## Problem 1

Given a graph G on $n$ nodes and $m$ edges, where $x$ out of $m$ edges belong to set X and $m$-$x$ edges belong to set Y, construct a spanning tree T with exactly $k$ edges such that each edge $e \in X$ or correctly report that no such tree exists, in polynomial time (or better).

### Solution

We can trivially eliminate the existence of T if the set X has cardinality less than k. Otherwise, we try to construct a spanning tree using as many X-edges as possible. We can do this by prioritizing X-edges above Y-edges (e.g., weighting the edges in X so that each edge $x \in X > y \in Y$. Then we can construct a maximum weight spanning tree, $T_{max}$, by selecting the maximum weight edges first (modified Kruskal's algorithm).

If $T_{max}$ has $j < k$ X-edges, then it's clear that no spanning tree with at least $k$ X-edges exists (since we used as many X-edges as possible to construct $T_{max}$). Else, $T_{max}$ has exactly $k$ edges (and we're done) OR $j > k$ edges. In the latter case, we can attempt to construct a minimum weight spanning tree, $T_{min}$, using Kruskal's algorithm.

If $T_{min}$ has $i > k$ X-edges, then we can conclude that no tree with exactly $k$ edges exists, since $T_{min}$ was constructed using as few X-edges as possible. Else, $T_{min}$ has exactly $k$ X-edges (and we're done), or $i < k$ X-edges. In the latter case, we swap Y-edges in $T_{min}$ with X-edges in $T_{max}$ until a spanning tree with the desired number of X-edges is produced.

**Claim**: The existence of $T_{max}$ with $j > k > i$ X-edges guarantees that we can swap at least $k$-$i$ edges in $T_{min}$ for X-edges in $T_{max}$ (proof below).

### Proof

We prove the claim that we can swap $k$-$i$ edges between $T_{min}$ and $T_{max}$ by induction on i, the number of X-edges in $T_{min}$.

**Base case (i = 1)**:
We add an X-edge in $T_{max}$, $e_1$, that is NOT also in $T_{min}$ to $T_{min}$, so that a cycle is created. Since $T_{min}$ only has one X-edge, and $e_1 \notin T_{min}$, we've either created a cycle with two X-edges, or a cycle with a Y-edge and an X-edge. Remove the old edge so that $T_{min}$ is now a spanning tree that has edge $e_1$ in common with $T_{max}$.
Now, $T_{min}$ either has two X-edges (removed a Y-edge) or one X-edge (removed an X-edge not in $T_{max}$). If $T_{min}$ has a single X-edge $e_1$, add a second X-edge $e_2 \in T_{max}$ such that $e_2 \notin T_{min}$, again creating a cycle. Now, the old edge must be a Y-edge, and removing it results in a spanning tree with two X-edges, both of which are shared with $T_{max}$. Assume that we do this until $T_{min}$ has $i$=$k$-$1$ X-edges, all shared with $T_{max}$.

**Case (i = k)**:
Since $T_{min}$ has $i < j$ X-edges and $n$-$k$-$2$ Y-edges, we can perform additional swaps. We add an X-edge in $T_{max}$ to nodes incident on some Y-edge in $T_{min}$, such that a cycle is created. Then, we erase the Y-edge so that $T_{min}$ is a spanning tree with $k$ X-edges.

### Time Complexity

Adding weights to the X edges takes $\mathcal{O}(x)$ steps and constructing the two trees using Kruskal's algorithm takes an additional $\mathcal{O}(mlogn)$ steps. Performing at most $k$+$1$ swaps takes an additional $\mathcal{O}(k)$ steps, so the total running time of the algorithm is $\mathcal{O}(mlogn)$, as desired.

## Problem 2

The exercise is similar to Solved Problem 1, and basically asks us to perform a binary search on a set S of 2n values that have been partitioned into 2 disjoint sets $S_1, S_2$, each with cardinality n, until we find the median ($n^{th}$) element in the original set of 2n values.

### Solution

We assume that $S_1, S_2$ are sets, and that $S_1 \cap S_2 = \emptyset$. Then, we just perform a modified binary search on $S_1, S_2$ by querying each database with $i = \frac{n}{2}$, which returns the $\frac{n}{2}^{th}$ smallest element in each set. The smaller of the two values is smaller than the top $\frac{n}{2}$ values in its set, and smaller than the top $\frac{n}{2} + 1$ values in the other set. Thus, it's smaller than the top $n + 1$ elements in $S_1 \cup S_2$. Similarly, the larger of the two values is larger than the bottom $\frac{n}{2} - 1$ values in its set and larger than the bottom $\frac{n}{2}$ in the other set, making it larger than the than the bottom $n - 1$ elements in $S_1 \cup S_2$. Since we want the median value ($n - 1 < n < n + 1$), we can disregard the top half of the set containing the larger value (which are all greater than the $n^{th}$ element) and the bottom half of the set containing the smaller value (which are all smaller than $n^{th}$ element). We query our databases again, this time with $i_1 = \frac{n}{2} + \frac{i}{2}$ for the database that returned the smaller value in the last step, and $i_2 = \frac{i}{2}$ for the other database. We do this recursively, each time disregarding the half of each subset of $S_1, S_2$ that doesn't contain the median value of the two subsets.

### Algorithm

```
// Assume that we take n, $mid_1$, and $mid_2$ as inputs and work on two sets, $S_1, S_2$.
// Assume query() is the function used to query the databases.
// Begin with n, n/2, n/2.
If n = 1, return min(query($S_1$, $mid_1$), query($S_2$, $mid_2$)).
If query($S_1$, $mid_1$) ¡ query($S_2$, $mid_2$),
    Recursive call with (n/2, n/2 + $mid_1$/2, $mid_2$/2).
Else,
    Recursive call with (n/2, $mid_1$/2, n/2 + $mid_2$/2).
```

### Time Complexity

The algorithm, being a binary search, requires at most $\mathcal{O}(logn)$ queries.

## Problem 3

The exercise asks us to count the number of *significant inversions* in a sequence, defined as $a_i > 2a_j$ for some pair of values in the sequence with $i < j$.

### Solution

This is just a variation of merge sort. We use the same merge sort algorithms described in the book on page 224-225.

### Algorithm

```
// Modified Sort-and-Count from page 225 of [KT].
If S has only one element,
    return.
Else,
    Divide the list into two halves:
        A contains the first n/2 elements,
        B contains the remaining n/2 elements.
    ($r_A$, A) = Sort-and-Count(A),
    ($r_B$, B) = Sort-and-Count(B),
    S = Merge(A,B).
    B' = 2B = 2b1, 2b2, .... 2bn.
    (r, S') = Merge-and-Count(A,B').
return r = $r_A$ + $r_B$ + r, and the sorted sequence S.
```

### Proof and Time Complexity

The only thing that the algorithm above does differently from the one described in the textbook is perform an additional merge operation to produce a second subsequence that is then checked for 'significant' inversions, taking an additional $\mathcal{O}(k * n)$ steps. Thus, the running time of the algorithm remains $\mathcal{O}(n * logn)$. Since the proof of correctness for this algorithm is identical to that of the Sort-and-Count algorithm described in [KT] section 5.3, I've omitted it for the sake of brevity.

## Problem 4

The exercise asks us, given a list S of n bank cards, a set A of bank accounts, and an equivalence relation on $S \times A$, defined as $s_1\ s_2$ if $a_1 = a_2$, to determine whether or not there is an equivalence class with cardinality *greater* than $\frac{n}{2}$. The caveat is that bank cards can only be compared with one another via a black box "equivalence tester", and we can only invoke this black box at most $\mathcal{O}(n * logn)$ times.

### Solution and Proof

If at least $\frac{n}{2} + 1$ cards are equivalent, by the pigeon-hole principle, if we divide up n cards into $\frac{n}{2}$ pairs, at least one pair of cards must be equivalent or there must be an odd card that remains unpaired. So, we can simply partition the n cards into two element subsets of S, and compare the two elements in each subset. If there are no equivalent cards and no odd card, then there is no equivalence class with at least $\frac{n}{2} + 1$ cards. If there's an odd card, just check it against the remaining n-1 cards - if there are less than $\frac{n}{2}$ matches, then the algorithm is finished (in $\frac{n}{2} + n - 1$ equivalence checks), and returns false. If there are $k \leq \frac{n}{2}$ pairs of equivalent cards, we only need to check one card from each pair, thus making $\frac{k}{2} \leq \frac{n}{4}$ equivalence checks. The algorithm continues in this manner, stopping after having found $\frac{n}{2} + 1$ equivalent cards or when no equivalent pairs remain. At this point, if there is an odd card remaining, we check it against the other n-1 cards. If there are at least $\frac{n}{2}$ matches, the algorithm returns true. Else, it returns false.

### Time Complexity

The first round takes $\frac{n}{2}$ checks, and the second takes at most $\frac{n}{4}$, and so on, until the last n-1 checks (if an odd card remains). The total number of checks is equal to $\sum_{i=0}^{n/2} n(\frac{1}{2})^i$, which is limited by 2n as n approaches infinity. Thus, the time complexity is $\mathcal{O}(n)$.