

# 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

```
1 //仔细分析是求那种情况，最左满足  $r = mid$ ，最右满足就是  $l = m$ 
2 //左边界  $r = mid$ ，适用于查找...vooooo v的这种情况
3 //o为满足条件的情况
4 int bsearch_1(int l, int r)
5 {
6     while (l < r)
7     {
8         int mid = l + r >> 1;
9         if (check(mid)) r = mid;
10        else l = mid + 1;
11    }
12    return l;
13 }
14 //有边界  $l = mid$ .适用于查找ooooov....的情况
15 int bsearch_2(int l, int r)
16 {
17     while (l < r)
18     {
19         int mid = l + r + 1 >> 1;
20         if (check(mid)) l = mid;
21         else r = mid - 1;
22     }
23     return l;
24 }
```

## 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

```
1 vector<int> add(vector<int> a,vector<int> b)
2 {
3     if(a.size() < b.size()) return add(b,a);
4     vector<int> res;
5     int t = 0;
6     //寸的时候把各位存在低位
7     for(int i = 0;i<a.size();i++)
8     {
```

```

9         t += a[i];
10        if(i < b.size()) t+=b[i];
11        //这时候结果存的是从高位到低位
12        res.push_back(t%10);
13        t /= 10;
14    }
15    //别忘了这个进位
16    if(t) res.push_back(t);
17    return res;
18 }

```

## 2.高精度减法

```

1  bool cmp(vector<int> a,vector<int> b)
2  {
3      if(a.size() != b.size()) return a.size() > b.size();
4      //两个都是从低位存到高位
5      //两个位数相同 从高到低
6      for(int i = a.size()-1;i>=0;i--)
7      {
8          if(a[i] != b[i]) return a[i]>b[i];
9          else continue;
10     }
11     return true;//两个相等 也可看作a>b
12 }
13 vector<int> sub(vector<int> &a,vector<int> &b)
14 {
15     int t = 0;//表示接位
16     vector<int> res;
17     for(int i = 0;i<a.size();i++)
18     {
19         t = a[i]-t;
20         if(i < b.size()) t-=b[i];
21         res.push_back((t+10)%10);
22         if(t >= 0) t = 0;//没有借位
23         else t = 1;//有一个借位
24     }
25     //return res;//还要除掉前缀0
26     //除掉前缀0 低位在前 高位在后
27     while(res.size()>1 && res.back() == 0)
28         res.pop_back();
29     return res;
30 }

```

## 3.高精度乘法

```

1  vector<int> mul(vector<int> &a,int b){
2      vector<int> res;
3      //t同样表示进位
4      int t = 0;
5      for(int i = 0;i<a.size();i++){
6          t = a[i] * b+t;
7          res.push_back( t % 10 );
8          t = t/10;
9      }
10     //if(t) res.push_back(t);

```

```

11     while(t) c.push_back(t%10),t/=10;
12     //消除前缀0
13     while(res.size() > 1 && res.back() == 0) res.pop_back();
14     return res;
15 }

```

## 4.高精度除法

```

1  //余数为r
2  vector<int> divide(vector<int> &a,int b,int &r)
3  {
4      r = 0;
5      vector<int> res;
6      //除法是从高位到低位进行
7      for(int i = a.size()-1;i>=0;i--)
8      {
9          // r = r + a[i];
10         r = r*10 + a[i]; //高位来的
11         res.push_back(r / b);
12         r %= b;
13     }
14     //r是余数
15     //去掉前缀0
16     reverse(res.begin(),res.end());
17     while(res.size() > 1 && res.back() == 0)
18         res.pop_back();
19     return res;
20 }

```

## 前缀和及差分

### 一维前缀和

```

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

```

### 二维前缀和

```

1  //画图
2  s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + w[i][j];
3  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

```

1  a[i] = s[i] - s[i-1];
2  add(l,r,c){
3      a[l] += c;
4      a[r+1] -= c;
5  }

```

- 二维差分

```

1  a[i][j] = s[i][j] - s[i][j-1] - s[i-1][j] + s[i-1][j-1];
2  void add(int x1,int y1,int x2,int y2,int c)
3  {
4      a[x1][y1] += c;
5      a[x2+1][y1] -= c;
6      a[x1][y2+1] -= c;
7      a[x2+1][y2+1] += c;
8  }

```

## 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

```

1  //find函数
2  //注意这里的find函数是找一个大于等于x的值
3  int find(int x)
4  {
5      int l = 0,r = alls.size()-1;
6      while(l < r)
7      {
8          int mid = l + r >> 1;
9          if(alls[mid] >= x) r = mid;
10         else l = mid + 1;
11     }
12     //下标从1开始
13     return l+1;
14 }

```

```

1  while(n0--)
2  {
3      scanf("%d %d",&x,&c);
4      op.push_back({x,c});
5      //记录要操作的坐标
6      alls.push_back(x);
7  }
8  int l,r;
9  while(m0--)
10 {
11     scanf("%d %d",&l,&r);
12     query.push_back({l,r});
13     //记录要操作的坐标
14     alls.push_back(l);
15     alls.push_back(r);
16 }
17 //去重
18 sort(alls.begin(),alls.end());
19 alls.erase(unique(alls.begin(),alls.end()),alls.end());
20 //要操作某个坐标的时候 先调用find函数映射一下
21 for(int i = 0;i<op.size();i++)
22 {
23     x = op[i].first,c= op[i].second;
24     int t_x = find(x);
25     a[t_x] += c;
26 }

```

```

27 //前缀和
28 for(int i = 1;i<=alls.size();i++)
29 {
30     s[i] = s[i-1] + a[i];
31 }
32 for(int i = 0;i<query.size();i++)
33 {
34     l = find(query[i].first),r = find(query[i].second);
35     //千万记住是l r是离散化之后的值
36     cout<<s[r] - s[l-1]<<endl;
37 }

```

、枚举、模拟

双指针算法

快速幂

```

1 int quick_mi(int a,ll b,int mod)
2 {
3     while(b)
4     {
5         if(b & 1) ans = ans*a%mod;
6         b >>= 1;
7         a = (ll)a*a%mod;
8     }
9 }

```

分解质因数

```

1 void divide(int x){
2     for(int i = 2;i<=x/i;i++){
3         if(x % i == 0){
4             printf("%d ",i);
5             int p = 0;
6             while(x % i == 0){
7                 x /= i;
8                 p++;
9             }
10            printf("%d\n",p);
11        }
12    }
13
14    if(x > 1) printf("%d 1\n",x); //最后剩下的那个质数
15    printf("\n");
16 }

```

## 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

```
1 //主要是树的add操作
2 //头插法 插入 a->b的一条路径 路径权值为c
3 void add(int a,int b,int c)
4 {
5     e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;
6 }
```

#### [2]单调栈 单调队列

```
1 //单调栈 求左边/右边第一个比它大的数
2 //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
3 int tt = 0;
4 for(int i = 1;i<=n;i++)
5 {
6     scanf("%d",&a[i]);
7     while(tt && a[i] <= a[sta[tt]]) tt--;
8     if(tt) printf("%d ",a[sta[tt]]);
9     else printf("-1 ");
10    sta[++tt] = i;
11 }
```

记住单调队列都是取队头元素，这样就不会写错了

```
1 //单调队列都是取队头元素 队列当中存的都是数组的下标
2 for(int i = 0;i<n;i++)
3 {
4     cin>>a[i];
5     //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
6     while(hh <= tt && up[hh] < i - k + 1) hh++;//左边太左
7     while(hh <= tt && a[up[tt]] >= a[i]) tt--;
8     up[++tt] = i;
9     if(i >= k-1) cout<<a[up[hh]]<<" ";
10 }
11 cout<<endl;
12 hh = 0,tt = -1;
13 for(int i = 0;i<n;i++)
14 {
15     //最大值 单调递减队列
16     while(hh <= tt && down[hh] < i - k + 1) hh++;
17     while(hh <= tt && a[down[tt]] <= a[i]) tt--;
18     down[++tt] = i;
19     if(i >= k-1) cout<<a[down[hh]]<<" ";
20 }
```

### [3]KMP

用于求next数组

```
1 //求next数组
2 //注意这里的下标是从1开始的
3 for(int i = 2, j = 0; i <= n; i++)
4 {
5     while(j && p[i] != p[j+1]) j = ne[j];
6     if(p[i] == p[j+1]) j++;
7     ne[i] = j;
8 }
9 // 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
10 for(int i = 1, j = 0; i <= m; i++)
11 {
12     while(j && s[i] != p[j+1]) j = ne[j];
13     if(s[i] == p[j+1]) j++;
14     if(j == n)
15     {
16         printf("%d ", i - n);
17         j = ne[j];
18     }
19 }
```

### [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

```
1 //模拟散列表
2 void insert(int x)
3 {
4     //找到映射的点的坐标
5     int k = (x % N + N) % N;
6     //头插法
7     e[idx] = x;
8     ne[idx] = h[k];
9     h[k] = idx++;
10 }
11 bool find(int x)
12 {
13     int k = (x % N + N) % N;
14     for(int i = h[k]; i != -1; i = ne[i])
15     {
16         if(e[i] == x) return true;
17     }
18     return false;
19 }
```

## [5]字符串前缀哈希

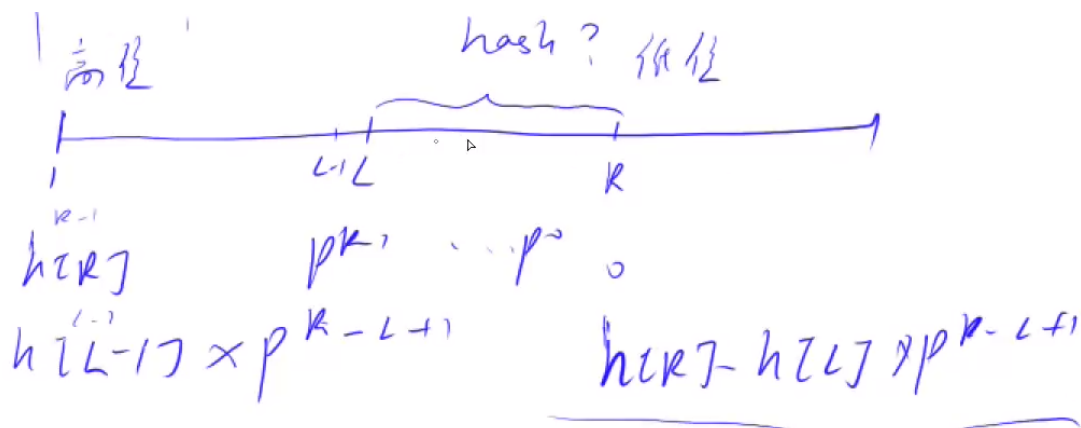
可以用来解决大部分kmp算法才能解决的问题，甚至可以拿来解决大部分问题

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

```
1 //固定值 p进制
2 const int P = 131;
3 //最后结果很大 需要对2^64取模
4 typedef long long LL;
5 LL h[N],p[N];
6 int get(int l,int r)
7 {
8     //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
9     return h[r] - h[l-1]*p[r-l+1];
10    //比如我们要获取12345五位数字当中的45 就需要把123*100
11 }
12 cin>>str+1;
13 p[0] = 1;
14 //预处理p数组
15 for(int i = 1;i<=n;i++)
16 {
17     p[i] = p[i-1]*P;
18     //原来字符串相当于从高位到低位
19     h[i] = h[i-1]*P + str[i];
20     //注意加的是str[i]
21 }
```



## [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

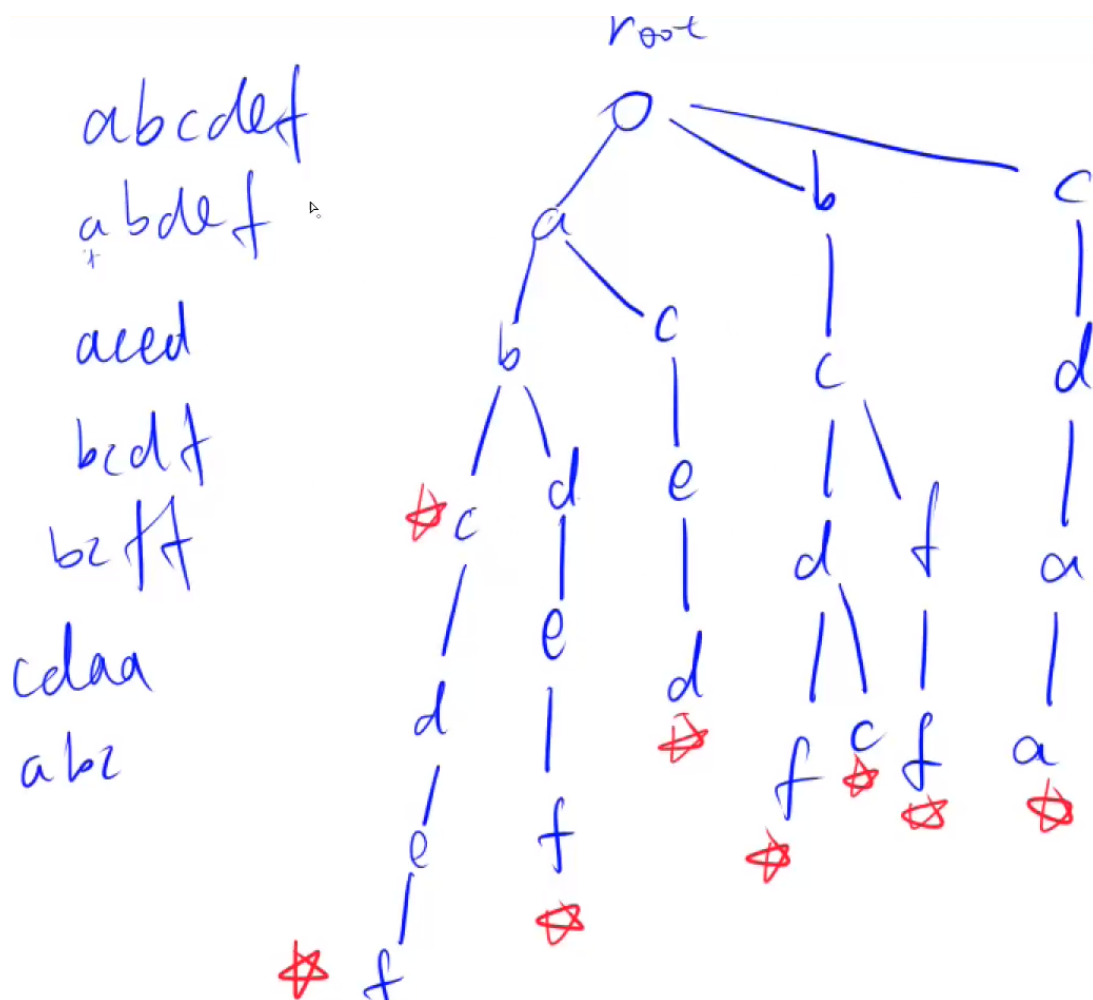
```
1 int trie[N][26]; //每个节点至多26个分支
2 int cnt[N];
```



```

3 char str[N]; //相当于为每个节点计数
4 int idx;
5 void insert(char str[])
6 {
7     int u = 0;
8     for(int i = 0; str[i]; i++)
9     {
10         int s = str[i] - 'a';
11         if(!trie[u][s]) trie[u][s] = ++idx;
12         u = trie[u][s];
13     }
14     cnt[u]++;
15 }
16 int query(char str[])
17 {
18     int u = 0;
19     for(int i = 0; str[i]; i++)
20     {
21         int s = str[i] - 'a';
22         if(!trie[u][s]) return 0;
23         else u = trie[u][s];
24     }
25     return cnt[u];
26 }

```



## [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

```
1 //核心是find函数 查找的时候直接指向
2 int find(int x)
3 {
4     if(fa[x] != x) fa[x] = find(fa[x]); //路径记录
5     //记住这里是直接return
6     return fa[x];
7 }
8 //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
9 int f_a = find(a), f_b = find(b);
10 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

```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4 int n,m;
5 int main()
6 {
7     //小根堆
8     priority_queue<int,vector<int>,greater<int> >heap;
9     cin>>n>>m;
10    int x;
11    while(n-->0)
12    {
13        scanf("%d",&x);
14        heap.push(x);
15    }
16    while(m-->0)
17    {
18        printf("%d ",heap.top());
19        heap.pop();
20    }
21    return 0;
22 }
```

## 常见STL库的使用

### algorithm库的使用

## [1] reverse翻转

```
1 reverse(a.begin(),a.end());
2 //存放在下标1-n的位置当中 这里自己经常搞错
3 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

```
1 //vector去重
2 int m = unique(a.begin(),a.end()) - a.begin();
3 a.erase(unique(a.begin(),a.end()),a.end());
4 //数组
5 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

```
1 int a[3] = {1,2,3};
2 do{
3     for(int i = 0;i<3;i++) cout<<a[i]<<" ";
4     cout<<endl;
5 }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

```
1 int i = lower_bound(a+1,a+n+1,x) - a;
2 //查找小于等于x的最大整数
3 --upper_bound(a.begin(),a.end(),x);
4 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

```

1  vector<etype> arr;
2  arr.size();//返回实际元素个数
3  arr.empty();//返回是否是空的逻辑判断
4  //所有容器都支持上述两个方法
5  arr.clear();//清空vector
6  vector<int>::iterator it;//迭代器 相当于指针 *it访问其中的元素
7  arr.begin();//返回第一个元素的迭代器
8  arr.end();//返回最后一个元素的后一个位置迭代器
9  arr.front();//返回vector的第一个元素
10 arr.back();//返回vector的最后一个元素
11 //支持的操作
12 arr.push_back(x);
13 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

```

1  queue<etype> q;
2  q.push(x);//尾部入队 O(1)
3  q.pop(x);//头部出队
4  q.front();//队头元素
5  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

```

1  priority_queue<etype> q;
2  q.push(x);//x元素入堆
3  q.pop();//队头元素出堆
4  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。

```

1  const double eps = 1e-8;
2  struct node{
3      int id;
4      double x,y;
5  }
6  //重载小于号
7  bool operator< (const node&a,const node&b){
8      return a.x + eps < b.x || a.x < b.x + eps && a.y<b.y;
9  }

```

priority实现小根堆

```

1  typedef pair<int,int> PII;
2  priority_queue< PII,vector<PII>,greater<PII> > heap;
3  //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

```
1 //重载符号 以为大的是小的
2 bool operator<(const node&a,const node&b)
3 {
4     return a.value > b.value;
5 }
6 //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.dequeue

支持在两端高效插入或删除元素，支持随机访问

```
1 dequeue<int> q;
2 cout<<q[0]<<endl;//随机访问
3 dequeue.begin();dequeue.end();//返回头/尾部迭代器
4 q.front();q.back();//头部、尾部元素
5 q.push_back(x);//从队尾入队
6 q.push_front(y);//从队头入队
7 q.pop_front();//从队头出列
8 q.pop_back();//从队尾出队
9 q.clear();
10 //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  $O(n\log n)$

```
1 set<int> s;
2 multiset<double> s;
3 s.size(),s.empty(),s.clear();
4 //set不支持随机访问 只支持双向迭代器访问
5 set<int>::iterator it;
6 it++,it--;//访问下一个元素，指定元素从小到大访问
7 //++ --操作的时间复杂度是O(log n)
8 s.begin();//访问集合当中最小元素的迭代器
9 s.end();//最后一个元素之后的元素的迭代器
10 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  $O(n\log n)$ ;

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

```
1 | //本质上是二分 时间复杂度为 $O(\log n)$ 
2 | s.lower_bound(x) 查找 $\geq x$ 元素最小的一个 并返回迭代器
3 | s.upper_bound(x) 查找 $> x$ 的元素当中最小一个 并返回迭代器
```

```
1 | s.erase(it); //it是指向元素的迭代器 时间复杂度是 $O(\log n)$ 
2 | //删除set当中所有x的元素
3 | s.erase(x);
4 | //删除至多一个等于x的数
5 | if((it = s.find(x)) != s.end()) s.erase(it);
6 | //返回set当中元素x的个数
7 | s.count(x); //时间复杂度  $O(k + \log n)$ 
```

## 6.map

map容器是一个键值对 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

```
1 | map<etype, etype> mp;
```

size/empty/clear/begin/end

```
1 | 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>`

```
1 | mp.insert(make_pair(1,2));
2 | mp.insert({1,2});
3 | map<int,int>::iterator it = mp.begin();
4 | pair<int,int> p = *it;
5 | mp.erase(it), h.erase({x,y});
6 | //查找key为x的二元组 不存在返回mp.end();
7 | //返回的时一个迭代器
8 | 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 | //1. 支持直接加减
2 | string s = s1 + s2;
3 | //2. 按照字典序可以直接比较
4 | if(s1 < s2) cout<<"yes"<<endl;
5 | //3.length()、size() 返回字符串长度
6 | cout<<s.length()<<endl;
7 | //4. 在某个位置插入某个字符
```

```

8 s.insert(2,s1);//2处插入一个，下标从0开始
9 //插入某个字符串的一部分字符串 迭代器
10 s.insert(s.begin()+1,s1.begin(),s1.end());
11 //5.erase
12 s.erase(iterator);//字符所在迭代器
13 s.erase(it1,it2);//删除字符串[it1,it2)区间内的字符
14 //6、substr(pos,len)
15 string sub = substr(0,4);//从0位置上删除长度为4的字符
16 //7、find(s1)
17 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
18 s.find(s1,pos);//从pos位开始查找
19 //貌似可以利用上面两个进行kmp算法
20 //replace()
21 s.replace(1,2,s1);//从1开始替换长度为2的s1
22 s.replace(iterator1,iterator2,s1);//[it1,it2)之间的替换为s1
23

```

## 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

```

1 s.any();//是否含有一个1
2 s.none();//是否一个1也没有
3 s.set();//把s所有位变为1
4 s.set(k,v);//s的第k位设置为v
5 s.reset();//把s所有位变成0
6 s.reset(k);//把s的第k位变成0
7 s.flip();//s的所有位取反
8 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)

```

1 int lowbit(int x)
2 {
3     return x&(-x);
4 }
5 void add(int x,int c)
6 {

```

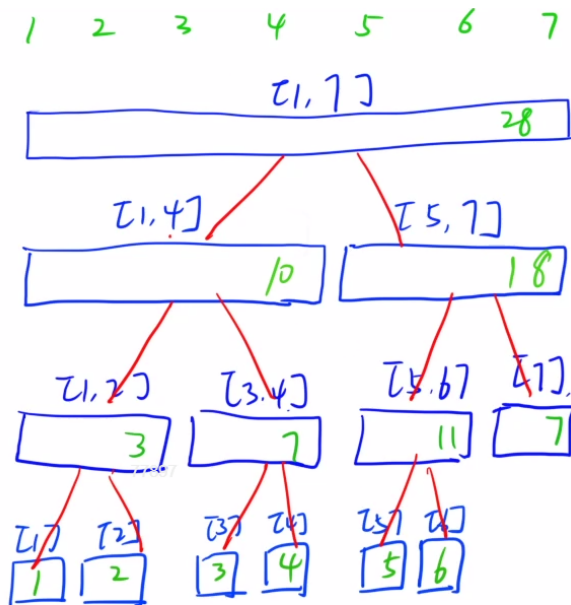
```

7 //x的位置加上c
8 for(int i = x; i <= n; i += lowbit(i)) //更新是从 x更新到n
9 {
10     tr[i] += c;
11 }
12 }
13 void sum(int x)
14 {
15     //[1,x]之间的值
16     int res = 0;
17     for(int i = x; i > 0; i -= lowbit(i)) //求和是从x 到 1
18     {
19         res += tr[i];
20     }
21     return res;
22 }
23 int main()
24 {
25     //对于读到的每个a[i]构造树状数组
26     add(i, a[i])
27 }

```

## 线段树

线段树 可以用于维护任意的信息



$[L, R]$   
 $\swarrow \searrow$   
 $[L, M], [M+1, R]$

```

struct Node
{
    int L, R;
    int sum;
}

```

$$M = \lfloor \frac{L+R}{2} \rfloor$$

```

1 //1.单点修改 logn
2 //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
3 struct node{
4     int l, r;
5     int sum; //维护的信息
6 } tr[N*4];
7 int n, m;
8 int w[N];
9 void pushup(int u)
10 {
11     tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
12     //这里的|1相当于加一是因为右移的时候 低位补0

```



```

13 }
14 //建树
15 void build(int u,int l,int r)
16 {
17     if(l == r) tr[u] = {l,r,w[l]};
18     else{
19         tr[u] = {l,r};
20         //建左右两棵树
21         int mid = l + r >> 1;
22         build(u<<1,l,mid),build(u<<1|1,mid+1,r);
23         pushup(u);
24     }
25 }
26 //查询
27 int query(int u,int l,int r)
28 {
29     //当前区间都包含在查询的区间里
30     if(tr[u].l >= l && tr[u].r <= r) return tr[u].sum;
31     else{
32         int mid = tr[u].l + tr[u].r >> 1;
33         int res = 0;
34         if(l <= mid) res += query(u<<1,l,r);
35         if(r > mid) res += query(u<<1|1,l,r);
36         return res;
37     }
38 }
39 //修改
40 void modify(int u,int x,int v)
41 {
42     if(tr[u].l == tr[u].r) tr[u].sum += v;
43     else{
44         int mid = tr[u].l + tr[u].r >> 1;
45         if(x <= mid) modify(u<<1,x,v);
46         else if(x > mid) modify(u<<1|1,x,v);
47         pushup(u);
48     }
49 }

```

## 三、图论

```

1 总结:
2 不考虑时间性能, 求任意两个点之间的距离--spfa  $O(n^3)$ 
3 正权边( $\geq 0$ ) 朴素dijkstra  $n^2$ 
4 堆优化dijkstra  $m \log n$ 
5 有边数限制的最短路 bellman_ford,last 和 dist数组更新k次
6 优化的spfa算法
7 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
8
9 prim 每次找一个最近的点加入到集合当中去,  $O(n^2)$ 复杂度
10 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
11
12 判断二分图 1、2两种颜色进行染色 看看是否有冲突
13 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

```
1  int g[MAXN]; // 记录状态的数组，可能是多个或者多维的
2  int ans = 初始情况
3  void dfs(根据题意传入合适的搜索参数) {
4      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
5          return; // 或记录解，视情况而定
6      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
7          ans = 从当前解与已有解中选最优; // 或输出解，视情况而定
8      //上面两个可以合成一个
9      //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
10     for (遍历边界内的所有情况)
11         if (解合法) {
12             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
13             dfs(继续下去);
14             撤回操作;
15             //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
16             //如果还想遍历其它情况可以接着dfs 参考七段码
17             dfs(继续下一个操作)
18         }
19         //当前层所有的都不行之后return掉
20 }
```

### 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

```
1  void bfs(参数){ //一般是初始状态
2      //1.定义queue
3      queue<etype> q;
4      //2.定义不同状态的距离、属性
5      int d[];
6      //3.初始化q、d
7      q.push(start);d[start] = 0;
8      //如果有终止状态也可以定义
9      while(q.size())
10     {
11         auto t = q.front();
12         q.pop();
13         if(t == end) return xx;
14         for(int i = 0;i<4;i++)
15         {
16             int x0 = ... y0 = ...
17             if(点越界了 break);
18             if(状态被遍历过了) break; //这个很重要 千万别忘了
19             //对合法的点、且没被遍历过的点 进行更新
20
21             //当前点恢复为原来的状态1
```

```

22         state back();
23     }
24 }
25 }

```

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

```

1 //树的深度遍历模板
2 int dfs(int u)
3 {
4     //当前子树往下遍历的最大深度
5     //保证每个点只被遍历一次
6     st[u] = true;
7     //当前节点剩余所有值 子树当中最大值
8     int sum = 0, res = 0;
9     for(int i = h[u]; i != -1; i = ne[i])
10    {
11        int x = e[i];
12        if(!st[x])
13        {
14            int k = dfs(x);
15            sum += k; //sum是当前节点子节点的和
16            res = max(res, k);
17        }
18    }
19    res = max(res, n - sum - 1);
20    ans = min(res, ans);
21    //sum只是统计了当前所有子节点的长度
22    return sum + 1;
23 }

```

```

1 //bfs模板
2 #include <iostream>
3 #include <cstring>
4 #include <algorithm>
5 #include <queue>
6 using namespace std;
7 const int N = 1e6 + 10;
8 int h[N], e[N], ne[N], idx;
9 int n, m;
10 int dist[N];
11 bool st[N];
12 //有向图
13 void add(int a, int b)
14 {
15     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
16 }
17 int bfs()
18 {
19     memset(dist, 0x3f, sizeof dist);

```

```

20     dist[1] = 0;
21     st[1] = true;
22     queue<int> q;
23     q.push(1);
24     while(q.size())
25     {
26         int t = q.front();
27         q.pop();
28         for(int i = h[t]; i!=-1; i=ne[i])
29         {
30             int j = e[i];
31             if(!st[j]){
32                 dist[j] = dist[t] + 1;
33                 q.push(j);
34                 //如果已经被更新了就标记一下
35                 st[j] = true;
36             }
37         }
38     }
39     if(dist[n] == 0x3f3f3f3f) return -1;
40     else return dist[n];
41 }

```

### 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

```

1 void bfs()
2 {
3     queue<int> q;
4     //注意初始入度为0的点不止一个
5     for(int i = 1; i<=n; i++)
6         if(!ind[i]){
7             q.push(i);
8             st[i] = true;
9         }
10    while(q.size())
11    {
12        int t = q.front();
13        q.pop();
14        out[cnt++] = t;
15        for(int i = h[t]; i!=-1; i=ne[i])
16        {
17            int j = e[i];
18            ind[j]--;
19            //要不能更新过 且入度为0的点
20            if(!ind[j] && !st[j]){
21                q.push(j);
22            }
23        }
24    }
25    if(cnt < n) puts("-1");
26    else{

```

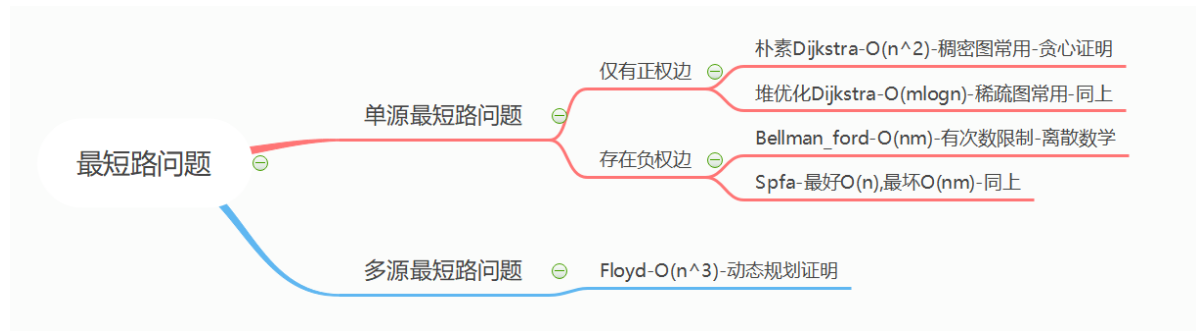
```

27     for(int i = 0;i<cnt;i++)
28         cout<<out[i]<<" ";
29     }
30 }

```

## 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

```

1 //朴素版dijkstra()算法
2 //点来更新点
3 bool dijkstra()
4 {
5     memset(dist,0x3f,sizeof dist);
6     dist[1] = 0;
7     for(int i = 1;i<n;i++)
8     {
9         int t = -1;
10        for(int j = 1;j<=n;j++)
11            if(!st[j] &&(t == -1 || dist[t] > dist[j]))
12                t = j;
13        st[t] = true;
14        for(int i = 1;i<=n;i++)
15        {
16            if(dist[i] > dist[t] + g[t][i])
17                dist[i] = dist[t] + g[t][i];
18        }
19    }
20    if(dist[n] > 0x3f3f3f3f/2) return false;
21    else return true;
22 }

```

同bfs相比较，dijkstra()算法是在边出队的时候才是最小值

```

1 //对比bfs bfs是压入队列的时候就进行标记
2 //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
3 void add(int a,int b,int c)
4 {

```

```

5     e[idx] = b, ne[idx] = h[a];
6     w[idx] = c, h[a] = idx++;
7 }
8 int dijkstra()
9 {
10     priority_queue<PII, vector<PII>, greater<PII> > heap;
11     memset(dist, 0x3f, sizeof dist);
12     dist[1] = 0;
13     heap.push({0, 1}); // dist node
14     // st[1] = true;
15     while(heap.size())
16     {
17         PII t = heap.top();
18         heap.pop();
19         int dis = t.first, x = t.second;
20         // 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
21         if(st[x]) continue;
22         st[x] = true;
23         for(int i = h[x]; i != -1; i = ne[i])
24         {
25             int j = e[i];
26             if(dist[j] > dis + w[i])
27             {
28                 dist[j] = dis + w[i];
29                 heap.push({dist[j], j});
30             }
31         }
32     }
33     if(dist[n] == 0x3f3f3f3f) return -1;
34     else return dist[n];
35 }

```

### 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 $O(m \cdot n^2)$

- 两重循环 第一重循环 边的限制
- 第二重循环 所有边遍历

```

1 //有边数限制的最短路
2 //注意存在负环 即最后距离可以为负数
3 //同时注意最短距离有可能为-1
4 int bellman_ford()
5 {
6     memset(dist, 0x3f, sizeof dist);
7     dist[1] = 0;
8     for(int i = 1; i <= k; i++)
9     {
10         //k条边
11         memcpy(backup, dist, sizeof dist);
12         for(int j = 0; j < m; j++)
13         {

```

```

14         //x -> y 有向边
15         int x = edges[j].a, y = edges[j].b, c = edges[j].w;
16         if(dist[y] > backup[x] + c)
17         {
18             dist[y] = backup[x] + c;
19         }
20     }
21 }
22 //注意存在负环 可以小于0x3f3f3f3f 距离也可以是-1嘛了
23 //if(dist[n] == 0x3f3f3f3f) return -1;
24 //if(dist[n] > 0x3f3f3f3f/2) return -1;
25 // else return dist[n];
26 return dist[n];
27 }

```

### 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

```

1  //是否在队列当中
2  int spfa()
3  {
4      memset(dist, 0x3f, sizeof dist);
5      dist[1] = 0;
6      //表示在队列当中
7      st[1] = true;
8      queue<int> q;
9      q.push(1);
10     while(q.size())
11     {
12         int t = q.front();
13         q.pop();
14         st[t] = false;
15         for(int i = h[t]; i != -1; i = ne[i])
16         {
17             int j = e[i];
18             if(dist[j] > dist[t] + w[i])
19             {
20                 dist[j] = dist[t] + w[i];
21                 if(!st[j]){
22                     q.push(j);
23                     st[j] = true;
24                 }
25             }
26         }
27     }
28     return dist[n];
29 }

```

```

1  //spfa判断负环
2  //
3  bool spfa()
4  {

```

```

5     queue<int> q;
6     //负环可以从任何一点出发
7     for (int i = 1; i <= n; i ++ )
8     {
9         st[i] = true;
10        q.push(i);
11    }
12
13    while (q.size())
14    {
15        int t = q.front();
16        q.pop();
17
18        st[t] = false;
19
20        for (int i = h[t]; i != -1; i = ne[i])
21        {
22            int j = e[i];
23            if (dist[j] > dist[t] + w[i])
24            {
25                dist[j] = dist[t] + w[i];
26                cnt[j] = cnt[t] + 1;
27                //当某个点更新次数为n次以上时候 说明有负环
28                if (cnt[j] >= n) return true;
29                if (!st[j])
30                {
31                    q.push(j);
32                    st[j] = true;
33                }
34            }
35        }
36    }
37
38    return false;
39 }

```

### 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

```

1 void floyd()
2 {
3     for(int i = 1;i<=n;i++) g[i][i] = 0;
4     //注意k要放在最外面 解决填空利器
5     for(int k = 1;k<=n;k++)
6         for(int i = 1;i<=n;i++)
7             for(int j = 1;j<=n;j++)
8             {
9                 g[i][j] = min(g[i][j],g[i][k] + g[k][j]);
10            }
11 }
12

```



## 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。

```
1 bool prim()
2 {
3     memset(dist, 0x3f, sizeof dist);
4     for(int i = 0; i < n; i++)
5     {
6         int t = -1;
7         //挑出一个到集合最短的点
8         for(int j = 1; j <= n; j++)
9             if(!st[j] && (t == -1 || dist[t] > dist[j]))
10                t = j;
11        //到集合当中距离最近的点是无穷 不存在满足条件的点
12        if(i && dist[t] > 0x3f3f3f3f / 2) return false;
13        //除了第一个点剩余点才能算距离
14        if(i) ans += dist[t];
15        st[t] = true;
16        //更新该点连接点
17        for(int j = 1; j <= n; j++)
18            //这里更新距离就相当于求该点到集和之间的距离
19            dist[j] = min(dist[j], g[t][j]);
20    }
21    return true;
22 }
```

### 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  $O(m \log m)$

- 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

```
1 int find(int x)
2 {
3     if(f[x] != x) f[x] = find(f[x]);
4     return f[x];
5 }
6 bool kruskal()
```

```

7  {
8      for(int i = 1;i<=m;i++)
9      {
10         int a = edges[i].a,b = edges[i].b,w = edges[i].w;
11         int fa = find(a),fb = find(b);
12         //fa == fb代表两个点已经在同一个连通块当中了 这个时候如果再连就成环了
13         if(fa == fb) continue;
14         else{
15             //加入
16             f[fa] = fb;
17             ans += w;
18             cnt++;
19         }
20     }
21     //注意结束的条件是边数达到了n-1
22     if(cnt == n-1) return true;
23     else return false;
24 }

```

## 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

```

1  bool solve(int node,int c)
2  {
3      color[node] = c;
4      //染它周围的点
5      for(int i = h[node];i!=-1;i=ne[i])
6      {
7          int j = e[i];
8          if(!color[j]){
9              bool t = solve(j,3-c);
10             if(!t) return false;
11         }else{
12             if(color[j] == c) return false;
13         }
14     }
15     return true;
16 }
17 //处理不连通的情况
18 for(int i = 1;i<=m;i++)
19 {
20     if(!color[i]){
21         if(!solve(i,1)){
22             flag = false;
23             break;
24         }
25     }
26 }

```

## 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

```
1  bool find(int x)//为x找一个对象
2  {
3      for(int i = h[x]; i != -1; i = ne[i])
4      {
5          int j = e[i];
6          if(!st[j])
7          {
8              //如果没有这个st 很可能一直递归下去
9              st[j] = true;
10             if(!match[j] || find(match[j]))
11             {
12                 //找到了该点就将match设置一下
13                 match[j] = x;
14                 return true;
15             }
16         }
17     }
18     //所有爱慕的都不行 返回false
19     return false;
20 }
21 //为每个左边的点找一个匹配
22 for(int i = 1; i <= n1; i++)
23 {
24     memset(st, false, sizeof st);
25     if(find(i)) ans++;
26 }
```

## 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：

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

$N$  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：

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

## 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

```
1 //用于筛出来 2-n当中的质数
2 for(int i = 2;i<=n;i++)
3 {
4     if(v[i] == 0) v[i] = i,prime[++cnt] = i;
5     //给当前数i乘上一个质因子
6     for(int j = 1;j<=cnt;j++)//数组是从1开始用的
7     {
8         //利用最小的 当大于v[i]说明不是最小的
9         //或者当primes[j]*i超过n的范围的时候 退出循环
10        //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
11        if(prime[j] > i || prime[j] > n/i) break;
12        //prime[j]是最小质因子
13        v[i*prime[j]] = prime[j];
14    }
15 }
```

### 1.2 求某个具体数的质因子 $O(\sqrt{n})$

```
1
2 void solve(int n)
3 {
4     //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
5     for(int i =2;i<=n/i;i++)
6     {
7         int k = 0;
8         if(n % i == 0)
9         {
10            cout<<i<<" ";
11            while(n % i == 0)
12            {
13                n /= i;
14                //质因数的多少次幂
15                k++;
16            }
17            cout<<k<<endl;
18        }
19    }
20    //还剩下一个 必然是素数
21    //注意这里的n > 1千万别忘了
22    if(n > 1) cout<<n<<" "<<1<<endl;
23 }
```

### 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：

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

记住公式(pi-1)/pi就行了，别忘记那个n

约数

```
1 //辗转相除法 比较少见
2 // gcd(a,b) == gcd(b,a) == gcd(a-b,b) == gcd(b,a-b)
3 LL gcd_sub(LL a,LL b)
4 {
5     if(a < b) swap(a,b);
6     if(b == 1) return a;
7     else return gcd_sub(b,a/b);
8 }
```

## 欧几里得算法

```
1 int gcd(int a,int b)
2 {
3     if(!b) return a;
4     return gcd(b,a%b);
5 }
```

## 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 $(\alpha_1+1)*(\alpha_2+1)*\dots$ .. $\alpha$ 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

```
1 void get_primes(int n)
2 {
3     phi[1] = 1;
4     for(int i = 2;i<=n;i++)
5     {
6         if(v[i] == 0) v[i] = i,primes[cnt++] = i,phi[i] = i-1;
7         for(int j = 0;j<cnt;j++)
8         {
9             if(primes[j] > v[i] || primes[j] > n/i) break;
10            v[primes[j]*i] = primes[j];
11            if(i % primes[j] == 0) phi[i*primes[j]] = phi[i] * primes[j];
12            else phi[i*primes[j]] = phi[i]*(primes[j] - 1);
13        }
14    }
15    LL res = 0;
16    for(int i = 1;i<=n;i++)
17    {
18        res += phi[i];
19    }
20    cout<<res<<endl;
21 }
```

## 快速幂

### 1.快速幂

```
1 void quick_mi(int a,int k,int p){
2     ll res = 1;
3     while(k){
4         if(k&1) res = res*a % p;
5         //k左移一位
6         k >>= 1;
7         //注意a在迭代的时候也会爆int
8         //a平方等于下一个数
9         a = a*(1ll)a % p;
10    }
11    cout<<res<<endl;
12 }
```

### 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 $b^{(p-2)} \% p$ ;

$$(x*a = x/b \pmod p)$$

```
1 //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
2 //当且仅当p为质数 且gcd(p,b) = 1时 a为b^p
3 int gcd(int a,int b)
```

```

4  {
5      if(!b) return a;
6      return gcd(b,a%b);
7  }
8  int quick_mi(int a,int b,int p)
9  {
10     LL res = 1;
11     while(b)
12     {
13         if(b & 1) res = (LL)res*a % p;
14         b >>= 1;
15         a = (LL)a*a % p;
16     }
17     return res;
18 }
19 int main()
20 {
21     int n;cin>>n;
22     int a,b;
23     while(n-->0)
24     {
25         cin>>a>>b;
26         if(gcd(a,b) != 1) puts("impossible");
27         else{
28             cout<<quick_mi(a,b-2,b)<<endl;
29         }
30     }
31     return 0;
32 }

```

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

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

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

# 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

Handwritten derivation of the extended Euclidean algorithm:

$$\begin{aligned} ax + by &= d \\ a \% b &= a - \lfloor \frac{a}{b} \rfloor \cdot b \\ by + a \% b x &= d \\ by + (a - \lfloor \frac{a}{b} \rfloor b) x &= d \\ ax + b(y - \lfloor \frac{a}{b} \rfloor x) &= d \\ \text{从而 } y &= y - \frac{a}{b} \cdot x \end{aligned}$$

```
1 //加了&x y的值才可以传回去
2 //返回值仍然是a和b的最大公约数
3 int exgcd(int a,int b,int &x,int &y)
4 {
5     if(!b){
6         x = 1,y = 0;
7         return a;
8     }
9     //by + a%b x = d
10    int d = exgcd(b,a%b,y,x);
11    //顺序颠倒了 推一下y是变了的 并且变了的y可以传回去
12    y = y - a/b*x;
13    return d;
14 }
```

## 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  $a \cdot x = m \cdot y + b$  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

```
1 void init(){
2     f[0][0] = 1;
3     for(int i = 1;i<=2000;i++)
4         //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
5         for(int j = 0;j<=i;j++)
6             {
7                 f[i][j] = (f[i-1][j-1]%mod + f[i-1][j]%mod) % mod; //第i个选了 没选
8             }
9 }
```

### 2.当a, b达到1e^5

```
1     fact[0] = 1;infact[0] = 1;
2     for(int i = 1;i<N;i++)
3     {
4         fact[i] = (LL)fact[i-1] * i%mod;
5         infact[i] = (LL)infact[i-1]*quick_mi(i,mod-2,mod)%mod;
6     }
7     int a,b;
8     while(n--)
9     {
10        scanf("%d %d",&a,&b);
11        cout<<fact[a]*(LL)infact[a-b]%mod*infact[b]%mod<<endl;
12    }
```

### 3.当a,b达到1e^18

lucas定理

▷

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

```
1 int quick_mi(int a,int b,int p)
2 {
3     int res = 1;
4     while(b)
5     {
6         if(b&1) res = (LL)res*a%p;
7         b >>= 1;
8         a = a*(LL)a % p;
9     }
10    return res;
11 }
12 int C(int a,int b)
13 {
```

```

14     int res = 1;
15     //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
16     for(int i = 1,j = a;i<=b;i++,j--)
17     {
18         res = (LL) res*j%p;
19         res = (LL) res*quick_mi(i,p-2,p) % p;
20     }
21     return res;
22 }
23 //直接算出来C(a,b)的结果
24 int lucas(LL a,LL b)
25 {
26     if(a < p && b < p) return C(a,b);
27     return (LL)C(a%p,b%p)*lucas(a/p,b/p)%p;
28 }

```