CS6161 Midterm Review

Jibang Wu

October 13, 2020

D & C

☆ Merge Sort / QuickSort

- FFT / Convex Hull

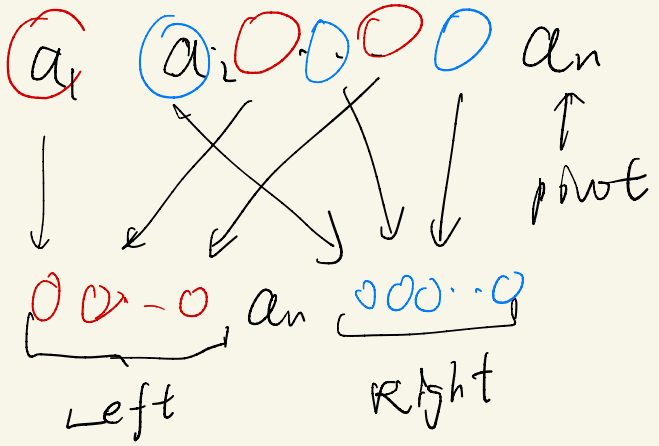
HW2 Problem 1.

- worst $O(n^2)$ → (2)
- average $O(n \log n)$ → $n! - 2$

} $O(n \log n)$

1 2 ... n-1 n $O(n^2)$

1 2 .. n-2 n n-1 $O(n^2)$



The left sequence remains the same order as the original sequence.

⇒ Left sequence is also uniformly random.

Shortest Path on Graph

1. single-source i 

- Dijkstra $O(n \log n + m)$ $w_{ij} \geq 0$
- Bellman-Ford $O(mn)$ otherwise

2. All-Pair

- n · Dijkstra
- n · Bellman Ford
- Floyd-Warshall $O(n^3)$ w_{ij} arbitrary

$G(V, E)$ $W = (w_{ij})$

Floyd:

$d_{ij}^{(k)}$ — the length of the shortest path from $i \to j$ only uses $\{v_1, v_2, \cdots, v_k\}$ as the intermediate nodes.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & k \geq 1 \end{cases}$$

New way to construct DP.

$l_{ij}^{(k)}$ — the length of the shortest path from $i \to j$ that uses at most k edges

$l_{ij}^{(n-1)}$

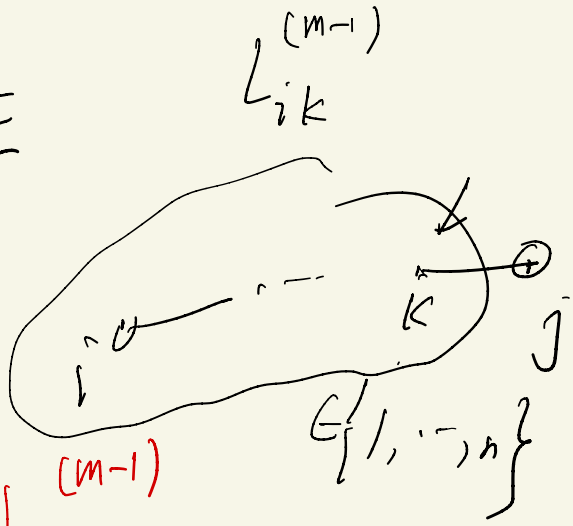
Initial steps:

$$l_{ij}^{(0)} = \begin{cases} 0, & i=j \\ \infty, & i \neq j \end{cases}$$

$$l_{ij}^{(1)} = \begin{cases} w_{ij}, & (i,j) \in E \\ \infty, & (i,j) \notin E \end{cases}$$

DP recursion:

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \{ l_{ik}^{(m-1)} + w_{kj} \}$$



Denote $L^{(m)} = (l_{ij}^{(m)})_{ij}$

$L^{(m)} \leftarrow L^{(m-1)}$

$n^2 \cdot n$

$$L^{(0)} \xrightarrow{O(n^2 \cdot n)} L^{(1)} \xrightarrow{O(n^3)} \cdots \longrightarrow L^{(n-1)} \xrightarrow{O(n^3)}$$

$\Rightarrow O(n^4)$

Given $L^{(m-1)}$, W, computing $L^{(m)}$ requires $O(n^3)$!

so $L^{(n-1)}$ takes $O(n^4)$

Can we improve this? Yes!

$$L^{(m)} = L^{(m-1)} \cdot W$$

$$l_{ij}^{(m)} = l_{i1}^{(m-1)} \times w_{1j} + l_{i2}^{(m-1)} \times w_{2j} + \cdots + l_{in}^{(m-1)} \times w_{nj}$$

row $L^{(m-1)}$ · column W

replace: $\times \to +$, $+ \to \min$

$$l_{ij}^{(m)} = l_{i1}^{(m-1)} + w_{1j} \min l_{i2}^{(m-1)} + w_{2j} \min \cdots \min l_{in}^{(m-1)} + w_{nj}$$

$L^{(m-1)}$ · W

$$L^{(m)} = L^{(m-1)} * W$$

$$\Longleftrightarrow \quad L_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ L_{ik}^{(m-1)} + w_{kj} \right\}$$

if we replace $\min \longrightarrow +$

$+ \longrightarrow \times$

then $L^{(n)} = L^{(n-1)} * W$

becomes matrix multiplication

$$L^{(n)} = L^{(n-1)} \cdot W$$

The operation '$*$' remains the structure of matrix multiplication!

$$L^{(n)} = L^{(n-1)} * W$$
$$= L^{(n-2)} * W * W$$
$$= \cdots$$
$$= W^n$$

Therefore, We can compute $L^{(n)}$ by directly computing W^n $(= w * w * \cdots * w)$

$w \longrightarrow w^n$

Do we need to compute $w^2, w^3, \cdots, w^{n-1}$ in order to obtain w^n? No!

e.g. $n = 2^m$. By D & C, $w^n = (w^{\frac{n}{2}})^2 = \cdots$

$$w \to w^2 \to w^4 \to w^8 \to \cdots \to w^{2^m}$$

$$O(m) = O(\log n)$$

$$L^{(0)} \to L^{(1)} \to \cdots \to L^{(n-1)} \quad O(n^4)$$

$$L^{(0)} \to L^{(1)} \to L^{(2)} \to L^{(4)} \to \cdots \to L^{2^{\lceil \log_2 n \rceil}}$$

$O(n^3 \log n)$

Prime EGZ

How to find the EGZ subset from any set of $2n-1$ integers?

Prime EGZ

How to find the EGZ subset from any set of $2n-1$ integers?

Define subproblem DP$[i][j][k]$,
among the first i elements of T, what are the j elements such that their sum $\mod n$ is k? If such j elements does not exists, the subproblem value is null.

Prime EGZ

How to find the EGZ subset from any set of $2n-1$ integers?

Define subproblem DP$[i][j][k]$,
among the first i elements of T, what are the j elements such that their sum $\mod n$ is k? If such j elements does not exists, the subproblem value is null.

Therefore, the decision problem for the original problem is DP$[2n-1][n][0]$.

Prime EGZ

We can derive the following DP update rules:

- The base case,

$$\text{DP}[0][0][i] = \begin{cases} \{\ \}, \text{ empty list} & i = 0 \\ \text{null} & i \neq 0 \end{cases}$$

Prime EGZ

We can derive the following DP update rules:

- The base case,

$$\mathrm{DP}[0][0][i] = \begin{cases} \{\ \}, \text{ empty list} & i = 0 \\ \text{null} & i \neq 0 \end{cases}$$

- The inductive update, denote $p = k - T[i] \mod n$

$$\mathrm{DP}[i][j][k] = \begin{cases} \mathrm{DP}[i-1][j][k] \\ \qquad \text{If } \mathrm{DP}[i-1][j-1][p] \text{ is null} \\ \{T[i]\} \cup \mathrm{DP}[i-1][j-1][p] \\ \qquad \text{Otherwise} \end{cases}$$

Bounded Knapsack

The *bounded* Knapsack problem has an additional constraint that item i only has k_i copies and thus you can pack at most k_i copies of i in the knapsack. Given W and $\{w_i, p_i, k_i\}_{i=1}^{n}$ as input, design a $O(Wn)$ time algorithm.

Bounded Knapsack

The *bounded* Knapsack problem has an additional constraint that item i only has k_i copies and thus you can pack at most k_i copies of i in the knapsack. Given W and $\{w_i, p_i, k_i\}_{i=1}^{n}$ as input, design a $O(Wn)$ time algorithm.

We can define the subproblem:
DP[i], as the optimal value with at most a total weight of i;
Used[i][j], as the usage of each item to achieve optimal value with at most a total weight of i.

Bounded Knapsack

The *bounded* Knapsack problem has an additional constraint that item i only has k_i copies and thus you can pack at most k_i copies of i in the knapsack. Given W and $\{w_i, p_i, k_i\}_{i=1}^{n}$ as input, design a $O(Wn)$ time algorithm.

We can define the subproblem:
DP[i], as the optimal value with at most a total weight of i;
Used[i][j], as the usage of each item to achieve optimal value with at most a total weight of i.

Therefore, the solution for the original problem is
DP[W], Used[W].

Bounded Knapsack

We can derive the following DP update rules:

- The base case,

$$\mathrm{DP}[0] = 0, \quad \mathrm{Used}[0][j] = 0, \ \forall j \in [n]$$

Bounded Knapsack

We can derive the following DP update rules:

- The base case,

$$\mathrm{DP}[0] = 0, \quad \mathrm{Used}[0][j] = 0, \ \forall j \in [n]$$

- The inductive update,

$$\mathrm{DP}[i] = \max_{j \in [n]} \left\{ \mathrm{DP}[i - w_j] + p_j \middle| i \geq w_j; \mathrm{Used}[i - w_j][j] \leq k_j \right\}$$

$$j^* = \arg\max_{j \in [n]} \left\{ \mathrm{DP}[i - w_j] + p_j \middle| i \geq w_j; \mathrm{Used}[i - w_j][j] \leq k_j \right\}$$

$$\mathrm{Used}[i][j] = \begin{cases} \mathrm{Used}[i - w_j][j] & j \neq j^* \\ \mathrm{Used}[i - w_j][j] + 1 & j = j^* \end{cases}$$

Generalized Knapsack

The *generalized* Knapsack problem: You have a total budget C for purchasing the items and total weight capacity W to pack those item. Design an $O(WCn)$ time algorithm to find out the number of copies of each item i, denoted as $x_i \in \mathbb{N}$, such that the total item value $\sum_{i=1}^{n} p_i x_i$ is maximized subject to the weight capacity, i.e., $\sum_{i=1}^{n} w_i x_i \leq W$ and balanced budget $\sum_{i=1}^{n} c_i x_i \leq C$. Note that we assume W, C, w_i, c_i's are all integers.

Generalized Knapsack

The *generalized* Knapsack problem: You have a total budget C for purchasing the items and total weight capacity W to pack those item. Design an $O(WCn)$ time algorithm to find out the number of copies of each item i, denoted as $x_i \in \mathbb{N}$, such that the total item value $\sum_{i=1}^{n} p_i x_i$ is maximized subject to the weight capacity, i.e., $\sum_{i=1}^{n} w_i x_i \leq W$ and balanced budget $\sum_{i=1}^{n} c_i x_i \leq C$. Note that we assume W, C, w_i, c_i's are all integers.

We can define the subproblem:
DP$[i][j]$, as the optimal value with at most a total weight capacity of i and a total budget of j.
Used$[i][j][k]$, as the usage of each item to achieve optimal value with at most a total weight capacity of i and a total budget of j.

Generalized Knapsack

The *generalized* Knapsack problem: You have a total budget C for purchasing the items and total weight capacity W to pack those item. Design an $O(WCn)$ time algorithm to find out the number of copies of each item i, denoted as $x_i \in \mathbb{N}$, such that the total item value $\sum_{i=1}^{n} p_i x_i$ is maximized subject to the weight capacity, i.e., $\sum_{i=1}^{n} w_i x_i \leq W$ and balanced budget $\sum_{i=1}^{n} c_i x_i \leq C$. Note that we assume W, C, w_i, c_i's are all integers.

We can define the subproblem:
DP$[i][j]$, as the optimal value with at most a total weight capacity of i and a total budget of j.
Used$[i][j][k]$, as the usage of each item to achieve optimal value with at most a total weight capacity of i and a total budget of j.

Therefore, the solution for the original problem is
DP$[W][C]$, Used$[W][C]$.

Generalized Knapsack

We can derive the following DP update rules:

- The base case,

$$\text{DP}[0][0] = 0, \quad \text{Used}[0][0][k] = 0, \ \forall k \in [n]$$

Generalized Knapsack

We can derive the following DP update rules:

- The base case,

$$\mathrm{DP}[0][0] = 0, \quad \mathrm{Used}[0][0][k] = 0, \ \forall k \in [n]$$

- The inductive update,

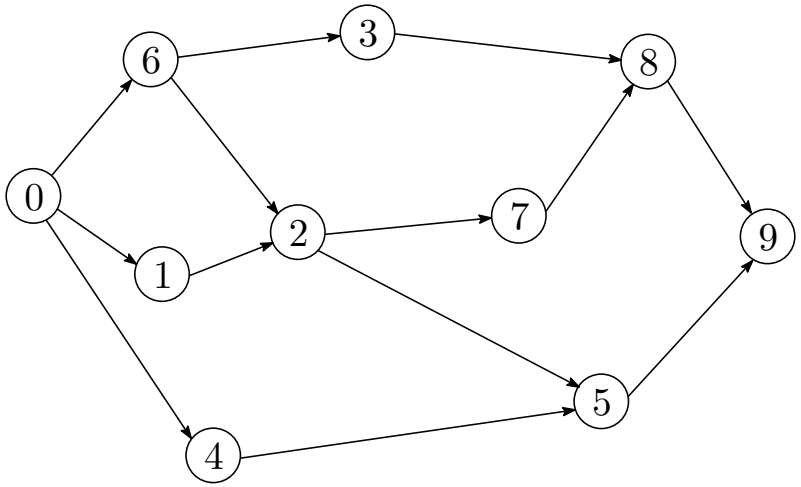
$$\mathrm{DP}[i][j] = \max_{k \in [n]} \left\{ \mathrm{DP}[i - w_k][j - w_k] + p_k \middle| i \geq w_k; j \geq c_k \right\}$$

Let
$k^* = \arg\max_{k \in [n]} \left\{ \mathrm{DP}[i - w_k][j - w_k] + p_k \middle| i \geq w_k; j \geq c_k \right\}$

$$\mathrm{Used}[i][j][k] = \begin{cases} \mathrm{Used}[i - w_k][j - w_k][k] & k \neq k^* \\ \mathrm{Used}[i - w_k][j - w_k][k] + 1 & k = k^* \end{cases}$$

Directed Graph

In a directed graph, we distinguish between edge (u, v) and edge (v, u)



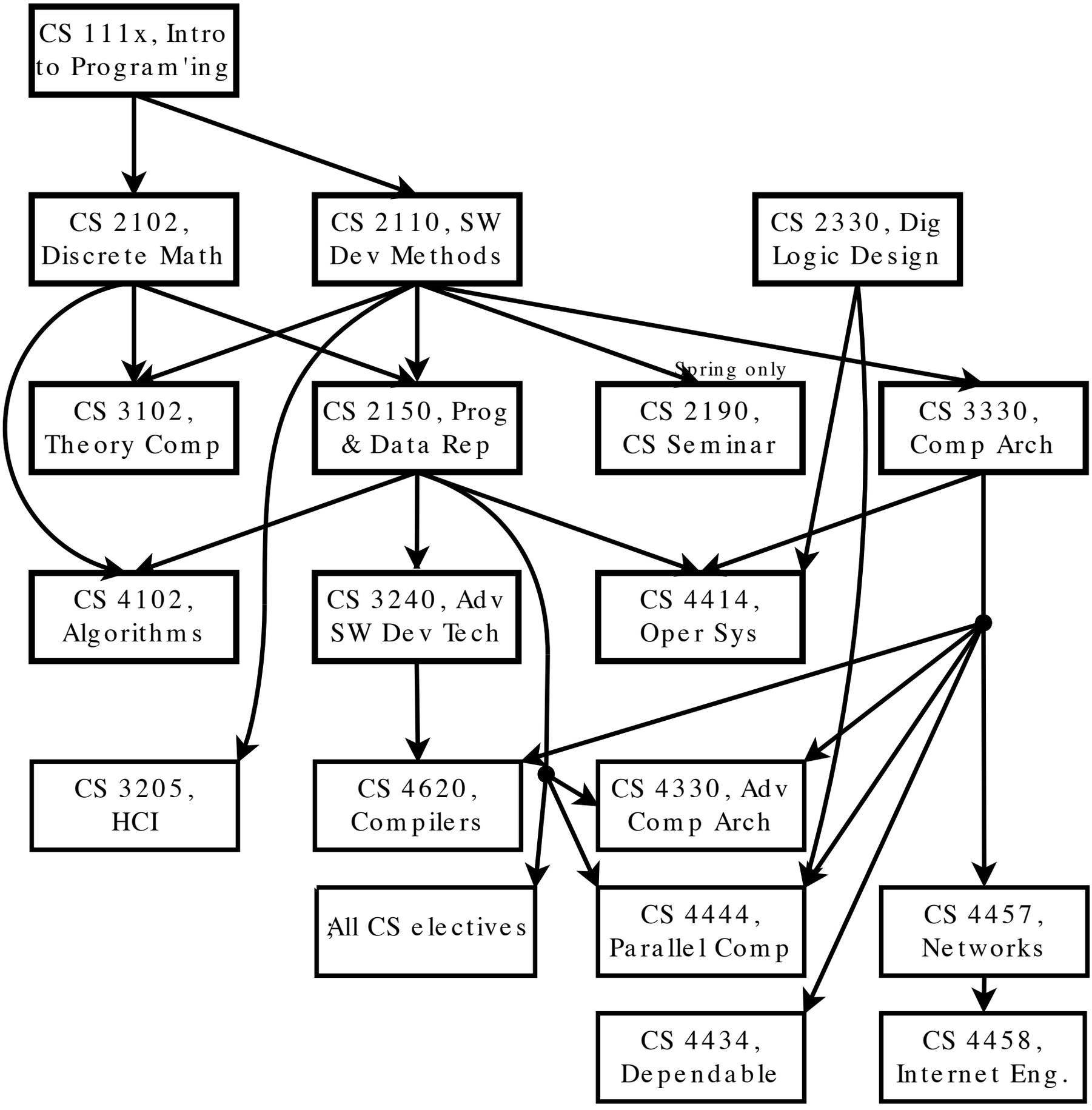
- out-degree of a vertex is the number of edges leaving it
- in-degree of a vertex is the number of edges entering it
- Each edge (u, v) contributes one to the out-degree of u and one to the in-degree of v

$$\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|$$

Usage of Directed Graph

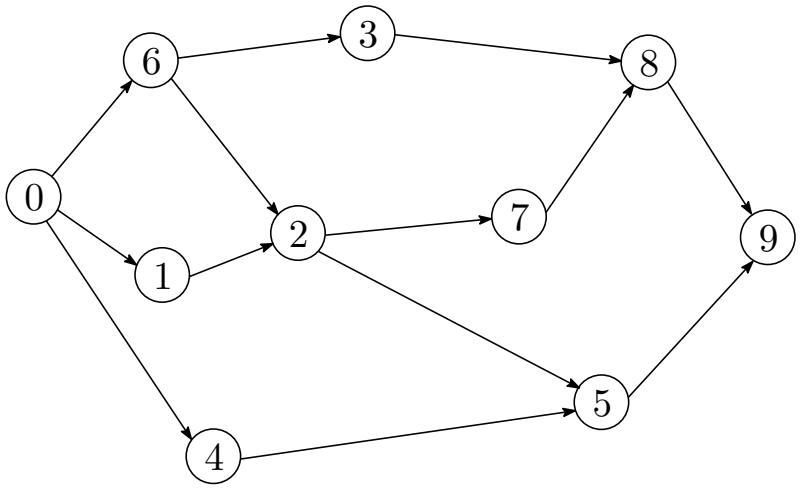
- Directed graphs are often used to represent order-dependent tasks
  - That is, we cannot start a task before another task finishes
- Edge (u, v) denotes that task v cannot start until task u is finished

- Clearly, for the system not to hang, the graph must be acyclic
  - It must be a directed acyclic graph (or DAG)


CS 111x, Intro to Program'ing
CS 2102, Discrete Math
CS 2110, SW Dev Methods
CS 2330, Dig Logic Design
Spring only
CS 3102, Theory Comp
CS 2150, Prog & Data Rep
CS 2190, CS Seminar
CS 3330, Comp Arch
CS 4102, Algorithms
CS 3240, Adv SW Dev Tech
CS 4414, Oper Sys
CS 3205, HCI
CS 4620, Compilers
CS 4330, Adv Comp Arch
All CS electives
CS 4444, Parallel Comp
CS 4457, Networks
CS 4434, Dependable
CS 4458, Internet Eng.

Topological Sort

- A Topological ordering of a graph is a linear ordering of the vertices of a DAG such that if (u, v) is in the graph, u appears before v in the linear ordering
- e.g., order in which classes can be taken



- Topological ordering may not be unique as there are many "equal" elements!
- E.G., there are several topological orderings
  - 0, 6, 1, 4, 3, 2, 5, 7, 8, 9
  - 0, 4, 1, 6, 2, 5, 3, 7, 8, 9
  - ...

Topological Sort Algorithm

- Observations
  - A DAG must contain at least one vertex with in-degree zero (why?)

- Algorithm: Topological Sort
  1. Output a vertex u with in-degree zero in current graph.
  2. Remove u and all edges (u, v) from current graph.
  3. If graph is not empty, goto step 1.

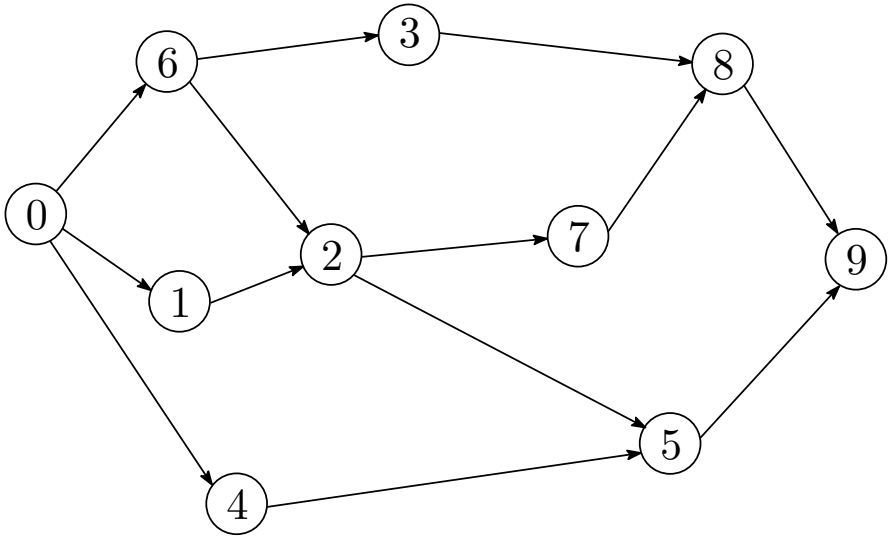
- Correctness
  - At every stage, current graph is a DAG (why?)
  - Because current graph is always a DAG, algorithm can always output some vertex. So algorithm outputs all vertices.
  - Suppose order output was **not** a topological order. Then there is some edge (u, v) such that v appears before u in the order. This is impossible, though, because v can not be output until edge (u, v) is removed!

Topological Sort Algorithm

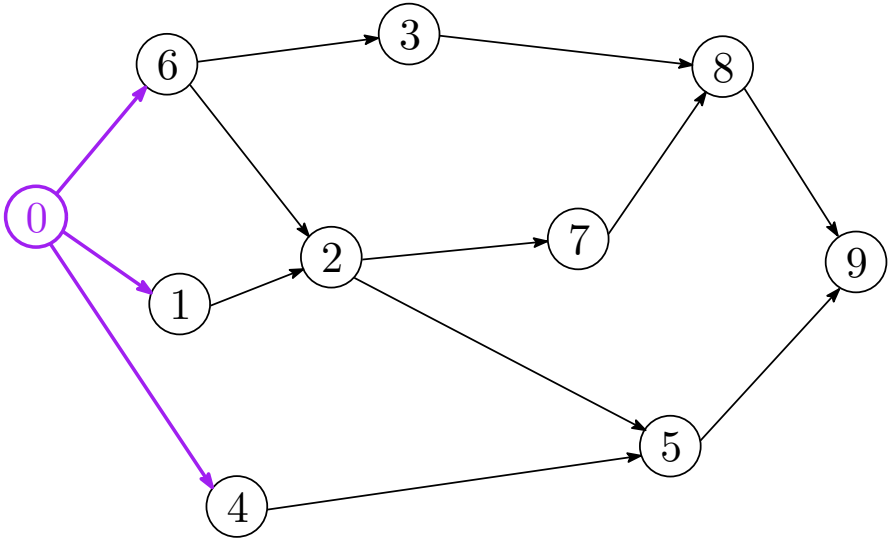
Topological_sort(G)

```
Initialize Q to be an empty queue;
foreach u in V do
    if in-degree(u) = 0 then
        // Find all starting vertices
        Enqueue(Q, u);
    end
end
while Q is not empty do
    u = Dequeue(Q);
    Output u;
    foreach v in Adj(u) do
        // remove u's outgoing edges
        in-degree(v) = in-degree(v) − 1;
        if in-degree(v) = 0 then
            Enqueue(Q, v);
        end
    end
end
```

Example

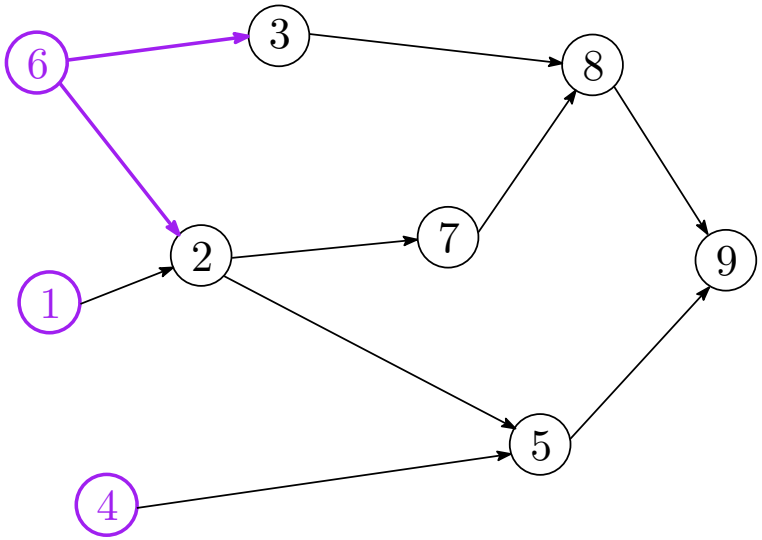


$Q = \{\}$



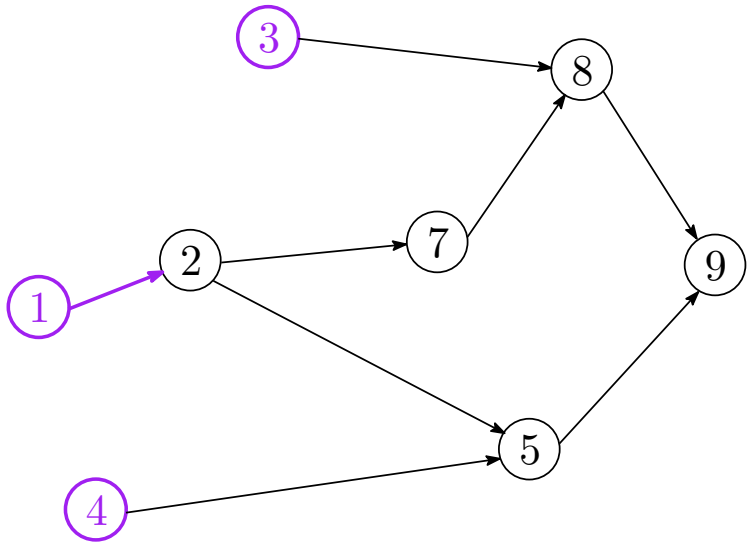
$Q = \{0\}$

Example



$Q = \{6, 1, 4\}$

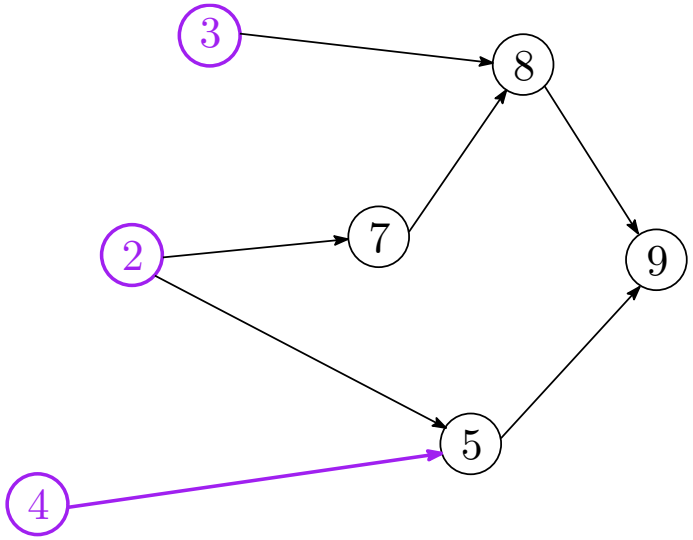
Output: 0



Q= {1, 4, 3}

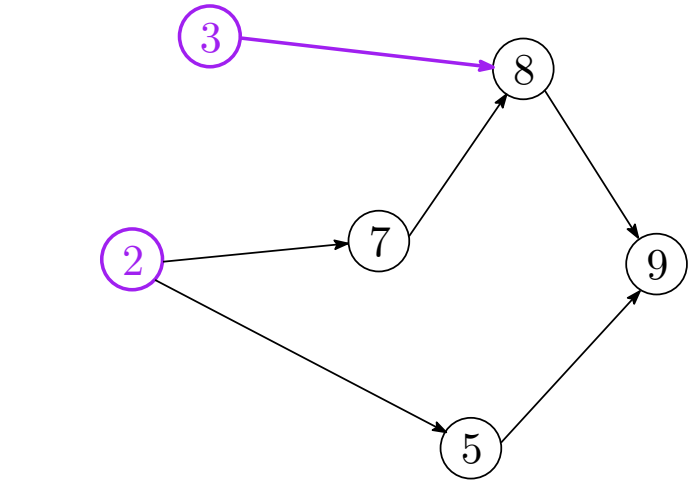
Output: 0, 6

Example



$Q = \{4, 3, 2\}$

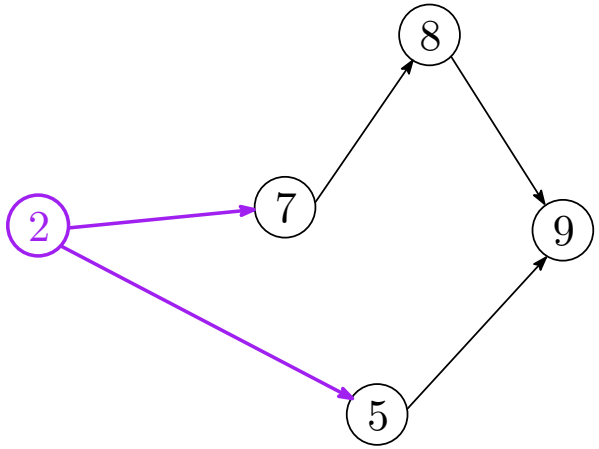
Output: 0, 6, 1



$Q = \{3, 2\}$

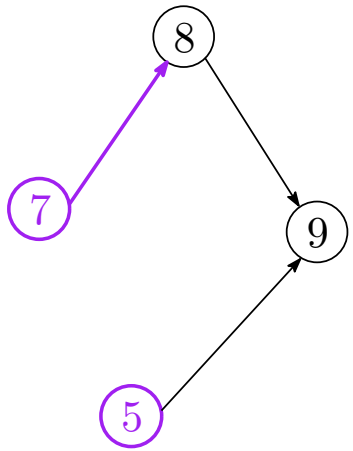
Output: 0, 6, 1, 4

Example



$Q = \{2\}$

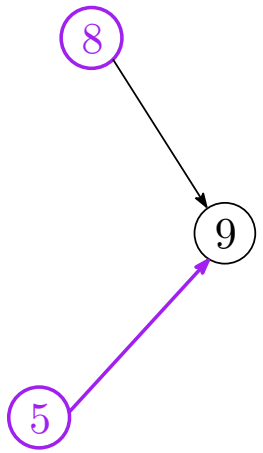
Output: 0, 6, 1, 4, 3



$Q = \{7, 5\}$

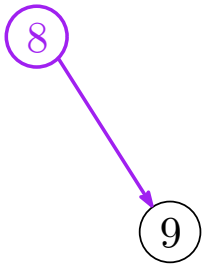
Output: 0, 6, 1, 4, 3, 2

Example



$Q = \{5, 8\}$

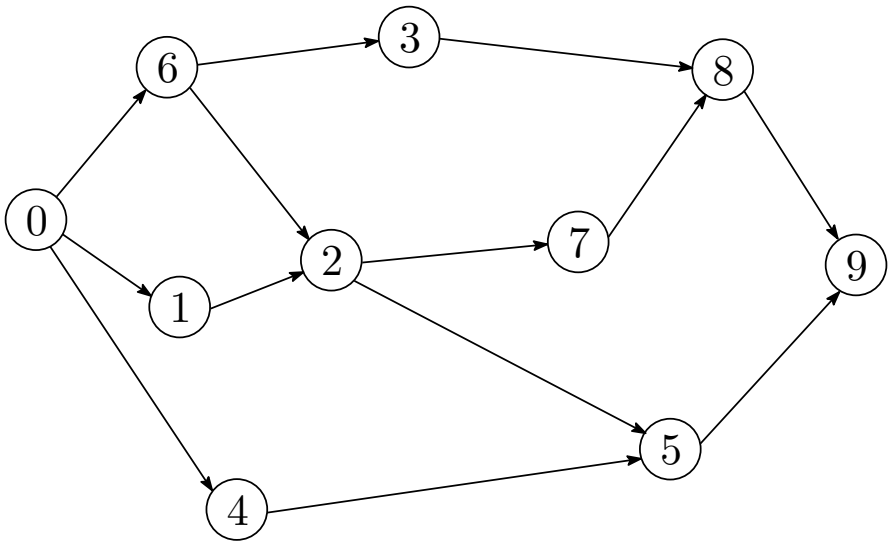
Output: 0, 6, 1, 4, 3, 2, 7



$Q = \{8\}$

Output: 0, 6, 1, 4, 3, 2, 7, 5

Example



9

$Q = \{9\}$

Output: 0, 6, 1, 4, 3, 2, 7, 5, 8

$Q = \{\}$

Output: 0, 6, 1, 4, 3, 2, 7, 5, 8, 9

Done!

Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
  - $\sum_{v \in V} \text{out-degree}(v) = E$
- Therefore, the running time is $O(V + E)$

Question

Can we use DFS to implement topological sort?