Shortest Path Problems

All-Pairs Shortest Path (APSP)

Problems Covered

1. All-Pairs Shortest Distances (Easy)
2. Reachability Between All Pairs (Easy - Medium)
3. City With the Fewest Neighbours in a Threshold (Medium)
4. Negative Edges and Negative Cycles (Medium)
5. Sparse Graph with Negative Edges (Johnson) (Medium - Hard)
6. All-Pairs Distances After New Roads (Hard)

All-Pairs Shortest Distances

Problem 1 | Easy

The Problem

- Given a **weighted directed graph** with n vertices (0 to n−1) and non-negative edge weights, find the **shortest distance between every pair** of vertices.
- Output an n × n matrix d, where d[i][j] is the shortest distance from i to j. If j cannot be reached from i, d[i][j] = ∞.

Example:

n = 4

Edges u → v (w): 0→1 (3), 0→3 (7), 1→0 (8), 1→2 (2), 2→0 (5), 2→3 (1), 3→0 (2)

Output: [[0, 3, 5, 6], [5, 0, 2, 3], [3, 6, 0, 1], [2, 5, 7, 0]]

Any ideas on how this can be solved?

Hints

- Running Dijkstra from every vertex works. Is there a simpler method that needs only a matrix and three loops?
- Suppose you already know the shortest distances that use only vertices 0 to k−1 in the middle of a path. When can vertex k improve the distance from i to j?
- Which of the three loops (i, j or k) must be the outermost one?

The Solution

Key Idea (Floyd-Warshall)

- Let d_k[i][j] be the shortest i → j distance that uses only vertices 0..k as intermediate vertices.
- Either vertex k is not used, or the path goes i → k → j: d_k[i][j] = min(d_k−1[i][j], d_k−1[i][k] + d_k−1[k][j]).

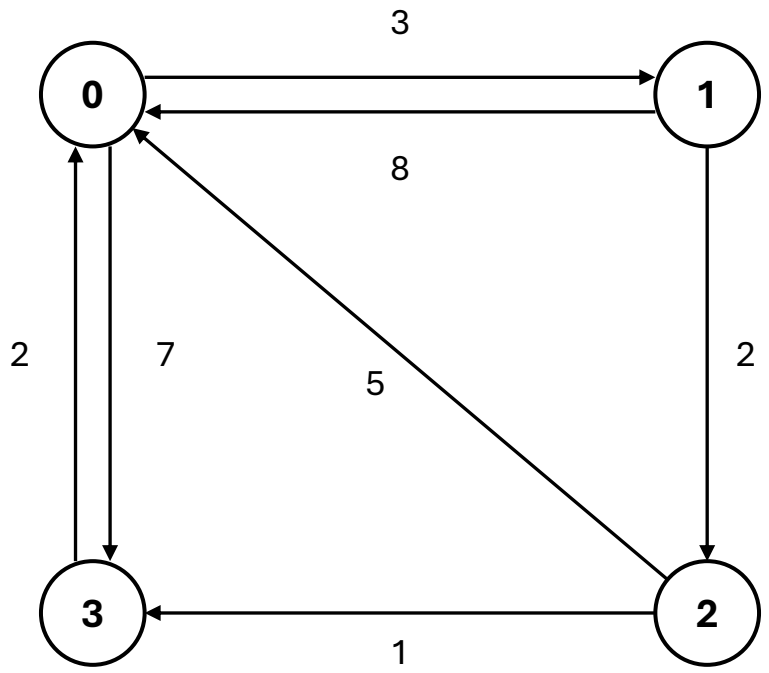
Steps

- d[i][i] = 0, d[u][v] = w for each edge, all other entries ∞.
- For k = 0 to n−1: for every pair (i, j), try going through k.
- The matrix d is the answer. k must be the outer loop.

```
FloydWarshall(n, edges):
  d[i][j] = INF, d[i][i] = 0
  for (u, v, w) in edges:
    d[u][v] = min(d[u][v], w)
  for k in 0..n-1:
    for i in 0..n-1:
      for j in 0..n-1:
        if d[i][k] + d[k][j] < d[i][j]:
          d[i][j] = d[i][k] + d[k][j]
  return d
```

Example

n = 4



k	Entries that improve
start	only direct edges
0	d[1][3] = 15, d[2][1] = 8, d[3][1] = 5
1	d[0][2] = 5, d[3][2] = 7
2	d[0][3] = 6, d[1][0] = 7, d[1][3] = 3
3	d[1][0] = 5, d[2][0] = 3, d[2][1] = 6

Example (contd.)

Final matrix d[i][j] (row i to column j)

	0	**1**	**2**	**3**
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

- d[1][3] = 3 through 1→2→3 (2 + 1). The direct route 1→0→3 would cost 15.
- d[2][1] = 6 through 2→3→0→1 (1 + 2 + 3).
- d[0][2] = 5 through 0→1→2 (3 + 2).
- Every vertex can reach every other here, so no ∞ appears.

Complexity Analysis

1. Time Complexity

- Three nested loops over k, i and j, each running n times.
- Time Complexity = **$O(V^3)$**
- Running Dijkstra from each vertex costs O(V (V + E) log V), which is better only for sparse graphs.

2. Space Complexity

- The n × n matrix is updated in place.
- Space Complexity = **$O(V^2)$**

Reachability Between All Pairs

Problem 2 | Easy - Medium

The Problem

- Given a **directed, unweighted graph** with n vertices, build the matrix R where R[i][j] = 1 if there is a path from i to j, and 0 otherwise.
- Every vertex reaches itself, so R[i][i] = 1.

Example:

n = 4

Edges: 0→1, 1→2, 2→0, 3→2

Output: [[1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1]]

Any ideas on how this can be solved?

Hints

- We only care whether a path exists, not how long it is. Can the distance matrix be replaced by a 0/1 matrix?
- In Floyd-Warshall, what should take the place of “+” and “min”?
- Whole rows of the matrix are combined at once. Can a bitset make this faster?

The Solution

Key Idea (Warshall)

- Same structure as Floyd-Warshall, but with booleans.
- i reaches j if it already does, or if i reaches k and k reaches j: R[i][j] = R[i][j] OR (R[i][k] AND R[k][j]).

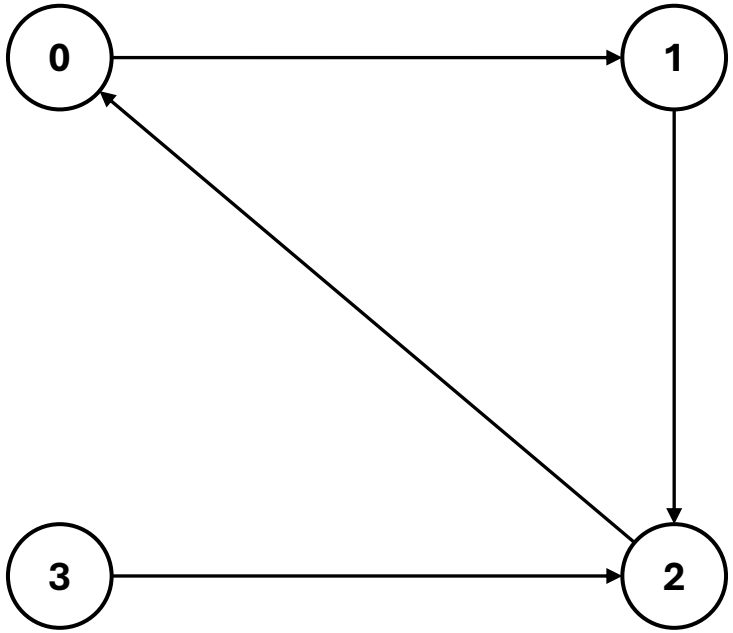
Steps

- R[i][i] = 1, R[u][v] = 1 for each edge, rest 0.
- For each k, and each i with R[i][k] = 1, OR the whole row k into row i.
- With bitsets the row OR handles 64 columns per step.

```
Warshall(n, edges):
  R[i][j] = 0, R[i][i] = 1
  for (u, v) in edges:
    R[u][v] = 1
  for k in 0..n-1:
    for i in 0..n-1:
      if R[i][k]:
        for j in 0..n-1:
          R[i][j] = R[i][j] OR R[k][j]
  return R
```

Example

n = 4



k	New entries set to 1
start	direct edges only
0	R[2][1] = 1
1	R[0][2] = 1
2	R[1][0] = 1, R[3][0] = 1, R[3][1] = 1
3	no change

Example (contd.)

Final matrix R[i][j]

	0	**1**	**2**	**3**
0	1	1	1	0
1	1	1	1	0
2	1	1	1	0
3	1	1	1	1

- Vertices 0, 1, 2 form a cycle, so each of them reaches the other two.
- Vertex 3 has an edge into this cycle, so it reaches 0, 1 and 2.
- No edge leads into vertex 3, so column 3 is 0 everywhere except R[3][3].

Complexity Analysis

1. Time Complexity

- Three nested loops over k, i and j.
- Time Complexity = **$O(V^3)$**, or **$O(V^3 / w)$** with bitsets (w = word size).
- Alternative: one DFS or BFS from every vertex costs O(V (V + E)).

2. Space Complexity

- The V × V boolean matrix: V^2 bits with bitsets.
- Space Complexity = **$O(V^2)$**

City With the Fewest Neighbours in a Threshold

Problem 3 | Medium

The Problem

- There are n cities connected by bidirectional roads (u, v, w). Given an integer **distanceThreshold** T, find the city that can reach the **smallest number of other cities** with a shortest-path distance of at most T.
- If several cities tie, return the one with the **greatest number**.

Example:

n = 4, T = 4

Roads: 0-1 (3), 1-2 (1), 1-3 (4), 2-3 (1)

Output: 3

Cities 0 and 3 each reach 2 cities within distance 4, and 3 is larger.

Any ideas on how this can be solved?

Hints

- A road can be used through other cities. What do you need to know first for every pair of cities?
- Once the all-pairs distances are known, what is left to do for each city?
- Check the tie rule carefully. In which order should the cities be scanned, and should the comparison be $<$ or $\leq$?

The Solution

Key Idea

- Compute all-pairs shortest distances with Floyd-Warshall (roads are undirected, so set d[u][v] and d[v][u]).
- For each city i, count the cities $j \neq i$ with $d[i][j] \leq T$.

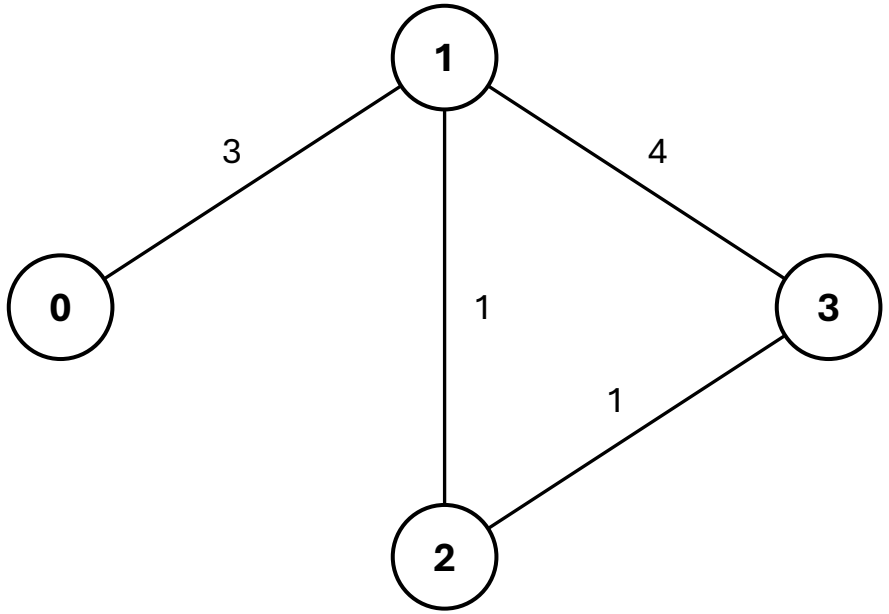
Steps

- Run Floyd-Warshall to get d.
- Scan $i = 0 .. n-1$ and count the cities within T.
- Update the answer whenever count ≤ best. Using ≤ makes a later (larger) city win ties.

```
d = FloydWarshall(n, roads)
best = INF
ans = -1
for i in 0..n-1:
  cnt = 0
  for j in 0..n-1:
    if j != i and d[i][j] <= T:
      cnt = cnt + 1
  if cnt <= best:
    best = cnt
    ans = i
return ans
```

Example

n = 4, T = 4



k	Entries that improve (symmetric pairs)
0	no change
1	d[0][2] = 4, d[0][3] = 7
2	d[0][3] = 5, d[1][3] = 2
3	no change

Pairs are listed once (i < j); d[j][i] changes the same way.

Example (contd.)

Distance matrix, and the cities within T = 4

	0	**1**	**2**	**3**
0	0	3	4	5
1	3	0	1	2
2	4	1	0	1
3	5	2	1	0

City	**Reachable within 4**	**Count**
0	{1, 2}	2
1	{0, 2, 3}	3
2	{0, 1, 3}	3
3	{1, 2}	2

Counts = [2, 3, 3, 2]. The minimum is 2, shared by cities 0 and 3.

Scanning in increasing order with ≤ keeps the later city, so the answer is **3**.

Complexity Analysis

1. Time Complexity

- Floyd-Warshall takes $O(V^3)$. Counting the cities within T takes $O(V^2)$.
- Time Complexity = **$O(V^3)$**
- For a sparse graph, Dijkstra from every city gives $O(V \cdot E \log V)$, which can be faster.

2. Space Complexity

- Distance matrix of size $V \times V$.
- Space Complexity = **$O(V^2)$**

Negative Edges and Negative Cycles

Problem 4 | Medium

The Problem

- Given a directed graph with n vertices and edge weights that **can be negative**, find the shortest distance between every pair of vertices.
- If the graph contains a **negative-weight cycle**, shortest distances are not defined: report "Negative cycle".

Example 1:

n = 4. Edges u → v (w): 0→1 (4), 0→2 (−2), 1→3 (2), 2→1 (3), 2→3 (5)

Output: [[0, 1, −2, 3], [∞, 0, ∞, 2], [∞, 3, 0, 5], [∞, ∞, ∞, 0]]

Example 2:

n = 3. Edges: 0→1 (1), 1→2 (−1), 2→0 (−1)

Output: Negative cycle

Any ideas on how this can be solved?

Hints

- Does Floyd-Warshall rely on weights being non-negative? Think about what its recurrence actually uses.
- The diagonal d[i][i] starts at 0. If i lies on a cycle of negative total weight, what can happen to d[i][i]?
- When d[i][k] is ∞, what is ∞ + (−5) in your code? How do you avoid a wrong update?

The Solution

Key Idea

- Floyd-Warshall works with negative edges, as long as there is no negative cycle.
- A negative cycle through i makes the path i → … → i cost less than 0, so d[i][i] < 0.

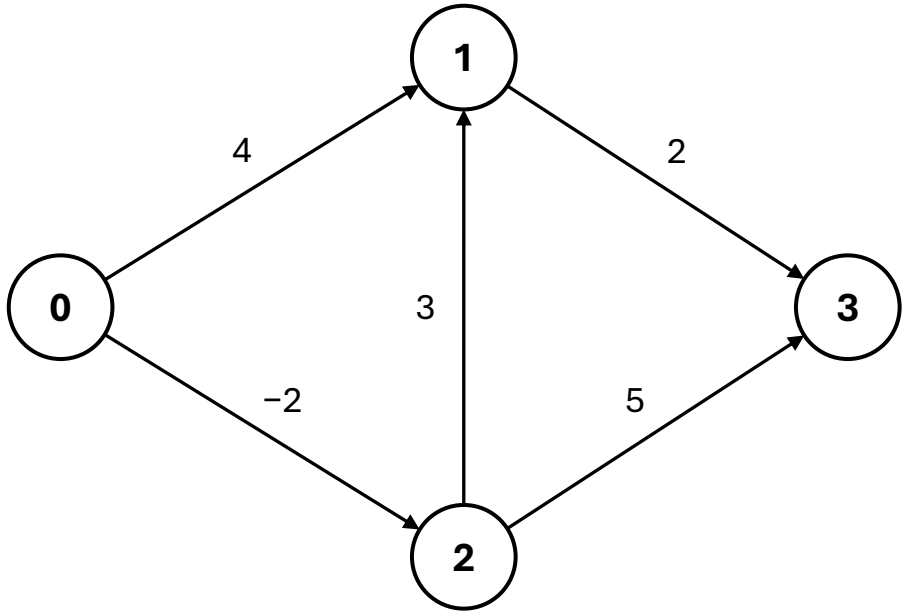
Steps

- Run Floyd-Warshall, skipping updates when d[i][k] or d[k][j] is ∞.
- If any d[i][i] < 0, report “Negative cycle”.
- Otherwise d is the answer.
- With a negative cycle, values can grow very large: use a wide integer type.

```
d[i][j] = INF, d[i][i] = 0
add all edges to d
for k in 0..n-1:
  for i in 0..n-1:
    if d[i][k] == INF: continue
    for j in 0..n-1:
      if d[k][j] == INF: continue
      if d[i][k] + d[k][j] < d[i][j]:
        d[i][j] = d[i][k] + d[k][j]
for i in 0..n-1:
  if d[i][i] < 0:
    return "Negative cycle"
return d
```

Example

Example 1: n = 4



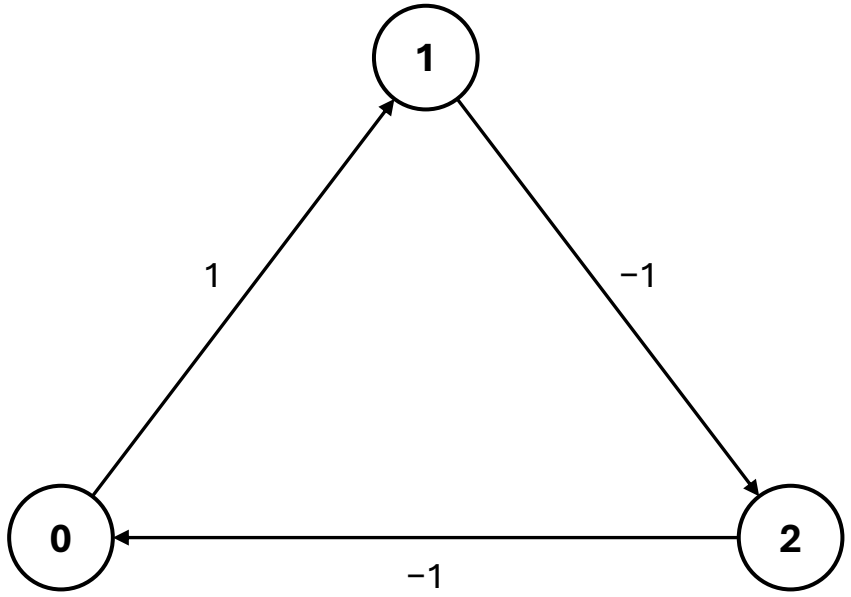
k	Entries that improve
start	only direct edges
0	no change
1	d[0][3] = 6
2	d[0][1] = 1, d[0][3] = 3
3	no change

The diagonal stays 0, so there is no negative cycle.

d[0][1] = 1 uses 0→2→1 (−2 + 3), cheaper than the direct edge (4).

Example (contd.)

Example 2: 0→1 (1), 1→2 (−1), 2→0 (−1)



k	d[0][0]	d[1][1]	d[2][2]
start	0	0	0
0	0	0	0
1	0	0	−1
2	−1	−1	−2

Cycle weight = 1 + (−1) + (−1) = −1.

After k = 2 the diagonal entries are negative: d[0][0] = −1, d[1][1] = −1, d[2][2] = −2.

Hence the answer: “Negative cycle”.

Complexity Analysis

1. Time Complexity

- Same three nested loops as before; the final diagonal check costs O(V).
- Time Complexity = **$O(V^3)$**
- Dijkstra from every vertex cannot be used here, because of the negative edges.

2. Space Complexity

- The V × V distance matrix, updated in place.
- Space Complexity = **$O(V^2)$**

Sparse Graph with Negative Edges (Johnson)

Problem 5 | Medium - Hard

The Problem

- Given a **sparse** directed graph (E is close to V) with edge weights that may be negative but **no negative cycle**, find all-pairs shortest distances.
- Floyd-Warshall at $O(V^3)$ is too slow for V = 5000, so we want something close to $O(V \cdot E \log V)$.

Example:

n = 4

Edges u → v (w): 0→1 (−2), 0→2 (4), 1→2 (3), 2→3 (−1), 3→1 (2)

Output: [[0, −2, 1, 0], [∞, 0, 3, 2], [∞, 1, 0, −1], [∞, 2, 5, 0]]

Any ideas on how this can be solved?

Hints

- Dijkstra from each vertex is fast on sparse graphs, but it breaks with negative edges. Can the weights be changed so that all become non-negative?
- Give each vertex a number h[v] and use w'(u, v) = w + h[u] − h[v]. What is the total change in cost of any path from a to b?
- How can h be computed so that every w' ≥ 0? Think of a new source vertex joined to all vertices by 0-weight edges.

The Solution

Key Idea (Johnson)

- With w' = w + h[u] − h[v], every a → b path changes by the same amount, so shortest paths stay the same.
- If h[v] is the distance to v from an extra source q, then h[v] ≤ h[u] + w, so w' ≥ 0.

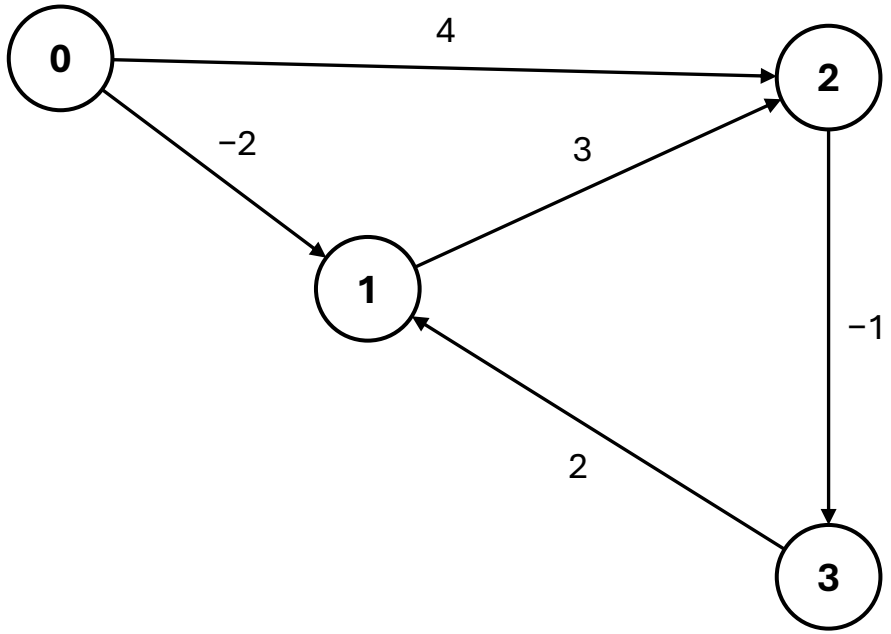
Steps

- Add q with 0-weight edges to all vertices; run Bellman-Ford from q to get h.
- Reweight the edges, run Dijkstra from every vertex.
- Undo the shift: d[u][v] = d'[u][v] − h[u] + h[v].

```
Johnson(G):
  add q, edges q -> v with weight 0
  h = BellmanFord(q)
  if negative cycle: return "Negative cycle"
  for (u, v, w) in edges:
    w' = w + h[u] - h[v]
  for u in 0..n-1:
    d' = Dijkstra(G', u)
    for v in 0..n-1:
      d[u][v] = d'[v] - h[u] + h[v]
  return d
```

Example

n = 4. Bellman-Ford from q gives h = [0, −2, 0, −1]



Edge	w	h[u] − h[v]	w'
0 → 1	−2	2	0
0 → 2	4	0	4
1 → 2	3	−2	1
2 → 3	−1	1	0
3 → 1	2	1	3

All w' ≥ 0, so Dijkstra is safe from every vertex.

Path 0→1→2→3 has w' total 0 + 1 + 0 = 1, so d[0][3] = 1 − h[0] + h[3] = 1 − 0 − 1 = 0.

Example (contd.)

Dijkstra on the reweighted graph, then shifting back

u	d'[u][0..3] on G'	d[u][0..3] on G
0	[0, 0, 1, 1]	[0, −2, 1, 0]
1	[∞, 0, 1, 1]	[∞, 0, 3, 2]
2	[∞, 3, 0, 0]	[∞, 1, 0, −1]
3	[∞, 3, 4, 0]	[∞, 2, 5, 0]

Row 0: d'[0][3] = 1, so d[0][3] = 1 − 0 + (−1) = 0, which is the path 0→1→2→3 (−2 + 3 − 1).

Unreachable pairs stay ∞ in both graphs. Vertex 0 has no incoming edge, so column 0 is ∞ except d[0][0].

Complexity Analysis

1. Time Complexity

- Bellman-Ford from q: $O(V \cdot E)$. Reweighting: $O(E)$.
- Dijkstra with a heap from each of V vertices: $V \times O((V + E) \log V)$.
- Time Complexity = **$O(V \cdot E + V (V + E) \log V)$**, i.e. $O(V \cdot E \log V)$ for connected graphs.
- Beats $O(V^3)$ whenever E is much smaller than V^2.

2. Space Complexity

- Output matrix: $O(V^2)$. Working space (h, heap, adjacency list): $O(V + E)$.
- Space Complexity = **$O(V^2)$** including the output

All-Pairs Distances After New Roads

Problem 6 | Hard

The Problem

- You are given the n × n matrix d of shortest distances between all pairs of n cities ($n \leq 300$).
- q new bidirectional roads (a, b, w) are built one after another ($q \leq 300$). After **each** road, print the **sum of shortest distances over all unordered pairs** ($i < j$).

Example:

n = 3, d = [[0, 4, 5], [4, 0, 9], [5, 9, 0]]

Roads: (1, 2, 8), then (0, 1, 1)

Output: 17 12

Any ideas on how this can be solved?

Hints

- Re-running Floyd-Warshall after every road costs $O(V^3)$ per query, which is too slow. Can one road be handled in $O(V^2)$?
- A new shortest path from i to j either ignores the new road, or uses it. How many times can it use it?
- If it uses road (a, b) once, the two remaining parts are old shortest paths. How can the new length be written with the old d?

The Solution

Key Idea

- A shortest path uses the new road at most once, in one of two directions: i → a → b → j or i → b → a → j.
- The pieces i → a and b → j are already shortest, so the new length is d[i][a] + w + d[b][j] (or with a and b swapped).

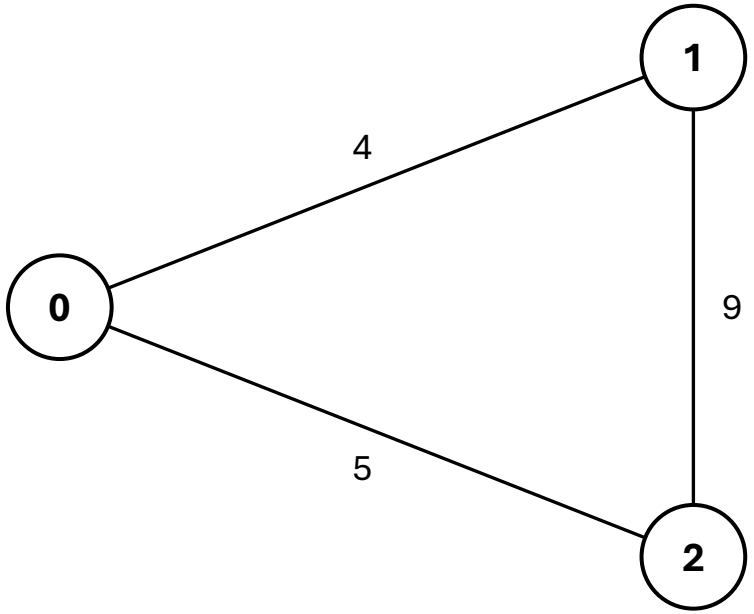
Steps

- For every pair (i, j): d[i][j] = min(d[i][j], d[i][a] + w + d[b][j], d[i][b] + w + d[a][j]).
- Update in place: values stay real path lengths and never go below the true distance.
- Add up d[i][j] for i < j and print it.

```
for each new road (a, b, w):
  for i in 0..n-1:
    for j in 0..n-1:
      d[i][j] = min(d[i][j],
                    d[i][a] + w + d[b][j],
                    d[i][b] + w + d[a][j])
  total = 0
  for i in 0..n-1:
    for j in i+1..n-1:
      total = total + d[i][j]
  print(total)
```

Example

n = 3. Graph of the starting distances: sum = 18



Road	Entries that change	Sum
start	—	18
(1, 2, 8)	d[1][2] = 8	17
(0, 1, 1)	d[0][1] = 1, d[1][2] = 6	12

Road (1, 2, 8): d[1][2] drops from 9 to 8. Sum = 4 + 5 + 8 = 17.

Road (0, 1, 1): d[0][1] becomes 1, and d[1][2] drops again to 6 (via 1→0→2 = 1 + 5). Sum = 1 + 5 + 6 = 12.

Complexity Analysis

1. Time Complexity

- Each new road updates all V^2 pairs in O(1) each.
- Time Complexity = **$O(V^3 + Q \cdot V^2)$** for Q roads, with the initial matrix computed once by Floyd-Warshall (if it is not given).
- Re-running Floyd-Warshall every time would cost $O(Q \cdot V^3)$.

2. Space Complexity

- Only the V × V matrix is stored and updated in place.
- Space Complexity = **$O(V^2)$**