Name: Alimulla Shaik
Email: shal5122@colorado.edu
Algorithms Problem Set 2
Collaborators: Praveen D

1) a) Sol.
We are given X is any nonnegative random variable and c >0.
For any event E, from the definition of expectation, E(X) = $\sum_{a=0}^{\infty} P(a)\,X(a)$

E(X) = $\sum_{a=0}^{c} P(a)\,X(a) + \sum_{a=c}^{\infty} P(a)\,X(a)$

E(X) >= $\sum_{a=c}^{\infty} P(a)\,X(a)$

because X(a) > = c for all a $\varepsilon\, c$

So E(X) > = $\sum_{a=c}^{\infty} P(a)\,c$

E(X) = c. $\sum_{a=c}^{\infty} P(a)$

E(X) = c P(X>=c)

$\Rightarrow P(X >= c) = (1/c)(E[X])$

1) b) Sol.

Let's consider a random variable X with mean μ and variance σ^2.
We know that $(X - \mu)^2 >= 0$,
Use Markov's inequality, $P[(X - \mu)^2 >= a^2] = E[(X - \mu)^2] / a^2$
$P[(X - \mu)^2 >= a^2] = \sigma^2 / a^2$

$(X - \mu)^2 >= a^2$ only when $|X - \mu| >= a$,
$\Rightarrow P[|X - \mu| >= a] = \sigma^2 / a^2$

Use $a = c.\sigma$ in above equation.
$\Rightarrow P[|X - \mu| >= c.\sigma] <= 1 / c^2$

1) c) Sol.
Let E[X/X'] = Y then E[E[X/X']] = E[Y]
From the expectation and conditional expectation definition,

E[Y] = $\sum_{x'\varepsilon X'} P(x').Y$

= $\sum_{x'\varepsilon X'} P(x').\ \sum_{x\varepsilon X} x.P(x|x')$

= $\sum_{x\varepsilon X} x \sum_{x'\varepsilon X'} P(x|x')\,P(x')$

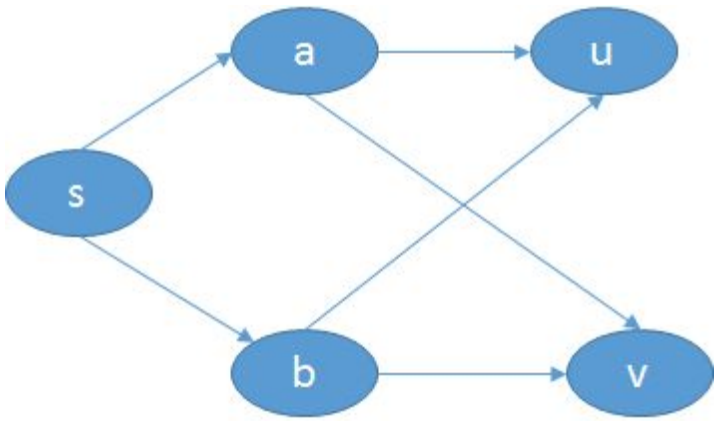
Using conditional probability properties,

E[Y] = $\sum_{x\varepsilon X} x \sum_{x'\varepsilon X'} P(x,x')$

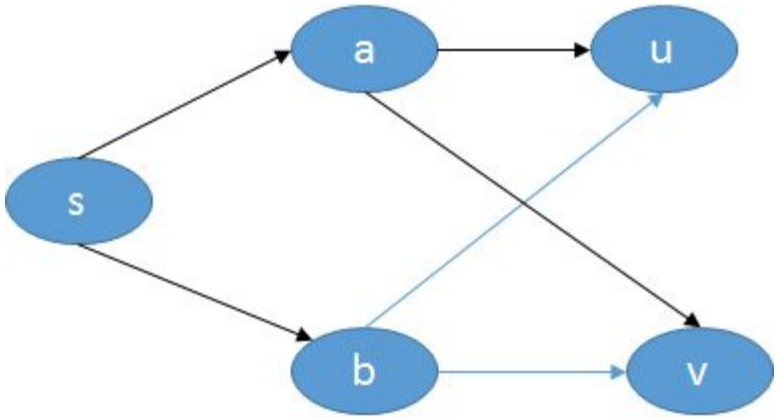
= $\sum_{x \varepsilon X} x\, P(x)$
= E[X]
Hence, E[E[X/X']] = E[X].

1) d) Sol.
We are given, a random variables $X_0, X_1 .. X_t$ such that $E[X_{t+1}|X_0...X_t] = X_t$ and $X_0 = 0$
$X_0 = 0 \Rightarrow E[X_0] = 0$
$E[X_{t+1}|X_0...X_t] = X_t$
$\Rightarrow E[X_{t+1} - X_t | X_0...X_t] = 0$ for all t>=0
$\Rightarrow E[X_{t+1}] = E[X_t]$ for all t>=0
For any martingale sequence no matter what are random variable values are, expected fortune will be equal to starting expected fortune.
So $\Rightarrow E[X_t] = E[X_0]$ for all t>=0
$\Rightarrow E[X_t] = 0$

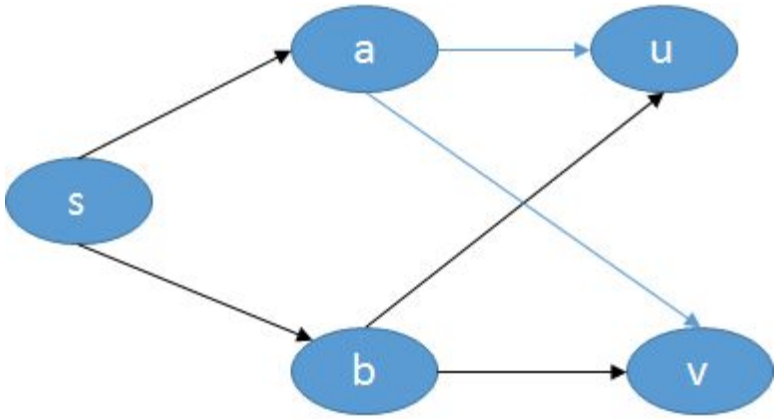
2) Sol.



Above example is the directed graph G = (V,E) with starting vertex 's'. As you can see, there exist unique shortest path from source vertex 's' to all nodes.
Let's apply BFS algorithm on above graph G.
In the initial step, we will visit the source vertex 's'. After that, we need to visit either 'a' or 'b' nodes. Here exists two possibilities and will apply BFS to both cases.
Case #1: Let's assume that 'a' comes before 'b' and visited first then (a,u) and (a,v) edges are visited & will be added to breadth first tree edges (shown below with black color edges). In next step, node 'b' is visited and all the vertices connected to it are already visited so it won't add any set of edges. So in this case, breadth first search did not include edges (b,u) and (b,v).



Case #2: Let's consider the other possibility of visiting 'b' node before 'a', in this case (b,u) and (b,v) edges are visited and will be added to breadth first tree edges (shown below with black color edges). In next step, node 'a' is visited and all the vertices connected to it are already visited so it won't add any set of edges. So in this case, breadth first search did not include edges (a,u) and (a,v).



As you can see, in both cases, BFS did not have {(a,u), (b,v)} and {(a,v), (b,u)} edges together in breadth first tree. Hence, no matter how the vertices are ordered in each adjacency list, all set of edges can not be produced by applying BFS on graph.

3) Sol.

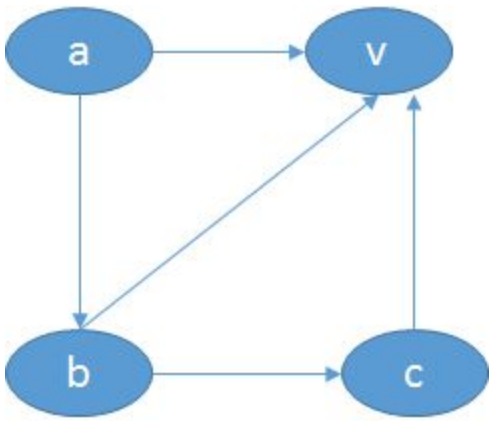
Algorithm to find lightest edge cut.
Input: Undirected weighted graph G = (V,E) with edge e ε E. Output: lightest edge cut or NO
Steps: 1. Consider all the edges E in the graph and sort in ascending order based on weight. 2. Consider safe edge set S and start with light weight edge first. 3. For each edge (a,b) in E, if find(a) $\neq$ find(b) then add (a,b) to S. 4. Union (a, b). 5. Continue step 3 for all edges in graph. 6. Set S has minimum spanning tree MST for the given graph.

7. Apply a cut starting from lightest edge in MST such that it is divided into 2 groups, S, S'.
8. I there exist any edge between these two groups S, S' which has light weight then return that cut, NO otherwise.

Correctness:
This algorithm determine lightweight edge across the cut in a given graph and returns it if it exists, otherwise returns NO. To find lightest edge, we first sort all the edges in the graph and starts with lightest weight edge to check if both endpoints of the edges lie in different components. Once, verification is done, add edge to safe edge set and continue the same process till you get minimum spanning tree. If the lightest edge across any cut exists then it will be present in MST. This algorithm has taken care of this property as well.

Running Time Analysis:
This algorithm has running time of kruskal's algorithm, i.e, = O(|E| log|V|).

4) Sol.
We are given adjacency matrix M(i,j) for a graph G. Which means, if there is an edge between i and j then M(i,j) = 1 else M(i,j) = 0. In the problem, universal sink is defined as the vertex which has no edges to other vertices and all other vertices have an edge to it.
Let's take an example to understand it. In this example vertex 'v' is universal sink. So it has no edge to all other vertices but all other vertices have edges to it.



Adjacency matrix M for the above graph is shown below.

	a	b	c	v
a	0	1	0	1
b	0	0	1	1
c	0	0	0	1
v	0	0	0	0

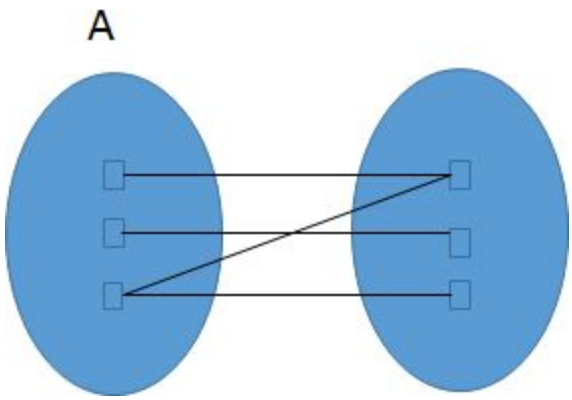
So, if a vertex is universal sink then row values of that vertex in matrix M are '0' and the column of that vertex in the matrix are 1 except for M(v,v).

Let's consider a case where M(i,j) = 0 and i ≠ j and it implies that vertex 'i' does not have an edge to 'j'. In this case, vertex 'j' can not be universal sink.
Take another case where M(i,j) =1, means vertex 'i' has an edge to 'j' and therefore vertex 'i' can not be universal sink.
As you can see, we can remove one vertex from the possible set of universal sink by just checking the value of each cell.

Algorithm to find universal sink in given directed graph.
Input: Adjacency Matrix 'M' of the graph G. Output: Universal sink set 'U'.
Steps: 1. Store all the vertices of the graph in the universal sink set U. 2. Consider two vertices i, j and check their values in adjacency matrix M. 3. If M(i,j) = 1 then remove 'i' from U. 4. Else remove 'j' from U (See above explanation for reason). 5. Continue step 2 till all vertices values are checked in pairs. 6. Only one vertex will be left at the end and check that vertex row and column values in M. 7. If the whole row values of that vertex are '0' and column values are '1' then that vertex is universal sink and return it. 8. Else universal sink does not exist and return NO.

Running Time Analysis:
We have considered (|V|-1) vertices and checked their values in adjacency matrix and left with one vertex. After that we have checked all row and column values of that vertex in matrix.
All these operations costs = (|V|-1) + (|V|) + (|V| -1) = 3|V|-2
Hence, this algorithm has running time = O(|V|).

5) a) Sol.
Bipartite Graph: A graph G = (V, E) is a bipartite graph if V can be partitioned into A and B such that all edges in E are between A and B.



Let's consider a Tree 'T' with root node 'r' and two vertex sets 'A' and 'B'. Keep the root node 'r' in vertex set A. In the tree, a path will exist from root to any vertex. If the path length is odd then keep that vertex in set A and if it is even then keep vertex in set B. From this, we have distributed all the vertexes in two sets based on the length from root to it. And we ensured each edge has one vertex in A and other in B.
We can conclude that, V is partitioned into A and B, and V= A U B such that all edges in E are between A and B.
Hence, From bipartite graph definition, Every tree is a bipartite graph.

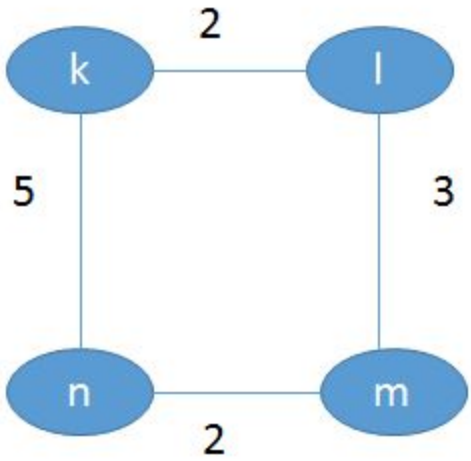
5) b) Sol.

Algorithm to find given graph is bipartite or not.
Input: Undirected graph G = (V, E) Output: G is bipartite or G is not bipartite
Steps: 1. Consider a undirected graph G = (V, E) and a Queue Q to store vertex information. 2. Start with any vertex 'v' and color it with green. Store or enqueue 'v' to queue Q. 3. Dequeue an element from Q and check all its neighbours color. 4. For all neighbours n of dequeued vertex, if n does not have any color then color it with opposite color of that vertex like yellow. And enqueue n to Q. 5. If n is colored and is the same color as dequeued vertex then halt the process and return G is not bipartite. 6. Otherwise continue step 3 till queue is empty. 7. Return G is bipartite.

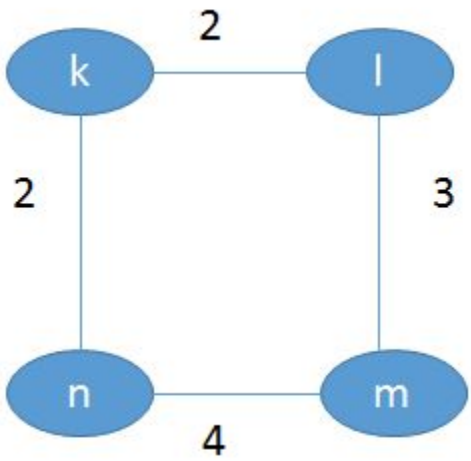
Correctness:
This algorithm starts with any vertex and there by checking all of its neighbours. If given graph is not bipartite then no matter how we partition the vertices, we will end up having both color vertexes in either of one group. Which contradicts bipartite property and this algorithm will find the same. If given graph is bipartite then this algorithm distribute all vertexes in two groups based on the color and each edge has one vertex in one group and another in other group. And thereby satisfies bipartite properties.

Running Time Analysis:
We have enqueue and dequeue operations in this algorithm and it costs O(|V|) time. We are checking all the neighbours colors of each vertex and it costs O(|E|). Hence total running time of this algorithm is O(|V| + |E|).

6) a) Sol.
Let's consider a graph with 4 vertices {k, l, m , n} and weights $W_{k, l} = 2$, $W_{l, m} = 3$, $W_{m, n} = 2$ and $W_{n, k} = 5$ and shown below. As you can see that graph has 2 equal weight edges and exist unique MST = {(k,l), (l,m), (m,n)}.

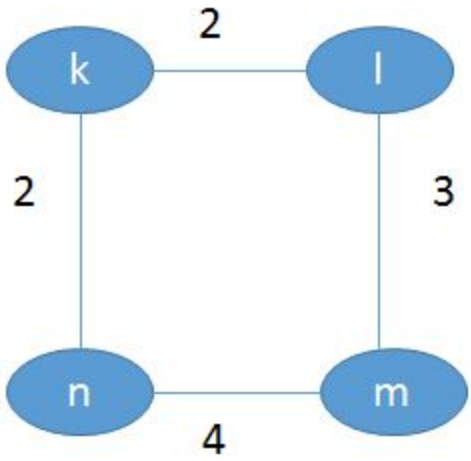


Algorithm to construct unique MST.
Input: Graph with (|V |) equal edge weights. Output: Unique MST
Steps: 1. Start with any vertex and mark it as source vertex ‘v’. 2. Edges connected to ‘v’ will have same minimum weight edge but edges not connected with ‘v’ will have more than minimum weight edge. 3. This will generate unique MST and example for the same is shown in below diagram.



6) b) i) Sol.

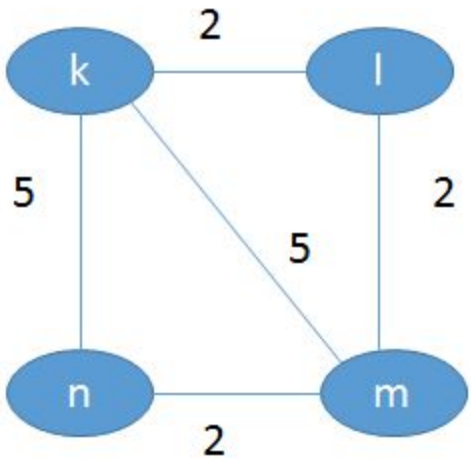
Case #1: Let’s consider a graph with 4 vertices {k, l, m , n} and weights $W_{k, l} = 2$, $W_{l, m} = 3$, $W_{m, n} = 4$ and $W_{n, k} = 2$.



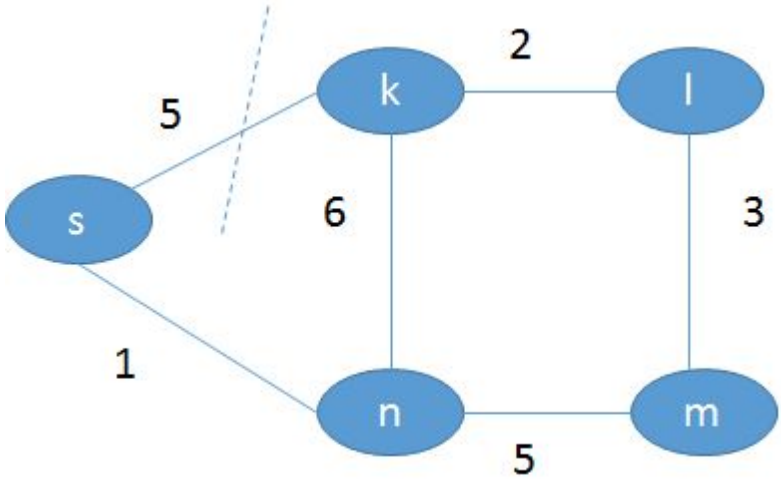
As you can see every cut has unique minimum weight edge but there exists two minimum spanning trees. MST_1 = {(k,l), (l,m), (m,n)} and MST_2 = {(k,n), (n,m), (m,l)}. This contradicts the assumption that if every cut has unique minimum weight edge then G has a unique MST.

Case #2: To prove the same, take unique MST and check if every cut has unique minimum weight edge or not. So let's consider a graph with 3 vertices {x, y, z } and weights $W_{x,y}$ = 1, $W_{y,z}$ = 1, $W_{z,x}$ = 2. Here there exists, minimum spanning tree with vertices {(x,y), (y,z)} but there is cut {(y), (z,x)}. As you can see this cut has no unique minimum weight edge but this case has unique minimum spanning tree. Which contradicts if G has unique MST then every cut has unique minimum weight edge.
Both cases contradicts assumption that graph has unique MST if and only if every cut has unique minimum weight edge.

6) b) ii) Sol.
Let's consider an graph with 4 vertices {k, l, m, n} which has a cycle. Graph has cycle {(k,l), (l,m), (m,k)} as shown in the below figure. As you can see there is no unique maximum weight edge. But there exist a unique MST = {(k,l), (l,m), (m,n)}. This contradicts our assumption that graph has unique MST if and only if the maximum-weight edge in any cycle of graph is unique.

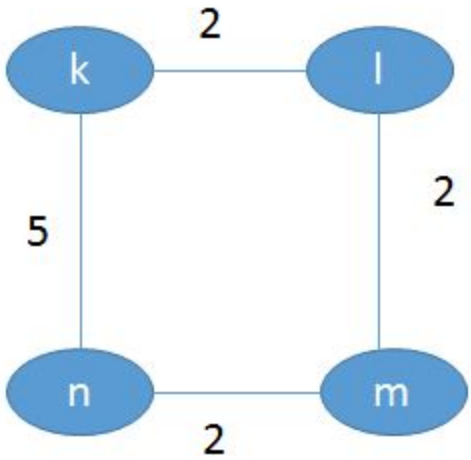


6) c) i) Sol.
Let's consider a graph with 4 vertices {s, k, l, m , n} and weights $W_{s, k}$ = 5, $W_{s, n}$ = 1, $W_{k, l}$ = 2, $W_{l, m}$ = 3, $W_{m, n}$ = 5 and $W_{n, k}$ = 6.

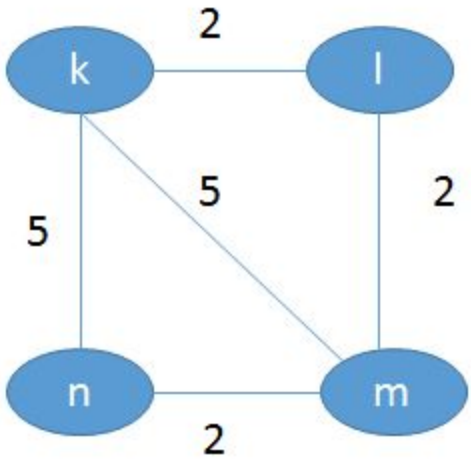


As you can see every cut has unique minimum weight edge. Consider a cut which removes an edge which belongs to MST. Here assume that we cut graph such that (s,k) edge which belongs to MST is broken. There exists unique minimum spanning trees. MST = {(s,n), (n,m), (m,l), (l,k)}. Hence we proved that G has a unique MST if and only if any cut has a unique minimum-weight edge crossing it.

6) c) ii) Sol.
Let's consider an graph with 4 vertices {k, l, m, n} as shown in the below figure. As you can see there is no unique maximum weight edge. But there exist a unique MST = {(k,l), (l,m), (m,n)}.



Let's add an edge e' (k, m) to construct a cycle in above graph. Now graph has cycle {(k,l), (l,m), (m,k)} as shown in below diagram. Still this graph has unique MST = {(k,l), (l,m), (m,n)} and newly added edge e' (k, m) is not included in MST. Hence we proved that maximum weight edge of any cycle constructed by adding one edge to MST is unique.



6) d) Sol.
We can determine an graph G has a unique MST using the below algorithm.

Algorithm to find G has unique MST or not.
Input: Graph G Output: YES if G has unique MST, otherwise NO
Steps: 1. Find the minimum spanning tree using Kruskal's algorithm and mark each safe edge with green color. 2. Now we have minimum spanning tree with safe edges are colored in green. 3. Run prim's algorithm on graph G again to find the minimum spanning tree and check each safe edge before adding it to minimum spanning tree. 4. If safe edge is colored with green then continue the process. 5. otherwise halt the process as we found minimum spanning tree different than what we got earlier and return NO. 6. Continue prim's algorithm till you get minimum spanning tree. 7. Return YES if we found the same colored safe edges using 2 algorithms which means graph has unique MST.

Correctness:
This algorithm uses two techniques to find minimum spanning tree. If given graph did not have unique MST then it will be detected while running prim's algorithm as we will get different colored safe edges in spanning tree. Halt the process here and our algorithm returns NO. If given graph has unique MST then both kruskal's and prim's algorithms will return same colored safe edges. Our algorithm returns YES in this case.

Running Time Analysis:
Kruskal's algorithm will run in O(|E| log|V|) time to find minimum spanning tree and to color the each safe edge in the graph. Prim's algorithm will run in same O(|E| log|V|) time to check safe edge is colored with same color or not.
Hence, this algorithm runs in total time = O(|E| log|V|) + O(|E| log|V|) = O(|E| log|V|)

7) Sol.

Algorithm to find minimized number of billboards.
Input: Intervals $[a_i, b_i]$ where $a_i < b_i$, for i = 1,2, .., n Output: Set of numbers S such that $\forall i \ \exists x \ \varepsilon \ S : a_i <= x <= b_i$ and |S| is minimized
Steps: 1. We are given intervals $[a_i, b_i]$. Sort all the values in a_i and b_i for all i = 1,2,3… n in ascending order and store it in two arrays A and B. 2. Consider i =0, j = 0 variables and Set of numbers S where |s| is minimized 3. For each value i in number of boundaries n, if i = n then return set S. 4. Else start with least value B[i] and add it to set S. 5. Check B[i] value against A array till you get B[j] < A[j] and increment j otherwise. 6. If j greater than number of jogger boundaries then return S. 7. Increment i and continue step 3 till i < n.

Correctness:
We can try to reduce number of billboards by selecting one common region shared by most of the joggers. To achieve this, we considered 2 arrays which contains start and end intervals respectively. Use any sorting technique to sort them and store it in 2 arrays. Now compare least end point value with array of starting points till you get end point less than start point. When there are no more endpoints to compare then we simply halt process and return set S. When there are no more start points that are less than end point values then it means region after this shared by all the joggers and who did not reach their destination point. So we will add a billboard at the least endpoint as this is shared by rest of the joggers. At the beginning, we added a billboard at the first end point as this is shared by most of the joggers which minimizes number of billboard count.

Running Time Analysis:
Use any sorting technique to sort two intervals points and we can achieve this using quicksort in O(n logn) time. Comparisons of two array values are achieved in O(n) time. Hence, total running time = O(n logn) + O(n) = O(n logn).

8) Sol.
We are given two intervals, $[a_i, b_i]$ and $[c_i, d_i]$ where $a_i < b_i < c_i < d_i$ for all i =1,2,..,n
In order to find billboards in this case, we need to check all the intervals for all joggers. As each jogger involves two levels of intervals and there are n joggers then it is really hard to determine common or shared region by all joggers. To satisfy the condition, that every jogger should see at least one billboard and reducing number of billboards, we need to consider all the possibilities. We can check joggers path and to find minimum number of possible billboards and it is achievable in polynomial time for a given intervals. This will ensure that problem is in NP (Nondeterministic Polynomial time).
There are few NP - complete problems available like vertex-cover, traveling salesman problem etc. For all the problems in NP is reducible to this problem.

From this, we can say that given problem is NP-Complete as it satisfies both conditions that problem should be in NP and every problem in NP is reducible or convertible to given problem.

9) Sol.

Algorithm for checking whether the d_v's are the correct distances or not.
Input: Strongly connected directed graph G = (V,E) with edge lengths $l_e > 0$ (for e ε E), node s ε V, numbers d_v (for v ε V) Output: YES or NO ($\forall$ v ε V where d_v is the shortest path distance from s to v)
Steps: 1. Let's consider an array 'D' to keep track of distances for all vertices and initialise all values to infinity. D[v] = ∞, $\forall$ v εV 2. Consider a Queue 'Q' and enqueue all vertices in the graph G. 3. Dequeue a vertex from Q and find lowest possible distance from source to it and store it in D array. 4. Continue step 3 till Q is empty. 5. We are given distance numbers d_v for all vertices and compare all values in d_v with D array. 6. If both values in d_v and D are equal then return YES else return NO.

Correctness:
We are given shortest distances for a graph and it need not be exact shortest distances. Our aim is to check whether these distances are shortest distances or not. Return YES if it is shortest distance, otherwise NO. To compare the values, we used a queue to compute lowest possible distance from source and start checking each value obtained with given shortest distances. If the shortest distances does not match, distances computed then this algorithm return NO.

Running Time Analysis:
To create distance array with V vertices and to create Queue to store all vertices takes O(|V|) time each. We dequeued vertex and checking to find lowest possible distance which takes O(|E| + |V|) time. At the end we compare all distance numbers with distance array in O(|V|) time.
Hence, this algorithm has running time = O(|V|) + O(|V|) + O(|V|) + O(|E| + |V|)
= O(|E| + |V|).

10) Sol.

Algorithm to find union by rank.
Input: Tree with root 'r' Output: Union by rank

Steps: 1. Find node of ‘a’ and ‘b’ using below Find algorithm. 2. If (‘a’ == ‘b’) then return. 3. If rank(‘a’) > rank (‘b’) then parent(‘b’) ⇐ ‘a’. 4. Else root (‘a’) ⇐ ‘b’ 5. If rank (‘a) == rank (‘b’) then increment rank(‘b’) by 1. 6. Continue till all nodes in the tree are ranked.

Algorithm to Find an element.
Input: An element or node in the tree. Output: An element or node if it is available in the tree.
Steps: 1. Scan all the elements or nodes in the tree. 2. If we encountered with element we are looking for then halt process and return that element. 3. Move to the next element and continue till all the nodes or elements or scanned.

Correctness:
This algorithm verifies rank for each node and it tries to link rank of smaller node to rank of large node. Find operation takes care of verifying each and every node possible to determine required node. This algorithms performs union and find operations efficiently. These operations can be useful in kruskal’s algorithm to find minimum spanning tree.

Running Time Analysis:
Using union by rank method, any union or find operation together takes O(logn).
If we have to union m elements and have to apply find operation on n elements then, above algorithms takes O(m log n) if n<m or O(m log m) if m < n.
Hence, this algorithm has complexity = Ω(m min(log n, log m)).

References:
1. https://en.wikipedia.org/wiki/Conditional_expectation
2. https://en.wikipedia.org/wiki/Martingale_(probability_theory)
3. https://en.wikipedia.org/wiki/Bipartite_graph
4. https://en.wikipedia.org/wiki/Minimum_spanning_tree
5. https://en.wikipedia.org/wiki/NP-completeness