编程学习笔记

Awsdkl

2024 年 10 月 25 日

目录

Part I

数学

Chapter 1

数论

1.1 最大公约数

1.1.1 定义

最大公约数（Greatest Common Divisor），常缩写为 gcd。一组整数的公约数，即同时是这组数中每一个数的约数的数。± 1 是任意一组整数的公约数。一组整数的最大公约数，是指所有公约数中最大的一个。我们对于不全为 0 的两个整数 a, b，将其最大公约数记作 $\gcd(a, b)$。对于不全为 0 的 n 个整数 $a_1, a_2, a_3 ... a_n$，将其最大公约数记作 $\gcd(a_1, a_2, a_3 ... a_n)$。

那么如何计算最大公约数呢？我们先考虑两个数的情况。

1.1.2 欧几里得算法

过程

已知我们有两个数 a, b，我们要求出其 $\gcd(a, b)$。
注意到：$\gcd(a, b) = \gcd(b, a \bmod b)$。
并且当 b 为 0 是，两数的最大公约数就是 a，且两数大小均减小，我们可以写出以下式子：

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases} \tag{1.1}$$

证明

设：$a = b \times k + c$，显然，$c = a \bmod b$。
设：$d|a, d|b$，即 d 为 a, b 的公约数。
则 $c = a - b \times k$，两边同时除以 d 后，得：$\frac{c}{d} = \frac{a}{d} - \frac{b}{d} \times k$。
显然，$\frac{c}{d}$ 也是一个整数。所以 d 也是 b, c 的公约数
可得：对于 a 和 b 的公约数，它也会是 b 和 $a \bmod b$ 的公约数。

反过来也需要证明：
设：$d|b, d|(a \bmod b)$。

我们仍可以得到与之前类似的式子：$\frac{a \bmod b}{d} = \frac{a}{d} - \frac{b}{d} \times k$，推出 $\frac{a \bmod b}{d} + \frac{b}{d} \times k = \frac{a}{d}$。
显然左边式子为整数，则 $\frac{a}{d}$ 为整数。
可得：b 和 $a \bmod b$ 的公约数也是 a 和 b 的公约数。

既然两式公约数相同，那么最大公约数也会相同。
则可得到式子 $\gcd(a,b) = \gcd(b, a \bmod b)$。

实现

```
int gcd(int a,int b)
{
    return !b ? a : gcd(b,a % b);
}
```

1.2 裴蜀定理

1.2.1 内容

设 a,b 是不全为零的整数，对任意整数 x,y，满足 $\gcd(a,b) \mid ax+by$，且存在整数 x,y，使得 $ax+by = \gcd(a,b)$。

1.3 乘法逆元

1.3.1 定义

如果一个线性同余方程 $ax \equiv 1 \pmod b$，则 x 称为 $a \bmod b$ 的逆元，记作 a^{-1}。

猜你不知道逆元有什么用。我们知道乘法是可以直接取模的，但是在涉及到除法的时候取模会发生错误。应此就有了逆元。在模 b 的意义下除以 a，就等于在模 b 的意义下将原数乘上 $a \bmod b$ 的逆元。逆元就相当于在模意义下的倒数，所以才会被记作 a^{-1}。

1.3.2 扩展欧几里得算法

扩展欧几里得算法（Extended Euclidean algorithm, EXGCD），常用于求 $ax+by = \gcd(a,b)$ 的一组可行性解。也可以用于求解乘法逆元。

过程

对于式子 $ax \equiv 1 \pmod b$，我们可以将其改写为：$ax+by = 1$，显然在这里，一定有 $\gcd(a,b) = 1$，因此求出第二个式子的解，其中的 x 就是 a 的乘法逆元。
设：
$ax_1 + by_1 = \gcd(a,b)$
$bx_2 + (a \bmod b)y_2 = \gcd(b, a \bmod b)$
在欧几里得算法中我们知道：$\gcd(a,b) = \gcd(b, a \bmod b)$
所以 $ax_1 + by_1 = bx_2 + (a \bmod b)y_2$

又因为 $a \bmod b = a - (\lfloor \frac{a}{b} \rfloor \times b)$
所以 $ax_1 + by_1 = bx_2 + (a - (\lfloor \frac{a}{b} \rfloor \times b))y_2$
推出 $ax_1 + by_1 = ay_2 + bx_2 - \lfloor \frac{a}{b} \rfloor \times b \times y_2 = ay_2 + b(x_2 - \lfloor \frac{a}{b} \rfloor \times y_2)$
因为 $a = a, b = b$，所以 $x_1 = y_2, y_1 = x_2 - \lfloor \frac{a}{b} \rfloor \times y_2$
所以我们可以将 x_2, y_2 带入递归中，直至求出 gcd，然后再递归 $x = 1, y = 0$ 回去求解。

实现

```
typedef long long ll;
void exgcd(ll a,ll b,ll &x,ll &y)
{
    if(b == 0)
    {
        x = 1,y = 0;
        return;
    }
    exgcd(b,a%b,y,x);
    y -= a / b * x;
}
```

Part II

数据结构

Chapter 2

并查集

2.1 并查集

2.1.1 实现

解释

fa：表示第 i 个点的父亲。

代码

fa 数组记得初始化。

```
int fa[(int)1e4+5];

int Find(int x)
{
    return x == fa[x]?x:fa[x] = Find(fa[x]);
}

int merge(int x,int y)
{
    int a = Find(x);
    int b = Find(y);
    if(a != b)
    {
        fa[b] = a;
    }
}
```

2.2 带权并查集

Part III

图论

一些常用概念:

时间戳，表示**每个点被第一次访问的时间** (可以简单理解为是第几个被访问的点)。我们用 **dfn** （dfs number）数组记录每个点的时间戳，即 dfn_i 表示第一次访问结点 i 的时间。若结点 i 未被访问，则 $\mathrm{dfn}_i = 0$，因此通常也可以用 dfn 数组判断一个结点是否被访问过。

Chapter 3

连通性相关

3.1 相关定义与前置知识

无向图的连通性主要研究割点和割边。

- 割点：在无向图中，删去该点后会使连通分量数增加的点为 **割点**。
- 割边：在无向图中，删去该边后会使连通分量数增加的边为 **割边**，也称作 **桥**。

孤立点和孤立边上的端点都不是给点，但孤立边是割边（这其实是显然的）。

显然，割点和割边重要的原因就是删去他们后相较于非割点和非割边对图的连通性有更大的影响。下面还有几个概念，都是基于刚才的割点和割边的。

- 点双连通图：不存在割点的无向连通图被称为 **点双连通图**。孤立点和孤立边均为点双连通图。
- 边双连通图：不存在割边的无向连通图被称为 **边双连通图**。孤立点是边双连通图，孤立边不是。
- 点双连通分量：一张图的极大点双连通子图称为 **点双连通分量（V-BCC）**，简称 **点双**。
- 边双连通分量：一张图的极大边双连通子图称为 **边双连通分量（E-BCC）**，简称 **边双**。

在连通性这一块，我们用的均为 Tarjan 算法。因此，还需要知道 **DFS 生成树**。

有向图的 DFS 生成树主要有 4 种边（不一定全部出现）：

- 树边（tree edge）：绿色边，每次搜索找到一个还没有访问过的结点时就形成了一条树边。
- 返祖边（back edge）：黄色边，即指向祖先结点的边（Tarjan 中用栈来判断）。
- 横插边（cross edge）：红色边，在搜索过程中访问过且不在栈中的结点的边。
- 前向边（forward edge）：蓝色边，搜索中遇到子树中的结点的边。

对于无向图，不存在横插边和前向边。

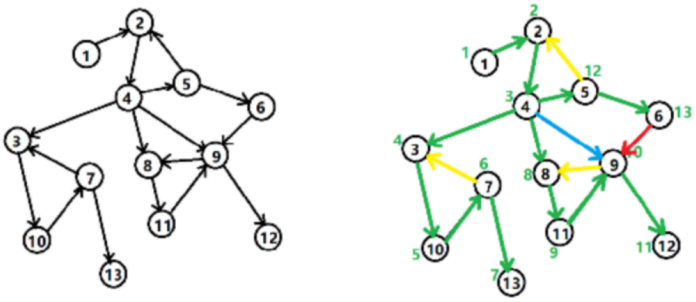

图 3.1: DFS 生成树图例

其实说实话这些东西知道了也没啥用。

3.2 割点

3.2.1 过程

不难想到可以尝试删除一个点，然后判断此图的连通性。但这样做复杂度会极高。应此我们需要用到 tarjan 算法。

设点 $x \in V$ 的子树为 x 在 DFS 树上的子树，包含 x 本身，记其为 $T(x)$。记 $\complement_V T(x) = T'(x)$。

对于非根节点的判定：

设 x 不为 DFS 生成树的根，则 $T'(x) \neq \varnothing$。

若 x 为割点，则删去后，对于 $z \in T'(x)$，存在 y 使 y, z 不连通，而删除 x 后，所有 $T'(x)$ 的边均存在，则 $y \in T(x)$。如果 y 与 z 不连通，因为 $T'(x)$ 的连通性不变，既均连通，则 y 与所有 $T'(x)$ 中的点都不连通。

推出：若存在 $y \in T(x)$ 在不经过 x 的情况下与 $T'(x)$ 中所有点均不连通，则 x 就是割点。

现在我们思考怎么判断对于一个 $y \in T(x)$，是否在不经过 x 的情况下与 $T'(x)$ 中所有点都不连通。

注意到，如果 $y \in T(x)$ 不经过 x 就和 $T'(x)$ 连通，就一定存在从 y 到 $T'(x)$ 的一条路径。我们称这条路径在 $T'(x)$ 的第一个结点为 v，倒数第二个，也就是最后一个在 $T(x)$ 上的结点为 u。如果 (u, v) 是树边，那么 $u = x$，矛盾。则 v 应该是 u 的祖先。又因为 x 是 u 的祖先，$v \in T'(x)$，则 v 也是 x 的祖先。

这时我们发现，x 不是割点，那么就会有一条**非树边**使 $y \in T(x)$ 连向 x 的祖先。这个可以用时间戳来判断。设 f_x 表示与 x 通过**非树边**相连的点的时间戳的最小值，则可写为 $f_x < dfn_x$。

因为 x 的不同儿子子树之间没有非树边（子树独立性），设 x 的儿子 y' 的子树包含 y，即 $y \in T(y')$。

- 对于 $T(y')$，如果存在 $u \in T(y')$ 满足 $f_u < dfn_u$，那么删除 x 后 $T(y')$ 的每个点与 $T'(x)$ 均连通。这时 u 与 $T'(x)$ 中的某个点通过非树边直接连通，$T(y')$ 中的所有点均通过树边连通。此时 x 不是割点。

- 反之，如果不存在 $u \in T(y')$ 满足 $f_u < dfn_u$，那么删除 x 后 $T(y')$ 的每个点与 $T'(x)$ 均不连通。此时 x 是割点。且只需要出现一个 y' 满足这个条件就可以判断出 x 是割点。

原本我们需要判断 $T(x)$ 上的所有点去检查 $f_u < dfn_u$ 的情况是否出现。现在令 low_x 表示与 x 及其子树上的所有点通过非树边相连接的点的时间戳的最小值，则可以只通过 low_x 判断即可。这样我们就得到了在 x 为非根节点情况下的割点判定方法：

对于一个非根节点 x，x 是割点当且仅当存在 y 是 x 的子节点，有 $low_y \geq dfn_x$。

对于根节点的判定：

根节点的判定相对简单。

设点 x 是 DFS 树上的根节点。

若 x 有大于一个子节点，根据子树独立性，删去 x 后其子节点的子树各不相通，所以 x 是割点。反之，若 x 有小于等于一个子节点，删去后剩余部分通过树边连通，x 不是割点。

这样我们就得到了判断割点的完整方法。

3.2.2 实现

解释

dfn：某个点被遍历到的时间戳。

low：某个点通过非树边到达最早的点的时间戳。

is_cut：是否为割点。

$cutcnt$：统计割点个数。

在 dfs 过程中求出 dfn 和 low，然后进行对割点的判断。

代码

此代码用 vector 存图。

```
vector<int> e[MAXN];
int dfn[MAXN];
int low[MAXN];
int cnt;
bool is_cut[MAXN];
int cutcnt;

void tarjan(int u,int f,int root)
{
    dfn[u] = low[u] = ++cnt;
    int son = 0; //用于统计子节点个数
```

```
12      for(int i = e[u].size()-1;i >= 0;i--)
13      {
14          int v = e[u][i];
15          if(v == f) continue;
16          if(!dfn[v])//说明这个 v 是在 x 的子树中的
17          {
18              son++;
19              tarjan(v,u,root);
20              low[u] = min(low[u],low[v]);
21              if(dfn[u] <= low[v])
22              {
23                  if(!is_cut[u] && u != root)
24                  {
25                      //非根节点的情况
26                      cutcnt++;
27                      is_cut[u] = true;
28                  }
29              }
30          }
31          else low[u] = min(low[u],dfn[v]);
32          //这个 else 中 v 不再 x 的子树当中。
33          //而我们求的是 x 子树当中 low 的最小值。
34          //因此 min 的第二个参数是 dfn[v] 而不是 low[v]。
35      }
36      if(u == root && son >= 2) is_cut[root] = true,cutcnt++;
37      //根节点的情况
38  }
```

3.3 割边

3.3.1 过程

和割点差不多。

树边能使整张图连通，因此割掉非树边后不影响连通性。因此 $e = (u, v)$ 是割边的必要条件是 e 为树边。

不妨设 v 是 u 的儿子。若割掉 e 后图不连通，根据求割点的经验，不难发现当 $low_v > dfn_u$ 时，e 为割边。

3.3.2 实现

还没写...

3.4 点双连通分量

3.4.1 实现

3.5 边双连通分量

3.5.1 实现

3.6 强连通分量

3.6.1 实现

解释

sta：栈。
top：栈顶。
insta：判断是否在栈中。
belong：判断点 i 在哪个强连通分量中。

代码

此代码用 vector 存图。

```
vector<int> e[MAXN];
int low[MAXN],dfn[MAXN],dfncnt,belong[MAXN],sc;
int sta[MAXN],top;bool insta[MAXN];
void tarjan(int u)
{
    low[u] = dfn[u] = ++dfncnt;
    sta[++top] = u,insta[u] = true;
    for(int i = e[u].size()-1;i >= 0;i--)
    {
        int v = e[u][i];
        if(!dfn[v])
        {
            tarjan(v);
            low[u] = min(low[u],low[v]);
        }
        else if(insta[v])
        {
            low[u] = min(low[u],low[v]);
        }
    }

    if(dfn[u] == low[u])
    {
        sc++;int now;
        do
```

```
26          {
27              now = sta[top];
28              insta[now] = false;
29              belong[now] = sc;
30              sz[sc]++;
31          } while (sta[top--]!=u);
32      }
33  }
```