算法笔记

一、STL sort

使用STL sort时，注意左闭右开！

当第一个元素为`a[1]`时

```
Len = unique(a + 1, a + n + 1) - a - 1//因为多算了第一个
Sort(a + 1, a + n + 1)
//查找元素值为x的下标：
int pos;
pos = lower_bound(a + 1, a + n + 1, x) - a
//注意与unique的异同！！
//因为lb返回的是指向元素的指针，而unique返回的是尾部后一位
```

eg:

```
int a[2] = {0, 1};//len = 1
pos = lower_bound(a + 1, a + n + 1, 1) - a  //Pos = 1
//而unique - a = 2 再 - 1才是len
```

二：STLsize

超超超！！就tm这玩意把我给整坏了！！

STL中的`size()`，是一个无符号整形变量`（unsigned int）`，在于其他数做运算时，会把其他数字强制转换为无符号整形。

而`unsigned int(-1)`的值是最大的`unsigned int`，即`4294967295`。
这也是为什么说`for (int i = 0; i < vec.size() - 1; i ++ )`这个写法是不被推荐的，
应该换成`for (int i = 0; i + 1 < vec.size(); i ++ )`
如果STL容器为空，那么`(unsigned int) 0 - 1 == 4294967294`，循环不仅不能跳出还会变成死循环。==（就这B，让我debug了一天）==

强制转换在处理防止溢出的时候很常用
比如二分的时候可以用`int mid = (l + r + 1ll) >> 1;(typedef long long ll;)`防止`l + r + 1`超过`int`的范围：
先把`l + r + 1`强制转换为`long long`，再赋值给`int`型的`mid`。
再例`int a = b + c - d + 0ll`这种操作就是为了避免`b+c`的结果超过`int`范围，所以通过`+0ll`先把`b, c, d`都转换为`long long`，再做减法计算。
不过用`printf`中`%d`占位符就不会出现这种情况

三：并查集查询连通集合数量

在并查集中，查找共有几个连通集合数量：
我先想出来的办法：

```
int ans = unique(myset.fa, myset.fa + n) - myset.fa;
```

为错误写法，且纯shabee（路径压缩没彻底怎么办？）
正确方法：

```
int ans = 0;
for(int i = 0; i < n; i++)
    if(myset.fa[i] == i) ans++;
```

对并查集更深的理解：

1. 我们发现 `fa[i] == i` 是并查集的命根子
   在 `find`（递归）函数里 `if(fa[i] == i) return i;` 是边界条件，这说明无论怎样查询，只有查到 `fa[i] == i` 才罢休，以前不熟悉时，受记忆化搜索影响，想能不能直接 `return fa[i]`，发现是错误的。这也说明，写并查集，要以改条件为本，才不会受各种奇奇怪怪的情况影响，`fa[i] == i` 代表一个子集必然正确。
2. 在 `union` 函数中，`fa[find(x)] = find(y)` 核心在 `find` 上，由1得 `find（i）` 中 `fa[i] == i` 必然成立，就导致合并时不会导致以前建立的关系破碎：即不能改自己的，必须改族长的。
   （我语文烂，写的什么乱七八糟的，反正就差不多这个意思）

> 2022.12.29批注: 我以前写的什么laji

四：笔记时间（2022.7.2）spfa与 优先对列优化的dij区别

注：队列中存的是点，不是边！！！！！
两者看形式实在是太像了

1. Dij：
    1. dis[s]归零，s入队
    2. 进去循环
        1. 取出队首（保证dis最小）
        2. 对队首连接的进行松弛
        3. 将成功松弛的点入队
    3. 输出答案
2. Spfa：
    1. dis[s]归零，入队
    2. 循环
        1. 取出队首（不保证最小）
        2. 松弛
        3. 成功松弛的入队
    3. 输出

可以看出，spfa就是有限制的层序遍历，也就是广度优先搜索
而dijkstra便是启发式搜索

但有一点要注意
`vis` 数组在dij中意味有没有确定（即答案定死，最短路就是现存的这个）
在 `spfa` 中 `vis` 意味着是否在队列中
所以 `spfa` 在取出队首时要归零 `vis`，而 `dij` 不需要
`Spfa` 之所以有 `vis` 区别，正是因为取出队首时没有取出最大值，而需要让点多次入队以保证正确性
并且，在 `vis` 对入队的影响上也有差别，这里给出两段代码

1. 代码1

```
void dijkstra_wrong() {
    //first = dist, second = idx
    priority_queue < pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>> > q;
    vector<int> dis(n + 1, INT_MAX);
```

```
    vector<bool> vis(n + 1);
    dis[s] = 0;
    q.push(make_pair(0, s));
    while (!q.empty()) {
        int now_idx = q.top().second;
        q.pop();
        for (edge ed : g[now_idx]) {
            if (dis[ed.nxt] > (long long)dis[now_idx] + ed.val) {
                dis[ed.nxt] = dis[now_idx] + ed.val;
                if (!vis[ed.nxt]) {
                    q.push(make_pair(dis[ed.nxt], ed.nxt));
                    vis[ed.nxt] = 1;//将入队策略和spfa完全同步，唯一区别就是出队时不取出了，但是wa了
                }//原因是因为，ed.nxt在后序入队时，其dis值可能已经改变，这会导致优先队列中优先级异常
                //而优先级是dij算法的核心，优先级寄了，算法也就寄了
            }
        }
    }
}
```

2. 代码2

```
    void dijkstra() {
    //first = dist, second = idx
    priority_queue < pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>> > q;
    vector<int> dis(n + 1, INT_MAX);
    vector<bool> vis(n + 1);
    dis[s] = 0;
    q.push(make_pair(0, s));
    while (!q.empty()) {
        int now_idx = q.top().second;
        q.pop();
        if (vis[now_idx])//存在入队重复，当一个更优松弛已经结束，应跳过本次
            continue;
        vis[now_idx] = 1;
        for (edge ed : g[now_idx]) {
            if (dis[ed.nxt] > (long long)dis[now_idx] + ed.val) {
                dis[ed.nxt] = dis[now_idx] + ed.val;
                if (!vis[ed.nxt])//不需要这里判断也能过，但少入队总是好的
                    q.push(make_pair(dis[ed.nxt], ed.nxt));//这里怎么能这么写？
                else
                    cnt++;
                //2022.12.29批注，完全不会导致少入队，没有必要这么写
                //这条边到达的点不可能有vis过
                //考虑vis的意义，是完全确定下来的点
                //而这里松弛成功，所以不可能vis过
            }
        }
    }
}
```

即dij算法有必要重复入队，但在确定dis值不变后，重复入队的数据可以跳过

> 2022.12.29批注 重复入队可能的，但绝对不会重复出队（后来的出队后直接continue）

五：dijkstra核心（2022.8.5）

Dijkstra的核心便在优先队列中的优先级，未优化过的dij以线性查找最小值本质上也是查找优先级最大的那个，都一样的，未优化dij退出循环的条件是now = 0，即在队列中找不到vis == 0 的了，相当于队列为空。

优先级是启发式搜索的特征，而基于优先级，dij能保证后出队的元素一定是在已出队的确定的答案上进行推演，这一点又有点像动态规划，所以dij其实是超级缝合怪?

而dij致命缺陷就是因为优先级原理在负权上寄了，所以dij也寄了，就这。

六：对于各种最短路算法之间联系的理解

都说队列优化的dij是普通dij的优化方案
Spfa是bellman_ford的队列优化方案

其实我觉得，bellman_ford是floyd的优化方案（后来思考表明，这么想并不严谨）
队列优化的dij是spfa的优化方案
而普通dij则是对队列dij的 负优化/在稠密图中的优化

下面解释原因：

Floyd的3重循环其实可以看做1 + 1：
（这也是为什么中转点枚举for一定要在最外层的原因）
伪代码：
For 枚举中转点 第一层
双重for 枚举两点组成的边 第二层
进行松弛

而bellman_ford算法是这样的：
伪代码：
For 枚举层数
For根据邻接表或边的集合枚举所有边
可以看出 bellman_ford也就只是让第二层循环中不会出现无效，不存在的边。
然而这只是bellman比floyd快的原因，但原理并不同。

Bell在松弛时是对该边松弛，即`dis[ed.to] = dis[ed.from] + ed.val;`
而floyd 是基于中转点松弛，即`g[i][j] = g[i][k] + g[k][j];`
出现这样原因是因为对dis数组的定义，bell一维，floyd二维。
Bell第一层循环的意义并非枚举中转点，而是：最多经过k条边后，可以达到的最短路径
这点类似层序遍历，不细细展开了。
基于这个性质，bell也经常用于“火车站问题”即“最多经过k站，到达目的地的最短路程是多少”的问题。

然而floyd能解决多源最短路问题，就是因为它枚举了无效边，事实上，在枚举无效边的时候，floyd自动建立了“虚边”，到最后，只要图是连通的，floyd就能让任意两个点都有边相连，即建立起一个“虚完全图”。这是我的理解。

至与spfa和dij的关系，上面讲了很多了，这里不再细讲了
Dij在spfa的队列基础上加了优先级这个条件

而根据大流说的spfa是bellman的队列优化，那么其实所有最短路都是一家？？
原来floyd和dijkstra是同源的吗？？笑死，难以想象，这边给出思维导图

祖宗：
Floyd
跳过无效边：
bellman_ford
进行队列优化：（这里有些牵强，我觉得不是基于bellman改进，而是建立了一个新算法，而弥补了bellman的缺陷罢了）
Spfa
进行优先级优化，以消除重复松弛：
队列Dijkstra
进行在稠密图中对寻找最大优先级的点的方案的优化：
普通dijkstra

这边额外说说spfa到底对bellman改进了啥：
虽说bellman跳过了无效边，但还只是暴力枚举了
Bellman-Ford算法在每一次实施松弛操作时，就会有一些顶点已经求得最短路径，此后这些顶点的最短路径的估计值就会一直保持不变，不再受后续松弛操作的影响，但是每次还要判断是否需要松弛，这里浪费了大量的时间.
Spfa让成功松弛的元素才有入队的资格，弥补了这一点不足
即（你自己还是INT_MAX呢，你也配来松弛我？）和（你自己还没松弛完呢，也配来松弛已经是最短的我？）
就这。

> 2022.12.29批注 我™写的啥？

位运算优先级 2022.10.19

血的教训，在做一道A*题目（P5507， 我题号都背下来了）的时候，一直查不出错在哪里，结果，在殚精竭虑两天后，发现居然是位运算优先级的问题
！！！下次算一次位运算就加一次括号！！！
>> 大于 == 大于 &
真tm服气了，我一直以为位移运算符的优先级是最低的。。。

最长上升子序列

先简单讲下怎么O(nlogn)搞一个最长上升子序列吧。
考虑一个数列`5 2 3 1 4`
首先，把5加入答案序列中，然后加2，发现2<5所以显然2替换5不会使结果更差，
那么答案序列就是{2},然后加3，发现3>2，所以直接把3加到答案序列中：{2,3}
然后加1，我们发现1<3，于是我们找到一个最小的但是比1大的数字2，然后把1替换2，**为什么这么做不会影响结果呢？你可以这么想，我们当前已经求出了一个当前最优的序列，如果我们用1替换2，然后后面来一个数字替换了3，那么我们就可以得到一个更优的序列，而如果没有数字替换3，那么这个1替换2也就是没有贡献的，不会影响我们结果的最优性。**至于，如何找到一个最小的但是大于某个数字的数字，弄个二分查找就行了，因为我们的答案序列是有序的呀

负数右移运算

负数右移运算高位补1！！！
这样子右移才能继续保持 对于二的幂次方的除法 的正确性

CSDN【C/C++】负数的右移运算

关于线段树

乘法标记下放

摘抄

第一种：先加再乘

此时该节点为 $(val + add_1) \times mul_1$ 当再遇到一个 $[mul_2, add_2]$ 的标记时，此时节点为 $[(val + add_1) \times mul_1 + add_2] \times mul_2$

把 mul_1 提出来，把原式重新化成 $(val + add') \times mul'$ 的形式，得 $(val + add_1 + \frac{add_2}{mul_1}) \times mul_2 \times mul_1$

我们发现这里有个除法，会损失很多精度，因此我们换一个思路

第二种：先乘再加

此时该节点为

$(val \times mul_1) + add_1$

当再遇到一个

$[mul_2, add_2]$

的标记时，此时节点为

$[(val \times mul_1) + add_1] \times mul_2 + add_2$

把式子展开并重新化为 $(val + add') \times mul'$ 的形式，得

$val \times mul_1 \times mul_2 + add_1 \times mul_1 + add_2$

我们发现这样不需要除法，因此我们选用第二种

区间修改事项

```
    void add(unsigned idx, unsigned l, unsigned r, unsigned tar_l, unsigned
tar_r, T val) {
        if(out_of_range()) {
            return;
        }
        if(in_range()) {
            make_add(idx, l, r, val);
            return;
        }
        auto mid = l + (r - l) / 2;
        //push_down(idx, l, r);//necessary
        add(idx << 1, l, mid, tar_l, tar_r, val);
        add(idx << 1 | 1, mid + 1, r, tar_l, tar_r, val);
        push_down(idx, l, r);//It's all right to put push_down here or there
        pull_up(idx);//anyway, we should ensure the validity of the child_data
before we execute pull_up
    }
```

关于DFS

> 我们常使用的不带返回值的dfs，void dfs(...)，一般是从起点搜索到终点，就像走迷宫一样，从本状态列出下一步可走的状态，然后dfs下一步可走的状态，一直达到终点。这类dfs属于**递推**(从小（边界条件）往大搜索)，比如二叉树的先根序序遍历（void preorder），先遍历根结点，再往左子树，右子树递推着遍历。
>
> 但是有时候，我们发现将大问题分解为小问题直至问题边界更利于代码的书写，这类就属于**递归**，比如求斐波那契数列，f[n]=f[n-1]+f[n-2]，求f[n]的问题就被转换为求f[n-1]和f[n-2]，这里可以使用**dp或者带返回值的dfs**。
>
> ——dfs终极总结_dfs带会返回值与不带会返回值的写法-CSDN博客

三种二分查找的解法

例题：P2249 【深基13.例1】查找 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

二分查找的前提是数组中的数据有单调性，且有一个关于此单调性的条件将数组分为两段，二分查找的本质就是去找这两段的分界线，而数组并不是一个数轴，所以这个分界线存在满足条件的末端和不满足条件的起点这两种表示方法。

所以二分查找可以通过查找这两个点有多种方法来实现。

注意点

1. 指针的意义

   尽量不要把`l`和`r`理解为区间，这样定义不完备，可以将其当做指针

2. 跳出循环的条件

3. 指针的初始位置

   注意满足数组全符合条件和全不符合条件的情况

4. 取mid时候取整方式

   > 当寻找性质的左边界的时候，由于更新区间为`r = mid, l = mid + 1`，此时`mid = (l + r ) / 2`，不需要加一。
   >
   > 为什么?
   > 因为不加一的时候`mid`是下取整
   > 更新`r`时直接赋值为`mid`的也会下取整，因此r一定会收缩，不会死循环。
   > 更新`l`时是赋值为`mid + 1`会上取整，因此l也一定会收缩，所以`mid`不需要加一。
   >
   > 当寻找性质的右边界的时候，由于更新区间为`l = mid`，`r = mid - 1`，此时`mid = (l + r + 1) / 2`，需要加一。
   >
   > 为什么?
   > 因为加一的时候`mid`是上取整
   > 更新l时直接赋值为`mid`也会上取整，因此l一定会收缩，不会死循环。
   > 更新r时赋值为`mid - 1`会抵消掉上取整，因此r也一定会收缩。
   >
   > 【算法总结】二分查找及边界问题_二分查找的边界问题-CSDN博客

实现

注：以下vector默认从1开始作为下标

1. 左指针指向满足条件的区域，右指针指向不满足的区域

```
int binary_search(int destination, const std::vector<int> &vec) {
    int l = 0, r = static_cast<int>(vec.size());
    //l = 0位于数组左端点的左方
    //r = n + 1位于末端的右方
    //l和r均处于数组外
    //为什么？考虑数组全符合条件和全不符合条件的情况
    //全符合条件，结束时：l = n r = n + 1
    //全不符合条件，结束时
    while(l + 1 < r) { // l和r相邻，退出循环
        int mid = (l + r) / 2;
        if(vec[mid] < destination) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return r;
}
```

2. 期望l，r都指向满足条件的最后一个

所以跳出循环的条件是l == r

```
int binary_search(int destination, const std::vector<int> &vec) {
    int l = 0, r = static_cast<int>(vec.size()) - 1;
    // 全都不满足条件，归0，区间外
    //全都满足就是最后一个，区间内
    while(l < r) {
        int mid = (l + r + 1) / 2;//attention
        if(vec[mid] < destination) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
    return r;
}
```

3.期望l，r都指向不满足条件的第一个

```
int binary_search(int destination, const std::vector<int> &vec) {
    int l = 1, r = static_cast<int>(vec.size());
    // 全都不满足条件，归1，区间内
    //全都满足就是最后一个 + 1，区间外
    while(l < r) {
        int mid = (l + r) / 2;//attention
        if(vec[mid] < destination) {
            l = mid + 1;
```

```
        } else {
            r = mid;
        }
    }
    return r;
}
```

其中，2,3返回的时候l == r 所以返回谁都行

参考资料

1. （不建议）二分查找算法细节与查找左右侧边界_二分查找边界-CSDN博客
2. （不建议？）【洛谷日报#13】浅谈二分的边界问题 - 知乎 (zhihu.com)
3. （没看过，有递归）【数据结构与算法】一篇文章彻底搞懂二分查找（思路图解+代码优化）两种实现方式，递归与非递归-阿里云开发者社区 (aliyun.com)

再探Dijkstra

参考资料

1. Negative weights using Dijkstra's Algorithm - Stack Overflow
2. (11 封私信 / 81 条消息) Dijkstra's 最短路径算法能不能解这个含有负权重的问题？ - 知乎 (zhihu.com)

SPFA和Dijkstra区别

十分相似，区别主要在vis数组的处理上

Dijkstra对vis的处理

1. 从优先队列取出后，验证标记，若已确定，则跳过（同一个点可能重复入队）
2. 从优先队列取出后，打标记，表示给点已经确定
3. for循环中，对nxt点验证

SPFA对vis的处理

为什么Dijkstra不能解决负边权?

单调栈

单调栈解决什么问题?

单调栈一般用于解决"NGE"问题，即Next Greater/Less Element，某个数左侧或右侧第一个比它小或大的数

编写单调栈要考虑什么问题?

1. 扫描方向，取决于要寻找的数在左侧或右侧，比如右侧第一个更大的数，考虑单调栈维护的性质应与寻找的数方向相同，所以应该从右向左扫描
2. 何时出栈、入栈，在出栈之前必须处理完之前所有的数，所以应该先出栈，然后再入栈
3. 何时维护数据信息？可以选择在入栈的时候维护

一些细节问题

单调栈内存储的是元素的下标，这样能维护的信息更多

样例代码

```cpp
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n;
    std::cin >> n;
    std::vector<int> stk, vec(n + 1), ans(n + 1);
    for(int i = 1, x; i <= n; ++i) {
        std::cin >> vec[i];
    }
    for(int i = n; i > 0; i--) {
        while(!stk.empty() && vec[stk.back()] <= vec[i]) {
            stk.pop_back();
        }
        ans[i] = stk.empty() ? 0 : stk.back();
        stk.push_back(i);
    }
    for(int i = 1; i <= n; ++i) {
        std::cout << ans[i] << ' ';
    }
}
```

P5788 【模板】单调栈 - 洛谷 | 计算机科学教育新生态

P2866 [USACO06NOV] Bad Hair Day S - 洛谷 | 计算机科学教育新生态

关于SCC

关于Kosaraju

关于遍历方式

对于二叉树来说，有三种遍历方式，前序遍历，中序遍历，后续遍历

但对于多叉树来说，中序遍历（先递归左，标记root，再递归右）是没有意义的，而图的遍历相当于遍历图的dfs生成树，所以也是没有意义的

我们一般说的dfn是指图的先序遍历编号，而在kosaraju中，节点的值是后续遍历编号（在dfs型拓扑排序也是这样的）

关于拓扑排序

继续理解kosaraju的dfs，我们必须先理解dfs求DAG的拓扑序是如何完成的，我们认为，**节点退出dfs的顺序（即后续遍历顺序）的倒置即是DAG的拓扑序**

证明：

前提:

1. 拓扑序靠后意味着节点的前置节点多，能够访问到的节点数目越少，拓扑序靠前意味着节点的前置节点少，能够访问到的节点多
2. dfs的方式是 for(int i = 1; i <= n; i++) { if (!order[i]) dfs(i) }

那么，只要证明，最后退出dfs的节点就是拓扑序最靠前的节点（其他节点可以类似Kahn算法，递归地证明）

假设

1. 节点A最后退出dfs
2. 节点B拓扑序大于A。

B拓扑序先于A表示A无法访问B，但是B可以访问A

将所有情况划分成两种：

1. 在for循环中，我们先访问了A，再访问了B
    1. A如果能访问到B，那么A的拓扑序先于B，矛盾
    2. A如果不能访问到B，那么for循环一定后访问B，B一定更后退出dfs，矛盾
2. 在for循环中，我们先访问了B，再访问了A
    1. B能访问A，则B更晚退出dfs，矛盾
    2. B不能访问A，则A拓扑序更大，矛盾

所以，不存在这样的B

在一般图中，这样dfs的意义是什么？（！！！！！！！）

1. **推论：如果有向图单向联通（存在某点可以访问所有节点），最后退出dfs的节点一定可以访问到所有节点**（因为这个点是拓扑序最大的SCC中的点）
2. **在DAG中，这个顺序（最后退出的节点位于第一位）就是拓扑序，而在非DAG中，这个序列相当于各个SCC的拓扑序**（不严格是这样，只有某SCC第一次出现的点能够代表SCC的拓扑序，其他点应该忽略，就是说，此“伪拓扑序”是有顺序限制的，只能保证正着读有正确性）
3. **所以，如果要找缩点后DAG的拓扑序，只需要在原图上跑dfs拓扑排序就可以了**

关于反图的含义

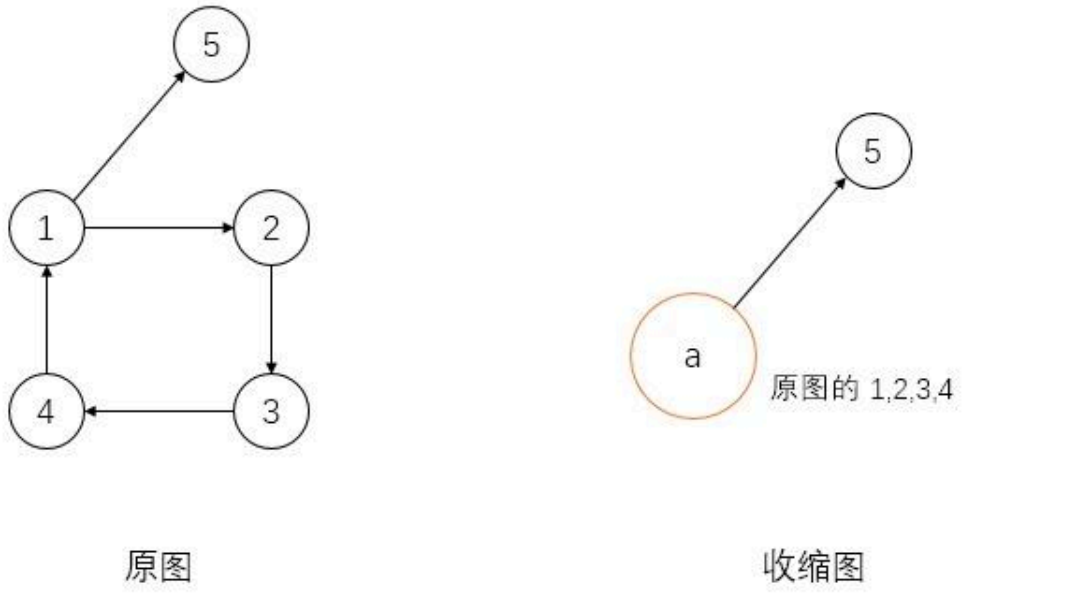
如何深入浅出的理解 Kosaraju-CSDN博客

我们知道，SCC中任意两点都可以互相访问，所以，建立反图以后，SCC还是SCC

我们在正向图中，已经找到了SCC的拓扑序，建立反图后，**拓扑序倒置**我们从原来拓扑序靠前的SCC往后访问，相当于从反图中拓扑序最最后的SCC访问，而拓扑序最后的SCC只能访问自己！所以反图dfs2能访问到（并且!vis[i]，考虑拓扑序倒数第二的SCC），即在一个SCC中

既然这样，我们为什么不在原图第二次dfs

> https://www.zhihu.com/question/50437680/answer/487457165
>
> 问这个问题说明你对topological sort还没有真正理解。
>
> 当compute一个DAG的topological ordering的时候有一个property就是topo ordering最后那个vertex肯定是原graph的sink vertex。
>
> 但是在一个cyclic directed graph上应用topo sort时这个property是不存在的。这时topo order最后那个位置上的vertex有可能在一个sink SCC中，也有可能不在，取决于开始dfs的那个initial vertex。
>
> 庆幸的是虽然在cyclic directed graph上topo sort这个property不存在，我们还有另外的一个property可以用，即这时topo sort第一个位置上的vertex是肯定在source SCC （只有出度，入读为0）里的。有了这个确定的property我们就可以通过在reverse graph上用原图topo order逆序进行DFS就可以了。
>
> 所以理解这个算法的关键就是要理解topological sort应用在DAG和cyclic directed graph有不同的property。



> dfs 序列为 4,3,2,5,1，其中 4,3,2,1 属于收缩图中的同一结点 (a)，它们中间插了一个 5，所以从前往后遍历和从后往前**并不是恰好相反**的，对应到收缩图里两个都会先走 a 结点。
>
> 图取转置之后算法保证了 a 是 sink，先走 a 结点能保证不会到5，但不取转置显然会在第一次遍历的时候外流到 5 这个另一连通分支。
>
> 如果你能用**额外信息**保证收缩图同一结点内原结点的排列是连续的就可以直接后序，比如你 dfs 的时候每次都保证先遍历属于其它连通分支的儿子。
>
> 总之这个算法的 point 在每个强连通分支内第一个遇到的结点（所谓 **leader**，这里是 1），而不在于整一个连通分支（即收缩结点）。
>
> 链接：https://www.zhihu.com/question/265266923/answer/912239192

就是说，我们的想法是在原图中先访问5，在访问a，可是，伪拓扑序的倒置4,3,2,5,1中（4,3,2）不能代表a，所以顺序会发生错误（上面讲过这个伪拓扑序是有顺序的，所以不能倒置）

如果我们建立反图，我们就要用伪拓扑序倒置（反图）的倒置，就是伪拓扑序本身，这样是可行的