TEMPLATE

2016CCPC 合肥准备

2016.10

目录

热身赛前准备：

0、编辑环境惯用配置（特别是 Ubuntu 下 Codeblock 使用配置）；

1、如果使用 JAVA，对大数、主类名及类似 isPrime 等常用方法的测试，熟悉 Java 内置算法的使用；

2、试验数组能开多大，是否有 MLE；

3、试验是否有 PE；

4、RE 的时候是返回 WA 还是 RE；

5、memset 是否要加 memory.h，还是 string.h；测试 bits/stdc++.h 是否支持

6、stdio.h 和 cmath 和 cstring 是否需要；

7、测算 CPU 运行速度，10^8

8、试验打印功能和与裁判交流的功能；

9、STL-vector,string,map,set 等尝试一下能不能用（一定要熟悉 STL 中的各种结构和算法）；

10、机器是否会开机还原；

11、C++和 JAVA 有没有自动补全；

12、代码长度是否有限制 ；

13、试一下栈空间大小，能否通过预处理指令增加栈空间（如果不明白此项，请忽略）；

14、64 位输入、输出的标识的确认（lld/I64d？？ ）!

15、搞清楚洗手间的位置；

16、热身赛当晚最好搞一场简单点的 2~3 小时的比赛，保持手的热度（避免手生）；

主体思路是跟题为主，但队伍需要商量确定如何跟题，另外在跟题的时候不要所有人都扑在一个题目上，要准备其他题目，做好火力连续输出的准备；

看题目要求输入加 while

交题之前检查：

运行所有样例，数组是否开够大，能否将 cin>>变成 scanf

头文件没有复制少了，printf 注释语句都注释掉了，

YES 还是 Yes

输入 while（）需不需要加！=EOF

！注意逛 clarification !!! 只看 admin 的。

字符串相关

基础知识

strstr(str1,str2) 函数用于判断字符串 str2 是否是 str1 的子串。如果是，则该函数返回 str2 在 str1 中首次出现的地址；否则，返回 NULL。

strcmp(const char *s1,const char * s2)这里面只能比较字符串，即可用于比较两个字符串常量，或比较数组和字符串常量，不能比较数字等其他形式的参数

当 s1<s2 时，返回为负数；当 s1=s2 时，返回值= 0；当 s1>s2 时，返回正数。

在 hash 时将 a 当做 1 可以区分 aa 和 aaa

KMP

（O(m+n)）

```
int Next[100005];
void getNext(char b[],int m)
{
    int j,k;
    j=0;k=-1;
    Next[0]=-1;
    while(j<m)
    {
        if(k==-1||b[j]==b[k]) Next[++j]=++k;
        else k=Next[k];
    }
}
int KMP_Index(char a[],char b[])
{
    int i=0,j=0;
    int n=strlen(a);
    int m=strlen(b);
    getNext(b,m);
    while(i<n&&j<m)
    {
        if(j==-1||a[i]==b[j])
        {
            i++;j++;
        }else j=Next[j];
    }
    if(j==m) return i-j+1;//返回第一个找到的点
////每次匹配一个字母 j++ 此时 j 为匹配字母下一位 若为 m
    else return -1;
}
```

```
int Next[100005];//优化求 next 数组方法（少用）
void getNext(char b[],int m)
{
    int j,k;
    j=0;k=-1;
    Next[0]=-1;
    while(j<m)
    {
        if(k==-1||b[j]==b[k])
        {
            j++;k++;
            if(b[j]!=b[k])
            Next[j]=k;
            else Next[j]=Next[k];
        }
        else k=Next[k];
    }
}
```

字典树（trie）

```
// 使用 trie 树结构。在 trie 树节点中加入两个域 isword 和 cnt。
//cnt 表示有多少个单词经过这个节点。
//先将所有单词保存在 trie 树中，然后一个一个地查找，当到达某个节点使用 cnt==1
//那么从根到该节点组成的字符串便是该单词的最短前缀。
struct Node {
int cnt;//cnt 表示有多少个单词经过这个节点。
int next[26];  //子树
Node() //c++默认构造函数
{
    cnt=0;
    memset(next,-1,sizeof next);
}
    void init(){
        cnt = 0;
        memset(next, -1, sizeof(next));
    }
};
Node T[100000];
char words[1005][25];
int len=1;//T 数组下标
char pre[30];
int idx=0;
void insert(char *tar)
```

```
{
    Node *p=&T[0];
    int mid;
    while(*tar)
    {
        p->cnt++;
        mid=*tar-'a';
        if(p->next[mid]==-1)
        {       T[len].init();
            p->next[mid]=len;//找到下一位 数组下标
            len++;
        }
        p=&T[p->next[mid]];//利用数组
        tar++;
    }
    p->cnt++;
}
void search(char *tar)
{
    Node *p=&T[0];
    int id;
    idx=0;
    while(p->cnt>1&& *tar)
    {
        pre[idx++]=*tar;
        id=*tar-'a';
        if(p->next[id]==-1)//一般搜索个数时要加这个
            return 0;
        p=&T[p->next[id]];
        tar++;
    }
    pre[idx]='\0';
}
int main()
{
    int i,j;
    i=0;len=1;
    while(~scanf("%s",&words[i]))
    {
        char temp[25];
        strcpy(temp,words[i]);
        insert(temp);
        i++;
    }
```

```
    for(j=0;j<i;++j)
    {
        char t[25];
        strcpy(t,words[j]);
        search(t);
        printf("%s %s\n",words[j],pre);
    }
    return 0;
}
//每次使用 main 函数加这两句
 len=1;
 T[0].init();
```

带指针的模板

```
const int KIND = 26;
const int MAXN = 500000;
int cnt_node;
struct node{
    int cnt;
    node* next[KIND];
    void init(){
        cnt = 0;
        memset(next, 0, sizeof(next));
    }
}Heap[MAXN];
inline node* new_node(){
    Heap[cnt_node].init();
    return &Heap[cnt_node++];
}
void insert(node* root, char *str){
    for(char *p=str; *p; ++p){
        int ch=*p-'a';
        if(root->next[ch]==NULL)
            root->next[ch] = new_node();
        root = root->next[ch];
        ++root->cnt;
    }
}
int count(node* root, char *str){
    for(char *p=str; *p; ++p){
        int ch=*p-'a';
        if(root->next[ch]==NULL)
            return 0;
        root=root->next[ch];
    }
```

```
    return root->cnt;
}
//每次使用在 main 中
 cnt_node=0;
 node *root = new_node();
```

图论

建图

```
int n,m,idx;  //去掉重边很不方便
struct Edge
{
    int v;
    int wight;
    int next;
}e[maxe];
int head[maxv]; //头结点
int vis[maxv]; //标记点,是否被访问过.
void Init()
{
    memset(head, -1, sizeof(head));
    memset(vis, 0, sizeof(vis));
    idx = 0;
}
void Addedge(int a, int b, int value) //分别为起点 a,终点 b,以及边的权值 value
{
     e[idx].wight = value;
     e[idx].v = b;
     e[idx].next = head[a];
     head[a] = idx++;  //head [a]的值表示以 a 为起点的某条边的数组下标，若其 next！=-1
则 next 为以 a 为起点的另一条边的数组下标
}
```

搜索（BFS,DFS）

```
// BFS
bool vis[N];
void bfs(int op) {
    fill(vis, vis + N, 0);
    queue <int> q;
    q.push(op);
```

```
    vis[op] = true;
    while(q.size()) {
        int x = q.front(); q.pop();
        for(int k = last[x]; ~k; k = edges[k].next) {
            int v = edges[k].v;
            if(vis[v]) continue;
            vis[v] = true;
            q.push(v);
        }
    }
}
// DFS
void dfs(int x) {
    if(vis[x]) return;
    vis[x] = true;
    for(int k = last[x]; ~k; k = edges[k].next) {
        int v = edges[k].v;
        dfs(v);
    }
}
```

tarjan（求解有向图强连通分量的线性时间的算法）

```
#define M 5010//题目中可能的最大点数
int STACK[M],top=0;//Tarjan 算法中的栈
bool InStack[M];//检查是否在栈中
int DFN[M];//深度优先搜索访问次序
int Low[M];//能追溯到的最早的次序
int ComponentNumber=0;//有向图强连通分量个数
int Index=0;//索引号
vector<int> Edge[M];//邻接表表示
vector<int> Component[M];//获得强连通分量结果
int InComponent[M];//记录每个点在第几号强连通分量里
int ComponentDegree[M];//记录每个强连通分量的度
void Tarjan(int i)
{
    int j;
    DFN[i]=Low[i]=Index++;
    InStack[i]=true;STACK[++top]=i;
    for (int e=0;e<Edge[i].size();e++)
    {
        j=Edge[i][e];
        if (DFN[j]==-1)
        {
```

```
            Tarjan(j);
            Low[i]=min(Low[i],Low[j]);
        }
        else
            if (InStack[j]) Low[i]=min(Low[i],DFN[j]);
    }
    if (DFN[i]==Low[i])
    {
        ComponentNumber++;
        do{
            j=STACK[top--];
            InStack[j]=false;
            Component[ComponentNumber].
            push_back(j);
            InComponent[j]=ComponentNumber;
        }
        while (j!=i);
    }
}
```

最小生成树 MST

Kruskal

```
//本程序用到了并查集的基本操作
//getfa 为查询祖先，merge 为将集合合并，same 是判断两个点是否处于同一集合
//getfa 操作中使用了路径压缩即 return fa[x] = getfa(fa[x])，这样可以减小并查集森林退化
所带来的时间复杂度
#define MAXN_E 100000
#define MAXN_V 100000
struct Edge{
    int fm,to,dist;
}e[MAXN_E];
int fa[MAXN_V],n,m;
bool cmp(Edge a,Edge b)
{
    return a.dist < b.dist;
}
int getfa(int x){//getfa 是在并查集森林中找到 x 的祖先
    if(fa[x]==x) return fa[x];
    else return fa[x] = getfa(fa[x]);
}
int same(int x,int y){
    return getfa(x)==getfa(y);
}
```

```
void merge(int x,int y)
{
    int fax=getfa(x),fay=getfa(y);
    fa[fax]=fay;
}
int main()
{
    scanf("%d%d",&n,&m);//n 为点数，m 为边数
    for(inti=1;i<=m;i++)
    scanf("%d%d%d",&e[i].fm,&e[i].to,&e[i].dist);//用边集数组存放边，方便排序和调用
    sort(e+1,e+m+1,cmp);//对边按边权进行升序排序
    for(int i=1;i<=n;i++)
    fa[i]=i;   //并查集初始化
 int rst=n,ans=0;//rst 表示目前的点共存在于多少个集合中，初始情况是每个点都在不同的集合
中
  for(int i=1;i<=m && rst>1;i++)
    {
        int x=e[i].fm,y=e[i].to;
        if(same(x,y)) continue;//same 函数是查询两个点是否在同一集合中
        else
        {
            merge(x,y);//merge 函数用来将两个点合并到同一集合中
            rst--;//每次将两个不同集合中的点合并，都将使 rst 值减 1
            ans+=e[i].dist;//这条边是最小生成树中的边，将答案加上边权
        }
    }
    printf("%d\n",ans);
    return0;
}
```

Prim

```
#define maxn 110
#define INF 1000    //预定于的最大值
int n;   //顶点数、边数
int linkma[maxn][maxn];      //邻接矩阵表示
struct node
{
    int v;
    int edge;
    friend bool operator<(node a, node b)   //自定义优先级，key 小的优先
    {
        return a.edge > b.edge;
    }
};
int parent[maxn];   //每个结点的父节点
```

```
bool visited[maxn]; //是否已经加入树种
node vx[maxn];      //保存每个结点与其父节点连接边的权值
priority_queue<node> q; //优先队列 stl 实现
void Prim()    //s 表示根结点
{
    for(int i = 1; i <= n; i++) //初始化
    {
        vx[i].v = i;
        vx[i].edge = INF;
        parent[i] = -1;
        visited[i] = false;
    }
    vx[1].edge = 0;
    q.push(vx[1]);
    while(!q.empty())
    {
        node temnode = q.top();  //取队首，记得赶紧 pop 掉
        q.pop();
        if(visited[temnode.v] == true)   //深意，因为 push 机器的可能是重复但是权值不同
的点，我们只取最小的
            continue;
        int connectv = temnode.v;
        visited[temnode.v] = true;
        for(int j = 1;  j <= n; j++)
        {
            if(j!=connectv && !visited[j] && linkma[connectv][j] < vx[j].edge)
//判断
            {
                parent[j] = connectv;
                vx[j].edge = linkma[connectv][j];
                q.push(vx[j]);
            }
        }
    }
}
int main()
{
    while(~scanf("%d",&n))//点的个数
    {
        for(int i = 1;i<= n; i++)//输入邻接矩阵
            for(int j = 1; j <= n; j++)
            {
                scanf("%d", &linkma[i][j]); //输入时 i j 和 j i 都要赋值
               // if(g[i][j] == 0)
```

```
                //   g[i][j] = INF;    //注意 0 的地方置为 INF
            }
        Prim();//调用
        int ans = 0;//权值和
        for(int i = 1; i <= n; i++)
            ans += vx[i].edge;
        printf("%d\n", ans);
    }
    return 0;
}
```

SPFA

(SPFA 无法处理带负环的图)

```
const int SIZE = 100010;
const int INF = 0x3f3f3f3f;
int u[4*SIZE], w[4*SIZE], v[4*SIZE], next[4*SIZE];   //一般是 4*SIZE
int first[SIZE], d[SIZE];
int sum[SIZE];
int n, m;    //n 个顶点，m 条边。
memset(first, -1, sizeof(first));
void read_graph(int u1, int v1, int w1)
{
    for(int e = 0; e < m; e++  )
    {
        scanf("%d%d%d", &u[e], &v[e], &w[e]);
        next[e] = first[u[e]];
        first[u[e]] = e;
    }
}
//队列
int spfa(int src)
{
    queue<int> q;
    bool inq[SIZE] = {0};
    for(int i = 0; i <= n; i++) d[i] = (i == src)? 0:INF;
    q.push(src);
    while(!q.empty())
    {
        int x = q.front(); q.pop();
        inq[x] = 0;
        for(int e = first[x]; e!=-1; e = next[e]) if(d[v[e]] > d[x]+w[e])
        {
            d[v[e]] = d[x]+w[e];
```

```
            if(!inq[v[e]])
            {
                inq[v[e]] = 1;
                if(++sum[v[e]] > n)   //判断负环
                {
                    return -1;
                }
                q.push(v[e]);
            }
        }
    }
    if(d[n] == INF) return -2;
    else return d[n];
}
```

利用 spfa 算法判断负环有两种方法：
spfa 的 bfs 形式，判断条件是存在一点入队次数大于总顶点数。

拓扑排序

对一个有向无环图(Directed Acyclic Graph 简称 DAG)G 进行拓扑排序，是将 G 中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若边(u,v)∈E(G)，则 u 在线性序列中出现在 v 之前。

Kahn 算法

```
int indegree[100];
int n,m;
int map[100][100];
int a[100];
int topu()
{
    queue<int> q;
    int cnt = 1;
    while(!q.empty())//清空队列
        q.pop();
    for(int i = 1; i <= n ; i++)
        if(indegree[i] == 0)
            q.push(i);//将 没有依赖顶点的节点入队
    int u;
    while(!q.empty())  //
    {
        u = q.front();
        a[cnt++] = u;//将上边选出的没有依赖顶点的节点加入到排序结果中
        q.pop();//删除队顶元素
        for(int i = 1; i <= n ; i++)
        {
            if(map[u][i])
```

```
            {
                indegree[i] --;//删去以 u 为顶点的边
                if(indegree[i] == 0) //如果节点 i 的所有依赖顶点连接边都已经删去
                    q.push(i);  //即变为无依赖顶点的节点    将其入队
            }
        }
        if(cnt == n)//如果排序完成输出结果
        {
            for(int i = 1 ; i <= n ; i++)
                printf(" %d",a[i]);
        }
    }
}
```

Floyd

```
//其实很好写 注意三重循环 k 在最外，初始化所有距离 inf 然后输入即可
int  floyd()
{
    int res=inf;
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            if(mp[i][k]!=inf)
            for(int j=1;j<=n;j++)
            {
                if(mp[i][j]>mp[i][k]+mp[k][j])
                    mp[i][j]=mp[i][k]+mp[k][j];
                if(near[i]&&goal[j])
                    res=min(res,mp[i][j]);
            }
    return res;
}
```

欧拉路径与欧拉回路

欧拉路径：可以将图中所有边一笔画走完的一条路径
欧拉回路：欧拉路径构成的回路（即欧拉路径起点终点相同）
存在条件：
欧拉回路：所有点的满足：入度 = 出度，那么沿任意一点为起点，一定存在方案形成欧拉回路。
欧拉路径：有仅有一个起点：入度 = 出度 - 1，有且仅有一个终点：出度 = 入度 - 1,。其他的点都满足：入度 = 出度。

数学

素数

筛法求素数（O(NlgN)）

```
const long N = 200000;
long prime[N] = {0},num_prime = 0;
//prime[n] store prime number ;num_prime is the number of prime;
int isNotPrime[N] = {1, 1};
        for(long i = 2 ; i < N ; i ++)
        {
         if(! isNotPrime[i])
             prime[num_prime ++]=i;
    for(long j = 0 ; j < num_prime && i * prime[j] <  N ; j ++)
         {
                 isNotPrime[i * prime[j]] = 1;
                 if( !(i % prime[j] ) ) //select prime[j]<i;
                 break;
           }
        }
```

Meisell-Lehmer O(N^(2/3))

```
//Meisell-Lehmer  模板   求 1e11 内素数个数
const int N = 5e6 + 2;
bool np[N];
int prime[N], pi[N];
int getprime()
{
    int cnt = 0;
    np[0] = np[1] = true;
    pi[0] = pi[1] = 0;
    for(int i = 2; i < N; ++i)
    {
        if(!np[i]) prime[++cnt] = i;
        pi[i] = cnt;
        for(int j = 1; j <= cnt && i * prime[j] < N; ++j)
        {
            np[i * prime[j]] = true;
            if(i % prime[j] == 0)   break;
        }
```

```
    }
    return cnt;
}
const int M = 7;
const int PM = 2 * 3 * 5 * 7 * 11 * 13 * 17;
int phi[PM + 1][M + 1], sz[M + 1];
void init()
{
    getprime();
    sz[0] = 1;
    for(int i = 0; i <= PM; ++i)  phi[i][0] = i;
    for(int i = 1; i <= M; ++i)
    {
        sz[i] = prime[i] * sz[i - 1];
        for(int j = 1; j <= PM; ++j) phi[j][i] = phi[j][i - 1] - phi[j /
prime[i]][i - 1];
    }
}
int sqrt2(LL x)
{
    LL r = (LL)sqrt(x - 0.1);
    while(r * r <= x)   ++r;
    return int(r - 1);
}
int sqrt3(LL x)
{
    LL r = (LL)cbrt(x - 0.1);
    while(r * r * r <= x)   ++r;
    return int(r - 1);
}
LL getphi(LL x, int s)
{
    if(s == 0)  return x;
    if(s <= M)  return phi[x % sz[s]][s] + (x / sz[s]) * phi[sz[s]][s];
    if(x <= prime[s]*prime[s])   return pi[x] - s + 1;
    if(x <= prime[s]*prime[s]*prime[s] && x < N)
    {
        int s2x = pi[sqrt2(x)];
        LL ans = pi[x] - (s2x + s - 2) * (s2x - s + 1) / 2;
        for(int i = s + 1; i <= s2x; ++i) ans += pi[x / prime[i]];
        return ans;
    }
    return getphi(x, s - 1) - getphi(x / prime[s], s - 1);
}
```

```
LL getpi(LL x)
{
    if(x < N)   return pi[x];
    LL ans = getphi(x, pi[sqrt3(x)]) + pi[sqrt3(x)] - 1;
    for(int i = pi[sqrt3(x)] + 1, ed = pi[sqrt2(x)]; i <= ed; ++i) ans -= getpi(x /
prime[i]) - i + 1;
    return ans;
}
LL lehmer_pi(LL x)
{
    if(x < N)   return pi[x];
    int a = (int)lehmer_pi(sqrt2(sqrt2(x)));
    int b = (int)lehmer_pi(sqrt2(x));
    int c = (int)lehmer_pi(sqrt3(x));
    LL sum = getphi(x, a) +(LL)(b + a - 2) * (b - a + 1) / 2;
    for (int i = a + 1; i <= b; i++)
    {
        LL w = x / prime[i];
        sum -= lehmer_pi(w);
        if (i > c) continue;
        LL lim = lehmer_pi(sqrt2(w));
        for (int j = i; j <= lim; j++) sum -= lehmer_pi(w / prime[j]) - (j - 1);
    }
    return sum;
}
int main()
{
    init();
    LL n=1e11;
    while(~scanf("%lld",&n))
    {
        printf("%lld\n",lehmer_pi(n));
    }
    return 0;
}
```

算数基本定理

性质：

- N的标准素因子分解表达式：$N = p_1^{a_1} p_2^{a_2} \ldots\ldots p_n^{a_n}$
- N的约数的个数：$(a_1+1)(a_2+1)\ldots\ldots(a_n+1)$
- N的所有约数的和：

  why？？

  $(1+p_1+p_1^2+\ldots\ldots+p_1^{a_1})\ldots\ldots(1+p_n+p_n^2+\ldots\ldots+p_n^{a_n})$

  等比数列求和 ？？？
- n! 的素因子分解中的素数P的幂为：$n/p+n/p^2+n/p^3$.....

分解质因数

```
vector <int> prime_factor
for(int i=2;i<=sqrt(m)+0.5;i++)
{
    if(m%i==0)
        prime_factor.push_back(i);
    while(m%i==0)
        m/=i;
}
if(m!=1) prime_factor.push_back(m);
```

找出所有约数

```
vector <int> div;
for(int i=1;i<=sqrt(m)+0.5;i++)
{
    if(m%i==0)
       {
        div.push_back(i);
        if(m/i!=i)
            div.push_back(m/i);
       }
.}
```

gcd 相关

```
  int gcd(int x , int y) //当 1 方为 0 返回另一方
  {
  if(!y)
  return x;
  else return gcd(y , x%y);
  }
  int lcm(int x,int y) //最小公倍数
  {
      return x/gcd(x,y)*y;
  }
  //计算 a*x+b*y=gcd(a,b)的一组解 x,y 返回 gcd(a,b)
  int exgcd(int a , int b,int &x,int &y)
  {
      if(b==0)
      {
          x=1;y=0;
```

```
            return a;
        }
        int d=exgcd(b,a%b,x,y);
        int tem=x;
        x=y;
        y=tem-a/b*y;
           return d;
       }
```

a*x+b*y=c 有解充要条件 c%gcd(a,b)==0；

逆元

```
///单变元模线性方程(求逆元)
//O (lgN) 已知 a,b,n; 输出所有[0,n)中满足 ax=b(mod n)的解
vector<long long>line_mod_equation(long long a,long long b,long long n)
{
    long long x,y;
    long long d=exgcd(a,n,x,y);///use exgcd
    vector<long long>ans;
    ans.clear();
    if(b%d==0){
        x%=n;x+=n;x%=n;
        ans.push_back(x*(b/d)%(n/d));
        for(long long i=1;i<d;i++)
            ans.push_back((ans[0]+i*n/d)%n);
    }
    return ans;
}
```

中国剩余定理（CRT）

```
//O(NlgM) M 和每个 mi 同阶 x=ai(mod mi)  即 x%mi=ai
int CRT(int a[],int m[],int n)
{
    int M=1;
    for(int i=0;i<n;i++) M*=m[i];
    int ret=0;
    for(int i=0;i<n;i++)
    {
        int x,y;
        int tm=M/m[i];
        exgcd(tm,m[i],x,y);
        ret=(ret+tm*x*a[i])%M;
    }
```

```
    return (ret+M)%M;
.}
```

容斥原理

求 1-r（inclusive r）中有多少个数与 n 互素

```
LL Solve(LL n,LL r)
{
    vector<LL> p;
    for(LL i=2; i*i<=n; i++)
    {
        if(n%i==0)
        {
            p.push_back(i);
            while(n%i==0) n/=i;
        }
    }
    if(n>1)
        p.push_back(n);
    LL ans=0;
    for(LL msk=1; msk<(1<<p.size()); msk++)
    {
        LL multi=1,bits=0;
        for(LL i=0; i<p.size(); i++)
        {
            if(msk&(1<<i))  //判断第几个因子目前被用到
            {
                ++bits;//判断有几位组成
                multi*=p[i];
            }
        }
        LL cur=r/multi;
        if(bits&1) ans+=cur;//奇数加
        else       ans-=cur;//偶数减
    }
    return r-ans;
}
```

欧拉函数

令 n=p^k, 小于 n 的正整数数共有 n-1 即(p^k-1)个, 其中与 p 不质的数共[p^(k-1)-1]个(分别为 1*p, 2*p, 3*p...p(p^(k-1)-1))。
所以 E(p^k)=(p^k-1)-(p^(k-1)-1)=p^k-p^(k-1).得证。
欧拉公式计算时 先让 x/p1 *(p1-1)..... 计算适用于 10^5 数量级

欧拉函数：对于一个正整数n,小于n且和n互质的正整数(包括1)的个数,记作φ(n).

若p是素数,则φ(p)=p-1;

特殊的:φ(1)=1;

若m,n互素,则φ(mn)=φ(m)φ(n)(积性函数)

若p是素数,n=p^k,则 $\varphi(n)=(p-1)p^{k-1}=p^k-p^{k-1}$

欧拉函数通式: $\varphi(x)=x(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots\cdots(1-\frac{1}{p_n})$

其中p1,p2.....pn为x的所有质因数,x是不为0的整数

```
ll oula(ll n)
{
    ll replaced=n;
    vector<ll> v;
    for(ll i=2;i*i<=n;i++)
    {
        if(n%i==0)
            v.push_back(i);
        while(n%i==0)
            n/=i;
    }
    if(n!=1) v.push_back(n);
    ll ans=replaced;
    ll number=v.size();
    for(ll i=0;i<number;i++)
        ans=ans/v[i]*(v[i]-1);
    return ans;
}
```

线性欧拉筛

```
const long long  N = 1000006;
long long oulaf[N]={0,0};//存每个数的欧拉函数 1 除外 输入 1 直接输出 1
long long prime[N] = {0},num_prime = 0;
//prime[n] store prime number ;num_prime is the number of prime;
int isNotPrime[N] = {1, 1};
void oula()
{
      for(long long i = 2 ; i < N ; i ++)
       {
       if(! isNotPrime[i])
           {prime[num_prime ++]=i;oulaf[i]=i-1;}
for(long long j = 0 ; j < num_prime && i * prime[j] <  N ; j ++)
       {
              isNotPrime[i * prime[j]] = 1;
              if(i%prime[j]!=0)
```

```
                oulaf[i*prime[j]]=(prime[j]-1)*oulaf[i];
                else //select prime[j]<i;
                {oulaf[i*prime[j]]=oulaf[i]*prime[j];break;}
            }
        }
    return ;
.}
```

组合数

```
long long Combination(long long n,long long m)
{
    if(m>n) return 0;
    long long ans=1;
    if(m>n-m) m=n-m;
    for(long long i=n;i>n-m;i--)
        ans=(ans*i);
    for(long long i=2;i<=m;i++)
    {
        ans/=i;
    }
    return ans;
}
```

杨辉三角预处理

```
int cc[1000][1000];
for(int i=2;i<=1000;i++)
    {cc[i][0]=1;cc[i][i]=0;cc[i][1]=i;}
    for(int i=3;i<=1000;i++)
        for(int j=2;j<i;j++)
    {
        cc[i][j]=(cc[i-1][j]+cc[i-1][j-1]);
    }
const  int mod =1000003;
    long long f[1000006];//需要变
    ll PowerMod(ll a, ll b, ll c)
    {
       ll ans = 1;
    a = a % c;
   while(b>0)
    {
    if(b % 2 == 1)
    ans = (ans * a) % c;
    b = b/2;
    a = (a * a) % c;
```

```
    }
    return ans;
    }
```

阶乘预处理

```
long long Combination(long long n,long long m,long long mod)
    {
    if(m==0) return 1;
    if(m>n) return 0;
    long long ans;
    return ans=f[n]*PowerMod(f[m],mod-2,mod)*PowerMod(f[n-m],mod-2,mod)%mod;
    }
    void fact(long long n)  //这个要在主函数调用
    {
    f[0]=1;
    for(int i=1;i<n;i++)
        f[i]=f[i-1]*i%mod;
    }
```

一般和次方取模 预处理数组板子一起用　额外使用 lucas 可以提升 15%左右速度

```
int Combination(ll n,ll m,ll p)//这个是次数较少时使用
{
 ll a = 1,b = 1;
 if(m > n) return 0;
 while(m)
 {
     a = (a * n) % p;
     b = (b * m) % p;
     m--;
     n--;
 }
 return ((ll)a * (ll)PowerMod(b,p-2,p))%p;
}
 ll lucas(ll n,ll m,ll mod)
 {
        if(m==0) return 1;
    ll ans=Combination(n%mod,m%mod,mod)*lucas(n/mod,m/mod,mod)%mod;
         return ans;
 }
```

组合数的性质

方程 x1+x2+...+xn = m 的解的个数，利用插板法可以得到方案数为
C(m+n-1,n-1)　而　C(m+n-1,m)=C(m+n-1,n-1)
C(n,k) = C(n-1,k)+C(n-1,k-1)　　所以
C(n-1,0)+C(n,1)+...+C(n+m-1,m)

= C(n,0)+C(n,1)+C(n+1,2)+...+C(n+m-1,m)
= C(n+m,m)

斐波那契数列

O (N)算法

```
int main()
{
        int n;
        long long add;
        while(scanf("%d", &n) != EOF)
        {
            long long f[2] = {0, 1};
            if(n < 0) printf("-1\n");
            else if(n < 2) printf("%d\n", f[n]);
            else {
                for(int i = 2; i <= n; ++i)
                {
                    add = f[0] + f[1];
                    f[0] = f[1];
                    f[1] = add;
                }
                //printf("%ld\n", add);
                cout<<add<<endl;
            }
        }
        return 0;
    }
```

O lg（N）算法

即结合矩阵快速幂

```
void myMultiply(long long (&f1)[4], long long (&f2)[4])
{
    long long res[4];
    for(int i = 0; i < 2; ++i)
        for(int j = 0; j < 2; ++j)
        {
            int idx = 2 * i + j;
            res[idx] = 0;
            for(int k = 0; k < 2; ++k)
                res[idx] += f1[2*i+k] * f2[2*k+j];
        }
    for(int i = 0; i < 4; ++i)
```

```
        f1[i] = res[i];
}
int main()
{
    int n;
    while(scanf("%d", &n) != EOF)
    {
        long long f[4] = {1, 1, 1, 0};
        if(n < 0) printf("-1\n");
        else if(n == 0) printf("0\n");
        else if(n == 1) printf("1\n");
        else
        {
            long long res[4] = {1, 0, 0, 1};
            n = n - 1;
            while(n > 0)
            {
                if(n & 1)
                {
                    myMultiply(res, f);
                }
                myMultiply(f, f);
                n >>= 1;
            }
            printf("%lld\n", res[0]);
        }
    }
    return 0;
}
```

卡特兰数

Catalan number　　其前几项（从第 0 项开始）为 ： 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, ...

递推公式 h(n)=h(n-1)*(4*n-2)/(n+1);

h(n)=C(2n,n)/(n+1) (n=0,1,2,...) = $\frac{(2n)!}{(n+1)!n!}$

主要应用：矩阵链乘的括号、凸多边形的三角形划分

勾股数组

对于一个奇数 x：有 x，x^2/2，x^2/2+1 构成一组勾股数（x>=3）

任意偶数 2n(n > 1)都构组勾股数三边别：2*n、n^2-1、n^2+1

数据结构

树状数组

```
//一维
int lowbit(int x)//计算 lowbit
{
    return x&(-x);
}
 void add(int i,int val)//将第 i 个元素增加 val 注意 i 不可为零
 {
    while(i<=n)//n 为所开数组上界
    {
        c[i]+=val;
        i+=lowbit(i);
    }
 }
 int sum(int i)//求前 i 项和
 {
    int s=0;
    while(i>0)
    {
        s+=c[i];
        i-=lowbit(i);
    }
    return s;
 }
//可升级为 2 维及多维：修改 2 维矩阵某个值，求区间[x1,y1]~[x2,y2]的和代码
如下：
int lowbit(int x)
 {
  return x&(-x);
 }
void add(int x,int y,int det)
 {
  int i,j;
  for(i=x;i<=n;i+=lowbit(i))
   for(j=y;j<=n;j+=lowbit(j))
    c[i][j]+=det;
 }
```

```
int getsum(int x,int y)
 {
  int i,j,sum=0;
  for(i=x;i>0;i-=lowbit(i))
   for(j=y;j>0;j-=lowbit(j))
    sum+=c[i][j];
  return sum;
 }
```

常见应用：区间求和，求逆序对，求真包含区间个数

求逆序对：

若数组元素值数据量过大 离散化处理。 将数组按下表从小到大排列，元素也是从小到大排列

i 循环 1—n 对于每个 a[i] 存入树状数组 此时逆序对数=i-sum（a[i]）；

总的逆序对数为所有的和

真包含区间个数：

自己包含的个数求法：x 从小到大排 y 从小到大

包含自己的区间个数：x 从小到大 y 从大到小

都需要判断是否和前一个区间完全相同

线段树

```
const int MXN = 100000 + 10;
struct SEG {
        int l, r, m, lazy;//m=value
        SEG( int _l = 0, int _r = 0 ) {l = _l, r = _r;}

} SGT[MXN << 2 | 1];
void create( SEG &T, int t ) {
        T.m = 0;//m 亦可以赋其他值
        T.lazy = 0;
}
void fresh( SEG &T, SEG &L, SEG &R ) {

}
void build( int id, int l, int r ) {//对结点 id 进行建立，区间为 l~r
        SGT[id] = SEG( l, r );
        int mid = ( l + r ) / 2;
        if ( l != r ) {
         build( id * 2, l, mid );
         build( id * 2 + 1, mid + 1, r );
         fresh( SGT[id], SGT[id << 1], SGT[id << 1 | 1] );
        } else create( SGT[id], l );//进行结点的初始化
    //SGT[id].m=SGT[l].m+SGT[r].m;//根据左右子节点的值来更新当前节点
}
void update( int id, int l, int r, int w ) {//根 id 在区间 l~r 上更新 m 的状态 w
```

```
        SEG &T = SGT[id];
        int mid = ( T.l + T.r ) / 2;
        if ( T.l == l && T.r == r ) {//如果找到区间 l~r 则修改 m
         T.m = +=w; //这里对 m 基于 w 进行修改，不定
         return;
        }
        if ( T.lazy ) {
         update( id << 1, T.l, mid, T.lazy );//左子树根据 lazy 值进行修改
         update( id << 1 | 1, mid + 1, T.r, T.lazy );
         T.lazy = 0;
        }
if ( r <= mid )update( id << 1, l, r, w );//只对左儿子更新
else if ( l > mid )update( id << 1 | 1, l, r, w );//只对右儿子更新
else {
update( id << 1, l, mid, w );//区间横跨左右儿子区间，对其两者均进行更新
update( id << 1 | 1, mid + 1, r, w );//区间横跨左右儿子区间，对其两者均进行更新
}
fresh( T, SGT[id << 1], SGT[id << 1 | 1] );//???
}
void query( int id, SEG &A ) {
       SEG &T = SGT[id];
       if ( T.lazy ) {
        int mid = (T.l + T.r) / 2;
        update( id << 1, T.l, mid, T.lazy );
        update( id << 1 | 1, mid + 1, T.r, T.lazy );
        T.lazy = 0;
       }
       if (T.l == A.l && T.r == A.r) {
           A = T;
           //ans+=A.m;区间求和，这里将 m 作为区间和
   return;
   }
    int mid = ( T.l + T.r ) / 2;
    if ( A.r <= mid )query( id << 1, A );
    else if ( A.l > mid )query( id << 1 | 1, A );
    else {
         SEG L( A.l, mid ), R( mid + 1, A.r );
         query( id << 1, L );
        query( id << 1 | 1, R );
       fresh( A, L, R );
    }
}
```

RMQ 问题 ST 算法

RMQ 问题是指求区间最值的问题 O(nlogn)*O(q) online

```
int f[1000005];
int maxsum[100005][20];
void RMQ(int num) //预处理->O(nlogn)
{
        for(int i=1;i<=num;i++)
        maxsum[i][0]=f[i];//f[i]为被操作数组，num 为 f 长度
    int k=log(double(num+1))/log(2.0);
        for(int j = 1; j <= k; ++j)
         for(int i = 1;i+(1<<j)-1<=num;++i)
                 maxsum[i][j]=max(maxsum[i][j-1], maxsum[i+(1<<(j-1))][j-1]);
}
int query(int l,int r)
{
    if(l>r) return 0;
    int k=floor(log2(r-l+1));
    return max(maxsum[l][k],maxsum[r-(1<<k)+1][k]);
}
```

并查集

```
int pre[1024];
int rank[1024];
int findfater(int x)
{
    int r=x;
    while(pre[r]!=r)
        r=pre[r];
    int i=x,j;
    while(i!=r)//路径压缩
    {
        j=pre[i];
        pre[i]=r;
        i=j;
    }
    return r;
}
void join(int x,int y)
{
    int fx=findfater(x),fy=findfater(y);
    if(fx==fy) return;
    if(rank[fx]<=rank[fy]){   //按秩合并
        pre[fx]=fy;
        if(rank[fx]==rank[fy])
```

```
            rank[fy]++;
    }else{
    pre[fy]=fx;
    }
}
void initial(int n)
{
    for(int i=1;i<=n;i++)
        pre[i]=i;
}
```

计算几何

三点顺序（判断给出三点是按顺/逆时针给出）：
利用矢量叉积判断是逆时针还是顺时针，
P × Q = （x1*y2 - x2*y1）k，若 P × Q > 0，则 P 在 Q 的顺时针方向。若 P × Q < 0，则 P 在 Q 的逆时针方向。若 P × Q = 0，则 P 与 Q 共线，但可能同向也可能反向。

```
/// 判断大小
const double eps=1e-8;
int cmp(double x)
{
    if(fabs(x)<eps) return 0;
    if(x>0) return 1;
    return -1;
}
///点-向量
const double pi=acos(-1.0);
inline double sqr(double x){
    return x*x;
}
struct point{
    double x,y;
    point(){}
    point(double a,double b):x(a),y(b){}
    void input(){
    scanf("%lf %lf",&x,&y);
    }
    friend point operator +(const point &a,const point &b){
        return point(a.x+b.x,a.y+b.y);
    }
    friend point operator -(const point &a,const point &b){
        return point(a.x-b.x,a.y-b.y);
```

```
    }
    friend bool operator ==(const point &a,const point &b){
        return cmp(a.x-b.x)==0&&cmp(a.y-b.y)==0;
    }
    friend point operator *(const double &a,const point &b){//将向量 b 放大 a 倍
        return point(a*b.x,a*b.y);
    }
    friend point operator *(const point &a,const double &b){//将向量 a 放大 b 倍
        return point(a.x*b,a.y*b);
    }
    friend point operator /(const point &a,const double &b){//将向量 a 放大 b 倍
        return point(a.x/b,a.y/b);
    }
    double norm(){//求向量模长
        return sqrt(sqr(x)+sqr(y));
    }
};
double det(const point &a,const point &b){//叉积
    return a.x*b.y-a.y*b.x;
}
double dot(const point &a,const point &b){//点积
    return a.x*b.x+a.y*b.y;
}
double dist(const point &a,const point &b){//距离
    return (a-b).norm();
}
point rotate_point(const point &p,double A){//op 绕原点逆时针转 A(弧度)
    double tx=p.x,ty=p.y;
    return point(tx*cos(A)-ty*sin(A),tx*sin(A)+ty*cos(A));
}
///线段
struct line{
    point a,b;//线段 a->b
    line(){}
    line(point x,point y):a(x),b(y){}
};
line point_make_line(const point a,const point b){//2 点确定 1 条直线
    return line (a,b);
}
double dis_point_segment(const point p,const point s,const point t){
```

```
    if(cmp(dot(p-s,t-s))<0) return (p-s).norm();//点 p 到线段 st 的距离
(不是直线)
    if(cmp(dot(p-s,s-t))<0) return (p-t).norm();
    return fabs(det(s-p,t-p)/dist(s,t));//p 到直线 st 距离,利用面积计算.
}
void PointProjLine(const point p,const point s,const point t,point
&cp){
    double r=dot((t-s),(p-s))/dot(t-s,t-s); //求 p 到 st 的垂足，保存在 cp
中
    cp=s+r*(t-s);
}
bool PointOnSegment(point p,point s,point t){
    return cmp(det(p-s,t-s))==0&&cmp(dot(p-s,p-t))<=0;
}
bool parallel(line a,line b){
    return !cmp(det(a.a-a.b,b.a-b.b));
}
bool line_make_point(line a,line b,point &res){//求 2 直线交点
    if(parallel(a,b)) return false;
    double s1=det(a.a-b.a,b.b-b.a);
    double s2=det(a.b-b.a,b.b-b.a);
    res=(s1*a.b-s2*a.a)/(s1-s2);//待证明公式.
    //if((res-a.a).norm()<(a.a-a.b).norm()&&(res-b.a).norm()<(b.a-
b.b).norm()
       &&(res-a.b).norm()<(a.a-a.b).norm()&&(res-b.b).norm()<(b.a-
b.b).norm())
    return true;
    //else return false; //加上 if else 判断 2 线段相交
}
line move_d(line a,const double &len){//将直线 a 沿法向量方向平移距离 len
    point d=a.b-a.a;
    d=d/d.norm();
    d=rotate_point(d,pi/2);//默认向量法向量逆时针转 90°
    return line(a.a+d*len,a.b+d*len);
}
```

动态规划

LIS

O(n^2)。

```
for(i=2; i<=N; i++)
        {
            for(j=1; j<i; j++)
```

```
            {
                if(A[i]>A[j] && D[i]<=D[j])
                {
                    D[i]=D[j]+1;
                }
            }
            D[i]=max(1,D[i]); //必须判断一次，如果不比 A[j]大，则 D[i]=1;
        }
```

0 n*logn

```
d[1]=a[1];
    int len=1,l,r,mid;
    for(int i=2;i<=n;i++)
    {
        if(d[len]<a[i])
            {
                d[++len]=a[i];
                continue;
            }
            l=1,r=len;
            while (l<=r)
            {
                mid=(l+(r-l)/2);
                if(d[mid]<a[i])
                    l=mid+1;
                else     r=mid-1;
            } //r=mid-1,l=mid+1,为了跳出循环
            d[l]=a[i];
    }
    printf("%d\n",len);
```

LCS

$$num[i][j]=\begin{cases}0 & i=0或者j=0\\ 1+num[i-1][j-1] & i,j>0,a[i]=b[j]\\ \max\{num[i][j-1],num[i-1][j]\} & i,j>0,a[i]\neq b[j]\end{cases}$$

最大子段和

0（n）

以 dp[i]表示以 i 结尾最大子段和，最后遍历求最大

1 如果 dp[i-1]>0,无论 ai 为何值，有 dp[i]=dp[i-1]+ai;

2 如果 dp[i-1]<=0;舍弃，重新令 dp[i]=ai;(因为 dp[i-1]为负数无论 ai 为什么值加上去都会减少)

状态转移方程：dp[i]=dp[i-1]+ai (dp[i-1]>0)

dp[i]=ai(dp[i-1]<=0)

一般求出前缀和，可以以 O （n^2）解出

背包问题

初始化的细节问题 我们看到的求最优解的背包问题题目中,事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解,有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。如果是第一种问法,要求恰好装满背包,那么在初始化时除了 f[0]为 0 其它 f[1..V]均设为-∞,这样就可以保证最终得到的 f[N]是一种恰好装满背包的最优解。如果并没有要求必须把背包装满,而是只希望价格尽量大,初始化时应该将 f[0..V] 全部设为 0。
为什么呢?可以这样理解:初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满,那么此时只有容量为 0 的背包可能被价值为 0 的 nothing “恰好装满”,其它容量的背包均没有合法的解,属于未定义的状态,它们的值就都应该是-∞了。如果背包并非必须被装满,那么任何容量的背包都有一个合法解“什么都不装”,这个解的价值为 0,所以初始时状态的值也就全部为 0 了。

01 背包

dp[i][j]表示第 i 件物品放入容量为 j 的背包所得的最大价值
dp[i][j]=max{dp[i-1][j-v[i]]+c[i],dp[i-1][j]};
由于 i 1->n 可转换为 1 维 dp[j]=max{dp[j],dp[j-v[i]]+c[i]};

```
for（i=1;i<=n;i++）{
      for(j=V;j>=0;j--){//这里 j 一定要从 v->0,因为从 2 维到 1 维 若从 1—>v dp[j-v[i]]表示 dp[i][j-v[i]] 而不是 dp[i-1][j-v[i]]
          if(j>=v[i])
            dp[j]=max{dp[j],dp[j-v[i]]+c[i]};
      }
}
dp[v]即为最大的价值
```

完全背包

```
for（i=1;i<=n;i++）{
   for(j=v[i];j<=V;j++){//注意这里是从 v[i]开始到 V， 因为可以放第 i 件物品多个， dp[j-v[i]]表示 dp[i][j-v[i]]这里 j>v[i]之后考虑的是放入第 i 件物品后的最优值， 符合题意
        if(j>=v[i])
          dp[j]=max{dp[j],dp[j-v[i]]+c[i]};
   }
 }
//注意这列求出的 dp[v]是最大的因为一直叠加
```

一个简单有效的优化 完全背包问题有一个很简单有效的优化,是这样的:若两件物品 i,j 满足 c[i] <= c[j]且 w[i] >= w[j],则将物品 j 去掉,不用考虑。

多重背包

第 i 种物品最多有 n[i]件可用;
dp[i][j]=max{dp[i-1][v-k*v[i]]+k*c[i]|0<=k<=n[i]}(k 表示第 i 种物品放入 k 件)

```
for(i=1;i<=n;i++){
    for(j=v;j>=0;j--){
        for(k=1;k<=n[i];k++){
           if(j>=k*v[i])
            dp[i][j]=max(dp[i-1][v-k*v[i]]+k*c[i], dp[i-1][j]))
           }
      }
}
```

多重背包问题 转化为 完全背包问题（复杂度优化），也就是每个物品都有无限个，但是在循环过程中用一个数组记录 ：某种硬币 i 使用的次数，如果使用次数超过 c [i]，则停止循环
例题 poj1742

混合 3 种背包

01 背包与完全背包的混合考虑:一类物品只能取一次,另一类物品可以取无限次,那么只需在对每个物品应用转移方程时,根据物品的类别选用顺序或逆序的循环即可,复杂度是 Θ(V N)

```
for i ← 1 to N
do if 第 i 件物品属于 01 背包
then for v ← V to 0
do f [v] = max{f [v], f [v - c[i]] + w[i]}
else if 第 i 件物品属于完全背包
then for v ← 0 to V
do f [v] = maxf [v], f [v - c[i]] + w[i]
```

加上完全背包只是再换个方程

二维费用背包

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价 1 和代价 2，第 i 件物品所需的两种代价分别为 a[i]和 b[i]。两种代价可付出的最大值（两种背包容量）分别为 V 和 U。物品的价值为 w[i]；
费用加了一维，只需状态也加一维即可。设 f[i][v][u]表示前 i 件物品付出两种代价分别为 v 和 u 时可获得的最大价值。状态转移方程就是：
f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}

```
for(int i=1;i<=g;i++)
    for(int j=15000;j>=1;j--)
```

```
      {
          for(int k=0;k<c;k++)
          {
        if(j-w[i]*pos[k]>=0&&j-w[i]*pos[k]<=15000&&dp[i-1][j-w[i]*pos[k]])
            dp[i][j]+=dp[i-1][j-w[i]*pos[k]];
          }
        }//这里求总数所以+=
```

物品总个数的限制

有时，“二维费用”的条件是以这样一种隐含的方式给出的:最多只能取 M 件物品。这事实上相当于每件物品多了一种“件数”的费用,每个物品的件数费用均为 1,可以付出的最大件数费用为 M。换句话说,设 f[v][m]表示付出费用 v、最多选 m 件时可得到的最大价值,则根据物品的类型(01、完全、多重)用不同的方法循环更新,最后在 f[0..V][0..M]范围内寻找答案。

多维背包

类似二维费用，不过可以只消耗一种费用。解法开多维，多个 for 循环。
与 for 循环的几个费用的顺序无关 但注意要开临时变量不行的话 加一维

```
for(i = 1; i<=n; i++){
   for(x = k; x>=0; x--){
     for(y = v1; y>=0; y--){
      for(z = v2; z>=0; z--){
      int tem = 0;
       if(x-1>=0)
       tem = max(tem,dp[x-1][y][z]+s[i].w);
     if(y-s[i].a>=0)
     tem = max(tem,dp[x][y-s[i].a][z]+s[i].w);
    if(z-s[i].b>=0)
     tem = max(tem,dp[x][y][z-s[i].b]+s[i].w);
     dp[x][y][z] = max(dp[x][y][z],tem);
                  }
              }
           }
        }
```

分组背包

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。
这个问题变成了每组物品有若干种策略：是选择本组的某一件，还是一件都不选。也就是说设 f[k][v]表示前 k 组物品花费费用 v 能取得的最大权值，则有：

f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品 i 属于组 k}

使用一维数组的伪代码如下：

```
for 所有的组 k
    for v=V..0
        for 所有的 i 属于组 k
            f[v]=max{f[v],f[v-c[i]]+w[i]}
```

注意这里的三层循环的顺序

关于背包问题的问法

如果要求的是“总价值最小”“总件数最小”,只需简单的将上面的状态转移方程中的 max 改成 min 即可。

输出方案 一般而言,背包问题是要求一个最优值,如果要求输出这个最优值的方案,可以参照一般动态规划问题输出方案的方法:记录下每个状态的最优值是由状态转移方程的哪一项推出来的,换句话说,记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态,从上一个状态接着向前推即可。

还是以 01 背包为例,方程为 f [i][v] = max{f [i-1][v], f [i-1][v-c[i]]+w[i]}。再用一个数组 g[i][v],

设 g[i][v]=0 表示推出 f[i][v]的值时是采用了方程的前一项(也即 f [i][v] = f [i - 1][v]),

g[i][v]=1 表示采用了方程的后一项。注意这两项分别表示了两种策略:未选第 i 个物品及选了第 i 个物品。那么输出方案的伪代码可以这样写(设最终状态为 f[N][V]):

```
i ← N
v ← V
while i > 0
    do if g[i][v] = 0
        then print" 未选第i项物品"
        else if g[i][v] = 1)
            then print" 选了第i项物品"v ← v − c[i]
```

对于一个给定了背包容量、物品费用、物品间相互关系(分组、依赖等)的背包问题,除了再给定每个物品的价值后求可得到的最大价值外,还可以得到装满背包或将背包装至某一指定容量的方案总数。对于这类改变问法的问题,一般只需将状态转移方程中的 max 改成 sum 即可。例如若每件物品均是完全背包中的物品,转移方程即为

```
f [i][v] = sum{f [i - 1][v], f [i][v - c[i]]}
```

初始条件 f[0][0]=1

事实上,这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

数字三角形

递归 int d(int i,int j)

```
{
    return a[i][j]+(i==n? 0:max(d(i+1,j),d(i+1,j+1)));
}
```

记忆化搜索 int dfs(int i,int j)

```
{
    if(d[i][j]>0)
    return d[i][j];
    return d[i][j]=a[i][j]+(i==n?0:max(dfs(i+1,j),dfs(i+1,j+1)));
```

}

递推

```
for(int i=1;i<=n;i++)
   for(int j=1;j<=i;j++)
           scanf("%d",&a[i][j]);
for(int i=1;i<=n;i++)
   d[n][i]=a[n][i];
for(int i=n-1;i>=1;i--)
   for(int j=1;j<=i;j++)
           d[i][j]=max(d[i+1][j],d[i+1][j+1])+a[i][j];
printf("%d\n",d[1][1])
```

区间 dp

```
for(int p = 1 ; p <= n ; p++){//p 是区间的长度，阶段
   for(int i = 1 ; i <= n-p ; i++){//i 是穷举区间的起点
        int j = i+p;//j 为区间的终点
    for(int k = i+1 ; k < j ; k++)//状态转移
       dp[i][j] = min{dp[i][k]+dp[k+1][j]+w[i][j]};//这个是看具体的状态转移方程
或
        dp[i][j] = max{dp[i][k]+dp[k+1][j]+w[i][j]};//求最大
  或  dp[i][j] = min{dp[i][k]+dp[k][j]+w[i][j]}//有的是要从 k 开始不是 k+1
         }
}
```

状态压缩 dp

状态中的某一维会比较小，一般不会超过 15,

数位 dp

数位 dp 的特点是 给一个 n 问 1-n 有多少个数满足性质 x
一般 问法：1 -N 中不含 49 （或者 62）的数有多少个
dp[i][j] 表示长度为 i 的数（也就是有 i 位数）状态为 j 的数的总数有多少
本题状态有三种：
①dp[i][0]代表长度为 i 且不包含 49 的数有多少个
②dp[i][1]代表长度为 i 且不包含 49 且左边第一位（最高位）为 9 的数有多少个
③dp[i][2]代表长度为 i 且包含 49 的数有多少个
打表预处理，0<=i<=21（21 位就够了）,主要是处理状态的转移
dp[i][0]=dp[i-1][0]*10-dp[i-1][1];//dp[i][0]高位随便加一个数字都可以，但是会出现 49XXX 的情况，要减去
dp[i][1]=dp[i-1][0]; //在不含 49 的情况下高位加 9

dp[i][2]=dp[i-1][2]*10+dp[i-1][1]; //在含有 49 的情况下高位随便加一位或者不含 49 但高位是 9，在前面最高位加上 4 就可以了

```
long long dp[22][3];
int bit[22];//注意 bit 为倒序排列 n eg: n=1234;bit=4321;
void init()
{
    dp[0][0]=1;//长度为 0 赋值为 1 为了使长度为 1 时 dp[1][0]=10;dp[1][1]=1;dp[1][2]=0;
    dp[0][1]=0;dp[0][2]=0;
    for(int i=1;i<=21;i++)
    {
        dp[i][0]=dp[i-1][0]*10-dp[i-1][1];
        dp[i][1]=dp[i-1][0];
        dp[i][2]=dp[i-1][2]*10+dp[i-1][1];
    }
}
long long cal(long long n)
{
    memset(bit,0,sizeof bit);
    int len=0;
    while(n)
    {
        bit[++len]=n%10;
        n/=10;
    }
    long long ans=0;
    bool has=false;//n 本身是否含 49
    for(int i=len;i>=1;i--)//倒序计算  使得 bit 变正序输出
    {
        ans+=dp[i-1][2]*bit[i];
        if(!has)
        {
            if(bit[i]>4)
                ans+=dp[i-1][1];
        }
        else ans+=dp[i-1][0]*bit[i];//一旦找到 49 随便选

        if(bit[i+1]==4&&bit[i]==9)
            has=true;
    }
    if(has)
            ans++;
    return ans;

}
```

```
//以 491 为例，先求出所有比 400 小的数中有多少符合题意的，然后 4 这一位确定以后，再求所有比 490 小，再求出所有比 491 小
//i=3 求出数 049 149 249 349
//i=2 求出数 449
//i=1 求出数 490
//自身包含 49 所以求出数 491
```

轮廓线 dp

n*m 的矩阵 有 2^m 种 状态 每个状态用一个 m 位 01 字符串表示后转化为 10 进制数

```
int n,m,cur;
const int maxn=11;
long long d[2][1<<maxn];//d[i][j]表示当前阶段为一点,状态为 j 时的种数，使用滚动数组
void update (int old,int news) //原来状态为 old 转移状态为 news
{
    if(news&(1<<m))        d[cur][news^(1<<m)]+=d[1-cur][old];
    //news 的从右向左(含第 0 位)第 m 位为 1  将 news 的第 m 位变成 0
{
}
int main()
{
    while(scanf("%d %d",&n,&m)==2)
    {
        if(n<m) swap(n,m);  //去 m 为较小的固定，降低时间复杂度
        memset(d,0,sizeof d);
        cur=0;
        d[0][(1<<m)-1]=1;
        for(int i=0;i<n;i++)
            for(int j=0;j<m;j++)
        {
            cur^=1;
            memset(d[cur],0,sizeof d[cur]);
            for(int k=0;k<(1<<m);k++){
                update(k,k<<1);//不放
            if(i&&!(k&(1<<(m-1)))) update(k,(k<<1)^(1<<m)^1);//新状态 m 位变 0 第 0 位变 1
            if(j&&!(k&1)) update(k,(k<<1)^3);//第 0 和 1 位变 1
            }
        }
        printf("%lld\n",d[cur][(1<<m)-1]);//  I64d  wa
    }
    return 0;
}
```

基础

Dfs

应用问题：n 皇后，合并 sticks

```
int n,m;
char a[1005][1005];
int used[1005][1005];
int ans[1005][1005];
int dx[]={0,1,0,-1};
int dy[]={1,0,-1,0};
char name[10]="DIMA";
int dfs(int i,int j)
{
    if(ans[i][j]+1>0) return ans[i][j];
    if(used[i][j]&&a[i][j]=='D')  return inf;//表示 infinite
    used[i][j]=1;
    int maxdfs=0;
    for(int idt=0;idt<4;idt++)
    {
        int temx=i+dx[idt];
        int temy=j+dy[idt];
        int id=(find(name,name+4,a[i][j])-name+1)%4;//注意到 A 需要取模
        if(temx>=0&&temx<n&&temy>=0&&temy<m&&a[temx][temy]==name[id])
        {
                maxdfs=max(maxdfs,dfs(temx,temy));
        }
    }
    if(a[i][j]=='A'&&maxdfs!=inf)
        {
              maxdfs+=1;
        }
    return  ans[i][j]= maxdfs;//0 表示没得
}
```

枚举排列

4 种方法

1 已知确定个数 m ， m 个 for 循环

2dfs

3 状态压缩

位运算

&与　　|或　　^异或　　~取反　　<<左移　　>>右移
&　　常用 x&1 判断奇偶 奇数为真 偶数为假
^　　位　　不同为 1 同为 0
~　　if（~x） 如果 x！= - 1
<<　　a<<b　　即 a*2*2*2....乘 b 个 2　　1<<x　　表示 2 的 x 次方
将第 i 位变成 0 产生新的 x
x2=x&（~（1<<i）） 或者 **x2= x-（1<<i）**
将 x 的第 i 为变成 1 产生新的 x
x2=x|(1<<i)
把 x 的第 i 位取反
x^(1<<i)
遍历点集 x 中都包含哪些点
for（j=0；(1<<j)<=x;j++）
{
if（((1<<j)&x)!=0）
　　点集 x　就包含点 j 即 j 点为 1
}
同样判断　　if((x>>i)&1)//x 的第 i 位为 1

gcc（GNU Complier Collection） 内建函数

__gcd(a,b)
__builtin_popcount() 计算一个 32 位无符号整数有多少个位为 1
__builtin_ffs(x)：返回 x 中最后一个为 1 的位是从后向前的第几位
__builtin_ctz(x)：x 末尾 0 的个数。x=0 时结果未定义
__builtin_clz(x)：x 前导 0 的个数。x=0 时结果未定义。
上面的宏中 x 都是 unsigned int 型的，如果传入 signed 或者是 char 型，会被强制转换成 unsigned int。

二分

```
while(L < R) {
int mid = (L + R) >>1;
If(check(mid)) R = mid; else L = mid + 1;
}
```

输入输出

```
#include<sstream>
string sub; int i=1;
stringstream ss;
ss<<i;
ss>>sub;
```

```
s=s+sub;                    itoa 请用 sprintf。sscanf char to int
```

离散化

思路：先排序，再删除重复元素，然后就是索引元素离散化后对应的值。
假定待离散化的序列为 a[n]，b[n]是序列 a[n]的一个副本，则对应以上三步为：

```
sort(a,a+n);
int size=unique(a,a+n)-a; //size 为离散化后元素个数
for(i=0;i<n;i++)
b[i]=lower_bound(a,a+size,b[i])-a + 1;//b[i]为经离散化后对应的值
```

映射回来就是 a[b[i]-1] b[i]缩小了原本的值，变成序列号，由 a[]调用序列号得到原值
对于第 3 步，若离散化后序列为 0, 1, 2, ..., size - 1 则用 lower_bound，序列为 1, 2, 3, ..., size 则用 upper_bound，其中 lower_bound 返回第 1 个不小于 b[i]的值的指针，
而 upper_bound 返回第 1 个大于 b[i]的值的指针，当然在这个题中也可以用 lower_bound 然后再加 1 得到与 upper_bound 相同结果

数学公式

a%b=a（a<b）
a%b<=a/2(a>b)

loga(b)=logc(b)/logc(a)
c++中 log 指取 ln
错排公式 ：考虑一个有 n 个元素的排列，若一个排列中所有的元素都不在自己原来的位置上，那么这样的排列就称为原排列的一个错排。用 D（n）表示；

D(n)=(n-1)[D(n-2)+D(n-1)]

特殊地，D(1) = 0, D(2) = 1

$$D(n)=n![(-1)_2/2!+...+(-1)_{(n-1)}/(n-1)!+(-1)_n/n!]$$

次方求模

```
int PowerMod(int a, int b, int c)
{
    int ans = 1;
    a = a % c;
    while(b>0)
    {
    if(b % 2 == 1)
    ans = (ans * a) % c;
    b = b/2;
    a = (a * a) % c;
```

```
    }
    return ans;
}
```

矩阵

```
const int MAXN=1010;
const int MAXM=1010;
struct Matrix{
int n,m;
int a[MAXN][MAXM];
void m_clear()
{
    n=0;m=0;
    memset(a,0,sizeof a);
}
Matrix operator + (const Matrix &b) const {
    Matrix tmp;
    tmp.m_clear();
    tmp.n=n;tmp.m=m;
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
                tmp.a[i][j]=a[i][j]+b.a[i][j];
    return tmp;
}
Matrix operator - (const Matrix &b) const {
    Matrix tmp;
    tmp.m_clear();
    tmp.n=n;tmp.m=m;
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
                tmp.a[i][j]=a[i][j]-b.a[i][j];
    return tmp;
}
Matrix operator * (const Matrix &b) const {
    Matrix tmp;
    tmp.m_clear();
    tmp.n=n;tmp.m=b.m;
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
            for(int k=0;k<m;k++)
                tmp.a[i][j]+=a[i][k]*b.a[k][j];
```

```
    return tmp;
}
void m_print()
{
    for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
                cout<<a[i][j]<<' ';
            cout<<endl;
        }
}
};
Matrix m_pow(Matrix A,int n)
{
    Matrix tmp;
    tmp.m_clear();
    tmp.n=A.n;tmp.m=A.n;
    for(int i=0;i<A.n;i++)
        tmp.a[i][i]=1;
    while(n>0)
    {
        if(n&1)
            tmp=tmp*A;
        A=A*A;
        n>>=1;
    }
    return tmp;
}
```

STL

sort

sort 处理 vector

sort(v.begin(), v.end(),less<int>());//升

sort(v.begin(), v.end(),greater<int>());//降

vector

vec.assign(size,value)对 vector vec 的 size 个赋值为 value

vector vec（size， value）初始化创建一个 vector 确定大小和 value

vector <int >vec (10) 确定 vector 的 capacity 为 10

vector <int > vec[10] 建立了一个向量数组 vector[i]表示 一个向量

vector<int > vec2(vec1) 复制

vec.insert(vec.begin()+2,5) 向迭代器 指向元素前插入新元素 5

vec.front () 第 0 个成员 vec.back()最后一个成员;

vec.at(4) 下标为 4 成员

vec.pop_back()删除尾元素

vec.erase（it） 删除迭代器所指向元素

vec.erase（first,last）删除迭代器所指定序列【 first ， last）

复杂度

erase insert find() O(N)

list

不支持【】复杂度 insert erase O(1) sort O (nlgn)(lg n)

deque

deque(int nSize,const T& t):创建一个 deque,元素个数为 nSize,且值均为 t

que.pop_front();弹出队首元素 不返回值

priority_queue

优先队列与队列的差别在于优先队列不是按照入队的顺序出队，而是按照队列中元素的优先权顺序出队（默认为大者优先，

也可以通过指定算子来指定自己的优先顺序）。

priority_queue 模板类有三个模板参数，第一个是元素类型，第二个容器类型，第三个是比较算子。其中后两个都可以省略

默认容器为 vector，默认算子为 less，即小的往前排，大的往后排（出队时序列尾的元素出队）。

priority_queue<int> q1;

priority_queue< pair<int, int> > q2; // 注意在两个尖括号之间

一定要留空格。

priority_queue<int, vector<int>, greater<int> > q3; // 定义小的先出队

如果要定义自己的比较算子，方法有多种，这里介绍其中的一种：重载比较运算符。优先队列试图将两个元素 x 和 y 代入比较运算符(对 less 算子，调用 x<y，对 greater 算子，调用 x>y)，若结果为真，则 x 排在 y 前面，y 将先于 x 出队，反之，则将 y 排在 x 前面，x 将先出队。

```
class T{
public:
int x, y, z;
T(int a, int b, int c):x(a), y(b), z(c){}
};
bool operator < (const T &t1, const T &t2){
return t1.z < t2.z; // 按照 z 的顺序来决定 t1 和 t2 的顺序
}
```

复杂度 O(N) push() pop()

map

在 STL 的头文件<map>中定义了模板类 map 和 multimap，用有序二叉树来存贮类型为 pair<const Key, T>的元素对序列。序列中的元素以 const Key 部分作为标识，map 中所有元素的 Key 值都必须是唯一的，multimap 则允许有重复的 Key 值。可以将 map 看作是由 Key 标识元素的元素集合，这类容器也被称为“关联容器”，可以通过一个 Key 值来快速确定一个元素，因此非常适合于需要按照 Key 值查找元素的容器。map 模板类需要四个模板参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

插入元素 m【key】=value；

m.insert(make_pair(key, value));

map<string, int>::iterator it = m.find(key);

如果 map 中存在与 key 相匹配的键值时，find 操作将返回指向该元素对的迭代器，否则，返回的迭代器等于 map 的 end()

m.erase(key)　m.erase(it);

复杂度　O(log n)　insert ，erase　find（）　【key】

hash_map

insert ()　erase()　【】　O(1)