



FOI2023 算法夏令营B班 第二讲

南方科技大学 匡亮



今天的目标

- 栈、队列、单调栈、单调队列
- 链表、哈希表、ST表
- 并查集
- 其他相关的数据结构或算法或技巧等等
- 提前讲一点明天的内容（明天的讲课人同意了）



数据结构

- 什么是数据结构？
- 数据结构是计算机存储、组织数据的方式。
- 数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。
- 通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。
- 数据结构往往同高效的检索算法和索引技术有关。
- （百度百科）



数据结构

- 算法竞赛中的数据结构和计算机科学中的数据结构并不完全是一个概念
- 今天讲座中的内容当然会以算法竞赛为主，但同时也会涉及到一些计算机科学的观念，为的是帮大家做到对原理有更深入的理解



大家猜猜

- 如果现在要存储若干多项式（次数都是非负整数），最适合用哪种数据结构？



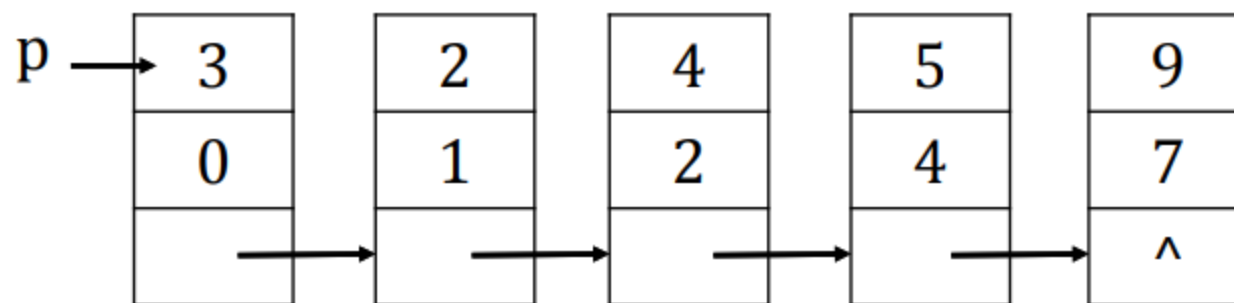
链表

- 什么链表？ 我会数组！
- 下标代表次数.....如果只有两项， 但是次数一百万呢
- 那就下标代表项数.....如果要对两个多项式做加减法呢

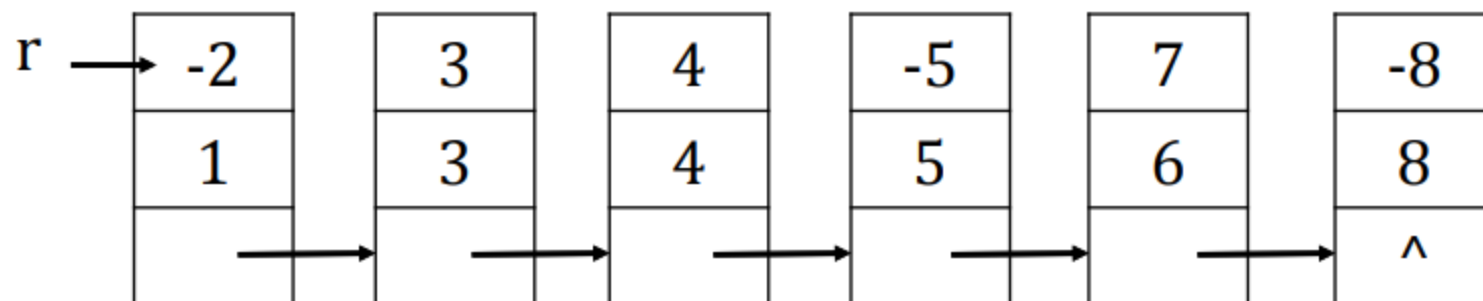


Adding two polynomials

◆ $p(x) = 3 + 2x + 4x^2 + 5x^4 + 9x^7$



◆ $r(x) = -2x + 3x^3 + 5x^4 - 5x^5 + 7x^6 - 8x^8$

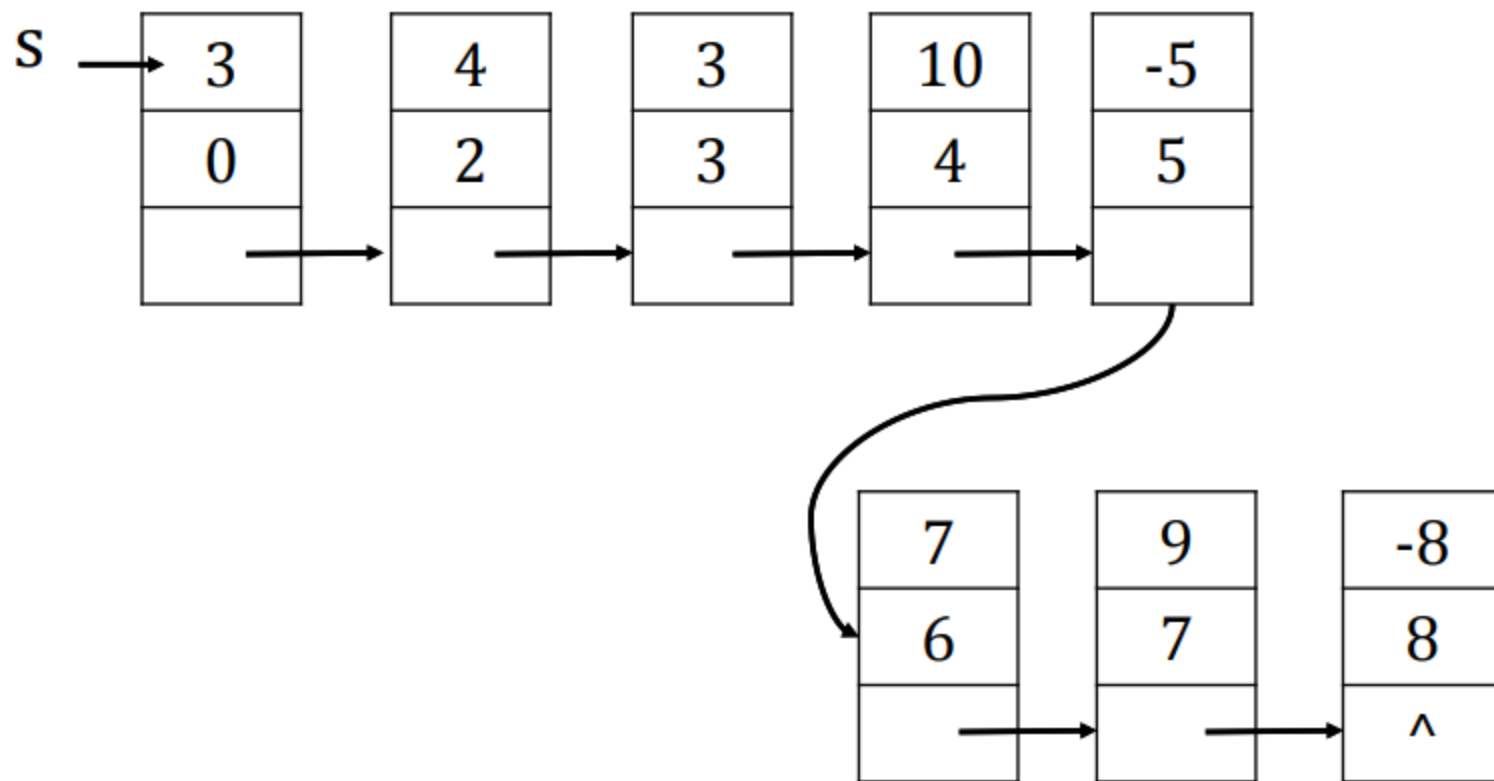


Author: 唐博



Adding two polynomials

◆ $s(x) = p(x) + r(x)$
 $= 3 + 4x^2 + 3x^3 + 10x^4 - 5x^5 + 7x^6 + 9x^7 - 8x^8$



Author: 唐博



链表

- 我们常用一个结构体来实现链表，用两个指针来存前后的节点，一个（或多个）int（或其他类型）来保存数据
- 实践中如果用指针，可以在执行任何操作以前无脑赋值为NULL，然后只以NULL作为不合法指针（空指针）
- 推荐阅读：
- 如果指针操作不熟练，在竞赛中用数组替代也没有问题

```
4 struct Node {  
5     Node *next, *last;  
6     Node(): next(NULL), last(NULL) {}  
7     int val;  
8 };
```



链表在操作系统中的运用

- 操作系统uCore是用于清华大学计算机系本科操作系统课程的教学试验内容
- 在进程管理、内存分页等很多地方都需要用链表来管理一个支持元素增删的集合
- 然而对每个不同的对象都定义一个next和last指针也太冗余了。设计操作系统的大神们是怎么做的呢？



代码片段

[ucore](#) / [labcodes](#) / [lab8](#) / [libs](#) / [list.h](#)

Code

Blame

163 lines (142 loc) · 4.52 KB

```
17 struct list_entry {
18     struct list_entry *prev, *next;
19 };
20
21 typedef struct list_entry list_entry_t;
```

[ucore](#) / [labcodes](#) / [lab8](#) / [kern](#) / [schedule](#) / [sched.h](#)

Code

Blame

73 lines (61 loc) · 2.37 KB

```
18 #define le2timer(le, member) \
19     to_struct((le), timer_t, member)
```

← Files

master

[ucore](#) / [labcodes](#) / [lab8](#) / [libs](#) / [defs.h](#)

Code

Blame

Raw



```
65 /* Return the offset of 'member' relative to the beginning of a struct type */
66 #define offsetof(type, member) \
67     ((size_t)((type *)0->member))
68
69 /* *
70  * to_struct - get the struct from a ptr
71  * @ptr:      a struct pointer of member
72  * @type:     the type of the struct this is embedded in
73  * @member:   the name of the member within the struct
74  * */
75 #define to_struct(ptr, type, member) \
76     ((type *)((char *)(ptr) - offsetof(type, member)))
```



代码片段

ucore / labcodes / lab8 / kern / schedule / sched.h

Code

Blame

73 lines (61 loc) · 2.37 KB

```
12  typedef struct {
13      unsigned int expires;    //the expire time
14      struct proc_struct *proc; //the proc wait in this timer.
15      list_entry_t timer_link; //the timer list
16  } timer_t;
```

ucore / labcodes / lab8 / kern / schedule / sched.c

Code

Blame

175 lines (158 loc) · 4.32 KB

```
144  // call scheduler to update tick related info, and check
145  void
146  run_timer_list(void) {
147      bool intr_flag;
148      local_intr_save(intr_flag);
149      {
150          list_entry_t *le = list_next(&timer_list);
151          if (le != &timer_list) {
152              timer_t *timer = le2timer(le, timer_link);
```



链表

- 访问复杂度： $O(N)$
- 删除复杂度： $O(1)$ （如果你知道要删哪个）
- 但是.....往往我们都需要找到了再删，这下删也变成了 $O(N)$ ，从竞赛来看不就废了
- 这时就需要一个常见的思路来辅助一下我们：把 n 个 $O(n)$ 一起做，试着让它保持 $O(n)$



数据范围

T2: 我遇到了更好的

对于所有数据, $N \leq 10^6, a_i \leq 10^9$

试题英文名: better

出题人的好兄弟终于还是和他的女朋友分手了。嘴硬如他, 坚称他遇到了更好的。

其实大家都知道的, 分手以后, 别说更好的了, 能遇到差不多的都不错了。

好兄弟的一生中一共会和 N 个女孩邂逅。虽然不建议大家这样做, 但他给每个女孩在心里打了一个正整数分数 a_i 。他想知道, 自己的每次分手后, 下一次遇到和这个女孩最像的女孩 (指分数差距最小) 时, 这个分数差距是多少。

即, 对于 $1 \leq i \leq N - 1$, 求出 $\min_{i+1 \leq j \leq N} |a_j - a_i|$ 。



例题

- 题意：对于每个数，找到在它之后出现的差距最小的数
- 简单的思路：颠倒加入的顺序，变成在它之前出现的差距最小的数，set即可
- 更好的思路：
 - 排序，按照大小顺序，将 <数值大小，在数组中的位置> 建成链表
 - 访问一遍链表，记录下数组中的每个位置对应的链表节点在哪里（这一步将n次访问链表的时间压成了 $O(n)$ ）
 - 按从后往前的顺序，访问这个链表上的节点以及它左右的节点，得到最小的差值，然后删除这个节点
 - 时间复杂度： $O(n+va)$ （基数排序）

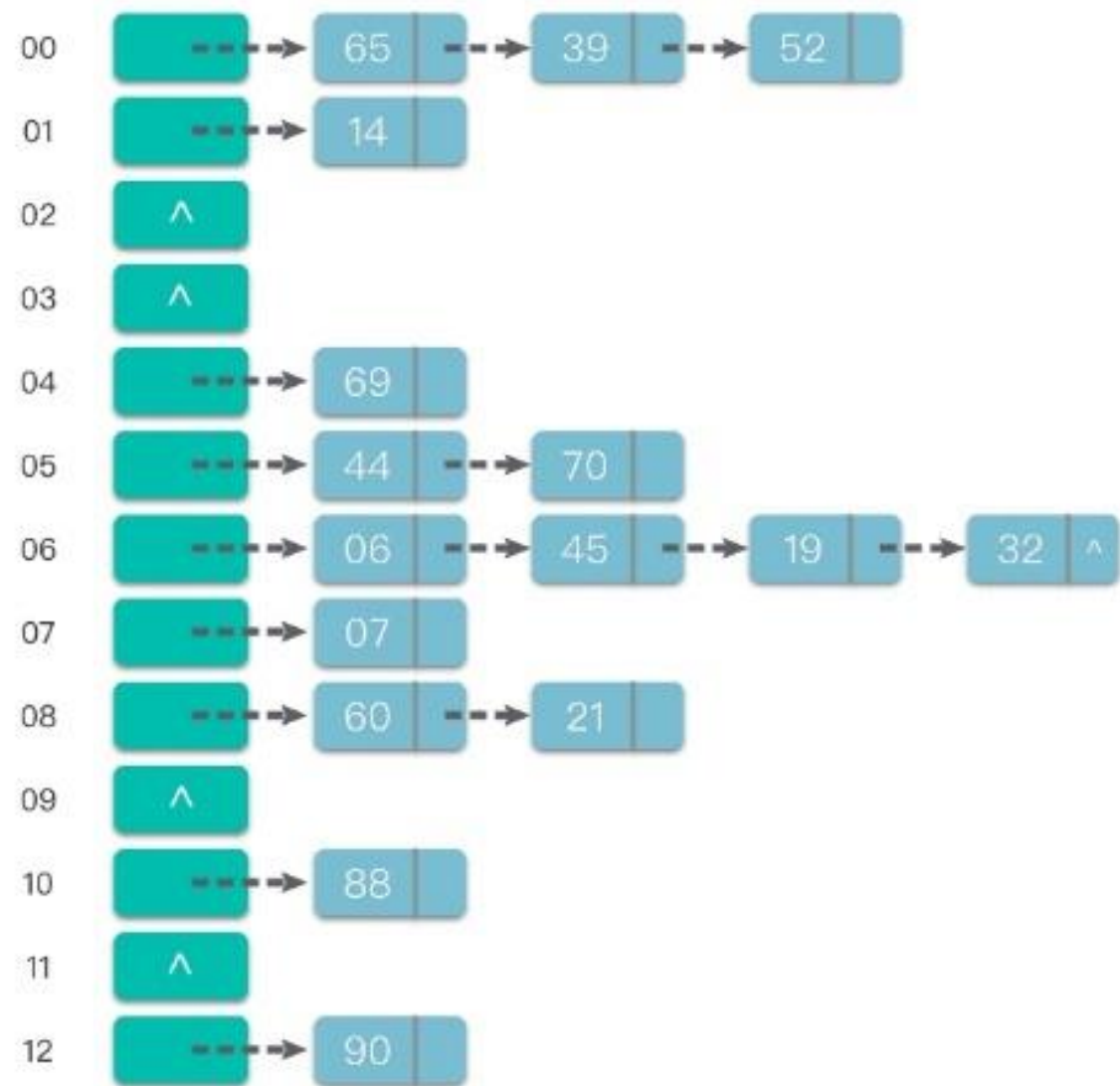


哈希表

- 给你一些数，再多次问你某个数有没有出现过怎么做呢？
- 除了平衡树、字典树，还有一种综合表现优秀的做法：哈希表
- 假设这些数都是均匀随机的，我们有理由相信它们模一个大质数 P 以后应该还是均匀随机的，相同的不会太多
- 我们建 P 个链表，用来存模的结果相同的元素，查询的时候就可以查询期望少量个元素
- 好处在于，如果哈希函数优秀，复杂度低，代码简单，在线，可删除
- 坏处在于.....



• 上一页实在放不下了



ST表

- 刚刚我们学习的数据结构，链表，相对来说比较泛用
- 接下来我们学习一种专门针对（静态）**可重复贡献问题**的数据结构：ST表（sparse table，又称稀疏表）
- 那么首先，什么是可重复贡献问题呢？
 - （以下几页均参考 OI wiki）



可重复贡献问题

- 当一种运算 opt 满足 ①结合律 ② $x \text{ opt } x = x$ 时，它的区间询问就是一个可重复贡献问题
- 结合上面的①和②，我们可以发现
- $a \text{ opt } x \text{ opt } x \text{ opt } x \dots = a \text{ opt } (x \text{ opt } x \dots) = a \text{ opt } x$
- 简单来说，即对一个元素进行多次运算不会影响我们的结果
- 类似地还有 $a \text{ opt } b \text{ opt } c \text{ opt } b \text{ opt } c = a \text{ opt } b \text{ opt } c$ （如何证明？）
- 求gcd、求按位或的和、求按位与的和等，都是常见的可重复贡献问题



RMQ 问题

- **RMQ问题**，即Range Maximum/Minimum Query问题，即区间最大（最小）值问题，简称**区间极值问题**
- 显然，**RMQ问题**是一个**可重复贡献问题**（准确地说应该是求极值是一个可重复贡献问题）
- 另一方面来说，其实很多**可重复贡献问题**都是一个潜在的**RMQ问题**
- 比如求区间gcd，本质上是在求区间里各个质因子次数的最小值。



ST表

- 理解了以上两个概念以后，我们来学习解决这种问题的一种数据结构：ST表
- 我们首先来解决一个最简单的RMQ问题：给定一个长度为 n 的序列，然后 m 次询问区间最大值
- 本题（以及之后任何没有特殊提及数据范围的题）默认 n 和 m 都是 10^5 级别，即接受 $n\log n$, $n\log^2 n$, $n\sqrt{n}$ 等级别的做法



ST表

- 做法1, 我会暴力! 每次询问暴力访问整个区间算出最大值, 时间复杂度 $O(mn)$
- 解决简单的数据结构问题时的一个常见思路: 看看我们重复做了哪些事情?
- 如果两次询问的区间分别是 $[l,r]$ $[l-1,r+1]$, 那么暴力访问 $[l,r]$ 这件事就被做了两次



ST表

- 于是，做法2，我会DP！ $f[i][j]$ 表示区间 $[i,j]$ 的极值，用方程 $f[i][j]=\max(f[i][j-1], a[j])$ 来dp即可
- 时间复杂度 $O(n^2+m)$ ，好但还不够
- 现在我们没有做重复的事，但我们做了很多多余的事：我们真的需要那么多区间的答案吗
- 注意到这是一个可重复贡献问题，我们似乎更应该取一些具有代表性的区间，然后每次询问拼出一个答案



ST表

- 于是，做法3：ST表！
- 我们用 $f[i][j]$ 表示从 i 开始 2^j 个元素的最大值，那么 j 的取值范围就只有0到 $\log n$
- 并且 f 数组也很好求： $f[i][j] = \max(f[i][j-1], f[i+2^{j-1}, j-1])$
- 有了 f 数组后，询问 $[l, r]$ 的最大值时，令 $s = \log(r-l+1)$ 下取整，则 $\text{ans} = \max(f[l, s], f[r-2^s+1, s])$
- 这样我们就得到了一个 $O(n \log n + m)$ 的做法



快问快答

- 现在我们已经学会ST表啦，可惜它是一个静态数据结构。那么我们如何解决下面这题呢？
- 给定一个初始长度为 n 的序列， m 次询问或操作：询问 $[l,r]$ 的最大值，或者在当前序列的末尾插入一个数，强制在线



快问快答

- 现在我们已经学会ST表啦，可惜它是一个静态数据结构。那么我们如何解决下面这题呢？
- 给定一个初始长度为 n 的序列， m 次询问或操作：询问 $[l,r]$ 的最大值，或者在当前序列的末尾插入一个数，强制在线
- 答案非常简单：ST表显然是支持在最后插入一个数的，只要增加 \log 个数据就好了（具体来说是 $f[n-2^s+1,s]$ ，其中 n 是增加后的长度， s 从0到 $\log n$ ）




另一种解题思路.....

- 于是，做法2，我会DP! $f[i][j]$ 表示区间 $[i, j]$ 的答案，用方程 $f[i][j] = \max(f[i][j-1], a[j])$ 来dp即可
- 时间复杂度 $O(n^2+m)$ ，好但还不够
- 现在我们没有做重复的事，但我们做了很多多余的事：我们真的需要那么多区间的答案吗
- 注意到这是一个可重复贡献问题，我们似乎更应该取一些具有代表性的区间，然后每次询问拼出一个答案



另一种解题思路.....

- ——现在我们没有做重复的事，但我们做了很多多余的事：我们真的需要那么多区间的答案吗
- 我知道了，你在说分治 
- 我们用分治来解决这个问题：
- 分治函数 $\text{solve}(S, l, r)$ ，其中 S 是当前需要解决的问题的集合， $[l, r]$ 是当前的分治区间，保证 S 中所有区间都属于 $[l, r]$
- 初始调用： $\text{solve}(Q, 1, n)$ ，其中 Q 是所有问题
- 基本问题： $l=r$ 时， $\text{solve}(S, l, r)$ 将 S 中所有问题的答案设为 $a[l]$



另一种解题思路.....

- 那么，如何解决最重要的分治部分呢？
- 首先，对于某个问题，如果它完全属于 $[l, mid]$ 或 $[mid+1, r]$ ，直接塞进下一层的问题集合里
- 那么我们要解决的问题一定都至少包含 $[mid, mid+1]$
- 我们从 mid 往左算出一个后缀最大值，即 $f1[x]$ 表示 $[x, mid]$ 的最大值；同样，从 $mid+1$ 往右算出一个前缀最大值，即 $f2[x]$ 表示 $[mid+1, x]$ 的最大值
- 那么对于一个询问 $[l0, r0]$ ，它的答案就是 $\max(f1[l0], f2[r0])$ 。
- 时间复杂度 $O((n+m)\log n)$ ，空间复杂度 $O(n+m)$ 。



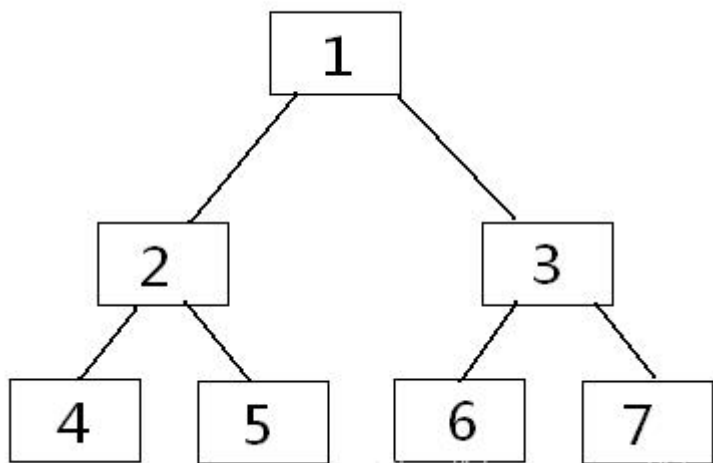
猫树

- 等一下等一下，这个分治怎么把时间复杂度变坏了（本来 m 不需要乘 $\log n$ ），而且还是离线算法
- 这是因为，我们不确定问题会在哪里被解决，从而每个问题会留在集合里一直递归到可以解决它的那层
- ——但是，这个问题被解决的位置真的是不可预见的吗？
- 我们转变一下思路，在递归的过程中不考虑去解决问题，反而把每次求的 $f1$ 和 $f2$ 数组保留下来，然后试着在线回答每个问题
- 为了方便，不妨假设 n 是2的整数次幂（不是就在末尾补0）



猫树

- 先画一棵可爱的二叉树，给它的每个节点打上编号
- 如果用这棵二叉树表示我们的分治过程，那么最下层有 n 个节点
- 我们的目标变成：给定两个最下层的节点，找到它们的LCA

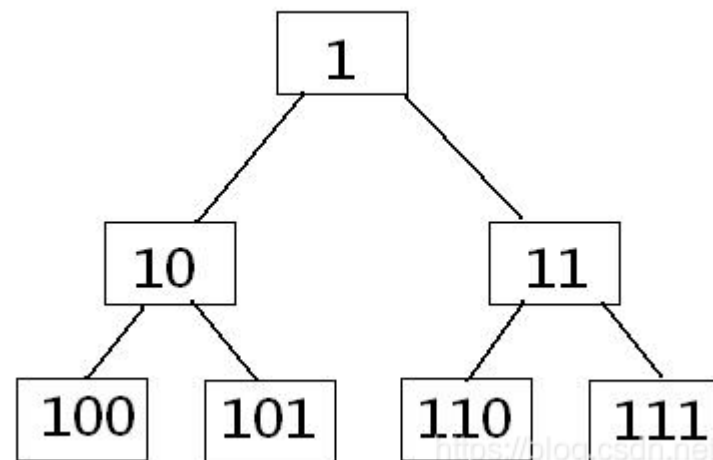
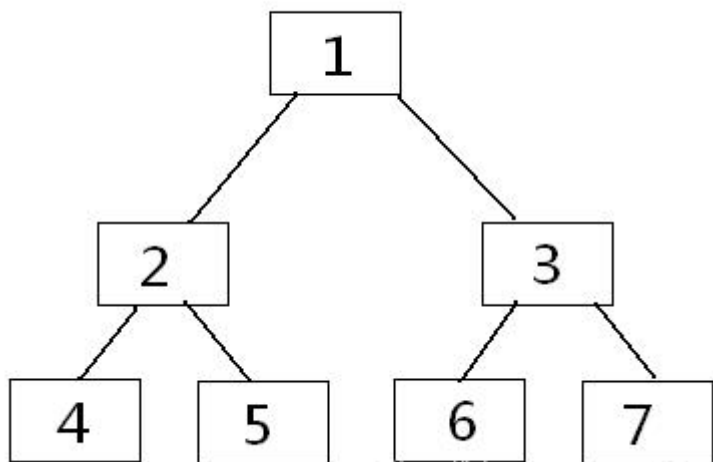


<https://blog.csdn.net/idxycw>



猫树

- 答案貌似不是很明显？那么把它变成二进制
- 原来LCA的编号就是两个底层节点的二进制的LCP（最长公共前缀）



猫树

- 于是现在问题就简单了（询问特判 $l=r$ 的情况）：
- 我们和刚刚分治一样计算 $f1$ $f2$ ，不过不处理问题，而是把它们存下来
- 他们可以存在一个 $\log n$ 行 n 列的数组里，而不用分成两个数组（见下一页的图）
- 数组中 i 位置对应的底层树节点编号 $\text{pos}(i)=i+n-1$
- 节点 l 和 r 的LCA的深度为 $\log n - 1 - \log_2(\text{pos}(l) \text{ xor } \text{pos}(r))$ ， \log_2 为下取整，可以线性时间预处理



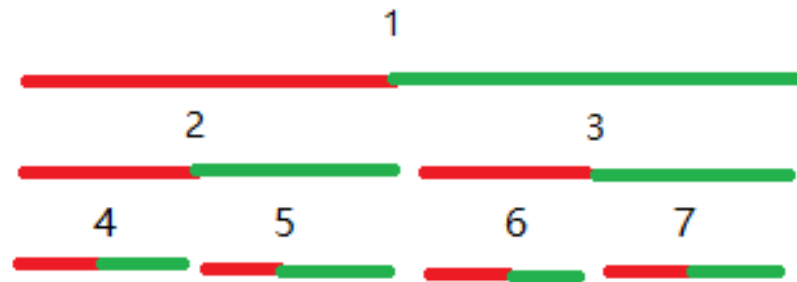
猫树

- 一棵建好的猫树($n=8$)大概长这样
- **红色**表示数组存的是**后缀**最大值, **绿色**表示数组存的是**前缀**最大值, 用一用就会发现它可以 $O(1)$ 回答RMQ问题
- 预处理时间复杂度为 $O(n\log n)$, 猫树的空间复杂度为 $O(n\log n)$, 和ST表没有区别。那我们费这么大劲是为了什么呢???



猫树

- 仔细看！猫树处理区间询问的时候没有用到重复的元素
- 也就是我们不再有“可重复贡献问题”这个约束了！！！！
- 比如猫树可以用来做查询最大子段和，ST表则不行
- 这样我们没有付出任何时空代价删掉了ST表的一个限制条件
- 再见了，ST表！哦不过猫树不支持在线末端插入这种神秘操作



大家猜猜

- 我们先做一道简单清新的数据结构题热热身。
- 有一个长度为 N 的 01 序列，初始每个位置都是 1。有 M 次操作，每次给一个区间 $[L,R]$ ，问 $[L,R]$ 中有多少个 1，问完之后请你把他们全改成 0。
- 假设输入输出不占时间，强制在线， N 和 M 均为 10^7 。



并查集

- 在讲刚刚那道题之前，我们先介绍另一种简单的数据结构：并查集
- 并查集可以支持元素间的两种操作：把两个集合合并成一个集合、查询两个元素是不是在同一个集合
- 并查集的实现是一种特殊的森林，每个节点只关心自己和父亲的连边，这使得它的空间复杂度很低且常数很小



并查集

- 先来实现查询操作
- 显然，并查集中，每个集合对应着一棵树
- 两个元素在不在同一个集合=两个点在不在同一棵树=两个点的树的树根是不是一样的
- 于是我们要做的就是快速帮树上的点找到它们的树根
- 在查询一个点的时候，我们先暴力递归到它的树根；接着，对于路上经过的所有点，显然它们的树根都是一样的，我们直接把它们父亲设为树根
- 这种优化叫做路径压缩



并查集

- 接下来实现合并操作
- 合并两个集合=让两棵树变成一棵树=将其中一棵树接到另一棵树上
- 显然我们首先得判断一下它们是不是同一棵树，判断的过程中我们一定会得到两棵树的树根，于是接下来把一个树根接到另一个树根上即可（当然比接到叶子上好）
- 当然，我们应该把点数少的树接到点数大的树上，这样所有点的深度之和比较小
- 这种优化叫做按秩合并



并查集

- 路径压缩优化是并查集最常用、最好写的优化，很多时候我写并查集也只写这一种优化，但理论上来说，要达到最优的复杂度，必须同时写按秩合并优化
- 结合了两种优化以后，并查集的复杂度是传说中的反阿克曼函数乘 n ，在竞赛的范围内，可以认为就是一种线性算法
- 但是，非常现实的问题是.....按秩合并比路径压缩难写多了（本来我们根本不需要记size的）
- 好消息是，Tarjan老爷子经过一番研究发现：其实随机情况下，只写路径压缩就已经是线性的了
- 下一页提供的代码就只写了路径压缩




```
1  #include <bits/stdc++.h>
2  #define For(_, L, R) for(int _ = L; _ <= R; ++_)
3  using namespace std;
4  const int MaxN = 100000 + 10;
5  int Find[MaxN];
6  int F(int N) {
7      return N == Find[N] ? N : Find[N] = F(Find[N]);
8  }
9  int main() {
10     int N = 100;
11     for(int i = 1; i <= N; ++i)
12         Find[i] = i;
13     int A = 20, B = 30;
14     if(F(A) == F(B)) /* A 和 B 在一个集合里 */;
15     else /* A 和 B 不在一个集合里 */;
16     int C = 40, D = 50;
17     /* 合并 C 和 D 所在的集合 */
18     if(F(C) != F(D)) Find[C] = D;
19     if(F(C) != F(D)) Find[C] = F(D);
20     if(F(C) != F(D)) Find[F(C)] = D;
21     if(F(C) != F(D)) Find[F(C)] = F(D);
22     if(F(C) != F(D)) Find[Find[C]] = Find[D];
23     /* 以上 5 种写法谁对谁不对? */
24 }
```

答案

- （主要是怕我自己忘了） 错错对对对



并查集

- 有没有按秩合并比路径压缩好的时候呢？当然也是有的，那就是我们不希望这个并查集森林结构变化太剧烈的时候，例如，当我们需要可持久化或者可撤销的时候
- 由于和今天的主题不太相关就不过多展开了



UVA11987 Almost Union-Find

- <https://www.luogu.com.cn/problem/UVA11987>

题意翻译

有 n 个集合， m 次操作。规定第 i 个集合里初始只有 i 。有三种操作：

1. 输入两个元素 p 和 q ，若 p 和 q 不在一个集合中，合并两个元素的集合。
2. 输入两个元素 p 和 q ，若 p 和 q 不在一个集合中，把 p 添加到 q 所在的集合。
3. 输入一个元素 p ，查询 p 所在集合的元素个数和所有元素之和。

【数据范围】

$$1 \leq n, m \leq 10^5, 1 \leq p, q \leq n.$$



UVA11987 Almost Union-Find

- 操作1和3非常常规——既然树根是一个集合的代表点，我们就用它来存集合点数和集合元素和，合并的时候直接更新树根（不是树根的点上存的信息统一无视）
- 可是操作2删除元素？我可没听说过这种操作啊.....
- 有一个非常经典的技巧：给节点打上废除标记代替删除
- 我们用一个数组来维护每个元素在并查集中的节点编号（没有被任何元素认领的节点就是被废除的节点），进行2操作的时候，设置一个新的节点作为p元素的节点（原先的节点就被自动废除了）并连进q所在的树，修改p和q所在树根的信息即可
- 时间复杂度 $O(n)$



扩展域并查集

题目描述

 复制Markdown  展开

动物王国中有三类动物 A, B, C ，这三类动物的食物链构成了有趣的环形。 A 吃 B ， B 吃 C ， C 吃 A 。

现有 N 个动物，以 $1 \sim N$ 编号。每个动物都是 A, B, C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 $1\ X\ Y$ ，表示 X 和 Y 是同类。
- 第二种说法是 $2\ X\ Y$ ，表示 X 吃 Y 。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

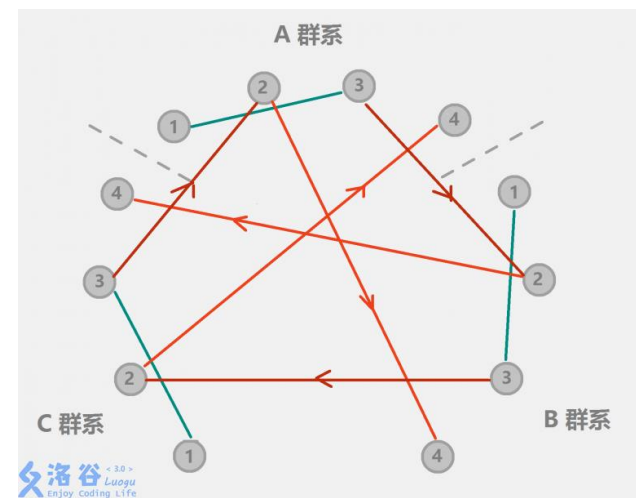
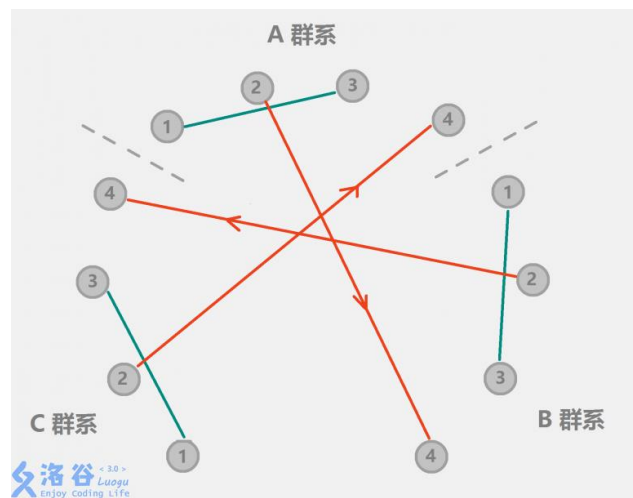
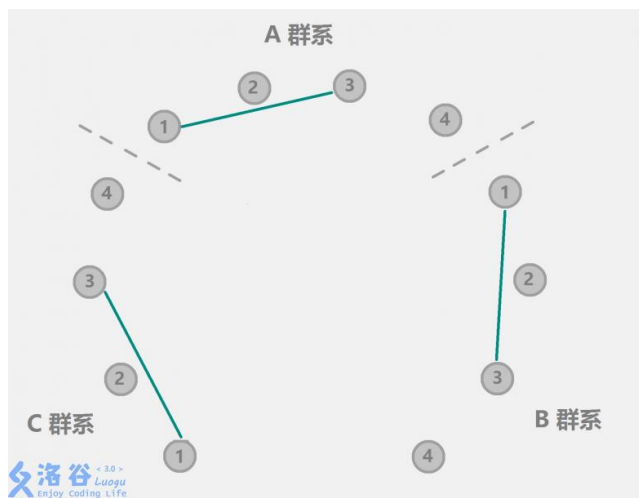
- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X ，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。



扩展域并查集

- 我们将并查集开成点数的三倍，链接关系相当于说：“如果我是A，那么你是B（反之亦然）”。
- 下图分别对应：1和3是同类、2吃4、3吃2
- 当一句话试图连接自己和自己的替身时就是假话。



带权并查集

• P1196 [NOI2002] 银河英雄传说

题目描述

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 $1, 2, \dots, 30000$ 。之后，他把自己的战舰也依次编号为 $1, 2, \dots, 30000$ ，让第 i 号战舰处于第 i 列，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 $M\ i\ j$ ，含义为第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令： $C\ i\ j$ 。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。



带权并查集

- 看上去，如果我们要模拟题意的话，就得维护这个树的结构了，这和我们的路径压缩和按秩合并都是背道而驰的
- 但仔细一看，其实树（链）的**结构**我们也没那么关心
- 对于两个点，我们无非就关心：它们**是不是一棵树上的**；它们的**深度**分别是多少
- 因此我们真正在意的其实只有**边权**，这也就是为什么这个数据结构叫做带权并查集



带权并查集

- 首先还是思考：哪个元素作为整个队列的**代表**？果然还是**队首**的元素最合适——被合并的时候，它不会变；合并到别人的时候，它的答案最好计算
- 接着思考：**代表**元素需要保存哪些信息？由于树的结构是不稳定的，我们得保存这棵树**一共有多少个节点**，这样被别人合并进来的时候才能更新别人的答案
- 那么对于其他元素，自然就需要保存**自己和根节点的距离**（对于代表元素，这个值是0）



带权并查集

- 那么如何更新答案呢？
- 合并的时候比较简单：i的dis设为j的num，然后j的num加上i的num（洛谷题解的这两步是多余的，dis[r2]此时一定是0）
- 查询的时候，相当于做了一次**惰性更新**：如果自己的父亲不再是集合的代表，就把自己的dis加上原本父亲的dis，再把自己的父亲设为集合的代表
- 时间复杂度 $O(T+N)$

```
int find(int x) {
    if (x != fa[x]) {
        int k = fa[x];
        fa[x] = find(fa[x]);
        dis[x] += dis[k];
        num[x] = num[fa[x]];
    }
    return fa[x];
}

void merge(int x, int y) {
    int r1 = find(x), r2 = find(y);
    if (r1 != r2) {
        fa[r1] = r2; dis[r1] = dis[r2] + num[r2];
        num[r2] += num[r1];
        num[r1] = num[r2];
    }
}
```



我的实现

```
12 int N = 30000;
13 int Find[MaxN], Dis[MaxN], Num[MaxN];
14 int F(int N) {
15     if(N == Find[N]) return N;
16     int Root = F(Find[N]);
17     Dis[N] += Dis[Find[N]];
18     Num[N] = Num[Find[N]];
19     return Find[N] = Root;
20 }
21 void Merge(int P, int Q) {
22     P = F(P);
23     Q = F(Q);
24     Find[P] = Q;
25     Dis[P] = Num[Q];
26     Num[Q] += Num[P];
27 }
28 void Query(int P, int Q) {
29     if(F(P) != F(Q)) cout << "-1\n";
30     else cout << abs(Dis[P] - Dis[Q]) - 1 << '\n';
31 }
```



回到这题

题目描述

 复制Markdown  展开

动物王国中有三类动物 A, B, C ，这三类动物的食物链构成了有趣的环形。 A 吃 B ， B 吃 C ， C 吃 A 。

现有 N 个动物，以 $1 \sim N$ 编号。每个动物都是 A, B, C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 $1\ X\ Y$ ，表示 X 和 Y 是同类。
- 第二种说法是 $2\ X\ Y$ ，表示 X 吃 Y 。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X ，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。



带权并查集

- 这题也可以用带权并查集做，互吃关系的运算本质上是模3的加法群，用于边权的运算即可，不多赘述



回到这题

- 我们先做一道简单清新的数据结构题热热身。
- 有一个长度为 N 的 01 序列，初始每个位置都是 1。有 M 次操作，每次给一个区间 $[L,R]$ ，问 $[L,R]$ 中有多少个 1，问完之后请你把他们全改成 0。
- 假设输入输出不占时间，强制在线， N 和 M 均为 10^7 。



并查集

- 单次操作 $O(1)$ 遥不可及、 $O(\log) T$ 的飞起，但我们发现，每个 1 只会被改成 0 一次，我们能否想办法把复杂度架在“区间中剩余的 1 的数量”上。
- 正确答案是**并查集**。将每个点建成一个并查集，初始时每个点就是一个集合。每次操作暴力循环 $[L, R]$ ，遇到 1（假设在位置 i ）就将答案 +1，并将这个 1 挪出序列，具体操作是把 i 的父亲指向 $i+1$ 。
- 这样访问到一坨 0 的某个时，直接跳到它的祖先，也就是下一个 1 的位置。
- 等一下，这样好像只能路径压缩不能按秩合并.....？不过数据随机的话还是够快的。还可以想一些卡常的损招，比如可以在挪出 1 的时候记录一下被挪出的位置，直接把它们父亲指向 R 。
- 时间复杂度 $O(N)$ 。



栈

- 接下来是两种比较简单的数据结构：栈和队列，我们先来讲栈
- 使用栈就像在日常生活中使用木桶：
 - 0，桶里一层只能放一个东西（栈是一维数据结构）
 - 1，当你放入东西时，只能放在桶里的最上面（只能放在栈顶）
 - 2，当你取出东西时，只能取出桶里最上面的（只能弹出栈顶）
- 可以发现，栈是先进后出（FILO）的数据结构
- 很多人容易把栈和堆这两种数据结构和操作系统中的堆栈概念搞混，这里必须点明他们是完全独立的概念，我们今天只讨论数据结构方面的概念



栈

- 你也许会好奇：我本来是好端端支持随机访问的数组，为什么要给自己套上枷锁，变得只支持访问其中一位这么愚笨
- 这是因为，栈这种数据结构非常好地抽象出了一种计算机中的常见场景：递归（例如深度优先搜索）
- 打一个不恰当的比方：如果不在地上画停车位，或许车可以停得更密；但往往在地上画了停车位，才能停更多车（好吧这个例子听起来更像内存管理里的分页.....）
- 一定程度的规则限制，让我们的数据结构功能明确、效率更高
- 当然我们是竞赛生，很多时候也要灵活地打破规则



栈

- 做一个简单的例题：给一棵有根树，对于每个叶子节点，输出从根到它的路径经过的所有节点的编号
- 最适合这题的数据结构就是栈：维护一个初始为空的栈，从树根开始dfs，访问一个节点时将它放入栈，如果是叶子就输出栈，访问过它所有孩子后，弹出栈顶（此时栈顶正好是这个节点）
- 输出栈时，我们倒也不必遵守栈的规则，直接访问一遍栈内的元素即可（如果严格按照栈的规则来，得把所有元素弹出来再塞回去，太多余了）
- 从这个例子也可以看出栈对深搜的天然支持



队列

- 接下来是队列
- 使用队列就像在日常生活中排队：
 - 0, 所有人排成一队（队列是一维数据结构）
 - 1, 人来排队的时候只能站在队尾（只能在队尾插入元素）
 - 2, 只有队首的人会离开队列（只能弹出队首的元素）
- 可以看出，队列是先进先出（FIFO）的数据结构
- 虽然工程上优先队列经常被放在一些名为queue的头文件里，但我认为它不是一种队列，因此今天也不讲



队列

- 和栈类似，队列也是一种优秀的抽象
- 不过它比栈更直观一点：它抽象的就是“排队”的过程
- 因此在操作系统的进程调度等场景中很常见（怎么今天一直在讲操作系统）
- 同时，正如栈天然地支持了深搜一样，队列天然地支持了广搜



循环队列

- 如果用一个数组来存放栈，数组的大小就是栈的极限大小，这没有问题
- 但如果用一个数组来存放队列，在多次入队出队后，实际被使用的内存空间会越来越右移，数组的大小是极限入队次数
- 这样就会出现明明占了很多内存，但队列使用空间却很有限的情况
- 为了解决这种情况，我们采用循环数组的形式来记录队列，可以简称之为循环队列



循环队列

- 在长度为 n 的循环队列中， i 的下一个元素为 $(i+1)\bmod n$
- 这是一个很好的解决方案，但有一个微小却不可忽视的缺陷：用长度为 n 的数组存储循环队列，队列中的元素不能超过 $n-1$ 个
- 这是因为，队列中存放 0 个和 n 个元素的时候，都有 $\text{head}=\text{tail}$ ，这两种情况无法区分，我们当然是舍弃 n 个元素的情况，默认为 0 个元素



队列

- 做一个简单的例题：给一张图，边的长度均为1，有一个点是起点，输出从它出发到其他各点最短的路径的长度
- 我们用一个队列存储目前有用的点，每次取出队首的点，把它连向的点中没有进入过队列的点加入队尾（访问过它周围所有点后，这个点就变成了没用的）
- 初始化时，队列为空，将起点的距离设为0并放入队列
- 当队列再次变为空时，结束整个过程



队列

- 我们来简单说明一下为什么这样做是对的
- 首先，这个做法如果记录下每个点被谁选中，可以导出一条合法的路径。因此，它不会把距离算得偏小
- 接下来，这个做法可以正确找到所有距离为0的点（即将起点放入空队列这一步）
- 接下来，假设这个做法找到了所有距离为 i 的点，那么，直到所有距离为 i 的点被弹出队列以前，这个做法一定找到了所有距离为 $i+1$ 的点并补在了队尾（它们之前一定没被找到过，否则它们的距离就不是 $i+1$ 了，并且它们至少和一个距离为 i 的点有连边），于是这个做法也能找到所有距离为 $i+1$ 的点



0-1BFS

- 稍微修改一下上面这道题：给定一张图，所有边的边权不是0就是1，有一个点是起点，求所有点的最短路



0-1BFS

- 稍微修改一下上面这道题：给定一张图，所有边的边权不是0就是1，有一个点是起点，求所有点的最短路
- 聪明的小朋友可能想到：边权为0，他们的最短路答案一定相同，用并查集把它们缩成一个点，重建这个图，跑上一问的bfs算法即可
- 这种做法挺好的，但我们还有一种只需要队列的更好的做法，只不过我们要再次打破规则



0-1BFS

- 如果大家有仔细思考上一问的证明过程，或许会发现一个好玩的性质：不论何时，队列中的点的距离大小只有2种，靠近队尾的一半是 $d+1$ ，靠近队首的一半是 d
- 在上一问，我们的边权都是1，因此新的点的距离是 $d+1$ ，我们自然地将它放进了队尾
- 这一问中，如果我们通过长度为0的边访问到了没访问过的点，说明它的距离就是 d ，那么我们打破队列的规则，给它开个特权，直接插入队首就好了
- 证明和上一问的证明方法类似，时间复杂度依旧是线性



栈和队列 小结

- 栈和队列是两种非常重要的基本数据结构，虽然它们的原理和实现都非常简单，但是有很多值得深挖的地方，希望大家可以借助各种方法来加深对它们的理解
- 队列的实现要注意可以进行循环化（竞赛中并不太常用，你往往可以把数组开到足够大）以及双端化
- 最重要的就是理解这两种数据结构分别在深搜和广搜中使用
- 比如可以结合一些基本问题的不同角度分析，例如，对于走迷宫问题，深搜找的是走出迷宫的所有方法，广搜找的是走出迷宫的最短路径



双栈模拟队列

- 这是一个跟竞赛不太有关（当然，也可以出成交互题.....）的经典题：如何只用两个栈来模拟一个队列



双栈模拟队列

- 这是一个跟竞赛不太有关（当然，也可以出成交互题.....）的经典题：如何只用两个栈来模拟一个队列
- 入队：入栈A
- 出队：如果栈B空，循环执行：出栈A，并将出栈元素入栈B，直到栈A空。出栈B
- 时间复杂度证明：每个元素恰好进出AB栈各一次，复杂度线性
- 正确性证明：越早进入栈A的元素，越晚弹出栈A，越晚进入栈B，越早弹出栈B，因此整个过程是FIFO的



单调栈

- 普通的栈和队列果然还是不太能满足我们的需求，接下来来介绍基于它们衍生的两种强大的数据结构：单调栈和单调队列
- 单调栈本质上就是一个栈，只不过我们通过设置特殊的出入栈规则，使得栈里的元素的某个性质是单调的
- 单调栈经常和二分结合使用，但也别把自己骗进去了，有些时候需要的不是在单调栈上二分，而是弹出无用的元素（具体看例题）



单调栈

- 有 n 位同学，第 i 位同学的座号为 i ，成绩为 a_i
- 做作业的时候，第 i 位同学会选择一个成绩比自己好、座号比自己小、座号尽可能大的同学来抄
- 输出每个同学会抄到谁的作业，或者-1（抄不到任何人的作业）



单调栈

- 我们考虑用一个栈来维护“对后面的同学有价值”的同学
- 一个同学有价值，要么是他的座号比较大，要么他的成绩比较好
- 显然越往栈顶座号越大，于是越往栈顶成绩越差
- 考虑到一个新的同学的时候，我们在单调栈上二分，找到第一个成绩比他好的同学，抄他的作业
- 之后，如果栈顶的同学（座号比自己小）成绩比自己差，他就从此失去了价值，可以弹出栈
- 循环上一步，直到栈顶的同学成绩比自己好，此时把自己加入栈（自己成为座号最大、成绩最差的）



单调栈

- 考虑到一个新的同学的时候，我们在单调栈上二分，找到第一个成绩比他好的同学，抄他的作业
- 之后，如果栈顶的同学（座号比自己小）成绩比自己差，他就从此失去了价值，可以弹出栈
-以上是我在南方科技大学2022-2023秋季学期数据结构与算法分析课期中考试中的答案，时间复杂度 $O(n\log n)$
- 当时得出这个答案，凭借的就是我不知道从哪里听来的口诀：在单调栈上二分



单调栈

- 考虑到一个新的同学的时候，我们在单调栈上二分，找到第一个成绩比他好的同学，抄他的作业
- 之后，如果栈顶的同学（座号比自己小）成绩比自己差，他就从此失去了价值，可以弹出栈
- 仔细一看这个做法就是纯扯淡啊，下面这步弹掉了所有成绩比自己差的同学，那上面这步在二分什么东西？？？



单调栈

- 我们考虑用一个栈来维护“对后面的同学有价值”的同学
- 一个同学有价值，要么是他的座号比较大，要么他的成绩比较好
- 显然越往栈顶座号越大，于是越往栈顶成绩越差
- 考虑到一个新的同学的时候，如果栈顶的同学（座号比自己小）成绩比自己差，他就从此失去了价值（我不能抄，后面的同学不如抄我的），可以弹出栈
- 循环上一步，直到栈顶的同学成绩比自己好，此时抄他的作业，然后把自己加入栈（自己成为座号最大、成绩最差的）
- 时间复杂度 $O(n)$



单调队列

- 理解了单调栈的话，单调队列也并不困难
- 单调队列本质上就是一个双端队列，只不过我们通过设置特殊的出入队规则，使得队列里的元素的某个性质是单调的
- 单调队列和单调栈都可以用来优化dp，其中单调队列还可以摇身一变升级成斜率优化来优化dp，但不是今天的重点



单调队列

- 有 n 位同学，有一个固定的抄作业距离限制 d ，第 i 位同学的座号为 i ，成绩为 a_i
- 做作业的时候，第 i 位同学会选择一个成绩尽可能好、座号比自己小、座号和自己距离不超过 d 的同学来抄
- 输出每个同学会抄到谁的作业



单调队列

- 这题和上一题有什么变化呢？ 我们想要极化的量从下标变成了值
- 如果不考虑距离 d ， 那么问题很简单： 记录当前成绩最好的同学就好了
- 然而考虑了距离 d 以后， 每个同学提供的价值都有了一个过期的时间
- 因此， 我们用一个队列来维护当前还没过期的有价值的同学们
- 由于 d 是定值， 先出现的同学一定先过期， 这也符合队列FIFO的特点



单调队列

- 那么，对于队列内的同学们，由于他们都没有过期，他们之间就有优劣之分了：
- 一个同学有价值，要么是座号比较大，要么是成绩比较好
- 因此，由于我们的单调队列里从队首到队尾是座号越来越大的，成绩应当是越来越差的
- 如果我们新加入的同学成绩比**队尾**的同学好，队尾的同学就失去了价值，我们应当不断弹出队尾，然后将新同学放在队尾
- 同时，如果**队首**的同学超出了抄作业的距离限制，我们就弹出队首



单调队列

- 总结一下，单调队列里，越靠近队首的元素值越好，但同时他也容易过期；越靠近队尾的元素值越差，但说不定它有机会等到队首的值过期以后自己发挥作用
- 从单调队列里取值的时候，应该不断弹出队首过期的元素，然后取队首的值
- 把当前的元素加入单调队列的时候，应该不断弹出队尾不如自己的元素，然后加入队尾
- 时间复杂度 $O(n)$



单调栈和单调队列 小结

- 第一题：做作业的时候，第 i 位同学会选择一个成绩比自己好、座号比自己小、座号尽可能大的同学来抄
 - 限制条件是元素的值，极化对象是下标
 - 座号越大的同学成绩应该越差
- 第二题：做作业的时候，第 i 位同学会选择一个成绩尽可能好、座号比自己小、座号和自己距离不超过 d 的同学来抄
 - 限制条件是下标，极化对象是元素的值
 - 座号越大的同学成绩应该越差



单调栈和单调队列 小结

- 那么，什么时候应该用单调栈，什么时候应该用单调队列呢？
- 这个问题很难给出一个明确的答案，需要大家深入理解这两种算法，做题的时候搞清楚：
 - 1，自己要极化的是什么
 - 2，元素之间互相取代的规则是什么（什么样的元素是有价值的）
- 再选用适合的一种



总结

- 今天我们了解了：
 - 1, 链表的实现、意义、使用场景, 以及哈希表
 - 2, ST表的思路、实现, 以及升级后的数据结构猫树
 - 3, 各种并查集和他们的用法
 - 4, 队列和栈, 以及单调队列和单调栈的原理、使用
- 希望大家熟练掌握, 有所收获





SUSTech

Southern University
of Science and
Technology

谢谢大家!

