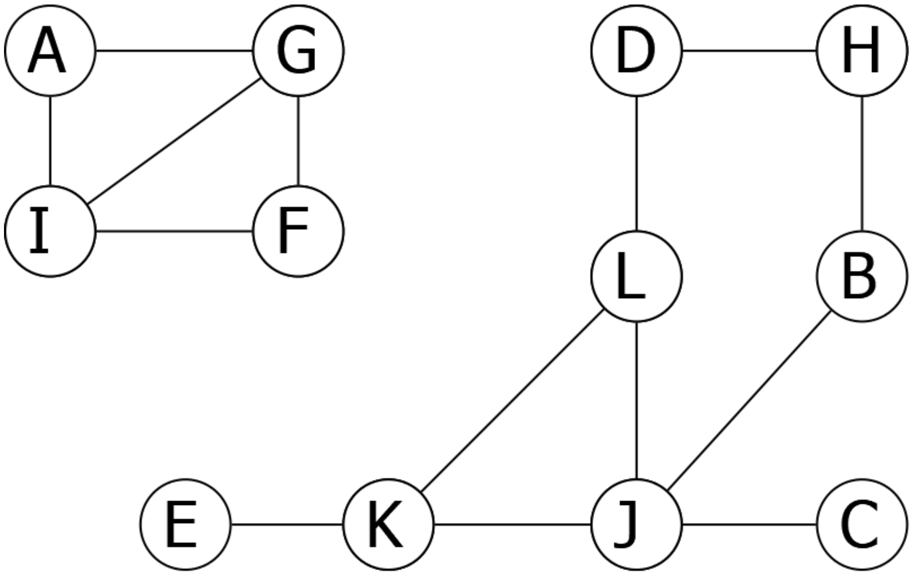
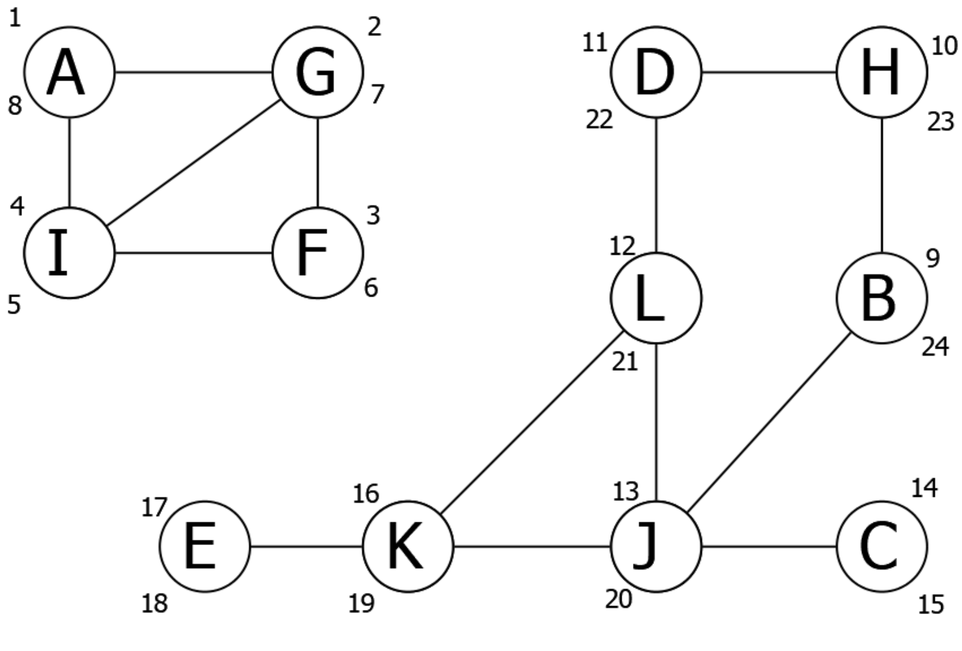
Question 1 (DFS, 30 points)**.** *Compute the pre- and postorders of all vertices when DFS is run on the graph below. Whenever DFS has several options of which order to do things in, always visit the alphabetically first vertex first.*



Running DFS, we get the following output:

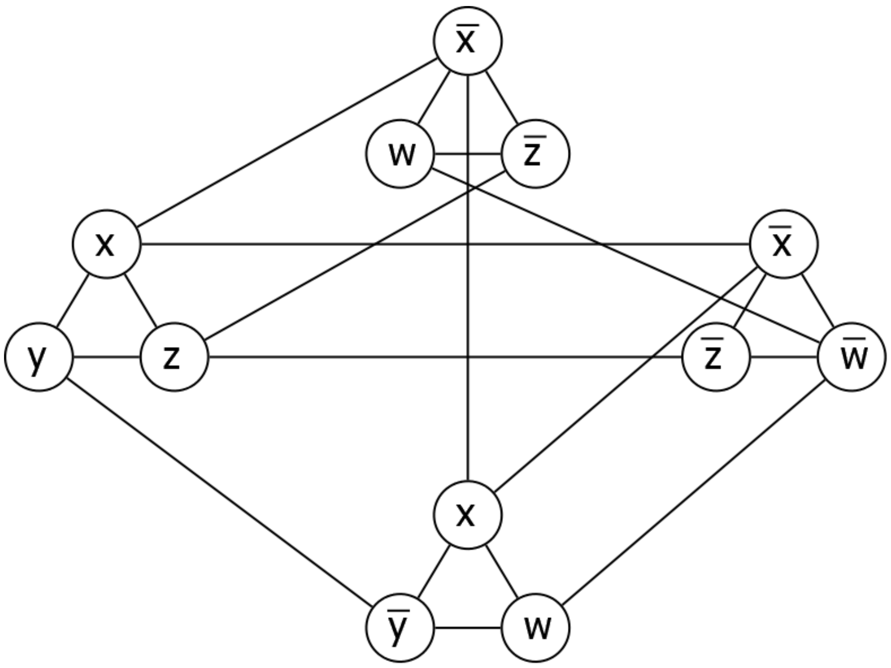


Question 2 (3SAT to MIS Reduction, 30 points). *Using the reduction discussed in class, what Maximum Independent Set problem can be used to solve the 3SAT instance*

$$(x \vee y \vee z) \wedge (\bar{x} \vee w \vee \bar{z}) \wedge (\bar{x} \vee \bar{z} \vee \bar{w}) \wedge (x \vee \bar{y} \vee w)$$

and how would you interpret the answer to solve the original 3SAT?

We get the following graph:



The original 3SAT has a solution if and only if the Maximum Independent Set of the above graph has size at least 4.

Question 3 (Finding Close Points, 35 points)**.** *Let L be a list of n different numbers in $[0, 1]$. Give a divide and conquer algorithm that finds a pair of numbers in L that differ by at most $1/(n-1)$ (if more than one pair exists, you only need to return one of them). For full credit your algorithm should run in time $O(n)$ and use divide and conquer.*
Hint: You may want to generalize this so that for any list of n elements in an interval $I = [a, b]$, you can find two elements that differ by at most $(b-a)/(n-1)$.

We can use the following algorithm:

```
ClosePair(L,a,b)
  If |L| = 2, return both elements
  Pick random x in L
  Sort L into:
    x
    B = {y > x}
    S = {y < x}
  If (x-a)/|S| <= (b-a)/(n-1)
    Return ClosePair(S U {x}, a, x)
  If (b-x)/|B| <= (b-a)/(n-1)
    Return ClosePair(B U {x}, x, b)
```

To analyze the runtime, we note that this is a divide and conquer algorithm with overhead $O(n)$ (from sorting L into sets). We only recurse on a single set and (as in the analysis of our order statistics algorithm) with probability at least 50% the recursive call is on a set of size at most $3n/4 + O(1)$. Thus we get a recurrence of $T(n) = T(3n/4) + O(1)$ giving $T(n) = O(n)$.

To show correctness, we use induction on $|L|$. If $|L| = 2$, the difference between the elements of L is at most $(b-a) = (b-a)/(n-1)$, and so it works. Assuming that the algorithm works for smaller sets L we first note that one of the later if statements must hold. This is because $|B| + |S| = |L| - 1 = n - 1$. Therefore,

$$[(b-a)|S|] + [(b-a)|L|] = (b-a)(n-1) = [(x-a)(n-1)] + [(b-x)(n-1)].$$

This means that either $[(b-a)|S|] \geq [(x-a)(n-1)]$ or $[(b-a)|L|] \geq [(b-x)(n-1)]$ so one of the if statements must hold. Whichever does hold, by the inductive hypothesis, the recursive call to `ClosePair` should return a pair that is sufficiently close.

Note: There are other solutions to this problem. One splits the interval I into two roughly equal pieces, determines the number of points in each sub-interval and recurses on an appropriate one. This algorithm has the advantage of being deterministic. One can also use a non-divide and conquer algorithm by splitting I into sub-intervals $[0, 1/(n-1)), [1/(n-1), 2/(n-1)), \ldots, [(n-2)/(n-1), 1]$ and finding some subinterval with at least two points in it.

Question 4 (Marked Vertex Shortest Paths, 35 points)**.** *Let G be an undirected graph with non-negative edge weights and two vertices s and t. Some of the vertices of G including s and t are marked as special. Your goal is to find the length of the shortest $s-t$ path in G so that this path doesn't have more than three non-special vertices in a row.*

For full credit, your algorithm should run in time $O(|V|\log(|V|)+|E|)$.

We first construct a new graph G'. The vertices of G' are the same as the vertices of G except that for each non-special vertex v of G, we have three copies v_1, v_2, v_3 in G'. For each edge $e=(v,w)$ in G if v and w are both special, we have a vertex (v,w) in G' of the same weight. If v is special, but w is not, we have an edge (v,w_1) in G' of the same weight. If w is special and v is not we have edges from each v_i to w, and if neither are special we have edges (v_1,w_2) and (v_2,w_3) all of the same weight. Our algorithm them uses Dijkstra to compute the shortest path length from s to t in G'.

In terms of runtime, we note that we can construct G' in linear time and run Dijkstra in time $O(|V|\log(|V|)+|E|)$.

To show correctness we claim that any path in G which has no more than 3 non-special vertices in a row has a corresponding path in G' of the same length and visa versa. It is not hard to see that for any path in G without more than 3 non-special vertices in a row, there is a corresponding path in G' where the path visits a vertex v_i in G' when the original path visited a vertex v and that was the i^{th} non-special vertex in a row.

Question 5 (Small Gap Interval Cover, 35 points). *Let $I = [a, b]$ be an interval of real numbers and let S be a set of n subintervals of I. Design an algorithm that given I and S determines whether or not there is a subset T of S so that:*

1. *No two intervals in T overlap.*
2. *The intervals in T cover all of I except for a number of gaps of length at most 1. In particular the gaps between consecutive elements of T and between the endpoints of I and the closest elements of T are all of length at most 1.*

For full credit, your algorithm should run in time $O(n \log(n))$ or better.

We set this up as a dynamic program. We let `Gap(x)` be the minimum value of t so that the interval $[x + t, b]$ has an appropriate set of subintervals from S with small gaps with the first interval starting at $x + t$. We note that for some interval $[x, y] \in S$ that $\texttt{Gap}(x) = 0$ if $\texttt{Gap}(y) \leq 1$ (since in this case, we can use the interval $[x, y]$ followed by an appropriate cover of $[y, b]$). Otherwise, $\texttt{Gap}(x) = (x' - x) + \texttt{Gap}(x')$ where x' is the next smallest interval start. This gives us the following algorithm:

```
SmallGapIntervalCover(a,b,S)
  Sort the elements of S by their starting endpoint
  Create arrays x[1...n+1], y[1...n+1], G[1...n+1]
  Let the ith interval of S by [x[i],y[i]] and x[n+1]=y[n+1]=t
  Let G[n+1] = 0
  For j = n...1
    Use binary search to find the smallest i so that x[i] > y[j]
    If G[i] + x[i] - y[j] <= 1
      G[j] = 0
    Else
      G[j] = G[j+1] + x[j+1] - x[j]
  If G[1] + x[1] - a <= 1
    Return 'Possible'
  Else
    Return 'Impossible'
```

To show correctness, we note that this correctly computes the recurrence relation as described above and that the final small gap cover is possible if and only if $x_1 - a + \texttt{Gap}(x_1) \leq 1$.

The runtime is $O(n \log(n))$. This is how long it takes to sort S. Additionally, the main loop runs n times and is dominated by the $O(\log(n))$-time binary search step.

Question 6 (Double Weight MST, 35 points). *In the* ***`Double-Weight-MST`*** *problem, you are given a graph G where each edge e is assigned two weights $w_1(e)$ and $w_2(e)$. For any spanning tree T we define two weights $w_1(T)$ and $w_2(T)$ to be the sum of $w_1(e)$ over all edges $e \in T$ and the sum of $w_2(e)$ over these edges, respectively. The goal is to find the tree T so that $\max(w_1(T), w_2(T))$ is as small as possible. Prove that* ***`Double-Weight-MST`*** *is NP-Hard.*
Hint: Construct a G so that $w_1(T) + w_2(T)$ is the same no matter what T you pick. Therefore, maximizing their product will require finding a T with $w_1(T) = w_2(T)$.

Clearly the problem is in NP. We prove hardness by reduction from SubsetSum.

In particular, given a set S of positive numbers and a target L, we construct a graph G with vertices a, b and a vertex v_i for each $x_i \in S$. We have edges between a and b and between each v_i and each of a and b.

Let R be the sum of the elements of S minus L. We note that the SubsetSum problem is solvable if and only if S can be partitioned into two subsets with sums T and R.

Define weights by $w_1(a, b) = -L, w_2(a, b) = -R$, $w_1(v_i, a) = 0 = w_2(v_i, b)$ and $w_1(v_i, b) = x_i = w_2(v_i, a)$. Note that any ideal tree must contain the edge (a, b) as it is the cheapest edge under both weights (and thus, for any other tree, we can replace some edge by (a, b) to decrease both weights). This means that for each i exactly one of (v_i, a) or (v_i, b) will be in T. This means that the sum $w_1(T) + w_2(T)$ will be 0. Therefore $\max(w_1(T), w_2(T))$ will always be at least 0. This is achievable if and only if we can find such a T so that $w_1(T) = w_2(T) = N$. This in turn happens if and only if the sum of x_i over $i's$ with (v_i, b) in T adds up to L. This is possible if and only if the original SubsetSum was solvable.

Thus, our original SubsetSum problem was solvable if and only if there is a tree T with $\max(w_1(T), w_2(T)) \leq 0$.