n-1,u]=OPT[k,u].

Problem 3

Let TT be a free tree with positive weights assigned to its vertices. A vertex VV is called a **center** of TT if it satisfies the following condition:

- If VV and its adjoining edges are removed, no connected component of the remaining tree has a total weight greater than half the total weight of TT.

For example, consider a tree with a total weight of 31:

- Vertex D is a center because, upon removing DD, the remaining components have weights {A,B,C}=9{A,B,C}=9, {E}=9{E}=9, and {F,G,H,I,J}=11{, none of which exceeds half the total weight (15.5).
- Vertex F is not a center because removing FF leaves a component {A,B,C,D,E} with a total weight of 20, which is greater than half of 31.

Let T be a tree with positive weights on the vertices. A vertex V is a center of T
if the following holds: If you delete
Vand its adjoining edges, then none of the remaining connected pieces has more
than half the total weight of
T
. For instance, the tree below has a total cost of 31. Vertex D is a
center, because if you delete D the remaining pieces are
{
A,B,C
}
with cost 9;
{
E
}
with cost 9;
and
{
F,G,H,I,J
}
with cost 11. Vertex F is not a center, because the piece
{
A,B,C,D,E
}
has cost
20, which is more than half of 31.
Give an
O
(
N
) algorithm to find the center of a free tree. Hint: begin by converting the tree to
a rooted tree, with an arbitrarily chosen root.
Answer:
Let R be a vertex in T. Let T’ be the rooted tree corresponding to T with root R.
Note that if you delete node N from T, the connected components are the subtrees rooted at children of N and the collection of all nodes not in the subtree rooted at N. We will call this the ”outer” component.

```
Using DFS, compute for every node N in T’ the total cost of the subtree
rooted at N.
```

```
N = R;
Total = R.cost;
while (true) {
if (there exists a child C of N such that C.cost >= Total/2)
N = C;
else return N;
}
}
```

It is clear that if T has a center, then the subtree of the child C that is chosen must contain the center, and that the loop will stop if N is the center. Therefore, if T has a center, then the algorithm returns that center. It is possible to prove that every tree has a center

Q9. [5 points] We call a graph ``compact' if there is a path with at most 2 edges between any two vertices in the same connected component.
def reduce(G): // G is any unweighted undirected graph
G' = copy of G
Add a vertex v to G'
Add edges from v to every vertex
Add another vertex u and add edge (u,v)
Return G'

Lemma: G has a Hamiltonian path iff reduce(G) is compact and has a
Hamiltonian path.
Fill in the blanks above such that reduce() is a polynomial-time algorithm and it satisfies the lemma below. Explain how both these criteria are satisfied.
Ans:
Reduce() adds 2 vertices and n+1 edges to a n-node graph. Hence, it takes polynomial time.
G' is compact since (1) u can reach v with 1 edge and reach every other vertex w within 2 edges, u to v and v to w, (2) v can reach every vertex with 1 edge, (3) all other vertices can reach any other vertex within 2 edges by going through v.
If G has a Hamilton path p then G' has this Hamiltonian path: u - v - p
If G' has a Hamiltonian path p then p must start at u (since u has degree 1), go to v, then visit all the other vertices. The subpath after v is then a Hamiltonian path of G.
Q10. [5 points] Complete the following divide and conquer algorithm to find all elements in an n-sized integer array that appear n/5 or more number of times. Note that there can be at most 5 such elements. Explain your approach and discuss the time complexity of your algorithm.
def FindFrequent(A[1...n]): // returns at most 5 numbers
AL = A[1... n/2]
AR = A[n/2+1 ... n]
(u1,u2,u3,u4,u5) = FindFrequent(AL) // some of these could be Null
(v1,v2,v3,v4,v5) = FindFrequent(AR) // some of these could be Null
i=1, w1=w2=w3=w4=s5=NULL
for each element x in {u1,u2,u3,u4,u5,v1,v2,v3,v4,v5}:
t = number of copies of x in A

If t >= n/5: wi = x, i=i+1 // x is frequent element
return (w1,w2,w3,w4,w5) // some of these could be Null

Explanation and analysis:
Recurrence T(n) = 2T(n/2) + O(n) // the loop makes a linear pass and the loop is over at most 10 elements. Solution of recurrence : O(n log n)
If x is a frequent element, then frequency of x in A >= n/5. However, if freq(x in AL) < (n/2)/5 and freq(x in
AR) < (n/2)/5 then freq(x in AL+AR) < n/5. So, x should appear at least 20% of the time in at least one of AL or AR. The algorithm identifies all elements that appear at least 20% of the time in both AL and AR
and checks which of them appear at least 20% of the time in A by calculating their frequency making
linear passes over the array.