1. You are given a list $D[n]$ of n words each of length k over an alphabet Σ in a language you don't know, although you are told that words are sorted in lexicographic order. Using $D[n]$, describe an algorithm to efficiently identify the order of the symbols in Σ. For example, given the alphabet $\Sigma = \{Q, X, Z\}$ and the list $D = \{QQZ, QZZ, XQZ, XQX, XXX\}$, your algorithm should return QZX. You may assume D always contains enough information to completely determine the order of the symbols. (Hint: use a graph structure, where each node represents one letter.)

Solution: Consider two words, $D[i]$, $D[i+1]$. Consider j such that $D[i][j] \neq D[i+1][j]$ and $\forall k < j$, $D[i][k] = D[i+1][k]$. That is, j is the index of the first different letter between $D[i]$ and $D[i+1]$. The comparison of $D[i][j]$ and $D[i+1][j]$ reveals the order between the two letters. Any further comparison of $D[i]$ and $D[j]$ would not help, since the following letters do not affect lexicographic order of $D[i]$ and $D[i+1]$. Also, for arbitrary x, y, z such that $x < y < z$, if you are given the comparisons of $D[x], D[y]$ and $D[y], D[z]$, then the comparison of $D[x], D[z]$ does not reveal any additional information about the order(Why? Let j, k be the first different index between $D[x], D[y]$ and $D[y], D[z]$ respectively. Reason about three cases: $j < k$, $j = k$, $j > k$). Therefore, the problem can be solved by constructing the following directed graph.

- $V = \{v \mid v \in \Sigma\}$
- $E = \{(u, v) \mid u, v \in \Sigma, u \neq v, \exists i, j \text{ s.t. } D[i][j] = u, D[i+1][j] = v, \forall k < j, D[i][k] = D[i+1][k]\}$

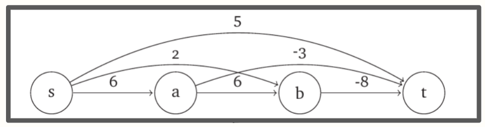
Note that for any edge $(u, v) \in E$, there is a corresponding pair of consecutive words $(D[i], D[i+1])$ such that if j is the index of the first different letter, then $D[i][j] = u$ and $D[i+1][j] = v$. This means for any edge (u, v), we know for sure that u comes before v in their language. Since there can be no cycle in the graph, it can be topologically sorted to obtain the order of symbols. The running time of the algorithm is $O(nk)$, since in worst case we should iterate over every symbol in D to construct the graph.

■

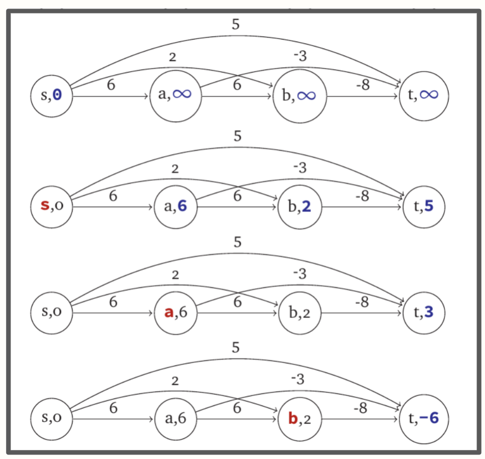
2. Given a directed-acyclic-graph ($G = (V, E)$) with integer (positive or negative) edge weights:

   (a) Give an algorithm to find the **shortest** path from a node s to a node t.

Solution: Because there are negative edge weights we cannot use a greedy method like Dijkstra's algorithm; however because the graph is directed and acyclic we can use a topological sort. Topologically sorting the graph means that every vertex can only reach vertices below them in the sort and cannot reach vertices above them in the sort.



This means after topologically sorting the graph we start at node s and compute the shortest path from node s to each of the nodes below it in sequential order. To do this we initialize the distance from s to all the other nodes as ∞ then we update the distance from node s to be the weight of the edges from s to the neighbors of s. Then we move on to the next sequential node say u, and look at its neighbor, say v. If the distance from s to u plus the edge weight from u to v is less than the current distance from s to v then we update the value. This repeats until we reach node t.



Let s and t be the nth and mth node in the topological sort and let $N(u)$ be the neighbor set of u. Then the algorithm is

```
DAGSP(V, E, s, t):
    (Vs, Es) ← TopSort(V, E)
    n ← index(s)
    m ← index(t)
    D[n] ← 0
    for k ← n + 1 to m
        D[k] ← ∞
    for i ← n to m − 1
        for v in N(v_i)
            j ← index(v)
            D[j] ← min{D[j], D[i] + Es[i][j]}
    return D[m]
```

A topological sort takes $\boldsymbol{O(V+E)}$ time and the for loops in the algorithm takes $\boldsymbol{O(V+E)}$. So the total running time is $\boldsymbol{O(V+E)}$. ■

(b) Give an algorithm to find the **longest** path from a node s to a node t.

Solution (Direct): To find the longest path from node s to node t we can do the same process as part a) but instead we initialize the values to $-\infty$ then take the maximum value.

```
DAGLP(V, E, s, t):
    (Vs, Es) ← TopSort(V, E)
    n ← index(s)
    m ← index(t)
    D[n] ← 0
    for k ← n + 1 to m
        D[k] ← −∞
    for i ← n to m − 1
        for v in N(v_i)
            j ← index(v)
            D[j] ← max{D[j], D[i] + Es[i][j]}
return D[m]
```

This has a running time of $\boldsymbol{O(V+E)}$.

■

Solution (Reduction): This problem can be reduced to the part a). Multiplying all of the edge weights by -1 then finding the shortest path then multiplying that value by -1 yields the longest path.

```
DAGLP(V, E, s, t):
    Et ← −1 ∗ E
    x ← DAGSP(V, Et, s, t)
    return −1 ∗ x
```

This has a running time of $\boldsymbol{O(V+E)}$.

■

3. You are given a directed acyclic graph $G = (V, E)$ with possibly negative weighted edges:

   (a) Give an algorithm that finds the length of the shortest path that contains at most k edges between two vertices u and v in $O(k(n+m))$ time.

   Solution: The idea is to create $k+1$ copies of the original vertices V and connect u in the (j)th copy to v in the $(j+1)$th copy, for every $(u, v) \in E$. Since every edge leads to the next copy, and we only have $k+1$ copies, every path in the new graph G' would contain no more than k edges. Then we can run the algorithm from problem 2 (a) on G' to find the shortest path.

   Define $G' = (V', E')$ as the following:

   - $V' = V \times \{0, 1, 2, 3, \ldots, k\}$
   - $E' = \{((u, j), (v, j+1)) \mid (u, v) \in E, 0 \leq j \leq k-1\}$

   Run the algorithm from problem 2 (a) starting from the vertex $(s, 0)$. Then, $\min_{0 \leq j \leq k} D(t, j)$ is the length of the shortest path with at most k edges, where $D(i, j)$ represents the shortest distance to the node (i, j) from $(s, 0)$ obtained by the algorithm.

   Since the graph construction takes $O(k(n+m))$ time, and the algorithm from problem 2 (a) is a linear time algorithm, the overall runtime would be $O(k(n+m))$. ■

   (b) Give an algorithm that finds the length of the shortest path that contains exactly k edges between two vertices u and v in $O(k(n+m))$ time.

   Solution: You can apply the same graph modification and the shortest path algorithm as part (a). The only difference is that the answer to this problem would be $D(t, k)$ instead of $\min_{0 \leq j \leq k} D(t, j)$. ■