基础

快速排序

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l +(r-l)/2];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

归并排序

```
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l +(r-l)/2;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;//[l,mid] 和 [mid+1,r]
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

二分

二分的精髓在于有一个性质，在一个区间上连续保持

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
int bsearch_1(int l, int r)//寻找mid
{
    while (l < r)
    {
        int mid = l +(r-l)/2;//mid+1<=r
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)//寻找mid
{
    while (l < r)
    {
        int mid = l +(r-l+1)/2;//mid>=l+1
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
```

```
}

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

高精度

```
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }

    if (t) C.push_back(t);
    return C;
}

// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    int t=0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();//去除前导0
    return C;
}
// C = A * b, A >= 0, b >= 0
vector<int> mul(vector<int> &A, int b)
{
    vector<int> C;

    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

```
//A>=0,b>0; A/b=C...r
vector<int> div(vector<int> &A, int b, int &r)//r是余数
{
    vertor<int> C;//商
    r=0;
    for(int i=A.size()-1;i>=0;i--)//从高位开始做除法
    {
        r=r*10+A[i];
        C.push_back(r/b);//注意是b
        r%=b;
    }
    reverse(C.begin(),C.end());//因为和高精度存数不一样 index大的 正常书写情况下在前面
    while(C.size() > 1 && C.back() == 0)C.pop_back();//去除前导0
    return C;
}
```

前缀和

```
//一维前缀和
S[i]=a[1]+a[2]+...+a[i]//下标要从1开始!! S[0]=0
a[l]+a[l+1]+...+a[r]=S[r]-S[l-1] //区间和
//二维前缀和
//S[i, j] = 第i行j列格子左上部分所有元素的和 下标都从1开始
//以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为：
//S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
S[x2][y2]-S[x1-1][y2]-S[x2][y1-1]+S[x1-1][y1-1]//求子矩阵的和
S[i][j]=S[i-1][j]+S[i][j-1]-S[i-1][j-1]+a[i][j];//求前缀和
```

差分

```
//一维差分 区间加上固定值
a[i]=b[1]+b[2]+...+b[i]
//给区间[l, r]中的每个数加上c：B[l] += c, B[r + 1] -= c
//二维差分
//给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：
//b[x1, y1] += c, b[x2 + 1, y1] -= c, b[x1, y2 + 1] -= c, b[x2 + 1, y2 + 1] += c
```

双指针（非常多）

```
//指向一个序列
for(i=0,j=0;i<n;i++)
{
    while(i<j && check(i,j) )i++;
    //具体逻辑
}
//指向两个序列


//核心思想
//把O(n^2)优化成O(n)
//下面是暴力做法
for(int i =0;i<n;i++)
{
  for(int j=0;j<=i;j++)
  {

  }
}
```

位运算

```
//求第k位 n>>k&1
//n的最后1位  1010 10 lowbit(n) = n & -n
```

整数的离散化(有序)

```
vector<int> alls;//所有待存储的离散化值
sort(alls.begin(),alls.end());//将所有值排序
alls.erase(unique(alls.begin(),alls.end()),alls.end());//去重
//unique(all.begin(),alls.end()) 返回多余代码的位置
//二分求出x对应的离散化值
int find(int x)//找到>=x的右分界点
{
    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;
    }
    return l+1;//或者l 为了映射到1.2.3...n
}
```

区间合并

```
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;//st维护区间左端点 ed维护区间的右端点
    for (auto seg : segs)
        if (ed < seg.first)//如果没有交集
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);//有交集

    if (st != -2e9) res.push_back({st, ed});//防止是空的

    segs = res;
}
//leetcode solution
  vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.size() == 0) {
            return {};
        }
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> merged;
        for (int i = 0; i < intervals.size(); ++i) {
            int L = intervals[i][0], R = intervals[i][1];
            if (!merged.size() || merged.back()[1] < L) {
                merged.push_back({L, R});
            }
            else {
                merged.back()[1] = max(merged.back()[1], R);
            }
        }
        return merged;
    }
```

数据结构

单链表

```
// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
void init()
{
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void insert(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}

// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
}
//下面的仅供参考
void delete_node(int k)//删除idx为k的后面那个数
{
    if(k==0)
    {
        head=ne[head];
    }
    else
    {
        ne[k]=ne[ne[k]];
    }
}

void insert_node(int k,int x)//在idx为k的后面插入一个数
{
    e[idx]=x;
    ne[idx]=ne[k];
    ne[k]=idx;
    idx++;
}
```

双链表

```
// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点
int e[N], l[N], r[N], idx;

// 初始化
void init()
{
    //0是左端点，1是右端点  0 1
    r[0] = 1, l[1] = 0;
    idx = 2;
}

// 在节点a的右边插入一个数x  在右边插入就是在a左边那个节点的右边插入
void insert(int a, int x)
{
    e[idx] = x;//先赋值
    l[idx] = a, r[idx] = r[a];//idx的值弄好
    l[r[a]] = idx, r[a] = idx ++ ;
}

// 删除节点a
void remove(int a)
{
    l[r[a]] = l[a];
```

```
    r[l[a]] = r[a];
}
```

栈

```
int stk[N], tt = 0;
// 向栈顶插入一个数
stk[ ++ tt] = x;
// 从栈顶弹出一个数
tt -- ;
// 栈顶的值
stk[tt];
// 判断栈是否为空
if (tt > 0)
{
}
stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素
```

队列

```
queue, 队列
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素
```

单调队列(窗口极值)

做题思路

1. 队头是否需要更新
2. 将队列中没用的元素删掉-->队列单调了 如果是求最大值 队列单减，如果求最小值 队列单增
3. O(1)取出队头 队尾

```
deque<int> Q; // 存储的是编号 k是区间长度 V中存放着原序列
for (int i = 0; i < n; i++)
{
    if (!Q.empty() && Q.front() < i-k+1 ) Q.pop_front();//毕业了 出了区间 队列头值 小于区间左值 i-k+1 区间左值
    while (!Q.empty() && V[Q.back()] < V[i])Q.pop_back();// 比新生弱的当场退役（求区间最小值把这里改成>即可）
    Q.push_back(i); // 新生入队
    if (i-k+1 >=0 )cout << V[Q.front()] << " ";//如果有输出的需求 了
}
```

单调栈(NGE)

next greater element

1. 新值取代旧值
2. 给答案
3. 入栈

```
stack<int> S;
//求右侧
for (int i = 1; i <= n; ++i)
{
    while (!S.empty() && V[S.top()] < V[i])//求NGE
    {
```

```
        ans[S.top()] = i;//i就是栈顶的答案
        S.pop();//pop确保栈顶的答案只会赋值一次
    }
    S.push(i);
}
//求左侧
for (int i = 0; i < n; i ++ )
{
    while(!S.empty() && V[i]<=V[S.top()])S.pop();
    if(S.empty())cout<<-1<<" ";
    else cout << V[S.top()]<<" ";
    S.push(i);
}
```

Trie(存查字符串 最大异或值)

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点 26的原因是 一个节点最多只有26个分支
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(string &str)
{
    int p =0;//parent
    for(auto i:str)
    {
        int u=i-'a';
        if(!son[p][u])son[p][u]=++idx;
        p=son[p][u];
    }
    cnt[p]++;
}
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(string &str)
{
    int p=0;
    for(auto i:str)
    {
        int u=i-'a';
        if(!son[p][u])return 0;
        p=son[p][u];
    }
    return cnt[p];
}
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

KMP(字符串匹配,求循环节)

```
//求pmt表
// pmt[0] = 0;单个字符的前缀集合为空
for (int i = 1, j = 0; i < p.length(); i++)
{//i是pmt表遍历时的下标
    while (j && p[i] != p[j]) j = pmt[j - 1];
    if (p[i] == p[j]) j++;
    pmt[i] = j;
}

for (int i = 0, j = 0; i < s.length(); i++)
{//第i位待匹配
    while (j && s[i] != p[j]) j = pmt[j - 1]; // 不断前移j指针，直到成功匹配或移到头为止
    if (s[i] == p[j]) j++;  // 当前位匹配成功，j指针右移
    if (j == p.length())//找到了匹配
    {
        // 对s[i - j + 1 .. i]进行一些操作
        j = pmt[j - 1];
    }
}
```

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度  字符产的下标均从1 开始
//ne[j] 在p[]中 以p[j]位结尾的后缀和 从1开始的非平凡前缀的最大长度
求模式串的Next数组：
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}

// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

AC自动机

trie+kmp

```
// KMP 这颗trie树为一条链
for (int i = 2; i <= m; i ++ )
{
    int j = ne[i - 1]; // 见第6行，新循环i++了，所以j = ne[i - 1]
    while (j && p[i] != p[j + 1]) j = ne[j]; // 看j的儿子节点中是否有值等于p[i]的节点
    if (p[i] == p[j + 1]) j ++ ; // 走到节点j儿子节点上去
    ne[i] = j; // 更新树中节点i的next值
}
```

```
// AC自动机与KMP一一对应
while(hh <= tt)
{
    int t = q[hh ++ ];
    for(int i = 0 ; i < 26 ; i ++ ) // 枚举t的所有儿子节点的值 KMP只有与一个儿子节点无须枚举
    {
        c = tr[t][i]; // t的儿子节点中值为i的节点的编号
        j = ne[t]; // 上一个字母的前缀到哪了
        while(j && !tr[j][i]) j = ne[j];
```

```
        if(tr[j][i]) j = tr[j][i];
        ne[c] = j;
        q[++ tt] = c; // 拓展出来了的点就放入队列
    }
}
```

trie图

```
const int N=10010,S=55;//N 表示有多少单词 S表示每个单词最多多少字母
int tr[N*S][26],cnt[N*s],idx;
char str[M];//待搜索的文章
queue<int> q,ne[N*s];

// 插入一个字符串
void insert(string &str)
{
    int p =0;//parent
    for(auto i:str)
    {
        int u=i-'a';
        if(!tr[p][u])tr[p][u]=++idx;
        p=tr[p][u];
    }
    cnt[p]++;
}

void build()
{
    for(int i=0;i<26;i++){
    if(tr[0][i])q.push(tr[0][i]);//最上面的两层因为肯定指向根节点
    }
    while(!q.empty()){
        int t=q.front(),q.pop();
        for(int i=0;i<26;i++){
            int p=tr[t][i];
            if(!p)tr[t][i]=tr[ne[t]][i];//如果当前节点的i这个儿子不存在  那么我们在匹配的时候肯定是要访问ne[t][i]节点的，所以直接做了这个优化
            else{
                ne[p]=tr[ne[t]][i];
                q.push(p);
            }
        }

    }
}
int main()
{
    int T;
    scanf("%d", &T);
    while (T -- )
    {
        memset(tr, 0, sizeof tr);
        memset(cnt, 0, sizeof cnt);
        memset(ne, 0, sizeof ne);
        idx = 0;

        scanf("%d", &n);
        for (int i = 0; i < n; i ++ )
        {
            scanf("%s", str);
            insert();
        }

        build();

        scanf("%s", str);

        int res = 0;
        for (int i = 0, j = 0; str[i]; i ++ )
```

```
        {
            int t = str[i] - 'a';
            j = tr[j][t];

            int p = j;
            while (p)
            {
                res += cnt[p];
                cnt[p] = 0;
                p = ne[p];
            }
        }

        printf("%d\n", res);
    }

    return 0;
}
```

并查集

```
//初始化
int fa[MAXN];
void init(int n)
{
    for (int i = 1; i <= n; ++i)
    {
        fa[i] = i;
        rank[i] = 1;
    }
}
//查询
int find(int x)
{
    return x == fa[x] ? x : (fa[x] = find(fa[x]));//！！！
}
//合并
void merge(int i, int j)
{
    fa[find(i)] = find(j);
}
//按秩合并
void merge(int i, int j)
{
    int x = find(i), y = find(j);    //先找到两个根节点
    if (rank[x] <= rank[y])
        fa[x] = y;
    else
        fa[y] = x;

    if (rank[x] == rank[y] && x != y)
        rank[y]++;                   //如果深度相同且根节点不同，则新的根节点的深度+1
}


(2)维护size的并查集：

    int fa[N], size[N];
    //fa[]存储每个点的祖宗节点，size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

    // 返回x的祖宗节点
    int find(int x)
    {
        if (fa[x] != x) fa[x] = find(fa[x]);
        return fa[x];
    }

    // 初始化，假定节点编号是1~n
```

```
    for (int i = 1; i <= n; i ++ )
    {
        fa[i] = i;
        size[i] = 1;
    }

    // 合并a和b所在的两个集合：  上下a bx
    //注意到如果a  b在同一个节点
    if(find(b)==find(a))return;
    size[find(b)] += size[find(a)];//b祖宗的size增加了 !!!
    fa[find(a)] = find(b);//a的祖宗现在是b!!!
```

(3)维护到祖宗节点距离的并查集：

```
    int fa[N], d[N];
    //fa[]存储每个点的祖宗节点, d[x]存储x到fa[x]的距离

    // 返回x的祖宗节点
    int find(int x)
    {
        if (fa[x] != x)
        {
            int u = find(fa[x]);//在这里指向完成之后才会有合适的d[fa[x]]
            d[x] += d[fa[x]];
            fa[x] = u;
        }
        return fa[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        fa[i] = i;
        d[i] = 0;
    }

    // 合并a和b所在的两个集合：
    fa[find(a)] = find(b);
    d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

(4)拓展域

堆

1. 插入一个数
2. 求集合当中的最小值(最值)
3. 删除最小值(最值)
4. 删除任意元素(stl无法直接实现)
5. 修改任意元素(stl无法直接实现)

```
// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1 h heap
// ph[k]存储第k个插入的点在堆中的位置 point order -> heap order
// hp[k]存储堆中下标是k的点  是第几个插入的  求点序号   heap order -> point order
int h[N], ph[N], hp[N], size;
//小根堆
//交换两个点，及其映射关系
void heap_swap(int a, int b)
{
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t])  t = u * 2;
```

```
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        heap_swap(u, u / 2);
        u /= 2;
    }
}

// O(n)建堆
for (int i = n / 2; i; i -- ) down(i);
1. 插入一个数 heap[++size]=x;up(size);
2. 求集合当中的最小值(最值) heap[1]
3. 删除最小值(最值) heap[1]=heap[size];size--;down(1);
4. 删除任意元素 heap[k]=heap[size];size--;do wn(k);up(k);
5. 修改任意元素 heap[k]=x;down(k);up(k);
```

hash

1. 添加一个数
2. 查找一个数

```
(1) 拉链法
    int h[N], e[N], ne[N], idx;/ /h[k]是 k对应的值在e中的下标

    // 向哈希表中插入一个数 N取质数 离2的幂次尽可能远
    void insert(int x)
    {
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx ++ ;
    }

    // 在哈希表中查询某个数是否存在
    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;
    }

(2) 开放寻址法
    int h[N];//N开到数据范围的2-3倍 需要让它不能都满

    // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置
    int find(int x)
    {
        int k = (x % N + N) % N;
        while (h[k] != null && h[k] != x)//没找到并且没人
        {
            k ++ ;
            if (k == N) k = 0;
        }
        return k;
    }

字符串hash  快速判断两个字符串是否相等
```

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[0]=0,h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64 字母不能映射为0

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];//str[i]最好不等于0
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];//p[r-l+1]是p^(r-l+1)的同余类
}
```

STL

```
    size() 复杂度为O(1)
    empty() 复杂度是O(1)


vector, 变长数组，倍增的思想   分配空间 所需时间只与申请次数有关
    size()  返回元素个数   申请次数是O(logn) copy的效率是O(1)(均摊之后)
    empty()  返回是否为空
    clear()  清空
    front()/back()
    push_back()/pop_back()
    begin()/end() end()指向的是最后一个数后面的一个数
    []
    支持比较运算，按字典序
    vector<int> a(10,3);

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）
    p={20,"2"};
    p=make_pair(20,"2");

string，字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标，(子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址
    a+="c";

queue, 队列
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素

priority_queue, 优先队列，默认是大根堆
    size()
    empty()
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式：priority_queue<int, vector<int>, greater<int>> q;

stack, 栈
```

```
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
    size()
    empty()
    clear()
    begin()/end()
    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset
        insert()  插入一个数
        find()   查找一个数 找不到返回end()
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数x，删除所有x   O(k + logn)
            (2) 输入一个迭代器，删除这个迭代器
        lower_bound()/upper_bound()
            lower_bound(x)  返回大于等于x的最小的数的迭代器
            upper_bound(x)  返回大于x的最小的数的迭代器
    map/multimap
        insert()  插入的数是一个pair
        erase()  输入的参数是pair或者迭代器
        find()
        []  注意multimap不支持此操作。 时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
    和上面类似，增删改查的时间复杂度是 O(1)
    不支持 lower_bound()/upper_bound(), 迭代器的++, --
    unordered_set<int> S;
    S.insert();
    S.count(i);

bitset, 压位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个1

    any()  判断是否至少有一个1
    none()  判断是否全为0

    set()  把所有位置成1
    set(k, v)  将第k位变成v
    reset()  把所有位变成0
    flip()  等价于~
    flip(k) 把第k位取反
```

搜索

树与图的存储

1. 邻接矩阵:g [a] [b] 存储边a->b
2. stl::vector

```
//如果没有边权可以不使用结构体，只存储终点即可
struct Edge
{
    int to, w;
};
std::vector<Edge> edges[MAXN];
inline void add(int from, int to, int w)
{
    Edge e = {to, w};
    edges[from].push_back(e);  //向vector的最后添加一条边
}
```

无向图调用两次

```
inline void add2(int u, int v, int w)
{
    add(u, v, w);
    add(v, u, w);
}
```

3. 邻接表:

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;

// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

树与图的遍历

可行性剪枝 分支不可行 和 最优性剪枝 分支不可能是最优的

搜索的时间复杂度 O(n+m), n 表示点数，m 表示边数

深度优先遍历 空间O(h) 不具有最短性 考虑搜索的顺序（不遗漏）

宽度优先遍历 空间O(2^h) 具有最短性

1. 深度优先遍历 空间要求高 思路奇怪 **搜索顺序要清楚** 树的重心

```
int dfs(int u)
{
    //查看是否到达叶子节点


    for (int i = h[u]; i != -1; i = ne[i])
    {//遍历可能

        int j = e[i];
        if (!st[j])
            st[j] = true,dfs(j),st[j]=false; // st[u] 表示点u已经被遍历过;
```

```
    }
    //恢复现场
}
```

2. 宽度优先遍历

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
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])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

拓扑排序

边都是从前往后 边的起点终点出现的顺序都是先起点后终点

时间复杂度O(n+m) n表示点数，m表示边数

```
// deg是入度，在存图的时候需要录入数据
// A是排序后的数组
/*
//如果没有边权可以不使用结构体，只存储终点即可
struct Edge
{
    int to, w;
};
std::vector<Edge> edges[MAXN];
inline void add(int from, int to, int w)
{
    Edge e = {to, w};
    edges[from].push_back(e);  //向vector的最后添加一条边
}
*/
int deg[MAXN], A[MAXN];
bool toposort(int n)
{
    int cnt = 0;
    queue<int> q;
    for (int i = 1; i <= n; ++i)
        if (deg[i] == 0)
            q.push(i);
    while (!q.empty())
    {
        int t = q.front();
        q.pop();
        A[cnt++] = t;
        for (auto to : edges[t])//遍历t节点的可达的点
        {
            deg[to]--;
            if (deg[to] == 0) // 出现了新的入度为0的点
                q.push(to);
        }
    }
    return cnt == n;
}
```

图论

最短路

难点在于建图 将实际问题抽象为最短路问题

dijkstra

单源最短路 所有边权都是正值

朴素版本 **适合稠密图**

时间复杂是 O(n^2+m), n 表示点数，m 表示边数

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);//设置无穷大
    dist[1] = 0;//太鬼畜了 一开始写成判断等于号了

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}

```

堆优化版本 **适合稀疏图**

时间复杂度 O(mlogm), n 表示点数，m 表示边数 优先队列中有一些冗余的节点

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

```

```
        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Bellman-Ford

单源最短路 **存在负权边**

时间复杂度 O(nm), n 表示点数，m 表示边数　**被更新不代表能到达**

```
int n, m;       // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离

struct Edge     // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
}edges[M];

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for (int i = 1; i <= n; i ++ )
    {
        for (int j = 0; j < m; j ++ )
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

spfa （队列优化的Bellman-Ford算法）

没有负环的情况下

时间复杂度 平均情况下 O(m)，最坏情况下 O(nm), n 表示点数，m 表示边数

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
int spfa()
```

```
{
    memset(dist, 0x3f, sizeof dist);//设置无穷大
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

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

        st[t] = false;//出队设置false

        for (int i = h[t]; i != -1; i = ne[i])//遍历t的出边
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入  存入更新后的节点
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa 判负权 环

时间复杂度是 O(nm), n 表示点数，m 表示边数

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数(经过的边数)
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。

    queue<int> q;
    for (int i = 1; i <= n; i ++ )//把所有点都加入队列中
    {
        q.push(i);
        st[i] = true;
    }

    while (q.size())
    {
        auto 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;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

floyd

时间复杂度是 O(n^3), n 表示点数 **没有负权环**

可以找最小环 在最外层循环中 寻找这样的环 环中最大节点为k 枚举i j 小于k

```
初始化：
    constexpr INF=0x3f3f3f3f;
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )//只经过前k个点的最短路
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

最小生成树

prim

稠密图 朴素版prim算法

时间复杂度是 O(n2+m), n 表示点数，m 表示边数

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边  ！！注意初始化g为无穷大
int dist[N];        // 存储其他点到当前最小生成树的距离 ！！注意初始化dist为无穷大
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f)，否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);//实际上我们可以把任意一个节点最为最小生成树的第一个节点

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )//找到集合外的最小节点
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;//如果找到的最小节点是不可达的

        if (i) res += dist[t];//第1个节点不算
        st[t] = true;

```

```
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);//t节点松弛每一个节点
    }

    return res;
}
```

kruskal

稀疏图 不需要存图

时间复杂度是 O(mlogm)， n 表示点数，m 表示边数

边按照权从小到大排序

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    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;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

二分图

染色法

一个图是二分图 当且仅当 图中不含奇数环 （奇数环 环的边数是奇数） 反证 与 构造

时间复杂度是 O(n+m), n 表示点数，m 表示边数 判别是否为二分图

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色
```

```

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

匈牙利算法

O(mn) 实际运行时间远小于此 最大匹配

最大匹配数 等于 **最小点覆盖数**

```
int M, N;            //M, N分别表示左、右侧集合的元素数量
int Map[MAXM][MAXN]; //邻接矩阵存图
int match[MAXN];          //记录当前右侧元素所对应的左侧元素
bool vis[MAXN];      //记录右侧元素是否已被访问过
bool find(int i)
{
    for (int j = 1; j <= N; ++j)
        if (Map[i][j] && !vis[j]) //有边且未访问
        {
            vis[j] = true;                  //记录状态为访问过
            if (match[j] == 0 || find(match[j])) //如果暂无匹配，或者原来匹配的左侧元素可以找到新的匹配
            {
                match[j] = i;    //当前左侧元素成为当前右侧元素的新匹配
                return true; //返回匹配成功
            }
        }
    return false; //循环结束，仍未找到匹配，返回匹配失败
}
int hungarian()
{
    int cnt = 0;
    for (int i = 1; i <= M; ++i)
    {
        memset(vis, 0, sizeof(vis)); //重置vis数组
        if (find(i))
            cnt++;
    }
    return cnt;
}
// 邻接表
int n1, n2;      // n1表示第一个集合中的点数，n2表示第二个集合中的点数
```

```cpp
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
    if (find(i)) res ++ ;
}
```

数学

数论 算时间复杂度

组合计数

高斯消元

简单博弈论

数论

试除法判质数

```cpp
bool is_prime(int x)  //O(sqrt(n))
{
    if (x < 2) return false;//质数 和 合数 针对从2开始的数
    for (int i = 2; i <= x / i; i ++ )//枚举每一对约数的较小值
        if (x % i == 0)
            return false;
    return true;
}
```

分解质因数

```
void divide(int x)//O(sqrt(n))
{
    for (int i = 2; i <= x / i; i ++ )//枚举的虽然是所有数 但是只有质数符合if判断
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;//
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;//n中最多包含一个大于sqrt(n)的质因子
    cout << endl;
}
```

筛质数

1-n中有n/ln(n)个质数 O(nloglogn)

```
//朴素筛法 O(nlogn) 在1e6时 两者速度差不多
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (st[i]) continue;//只筛质数的倍数
        primes[cnt ++ ] = i;
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}
//线性筛法  n只会被最小质因子筛掉
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;//primes[j]也是primes[j] * i的最小质因子
            if (i % primes[j] == 0) break;//primes[j]一定是i的最小质因子 同时primes[j]也是primes[j]*i的最小质因子
            /*
            当primes[j] | i时  primes[j+1]*i 就应当被 primes[j]筛掉（一个合数会被最小质因子筛掉s）
            */
        }
    }
}
```

约数

```
//求所有约数 O(sqrt(n))
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )//求约数
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);//一对约数
        }
    sort(res.begin(), res.end());//一个数的约数个数 平均是logn
    return res;
}
//约数个数和约数之和  组合
```

```
如果 N = p1^c1 * p2^c2 * ... *pk^ck  算术基本定理
约数个数：  (c1 + 1) * (c2 + 1) * ... * (ck + 1)  乘法原理
约数之和：  (p1^0 + p1^1 + ... + p1^c1) * ... * (pk^0 + pk^1 + ... + pk^ck)  展开即可
求p1^0 + p1^1 + ... + p1^c1
sum(p, k)表示p1^0 + p1^1 + ... + p1^(k-1)
//p0 + .. + pk-1
int sum(int p, int k) {
    if(k == 1) return 1;  //边界
    if(k % 2 == 0) {
        return (LL)(qmid(p, k / 2) + 1) * sum(p, k / 2) % mod;//qmid  快速幂
    }
    return (qmid(p, k - 1) + sum(p, k - 1)) % mod;
}

```

欧几里得

```
int gcd(int a, int b)//O(logN)
{
    return b ? gcd(b, a % b) : a;
}
__gcd() 默认函数
```

欧拉函数

1-x中与x互质的个数

x=p1^c1 * p2^c2 * ... * pk^ck

公式一

phi(x)=N(1-1/p1)(1-1/p2)...(1-1/pk)

思路二

1. 从1-N中去掉p1,p2, pk的所有倍数
2. 加上所有pi*pj的倍数
3. 减去pi * pj *pk的倍数

思路二和公式一是一致的

欧拉定理

a n互质 则

$$a^{\phi(n)}同余1(modn).$$

证明 a1 a2 a phi n 与 aa1 aa2 a aphi n是一组数 两边连乘相等

$$a^{p-1}同余1(modp)$$

```
//x的欧拉函数
int phi(int x)//O(sqrt(n))
{
    int res = x;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res = res / i * (i - 1);// *（1-1/p)
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}
//筛法求欧拉函数 求每一个数的欧拉函数 O(n)
int primes[N], cnt;     // primes[]存储所有素数
int euler[N];           // 存储每个数的欧拉函数
```

```cpp
bool st[N];         // st[x]存储x是否被筛掉


void get_eulers(int n)
{
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;//phi(i)=i-1
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)//pj是i的最小质因子  phi(pj*i) = phi(i)*pj
            {
                euler[t] = euler[i] * primes[j];
                break;
            }
            euler[t] = euler[i] * (primes[j] - 1);//phi(pj*i)= phi(i)*pj*(1-1/pj)
        }
    }
}
```

快速幂

b|a　**b m 互质** b才有逆元 若m是质数b^{m-2}是b模m意义下的逆元　**b如果是m的倍数 一定无解**

$$\frac{a}{b}\text{同余}a \times x(mod m)$$

```cpp
求 m^k mod p，时间复杂度 O(logk)。

int qmi(int m, int k, int p)
{
    LL res = 1 % p, t = m;
    while (k)
    {
        if (k&1) res = res * t % p;//如果k是奇数
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

扩展欧几里得算法

显然 ax+by =d 的 gcd(a,b)的倍数 a b 不全为0

求解线性同余方程

```cpp
// 求x, y，使得ax + by = gcd(a, b)  同时也是 ax+by的最小正整数
int exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1; y = 0;//a+0 = gcd(a,0)
        return a;
    }
    int d = exgcd(b, a % b, y, x);//(a%b)x+by=gcd(a,b)
    y -= (a/b) * x;//ax + b(y-a/b * x)= gcd(a,b)
    return d;
}
```

中国剩余定理

求同余方程的解

$$m_1, m_2, \dots m_k两两互质$$
$$x同余a_1(mod m_1)$$
$$x同余a_2(mod m_2)$$
$$x同余a_k(mod m_k)$$
$$M = m_1 m_2 \dots m_k$$
$$M_i = \frac{M}{m_i}$$
$$x = a_1 M_1 {M_1}^{-1} + a_2 M_2 {M_2}^{-1} + \dots + a_k M_k {M_k}^{-1}$$

线性代数

高斯消元

$O(n^3)$

枚举每一列c

1. 找到c列 绝对值最大的一行（在剩下的n-r行中）
2. 将该行换到最上面
3. 将改行第一个数变成1
4. 将下面所有行的第c列消成0

```
// a[N][N]是增广矩阵
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c ++ )//用第r行把第c类给清0
    {
        int t = r;
        for (int i = r; i < n; i ++ )   // 找到这一列 绝对值最大的行(r-n)中
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;//t为绝对值对大的行

        if (fabs(a[t][c]) < eps) continue;// 这一列全是0

        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   包含等于n
        for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c];      // 将当前行的首位变成1 之所以要从n开始是因为我们也要修改a[r][c]的值
        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)//如果不是0
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[i][c] * a[r][j];//a[i][c]的值最后修改

        r ++ ;//第r行的消元结束了 考虑用第r+1行的消元
    }

    if (r < n)
    {
        for (int i = r; i < n; i ++ )//不包含等于n
            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 -- )//从最后一行开始回代
        for (int j = i + 1; j < n; j ++ )//
            a[i][n] -= a[i][j] * a[j][n];

    return 0; // 有唯一解
}
```

解异或线性方程组

枚举每一列c

```
int gauss()
{
    int c,r;
    for(c=0,r=0;c<n;c++)   //按列进行枚举
    {
        int t = r;     //找到非0行，用t进行存储
        for(int i=r;i<n;i++)
            if(g[i][c])
                t=i;

        if(!g[t][c]) continue; //没有找到1，继续下一层循环

        for(int i=c;i<=n;i++) swap(g[r][i],g[t][i]);  //把第r行的数与第t行交换。

        for(int i=r+1;i<n;i++)    //用r行把下面所有行的当前列消成0
            if(g[i][c])
                for(int j=n;j>=c;j--)
                    g[i][j] ^= g[r][j];
        r++;
    }
    if(r<n)
    {
        for(int i=r;i<n;i++)
            if(g[i][n]) return 2;//无解
        return 1;//无穷多组解
    }

    for(int i=n-1;i>=0;i--)
        for(int j=i+1;j<n;j++)
            g[i][n] ^= g[i][j] * g[j][n];
    return 0;
}
```

组合计数

递推法求组合数

$C_a^b = C_{a-1}^b + C_{a-1}^{b-1}$ 第a个选与不选

预处理一 出所有的组合数 适用于询问特别多的情况 a b 范围不大 $O(n^2)$

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

预处理逆元求组合数

预处理二

$\frac{a}{b} \neq \frac{a \mod p}{b \mod p}$ 所以需要考虑逆元

$C_a^b = \frac{a!}{(a-b)!b!}$ 预处理所有阶乘的逆元 $O(nlogn)$

```
首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
如果取模的数是质数，可以用费马小定理求逆元
int qmi(int a, int k, int p)    // 快速幂模板 O(logn)
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
```

```
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i ++ )
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

Lucas 定理

$C_n^m = C_{n \mod p}^{m \mod p} \times C_{n/p}^{m/p}(\mod p)$ 证明 将a b作为p进制分解 之后考虑$(1+x)^p$和$(1+x)^a$ 考虑$(1+x)^a$中的x^b的系数

$O(log_p n) \times O(plogp)$ 求超大的m n

```
若p是质数，则对于任意整数 1 <= m <= n，有：
    C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)

int qmi(int a, int k, int p)  // 快速幂模板
{
    int res = 1 % p;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

int C(int a, int b, int p)  // 通过定理求组合数C(a, b) O(N)
{
    if (a < b) return 0;

    LL x = 1, y = 1;  // x是分子，y是分母
    for (int i = a, j = 1; j <= b; i --, j ++ )
    {
        x = (LL)x * i % p;
        y = (LL) y * j % p;
    }

    return x * (LL)qmi(y, p - 2, p) % p;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p) return C(a, b, p);
    return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}
```

分解质因数法求组合数

```
当我们需要求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较好用：
    1. 筛法求出范围内的所有质数
    2. 通过 C(a, b) = a! / b! / (a - b)! 这个公式求出每个质因子的次数。 n! 中p的次数是 n / p + n / p^2 + n / p^3 + ...   1 2 3 4 5 6 ... p ... 2p .... np...
    3. 用高精度乘法将所有质因子相乘

int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉


void get_primes(int n)      // 线性筛法求素数
{
    for (int i = 2; i <= n; i ++ )
    {
```

```
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}


int get(int n, int p)       // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ ) //每个质因子 循环这个质因子的次数遍
        res = mul(res, primes[i]);


```

卡特兰数

几何意义 方格路径 0 向右走一格 1向上走一格 C(2n,n)是总的方案数 一共2n步 从中选择n步向上

将不合法的路径转化到另一个点的方案 这个点数(n-1,n+1)

```
给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为：
                                Cat(n) = C(2n, n) / (n + 1) = C(2n,n) - C(2n,n-1)
```

容斥原理

表达式中 一共有 $C_n^1 + C_n^2 + \ldots + Cn^n = 2^n - 1$ 项 $O(2^n)$

考察容斥原理的等式两边 每一个数是否只被算了1次

考虑一个数x 属于k个集合 k<=n

组合恒等式

$kC_n^k = nC_{n-1}^{k-1}$

$C_n^k C_k^m = C_n^m C_{n-m}^{k-m}$

$(-1)^i C_n^i = 0 \ i = 0, 1, 2, \ldots, n$

$C_n^k + C_{n+1}^k + \ldots + C_{n+m}^k = C_{n+m+1}^{k+1}$

$C_n^0 C_m^p + C_n^1 C_m^{p-1} + C_n^2 C_m^{p-2} \ldots + C_n^p C_m^0 = C_{m+n}^p$

博弈论

Nim游戏

给定N堆物品，第i堆物品有Ai个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否必胜。

我们把这种游戏称为NIM博弈。把游戏过程中面临的状态称为局面。整局游戏第一个行动的称为先手，第二个行动的称为后手。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面必败。
所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对面面临必败局面，则优先采取该行动。同时，这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况，即两人均无失误，都采取最优策略行动时游戏的结果。
NIM博弈不存在平局，只有先手必胜和先手必败两种情况。

定理： NIM博弈先手必胜，当且仅当 A1 ^ A2 ^ ... ^ An != 0 (所有数成对 异或值必为0 但是异或值为0 不一定所有数成对 如1^ 2 ^3)

举例 有两堆物体 2 3 先手拿3中的1个 之后和对手镜像的拿 即必胜

一般的 x的二进制表示中最高的一位1在第k位， 设a_i的第k位是1 使得ai中只剩下a_i^x个 即可使得异或和为0

先手必胜状态：可以走到某一个必败状态

先手必败状态：走不到任何一个必败状态

台阶-Nim游戏 可以看成 只有奇数台阶的Nim游戏 如果a把石子从偶数移到奇数 可以再把它移到偶数 直到0为止

公平组合游戏ICG

若一个游戏满足：

由两名玩家交替行动；
在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
不能行动的玩家判负；
则称该游戏为一个公平组合游戏。
NIM博弈属于公平组合游戏，但城建的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件2和条件3。

有向图游戏

给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。
任何一个公平组合游戏都可以转化为有向图游戏。具体方法是，把**每个局面看成图中的一个节点**，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

Mex运算

设S表示一个非负整数集合。定义mex(S)为求出不属于集合S的最小非负整数的运算，即：
mex(S) = min{x}, x属于自然数，且x不属于S

SG函数

在有向图游戏中，对于每个节点x，设从x出发共有k条有向边，分别到达节点y1, y2, ..., yk，定义SG(x)为x的后继节点y1, y2, ..., yk的SG函数值构成的集合再执行mex(S)运算的结果，即：
SG(x) = mex({SG(y1), SG(y2), ..., SG(yk)})
特别地，整个有向图游戏G的SG函数值被定义为有向图游戏起点s的SG函数值，即SG(G) = SG(s)。**终点的SG函数是0**

有向图游戏的和

设G1, G2, ..., Gm 是m个有向图游戏。定义有向图游戏G，它的行动规则是任选某个有向图游戏Gi，并在Gi上行动一步。G被称为有向图游戏G1, G2, ..., Gm的和。
有向图游戏的和的SG函数值等于它包含的各个子游戏SG函数值的异或和，即：
SG(G) = SG(G1) ^ SG(G2) ^ ... ^ SG(Gm)

Nim游戏+单个有向图游戏

定理

有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0。
有向图游戏的某个局面必败，当且仅当该局面对应节点的SG函数值等于0。

因为SG函数值大于0的节点可以使得到达的节点SG函数是0 ， 而SG函数是0的节点只能到达SG函数是非0的节点。

计算几何

准备工作

```
#define LD double
#define Vector Point
#define Re register int
const LD eps=1e-8;//据说：出题的大学生们基本上都是用的这个值
inline int dcmp(LD a){return a<-eps?-1:(a>eps?1:0);}//处理精度
inline LD Abs(LD a){return a*dcmp(a);}//取绝对值
struct Point{
    LD x,y;Point(LD X=0,LD Y=0){x=X,y=Y;}
    inline void in(){scanf("%lf%lf",&x,&y);}
    inline void out(){printf("%.2lf %.2lf\n",x,y);}
};
```

枚举

位运算枚举

```
//枚举 1- 2^n
for(i=1;i<2^n;i++)//把i看成二进制数 每一位表示对应集合的选择情况 (n个集合)
```

动态规划

状态表示 $f[i][j]$ **表示满足i j条件的集合 维护一个属性** max min 数量之类

状态计算 集合划分 把$f[i][j]$集合分成多个子集 不重 不漏

优化 等价变形 先写基本的 之后优化

dp 如果i-1 那么下标最好是从1开始

复杂度 状态数量*转移计算量

能求出答案 维度越小越好

背包问题

01背包

每件物品最多只用一次

$f[i][j]$ 表示所有选法 : 只考虑前i个物品 容量在j以内

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

优化 由于每次迭代f[i,j]只依赖于f[i-1,j-k] 所以可以使用滚动数组 优化掉i 节省空间

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

逆序保证 f[j]中只会有1个第i件物品

完全背包

每间物品有无限个

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

优化 一是空间 二从状态转移方程入手

$f[i,j]=max(f[i-1,j],f[i-1,j-v]+w,f[i-1,j-2v]+2w,f[i-1,j-3v]+3w,\dots,f[i-1,j-kv]+kw)j>=kv$

$f[i,j-v]=max(f[i-1,j-v],f[i-1,j-2v]+w,f[i-1,j-3v]+2w,\dots,f[i-1,j-kv]+(k-1)w)j>=kv$

可以看到 $f[i,j]$和$f[i,j-v]$高度相似 所以 $f[i,j]=max(f[i-1,j]+f[i,j-v]+w)$

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

顺序更新 保证 f[j]中会有任意多件第i件物品

多重背包

```
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;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]);
```

优化 二进制优化 把s 分成logs件 比如 s=7 分成三组 1 2 4

```
        int k = 1;
        while (k <= s)
        {
            cnt ++ ;
            v[cnt] = a * k;
            w[cnt] = b * k;
            s -= k;
            k *= 2;
        }
        if (s > 0)
        {
            cnt ++ ;
            v[cnt] = a * s;
            w[cnt] = b * s;
        }
```

分组背包

f[i,j] 只从前i组物品中选 且总体积大于j的所有选法

```
    for (int i = 1; i <= n; i ++ )
        for (int j = m; j >= 0; j -- )
            for (int k = 0; k < s[i]; k ++ )//第i类的k种物品
                if (v[i][k] <= j)
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
```

线性DP

一维线性 二维 多维状态

数字三角形

状态表示 f[i,j] 集合表示到第i行第j列的所有走法 属性 数字和最大值

状态计算 集合划分 左上 右下

```
    for (int i = 0; i <= n; i ++ )
        for (int j = 0; j <= i + 1; j ++ )//初始化要覆盖从0 到i+1的位置
            f[i][j] = -INF;

    f[1][1] = a[1][1];
    for (int i = 2; i <= n; i ++ )
        for (int j = 1; j <= i; j ++ )//从1开始比较方便
            f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i][j]);

    int res = -INF;
    for (int i = 1; i <= n; i ++ ) res = max(res, f[n][i]);

```

最长上升子序列

f[i]表示第i位数字为结尾的 子序列的最大值

状态表示 **集合** 第i位数字为结尾的子序列 **属性** 长度最大值

状态计算 以倒数第二位的小标作为区分 0(没有第二位) 1 2

```
    for (int i = 1; i <= n; i ++ )
    {
        f[i] = 1; // 只有a[i]一个数
        for (int j = 1; j < i; j ++ )//考虑倒数第二个数是第j位的数
            if (a[j] < a[i])//保证上升
                f[i] = max(f[i], f[j] + 1);
    }
```

优化 f[i]表示 i长度的子序列的最后一个值

状态表示 **集合** 长度为i的上升子序列 **属性** 子序列最后一位的最小值

状态计算 考察长度为i-1的子序列 关注f[i-1]和a[i]的关系

```
memset(f, 0x3f, sizeof f);
f[1] = a[1];
f[0] = -INF;
  for (int i = 2; i <= n; i++) {//每一个新数进来 更新f[] f[i]被更新当且仅当f[i-1]<a[i] 或者 a[i]>f[last]
    for (int j = 1; j <= i; j++) {//f[]有序
      if(f[j-1]<a[i])//只会更新这样的j f[j-1]<a[i] 而 f[j]>=a[i]
          f[j] = min(a[i], f[j]);
      }
    }
```

再优化 实际上 每加入一个新数 f[] 至多只有一项被更新一次 因为在f[0]=-INF 所以必定会更新一次 (无论a[i]多小 a[i]总大于-INF)

在f[j]被更新之后 下一个循环 f[j+1]=a[i] 不可能小于a[i] 同时f[]又是单调的 所以 会且只会更新一次

```
int len = 0;
for (int i = 0; i < n; ++i)
{
    if (f[len] < a[i])//如果是不严格上升子序列  那么这边可以带等号
        f[++len] = a[i];
    else
        *lower_bound(f + 1, f + len + 1, a[i]) = a[i];//如果是不严格上升子序列  那么这边改成upper_bound
}
```

最长公共子序列

两个序列 两维的表示

状态表示 **集合** 所有在第一个序列的前i个字母中出现且在第二个序列的前j个字母中出现的子序列 **属性** Max

状态计算 以a[i] b[j] 是否包含在f[i] [j]中 f[i-1] [j-1]+1 f[i-1] [j-1] f[i] [j-1] f[i-1] [j] 这四种有重合 但是是不漏的

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

最短编辑距离

f[i] [j]表示a的前i个字母 到 b的前j个字母的编辑距离

状态表示 **集合** f[i] [j] 将a的前i个字母 和 b的前个字母变成一样的操作方式 **属性** 最小编辑距离

状态计算 如果a[i]=b[j] 那么考虑前面的 如果不等 那么删 替换 添加 （如果i j 为0 那么只能添加或者删除）

考虑最后一步是什么 添加 删除 替换

实际上证明参考 (28 封私信 / 80 条消息) 编辑距离算法的正确性证明，请问有没有相关的资料呢？ - 知乎 (zhihu.com) 这个正确性不是很显然

考虑从s1 变换到 s2 考虑变换后 s1[i]这个节点最后落到了s2[j]上(s1[i]被替换不影响) 那么在s1[i]和s2[j]之间连上一根线

容易知道 一个点上不会有两根线 并且线之间不会有交叉（插入 删除 和替换 是保序的）

编辑距离 **等于** 替换的次数+删除和插入的次数 **删除次数** 等于 s1中没有连线的个数 **插入次数** 等于 s2长度-（s1长度-删除次数）

编辑距离等于 2* s1中没有连线的个数 +s2 长度 -s1长度 +连线两端不等的个数

考虑三种将s1 s2 的划分 如下 考虑最后两个元素之间的连线 对应三种情况 最后两个元素相连 最后一个元素都没连 只有一个连 分别对应于这三种情况

1. s1[1-i] s2[1-(j-1)]
2. s1[1-(i-1)] s2[1-j]
3. s1[1-(i-1)] s2[1-(j-1)]

   证毕

```
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= m; j ++ )
        {
            f[i][j] = min(f[i - 1][j] + 1, f[i][j - 1] + 1);//添加
            if (a[i] == b[j]) f[i][j] = min(f[i][j], f[i - 1][j - 1]);//考虑前面的
            else f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);//替换
        }
```

区间dp

石子合并

```
    for (int i = 1; i <= n; i ++ ) s[i] += s[i - 1];

    for (int len = 2; len <= n; len ++ )
        for (int i = 1; i + len - 1 <= n; i ++ )
        {
            int l = i, r = i + len - 1;
            f[l][r] = 1e8;
            for (int k = l; k < r; k ++ )
                f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s[l - 1]);
        }
```

计数类dp

整数划分

考虑类似于完全背包问题

背包体积为n 可选物品 1 物品 2 物品3

状态表示 **集合** f[i,j]所有体积为j 只是用前i个数的选法 **属性** 数量

状态计算 第i个数用了几次 有类似于完全背包的优化

```
 for (int i = 0; i <= n; i ++) {
        f[i][0] = 1; // 容量为0时，前 i 个物品全不选也是一种方案
    }

    for (int i = 1; i <= n; i ++) {
        for (int j = 0; j <= n; j ++) {
            f[i][j] = f[i - 1][j] % mod; // 特殊 f[0][0] = 1
            if (j >= i) f[i][j] = (f[i - 1][j] + f[i][j - i]) % mod;
        }
    }
```

空间优化

```
f[0] = 1; // 容量为0时，前 i 个物品全不选也是一种方案

    for (int i = 1; i <= n; i ++) {
        for (int j = i; j <= n; j ++) {
            f[j] = (f[j] + f[j - i]) % mod;
        }
    }
```

状态表示 **集合** f[i,j]表示 所有总和是i 并且恰好表示成j个数的和的方案 **属性** 数量

状态计算 最小值是1 最小值大于1 f[i,j]=f[i-1,j-1] +f[i-j,j]

```
    f[1][1] = 1;
    for (int i = 2; i <= n; i ++ )
        for (int j = 1; j <= i; j ++ )
            f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % mod;

    int res = 0;
    for (int i = 1; i <= n; i ++ ) res = (res + f[n][i]) % mod;
```

数位统计dp

分情况讨论

计数问题

给定两个整数a b 求 a b之间所有数字中 0-9出现的次数

count(n,x),1-n中x出现的次数 x 属于0-9

n=abcdefg 分别求出i在每一位上出现的次数

求i在第4位出现的次数 i>0

1<=xxxiyyy<=abcdefg

1. 前缀xxx=000-(abc-1) yyy 就有1000中选法
2. 前缀xxx=abc
    1. d<i 因为abc1yyy >abc0efg   yyy没有取法
    2. d==i abc1yyy 有efg+1种选法
    3. d>i yyy 有1000种选法

求0在第k位出现的次数

1. 前缀001-(abc-1)

```
int count(int n, int i) {  // 0到n中 数字x出现的次数
  int ans = 0;
  int t = n, s = 1;  // s=10^nlast nlast是当前枚举位的后缀长度,t是当前枚举的前缀和当前枚举位的拼接
  if (i) {
    while (t) {
      //先计算前导小于pre[t]的情况
      ans += (t / 10) * s;  // 0-t-1 一共t 个
      //计算前导等于t的情况
      int x = t % 10;  // x是n中当前枚举这位的值
      if (x == i) ans += (n % s + 1);
      if (x > i) ans += s;
      t /= 10,s *= 10;
    }
  }
  if (i == 0) {//枚举0的时候需要额外考虑
    while (t) {
      //先计算前导小于pre[t]的情况
      int pre = t / 10;
      if (pre >= 1) {//前导必须大于0
        ans += (pre - 1) * s;
        int x = t % 10;  // x是n中当前枚举这位的值
        if (x == i) ans += (n % s + 1);
        if (x > i) ans += s;
      }
      t /= 10, s *= 10;
    }
  }

  return ans;
}
```

另一种求法

```
int power10(int x){
    int res=1;
    while (x--)res*=10;
    return res;
}
int get(vector<int>num,int l,int r){
    int res=0;
    for(int i=l;i>=r;i--){
        res=res*10+num[i];
    }
    return res;
}
int count(int n,int x){
    if(!n)return 0;
    vector<int>num;
    while (n)
    {
        int t=n%10;
        n/=10;
        num.push_back(t);
    }
    n=num.size();
```

```
    int res=0;
    for(int i=n-1-!x;i>=0;i--){
        if(i<n-1){
            res+= get(num,n-1,i+1)* power10(i);
            if(!x)res-= power10(i);
        }
        if(num[i]==x)res+=get(num,i-1,0)+1;
        else if(num[i]>x){
            res+= power10(i);
        }
    }
    return res;
}
```

状态压缩dp

把f[i] [j] 中的i j这样的数 看成是一个二进制数 每一个bit表示一种状态 n 小于20

蒙德里安的梦想

摆放方块的时候，先放横着的，再放竖着的。总方案数等于只放横着的小方块的合法方案数。

如何判断，当前方案数是否合法？ 所有剩余位置能否填充满竖着的小方块。可以按列来看，每一列内部所有连续的空着的小方块需要是偶数个。

用一个N位的二进制数，每一位表示一个物品，0/1表示不同的状态。 2^N-1（N二进制对应的十进制数）中的所有数来枚举全部的状态。
状态表示：f[i] [j]表示已经将前 i -1 列摆好，且从第i-1列，伸出到第 i列的状态是 j 的所有方案。其中j是一个二进制数，用来表示哪一行的小方块是横着放的，其位数和棋盘的行数一致。

状态转移

首先k不能和 j在同一行既然不能同一行伸出来，那么对应的代码为(k & j) == 0 ，表示两个数相与，如果有1位相同结果就不是0，(k & j) ==0表示 k和j没有1位相同， 即没有1行有冲突。

既然从第i-1列到第i列横着摆的，和第i-2列到第i-1列横着摆的都确定了，那么第i-1列 空着的格子就确定了，这些空着的格子将来用作竖着放。如果 某一列有这些空着的位置，那么该列所有连续的空着的位置长度必须是偶数。

时间复杂度
dp的时间复杂度 =状态表示 × 状态转移

状态表示 f[i] [j] 第一维i可取11，第二维j（二进制数）可取2^{11} ，所以状态表示 11×2^{11}
状态转移 也是2^{11}
所以总的时间复杂度 4×1e7 可以过

```
        for (int i = 0; i < 1 << n; i ++ )//预处理
        {
            int cnt = 0; //从第一行开始计数
            st[i] = true;//i 表示的每列方格状态
            for (int j = 0; j < n; j ++ )//遍历每一行
                if (i >> j & 1)//第j行有方格占了
                {
                    if (cnt & 1) st[i] = false;
                    cnt = 0;
                }
                else cnt ++ ;//cnt统计空格
            if (cnt & 1) st[i] = false;//最后一串空格
        }

        memset(f, 0, sizeof f);
        f[0][0] = 1;
        for (int i = 1; i <= m; i ++ )//
            for (int j = 0; j < 1 << n; j ++ )//枚举第m列每一个的状态
                for (int k = 0; k < 1 << n; k ++ )//枚举 上一列 的每一个状态
                    if ((j & k) == 0 && st[j | k])
                        f[i][j] += f[i - 1][k];//f[i][j] 第i列的放置的情况是j 指的是方块的左端点是j

        cout << f[m][0] << endl;
```

最短hamiton路径

状态表示 f[i] [j] **集合** 所有从0走到j 走过的所有点是i (二进制表示) 的所有路径 **属性** min

状态计算 根据倒数第二位分类

```
    memset(f, 0x3f, sizeof f);
    f[1][0] = 0;

    for (int i = 0; i < 1 << n; i ++ )
        for (int j = 0; j < n; j ++ )
            if (i >> j & 1)//找到到达了j这个点的 i 表示
                for (int k = 0; k < n; k ++ )//从第k位来的
                    if (i >> k & 1)//i也是到了k的
                        f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j]);

    cout << f[(1 << n) - 1][n - 1];
```

树形dp

没有上司的舞会

状态表示 **集合** f[u] [0] 从所有以u为根节点的子树中选的方案 不选取 u f[u] [1] 从所有以u为根节点的子树中选 选取u的方案 **属性** max

状态计算 考虑选不选u 那么子树的根就可以知道选不选

```
void dfs(int u)
{
    f[u][1] = happy[u];

    for (int i = h[u]; ~i; i = ne[i])//遍历儿子
    {
        int j = e[i];
        dfs(j);//得到f[j][0] 和f[j][1]

        f[u][1] += f[j][0];//f[u][1]选取u 所以 f[j][0] 不能选取j 这个u的子节点
        f[u][0] += max(f[j][0], f[j][1]);//如果我们不选取u 那么j 可以选也可以不选
    }
}

```

记忆化搜索

滑雪

```
int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};

int dp(int x, int y)
{
    int &v = f[x][y];
    if (v != -1) return v;

    v = 1;
    for (int i = 0; i < 4; i ++ )//枚举四个方向
    {
        int a = x + dx[i], b = y + dy[i];
        if (a >= 1 && a <= n && b >= 1 && b <= m && g[x][y] > g[a][b])
            v = max(v, dp(a, b) + 1);
    }

    return v;
}
```

贪心

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如n=1 ）的最优解F1 ，然后再证明：对于每个n+1， 都可以由 Fn推导出结果。

排序

选择当前最好的情况

区间问题

区间选点

按区间**右端点** 从小到达排序 从前往后 枚举每一个区间 已经包含 pass 否则选取右端点 O(n)

证明 ans<=cnt ans表示最优解 cnt表示算法得到的解

cnt>=ans cnt 覆盖了cnt个 没有交集的区间 而 ans 还覆盖了一些额外的区间

```
sort(p + 1, p + n + 1);
int point = -INF, ans = 0;
for (int i = 1; i <= n; i++) {
  if (point >= p[i].second && point <= p[i].first)
    continue;
  else
    point = p[i].first, ans++;
}
cout << ans;
```

最大不相交区间数量

算法和上面一样

证明 ans<=cnt 反证 ans>cnt cnt 个点覆盖了所有的区间 但是 ans个点覆盖了没有交集的区间 而ans>cnt 这是不可能的

cnt<=ans cnt 选出的一定没有交集 所以是符合条件的解 ans是最优解是最大值

区间分组

将所有区间按左端点从小到大枚举 O(nlogn)

从前往后处理每个区间 判断能否将其放到某个现有的组中

证明 ans<=cnt ans是最优的区间分组数 cnt也是一个合法的分组数

ans>=cnt 证明 ans>cnt-1 算法在进行到cnt-1时 第一次遇到需要新建组的情况 实际上就是这个组的左端点 和cnt-1个组的点都有交集了 所以cnt-1个组无法完成没有交集的组的划分

```
    sort(range, range + n);

    priority_queue<int, vector<int>, greater<int>> heap;
    for (int i = 0; i < n; i ++ )
    {
        auto item = range[i];
        if (heap.empty() || heap.top() >= r.l) heap.push(item.r);
        else
        {
            heap.pop();
            heap.push(item.r);
        }
    }
```

区间覆盖

将所有区间**按左端点从小到大排序**

从前往后依次枚举每个区间 选取 要覆盖区间的左端点 可以被覆盖 (此选取区间的右端点最大) 然后更新要覆盖区间的左端点

证明 ans<=cnt 算法做出来的解cnt是合法的

ans>=cnt 反证 假设 ans<cnt 把ans中的区间和cnt中的替换 解依然是合法的 因为cnt找到的是右端点最大的

```
    sort(range, range + n);

    int res = 0;
    bool success = false;
    for (int i = 0; i < n; i ++ )
    {
        int j = i, r = -2e9;
        while (j < n && range[j].l <= st)
        {
            r = max(r, range[j].r);
            j ++ ;
        }//遍历所有左端点在st左边 右端点的最大值

        if (r < st)//如果不能覆盖
        {
            res = -1;
            break;
        }

        res ++ ;
        if (r >= ed)//如果覆盖到了结束
        {
            success = true;
            break;
        }

        st = r;//更新st
        i = j - 1;//j复原
    }

    if (!success) res = -1;
    printf("%d\n", res);
```

Huffman树

每次挑权值最小的点合并

最小的两个点 深度一定最深并且是相同的(因为huffman树是二叉树 最深的点至少两个) 不然通过调整 权值路径和可以优化

这意味着我们需要先合并这两个最小的节点

合并果子

```
    priority_queue<int, vector<int>, greater<int>> heap;
    while (n -- )
    {
        int x;
        scanf("%d", &x);
        heap.push(x);
    }

    int res = 0;
    while (heap.size() > 1)
    {
        int a = heap.top(); heap.pop();
        int b = heap.top(); heap.pop();
        res += a + b;
        heap.push(a + b);
    }

    printf("%d\n", res);
```

排序不等式

排队打水

求等待时间最小

总时间 等于t1*(n-1) + t2 *(n-2) + ...； **调整法**

从小到大排序

绝对值不等式

货仓选址

列出绝对值不等式 观察和的特点 把每一对和分组

推公式

耍杂技的牛

这 N 头奶牛中的每一头都有着自己的重量 Wi 以及自己的强壮程度 Si。

一头牛支撑不住的可能性取决于它头上所有牛的总重量（不包括它自己）减去它的身体强壮程度的值，现在称该数值为风险值，风险值越大，这只牛撑不住的可能性越高。

您的任务是确定奶牛的排序，使得所有奶牛的风险值中的最大值尽可能的小。

按照wi+si从小到大的顺序排，最大的危险系数一定是最小的

考虑交换第i个位置上的牛和第i+1个位置上的牛 交换前后 值的影响

后悔法

Work Scheduling G

每一项工作安装截止时间 收益大小 排序 收入数组中

使用一个优先队列维护已经完成的工作中的最小值 优先队列中存放所有预计完成的工作

从前往后枚举数组 如果当前工作无法完成那么 之前的一项工作就有可能被替换成这个工作

```
sort(a, a + n);//按照ddl时间 从小到大排序
long long ans = 0;
priority_queue<int, vector<int>, greater<int>> q;//q中存放着所有预计完成的任务 和最小的p
for (int i = 0; i < n; i++) {
  if (a[i].d <= q.size()) {//截止日期超过了工作数
    if (q.top() < a[i].p)
      ans -= q.top(), ans += a[i].p, q.pop(), q.push(a[i].p);//p表示收益
  } else
    ans += a[i].p, q.push(a[i].p);
}

cout << ans;
```

枚举到第i项工作时如果工作超时了

那么就穿越回过去在之前做过的工作中挑一个报酬最低的

然后在那一项工作那天做第i项的工作（前提是第i项工作报酬要高于过去最低报酬的那天的工作，不然没有收益）

时空复杂度

基础算法	时间复杂度
快排 归并	$O(nlogn)$
二分	$O(logn)$
高精度	$O(logn)$
前缀和 差分	初始化$O(n)$操作$O(1)$
双指针	$O(n)$
位运算	$O(1)$
离散化	初始化$O(nlogn)$操作$O(logn)$
区间合并	$O(nlogn)$

数据结构、搜索与图论	时间复杂度
单调队列 单调栈 kmp trie树	$O(n)$
并查集	$O(nlogn)$但是实际很快
搜索	$O(n+m)$$n$表示点数,$m$表示边数
拓扑排序	$O(n+m)$
朴素dijkstra	$O(n^2+m)$
堆优化dijstra	$O(mlogm)$
spfa bellman-ford	$O(nm)$
floyd	$O(n^3)$
prim	$O(n^2+m)$
kruskal	$O(mlogm)$
匈牙利算法	$O(nm)$

树形dp的时间复杂度是$O(n)$

递归也是需要空间的，递归调用了系统栈，快速排序使用了递归，所以空间复杂度是O(logn)

int 数组1e7

C++

O2

```
#pragma GCC optimize(2)
```

stoi 和atoi

```
stoi（字符串，起始位置，n进制），将 n 进制的字符串转化为十进制

示例：
stoi(str, 0, 2); //将字符串 str 从 0 位置开始到末尾的 2 进制转换为十进制

atoi()的参数是 const char* ,因此对于一个字符串str我们必须调用 c_str()的方法把这个string转换成 const char*类型的,而stoi()的参数是const string*,不需要转化为 const char*;
```

puts

```
int puts(const char *s);
//系统会自动在其后添加一个换行符
```

scanf

读入字符时 不会忽略空格 读入字符串时会

需要使用字符串来读字符

mod

负数%正数 得到负数 c++是这样 数学上余数应该都是正数

(x % N + N)%N

memset

```
memset(h,0,size(h));
memset(h,-1,size(h));//最为常用的两种

void* memcpy( void* dest, const void* src, std::size_t count );
```

strcpy和memcpy主要有以下3方面的区别。

1、复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等。

2、复制的方法不同。strcpy不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy则是根据其第3个参数决定复制的长度。

3、用途不同。通常在复制字符串时用strcpy，而需要复制其他类型数据时则一般用memcpy。

枚举类型

```
enum <类型名> {<枚举常量表>};
enum color_set1 {RED, BLUE, WHITE, BLACK};
color3=RED;            //将枚举常量值赋给枚举变量
color4=color3;         //相同类型的枚举变量赋值，color4的值为RED
int  i=color3;         //将枚举变量赋给整型变量，i的值为1
int  j=GREEN;          //将枚举常量赋给整型变量，j的值为0
```

重载运算符

大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box operator+(const Box&)const;
Box operator+(const Box&, const Box&);
```

反三角函数

```
arcsin用asin
arccos用acos
arctan用atan
```

读入数据

```
string s;
getline(cin, s);
cout << s << endl;//读入一整行的数据
```

从数据范围反推复杂度和算法内容

1. $n \leq 30$, 指数级别 , dfs+剪枝，状态压缩dp (这个n还是越小越好)
2. $n \leq 10^2 => O(n^3), O(n^3logn)$，floyd，dp，高斯消元
3. $n \leq 10^3 => O(n^2), O(n^2logn)$，dp，二分+x，朴素版Dijkstra、朴素版Prim、Bellman-Ford
4. $n \leq 10^4 => O(n * \sqrt{n})$，块状链表、分块、莫队
5. $n \leq 10^5 => O(nlogn)$各种sort，线段树、树状数组、set/map、heap、拓扑排序、dijkstra+heap、prim+heap、Kruskal、spfa、求凸包、求半平面交、二分+x、CDQ分治、整体二分+x、后缀数组、树链剖分、动态树
6. $n \leq 10^6 => O(n), O(nlogn)$单调队列、 hash、双指针扫描、并查集，kmp、AC自动机，常数比较小的O(nlogn)O(nlogn) 的做法：sort、树状数组、heap、dijkstra、spfa
7. $n \leq 10^7 => O(n), O(nlogn)$双指针扫描、kmp、AC自动机、线性筛素数
8. $n \leq 10^9 => O(\sqrt{n})$判断质数
9. $n \leq 10^{18} => O(logn)$最大公约数，快速幂，数位DP
10. $n \leq 10^{1000} => O((logn)^2)$高精度加减乘除
11. $n \leq 10^{100000} => O((logk \times loglogk))$k表示位数 ， 高精度加减、FFT/NTT

出错解决方法

segmentation fault / runtime error

删除局部代码

出错

cout大法

== = 写错 ！漏掉

自环 负权

数组越界之后 有可能会发生所有错误

更新顺序

1. 在01背包的一维优化中 对于体积的枚举只能从大到小 如果从小达到 那么在一轮更新中 会使用刚刚更新的值
2. 在有边数限制的最短路一题中 要使用额外的数组存储上一次得到的距离 不然就会出现使用刚更新的值更新
3. 在高斯消元中 由于a[r] [c]也要被更新 但是也会被使用 所以也要逆序更新