Tutorial 10 — Shortest Paths

CS2040C Semester 2 2018/2019

By Jin Zhe, adapted from slides by Ranald, AY1819 S2 Tutor

Shortest Path

SSSP

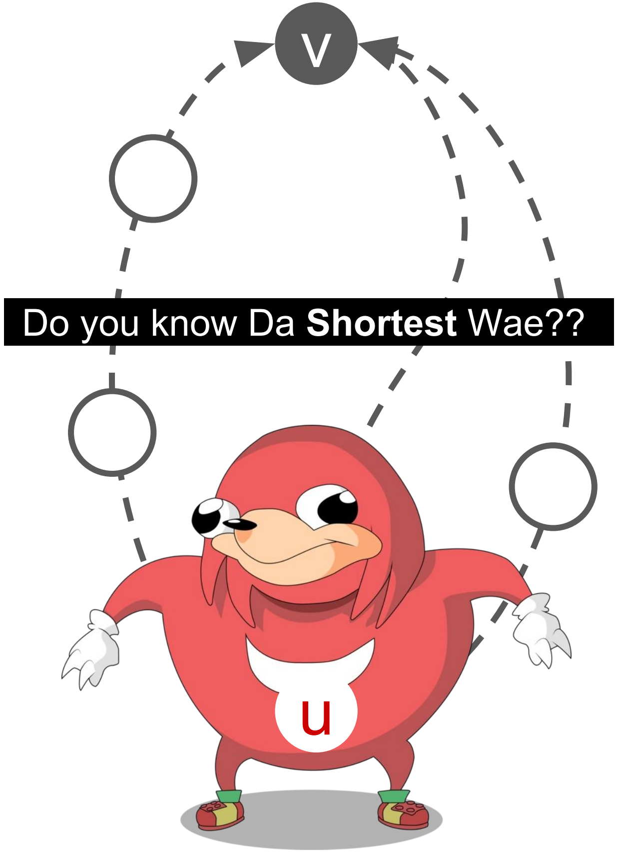
APSP

Shortest Path Problems

Given a graph with weighted/unweighted edges,

what is $\delta(u, v)$, the **shortest path** from a vertex u to another vertex v?

Shortest means the path with lowest "length".



Shortest Path Problems

The definition of “length” varies from problem to problem:

- [Most common] Sum of all edge weights in the path
- Number of edges in the path. Applies for unweighted graph or graph weight all edge weight equal
- Sum of all *vertex weights* in the path
- *Product* of all edge weights in the path
  - Only applicable if all weights are positive

Shortest Path Problems

In general, 2 main types of shortest path problems:

Single Source Shortest Path (SSSP)

- Shortest path from a *single* starting vertex s, to *every other vertex* in the graph

All Pairs Shortest Path (APSP)

- Shortest path between *every pair* of vertices in the graph
- Just run SSSP on each of the V vertices in graph!

SSSP Algorithms

SSSP Algorithms

Algorithm	Time complexity
Bellman Ford	$O(VE)$
Dijkstra	$O((V+E)\ log\ V)$
Modified Dijkstra	$\approx O((V+E)\ log\ V)$

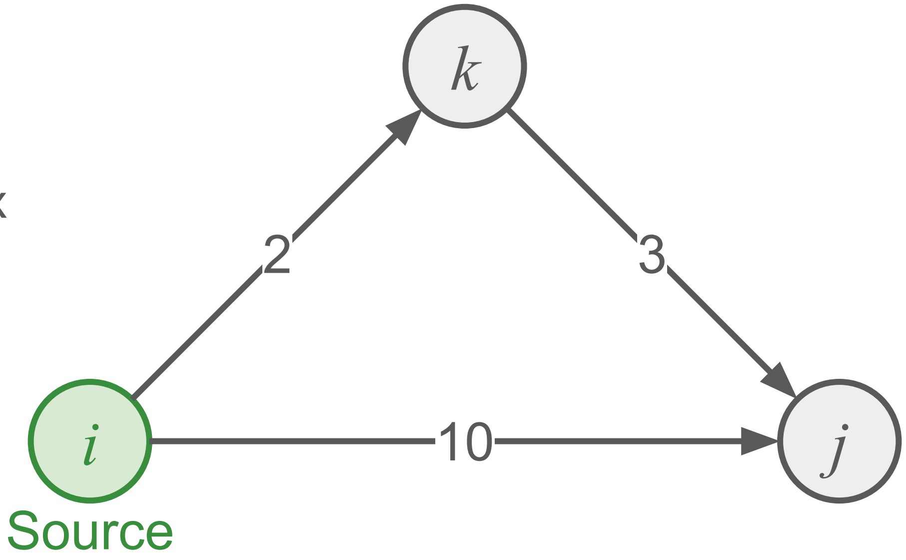
SSSP Algorithms—Main idea

- Modelled as an optimization problem
- Maintain a tentative “shortest distance” table D
- Incrementally work towards an optimal solution by “Relaxing” vertices
- At the end of the process, D converges to optimal solution and reflects the actual shortest path to every vertex from the given source vertex. i.e. $D=\delta$
- Different algorithms essentially carry out the relaxations in different order!

What is Relax?

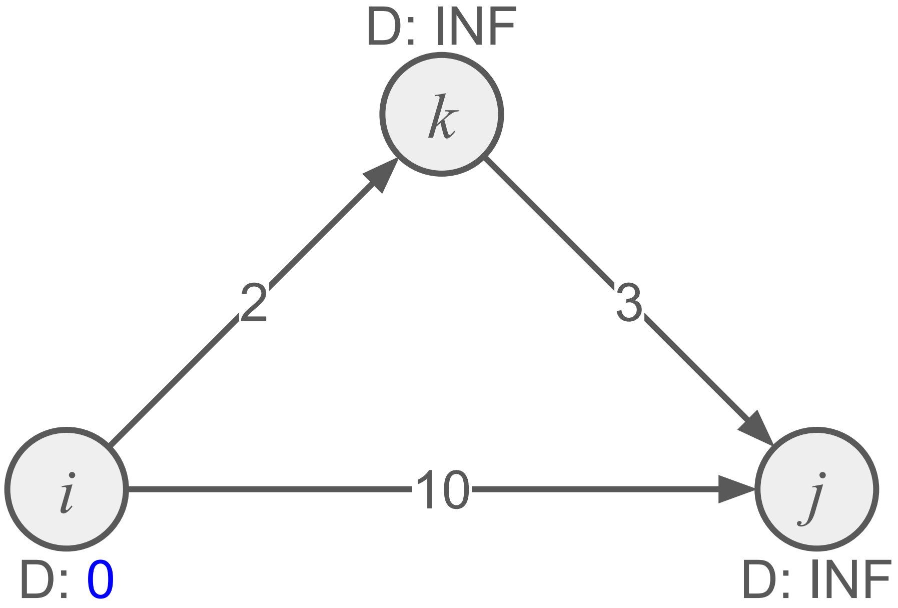
What is Relax?

Let's illustrate by running SSSP on this graph with the source being vertex i.



What is Relax?

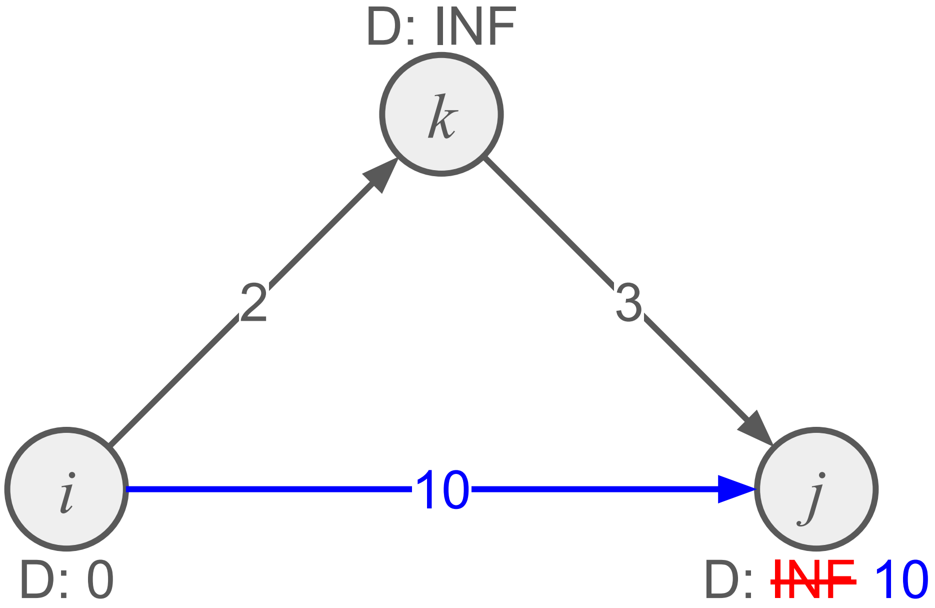
We initialize all distances to be infinity, with the exception of source vertex, which trivially has a distance of 0.



What is Relax?

We found a path from i to j with total weight 10 ($i \to j$), which is shorter than INF!

So we relax $i \to j$.

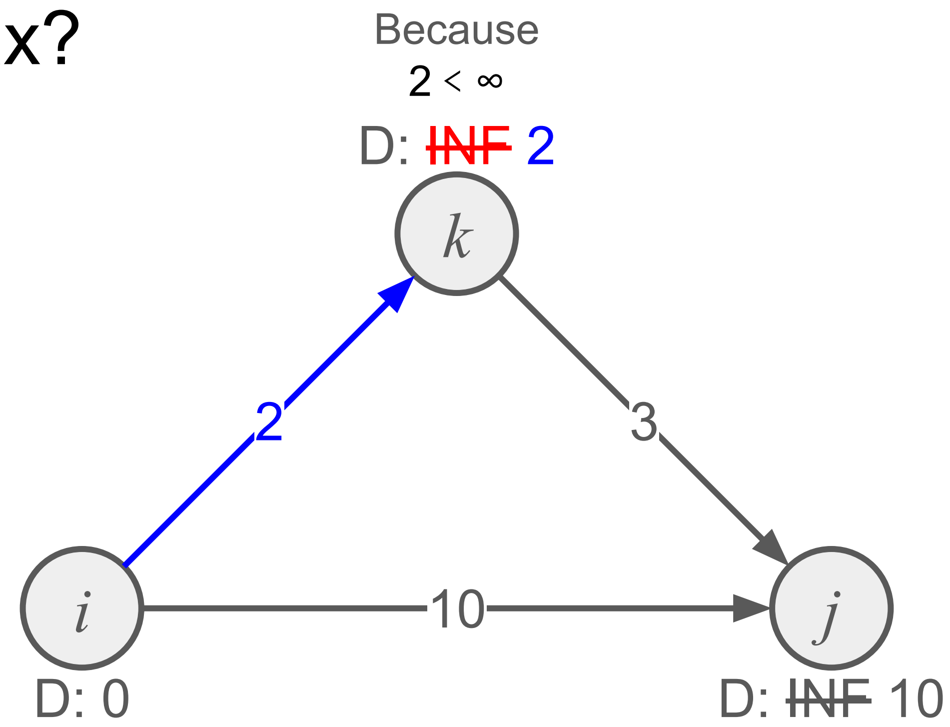


Because
$10 < \infty$

What is Relax?

We found a path from i to k with total weight 2 ($i \to k$), which is shorter than INF!

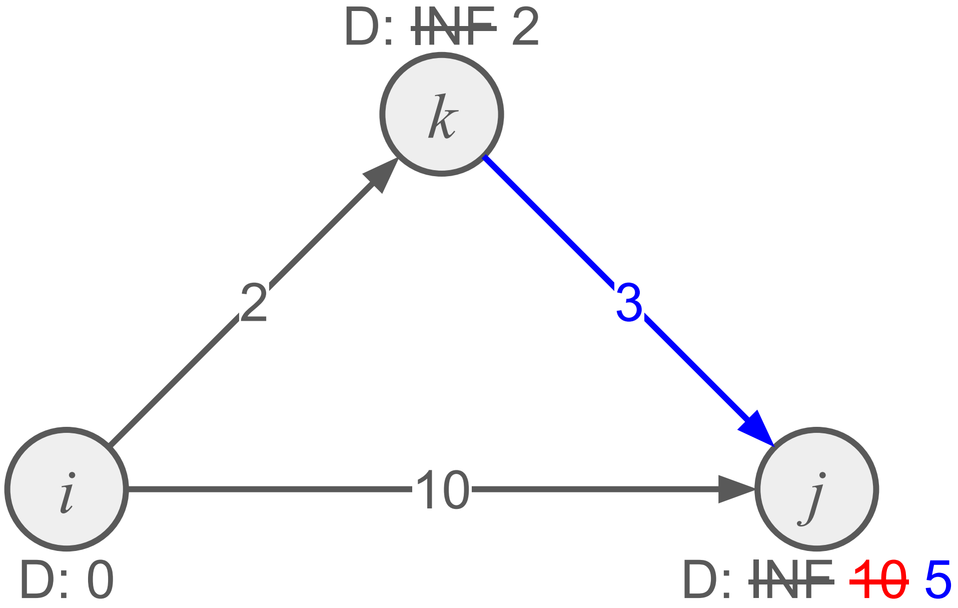
So we relax $i \to k$.



What is Relax?

We found a path from i to j with total weight 5 ($i \to k \to j$), which is shorter than 10!

So we relax $k \to j$.

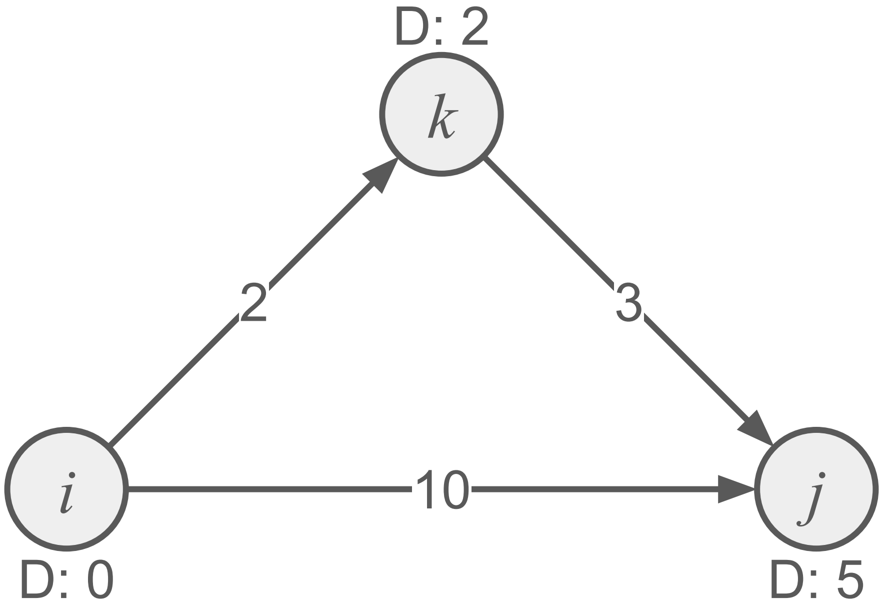


Because
$2+3=5 < 10$

What is Relax?

We are now done and `D` captures all the shortest path lengths from i!

Essentially we traversed each edge and updated the tentative 'shortest distance' every time.



What is Relax?

So formally, for a edge $u \to v$ with weight w, when we say "relax **edge** $u \to v$", we mean the following:

- $D[u] + w < D[v]$
- Update $D[v] \coloneqq D[u] + w$
- Shortest path to v is now shortest path to u followed by $u \to v$

We sometimes also refer to this as "relaxing **vertex** v with edge $u \to v$".

Bellman-Ford — Paying homage

Richard E. **Bellman**

Lester Randolph **Ford** Jr.

*"The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively. Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**."*

Source: Wikipedia

Image source: Wikipedia

Bellman-Ford — Algorithm

For a graph with V vertices and E edges:

1. For an edge $u \rightarrow v$ with weight w, relax v with the edge if possible
2. Carry out 1. across all E edges in the graph **in any order**
3. Repeat 1. and 2. $V - 1$ times

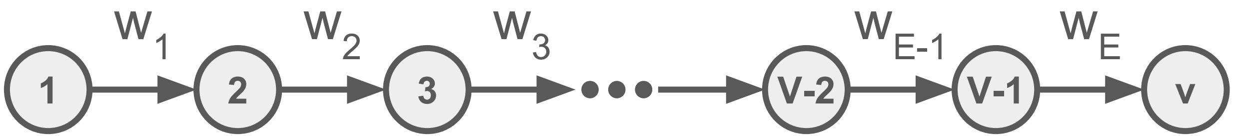
Bellman-Ford—Time complexity

Relaxing a vertex is $O(1)$

Iterating across all edges (1 full pass) takes $O(E)$

Carrying out $V-1$ full passes therefore incur a total time compexity of $O((V-1)\mathrm{E})=O(VE)$

Bellman-Ford — Why require $V-1$ full passes?

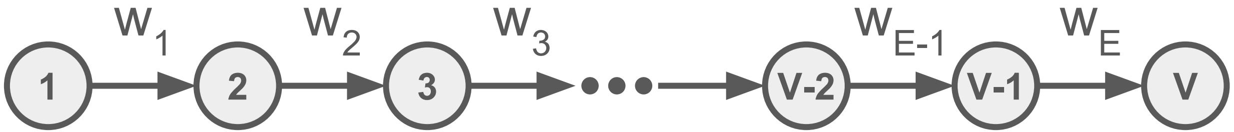


(u, v, w)
(V-1, V, w_E)
(V-2, V-1, w_{E-1})
...
(2, 3, w_2)
(1, 2, w_1)

Consider the Bellman-Ford killer graph above (worst case) and its Edge List on the right.

We run Bellman-Ford with source vertex 1 and iterate through each edge in the order defined in the edge list.

Bellman-Ford—Why require $V-1$ full passes?



- 1st pass: only managed to relax vertex 2
- 2nd pass: only managed to relax vertex 3
- And so on…
- Realize that in this graph, each vertex just need 1 relaxation to attain its optimal distance from source
- Therefore vertex 3's distance could only be determined on the $V-1^{\text{th}}$ pass

Note: The maximum of the minimum path length seperating 2 vertices in a graph of V vertices is $V-1$

(u, v, w)
$(V-1, V, w_E)$
$(V-2, V-1, w_{E-1})$
...
$(2, 3, w_2)$
$(1, 2, w_1)$

Bellman-Ford — Jin's informal proof of correctness

Realize that on every complete pass of the graph in Bellman-Ford's algorithm, we are guaranteed to have completely resolved the shortest distance of **at least 1** vertex. i.e

- 1st pass: At least one of source v_1's neighbours now have the correct shortest distance. Let's call that neighbour v_2
- 2nd pass: At least one of v_2's neighbours now have the correct shortest distance. Let's call that neighbour v_3
- And so on…

There are $V-1$ other vertices excluding the source. Thus after $V-1$ passes, we are guaranteed that all vertices now have their correct shortest distance! $Q.E.D$

Bellman-Ford — Worst case

So how might we construct a **worst-case** graph for Bellman-Ford?

We just need to ensure that we can only finalize the distance of **a single vertex** each time we conduct a complete pass of relaxations on the graph. This is ensured if we force the order of edges to be relaxed to be: farthest from source ⇒ nearest to source.

Look at Bellman-Ford killer graph again, can you now see how we came up with it?

Bellman-Ford—Best case

So how might we construct a **best-case** graph for Bellman-Ford?

Best case happens when we can converge all distances within a single pass of relaxations.

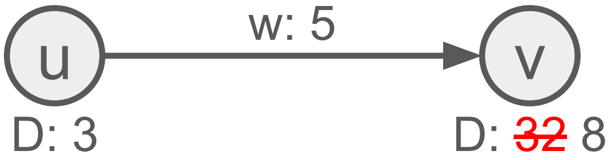
Look at the Bellman-Ford killer graph again. We can simply turn it into a best case by reversing the edge list!

This happens to be a special case because the graph is a DAG. When we have a DAG, we just need to relax edges in a **single pass** in the toposorted order starting from the source vertex!

Bellman-Ford — Implementation

```
vector<int> D(V, INF);               // Initialize all distances = INF
D[s] = 0;                            // s→s is trivially 0
for (int i = 0; i < V-1; i++){       // Iterate V-1 times
    for (auto &e: EL){               // Iterate across all edges
        /* Relax if new distance is shorter */
        int u = e.u, v = e.v, w = e.w;
        D[v] = min(D[v], D[u] + e.w);
    }
}
```

Example:



Adapted from materials for NOI 2015 Dec Training Team

What should *infinity* be?

What should *infinity* be?

If we are using integers, should we choose `INF` to be `INT_MAX`, which is $2^{31} - 1 = 2{,}147{,}483{,}647$?

What should *infinity* be?

If we are using integers, should we choose `INF` to be `INT_MAX`, which is $2^{31} - 1 = 2{,}147{,}483{,}647$?

Answer: No! If `D[u]` is `INT_MAX`, then the relax operation `D[v] = min(D[v], D[u] + w)` will result in **arithmetic overflow!**

`INT_MAX + 1` already results in `-2`^{31}

What should *infinity* be?

If we are using integers:

- Calculate an impossible value, based on the problem limits. (Eg: 10^9)
- Ensure `INF + max_edge_weight` fits into the data type
- Use `long long` if unsure!

What should *infinity* be?

Alternatively,

- We can use impossible ‘placeholder’ values such as -1 (less bug-prone)
- Need to handle such cases explicitly
- Does not work with negative values

Test yourself!

How can we optimize Bellman-Ford to stop further passes after results have converged?

Hint: How did we optimize bubble sort?

Test yourself!

How can we optimize Bellman-Ford to stop further passes after results have converged?

Hint: How did we optimize bubble sort?

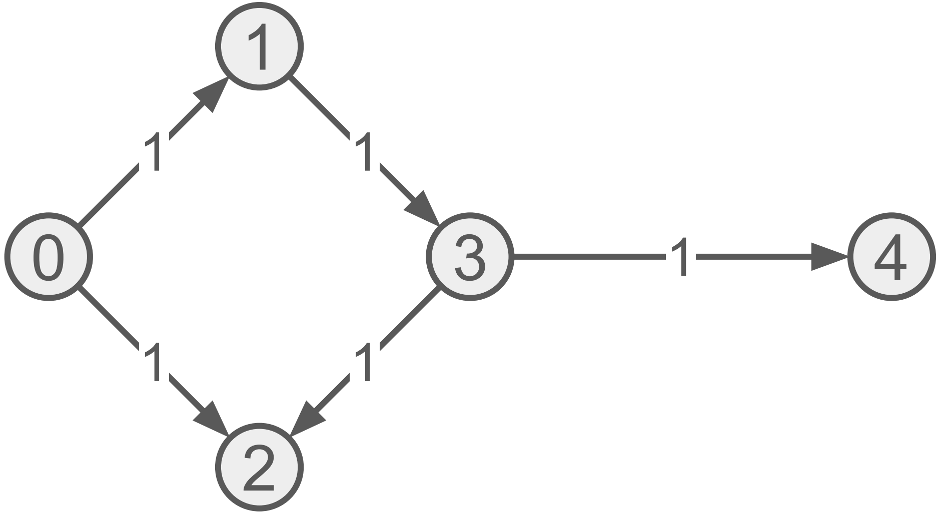
Answer: We stop whenever a full pass did not result in any relaxations.

SSSP—A special case

Before we continue, let's consider the special case of an unweighted graph or graph with all edge weights being equal.

How can we solve this?

Hint: You already know the algorithm



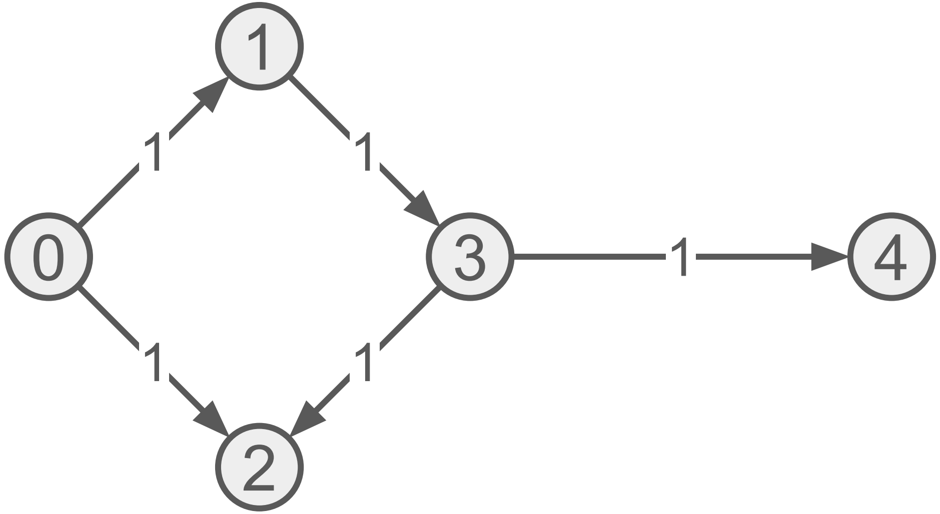
SSSP — A special case

Before we continue, let's consider the special case of an unweighted graph or graph with all edge weights being equal.

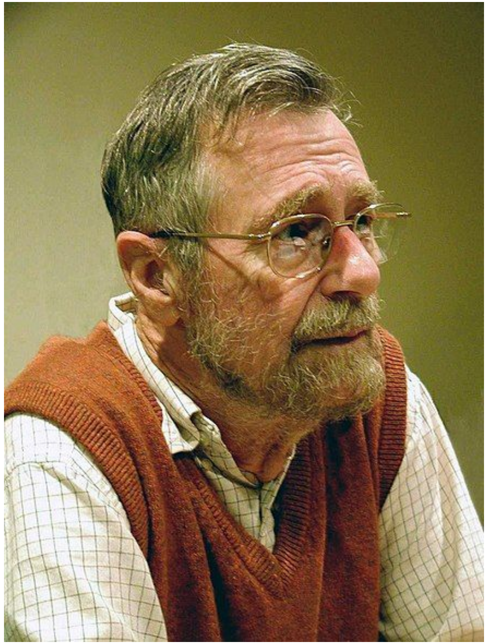
How can we solve this?

Answer: BFS!

Can we generalise this approach to the general weighted graph?



Dijkstra—Paying homage



Edsger Wybe **Dijkstra**
Source: Wikipedia

"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I ***designed in about twenty minutes****. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame."* Source: Wikipedia

How to pronounce Dijkstra? Click here to listen.

Dijkstra — Algorithm

1. Initialize distances of all vertices to be infinity, except the source vertex’s which is trivially 0
2. Mark all vertices to as “unresolved”
3. Greedily select the **unresolved** vertex u in the graph with the **least distance** so far
   a. Mark it as **resolved**
   b. For all its neighbours v, relax v with $u \to v$ if possible
4. Repeat 3. for as long as there are unresolved vertices in the graph

Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection?

$O(N\ log\ N)$ sort every time? That would take

$O(V\ log\ V) + O((V-1)\ log\ (V-1)) + \ldots + O(2\ log\ 2) + O(1\ log\ 1)$

$= O(V^2\ log\ V)$

Which is pretty bad! :O

Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection **in an efficient manner**?

Dijkstra—How to be greedy?

How can we greedily select the vertex with the least distance at every selection **in an efficient manner**?

Answer: We could use a data structure!

Dijkstra—How to be greedy?

Which ADT do we have that maintains ordered data and what data-structure(s) can be used to implement it?

Dijkstra—How to be greedy?

Which ADT do we have that maintains ordered data and what data-structure(s) can be used to implement it?

Answer: Priority Queue ADT!

Can be implemented via Heap or BBST.

Original Dijkstra — Implementation

```
typedef pair<int,int> ii;
vector<ii> AL; // Adjacency list of (vertex, weight) pairs
int V, source;
… // Read in AL, V, source

/* Declarations and initializations */
vector<bool> resolved(V, false); // Vertex u resolved → D[u] finalized
vector<int> D(V, INF);           // Distance table
D[source] = 0;                   // Assign source to distance 0
MinPriorityQueue pq();           // Instantiate custom Min-Priority Queue
for (int u=0; u<V; u++)          // Initialize pq with (distance, vertex)
    pq.enqueue({D[u], u});
```

Continued on next slide

Original Dijkstra — Implementation

```
while(!pq.empty()){
    ii curr = pq.dequeue();                    // Dequeue
    int D_u = curr.first, u = curr.second;    // D_u: distance, u: vertex
    resolved[u] = true;                        // Mark current as resolved
    for (auto &e: AL[u]) {                     // Iterate each edge of u
        int v = e.first, w = e.second;         // v: neighbour, w: e weight
        int D_v = D_u + w;                     // New distance to check
        /* If v not yet resolved and can be relaxed with u→v */
        if (!resolved[v] && D_v < D[v]){
            pq.update_key({D[v], v}, {D_v, v}); // Update key
            D[v] = D_v;                            // Relax u→v
        }
    }
}
```

Dijkstra — Implementation woes

Realize we would need to call `update_key(...)` whenever a vertex is relaxed.

If we don't have a custom implemented Min Priority Queue supporting that operation, then we're in a bit of trouble! This is because neither C++ nor Java Heap/BBST supports `update_key()`! So in practice, if using:

- Library BBST: Find vertex data with lowest distance, remove invalid vertex data and insert new vertex data with updated distance
- Library Heap: Lazy removal! :O

Self-read

Original Dijkstra — "Lazy removal" implementation

```
typedef pair<int,int> ii;
vector<ii> AL; // Adjacency list of (vertex, weight) pairs
int V, source;
... // Read in AL, V, source

/* Declarations and initializations */
vector<bool> resolved(V, false); // Vertex u resolved → D[u] finalized
vector<int> D(V, INF);           // Distance table
D[source] = 0;                   // Assign source to distance 0
priority_queue<ii, vector<ii>, greater<ii>> pq; // min-heap
for (int u=0; u<V; u++)          // Initialize pq with (distance, vertex)
    pq.push({D[u], u});
```

Continued on next slide

Adapted from materials for NOI 2015 Dec Training Team

Self-read

Original Dijkstra — "Lazy removal" implementation

```
while(!pq.empty()){
    ii curr = pq.top(); pq.pop();               // Dequeue
    int D_u = curr.first, u = curr.second;      // D_u: distance, u: vertex
    if (resolved[u]) continue;                  // Lazy removal method
    resolved[u] = true;                         // Mark current as resolved
    for (auto &e: AL[u]) {                      // Iterate each edge of u
        int v = e.first, w = e.second;          // v: neighbour, w: e weight
        int D_v = D_u + w;                      // New distance to check
        /* If v not yet resolved and can be relaxed with u→v */
        if (!resolved[v] && D_v < D[v]){
            pq.push({D_v, v});                  // Push updated dist into pq
            D[v] = D_v;                         // Relax u→v
        }
    }
}
```

Adapted from materials for NOI 2015 Dec Training Team

Original Dijkstra—Looks familiar?

Does the main procedure (after initialization and declarations) remind you of an algorithm you’ve seen before?

Original Dijkstra—Looks familiar?

Does the main procedure (after initialization and declarations) remind you of an algorithm you’ve seen before?

Answer: BFS!

Original Dijkstra—Looks familiar?

Realize that Dijkstra's algorithm is indeed just a few simple modifications of BFS:

1. Swapping out the Queue in BFS with a Min-Priority Queue
2. Visitation table is just resolved table
3. Update a neighbour's distance if it can be relaxed via the edge

It's no wonder that Dijkstra could come up the algorithm in just 20 minutes, while out having coffee, and tired from shopping with his fiancée. A moment's boredom that led to a lifetime's glory! :O

Original Dijkstra—Time complexity

Each time we dequeue a vertex from the min-heap, we incur $O(log\ V)$ time. Since the algorithm terminates when the min-heap is empty, the total time complexity due to dequeuing every vertex from the heap is $O(V\ log\ V)$

Each time we dequeue a vertex, we will potentially relax all its edges. When we relax an edge, we have to call `update_key` in the heap which incurs $O(log\ V)$ time. Since we will potentially relax all E edges in the graph when the algorithm terminates, the total time complexity due to all relaxations is $O(E\ log\ V)$.

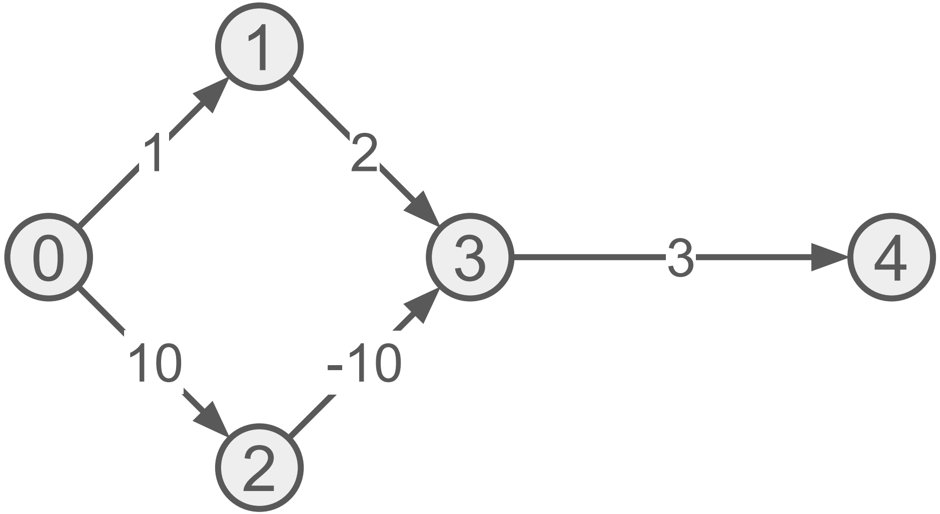
Hence total time complexity is $O(V\ log\ V) + O(E\ log\ V) = O((V+E)\ log\ V)$

Original Dijkstra — A problem

What happens when we run original Dijkstra on the following graph that has a negative weighted edge?

It’s a good exercise to work it out by hand!

Verify on VisuAlgo using Example Graph “CP3 4.18 -ve weight”



Modified Dijkstra

We observe that each dequeued vertex must be allowed to be relaxed again in the future, so we need to make the following changes:

- Do away with resolved table. i.e. until the algorithm has terminated, we cannot determine if a vertex's distance is finalized
- Enqueue a neighbour v whenever it can be relaxed with the current edge, regardless if v has been visited before
- No longer need to enqueue all distances at the beginning since we have a new propagation criteria
- Lazy removal of invalid vertex distances in queue. Invalid when distance table has a lower value

Modified Dijkstra – Implementation

```
typedef pair<int,int> ii;
vector<ii> AL; // Adjacency list of (vertex, weight) pairs
int V, source;
… // Read in AL, V, source

/* Declarations and initializations */
vector<int> D(V, INF);                // Distance table
D[source] = 0;                        // Assign source to distance 0
priority_queue<ii, vector<ii>, greater<ii>> pq; // min-heap
pq.push({0, source});                 // Enqueue source only
```

Continued on next slide

Modified Dijkstra – Implementation

```
while(!pq.empty()){
    ii curr = pq.top(); pq.pop();                // Dequeue
    int D_u = curr.first, u = curr.second;       // D_u: distance, u: vertex
    if (D_u > D[u]) continue;                    // Lazy: ignore outdated ones
    for (auto &e: AL[u]) {                       // Iterate each edge of u
        int v = e.first, w = e.second;           // v: neighbour, w: e weight
        int D_v = D_u + w;                       // New distance to check
        if (D_v < D[v]){                         // If can relax, then relax
            pq.push({D_v, v});                   // Push updated dist into pq
            D[v] = D_v;                          // Relax u→v
        }
    }
}
```

Test yourself!

In our algorithms presented for Bellman-Ford and (un)modified Dijkstra, we only obtained the distances of the shortest paths.

What if we also want the actual shortest **path taken**, from source to every vertex in the graph? How might we modify the algorithms?

Test yourself!

In our algorithms presented for Bellman-Ford and (un)modified Dijkstra, we only obtained the distances of the shortest paths.

What if we also want the actual shortest **path taken**, from source to every vertex in the graph? How might we modify the algorithms?

Answer: Maintain a parent/predecessor/previous table P that also gets updated during every vertex's relax operation

Facebook Privacy Setting

CS2010 Finals AY2013/2014 Sem 1

$O(V+E)$ solution

$O(k)$ solution

Problem statement

- You have a friendship graph of V vertices, E edges
- You are given a pair of vertices i and j.
  - Compute whether vertex i and j are *at most* 2 edges apart. i.e. degree of separation is 2
- Given: The friend list of profile i is stored in `adjList[i]` and sorted ascending
- $O(V+E)$ for 7 marks
- $O(k)$ for 19 marks
  - k is sum of number of adjacent vertices of i and j

How to get $O(V+E)$ solution?

Bellman Ford?

- $O(VE)$

Dijkstra?

- $O((V+E)\ log\ V)$

$O(V+E)$ solution

Breadth First Search!

- Start from vertex i, compute shortest path from i to every other vertex
- Check if $\text{distance}(i, j) \leq 2$

Can we do better?

Observation

We only need $\text{distance}(i, j) \leq 2$

3 cases:

1. $\text{distance}(i, j) = 0$ $\quad$ $i = j$
2. $\text{distance}(i, j) = 1$ $\quad$ j is a neighbour of i
3. $\text{distance}(i, j) = 2$ $\quad$ j is a neighbour of a neighbour of i

Case 1: $\text{distance}(i, j) = 0$

This is trivial, just $O(1)$ to check if $i = j$

Case 2: $\text{distance}(i, j) = 1$

To check if j is a neighbour of i, we search for j in each of i's k_i neighbours.

Since given that friends list is sorted, binary search will take $O(log\ k_i)$

Is j within green?

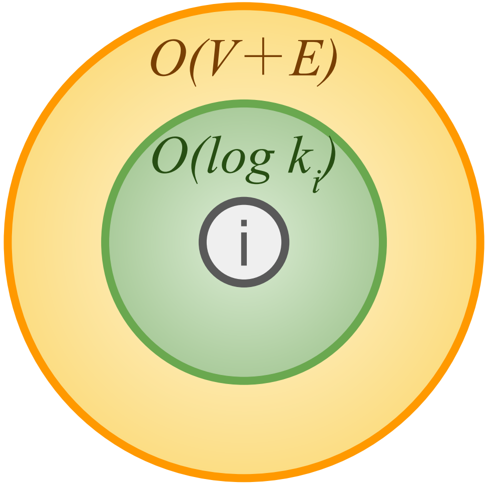
Green: $\text{distance} = 1$

Case 3: $\text{distance}(i, j) = 2$

To check if j is a neighbour of a neighbour of i, we potentially traverse the entire graph and so this will take $O(V+E)$.

Can we do better than this?

Is j within orange?



Green: $\text{distance} = 1$
Orange: $\text{distance} = 2$

A relevant question

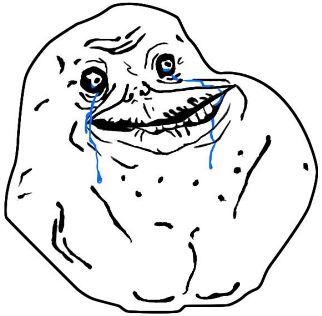
You are in **NUS** now. (West)

Your partner just touched down at **Changi Airport**. (East)

What should y'all do in order to meet each other in the shortest possible time?

A relevant question

A. Meet at NUS
B. Meet at Changi
C. Meet somewhere in the middle
D. Use video call
E. This is a scam. I don't have a partner

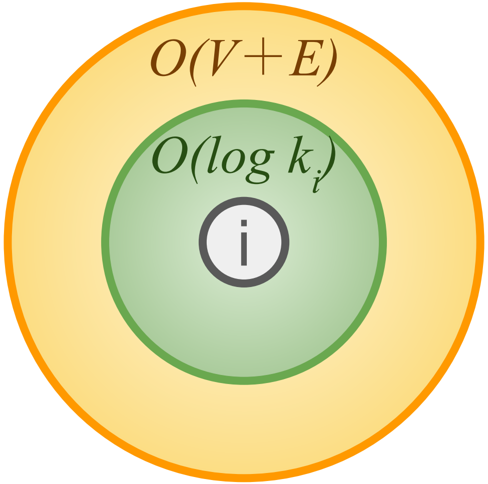


A relevant question

A. Meet at NUS
B. Meet at Changi
C. Meet somewhere in the middle
D. Use video call
E. This is a scam. I don’t have a partner

Case 3: $\text{distance}(i, j) = 2$

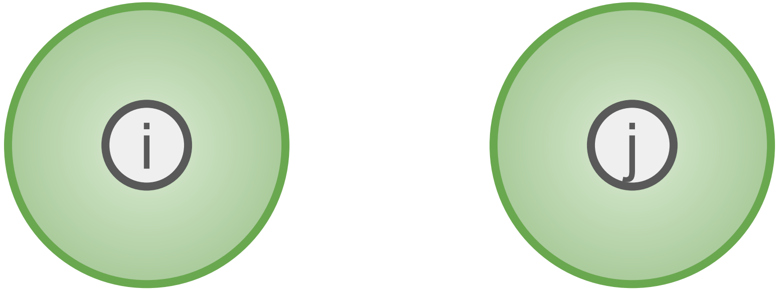
Is your partner in orange?



Green: $\text{distance} = 1$
Orange: $\text{distance} = 2$

Case 3: $\text{distance}(i, j) = 2$

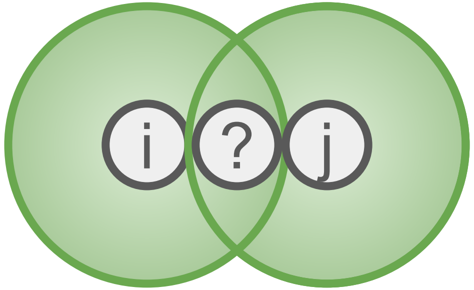
If vertex i and j are distance 2 away from each other, what do you realize?



Case 3: $\mathrm{distance}(i, j) = 2$

If vertex i and j are distance 2 away from each other, what do you realize?

Answer: There must be at least one common friend!



Transformed Problem

Given 2 integer arrays with sizes up to N each, find whether there exist an integer that is in both arrays.

Approaches:

1. For every number in one array, search for a correspondence in the other array $O(N^2)$
2. Sort both arrays $O(N \log N)$, then iterate through both arrays with 2 pointers to look for a corresponding pair of numbers $O(N)$

Facebook Privacy Setting

Realize that the transformed problem is essentially what we want to solve in the original problem!

Solution 2 (previous slide) would just take $O(k)$ in the original problem since we are already given that the friends list is sorted in the adjacency list! We just need to iterate down the friends list of i and j once!

Moral of the Story

Don’t judge a book by its cover!

What might appear like a Graph question, might not actually be a Graph question.

What might appear not a Graph question, might actually be a Graph question.

Online Quiz

General Instruction

Hard Problems

Online Quiz

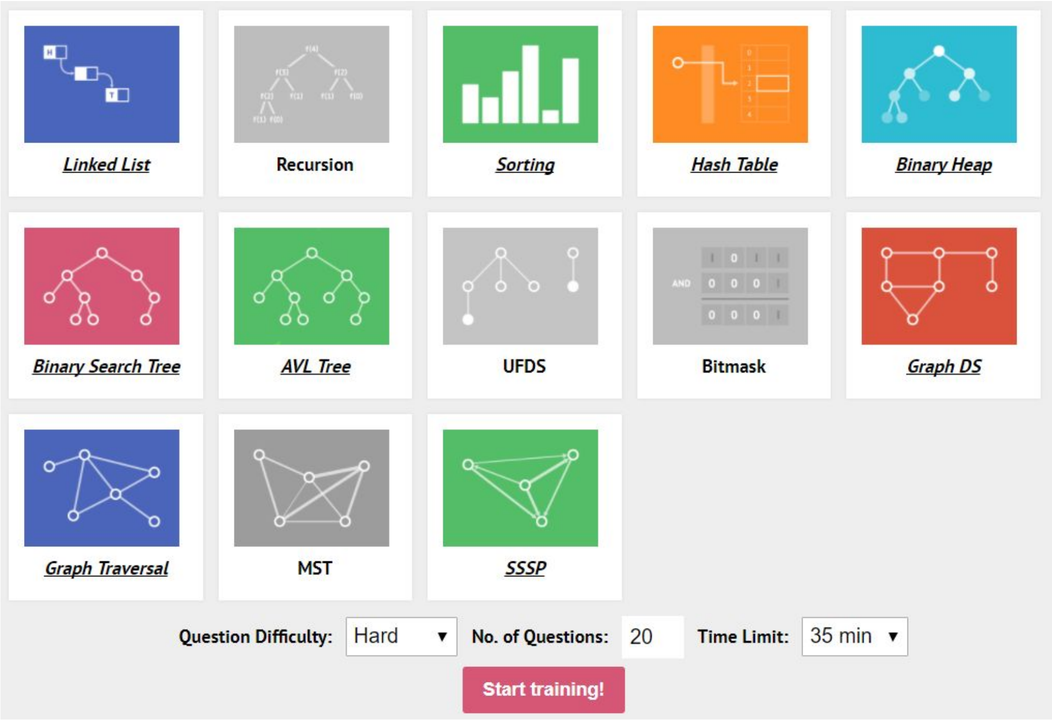
- Only one browser window: VisuAlgo
    - No google/yahoo/bing/baidu/duckduckgo/stackoverflow
    - No alt-tab to another VisuAlgo
    - No virtual or physical communication
    - Full screen mode
- Open book, **must use programming lab comp**
- Tip: Bring some pencil and paper to do rough working
    - Steven’s protip: Being transparent plastic and erasable marker

See what I did there?

Online Quiz

- Lab TA will click ‘Start’ for you
- Up to 2 attempts
- Take score of the **last** attempt
  - E.g. First attempt 18, second attempt 15 → score 15
  - E.g. First attempt 10, second attempt 18 → score 18

Quiz Configuration



DAG questions

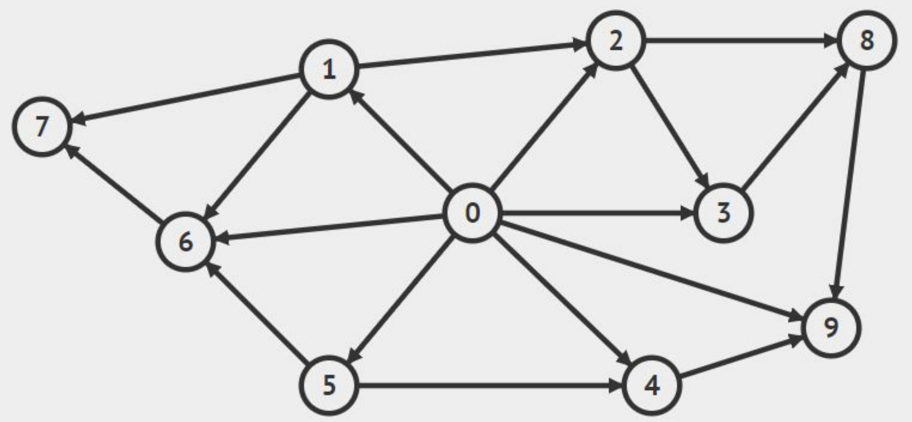
Select the topo sort

“Manual topological sort” / Kahn’s Algorithm

1. In any order, identify vertices with no *incoming edges*
2. Add them to toposort
3. Remove them and all the edges they have
4. Repeat steps 1-3 until no more vertices left in graph

DAG questions

3. Click the sequence of vertices such that when all the outgoing edges of these vertices are relaxed in this order using One-Pass Bellman-Ford's algorithm, the SSSP problem can be solved in O(V+E) time.

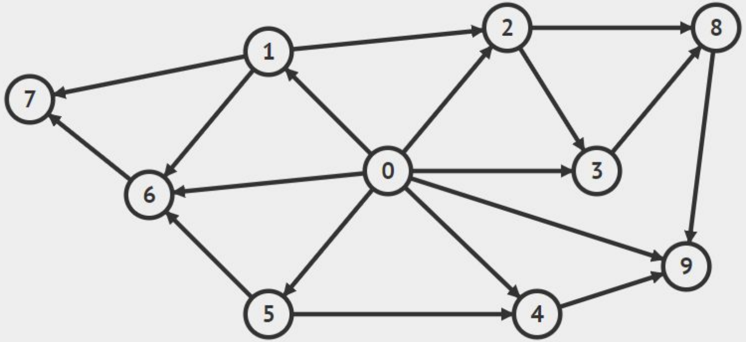


This question is indirectly asking you for the topological order:

[0, 1, 5, 6, 7, 2, 3, 4, 8, 9]

DAG questions

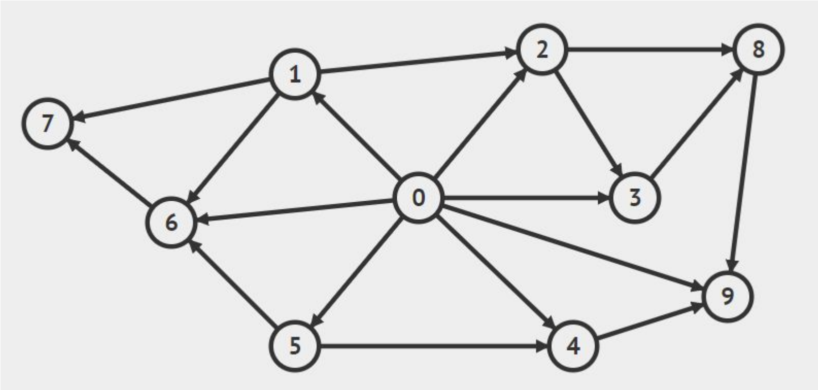
4. In the following Directed Acyclic Graph, how many simple paths are there from vertex **0** to **9**?



DAG questions

- Do in topological or reverse topological order
- Source/Destination Vertex

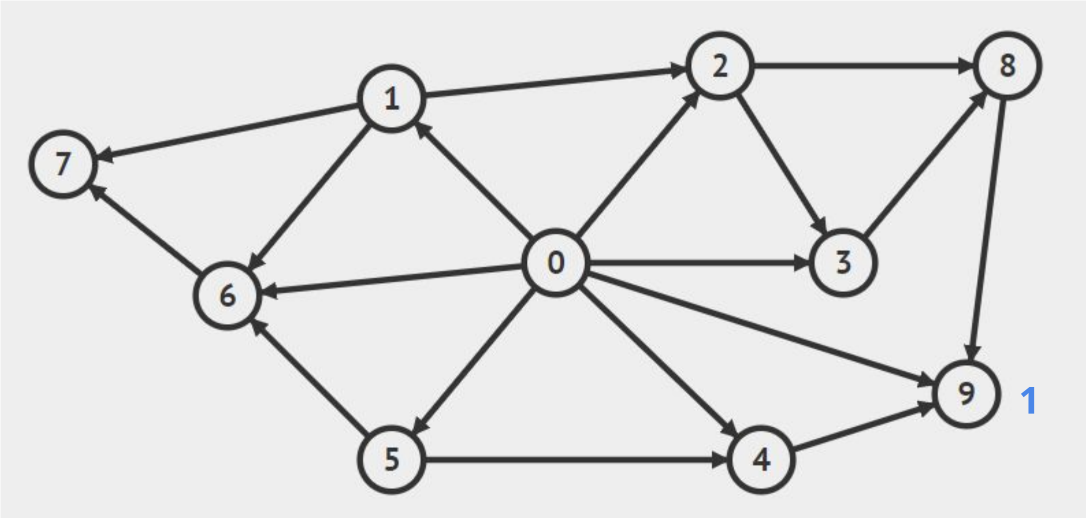
[0, 1, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

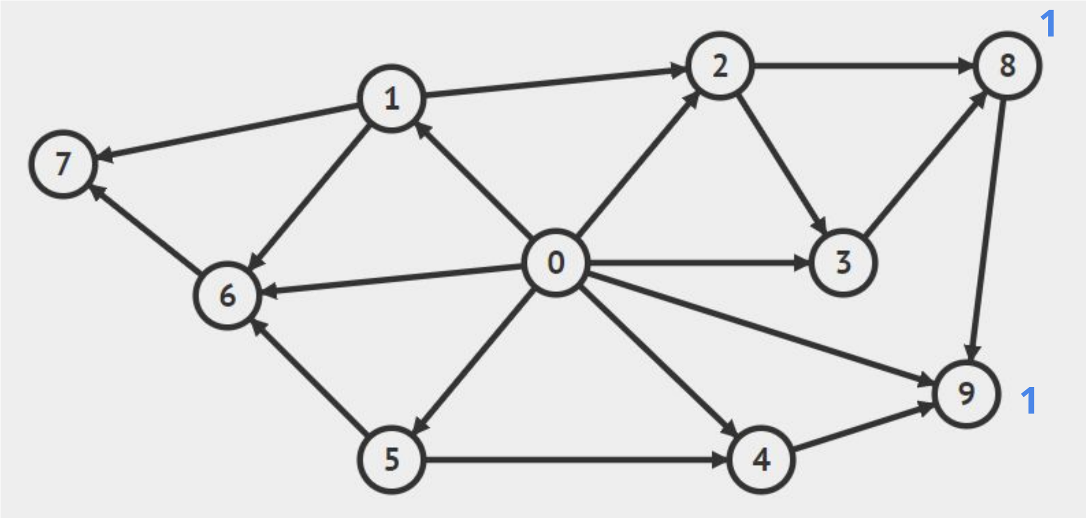
[0, 1, 5, 6, 7, 2, 3, 4, 8, **9**]



DAG questions

To count number of paths, we start at dest vertex

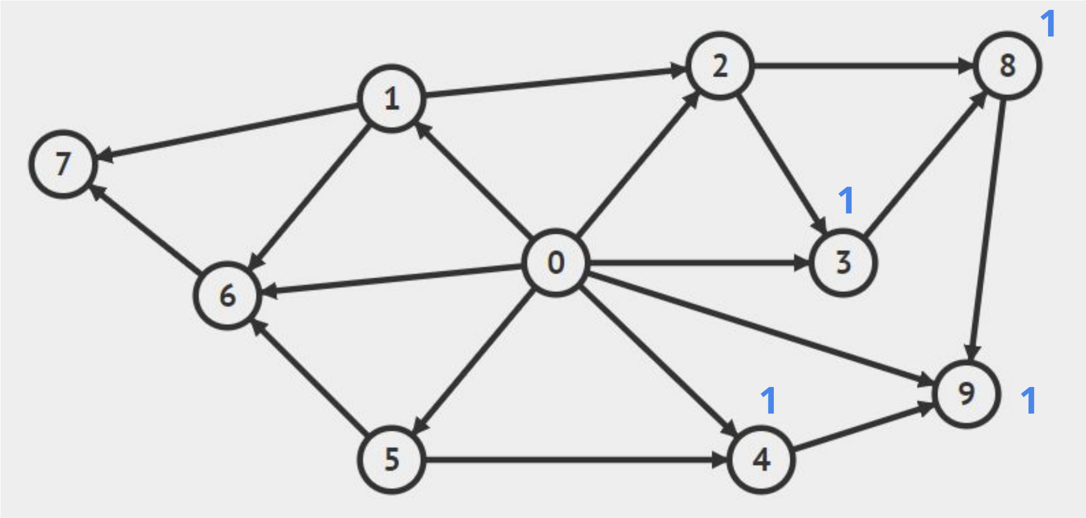
[0, 1, 5, 6, 7, 2, 3, 4, **8**, 9]



DAG questions

To count number of paths, we start at dest vertex

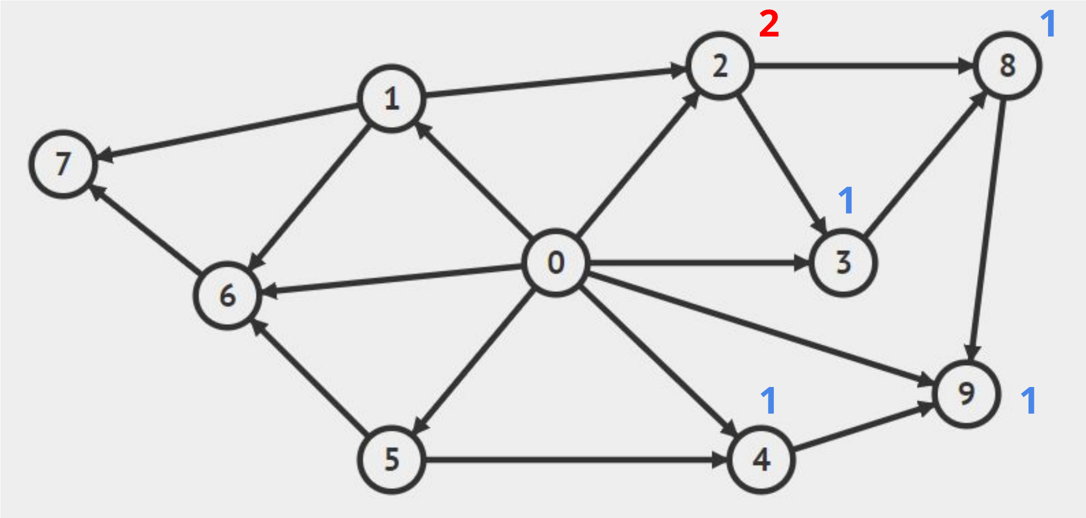
[0, 1, 5, 6, 7, 2, **3, 4**, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

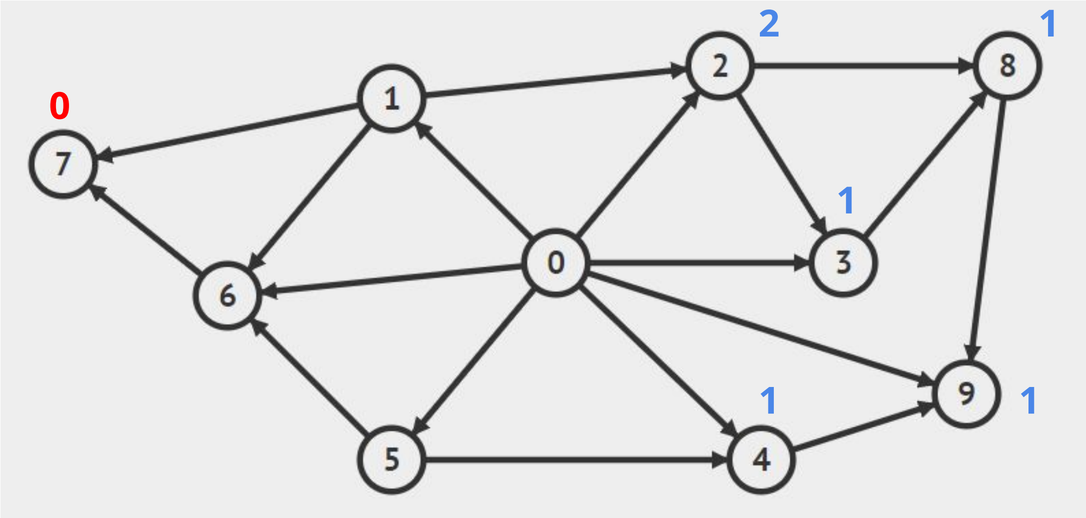
[0, 1, 5, 6, 7, **2**, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

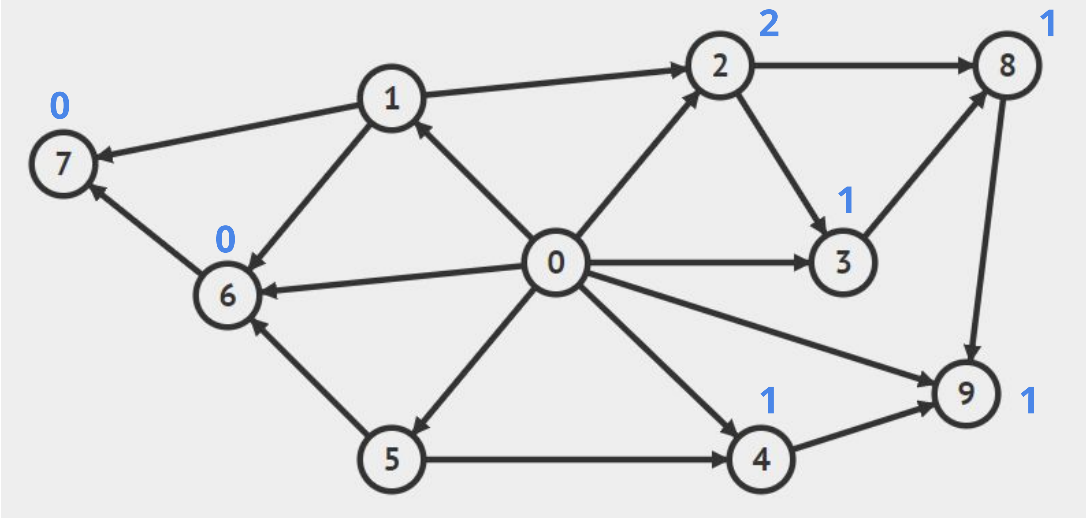
[0, 1, 5, 6, **7**, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

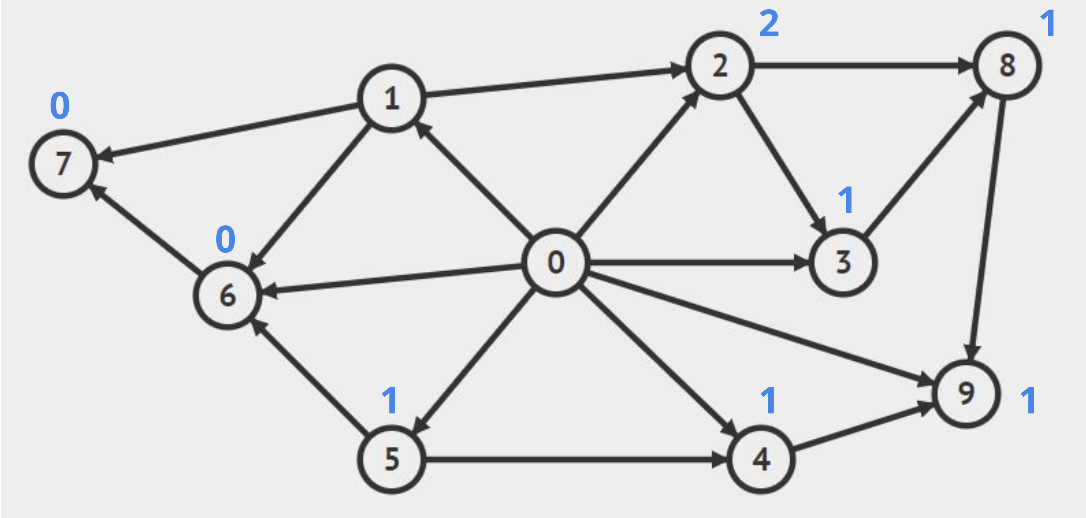
[0, 1, 5, **6**, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

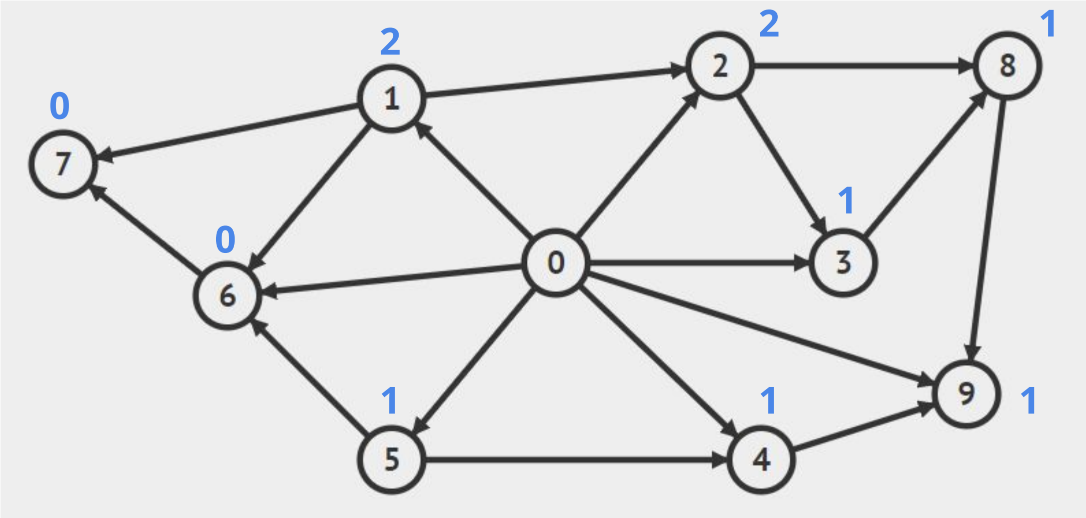
[0, 1, **5**, 6, 7, 2, 3, 4, 8, 9]



DAG questions

To count number of paths, we start at dest vertex

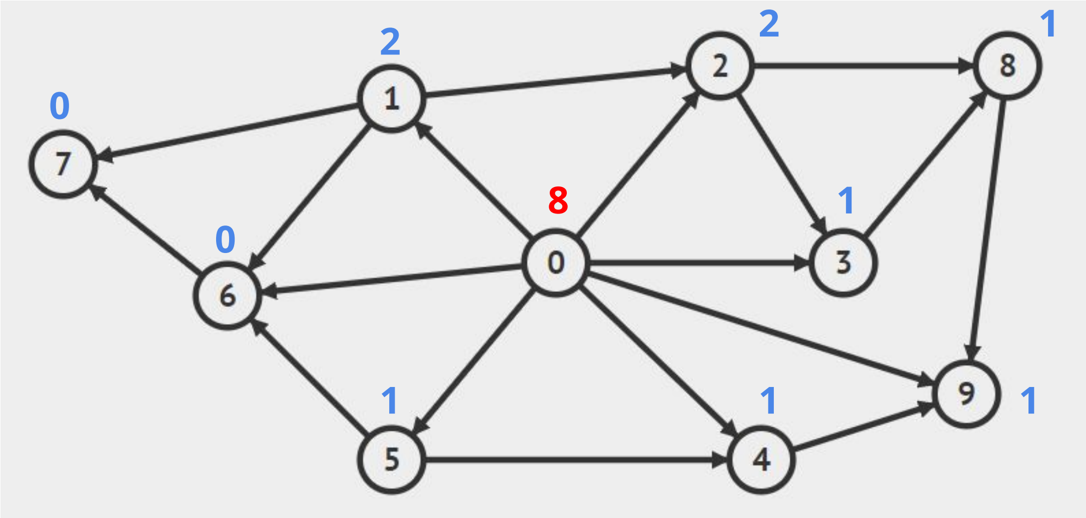
[0, **1**, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

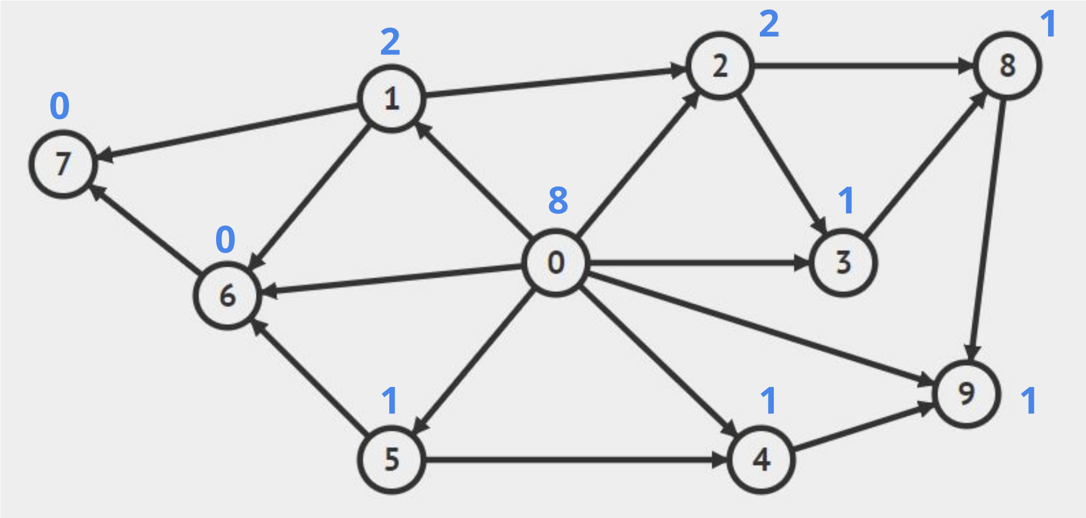
To count number of paths, we start at dest vertex

[**0**, 1, 5, 6, 7, 2, 3, 4, 8, 9]



DAG questions

1. Process in reverse topo order
2. Each vertex = sum of vertices of outgoing edges



Simulation Questions

17. You are given the following graph and source vertex **1**.
After **1** pass(es) of the Bellman-Ford's algorithm, what is the value of **D[6]**?
The order of edges is: **(0 , 1)** , **(0 , 4)** , **(0 , 6)** , **(0 , 7)** , **(1 , 2)** , **(2 , 1)** , **(2 , 3)** , **(2 , 5)** , **(3 , 0)** , **(4 , 2)** , **(5 , 0)** , **(5 , 2)** , **(6 , 0)** , **(7 , 0)**.
If **D[6]** is INFINITY, select 'No Answer'.
Note that all edge weights are printed closer to the arrowheads of the respective arrows.

Sadly, no fast way to do so.

>.<

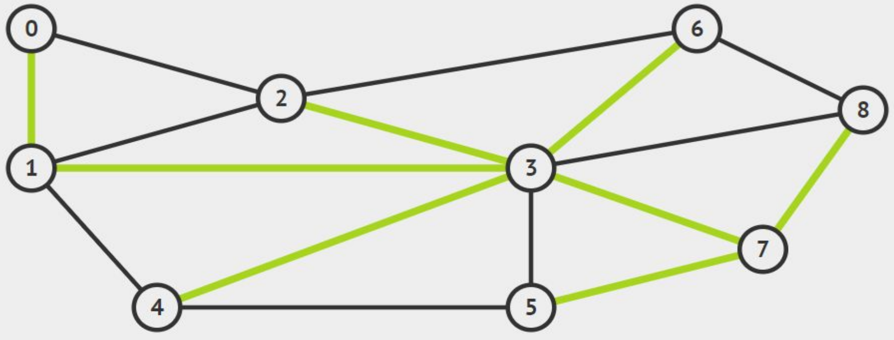
If the source is ‘INFINITY’ you can ignore the edge.

Spanning Tree

9. Click all the edges that make up the spanning tree induced by **BFS** starting from source vertex **7** for this graph.
The neighbours of a vertex are listed in ascending vertex number.
Please select the edges in the order that they are processed.

No answer

Your answer is: **(3,7),(5,7),(7,8),(1,3),(2,3),(3,4),(3,6),(0,1)**



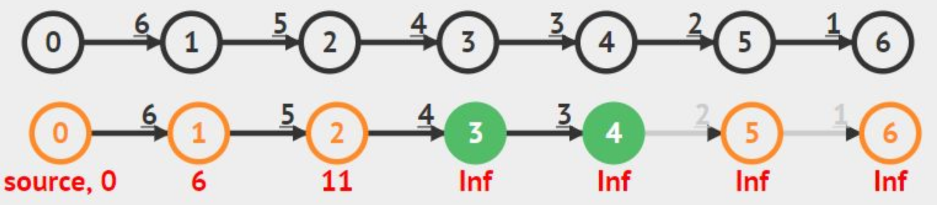
Drawing Graph Question

Draw a simple connected weighted directed graph with **9** vertices such that running
Optimized Bellman-Ford's algorithm from source vertex 0 uses at least **7** rounds to get the correct shortest paths.
Your graph cannot contain a negative weight cycle.
Note that all edges are processed according to the Adjacency List
i.e. all outgoing edges from vertex 0 is processed then edges from vertex 1 and so on.

As many rounds as possible.

Draw Graph

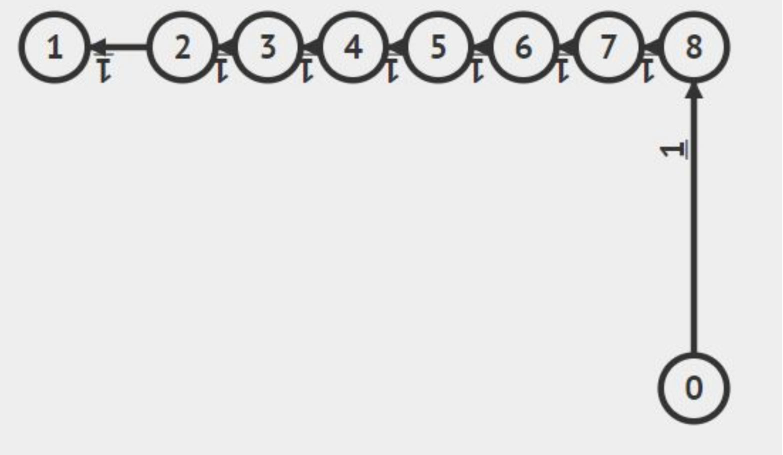
Note: **Unlike** quiz setting, VisuAlgo e-lecture example actually processes edges in decreasing vertex order.



Drawing Graph Question

Draw a simple connected weighted directed graph with **9** vertices such that running
Optimized Bellman-Ford's algorithm from source vertex 0 uses at least **7** rounds to get the correct shortest paths.
Your graph cannot contain a negative weight cycle.
Note that all edges are processed according to the Adjacency List
i.e. all outgoing edges from vertex 0 is processed then edges from vertex 1 and so on.

Draw Graph



We can mimic this as such:

1st round: 0 → 8, 8 → 7

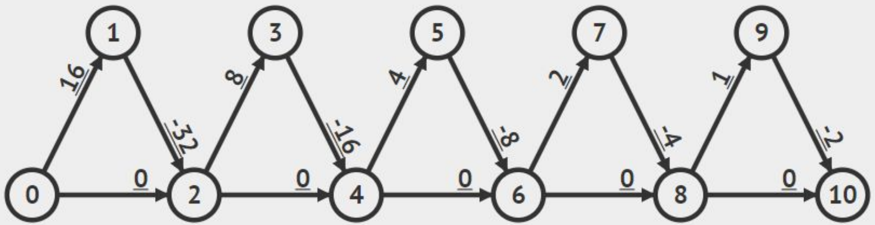
2nd round: 7 → 6

3rd round: 6 → 7

...

Drawing Graph Question

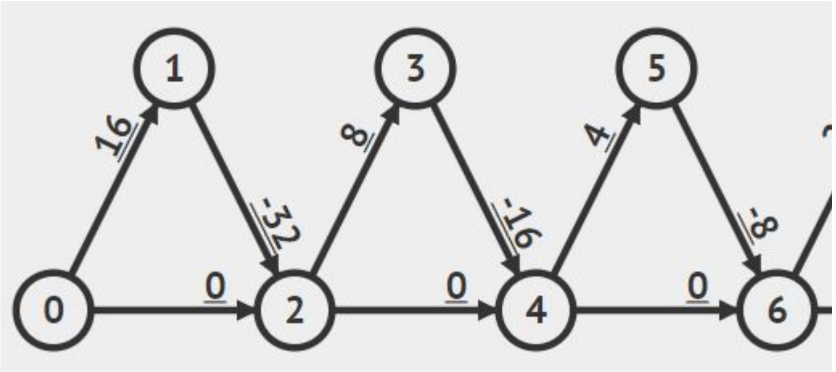
2. Draw a simple connected weighted directed graph with **9** vertices and at most **10** edges such that running <u>**Modified Dijkstra's**</u> **algorithm** from source vertex 0 successfuly relaxes $\geq$**16** edges to get the correct shortest paths.
We count 1 successful relaxation if relax(u, v, w_u_v) decreases D[v].
Your graph cannot contain a negative weight cycle. **As many successful relaxes as possible.**



You only have 10 edges, but you need 16 ‘relaxes’. How?

What graph can make Modified Dijkstra relax the same edge multiple times? Answer: **Dijkstra Killer!**

Drawing Graph Question

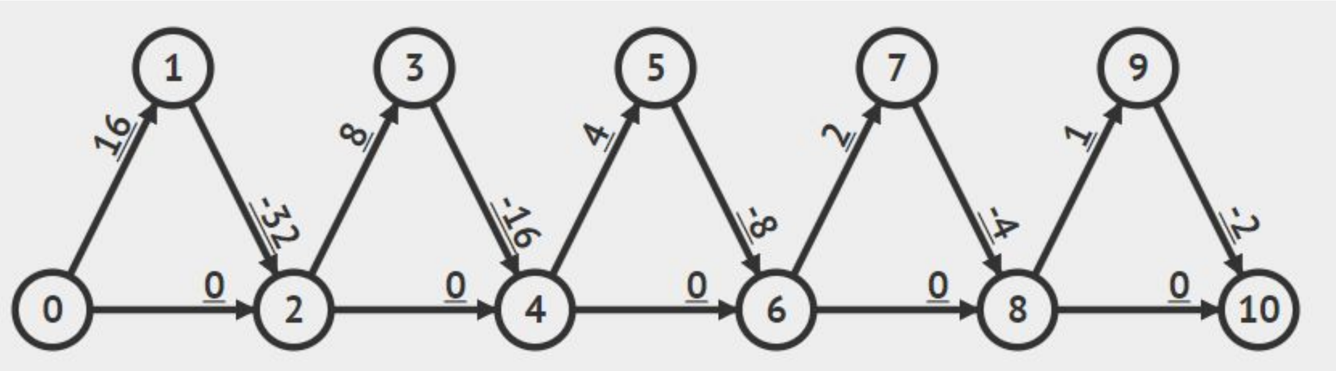


Okay.. now I have 7 vertices and 9 edges.

Can I join 2 vertices to the graph with 1 edge?

No.

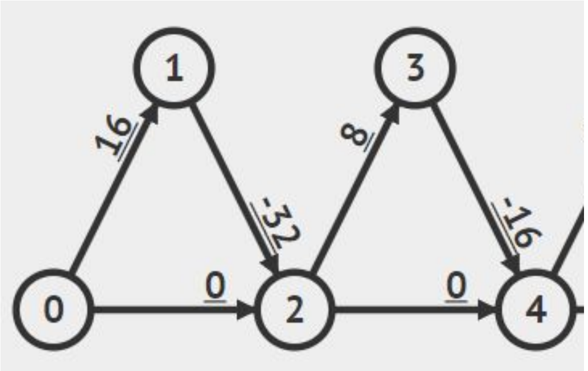
Drawing Graph Question



Oh no, I have 11 vertices and 15 edges.

That’s 2 vertex and 5 edges too many!

Drawing Graph Question

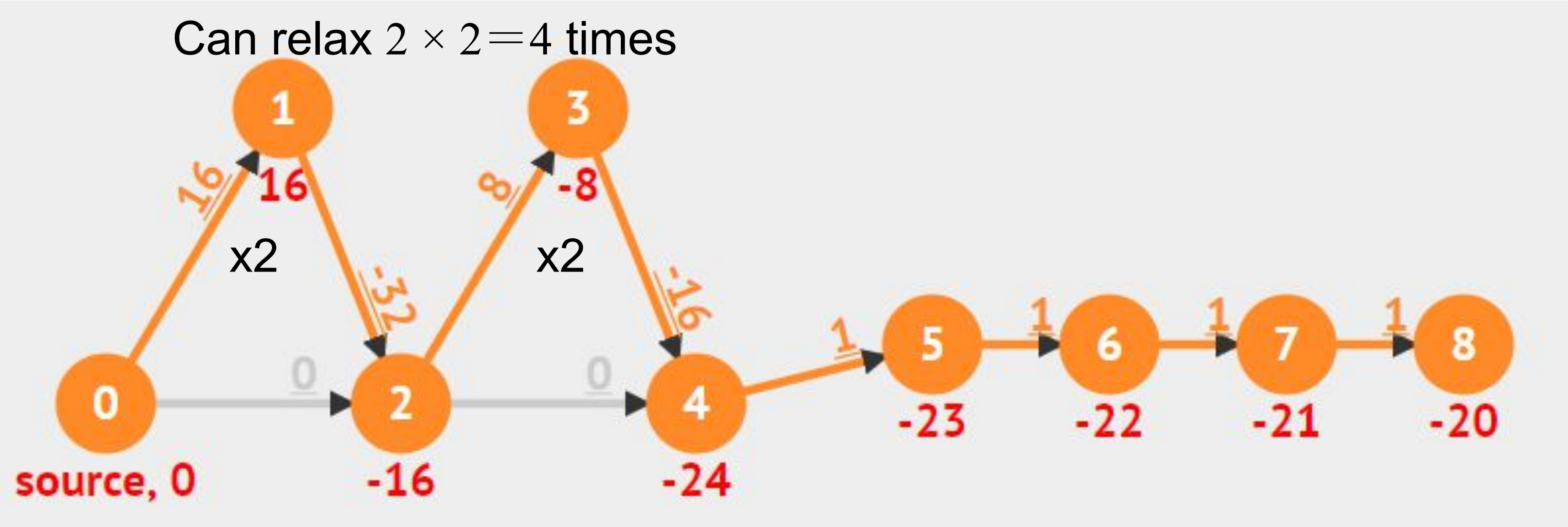


Okay.. now I have 5 vertices and 6 edges.

Can I join 4 vertices to the graph with 4 edge?

Yes!

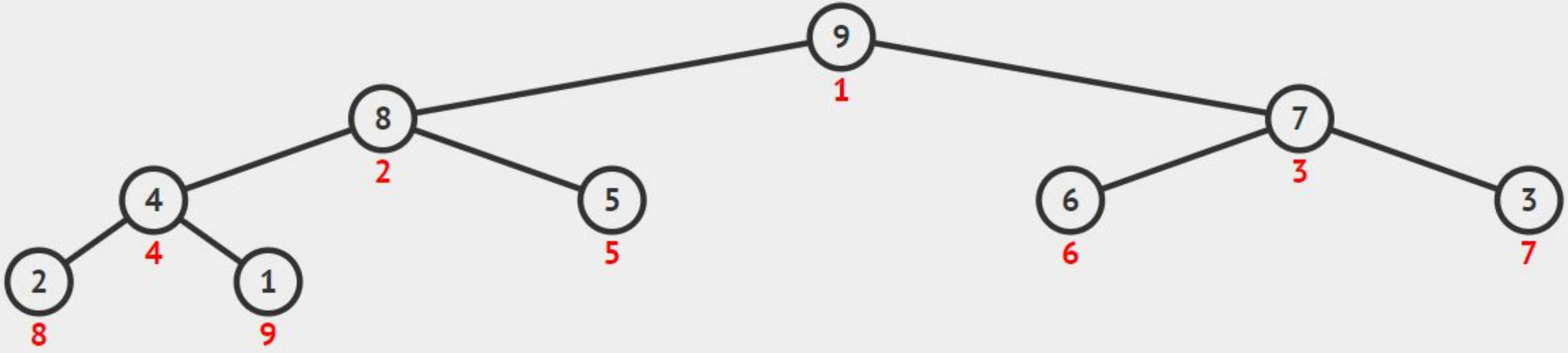
Drawing Graph Question



$$3 + 2 \times (3 + 2 \times 4) = 25 > 16$$

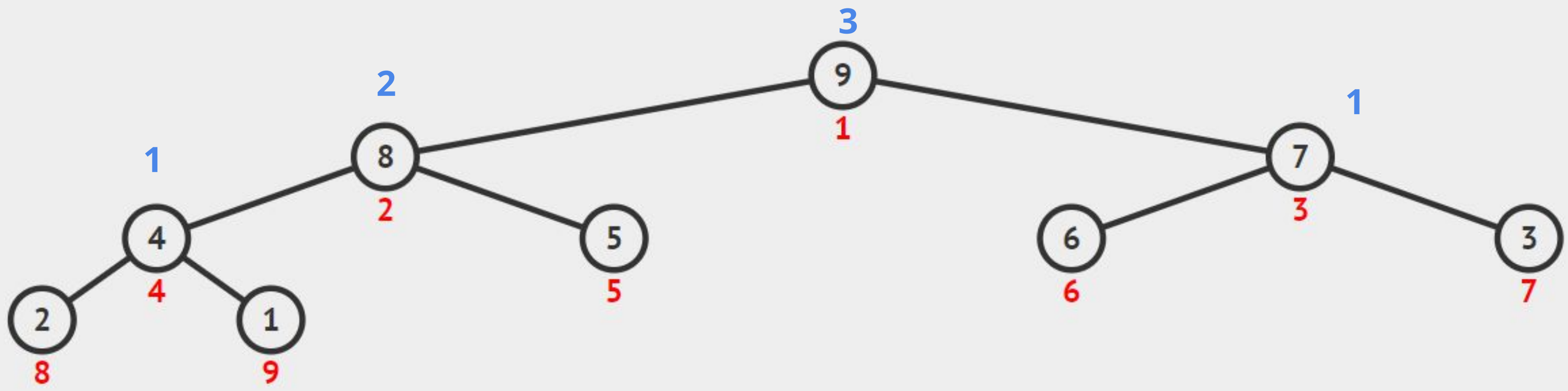
Binary Heap

12. What is the **MAXimum** number of **swaps** between heap elements required to construct a max heap of 9 elements using the **O(n)** BuildHeap(array)?



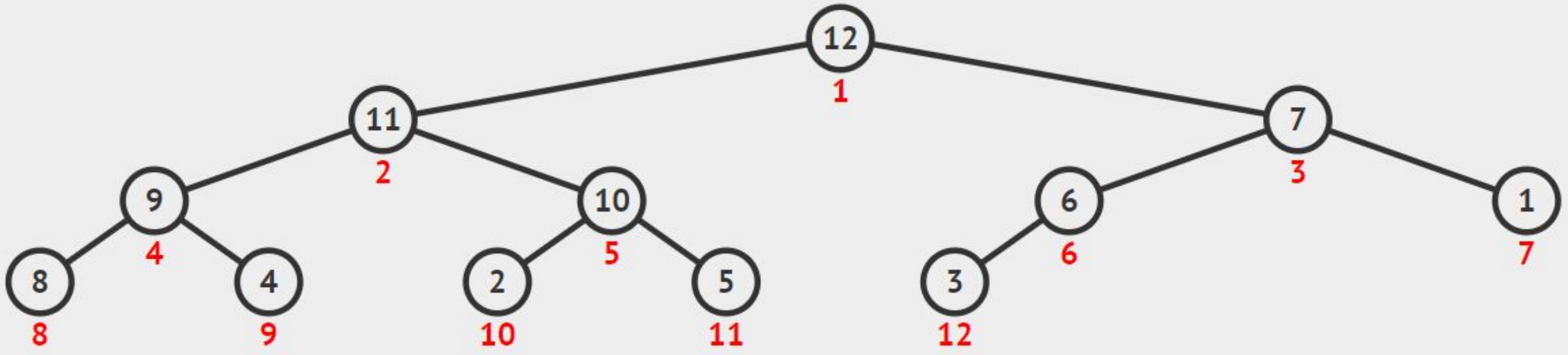
Binary Heap

12. What is the **MAXimum** number of **swaps** between heap elements required to construct a max heap of 9 elements using the **O(n)** BuildHeap(array)?



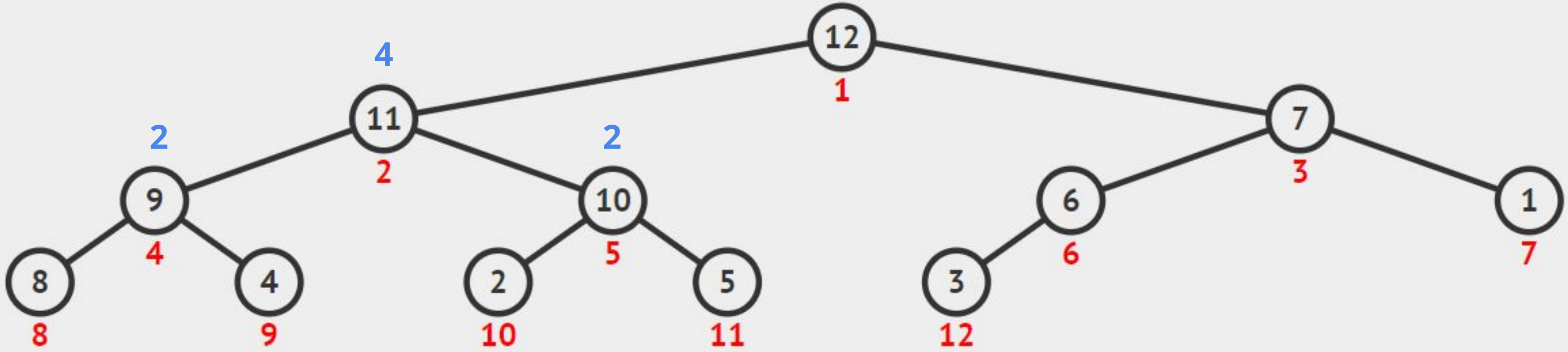
Binary Heap

6. What is the **MAXimum** number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



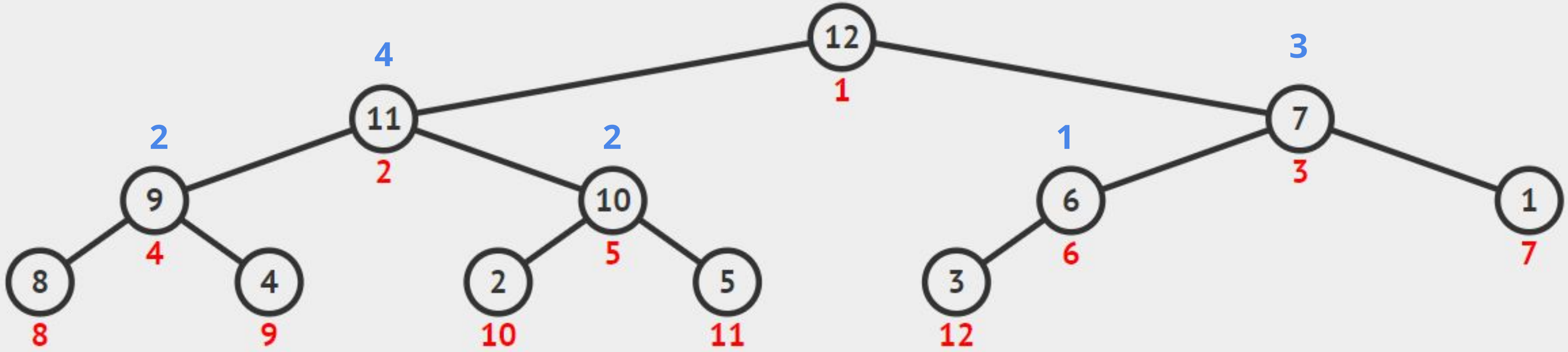
Binary Heap

6. What is the **MAXimum** number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



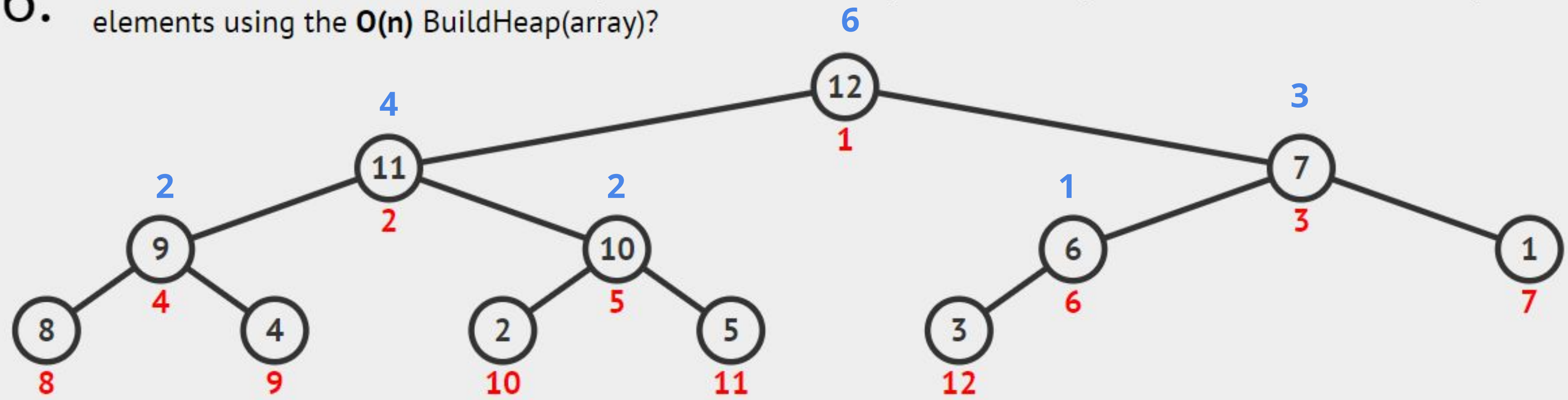
Binary Heap

6. What is the MAXimum number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



Binary Heap

6. What is the **MAXimum** number of **comparisons** between heap elements required to construct a max heap of 12 elements using the **O(n)** BuildHeap(array)?



Binary Search Tree

8. What is the **minimum** number of vertices in an AVL tree of **height** 6?

Recall height as the maximum *number of edges* from root to leaf.

Let $S(n)$ be min number of vertices of a height n AVL tree.

$S(0) = 1,\ \ S(1) = 2,\ \ S(2) = 4$

Binary Search Tree

8. What is the minimum number of vertices in an AVL tree of **<u>height</u>** 6?

Recall <u>height as the maximum *number of edges* from root to leaf.</u>

Let $S(n)$ be min number of vertices of a height n AVL tree.

$$
\begin{aligned}
&S(0)=1,\ \ S(1)=2,\ \ S(2)=4\\
&S(n)=S(n-1)+S(n-2)+1\\
S(6) &= S(5) + S(4) + 1\\
&= 2\times S(4)+S(3) + 2\\
&= 3\times S(3)+2\times S(2) + 4\\
&= 5\times S(2)+3\times S(1) + 7\\
&= 5\times 4 + 3\times 2 + 7\\
&= 33
\end{aligned}
$$

Termination

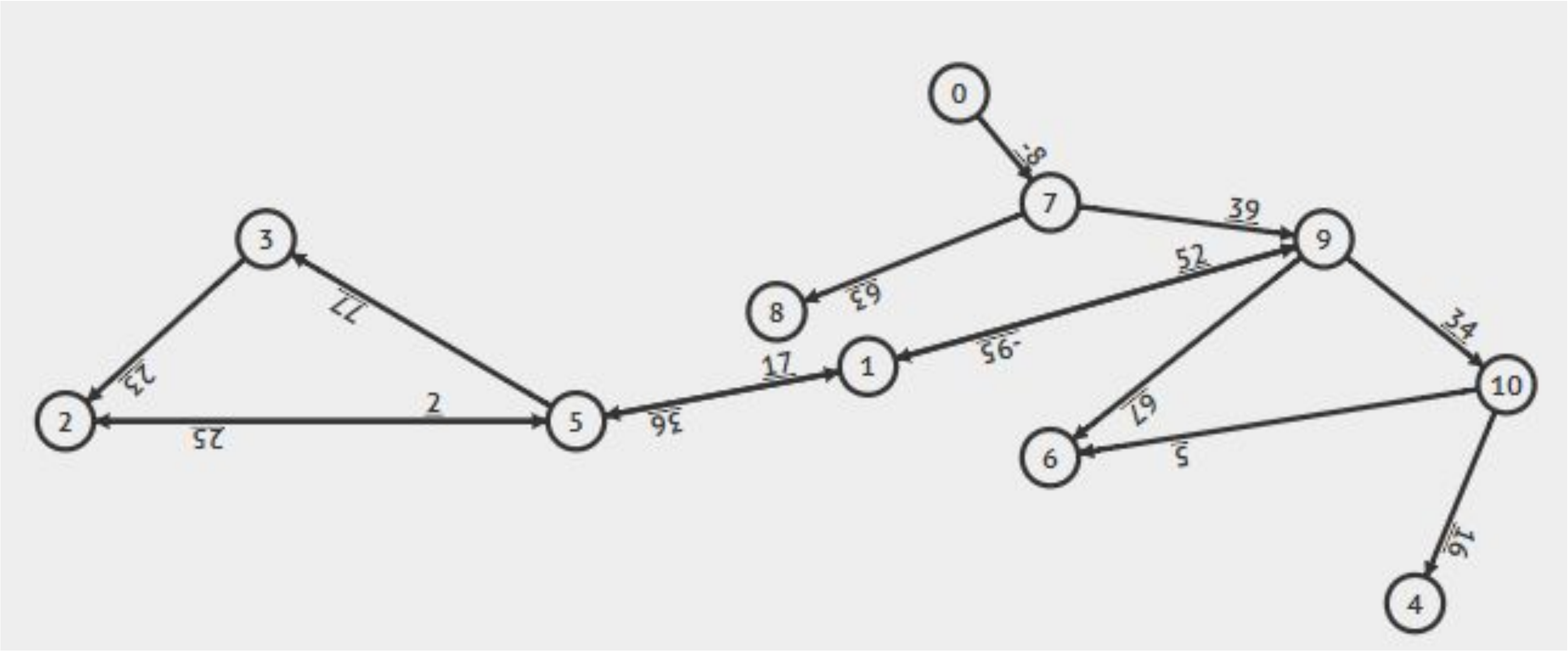
2. Run **Modified Dijkstra's algorithm** and **Bellman-Ford's algorithm** from source vertex **9** to solve the SSSP problem for this graph and compare the two algorithms.
Which one of the following statements is true?

- Both algorithms will terminate.
- Only Dijkstra's algorithm will terminate.
- Only Bellman-Ford's algorithm will terminate.

You answered this question wrongly.

The correct answer is: **Only Bellman-Ford's algorithm will terminate.**

Termination



Termination

Graph property	Optimised Bellman-Ford		Original Dijkstra		Modified Dijkstra	
	Terminate	Result	Terminate	Result	Terminate	Result
Negative **cycle**	YES	WA	YES	WA	NO	NA
Negative **edge**	YES	AC	YES	WA/AC*	YES	AC†
Dijkstra Killer	YES	AC	YES	WA	YES	AC‡
BF Killer	YES	AC	YES	AC	YES	AC

Special graphs:

- Dijkstra Killer: No negative **cycle**
- BF Killer: No negative **edge**

Footnotes:

*: Depends on the graph, could be either!

†: Might take more than $O((V+E)\ log\ V)$

‡: Takes exponential time (very long)!

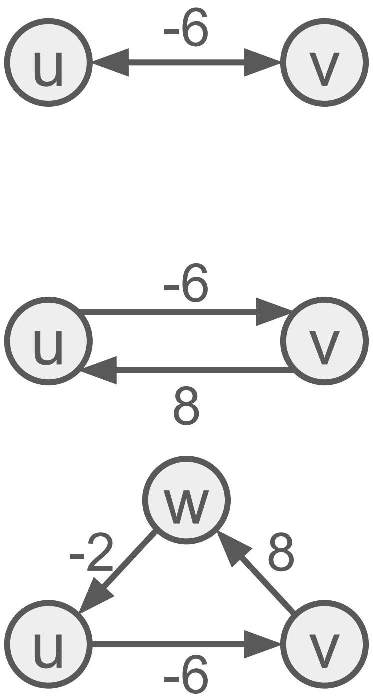
SSSP Strategies

Graph property	Best strategy	Time complexity
Tree	BFS/DFS	$O(V+E)$
DAG	Relax vertices in topologically sorted order	$O(V+E)$
Unweighted	BFS	$O((V+E)\ log\ V)$
No negative weighted edge	Original Dijkstra	$O((V+E)\ log\ V)$
No negative weighted cycle	Modified Dijkstra	$\approx O((V+E)\ log\ V)$
Negative weighted cycle	None! Not solvable, but can be detected using Bellman-Ford	N.A

Some caveats

Be careful!

- Some edges are **bidirectional** and so if they have negative weight then they entail a negative cycle! i.e. $u \to v \to u \to \ldots$ will get more and more negative
- Bidirectional edges can have different weights for different directions
- Cycles of edge weight sum zero is **NOT** a negative weight cycle!



Terminologies

Bipartite Graph

A graph which vertices can be partitioned into 2 disjoint sets such that there are no edges between vertices in the same set.

For example:

- Male and Female vertices
- No edges between male-male, female-female.

Terminologies

Spanning tree

"Click all the edges that must belong to every spanning tree of the graph shown below."

Select all the **edges** that will disconnect the graph when removed.

Final Advice (tips)?

- Some questions are *tedious*, cannot be memorized.
- For those with *small number of input cases*, it might be wise to prepare answers beforehand for example:
    - Max number of swaps for heap with n vertices
    - Preprocess for $5 < n \leq 13$

Final Advice (tips)?

Give each question a chance

- Every question has equal weightage, easy or hard
  - Actually 12%/20 = 0.6% per question
- Skip first when you encounter hard(er) questions