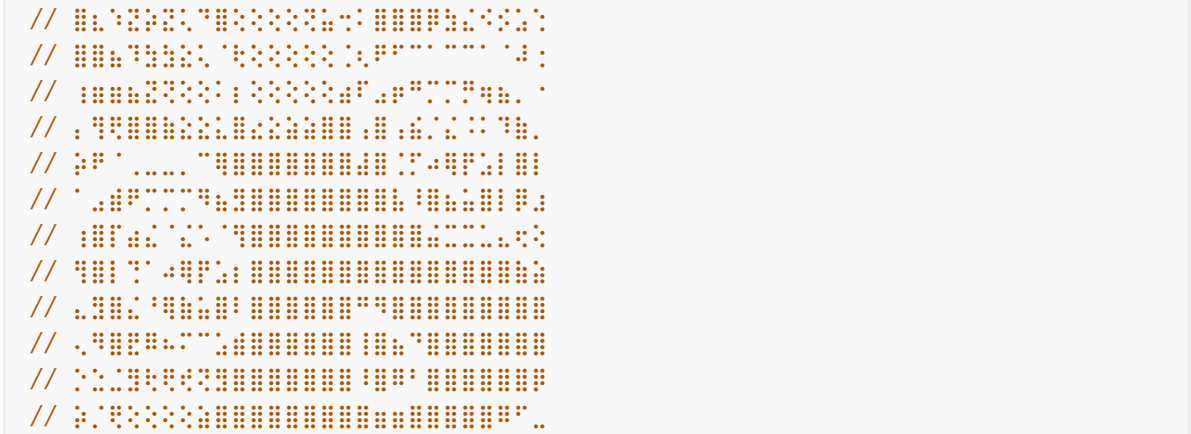
火车头

```
#include <bits/stdc++.h>
#include <unordered_map>
#include <unordered_set>
#define endl '\n'
```

```cpp
#define ll long long
#define lll __int128
#define ull unsigned long long
#define db double
#define pb push_back
#define inf 0x3f3f3f3f
#define fi first
#define se second
#define Please return
#define Psycho -1
#define AC 0
const long double eps = 1e-12;
const double pi = acos(-1);
const int N = 2e5 + 10, M = 1e4 + 10;
const int mod = 998244353;
using namespace std;
void solve() {}
int main()
{
    ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
    cout << fixed << setprecision(12);
    int t;
    cin >> t;
    while (t--) solve();
    Please AC;
}
```



吸氧

```cpp
#pragma GCC optimize(2)
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
#pragma GCC optimize("-fgcse")
#pragma GCC optimize("-fgcse-lm")
#pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
#pragma GCC optimize("-ftree-vrp")
#pragma GCC optimize("-fpeephole2")
#pragma GCC optimize("-ffast-math")
#pragma GCC optimize("-fsched-spec")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("-falign-jumps")
#pragma GCC optimize("-falign-loops")
```

```
#pragma GCC optimize("-falign-labels")
#pragma GCC optimize("-fdevirtualize")
#pragma GCC optimize("-fcaller-saves")
#pragma GCC optimize("-fcrossjumping")
#pragma GCC optimize("-fthread-jumps")
#pragma GCC optimize("-funroll-loops")
#pragma GCC optimize("-fwhole-program")
#pragma GCC optimize("-freorder-blocks")
#pragma GCC optimize("-fschedule-insns")
#pragma GCC optimize("inline-functions")
#pragma GCC optimize("-ftree-tail-merge")
#pragma GCC optimize("-fschedule-insns2")
#pragma GCC optimize("-fstrict-aliasing")
#pragma GCC optimize("-fstrict-overflow")
#pragma GCC optimize("-falign-functions")
#pragma GCC optimize("-fcse-skip-blocks")
#pragma GCC optimize("-fcse-follow-jumps")
#pragma GCC optimize("-fsched-interblock")
#pragma GCC optimize("-fpartial-inlining")
#pragma GCC optimize("no-stack-protector")
#pragma GCC optimize("-freorder-functions")
#pragma GCC optimize("-findirect-inlining")
#pragma GCC optimize("-fhoist-adjacent-loads")
#pragma GCC optimize("-frerun-cse-after-loop")
#pragma GCC optimize("inline-small-functions")
#pragma GCC optimize("-finline-small-functions")
#pragma GCC optimize("-ftree-switch-conversion")
#pragma GCC optimize("-foptimize-sibling-calls")
#pragma GCC optimize("-fexpensive-optimizations")
#pragma GCC optimize("-funsafe-loop-optimizations")
#pragma GCC optimize("inline-functions-called-once")
#pragma GCC optimize("-fdelete-null-pointer-checks")
```

常用函数

max和min

```
max(a, b); min(a, b); max({a, b, c}); min({a, b, c});
```

sort

```
//用法
sort(begin, end);
//自定义排序
bool cmp(int x1,int x2) { return x1 < x2; }
sort(arr, arr + n, cmp);
//Lambda表达式
sort(arr, arr + n, [](int x1, int x2){ return x1 < x2; });
```

二分查找

binary_search

```
bool flag = binary_search(arr + l, arr + r, val); //查[l, r),找到结果为true，没有则为false
//binary_search利用的也是指针
```

lower_bound

```
lower_bound(arr + l, arr + r, val) //找到[l, r)第一个大于等于val的数
//返回地址,如果返回尾地址,则没找到
```

upper_bound

```
upper_bound(arr + l, arr + r, val) //找到[l, r)第一个严格大于val的数
//返回地址,如果返回尾地址,则没找到
```

equal_range

equal_range综合了lower_bound和upper_bound

```
pair<T, T> = equal_range(arr + l, arr + r, val); //返回pair类型
//等价于{lower_bound(arr + l, arr + r, val) ,upper_bound(arr + l, arr + r, val)}
```

copy

```
copy(arr1 + l, arr1 + r, arr2); //把arr1[l, r)的部分复制到arr2
```

copy_n

```
copy_n(arr1 + l, x, arr2); //把arr1[l, l + x)的部分复制到arr2
```

swap

```
swap(a, b); //交换a, b
```

replace

```
replace(arr + l, arr + r, oldv, newv); //把[l, r)区间的oldv全部替换为newv
```

fill

```
fill(arr + l, arr + r, val); //把[l, r)区间赋值为val
```

reverse

```
reverse(arr + l, arr + r); //把[l, r)区间反转
```

rotate

```
rotate(arr + l, arr + mid, arr + r); //滚动数组,arr + mid向前滚动,成为新的[l, r)区间首元素
```

find

```
find(arr + l, arr + r, val); //可用于对无序数组查找
//返回地址,如果返回尾地址,则没找到
```

find_end

```
find_end(arr1 + l, arr1 + r, arr2 + l, arr2 + r); //查询arr1中是否有子数组arr2
//返回地址,如果返回尾地址,则没找到
```

unique

unique的作用是“去掉”容器中相邻元素的重复元素

它实质上是一个伪去除，它会把重复的元素添加到容器末尾，而返回值是去重部分的尾地址

> ⚠ unique针对的是相邻元素，所以对于顺序错乱的数组成员，或者容器成员，需要先进行排序

```
unique(a.begin(), a.end()); //去重，将重复元素放在末尾
```

count

```
auto cnt = count(arr + l, arr + r, val); //统计[l, r)中val的个数
```

equal

```
bool flag = equal(arr1 + l, arr1 + r, arr2); //arr1的[l, r)区间是否与arr2相等
//找到结果为true，没有则为false
```

merge

```
merge(arr1, arr1 + n, arr2, arr2 + m, arr3); //将arr1,arr2有序合并为arr3,归并排序
```

includes

```
includes(arr1 + l, arr1 + r, arr2 + l, arr2 + r); //查询arr1中是否有子序列arr2
//但是两个序列必须都是升序或者降序
```

集合相关

set_union

其作用是求两个集合的并集，但是要求输入的两个集合必须是有序的

```
int len = set_union(arr1, arr1 + n1, arr2, arr2 + n2, arr3) - arr3;
//返回新集合大小
```

set_intersection

其作用是求两个集合的交集，但是要求输入的两个集合必须是有序的

```
int len = set_intersection(arr1, arr1 + n1, arr2, arr2 + n2, arr3) - arr3;
//返回新集合大小
```

set_difference

其作用是求两个集合的差（A - B），但是要求输入的两个集合必须是有序的

```
int len = set_difference(arr1, arr1 + n1, arr2, arr2 + n2, arr3) - arr3;
//返回新集合大小
```

set_symmetric_difference

其作用是求两个集合的对称差集 $A \triangle B = (A - B) \cup (B - A)$，但是要求输入的两个集合必须是有序的

```
int len = set_symmetric_difference(arr1, arr1 + n1, arr2, arr2 + n2, arr3) -
arr3;
//返回新集合大小
```

字典序函数

next_permutation

```
next_permutation(arr ,arr + n); //得到下个一个字典序
```

prev_permutation

```
prev_permutation(arr ,arr + n); //得到上个一个字典序
```

STL

use

STL容器使用时机

.	vector	deque	list	set	multiset	map	multimap
典型内存结构	单端数组	双端数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对key而言：不是	否
元素搜寻速度	慢	慢	非常慢	快	快	对key而言：快	对key而言：快
元素安插移除	尾端	头尾两端	任何位置	-	-	-	-

vector的使用场景：比如软件历史操作记录的存储，我们经常要查看历史记录，比如上一次的记录，上上次的记录，但却不会去删除记录，因为记录是事实的描述。

deque的使用场景：比如排队购票系统，对排队者的存储可以采用deque，支持头端的快速移除，尾端的快速添加。如果采用vector，则头端移除时，会移动大量的数据，速度慢。

vector与deque的比较：
一：vector.at()比deque.at()效率高，比如vector.at(0)是固定的，deque的开始位置 却是不固定的。
二：如果有大量释放操作的话，vector花的时间更少，这跟二者的内部实现有关。
三：deque支持头部的快速插入与快速移除，这是deque的优点。

list的使用场景：比如公交车乘客的存储，随时可能有乘客下车，支持频繁的不确实位置元素的移除插入。

set的使用场景：比如对手机游戏的个人得分记录的存储，存储要求从高分到低分的顺序排列。

map的使用场景：比如按ID号存储十万个用户，想要快速要通过ID查找对应的用户。二叉树的查找效率，这时就体现出来了。如果是vector容器，最坏的情况下可能要遍历完整个容器才能找到该用户。

iterator

迭代器	功能	描述
输入迭代器	提供对数据的只读访问	只读，支持++、==、！=
输出迭代器	提供对数据的只写访问	只写，支持++
前向迭代器	提供读写操作，并能向前推进迭代器	读写，支持++、==、！=
双向迭代器	提供读写操作，并能向前和向后操作	读写，支持++、–，
随机访问迭代器	提供读写操作，并能以跳跃的方式访问容器的任意数据，是功能最强的迭代器	读写，支持++、–、[n]、-n、<、<=、>、>=

```
//举例:
vector<int>::iterator iter = vec.begin();
auto iter = vec.begin();
//常用
begin();//首地址
end();  //尾地址
rbegin();   //逆序首地址
rend();     //逆序尾地址
advance(iter, n); //改变iter,使iter后移三位,vector中等价于iter += n,set中等价于n次++iter
next(iter, n); //不改变iter,返回后n个迭代器,默认 n = 1
prev(iter, n); //不改变iter,返回前n个迭代器,默认 n = 1
```

pair

pair只含有两个元素，可以看作是只有两个元素的结构体

pair定义

```
pair<T1,T2> p(t1, t2);
pair<T1,T2> p = make_pair(t1, t2);
```

pair访问

```
cout << p.first << ' ' << p.second;
```

tuple

可以把tuple理解为pair的扩展，tuple可以声明二元组，也可以声明多元组

tuple定义

```
tuple<T1,T2> t = make_pair(t1, t2);
tuple<T1,T2,T3> t = make_tuple(t1, t2, t3);
tuple<T1,T2,T3,...,Tn> t(t1, t2, t3,...,tn);
```

tuple操作

```
get<idx>(t); //访问第idx个元素,下标从0开始,idx不能为变量
get<idx>(t) = newV; //修改
tuple_size<decltype(t)>::value; //返回tuple大小
tie(t1, t2, t3,...,tn) = t; //tuple变量中的值依次赋到tie的变量中
```

array

array是C++11新增的容器，效率与普通数据相差无几，比vector效率要高，自身添加了一些成员函数。

和其它容器不同，array容器的大小是**固定**的，无法动态的扩展或收缩，**只允许访问或者替换存储的元素**

array定义

```
array<T, N> arr; //类型和大小
array<T, N> arr{t1, t2, t3,...,tn}; //初始化
array<T, N> arr = {t1, t2, t3,...,tn}; //初始化
```

array操作

```
arr[index]; //与正常数组一样
get<idx>(arr); //访问第idx个元素,下标从0开始,idx不能为变量
get<idx>(arr) = newV; //修改
```

array方法函数

```
begin();//首地址
end();  //尾地址
rbegin();//逆序首地址
rend();  //逆序尾地址
size();  //返回容器中元素的个数,其值等于初始化array类的第二个模板参数N
max_size(); //返回容器中元素的最大个数,其值恒等于初始化array类的第二个模板参数N
empty();//判断容器是否为空
at(n);  //返回容器中n位置处元素的引用
front(); //返回容器中第一个元素的直接引用
back();  //返回容器中最后一个元素的直接引用
data();  //返回一个指向容器首个元素的指针
fill(x); //全部赋值为x
```

> ⚠ array没有迭代器，只有指针

string

string有迭代器

string构造函数

```
string(); //创建一个空的字符串 例如: string str;
string(const string& str); //使用一个string对象初始化另一个string对象
string(const char* s); //使用字符串s初始化
string(int n, char c); //使用n个字符c初始化
```

string基本赋值操作

```
string& operator=(const char* s);        //char*类型字符串 赋值给当前的字符串
string& operator=(const string &s);      //把字符串s赋给当前的字符串
string& operator=(char c);               //字符赋值给当前的字符串
string& assign(const char *s);           //把字符串s赋给当前的字符串
string& assign(const char *s, int n);    //把字符串s的前n个字符赋给当前的字符串
string& assign(const string &s);         //把字符串s赋给当前字符串
string& assign(int n, char c);           //用n个字符c赋给当前字符串
string& assign(const string &s, int start, int n); //将s从start开始n个字符赋值给字符
串
```

string存取字符操作

```
char& operator[](int n); //通过[]方式取字符
char& at(int n); //通过at方法获取字符
```

string拼接操作

```
string& operator+=(const string& str);            //重载+=操作符
string& operator+=(const char* str);              //重载+=操作符
string& operator+=(const char c);                 //重载+=操作符
string& append(const char *s);                    //把字符串s连接到当前字符串结尾
string& append(const char *s, int n);             //把字符串s的前n个字符连接到当前字符
串结尾
string& append(const string &s);                  //同operator+=()
string& append(const string &s, int pos, int n); //把字符串s中从pos开始的n个字符连接到
当前字符串结尾
string& append(int n, char c);                    //在当前字符串结尾添加n个字符c
```

string查找和替换

```
int find(const string& str, int pos = 0) const;      //查找str第一次出现位置,从pos开始查找
int find(const char* s, int pos = 0) const;          //查找s第一次出现位置,从pos开始查找
int find(const char* s, int pos, int n) const;       //从pos位置查找s的前n个字符第一次位置
int find(const char c, int pos = 0) const;           //查找字符c第一次出现位置
int rfind(const string& str, int pos = npos) const; //查找str最后一次位置,从pos开始查找
int rfind(const char* s, int pos = npos) const;      //查找s最后一次出现位置,从pos开始查找
int rfind(const char* s, int pos, int n) const;      //从pos查找s的前n个字符最后一次位置
int rfind(const char c, int pos = 0) const;          //查找字符c最后一次出现位置
string& replace(int pos, int n, const string& str); //替换从pos开始n个字符为字符串str
string& replace(int pos, int n, const char* s);      //替换从pos开始的n个字符为字符串s
```

string子串

```
string substr(int pos = 0, int n = npos) const; //返回由pos开始的n个字符组成的字符串
```

string插入和删除

```
string& insert(int pos, const char* s); //插入字符串
string& insert(int pos, const string& str); //插入字符串
string& insert(int pos, int n, char c); //在指定位置插入n个字符c
string& erase(int pos, int n = npos); //删除从Pos开始的n个字符
```

vector

vector有迭代器

> vector动态增加大小，并不一定在原空间之后续接新空间(因为无法保证原空间之后尚有可配置的空间)，而是一块更大的内存空间，然后将原数据拷贝新空间，并释放原空间。因此，对vector的任何操作，一旦引起空间的重新配置，指向原vector的所有迭代器就都失效了。这是程序员容易犯的一个错误，务必小心。

vector构造函数

```
vector<T> v;                //采用模板实现类实现，默认构造函数
vector(v.begin(), v.end()); //将v[begin(), end())区间中的元素拷贝给本身。
vector(n, elem);            //构造函数将n个elem拷贝给本身。
vector(const vector &vec);  //拷贝构造函数。
```

vector空间操作

```
size(); //返回容器中元素的个数
empty();//判断容器是否为空
resize(int num);
//重新指定容器的长度为num,容器变长，则以默认值填充新位置.器变短,末尾超出容器长度的元素被删
resize(int num, T elem);
//重新指定容器的长度为num,容器变长,以elem值填充新位置.器变短,末尾超出容器长度的元素被删除,
capacity();          //容器的容量
reserve(int len);    //容器预留len个元素长度,留位置不初始化,素不可访问
```

vector数据存取

```
at(int idx);//返回索引idx所指的数据,如果idx越界,抛出out_of_range异常
operator[]; //返回索引idx所指的数据,越界时，运行直接报错
front();    //返回容器中第一个数据元素
back();     //返回容器中最后一个数据元素
```

vector插入和删除

```
insert(const iterator pos, elem); //迭代器指向位置pos插入元素elem,返回新数据的迭代器
insert(const iterator pos, int count, elem); //迭代器指向位置pos插入count个元素elem
push_back(elem); //尾部插入元素elem
pop_back();      //删除最后一个元素
erase(const iterator start, const iterator end); //删除迭代器从start到end之间的元素
erase(const iterator pos); //删除迭代器指向的元素
clear();         //删除容器中所有元素
```

queue

queue没有迭代器

队列是一种先进先出的数据结构

queue赋值与初始化

```
queue<T> queT; //queue采用模板类实现，queue对象的默认构造形式
queue(const queue &que); //拷贝构造函数
queue& operator=(const queue &que); //重载等号操作符
```

queue方法函数

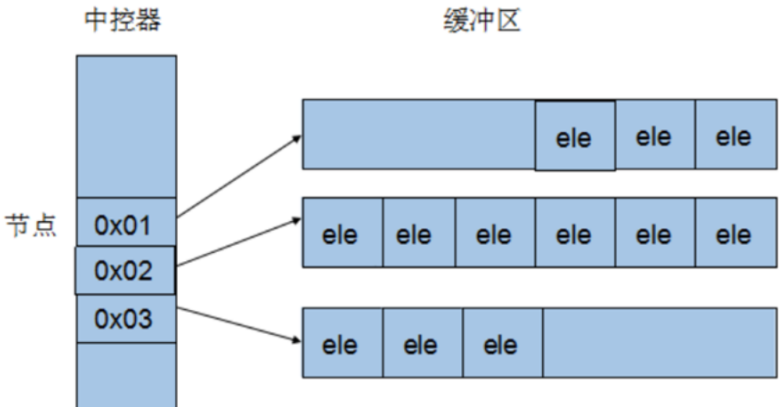
```
push(elem); //往队尾添加元素
pop();      //从队头移除第一个元素
back();     //返回最后一个元素
front();    //返回第一个元素
empty();    //判断队列是否为空
size(); //返回队列的大小
```

deque

deque没有迭代器

双端队列

> deque是由一段一段的定量的连续空间构成。一旦有必要在deque前端或者尾端增加新的空间，便配置一段连续定量的空间，串接在deque的头端或者尾端。



deque构造函数

```
deque<T> deqT;  //默认构造形式
deque(beg, end);//构造函数将[beg, end)区间中的元素拷贝给本身。
deque(n, elem);          //构造函数将n个elem拷贝给本身。
deque(const deque &deq);//拷贝构造函数。
```

deque赋值操作

```
assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem);  //将n个elem拷贝赋值给本身。
deque& operator=(const deque &deq); //重载等号操作符
swap(deq);// 将deq与本身的元素互换
```

deque空间操作

```
deque.size();//返回容器中元素的个数
deque.empty();//判断容器是否为空
deque.resize(num);
//重新指定容器的长度为num,若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
deque.resize(num, elem);
//重新指定容器的长度为num,若容器变长，则以elem值填充新位置,如果容器变短，则末尾超出容器长度的元素被删除。
```

deque插入和删除

```
//双端
push_back(elem);//在容器尾部添加一个数据
push_front(elem);//在容器头部插入一个数据
pop_back();//删除容器最后一个数据
pop_front();//删除容器第一个数据
//非双端
insert(pos, elem);       //在pos位置插入一个elem元素的拷贝,返回新数据的迭代器
insert(pos, n, elem);    //在pos位置插入n个elem数据,无返回值
insert(pos, beg, end);   //在pos位置插入[beg,end)区间的数据,无返回值
clear();         //移除容器的所有数据
erase(beg, end);//删除[beg,end)区间的数据,返回下一个数据的位置
erase(pos);      //删除pos位置的数据,返回下一个数据的位置
```

deque数据存取

```
at(idx);//返回索引idx所指的数据，如果idx越界，抛出out_of_range。
front();//返回第一个数据。
back(); //返回最后一个数据
operator[]; //返回索引idx所指的数据，如果idx越界，不抛出异常，直接出错。
```

priority_queue

priority_queue没有迭代器

优先队列，本质为大根堆

priority_queue方法函数

```
q.top(); //访问队首元素 O(1)
q.push();//入队 O(logN)
q.pop(); //出队 O(logN)
q.size();//队列元素个数 O(1)
q.empty();//是否为空
priority_queue<T, vector<T>, greater<T>>q;//小根堆
priority_queue<T, vector<T>, less<T>>q;//大根堆,等价于priority_queue<T>
```

> ⚠ priority_queue仅能通过top()访问；**注意**没有clear()

stack

stack没有迭代器

栈是STL中实现的一个先进后出，后进先出的容器

stack赋值与初始化

```
stack<T> staT; //stack采用模板类实现，stack对象的默认构造形式
stack(const stack &sta); //拷贝构造函数
stack& operator=(const stack &sta); //重载等号操作符
```

stack方法函数

```
push(elem); //往栈顶添加元素
pop();      //从栈顶移除一个元素
top();      //返回栈顶元素
empty();    //判断栈是否为空
size();     //返回栈的大小
```

list

list有迭代器

List容器是一个双向链表

> - 采用动态存储分配，不会造成内存浪费和溢出
> - 链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素
> - 链表灵活，但是空间和时间额外耗费较大

list构造函数

```
list<T> lstT; //list采用采用模板类实现,对象的默认构造形式:
list(beg,end);//构造函数将[beg, end)区间中的元素拷贝给本身。
list(n,elem); //构造函数将n个elem拷贝给本身。
list(const list &lst); //拷贝构造函数。
```

list赋值操作

```
assign(beg, end);   //将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem);    //将n个elem拷贝赋值给本身。
list& operator=(const list &lst);//重载等号操作符
swap(lst);  //将lst与本身的元素互换。
```

list数据读取

```
front();//返回第一个元素
back(); //返回最后一个元素
```

list空间操作

```
size(); //返回容器中元素的个数
empty();    //判断容器是否为空
resize(num);//重新指定容器的长度为num
//若容器变长,则以默认值填充新位置;如果容器变短,则末尾超出容器长度的元素被删除
resize(num, elem);//重新指定容器的长度为num
//若容器变长，则以elem值填充新位置;如果容器变短，则末尾超出容器长度的元素被删除。
```

list插入和删除

```
push_back(elem);    //在容器尾部加入一个元素
pop_back(); //删除容器中最后一个元素
push_front(elem);   //在容器开头插入一个元素
pop_front();    //从容器开头移除第一个元素
insert(pos,elem);   //在pos位置插elem元素的拷贝，返回新数据的位置
insert(pos,n,elem); //在pos位置插入n个elem数据，无返回值
insert(pos,beg,end);//在pos位置插入[beg,end)区间的数据，无返回值
clear();//移除容器的所有数据
erase(beg,end); //删除[beg,end)区间的数据，返回下一个数据的位置
erase(pos);     //删除迭代器pos位置的数据，返回下一个数据的位置
remove(elem);   //删除容器中所有与elem值匹配的元素
```

list反转排序

```
reverse();  //反转链表
sort(); //list排序
```

bitset

bitset没有迭代器

> ⚠ bitset具有所有的位运算operator

bitset初始化

```
bitset<N>b;
bitset<N>b(0xce);//使用整型初始化
bitset<N>b(“01101001”);//使用字符串初始化
bitset<N>b(str);//使用string初始化
```

> 初始化从右往左填，例子：std::bitset<16> baz2("01101001"); //0000000001101001
>
> 下标从右往左，类似二进制表示，例子：bit(001); //bit[0] == 1

bitset的访问

```
b[idx]; //operator[]
```

bitset方法函数

```
size(); //返回bitset大小
count(); //统计true的个数
set(); //全部赋值为true
set(idx, bool flag); //idx赋值为flag(默认true)
reset(); //全部赋值为true
reset(idx); //idx位赋值qu为false
flip(); //全部反转
flip(idx); //idx位反转
all(); //全为true返回true
any(); //至少有一个bit位为true返回true
none();//全为false返回true
test(idx); //返回bit[idx]
```

set和multiset

set和multiset有迭代器

set中所有元素会被排序，set不允许两个元素相同

multiset特性及用法和set完全相同，唯一的差别在于允许两个元素相同

> set和multiset的底层实现是红黑树
>
> set拥有和list某些相同的性质，当对容器中的元素进行插入操作或者删除操作的时候，操作之前所有的迭代器，在操作完成之后依然有效，被删除的那个元素的迭代器必然是一个例外

set构造函数

```
set<T> st;          //set默认构造函数:
mulitset<T> mst;    //multiset默认构造函数:
set(const set &st); //拷贝构造函数
```

set赋值操作

```
set& operator=(const set &st); //重载等号操作符
swap(st); //交换两个集合容器
```

set空间操作

```
size();  //返回容器中元素的数目
empty(); //判断容器是否为空
```

set插入和删除

```
insert(elem);//在容器中插入元素,返回pair<std::set<T>::iterator, bool>
//插入失败bool为false
clear();//清除所有元素
erase(pos);//删除pos迭代器所指的元素,回下一个元素的迭代器
erase(beg, end);//删除区间[beg,end)的所有元素,返回下一个元素的迭代器
erase(elem);//删除容器中值为elem的元素
```

set查找操作

```
find(key);  //查找键key是否存在,若存在,返回该键的元素的迭代器;若不存在,返回
set.end(),O(logn)
count(key); //查找键key的元素个数
lower_bound(keyElem);//返回第一个key>=keyElem元素的迭代器
upper_bound(keyElem);//返回第一个key>keyElem元素的迭代器
equal_range(keyElem);//返回容器中key与keyElem相等的上下限的[l, r)两个迭代器
```

map和multimap

map和multimap有迭代器

map中所有元素会被排序，map所有的元素都是pair,同时拥有实值和键值，pair的第一元素被视为键值，第二元素被视为实值，map不允许两个元素有相同的键值

multimap特性及用法和map完全相同，唯二的差别在于允许两个键值相同，不支持operator[]

> map和multimap的底层实现是红黑树
>
> map拥有和list某些相同的性质，当对容器中的元素进行插入操作或者删除操作的时候，操作之前所有的迭代器，在操作完成之后依然有效，被删除的那个元素的迭代器必然是一个例外

map构造函数

```
map<T1, T2> mapTT;  //map默认构造函数
map(const map &mp); //拷贝构造函数
```

map赋值操作

```
map& operator=(const map &mp); //重载等号操作符
swap(mp); //交换两个集合容器
```

map空间操作

```
size(); //返回容器中元素的数目
empty();//判断容器是否为空
```

map插入数据元素

```
map.insert(...); //往容器插入元素，返回pair<iterator,bool>
map<key, val> mapStu;
// 第一种 通过pair的方式插入对象
mapStu.insert(pair<key, val>(key, val));
// 第二种 通过pair的方式插入对象
mapStu.inset(make_pair(key, val));
// 第三种 通过value_type的方式插入对象
mapStu.insert(map<key, val>::value_type(key, val));
// 第四种 通过数组的方式插入值
mapStu[key] = val;
```

> ⚠ multimap不支持operator[key]

map删除操作

```
clear();    //删除所有元素
erase(pos); //删除pos迭代器所指的元素，返回下一个元素的迭代器。
erase(beg, end);//删除区间[beg,end)的所有元素 ，返回下一个元素的迭代器。
erase(keyElem);//删除容器中key为keyElem的对组。
```

map查找操作

```
find(key);  //查找键key是否存在,若存在,返回该键的元素的迭代器;若不存在，返回map.end()
count(keyElem); //返回容器中key为keyElem的对组个数,对map来说,要么是0,要么是1,对multimap来说,值可能大于1
lower_bound(keyElem);//返回第一个key>=keyElem元素的迭代器
upper_bound(keyElem);//返回第一个key>keyElem元素的迭代器
equal_range(keyElem);//返回容器中key与keyElem相等的上下限的[l, r)两个迭代器
```

unordered

unordered_set和unordered_map存储元素时是没有顺序的，其中所有元素**不会**被排序

数学

数论

素数判定

```
bool is_prime(int x)  // 判定质数
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

素数线性筛

时间复杂度 $O(n)$
三种情况

设 x 的最小质因子为 y，即 $x = i \times y$

1. $x \in$ prime: 2. $i \bmod y = 0$,
3. $i \bmod y \neq 0$,

```
int pri[N], pripos, non_pri[N]; // int mind[N];
void linear_prime(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!non_pri[i])
        {
            pri[++pripos] = i;
            // mind[i] = i; // 记录每个数最小质因数
        }
        for (int j = 1; j <= pripos && pri[j] * i <= n; j++)
        {
            non_pri[pri[j] * i] = 1;
            // mind[pri[j] * i] = pri[j]; // 记录每个数最小质因数
            if (i % pri[j] == 0)
                break;
        }
    }
}
```

质因数分解

时间复杂度最差 $O(\sqrt{n})$，最优 $O(logn)$

```
int cnt[N];
void factor(ll n)
{
    for (int i = 2; i <= n / i; ++i)
    {
        while (n % i == 0)
        {
            cnt[i]++;
            n /= i;
        }
    }
    if(n != 1) cnt[n]++; //大于sqrt(n)的质因子
}
```

因子预处理

```
void factor() //O(nlogn)
{
    for (int i = 1; i <= 100000; i++)
        for (int j = i; j <= 100000; j += i)
            factor[j].push_back(i);
}
```

欧拉函数

$$\phi(pn) = (p-1)\phi(n)(p \in \text{Prime}, p \nmid n)$$

$$\varphi(\mathrm{N}) = \mathrm{N} * \prod(1 - 1/\mathrm{p})（\mathrm{P}是数\mathrm{N}的质因数）$$

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

```
ll euler_phi(ll n)
{
    ll ans = n;
    for (ll i = 2; i <= n / i; i++)
        if (n % i == 0)
        {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    if (n > 1) ans = ans * (n - 1) / n;
    return ans;
}
```

线性欧拉函数

通过线性筛从而线性推出欧拉函数

```
int pri[N], phi[N], pripos;
bool non_pri[N];
void linear_phi(int n)
{
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        if (!non_pri[i]) pri[++pripos] = i, phi[i] = i - 1;
        for (int j = 1; j <= pripos && pri[j] * i <= n; j++)
```

```
        {
            non_pri[pri[j] * i] = 1;
            if (i % pri[j] == 0)
            {
                phi[i * pri[j]] = phi[i] * pri[j];
                break;
            }
            phi[i * pri[j]] = phi[i] * phi[pri[j]];
        }
    }
}
```

线性莫比乌斯

μ 为莫比乌斯函数，定义为

$$\mu(n)=\begin{cases}1 & n=1\\ 0 & n\text{ 含有平方因子}\\ (-1)^k & k\text{ 为 }n\text{ 的本质不同质因子个数}\end{cases}$$

性质：

莫比乌斯函数不仅是积性函数，还有如下性质：

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

即 $\sum_{d|n}\mu(d)=\varepsilon(n)$，$\mu*1=\varepsilon$

根据二项式定理，易知该式子的值在 $k=0$ 即 $n=1$ 时值为 1 否则为 0，这也同时证明了 $\sum_{d|n}\mu(d)=[n=1]=\varepsilon(n)$ 以及 $\mu*1=\varepsilon$

$$\sum_{i=1}^{n}\sum_{d|i}\mu(d)=\sum_{d=1}^{n}\left\lfloor\frac{n}{d}\right\rfloor\mu(d)$$

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

$$\sum_{d|n}\frac{\mu(d)}{d}=\frac{\varphi(n)}{n}$$

其中φ(n)是 欧拉函数

$$\sum_{i=1}^{n}\sum_{j=1}^{m}[gcd(i,j)=1]=\sum_{d=1}^{min(n,m)}\mu(d)*\left\lfloor\frac{n}{d}\right\rfloor*\left\lfloor\frac{m}{d}\right\rfloor$$

```
int pri[N], pripos, mu[N];
bool non_pri[N];
void linear_mu(int n)
{
    mu[1] = 1;
    for (int i = 2; i <= n; ++i)
```

```
    {
        if (!non_pri[i]) pri[++pripos] = i, mu[i] = -1;
        for (int j = 1; j <= pripos && i * pri[j] <= n; ++j)
        {
            non_pri[i * pri[j]] = 1;
            if (i % pri[j] == 0)
            {
                mu[i * pri[j]] = 0;
                break;
            }
            mu[i * pri[j]] = -mu[i];
        }
    }
}
```

欧拉定理

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

阶与原根

阶

定义

由欧拉定理可知，对 $a\in \mathbf{Z}$，$m\in \mathbf{N}^*$，若 $(a,m)=1$，则 $a^{\varphi(m)}\equiv 1 \pmod m$.

因此满足同余式 $a^n\equiv 1 \pmod m$ 的最小正整数 n 存在，这个 n 称作 a 模 m 的阶，记作 $\delta_m(a)$ 或 $\mathrm{ord}_m(a)$.

性质 1

$a, a^2, \cdots, a^{\delta_m(a)}$ 模 m 两两不同余。

性质 2

若 $a^n\equiv 1 \pmod m$，则 $\delta_m(a)\mid n$.

性质 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

$$(\delta_m(a),\delta_m(b))=1$$

性质 4

设 $k\in \mathbf{N}$，$m\in \mathbf{N}^*$，$a\in \mathbf{Z}$，$(a,m)=1$，则

$$\delta_m(a^k)=\frac{\delta_m(a)}{(\delta_m(a),k)}$$

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 $\delta_m(g)=|\mathbf{Z}_m^*|=\varphi(m)$. 当 m 是质数时，我们有 $g^i \bmod m$, $0<i<m$ 的结果互不相同。

原根个数

若一个数 m 有原根，则它原根的个数为 $\varphi(\varphi(m))$.

原根存在定理

原根存在定理

一个数 m 存在原根当且仅当 $m=2,4,p^\alpha,2p^\alpha$，其中 p 为奇素数，$\alpha \in \mathbf{N}^*$.

最小原根的范围估计

王元[2]和 Burgess[1]证明了素数 p 的最小原根 $g_p=O\left(p^{0.25+\epsilon}\right)$，其中 $\epsilon>0$.

Fridlander[3]和 Salié[4]证明了素数 p 的最小原根 $g_p=\Omega(\log p)$.

这保证了我们暴力找一个数的最小原根，复杂度是可以接受的。

幂塔

幂塔的无限层取模是定值

幂塔函数：形如

$$a^{a^{a^{\cdots}}} \bmod m$$

```
bool check(ll a, ll k, ll p)
{
    if (a >= p) return true; // 底数a>=p
    if (k == 0) return p <= 1; // 0层幂塔是1
    return check(a, k - 1, log(p) / log(a)); // 取对数，消去一层，继续判断
}
ll tower(ll a, ll k, ll p) // 返回k层幂塔 || k足够大就是无限
{
    if (p == 1) return 0; //mod 1 == 0
    if (k <= 1) return qpow(a, k, p);
    if (gcd(a, p) == 1) return qpow(a, tower(a, k - 1, phi[p]), p); // 欧拉定理
    if (check(a, k - 1, phi[p])) return qpow(a, tower(a, k - 1, phi[p]) +
phi[p], p); // 拓展欧拉定理情况2
    return qpow(a, tower(a, k - 1, phi[p]), p); // 拓展欧拉定理情况3
}
```

快速幂

时间复杂度 $O(logn)$

```
ll qpow(ll base, ll pow, ll mod)
{
    ll res = 1;
    base %= mod;
    while (pow)
    {
        if (pow & 1) res = res * base % mod;
        base = base * base % mod;
        pow >>= 1;
    }
    return res;
}
```

拓展欧几里得

指可找出一对整数(x,y)，使$ax+by=gcd(a,b)$成立，这里的x和y**不一定是正数**也可能是0或负数。

裴蜀定理：对于任何整数a，b和他们的最大公约数$gcd(a,b)$，其关于x，y的线性方程$ax+by=c$有整数解，当且仅当c是$gcd(a,b)$的倍数。裴蜀定理有解时必有无穷多个解。

```
//ax+by=gcd(a,b)//(eg_res = x, eg_temp = y)
//返回gcd(a,b), eg_res = a关于模b的逆元
ll exgcd(ll a, ll b, ll &eg_res, ll &eg_temp)
{
    if (b == 0)
    {
        eg_res = 1, eg_temp = 0;
        return a;
    }
    ll exi = exgcd(b, a % b, eg_temp, eg_res);
    eg_temp -= a / b * eg_res;
    return exi;
}
ll getInv(int a,int mod)//求a在mod下的逆元，不存在逆元返回-1
{
    ll x,y;
    ll d = exgcd(a, mod, x, y);
    return d == 1 ? (x % mod + mod) % mod : -1;
}
```

类欧几里得算法

$$求解\ f(a,b,c,n)=\sum_{i=0}^{n}\lfloor\frac{ai+b}{c}\rfloor$$

时间复杂度$O(logn)$

```
int floor_sum(int a, int b, int c, int n)//偶数项求和a=2,n=n/2;奇数项求和再设置b=1;求x%y=z,设置a=y,b=z,c=1,n=(n-z)/y
{
    int res = 0;
    if(a >= c)
    {
        res += n * (n + 1) * (a / c) / 2;
        a %= c;
    }
```

```
    if(b >= c)
    {
        res += (n + 1) * (b / c);
        b %= c;
    }
    int m = (a * n + b) / c;
    if(m == 0) return res;
    res += n * m - floor_sum(c, c - b - 1, a, m - 1);
    return res;
}
```

费马定理逆元

要求 b 为质数

```
inline ll inv(ll a, ll b) { return qpow(a, b - 2, b); }
```

线性逆元

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

阶乘逆元

```
ll C(ll n, ll m)
{
    if(n < m) return 0;
    return fact[n] * inv[m] % mod * inv[n - m] % mod;
}
```

中国剩余定理CRT

中国剩余定理 (Chinese Remainder Theorem, CRT) 可求解如下形式的一元线性同余方程组（其中 $n_1, n_2, \cdots, n_k$ 两两互质）：

$$\begin{cases} x & \equiv a_1 \pmod{n_1} \\ x & \equiv a_2 \pmod{n_2} \\ & \vdots \\ x & \equiv a_k \pmod{n_k} \end{cases}$$

上面的「物不知数」问题就是一元线性同余方程组的一个实例。

过程

1. 计算所有模数的积 n；
2. 对于第 i 个方程：
    a. 计算 $m_i = \frac{n}{n_i}$；
    b. 计算 m_i 在模 n_i 意义下的 逆元 m_i^{-1}；
    c. 计算 $c_i = m_i m_i^{-1}$（**不要对 n_i 取模**）。
3. 方程组在模 n 意义下的唯一解为：$x = \sum_{i=1}^{k} a_i c_i \pmod{n}$。

可以转化为

调用拓展欧几里得求逆元函数

```
//缺少exgcd函数
//ll modul[N], remain[N];//模数和余数
ll crt(ll k, ll *remain, ll *modul)
{
    ll n = 1, ans = 0;
    for (int i = 1; i <= k; i++) n = n * modul[i];
    for (int i = 1; i <= k; i++)
    {
        ll m = n / modul[i], b, y;
        exgcd(m, modul[i], b, y);//拓展欧几里得求逆元
        ans = (ans + remain[i] * m * b % n) % n;
    }
    return (ans % n + n) % n;
}
```

拓展中国剩余定理

对比CRT，模数不要求互质

```
ll modul[N], remain[N];
ll excrt(int number)
{
    ll lcm = modul[1], sum = remain[1], x, y, gcd;
    bool fail = false;
    for (int i = 2; i <= number; ++i)
    {
        remain[i] = ((remain[i] - sum) % modul[i] + modul[i]) % modul[i];
        gcd = exgcd(lcm, modul[i], x, y);//拓展欧几里得
        if (remain[i] % gcd == 0) x = x * (remain[i] / gcd) % modul[i];
        else
        {
            fail = true;
            break;
        }
        sum += x * lcm;
        lcm = lcm / gcd * modul[i];
        sum = (sum % lcm + lcm) % lcm;
    }
    return fail ? -1 : sum;
}
```

辛普森积分Simpson

一、牛顿 - 莱布尼茨公式（微积分的基本公式）：

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

其中，函数F(x) 是连续函数f(x)在区间[a,b]上的原函数。

二、辛普森（Simpson）积分公式

1.Simpson积分公式是将区间端点和区间中点三个点近似看成抛物线上对应的三个点，以二次曲线逼近的方式取代矩形或梯形积分公式，以求得定积分的数值近似解。

$$\int_a^b f(x)\,dx \approx \frac{(b-a)}{6} \bullet \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

```
f(double x)
{
    // TODO: 实现所求的函数
}
double simpson(double l, double r)  // 辛普森积分公式
{
    auto mid = (l + r) / 2;
    return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;
}
double asr(double l, double r, double s)  // 自适应
{
    auto mid = (l + r) / 2;
    auto left = simpson(l, mid), right = simpson(mid, r);
    if (fabs(left + right - s) < eps) return left + right;
    return asr(l, mid, left) + asr(mid, r, right);
}
```

威尔逊定理

素数判别定理

$$\frac{1\cdot 2\cdot 3\cdot 4\cdots(P-1)}{P} = x余(P-1)$$

$$(p-1)! \equiv 1 * (p-1) \equiv -1(modp)$$

牛顿迭代

牛顿迭代公式

设r是$f(x)=0$的根，选取x_0作为r的初始近似值，则我们可以过点$(x_0,f(x_0))$做曲线$y=f(x)$的切线L,我们知道切线与x轴有交点，我们已知切线L的方程为$L:y=f(x_0)+f^{'}(x_0)(x-x_0)$我们求的它与$x$轴的交点为$x_1=x_0-\frac{f(x_0)}{f^{'}(x_0)}$. 我们在以$(x_1,f(x_1))$斜率为$f^{'}(x_1)$做斜线，求出与$x$轴的交点，重复以上过程直到$f(x_n)$无限接近于0即可。其中第n次的迭代公式为：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

BSGS (大步小步)

问题：

给出a，b，p，其中$\gcd(a,p)=1$，求x满足

$$a^x\equiv b(\bmod p)$$

思路：

设$x=A\sqrt{p}-B$其中$A\in[1,\sqrt{p}],B\in[0,\sqrt{p}]$，得到问题的变形

$$a^{A\sqrt{p}-B}\equiv b(\bmod p)$$
$$a^{A\sqrt{p}}\equiv ba^B(\bmod p)$$

我们先枚举B，算出每个$ba^B \bmod p$，用unordered_map存起来，再枚举A，计算出$a^{A\sqrt{p}}$，在unordered_map中找相同的值，这样的A,B就能恰好凑成一对答案。复杂度$O(\sqrt{p})$，如果用map的话，就多一个log。

```
unordered_map<ll, ll> mp;
ll bsgs(ll a, ll b, ll p)
{
    if (a % p == 0) return -1;
    mp.clear();
    ll k = ceil(sqrt(p));
    //枚举B
    for (int i = 0; i <= k; i++)
    {
        mp[b] = i;
        b = b * a % p;
    }
    ll suba = qpow(a, k, p), A = suba;
    //枚举A
    for (int i = 1; i <= k; i++)
    {
        if (mp[suba]) return 1ll * i * k - mp[aa] + 1;
        suba = suba * A % p;
    }
    return -1;
}
```

SQRT

优化牛顿迭代求sqrt，如今系统的sqrt更快（近似“ope+”时间复杂度）

```
double Q_sqrt(float x)//John Carmack
{
    float M = x * 0.5F;
    int i = *(int *)&x;
    i = 0X5f3759ce - (i >> 1);
    x = *(float *)&i;
    x = x * (1.5F - (M * x * x));//iteration ...
    return 1/x;
}
```

GCD

```
ll kgcd(ll a, ll b)  //更相减损术
{
    if (a == 0) return b;
    if (b == 0) return a;
    if (!(a & 1) && !(b & 1)) return kgcd(a>>1, b>>1)<<1;
    else if (!(b & 1)) return kgcd(a, b>>1);
    else if (!(a & 1)) return kgcd(a>>1, b);
    else  return kgcd(abs(a - b), min(a, b));
}
```

```
inline ll gcd(ll a, ll b)  //辗转相除||欧几里得法
{
    while(b) b ^= a ^= b ^= a %= b;
    return a;
}
```

```
inline int gcd(int a, int b)  //Binary GCD(用int最快,减少指令运算位数)||(本质是更相减损术)||O(lg a)
{
    if (!(a && b)) return a | b;
    int sufa = __builtin_ctz(a), sufb = __builtin_ctz(b), t;
    int suf = sufa < sufb ? sufa : sufb;
    b >>= sufb;
    while (a)
    {
        a >>= sufa;
        t = a - b;
        b = a < b ? a : b;
        a = t > 0 ? t : -t;
        sufa = __builtin_ctz(t);
    }
    return b << suf;
}
```

基于值域预处理GCD

gcd的查询复杂度 $O(1)$
时间复杂度 $O(V)$， V为值域范围

```
int pri[N], pripos, non_pri[N], k[N][3], subn; // sqrt(N),在lemma中初始化;
vector<vector<int>> _gcd; // 在lemma中重置大小;
inline int gcd(int a, int b)
```

```
{
    int g = 1;
    for (int temp, i = 0; i < 3; i++, b /= temp, g *= temp)
        if (k[a][i] > subn) temp = b % k[a][i] ? 1 : k[a][i]; // 和数据范围相关
        else temp = _gcd[k[a][i]][b % k[a][i]];
    return g;
}
void lemma(int n) // 值域数据范围n(填N小心越界)
{
    k[1][0] = k[1][1] = k[1][2] = non_pri[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        if (!non_pri[i]) pri[++pripos] = k[i][2] = i, k[i][1] = k[i][0] = 1;
        for (int j = 1; pri[j] * i <= n; j++)
        {
            non_pri[i * pri[j]] = 1;
            int *temp = k[i * pri[j]];
            temp[0] = k[i][0] * pri[j], temp[1] = k[i][1], temp[2] = k[i][2];
            sort(temp, temp + 3);
            if (i % pri[j] == 0) break;
        }
    }
    subn = sqrt(n);
    _gcd.resize(subn + 10);
    for (auto &it : _gcd) it.resize(subn + 10);
    for (int i = 1; i <= subn; i++) _gcd[i][0] = _gcd[0][i] = i;
    for (int i = 1; i <= subn; i++)
        for (int j = 1; j <= i; j++)
            _gcd[j][i] = _gcd[i][j] = _gcd[i % j][j];
}
```

基于值域GCD求和

$$\text{问题：求解} \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} gcd(a_i, a_j)$$

首先我们如果单纯的两两求gcd肯定是不行的，观察数据后发现a数组的数据很小，也就是说a的因子个数很小。而两个数的gcd也就是他们的最大公共因子。因此可以考虑统计区间内所有因子的个数，然后对于每个右端点 a_j 而言，遍历其所有因子，区间之和就是（右端点的因子大小 * 区间内该因子的个数）的总和。

但是仔细想想之后发现是有重复的：假如当前因子是2，用2 * 区间内2的因子个数是不正确的，因为有2这个因子也必然会有1这个因子。gcd是两个数的最大公共因子，所以统计因子2时会把1这个因子给撤销掉，后面任意因子也是同理。也就是说，2这个因子的实际价值不是2而是1。3也是同理，3的实际价值需要减去1的实际价值， 而6的实际价值需要减去1、2、3实际价值。因此对于任意一个因子而言，其实际价值需要需要减去其所有不为他本身的因子的实际价值。

```
void init() //贡献预处理
{
    for(int i = 1; i<=100000; ++i)
    {
        factor[i].push_back(i);
        val[i]=i;
    }
    for(int i = 1; i<=100000; ++i)
        for(int j = 2*i; j<=100000;j+=i)
```

```
        {
            factor[j].push_back(i);
            val[j]-=val[i];
        }
}
```

组合数学

> *组合意义天地灭，代数推导保平安。*

排列组合数大全

排列数Permutation

排列的计算公式如下：

$$A_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

组合数Combination

组合数计算公式

$$\binom{n}{m} = \frac{A_n^m}{m!} = \frac{n!}{m!(n-m)!}$$

杨辉三角求法

```
ll c[2023][2023]; //c[n][m];n>=m;
inline void build_C()
{
    c[0][0] = c[1][0] = c[1][1] = 1;
    for (int i = 2; i <= 2000; i++)
    {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++)
            c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
    }
}
```

求单个组合数（含 $mod\ p$ ）

```
ll C(ll n, ll m, ll p)
{
    if (n < m) return 0;
    if (m > n - m) m = n - m;
    ll a = 1, b = 1;
    for (int i = 0; i < m; i++)
    {
        a = (a * (n - i)) % p;
        b = (b * (i + 1)) % p;
    }
    return a * inv(b, p) % p;
}
```

圆排列Circular

圆排列 ¶

n 个人全部来围成一圈，所有的排列数记为 Q_n^n。考虑其中已经排好的一圈，从不同位置断开，又变成不同的队列。 所以有

$$\mathrm{Q}_n^n \times n = \mathrm{A}_n^n \Longrightarrow \mathrm{Q}_n = \frac{\mathrm{A}_n^n}{n} = (n-1)!$$

由此可知部分圆排列的公式：

$$\mathrm{Q}_n^r = \frac{\mathrm{A}_n^r}{r} = \frac{n!}{r \times (n-r)!}$$

错位排序Derangement

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

这里也给出另一个递推关系：

$$D_n = nD_{n-1} + (-1)^n$$

$$D_n = \left\lfloor \frac{n!}{\mathrm{e}} \right\rfloor$$

随着元素数量的增加，形成错位排列的概率 P 接近：

$$P = \lim_{n \to \infty} \frac{D_n}{n!} = \frac{1}{\mathrm{e}}$$

卡特兰数Catalan

递推式

该递推关系的解为：

$$H_n = \frac{\binom{2n}{n}}{n+1} (n \geq 2, n \in \mathbf{N}_+)$$

关于 Catalan 数的常见公式：

$$H_n = \begin{cases} \sum_{i=1}^{n} H_{i-1} H_{n-i} & n \geq 2, n \in \mathbf{N}_+ \\ 1 & n = 0, 1 \end{cases}$$

$$H_n = \frac{H_{n-1}(4n-2)}{n+1}$$

$$H_n = \binom{2n}{n} - \binom{2n}{n-1}$$

斯特林数Stirling

LinXce:跟组合数太相似了，递推式和组合数杨辉三角也很像.

第一类斯特林数（斯特林轮换数）$\begin{bmatrix} n \\ k \end{bmatrix}$，也可记做 $s(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D]=[B,C,D,A]=[C,D,A,B]=[D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D]\neq[D,C,B,A]$。

递推式

$$\begin{bmatrix} n \\ k \end{bmatrix}=\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}+(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$$

边界是 $\begin{bmatrix} n \\ 0 \end{bmatrix}=[n=0]$。

通项公式

第一类斯特林数没有实用的通项公式。

```
ll sti1[5010][5010]; //O(n)第一类斯特兰数
ll stirling1(ll n, ll k)
{
    if (sti1[n][k]) return sti1[n][k];
    if (k <= 0 || n < k) return sti1[n][k] = (n == k);
    return sti1[n][k] = (stirling1(n - 1, k - 1) + (n - 1) * stirling1(n - 1, k)
% mod) % mod;
}
```

第二类斯特林数（斯特林子集数）$\begin{Bmatrix} n \\ k \end{Bmatrix}$，也可记做 $S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。

递推式

$$\begin{Bmatrix} n \\ k \end{Bmatrix}=\begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}+k\begin{Bmatrix} n-1 \\ k \end{Bmatrix}$$

边界是 $\begin{Bmatrix} n \\ 0 \end{Bmatrix}=[n=0]$。

通项公式

$$\begin{Bmatrix} n \\ m \end{Bmatrix}=\sum_{i=0}^{m}\frac{(-1)^{m-i}i^n}{i!(m-i)!}$$

```
ll sti2[5010][5010];//O(n)第二类斯特兰数
ll stirling2(ll n, ll k)
{
    if (sti2[n][k]) return sti2[n][k];
    if (k <= 0 || n < k) return sti2[n][k] = (n == k);
    return sti2[n][k] = (stirling2(n - 1, k - 1) + k * stirling2(n - 1, k) %
mod) % mod;
}
```

暴力求单个第二类斯特林数

```
/*qpow略*/
ll fact[N], inv[N];
void bulid_A(int n)
{
    fact[0] = inv[0] = 1;
    for (int i = 1; i <= n; ++i)
        fact[i] = fact[i - 1] * i % mod;
    for (int i = 1; i <= n; ++i)
        inv[i] = qpow(fact[i], mod - 2, mod);
}
ll stirling2(int n, int m)
{
    ll ans = 0;
    for (int i = 0; i <= m; ++i)
        ans = (ans + qpow(-1,m-i, mod) * qpow(i,n,mod)%mod * inv[i]%mod * inv[m-
i]%mod)%mod;
    return (ans + mod) % mod;
}
```

同一列的斯特林数 $O(nlogn)$ (NTT)

```
ll invn, invG, G = 3, mod = 998244353; // mod取值有限制(NTT)
ll fac[N], inv[N], r[N << 2];
ll powM(ll a, ll t = mod - 2)
{
    ll ans = 1, buf = a;
    while (t)
    {
        if (t & 1) ans = (ans * buf) % mod;
        buf = (buf * buf) % mod;
        t >>= 1;
    }
    return ans;
}
void NTT(ll *f, bool op, int n)
{
    for (int i = 0; i < n; i++) if (r[i] < i) swap(f[r[i]], f[i]);
    for (int len = 1; len < n; len <<= 1)
    {
        int w = powM(op == 1 ? G : invG, (mod - 1) / len / 2);
        for (int p = 0; p < n; p += len + len)
        {
            ll buf = 1;
            for (int i = p; i < p + len; i++)
            {
```

```
                int sav = f[i + len] * buf % mod;
                f[i + len] = f[i] - sav;
                if (f[i + len] < 0)  f[i + len] += mod;
                f[i] = f[i] + sav;
                if (f[i] >= mod)  f[i] -= mod;
                buf = buf * w % mod;
            } // F(x)=FL(x^2)+x*FR(x^2)  // F(W^k)=FL(w^k)+W^k*FR(w^k) //
F(W^{k+n/2})=FL(w^k)-W^k*FR(w^k)
        }
    }
}
ll g[N << 2];
void rev(ll *f, int len)
{
    for (int i = 0; i < len; i++) g[i] = f[i];
    for (int i = 0; i < len; i++) f[len - i - 1] = g[i];
}
// 令f=f*g (mod x^lim)
void times(ll *f, ll *gg, int len, int lim)
{
    int m = len + len, n;
    for (int i = 0; i < len; i++) g[i] = gg[i];
    for (n = 1; n < m; n <<= 1);
    invn = powM(n);
    for (int i = len; i < n; i++) g[i] = 0;
    for (int i = 0; i < n; i++) r[i] = (r[i >> 1] >> 1) | ((i & 1) ? n >> 1 :
0);
    NTT(f, 1, n);
    NTT(g, 1, n);
    for (int i = 0; i < n; ++i) f[i] = (f[i] * g[i]) % mod;
    NTT(f, 0, n);
    for (int i = 0; i < lim; ++i) f[i] = (f[i] * invn) % mod;
    for (int i = lim; i < n; ++i) f[i] = 0;
}
void Init(int lim)
{
    inv[1] = inv[0] = fac[0] = 1;
    for (int i = 1; i <= lim; i++) fac[i] = fac[i - 1] * i % mod;
    for (int i = 2; i <= lim; i++) inv[i] = inv[mod % i] * (mod - mod / i) %
mod;
    for (int i = 2; i <= lim; i++) inv[i] = inv[i - 1] * inv[i] % mod;
    for (int i = 1; i <= lim; i++) inv[i] = powM(fac[i]);
}
ll p[N << 2];
// 求出F(x-c)
void fminus(ll *s, ll *f, int len, int c)
{
    c = mod - c;
    for (int i = 0; i < len; i++) p[len - i - 1] = f[i] * fac[i] % mod;
    ll buf = 1;
    for (int i = 0; i < len; i++, buf = buf * c % mod) s[i] = buf * inv[i] %
mod;
    times(p, s, len, len);
    for (int i = 0; i < len; i++) s[len - i - 1] = p[i] * inv[len - i - 1] %
mod;
    for (int i = len; i < len + len; i++) s[i] = 0;
}
ll f[N << 2], s[N << 2];
```

```
void solve(ll *f, int n)
{
    if (n == 1) f[0] = 0, f[1] = 1;
    else if (n & 1)
    {
        solve(f, n - 1);
        f[n] = 0;
        // 再乘上(x-n+1)就好了
        for (int i = n; i > 0; i--) f[i] = (f[i - 1] + (mod - n + 1) * f[i]) %
mod;
        f[0] = f[0] * (mod - n + 1) % mod;
    }
    else
    {
        solve(f, n / 2);
        // S(x)=F(x+n/2)
        fminus(s, f, n / 2 + 1, n / 2);
        times(f, s, n / 2 + 1, n + 1);
    }
}
void invp(ll *f, int len, int k)
{
    for (int i = 0; i < k + 1; i++) s[i] = p[i] = 0; // 注意清空
    ll *r = s, *rr = p;
    int n = 1;
    for (; n < len; n <<= 1);
    rr[0] = powM(f[0]);
    for (int len = 2; len <= n; len <<= 1)
    {
        for (int i = 0; i < len; i++) r[i] = rr[i] * 2 % mod;
        times(rr, rr, len / 2, len);
        times(rr, f, len, len);
        for (int i = 0; i < len; i++) rr[i] = (r[i] - rr[i] + mod) % mod;
    }
    for (int i = 0; i < len; i++) f[i] = rr[i];
}
void stirow(int n, int k) // 第k列的斯特林数 (n >= k)
{
    invG = powM(G);
    Init(k);
    solve(f, k + 1);
    for (int i = 0; i < k + 1; i++) f[i] = f[i + 1];
    rev(f, k + 1);
    for (int i = n - k + 1; i < k + 1; i++) f[i] = 0;
    for (int i = k + 1; i < n - k + 1; i++) f[i] = 0;
    invp(f, n - k + 1, k); // f[i] = {k + i, k}; 下标(0 ~ n-k)
}
```

上升幂与普通幂的相互转化

我们记上升阶乘幂 $x^{\bar{n}} = \prod_{k=0}^{n-1}(x+k)$。

则可以利用下面的恒等式将上升幂转化为普通幂：

$$x^{\bar{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^{\bar{k}}$$

下降幂与普通幂的相互转化

我们记下降阶乘幂 $x^{\underline{n}} = \dfrac{x!}{(x-n)!} = \prod_{k=0}^{n-1}(x-k)$。

则可以利用下面的恒等式将普通幂转化为下降幂：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式：

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

补充：

> $$\begin{bmatrix} k \\ k-1 \end{bmatrix} = \begin{Bmatrix} k \\ k-1 \end{Bmatrix} = \binom{k}{2}$$

贝尔数Bell

B_n 是基数为 n 的集合的划分方法的数目。集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是 S。例如 $B_3 = 5$ 因为 3 个元素的集合 a, b, c 有 5 种不同的划分方法：

$$\{\{a\},\{b\},\{c\}\}$$
$$\{\{a\},\{b,c\}\}$$
$$\{\{b\},\{a,c\}\}$$
$$\{\{c\},\{a,b\}\}$$
$$\{\{a,b,c\}\}$$

B_0 是 1 因为空集正好有 1 种划分方法。

贝尔数适合递推公式：

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

贝尔三角形

用以下方法构造一个三角矩阵（形式类似杨辉三角形）：

- $a_{0,0}=1$；
- 对于 $n\geq 1$，第 n 行首项等于上一行的末项，即 $a_{n,0}=a_{n-1,n-1}$；
- 对于 $m,n\geq 1$，第 n 行第 m 项等于它左边和左上角两个数之和，即 $a_{n,m}=a_{n,m-1}+a_{n-1,m-1}$。

部分结果如下：

1						
1	2					
2	3	5				
5	7	10	15			
15	20	27	37	52		
52	67	87	114	151	203	
203	255	322	409	523	674	877

每行的首项是贝尔数。可以利用这个三角形来递推求出贝尔数。

伯努利数Bernoulli

主要与数论有关

伯努利数由隐含的递推关系定义：

$$\sum_{j=0}^{m}\binom{m+1}{j}B_j=0,(m>0)$$

$$B_0=1$$

恩特林格数Entringer

恩特林格数

恩特林格数（Entringer number，OEIS A008281）$E(n,k)$ 是满足下述条件的 0 到 n 共 $n+1$ 个数的置换数目：

- 首元素是 k；
- 首元素的下一个元素比首元素小，再下一个元素比前一个元素大，再下一个元素比前一个元素小……后面相邻元素的大小关系均满足这样的规则。

恩特林格数的初值有：

$$E(0,0)=1$$

$$E(n,0)=0$$

有递推关系：

$$E(n,k)=E(n,k-1)+E(n-1,n-k)$$

之字形数Zigzag

> 一个 zigzag 置换（zigzag permutation）是一个1到n的排列c_1到c_n，使得任意一个元素c_i的大小都不介于c_i-1和c_i+1之间。
>
> 每个 zigzag 置换翻转过来仍旧为 zigzag 置换，可以两两配对，所以必然为偶数。

对于大于 1 的 n，记：

$$A_n=\frac{Z_n}{2}$$

定义初值：

$$A_0=A_1=1$$

这里的 A_n 称为 zigzag 数（Euler zigzag number，OEIS A000111），从 $n=0$ 开始有：

$$1,1,1,2,5,16,61,272,\cdots$$

对于 zigzag 置换的个数 Z_n（OEIS A001250），从 $n=0$ 开始有：

$$1,1,2,4,10,32,122,544,\cdots$$

有递推关系：

$$2A_{n+1}=\sum_{k=0}^{n}\binom{n}{k}A_kA_{n-k}$$

$$2(n+1)\frac{A_{n+1}}{(n+1)!}=\sum_{k=0}^{n}\frac{A_k}{k!}\frac{A_{n-k}}{(n-k)!}$$

当 n 为 0 时并不满足这个递推式，初值 A_0 和 A_1 都是 1。

恩特林格数与 zigzag 数的关系

根据恩特林格数的定义，恩特林格数 $E(n,k)$ 是首元素为 k 的 0 到 n 的交替置换个数。因此恩特林格数与 zigzag 数事实上有关系：

$$A_n=E(n,n)$$

将 A_n 称为「zigzag 数」也有原因：记 E_n 是欧拉数（Euler number），B_n 是伯努利数。

当 n 为奇数时，奇数项下标的 zigzag 数也称「正切数」T_n 或者「zag 数」。有关系：

$$A_n=-\frac{2i^{n+1}(2^{n+1}-1)B_{n+1}}{n+1}$$

当 n 为偶数时，偶数项下标的 zigzag 数也称「正割数」S_n 或者「zig 数」。有关系：

$$A_n=i^nE_n$$

或者写到一起：

$$\sec x+\tan x=A_0+A_1x+A_2\frac{x^2}{2!}+A_3\frac{x^3}{3!}+A_4\frac{x^4}{4!}+A_5\frac{x^5}{5!}+\cdots$$

构成 zigzag 数的生成函数。

欧拉数Eulerian

> ⚠注意区分欧拉数符号和排列数符号
>
> ⚠下文中的欧拉数特指 Eulerian number。注意与 Euler number，以及 Euler's number（指与欧拉相关的数学常数例如 γ 或 e ）作区分。

在计算组合中，**欧拉数**（Eulerian Number）是从 1 到 n 中正好满足 m 个元素大于前一个元素（具有 m 个「上升」的排列）条件的排列 **个数**。定义为：

$$A(n,m) = \left\langle {n \atop m-1} \right\rangle$$

$$A(n,m) = \begin{cases} 0 & m > n \text{ or } n = 0 \\ 1 & m = 0 \\ (n-m)\cdot A(n-1,m-1) + (m+1)\cdot A(n-1,m) & \text{otherwise} \end{cases}$$

```
int eulerianNum(int n, int m)
{
    if (m >= n || n == 0) return 0;
    if (m == 0) return 1;
    return (((n-m) * eulerianNum(n-1, m-1)) + ((m+1) * eulerianNum(n-1, m)));
}
```

分拆数Partition

分拆：将自然数 n 写成递降正整数和的表示。

$$n = r_1 + r_2 + \ldots + r_k \quad r_1 \geq r_2 \geq \ldots \geq r_k \geq 1$$

和式中每个正整数称为一个部分。

分拆数：p_n。自然数 n 的分拆方法数。

k 部分拆数

将 n 分成恰有 k 个部分的分拆，称为 k 部分拆数，记作 $p(n,k)$。

$$p(n,k) = p(n-1,k-1) + p(n-k,k)$$

```
p[0][0] = 1;
for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= k; ++j)
        if (i - j >= 0) /*p[i-j][j]所有部分大于1*/
            p[i][j] = (p[i - j][j] + p[i - 1][j - 1]) % mod;//p[i-1][j-1]至少有一个部分为1
```

Ferrers 图 ¶

Ferrers 图：将分拆的每个部分用点组成的行表示。每行点的个数为这个部分的大小。

根据分拆的定义，Ferrers 图中不同的行按照递减的次序排放。最长行在最上面。

互异分拆数

互异分拆数：pd_n。自然数 n 的各部分互不相同的分拆方法数。（Different）

同样地，定义互异 k 部分拆数 $pd(n,k)$，表示最大拆出 k 个部分的互异分拆，是这个方程的解数：

$$n=r_1+r_2+\ldots+r_k \quad r_1>r_2>\ldots>r_k\geq 1$$

$$pd(n,k)=pd(n-k,k-1)+pd(n-k,k)$$

插板法

问题一：现有 n 个 **完全相同** 的元素，要求将其分为 k 组，保证每组至少有一个元素，一共有多少种分法?

考虑拿 $k-1$ 块板子插入到 n 个元素两两形成的 $n-1$ 个空里面。

因为元素是完全相同的，所以答案就是 $\binom{n-1}{k-1}$。

本质是求 $x_1+x_2+\cdots+x_k=n$ 的正整数解的组数。

问题二：如果问题变化一下，每组允许为空呢?

显然此时没法直接插板了，因为有可能出现很多块板子插到一个空里面的情况，非常不好计算。

我们考虑创造条件转化成有限制的问题一，先借 k 个元素过来，在这 $n+k$ 个元素形成的 $n+k-1$ 个空里面插板，答案为

$$\binom{n+k-1}{k-1}=\binom{n+k-1}{n}$$

本质是求 $x_1+x_2+\cdots+x_k=n$ 的非负整数解的组数（即要求 $x_i\geq 0$）。

问题三：如果再扩展一步，要求对于第 i 组，至少要分到 $a_i, \sum a_i\leq n$ 个元素呢?

本质是求 $x_1+x_2+\cdots+x_k=n$ 的解的数目，其中 $x_i\geq a_i$。

类比无限制的情况，我们借 $\sum a_i$ 个元素过来，保证第 i 组至少能分到 a_i 个。也就是令

$$x_i'=x_i-a_i$$

得到新方程：

$$\begin{aligned}(x_1'+a_1)+(x_2'+a_2)+\cdots+(x_k'+a_k)&=n\\ x_1'+x_2'+\cdots+x_k'&=n-a_1-a_2-\cdots-a_k\\ x_1'+x_2'+\cdots+x_k'&=n-\sum a_i\end{aligned}$$

然后问题三就转化成了问题二，直接用插板法公式得到答案为

$$\binom{n-\sum a_i+k-1}{n-\sum a_i}$$

> 对于上界问题：求$x_1+x_2+\cdots+x_k=n$，其中$x_i<=a_i$解的数目，与容斥原理相关

多重集的组合数 1

设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1，n_2 个 a_2，…，n_k 个 a_k 组成的多重集。那么对于整数 $r(r<n_i,\forall i\in[1,k])$，从 S 中选择 r 个元素组成一个多重集的方案数就是 **多重集的组合数**。这个问题等价于 $x_1+x_2+\cdots+x_k=r$ 的非负整数解的数目，可以用插板法解决，答案为

$$\binom{r+k-1}{k-1}$$

多重集的组合数 2

考虑这个问题：设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k,\}$ 表示由 n_1 个 a_1，n_2 个 a_2，…，n_k 个 a_k 组成的多重集。那么对于正整数 r，从 S 中选择 r 个元素组成一个多重集的方案数。

这样就限制了每种元素的取的个数。同样的，我们可以把这个问题转化为带限制的线性方程求解：

$$\forall i\in[1,k],\ x_i\le n_i,\ \sum_{i=1}^{k}x_i=r$$

$$Ans=\sum_{p=0}^{k}(-1)^p\sum_{A}\binom{k+r-1-\sum_A n_{A_i}-p}{k-1}$$

其中 A 是充当枚举子集的作用，满足 $|A|=p,\ A_i<A_{i+1}$。

插空法

不相邻的排列

$1\sim n$ 这 n 个自然数中选 k 个，这 k 个数中任何两个数都不相邻的组合有 $\binom{n-k+1}{k}$ 种。

排列组合性质 | 二项式推论

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{m+n}{m}\quad(n\ge m)$$

$$\sum_{i=0}^{n}i\binom{n}{i}=n2^{n-1}$$

$$\sum_{i=0}^{n}i^2\binom{n}{i}=n(n+1)2^{n-2}$$

$$\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$$

$$\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k}$$

$$\sum_{l=0}^{n}\binom{l}{k}=\binom{n+1}{k+1}$$

$$\sum_{i=0}^{n}\binom{n-i}{i}=F_{n+1}$$

- 带权值二项式 $(q=1-p)$

$$\sum_{i=0}^{n}\binom{n}{i}p^i q^{n-i} i^k = \sum_{j=0}^{k}\begin{Bmatrix}k\\j\end{Bmatrix} n^{\underline{j}} p^j$$

容斥原理

定义

设 U 中元素有 n 种不同的属性，而第 i 种属性称为 P_i，拥有属性 P_i 的元素构成集合 S_i，那么

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots + (-1)^{m-1} \sum_{a_i<a_{i+1}} \left|\bigcap_{i=1}^{m} S_{a_i}\right| + \cdots + (-1)^{n-1} |S_1 \cap \cdots \cap S_n|$$

即

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{m=1}^{n} (-1)^{m-1} \sum_{a_i<a_{i+1}} \left|\bigcap_{i=1}^{m} S_{a_i}\right|$$

例子：m 盒不同小球，每盒 $a[i]$ 个相同小球，放入 n 个箱子，要求每个箱子至少有一个小球的方案数.

假设现在我们在求有x个盒子为空的情况，就相当于把$a[i]$个球放进了$n-x$个盒子当中，方案数$C(n-x+a[i]-1, n-x-1)$，根据乘法原理，有i个盒子为空对答案的贡献就应该是$\prod_{j=1}^{m} C(n-i+a[j]-1, n-i-1)$，又考虑到盒子是有标号的，我们选择不同的盒子为空，方案数也不同，于是就要在方案数基础上*一个$C(n,i)$，那么，有i个盒子为空的总贡献就应该是

$C(n,i) * \prod_{j=1}^{m} C(n-i+a[j]-1, n-i-1)$

已经到了这一步，相信大家就已经看出来接下来就是一个容斥了，那么最终的通式就应该是：

$\sum_{i=0}^{n-1} (-1)^i C(n,i) \prod_{j=1}^{m} C(n+a[j]-i-1, n-i-1)$

全排列

```
void Perm(int* br, int k, int m)/*br代表要进行全排列的数组，k~m代表这个数组中要进行全排列数字的范围*/
{
    if (k == m)
    {
        for (int i = 0; i <= m; ++i) cout << br[i] << ' ';
        cout << endl;
    }
    else
    {
        for (int j = k; j <= m; ++j)
        {
            swap(br[j], br[k]);
            Perm(br, k + 1, m);
            swap(br[j], br[k]);
        }
    }
```

```
}
//stl的函数next_permutation(a.begin(), a.end());也可以枚举
```

卢卡斯定理

p一般在 $1e5$ $1e6$ 左右

Lucas 定理主要用于大组合数取膜。

他的结论很简单：当且仅当 p 为质数时，$C_m^n \bmod p = C_{m \bmod p}^{n \bmod p} \times C_{m \div p}^{n \div p} \bmod p$ *(主要运用公式)

$C_n^m = C_{a_0}^{b_0} \cdot C_{a_1}^{b_1} \cdot C_{a_2}^{b_2} \cdots C_{a_k}^{b_k} \text{ (mod p)} = \prod_{i=0}^{k} C_{a_i}^{b_i} \text{ (mod p)}$

(mod p)表示只是在模p的条件下成立

其中$n = a_0 + a_1 p + a_2 p^2 + \cdots + a_k p^k$,$m = b_0 + b_1 p + b_2 p^2 + \cdots + b_k p^k$

```
//p为质数
ll lucas(ll n, ll m, ll p)
{
    if (m == 0) return 1; // C(n,m)也可以用C[n][m]预处理替代
    return lucas(n / p, m / p, p) * C(n % p, m % p, p) % p;
}
```

拓展卢卡斯定理

求解思想

由于p不是质数，那么我们考虑强行对其进行质因数分解：

$$p = \prod_{i=1}^{n} p_i^{a_i}$$

那么分解完后每一项$p_i^{a_i}$ 之间两两互质，只要我们能得出每组$C_n^m \mod p_i^{a_i}$ 的答案，就可以用中国剩余定理合并得到最后的答案。

```
// 缺少crt,qpow,inv,exgcd函数
// n!中把x因子去除
ll fac(ll n, ll x, ll P) // P = x ^ k;
{
    if (!n)
        return 1;
    ll s = 1;
    for (ll i = 1; i <= P; i++) if (i % x) s = s * i % P;
    s = qpow(s, n / P, P);
    for (ll i = n / P * P + 1; i <= n; i++) if (i % x) s = i % P * s % P;
    return s * fac(n / x, x, P) % P;
}
ll mullucas(ll n, ll m, ll x, ll P)
{
    int cnt = 0;
    for (ll i = n; i; i /= x) cnt += i / x;
    for (ll i = m; i; i /= x) cnt -= i / x;
    for (ll i = n - m; i; i /= x) cnt -= i / x;
    return qpow(x, cnt, P) % P * fac(n, x, P) % P * inv(fac(m, x, P), P) %
           P * inv(fac(n - m, x, P), P) % P;
}
ll exlucas(ll n, ll m, ll P)
```

```
{
    int cnt = 0;
    ll p[20], a[20];
    for (ll i = 2; i * i <= P; i++)
    {
        if (P % i == 0)
        {
            p[++cnt] = 1;
            while (P % i == 0) p[cnt] = p[cnt] * i, P /= i;
            a[cnt] = mullucas(n, m, i, p[cnt]);
        }
    }
    if (P > 1) p[++cnt] = P, a[cnt] = mullucas(n, m, P, P);
    return crt(cnt, a, p);
}
```

线性代数

矩阵的线性变换

> **矩阵的线性变换**

当一个矩阵和一个向量做乘法时，将产生一个新的向量（可以理解为：向量和矩阵做乘法描述了一个运动），矩阵描述了一个二维空间如何变换（旋转、拉伸等），向量相当于一个入参。

这与函数类似，当一个矩阵描述了二维空间逆时针旋转90°，那么任意一个向量与该矩阵做乘法时，都会得到一个逆时针旋转了90°的新向量。

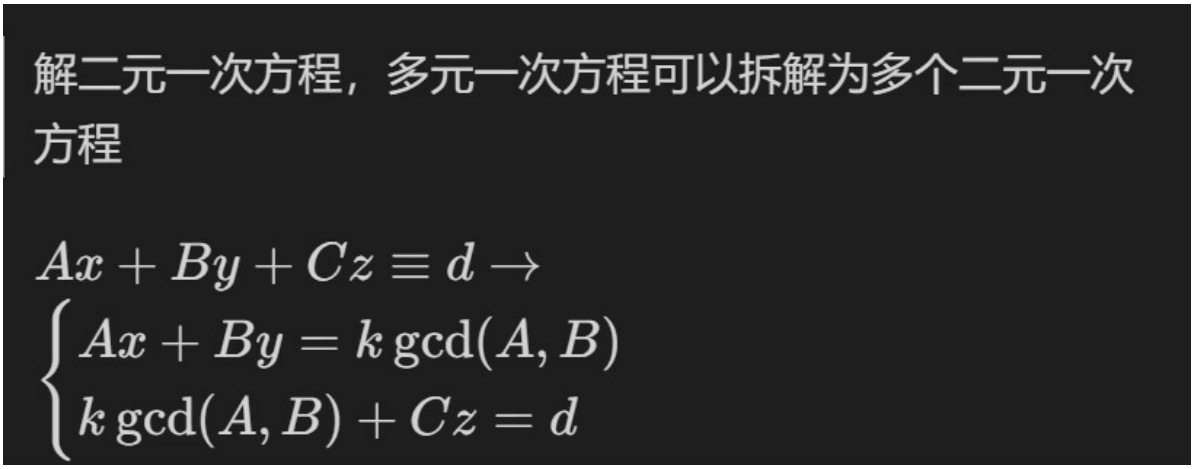
$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}\times\begin{pmatrix}x\\y\end{pmatrix}=x\begin{pmatrix}1\\3\end{pmatrix}+y\begin{pmatrix}2\\4\end{pmatrix}=\begin{pmatrix}1x+2y\\3x+4y\end{pmatrix}$$

向量的旋转指的是将一个已知向量旋转给定的弧度或角度后得到一个旋转后的新向量。矩阵描述了一个二维空间如何变换，当一个矩阵和一个向量做乘法时，将产生一个新的向量。而这个描述了空间如何变换的矩阵可以通过弧度计算得出。

至此，我们已经掌握了向量旋转的理论知识，一个完整的向量旋转矩阵计算如下所示：

$$\begin{pmatrix}cos\theta & -sin\theta\\sin\theta & cos\theta\end{pmatrix}\times\begin{pmatrix}x\\y\end{pmatrix}=x\begin{pmatrix}cos\theta\\sin\theta\end{pmatrix}+y\begin{pmatrix}-sin\theta\\cos\theta\end{pmatrix}=\begin{pmatrix}x*cos\theta+y*-sin\theta\\x*sin\theta+y*cos\theta\end{pmatrix}$$

解多元一次方程



高斯消元

对于一个线性方程组 $\begin{cases} a_1x+b_1y+c_1z=d_1 \\ a_2x+b_2y+c_2z=d_2 \\ a_3x+b_3y+c_3z=d_3 \end{cases}$，我们可以写出两个矩阵（系数矩阵 coefficient matrix 和增广矩阵 Augmented Matrix），如下图所示。

线性系统 $\begin{cases} a_1x+b_1y+c_1z=d_1 \\ a_2x+b_2y+c_2z=d_2 \\ a_3x+b_3y+c_3z=d_3 \end{cases}$ $\Rightarrow$ 系数矩阵 $\begin{bmatrix} a_1\,b_1\,c_1 \\ a_2\,b_2\,c_2 \\ a_3\,b_3\,c_3 \end{bmatrix}$ $\Leftrightarrow$ 增广矩阵 $\left[\begin{array}{ccc|c} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{array}\right]$

高斯消去步骤

高斯消去法的过程可以分为以下几步：

（一）、构造增广矩阵。即系数矩阵 A 加上常数向量 b，也就是 (A|b)。

（二）、通过以交换行、某行乘以非负常数和两行相加这三种初等变化将原系统转化为更简单的三角形式

（三）、得到简化的三角方阵组。

（四）、使用向后替换算法（Algorithm for Back Substitution）求解得。

布尔值版本

```
int gauss()  // 高斯消元，答案存于a[i][n]中，0 <= i < n
{
    int c, r;
    for (c = 0, r = 0; c < n; c++)
    {
        int t = r;
        for (int i = r; i < n; i++) if (a[i][c]) t = i; // 找非零行
        if (!a[t][c]) continue;
        for (int i = c; i <= n; i++) swap(a[r][i], a[t][i]);  // 将非零行换到最顶端
        for (int i = r + 1; i < n; i++)  // 用当前行将下面所有的列消成0
            if (a[i][c])
                for (int j = n; j >= c; j--)
                    a[i][j] ^= a[r][j];
        r++;
    }
    if (r < n)
    {
        for (int i = r; i < n; i++) if (a[i][n]) return 2;  // 无解
        return 1;  // 有多组解
    }
    for (int i = n - 1; i >= 0; i--)
        for (int j = i + 1; j < n; j++)
            a[i][n] ^= a[i][j] * a[j][n];
    return 0;  // 有唯一解
}
```

浮点数版本

```
int gauss()  // 高斯消元，答案存于a[i][n]中，0 <= i < n
{
    int c, r;
    for (c = 0, r = 0; c < n; c++)
    {
        int t = r;
        for (int i = r; i < n; i++)  // 找绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c])) t = i;
        if (fabs(a[t][c]) < eps) continue;
```

```
        for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]);  // 将绝对值最大的行换到最顶端
        for (int i = n; i >= c; i--) a[r][i] /= a[r][c];  // 将当前行的首位变成1
        for (int i = r + 1; i < n; i++)  // 用当前行将下面所有的列消成0
            if (fabs(a[i][c]) > eps)
                for (int j = n; j >= c; j--)
                    a[i][j] -= a[r][j] * a[i][c];
        r++;
    }
    if (r < n)
    {
        for (int i = r; i < n; i++) if (fabs(a[i][n]) > eps) return 2; // 无解
        return 1; // 有无穷多组解
    }
    for (int i = n - 1; i >= 0; i--)
        for (int j = i + 1; j < n; j++)
            a[i][n] -= a[i][j] * a[j][n];
    return 0; // 有唯一解
}
```

矩阵操作运算

包含矩阵快速幂

```
struct mat
{
    ll a[N+1][N+1];
    mat() { memset(a, 0, sizeof a); }
    mat operator-(const mat &T) const
    {
        mat res;
        for (int i = 1; i <= N; ++i)
            for (int j = 1; j <= N; ++j)
                res.a[i][j] = (a[i][j] - T.a[i][j]) % mod;
        return res;
    }
    mat operator+(const mat &T) const
    {
        mat res;
        for (int i = 1; i <= N; ++i)
            for (int j = 1; j <= N; ++j)
                res.a[i][j] = (a[i][j] + T.a[i][j]) % mod;
        return res;
    }
    mat operator*(const mat &T) const
    {
        mat res;
        int r;
        for (int i = 1; i <= N; ++i)
            for (int k = 1; k <= N; ++k)
            {
                r = a[i][k];
                for (int j = 1; j <= N; ++j)
                    res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= mod;
            }
        return res;
    }
```

```
    mat operator^(ll x) const
    {
        mat res, bas;
        for (int i = 1; i <= N; ++i) res.a[i][i] = 1;
        for (int i = 1; i <= N; ++i)
            for (int j = 1; j <= N; ++j)
                bas.a[i][j] = a[i][j] % mod;
        while (x)
        {
            if (x & 1) res = res * bas;
            bas = bas * bas;
            x >>= 1;
        }
        return res;
    }
};
```

矩阵维护[斐波那契数列]问题

$$\begin{bmatrix} \text{feibo}_n & \text{feibo}_{n+1} \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \text{feibo}_{n+1} & \text{feibo}_{n+2} \end{bmatrix}$$

$$\begin{bmatrix} \text{feibo}_1 & \text{feibo}_2 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} = \begin{bmatrix} \text{feibo}_n & \text{feibo}_{n+1} \end{bmatrix}$$

多项式

多项式乘法O(n^2)(懒得写)

快速傅里叶变换

快速傅立叶变换（Fast Fourier Transform，FFT）

傅里叶变换（Fourier Transform）是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，傅里叶变换用正弦波作为信号的成分。

设 $f(t)$ 是关于时间 t 的函数，则傅里叶变换可以检测频率 ω 的周期在 $f(t)$ 出现的程度：

$$F(\omega) = \mathbb{F}[f(t)] = \int_{-\infty}^{\infty} f(t) \mathrm{e}^{-\mathrm{i}\omega t} dt$$

它的逆变换是

$$f(t) = \mathbb{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \mathrm{e}^{\mathrm{i}\omega t} d\omega$$

逆变换的形式与正变换非常类似，分母 2π 恰好是指数函数的周期。

傅里叶变换相当于将时域的函数与周期为 2π 的复指数函数进行连续的内积。逆变换仍旧为一个内积。

傅里叶变换有相应的卷积定理，可以将时域的卷积转化为频域的乘积，也可以将频域的卷积转化为时域的乘积。

FFT递归版本（含complex的std类型）
时间复杂度(O(nlogn))
递归本身运行较慢

因为是单位复根，所以说我们需要令 n 项式的高位补为零，使得 $n=2^k,k\in \mathbf{N}^*$。

```
int lim = 1;
while (lim <= n + m) lim <<= 1;
```

逆变换时需要对结果 $a[i]$ 进行 $(int)(a[i]/lim+0.5)$ 强制取整

```
typedef std::complex<double> Comp; // STL complex
const Comp I(0, 1); // i
const int MAX_N = 1 << 20;
Comp tmp[MAX_N];
// rev=1,DFT; rev=-1,IDFT
void DFT(Comp *f, int n, int rev)
{
    if (n == 1) return;
    for (int i = 0; i < n; ++i) tmp[i] = f[i];
    // 偶数放左边，奇数放右边
    for (int i = 0; i < n; ++i)
        if (i & 1)
            f[n / 2 + i / 2] = tmp[i];
        else
            f[i / 2] = tmp[i];
    Comp *g = f, *h = f + n / 2;
    // 递归 DFT
    DFT(g, n / 2, rev), DFT(h, n / 2, rev);
    // cur 是当前单位复根，对于 k = 0 而言，它对应的单位复根 omega^0_n = 1。
    // step 是两个单位复根的差，即满足 omega^k_n = step*omega^{k-1}*n，
    // 定义等价于 exp(I*(2*M_PI/n*rev))
    Comp cur(1, 0), step(cos(2 * M_PI / n), sin(2 * M_PI * rev / n));
    for (int k = 0; k < n / 2; ++k)
    { // F(omega^k_n) = G(omega^k*{n/2}) + omega^k*n\*H(omega^k*{n/2})
        tmp[k] = g[k] + cur * h[k];
        // F(omega^{k+n/2}*n) = G(omega^k*{n/2}) - omega^k_n*H(omega^k\_{n/2})
        tmp[k + n / 2] = g[k] - cur * h[k];
        cur *= step;
    }
    for (int i = 0; i < n; ++i) f[i] = tmp[i];
}
```

非递归解决办法（节省常量时间）

自构建复数结构，防止弱智 $std::complex$ 犯病

```
ll c[N];
struct comp
{
    double real, imag;
    comp(double X = 0, double Y = 0) { real = X; imag = Y; }
} a[N], b[N];
inline comp operator+(comp a, comp b) { return comp(a.real + b.real, a.imag +
b.imag); }
```

```
inline comp operator-(comp a, comp b) { return comp(a.real - b.real, a.imag -
b.imag); }
inline comp operator*(comp a, comp b) { return comp(a.real * b.real - a.imag *
b.imag, a.real * b.imag + a.imag * b.real); }
void FFT(comp *a, ll lim, ll op)
{
    for (ll i = 0; i < lim; i++) if (i < c[i]) swap(a[i], a[c[i]]);
    for (ll mid = 1; mid < lim; mid <<= 1)
    {
        comp W(cos(pi / mid), op * sin(pi / mid));
        for (ll r = mid << 1, j = 0; j < lim; j += r)
        {
            comp w(1, 0);
            for (ll l = 0; l < mid; l++, w = w * W)
            {
                comp x = a[j + l], y = w * a[j + mid + l];
                a[j + l] = x + y;
                a[j + mid + l] = x - y;
            }
        }
    }
}
```

计算几何

基本公式

正余弦定理

正弦定理 ¶

在三角形 $\triangle ABC$ 中，若角 A, B, C 所对边分别为 a, b, c，则有：

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$$

其中，R 为 $\triangle ABC$ 的外接圆半径。

余弦定理

在三角形 $\triangle ABC$ 中，若角 A, B, C 所对边分别为 a, b, c，则有：

$$a^2 = b^2 + c^2 - 2bc\cos A$$
$$b^2 = a^2 + c^2 - 2ac\cos B$$
$$c^2 = a^2 + b^2 - 2ab\cos C$$

椭圆

$$S = \pi * a * b$$
$$C = \pi * \sqrt{2 * (a^2 + b^2)} (\text{低精度})$$
$$C = \pi * (a + b) * (1 + \frac{3\lambda^2}{10 + \sqrt{4 - 3\lambda^2}}); \ (\lambda = \frac{a - b}{a + b}) (\text{高精度})$$

三点定圆

过 xOy 平面不共线三点 $M_i(x_i,y_i)$ ，$i=1,2,3$ 的圆 $\odot P$ 曲线方程为：

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ {x_1}^2+{y_1}^2 & x_1 & y_1 & 1 \\ {x_2}^2+{y_2}^2 & x_2 & y_2 & 1 \\ {x_3}^2+{y_3}^2 & x_3 & y_3 & 1 \end{vmatrix} = 0$$

而圆心 P 的坐标（按圆曲线方程中行列式第一行展开可得）为：

$$x_P = \frac{\begin{vmatrix} \frac{{x_1}^2+{y_1}^2}{2} & y_1 & 1 \\ \frac{{x_2}^2+{y_2}^2}{2} & y_2 & 1 \\ \frac{{x_3}^2+{y_3}^2}{2} & y_3 & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}} \qquad y_P = \frac{\begin{vmatrix} x_1 & \frac{{x_1}^2+{y_1}^2}{2} & 1 \\ x_2 & \frac{{x_2}^2+{y_2}^2}{2} & 1 \\ x_3 & \frac{{x_3}^2+{y_3}^2}{2} & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}}$$

向量的旋转

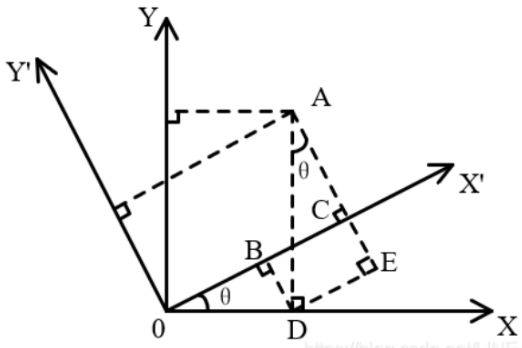
坐标轴逆时针旋转，等价于向量顺时针旋转

二维坐标系旋转与向量（坐标）旋转

坐标系的旋转和向量的旋转在工程应用过程中经常会遇到，在这里对二维坐标系的旋转和向量旋转做一个简单的推导，方便大家的理解。

坐标系旋转

一个平面坐标系逆时针旋转一个角度后得到另一个坐标系，则同一个点在这两个坐标系之间的几何关系如下：



由上图可得：

$$\begin{aligned} x' &= OB + BC \\ &= OD\cos\theta + AD\sin\theta \\ &= x\cos\theta + y\sin\theta \end{aligned} \qquad \begin{aligned} y' &= AE - CE \\ &= AD\cos\theta - OD\sin\theta \\ &= y\cos\theta - x\sin\theta \end{aligned}$$

向量顺时针旋转的结果

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

向量逆时针旋转的结果

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

两个圆的公切线

```
const double eps=1e-9;
const double pi=acos(-1.0);
class Point;
typedef Point Vec;
//三态函数比较;精度问题
int dcmp(double x){
    if(fabs(x)<eps) return 0;
    return x<0?-1:1;
}
struct Point{
    double x,y;
    Point(double _x=0,double _y=0):x(_x),y(_y){}
    //向量与常数 注意常数要放在后面
    Vec operator*(double p) {return Vec(x*p,y*p);}
    Vec operator/(double p) {return Vec(x/p,y/p);}
    Vec operator-(Vec obj)  {return Vec(x-obj.x,y-obj.y);}
    Vec operator+(Vec obj)  {return Vec(x+obj.x,y+obj.y);} //点积
    double operator*(Vec obj) {return x*obj.x+y*obj.y;} //叉积
    double operator^(Vec obj) {return x*obj.y-y*obj.x;}
    //两个向量的夹角 A*B=|A|*|B|*cos(th)
    double Angle(Vec B)  {return acos((*this)*B/(*this).len()/B.len());}
    //两条向量平行四边形的面积
    double Area(Vec B)   {return fabs((*this)^B);}
    //向量旋转
    //旋转公式
    //  Nx     (cos  -sin) x
    //  Ny     (sin   cos) y
    Vec Rotate(double rad)  {return Vec(x*cos(rad)-
y*sin(rad),x*sin(rad)+y*cos(rad));}
    //返回向量的法向量，即旋转pi/2
    Vec Normal()  {return Vec(-y,x);}
    //返回向量的长度,或者点距离原点的距离
    double len()  {return hypot(x,y);}
    double len2() {return x*x+y*y;} //返回两点之间的距离
    double dis(Point obj)   {return hypot(x-obj.x,y-obj.y);} //hypot 给定直角三角形
的两条直角边，返回斜边边长
    //向量的极角 atan2(y,x)
    bool operator==(Point obj) {return dcmp(x-obj.x)==0&&dcmp(y-obj.y)==0;}
    bool operator<(Point obj)  {return x<obj.x||(x==obj.x&&y<obj.y);}
};
struct Circle{
    Point c; double r;
    Circle(Point c,double r):c(c),r(r){}
    Point getpoint(double a){return Point(c.x+cos(a)*r,c.y+sin(a)*r);}
```

```
};
/* 求圆的公切线 */
int getTan(Circle A,Circle B,Point*  va,Point* vb){
    int cnt=0;
    if(A.r<B.r){swap(A,B);swap(va,vb);}
    double d=(A.c-B.c).len();
    double rdif=A.r-B.r,rsum=A.r+B.r;
    //内含，没有公切线
    if(dcmp(d-rdif)<0)return 0;
    //内切，有一条公切线
    double base=atan2(B.c.y-A.c.y,B.c.x-A.c.x);
    if(dcmp(d)==0&&dcmp(A.r-B.r)==0)return -1;
    if(dcmp(d-rdif)==0){
        va[cnt]=A.getpoint(base);vb[cnt]=B.getpoint(base);cnt++;
        return cnt;
    }
    //一定有两条外公切线
    double th=acos((A.r-B.r)/d);
    va[cnt]=A.getpoint(base+th);vb[cnt]=B.getpoint(base+th);cnt++;
    va[cnt]=A.getpoint(base-th);vb[cnt]=B.getpoint(base-th);cnt++;
    //可能有一条公切线
    if(dcmp(d-rsum)==0){
        va[cnt]=A.getpoint(base);vb[cnt]=B.getpoint(base+pi);cnt++;
    }
    else if(dcmp(d-rsum)>0)
    {
        double th2=acos((A.r+B.r)/d);
        va[cnt]=A.getpoint(base+th2);vb[cnt]=B.getpoint(base+th2+pi);cnt++;
        va[cnt]=A.getpoint(base-th2);vb[cnt]=B.getpoint(base-th2+pi);cnt++;
    }
    return cnt;
}
```

距离

欧氏距离

$\vec{A}(x_{11},x_{12},\cdots,x_{1n})$，$\vec{B}(x_{21},x_{22},\cdots,x_{2n})$，有

$$\begin{aligned}\|\overrightarrow{AB}\| &= \sqrt{(x_{11}-x_{21})^2+(x_{12}-x_{22})^2+\cdots+(x_{1n}-x_{2n})^2} \\ &= \sqrt{\sum_{i=1}^{n}(x_{1i}-x_{2i})^2}\end{aligned}$$

欧氏距离虽然很有用，但也有明显的缺点。两个整点计算其欧氏距离时，往往答案是浮点型，会存在一定误差。

曼哈顿距离Manhattan

定义

在二维空间内，两个点之间的曼哈顿距离（Manhattan distance）为它们横坐标之差的绝对值与纵坐标之差的绝对值之和。设点 $A(x_1,y_1),B(x_2,y_2)$，则 A,B 之间的曼哈顿距离用公式可以表示为：

$$d(A,B)=|x_1-x_2|+|y_1-y_2|$$

切比雪夫距离Chebyshev

定义

切比雪夫距离（Chebyshev distance）是向量空间中的一种度量，二个点之间的距离定义为其各坐标数值差的最大值。[1]

在二维空间内，两个点之间的切比雪夫距离为它们横坐标之差的绝对值与纵坐标之差的绝对值的最大值。设点 $A(x_1,y_1),B(x_2,y_2)$，则 A,B 之间的切比雪夫距离用公式可以表示为：

$$d(A,B)=\max(|x_1-x_2|,|y_1-y_2|)$$

切比雪夫和曼哈顿的转化

结论

- 曼哈顿坐标系是通过切比雪夫坐标系旋转 45° 后，再缩小到原来的一半得到的。
- 将一个点 (x,y) 的坐标变为 $(x+y,x-y)$ 后，原坐标系中的曼哈顿距离等于新坐标系中的切比雪夫距离。
- 将一个点 (x,y) 的坐标变为 $(\frac{x+y}{2},\frac{x-y}{2})$ 后，原坐标系中的切比雪夫距离等于新坐标系中的曼哈顿距离。

Lm距离

L_m 距离

一般地，我们定义平面上两点 $A(x_1,y_1)$，$B(x_2,y_2)$ 之间的 L_m 距离为

$d(L_m)=(|x_1-x_2|^m+|y_1-y_2|^m)^{\frac{1}{m}}$

特殊的，L_2 距离就是欧几里得距离，L_1 距离就是曼哈顿距离。

Pick定理

可用于知面积反推图形内点的数量

Pick 定理：给定顶点均为整点的简单多边形，皮克定理说明了其面积 A 和内部格点数目 i、边上格点数目 b 的关系：$A = i + \frac{b}{2} - 1$。

它有以下推广：

- 取格点的组成图形的面积为一单位。在平行四边形格点，皮克定理依然成立。套用于任意三角形格点，皮克定理则是 $A = 2 \times i + b - 2$。
- 对于非简单的多边形 P，皮克定理 $A = i + \frac{b}{2} - \chi(P)$，其中 $\chi(P)$ 表示 P 的 **欧拉特征数**。
- 高维推广：Ehrhart 多项式
- 皮克定理和 **欧拉公式**（$V - E + F = 2$）等价。

凸包

静态二维凸包

现已加入计算几何操作集

```
using typedef pair<double, double> PDD;
#define define x first
#define define y second
int stk[N], top;
PDD q[N];
bool used[N];
PDD operator- (PDD a, PDD b)  // 向量减法
{
    return {a.x - b.x, a.y - b.y};
}
double operator* (PDD a, PDD b)  // 叉积、外积
{
    return a.x * b.y - a.y * b.x;
}
double operator& (PDD a, PDD b)  // 内积、点积
{
    return a.x * b.x + a.y * b.y;
}
double area(PDD a, PDD b, PDD c)  // 以a, b, c为顶点的有向三角形面积
{
    return (b - a) * (c - a);
}
double get_len(PDD a)  // 求向量长度
{
    return sqrt(a & a);
}
double get_dist(PDD a, PDD b)  // 求两个点之间的距离
{
    return get_len(b - a);
}
void andrew()  // Andrew算法，凸包节点编号逆时针存于stk中，下标从0开始
{
    sort(q, q + n);
    for (int i = 0; i < n; i ++ )
    {
        while (top >= 2 && area(q[stk[top - 2]], q[stk[top - 1]], q[i]) <= 0)
```

```
        {
            if (area(q[stk[top - 2]], q[stk[top - 1]], q[i]) < 0)
                used[stk[ -- top]] = false;
            else
                top -- ;
        }
        stk[top ++ ] = i;
        used[i] = true;
    }
    used[0] = false;
    for (int i = n - 1; i >= 0; i -- )
    {
        if (used[i]) continue;
        while (top >= 2 && area(q[stk[top - 2]], q[stk[top - 1]], q[i]) <= 0)
            top -- ;
        stk[top ++ ] = i;
    }
    top -- ;  // 起点重复添加了一次，将其去掉
}
```

动态加点二维凸包

```
struct Point
{
    double x, y;
    Point(double _x = 0, double _y = 0) { x = _x; y = _y; }
    friend Point operator+(const Point &a, const Point &b)
    {
        return Point(a.x + b.x, a.y + b.y);
    }
    friend Point operator-(const Point &a, const Point &b)
    {
        return Point(a.x - b.x, a.y - b.y);
    }
    friend double operator^(const Point &a, const Point &b)
    {
        return a.x * b.y - a.y * b.x;
    }
    friend bool operator==(const Point &a, const Point &b)
    {
        return fabs(a.x - b.x) < eps && fabs(a.y - b.y) < eps;
    }
};
struct V
{
    Point start, end;
    V(Point _start = Point(0, 0), Point _end = Point(0, 0))
    {
        start = _start;
        end = _end;
    }
};
Point Basic, str[N];
set<Point> Set;
int n;
double Distance(Point a, Point b)
{
```

```
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
bool operator<(Point a, Point b)
{
    a = a - Basic;
    b = b - Basic;
    double Ang1 = atan2(a.y, a.x), Ang2 = atan2(b.y, b.x);
    double Len1 = Distance(a, Point(0.0, 0.0)), Len2 = Distance(b, Point(0.0,
0.0));
    if (fabs(Ang1 - Ang2) < eps) return Len1 < Len2;
    return Ang1 < Ang2;
}
set<Point>::iterator Pre(set<Point>::iterator it)
{
    if (it == Set.begin()) it = Set.end();
    return --it;
}
set<Point>::iterator Nxt(set<Point>::iterator it)
{
    ++it;
    return it == Set.end() ? Set.begin() : it;
}
int Query(Point p) //查询点p是否在凸包内(1为在,0为不在)
{
    set<Point>::iterator it = Set.lower_bound(p);
    if (it == Set.end()) it = Set.begin();
    return ((p - *(Pre(it))) ^ (*(it) - *(Pre(it)))) < eps;
}
void Insert(Point p)
{
    if (Query(p)) return;
    Set.insert(p);
    set<Point>::iterator it = Nxt(Set.find(p));
    while (Set.size() > 3 && ((p - *(Nxt(it))) ^ (*(it) - *(Nxt(it)))) < eps)
    {
        Set.erase(it);
        it = Nxt(Set.find(p));
    }
    it = Pre(Set.find(p));
    while (Set.size() > 3 && ((p - *(it)) ^ (*(it) - *(Pre(it)))) > -eps)
    {
        Set.erase(it);
        it = Pre(Set.find(p));
    }
}
void Build()
{
    for (int i = 1; i <= 3; ++i) Basic = Basic + str[i];
    Basic.x /= 3; Basic.y /= 3;
    for (int i = 1; i <= 3; ++i) Set.insert(str[i]);
    for (int i = 4; i <= n; ++i) Set.Insert(str[i]);
}
```

动态加边二维凸包

```
const ll is_query = -(1ll << 62);
struct Line
{
    ll m, b;
    mutable function<const Line *()> succ;
    bool operator<(const Line &rhs) const
    {
        if (rhs.b != is_query) return m < rhs.m;
        const Line *s = succ();
        if (!s) return false;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line>
{
    bool bad(iterator y)
    {
        auto z = next(y);
        if (y == begin())
        {
            if (z == end()) return false;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (x->b - y->b) * (z->m - y->m) >= (y->b - z->b) * (y->m - x->m);
    }
    void insert_line(ll m, ll b)
    {
        auto y = insert({m, b});
        y->succ = [=] { return next(y) == end() ? nullptr : &*next(y); };
        if (bad(y))
        {
            erase(y);
            return;
        }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(ll x)
    {
        auto l = *lower_bound((Line){x, is_query});
        return l.m * x + l.b;
    }
};
```

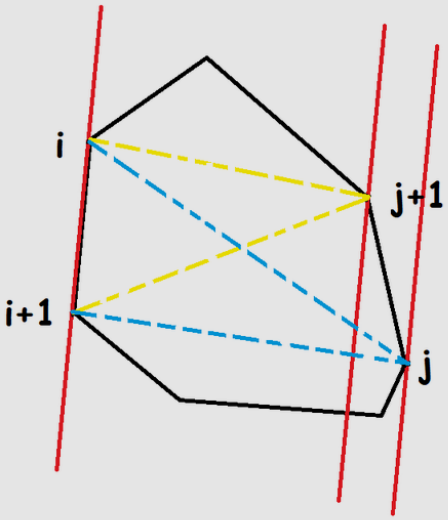
三维凸包

```
int n, cnt, vis[N][N];
double ans;
double Rand() { return rand() / (double)RAND_MAX; }
double reps() { return (Rand() - 0.5) * eps; }
struct Node
```

```
{
    double x, y, z;
    void shake()
    {
        x += reps();
        y += reps();
        z += reps();
    }
    double len() { return sqrt(x * x + y * y + z * z); }
    Node operator-(Node A) { return {x - A.x, y - A.y, z - A.z}; }
    Node operator*(Node A)
    {
        return {y * A.z - z * A.y, z * A.x - x * A.z, x * A.y - y * A.x};
    }
    double operator&(Node A) { return x * A.x + y * A.y + z * A.z; }
} A[N];
struct Face
{
    int v[3];
    Node Normal() { return (A[v[1]] - A[v[0]]) * (A[v[2]] - A[v[0]]); }
    double area() { return Normal().len() / 2.0; }
} f[N], C[N];
int see(Face a, Node b) { return ((b - A[a.v[0]]) & a.Normal()) > 0; }
void Convex_3D()
{
    f[++cnt] = {1, 2, 3}; f[++cnt] = {3, 2, 1};
    for (int i = 4, cc = 0; i <= n; i++)
    {
        for (int j = 1, v; j <= cnt; j++)
        {
            if (!(v = see(f[j], A[i]))) C[++cc] = f[j];
            for (int k = 0; k < 3; k++) vis[f[j].v[k]][f[j].v[(k + 1) % 3]] = v;
        }
        for (int j = 1; j <= cnt; j++)
            for (int k = 0; k < 3; k++)
            {
                int x = f[j].v[k], y = f[j].v[(k + 1) % 3];
                if (vis[x][y] && !vis[y][x]) C[++cc] = {x, y, i};
            }
        for (int j = 1; j <= cc; j++) f[j] = C[j];
        cnt = cc;
        cc = 0;
    }
}
int input()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> A[i].x >> A[i].y >> A[i].z, A[i].shake();
    Convex_3D();
    for (int i = 1; i <= cnt; i++) ans += f[i].area();
    printf("%.3f\n", ans); //凸包面积
    return 0;
}
```

旋转卡壳

旋转卡壳（Rotating Calipers，也称「旋转卡尺」）算法，在凸包算法的基础上，通过枚举凸包上某一条边的同时维护其他需要的点，能够在线性时间内求解如凸包直径、最小矩形覆盖等和凸包性质相关的问题。



枚举过程中，对于每条边，都检查 $j+1$ 和边 $(i,i+1)$ 的距离是不是比 j 更大，如果是就将 j 加一，否则说明 j 是此边的最优点。判断点到边的距离大小时可以用叉积分别算出两个三角形的面积（如图，黄、蓝两个同底三角形的面积）并直接比较。

```
struct Point
{
    double x, y;
    Point() {}
    Point(double a, double b) { x = a, y = b; }
    friend Point operator+(const Point &a, const Point &b) { return Point(a.x +
b.x, a.y + b.y); }
    friend Point operator-(const Point &a, const Point &b) { return Point(a.x -
b.x, a.y - b.y); }
    friend double operator^(Point a, Point b) { return a.x * b.y - a.y * b.x; }
    double distance(Point m) { return sqrt((m.x - x) * (m.x - x) + (m.y - y) *
(m.y - y)); }
};
double RotateCalipers(vector<Point> ch) //需要凸包化
{
    double ans = 0;
    int q = 1, n = ch.size();
    ch.pb(ch[0]);
    for (int i = 0; i < n; ++i)
    {
        while(((ch[q]-ch[i+1])^(ch[i]-ch[i+1])) < ((ch[q+1]-ch[i+1])^(ch[i]-
ch[i+1])))
            q = (q + 1) % n;
        ans = max(ans, max(ch[q].distance(ch[i]), ch[q + 1].distance(ch[i +
1])));
    }
    return ans;
}
```

半平面交

它可以理解为向量集中每一个向量的右侧的交，或者是下面方程组的解。

$$\begin{cases} A_1x + B_1y + C \geq 0 \\ A_2x + B_2y + C \geq 0 \\ \cdots \end{cases}$$

多边形的核

如果一个点集中的点与多边形上任意一点的连线与多边形没有其他交点，那么这个点集被称为多边形的核。

把多边形的每条边看成是首尾相连的向量，那么这些向量在多边形内部方向的半平面交就是多边形的核。

排序增量法求半平面交--算法步骤：

Step 1：将所有的半平面按照极角排序，排序过程还要将平行的半平面去重。

Step 2：使用一个双端队列deque，加入极角最小的半平面。

Step 3：扫描过程每次考虑一个新的半平面：

A：while deque顶端两个半平面的交点在当前半平面外：删除deque顶端的半平面。

B：while deque底部两个半平面的交点在当前半平面外：删除deque底部的半平面。

C：将当前半平面加入deque顶端。

Step 4：删除deque两端延伸出的对于半平面：

A：while deque顶端两个半平面的交点在底部半平面外：删除deque顶端的半平面。

B：while deque底部两个半平面的交点在顶部半平面外：删除deque底部的半平面。

Step 5：按顺序求取deque中下那个林半平面的交点，得到n个半平面交出的凸多边形。

板子1

```
using typedef pair<double, double> PDD;
#define define x first
#define define y second
struct Line  // 直线
{
    PDD st, ed;  // 直线上的两个点
}line[N];
int q[N];  // 双端队列
int sign(double x)  // 符号函数
{
    if (fabs(x) < eps) return 0;  // x为0，则返回0
    if (x < 0) return -1;  // x为负数，则返回-1
    return 1;  // x为正数，则返回1
}
int dcmp(double x, double y)  // 比较两数大小
{
    if (fabs(x - y) < eps) return 0;  // x == y, 返回0
    if (x < y) return -1;  // x < y, 返回-1
```

```
    return 1;  // x > y, 返回1
}
PDD operator+ (PDD a, PDD b)  // 向量加法
{
    return {a.x + b.x, a.y + b.y};
}
PDD operator-(PDD a, PDD b)  // 向量减法
{
    return {a.x - b.x, a.y - b.y};
}
double operator* (PDD a, PDD b)  // 外积、叉积
{
    return a.x * b.y - a.y * b.x;
}
PDD operator* (PDD a, double t)  // 向量数乘
{
    return {a.x * t, a.y * t};
}
double area(PDD a, PDD b, PDD c)  // 以a, b, c为顶点的有向三角形面积
{
    return (b - a) * (c - a);
}
PDD get_line_intersection(PDD p, PDD v, PDD q, PDD w) //两直线交点p + vt,q + wt
{
    auto u = p - q;
    auto t = w * u / (v * w);
    return p + v * t;
}
PDD get_line_intersection(Line a, Line b)  // 求两直线交点
{
    return get_line_intersection(a.st, a.ed - a.st, b.st, b.ed - b.st);
}
bool on_right(Line& a, Line& b, Line& c) // bc的交点是否在a的右侧
{
    auto o = get_line_intersection(b, c);
    return sign(area(a.st, a.ed, o)) <= 0;
}
double get_angle(const Line& a)  // 求直线的极角大小
{
    return atan2(a.ed.y - a.st.y, a.ed.x - a.st.x);
}
bool cmp(const Line& a, const Line& b)  // 将所有直线按极角排序
{
    double A = get_angle(a), B = get_angle(b);
    if (!dcmp(A, B)) return area(a.st, a.ed, b.ed) < 0;
    return A < B;
}
void half_plane_intersection()  // 半平面交，交集的边逆时针顺序存于q[]中
{
    sort(line, line + cnt, cmp);
    int hh = 0, tt = -1;
    for (int i = 0; i < cnt; i++)
    {
        if (i && !dcmp(get_angle(line[i]), get_angle(line[i - 1]))) continue;
        while (hh + 1 <= tt && on_right(line[i], line[q[tt - 1]], line[q[tt]]))
tt--;
        while (hh + 1 <= tt && on_right(line[i], line[q[hh]], line[q[hh + 1]]))
hh++;
```

```
        q[++tt] = i;
    }
    while (hh + 1 <= tt && on_right(line[q[hh]], line[q[tt - 1]], line[q[tt]]))
tt--;
    while (hh + 1 <= tt && on_right(line[q[tt]], line[q[hh]], line[q[hh + 1]]))
hh++;
    q[++tt] = q[hh];
    // 交集的边逆时针顺序存于q[]中
    // TODO: 求出半平面交后，根据题目要求求答案
}
```

板子2

```
//板子使用时记得初始化向量数组，包括(Point start, end; double ang;)
struct Point
{ // 点的表示
    double x, y;
    Point(double _x = 0, double _y = 0) { x = _x; y = _y; }
    friend Point operator+(const Point &a, const Point &b) { return Point(a.x +
b.x, a.y + b.y); }
    friend Point operator-(const Point &a, const Point &b) { return Point(a.x -
b.x, a.y - b.y); }
} poi[N], convex[N]; //poi原多边形点集，convex多边形核点集
struct V // 向量的表示
{
    Point start, end;
    double ang; // 角度[-180,180]
    V(Point _start = Point(0, 0), Point _end = Point(0, 0))
    {
        start = _start;
        end = _end;
        ang = atan2(end.y - start.y, end.x - start.x);
    }
} l[N], st[N];
int n, ccnt;
double DotMul(V a, V b)// 点积
{
    a.end = a.end - a.start;
    b.end = b.end - b.start;
    return a.end.x * b.end.x + a.end.y * b.end.y;
}
double CroMul(V a, V b)// 叉积 a×b
{
    a.end = a.end - a.start;
    b.end = b.end - b.start;
    return a.end.x * b.end.y - b.end.x * a.end.y;
}
int IsLineInter(V l1, V l2)// 相交
{
    if (max(l1.start.x, l1.end.x) >= min(l2.start.x, l2.end.x) &&
        max(l2.start.x, l2.end.x) >= min(l1.start.x, l1.end.x) &&
        max(l1.start.y, l1.end.y) >= min(l2.start.y, l2.end.y) &&
        max(l2.start.y, l2.end.y) >= min(l1.start.y, l1.end.y))
        if (CroMul(l2, V(l2.start, l1.start)) * CroMul(l2, V(l2.start, l1.end))
<= 0 &&
            CroMul(l1, V(l1.start, l2.start)) * CroMul(l1, V(l1.start, l2.end))
<= 0)
```

```
            return 1;
    return 0;
}
Point LineInterDot(V l1, V l2)// 交点
{
    Point p;
    double S1 = CroMul(V(l1.start, l2.end), V(l1.start, l2.start));
    double S2 = CroMul(V(l1.end, l2.start), V(l1.end, l2.end));
    p.x = (l1.start.x * S2 + l1.end.x * S1) / (S1 + S2);
    p.y = (l1.start.y * S2 + l1.end.y * S1) / (S1 + S2);
    return p;
}
int JudgeOut(const V &x, const Point &p) // 点在线的左侧
{
    return CroMul(V(x.start, p), x) > eps; // 点在左侧返回0,右侧返回1
}
int Parellel(const V &x, const V &y) { return fabs(CroMul(x, y)) < eps; } //平行
//返回1平行
void ChangeDirection() //切换顺逆方向
{
    for(int i=1;i<n;++i)
        swap(l[i].start,l[i].end);
}
double CheckDirection()
{
    double ans=0;
    for(int i=0;i<n;++i)//判断是否是顺时针
        ans+=CroMul(V(Point(0,0),l[i].start),V(Point(0,0),l[i].end));
    return ans;//ans>0逆时针，sum<0顺时针
}
int Cmp(V a, V b)
{
    if (fabs(a.ang - b.ang) < eps)
        // 角度相同时，不同的边在不同的位置
        // 左边的边在后面的位置，这样的话，进行计算的时候就可以忽略 相同角度边的影响了
        return CroMul(V(b.end - a.start), V(a.end - b.start)) > eps;
        // 左边的边在前面的位置，要进行进行去重判断 。
    return a.ang < b.ang;
}
double HplaneIntersection()
{
    if(CheckDirection() < 0) ChangeDirection();//改成逆时针
    int top = 1, bot = 0;
    sort(l, l + n, Cmp); //下标从0开始
    int tmp = 1;
    for (int i = 1; i < n; ++i)
        if (l[i].ang - l[i - 1].ang > eps)
            l[tmp++] = l[i]; // 去重,如果该边和前面的边平行，则忽略。
    n = tmp, st[0] = l[0], st[1] = l[1];
    for (int i = 2; i < n; ++i)
    {
        if (Parellel(st[top], st[top - 1]) || Parellel(st[bot], st[bot + 1]))
return 0;
        while (bot < top && JudgeOut(l[i], LineInterDot(st[top], st[top - 1])))
--top;
        while (bot < top && JudgeOut(l[i], LineInterDot(st[bot], st[bot + 1])))
++bot;
        st[++top] = l[i];
```

```
    }
    while (bot < top && JudgeOut(st[bot], LineInterDot(st[top], st[top - 1]))) --top;
    while (bot < top && JudgeOut(st[top], LineInterDot(st[bot], st[bot + 1]))) ++bot;
    if (top <= bot + 1) return 0.00;
    st[++top] = st[bot], ccnt = 0;
    for (int i = bot; i < top; ++i) convex[ccnt++] = LineInterDot(st[i], st[i + 1]);
    //计算面积，半平面凸包交点集convex[0, ccnt - 1]
    double ans = 0;
    convex[ccnt] = convex[0];
    for (int i = 0; i < ccnt; ++i)
        ans += CroMul(V(Point(0, 0), convex[i]), V(Point(0, 0), convex[i + 1]));
    return ans / 2;
}
```

平面最近点对

时间复杂度 $O(nlogn)$
分治方法Ⅰ：比第二种更慢但是码量少一点

```
struct Point
{
    double x, y;
};
typedef vector<Point>::iterator Iter;
bool cmpx(const Point a, const Point b) { return a.x < b.x; }
bool cmpy(const Point a, const Point b) { return a.y < b.y; }
double dis(const Point a, const Point b)
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
void slv(const Iter l, const Iter r, double &d)
{
    if (r - l <= 1) return;
    vector<Point> Q;
    Iter t = l + (r - l) / 2;
    double w = t->x;
    slv(l, t, d), slv(t, r, d), inplace_merge(l, t, r, cmpy);
    for (Iter x = l; x != r; ++x) if (abs(w - x->x) <= d) Q.push_back(*x);
    for (Iter x = Q.begin(), y = x; x != Q.end(); ++x)
    {
        while (y != Q.end() && y->y <= x->y + d) ++y;
        for (Iter z = x + 1; z != y; ++z) d = min(d, dis(*x, *z));
    }
}
vector<Point> Poi; int n;
double preparata()
{
    double ans = 1e18;
    sort(Poi.begin(),Poi.end(), cmpx);
    slv(Poi.begin(),Poi.end(), ans);
    return ans;
}
```

分治方法II：比第一种更快但是码量大一点

```
struct pt
{
    double x, y;
    int id;
};
bool cmpx(const pt a, const pt b) { return a.x < b.x; }
bool cmpy(const pt a, const pt b) { return a.y < b.y; }
int n, ansa, ansb; // 答案在这
vector<pt> a;     // 下标0开始
double mindist;
void upd_ans(const pt &a, const pt &b)
{
    double dist = sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
    if (dist < mindist)
        mindist = dist, ansa = a.id, ansb = b.id;
}
void rec(int l, int r)
{
    if (r - l <= 3)
    {
        for (int i = l; i <= r; ++i)
            for (int j = i + 1; j <= r; ++j)
                upd_ans(a[i], a[j]);
        sort(a.begin() + l, a.begin() + r + 1, cmpy);
        return;
    }
    int m = (l + r) >> 1;
    double midx = a[m].x;
    rec(l, m), rec(m + 1, r);
    inplace_merge(a.begin() + l, a.begin() + m + 1, a.begin() + r + 1, cmpy);
    static pt t[N]; // 缓存数组
    int tsz = 0;
    for (int i = l; i <= r; ++i)
        if (abs(a[i].x - midx) < mindist)
        {
            for (int j = tsz - 1; j >= 0 && a[i].y - t[j].y < mindist; --j)
upd_ans(a[i], t[j]);
            t[tsz++] = a[i];
        }
}
void preparata()
{
    sort(a.begin(), a.begin() + n, cmpx);
    mindist = (ll)1e18;
    rec(0, n - 1);
}
```

非分治：更快（常数小）

其实，除了上面提到的分治算法，还有另一种时间复杂度同样是 $O(n \log n)$ 的非分治算法。

我们可以考虑一种常见的统计序列的思想：对于每一个元素，将它和它的左边所有元素的贡献加入到答案中。平面最近点对问题同样可以使用这种思想。

具体地，我们把所有点按照 x_i 为第一关键字、y_i 为第二关键字排序，并建立一个以 y_i 为第一关键字、x_i 为第二关键字排序的 multiset。对于每一个位置 i，我们执行以下操作：

1. 将所有满足 $x_i - x_j >= d$ 的点从集合中删除。它们不会再对答案有贡献。
2. 对于集合内满足 $|y_i - y_j| < d$ 的所有点，统计它们和 p_i 的距离。
3. 将 p_i 插入到集合中。

```
int n;
double ans = 1e20;
struct Point
{
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
};
struct cmp_y { bool operator()(const Point &a, const Point &b) const { return
a.y < b.y; } };
void upd_ans(const Point &a, const Point &b)
{
    double dist = sqrt(pow((a.x - b.x), 2) + pow((a.y - b.y), 2));
    if (ans > dist) ans = dist;
}
Point a[N];//下标0开始
multiset<Point, cmp_y> s;
void preparata()
{
    sort(a, a + n, [&](Point a, Point b) -> bool { return a.x < b.x || (a.x ==
b.x && a.y < b.y); });
    for (int i = 0, l = 0; i < n; i++)
    {
        while (l < i && a[i].x - a[l].x >= ans) s.erase(s.find(a[l++]));
        for (auto it=s.lower_bound({a[i].x, a[i].y - ans}); it != s.end() && it-
>y - a[i].y < ans; it++)
            upd_ans(*it, a[i]);
        s.insert(a[i]);
    }
}
```

最小覆盖圆(随机增量法)

随机打乱后期望时间复杂度 $O(n)$

假设圆 O 是前 $i-1$ 个点的最小覆盖圆，加入第 i 个点，如果在圆内或边上则什么也不做。否则，新得到的最小覆盖圆肯定经过第 i 个点。

然后以第 i 个点为基础（半径为 0），重复以上过程依次加入第 j 个点，若第 j 个点在圆外，则最小覆盖圆必经过第 j 个点。

重复以上步骤。（因为最多需要三个点来确定这个最小覆盖圆，所以重复三次）

遍历完所有点之后，所得到的圆就是覆盖所有点得最小圆。

复杂度证明

由于一堆点最多只有3个点确定了最小覆盖圆，因此n个点中每个点参与确定最小覆盖圆的概率不大于$\frac{3}{n}$

所以，每一层循环在第i个点处调用下一层的概率不大于$\frac{3}{i}$

那么设算法的三个循环的复杂度分别为$T_1(n), T_2(n), T_3(n)$，则有：

$$T_1(n) = O(n) + \sum_{i=1}^{n} \frac{3}{i} T_2(i)$$

$$T_2(n) = O(n) + \sum_{i=1}^{n} \frac{3}{i} T_3(i)$$

$$T_3(n) = O(n)$$

不难解得，$T_1(n) = T_2(n) = T_3(n) = O(n)$

```
const double zero = 1e-9;
class Point
{
public:
    double x, y;
    Point() {}
    Point(double a, double b) { x = a, y = b; }
    double distance(Point m) { return sqrt((m.x - x) * (m.x - x) + (m.y - y) *
(m.y - y)); }
};
double squ(double x) { return x * x; }
Point geto(Point a, Point b, Point c)//三点最小覆盖圆
{
    double a1, a2, b1, b2, c1, c2;
    Point ans;
    a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
    c1 = squ(b.x) - squ(a.x) + squ(b.y) - squ(a.y);
    a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
    c2 = squ(c.x) - squ(a.x) + squ(c.y) - squ(a.y);
    if (fabs(a1) < zero) ans = {(c2 - c1 / b1 * b2) / a2, c1 / b1};
    else if (fabs(b1) < zero) ans = {c1 / a1, (c2 - c1 / a1 * a2) / b2};
    else ans = {(c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2), (c2 * a1 - c1 * a2) /
(b2 * a1 - b1 * a2)};
    return ans;
}
void RandIncAlg(vector<Point> p, Point &o, double &r)
{
    int n = p.size();
    for (int i = 0; i < n; ++i) swap(p[rand() % n], p[rand() % n]);
    o = p[0];
```

```
    for (int i = 0; i < n; ++i)
    {
        if (o.distance(p[i]) < r || fabs(o.distance(p[i]) - r) < zero) continue;
        o.x = (p[i].x + p[0].x) / 2;
        o.y = (p[i].y + p[0].y) / 2;
        r = p[0].distance(p[i]) / 2;
        for (int j = 1; j < i; ++j)
        {
            if (o.distance(p[j]) < r || fabs(o.distance(p[j]) - r) < zero)
continue;
            o.x = (p[i].x + p[j].x) / 2;
            o.y = (p[i].y + p[j].y) / 2;
            r = p[i].distance(p[j]) / 2;
            for (int k = 0; k < j; ++k)
            {
                if (o.distance(p[k]) < r || fabs(o.distance(p[k]) - r) < zero)
continue;
                o = geto(p[i], p[j], p[k]);
                r = o.distance(p[i]);
            }
        }
    }
}
```

三角剖分DT

Delaunay 三角剖分

定义

在数学和计算几何中，对于给定的平面中的离散点集 P，其 Delaunay 三角剖分 DT(P) 满足：

1. 空圆性：DT(P) 是 **唯一** 的（任意四点不能共圆），在 DT(P) 中，**任意** 三角形的外接圆范围内不会有其它点存在。
2. 最大化最小角：在点集 P 可能形成的三角剖分中，DT(P) 所形成的三角形的最小角最大。从这个意义上讲，DT(P) 是 **最接近于规则化** 的三角剖分。具体的说是在两个相邻的三角形构成凸四边形的对角线，在相互交换后，两个内角的最小角不再增大。

性质

1. 最接近：以最接近的三点形成三角形，且各线段（三角形的边）皆不相交。
2. 唯一性：不论从区域何处开始构建，最终都将得到一致的结果（点集中任意四点不能共圆）。
3. 最优性：任意两个相邻三角形构成的凸四边形的对角线如果可以互换的话，那么两个三角形六个内角中最小角度不会变化。
4. 最规则：如果将三角剖分中的每个三角形的最小角进行升序排列，则 Delaunay 三角剖分的排列得到的数值最大。
5. 区域性：新增、删除、移动某一个顶点只会影响邻近的三角形。
6. 具有凸边形的外壳：三角剖分最外层的边界形成一个凸多边形的外壳。

Voronoi 图 ¶

Voronoi 图由一组由连接两邻点直线的垂直平分线组成的连续多边形组成，根据 n 个在平面上不重合种子点，把平面分成 n 个区域，使得每个区域内的点到它所在区域的种子点的距离比到其它区域种子点的距离近。

Voronoi 图是 Delaunay 三角剖分的对偶图，可以使用构造 Delaunay 三角剖分的分治算法求出三角网，再使用最左转线算法求出其对偶图实现在 $O(n \log n)$ 的时间复杂度下构造 Voronoi 图。

```
const int E = 1e3;
struct Point
{
    double x, y;
    int id;
    Point(double a = 0, double b = 0, int c = -1) : x(a), y(b), id(c) {}
    bool operator<(const Point &a) const { return x < a.x || (fabs(x - a.x) <
eps && y < a.y); }
    bool operator==(const Point &a) const { return fabs(x - a.x) < eps && fabs(y
- a.y) < eps; }
    double dist2(const Point &b) { return (x - b.x) * (x - b.x) + (y - b.y) * (y
- b.y); }
};
struct Point3D
{
    double x, y, z;
    Point3D(double a = 0, double b = 0, double c = 0) : x(a), y(b), z(c) {}
    Point3D(const Point &p) { x = p.x, y = p.y, z = p.x * p.x + p.y * p.y; }
    Point3D operator-(const Point3D &a) const { return Point3D(x - a.x, y - a.y,
z - a.z); }
    double dot(const Point3D &a) { return x * a.x + y * a.y + z * a.z; }
};
struct Edge
{
    int id;
    std::list<Edge>::iterator c;
    Edge(int id = 0) { this->id = id; }
};
int cmp(double v) { return fabs(v) > eps ? (v > 0 ? 1 : -1) : 0; }
double cross(const Point &o, const Point &a, const Point &b)
{
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}
Point3D cross(const Point3D &a, const Point3D &b)
{
    return Point3D(a.y * b.z - a.z * b.y, -a.x * b.z + a.z * b.x,
                   a.x * b.y - a.y * b.x);
}
int inCircle(const Point &a, Point b, Point c, const Point &p)//抛物面判法
{
    if (cross(a, b, c) < 0) std::swap(b, c);
    Point3D a3(a), b3(b), c3(c), p3(p);
    b3 = b3 - a3, c3 = c3 - a3, p3 = p3 - a3;
    Point3D f = cross(b3, c3);
    return cmp(p3.dot(f)); // check same direction, in: < 0, on: = 0, out: > 0
}
int intersection(const Point &a, const Point &b, const Point &c, const Point &d)
{ // seg(a, b) and seg(c, d)
```

```
    return cmp(cross(a, c, b)) * cmp(cross(a, b, d)) > 0 &&
           cmp(cross(c, a, d)) * cmp(cross(c, d, b)) > 0;
}
class Delaunay
{
public:
    std::list<Edge> head[E]; // graph
    Point p[E];
    int n, rename[E];
    void init(Point p[], int n)
    {
        memcpy(this->p, p, sizeof(Point) * n);
        std::sort(this->p, this->p + n);
        for (int i = 0; i < n; i++) rename[p[i].id] = i;
        this->n = n;
        divide(0, n - 1);
    }
    void addEdge(int u, int v) //十字list
    {
        head[u].push_front(Edge(v));
        head[v].push_front(Edge(u));
        head[u].begin()->c = head[v].begin();
        head[v].begin()->c = head[u].begin();
    }
    void divide(int l, int r)
    {
        if (r - l <= 2) // #point <= 3
        {
            for (int i = l; i <= r; i++)
                for (int j = i + 1; j <= r; j++)
                    addEdge(i, j);
            return;
        }
        int mid = l + r >> 1;
        divide(l, mid);
        divide(mid + 1, r);
        std::list<Edge>::iterator it;
        int nowl = l, nowr = r;
        for (int update = 1; update;)
        {
            // find left and right convex, lower common tangent
            update = 0;
            Point ptL = p[nowl], ptR = p[nowr];
            for (it = head[nowl].begin(); it != head[nowl].end(); it++)
            {
                Point t = p[it->id];
                double v = cross(ptR, ptL, t);
                if (cmp(v) > 0 || (cmp(v) == 0 && ptR.dist2(t) <
ptR.dist2(ptL)))
                {
                    nowl = it->id, update = 1;
                    break;
                }
            }
            if (update) continue;
            for (it = head[nowr].begin(); it != head[nowr].end(); it++)
            {
                Point t = p[it->id];
```

```
                double v = cross(ptL, ptR, t);
                if (cmp(v) < 0 || (cmp(v) == 0 && ptL.dist2(t) <
ptL.dist2(ptR)))
                {
                    nowr = it->id, update = 1;
                    break;
                }
            }
        }
        addEdge(nowl, nowr); // add tangent
        for (int update = 1; true;)
        {
            update = 0;
            Point ptL = p[nowl], ptR = p[nowr];
            int ch = -1, side = 0;
            for (it = head[nowl].begin(); it != head[nowl].end(); it++)
                if (cmp(cross(ptL, ptR, p[it->id])) > 0 &&
                    (ch == -1 || inCircle(ptL, ptR, p[ch], p[it->id]) < 0))
                    ch = it->id, side = -1;
            for (it = head[nowr].begin(); it != head[nowr].end(); it++)
                if (cmp(cross(ptR, p[it->id], ptL)) > 0 &&
                    (ch == -1 || inCircle(ptL, ptR, p[ch], p[it->id]) < 0))
                    ch = it->id, side = 1;
            if (ch == -1) break; // upper common tangent
            if (side == -1)
            {
                for (it = head[nowl].begin(); it != head[nowl].end();)
                {
                    if (intersection(ptL, p[it->id], ptR, p[ch]))
                    {
                        head[it->id].erase(it->c);
                        head[nowl].erase(it++);
                    }
                    else it++;
                }
                nowl = ch;
                addEdge(nowl, nowr);
            }
            else
            {
                for (it = head[nowr].begin(); it != head[nowr].end();)
                {
                    if (intersection(ptR, p[it->id], ptL, p[ch]))
                    {
                        head[it->id].erase(it->c);
                        head[nowr].erase(it++);
                    }
                    else it++;
                }
                nowr = ch;
                addEdge(nowl, nowr);
            }
        }
    }
    std::vector<std::pair<int, int>> getEdge()
    {
        std::vector<std::pair<int, int>> ret;
        ret.reserve(n);
```

```
        std::list<Edge>::iterator it;
        for (int i = 0; i < n; i++)
        {
            for (it = head[i].begin(); it != head[i].end(); it++)
            {
                if (it->id < i) continue;
                ret.push_back(std::make_pair(p[i].id, p[it->id].id));
            }
        }
        return ret;
    }
};
```

计算几何操作集

```
const double eps = 1e-10;
const double pi = acos(-1);
const int inf = 0x3f3f3f3f;

int sign(double x)
{
    if (fabs(x) <= eps) return 0;
    return x < 0 ? -1 : 1;
}
class Point
{
public:
    double x, y;
    Point() {}
    Point(double a, double b) { x = a, y = b; }
    friend Point operator+(const Point &a, const Point &b) { return Point(a.x +
b.x, a.y + b.y); }
    friend Point operator-(const Point &a, const Point &b) { return Point(a.x -
b.x, a.y - b.y); }
    friend double operator^(Point a, Point b) { return a.x * b.y - a.y * b.x; }
    bool operator<(const Point &m) const { return x < m.x || (x == m.x && y <
m.y); }
    double distance(Point m) { return sqrt((m.x - x) * (m.x - x) + (m.y - y) *
(m.y - y)); }
    double manhattan(Point m) { return fabs(m.x - x) + fabs(m.y - y); }
};
class Line
{
public:
    double a, b, c;
    double k, d;
    Line() {}
    Line(Point x, Point y)
    {
        a = y.y - x.y;
        b = x.x - y.x;
        c = y.x * x.y - x.x * y.y;
    }
    Line(double x, double y)
    {
        k = x, d = y;
        a = k, b = -1, c = d;
```

```
    }
    Line(double x, double y, double z)
    {
        a = x, b = y, c = z;
        if (b != 0) k = -a / b, d = -c / b;
    }
    int relationP(Point m)
    {
        double p = a * m.x + b * m.y + c;
        if (p > eps) return -1; // 上方
        else if (p < -eps) return 1; // 下方
        return 0;     // 直线上
    }
    Point intersectL(Line m)
    {
        double D = a * m.b - m.a * b;
        if (D == 0) return Point(inf, inf); // 平行
        return Point(b * m.c - m.b * c, m.a * c - a * m.c);
    }
    double distanceP(Point m) { return fabs(a * m.x + b * m.y + c) / sqrt(a * a
+ b * b); }
};
class Vector
{
public:
    double x, y;
    Vector() {}
    Vector(Point n) { x = n.x, y = n.y; }
    Vector(double n, double m) { x = n, y = m; }
    Vector(Point n, Point m) { x = m.x - n.x, y = m.y - n.y; }
    Vector(double a, double b, double c, double d) { x = c - a, y = d - b; }
    double vabs() { return sqrt(x * x + y * y); }
    double cosV(Vector m) { return *this * m / this->vabs() / m.vabs(); }
    double angle() { return atan2(y, x); } // Get_Polar_Angle
    Vector operator-() const { return {-x, -y}; }
    Vector operator+(const Vector &m) const { return {x + m.x, y + m.y}; }
    Vector operator-(const Vector &m) const { return {x - m.x, y - m.y}; }
    Vector operator*(const double &m) const { return {x * m, y * m}; }
    double operator*(const Vector &m) const { return x * m.x + y * m.y; }
    double operator^(const Vector &m) const { return x * m.y - y * m.x; } //正数
为逆时针左拐
};
class Vecline
{
public:
    Vector u;
    Point a, b;
    Vecline() {}
    Vecline(Point n, Point m) { a = n, b = m, u = {n, m}; }
    Vecline(Vector n, Point m) { a = m, b = {n.x + m.x, n.y + m.y}, u = n; }
    Vecline(double w, double x, double y, double z) { u = {w, x, y, z}, a = {w,
x}, b = {y, z}; }
    int straddle(Vecline m) // 跨立实验||线段形
    {
        if (m.relationP(a) * m.relationP(b) <= 0)
            if (this->relationP(m.a) * this->relationP(m.b) <= 0)
                return 1; // 通过
        return 0;         // 不通过
```

```
    }
    int rejection(Vecline m) // 快速排斥实验||线段形||不通过即相离
    {
        return min(a.x, b.x) <= max(m.a.x, m.b.x) &&
               min(m.a.x, m.b.x) <= max(a.x, b.x) &&
               min(a.y, b.y) <= max(m.a.y, m.b.y) &&
               min(m.a.y, m.b.y) <= max(a.y, b.y);
    }
    int relationP(Point m)
    {
        double temp = u ^ Vector(a, m);
        if (temp > eps) return -1; // 左边
        else if (temp < -eps) return 1; // 右边
        return 0;     // 上方
    }
    int relationL(Vecline m) { return rejection(m) && straddle(m); }
    Point intersectL(Vecline m)
    {
        Vector v(m.a, a);
        double t = (v ^ m.u) / (u ^ m.u);
        Vector e = u * t;
        return Point(a.x - e.x, a.y - e.y);
    }
    double distanceP(Point m)
    {
        Vector v(a, m);
        double c = v.cosV(u);
        return v.vabs() * sqrt(1 - c * c);
    }
    operator Line() { return static_cast<Line>(Line(a, b)); }
};
class Circle
{
public:
    Point o;
    double r;
    Circle(){};
    Circle(Point x) { o = x; }
    Circle(double y) { r = y; }
    Circle(Point x, double y) { o = x, r = y; }
    Circle(Point a, Point b) { o = {(a.x + b.x) / 2, (a.y + b.y) / 2}, r =
a.distance(b) / 2; }
    Circle(Point a, Point b, Point c)
    {
        Vecline x(Vector{a.y - b.y, b.x - a.x}, Point{(a.x + b.x) / 2, (a.y +
b.y) / 2});
        Vecline y(Vector{a.y - c.y, c.x - a.x}, Point{(a.x + c.x) / 2, (a.y +
c.y) / 2});
        o = x.intersectL(y);
        r = o.distance(a);
    }
    int relation(double p)
    {
        if (r - p > eps) return 1; // 里面||相交
        else if (p - r > eps) return -1; // 外面||相离
        return 0;      // 圆上||相切
    }
    int relationC(Circle m)
```

```
    {
        double d = m.o.distance(o);
        if (d - (r + m.r) > eps) return 0; // 外离
        else if (fabs(d - (r + m.r)) < eps) return 1; // 外切
        else if (fabs(d - fabs(r - m.r)) < eps) return 2; // 内切
        else if (fabs(r - m.r) - d > eps) return 3; // 内含
        else return 4; // 相交
    }
    int relationP(Point m) { return relation(m.distance(o)); }
    int relationL(Line m) { return relation(m.distanceP(o)); }
    int relationL(Vecline m) { return relation(m.distanceP(o)); }
    double distanceP(Point m) { return m.distance(o) - r; }
    double distanceL(Line m) { return m.distanceP(o) - r; }
    double distanceC(Circle m) { return m.o.distance(o) - r - m.r; }
    Point getPoint(double m) { return Point(o.x + r * cos(m), o.y + r * sin(m));
} // 倾斜角方向的交点
    Point getPoint(Vector m) { return Point(o.x + r * cos(m.angle()), o.y + r *
sin(m.angle())); }
    vector<Point> intersectL(Vecline m)
    {
        int rel = this->relationL(m);
        vector<Point> poi;
        Vector v = {-m.u.y, m.u.x};
        double d = m.distanceP(o);
        if (m.relationP(o) < 0) v = -v;
        v = v * d * (1.0 / v.vabs());
        if (rel == 0) poi.push_back({o.x + v.x, o.y + v.y});
        else if (rel == 1)
        {
            double temp = sqrt(r * r - d * d) / m.u.vabs();
            Vector e = m.u * temp + v;
            poi.push_back({o.x + e.x, o.y + e.y});
            poi.push_back({o.x - e.x, o.y - e.y});
        }
        return poi;
    }
    vector<Vecline> GCCI(Circle A) // Get_Circle_Circle_Intersection
    {
        vector<Vecline> res;
        Circle B = *this;
        if (A.r < B.r) swap(A, B);
        Vector u = {A.o, B.o};
        double d = u.vabs(), rdec = A.r - B.r, radd = A.r + B.r;
        if (sign(d - rdec) < 0) return res; // 内含
        if (sign(d) == 0 && A.r == B.r) return res; // 重合，无线多
        double ua = u.angle();
        if (sign(d - rdec) == 0) // 内切
        {
            res.push_back({A.getPoint(ua), B.getPoint(ua)});
            return res;
        }
        double da = acos((A.r - B.r) / d); // 2条外公切线
        res.push_back({A.getPoint(ua + da), B.getPoint(ua + da)});
        res.push_back({A.getPoint(ua - da), B.getPoint(ua - da)});
        if (sign(d - radd) == 0) res.push_back({A.getPoint(ua), B.getPoint(ua +
pi)}); // 1条内公切线
        else if (sign(d - radd) > 0) // 2条内公切线
        {
```

```
            da = acos((A.r + B.r) / d);
            res.push_back({A.getPoint(ua + da), B.getPoint(ua + da + pi)});
            res.push_back({A.getPoint(ua - da), B.getPoint(ua - da + pi)});
        }
        return res;
    }
};
class Polygon
{
public:
    vector<Point> p;
    Polygon() {}
    Polygon(vector<Point> poi) { p = poi; }
    double getS()//需按顺或逆时针排序后使用
    {
        double res = 0;
        for (int i = 0; i < p.size(); ++i) res += Vector(p[i]) ^ Vector(p[(i +
1) % p.size()]);
        return fabs(res / 2);
    }
    void andrew() // 凸包化
    {
        int tp = 0;
        sort(p.begin(), p.end());                              // 对点进行排序
        vector<int> stk(p.size() + 5), used(p.size() + 5); // stk[] 是整型，存的是
下标
        stk[++tp] = 0;
        // 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
        for (int i = 1; i < p.size(); ++i)
        {
            while (tp >= 2 && (Vector(p[stk[tp]], p[stk[tp - 1]]) ^ Vector(p[i],
p[stk[tp]])) <= 0)
                used[stk[tp--]] = 0;
            used[i] = 1; // used 表示在凸壳上
            stk[++tp] = i;
        }
        int tmp = tp; // tmp 表示下凸壳大小
        for (int i = p.size() - 1; i >= 0; --i)
            if (!used[i]) // ↓求上凸壳时不影响下凸壳
            {
                while (tp > tmp && (Vector(p[stk[tp]], p[stk[tp - 1]]) ^
Vector(p[i], p[stk[tp]])) <= 0)
                    used[stk[tp--]] = 0;
                used[i] = 1;
                stk[++tp] = i;
            }
        vector<Point> newp;
        for (int i = 1; i <= tp; ++i) newp.push_back(p[stk[i]]);// 复制到新数组中去
        newp.pop_back(); //求周长注释掉这行||第一个点存了两次
        this->p = newp;
    }
};
```

具体数学

约瑟夫问题

线性算法

设 $J_{n,k}$ 表示规模分别为 n, k 的约瑟夫问题的答案。我们有如下递归式

$$J_{n,k} = (J_{n-1,k} + k) \bmod n$$

这个也很好推。你从 0 开始数 k 个，让第 $k-1$ 个人出局后剩下 $n-1$ 个人，你计算出在 $n-1$ 个人中选的答案后，再加一个相对位移 k 得到真正的答案。这个算法的复杂度显然是 $\Theta(n)$ 的。

```
int josephus(int n, int k) //O(n)
{
    int res = 0;
    for (int i = 1; i <= n; ++i) res = (res + k) % i;
    return res;
}
```

```
int josephus(int n, int k) //O(klogn)
{
    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;
}
```

玄学算法

> *领域展开，坐杀博徒！All My People！*

随机数

```
//基础随机数
srand(time(NULL));
rand();
double getrand() { return (double)rand() / RAND_MAX; } //取[0,1]
//random_device{}()当种子(自己用着好像有问题),也可填time(0)
mt19937 eng(random_device{}()); //返回值是unsigned int,区间范围为[0 ,4e9],即
UINT32_MAX
mt19937_64 eng64(random_device{}()); //返回值是unsigned long long,范围更大,
UINT64_MAX
uniform_int_distribution<int> myrand1(l,r); //返回[l,r]的整数,默认int,保证均匀分布,可
为负数
uniform_real_distribution<double> myrand2(l,r); //返回[l,r)的实数,默认double,保证均
匀分布,可为负数
myrange1(myrand), myrange2(myrand);//使用
double myrand() { return (double)mrand() / UINT32_MAX; } //取[0,1]
```

模拟退火

先用一句话概括：如果新状态的解更优则修改答案，否则以一定概率接受新状态。

我们定义当前温度为 T，新状态 S' 与已知状态 S（新状态由已知状态通过随机的方式得到）之间的能量（值）差为 ΔE（$\Delta E \geqslant 0$），则发生状态转移（修改最优解）的概率为

$$P(\Delta E) = \begin{cases} 1, & S' \text{ is better than } S, \\ \mathrm{e}^{\frac{-\Delta E}{T}}, & \text{otherwise.} \end{cases}$$

注意：我们有时为了使得到的解更有质量，会在模拟退火结束后，以当前温度在得到的解附近多次随机状态，尝试得到更优的解（其过程与模拟退火相似）。

模拟退火时我们有三个参数：初始温度 T_0，降温系数 d，终止温度 T_k。其中 T_0 是一个比较大的数，d 是一个非常接近 1 但是小于 1 的数，T_k 是一个接近 0 的正数。

首先让温度 $T = T_0$，然后按照上述步骤进行一次转移尝试，再让 $T = d \cdot T$。当 $T < T_k$ 时模拟退火过程结束，当前最优解即为最终的最优解。

注意为了使得解更为精确，我们通常不直接取当前解作为答案，而是在退火过程中维护遇到的所有解的最优值。

> 算法改进
> (1.设计合适的状态产生函数，使其根据搜索进程的需要表现出状态的全空间分散性或局部区域性；
> (2.设计高效的退火策略；
> (3.避免状态的迂回搜索；
> (4.采用并行搜索结构；
> (5.为避免陷入局部极小，改进对温度的控制方式，选择合适的初始状态；
> (6.可以采用分块；
> (7.设计合适的算法终止准则。

```
//可使用循环限制退火时间 || MAX_TIME为自己设置的略小于时限的数（单位ms）|| clock()单位ms
const double begT= 10000, endT= 1e-12, delT= 0.99; //ChangeT一般根据题目n决定
while ((double)clock()/CLOCKS_PER_SEC < MAX_TIME && BegT > EndT) { begT *= delT;
}
----------------------------------------执行次数----------------------------------
--------
double begT = 1e5, endT = 1e-9, delT = 0.999; //cnt == 32221
double begT = 1e5, endT = 1e-12, delT = 0.9999; //cnt == 391420
double begT = 1e9, endT = 1e-9, delT = 0.99998; //cnt == 2072306
double begT = 1e9, endT = 1e-9, delT = 0.99999; //cnt == 4144633
//cnt(0.xxxxx9) == 2 * cnt(0.xxxxx8),以....999的执行次数约为....998的两倍
```

```
//simulateAnneal过程 (以求min为例)
const double begT = 10000, endT = 1e-12, delT = 0.999, MAX_TIME = 0.985s; //
ChangeT一般根据题目n决定
double nowx = ansx, nowy = ansy;
while ((double)clock() / CLOCKS_PER_SEC < MAX_TIME) // && begT > endT(为正确率可以
不加,卡极限时间)
{
    double nxtx = nowx + begT * (Rand(-1, 1)); //随着温度的降低，跳跃越来越不随机
    double nxty = nowy + begT * (Rand(-1, 1));
    double delta = cal(nxtx, nxty) - cal(nowx, nowy); //新值与当前值的差,cal未给出
    //在cal(x,y)中更新ans：if (res < ans) ansx = x, ansy = y; //新值更优,取新值
    if (exp(-delta / begT) > Rand(0, 1)) //温度越低概率越小,概率接受非最优可能值
        nowx = nxtx, nowy = nxty; //接受非最优值,但不更新ans
```

```
    begT *= delT; //降温
}
for (int i = 1; i <= 1e5; ++i) //以当前得到的最优解附近多次随机状态，尝试得到更优的解
{
    double nxtx = ansx + t * (Rand(-1, 1));
    double nxty = ansy + t * (Rand(-1, 1));
    cal(nxtx, nxty);
}
```

位运算

GCC__builtin函数

表1 gcc中builtin函数及作用

函数名称	功能简介
__builtin_clz(x)	计算x前导0的个数。x=0时结果未定义
__builtin_ctz(x)	计算x末尾0的个数。x=0时结果未定义
__builtin_ffs(x)	返回x中最后一个为1的位是从后向前的第几位，如__builtin_ffs(0x789)=1, __builtin_ffs(0x78c)=3
__builtin_popcount(x)	计算x中1的个数
__builtin_parity(x)	计算x中1个数的奇偶性
__builtin_nearbyint(x)	计算参数x经过四舍五入后的值
__builtin_floor(x)	向下取整，既取不大于x的最大整数
__builtin_ceil(x)	向上取整，既返回不小于x的最小整数
__builtin_trunc(x)	将数字x的小数部分直接去掉，返回整数
__builtin_sqrt(x)	求x的开平方根，返回结果

AND

- 交换律： A&B = B&A
- 结合律：(A&B)&C = A&(B&C)
- 与值范围：0 <= A&B <= min(A, B)
- 不能和基本运算结合分配：(A+B)&C != A&C + B&C　　(A*B)&C != A&C * B&C

a与0xaaaaaaaa可以保留a的二进制位上偶数位上的1， a与0x55555555可以保留a的二进制位上奇数位上的1
0xaaaaaaaa 的二进制形式为：10101010 10101010 10101010 10101010
0x55555555 的二进制形式为：01010101 01010101 01010101 01010101

n&(n-1) 一次运算就相当于把n二进制末位1变成0

```
int modPowerOfTwo(int x, int mod) { return x & (mod - 1); } //求x对2的非负整数次幂取模
bool isPowerOfTwo(int x) { return x > 0 && (x & (x - 1)) == 0; } //判断一个数是不是2的非负整数次幂
```

OR

- 交换律： A|B = B|A
- 结合律：(A|B)|C = A|(B|C)
- 或值范围：max(A, B) <= A|B <= A+B

- 不能和基本运算结合分配：(A+B)|C != A|C + B|C　　(A*B)|C != A&C * B|C

```
//用二进制记录小写字母字符串有哪些字母
int mark = 0;
for(auto i : str) mark |= 1 << (i - 'a');
//如果没有相同字母(mark & other)值位0
if(!(mark & other)) cout << "没有相同字母" << endl;
else if(mark & other) cout << "有相同字母" << endl;
```

XOR

- 交换律： A^B = B^A
- 结合律：(A^B)^C = A^(B^C)
- 自反性：A^B^B = A
- 异或值范围：abs(A - B) <= A^B <= A+B
- 不能和基本运算结合分配：(A+B)^C != A^C + B^C　　(A*B)^C != A^C * B^C

```
//求数组中唯一出现奇数次的值x
int exor(int array[],int length)
{
    int eor = 0;
    for(int i = 0; i < length; ++i) eor^=array[i];
    return eor;
}
//求数组中唯二出现奇数次的值a,b
int exor(int array[],int length)
{
    int eor = 0, a = 0, b;
    for(int i = 0; i < length; ++i) eor^=array[i];
    /*因为a，b肯定不相等，所以a^b肯定不为0，所以enr肯定有一位为1，通过这一位，将数组分类，a，
b属于不同的类*/
    int c = ((-eor)&(eor));
    for(int i=0; i < length; ++i) if((array[i]&c)) a ^= array[i];
    b = a^eor;
    return 0;
}
//异或枚举部分全排列
int n, a[n]; //a[i] = i;i:[0~n-1]
for(int i = 0; i < (n&(-n)); ++i) //枚举((n&(-n)) - 1)组[0~n-1]的全排列
    for(int j = 0; j < n; ++j)
        cout << a[j] ^ i << ' ';
```

DP

最大子段和

```
ll calc(int n)
{
    ll res = a[1], sum = 0;
    for (int i = 1; i <= n; ++i)
    {
        if (sum > 0) sum += a[i];
        else sum = a[i];
        res = max(sum, res);
    }
    return res;
}
```

LIS(最长上升子序列)

非严格递增

```
int a[N], dp[N], len;
int LIS(int n) //O(nlogn)
{
    for (int i = 1; i <= n; ++i)
    {
        int *p = upper_bound(dp, dp + 1 + len, a[i]); //非严格递增
        //int *p = lower_bound(dp, dp + 1 + len, a[i]); //严格递增
        *p = a[i];
        len += (p - dp == len + 1);
    }
    return len;
}
```

LCS(最长公共子序列)

```
int a[N], b[M], dp[N][M];//一般情况
int LCS(int n, int m) //O(nm)
{
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j)
            if(a[i] == b[j]) dp[i][j] = dp[i - 1][j - 1] + 1;
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
    return dp[n][m];
}
```

- 特殊情况

  因为两个序列都是1 n的全排列，那么两个序列元素互异且相同，也就是说只是位置不同罢了，那么我们通过一个map数组将A序列的数字在B序列中的位置表示出来——

- 因为**最长公共子序列是按位向后比对的，所以a序列每个元素在b序列中的位置如果递增，就说明b中的这个数在a中的这个数整体位置偏后**，可以考虑纳入LCS——那么就可以转变成$nlogn$求用来记录新的位置的map数组中的**LIS**。

```
int a[N], b[N], dp[N], pos[N], len;//(1 ~ n)全排列优化写法
int LCS(int n) //O(nlogn)
{
    for (int i = 1; i <= n; ++i) pos[b[i]] = i;
    for (int i = 1; i <= n; ++i)
    {
        int *p = upper_bound(dp, dp + 1 + len, pos[a[i]]);
        *p = pos[a[i]];
        len += (p - dp == len + 1);
    }
    return len;
}
```

01背包(免费k次问题)

> - 如果存在物品a,b满足a免费选走，b付费选走，则 $W_a >= W_b$
> - 如果存在物品a,b满足a免费选走，b没有选走，则 $V_a >= V_b$
>
> 所以将物品按代价从小到大排序，一定是前 $[1,x]$ 个物品01背包选走，后 $[x+1,n]$ 的物品选价值最大的选走

```
pair<int, ll> wv[N]; //fi为代价,se为价值
ll valk[N], dp[N][M]; //valk存[i ~ n]的k个物品最大价值和
ll begForFreeK(int n, int m, int k)
{
    ll res = 0;
    sort(wv + 1, wv + 1 + n); //按代价排序
    priority_queue<ll, vector<ll>, greater<ll>> maxk; //小根堆维护valk
    for (int i = n; i > n - k; --i)
    {
        valk[n - k + 1] += wv[i].se;
        maxk.push(wv[i].se);
    }
    if (!maxk.empty()) //小根堆维护valk
        for (int i = n - k; i > 0; --i)
            if (maxk.top() < wv[i].se)
            {
                valk[i] = valk[i + 1] + wv[i].se - maxk.top();
                maxk.pop();
                maxk.push(wv[i].se);
            }
    for (int i = 1; i <= n - k; ++i)
        for (int j = 0; j <= m; ++j)
            if (j < wv[i].fi) dp[i][j] = dp[i - 1][j];
            else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - wv[i].fi] +
wv[i].se);
    for (int i = 0; i <= n - k; ++i) res = max(res, valk[i + 1] + dp[i][m]); //枚
举分界点x求最值
    return res;
}
```

单调栈优化dp

单调队列，单调栈，动态规划。

当我们为了实现给动态规划的复杂度降维的时候，通常就需要单调栈/队列，通常用来维护前面状态下可以取到的最大值或者最小值，然后直接进行转移。

```
//滑动窗口,维护区间[i - k, i]最小值 || min为dq.front()
while(!dq.empty()&&dq.front().idx < i-k) dq.pop_front();
while(!dq.empty()&&dq.back().val>=s[i]) dq.pop_back();
dq.push_back({s[i],i});
```

区间基站建设最小代价问题:
要求区间内必须建一个基站，求最小成本

维护 $f(i)$ 表示只考虑前 i 个位置，且第 i 个位置必须建设基站的最小总成本。考虑上一个建设基站的位置 j，得到 dp 方程

$$f(i) = \min_j f(j) + a_i$$

由于题目要求每个区间里都至少要有一个基站，因此 $[j+1, i-1]$ 之间不能存在一个完整的区间。因此，对于每个 $1 \le i \le n$，我们计算 p_i 满足 $[p_i, i]$ 之间不存在一个完整的区间，且 p_i 尽可能小，则 $j \ge p_{i-1} - 1$。所有 p_i 的值可以用双指针法求出（因为如果 $[l, r]$ 里存在一个完整的区间，那么 $[l' \le l, r' \ge r]$ 里肯定也存在一个完整的区间，满足双指针法的特性）。

上述 dp 可以用 单调队列 优化到 $\mathcal{O}(n)$。

```
int A[N], LIM[N], q[N], head = 1, tail = 1;//A建设代价，B区间存放
long long f[N];A
vector<int> B[N]; // 负数表示为l的对应-r右边界//正数表示为r的对应l左边界
long long BaseConstruct(int n, int m)
{
    A[++n] = 0;          // 为了方便得到最终答案，可以令 A[n + 1] = 0
    B[n].push_back(-n); // 然后要求必须取 [n + 1, n + 1]
    B[n].push_back(n);
    int now = 0;
    for (int i = 1, j = 1; i <= n; i++)
    {
        // 双指针右端点移动一步，增加右端点为 i 且位于 [j, i] 里的需求区间
        for (int x : B[i]) if (x > 0 && x >= j) now++;
        while (now > 0 && j <= i)
        {
            // 双指针左端点移动一步，减少左端点为 j 且位于 [j, i] 里的需求区间
            for (int x : B[j]) if (x < 0 && -x <= i) now--;
            j++;
        }
        assert(now == 0);//成立就捕捉到这种错误，并打印出错误信息
        LIM[i] = j;
    }
    f[0] = 0, f[1] = A[1];
    q[tail++] = 0;
    q[tail++] = 1;
    for (int i = 2; i <= n; i++)
    {
        int lim = LIM[i - 1] - 1;// 要求上一个基站的位置 >= p_{i - 1} - 1
        while (q[head] < lim) head++;
        f[i] = f[q[head]] + A[i];
        while (head < tail && f[q[tail - 1]] >= f[i]) tail--;
```

```
            q[tail++] = i;
        }
        return f[n];
    }
```

区间dp

例题：给定长度为n的数组，前后值一样可以合并：$((n,n)->n+1)$，求可以合并出的最大值.

我们定义$f[i][j]$,里面存的值是左端点为j，能合并出i这个数字的右端点的位置

那么状态转移方程如下:

$$f[i][j]=f[i-1][f[i-1][j]]$$

```
for (int i = 1; i <= n; ++i) dp[a[i]][i] = i + 1; //初始化 题目中(0 <= a[i] <= 40)
for (int i = 1; i <= 58; ++i)
{
    for (int j = 1; j <= n; ++j)
    {
        if (!dp[i][j]) dp[i][j] = dp[i - 1][dp[i - 1][j]];
        if (dp[i][j]) ans = std::max(ans, i);
    }
}
```

树形dp

```
//求dfs序
//求深度
//求树直径
//求树的重心
```

数位dp

数位dp有个通用的套路，就是先采用**前缀和思想**，将求解“$[l,r]$这个区间内的满足约束的数的数量”，转化为“$[1,r]$满足约束的数量 - 区间$[1,l-1]$满足约束的数量“

所以我们最终要求解的问题通通转化为：$[1,x]$中满足约束的数量，或者$[0,x]$中的满足约束的数量（左边界取决于题目）

然后将数字x拆分为一个个数位

> 数位，如个位、十位、百位等，单个**数码**（比如十进制，此处就是指$0\sim9$）在数x中所占据的一个位置

在代码中表现为：

- $a[1\cdots len]$：将数x分解为R进制（一般为十进制或者二进制），用数组存储，$a[i]$表示x在R^{i-1}处的系数
- 即x这个数的长度为len，最高位上的数字为$a[len]$，最低位上的数字为$a[1]$
- 比如十进制数字4321，转化为a数组后，$a_4=4,a_3=3,a_2=2,a_1=1$

以下为记忆化搜索函数dfs的常设定的形参

- pos：int型变量，表示当前枚举的位置，一般从高到低
- $limit$：bool型变量，表示枚举的第pos位是否受到**限制**，
  - 为true表示取的数不能大于$a[pos]$，而只有在$[pos+1,len]$的位置上填写的数都等于$a[i]$时该值才为true
  - 否则表示当前位没有限制，可以取到$[0,R-1]$，因为R进制的数中数位最多能取到的就是$R-1$
- $last$：int型变量，表示上一位（第$pos+1$位）填写的值
  - 往往用于约束了相邻数位之间的关系的题目
- $lead0$：bool型变量，表示是否有**前导零**，即在$len\to(pos+1)$这些位置是不是都是**前导零**
  - 基于常识，我们往往默认一个数没有前导零，也就是最高位不能为0，即不会写为000123，而是写为123
  - 只有没有前导零的时候，才能计算0的贡献。
  - 那么前导零何时跟答案有关？
    - 统计0的出现次数
    - 相邻数字的差值
    - 以最高位为起点确定的奇偶位
- sum：int型变量，表示当前$len\to(pos+1)$的数位和
- r：int型变量，表示整个数前缀取模某个数m的余数
  - 该参数一般会用在：约束中出现了能被m整除
  - 当然也可以拓展为数位和取模的结果
- st：int型变量，用于状态压缩
  - 对一个集合的数在数位上的出现次数的奇偶性有要求时，其二进制形式就可以表示每个数出现的奇偶性

```
ll dfs(int pos, bool limit, int sum) //举例
{
    if(!pos) return sum; //递归边界
    if(!limit && ~f[pos][sum]) return f[pos][sum]; //没限制并且dp值已搜索过
    int up = limit ? a[pos] : 9;  // a[] : (12345 -> a[1] = 1, a[2] = 2.....a[5] = 5)
    ll res = 0;
    for(int i = 0; i <= up; i++) res = (res + dfs(pos - 1, limit && i == up, sum + i)) % md;
    if(!limit) f[pos][sum] = res; // 记搜，可复用
    return res;
}
```

数据结构

树论

二叉排序树

要插入的节点小于当前节点，取左儿子；要插入的节点大于等于当前节点，取右儿子

> ⚠ 全随机数据偏向 $O(logn)$ 级复杂度，对应特殊情况容易退化为 $O(n)$

```
typedef struct BSTNode
{
    int data;
```

```
    struct BSTNode *lchild = nullptr;
    struct BSTNode *rchild = nullptr;
} *BSTree;
void BST_insert(BSTree *h, int &num)
{
    if ((*h) == nullptr)
    {
        *h = new BSTNode;
        (*h)->data = num;
        return;
    }
    if (num < (*h)->data)
        BST_insert(&(*h)->lchild, num);
    else
        BST_insert(&(*h)->rchild, num);
}
//递归查询
bool BST_Recursion_query(BSTree h, int &num)
{
    if (h == nullptr)
        return false;
    if (num < h->data)
        return BST_Recursion_query(h->lchild, num);
    else if (num > h->data)
        return BST_Recursion_query(h->rchild, num);
    else
        return true;
}
//循环查询(约快1/5)
bool BST_Loop_query(BSTree h, int &num)
{
    while (h != nullptr)
    {
        if (num < h->data)
            h = h->lchild;
        else if (num > h->data)
            h = h->rchild;
        else
            return true;
    }
    return true;
}
//生成二叉排序树
BSTree BST_summon()
{
    BSTree BST_h = nullptr;
    for (int i = 1; i <= M; ++i)
        BST_insert(&BST_h, num[i]);
    return BST_h;
}
```

平衡二叉排序树

```
// AVLTree
typedef struct AVLNode
{
    int data, BF = 0;
```

```
    struct AVLNode *lchild = nullptr;
    struct AVLNode *rchild = nullptr;
} *AVLTree;
inline void L_rotate(AVLTree *p)
{
    AVLTree L = new AVLNode;
    L = (*p)->rchild;
    (*p)->rchild = L->lchild;
    L->lchild = (*p);
    *p = L;
}
inline void R_rotate(AVLTree *p)
{
    AVLTree R;
    R = (*p)->lchild;
    (*p)->lchild = R->rchild;
    R->rchild = (*p);
    *p = R;
}
void LeftBalance(AVLTree *h)
{
    AVLTree L, Lr;
    L = (*h)->lchild;
    if (L->BF == 1)
    {
        (*h)->BF = L->BF = 0;
        R_rotate(h);
    }
    else if (L->BF == -1)
    {
        Lr = L->rchild;
        if (Lr->BF == 1)
        {
            (*h)->BF = -1;
            L->BF = 0;
        }
        else if (Lr->BF == 0)
            (*h)->BF = L->BF = 0;
        else if (Lr->BF == -1)
        {
            (*h)->BF = 0;
            L->BF = 1;
        }
        Lr->BF = 0;
        L_rotate(&(*h)->lchild);
        R_rotate(h);
    }
}
void RightBalance(AVLTree *h)
{
    AVLTree R, Rr;
    R = (*h)->rchild;
    if (R->BF == -1)
    {
        (*h)->BF = R->BF = 0;
        L_rotate(h);
    }
    else if (R->BF == 1)
```

```
    {
        Rr = R->lchild;
        if (Rr->BF == 1)
            R->BF = -1;
        else if (Rr->BF == 0)
            R->BF = 0;
        else if (Rr->BF == -1)
            R->BF = 1;
        (*h)->BF = Rr->BF = 0;
        R_rotate(&(*h)->rchild);
        L_rotate(h);
    }
}
bool AVL_insert(AVLTree *h, int num, int *flag)
{
    if ((*h) == nullptr)
    {
        *h = new AVLNode;
        (*h)->data = num;
        (*h)->BF = 0;
        *flag = true;
        return true;
    }
    if (num < (*h)->data)
    {
        if (!AVL_insert(&(*h)->lchild, num, flag))
            return false;
        if (*flag)
        {
            if ((*h)->BF == 1)
                LeftBalance(h);
            else if ((*h)->BF == 0)
                (*h)->BF = 1;
            else if ((*h)->BF == -1)
                (*h)->BF = 0;
        }
    }
    else if ((num > (*h)->data))
    {
        if (!AVL_insert(&(*h)->rchild, num, flag))
            return false;
        if (*flag)
        {
            if ((*h)->BF == 1)
                (*h)->BF = 0;
            else if ((*h)->BF == 0)
                (*h)->BF = -1;
            else if ((*h)->BF == -1)
                RightBalance(h);
        }
    }
    else
        *flag = false;
    return num != (*h)->data;
}
ll AVL_query(AVLTree h, int option)
{
```

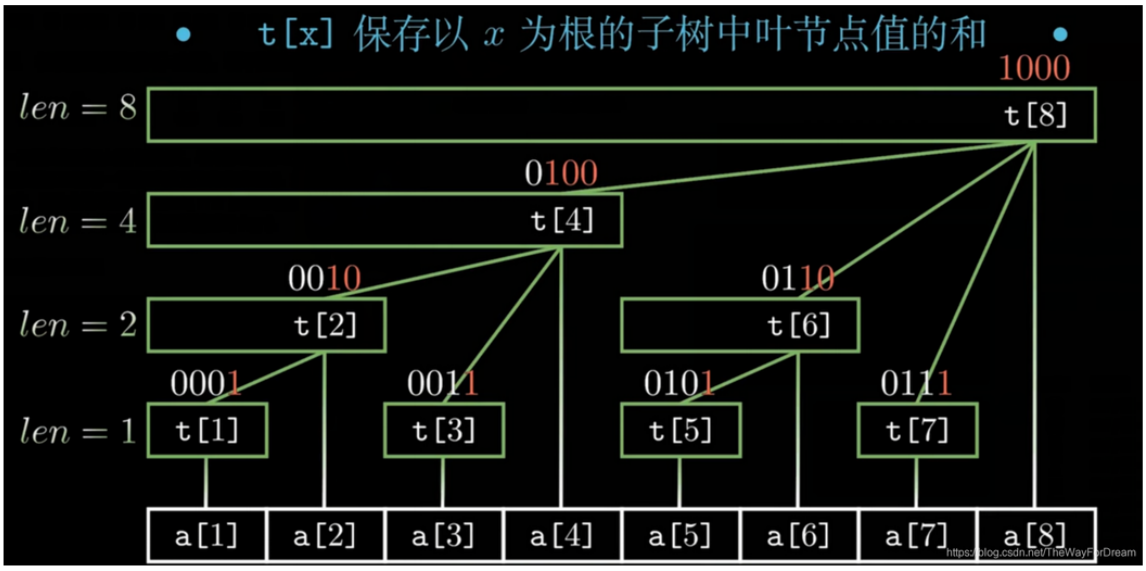
```
    function<bool(AVLTree, int)> AVL_Recursion_query = [&AVL_Recursion_query]
(AVLTree h, int num)
    {
        AVLcnt++;
        if (h == nullptr)
            return false;
        if (num < h->data)
            return AVL_Recursion_query(h->lchild, num);
        else if (num > h->data)
            return AVL_Recursion_query(h->rchild, num);
        else
            return true;
    };
    function<bool(AVLTree, int)> AVL_Loop_query = [](AVLTree h, int num)
    {
        while (h != nullptr)
        {
            AVLcnt++;
            if (num < h->data)
                h = h->lchild;
            else if (num > h->data)
                h = h->rchild;
            else
                return true;
        }
        return true;
    };
    clock_t start = clock();
    if (option == 1)
        for (int i = 1; i <= M; ++i)
            AVL_Recursion_query(h, query[i]);
    else
        for (int i = 1; i <= M; ++i)
            AVL_Loop_query(h, query[i]);
    return clock() - start;
}
AVLTree AVL_summon()
{
    AVLTree AVL_h = nullptr;
    for (int i = 1; i <= N; ++i)N
    {
        int flag = 0;
        AVL_insert(&AVL_h, num[i], &flag);
    }
    return AVL_h;
}
```

字典树Trie

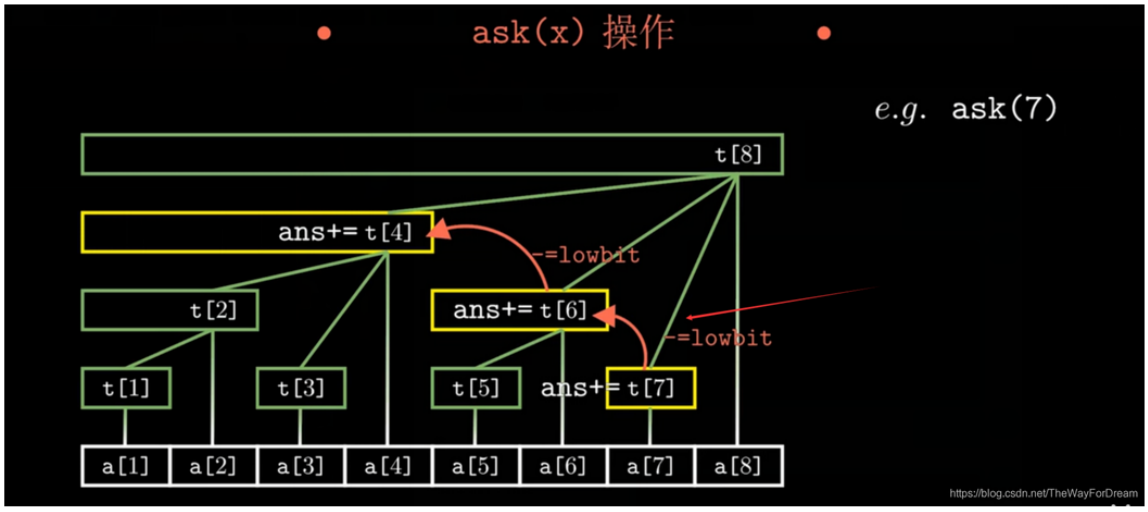
```
struct Trie
{
    vector<Trie *> child;
    Trie() : child(vector<Trie *>(2, NULL)) {}
};
void add(ll x, Trie *root)
{
    for (int i = 31; i >= 0; i--)
```

```
    {
        ll bit = (x >> i) & 1ll;
        if (!root->child[bit])
            root->child[bit] = new Trie();
        root = root->child[bit];
    }
}
//search(~x, root) //查询有多少位不匹配||查询异或为1的位置(两两异或最大值)
//(x ^ search(~x, root)) //得到异或的另一个数
ll search(ll x, Trie *root) //查询有多少位匹配
{
    ll ans = 0;
    for (int i = 31; i >= 0; i--)
    {
        ll bit = (x >> i) & 1ll;
        ans <<= 1;
        if (root->child[bit]) //匹配就将该位设置为1
            ans |= 1, root = root->child[bit]; //ans |= bit;
        else
            root = root->child[!bit]; //ans |= !bit;
    }
    return ans;
}
```

树状数组



我们通过观察节点的二进制数，进一步发现，树状数组中节点x的父节点为**x+lowbit(x),例如t[2]的父节点为t[4]=t[2+lowbit(2)]**



单点修改，区间查询

```
int n, a[N], tr[N];  //tr[x]存储[x-2^k,x]即[x-lowbit(x),x]的和,下标从1开始
int lowbit(int x) { return x & -x; }
void add(int x,int v)  //给x位置+v,修改该点和上枝的值
{
    for(int i = x; i <= n; i += lowbit(i)) tr[i] += v;
}  //变成一个数;要使 a[x] = b,先求 c = s[x] - s[x-1],再 add(x, b-c)
int ask(int x)//查询1~x前缀和
{
    int res = 0;
    for(int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}
inline void build() { for(int i = 1; i <= n; i++) add(i,a[i]); }
```

区间修改，单点查询

对于这一类操作，我们需要构造出原数组的差分数组b，然后用树状数组维护b数组即可

对于区间修改的话，我们只需要对差分数组进行操作即可，例如对区间[L,R]+k,那么我们只需要更新差分数组add(L,k),add(R+1,-k)，这是差分数组的性质.

代码

```
int update(int pos,int k)//pos表示修改点的位置,k表示修改的值也即+k操作
{
    for(int i=pos;i<=n;i+=lowbit(i))
    c[i]+=k;
    return 0;
}
update(L,k);
update(R+1,-k);
```

对于单点查询操作，求出b数组的前缀和即可，因为a[x]=差分数组b[1]+b[2]+…+b[x]的前缀和，这是差分数组的性质之一.

代码

```
ll ask(int pos)//返回区间pos到1的总和
{
    ll ans=0;
    for(int i=pos;i;i-=lowbit(i)) ans+=c[i];
    return ans;
}
```

线段树

> ⚠ 单点修改，区间查询 和 区间修改，区间查询 的函数别混用（一个带mark，一个不带）

```
int tree[N], sum[N << 2], mi[N << 2], ma[N << 2];
void build(int k, int l, int r) // k表示当前结点的编号，l,r为当前结点所代表的区间
{
    if (l == r) // 当前结点为叶子结点
    {
        sum[k] = ma[k] = mi[k] = tree[l]; // 对应区间的最小值为原序列中的对应值
        return;
    }
    int mid = l + r >> 1;
    build(k << 1, l, mid);                  // 构造左子树
    build(k << 1 | 1, mid + 1, r);          // 构造右子树
    sum[k] = sum[k << 1] + sum[k << 1 | 1]; // 更新
    mi[k] = min(mi[k << 1], mi[k << 1 | 1]);
    ma[k] = max(ma[k << 1], ma[k << 1 | 1]);
}
/*单点修改，区间查询*/
```

```
void change(int k, int l, int r, int x, int v) // x为原序列的位置，v为要改成的值
{
    if (r < x || l > x) return;          // 当前区间与原序列的位置完全无交集
    if (l == r && l == x)                // 当前结点为对应的叶子结点
    {
        sum[k] = ma[k] = mi[k] = tree[l] = v; // 修改叶子结点
        return;
    }
    int mid = l + r >> 1;
    change(k << 1, l, mid, x, v);            // 修改左子区间
    change(k << 1 | 1, mid + 1, r, x, v);    // 修改右子区间
    sum[k] = sum[k << 1] + sum[k << 1 | 1]; // 更新
    mi[k] = min(mi[k << 1], mi[k << 1 | 1]);
    ma[k] = max(ma[k << 1], ma[k << 1 | 1]);
}
int queryA(int k, int l, int r, int x, int y) // 区间查询:区间和,不带标记
{
    if (y < l || x > r) return 0;
    if (l >= x && r <= y) return sum[k]; // 查询最值修改一下
    int mid = l + r >> 1;
    return queryA(k << 1, l, mid, x, y) + queryA(k << 1 | 1, mid + 1, r, x, y);
    // return min(queryA(k << 1, l, mid, x, y), queryA(k << 1 | 1, mid + 1, r, x,
y));
    // return max(queryA(k << 1, l, mid, x, y), queryA(k << 1 | 1, mid + 1, r, x,
y));
}
/*区间修改，区间查询*/
int mark[N << 2];
void add(int k, int l, int r, int v) // 给区间[l,r]所有数加上v
{
    mark[k] += v;               // 打标记
    sum[k] += (r - l + 1) * v; // 维护区间和
    mi[k] += v;
    ma[k] += v;
}
void pushdown(int k, int l, int r, int mid) // 标记下传
{
    if (!mark[k]) return;                  // 没有标记则不用考虑
    add(k << 1, l, mid, mark[k]);          // 下传到左子树
    add(k << 1 | 1, mid + 1, r, mark[k]); // 下传到右子树
    mark[k] = 0;                           // 清零
}
void modify(int k, int l, int r, int x, int y, int v) // 给定区间[x,y]所有数加上v
{
    if (l >= x && r <= y)
        return add(k, l, r, v);
    int mid = l + r >> 1;
    pushdown(k, l, r, mid); // 到达每一个结点都要下传标记
    if (x <= mid) modify(k << 1, l, mid, x, y, v);
    if (mid < y)  modify(k << 1 | 1, mid + 1, r, x, y, v);
    sum[k] = sum[k << 1] + sum[k << 1 | 1]; // 下传后更新
    mi[k] = min(mi[k << 1], mi[k << 1 | 1]);
    ma[k] = max(ma[k << 1], ma[k << 1 | 1]);
}
int queryB(int k, int l, int r, int x, int y) // 询问区间[x,y]的和,带标记
{
    if (l >= x && r <= y) return sum[k];
    int mid = l + r >> 1, res = 0;
```

```
    pushdown(k, l, r, mid); // 下传标记
    // 查询最值修改一下, 记得res = +-inf;
    if (x <= mid) res += queryB(k << 1, l, mid, x, y);
    if (mid < y)  res += queryB(k << 1 | 1, mid + 1, r, x, y);
    return res;
}
```

李超线段树

李超线段树

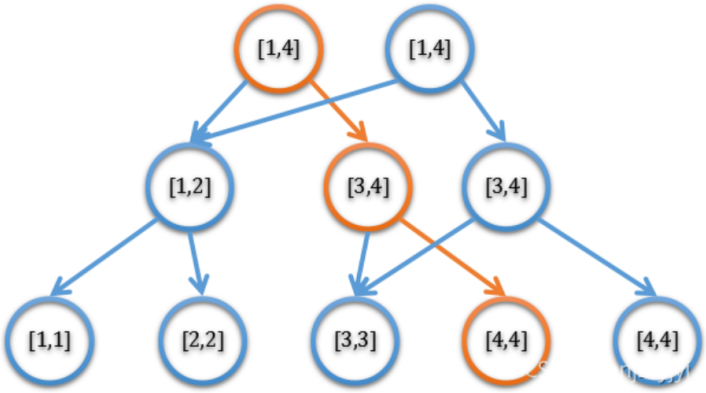
李超线段树是一种用于维护平面直角坐标系内线段关系的数据结构。它常被用来处理这样一种形式的问题：给定一个平面直角坐标系，支持动态插入一条线段，询问从某一个位置 $(X,+\infty)$ 向下看能看到的最高的一条线段（也就是给一条竖线，问这条竖线与所有线段的最高的交点

李超线段树维护的是区间中间 mid 位置的最高线段，将每次询问 (x,inf) 变为单点查询

```
int lastans = 0;
double h = -inf;
struct Line
{
    double k, b;
    int l, r, flag, idx;
    Line() {}
    Line(double k, double b, int l, int r, int flag, int idx)
    {
        this->k = k, this->b = b, this->l = l, this->r = r, this->flag = flag,
this->idx = idx;
    }
    double calc(int pos) { return k * pos + b; }
 // 直线里面pos位置的y值
    int cross(const Line &rhs) const { return floor((b - rhs.b) / (rhs.k - k));
} // 线段交点的x坐标
} seg[N << 1];
inline void build(int rt, int l, int r)
{
    seg[rt] = (Line){0.0, 0.0, l, r, 0, 0};
    if (l == r) return;
    build(rt << 1, l, (l + r) >> 1);
    build(rt << 1 | 1, ((l + r) >> 1) + 1, r);
}
inline void update(int cutl, int cutr, int rt, Line rhs)
{
    if (rhs.l <= cutl && rhs.r >= cutr)
    {
        if (!seg[rt].flag) seg[rt] = rhs, seg[rt].flag = 1;
        else if (rhs.calc(cutl)-seg[rt].calc(cutl)>eps && rhs.calc(cutr)-
seg[rt].calc(cutr)>eps)
            seg[rt] = rhs;
        else if (rhs.calc(cutl)-seg[rt].calc(cutl)>eps || rhs.calc(cutr)-
seg[rt].calc(cutr)>eps)
        {
            int Mid = (cutl + cutr) >> 1;
            if (rhs.calc(Mid) - seg[rt].calc(Mid) > eps)
                swap(rhs, seg[rt]);
            if (rhs.cross(seg[rt]) - Mid < -eps)
                update(cutl, Mid, rt << 1, rhs);
            else
```

```
                update(Mid + 1, cutr, rt << 1 | 1, rhs);
        }
    }
    else
    {
        int Mid = (cutl + cutr) >> 1;
        if (rhs.l <= Mid) update(cutl, Mid, rt << 1, rhs);
        if (rhs.r > Mid)  update(Mid + 1, cutr, rt << 1 | 1, rhs);
    }
}
inline void ask(int rt, int l, int r, int x)
{
    double tmp = seg[rt].calc(x);
    if (tmp - h > eps) lastans = seg[rt].idx, h = tmp;
    else if (fabs(h - tmp) < eps) lastans = min(lastans, seg[rt].idx);
    if (l == r) return;
    int mid = l + r >> 1;
    if (x <= mid) ask(rt << 1, l, mid, x);
    if (x > mid) ask(rt << 1 | 1, mid + 1, r, x);
}
```

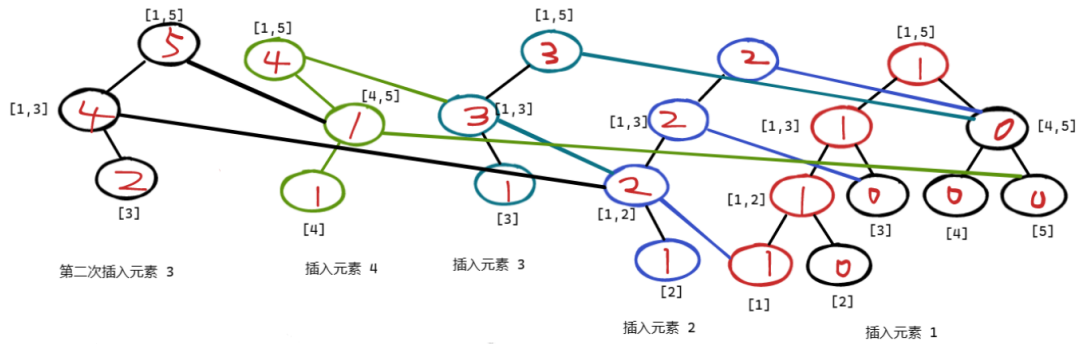
可持久化线段树



[静态可持久化线段树]（主席树）

只添加链，不适用新建权值线段树的方式创建的主席树：

- 这种只添加链的方式构建的线段树就是**主席树**




```
int a[N], root[N], cnt, n;
vector<int> v;
struct node
{
    int left;
```

```
    int right;
    int sum;
} hjt[N << 5];
int getid(int x) { return lower_bound(v.begin(),v.end(),x) - v.begin() + 1; }
// Note that cur is used with &cur, for it will change
void insert(int left, int right, int pre, int &cur, int val)
{
    hjt[++cnt] = hjt[pre]; // Assign position of the left and right subtrees to
new segment tree
    cur = cnt;
    hjt[cur].sum++; // insert operation, total record +1
    if (left == right) return;
    int mid = left + right >> 1;
    if (val <= mid) insert(left, mid, hjt[pre].left, hjt[cur].left, val);
    else insert(mid + 1, right, hjt[pre].right, hjt[cur].right, val);
}
int query(int left, int right, int L, int R, int k)
{
    if (left == right) return left;
    int mid = left + right >> 1;
    int t = hjt[hjt[R].left].sum - hjt[hjt[L].left].sum;
    if (t >= k)
        return query(left, mid, hjt[L].left, hjt[R].left, k); // find kth
smallest number in the left subtree
    else
        return query(mid + 1, right, hjt[L].right, hjt[R].right, k - t); //
Otherwise go to the right subtree and look for (k-t)th smallest number
}//The query function returns the index
// 查询[x, y]的第k小值:a[query(1, n, root[x - 1], root[y], k) - 1];
void build()
{
    for (int i = 1; i <= n; ++i) v.push_back(a[i]);
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    for (int i = 1; i <= n; i++) insert(1, n, root[i - 1], root[i],
getid(a[i]));
}
```

树套树

单点修改线段树套multiset
用multiset降低空间使用，但增加时间复杂度

```
struct node
{
    int l, r;
    multiset<int> s;
}tr[N * 4];
void build(int u, int l, int r) //u为根节点，一般为1
{
    tr[u] = {l, r};
    tr[u].s.insert(inf); tr[u].s.insert(-inf);
    for (int i = l; i <= r; ++i) tr[u].s.insert(num[i]);
    if (l == r) return;
    int mid = l + r >> 1;
    build(u << 1, l, mid); build(u << 1 | 1, mid + 1, r);
}
```

```
void modify(int u, int p, int x) //将num[p]修改为x
{
    tr[u].s.erase(tr[u].s.find(num[p]));
    tr[u].s.insert(x);
    if (tr[u].l == tr[u].r) return;
    int mid = tr[u].l + tr[u].r >> 1;
    if (p <= mid) modify(u << 1, p, x);
    else modify(u << 1 | 1, p, x);
}
int query(int u, int l, int r, int x) //查询[l, r]中x的非严格后继数
{
    if (tr[u].l >= l && tr[u].r <= r)
    {
        auto it = tr[u].s.lower_bound(x);
        return *(it);
    }
    int mid = tr[u].l + tr[u].r >> 1, res = inf;
    if (mid >= l) res = min(res, query(u << 1, l, r, x));
    if (mid < r) res = min(res, query(u << 1 | 1, l, r, x));
    return res;
}
```

线段树套splay树

```
struct node {
    int s[2], p, v;
    int size;
    void init(int _v, int _p) {
        v = _v; p = _p;
        size = 1;
    }
}tr[N<<4];
int L[N<<4], R[N<<4], T[N<<4], w[N<<4], idx; //w存放初始数组
//数组开大一点
void pushup(int u)
{
    tr[u].size = tr[tr[u].s[0]].size + tr[tr[u].s[1]].size + 1;
}
void rotate(int x)
{
    int y = tr[x].p, z = tr[y].p;
    int k = tr[y].s[1] == x;
    tr[z].s[tr[z].s[1] == y] = x; tr[x].p = z;
    tr[y].s[k] = tr[x].s[k ^ 1]; tr[tr[x].s[k ^ 1]].p = y;
    tr[x].s[k ^ 1] = y; tr[y].p = x;
    pushup(y); pushup(x);
}
void splay(int &root, int x, int k)
{
    while (tr[x].p != k)
    {
        int y = tr[x].p, z = tr[y].p;
        if (z != k)
            if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
            else rotate(y);
        rotate(x);
    }
```

```
    if (!k) root = x;
}
void insert(int& root, int v)
{
    int u = root, p = 0;
    while (u) p = u, u = tr[u].s[tr[u].v < v];
    u = ++idx;
    if (p) tr[p].s[tr[p].v < v] = u;
    tr[u].init(v, p);
    splay(root, u, 0);
}
int get_k(int root, int x) //查找平衡树root的x的排名/*O(log^2(n))*/
{
    int u = root, res = 0;
    while (u)
        if (tr[u].v < x) res += tr[tr[u].s[0]].size + 1, u = tr[u].s[1];
        else u = tr[u].s[0];
    return res;
}
void update(int &root, int x, int y)
{
    int u = root;
    while (u)
    {
        if (tr[u].v == x) break;
        u = tr[u].s[tr[u].v < x];
    }
    splay(root, u, 0);
    int l = tr[u].s[0], r = tr[u].s[1];
    while (tr[l].s[1]) l = tr[l].s[1];
    while (tr[r].s[0]) r = tr[r].s[0];
    splay(root, l, 0); splay(root, r, l);
    tr[r].s[0] = 0;
    pushup(r); pushup(l);
    insert(root, y);
}
void build(int u, int l, int r) //u为根节点，一般为1
{
    L[u] = l; R[u] = r;
    insert(T[u], -inf); insert(T[u], inf);
    for (int i = l; i <= r; i++) insert(T[u], w[i]);
    if (l == r) return;
    int mid = l + r >> 1;
    build(u << 1, l, mid); build(u << 1 | 1, mid + 1, r);
}
int query(int u, int l, int r, int x)  //查询区间[l, r]里x的小于x的数的个数
{
    if (L[u] >= l && R[u] <= r) return get_k(T[u], x) - 1;
    int res = 0, mid = L[u] + R[u] >> 1;
    if (l <= mid) res += query(u << 1, l, r, x);
    if (r > mid) res += query(u << 1 | 1, l, r, x);
    return res;
}
int query_k(int u, int a, int b, int k) //查询区间[l, r]里排名为k的数/*O(log^3(n))*/
{
    int l = 0, r = 1e8;
    while (l < r)
    {
```

```
        int mid = l + r + 1 >> 1;
        if (query(u, a, b, mid) + 1 <= k) l = mid;
        else r = mid - 1;
    }
    return r;
}
void modify(int u, int p, int x) //修改w[p]的值为x/*O(log^2(n))*/
{
    update(T[u], w[p], x);
    if (L[u] == R[u]) return;
    int mid = L[u] + R[u] >> 1;
    if (p <= mid) modify(u << 1, p, x);
    else modify(u << 1 | 1, p, x);
}
int get_suc(int root, int x)
{
    int u = root, res = inf;
    while(u)
        if (tr[u].v > x) res = min(res, tr[u].v), u = tr[u].s[0];
        else u = tr[u].s[1];
    return res;
}
int get_pre(int root, int x)
{
    int u = root, res = -inf;
    while(u)
        if (tr[u].v < x) res = max(res, tr[u].v), u = tr[u].s[1];
        else u = tr[u].s[0];
    return res;
}
int query_pre(int u, int l, int r, int x) //查询x的前驱(严格小于x且最
大)/*O(log^2(n))*/
{
    if (L[u] >= l && R[u] <= r) return get_pre(T[u], x);
    int mid = L[u] + R[u] >> 1, res = -inf;
    if (l <= mid) res = max(res, query_pre(u << 1, l, r, x));
    if (r > mid) res = max(res, query_pre(u << 1 | 1, l, r, x));
    return res;
}
int query_suc(int u, int l, int r, int x) //查询x的后继(严格大于x且最
小)/*O(log^2(n))*/
{
    if (L[u] >= l && R[u] <= r) return get_suc(T[u], x);
    int mid = L[u] + R[u] >> 1, res = inf;
    if (l <= mid) res = min(res, query_suc(u << 1, l, r, x));
    if (r > mid) res = min(res, query_suc(u << 1 | 1, l, r, x));
    return res;
}
```

二维树状数组

单点修改+区间查询

```
int lowbit(int x) { return x&-x; }
void add(int x,int y,int v)
{
    while(x<=n)
```

```
    {
        int ty=y;
        while(ty<=n) tree[x][ty] += v, ty += lowbit(ty);
        x+=lowbit(x);
    }
}
int ask(int x, int y)
{
    int res = 0;
    while(x)
    {
        int ty = y;
        while(ty) res += tree[x][ty],ty -= lowbit(ty);
        x -= lowbit(x);
    }
    return res;
}
```

区间修改+单点查询

```
void add(int x,int y,int v)
{
    while(x<=n)
    {
        int ty = y;
        while(ty<=n) tree[x][ty] += v,ty += lowbit(ty);
        x += lowbit(x);
    }
}
void real_add(int x1, int y1, int x2, int y2, int v)
{
    add(x1, y1, v);
    add(x1, y2+1, -v);
    add(x2+1, y1, -v);
    add(x2+1, y2+1, v);
}
int ask(int x, int y)
{
    int res=0;
    while(x)
    {
        int ty = y;
        while(ty) res += tree[x][ty], ty -= lowbit(ty);
        x -= lowbit(x);
    }
    return res;
}
```

分块套树状数组

统计二维矩阵区间内点的个数

> ⚠ 限制条件：一个x只能对应一个y，同一个y可以对应多个x，同高中函数定义
>
> （例：add（1，2，1）再 add（1，3，1）是错误的，此时 x[1] =2 or 3）

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

```
const int M = sqrt(N) + 5;
int subn, size, cnt, id[N], L[N], R[N], T[M][N], posx[N];
int lb(int x) { return x & -x; }
void build(int n)
{
    subn = n;
    size = sqrt(subn);
    cnt = subn / size;
    for (int i = 1; i <= cnt; ++i)
    {
        L[i] = R[i - 1] + 1;
        R[i] = i * size;
    }
    if (R[cnt] < subn)
    {
        ++cnt;
        L[cnt] = R[cnt - 1] + 1;
        R[cnt] = subn;
    }
    for (int j = 1; j <= cnt; ++j)
        for (int i = L[j]; i <= R[j]; ++i)
            id[i] = j;
}
void add(int p, int v, int d)
{
    posx[p] = v;
    for (int i = id[p]; i <= cnt; i += lb(i))
        for (int j = v; j <= subn; j += lb(j))
            T[i][j] += d;
}
int getsum(int p, int v)
{
    if (!p)
        return 0;
    int res = 0;
    int idx = id[p];
    for (int i = L[idx]; i <= p; ++i)
        if (posx[i] <= v)
            ++res;
    for (int i = idx - 1; i; i -= lb(i))
        for (int j = v; j; j -= lb(j))
            res += T[i][j];
    return res;
}
int query(int lx, int rx, int ly, int ry)
{
    int res = getsum(rx, ry) - getsum(rx, ly - 1) - getsum(lx - 1, ry) +
getsum(lx - 1, ly - 1);
    return res;
}
```

二维线段树

单点修改，区间查询

```
ll MAX[N<<2][N<<2], MIN[N<<2][N<<2], minV = inf, maxV = -inf; // 维护最值
ll a[N<<2][N<<2];//初始矩阵
```

```
ll SUM[N<<2][N<<2], sumV;//维护求和
int n;
void pushupX(int deep, int rt)
{
    MAX[deep][rt] = max(MAX[deep << 1][rt], MAX[deep << 1 | 1][rt]);
    MIN[deep][rt] = min(MIN[deep << 1][rt], MIN[deep << 1 | 1][rt]);
    SUM[deep][rt] = SUM[deep<<1][rt] + SUM[deep<<1|1][rt];
}
void pushupY(int deep, int rt)
{
    MAX[deep][rt] = max(MAX[deep][rt << 1], MAX[deep][rt << 1 | 1]);
    MIN[deep][rt] = min(MIN[deep][rt << 1], MIN[deep][rt << 1 | 1]);
    SUM[deep][rt] = SUM[deep][rt<<1] + SUM[deep][rt<<1|1];
}
void buildY(int ly, int ry, int deep, int rt, int flag)
{
    //y轴范围ly,ry;deep,rt;标记flag
    if (ly == ry)
    {
        if (flag!=-1) MAX[deep][rt] = MIN[deep][rt] = SUM[deep][rt] = a[flag]
[ly];
        else pushupX(deep, rt);
        return;
    }
    int m = (ly + ry) >> 1;
    buildY(ly, m, deep, rt << 1, flag);
    buildY(m + 1, ry, deep, rt << 1 | 1, flag);
    pushupY(deep, rt);
}
void buildX(int lx, int rx, int deep)
{
    //建树x轴范围lx,rx;deep
    if (lx == rx)
    {
        buildY(1, n, deep, 1, lx);
        return;
    }
    int m = (lx + rx) >> 1;
    buildX(lx, m, deep << 1);
    buildX(m + 1, rx, deep << 1 | 1);
    buildY(1, n, deep, 1, -1);
}
void updateY(int Y, int val, int ly, int ry, int deep, int rt, int flag)
{
    //单点更新y坐标;更新值val;当前操作y的范围ly,ry;deep,rt;标记flag
    if (ly == ry)
    {
        if (flag) //注意读清楚题意，看是单点修改值还是单点加值
            MAX[deep][rt] = MIN[deep][rt] = SUM[deep][rt] = val;
        else pushupX(deep, rt);
        return;
    }
    int m = (ly + ry) >> 1;
    if (Y <= m) updateY(Y, val, ly, m, deep, rt << 1, flag);
    else updateY(Y, val, m + 1, ry, deep, rt << 1 | 1, flag);
    pushupY(deep, rt);
}
void updateX(int X, int Y, int val, int lx, int rx, int deep)
```

```
{
    //单点更新范围x,y;更新值val;当前操作x的范围lx,rx;deep
    if (lx == rx)
    {
        updateY(Y, val, 1, n, deep, 1, 1);
        return;
    }
    int m = (lx + rx) >> 1;
    if (X <= m) updateX(X, Y, val, lx, m, deep << 1);
    else updateX(X, Y, val, m + 1, rx, deep << 1 | 1);
    updateY(Y, val, 1, n, deep, 1, 0);
}
void queryY(int Yl, int Yr, int ly, int ry, int deep, int rt)
{
    //询问区间y轴范围y1,y2;当前操作y的范围ly,ry;deep,rt
    if (Yl <= ly && ry <= Yr)
    {
        minV = min(MIN[deep][rt], minV);
        maxV = max(MAX[deep][rt], maxV);
        sumV += SUM[deep][rt];
        return;
    }
    int m = (ly + ry) >> 1;
    if (Yl <= m) queryY(Yl, Yr, ly, m, deep, rt << 1);
    if (m < Yr) queryY(Yl, Yr, m + 1, ry, deep, rt << 1 | 1);
}
void queryX(int Xl, int Xr, int Yl, int Yr, int lx, int rx, int rt)
{
    //询问区间范围x1,x2,y1,y2;当前操作x的范围lx,rx;rt
    if (Xl <= lx && rx <= Xr)
    {
        queryY(Yl, Yr, 1, n, rt, 1);
        return;
    }
    int m = (lx + rx) >> 1;
    if (Xl <= m) queryX(Xl, Xr, Yl, Yr, lx, m, rt << 1);
    if (m < Xr) queryX(Xl, Xr, Yl, Yr, m + 1, rx, rt << 1 | 1);
}
inline void query(int x1, int x2, int y1, int y2)
{
    minV = inf, maxV = -inf, sumV = 0;
    queryX(x1, x2, y1, y2, 1, n, 1);
}
inline void modify(int x, int y, ll val)
{
    a[x][y] = val; // 注意读清楚题意，看是单点修改值还是单点加值
    updateX(x, y, val, 1, n, 1);
}
```

矩阵线段树

二维线段树

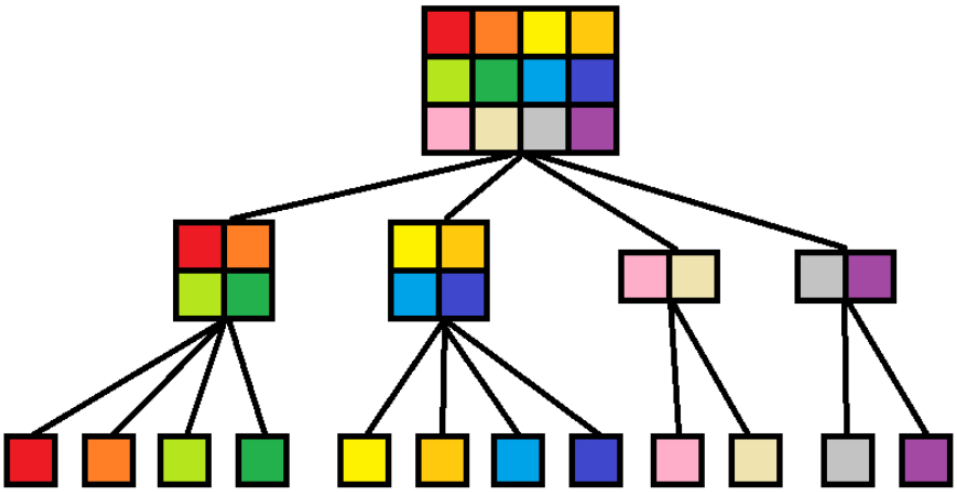
二维线段树最主要用于平面统计问题。类似一维线段树，最经典的就是求区间最值（或区间和），推广到二维，求得就是矩形区域最值（或矩形区域和），对于矩形区域和，二维树状数组更加高效，而矩形区域最值，更加高效的方法是二维RMQ，但是二维RMQ不支持动态更新，所以二维线段树还是有用武之地的。

如果对一维线段树已经驾轻就熟，那么直接来看下面两段对比，就可以轻松理解二维线段树了。

一维线段树是一棵 二叉树，树上每个结点保存 一个区间和一个域，非叶子结点一定有 两个儿子结点，儿子结点表示的 两个区间交集为空，并集为父结点表示的 区间；叶子结点的表示区间长度为1，即单位长度；域则表示了需要求的数据，每个父结点的域可以通过 两个儿子结点得出。

二维线段树是一棵 四叉树，树上每个结点保存 一个矩形和一个域，非叶子结点一定有 二或四 个儿子结点，儿子结点表示的 四个矩形交集为空，并集为父结点表示的 矩形；叶子结点表示的矩形长宽均为1，域则表示了需要求的数据，每个父结点的域可以通过 四个儿子结点得出。

一个4x3的矩形，可以用图1的树形结构来表示，给每个单位方块标上不同的颜色易于理解。



```
//建树 n*m: 1e6，2e4次区间修改，2e4次区间查询，O2跑950ms
struct dataInfo // 最值信息
{
    int posx, posy, val;
    dataInfo() : posx(-1), posy(-1), val(-1) {}
    dataInfo(int _x, int _y, ll _v) : posx(_x), posy(_y), val(_v) {}
};
// 线段树结点信息
struct treeNode
{
    dataInfo maxv, minv;
    inline void reset()
    {
        maxv = dataInfo(0, 0, -inf);
        minv = dataInfo(0, 0, inf);
    }
    inline void nonexist() { maxv = minv = dataInfo(-1, -1, -1); }
    inline int modify(int _val) {  return maxv.val = minv.val = _val; }
} node[N * N << 1];
int lazy[N * N << 1]; //memset(lazy, 0x3f, sizeof(lazy)); //使用记得初始化
// 注意，这里需要返回指针，因为后续使用需要对这个结点进行修改
struct Interval
{
    int l, r;
    Interval() {}
    Interval(int _l, int _r) : l(_l), r(_r) {}
    inline int mid() { return (l + r) >> 1; }                    // 区间中点
    inline int len() { return r - l + 1; }                       // 区间长度
```

```
    inline Interval left() { return Interval(l, mid()); }      // 左半区间
    inline Interval right() { return Interval(mid() + 1, r); } // 右半区间
    // 区间判交
    inline bool isIntersect(Interval &tarI) { return !(l > tarI.r || r <
tarI.l); }
    // 区间判包含
    inline bool isInclude(Interval &tarI) { return l <= tarI.l && tarI.r <= r; }
    inline bool in(int v) { return l <= v && v <= r; }
};
inline int son(int p, int x) { return p * 4 - 2 + x; }
inline void push_down(int p, Interval xI, Interval yI)
{
    lazy[son(p, 0)] = node[son(p, 0)].modify(lazy[p]);
    if (xI.right().len() > 0 && yI.left().len() > 0)
        lazy[son(p, 1)] = node[son(p, 1)].modify(lazy[p]);
    if (xI.left().len() > 0 && yI.right().len() > 0)
        lazy[son(p, 2)] = node[son(p, 2)].modify(lazy[p]);
    if (xI.right().len() > 0 && yI.right().len() > 0)
        lazy[son(p, 3)] = node[son(p, 3)].modify(lazy[p]);
    lazy[p] = inf;
}
int build_segtree(int p, Interval xI, Interval yI)
{
    if (xI.len() <= 0 || yI.len() <= 0) return 0; // 空矩形
    treeNode *now = &node[p];
    if (xI.len() == 1 && yI.len() == 1)
    {
        now->maxv = now->minv = dataInfo(xI.l, yI.l, 0);
        return 1;
    } // 单位矩形
    int isvalid[4];
    isvalid[0] = build_segtree(son(p, 0), xI.left(), yI.left());
    isvalid[1] = build_segtree(son(p, 1), xI.right(), yI.left());
    isvalid[2] = build_segtree(son(p, 2), xI.left(), yI.right());
    isvalid[3] = build_segtree(son(p, 3), xI.right(), yI.right());
    now->reset(); // 结点初始化
    for (int i = 0; i < 4; i++)
    {
        if (!isvalid[i]) continue;
        treeNode *sonNode = &node[son(p, i)];
        now->maxv = sonNode->maxv.val > now->maxv.val ? sonNode->maxv : now-
>maxv;
        now->minv = sonNode->minv.val < now->minv.val ? sonNode->minv : now-
>minv;
    }
    return 1;
}
int insert_segtree(int p, Interval xI, Interval yI, Interval tarXI, Interval
tarYI, ll val)
{
    if (xI.len() <= 0 || yI.len() <= 0) return 0;
    if (!tarXI.isIntersect(xI) || !tarYI.isIntersect(yI)) return 1;
    treeNode *now = &node[p];
    if (tarXI.isInclude(xI) && tarYI.isInclude(yI))
    {
        dataInfo tmp(xI.l, yI.l, val);
        now->maxv = now->minv = tmp;
        lazy[p] = val;
```

```
        return 1;
    }
    if (lazy[p] != inf) push_down(p, xI, yI);
    int isvalid[4];
    isvalid[0] = insert_segtree(son(p, 0), xI.left(), yI.left(), tarXI, tarYI,
val);
    isvalid[1] = insert_segtree(son(p, 1), xI.right(), yI.left(), tarXI, tarYI,
val);
    isvalid[2] = insert_segtree(son(p, 2), xI.left(), yI.right(), tarXI, tarYI,
val);
    isvalid[3] = insert_segtree(son(p, 3), xI.right(), yI.right(), tarXI, tarYI,
val);
    now->reset();
    for (int i = 0; i < 4; i++)
    {
        if (!isvalid[i]) continue;
        treeNode *sonNode = &node[son(p, i)];
        now->maxv = sonNode->maxv.val > now->maxv.val ? sonNode->maxv : now-
>maxv;
        now->minv = sonNode->minv.val < now->minv.val ? sonNode->minv : now-
>minv;
    }
    return 1;
}
void query_segtree(int p, Interval xI, Interval yI, Interval tarXI, Interval
tarYI, treeNode &ans)
{
    if (xI.len() <= 0 || yI.len() <= 0) return;
    if (!tarXI.isIntersect(xI) || !tarYI.isIntersect(yI)) return;
    treeNode *now = &node[p];
    if (ans.minv.val <= now->minv.val && ans.maxv.val >= now->maxv.val) return;
// 最值优化
    if (tarXI.isInclude(xI) && tarYI.isInclude(yI))
    {
        ans.minv = ans.minv.val < now->minv.val ? ans.minv : now->minv;
        ans.maxv = ans.maxv.val > now->maxv.val ? ans.maxv : now->maxv;
        return;
    }
    if (lazy[p] != inf) push_down(p, xI, yI);
    query_segtree(son(p, 0), xI.left(), yI.left(), tarXI, tarYI, ans);
    query_segtree(son(p, 1), xI.right(), yI.left(), tarXI, tarYI, ans);
    query_segtree(son(p, 2), xI.left(), yI.right(), tarXI, tarYI, ans);
    query_segtree(son(p, 3), xI.right(), yI.right(), tarXI, tarYI, ans);
}
```

珂朵莉树Chtholly

什么是珂朵莉树?

珂朵莉树是一种以近乎暴力的形式存储区间信息的一个数据结构。方式是通过set存放若干个用结构体表示的区间，每个区间的元素都是相同的。

$O(nlog^2n)$级别

珂朵莉树的用途?

只要是涉及到区间赋值操作的题，就可以用珂朵莉树处理几乎任何关于区间信息的询问

什么情况下可以用珂朵莉树而不被卡?

珂朵莉树是一种优美的暴力，他的优美是建立在区间的合并操作上，即区间赋值，那么如果构造出一组数据使得其几乎不含区间赋值操作，那珂朵莉树就会被轻易的卡掉

所以珂朵莉树要求题目必须存在区间赋值操作，且数据有高度的随机性

```
struct Chtholly //[l, r]闭区间
{
    ll l, r;
    mutable ll v;
    Chtholly(ll L, ll R = -1, ll V = 0) : l(L), r(R), v(V) {}
    bool operator < (const Chtholly &x)const{ return l < x.l; } //按左端点排序
};
using It = set<Chtholly>::iterator;
set<Chtholly>tr;
It split(int pos) //分割区间
{
    It it = tr.lower_bound(Chtholly(pos)); //找到所需的pos的迭代器
    if(it != tr.end() && it->l == pos) return it; //看看这个迭代器的l是不是所需要的pos，是的话直接返回就行
    --it; //不是的话就说明肯定是在前一个里面
    ll l = it->l, r = it->r, v = it->v;
    tr.erase(it);
    tr.insert(Chtholly(l, pos - 1, v)); //拆分成两个区间[l, pos),[pos, r] //[pos, r]就是闭区间
    return tr.insert(Chtholly(pos, r, v)).first; //返回以pos开头的区间的迭代器
}
void emerge(int l, int r, ll x) //合并区间||区间修改
{
    It itr = split(r + 1), itl = split(l); //先找到r+1的迭代器位置，再找l的迭代器位置
    tr.erase(itl, itr); //删掉这一段迭代器
    tr.insert(Chtholly(l, r, x)); //重新插入所需区间
}
void delta(int l, int r, ll x) //区间值加减,O(m),m为[l,r]内区间个数
{
    It itr = split(r + 1), itl = split(l);
    for(;itl != itr; ++itl) itl->v += x;
}
ll query(ll l, ll r, ll pos) //单点查询
{
    It it = tr.lower_bound(Chtholly(pos));
    if(it != tr.end() && it->l == pos) return it.v;
    return (--it).v;
}
ll query_k(ll l, ll r, ll k) //查询区间[l, r]里排名为k的数,O(mlogm),m为[l,r]内区间个数
{
    It itr = split(r + 1), itl = split(l);
    vector<pair<ll, ll>> tmp;
    for (; itl != itr; ++itl) tmp.pb(make_pair(itl->v, itl->r - itl->l + 1));
    sort(tmp.begin(), tmp.end());
    for (auto it : tmp) if ((k -= it.se) <= 0) return it.fi;
```

```
    return -1; //没k个数返回-1
}
```

KD-Tree

kd-tree（全称为k-dimensional tree），它是一种分割k维数据空间的点，并进行存储的数据结构；在计算机科学里，kd-tree是在k维欧几里德空间组织点的数据结构。Kd-tree是二进制空间分割数的特殊情况，常应用于多为空间关键数据的搜索，例如范围搜索和最近邻搜索。

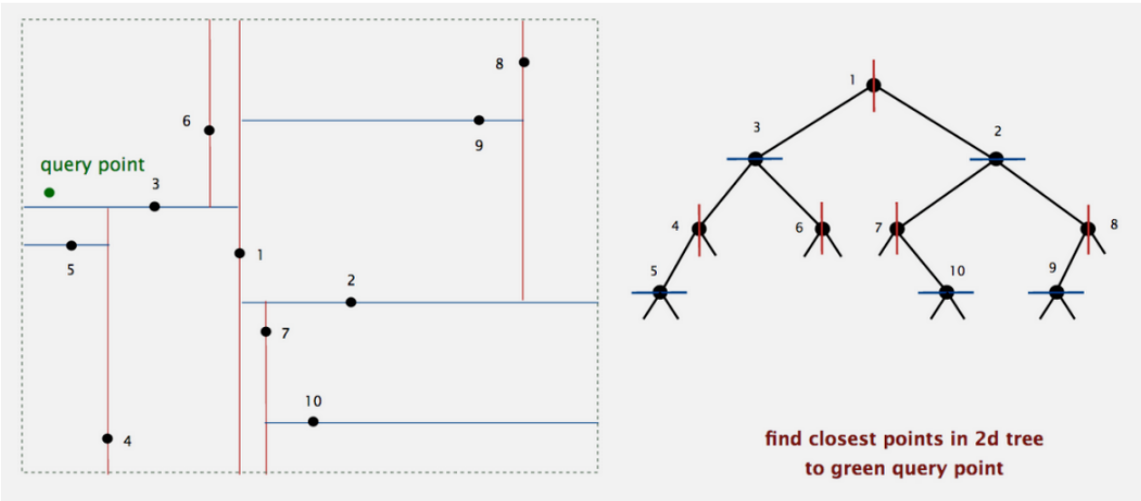
kd-tree的每个节点都是k维点的 二叉树。所有的非叶子节点可以看做为将一个空间分割成两个半空间的超平面。节点左边的子树代表在超平面左边的点（即在分割的维度上小于超平面的点集合），节点右边的子树代表在超平面右边的点（即在分割的维度上大于超平面的点集合）。

从k-d树节点的数据类型的描述可以看出构建k-d树是一个逐级展开的递归过程。构建k-d树伪码。

> **输入**：数据点集Data-set和其所在的空间Range
> **输出**：Kd，类型为k-d tree
> 1.If Data-set为空，则返回空的k-d tree
>
> 2. 调用节点生成程序
> 3. 确定split域：对于所有描述子数据(特征矢量)，统计它们在每个维上的数据方差。以SURF特征为例，描述子为64维，可计算64个方差。挑选出最大值，对应的维就是split域的值。数据方差大表明沿该坐标轴方向上的数据分散得比较开，在这个方向上进行数据分割有较好的分辨率；
> 4. 确定Node-data域：数据点集Data-set按其第split域的值排序。位于正中间的那个数据点被选为Node-data。此时新的Data-set' = Data-set\Node-data（除去其中Node-data这一点）。
> 5. dataleft = {d属于Data-set' && d[split] ≤ Node-data[split]}
> Left_Range = {Range && dataleft}
> dataright = {d属于Data-set' && d[split] > Node-data[split]}
> Right_Range = {Range && dataright}
> 6. left = 由（dataleft，Left_Range）建立的k-d tree，即递归调用createKDTree（dataleft，Left_Range）。并设置left的parent域为Kd；
> right = 由（dataright，Right_Range）建立的k-d tree，即调用createKDTree（dataleft，Left_Range）。并设置right的parent域为Kd。

k-d树是一个二叉树，每个节点表示一个空间范围。下表给出的是k-d树每个节点中主要包含的数据结构。

域名	数据类型	描述
Node-data	数据矢量	数据集中某个数据点，是n维矢量（这里也就是k维）
Range	空间矢量	该节点所代表的空间范围
split	整数	垂直于分割超平面的方向轴序号
Left	k-d树	由位于该节点分割超平面左子空间内所有数据点所构成的k-d树
Right	k-d树	由位于该节点分割超平面右子空间内所有数据点所构成的k-d树
parent	k-d树	父节点




```
//超维要手动添维，这是二维模板
```

```
struct point { double x = 0, y = 0; };
struct Tnode
{
    int split;
    struct point dom_elt;
    struct Tnode *left, *right;
};
bool cmpx(point a, point b) { return a.x < b.x; }
bool cmpy(point a, point b) { return a.y < b.y; }
bool equal(point a, point b) { return (a.x == b.x && a.y == b.y); }
void ChooseSplit(point exm_set[], int size, int &split, point &SplitChoice)
{
    double tmp1, tmp2;
    tmp1 = tmp2 = 0;
    for (int i = 0; i < size; ++i)
    {
        tmp1 += 1.0 / (double)size * exm_set[i].x * exm_set[i].x;
        tmp2 += 1.0 / (double)size * exm_set[i].x;
    }
    double v1 = tmp1 - tmp2 * tmp2; // 计算X维度的方差
    tmp1 = tmp2 = 0;
    for (int i = 0; i < size; ++i)
    {
        tmp1 += 1.0 / (double)size * exm_set[i].y * exm_set[i].y;
        tmp2 += 1.0 / (double)size * exm_set[i].y;
    }
    double v2 = tmp1 - tmp2 * tmp2; // 计算Y维度的方差
    split = v1 > v2 ? 0 : 1; // set the split dimension
    if (!split) sort(exm_set, exm_set + size, cmpx);
    else sort(exm_set, exm_set + size, cmpy);
    SplitChoice.x = exm_set[size / 2].x;
    SplitChoice.y = exm_set[size / 2].y;
}
Tnode *build_kdtree(point exm_set[], int size, Tnode *T)
{
    if (size == 0) return NULL;
    else
    {
        int split;
        point dom_elt;
        ChooseSplit(exm_set, size, split, dom_elt);
        point exm_set_right[N];
        point exm_set_left[N];
        int sizeleft, sizeright;
        sizeleft = sizeright = 0;
        if (!split)
        {
            for (int i = 0; i < size; ++i)
            {

                if (!equal(exm_set[i], dom_elt) && exm_set[i].x <= dom_elt.x)
                {
                    exm_set_left[sizeleft].x = exm_set[i].x;
                    exm_set_left[sizeleft].y = exm_set[i].y;
                    sizeleft++;
                }
                else if (!equal(exm_set[i], dom_elt) && exm_set[i].x >
dom_elt.x)
```

```
                {
                    exm_set_right[sizeright].x = exm_set[i].x;
                    exm_set_right[sizeright].y = exm_set[i].y;
                    sizeright++;
                }
            }
        }
        else
        {
            for (int i = 0; i < size; ++i)
            {

                if (!equal(exm_set[i], dom_elt) && exm_set[i].y <= dom_elt.y)
                {
                    exm_set_left[sizeleft].x = exm_set[i].x;
                    exm_set_left[sizeleft].y = exm_set[i].y;
                    sizeleft++;
                }
                else if (!equal(exm_set[i], dom_elt) && exm_set[i].y >
dom_elt.y)
                {
                    exm_set_right[sizeright].x = exm_set[i].x;
                    exm_set_right[sizeright].y = exm_set[i].y;
                    sizeright++;
                }
            }
        }
        T = new Tnode;
        T->dom_elt.x = dom_elt.x;
        T->dom_elt.y = dom_elt.y;
        T->split = split;
        T->left = build_kdtree(exm_set_left, sizeleft, T->left);
        T->right = build_kdtree(exm_set_right, sizeright, T->right);
        return T;
    }
}
double Distance(point a, point b)
{
    double tmp = (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
    return sqrt(tmp);
}
void searchNearest(Tnode *Kd, point target, point &nearestpoint, double
&distance)
{
    // 1. 如果Kd是空的，则设dist为无穷大返回
    // 2. 向下搜索直到叶子结点
    stack<Tnode *> search_path;
    Tnode *pSearch = Kd;
    point nearest;
    double dist;
    while (pSearch != NULL)
    {
        // pSearch加入到search_path中;
        search_path.push(pSearch);
        if (pSearch->split == 0)
            if (target.x <= pSearch->dom_elt.x) pSearch = pSearch->left;//小于就进
入左子
            else pSearch = pSearch->right;
```

```
        else
            if (target.y <= pSearch->dom_elt.y) pSearch = pSearch->left; //小于就
进入左子
            else pSearch = pSearch->right;
    }
    // 取出search_path最后一个赋给nearest
    nearest.x = search_path.top()->dom_elt.x;
    nearest.y = search_path.top()->dom_elt.y;
    search_path.pop();
    dist = Distance(nearest, target);
    //3.回溯搜索路径
    Tnode *pBack;
    while (search_path.size() != 0)
    {
        // 取出search_path最后一个结点赋给pBack
        pBack = search_path.top();
        search_path.pop();
        if (pBack->left == NULL && pBack->right == NULL) /* 如果pBack为叶子结点 */
            if (Distance(nearest, target) > Distance(pBack->dom_elt, target))
            {
                nearest = pBack->dom_elt;
                dist = Distance(pBack->dom_elt, target);
            }
        else
        {
            int s = pBack->split;
            if (!s)
            {
/* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子
空间去搜索 */
                if (fabs(pBack->dom_elt.x - target.x) < dist)
                {
                    if (Distance(nearest, target) > Distance(pBack->dom_elt,
target))
                    {
                        nearest = pBack->dom_elt;
                        dist = Distance(pBack->dom_elt, target);
                    }
                    if (target.x <= pBack->dom_elt.x) /* target位于pBack的左子空
间，跳到右子空间 */
                        pSearch = pBack->right;
                    else pSearch = pBack->left;
                    if (pSearch != NULL) search_path.push(pSearch);// pSearch加入
到search
                }
            }
            else
            {
                if (fabs(pBack->dom_elt.y - target.y) < dist)
                {
                    if (Distance(nearest, target) > Distance(pBack->dom_elt,
target))
                    {
                        nearest = pBack->dom_elt;
                        dist = Distance(pBack->dom_elt, target);
                    }
                    if (target.y <= pBack->dom_elt.y) /* target位于pBack的左子空
间，跳到右子空间 */
```

```
                    pSearch = pBack->right;
                else pSearch = pBack->left;
                if (pSearch != NULL) search_path.push(pSearch); // pSearch加入到search中
            }
        }
    }
    nearestpoint.x = nearest.x;
    nearestpoint.y = nearest.y;
    distance = dist;
}
```

树的直径

我们记录当 1 为树的根时，每个节点作为子树的根向下，所能延伸的最长路径长度 d_1 与次长路径（与最长路径无公共边）长度 d_2，那么直径就是对于每一个点，该点 $d_1 + d_2$ 能取到的值中的最大值。

树形 DP 可以在存在负权边的情况下求解出树的直径。

```
int d, d1[N], d2[N];
vector<int> tr[N];
void dfs(int u, int f)
{
    d1[u] = d2[u] = 0;
    for (int v : tr[u])
        if (v != f)
        {
            dfs(v, u);
            int t = d1[v] + 1;
            if (t > d1[u])
                d2[u] = d1[u], d1[u] = t;
            else if (t > d2[u])
                d2[u] = t;
        }
    d = max(d, d1[u] + d2[u]);
}
```

树的重心

定义：**所有的子树中最大的子树节点数最少，那么这个点就是这棵树的重心，删去重心后，生成的多棵树尽可能平衡。**

> **性质**
>
> - 树的重心如果不唯一，则至多有两个，且这两个重心相邻。
> - 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。
> - 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。
> - 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。
> - 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

求法

在 DFS 中计算每个子树的大小，记录「向下」的子树的最大大小，利用总点数 - 当前子树（这里的子树指有根树的子树）的大小得到「向上」的子树的大小，然后就可以依据定义找到重心了。

```
/*树的预处理*/
int weight[N];
int getWeight(int n)
{
    for(int i = 1; i <= n; ++i)
    {
        for(int v : tr[i])
            weight[i] = max(siz[v], weight[i]);
        weight[i] = max(n - siz[i], weight[i]);
    }
    return min_element(weight + 1, weight + 1 + n) - weight;
}
```

直接求重心：

> ⚠ 找到重心之后直接退出，所以siz数组并没有完全更新，不能将其视作记录了子树大小

```
vector<int> tr[N];
int siz[N], wtcen, tot; // wtcen记录树的重心，tot记录树的大小
//int del[N]; //在点分治中使用
void dfs(int u, int f)
{
    int mx = 0;
    siz[u] = 1;
    for (auto v : tr[u])
        if (v != f) //if(v != f && !del[v]) //在点分治中使用
        {
            dfs(v, u);
            if (wtcen) return; //找到重心就退出
            siz[u] += siz[v];
            mx = max(mx, siz[v]);
        }
    mx = max(mx, tot - siz[u]);
    if (mx <= tot / 2)
        wtcen = u;
}
```

树的预处理

big 为重儿子，hig 为长儿子，$depth$ 为节点深度，ht 为节点高度，siz 为子树大小，L 为子树 dfs 序区间左端点，R 为子树 dfs 序区间右端点，rnk 为 dfs 序对应节点序号，fa 是当前节点父亲

```
vector<int> tr[N];
int big[N], hig[N], depth[N], ht[N], siz[N], L[N], R[N], rnk[N], fa[N], idx;
void dfs(int u, int f)
{
    siz[u] = 1;
    L[u] = ++idx; // 记录子树dfn区间左端点
    rnk[idx] = u; // 记录dfn对应的节点
    fa[u] = f;
    depth[u] = depth[f] + 1;
    ht[u] = 1; // 高度至少为1
    for (auto v : tr[u])
```

```
        if (v != f)
        {
            dfs(v, u);
            siz[u] += siz[v];
            if (!big[u] || siz[big[u]] < siz[v])
                big[u] = v;
        }
    ht[f] = ht[u] + 1;
    R[u] = idx; // 记录子树dfn区间右端点
    // 处理长儿子
    for (auto v : tr[u])
        if (v != f)
        {
            dfs(v, u);
            if (!hig[u] || ht[hig[u]] < ht[v])
                hig[u] = v;
        }
}
// for (int i = L[v]; i <= R[v]; ++i); //dfs序遍历子树
```

最近公共祖先LCA

时间复杂度 $O(nlogn)$

```
vector<int> tr[N];
int depth[N], fa[N][22];
void bfs(int root) // 预处理倍增数组
{
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[root] = 1; // depth存储节点所在层数
    queue<int> q;
    q.push(root);
    while (!q.empty())
    {
        int fro = q.front();
        q.pop();
        for (auto i : tr[fro])
        {
            if (depth[i] > depth[fro] + 1)
            {
                depth[i] = depth[fro] + 1;
                q.push(i);
                fa[i][0] = fro; // j的第二次幂个父节点
                for (int k = 1; k <= 20; k++)
                    fa[i][k] = fa[fa[i][k - 1]][k - 1];
            }
        }
    }
}
int lca(int a, int b) // 返回a和b的最近公共祖先
{
    if (depth[a] < depth[b])
        swap(a, b);
    for (int k = 20; k >= 0; k--)
        if (depth[fa[a][k]] >= depth[b])
            a = fa[a][k];
    if (a == b)
```

```
        return a;
    for (int k = 20; k >= 0; k--)
        if (fa[a][k] != fa[b][k])
        {
            a = fa[a][k];
            b = fa[b][k];
        }
    return fa[a][0];
}
```

树链剖分LCA：

时间复杂度 $O(0.37nlogn)$

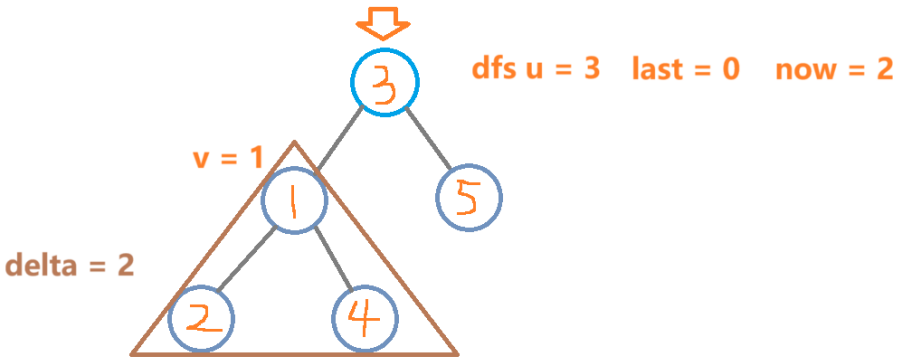
树链剖分是我认为好的求LCA方式，它求LCA的时间复杂度是O（0.37nlog$_2$n）（鬼知道怎么算的），比倍增快3倍左右。编程复杂度也不比倍增多多少，NOIP中绝对有用。

```
int lca(int u, int v)
{
    while (top[u] != top[v])
        depth[top[u]] > depth[top[v]] ? u = fa[top[u]] : v = fa[top[v]];
    return depth[u] > depth[v] ? v : u;
}
```

树上逆序对

单根节点

求以某节点的儿子为根的子树中有多少个数小于当前节点：




```
/*树状数组操作集*/
void dfs(int u, int fa)
{
    add(u);
    int last = query(u - 1);
    for (int v : g[u])
    {
        if (v != fa)
        {
            dfs(v, u);
            int now = query(u - 1);
            cnt[u] += now - last;
            last = now;
        }
    }
```

```
}
```

$1-n$ 分别为根的树上的逆序对总个数：

假设树根为节点 1，枚举中间点 w，考虑 v 的位置：

- 在 w 的子节点 s 的子树中，若有 cnt 个标号小于 w 的节点 v，则 w 的其它分支（包括 w）中所有点 u 有 $f(u)+=cnt$。为了方便维护，可以对全部点 $f(u)+=cnt$，再对 s 的子树中的点 $f(u)-=cnt$
- 在 w 的父亲分支，若有 cnt 个标号小于 w 的节点 v，则 w 的子树中所有点 u 有 $f(u)+=cnt$

考虑dfs序进行差分

```
void dfs(int u, int fa)
{
    add(u);
    dfn[u] = ++cur;
    int last = query(u - 1), lower = u - 1;
    for (int v : g[u])
    {
        if (v != fa)
        {
            dfs(v, u);
            int now = query(u - 1);
            int delta = now - last;
            last = now;
            lower -= delta;
            sub[1] += delta; //情况一
            sub[dfn[v]] -= delta;
            sub[cur + 1] += delta;
        }
    }
    //情况二，父亲分支小于u的个数为余下的lower，对当前子树的dfn区间+lower
    sub[dfn[u]] += lower;
    sub[cur + 1] -= lower;
}
void work()
{
    for (int i = 1; i <= n; i++) sub[i] += sub[i - 1];
    for (int i = 1; i <= n; i++) cout << sub[dfn[i]] << " ";
}
```

树链剖分

树链剖分用于将树分割成若干条链的形式，以维护树上路径的信息。

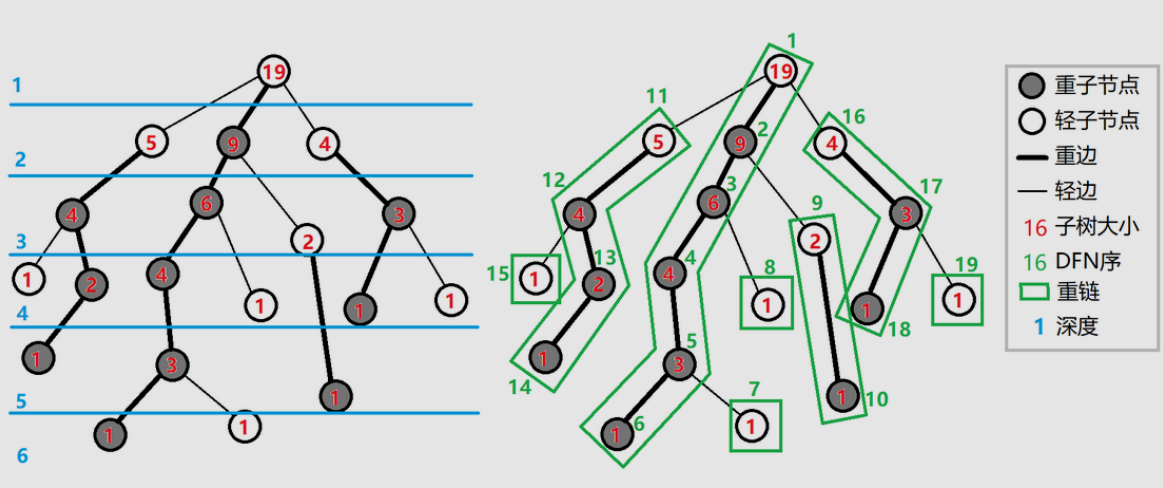
具体来说，将整棵树剖分为若干条链，使它组合成线性结构，然后用其他的数据结构维护信息。

树链剖分（树剖/链剖）有多种形式，如 **重链剖分**，**长链剖分** 和用于 Link/cut Tree 的剖分（有时被称作「实链剖分」），大多数情况下（没有特别说明时），「树链剖分」都指「重链剖分」。

重链剖分

重链剖分可以将树上的任意一条路径划分成不超过 $O(\log n)$ 条连续的链，每条链上的点深度互不相同（即是自底向上的一条链，链上所有点的 LCA 为链的一个端点）。

重链剖分还能保证划分出的每条链上的节点 DFS 序连续，因此可以方便地用一些维护序列的数据结构（如线段树）来维护树上路径的信息。




```
/*重儿子预处理*/
int dfn[N], rnk[N], top[N], idx;
void dfs(int u, int f, int t) //t为当前节点重链的头
{
    dfn[u] = ++idx;
    rnk[idx] = u;
    top[u] = t; //记录重链的头
    if(big[u])
        dfs(big[u], u, t); //优先对重儿子进行DFS，可以保证同一条重链上的点DFS序连续
    for(int v : tr[u])
        if(v != f && v != big[u])
            dfs(v, u, v);
}
```

长链剖分

长链剖分本质上就是另外一种链剖分方式。

定义 **重子节点** 表示其子节点中子树深度最大的子结点。如果有多个子树最大的子结点，取其一。如果没有子节点，就无重子节点。

定义 **轻子节点** 表示剩余的子结点。

树上启发式合并

树上启发式合并（常常也叫DSU On Tree，但其实和DSU并没有特别大关系），是一种解决某些**树上离线问题**的算法，尤其常被用于解决“对每个节点，询问关于其子树的某些信息”这样的问题。

假设我们要对树上的每个节点 p 求 $ans[p]$，且这个 $ans[p]$ 可以通过合并 p 的子节点的某些信息得知，一般来说我们可以用树形DP解决。但如果“子节点的某些信息”的规模较大，简单的树形DP在时间和空间上都可能爆炸。所以我们不能存储每个节点的信息，而是要实现某种**资源复用**。

> 每次将轻儿子的信息暴力合并到重儿子的信息上从而得到自己的信息，本质上和一般的启发式合并是一样的：将较小的块合并到较大的块，从而保证每个点至多被合并 $logn$ 次，因为一个点每次合并后所在的块的大小至少翻倍。

```
/*重子树与子树dfn区间*/
int cnt[N];
void dsu(int u, int f, bool keep)
{
    for (int v : tr[u])
        if (v != f && v != big[u])
            dsu(v, u, false);
    if (big[u])
        dsu(big[u], u, true);
    for (int v : tr[u])
        if (v != f && v != big[u])
            for (int i = L[v]; i <= R[v]; ++i)
                cnt[depth[rnk[i]]]++;
    cnt[depth[u]]++;
    /*getAns()*/ //记录答案
    if (!keep)
        for (int i = L[u]; i <= R[u]; ++i)
            cnt[depth[rnk[i]]]--;
}
```

虚树

虚树的概念

虚树，是对于一棵给定的节点数为 n 的树 T，构造一棵新的树 T' 使得总结点数**最小**且**包含指定的某几个节点和他们的LCA**。

虚树解决的问题

利用虚树，可以对于指定**多组**点集 S 的询问进行**每组** $O(|S|(\log n+\log |S|)+f(|S|))$ 的回答，其中 $f(x)$ 指的是对于一棵 x 个点的树**单组询问**这个问题的时间复杂度。可以看到，这个复杂度基本上(除了那个 $\log n$ 以外)与 n 无关了。这样，对于多组询问的回答就可以省去**每次询问**都遍历一整棵树的 $O(n)$ 复杂度了。

时间复杂度 $O(mlogn)$，其中 m 为关键点数，n 为总点数。

构造方法 I： 二次排序 + LCA连边

```
/*lca模板*/
//可以将1作为虚树根节点方便操作
int dfn[N];
vector<int> kp, a, vt[N]; // kp存储关键点，a存储虚树序列结果，vt为新建虚树
void build_virtual_tree()
```

```
{
    auto cmp = [&](int a, int b) -> bool
    { return dfn[a] < dfn[b]; };
    sort(kp.begin(), kp.end(), cmp); // 把关键点按照 dfn 序排序
    if(!kp.empty()) a.push_back(kp.front());
    for (int i = 1; i < kp.size(); ++i)
    {
        a.push_back(kp[i]);
        a.push_back(lca(kp[i - 1], kp[i])); // 插入 lca
    }
    sort(a.begin(), a.end());
    a.erase(unique(a.begin(), a.end()), a.end());
    // 建虚树
    for (int i = 1; i < a.size(); ++i)
    {
        int lc = lca(a[i - 1], a[i]);
        vt[lc].push_back(a[i]);
        vt[a[i]].push_back(lc);
    }
}
```

构造方法Ⅱ：单调栈

```
/*lca模板*/
int dfn[N];
vector<int> kp, vt[N]; // kp存储关键点
void build_virtual_tree()
{
    auto cmp = [&](int a, int b) -> bool
    { return dfn[a] < dfn[b]; };
    auto conn = [&](int a, int b) -> void
    { vt[a].push_back(b), vt[b].push_back(a); };
    sort(kp.begin(), kp.end(), cmp);
    stack<int> s;
    s.push(1);
    //vt[1].clear(); //重复使用的初始化
    for (auto i : kp) // 如果1号节点是关键节点就不要重复添加
        if (i != 1)
        {
            int lc = lca(i, s.top()); // 计算当前节点与栈顶节点的 LCA
            if (lc != s.top()) // 不同，说明当前节点不再当前栈所存的链上
            {
                int last = s.top();
                s.pop();
                while (dfn[lc] < dfn[s.top()]) // 当次大节点的Dfn大于LCA的Dfn
                {
                    conn(s.top(), last);
                    last = s.top();
                    s.pop();
                }                              // 把与当前节点所在的链不重合的链连接掉并且
弹出
                if (dfn[lc] > dfn[s.top()]) // 说明LCA是第一次入栈，清空其邻接表，连边
后弹出栈顶元素，并将 LCA入栈
                {
                    //vt[lc].clear();
                    conn(lc, last);
                    s.push(lc);
```

```
                }
                else // 说明LCA就是次大节点，直接弹出栈顶元素
                    conn(lc, last);
            }
            //vt[i].clear();
            s.push(i);
        }
    int last = s.top();
    s.pop();
    while (!s.empty()) // 剩余的最后一条链连接一下
    {
        conn(last, s.top());
        last = s.top();
        s.pop();
    }
}
```

点分治

点分治或**重心剖分**（Centroid Decomposition）是树分治的一种，主要处理一些树上路径问题。

为了效率更高地解决问题，我们引入**分治**思想。对于每个点，我们分别考虑**包含这个点**的路径和**不包含这个点**的路径。对于前者，我们做一趟dfs；对于后者，我们删除该点后，对所有子树**递归**地处理即可。

每次都选择子树的**重心**作为子树的根，那么复杂度就可以保证为 $O(nlogn)$

> ⚠ 点分治因为递归的处理，时间常数很大

```
vector<int> tr[N];
int siz[N], del[N], wtcen, tot; // wtcen记录树的重心，tot记录树的大小
void getwt(int u, int f)        // 找重心
{
    int mx = 0;
    siz[u] = 1;
    for (auto v : tr[u])
        if (v != f && !del[v])
        {
            getwt(v, u);
            if (wtcen)
                return; // 找到重心就退出
            siz[u] += siz[v];
            mx = max(mx, siz[v]);
        }
    mx = max(mx, tot - siz[u]);
    if (mx <= tot / 2)
        wtcen = u, siz[f] = tot - siz[u];
}
void calc(int u, int f)
{
    /*计算或统计操作*/
    for (int v : tr[u])
        if (v != f && !del[v])
            calc(v, u);
}
void CenDec(int rt)
{
    for (int v : tr[rt]) // 遍历当前树
        if (!del[v])
```

```
            calc(v, rt);
    del[rt] = 1; // 删除节点
    for (int v : tr[rt])
        if (!del[v])
        {
            tot = siz[v];
            wtcen = 0;
            getwt(v, 0);
            CenDec(wtcen);
        }
}
```

树上随机游走

给定一棵有根树，树的某个结点上有一个硬币，在某一时刻硬币会等概率地移动到邻接结点上，问硬币移动到邻接结点上的期望距离。

设 $f(u)$ 代表 u 结点走到其父结点 p_u 的期望距离，则有：

$$f(u)=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v)+f(u))}{d(u)}$$

分子中的前半部分代表直接走向了父结点，后半部分代表先走向了子结点再由子结点走回来然后再向父结点走；分母 $d(u)$ 代表从结点 u 走向其任何邻接点的概率相同。

化简如下：

$$\begin{aligned}f(u)&=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v)+f(u))}{d(u)}\\&=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v))+(d(u)-1)f(u)}{d(u)}\\&=w(u,p_u)+\sum_{v\in son_u}(w(u,v)+f(v))\\&=\sum_{(u,t)\in E}w(u,t)+\sum_{v\in son_u}f(v)\end{aligned}$$

当树上所有边的边权都为 1 时，上式可化为：

$$f(u)=d(u)+\sum_{v\in son_u}f(v)$$

即 u 子树的所有结点的度数和，也即 u 子树大小的两倍 -1（每个结点连向其父亲的边都有且只有一条，除 u 与 p_u 之间的边只有 1 点度数的贡献外，每条边会产生 2 点度数的贡献）。

设 $g(u)$ 代表 p_u 结点走到其子结点 u 的期望距离，则有：

$$\begin{aligned}g(u)&=\frac{w(p_u,u)+(w(p_u,p_{p_u})+g(p_u)+g(u))+\sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)+g(u))}{d(p_u)}\\&=g(p_u)+f(p_u)-f(u)\end{aligned}$$

初始状态为 $g(\text{root})=0$。

```
vector<int> G[N];
int f[N], g[N];
```

```
void dfs1(int u, int p)
{
    f[u] = G[u].size();
    for (auto v : G[u])
    {
        if (v == p)
            continue;
        dfs1(v, u);
        f[u] += f[v];
    }
}
void dfs2(int u, int p)
{
    if (u != root)
        g[u] = g[p] + f[p] - f[u];
    for (auto v : G[u])
    {
        if (v == p)
            continue;
        dfs2(v, u);
    }
}
```

图论

链式前向星

```
typedef struct Edge
{
    int to;
    int next;
    int val;
};
Edge edge[N];
int head[N], cnt = 1;
void creat_sta(int begin, int end, int val)//创建链式前向星
{
    edge[cnt].to = end;
    edge[cnt].next = head[begin];
    edge[cnt].val = val;
    head[begin] = cnt++;
}
void putout(int node)//访问链式前向星有向图
{
    multiset<int>out;
    cout << node << " ->";
    for (int i = head[node]; i != 0; i = edge[i].next)
        out.insert(edge[i].to);
    for (auto iter : out)
        cout << ' ' << iter;
    cout << endl;
}
```

最小生成树

Kruskal

时间复杂度 $O(mlogm)$

> 步骤1：先对图中所有的边按照权值进行排序
> 步骤2：如果当前这条边的两个顶点不在一个连通块里面，那么咋就用 并查集 的Union函数把他们合并在一个连通块里面(也就是把他们放在最小生成树里面)，如果再在一个并查集里面，我们就舍弃这条边，不需要这条边。
> 步骤3：一直执行步骤2，知道当边数等于定点数的数目减去1，那就说明这n个顶点就连合并在一个集合里面了；如果边数不等于顶点数目减去1，那么说明这些边就不连通。

```
int n, m, idx;//n是点数,m是边数,idx是生成树当前边数
int p[N];//并查集的父节点数组
struct Edge//结构体数组存边
{
    int a, b, w;
    bool operator< (const Edge &_w)const { return w < _w.w; }
}edges[M], tree[N]; //下标从0开始
int find(int x)//并查集
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}
int kruskal()
{
    sort(edges, edges + m);//将所有边按边权排序
    for (int i = 1; i <= n; i ++ ) p[i] = i;//初始化并查集
    int res = 0, cnt = 0;//边权和,点数
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);
        if (a != b)//如果两个连通块不连通,则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            tree[idx++] = edges[i];
            cnt++;
        }
    }
    if (cnt < n - 1) return inf;//边数小于n-1说明不连通
    return res;
}
```

Prim

适用于稠密图尤其完全图
时间复杂度 $O(n^2 + m)$

```
int n;//n点数
int g[N][N];//邻接矩阵,存所有边
int dist[N];//存储其他点到当前最小生成树的距离
bool st[N];//存储每个点是否已经在生成树中
int prim()//如果图不连通,则返回inf,否则返回最小生成树的树边权重之和
{
    memset(dist, 0x3f, sizeof dist);//初始化所有点距离为正无穷
    int res = 0;
    for (int i = 0; i < n; i++)
```

```
    {
        int t = -1;
        for (int j = 1; j <= n; j++)//每次找到不在当前生成树中的点到树的最短距离
            if (!st[j] && (t == -1 || dist[t] > dist[j])) t = j;
        if (i && dist[t] == inf) return inf;
        if (i) res += dist[t];
        st[t] = true;//把该点加入生成树
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);//用该点更新其他点到生成树的距离(存在自环可能会更新自己)
    }
    return res;
}
```

最短路算法

Floyd

时间复杂度 $O(n^3)$ 空间复杂度 $O(n^2)$
求任意两点间最短路，不能有负环

```
memset(d, 0x3f, sizeof(d));//初始化所有边权正无穷
for (int i = 1; i <= n; i++) d[i][i] = 0;//自环边权0,有直连边赋值为边权
void floyd()//d[a][b]表示a到b的最短距离
{
    for (int k = 1; k <= n; k++)//经过k点
        for (int i = 1; i <= n; i++)//i起点
            for (int j = 1; j <= n; j++)//j终点
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);//i->j多经过k点距离是否变短
}
```

Bellman-Ford

时间复杂度 $O(nm)$
求有负权的图的最短路，可对最短路不存在的情况进行判断
只能说明所在连通块存在负环；判整个图是否存在负环时，建立一个超级源点，向所有节点连一条边权 0 的边，以超级源点为起点运行，可求经过不超过 k 条边的最短路（备份上次迭代结果更新本轮迭代防止串联）

```
struct edge {
  int v, w;
};
vector<edge> e[N];
int dis[N];
bool bellmanford(int n, int s)//n点数,s起点,返回s点能否抵达一个负环
{
    memset(dis,0x3f, sizeof dis);
    dis[s] = 0;
    bool flag;  //判断一轮循环过程中是否发生松弛操作
    for (int i = 1; i <= n; i++)//经过不超过n条边的最短距离,循环n次之后所有边满足 dis[b] <=dis[a]+w
    {
        flag = false;
        for (int u = 1; u <= n; u++)
        {
            if (dis[u] == inf) continue;//最短路长度为inf的点引出的边不可能发生松弛操作
```

```
            for (auto ed : e[u])
            {
                int v = ed.v, w = ed.w;
                if (dis[v] > dis[u] + w)
                {
                    dis[v] = dis[u] + w;//更新点的距离(松弛操作)
                    flag = true;
                }
            }
        }
        if (!flag) break;//没有可以松弛的边时就停止算法
    }
    return flag;// 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
}//结束时dis[ed]>inf/2说明不可达
```

SPFA:bellman-ford优化

时间复杂度平均 $O(km)$，最坏 $O(mn)$，可能被卡，一般用于带负环图

```
struct edge {
  int v, w;
};
vector<edge> e[N];
int dis[N], cnt[N], vis[N];
queue<int> q;
bool spfa(int n, int s)//s所在连通块存在负环返回false
{
    memset(dis,0x3f, sizeof dis);//初始化所有距离正无穷
    dis[s] = 0, vis[s] = 1;//起点距离0
    q.push(s);//向队列插入起点
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (auto ed : e[u])
        {
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w)//距离变小的点入队
            {
                dis[v] = dis[u] + w;
                cnt[v] = cnt[u] + 1;  //记录最短路经过的边数
                if (cnt[v] >= n) return false;//最短路至多经过n-1条边,否则一定经过了负
环
                if (!vis[v]) q.push(v), vis[v] = 1;//如果队列中已存在v,则不需要将v重复
插入
            }
        }
    }
    return true;
}//结束时dis[ed]>inf/2说明不可达
```

Dijkstra

时间复杂度 $O(n^2)$
求解非负权图单源最短路，适合稠密图

```
struct edge {
  int v, w;
};
vector<edge> e[N];
int dis[N], vis[N];//存储每个点的最短路是否已经确定
void dijkstra(int n, int s)//求起点到所有点最短路
{
    memset(dis, 0x3f, sizeof dis );//初始化1号点距离为零,其它点距离正无穷
    dis[s] = 0;
    for (int i = 1; i <= n; i++)
    {
        int u = 0, mind = inf;
        for (int j = 1; j <= n; j++)
            if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];//在还未确定最短路的点中,寻找距离最小的点
        vis[u] = true;//确定该点最短距离
        for (auto ed : e[u])
        {
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;//更新其他点的距离
        }
    }
}//dis[ed]==inf说明不可达
```

Dijkstra优先队列优化

时间复杂度 $O(mlogm)$
求解非负权图单源最短路，适合稀疏图

```
typedef pair<int, int> pii;
struct edge
{
  int v, w;
};
vector<edge> e[N];
int dis[N];
bool vis[N];
void dijkstra(int s)
{
    memset(dis, 0x3f, sizeof dis); //memset(dis, 0x3f, sizeof(int) * (n + 1));
    //memset(path,-1,sizeof(path));//记录路径初始化
    dis[s] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> pq;//小根堆
    pq.push({0, s});//first存储距离,second存储节点编号
    while (!pq.empty())
    {
        int u = pq.top().se;
        pq.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (auto ed : e[u])
```

```
        {
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w)
            {
                dis[v] = dis[u] + w;
                pq.push({dis[v], v});
                //path[v] = u;//记录路径前驱
            }
        }
    }
}//dis[ed]==inf说明不可达
```

Dijkstra链式前向星+优先队列优化

时间复杂度 $O(mlogn)$

```
struct edge{//存储边
    int u, v, w, next;//u为起点，v为终点，w为权值，next为前继
};
edge e[N];
int head[N],dis[N],n,m,s,cnt;//head为链中最上面的，dis表示当前答案，n为点数，m为边数，s为起点，cnt记录当前边的数量
bool vis[N];
struct node
{
    int w, to;//w表示累加的权值，to表示到的地方
    bool operator < (const node &x)const{ return w>x.w; }
};
priority_queue<node>q;
void add(int u, int v, int w)//链式前向星（加边）
{
    ++cnt;//增加边的数量
    e[cnt].u = u;//存起点
    e[cnt].v = v;//存终点
    e[cnt].w = w;//存权值
    e[cnt].next = head[u];//存前继
    head[u] = cnt;//更新链最上面的序号
}
void dijkstra()
{
    memset(dis,0x3f,sizeof(dis));//初始化，为dis数组附一个极大值，方便后面的计算
    dis[s] = 0;//起点到自己距离为0
    q.push(node{0,s});//压入队列
    while(!q.empty())//队列不为空
    {
        node x = q.top();//取出队列第一个元素
        q.pop();//弹出
        int u = x.to;//求出起点
        if(vis[u]) continue;
        vis[u] = true;//标记已访问
        for(int i = head[u]; i; i = e[i].next)
        {
            int v = e[i].v;//枚举终点
            if(dis[v]>dis[u]+e[i].w)//若中转后更优，就转
            {
                dis[v] = dis[u] + e[i].w;
                q.push(node{dis[v],v});//压入队列
```

```
            }
        }
    }
}
```

最短路径打印问题

我们可以定义一个pre数组，然后pre[i]记录的是上一个位置是哪一个节点，当然初始的时候我们全部初始化为-1，然后每次松弛操作的时候就更新一下上一个节点的位置，你有没有发现这就是链式前向星，然后最后打印的时候要么递归打印，那么手动写栈打印，这个方法不只是适用于Dijkstra，而且也适用于其他最短路算法，如SPFA、bellman_ford、Floyd等等

那么简单描述一下打印函数

```
int path[N];
void print(int x) //x为终点
{
    if(x == -1) return;
    print(path[x]); //递归打印
    printf("%d->", x);
}
```

Tarjan

基于深度优先搜索的，用于求解图的连通性问题的算法。

Tarjan 算法

原本还有一个 **kosaraju** 算法，因为没有什么用，在这里就不专门介绍了。

这里我们用 **dfn[i]** 表示编号为 i 的节点在 **dfs 的过程**中的遍历顺序，就是一个 **dfs 序**。（也可以叫时间戳）

用 **low[i]** 表示 i 节点及其下方节点**所能到达的开始时间最早的节点的开始时间**。（初始时 **low[i] = dfn[i]**）

这里有 1 个性质：因为在 dfs 的过程中会形成一棵**搜索树**，所以在**越上面的节点显然 dfn 就会越小**。

如果发现一个点有边**连到了搜索树中的自己的祖宗节点**，那么就**更新其 low 的值**。

关于 low 值与 dfn 值

1、如果一个节点的 **low 值小于 dfn 值**，那么就说明**它或者它的子孙节点有边连到自己上方的节点**。

2、如果一个节点的 **low 值等于 dfn 值**，则说明**其下方的节点不能走到其上方节点**，那么**该节点就是一个强连通分量在搜索树中的根**。

3、但是 **u 的子孙节点就未必和 u 处于同一个强连通分量**，用**栈存储**即可（具体看代码）。

```
int dfn[N], low[N], id[N], in_stk[N];
int timestamp, scc_cnt;
vector<int> gra[N];
stack<int> stk;
void tarjan(int u)
{
    dfn[u] = low[u] = ++timestamp;
    stk.push(u), in_stk[u] = true;
    for (auto i : gra[u])
    {
        if (!dfn[i])
        {
            tarjan(i);
            low[u] = min(low[u], low[i]);
        }
        else if (in_stk[i])
            low[u] = min(low[u], dfn[i]);
```

```
    }
    if (dfn[u] == low[u])
    {
        ++scc_cnt;
        int y;
        do
        {
            y = stk.top();
            stk.pop();
            in_stk[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
```

确保所有点进行了tarjan

```
for (int i=1; i<=n; ++i)
    if (!dfn[i]) tarjan(i);
```

缩点后的新图

```
set<int> newgra[N];
void tarjan_graph(int n)
{
    for (int i = 1; i <= n; ++i)
        for (int iter : gra[i])
            if (id[i] != id[iter])
                newgra[id[i]].insert(id[iter]);
}
```

二分图

匈牙利算法

又称为 **KM** 算法，可以在时间 $O(n^3)$ 内求出二分图的 **最大权完美匹配。**

邻接矩阵存储图

```
int n, m, e, ans, g[N][N], ask[N], matched[N];
inline bool found(int x) // dfs找增广路
{
    for (int i = 1; i <= m; i++)
        if (g[x][i])
        {
            if (ask[i])
                continue;
            ask[i] = 1;
            if (!matched[i] || found(matched[i]))
            {
                matched[i] = x;
                return true;
            }
        }
    return false;
}
```

```
inline void match()
{
    int cnt = 0; // cnt是计数器
    for (int i = 1; i <= n; i++)
        if (found(i))
            cnt++; // 找到了就加1
    ans = cnt;
}
```

优化建图

虚点建图

对于暴力建图后边数过大的图论问题，考虑构造虚点，减少边的数量，提升边的访问效率。

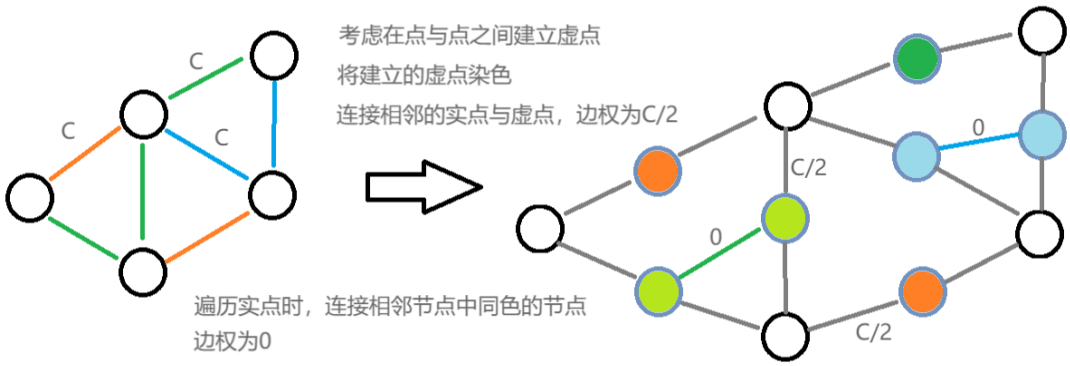
常用于解决最短路问题。

巧妙的虚点建图可以优化到常数级别。

> ⚠ 虚点的构建需要一定程度的顿悟

例题：有 n 个点和 m 条边，每条边有一种颜色，若经过的路径上只有 i 一种颜色，则需要支付 c 的代价，若经过的路径上有k种颜色，只要当前边与前一条边颜色不同，则需要支付代价 c 。

虚点优化建图过程：



线段树优化建图

在最短路、强连通分量、2-SAT、网络流等图论问题中，边数有时达到 O(n2) 甚至 O(n3)，成为时间或空间复杂度的瓶颈所在，使用优化建图可以在不影响效果的情况下建出边数更少的图。

支持以下四种操作：

> $u \to v$连边　$u \to [l,r]$ 连边　$[l,r] \to v$ 连边　$[l1,r1] \to [l2,r2]$连边

区间操作想到线段树，可以把区间拆成 $logn$ 个线段树上区间。先按照线段树建出两棵树：外向树和内向树。对于所有连入一个区间的操作，将边连向外向树，这代表着连入一个大区间的边在经过外向树边后，也可以连入小区间；对于所有从一个区间连出的操作，将边从内向树连出，这代表着一个小区间在经过外向树的边后，可以连出大区间的边。

为了避免冲突，将外向树的编号从 1 开始，内向树的编号从 $4n$ 开始，单个节点的编号从 $8n$ 开始，且要与两棵树的叶子相连。

区间与区间相连按照上面做法是 $O(log^2 n)$ 条边，可以新建两个节点以一条边相连，将边权设置在这条边上。内向树对应区间全部由其中一个节点连入，外向树对应区间全部连到另一个节点，这样就只有 $O(logn)$ 条边。总点数大概在 $10n$ 左右，总边数是 $O(mlogn)$ 级别。

前后缀优化建图

可以用于连边区间序列的前后缀或树上根链的情况。

对于序列的前后缀，可以对每个节点 i 新建两个复制节点 pre_i, suf_i，连边为 $pre_i \to i, pre_i - 1, suf_i \to i, suf_i + 1$，这样当需要连前缀 $[1, i]$ 时，直接同 pre_i 相连即可，后缀同理。

对于树上根链，直接建内向树，连根链末尾即可。

杂项

并查集

```
int p[N];//存储每个点的祖宗节点
int find(int x)//返回x的祖宗节点同时路径压缩
{
    if (p[x] != x) p[x] = find(p[x]);//路径压缩优化:向上找到根节点后,将路径上所有点直接指向根节点
    return p[x];
}
void merge(int a, int b){ p[find(b)] = find(a); } //a与b所在集合合并
```

拓扑排序

拓扑排序

那么拓扑排序可表示为：计算一个有向无环图的拓扑序列。

逻辑

拓扑排序的逻辑如下：

1. 集合S表示所有入度为0的点；队列L表示拓扑序列。初始时为空。
2. 找到所有入度为0的点，放入S中。
3. 从S中取出一个点u，放入L中。
4. 在图中删除s，并删除所有以s为起点的边。
5. 重复2~4，直到S为空。

```
int inde[N], n; //入度和序号
vector<int> vex[N]; //邻接存储
vector<int> toposort()
{
    queue<int> qu;
    vector<int> res;
    for (int i = 1; i <= n; ++i) if (!inde[i]) qu.push(i), res.push_back(i);
    while (!qu.empty())
    {
        auto fro = qu.front(); qu.pop();
        for (auto it : vex[fro]) if (!(--inde[it])) qu.push(it),
res.push_back(it);
    }
    for (int i = 1; i <= n; ++i) if(inde[i]) { res.clear(); break; }
    return res; //返回的empty表示存在环
} //结束后入度数组被修改
```

关键路径AOE网

- AOE网的性质：
  - （1）只有在某顶点代表的事件发生后，从该顶点发出去的弧所代表的各项活动才能开始；
  - （2）只有进入某顶点的各条弧所代表的活动都已经结束，该顶点所代表的事件才能发生。
- 求解关键路径需要求解出：**事件**（顶点）的最早、最迟发生时间；
- 求解关键活动需要求解出：**活动**（边）的最早、最迟发生时间完成。

```
struct node { int to, val; };
int inde[N], outde[N], ve[N], vl[N], n;
vector<node> gra[N], dgra[N]; // 图和反建图
void AOE()                    // 默认已经是无环图
{
    queue<int> qu, subqu;
    for (int i = 1; i <= n; ++i)
        if (!inde[i])
        {
            qu.push(i);
            ve[i] = 0;
        }
    while (!qu.empty())
    {
        int fro = qu.front(); qu.pop();
        for (auto it : gra[fro])
        {
            qu.push(it.to);
            ve[it.to] = max(ve[it.to], ve[fro] + it.val);
        }
    }
    // 反建图求vl,最晚发生时间
    memset(vl, 0x3f, sizeof(vl));
    for (int i = 1; i <= n; ++i)
        if (!outde[i])
        {
            subqu.push(i);
            vl[i] = ve[i];
        }
    while (!subqu.empty())
    {
        int fro = subqu.front(); subqu.pop();
        for (auto it : dgra[fro])
        {
            subqu.push(it.to);
            vl[it.to] = min(vl[it.to], vl[fro] - it.val);
        }
    }
}
```

根据上述求出的`ve、vl`数组，我们可以得出关键路径上的点，所有`vl[i]-ve[i]==0`的点是关键路径上的点

根据 ve 和 vl，通过 bfs 求关键路径

离散化

```
//a为初始数组,下标1~n;len为离散化后数组的有效长度
sort(a + 1, a + 1 + n);
len = unique(a + 1, a + n + 1) - a - 1;//求出离散化后不同数的个数
lower_bound(a + 1, a + len + 1, x) - a;//查询x离散化后对应的编号,下标从1开始
//vector<int>a,b;b是a的一个副本
sort(a.begin(), a.end());
a.erase(unique(a.begin(), a.end()), a.end());//排序后去重
for (int i = 0; i < n; ++i)
    b[i] = lower_bound(a.begin(), a.end(), b[i]) - a.begin();//离散化,下标从0开始
```

区间合并

思路:

先按每一项的第一个数字排序，选取第一项为当前项cur。

有三种情况:

1.在当前项内部：待处理区间end <= 当前项end（无需处理，合并到情况2）

2.与当前项相交：待处理区间start <= 当前项end 拓宽当前项end

3.与当前项不相交：待处理区间end > 当前项end 待处理区间为当前项，result+1

稀疏表ST

设A[i]是要求区间最值的数列，F[i, j]表示从第i个数起连续2^j个数中的最大值。（DP的状态）
例如:
A数列为：3 2 4 5 6 8 1 2 9 7
F[1，0]表示第1个数起，长度为2^0=1的最大值，其实就是3这个数。同理 F[1,1] = max(3,2) = 3, F[1，2]=max(3,2,4,5) = 5，F[1，3] = max(3,2,4,5,6,8,1,2) = 8;
并且我们可以容易的看出F[i,0]就等于A[i]。（DP的初始值）
这样，DP的状态、初值都已经有了，剩下的就是状态转移方程。
我们把F[i，j]平均分成两段（因为f[i，j]一定是偶数个数字），从 i 到i + 2 ^ (j - 1) - 1为一段，i + 2 ^ (j - 1)到i + 2 ^ j - 1为一段(长度都为2 ^ (j - 1))。用上例说明，当i=1，j=3时就是3,2,4,5 和 6,8,1,2这两段。F[i，j]就是这两段各自最大值中的最大值。于是我们得到了状态转移方程F[i, j]=max（F[i，j-1], F[i + 2^(j-1)，j-1]）。

```
void RMQ(int num) //预处理->O(nlogn)
{
    for(int j = 1; j < 20; ++j)
        for(int i = 1; i <= num; ++i)
            if(i + (1 << j) - 1 <= num)
            {
                maxsum[i][j] = max(maxsum[i][j - 1], maxsum[i + (1 << (j - 1))]
[j - 1]);
                minsum[i][j] = min(minsum[i][j - 1], minsum[i + (1 << (j - 1))]
[j - 1]);
            }
}
int minc(int i,int j) { return a[i]<=a[j] ? i : j; }//这里一定是<=
int getmin(int x,int y) //查询
{
    int k=0;
    while((1<<k+1)<=(y-x+1)) k++;
    return minc(minsum[x][k],minsum[y+1-(1<<k)][k]); //求max改改就行
}
```

离线算法

二维偏序

二维偏序可以看作是：给定查询坐标 (a, b)，查找有多少个点 (x, y) 满足 $x <= a, y <= b$。

```
/*树状数组模板*/
struct point
{
    int x, y, type; //(x, y)为坐标，type为0表示点，为1表示查询
    bool operator<(const point &a) const
    {
        return x < a.x || (x == a.x && y < a.y);
    }
};
vector<point> p, q; // p为点集，q为查询
vector<int> TwoDimPO()
{
    vector<int> res;
    vector<point> list(p.size() + q.size());
    sort(p.begin(), p.end());
    sort(q.begin(), q.end());
    merge(p.begin(), p.end(), q.begin(), q.end(), res.begin());
    for (auto it : list)
    {
        if (it.type)
            res.pb(ask(it.y));
        else
            add(it.y, 1);
    }
}
```

CDQ 分治

这类问题多数类似于「给定一个长度为 n 的序列，统计有一些特性的点对 (i, j) 的数量/找到一对点 (i, j) 使得一些函数的值最大」。

CDQ 分治解决这类问题的算法流程如下：

1. 找到这个序列的中点 mid；
2. 将所有点对 (i, j) 划分为 3 类：
    a. $1 \le i \le mid, 1 \le j \le mid$ 的点对；
    b. $1 \le i \le mid, mid + 1 \le j \le n$ 的点对；
    c. $mid + 1 \le i \le n, mid + 1 \le j \le n$ 的点对。
3. 将 $(1, n)$ 这个序列拆成两个序列 $(1, mid)$ 和 $(mid + 1, n)$。此时第一类点对和第三类点对都在这两个序列之中；
4. 递归地处理这两类点对；
5. 设法处理第二类点对。

可以看到 CDQ 分治的思想就是不断地把点对通过递归的方式分给左右两个区间。

在实际应用时，我们通常使用一个函数 `solve(l,r)` 处理 $l \le i \le r, l \le j \le r$ 的点对。上述算法流程中的递归部分便是通过 `solve(l,mid)` 与 `solve(mid,r)` 来实现的。剩下的第二类点对则需要额外设计算法解决。

博弈论

SG函数

先来说一下可以用SG函数解决的问题所需要满足的条件：

（1）游戏人数为两人

（2）两人交替进行某种游戏规定的操作，每次操作，选手都可以在当前合法的操作中选取一种

（3）对于游戏的任意一种可能的局面，合法的操作集合只取决于这个局面的本身，而与操作者无关

就拿巴什博弈举例，每个操作者每次都可以拿走1~m块石头，每次拿走石头的个数取决于当前还剩多少块石头，而与之前的操作无关，这就是一个可以利用SG函数来解决的问题。

必败点和必胜点满足的性质：

（1）终结点是必败点

（2）可以进行一次操作到达必败点的为必胜点

（3）从必败点只能到达必胜点

介绍SG函数前先介绍一下mex函数，表示最小的不属于这个集合的非负整数，比如mex{0,1,3,4}=2,mex{1，2，3}=0;

任何一个可以用SG函数解决的问题都可以抽象成一个有向图游戏，SG函数是用于为这种有向图游戏提供最优策略的。（SG(x)=0代表x为必败态，SG（x）!=0代表x为必胜态）

对于给定游戏规则下的有向无环图，定义SG（x）=mex{SG(y) | y是x的后继}

容易知道终止状态的SG值一定是0，对于一个x满足SG(x)=0,则对于x的所有后继（经过一次操作可到达的状态）y一定有SG（y）!=0,类似的，对于一个x满足SG(x)！=0,则对于x的所有后继y均有SG（y）==0。

顶点SG值的意义：

当SG（x）=k时，表明对于任何一个0<=i<k,都存在x的一个后继y满足SG(y)=i。

下面给出常见博弈论的解题方法：
把原问题按照游戏规则的差别分解成多个独立的子游戏，并计算每个子游戏下的SG值，最后求出所有子游戏的SG值的异或

```
int f[N], sg[N], vis[N];
//f[]记录可以进行的操作（比如取走多少个石子）
//sg[]记录每种状态的sg值
//vis[x]用于标记出现过的x的后继的sg值
void SG(int n)
{
    memset(sg, 0, sizeof sg); //0是终止状态（必败态），所以sg[0]一定为0
    for(int i = 1; i <= n; ++i)
    {
        memset(vis, 0, sizeof vis);
        for(int j = 1; f[j] <= i; ++j) vis[sg[i-f[j]]] = 1;
        //i-f[j]是状态i进行一次操作可以到达的状态
        for(int j = 0; j <= n; ++j)//求解i的后继的sg值中未出现的最小自然数
            if(!vis[j])
            {
                sg[i] = j;
                break;
            }
    }
}
```

NIM

尼姆游戏的规则：

有n堆石子，数量分别是{a_1,a_2,a_3,…,a_n},两个玩家轮流拿石子，每次从任意一堆中拿走任意数量的石子，拿到最后一个石子的玩家获胜。

尼姆游戏有个极为简单的判断胜负的方法，即做异或运算

（在这之前要了解一下巴什游戏中的P-position和N-position）

定理：

若$a_1⊕a_2⊕a_3⊕…⊕a_n$！=0则先手必胜，此时是N-position
若$a_1⊕a_2⊕a_3⊕…⊕a_n$==0则先手必败也就是后手胜，此时是P-position

思路：证明所有石子异或和为0则先手必输

证明：

1. 反正最终情况就是每堆都为0，先手必输，所以我们考虑怎么把情况转换到这里。
2. 如果异或和的最高位为i，则有一堆石子第i为为1（不然怎么会有i位）
3. 设A1就为那堆石子，其他堆石子异或和设为x，总异或和设为k，则 A1 xor x=k，把A1变成A1 xor k，那么后手面对的则是（A1 xor k）xor x=0，

   举个例子:11001 xor 11100=101，则有（11001 xor 101）xor 11100=0
4. 如果现在的异或和已经为0了（不为最终情况），那么怎么转换异或和都不能为0
5. 好，我们根据3 4点得出：如果先手异或和不为0，可以一步让后手的情况为异或和为0；如果先手异或和为0，那么后手异或和就不为0

ANTI-NIM

定义

游戏规则与Nim类似，只是最后把石子取完的人输。

结论

先手必胜的条件为
①：所有堆的石子数均=1，且有偶数堆。
②：至少有一个堆的石子数>1，且石子堆的异或和≠0。

证明

> 一、当所有堆的石子数均为1时
> （1）：石子异或和(t)=0，即有偶数堆。此时显然先手必胜。
> （2）：t≠0，即有奇数堆。此时显然先手必败。
> 二、当有一堆的石子数>1时，显然t≠0
> （1）：总共有奇数堆石子，此时把>1的那堆取至1个石子，此时便转化为一.（2），先手必胜。
> （2）：总共有偶数堆石子，此时把>1的那堆取完，同样转化为一.（2）,先手必胜。
> 三、当有两堆及以上的石子数>1时
> （1）：t=0，那么可能转化为以下两个子状态：
> ①：至少两堆及以上的石子数>1且t≠0，即转为三.（2）。
> ②：至少一堆石子数>1，由二可知此时必胜。
> （2）：t≠0，根据Nim游戏的证明，可以得到总有一种方法转化为三.（1）状态。
> 观察三我们发现，三.（2）能把三.（1）扔给对面，而对面只能扔给你三.（2）或必胜态。所以当三.（2）时先手必胜。
> 综上，所有堆的石子数均=1且t=0/至少有一个堆的石子数>1且t≠0时，先手必胜。

字符

字符串哈希

$$Hash = \sum_{i=1}^{n} base^{n-i} a_i$$

把字符串看成 进制数，并模一个大数解决溢出
0 < ai < base < mod, gcd(base, mod) == 1;
例如把只有小写字母的字符串的 **ai**设为**si - 'a' + 1**
这些限制都是为了减少冲突可能性
base可以取**131**或**13331**，冲突小
大数不可以取 **2^64**(即ull自然溢出)，因为会被卡掉
大数最好用大质数，可以取**1e9+9**
或者使用双模数(即使用两个模数对一个字符串计算两遍哈希值，根据中国剩余定理，这可以使哈希范围扩大到两数乘积
当然可以类似的使用更多模数)

假设有一个 $S = s_1 s_2 s_3 s_4 s_5$ 的字符串，根据定义，获取其 Hash值如下（我们先忽略MOD，方便理解）：

$hash[0] = 0$

$hash[1] = s_1$

$hash[2] = s_1 * Base + s2$

$hash[3] = s_1 * Base^2 + s2 * Base + s_3$

$hash[4] = s_1 * Base^3 + s2 * Base^2 + s_3 * Base + s_4$

$hash[5] = s_1 * Base^4 + s2 * Base^3 + s_3 * Base^2 + s_4 * Base + s_5$

```
typedef unsigned long long ull;
const int N=1e6+1;
const ull base=131;
const ull mod=1e9+9;
ull has[N], power[N];
void init()
{
    power[0] = 1;
    for (int i = 1; i < N; ++i)
        power[i] = power[i - 1] * base % mod;
}
void Hash(string s)
{
    s = " " + s;
    int len = s.length();
    has[0] = 0;
    for (int i = 1; i <= len; ++i)
        has[i] = (has[i - 1] * base + s[i] - 'a' + 1) % mod;
}
ull getSectionHash(int l, int r) {return (has[r] + mod - has[l-1] * power[r-l+1]
% mod)%mod;}
```

判断回文串

原哈希值和reverse哈希值一样

```
ull rev[N];
void Hash(string s)
{
    ...
    has[0]=0;
    rev[0]=0;
    for(int i=1;i<=len;++i)
    {
        has[i]=(has[i-1]*base+s[i]-'a'+1)%mod;
        rev[i]=(rev[i-1]*base+s[len+1-i]-'a'+1)%mod;
    }
}
ull getRevSectionHash(int l,int r) { return (rev[n-l+1]+mod-has[n-r]*power[r-
l+1]%mod)%mod; }
```

KMP

时间复杂度 $O(n+m)$
n 为主串长度，m 为字串长度

```
char txt[N], str[N];//0-Index
int pmt[N];//P[0]~P[i] 字符串前缀的border长度
void getpmt()
{
    int len=strlen(str);
    pmt[0]=0;
    for(int i=1,j=0;i<len;++i)
    {
        while(j && str[i]!=str[j]) j=pmt[j-1];
        if(str[i] == str[j]) ++j;
        pmt[i] = j;
    }
}
void KMP()
{
    int len1 = strlen(txt), len2 = strlen(str);
    for(int i=0,j=0;i<len1;++i)
    {
        while(j && txt[i]!=str[j]) j=pmt[j-1];
        if(txt[i] == str[j]) ++j;
        if(j == len2)
        {
             //允许重复匹配 j=pmt[j-1];
             j = 0; //不允许重复匹配
            cout<<i-len2+1<<"\n";
        }
    }
}
```

马拉车Manacher

时间复杂度 $O(n)$

对于奇偶字符串的处理，manacher采用的是填充特殊字符的方法，并且在字符串两端都加入不同的字符，防止越界，比如字符串“abbccbba”，增加字符后变成 “@#a#b#b#c#c#b#b#a#&”.

我们设有字符串 S ， $S[l \ldots r]$为串 S 中区间为 $[l, r]$ 的子串，我们用 $d[i]$ 表示以第 i 个字符为中心的回文半径。

板子 I

```
int Manacher(string &s)
{
    vector<int> d(s.size() * 2 + 3);
    string str("@");
    for (char ch : s)
        str += "#", str += ch;
    str += "#$";
    // 我们用mid,r来维护最右回文子串
    // len是最长回文子串的长度
    int r = 0, n = (int)str.size() - 1, len = 0, mid = 0;
    for (int i = 1; i < n; ++i)
    {
        // 判断i是否在最右回文区间内
        if (i <= r)
            d[i] = min(d[(mid << 1) - i], r - i + 1);
        else
            d[i] = 1;
        // 中心扩散法求d[i]
        while (str[i + d[i]] == str[i - d[i]])
            ++d[i];
        if (i + d[i] - 1 > r) // 更新最右回文子串
            mid = i, r = i + d[i] - 1;
        len = max(len, d[i] - 1); // 更新最长回文子串的长度
    }
    return len;
}
```

板子 II

```
char ma[M];             // 隔板字符串
int p[M];               // 以i为中心的最长回文长度=p[i]-1
int manacher(string a) // 求s的最长回文子串长度
{
    int l = 0;
    ma[l++] = '$'; // 插起点隔板
    ma[l++] = '#';
    for (int i = 0; i < a.size(); i++)
    {
        ma[l++] = a[i];
        ma[l++] = '#'; // 插隔板
    }
    ma[l] = 0; // 插终点隔板,此时l=2*a.size()+2
    int res = 0;
    for (int i = 0, mx = 0, id = 0; i < l; ++i) // 以隔板字符串每个点作为对称中心
    {
        p[i] = (mx > i) ? min(p[2 * id - i], mx - i) : 1; // 在最右回文串右端点以内
        while (ma[i + p[i]] == ma[i - p[i]])
            p[i]++; // 朴素算法扩展
        if (i + p[i] > mx)
        {
            mx = i + p[i]; // 更新最右回文右端点
            id = i;         // 更新对称中心
```

```
        }
        res = max(res, p[i] - 1); // 更新最长回文子串长度
    }
    return res;
}
```

杂项

快读

```
template<typename T>inline void read(T &x){
    char c = getchar();x = 0;
    for(;!isdigit(c);c = getchar());
    for(;isdigit(c);c = getchar()) x = ((x<<3)+(x<<1)+(c^48));
}
//整数
inline int Read()
{
    int x=0, f=1; char c=getchar();
    while(c>'9'||c<'0') {if(c=='-') f=-1;c=getchar();}
    while(c>='0'&&c<='9') {x=x*10+c-'0'; c=getchar();}
    return x*f;
}
inline void out(int x)
{
    if(x>=10) out(x/10);
    putchar(x%10+'0');
}
```

__int 128 输入输出

```
//输入
__int128 write_int128()
{
    __int128 ans = 0, f = 1;
    char c = getchar();
    while (!isdigit(c))
    {
        if (c == '-') f = -1;
        c = getchar();
    }
    while (isdigit(c))
    {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans * f;
}
//输出
void print_int128(__int128 x)
{
    if (x < 0) x = -x, putchar('-');
    if (x > 9) print_int128(x / 10);
    putchar(x % 10 + '0');
```

```
}
```

字符流

```
stringstream s;
s << fixed << setprecision(0) << pow(2, n);
string str = s.str();
```

前缀和

```
//普通写法
int pre[N], a[N];
for(int i = 1; i <= n; ++i) pre[i] = pre[i - 1] + a[i];
//区间修改的普通写法
int pre[N], a[N];
struct range{ int l, r, v; }p[M];
for(int i = 1; i <= m; ++i) //遍历区间
    a[p[i].l] += p[i].v, a[p[i].r + 1] -= p[i].v;
for(int i = 1; i <= n; ++i) pre[i] = pre[i - 1] + a[i];
//区间修改的set写法，具有记录作用
set<int>s;
vector<int>v[N];
for(int i = 1; i <= m; ++i)
    v[l].push_back(i), v[r + 1].push_back(-i);
for(int i = 1; i <= n; ++i)
    for(int j : v[i])
    {
        if(j > 0) s.insert(j);
        else s.erase(-j);
        //记录操作添加在这
    }
```

二分

整数二分

左半为真

```
while (l < r)
{
    int mid = l + r + 1 >> 1;//向上取整避免更新l=mid值不变死循环,区间[l,r]被划分成[l,mid-1]和[mid,r]
    if (check(mid)) l = mid;//mid在左半,则边界在右半,mid满足,边界可能在mid
    else r = mid - 1;//mid在右半,则边界在左半,mid不满足,边界最多在mid-1
}//此时l=r,任一都可作为二分结果
```

右半为真

```
while (l < r)
{
    int mid = l + r >> 1;//区间[l,r]被划分成[l,mid]和[mid+1,r]
    if (check(mid)) r = mid;//mid在右半,则边界在左半,mid满足,边界可能在mid
    else l = mid + 1;//mid在左半,则边界在右半,mid不满足,边界至少在mid+1
}//此时l=r,任一都可作为二分结果
```

浮点数二分

```
const db eps = 1e-8;//精度比要求高两位
while (r - l > eps)//或循环一定次数终止
{
    db mid = (l + r) / 2;
    if (check(mid)) r = mid;
    else l = mid;
}//此时l≈r,任一都可作为二分结果
```

三分

```
while (r - l > eps)
{
    mid = (l + r) / 2;
    double fl = f(mid - eps), fr = f(mid + eps);
    if (fl < fr) l = mid; // 这里不写成mid-eps,防止死循环;可能会错过极值,但在误差范围以内所以没关系
    else r = mid;
}
```

区间合并

时间复杂度 $O(nlogn)$

```
pair<int, int> inr[N];
vector<pair<int, int>> merge_inr(int n)
{
    vector<pair<int, int>> res;
    sort(inr + 1, inr + 1 + n);
    for (int i = 1; i <= n; ++i)
    {
        if (res.empty())
        {
            res.push_back(inr[i]);
            continue;
        }
        if (res.back().second >= inr[i].first)
            prev(res.end())->second = max(prev(res.end())->second, inr[i].second);
        else
            res.push_back(inr[i]);
    }
    return res;
}
```

高精度算法操作集

HAA：High Accuracy Algorithm
压位：BIT 进制，压 $index$ 位

```
#define ll long long
const ll BIT = 1e1; //BIT进制
const int IDX = 1; //BIT = 10^IDX
```

```
class HAA
{
public:
    vector<ll> num;
    bool flag = 0;
    HAA() {}
    HAA(bool y) { flag = y; }
    HAA(vector<ll> y) { num = y; }
    HAA(vector<ll> y, bool z) { num = y, flag = z; }
    HAA(ll y) // ll范围内直接修改
    {
        num.clear();
        string temp = to_string(y);
        reverse(temp.begin(), temp.end());
        ll len = temp.size() - 1;
        if (temp[len] == '-') flag = 1, len--, temp.pop_back();
        for (ll i = 0; i <= len; i += IDX)
        {
            ll k = (i + IDX - 1) / IDX, templl = 0; //variable k is not used
            for (ll j=0,bit=1; j<IDX && i+j<=len; ++j, bit*=10) templl+=(temp[i
+ j] - '0')*bit;
            num.push_back(templl);
        }
    }
    HAA(string y) // string直接修改
    {
        num.clear();
        reverse(y.begin(), y.end());
        ll len = y.size() - 1;
        if (y[len] == '-') flag = 1, len--, y.pop_back();
        for (ll i = 0; i <= len; i += IDX)
        {
            ll k = (i + IDX - 1) / IDX, templl = 0; //variable k is not used
            for (ll j=0,bit=1; j<IDX && i+j<=len; ++j, bit*=10) templl += (y[i +
j] - '0')*bit;
            num.push_back(templl);
        }
    }
    HAA habs(HAA z) { return HAA(z.num, 0); }
    friend istream &operator>>(istream &is, HAA &a);
    friend ostream &operator<<(ostream &os, const HAA &a);
    bool operator!() const
    {
        if (num.size() != 1 || num[0]) return false;
        return true;
    }
    bool operator==(const HAA &x) const
    {
        if (num.size() != x.num.size() || flag != x.flag) return false;
        for (int i = num.size() - 1; i >= 0; i--) if (num[i] != x.num[i]) return
false;
        return true;
    }
    bool operator<(const HAA &x) const
    {
        if (flag != x.flag) return flag > x.flag;
        if (num.size() != x.num.size()) return num.size() < x.num.size() ^ flag;
```

```
        for (int i = num.size()-1; i >= 0; i--) if (num[i] != x.num[i]) return
num[i] < x.num[i] ^ flag;
        return false;
    }
    bool operator<=(const HAA &x) const
    {
        if (operator==(x)) return true;
        return operator<(x);
    }
    bool operator!=(const HAA &x) const { return !operator==(x); }
    bool operator>(const HAA &x) const { return x < *this; }
    bool operator>=(const HAA &x) const { return x <= *this; }
    HAA operator+(const HAA &x) const
    {
        if (flag ^ x.flag)
        {
            if (flag) return x.operator-(HAA(num, 0));
            return operator-(HAA(x.num, 0));
        }
        if (num.size() < x.num.size()) return x.operator+(*this);
        HAA res(flag);
        int t = 0; // 进位
        for (int i = 0; i < num.size(); i++)
        {
            t += num[i]; // 逐位相加
            if (i < x.num.size()) t += x.num[i]; // 不足位看作零
            res.num.push_back(t % BIT);
            t /= BIT; // 逢十进一
        }
        if (t) res.num.push_back(t);
        return res;
    }
    HAA operator-(const HAA &x) const // this >= x
    {
        if (flag ^ x.flag)
        {
            if (flag) return HAA(x.num, 1).operator+(*this);
            return operator+(HAA(x.num, 0));
        }
        if (HAA(num, 0) < HAA(x.num, 0)) return HAA(x.operator-(*this).num,
!flag);
        HAA res(flag);
        int t = 0;
        for (int i = 0; i < num.size(); i++)
        {
            t = num[i] - t; // 减借位
            if (i < x.num.size()) t -= x.num[i]; // 逐位相减,不足位看作0
            res.num.push_back((t + BIT) % BIT);
            t < 0 ? t = 1 : t = 0; // 不够从前借一位
        }
        while (res.num.size() > 1 && res.num.back() == 0) res.num.pop_back();
        return res;
    }
    HAA operator*(const HAA &x) const
    {
        HAA res(bool(flag ^ x.flag));
        res.num.resize(num.size() + x.num.size());
        for (int i = 0; i < num.size(); i++)
```

```
            for (int j = 0; j < x.num.size(); j++)
                res.num[i + j] += num[i] * x.num[j];
        for (ll i = 0, t = 0; i < res.num.size(); i++) // i=C.size()-1时t一定<10
        {
            t += res.num[i];
            res.num[i] = t % BIT;
            t /= BIT;
        }
        while (res.num.size() > 1 && res.num.back() == 0) res.num.pop_back(); //
去掉前导0
        return res;
    }
    HAA operator/(const HAA &x) const //IDX<=2使用
    {
        HAA res(bool(flag ^ x.flag)), r, subx = HAA(x.num, false);
        if (HAA(num, 0) < subx)
        {
            res.num.push_back(0);
            r.num.assign(num.begin(), num.end());
            return res;
        }
        int j = subx.num.size();
        r.num.assign(num.end() - j, num.end());
        for (ll k = 0; j <= num.size(); ++j, k = 0)
        {
            while (r >= subx) r = r - subx, k++;
            res.num.push_back(k);
            if (j < num.size()) r.num.insert(r.num.begin(), num[num.size() - j -
1]);
            if (r.num.size() > 1 && r.num.back() == 0) r.num.pop_back();
        }
        reverse(res.num.begin(), res.num.end());
        while (res.num.size() > 1 && res.num.back() == 0) res.num.pop_back(); //
去掉前导0
        return res;
    }
    HAA operator%(const HAA &x) const
    {
        HAA r(flag), subx = HAA(x.num, false);
        if (HAA(num, 0) < subx)
        {
            r.num.assign(num.begin(), num.end());
            return r;
        }
        int j = x.num.size();
        r.num.assign(num.end() - j, num.end());
        for (ll k = 0; j <= num.size(); ++j, k = 0)
        {
            while (r >= x) r = r - x, k++; //variable k is not used
            if (j < num.size()) r.num.insert(r.num.begin(), num[num.size() - j -
1]);
            if (r.num.size() > 1 && r.num.back() == 0) r.num.pop_back();
        }
        return r;
    }
    HAA operator=(const HAA &x)
    {
        num = x.num, flag = x.flag;
```

```
        return *this;
    }
    void operator+=(const HAA &x) { *this = *this + x; }
    void operator-=(const HAA &x) { *this = *this - x; }
    void operator*=(const HAA &x) { *this = *this * x; }
    void operator/=(const HAA &x) { *this = *this / x; }
    void operator%=(const HAA &x) { *this = *this % x; }
    operator ll()
    {
        ll tolong = 0;
        HAA temp = *this % (HAA)(ll)1e18;
        int len = temp.num.size() - 1;
        for (int i = len; i >= 0; i--)
        {
            tolong *= BIT;
            tolong += temp.num[i];
        }
        if (flag) tolong = -tolong;
        return static_cast<ll>(tolong);
    }
    inline void div(ll x)
    {
        for (ll i = (int)num.size() - 1, r = 0; i >= 0; --i)
        {
            r = r * BIT + num[i];
            num[i] = r / x;
            r %= x;
        }
        while (num.size() > 1 && num.back() == 0) num.pop_back(); // 去掉前导0
    }
};
// 为了与iostream标准库兼容,非成员函数重载
istream &operator>>(istream &is, HAA &x)
{
    string temp;
    is >> temp;
    x.num.clear();
    reverse(temp.begin(), temp.end());
    ll len = temp.size() - 1;
    if (temp[len] == '-') x.flag = 1, len--, temp.pop_back();
    for (ll i = 0; i <= len; i += IDX)
    {
        ll k = (i + IDX - 1) / IDX, templl = 0;
        for (ll j=0, bit=1; j < IDX && i+j <= len; ++j, bit *= 10) templl +=
(temp[i + j] - '0') * bit;
        x.num.push_back(templl);
    }
    return is;
}
ostream &operator<<(ostream &os, const HAA &x)
{
    int len = x.num.size() - 1;
    if (x.flag && HAA(x.num, 0) != (HAA)0ll) cout << '-';
    os << x.num[len--];
    for (int i = len; i >= 0; i--) os << setw(IDX) << setfill('0') << x.num[i];
    return os;
}
```