Synchronous Shopping editorial

Required knowledge: Dijkstra, bitmasks.

(if you don’t know the Dijkstra algorithm or the bitmaks, you can reference to my github:

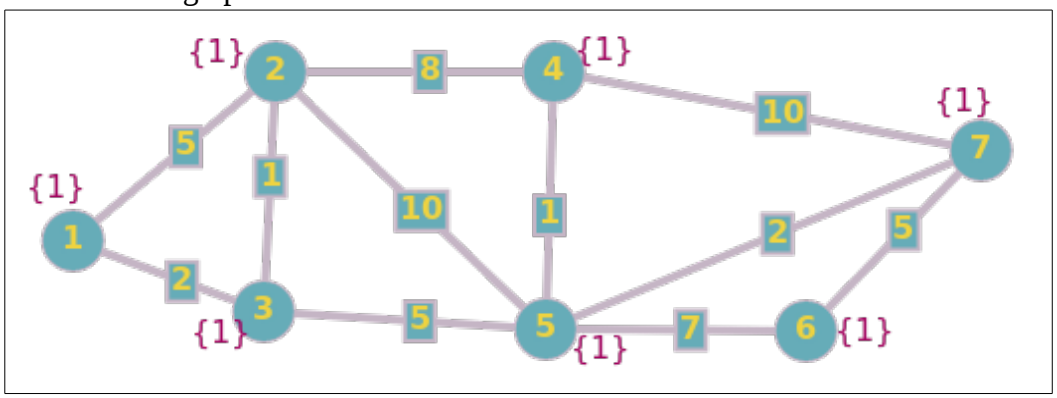
https://github.com/Mourad-NOUAILI/Graph-theory)

Problem: https://www.hackerrank.com/challenges/synchronous-shopping/problem

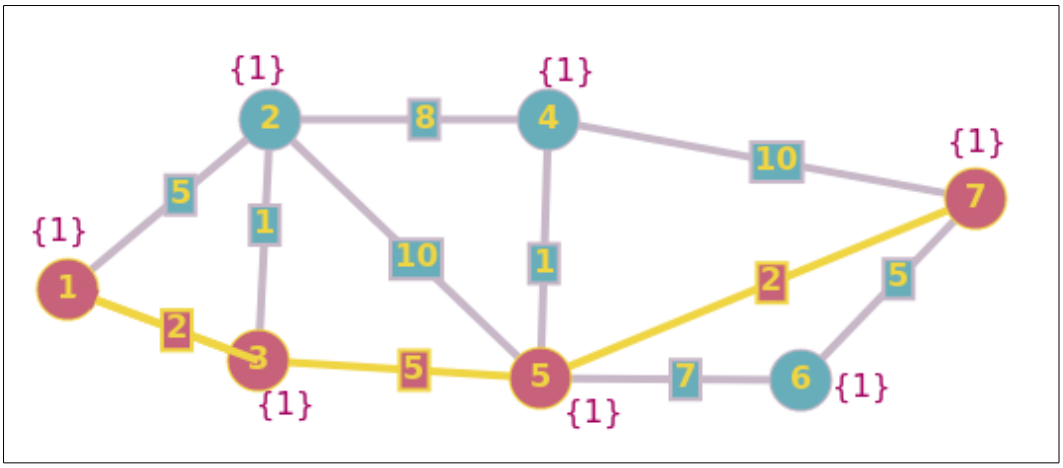
To make it easier, we gonna start with some simple version of the problem.

One cat, one fish in all shops

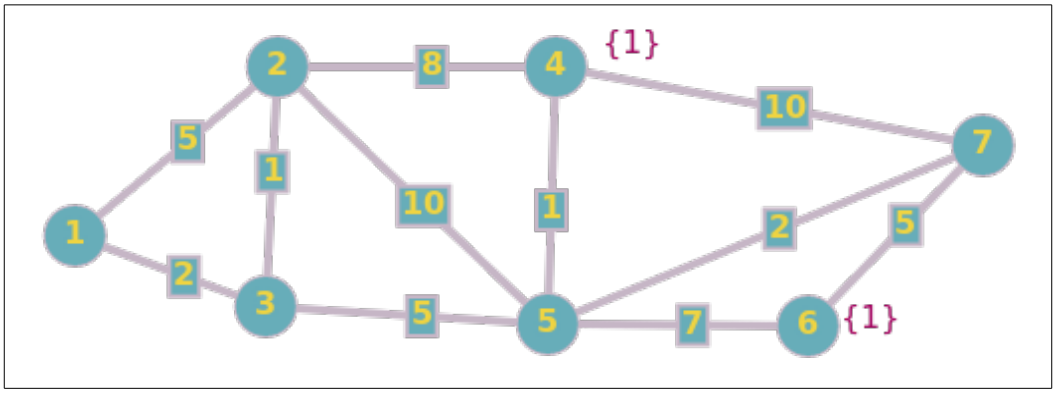
Let’s take this graph:



To resolve this, just run the Dijkstra algorithm:

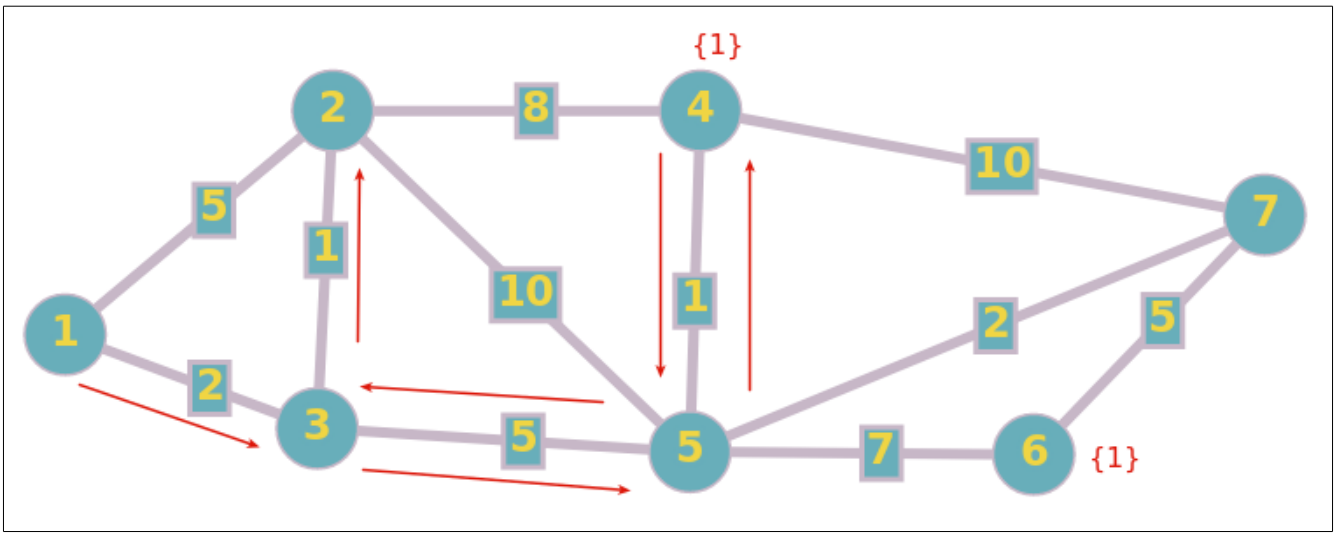


One cat, one fish type, that may or may not exits



At each visited shop, we have to know the distance with or without the fish.

For example, to reach shop #2, without the fish, the cat must run a distance of 3. But, if he reach the shop #2, with the fish, he run a distance of 15.



As you see, the same shop could be visited multiple times. For each visit (to the same shop), the collections states of fishes could be changed.

So, for each shop, we need to store distances for each different state. Here we have two states, at a shop i, $(1 \leq i \leq n)$, the cat have or have get not the fish.

To do that, we need a $(n * 2)$ matrix to store distances.

Needed data structures (see code below)

- An array of lists (or vectors) for the adjacency lists.
- An array of boolean to know if a shop i, $(1 \leq i \leq n)$ has the the fish or not.
- A $n * 2$ matrix to save distances.
- A priority queue for Dijkstra algorithm.

How to do

(1) Fill the the array of boolean.

(2) Build the graph.

(3) run Dijkstra to fill the $n * 2$ matrix.

(4) The shortest distance is d[n][true]

Dijkstra algorithm

The Dijkstra algorithm will compute the distance to reach a vertex v from the source (vertex 1), by considering the fact all collections states of v.

for this graph, we have a $7 * 2$ matrix of distances:

d		
	false	true
1	0	∞
2	∞	∞
3	∞	∞
4	∞	∞
5	∞	∞
6	∞	∞
7	∞	∞

The principle of Dijkstra algorithm doesn't change. Two thing will change compared to conventional Dijkstra algorithm:

The priority queue stores the distance computed of a vertex v ($d[v][mask]$) with v and its $mask$. So, each vertex v, ($d[v][mask]$, (v, $mask$)) is stored.

The relaxation pseudo-code will be:

$for\ each\ v \in adj[u]$

$\quad if\ d[v][masks[u]\ ||\ masks[v]] > d[u][mask[u]] + w\ (u, v)$

$\quad\quad d[v][masks[u]\ ||\ masks[v]] = d[u][mask[u]] + w\ (u, v)$

$\quad\quad \mathbf{q \leftarrow q \cup \{(d[v][masks[u]\ ||\ masks[v]], (v, masks[u]\ ||\ masks[v]))\}}$

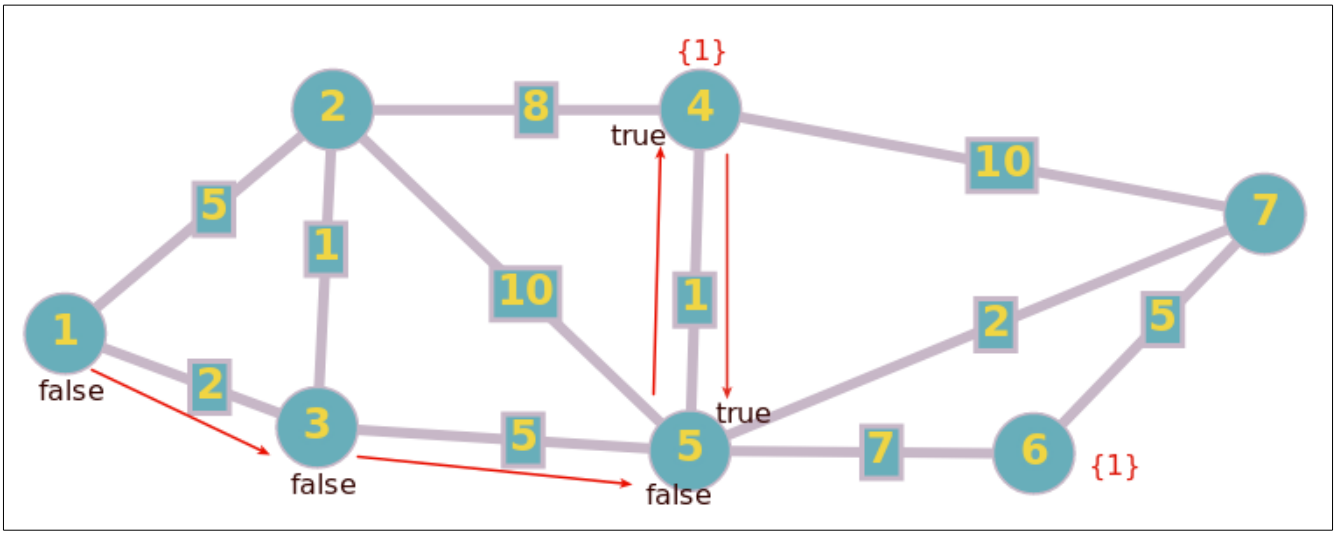
why the 'or' operation $masks[u]$ | $masks[v]$?

By moving from the vertex u to v, the collection states will change.

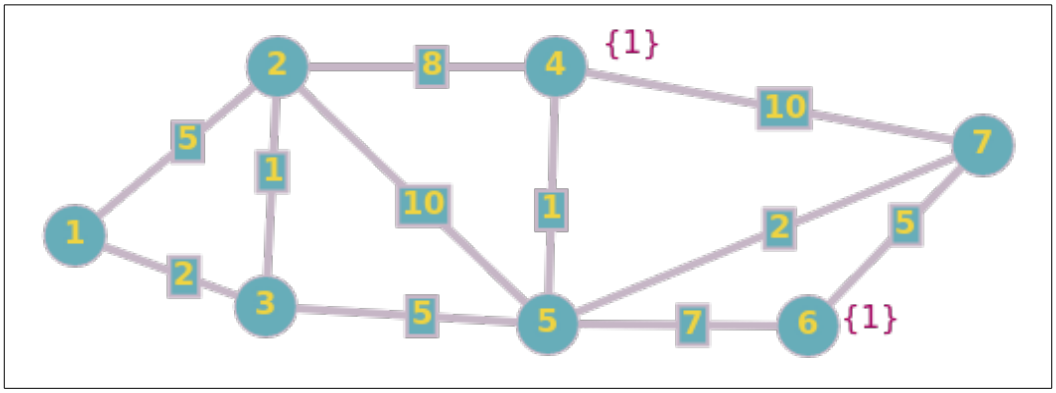
For example:

by going from shop #1 → shop #3 → shop #5: the cat does not getting the fish yet ($false$). From shop #5 to shop #4, the collection state change to $true$, because the cat get the fish.

By coming back to shop #5, the collection state is $true$ (because the cat got the fish at shop #4)



Run by hand on an example



d		
	false	true
1	0	∞
2	~~∞~~ ~~5~~ 3	~~∞~~ 16
3	~~∞~~ 2	~~∞~~ 14
4	∞	~~∞~~ ~~11~~ 8
5	~~∞~~ 7	~~∞~~ 9
6	∞	~~∞~~ 14
7	~~∞~~ 9	~~∞~~ ~~18~~ 11

masks						
1	**2**	**3**	**4**	**5**	**6**	**7**
false	false	false	true	false	true	false

q = {(0, (1, false))}

- 1, false

q = {}

  - 2

    d[2][false || false] > d[1][false] + w(1, 2)

    ∞ > 0 + 5 ==> d[2][false] = 5

    q = {(5, (2, false))}

  - 3

    d[3][false || false] > d[1][false] + w(1, 3)
    ∞ > 0 + 2 ==> d[3][false] = 2

    q = {(2, (3, false)), (5, (2, false))}

- 3, false

q = {(5, (2, false))}

  - 1

    d[1][false || false] > d[3][false] + w(3, 1)

    0 > 2 + 2 ==> Nothing to do

  - 2

    d[2][false || false] > d[3][false] + w(3, 2)
    5 > 2 + 1 ==> d[2][false] = 3

    q = {(3, (2, false)), (5, (2, false))}

  - 5

    d[5][false || false] > d[3][false] + w(3, 5)
    ∞ > 2 + 5 ==> d[5][false] = 7

    q = {(3, (2, false)), (5, (2, false)), (7, (5, false))}

- 2, false

q = {(5, (2, false)), (7, (5, false))}

  - 1

    d[1][false || false] > d[2][false] + w(2, 1)

    0 > 3 + 5 ==> Nothing to do

  - 3

    d[3][false || false] > d[2][false] + w(2, 3)
    2 > 3 + 1 ==> Nothing to do

  - 5

    d[5][false || false] > d[2][false] + w(2, 5)
    7 > 3 + 10 ==> Nothing to do.

  - 4

    d[4][false || true] > d[2][false] + w(2, 4)
    ∞ > 3 + 8 ==> d[4][true] = 11

    q = {(5, (2, false)), (7, (5, false)), (11, (4, true))}

- 2, false

q = {(7, (5, false)), (11, (4, true))}

  - 1

    d[1][false || false] > d[2][false] + w(2, 1)

    0 > 3 + 5 ==> Nothing to do

  - 3

    d[3][false || false] > d[2][false] + w(2, 3)
    2 > 3 + 1 ==> Nothing to do

  - 5

    d[5][false || false] > d[2][false] + w(2, 5)
     7 > 3 + 10 ==> Nothing to do.

  - 4

    d[4][false || true] > d[2][false] + w(2, 4)
     11 > 3 + 8 ==> Nothing to do.

- 5, false

q = {(11, (4, true))}

  - 3

    d[3][false || false] > d[5][false] + w(5, 3)

    2 > 7 + 5 ==> Nothing to do

  - 2

    d[2][false || false] > d[5][false] + w(5, 2)
    3 > 7 + 11 ==> Nothing to do

  - 7

    d[7][false || false] > d[5][false] + w(5, 7)
    ∞ > 7 + 2 ==> d[7][false] = 9

    q = {(9, (7, false)), (11, (4, true))}

  - 4

    d[4][false || true] > d[5][false] + w(5, 4)
    11 > 7 + 1 ==> d[4][true] = 8

    q = {(8, (4, true)), (9, (7, false)), (11, (4, true))}

  - 6

    d[6][false || true] > d[5][false] + w(5, 6)
    ∞ > 7 + 7 ==> d[6][true] = 14

    q = {(8, (4, true)), (9, (7, false)), (11, (4, true)), (14, (6, true))}

- 4, true

q = {(9, (7, false)), (11, (4, true)), (14, (6, true))}

  - 2

    d[2][true || false] > d[4][true] + w(4, 2)

    ∞ > 8 + 8 ==> d[2][true] = 16

    q = {(9, (7, false)), (11, (4, true)), (14, (6, true)), (16, (2, true))}

  - 5

    d[5][true || false] > d[4][true] + w(4, 5)
    ∞ > 8 + 1 ==> d[5][true] = 9

    q = {(9, (7, false)), (9, (5, true)), (11, (4, true)), (14, (6, true)), (16, (2, true))}

  - 7

    d[7][true || false] > d[4][true] + w(4, 7)
    ∞ > 8 + 10 ==> d[7][true] = 18

    q = {(9, (7, false)), (9, (5, true)), (11, (4, true)), (14, (6, true)), (16, (2, true)),

    (18, (7, true))}

- 7, false

q = {(9, (5, true)), (11, (4, true)), (14, (6, true)), (16, (2, true)), (18, (7, true))}

  - 4

    d[4][false || true] > d[7][false] + w(7, 4)

    8 > 9 + 10 ==> Nothing to do.

  - 5

    d[5][false || false] > d[7][false] + w(7, 5)
    7 > 9 + 2 ==> Nothing to do.

  - 6

    d[6][false || true] > d[7][false] + w(7, 6)
    14 > 9 + 5 ==> Nothing to do

- 5, true

q = {(11, (4, true)), (14, (6, true)), (16, (2, true)), (18, (7, true))}

  - 4

    d[4][true || true] > d[5][true] + w(5, 4)

    8 > 9 + 1 ==> Nothing to do.

  - 2

    d[2][true || false] > d[5][true] + w(5, 2)
    16 > 9 + 10 ==> Nothing to do.

  - 3

    d[3][true || false] > d[5][true] + w(5, 3)
    ∞ > 9 + 5 ==> d[3][true] = 14

    q = {(11, (4, true)), (14, (6, true)), (14, (3, true)), (16, (2, true)), (18, (7, true))}

  - 6

    d[6][true || true] > d[5][true] + w(5, 6)
    14 > 9 + 7 ==> Nothing to do.

  - 7

    d[7][true || false] > d[5][true] + w(5, 7)
    18 > 9 + 2 ==> d[7][true] = 11

    q = {(11, (4, true)), (11, (7, true)), (14, (6, true)), (14, (3, true)), (16, (2, true)),

    (18, (7, true))}

We continue until the priority queue become empty.

At the end of Dijkstra algorithm we get:

d		
	false	true
1	0	16
2	3	15
3	2	14
4	∞	8
5	7	9
6	∞	14
7	9	11

The minimum distance to get the fish at the 7^{th} shopping center is 11.

C++ code: $O(m \log n \cdot 2)$

```
#include <bits/stdc++.h>


using namespace std;

const int INF = 1e9;

int n, m;

int **dist = NULL;
vector<pair<int, int>> *adj = NULL;
bool *masks = NULL;

void createAdjList(void) {
  for (int edge = 0 ; edge < m ; ++edge) {
    int x , y , z;
    cin >> x >> y >> z;

    x--;
    y--;

    adj[x].push_back({y, z});
    adj[y].push_back({x, z});
  }
}
```

```
void dijkstra (int start) {
  for (int i = 0 ; i < n ; ++i)
    for (int j = 0 ; j < 2 ; ++j)
      dist[i][j] = INF;

  dist[start][masks[start]] = 0;

  priority_queue<pair<int, pair<int, int>>,
    vector<pair<int, pair<int, int>>>,
    greater<pair<int, pair<int, int>>>> q;

  q.push({dist[start][masks[start]], {start, masks[start]}});

  while (!q.empty()) {
    int current = q.top().second.first;
    int currentMask = q.top().second.second;
    q.pop();

    for (auto it: adj[current]) {
      int v = it.first;
      int w = it.second;
      if (dist[v][currentMask || masks[v]] > dist[current][currentMask] + w ) {
        dist[v][currentMask || masks[v]] = dist[current][currentMask] + w;
        q.push({dist[v][currentMask || masks[v]], {v, currentMask || masks[v]}});
      }
    }
  }
}
```

```
int main() {
  ios_base::sync_with_stdio(false);

  cin >> n >> m;

  adj= new vector<pair<int, int>>[n];
  masks = new bool[n];

  dist = new int*[n];
  for (int i = 0 ; i < n ; ++i)
    dist[i] = new int [2];


  for (int i = 0 ; i < n ; ++i) {
    int ti;
    cin >> ti;
    assert(ti == 0 | ti == 1);
    masks[i] = ti;
  }


  createAdjList();


  dijkstra(0);

  int ans = dist[n-1][1];
  cout << ans << '\n';


  return 0;
}
```

Input Format

The first line contains 2 space-separated integers: N (the number of shopping centers) and M (the number of roads), respectively.

Each line i of the N subsequent lines ($1 \leq i \leq N$) describes a shopping center as a line of space-separated integers. Each line takes the following form:

- 0 the i^{th} shopping center doesn't contain the fish.
- 1 the i^{th} shopping center contain the fish.

Each line j of the M subsequent lines ($1 \leq j \leq M$) contains 3 space-separated integers describing a road. The first two integers, X_j and Y_j, describe the two shopping centers it connects. The third integer, Z_j, denotes the amount of time it takes to travel the road (i.e., travel time).

Output Format

Print the minimum amount of time it will take for the cat to purchase the fish and reach the shopping center N.

Sample Input

```
7 11
0
0
0
1
0
1
0
1 2 5
1 3 2
2 3 1
2 4 8
2 5 10
3 5 5
4 5 1
4 7 11
5 6 7
5 7 2
6 7 5
```

Sample Output

```
11
```

Two cats, one fish that may exist or not

Now the actual problem has two cats. But that doesn't make much difference as we know shortest distance to reach n^{th} shopping center with any of the fish. So why not try all combinations, i.e. mask $m1$ and $m2$ such that $m1 \; or \; m2 = true$ and we take the minimum for all such combos.

$BestTime \leftarrow \infty$

$For \; m1 \in [1, 1 << 1]$

$\quad for \; m2 \in [m1, 1 << 1]$

$\quad\quad if \; (m1 \mid m2) = true$

$\quad\quad\quad bestTime \leftarrow min(bestTime, max(d[n][m1], d[n][m2]))$

Code C++: $O(m \; log \; n \; . \; 2)$

```
#include <bits/stdc++.h>


using namespace std;

const int INF = 1e9;

int n, m;

int **dist = NULL;
vector<pair<int, int>> *adj = NULL;
bool *masks = NULL;

void createAdjList(void) {
  for (int edge = 0 ; edge < m ; ++edge) {
    int x , y , z;
    cin >> x >> y >> z;

    x--;
    y--;

    adj[x].push_back({y, z});
    adj[y].push_back({x, z});
  }
}
```

```
void dijkstra (int start) {
  for (int i = 0 ; i < n ; ++i)
    for (int j = 0 ; j < 2 ; ++j)
      dist[i][j] = INF;

  dist[start][masks[start]] = 0;

  priority_queue<pair<int, pair<int, int>>,
    vector<pair<int, pair<int, int>>>,
    greater<pair<int, pair<int, int>>>> q;

  q.push({dist[start][masks[start]], {start, masks[start]}});

  while (!q.empty()) {
    int current = q.top().second.first;
    int currentMask = q.top().second.second;
    q.pop();

    for (auto it: adj[current]) {
      int v = it.first;
      int w = it.second;
      if (dist[v][currentMask || masks[v]] > dist[current][currentMask] + w ) {
        dist[v][currentMask || masks[v]] = dist[current][currentMask] + w;
        q.push({dist[v][currentMask || masks[v]], {v, currentMask || masks[v]}});
      }
    }
  }
}
```

```
int main() {
  ios_base::sync_with_stdio(false);

  cin >> n >> m;

  adj= new vector<pair<int, int>>[n];
  masks = new bool[n];

  dist = new int*[n];
  for (int i = 0 ; i < n ; ++i)
    dist[i] = new int [2];


  for (int i = 0 ; i < n ; ++i) {
    int ti;
    cin >> ti;
    assert(ti == 0 | ti == 1);
    masks[i] = ti;
  }


  createAdjList();


  dijkstra(0);

  int ans = INF;
  for (int i = 0 ; i < 2 ; ++i)
    for (int j = i ; j < 2 ; ++j)
      if (i || j == true)
        ans = min(ans, max(dist[n-1][i], dist[n-1][j]));

  cout << ans << '\n';
  return 0;
}
```

Two cats, many fishes

Problem: https://www.hackerrank.com/challenges/synchronous-shopping/problem

Editorial by zxqfd555: https://www.hackerrank.com/challenges/synchronous-shopping/editorial

This problem can be solved with Dijkstra's algorithm. Let's denote the state as (V, B), where V is the number of shopping centers and P is a bitmask of the first K bits denoting the kinds of fish which have already been bought.

The starting state is $(1, 0)$, meaning we start at shopping center 1 and have not yet purchased any fish. The shortest distance to the state $D_{(V,B)}$ denotes the minimum time required to visit shopping center V with fish from the mask B bought.

While spreading from the current (V, B) state, there are two possible options:

1. To state (V, B') with the time $D_{(V,B)}$ where $B' = B$ or $maskOfFishSoldAt[V]$. Recall that buying any amount of fish doesn't take any time, so it is always optimal to buy all fish sold in the shopping centers. This transition corresponds to buying the fish.
2. To state (Y, B) where Y is adjacent to V with the time $D_{(V,B)} + W$, and W is the time required to pass the road from V to Y. This transition corresponds to moving by a road.

When all the minimal times are calculated, let's brute-force the mask B_1 of the fish that will be bought by Little Cat and mask B_2 of the fish that will be bought by Big Cat. The essential condition is B_1 or $B_2 = 2^K - 1$. Then, the minimal time for this configuration will simply be equal to $\min(D_{(N,B_1)}, D_{(N,B_2)})$. Among all these configurations, choose the best one (i.e., the one having the minimal answer).

My editorial

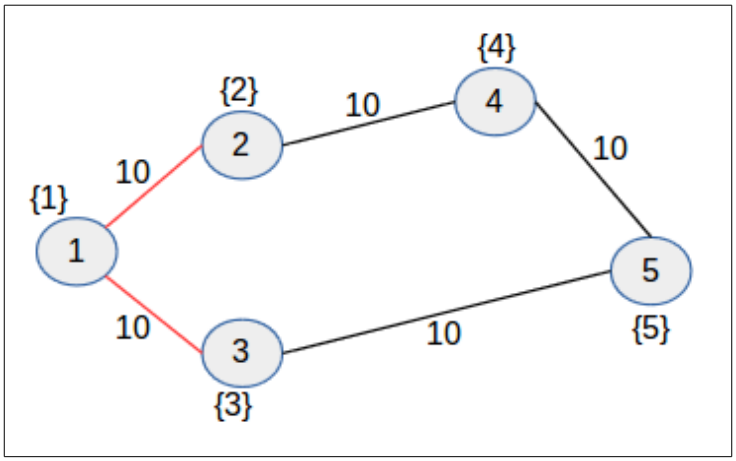
As we know the strategy, let's solve it for on cat, then we run into the d() values matrix to get the solution for two cats.

We have k type of fishes. As seen for one cat, one fish that may exist or not, we have to know all the states on each shop.

At each visited shop, we have to know the distance after the collection of some fishes.

For example, fishes {1, 2, 3} could be collected after running a distance of 30:

path = <1, 2, 1, 3>



As you see, the same shop could be visited multiple times. For each visit (to the same shop), the collections states of fishes could be changed.

So, for each shop, we need to store distances for each different state.

To do that, we need a $(n * all\_collections\_states)$ matrix to store distances.

To compute $all\_collections\_states$, we use bitmaks.

For example:

for k = 5 (the number of types of fish sold in Bitville):

- 00001: fish of type 1 is collected.
- 01010: fishes of types 2 and 4 are collected
- …….

we need a $(n * 2^{k)} matrix$ $(2^k = 1 << k)$

Needed data structures (see code below)

- An array of lists (or vectors) to store the each vertex neighbors.
- An array of each vertex bitmask (or mask).
- A n * (1 << k) matrix to save distances.
- A priority queue for Dijkstra algorithm.

How to do

(1) Fill the the array of each vertex bitmask.

(2) Build the graph.

(3) run Dijkstra to fill the n * (1 << k) matrix.

(4) Simulate the two cats run into the n * (1 << k) matrix to compute the minimum amount of time it will take for the cats to collectively purchase all k fish.

Dijkstra algorithm

The Dijkstra algorithm will compute the distance to reach a vertex v from the source (vertex 1), by considering the fact all collections states of v.

for this graph, we have a 3 * (1 << 3) matrix of distances:

d							
	001	010	011	100	101	110	111
1	0	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	∞	∞

The principle of Dijkstra algorithm doesn't change. Two thing will change compared to conventional Dijkstra algorithm:

The priority queue stores the distance computed a vertex v ($d[v][mask]$) with v and its $mask$. So, each vertex v, ($d[v][mask]$, $(v, mask)$) is stored.

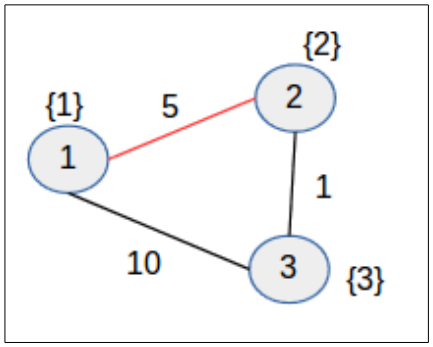
The relaxation pseudo-code will be:

$$
\begin{aligned}
&for\ each\ v \in adj[u] \\
&\quad if\ d[v][masks[u] \mid masks[v]] > d[u][mask[u]] + w\ (u, v) \\
&\qquad d[v][masks[u] \mid masks[v]] = d[u][mask[u]] + w\ (u, v) \\
&\qquad q \leftarrow q \cup \{(d[v][mask], (v, mask))\}
\end{aligned}
$$

why the bitwise operation $masks[u] \mid masks[v]$?

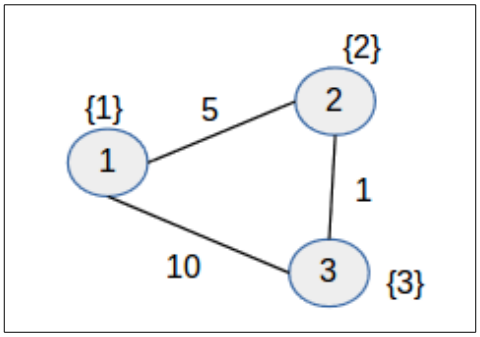
By moving from the vertex u to v, the collection states of fishes will change.

For example:

	In vertex 1, mask = 001. In vertex 2, mask = 010. By moving from vertex 1 to vertex 2, the mask will be equal to 001 | 010 = 011.

Run by hand on an example

let’s take this graph:



Fill the the array of each vertex bitmask.

masks		
1	2	3
001	010	100

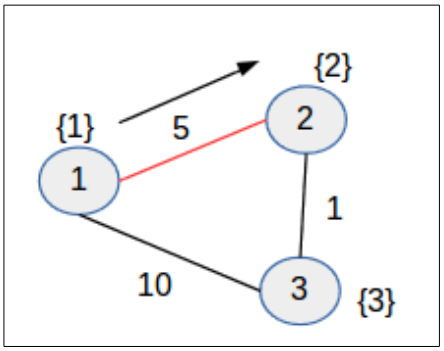
Run Dijkstra algorithm

d							
	001	010	011	100	101	110	111
1	0	∞	10	∞	20	∞	~~16~~ 12
2	∞	∞	5	∞	∞	∞	7
3	∞	∞	∞	∞	10	∞	6

- d[1][masks[1]] = d[1][001] = 0
  q = {(0, (1, 001))}

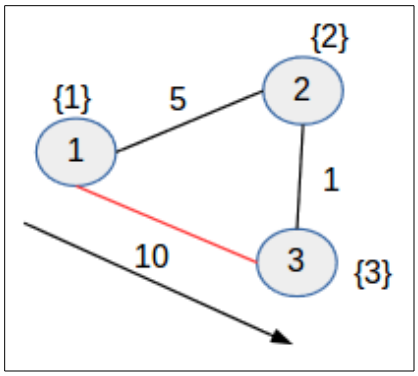
- 1’s neighbors, mask = 001:
  q = {}
  - 2:

    d[2][1’s mask | masks[2]] > d[1][masks[1]] + w (1, 2)
    d[2][001 | 010] > d[1][001] + w (1, 2)
    d[2][011] > 0 + 5
    ∞ > 5 => d[2][011] = 5
    q = {(5, (2, 011))}

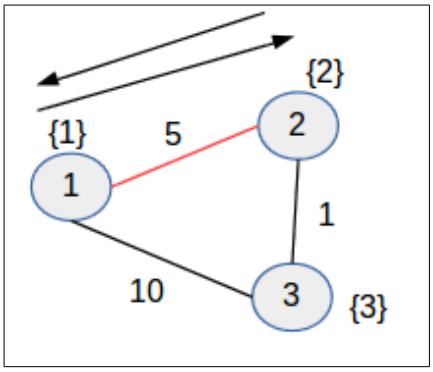


  - 3:

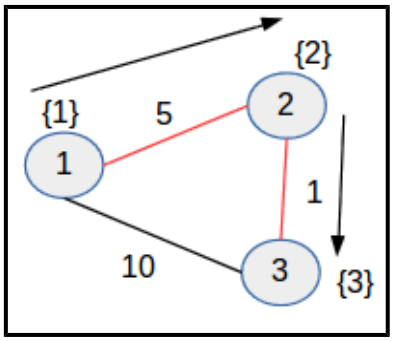
    d[3][masks[1] | masks[3]] > d[1][masks[1]] + w (1, 3)
    d[3][001 | 100] > d[1][001] + w (1, 3)
    d[3][101] > 0 + 10
    ∞ > 10 => d[3][101] = 10
    q = {(5, (2, 011)), (10, (3, 101))}



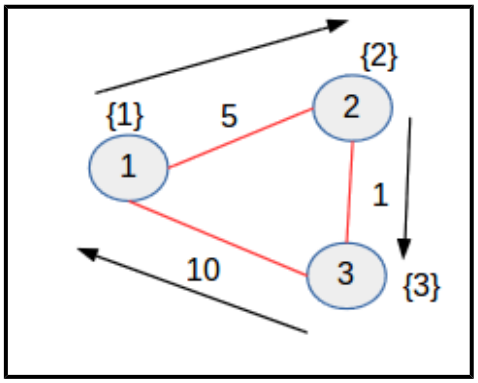
- 2's neighbors, mask = 011:
  q = {(10, (3, 101))}
  - 1:
    d[1][011 | 001] > d[2][011] + w (2, 1)
    d[1][011] > 5 + 5
    ∞ > 10 => d[1][011] = 10
    q = {(10, (3, 101)), (10, (1, 011))}



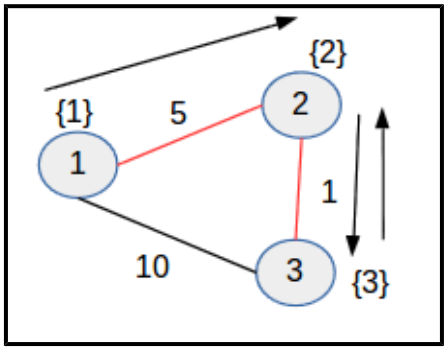
  - 3:
    d[3][011 | 100] > d[2][010] + w (2, 3)
    d[3][111] > 5 + 1
    ∞ > 6 => d[3][111] = 6
    q = {(6, (3, 111)), (10, (3, 101)), (10, (1, 011))}



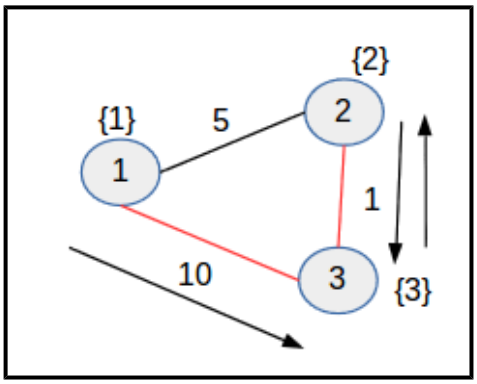
- 3's neighbors, mask = 111:
  q = {(10, (3, 101)), (10, (1, 011))}
  - 1:
    d[1][111 | 001] > d[3][111] + w (3, 1)
    d[1][111] > 6 + 10
    ∞ > 16 => d[1][111] = 16
    q = {(10, (3, 101)), (10, (1, 011)),
    (16, (1, 111))}



  - 2:
    d[2][111 | 010] > d[3][111] + w (3, 2)
    d[2][111] > 6 + 1
    ∞ > 7=> d[2][111] = 7
    q = {(7, (2, 111)), (10, (3, 101)), (10, (1, 011)),
    (16, (1, 111))}

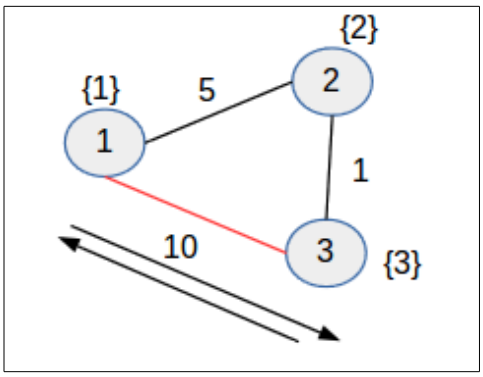


- 2's neighbors, mask = 111:
  q = {(10, (3, 101)), (10, (1, 011)), (16, (1, 111))}
  - 1:
    d[1][111 | 001] > d[2][111] + w (2, 1)
    d[1][111] > 7 + 5
    16 > 12 => d[1][111] = 12
    q = {(10, (3, 101)), (10, (1, 011)), (12, (1, 111)),
    (16, (1, 111))}



  - 3:
    d[3][111 | 100] > d[2][111] + w (2, 3)
    d[3][111] > 7 + 1
    6 > 8=> No.

- 3's neighbors, mask = 101:
  q = {(10, (1, 011)), (12, (1, 111)), (16, (1, 111))}
  - 1:
    d[1][101 | 001] > d[3][101] + w (3, 1)
    d[1][101] > 10 + 10
    ∞ > 20 => d[1][101] = 20
    q = {(10, (1, 011)), (12, (1, 111)), (16, (1, 111)),
    (20, (1, 101))}



  - 2:
    d[2][101 | 010] > d[3][101] + w (3, 2)
    d[2][111] > 10 + 1
    7 > 11=> No.

- 1’s neighbors, mask = 011:
  q = {(12, (1, 111)), (16, (1, 111)), (20, (1, 101))}
  - 2:
    d[2][011 | 010] > d[1][011] + w (1, 2)
    d[2][011] > 10 + 5
    5 > 15 => No.
  - 3:
    d[3][011 | 100] > d[1][011] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 111:
  q = {(16, (1, 111)), (20, (1, 101))}
  - 2:
    d[2][111 | 010] > d[1][111] + w (1, 2)
    d[2][111] > 12 + 5
    7 > 17 => No.

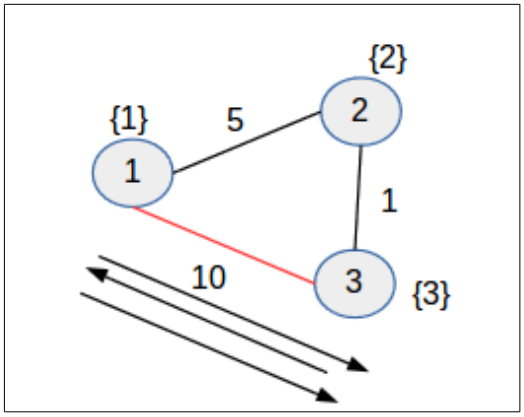
  - 3:
    d[3][111 | 100] > d[1][111] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 111:
  q = {(20, (1, 101))}
  - 2:
    d[2][111 | 010] > d[1][111] + w (1, 2)
    d[2][111] > 12 + 5
    7 > 17 => No.

  - 3:
    d[3][111 | 100] > d[1][111] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 101:
  q = {}
  - 2:
    d[2][101 | 010] > d[1][101] + w (1, 2)
    d[2][111] > 20 + 5
    7 > 25 => No.

  - 3:
    d[3][101 | 100] > d[1][101] + w (1, 3)
    d[3][101] > 20 + 10
    ∞ > 30=> d[3][101] = 30.
    q = {(30, (3, 101))}



- 3's neighbors, mask = 101:
  q = {}
  - 2:
    d[2][101 | 010] > d[3][101] + w (3, 2)
    d[2][111] > 30 + 1
    7 > 31 => No.

  - 1:
    d[1][101 | 001] > d[3][101] + w (3, 1)
    d[1][101] > 20 + 10
    20 > 30=> No.

q = ∅

The final d() values:

d							
	001	010	011	100	101	110	111
1	0	∞	10	∞	20	∞	~~16~~ 12
2	∞	∞	5	∞	∞	∞	7
3	∞	∞	∞	∞	10	∞	6

Some distances were not computed, because there is no need to compute some of them or impossible to compute them. Like $d[i][100]$, $(1 \leq i \leq 3)$, is impossible to compute because it's not possible to get fish type #3, without getting fish type #1 (we start at shop #1)

Simulate the two cats run into the $n * (1 << k)$ matrix to compute the minimum amount of time it will take for the cats to collectively purchase all k fish.

"*If one cat finishes shopping before the other, he waits at shopping center n for his partner to finish; this means that the total shopping time is the maximum of Little and Big Cats' respective shopping times.*"

Now the actual problem has two cats. But that doesn't make much difference as we know shortest distance to reach n^{th} shopping center with any of the k fishes. So why not try all combinations i.e. mask $m1$ and $m2$ such that $m1$ *or* $m2 = k$ and we take the minimum for all such combos.

Little & Big Cat, will run throw all the matrix of the d() values and when they collect all the k type of fishes and reach the shopping center n, will take the minimum of the maximum of Little and Big Cat's respective shopping times.

```
BestTime ← ∞
For m1 ∈ [1, 1 << k ]
    for m2  [cat1, 1 << k ]
        if (m1 | m2) = (1 << k) – 1
            bestTime ← min(bestTime, max(d[n][m1], d[n][m2]))
```

Executing this pseudo-code on the d() values matrix, the computed $bestTime$ is equal to 6

C++ Code: $O(MlogN \cdot 2^k)$

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e+9;
const int MAX_NODES = 1000 + 10;
int n, m, k;

int **dist = NULL;
int *masks = NULL;
vector<pair<int, int> > *adj = NULL;

void createAdjList(void) {
  for (int edge = 0 ; edge < m ; ++edge) {
    int x , y , z;
    cin >> x >> y >> z;
    x--;
    y--;
    adj[x].push_back(make_pair(y, z));
    adj[y].push_back(make_pair(x, z));
  }
}
```

```
void dijkstra(int start) {
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < (1 << k); ++j)
      dist[i][j] = INF;

  dist[start][masks[start]] = 0;

  priority_queue<pair<int, pair<int, int> >,
    vector<pair<int, pair<int, int> > >,
    greater<pair<int, pair<int, int> > > > q;

  q.push({dist[start][masks[start]], {start, masks[start]}});
  while (!q.empty()){
    int current = q.top().second.first;
    int currentMask = q.top().second.second;

    q.pop();

    for (auto neighbors: adj[current]) {
      int v = neighbors.first;
      int w = neighbors.second;
      if (dist[v][currentMask | masks[v]] > dist[current][currentMask] + w ) {
        dist[v][currentMask | masks[v]] = dist[current][currentMask] + w;
        q.push( {dist[v][currentMask | masks[v]], {v, currentMask | masks[v]}} );

      }

    }

  }
}
```

```
int main () {
  ios_base::sync_with_stdio(false);

  cin >> n >> m >> k;

  dist = new int*[n];
  for (int i = 0 ; i < n ; ++i)
    dist[i] = new int[1 << k];

  adj = new vector<pair<int, int> >[n];
  masks = new int[n];

  for(int i = 0; i < n; ++i) {
    int ti;
    cin >> ti;

    for(int j = 1; j <= ti; ++j) {
      int ai;
      cin >> ai;

      masks[i] |= (1 << (ai - 1));
    }
  }

  createAdjList();


  dijkstra(0);

  int ans = INF;
  for(int i = 0; i < (1 << k); ++i)
    for(int j = i; j < (1 << k); ++j)
      if ((i | j) == ((1 << k) - 1))
        ans = min(ans, max(dist[n-1][i], dist[n-1][j]));

  cout << ans << endl;

  return 0;
}
```

References:

https://www.quora.com/How-do-I-solve-synchronous-shopping-graph-problem-from-HackerRank

https://www.hackerrank.com/challenges/synchronous-shopping/editorial