**Solutions**

1. Given an edge-weighted undirected connected chain-graph $G = (V, E)$, all vertices having degree 2, except two endpoints which have degree 1 (there is no cycle). Design an algorithm that preprocesses the graph in linear time and can return the distance of the shortest path between any two vertices in constant time (i.e., the $\mathcal{O}(|V|)$ preprocessing enables return of the shortest-path distance between any two vertices in $\mathcal{O}(1)$ time).

**Solution:** Given that this is a linear chain, we can run DFS from the source vertex and record the distance from the source to each vertex in this chain (shortest path to each vertex). To get the distance between any two vertices $u$ and $v$, we would simply do $|dist(u) - dist(v)|$.

Pseudocode

```
v is vertex with v.degree = 1
Preprocess(Graph g, Vertex v)
 visited(v) = true
 v.dist = 0
 Stack s
 s.push(v)
 while(s.size()>0)
  v' = s.top()
  flag = 0
  for each edge(v', u) in E
   if not visited(u):
    visited(u) = true
    u.dist = v'.dist + edge(v',u)
    s.push(u)
    flag = 1
    break
  if flag==0 // all neighbors visited
   stack.pop()


Dist(Vertex u, Vertex v, Graph preprocessed)
 return |u.dist−v.dist|
```

Preprocess has run time of DFS, $O(|V| + |E|)$. Preprocess must iterate through every vertex and edge, thus the best run time must be linear. Dist has a runtime of $O(1)$ since preprocess will have labeled each vertex with a distance from source, we only need one operation to get the distance between two vertices.

2. The interval scheduling problem is to find a largest compatible set - a set of non-overlapping intervals of maximum size. Suppose that instead of always selecting the first activity to finish, we select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

**Solution:** How Algorithm is Greedy:
Greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. The algorithm proposed here chooses jobs that start last as the locally optimal choice.
Proof of optimality:
We are given a set $S = a_1, a_2, \cdots, a_n$ of activities, where $a_i = [s_i, f_i)$, and the algorithm proposed finds an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Instead, let us create a set $S' = a'_1, a'_2, \cdots, a'_n$, where $a'_i = [f_i, s_i)$. That is, $a'_i$ is $a_i$ in reverse. Clearly, a subset of $a_{i1}, a_{i2}, \cdots, a_{ik} \subseteq S$ is mutually compatible if and only if the corresponding subset $a'_{i1}, a'_{i2}, \cdots, a'_{ik} \subseteq S'$ is also mutually compatible. Thus, an optimal solution for $S$ maps directly to an optimal solution for $S'$ and vice versa.
The algorithm proposed selects the last activity to start (in $S$) that is compatible with all previously selected activities in $S$, this algorithm is equivalent to selecting the first activity to finish (in $S'$) that is compatible with all previously selected activities in $S'$. Since an optimal solution for $S'$ maps directly to an optimal solution for $S$, the algorithm proposed yields an optimal solution.

3. Let $T$ be an MST of a weighted, undirected graph $G$. Given a connected subgraph $H$ of $G$, show that $T \cap H$ is contained in some MST of $H$.

**Solution:** Let $T \cap H = \{e_1, e_2, \cdots e_k\}$. We use the cut property repeatedly to show that there exists an MST of H containing $T \cap H$. Cut property states that, for a graph $G'$, let $R$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $R$. Then the MST $T'$ contains $e$.

Suppose for $i \geq 0$, $X = \{e_1, e_2, \cdots e_i\}$ is contained in some MST of $H$. Removing the edge $e_{i+1}$ from $T$ divides $T$ in two parts giving a cut $(S, G \backslash S)$ and a corresponding cut $(S_1, H \backslash S_1)$ of $H$ with $S_1 = S \cap H$. Now, $e_{i+1}$ must be the lightest edge in $G$ (and hence also in $H$) crossing the cut, else we can include the lightest and remove $e_{i+1}$ to get a better tree. Also, no other edges in $T$, and hence in $X$, cross the cut. We can then apply the cut property to get that $X \cup e_{i+1}$ must be contained in some MST of $H$. Continuing in this manner we will get that $T \cap H = \{e_1, e_2, \cdots e_k\}$ must be contained in some MST of $H$.

4. Given an edge-weighted undirected graph $G = (V, E)$, such that $|E| > |V|$ and all **edge-weights are distinct**.
We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of $G$, and let $T'$ be a MST of $G$. Then a second-best minimum spanning tree is a spanning tree $T$ such that $W(T) = \min_{T'' \in \mathcal{T} - \{T'\}}(W(T''))$, $W(\cdot)$ denotes weight of spanning tree.

   (a) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

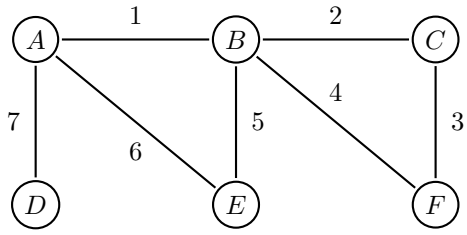
   (b) Let $T'$ be the minimum spanning tree of $G$. Prove that $G$ contains edges $(u, v) \in T'$ and $(x, y) \notin T'$ such that $T'$ - $\{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of $G$.

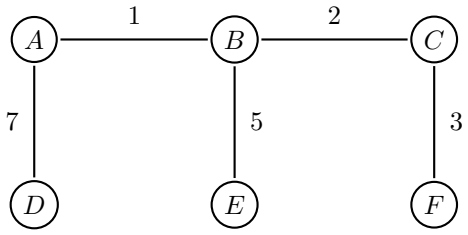
   (c) Let $T$ be a spanning tree of $G$ and for any two vertices $u$, $v \in V$, let $\max(u, v)$ denote an edge of maximum weight on the unique simple path between $u$ and $v$ in $T$. Design an $\mathcal{O}(|V|^2)$ time algorithm that, given $T$, computes $\max(u, v)$; for all $u$, $v \in V$.

   (d) Design an efficient algorithm to compute the second-best minimum spanning tree of $G$.

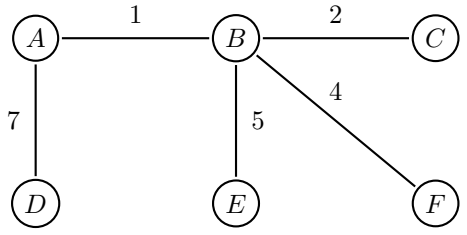
**Solution:**

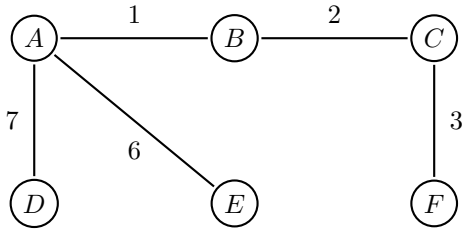
(a) Since the graph is connected and all edge weights are distinct, for each cut $(S, S')$, there is a unique light edge (i.e., of smallest cost $c_e$) crossing the cut (call it $e_S$). By the cut property, each such edge $e_S$ must be part of every minimum spanning tree. Define $E = \{e_S | (S, S') \text{ is a cut}\}$, where $E$ itself is already connected, because it contains an edge for every cut. Hence, $E$ itself is the MST of G which would be unique.
The second-best minimum spanning tree need not be unique, below is a weighted, undirected graph with a unique minimum spanning tree of weight 18 and two second-best minimum spanning trees of weight 19.
Graph and MST:



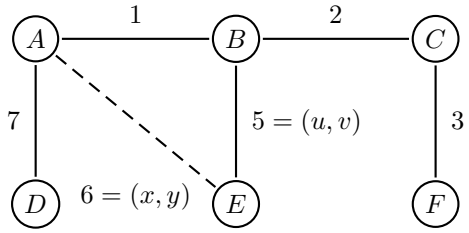

Second Best minimum spanning trees:




(b) Since any spanning tree has exactly $|V| - 1$ edges, any second-best minimum spanning tree must have at least one edge that is not in the (best) minimum spanning tree. If a second-best minimum spanning tree has exactly one edge, say $(x, y)$, that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except

that $(x, y)$ replaces some edge, say $(u, v)$, of the minimum spanning tree. In this case, $T = T' - \{(u, v)\} \cup \{(x, y)\}$, as we wished to show.
Thus, all we need to show is that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree. Let $T'$ be the minimum spanning tree of $G$, and suppose that there exists a second-best minimum spanning tree $T$ that differs from $T'$ by two or more edges. There are at least two edges in $T' - T$, and let $(u, v)$ be the edge in $T' - T$ with minimum weight. If we were to add $(u, v)$ to $T$, we would get a cycle $c$. This cycle contains some edge $(x, y)$ in $T - T'$ (since otherwise, $T'$ would contain a cycle). Since the edges $(u, v)$ and $(x, y)$ would be on a common cycle $c$, the set of edges $T - \{(x, y)\} \cup \{(u, v)\}$ is a spanning tree, and its weight is less than $w(T)$. Moreover, it differs from $T'$ (because it differs from $T$ by only one edge). Thus, we have formed a spanning tree whose weight is less than $w(T)$ but is not $T'$ . Hence, $T$ was not a second-best minimum spanning tree.
Figure below indicates edge $(u, v)$ and $(x, y)$ for example taken in part (a).




(c) Algorithm: We will do breadth first search from vertex $u$, having restricted the edges visited to those of the spanning tree $T$. This will give us $max(u, v)$ for all vertex $v \neq u$ since we are visiting via edges in a spanning tree of an undirected graph, we are guaranteed that the search from each vertex $u$ will visit all vertices. This procedure is done for all the vertices $u \in V$ to get $max(u, v)$ for all vertices $u$ and $v$

Pseudocode:

```
max[V,V] = NULL   //Initialization
Bfs-Max(G ,T)
 for each vertex u ∈ V
 Q.enqueue(u)
 while Q is not empty
   x = Q.dequeue()
   for each v ∈ Adj[x]
     if max[u, v] == NULL and v ≠ u
       if x == u or w(x, v) > max[u, x]
         max[u, v] = (x, v)
       else max[u, v] = max[u, x]
     Q.enqueue(v)
```

Runtime Complexity: We are doing BFS from each vertex $u \in V$. Hence runtime complexity of above algorithm is $\mathcal{O}|V|(|V| + |E|) = \mathcal{O}(|V|^2)$. $|E| = |V - 1|$ for spanning tree.

(d) Algorithm to find second best minimum spanning tree is as follows:

i. Compute the minimum spanning tree $T'$ using Prim's or Kruskal's Algorithm.
ii. Given the minimum spanning tree $T'$, compute the max table, as in part (c).
iii. Find an edge $(u, v) \notin T'$ that minimizes $w(max[u, v]) - w(u, v)$.

iv. Having found an edge $(u, v)$ in step (iii), return $T = T' - \{max[u, v]\} \cup \{(u, v)\}$ as a second best minimum spanning tree. Since adding $(u, v)$ completes a cycle, and we remove an MST edge $max[u, v]$ to regain a spanning tree topology, minimizing the cost increase $w(u, v) - w(max[u, v])$.

Pseudocode

```
Second-best-mst(G = (V,E))
T = (V, E') ← Prim'sAlgorithm(G= (V,E))
max[V,V] ← Bfs-Max(G,T)
max_val = 0
for all edges (u,v) ∉ E'
  if w(max[u,v]) - w(u,v) > max_val
    edge_picked = (u, v)
return T - {max[edge_picked]}∪{edge_picked}
```

Runtime complexity Time complexity for Prim's algorithm is $\mathcal{O}(E + Vlog(V))$ which is $\mathcal{O}(|V|^2)$, time complexity to fill matrix max[V,V] is $\mathcal{O}(|V|^2)$ (from part (c)). To find out edge that minimizes $w(max[u, v]) - w(u, v)$ will take time $\mathcal{O}(|E|)$ which is again $\mathcal{O}(|V|^2)$, hence overall time complexity of the algorithm is $\mathcal{O}(|V|^2)$.

5. Given a graph $G = (V, E)$, a subset $S \subseteq V$ of vertices is said to be a *vertex cover* of $G$ if for every edge $(u, v) \in E$, at least one of $u$, $v$ belongs to the subset $S$. A minimum vertex cover of $G$ is a vertex cover with minimum cardinality (i.e., smallest vertex cover).
Design an algorithm to find a minimum vertex cover of a given tree $T = (V, E)$. Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

**Solution:** The following claim will lead to an algorithm.
Claim: Let $v$ be any vertex in $G$ that has only one incident edge and let this edge connect $v$ to another vertex $u$. Let $G'$ be the graph obtained from $G$ by removing $u$ and its incident edges from $G$. Let $S$ be a minimum vertex cover of the graph $G'$. Then $S \cup \{u\}$ is a minimum vertex cover of $G$.
Proof: First, we argue that there is a minimum vertex cover of $G$ that includes the vertex $u$. Suppose for the sake of contradiction that no minimum vertex cover of the graph includes $u$. Consider a minimum vertex cover $R$. Since $u \notin R$, this means that $v \in R$. Consider the set $R' = R - \{v\} \cup \{u\}$.
Note that $R'$ is also a vertex cover which is a contradiction.
Suppose for the sake of contradiction assume that $S \cup \{u\}$ is not a minimum vertex cover of $G$. Let $Q$ be a minimum vertex cover of $G$ that includes $u$ (from the argument above we know there exists such a set). So, we can write $Q = Q' \cup \{u\}$ such that $u \notin Q'$. Note that $Q'$ is a vertex cover of $G'$ (since $u$ does not "cover" any edges in $G'$). This is a contradiction, since from our assumption $|Q'| < |S|$.
The correctness of the following algorithm easily follows from the above claim.
Note that tree (or the forest of trees) will always have a vertex $v$ that has only one incident edge (leaf vertex) unless there are no edges in the forest of trees.

Pseudocode:

```
TreeVertexCover(G = (V, E))
 S = {}
 while there are no edges in G
   Let v be a vertex with exactly one incident edge (u, v)
   S = S ∪ {u}
   Remove u and its incident edges from G to obtain G'
   G = G'
 return S
```

Runtime complexity: We need to maintain the degrees of vertices in the current graph and vertices with degree 1. Updating the degrees after every iteration will cost time proportional to the degree of the vertex being removed. So, the total time will be proportional to the sum of degrees which in this case would be $\mathcal{O}(|V|)$.

6. Given an edge-weighted directed graph $G = (V, E)$ such that all the edge-weights are positive. Let $s$ and $t$ be two vertices in $G$ and $k \leq |V|$ be an integer. Design an algorithm to find the shortest path from $s$ to $t$ that contains exactly $k$ edges.
Note that the path need not be simple, and is permitted to visit vertices and edges multiple times.

**Solution:** For this, we will use dynamic programming. Our base condition is that for $k = 0$ all distances will be set to infinity except $s = 0$. At any given $m$, from $m = 1$ to $k$, we can find the distance to any vertex $u$ using exactly $m$ edges as:

$$D[m, u] = min\{D[m-1, x] + w[x, u]\}$$

for all edges $(x, u)$ where $x$ is a predecessor of $u$ with an edge from $x$ to $u$. If there are no incoming edges, $D[m, u] = \infty$.

Using this formula, we can build a $k$ by $|V|$ matrix, which will keep track of all shortest path weights using exactly $m$ edges from $s$ at iteration $m$. We will find the weight of the shortest path in row $k$, column vertex $t$ (which will give shortest path weight from $s$ to $t$ using exactly $k$ edges.), from there we must backtrack to reconstruct the shortest path.

Pseudocode:

```
Shortest-path-k-edges(G, k, s , t)
D[k,V], P[k,V]    -Initialization
D[0,s] = 0;
for all vertices u except s
  for m = 0 to k:
    D[m,u] = ∞
for m = 1 to k:
  for every edge(x,u):
    if (D[m-1,x] + w[x,u]) < D[m, u]
      D[m, u] = D[m-1,x] + w[x,u]
      P[m, u] = x
Path = emtpy list
if D[k,t] = ∞
  Return no path
last = t
for m = k down to 0:
  Path.append(P[m,last])
  last = P[m,last]
Path = Path.reverse()
Return Path
```

Runtime Complexity Initializing matrix has complexity $\mathcal{O}(k|V|)$ and for each iteration, we are going through each edge $\in E$ to update Distance matrix $D$ and Path matrix $P$. In total there will be $k$ iterations to fill matrices. Hence time complexity in building matrices is $\mathcal{O}(k|E|)$. Traversing through the path has time complexity $k$. Hence, overall time complexity is $\mathcal{O}(k(|E| + |V|))$.

7. You are going on a long trip. You start on the road at mile post 0. Along the way there are $n$ hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each $a_i$ is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance $a_n$), which is your destination.

   You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel $x$ miles during a day, the penalty for that day is $(200 - x)^2$ . You want to plan your trip so as to minimize the total penalty- that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

**Solution: Suproblems definition**
Let $OPT(i)$ be the minimum total penalty to get to hotel $i$.

**Recursive formulation**
To get $OPT(i)$, we consider all possible hotels $j$ we can stay at the night before reaching hotel $i$. For each of these possibilities, the minimum penalty to reach $i$ is the sum of:

- the minimum penalty $OPT(j)$ to reach $j$,
- and the cost $(200 - (a_j - a_i))^2$ of a one-day trip from $j$ to $i$.

Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all the j:

$$OPT(i) = \min_{0 \leq j < i} \{OPT(j) + (200 - (a_j - a_i))^2\}$$

And the base case is $OPT(0) = 0$.

**Pseudo-code**

```
// base case
OPT[0] = 0
// main loop
for i = 1...n:
  OPT[i] = min([OPT[j] +(200 − (a_j − a_i))^2 for j=0...i-1])
// final result
return OPT[n]
```

**Complexity**
- We have n subproblems,
- each subproblems i takes time O(i)

The overall complexity is

$$\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n}) i = O\frac{n(n-1)}{2} = O(n^2)$$