目录

[链式向前星 2](#_Toc11141)

[VECTOR 3](#_Toc3124)

[Queue 5](#_Toc19283)

[String 5](#_Toc27109)

[Set 6](#_Toc15222)

[bitset 7](#_Toc15368)

[子串个数 8](#_Toc18270)

[组合数 8](#_Toc2473)

[位运算 9](#_Toc13204)

[素数 11](#_Toc19037)

[欧拉函数 12](#_Toc21398)

[欧几里得算法 13](#_Toc28060)

[快速幂 14](#_Toc25381)

[进制转换 15](#_Toc18856)

[数学结论题 16](#_Toc19193)

[格雷码与二进制码 17](#_Toc18921)

[高精度整数四则运算 19](#_Toc8224)

[分数 21](#_Toc20823)

[乘法逆元 24](#_Toc23373)

[Trie 25](#_Toc30986)

[分割字符串 28](#_Toc21604)

[Manacher算法 29](#_Toc28532)

[KMP 29](#_Toc8573)

[并查集 31](#_Toc2354)

[树状数组 31](#_Toc8798)

[线段树 39](#_Toc11875)

[ST 表 49](#_Toc23193)

[平衡树 50](#_Toc29336)

[配对堆 54](#_Toc27750)

[子集生成和全排列 57](#_Toc12944)

[整数取整 63](#_Toc6791)

[日期处理 63](#_Toc29617)

[内部排序算法 64](#_Toc13235)

[划分算法 70](#_Toc21788)

[二维前缀和 70](#_Toc16579)

[二维差分 71](#_Toc29079)

[二分查找 72](#_Toc6510)

[区间覆盖问题 73](#_Toc29153)

[二叉树 76](#_Toc15314)

[多叉树 83](#_Toc23182)

[背包问题 84](#_Toc25948)

[线性动态规划 91](#_Toc14814)

[线性动态规划 91](#_Toc8137)

[Bellman-ford 94](#_Toc15657)

[Edmond-karp 95](#_Toc19286)

[最长上升子序列 96](#_Toc13017)

[并查集 时间复杂度O(n) 97](#_Toc14175)

[树状数组 时间复杂度O(ElogE) 99](#_Toc25868)

[Dijkstra，路径的花费不能有负 时间复杂度O(ElogE) 101](#_Toc23112)

[spfa算法求最短路径，允许负环 103](#_Toc13557)

[次短路径 105](#_Toc25086)

[最大公共祖先（LCA） 107](#_Toc16012)

**链式向前星**

**#include<bits/stdc++.h>**

**using namespace std;**

**const int maxn = 1005;//点数最大值**

**int n, m, cnt;//n个点，m条边**

**struct Edge**

**{**

**int to, w, next;//终点，边权，同起点的上一条边的编号**

**}edge[maxn];//边集**

**int head[maxn];//head[i],表示以i为起点的第一条边在边集数组的位置（编号）**

**void init()//初始化**

**{**

**for (int i = 0; i <= n; i++) head[i] = -1;**

**cnt = 0;**

**}**

**void add\_edge(int u, int v, int w)//加边，u起点，v终点，w边权**

**{**

**edge[cnt].to = v; //终点**

**edge[cnt].w = w; //权值**

**edge[cnt].next = head[u];//以u为起点上一条边的编号，也就是与这个边起点相同的上一条边的编号**

**head[u] = cnt++;//更新以u为起点上一条边的编号**

**}**

**int main()**

**{**

**cin >> n >> m;**

**int u, v, w;**

**init();//初始化**

**for (int i = 1; i <= m; i++)//输入m条边**

**{**

**cin >> u >> v >> w;**

**add\_edge(u, v, w);//加边**

**/\***

**加双向边**

**add\_edge(u, v, w);**

**add\_edge(v, u, w);**

**\*/**

**}**

**for (int i = 1; i <= n; i++)//n个起点**

**{**

**cout << i << endl;**

**for (int j = head[i]; j != -1; j = edge[j].next)//遍历以i为起点的边**

**{**

**cout << i << " " << edge[j].to << " " << edge[j].w << endl;**

**}**

**cout << endl;**

**}**

**return 0;**

**}**

**子串个数**

int find\_ab\_or\_ba(string origin , string find\_str){

int num = 0;

for(

int i = 0 ;

(i = origin.find(find\_str,i)) != std::string::npos ;

i++ , num ++

);

return num;

}

# **组合数**

## **求*Cnk*​**

gg C[MAX];void Cvalue(gg n){

C[0] = 1;

for (gg i = 1; i <= n; ++i){

C[i] = C[i - 1] \* (n - i + 1) % mod \* powMod(i, mod - 2, mod) % mod;

}}

## **求1~n内所有的C*Cnk*​的值**

gg C[MAX][MAX];void Cvalue(gg n){

for (gg i = 0; i <= n; ++i) {

C[i][0] = 1;

for (gg j = 1; j <= i; ++j){

C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % mod;

}

}}

# **位运算**

## **基础的位运算**

经常会将一个状态压缩成一个整数，这时掌握如何更改这个整数的二进制位是非常有必要的。

num |= 1 << n; //将num的第n位二进制位变为1

num &= ~(1 << n); //将num的第n位二进制位变为0

num ^= 1 << n; //将num的第n位二进制位翻转

可以这样枚举一个二进制数 n 的所有为 1 的二进制位的组合：

for (gg t = n; t != 0; t = (t - 1) & n){

//对状态t进行操作}

## **找出数组中唯一的元素**

### **问题描述**

给定一个整数数组，一个元素出现p次，其余元素出现k次。找到出现p次的元素（k>1,p>=1,p\ mod\ k\not ={0}*k*>1,*p*>=1,*p* *mod* *k*=0）。

### **算法**

设需要m个计数器*xm*​,*xm*−1​,⋯,*x*2​,*x*1​，m满足*m*>=*log*2​*k*。如果2^m>k则说明需要一个mask变量，mask的计算由k的二进制位决定，假设k=0b101，则mask=~(x3&~x2&x1)；假设k=0b1100，则mask=~(x4&x3&~x2&~x1)。最后的结果由p的二进制位决定，假设p=0b01，则结果是x1；假设p=0b10，则结果是x2；假设p=0b110，则结果是x3或者x2。

### **C++代码**

gg nums[MAX];

gg findNumber() {

gg xm = 0, ..., x2 = 0, x1 = 0, mask = 0;

for (gg i : nums) {

xm ^= x\_{m-1}... & x1 & i;

...

x2 ^= x1 & i;

x1 ^= i;

mask = ~(xm & ... & x2 & x1); //按要求计算mask

xm &= mask;

...

x2 &= mask;

x1 &= mask;

}

return xi; //按要求返回一个x变量}

## **使用异或、位与运算做加法**

gg plusANDXOR(gg a, gg b) {

while (b != 0) {

gg c = (unsigned gg)(a & b) << 1;

a ^= b;

b = c;

}

return a;}

# **素数**

## **用暴力方法判断 n 是否为素数**

算法的时间复杂度为O(\sqrt n)*O*(*n*​)

bool isPrime(gg n) {

if (n < 2) {

return false;

}

for (gg i = 2; i \* i <= n; ++i) {

if (n % i == 0) {

return false;

}

}

return true;}

## **利用埃氏筛法求解 [2,n) 以内的所有数字的一个质因子**

算法的时间复杂度为*O*(*nloglogn*)

// prime[i]表示i的一个质因子，若prime[i]==0，表示i是一个素数

gg prime[MAX];void getPrime(gg n) {

for (gg i = 2; i \* i <= n; ++i) {

if (prime[i] == 0) {

for (gg j = i + i; j < n; j += i) {

prime[j] = i;

}

}

}}

# **欧拉函数**

欧拉函数*φ*(*n*)的值是小于等于 n 且与 n 互质的正整数个数。注意，1 与任何正整数都互质，所以*φ*(1)=1。

## **欧拉函数的性质**

1. 欧拉函数是积性函数，即如果有gcd(a,b)=1，则*φ*(*ab*)=*φ*(*a*)⋅*φ*(*b*)。
2. n=∑*d*∣*n*​*φ*(*d*)

## **求 n 的欧拉函数值**

gg euler\_phi(gg n) {

gg m = (gg)sqrt(n + 0.5), ans = n;

for (gg i = 2; i <= m; 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;}

## **求 1~n 的欧拉函数值**

gg phi[MAX];void phi\_table(gg n) {

for (gg i = 2; i <= ni; i++) {

phi[i] = 0;

}

phi[1] = 1;

for (gg i = 2; i <= ni; i++) {

if (phi[i] == 0) {

for (gg j = i; j <= ni; j += i) {

if (phi[j] == 0)

phi[j] = j;

phi[j] = phi[j] / i \* (i - 1);

}

}

}}

# **欧几里得算法**

1. 欧几里得算法可以求解 a,b 两数的最大公约数，时间复杂度为O(log(max(a,b)))，递归实现一般不会栈溢出。用欧几里得算法去求斐波那契数列相邻两项的最大公约数，该算法会达到最坏复杂度。
2. 如果要求多个数的最大公约数（最小公倍数），当我们算出两个数的最大公约数（最小公倍数）之后，将它放入序列中对后面的数继续求解即可。
3. a,b,c 为任意整数，扩展欧几里得算法可以求解方程ax+by=gcd(a,b)的一组整数解。
   1. 调用扩展欧几里得算法时，需保证 a,b 为正整数。
   2. 若方程ax+by=c的一组整数解为(*x*0​,*y*0​)，，则它的任意整数解都可以写成(*x*0​+*kb*′,*y*0​−*ka*′)，其中a'=a/gcd(a,b)，b'=b/gcd(a,b)，k 取任意整数。
   3. 若g=gcd(a,b)，方程ax+by=g的一组解是(*x*0​,*y*0​)，则当 c 是 g 的倍数时，ax+by=c的一组解是(cx\_0/g, cy\_0/g)；当 c 不是 g 的倍数时无整数解。

## **通过欧几里得算法计算两个正整数的最大公约数**

gg gcd(gg a, gg b) { return b == 0 ? a : gcd(b, a % b); }

## **扩展欧几里得算法求解方程 ax+by=gcd(a,b)的一组整数解，并返回 gcd(a,b)的值**

//返回{gcd(a,b),x,y}，x,y是方程ax+by=gcd(a,b)的一组解

array<gg, 3> extendGcd(gg a, gg b) {

if (b == 0) {

return {a, 1, 0};

}

auto d = extendGcd(b, a % b);

return {d[0], d[2], d[1] - (a / b) \* d[2]};}//返回方程ax+by=c的解中x的最小非负整数值

gg getMinX(gg a, gg b, gg c) {

auto d = extendGcd(a, b);

return (c / d[0] \* d[1] % (b / d[0]) + b / d[0]) % (b / d[0]);}

# **快速幂**

## **快速幂**

gg binpow(gg a, gg b) {

gg ans = 1;

while (b > 0) {

if (b & 1)

ans = ans \* a;

a = a \* a;

b >>= 1;

}

return ans;}

## **快速幂取模**

//返回a^b%m

gg powMod(gg a, gg b, gg m) {

if (m == 1) {

return 0;

}

a %= m;

gg ans = 1;

while (b > 0) {

if (b & 1)

ans = ans \* a % m;

a = a \* a % m;

b >>= 1;

}

return ans;}

## **高精度快速幂**

//返回a^b%mod，b是高精度整数

gg superPow(gg a, string& b, gg mod) {

a %= mod;

if (b.empty())

return 1;

gg k = b.back();

b.pop\_back();

return powMod(superPow(a, b, mod), 10, mod) \* powMod(a, k, mod) % mod;}

# **进制转换**

## **用除基取余法将十进制数转换成 R 进制数**

算法的时间复杂度为O(log\_{R}n)*O*(*logR*​*n*)

// n 为要转换的十进制数，R 为要转换的进制//返回用 vector<gg>存储的转换成的 R 进制数

vector<gg> decToR(gg n, gg R) {

vector<gg> ans; //存储 R 进制数

do {

ans.push\_back(n % R); //取余

n /= R; //除基

} while (n != 0); // n==0 时跳出循环

reverse(begin(ans), end(ans)); //翻转整个数组

return ans;}

## **将 R 进制数转换成十进制数**

算法的时间复杂度为O(n)*O*(*n*)

// R 为转换的进制，r 为要存储在 vector<gg>中的转换的 R 进制数，n为r的元素个数//返回对应的十进制数

gg rToDec(const vector<gg>& r, gg R) {

gg d = 0;

for (gg i : r)

d = d \* R + i;

return d;}

# **数学结论题**

## **正整数拆分**

给定一个正整数 n，将其拆分为至少多个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

//如果最少分解成2个正整数

gg integerBreak2(gg n, gg mod) {

if (n <= 3) {

return n - 1;

}

gg a = n / 3, b = n % 3;

if (b == 0) {

return powMod(3, a, mod);

} else if (b == 1) {

return (powMod(3, a - 1, mod) \* 4) % mod;

} else {

return (powMod(3, a, mod) \* 2) % mod;

}}//如果最少可以分解成1个正整数

gg integerBreak1(gg n, gg mod) {

if (n == 1) {

return 1;

}

gg a = n / 3, b = n % 3;

if (b == 0) {

return powMod(3, a, mod);

} else if (b == 1) {

return (powMod(3, a - 1, mod) \* 4) % mod;

} else {

return (powMod(3, a, mod) \* 2) % mod;

}}

# **格雷码与二进制码**

在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码（Gray Code）。3 位格雷码如下：

| **十进制** | **二进制** | **格雷码** |
| --- | --- | --- |
| 0 | 000 | 000 |
| 1 | 001 | 001 |
| 2 | 010 | 011 |
| 3 | 011 | 010 |
| 4 | 100 | 110 |
| 5 | 101 | 111 |
| 6 | 110 | 101 |
| 7 | 111 | 100 |

设G、B为对应的格雷码与二进制码表示，从低位向高位由 0 开始编号，就像下面这样：

|  | **2** | **1** | **0** |
| --- | --- | --- | --- |
| G | 0 | 1 | 1 |
| B | 0 | 1 | 0 |

## **递归生成格雷码**

1. 1 位格雷码为 0；
2. n+1 位格雷码的集合 = n 位格雷码集合（顺序）加前缀 0 + n 位格雷码集合（逆序）加前缀 1。

## **n 位二进制码转 n 位格雷码**

## **C++代码**

gg getGrayFromBinary(gg n) { return n >> 1 ^ n; }

## **n 位格雷码转 n 位二进制码**

## **C++代码**

gg getBinaryFromGray(gg n) {

gg ans = 0;

while (n != 0) {

ans ^= n;

n /= 2;

}

return ans;}

# **高精度整数四则运算**

一般情况下，如果要用到高精度运算，最好使用 python 语言编写程序，因为 python 语言本身的整数类型就自带高精度运算。

## **高精度与高精度的加法**

//默认输入的a和b均为非负整数

string Plus(const string& a, const string& b) {

string ans;

gg carry = 0; //进位

for (gg i = a.size() - 1, j = b.size() - 1; i >= 0 || j >= 0 || carry != 0; --i, --j) {

gg p1 = i >= 0 ? a[i] - '0' : 0, p2 = j >= 0 ? b[j] - '0' : 0;

gg k = p1 + p2 + carry;

ans.push\_back(k % 10 + '0');

carry = k / 10;

}

reverse(begin(ans), end(ans)); //要进行翻转

return ans;}

## **高精度与高精度的减法**

//默认输入的a和b均为非负整数，且a>=b

string Sub(string a, const string& b) {

string ans;

for (gg i = a.size() - 1, j = b.size() - 1; i >= 0 || j >= 0; --i, --j) {

gg p1 = i >= 0 ? a[i] - '0' : 0, p2 = j >= 0 ? b[j] - '0' : 0;

if (p1 < p2) { //不够减，要借位

a[i - 1]--;

p1 += 10;

}

gg k = p1 - p2;

ans.push\_back(k % 10 + '0');

}

ans.erase(ans.find\_last\_not\_of('0') + 1, ans.size());

reverse(begin(ans), end(ans)); //要进行翻转

return ans.empty() ? "0" : ans;}

## **高精度与高精度的乘法**

//默认a和b均为非负整数

string Multiply(const string& a, const string& b) {

gg m = a.size(), n = b.size();

string ans(n + m, '0'); //最终乘积最多有n+m位

for (gg i = m - 1; i >= 0; --i) {

for (gg j = n - 1; j >= 0; --j) {

gg k = (a[i] - '0') \* (b[j] - '0');

gg t = k + (ans[i + j + 1] - '0');

ans[i + j] += t / 10; //向乘积的i+j位进位

ans[i + j + 1] = t % 10 + '0'; //填充乘积的i+j+1位

}

}

ans.erase(0, ans.find\_first\_not\_of("0")); //删除前导0

return ans.empty() ? "0" : ans;}

## **高精度与低精度的除法**

//默认a为非负整数，b为正整数

pair<string, gg> DivMod(const string& a, gg b) {

string ans; //商

gg mod = 0; //余数

for (char c : a) {

mod = c - '0' + mod \* 10;

ans.push\_back(mod / b + '0');

mod %= b;

}

ans.erase(0, ans.find\_first\_not\_of('0')); //删除多余的前导0

return {ans.empty() ? "0" : ans, mod};}

## **高精度与低精度的取模运算**

//默认a为非负整数，b为正整数，返回a%b

gg Mod(const string& a, gg b) {

gg ans = 0;

for (char c: a){

ans = ((ans \* 10 + c - '0') % m);

}

return ans;}

# **分数**

我们用假分数的形式表示一个分数，这种形式下，一个分数只包含分子和分母两个整数，而且它还可以表示大于 1 的分数。可以使用array<gg, 2>来表示这样一个假分数。由于array<gg, 2>类型名过长，我们可以使用为它取一个类型别名，即using F=array<gg, 2>（F 是分数的英文 fraction 的首字母）。为了表述和编码方便，如果我们有这样一个分数对象 f，我们针对这样一种表示方法约定以下规则：

1. 分数对象 f 应该是 F 类型，其中 f[0] 表示分子，f[1] 表示分母；
2. 分母 f[1] 永远都是正整数，如果 f 的值为负，那么让分子 f[0] 为负；
3. 分子 f[1] 和分母 f[1] 应该一直是互质的，即两者除了 1 以外没有其他公约数；
4. 如果 f 的值为 0，则令 f[0]=0，f[1]=1。

本模板给出的读入程序针对的是按 a/b 的格式给出分数形式，其中分子和分母全是整型范围内的整数，分母不为 0。

本模板输出的分数形式满足以下要求：

1. 如果分数的值是一个整数，只输出整数部分；
2. 如果分数的值大于 1，按带分数k a/b形式输出，k 为整数部分，a/b 为约分后的分数部分，如果有负号，只出现在整数部分；
3. 如果分数的值小于 1，按带分数a/b形式输出，a/b 为约分后的分数部分，如果有负号，只出现在分子前；
4. 如果分数的值为负，则会在分数前后输出圆括号()。

注意，进行分数除法时，你应该保证传入这个函数的除数不为 0。

using F = array<gg, 2>;//分数的输入，针对的是按 a/b 的格式给出分数形式，分母不为 0

F input() {

F f;

char c; //吸收'/'符号

cin >> f[0] >> c >> f[1];

return f;}//分数的化简void simplify(F& f) {

if (f[0] == 0) { //如果分子 f[0] 为 0，则令 f[1]=1

f[1] = 1;

return;

}

if (f[1] < 0) { //如果分母 f[1] 为负，将分子 f[0] 和分母 f[1] 都取相反数

f[1] = -f[1];

f[0] = -f[0];

}

gg d = gcd(abs(f[0]), abs(f[1])); //求出分子 f[0] 和分母 f[1] 绝对值的最大公约数

f[0] /= d;

f[1] /= d;}//分数的加法

F Plus(const F& f1, const F& f2) {

F f;

f[0] = f1[0] \* f2[1] + f2[0] \* f1[1];

f[1] = f1[1] \* f2[1];

simplify(f);

return f;}//分数的减法

F Sub(const F& f1, const F& f2) {

F f;

f[0] = f1[0] \* f2[1] - f2[0] \* f1[1];

f[1] = f1[1] \* f2[1];

simplify(f);

return f;}//分数的乘法

F Multiply(const F& f1, const F& f2) {

F f;

f[0] = f1[0] \* f2[0];

f[1] = f1[1] \* f2[1];

simplify(f);

return f;}//分数的除法

F Div(const F& f1, const F& f2) {

F f;

f[0] = f1[0] \* f2[1];

f[1] = f1[1] \* f2[0];

simplify(f);

return f;}//分数输出void output(const F& f) {

if (f[0] < 0)

cout << '(';

if (f[1] == 1) {

cout << f[0];

} else if (abs(f[0]) < f[1]) {

cout << f[0] << "/" << f[1];

} else

cout << f[0] / f[1] << " " << abs(f[0]) % f[1] << "/" << f[1];

if (f[0] < 0)

cout << ')';}

# **乘法逆元**

如果一个线性同余方程ax≡1(*mod* *n*)，则x称为a *mod* n的逆元，记作*a*−1。

## **扩展欧几里得算法求逆元**

若gcd(a,n)=1，则方程无解。所以如果方程有解的话，就可以通过[扩展欧几里得算法](https://file+.vscode-resource.vscode-cdn.net/c:/Users/hp/Desktop/code-templates/%E6%95%B0%E5%AD%A6/%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%97%E7%AE%97%E6%B3%95.md)求逆元。

## **快速幂法求逆元**

根据费马小定理，若 n为素数，且gcd(a,n)=1 ，a mod n的逆元可以计算为*an*−2，可通过[快速幂](https://file+.vscode-resource.vscode-cdn.net/c:/Users/hp/Desktop/code-templates/%E6%95%B0%E5%AD%A6/%E5%BF%AB%E9%80%9F%E5%B9%82.md)求解。

## **求1,2,⋯,*n*中每个数关于p的逆元**

算法时间复杂度为O(n)。

gg inv[MAX];void invEle(gg n, gg p) {

inv[1] = 1;

for (gg i = 2; i <= n; ++i) {

inv[i] = (p - p / i) \* inv[p % i] % p;

}}

## **求任意 n 个数关于p的逆元**

首先计算 n 个数的前缀积，记为*si*​，然后利用快速幂或扩展欧几里得算法计算*sn*​的逆元，记作*svn*​。因为 *svn*​ 是 n 个数的积的逆元，所以当我们把它乘上*an*​ 时，就会和 *an*​ 的逆元抵消，于是就得到了 *a*1​ 到 *an*​ 的积逆元，记为 *svn*−1​。可以依次计算出所有的 *svi*​ ，于是 *ai*−1​ 就可以用 *si*−1​⋅*svi*​ 求得。算法时间复杂度为O(n+logp)

gg inv[MAX], s[MAX], a[MAX], sv[MAX];void invEle(gg n, gg p) {

s[0] = 1;

for (gg i = 1; i <= n; ++i) {

s[i] = s[i - 1] \* a[i] % p;

}

//也可以用扩展欧几里得来求逆元,视个人喜好而定.

sv[n] = powMod(s[n], p - 2, p);

for (gg i = n; i >= 1; --i) {

sv[i - 1] = sv[i] \* a[i] % p;

}

for (gg i = 1; i <= n; ++i) {

inv[i] = sv[i] \* s[i - 1] % p;

}}

# **Trie**

## **字符串 Trie**

class Trie {

public:

Trie() : root(new TrieNode()) {}

//插入一个单词

void insert(const string& word) {

auto i = root;

for (char c : word) {

if (not i->children[c]) {

i->children[c] = new TrieNode();

}

i = i->children[c];

}

++i->count;

}

//查找一个单词

bool search(const string& word) {

auto i = root;

for (char c : word) {

if (not i->children[c]) {

return false;

}

i = i->children[c];

}

return i->count > 0; //如果是查找前缀把这条语句换成return true;就可以了

}

//删除一个单词

void remove(const string& word) { root = dfs(root, word, 0); }

private:

struct TrieNode {

gg count = 0; //该结点代表的单词个数,count>0表示这是一个单词结点

TrieNode\* children[128]{};

};

TrieNode\* root;

TrieNode\* dfs(TrieNode\* r, const string& word, gg p) {

if (p >= word.size()) {

return nullptr;

}

r->children[word[p]] = dfs(r->children[word[p]], word, p + 1);

if (all\_of(begin(r->children), end(r->children), [](TrieNode\* t) { return not t; })) {

return nullptr;

} else {

return r;

}

}};

## **01-Trie**

template <gg bits = 32>class Trie {

public:

Trie() : root(new TrieNode()) {}

//插入一个不超过bits位的非负整数

void insert(gg n) {

const bitset<bits>& bin(n);

auto i = root;

for (gg j = bits - 1; j >= 0; --j) {

if (not i->children[bin[j]]) {

i->children[bin[j]] = new TrieNode();

}

i = i->children[bin[j]];

}

}

void remove(gg n) {

const bitset<bits>& bin(n);

root = dfs(root, bin, bits - 1);

}

//与bin异或的最大值

gg search(gg n) {

const bitset<bits>& bin(n);

auto i = root;

gg ans = 0;

for (gg j = bits - 1; j >= 0; --j) {

gg k = bin[j] ? 0 : 1;

if (i->children[k]) {

ans = ans \* 2 + 1;

i = i->children[k];

} else {

ans <<= 1;

i = i->children[k ^ 1];

}

}

return ans;

}

private:

struct TrieNode {

TrieNode\* children[2]{};

};

TrieNode\* root;

TrieNode\* dfs(TrieNode\* r, const bitset<bits>& bin, int p) {

if (p < 0) {

return nullptr;

}

r->children[bin[p]] = dfs(r->children[bin[p]], bin, p - 1);

if (not r->children[0] and not r->children[1]) {

return nullptr;

} else {

return r;

}

}};

# **分割字符串**

vector<string> split(const string& s, string c = " ") {

vector<string> ans;

for (gg i = 0, j = 0; i < s.size(); i = j + 1) {

j = s.find(c, i);

if (j == string::npos)

j = s.size();

ans.push\_back(s.substr(i, j - i));

}

return ans;

}

# **Manacher算法**

求最大回文子串

string longestPalindrome(string& s) {

string news = "$#";

for (gg i = 0; i < s.size(); ++i)

news = news + s[i] + "#";

vector<gg> p(news.size(), 1);

gg x = 0, xr = 0;

for (gg i = 1; i < news.size(); ++i) {

if (i < xr)

p[i] = min(xr - i, p[2 \* x - i]);

while (i + p[i] < news.size() && news[i - p[i]] == news[i + p[i]])

++p[i];

if (xr < i + p[i]) {

x = i;

xr = i + p[i];

}

}

gg k = max\_element(begin(p) + 1, end(p)) - begin(p);

return s.substr((k - p[k]) / 2, p[k] - 1); //返回最长回文子串

}

# **KMP**

#include <bits/stdc++.h>

using namespace std;

using gg = long long;

constexpr gg MAX = 1e6 + 5;

constexpr gg mod = 1e9 + 7;

constexpr gg INF = 2e18;

constexpr double thre = 1e-7; //用以判断两个浮点数是否相等的阈值

gg ti, ni, mi, ki, di, pi, xi, yi;

void getNext(string pattern,gg\*next){

gg j=-1;

next[0]=-1;

for(gg i=1;i<pattern.size();++i){

while(j!=-1&&pattern[i]!=pattern[j+1])

j=next[j];

if(pattern[i]==pattern[j+1])

++j;

next[i]=j;

}

}

bool KMP(string text,string pattern){

gg next[pattern.size()];

getNextval(pattern,next);

for(gg i:next)

prLLf("%d ",i);

gg j=-1;

for(gg i=0;i<text.size();++i){

while(j!=-1&&text[i]!=pattern[j+1])

j=next[j];

if(text[i]==pattern[j+1])

++j;

if(j==pattern.size()-1)

return true;

}

return false;

}

void getNextval(string pattern,gg\*next){

gg j=-1;

next[0]=-1;

for(gg i=1;i<pattern.size();++i){

if(j!=-1&&pattern[i]!=pattern[j+1])

j=next[j];

if(pattern[i]==pattern[j+1])

++j;

if(j==-1||pattern[i+1]!=pattern[j+1])

next[i]=j;

else

next[i]=next[j];

}

}

gg KMPcount(string text,string pattern){

gg next[pattern.size()];

getNextval(pattern,next);

for(gg i:next)

prLLf("%d ",i);

gg j=-1,result=0;

for(gg i=0;i<text.size();++i){

while(j!=-1&&text[i]!=pattern[j+1])

j=next[j];

if(text[i]==pattern[j+1])

++j;

if(j==pattern.size()-1){

++result;

j=next[j];

}

}

return result;

}

**并查集**

class UFS {

public:

//初始化并查集

UFS(gg n) : ufs(n + 5) { iota(begin(ufs), end(ufs), 0); }

//查找结点所在树的根结点并进行路径压缩

gg findRoot(gg x) { return ufs[x] == x ? x : ufs[x] = findRoot(ufs[x]); }

//合并两个结点所在集合，如果已在同一集合，返回false

bool unionSets(gg a, gg b) {

a = findRoot(a), b = findRoot(b);

if (a == b) {

return false;

}

ufs[a] = b;

return true;

}

private:

vector<gg> ufs;

};

# **树状数组**

## **概述**

树状数组又称二叉索引树，简称 BIT，是算法竞赛中常用的用来维护区间信息的数据结构。树状数组中，下标均从 1 开始。getSum 和 update 操作时间复杂度均为O(log n)*O*(*logn*)。看不懂，但是还是抄上吧

## **lowbit**

在计算机中，一个补码表示的整数 x 变成其相反数-x 的过程相当于把 x 的二进制的每一位都取反， 然后末位加 1，而这等价于**直接把 x 的二进制最右边的 1 左边的每一位都取反**。因此lowbits(x)=x&(-x)表示就是把 x 的二进制最右边的 1 左边的每一位都置为 0 得到的结果，它的值是能整除 x 的最大 2 的幂次数。

## **一维树状数组**

### **单点修改，区间查询**

class BIT {

public:

BIT(gg len) : c(len + 5) {}

//实现将A[x] + v的功能

void update(gg x, gg v) {

for (gg i = x; i < c.size(); i += lowbit(i)) {

c[i] += v;

}

}

//求A[1]~A[x]之和

gg getSum(gg x) {

gg sum = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

sum += c[i];

}

return sum;

}

private:

vector<gg> c;

inline gg lowbit(gg x) { return x & (-x); };};

### **区间修改，单点查询**

update 和 getSum 函数与单点修改，区间查询完全一致，但是需要使用差分建树和查询，下面的代码在单点修改，区间查询的基础上进行了进一步的封装。

class BIT {

public:

BIT(gg len) : c(len + 5) {}

//利用差分建树

BIT(vector<gg>& v) : c(v.size() + 5) {

for (gg i = 1; i < v.size(); ++i) {

update(i, v[i] - v[i - 1]);

}

}

//将区间[l,r]都加上v

void realUpdate(gg l, gg r, gg v) {

update(l, v);

update(r + 1, -v);

}

//求A[x]

gg getSum(gg x) {

gg sum = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

sum += c[i];

}

return sum;

}

private:

vector<gg> c;

inline gg lowbit(gg x) { return x & (-x); };

void update(gg x, gg v) {

for (gg i = x; i < c.size(); i += lowbit(i)) {

c[i] += v;

}

}};

### **区间修改，区间查询**

class BIT {

public:

BIT(gg len) : sum1(len + 5), sum2(len + 5) {}

//利用差分建树(如果序列有初始值)

BIT(vector<gg>& v) : sum1(v.size() + 5), sum2(v.size() + 5) {

for (gg i = 1; i < v.size(); ++i) {

update(i, v[i] - v[i - 1]);

}

}

//将区间[l,r]都加上v

void realUpdate(gg l, gg r, gg v) {

update(l, v);

update(r + 1, -v);

}

//求区间[l,r]的和

gg getRealSum(gg l, gg r) { return getSum(r) - getSum(l - 1); }

private:

vector<gg> sum1, sum2;

inline gg lowbit(gg x) { return x & (-x); };

void update(gg x, gg v) {

for (gg i = x; i < sum1.size(); i += lowbit(i)) {

sum1[i] += v;

sum2[i] += v \* (x - 1);

}

}

//求前缀和

gg getSum(gg x) {

gg ans = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

ans += x \* sum1[i] - sum2[i];

}

return ans;

}};

## **二维树状数组**

### **单点修改，区间查询**

class BIT {

public:

BIT(gg len1, gg len2) : sum(len1 + 5, vector<gg>(len2 + 5)) {}

//将A[x][y]的值增加v

void update(gg x, gg y, gg v) {

for (gg i = x; i < sum.size(); i += lowbit(i)) {

for (gg j = y; j < sum[i].size(); j += lowbit(j)) {

sum[i][j] += v;

}

}

}

// 求左上角为(a,b)，右下角为(c,d)的矩阵的和

gg getRealSum(gg a, gg b, gg c, gg d) {

return getSum(c, d) - getSum(c, b - 1) - getSum(a - 1, d) +

getSum(a - 1, b - 1);

}

private:

vector<vector<gg>> sum;

inline gg lowbit(gg x) { return x & (-x); };

// 求维护左上角为(1,1)，右下角为(x,y)的矩阵的和

gg getSum(gg x, gg y) {

gg ans = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

for (gg j = y; j > 0; j -= lowbit(j)) {

ans += sum[i][j];

}

}

return ans;

}};

### **区间修改，单点查询**

class BIT {

public:

BIT(gg len1, gg len2) : sum(len1 + 5, vector<gg>(len2 + 5)) {}

//利用差分建树(如果序列有初始值)

BIT(vector<vector<gg>>& v) : sum(v.size() + 5, vector<gg>(v[0].size() + 5)) {

for (gg i = 1; i < v.size(); ++i) {

for (gg j = 1; j < v[0].size(); ++j) {

update(i, j,

v[i][j] - v[i][j - 1] - v[i - 1][j] + v[i - 1][j - 1]);

}

}

}

//将左上角为(a,b)，右下角为(c,d)的矩阵的值都加上v

void realUpdate(gg a, gg b, gg c, gg d, gg v) {

update(a, b, v);

update(a, d + 1, -v);

update(c + 1, b, -v);

update(c + 1, d + 1, v);

}

// 求A[x][y]

gg getSum(gg x, gg y) {

gg ans = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

for (gg j = y; j > 0; j -= lowbit(j)) {

ans += sum[i][j];

}

}

return ans;

}

private:

vector<vector<gg>> sum;

inline gg lowbit(gg x) { return x & (-x); };

void update(gg x, gg y, gg v) {

for (gg i = x; i < sum.size(); i += lowbit(i)) {

for (gg j = y; j < sum[0].size(); j += lowbit(j)) {

sum[i][j] += v;

}

}

}};

### **区间修改，区间查询**

class BIT {

public:

BIT(gg len1, gg len2) :

sum1(len1 + 5, vector<gg>(len2 + 5)), sum2(len1 + 5, vector<gg>(len2 + 5)),

sum3(len1 + 5, vector<gg>(len2 + 5)), sum4(len1 + 5, vector<gg>(len2 + 5)) {}

//左上角为(a,b)右下角为(c,d)的矩阵全部加上x

void realUpdate(gg a, gg b, gg c, gg d, gg x) {

update(a, b, x);

update(a, d + 1, -x);

update(c + 1, b, -x);

update(c + 1, d + 1, x);

}

//查询左上角为(a,b)右下角为(c,d)的矩阵和

gg getRealSum(gg a, gg b, gg c, gg d) {

return getSum(c, d) - getSum(a - 1, d) - getSum(c, b - 1) +

getSum(a - 1, b - 1);

}

private:

vector<vector<gg>> sum1, sum2, sum3, sum4;

inline gg lowbit(gg x) { return x & (-x); };

void update(gg x, gg y, gg val) {

for (gg i = x; i < sum1.size(); i += lowbit(i)) {

for (gg j = y; j < sum1[0].size(); j += lowbit(j)) {

sum1[i][j] += val;

sum2[i][j] += val \* x;

sum3[i][j] += val \* y;

sum4[i][j] += val \* x \* y;

}

}

}

//查询左上角为(1,1)右下角为(x,y)的矩阵和

gg getSum(gg x, gg y) {

gg ret = 0;

for (gg i = x; i > 0; i -= lowbit(i)) {

for (gg j = y; j > 0; j -= lowbit(j)) {

ret += (x + 1) \* (y + 1) \* sum1[i][j];

ret -= (y + 1) \* sum2[i][j] + (x + 1) \* sum3[i][j];

ret += sum4[i][j];

}

}

return ret;

}};

# **线段树**

线段树是算法竞赛中常用的用来维护区间信息的数据结构。线段树可以在O(logn)的时间复杂度内实现单点修改、区间修改、区间查询（区间求和，求区间最大值，求区间最小值）等操作。线段树维护的信息，需要满足可加性，即能以可以接受的速度合并信息和修改信息，包括在使用懒惰标记时，标记也要满足可加性（例如取模就不满足可加性，两个取模操作就不能合并在一起做）。注意，线段树和输入序列下标均需从 1 开始。

## **单点修改，区间查询**

template <typename T, typename Update\_Operation = plus<T>, typename Union\_Operation = plus<T>>class SegmentTree {

public:

struct TreeNode {

T val;

TreeNode \*left, \*right;

TreeNode(const T& \_val, TreeNode\* \_left = nullptr, TreeNode\* \_right = nullptr) :

val(\_val), left(\_left), right(\_right) {}

};

// A表示原始的输入序列，如果没有这样的原始序列，用default\_value对线段树中的值进行初始化

SegmentTree(gg len, const T& \_default\_value, T\* A = nullptr) :

n(len), default\_value(\_default\_value), update\_op(Update\_Operation()), union\_op(Union\_Operation()) {

root = init(1, n, A);

}

// update\_op(A[p],v)

void realUpdate(gg p, const T& v) { update(root, 1, n, p, v); }

//查询A的[left,right]区间执行union\_op后的值

T realGetResult(gg left, gg right) { return getResult(root, 1, n, left, right); }

private:

TreeNode\* root; //根结点

gg n; //记录输入序列的长度

T default\_value;

Update\_Operation update\_op;

Union\_Operation union\_op;

TreeNode\* init(gg left, gg right, T\* A) {

if (left == right) {

return A ? new TreeNode(A[left]) : new TreeNode(default\_value);

}

TreeNode\* root = new TreeNode(default\_value);

gg mid = (left + right) / 2;

root->left = init(left, mid, A);

root->right = init(mid + 1, right, A);

root->val = union\_op(root->left->val, root->right->val);

return root;

}

// update\_op(A[p],v)，[l,r]表示当前root结点包含的区间

void update(TreeNode\* root, gg l, gg r, gg p, const T& v) {

if (l == r) {

root->val = update\_op(root->val, v);

return;

}

gg m = (l + r) / 2;

if (p <= m) {

update(root->left, l, m, p, v);

} else {

update(root->right, m + 1, r, p, v);

}

root->val = union\_op(root->left->val, root->right->val);

}

//查询A的[left,right]区间执行union\_op后的值，[l,r]表示当前root结点包含的区间

T getResult(TreeNode\* root, gg l, gg r, gg left, gg right) {

if (left <= l and r <= right) {

return root->val;

}

gg m = (l + r) / 2;

T ans;

bool flag = false; //标记ans是否被赋过值

if (left <= m) {

ans = getResult(root->left, l, m, left, min(m, right));

flag = true;

}

if (right > m) {

auto res = getResult(root->right, m + 1, r, max(left, m + 1), right);

if (flag) {

ans = union\_op(ans, res);

} else {

ans = res;

}

}

return ans;

}};

## **区间修改（只有加法），区间查询**

template <typename T, typename Update\_Operation = plus<T>, typename Union\_Operation = plus<T>,

typename Lazy\_Operation = multiplies<T>, typename Lazy\_Union\_Operation = plus<T>>class SegmentTree {

public:

struct TreeNode {

T val, lazy;

TreeNode \*left, \*right;

TreeNode(const T& \_val, const T& \_lazy, TreeNode\* \_left = nullptr, TreeNode\* \_right = nullptr) :

val(\_val), lazy(\_lazy), left(\_left), right(\_right) {}

};

// A表示原始的输入序列，如果没有这样的原始序列，用default\_value对线段树中的值进行初始化

SegmentTree(gg len, const T& \_sequence\_default\_value, const T& \_lazy\_default\_value, T\* A = nullptr) :

n(len), sequence\_default\_value(\_sequence\_default\_value), lazy\_default\_value(\_lazy\_default\_value),

update\_op(Update\_Operation()), union\_op(Union\_Operation()), lazy\_op(Lazy\_Operation()),

lazy\_union\_op(Lazy\_Union\_Operation()) {

root = init(1, n, A);

}

//将A的[left,right]区间的值都进行update\_op操作

void realUpdate(gg left, gg right, gg v) { update(root, 1, n, left, right, v); }

//查询A的[left,right]区间执行union\_op后的值

T realGetResult(gg left, gg right) { return getResult(root, 1, n, left, right); }

private:

TreeNode\* root; //根结点

gg n; //记录输入序列的长度

T sequence\_default\_value, lazy\_default\_value;

Update\_Operation update\_op;

Union\_Operation union\_op;

Lazy\_Operation lazy\_op;

Lazy\_Union\_Operation lazy\_union\_op;

TreeNode\* init(gg left, gg right, T\* A) {

if (left == right) {

return A ? new TreeNode(A[left], lazy\_default\_value) :

new TreeNode(sequence\_default\_value, lazy\_default\_value);

}

TreeNode\* root = new TreeNode(sequence\_default\_value, lazy\_default\_value);

gg mid = (left + right) / 2;

root->left = init(left, mid, A);

root->right = init(mid + 1, right, A);

root->val = union\_op(root->left->val, root->right->val);

return root;

}

//下传懒惰标记

void pushdown(TreeNode\* root, gg l, gg r) {

gg m = (l + r) / 2;

if (root->lazy != lazy\_default\_value and l != r) {

root->left->val = update\_op(root->left->val, lazy\_op(m - l + 1, root->lazy));

root->left->lazy = lazy\_union\_op(root->left->lazy, root->lazy);

root->right->val = update\_op(root->right->val, lazy\_op(r - m, root->lazy));

root->right->lazy = lazy\_union\_op(root->right->lazy, root->lazy);

root->lazy = lazy\_default\_value;

}

}

// 将A的[left,right]区间的值都进行update\_op操作，[l,r]表示当前root结点包含的区间

void update(TreeNode\* root, gg l, gg r, gg left, gg right, const T& v) {

if (left <= l and r <= right) {

root->val = update\_op(root->val, lazy\_op(r - l + 1, v));

root->lazy = lazy\_union\_op(root->lazy, v);

return;

}

pushdown(root, l, r);

gg m = (l + r) / 2;

if (left <= m) {

update(root->left, l, m, left, min(m, right), v);

}

if (right > m) {

update(root->right, m + 1, r, max(left, m + 1), right, v);

}

root->val = union\_op(root->left->val, root->right->val);

}

//查询A的[left,right]区间执行union\_op后的值，[l,r]表示当前root结点包含的区间

T getResult(TreeNode\* root, gg l, gg r, gg left, gg right) {

if (left <= l and r <= right) {

return root->val;

}

pushdown(root, l, r);

gg m = (l + r) / 2;

T ans;

bool flag = false; //标记ans是否被赋过值

if (left <= m) {

ans = getResult(root->left, l, m, left, min(m, right));

flag = true;

}

if (right > m) {

auto res = getResult(root->right, m + 1, r, max(left, m + 1), right);

if (flag) {

ans = union\_op(ans, res);

} else {

ans = res;

}

}

return ans;

}};

## **区间修改（加法和乘法），区间查询**

template <typename T>struct MulAdd {

T operator()(const T& a, const T& b, const T& c) const { return (a \* b % mod + c) % mod; }};template <typename T>struct Mul {

T operator()(const T& a, const T& b) const { return a \* b % mod; }};template <typename T>struct Add {

T operator()(const T& a, const T& b) const { return (a + b) % mod; }};template <typename T, typename Update\_Operation = MulAdd<T>, typename Union\_Operation = Add<T>,

typename Lazy\_Add = Mul<T>, typename Lazy\_Union\_Add = MulAdd<T>, typename Lazy\_Union\_Mul = Mul<T>>class SegmentTree {

public:

struct TreeNode {

T val, lazy\_add, lazy\_mul;

TreeNode \*left, \*right;

TreeNode(const T& \_val, const T& \_lazy\_mul, const T& \_lazy\_add, TreeNode\* \_left = nullptr,

TreeNode\* \_right = nullptr) :

val(\_val),

lazy\_add(\_lazy\_add), lazy\_mul(\_lazy\_mul), left(\_left), right(\_right) {}

};

// A表示原始的输入序列，如果没有这样的原始序列，用default\_value对线段树中的值进行初始化

SegmentTree(gg len, const T& \_sequence\_default\_value, const T& \_lazy\_mul\_value, const T& \_lazy\_add\_value,

T\* A = nullptr) :

n(len),

sequence\_default\_value(\_sequence\_default\_value), lazy\_add\_value(\_lazy\_add\_value),

lazy\_mul\_value(\_lazy\_mul\_value), update\_op(Update\_Operation()), union\_op(Union\_Operation()),

lazy\_add(Lazy\_Add()), lazy\_union\_add(Lazy\_Union\_Add()), lazy\_union\_mul(Lazy\_Union\_Mul()) {

root = init(1, n, A);

}

//将A的[left,right]区间的值都加上v

void realAdd(gg left, gg right, const T& v) { update(root, 1, n, left, right, lazy\_mul\_value, v); }

//将A的[left,right]区间的值都乘上v

void realMul(gg left, gg right, const T& v) { update(root, 1, n, left, right, v, lazy\_add\_value); }

//查询A的[left,right]区间执行union\_op后的值

T realGetResult(gg left, gg right) { return getResult(root, 1, n, left, right); }

private:

TreeNode\* root; //根结点

gg n; //记录输入序列的长度

T sequence\_default\_value, lazy\_add\_value, lazy\_mul\_value;

Update\_Operation update\_op;

Union\_Operation union\_op;

Lazy\_Add lazy\_add;

Lazy\_Union\_Add lazy\_union\_add;

Lazy\_Union\_Mul lazy\_union\_mul;

TreeNode\* init(gg left, gg right, T\* A) {

if (left == right) {

return A ? new TreeNode(A[left], lazy\_mul\_value, lazy\_add\_value) :

new TreeNode(sequence\_default\_value, lazy\_mul\_value, lazy\_add\_value);

}

TreeNode\* root = new TreeNode(sequence\_default\_value, lazy\_mul\_value, lazy\_add\_value);

gg mid = (left + right) / 2;

root->left = init(left, mid, A);

root->right = init(mid + 1, right, A);

root->val = union\_op(root->left->val, root->right->val);

return root;

}

//下传懒惰标记

void pushdown(TreeNode\* root, gg l, gg r) {

gg m = (l + r) / 2;

if ((root->lazy\_add != lazy\_add\_value or root->lazy\_mul != lazy\_mul\_value) and l != r) {

root->left->val = update\_op(root->left->val, root->lazy\_mul, lazy\_add(m - l + 1, root->lazy\_add));

root->left->lazy\_add = lazy\_union\_add(root->left->lazy\_add, root->lazy\_mul, root->lazy\_add);

root->left->lazy\_mul = lazy\_union\_mul(root->left->lazy\_mul, root->lazy\_mul);

root->right->val = update\_op(root->right->val, root->lazy\_mul, lazy\_add(r - m, root->lazy\_add));

root->right->lazy\_add = lazy\_union\_add(root->right->lazy\_add, root->lazy\_mul, root->lazy\_add);

root->right->lazy\_mul = lazy\_union\_mul(root->right->lazy\_mul, root->lazy\_mul);

root->lazy\_add = lazy\_add\_value;

root->lazy\_mul = lazy\_mul\_value;

}

}

// 将A的[left,right]区间的值都进行update\_op操作，[l,r]表示当前root结点包含的区间

void update(TreeNode\* root, gg l, gg r, gg left, gg right, const T& mul, const T& add) {

if (left <= l and r <= right) {

root->val = update\_op(root->val, mul, lazy\_add(r - l + 1, add));

root->lazy\_add = lazy\_union\_add(root->lazy\_add, mul, add);

root->lazy\_mul = lazy\_union\_mul(root->lazy\_mul, mul);

return;

}

pushdown(root, l, r);

gg m = (l + r) / 2;

if (left <= m) {

update(root->left, l, m, left, min(m, right), mul, add);

}

if (right > m) {

update(root->right, m + 1, r, max(left, m + 1), right, mul, add);

}

root->val = union\_op(root->left->val, root->right->val);

}

//查询A的[left,right]区间执行union\_op后的值，[l,r]表示当前root结点包含的区间

T getResult(TreeNode\* root, gg l, gg r, gg left, gg right) {

if (left <= l and r <= right) {

return root->val;

}

pushdown(root, l, r);

gg m = (l + r) / 2;

T ans;

bool flag = false; //标记ans是否被赋过值

if (left <= m) {

ans = getResult(root->left, l, m, left, min(m, right));

flag = true;

}

if (right > m) {

auto res = getResult(root->right, m + 1, r, max(left, m + 1), right);

if (flag) {

ans = union\_op(ans, res);

} else {

ans = res;

}

}

return ans;

}};

# **ST 表**

## **概述**

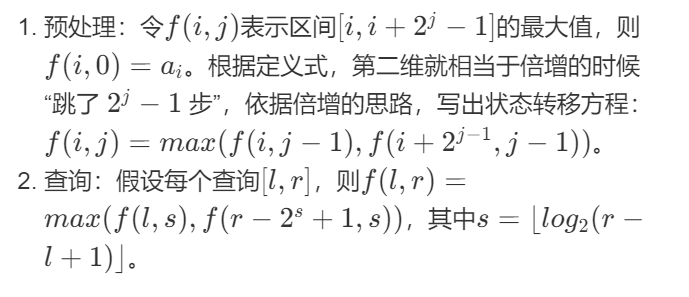
ST 表全称为Sparse-Table，是用于解决可重复贡献问题的数据结构。可重复贡献问题是指对于运算opt，满足*x* *opt* *x*=*x*，则对应的区间询问就是一个可重复贡献问题。例如，最大值有max(x,x)=x，gcd 有gcd(x,x)=x，所以 RMQ（RMQ 是英文 Range Maximum/Minimum Query的缩写，表示区间最大/最小值。） 和区间 GCD 就是一个可重复贡献问题。像区间和就不具有这个性质，如果求区间和的时候采用的预处理区间重叠了，则会导致重叠部分被计算两次。另外，opt还必须满足结合律才能使用 ST 表求解。

## **模板题**

题目大意：给定 n 个数，有 m 个询问，对于每个询问，你需要回答区间 [l,r] 中的最大值。

## **分析**

ST 表基于 倍增 思想，可以做到Θ(*nlogn*) 预处理， O(1)回答每个询问。但是不支持修改操作。注意，查询的下标需从 1 开始。



## **C++代码**

template <typename T, typename Union\_Operation, gg n2 = 20>class ST {

public:

ST(gg len, T\* A, const T& default\_value) :

n(len), st(len + 5, vector<T>(n2, default\_value)), union\_op(Union\_Operation()) {

STinit(A);

}

//求[l,r]区间执行union\_op后的值

gg STquery(gg l, gg r) {

gg s = log2(r - l + 1);

return union\_op(st[l][s], st[r - (1 << s) + 1][s]);

}

private:

gg n; //记录输入序列的长度

vector<vector<T>> st;

Union\_Operation union\_op;

void STinit(T\* A) {

for (gg i = 1; i <= n; ++i) {

st[i][0] = A[i];

}

for (gg j = 1; j <= n2; ++j) {

for (gg i = 1; i + (1 << j) - 1 <= n; ++i) {

st[i][j] = union\_op(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);

}

}

}};

# **平衡树**

//与根结点数据域相等时，一律插入到右子树

struct AVLTreeNode {

gg data, height, num; //关键字、高度

AVLTreeNode \*left, \*right;

AVLTreeNode(gg v, AVLTreeNode\* l = nullptr, AVLTreeNode\* r = nullptr,

gg h = 0) :

data(v),

height(h), left(l), right(r) {}

};

gg getHeight(AVLTreeNode\* r) { return r ? r->height : 0; }

AVLTreeNode\* findAVL(AVLTreeNode\* root, gg data) {

if (not root or root->data == data) {

return root;

} else if (data < root->data) {

return findAVL(root->left, data);

} else {

return findAVL(root->right, data);

}

}

// LL：左左对应的情况(左单旋转),返回旋转后的根节点

AVLTreeNode\* leftLeftRotation(AVLTreeNode\* k2) {

AVLTreeNode\* k1 = k2->left;

k2->left = k1->right;

k1->right = k2;

k2->height = max(getHeight(k2->left), getHeight(k2->right)) + 1;

k1->height = max(getHeight(k1->left), k2->height) + 1;

return k1;

}

// RR：右右对应的情况(右单旋转),返回旋转后的根节点

AVLTreeNode\* rightRightRotation(AVLTreeNode\* k1) {

AVLTreeNode\* k2 = k1->right;

k1->right = k2->left;

k2->left = k1;

k1->height = max(getHeight(k1->left), getHeight(k1->right)) + 1;

k2->height = max(getHeight(k2->right), k1->height) + 1;

return k2;

}

// LR：左右对应的情况(左双旋转),返回旋转后的根节点

AVLTreeNode\* leftRightRotation(AVLTreeNode\* k3) {

k3->left = rightRightRotation(k3->left);

return leftLeftRotation(k3);

}

// RL：右左对应的情况(右双旋转),返回旋转后的根节点

AVLTreeNode\* rightLeftRotation(AVLTreeNode\* k1) {

k1->right = leftLeftRotation(k1->right);

return rightRightRotation(k1);

}

//将结点插入到AVL树中，并返回根节点

AVLTreeNode\* insertNode(AVLTreeNode\*& root, gg data) {

if (not root) {

root = new AVLTreeNode(data);

} else if (data < root->data) {

root->left = insertNode(root->left, data);

// 插入节点后，若AVL树失去平衡，则进行相应的调节。

if (getHeight(root->left) - getHeight(root->right) == 2) {

if (data < root->left->data) {

root = leftLeftRotation(root);

} else {

root = leftRightRotation(root);

}

}

} else { // 应该将data插入到"root的右子树"的情况

root->right = insertNode(root->right, data);

// 插入节点后，若AVL树失去平衡，则进行相应的调节。

if (getHeight(root->right) - getHeight(root->left) == 2) {

if (data > root->right->data) {

root = rightRightRotation(root);

} else {

root = rightLeftRotation(root);

}

}

}

root->height = max(getHeight(root->left), getHeight(root->right)) + 1;

return root;

}

//查找最大结点：返回root为根结点的AVL树的最大结点。

AVLTreeNode\* maximum(AVLTreeNode\* root) {

while (root and root->right) {

root = root->right;

}

return root;

}

//查找最小结点：返回tree为根结点的AVL树的最小结点。

AVLTreeNode\* minimum(AVLTreeNode\* root) {

while (root and root->left)

root = root->left;

return root;

}

//删除结点(z)，返回根节点

AVLTreeNode\* remove(AVLTreeNode\*& root, AVLTreeNode\* z) {

// 根为空或者没有要删除的节点，直接返回nullptr。

if (not root or not z) {

return nullptr;

}

if (z->data < root->data) { // 待删除的节点在"root的左子树"中

root->left = remove(root->left, z);

// 删除节点后，若AVL树失去平衡，则进行相应的调节。

if (getHeight(root->right) - getHeight(root->left) == 2) {

AVLTreeNode\* r = root->right;

if (getHeight(r->left) > getHeight(r->right)) {

root = rightLeftRotation(root);

} else {

root = rightRightRotation(root);

}

}

} else if (z->data > root->data) { // 待删除的节点在"root的右子树"中

root->right = remove(root->right, z);

// 删除节点后，若AVL树失去平衡，则进行相应的调节。

if (getHeight(root->left) - getHeight(root->right) == 2) {

AVLTreeNode\* l = root->left;

if (getHeight(l->right) > getHeight(l->right)) {

root = leftRightRotation(root);

} else {

root = leftLeftRotation(root);

}

}

} else { // root是对应要删除的节点。

if (root->left and root->right) {

if (getHeight(root->left) > getHeight(root->right)) {

// 如果root的左子树比右子树高；

// 则(01)找出root的左子树中的最大节点

// (02)将该最大节点的值赋值给root。

// (03)删除该最大节点。

// 这类似于用"root的左子树中最大节点"做"root"的替身；

// 采用这种方式的好处是：删除"root的左子树中最大节点"之后，AVL树仍然是平衡的。

AVLTreeNode\* Max = maximum(root->left);

root->data = Max->data;

root->left = remove(root->left, Max);

} else {

// 如果root的左子树不比右子树高(即它们相等，或右子树比左子树高1)

// 则(01)找出root的右子树中的最小节点

// (02)将该最小节点的值赋值给root。

// (03)删除该最小节点。

// 这类似于用"root的右子树中最小节点"做"root"的替身；

// 采用这种方式的好处是：删除"root的右子树中最小节点"之后，AVL树仍然是平衡的。

AVLTreeNode\* Min = minimum(root->right);

root->data = Min->data;

root->right = remove(root->right, Min);

}

} else {

AVLTreeNode\* tmp = root;

root = root->left ? root->left : root->right;

delete tmp;

}

}

return root;

}

void remove(AVLTreeNode\*& root, gg data) {

AVLTreeNode\* z;

if ((z = findAVL(root, data))) {

root = remove(root, z);

}

}

# **配对堆**

以下给出的代码实现了小根配对堆，如果需要 decreaseKey 操作，配对堆结点类还需添加 prev 域，代码实现也会复杂一些。因此下面给出没实现和实现了 decreaseKey 操作的两份代码。

## **未实现 decreaseKey 操作的代码**

struct Node {

gg val;

Node \*child = nullptr, \*sibling = nullptr;

Node(gg v) : val(v) {}};

Node\* findMin(Node\* x) { return x; }

Node\* mergeTwoHeaps(Node\* a, Node\* b) {

if (not a) {

return b;

}

if (not b) {

return a;

}

if (a->val > b->val) { //如果要实现大根配对堆，只需将这里的>改成<

swap(a, b);

}

b->sibling = a->child;

a->child = b;

return a;}

Node\* mergeSiblings(Node\* x) {

if (not x or not x->sibling) {

return x;

}

Node \*a = x->sibling, \*b = a->sibling;

x->sibling = a->sibling = nullptr;

return mergeTwoHeaps(mergeTwoHeaps(x, a), mergeSiblings(b));}//如果先使用findMin，再使用deleteMin，务必不要将前面得到的findMin的返回值再合并到堆中，应该重建一个结点

Node\* deleteMin(Node\* x) { return mergeSiblings(x->child); }

## **实现了 decreaseKey 操作的代码**

struct Node {

gg val;

Node \*child = nullptr, \*sibling = nullptr, \*prev = nullptr;

Node(gg v) : val(v) {}};

Node\* findMin(Node\* x) { return x; }

Node\* mergeTwoHeaps(Node\* a, Node\* b) {

if (not a) {

return b;

}

if (not b) {

return a;

}

if (a->val < b->val) { //如果要实现大根配对堆，只需将这里的<改成>

swap(a, b);

}

a->prev = nullptr;

b->prev = a;

b->sibling = a->child;

if (a->child) {

a->child->prev = b;

}

a->child = b;

return a;}

Node\* mergeSiblings(Node\* x) {

if (not x) {

return x;

}

x->prev = nullptr;

if (not x->sibling) {

return x;

}

Node \*a = x->sibling, \*b = a->sibling;

x->sibling = a->sibling = nullptr;

a->prev = nullptr;

return mergeTwoHeaps(mergeTwoHeaps(x, a), mergeSiblings(b));}//如果先使用findMin，再使用deleteMin，务必不要将前面得到的findMin的返回值再合并到堆中，应该重建一个结点

Node\* deleteMin(Node\* x) { return mergeSiblings(x->child); }// root为堆的根，x为要操作的节点，v为新的权值，调用时需保证x->val>=v

Node\* decreaseKey(Node\* root, Node\* x, gg v) {

x->val = v;

if (not x->prev)

return x;

if (x->prev->child == x) {

x->prev->child = x->sibling;

} else {

x->prev->sibling = x->sibling;

}

x->sibling->prev = x->prev;

x->sibling = nullptr;

x->prev = nullptr;

return mergeTwoHeaps(root, x);}



# **子集生成和全排列**

## **子集生成**

使用vector<vector<gg>>类型的ans变量存储最终产生的所有子集。

### **无重复元素**

#### **递归构造法**

vector<vector<gg>> ans;

vector<gg> temp;void sub(vector<gg>& nums, gg p) {

ans.push\_back(temp);

for (gg i = p; i < nums.size(); ++i) {

temp.push\_back(nums[i]);

sub(nums, i + 1);

temp.pop\_back();

}}

vector<vector<gg>> subsets(vector<gg>& nums) {

sub(nums, 0);

return ans;}

#### **位向量法**

vector<vector<gg>> ans;void sub(vector<gg>& nums, vector<bool>& f, gg p) {

if (p == nums.size()) {

ans.push\_back({});

for (gg i = 0; i < f.size(); ++i) {

if (f[i]) {

ans.back().push\_back(nums[i]);

}

}

return;

}

f[p] = true;

sub(nums, f, p + 1);

f[p] = false;

sub(nums, f, p + 1);}

vector<vector<gg>> subsets(vector<gg>& nums) {

vector<bool> f(nums.size());

sub(nums, f, 0);

return ans;}

#### **迭代构造法**

vector<vector<gg>> subsets(vector<gg>& nums) {

vector<vector<gg>> ans{{}};

for (gg i = 0; i < nums.size(); ++i) {

gg s = ans.size();

for (gg j = 0; j < s; ++j) {

ans.push\_back(ans[j]);

ans.back().push\_back(nums[i]);

}

}

return ans;}

#### **二进制法**

vector<vector<gg>> subsets(vector<gg>& nums) {

gg n = nums.size();

vector<vector<gg>> ans(1 << n);

for (gg i = 0; i < (1 << n); ++i) {

for (gg j = 0; j < n; ++j) {

if ((i >> j) & 1) {

ans[i].push\_back(nums[j]);

}

}

}

return ans;}

### **有重复元素**

#### **递归构造法**

vector<vector<gg>> ans;

vector<gg> temp;void sub(vector<gg>& nums, gg p) {

ans.push\_back(temp);

for (gg i = p; i < nums.size(); ++i) {

if (i > p and nums[i] == nums[i - 1]) {

continue;

}

temp.push\_back(nums[i]);

sub(nums, i + 1);

temp.pop\_back();

}}

vector<vector<gg>> subsetsWithDup(vector<gg>& nums) {

sort(begin(nums), end(nums));

sub(nums, 0);

return ans;}

#### **位向量法**

vector<vector<gg>> ans;void sub(vector<gg>& nums, vector<bool>& f, gg p) {

if (p == nums.size()) {

ans.push\_back({});

for (gg i = 0; i < f.size(); ++i) {

if (f[i]) {

ans.back().push\_back(nums[i]);

}

}

return;

}

gg j = find\_if(begin(nums) + p, end(nums),

[&nums, p](gg a) { return a != nums[p]; }) -

begin(nums);

for (gg i = p; i < j; ++i) {

f[i] = true;

sub(nums, f, j);

}

for (gg i = p; i < j; ++i) {

f[i] = false;

}

sub(nums, f, j);}

vector<vector<gg>> subsetsWithDup(vector<gg>& nums) {

vector<bool> f(nums.size());

sort(begin(nums), end(nums));

sub(nums, f, 0);

return ans;}

#### **迭代构造法**

vector<vector<gg>> subsetsWithDup(vector<gg>& nums) {

vector<vector<gg>> ans{{}};

sort(begin(nums), end(nums));

gg s = 1;

for (gg i = 0; i < nums.size(); ++i) {

gg k = ans.size();

if (i == 0 or nums[i] != nums[i - 1]) {

s = k;

}

for (gg j = k - s; j < k; ++j) {

ans.push\_back(ans[j]);

ans.back().push\_back(nums[i]);

}

}

return ans;}

## **全排列**

使用vector<vector<gg>>类型的ans变量存储最终产生的所有排列。

### **无重复元素**

//如果要求字典序输出全排列，调用之前请先对nums排序

vector<vector<gg>> ans;void per(vector<gg>& nums, gg p) {

if (p == nums.size()) {

ans.push\_back(nums);

return;

}

for (gg i = p; i < nums.size(); ++i) {

swap(nums[i], nums[p]);

per(nums, p + 1);

swap(nums[i], nums[p]);

}}

vector<vector<gg>> permute(vector<gg>& nums) {

per(nums, 0);

return ans;}

### **有重复元素**

vector<vector<gg>> ans;void per(vector<gg> nums, gg p) {

if (p == nums.size()) {

ans.push\_back(nums);

return;

}

for (gg i = p; i < nums.size(); ++i) {

if (i == p or nums[i] != nums[p]) {

swap(nums[i], nums[p]);

per(nums, p + 1);

}

}}

vector<vector<gg>> permuteUnique(vector<gg>& nums) {

sort(begin(nums), end(nums));

per(nums, 0);

return ans;}

# 整数取整

gg up(gg n, gg m) { return n >= 0 ? (n + m - 1) / m : n / m; }

gg down(gg n, gg m) { return n >= 0 ? n / m : (n - m + 1) / m; }

# 日期处理

//周日用数字0表示

**gg monthdays[13]** = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; //平年时每个月有多少天

**gg daysOfMonth(gg y, gg m)** { //判断y年m月有几天

if ((y % 400 == 0 || y % 4 == 0 && y % 100 != 0) && m == 2) // y年是闰年且查询2月有几天

return 29; //闰年2月有29天

return monthdays[m]; //直接返回monthdays的相应位置的天数

}

**gg determineWeek**(gg y, gg m, gg d, gg week = 2) { //根据1850年1月1日是周二，返回y年m月d日是周几

for (gg i = 1850; i < y; ++i) { //检查1850年到y年经历的年份

gg temp = (i % 400 == 0 || i % 4 == 0 && i % 100 != 0) ? 366 : 365; //平年有365天，闰年有366天

week = (week + temp) % 7; //更新week

}

for (gg i = 1; i < m; ++i) //检查1月到m月的月份

week = (week + daysOfMonth(y, i)) % 7; //求出该月有几天，并更新week

return (week + d - 1) % 7; //返回周几，注意周日用0表示

}

**gg determineDayOfNumberWeek**(gg y, gg m, gg b, gg c) { //判断y年m月第b个星期c是几号

gg week = determineWeek(y, m, 1); //确定y年m月1日是周几

return 1 + (c + 7 - week) % 7 + 7 \* (b - 1);

}

# **内部排序算法**

可以使用[洛谷题目P1177 【模板】快速排序](https://www.luogu.com.cn/problem/P1177)验证自己编写的算法是否正确（时间复杂度过高的排序算法会TLE）。

## **算法分类**

排序算法可以分为以下5类：

1. 插入类排序：直接插入排序、希尔排序
2. 选择类排序：简单选择排序、堆排序
3. 交换类排序：冒泡排序、快速排序
4. 归并类排序：归并排序
5. 基数类排序：基数排序

**注：本博客编写的算法均是针对**vector<gg>**类型数据进行从小到大排序**

## **八大内部排序算法的C++代码**

### **直接插入排序**

void InsertSort(vector<gg>& v) {

for (gg i = 0; i < v.size(); ++i) {

gg t = v[i], j;

for (j = i; j >= 1 and v[j - 1] > t; --j) {

v[j] = v[j - 1];

}

v[j] = t;

}}

### **希尔排序**

直接插入排序就是增量为1时的希尔排序，二者的代码非常相似，可类比记忆。  
以下代码采取希尔提出的选取增量的方法，即每次增量分别为：  
\lfloor n/2 \rfloor、\lfloor n/4 \rfloor、··· ···、\lfloor n/2^k \rfloor、··· ···、2、1⌊*n*/2⌋、⌊*n*/4⌋、⋅⋅⋅⋅⋅⋅、⌊*n*/2*k*⌋、⋅⋅⋅⋅⋅⋅、2、1

void ShellSort(vector<gg>& v) {

for (gg inc = v.size() / 2; inc >= 1; inc /= 2) {

for (gg i = inc; i < v.size(); ++i) {

gg t = v[i], j;

for (j = i; j >= inc and v[j - inc] > t; j -= inc) {

v[j] = v[j - inc];

}

v[j] = t;

}

}}

### **简单选择排序**

void SelectSort(vector<gg>& v) {

for (gg i = 0; i < v.size(); ++i) {

gg k = i;

for (gg j = i + 1; j < v.size(); ++j) {

if (v[j] < v[k]) {

k = j;

}

}

gg t = v[k];

v[k] = v[i];

v[i] = t;

}}

### **堆排序**

以下代码建立的是大根堆

void Down(vector<gg>& v, gg n, gg p) {

gg t = v[p];

while (2 \* p + 1 < n) {

gg child = 2 \* p + 1;

if (child + 1 < n and v[child] < v[child + 1]) {

++child;

}

if (t < v[child]) {

v[p] = v[child];

p = child;

} else {

break;

}

}

v[p] = t;}void HeapSort(vector<gg>& v) {

for (gg i = v.size() / 2; i >= 0; --i) {

Down(v, v.size(), i);

}

for (gg i = v.size() - 1; i > 0; --i) {

gg t = v[i];

v[i] = v[0];

v[0] = t;

Down(v, i, 0);

}}

### **冒泡排序**

以下代码以**一趟排序过程中如果没有发生关键字交换**作为冒泡排序结束的标志。

void BubbleSort(vector<gg>& v) {

for (gg i = 0; i < v.size(); ++i) {

bool flag = true; //标记是否有关键字交换

for (gg j = 1; j < v.size() - i; ++j) {

if (v[j] < v[j - 1]) {

flag = false;

gg t = v[j];

v[j] = v[j - 1];

v[j - 1] = t;

}

}

if (flag) {

break;

}

}}

### **快速排序**

以下代码选择每个序列的第一个数字作为划分元素

void QSort(vector<gg>& v, gg left, gg right) {

if (left >= right)

return;

gg pivot = v[left], i = left, j = right;

while (i < j) {

while (i < j and v[j] >= pivot) {

--j;

}

if (i < j) {

v[i++] = v[j];

}

while (i < j and v[i] <= pivot) {

++i;

}

if (i < j) {

v[j--] = v[i];

}

}

v[i] = pivot;

QSort(v, left, i - 1);

QSort(v, i + 1, right);}void QuickSort(vector<gg>& v) { QSort(v, 0, v.size() - 1); }

### **归并排序**

void Merge(vector<gg>& v, vector<gg>& a, gg left, gg mid, gg right) {

for (gg i = left, j = mid + 1, k = left; k <= right; ++k) {

if (i > mid) {

a[k] = v[j++];

} else if (j > right) {

a[k] = v[i++];

} else if (v[i] < v[j]) {

a[k] = v[i++];

} else {

a[k] = v[j++];

}

}

for (gg i = left; i <= right; ++i) { //将合并后的有序序列拷贝回原数组

v[i] = a[i];

}}void MSort(vector<gg>& v, vector<gg>& a, gg left, gg right) {

if (left >= right) { //待排序序列中只有0或1个关键字，不必排序，直接返回

return;

}

gg mid = (left + right) / 2; //中间位置

MSort(v, a, left, mid); //递归排序左半部分

MSort(v, a, mid + 1, right); //递归排序右半部分

Merge(v, a, left, mid, right); //合并两个有序序列}void MergeSort(vector<gg>& v) {

vector<gg> a(v.size()); //合并两个有序序列时的输出序列

MSort(v, a, 0, v.size() - 1);}

### **基数排序**

以下代码假设待排序序列以10为基数，且使用了stl中的队列queue作为桶

//返回数组中最大值的位数，即基数排序的趟数

gg getNum(vector<gg>& v, gg radix = 10) {

gg t = v[0];

for (gg i = 1; i < v.size(); ++i) {

if (v[i] > t) {

t = v[i];

}

}

gg i = 0;

for (i = 0; t != 0; t /= radix) {

++i;

}

return i;}void RadixSort(vector<gg>& v, gg radix = 10) {

gg num = getNum(v);

queue<gg> q[radix];

for (gg i = 0, factor = 1; i < num; ++i, factor \*= radix) {

for (gg j = 0; j < v.size(); ++j) {

q[v[j] / factor % radix].push(v[j]);

}

for (gg j = 0, k = 0; j < radix; ++j)

while (!q[j].empty()) {

v[k++] = q[j].front();

q[j].pop();

}

}}

# 划分算法

//划分算法，将数组nums分成<t、==t、>t三部分，返回一个array，表示==t的部分的首元素索引、尾元素的下一个元素的索引

gg nums[MAX];

array<gg, 2> partition(gg t) {

gg i = 0, j = 0, k = nums.size() - 1;

while (i <= k)

if (nums[i] > t)

swap(nums[i], nums[k--]);

else if (nums[i] < t)

swap(nums[i++], nums[j++]);

else

++i;

return {j, i};

}

# **二维前缀和**

## **问题描述**

给出一个m\*n的矩阵 a 以及四个整数*r*1​,*c*1​,*r*2​,*c*2​(1≤*r*1​≤*r*2​<=*m*,1≤*c*1​≤*c*2​<=*n*)，求 ∑*i*=*r*1​*r*2​​∑*j*=*c*1​*c*2​​*a*[i][j]的值。注意，这里的下标都从 1 开始。

## **简析**

## **C++代码**

gg a[MAX][MAX], preSum[MAX][MAX];//计算前缀和void getPrefixSum(gg m, gg n) {

for (gg i = 1; i <= m; ++i) {

for (gg j = 1; j <= n; ++j) {

preSum[i][j] = preSum[i - 1][j] + preSum[i][j - 1] - preSum[i - 1][j - 1] + a[i][j];

}

}}//计算将左上角为(r1,c1)，右下角为(r2,c2)的矩阵的和

gg getSum(gg r1, gg c1, gg r2, gg c2) {

return preSum[r2][c2] - preSum[r1 - 1][c2] - preSum[r2][c1 - 1] + preSum[r1 - 1][c1 - 1];}

# **二维差分**

## **问题描述**

给出一个m\times n*m*×*n*的零矩阵 a 以及多个修改操作，每个修改操作将以左上角坐标(r\_1,c\_1)(*r*1​,*c*1​)，右下角坐标(r\_2,c\_2)(*r*2​,*c*2​)的矩阵中所有的值都增加v*v*，(1\le r\_1 \le r\_2 <=m, 1\le c\_1 \le c\_2 <=n)(1≤*r*1​≤*r*2​<=*m*,1≤*c*1​≤*c*2​<=*n*)，求所有修改操作结束后矩阵 a 的值。注意，这里的下标都从 1 开始。

## **简析**

利用差分的思想进行修改操作，然后利用前缀和的思想查询。能够保证每次修改操作的时间复杂度为O(1)*O*(1)，查询操作的时间复杂度为O(mn)*O*(*mn*)。

## **C++代码**

gg a[MAX][MAX], preSum[MAX][MAX];//计算差分，将左上角为(r1,c1)，右下角为(r2,c2)的矩阵的值都加上vvoid update(gg r1, gg c1, gg r2, gg c2, gg v) {

preSum[r1][c1] += v;

preSum[r1][c2 + 1] -= v;

preSum[r2 + 1][c1] -= v;

preSum[r2 + 1][c2 + 1] += v;}//计算矩阵void getResult(gg m, gg n) {

for (gg i = 1; i <= m; ++i) {

for (gg j = 1; j <= n; ++j) {

a[i][j] = preSum[i][j] + a[i][j - 1] + a[i - 1][j] - a[i - 1][j - 1];

}

}}

# 二分查找

二分查找算法的时间复杂度为O(logn)*O*(*logn*)

## **在升序序列中二分查找某数 x 的位置**

//在升序序列中二分查找某数 x 的位置，二分区间为 [left,right]，如果不存在，返回-1

gg v[MAX];

gg binarySearch(gg left, gg right, gg x) {

while (left <= right) {

gg mid = (left + right) / 2;

if (v[mid] == x)

return mid;

else if (v[mid] < x)

left = mid + 1;

else

right = mid - 1;

}

return -1;}

## **在非降序序列中二分查找第一个大于等于x的位置**

//在非降序序列中二分查找第一个大于等于x的位置，二分区间为[left, right]//如果不存在这样的元素，返回-1

gg v[MAX];

gg lowerBound(gg left, gg right, gg x) {

while (left < right) {

gg mid = (left + right) / 2;

if (v[mid] >= x)

right = mid;

else

left = mid + 1;

}

return left > right or v[left] >= x ? -1: left;}

# **区间覆盖问题**

这里用array<gg,2>类型存储一个区间，区间起点存储在下标为0的位置，区间终点存储在下标为1的位置。

## **区间完全覆盖问题**

### **问题描述**

给定一个区间[0,m][0,*m*]，再给出n*n*条区间[a\_i,b\_i][*ai*​,*bi*​]，求最少使用多少条区间可以将整个区间完全覆盖。

### **贪心策略**

1. 将每一个区间按照左端点升序排序，如果左端点相同，按右端点升序排序；
2. 设置一个变量right表示已经覆盖的区域的右端点，在左端点小于等于right的区间中，每次选择右端点最大且大于right的区间。

### **C++代码**

//只覆盖区间内的所有整点

gg inter[MAX][2];

gg rangeFullCoverage(gg m) {

sort(begin(inter), end(inter));

gg right = -1, ans = 0;

while (right < m) {

gg maxr = 0;

for (gg i = 0; i < inter.size(); ++i) {

if (inter[i][0] <= right + 1) {

maxr = max(maxr, inter[i][1]);

}

}

if(right==maxr){ //如果给出的区间不能完全覆盖[0,m]，返回-1

return -1;

}

right = maxr;

++ans;

}

return ans;}

//覆盖整个区间

gg inter[MAX][2];

gg rangeFullCoverage(gg m) {

sort(begin(inter),end(inter));

gg right = 0, ans = 0;

while (right < m) {

gg maxr = 0;

for (gg i = 0; i < inter.size(); ++i) {

if (inter[i][0] <= right) {

maxr = max(maxr, inter[i][1]);

}

}

if(right==maxr){ //如果给出的区间不能完全覆盖[0,m]，返回-1

return -1;

}

right = maxr;

++ans;

}

return ans;}

## **最大不相交问题**

### **问题描述**

给出n*n*条区间[a\_i,b\_i][*ai*​,*bi*​]，从中选取尽量多的区间，使得这些区间两两没有公共点。

### **贪心策略**

1. 将每一个区间按照右端点升序排序，如果右端点相同，按左端点升序排序；
2. 设置一个变量right表示已经覆盖的区域的右端点，如果当前区间左端点小于right，就选取该区间；否则用同样的方式尝试选取其它区间。

### **C++代码**

//覆盖整个区间

gg inter[MAX][2];

gg rangeDisjoint() {

sort(begin(inter), end(inter),

[](const array<gg, 2>& a, const array<gg, 2>& b) {

return a[1] != b[1] ? a[1] < b[1] : a[0] < b[0];

});

gg ans = 0, right = -1;

for (gg i = 0; i < inter.size(); ++i) {

if (right < inter[i][0]) {

++ans;

right = inter[i][1];

}

}

return ans;}

## **区间选点问题**

### **问题描述**

给出n*n*条区间[a\_i,b\_i][*ai*​,*bi*​]，要求选取尽量少的点，使得每个区间内都至少有一个点（不同区间内含的点可以是同一个）。

### **贪心策略**

1. 将每一个区间按照左端点升序排序，左端点相同的按右端点升序排序；
2. 从第一个区间右端点开始往后找，如果下一个区间的左端点大于当前已选区间的右端点，说明要新开一个点，计数器加1，同时更新右区间能覆盖的最远距离；如果下一个区间右端点小于当前已选区间的右端点，说明共享的区间范围缩短了，那么就更新区间右端点为下一个区间右端点，重复以上操作，直至筛选完所有区间。

### **C++代码**

//区间选点

gg inter[MAX][2];

gg rangeSelectPoint() {

sort(begin(inter), end(inter));

gg ans = 0, right = -1;

for (gg i = 0; i < inter.size(); ++i) {

if (right < inter[i][0]) {

++ans;

}

right = min(inter[i][1], right);

}

return ans;}

# **二叉树**

## **结点类定义**

struct BTNode {

gg val;

BTNode \*left, \*right;

BTNode(gg v, BTNode\* l = nullptr, BTNode\* r = nullptr) : val(v) {}};

## **深度优先遍历（DFS）的递归写法**

算法的时间复杂度为O(n)

//二叉树的先根遍历void preOrder(BTNode\* root) {

if (not root) //是空树直接返回

return;

cout << root->val << ' '; //访问根结点

preOrder(root->left); //递归遍历左子树

preOrder(root->right); //递归遍历右子树}//二叉树的中根遍历void inOrder(BTNode\* root) {

if (not root) //是空树直接返回

return;

inOrder(root->left); //递归遍历左子树

cout << root->val << ' '; //访问根结点

inOrder(root->right); //递归遍历右子树}//二叉树的先根遍历void postOrder(BTNode\* root) {

if (not root) //是空树直接返回

return;

postOrder(root->left); //递归遍历左子树

postOrder(root->right); //递归遍历右子树

cout << root->val << ' '; //访问根结点}

## **深度优先遍历（DFS）的非递归写法**

算法的时间复杂度为O(n)

//二叉树的先根遍历void preOrder(BTNode\* root) {

stack<BTNode\*> st;

while (root or not st.empty()) {

while (root) {

cout << root->val << " ";

st.push(root);

root = root->left;

}

root = st.top();

st.pop();

root = root->right;

}}//二叉树的中根遍历//注意中根遍历和先根遍历代码的差别仅仅在于ans.push\_back(root->val);的位置void inOrder(BTNode\* root) {

stack<BTNode\*> st;

while (root or not st.empty()) {

while (root) {

st.push(root);

root = root->left;

}

root = st.top();

st.pop();

cout << root->val << " ";

root = root->right;

}}//二叉树的后根遍历//注意后根遍历和先根遍历代码的差别在于先将右子树入栈，再将左子树入栈，最后进行一次翻转void postOrder(BTNode\* root) {

vector<gg> ans;

stack<BTNode\*> st;

while (root or not st.empty()) {

while (root) {

ans.push\_back(root->val);

st.push(root);

root = root->right;

}

root = st.top();

st.pop();

root = root->left;

}

reverse(begin(ans), end(ans));

for(gg i: ans){

cout << i << " ";

}}

## **广度优先遍历（BFS）**

算法的时间复杂度为O(n)

//二叉树的层次遍历void levelOrder(BTNode\* root) {

queue<BTNode\*> q;

q.push(root);

while (not q.empty()) {

gg s = q.size();

while (s--) {

auto t = q.front();

q.pop();

cout << t->val << (s == 0 ? '\n' : ' ');

if (t->left)

q.push(t->left);

if (t->right)

q.push(t->right);

}

}}

## **判断二叉树是否为完全二叉树**

算法的时间复杂度为O(n)

bool isCompleteTree(BTNode\* root) {

queue<pair<BTNode\*, gg>> q; // pair的second成员存储结点编号

q.push({root, 1});

for (gg i = 1; !q.empty(); ++i) {

auto t = q.front();

q.pop();

if (i != t.second)

return false;

if (t.first->left != nullptr)

q.push({t.first->left, t.second \* 2});

if (t.first->right != nullptr)

q.push({t.first->right, t.second \* 2 + 1});

}

return true;}

## **二叉查找树的相关操作**

插入、查找操作时间复杂度均为O(h)，其中 h 为树的高

//向二叉查找树中插入元素xvoid insertElement(BTNode\*& root, gg x) {

if (not root) { //根结点为空，新建一个结点

root = new BTNode(x);

} else if (x <= root->val) { //向左子树中插入

insertElement(root->left, x);

} else { //向右子树中插入

insertElement(root->right, x);

}}//在二叉查找树中查找元素x的位置，查找失败则返回空指针

BTNode\* findElement(BTNode\* root, gg x) {

if (not root or root->val == x) { //查找失败或查找成功

return root;

} else if (x <= root->val) { //向左子树中查找

return findElement(root->left, x);

} else { //向右子树中查找

return findElement(root->right, x);

}}

## **根据遍历序列创建二叉树问题**

注意，**这里的代码模板针对的二叉树，结点数据域的值都是唯一的，不会出现重复**

### **由中根序列和先根序列创建二叉树**

算法的时间复杂度为O(n)

// pre为先根序列，in为中根序列，r为根结点在pre中的下标//[left, right]为当前创建的树的中根序列区间

BTNode\* buildTree(vector<gg>& pre, vector<gg>& in, gg r, gg left, gg right) {

if (left > right) //序列为空，返回空指针

return nullptr;

//查找根结点在中根序列中的位置

gg i = find(begin(in), end(in), pre[r]) - begin(in);

auto root = new BTNode(pre[r]); //创建根结点

root->left = buildTree(pre, in, r + 1, left, i - 1); //创建左子树

root->right = buildTree(pre, in, r + 1 + i - left, i + 1, right); //创建右子树

return root; //返回根结点}

### **由中根序列和先根序列得到后根序列**

算法的时间复杂度为O(n)

// pre为先根序列，in为中根序列，r为根结点在pre中的下标//[left, right]为当前创建的树的中根序列区间void getPostFromPreIn(vector<gg>& pre, vector<gg>& in, gg r, gg left, gg right) {

if (left > right) //序列为空，直接返回

return;

//查找根结点在中根序列中的位置

gg i = find(begin(in), end(in), pre[r]) - begin(in);

getPostFromPreIn(pre, in, r + 1, left, i - 1); //递归遍历左子树

getPostFromPreIn(pre, in, r + 1 + i - left, i + 1, right); //递归遍历右子树

cout << pre[r] << ' '; //输出后根序列}

### **由先根序列创建二叉查找树**

算法的时间复杂度为O(n)

// pre为先根序列，[left, right]为当前创建的树的先根序列区间

BTNode\* buildBST(vector<gg>& pre, gg left, gg right) {

if (left > right) //序列为空，返回空指针

return nullptr;

//查找右子树根结点在先根序列中的位置

gg i = find\_if(begin(pre) + left, begin(pre) + right + 1,

[&pre, left](gg a) { return a > pre[left]; }) - begin(pre);

auto root = new BTNode(pre[left]); //创建根结点

root->left = buildBST(pre, left + 1, i - 1); //创建左子树

root->right = buildBST(pre, i, right); //创建右子树

return root; //返回根结点}

### **由二叉查找树的先根序列得到后根序列**

算法的时间复杂度为O(n)

// pre为先根序列，[left, right]为当前创建的树的先根序列区间void getPostFromBSTPre(vector<gg>& pre, gg left, gg right) {

if (left > right) //序列为空，直接返回

return;

//查找右子树根结点在先根序列中的位置

gg i = find\_if(begin(pre) + left, begin(pre) + right + 1,

[&pre, left](gg a) { return a > pre[left]; }) - begin(pre);

getPostFromBSTPre(pre, left + 1, i - 1); //递归遍历左子树

getPostFromBSTPre(pre, i, right); //递归遍历右子树

cout << pre[left] << ' ';}

## **最近公共祖先（LCA）问题**

这里的代码模板针对的二叉树和要查找的指针 p、q，满足以下约定：

1. 二叉树中结点数据域的值都是唯一的，不会出现重复；
2. p 指针和 q 指针指向的结点一定在树中；
3. 非空结点与空结点的最近公共祖先为该非空结点。

### **普通二叉树**

//在普通二叉树中查找p和q的最近公共祖先

BTNode\* LCA(BTNode\* root, BTNode\* p, BTNode\* q) {

if (not root or root == p or root == q)

return root;

BTNode \*left = LCA(root->left, p, q), \*right = LCA(root->right, p, q);

return not left ? right : not right ? left : root;}

### **二叉查找树**

//在二叉查找树中查找p和q的最近公共祖先

BTNode\* LCA(BTNode\* root, BTNode\* p, BTNode\* q) {

return (root->val - p->val) \* (root->val - q->val) <= 0 ?

root : LCA(p->val < root->val ? root->left : root->right, p, q);}

# **多叉树**

## **结点类定义**

struct TreeNode {

gg val;

vector<TreeNode\*> child;

TreeNode(gg v) : val(v) {}};

## **深度优先遍历（DFS）**

算法的时间复杂度为O(n)

//树的先根遍历void preOrder(TreeNode\* root) {

if (not root) //是空树直接返回

return;

cout << root->val << ' '; //访问根结点

for (auto i : root->child) //递归遍历所有子树

preOrder(i);}//树的后根遍历void postOrder(TreeNode\* root) {

if (not root) //是空树直接返回

return;

for (auto i : root->child) //递归遍历所有子树

postOrder(i);

cout << root->val << ' '; //访问根结点}

## **广度优先遍历（BFS）**

算法的时间复杂度为O(n)

//树的层次遍历void levelOrder(TreeNode\* root) {

queue<TreeNode\*> q;

q.push(root);

while (not q.empty()) {

gg s = q.size();

while (s--) {

auto t = q.front();

q.pop();

cout << t->val << (s == 0 ? '\n' : ' ');

for (auto i : t->child)

q.push(i);

}

}}

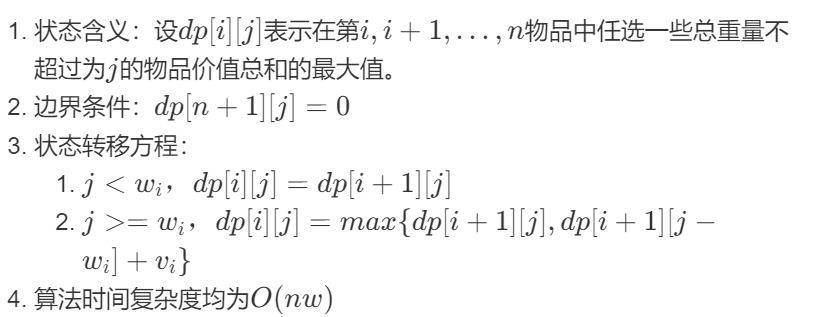
# **背包问题**

## **0-1 背包问题**

### **问题描述**

有 n 种重量和价值分别为*w i*​ , *vi*​的物品。从这些物品中挑选出总重量不超过w的物品，求所有挑选方案中价值总和的最大值（物品下标为 1 到 n）。每种物品最多只能选一件。

### **从后向前递推**



C++代码：

gg backpack01(){

for (gg i = ni; i >= 1; --i) {

for (gg j = 0; j <= w; ++j) {

dp[i][j] = dp[i + 1][j];

if (j >= wi[i]) {

dp[i][j] = max(dp[i][j], dp[i + 1][j - wi[i]] + vi[i]);

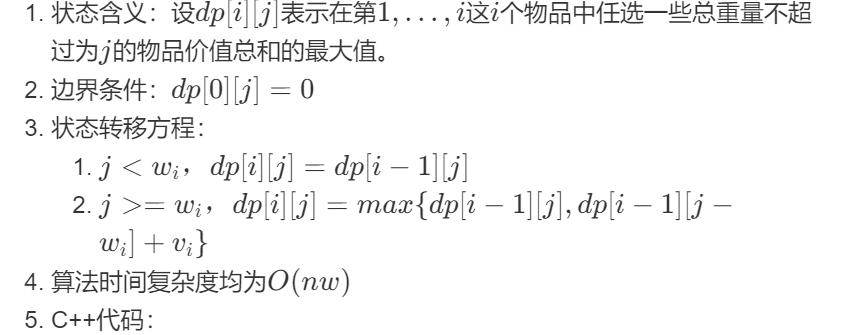
}

}

}

return dp[1][w];}

### **从前向后递推**

gg backpack01() {

for (gg i = 1; i <= ni; ++i) {

for (gg j = 0; j <= w; ++j) {

dp[i][j] = dp[i - 1][j];

if (j >= wi[i])

dp[i][j] = max(dp[i][j], dp[i - 1][j - wi[i]] + vi[i]);

}

}

return dp[ni][w];}

### **一维滚动数组**

算法时间复杂度均为O(nw)

gg backpack01() {

for (gg i = 1; i <= ni; ++i) {

for (gg j = w; j >= wi[i]; --j) {

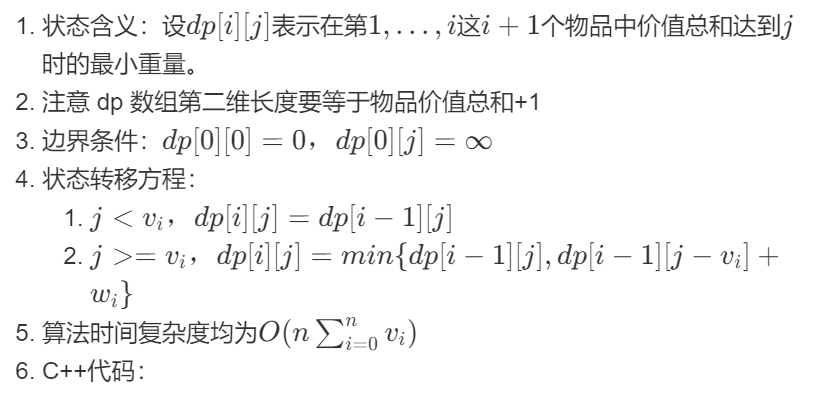
dp[j] = max(dp[j], dp[j - wi[i]] + vi[i]);

}

}

return dp[w];}

### **当总重量 w 过大时，更换 DP 对象**



gg backpack01() {

gg v = accumulate(vi + 1, vi + n + 1, 0ll);

fill(begin(dp[0]), begin(dp[0]) + v + 1, INF);

dp[0][0] = 0;

for (gg i = 1; i <= ni; ++i) {

for (gg j = 0; j <= v; ++j) {

dp[i][j] = dp[i - 1][j];

if (j >= vi[i]) {

dp[i][j] = min(dp[i][j], dp[i - 1][j - vi[i]] + wi[i]);

}

}

}

for (gg j = v; j >= 0; --j)

if (dp[ni][j] <= w)

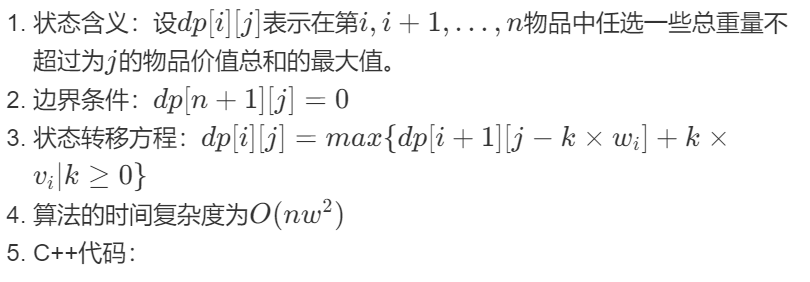
return j;}

## **完全背包问题**

### **问题描述**

有 n 种重量和价值分别为w\_i,v\_i*wi*​,*vi*​的物品。从这些物品中挑选出总重量不超过w*w*的物品，求所有挑选方案中价值总和的最大值（物品下标为 1 到 n）。每种物品可以挑选任意多件。

### **从后向前递推**



gg backpackComplete(){

for (gg i = ni; i >= 1; --i) {

for (gg j = 0; j <= w; ++j) {

for (gg k = 0; k \* wi[i] <= j; ++k) {

dp[i][j] = max(dp[i][j], dp[i + 1][j - k \* wi[i]] + k \* vi[i]);

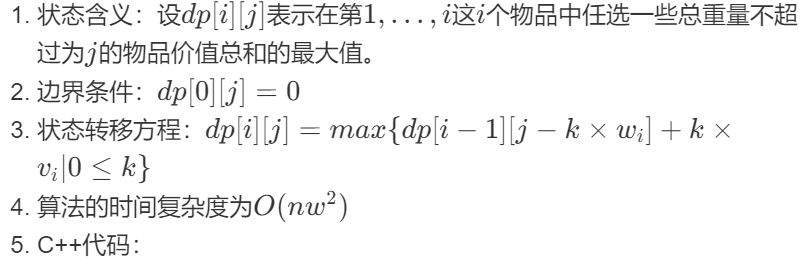
}

}

}

return dp[1][w];}

### **从前向后递推**



gg backpackComplete() {

for (gg i = 1; i <= ni; ++i) {

for (gg j = 0; j <= w; ++j) {

for (gg k = 0; k \* wi[i] <= j; ++k) {

dp[i][j] = max(dp[i][j], dp[i - 1][j - k \* wi[i]] + k \* vi[i]);

}

}

}

return dp[ni][w];}

### **一维滚动数组**

算法的时间复杂度为O(nw)，与 0-1 背包问题的一维滚动数组差别仅在于 j 循环的方向。

gg backpackComplete() {

for (gg i = 1; i <= ni; ++i) {

for (gg j = wi[i]; j <= w; ++j) {

dp[j] = max(dp[j], dp[j - wi[i]] + vi[i]);

}

}

return dp[w];}

## **分组背包问题**

### **问题描述**

有 n 种重量和价值分别为*wi*​,*vi*​且组号*gi* ​的物品。同一组的物品互相冲突，不能放在一起。从这些物品中挑选出总重量不超过w的物品，求所有挑选方案中价值总和的最大值（物品下标为 1 到 n）。每种物品可以挑选任意多件。

### **算法设计**

这是 0-1 背包问题的变形、因为背包 dp 需要背包空间大小循环完一遍才能表示把一个物体放进去了，而在循环背包空间的过程中枚举物品，就相当于只放一种物品，因此我们可以在循环背包空间的过程外部枚举所有组，在循环背包空间的过程内部枚举同一组的所有物品，然后套用 0-1 背包的模板即可。

unordered\_map<gg, vector<gg>> groups; //存储组号和对应组的所有物品索引

gg backpackGroup(){

for (auto& g : groups) {

for (gg j = mi; j >= 0; --j) {

for (gg i : g.second) {

if (j >= wi[i]) {

dp[j] = max(dp[j], dp[j - wi[i]] + vi[i]);

}

}

}

}}

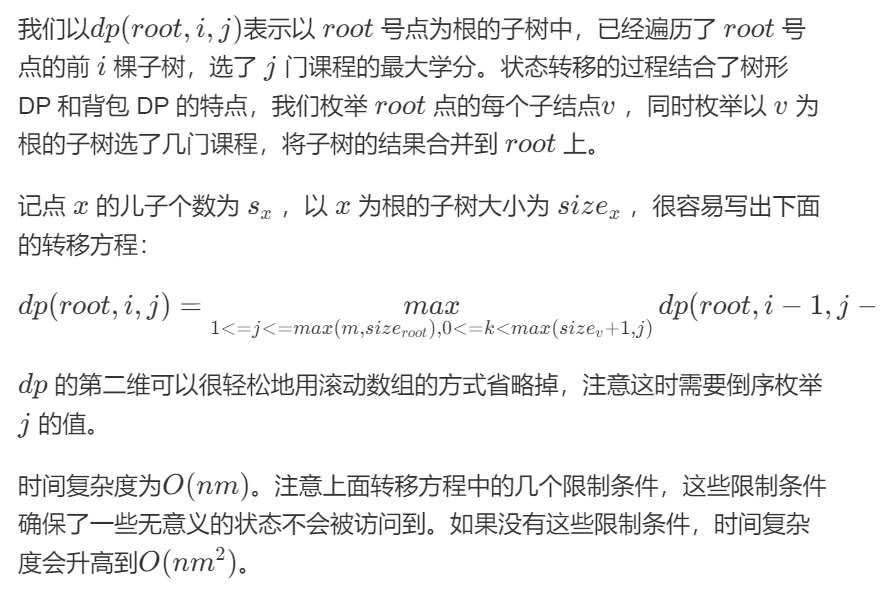
## **树状+背包 DP 问题**

### **问题描述**

[模板题——P2014 [CTSC1997]选课](https://www.luogu.com.cn/problem/P2014)

现在有 N 门功课，每门课有个学分，每门课有一门或没有直接先修课。一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

### **算法设计**

vector<gg> tree[MAX];

gg dfs(gg root) {

gg s = 1;

for (gg i : tree[root]) {

gg t = dfs(i);

s += t;

for (gg j = min(mi, s); j >= 1; --j) {

for (gg k = 0; k < min(j, t + 1); ++k) {

dp[root][j] = max(dp[root][j], dp[i][k] + dp[root][j - k]);

}

}

}

return s;}

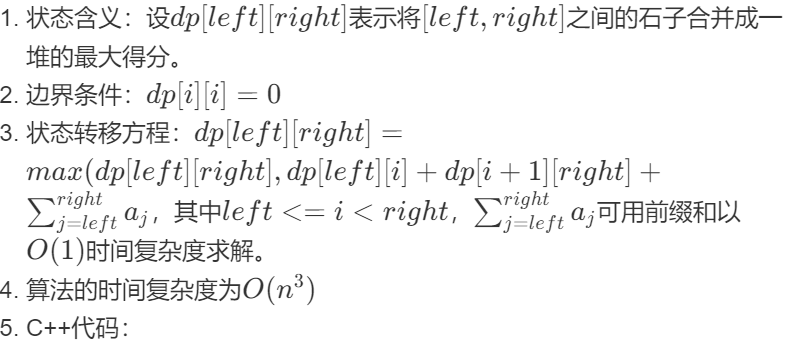
# **线性动态规划**

## **合并石子问题**

### **问题描述**

n 堆石子排成一排，每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。求将 n 堆石子合并成 1 堆的最大得分。

### **算法设计**



gg intervalDP(){

for (gg len = 2; len <= ni; ++len) { //枚举区间长度

for (gg left = 1; left <= ni - len + 1; ++left) { //枚举左端点

gg right = left + len - 1; //右端点

for (gg i = left; i < right; ++i) {

dp[left][right] = max(dp[left][right], dp[left][i] + dp[i + 1][right] + sum[right] - sum[left - 1]);

}

}

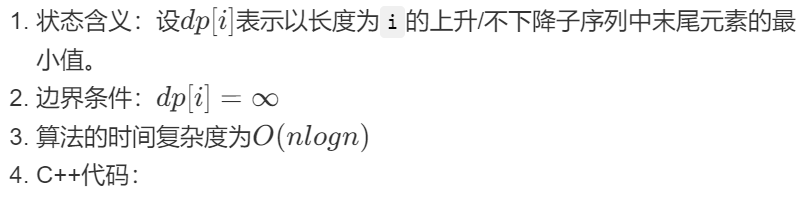
}

return dp[1][ni];}

# **线性动态规划**

本文数组下标均从 1 开始。

## **最长上升/不下降子序列（LIS）问题**



//最长上升子序列

gg dp[MAX];

gg LIS(vector<gg>& nums) {

fill(begin(dp) + 1, begin(dp) + ni + 1, INF);

for (gg i : nums) {

\*lower\_bound(begin(dp) + 1, begin(dp) + ni + 1, i) = i;

}

return lower\_bound(begin(dp) + 1, begin(dp) + ni + 1, INF) - begin(dp) - 1;}

//最长不下降子序列

gg dp[MAX];

gg LIS(vector<gg>& nums) {

fill(begin(dp) + 1, begin(dp) + ni + 1, INF);

for (gg i : nums) {

\*upper\_bound(begin(dp) + 1, begin(dp) + ni + 1, i) = i;

}

return lower\_bound(begin(dp) + 1, begin(dp) + ni + 1, INF) - begin(dp) - 1;}

## **最长公共子序列（LCS）问题**

1. 算法设计：记录下 a 中所有数字的下标，将 b 中数字转换成对应在 a 中的下标，求转换后 b 数组的最长上升子序列。
2. 算法的时间复杂度为O(nlogm)
3. C++代码：

gg dp[MAX];

gg longestCommonSubsequence(vector<gg>& a, vector<gg>& b) {

map<gg, vector<gg>> m;

fill(begin(dp) + 1, begin(dp) + ni + 1, INF);

for (gg i = 1; i <= ni; ++i) {

m[a[i]].push\_back(i);

}

for (gg i = 1; i <= mi; ++i) {

if (m.count(b[i])) {

for (auto j = m[b[i]].rbegin(); j != m[b[i]].rend(); ++j) {

\*lower\_bound(begin(dp) + 1, begin(dp) + ni + 1, \*j) = \*j;

}

}

}

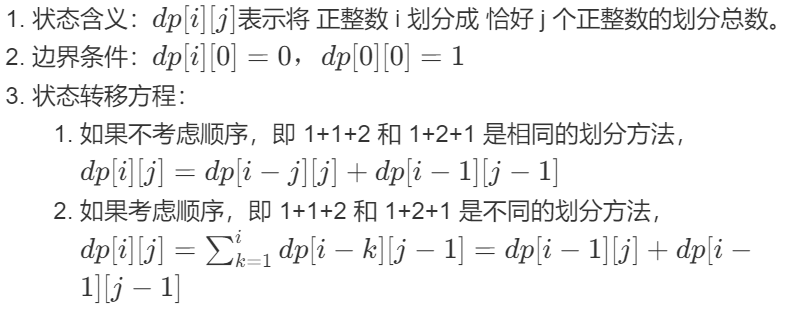
return lower\_bound(begin(dp) + 1, begin(dp) + ni + 1, INF) - begin(dp) - 1;}

## **数的划分问题**

### **将正整数 n 划分成不超过 m 个正整数，这些正整数之和恰好等于 n，求划分方法数**

1. 状态含义：dp[i][j]表示将 正整数 i 划分成 j 个正整数的划分总数。
2. 边界条件：dp[i][0]=0，dp[0][0]=1

### **将正整数 n 划分成恰好 m 个正整数，这些正整数之和恰好等于 n，求划分方法数**



# Bellman-ford

void insertEdge(gg from, gg to, gg cap, gg cost) { //插入边

graph[from].push\_back(edges.size());

edges.push\_back(Edge(from, to, cap, 0, cost));

graph[to].push\_back(edges.size());

edges.push\_back(Edge(to, from, 0, 0, -cost));

}

bool BellmanFord(gg s, gg t, gg& flow, long long& cost) { //最大流算法,s为源点,t为汇点

gg a[MAXV] = {0}, p[MAXV] = {0},

dis[MAXV] = {0}; // a数组表示源点到结点a[i]的残量,p数组表示最短路树上到达结点p[i]的边在edges数组中的序号

fill(dis, dis + MAXV, INF);

bool inQueue[MAXV] = {false};

dis[s] = 0;

inQueue[s] = true;

a[s] = INF; //起点的残量置为无穷大

queue<gg> q;

q.push(s);

while (!q.empty()) { //广度优先遍历查找从源点到达汇点的增广路

gg u = q.front();

q.pop();

inQueue[u] = false;

for (gg i : graph[u]) { //遍历以x为起点的边

Edge& e = edges[i];

if (e.cap > e.flow && dis[e.to] > dis[u] + e.cost) { //当前边的终点的残量为0且容量大于流量

dis[e.to] = dis[u] + e.cost;

p[e.to] = i; //更新到达该终点的边的编号

a[e.to] = min(a[u], e.cap - e.flow); //更新源点到该终点的残量

if (!inQueue[e.to]) {

inQueue[e.to] = true;

q.push(e.to); //压入队列

}

}

}

}

if (dis[t] == INF)

return false;

flow += a[t];

cost += dis[t] \* 1ll \* a[t];

for (gg u = t; u != s; u = edges[p[u]].from) { //从汇点向前遍历增广路经，更新每条增广路的流量

edges[p[u]].flow += a[t];

edges[p[u] ^ 1].flow -= a[t];

}

return true;

}

//需要保证初始网络中没有负权环

pair<gg, long long> MinCostMaxFlow(gg s, gg t) {

gg flow = 0;

long long cost = 0;

while (BellmanFord(s, t, flow, cost))

;

return {flow, cost};

}

# Edmond-karp

void insertEdge(gg from, gg to, gg cap) { //插入边

graph[from].push\_back(edges.size());

edges.push\_back(Edge(from, to, cap, 0));

graph[to].push\_back(edges.size());

edges.push\_back(Edge(to, from, 0, 0));

}

gg a[MAX], p[MAX]; // a数组表示源点到结点a[i]的残量,p数组表示最短路树上到达结点p[i]的边在edges数组中的序号

gg MaxFlow(gg s, gg t) { //最大流算法,s为源点,t为汇点

gg flow = 0; //最大流量

while (true) {

memset(a, 0, sizeof(a)); //将源点到达每个结点的残量初始化为0

//广度优先遍历查找从源点到达汇点的增广路

queue<gg> q;

q.push(s);

a[s] = LONG\_LONG\_MAX; //起点的残量置为无穷大

while (!q.empty()) {

gg x = q.front();

q.pop();

for (gg i : graph[x]) { //遍历以x为起点的边

Edge& e = edges[i];

if (a[e.to] == 0 && e.cap > e.flow) { //当前边的终点的残量为0且容量大于流量

p[e.to] = i; //更新到达该终点的边的编号

a[e.to] = min(a[x], e.cap - e.flow); //更新源点到该终点的残量

q.push(e.to); //压入队列

}

}

if (a[t] != 0) //终点的残量不为零，跳出循环

break;

}

if (a[t] == 0) //终点的残量为零，表示不存在增广路了，跳出外层死循环

break;

for (gg u = t; u != s; u = edges[p[u]].from) { //从汇点向前遍历增广路经，更新每条增广路的流量

edges[p[u]].flow += a[t];

edges[p[u] ^ 1].flow -= a[t];

}

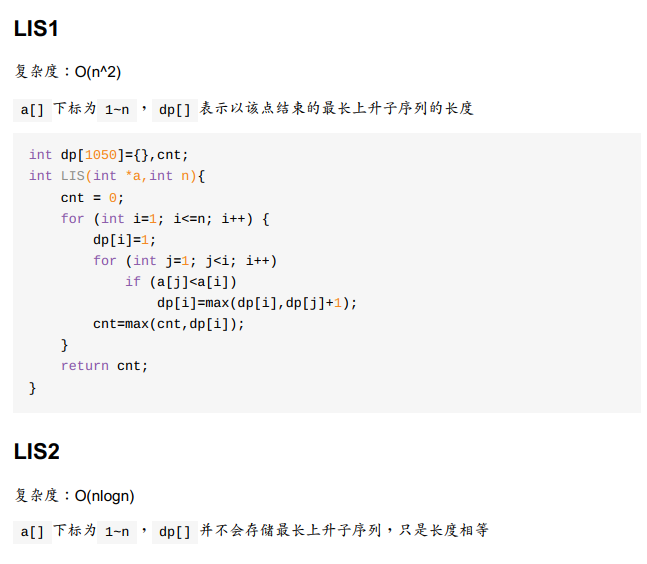
flow += a[t]; //增加最大流量

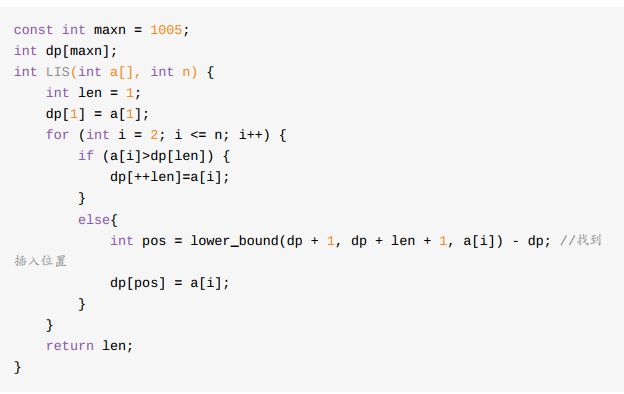
}

return flow;

}

# 最长上升子序列





# //并查集 时间复杂度O(n)

/\*并查集一般用于对动态连通性的判断，主要应用于判断两个元素是否同集合，

\*是否连通，间接好友的判断。判断图是否连通，是否有环。

\*并查集分为带权和不带权

\*/

//不带权并查集，合并时将序号小的作为fa，大多数情况直接套用

void init(vector<int>&fa,int n) {

for (int i=0; i<n; ++i)

fa[i]=i;

}

int findFather(vector<int>&fa,int r) {

if (fa[r]!=r)

fa[r]=findFather(fa,fa[r]);

return fa[r];

}

void Union (vector<int>&fa,int u,int v) {

int ufa=findFather(fa,u);

int vfa=findFather(fa,v);

if (ufa<vfa)

fa[v]=fa[vfa]=ufa;

else

fa[u]=fa[ufa]=vfa;

}

# //树状数组 时间复杂度O(ElogE)

/\*树状数组用于查询任意两位间元素和，每次只能修改一个元素的值，代码简洁

\*一般情况下树状数组能解决的问题线段树都能解决，反之不行。

\*/

const int N = 500005;

struct Node {

int val;

int pos;

};

Node node[N];

int reflect[N], n;

bool cmp(const Node& a, const Node& b) {

return a.val < b.val;

}

//完全功能模板

//注意c中元素位置从1开始

int c[N];

int lowbit(int x) {

return x & (-x);

}

void update(int x,int add) { //一维

while(x<=n) { //n为元素个数 ，与MAXN不同，x为位置

a[x]+=add;

x+=lowbit(x);

}

}

int getsum(int x) {

int sum = 0;

while (x > 0) {

sum += c[x];

x -= lowbit(x);

}

return sum;

}

int main() {

for (int i = 1; i <= n; ++i) c[i] = 0; //初始化树状数组

sort(node + 1, node + n + 1, cmp); //排序

for (int i = 1; i <= n; ++i) reflect[node[i].pos] = i; //离散化

for (int i = 1; i <= n; ++i) { update(reflect[i],1);

ans += i - getsum(reflect[i]); //反面思考，总个数-小于等于的元素个数=比他大的个数

} printf("%lld\n", ans);return 0;}

void modify(int x,int y,int data) { //二维

for(int i=x; i<MAXN; i+=lowbit(i))

for(int j=y; j<MAXN; j+=lowbit(j))

a[i][j]+=data;

}

int get\_sum(int x,int y) {

int res=0;

for(int i=x; i>0; i-=lowbit(i))

for(int j=y; j>0; j-=lowbit(j))

res+=a[i][j];

return res;

}

# // Dijkstra，路径的花费不能有负 时间复杂度O(ElogE)

/\*注意具体题目变化，一般需要多加考虑限制条件和其它变量值

摘自 ZOJ3794 贪心驾驶员

\*/

const int maxn = 1500;

const int INF = 0x3f3f3f3f;

struct Edge {

int from, to, dist;

};

struct HeapNode {

int d, u;

bool operator < (const HeapNode & rhs) const {

return d > rhs.d;

}

};

struct Dijkstra {

int n, m;

vector<Edge> edges;

vector<int> G[maxn];

bool done[maxn]; //标记

int d[maxn]; //花费

void init(int n) {

this -> n = n;

for (int i = 0; i < n; i++)

G[i].clear();

edges.clear();

}

void AddEdge(int from, int to, int dist) {

edges.push\_back((Edge) {from, to, dist});

m = edges.size();

G[from].push\_back(m-1);

}

void dijkstra(int s) {

priority\_queue<HeapNode> Q;

memset(d,0x3f,sizeof(d));

memset(done, 0, sizeof(done));

d[s] = 0;

Q.push((HeapNode) {0, s});

while (!Q.empty()) {

HeapNode x = Q.top();

Q.pop();

int u = x.u;

if (done[u]) continue;

done[u] = true;

for (int i = 0; i < (int)G[u].size(); i++) {

Edge &e = edges[G[u][i]];

if (d[e.to] > d[u] + e.dist//&& d[u] + e.dist <= c //一定注意具体题目限制 ) {

d[e.to] = d[u] + e.dist;

//if (pp[e.to]) d[e.to] = 0;

//p[e.to] = G[u][i]; //路径

Q.push((HeapNode) {d[e.to], e.to});

}

}

}

} G, H;

int main() {

H.init(n+1);

G.init(n+1);

H.AddEdge(u, v, w);

G.AddEdge(v, u, w);

H.dijkstra(1);

G.dijkstra(n);

}

# // spfa算法求最短路径，允许负环

/\*有两种路，一种走完这条路需要的时间是正的，另一种需要的时间是负的，问有没

\*有这样一条回路，走完整条回路后，需要的时间的和是负的(判负环)

\*判断每个点的入队次数，如果大于N（图中总的点数），就是有负环

\*摘自 POJ 3259 虫洞

\*/

#include<iostream>

#include<vector>

#include<queue>

using namespace std;

#define INF 0x3f3f3f3f

struct node {

int to;

int len;

};

int F,N,M,W;

vector<vector<node> >graph;

//设置相应全局变量后，完全模板函数

bool spfa(int s) {

vector<int>dis(N+1,INF);

dis[s]=0;

vector<int>count(N+1,0);

count[s]=1;

vector<bool>inque(N+1,0);

queue<int>q;

q.push(s);

while(!q.empty()) {

int t=q.front();

q.pop();

inque[t]=false;

for (int i=0; i<graph[t].size(); ++i) {

node &nd=graph[t][i];

if(dis[nd.to]>dis[t]+nd.len) {

dis[nd.to]=dis[t]+nd.len;

if (!inque[nd.to]) {

inque[nd.to]=true;

q.push(nd.to);

if(++count[nd.to]>N)

return false;

}

}

}

}

return true;

}

int main() {

cin>>F;

while(cin>>N>>M>>W) {

graph.assign(N+1,vector<node>());

int s,e,t;

int i;

for(i=0; i<M; ++i) {

cin>>s>>e>>t;

graph[s].push\_back({e,t});

graph[e].push\_back({s,t});

}

for (i=0; i<W; ++i) {

cin>>s>>e>>t;

graph[s].push\_back({e,-t});

}

if(spfa(1))

cout<<"NO"<<endl;

else

cout<<"YES"<<endl;

}

return 0;

}

# //次短路径 算法复杂度O（eloge）

/\*次短路是的长度有可能和最短路一样长。

\*求次短路：Dijkstra的dist数组和vis数组再加一维，松弛的时候讨论

\*当前的路小于最短路，或者大于最短路但小于次短路这两种情况

\*/

const int maxn = 1000 + 5;

const int INF = 0x3f3f3f3f;

struct Node {

int v, c, flag; //节点、耗费、最次标记

Node (int \_v = 0, int \_c = 0, int \_flag = 0) : v(\_v), c(\_c), flag(\_flag) {}

bool operator < (const Node &rhs) const {

return c > rhs.c;

}

};

struct Edge {

int v, cost; //节点、耗费

Edge (int \_v = 0, int \_cost = 0) : v(\_v), cost(\_cost) {}

};

vector<edge>E[maxn];

bool vis[maxn][2]; //0,1分别表示最短和次短

int dist[maxn][2];

void Dijkstra(int n, int s) {

memset(vis, false, sizeof(vis));

memset(dist,0x3f,sizeof (dist));

priority\_queue<Node>que;

dist[s][0] = 0;

que.push(Node(s, 0, 0));

while (!que.empty()) {

Node tep = que.top();

que.pop();

int u = tep.v;

int flag = tep.flag;

if (vis[u][flag])

continue;

vis[u][flag] = true;

for (int i = 0; i < (int)E[u].size(); i++) {

int v = E[u][i].v;

int cost = E[u][i].cost;

if (!vis[v][0] && dist[v][0] > dist[u][flag] + cost) {

dist[v][1] = dist[v][0]; //最短

dist[v][0] = dist[u][flag] + cost;

que.push(Node(v, dist[v][0], 0));

que.push(Node(v, dist[v][1], 1));

} else if (!vis[v][1] && dist[v][1] > dist[u][flag] + cost) {

dist[v][1] = dist[u][flag] + cost; //次短

que.push(Node(v, dist[v][1], 1));

}

}

}

}

void addedge(int u, int v, int w) {

E[u].push\_back(Edge(v, w));

}

int main() {

int n, m, v, w;

while (scanf("%d", &n) != EOF) {

for (int i = 0; i <= n; i++)

E[i].clear();

for (int u = 1; u <= n; u++) {

scanf("%d", &m);

for (int j = 0; j < m; j++) {

scanf("%d%d", &v, &w);

addedge(u, v, w);

}

}

Dijkstra(n, 1);

printf("%d\n", dist[n][1]);

}

return 0;

}

# //最大公共祖先（LCA） 算法复杂度O（nlogn）

/\*给出一棵有边权的树，再加上一条边，Q组询问，求两点之间的最短距离缩短了多少

\*/

const int maxn=100005;

const int maxe=100005;

const int maxdep=20;

struct edge {

int to;

int w; //无权树时可省略

int next;

} e[maxn<<1];

int head[maxn],tot;

int dis[maxn]; //无权树时可省略

int dep[maxn];

int fa[maxn][maxdep];

void init() {

tot=0;

memset(head,-1,sizeof (head));

}

void addedge(int u,int v,int w) {

e[tot].to=v;

e[tot].w=w; //可省略

e[tot].next=head[u];

head[u]=tot++;

}

void dfs(int u,int pre,int d) {

dep[u]=d;

fa[u][0]=pre;

for (int i=1; i<maxdep; ++i)

fa[u][i]=fa[fa[u][i-1]][i-1];

for (int i=head[u]; i!=-1; i=e[i].next) {

int v=e[i].to;

if (v==pre)

continue;

dis[v]=dis[u]+e[i].w; //可省略

dfs(v,u,d+1);

}

}

/\*完全模板函数 返回u，v两个节点的 LCA

\*/

int lca(int u,int v) {

if (dep[u]>dep[v])

swap(u,v);

int hu=dep[u],hv=dep[v];

int tu=u,tv=v;

//找v节点向前第det祖先

for (int det=hv-hu,i=0; det; det>>=1,i++)

if(det&1)

tv=fa[tv][i];

if (tu==tv)

return tu;

//逐步逼近tv,tu的LCA

for (int i=maxdep-1; i>=0; --i) {

if (fa[tu][i]==fa[tv][i])

continue;

tu=fa[tu][i];

tv=fa[tv][i];

}

return fa[tu][0];

}

int main() {

init();addedge(u,v,w);addedge(v,u,w);

dis[1]=0;

dfs(1,1,0);

L1=dis[a]+dis[b]2\*dis[lca(a,b)];

L2=dis[a]+dis[u]2\*dis[lca(a,u)]+dis[b]+dis[v]-2\*dis[lca(b,v)]+w;

L3=dis[a]+dis[v]2\*dis[lca(a,v)]+dis[b]+dis[u]-2\*dis[lca(b,u)]+w;

printf("%d\n",L1-min(L1,min(L2,L3)));

return 0;

}

**C++ sort 函数十分方便，可以对内置类型也可对自定义类型进行快速排序，内置类型的使用比较简单，下面主要讨论自定义类型的排序，一般有如下几种使用方法：**

1.1 重载比较操作符

比如，我们现有一批学生，要根据他们的成绩进行升序排序，成绩如果相等则根据名字升序排序，那么我们可以如下操作：

struct Student{

string name;

int grade;

Student(string name, int grade) : name(name), grade(grade){}

bool operator < (const Student& rhs) const{

return grade < rhs.grade|| (grade == rhs.grade && name < rhs.name);

}

friend void operator << (ostream& output, const Student& s){

output << s.name << " " << s.grade << endl;

}

};

int main()

{

vector<Student> vec;

vec.emplace\_back("Jack", 20);

vec.emplace\_back("John", 30);

vec.emplace\_back("Amy", 20);

vec.emplace\_back("Bill", 90);

**sort(begin(vec),end(vec));**

return 0;

}

**1.2 比较函数**

当然，我们也可以自己写比较函数，实现如下：

bool cmp(const Student& lhs, const Student& rhs){

Return lhs.grade< rhs.grade|| (lhs.grade == rhs.grade && lhs.name < rhs.name);

}

**sort(begin(vec),end(vec), cmp);**

1.3 函数对象

另外一种方式，即构造一个函数对象，抑或叫 functor，其实就是实现了重载 operator() 的一个类，代码如下：

struct Compare{

bool operator()(const Student& lhs, const Student& rhs){

return lhs.grade < rhs.grade

|| (lhs.grade == rhs.grade && lhs.name < rhs.name);

}

};

按如下方式调用：

**sort(begin(vec),end(vec), Compare());**

二、降序

降序排序的方法与升序类似，如果采用比较函数、Lambda 或者比较函数的方式，只需要改一改比较条件就OK了，但是，如果对于Student类，我们定义了 operator < 之后，不想为了降序排序再定义一个 operator > 怎么办？两种办法！

2.1 reverse

升序排序之后，用 reverse 反转即可。

2.2 反向迭代

直接按如下方式调用即可，不用再去重载 operator >

**sort(vec.rbegin(), vec.rend());**

**abs**

**返回一个数的绝对值（int、long long、 double）**

**2. min和max**

用于比较两个同类型（注意是同类型）的最小值和最大值

**在c++11中还提供了多个数一起比较大小**

cout << max({100, 99, 10, 1000}) << endl; // 1000

cout << min({100, 99, 10, 1000}) << endl; // 10

**3. swap**

用于交换两个同类型（注意是同类型）的变量

**4. reserve**

用来翻转指定范围的数组，传参规则和sort第一种传参形式一样

**5. lower\_bound**

用于查找一个有序序列指定范围内的第一个大于等于给定值的位置，返回的是指针或者迭代器，即如果是数组，找不到返回范围的尾部，容器则返回end()

时间复杂度为O(logn)如果是自己定义的类型，记得重写小于符号（那还不如自己手写二分呢）

vector<int> vt = {0,2,4,6,8,10,12};

cout << \*lower\_bound(vt.begin(), vt.end(), 3) << endl; // 4

cout << \*lower\_bound(vt.begin(), vt.end(), 4) << endl; // 4

**6. upper\_bound**

用于查找一个有序序列指定范围内的第一个大于给定值的位置，返回的是指针或者迭代器，即如果是数组，找不到返回范围的尾部，容器则返回end()

时间复杂度为$O(logn)$

vector<int> vt = {0,2,4,6,8,10,12};

cout << \*upper\_bound(vt.begin(), vt.end(), 3) << endl; // 4

cout << \*upper\_bound(vt.begin(), vt.end(), 4) << endl; // 6

**7. unique**

用于将去除一个有序序列中指定范围内的重复元素（实际没有去除）,返回去除后的尾指针

vector<int> vt = {0,0,2,2,4,4,6,6,8,8,10,10,10,12,12};

int index = unique(vt.begin(), vt.end()) - vt.begin();

cout << index << endl; // 7

# 巴什博弈

什么是巴什博弈：只有一堆n个物品，两个人轮流从这堆物品中取物， 规定每次至少取一个，最多取m个。最后取光者得胜。我们称先进行游戏的人为先手，另一个人为后手。

1、如果 ，那么由于一次最多只能取 个，所以，无论先手拿走多少个， 后手都能够一次拿走剩余的物品，后者取胜。

2、如果 ，(r为任意自然数， ),先手要拿走s个物品，如果

后手拿走 个，那么先手再拿走 个，结果剩下

个，以后保持这样的取法，那么先取者肯定获胜。我们得到如下结论:要保持给对手 留下 的倍数，就能最后获胜。必胜态必败态只要 不能整除,那么必然是先手取胜，否则后手取胜。如果我们规定最后取光者输，那么又会如何呢？

% 则后手胜利先手会重新决定策略，所以不是简单的相反的。

例题1.hdu 1846

题意：本游戏是一个二人游戏,有一堆石子一共有n个,两人轮流进行,每走一步可以取 走1…m个石子,最先取光石子的一方为胜,如果游戏的双方使用的都是最优策略，请输出哪个人能赢。

要求：Time Limit: 1000 MS , Memory Limit: 32768 K

思路：最基础的巴什博奕模版题，不多说。

#include<cstdio>

int main()

{

int t,n,m; scanf("%d",&t); while(t--)

{

scanf("%d%d",&n,&m); if(n%(m+1))

printf("first\n"); else

printf("second\n");

}

return 0;

}

例题2.hdu 2147

题意：题目大意：就是有一个游戏，在一个n\*m 的矩阵中起始位置是 （1，m） ，走到终止位置 （n，1） ；游戏规则是只能向左，向下，左下方向走，先走到终点的为获 胜者。

要求：Time Limit: 1000 MS , Memory Limit: 1000 K

思路：我们可以寻找必败态和必胜态，我们假设必败态是P，必胜态是N，那么当

、 时，必胜态必败态矩阵如下：

NNNNNNNNN

PNPNPNPNP

NNNNNNNNN

PNPNPNPNP

NNNNNNNNN

PNPNPNPNP

NNNNNNNNN

PNPNPNPNP

多试几组样例，我们可以很容易的发现，当 和 都为奇数的时候，按照题意，应

输出“What a pity！”，否则输出“Wonderful！”。

#include<cstdio> #include<cstring>

int main()

{

int n,m; while(~scanf("%d%d",&n,&m))

{

if(n==0&&m==0) return 0;

if(n%2==1&&m%2==1) printf("What a pity!\n");

else

printf("Wonderful!\n");

}

return 0;

}

## 威佐夫博奕

简述

威佐夫博弈(Wythoff Game)：有两堆各若干个物品，两个人轮流从某一堆或同时从

两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

分析

我们用 表示两堆物品的数量并称其为局势， 如果甲面对  ，那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是：  、  、  、  、  、  、  、 

、 。 可以看出： , 是未在前面出现过的最小自然数,而

。

满足 ，后手必胜，否则先手必胜。

例题1.hdu 1527

题意：有两堆石子，数量任意，可以不同。游戏开始由两个人轮流取石子。游戏规 定，每次有两种不同的取法，一是可以在任意的一堆中取走任意多的石子；二是可 以在两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。现在给出初 始的两堆石子的数目，如果轮到你先取，假设双方都采取最好的策略，问最后你是 胜者还是败者。

要求：Time Limit: 1000 MS , Memory Limit: 32768 K

思路：基本是威佐夫博奕模版题。

#include<cstdio> #include<cstring> #include<algorithm>

#include<cmath>

using namespace std;

int main()

{

int a,b; while(scanf("%d%d",&a,&b)!=EOF)

{

int aa=max(a,b); int bb=min(a,b); int k=aa-bb;

int temp=(k\*(1+sqrt(5))/2);

if(bb==temp)

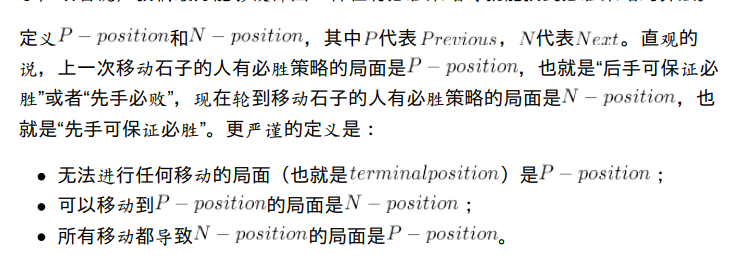
}

return 0;

}

**Nim**博弈

通常的Nim游戏的定义是这样的：有若干堆石子，每堆石子的数量都是有限的，合 法的移动是“选择一堆石子并拿走若干颗（不能不拿）”，如果轮到某个人时所有的 石子堆都已经被拿空了，则判负（因为他此刻没有任何合法的移动）。这游戏看上去有点复杂，先从简单情况开始研究吧。如果轮到你的时候，只剩下一 堆石子，那么此时的必胜策略肯定是把这堆石子全部拿完一颗也不给对手剩，然后 对手就输了。如果剩下两堆不相等的石子，必胜策略是通过取多的一堆的石子将两 堆石子变得相等，以后如果对手在某一堆里拿若干颗，你就可以在另一堆中拿同样 多的颗数，直至胜利。如果你面对的是两堆相等的石子，那么此时你是没有任何必 胜策略的，反而对手可以遵循上面的策略保证必胜。如果是三堆石子……好像已经 很难分析了，看来我们必须要借助一些其它好用的（最好是程式化的）分析方法了，或者说，我们最好能够设计出一种在有必胜策略时就能找到必胜策略的算法。



例题1.hdu 1849

题意：游戏的规则是这样的：

棋盘包含1\*n个方格，方格从左到右分别编号为0，1，2，…，n-1；

m个棋子放在棋盘的方格上，方格可以为空，也可以放多于一个的棋子； 双方轮流走棋；

每一步可以选择任意一个棋子向左移动到任意的位置（可以多个棋子位于同一 个方格），当然，任何棋子不能超出棋盘边界；

如果所有的棋子都位于最左边（即编号为0的位置），则游戏结束，并且规定 最后走棋的一方为胜者。

给出初始棋子状态，输出先手赢，或者后手赢。

#include<cstdio> #include<cstring>

int num[1500]; int sg[1500];

int main()

{

int n; while(~scanf("%d",&n))

{

if(n==0)

return 0; int sum=0;

memset(num,0,sizeof(num)); for(int i=1;i<=n;i++)

{

scanf("%d",&num[i]); sum^=num[i];

}

if(sum)

printf("Rabbit Win!\n"); else

printf("Grass Win!\n");

}

return 0;

}





