# STL初步

自己定义的struct必须要重定义小于号

## 全排列

不用写dfs了。

// 如果vec不是有序的，先排序可以枚举所有排列  
 do {  
 ++tot;  
 for (int i = 0; i < 8; ++i) {  
 sort(line[i], line[i] + 3, [&](int aa, int bb) {  
 return ord[aa] < ord[bb];   
 });  
 if (a[line[i][0]] == a[line[i][1]] && a[line[i][1]] != a[line[i][2]]) {  
 ++cnt;  
 break;  
 }  
 }  
 } while(next\_permutation(ord + 1, ord + 10));  
// 如果是C数组，就传两个指针进去

这个函数会修改容器。

## string

定义

string s;

长度（复杂度是O(1)的）

s.length();

定义流

#include <sstream>  
//...  
string line;  
stringstream ss(line);  
ss >> a; //就可以直接从line里面读或写

读取一整行

getline(cin, line);

## 重定义运算符

struct note{...}  
note operator +(note aa, note bb) {...}

## vector（可变长数组）

vector<int> a; //定义  
a.size(); //a的大小  
a.resize(); //改变a的大小  
a.push\_back(); //插入元素  
a.pop\_back(); //删除元素

## set（集合）

**集合中的元素总是按照升序排列好的**，也就是说集合可以直接lower\_bound()以及upper\_bound()

set<string> dict; //定义  
dict.insert(s); //插入元素  
for (set<string>::iterator it = dict.begin(); it != dict.end(); ++it)  
//↑遍历，其中定义了set<string>类型的名为it的迭代器  
dict.erease(a); //删除元素  
dict.find(s); //查找

### multiset

不去重的set，可以有效平替平衡树

## map（映射）

对于集合的大部分操作也可适用于map，但更多时候map是作为下标强大的数组使用的。

map<string, int> m; //这里m是string为下标，里面存int的数组  
m["July"] = 7; //于是就可以有这样的操作

由于其下标功能强大，map通常可以用来做离散处理，只是在复杂度上套了一个log。（还有一个很大的常数）

如果访问到了不存在的键，map会自动给该键赋值，int类型的默认值为0。若需要判断某个键是否存在，可以用count()函数，返回某个键出现的次数（map中的键是唯一的）。

### unordered\_map

unordered\_map是基于哈希实现的，其内部没有顺序，但复杂度是O(1)的。支持如下操作：

unorderd\_map<string, int> list; //定义  
list["233"] = 233; //赋值  
list.count(233); //可以用来判断是否存在  
list.erease(233); //删除映射关系  
list.clear(); //清空

## stack, queue（栈、队列）

int a;  
stack<int> s;  
s.push(1);  
a = s.top();  
s.pop();

对于队列而言，操作是类似的，只有取队首不同：a = q.front()

不过于我个人而言，我更倾向于手写队列

### priority\_queue（优先队列）

常用于Dijkstra的堆优化，求单源最短路。默认的堆顶是最大的元素。其操作和stack相同，也是用top()取堆顶。

如果想要堆顶元素最小，可以有以下几种方法：

* 存相反数进去；
* 定义一个struct，然后把小于号重定义为大于号；
* 按照如下方法定义优先队列，但需要注意的是，最后“< <”中间必须有空格，不然有些（并非所有）编译器会认为这是位移运算符。

priority\_queue<int, vector<int>, greater<int> > //top()取出来的就是最大的

# DP

Future never has to do with past time, but present does.

未来与过去无关，现在决定未来。

问题需要满足无后效性才能用动态规划去解。

## 线性DP

### 最大连续区间和

显然，若序列非负，那越长越好。

必须考虑有正有负的情况。

求最小和同理

以上です。

↑~~不过只适用于写报告，写信千万别用，太冷漠~~

### 最长上升子序列（LIS）

设f[i]表示第位置1到i的以i为结尾的最长上升子序列的长度，那么状态转移方程为：f[i] = max(f[j]) + 1 (j<=i, a[j]<=a[i])，很好理解。

算法竞赛入门经典（第二版）》中未给出优化到的版本，ChatGPT给出了如下代码：

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int LIS(vector<int>& nums) {  
 vector<int> tails;  
 for (int num : nums) {  
 auto it = lower\_bound(tails.begin(), tails.end(), num);  
 //如果不要求严格递增，即求最长不下降子序列，替换为upper\_bound()即可  
 if (it == tails.end()) tails.push\_back(num);  
 else \*it = num;  
 }  
 return tails.size();  
}  
  
int main() {  
 vector<int> nums{10, 9, 2, 5, 3, 7, 101, 18};  
 cout << LIS(nums) << endl; // output: 4  
 return 0;  
}

具体实现时，我们可以使用一个数组tails，其中tails[i]表示长度为i+1的所有LIS中最后一个元素的最小值。遍历原序列，对于每个元素x，使用二分查找在tails数组中找到第一个大于等于x的位置k，更新tails[k]为x，长度为k+1的LIS的末尾元素为x。遍历完整个序列后，tails数组中的最大下标即为LIS的长度。

不难理解，长度越长的最长上升子序列的结尾越大，那么我们可以通过二分查找的方式把第i个元素插入到最长的LIS后面，然后用它的值更该新长度的LIS的结尾元素的最小值。

需要注意的是，求上升子序列用lower\_bound()，而求不下降子序列用upper\_bound()，你可以想想为什么。

### 最长公共子序列

现有两个数列AB，欲求其最长公共子序列。可以在B中找到每个A中元素a[i]第一次在B中出现的位置，若为A中重复的第n个元素就找第n次出现的位置。按照出现位置的先后重新定义数列A的大小关系，求最长不下降子序列即可。

### 01背包

有种物品，每种只有一个。第中物品的体积为，重量为，选一些物品装入容量为的背包，使得背包内物品在总体积不溢出的情况下最大。

设表示“把前个物品装入容量为的背包中的最大总重量，状态转移方程也不难得出：，先枚举i，后枚举j即可。

* 滚动数组

显然，更新只用到了上一行的数据，那么数组第一维长度可以是2。更进一步，第一维可以直接去掉，**但是j要逆序枚举**

for (int i = 1; i <= n; ++i)  
 for (int j = v; j >= 0; --j)  
 f[j] = max(f[j], f[j - c[i]] + w[i]);

## 树形DP

### 树的最大独立集

对于一颗个节点的无根树，选尽量多的节点，使其互不相邻。

只要任选一根r，无根树就变成了有根树。定义表示以为根的子树的最大独立集的大小，那么节点i只两种决策，选或不选。若选，那其儿子都不能选；若不选，那其儿子都可以选。

由此不难得出状态转移方程：。其中表示的孙子，而表示的儿子。

上述方案需要枚举某个节点的儿子和孙子（**填表法**），看起来不是那么优雅。如果计算出一个的值后，用其更新i的父亲和祖父节点的累加值（即$$的两项）（**刷表法**），看起来就优雅许多了。

## 数位DP

数位DP一般解决如下问题：

1. 要求统计满足一定条件的数量（如：计数）；
2. 这些条件经过转化后可以使用「数位」的思想去理解和判断；
3. 输入会提供一个数字区间（有时也只提供上界）来作为统计的限制；
4. 上界很大，暴力枚会超时

一个常用的计数问题的技巧：

那么有了通用答案数组，接下来就是统计答案。统计答案可以选择记忆化搜索，也可以选择循环迭代递推。为了不重不漏地统计所有不超过上限的答案，要从高到低枚举每一位，再考虑每一位都可以填哪些数字，最后利用通用答案数组统计答案。

# 基本算法

## 暴搜

### 全排列

全排列是可以不用写dfs的

do {  
 for (int num : sequence) {  
 std::cout << num << " ";  
 }  
 std::cout << std::endl;  
} while (std::next\_permutation(sequence.begin(), sequence.end()));

## 贪心

贪心算法只关注当前问题的最优解，并不关注全局。在一些经典算法中有贪心算法的体现，比如Dijkstra算法以及Kruskal算法。

根据这样的性质，要求贪心法解决的问题有“无后效性”——当前的决策不会影响到后续的决策。

一般来说，对于贪心问题，需要抓住两个方面：

* 找到合适的“贪心标准”，如区间安排问题中找最小的结束时间；
* 进行恰当的排序，如Kruskal算法中对边权尽行排序。

## Huffman编码

一套正确的不定长编码一定是互不为前缀的，我们把这样的编码成为前缀码。那么，如果构建一棵树，每个节点连向左右儿子的边分别赋值为0及1，那么从根到每个叶子的路径即可表示一个编码，包括n个叶子的一棵树就能构成一套前缀码。

霍夫曼Huffman编码是基于贪心的压缩空间方法构建前缀码的，具体思想为将更短的编码赋给出现频率高的字串。构建Huffman树满足以下两个性质：

* 贪心选择性质：第一步贪心法选择保留最优解。
* 最优子结构性质：原问题的最优解包含子问题的最最优解。

基于这两个性质，算法实现如下

1. 将所有字串按照出现频率排成表P，存入队列中，记为队列P；
2. 将两个频率最低的节点取出并合并，**即指向同一个根节点**；将其根节点视为新元素存入新队列Q中，新节点的频率为两个节点的频率和；
3. 每次比较Q和P的队首，将最小的两个元素取出合并，放入P的队尾。

复杂度主要在排序上，是O(nlogn)的

上述方法的Huffman树是二叉的，但是NOI2015荷马史诗中要求构造k叉Huffman树。对于k叉的情况，只需要维护一个堆，每次取最小的k个节点合并即可。

## 二分法

### 二分答案

以P3853为例

while (left <= right) {  
 int mid = (left + right)/2;  
 if (ok(mid)) ans = mid, right = mid - 1;  
 else left = mid + 1;  
}

二分的是答案，而ok中判断的是该答案是否可行

### 二分查找（STL）

lower\_bound(a, a+n, x); //开始指针，结束指针，查找变量  
upper\_bound(a, a+n, x);

两个函数都要求a是升序排列的，lower\_bound()返回≥x的第一个位置，而upper\_bound()返回第一个>x的位置。

值得注意的是，如果是STL中的数据结构（尤其是vector），那么开始和结束的位置可以是两个迭代器，比如：

vector<int> a;  
//...  
lower\_bound(a.begin(), a.end(), x);

map内置了二分查找方法，可以通过.lower\_bound()来查找相应的的键

auto it = myMap.lower\_bound(3);  
  
if (it != myMap.end()) {  
 // 找到了  
} else {  
 // 没找到  
}

## 差分

可以看做前缀和的逆操作，数学上，差分可以看做离散的导数。

简单定义：记录上一个数和这个数的差值（因为数组里相邻数的间隔是一定的）

那显然，差分数组的前缀和就是原来的数组。由此优秀的性质，**可以地进行区间修改，但是查询是，因此适用于离线的数据修改**

裸的差分考察的很少，更多的是作为辅助，如NOIP2012 day2 T2借教室（P1083），就是二分答案然后差分维护。

## 排序

### 归并排序

基本应用：求逆序对

在归并的时候，如果右边区间的某个数放到了左边，那说明左边区间剩余的所有数都会产生逆序对。

下面给出py代码

def merge\_sort(arr):  
 if len(arr) <= 1:  
 return arr, 0  
   
 mid = len(arr) // 2  
 left, left\_count = merge\_sort(arr[:mid])  
 right, right\_count = merge\_sort(arr[mid:])  
 merged, merge\_count = merge(left, right)  
   
 return merged, left\_count + right\_count + merge\_count  
  
def merge(left, right):  
 result = []  
 count = 0  
 i = j = 0  
   
 while i < len(left) and j < len(right):  
 if left[i] <= right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
 count += len(left) - i  
   
 result.extend(left[i:])  
 result.extend(right[j:])  
   
 return result, count  
   
n = int(input())  
arr = input().split()  
arr = [int(num) for num in arr]  
sorted\_arr, ans = merge\_sort(arr)  
print(ans)

# 字符串

## 哈希

### 哈希表

#include <unordered\_map>

其余略

### 字符串哈希

和oi-wiki上不同，这里介绍另一种哈希，这样做方便进行单点修改。

选定的进制为，并且要对取模（一个大质数）。

* 求算哈希值

如字符串的哈希值可以表示为，我们可以把字符串中下表较大的位看成高位（反过来也可以，但利用前缀求子串不同）

* 代码

先给出几个可以用于双模哈希的模数。

915650123  
915707077  
945519319  
1000000123  
1000000321  
1000008899  
1000001213  
1000018333  
1000019333  
1000222337  
1000223377

下面是一个值域为的双模哈希，一般认为可以处理级别的不同字符串。但是**实际应用的时候从来不会这么写**，因为效率太低。

下面给出前缀哈希数组的求解函数，这份代码中求解了正反两个方向的双模哈希（共4组）用以判断回文子串。

int P[2] = {915707077, 1000223377}, B[2] = {77, 99};  
  
void get\_hash(char \*s) {  
 int len = strlen(s + 1);  
 for (int t = 0; t < 2; ++t) { // 双模hash  
 ll res = 0, b = 1;  
 for (int i = 1; i <= len; ++i) { // 正向hash  
 res = (res + b \* (s[i] - 'a')) % P[t];  
 b = (b \* B[t]) % P[t];  
 // tr[t].add(i, res);  
 // tr[t].add(i, res, P[t]);  
 tr[t].a[i] = res;  
 }  
 tr[t].build(1, 1, len, P[t]);  
  
 res = 0;  
 b = 1;  
 for (int i = len; i >= 1; --i) { // 反向hash  
 res = (res + b \* (s[i] - 'a')) % P[t];  
 b = (b \* B[t]) % P[t];  
 re[t].a[i] = res;  
 }  
 re[t].build(1, 1, len, P[t]);  
 }  
}

特别注意**所有涉及到乘法的运算都有更可能溢出**，因此要用long long维护

* 前缀哈希

可以预处理出一个数组表示的前缀的哈希值。

观察两个式子：

那么子串的哈希值可以表示为：

若要比较两个子串是否相同，只需要注意将式子左边的指数对其即可。

下面给出用线段树维护前缀hash数组，并通过求子串正反向哈希值并判断回文的例子：

int P[2] = {915707077, 1000223377}, B[2] = {30, 33};  
int BINV[2] = {fpow(B[0], P[0] - 2, P[0]), fpow(B[1], P[1] - 2, P[1])};  
// 费马小定理求大质数取模乘法的逆元  
//...  
int l, r;  
cin >> l >> r;  
for (int t = 0; t < 2; ++t) {  
 int h1, h2, exp1 = l - 1 + 1, exp2 = n - r + 1;  
 // 这里少计算了第一位的哈希，所以指数为做相应调整，不过无伤大雅  
  
 h1 = mod(tr[t].query(1, 1, n, r, r, P[t]) -   
 tr[t].query(1, 1, n, l, l, P[t]), P[t]);  
 h2 = mod(re[t].query(1, 1, n, l, l, P[t]) -   
 re[t].query(1, 1, n, r, r, P[t]), P[t]);  
  
 // 利用费马小定理求算乘法逆元，并将多余的指数除掉  
 h1 = mod((ll)h1 \* fpow(BINV[t], exp1, P[t]), P[t]);   
 h2 = mod((ll)h2 \* fpow(BINV[t], exp2, P[t]), P[t]);  
  
 if (h1 != h2 || s[l] != s[r]) {  
 ans = false;  
 break;  
 }  
}

显然，若不用支持修改，那线段树也没有开的必要。

单点修改也是同理，若要修改位置，显然只影响了的区间，并且在区间内的每一个前缀，的贡献是相同的。若要单点修改，只要转化成修改的后缀区间前缀哈希值即可。

以下是单点修改转换为区间修改哈希值的例子：

int x;  
char c;  
cin >> x >> c;  
int offset = c - s[x];  
s[x] = c; // 修改字符串  
for (int t = 0; t < 2; ++t) {  
 tr[t].add(1, 1, n, x, n, mod((ll)offset \*   
 fpow(B[t], x - 1, P[t]), P[t]), P[t]);  
 re[t].add(1, 1, n, 1, x, mod((ll)offset \*   
 fpow(B[t], n - x, P[t]), P[t]), P[t]);  
}

* **如何用哈希判断字符串是否相等？**

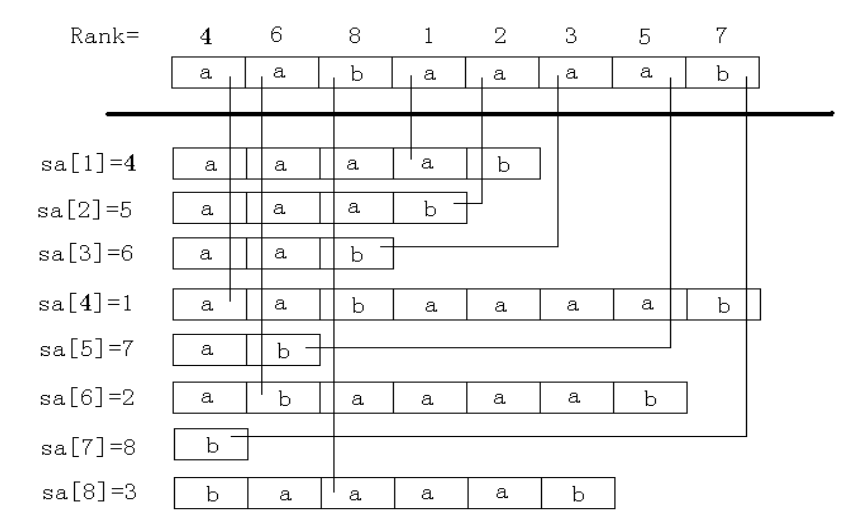
基本思路是，二分查找两个串最长的相同的前缀，然后看的位置的大小关系

inline int halffind() {  
 int l = 1, r = (rig - lef + 1) >> 1;  
 int mid, res = 1;  
 while (l <= r) {  
 mid = (l + r) >> 1;  
 if (check(mid)) { // 判断两段的hash值是否相等  
 res = mid;  
 l = mid + 1;  
 } else {  
 r = mid - 1;  
 }  
 }  
 return res;  
}

## 后缀数组（SA）

后缀数组（Suffix Array）主要关系到两个数组：和。

其中，表示将所有后缀排序第小的后缀数组编号，也就是所说的后缀数组；表示后缀的排名。这两个数组满足性质：



SA

求解sa数组的基本思想涉及到倍增和基数排序：

* 倍增

若已知长度为的子串的排名，那只需将前半部分作为第一关键字，后半部分作为第二关键字，即可求得长度为的子串的排名。

* 基数排序

对于上文中的双关键字排序还可进一步使用基数排序优化到级别。

当然，还需要一些**常数优化**，于是我们就得到了OI Wiki上的板子：

#include <algorithm>  
#include <cstdio>  
#include <cstring>  
#include <iostream>  
  
using namespace std;  
  
const int N = 1000010;  
  
char s[N];  
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];  
  
int main() {  
 int i, m, p, w;  
  
 scanf("%s", s + 1);  
 n = strlen(s + 1);  
 m = 127;  
 for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;  
 memcpy(oldrk + 1, rk + 1, n \* sizeof(int));  
 for (p = 0, i = 1; i <= n; ++i) {  
 if (oldrk[sa[i]] == oldrk[sa[i - 1]]) {  
 rk[sa[i]] = p;  
 } else {  
 rk[sa[i]] = ++p;  
 }  
 }  
  
 for (w = 1; w < n; w <<= 1, m = n) {  
 // 对第二关键字：id[i] + w进行计数排序  
 memset(cnt, 0, sizeof(cnt));  
 memcpy(id + 1, sa + 1,  
 n \* sizeof(int)); // id保存一份儿sa的拷贝，实质上就相当于oldsa  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];  
  
 // 对第一关键字：id[i]进行计数排序  
 memset(cnt, 0, sizeof(cnt));  
 memcpy(id + 1, sa + 1, n \* sizeof(int));  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];  
  
 memcpy(oldrk + 1, rk + 1, n \* sizeof(int));  
 for (p = 0, i = 1; i <= n; ++i) {  
 if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&  
 oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {  
 rk[sa[i]] = p;  
 } else {  
 rk[sa[i]] = ++p;  
 }  
 }  
 }  
  
 for (i = 1; i <= n; ++i) printf("%d ", sa[i]);  
  
 return 0;  
}

* 应用

给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中字典序最小的一个。

显然首尾不同的时候就贪心地选最小的那个，若首尾相同则需要比较原字符串和反字符串的后缀的字典序，取较小的更优。

一种做法是上面提到的哈希，然后二分比较字符串大小。

使用后缀数组需要将反串复制到原串的末尾，中间插入一个比合法字符都 小的字符以免影响，然后求相应的后缀数组。

比如这样：

// 复制反字符串  
int nn;  
scanf("%d", &nn);  
n = nn \* 2 + 1;  
for (int i = 1; i <= nn; ++i) {  
 int c = getchar();  
 while (c > 'Z' || c < 'A') {  
 c = getchar();  
 }  
 s[i] = c;  
 s[n - i + 1] = c;  
}  
s[nn + 1] = 'A' - 1; // 设定最小值  
  
// 利用后缀数组求解  
int l = 1, r = nn, tot = 0;  
while (l <= r) {  
 putchar(rk[l] < rk[n - r + 1] ? s[l++] : s[r--]);  
 ++tot;  
 if (tot == 80) {  
 putchar('\n');  
 tot = 0;  
 }  
}

于是乎，我们似乎能得到后缀数组的规律——大部分用后缀数组解决的问题都需要**按一定方式将原字符串展开**，比如直接复制两份，或者复制反串等。

### height 数组

定义：，即第名后缀与第名后缀的最长公共前缀的长度。认为。

for (i = 1, k = 0; i <= n; ++i) {  
 if (rk[i] == 0) continue;  
 if (k) --k;  
 while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;  
 height[rk[i]] = k;  
}

有了 height 数组，我们可以更好理解“子串”这一概念——**后缀的前缀**。

## 自动机

而非。

自动机是一个对信号序列进行判定的**数学模型**，并非算法或数据结构，实现其的方法可能有许多种。

「信号序列」是指一连串有顺序的信号，例如字符串从前到后的每一个字符、数组从 1 到 n 的每一个数、数从高到低的每一位等。「判定」是指针对某一个命题给出或真或假的回答。

自动机的结构是一张有向图，节点是单纯的状态。OI中的自动机一般指确定有限状态自动机（DFA）

### 子序列自动机

有次询问，判断是否为的子串。

pos[i]用于维护字符i在串中出现的所有位置，也就是说对于当前需要匹配的字符，只需要在对应的pos数组中查找即可。

下面的代码中，需要特别注意last\_p的初始值是-1，数组的下标从0开始。

for (int i = 0; i < n; ++i) {  
 a[i] = get\_int();  
 pos[a[i]].push\_back(i);  
}  
while (q--) {  
 bool flg = true;  
 int l = get\_int(), last\_p = -1; // 开始在没有字符的地方  
 for (int i = 0; i < l; ++i) {  
 int b = get\_int();  
 auto it = upper\_bound(pos[b].begin(), pos[b].end(), last\_p);  
 if (it == pos[b].end()) {  
 for (int j = i + 1; j < l; ++j) b = get\_int();  
 flg = false;  
 break;  
 }  
 last\_p = \*it;  
 }  
 printf("%s\n", flg? "Yes" : "No");  
}

### KMP

~~看毛片~~

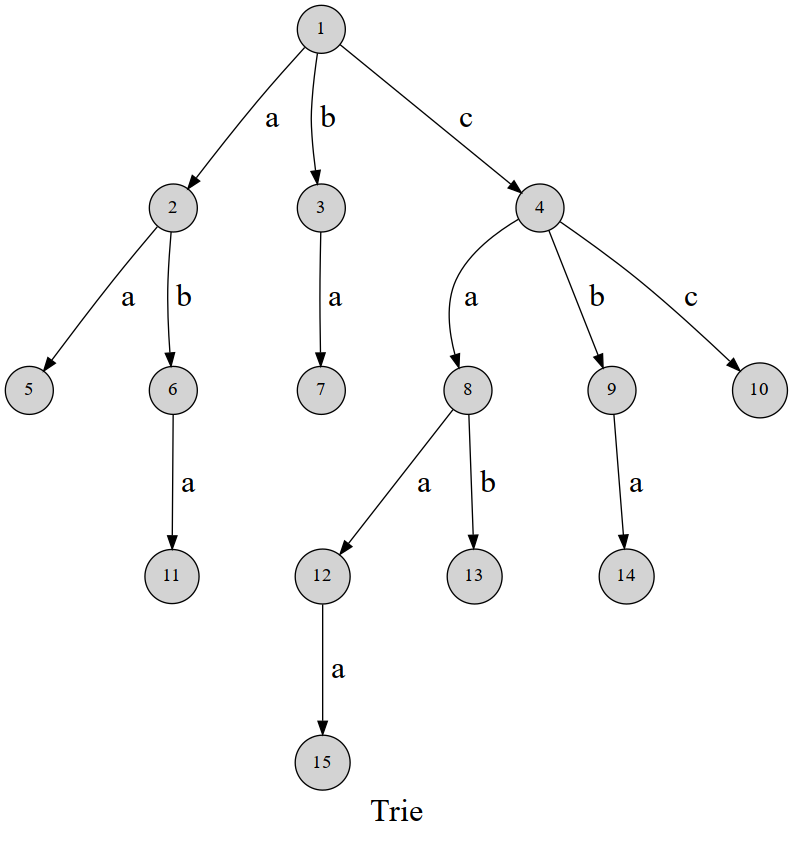
KMP可以视作自动机。

注释中已经标注了各个变量与数组的作用。

kmp[0] = 0; // 表示前i个字符的前缀中，真前缀与真后缀匹配的最大长度（并非本身）  
int prefix\_len = 0; // 当前前后缀匹配长度  
for (int i = 1; i < pattern.length(); ++i) {  
 while (prefix\_len != 0 && pattern[i] != pattern[prefix\_len]) {  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
 if (pattern[i] == pattern[prefix\_len]) {  
 ++prefix\_len;  
 }  
 kmp[i] = prefix\_len;  
}  
prefix\_len = 0;  
for (int i = 0; i < text.length(); ++i) {  
 while (prefix\_len != 0 && text[i] != pattern[prefix\_len]) {  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
 if (text[i] == pattern[prefix\_len]) {  
 ++prefix\_len;  
 }  
 if (prefix\_len == pattern.length()) {  
 cout << i - pattern.length() + 2 << '\n';  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
}

### 字典树(Trie)

字典树也可以视作是自动机，接受且只接受指定字符串集合中的元素。



Trie

从OI Wiki上扒下来的

struct Trie {  
 int nex[100000][26], cnt;  
 bool exist[100000]; // 该结点结尾的字符串是否存在  
 void insert(char \*s, int l) { // 插入字符串  
 int p = 0;  
 for (int i = 0; i < l; i++) {  
 int c = s[i] - 'a';  
 if (!nex[p][c]) nex[p][c] = ++cnt; // 如果没有，就添加结点  
 p = nex[p][c];  
 }  
 exist[p] = 1;  
 }  
 bool find(char \*s, int l) { // 查找字符串  
 int p = 0;  
 for (int i = 0; i < l; i++) {  
 int c = s[i] - 'a';  
 if (!nex[p][c]) return 0;  
 p = nex[p][c];  
 }  
 return exist[p];  
 }  
};

### AC自动机

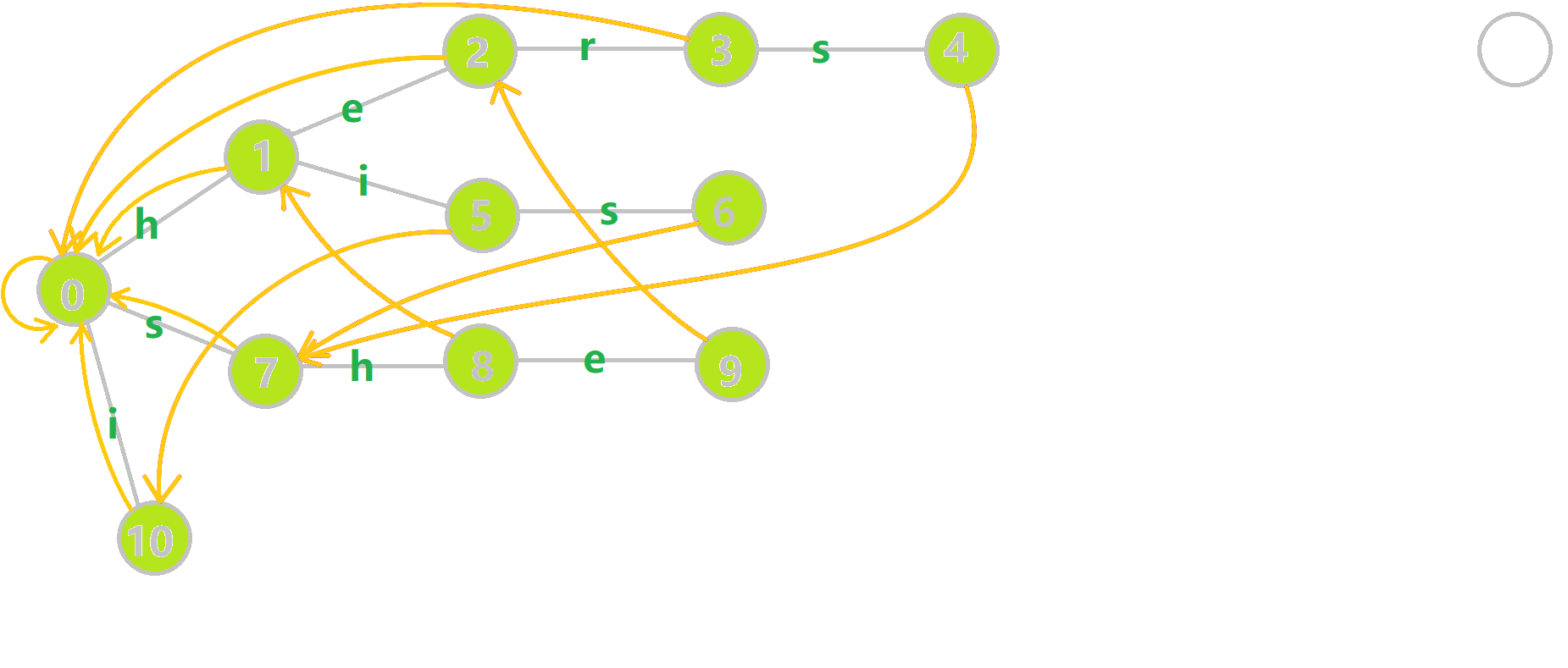
而非，用于实现多模式串匹配。

前置算法：KMP和Trie；实现方法：~~在树上看毛片~~。

两个概念：Trie和fail指针

1. 先将所有模式串丢到一个Trie里，这里用word[i]代表Trie上从根节点走到节点i所代表的字串。
2. 然后构建fail指针，其中fail[i] = j表明word[j]是word[i]的最长后缀，若指向根就表明找不到后缀；当前串失配之后跳转到fail指针，这样的话失配之后的前缀信息不会浪费，而是会利用已有的前缀信息继续匹配。

假设模式串分别是i、she、he、hers、his，Trie和fail指针就该是下面的样子



fail指针

~~fail指针告诉我们，不要当舔狗，此处不留爷，必有留爷处~~

构建AC自动机，需要使用BFS，先将第一层节点（0号的所有儿子）压入队列

if(tr[u][i])fail[tr[u][i]]=tr[fail[u]][i],q.push(tr[u][i]);  
else tr[u][i]=tr[fail[u]][i];

这里相当于做了路径压缩，可以地进行状态转移，重新构建了字典树，这些新加的边使字典树变成了**字典图**

假设模式串分别为r、er、her，文本串为herr，当前匹配到了文本串第4位（第二个r）。如果没有路径压缩就要按虚线（蓝色）跳转，若有路径压缩就可以按粗线（红线）跳转，这就是上面代码的else的作用；由此，也不需要用while来实现对fail的求解，只需要简单赋值即可

![字典图](data:application/octet-stream;base64,)

字典图

结合下面的query()能更好地理解，可以直接从字典图的边进行状态转移，而统计答案的时候需要逐步fail

* **简单版**（P3808）：求有多少个不同的模式串在文本串中出现过

简单之处在于，所有节点的val只需统计一遍即可，因为只要求得“出现过”而非次数

在主函数里先insert()模式串，再build()，最后query()文本串

int tr[MAXN][26], val[MAXN], fail[MAXN], cnt;  
// 字典树，节点模式串数量，fail指针，节点总数  
void insert(const char \*s) { // 构建Trie  
 size\_t len = strlen(s);  
 int now = 0;  
 for (int i = 0; i < len; ++i) {  
 int c = s[i] - 'a';  
 if (tr[now][c] == 0) {  
 tr[now][c] = ++cnt;  
 }  
 now = tr[now][c];  
 }  
 ++val[now]; // 记录当前的点权(该点有多少模式串)  
}  
void build() { // 也可以叫get\_fail  
 for (int i = 0; i < 26; ++i) { // 第一层节点压入队列  
 if (tr[0][i] != 0) {  
 fail[tr[0][i]] = 0;  
 que.push(tr[0][i]);  
 }  
 }  
 while(que.empty() == false) {  
 int u = que.front();  
 que.pop();  
 for (int i = 0; i < 26; ++i) {  
 if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 que.push(tr[u][i]);  
 } else {  
 tr[u][i] = tr[fail[u]][i];  
 }  
 }  
 }  
}  
int query(const char \*s) { // 查询  
 size\_t len = strlen(s);  
 int now = 0, ans = 0;  
 for (int i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 for (int j = now; j != 0 && val[j] != -1; j = fail[j]) {  
 ans += val[j];  
 val[j] = -1;  
 }  
 }  
 return ans;  
}

* **加强版**（P3796）：求哪些模式串（）出现的次数最多（）

对于上面的insert()和query()稍加修改即可，val[]不再保存出现的次数，而是保存哪个字符串以此结尾；修改相应的查询语句，并去掉val[j] = -1;即可

* **二次加强版**（P5357）：求模式串出现的次数（）

只保留所有的fail，可以构成一个由儿子指向父亲的树，那么每次不用暴力fail，而只统计当前节点，最后query之后再根据fail构成的图求前缀和即可。

并不用真的把图建出来，只用统计入度就行。

#include <iostream>  
#include <queue>  
#include <string>  
#include <vector>  
using namespace std;  
#define MAXN 2000007  
  
queue<int> que;  
vector<int> flag[MAXN];  
  
// Trie，节点存储的模式串编号，fail，答案，前缀和，入度  
int tr[MAXN][26], fail[MAXN]  
 , prefix\_sum[MAXN], ans[MAXN], in[MAXN], cnt;  
  
string pattern[MAXN]; // 模式串（编号从1开始，因为val为0表示没有）  
  
void insert(const string &s, int idx) { // 构建Trie  
 size\_t len = s.size();  
 int now = 0;  
 for (size\_t i = 0; i < len; ++i) {  
 int c = s[i] - 'a';  
 if (tr[now][c] == 0) {  
 tr[now][c] = ++cnt;  
 }  
 now = tr[now][c];  
 }  
 flag[now].push\_back(idx); // 可能会有重复的模式串  
}  
  
void build() { // 也可以叫get\_fail  
 for (int i = 0; i < 26; ++i) { // 第一层节点压入队列  
 if (tr[0][i] != 0) {  
 fail[tr[0][i]] = 0;  
 que.push(tr[0][i]);  
 }  
 }  
 while(que.empty() == false) {  
 int u = que.front();  
 que.pop();  
 for (int i = 0; i < 26; ++i) {  
 if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 ++in[fail[tr[u][i]]];  
 que.push(tr[u][i]);  
 } else {  
 tr[u][i] = tr[fail[u]][i];  
 }  
 }  
 }  
}  
  
void query(const string &s) { // 查询  
 size\_t len = s.size();  
 int now = 0;  
 for (size\_t i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 ++prefix\_sum[now]; // 这里改成前缀和  
 }  
}  
  
void topo() {  
 for (int i = 1; i <= cnt; ++i) {  
 if (in[i] == 0) {  
 que.push(i);  
 }  
 }  
 while (que.empty() == false) {  
 int now = que.front();   
 que.pop();  
 --in[fail[now]]; // 更新fail所指节点的入度  
 prefix\_sum[fail[now]] += prefix\_sum[now]; // 更新fail所指节点的前缀和  
 // ans[val[now]] = prefix\_sum[now];  
 for (size\_t i = 0; i < flag[now].size(); ++i) { // 统计答案  
 ans[flag[now][i]] = prefix\_sum[now];  
 }  
 if (in[fail[now]] == 0) {  
 que.push(fail[now]);  
 }  
 }  
}  
  
int n;  
string p;  
  
int main() {  
 #ifdef LOCAL  
 freopen("in.txt", "r", stdin);  
 freopen("out.txt", "w", stdout);  
 #endif // LOCAL  
 cin >> n;  
 for (int i = 1; i <= n; ++i) {  
 cin >> pattern[i];  
 insert(pattern[i], i);  
 }  
 build();  
 cin >> p;  
 query(p);  
 topo();  
 for (int i = 1; i <= n; ++i) {  
 cout << ans[i] << '\n';  
 }  
 cin >> p;  
 return 0;  
}

这一版代码没有封装到结构体里，好像没大用……

* **更进一步的优化**（CSP202009-5 密信与计数）

如果有很多次查询，每次只需要查询当前状态有无模式串，可以不必每次跳转fail，只需要build的时候做一些小改动，把fail指向的状态继承过来即可

if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 val[tr[u][i]] |= val[fail[tr[u][i]]];  
 que.push(tr[u][i]);  
}

那query就被改成了这个样子

int query(const string &s, int now) { // 查若存在返回-1  
 size\_t len = s.size();  
 for (size\_t i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 if (val[now] != 0) {  
 return -1; // 若存在返回-1  
 }  
 }  
 return now; // 若不存在返回结束时状态  
}

但是以上写法在这道题里会被卡常，**不能用string，有时候它很慢**；这道题只能边生成对应密文，边转移状态

### 后缀自动机（SAM）

直观上，字符串的SAM可以理解为给定字符串的所有子串的压缩形式，构造的时空复杂度均为，可以线性复杂度解决：

* 在另一个字符串中搜索一个字符串的所有出现位置
* 计算给定的字符串中有多少个不同的子串

（学不动了）

# 对拍

## checker

:loop  
rand.exe  
baoli.exe < data.txt > b.out  
std.exe < data.txt > c.out  
fc b.out c.out  
if not errorlevel 1 goto loop  
pause

造数据啥的自己 yy 就好了罢

当然，涉及到获取毫秒时间的还是有必要贴一下的，以便 srand()

#include <sys/time.h>  
#include <cstdlib>  
  
ll get\_ms() {  
 struct timeval current\_time;  
 gettimeofday(&current\_time, NULL);  
 ll ms = current\_time.tv\_sec \*   
 1000LL + current\_time.tv\_usec / 1000;  
 return ms;  
}

这也有一个cpp的随机数函数：

#include <iostream>  
#include <random>  
  
// 生成指定范围内的随机数  
size\_t generateRandomNumber(size\_t lower\_bound, size\_t upper\_bound) {  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<size\_t> dis(lower\_bound, upper\_bound);  
 return dis(gen);  
}  
  
int main() {  
 size\_t lower = 1; // 下界  
 size\_t upper = 1000000000000000000ULL; // 上界  
  
 // 生成随机数  
 size\_t random\_number = generateRandomNumber(lower, upper);  
  
 // 输出随机数  
 std::cout << "Generated random number: " << random\_number << std::endl;  
  
 return 0;  
}

## cin 关闭流同步

std::ios\_base::sync\_with\_stdio(false);

# 数学

~~玄之又玄~~

## 快速幂

ll fpow(ll x, ll y, ll p) {  
 ll ret = 1;  
 while (y > 0) {  
 if (y % 2 == 1) {  
 ret = ret \* x % p;  
 }  
 x = x \* x % p;  
 y /= 2;  
 }  
 return ret;  
}

上面代码中，ans用于存储答案，b表示m的2^x次方

## 分解质因数

先提供一种朴素做法：

vector<int> breakdown(int N) {  
 vector<int> result;  
 for (int i = 2; i \* i <= N; i++) {  
 if (N % i == 0) { // 如果 i 能够整除 N，说明 i 为 N 的一个质因子。  
 while (N % i == 0) N /= i;  
 result.push\_back(i);  
 }  
 }  
 if (N != 1) { // 说明再经过操作之后 N 留下了一个素数  
 result.push\_back(N);  
 }  
 return result;  
}

## 欧拉筛（线性筛）

基本的思想是筛到 i % prime[j] == 0 就不筛了，保证每个数只被筛一次。

struct Euler\_sieve {  
 vector<int> pri;  
 bool not\_prime[N];  
  
 void init(int k) {  
 for (int i = 2; i <= n; ++i) {  
 if (!not\_prime[i]) {  
 pri.push\_back(i);  
 }  
 for (int pri\_j : pri) {  
 if (i \* pri\_j > n) break;  
 not\_prime[i \* pri\_j] = true;  
 if (i % pri\_j == 0) break;  
 // i % pri\_j == 0  
 // 换言之，i 之前被 pri\_j 筛过了  
 // 由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被 pri\_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break 掉就好了  
 }  
 }  
 }  
   
 void is\_prime(int x) {  
 return !not\_prime[x];  
 }  
};

### 欧拉函数 $ $

$ (n) $ 表示小于等于 $ n $ 和 $ n $ 互质的数的个数，譬如 $ (1) = 1 $。

下面列举一些性质

* 欧拉函数是积性函数，但不是完全的积性函数。具体而言，若 $ (a, b) = 1 $，那么 $ (a b) = (a) (b) $。特别地，当 $ n $ 是奇数时，有 $ (2n) = 2 (n) $。
* $ n = \_{d n} (d) $。
* 由定义，若 $ n = p^k $，其中 $ p $ 是质数，那么 $ (n) = p^k - p^{k - 1} $。
* 设 $ n $ 的质因数分解为 $ n = *{i = 1}^{s} p*{i}^{k\_i} $，其中 $ p\_i $ 是质数，则有 $ (n) = n \_{i = 1}^s $。

由于第四条性质，显然如果只求一个数的欧拉函数，那可以在分解质因数的同时求解。

int euler\_phi(int n) {  
 int ans = n;  
 for (int i = 2; i \* i <= n; i++)  
 if (n % i == 0) {  
 ans = ans / i \* (i - 1);  
 while (n % i == 0) n /= i;  
 }  
 if (n > 1) ans = ans / n \* (n - 1);  
 return ans;  
}

* **线性筛求欧拉函数**

### 线性筛之约数个数

## 数论

### GCD & LCM

记得开long long

long long gcd(long long a, long long b) {  
 if (b == 0) return a;  
 else return gcd(b, a % b);  
}

### 同余理论和乘法逆元

以上不仅适用于加法，亦适用于减法、乘法。

* 模意义下的除法（**乘法逆元**）

若存在，使得，则称是在对取模意义下的乘法逆元。也就是说相当于在取模意义下，除以就相当于乘以。

拓展欧几里得求法：

long long ExGCD(long long a, long long b, long long &x, long long &y) {  
 if(b == 0) {  
 x = 1, y = 0;  
 return a;  
 }  
 long long d = ExGCD(b, a % b, x, y), t = x;  
 x = y, y = t - a / b \* x;  
 return d;   
}  
  
int ExGcdInv(int a, int b) { // a在模b意义下的逆元  
 int x, y;  
 ExGCD(a, b, x, y);  
 return x;  
}

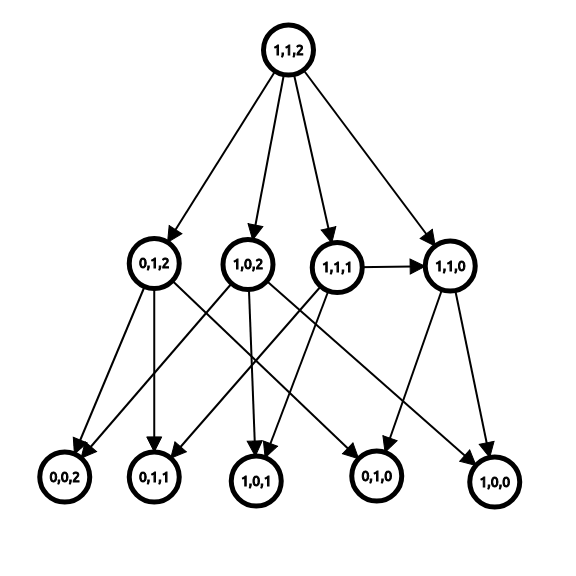
费马小定理求法（使用模数是大质数的情况）：

int FermatInv(int a, int b) {  
 return PowMod(a, b - 2, b); // 快速幂  
}

## 博弈论

一般的公平组合游戏以玩家无法行动为结束；每个状态能做出的操作与玩家无关；同一状态无法多次到达。

若把状态视作点，边视作转移，那公平组合游戏可以看成一个 DAG：



DAG

先定义必胜状态和必败状态：

1. 没有后继的状态是必败状态；
2. 至少有一个后继的状态是必胜状态；
3. 后继全是必败的状态是必败状态。

* Nim 游戏

堆物品，每一堆物品有个，两个玩家轮流取走任意一堆的任意个物品，但不能不取。取走最后一个物品的人获胜。

**Nim 和**，当且仅当 Nim 和为0时该状态为必败状态。

#include <cstdio>  
#include <iostream>  
using namespace std;  
  
int main() {  
 int t; scanf("%d", &t);  
 while (t--) {  
 int n, x, s = 0; scanf("%d", &n);  
 while (n--) {  
 scanf("%d", &x); s ^= x;  
 }  
 if (s) printf("Yes\n"); else printf("No\n");  
 }  
 return 0;  
}

### SG 定理

定义 函数表示集合中没有出现的最小的元素：

给定DAG 中节点 和它的所有**直接相连**后继 ，可以定义 函数如下：

表示与 节点**直接相连**的所有节点后继的 值中没有出现的最小的，规定**没有后继的必败状态节点** 。

对于有多个有向图组成的组合游戏，它们的起点分别为 ，则有定力：**当且仅当** **时，这个游戏是先手必败的。**

这一定理被称作 Sprague-Grundy 定理，简称 SG 定理。

SG可以递归地求。——林ye

# 数据结构

## 并查集

就是构建n棵树表示n个集合，两个点在同一集合内当且仅当两点的祖先相同。

* 构建并查集

让每个点的祖先等于自己，于是每个集合只包括一个点。

struct Node {  
 int fa, n; //保存自己的父亲，自己有多少个孩子  
}node[MAXN];  
//...  
for (int i = 0; i < n; ++i) node[i].fa = i, node[i].n = 1;

* 查询祖先（路径压缩）

这里要用到一个重要的优化——路径压缩，即让通往祖先的路径上的每一个节点都直接指向祖先。这里介绍递归法的实现方法。

int find(int x) {  
 int ret;  
 if (node[x].fa == x) ret = x;  
 else ret = find(node[x].fa);  
 return node[x].fa = ret; //路径压缩，直接让x指向自己的祖先  
}

使用路径压缩优化，复杂度期望值为O(α(N))，最坏情况下为O(logN)。对于Kruskal算法求解最小生成树，单独使用路径压缩已经足够了，因为Kruskal算法要对边进行排序，复杂度为O(logM)。

* 判断两点是否在一个集合

bool same(int x, int y) {return find(x) == find(y); }

* 合并（按秩合并）

按秩合并的基本思想是将深度小的树的祖先的父亲变更为深度大的树的祖先。这里为了实现简便，把将高度替换为孩子个数。（其实是我不会）

void merge(int x, int y) {  
 int xf = find(x), yf = find(y);  
 if (node[xf].n > node[yf].n) swap(xf, yf); //要把x并到y里面去，所以x小  
 if (xf != yf) {  
 node[xf].fa = yf;  
 node[yf].n += node[xf].n;  
 //只有根节点的n是有意义的，别的地方错了也无所谓  
 }  
}

运用了按秩合并和路径压缩后并查集的复杂度非常优秀，可以认为是O(1)的。

### 并查集维护序列的连通性

**洛谷 P2391 白雪皑皑**：一个常用的套路是，用 $ fa\_x $ 表示 $ x $ 后第一个可以操作的点。

下面的代码可以顺序遍历 $ [l, r] $ 中所有没被修改的点，并将其修改。有一点需要注意的是，初始化时务必 fa[n + 1] = n + 1，因为 n 要合并到 n + 1。

while (l <= r) {  
 int f = find\_fa(l);  
 if (l == f) {  
 ans[l] = i;  
 fa[l] = l + 1;  
 }  
 l = f;  
}

更进一步，**洛谷 P4145 上帝造题的七分钟 2 / 花神游历各国**：需要实现区间开放，并且要维护区间和。

思考一件事，一个数开方到 $ 1 $ 的次数是有限的，并且 $ 1 $ 和 $ 0 $ 开方不会影响区间和，于是可以树状数组+并查集暴力维护， $ fa\_x$ 维护 $ x $ 后第一个 $ > 1 $ 的位置。

while (l <= r) {  
 ll sq = sqrt(a[l]);  
 tr.add(l, sq - a[l]);  
 a[l] = sq;  
  
 if (a[l] <= 1) {  
 fa[l] = l + 1;  
 l = find\_fa(fa[l]);  
 } else {  
 l = l + 1;  
 }  
}

综上所述，第一份代码给出了**只修改一次**的版本，第二份代码给出了**修改多次**的版本。

## 树状数组

struct binary\_indexed\_tree{  
 int n;  
 ll c[N] = {0};  
 int lowbit(int x) {  
 // x 的二进制中，最低位的 1 以及后面所有 0 组成的数。  
 // lowbit(0b01011000) == 0b00001000  
 // ~~~~^~~~  
 // lowbit(0b01110010) == 0b00000010  
 // ~~~~~~^~  
 return x & -x;  
 }  
  
 ll get\_sum(int x) { // a[1]..a[x]的和  
 ll ans = 0;  
 while (x > 0) {  
 ans = ans + c[x];  
 x = x - lowbit(x);  
 }  
 return ans;  
 }  
  
 void add(int x, ll k) { // 在x的下标加上数值k  
 while (x <= n) { // 不能越界  
 c[x] = c[x] + k;  
 x = x + lowbit(x);  
 }  
 }  
}tr;

本质上维护的是前缀和，实现单点修改区间（前缀和）查询。

当然也可以看做维护的是差分数组，修改首尾，再求前缀和，实现单点查询区间修改。

当涉及到带取模的运算时，树状数组或许无法胜任。

## 线段树

* 几个操作以及一些定义

#define ls(xx) ((xx) << 1) + 1  
#define rs(xx) ((xx) << 1) + 2 //根节点编号从0开始  
  
long long tree[666666], tag\_add[666666], tag\_mul[666666], a[666666];  
  
void build(int l, int r, int node);  
void update\_add(int l, int r, int node, int nl, int nr, long long val, int p);  
void update\_mul(int l, int r, int node, int nl, int nr, long long val, int p);  
long long query(int l, int r, int node, int nl, int nr, int p);

* push\_up以及push\_down

void push\_up(int node) { //向上更新节点  
 tree[node] = tree[ls(node)] + tree[rs(node)];  
}  
  
//更新标记  
void fill(int l, int r, int node, long long add, long long mul, int p) {  
 tree[node] \*= mul; tree[node] %= p; tree[node] += (r - l + 1) \* add; tree[node] %= p;  
 tag\_add[node] \*= mul; tag\_add[node] %= p; tag\_add[node] += add; tag\_add[node] %= p;   
 tag\_mul[node] \*= mul; tag\_mul[node] %= p;  
}  
  
//向下释放标记  
void push\_down(int l, int r, int node, int p) {  
 int mid = (l + r) >> 1;  
 fill(l, mid, ls(node), tag\_add[node], tag\_mul[node], p);  
 fill(mid + 1, r, rs(node), tag\_add[node], tag\_mul[node], p);  
 tag\_add[node] = 0; tag\_mul[node] = 1;  
}

* 建树（build）

void build(int l, int r, int node) {  
 int mid = (l + r) >> 1;  
 tag\_mul[node] = 1;  
 if (l == r) {  
 tree[node] = a[l]; return ;  
 }  
 build(l, mid, ls(node));  
 build(mid + 1, r, rs(node));  
 push\_up(node);  
}

建树其实就是把每个节点的值存成区间和，如果该节点区间长度是1，那么区间和就是数组的值。

* 区间加（update\_add）

void update\_add(int l, int r, int node, int nl, int nr, long long val, int p) {  
 int mid = (l + r) >> 1;  
 if (nl <= l && r <= nr) {  
 tree[node] += (r - l + 1) \* val; tree[node] %= p;  
 tag\_add[node] += val; tag\_add[node] %= p; return ;  
 }  
 push\_down(l, r, node, p);  
 if (nl <= mid) update\_add(l, mid, ls(node), nl, nr, val, p);  
 if (mid < nr) update\_add(mid + 1, r, rs(node), nl, nr, val, p);  
 push\_up(node); tree[node] %= p;  
}

* 区间乘（update\_mul）

void update\_mul(int l, int r, int node, int nl, int nr, long long val, int p) {  
 int mid = (l + r) >> 1;  
 if (nl <= l && r <= nr) {  
 tree[node] \*= val; tree[node] %= p;  
 tag\_add[node] \*= val; tag\_add[node] %= p;  
 tag\_mul[node] \*= val; tag\_mul[node] %= p;  
 return ;  
 }  
 push\_down(l, r, node, p);  
 if (nl <= mid) update\_mul(l, mid, ls(node), nl, nr, val, p);  
 if (mid < nr) update\_mul(mid + 1, r, rs(node), nl, nr, val, p);  
 push\_up(node); tree[node] %= p;  
}

区间加和区间乘的操作是类似的，基本可以分为几个步骤：

1. 如果操作区间完全包含了当前区间，直接打上tag并对区间值进行修改。需要注意的是，乘法操作需要修改加法tag；
2. 下放标记（push\_down）；
3. 分别判断操作区间和当前区间的左右部分是否有交集，若有，递归操作；
4. 更新节点（push\_up）。

* 查询

long long query(int l, int r, int node, int nl, int nr, int p) {  
 int mid = (l + r) >> 1;  
 int ret = 0;  
 if (nl <= l && r <= nr) return tree[node];  
 push\_down(l, r, node, p);  
 if (nl <= mid) ret += query(l, mid, ls(node), nl, nr, p), ret %= p;  
 if (mid < nr) ret += query(mid + 1, r, rs(node), nl, nr, p), ret %= p;  
 return ret;  
}

对于线段树的操作，有很多细节。下面针对push\_down和push\_up两个操作说一些有共性的地方。

1. 如果要从某个节点访问其左右儿子，之前一定要下放标记；
2. 如果更新了左右儿子的节点值，一定要push\_up；
3. 不要忘记取模！

* 应用：求逆序对（常数太大，用树状数组好一些）

思路：先对于输入的数离散化，需要特殊处理值相等的情况。如果是严格的逆序对的话，那相等的值不会产生逆序对，下面给出的代码就是这个版本，同时也给出了**只有加法的线段树**。

一定要注意**add**和**query**的参数。

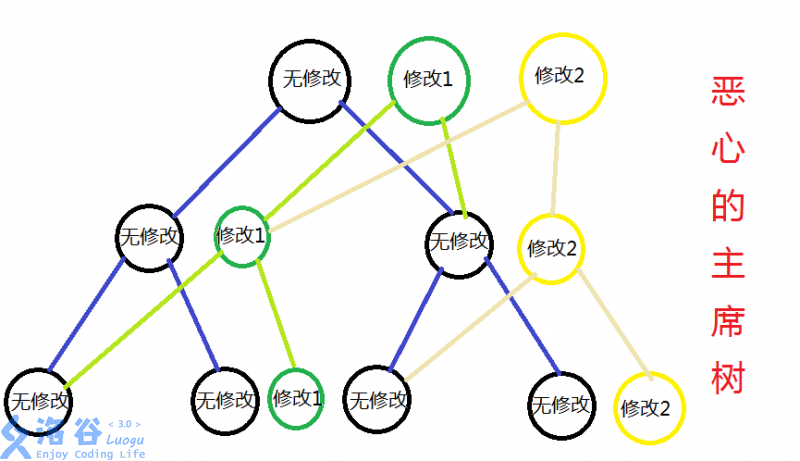
#include <iostream>  
#include <unordered\_map>  
#include <set>  
using namespace std;  
#define MAXN 666666  
#define P 19260817  
#define ls(x) ((x)<<1)  
#define rs(x) (((x)<<1)+1)  
#define mod(x) ((x))  
#define len(l, r) ((r)-(l)+1)  
  
unordered\_map<int, int> mp;  
set<int> s;  
int a[MAXN], aa[MAXN];  
long long su[MAXN\*2], tag[MAXN\*2];  
  
void push\_up(int now) {su[now] = mod(su[ls(now)] + su[rs(now)]); }  
  
void full(int now, int nl, int nr, long long k) {  
 tag[now] = mod(tag[now] + k);  
 su[now] = mod(su[now] + len(nl, nr) \* k);  
}  
  
void push\_down(int now, int nl, int nr) {  
 int mid = (nl + nr) >> 1;  
 full(ls(now), nl, mid, tag[now]);  
 full(rs(now), mid + 1, nr, tag[now]);  
 tag[now] = 0;  
}  
/\* 不用build了  
void build(int now, int nl, int nr) {  
 int mid = (nl + nr) >> 1;  
 if (nl == nr) {su[now] = a[nl]; return; }  
 build(ls(now), nl, mid); build(rs(now), mid + 1, nr);  
 push\_up(now);  
}  
\*/  
void add(int now, int nl, int nr, int l, int r, long long k) {  
 int mid = (nl + nr) >> 1;  
 if (l <= nl && nr <= r) { //完全包含  
 su[now] = mod(su[now] + len(nl, nr) \* k);  
 tag[now] = mod(tag[now] + k); return;  
 }  
 push\_down(now, nl, nr);  
 if (l <= mid) add(ls(now), nl, mid, l, r, k);  
 if (mid + 1 <= r) add(rs(now), mid + 1, nr, l, r, k);  
 push\_up(now);  
}  
  
long long query(int now, int nl, int nr, int l, int r) {  
 int mid = (nl + nr) >> 1; long long ret = 0;  
 if (l <= nl && nr <= r) return su[now];  
 push\_down(now, nl, nr);  
 if (l <= mid) ret = mod(ret + query(ls(now), nl, mid, l, r));  
 if (mid + 1 <= r) ret = mod(ret + query(rs(now), mid + 1, nr, l, r));  
 return ret;  
}  
  
int get\_int(void) {  
 // 快读就省略了  
}  
  
int main() {  
 #ifdef LOCAL  
 freopen("in.txt", "r", stdin);  
 freopen("out.txt", "w", stdout);  
 #endif // LOCAL  
 long long n, ans = 0;  
 // cin >> n;  
 n = get\_int();  
 for (int i = 0; i < n; ++i) {  
 a[i] = get\_int();  
 s.insert(a[i]);  
 }  
 int cnt = 1;  
 for (auto it = s.begin(); it != s.end(); ++it) {  
 mp[\*it] = cnt++; // 离散化，相同的数值离散化成同一值  
 }  
 for (int i = 0; i < n; ++i) {  
 int t = mp[a[i]];  
 ans += query(1, 1, n + 1, t + 1, n + 1);  
 add(1, 1, n + 1, t, t, 1); // 多维护一位  
 }  
 cout << ans;  
 return 0;  
}

由于只支持加法的线段树更加常用，在这里再贴个板子：

struct segment\_tree{  
 #define ls(x) ((x)<<1)  
 #define rs(x) (((x)<<1)+1)  
 #define len(l, r) ((r)-(l)+1)  
 #define mod(x, p) (((x)%(p)+(p))%(p))  
   
 ll su[MAXN\*2], tag[MAXN\*2], a[MAXN\*2];  
  
 void push\_up(int now, int p) {su[now] = mod(su[ls(now)] + su[rs(now)], p); }  
  
 void full(int now, int nl, int nr, ll k, int p) {  
 tag[now] = mod(tag[now] + k, p);  
 su[now] = mod(su[now] + len(nl, nr) \* k, p);  
 }  
  
 void push\_down(int now, int nl, int nr, int p) {  
 int mid = (nl + nr) >> 1;  
 full(ls(now), nl, mid, tag[now], p);  
 full(rs(now), mid + 1, nr, tag[now], p);  
 tag[now] = 0;  
 }  
  
 void build(int now, int nl, int nr, int p) {  
 int mid = (nl + nr) >> 1;  
 if (nl == nr) {su[now] = a[nl]; return; }  
 build(ls(now), nl, mid, p); build(rs(now), mid + 1, nr, p);  
 push\_up(now, p);  
 }  
  
 void add(int now, int nl, int nr, int l, int r, ll k, int p) {  
 int mid = (nl + nr) >> 1;  
 if (l <= nl && nr <= r) { //完全包含  
 su[now] = mod(su[now] + len(nl, nr) \* k, p);  
 tag[now] = mod(tag[now] + k, p); return;  
 }  
 push\_down(now, nl, nr, p);  
 if (l <= mid) add(ls(now), nl, mid, l, r, k, p);  
 if (mid + 1 <= r) add(rs(now), mid + 1, nr, l, r, k, p);  
 push\_up(now, p);  
 }  
  
 ll query(int now, int nl, int nr, int l, int r, int p) {  
 int mid = (nl + nr) >> 1; ll ret = 0;  
 if (l <= nl && nr <= r) return su[now];  
 push\_down(now, nl, nr, p);  
 if (l <= mid) ret = mod(ret + query(ls(now), nl, mid, l, r, p), p);  
 if (mid + 1 <= r) ret = mod(ret + query(rs(now), mid + 1, nr, l, r, p), p);  
 return ret;  
 }  
};

## 可持久化线段树

**主席树**又称可持久化权值线段树。



可持久化线段树

### 可持久化数组

支持：

* 查询某一历史版本单点
* 修改某一历史版本单点

所有的修改和查询都会新建一个版本。

struct HJT\_tree {  
 int n, tot = 0, ver\_cnt = 0, root[N<<5];  
 struct Node{int ls, rs, val; }node[N<<5];  
  
 int create\_node(int now) {  
 node[++tot] = node[now]; // 传递信息  
 return tot;  
 }  
  
 void init(int nn) {  
 n = nn;  
 root[0] = build(1, 1, n);  
 }  
  
 int build(int now, int nl, int nr) {  
 now = ++tot;  
 if (nl == nr) {  
 node[now].val = a[nl];  
 return tot;  
 }  
 int mid = (nl + nr) / 2;  
 node[now].ls = build(now, nl, mid);  
 node[now].rs = build(now, mid + 1, nr);  
 return now;  
 }  
  
 int query(int pos, int ver) {  
 root[++ver\_cnt] = root[ver]; // 新建版本  
 return query\_work(root[ver], 1, n, pos);  
 }  
  
 int query\_work(int now, int nl, int nr, int pos) {  
 if (nl == nr) {  
 return node[now].val;  
 }  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 return query\_work(node[now].ls, nl, mid, pos);  
 } else {  
 return query\_work(node[now].rs, mid + 1, nr, pos);  
 }  
 }  
  
 void update(int pos, int val, int ver) {  
 root[++ver\_cnt] = update\_work(root[ver], 1, n, pos, val); // 新建版本  
 }  
  
 int update\_work(int now, int nl, int nr, int pos, int val) {  
 now = create\_node(now);  
 if (nl == nr) {  
 node[now].val = val;  
 } else {  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 node[now].ls = update\_work(node[now].ls, nl, mid, pos, val);  
 } else {  
 node[now].rs = update\_work(node[now].rs, mid + 1, nr, pos, val);  
 }  
 }  
 return now;  
 }  
}tr;

### 静态区间第 k 小

对于原序列的每一个前缀建立出一棵线段树维护值域上每个数出现的次数，则其树是可减的。——HJT

对区间 $ [1, n] $ 查找第 $ k $ 小是容易的，在线段树上只需要看左右子树权值和即可判断答案在哪边。

考虑使用主席树。对于每次对于 $ a\_i $ 的插入操作，我们都可以保存相应的版本，那区间 $ [x, y] $ 对应的信息可以由 $ver\_y - ver\_{x - 1} $ 得到，于是就可以用主席树维护了。

struct Kth\_tree {  
 int n, tot = 0, root[N];  
 struct Node{int ls, rs, sum; }node[N<<5];  
  
public:  
 void init(int nn) {  
 n = nn;  
  
 int cnt = 0;  
 set<int> s;  
 for (int i = 1; i <= n; ++i) {  
 s.insert(a[i]);  
 }  
 for (auto c : s) {  
 mp[c] = ++cnt;  
 origin[cnt] = c; // 离散化  
 }  
 // n = mp.size();  
  
 root[0] = 1;  
 build(1, 1, cnt);  
 for (int i = 1; i <= n; ++i) { // 更新版本  
 root[i] = update(root[i - 1], 1, cnt, mp[a[i]]);  
 }  
 n = mp.size();  
 }  
  
 int query(int x, int y, int k) { // k-th number on [x, y]  
 return origin[query\_work(root[x - 1], root[y], 1, n, k)];  
 }  
  
private:  
 int create\_node(int now) {  
 node[++tot] = node[now]; // 传递信息  
 return tot;  
 }  
  
 int build(int now, int nl, int nr) {  
 now = ++tot;  
 node[now].sum = 0;  
 if (nl == nr) {  
 return now;  
 }  
 int mid = (nl + nr) / 2;  
 node[now].ls = build(now, nl, mid);  
 node[now].rs = build(now, mid + 1, nr);  
 return now;  
 }  
  
 int update(int now, int nl, int nr, int pos) { // a[pos] <- a[pos] + 1  
 now = create\_node(now);  
 node[now].sum += 1;  
 if (nl < nr) {  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 node[now].ls = update(node[now].ls, nl, mid, pos);  
 } else {  
 node[now].rs = update(node[now].rs, mid + 1, nr, pos);  
 }  
 }  
 return now;  
 }  
  
 int query\_work(int x, int y, int nl, int nr, int k) {  
 if (nl == nr) {  
 return nl;  
 }  
 int t = node[node[y].ls].sum - node[node[x].ls].sum, mid = (nl + nr) / 2;  
 if (t >= k) {  
 return query\_work(node[x].ls, node[y].ls, nl, mid, k);  
 } else {  
 return query\_work(node[x].rs, node[y].rs, mid + 1, nr, k - t);  
 }  
 }  
}tr;

所有功能已经封装到结构体中，调用 privite: 里的相应方法即可。不要忘记查询之前先要 init。

### HH 的项链

题意：给定一个序列，有多次询问，每次询问一个区间内有多少种数字。

* 离线/树状数组

按照 $ r $ 的升序排列所有询问，可以通过下面的方法将 $ r $ 相同的询问转换为前缀和 get\_sum(r) - get\_sum(l - 1)。

我们可以 add(i, 1) 表示 $ i $ 处出现了一个新数字。由于要求前缀和，我们只关心 $ a\_i $ 最后出现的位置，可以用 vis 记录其上次出现的位置。若 $ a\_i $ 出现过，那么在修改之前，首先应 add(vis[a[i]], -1) 消除其之前的影响。

int n;  
cin >> n;  
for (int i = 1; i <= n; ++i) {  
 cin >> a[i];  
}  
  
int m;  
cin >> m;  
for (int i = 1; i <= m; ++i) {  
 int l, r;  
 cin >> l >> r;  
 q[r].push\_back({l, i});  
}  
  
tr.n = n;  
  
for (int i = 1; i <= n; ++i) { // r = i  
 if (vis[a[i]]) {  
 tr.add(vis[a[i]], -1);  
 }  
 tr.add(i, 1);  
 vis[a[i]] = i;  
 for (auto c : q[i]) {  
 int &l = c.first, &id = c.second;  
 ans[id] = tr.get\_sum(i) - tr.get\_sum(l - 1);  
 }  
}  
  
for (int i = 1; i <= m; ++i) {  
 cout << ans[i] << "\n";  
}

* 在线/主席树

显然，上面的修改操作可以用主席树维护，然后就可以在线地求解了。

## 单调队列

譬如最经典的滑动窗口问题：

简单来说，**如果有人比你小并且比你强，那你就没用了**

需要做两件事

1. 维护队首（保证间隔）
2. 从队尾插入元素并向前remove冗余元素（去掉老而菜）

订正我的一个误区，是需要构建一个队列的，并不是在原数组上进行双指针操作。

经典例题：滑动窗口

int n, k, head = 0, tail = 0; scanf("%d %d", &n, &k);  
for (int i = 0; i < n; ++i) scanf("%d", a + i);  
for (int type = 0; type < 2; ++type) {  
 memset(que, 0, sizeof(int)\*MAXN);  
 for (int i = 0; i < n; ++i) {  
 if (i - que[head] >= k) ++head;  
 while ((cmp(a[i], a[que[tail]], type) || a[i] == a[que[tail]])  
 && head <= tail) --tail;  
 que[++tail] = i;  
 if (i + 1 >= k) printf("%d ", a[que[head]]);  
 }  
 printf("\n");  
}

# 图论

## 树、二叉树

和自然界中的树不同，计算机中的树是上下颠倒的，即根在上，叶子在下。

**树的性质**：

1. 任意两点路径唯一
2. 任意非根节点，爹唯一

有关树的题目需着重关注其性质

### 二叉树的编号

对于编号从1开始的节点k，其左右儿子分别为2k、2k+1。

这样我们就可以用数组保存二叉树，但有时树的深度很深，但是很多节点没有满，这样存比较浪费空间。

### 存二叉树

struct node{  
 int n, l, r; //编号、左右儿子  
}a[233];

有时也需要保存父亲是谁。

### 二叉树的深度优先遍历

* 先序遍历 - 根、左儿子、右儿子
* 中序遍历 - 左儿子、根、右儿子
* 后序遍历 - 左儿子、右儿子、根

这三种都是深度优先遍历（Depth-First Search），总是往深处访问。

已知后序遍历、中序遍历，如何求这棵树？

——可以通过后序遍历找到这棵树的根，然后在中序遍历中找到左子树、右子树分别有哪些节点，然后就可在后序遍历中知道左右儿子，然后递归求解即可。

### 树的直径

两种求解方法都适用边权不一定为1的情况

* 两遍dfs

随便选取一起点，dfs到距离最远的点；再以这个距离最远的点为起点，dfs到距离最远的点，于是可以求出直径。

* dp（记忆化搜索）

其中为过的最长路径，和分别为以为端点的最长路和次长路，这样的话可以通过记忆化搜索来实现。

### 树上最近公共祖先（LCA）

去洛谷看板子题的时候竟然发现之前打过（好像还不是抄的题解），于是直接把洛谷的提交粘过来了（甚至第一遍AC是在2019-11-15 15:59，第一次提交是在2019-11-15 15:48）

基本思路是一遍dfs统计深度和，后者表示节点的往上的第个爹是谁，可以这样求

用人话解释一遍就是，我的第个爹，当然是第个爹的第个爹，很合理。不过一般实现程序的时候，i都会被用作循环变量，还请务必注意（下面的程序也是）

调用这个dfs时，可以dfs(root, 0)，相当于设置一个虚空节点，可以让所有超范围的fa数组的值都指向这个虚空节点

void dfs(int node, int fat) { // fat表示node的爹是谁  
 fa[node][0] = fat; dep[node] = dep[fat] + 1;  
 for (int i = 1; i <= 19; ++i)  
 fa[node][i] = fa[fa[node][i - 1]][i - 1]; // 我的爷爷是我的爹的爹  
 for (int i = 0; i < size[node]; ++i)  
 if (edge[node][i] != fat) dfs(edge[node][i], node);  
}

在预处理完毕之后，照着下面打就对了……

int lca(int a, int b) {  
 if (dep[a] < dep[b]) swap(a, b);  
 for (int i = 19; i >= 0; --i)  
 if (dep[fa[a][i]] >= dep[b]) a = fa[a][i]; // 先找到同层的点  
 if (a == b) return b; // 发现b是a的某个爹就return  
 for (int i = 19; i >= 0; --i)  
 if (fa[a][i] != fa[b][i])  
 a = fa[a][i], b = fa[b][i];  
 return fa[a][0]; // 到最后一定是找到了共同爹的俩儿子  
}

### 树的重心（质心）

定义：删除树的重心后，树的各个连通分量中，节点数最多的连通分量其节点数达到最小值

**性质**：

1. 树上所有的点到树的重心的距离之和是最短的，如果有多个重心，那么总距离相等。
2. 把两棵树通过一条边相连，新的树的重心在原来两棵树重心的连线上。
3. 一棵树添加或者删除一个节点，树的重心最多只移动一条边的位置。
4. 一棵树最多有两个重心，且相邻。

具体求解可以通过一遍dfs（记忆化搜索）来求，很好求的。

### 树上前缀和

表示节点到根的权值和

* 若是点权，路径上权值和为
* 若是边权，路径上权值和为

### 树上差分

树上差分之前需要求LCA，具体而言，有对于点的差分，也有对于边的差分

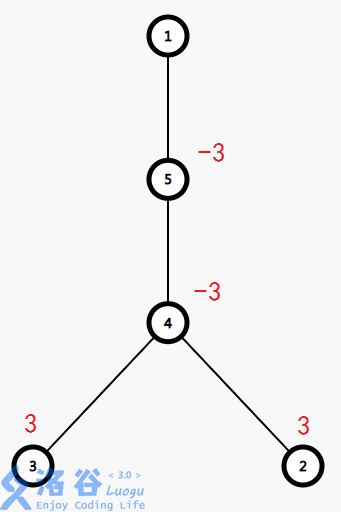
* 点的差分

eg：给定表示从走到，统计树上的点被经过的次数

若想将从到路径上的点权+x，可以这样维护差分数组：

diff[u] += x; diff[LCA(u, v)] -= x; // 可以看做维护从u到LCA的儿子  
diff[v] += x; diff[fa[LCA(u, v)][0]] -= x; // 可以看做维护从v到LCA

结合下面图片更好理解：



树上点的差分

这样的话，从这张图可以看出，实际求前缀和维护点权的时候是**从叶子向根节点更新的**

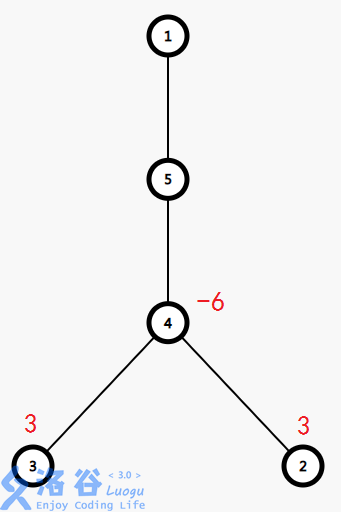
代码实现可以从根开始dfs，从儿子节点返回后累加diff数组。

void dfs\_solve(int now, int fat) {  
 tot[now] = diff[now];  
 for (int i = 0; i < to[now].size(); ++i) {  
 if (to[now][i] == fat) {  
 continue;  
 }  
 dfs\_solve(to[now][i], now); // 一定是递归调用自己  
 tot[now] += tot[to[now][i]];  
 }  
 ans = max(ans, tot[now]);  
}

注意这里的函数名，因为我的板子里求LCA也有一个叫dfs的函数，这个一定要区分好

* 边的差分

diff[u] += x; diff[v] += x;   
diff[LCA(u, v)] += x + x;



树上边的差分

只要dfs一遍，遍历时统计以每个节点为根的树的节点的权值和，就是当前节点到父亲节点的边的最终权值了

* eg：NOIP2015 运输计划

给你一个n个点的树，对于n-1条边各有边权。给出一些点对(x,y),同时定义d(x,y)表示x,y两点间的树上距离，现允许你将一条边的权值变为0，请你最小化最大的d值

思路：

两点路径一遍dfs预处理之后，求解，表示到根节点的路径和，那不难得出

再二分答案，即二分最大的dis。对于每次check，标记边权为这条边有多少条的路径经过（边的差分），并统计当前有多少条路径和长度大于。

check为true的条件是，存在一条边，全部d值超过mid的路径都经过这条边，并且把这条边的长度变成0之后，可以使得

## 图

不难证明（虽然我不会），连通的无环图就是树。

需要注意的是，图论问题并不都是直接以图的形式展示出来的，有时候需要选手根据题目自行建图。二元关系（两个元素之间的关系）都可以用图来描述，节点表示元素，边表示关系。

下面给出对于图的一些最基本的操作。

* 存图

int out[MAXN]; //存储每个点的出度  
struct node{int node; llint dis; }; //表示节点编号、单元最短路距离  
struct edge{int f, t, w; };  
vector<edge> v[MAXN];

* 加边

//加一条从f到t的边，其权值为w。这样加的边是单向的，若想加双向边加两次即可。  
void add(int f, int t, int w) {v[f].push\_back((edge){f, t, w}); ++out[f]; }

当然，我们可以用邻接矩阵，开一个二维数组dis[n][n]来表示边，不过在某些情况下这样可能很占空间。

### 求联通块

dfs即可，注意开一个mark数组标记某一点是否访问过

### BFS求最短路

对于边权为1的图，显然先遍历到的就是最近的。入队列时应把距离一并压入。

### 拓扑排序

适用于有向无环图（DAG）。值得注意的一点是，有向无环图并不是树，因为树的边是无向的。下面给出拓扑排序的步骤：

1. 统计所有入度为0的点，将其加入队列中。
2. 对于BFS到的某一个点，使其指向的点入度减一。如果某点入度被更新为0，将其加入队列中。

值得注意的是，如果结果中的节点个数不等于图的节点个数，说明该图存在环，无法进行拓扑排序。

### 欧拉回路

欧拉回路为走过图上所有边且每条边只经过一次的回路，但同一个节点可以走很多次。我们把存在欧拉回路的图成为欧拉图，可以形象地理解为能够“一笔画”的图。

对于无向图而言，若存在欧拉回路，则最多可以有两个奇数度数的节点，分别是起点和重点。

对于有向图，在忽略边的方向后，图必须是联通的；且最多只能有两个点入度和出度不相等，其中一个入度比出度多1，作为终点，另一个出度比入度多1，作为起点。

判断并打印欧拉回路只需要进行DFS即可，建一个vis[][]数组表示某条边是否被遍历过。

### 最小生成树（Kruskal算法）

即给定无向图，求边权和最小的生成树。

Kruskal算法实现步骤如下：

1. 将所有边按照权重升序排序；
2. 创建并查集；
3. 按照升序遍历所有边，如果不在一个集合中，则加入该条边；
4. 加入边数为(n - 1)时结束。

并查集相关请见“数据结构初步”。

### 单源最短路（Dijkstra算法）

~~你说SPFA？它死了。~~

Dijkstra适用于边权为正的情况。基本思路为：

下面的代码是高中的时候写的，建图存图和本篇方法相同。

//之前已经将小于号重定义成大于号  
memset(dis, 0x3f, sizeof dis); dis[s] = 0; q.push((node){s, 0});  
while (!q.empty()) {  
 int now; llint ndis; node tmp; tmp = q.top(); q.pop();  
 now = tmp.node; ndis = tmp.dis;  
 if (vis[now]) continue; vis[now] = true;  
 for (int i = 0; i < out[now]; ++i) {  
 int t = v[now][i].t, w = v[now][i].w  
 llint tdis = dis[t]; // dis要开long long  
 if (vis[t]) continue;  
 if (tdis > ndis + w) {  
 dis[t] = ndis + w; q.push((node){t, ndis + w});   
 }  
 }  
}

算法的基本实现方法如下：

1. 初始化dis数组，并创建vis数组标记某点的dis是否确定；
2. 创建优先队列q，并重定义小于号；
3. 把起始点的dis赋值为0，并将其加入q中；
4. 当q不为空时，重复取出队首执行以下步骤：
   * 如果vis过，那么continue，否则将队首的vis标记为true；
   * 遍历该点的每个出边，尝试更新目标点的dis；
   * 若没有vis且更新了dis，则将目标点加入队列中。

### SPFA

它死了。

实现基本与SPFA相似，不同的区别在于用普通队列维护需要进行松弛操作的点，用vis维护当前有哪些点在队列中（避免重复放入队列），对于取到的点，进行与Dijkstra相同的松弛操作。

### 多源最短路（Floyd-Warshall算法）

记住下面的代码，注意是**先枚举k**

memset(dis, 0x3f, sizeof(dis));  
for (int i = 0; i < n; ++i) dis[i][i] = 0;  
//中间省略赋值环节，即d[i][j]初始值是边的长度  
for (int k = 0; k < n; ++k)   
 for (int i = 0; i < n; ++i)  
 for (int j = 0; j < n; ++j)  
 d[i][j] = min(d[i][k] + d[j][k], d[i][j]);

### 图匹配

先给些定义：

1. 两两没有公共点的边组成的集合称为**匹配**
2. 所有匹配中边数最大的称为**最大匹配**
3. 图中带权时，边权和最大的匹配称为**最大权匹配**

* **二分图最大匹配**

（学不动了）