简单图论

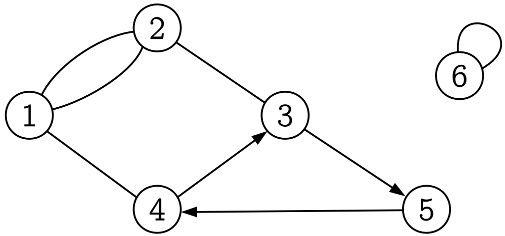
July 21, 2018, riteme

图：基本概念

图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。

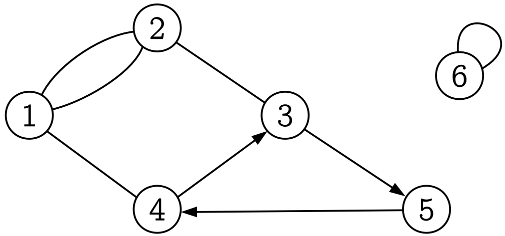
图：基本概念

图由点集和边集构成，记为 $G = (V, E)$，V 是点集，E 是边集。通常令 $n = |V|$，$m = |E|$。



图：基本概念

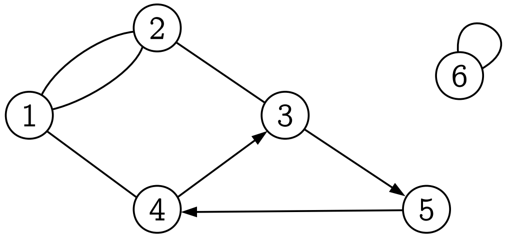
图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。



无向边用 $u-v$ 表示，**有向边**用 $u\to v$ 表示。无向边可以用两条方向相反的有向边表示。

图：基本概念

图由点集和边集构成，记为 $G = (V, E)$，V 是点集，E 是边集。通常令 $n = |V|$，$m = |E|$。

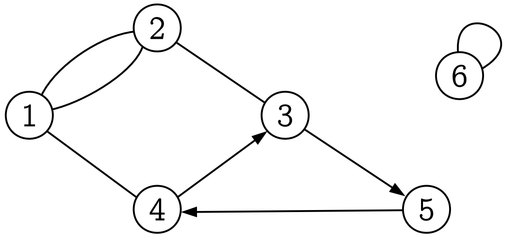


无向边用 $u - v$ 表示，**有向边**用 $u \to v$ 表示。无向边可以用两条方向相反的有向边表示。

边可能带有权值，用来表示长度或者其他的意义。

图：基本概念

图由点集和边集构成，记为 $G = (V, E)$，V 是点集，E 是边集。通常令 $n = |V|$，$m = |E|$。



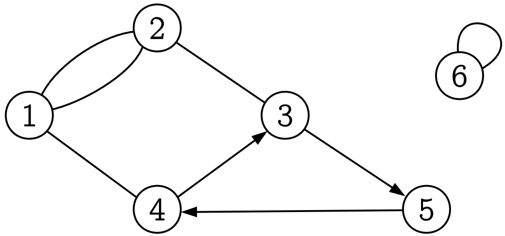
无向边用 $u - v$ 表示，**有向边**用 $u \to v$ 表示。无向边可以用两条方向相反的有向边表示。

边可能带有权值，用来表示长度或者其他的意义。

无向图中与一个节点相连的边的数量称为**度数**。有向图中进入一个点的边数为**入度**，从一个点出发的边数为**出度**。

图：基本概念

图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。



无向边用 $u-v$ 表示，**有向边**用 $u\to v$ 表示。无向边可以用两条方向相反的有向边表示。

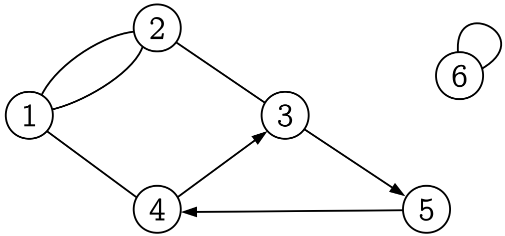
边可能带有权值，用来表示长度或者其他的意义。

无向图中与一个节点相连的边的数量称为**度数**。有向图中进入一个点的边数为**入度**，从一个点出发的边数为**出度**。

如果边的两个端点相同，则称为**自环**。某两个节点之间可能有多条边相连，这些边都称为**重边**。没有重边和自环的图是**简单图**。

图：基本概念

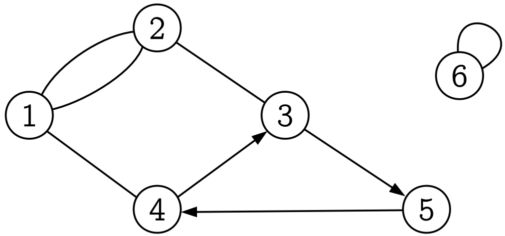
图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。



路径是一个序列 $[x_1, x_2, \ldots, x_k]$，其中相邻两个节点之间有无向边或有向边相连。如果 $x_1 = x_k$，则称为**环**。如果路径中没有重复元素，则称为**简单路径**。

图：基本概念

图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。

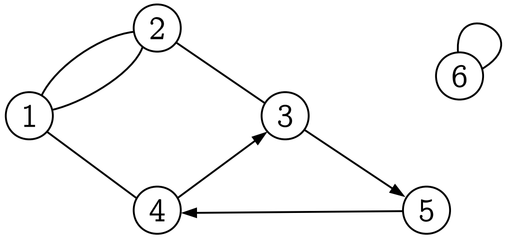


路径是一个序列 $[x_1,x_2,\ldots,x_k]$，其中相邻两个节点之间有无向边或有向边相连。如果 $x_1=x_k$，则称为**环**。如果路径中没有重复元素，则称为**简单路径**。

如果只有无向边，则 G 为**无向图**；如果只有有向边，则为**有向图**。否则为**混合图**。

图：基本概念

图由点集和边集构成，记为 $G=(V,E)$，V 是点集，E 是边集。通常令 $n=|V|$，$m=|E|$。



路径是一个序列 $[x_1,x_2,\ldots,x_k]$，其中相邻两个节点之间有无向边或有向边相连。如果 $x_1=x_k$，则称为**环**。如果路径中没有重复元素，则称为**简单路径**。
如果只有无向边，则 G 为**无向图**；如果只有有向边，则为**有向图**。否则为**混合图**。
在无向图中，如果两个节点之间有路径相连则这两个节点是**连通**的。**连通块**是一个极大的点集，其中任意两个节点之间都是连通的。如果所有节点之间都是连通的，那么 G 是**连通图**。

图：存储

图的存储一般有两种方式：邻接矩阵和邻接表（前向星）。

图：存储

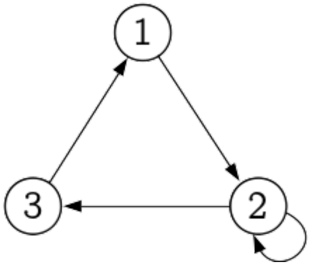
图的存储一般有两种方式：邻接矩阵和邻接表（前向星）。

邻接矩阵是一个 $n \times n$ 的矩阵 M，如果 u 和 v 之间有有向边 $u \to v$，则 $M[u][v] = 1$。

如果是无向边，则 $M[u][v] = M[v][u] = 1$。

图：存储

图的存储一般有两种方式：邻接矩阵和邻接表（前向星）。
邻接矩阵是一个 $n \times n$ 的矩阵 M，如果 u 和 v 之间有有向边 $u \to v$，则 $M[u][v] = 1$。
如果是无向边，则 $M[u][v] = M[v][u] = 1$。



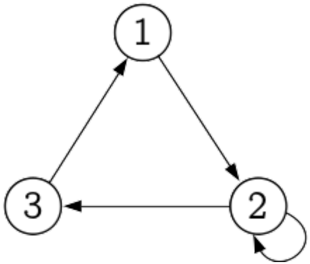
$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

图：存储

图的存储一般有两种方式：邻接矩阵和邻接表（前向星）。

邻接矩阵是一个 $n \times n$ 的矩阵 M，如果 u 和 v 之间有有向边 $u \to v$，则 $M[u][v] = 1$。

如果是无向边，则 $M[u][v] = M[v][u] = 1$。



$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

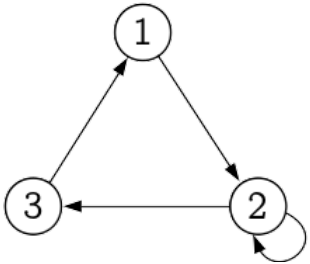
如果有非零边权，可以设 $M[u][v]$ 为边权。

图：存储

图的存储一般有两种方式：邻接矩阵和邻接表（前向星）。

邻接矩阵是一个 $n \times n$ 的矩阵 M，如果 u 和 v 之间有有向边 $u \to v$，则 $M[u][v] = 1$。

如果是无向边，则 $M[u][v] = M[v][u] = 1$。



$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

如果有非零边权，可以设 $M[u][v]$ 为边权。

邻接矩阵的缺点：稀疏图中查找边的效率不高，处理重边麻烦，空间复杂度 $\Theta(n^2)$。

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

```
struct Edge:
    int u, v, w  // 起点、终点、边权
    Edge *nxt    // 链表的下一个元素
Edge *G[N]
```

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

```
struct Edge:
    int u, v, w  // 起点、终点、边权
    int nxt      // 链表的下一个元素
Edge e[]         // 边数组
int G[N]         // 链表的第一个元素
```

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

```
struct Edge:
    int u, v, w  // 起点、终点、边权
    int nxt      // 链表的下一个元素
Edge e[]         // 边数组
int G[N]         // 链表的第一个元素
```

如果采用数组存边，一般下标从 0 开始用。因为对于无向图，每条无向边拆分出来的两条边在数组中相邻，如果其中一条边的下标是 `x` 的话，那么另外一条边的下标就是 `x ^ 1`。

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

```
struct Edge:
    int u, v, w  // 起点、终点、边权
    int nxt      // 链表的下一个元素
Edge e[]         // 边数组
int G[N]         // 链表的第一个元素
```

如果采用数组存边，一般下标从 0 开始用。因为对于无向图，每条无向边拆分出来的两条边在数组中相邻，如果其中一条边的下标是 `x` 的话，那么另外一条边的下标就是 `x ^ 1`。

如果要访问节点 u 的所有边，从 $G[u]$ 开始遍历链表即可。

图：存储

邻接表类似于存储稀疏矩阵：给每个节点开一个链表，存储从这个节点出发的所有边。

```
struct Edge:
    int u, v, w  // 起点、终点、边权
    int nxt      // 链表的下一个元素
Edge e[]         // 边数组
int G[N]         // 链表的第一个元素
```

如果采用数组存边，一般下标从 0 开始用。因为对于无向图，每条无向边拆分出来的两条边在数组中相邻，如果其中一条边的下标是 `x` 的话，那么另外一条边的下标就是 `x ^ 1`。

如果要访问节点 u 的所有边，从 $G[u]$ 开始遍历链表即可。

或者是使用 `vector`、`deque`。

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

树：基本概念

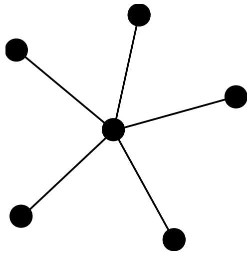
树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

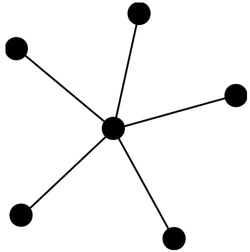
另一种说法：任意两个节点之间有且仅有一条简单路径。



树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。



~~菊花树 Patrick!~~

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。

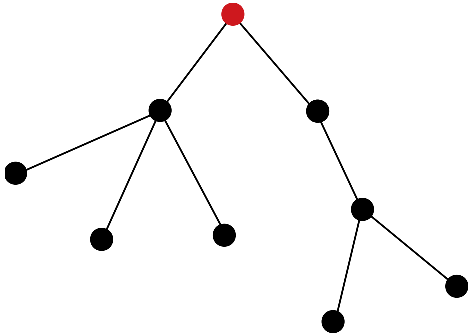
树可以选定**树根**，因此分为**无根树**和**有根树**。树最外围的节点被称为**叶子节点**，准确的讲，度数为 1 的非根节点都是叶节点。

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。

树可以选定**树根**，因此分为**无根树**和**有根树**。树最外围的节点被称为**叶子节点**，准确的讲，度数为 1 的非根节点都是叶节点。

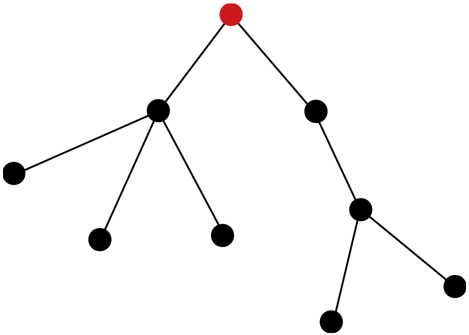


树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。

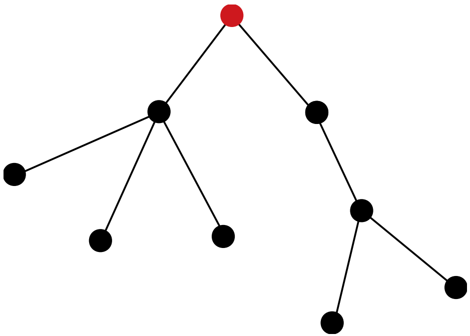
树可以选定**树根**，因此分为**无根树**和**有根树**。树最外围的节点被称为**叶子节点**，准确的讲，度数为 1 的非根节点都是叶节点。



对于有根树，一个节点到根节点的简单路径的长度为这个节点的**深度**。一条边的两个端点深度会不相同，设深度小的为 u，深度大的为 v，则 u 是 v 的**父亲**，v 是 u 的**儿子**。

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。
另一种说法：任意两个节点之间有且仅有一条简单路径。
树可以选定**树根**，因此分为**无根树**和**有根树**。树最外围的节点被称为**叶子节点**，准确的讲，度数为 1 的非根节点都是叶节点。



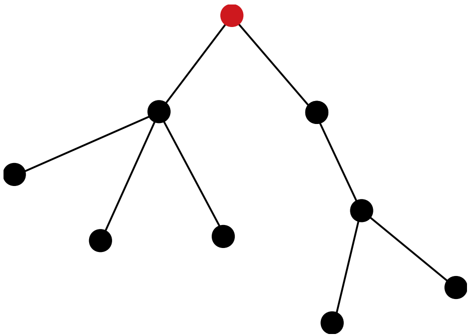
对于有根树，一个节点到根节点的简单路径的长度为这个节点的**深度**。一条边的两个端点深度会不相同，设深度小的为 u，深度大的为 v，则 u 是 v 的**父亲**，v 是 u 的**儿子**。
从某个节点 v 的父亲开始到根节点的简单路径上所有节点都是 v 的**祖先**。从 v 向下走的所有节点是 v 的**后代**。v 的所有后代和 v 以及相关的边构成一棵**子树**。

树：基本概念

树是一类特殊的连通图：由 n 个点和 $n-1$ 条边组成。

另一种说法：任意两个节点之间有且仅有一条简单路径。

树可以选定**树根**，因此分为**无根树**和**有根树**。树最外围的节点被称为**叶子节点**，准确的讲，度数为 1 的非根节点都是叶节点。



对于有根树，一个节点到根节点的简单路径的长度为这个节点的**深度**。一条边的两个端点深度会不相同，设深度小的为 u，深度大的为 v，则 u 是 v 的**父亲**，v 是 u 的**儿子**。

从某个节点 v 的父亲开始到根节点的简单路径上所有节点都是 v 的**祖先**。从 v 向下走的所有节点是 v 的**后代**。v 的所有后代和 v 以及相关的边构成一棵**子树**。

由多棵树组成的无向图为**森林**。

【LG P1141】01 迷宫

给出一个 $n \times n$ 的 $0/1$ 矩阵。如果你站在 0 上，则可以移动到上下左右相邻四格的 1 处；如果站在 1 上，就可以移动到相邻四格的 0 上。然后有 m 次询问，每次给出一个位置，问你最多可以到达多少个位置。

$n \leqslant 1000$，$m \leqslant 10^5$

【LG P1141】01 迷宫

给出一个 $n \times n$ 的 $0/1$ 矩阵。如果你站在 0 上，则可以移动到上下左右相邻四格的 1 处；如果站在 1 上，就可以移动到相邻四格的 0 上。然后有 m 次询问，每次给出一个位置，问你最多可以到达多少个位置。

$n \leqslant 1000$，$m \leqslant 10^5$

构建一个 n^2 个点的图，相邻两个数字不同的点之间连边。

【LG P1141】01 迷宫

给出一个 $n \times n$ 的 $0/1$ 矩阵。如果你站在 0 上，则可以移动到上下左右相邻四格的 1 处；如果站在 1 上，就可以移动到相邻四格的 0 上。然后有 m 次询问，每次给出一个位置，问你最多可以到达多少个位置。

$n \leqslant 1000$，$m \leqslant 10^5$

构建一个 n^2 个点的图，相邻两个数字不同的点之间连边。

每次询问相当于查询一个点所处的连通块的点数。

【LG P1141】01 迷宫

给出一个 $n \times n$ 的 0/1 矩阵。如果你站在 0 上，则可以移动到上下左右相邻四格的 1 处；如果站在 1 上，就可以移动到相邻四格的 0 上。然后有 m 次询问，每次给出一个位置，问你最多可以到达多少个位置。

$n \leqslant 1000$，$m \leqslant 10^5$

构建一个 n^2 个点的图，相邻两个数字不同的点之间连边。

每次询问相当于查询一个点所处的连通块的点数。

使用 DFS 或者 BFS 找出所有连通块：DFS 或 BFS 的过程中标记已经访问过的节点，避免重复访问。

【LG P1141】01 迷宫

给出一个 $n \times n$ 的 $0/1$ 矩阵。如果你站在 0 上，则可以移动到上下左右相邻四格的 1 处；如果站在 1 上，就可以移动到相邻四格的 0 上。然后有 m 次询问，每次给出一个位置，问你最多可以到达多少个位置。

$n \leqslant 1000$，$m \leqslant 10^5$

构建一个 n^2 个点的图，相邻两个数字不同的点之间连边。

每次询问相当于查询一个点所处的连通块的点数。

使用 DFS 或者 BFS 找出所有连通块：DFS 或 BFS 的过程中标记已经访问过的节点，避免重复访问。

时间复杂度 $\Theta(n+m)$。

【NOIP 2014 / LG P2296】寻找道路

在有向图 G 中，每条边的长度为 1。现给定起点和终点，要求找出一条最短的路径，满足这条路径上所有点的出边所指向的点都直接或间接与终点连通。

保证终点没有出边。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

【NOIP 2014 / LG P2296】寻找道路

在有向图 G 中，每条边的长度为 1。现给定起点和终点，要求找出一条最短的路径，满足这条路径上所有点的出边所指向的点都直接或间接与终点连通。

保证终点没有出边。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

将图 G 中的边反向得到图 G'，然后从终点出发 BFS，标记所有能够直接或间接到达终点的节点。

【NOIP 2014 / LG P2296】寻找道路

在有向图 G 中，每条边的长度为 1。现给定起点和终点，要求找出一条最短的路径，满足这条路径上所有点的出边所指向的点都直接或间接与终点连通。

保证终点没有出边。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

将图 G 中的边反向得到图 G'，然后从终点出发 BFS，标记所有能够直接或间接到达终点的节点。

所有没有被标记的点是不合法的。

【NOIP 2014 / LG P2296】寻找道路

在有向图 G 中，每条边的长度为 1。现给定起点和终点，要求找出一条最短的路径，满足这条路径上所有点的出边所指向的点都直接或间接与终点连通。

保证终点没有出边。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

将图 G 中的边反向得到图 G'，然后从终点出发 BFS，标记所有能够直接或间接到达终点的节点。

所有没有被标记的点是不合法的。

同时需要注意，即使是被标记的点，也可能不合法。对于每个未被标记的点 v，枚举图 G 中的每条入边 $u \to v$，可知 u 不满足条件，所以 u 也是不合法的。

【NOIP 2014 / LG P2296】寻找道路

在有向图 G 中，每条边的长度为 1。现给定起点和终点，要求找出一条最短的路径，满足这条路径上所有点的出边所指向的点都直接或间接与终点连通。

保证终点没有出边。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

将图 G 中的边反向得到图 G'，然后从终点出发 BFS，标记所有能够直接或间接到达终点的节点。

所有没有被标记的点是不合法的。

同时需要注意，即使是被标记的点，也可能不合法。对于每个未被标记的点 v，枚举图 G 中的每条入边 $u \to v$，可知 u 不满足条件，所以 u 也是不合法的。

由于边权都为 1，寻找最短路可以直接在所有的合法点上 BFS。

【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。

游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮?

$n \leqslant 10^5$

【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。
游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮?
$n \leqslant 10^5$
转换为图论模型：在一个 n 个点的有向图中，每个点只有一条出边。求出图中最小环的长度。

【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。

游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮?

$n \leqslant 10^5$

转换为图论模型：在一个 n 个点的有向图中，每个点只有一条出边。求出图中最小环的长度。

由于每个点只有一条出边，所以每个点最多处于一个环中。

【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。

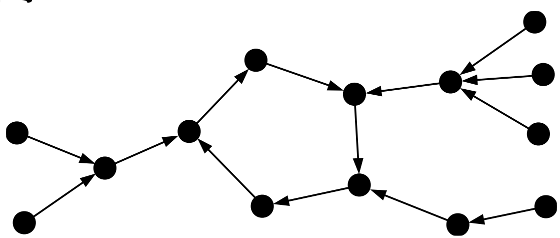
游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮?

$n \leqslant 10^5$

转换为图论模型：在一个 n 个点的有向图中，每个点只有一条出边。求出图中最小环的长度。

由于每个点只有一条出边，所以每个点最多处于一个环中。

所以最后的图大概是这个样子:



【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。

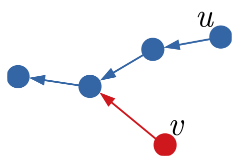
游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮?

$n \leqslant 10^5$

转换为图论模型：在一个 n 个点的有向图中，每个点只有一条出边。求出图中最小环的长度。

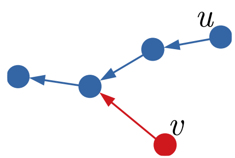
由于每个点只有一条出边，所以每个点最多处于一个环中。

使用 DFS 来找环。但是要注意“假环”的情况：



【NOIP 2015 / LG P2661】信息传递

有 n 个同学（编号为 1 到 n ）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为 T_i 的同学。
游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息， 但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮？
$n \leqslant 10^5$
转换为图论模型：在一个 n 个点的有向图中，每个点只有一条出边。求出图中最小环的长度。
由于每个点只有一条出边，所以每个点最多处于一个环中。
使用 DFS 来找环。但是要注意 “假环” 的情况：



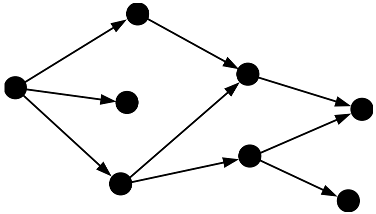
如果第一次从 u 出发，标记了所有蓝色节点，没有发现环。然后从 v 开始，访问到了被标记的节点，但此时不能认为找到了环。必须要是当次 DFS 访问过的节点的才能认为是找到了环。

拓扑图

没有环的无向图是森林，没有环的有向图是**拓扑图**。

拓扑图

没有环的无向图是森林，没有环的有向图是**拓扑图**。



拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

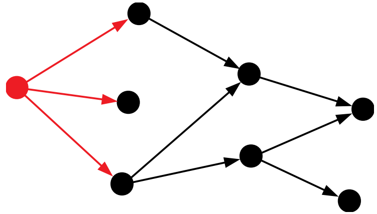
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。

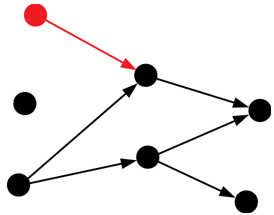
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。



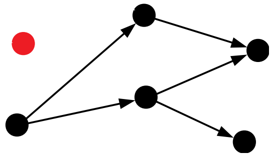
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



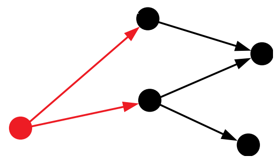
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



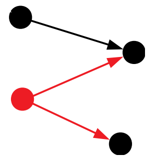
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



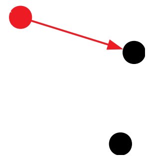
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。

拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。

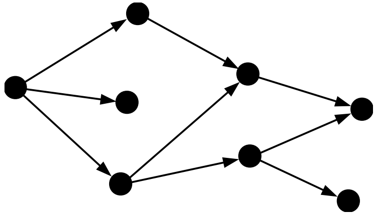
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。

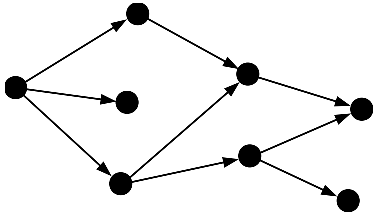
拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。



拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。
每次任意删去一个入度为 0 的点，最后可以得到一个拓扑序。

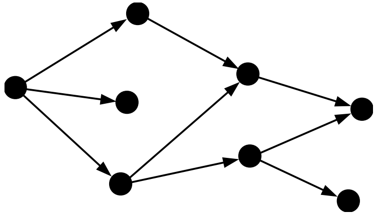


实现非常简单，删边的时候注意下是否入度变为了 0。如果是则进入队列。

拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。



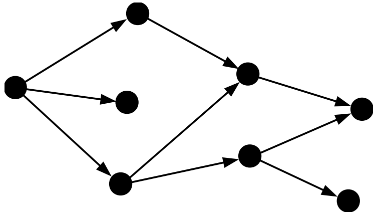
实现非常简单，删边的时候注意下是否入度变为了 0。如果是则进入队列。

拓扑序的特点：所有点 u 能到达的点的拓扑序都在 u 后面。

拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。



实现非常简单，删边的时候注意下是否入度变为了 0。如果是则进入队列。

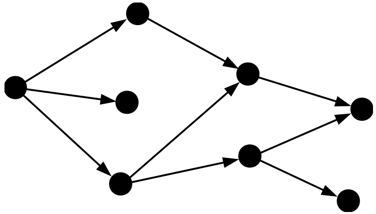
拓扑序的特点：所有点 u 能到达的点的拓扑序都在 u 后面。

另一方面，所有能够到达 u 的点都在 u 前面。

拓扑排序

拓扑图由于没有环，所以节点之间可以排出一个相对的顺序。

每次任意删去一个入度为 0 的点，最后可以得到一个**拓扑序**。



实现非常简单，删边的时候注意下是否入度变为了 0。如果是则进入队列。

拓扑序的特点：所有点 u 能到达的点的拓扑序都在 u 后面。

另一方面，所有能够到达 u 的点都在 u 前面。

因此拓扑序可以用作 DP 的顺序，昨天的课也提到过。

【LG P1137】旅行计划

给定拓扑图，求出以每个点为终点的最长路径的长度。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

【LG P1137】旅行计划

给定拓扑图，求出以每个点为终点的最长路径的长度。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

设 $f(x)$ 为 DP 数组，对于每个进入点 x 的边 $u \to x$，用 $f(u)+1$ 来更新 $f(x)$。

【LG P1137】旅行计划

给定拓扑图，求出以每个点为终点的最长路径的长度。

$n \leqslant 10^5$，$m \leqslant 2 \times 10^5$

设 $f(x)$ 为 DP 数组，对于每个进入点 x 的边 $u \to x$，用 $f(u) + 1$ 来更新 $f(x)$。

按照拓扑序来进行 DP。

求拓扑图的“割点”

给出一张 n 个点的拓扑图，保证点 1 能够到达点 n。找出所有的节点 x，满足删除 x 后，点 1 不能到达点 n。

$n \leqslant 10^5$

求拓扑图的“割点”

给出一张 n 个点的拓扑图，保证点 1 能够到达点 n。找出所有的节点 x，满足删除 x 后，点 1 不能到达点 n。

$n \leqslant 10^5$

DP 求出 $S[x]$ 和 $T[x]$，分别表示从点 1 到 x 的路径总数，和 x 到 n 的路径总数。

求拓扑图的“割点”

给出一张 n 个点的拓扑图，保证点 1 能够到达点 n。找出所有的节点 x，满足删除 x 后，点 1 不能到达点 n。

$n \leqslant 10^5$

DP 求出 $S[x]$ 和 $T[x]$，分别表示从点 1 到 x 的路径总数，和 x 到 n 的路径总数。

所谓割点，就是每条从 1 到 n 的路径都会经过的点。

求拓扑图的“割点”

给出一张 n 个点的拓扑图，保证点 1 能够到达点 n。找出所有的节点 x，满足删除 x 后，点 1 不能到达点 n。

$n \leqslant 10^5$

DP 求出 $S[x]$ 和 $T[x]$，分别表示从点 1 到 x 的路径总数，和 x 到 n 的路径总数。

所谓割点，就是每条从 1 到 n 的路径都会经过的点。

那么对于割点 x，$S[x] \cdot T[x]$ 必定等于 $S[n]$。

求拓扑图的“割点”

给出一张 n 个点的拓扑图，保证点 1 能够到达点 n。找出所有的节点 x，满足删除 x 后，点 1 不能到达点 n。

$n \leqslant 10^5$

DP 求出 $S[x]$ 和 $T[x]$，分别表示从点 1 到 x 的路径总数，和 x 到 n 的路径总数。

所谓割点，就是每条从 1 到 n 的路径都会经过的点。

那么对于割点 x，$S[x] \cdot T[x]$ 必定等于 $S[n]$。

每个点都试一下就好了。

强连通分量

无向图中连通的概念，有向图中也有。

强连通分量

无向图中连通的概念，有向图中也有。

弱连通：将有向边转为无向边，如果转换后的无向图是连通的，则原有向图是弱连通的。

强连通分量

无向图中连通的概念，有向图中也有。

弱连通：将有向边转为无向边，如果转换后的无向图是连通的，则原有向图是弱连通的。

强连通：如果对于任意两个点之间都至少有一条有向路径，则为强连通的。

强连通分量

无向图中连通的概念，有向图中也有。

弱连通： 将有向边转为无向边，如果转换后的无向图是连通的，则原有向图是弱连通的。

强连通： 如果对于任意两个点之间都至少有一条有向路径，则为强连通的。

CS 大佬 Tarjan 老爷爷首先给出了找有向图中所有强连通分量的算法：一个简单的 DFS。

强连通分量

无向图中连通的概念，有向图中也有。

弱连通：将有向边转为无向边，如果转换后的无向图是连通的，则原有向图是弱连通的。

强连通：如果对于任意两个点之间都至少有一条有向路径，则为强连通的。

CS 大佬 Tarjan 老爷爷首先给出了找有向图中所有强连通分量的算法：一个简单的 DFS。

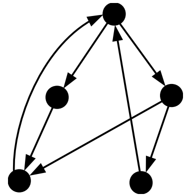
最简单的强连通分量就是一个环。Tarjan 的算法的主要思想就是将找到的环全部缩起来，最后每个强连通分量就会缩成一个点。同时，缩完点后的图由于没有环，所以是一个拓扑图。

DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。

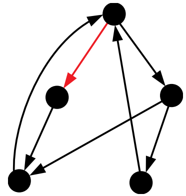
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



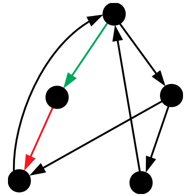
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



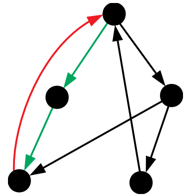
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



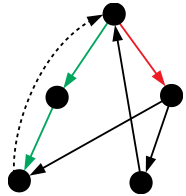
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



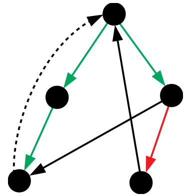
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



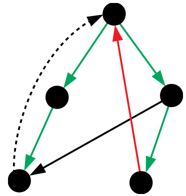
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



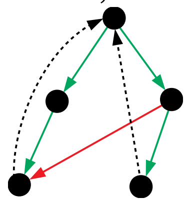
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



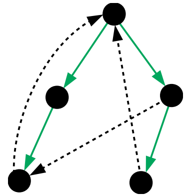
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



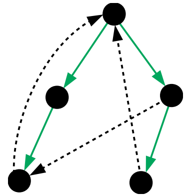
DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



DFS 生成树

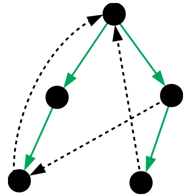
从任意一个点开始 DFS，保留所有走过的边，构成一棵树。



上图中绿色边构成了 DFS 生成树，虚线边是**非树边**。

DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。

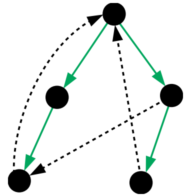


上图中绿色边构成了 DFS 生成树，虚线边是**非树边**。

有向图的非树边分为两类：一类是**返祖边**，边的终点是起点的祖先；其余的是**横跨边**。

DFS 生成树

从任意一个点开始 DFS，保留所有走过的边，构成一棵树。

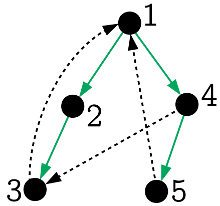


上图中绿色边构成了 DFS 生成树，虚线边是**非树边**。

有向图的非树边分为两类：一类是**返祖边**，边的终点是起点的祖先；其余的是**横跨边**。每条返祖边都代表一个环，所以在 Tarjan 算法中是需要被缩点的。横跨边一般不会构成环，但是返祖边缩环之后，横跨边也有可能成为返祖边。上图就是这样的例子。

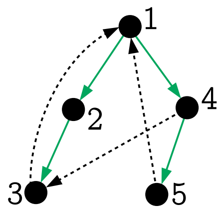
DFS 序

还可以记录下 DFS 中每个节点首次访问的时间 `dfn`:



DFS 序

还可以记录下 DFS 中每个节点首次访问的时间 `dfn`：

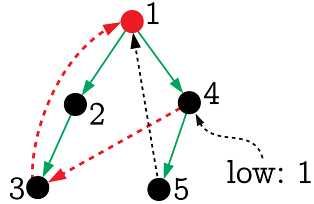


`dfn` 就是 DFS 序。在 DFS 树上，父亲节点的 `dfn` 比儿子要小。

除了 `dfn` 数组外，算法还会记录 `low` 数组，用于记录 “返祖路径”。

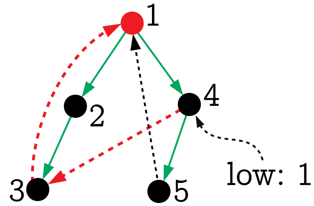
Tarjan 强连通分量算法

除了 `dfn` 数组外，算法还会记录 `low` 数组，用于记录 “返祖路径”。



Tarjan 强连通分量算法

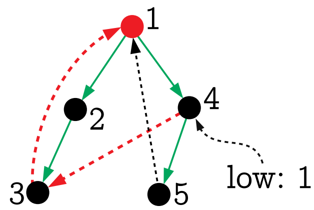
除了 `dfn` 数组外，算法还会记录 `low` 数组，用于记录“返祖路径”。



正如字面意思 `low[x]` 记录的是从 x 出发能到达的 DFS 序最小的祖先。DFS 序越小，找到的强连通分量越大。

Tarjan 强连通分量算法

除了 `dfn` 数组外，算法还会记录 `low` 数组，用于记录“返祖路径”。

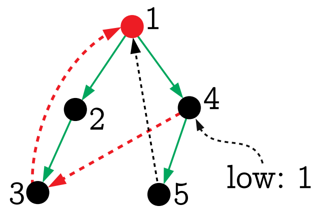


正如字面意思 `low[x]` 记录的是从 x 出发能到达的 DFS 序最小的祖先。DFS 序越小，找到的强连通分量越大。

当 `low` 等于 `dfn` 时，说明此时已经没有能够走出 DFS 子树的路径了，子树之外的部分对于子树而言是不可达的，因此强连通分量就此而止。除去子树内部已经找出的强连通分量外，其余的部分属于同一个强连通分量。

Tarjan 强连通分量算法

除了 `dfn` 数组外，算法还会记录 `low` 数组，用于记录“返祖路径”。



正如字面意思 `low[x]` 记录的是从 x 出发能到达的 DFS 序最小的祖先。DFS 序越小，找到的强连通分量越大。

当 `low` 等于 `dfn` 时，说明此时已经没有能够走出 DFS 子树的路径了，子树之外的部分对于子树而言是不可达的，因此强连通分量就此而止。除去子树内部已经找出的强连通分量外，其余的部分属于同一个强连通分量。

这个强连通分量内其余的点都是自己的后代，所以它们的 DFS 序都比自己大。为了快速访问，使用一个栈来记录所有访问过的节点。每当找到强连通分量时，就可以从栈的尾部依次弹出强连通分量里面的节点。

强连通分量：实现

~~这个代码是我随手打的……~~

强连通分量：实现

~~这个代码是我随手打的…~~

```
int cur = 0, dfn[], low[]
bool ok[]  // 是否已经出栈
function dfs(int x):
    dfn[x] = low[x] = ++cur
    stk.push(x)
    for v in G[x]:  // 邻接表实现：遍历 x 的所有出边 x -> v
        if dfn[v] == 0:  // 还未访问过
            dfs(v)
        if not ok[v]:  // 还未出栈
            low[x] = min(low[x], low[v])
    if low[x] == dfn[x]:
        do:
            int u = stk.pop()  // 弹出来的所有 u 在同一个强连通分量内
            ok[u] = true
        until u == x
```

【LG P3916】图的遍历

给出一张有向图，求出每个点能到达的点中编号最大者。

$n, m \leqslant 10^5$

【LG P3916】图的遍历

给出一张有向图，求出每个点能到达的点中编号最大者。

$n, m \leqslant 10^5$

之前提到过，将所有找出的强连通分量缩点后，会得到一张拓扑图。

【LG P3916】图的遍历

给出一张有向图，求出每个点能到达的点中编号最大者。

$n,\ m \leqslant 10^5$

之前提到过，将所有找出的强连通分量缩点后，会得到一张拓扑图。

对于每个强连通分量，内部的点是互相可达的。此外还需要在拓扑图上 DP，找到其他可到达的点。

【LG P3916】图的遍历

给出一张有向图，求出每个点能到达的点中编号最大者。

$n, m \leqslant 10^5$

之前提到过，将所有找出的强连通分量缩点后，会得到一张拓扑图。

对于每个强连通分量，内部的点是互相可达的。此外还需要在拓扑图上 DP，找到其他可到达的点。

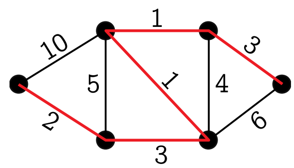
这题还有一个贪心的做法：将图反着建，然后按编号从大到小依次 DFS，填充答案数组。

最短路

每条边带上一个长度，最短路问题询问两点之间最短路径的长度是多少。

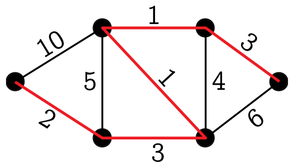
最短路

每条边带上一个长度，最短路问题询问两点之间最短路径的长度是多少。



最短路

每条边带上一个长度，最短路问题询问两点之间最短路径的长度是多少。



如何求解最短路呢?

松弛操作

设置起点为 s，以及数组 $\mathrm{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。
在已知当前的 dist 的情况下，再往图中加入一条边 $e : u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。

松弛操作

设置起点为 s，以及数组 $\mathrm{dist}[x]$ 表示从 s 到 x 的最短路径的长度。
在已知当前的 dist 的情况下，再往图中加入一条边 $e : u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。
为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。
在已知当前的 dist 的情况下，再往图中加入一条边 $e: u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。
为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。
如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

在已知当前的 dist 的情况下，再往图中加入一条边 $e : u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。

为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

Bellman-Ford 算法就是利用松弛操作：一开始只有 $\text{dist}[s] = 0$，其余均为 ∞。然后枚举每条边，进行松弛操作。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

在已知当前的 dist 的情况下，再往图中加入一条边 $e: u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的 "松弛" 。

为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

Bellman-Ford 算法就是利用松弛操作：一开始只有 $\text{dist}[s] = 0$，其余均为 ∞。然后枚举每条边，进行松弛操作。

当然一遍松弛肯定不够。Bellman-Ford 算法会一直重复上述过程，直到 dist 没有变化为止。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

在已知当前的 dist 的情况下，再往图中加入一条边 $e : u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。

为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

Bellman-Ford 算法就是利用松弛操作：一开始只有 $\text{dist}[s] = 0$，其余均为 ∞。然后枚举每条边，进行松弛操作。

当然一遍松弛肯定不够。Bellman-Ford 算法会一直重复上述过程，直到 dist 没有变化为止。

如果图中没有边权总和为负值的环（负环），那么最短路中每个点只会经过一次，此时最短路中最多只会有 $n - 1$ 条边。如果存在负环，那么可以一直沿着负环无限走下去，每走一圈路径长度越短，因此不存在最短路。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

在已知当前的 dist 的情况下，再往图中加入一条边 $e : u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。

为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

Bellman-Ford 算法就是利用松弛操作：一开始只有 $\text{dist}[s] = 0$，其余均为 ∞。然后枚举每条边，进行松弛操作。

当然一遍松弛肯定不够。Bellman-Ford 算法会一直重复上述过程，直到 dist 没有变化为止。

如果图中没有边权总和为负值的环（负环），那么最短路中每个点只会经过一次，此时最短路中最多只会有 $n - 1$ 条边。如果存在负环，那么可以一直沿着负环无限走下去，每走一圈路径长度越短，因此不存在最短路。

Bellman-Ford 算法每次更新成功时，都会使原来的最短路长度加 1。因此，如果第 n 次更新还有变动，则可以判定图中有负环。否则更新次数不会超过 $n - 1$ 次。

松弛操作

设置起点为 s，以及数组 $\text{dist}[x]$ 表示从 s 到 x 的最短路径的长度。

在已知当前的 dist 的情况下，再往图中加入一条边 $e: u \to v$，边权为 w。有可能会有部分最短路将经过 e。这就需要看 e 能否使某些 dist 变小，也就是所谓的“松弛”。

为了代码方便，除起点外的 dist 一般初始为一个比较大的数字，如 10^9。

如果 $\text{dist}[u] + w < \text{dist}[v]$，那么先从 s 走到 u，再经过边 e，会得到一条比以前更短的路径。

Bellman-Ford 算法就是利用松弛操作：一开始只有 $\text{dist}[s] = 0$，其余均为 ∞。然后枚举每条边，进行松弛操作。

当然一遍松弛肯定不够。Bellman-Ford 算法会一直重复上述过程，直到 dist 没有变化为止。

如果图中没有边权总和为负值的环（负环），那么最短路中每个点只会经过一次，此时最短路中最多只会有 $n - 1$ 条边。如果存在负环，那么可以一直沿着负环无限走下去，每走一圈路径长度越短，因此不存在最短路。

Bellman-Ford 算法每次更新成功时，都会使原来的最短路长度加 1。因此，如果第 n 次更新还有变动，则可以判定图中有负环。否则更新次数不会超过 $n - 1$ 次。

因此时间复杂度为 $O(nm)$。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\text{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\mathrm{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

所以，设边 $e : u \to v$，每次松弛成功时，$\mathrm{dist}[v]$ 会变动，此时 v 的出边就可以执行松弛操作了。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\text{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

所以，设边 $e : u \to v$，每次松弛成功时，$\text{dist}[v]$ 会变动，此时 v 的出边就可以执行松弛操作了。

使用一个队列记录可以进行更新的点，初始时只有起点 s。当松弛边 e 成功时，就将 v 加入队列。这个算法也被叫做 SPFA（Shortest Path Faster Algorithm）。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\mathrm{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

所以，设边 $e : u \to v$，每次松弛成功时，$\mathrm{dist}[v]$ 会变动，此时 v 的出边就可以执行松弛操作了。

使用一个队列记录可以进行更新的点，初始时只有起点 s。当松弛边 e 成功时，就将 v 加入队列。这个算法也被叫做 SPFA（Shortest Path Faster Algorithm）。

如果图中没有负环，那么更新的总次数不会比原始的 Bellman-Ford 算法多，即时间复杂度上界依然为 $O(nm)$。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\operatorname{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

所以，设边 $e: u \to v$，每次松弛成功时，$\operatorname{dist}[v]$ 会变动，此时 v 的出边就可以执行松弛操作了。

使用一个队列记录可以进行更新的点，初始时只有起点 s。当松弛边 e 成功时，就将 v 加入队列。这个算法也被叫做 SPFA（Shortest Path Faster Algorithm）。

如果图中没有负环，那么更新的总次数不会比原始的 Bellman-Ford 算法多，即时间复杂度上界依然为 $O(nm)$。

如果有负环该怎么办？按照 Bellman-Ford 的理论，每个点的更新次数不会超过 $n-1$。所以记录一个 `cnt` 数组表示节点被松弛的次数。如果出现了 `cnt` 等于 n 的节点，则表明有负环。

队列优化

实际上，每次更新有很多步骤是不必要的。如果上次更新时 $\text{dist}[x]$ 没有变动，那么对于从 x 出发的边就无需松弛。

所以，设边 $e: u \to v$，每次松弛成功时，$\text{dist}[v]$ 会变动，此时 v 的出边就可以执行松弛操作了。

使用一个队列记录可以进行更新的点，初始时只有起点 s。当松弛边 e 成功时，就将 v 加入队列。这个算法也被叫做 SPFA（Shortest Path Faster Algorithm）。

如果图中没有负环，那么更新的总次数不会比原始的 Bellman-Ford 算法多，即时间复杂度上界依然为 $O(nm)$。

如果有负环该怎么办？按照 Bellman-Ford 的理论，每个点的更新次数不会超过 $n-1$。所以记录一个 `cnt` 数组表示节点被松弛的次数。如果出现了 `cnt` 等于 n 的节点，则表明有负环。

实际上，很多出题人都很懒，造的数据都是随机的，所以才有很多题目即使 $n, m \approx 10^5$ 却也能跑过。

Bellman-Ford 算法：实现

Bellman-Ford 算法：实现

没有堆优化的 Bellman-Ford 算法:

```
initialize dist with inf
dist[s] = 0
for i in [1..n - 1]:
    for e in G:
        e: u -> v, weight w
        if dist[v] > dist[u] + w:
           dist[v] = dist[u] + w
```

Bellman-Ford 算法：实现

加上堆优化的 Bellman-Ford 算法:

```
initialize dist with INF
dist[s] = 0
// 此外再记录 exist 布尔数组表示节点是否已经在队列中
Queue<int> q
q.push(s)
exist[s] = true
while q is not empty:
    u = q.pop_front()
    exist[u] = false
    for e: u -> v, w in G:
        if dist[v] > dist[u] + w:
           dist[v] = dist[u] + w
           if not exist[v]:
               exist[v] = true
               q.push(v)
```

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。
按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。
从初始状态开始（位置在起点并且时间为 0），之后每次抽出时间最短的状态，尝试向四周流水。如果流到旁边的节点的时间小于记录的 `dist`，就说明当前是更快的水流。

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。
按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。
从初始状态开始（位置在起点并且时间为 0），之后每次抽出时间最短的状态，尝试向四周流水。如果流到旁边的节点的时间小于记录的 `dist`，就说明当前是更快的水流。
使用堆来维护时间顺序即可，时间复杂度为 $O(m \log n)$。

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。
按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。
从初始状态开始（位置在起点并且时间为 0），之后每次抽出时间最短的状态，尝试向四周流水。如果流到旁边的节点的时间小于记录的 `dist`，就说明当前是更快的水流。
使用堆来维护时间顺序即可，时间复杂度为 $O(m \log n)$。

```
struct State:
    int u, t
int dist[]
Heap<State> Q  // 按 t 排序的小根堆
while Q is not empty:
    s = Q.pop()
    if s.t > dist[s.u]: continue  // 略去不必要的更新
    for w, v in G[s.u]:  // 边长为 w, 终点为 v
        if  dist[v] > dist[s.u] + w:
            dist[v] = dist[s.u] + w
            Q.push(new State(v, dist[v]))
```

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。

按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。

从初始状态开始（位置在起点并且时间为 0），之后每次抽出时间最短的状态，尝试向四周流水。如果流到旁边的节点的时间小于记录的 `dist`，就说明当前是更快的水流。

使用堆来维护时间顺序即可，时间复杂度为 $O(m \log n)$。

```
struct State:
    int u, t
int dist[]
Heap<State> Q  // 按 t 排序的小根堆
while Q is not empty:
    s = Q.pop()
    if s.t > dist[s.u]: continue  // 略去不必要的更新
    for w, v in G[s.u]:  // 边长为 w, 终点为 v
        if  dist[v] > dist[s.u] + w:
            dist[v] = dist[s.u] + w
            Q.push(new State(v, dist[v]))
```

这就是 Dijkstra 算法。

流水模型

在起点插上一根水管，如果水流速度固定，那么最短路上的水流将会最先到达终点。
按照时间顺序模拟水流。每个状态记录水流到达的节点和到达的时间。同时记录 `dist` 数组表示第一次水流到达的时间，也就是从起点开始的最短路径的长度。
从初始状态开始（位置在起点并且时间为 0），之后每次抽出时间最短的状态，尝试向四周流水。如果流到旁边的节点的时间小于记录的 `dist`，就说明当前是更快的水流。
使用堆来维护时间顺序即可，时间复杂度为 $O(m \log n)$。

```
struct State:
    int u, t
int dist[]
Heap<State> Q  // 按 t 排序的小根堆
while Q is not empty:
    s = Q.pop()
    if s.t > dist[s.u]: continue  // 略去不必要的更新
    for w, v in G[s.u]:  // 边长为 w, 终点为 v
        if  dist[v] > dist[s.u] + w:
            dist[v] = dist[s.u] + w
            Q.push(new State(v, dist[v]))
```

这就是 Dijkstra 算法。
模板题：【LG P3371】

【WF2008 K】Steam Rollar

给出一个 $R \times C$ 的带权网格图，图上部分边可能不存在。现在有一台老爷车，全速跑时每单位时间只能跑一个单位长度。坑爹的是，老爷车在起步、刹车和转弯前后的边上跑时只有半速。现在给出起点和终点，为这辆老爷车最少要多少时间才能到达终点。

$R, C \leqslant 500$

【WF2008 K】Steam Rollar

给出一个 $R \times C$ 的带权网格图，图上部分边可能不存在。现在有一台老爷车，全速跑时每单位时间只能跑一个单位长度。坑爹的是，老爷车在起步、刹车和转弯前后的边上跑时只有半速。现在给出起点和终点，为这辆老爷车最少要多少时间才能到达终点。

$R, C \leqslant 500$

如果不转弯，那么车子只能横向或者纵向移动。

【WF2008 K】Steam Rollar

给出一个 $R \times C$ 的带权网格图，图上部分边可能不存在。现在有一台老爷车，全速跑时每单位时间只能跑一个单位长度。坑爹的是，老爷车在起步、刹车和转弯前后的边上跑时只有半速。现在给出起点和终点，为这辆老爷车最少要多少时间才能到达终点。

$R, C \leqslant 500$

如果不转弯，那么车子只能横向或者纵向移动。

为每个交叉路口建两个点，分别表示老爷车在横向移动和纵向移动的情况。表示横向移动的点之间只能横向连边，表示纵向移动的点之间同理。

【WF2008 K】Steam Rollar

给出一个 $R \times C$ 的带权网格图，图上部分边可能不存在。现在有一台老爷车，全速跑时每单位时间只能跑一个单位长度。坑爹的是，老爷车在起步、刹车和转弯前后的边上跑时只有半速。现在给出起点和终点，为这辆老爷车最少要多少时间才能到达终点。

$R, C \leqslant 500$

如果不转弯，那么车子只能横向或者纵向移动。

为每个交叉路口建两个点，分别表示老爷车在横向移动和纵向移动的情况。表示横向移动的点之间只能横向连边，表示纵向移动的点之间同理。

转弯就是在两类点之间切换：只需要将转弯前后的点连起来即可，边权是实际耗时。

【WF2008 K】Steam Rollar

给出一个 $R \times C$ 的带权网格图，图上部分边可能不存在。现在有一台老爷车，全速跑时每单位时间只能跑一个单位长度。坑爹的是，老爷车在起步、刹车和转弯前后的边上跑时只有半速。现在给出起点和终点，为这辆老爷车最少要多少时间才能到达终点。

$R, C \leqslant 500$

如果不转弯，那么车子只能横向或者纵向移动。

为每个交叉路口建两个点，分别表示老爷车在横向移动和纵向移动的情况。表示横向移动的点之间只能横向连边，表示纵向移动的点之间同理。

转弯就是在两类点之间切换：只需要将转弯前后的点连起来即可，边权是实际耗时。

对于起点和终点再额外各开一个点，用于表示起步和刹车的额外耗时。

所有点对最短路

之前的算法都是单源最短路，另外有一个常用的 Floyd 算法，可以在 $O(n^3)$ 的时间内计算出所有点对间的最短路。

所有点对最短路

之前的算法都是单源最短路，另外有一个常用的 Floyd 算法，可以在 $O(n^3)$ 的时间内计算出所有点对间的最短路。

每一条路径都有起点和终点，此外还可能会有许多的中介点。Floyd 算法直接采用邻接矩阵记录原图 G，用矩阵 W 表示点对间的最短路。G 相当于不经过任何中介点的最短路矩阵，于是 Floyd 算法尝试不断加入中介点来更新最短路矩阵。

所有点对最短路

之前的算法都是单源最短路，另外有一个常用的 Floyd 算法，可以在 $O(n^3)$ 的时间内计算出所有点对间的最短路。

每一条路径都有起点和终点，此外还可能会有许多的中介点。Floyd 算法直接采用邻接矩阵记录原图 G，用矩阵 W 表示点对间的最短路。G 相当于不经过任何中介点的最短路矩阵，于是 Floyd 算法尝试不断加入中介点来更新最短路矩阵。

从 1 开始枚举新加入的中介点 k，对于任意两个点 i 和 j，其最短路有两种选择：一是保持原样，即 $W[i][j]$；二是经过新的中介点 k，即 $W[i][k]+W[k][j]$。两者取最小值即可。

所有点对最短路

之前的算法都是单源最短路，另外有一个常用的 Floyd 算法，可以在 $O(n^3)$ 的时间内计算出所有点对间的最短路。

每一条路径都有起点和终点，此外还可能会有许多的中介点。Floyd 算法直接采用邻接矩阵记录原图 G，用矩阵 W 表示点对间的最短路。G 相当于不经过任何中介点的最短路矩阵，于是 Floyd 算法尝试不断加入中介点来更新最短路矩阵。

从 1 开始枚举新加入的中介点 k，对于任意两个点 i 和 j，其最短路有两种选择：一是保持原样，即 $W[i][j]$；二是经过新的中介点 k，即 $W[i][k]+W[k][j]$。两者取最小值即可。

```
for k in [1..n]:
    for i in [1..n]:
        for j in [1..n]:
            W[i][j] = min(W[i][j], W[i][k] + W[k][j])
```

【LG P2935】Best Spot

给出一个无向带权图和 F 个特殊点，找出到这 F 个点的距离之和最短的点。

$n \leqslant 500$

【LG P2935】Best Spot

给出一个无向带权图和 F 个特殊点，找出到这 F 个点的距离之和最短的点。

$n \leqslant 500$

用 Floyd 求出所有最短路后枚举。

【JSOI 2007 / LG P1841】重要的城市

给出一张 n 个点的无向带权图，找出所有的点 x，满足删除 x 后，存在两个点之间的最短路会变长。

$n \leqslant 500$

【JSOI 2007 / LG P1841】重要的城市

给出一张 n 个点的无向带权图，找出所有的点 x，满足删除 x 后，存在两个点之间的最短路会变长。

$n \leqslant 500$

除了最短路矩阵 W 外，还记录最短路的条数矩阵 C。

【JSOI 2007 / LG P1841】重要的城市

给出一张 n 个点的无向带权图，找出所有的点 x，满足删除 x 后，存在两个点之间的最短路会变长。

$n \leqslant 500$

除了最短路矩阵 W 外，还记录最短路的条数矩阵 C。

Floyd 的过程中可以同时更新 C。

【JSOI 2007 / LG P1841】重要的城市

给出一张 n 个点的无向带权图，找出所有的点 x，满足删除 x 后，存在两个点之间的最短路会变长。

$n \leqslant 500$

除了最短路矩阵 W 外，还记录最短路的条数矩阵 C。

Floyd 的过程中可以同时更新 C。

对于点 x，如果存在 i 和 j 满足 $W[i][k] + W[k][j] = W[i][j]$ 并且 $C[i][j] = C[i][k] \cdot C[k][j]$，则点 x 是重要的。