# 一、基础算法

递归和递推

## 二分

二分之前先要排好序

> 一般情况下是 l = mid,r = mid-1
>
> 或 r = mid,l = mid + 1 涉及到特殊情况特殊判断，如求三次方根

```
//仔细分析是求那种情况，最左满足r = mid，最右满足就是l = m
//左边界 r = mid，适用于查找....vooooo v的这种情况
//o为满足条件的情况
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}
//有边界 l = mid.适用于查找oooooov....的情况
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

## 高精度

> 总结：除了高精度加法之外，其余的都要去掉前缀0，除了高精度除法是从高位开始外，其余的都是从低位，注意进位/借位 t,以及余数r

### 1.高精度加法

```
vector<int> add(vector<int> a,vector<int> b)
{
    if(a.size() < b.size()) return add(b,a);
    vector<int> res;
    int t = 0;
    //寸的时候把各位存在低位
    for(int i = 0;i<a.size();i++)
    {
```

```
        t += a[i];
        if(i < b.size()) t+=b[i];
        //这时候结果存的是从高位到低位
        res.push_back(t%10);
        t /= 10;
    }
    //别忘了这个进位
    if(t) res.push_back(t);
    return res;
}
```

## 2.高精度减法

```
bool cmp(vector<int> a,vector<int> b)
{
    if(a.size() != b.size()) return a.size() > b.size();
    //两个都是从低位存到高位
    //两个位数相同  从高到低
    for(int i = a.size()-1;i>=0;i--)
    {
        if(a[i] != b[i]) return a[i]>b[i];
        else continue;
    }
    return true;//两个相等  也可看作a>b
}
vector<int> sub(vector<int> &a,vector<int> &b)
{
    int t = 0;//表示接位
    vector<int> res;
    for(int i = 0;i<a.size();i++)
    {
        t = a[i]-t;
        if(i < b.size()) t-=b[i];
        res.push_back((t+10)%10);
        if(t >= 0) t = 0;//没有借位
        else t = 1;//有一个借位
    }
    //return res;//还要除掉前缀0
    //除掉前缀0  低位在前  高位在后
    while(res.size()>1 && res.back() == 0)
        res.pop_back();
    return res;
}
```

## 3.高精度乘法

```
vector<int> mul(vector<int> &a,int b){
    vector<int> res;
    //t同样表示进位
    int t = 0;
    for(int i = 0;i<a.size();i++){
        t = a[i] * b+t;
        res.push_back( t % 10 );
        t = t/10;
    }
    //if(t) res.push_back(t);
```

```
     while(t) c.push_back(t%10),t/=10;
    //消除前缀0
    while(res.size() > 1 && res.back() == 0) res.pop_back();
    return res;
}
```

### 4.高精度除法

```
//余数为r
vector<int> devide(vector<int> &a,int b,int &r)
{
    r = 0;
    vector<int> res;
    //除法是从高位到低位进行
    for(int i = a.size()-1;i>=0;i--)
    {
       // r = r + a[i];
       r = r*10 + a[i];//高位来的
        res.push_back(r / b);
        r %= b;
    }
    //r是余数
    //去掉前缀0
    reverse(res.begin(),res.end());
    while(res.size() > 1 && res.back() == 0)
          res.pop_back();
    return res;
}
```

## 前缀和及差分

### 一维前缀和

```
s[i] = s[i-1] + w[i];//递推公式
sum(l,r) = s[r] - s[l-1];
```

### 二维前缀和

```
//画图
s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + w[i][j];
sum(x1,y1,x2,y2) = s[x2][y2] - s[x1-1][y2] - s[x2][y1-1] + s[x1-1][y1-1]
```

### 差分

- 一维差分

```
a[i] = s[i] - s[i-1];
add(l,r,c){
    a[l] += c;
    a[r+1] -= c;
}
```

- 二维差分

```
a[i][j] = s[i][j] - s[i][j-1] - s[i-1][j] + s[i-1][j-1];
void add(int x1,int y1,int x2,int y2,int c)
{
    a[x1][y1] += c;
    a[x2+1][y1] -= c;
    a[x1][y2+1] -= c;
    a[x2+1][y2+1] += c;
}
```

## 离散化

> 离散化的几个步骤 先统计整个程序操作的位置，对操作位置进行去重 然后每次操作的时候，映射所有的需要操作的坐标

```
//find函数
//注意这里的find函数是找一个大于等于x的值
int  find(int x)
{
    int l = 0,r = alls.size()-1;
    while(l < r)
    {
        int mid = l + r >> 1;
        if(alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    //下标从1开始
    return l+1;
}
```

```
while(n0--)
    {
        scanf("%d %d",&x,&c);
        op.push_back({x,c});
        //记录要操作的坐标
        alls.push_back(x);
    }
    int l,r;
    while(m0--)
    {
        scanf("%d %d",&l,&r);
        query.push_back({l,r});
        //记录要操作的坐标
        alls.push_back(l);
        alls.push_back(r);
    }
    //去重
    sort(alls.begin(),alls.end());
    alls.erase(unique(alls.begin(),alls.end()),alls.end());
    //要操作某个坐标的时候 先调用find函数映射一下
    for(int i = 0;i<op.size();i++)
    {
        x = op[i].first,c= op[i].second;
        int t_x = find(x);
        a[t_x] += c;
    }
```

```
    //前缀和
    for(int i = 1;i<=alls.size();i++)
    {
        s[i] = s[i-1] + a[i];
    }
    for(int i = 0;i<query.size();i++)
    {
        l = find(query[i].first),r = find(query[i].second);
        //千万记住是l r是离散化之后的值
        cout<<s[r] - s[l-1]<<endl;
    }
```

、枚举、模拟

双指针算法

快速幂

```
int quick_mi(int a,ll b,int mod)
{
    while(b)
    {
        if(b & 1) ans = ans*a%mod;
        b >>= 1;
        a = (ll)a*a%mod;
    }
}
```

分解质因数

```
void divide(int x){
    for(int i = 2;i<=x/i;i++){
        if(x % i == 0){
            printf("%d ",i);
            int p = 0;
            while(x % i == 0){
                x /= i;
                p++;
            }
            printf("%d\n",p);
        }
    }

    if(x > 1) printf("%d 1\n",x);//最后剩下的那个质数
    printf("\n");
}
```

## 排序

快排 归并排序merge_sort()、求逆序对

# 二、数据结构

## 常用数据结构

#### [1]单、双链表

```
//主要是树的add操作
//头插法 插入 a->b的一条路径 路径权值为c
void add(int a,int b,int c)
{
    e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;
}
```

#### [2]单调栈 单调队列

```
//单调栈 求左边/右边第一个比它大的数
//如果有 a[x] >= a[y] && x > y 即a[x]永远都不会用到 所以把它删除
int tt = 0;
for(int i = 1;i<=n;i++)
{
    scanf("%d",&a[i]);
    while(tt && a[i] <= a[sta[tt]]) tt--;
    if(tt) printf("%d ",a[sta[tt]]);
    else printf("-1 ");
    sta[++tt] = i;
}
```

记住单调队列都是取队头元素，这样就不会写错了

```
//单调队列都是取队头元素 队列当中存的都是数组的下标
for(int i = 0;i<n;i++)
{
    cin>>a[i];
    //最左边的坐标只能到达i-k的位置 由于要留一个给当前数 所以是i-k+1
    while(hh <= tt && up[hh] < i - k + 1) hh++;//左边太左
    while(hh <= tt && a[up[tt]] >= a[i]) tt--;
    up[++tt] = i;
    if(i >= k-1) cout<<a[up[hh]]<<" ";
}
cout<<endl;
hh = 0,tt = -1;
for(int i = 0;i<n;i++)
{
    //最大值 单调递减队列
    while(hh <= tt && down[hh] < i - k + 1) hh++;
    while(hh <= tt && a[down[tt]] <= a[i]) tt--;
    down[++tt] = i;
    if(i >= k-1) cout<<a[down[hh]]<<" ";
}
```

**[3]KMP**

用于求next数组

```
    //求next数组
    //注意这里的下标是从1开始的
for(int i = 2,j = 0;i<=n;i++)
{
    while(j && p[i] != p[j+1])   j = ne[j];
    if(p[i] == p[j+1]) j++;
    ne[i] = j;
}
// j = ne[j] 的含义是指[1-ne[j]]部分的字符串和[ne[j] - j]部分的字符串相等
for(int i = 1,j=0;i<=m;i++)
{
    while(j && s[i]!=p[j+1]) j = ne[j];
    if(s[i] == p[j+1]) j++;
    if(j == n)
    {
        printf("%d ",i-n);
        j = ne[j];
    }
}
```

**[4] 哈希表**

添加或者查找某个数

一条链开若干个环，有冲突的点不断放入对应的链条当中

删除是一个特殊的标记

注意映射取模的那个数需要是比范围大的一个质数

```
//模拟散列表
void insert(int x)
{
    //找到映射的点的坐标
    int k = (x % N + N) % N;
    //头插法
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx++;
}
bool find(int x)
{
    int k = (x % N + N) % N;
    for(int i = h[k];i!=-1;i=ne[i])
    {
        if(e[i] == x) return true;
    }
    return false;
}
```

**[5]字符串前缀哈希**

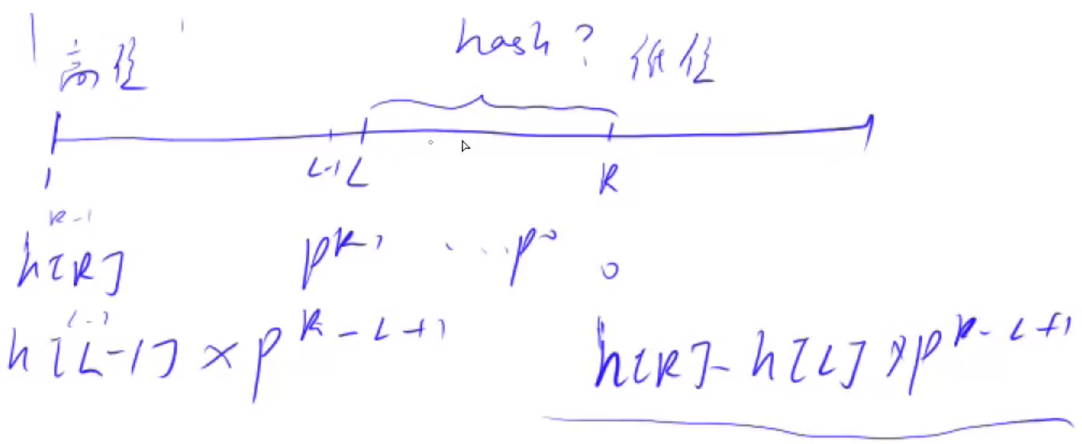
可以用来解决大部分kmp算法才能解决的问题，甚至可以拿来解决大部分问题

> 将这个字符串看作是一个p进制的数，然后再转换成十进制的数，这里假定不会出现冲突，核心思想就是把字符串看成一堆前缀的十进制数，利用差值比较中间两个字符的相同与否

注意

- P进制的p一般情况下取131或者13331 ，为了防止溢出一般要对一个值进行取模，模数Q一般取2^64 直接定义为long long 即可

```
//固定值 p进制
const int P = 131;
//最后结果很大 需要对2^64取模
typedef long long LL;
LL h[N],p[N];
int get(int l,int r)
{
    //l-1的最高位要比r的最高位低P^(r-l+1)次方 所以这里相乘才能得到后面那段的乘积
    return h[r] - h[l-1]*p[r-l+1];
    //比如我们要获取12345五位数字当中的45 就需要把123*100
}
cin>>str+1;
p[0] = 1;
//预处理p数组
for(int i = 1;i<=n;i++)
{
    p[i] = p[i-1]*P;
    //原来字符串相当于从高位到低位
    h[i] = h[i-1]*P + str[i];
    //注意加的是str[i]
}
```



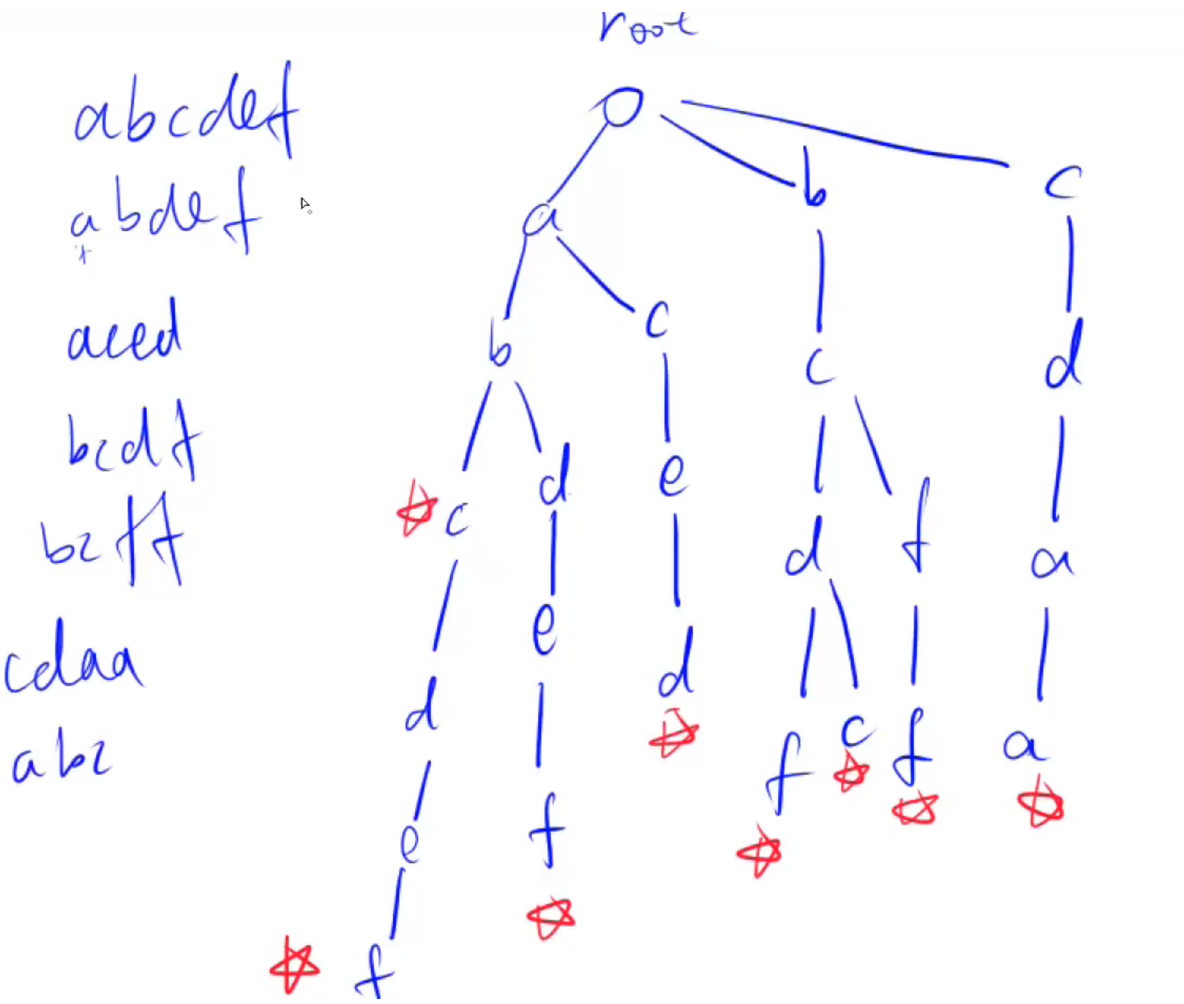

**[6]Tire字典树**

以空间换时间，减少无所谓的比较

高效得存储和查找字符串的集合，一般都要么都是小写字母，要么都是大写字母，要么都是01串

```
int trie[N][26];//每个节点至多26个分支
int cnt[N];
```

```
char str[N];//相当于为每个节点计数
int idx;
void insert(char str[])
{
    int u = 0;
    for(int i = 0;str[i];i++)
    {
        int s = str[i] - 'a';
        if(!trie[u][s]) trie[u][s] = ++idx;
        u = trie[u][s];
    }
    cnt[u]++;
}
int query(char str[])
{
   int u = 0;
   for(int i = 0;str[i];i++)
   {
       int s  = str[i] - 'a';
       if(!trie[u][s]) return 0;
       else u = trie[u][s];
   }
   return cnt[u];
}
```



**[7]并查集**

用于解决一些元素分组，判断元素是否属于同一个集合的问题

并查集相当容易考，2020考过数码段，维护一堆一堆的分类

```
//核心是find函数 查找的时候直接指向
int find(int x)
{
    if(fa[x] != x)  fa[x] = find(fa[x]);//路径记录
    //记住这里是直接return
    return fa[x];
}
//把a所在集合合并到b所在集合 即a的根节点的父节点是b
int f_a = find(a),f_b = find(b);
if(f_a != f_b) fa[f_a] = f_b;
```

食物链这题有很大的借鉴意义，可以多看看

**[8]堆（一般用priority_queue直接用）**

dijkstra()算法优化和相关dp问题优化

```
#include <iostream>
#include <queue>
using namespace std;
int n,m;
int main()
{
    //小根堆
    priority_queue<int,vector<int>,greater<int> >heap;
    cin>>n>>m;
    int x;
    while(n--)
    {
        scanf("%d",&x);
        heap.push(x);
    }
    while(m--)
    {
        printf("%d ",heap.top());
        heap.pop();
    }
    return 0;
}
```

## 常见STL库的使用

**algorithm库的使用**

#### [1] reverse翻转

```
reverse(a.begin(),a.end());
//存放在下标1-n的位置当中 这里自己经常搞错
reverse(a+1,a+1+n);
```

#### [2] unique去重

unique 函数返回去重之后的尾迭指针，即去重元素末尾元素的下一个元素位置,unique本身只是把重复元素放到尾巴后面藏起来了

```
//vector去重
int m = unique(a.begin(),a,end()) - a.begin();
a.erase(unique(a.begin(),a.end()),a.end());
//数组
int m = unique(a+1,a+1+n);
```

#### [3] next_permutation()返回全排列

`next_permutation(a,a+n)` 将a数组重新排列，如果还有全排列则返回true，否则返回false

```
int a[3] = {1,2,3};
do{
    for(int i = 0;i<3;i++) cout<<a[i]<<" ";
    cout<<endl;
}while(next_permutation(a,a+3));
```

#### [4] lower_bound/upper_bound 二分

upper_bound()返回 >x的位置

lower_bound()返回大于等于x的位置

使用之前这个数组需要事先排好序的数组

```
int i = lower_bound(a+1,a+n+1,x) - a;
//查找小于等于x的最大整数
--upper_bound(a.begin(),a.end(),x);
int y = *--upper_bound(a.begin(),a.end(),x);
```

### 1.vector 变长数组

- 支持随机访问，一般操作都在尾部

```
vector<eletype> arr;
arr.size();//返回实际元素个数
arr.empty();//返回是否是空的逻辑判断
//所有容器都支持上述两个方法
arr.clear();//清空vector
vector<int>::iterator it;//迭代器 相当于指针 *it访问其中的元素
arr.begin();//返回第一个元素的迭代器
arr.end();//返回最后一个元素的后一个位置迭代器
arr.front();//返回vector的第一个元素
arr.end();//返回vector的最后一个元素
//支持的操作
arr.push_back(x);
arr.pop_back(x);//弹出来最后一个元素
```

#### 2.queue

- queue是正常的队列，从尾部入队 头部出队

```
queue<eletype> q;
q.push(x);//尾部入队 O(1)
q.pop(x);//头部出队
q.front();//队头元素
q.back();//队尾元素
```

#### 3. priority_queue

- priority_queue 用来实现小根堆和大根堆

  priority_queue默认情况下是大根堆

```
priority_queue<eletype> q;
q.push(x);//x元素入堆
q.pop();//队头元素出堆
q.top();//查询队头元素 最大值
```

priority_queue 中存储的元素类型**必须定义“小于号”**，较大的元素会被放在堆顶。内置的 int, string 等类型本身就可以比较大小。若使用自定义的结构体类型，则需要重载“<”运算符。

```
const double eps = 1e-8;
struct node{
    int id;
    double x,y;
}
//重载小于号
bool operator< (const node&a,const node&b){
    return a.x + eps < b.x || a.x < b.x + eps && a.y<b.u;
}
```

priority实现小根堆

```
typedef pair<int,int> PII;
priority_queue< PII,vector<PII>,greater<PII> > heap;
//PII是以第一个元素为第一个关键字 第二个元素为第二关键字进行排序
```

或

```
//重载符号 以为大的是小的
bool operator<(const node&a,const node&b)
{
    return a.value > b.value;
}
//这时候系统就会认为大的反而小
```

懒惰删除法

懒惰删除法（又称延迟删除法）就是一种应对策略。当遇到删除操作时，仅在优先队列之外做一些特殊的记录（例如记录元素的最新值），用于辨别那些堆中尚未清除的“过时”元素。当从堆顶取出最值时，再检查“最值元素”是不是“过时”的，若是，则重新取出下一个最值。换言之，元素的“删除”被延迟到堆顶进行。

**4.deque**

支持在两端高效插入或删除元素，支持随机访问

```
deque<int> q;
cout<<q[0]<<endl;//随机访问
deque.begin();deque.end();//返回头/尾部迭代器
q.front();q.back();//头部、尾部元素
q.push_back(x);//从队尾入队
q.push_front(y);//从队头入队
q.pop_front();//从队头出列
q.pop_back();//从队尾出队
q.clear();
//上面的方法 时间复杂度都是O(1)
```

**5.set、multiset**

set 元素不能重复、multiset元素可以重复元素，内部实现是一颗红黑树，（平衡树的一种），支持的元素基本相同

存储的元素必须定义 '小于号' 运算符

把n个数插入到集合当中，再次输出时候，集合当中的元素已经从小到大排好了序，时间复杂度是O(nlogn)

```
set<int> s;
multiset<double> s;
s.size(),s.empty(),s.clear();
//set不支持随机访问 只支持双向迭代发放问问
set<int>::iterator it;
it++,it--;//访问下一个元素，指定元素从小到大访问
//++ --操作的时间复杂度是O(log n)
s.begin();//访问集合当中最小元素的迭代器
s.end();//最后一个元素之后的元素的迭代器
s.insert(x);//把一个元素x插入到集合s中，时间复杂度为O(logn)
```

`s.find(x)` 在集合s中查找等于x的元素，并返回指向该元素的迭代器，==若不存在改元素则返回 `s.end()` =,时间复杂度为O(nlogn);

```
if(s.find(x) == s.end()) //集合当中不存在该元素
```

```
//本质上是二分 时间复杂度为O(logn)
s.lower_bound(x) 查找>= x元素最小的一个 并返回迭代器
s.upper_bound(x) 查找>x的元素当中最小一个 并返回迭代器
```

```
s.erase(it);//it是指向元素的迭代器 时间复杂度是O(logn)
//删除set当中所有x的元素
s.erase(X);
//删除至多一个等于x的数
if((it = s.find(X)) != s,end()) s.erase(it);
//返回set当中元素x的个数
s.count(x);//时间复杂度 O(k + logn)
```

#### 6.map

map容器时一个键值对 key-value,key必须定义小于号（说明可以自动排序）

> 在很多时候，map 容器被当作 Hash 表使用，建立从复杂信息 key（如字符串）到简单信息 value（如一定范围内的整数）的映射。
>
> 因为 map 基于平衡树实现，所以它的大部分操作的时间复杂度都在 $O(\log n)$ 级别，略慢于使用 Hash 函数实现的传统 Hash 表。从 C++11 开始，STL 中新增了

```
map<eletype,eletype> mp;
```

size/empty/clear/begin/end

```
mp.size();mp.empty();mp.clear();mp.begin();mp.end();
```

同set类似，map的迭代器也是双向访问，使用*解除访问限制时，得到的自然是一个二元组 `pair<key_value,value>`

```
mp.insert(make_pair(1,2));
mp.insert({1,2});
map<int,int>::iterator it = mp.begin();
pair<int,int> p = *it;
mp.erase(it),h.erase({x,y});
//查找key为x的二元组 不存在返回mp.end();
//返回的时一个迭代器
mp.find(x);
```

`mp[key]`可以很方便得引用键为key的值，当键值对应键不存在的时候，返回的是一个广义的`0`要注意

#### 7.string常用类函数

```
//1.支持直接加减
string s = s1 + s2;
//2.按照字典序可以直接比较
if(s1 < s2) cout<<"yes"<<endl;
//3.length()、size()返回字符串长度
cout<<s.length()<<endl;
//4.在某个位置插入某个字符
```

```
s.insert(2,s1);//2处插入一个，下标从0开始
//插入某个字符串的一部分字符串 迭代器
s.insert(s.begin()+1,s1.begint(),s1.end());
//5.erase
s.erase(iterator);//字符所在迭代器
s.erase(it1,it2);//删除字符串[it1.it2)区间内的字符
//6、substr(pos,len)
string sub = substr(0,4);//从0位置上删除长度为4的字符
//7、find(s1)
s.find(s1);//返回s1在s当中第一次出现的下标位置 下标从0开始
s.find(s1,pos);//从pos位开始查找
//貌似可以利用上面俩个进行kmp算法
//replace()
s.replace(1,2,s1);//从1开始替换长度为2的s1
s.replace(iterator1,iterator2,s1);//[it1,it2]之间的替换为s1

```

### 8.bitset

`bitset<1000> s` 表示一个1000位的二进制数

~s：返回对 bitset s 按位取反的结果。

&, |, ^：返回对两个位数相同的 bitset 执行按位与、或、异或运算的结果。

>>, <<：返回把一个 bitset 右移、左移若干位的结果。

==, !=：比较两个 bitset 代表的二进制数是否相等。

**[ ] 操作符**

s[$k$] 表示 s 的第 $k$ 位，既可以取值，也可以赋值。

在 10000 位二进制数中，最低位为 s[0]，最高位为 s[9999]。

`s.count()` 返回有多少位1

```
s.any();//是否含有一个1
s.none();//是否一个1也没有
s.set();//把s所有位变为1
s.set(k,v);//s的第k位设置为v
s.reset();//把s所有位变成0
s.reset(k);//把s的第k位变成0
s.flip();//s的所有位取反
s.flip(k);//s的第k位取反
```

## 高级数据结构

### 树状数组

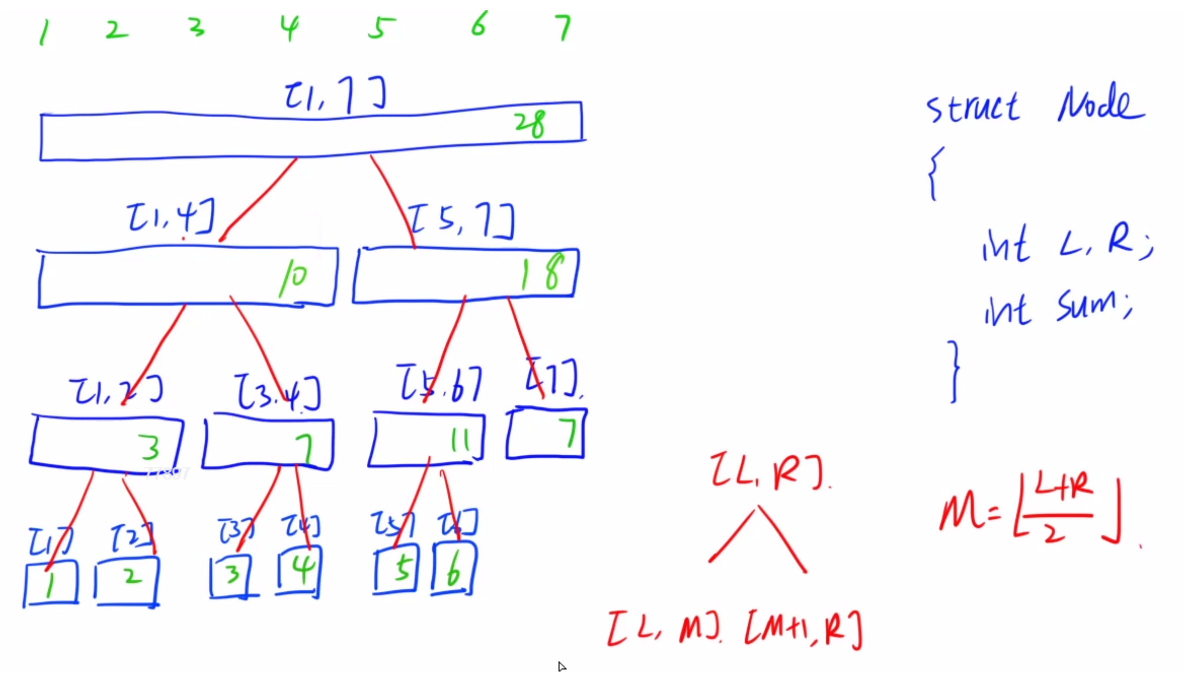
单点修改、区间查询（只能像前缀和那样维护和）

```
int lowbit(int x)
{
    return x&(-x);
}
void add(int x,int c)
{
```

```
    //x的位置加上c
    for(int i = x;i<=n;i+=lowbit(i)) //更新是从 x更新到n
    {
        tr[i] += c;
    }
}
void sum(int x)
{
    //[1,x]之间的值
    int res = 0;
    for(int i = x;i>0;i-=lowbit(i))//求和是从x 到 1
    {
        res += tr[i];
    }
    return res;
}
int main()
{
    //对于读到的每个a[i]构造树状数组
    add(i,a[i])
}
```

**线段树**

线段树 可以用于维护任意的信息




```
//1.单点修改 logn
//2.区间查询  区间整个都在查询范围内 直接返回这一部分的值 logn
struct node{
    int l,r;
    int sum;//维护的信息
}tr[N*4];
int n,m;
int w[N];
void pushup(int u)
{
    tr[u].sum = tr[u<<1].sum + tr[u<<1|1].sum;
    //这里的|1相当于加一是因为右移的时候 低位补0
```

```
}
//建树
void build(int u,int l,int r)
{
    if(l == r) tr[u] = {l,r,w[l]};
    else{
        tr[u] = {l,r};
        //建左右两棵树
        int mid  = l + r >> 1;
        build(u<<1,l,mid),build(u<<1|1,mid+1,r);
        pushup(u);
    }
}
//查询
int query(int u,int l,int r)
{
   //当前区间都包含在查询的区间里
   if(tr[u].l >= l && tr[u].r <= r) return tr[u].sum;
   else{
      int mid = tr[u].l + tr[u].r >> 1;
      int res = 0;
      if(l <= mid) res += query(u<<1,l,r);
      if(r > mid) res += query(u<<1|1,l,r);
      return res;
   }
}
//修改
void modify(int u,int x,int v)
{
    if(tr[u].l == tr[u].r) tr[u].sum += v;
    else{
        int mid = tr[u].l + tr[u].r >> 1;
        if(x <= mid) modify(u<<1,x,v);
        else if(x > mid) modify(u<<1|1,x,v);
        pushup(u);
    }
}
```

# 三、图论

```
总结：
不考虑时间性能，求任意两个点之间的距离--spfa  O（n＾3）
正权边(>=0)    朴素dijkstra n*n
               堆优化dijkstra mlogn
有边数限制的最短路  bellman_ford,last 和 dist数组更新k次
优化的spfa算法
算有负环的最短路     spfa算法 cnt[n]统计每个点被更新的次数，超过边数说明有负环存在

prim          每次找一个最近的点加入到集合当中去，O(n*n)复杂度
kruscal       将边排好序，加到n-1条边为止，并查集表示是否在同一个集合当中

判断二分图       1、2两种颜色进行染色 看看是否有冲突
二分图最大匹配    依次为左边的点寻找对象 看看能不能找到对象
```

# 3.1 DFS、BFS

## DFS模板

```
int g[MAXN];  // 记录状态的数组，可能是多个或者多维的
int ans = 初始情况
void dfs(根据题意传入合适的搜索参数) {
  if (遍历完成) //这个if语句只是一个形式，实际程序中未必有if
      return;  // 或记录解，视情况而定
  if (到达目的地) //这个if语句只是一个形式，实际程序中未必有if
      ans = 从当前解与已有解中选最优;  // 或输出解，视情况而定
  //上面两个可以合成一个
  //注意在遍历两维的图的时候，我们往往取u为一行，这时候只要遍历这一行的情况即可
  for (遍历边界内的所有情况)
    if (解合法) {
      进行操作;//这里的操作/路径都是会被我们枚举的情况替代的，所以说如果有固定点的话，在调用之前就要j
      dfs(继续下去);
      撤回操作;
     //这就是回溯，当到达“死胡同”时，for循环内的dfs调用无法进入for循环，无法递归，只能执行撤回语句，一直回溯到能走下去为止
      //如果还想遍历其它情况可以接着dfs 参考七段码
       dfs(继续下一个操作)
    }
    //当前层所有的都不行之后return掉
}
```

## BFS模板

注意

1.已经遍历过的点不能再遍历或更新 可以设置st数组进行标记

2.对周围的点进行更新的时候不能改变更新这些点的点 要注意back一下

```
void bfs(参数){ //一般是初始状态
    //1.定义queue
    queue<eletype> q;
    //2.定义不同状态的距离、属性
    int d[];
    //3.初始化q、d
    q.push(start);d[start] = 0;
    //如果有终止状态也可以定义
    while(q.size())
    {
        auto t = q.front();
        q.pop();
        if(t == end) return xx;
        for(int i = 0;i<4;i++)
        {
            int x0 = ... y0 = ...
            if(点越界了 break);
            if(状态被遍历过了) break；//这个很重要 千万别忘了
            //对合法的点、且没被遍历过的点 进行更新

            //当前点恢复为原来的状态l
```

```
            state back();
        }
    }
}
```

BFS在遍历的时候是不能走回头路

## 3.2 树和图的BFS、DFS

注意

每个点只能被遍历一次，所以深搜开始的时候，要标记一下数组

```
//树的深度遍历模板
int dfs(int u)
{
   //当前子树往下遍历的最大深度
   //保证每个点只被遍历一次
   st[u] = true;
   //当前节点剩余所有值  子树当中最大值
   int sum = 0,res = 0;
   for(int i = h[u];i!=-1;i=ne[i])
   {
       int x = e[i];
       if(!st[x])
       {
           int k = dfs(x);
           sum += k;//sum是当前节点子节点的和
           res = max(res,k);
       }
   }
   res = max(res,n-sum-1);
   ans = min(res,ans);
   //sum只是统计了当前所有子节点的长度
   return sum+1;
}
```

```
//bfs模板
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
using namespace std;
const int N = 1e6+10;
int h[N],e[N],ne[N],idx;
int n,m;
int dist[N];
bool st[N];
//有向图
void add(int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}
int bfs()
{
   memset(dist,0x3f,sizeof dist);
```

```
    dist[1] = 0;
    st[1] = true;
    queue<int> q;
    q.push(1);
    while(q.size())
    {
       int t = q.front();
       q.pop();
       for(int i = h[t];i!=-1;i=ne[i])
       {
          int j = e[i];
          if(!st[j]){
              dist[j] = dist[t] + 1;
              q.push(j);
              //如果已经被更新了就标记一下
              st[j] = true;
           }
       }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    else return dist[n];
}
```

## 3.3 拓扑排序

Topsort,先找到一系列入度为0的点，将这些点加入到BFS的队列当中去，每次取出队列当中的一个点将和他相邻的点的入度indu--，同时将没有更新的、入度数量减少为0的点加入到队列当中去。

- 将所有入度为0的点加入到队列当中去，并用path记录一下，此时对应st数组置为true
- 用队列中的点来更新相邻点，如果有入度

```
void bfs()
{
    queue<int> q;
    //注意初始入度为0的点不止一个
    for(int i = 1;i<=n;i++)
        if(!ind[i]){
            q.push(i);
            st[i] = true;
        }
    while(q.size())
    {
        int t = q.front();
        q.pop();
        out[cnt++] = t;
        for(int i = h[t];i!=-1;i=ne[i])
        {
            int j = e[i];
            ind[j]--;
            //要不能更新过 且入度为0的点
            if(!ind[j] && !st[j]){
                q.push(j);
            }
        }
    }
    if(cnt < n) puts("-1");
    else{
```

```
        for(int i = 0;i<cnt;i++)
            cout<<out[i]<<" ";
    }
}
```

# 3.4 最短路的几种算法

## 3.4.1堆优化的dijkstra()算法


注意

朴素版dijkstra()和堆优化版的dijkstra()都有一个共同的特点就是，在出队的时候才将对应点的状态置为 `true`，因为dijkstra处理的都是正权边，第一次出队的时候最小

```
//朴素版dijkstra()算法
//点来更新点
bool dijkstra()
{
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    for(int i = 1;i<n;i++)
    {
        int t = -1;
        for(int j = 1;j<=n;j++)
            if(!st[j] &&(t == -1 || dist[t] > dist[j]))
                t = j;
        st[t] = true;
        for(int i = 1;i<=n;i++)
        {
            if(dist[i] > dist[t] + g[t][i])
                dist[i] = dist[t] + g[t][i];
        }
    }
    if(dist[n] > 0x3f3f3f3f/2) return false;
    else return true;
}
```

同bfs相比较，dijkstra()算法是在边出队的时候才是最小值

```
//对比bfs bfs是压入队列的时候就进行标记
//这里的dijkstra()是第一次出队才是最小值 因为压队的时候 可能由其它值更新
void add(int a,int b,int c)
{
```

```
    e[idx] = b,ne[idx] = h[a];
    w[idx] = c,h[a] = idx++;
}
int dijkstra()
{
    priority_queue<PII,vector<PII>,greater<PII> >heap;
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    heap.push({0,1});//dist node
    //st[1] = true;
    while(heap.size())
    {
       PII t = heap.top();
       heap.pop();
       int dis = t.first,x = t.second;
       //因为小根堆不能直接删除 所以堆优化的情况下要跳过已确定的点 防止一直迭代更新
       if(st[x]) continue;
       st[x] = true;
       for(int i = h[x];i!=-1;i=ne[i])
       {
           int j = e[i];
           if(dist[j] > dis + w[i])
           {
               dist[j] = dis + w[i];
               heap.push({dist[j],j});
           }
       }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    else return dist[n];
}
```

### 3.4.2 bellman_ford()算法 有边数限制的最短路

注意

bellman_ford（）算法每次只能由上一次的结果进行更新，所以每次遍历所有边的时候，需要copy一下原来的数组

时间复杂度O(m*n^2)

- 两重循环 第一重循环 边的限制
- 第二重循环 所有边遍历

```
//有边数限制的最短路
//注意存在负环 即最后距离可以为负数
//同时注意最短距离有可能为-1
int bellman_ford()
{
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    for(int i = 1;i<=k;i++)
    {
        //k条边
        memcpy(backup,dist,sizeof dist);
        for(int j = 0;j<m;j++)
        {
```

```
            //x - > y 有向边
            int x = edges[j].a,y = edges[j].b,c = edges[j].w;
            if(dist[y] > backup[x] + c)
            {
                dist[y] = backup[x] + c;
            }
        }
    }
    //注意存在负环 可以小于0x3f3f3f3f 距离也可以是-1麻了
    //if(dist[n] == 0x3f3f3f3f) return -1;
    //if(dist[n] > 0x3f3f3f3f/2) return -1;
    //  else return dist[n];
    return dist[n];
}
```

### 3.4.3 spfa()算法

注意

spfa()算法的st数组是防止一个点多次更新 反复入队,即是用来判断点是否在队列当中,所以出堆的时候st置为false，入队还为true

```
//是否在队列当中
int spfa()
{
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    //表示在队列当中
    st[1] = true;
    queue<int> q;
    q.push(1);
    while(q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;
        for(int i = h[t];i!=-1;i=ne[i])
        {
              int j = e[i];
              if(dist[j] > dist[t] + w[i])
              {
                  dist[j] = dist[t] + w[i];
                  if(!st[j]){
                    q.push(j);
                    st[j] = true;
                  }
              }
        }
    }
    return dist[n];
}
```

```
//spfa判断负环
//
bool spfa()
{
```

```
    queue<int> q;
    //负环可以从任何一点出发
    for (int i = 1; i <= n; i ++ )
    {
        st[i] = true;
        q.push(i);
    }

    while (q.size())
    {
        int t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                //当某个点更新次数为n次以上时候  说明有负环
                if (cnt[j] >= n) return true;
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

## 3.4.4 floyd()算法

特殊的dp 记住就好

注意 g存的是本来图的大小，在初始读入的时候，g要进行赋无穷大的初值

```
void floyd()
{
  for(int i = 1;i<=n;i++) g[i][i] = 0;
  //注意k要放在最外面  解决填空利器
  for(int k = 1;k<=n;k++)
   for(int i = 1;i<=n;i++)
    for(int j = 1;j<=n;j++)

          {
             g[i][j] = min(g[i][j],g[i][k] + g[k][j]);
          }
}
```

## 3.5 最小生成树

定义

一个连通图的生成树是一个极小的连通子图，它包含图中全部的n个顶点，但只有构成一棵树的n-1条边。最小生成树就是所有生成树当中所有边的权值和最小的算法

### 3.5.1 prim()算法生成最小生成树

稠密图，按照点来更新，普里姆算法在找最小生成树时，将顶点分为两类，一类是在查找的过程中已经包含在生成树中的顶点（假设为 A 类），剩下的为另一类（假设为 B 类）。O(N^2)

对于给定的连通网，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A 类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止。所走过的顶点和边就是该连通图的最小生成树。

```
bool prim()
{
    memset(dist,0x3f,sizeof dist);
    for(int i = 0;i<n;i++)
    {
        int t = -1;
        //挑出一个到集合最短的点
        for(int j = 1;j<=n;j++)
           if(!st[j] &&(t == -1 || dist[t] > dist[j]))
               t  = j;
        //到集合当中距离最近的点是无穷 不存在满足条件的点
        if(i && dist[t] > 0x3f3f3f3f / 2) return false;
        //除了第一个点剩余点才能算距离
        if(i) ans += dist[t];
        st[t] = true;
        //更新该点连接点
        for(int j = 1;j<=n;j++)
           //这里更新距离就相当于是求该点到集和之间的距离
           dist[j] = min(dist[j],g[t][j]);
    }
    return true;
}
```

### 3.5.2 kruskal()算法生成最小生成树算法

稀疏图，按照边来更新，通过对所有的边进行排序，依次选出最短的边，这个边的前提要保证不会在已生成的树当中形成回路 O(mlogm)

- 将每条边从小到大进行排序
- 枚举每条边的ab 权重c

  if(a,b) 不联通 将这条边加入集合当中去 并查集来处

```
int find(int x)
{
    if(f[x] != x) f[x] = find(f[x]);
    return f[x];
}
bool kruskal()
```

```
{
    for(int i = 1;i<=m;i++)
    {
        int a = edges[i].a,b = edges[i].b,w = edges[i].w;
        int fa = find(a),fb = find(b);
        //fa == fb代表两个点已经在一个连通块当中了 这个时候如果再连就成环了
        if(fa == fb) continue;
        else{
            //加入
            f[fa] = fb;
            ans += w;
            cnt++;
        }
    }
    //注意结束的条件是边数达到了n-1
    if(cnt == n-1) return true;
    else return false;
}
```

## 3.6 二分图的判断

一个图是二分图，当且仅当图中不含奇数环。

核心思想就是 某个点周围点的颜色不能和这个点的颜色一样

注意图可以是不连通的图

```
bool solve(int node,int c)
{
    color[node] = c;
    //染它周围的点
    for(int i = h[node];i!=-1;i=ne[i])
    {
       int j = e[i];
       if(!color[j]){
           bool t = solve(j,3-c);
           if(!t) return false;
       }else{
           if(color[j] == c) return false;
       }
    }
    return true;
}
//处理不联通的情况
for(int i = 1;i<=m;i++)
{
    if(!color[i]){
        if(!solve(i,1)){
            flag = false;
            break;
        }
    }
}
```

## 3.7 二分图最大匹配

绿帽算法，这个算法不是很熟悉

每次匹配的时候，如果匹配的点已经属于别的点了，那所属的那个点能不能换一个点进行匹配

最后悔的不是做错了，而是错过了

做法就是find（）为每个左边的点找一个右边的点与之匹配

```
bool find(int x)//为x找一个对象
{
    for(int i  =h[x];i!=-1;i=ne[i])
    {
        int j = e[i];
        if(!st[j])
        {
            //如果没有这个st 很可能一直递归下去
            st[j] = true;
            if(!match[j]||find(match[j]))
            {
                //找到了该点就将match设置一下
                match[j] = x;
                return true;
            }
        }
    }
    //所有爱慕的都不行 返回false
    return false;
}
//为每个左边的点找一个匹配
for(int i = 1;i<=n1;i++)
{
    memset(st,false,sizeof st);
    if(find(i)) ans++;
}
```

# 四、数论

质数的线性筛法

**算术基本定理的推论**

在算术基本定理中，若正整数 $N$ 被唯一分解为 $N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$，其中 $c_i$ 都是正整数，$p_i$ 都是质数，且满足 $p_1 < p_2 < \cdots < p_m$，则 $N$ 的正约数集合可写作：

$$\{p_1^{b_1} p_2^{b_2} \cdots p_m^{b_m}\}, \text{其中 } 0 \le b_i \le c_i$$

$N$ 的正约数个数为（$\prod$ 表示连乘积符号，与 $\sum$ 类似）：

$$(c_1+1) * (c_2+1) * \cdots * (c_m+1) = \prod_{i=1}^{m} (c_i+1)$$

$N$ 的所有正约数的和为：

$$\left(1 + p_1 + p_1^2 + \cdots + p_1^{c_1}\right) * \cdots * \left(1 + p_m + p_m^2 + \cdots + p_m^{c_m}\right) = \prod_{i=1}^{m} \left( \sum_{j=0}^{c_i} (p_i)^j \right)$$

# 1.质数部分

1-n中所有数的不同质因数的个数不会超过10个，且所有幂的和不会超过30

## 1.1 1-n当中的质数 O(n) 线性筛法

```
//用于筛出来 2-n当中的质数
for(int i = 2;i<=n;i++)
    {
        if(v[i] == 0)  v[i] = i,prime[++cnt] = i;
        //给当前数i乘上一个质因子
        for(int j = 1;j<=cnt;j++)//数组是从1开始用的
        {
            //利用最小的 当大于v[i]说明不是最小的
            //或者当primes[j]*i超过n的范围的时候 退出循环
            //v[i]里头存的是i当前最小的质因子 如果prime[j]还要大就没必要算
            if(prime[j] > i || prime[j] > n/i) break;
            //prime[j]是最小质因子
            v[i*prime[j]] = prime[j];
        }
    }
```

## 1.2 求某个具体数的质因子 O(√n)

```

void solve(int n)
{
    //利用筛素数的思想去筛 如果碰到一个没有筛掉的就直接全部都
    for(int i =2;i<=n/i;i++)
    {
        int k = 0;
        if(n % i == 0)
        {
             cout<<i<<" ";
             while(n % i == 0)
             {
                  n /= i;
                  //质因数的多少次幂
                  k++;
             }
             cout<<k<<endl;
        }
    }
    //还剩下一个 必然是素数
    //注意这里的n > 1千万别忘了
    if(n > 1) cout<<n<<" "<<1<<endl;
}
```

## 1.3 欧拉函数

1-n中和n互质的数的个数

1~$N$ 中与 $N$ 互质的数的个数被称为欧拉函数，记为 $\varphi(N)$。

若在算术基本定理中，$N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$，则：

$$\varphi(N) = N * \frac{p_1 - 1}{p_1} * \frac{p_2 - 1}{p_2} * \cdots * \frac{p_m - 1}{p_m} = N * \prod_{质数p|N} \left(1 - \frac{1}{p}\right)$$

记住公式(pi-1)/pi就行了，别忘记那个n

约数

```
//辗转相除法  比较少见
// gcd(a,b) == gcd(b,a) == gcd(a-b,b) == gcd(b,a-b)
LL gcd_sub(LL a,LL b)
{
    if(a < b) swap(a,b);
    if(b == 1) return a;
    else return gcd_sub(b,a/b);
}
```

## 欧几里得算法

```
int gcd(int a,int b)
{
    if(!b) return a;
    return gcd(b,a%b);
}
```

# 质数

分解质因数

筛质数 质数的线性筛法

# 约数

试除法求约数 `for(int i = 1;i<=n/i;i++)` 注意是<=

约数个数

基于算数基本定理 约数个数等于(α1+1)*(α2+1)*..α为质因子的指数.

约数之和

算数基本定理，从每个质因子的0次加到最高次就能得到结果

最大公约数 gcd

## 欧拉函数

从定义出发求欧拉函数

筛法求欧拉函数 1-n中每个数的欧拉函数的和

```
void get_primes(int n)
{
    phi[1] = 1;
    for(int i = 2;i<=n;i++)
    {
        if(v[i] == 0) v[i] = i,primes[cnt++] = i,phi[i] = i-1;
        for(int j = 0;j<cnt;j++)
        {
            if(primes[j] > v[i] || primes[j] > n/i) break;
            v[primes[j]*i] = primes[j];
            if(i % primes[j] == 0) phi[i*primes[j]] = phi[i] * primes[j];
            else phi[i*primes[j]] = phi[i]*(primes[j] - 1);
        }
    }
    LL res = 0;
    for(int i = 1;i<=n;i++)
    {
        res += phi[i];
    }
    cout<<res<<endl;
}
```

## 快速幂

### 1.快速幂

```
void quick_mi(int a,int k,int p){
    ll res = 1;
    while(k){
        if(k&1) res = res*a % p;
        //k左移一位
        k >>= 1;
        //注意a在迭代的时候也会爆int
        //a平方等于下一个数
        a  = a*(ll)a % p;
    }
    cout<<res<<endl;
}
```

### 2.快速幂求逆元

费马小定理，注意p是质数

这里b的逆元就是b^(p-2) % p;

（x*a = x/b（mod p)）

```
//即x/b模一个数相当于x*a模一个数，此时a为b的模p的乘法逆元
//当且仅当p为质数 且gcd(p,b) = 1时 a为b^p
int gcd(int a,int b)
```

```cpp
{
    if(!b) return a;
    return gcd(b,a%b);
}
int quick_mi(int a,int b,int p)
{
    LL res = 1;
    while(b)
    {
        if(b & 1) res = (LL)res*a % p;
        b >>= 1;
        a = (LL)a*a % p;
    }
    return res;
}
int main()
{
    int n;cin>>n;
    int a,b;
    while(n--)
    {
        cin>>a>>b;
        if(gcd(a,b) != 1) puts("impossible");
        else{
            cout<<quick_mi(a,b-2,b)<<endl;
        }
    }
    return 0;
}
```

$b \cdot x \equiv 1 \pmod{p}$

$b^{p-1} \equiv 1 \pmod{p}$

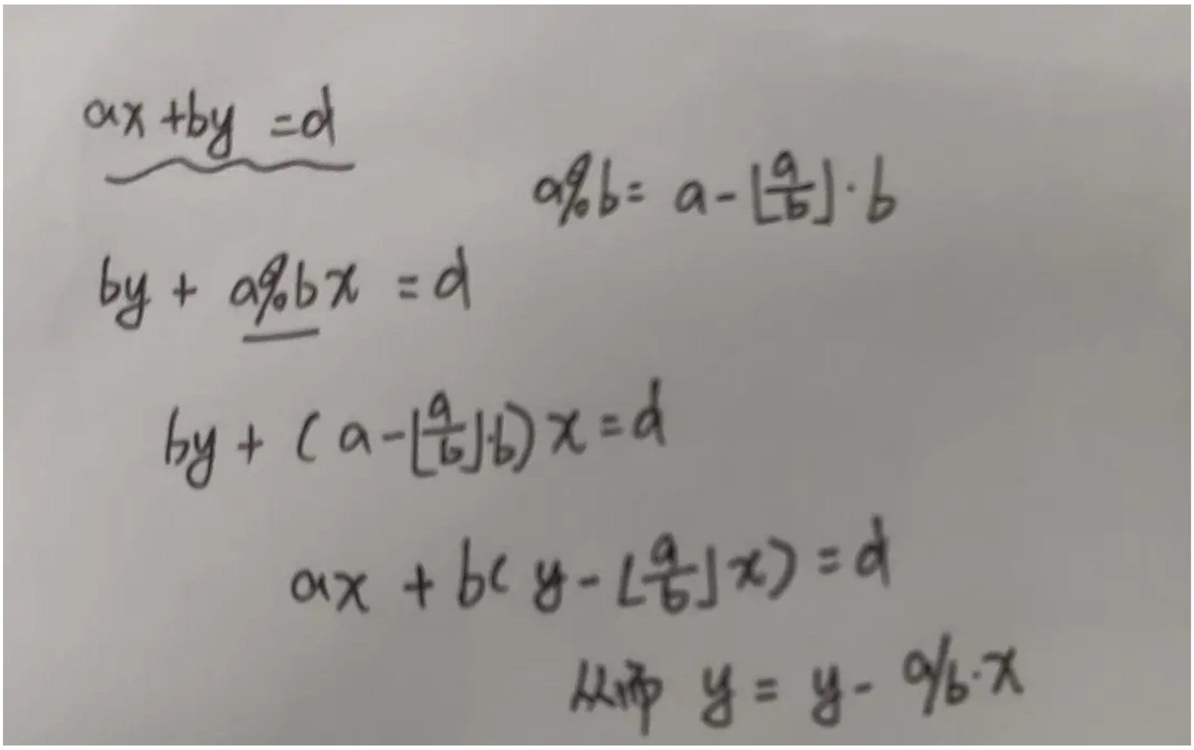
$b \cdot b^{p-2} \equiv 1 \pmod{p}$

# 扩展欧几里得算法

## 1.扩展欧几里得算法

给定 $n$ 对正整数 $a_i, b_i$，对于每对数，求出一组 $x_i, y_i$，使其满足 $a_i \times x_i + b_i \times y_i = gcd(a_i, b_i)$。

这里的变化最好还是自己手动推到一下




```
//加了&x y的值才可以传回去
//返回值仍然是a和b的最大公约数
int exgcd(int a,int b,int &x,int &y)
{
    if(!b){
        x = 1,y = 0;
        return a;
    }
    //by + a%b x = d
    int d = exgcd(b,a%b,y,x);
    //顺序颠倒了  推一下y是变了的  并且变了的y可以传回去
    y = y - a/b*x;
    return d;
}
```

## 2.线性同余方程

给定 $n$ 组数据 $a_i, b_i, m_i$，对于每组数求出一个 $x_i$，使其满足 $a_i \times x_i \equiv b_i \pmod{m_i}$，如果无解则输出 `impossible`。

转换为 a*x = m*y + b   两边同时模m就能恢复到原来样子

然后就可以像上面那样找到一组线性同余方程的解

# 中国剩余定理

# 高斯消元

# 求组合数

## 1.朴素版求组合数

```
void init(){
    f[0][0] = 1;
    for(int i = 1;i<=2000;i++)
      //这里的j一定不能超过i了 不然没意义且会导致结果错误
      for(int j = 0;j<=i;j++)
      {
          f[i][j] = (f[i-1][j-1]%mod + f[i-1][j]%mod) % mod;//第i个选了 没选
      }
}
```

## 2.当a，b达到1e^5

```
    fact[0] = 1;infact[0] = 1;
    for(int i = 1;i<N;i++)
    {
        fact[i] = (LL)fact[i-1] * i%mod;
        infact[i] = (LL)infact[i-1]*quick_mi(i,mod-2,mod)%mod;
    }
    int a,b;
    while(n--)
    {
        scanf("%d %d",&a,&b);
        cout<<fact[a]*(LL)infact[a-b]%mod*infact[b]%mod<<endl;
    }
```

## 3.当a,b达到1e^18

lucas定理

$$C_a^b \equiv C_{a \bmod p}^{b \bmod p} \cdot C_{a/p}^{b/p} \pmod p$$

```
int quick_mi(int a,int b,int p)
{
    int res = 1;
    while(b)
    {
        if(b&1) res = (LL)res*a%p;
        b >>= 1;
        a = a*(LL)a % p;
    }
    return res;
}
int C(int a,int b)
{
```

```
    int res = 1;
    //逆元 元素除以逆元模一个数 等于元素乘以这个逆元再模一个数得到的结果
    for(int i = 1,j = a;i<=b;i++,j--)
    {
        res = (LL) res*j%p;
        res = (LL) res*quick_mi(i,p-2,p) % p;
    }
    return res;
}
//直接算出来C(a,b)的结果
int lucas(LL a,LL b)
{
    if(a < p && b < p) return C(a,b);
    return (LL)C(a%p,b%p)*lucas(a/p,b/p)%p;
}
```

# 五、动态规划

状态表示 分情况讨论

dp的精髓在于如何用一个数代表一类物品

只有01背包问题一维优化版和有依赖的背包问题是从小到大枚举体积

其他的都是从小到大枚举体积

动态规划就是怎么拿上一步的结果推出来这一步的结果

## 5.1 背包问题

给一堆东西选出来一个最佳值

### 5.1.0 记忆化搜索

记忆化搜索 = 深搜的形式 + 动态规划的思想

即 每次搜索的时候，都将子问题的最优解比如`dp[][]`记录下来,每次开始搜索的时候，如果当前值已经被搜索过就可以直接返回，而不用重复搜索相关的子问题

```
int dfs(int x,int y)
{
    //代表已经搜索过了 不用重复计算
    if(dp[x][y] != 0) return dp[x][y];
    //别忘了深搜的边界
    if(边界条件) return;
    //迭代搜索子过程
    dfs();
    //记录搜索的结果
    dp[x][y] = t;
    return t;
}
```

### 5.1.1 01背包问题

每个物品选或者不选 事件复杂度o(n^2)

从前i个物品当中选，体积不超过j的所有选法的集合，属性为体积的最大值

只有当更新的时候是从[i-1]转移到[i]的时候才需要逆序去枚举体积

```
//0-1背包问题 从前i个物品当中选择或者不选择 属性是什么
//体积不超过 体积恰好为多少 体积至少是 体积最多是
//对应的属性不一定就是恰好 根据具体情况有多种bia'ni
for(int i = 1;i<=n;i++)
      for(int j = V;j >= v[i];j--)
       {
           //f[i][j] = f[i-1][j];
           //if(j >= v[i]) f[i][j] = max(f[i-1][j-v[i]]+w[i],f[i][j]);
           //逆序j - v[i] < j 算f[j]的时候f[j-v[i]]还是上一层的结果 故而要逆序
           //如果不是逆序的话本来应该用i-1层的结果来更新结果用了第i层
           f[j] = max(f[j],f[j-v[i]]+w[i]);
       }
```

### 5.1.2 完全背包问题

每个物品有无限多个

同01背包问题的区别：完全背包的i轮状态是由i轮状态本身更新过来的 所以不需要逆序

`f[i][j]` 含义同上

```
    for(int i = 1;i<=n;i++)
      for(int j = v[i];j<=V;j++){
        //f[i][j] = f[i-1][j];
        //同0-1背包相比 只是第i轮的状态还是由i轮更新过来
        //if(j >= v[i]) f[i][j] = max(f[i][j],f[i][j-v[i]]+w[i]);
        //这里不用逆序是因为f[j-v[i]]就是本轮更新过来的 逆序了反而会错误
        f[j] = max(f[j],f[j-v[i]]+w[i]);
        //f[j]  = max(f[j],f[j-v[i]]+w[i]);//直接写这个肯定漏了
}
```

### 5.1.3 多重背包问题

物品个数有限制

#### 暴力版

实在不记得了再用

```
    //与前面两个相比较多了一个数量第i个物品数量k的限制
    //三重循环 依次枚举前i个 体积 数量
    for(int i = 1;i<=n;i++)
      for(int j = 0;j<=V;j++)
        for(int k = 0;k <= num[i] && k*v[i] <= j;k++)
          f[i][j] = max(f[i][j],f[i-1][j-k*v[i]]+k*w[i]);
```

### 二进制优化版

优化思路

对每组的物品进行二进制化 比如一组物品有五个 可以分成 1 2 4 三种情况

选的时候可以凑出 0-5当中任意一个情况，把 1 2 4 三种物品打包放入到新的背包中，所有物品处理完成之后

只要对新的背包进行01背包问题处理就可以了

```
    for(int i = 1;i<=n;i++){
        int v,w,s;
        cin>>v>>w>>s;
        //对每个物品组进行二进制处理
        for(int i = 1;i<=s;i*=2){
            g.push_back({i*v,i*w});
            s -= i;
        }
        //除了刚好而二进制以外还剩下的
        if(s) g.push_back({s*v,s*w});
    }
    //剩下的就是对g当中的物品进行01背包问题的处理
    for(int i = 1;i<=g.size();i++){
        int v = g[i-1].first,w = g[i-1].second;
        for(int j = V;j >= v;j--){
            f[j] = max(f[j],f[j-v] + w);
        }
    }
```

## 5.1.4 分组背包问题

从小到大枚举体积

有 N组物品和一个容量是 V 的背包。

分组背包问题的核心就在于组内的物品都是相互独立的，所以后面 开心的金明那题可以转换为分组背包来处理

每组物品有若干个，同一组内的物品最多只能选一个。

```
//维度优化版 注意分组背包的时候 还是要判断一下体积
for(int i = 1;i<=n;i++)
       for(int j = m;j>=0;j--)
       {
          for(int k = 1;k<=cnt[i];k++)
             if(j >= v[i][k]) f[j] = max(f[j],f[j-v[i][k]]+w[i][k]);
       }
```

总结

对于0-1背包、分组背包、完全背包维度优化之后，都不用处理不选的情况，因为f[j] = f[j]相当于自动考虑过了

```
//三重循环 前i组 体积j  第i组里面选择哪个
for(int i = 1;i<=n;i++){
       //枚举体积
        for(int j = 1;j<=V;j++){
            f[i][j] = f[i-1][j];
            for(int k = 1;k<=s[i];k++){
                //对组内的物品选或者不选进行分析
                if(j >= v[i][k]) f[i][j] = max(f[i][j],f[i-1][j-v[i][k]] + w[i][k]);
            }
        }
    }
```

## 5.1.5 混合背包问题

```
//混合背包问题 是在二进制化里面进行体积的m
for(int i = 1;i<=n;i++)
{
    if(s[i] == 0)
    {
        for(int j = v[i];j<=m;j++)
            f[j] = max(f[j],f[j-v[i]]+w[i]);
    }
    else{
        //单独处理s[i] == -1 和 s[i] > 0的情况
        if(s[i] == -1) s[i] = 1;
        for(int k = 1;k<=s[i];k*=2)
        {
            for(int j = m;j>=k*v[i];j--)
            {
                f[j] = max(f[j],f[j-k*v[i]] + k*w[i]);
            }
            s[i] -= k;
        }
        if(s[i]){
            for(int j = m;j>=s[i]*v[i];j--)
            {
                f[j] = max(f[j],f[j-s[i]*v[i]] + s[i]*w[i]);
            }
        }
    }
}
```

## 5.1.6 有依赖的背包问题

父节点从大到小枚举体积，并且要预留一部分给父节点

子节点从小到大枚举体积

选某个物品必须连着某个物品一起选

Acwing 10.有依赖的背包问题

```
void dfs(int u)
```

```
{
    //分组背包问题
    for(int i = v[u];i<=m;i++) f[u][i] = w[u];
       //对当前结点的边进行遍历
       for(int i = h[u];i!=-1;i = ne[i]){
        //e数组的值是当前边的终点，即儿子结点
        int son = e[i];
        dfs(son);
        //省略了一维i 所以要从大到小枚举 因为默认了加父节点 所以j要大于v[u]
        for(int j = m;j>=v[u];j--){
            //去遍历子节点的组合
            for(int k = 0;k<=j-v[u];k++){
                //这里的f[u][j-k]就相当于除了当前节点son以外，其余的最大值
                f[u][j] = max(f[u][j],f[u][j-k]+f[son][k]);
            }
        }
    }
}
```

Acwing 1074 二叉苹果树

```
void dfs(int u,int fa)
{
    for(int i = h[u];i!=-1;i=ne[i])
    {
        if( e[i] == fa) continue;
        dfs(e[i],u);
        for(int j = m;j>=1;j--)
           for(int k = 0;k<j;k++)
              f[u][j] = max(f[u][j],f[u][j-k-1] + f[e[i]][k]+w[i]);//除了给当
前分支 还要留一点给其它分支
    }
}
```

## 5.1.7 背包问题达到最大价值的时候求方案数

求达到最大价值，有多少种可选方案

//原来的有向无环图是 `f[1][] -> f[2][] -> f[3][]...->f[i-1][]->f[i][j]` 所以正常求路径是从后往前求

最小字典序 将有向无环图改成 `f[i][j] -> f[i-1][] -> f[i-2][]->....->f[2][]->f[1][]` 也是从后往前求 但这时候就是最小字典序

```
//体积为0的时候还是有一种方案数的
g[0] = 1;
for(int i = 1;i<=n;i++)
    for(int j = m;j>=v[i];j--)
    {
        int cnt = 0;
        int maxv = max(f[j],f[j-v[i]]+w[i]);
        if(f[j] == maxv) cnt += g[j] % mod;
        if(f[j-v[i]] + w[i] == maxv) cnt += g[j-v[i]]%mod;
        g[j] = cnt % mod;
        f[j] = maxv;
    }
```

```
int t = 0;
for(int i = 1;i<=m;i++)
{
    t = max(t,f[i]);
}
for(int i = 0;i<=m;i++)
{
    if(f[i] == t){
        ans = (ans%mod + g[i] % mod)%mod;
    }
}
```

### 5.1.8 记录最小字典序路径

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

    int k = m;
    int cnt = 0;
    for(int i = 1;i <= n;i++)
    {
         if(k >= v[i] && f[i][k] == f[i+1][k-v[i]] + w[i])
         {
             cout<<i<<" ";
             k -= v[i];
         }
    }
```

## 5.2 线性DP

### 5.2.1 数字三角形

从起始点 按照 上下左右/左下右下等顺序走到终点得到的最大值

一般 `f[i][j]` 就表示起始点走到（i，j）的所有路程的集合，然后对集合进行分析以求出状态转移方程，从而得到最后的结果

898.数字三角形

https://www.acwing.com/problem/content/900/

### 5.2.2 最长上升子序列模型

题意 给定一个序列 求最长的上升子序列的长度

设 `f[i]` 为以第i个点结尾的上升子序列 属性为最长值

注意

f[i]前面最长的不一定就是f[i-1]，f[i-1]只是代表以a[i-1]结尾的最长的一个

## 暴力双重循环版

```
 for(int i = 1;i<=n;i++) f[i] = 1;
   for(int i = 1;i<=n;i++)
   {
     for(int j = 1;j<i;j++){
         if(a[i] > a[j]){
             f[i] = max(f[i],f[j]+1);
         }
     }
   }
```

895.最长上升子序列模型

https://www.acwing.com/problem/content/897/

## 单调栈优化

其实这题就是一道数据结构

因为要求的是最长上升的那个队列，所以可以用一个单调上升的栈来维护，只有当栈里面的元素越小，这个栈才有可能维护得越大，所以对于一个元素有下面两种情况来解决

- 如果 `a[i] > stack.top()` `a[i]` 入栈
- 找到一个比 `a[i]` 大于等于的数 进行替换即可

二分nlogn

```
int find(int x){
    int l = 0,r = tt-1;
    while(l < r){
        int mid = l + r >> 1;
        if(s[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}
int main(){
    cin>>n;
    for(int i = 0;i<n;i++)
       scanf("%d",&a[i]);

    s[tt++] = a[0];
    for(int i = 1;i<n;i++)
    {
        if(a[i] > s[tt-1]) s[tt++] = a[i];
        else{
            int x = find(a[i]);//第一个大于等于a[i]的数
            s[x] = a[i];
        }
    }
    printf("%d",tt);
    return 0;
}
```

## 5.3 状态机模型

## 5.4 区间dp

- 链式区间dp O(n3)
- 环形区间dp O(n3)

```
//迭代式 常用
//len一般从2开始
//一般情况下len为表示状态可以计算的最小值
for(int len = 可以计算的最小长度; len<=用到的最大的长度;len++)
    for(int l = 1;l+len-1<=2n或者n;l++)
    {
        int r = l + len - 1;
        for(int k = l + 可以使得f数组有意义的最小长度;k<r;k++)//注意这里是k<r！！！不能取到r
            //是k还是k+1根据具体题目的定义来
            f[l][r] = max/min(f[l][r],f[l][k]+f[k+1][r]+具体情况计算的值)
    }
//对于不同的开头 结果可能也不同
int ans = 0;
for(int i = 1;i<=n;i++)
     ans = max(f[i][i+n+x],ans);
//记忆化搜索
碰到已经搜索过的状态则return
```