图论 I

FOI 2022 算法夏令营基础班 Day 3

Jul 18, 2022

Contents

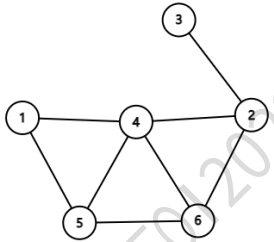
章节目录

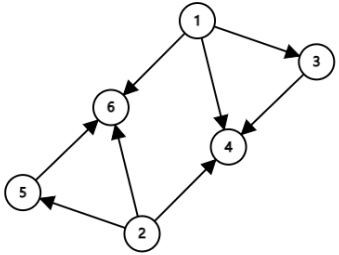
图的构成

图是由一个点集 V 和一个边集 E 构成的集合。

图的构成

图是由一个点集 V 和一个边集 E 构成的集合。
说人话版本：图是由点和边构成的。





图的分类

有向图

每一条边有一个固定的方向的图。

无向图

所有的边都没有方向的图。

混合图

有的边有方向而有的边没有方向的图。

度

对于无向图，定义一个点的**度**为这个点连接的边的数量。

度

对于无向图，定义一个点的**度**为这个点连接的边的数量。

在有向图中，一个点连接的边包括了从另一个点指向这个点和从这个点指向另一个点的情况，于是我们将度进一步细分为**入度**和**出度**。

度

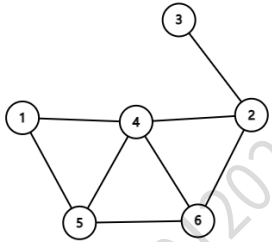
对于无向图，定义一个点的**度**为这个点连接的边的数量。

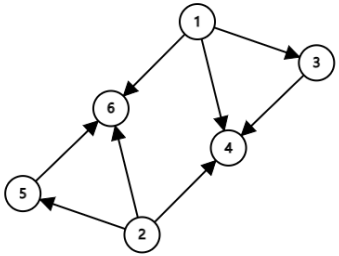
在有向图中，一个点连接的边包括了从另一个点指向这个点和从这个点指向另一个点的情况，于是我们将度进一步细分为**入度**和**出度**。

入度是指与一个点相连且指向这个点的边的数量，出度则是与一个点相连且从这个点指向其他的点的边的数量。

小测验 1

写出下面两张图中每个点的度：





点和边的基本要素

对于图上的一个点，其包含的基本信息有它的编号和它所连接的边。

点和边的基本要素

对于图上的一个点，其包含的基本信息有它的编号和它所连接的边。
图上的一条边包括的信息则有编号、连接的两点以及方向。

点和边的基本要素

对于图上的一个点，其包含的基本信息有它的编号和它所连接的边。

图上的一条边包括的信息则有编号、连接的两点以及方向。

在很多题目中，点和边上还要记录其它的信息（如边的长度等），我们把这些额外的信息称为**权值**，其中记录在点上的称为**点权**，记录在边上的称为**边权**。

概念补充

重边

在一张图中，如果存在两条边使得它们的端点和方向（针对有向图）都相同，那么图上存在重边。

概念补充

重边

在一张图中，如果存在两条边使得它们的端点和方向（针对有向图）都相同，那么图上存在重边。

自环

若一条边连接的两端点相同，那么这条边称为一个自环。

概念补充

重边

在一张图中，如果存在两条边使得它们的端点和方向（针对有向图）都相同，那么图上存在重边。

自环

若一条边连接的两端点相同，那么这条边称为一个自环。

简单图

不存在重边和自环的图称为简单图。

概念补充

完全无向图

一个由 n 个点构成且每个点的度均为 $n-1$ 的简单无向图称为完全无向图。

概念补充

完全无向图

一个由 n 个点构成且每个点的度均为 $n-1$ 的简单无向图称为完全无向图。
说人话版本：所有的点两两连接的无向图。

概念补充

完全无向图

一个由 n 个点构成且每个点的度均为 $n-1$ 的简单无向图称为完全无向图。
说人话版本：所有的点两两连接的无向图。

完全有向图（竞赛图）

给完全无向图的每一条边定方向后即可得到完全有向图（竞赛图）。

概念补充

完全无向图

一个由 n 个点构成且每个点的度均为 $n-1$ 的简单无向图称为完全无向图。
说人话版本：所有的点两两连接的无向图。

完全有向图（竞赛图）

给完全无向图的每一条边定方向后即可得到完全有向图（竞赛图）。

补图

对于两张图 A, B，如果图 A 中某两点间有连边则图 B 中这两点间没有连边，而当图 A 中某两点间没有连边时图 B 中这两点间有连边，那么图 A, B 互为补图。

章节目录

邻接矩阵

存储图的第一种方法是构建一个矩阵（称为**邻接矩阵**），若在一张有向图中存在一条从 u 到 v 的边，那么令 $map[u][v] = 1$ （或边的权值）。

邻接矩阵

存储图的第一种方法是构建一个矩阵（称为**邻接矩阵**），若在一张有向图中存在一条从 u 到 v 的边，那么令 $map[u][v] = 1$ （或边的权值）。

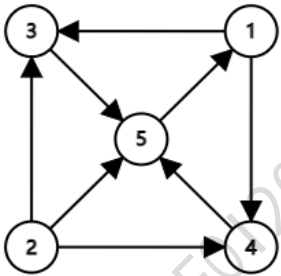
对于无向图，将每条边拆成两条方向相反的有向边即可。

邻接矩阵

存储图的第一种方法是构建一个矩阵（称为**邻接矩阵**），若在一张有向图中存在一条从 u 到 v 的边，那么令 $map[u][v] = 1$ （或边的权值）。

对于无向图，将每条边拆成两条方向相反的有向边即可。

小测验 2：写出下图对应的邻接矩阵。



邻接矩阵的优缺点分析

邻接矩阵最大的优点就是可以直观地呈现两点之间的连边情况，但与此同时也有两个不可忽视的问题：

邻接矩阵的优缺点分析

邻接矩阵最大的优点就是可以直观地呈现两点之间的连边情况，但与此同时也有两个不可忽视的问题：

1. 使用邻接矩阵记录一张包含 n 个点的图就意味着要开一个 $n \times n$ 的数组，而这在大多数题目的空间限制下是不允许的。

邻接矩阵的优缺点分析

邻接矩阵最大的优点就是可以直观地呈现两点之间的连边情况，但与此同时也有两个不可忽视的问题：

1. 使用邻接矩阵记录一张包含 n 个点的图就意味着要开一个 $n \times n$ 的数组，而这在大多数题目的空间限制下是不允许的。
2. 邻接矩阵不支持处理存在重边的情况。

邻接矩阵的优缺点分析

邻接矩阵最大的优点就是可以直观地呈现两点之间的连边情况，但与此同时也有两个不可忽视的问题：

1. 使用邻接矩阵记录一张包含 n 个点的图就意味着要开一个 $n \times n$ 的数组，而这在大多数题目的空间限制下是不允许的。

2. 邻接矩阵不支持处理存在重边的情况。

在大多数题目中，边数与点数的数量级相差不大，这就意味着邻接矩阵因存在大量的 0 而造成空间的浪费，于是我们可以考虑以边为单位存储图。

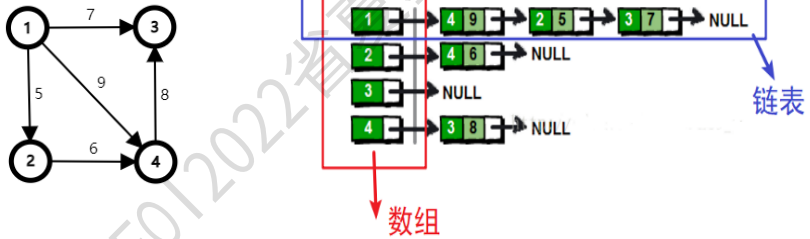
邻接表

根据上面的分析，我们可以直接记录从每个点出发有连边的点的编号以及边的边权，这样可以大大节省空间。

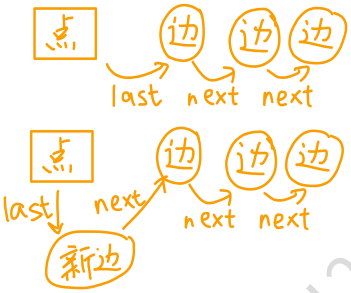
邻接表

根据上面的分析，我们可以直接记录从每个点出发有连边的点的编号以及边的边权，这样可以大大节省空间。

在具体操作过程中，可以使用链表（称为**邻接表**）来实现这一操作。

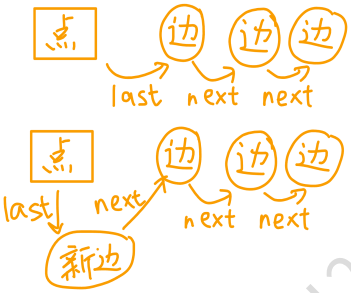


边链表（链式前向星）



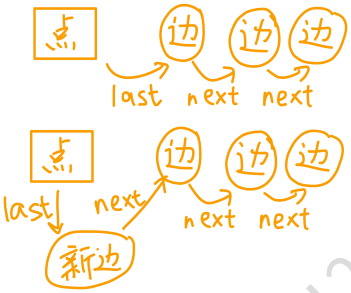
边链表是一种与邻接表原理类似的存储方法，也通过链表实现。

边链表（链式前向星）



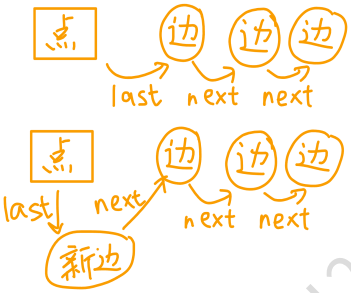
边链表是一种与邻接表原理类似的存储方法，也通过链表实现。
对于每个点 u，$last[u]$ 表示与 u 相连且最后加入的边。

边链表（链式前向星）



边链表是一种与邻接表原理类似的存储方法，也通过链表实现。
对于每个点 u，$last[u]$ 表示与 u 相连且最后加入的边。
对于每条边 l，$next[l]$ 表示 l 的起点 u 对应的链表中 l 下一条边的编号。

边链表（链式前向星）



边链表是一种与邻接表原理类似的存储方法，也通过链表实现。

对于每个点 u，$last[u]$ 表示与 u 相连且最后加入的边。

对于每条边 l，$next[l]$ 表示 l 的起点 u 对应的链表中 l 下一条边的编号。

如左图所示，在插入一条新的边的时候把原来链表中的第一条边作为当前边的下一条边，再将新加入的边作为起点的链表中的第一条边。

边链表参考代码

```
struct edge{                    //用一个结构体记录一条边
    int to;                     //边的终点
    int next;                   //链表指针，记录下一条边的编号
    int val;                    //边权
}e[size_of_m];                  //如果是无向边，数组大小要开双倍

int last[size_of_n];            //记录每个点最新加入的边的编号
int tot=0;                      //记录边的总数
```

边链表参考代码

```
9  void add_edge(int u,int v,int d) {
10     //新增一条从u到v，边权为d的边
11     ++tot;                  //新增一条边
12     e[tot].to=v;            //这条边的终点是v
13     e[tot].next=last[u];
14     last[u]=tot;            //这两句话是重点
15     //将新加入的边作为起点的链表中的第一条边
16     //把原来链表中的第一条边作为当前边的下一条边
17     e[tot].val=d;           //还有边权也要记录
18     return;
19 }
```

边链表参考代码

```
20 //遍历所有从u出发的边
21 for (int i=last[u];i;i=e[i].next) {
22     int ot=e[i].to;
23     //do sth.
24 }
```

章节目录

树的概念

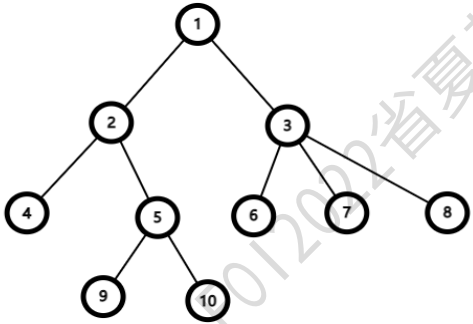
树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。

树的概念

树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。



树的概念

树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。

树的概念

树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。

根节点

根节点是树上的一个节点。一般情况下，我们从根开始对树进行遍历。

树的概念

树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。

根节点

根节点是树上的一个节点。一般情况下，我们从根开始对树进行遍历。

有根树、无根树

树根据是否存在指定的根节点，可以分为**有根树**和**无根树**。

树的概念

树

由 n 个点、$n-1$ 条边构成的**连通图**称为**树**。

根节点

根节点是树上的一个节点。一般情况下，我们从根开始对树进行遍历。

有根树、无根树

树根据是否存在指定的根节点，可以分为**有根树**和**无根树**。
对于无根树，我们一般随机选取一个点作为根然后转化为有根树处理。

树的概念

补充一个结论

树上两点之间的简单路径（不重复经过一个点的路径）是唯一的。

树的概念

补充一个结论

树上两点之间的简单路径（不重复经过一个点的路径）是唯一的。

直观来看，如果将一棵树从根节点“拎起来”，那么可以得到一个类似于现实中的树倒过来的结构。此时，如果两个点 u, v 通过一条边相连且 u 在 v 的上方，那么称 u 为 v 的父节点，v 为 u 的子节点，定义一个点的深度为这个点到根的简单路径所经过的边的数量加 1。

树的概念

补充一个结论

树上两点之间的简单路径（不重复经过一个点的路径）是唯一的。

直观来看，如果将一棵树从根节点“拎起来”，那么可以得到一个类似于现实中的树倒过来的结构。此时，如果两个点 u, v 通过一条边相连且 u 在 v 的上方，那么称 u 为 v 的父节点，v 为 u 的子节点，定义一个点的深度为这个点到根的简单路径所经过的边的数量加 1。

叶节点

没有子节点的节点称为**叶节点**。

树的概念

补充一个结论

树上两点之间的简单路径（不重复经过一个点的路径）是唯一的。

直观来看，如果将一棵树从根节点“拎起来”，那么可以得到一个类似于现实中的树倒过来的结构。此时，如果两个点 u, v 通过一条边相连且 u 在 v 的上方，那么称 u 为 v 的父节点，v 为 u 的子节点，定义一个点的深度为这个点到根的简单路径所经过的边的数量加 1。

叶节点

没有子节点的节点称为**叶节点**。

二叉树、多叉树

若树上的每个节点都至多有两个子节点，那么称这棵树为**二叉树**，否则称这棵树为**多叉树**。

树的遍历

在处理树的相关问题时，我们通常会从根节点开始对树进行深度优先遍历（类比深度优先搜索的访问顺序）。

树的遍历

在处理树的相关问题时，我们通常会从根节点开始对树进行深度优先遍历（类比深度优先搜索的访问顺序）。

如果在深度优先遍历过程中每到达一个未访问过的节点，就将它的编号记录下来，那么就可以得到这棵树的 **DFS 序**。

树的遍历

在处理树的相关问题时，我们通常会从根节点开始对树进行深度优先遍历（类比深度优先搜索的访问顺序）。

如果在深度优先遍历过程中每到达一个未访问过的节点，就将它的编号记录下来，那么就可以得到这棵树的 **DFS 序**。

如果将记录点的编号的规则变为“第一次和最后一次访问某一节点时记录其编号”，则可以得到这棵树的**欧拉序**。

树的遍历

在处理树的相关问题时，我们通常会从根节点开始对树进行深度优先遍历（类比深度优先搜索的访问顺序）。

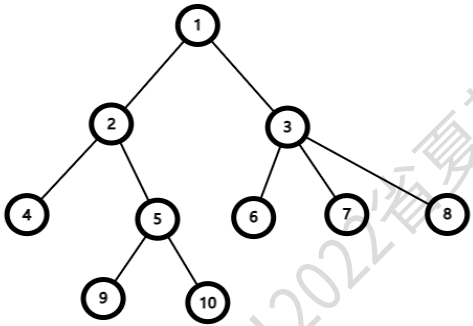
如果在深度优先遍历过程中每到达一个未访问过的节点，就将它的编号记录下来，那么就可以得到这棵树的 **DFS 序**。

如果将记录点的编号的规则变为“第一次和最后一次访问某一节点时记录其编号”，则可以得到这棵树的**欧拉序**。

注意：欧拉序的长度是节点数的 2 倍，所以在开数组的时候也要开 2 **倍的空间**。

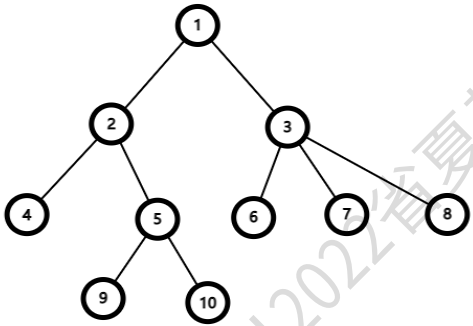
小测验 3

写出下面这棵树的 DFS 序和欧拉序。



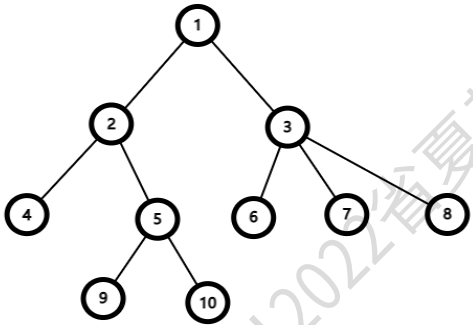
小测验 3

写出下面这棵树的 DFS 序和欧拉序。



小测验 3

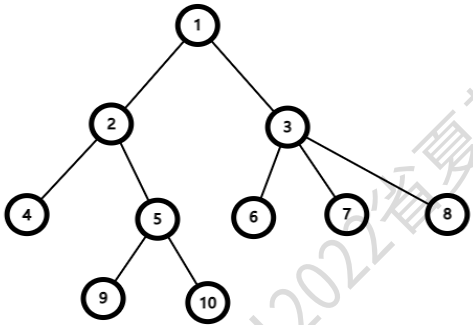
写出下面这棵树的 DFS 序和欧拉序。



DFS 序：$1, 2, 4, 5, 9, 10, 3, 6, 7, 8$。

小测验 3

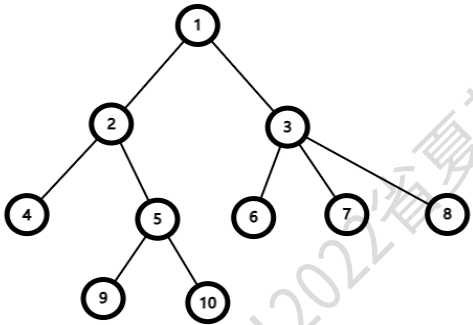
写出下面这棵树的 DFS 序和欧拉序。



DFS 序：$1, 2, 4, 5, 9, 10, 3, 6, 7, 8$。

小测验 3

写出下面这棵树的 DFS 序和欧拉序。



DFS 序：$1, 2, 4, 5, 9, 10, 3, 6, 7, 8$。
欧拉序：$1, 2, 4, 4, 5, 9, 9, 10, 10, 5, 2, 3, 6, 6, 7, 7, 8, 8, 3, 1$。

树的遍历

对于一颗二**叉树**，我们还可以定义它的前序、中序、后序遍历。

树的遍历

对于一颗二**叉树**，我们还可以定义它的前序、中序、后序遍历。
前序遍历（根左右）：先将自身记入序列，然后分别对左右子树做前序遍历。

树的遍历

对于一颗二**叉树**，我们还可以定义它的前序、中序、后序遍历。
前序遍历（根左右）：先将自身记入序列，然后分别对左右子树做前序遍历。
中序遍历（左根右）：对左子树做中序遍历后将自身记入序列，最后再遍历右子树。

树的遍历

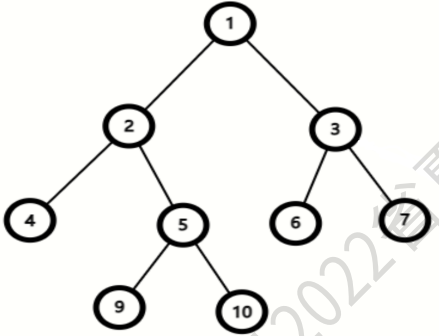
对于一颗二**叉树**，我们还可以定义它的前序、中序、后序遍历。
前序遍历（根左右）：先将自身记入序列，然后分别对左右子树做前序遍历。
中序遍历（左根右）：对左子树做中序遍历后将自身记入序列，最后再遍历右子树。
后序遍历（左右根）：先分别对左右子树做后序遍历，最后将自身记入序列。

树的遍历

对于一颗二**叉树**，我们还可以定义它的前序、中序、后序遍历。
前序遍历（根左右）：先将自身记入序列，然后分别对左右子树做前序遍历。
中序遍历（左根右）：对左子树做中序遍历后将自身记入序列，最后再遍历右子树。
后序遍历（左右根）：先分别对左右子树做后序遍历，最后将自身记入序列。
记忆技巧：根节点出现的位置决定是哪种遍历。

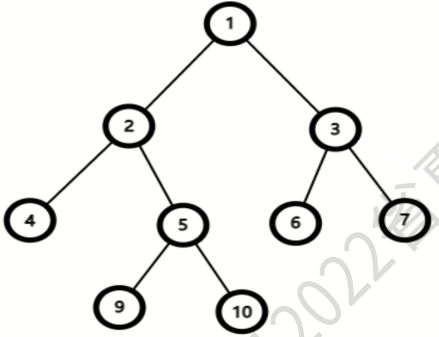
小测验 4.1

写出下面这棵树的前、中、后序遍历。



小测验 4.1

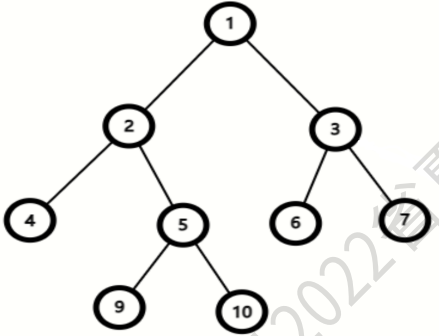
写出下面这棵树的前、中、后序遍历。



前序遍历：$1, 2, 4, 5, 9, 10, 3, 6, 7$。

小测验 4.1

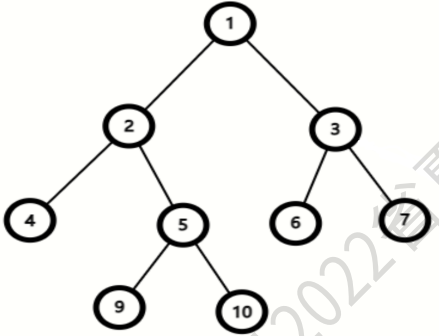
写出下面这棵树的前、中、后序遍历。



前序遍历：$1, 2, 4, 5, 9, 10, 3, 6, 7$。
中序遍历：$4, 2, 9, 5, 10, 1, 6, 3, 7$。

小测验 4.1

写出下面这棵树的前、中、后序遍历。



前序遍历：$1, 2, 4, 5, 9, 10, 3, 6, 7$。
中序遍历：$4, 2, 9, 5, 10, 1, 6, 3, 7$。
后续遍历：$4, 9, 10, 5, 2, 6, 7, 3, 1$。

小测验 4.2

给定一棵二叉树的中序遍历和后序遍历，求这棵树的前序遍历。

小测验 4.2

给定一棵二叉树的中序遍历和后序遍历，求这棵树的前序遍历。
这题的基本思路是先根据中序遍历和后序遍历将树还原出来，然后再求出前序遍历。

小测验 4.2

给定一棵二叉树的中序遍历和后序遍历，求这棵树的前序遍历。
这题的基本思路是先根据中序遍历和后序遍历将树还原出来，然后再求出前序遍历。
如果从中序遍历出发解决问题，我们会发现很难切分出左右子树对应的区间。

小测验 4.2

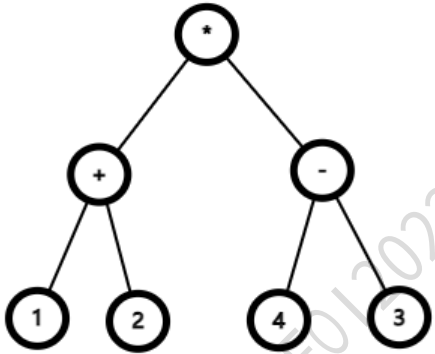
给定一棵二叉树的中序遍历和后序遍历，求这棵树的前序遍历。

这题的基本思路是先根据中序遍历和后序遍历将树还原出来，然后再求出前序遍历。

如果从中序遍历出发解决问题，我们会发现很难切分出左右子树对应的区间。

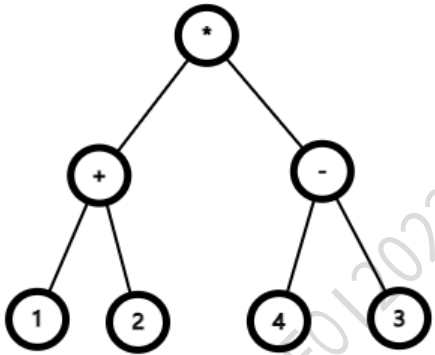
注意到后序遍历中，子树的根节点一定位于序列的末端，于是我们可以利用根节点的编号划分出中序遍历和后序遍历中对应左右子树的子序列，然后将原问题拆分为两个子问题分别递归求解。

表达式树



表达式树是一类特殊的二叉树，每个非叶节点上记录有一个运算符号（加减乘除等），每个叶节点上则记录有一个数。

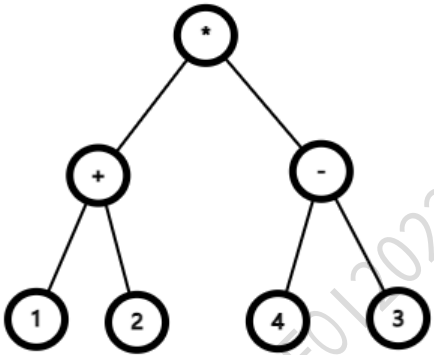
表达式树



表达式树是一类特殊的二叉树，每个非叶节点上记录有一个运算符号（加减乘除等），每个叶节点上则记录有一个数。

在计算表达式的值时，先分别计算左右子树对应的表达式的值，再将两个值通过当前节点的运算符号连接，计算出整个子树的表达式的值。

表达式树

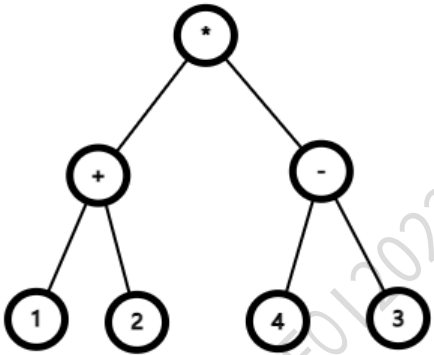


表达式树是一类特殊的二叉树，每个非叶节点上记录有一个运算符号（加减乘除等），每个叶节点上则记录有一个数。

在计算表达式的值时，先分别计算左右子树对应的表达式的值，再将两个值通过当前节点的运算符号连接，计算出整个子树的表达式的值。

表达式树的前序遍历就是前缀表达式，后序遍历就是后缀表达式。那中序遍历是我们常见（可以直接计算）的表达式吗？

表达式树



表达式树是一类特殊的二叉树，每个非叶节点上记录有一个运算符号（加减乘除等），每个叶节点上则记录有一个数。

在计算表达式的值时，先分别计算左右子树对应的表达式的值，再将两个值通过当前节点的运算符号连接，计算出整个子树的表达式的值。

表达式树的前序遍历就是前缀表达式，后序遍历就是后缀表达式。那中序遍历是我们常见（可以直接计算）的表达式吗？

不是！由于符号优先级的问题，表达式树中序遍历的结果还要再适当位置加上括号才能变成能够直接计算的表达式。

章节目录

树的直径

在数学中，圆的直径是圆内最长的弦，类似地，我们定义**树的直径**为树上最长的链。

树的直径

在数学中，圆的直径是圆内最长的弦，类似地，我们定义**树的直径**为树上最长的链。

一个结论

从树的任意一个节点出发，找出离这个点最远的一个点 u，再从 u 出发找到离它最远的点 v，路径 (u, v) 即为树的直径。

树的直径

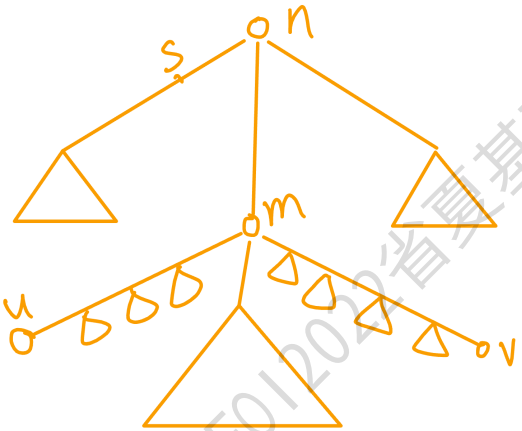
在数学中，圆的直径是圆内最长的弦，类似地，我们定义**树的直径**为树上最长的链。

一个结论

从树的任意一个节点出发，找出离这个点最远的一个点 u，再从 u 出发找到离它最远的点 v，路径 (u, v) 即为树的直径。

注意：只有当所有边的边权为正时这个结论才成立。

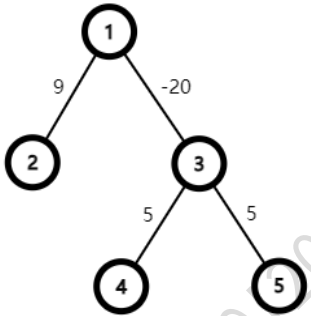
结论证明



不妨设 $dis(m,u) \le dis(m,v)$

如果边权可以为负

先给出一个用结论不能处理的反例：



用树形 DP 求树的直径

记 $f[i]$ 表示 i 的子树中以 i 为一端的最长链的长度，则可用下面这种方式求解树的直径。

```
void dfs(int u) {          //处理以点u为根的子树
    for (int i=last[u];i;i=e[i].next) {
        int v=e[i].to;   //枚举u的所有子节点v
        dfs(v);
        ans=max(ans,f[u]+f[v]+e[i].val);
        //f[u]此时表示v之前的子树中以u为一端的最长链
        //用这种方式可以用u的子树中过u的最长链更新答案
        //而且保证了这条链是简单路径
        f[u]=max(f[u],f[v]+e[i].val);
        //注意要先更新ans再更新f[u]
    }  return;
}
```

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a, b, c)，使 $dis(a, b) + dis(b, c) + dis(c, a)$ 最大。
数据范围：$3 \leq n \leq 2 \times 10^5$。

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a, b, c)，使 $dis(a, b) + dis(b, c) + dis(c, a)$ 最大。
数据范围：$3 \leq n \leq 2 \times 10^5$。
先直观感受一下，a, b, c 三个点中有两个很像是直径两端?

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a, b, c)，使 $dis(a, b) + dis(b, c) + dis(c, a)$ 最大。

数据范围：$3 \leq n \leq 2 \times 10^5$。

先直观感受一下，a, b, c 三个点中有两个很像是直径两端?

证明：考虑任取一个点 a（不一定是最终最优的那个点 a），根据树的直径的贪心结论，如果找到离点 a 最远的点 b，那么 b 必为直径的一个端点。

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a,b,c)，使 $dis(a,b)+dis(b,c)+dis(c,a)$ 最大。

数据范围：$3 \le n \le 2 \times 10^5$。

先直观感受一下，a,b,c 三个点中有两个很像是直径两端?

证明：考虑任取一个点 a（不一定是最终最优的那个点 a），根据树的直径的贪心结论，如果找到离点 a 最远的点 b，那么 b 必为直径的一个端点。

再从点 b 出发，找到一个离它最远的点 c，那么 (b,c) 是这棵树的一条直径。

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a,b,c)，使 $dis(a,b)+dis(b,c)+dis(c,a)$ 最大。

数据范围：$3 \le n \le 2 \times 10^5$。

先直观感受一下，a,b,c 三个点中有两个很像是直径两端?

证明：考虑任取一个点 a（不一定是最终最优的那个点 a），根据树的直径的贪心结论，如果找到离点 a 最远的点 b，那么 b 必为直径的一个端点。

再从点 b 出发，找到一个离它最远的点 c，那么 (b,c) 是这棵树的一条直径。

由于 a 是一个任取的点，我们可以得出在最终选点方案中 b,c 也必为直径两端点的结论。

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a,b,c)，使 $dis(a,b)+dis(b,c)+dis(c,a)$ 最大。

数据范围：$3 \le n \le 2\times 10^5$。

先直观感受一下，a,b,c 三个点中有两个很像是直径两端?

证明：考虑任取一个点 a（不一定是最终最优的那个点 a），根据树的直径的贪心结论，如果找到离点 a 最远的点 b，那么 b 必为直径的一个端点。

再从点 b 出发，找到一个离它最远的点 c，那么 (b,c) 是这棵树的一条直径。

由于 a 是一个任取的点，我们可以得出在最终选点方案中 b,c 也必为直径两端点的结论。

于是问题变为，已知两点 b,c，选出一个点 a，使 $dis(a,b)+dis(a,c)$ 最大。此时从 b,c 分别出发遍历整棵树，得到每个点到 b 和 c 的距离和，选出最大的作为点 a 即可。

「CF615F」Three Paths on a Tree

给定一棵 n 个点的树，选出三个点 (a,b,c)，使 $dis(a,b)+dis(b,c)+dis(c,a)$ 最大。

数据范围：$3 \le n \le 2\times 10^5$。

先直观感受一下，a,b,c 三个点中有两个很像是直径两端?

证明：考虑任取一个点 a（不一定是最终最优的那个点 a），根据树的直径的贪心结论，如果找到离点 a 最远的点 b，那么 b 必为直径的一个端点。

再从点 b 出发，找到一个离它最远的点 c，那么 (b,c) 是这棵树的一条直径。

由于 a 是一个任取的点，我们可以得出在最终选点方案中 b,c 也必为直径两端点的结论。

于是问题变为，已知两点 b,c，选出一个点 a，使 $dis(a,b)+dis(a,c)$ 最大。此时从 b,c 分别出发遍历整棵树，得到每个点到 b 和 c 的距离和，选出最大的作为点 a 即可。

解题技巧：对于构造、选点之类的问题，可以从一般情况入手找出通解的性质，再根据性质进一步解题。

章节目录

Floyd 算法

记 $dis[i][j]$ 为 i,j 两点间的距离（初始条件下只有当 (u,v) 间有连边时 $dis[u][v]$ 等于边权，否则等于无穷大），那么一种直观的求最短路方法是枚举一个中转的点 k，然后通过 $dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])$ 更新数组。

Floyd 算法

记 $dis[i][j]$ 为 i,j 两点间的距离（初始条件下只有当 (u,v) 间有连边时 $dis[u][v]$ 等于边权，否则等于无穷大），那么一种直观的求最短路方法是枚举一个中转的点 k，然后通过 $dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j])$ 更新数组。以下先给出主要部分的参考代码：

```
1 for (int k=1;k<=n;++k)
2     for (int i=1;i<=n;++i)
3         for (int j=1;j<=n;++j)
4             dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
```

Floyd 算法

记 $dis[i][j]$ 为 i,j 两点间的距离（初始条件下只有当 (u,v) 间有连边时 $dis[u][v]$ 等于边权，否则等于无穷大），那么一种直观的求最短路方法是枚举一个中转的点 k，然后通过 $dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j])$ 更新数组。以下先给出主要部分的参考代码：

```
1 for (int k=1;k<=n;++k)
2     for (int i=1;i<=n;++i)
3         for (int j=1;j<=n;++j)
4             dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
```

诶等等，中转点为什么要放在最外层枚举呢？如果放在最内层枚举会发生什么情况？

Floyd 算法

考虑 dis 数组的三维形式，$dis[k][i][j]$ 表示使用前 k 个点作为中转点的情况下，从 i 到 j 的最短路长度，那么有 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$。

Floyd 算法

考虑 dis 数组的三维形式，$dis[k][i][j]$ 表示使用前 k 个点作为中转点的情况下，从 i 到 j 的最短路长度，那么有 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$。

此时若 $k < i$，则 $dis[k][i][k], dis[k][k][j]$ 都已经被更新过了，不过由于不违反仅使用前 k 个点作为中转点的限制，所以也可以用 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$ 更新答案。

Floyd 算法

考虑 dis 数组的三维形式，$dis[k][i][j]$ 表示使用前 k 个点作为中转点的情况下，从 i 到 j 的最短路长度，那么有 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$。

此时若 $k < i$，则 $dis[k][i][k], dis[k][k][j]$ 都已经被更新过了，不过由于不违反仅使用前 k 个点作为中转点的限制，所以也可以用 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$ 更新答案。

若 $k > i$，则 $dis[k][i][k], dis[k][k][j]$ 都还没被更新过。于是我们可以进行状态压缩，将方程简化为 $dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])$，也就是上文中出现的形式了。

Floyd 算法

考虑 dis 数组的三维形式，$dis[k][i][j]$ 表示使用前 k 个点作为中转点的情况下，从 i 到 j 的最短路长度，那么有 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$。

此时若 $k < i$，则 $dis[k][i][k], dis[k][k][j]$ 都已经被更新过了，不过由于不违反仅使用前 k 个点作为中转点的限制，所以也可以用 $dis[k][i][j] = min(dis[k-1][i][j], dis[k][i][k] + dis[k][k][j])$ 更新答案。

若 $k > i$，则 $dis[k][i][k], dis[k][k][j]$ 都还没被更新过。于是我们可以进行状态压缩，将方程简化为 $dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])$，也就是上文中出现的形式了。

由上面的分析可知，中转点应放在最外层枚举。

Floyd 算法

那如果我偏要把中转点放在最内层枚举呢?

Floyd 算法

那如果我偏要把中转点放在最内层枚举呢?

此时由于 $dis[i][j]$ 的更新顺序混乱（无法确定 $dis[i][k]$ 和 $dis[k][j]$ 已经更新到第几次了），所以很容易导致错误。

Floyd 算法

那如果我偏要把中转点放在最内层枚举呢?

此时由于 $dis[i][j]$ 的更新顺序混乱（无法确定 $dis[i][k]$ 和 $dis[k][j]$ 已经更新到第几次了），所以很容易导致错误。

以下提供一组反例：

Floyd 算法

那如果我偏要把中转点放在最内层枚举呢?

此时由于 $dis[i][j]$ 的更新顺序混乱（无法确定 $dis[i][k]$ 和 $dis[k][j]$ 已经更新到第几次了），所以很容易导致错误。

以下提供一组反例：

显然，$dis[1][2] = 4$ 而非正无穷。

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。
时间复杂度：$\mathbb{O}(n^3)$，适用情况较少。

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。
时间复杂度：$\mathbb{O}(n^3)$，适用情况较少。
是否可以处理边权为负的情况？

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。
时间复杂度：$\mathbb{O}(n^3)$，适用情况较少。
是否可以处理边权为负的情况？可以！

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。
时间复杂度：$\mathbb{O}(n^3)$，适用情况较少。
是否可以处理边权为负的情况？可以！
是否可以处理图中由负环（环上的边的权值之和小于零）的情况？

Floyd 算法性能分析

算法特点：直接求出整张图每两个点之间的最短路。

时间复杂度：$\mathbb{O}(n^3)$，适用情况较少。

是否可以处理边权为负的情况？可以！

是否可以处理图中由负环（环上的边的权值之和小于零）的情况？不只是 Floyd 算法，所有的最短路算法都不可以，因为此时的最短路一定是从起点出发到达负环，绕负环走无数圈后再去终点，即实际上不存在最短路。

SPFA 算法

算法简介

SPFA 算法即为队列优化下的 Bellman-Ford 算法，为经典的单源最短路算法，曾在 OI 中有广泛的应用，不过由于其复杂度具有不确定性且最坏情况下复杂度过高，在卡 SPFA 的题目不断增多的环境下被使用的频率逐渐降低。

SPFA 算法

算法简介

SPFA 算法即为队列优化下的 Bellman-Ford 算法，为经典的单源最短路算法，曾在 OI 中有广泛的应用，不过由于其复杂度具有不确定性且最坏情况下复杂度过高，在卡 SPFA 的题目不断增多的环境下被使用的频率逐渐降低。

算法流程

1. 记 $dis[i]$ 表示从源点（出发点）到点 i 的最短路的长度，初始时除了 $dis[s] = 0$，其它所有的 dis 均为正无穷。同时维护一个队列 q，初始时只有一个元素 s。

SPFA 算法

算法简介

SPFA 算法即为队列优化下的 Bellman-Ford 算法，为经典的单源最短路算法，曾在 OI 中有广泛的应用，不过由于其复杂度具有不确定性且最坏情况下复杂度过高，在卡 SPFA 的题目不断增多的环境下被使用的频率逐渐降低。

算法流程

1. 记 $dis[i]$ 表示从源点（出发点）到点 i 的最短路的长度，初始时除了 $dis[s] = 0$，其它所有的 dis 均为正无穷。同时维护一个队列 q，初始时只有一个元素 s。
2. 每次从队列取出第一个元素，枚举它的所有（出）边，尝试用它更新相邻的点的最短路，若更新成功且被更新的点不在队列中，就将它加入队列。

SPFA 算法

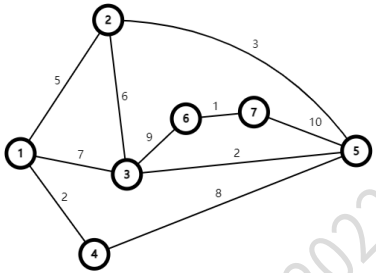
算法简介

SPFA 算法即为队列优化下的 Bellman-Ford 算法，为经典的单源最短路算法，曾在 OI 中有广泛的应用，不过由于其复杂度具有不确定性且最坏情况下复杂度过高，在卡 SPFA 的题目不断增多的环境下被使用的频率逐渐降低。

算法流程

1. 记 $dis[i]$ 表示从源点（出发点）到点 i 的最短路的长度，初始时除了 $dis[s] = 0$，其它所有的 dis 均为正无穷。同时维护一个队列 q，初始时只有一个元素 s。
2. 每次从队列取出第一个元素，枚举它的所有（出）边，尝试用它更新相邻的点的最短路，若更新成功且被更新的点不在队列中，就将它加入队列。
3. 不断重复步骤 2 直至队列为空（所有的点的最短路已经达到最优，无法继续更新）。

算法演示



参考代码

```
queue<int> q;  while (!q.empty()) q.pop();//清空队列
memset(dis,0x3f,sizeof(dis));//注意不要用0x7f，否则可能会溢出
memset(inq,0,sizeof(inq));//所有的点标记为不在队列中
dis[s]=0,q.push(s),inq[s]=true;//起点初始化
while (!q.empty()) {  int cur=q.front();//队首拿出一个点
    q.pop(),inq[cur]=false;//出队
    for (int i=last[cur];i;i=e[i].next) {
        int ot=e[i].to;//枚举所有的（出）边
        if (dis[cur]+e[i].val<dis[ot]) {//可以更新
            dis[ot]=dis[cur]+e[i].val;//更新dis
            if (!inq[ot]) q.push(ot),inq[ot]=1;//入队
        }
    }  return;
}
```

SPFA 算法的时间复杂度问题

SPFA 算法的时间复杂度为 $\mathbb{O}(kE)$，其中 k 为平均进队次数，由图的性质决定。通过进一步的计算还可得出期望复杂度为 $\mathbb{O}(n \log n \log \frac{m}{n} + m)$，在出题人比较善良的时候跑起来还挺快的。

SPFA 算法的时间复杂度问题

SPFA 算法的时间复杂度为 $\mathbb{O}(kE)$，其中 k 为平均进队次数，由图的性质决定。通过进一步的计算还可得出期望复杂度为 $\mathbb{O}(n \log n \log \frac{m}{n} + m)$，在出题人比较善良的时候跑起来还挺快的。

不过在大多数数据强度较大的题目中，出题人都会通过对图的构造来加大平均进队次数从而提高 SPFA 算法的复杂度。由于 SPFA 算法的复杂度最高可以达到 $\mathbb{O}(mn)$，故不建议在考场使用。

Dijkstra 算法

和 SPFA 算法一样，Dijkstra 算法也是一个单源最短路算法，但它具有稳定且优秀的时间复杂度，所以在 OI 中更为常见。

Dijkstra 算法

和 SPFA 算法一样，Dijkstra 算法也是一个单源最短路算法，但它具有稳定且优秀的时间复杂度，所以在 OI 中更为常见。

算法流程

Dijkstra 算法和 SPFA 算法步骤类似，但是在选一个点来更新其周围的点的最短路时不再直接选用队首的点，而是维护一个小根堆，每次选取当前堆中 dis 最小的点。

Dijkstra 算法

和 SPFA 算法一样，Dijkstra 算法也是一个单源最短路算法，但它具有稳定且优秀的时间复杂度，所以在 OI 中更为常见。

算法流程

Dijkstra 算法和 SPFA 算法步骤类似，但是在选一个点来更新其周围的点的最短路时不再直接选用队首的点，而是维护一个小根堆，每次选取当前堆中 dis 最小的点。
注意：Dijikstra 算法在选取 dis 最小的点来更新的时候已经使用了贪心的思想，所以 Dijkstra 算法只有在**所有边权非负**的时候才能使用。

Dijkstra 算法

和 SPFA 算法一样，Dijkstra 算法也是一个单源最短路算法，但它具有稳定且优秀的时间复杂度，所以在 OI 中更为常见。

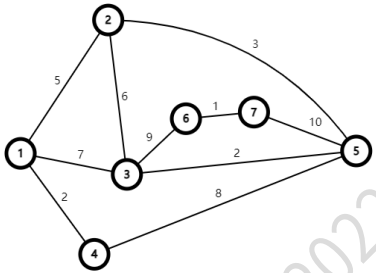
算法流程

Dijkstra 算法和 SPFA 算法步骤类似，但是在选一个点来更新其周围的点的最短路时不再直接选用队首的点，而是维护一个小根堆，每次选取当前堆中 dis 最小的点。

注意：Dijikstra 算法在选取 dis 最小的点来更新的时候已经使用了贪心的思想，所以 Dijkstra 算法只有在**所有边权非负**的时候才能使用。

由于每次都选择 dis 最小的点来更新，每个点不会因为多次更新而浪费时间，总的时间复杂度为 $\mathbb{O}(m \log n)$。

算法演示



参考代码

```
priority_queue<pair<int,int> > q;//q是一个二元数对的大根堆
while (!q.empty()) q.pop();q.push(make_pair(0,s));
//第一维是距离的相反数（这样改用大根堆写起来方便），第二维是编号
memset(dis,0x3f,sizeof(dis)),dis[s]=0;//dis数组初始化
while (!q.empty()) {//↓找到dis最小的点，注意diss这边有个负号
    int cur=q.top().second,diss=-q.top().first;q.pop();
    if (diss>dis[cur]) continue;//当前情况已经不是最优解了，跳过
    for (int i=last[cur];i;i=e[i].next)
        if (e[i].val+diss<dis[e[i].to]) {
            dis[e[i].to]=e[i].val+diss;//更新dis
            q.push(make_pair(-dis[e[i].to],e[i].to));//注意负号
        }
}
```

最短路算法总结

求两两之间最短路：

最短路算法总结

求两两之间最短路：Floyd

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra
有负边权可以考虑：

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra
有负边权可以考虑：SPFA

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra
有负边权可以考虑：SPFA
边权均为 1 还可用：

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra
有负边权可以考虑：SPFA
边权均为 1 还可用：BFS

最短路算法总结

求两两之间最短路：Floyd
非负边权才能用：Dijkstra
有负边权可以考虑：SPFA
边权均为 1 还可用：BFS

上面讲的三种算法都是处理最短路问题的工具，在解决相关问题时，用来跑最短路的图往往不知直接给出的，而需要自行构造或者在原图基础上变形才能跑求最短路的算法。

小测验 5.1

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

小测验 5.1

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

1. 输出一条从 S 到 T 的最短路。

小测验 5.1

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

1. 输出一条从 S 到 T 的最短路。

每次更新最短路的时候记录下这个点的最短路是被那个点更新的，最后就可以从 T 一路倒着找出这条路径了。

小测验 5.1

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

1. 输出一条从 S 到 T 的最短路。

每次更新最短路的时候记录下这个点的最短路是被那个点更新的，最后就可以从 T 一路倒着找出这条路径了。

2. 求图上最短路有多少条?

小测验 5.1

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

1. 输出一条从 S 到 T 的最短路。

每次更新最短路的时候记录下这个点的最短路是被那个点更新的，最后就可以从 T 一路倒着找出这条路径了。

2. 求图上最短路有多少条?

再开个数组 cnt 表示这个点有多少种满足 $dis(s,i) = dis[i]$ 的路径并再更新 dis 的同时更新即可。

小测验 5.2

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

小测验 5.2

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

3. 求图上经过边数最少的最短路有多少条?

小测验 5.2

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

3. 求图上经过边数最少的最短路有多少条?

在每条边的边权上加一个很小的数（如 10^{-7}），这样既不会影响原图上最短路的计算，同时可以让经过边数少的路径以微弱的优势成为新的最短路，此时再用上题的方法求解即可。

小测验 5.2

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

3. 求图上经过边数最少的最短路有多少条?

在每条边的边权上加一个很小的数（如 10^{-7}），这样既不会影响原图上最短路的计算，同时可以让经过边数少的路径以微弱的优势成为新的最短路，此时再用上题的方法求解即可。

4. 判断每条边是否可能出现在某条最短路中。

小测验 5.2

给定一个 n 个点 m 条边的有向图，给定一个源点（出发点）S 和一个汇点（终点）T，要求在 $\mathbb{O}(m \log n)$ 的时间复杂度要求下完成以下任务。

3. 求图上经过边数最少的最短路有多少条?

在每条边的边权上加一个很小的数（如 10^{-7}），这样既不会影响原图上最短路的计算，同时可以让经过边数少的路径以微弱的优势成为新的最短路，此时再用上题的方法求解即可。

4. 判断每条边是否可能出现在某条最短路中。

（反向跑图）以 S 和 T 为源点分别跑出到每个点的最短路，对于边长为 d 的一条边 (u, v)，若 $dis(S, T) = dis(S, u) + d + dis(v, T)$ 则这条边可能出现再最短路中。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

从 n 开始反向跑图。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

从 n 开始反向跑图。

2. $1 \sim 10$ 号点到 $n-9 \sim n$ 号点的最短路。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

从 n 开始反向跑图。

2. $1 \sim 10$ 号点到 $n-9 \sim n$ 号点的最短路。

大力出奇迹，从 $1 \sim 10$ 号点开始分别跑一遍最短路。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

从 n 开始反向跑图。

2. $1 \sim 10$ 号点到 $n-9 \sim n$ 号点的最短路。

大力出奇迹，从 $1 \sim 10$ 号点开始分别跑一遍最短路。

3. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 $\lfloor \frac{2n}{3} \rfloor \sim n$ 号点的最短路。

小测验 6

给定一个 n 个点 m 条边的有向图，求：

1. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 n 号点的最短路。

从 n 开始反向跑图。

2. $1 \sim 10$ 号点到 $n-9 \sim n$ 号点的最短路。

大力出奇迹，从 $1 \sim 10$ 号点开始分别跑一遍最短路。

3. $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点到 $\lfloor \frac{2n}{3} \rfloor \sim n$ 号点的最短路。

（超级源点、超级汇点）假设一个 0 号点作为源点，向 $1 \sim \lfloor \frac{n}{3} \rfloor$ 号点连长度为 0 的边，同理从 $\lfloor \frac{2n}{3} \rfloor \sim n$ 号点向一个自行添加的汇点 $n+1$ 连长度为 0 的边，然后跑从 0 到 $n+1$ 的最短路即可。

「AcWing 383」观光

给定一张 n 个点、m 条边有向图，求从 s 到 t 的路径中有多少条的长度不超过最短路的长度加 1。

数据范围：$2 \le n \le 1\,000, 1 \le m \le 10^4$。

「AcWing 383」观光

给定一张 n 个点、m 条边有向图，求从 s 到 t 的路径中有多少条的长度不超过最短路的长度加 1。

数据范围：$2 \le n \le 1\,000, 1 \le m \le 10^4$。

在 Dijkstra 算法上稍作修改，每个点在记录最短路长度和数量的基础上再记录次短路长度和数量，若从 s 到 t 的次短路满足条件，那么总方案数为最短路数加次短路数，否则总方案数为最短路数。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \leq n \leq 10^5, 0 \leq m \leq 2 \times 10^5$。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \leq n \leq 10^5, 0 \leq m \leq 2 \times 10^5$。

这题一个最直接的做法是在每一个同色的连通块的点之间两两连边权为 1 的边然后跑最短路，不过只需要一个依次连接 1 到 n 号点的同色的链就可以让边的数量变成 $\mathbb{O}(n^2)$ 级别，显然无法通过。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \le n \le 10^5, 0 \le m \le 2 \times 10^5$。

这题一个最直接的做法是在每一个同色的连通块的点之间两两连边权为 1 的边然后跑最短路，不过只需要一个依次连接 1 到 n 号点的同色的链就可以让边的数量变成 $\mathbb{O}(n^2)$ 级别，显然无法通过。

既然不能在边上找到突破口，我们就再来看看点。颜色段数的增加并不是因为走了某一条路径，而是走了一条边之后在某个点 u 选择走另一条不同颜色的路径。所以我们只需要将这个贡献在 u 计算就可以了。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \le n \le 10^5, 0 \le m \le 2 \times 10^5$。

这题一个最直接的做法是在每一个同色的连通块的点之间两两连边权为 1 的边然后跑最短路，不过只需要一个依次连接 1 到 n 号点的同色的链就可以让边的数量变成 $\mathbb{O}(n^2)$ 级别，显然无法通过。

既然不能在边上找到突破口，我们就再来看看点。颜色段数的增加并不是因为走了某一条路径，而是走了一条边之后在某个点 u 选择走另一条不同颜色的路径。所以我们只需要将这个贡献在 u 计算就可以了。

于是我们可以定出一个新的思路：可以建出一张分层图，点 (i, j) 表示在只保留颜色 i 的情况下的点 j。如果原图上 u, v 两点之间存在颜色为 i 的边，那么就在 $(i, u), (i, v)$ 之间连边权为 0 的边，在每个点的不同有连边的层之间两两连边权为 1 的边，之后就可以在图上跑最短路了。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \leq n \leq 10^5, 0 \leq m \leq 2 \times 10^5$。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \le n \le 10^5, 0 \le m \le 2 \times 10^5$。

但是这样的算法还存在一个问题，如果存在一个点连了 m 条不同颜色的边，那么这个点内部的边的数量就会有 $\mathbb{O}(n^2)$ 级别，还是不行。

「ARC061E」Snuke's Subway Trip

给定一张 n 个点、m 条边无向图，每条边有一种颜色，求从点 1 到点 n 的所有路径中经过颜色段数最少的路径要经过几段不同的颜色，如“红-橙-黄-黄-红”算作 4 段。

数据范围：$2 \le n \le 10^5, 0 \le m \le 2 \times 10^5$。

但是这样的算法还存在一个问题，如果存在一个点连了 m 条不同颜色的边，那么这个点内部的边的数量就会有 $\mathbb{O}(n^2)$ 级别，还是不行。

注意到在某一个点改走不同颜色的边的时候，我们并不在意原来和后来要走什么颜色的边，而只在意走的边的颜色在这个点是否发生了变化。于是我们可以进一步优化，对每个点 x 在建立一个特殊的中转点 $(0, x)$，这个点和每个有连边的层对应的点 x 之间连长度为 0.5 的边，这样一旦发生颜色的变化（假设从颜色 a 变为 b，那么在新的图中就会走出 $(a, x) \to (0, x) \to (b, x)$ 的路径，对总的最短路长度的贡献也是 1。此时再跑最短路即可。

章节目录

负环

负环

负环（又称**负权环**、**负权回路**），是指图中一个各边的边权之和小于 0 的回路。

负环

负环

负环（又称**负权环**、**负权回路**），是指图中一个各边的边权之和小于 0 的回路。

判断方法

对原图跑 SPFA 算法，同时记录每个点的进队次数，若存在某个点进队超过 n（总的点数）次，那么图中存在负环。

负环

负环

负环（又称**负权环**、**负权回路**），是指图中一个各边的边权之和小于 0 的回路。

判断方法

对原图跑 SPFA 算法，同时记录每个点的进队次数，若存在某个点进队超过 n（总的点数）次，那么图中存在负环。

正确性证明：如果图中不存在负环，那么每个点的最短路最多只会被剩下的 $n-1$ 个点各更新一次而表现为进队 $n-1$ 次。故进队次数大于等于 n 时图中必定存在负环。

负环

负环

负环（又称**负权环**、**负权回路**），是指图中一个各边的边权之和小于 0 的回路。

判断方法

对原图跑 SPFA 算法，同时记录每个点的进队次数，若存在某个点进队超过 n（总的点数）次，那么图中存在负环。

正确性证明：如果图中不存在负环，那么每个点的最短路最多只会被剩下的 $n-1$ 个点各更新一次而表现为进队 $n-1$ 次。故进队次数大于等于 n 时图中必定存在负环。

附赠模板题：「USACO 2006 Dec. (Gold)」Wormholes

「Ulm Local 1996」Arbitrage

有 n 中货币和 m 条兑换规则，每条规则形如 1 单位的 i 货币可以兑换 $w_{i,j}$ 单位的 j 货币，求是否存在一种情况使得在拥有 1 单位某种货币的情况下，在经过若干次兑换后，可以获得超过 1 单位的该种货币。

数据范围：$1 \leq n \leq 100$。

「Ulm Local 1996」Arbitrage

有 n 中货币和 m 条兑换规则，每条规则形如 1 单位的 i 货币可以兑换 $w_{i,j}$ 单位的 j 货币，求是否存在一种情况使得在拥有 1 单位某种货币的情况下，在经过若干次兑换后，可以获得超过 1 单位的该种货币。

数据范围：$1 \leq n \leq 100$。

这题本质上问的是图上是否存在一个环，使得环上的各边边权之积大于 1。

「Ulm Local 1996」Arbitrage

有 n 中货币和 m 条兑换规则，每条规则形如 1 单位的 i 货币可以兑换 $w_{i,j}$ 单位的 j 货币，求是否存在一种情况使得在拥有 1 单位某种货币的情况下，在经过若干次兑换后，可以获得超过 1 单位的该种货币。

数据范围：$1 \leq n \leq 100$。

这题本质上问的是图上是否存在一个环，使得环上的各边边权之积大于 1。

联系刚才说过的负环问题，我们更希望求的是环上边权之和而不是乘积，于是考虑将所有的边权取对数（此处要注意精度误差），然后问题就转化为图上是否存在边权之和为正的环了。再将所有的边权取相反数，即可用判断负环的方法进行判断。

章节目录

差分约束系统

在图论中，点不仅可以表示某一个位置，还可以表示一个数值或一种状态，而边也可以表示为状态之间的转移。

差分约束系统

在图论中，点不仅可以表示某一个位置，还可以表示一个数值或一种状态，而边也可以表示为状态之间的转移。

差分约束系统

有 n 个变量 $x_1 \sim x_n$ 和 m 个形如 $x_{i1} - x_{i2} \leq l_i$ 的不等式，求一组可行解。

差分约束系统

在图论中，点不仅可以表示某一个位置，还可以表示一个数值或一种状态，而边也可以表示为状态之间的转移。

差分约束系统

有 n 个变量 $x_1 \sim x_n$ 和 m 个形如 $x_{i1} - x_{i2} \leq l_i$ 的不等式，求一组可行解。

观察不等式 $x_{i1} - x_{i2} \leq l_i$，我们会发现它和上面刚刚提到的 $dis[v] \leq dis[u] + dis(u, v)$ 十分相似。

差分约束系统

在图论中，点不仅可以表示某一个位置，还可以表示一个数值或一种状态，而边也可以表示为状态之间的转移。

差分约束系统

有 n 个变量 $x_1 \sim x_n$ 和 m 个形如 $x_{i1} - x_{i2} \leq l_i$ 的不等式，求一组可行解。

观察不等式 $x_{i1} - x_{i2} \leq l_i$，我们会发现它和上面刚刚提到的 $dis[v] \leq dis[u] + dis(u, v)$ 十分相似。

于是可以建一张图，用源点到点 i 的最短路距离 $dis[i]$ 表示 x_i 的值，并在点 $i2, i1$ 之间连上长度为 l_i 的有向边即可保证不等式 $x_{i1} - x_{i2} \leq l_i$ 的成立。

差分约束系统

在图论中，点不仅可以表示某一个位置，还可以表示一个数值或一种状态，而边也可以表示为状态之间的转移。

差分约束系统

有 n 个变量 $x_1 \sim x_n$ 和 m 个形如 $x_{i1} - x_{i2} \leq l_i$ 的不等式，求一组可行解。

观察不等式 $x_{i1} - x_{i2} \leq l_i$，我们会发现它和上面刚刚提到的 $dis[v] \leq dis[u] + dis(u, v)$ 十分相似。

于是可以建一张图，用源点到点 i 的最短路距离 $dis[i]$ 表示 x_i 的值，在点 $i2, i1$ 之间连上长度为 l_i 的有向边即可保证不等式 $x_{i1} - x_{i2} \leq l_i$ 的成立。

若此时图上存在正环或者负环，那么一定无解（正环和负环可以使用 SPFA 判断），如果需要给出可行解，则可以通过 dis 的值得出答案。

小测验 7:「SCOI2011」糖果

用差分约束系统可以处理的形式（形如 $x_1 - x_2 \leq d$）描述以下关系（假设 x 为整数）：

小测验 7:「SCOI2011」糖果

用差分约束系统可以处理的形式（形如 $x_1 - x_2 \leq d$）描述以下关系（假设 x 为整数）：

1. $x_1 = x_2$：

小测验 7:「SCOI2011」糖果

用差分约束系统可以处理的形式（形如 $x_1 - x_2 \le d$）描述以下关系（假设 x 为整数）：

1. $x_1 = x_2$：$x_1 - x_2 \le 0, x_2 - x_2 \le 0$

小测验 7:「SCOI2011」糖果

用差分约束系统可以处理的形式（形如 $x_1 - x_2 \le d$）描述以下关系（假设 x 为整数）：

1. $x_1 = x_2$：$x_1 - x_2 \le 0, x_2 - x_2 \le 0$
2. $x_1 < x_2$：

小测验 7:「SCOI2011」糖果

用差分约束系统可以处理的形式（形如 $x_1 - x_2 \le d$）描述以下关系（假设 x 为整数）：

1. $x_1 = x_2$：$x_1 - x_2 \le 0, x_2 - x_2 \le 0$
2. $x_1 < x_2$：$x_1 - x_2 \le -1$

最短路算法技术总结

跑图：反向跑图、正反跑图、多源点跑图

最短路算法技术总结

跑图：反向跑图、正反跑图、多源点跑图
建图：超级源点、超级汇点、拆点、拆边

最短路算法技术总结

跑图：反向跑图、正反跑图、多源点跑图
建图：超级源点、超级汇点、拆点、拆边
算法修修改改：Floyd 状态妙用、SPFA 判断负环、Dijkstra 计数记路径

最短路算法技术总结

跑图：反向跑图、正反跑图、多源点跑图
建图：超级源点、超级汇点、拆点、拆边
算法修修改改：Floyd 状态妙用、SPFA 判断负环、Dijkstra 计数记路径
经典应用：最短路树、负环、差分约束系统

最短路算法技术总结

跑图：反向跑图、正反跑图、多源点跑图
建图：超级源点、超级汇点、拆点、拆边
算法修修改改：Floyd 状态妙用、SPFA 判断负环、Dijkstra 计数记路径
经典应用：最短路树、负环、差分约束系统
点还可以表示状态、边可以表示状态之间的转移

完结撒花