# Graph

The graph problem covers many topics, for example, graph traverse, shortest distance, topology sort, and even union find.

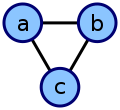
We will start from the definition of the graph. The graph is a set of vertexes and edges, and there may be weight on every edge.

<https://en.wikipedia.org/wiki/Graph_(abstract_data_type)>

The graph can be direct or undirect.

There are many ways to represent a graph, for example list (linked node), adjacency matrix, indigence matrix.

**Adjacency list**



This undirected cyclic graph can be described by the three unordered lists {b, c}, {a, c}, {a, b}.

**Indigence matrix**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | **e1** | **e2** | **e3** | **e4** | | **1** | 1 | 1 | 1 | 0 | | **2** | 1 | 0 | 0 | 0 | | **3** | 0 | 1 | 0 | 1 | | **4** | 0 | 0 | 1 | 1 | |  | {\displaystyle {\begin{pmatrix}1&1&1&0\\1&0&0&0\\0&1&0&1\\0&0&1&1\\\end{pmatrix}}.} |

To handle the graph problem easily, I prefer the following two data structures:

1. A hashed table with vertex id as key and a set of neighbours where the vertex can reach as the value. For weighted graph, the value should be a hash map.
2. A two dimension of matrix (in C++, it is vector<vector<int>> G), if vertex i can reach vertex j then G[i][j] = 1, otherwise G[i][j] =0. For weighted graph the value is the weight.

The first data structure has a lot of advantage to calculate the following:

1. Whether vertex i and j are connected.
2. How many neighbors does the vertex i have.
3. Add an edge
4. Remove an edge.
5. BFS traverse the graph from any vertex.

The second data structure is used when there is only limited functionality is required and fast access is needed, consider vector access is almost 40 times faster than the hash table.

## Traverse the Graph

There are three typical ways to traverse the graph. BFS, DFS and shortest path (Dijkstra's algorithm).

There are many other shortest path algorithms, for example Floyd–Warshall algorithm, O(V^3)) and Bellman Ford algorithm, O(E^3), but Dijkstra is the most popular one.

Between BFS and DFS, when the question such as shortest step is asked, then BFS should be used. This is why BFS is far more popular in leet code solutions than DFS.

The common algorithm code pattern you may need to practice a lot are listed below:

1. Flood fill
2. BFS
3. DFS
4. Topology Sort
5. Dijkstra
6. Union Find
7. Balance Weight
8. Travelling Salesman
9. MinMax Problem

The last two are normally beyond the scope of interview.

## Travelling Salesman

<https://simple.wikipedia.org/wiki/Travelling_salesman_problem>

The travelling salesman is to say for a list of given cites, the distances between the cities, the sales man need to travel all the cities at least once, what is the shortest way.

This an NP hard problem, so the time complexity is exponential. We should make our code pattern clean, short and easy to understand.

Please think how you handle the following issues in this problem.

1. How to search the path?

This can be a Breadth First Search, but it will not search once, you need to start from every city. But generally speaking, it is similar.

1. How to mark the current state?

During the search you need to represent what is the current state. For travelling salesman problem, the current state is which city the salesman currently reached (last visited), plus all the cities the salesman has already visited.

1. How to mark the cities already travelled?

We can use a bitmap, (a long integer will cover for 64), each bit represents the city is already visited or not.

1. How to optimize the algorithm by cutting branches?

For any state above, you only need to remember the shortest distance.

The following is an example:

### **847. Shortest Path Visiting All Nodes**

An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph.

graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

**Example 1:**

**Input:** [[1,2,3],[0],[0],[0]]

**Output:** 4

**Explanation**: One possible path is [1,0,2,0,3]

**Example 2:**

**Input:** [[1],[0,2,4],[1,3,4],[2],[1,2]]

**Output:** 4

**Explanation**: One possible path is [0,1,4,2,3]

**Note:**

1. 1 <= graph.length <= 12
2. 0 <= graph[i].length < graph.length

/// <summary>

/// Leet code #847. Shortest Path Visiting All Nodes

///

/// An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1)

/// is given as graph.

///

/// graph.length = N, and j != i is in the list graph[i] exactly once, if

/// and only if nodes i and j are connected.

///

/// Return the length of the shortest path that visits every node. You may

/// start and stop at any node, you may revisit nodes multiple times, and

/// you may reuse edges.

///

/// Example 1:

///

/// Input: [[1,2,3],[0],[0],[0]]

/// Output: 4

/// Explanation: One possible path is [1,0,2,0,3]

///

/// Example 2:

///

/// Input: [[1],[0,2,4],[1,3,4],[2],[1,2]]

/// Output: 4

/// Explanation: One possible path is [0,1,4,2,3]

///

///

/// Note:

/// 1. 1 <= graph.length <= 12

/// 2. 0 <= graph[i].length < graph.length

/// </summary>

int LeetCode::shortestPathLength(vector<vector<int>>& graph)

{

size\_t size = graph.size();

vector<vector<int>> dp(1 << size, vector<int>(size, -1));

queue<pair<int, int>> search;

// we start from each node

for (size\_t i = 0; i < size; i++)

{

int cover = 1 << i;

dp[cover][i] = 0;

// make the current node as last node

search.push(make\_pair(cover, i));

}

while (!search.empty())

{

pair<int, int> route = search.front();

search.pop();

// we check from current last node what are next nodes to travel

for (size\_t i = 0; i < graph[route.second].size(); i++)

{

int next = graph[route.second][i];

// Normally we need to check if we have already visited the

// next node, however the distance check will save us this

// effort, if the node is already visited, visit again will

// give us no chance to shorten the path

int cover = route.first | (1 << next);

if (dp[cover][next] == -1 ||

dp[cover][next] > dp[route.first][route.second] + 1)

{

dp[cover][next] = dp[route.first][route.second] + 1;

search.push(make\_pair(cover, next));

}

}

}

int full = (1 << size) - 1;

int result = dp[full][0];

// starting from every node, look for shortest path

for (size\_t i = 1; i < size; i++)

{

result = min(result, dp[full][i]);

}

return result;

}

## MinMax Problem

<https://en.wikipedia.org/wiki/Minimax>

The MinMax problem is not a sorting problem, it is a gaming problem, which is to say assuming two people are playing a game, for example chess, every one will choose the best strategy for himself, then what will be the end result?

The key question to ask in a MinMax Problem is very similar to a saleman problem (may be to every graph problem. 😊). This is to say how to represent the current state.

In most of the cases, it is where we are (each player) plus who should move next. So it can be G(S(a), S(b), M), S(a) is where play A is and M is who should move next.

To solve the problem we can start from some sure win state for one party, assume it is play A, then reversely search for all possible states which can directly transit to this state, if on those state it is A will move next, then it is also a A sure win state, otherwise, if all possible B move will end up with a A sure win state, then these states are also A sure win state. Same logic can be applied on play B.

Let’s look at an example:

### **913. Cat and Mouse**

A game on an **undirected** graph is played by two players, Mouse and Cat, who alternate turns.

The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph.

Mouse starts at node 1 and goes first, Cat starts at node 2 and goes second, and there is a Hole at node 0.

During each player's turn, they **must** travel along one edge of the graph that meets where they are.  For example, if the Mouse is at node 1, it **must** travel to any node in graph[1].

Additionally, it is not allowed for the Cat to travel to the Hole (node 0.)

Then, the game can end in 3 ways:

* If ever the Cat occupies the same node as the Mouse, the Cat wins.
* If ever the Mouse reaches the Hole, the Mouse wins.
* If ever a position is repeated (ie. the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.

Given a graph, and assuming both players play optimally, return 1 if the game is won by Mouse, 2 if the game is won by Cat, and 0 if the game is a draw.

**Example 1:**

**Input:** [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

**Output:** 0

**Explanation:**

4---3---1

|   |

2---5

 \ /

  0

**Note:**

1. 3 <= graph.length <= 50
2. It is guaranteed that graph[1] is non-empty.
3. It is guaranteed that graph[2] contains a non-zero element.

In this probkem, the state S can be represented as S(c,m,t), c and m are the current position of cat and mouse and t is the who should move next.

Let’s starts with some sure win scenario. S(0, \*, \*) is a sure win state for mouse, this ie because mouse enter the mouse hole, so regardless where cat is and who will move it is mouse win.

We can write S(0,\*,\*) = MOUSE.

This is to say if the mouse is at 2 and 5 in the above graph and it is mouse move, mouse will also win. S(2, \*, ‘m’) = MOUSE and S(5, \*, ‘m’) = MOUSE

On the other hand, if mouse and cat are in the same location, it is CAT win, regardless who will move next. So S(x,x,\*) = CAT. This means if it is the cat move, and cat can reach mouse place, it it is cat win for example S(1,3,’c’) = CAT, S(3,5,’c’) = CAT, S(4,2,’c’) = CAT…

There is another situation not that obvious, assume the CAT is at 3 and mouse is at 1, and it is mouse move. For all the possible mouse moves (in this case only one), it will end up in CAT’s location, so CAT win. S(1,3,’m’) = CAT.

The algorithm to reach this conclusion is that start from S(1,3, ’m’), play all possible mouse moves, if it is CAT win, take this path out from existing path, when no more path left it is your rival, CAT win. If after such play, there are still some paths left, it is a DRAW scenario (no one win).

Let’s use a simplified scenario as below:



/// <summary>

/// Leet code #913. Cat and Mouse

///

/// A game on an undirected graph is played by two players, Mouse and Cat,

/// who alternate turns.

///

/// The graph is given as follows: graph[a] is a list of all nodes b such

/// that ab is an edge of the graph.

///

/// Mouse starts at node 1 and goes first, Cat starts at node 2 and goes

/// second, and there is a Hole at node 0.

///

/// During each player's turn, they must travel along one edge of the graph

/// that meets where they are. For example, if the Mouse is at node 1, it

/// must travel to any node in graph[1].

///

/// Additionally, it is not allowed for the Cat to travel to the Hole (node 0.)

///

/// Then, the game can end in 3 ways:

///

/// If ever the Cat occupies the same node as the Mouse, the Cat wins.

/// If ever the Mouse reaches the Hole, the Mouse wins.

/// If ever a position is repeated (ie. the players are in the same position

/// as a previous turn, and it is the same player's turn to move), the game is

/// a draw.

/// Given a graph, and assuming both players play optimally, return 1 if the

/// game is won by Mouse, 2 if the game is won by Cat, and 0 if the game is a

/// draw.

///

/// Example 1:

///

/// Input: [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

/// Output: 0

/// Explanation:

/// 4---3---1

/// | |

/// 2---5

/// \ /

/// 0

///

/// Note:

///

/// 1. 3 <= graph.length <= 50

/// 2. It is guaranteed that graph[1] is non-empty.

/// 3. It is guaranteed that graph[2] contains a non-zero element.

/// </summary>

int LeetCode::catMouseGame(vector<vector<int>>& graph)

{

int N = graph.size();

vector<vector<vector<int>>> color(N, vector<vector<int>>(N, vector<int>(2)));

vector<vector<vector<int>>> degree(N, vector<vector<int>>(N, vector<int>(2)));

enum COLOR { DRAW = 0, MOUSE = 1, CAT = 2 };

// For every mouse and cat locations, it is N^2

for (int m = 0; m < N; ++m)

{

for (int c = 1; c < N; ++c)

{

// initialize all possible path count for mouse

degree[m][c][0] = graph[m].size();

// initialize all possible path count for cat

degree[m][c][1] = graph[c].size();

for (int x : graph[c])

{

// because CAT can not go to hole, deduct path to zero

if (x == 0)

{

degree[m][c][1]--;

break;

}

}

}

}

// Initialize all the scenarios where mouse will win or cat will win

queue<vector<int>> search;

for (int i = 1; i < N; ++i)

{

// for all the move

for (int t = 0; t < 2; ++t)

{

// mark mouse win

color[0][i][t] = MOUSE;

search.push({ 0, i, t });

cout << 0 << i << t << MOUSE << endl;

// mark CAT win

if (i > 0)

{

color[i][i][t] = CAT;

search.push({i, i, t});

cout << i << i << t << CAT << endl;

}

}

}

// This is a BFS search, we initialize some sure win state for one party

// in the queue, and reversly search more sure win state and mark them.

// After the search complete, we may have some undetermined state, this

// means DRAW state.

while (!search.empty())

{

vector<int> node = search.front();

search.pop();

// for nodes that are colored :

int m = node[0], c = node[1], t = node[2];

int state = color[m][c][t];

// The children contains all the possible path coming to current state.

vector<vector<int>> children;

// mouse step, iterate all possible cat paths coming here

if (t == 0)

{

for (int c1 : graph[c])

{

if (c1 > 0)

{

children.push\_back({ m, c1, 1 - t });

}

}

}

// cat step, iterate all possible cat paths coming here

else

{

for (int m1 : graph[m])

{

children.push\_back({ m1, c, 1 - t });

}

}

for (auto child : children)

{

int m = child[0], c = child[1], t = child[2];

// by default it is all DRAW state.

// when it is not decide, it is also DRAW state

// so this is to resolve everything not resolved yet

// remeber we only add a sure win state to the BFS search queue

if (color[m][c][t] == DRAW)

{

// This means current move is me and next state is my

// win, Then current state is my win, for example if

// state == 1, which is mouse win and previous move t = 0

// which is mouse move, mouse will definitely choose this path

// same applies when state == 2 and t = 1 when it is cat's turn.

if (state == t + 1)

{

color[m][c][t] = state;

search.push({ m, c, t });

}

// Otherwise if current winning state and previous move are

// different, which is to say the rival should avoid this path,

// This means it should cut this path from choice, but if

// all the paths are dead, then means the previous state is a

// dead state for the rival, we can mark it.

// But iff there are remaing paths, leave it for DRAW.

else

{

degree[m][c][t]--;

if (degree[m][c][t] == 0)

{

color[m][c][t] = state;

search.push({ m, c, t });

}

}

}

}

}

return color[1][2][0];

}

## Balance Weight

Assuming you have a graph, which is a series of line segments parallel to the X axis and each edge has a weight. Can you find a point with weighted distance to left end and right end equal or most close to equal?

There are many methods to solve the problem, one way is to have two pointers from left and right moving to the center, each start one step, if total weighted distance on left is less than the one on right, we move left pointer one step further, otherwise we move right pointer one step further.

Please note one important fact, the weight sum from the total objects on one side, it has nothing to do with distance. While the total cost is SUM(weight \* distance).

In the following example the weight is number of people. You can see if right end has two people but left side has one, we should keep on move the meeting point to right regardless how far from it is from the left.

### **296. Best Meeting Point**

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using [Manhattan Distance](http://en.wikipedia.org/wiki/Taxicab_geometry), where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

**Example:**

**Input:**

1 - 0 - 0 - 0 - 1

| | | | |

0 - 0 - 0 - 0 - 0

| | | | |

0 - 0 - 1 - 0 - 0

**Output: 6**

**Explanation:** Given three people living at (0,0), (0,4), and (2,2):

  The point (0,2) is an ideal meeting point, as the total travel distance

  of 2+2+2=6 is minimal. So return 6.

To find out a best meeting point for multiple people, you need to use the method called weighted distance. Basically, the best meeting point for two persons are the middle point between them, but when you have multiple people and in a 2D grid you should use weighted distance. The weight is the number of people, to find out the fairest meeting point, you can use two pointers, and move from the light side to the heavy.

There is a catch in the weighted distance, in case there is any obstacle in the route, weighted distance may fail. This is why #317 cannot use this method.

/// <summary>

/// Leet code #296. Best Meeting Point

/// </summary>

int LeetCode::minTotalDistance(vector<int> & nums)

{

if (nums.empty()) return 0;

int i = 0, j = nums.size() - 1;

// Here the left and right are the sum of people considered as weight,

// which has nothing to do with distance.

int left = nums[i], right = nums[j], sum = 0;

while (i < j)

{

if (left < right)

{

sum += left;

i++;

left += nums[i];

}

else

{

sum += right;

j--;

right += nums[j];

}

}

return sum;

}

/// <summary>

/// Leet code #296. Best Meeting Point

///

/// A group of two or more people wants to meet and minimize the total travel

/// distance. You are given a 2D grid of values 0 or 1, where each 1 marks

/// the home of someone in the group. The distance is calculated using

/// Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

/// For example, given three people living at (0,0), (0,4), and (2,2):

/// 1 - 0 - 0 - 0 - 1

/// | | | | |

/// 0 - 0 - 0 - 0 - 0

/// | | | | |

/// 0 - 0 - 1 - 0 - 0

/// The point (0,2) is an ideal meeting point, as the total travel distance of

/// 2+2+2=6 is minimal. So return 6.

///

/// Hint:

/// 1.Try to solve it in one dimension first. How can this solution apply to

/// the two dimension case?

/// </summary>

int LeetCode::minTotalDistance(vector<vector<int>>& grid)

{

if (grid.empty() || grid[0].empty()) return 0;

vector<int> row(grid.size());

vector<int> col(grid[0].size());

for (size\_t i = 0; i < grid.size(); i++)

{

for (size\_t j = 0; j < grid[i].size(); j++)

{

if (grid[i][j] == 1)

{

row[i]++;

col[j]++;

}

}

}

return minTotalDistance(row) + minTotalDistance(col);

}

/// <summary>

/// Leet code #902. Truck delivery

///

/// There are a number of houses in a line, the distance between two houses

/// are given in an array.

/// for example the distance between house(0) and house(1) is in distance[0]

/// and the distance between house(n-2) and house(n-1) is given in

/// distance[n-2]. A delivery truck stops in front of the houses, and the

/// driver has a number of packages to deliver, each with a weight[i],

/// if no package for this house it is 0.

/// The truck can only stop in front of one house, the driver should minimize

/// the total cost of the delivery.

///

/// Note:

/// 1. The cost is defined by multiply the distance the driver should walk

/// with the weight of package in hand, i.e.

/// It is weight[i] \* distance[i].

/// 2. If the driver walk back with empty hand the cost is zero.

/// 3. The driver is only allowed to carry one package at a time.

/// </summary>

int LeetCode::minDeliveryCost(vector<int> distances, vector<int> weight)

{

// at very beginning, we assume the truck stops at house0, so the

// weight of total packages on left is 0 and the total weight

// on right is from 1 to n-1.

int weight\_left = 0, weight\_right = 0;

int distance = 0, sum = 0, min\_cost = INT\_MAX;

for (size\_t i = 1; i < weight.size(); i++)

{

weight\_right += weight[i];

distance += distances[i];

sum += weight[i] \* distance;

}

min\_cost = sum;

for (size\_t i = 1; i < weight.size(); i++)

{

// assume truck is at house i, house[i-1] is moved to left

weight\_left += weight[i - 1];

// on the right side, the cost to each house deduct the cost of weight \* distance[i-1];

sum -= weight\_right \* distances[i];

// on the left side, the cost to each house add (weight \* distance[i-1]

sum += weight\_left \* distances[i];

// house[i] is no longer on right, it is in front of track

weight\_right -= weight[i];

// calculate min\_cost

min\_cost = min(sum, min\_cost);

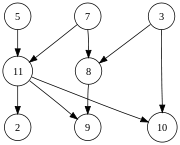
}

return min\_cost;

}

## Topology Sort

The topology sort is one of the most common problem in Graph Theory. The idea of the topology sort is that we traverse the graph in an order of dependency. We start all the nodes without dependencies, visit them, then remove the dependencies on them and select all the unvisited node without dependency as the next iteration.



For example, we can visit the above graph in the order of 3 iterations.

5,7,3,11,8,2,9,10

The code pattern of the topology sort is that you need to maintain two data structures. The first is the count of dependency on each vertex in the graph, which will help us to find the free nodes. The second is the mapping table of dependency, helping us to update the dependency count when a specific node is visited.

The processing logic is a BFS pattern.

Here is an example:

### **207. Course Schedule**

There are a total of *n* courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

**Example 1:**

**Input:** 2, [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0. So it is possible.

**Example 2:**

**Input:** 2, [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0, and to take course 0 you should

  also have finished course 1. So it is impossible.

**Note:**

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
2. You may assume that there are no duplicate edges in the input prerequisites.

/// <summary>

/// LeetCode #207. Course Schedule

/// There are a total of n courses you have to take, labeled from 0 to

/// n - 1.

/// Some courses may have prerequisites, for example to take course 0 you

/// have to first take course 1, which is expressed as a pair: [0,1]

/// Given the total number of courses and a list of prerequisite pairs,

/// is it possible for you to finish all courses?

///

/// For example:

/// 2, [[1,0]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0. So it is possible.

///

/// 2, [[1,0],[0,1]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0,

/// and to take course 0 you should also have finished course 1. So it

/// is impossible.

/// Note:

/// The input prerequisites is a graph represented by a list of edges,

/// not adjacency matrices. Read more about how a graph is represented.

/// </summary>

bool LeetCode::canFinishCourse(int numCourses, vector<pair<int, int>>& prerequisites)

{

vector<int> courseDependency(numCourses);

vector<unordered\_set<int>> whoDependOnMe(numCourses);

unordered\_set<int> freeCourse;

queue<int> search\_queue;

// remember which course dependes on others and which ones depends on me

for (pair<int, int> pair : prerequisites)

{

if (whoDependOnMe[pair.second].count(pair.first) == 0)

{

courseDependency[pair.first]++;

whoDependOnMe[pair.second].insert(pair.first);

}

}

// get all the course not depends on others, this is our starting search scope

for (size\_t i = 0; i < courseDependency.size(); i++)

{

if (courseDependency[i] == 0)

{

freeCourse.insert(i);

search\_queue.push(i);

}

}

// Using queue to manage BFS and get every free course and clear the

// dependency with a free course, i.e. you depend on a free course,

// then such dependency

// does not matter. If all dependencies are clear, we got a new

// free course

while (!search\_queue.empty())

{

int free\_course = search\_queue.front();

search\_queue.pop();

unordered\_set<int> course\_depend\_on\_me = whoDependOnMe[free\_course];

for (int dependent\_course : course\_depend\_on\_me)

{

courseDependency[dependent\_course]--;

if (courseDependency[dependent\_course] == 0)

{

freeCourse.insert(dependent\_course);

search\_queue.push(dependent\_course);

}

}

}

// if number of free courses equals to the total course, we can finish

// all courses

if (freeCourse.size() == numCourses)

{

return true;

}

else

{

return false;

}

}

### **269. Alien Dictionary**

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of **non-empty** words from the dictionary, where **words are sorted lexicographically by the rules of this new language**. Derive the order of letters in this language.

**Example 1:**

**Input:**

[

"wrt",

"wrf",

"er",

"ett",

"rftt"

]

**Output:** "wertf"

**Example 2:**

**Input:**

[

"z",

"x"

]

**Output:** "zx"

**Example 3:**

**Input:**

[

"z",

"x",

"z"

]

**Output:** ""

**Explanation:** The order is invalid, so return "".

**Note:**

1. You may assume all letters are in lowercase.
2. You may assume that if a is a prefix of b, then a must appear before b in the given dictionary.
3. If the order is invalid, return an empty string.
4. There may be multiple valid order of letters, return any one of them is fine.

/// <summary>

/// Leet code #269. Alien Dictionary

/// There is a new alien language which uses the latin alphabet. However, the

/// order among letters are unknown to you. You receive a list of words from

/// the dictionary, where words are sorted lexicographically by the rules of

/// this new language. Derive the order of letters in this language.

///

/// For example,

/// Given the following words in dictionary,

/// [

/// "wrt",

/// "wrf",

/// "er",

/// "ett",

/// "rftt"

/// ]

/// The correct order is: "wertf".

/// Note:

/// 1.You may assume all letters are in lowercase.

/// 2.If the order is invalid, return an empty string.

/// 3.There may be multiple valid order of letters, return any one of them is

/// fine.

/// </summary>

string LeetCode::alienOrder(vector<string>& words)

{

string result;

unordered\_map<char, unordered\_set<char>> next\_set;

unordered\_map<char, unordered\_set<char>> prev\_set;

unordered\_set<char> root\_set;

for (size\_t i = 0; i < words.size(); i++)

{

root\_set.insert(words[i].begin(), words[i].end());

if (i > 0)

{

string prev\_word = words[i - 1];

string curr\_word = words[i];

for (size\_t i = 0; i < max(prev\_word.size(), curr\_word.size()); i++)

{

if (i == prev\_word.size()) break;

else if (i == curr\_word.size()) return "";

else

{

if (prev\_word[i] != curr\_word[i])

{

next\_set[prev\_word[i]].insert(curr\_word[i]);

prev\_set[curr\_word[i]].insert(prev\_word[i]);

break;

}

}

}

}

}

// remove all characters has predecessor

for (auto itr : prev\_set) root\_set.erase(itr.first);

while (!root\_set.empty())

{

char prev\_ch = \*root\_set.begin();

for (char next\_ch : next\_set[prev\_ch])

{

prev\_set[next\_ch].erase(prev\_ch);

if (prev\_set[next\_ch].empty())

{

root\_set.insert(next\_ch);

prev\_set.erase(next\_ch);

}

}

root\_set.erase(prev\_ch);

result.push\_back(prev\_ch);

next\_set.erase(prev\_ch);

}

if (next\_set.empty()) return result;

else return "";

}

## BFS or DFS

When we traverse the graph, normally we have two options, one is the BFS another is DFS. When it is required to find minimum steps for the solution, BFS will be the choice, because it will always give you the minimum hop in the path.

The BFS algorithm is based on a queue data structure.

### **815. Bus Routes**

We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.

We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number of buses we must take to reach our destination? Return -1 if it is not possible.

**Example:**

**Input:**

routes = [[1, 2, 7], [3, 6, 7]]

S = 1

T = 6

**Output:** 2

**Explanation:**

The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Note:**

* 1 <= routes.length <= 500.
* 1 <= routes[i].length <= 500.
* 0 <= routes[i][j] < 10 ^ 6.

/// <summary>

/// Leet code #815. Bus Routes

///

/// We have a list of bus routes. Each routes[i] is a bus route that the

/// i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this

/// means that the first bus (0-th indexed) travels in the sequence

/// 1->5->7->1->5->7->1->... forever.

///

/// We start at bus stop S (initially not on a bus), and we want to go to

/// bus stop T. Travelling by buses only, what is the least number of

/// buses we must take to reach our destination? Return -1 if it is not

/// possible.

///

/// Example:

/// Input:

/// routes = [[1, 2, 7], [3, 6, 7]]

/// S = 1

/// T = 6

/// Output: 2

/// Explanation:

/// The best strategy is take the first bus to the bus stop 7, then take

/// the second bus to the bus stop 6.

/// Note:

///

/// 1. 1 <= routes.length <= 500.

/// 2. 1 <= routes[i].length <= 500.

/// 3. 0 <= routes[i][j] < 10 ^ 6.

/// </summary>

int LeetCode::numBusesToDestination(vector<vector<int>>& routes, int S, int T)

{

unordered\_map<int, set<int>> stop\_map;

unordered\_map<int, int> dist\_map;

for (size\_t i = 0; i < routes.size(); i++)

{

for (size\_t j = 0; j < routes[i].size(); j++)

{

stop\_map[routes[i][j]].insert(i);

}

}

queue<int> stop\_queue;

stop\_queue.push(S);

dist\_map[S] = 0;

while (!stop\_queue.empty())

{

int stop = stop\_queue.front();

stop\_queue.pop();

if (stop == T) return dist\_map[T];

for (int bus : stop\_map[stop])

{

for (int next : routes[bus])

{

if ((dist\_map.count(next) > 0) && (dist\_map[next] <= dist\_map[stop] + 1))

{

continue;

}

stop\_queue.push(next);

dist\_map[next] = dist\_map[stop] + 1;

stop\_map[next].erase(bus);

}

}

stop\_map[stop].clear();

}

return -1;

}

## Flood Fill

Flood Fill is a special BFS or say Union Find, which easily merge all the adjacent nodes in an area with same property. In some algorithm book, this is also called as coloring. This algorithm is used to calculate the number of points in an area.

The code pattern is also BFS based, with a 2 D array to record the color, and an array of count which remember the total points in each area.

### **924. Minimize Malware Spread**

In a network of nodes, each node i is directly connected to another node j if and only if graph[i][j] = 1.

Some nodes initial are initially infected by malware.  Whenever two nodes are directly connected and at least one of those two nodes is infected by malware, both nodes will be infected by malware.  This spread of malware will continue until no more nodes can be infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network, after the spread of malware stops.

We will remove one node from the initial list.  Return the node that if removed, would minimize M(initial).  If multiple nodes could be removed to minimize M(initial), return such a node with the smallest index.

Note that if a node was removed from the initial list of infected nodes, it may still be infected later as a result of the malware spread.

**Example 1:**

**Input:** graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

**Output:** 0

**Example 2:**

**Input:** graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]

**Output:** 0

**Example 3:**

**Input:** graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]

**Output:** 1

**Note:**

1. 1 < graph.length = graph[0].length <= 300
2. 0 <= graph[i][j] == graph[j][i] <= 1
3. graph[i][i] = 1
4. 1 <= initial.length < graph.length
5. 0 <= initial[i] < graph.length

/// <summary>

/// Leet code #924. Minimize Malware Spread

///

/// In a network of nodes, each node i is directly connected to another node j

/// if and only if graph[i][j] = 1.

///

/// Some nodes initial are initially infected by malware. Whenever two nodes

/// are directly connected and at least one of those two nodes is infected by

/// malware, both nodes will be infected by malware. This spread of malware

/// will continue until no more nodes can be infected in this manner.

///

/// Suppose M(initial) is the final number of nodes infected with malware in

/// the entire network, after the spread of malware stops.

///

/// We will remove one node from the initial list. Return the node that if

/// removed, would minimize M(initial). If multiple nodes could be removed to

/// minimize M(initial), return such a node with the smallest index.

///

/// Note that if a node was removed from the initial list of infected nodes,

/// it may still be infected later as a result of the malware spread.

///

/// Example 1:

/// Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

/// Output: 0

///

/// Example 2:

/// Input: graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]

/// Output: 0

///

/// Example 3:

/// Input: graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]

/// Output: 1

///

/// Note:

///

/// 1. 1 < graph.length = graph[0].length <= 300

/// 2. 0 <= graph[i][j] == graph[j][i] <= 1

/// 3. graph[i][i] = 1

/// 4. 1 <= initial.length < graph.length

/// 5. 0 <= initial[i] < graph.length

/// </summary>

int LeetCode::minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial)

{

vector<int> flag(graph.size(), -1);

vector<int> count(graph.size());

for (size\_t i = 0; i < initial.size(); i++)

{

queue<int> search;

search.push(initial[i]);

if (flag[initial[i]] != -1)

{

count[flag[initial[i]]] = 0;

continue;

}

while (!search.empty())

{

int node = search.front();

search.pop();

for (size\_t j = 0; j < graph[node].size(); j++)

{

if (graph[node][j] == 1 && flag[j] == -1)

{

flag[j] = initial[i];

count[initial[i]] += 1;

search.push(j);

}

}

}

}

int result = initial[0];

int reduce = 0;

for (size\_t i = 0; i < initial.size(); i++)

{

if ((count[initial[i]] > reduce) ||

((count[initial[i]] == reduce) && (initial[i] < result)))

{

reduce = count[initial[i]];

result = initial[i];

}

}

return result;

}

### **928. Minimize Malware Spread II**

(This problem is the same as Minimize Malware Spread, with the differences bolded.)

In a network of nodes, each node i is directly connected to another node j if and only if graph[i][j] = 1.

Some nodes initial are initially infected by malware.  Whenever two nodes are directly connected and at least one of those two nodes is infected by malware, both nodes will be infected by malware.  This spread of malware will continue until no more nodes can be infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network, after the spread of malware stops.

We will remove one node from the initial list, **completely removing it and any connections from this node to any other node**.  Return the node that if removed, would minimize M(initial).  If multiple nodes could be removed to minimize M(initial), return such a node with the smallest index.

**Example 1:**

**Input:** graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

**Output:** 0

**Example 2:**

**Input:** graph = [[1,1,0],[1,1,1],[0,1,1]], initial = [0,1]

**Output:** 1

**Example 3:**

**Input:** graph = [[1,1,0,0],[1,1,1,0],[0,1,1,1],[0,0,1,1]], initial = [0,1]

**Output:** 1

**Note:**

1. 1 < graph.length = graph[0].length <= 300
2. 0 <= graph[i][j] == graph[j][i] <= 1
3. graph[i][i] = 1
4. 1 <= initial.length < graph.length
5. 0 <= initial[i] < graph.length

/// <summary>

/// Leet code #928. Minimize Malware Spread II

///

/// In a network of nodes, each node i is directly connected to another node j

/// if and only if graph[i][j] = 1.

///

/// Some nodes initial are initially infected by malware. Whenever two nodes

/// are directly connected and at least one of those two nodes is infected by

/// malware, both nodes will be infected by malware. This spread of malware

/// will continue until no more nodes can be infected in this manner.

///

/// Suppose M(initial) is the final number of nodes infected with malware in

/// the entire network, after the spread of malware stops.

///

/// We will remove one node from the initial list, completely removing it and

/// any connections from this node to any other node. Return the node that if

/// removed, would minimize M(initial). If multiple nodes could be removed to

/// minimize M(initial), return such a node with the smallest index.

///

/// Note that if a node was removed from the initial list of infected nodes,

/// it may still be infected later as a result of the malware spread.

///

/// Example 1:

/// Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

/// Output: 0

///

/// Example 2:

/// Input: graph = [[1,1,0],[1,1,1],[0,1,1]], initial = [0,1]

/// Output: 1

///

/// Example 3:

///

/// Input: graph = [[1,1,0,0],[1,1,1,0],[0,1,1,1],[0,0,1,1]], initial = [0,1]

/// Output: 1

///

/// Note:

///

/// 1. 1 < graph.length = graph[0].length <= 300

/// 2. 0 <= graph[i][j] == graph[j][i] <= 1

/// 3. graph[i][i] = 1

/// 4. 1 <= initial.length < graph.length

/// 5. 0 <= initial[i] < graph.length

/// </summary>

int LeetCode::minMalwareSpreadII(vector<vector<int>>& graph, vector<int>& initial)

{

unordered\_set<int> virus;

for (auto v : initial) virus.insert(v);

vector<int> flag(graph.size(), -1);

vector<int> count(graph.size());

for (size\_t i = 0; i < initial.size(); i++)

{

queue<int> search;

search.push(initial[i]);

if (flag[initial[i]] == -1)

{

flag[initial[i]] = initial[i];

count[initial[i]] += 1;

}

else if (flag[initial[i]] != -2)

{

count[flag[initial[i]]]--;

flag[initial[i]] = -2;

}

else

{

continue;

}

while (!search.empty())

{

int node = search.front();

search.pop();

for (size\_t j = 0; j < graph[node].size(); j++)

{

if (graph[node][j] == 1)

{

if ((flag[j] == -2) ||

(flag[j] == initial[i]) ||

(virus.count(j) > 0)) continue;

if (flag[j] == -1)

{

flag[j] = initial[i];

count[initial[i]] += 1;

}

else

{

count[flag[j]]--;

flag[j] = -2;

}

search.push(j);

}

}

}

}

int result = initial[0];

int reduce = 0;

for (size\_t i = 0; i < initial.size(); i++)

{

if ((count[initial[i]] > reduce) || ((count[initial[i]] == reduce) && (initial[i] < result)))

{

reduce = count[initial[i]];

result = initial[i];

}

}

return result;

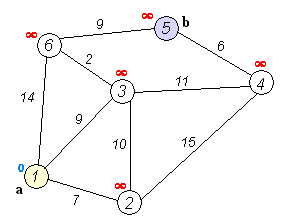
}

## Dijkstra Shortest Distance

The Dijkstra Shortest Distance algorithm is applied on a weighted-edge graph to find the shortest path from one vertex to another. It is guaranteed to visit each edge once. So it is also O(n^2), n is vertices. However, the algorithm does not store the result from any node to any node, only calculate the shortest distance from source to destination.

Another common shortest distance algorithm is Floyd-Warshall Algorithm, which is O(n^3, but it record shortest distance from any to any node.

The code pattern of Dijkstra is based on BFS and a priority queue, by default we assume the distance from the source node to all the node are infinite, we start from the source node and search for its neighbors, starting from the shortest path, then use the neighbor as the intermediate point and check the next hop and update the shorted distance from the source to these new locations. Every time if find a shorter path from S to N, we will overwrite the old value. and assume the N as the new searching point.



### **505. The Maze II**

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up**, **down**, **left** or **right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's **start position**, the **destination** and the **maze**, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of **empty spaces** traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

**Example 1:**

**Input 1:** a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

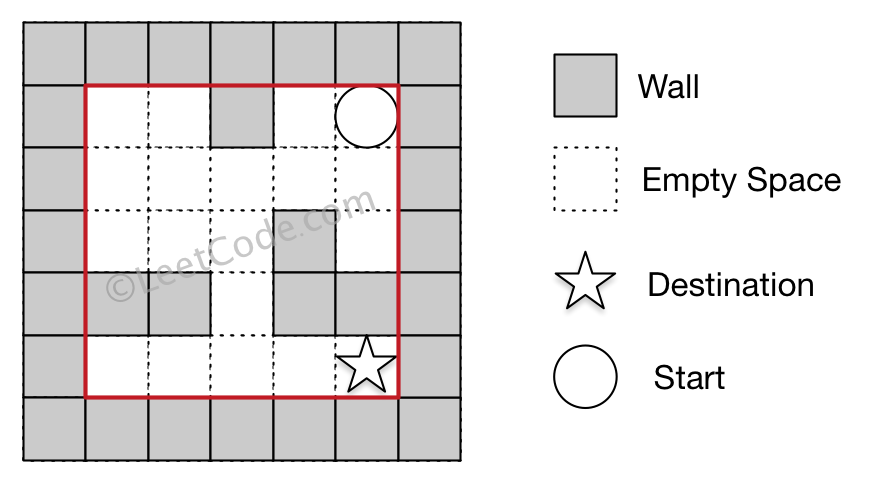
**Input 2:** start coordinate (rowStart, colStart) = (0, 4)

**Input 3:** destination coordinate (rowDest, colDest) = (4, 4)

**Output:** 12

**Explanation:** One shortest way is : left -> down -> left -> down -> right -> down -> right.

The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.



**Example 2:**

**Input 1:** a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

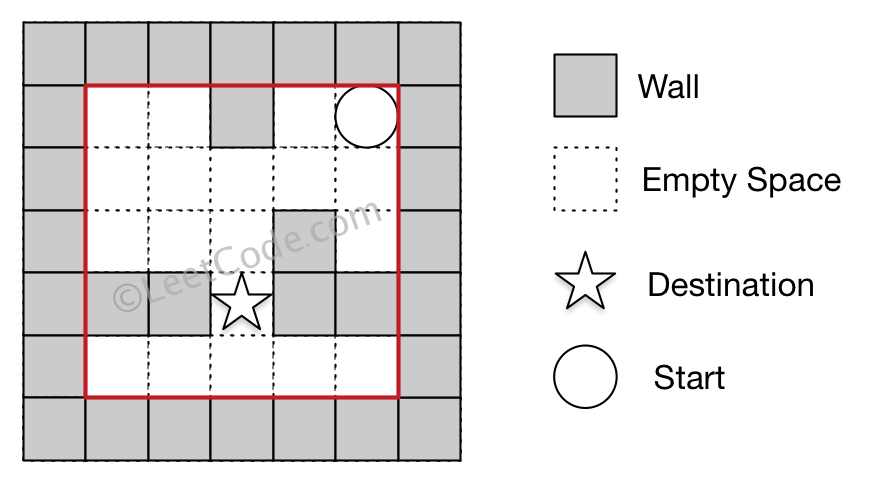
0 0 0 0 0

**Input 2:** start coordinate (rowStart, colStart) = (0, 4)

**Input 3:** destination coordinate (rowDest, colDest) = (3, 2)

**Output:** -1

**Explanation:** There is no way for the ball to stop at the destination.



**Note:**

1. There is only one ball and one destination in the maze.
2. Both the ball and the destination exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

/// <summary>

/// Leet code #505. The Maze II

/// </summary>

void LeetCode::shortestDistance(vector<vector<int>>& maze, vector<vector<int>>& visited, vector<int>& start,

priority\_queue<pair<int, vector<int>>> &process\_queue)

{

vector<vector<int>> next\_list;

int distance = visited[start[0]][start[1]];

for (size\_t i = 0; i < 4; i++)

{

int step = 0;

vector<int> pos = start;

if (i == 0)

{

while ((pos[0] > 0) && (maze[pos[0] - 1][pos[1]] == 0))

{

step++;

pos[0]--;

}

}

else if (i == 1)

{

while ((pos[0] < (int)maze.size() - 1) && (maze[pos[0] + 1][pos[1]] == 0))

{

step++;

pos[0]++;

}

}

else if (i == 2)

{

while ((pos[1] > 0) && (maze[pos[0]][pos[1] - 1] == 0))

{

step++;

pos[1]--;

}

}

else

{

while ((pos[1] < (int)maze[0].size() - 1) && (maze[pos[0]][pos[1] + 1] == 0))

{

step++;

pos[1]++;

}

}

if (pos == start) continue;

if (visited[pos[0]][pos[1]] > distance + step)

{

visited[pos[0]][pos[1]] = distance + step;

process\_queue.push(make\_pair(-visited[pos[0]][pos[1]], pos));

}

}

}

/// <summary>

/// Leet code #505. The Maze II

///

/// There is a ball in a maze with empty spaces and walls. The ball can

/// go through empty spaces by rolling up, down, left or right, but it

/// won't stop rolling until hitting a wall. When the ball stops, it could

/// choose the next direction.

///

/// Given the ball's start position, the destination and the maze, find the

/// shortest distance for the ball to stop at the destination. The distance

/// is defined by the number of empty spaces traveled by the ball from the

/// start position (excluded) to the destination (included). If the ball

/// cannot stop at the destination, return -1.

///

/// The maze is represented by a binary 2D array. 1 means the wall and 0 means

/// the empty space. You may assume that the borders of the maze are all walls.

/// The start and destination coordinates are represented by row and column

/// indexes.

///

/// Example 1

/// Input 1: a maze represented by a 2D array

///

/// 0 0 1 0 0

/// 0 0 0 0 0

/// 0 0 0 1 0

/// 1 1 0 1 1

/// 0 0 0 0 0

/// Input 2: start coordinate (rowStart, colStart) = (0, 4)

/// Input 3: destination coordinate (rowDest, colDest) = (4, 4)

///

/// Output: 12

/// Explanation: One shortest way is :

/// left -> down -> left -> down -> right -> down -> right.

/// The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.

///

/// Example 2

/// Input 1: a maze represented by a 2D array

///

/// 0 0 1 0 0

/// 0 0 0 0 0

/// 0 0 0 1 0

/// 1 1 0 1 1

/// 0 0 0 0 0

/// Input 2: start coordinate (rowStart, colStart) = (0, 4)

/// Input 3: destination coordinate (rowDest, colDest) = (3, 2)

/// Output: -1

/// Explanation: There is no way for the ball to stop at the destination.

///

/// Note:

/// 1.There is only one ball and one destination in the maze.

/// 2.Both the ball and the destination exist on an empty space,

/// and they will not be at the same position initially.

/// 3.The given maze does not contain border (like the red rectangle in the

/// example

/// pictures), but you could assume the border of the maze are all walls.

/// 4.The maze contains at least 2 empty spaces, and both the width and height of

/// the maze won't exceed 100.

/// </summary>

int LeetCode::shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination)

{

if (maze.empty() || maze[0].empty()) return -1;

vector<vector<int>> visited(maze.size(), vector<int>(maze[0].size(), INT\_MAX));

visited[start[0]][start[1]] = 0;

priority\_queue<pair<int, vector<int>>> process\_queue;

process\_queue.push(make\_pair(0, start));

while (!process\_queue.empty())

{

start = process\_queue.top().second;

process\_queue.pop();

if (start == destination) break;

shortestDistance(maze, visited, start, process\_queue);

}

int shortest\_distance = visited[destination[0]][destination[1]];

if (shortest\_distance == INT\_MAX) return -1;

else return shortest\_distance;

}

### **882. Reachable Nodes In Subdivided Graph**

Starting with an **undirected** graph (the "original graph") with nodes from 0 to N-1, subdivisions are made to some of the edges.

The graph is given as follows: edges[k] is a list of integer pairs (i, j, n) such that (i, j) is an edge of the original graph,

and n is the total number of **new** nodes on that edge.

Then, the edge (i, j) is deleted from the original graph, n new nodes (x\_1, x\_2, ..., x\_n) are added to the original graph,

and n+1 new edges (i, x\_1), (x\_1, x\_2), (x\_2, x\_3), ..., (x\_{n-1}, x\_n), (x\_n, j) are added to the original graph.

Now, you start at node 0 from the original graph, and in each move, you travel along one edge.

Return how many nodes you can reach in at most M moves.

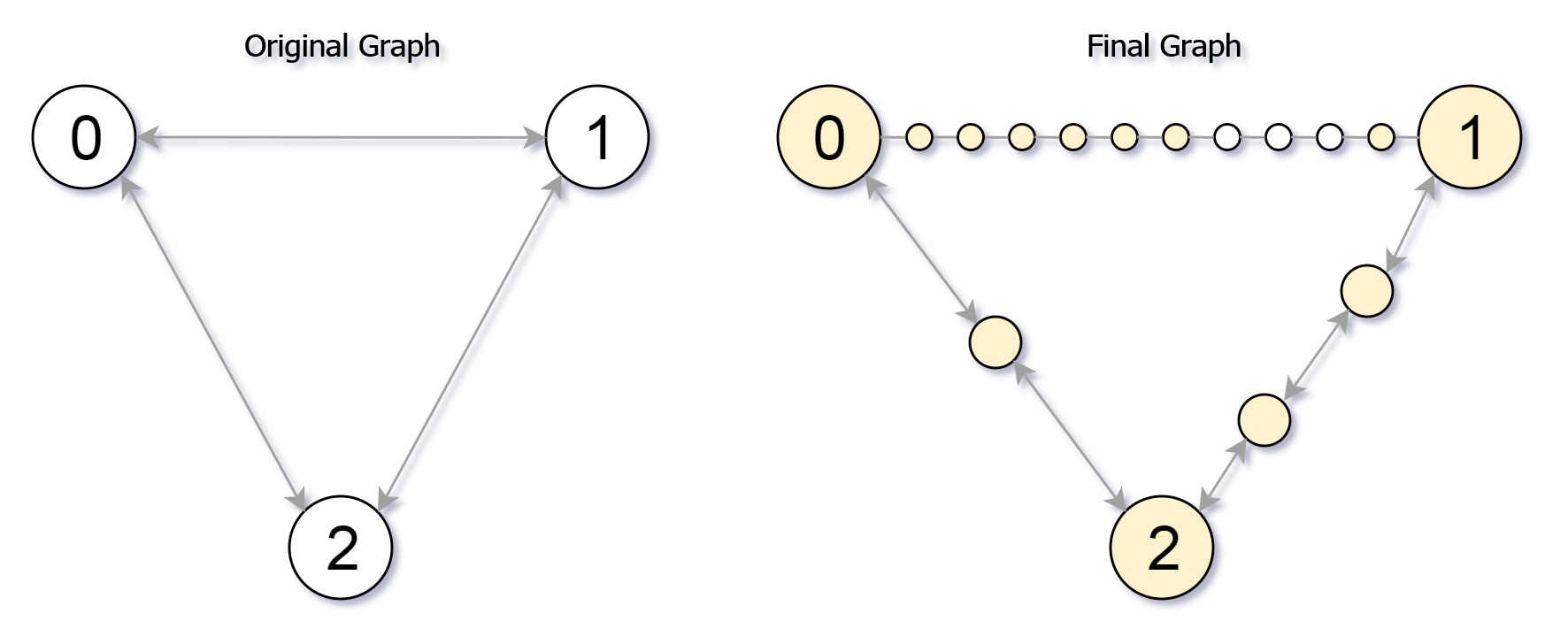
**Example 1:**

**Input:** edges = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3

**Output:** 13

**Explanation:**

The nodes that are reachable in the final graph after M = 6 moves are indicated below.



**Example 2:**

**Input:** edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4

**Output:** 23

**Note:**

1. 0 <= edges.length <= 10000
2. 0 <= edges[i][0] < edges[i][1] < N
3. There does not exist any i != j for which edges[i][0] == edges[j][0] and edges[i][1] == edges[j][1].
4. The original graph has no parallel edges.
5. 0 <= edges[i][2] <= 10000
6. 0 <= M <= 10^9
7. 1 <= N <= 3000
8. A reachable node is a node that can be travelled to using at most M moves starting from node 0.

/// <summary>

/// Leet code #882. Reachable Nodes In Subdivided Graph

///

/// Starting with an undirected graph (the "original graph") with nodes from

/// 0 to N-1, subdivisions are made to some of the edges.

///

/// The graph is given as follows: edges[k] is a list of integer pairs

/// (i, j, n) such that (i, j) is an edge of the original graph,

///

/// and n is the total number of new nodes on that edge.

///

/// Then, the edge (i, j) is deleted from the original graph, n new nodes

/// (x\_1, x\_2, ..., x\_n) are added to the original graph,

///

/// and n+1 new edges (i, x\_1), (x\_1, x\_2), (x\_2, x\_3), ..., (x\_{n-1}, x\_n),

/// (x\_n, j) are added to the original graph.

///

/// Now, you start at node 0 from the original graph, and in each move, you

/// travel along one edge.

///

/// Return how many nodes you can reach in at most M moves.

///

/// Example 1:

///

/// Input: edges = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3

/// Output: 13

/// Explanation:

/// The nodes that are reachable in the final graph after M = 6 moves are

/// indicated below.

///

/// Example 2:

///

/// Input: edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4

/// Output: 23

///

/// Note:

///

/// 1. 0 <= edges.length <= 10000

/// 2. 0 <= edges[i][0] < edges[i][1] < N

/// 3. There does not exist any i != j for which edges[i][0] == edges[j][0] and

/// edges[i][1] == edges[j][1].

/// 4. The original graph has no parallel edges.

/// 5. 0 <= edges[i][2] <= 10000

/// 6. 0 <= M <= 10^9

/// 7. 1 <= N <= 3000

/// 8. reachable node is a node that can be travelled to using at most M moves

/// starting from node 0.

/// </summary>

int LeetCode::reachableNodes(vector<vector<int>>& edges, int M, int N)

{

int result = 0;

unordered\_map<int, unordered\_map<int, int>> edge\_nodes;

vector<int> visited(N);

for (size\_t i = 0; i < edges.size(); i++)

{

edge\_nodes[edges[i][0]][edges[i][1]] = edges[i][2];

edge\_nodes[edges[i][1]][edges[i][0]] = edges[i][2];

}

priority\_queue<pair<int, int>>search;

search.push(make\_pair(M, 0));

while (!search.empty())

{

pair<int, int> pos = search.top();

search.pop();

int node = pos.second;

int step = pos.first;

if (visited[node] == 1) continue;

result++;

visited[node] = 1;

for (auto itr : edge\_nodes[node])

{

int target\_node = itr.first;

int nodes\_in\_middle = itr.second;

if (nodes\_in\_middle + 1 <= step)

{

result += nodes\_in\_middle;

if (visited[target\_node] == 0)

{

search.push(make\_pair(step - nodes\_in\_middle - 1, target\_node));

edge\_nodes[node][target\_node] = 0;

edge\_nodes[target\_node][node] = 0;

}

}

else

{

result += step;

edge\_nodes[node][target\_node] -= step;

edge\_nodes[target\_node][node] -= step;

}

}

}

return result;

}

## Union Find

The union find is used for grouping the related vertices in a graph. Every union group can be considered as a tree. When merge two groups we simply point the root of one group to another root.

The code pattern is that we normally use a hash table (or array to map to the parents, finally to the root, by default every new node point to itself, when need to merge it will point to a new root. This is also to say after union find process, the one who point to itself is a root.

### **952. Largest Component Size by Common Factor**

Given a non-empty array of unique positive integers A, consider the following graph:

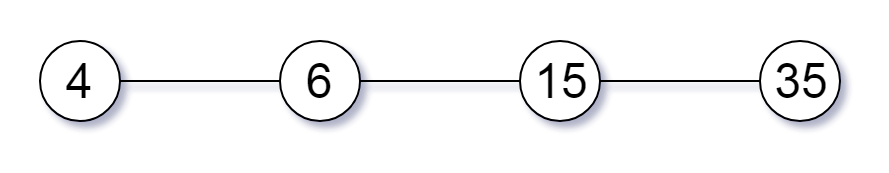
* There are A.length nodes, labelled A[0] to A[A.length - 1];
* There is an edge between A[i] and A[j] if and only if A[i] and A[j] share a common factor greater than 1.

Return the size of the largest connected component in the graph.

**Example 1:**

**Input:** [4,6,15,35]

**Output:** 4



**Example 2:**

**Input:** [20,50,9,63]

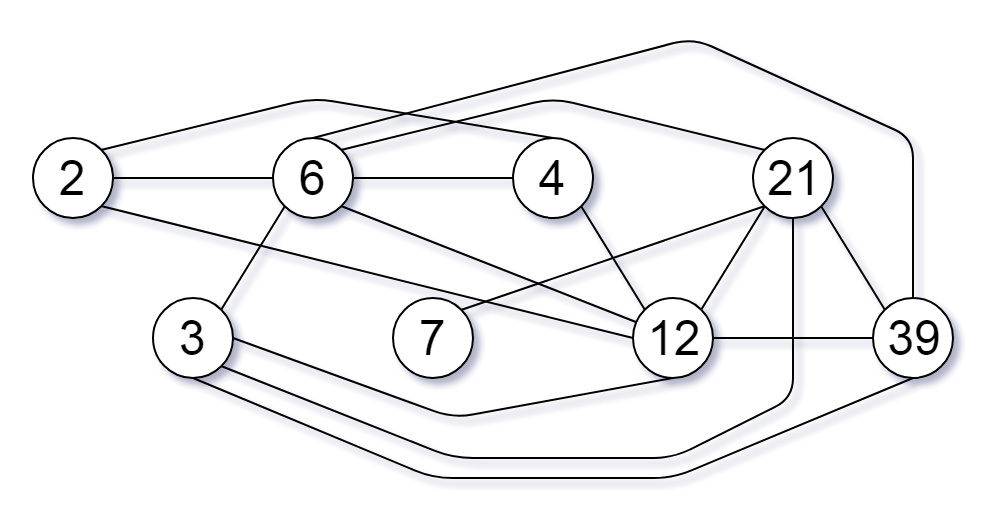
**Output:** 2



**Example 3:**

**Input:** [2,3,6,7,4,12,21,39]

**Output:** 8



**Note:**

1. 1 <= A.length <= 20000
2. 1 <= A[i] <= 100000

/// <summary>

/// Leet code #952. Largest Component Size by Common Factor

///

/// Given a non-empty array of unique positive integers A, consider the

/// following graph:

///

/// There are A.length nodes, labelled A[0] to A[A.length - 1];

/// There is an edge between A[i] and A[j] if and only if A[i] and A[j]

/// share a common factor greater than 1.

/// Return the size of the largest connected component in the graph.

///

/// Example 1:

/// Input: [4,6,15,35]

/// Output: 4

///

/// Example 2:

/// Input: [20,50,9,63]

/// Output: 2

///

/// Example 3:

/// Input: [2,3,6,7,4,12,21,39]

/// Output: 8

///

/// Note:

/// 1. 1 <= A.length <= 20000

/// 2. 1 <= A[i] <= 100000

/// </summary>

int LeetCode::largestComponentSize(vector<int>& A)

{

vector<vector<int>> factors(A.size());

unordered\_map<int, int> prime\_map;

vector<int> union\_map(A.size());

vector<int> count(A.size(), 1);

for (size\_t i = 0; i < A.size(); i++)

{

int x = A[i];

for (size\_t d = 2; d <= sqrt(x); d++)

{

if (x % d == 0)

{

while (x % d == 0) x /= d;

factors[i].push\_back(d);

}

}

if (x > 1) factors[i].push\_back(x);

}

int result = 1;

for (size\_t i = 0; i < A.size(); i++)

{

union\_map[i] = i;

count[i] = 1;

for (auto p : factors[i])

{

if (prime\_map.count(p) == 0)

{

prime\_map[p] = i;

}

else

{

int source = i;

int target = prime\_map[p];

while (union\_map[source] != source) source = union\_map[source];

while (union\_map[target] != target) target = union\_map[target];

if (source != target)

{

union\_map[source] = target;

count[target] += count[source];

result = max(count[target], result);

}

}

}

}

return result;

}