字符串

马拉车

介绍

统计字符串中回文串的数量和长度及位置，时间复杂度$O(n)$。

原理

根据回文串的性质，获取当前位置对称的点的属性（已经计算过的回文串的长度）来减少当前位置的重复运算。

实现

我们需要记录回文串半径在右端最大的情况下的回文区间（就是以一个位置(mid)为中心，最大半径是r,那么区间的就是$(mid-r, mid+r)$），因为我们型尽可能地让我们接下来的字符在我们能已经计算过的回文串区间当中，这样就能够利用当前点的对称点的信息。因为回文串有奇数或者偶数的情况，所以有两种方式来处理。这里先讲奇数，因为偶数的情况也能够转化为奇数的情况进行计算。

回文串为奇数的情况

我们次先判断当前的点是否在我们当前的最靠右的回文区间当中，如果不在，那么初始k就是1，否则我们就可以获取其对称点的最大半径，然后和这个对称点和左边界的长度取一个min，因为我们所获取的信息不不能超出这个区间，因为这个区间的右侧是未知的。接着我们通过$while$循环找到当前位置的最大半径，然后重新更新我们的最大值r。

```
for(int i = 0, l = 0, r = -1 ; i < s.size() ; ++ i){
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
    while( i-k >= 0 && i + k < s.size() && s[i-k] == s[i+k]){
        k ++ ;
    }
    d1[i] = k -- ;
    if(i + k > r){
        r = i + k ;
        l = i - k ;
    }
}
```

回文串为偶数数的情况

其实也就是取中心点和左端点不太一样。

```
for(int i = 0, l = 0, r = -1 ; i < s.size() ; ++ i){
    int k = (i > r) ? 0 : min(d2[l + r - i+1], r - i + 1);
    while( i - k - 1 >= 0 && i + k < s.size() && s[i-k-1] == s[i+k]){
        k ++ ;
    }
    d2[i] = k -- ;
    if(i + k > r){
        r = i + k ;
        l = i - k - 1 ;
    }
    ans = max(ans,d2[i]*2) ;
}
```

关于偶数如何转换为奇数的情况

我们可以每个字符之间和两侧插入一个字符串当中都不存在的字符，$abba$ ->$*a*b*b*a*$，这样我们就处理成了奇数的情况,从0开始，奇数位置是奇数长度的回文串，半径为 $d/2$,包含中间的数，偶数位置是偶数长度的回文串,半径为 $d/2$。

```
int d[N] ;
string s ;
void ma(string ss){
    for(int i = 0, l = 0, r = -1 ; i < ss.size() ; ++ i){
        int k = (i > r) ? 1 : min(d[l + r - i], r - i + 1);
        while( i-k >= 0 && i + k < ss.size() && ss[i-k] == ss[i+k]){
            k ++ ;
        }
        d[i] = k -- ;
        if(i + k > r){
            r = i + k ;
            l = i - k ;
        }
    }
}
void solve(){
    cin >> s ;
    string t = "*";
    for(auto e : s){
```

```
        t.push_back(e) ;
        t.push_back('*') ;
    }
    ma(t) ;
    //从0开始，奇数位置是奇数长度的回文串，半径为 d / 2,包含中间的数
    //偶数位置是偶数长度的回文串,半径为 d / 2
    cout << endl ;
}
```

KMP

介绍

$O(n)$时间复杂度匹配目标字符串的算法。

关于前缀函数

我觉得在KMP之前先讲一下前缀函数会更好一些，所谓前缀函数就是指当前字符串s的前i个字符组成的子串中前缀和后缀相同的字串的最大长度。举例来说：

> $s = abcab$，那么前缀函数$f[5] = 2$，相同的子串为ab(我这里下标从1开始)。

前缀函数在KMP中的应用

我们观察一个这样的字符串$abcabd$，假如它在与母串匹配的过程中在最后一个字符d匹配失败了，按照朴素算法我们会重头开始，但是通过观察可以看出直接让当前母串的字符和第3个字符c匹配时更为合适的，因为我们发现在前5个字符当中存在相同的前缀和后缀子串，这实际上是可以重复利用的，而这个新的回溯位置我们也正好能够通过前缀函数得到。

初始化next数组

```
int ne[N] ;
void init(){
    for(int i = 2 , j = 0; i < s.size() ; ++i){
        while( j && s[i] != s[j+1]) j = ne[j] ;
        if(s[i] == s[j+1]) ++ j ;
        ne[i] = j ;
    }
}
```

匹配字符串

```
for(int i = 0, j = 0 ; i < t.size() ; ++ i){
    while( j && t[i] != s[j+1]) j = ne[j] ;
    if(t[i] == s[j+1]) j ++ ;
    if(j == s.size()-1){
        cout << i - s.size() + 2 << " ";
        j = ne[j] ;
    }
 }
```

时间复杂度为什么是$O(n)$

我们可以发现j的增加只会来自`if(t[i] == s[j+1]) j ++ ;`这一步，(j)的最大值就是母串长度，假设$while$每次循环只会让j减少1，那么总共也只会减少母串长度次，所以总的时间复杂度就是$O(n)$。

KMP自动机（单模式匹配）

题目链接

$f[i][j]$ 表示的是密码已经生成了 i 位，第 i 位处于 j 状态（j 状态可以看作构造串 $[i-j+1,i]$ 与模板串的 $[1,j]$ 已经匹配）的情况，第 $i+1$ 个位置可能为 26 个英文字母中的任意一个，每一种英文字母都能将当前的状态导向相同或者不相同的状态

在第计算 $i+1$ 位置的状态时我们就可以通过上一级 i 计算出来的状态进行转移

对于状态 $0,1,2,\ldots,m-1,m$，其中 0 为**不接受状态**，m 为**完全接受状态**。所以最后的 答案就是 $\sum_{i=0}^{m-1} dp[n][i]$。除了 m 的所有状态的和。

```
#include <bits/stdc++.h>
#define IOS  ios::sync_with_stdio(0); cin.tie(0);cout.tie(0)
#define endl '\n'
#define int long long
using namespace std;
const int N = 1e3 + 10;
const int INF = 0x3f3f3f3f3f3f3f3f;
const int MOD = 1e9 + 7; // 全一才位1
```

```cpp
int T = 1;
int ans = 0 ;
int n, m, k;
char s[N] ;
int ne[N] ;
int f[N][N] ;
void solve(){
    cin >> n ;
    cin >> s + 1 ;
    m = strlen(s+1) ;
    for(int i = 2 , j = 0 ; i <= m ; ++ i){//kmp构造nex数组
        while(j && s[i] != s[j+1]) j = ne[j] ;
        if(s[i] == s[j+1]) j ++ ;
        ne[i] = j ;
    }
    f[0][0] = 1 ;
    for(int i = 0 ; i < n ; ++ i){
        for(int j = 0 ; j < m ; ++ j){//枚举已经匹配了多少个模板串的字符串
            for(char c = 'a' ; c <= 'z' ; ++ c){//枚举下一个匹配的字符
                int ptr = j ;
                while(ptr && s[ptr + 1] != c) ptr = ne[ptr] ;
                if(s[ptr + 1] == c) ptr ++ ;//寻找可以转移的下一个状态
                f[i+1][ptr] = (f[i+1][ptr] + f[i][j]) % MOD ;//状态转移
            }
        }
    }
    ans = 0 ;
    for(int i = 0 ; i < m ; ++ i){//统计除m以外的所有状态
        ans = (ans + f[n][i]) % MOD ;
    }
    cout << ans << endl ;
}
signed main(){
    IOS;
    while (T--){
        solve();
    }
    return 0;
}
```

AC 自动机（多模式匹配）

介绍

多模式匹配算法，给定 n 个模式串和一个主串，查找有多少个模式串在主串中出现过。

算法流程

- 构造 $Trie$ 树
  我们先用 n 个模式串构造一颗 $Trie$ 树 。
  $Trie$ 中的一个节点表示一个从根到当前节点的字符串。
  根节点表示空串。
  如果节点是个模式串就打个标记。
- 构造 AC 自动机
  在 $Trie$ 上构建两类边：**回跳边**和**转移边**
- 扫描主串匹配

回跳边的构建

$nec[v]$ 存节点 v 的**回跳边**的终点。
回跳边指向父节点（u）的回跳边所指节点的儿子。 $(v, u, ne[u], ch[][])$ 构造成一个四边形。
回跳边所指向节点一定是当前节点的最长后缀。

转移边的构建

$ch[u][i]$ 存节点 u 的树边的终点。
$ch[u][i]$ 存节点 u 的转移边的终点。

转移边指向当前节点的**回跳边**所指节点的儿子 。
三个点 $(u, ne[u], ch[][])$ ，构成三角形。
转移边所指向的节点一定是当前节点的**最短路**。

实现

> 用 BFS 构建 AC 自动机
> 初始化，吧根节点的儿子们全部入队。只要队不空，节点 u 出队。
> 枚举 u 的 26 个儿子，

- 若儿子存在，建立**回跳边**，并且让儿子入队
- 若儿子不存在，则父亲自建**转移边**。

建 $Trie$ 树

```
void insert(string s){
    int p = 0 ;
    for(int i = 0 ; i < s.size() ; ++ i){
        int j = s[i] - 'a' ;
        if(!ch[p][j]) ch[p][j] = ++idx ;
        p = ch[p][j] ;
    }
    vec.push_back(p) ;
    cnt[p] ++ ;
}
```

构造 AC 自动机

```
void build(){
    queue<int> q ;
    for(int i = 0 ; i < 26 ; ++ i){
        if(ch[0][i]) q.push(ch[0][i]) ;
    }
    while(q.size()){
        int t = q.front() ;
        q.pop() ;
        for(int i = 0 ; i < 26 ; ++ i){
            int v = ch[t][i] ;
            if(v) ne[v] = ch[ne[t]][i] , q.push(v) ;//子节点存在则构造回跳边
            else ch[t][i] = ch[ne[t]][i] ;//子节点不存在构造转移边
        }
    }
}
```

查询

```
int querry(string s){
    int res = 0 ;
    for(int  k = 0, i = 0  ; k < s.size() ; ++ k ){//i沿着树边或者转移边走
        i = ch[i][s[k] - 'a'] ;//
        for(int j = i ; j && ~cnt[j] ; j = ne[j]){
            res += cnt[j] ;
            cnt[j] = -1 ;//只统计一次，计算有多少种不同的模式串
        }
    }
    return res ;
}
```

总结

	KMP	AC自动机
匹配	**单模式匹配**算法	**多模式匹配**算法
形态	**链上**构造自动机	Trie 树上构造自动机
数据维护	**回跳边** ne[]	**回跳边** ne[] 树边，**转移边** ch[][]
算法流程	双指针建边、双指针匹配	BFS 建边、双指针匹配
时间复杂度	$O(n+m)$	$O(26n+m)$

字典树Trie

介绍

Trie是一种能够快速插入和查询字符串的多叉树结构。

节点的编号各不相同，根节点编号是 0 ，其他节点用来标识路径，还可以标记单词插入的次数。边表示字符。

Trie维护字符串的集合，支持两种操作：

1. 向集合中插入一个字符串，`void insert(char * s)`
2. 在集合中查寻一个字符串，`int querry(char * s)`

建字典树

儿子数组 `ch[p][j]` 存储从节点 **p** 沿着 **j** 这条边走到的子节点。

- 边为 26 个英文字母对应映射位 $0-25$。
- 每个节点最多会有 26 个分叉。

计数数组 `cnt[p]` 存储以节点 **p** 结尾的单词的插入次数。

节点编号 **idx** 用来个节点编号。

1. 空 **Trie** 仅有一个根节点，编号为 0；
2. 从根节点开始插，枚举字符串的每个字符；
   如果没有儿子，则 **p** 指针走到儿子，
   如果没有儿子，则先创建儿子，p 指针再走到儿子。
3. 在单词结束点记录插入次数。

```
char ch[N][26] ;
int cnt [N] ;
int idx ;
char s[N] ;
void insert(char * s){
    int p = 0 ;
    for(int i = 0 ; s[i] ; ++ i){
        int j = s[i] - 'a' ;
        if(!ch[p][j]) ch[p][j] = ++ idx ;
        p = ch[p][j] ;
    }
    cnt[p] ++ ;
}
```

查询

1. 从根开始查，扫描字符串。
2. 有字母 `s[i]`，则走下来，能走到词尾，则返回插入次数。
3. 无字母 `s[i]`，返回 0；

```
int querry(char * s){
    int p = 0 ;
    for(int i = 0 ; s[i] ; ++ i){
        int j = s[i] - 'a' ;
        if(!ch[p][j]) return 0 ;//说明该单词不存在
        p = ch[p][j] ;
    }
    return cnt[p] ;
}
```

相关题目

E. Collapsing Strings

字符串Hash

```
const int maxn = 5000010;
char s1[maxn];
ll ha1[maxn], pow1[maxn], rha1[maxn];
struct string_hash{
    const int nn;
    const ll Base1 = 29, MOD1 = 1e9 + 7;
    string_hash(int n1) : nn(n1)
    {
        pow1[0] = 1;
        for(int i = 1; i <= nn; i++)
            pow1[i] = pow1[i - 1] * Base1 % MOD1;
        for(int i = 1; i <= nn; i++)
        {
            ha1[i] = (ha1[i - 1] * Base1 + s1[i - 1]) % MOD1;
            rha1[i] = (rha1[i - 1] * Base1 + s1[nn - i]) % MOD1;
        }
    }
    int get_hash(int l,int r)
    {
```

```
        int res1 = ((ha1[r] - ha1[l - 1] * pow1[r - l + 1]) % MOD1 + MOD1) % MOD1;
        return res1;
    }
    int get_rhash(int l, int r)
    {
        int res1 = ((rha1[n - l + 1] - rha1[n - r] * pow1[r - l + 1]) % MOD1 + MOD1) % MOD1;
        return res1;
    }
    bool same(int l1, int r1, int l2, int r2)
    {
        return get_hash(l1, r1) == get_hash(l2, r2);
    }
    ll add(ll aa,ll bb)
    {
        ll res = (aa + bb) % MOD1;
        return res;
    }
    ll mul(ll aa, ll kk) //aa *= Base的k次方
    {
        ll res = aa * pow1[kk] % MOD1;
        return res;
    }
    ll pin(int l1, int r1, int l2, int r2) //拼接字符串 r1 < l2  ss = s1 + s2
    {
        return add(mul(get_hash(l2, r2), r1 - l1 + 1), get_hash(l1, r1));
    }
};

```

Z函数 （扩展KMP）

定义

对于一个长度为 n 的字符串 s，定义函数 $z[i]$ 表示 s 和 $s[i,n-1]$ （即以 [s[i]] 开头的后缀）的最长公共前缀LCP的长度，则 z 被称为 s 的 Z 函数。特别地，$z[0]=0$。

```
vector<int> z_function(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && z[i - l] < r - i + 1) {
      z[i] = z[i - l];
    } else {
      z[i] = max(0, r - i + 1);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    }
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

数据结构

线段树

可以用用来维护区间信息（区间和，区间最值，区间GCD等），可以在$loogn$的时间内执行**区间查询**和**区间查询**。

线段树中每个**叶子节点**存储元素本身，**非叶子节点**存储区间内的元素的统计值。

1. 节点数组$tr[]$

结构体包含三个变量：l,r,sum。，分别是区间的左右端点区间和。

2. 递归建树

父亲节点的编号是**p**，左孩的编号是**2*p**，右孩是**2*p+1**。

```
#define lc p << 1
#define rc p << 1 | 1
struct node{
    int l, r, sum ;
}tr[N*4] ;

void build(int p, int l, int r){
    tr[p] = {l,r,w[l]} ;//如果不是叶子节点那么在之后的计算会覆盖
    if(l == r) return ;
    int m = l + r >> 1 ;
    build(lc,l,m);
    build(rc,m+1,r);
    tr[p].sum = tr[lc].sum + tr[rc].sum ;
}
```

线段树为什么空间大小开 $4*n$

如果叶子节点的数量恰好是2的幂（2^m），那么空间开$2*m-1$即可。

但如果不是2的幂，是2^m+k，那么节点的最大编号就是

$$(4-\frac{2}{k})\times n-(4k-3)<4k(k=2,4,8\ldots)$$

点修改

从根节点进入，递归找到叶子节点[x,x]，把该节点的值增加k。然后从下往上更新祖先节点上的统计值。

```
void update(int p, int x, int k){
    if(tr[p].l == x && tr[p].r == x){
        tr[p].sum += k ;
        return ;
    }
    int m = tr[p].l + tr[p].r >> 1 ;
    if(x <= m) update(lc, x, k) ;
    if(x > m) update(rc, x, k) ;
    tr[p].sum = tr[lc].sum + tr[rc].sum ;
}
```

区间查询

拆分与拼凑的思想。例如，查询区间[4,9]可以拆分成[4,5]，[6,8]，[9,9]，

从根节节点进入，递归执行：

1. 如果查询区间[x,y]**完全覆盖**当前节点区间，则，立即回溯，并返回该节点的sum值。
2. 若左子节点与[x,y]有重叠，则递归访问左子树。
3. 若右子节点与[x,y]有重叠，则递归访问右子树。

```
int querry(int p, int x, int y){
    if(x <= tr[p].l && y >= tr[p].r){
        return tr[p].sum ;
    }
    int sum = 0 ;
    int m = tr[p].l + tr[p].r >> 1 ;
    if(x <= m) sum += querry(lc, x, y) ;
    if(y > m) sum += querry(rc, x, y) ;
    return sum ;
}
```

区间修改

懒惰修改，当[x,y]完全覆盖节点区间[a,b]时，先修改该区间的sum值，再打上一个懒惰标记，然后立即返回。等到下一次需要时，在下传懒惰标记，这样修改和查询都是$logn$。

```
void pushup(int p){
    tr[p].sum = tr[lc].sum + tr[rc].sum ;
}

void pushdown(int p){
    if(tr[p].add){
        tr[lc].sum += tr[p].add * (tr[lc].r - tr[lc].l + 1) ;
        tr[rc].sum += tr[p].add * (tr[rc].r - tr[rc].l + 1) ;
        tr[lc].add += tr[p].add ;
        tr[rc].add += tr[p].add ;
        tr[p].add = 0 ;
    }
}

void update(int p, int l, int r, int k){
    if(l <= tr[p].l && r >= tr[p].r){
        tr[p].sum += (tr[p].r - tr[p].l + 1) * k ;
        tr[p].add += k ;
        return ;
    }
    int m = (tr[p].l + tr[p].r) >> 1 ;
    pushdown(p) ;
    if(l <= m) update(lc, l, r, k) ;
    if(r > m) update(rc, l, r, k) ;
    pushup(p) ;
}
```

懒惰标记的查询

```
int querry(int p, int l, int r){
    if(l <= tr[p].l && r >= tr[p].r){
        return tr[p].sum ;
    }
    int m = (tr[p].l + tr[p].r) >> 1 ;
    pushdown(p) ;
    int sum = 0 ;
    if(l <= m)  sum += querry(lc, l, r) ;
    if(r > m) sum += querry(rc, l, r) ;
    return sum ;
}
```

GCD 线段树

$gcd(x,y,z) = gcd(x,y-x,z-y)$ 对于 n 个数也一样

```
#define lc p << 1
#define rc p << 1  | 1
struct node{
    int l ;
    int r ;
    int d ; //gcd
    int sum ;//区间和
    int add ; //懒惰标记
} tr[N*4];
int ans = 0 ;
int n , m, k, q ;
int x ;
int a[N], c[N] ;
void pushup(int p){//gcd线段树的更新
    tr[p].d = gcd(tr[lc].d , tr[rc].d) ;
}
void pushdown2(int p){//区间和懒惰标记的更新
    if(tr[p].add){
        tr[lc].sum += tr[p].add * (tr[lc].r - tr[lc].l + 1) ;
        tr[rc].sum += tr[p].add * (tr[rc].r - tr[rc].l + 1) ;
        tr[lc].add += tr[p].add ;
        tr[rc].add += tr[p].add ;
        tr[p].add = 0 ;
    }
}
void pushup2(int p){//区间和线段树的更新
    tr[p].sum = tr[lc].sum + tr[rc].sum ;
}
void update2(int p, int l, int r, int k){//区间和更新
    if(l <= tr[p].l && r >= tr[p].r){
        tr[p].sum += (tr[p].r - tr[p].l + 1) * k ;
        tr[p].add += k ;
        return ;
    }
    int m = (tr[p].l + tr[p].r) >> 1 ;
    pushdown2(p) ;
    if(l <= m) update2(lc, l, r, k) ;
    if(r > m) update2(rc, l, r, k) ;
    pushup2(p) ;
}
int querry2(int p, int l, int r){//查询区间和
    if(l <= tr[p].l && r >= tr[p].r){
        return tr[p].sum ;
    }
    int m = (tr[p].l + tr[p].r) >> 1 ;
    pushdown2(p) ;
    int sum = 0 ;
    if(l <= m)  sum += querry2(lc, l, r) ;
    if(r > m) sum += querry2(rc, l, r) ;
    return sum ;
}

void build(int p, int l, int r){//建立区间和线段数 和 gcd线段树
    tr[p] = {l, r, c[l],a[l],0} ;
    if(l == r) return ;
    int mid = l + r >> 1 ;
    build(lc, l, mid) ;
    build(rc, mid + 1, r) ;
    pushup(p) ;
    pushup2(p) ;
}
void update(int p, int x,int v){//区间gcd更新，只改 l 和 r + 1 即可
    if(tr[p].l == tr[p].r){
        // cout << tr[p].l << " " << x << endl ;
        tr[p].d += v ;
        return ;
```

```
    }
    int mid = tr[p].l + tr[p].r >> 1 ;
    if(x <= mid) update(lc,x,v) ;
    else  update(rc,x,v) ;
    pushup(p) ;
}

node merge(node a,node b){//gcd合并
    node res ;
    res.d = gcd(a.d , b.d );
    return res ;
}
node querry(int p,int l, int r){//求解区间gcd
    if(l <= tr[p].l && r >= tr[p].r){
        return tr[p] ;
    }
    int mid = (tr[p].l + tr[p].r) >> 1 ;
    node L = {0,0,0}, R= {0,0,0} ;
    if(l <= mid) L = querry(lc,l,r) ;
    if(r > mid) R = querry(rc,l,r) ;
    return merge(L, R) ;//更新区间gcd
}

int gcd_plus(int l, int r){//得到区间gcd
    int val = querry2(1,l,l) ;
    node R = querry(1,l + 1,r) ;
    return abs(gcd(val, R.d)) ;
}
void update3(int l,int r,int val){//区间增加一个数和来更新区间和和区间gcd
    update2(1,l,r,val) ;//更新区间和
    update(1,l,val) ;
    if(r+1<=n) update(1,r+1,-val) ;
}
void solve(){
    cin >> n ;
    for(int i = 1; i <= n ; ++ i){
        cin >> a[i] ;
        c[i] = a[i] - a[i-1] ;
    }
    build(1, 1, n) ;
    cin >> q ;
    int op  ;
    int l, r, x ;
    while(q -- ){
        cin >> op ;
        if(op == 1){
            cin >> l >> r ;
            cout << gcd_plus(l,r) << endl ;
        }else{
            cin >> l >> r >> x;
            update3(l, r, x) ;
        }
    }
    cout << endl ;
}
signed main(){
    IOS ;
    // cin >> _ ;
    while(_ --){
        solve() ;
    }
    return 0;

}
```

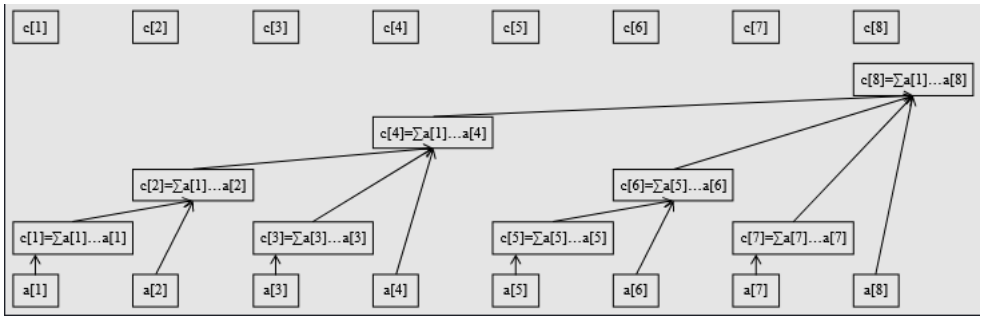
树状数组

	线段树	树状数组
结构	二叉树	阉割了一些点
空间	$4N$	N
时间	$logN$	$logN$
用途	维护区间信息（区间gcd，区间和，区间最大/小）	维护具有 **结合律** 且 **可差分** 的信息，入加法、异或等。 树状数组能解决的问题是线段树能解决问题的子集。
性能	代码长，时间常数大。	代码短，时间常数小。

树状数组维护的是一个前缀区间

我们观察一个数n的二进制，假如这个数是111_2，我们想算前111_2个数的前缀和，我们就可以将这7个数拆分给$111_2, 110_2, 100_2$这些编号来进行分配。那么问题就是如何分配数，把哪些数交给谁。

我们观察这个图：



可以发现每个数都分给了 $(x + lowbit(x))$ 对应的编号，我们设k为一个二进制数中最低位`1`所在的二进制位，那么每个编号管理的数的数目就是2^k。

$lowbit$函数（提取x对应的低位2次幂）

```
int lowbit(int x){
    return x & -x ;
}
```

加点

```
void add(int x,int k){
    while(x<=n){
        s[x] += k ;
        x += lowbit(x) ;
    }
}
```

查询

```
int querry(int x){
    int t = 0 ;
    while(x){
        t += s[x] ;
        x -= lowbit(x) ;
    }
    return t ;
}
```

参考链接

- 树状数组（BIT）
- 图片参考

单调栈和单调队列

单调栈

使用栈维护一个固定的边长窗口[1,i]内的单调序列，从栈顶取最值，进行计算或转移。

```
tt = -1 ;
_rep(i,1,n){
    cin >> a[i] ;
    while(tt != -1 && a[h[tt]] < a[i]){
        b[h[tt]] = i ;
        tt -- ;
    }
    h[++tt] = i ;
}
```

求解**最长上升 / 下降子序列**就用到了这个优化。

单调队列

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

以最小值为例

1. **队尾出队**的条件：队列不为空且新元素更优，队中旧元素队尾出队
2. 每个元素必然从**队尾进队**一次
3. **队头出队**的条件：队头元素滑出了窗口

注意：队列中存储元素的下标，方便判断队头出队。

```
hh =0 , tt = -1 ;//清空队列
_rep(i,1,n){
    cin >> a[i] ;
    while(hh <= tt && a[h[tt]] >= a[i]){//新的元素比队尾更优
        tt -- ;
    }
    h[++tt] = i ;//队尾入队新元素，如果并没有比新的元素更优也不会破坏队头的最小值
    if(h[hh] < i - k + 1) hh ++ ;//对头出队
    if(i >=k){
        cout << a[h[hh]] << " " ;
    }
}
```

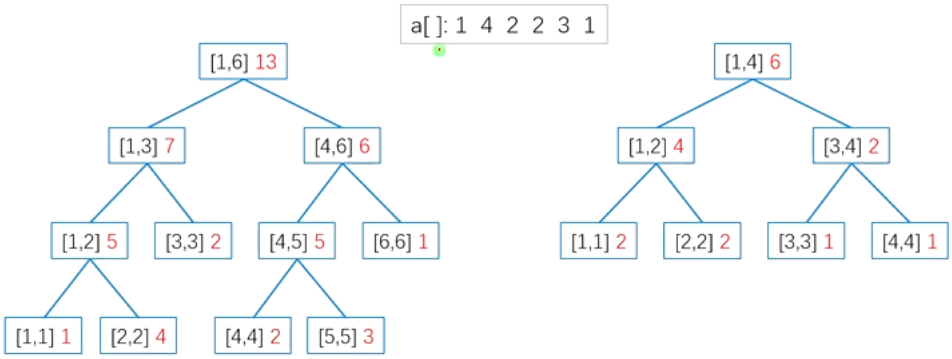
求解多**重背包问题**的优化

主席树

可持久化线段树：支持回退，访问之前版本的线段树。
思想：保存每一次插入操作时的历史版本。

每一次插入操作实际上只会修改 $nlog(n)+1$ 个节点

	普通线段树 （下标线段树）	权值线段树 （值域线段树）
节点区间	序列的下标区间	序列的值域
节点维护的信息	区间最值、区间和 等	值域内数的出现次数




```
const int N = 1e5+10 ;
#define lc(x) tr[x].ch[0]
#define rc(x) tr[x].ch[1]
struct node
{
    int ch[2] ;
    int s ;//值域中有多少个数
}tr[N*22];
int idx = 0 ;//节点编号
int n , m, k, q ;
int a[N] ;
int root[N] ;//每一个版本的根
vector<int> vec ;
void build(int & x,int l,int r){//创建初始记录
    x = ++idx ;
    if(l == r) return ;
    int mid = l + r  >> 1 ;
    build(lc(x),l, mid) ;
    build(rc(x),mid + 1, r) ;
}
void insert(int x, int &y, int l, int r,int v ){//创建主席树，创建版本记录
```

```
    y = ++idx ;
    tr[y] = tr[x] ;
    tr[y].s ++ ;//值域内数量++
    if(l == r) return ;
    int mid = l + r >> 1 ;
    if(v <= mid) insert(lc(x), lc(y), l, mid, v) ;
    else insert(rc(x), rc(y), mid + 1, r, v) ;
}
int querry(int x,int y, int l, int r,int k){//x表示插入l是时的版本，y表示插入r时的版本，k是第k小
    if(l == r) return l ;
    int mid = l + r >> 1 ;
    int s = tr[lc(y)].s - tr[lc(x)].s ;//一个区间两个版本之差表示这个区间内的数的数量，根据k是否小于等于左区区数的数量来进行二分查找
    if(k <= s) return querry(lc(x), lc(y), l , mid, k) ;
    else return querry(rc(x), rc(y), mid + 1 , r, k - s) ;
}
int getId(int x){
    return lower_bound(vec.begin(), vec.end(), x) - vec.begin()  + 1;
}
void solve(){
    cin >> n  >> q ;
    for(int i = 1; i <= n ; ++ i){
        cin >> a[i] ;
        vec.push_back(a[i]) ;
    }
    sort(vec.begin(), vec.end()) ;
    vec.erase(unique(vec.begin(), vec.end()),vec.end()) ;
    build(root[0],1,vec.size()) ;

    for(int i = 1 ; i <= n ; ++ i){
        insert(root[i-1],root[i],1,vec.size(),getId(a[i])) ;
    }

    int l, r ;
    int ans ;
    while(q -- ){
        cin >> l >> r >> k ;
        ans = querry(root[l-1],root[r],1,vec.size(), k) ;
        cout << vec[ans-1] << endl ;
    }
}
```

主席树 求区间不同数的数量

对于每一个数我们标记其下一次出现的位置为 $nex[i]$ ，那么我们想求 (l,r) 范围内的不同数的数量就可以转化为 (l,r) 区间内有多少个 $nex[i] \geq r+1$

```
struct node
{
    int ch[2] ;
    int sum = 0 ;
}tr[N];
int root[N] ;
int idx ;
void build(int & p,int l, int r){
    p = idx ++ ;
    int mid = l + r  >> 1 ;
    if(l == r) return ;
    build(lc(p), l , mid) ;
    build(lc(p), mid + 1 , r) ;
}
int p = -1 ;
void modify(int x,int &y,int l, int r,int val){
    y = idx ++ ;
    tr[y] = tr[x] ;
    tr[y].sum = tr[x].sum + val ;//
    if(l == r) return ;
    int mid = l + r >> 1LL ;
    if(p <= mid) modify(lc(x),lc(y), l , mid, val) ;
    else modify(rc(x),rc(y), mid + 1 , r, val) ;
}
int querry(int x,int y,int l,int r, int k){
    int mid = l + r >> 1 ;
    int res = 0 ;
    if(l == r) return tr[y].sum - tr[x].sum ;
    if(k <= mid){
        int sum = tr[rc(y)].sum - tr[rc(x)].sum ;
        res += querry(lc(x), lc(y),l ,mid, k) + sum;
    }else{
        res += querry(rc(x), rc(y),mid + 1, r, k) ;
    }
    return res ;
```

```
}
int getUniqueNum(int l,int r){
    return querry(root[l-1], root[r], 1,n + 1,r + 1) ;
}
```

ST表

介绍

$nlog(n)$时间复杂度解决RMQ问题。

原理

让我们的每一对区间都可以用至多两个区间来进行覆盖，这样我们就可以通过两个区间的最大值比较来得到新的区间的最大值。

实现

我们用 $f[i][j]$表示区间$(i,i+2^j-1)$的区间的最大值，我们的这个区间也可以用$(i,i+2^{j-1}-1)$和$(i+2^{j-1},(i+2^{j-1})+2^{j-1}-1)$这两个区间来进行覆盖，用f数组来表示就是$f[i][j-1]$和$f[i+(1LL<<(j-1))][j-1]$。因此我们就可以得到:

$$f[i][j]=max(f[i][j-1],f[i+(1LL<<(j-1))][j-1])$$

我们就可以得到**初始化**时的代码

```
for(int j = 1 ; j <= (int)log2(n) + 1 ; ++ j ){
    for(int i = 1;i <= n ; ++ i){
        f[i][j] = max(f[i][j-1],f[i + (1LL << (j-1))][j-1]) ;
    }
}
```

对于**查询**时，根据 **l** 和 **r** ，我们可以通过这个区间的长度为$k=log2(r-l+1)$，就可以得到我们的第一个区间$(l,l+2^k-1)$，但很明显我们的区间有可能没有完全覆盖因为我们的指数是向下取整了，所以我们还需要从右边**r**开始往左来划定一个区间$(r-2^k+1,r)$，这两个区间一定会有重叠，但一定能覆盖我们的这个个区间，这两个区间用(f)数组来表示就是$f[l][k]$和$f[r-(1<<k)+1][k]$，由此我们就可以得到我们查询代码。

```
l = read() ;
r = read() ;
int k = log2(r - l + 1) ;
ans = max(f[l][k],f[r-(1<<k)+1][k]) ;
```

01 Trie

01 Trie

应用

计算最大异或对

思路

异或运算 → 二进制位 → N 个整数均转化为二进制数表示 → 二进制是 "01" 构成的串→构造 Trie 树 → 在树上进行异或运算

代码

```
int idx ;
vector<vector<int>> ch(31 * (n + 2), vector<int>(2,0));
auto insert = [&](int x,int t) -> void{
    int p = 0 ;
    for(int i = 30 ; i >= 0 ; -- i){
        int j = x >> i & 1 ;
        if(!ch[p][j]) ch[p][j] = ++idx ;
        p = ch[p][j] ;
        cnt[p] += t ;
    }
};
auto querry = [&](int x) -> int{
    int p = 0 , res = 0 ;
    for(int i = 30 ; i >= 0 ; -- i){
        int j = x >> i & 1 ;
        if(ch[p][!j]){//尽量选择大的
            res += 1 << i ;
            p = ch[p][!j] ;
        }else{
            p = ch[p][j] ;
```

```
        }
    }
    return res ;
};
```

树链刨分

重儿子：父节点的所有儿子中，子树结点数目最多的节点。
轻儿子：父节点的所有儿子中，除重儿子以外的儿子。
重边：父节点和儿子连成的边。
重链：由多条重边连接而成的路径。

1. 整棵树会被刨分成若干条重链。
2. 轻儿子一定是每条重链的顶点。
3. 任意一条路径被切分成不超过$logn$条重链。

数组

- fa[u]：存u的父节点
- dep[u]：存u的深度
- son[u]：存u的重儿子
- sz[u]：存以u为根的子树节点数
- top[u]：存u所在重连的顶点
- id[u]：存u刨分后的新编号
- nw[cnt]：存新编号在树中所对应节点的权值

算法

- dfs1：搞出fa，dep，sz，son

```
void dfs1(int x, int f){
    dep[x] =dep[f] + 1 ;//深度
    fa[x] = f ;//父节点
    sz[x] = 1 ;
    for(auto e : e[x]){
        if(e == f) continue ;
        dfs1(e,x) ;
        sz[x] += sz[e] ;
        if(sz[son[x]] < sz[e]) son[x] = e;
    }
}
```

- dfs2：搞出top，id，nw

```
void dfs2(int x,int y){
    top[x] = y ;//记录链头
    if(!son[x]) return ;
    dfs2(son[x],y) ;//搜索重链上的重儿子
    for(auto e : e[x]){
        if(e == fa[x] || e == son[x]) continue ;
        dfs2(e,e) ;//新开一条重链
    }
}
```

- 让两个游标沿着各自的重链向上跳，跳到同一条重链上时，深度较小的那个游标所指向的点就是 **LCA** 。

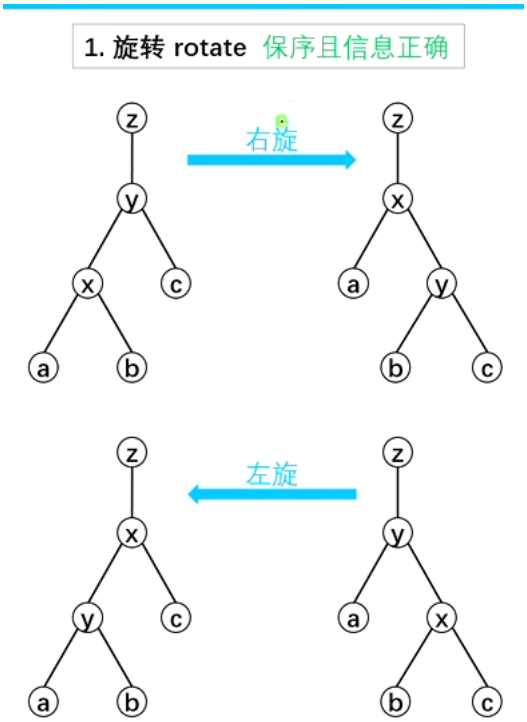
```
int lca(int x, int y){
    while(top[x] != top[y]){
        if(dep[top[x]] < dep[top[y]]) swap(x,y) ;
        x = fa[top[x]] ;
    }
    return dep[x] < dep[y] ? x : y ;
}
```

平衡树

二叉查找树（BST） 是一种能存储特定数据类型的容器。二叉查找树允许快速**查找，插入或者删除**一节点。重要性质：左小右大，中序遍历是有序的。

Splay（伸展树） 是一种平衡二叉树，它通过将一个节点**旋转**到根结点，使得整个棵树仍然满足 **BST** 的性质，并且保持平衡而不至于退化成链。

1.旋转



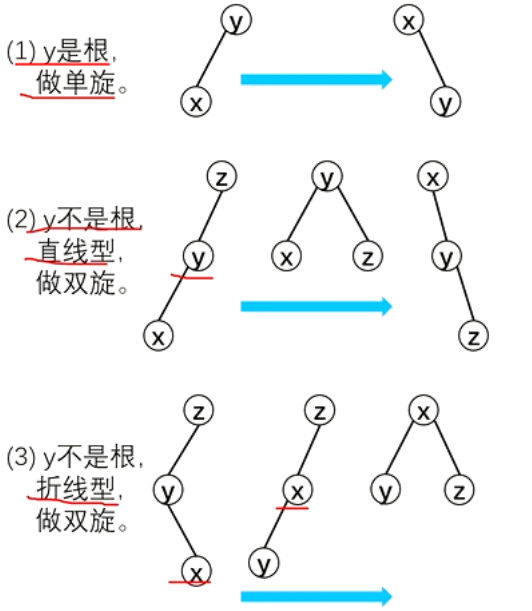

```
struct node{
    int s[2] ;//左右儿子
    int p ; //父亲
    int v ;//节点权值
    int cnt;//权值出现次数
    int size;// 子树大小
    void init(int p1, int v1){
        p = p1, v = v1 ;
        cnt = size = 1 ;
    }
}tr[N];
int root ;//根节点编号
int idx ;//节点个数
```

```
void rotate(int x){//旋转，x为旋转后往上走的节点
    //y 为当前节点的 父节点， z 为原先当前节点父节点的父节点
    int y = tr[x].p , z = tr[y].p ;
    int k = tr[y].s[1] == x ;//k = 1 说明左旋 反之说明右旋
    //以左旋为例
    //旋转后， 当前节点的左儿子节点变成原先当前节点父节点的右儿子节点
    tr[y].s[k] = tr[x].s[k^1] ;
    //同时当前节点的左儿子节点的父节点变成原先当前节点的父节点
    tr[tr[x].s[k^1]].p = y ;
    //当前节点的左儿子节点变成原先当前节点父节点
    tr[x].s[k^1] = y ;
    //当前节点的父节点的父节点变成原先当前节点
    tr[y].p = x ;
    //当前父节点的父节点的儿子节点变成原先当前节点，替换原先父亲节点的位置
    tr[z].s[tr[z].s[1]==y] = x ;
    //当前节点的父节点变成原先父节点的父节点
    tr[x].p = z ;
    pushup(x) ;
    pushup(y) ;
}
```

```
void pushup(x){
    tr[x].size = tr[tr[x].s[0]].size + tr[tr[x].s[1]].size + tr[x].cnt ;
}
```

2. 伸展

访问一个节点 x 就要把 x 旋转到根结点。




```
//k >0 时，把 x 转到 k 下面
//k = 0 时，把 x 转到根
void splay(int x,int k){
    while(tr[x].p != k){
        int y = tr[x].p, z = tr[y].p ;
        if(z!= k){//直线型先转y后转x。折线形转两次x
            ((tr[z].s[0] == y) ^ (tr[y].s[0] == x)) : rotate(x) ? rotate(y) ;
        }
        rotate(x) ;
    }
    if(k == 0) root = x ;
}
```

3. 查找 find

找到 v 所在节点并把其转到根上

```
void find(int v){
    int x = root ;
    while(tr[x].s[v > tr[x].v] && v! = tr[x].v){//两个条件， 第一个是找不到目标节点， 第二个时，找到目标节点
        x = tr[x].s[v > tr[x].v] ;
    splay(x,0) ;
    }
}
```

4. 前驱 get_pre

求 v 的前驱， 返回其节点编号

```
int get_pre(int v){
    find(v) ;
    int x = root ;
    if(tr[x].v < v) retrun x ; //说明当前根节点就是 v 的前驱
    //如果 tr[x].v = v，那么v 的前驱就一定在树伸展后的左子树的右下角
    //如果 tr[x].v > v，因为此时的根节点与v一定是最接近的，所以此时v 大于当前根节点左儿子的v，也就是说其前驱也一样是当前伸展后根节点左子树的右下角
    x = tr[x].s[0] ;//得到左子树右下角
    while(tr[x].s[1]) x = tr[x].s[1] ;
    return x ;
}
```

5. 后继 get_suc

求 v 的后继， 返回其节点编号

```
//和前驱同理
int get_suc(int v){
    find(v) ;
    int x = root ;
    if(tr[x].v > v) retrun x ; //说明当前根节点就是 v 的后继驱
    x = tr[x].s[1] ;//得到右子树左下角
    while(tr[x].s[0]) x = tr[x].s[0] ;
    return x ;
}
```

6. 删除

删除 v （如果有多个相同的，只删除一个）

```
void del(int v){
    int pre = get_pre(v) ;//把前驱节点放在根节点
    int suc = get_suc(v) ;//后继节点放在根节点之下
    splay(pre,0), splay(suc,pre) ;
    //伸展后，比我们要删除的节点大的节点在suc的右子树上， 比其小的在 pre的左子树上，
    int del = tr[suc].s[0] ;//也就是说我们现在要删除的节点没有子节点，为叶子节点
    if(tr[del].cnt > 1){
        tr[del].cnt --, splay(del,0) ;
    }else{
        tr[suc].s[0] = 0, splay(suc,0) ;
    }
}
```

7. 插入

```
void insert(int v){
    int x = root, p = 0 ;
    while(x && tr[x],v !=v ){
         p = x , x = tr[p].s[v>tr[x].v]
    }
    if(x) tr[x].cnt++ ;//该值第一次出现
    else{//新出现的值
        x = ++ idx ;
        tr[p].s[v>tr[p].v] = x ;//记录新的节点是谁的哪个儿子节点
        tr[x].init(p,v) ;
    }
    splay(x, 0);
}
```

8. 查询数据排名（该数也有可能不存在）

```
int get_rank(int v){
    insert(v) ;//先添加一下这个值，可能不存在
    int res = tr[tr[root].s[0]].size ;//为什么不用+1是因为还有哨兵节点无穷小
    del(v);//最后删除，因为是辅助用，本不应该存在
    return res ;
}
```

9. 查询排名为 k 的数值

```
int get_val(int k){
    int x = root ;
    while(1){
        int y = tr[x].s[0] ;
        if(tr[y].size() + tr[x].cnt < k){//说明目标在右子树上
            k -= tr[y].size() + tr[x].cnt ;
            x = tr[x].s[1] ;
        }else{
            if(tr[y].size() >= k) x = tr[x].s[0] ;
            else break ;
        }
    }
    splay(x,0) ;//避免变成链
    return tr[x].v ;
}
```

完整代码

```cpp
#include<bits/stdc++.h>
#define int long long
#define lc(x) tr[x].s[0] ;
#define rc(x) tr[x].s[1] ;
using namespace std;
const int N = 1e6 + 10;
const int INF = 0x3f3f3f3f3f3f3f3f ;
struct node
{
    int v ;//权值大小
    int p ;//父亲节点
    int cnt ; //权值数量
    int siz ; //树的大小
    int s[2] ; //儿子节点
    void init(int p1, int v1){
        p = p1 , v = v1 ;
        cnt = siz = 1 ;

    }
}tr[N];
int root ; //表示根节点的编号
int idx ; //表示新的节点的编号
void pushup(int x){
    tr[x].siz = tr[tr[x].s[0]].siz + tr[tr[x].s[1]].siz + tr[x].cnt ;
}
void rotate(int x){//旋转节点
    int y = tr[x].p, z = tr[y].p ;
    int k = tr[y].s[1] == x ;//1说明要左旋
    tr[y].s[k] = tr[x].s[k^1] ;//原来节点的左儿子，变成父节点的右儿子节点
    tr[tr[x].s[k^1]].p = y ;//原来节点的左儿子节点的父亲节点变成原先节点的父亲节点
    tr[x].s[k^1] = y ;//原来节点的左儿子变成其父亲节点
    tr[y].p = x ;
    tr[z].s[tr[z].s[1] == y] = x ;
    tr[x].p = z ;
    //更新树 大小
    pushup(x) ;
    pushup(y) ;
}
void splay(int x, int k){//伸展 两个参数， 第一个参数所代表的节点要变成 第二个参数的节点的儿子节点
    while(tr[x].p != k){
        int y = tr[x].p , z = tr[y].p ;
        if(z != k){
            (tr[z].s[0] == y) ^ (tr[y].s[0] == x) ? rotate(x) : rotate(y) ;
        }
        rotate(x) ;
    }
    if(k == 0) root = x ;
}
void insert(int v){
    int x = root , p = 0 ;
    while(x && tr[x].v != v){
        p = x , x = tr[p].s[v > tr[x].v] ;
    }

    if(x){//说明不是新的点
        tr[x].cnt ++ ;
    }else{
        x = ++idx ;
        tr[p].s[v > tr[p].v] = x ;
        tr[x].init(p,v) ;
    }
    splay(x,0) ;
}
void find(int v){//把v做权值所在节点转移到根节点上
    int x = root ;
    while(tr[x].s[v > tr[x].v] && tr[x].v != v){//还有可能目标节点不存在
        x = tr[x].s[v > tr[x].v] ;
    }
    splay(x,0) ;
}
int get_pre(int v){
    find(v) ;
    int x = root ;
    if(tr[x].v < v) return x ;
    x = tr[x].s[0] ;
    while(tr[x].s[1]){
        x = tr[x].s[1] ;
    }
    return x ;
}

```

```
int get_suc(int v){
    find(v) ;
    int x = root ;
    if(tr[x].v > v) return x ;
    x = tr[x].s[1] ;
    while(tr[x].s[0]){
        x = tr[x].s[0] ;
    }
    return x ;
}

void del(int v){//要把目标节点放在叶子节点上
    //让后继变成前驱的右儿子，  让当前变成后继的左儿子
    int pre = get_pre(v) ;
    int suc = get_suc(v) ;
    splay(pre,0), splay(suc, pre) ;
    int del = tr[suc].s[0] ;//这就是目标删除的节点
    if(tr[del].cnt > 1){
        tr[del].cnt -- ;
        splay(del,0) ;
    }else{
        tr[suc].s[0] = 0 ;
        splay(suc,0) ;
    }
}

int get_rank(int v){
    insert(v) ;
    int res = tr[tr[root].s[0]].siz ;
    del(v) ;
    return res ;
}
int get_val(int k){
    int x = root ;
    while(1){
        if(k <= tr[tr[x].s[0]].siz + tr[x].cnt){
            if(k > tr[tr[x].s[0]].siz){
                break ;
            }else{
                x = tr[x].s[0] ;
            }
        }else{
            k -= tr[tr[x].s[0]].siz + tr[x].cnt ;
            x = tr[x].s[1] ;
        }
    }
    return x ;

}
int n ;
void solve(){
    cin >> n ;
    int op ;
    int v ;
    //两个哨兵
    insert(INF) ;
    insert(-INF) ;
    while(n -- ){
        cin >> op ;
        cin >> v ;
        if(op == 1){
            insert(v) ;
        }else if(op == 2){
            del(v) ;
        }else if(op == 3){
            int res = get_rank(v) ;
            cout << res << endl ;
        }else if(op == 4){
            int res = get_val(v + 1) ;
            res = tr[res].v ;
            cout << res << endl ;
        }else if(op == 5){
            int res = get_pre(v) ;
            res = tr[res].v ;
            cout << res << endl ;
        }else{
            int res = get_suc(v) ;
            res = tr[res].v ;
            cout << res << endl ;
        }
    }

}

```

```
signed main()
{
    solve() ;
}
```

图论

树的两心一直

树的重心

给定一颗树，树中包含 n个结点（编号$1 \sim n$）和 $n-1$ 条无向边。

请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。

重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。

```
vector<int> g[N] ;
bool st[N] ;
int dfs(int x){
    st[x] = true ;
    int sum = 0 ;
    int res = -INF ;
    for(int i = 0 ; i < g[x].size() ; ++ i){
        int j = g[x][i] ;
        if(st[j]) continue ;
        int s = dfs(j) ;
        res = max(res,s) ;//子树的最大值
        sum += s ;
    }
    res = max(res, n - (1 + sum)) ;//子树和除去子树的最大值
    ans = min(res,ans) ;//最大中取最小
    return sum + 1 ;
}
```

树的中心

给定一棵树，树中包含 *n* 个结点（编号1~*n*）和 *n*-1

条无向边，每条边都有一个权值。

请你在树中找到一个点，使得该点到树中其他结点的最远距离最近。

```
vector<PII> g[N] ;
int d1[N] ;//i节点往下走的最长路径
int d2[N] ;//i节点往下走的次长路径
int up[N] ;//i节点往上走的最长路径
int p[N] ;//表示i节点的向下最长的路径经过的是哪个子节点
int dfs(int x,int fa){//从下往上走
    for(int i = 0 ; i < g[x].size() ; ++ i){
        int j = g[x][i].first ;
        if(fa == j) continue ;
        int d = dfs(j,x) + g[x][i].second;
        if(d >= d1[x]){
            p[x] = j ;
            d2[x] = d1[x] , d1[x] = d ;
        }else if(d > d2[x]){
            d2[x] = d ;
        }
    }
    return d1[x] ;
}
void dfs2(int x,int fa){//从上往下走
    for(int i = 0 ; i < g[x].size() ; ++ i){
        int j = g[x][i].first ;
        if(j == fa) continue ;
        if(j == p[x]) up[j] = max(up[x], d2[x]) + g[x][i].second ;//一种往父节点的方向走，一种往次长兄弟节点的方向走
        else up[j] = max(up[x], d1[x]) + g[x][i].second ;//一种往父节点的方向走，一种往最长兄弟节点的方向走
        dfs2(j,x) ;
    }
}
void solve(){
    cin >> n ;
    int u, v, w ;
    ans = -INF ;
    for(int i = 1; i<= n-1 ; ++ i){
        cin >> u >> v >> w;
        g[u].push_back({v,w}) ;
        g[v].push_back({u,w}) ;
    }
```

```
38
39      dfs(1, 0) ;
40      dfs2(1, 0) ;
41      ans = INF ;
42      for(int i = 1; i<= n ; ++ i){
43          ans = min(ans,max(d1[i],up[i])) ;
44      }
45      cout << ans << endl ;
46  }
```

树的直径

给定一棵树，树中包含 n 个结点（编号1~n）和 n−1

条无向边，每条边都有一个权值。

现在请你找到树中的一条最长路径。

换句话说，要找到一条路径，使得使得路径两端的点的距离最远。

注意：路径中可以只包含一个点。

```
1   int a[N] ;//排列
2   vector<PII> g[N] ;
3   bool st[N] ;
4   int dfs(int x){
5       st[x] = true ;
6       int d1 = 0 ,d2 = 0 ;
7       for(int i = 0 ; i < g[x].size() ; ++ i){
8           int j = g[x][i].first ;
9           if(st[j]) continue ;
10          int d = dfs(j) + g[x][i].second;
11          if(d > d1){
12              d2 = d1 , d1 = d ;
13          }else if(d > d2){
14              d2 = d ;
15          }
16      }
17      ans = max(ans, d1 + d2 ) ;
18      return d1 ;
19  }
20  void solve(){
21      cin >> n ;
22      int u, v, w ;
23      ans = -INF ;
24      for(int i = 1; i<= n ; ++ i){
25          cin >> u >> v >> w;
26          g[u].push_back({v,w}) ;
27          g[v].push_back({u,w}) ;
28      }
29      dfs(1) ;
30      cout << ans << endl ;
31  }
```

差分约束

概述

n 个变量和 m 个约束条件。每一个约束条件是由其中的两个变量做差构成的，如 $x_i - x_j \le c_k$，其中 $1 \le i, j \le n, i \ne j, 1 \le k \le m$。

约束条件 $x_i - x_j \le c_k$ 可以变成 $x_i \le c_k + x_j$，这和单源最短路中的三角不等式相似。因此，为破门可以把变量 x_i 看作一个节点，对于每一个约束条件 $x_i - x_j \le c_k$，从 j 向 i 连一条长度为 c_k 的有向边。

题意	转化	边长
$x_a - x_b \ge c$	$x_b \le x_a - c$	`add(a,b,-c)`
$x_a - x_b \le c$	$x_a \le x_b + c$	`add(b,a,c)`
$x_a = x_b$	$x_a \le x_b + 0$ $x_b \le x_a + 0$	`add(a,b,0)` `add(b,a,0)`

这类问题一般通过 $spfa$ 算法解决，如果存在正/负环则说明无解。
对于 $x_b \le x_a + c$, 求最长路， 其中 c 为正整数
对于 $x_b \ge x_a + c$, 求最短路， 其中 c 为正整数

最后的一组解为$x_1, x_2. x_3, \dots$，呢么 $x_1 + d, x_2 + d$，$x_3 + d, \dots (d \in Z)$，也都是解。

求最小值时用最长路，求最大值时用最短路。

二分图最大匹配

匈牙利算法

假设当前 x 需要分配一个点， 但是已经没有可以分配的点给它了，呢么我们就可以询问那些已经分配好的点的对象是否有其他空余的可选点，使得让给当前点一个点。

$vis[N]$ 当前点是否已经被分配了
$match[v]$ 表示当前点分配了给谁

```
bool find (int x){
    _rep(i,0,m){//遍历寻找可以匹配的边
        if(!st[i] && vis[x][i]){//如果这条边在这一次已经被使用过（没能找到空余出来的边），就不再使用，
            st[i] = true ;
            if( match[i] == -1 || find(match[i])){//如果匹配边还没有被匹配或者已经被匹配，但是他的主人可以选其他的边，使得这条边被空余出来了
                match[i] = x ;//更新这点的主人
                return true ;
            }
        }

    }
    return false ;
}
for(int i = 1 ; i <= n ; ++i){
    memset(st,false,sizeof st) ;
    if(find(i)) ans ++ ;
}
```

- 在二分图中，**最小点覆盖数**等于**最大匹配数**。这被称为**Konig定理**。
- 二分图的最小边覆盖数=图中的顶点数-（最小点覆盖数）该二分图的最大匹配数
- **DAG图（有向无环图）**的最小路径覆盖可以转化为二分图
- **最大匹配数** = 最小点覆盖 = 总点数-**最大独立集** = 总点数-**最小路径覆盖**

最小路径覆盖：用最少**互不相交的路径**覆盖**所有的点**

最大独立集：选出最多的点 使得选出的点之间没有边

Dinic 网路流

二分图转化成**网络流**模型。创建**虚拟源点**和**汇点**，将源点连上左边所有点，右边所有点连上汇点，容量都是 1。原来的每一条边从左往右连边，容量也是 1 。最大流即最大匹配

```
int S, E ;
int h[N], e[N], ne[N], w[N], idx ;
void add(int u,int v,int val){
    e[idx] = v ;
    ne[idx] = h[u] ;
    w[idx] = val ;
    h[u] = idx ++ ;

    e[idx] = u ;
    ne[idx] = h[v] ;
    w[idx] = 0 ;
    h[v] = idx ++ ;
}
int d[N], cur[N] ;
bool bfs(){
    queue<int> q ;
    q.push(S) ;
    memset(d,0,sizeof d) ;
    d[S] = 1 ;
    while(q.size()){
        int t = q.front() ;
        q.pop() ;
        for(int i = h[t] ; ~i ; i = ne[i]){
            int j = e[i] ;
            if(!d[j]&&w[i]){
                d[j] = d[t] + 1 ;
                if(j == E) return true ;
                q.push(j) ;
            }
        }
    }
    return false ;
}
int dfs(int x, int mf){
    if(x == E) return mf ;
    int sum = 0 ;
    for(int i = cur[x] ; ~i ; i = ne[i]){
```

```
        int j = e[i] ;
        cur[x] = i ;
        if(d[j] == d[x] + 1 && w[i]){
            int f = dfs(j,min(w[i],mf)) ;
            w[i] -= f ;
            w[i^1] += f ;
            sum += f ;
            mf -= f ;
            if(mf == 0) break ;
        }
    }
    if(sum == 0) d[x] = 0 ;
    return sum ;
}
int dinic(){
    int res = 0 ;
    while (bfs()){
        memcpy(cur, h, sizeof cur) ;
        res += dfs(S, INF) ;
    }
    return res ;

}
void solve(){
    cin >> n >> m >> k ;
    int u, v ;
    S = 0 ;
    E =  n + m + 1 ;
    for(int i = 0 ; i <= n + m + 1 ; ++ i) h[i] = -1 ;
    for(int i = 1; i <= k ; ++ i){
        cin >> u >> v ;
        add(u,v+n,1) ;
    }
    for(int i = 1; i <= n ; ++ i){
        add(S,i,1) ;
    }
    for(int i = 1; i <= m ; ++ i){
        add(i+n,E,1) ;
    }
    ans = dinic() ;
    cout << ans << endl ;
}
```

网络流

网路是指一个有向图 $G=(V,E)$，有两个特殊的节点：**源点** S 和 **汇点** T。每条有向边 $(x,y)\in E$ 都有一个权值 $C(x,y)$，称为边的 **容量**。若 $(x,y)\notin E$，则 $C(x,y)=0$。

- **最大流**：从源点流向汇点的最大流量。
- **增广路**：一条从源点到汇点的所有边得剩余容量 ≥ 0 的路径。
- **残留网**：有网络中所有节点和剩余容量大于 0 的便所构成的子图，这里的边包括有向边何其反向边。

建图时每条有向边 (x,y) 都构建一个构建反向边 (x,y)，初始容量为 $C(x,y)=0$。目的是提供一个“**退流管道**”，一旦前面的增广路堵死可行流，可以通过“**退流管道**”退流。

最大流

EK 算法

循环找增广路，每找到一条

1. 逆序更新残留网络，容量此消彼长
2. 累加可行流，最后返回最大流

```
int mfs[N] ;// 到达v点的最大流量
int pre[N] ; //粗存储一条增广路上的点的前驱边，方便找到路径后，遍历路径
bool bfs(){
    queue<int> q ;
    q.push(1) ;
    memset(mfs,0, sizeof mfs) ;//每一次更新
    mfs[1] = INF ;
    while(q.size()){
        int t = q.front() ;
        q.pop() ;
        for(int i = h[t] ; ~i ; i = ne[i]){
            if(mfs[e[i]] == 0 && w[i]){
                mfs[e[i]] = min(w[i],mfs[t]) ;//更新这条增广路上的最小值，也就是这台路径上的最大流量。
                q.push(e[i]) ;
                pre[e[i]] = i ;
                if(e[i] == m) return true ;
```

```
                q.push(e[i]) ;
            }
        }
    }
    return false ;
}
int EK(){
    int res = 0 ;
    while(bfs()){//维护这条增广路， 构造残留网
        int v = m ;
        while(v != 1){
            w[pre[v]] -= mfs[m] ;
            w[pre[v] ^ 1] += mfs[m] ;
            v = e[pre[v]^1] ;
        }
        res += mfs[m] ;
    }
    return res ;
}
```

Dinic 算法

深度 $d[u]$ 存储当前 u 点的深度——所在的图层
前弧 $cur[u]$ 存储 u 点的当前出边

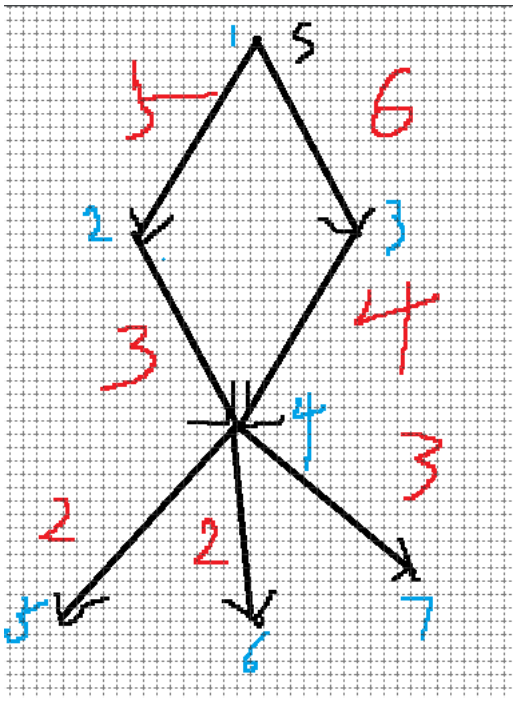
1. bfs 对点分层，找增广路。
2. dfs 多路增广
    1. 搜索顺序优化
    2. 当前弧优化
    3. 剩余流量优化
    4. 残枝优化
3. $dinic$ 累加可行流

> 一般网络，时间复杂度为 $O(V^2E)$
>
> 单位容量网络中（每条边的容量为1），时间复杂度为 $O(E\sqrt{V})$

```
int d[N], cur[N*2] ;
bool bfs(){
    queue<int> q ;
    q.push(1) ;
    memset(d,0, sizeof d) ;
    d[1] = 1 ;
    while(q.size()){
        int t = q.front() ;
        q.pop() ;
        for(int i = h[t] ; ~i ; i = ne[i]){
            int j = e[i] ;
            if(d[j] == 0 && w[i]){
                d[j] = d[t] + 1 ;
                q.push(j) ;
                if(j == m) return true ;
            }
        }
    }
    return false ;
}
int dfs(int u, int mf){
    if(u == m) return mf ;
    int sum = 0 ;
    for(int i =cur[u] ; ~i ; i = ne[i]){
        cur[u] = i ;
        int j = e[i] ;
        if(d[j] == d[u] + 1 && w[i]){
            int f = dfs(j,min(mf,w[i])) ;
            w[i] -= f ;
            w[i^1] += f ;
            sum += f ;
            mf -= f ;//剩余还剩的流量
            if(mf == 0) break ;
        }
    }
    if(sum == 0) d[u] = 0 ;
    return sum ;

}
int dinic(){
    int res = 0 ;
    while(bfs()){
        memcpy(cur, h,sizeof h) ;
```

```
        res += dfs(1,INF) ;
    }
    return res ;
}
```

为什么使用 cur 优化



红色表示对应边的容量，蓝色表示点的编号。

当我们 dfs 顺序为 $1,2,4$ 时，4 这个位置的 $mf=3$，
走过 5 点后，分配给 $(4,5)$ 大小为2 流量，然后$mf=mf-2=1$
当我们接着走 6 点时， 通过 $min(mf,w[i])$ ，显然分配给 $(4,6)$ 的流量为 1，然后 mf 就会变成 0 ,退出当前层，同时用 cur 标记此层之后的从哪条边开始的边的容量还有剩余。
接着我们就可以通过 $1,3,4$ 这条路径从 cur 标记位置开始继续利用那些还没有被分配的容量，
综上，cur 起到了剪枝的作用。

最小割

Dinic 算法

割的定义：对于一个网络 $G(V,E)$，我们切一刀，将所有的点划分为 S 和 T 两个集合，称为割 (S,T)，其中源点 $s\in S$，汇点 $t\in T$。对于任何一个割 (S,T) 都会使得网络断流。

割的容量：割的容量 $c(S,T)$ 表示所有从 S 到 T 的**出边**容量之和

最小割：最小割就是求得一个割 $(S，T)$，使得其 $C(S,T)$ 最小，方案并不一定为唯一。

最大流最小割定理：$最大流=最小割$

三个问题

- 求最小割（直接求最大流即可）
- 求最小割的划分
- 求最小割的边数

求划分

```
//遍历残留网，走所有能够达到的点(点位走过，且边的容量不为 0 )并将其标记，最后所有标记的点和所有未被标记的点就即使两个集合 S 和 T
void mincut(int u){
    for(int i = h[u] ;  ~i ; i = ne[i]){
        int j = e[i] ;
        vis[u] = 1 ;
        if(!vis[j] && w[i]){
            mincut(j) ;
        }
    }
}
```

费用流

SPFA + dinic

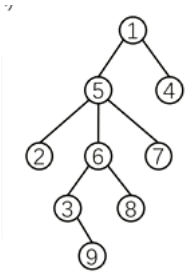
```cpp
int ans = 0 ;
int ans_spend = 0 ;
int n , m, k, q ;
int S, E ;
int h[N], e[N], ne[N], w[N], c[N], idx ;
void add(int u,int v,int val,int cost){
    e[idx] = v ;
    ne[idx] = h[u] ;
    w[idx] = val ;
    c[idx] = cost ;
    h[u] = idx ++ ;

    e[idx] = u ;
    ne[idx] = h[v] ;
    w[idx] = 0 ;//反向边 0 容量
    c[idx] = -cost ;//反向边负费用
    h[v] = idx ++ ;
}
int d[N], cur[N] ;
bool vis[N] ;
bool spfa(){
    for(int i = 1; i <= n ; ++i){
        d[i] = INF ;
        vis[i] = false ;
    }
    queue<int> q ;
    q.push(S) ;
    d[S] = 0 ;
    vis[S] = true ;
    while(q.size()){
        int t = q.front() ;
        q.pop() ;
        vis[t] = false ;
        for(int i = h[t] ; ~i ; i = ne[i]){
            int j = e[i] ;
            if(d[j] > d[t] + c[i] && w[i]){
                d[j] = d[t] + c[i] ;
                if(!vis[j]){
                    q.push(j) ;
                    vis[j] = true ;
                }
            }
        }
    }//队列为空时vis也被全部重置为 false ， 所以dfs 可以直接用
    return d[E] != INF ;
}
int dfs(int x, int mf){
    vis[x] = true ;
    if(x == E) return mf ;
    int sum = 0 ;
    for(int i = cur[x] ; ~i ; i = ne[i]){
        int j = e[i] ;
        cur[x] = i ;
        if(w[i] && !vis[j] && d[j] == d[x] + c[i]){
            int f = dfs(j,min(w[i],mf)) ;
            w[i] -= f ;
            w[i^1] += f ;
            sum += f ;
            mf -= f ;
            ans_spend += c[i] * f ;
            if(mf == 0) break ;
        }
    }
    vis[x] = false ;//还原标记的点
    return sum ;
}
int dinic(){
    int res = 0 ;
    while (spfa()){
        for(int i = 1; i <= n ; ++ i){
            cur[i] = h[i] ;
        }
        res += dfs(S, INF) ;
    }
    return res ;

}
void solve(){
    cin >> n >> m >> S >> E ;
```

```
    int u, v, val, cost ;
    for(int i = 0 ; i <= n  ; ++ i) h[i] = -1 ;
    for(int i = 1; i <= m ; ++ i){
        cin >> u >> v >> val >> cost;
        add(u, v, val, cost) ;
    }
    ans = dinic() ;
    cout << ans << " " << ans_spend <<  endl ;
}
```

LCA（最近公共祖先）

倍增算法

- $dep[u]$：存点u的深度
- $fa[i][j]$：从u点往上跳2^j层的祖先节点。



例如：9节点往上跳2^0的祖先节点是3，跳2^1的祖先节点是6，跳2^2的祖先节点是1，跳2^3的祖先节点是0

```
int fa[N][30] ;
int dep[N] ;
vector<int> e[N] ;//存图
void add(int u, int v){
    e[u].push_back(v) ;
}
void dfs(int x,int f){
    dep[x] = dep[f] + 1 ;
    fa[x][0] = f ;
    for(int i = 1 ; i <= 20 ; ++ i){
        fa[x][i] = fa[fa[x][i-1]][i-1] ;//倍增
    }
    for(auto y : e[x]){
        if(f != y) dfs(y,x) ;
    }
}
int lca(int u, int v){
    if(dep[u] < dep[v]) swap(u,v) ;
    for(int i = 20 ; i >= 0 ; -- i){//贪心让两个点深度相等
        if(dep[fa[u][i]] >= dep[v] ){
            u =  fa[u][i] ;
        }
    }
    if(u == v) return v ;//一个节点是另一个节点的祖先节点
    for(int i = 20 ; i >= 0 ; -- i){//贪心让两个深度相等的点最终上一层的点相等，也可以说是找到深度最低的两个不相等的点。
        if(fa[u][i] != fa[v][i] ){
            u =  fa[u][i] ;
            v =  fa[v][i] ;
        }
    }
    return fa[u][0] ;
}
```

Tarjan算法

离线算法，利用**并查集**维护祖先节点。

维护两个点的公共祖先，我们可以在判断（u,v）的最近公共祖先时，如果其中一个点（u）已经搜索了（st标记为true），在搜索完另一个节点(v)时，另一个节点（u）的祖先节点就是这两个节点的最近公共祖先，因为我们搜索完成一个节点才会添加边，相当于时从下往上，从左往右添加边，所以等到搜索完两个节点时，此时图中的根节点就是两者的公共祖先。

```
void tarjan(int x){
    st[x] = true ;
    for(auto y : e[x]){
        if(!st[y]){
```

```
            tarjan(y) ;
            p[y] = x ;
        }
    }
    for(auto y : query[x]){
        int i = y.second ;
        int v = y.first ;
        if(st[v]) ans[i] = find(v) ;//x节点走完后，v点已经走过（这里的x和v是一组查询），说明最近公共祖先就是现在（现有的边所构成的）图中的v节点祖先节点
    }
}
int s ;
void solve(){
    cin >> n  >> m >> s;
    int u, v ;
    int t = n- 1 ;
    _rep(i,1,n) p[i] = i ;
    while(t--){
        cin >> u >> v ;
        add(u,v) ;
        add(v,u) ;
    }
    _rep(i,1,m){
        cin >> u >> v ;
        if(u == v){
            ans[i] = u ;
            continue ;
        }
        query[u].push_back({v,i}) ;
        query[v].push_back({u,i}) ;
    }
    tarjan(s) ;
    _rep(i,1,m){
        cout << ans[i] << endl ;
    }
}
```

树链刨分

重儿子：父节点的所有儿子中，子树结点数目最多的节点。
轻儿子：父节点的所有儿子中，除重儿子以外的儿子。
重边：父节点和儿子连成的边。
重链：由多条重边连接而成的路径。

1. 整棵树会被刨分成若干条重链。
2. 轻儿子一定是每条重链的顶点。
3. 任意一条路径被切分成不超过$logn$条重链。

数组

- fa[u]：存u的父节点
- dep[u]：存u的深度
- son[u]：存u的重儿子
- sz[u]：存以u为根的子树节点数
- top[u]：存u所在重连的顶点
- id[u]：存u刨分后的新编号
- nw[cnt]：存新编号在树中所对应节点的权值

算法

- dfs1：搞出fa，dep，sz，son

```
void dfs1(int x, int f){
    dep[x] =dep[f] + 1 ;//深度
    fa[x] = f ;//父节点
    sz[x] = 1 ;
    for(auto e : e[x]){
        if(e == f) continue ;
        dfs1(e,x) ;
        sz[x] += sz[e] ;
        if(sz[son[x]] < sz[e]) son[x] = e;
    }
}
```

- dfs2：搞出top，id，nw

```cpp
void dfs2(int x,int y){
    top[x] = y ;//记录链头
    if(!son[x]) return ;
    dfs2(son[x],y) ;//搜索重链上的重儿子
    for(auto e : e[x]){
        if(e == fa[x] || e == son[x]) continue ;
        dfs2(e,e) ;//新开一条重链
    }
}
```

- 让两个游标沿着各自的重链向上跳，跳到同一条重链上时，深度较小的那个游标所指向的点就是 **LCA** 。

```cpp
int lca(int x, int y){
    while(top[x] != top[y]){
        if(dep[top[x]] < dep[top[y]]) swap(x,y) ;
        x = fa[top[x]] ;
    }
    return dep[x] < dep[y] ? x : y ;
}
```

Tarjan算法

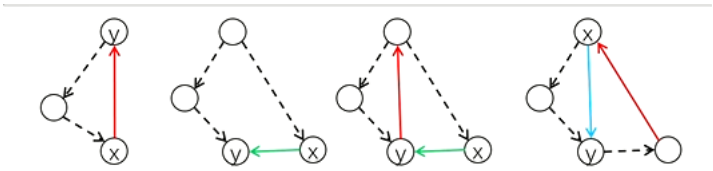
- **用途**：求解强连通分量。

DFS生成树

对图深搜时，每一个节点之访问一次，被访问过的节点和边生成的树。

有向边的访问有四种情况

1. **树边**：访问节点走过的边。黑色
2. **返祖边**：指向祖先节点的边。红色
3. **横叉边**：右子树指向左子树的边。绿色
4. **前向边**：指向子树中节点的边。蓝色



返祖边与**树边**必定构成环，**横叉边**可能与**树边**构成环（依旧需要**返祖边**），**前向边**无用。

如果节点x是某个强连通分量在**搜索树**中遇到的第一个节点，呢么这个强连通分量的剩余节点肯定在搜索树中以x为根的子树中。**节点x被称为这个强连分量的根**

塔扬算法

- **时间戳**dfn[x]：节点x第一次被访问的顺序。
- **追溯值**low[x]：从节点x出发，所能访问的最早的**时间戳**。

1. **入**x时，盖戳、入栈
2. 枚举x的邻点y，分三种情况。
    1. 若y尚未被访问：对y深搜。**回**x时，用$low[y]$更新$low[x]$。因为x是y的父节点，y能够访问到的点，x也一定能够访问到。
    2. y已经访问且在栈中：说明y是祖先节点或者左子树的节点，用$low[y]$更新$low[x]$
    3. 若y已访问并且不在栈中：说明y已经搜索完毕，其所在的连通分量已经被处理，所以不用对其进行出栈。
3. **离**x时，记录 SCC，只有遍历完一个SCC，才可以出栈。更新low值的意义：避免SCC的节点提亲出栈。

```cpp
int dfn[N], low[N], tot ;//时间戳,追溯值
int stk[N], top;//栈
bool instk[N] ;
int scc[N], siz[N], cnt ;//每个节点所属的强连通分量，每个强连通分量的节点数量
void tarjan(int x){
    dfn[x] = low[x] = ++ tot ;
    stk[++top] = x , instk[x] = true ;
    for(int y : e[x]){
        if(!dfn[y]){
            tarjan(y) ;
            low[x] = min(low[x],low[y]) ;
        }
        else if(instk[y]){//还在在栈中的
            low[x] = min(low[x],low[y]) ;
        }
    }
    if(dfn[x] == low[x]){// 得到极大连通子图
```

```
        int y  ;
        cnt ++ ;
        do{
            y = stk[top--];
            instk[y] = false ;
            siz[cnt] ++ ;
        }while(y != x) ;
    }
}
```

SCC缩点

Tarjan缩点，把**有环图**变成**无环图**，观察新图中点的入读和出度的情况，构造答案。

P2812 校园网络

- 第一问：答案是缩点后入度为 0 的点的个数。
- 第二问：答案是 $max(din == 0, dout == 0)$ 。加边后构成一个环，及所有点的入读都不为 0 且出度不为 0 。固缩点后分别统计入度为0出度为0的带你的个数，取最大值即可。

```
vector<int> e[N] ;//存图
int dfn[N], low[N] , tot;//时间戳
int scc[N], sz[N], cnt ;//分组
int stk[N], top ;//入栈
bool instk[N] ;//收收存在栈中
int din[N], dout[N] ;//缩点后入度和出读
void add(int u, int v){
    e[u].push_back(v) ;
}
void tarjan(int x){
    dfn[x] =  low[x] = ++tot ;//已经走过的
    stk[++top] = x , instk[x] = true ;//入栈
    for(auto y : e[x]){
        if(!dfn[y]){//还未走过的
            tarjan(y) ;
            low[x] = min(low[x],low[y]) ;
        }else if(instk[y]){//还在在栈中的
            low[x] = min(low[x],low[y]) ;
        }
    }
    if(dfn[x] == low[x]){
        int y;
        cnt ++ ;
        do{
            y = stk[top--] ;
            instk[y] = false ;
            sz[cnt] ++ ;
            scc[y] = cnt ;
        }while(x != y ) ;
    }

}

int s ;
void solve(){
    cin >> n  ;
    int u, v ;
    for(int i = 1; i <= n ; ++ i){
        int x ;
        while(cin >> x , x){
            e[i].push_back(x) ;
        }
    }
    for(int i = 1 ; i <= n ; ++ i){
        if(!df  n[i]) tarjan(i) ;
    }
    for(int i = 1;i <= n ; ++ i){
        for(auto y : e[i]){
            if(scc[i] != scc[y]){
                din[scc[y]] ++ ;
                dout[scc[i]] ++ ;
            }
        }
    }
    int ans2 = 0 ;
    for(int i = 1 ; i <= cnt ; ++ i){
        ans += (din[i] == 0) ;
         ans2 += (dout[i] == 0) ;
    }
    cout << ans << endl ;
    if(cnt == 1){
        cout << 0 << endl ;
        return ;
```

```
    }
    cout << max(ans,ans2) << endl ;
}
```

其他题目：

P3387 【模板】缩点
P2341 受欢迎的牛 G

tarjan割点

割点：对于一个无向图，如果把一个点删除后，联通块的数量增加了，那么这个点就是割点。

割点判定法则：

> 如果x不是根节点，当搜索树上存在x的一个子结点y，满足$low[y] \geq dfn[x]$，那么点x就是割点。
>
> 如果x是根节点，当搜索树上存在至少两个子节点$y1, y1$，满足上述条件，那么x就是割点。

$low[y] \geq dfn[x]$，说明从y出发，在不通过x点的前提下，不管走哪条边，都无法到达比x更早访问的节点。故删除x后，以y为根的子树$subtree(y)$也就断开了。 环顶的点割的掉。

反之，若$low[y] < dfn[x]$，则说明y能绕行其他边到达比x更早访问的节点，x就不是割点了，即环内的点割不掉。

```
void tarjan(int x){
    int res = 0 ;
    dfn[x] =  low[x] = ++tot ;//已经走过的
    int num = 0 ;
    for(auto y : e[x]){
        if(!dfn[y]){//还未走过的
            tarjan(y) ;
            low[x] = min(low[x],low[y]) ;
            if(low[y] >= dfn[x]){//不能够不经过x到达x节点之前已经访问过的点，
                num ++ ;
                if(x != root || num > 1){//如果x是当前的根，则需要两个这样的点（因为根节点可能只有一个子节点），反之只要一个就可以
                    st[x] = true ;
                }
            }
        }else{//能够不经过x到达x节点之前已经访问过的点
            low[x] = min(low[x],dfn[y]) ;
        }
    }
}

int s ;
void solve(){
    cin >> n >> m  ;
    int u, v ;
    for(int i = 1; i <= m ; ++ i){
        cin >> u >> v ;
        e[u].push_back(v) ;
        e[v].push_back(u) ;
    }
    for(int i = 1 ; i <= n ; ++ i){
        if(!dfn[i]){
            root = i ;
            tarjan(i) ;
        }
    }
    vector<int> vec ;
    for(int i = 1; i<= n ; ++ i){
        if(st[i]) vec.push_back(i) ;
    }
    cout << vec.size() << endl ;
    for(auto e : vec){
        cout << e << " " ;
    }
    cout << endl ;
}
```

tarjan割边

割边：对于一个无向图，如果删掉一条边后图中的连通块的数量增加了，则称这条边是割边或者桥。

割边的判定法则：

> 当搜索树上存在x的一个子结点y，满足$low[y] > dfn[x]$，则(x, y)这条边就是割边。

$low[y] > dfn[x]$，说明从y出发，在不经过(x, y)这条边的前提下，不管走哪条边，都无法到达x或更早访问的节点。故删除 (x, y)这条边，以y为根的子树$subtree(y)$也就断a开了。即**环外的边割的断。**

反之，若$low[y] \leq dfn[x]$，则说明y能绕行其他边到达x或更早访问的节点，(x, y)就不是割边了。即**环内的边割不断**

> 割点判定：$low[y] \geq dfn[x]$。允许走(x,y)的反边更新low值。
>
> 割边判定：$low[y] > dfn[x]$。不允许走(x,y)的反边更新low值。

```cpp
vector<edge> e ;//边的集
vector<int> h[N] ;//出边
edge bri[N] ;
int cnt ;
int dfn[N], low[N] , tot;//时间戳
void add(int u, int v){
    e.push_back({u,v}) ;
    h[u].push_back(e.size() - 1) ;//表示存的边的编号
}
void tarjan(int x, int in_edg){//in_edg表示入边，防止走这条边的反边
    dfn[x] =  low[x] = ++tot ;//已经走过的
    int num = 0 ;
    for(int i = 0 ; i < h[x].size() ; ++ i){
        int j =     h[x][i],y = e[j].v;//对应边，对应的另一个点
        if(!dfn[y]){//还未走过的
            tarjan(y,j) ;
            low[x] = min(low[x],low[y]) ;
            if(low[y] > dfn[x]){
                bri[++cnt] = {x,y} ;
            }
        }else if(j != (in_edg^1)){//不能是反边，因为边是一正一反存进去的，所以两条边的编号恰好为相邻的一对奇偶数
            low[x] = min(low[x],dfn[y]) ;
        }
    }
}
void solve(){
    cin >> n >> m  ;
    int u, v ;
    for(int i = 1; i <= m ; ++ i){
        cin >> u >> v ;
        add(u,v) ;
        add(v,u) ;
    }
    for(int i = 1 ; i <= n ; ++ i){
        if(!dfn[i]){
            tarjan(i,0) ;
        }
    }
    for(int i = 1 ; i <= cnt ; ++ i){
        if(bri[i].u > bri[i].v){
            swap(bri[i].u , bri[i].v);
        }
    }
    sort(bri + 1, bri + 1 + cnt, [](edge a, edge b){
        if(a.u != b.u) return a.u < b.u ;
        return a.v < b.v ;

    }) ;
    for(int i = 1 ; i <= cnt ; ++ i){
        cout << bri[i].u << " " << bri[i].v << endl ;
    }
}
```

总结

Tarjan 算法

	缩点	割点	割边
处理	有向有环图	无向图	无向图
更新 low 值	走返祖边或横叉边 更新 low 值	允许走 (x,y) 的反边 更新 low 值	不允许走 (x,y) 的反边 更新 low 值
更新 low 条件	y 已访问且在栈中	y 已访问	j 不是反边
判定时机与条件	离 x 时， low[x]==dfn[x]	回 x 时， low[y]>=dfn[x]	回 x 时， low[y]>dfn[x]
时间复杂度	$O(n+m)$	$O(n+m)$	$O(n+m)$

2-SAT 算法

什么是 $2-SAT$ 问题

给出 n 个变量 x_i，每一个变量只能取 $1/0$，给出 m 个条件，如 $x_i=0/1$或$x_j=0/1$，$2-SAT$ 就是求满足这 m 个条件的一组解。

逻辑关系

根据逻辑关系，我们就可以把每一组条件进行转化，设每一组条件中的第一个是 a，另一个是 b，那么 a 不成立 b 一定成立，并且b 不成立 a 一定成立。

$$a \vee b \Leftrightarrow (\neg a \rightarrow b) \wedge (\neg b \rightarrow a)$$

拆点建图

我们把每一个变量看作一个点，把 n 个点拆成 $2n$ 个点， 把 x_i 拆成 x_i和x_{i+n}。
建图连有向边，例如对于条件 $x_i=a$或$x_j=b$，

$x_i!=a$且$x_j=b$ 可以表示为从 i+!$a*n$ 到 j+!$b*n$ 连接一条有向边。(!表示取反)
$x_j!=b$且$x_i=a$ 可以表示为从 j+!$b*n$ 到 i+!$a*n$ 连接一条有向边。

Tarjan 缩点

1. 如果 i 和 $i+n$ 在一个 SCC 内， 说明无解，因为不能同时取 1 和 0；
2. 否则一定存在可行解。

构造可行解

如果有 $i->...->i+n$，哪那么就选择 $i+n$，$x_i=1$；
如果有 $i+n->...->i$，哪那么就选择 i，$x_i=0$；
蕴含关系具有传递性，选择后者，冲突性更小。

Tarjan 缩点后我们得到的是拓扑排序
因为先访问的节点后出栈，后出栈的节点SCC编号就大

所以如果 $SCC[i]>SCC[i+n], x_i=1$;
所以如果 $SCC[i]<SCC[i+n], x_i=0$;

最小生成树

kruskal 克鲁斯卡尔

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
const int N =1e6+10;
struct edges{
   int a,b,w;
   bool operator <(const edges &W) const{
       return w<W.w;
   }
}edge[N];

int p[N];

int find(int x){
    if(p[x]!=x) p[x] = find(p[x]);
    return p[x];
}
int n,m;
int a,b,w;
int main(){
    cin>>n>>m;
    for(int i =1 ; i<=n ; i++) p[i] = i;
    for(int i=1 ;i<=m; i++){
        cin>>a>>b>>w;
        edge[i].a = a,edge[i].b = b,edge[i].w = w;
    }
    sort(edge+1,edge+m+1);
    int res=0,cnt= 0;//cnt表示当前边的数量
    for(int i =1 ; i<=m;i++){
        int na =edge[i].a , nb =edge[i].b;
        na = find(na),nb =find(nb);
        if(na!=nb){
            p[na] = nb;
            cnt++;
            res+= edge[i].w;
```

```
        }
    }
    if(cnt<n-1) cout<<"impossible";
    else cout<<res;
    return 0;
}
```

PRM 普利姆

```
#include<iostream>
#include<vector>
#include<queue>
#include<cstring>

using namespace std;
const int N = 1e3 + 10 ,INF = 0x3f3f3f3f;

int g[N][N],n,m;
int dist[N];//距离集合的距离
bool st[N] ;
int prim(){
 memset(dist , 0x3f , sizeof dist) ;
 int res = 0;
 for(int i = 0 ;i< n; i++){
    int t =-1;
    for(int j = 1; j <= n ; j ++){
        if((t==-1||dist[t]>dist[j])&&!st[j]){
            t = j;
        }
    }

    if(i&&dist[t] ==INF) return INF;
    if(i) res+=dist[t];

    for(int j = 1 ; j <= n ; j ++){
        if(!st[j]&&dist[j]>g[t][j])
            dist[j] = g[t][j];
    }
    st[t] = true;
 }
 return res;

}

int main(){
    memset(g, 0x3f, sizeof g);
    cin>>n>>m;
    while(m--){
        int x, y ,l;
        cin>>x>>y>>l;
        g[x][y] = g [y][x] = min(g[x][y],l);
    }
    int ans = prim();
    if(ans == INF) cout<<"impossible"<<endl;
    else cout<<ans<<endl;
    return 0;
}
```

数论

筛

筛质数（欧拉筛）

```
bool isprime [N];
int prime[N],cnt;
void ss(ll n){//欧拉筛
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;//1不是素数
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]) prime[++cnt] = i%MOD;//如果i是素数，那么记录到prime当中
        for(int j = 1 ; j <= cnt && i * prime[j]%MOD <= n ; j ++){//
            isprime[i*prime[j]%MOD] = false;//筛掉i的素数倍
            if(i%prime[j]%MOD==0) break;
            //说明 i 已经被 prime[j]筛掉了
            // 所以 i * 其他质数的结果一定会被  prime[j] 的倍数筛掉
            //就不需要在这里先筛一次
            //保证每一个数都被其最小的质因子数筛掉

```

```
        }
    }
}
```

筛约数个数和约数和

约数个数

约数个数定理

若 $n=\prod_{i=1}^{m} p_i^{c_i}$ ，则 $d_i=\prod_{i=1}^{m} c_i+1$ ，（$+1$ 是因为 还有 $c_i=0$ 的情况）

实现

d_i 表示 i 的约数个数，num_i 表示最小质因子出现的次数

1. 当 i 为质数时，$num_i=1$，$d_i=2$
2. 当 p 为 q 的质因子(最小质因子，由线性筛保证)时，$num_i=num_q+1$，$d_i=d_{p*q}=\frac{d_q}{num_i}*(num_i+1)$
3. 当 p 和 q 互质时，$num_i=1, d_i=d_q*(num_i+1)$

```
bool isprime [N];
int prime[N],cnt;
int num[N], d[N] ;
void ss(int n){//欧拉筛
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;//1不是素数
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i;
            num[i] = 1 ;
            d[i] = 2 ;
        }
        for(int j = 1 ; j <= cnt && i * prime[j] <= n ; j ++){//
            isprime[i*prime[j]] = false;
            if(i%prime[j]==0){
                num[i*prime[j]] = num[q] + 1 ;
                d[i*prime[j]] = d[i] / num[i*prime[j]] * (num[i*prime[j]] + 1) ;
                break;
            }
            num[i*prime[j]] = 1 ;
            d[i*prime[j]] = d[i] << 1 ;
        }
    }
}
```

约数和

f_i 表示 i 的约数和，g_i 表示 i 的最小质因子的 $p^0+p^1+p^2+p^3+\cdots+p^n$

```
bool isprime [N];
int prime[N],cnt;
int f[N], g[N] ;
void ss(int n){//欧拉筛
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;//1不是素数
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i;
            f[i] = i + 1; //本身和 1
            g[i] = i + 1 ;// p^0, p^1
        }
        for(int j = 1 ; j <= cnt && i * prime[j] <= n ; j ++){//
            isprime[i*prime[j]] = false;
            if(i%prime[j]==0){
                g[i*prime[j]] = g[i] * prime[j] + 1 ;
                f[i*prime[j]] = f[i] / g[i] * g[i*prime[j]] ;
                break;
            }
            f[i*prime[j]] = f[i] * f[prime[j]] ;
            g[i*prime[j]] = prime[j] + 1 ;//prime[j] 就是 i * prime[j]的最
        }
    }
}
```

筛欧拉函数

```
bool isprime [N];
int phi[N] ;
int prime[N],cnt;
void ss(ll n){
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;//1不是素数
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i%MOD ;
            phi[i] = i - 1 ;
        }
        for(int j = 1 ; j <= cnt && i * prime[j]%MOD <= n ; j ++){
            isprime[i*prime[j]%MOD] = false;
            if(i%prime[j]%MOD==0){
                phi[i*prime[j]] = phi[i] * prime[j];
                break;
            }
            phi[i*prime[j]] = phi[i] * phi[prime[j]];

        }
    }
}
```

筛莫比乌斯函数

```
void getMu(int n){//欧拉筛
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;
    mu[1] = 1;// n = 1 时
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i%MOD;
            mu[i] = -1 ;//只有一个质因子其本身
        }
        for(int j = 1 ; j <= cnt && i * prime[j] <= n ; j ++){//
            isprime[i*prime[j]] = false;//筛掉i的素数倍
            mi[i*prime[j]] = prime[j] ;
            if(i%prime[j]==0){
                mu[i*prime[j]] = 0 ;//存在平方因子
                break;
            }
            mu[i*prime[j]] -= mu[i] ;//本质不同因子数增加
        }
    }
}
```

组合数学

错位排列

定义

对于 $1 \sim n$ 的排列 P，如果满足 $P_i \neq i$，则称 P 是 n 错位排列。

递推式的推理

递推式：$D_n = (n-1) * (D_{n-1} + D_{n-2})$
D_n 表示的就是 $1 \sim n$ 的错位排列的数量。

我考虑这样一个问题，我们有 n 封不同的信，编号分别为 $1, 2, 3, \ldots, n$，把这些信放进 n 个不同的邮箱当中，邮箱的编号为 $1, 2, 3, \ldots, n$，并且保证信的编号和邮箱的编号不一样。

当我们就要放第 n 个信时，我们可以分为以下两种情况。

- 前面 $n-1$ 封信全部放错，可以通过让我们要放的第 n 封信与前面的 $n-1$ 封信的其中一封信进行交换一次即可，表示出来就是 $D_{n-1} * (n-1)$
- 前面 $n-1$ 封信有一个没有放错，其他全部放错，可以通过让我们要放的第 n 封信与前面的 $n-1$ 封信的中的那封没有放错的信进行交换即可，表示出来就是 $D_{n-2} * (n-1)$。（$n-1$ 是枚举那封没有放错的信）

所以: $D_n = (n-1) * (D_{n-1} + D_{n-2})$

参考 Olwiki

多重集

多重集排列数

假设多重集一共有N个元素。那么对这N个元素全排列，除掉相同元素的全排列的积即可。也就是：

$$A=\frac{N!}{a_1!a_2\cdots!a_n!}$$

因为相同元素互换后也不会改变原有排列，所以要将相同数之间的排列除去。

多重集组合数

- 情况一

给定一个多重集$S=\{n_1\cdot a_1,n_2\cdot a_2,\dots,n_k\cdot a_k\}$，以及一个整数$m$，且有$m\le n_i(i\in[1,k])$，求从$S$中取出$m$个任意元素的**不同多重集**的数量。

$$ans=C_{k+m-1}^{k-1}$$

这种情况我们可以反过来想，我们可以把这m个元素分成k组，只不过可以有的组合是空集。所以我们可以查看成m个0和$(k-1)$个1的排列数，正好可以通过我们上面的**多重集排列数**来计算：

$$ans=\frac{(m+k-1)!}{m!(k-1)!}$$

- 情况2

给定一个多重集$S=\{n_1\cdot a_1,n_2\cdot a_2,\dots,n_k\cdot a_k\}$，以及一个整数$m$，且有$m=\sum\limits_{i=1}^{k}n_i(i\in[1,k])$，求从$S$中取出$m$个任意元素的**不同多重集**的数量。

我们先不考虑n_i的限制，那么选择m个元素的组合树就和**情况1**一样，方案数是C_{k+m-1}^{k-1}。

但是这样的计算显然多算了$x_i>n_i$的不合法方案，也就是说我们还要减去至少包含了n_i+1个 a_i的情况（我们设这种情况为S_i），我们先从 S中取出n_i+1个a_i再在S中任选$m-n_i-1$个元素得到$C_{k+m-n_i-2}^{k-1}$

但是我们又多减去了$S_i\cap S_j$的情况（也就是同时包含了n_i+1个a_i和n_j+1个a_j的情况），所以我们要再加上$C_{k+m-n_i-n_j-3}^{k-1}$

进行容斥原理得到

$$ans=C_{k+m-1}^{k-1}-\sum_{i=1}C_{k+m-n_i-2}^{k-1}+\sum_{1\le i,j\le k,i\ne j}C_{k+m-n_i-n_j-3}^{k-1}-\cdots+(-1)^kC_{k+m-\sum_{i=1}^{k}-(k+1)}^{k-1}$$

卡特兰数

通项公式：

(1) $H_n=C_{2n}^{n}-C_{2n}^{n+1}$
(2) $H_n=\frac{\binom{2n}{n}}{n+1}=\frac{1}{n+1}C_{2n}^{n}=\frac{(2n)!}{(n+1)!n!}(\ n\ge2,n\in N_+)$
(3) $H_n=H_0*H_{n-1}+H_1*H_{n-2}+H_2*H_{n-3}+\cdots+H_{n-1}*H_0$
(4) $H_n=\frac{4n-2}{n+1}H_{n-1}$

H_0	H_1	H_2	H_3	H_4	H_5	H_6	$\dots$
1	1	2	5	14	42	132	...

证明 (1) 式：

以走网格数为例子，从格点 $(0,0)$ 走到 (n,n) ，只能**向右或向上走**，并且不能越过对角线的路径的条数，就是卡特兰数。记为 H_n。

- 我们可以先求出**路径总数**，一共向右移动了 n 次，显然为 C_{2n}^{n} 。
- 对于**非法路径。**
  我们先把 $y=x+1$ 这条线画出来，我们可以发现所有**非法路径**都一定与这条线至少有一个交点，从其第一个交点处开始沿 $y=x+1$ 进行对称翻转，发现终点就会变成 $(n-1,n+1)$ 。
  我们可以发现每一个**非法路径**都对应这样的一个对称翻转路径到达 $(n-1,n+1)$。
  我们由此可以发现非法的路径数也就等于从 $(0,0)$ 到 $(n-1,n+1)$ 的路径总数，所以**合法路径**数就是**总路径**数 C_{2n}^{n} 减去**非法路径**数 $C_{2n}^{n+1}=C_{2n}^{n-1}$，$C_{2n}^{n}-C_{2n}^{n+1}$。(如果是求 $y\le x+k$的方案数， 就可以转化为 $C_{n+m}^{n}-C_{n+m}^{n++k+1}$)

证明 (2) 式：

$$\begin{aligned}H_n&=C_{2n}^{n}-C_{2n}^{n+1}\\&=\frac{(2n)!}{n!n!}-\frac{(2n)!}{(n+1)!(n-1)!}\\&=\frac{(2n)!}{n!(n-1)!}(\frac{1}{n}-\frac{1}{n+1})\\&=\frac{(2n)!}{n!n!(n+1)}\\&=\frac{1}{n+1}C_{2n}^{n}\end{aligned}$$

证明(3)式

我们拿括号组合来举例，一共 n 个左括号和 n 个右括号共 n 对括号。我们假设最后一个右括号已经和第 i 个（左括号之中的）左括号匹配，这说明之前已经有 $i-1$ 个左括号已经匹配完毕，此种情况是 $H_{i-1}*H_{n-i}$，H_{n-i} 这里不是 H_{n-i+1} 是因为 n 和 i 已经匹配好了。

证明(4)式

$$H_{n-1}=\frac{1}{n}C_{2(n-1)}^{n-1}=\frac{(2(n-1))!}{n(n-1)!(n-1)!}=\frac{(2n-2)!}{n!(n-1)!}$$

$$\begin{aligned}\frac{4n-2}{n+1}H_{n-1} &= \frac{2(2n-1)}{n+1}*\frac{(2n-2)!}{n!(n-1)!}\\ &= \frac{2(2n-1)!}{(n+1)n!(n-1)!}\\ &= \frac{2n(2n-1)!}{(n+1)n!(n-1)!n}\\ &= \frac{(2n)!}{(n+1)n!n!}\\ &= \frac{1}{n+1}C_{2n}^{n}\\ &= H_n\end{aligned}$$

卡特兰数的特征：

一种操作数不能超过另外一种操作数，或者两种操作不能有交集，这些操作的合法方案书，通常是**卡特兰数**。

具体应用：

1. 一个有 n 个 0 和 n 个 1 组成的字符串，且所有前缀都满足 1 的个数不超过 0 的个数。这样的串有多少个。
   把 1 看成向上走，0 看成向右走，
2. 包含 n 组括号的合法运算式的个数有多少个?
   右括号看成向上走，左括号看成向右走。
3. 一个栈的进栈序列是 $1,2,3,\ldots,n-1,n$，有多少种不同的出栈序列。
   出栈看成向上走，进栈看成向上走，
4. n 个节点可以构造多少个不同的二叉树。
   可以用**公式(3)**来推，我们把子节点分别分配在左子树和右子树上，形式就和**公式(3)**一样。
5. 在圆上选择 $2n$ 个点，并将这些点成对连接起来使得所得到的 n 条弦互不相交的方法。
   偶数点为向上走，偶数点为向右走。（奇偶交换也一样）
6. 通过连结顶点而将 $n+2$ 边的凸多边形分成 n 个三角形的方法数。
   同构造二叉树，把多边形划分成两个新的多边形，然后在两个新的多边形中继续进行划分。

卢卡斯定理

大组合数取模

给定整数 N,M,P,求出 $C_n^m(mod\ \ p)$的值。
其中$1\leq m\leq n\leq 10^{18},1\leq p\leq 10^5$，保证 p 为质数。

卢卡斯定理

$C_n^m<spanclass=\prime\prime md-search-hit\prime\prime>\equivC_{n/p}^{m/p}C_{n\ mod\ p}^{m\ mod\ p}(mod\ \ p)$其中 p 为质数。

$n\ mod\ p$和$m\ mod\ p$一定是小于 p 的质数，可以直接求解，$C_{n/p}^{m/p}$可以继续用Lucas定理求解。

边界条件：当 $m=0$ 时，返回 1 。

```
int getC(int n ,int m){
    return f[n] * g[m] * g[n - m] % mod ;
}
int lucas(int n ,int m){
    if(m == 0) return 1 ;
    return lucas(n/mod,m/mod) * getC(n % mod ,m % mod) % mod ;
}
```

引理1：$C_p^x\equiv 0(mod\ p)$

引理2：$(1+x)^p\equiv 1+x^p(mod\ p)$
由二项式定理可知 $1+x^p=\sum_{j=0}^{p}C_p^ix^i$,
由引理1可知，只剩下 $i=0,p$ 两项。

矩阵快速幂

矩阵快速幂是一种高效计算线性递推关系的方法，常用于求解斐波那契数列等问题。其基本原理是将递推关系转化为矩阵乘法，通过快速幂算法减少计算复杂度。具体步骤如下：

> **建立递推式**：首先列出递推关系，例如 f(n) = f(n-1) + f(n-2)。
> **构造转移矩阵**：将递推关系转化为矩阵形式，通常使用一个 2x2 的常数矩阵 T。
> **快速幂计算**：利用快速幂算法计算 T 的 n 次方，从而得到所需的结果。
> **这种方法的时间复杂度为** $O(n^3 logn)$，大大提高了计算效率。

```
struct Matrix {
    int n, m, s[105][105] ;
    void clear(){
        memset(s, 0, sizeof s) ;
    }
    void indentity(){//生成单位矩阵
        for(int i = 0 ; i < n ; ++ i){
            s[i][i] = 1 ;
        }
    }
    Matrix(int n,int m) :n(n), m(m){
        memset(s, 0, sizeof s) ;
    }
};
Matrix mul(const Matrix & a, const Matrix & b){
    Matrix c(a.n,b.m) ;
    for(int i = 0 ; i < a.n ; ++i){
        for(int j = 0 ; j < b.m ; ++ j){
            for(int k = 0 ; k < a.m ; ++ k){
                c.s[i][j] = (c.s[i][j] + a.s[i][k] * b.s[k][j] % MOD + MOD) % MOD ;
            }
        }
    }
    return c ;
}
Matrix pow(Matrix & a, int p){
    Matrix res(a.n,a.n) ;
    res.indentity() ;
    Matrix base = a ;
    while(p){
        if(p & 1) res = mul(res,base) ;
        base = mul(base,base) ;
        p >>= 1 ;
    }
    return res ;
}
Matrix init(){
    Matrix res(ss,ss) ;
    for(int i = 0 ; i < ss ; ++ i){
        int t = i ^ (ss - 1) ;
        for(int j = 0 ; j < ss ; ++ j){
            if((j & t) == t){
                res.s[i][j] = cnt[i&j] ;
            }
        }
    }
    return res ;
}
```

康托展开

康托展开

康托展开是一个全排列到自然数的映射。康托展开的实质是计算当前排列在所有由小到大全排列中的顺序，因此是可逆的。

公式:

$$X = a_n(n-1)! + a_{n-1}(n-2) + \cdots + a_1 * 0!$$

- a_i 表示在位置 i 后面并且小于 a_i 的值的个数。

比如所 2143 的康托展开就是

$$\begin{aligned} X &= 1*(4-1)! + 0*(4-2)! + 1*(4-3)! + 0*0! \\ &= 3! + 1 \\ &= 7 \end{aligned}$$

也就是说这是第 8 个全排列（第一个全排列的康托展开是 0）。

```
int n, m, k;
int a[N] ;//排列
int fac[N] ;//阶乘
int b[N] ;//树状数组
int lowbit(int x){
    return x & -x ;
}
void add(int x, int k){
```

```
    while(x <= n){
        b[x] += k ;
        x += lowbit(x) ;
    }
}
int querry(int x){
    int t = 0 ;
    while(x){
        t += b[x] ;
        x -= lowbit(x) ;
    }
    return t ;
}
void solve(){
    cin >> n ;
    for(int i = 1; i <= n ; ++ i){
        cin >> a[i] ;
    }
    ans = 1 ;
    for(int i = n ; i >= 1; -- i ){//倒着来进行公式
        int num = querry(a[i]) ;
        ans = (ans +  num * fac[n-i] % MOD) % MOD ;
        add(a[i],1) ;
    }
    cout << ans << endl ;

}
signed main(){
    IOS;
    fac[1] = 1 ;
    for(int i = 2; i<= 1e6 ; ++ i){
        fac[i] = fac[i-1] * i % MOD;
    }
    while (T--){
        solve();
    }
    return 0;
}
```

逆康托展开

还以 7 和 2143 为例，

- $\frac{7}{(4-1)!}=\frac{7}{3!}=\frac{7}{6}$，商 1 余 1，说明在第一位后面只有一个比该为小的数，呢么就说明该位是第 2 小的数，即 2。
- $\frac{1}{(4-2)!}=\frac{1}{2}$，商 0 余 1，说明该位置后没有比当前位置还小的数，所以就填当前能填的最小的数，即 1。
- $\frac{1}{(4-3)!}=\frac{1}{1}$，商 1 余 0，说明该位置后有一个比当前位置还小的数，，所以就填当前能填的第 2 小的数，即 4。
- 最后剩下一个数 3 直接填到末尾即可。

```
int n, m, k;
int a[N] ;
int fac[N] ;
void solve(){
    cin >> n >> k ; //长度和位于排列中的第几个数
    k -- ;
    vector<int> vec ;
    vector<int> res ;
    for(int i = 1 ; i <= n ; ++ i) vec.push_back(i) ;
    for(int i = n ; i > 1; -- i){
        int pos = 0 ;
        if(i-1){
            pos = k / fac[i-1] ;
            k %= fac[i-1] ;
        }
        res.push_back(vec[pos]) ;
        vec.erase(vec.begin() + pos) ;
    }
    res.push_back(vec[0]) ;

    for(auto e : res){
        cout << e << " " ;
    }
    cout << endl ;

}
signed main(){
    IOS;
    fac[1] = 1 ;
    for(int i = 2; i<= 1e6 ; ++ i){
        fac[i] = fac[i-1] * i % MOD;
    }
```

```
    while (T--){
        solve();
    }
    return 0;
}
```

扩展欧几里得

```
// 求x, y, 使得ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y){
    if (!b){
        x = 1; y = 0;
        return a;   //到达递归边界开始向上一层返回
    }
    int d = exgcd(b, a % b, x, y);
    int temp=y;    //推出这一层的x, y
    y=x-(a/b)*y;
    x=temp;
    return d;
}
```

莫比乌斯反演

狄利克雷生成函数

$F(x)G(x)=\sum_{n=1}^{\infty}\frac{1}{n^x}\sum_{d|n}a_d b_{\frac{n}{d}}$

莫比乌斯函数

如果 n 含有平方因子，那么 $\mu(n)=0$；否则 $\mu(n)=(-1)^k$，其中 k 为 n 的本质不同的质因子个数。

莫比乌斯反演

$$\mu(n)=\begin{cases}1, & n=1\\ 0, & \text{存在平方因子}\\ (-1)^k, & k\text{为本质不同质因子数量}\end{cases}$$

性质

$$\sum_{d|n}\mu(d)=\begin{cases}1, & n=1\\ 0, & n\neq 1\end{cases}$$

补充结论

$$[gcd(i,j)=1]=\sum_{d|gcd(i,j)}\mu(d)$$

线性筛计算 μ

和欧拉筛基本相同

```
void getMu(int n){//欧拉筛
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;
    mu[1] = 1;// n = 1 时
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i%MOD;
            mu[i] = -1 ;//只有一个质因子其本身
        }
        for(int j = 1 ; j <= cnt && i * prime[j] <= n ; j ++){//
            isprime[i*prime[j]] = false;//筛掉i的素数倍
            mi[i*prime[j]] = prime[j] ;
            if(i%prime[j]==0){
                mu[i*prime[j]] = 0 ;//存在平方因子
                break;
            }
            mu[i*prime[j]] -= mu[i] ;//本质不同因子数增加
        }
    }
```

```
}
```

问题形式

求值（多组数据）

$\sum_{i=x}^{n}\sum_{j=y}^{m}[\gcd(i,j)=k]\qquad(1\leqslant T,x,y,n,m,k\leqslant 5\times 10^4)$

根据容斥原理，原式可以分成 块来处理，每一块的式子都为

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

考虑化简该式子

$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}[\gcd(i,j)=1]$

因为 时对答案才用贡献，于是我们可以将其替换为 （ 当且仅当 时值为 否则为 ），故原式化为

$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}\varepsilon(\gcd(i,j))$

将 函数展开得到

$$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}\sum_{d\mid\gcd(i,j)}\mu(d)$$

变换求和顺序，先枚举 可得

$$\sum_{d=1}\mu(d)\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}[d\mid i]\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}[d\mid j]$$

易知 中 的倍数有 个，故原式化为

$$\sum_{d=1}^{\min(\lfloor\frac{n}{k}\rfloor,\lfloor\frac{m}{k}\rfloor)}\mu(d)\lfloor\frac{n}{kd}\rfloor\lfloor\frac{m}{kd}\rfloor$$

很显然，式子可以数论分块求解。

逆元

线性求法

```
inv[1] = 1;
for (int i = 2; i <= n; ++i) {
  inv[i] = (long long)(p - p / i) * inv[p % i] % p;
}
```

扩展欧几里得

```
//求解 a 在取 b 模数 的情况下的逆元 x
int exgcd(int a, int b, int &x, int &y){
    if (!b){
        x = 1; y = 0;
        return a;   //到达递归边界开始向上一层返回
    }
    int d = exgcd(b, a % b, x, y);
    int temp=y;    //推出这一层的x，y
    y=x-(a/b)*y;
    x=temp;
    return d;
}
```

欧拉函数

欧拉函数

定义

$\varphi(n)$ 表示的是小于等于 n 和 n 互质的数的个数。

性质

- 欧拉函数是积性函数
- 当 n 为奇数的时候 $\varphi(2n)=\varphi(n)$
- $n=\sum_{d\mid n}\varphi(d)$
- 若 $n=p^k$，其中 p 为质数，那么 $\varphi(n)=p^k-p^{k-1}$
- 由唯一分解定理，设 $b=\prod_{i=1}^{s}p_i^{k_i}$，其中 p_i 是质数，有 $\varphi(n)=n*\prod_{i=1}^{s}\frac{p_i-1}{p_i}$

证明

> 我们由**性质4**可以知道当 p 为质数时，$\varphi(n) = p^k - p^{k-1}$，又因为**唯一分解定理**和欧拉函数是**积性函数**，我们就可以得到 $\mu(n) = \prod_{i=1}^{n} \mu(p_i^{k_i})$，最后化简就可以得到
>
> $$\varphi(n) = n * \prod_{i=1}^{s} \frac{p_i - 1}{p_i}$$

$gcd(i,j)$ 可以转化为 $\sum_{d|gcd(i,j)} \varphi(d)$

欧拉定理

若 $gcd(a,m) = 1$，则 $a^{\varphi(m)} \equiv 1 \ (mod\ m)$

因数分解求欧拉函数

```cpp
#include<iostream>
#define endl '\n'
using namespace std;
int n,x,res;
int main(){
    cin>>n;
    while(n--){
        cin>>x;
        res =x;
        for(int i =2 ; i <= x / i ; i ++){
            if(x%i == 0){
                res = res / i * (i-1);
                while(x%i ==0) x/=i;
            }
        }
        if( x >1) res = res / x * (x-1);
        cout<<res<<endl;
    }

    return 0;
}
```

筛法求欧拉函数 (根据**性质5**)

在线性筛中，每一个和书都是被其最小质因子筛掉的，我们假设 p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，那么线性筛中 n 是由 $n' * p_1$ 筛掉的。

如果 $n'\ mod\ p1 = 0$，那么 n' 包含了 n 的所有质因子。

$$\varphi(n) = n * \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = n' * p_1 * \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = p_1 * \varphi(n')$$

若 $n'\ mod\ p1 \neq 0$，此时 n' 和 p_1 是互质的

$$\varphi(n) = \varphi(n') * \varphi(p_1)$$

```cpp
bool isprime [N];
int phi[N] ;
int prime[N],cnt;
void ss(ll n){
    memset(isprime , true, sizeof isprime);
    isprime[1] = false;//1不是素数
    for(int i =2 ; i <= n ; i ++){
        if(isprime[i]){
            prime[++cnt] = i%MOD ;
            phi[i] = i - 1 ;
        }
        for(int j = 1 ; j <= cnt && i * prime[j]%MOD <= n ; j ++){
            isprime[i*prime[j]%MOD] = false;
            if(i%prime[j]%MOD==0){
                phi[i*prime[j]] = phi[i] * prime[j];
                break;
            }
            phi[i*prime[j]] = phi[i] * phi[prime[j]];

        }
    }
}
```

生成函数

生成函数

某个序列a的生成函数是一种**形式幂级数**，其每一项的系数可以提供关于这个序列的信息。根据问题的不同，可以构造不同形式的生成函数。

包括**普通生成函数、指数生成函数、狄利克雷生成函数。**

普通生成函数(OGF)

可以用来解决多重组合数问题，**指数**即物品个数，**系数**即组合数。

$$f(x)=\sum_{i\geq 0}a_ix^i$$

a_n可以是有穷序列，也可以是无穷序列。例如，
<1,2,3>：$1+2x+3x^2$
<1,1,1>：$1+x+x^2+\cdots=\sum_{n\geq 0}x^n$

加减运算

$$F(x)\pm G(x)=\sum_{i\geq 0}a_ix^i\pm\sum_{j\geq 0}b_jx^j=\sum_{n\geq 0}(a_n\pm b_n)x^n$$

乘法运算（卷积）

$$F(x)G(x)=\sum_{i\geq 0}a_ix^i\sum_{j\geq 0}b_jx^j=\sum_{n\geq 0}x^n\sum_{i=0}^{n}a_ib_{n-i}(令i+j=n)$$

 image-20241012163629697

例如$n=3$时，x^3的系数就有$a_0b_3+a_1b_2+a_2b_1+a_3b_0$

指数型生成函数

$F(x)=\sum_{n\geq 0}a_n\frac{x^n}{n!}$

$F(x)G(x)=\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=0}^{n}C_n^ia_ib_{n-i}$

	普通生成函数	指数生成函数
定义	$F(x)=\sum a_nx^n$	$F(x)=\sum a_n\frac{x^n}{n!}$
卷积	$\sum_{i\geq 0}a_ix^i\sum_{j\geq 0}b_jx^j=\sum_{n\geq 0}x^n\sum_{i=0}^{n}a_ib_{n-i}$	$\sum_{i\geq 0}a_i\frac{x^i}{i!}\sum_{j\geq 0}b_j\frac{x^j}{j!}=\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=0}^{n}\frac{n!}{i!\,(n-i)!}a_ib_{n-i}$
构造	$(1+x^1+\cdots+x^{a_1})\cdots(1+x^1+\cdots+x^{a_n})$	$(1+\frac{x^1}{1!}+\frac{x^2}{2!}+\cdots+\frac{x^{a_1}}{a_1!})\cdots(1+\frac{x^1}{1!}+\frac{x^2}{2!}+\cdots+\frac{x^{a_n}}{a_n!})$
用途	多重集组合数 x^m 的**系数**即组合数	多重集排列数 $\frac{x^m}{m!}$ 的**系数**即排列数

狄利克雷生成函数

$F(x)=\frac{a_1}{1^x}+\frac{a_2}{2^x}+\frac{a_3}{3^x}+\cdots=\sum_{n=1}^{\infty}\frac{a_n}{n^x}$

乘法运算

$F(x)G(x)=\sum_{n=1}^{\infty}\frac{1}{n^x}\sum_{d|n}a_db_{\frac{n}{d}}$

数论分块

富比尼定理

用两种不同的方法计算同一个量，从而建立相等关系。

引理1

$$\lfloor \frac{a}{bc} \rfloor = \lfloor \frac{\lfloor \frac{a}{b} \rfloor}{c} \rfloor$$

引理2

$$|\{ \lfloor \frac{n}{d} \rfloor \mid d \in N_+, d \le n \}| \le \lfloor 2\sqrt{n} \rfloor$$

当 $d \le \lfloor \sqrt{n} \rfloor$ 时，有 $\lfloor \sqrt{n} \rfloor$ 种取值
当 $d > \lfloor \sqrt{n} \rfloor$ 时，有 $\lfloor \sqrt{n} \rfloor$ 种取值

数论分块结论

> 向下取整的数论分块

对于常数 n

$$\lfloor \frac{n}{i} \rfloor = \lfloor \frac{n}{j} \rfloor$$

成立且满足 $i \le j \le n$ 的 j 的最大值为 $\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \rfloor$，即 $\lfloor \frac{n}{i} \rfloor$ 所在块的右端点为 $\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \rfloor$

> 向上取整的数论分块

$$\lceil \frac{n}{i} \rceil = \lceil \frac{n}{j} \rceil$$

成立且满足 $i \le j \le n$ 的 j 的最大值为 $\lfloor \frac{n-1}{\lfloor \frac{n-1}{i} \rfloor} \rfloor$，即 $\lfloor \frac{n}{i} \rfloor$ 所在块的右端点为 $\lfloor \frac{n-1}{\lfloor \frac{n-1}{i} \rfloor} \rfloor$

N 维数论分块

求含有 $\lfloor \frac{a_1}{i} \rfloor$、$\lfloor \frac{a_1}{i} \rfloor$、$\lfloor \frac{a_1}{i} \rfloor \dots$ 的和式的数论分块，呢么右端点就会变成 $min_{j=1}^{n}\{\frac{a_j}{\lfloor \frac{a_j}{i} \rfloor}\}$

数论分块的扩展

$$\lfloor \sqrt{\frac{n}{p}} \rfloor = \lfloor \sqrt{\frac{n}{q}} \rfloor$$

成立最大的 q 需要满足 $p \le q \le n$ 为

$$\lfloor \frac{n}{\sqrt{n/p}^2} \rfloor$$

数位之和被整除

被 3 整除

一个数的所有位数之和能被 3 或 9 整除。呢么这个数就能被 3 或 9 整除

```
abcd=1000a+100b+10c+d
    =999a+99b+9c+a+b+c+d
    =9×(111a+11b+c)+a+b+c+d
```

所以最后只要 $a+b+c+d$ 能被 3 整除即可 。

被 7，11，13 整除

对于一个数 $a_1a_2a_3a_4a_5$

从左到右每三位一截 a_1a_2 , $a_3a_4a_5$ ，奇数位置之和和偶数位置之和的差值能被 7 (11, 13) 整除，说明该数就能被 7 (11, 13) 整除。对于该例子就是

$a_3a_4a_5 - a_1a_2$ 。

适用于X位截断奇偶位之差判断整除性的除数

截断法	适用的除数
一位截断	11
二位截断	101
三位截断	7，11，13，77，91，143，1001
四位截断	73，137，10001
五位截断	11，9091，100001

适用于X位截断后奇偶位之和判断整除性的除数

截断法	适用的除数
一位截断法	3,9
二位截断法	3,9,11,33,99
三位截断法	3,9,27,37,111,333,999
四位截断法	3,9,11,33,99,101,303,909,1111,3333,9999
五位截断法	3,9,41,123,271,369,813,2439,11111,33333,99999

约瑟夫环问题

问题描述

n 个人标号$0,1,\cdots$。逆时针站一圈，从0号开始，每一次从当前的人逆时针数 k个，然后让这个人出局。问最后剩下的人是谁。

递归推理

因为循环链表和数组模拟都比较好理解，所以直接写递归的

我们设$f(n,k)$是当前映射下最后剩下的人编号，例如显然$f(1,k)=1$。

$1,2,3,4...,5,\ldots,n$我们第一次删除数后，序列变成$1,2,3,4\cdots,k-1,k+1,\ldots,n$我们按照从$k+1$开始重新映射后得到为

```
old    new

k+1   1
k+2   2
.
.
n       n-k
1       n-k+1
2       n-k+2
.
.
k-2     n-2
k-1     n-1
```

由此我们可以推出映射关系

$$new=(old-k+n-1)\%n+1$$
$$old=(new+k-1)\%n+1$$

我们的递推式就可以写成$f(n,k)=(f(n-1)+k-1)\%n+1$,又因为$f(1,k)=1$,所以我们就可以列出下式。

$$\begin{cases} f(n)=(f(n-1)+k-1)\%n+1, & n>1 \\ f(1)=1, n==1 \end{cases}$$

写成递推式就是$ans=(ans+k-1)\%i+1$

O(n)

```
void solve(){
    cin >> n >> k ;
    int ans = 0 ;
    for(int i = 1;i <= n ; ++ i){
        ans = (ans + k - 1) % i + 1 ;
    }
    cout << ans << endl ;
    return ;
}
```

O(klog(n))

```
int josephus(int n, int k) {
  if (n == 1) return 0;
  if (k == 1) return n - 1;
  if (k > n) return (josephus(n - 1, k) + k) % n;  // 线性算法
  int res = josephus(n - n / k, k);
  res -= n % k;
  if (res < 0)
    res += n;  // mod n
  else
    res += res / (k - 1);  // 还原位置
  return res;
}
```

拉格朗日插值法

问题

已知 n 个点 $(x_1,y_1),(x_2,y_2)\dots,(x_n,y_n)$ 构造一个多项式函数 $f(x)$ 满足过这 n 个点。

实现

构造 n 个函数 $f_1(x),f_2(x),\dots f_n(x)$ ，使得第 i 个函数经过 (x_i,y_i) ，和 $(x_j,0)$ ，$j\neq i$，则所求函数 $f(x)=\sum_{i=1}^{n}f_i(x)$

设 $f_i(x)=a\prod_{i\neq j}(x-x_j)$ ，将点 (x_i,y_i) 代入可得 $a=\dfrac{y_i}{\prod_{j\neq i}*(x_i-x_j)}$

$$f(x)=\sum_{i=1}^{n}\prod_{j\neq i}\frac{x-x_i}{x_i-x_j}y_i$$

```
int x[N], y[N] ;
void solve(){
    cin >> n >> k ;
    for(int i = 1; i <= n ; ++ i){
        cin >> x[i] >> y[i] ;
    }
    for(int i = 1; i <= n ; ++ i){
        int a = y[i], b = 1 ;
        for(int j = 1; j <= n ; ++ j){
            if( i == j) continue ;
            a = a * (k - x[j]+MOD) % MOD ;
            b = b * (x[i] - x[j] + MOD) % MOD ;
        }
        ans = (ans + a * qmi(b,MOD-2)%MOD)%MOD ;
    }
    cout << ans << endl ;
}
```

快速傅里叶变换 FFT

单位圆

圆上的点 $z=cos\theta+i\ sin\theta$

把圆分成 n 等份，可得方程 $z^n=1$ 的 n 个复数根

单位根 $\omega_n^k=cos\dfrac{2\pi k}{n}+i\ sin\dfrac{2\pi k}{n}$

$\omega_n^k\omega_n^m=\omega_n^{k+m}$

$(\omega_n^k)^m=\omega_n^{km}$

FFT 要求 $n=2^{b\in N}$

1. 周期性 $\omega_n^{k+n}=\omega_n^k$
2. 对称性 $\omega_n^{k+\frac{n}{2}}=-\omega_n^k$
3. 折半性 $\omega_n^{2k}=\omega_{n/2}^k$

由系数求点值（正变换）

设 $A(x)$ 的系数是 $(a_1, a_2, a_3, \dots, a_{n-1})$

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \dots a_{n-1} x^{n-1}, \\ &= (a_0 + a_2 x^2 + \dots a_{n-2} x^{n-2}) + (a_1 + a_3 x^2 + \dots a_{n-1} x^{n-2}) x, \end{aligned}$$

设
$A_1(x) = a_0 + a_2 x + \dots a_{n-2} x^{\frac{n}{2}-1}$
$A_2(x) = a_1 + a_3 x + \dots a_{n-2} x^{\frac{n}{2}-1}$

则
$A(x) = A_1(x^2) + A_2(x^2)x$

将 $\omega_n^k (k > \frac{n}{2})$ 代入得

$$\begin{aligned} A(\omega_n^k) &= A_1(\omega_n^{2k}) + A_2(\omega_n^{2k})\omega_n^k \\ &= A_1(\omega_{\frac{n}{2}}^k) + A_2(\omega_{\frac{n}{2}}^2)\omega_n^k \end{aligned}$$

将 $\omega_n^{k+\frac{n}{2}}$ 代入得

$$\begin{aligned} A(\omega_n^{k+\frac{n}{2}}) &= A_1(\omega_n^{2k+n}) + A_2(\omega_n^{2k+n})\omega_n^{k+\frac{n}{2}} \\ &= A_1(\omega_{\frac{n}{2}}^k) - A_2(\omega_{\frac{n}{2}}^2)\omega_n^k \end{aligned}$$

A数组初值，实部存系数，虚部存 0，

递归

```cpp
using Comp = std::complex<double>;
void FFT(Comp A[],int n,int inv){
    if(n == 1) return ;
    Comp A1[n/2], A2[n/2] ;
    for(int i = 0 ; i < n / 2;  ++ i){
        A1[i] = A[i*2] ;
        A2[i] = A[i*2+1] ;
    }
    FFT(A1,n/2,inv), FFT(A2,n/2,inv) ;
    Comp w1(cos(2 * PI / n),inv * sin(2 * PI / n)) ;
    Comp w2(1, 0) ;
    for(int i = 0 ; i < n / 2; ++ i){
        A[i] = A1[i] + A2[i] * w2 ;
        A[i + n / 2] = A1[i] - A2[i] * w2 ;
        w2 = w2 * w1 ;
    }
}
```

迭代

```cpp
using Comp = std::complex<double>;
int R[N] ;
void change(Comp A[], int n){
    for(int i = 0 ; i < n ; ++ i){
        R[i] = R[i/2]/2 + ((i & 1LL)? n/2 : 0) ;
    }
    for(int i = 0 ; i < n ; ++ i){
        if(i < R[i]) swap(A[i],A[R[i]]) ;
    }
}
void FFT(Comp A[],int n,int inv){
    change(A,n) ;
    for(int m = 2 ; m <= n ; m <<= 1){
        Comp w1(cos(2.0 * PI / m), inv * sin(2.0 * PI / m));
        for(int i = 0 ; i < n ; i += m){
            Comp wk(1,0) ;
            for(int j = 0 ; j < m / 2 ; ++ j){
                Comp x = A[i + j] , y = A[i + j  + m / 2] * wk ;
                A[i + j] = x + y ;
                A[i + j + m / 2] = x - y ;
                wk = wk * w1 ;
            }
        }
    }
}
```

细节

1. 确定 m 次 多项式，至少取 $m+1$ 个点，选取 n 个单位根，必须满足：$m < n = 2^{b \in N}$, 例如 $n, m \le 10^6$， 数组的大小就开 4×10^6 `for(m = n + m , n = 1; n <= m ; n <<= 1) ;`
2. 浮点数计算有误差，例如 $[5.9999998 + 0.5] = 6$

板子

```cpp
const int mod = 998244353;
namespace FFT {
typedef double db;
const db PI = acos(-1.0);
struct Comp {
    db x, y;
    Comp() {}
    Comp(db _x, db _y): x(_x), y(_y) {}
    Comp operator + (const Comp&rhs)const {
        return Comp(x + rhs.x, y + rhs.y);
    }
    Comp operator - (const Comp&rhs)const {
        return Comp(x - rhs.x, y - rhs.y);
    }
    Comp operator * (const Comp&rhs)const {
        return Comp(x * rhs.x - rhs.y * y, x * rhs.y + y * rhs.x);
    }
};
Comp conj(const Comp&rhs) {
    return Comp(rhs.x, -rhs.y);
}
Comp exp_i(const db &x) {
    return Comp(cos(x), sin(x));
}
const int L = 20, N = 1 << L;
Comp roots[N];
int rev[N];
void init() {
    for (int i = 0; i < N; i++) {
        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << L - 1);
    }
    roots[1] = {1, 0};
    for (int i = 1; i < L; i++) {
        db angle = 2 * PI / (1 << i + 1);
        for (int j = 1 << i - 1; j < 1 << i; j++) {
            roots[j << 1] = roots[j];
            roots[j << 1 | 1] = exp_i((j * 2 + 1 - (1 << i)) * angle);
        }
    }
}
inline void f(Comp &a, Comp &b, const Comp &c) {
    Comp d = c * b;
    b = a - d;
    a = a + d;
}
void fft(vector<Comp> &a, int n) {
    assert((n & (n - 1)) == 0);
    int zeros = __builtin_ctz(n), shift = L - zeros;
    for (int i = 0; i < n; i++)
        if (i < (rev[i] >> shift))
        { swap(a[i], a[rev[i] >> shift]); }
    for (int i = 1; i < n; i <<= 1)
        for (int j = 0; j < n; j += i * 2)
            for (int k = 0; k < i; k++)
            { f(a[j + k], a[i + j + k], roots[i + k]); }
}
vector<Comp> fa, fb;
vector<int> multiply(const vector<int> &a, const vector<int> &b) {
    int la = a.size(), lb = b.size();
    int need = la + lb - 1, n = 1 << (32 - __builtin_clz(need - 1));
    if (n > fa.size()) {
        fa.resize(n);
    }
    for (int i = 0; i < n; i++) {
        fa[i] = Comp(i < la ? a[i] : 0, i < lb ? b[i] : 0);
    }
    fft(fa, n);
    Comp r(0, -0.25 / n);
    for (int i = 0, j = 0; i <= (n >> 1); i++, j = n - i) {
        Comp x = fa[i] * fa[i], y = fa[j] * fa[j];
        if (i != j) {
            fa[j] = (x - conj(y)) * r;
        }
        fa[i] = (y - conj(x)) * r;
    }
    fft(fa, n);
    vector<int> c(need);
    for (int i = 0; i < need; i++) {
        c[i] = fa[i].x + 0.5;
    }
    return c;
```

```
}
const int M = (1 << 15) - 1;
vector<int> multiply_mod(const vector<int> &a, const vector<int> &b, bool eq = 0) {
    int la = a.size(), lb = b.size();
    int need = la + lb - 1, n = 1 << (32 - __builtin_clz(need - 1));
    if (fa.size() < n) {
        fa.resize(n);
    }
    if (fb.size() < n) {
        fb.resize(n);
    }
    for (int i = 0; i < n; i++) {
        fa[i] = i < la ? Comp(a[i] >> 15, a[i] & M) : Comp(0, 0);
    }
    fft(fa, n);
    if (eq) {
        copy(fa.begin(), fa.begin() + n, fb.begin());
    } else {
        for (int i = 0; i < n; i++) {
            fb[i] = i < lb ? Comp(b[i] >> 15, b[i] & M) : Comp(0, 0);
        }
        fft(fb, n);
    }
    Comp r(0.5 / n, 0);
    for (int i = 0, j = 0; i <= (n >> 1); i++, j = n - i) {
        Comp x = (fa[i] + conj(fa[j])) * fb[i] * r; // (a, 0)*(c, d)
        Comp y = (fa[i] - conj(fa[j])) * conj(fb[j]) * r; // (0, b)*(c, d)
        if (i != j) {
            Comp _x = (fa[j] + conj(fa[i])) * fb[j] * r;
            Comp _y = (fa[j] - conj(fa[i])) * conj(fb[i]) * r;
            fa[i] = _x, fb[i] = _y;
        }
        fa[j] = x, fb[j] = y;
    }
    fft(fa, n);
    fft(fb, n);
    vector<int> c(need);
    for (int i = 0; i < need; i++) {
        ll x = fa[i].x + 0.5, y = fa[i].y + 0.5;
        ll z = fb[i].x + 0.5, w = fb[i].y + 0.5;
        c[i] = (((x % mod) << 30) + z + (((y + w) % mod) << 15)) % mod;
    }
    return c;
}
vector<int> sqr_mod(const vector<int> &a) {
    return multiply_mod(a, a, 1);
}
}
using FFT::multiply;
using FFT::multiply_mod;
using FFT::sqr_mod;
```

原根

阶

定义

满足同余式 $a^n = 1(mod\ m)$ 的最小整数 n 存在，这个 n 就称为 a 模 m 的阶，记作 $\delta_m(a)$

性质

1. $a^1, a^2, a^3, \dots a^{\delta_m(a)}$, 模 m 两两不同余
2. 若 $a^n \equiv 1 \pmod{m}]$，则$[\delta_m(a) \mid n]$.
   若 $a^p \equiv a^q \pmod{m}$，则有 $p \equiv q \pmod{\delta_m(a)}$.
3. 设 $m \in \mathbf{N}^*$， $a, b \in \mathbf{Z}$， $(a, m) = (b, m) = 1$，则
   $\delta_m(ab) = \delta_m(a)\delta_m(b)$

   的充分必要条件是
   $gcd\,(\delta_m(a), \delta_m(b)) = 1$ (和欧拉函数一样)
4. 设 $k \in \mathbf{N}$， $m \in \mathbf{N}^*$， $a \in \mathbf{Z}$， $(a, m) = 1$，则
   $\delta_m(a^k) = \dfrac{\delta_m(a)}{(\delta_m(a), k)}$

原根

定义

设 $m \in \mathbf{N}^*$， $g \in \mathbf{Z}$. 若 $(g, m) = 1$，且 $\delta_m(g) = \varphi(m)$，则称 g 为模 m 的原根。

即 g 满足
$\delta_m(g) = |\mathbf{Z}_m^*| = \varphi(m)$.

> 特别的，当模数是质数时，原根就是 $m-1$

原根判定定理

设 $m \geqslant 3,(g, m)=1$，则 g 是模 m 的原根的充要条件是，对于 $\varphi(m)$ 的每个素因数 p，都有 $g^{\frac{\varphi(m)}{p}} \not\equiv 1 \pmod{m}$.

原根个数

若一个数 [m] 有原根 g，那么对于任意 $\varphi(m)$ 的因子 d，模 m 的 d 阶元素存在且个数为 $\varphi(d)$。

特别地，m 的原根个数为 $\varphi(\varphi(m))$。

动态规划

01背包

```cpp
#include<iostream>
#include<vector>
#include<queue>
#include<cstring>
using namespace std;

const int N = 1e4+10;

int dp[N];
int v[N],w[N];
int n,m;


int main(){
    cin>>n>>m;
    for(int i=1;i <= n ; i ++){
        cin>>v[i]>>w[i];
    }
    for(int i=1 ;i<=n;i++){
        for(int j = m ;j >= v[i] ; j --){
            dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
        }
    }
    cout<<dp[m];
    return 0;
}
```

完全背包

```cpp
#include<iostream>
#include<cstring>

using namespace std;
const int N =1e3+10;
int n, m;
int dp[N],v[N],w[N];
int main(){
    cin>>n>>m;
    for(int i =1; i<= n; i++) cin>>v[i]>>w[i];

    for(int i = 1; i<= n ;i++){
        for(int j = v[i] ; j <= m ;  j ++){
            dp[j] = max(dp[j],dp[j-v[i]]+w[i]);//两者都处于第i层所以无需颠倒
        }
    }
    cout<<dp[m];

    return 0;
}
```

分组背包

```cpp
#include<iostream>
#include<cstring>

using namespace std;

const int N = 1e3+10;
int v[N][N],w[N][N];
```

```cpp
int dp[N],s[N];
int n,m;

int main(){
    cin>>n>>m;
    for(int i = 1 ; i <= n ;i++ ){
        cin>>s[i];
        for(int j = 1 ;j<=s[i]; j++ ){
            cin>>v[i][j]>>w[i][j];
        }
    }
    for(int i =1; i<=n;i++){//分析每一组
        for(int j = m ; j >= 0 ; j--){
            for(int k =1 ;k <=s[i]; k++){//因为每组最多选一个，所以要后枚举物品，把空间限制作为每一组中每一个物品单独分析的前提
                if(v[i][k] <= j) dp[j] = max(dp[j],dp[j-v[i][k]]+w[i][k]);
            }
        }
    }
    cout<<dp[m];


    return 0;
}
```

多重背包

二进制优化版本

```cpp
/*
二进制优化法，将每种物品的数量按照二进制进行划分，
例如 10 可以划分成 1 2 4 3(10-7),因为这样的组合在01
背包(每种物品只选择一次)的选择条件下就可以做到将该物品的所有不同数量的
选择情况对划分后的新物品类进行组合得到
*/


#include<iostream>
#include<cstring>

using namespace std;
const int N =1e6 + 10;

int n,m;
int dp[N];
int v1 ,w1 ,s1;
int v[N],w[N],cnt;
int main(){
    cin>>n>>m;

    for(int i =1 ;i<= n;i++){
        cin>>v1>>w1>>s1;
        int k = 1;
        while(s1>=k){
            cnt++;
            v[cnt] = k * v1;
            w[cnt] = k * w1;
            s1 -= k;
            k *= 2;

        }
        if(s1>0){
            cnt ++;
            v[cnt] = s1 * v1;
            w[cnt] = s1 * w1;
        }

    }

    n = cnt ;

    for( int i=1;i <= n ; i ++){
        for(int j = m ; j >= v[i] ; j-- ){
            dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
        }
    }
    cout<<dp[m];
    return 0;

}

```

单调队列优化版本

题目大意

题目链接

有 N 种物品和一个容量是 V 的背包。第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i，求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。输出最大价值。

从朴素思考

我个人觉得先从朴素来推出我们的单调队列优化比较容易。

在朴素版本当中，我们要枚举每一种物品选择多少个，然后枚举每一种可能的状态。

```
for(int i = 1; i <= n ;i++){
    for(int j= 0; j<=m ;j++){
        for(int k =0 ; k <= s[i] && k * v[i] <= j ; k ++ ){
            dp[i][j] = max(dp[i][j],dp[i-1][j-k*v[i]]+k*w[i]);
        }
    }
}
```

根据递推式推导

我们对单独一种物品分析，$dp[i][j]$ 可以从 $dp[i-1][j-k*v]$ 转移而来,所以我们可以得到下式。

$$\begin{cases} F[i][j-0*v] = max(F[i-1][j-0*v]+0*w, & F[i-1][j-1*v]+1*w, & F[i-1][j-2*v]+2*w, & \dots, & F[i-1][j-(s-1)*v \\ F[i][j-1*v] = & max(F[i-1][j-1*v]+0*w, & F[i-1][j-2*v]+1*w, & F[i-1][j-3*v]+2*w, & \\ F[i][j-2*v] = & & max(F[i-1][j-2*v]+0*w, & F[i-1][j-3*v]+1*w, & \\ \vdots \end{cases}$$

我们由这个递推式发现这个东西和滑动窗口非常相似啊，我们要算的其实就是以 s 为大小的滑动窗口中的最大值,并且可以看出最终体积$\%v[i]$ 相等的是一类情况，因为要算$f[j]$一定要有$f[j-v[i]],f[j-2*v[i]]\dots f[r](r=j\%v[i])$，也就是说必须先把这一类一起算完才行，这样我们每一类用一次单调队列就可以了，否则我们就要同时维护多个单调队列。

还有我们可以发现我们的数转移过来的时候是不能直接使用的，我们还需要加上 $k*w(k\in 1,2,3,\dots,s)$，这个 k 值就是我们的$\frac{体积之差}{v[i]}$（因为我们把体积当做了下标，所以直接取单调队列队头和当前的j相减即可），这通过上面的递推式就可以看出。

还有就是这道题必须要有两维，否则就要使用滚动数组，因为我们根据递推式必须要正着遍历，但我们如果正着遍历，当我们要用到前面的数时，其已经更新了，变着你成了这一层，但我们想要的是上一层。

代码

```
int v[N], w[N], s[N] ;
int q[N] ;
int f[2][N], g[N] ;
void solve(){
    cin >> n >> m;
    _rep(i,1,n){
        cin >> v[i] >> w[i] >> s[i] ;
    }
    _rep(i, 1, n){
        _rep(r, 0, v[i] - 1){//分类
            int hh = 0 , tt = -1 ;
            for(int j = r ; j <= m ; j += v[i]){
                if(hh <= tt && (j - q[hh]) / v[i]  >  s[i]) hh ++ ;
                while(hh <= tt && f[(i&1)^1][q[tt]] + (j - q[tt]) / v[i] * w[i] <= g[(i&1)^1][j] ){//新的元素更优
                    tt -- ;
                }
                q[++tt] = j ;
                f[i&1][j] = f[(i&1)^1][q[hh]] + (j - q[hh]) / v[i] * w[i] ;
            }
        }
    }
    cout << f[n&1][m] << endl ;
}
```

有依赖的背包问题

题目链接

有 N 个物品和一个容量是 V 的背包。物品之间存在依赖关系，并且关系成一颗树形，如果选择一个物品，呢么必须选择其父节点。每件物品的编号是 i，体积是 v_i，价值是 w_i，依赖的父节点编号是 p_i。

我们设 $d[i][j]$ 表示的是选择以 i 节点为根节点并且体积不超过 j 的最大价值。很明显我们要从下往上来算，因为我们算父节点 p 的 dp 数组时会用到子节点的 dp 数组。

又因为我们我们的子节点依赖父节点，所以要提前预留父节点的体积，然后我们枚举这一个子节点占用了多少体积。

这里其实和分组背包一样，父节点的每一个子节点是一个组，每个组中有占用不同体积拥有着不同价值的“**物品**”（实际上就是我们算的这些子节点不同体积下所能有的最大价值，也就是我们的 dp 数组）。

最后我们使用给父节点预留出来的空间，把父节点放进去，并且把没放入父亲节点 x 的状态清 0，也就是 $j < v[x]$ 的状态要清 0，那些数据只是用来算 `dp[x][j] = dp[x][j - v[x]] + w[x]` 的，在这之后就没有用了。

```
void dfs(int x){
    for(auto e : g[x]){
        dfs(e) ;
        for(int j = m - v[x] ; j >= 0 ; -- j){//能分配给子节点的体积要保证x节点一定能选
            for(int k = 0 ; k <= j ; ++ k){//枚举这个节点用了多少体积
                dp[x][j] = max(dp[x][j], dp[x][j-k] + dp[e][k]) ;
            }
        }
    }
    for(int j = m ; j >= v[x] ; -- j){//最后加上x节点，倒序遍历不提前更新要用的状态。
        dp[x][j] = dp[x][j - v[x]] + w[x] ;
    }
    for(int j = 0 ; j < v[x] ; ++ j){//清空没选上的状态
        dp[x][j] = 0 ;
    }
}
```

数位 dp

题目链接

```
int f[M][M] ;
int a[M] ;
int dp(int pos,int st,bool op){//op表示是否紧贴上界。st表示当前已经有的1的数目
    if(!pos) return st == k ;
    if(!op && ~f[pos][st]) return f[pos][st] ;//记忆化搜索
    int maxx = op ? min(1LL,a[pos]) : 1 ;//根据是否紧贴上界来决定数据范围
    int res = 0 ;
    for(int i = 0 ;i <= maxx ; ++ i){//mie'jv范围内尽可能的数
        res += dp(pos-1, st + i, op && i == a[pos]) ;
    }
    return op ? res : f[pos][st] = res ;
}
int clac(int x){
    int cnt = 0 ;
    while(x) a[++cnt] = x % b , x /= b ;
    return dp(cnt,0,1) ;
}
void solve(){
    memset(f, -1 ,sizeof f) ;
    cin >> l >> r ;
    cin >> k >> b ;
    ans = clac(r) - clac(l-1) ;
    cout << ans << endl ;

}
```

最长上升/下降子序列

求**单调不增**子序列，后一项总是**小于等于**前一项
二分找已有序列中**小于**当前数的位置， 存在就替换，否则说明当前数比所有数都**小于等于**，添加放到末尾

求**单调增加**子序列，后一项总是**大于**前一项

二分找已有序列中**大于等于**当前数的位置， 存在就替换，否则说明当前数比所有数都**大于**，添加放到末尾

求**单调不下降**子序列，后一项总是**大于等于**前一项
二分找已有序列中**大于**当前数的位置， 存在就替换，否则说明当前数比所有数都**大于等于**，添加放到末尾

求**单调下降**子序列，后一项总是**小于**前一项
二分找已有序列中**小于**当前数的位置， 存在就替换，否则说明当前数比所有数都**小于等于**，添加放到末尾

```
#include<bits/stdc++.h>
#define IOS ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define endl '\n'
#define int long long
#define PII pair<int, int>
using namespace std;
const int N = 1e6 + 10 ;
const int INF = 0x3f3f3f3f3f3f3f3f;
const int MOD = 998244353 ;//全一才位1
```

```cpp
const double eps = 1e-9 ;
int gcd(int a, int b){ return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b){ return a * b / gcd(a,b); }
int qmi(int a, int b){
    int res = 1 ;
    while(b){
        if(b&1) res = a * res % MOD ;
        b >>= 1 ;
        a = a * a % MOD ;
    }
    string s ;
    return res ;
}
int T = 1 ;
int ans = 0 ;
int n , m, k ;
int f1[N], f2[N] ;
void solve(){
    vector<int> a ;
    int t ;
    while(cin >> t){
        a.push_back(t) ;
    }
    int num = a.size() ;
    int cnt1 = 0 ;
    int cnt2 = 0 ;
    f1[++cnt1] = a[0] ;
    f2[++cnt2] = a[0] ;
    for(int i = 1 ; i < a.size() ; ++ i){
        if(a[i] <= f1[cnt1]) f1[++cnt1] = a[i] ;
        else{
            f1[(int)(upper_bound(f1+1,f1+cnt1+1 , a[i], greater<int>())-f1)] = a[i] ;
        }
        if(a[i] > f2[cnt2]) f2[++cnt2] = a[i] ;
        else{
            f2[(int)(lower_bound(f2+1,f2+cnt2+1 , a[i])-f2)] = a[i] ;
        }
    }
    cout << cnt1 << endl ;
    cout << cnt2 << endl ;

}
signed main(){
    IOS ;
    // cin >> T ;
    while(T --){
        solve() ;
    }
    return 0;

}
```

博弈论

大部分的公平组合游戏都可以转化为有向图游戏。

有向无环图棋子游戏

在一个有向无环图中没只有一个起点，上面有一个棋子，两个玩家按照有向图轮流来推棋子，直到不能继续走。

---

mex运算

$mex(S)$为不属于S集合中的最小非负整数，

$$mex(S) = min\{x\}(x \in N, x \notin S)$$

例如，$mex(\{0,1,2\}) = 3, mex(\{1,2\}) = 0$。

SG函数

设状态节点x有k个后继状态节点$y_1, y_2, \ldots, y_k$，

$$SG(x) = mex(\{SG(y_1) \oplus SG(y_2) \oplus SG(y_3) \oplus \ldots \oplus SG(y_k)\})$$

若果说当前状态 SG值 不是 0，呢么其子状态中一定有一个状态的 SG值 是 0，也就是说处于当前状态的人必胜，因为其一定能导向其对手的必败态（SG值 是 0）。

若果说当前状态 SG值 是 0，呢么其子状态中一定没有有一个状态的 SG值 是 0，也就是说处于当前状态的人必败，因为其一定会导向其对手的必胜态（SG值 不是 0）。

SG定理

由n个有向图游戏组合的游戏，设起点分别为$s_1, s_2, s_3, \ldots, s_n$,当$SG(s_1) \oplus SG(s_2) \oplus SG(s_3) \oplus \ldots \oplus SG(s_n) \neq 0$时，先手必胜，反之必败。

```
vector<int> vec[N] ;
int f[N] ;
int sg(int x){
    if(f[x] != -1) return f[x] ;
    set<int> se ;
    for(auto e : vec[x]){//统计子结点中的sg值
        se.insert(sg(e)) ;
    }
    for(int i = 0 ; ; ++ i){//计算当前的SG函数值
        if(!se.count(i)) return f[x] = i ;
    }
}
void solve(){
    cin >> n >> m >> k;
    int u, v, w ;
    for(int i = 1; i <= m ; ++ i){
        cin >> u >> v ;
        vec[u].push_back(v) ;
    }
    memset(f, -1, sizeof f) ;
    int x ;
    for(int i = 1; i <= k ; ++ i){
        cin >> x ;
        ans ^= sg(x) ;
    }
    if(ans){
        cout << "win" << endl ;
    }else{
        cout << "lose" << endl ;
    }
}
```

集合型 Nim 游戏

给定两个 m 个整数构成的集合 a_i ，给定 n 对石子的数量分别是 b_i 。两个玩家轮流操作，每次操作可以从任意一堆石子中拿去石子，但是拿去石子的数量一定是集合 a_i 中的一个数，最后无法进行操作就的人失败。

```
int sg(int x){
    if(f[x] != -1) return f[x] ;
    set<int> se ;
    for(int i = 1 ; i <= m ; ++ i){
        if(x >= a[i]) se.insert(sg(x-a[i])) ;
        else break ;
    }
    for(int i = 0 ; ; ++ i){
        if(!se.count(i)) return f[x] = i ;
    }
}
void solve(){
    cin >> m ;
    for(int i = 1; i <= m ; ++ i) cin >> a[i] ;
    sort(a + 1, a + 1 + m) ;
    memset(f, -1, sizeof f) ;
    int x ;
    cin >> n ;
    for(int i = 1; i <= n ; ++ i){
        cin >> x ;
        ans ^= sg(x) ;
    }
    if(ans){
        cout << "Yes" << endl ;
    }else{
        cout << "No" << endl ;
    }
}
```

剪纸游戏

给定一张 $N \times M$

的矩形网格纸，两名玩家轮流行动。

在每一次行动中，可以任选一张矩形网格纸，沿着某一行或某一列的格线，把它剪成两部分。

首先剪出 1×1 的格纸的玩家获胜。

两名玩家都采取最优策略行动，求先手是否能获胜。

提示：开始时只有一张纸可以进行裁剪，随着游戏进行，纸张被裁剪成 $2, 3, \ldots$

更多张，可选择进行裁剪的纸张就会越来越多。

```cpp
int f[M][M] ;
int sg(int x,int y){
    if(f[x][y] != -1) return f[x][y] ;
    set<int> se ;
    for(int i = 2 ; i <= x - 2 ; ++ i){
        se.insert(sg(x-i,y) ^ sg(i,y)) ;
    }
    for(int i = 2 ; i <= y - 2 ; ++ i){
        se.insert(sg(x,y-i)^ sg(x,i)) ;
    }
    for(int i = 0 ; ; ++ i){
        if(!se.count(i)) return f[x][y] = i ;
    }
}
void solve(){
    ans = 0 ;
    ans = sg(n,m) ;
    if(ans){
        cout << "WIN" << endl ;
    }else{
        cout << "LOSE" << endl ;
    }
}
signed main(){
    IOS ;
    memset(f, -1, sizeof f) ;
    while(cin >> n >> m){
        solve() ;
    }
    return 0;

}
```

Nim游戏

基础模型

> n堆物品，每堆有$a[i]$个，两个玩家轮流取走任意一堆的任意数量的物品，但不能不取。取走最后一个物品的人获胜，或者说谁最后没得选就输了。

结论

如果每一堆石子异或和不为0的话，那么先手必赢，否则后手必赢。

证明

玩家可以做的操作是对这n堆中的一堆中的石子进行选择，使得这堆石子的数量减少，这种减少的操作实际上我们也可以看作一个数x堆这堆石子的数量$a[i]$进行异或。

游戏中也只会出现两种局面:

- **情况1**：$a_1 \oplus a_2 \oplus a_3 \ldots \oplus a_n \neq 0$，一定存在一个数（也就是这些数异或和，假设该数为$k$）使得异或和为$0$，并且能够保证一定存在一个$a[i]$符合$a[i] \oplus k < a[i]$，因为二进制下的$k$的最高位一定是$1$，这说明一定有一个$a[i]$的二进制表示下这一个位置也是$1$,那么让这一个$a[i]$与k异或一定能使得$a[i] \oplus k < a[i]$成立。
- **情况2**：$a_1 \oplus a_2 \oplus a_3 \ldots \oplus a_n = 0$，因为每次不能不选，所以无论如何操作一定会破坏异或和为0的情况

综上，可以看出如果一个人能够始终处于**情况2**下，那么这个人必输，先来看**情况1**，通过一次异或我们发现**情况1**就可以转化为**情况2**，由此看出：

- **情况1**：先手必赢，后手必输。
- **情况2**：先手必输，后手必赢。

计算几何

扫描线

1. 扫描线是一根先，从下往上扫，以每一个矩形的上下边为边界，可以把 n 个矩形组成的区域分割成 $2n+1$ 个区块。
2. 区块的高度就是扫过的距离即 $Y_{i+1}-Y_i$ 。
3. 区块长度就是区块内矩形长度的并，即 len_i。
4. 面积并 $\sum\{len_i \times (Y_{i+1}-Y_i)\}$
5. 区块长度可以用线段树维护。每当扫到一个矩形的下边时，就加入该矩形长度的贡献；当扫到这个矩形的上边时，就减去这个矩形长度的贡献。
6. 区块长度需要通过过 矩形的 X 坐标来拼凑。呢么我们就需要离散化，把稀疏的数映射为连续的整数。

矩形并面积

```cpp
#include<bits/stdc++.h>
#define IOS ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define endl '\n'
#define int long long
#define lc(u) u << 1LL
#define rc(u) u << 1LL | 1
#define PII pair<int, int>
#define _rep(i,a,b) for( int i=(a); i<=(b); ++i)
#define _per(i,a,b) for( int i=(a); i>=(b); --i)
using namespace std;
const int N = 1e6+10 ;
const int M = 1e3+10 ;
const int INF = 0x3f3f3f3f3f3f3f3f;
const int MOD = 1e9 + 7  ;
const double eps = 1e-9 ;
int T = 1 ;
int ans ;
int n, a, b;
int X[N*8] ;
int cnt = 0 ;
struct line//扫描线
{
    int x1, x2, y  ;//分别表示线的两端和高度
    int tag ;//入线 + 1,  出线 -1
    bool operator < (line &t){
        return y < t.y ;
    }
}L[N*8];
struct node
{
    int l, r ;
    int cnt ;//表示是否存在重复
    int len ;//表示区间矩阵长度
}tr[N*8];
void build(int l, int r,int u ){
    tr[u] = {l, r, 0, 0} ;
    if(l == r) return ;
    int mid = l + r >> 1 ;
    build(l, mid, lc(u)) ;
    build(mid + 1, r,rc(u)) ;
}
void pushup(int u){
    int l = tr[u].l ,r = tr[u].r ;
    if(tr[u].cnt) tr[u].len = X[r + 1] - X[l] ;//如果说存在重合部分，说明一定覆盖，为什么要 r + 1是因为我们可能存在两个区间他们并不连续，中间会出现空白的部分，为了保证他们连续，
    else  tr[u].len = tr[lc(u)].len + tr[rc(u)].len ;//不一定存在重合部分，所以说不一定覆盖，  也就不能直接用坐标求，
}
void modify(int u, int l, int r,int tag){//传参r向左偏移一位
    if(l > tr[u].r || r < tr[u].l ) return ;
    if(l <=tr[u].l &&  r >=tr[u].r){
        tr[u].cnt += tag ;
        pushup(u) ;
        return ;
    }
    modify(lc(u), l, r, tag) ;
    modify(rc(u), l, r, tag) ;
    pushup(u) ;
}

void solve(){
    cin >> n ;
    int x1, y1 , x2, y2 ;
    for(int i = 1 ; i <= n ; ++ i){
        cin >> x1 >> y1 >> x2 >> y2 ;
        L[i] = {x1,x2,y1,1} ;
        L[i + n] = {x1,x2,y2,-1} ;
```

```
        X[i] = x1 , X[i + n] = x2 ;
    }
    sort(L + 1, L + 1 + n * 2) ;
    sort(X + 1, X + 1 + n * 2) ;
    int s = unique(X + 1, X + n * 2 + 1) - X - 1 ;//去重
    build(1, s - 1, 1) ;//为了保证我们合并区域的 r + 1 操作的正确性
    for(int i = 1 ; i < n * 2 ; ++ i){
        int l = lower_bound(X + 1 , X + s  + 1, L[i].x1) - X ;//寻找对应值映射的下标
        int r = lower_bound(X + 1 , X + s + 1 , L[i].x2) - X ;
        // cout << X[l] << " " << X[r] << endl ;
        modify(1, l, r - 1, L[i].tag) ;//为了保证我们合并区域的 r + 1 操作的正确性
        ans +=  tr[1].len * (L[i + 1].y - L[i].y) ;
        // cout << tr[1].len << endl ;
    }
    cout << ans << endl ;
}
signed main()
{
    //  cin >>  T ;
     while( T -- ){
         solve() ;
    }


}
```

矩形周长并

```
#include<bits/stdc++.h>
#define IOS ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define endl '\n'
#define int long long
#define lc(u) u << 1LL
#define rc(u) u << 1LL | 1
#define PII pair<int, int>
#define _rep(i,a,b) for( int i=(a); i<=(b); ++i)
#define _per(i,a,b) for( int i=(a); i>=(b); --i)
using namespace std;
const int N = 1e6+10 ;
const int M = 1e3+10 ;
const int INF = 0x3f3f3f3f3f3f3f3f;
const int MOD = 1e9 + 7  ;
const double eps = 1e-9 ;
int T = 1 ;
int ans ;
int n, a, b;
int X[N*8] ;
int cnt = 0 ;
struct line//扫描线
{
    int x1, x2, y  ;//分别表示线的两端和高度
    int tag ;//入线 + 1,  出线 -1
    bool operator < (line &t){//双关键字排序
        if(y == t.y) return tag > t.tag ;
        return y < t.y ;
    }
}L[N*8];
struct node
{
    int l, r ;
    int cnt ;//表示是否存在重复
    int len ;//表示区间矩阵长度
    int sum ;
    bool lcover, rcover ;
}tr[N*8];
void build(int l, int r,int u ){
    tr[u] = {l, r} ;
    if(l == r) return ;
    int mid = l + r >> 1 ;
    build(l, mid, lc(u)) ;
    build(mid + 1, r,rc(u)) ;
}
void pushup(int u){
    int l = tr[u].l ,r = tr[u].r ;
    if(tr[u].cnt){
        tr[u].len = X[r + 1] - X[l] ;
        tr[u].sum = 2 ;
        tr[u].lcover = tr[u].rcover = true ;
    }
    else{
        tr[u].len = tr[lc(u)].len + tr[rc(u)].len ;
        tr[u].sum = tr[lc(u)].sum + tr[rc(u)].sum ;
```

```
        tr[u].lcover = tr[lc(u)].lcover ;
        tr[u].rcover = tr[rc(u)].rcover ;
        if(tr[lc(u)].rcover && tr[rc(u)].lcover){
            tr[u].sum -= 2 ;
        }
    }
}
void modify(int u, int l, int r,int tag){//传参r向左偏移一位
    if(l > tr[u].r || r < tr[u].l ) return ;
    if(l <=tr[u].l &&  r >=tr[u].r){
        tr[u].cnt += tag ;
        pushup(u) ;
        return ;
    }
    modify(lc(u), l, r, tag) ;
    modify(rc(u), l, r, tag) ;
    pushup(u) ;
}

void solve(){
    cin >> n ;
    int x1, y1 , x2, y2 ;
    for(int i = 1 ; i <= n ; ++ i){
        cin >> x1 >> y1 >> x2 >> y2 ;
        L[i] = {x1,x2,y1,1} ;
        L[i + n] = {x1,x2,y2,-1} ;
        X[i] = x1 , X[i + n] = x2 ;
    }
    sort(L + 1, L + 1 + n * 2) ;
    sort(X + 1, X + 1 + n * 2) ;
    int s = unique(X + 1, X + n * 2 + 1) - X - 1 ;//去重
    build(1, s - 1, 1) ;//为了保证我们合并区域的 r + 1 操作的正确性
    int la = 0 ;
    for(int i = 1 ; i < n * 2 ; ++ i){
        int l = lower_bound(X + 1 , X + s + 1, L[i].x1) - X ;//寻找对应值映射的下标
        int r = lower_bound(X + 1 , X + s + 1 , L[i].x2) - X ;
        // cout << X[l] << " " << X[r] << endl ;
        modify(1, l, r - 1, L[i].tag) ;//为了保证我们合并区域的 r + 1 操作的正确性
        ans += abs(la - tr[1].len) ;
        la = tr[1].len ;
        ans +=  tr[1].sum * (L[i + 1].y - L[i].y) ;
        // cout << tr[1].len << endl ;
    }
    ans += L[2*n].x2 - L[2*n].x1 ;
    cout << ans << endl ;
}
signed main()
{
    //  cin >>  T ;
     while( T -- ){
         solve() ;
    }


}
```

凸包

Andrew 算法

```
struct node{
    double x, y ;
}p[N], s[N];
int n, top ;
double cross(node a, node b , node c){//负值说明是钝角
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x) ;
}
double dis(node a, node b ){
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)) ;
}
bool cmp(node a, node b){
    if(a.x != b.x) return a.x < b.x ;
    return a.y < b.y ;
}
double Andrew(){
    sort(p + 1, p + 1 + n, cmp) ;
    for(int i = 1; i <= n ; ++ i){//下凸包
        while(top > 1 && cross(s[top-1], s[top],p[i]) <= 0.0) top -- ;
        s[++top] = p[i] ;
    }
    int t = top ;
    for(int i = n - 1 ; i >= 1 ; -- i){//上凸包
```

```
        while(top > t && cross(s[top-1], s[top],p[i]) <= 0.0) top -- ;
        s[++top] = p[i] ;
    }
    double res = 0 ;

    for(int i = 1 ; i < top ; ++ i){
        res += dis(s[i], s[i + 1]) ;
    }
    return res ;
}
```

旋转卡壳

```
#include<bits/stdc++.h>
#define IOS ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define endl '\n'
#define int long long
#define double long double
#define PII pair<int, int>
#define _rep(i,a,b) for( int i=(a); i<=(b); ++i)
#define _per(i,a,b) for( int i=(a); i>=(b); --i)
using namespace std;
const int N = 1e6+10 ;
const int M = 1e3+10 ;
const int INF = 0x3f3f3f3f3f3f3f3f;
const int MOD = 1e9 + 7 ;
const double eps = 1e-9 ;
int gcd(int a, int b){ return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b){ return a * b / gcd(a,b); }
int qmi(int a, int b){
    int res = 1 ;
    while(b){
        if(b&1) res = a * res % MOD ;
        b >>= 1 ;
        a = a * a % MOD ;
    }
    return res ;
}
int T = 1 ;
struct node{
    double x, y ;
}p[N], s[N];
int n, top ;
int ans = 0 ;
int cross(node a, node b , node c){//负值说明是钝角
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x) ;
}
int dis(node a, node b ){
    return ((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)) ;
}
bool cmp(node a, node b){
    if(a.x != b.x) return a.x < b.x ;
    return a.y < b.y ;
}
double Andrew(){
    sort(p + 1, p + 1 + n, cmp) ;
    for(int i = 1; i <= n ; ++ i){//下凸包
        while(top > 1 && cross(s[top-1], s[top],p[i]) <= 0.0) top -- ;
        s[++top] = p[i] ;
    }
    int t = top ;
    for(int i = n - 1 ; i >= 1 ; -- i){//上凸包
        while(top > t && cross(s[top-1], s[top],p[i]) <= 0.0) top -- ;
        s[++top] = p[i] ;
    }
    double res = 0 ;

    for(int i = 1 ; i < top ; ++ i){
        res += dis(s[i], s[i + 1]) ;
    }
    return res ;
}
int roatint_calipers(){
    int res = 0 ;
    for(int i = 1 , j = 2; i <= n ; ++ i){
        while(cross(s[i],s[i%n+1], s[j]) < cross(s[i],s[i%n+1], s[j % n + 1])){
            j = j % n + 1 ;
        }
        res = max(res, max(dis(s[i], s[j]), dis(s[i + 1], s[j]))) ;
    }
    return res ;
```

```
}
void solve(){
    cin >> n ;
    for(int i = 1; i <= n ; ++ i){
        cin >> p[i].x >> p[i].y ;
    }
    Andrew() ;
    n = top - 1 ;
    ans = roatint_calipers() ;
    cout << ans << endl ;
}
signed main(){
    IOS ;
    while(T --){
        solve() ;
    }
    return 0;

}
```

多边形面积公式（鞋带定理）

$$S = \frac{1}{2}|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)$$

$x_{n+1} = x_1$，$y_{n+1} = y_1$

无需对点排序，答案取绝对值即可

```
inline double Ploygon_area(Polygon s)
{
    int n = s.size();
    double ans = 0.0;
    for (int i = 0; i < n; i++)
    {
        ans += cross(s[i], s[(i + 1) % n]);
    }
    return ans * 0.5;
}
```

Pick定理 （网格图中多边形面积）

计算顶点都在整数格点的封闭图形的面积。

定义

- 格点：数学上把在平面直角坐标系中横纵坐标均为整数的点称为格点或整点。
- 格点多边形：坐标平面内顶点为格点的多边形称作格点多边形。

$$S = a + \frac{b}{2} - 1$$

a 表示图形内部的格点的数目， b 表示在图形边界上的格点的数目，S 表示图形面积。

辅助

在直角坐标系中，一个机器人从任意点出发进行 n 次移动，每次向右移动 dx，向上移动 dy，最后会形成一个平面上的封闭简单多边形，求边上的点的数量，多边形内的点的数量，多边形面积。

以整点为顶点的线段，如果 dx 和 dy 都不为 0，经过的格点数就是 $gcd(dx, dy) + 1$，如果要计算封闭图形中的边上的格点数就可以不用 $+1$，因为会被下一条相连的边加上。

计算面积使用鞋带定理，然后用 $Pick$ 定理就可以计算出多边形内部的格点数

坐标系

```
const int CC = 1;//逆时针方向
const int C = -1;//顺时针方向
const int OB = 2;//方向相反
const int OF = -2;//方向相同 |b|>|a|
const int OS = 0;//方向相同 |a|>|b|

```

```cpp
struct Point{//同时为点和向量
    double x,  y ;
    Point() {}
    Point(double x, double y) :x(x), y(y) {}
    Point operator - (const point & b) const{return Point{x - b.x,y -b.y};}//向量的减法
    Point operator + (Point& b){return Point(x + b.x, y + b.y);}//向量的加法
    Point operator * (double k){return Point(x * k, y * k);}//向量的乘法
    Point operator / (double k){return Point(x / k, y / k);}//向量的乘法

};
typedef Point Vector;//向量和点共享结构体
double norm(Vector a){return a.x * a.x + a.y * a.y;}//范数
double absNorm(Vector a){return sqrt(norm(a));}//模长
double dot(Vector a, Vector b){return a.x * b.x + a.y * b.y;} //内积，为负时夹角为钝角
double cross(Vector a, Vector b){return a.x * b.y - a.y * b.x;} //外积，同时也可以表示为两向量夹角三角形面积的二倍，当外积为正时说明b向量的点在a向量的左侧，为负数时在右侧，为 0 时说明平行

struct Line{//线段
    Point s;    // 起点
    Point e;    // 终点
    bool is_seg; // 是否是线段
    Line() {};    // 默认构造函数
    Line(Point a, Point b, bool _is_seg = true) { s = a; e = b; is_seg = _is_seg; }    // 构造函数(默认是线段)
};
Point project(Point p, Line l){//p在l上的投影点
    Vector base = l.s - l.e ;
    double r = dot(p - l.s,base) / norm(base) ;//投影长度
    base = base * r ;
    return Point(l.s + base) ;

}
Point reflect(Point p , Line l){//p关于l的对称点,构造三角形
    Point p1 = (project(p,l) - p) * 2 ;
    return Point(p+p1) ;
}
double get_dis(Point p1,Line l){//点与线段的距离
    return abs(cross(l.e - l.s,p1 - l.s)) / absNorm(l.e - l.s) ;
}
inline int ccw(Vector a, Vector b){
    if(cross(a,b) > eps) return CC ;//逆时针
    if(cross(a,b) < -eps) return C ;//顺时针
    if(dot(a,b) < -eps) return OB ; // 方向相反
    if(norm(a) < norm(b)) return OF ; // 方向相同,但 b 更长
    return OS ;
}
inline int ccw(Point p1, Point p2, Point p3){
    Vector a = p2 - p1 ;
    Vector b = p3 - p1 ;
    return ccw(a, b) ;
}
Point getCrossPoint(Line l1, Line l2){//线段相交的点,利用相似三角形
    Vector base = l2.e - l2.s ;
    double d1 = abs(cross(base,l1.s - l2.s)) ;
    double d2 = abs(cross(base,l1.e - l2.e)) ;
    double t = d1 / (d1 + d2) ;
    Point x = (l1.e - l1.s) * t ;
    return l1.s + x ;
}
struct Circle
{
    Point c;
    double r;
    Circle() {}
    Circle(Point c, double r) :c(c), r(r) {}
};
pair<Point,Point> getCrossPoint(Line l, Circle C){
    if(get_dis(C.c,l) < C.r) return make_pair(Point(INF, INF),Point(INF, INF)) ;
    Point pr = project(C.c,l) ;//投影点
    Vector e = (l.e - l.s) / absNorm(l.e - l.s);
    double base = sqrt(C.r * C.r - norm(pr - C.c)) ;
    Vector x = e*base ;
    return make_pair(pr + x, pr - x) ;
}
```

杂

快读快写

```
inline int read()
{
    int x = 0, f = 1;
    char ch = getchar();
    while(ch < '0' || ch > '9'){
        if(ch == '-') f = -1;
        ch = getchar();
    }
    while(ch >= '0' && ch <= '9'){
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}

void write(int x)
{
    if(x < 0) putchar('-'), x = -x ;
    if(x > 9) write(x / 10) ;
    putchar(x % 10 + '0') ;
    return ;
}

```

```
using i128 = __int128;

inline void read(i128 &res) {
    i128 x = 0, f = 1;
    char ch = getchar();
    while(ch < '0' || ch > '9'){
        if(ch == '-') f = -1;
        ch = getchar();
    }
    while(ch >= '0' && ch <= '9'){
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    res = x * f;
}

inline void print(i128 x) {
    if(x < 0) putchar('-'), x = -x ;
    if (x > 9) print(x / 10);
    putchar(x % 10 + '0');
    return ;
}

```

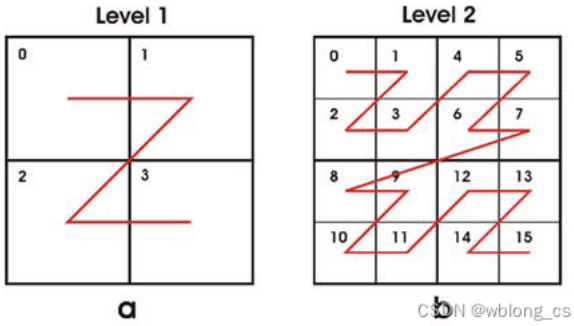
全排列函数

#include
int perm[N];
next_permutation(perm, perm+2);
按照字典序由小到大生成下一个全排列

基本格式：

```
int a[]={1,2,2...};//会展示出一个升序数组的所有全排列
do{

}while(next_permutation(a,a+n));
```

Z顺序曲线



第 k 个数的位置和其坐标位置 (x, y) 的关系是，将 x 和 y 转化成二进制形式交错排列，比如坐标为 $(1, 2)$ (从0开始)的位置，$x = 01_2, y = 10_2$，$k = x_1y_1x_0y_0 = 0110_2 = 6_{10}$