Part 1 动态规划

Part 1.1 线性dp

LIS

$O(n^2)$

```
const int N = 1005;

int a[N];
int dp[N];
int res;

int main(){
        int n; cin >> n;
        for(int i = 1, x; i <= n; i ++)  cin >> a[i];
        for(int i = 1; i <= n; i ++){ dp[i] = 1;//初始化为1，因为自己本身一个数就是一个LIS
                for(int j = 1; j < i; j ++) if(a[j] < a[i]) dp[i] = MAX(dp[j] + 1, dp[i]);//如果前面的某一位小于当前的这一位，完全可以把这一位接到前面那一位后面
                res = MAX(res, dp[i]);//维护一下最大答案
        }cout << vec.size() << endl;
        return 0;
}
```

$O(nlogn)$

```
int a[1000005];
vector<int> vec;

int main(){
        int n; cin >> n;
        for(int i = 1, x; i <= n; i ++)  cin >> a[i];
        for(int i = 1, x; i <= n; i ++){
                if(vec.empty() || vec.back() < a[i]) vec.push_back(a[i]);//这个是最大的就塞到队尾
                else                                    vec[lower_bound(vec.begin(), vec.end(), a[i]) - vec.begin()] = a[i];//能变小的话替换一下
                //其实顺序被打乱了，但是在求解size的时候并不影响，因为我们打乱完一遍也就是一个新的子序列，打乱一半是无影响的
        }cout << vec.size() << endl;
        return 0;
}
```

LCS

$O(nlogn)$

```
const int N = 1e6 + 10;

int id[N], n;
vector<int> vec;

int main(){
	cin >> n;
	for(int i = 1, x; i <= n; i ++)  cin >> x, id[x] = i;
	for(int i = 1, x; i <= n; i ++){ cin >> x;
		if(!id[x]) continue;//注意：如果没出现过那就不要加进去了
		if(vec.empty() || vec.back() < id[x]) vec.push_back(id[x]);
		else 					vec[lower_bound(vec.begin(), vec.end(), id[x]) - vec.begin()] = id[x];
	}cout << vec.size() << endl;
	return 0;
}
```

Part 1.2 背包dp

Part 1.2.1 01

```
const int maxV = 1000;
int dp[maxV];

for ( int i = 1; i <= n; i ++ ) {
	for ( int j = V; j >= v[i]; j -- ) {
		dp[j] = MAX(dp[j], dp[j - v[i]] + w[i]);
	}
}
```

Part 1.2.2 完全

```
for ( int i = 1; i <= n; i ++ ) {
	for ( int j = v[i]; j <= V; j ++ ) {
		dp[j] = MAX(dp[j], dp[j - v[i] + w[i]);
	}
}
```

Part 1.2.3 多重

```
vector<int> V, W; //拆分后每一块的物品和价值
inline void manage ( int x, int v, int w ) { // 个数，体积，价值
	int t = 1; // 拆到的块一块包含的物品数
	while(x >= t){
		V.push_back(v * t);
		W.push_back(w * t);
		x -= t;
		t <<= 1;
	}
	if(x) V.push_back(v * x), W.push_back(w * x);
}

...
for(int i = 1; i <= n; i ++ ) {
	int x = inputInt(), v = inputInt(), w = inputInt();
	Manage(x, v, w);
}
for ( int i = 0; i < V.size(); i ++ ) {
	for ( int j = V; j >= V; j ++ ) {
		dp[j] = MAX(dp[j], dp[j - V[i] + W[i]);
	}
}
```

Part 1.2.4 分组

```
for ( int group = 1; group <= groups; group ++ ) {
	for ( int j = m; j >= 0; j -- ) {
		for ( int i = 1; i <= n; i ++ ) {
			if ( gp[i] == group && j >= v[i] ) dp[j] = MAX(dp[j], dp[j - v[i]] + w[i]);
		}
	}
}
```

Part 1.2.5 二维

```
const int maxV = 100, maxM = 100;
int dp[maxV][maxM];

for ( int i = 0; i < n; i ++ ) {
	for ( int j = V; j >= v[i]; j -- ) {
		for ( int k = M; k >= m[i]; k -- ) {
			dp[j][k] = MAX(dp[j][k], dp[j - v[i]][k - m[i]] + w[i]);
		}
	}
}
```

Part 1.2.6 混合

```
int maxV = 1000;
int id[10000]; // 标记，0为01背包，1为完全背包

inline void Manage(){
	for ( int i = 1; i <= N; i ++ ) {
		if() {} // 若多重或者01就二进制拆分一下
		else {} // 若不是就自己开一个，两者做好区分标记
	}
}



for ( int i = 0; i < n; i ++ ) {
	if ( id[i] ) {
		for ( int j = v[i]; j <= V; j ++ ) {
			dp[j] = MAX(dp[j], dp[j - v[i]] + w[i]);
		}
	} else {
		for ( int j = V; j >= v[i]; j -- ) {
			dp[j] = AMX(dp[j], dp[j - v[i]] + w[i]);
		}
	}
}
```

Part 1.2.7 依赖性

```
int v[100][3], w[100][3]; //v[i][j]表示第i套物品的前j件的体积，w[i][j]表示第i套物品的前j件价值
int V;//背包容量
int n;//物品个数

int main()
{
    cin >> V >> n;
    for (int i = 1; i <= n;i++)//优化：主附件并为一个组合，每次遇到附件就将它挪到主件那一组
    {
        int a, b, c;
        cin >> a >> b >> c;//a表示这件物品的体积，a*b表示这件物品的价值，c表示这件物品的主件情况
        if(!c)//若为主件
            v[i][0] = a, w[i][0] = a * b;
        else//若为附件
        {
            if(!w[c][1])//主件后面第一个没有被占，放在第一个
                v[c][1] = a, w[c][1] = a * b;
            else//被占了，放在第二个
                v[c][2] = a, w[c][2] = a * b;
        }
    }

    int dp[32010];
    for (int i = 1; i <= n;i++)
    {
        for (int j = V; j >= v[i][0] && v[i][0]; j--)//稍微优化一下时间，记住：附件是没有自己的i的（地位好低）
        {
            dp[j] = max(dp[j], dp[j - v[i][0]] + w[i][0]);//只选主件
            v[i][0] + v[i][1] > j ?: dp[j] = max(dp[j], dp[j - v[i][0] - v[i][1]] + w[i][0] + w[i][1]);//买主件与第一个附件
            v[i][0] + v[i][2] > j ?: dp[j] = max(dp[j], dp[j - v[i][0] - v[i][2]] + w[i][0] + w[i][2]);//买主件与第二个附件
            v[i][0] + v[i][1] + v[i][2] > j ?: dp[j] = max(dp[j], dp[j - v[i][0] - v[i][1] - v[i][2]] + w[i][0] + w[i][1] + w[i][2]);//买主件与两个附件
        }
    }
    cout << dp[V] << endl;
    return 0;
}
```

Part 1.2.8 泛化

```
inline int getW ( int i ) {
        return w[i] * 10;
}
inline int getV ( int i ) {
        return v[i] * 10;
}
for ( int i = 0; i < n; i ++ ) {
        for ( int j = V; j >= v[i]; j -- ) {
                dp[j] = MAX ( dp[j], dp[j - getV ( i )] + getW ( i ) );
        }
}
```

Part 1.3 区间dp

```
for(int len = 2; len <= n; len ++){
        for(int i = 0; i + len - 1 < n; i ++){
                int j = i + len - 1;
                // 操作
        }
}
```

Part 1.4 树形dp

树的最小点覆盖

```
inline void DFS ( ll x ) {
        if ( num[x] == 0 ) { dp[x][0] = 0; dp[x][1] = 1; return; }
        vis[x] = 1;
        dp[x][0] = 0, dp[x][1] = 1;
        for ( ll i = head[x]; ~i; i = edge[i].nxt ) {
                if ( vis[edge[i].to] ) continue;
                DFS ( edge[i].to );
                dp[x][1] += min ( dp[edge[i].to][0], dp[edge[i].to][1]);
                dp[x][0] += dp[edge[i].to][1];
        }
}
```

Part 1.5 数位dp

数中不出现62的数字个数

```
/*
is_max : 上界，表示 "这一位枚举数字是否达到了当前数"
state : 上一位是几
*/
int dp[len][/*状态*/], b[10]; // dp数组要根据实际情况决定用几维的数组，b数组用来保存数字位
inline int DFS( int pos, int state, bool is_max ) { // 可以根据需要增加state参数的数量
        if ( pos == 0 ) return 1;
        if ( !is_max && ~dp[pos][state] ) return dp[pos][state];
        int end = is_max ? b[pos] : 9; // 如果前面放满了就只能循环到 b[pos]，否则到9
        int res = 0;
        for ( int i = 0; i <= end; i ++ ) {
            if ( /*满足某种条件*/ ) res += DFS (pos - 1, state, is_max && i == end ) ; // 最后一个参数：前面都放满，本位又最大
        }
        if ( !is_max ) dp[pos][state] = res;
        return res;
}
```

Part 2 字符串

Part 2.2 KMP

```
const int N = 1e5 + 10;

class Manacher_Implement{
private:
	int n, m; // a.size = n, b.size = m
	string a, b;
	int nxt[N];

public:
	inline Manacher_Implement () {
		cin >> n >> m;
		cin >> a >> b;
	}
	inline void Get_Next ( ) {
		for ( int i = 0; i < N; i ++ ) nxt[i] = -1;
		int j = -1;
		for ( int i = 0; i + 1 < m; i ++ ) {
			while ( j >= 0 && b[i + 1] != b[j + 1] ) j = nxt[j];
			if ( b[i + 1] == b[j + 1] ) j ++;
			nxt[i + 1] = j;
		}
	}
	inline int KMP ( ) {
		int j = -1;
		for ( int i = -1; i + 1 < n; i ++ ) {
			while ( j >= 0 && a[i + 1] != b[j + 1] ) j = nxt[j];
			if ( a[i + 1] == b[j + 1]) j ++;
			if ( j == m - 1 ) {
				return i - j + 2;
			}
		}
		return -1;
	}
};
```

Part 2.3 Manacher

```
class Manacher_Implement {
private:
	string s;
public:
	inline Manacher_Implement ( string ss ) { s = ss; }
	inline pair<string, int> Manacher () {
		string t = "$#";
		for ( int i = 0; i < s.size(); i ++ ) t += s[i], t += "#";

		int resLen = 0, resCenter = 0;
		int mx = 0, id = 0;
		vector<int> p(t.size(), 0);

		for ( int i = 0; i < t.size(); i ++ ) {
			p[i] = i < mx ? MIN(p[id * 2 - i], mx - i) : 1;
			while ( t[p[i] + i] == t[i - p[i]] ) p[i] ++;
			if ( mx < i + p[i] ) id = i, mx = i + p[i];
			if ( resLen < p[i] )
				resLen = p[i], resCenter = i;
		}
		return {s.substr((resCenter - resLen) / 2, resLen - 1), resLen - 1}; // first = 最长回文串, second = 串长
	}
};
```

Part 3 数学

Part 3.1 整除相关

Part 3.1.1 素数

Part 3.1.1.1 米勒拉宾素数测试

```
ll ksm(ll a,ll b,ll mod) { //快速幂
	ll ans = 1;
	while(b){
		if(b&1) ans=(ans*a)%mod;
		a = (a*a)%mod;
		b>>=1;
	}
	return ans;
}
bool Miller_Rabbin(int n,int a){//米勒拉宾素数测试
	int r = 0, s = n - 1, j;
	if(!(n % a))
		return false;
	while(!(s & 1)) { //查找奇数
		s >>= 1;
		r ++;
	}
	ll k = ksm(a, s, n);
	if(k == 1)
		return true;
	for(j = 0; j < r; j ++, k = k * k % n)
		if(k == n - 1)
		return true;
	return false;
}
bool IsPrime(int n){//判断是否是素数
	int tab[]={2,3,5,7};
	for(int i=0;i<4;i++)
	{
		if(n==tab[i])
		return true;
		if(!Miller_Rabbin(n,tab[i]))
		return false;
	}
	return true;
}
```

Part 3.1.2 最大公约数

Part 3.1.2.1 欧几里得

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

Part 3.1.2.2 扩展欧几里得

```
inline void ex_Gcd ( int a, int b, int &x, int &y ) {
        if ( b == 0 ) { x = 1, y = 0; return a; }
        int d = ex_Gcd ( b, a % b, x, y );
        int tmp = x;
        x = y;
        y = tmp - a / b * y;
        return d;
}
```

Part 3.1.3 欧拉函数

$\phi(n) = \sum_{i=1}^{n} [gcd(i, n) = 1]$

```
const int N = 40010;
bool isprime[N];
int prime[N];
int phi[N];
int cnt = 0;
inline void GET_PHI(){
        phi[1] = 1;
        for(int i = 2; i <= N; i ++){
                if(!isprime[i]) prime[++cnt] = i, phi[i] = i - 1;
                for(int j = 1; j <= cnt && i * prime[j] <= N; j ++){
                        isprime[i * prime[j]] = true;
                        if(i % prime[j] == 0){
                                phi[i * prime[j]] = phi[i] * prime[j];
                                break;
                        } else phi[i * prime[j]] = phi[i] * (prime[j] - 1);
                }
        }
}
```

Part 3.1.4 莫比乌斯函数

$\mu(n) = \begin{cases} 1 & n = 1 \\ (-1)^k & n\text{无平方因数，}n = p1p2p3...pk \\ 0 & n\text{有大于1的平方因数} \end{cases}$

可以简化为：
在n无平方因数时： $\mu(n) = (-1)^{n\text{的不同质因子的个数}}$
其他情况： $\mu(n) = 0$

```
const int maxn = 2005;

bool isprime[maxn];
ll mu[maxn];//Mobius函数表
vector<ll> prime;

inline void Mobius(){//线性筛
        isprime[0] = isprime[1] = 1;
        mu[1] = 1;//特判mu[i] = 1
        for(ll i = 2; i <= maxn; i ++){
                if( !isprime[i] ) mu[i] = -1, prime.push_back(i);//质数的质因子只有自己，所以为-1
                for(ll j = 0; j < prime.size() && i * prime[j] <= maxn; j ++){
                        isprime[i * prime[j]] = 1;
                        if(i % prime[j] == 0) break;
                        mu[i * prime[j]] = -mu[i];//积性函数性质： (i * prime[j])多出来一个质数因数(prime[j])，修正为 (-1) * mu[i]
                }
        }
        //剩余的没筛到的是其他情况，为0
}
```

Part 3.2 同余方程

Part 3.2.1 线性同余方程

解 $ax \equiv c (mod\ b)$ 问题中的x的最小正整数解

```
inline ll ex_Gcd ( ll a, ll b, ll &x, ll &y ) {
        if ( !b ) { x = 1; y = 0; return a; }

        ll d = ex_Gcd ( b, a % b, x, y );
        ll tmp = x;
        x = y;
        y = tmp - a / b * y;
        return d;
}

int main () {
        int a, b, x, y, c;
        cin >> a >> b >> c;
        int gcd = ex_Gcd ( a, b, x, y );
        if ( c % gcd ) puts("No Solution");
        else {
                b /= gcd, c /= gcd;
                cout << ( x * c % b + b ) % b << endl;
        }
}
```

Part 3.2.2 乘法逆元

要求要逆的数和模数互质

Part 3.2.2.1 质模数

费马小定理：
$inv(x) = x^{mod-2}$

Part 3.2.2.2 合模数

欧拉定理：
$inv(x) = x^{\phi(mod)-1}$

扩展欧几里得：
$inv(x)x \equiv 1 (mod\ m)$
$inv(x)x + my = 1$
$\Rightarrow exGcd(x, m, inv(x), y);$

Part 3.2.2.3 线性递推

$inv[1] = 1, \quad inv[i] = (p - p/i) * inv[i\%p]\%p$

Part 3.2.3 扩展中国剩余定理

解：

$$\begin{gathered} x \equiv a[1] (mod\ m[1]) \\ x \equiv a[2] (mod\ m[2]) \\ ... \\ x \equiv a[n] (mod\ m[n]) \end{gathered}$$

这样的问题

```
inline ll Ex_crt(){
	ll X = a[1], M = m[1];                //前一步的X，前一步的lcm
	for (ll i = 2, t, y; i <= n; i ++){
		ll gcd = Ex_gcd(M, m[i], t, y), miDIVgcd = m[i] / gcd; // 求得gcd，并使m[i]约分一下好乘进M里面
		ll c = (a[i] - X % m[i] + m[i]) % m[i];//ax≡c(mod b) 等式右侧的c，并让他变成可行的最小正整数
		if(c % gcd) return -1;
		t = ksc(t, c / gcd, miDIVgcd); // 因为扩欧求得的是等号右侧为gcd时的x解，而此时等号右端为c，需要让X乘上c/gcd个t，此时先给t变了再说


		X += t * M;
		M *= miDIVgcd; //计算LCM
		X = (X % M + M) % M; //保持最小正整数解
	}return X;
}
```

Part 3.2.3 高次同余方程

求解 $a^x \equiv b(mod\ p)$ 问题

```
const int INF = 1e8;

inline int exgcd ( int a, int b, int& x, int& y ) {
	if ( !b ) { x = 1, y = 0; return a; }
	int d = exgcd ( b, a % b, y, x );
	y -= a / b * x;
	return d;
}
inline int bsgs ( int a, int b, int p ) { // 在gcd(a,p)=1时可以直接使用这个函数
	if ( 1 % p == b % p ) return 0;
	int k = sqrt(p) + 1;
	unordered_map<int, int> hash;
	for ( int i = 0, j = b % p; i < k; i ++ ) {
		hash[j] = i;
		j = (ll)j * a % p;
	}
	int ak = 1;
	for ( int i = 0; i < k; i ++ ) ak = (ll)ak * a % p;
	for ( int i = 1, j = ak; i <= k; i ++ ) {
		if (hash.count(j)) return (ll)i * k - hash[j];
		j = (ll)j * ak % p;
	}
	return -INF; //无解
}
inline int exbsgs ( int a, int b, int p ) { // 返回值就是解
	b = ( b % p + p ) % p; // b变成正的
	if ( 1 % p == b % p ) return 0;
	int x, y;
	int d = exgcd ( a, p, x, y );
	if ( d > 1 ) { // a与p不互质，继续递归
		if ( b % d ) return -INF; // 若b不是gcd的倍数
		exgcd (a / d, p / d, x, y); // exgcd求逆元
		return exbsgs ( a, (ll)b / d * x % p % (p / d), p / d) + 1; // 因为本来求的是t-1的最小值，+1得t
	}
	return bsgs(a, b, p);

}
```

Part 3.2.4 欧拉降幂

$$a^b \equiv \begin{cases} a^{b\%\varphi(n)} & \gcd(a,n)=1 \\ a^b & gcd(a,n)\neq 1, b<\varphi(n) \\ a^{b\%\varphi(n)+\varphi(n)} & \gcd(a,n)\neq 1, b\geq\varphi(n) \end{cases} \pmod n$$

```
int main(){
	cin >> a >> b >> n;
	ll len = strlen(b), p = phi(c), up = 0;
	for(ll i = 0; i < len; i ++) up = (up * 10 + b[i] - '0') % p;
	up += p; // 加还是不加取决于 [gcd(a, n) = 1]
	outLL(ksm(a, up, n));
}
```

递归降重幂

求 $k^{k^{k^{\dots k}}} mod\ p$

```
const int N = 1e7 + 10, mod = 1e9 + 7;
int phi[N], isprime[N];
vector<int> prime;

inline void GET_PHI () {
	isprime[0] = isprime[1] = 1, phi[1] = 1;
	for ( int i = 2; i < N; i ++ ) {
		if ( !isprime[i] ) prime.push_back(i), phi[i] = i - 1;
		for ( int j = 0; j < prime.size() && i * prime[j] < N; j ++ ) {
			isprime[i * prime[j]] = 1;
			if ( i % prime[j] == 0 ) {
				phi[i * prime[j]] = phi[i] * prime[j];
				break;
			}else   phi[i * prime[j]] = phi[i] * (prime[j] - 1);
		}
	}
}

inline ll ksm ( ll a, ll b, ll mod ) {
	ll res = 1;
	while ( b ) {
		if ( b & 1 ) res = res * a % mod;
		a = a * a % mod;
		b >>= 1;
	}return res;
}

inline ll get(ll k, ll p){
	if ( p == 2 ) return k & 1;
	return ksm ( k, get(k, phi[p]) + phi[p], p );
}

int main() {GET_PHI();
	ll k, p; cin >> k >> p;
	outLL(get(k, p)); puts("");
}
```

Part 3.3 博弈论

```
inline int SG ( int n ) { // 0必败点，!0必胜点
	bool vis[1100] = {0}; // 标记是否出现过
	for ( int i = 1; i <= m; i ++ ) { // 可拿石子个数
		int tmp = n - i; // 剩余个数
		if ( tmp < 0 ) break;
		if ( sg[tmp] == -1 ) sg[tmp] = SG(tmp); // 记忆化
		vis[sg[tmp]] = 1; // 后继的sg设为出现过
	}
	for ( int i = 0; ; i ++ ) if ( !vis[i] ) return i;
}
```

Part 3.4 组合数学

Part 3.4.1 Lucas定理

```
const int mod = 1e9 + 7;

ll inv[1000010];

inline ll ksm ( ll a, ll b, ll p ) {
	ll res = 1;
	while ( b ) {
		if ( b & 1 ) res = res * a % p;
		a = a * a % p;
		b >>= 1;
	}
	return res;
}

inline void Init () {
	for ( ll i = 1; i < 1000010; i ++ ) inv[i] = ksm(i, mod - 2, mod);
}

inline ll C ( ll n, ll m, ll p ) {
	if ( m > n ) return 0;
	ll res = 1;
	for ( ll i = 1; i <= m; i ++ ) {
		ll a = (n + i - m) % p;
		ll b = i % p;
		res = res * (a * ksm(b, mod - 2, mod) % p) % p;
	}
	return res;
}

inline ll lucas ( ll n, ll m, ll p ) {
	if ( m == 0 ) return 1;
	return C( n % p, m % p, p ) * lucas ( n / p, m / p, p ) % p;
}
```

Part 3.4.2 卡特兰数

$h(n) = h(1) * h(n-1) + h(2) * h(n-2) + ... + h(n-2) * h(2) + h(n-1) * h(1)$
第0项开始为：$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845$
另类递归式： $h(n) = h(n-1) * (4 * n - 2)/(n+1)$
该递推式解为： $h(n) = \frac{C_{2n}^{n}}{n+1}$

下面为卡特兰数n的打表，`a[n][0]`储存长度，后面数是**倒着存的**，输出要倒过来

```
int a[105][100];

inline void Ktl () {
	a[2][0] = 1; a[2][1] = 2;
	a[1][0] = 1; a[1][1] = 1;
	a[0][0] = 1; a[0][1] = 1;
	int len = 1;
	for ( int i = 3; i < 101; i ++ ) {
		int tmp = 0;
		for ( int j = 1; j <= len; j ++ ) {
			int t = (a[i - 1][j]) * (4 * i - 2) + tmp;
			tmp = t / 10;
			a[i][j] = t % 10;
		}
		while ( tmp ) {
			a[i][++len] = tmp % 10;
			tmp /= 10;
		}
		for ( int j = len; j >= 1; j -- ) {
			int t = a[i][j] + tmp * 10;
			a[i][j] = t / (i + 1);
			tmp = t % (i + 1);
		}
		while ( ! a[i][len] ) len --;
		a[i][0] = len;
	}
}

int main () {
	Ktl();
	int n; while ( cin >> n ) {
		cout << n << " : ";
		for ( int i = a[n][0]; i > 0; i -- ) cout << a[n][i];
		puts("");
	}
}
```

Part 3.5 线性代数

Part 3.5.1 矩阵ksm加速

```
const int N = /*...*/;
const int mod = /*...*/;
struct Mat{
	ll m[N][N];//j矩阵
	Mat(int flag = 0){//初始化构造函数
		for(int i = 0; i < N; i ++)
			for(int j = 0; j < N; j ++)
				m[i][j] = flag * (i == j);
	}
	inline Mat Mul(Mat a, Mat b){//矩阵乘法
		Mat res(0);
		for(int i = 0; i < N; i ++)
			for(int j = 0; j < N; j ++)
				for(int k = 0; k < N; k ++)
					res.m[i][j] = (res.m[i][j] + a.m[i][k] * b.m[k][j] % mod) % mod;
		return res;
	}
	inline Mat ksm(Mat a, ll b){//矩阵的ksm实现
		Mat res(1);
		while(b){
			if(b & 1) res = Mul(res, a);
			a = Mul(a, a);
			b >>= 1;
		}
		return res;
	}
};
```

Part 3.5.2 高斯消元

```
const int N = 110;
const double eps = 1e-6; // 有较多比较，使用eps
int n; // 组内一共n个方程
double a[N][N]; // 系数抽出来后的矩阵

inline int Gauss () { // 返回值：0唯一解，1无穷多解，2无解
        int c, r;
        for ( c = 0, r = 0; c < n; c ++ ) { // 一列一列向后枚举
                int t = r; // 维护当前列元素最大的是第几行
                for ( int i = r; i < n; i ++ )
                        if ( fabs(a[i][c]) > fabs(a[t][c]) ) t = i; // 维护
                if ( fabs(a[t][c]) < eps ) continue; // 如果最大的都是0的话，那么就跳过这一列
                for ( int i = c; i <= n; i ++ ) swap(a[t][i], a[r][i]); // 把第t行换到上面
                for ( int i = n; i >= c; i -- ) a[r][i] /= a[r][c]; // 把上面这一行的第一个数约成1
                for ( int i = r + 1; i < n; i ++ ) // 下面所有行的当前列消成0
                        if ( fabs(a[i][c]) > eps ) // 如果是0就不消了
                                for ( int j = n; j >= c; j -- ) a[i][j] -= a[r][j] * a[i][c]; // 这个方程的系数都要减
                r ++;
        }
        if ( r < n ) {
                for ( int i = r; i < n; i ++ )
                        if ( fabs(a[i][n]) > eps ) return 2; // 无解
                return 1; // 有无穷多组解
        }

        for ( int i = n - 1; i >= 0; i -- )
                for ( int j = i + 1; j < n; j ++ )
                        a[i][n] -= a[i][j] * a[j][n]; // 只保留第一个1，后面的都消为0

        return 0; // 有唯一解
}
```

Part 3.5.3 线性基

```
const int N = 1e4 + 5;
const int MaxBit = 60;

ll a[N];
ll d[MaxBit]; // 初始插入线性基
ll l[MaxBit]; // 新简化的线性基
ll cnt;

inline void Init () { // 线性基的初始化
        cnt = 0;
        memset(d, 0, sizeof d);
        memset(l, 0, sizeof l);
}

inline void Insert ( ll x ) { // 将数据x放入集合建立线性基
        for ( ll i = MaxBit; i >= 0; i -- ) {
                if ( (x >> i) & 1 ) {
                        if ( d[i] ) x ^= d[i];
                        else { d[i] = x; break; }
                }
        }
}
```

Part 3.6 多项式

Part 3.6.1 FFT多项式乘法

```
const int N = 3000010;
const double PI = acos(-1.0);

int n, m;
struct Complex{ // 复数定义结构体
        double x, y;
        inline Complex () {} inline Complex ( double _x, double _y ) : x(_x), y(_y) {}
        Complex operator + ( const Complex& t ) const { return Complex(x + t.x, y + t.y); }
        Complex operator - ( const Complex& t ) const { return Complex(x - t.x, y - t.y); }
        Complex operator * ( const Complex& t ) const { return Complex(x * t.x - y * t.y, x * t.y + y * t.x); }
} a[N], b[N];
int rev[N]; // 分治时候的二进制表示对应的结果二进制表示，即反过来了
int bit, tot; // 位数，总个数

inline void fft ( Complex a[], int inv ) {
        for ( int i = 0; i < tot; i ++ ) if ( i < rev[i] ) swap ( a[i], a[rev[i]] ); // 变成正确的分治结果位置
        for ( int mid = 1; mid < tot; mid <<= 1 ) {
                Complex w1 = Complex(cos(PI / mid), inv * sin(PI / mid)); // 表示 w(down: n, up: 1)
                for ( int i = 0; i < tot; i += mid * 2 ) {
                        Complex wk = Complex(1, 0);
                        for ( int j = 0; j < mid; j ++, wk = wk * w1 ) {
                                Complex x = a[i + j], y =  wk * a[i + j + mid];
                                a[i + j] = x + y, a[i + j + mid] = x - y;
                        }
                }
        }
}

int main () {
        cin >> n >> m;
        // 都放到实部里面
        for ( int i = 0; i <= n; i ++ ) cin >> a[i].x;
        for ( int i = 0; i <= m; i ++ ) cin >> b[i].x;
        while ( (1 << bit) < n + m + 1 ) bit ++;
        tot = 1 << bit;
        for ( int i = 0; i < tot; i ++ ) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
        fft ( a, 1 ); fft ( b, 1 ); // 系数表示 -> 点表示
        for  ( int i = 0; i  < tot; i ++ ) a[i] = a[i] * b[i];
        fft ( a, -1 );               // 点表示 -> 系数表示
        for ( int i = 0; i <= n + m; i ++ ) cout << (int)(a[i].x / tot + 0.5) << " ";
        return 0;
}
```

Part 3.6.2 生成函数

1.dp递推求解

```
const int maxn=10000;
int c1[maxn + 1];//c1[i]表示母函数第一个小括号内的表达式中，指数为i的系数
int c2[maxn + 1];//第二个的系数（职业备胎（狗头））
int main(){
	int n;//组合出来n
	while(cin >> n)
	{
		memset(c1, 0, sizeof(c1));
		memset(c2, 0, sizeof(c2));
		for(int i = 0; i <= n; i += elem[1]) c1[i] = 1;//对第一个括号的内容进行赋1

		//=======================================主体
		for(int i = 2; i <= n; i ++)//做n-1趟前两括号合并
		{
		for(int j = 0; j <= n; j ++)//第1个括号
		{
			for(int k = 0; k + j <= n; k += elem[i])//第2个括号，(k+=i)是因为下一个括号指数会+=i
			c2[j + k] += c1[j];//合并后系数相加，要继承一下上一个
		}
		for(int j = 0; j <= n; j ++)//替换一下第一个括号数组，初始化下一个括号数组
		{
			c1[j] = c2[j];
			c2[j] = 0;
		}
		}
		//===========================================
		cout << c1[n] << endl;//此时就剩一个括号，c1[n]即为合并结束后的n次方系数
	}
}
```

2.fft加速

```
const int N = 10210;
const double PI = acos(-1.0);

int n, m, num;
struct Complex { // 复数结构体
	double x, y;
	Complex () {}
	Complex ( double _x, double _y ) : x(_x), y(_y) {}
	Complex friend operator+(Complex a, Complex b) { return {a.x + b.x, a.y + b.y}; }
	Complex friend operator-(Complex a, Complex b) { return {a.x - b.x, a.y - b.y}; }
	Complex friend operator*(Complex a, Complex b) { return {a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x}; }
} a[N], b[N];
int rev[N];
int bit, tot;

inline void FFT ( Complex a[], int inv ) {
	for ( int i = 0; i < tot; i ++ ) if ( i < rev[i] ) swap(a[i], a[rev[i]]);
	for ( int mid = 1; mid < tot; mid <<= 1 ) {
		Complex w1 = {cos(PI / mid), inv * sin(PI / mid)};
		for ( int i = 0; i < tot; i += mid * 2 ) {
			Complex wk = {1, 0};
			for ( int j = 0; j < mid; j ++, wk = wk * w1 ) {
				Complex x = a[i + j], y = wk * a[i + j + mid];
				a[i + j] = x + y, a[i + j + mid] = x - y;
			}
		}
	}
}

int main(){
	num = 300;
	for ( int i = 0; i <= num; i ++ ) a[i].x = 1;
	n = num;
	for ( int k = 2; k <= 17; k ++ ){
		m = num / (k * k) * (k * k);

		while ( (1 << bit) < n + m + 1 ) bit ++;
		tot = 1 << bit;

		// b的重启读入
		for ( int i = 0; i <= m; i ++ ) b[i].x = (i % (k * k) == 0), b[i].y = 0; // k的倍数为1，否则为0。 虚部固定为0
		for ( int i = m + 1; i < tot; i ++ ) b[i].x = 0, b[i].y = 0; // 后面的实部和虚部也要为0

		// rev数组的更新
		for ( int i = 0; i < tot; i ++ ) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1)); // 二进制反转

		FFT(a, 1); FFT(b, 1);
		for ( int i = 0; i < tot; i ++ ) a[i] = a[i] * b[i];
		FFT(a, -1);

		// a的重启读入
		for ( int i = 0; i <= n + m; i ++ ) a[i] = {(double)(int)(a[i].x / tot + 0.5), 0}; // 读入后虚部重启为0
		for ( int i = n + m + 1; i <= N; i ++ ) a[i] = {0, 0}; // 实部虚部重启为0

		n += m; // 第一个多项式扩到n + m
	}

	while ( scanf("%d", &num) == 1 && num ){
		printf("%d\n", (int)(a[num].x + 0.5));
	}
}
```

Part 3.7 莫比乌斯反演

解 $f(k)=\sum_{x=A}^{B}\sum_{y=C}^{D}[gcd(x,y)=k]$

为了满足：
$F(k)=\sum\limits_{n|d}f(d)$

设：
$F(k)=\sum\limits_{x=A}^{B}\sum\limits_{x=C}^{D}[k|gcd(x,y)]$

为使枚举的 x,y 均为 k 的倍数
令 $x'=\frac{x}{k},\quad y'=\frac{y}{k}$ ，我们枚举倍数
则 $F(k)=\sum\limits_{x'=\frac{A-1}{k}}^{\frac{B}{k}}\sum\limits_{y'=\frac{C-1}{k}}^{\frac{D}{k}}=(\lfloor\frac{B}{k}\rfloor-\lfloor\frac{A-1}{k}\rfloor)*(\lfloor\frac{D}{k}\rfloor-\lfloor\frac{C-1}{k}\rfloor)$

根据莫比乌斯反演定理得：
$f(k)=\sum\limits_{k|d}\mu(\frac{d}{k})F(d)$
为了使枚举到的d均为k的倍数
我们设 $d'=\frac{d}{k}\quad H'=\frac{H}{k}$，此时$d=d'k$

则 $f(k)=\sum\limits_{d'=1}^{min(\frac{B}{k},\frac{D}{k})}\mu(d')F(d'k)$

$\because F(d'k)=(\lfloor\frac{B}{d'k}\rfloor-\lfloor\frac{A-1}{d'k}\rfloor)*(\lfloor\frac{D}{d'k}\rfloor-\lfloor\frac{C-1}{d'k}\rfloor$

令 $A'=\frac{A-1}{k},\quad B'=\frac{B}{k},\quad C'=\frac{C-1}{k},\quad D'=\frac{D}{k}$

$\therefore f(k)=\sum\limits_{d'=1}^{min(B',D')}\mu(d')(\left\lfloor\frac{B'}{d'}\right\rfloor-\left\lfloor\frac{A'}{d'}\right\rfloor)(\left\lfloor\frac{D'}{d'}\right\rfloor-\left\lfloor\frac{C'}{d'}\right\rfloor)$

```
const ll maxn = 2e6 + 10;//杜教筛的安全maxn

ll mu[maxn];//Mobius函数表
vector<ll> prime;
ll isprime[maxn];
ll sum[maxn];//mu的前缀和

inline void Mobius(){//线性筛
	mu[1] = 1;//特判mu[i] = 1
	isprime[0] = isprime[1] = 1;
	for(ll i = 2; i < maxn; i ++){
		if(!isprime[i]) mu[i] = -1, prime.push_back(i);//质数的质因子只有自己，所以为-1
		for(ll j = 0; j < prime.size() && i * prime[j] < maxn; j ++){
			isprime[i * prime[j]] = 1;
			if(i % prime[j] == 0) break;
			mu[i * prime[j]] = - mu[i];//积性函数性质： (i * prime[j])多出来一个质数因数(prime[j])，修正为 (-1) * mu[i]
		}
	}
	//剩余的没筛到的是其他情况，为0
	for(ll i = 1; i < maxn; i ++) sum[i] = sum[i - 1] + mu[i];//记录前缀和，为了整除分块
}

inline ll g(ll k, ll x){ return k / (k / x); }//整除分块的r值


map<ll, ll> S;//杜教筛处理出的前缀和


inline ll SUM(ll x){//杜教筛
	if(x < maxn) return sum[x];
	if(S[x]) return S[x];
	ll res = 1;
	for(ll L = 2, R; L <= x; L = R + 1){
		R = MIN(x, g(x, L));
		res -= (R - L + 1) * SUM(x / L);//模数相减会出负数，所以加上一个mod
	}return S[x] = res;
}

inline void solve(){
	ll A, B, C, D, K; cin >> A >> B >> C >> D >> K;
	A = (A - 1) / K, B = B / K, C = (C - 1) / K, D = D / K;
	ll n = MIN(B, D);
	ll res = 0;
	for(ll l = 1, r; l <= n; l = r + 1){
		ll cmp1 = (A / l)? MIN(g(A, l), g(B, l)) : g(B, l);//防止除0
		ll cmp2 = (C / l)? MIN(g(C, l), g(D, l)) : g(D, l); //防止除0
		r = MIN(cmp1, cmp2);//确定块右端点

		res += (sum[r] - sum[l - 1]) * (B / l - A / l) * (D / l - C / l);//公式
	}cout << res << endl;
}

int main () { Mobius();
	ll cass;
	for ( cin >> cass; cass; cass -- ) {
		solve();
	}
	return 0;
}
```

Part 3.8 康托展开

Part 3.8.1 正

```
const int mod = 998244353;
const int maxn = 1e6 + 10;

vector<ll> f;//阶乘表
inline void get_F(){
	f.push_back(1);
	for(int i = 1; i <= maxn; i ++) f.push_back(f.back() * i % mod);
}

ll C[maxn];//树状数组
ll n;//n排列
ll a[maxn];//排列的每一位

inline ll SUM(ll i){//统计1 ~ i的区间和
	ll res = 0;
	while(i) res = (res + C[i]) % mod, i -= LOWBIT(i);
	return res;
}

inline void UPDATE(ll i, ll val){//对下标为i的点单点更新+val
	while(i <= maxn) C[i] += val, i += LOWBIT(i);
}

//x前面有m个比x小的，后面就有x - 1 - m个小的

inline void solve(){
	get_F();
	cin >> n;
	for(int i = 0; i < n; i ++) cin >> a[i];
	ll res = 1;//因为康托展开从12345....开始展，所以初始值为1
	for(int i = 0; i < n; i ++){
		UPDATE(a[i], 1);
		res = (res + (a[i] - 1 - SUM(a[i] - 1)) * f[n - i - 1] % mod) % mod;//累加比a[i]小且未出现的个数*(n - i - 1)的全排列
	}cout << res << endl;
}
```

Part 3.8.2 逆

```
ll n, m;

vector<ll> f;
inline void get_F(){
	f.push_back(1);
	for(int i = 1; i <= n; i ++) f.push_back(f.back() * i);
}

vector<ll> NotUse;

inline void solve(){
	cin >> n >> m; get_F();
	for(int i = 1; i <= n; i ++) NotUse.push_back(i);

	vector<ll> res;
	m --;//同理从1234...开始，所以减一项
	for(int i = 1; i <= n; i ++){
		res.push_back(NotUse[m / f[n - i]]);//放置未出现的第[m / f[n - i]] + 1个
		NotUse.erase(NotUse.begin() + m / f[n - i]);//删除这一个
		m %= f[n - i];//剩余m
	}
	cout << res << endl;
}
```

Part 3.10 筛法

Part 3.10.1 埃氏筛

```
int nprime[maxn];//nprime[i]表示第i个素数
bool isprime[maxn];//isprime[i]表示i是否为素数
inline void Eratosthenes()
{
	//--------------------------------------------initialize
	for( int i = 0;i <= maxn; i ++ ) isprime[i]=true;
	isprime[1] = isprime[0]=false;
	//----------------------------------------------------

	int cnt=0;//记录是第几个素数
	for(int i = 2; i <= maxn; i ++){
		if(isprime[i])//若是素数{
			nprime[++cnt] = i;//记录一下
			for(int j = i * 2; j <= maxn; j += i) isprime[j] = false;//所有倍数为合数
		}
	}
}
```

Part 3.10.2 欧拉筛

```
int nprime[maxn];//nprime[i]表示第i个素数
bool isprime[maxn];//isprime[i]表示i是否为素数
inline void Euler(){
	//----------------------------------------initialize
	for (int i = 0; i <= maxn; i++)
		isprime[i] = true;
	int cnt = 0;
	isprime[1] = false;
	isprime[0] = false;
	//-------------------------------------------------

	for (int i = 2; i <= maxn; i++) {
		if ( isprime[i] ) //若是素数
			nprime[++cnt] = i;//记录一下
		for (int j = 1; j <= cnt; j++){
			if (i * nprime[j] > maxn)//超过的剪掉
				break;
			isprime[i * nprime[j]] = false;//第j个素数的i倍为合数
			if (i % nprime[j] == 0)//去重（后面的不能用这个i筛，因为可以被现在的nprime[j]通过更大的i筛掉）
				break;
		}
	}
}
```

Part 3.10.3 杜教筛

求出来的是前缀和，可以借此获取到区间和或者单个大的积性函数值

```
ll sum[N]; // 某个积性函数的部分前缀和
map<int, ll> mp; // 筛出来然后记忆化的积性函数值
inline ll get_Phi(ll x) {
	if(x < maxn) return sum[x];
	if(mp[x]) return mp[x];

	// 下面的res取值选一个
	/*莫比乌斯函数*/ll res = 1;
	/*欧拉函数*/ll res = x * (x + 1) / 2;

	for(ll l = 2, r; l <= x; l = r + 1) {
		r = x / (x / l);
		res -= (r - l + 1) * get_Phi(x / l);
	}
	return mp[x] = res;
}
```

Part 4 数据结构

Part 4.1 Trie树

成员

```
const int maxn = 1000;//内含字符数
struct Trie{//字典树结构体
	int to;
	/*

	这里可以加入需要的成员变量

	*/
}trie[maxn][/*这里存放字符串内容*/];
int tot_trie;//代表现存的指向型节点
bool end_s[maxn];//判断该字符串是否走完了
```

插入

```
inline void Insert(string s){
	int to = 1;
	for(int i = 0; i < s.size(); i ++){
		int cur_c=s[i]-'a';//这个字符分离为一个数放在结构体第二维
		if(trie[to][cur_c] == 0) trie[to][cur_c] = ++tot_trie;//若为空节点
		to = trie[to][cur_c];//替换
	}
	endd[to] = true;
}
```

查询

```
inline bool Search(string s){
	int to = 1;//检索指针
	for(int i = 0; i < s.size(); i ++){
		to = trie[to][s[i] - 'a'];//替换为下一个指针
		if(to == 0) return false;//指向空节点
	}
	return endd[to];//判断是否刚好走完
}
```

Part 4.3 线段树

这里问题是求区间和

```
inline void push_up(int rt) { //向上更新val
        SegTree[rt].val = SegTree[rt << 1].val + SegTree[rt << 1 | 1].val;//一个父节点的值=两个儿子的值相加
}
inline void push_down(int l, int r, int rt) {//向下继承lazy与更新val
        if(!SegTree[rt].lazy) return ;//lazy为0，没有向下走的必要
        int mid = ( l + r ) >> 1;
        SegTree[rt << 1].val += SegTree[rt].lazy * ( mid - l + 1 );//左儿子val值+父亲的lazy*(内部包含的节点数)（拿回属于自己的东西，因为需要它）
        SegTree[rt << 1 | 1].val += SegTree[rt].lazy * (r - mid);//右儿子val值+父亲的lazy父亲的lazy*(内部包含的节点数)（拿回属于自己的东西，因为需要它）
        SegTree[rt << 1].lazy += SegTree[rt].lazy;//右儿子lazy继承了祖先们的lazy和并加上自己的lazy（因为它的儿子们可还没更新了）
        SegTree[rt << 1 | 1].lazy += SegTree[rt].lazy;//左儿子lazy继承了祖先们的lazy和并加上自己的lazy（因为它的儿子们可还没更新了）
        SegTree[rt].lazy = 0;//清零
}
inline void build_tree(int l, int r, int rt){//构建树
        SegTree[rt].lazy = 0;//初始lazy为0
        if(l == r){ //到达目标节点
                SegTree[rt].val = a[l];
                return ;
        }
        int mid = (l + r) >> 1;
        build_tree(l, mid, rt << 1);//构建左子树
        build_tree(mid + 1, r, rt << 1 | 1);//构建右子树
        push_up(rt);//构建完了向上更新一下值
}
inline void up_date(int a, int b, int c, int l, int r, int rt){//在l~r的总区间下对a[a~b]区间更新一下+c
    if(a > r || b < l) return ;//不沾边
    if(a <= l && b >= r){//二分到的区间是[a~b]的一部分，这个节点更新了，然后递归出口
        SegTree[rt].val += c * ( l - r + 1 );//包含了多少个点就加多少次c
        SegTree[rt].lazy += c;//lazy带一下要加的值
        return ;
    }
    push_down(l, r, rt);//拖着这个节点继续向下更新
    int mid = (l + r) >> 1;
    up_date(a, b, c, l, mid, rt << 1);//往左子树更新
    up_date(a, b, c, mid + 1, r, rt << 1 | 1);//往右子树更新
    push_up(rt);//向上重新更新一下树的节点值
}
inline int query(int a, int b, int l, int r, int rt){//在l~r的总区间下对a[a~b]的和进行查询
    if(a > r || b < l) return 0;//不沾边
    if(a <= l && b >= r) return SegTree[rt].val;//二分到的区间是[a~b]的一部分，这个节点算上去，然后递归出口
    push_up(rt);
    int mid = (l + r) >> 1;
    return query(a, b, l, mid, rt << 1)/*左子树的值*/+query(a, b, mid + 1, r, rt << 1 | 1)/*右子树的值*/;
}
```

Part 4.4 树链剖分

成员

```
const int maxn = 4e4 + 10;
struct edge{ int nxt, to, val; } edge[maxn << 1];
int cnt, head[maxn];
inline void add_edge(int from, int to, int val){edge[++cnt] = {head[from], to, val}; head[from] = cnt;}
int d[maxn];//每个节点的深度
int f[maxn];//每个节点的父节点
int top[maxn];//每个节点的所在重链的链头
int sz[maxn];//每个节点为祖先的子树的节点个数
int son[maxn];//重儿子
ll dis[maxn];//每个节点与地面的距离
int id[maxn]; //id[x]表示这个老序指向新序
int dfn[maxn]; //dfn[x]表示这个新序指向老序
int dfsid; //建立的新序个数
```

两遍DFS获取成员数值

```
inline void DFS1(int x, int fa){
        d[x] = d[fa] + 1, f[x] = fa;//利用父亲，先序处理出d、f
        sz[x] = 1, son[x] = 0;//初始化
        for(int i = head[x]; ~i; i = edge[i].nxt){
                int to = edge[i].to;
                if(to == fa) continue;
                dis[to] = dis[x] + edge[i].val;//先序处理出距离
                DFS1(to, x);
                sz[x] += sz[to];//利用孩子，后序处理出sz
                if(sz[son[x]] < sz[to]) son[x] = to;//后序处理出重儿子
        }
}

inline void DFS2(int x, int y){
        top[x] = y;//先序遍历出top，可根据父节点得出
        dfn[++dfsid] = x; // 老序指向
        id[x] = dfsid; // 新序建立
        if(son[x]) DFS2(son[x], y);//先优先遍历这条重链
        for(int i = head[x]; ~i; i = edge[i].nxt){
                int to = edge[i].to;
                if(to == f[x] || to == son[x]) continue;//如果是父亲或者是重儿子，都不继续进行
                DFS2(to, to);//每个轻儿子都额外开启一条重链
        }
}
```

获取x与y的LCA

```
inline int LCA(int x, int y){
        while(top[x] != top[y]){
                if(d[top[x]] < d[top[y]]) SWAP(x, y);
                x = f[top[x]];
        }if(d[x] > d[y]) SWAP(x, y);
        return x;
}
```

更新x到y的路径点权

```
inline void Change(int x, int y, ll c){
        while(top[x] != top[y]){
                if(d[top[x]] < d[top[y]]) swap(x, y);
                Update(id[top[x]], id[x], c); // 线段树
                x = f[top[x]];
        }
        if(d[x] > d[y]) swap(x, y);
        Update(id[x], id[y], c);
}
```

更新一棵子树的点权

```
...
Update(id[x], id[x] + sz[x] - 1, 1);
...
```

Part 4.5 树状数组

预处理操作

```
int c[N]; // 树状数组
inline int lowbit ( int x ) { return x & (-x); }
inline void makeC ( int i ) {
        int res = 0;
        int num = i + 1 - lowbit(i);
        while ( i >= num ) res += a[i], i --;
        c[i] = res;
}
```

计算前缀和

```
inline int Sum ( int i ) {
        int res = 0;
        // 下面是表示a[1~n]下的前缀和统计
        while ( i > 0 ) res += c[i], i -= lowbit(i);
        return res;
}
```

单点更新

```
inline void Update ( int i, int val ) {
        while ( i <= N ) c[i] += val, i += lowbit(i);
}
```

区间更新

可以让c[i]作为差分数组进行更新，这题可以用线段树了。。。

Part 4.6 ST表

成员： $st[i][j]$ 表示以 i 为起点，长度为 2^i 的区间

```
const int N;
int n;
int a[N], st[N][25];
```

例：求区间MAX

建表

```
inline void BuildST () {
        int k = 32 - __builtin_clz(n) - 1;
        for ( int j = 1; j <= k; j ++ ) {
                ( int i = 1; i + (1 << j) - 1 <= n; i ++ ) {
                        st[i][j] = MAX( st[i][j - 1], st[i + (1 << (j - 1))][j - 1] );
                }
        }
}
```

查询

```
inline int Query ( int l, int r ) {
        int k = 32 - __builtin_clz(r - l + 1) - 1;
        return MAX ( st[l][k], st[r - (1 << k) + 1][k] );
}
```

Part 5 图论

Part 5.1 最短路问题

Part 5.1.1 DIJKSTRA

堆优化

```
const int N = 4100;

struct Edge {
        int nxt, to, val;
        inline Edge () {};
        inline Edge (int _nxt, int _to, int _val) : nxt(_nxt), to(_to), val(_val) {};
} edge[N];
struct Node {
        int x, dis;
        inline Node ( int _x, int _dis ) : x(_x), dis(_dis) {};
        friend bool operator < ( Node a, Node b ) { return a.dis > b.dis; }
};
int cnt, head[N];
int dis[N], vis[N];
int n, m;

inline void Init ( ) {
        cnt = 0;
        for ( int i = 0; i < N; i ++ )
                head[i] = -1,
                dis[i] = INF,
                vis[i] = 0;
}

inline void Add_Edge ( int from, int to, int val ) {
        edge[ ++cnt ] = Edge ( head[from], to, val );
        head[from] = cnt;
}

inline void DIJKSTRA ( ) {
        dis[1] = 0;
        priority_queue<Node> pque;
        pque.push ( Node( 1, dis[1] ) );
        while ( pque.size() ) {
                Node stt = pque.top(); pque.pop();
                if ( vis[stt.x] ) continue; vis[stt.x] = 1;
                for ( int i = head[stt.x]; ~i; i = edge[i].nxt ) {
                        if ( dis[edge[i].to] > dis[stt.x] + edge[i].val )
                                dis[edge[i].to] = dis[stt.x] + edge[i].val,
                                pque.push(Node(edge[i].to, dis[edge[i].to]));
                }
        }
}
```

Part 5.1.2 Floyd

```
for(int k = 1; k <= n; k ++)        //插入点在外层循环
        for(int i = 1; i <= n; i ++)
                for(int j = 1; j <= n; j ++)
                        dis[i][j] = MIN(dis[i][j], dis[i][k] + dis[k][j]);
```

Part 5.1.3 Bellman_Ford

```
for(int k = 1; k <= n - 1; k ++)
        for(int i = 1; i <= m; i ++)
                dis[v[i]] = MIN(dis[v[i]], dis[u[i]] + w[i]);

//正常情况下，做了n-1趟松弛操作后整个图的最优解就稳定了，如果还有可以松弛的，就是负环
int flag = 0;
for(int i = 1; i <= m; i ++)
        if(dis[v[i]] > dis[u[i]] + w[i]) flag = 1;
if(flag) ... //有负环的情况
```

Part 5.2 强连通分量

统计多少对点处在同一连通分量中

```
const int maxn = 2e5 + 10;

int x[maxn], y[maxn];
int n, m;

struct Edge{//前向星中，前标为0表示正图，1表示反图
        int nxt, to;
}edge[2][maxn];
int head[2][maxn];
int cnt[2];

int nod[maxn];//集合代表（类似并查集）
vector<int> inv;//DFS1得到的反序列
map<int, int> vis;//记录是否使用过
map<int, ll> num;//记录每个集合有多少个元素

inline void Init(){
        vis.clear();
        inv.clear();
        num.clear();
        for(int i = 0; i <= m; i ++) head[1][i] = head[0][i] = -1;
        for(int i = 0; i <= n; i ++) nod[i] = i;
        cnt[1] = cnt[0] = 0;
}
inline void add_edge(int id, int from, int to){
        edge[id][++cnt[id]] = {head[id][from], to};
        head[id][from] = cnt[id];
}
inline void DFS1(int x){//1.正向遍历出反向（当成无向图的）遍历顺序
        vis[x] = 1;
        for(int i = head[0][x]; ~i; i = edge[0][i].nxt){//正向一个集合一个集合地遍历一遍
                if(!vis[edge[0][i].to]) DFS1(edge[0][i].to);
        }inv.push_back(x);//后序遍历得到反遍历序列
}
inline void DFS2(int x, int y){
        vis[x] = 0;
        nod[x] = y;//压入集合
        for(int i = head[1][x]; ~i; i = edge[1][i].nxt){//反向对之前遍历过的集合进行遍历
                if(vis[edge[1][i].to]) DFS2(edge[1][i].to, y);
        }
}
inline void solve(){
        scanf("%d%d", &n, &m);Init();
        for(int i = 0; i < m; i ++){
                scanf("%d%d", &x[i], &y[i]);
                add_edge(0, x[i], y[i]);
                add_edge(1, y[i], x[i]);
        }

        for(int i = 0; i < m; i ++){
                if(!vis[x[i]]) DFS1(x[i]);
        }
        for(int i = inv.size() - 1; i >= 0; i --){//按正序后的序列进行遍历
                if(vis[inv[i]]) DFS2(inv[i], inv[i]);
        }

        map<int, int> used;
        for(int i = 0; i < m; i ++){//计算一个集合内的元素个数
                if(!used[x[i]]) used[x[i]] = 1, num[nod[x[i]]] ++;
        }

        ll res = 0;
        for(auto &i : num){//得到C(k, 2)的累加和
                res += i.second * (i.second - 1) / 2;
        }printf("%lld\n", res);
}
```

Part 5.3 点分治

成员

```
const int N = 1e5 + 10;
const int K = 1e8 + 10;

struct Edge {
        int nxt, to, val;
        inline Edge () {}
        inline Edge ( int _nxt, int _to, int _val ) : nxt(_nxt), to(_to), val(_val) {}
} edge[N << 1];
int tot, head[N];

inline void Add_Edge ( int from, int to, int val ) {
        edge[++tot] = Edge(head[from], to, val);
        head[from] = tot;
}

// rt记录重心, sum记录当前树大小, cnt是计数器
int n, m, rt, sum, cnt;
// tmp记录算出的距离,  siz记录子树大小, dis[i]为rt与i之间的距离
// maxp用于找重心, q用于记录所有询问
int tmp[N], siz[N], dis[N], maxp[N], q[105];
// judge[i]记录在之前子树中距离i是否存在, ans记录第k个询问是否存在, vis记录被删除的节点
bool judge[K], ans[105], vis[N];
```

找重心

```
inline void Get_Rt ( int u, int f ) {
	siz[u] = 1, maxp[u] = 0; // maxp初始化魏最小值
	// 遍历所有儿子，用maxp保存最大大小的儿子的的大小
	for ( int i = head[u]; ~i; i = edge[i].nxt ) {
		int v = edge[i].to;
		if ( v == f || vis[v] ) continue; // 被删掉的也不要算
		Get_Rt ( v, u );
		siz[u] += siz[v];
		if ( siz[v] > maxp[u] ) maxp[u] = siz[v]; // 更新maxp
	}
	maxp[u] = max ( maxp[u], sum - siz[u] ); // 考虑u的祖先节点
	if ( maxp[u] < maxp[rt] ) rt = u; // 更新重心（最大子树大小最小）
}
```

计算各节点与根节点之间的距离并全部存在tmp里面

```
inline void Get_Dis ( int u, int f ) {
	tmp[cnt ++] = dis[u];
	for ( int i = head[u]; ~i; i = edge[i].nxt ) {
		int v = edge[i].to;
		if ( v == f || vis[v] ) continue;
		dis[v] = dis[u] + edge[i].val;
		Get_Dis ( v, u );
	}
}
```

处理经过根节点的路径

```
// ! 注意judge数组要存放置钱子树里面存在的路径长度，排除折返路径的可能
inline void solve ( int u ) {
	queue<int> que;
	for ( int i = head[u]; ~i; i = edge[i].nxt ) {
		int v = edge[i].to;
		if ( vis[v] ) continue;
		cnt = 0;           // 计数器重置
		dis[v] = edge[i].val;
		Get_Dis ( v, u ); // 处理出所有距离
		for ( int j = 0; j < cnt; j ++ ) {           // 遍历所有距离
			for ( int k = 0; k < m; k ++ ) {      // 遍历所有询问
				if ( q[k] >= tmp[j] )         // 如果询问大雨单条路径长度，那就有可能存在
					ans[k] |= judge[q[k] - tmp[j]]; // 如果能用两条路径拼出来，那就存在
			}
		}
		for ( int j = 0; j < cnt; j ++ ) { // 把unsaid单条路径长度标上true，供下个子树使用
			que.push ( tmp[j] );
			judge[tmp[j]] = true;
		}
	}
	while ( que.size() ) judge[que.front()] = false, que.pop(); // 清空judge数组，不要用memset
}
```

分治

```
inline void Divide ( int u ) {
	vis[u] = judge[0] = true; // 删除根节点
	solve(u);                 // 计算经过根结点的路径
	for ( int i = head[u]; ~i; i = edge[i].nxt ) { // 分治剩余部分
		int v = edge[i].to;
		if ( vis[v] ) continue;
		maxp[rt = 0] = sum = siz[v]; // 把重心置为0，并把maxp[0]置为最大值
		Get_Rt ( v, 0 );
		Get_Rt ( rt, 0 );       // 与主函数相同，第二次更新siz大小
		Divide ( rt );
	}
}
```

主函数

```
int main () {
	memset ( head, -1, sizeof(head) ); tot = 0;
	cin >> n >> m;
	for ( int i = 1; i < n; i ++ ) {
		int u, v, w; cin >> u >> v >> w;
		Add_Edge ( u, v, w );
		Add_Edge ( v, u, w );
	}
	for ( int i = 0; i < m; i ++ ) cin >> q[i];
	maxp[0] = sum = n; // maxp[0]置为最大值（一开始rt=0）
	Get_Rt ( 1, 0 );   // 找重心
	// 此时siz数组存放的是以1为根的各树大小，需要以找出的重心为根重复
	Get_Rt ( rt, 0 );
	Divide ( rt );  // 找好重心开始分治
	for ( int i = 0; i < m; i ++ ) {
		if ( ans[i] ) puts("AYE");
		else puts("NAY");
	}
}
```

Part 5.4 差分约束

目的判断

$\leq$ 求最短路

$\geq$ 求最长路

过程标准化

如果求两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路

如果求两个变量差的最小值，那么需要将所有不等式转变成">="的形式，建图后求最长路

如果出现 $A-B=C$ 这样的等式，变化成两个不等式 $A-B\geq C$ 和 $A-B\leq C$

如果变量都是整数域上的，那么遇到 $A-B<C$ 这样的不带等号的不等式，需要变化成带等号的不等式例如 $A-B\leq C-1$

Part 6 杂项

Part 6.1 c++输入流加速

```
std::ios::sync_with_stdio(false);
```

Part 6.2 Java大数

```
import java.math.BigInteger;

...
BigInteger a, b;

a = BigInteger.valueOf(); // 数值设置
a = a.add(b); //加法
a = a.subtract(b); // 减法
a = a.multiply(b); // 乘法
a = a.divide(b); // 除法
a = a.mod(b); // 取余

a.compareTo(b) = /*1: a>b, 0: a=b, -1: a<b*/;
...
```

Part 6.3 离散化

```
struct node {
        int val;//原始数值
        int order;//原始下标
        friend bool operator < ( node a, node b ) {
                return a.val < b.val;
        }
}a[100001];
int b[100001], cnt;

...
sort(a + 1, a + 1 + N);
b[a[1].order] = 1;
for(int i = 2, cnt = 1; i <= N; i ++)
    if(a[i].val == a[i - 1].val) b[a[i].order] = cnt;
    else b[a[i].order] = ++cnt;
...
```