1.

**Algorithm**:

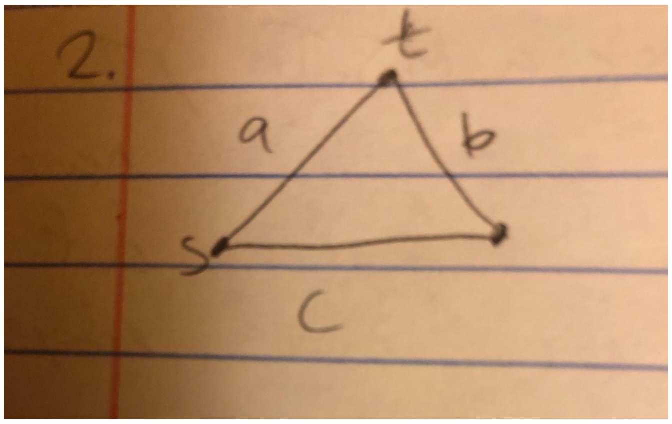
```
Initialize Cost (a, b) = Cost of edge (a, b)
T' = the Minimum spanning tree (MST) of G that has edge (u, v)
Delete the edge (u, v) from T'
Initialize X = M's component where u in X
Initialize Y = M's component where v in Y
Initialize Array[x] = 1 for all nodes x in X
Initialize Array[y] = 12 for all nodes y in Y

for each edge (c,d) in E:
    if Cost(c, d) < Cost(x, y) && Array[c] = Array[c]
        return true
    return false
```

**Correctness Justification**: We can first split the MST into 2 different components and then remove the edge (u, v) from the MST. Once we do this, we will have the minimum spanning tree be split into two different components (let's call them X and Y). Here, we know that u $\in$ X and v $\in$ Y, so we can do Breadth First Search on both of these to see the component that each node is supposed to be a part of. Next, we go through all edges of G with a loop and check edges that connect X and Y since these edges ONLY will create a spanning tree by connecting with the components. Once we find the one with the lowest weight, this is the one that is part of the new MST.

**Time Complexity Justification**: Since we are going through the edges three times total, this is going to be O(m). BFS is run 2 times in order to create the array that has each node's component and this accounts for the first couple traversals. Then, the last one is accounted for when we traverse through G's edges and try to look for the edges that are connecting 2 components.

2.

a. In order to disprove this statement, we need to find an example of an undirected graph with positive weight such that when we square the weight of each of its edges, we end up with the original shortest path P no longer as a shortest path from s to T.



If we have a graph (like up above) with 3 nodes and 3 edges such that a, b, c are edges and s, t are nodes, then we need a < b + c and a^2 > b^2 + c^2.

- An example of this would be if a = 4, b = 2, and c = 3. If we end up squaring everything, then a is now 16, b is 4, and c is 9. Now, the shortest path P is no longer by using the edge a to get from s to t. It is now starting at node s using edge c and then edge b because 9+4 = 13 and 12 < 16. Therefore, **we disprove this and this is false** since after we replace each edge weight with its square, the shortest path P is no longer the shortest path from s to t.

b.

**Proof by Contradiction:**

Supposed by contradiction that T' is a MST but when we square the edges of T, then T is no longer a MST. For reference, T and T' are the two MST's such that T' has the edge weights squared AND G and G' are the two graphs where G' has the edge weights square.

Let e be an edge in T, but not T'. If we removed that edge, we would have 2 subtrees. Now, we can group these nodes into 2 different components or subtrees. T' has some edge that connects these two groups and we can call this w' whereas this would be w in T.

Since w’ is in T’, but w is NOT, that implies that w’ ^ 2 < w^2. Since all the edge values are positive, that implies that w’ < w. But, this contradicts T being the MST of the original graph since the MST contains the set of edges connecting all the nodes with total minimal weight of the edges, then w has to be less than w’ in that graph. Therefore, when we square all the edges of T (which is already a MST) does not negate that it's a MST. **In conclusion, this is true and we have proven this**.

3.

**Algorithm**:

```
Initialize I to be the set of intervals
Sort I by starting time

searchForOverlap(Y): //Y is a set of intervals that is sorted
    if (the size of X == 1 OR 0)
        return 0

    Initialize L to be the first half of Y
    Initialize R to be the second half of Y

    Initialize L_l to be the interval in L with the latest finish time starting with L[0]
    for each interval i in L:
        if ( the end time of L_l < the end time of i)
            L_l = i

    Initialize overlap to be 0
    for each interval j in R
        if (end time of  L_l < the start time of j)
            break

        Initialize Interval to be min(v.endTime, Li.endTime) - v.startTime
        if overlap < interval
            overlap = overlappingInterval

    Initialize OverlapLeft to be searchForOverlap(L)
    let OverlapRight to be searchForOverlap(R)
    return maximum of (OverlapLeft, OverlapRight, overlap)

// call the function
findMaxOverlap(I)
```

**Correctness Justification**: We separate the intervals into half sets of two and then recursively call the function to find the greatest overlap in the half sets. When there exists an interval from a set that has an overlap with the other set, then this is going to be greater than the other two values where the recursive values return those. The greatest of these 3 total values will be what the maximum overlap is.

**Time Complexity Justification**: This is O(nlogn) since we first sort at the beginning by starting time and we can do this in O(n). Then we try to traverse through the half set before L_1. Since these will overlap, we can find which one has the largest oberalp with L_1 and this will be in linear time through the 2nd "half-set." This will be O(nlogn) too since we are using recursion and there will be a total of n elements and there will be nlogn calls of recursion.

4.

List of n bank cards:

- L = left half of list
- R = right half of list

**2 Functions:**

```
//takes 2 lists
Merge(A, B):
    //Case 1
    If (both lists of A and B have > 1 element)
        Print("false")
        Exit(1)

    //Case 2
    If (A and B both have a single element in them)
        If a[0] == b[0]]
            Return [A+B];

    //Case 3
    List = [[]]
    If (A has a single element)
        For b in B:
            If b[0] == A[0][0]
                List += b+A[0]
                B.remove(b)
                Break
        Let N = length(A[0]) + length(Flatten(B))
        If (List[0].size <= N/2)
            List.addAll(B)
        Return List

    If (B has a single element)
        For a in A:
            If a[0] == B[0][0]
                List += a+B[0]
                A.remove(a)
                Break
        Let N = length(B[0]) + length(Flatten(A))
        If (List[0].size <= N/2)
            List.addAll(A)
        Return List

Helper Function(List of Cards):
    //Base Case: 1 element
        Return [[element]]
    //Recursive Case:
        L = Helper(Left Half of List)
        R = Helper(Right Half of List)
        Return Merge(L, R)

Main(List of Cards):
    G = Helper(List of Cards)
    If length(G) == 1:
        print("false")
    Else:
        print("true")
```

**Time Complexity Justification**: The time complexity is O(nlogn) and this can be justified based on the two functions we have. The helper function gets the O(logn) runtime since we are recursively calling the function on one half of the list and then the other half of the list. Similar to binary search, we know that this would be log with a base of 2, but this really simplifies to just logn. Then, we can focus on the Merge function which is strictly linear and this creates the O(n)*O(logn) time complexity for a total of O(nlogn). The reason Merge is linear is because if we look closely at the operations in the function, the only operation that causes it to be linear is the for-loop. Other than that, we have constant operations within the for-loop and therefore, this part is just (n).