可以给卡车加 $B_i$ 单位汽油。假设卡车的燃料箱的容量是无限大的，无论加多少油都没有问题。那么请问卡车是否能到达终点？如果可以，最少需要加多少次油？如果可以到达终点，输出最少的加油次数，否则输出-1。

**限制条件**

- $1 \leqslant N \leqslant 10000$
- $1 \leqslant L \leqslant 1000000, 1 \leqslant P \leqslant 1000000$
- $1 \leqslant A_i < L, 1 \leqslant B_i \leqslant 100$

**样例**

**输入**

```
N = 4, L = 25, P = 10
A = {10, 14, 20, 21}
B = {10, 5, 2, 4}
```

**输出**

```
2（在第1个和第2个加油站加油）
```

由于加油站的数量$N$非常大，必须想一个高效的解法。

我们稍微变换一下思考方式。在卡车开往终点的途中，只有在加油站才可以加油。但是，如果认为“在到达加油站$i$时，就获得了一次在之后的任何时候都可以加$B_i$单位汽油的权利”，在解决问题上应该也是一样的。而在之后需要加油时，就认为是在之前经过的加油站加的油就可以了。

加油的例子

那么，因为希望到达终点时加油次数尽可能少，所以当燃料为0了之后再进行加油看上去是一个不错的方法。在燃料为0时，应该使用哪个加油站来加油呢？显然，应该选能加油量$B_i$最大的加油站。

为了高效地进行上述操作，我们可以使用从大到小的顺序依次取出数值的优先队列。

- 在经过加油站$i$时，往优先队列里加入$B_i$。
- 当燃料箱空了时，
  - 如果优先队列也是空的，则无法到达终点。
  - 否则取出优先队列中的最大元素，并用来给卡车加油。

```
// 输入
int L, P, N;
int A[MAX_N + 1], B[MAX_N + 1];

void solve() {
  // 为了写起来方便，我们把终点也认为是加油站
  A[N] = L;
  B[N] = 0;
  N++;

  // 维护加油站的优先队列
  priority_queue<int> que;

  // ans: 加油次数, pos: 现在所在位置, tank: 油箱中汽油的量
  int ans = 0, pos = 0, tank = P;

  for (int i = 0; i < N; i++) {
    // 接下去要前进的距离
    int d = A[i] - pos;

    // 不断加油直到油量足够行驶到下一个加油站
    while (tank - d < 0) {
      if (que.empty()) {
        puts("-1");
        return;
      }
      tank += que.top();
      que.pop();
      ans++;
    }

    tank -= d;
    pos = A[i];
    que.push(B[i]);
  }

  printf("%d\n", ans);
}
```

## Fence Repair （PKU 3253）

题目描述请参照贪心法的“Fence Repair”（详见 2.2 节）

让我们再考虑一下贪心法一节中的“Fence Repair”问题。如果用朴素的方法实现的话，时间复杂度是$O(N^2)$。

由于只需从板的集合里取出最短的两块，并且把长度为两块板长度之和的板加入集合中即可，因此如果使用优先队列就可以高效地实现。一共需要进行$O(N)$次$O(\log N)$的操作，因此总的时间复杂度是$O(N \log N)$。

```
typedef long long ll;

// 输入
int N, L[MAX_N];

void solve() {
  ll ans = 0;

  // 声明一个从小到大取出数值的优先队列
  priority_queue<int, vector<int>, greater<int> > que;
  for (int i = 0; i < N; i++) {
    que.push(L[i]);
  }

  // 循环到只剩一块木板为止
  while (que.size() > 1) {
    // 取出最短的木板和次短的木板
    int l1, l2;
    l1 = que.top();
    que.pop();
    l2 = que.top();
    que.pop();

    // 把两块木板合并
    ans += l1 + l2;
    que.push(l1 + l2);
  }

  printf("%lld\n", ans);
}
```

## 2.4.3 二叉搜索树

### 1. 二叉搜索树的结构

二叉搜索树是能够高效地进行如下操作的数据结构。

- 插入一个数值
- 查询是否包含某个数值
- 删除某个数值

根据实现的不同，还可以实现其他各种各样的操作，是一种实用性很高的数据结构。二叉搜索树

如何储存数值请参见下图。

二叉搜索树的例子

所有的节点，都满足左子树上的所有节点都比自己的小，而右子树上的所有节点都比自己大这一条件。

二叉搜索树能够高效地管理数的集合。例如，可以通过如下方法在上图的二叉搜索树中查询是否存在10。

- 根节点的数值是7，比10小，所以往右儿子节点走。
- 走到的节点的数值是15，比10大，所以往左儿子节点走。
- 走到的节点是10，因此10在集合中。

查找的例子

接下来，如何往树中插入新的数值呢？

如果我们按照查找数值的方法试图查找这个数值的节点，就可以知道其对应的节点的所在位置，之后在那个位置插入新的节点即可。例如，我们需要插入数值6。和查找的方法类似，从根节点出发，通过“左→右”两步，就可以知道6应该是5的右儿子，因此在5的右儿子的位置插入6的节点即可（请参见图A）。

最后是删除数值。数值的删除比起之前提到的操作稍微麻烦一些。例如，我们要删除数值15。如果删除了15所在的节点，那么它的两个儿子10和17就悬空了。于是，把11提到15所在的位置就可以解决问题（请参见图B）。

图A：插入数值的例子

图B：删除的例子

一般来说，需要根据下面几种情况分别进行处理。

- 需要删除的节点没有左儿子，那么就把右儿子提上去。
- 需要删除的节点的左儿子没有右儿子，那么就把左儿子提上去。
- 以上两种情况都不满足的话，就把左儿子的子孙中最大的节点提到需要删除的节点上。

### 2. 二叉搜索树的复杂度

不论哪一种操作，所花的时间都和树的高度成正比。因此，如果共有$n$个元素，那么平均每次操作需要$O(\log n)$的时间。

### 3. 二叉搜索树的实现

下面是二叉搜索树的一种实现方法。

```
node *root = NULL;
root = insert(root, 1);
find(root, 1);
```

像代码中一样，我们使用函数的返回值进行操作。另外需要注意insert、delete返回的是node结构体的指针。

```
// 表示节点的结构体
struct node {
  int val;
  node *lch, *rch;
};

// 插入数值x
node *insert(node *p, int x) {
  if (p == NULL) {
    node *q = new node;
    q->val = x;
    q->lch = q->rch = NULL;
    return q;
  }
  else {
    if (x < p->val) p->lch = insert(p->lch, x);
    else p->rch = insert(p->rch, x);
    return p;
  }
}

// 查找数值x
bool find(node *p, int x) {
  if (p == NULL) return false;
  else if (x == p->val) return true;
  else if (x < p->val) return find(p->lch, x);
  else return find(p->rch, x);
}

// 删除数值x
node *remove(node *p, int x) {
  if (p == NULL) return NULL;
  else if (x < p->val) p->lch = remove(p->lch, x);
  else if (x > p->val) p->rch = remove(p->rch, x);
  else if (p->lch == NULL) {
    node *q = p->rch;
    delete p;
    return q;
  }
  else if (p->lch->rch == NULL) {
    node *q = p->lch;
    q->rch = p->rch;
    delete p;
    return q;
```

```
    }
    else {
      node *q;
      for (q = p->lch; q->rch->rch != NULL; q = q->rch);
      node *r = q->rch;
      q->rch = r->lch;
      r->lch = p->lch;
      r->rch = p->rch;
      delete p;
      return r;
    }
    return p;
  }
```

### 4. 编程语言的标准库

和堆一样，实际上在许多情况下都不需要自己实现二叉搜索树。许多编程语言都在标准库里实现了简单易用的二叉搜索树。例如在C++中，STL里有set和map容器。set是像前面所说的一样使用二叉搜索树维护集合的容器，而map则是维护键和键对应的值的容器。

下面是一些使用set和map的例子。详细的内容可以参考STL相关的文档和书籍。

```
#include <cstdio>
#include <set>
using namespace std;

int main() {
  // 声明
  set<int> s;

  // 插入元素
  s.insert(1);
  s.insert(3);
  s.insert(5);

  // 查找元素
  set<int>::iterator ite;;

  ite = s.find(1);
  if (ite == s.end()) puts("not found");
  else puts("found");                        // 输出found

  ite = s.find(2);
  if (ite == s.end()) puts("not found"); // 输出not found
  else puts("found");

  // 删除元素
  s.erase(3);

  // 其他的查找元素的方法
```

```
  if (s.count(3) != 0) puts("found");
  else puts("not found");                    // 输出not found

  // 遍历所有元素
  for (ite = s.begin(); ite != s.end(); ++ite) {
    printf("%d\n", *ite);
  }

  return 0;
}
```

下面是map的使用方法：

```
#include <cstdio>
#include <map>
#include <string>
using namespace std;

int main() {
  // 声明(int为键, const char*为值)
  map<int, const char*> m;

  // 插入元素
  m.insert(make_pair(1, "ONE"));
  m.insert(make_pair(10, "TEN"));
  m[100] = "HUNDRED";                        // 其他的写法

  // 查找元素
  map<int, const char*>::iterator ite;
  ite = m.find(1);
  puts(ite->second);                         //（输出）ONE

  ite = m.find(2);
  if (ite == m.end()) puts("not found");     // not found
  else puts(ite->second);

  puts(m[10]);                               // 其他的写法

  // 删除元素
  m.erase(10);

  // 遍历一遍所有元素
  for (ite = m.begin(); ite != m.end(); ++ite) {
    printf("%d: %s\n", ite->first, ite->second);
  }

  return 0;
}
```

此外，还有能存放重复键值的multiset和multimap等容器。

**专栏 平衡二叉树**

考虑一下根据之前的说明，在向二叉树插入节点时，如果以 1, 2, 3, 4, 5……的顺序插入的话会产生什么问题。如果这样插入节点的话，就会像下图一样，变得如同一条链表一般。

退化的二叉树的例子

如果按照这个顺序插入 $n$ 个元素，树的高度就会变成 $n$，那么所有的操作就都需要 $O(n)$时间才能完成了。所有操作本该都是 $O(\log n)$时间的二叉搜索树，如果变成了 $O(n)$，计算的复杂度就完全不同了。

而平衡二叉树恰好能避免这样的问题。平衡二叉树为了避免这种退化的情况发生，巧妙地使用旋转操作来保持树的平衡。

旋转操作的例子

比起普通的二叉搜索树，平衡二叉树在实现上复杂得多。由于在编程语言标准中的二叉搜索树都很好地实现了平衡二叉树，因此，如果功能上足够，应该尽可能使用标准库里的实现。

## 2.4.4　并查集

### 1. 并查集是什么

并查集是一种用来管理元素分组情况的数据结构。并查集可以高效地进行如下操作。不过需要注意并查集虽然可以进行合并操作，但是却无法进行分割操作。

- 查询元素a和元素b是否属于同一组。
- 合并元素a和元素b所在的组。

并查集的功能示意图

### 2. 并查集的结构

并查集也是使用树形结构实现的。不过，不是二叉树。

分组和对应的树的例子

每个元素对应一个节点，每个组对应一棵树。在并查集中，哪个节点是哪个节点的父亲以及树的形状等信息无需多加关注，整体组成一个树形结构才是重要的。

(1) 初始化

我们准备$n$个节点来表示$n$个元素。最开始时没有边。

初始化状态的例子

(2) 合并

像下图一样，从一个组的根向另一个组的根连边，这样两棵树就变成了一棵树，也就把两个组合并为一个组了。

合并的例子

(3) 查询

为了查询两个节点是否属于同一组，我们需要沿着树向上走，来查询包含这个元素的树的根是谁。如果两个节点走到了同一个根，那么就可以知道它们属于同一组。

在下图中，元素2和元素5都走到了元素1，因此它们属于同一组。另一方面，由于元素7走到的是元素6，因此同元素2或元素5属于不同组。

查询的例子

### 3. 并查集实现中的注意点

正如二叉搜索树的补充讲解（详见2.2节的专栏）中提到的那样，在树形数据结构里，如果发生了退化的情况，那么复杂度就会变得很高。因此，有必要想办法避免退化的发生。在并查集中，只需按照如下方法就可以避免退化。

- 对于每棵树，记录这棵树的高度（rank）。
- 合并时如果两棵树的rank不同，那么从rank小的向rank大的连边。

考虑了高度的合并的例子

此外，通过路径压缩，可以使得并查集更加高效。对于每个节点，一旦向上走到了一次根节点，就把这个点到父亲的边改为直接连向根。

路径压缩的例子1

在此之上，不仅仅是所查询的节点，在查询过程中向上经过的所有的节点，都改为直接连到根上。这样再次查询这些节点时，就可以很快知道根是谁了。

路径压缩的例子2

在使用这种简化的方法时，为了简单起见，即使树的高度发生了变化，我们也不修改rank的值。

### 4. 并查集的复杂度

加入了这两个优化之后的并查集效率非常高。对$n$个元素的并查集进行一次操作的复杂度是$O(\alpha(n))$。在这里，$\alpha(n)$是阿克曼（Ackermann）函数的反函数[①]。这比$O(\log(n))$还要快。

不过，这是“均摊复杂度”。也就是说，并不是每一次操作都满足这个复杂度，而是多次操作之后平均每一次操作的复杂度是$O(\alpha(n))$的意思。

### 5. 并查集的实现

下面是并查集的实现的例子。在例子中，我们用编号代表每个元素。数组par表示的是父亲的编号，par[x]=x时，x是所在的树的根。

```
int par[MAX_N];  // 父亲
int rank[MAX_N]; // 树的高度

// 初始化n个元素
void init(int n) {
  for (int i = 0; i < n; i++) {
    par[i] = i;
    rank[i] = 0;
  }
}

// 查询树的根
int find(int x) {
  if (par[x] == x) {
    return x;
  } else {
    return par[x] = find(par[x]);
  }
}

// 合并x和y所属的集合
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if (x == y) return;

  if (rank[x] < rank[y]) {
    par[x] = y;
  } else {
    par[y] = x;
    if (rank[x] == rank[y]) rank[x]++;
  }
}
```

① 准确来说，如果记阿克曼函数为$A$的话，是$f(n)=A(n,n)$的反函数。

```
// 判断x和y是否属于同一个集合
bool same(int x, int y) {
  return find(x) == find(y);
}
```

### 6. 需要用到并查集的问题

**食物链（POJ 1182）**

有 $N$ 只动物，分别编号为 $1, 2, \cdots, N$。所有动物都属于 $A, B, C$ 中的其中一种。已知 $A$ 吃 $B$、$B$ 吃 $C$、$C$ 吃 $A$。按顺序给出下面的两种信息共 $K$ 条。

- 第一种：$x$ 和 $y$ 属于同一种类。
- 第二种：$x$ 吃 $y$。

然而这些信息有可能会出错。有可能有的信息和之前给出的信息矛盾，也有的信息可能给出的 $x$ 和 $y$ 不在 $1, 2, \cdots, N$ 的范围内。求在 $K$ 条信息中有多少条是不正确的。计算过程中，我们将忽视诸如此类的错误信息。

**限制条件**

- $1 \leqslant N \leqslant 50000$
- $0 \leqslant K \leqslant 100000$

**样例**

**输入**

```
N = 100, K = 7
信息有下面7条
第一种, x = 101, y = 1
第二种, x = 1, y = 2
第二种, x = 2, y = 3
第二种, x = 3, y = 3
第一种, x = 1, y = 3
第二种, x = 3, y = 1
第一种, x = 5, y = 5
```

**输出**

```
3（第1、4、5条是错误的信息）
```

由于$N$和$K$很大，所以必须高效地维护动物之间的关系，并快速判断是否产生了矛盾。并查集是维护“属于同一组”的数据结构，但是在本题中，并不只有属于同一类的信息，还有捕食关系的存在。因此需要开动脑筋维护这些关系。

对于每只动物$i$创建3个元素$i$–$A$, $i$–$B$, $i$–$C$，并用这$3 \times N$个元素建立并查集。这个并查集维护如下信息：

- $i$–$x$表示“$i$属于种类$x$”。
- 并查集里的每一个组表示组内所有元素代表的情况都同时发生或不发生。

例如，如果$i$–$A$和$j$–$B$在同一个组里，就表示如果$i$属于种类$A$那么$j$一定属于种类$B$，如果$j$属于种类$B$那么$i$一定属于种类$A$。因此，对于每一条信息，只需要按照下面进行操作就可以了。

- 第一种，$x$和$y$属于同一种类……合并$x$–$A$和$y$–$A$、$x$–$B$和$y$–$B$、$x$–$C$和$y$–$C$。
- 第二种，$x$吃$y$……合并$x$–$A$和$y$–$B$、$x$–$B$和$y$–$C$、$x$–$C$和$y$–$A$。

不过在合并之前，需要先判断合并是否会产生矛盾。例如在第一种信息的情况下，需要检查比如$x$–$A$和$y$–$B$或者$y$–$C$是否在同一组等信息。

```
// 输入（T是信息的类型）
int N, K;
int T[MAX_K], X[MAX_K], Y[MAX_K];

// 在这里省略了并查集部分的代码
void solve() {
  // 初始化并查集
  // 元素x, x + N, x + 2 * N分别代表x-A, x-B, x-C
  init(N * 3);

  int ans = 0;
  for (int i = 0; i < K; i++) {
    int t = T[i];
    int x = X[i] - 1, y = Y[i] - 1;  // 把输入变成0, …, N-1的范围

    // 不正确的编号
    if (x < 0 || N <= x || y < 0 || N <= y) {
      ans++;
      continue;
    }

    if (t == 1) {
      // "x和y属于同一类"的信息
      if (same(x, y + N) || same(x, y + 2 * N)) {
        ans++;
      }
      else {
        unite(x, y);
        unite(x + N, y + N);
        unite(x + N * 2, y + N * 2);
      }
    }
    else {
      // “x吃y” 的信息
      if (same(x, y) || same(x, y + 2 * N)) {
```

[redacted]

# 2.5 它们其实都是“图”

➠图是表示一些事物或者状态的关系的表达方法。由于许多问题都可以归约为图的问题，人们提出了许多和图相关的算法。因此，在程序设计竞赛中有许多需要直接对图进行处理或是间接用图解决的问题。

## 2.5.1 图是什么

图由顶点（vertex, node）和边（edge）组成。顶点代表对象。在示意图中，我们使用点或圆来表示。边表示的是两个对象的连接关系。在示意图中，我们使用连接两顶点之间的线段来表示。顶点的集合是$V$、边的集合是$E$的图记为$G=(V, E)$，连接两点$u$和$v$的边用$e=(u, v)$表示。

图的例子

### 1. 图的种类

图大体上分为2种。边没有指向性的图叫做无向图，边具有指向性的图叫做有向图。表示朋友关系的图（顶点表示人、边表示朋友关系的图）和路线图是无向图。表示数值的大小关系的图（顶点表示数值、$A>B$时从$A$向$B$连一条边得到的图）和流程图是有向图。

无向图的例子

有向图的例子

我们可以给边赋予各种各样的属性。比较具有代表性的有权值（cost）。边上带有权值的图叫做带权图。在不同问题中，权值可以代表距离、时间以及价格等不同的属性。

边表示连接城市的道路，权值表示距离

## 2. 无向图的术语

两个顶点之间如果有边连接，那么就视为两个顶点相邻。相邻顶点的序列称为路径。起点和终点重合的路径叫做圈。任意两点之间都有路径连接的图叫做连通图。顶点连接的边数叫做这个顶点的度。

1–3–4–5是路径
2–3–4–5–2是圈

路径和圈的例子

左图是连通图、右图是非连通图

度

没有圈的连通图叫做树（tree），没有圈的非连通图叫做森林。一棵树的边数恰好是顶点数−1。反之，边数等于顶点数−1的连通图就是一棵树。

如果在树上选择一个顶点作为根（root），就可以把根提到最上面，而离根越远的顶点越往下安排其位置。这样的树叫做有根树。不过，对于无根树，有时选择适当的顶点作为根使之变成有根数，可以使问题得到简化。如果把有根树看作家谱图，则可以在顶点之间建立父子关系。也可以认为这是给边加上了方向。

树的例子

有根树的例子

### 3. 有向图的术语

在本书中，以有向图的顶点$v$为起点的边的集合记作 $\delta_+(v)$，以顶点$v$为终点的边的集合记作 $\delta_-(v)$。$|\delta_+(v)|$叫做$v$的出度，$|\delta_-(v)|$叫做边的入度。

出度和入度

没有圈的有向图叫做DAG（Directed Acyclic Graph）。例如，让我们用顶点表示整数，$n$能整除$m$时从$n$向$m$连一条边的图，这就构成一个DAG。像下图一样，在DAG中我们可以给顶点标记一个先后顺序。

DAG的例子

对于每个顶点我们给它一个编号，第$i$号顶点叫做$v_i$。那么存在从顶点$v_i$到顶点$v_j$的边时就有$i<j$成立，这样的编号方式叫做拓扑序。

上图的拓扑序

如果把图中的顶点按照拓扑序从左到右排列，那么所有的边都是从左指向右的。因此，通过这样的编号方式，有些DAG问题就可以使用DP来解决了。求解拓扑序的算法叫做拓扑排序。

## 2.5.2　图的表示

为了能在程序中对图进行处理，需要把顶点和边用具体的数据结构存储下来。在图的表示方法中，比较具有代表性的有邻接矩阵和邻接表。需要注意的是，两种表示方法都有各自的优缺点，根据问题的不同，使用不同的存储方式可能会影响算法的时间复杂度。接下来，记顶点和边的集合为$V$和$E$，$|V|$和$|E|$表示顶点和边的个数。另外，在$V$中，顶点被编号为$0 \sim |V|-1$。

### 1. 邻接矩阵

邻接矩阵使用$|V| \times |V|$的二维数组来表示图。$g[i][j]$表示的是顶点$i$和顶点$j$的关系。

由于在无向图中，只需知道“顶点$i$和顶点$j$之间是否有边连着”这样的信息，因此如果顶点$i$和顶点$j$之间有边相连，那么$g[i][j]$和$g[j][i]$就设为1，否则设为0。这样就可以表示一个无向图了。

| i\j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | 1 | 0 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 |

无向图和对应的邻接矩阵

由于在有向图中，只需要知道“是否有从顶点$i$发出指向顶点$j$的边”这样的信息，因此如果顶点$i$

有一条指向顶点$j$的边，那么$g[i][j]$就设为1，否则设为0。这样就可以表示一个有向图了。有向图与无向图不同，并不需要满足$g[i][j]=g[j][i]$。

| i\j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 |

有向图和对应的邻接矩阵

在带权图中，$g[i][j]$表示的是顶点$i$到顶点$j$的边的权值。由于在边不存在的情况下，如果将$g[i][j]$设为0，就无法和权值为0的情况区分开来，因此选取适当的较大的常数INF（只要能和普通的权值区别开来就可以了），然后令$g[i][j]$=INF就好了。当然，在无向图中还是要保持$g[i][j]=g[j][i]$。在一条边上有多种不同权值的情况下，定义多个同样的$|V|\times|V|$数组，或者是使用结构体或类作为数组的元素，就可以和原来一样对图进行处理了。

| i\j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | ∞ | 3 | ∞ |
| 1 | 2 | 0 | 5 | 4 | ∞ |
| 2 | ∞ | 5 | 0 | 10 | ∞ |
| 3 | 3 | 4 | 10 | 0 | 7 |
| 4 | ∞ | ∞ | ∞ | 7 | 0 |

带权图和对应的邻接矩阵

使用邻接矩阵的好处是可以在常数时间内判断两点之间是否有边存在，但是需要花费$O(|V|^2)$的空间。在边很少的稀疏图里十分浪费。例如，如果图是一棵树，因为边数只有$|V|-1$条，所以数组$g$绝大部分的元素都变成了0。在$|V|$达到1000000时，即使$g$的每个元素只需要1个字节的空间，整个数组也需要1TB才能存下。

此外，两点之间有重边或者某个顶点有自环（参照下图）的情况需要特别注意。在无权图中，只需要设$g[i][j]$为顶点$i$到顶点$j$的边数即可，但是在带权图中却无法这样。大部分情况下，只需要保存权值最小（最大）的边就可以了，所以在这种情况下可以无视其他的边。必须保存所有的边时，可以使用邻接表。

重边和自环

### 2. 邻接表

用邻接矩阵表示稀疏图会浪费大量内存空间。而在邻接表中，是通过把“从顶点0出发有到顶点2, 4, 5的边”这样的信息保存在链表中来表示图的。这样只需要$O(|V|+|E|)$的内存空间。

| 顶点 | 相邻的顶点的列表 |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 1 |
| 3 | 0,2 |
| 4 | 3 |

邻接表

事实上，实现邻接表的方式多种多样，每个人的写法可能都有所不同。下面是邻接表的一种实现方式。输入数据如下所示。

```
3 3（顶点数 边数）
0 1（有一条0到1的边）
0 2（有一条0到2的边）
1 2（有一条1到2的边）
```

**样例 1**

```
vector<int> G[MAX_V];
/*
 * 边上有属性的情况
 * struct edge { int to, cost; };
 * vector<edge> G[MAX_V];
 */

int main() {
  int V, E;
  scanf("%d %d", &V, &E);
  for (int i = 0; i < E; i++) {
    // 从s向t连边
    int s, t;
    scanf("%d %d", &s, &t);
    G[s].push_back(t);
    // 如果是无向图，则需要再从t向s连边
  }
  /*
   * 图的操作
   */
  return 0;
}
```

样例 2

```
struct vertex {
  vector<vertex*> edge;
  /*
   * 顶点的属性
   */
};
vertex G[MAX_V];

int main() {
  int V, E;
  scanf("%d %d", &V, &E);
  for (int i = 0; i < E; i++) {
    int s, t;
    scanf("%d %d", &s, &t);
    G[s].edge.push_back(&G[t]);
    // G[t].edge.push_back(&G[s]);
  }
  /*
   * 图的操作
   */
  return 0;
}
```

在带权图等边有附加属性的图中，将边用结构体或者类来表示就可以很方便地存储了。邻接表虽然在边数稀少时只需要占用少量内存，但是和邻接矩阵相比实现较为复杂。而且，在邻接表中查询两点间是否有边需要遍历一遍链表才能知道。

### 2.5.3 图的搜索

通过前面的铺垫，我们已经可以通过程序对图进行处理了。让我们来试着解决下面这道题。

**二分图判定**

给定一个具有 $n$ 个顶点的图。要给图上每个顶点染色，并且要使相邻的顶点颜色不同。问是否能最多用 2 种颜色进行染色？题目保证没有重边和自环。

**限制条件**

- $1 \leq n \leq 1000$

样例 1

**输入**

n=3（请参见下图）

**输出**

No（需要3种颜色）

**样例 2**

**输入**

n=4（请参见下图）

**输出**

Yes（{0, 2}染成红色，{1, 3}染成白色即可）

把相邻顶点染成不同颜色的问题叫做图着色问题。对图进行染色所需要的最小颜色数称为最小着色数。最小着色数是2的图称作二分图。

**最小着色数是5的图**

如果只用2种颜色，那么确定一个顶点的颜色之后，和它相邻的顶点的颜色也就确定了。因此，选择任意一个顶点出发，依次确定相邻顶点的颜色，就可以判断是否可以被2种颜色染色了。这个问题如果用深度优先搜索的话，能够简单地实现。

```
// 输入
vector<int> G[MAX_V]; // 图
int V;                // 顶点数
```

```
int color[MAX_V];      // 顶点i的颜色（1 or -1）

// 把顶点染成1或-1
bool dfs(int v, int c) {
  color[v] = c; // 把顶点v染成颜色c
  for (int i = 0; i < G[v].size(); i++) {
    // 如果相邻的顶点同色，则返回false
    if (color[G[v][i]] == c) return false;
    // 如果相邻的顶点还没被染色，则染成-c
    if (color[G[v][i]] == 0 && !dfs(G[v][i], -c)) return false;
  }
  // 如果所有顶点都染过色了，则返回true
  return true;
}

void solve() {
  for (int i = 0; i < V; i++) {
    if (color[i] == 0) {
      // 如果顶点i还没被染色，则染成1
      if (!dfs(i, 1)) {
        printf("No\n");
        return;
      }
    }
  }
  printf("Yes\n");
}
```

如果是连通图，那么一次dfs就可以访问到所有的顶点。如果题目描述中没有说明，那么有可能图是不连通的，这样就需要依次检查每个顶点是否访问过。判断图是否连通或者是否是一棵树，都只需将dfs进行一些修改就可以了。通过dfs也可以求图的拓扑序。由于每个顶点和每条边都只访问了一次，因此复杂度是$O(|V|+|E|)$。

### 2.5.4 最短路问题

最短路问题是图论中最基础的问题，在程序设计竞赛试题中也经常出现。最短路是给定两个顶点，在以这两个点为起点和终点的路径中，边的权值和最小的路径。如果把权值当作距离，考虑最短距离的话就很容易理解了。智力游戏中的求解最少步数问题也可以说是一种最短路问题。

最短路的例子

### 1. 单源最短路问题1（Bellman-Ford算法）

单源最短路问题是固定一个起点，求它到其他所有点的最短路的问题。终点也固定的问题叫做两点之间最短路问题。但是因为解决单源最短路问题的复杂度也是一样的，因此通常当作单源最短路问题来求解。

记从起点$s$出发到顶点$i$的最短距离为$d[i]$。则下述等式成立。

$$d[i]=\min\{d[j]+(\text{从}j\text{到}i\text{的边的权值})|e=(j,i)\in E\}$$

如果给定的图是一个DAG，就可以按拓扑序给顶点编号，并利用用这条递推关系式计算出$d$。但是，如果图中有圈，就无法依赖这样的顺序进行计算。在这种情况下，记当前到顶点$i$的最短路长度为$d[i]$，并设初值$d[s]=0, d[i]=\text{INF}$（足够大的常数），再不断使用这条递推关系式更新$d$的值，就可以算出新的$d$。只要图中不存在负圈，这样的更新操作就是有限的。结束之后的$d$就是所求的最短距离了。

```
// 从顶点from指向顶点to的权值为cost的边
struct edge { int from, to, cost; };

edge es[MAX_E];  // 边

int d[MAX_V];    // 最短距离
int V, E;        // V是顶点数，E是变数

// 求解从顶点s出发到所有点的最短距离
void shortest_path(int s) {
  for (int i = 0; i < V; i++) d[i] = INF;
  d[s] = 0;
  while (true) {
    bool update = false;
    for (int i = 0; i < E; i++) {
      edge e = es[i];
      if (d[e.from] != INF && d[e.to] > d[e.from] + e.cost) {
        d[e.to] = d[e.from] + e.cost;
        update = true;
      }
    }
    if (!update) break;
  }
}
```

这个算法叫做Bellman-Ford算法。如果在图中不存在从$s$可达的的负圈，那么最短路不会经过同一个顶点两次（也就是说，最多通过$|V|-1$条边），while(true)的循环最多执行$|V|-1$次，因此，复杂度是$O(|V|\times|E|)$。反之，如果存在从$s$可达的负圈，那么在第$|V|$次循环中也会更新$d$的值，因此也可以用这个性质来检查负圈。如果一开始对所有的顶点$i$，都把$d[i]$初始化为0，那么可以检查出所有的负圈。

```
// 如果返回true则存在负圈
bool find_negative_loop() {
  memset(d, 0, sizeof(d));

  for (int i = 0; i < V; i++) {
    for (int j = 0; j < E; j++) {
      edge e = es[j];
      if (d[e.to] > d[e.from] + e.cost) {
        d[e.to] = d[e.from] + e.cost;

        // 如果第n次仍然更新了，则存在负圈
        if (i == V - 1) return true;
      }
    }
  }
  return false;
}
```

### 2. 单源最短路问题2（Dijkstra算法）

让我们考虑一下没有负边的情况。在Bellman-Ford算法中，如果$d[i]$还不是最短距离的话，那么即使进行$d[j]=d[i]+$(从$i$到$j$的边的权值)的更新，$d[j]$也不会变成最短距离。而且，即使$d[i]$没有变化，每一次循环也要检查一遍从$i$出发的所有边。这显然是很浪费时间的。因此可以对算法做如下修改。

(1) 找到最短距离已经确定的顶点，从它出发更新相邻顶点的最短距离。
(2) 此后不需要再关心1中的“最短距离已经确定的顶点”。

在(1)和(2)中提到的“最短距离已经确定的顶点”要怎么得到是问题的关键。在最开始时，只有起点的最短距离是确定的。而在尚未使用过的顶点中，距离$d[i]$最小的顶点就是最短距离已经确定的顶点。这是因为由于不存在负边，所以$d[i]$不会在之后的更新中变小。这个算法叫做Dijkstra算法。

**按照A->B->C的顺序确定最短距离后的d的值。下一个使用的是d=7的顶点D。**

```
int cost[MAX_V][MAX_V];   // cost[u][v]表示边e=(u,v)的权值（不存在这条边时设为INF）
int d[MAX_V];             // 顶点s出发的最短距离
bool used[MAX_V];         // 已经使用过的图
int V;                    // 顶点数

// 求从起点s出发到各个顶点的最短距离
```

```
void dijkstra(int s) {
  fill(d, d + V, INF);
  fill(used, used + V, false);
  d[s] = 0;

while(true) {
  int v = -1;
  // 从尚未使用过的顶点中选择一个距离最小的顶点
  for (int u = 0; u < V; u++) {
    if (!used[u] && (v == -1 || d[u] < d[v])) v = u;
  }

  if (v == -1) break;
    used[v] = true;

    for (int u = 0; u < V; u++) {
      d[u] = min(d[u], d[v] + cost[v][u]);
    }
  }
}
```

使用邻接矩阵实现的Dijkstra算法的复杂度是$O(|V|^2)$。使用邻接表的话，更新最短距离只需要访问每条边一次即可，因此这部分的复杂度是$O(|E|)$。但是每次要枚举所有的顶点来查找下一个使用的顶点，因此最终复杂度还是$O(|V|^2)$。在|E|比较小时，大部分的时间花在了查找下一个使用的顶点上，因此需要使用合适的数据结构对其进行优化。

需要优化的是数值的插入（更新）和取出最小值两个操作，因此使用堆就可以了。把每个顶点当前的最短距离用堆维护，在更新最短距离时，把对应的元素往根的方向移动以满足堆的性质。而每次从堆中取出的最小值就是下一次要使用的顶点。这样堆中元素共有$O(|V|)$个，更新和取出数值的操作有$O(|E|)$次，因此整个算法的复杂度是$O(|E| \log |V|)$。

下面是使用STL的priority_queue[①]的实现。在每次更新时往堆里插入当前最短距离和顶点的值对。插入的次数是$O(|E|)$次，因此元素也是$O(|E|)$个。当取出的最小值不是最短距离的话，就丢弃这个值。这样整个算法也可以在同样的复杂度内完成。

```
struct edge { int to, cost; };
typedef pair<int, int> P;  // first是最短距离，second是顶点的编号

int V;
vector<edge> G[MAX_V];
int d[MAX_V];

void dijkstra(int s) {
  // 通过指定greater<P>参数，堆按照first从小到大的顺序取出值
  priority_queue<P, vector<P>, greater<P> > que;
```

① 当所有的边的权值都相等时，单源最短路可以通过广度优先搜索来求解。在这种情况下，Dijkstra算法使用priority_queue和queue具有相同效果。因此复杂度会产生变化，请特别注意。

```
    fill(d, d + V, INF);
    d[s] = 0;
    que.push(P(0, s));

    while (!que.empty()) {
      P p = que.top(); que.pop();
      int v = p.second;
      if (d[v] < p.first) continue;
      for (int i = 0; i < G[v].size(); i++) {
        edge e = G[v][i];
        if (d[e.to] > d[v] + e.cost) {
          d[e.to] = d[v] + e.cost;
          que.push(P(d[e.to], e.to));
        }
      }
    }
  }
```

相对于Bellman-Ford的$O(|V||E|)$的复杂度，Dijkstra算法的复杂度是$O(|E|\log|V|)$，可以更加高效地计算最短路的长度。但是，在图中存在负边的情况下，Dijkstra算法就无法正确求解问题，还是需要使用Bellman-Ford算法。

### 3. 任意两点间的最短路问题（Floyd-Warshall算法）

求解所有两点间的最短路的问题叫做任意两点间的最短路问题。让我们试着用DP来求解任意两点间的最短路问题。只使用顶点$0$~$k$和$i, j$的情况下，记$i$到$j$的最短路长度为$d[k+1][i][j]$。$k=-1$时，认为只使用$i$和$j$，所以$d[0][i][j]=\text{cost}[i][j]$。接下来让我们把只使用顶点0~$k$的问题归约到只使用0~$k-1$的问题上。

只使用0~$k$时，我们分$i$到$j$的最短路正好经过顶点$k$一次和完全不经过顶点$k$两种情况来讨论。不经过顶点$k$的情况下，$d[k][i][j]=d[k-1][i][j]$。通过顶点$k$的情况下，$d[k][i][j]=d[k-1][i][k]+d[k-1][k][j]$。合起来，就得到了$d[k][i][j]=\min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])$。这个DP也可以使用同一个数组，不断进行$d[i][j]=\min(d[i][j], d[i][k]+d[k][j])$的更新来实现。

这个算法叫做Floyd-Warshall算法，可以在$O(|V|^3)$时间里求得所有两点间的最短路长度。Floyd-Warshall算法和Bellman-Ford算法一样，可以处理边是负数的情况。而判断图中是否有负圈，只需检查是否存在$d[i][i]$是负数的顶点$i$就可以了。

```
int d[MAX_V][MAX_V];   // d[u][v]表示边e=(u,v)的权值（不存在时设为INF，不过d[i][i]=0）
int V;                 // 顶点数

void warshall_floyd() {
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

这样通过三重循环非常简单地就可以求出所有两点间的最短路长度。由于实现起来非常简单，如果复杂度在可以承受的范围之内，单源最短路也可以使用Floyd-Warshall算法进行求解。

### 4. 路径还原

截至目前，我们都只是在求解最短距离。虽然许多问题只需输出最短距离就可以了，但是也有的问题需要求解最短路的路径。我们以Dijkstra算法为例，试着来求解最短路径。在求解最短距离时，满足$d[j]=d[k]+\mathrm{cost}[k][j]$的顶点$k$，就是最短路上顶点$j$的前趋节点，因此通过不断寻找前趋节点就可以恢复出最短路。时间度杂度是$O(|E|)$。

此外，如果用prev[$j$]来记录最短路上顶点j的前趋，那么就可以在$O(|V|)$的时间内完成最短路的恢复。在$d[j]$被$d[j]=d[k]+\mathrm{cost}[k][j]$更新时，修改prev[$j$]=$k$，这样就可以求得prev数组。在计算从$s$出发到$j$的最短路时，通过prev[$j$]就可以知道顶点$j$的前趋，因此不断把$j$替换成prev[$j$]直到$j=s$为止就可以了。Bellman-Ford算法和Floyd-Warshall算法都可以用类似的方法进行最短路的还原。

```
int prev[MAX_V];  // 最短路上的前趋顶点

// 求从起点s出发到各个顶点的最短距离
void dijkstra(int s) {
  fill(d, d + V, INF);
  fill(used, used + V, false);
  fill(prev, prev + V, -1);
  d[s] = 0;

  while(true) {
    int v = -1;
    for (int u = 0; u < V; u++) {
      if (!used[u] && (v == -1 || d[u] < d[v])) v = u;
    }

    if (v == -1) break;
    used[v] = true;

    for (int u = 0; u < V; u++) {
      if (d[u] > d[v] + cost[v][u]) {
        d[u] = d[v] + cost[v][u];
        prev[u] = v;
      }
    }
  }
}

// 到顶点t的最短路
vector<int> get_path(int t) {
  vector<int> path;
  for (; t != -1; t = prev[t]) path.push_back(t); // 不断沿着prev[t]走直到t=s
  // 这样得到的是按照t到s的顺序，所以翻转之
  reverse(path.begin(), path.end());
  return path;
}
```

### 2.5.5 最小生成树

给定一个无向图，如果它的某个子图中任意两个顶点都互相连通并且是一棵树，那么这棵树就叫做生成树（Spanning Tree）。如果边上有权值，那么使得边权和最小的生成树叫做最小生成树（MST，Minimum Spanning Tree）。

例如我们假设有这样一个图：把顶点看作村庄，边看作计划要修建的道路。为了在所有的村庄间通行，恰好修建村庄数目-1条道路时的情形就对应了一棵生成树。修建道路需要投入建设费，那么求解使得道路建设费用最小的生成树就是最小生成树问题。

最小生成树（权值和17）

生成树

常见的求解最小生成树的算法有Kruskal算法和Prim算法。很显然，生成树是否存在和图是否连通是等价的，因此我们假定图是连通的。

#### 1. 最小生成树问题1（Prim算法）

首先我们介绍Prim算法。Prim算法和Dijkstra算法十分相似，都是从某个顶点出发，不断添加边的算法。

首先，我们假设有一棵只包含一个顶点$v$的树T。然后贪心地选取T和其他顶点之间相连的最小权值的边，并把它加到T中。不断进行这个操作，就可以得到一棵生成树了。接下来我们来证明通过这个方法得到的生成树就是最小生成树。

T和T以外的顶点之间的边的最小权值

我们令 V 表示顶点的集合。假设现在已经求得的生成树的顶点的集合是 X（⊂V），并且存在在 V 上的最小生成树使得 T 是它的一个子图。下面我们证明存在一棵最小生成树使得 T 是它的一个子图并且它包含了连接 X 和 V\X 之间的边中权值最小的边。记连接 X 和 V\X 的权值最小的边为 $e$，它连接着 V(∈X)和 $u$(∈V\X)。根据假设，存在一棵 V 上的最小生成树使得 T 是它的一个子图。如果 $e$ 也在这棵最小生成树上，问题就得到证明了，所以我们假设 $e$ 不在这棵树上。因为生成树本质是一棵树，所以在添加了 $e$ 之后就产生了圈。

圈上的边中，必然存在一条和 $e$ 不同的边 $f$ 连接着 X 和 V\X。从 $e$ 的定义可以知道 $f$ 的权值不会比 $e$ 小。因此，我们把 $f$ 从树中删除，然后加上 $e$ 就可以得到一棵新的生成树，并且总权值不超过原来的生成树。因此可以说存在同时包含 $e$ 和 T 的最小生成树。所以把 $e$ 加入 T 中满足最初的假设。可以这样不断地加入新的边，直到 X=V。因为存在 V 上的最小生成树使得 T 是它的一个子图，而 X=V，所以 T 就是 V 上的最小生成树。

让我们看一下如何查找最小权值的边。把X和顶点V连接的边的最小权值记为mincost[$v$]。在向X里添加顶点$u$时，只需要查看和$u$相连的边就可以了。对于每条边，更新mincost[$v$]=min(mincost[$v$], 边($u$,$v$)的权值)即可。

如果每次都遍历未包含在X中的点的mincost[$v$]，需要$O(|V|^2)$时间。不过和Dijkstra算法一样，如果使用堆来维护mincost时间复杂度就是$O(|E| \log |V|)$。

```
int cost[MAX_V][MAX_V];   // cost[u][v]表示边e=(u,v)的权值（不存在的情况下设为INF）
int mincost[MAX_V];       // 从集合X出发的边到每个顶点的最小权值
bool used[MAX_V];         // 顶点i是否包含在集合X中
int V;                    // 顶点数

int prim() {
  for (int i = 0; i < V; ++i) {
    mincost[i] = INF;
    used[i] = false;
  }
  mincost[0] = 0;
  int res = 0;

  while(true) {
    int v = -1;
    // 从不属于X的顶点中选取从X到其权值最小的顶点
    for (int u = 0; u < V; u++) {
      if (!used[u] && (v == -1 || mincost[u] < mincost[v])) v = u;
    }

    if (v == -1) break;
    used[v] = true;       // 把顶点v加入X
    res += mincost[v];    // 把边的长度加到结果里
```

```
    for (int u = 0; u < V; u++) {
      mincost[u] = min(mincost[u], cost[v][u]);
    }
  }
  return res;
}
```

### 2. 最小生成树问题2（Kruskal算法）

下面我们介绍Kruskal算法。Kruskal算法按照边的权值的顺序从小到大查看一遍，如果不产生圈（重边等也算在内），就把当前这条边加入到生成树中。至于这个算法为什么是正确的，其实和Prim算法证明的思路基本相同，在此就不详细说明了。

接下来我们介绍如何判断是否产生圈。假设现在要把连接顶点$u$和顶点$v$的边$e$加入生成树中。如果加入之前$u$和$v$不在同一个连通分量里，那么加入$e$也不会产生圈。反之，如果$u$和$v$在同一个连通分量里，那么一定会产生圈。可以使用并查集高效地判断是否属于同一个连通分量。

Kruskal算法在边的排序上最费时，算法的复杂度是$O(|E| \log |V|)$。

```
struct edge { int u, v, cost; };

bool comp(const edge& e1, const edge& e2) {
  return e1.cost < e2.cost;
}

edge es[MAX_E];
int V, E;                    // 顶点数和边数

int kruskal() {
  sort(es, es + E, comp);    // 按照edge.cost的顺序从小到大排列
  init_union_find(V);        // 并查集的初始化
  int res = 0;
  for (int i = 0; i < E; i++) {
    edge e = es[i];
    if (!same(e.u, e.v)) {
      unite(e.u, e.v);
      res += e.cost;
    }
  }
  return res;
}
```

## 2.5.6 应用问题

让我们试着用图论的算法解决问题。

## Roadblocks（POJ No.3255）

某街区共有 $R$ 条道路、$N$ 个路口。道路可以双向通行。问 1 号路口到 $N$ 号路口的次短路长度是多少？次短路指的是比最短路长度长的次短的路径。同一条边可以经过多次。

**限制条件**

- $1 \leqslant N \leqslant 5000$
- $1 \leqslant R \leqslant 100000$

**样例**

**输入**

N=4，R=4，图如下图所示

**输出**

450（1->2->4是长度为300的最短路，1->2->3->4是长度为450的次短路）

我们把路口看作顶点，把道路看作边的无向图。虽然用Dijkstra等算法可以简单地求得最短路，但是次短路应该怎么算呢？Dijkstra算法的思路是依次确定尚未确定的顶点中距离最小的顶点。按照这个思路对算法进行少许修改，就可以简单地求出次短路了。

到某个顶点$v$的次短路要么是到其他某个顶点$u$的最短路再加上$u \to v$的边，要么是到$u$的次短路再加上$u \to v$的边，因此所需要求的就是到所有顶点的最短路和次短路。因此，对于每个顶点，我们记录的不仅仅是最短距离，还有次短的距离。接下去只要用与Dijkstra算法相同的做法，不断更新这两个距离就可以求出次短路了。

```
// 输入
int N, R;
vector<edge> G[MAX_N];  // 图的邻接表表示

int dist[MAX_N];        // 最短距离
int dist2[MAX_N];       // 次短距离

void solve() {
  priority_queue<P, vector<P>, greater<P> > que;
```

```
  fill(dist, dist + N, INF);
  fill(dist2, dist2 + N, INF);
  dist[0] = 0;
  que.push(P(0, 0));

  while (!que.empty()) {
    P p = que.top(); que.pop();
    int v = p.second, d = p.first;
    if (dist2[v] < d) continue;
    for (int i = 0; i < G[v].size(); i++) {
      edge &e = G[v][i];
      int d2 = d + e.cost;
      if (dist[e.to] > d2) {
        swap(dist[e.to], d2);
        que.push(P(dist[e.to], e.to));
      }
      if (dist2[e.to] > d2 && dist[e.to] < d2) {
        dist2[e.to] = d2;
        que.push(P(dist2[e.to], e.to));
      }
    }
  }
  printf("%d\n", dist2[N - 1]);
}
```

## Conscription（POJ No.3723）

需要征募女兵 $N$ 人，男兵 $M$ 人。每征募一个人需要花费 10000 美元。但是如果已经征募的人中有一些关系亲密的人，那么可以少花一些钱。给出若干的男女之间的 1~9999 之间的亲密度关系，征募某个人的费用是 10000−（已经征募的人中和自己的亲密度的最大值）。要求通过适当的征募顺序使得征募所有人所需费用最小。

**限制条件**

- $1 \leqslant N, M \leqslant 10000$
- $0 \leqslant R \leqslant 50000$
- $0 < d < 10000$

**样例**

**输入**

```
N=5,M=5,R=8
关系有如下R个（(x,y,d)表示的是第x号男兵和第y号女兵之间的亲密度是d）

(x,y,d)={(4,3,6831),(1,3,4583),(0,0,6592),(0,1,3063),(3,3,4975),(1,3,2049),(4,2,2104),
(2,2,781)}
```

**输出**

```
71071
```

让我们设想一下这样一个无向图：在征募某个人a时，如果使用了a和b之间的关系，那么就连一条a到b的边。假设这个图中存在圈，那么无论以什么顺序征募这个圈上的所有人，都会产生矛盾。因此可以知道这个图是一片森林。反之，如果给了一片森林那么就可以使用对应的关系确定征募的顺序。

给定无向森林　　变成有向森林并加上顺序

因此，把人看作顶点，关系看作边，这个问题就可以转化为求解无向图中的最大权森林问题。最大权森林问题可以通过把所有边权取反之后用最小生成树的算法求解。[①]

```
// 输入
int N, M, R;
int x[MAX_R], y[MAX_R], d[MAX_R];

void solve() {
  V = N + M;
  E = R;
  for (int i = 0; i < R; i++) {
    es[i] = (edge){x[i], N + y[i], -d[i]};
  }
  printf("%d\n", 10000 * (N + M) + kruskal());
}
```

## Layout（POJ No.3169）

农夫约翰养了 $N$ 头牛，编号分别是 1 到 $N$。现在，它们要进食，按照编号顺序排成了一排。在它们之间有一些牛关系比较好，所以希望彼此之间不超过一定距离，也有一些牛关系比较不好，所以希望彼此之间至少要满足某个距离。此外，牛的性格比较犟，所以有可能有多头牛挤在同一个位置上。给出了 $ML$ 个关系好的牛的信息（$AL,BL,DL$）以及 $MD$ 个关系不好的牛的信息（$AD,BL,DD$）。这表示的是牛 $AL$ 与牛 $BL$ 之间的最大距离 $DL$ 和牛 $AD$ 与牛 $BD$ 之间的最小距离 $DD$。在满足这些条件的排列方法中，求 1 号牛和 $N$ 号牛之间的最大距离。如果不存在任何一种排列方法满足条件则输出−1。无限大的情况输出−2。

① 在这个问题中，完全没有用到男女之间的二分图结构。而在许多问题中，如果有特殊的结构，往往会考虑如何利用这个结构，当然也有像此题一样设置了无用的陷阱条件的题目，需要注意。

**限制条件**

- $2 \leqslant N \leqslant 1000$
- $1 \leqslant ML, MD \leqslant 10000$
- $1 \leqslant AL < BL \leqslant N$
- $1 \leqslant AD < BD \leqslant N$
- $1 \leqslant DL, DD \leqslant 1000000$

**样例**

**输入**

```
N=4, ML=2, MD=1
(AL, BL, DL)={(1,3,10),(2,4,20)}
(AD, BD, DD)={(2,3,3)}
```

**输出**

```
27(0,7,10,27)
```

记第$i$号牛的位置是$d[i]$。首先，牛是按照编号顺序排列的，所以有$d[i] \leqslant d[i+1]$成立。其次，对于每对关系好的牛之间的最大距离限制，都有$d[AL]+DL \geqslant d[BL]$成立。同样，对于每对关系不好的牛，都有$d[AD]+DD \leqslant d[BD]$成立。因此，原问题可以转化为在满足这三类不等式的情况下，求解$d$的$d[N]-d[1]$的最大值的问题。这是线性规划问题，可以使用单纯形法等较复杂的算法求解。但是这道题有更加简单的解法。

这些不等式的特点是所有的式子的两边都只出现了1个变量[①]。实际上，图上的最短路问题也可以用这样的形式表示出来。记从起点$s$出发，到各个顶点$v$的最短距离为$d(v)$。因此，对于每条权值为$w$的边$e=(v,u)$，都有$d(v)+w \geqslant d(u)$成立。反之，在满足全部这些约束不等式的$d$中，$d(v)-d(s)$的最大值就是从$s$到$v$的最短距离。需要注意这里不是最小值，而是最大值对应着最短距离。

把原来的问题和最短路问题进行比较就可以发现，两个问题都是完全一样的形式。也就是说，可以通过把原来的问题的每一个约束不等式对应成图中的一条边来构图，然后通过解决最短路问题来解决原问题。首先把顶点编号为$1 \sim N$。$d[i] \leqslant d[i+1]$变形为$d[i+1]+0 \geqslant d[i]$，因此从顶点$i+1$向顶点$i$连一条权值为0的边。同样$d[AL]+DL \geqslant d[BL]$对应从顶点$AL$向顶点$BL$连一条权值为$DL$的边，$d[AD]+DD \leqslant d[BD]$对应从顶点$BD$向顶点$AD$连一条权值为$-DD$的边。所求的问题是$d[N]-d[1]$的最大值，对应为顶点1到顶点$N$的最短距离。由于图中存在负权边，因此不使用Dijkstra算法而是使用Bellman-Ford算法求解。即使这样复杂度也只有$O(N(N+ML+MD))$，可以在规定时间内求解。

① 这种特殊形式的不等式方程组又叫做差分约束系统。——译者注

样例的输入对应的图

```
// 输入
int N, ML, MD;
int AL[MAX_ML], BL[MAX_ML], DL[MAX_ML];
int AD[MAX_MD], BD[MAX_MD], DD[MAX_MD];

int d[MAX_N]; // 最短距离

void solve() {
  fill(d, d + N, INF);
  d[0] = 0;

  // 用Bellman-Ford算法计算d
  for (int k = 0; k < N; k++) {
    // 从i+1到i的权值为0
    for (int i = 0; i + 1 < N; i++) {
      if (d[i + 1] < INF) d[i] = min(d[i], d[i + 1]);
    }
    // 从AL到BL的权值为DL
    for (int i = 0; i < ML; i++) {
      if (d[AL[i] - 1] < INF) {
        d[BL[i] - 1] = min(d[BL[i] - 1], d[AL[i] - 1] + DL[i]);
    }
    // 从BD到AD的权值为-DD
    for (int i = 0; i < MD; i++) {
      if (d[BD[i] - 1] < INF) {
        d[AD[i] - 1] = min(d[AD[i] - 1], d[BD[i] - 1] - DD[i]);
    }
  }

  int res = d[N - 1];
  if (d[0] < 0) {
    // 存在负圈则无解.
    res = -1;
  } else if (res == INF) {
    res = -2;
  }
  printf("%d\n", res);
}
```

# 2.6 数学问题的解题窍门

➠数学，特别是数论与计算机科学有着密切的联系，所以也常被选作题材。虽然数学问题大多需要使用特定方法求解，但其中有几个基础算法扮演着重要的角色。

## 2.6.1 辗转相除法

### 1. 求最大公约数

让我们来看一下如下问题。

**线段上格点的个数**

给定平面上的两个格点[①]$P_1=(x_1, y_1)$和$P_2=(x_2, y_2)$，线段$P_1P_2$上，除$P_1$和$P_2$以外一共有几个格点？

⚠限制条件

- $-10^9 \leqslant x_1, x_2, y_1, y_2 \leqslant 10^9$

**样例**

输入

```
P1 = (1, 11), P2 = (5, 3)
```

输出

```
(2, 9)、(3, 7)、(4, 5)三个点
```

检查所有满足$\min(x_1, x_2) \leqslant x \leqslant \max(x_1, x_2)$且$\min(y_1, y_2) \leqslant y \leqslant \max(y_1, x_2)$的格点虽然可以得到正确的答案，但复杂度却是$O(|x_1-x_2|\times|y_1-y_2|)$，对坐标的绝对值较大的情况难以处理。其实这道题的答案如右图所示，是$|x_1-x_2|$和$|y_1-y_2|$的最大公约数−1（要注意特判$|x_1-x_2|=0$且$|y_1-y_2|=0$时的答案是0）。

那么，该怎样计算最大公约数呢？虽然可以从1开始依次检查是否能整除，这也比$O(|x_1-x_2|\times|y_1-y_2|)$要快得多了，不过在此要介绍给大家一个更为快速的辗转相除法。

① 格点是指横纵坐标均为整数的点。——译者注

小三角形的边长是(4/gcd(4,8), 8/gcd(4,8))

设gcd($a$, $b$)是计算自然数$a$和$b$的最大公约数的函数，$a$除$b$得到的商和余数分别为$p$和$q$。因为$a=b\times p+q$，所以gcd($b$, $q$)既整除$a$又整除$b$，也就整除gcd($a$, $b$)。反之，因为$q=a-b\times p$，同理可证gcd($a$, $b$)整除gcd($b$, $q$)。因此可以知道gcd($a$, $b$)=gcd($b$, $a$ % $b$)。不断这样操作下去，由于gcd的第二个参数总是不断减小的，最终会得到gcd($a$, $b$)=gcd($c$, 0)。0和$c$的最大公约数是$c$，所以gcd($c$, 0)=$c$，这样就计算出了gcd($a$, $b$)。辗转相除法的程序实现如下所示。

用最小的正方形可以无缝隙地填满整个大长方形

```
int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}
```

## 2. 复杂度

接下来让我们来估算辗转相除法的复杂度。假设$a$和$b$是两个自然数。如果$b>a$，就有gcd($b$, $a$ % $b$)=gcd($b$, $a$)，经过一次递归以后就变成了$a>b$，所以不妨假设$a>b$。这时函数会按照gcd($a$, $b$) → gcd($b$,

$a \% b) \to \gcd(a \% b, b \% (a \% b))$这样递归下去。当$b>a/2$时有$a\%b=a-b<a/2$，当$b<a/2$时则$a\%b<b<a/2$，于是经过两次递归后，第一个参数要小于原来的一半。所以其复杂度在$O(\log \max(a, b))$以内。虽然这只是粗略的估计，但已经足以证明辗转相除法是非常高效的了。

### 3. 扩展欧几里德算法[1]

对辗转相除法做一些扩展，就能求解如下问题。

**双六[2]**

一个双六上面有向前向后无限延续的格子，每个格子都写有整数。其中 0 号格子是起点，1 号格子是终点。而骰子上只有 $a, b, -a, -b$ 四个整数，所以根据 $a$ 和 $b$ 的值的不同，有可能无法到达终点。

双六

掷出四个整数各多少次可以到达终点呢？如果解不唯一，输出任何一组皆可。如果无解，输出–1。

**限制条件**

- $1 \leqslant a, b \leqslant 10^9$

**样例**

**输入**

```
a = 4, b = 11
```

**输出**

```
3001（3×a-1×b=1）
```

这个问题用数学语言表述就是“求整数$x$和$y$使得$ax+by=1$”。可以发现，如果$\gcd(a, b)\neq 1$，显然无解。反之，如果$\gcd(a, b)=1$，就可以通过扩展原来的辗转相除法来求解。事实上，一定存在整数对$(x, y)$使得$ax+by=\gcd(a, b)$，并可以用同样的算法求得。

设`int extgcd(int a, int b, int& x, int& y)`是求解该方程的函数，其返回值是$\gcd(a, b)$。与gcd一样，我们可以递归地定义extgcd。假设已经求得了

① 按照习惯，这里将Euclidean algorithm译作辗转相除法，而将Extended Euclidean algorithm译作扩展欧几里德算法。——译者注

② 双六是一种类似大富翁的桌上游戏。——译者注

$$bx'+(a\%b)y'=\gcd(a, b)$$

的整数解$x'$和$y'$。再将

$$a\%b=a-(a/b)\times b$$

代入后就得到

$$ay'+b(x'-(a/b)\times y')=\gcd(a, b)$$

而当$b=0$时则有

$$a\times 1+b\times 0=a=\gcd(a, b)$$

将上述数学语言转化成代码后，就得到了如下程序。

```
int extgcd(int a, int b, int& x, int& y) {
  int d = a;
  if (b != 0) {
    d = extgcd(b, a % b, y, x);
    y -= (a / b) * x;
  } else {
    x = 1; y = 0;
  }
  return d;
}
```

### 4. ax+by=gcd(a,b)的解的大小

只要看一下递归的方法就能知道，extgcd的复杂度和gcd的复杂度是相同的。但该函数所求的$ax+by=\gcd(a,b)$的解的大小又如何呢？事实上，如果$ab\neq 0$，可以知道$|x|\leqslant b$且$|y|\leqslant a$。下面用归纳法来证明这一结论。

在$b=0$的前一步，即$a\%b=0$时有$x=0$且$y=1$，结论显然成立。假设调用extgcd($b$, $a$ % $b$, $y'$, $x'$)后有$|x'|\leqslant b$且$|y'|\leqslant a\%b$。在extgcd($a$, $b$, $x$, $y$)中$x=x'$, $y=y'-(a/b)x'$，所以有如下不等式成立。

$$|x|=|x'|\leqslant b,\ |y|=|y'-(a/b)x'|\leqslant|y'|+(a/b)\times|x'|\leqslant a\%b+(a/b)\times b=a$$

**专栏　关于证明与规律**

- 程序设计竞赛中，常常需要证明自己针对问题的设想是否成立。虽说是证明，但不光可以从数学上严密地证明，还可以通过尝试一些样例来验证设想是否成立，或是利用历史积累的已知结论，或是凭借直觉勇往直前。能够轻而易举地证明或是已经了解既有结论的情况除外。不过时间紧迫又缺少证明思路时，建议大家至少要测试一些数据来验证。最好不要单凭直觉盲目冒进。另外，有些规律虽然对一般情况成立，却不能很好处理一些边界情况。在本节最开始的问题中，线段两端点重合的情况就是这样一个例子。即使找到了规律，也不要就此大意，最好充分考虑一下是否有特殊情况或反例。

### 2.6.2 有关素数的基础算法

素数广泛应用于密码学中，因而也有很多相关算法。不过程序设计竞赛涉及的主要是埃氏筛法、简单的素性测试和整数分解这类算法。

#### 1. 素性测试

**素数判定**

给定整数 $n$，请判断 $n$ 是不是素数。

**限制条件**

- $1 \leqslant n \leqslant 10^9$

**样例 1**

**输入**

```
53
```

**输出**

```
Yes
```

**样例 2**

**输入**

```
295927
```

**输出**

```
No（295927=541×547）
```

所谓素数，是指恰好有2个约数的整数。因为$n$的约数都不超过$n$，所以只要检查2~$n$−1的所有整数是否整除$n$就能判定$n$是不是素数。在此，如果$d$是$n$的约数，那么$n/d$也是$n$的约数。由$n=d\times n/d$可知$\min(d, n/d)\leqslant\sqrt{n}$，所以只要检查$2\sim\sqrt{n}$的所有整数就足够了。同理可知，整数分解和约数枚举都可以在$O(\sqrt{n})$时间完成。虽然还有更为高效的算法[①]，不过多数情况下这已经足够了。

① 费马测试、ρ算法、数域筛法等。

```
// 假设输入都是正数
// 素性测试O(√n)
bool is_prime(int n) {
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) return false;
  }
  return n != 1; // 1是例外
}

// 约数枚举O(√n)
vector<int> divisor(int n) {
  vector<int> res;
  for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
      res.push_back(i);
      if (i != n / i) res.push_back(n / i);
    }
  }
  return res;
}

// 整数分解O(√n)
map<int, int> prime_factor(int n) {
  map<int, int> res;
  for (int i = 2; i * i <= n; i++) {
    while (n % i == 0) {
      ++res[i];
      n /= i;
    }
  }
  if (n != 1) res[n] = 1;
  return res;
}
```

### 2. 埃氏筛法

如果只对一个整数进行素性测试，通常$O(\sqrt{n})$的算法就足够了。但如果要对许多整数进行素性测试，则有更为高效的算法。我们来看一下如下问题。

**素数的个数**

给定整数$n$，请问$n$以内有多少个素数？

**限制条件**

- $n \leqslant 10^6$

**样例 1**

**输入**

```
11
```

**输出**

```
5(2、3、5、7、11共5个素数)
```

**样例 2**

**输入**

```
1000000
```

**输出**

```
78498
```

要枚举$n$以内素数，可以用埃氏筛法。这是一个与辗转相除法一样古老的算法。

首先，将2到$n$范围内的所有整数写下来。其中最小的数字2是素数。将表中所有2的倍数都划去。表中剩余的最小数字是3，它不能被更小的数整除，所 以是素数。再将表中所有3的倍数都划去。依此类推，如果表中剩余的最小数字是$m$时，$m$就是素数。然后将表中所有$m$的倍数都划去。像这样反复操作，就能依次枚举$n$以内的素数。

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | - | 5 | - | 7 | - | 9 | - | 11 | - | 13 | - | 15 | - | 17 | - | 19 | - |
| 2 | 3 | - | 5 | - | 7 | - | - | - | 11 | - | 13 | - | - | - | 17 | - | 19 | - |

```
int prime[MAX_N];           // 第i个素数
bool is_prime[MAX_N + 1];   // is_prime[i]为true表示i是素数

// 返回n以内素数的个数
int sieve(int n) {
  int p = 0;
  for (int i = 0; i <= n; i++) is_prime[i] = true;
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; i++) {
    if (is_prime[i]) {
      prime[p++] = i;
      for (int j = 2 * i; j <= n; j += i) is_prime[j] = false;
    }
  }
  return p;
}
```

埃氏筛法的复杂度仅有$O(n\log\log n)$。对于程序设计竞赛中的数据规模，将它的复杂度看作大致是线性的也无妨。

### 3. 区间筛法

**区间内素数的个数**

给定整数 $a$ 和 $b$，请问区间$[a, b)$内有多少个素数？

**限制条件**

- $a < b \leqslant 10^{12}$
- $b - a \leqslant 10^{6}$

**样例 1**

**输入**

```
22 37
```

**输出**

```
3(23、29、31共3个素数)
```

**样例 2**

**输入**

```
a = 22801763489, b = 22801787297
```

**输出**

```
1000
```

区间$[a, b)$指的是所有满足$a \leqslant x < b$的整数（根据背景也可能是实数）所构成的集合[①]。

在素性判定这一小节中已经讲过，$b$以内的合数的最小质因数一定不超过$\sqrt{b}$。如果有$\sqrt{b}$以内的素数表的话，就可以把 埃氏筛法运用在$[a, b)$上了。也就是说，先分别做好$[2, \sqrt{b})$的表和$[a, b)$的表，然后从$[2, \sqrt{b})$的表中筛得素数的同时，也将其倍数从$[a, b)$的表中划去，最后剩下的就是区间$[a, b)$内的素数了。

① 作为题外话，表示区间的时候，用左闭右开的形式往往更方便。STL和Java标准库（如iterator和substring）中的区间也大多是左闭右开的。

```
typedef long long ll;

bool is_prime[MAX_L];
bool is_prime_small[MAX_SQRT_B];

// 对区间[a, b)内的整数执行筛法。is_prime[i - a] = true ⇔ i是素数
void segment_sieve(ll a, ll b) {
  for (int i = 0; (ll)i * i < b; i++) is_prime_small[i] = true;
  for (int i = 0; i < b - a; i++) is_prime[i] = true;

  for (int i = 2; (ll)i * i < b; i++) {
    if(is_prime_small[i]) {
      for (int j = 2 * i; (ll)j * j < b; j += i) is_prime_small[j] = false;  // 筛[2, √b )
      for (ll j = max(2LL, (a + i - 1) / i) * i; j < b; j += i) is_prime[j - a] = false;
       // 筛[a, b)
    }
  }
}
```

## 2.6.3 模运算

### 1. 为什么需要求余数

在程序设计竞赛中，如果计算结果超出了64位整数的范围，则可能会要求输出结果对合适的数取模后的余数。这样可以消除在高精度计算方面，语言的差异所带来的不利因素。例如，Java中有支持高精度计算的类BigInteger，而C++和C则不得不靠自己实现。另一方面，高精度乘法不同实现的复杂度也不一样，因此，对算法本身的评价变得更为困难。正因为此类理由，程序设计竞赛中经常出现余数的计算。

### 2. 基本的模运算

计算除以$m$的余数，可以说成是“对$m$取模”或“以$m$为模”。接下来我们都统一用“对$m$取模”。为了使表述更简单，我们将$a$和$b$除以$m$后所得的余数相等记作$a \equiv b \pmod m$，又将$a$除以$m$所得的余数记作$a \bmod m$，并规定$0 \leqslant a \bmod m \leqslant m-1$。在某些环境中，$a$是负数时$a \% m$的结果也是负的，此时改为$a \% m + m$就能保证结果在$0 \sim m-1$的范围内[①]。假设$a \equiv c \pmod m$且$b \equiv d \pmod m$，那么有以下基本的模运算律成立。

$$a+b \equiv c+d \pmod m$$
$$a-b \equiv c-d \pmod m$$
$$a \times b \equiv c \times d \pmod m$$

实际问题中$m$可能达到$10^9$的规模，计算$a \times b$时，虽然余数可以保存在32位整数中，但求模以前的结果可能无法保存在32位整数中，需要注意避免发生溢出。当有发生溢出的潜在危险时，从最开

① 当然，如果恰好整除的话，结果是0，不需要加$m$。——译者注

始就将所有的变量设为64位整数类型不失为一个好办法。

在模运算的世界中可以很自然地计算+、−、×，但对于除法却需要多加注意。例如，虽然$2\equiv 8 \pmod 6$，但是$2/2=1\not\equiv 4=8/2 \pmod 6$。当$a\times c\equiv b\times c \pmod m$时，$(a-b)\times c$可以被$m$整除。假设$d=\gcd(c, m)$，$(a-b)\times(c/d)$就可以被$m/d$整除，又因为$c/d$和$m/d$互素，所以$a-b$被$m/d$整除，即$a\equiv b \pmod{m/\gcd(m, c)}$。虽然≡构成的表达式和=构成的表达式基本支持相同的运算，不过也有像这样，需要回到定义，当作$a-b$整除$m$来理解的情况。

## 2.6.4　快速幂运算[①]

除数学问题之外，也有很多地方用到了幂运算。在此，给大家介绍一种能够非常高效地计算幂运算的快速幂运算算法——反复平方法。

### Carmichael Numbers（UVa No.10006）

我们把对任意的 $1<x<n$ 都有 $x^n\equiv x \pmod n$成立的合数 $n$ 称为 Carmichael Number。对于给定的整数 $n$，请判断它是不是 Carmichael Number。

**限制条件**

- $2 < n < 65000$

**样例 1**

**输入**

```
17
```

**输出**

```
No(17是素数)
```

**样例 2**

**输入**

```
561
```

**输出**

```
Yes
```

① 这里所介绍的是Exponentiation by Squaring最基本的算法$2^k$-ary方法。在Knuth所著的TAOCP中，该算法被称为Russian Peasant algorithm。——译者注

样例 3

**输入**

```
4
```

**输出**

```
NO ( 2^4≡ 0(mod 4) )
```

此题中，有$n$个待检查的数，如果每个数都按定义$O(n)$复杂度来计算幂，则总的复杂度为$O(n^2)$，不能满足要求。让我们来考虑加速幂运算的方法。如果$n=2^k$，可以将其表示为

$$x^n = ((x^2)^2)\cdots$$

只要做$k$次平方运算就可以轻松求得。由此我们可以想到，先将$n$表示为2的幂次的和。

$$n = 2^{k_1} + 2^{k_2} + 2^{k_3} \cdots$$

就有

$$x^n = x^{2^{k_1}} x^{2^{k_2}} x^{2^{k_3}} \cdots$$

只要在依次求 $x^{2^i}$ 的同时进行计算就好了，最终得到了$O(\log n)$计算幂运算的算法。大家不妨自己选择合适的数字模拟一下以便加深理解。

$x^{22}=x^{16}\times x^4\times x^2$

(22转成二进制数是10110)

```
typedef long long ll;

ll mod_pow(ll x, ll n, ll mod) {
  ll res = 1;
  while (n > 0) {
    if (n & 1) res = res * x % mod;  // 如果二进制最低位为1，则乘上x^(2^i)
    x = x * x % mod;                 // 将x平方
    n >>= 1;
  }
  return res;
}
```

也可以像下面这样来理解。当$n$为偶数时有$x^n=((x^2)^{(n/2)})$，递归转为$n/2$的情况。$n$为奇数时有$x^n=((x^2)^{(n/2)})\times x$，同样也递归转为$n/2$的情况。这样不断递归下去，每次$n$都减半，于是可以在$O(\log n)$时间内完成幂运算。

```
ll mod_pow(ll x, ll n, ll mod) {
  if (n == 0) return 1;
  ll res = mod_pow(x * x % mod, n / 2, mod);
  if (n & 1) res = res * x % mod;
  return res;
}
```

事实上，这道题不用幂运算也能求解。对于给定的$n$，存在$O(\sqrt{n})$时间的判定方法。另外，还有先通过$O(n)$时间预处理，再对每个查询在$O(1)$时间内判定的算法。请大家试着思考一下。

# 2.7 一起来挑战 GCJ 的题目（1）

➠让我们一起运用迄今所介绍的技巧，实际挑战一下GCJ的题目吧。

## 2.7.1 Minimum Scalar Product

**Minimum Scalar Product（2008 Round1A A）**

有两个向量 $v_1=(x_1, x_2, ..., x_n)$和 $v_2=(y_1, y_2, \cdots, y_n)$，允许任意交换 $v_1$ 和 $v_2$ 各自的分量的顺序。请计算 $v_1$ 和 $v_2$ 的内积 $x_1y_1+\cdots+x_ny_n$ 的最小值。

**限制条件**

Small
- $1 \leqslant n \leqslant 8$
- $-1000 \leqslant x_i, y_i \leqslant 1000$

Large
- $100 \leqslant n \leqslant 800$
- $-100000 \leqslant x_i, y_i \leqslant 100000$

**样例 1**

输入

```
n = 3
v1 = (1, 3, -5)
v2 = (-2, 4, 1)
```

输出

```
令v1 = (-5, 1, 3), v2 = (4, 1, -2)就可以得到最小值v1 × v2 = -25
```

**样例 2**

输入

```
n = 5
v1 = (1, 2, 3, 4, 5)
v2 = (1, 0, 1, 0, 1)
```

**输出**

```
令v1 = (1, 2, 3, 4, 5), v2 = (1, 1, 1, 0, 0)就可以得到最小值6
```

$v_1$和$v_2$各自的分量的顺序都可以任意交换，因此可以先把$v_1$的顺序固定下来只交换$v_2$的顺序。为了方便分析先将$v_1$按升序排好序。接下来枚举$v_2$的分量所有的排列顺序，一共有$n!$种排列，还需要对每种排列计算内积，总的复杂度是$O(n!\times n)$。这个算法在Small的情况，因为$n \leqslant 8$所以没有问题，但在Large的情况时就远远不够了。

在此隐约会觉得把$v_2$按降序或者升序排序的话，所得的内积是最小的。事实上，如果将$v_2$按降序排序的话，所给的两个样例都能够得到最小值。这个设想的确是对的，$v_1$和$v_2$的内积在将$v_1$按升序，将$v_2$按降序排序时取得最小值。下面我们来证明这一设想。首先考虑$n$=2的情况。

考虑$v_1=(x_1, x_2)$, $v_2=(y_1, y_2)$，假设$v_1$已经按升序排好序了，即有$x_1 \leqslant x_2$，比较$x_1 \times y_1 + x_2 \times y_2$和$x_2 \times y_1 + x_1 \times y_2$的大小关系。

$$x_1 \times y_1 + x_2 \times y_2 - x_2 \times y_1 - x_1 \times y_2 = x_1 (y_1 - y_2) + x_2 (y_2 - y_1) = (x_1 - x_2) (y_1 - y_2)$$

由此可知，$y_1 \geqslant y_2 \Leftrightarrow x_1 \times y_1 + x_2 \times y_2 \leqslant x_2 \times y_1 + x_1 \times y_2$。因此$n$=2时结论成立。

接下来考虑$n$大于2的情况。如果$v_2$不是按降序排序的，那么存在$i<j$使得$y_i<y_j$，根据对$n$=2的情况的分析可以知道，交换$y_i$和$y_j$后就得到了更小的内积。因此，当将$v_2$按降序排序时，所得的内积最小。

数组排序的复杂度为$O(n\log n)$，所以发现这一结论后，只要做两次排序就可以简单高效地计算答案了。那么，要怎样才能去想到这个设想呢？

首先就是靠直觉。比较容易想到的姑且先排序试试看。

第二个就是样例。原本的问题描述中并没有说明$v_1$和$v_2$在什么情况下取得最小值。但是，因为样例的规模比较小，所以可以手工验算。因而可以找到使得内积最小的$v_1$和$v_2$。通过观察对应的$v_1$和$v_2$，想要得到刚才的结论也并不是那么难。

第三个就是像证明中的那样，从像$n$=2这样小规模的情况出发，推广到一般的情况从而证明结论。在这个问题中，很容易证明对$n$=2的情况结论成立，接着同样可以证明对一般的情况结论都成立。

最后需要注意的是，即使推导出了正确的算法，如果程序的实现中有漏洞的话也是徒劳。在这道题中，$v_1$的分量和$v_2$的分量的乘积可能会导致32位整数溢出。即使认为找到了正解，不到最后提交结果正确的那一刻，都不能掉以轻心呢。

```
typedef long long ll;

// 输入
int n;
int v1[MAX_N], v2[MAX_N];

void solve() {
  sort(v1, v1 + n);
  sort(v2, v2 + n);
  ll ans = 0;
  for (int i = 0; i < n; i++) ans += (ll)v1[i] * v2[n - i - 1];
  printf("%lld\n", ans);
}
```

## 2.7.2　Crazy Rows

**Crazy Rows（2009 Round2 A）**

给定一个由 0 和 1 组成的矩阵。只允许交换相邻的两行（第 $i$ 行和第 $i$+1 行），要把矩阵化成下三角矩阵（主对角线上方的元素都是 0），最少需要交换几次?输入的矩阵保证总能化成下三角矩阵。

**限制条件**

Small

- $1 \leqslant N \leqslant 8$

Large

- $1 \leqslant N \leqslant 40$

**样例 1**

**输入**

```
N = 2
矩阵
10
11
```

**输出**

```
0(输入已经是下三角矩阵)
```

**样例 2**

**输入**

```
N = 3

矩阵
001
100
010
```

**输出**

```
2(交换第1行和第2行，再交换第2行和第3行，得到下三角矩阵)
```

**样例 3**

**输入**

```
N = 4

矩阵
1110
1100
1100
1000
```

**输出**

```
4
```

最先想到的是尝试所有$N!$种交换方案。但在Large中，由于最大的$N$=40，这当然是行不通的。

暂且先考虑一下最后应该把哪一行交换到第1行。最后的第1行应该具有00...0或是10...0的形式。可以交换到第1行的行当然也可以交换到第2及之后的行，当有多个满足条件的行时，选择离第1行近的行对应的最终费用要小。大家肯定都已注意到了这一点吧。有兴趣的读者不妨自己证明一下。

确定第1行之后，就没有必要再移动它了，于是对于之后的行就可以以同样的思路处理。

在这道题中，每行的0和1的位置并不重要，只要知道每行最后一个1所在的位置就足够了。如果先将这些位置预先计算好，那么就能降低行交换时的复杂度。直接按矩阵的形式处理的复杂度是$O(N^3)$，而预先计算后再处理的复杂度降为$O(N^2)$。

```
// 输入
int N;
int M[MAX_N][MAX_N];  // 矩阵

int a[MAX_N];         // a[i]表示第i行最后出现的1的位置——1~n-1

void solve() {
  int res = 0;
  for (int i = 0; i < N; i++) {
    a[i] = -1;  // 如果第i行不含1的话，就当作-1
    for (int j = 0; j < N; j++) {
      if (M[i][j] == 1) a[i] = j;
    }
  }
  for (int i = 0; i < N; i++) {
    int pos = -1;    // 要移动到第i行的行
    for (int j = i; j < N; j++) {
      if (a[j] <= i) {
        pos = j;
        break;
      }
    }

    // 完成交换
    for (int j = pos; j > i; j--) {
      swap(a[j], a[j - 1]);
      res++;
    }
  }
  printf("%d\n", res);
}
```

## 2.7.3　Bribe the Prisoners

**Bribe the Prisoners（2009 Round 1C C）**

如下图所示，一个监狱里有 $P$ 个并排着的牢房。从左至右依次编号为 $1, 2, \cdots, P$。最初所有的牢房里都住着一个囚犯。相邻的两个牢房之间有一个窗户，可以通过它与相邻牢房里的囚犯对话。

监狱的情况

现在要释放一些囚犯。如果释放某个牢房里的囚犯，其相邻的牢房里的囚犯就会知道，因而发怒暴动。所以，释放某个牢房里的囚犯同时，必须要贿赂两旁相邻牢房里的囚犯一枚金币。

另外，为了防止释放的消息在相邻牢房间传开，不仅两旁直接相邻的牢房，所有可能听到消息的囚犯，即直到空牢房为止或直到监狱两端为止,此间的所有囚犯都必须给一枚金币。

释放后给金币的例子

现在要释放 $a_1, a_2, \cdots, a_Q$ 号牢房里的 $Q$ 名囚犯，释放的顺序还没确定。如果选择所需金币数量尽量少的顺序释放，最少需要多少枚金币？

**限制条件**

- $1 \leqslant N \leqslant 100$
- $Q \leqslant P$

Small

- $1 \leqslant P \leqslant 100$
- $1 \leqslant Q \leqslant 5$

Large

- $1 \leqslant P \leqslant 10000$
- $1 \leqslant Q \leqslant 100$

**样例 1**

**输入**

```
P = 8, Q = 1, A = {3}
```

**输出**

```
7（必须要给剩下的7个人一枚金币）
```

**样例 2**

**输入**

```
P = 20, Q = 3, A = {3, 6, 14}
```

**输出**

```
35(按照牢房14、牢房6、牢房3的顺序释放，则需要19 + 12 + 4即35枚金币，是最少的)
```

这道题的关键是，释放了某个囚犯之后，就把连续的牢房分成了两段，此后这两段就相互独立了。

释放后分成两部分的情况

释放上图中☆里的囚犯时

- 此时所需的金币数量
- 释放左侧部分（①）所需的金币总数
- 释放右侧部分（②）所需的金币总数

这三者的总和就是所需的金币总数。只要不断递归地枚举最初释放的囚犯并计算对应的金币总数，就能求出答案了。

这里，递归计算过程中作为计算对象的连续部分，其两端是空牢房或监狱两端。因此，作为计算对象的连续部分一共有$O(Q^2)$个。所以，利用动态规划按顺序计算，就能够在$O(Q^3)$时间内求解。

```
// 输入
int P, Q, A[MAX_Q + 2];  // A中保存输入数据，下标从1开始

// dp[i][j] := 释放(i, j)所需的金币①
int dp[MAX_Q + 1][MAX_Q + 2];

void solve() {
  // 为了处理方便，将两端加入A中②
  A[0] = 0;
  A[Q + 1] = P + 1;

  // 初始化
  for (int q = 0; q <= Q; q++) {
    dp[q][q + 1] = 0;
  }

  // 从短的区间开始填充dp
  for (int w = 2; w <= Q + 1; w++) {
    for (int i = 0; i + w <= Q + 1; i++) {
      // 计算dp[i][j]
      int j = i + w, t = INT_MAX;
```

① dp[i][j]表示的是，将从A[i]号囚犯到A[j]号囚犯（不含两端的囚犯）的连续部分里的所有囚犯都释放时,所需的最少金币总数。

② 为了更方便地处理两端的情况，我们把左端当成0号囚犯，右端当成Q+1号囚犯。这样，dp[0][Q+1]就是答案。

```
        // 枚举最初释放的囚犯，计算最小的费用
        for (int k = i + 1; k < j; k++) {
          t = min(t, dp[i][k] + dp[k][j]);
        }

        // 最初的释放还需要与所释放囚犯无关的A[j]-A[i]-2枚金币
        dp[i][j] = t + A[j] - A[i] - 2;
      }
    }

    printf("%d\n", dp[0][Q + 1]);
  }
```

## 2.7.4　Millionaire

**Millionaire（2008 APAC local onsites C）**

你被邀请到某个电视节目中去玩下面这个游戏。一开始你有 $x$ 元钱，接着进 $M$ 轮赌博。每一轮，可以将所持的任意一部分钱作为赌注。赌注不光可以是整数，也可以是小数。一分钱不押或全押都没有关系。每一轮都有 $P$ 的概率可以赢，赢了赌注就会翻倍，输了赌注就没了。如果你最后持有 1000000 元以上的钱的话，就可以把这些钱带回家。请计算当你采取最优策略时，获得 1000000 元以上的钱并带回家的概率。

**限制条件**

- $0 \leqslant P \leqslant 1.0$
- $1 \leqslant X \leqslant 1000000$

Small

- $1 \leqslant M \leqslant 5$

Large

- $1 \leqslant M \leqslant 15$

**样例 1**

**输入**

```
M = 1, P = 0.5, X = 500000
```

**输出**

```
0.500000(一开始便全押)
```

**样例 2**

**输入**

```
M = 3, P = 0.75, X = 600000
```

**输出**

```
0.843750
```

### 1. 该问题的难点

“连续性”是这个问题的一大特点。每一轮可押的赌注不一定非是整数，因而有无限种可能，所以完全无法穷竭搜索。

### 2. 化连续为离散

但是，认真思考一下就会发现，我们只需要检查“无限种可能”中的“有限种可能”就足够了，因而可以设计对应的算法。首先来思考一下最后一轮会出现哪些情况。

- 如果你持有1000000元以上的钱，就没有再赌的必要了。有1的概率可以带钱回家。
- 如果你持有500000元以上的钱，不妨全押了。有*P*的概率可以带钱回家。虽然不全押也是可以的，不过因为要求要达到1000000元，所以不论怎样都是这轮赢了就能带钱回家，输了就不能带钱回家。
- 如果你持有不到500000元的钱，那已经无可奈何了。有0的概率可以带钱回家。

最后一轮带钱回家的概率

虽然赌注是连续的（有无限种可能），但实际的概率是像上面这样的阶梯函数。因此，只需考虑处于这三个范围中的哪一个就可以了。

那么，最后两轮时又如何呢？同样地，这次也分为5种情况。

最后两轮带钱回家的概率

某个范围中，即使所持的钱数不同，最后可以带钱回家的概率也是完全一样的。而且同样地，$M$轮时只要考虑$2^M+1$种情况就足够了。这样，就可以设计出穷竭搜索的算法了。

### 3. 动态规划

接下来，将穷竭搜索改成动态规划，就能够求解Large了。

```
// 输入
int M, X;
double P;

double dp[2][(1 << MAX_M) + 1];

void solve() {
  int n = 1 << M;

  double *prv = dp[0], *nxt = dp[1];
  memset(prv, 0, sizeof(double) * (n + 1));
  prv[n] = 1.0;

  for (int r = 0; r < M; r++) {
    for (int i = 0; i <= n; i++) {
      int jub = min(i, n - i);
      double t = 0.0;
      for (int j = 0; j <= jub; j++) {
        t = max(t, P * prv[i + j] + (1 - P) * prv[i - j]);
      }
      nxt[i] = t;
    }
    swap(prv, nxt);
  }

  int i = (ll)X * n / 1000000;
  printf("%.6f\n", prv[i]);
}
```

## 练 习 题

### 2.1 最基础的“穷竭搜索”

■ 深度优先搜索

POJ 1979: Red and Black
AOJ 0118: Property Distribution
AOJ 0033: Ball
POJ 3009: Curling 2.0

■ 广度优先搜索

AOJ 0558: Cheese
POJ 3669: Meteor Shower
AOJ 0121: Seven Puzzle

■ 穷竭搜索

POJ 2718: Smallest Difference
POJ 3187: Backward Digit Sums
POJ 3050: Hopscotch
AOJ 0525: Osenbei

### 2.2 一往直前！贪心法

■ 区间

POJ 2376: Cleaning Shifts
POJ 1328: Radar Installation
POJ 3190: Stall Reservations

■ 其他

POJ 2393: Yogurt factory
POJ 1017: Packets
POJ 3040: Allowance
POJ 1862: Stripies
POJ 3262: Protecting the Flowers

### 2.3 记录结果再利用的“动态规划”

■ 基础的动态规划算法

POJ 3176: Cow Bowling
POJ 2229: Sumsets
POJ 2385: Apple Catching
POJ 3616: Milking Time
POJ 3280: Cheapest Palindrome

■ 优化递推关系式

POJ 1742: Coins
POJ 3046: Ant Counting
POJ 3181: Dollar Dayz

■ 需稍加思考的题目

POJ 1065: Wooden Sticks
POJ 1631: Bridging signals
POJ 3666: Making the Grade
POJ 2392: Space Elevator
POJ 2184: Cow Exhibition

### 2.4 加工并存储数据的数据结构

■ 优先队列

POJ 3614: Sunscreen
POJ 2010: Moo University - Financial Aid

■ 并查集

POJ 2236: Wireless Network
POJ 1703: Find them, Catch them
AOJ 2170: Marked Ancestor

### 2.5 它们其实都是“图”

■ 最短路

AOJ 0189: Convenient Location
POJ 2139: Six Degrees of Cowvin Bacon
POJ 3259: Wormholes
POJ 3268: Silver Cow Party
AOJ 2249: Road Construction
AOJ 2200: Mr. Rito Post Office

■ 最小生成树

POJ 1258: Agri-Net
POJ 2377: Bad Cowtractors
AOJ 2224: Save your cat
POJ 2395: Out of Hay

### 2.6 数学问题的解题窍门

■ 辗转相除法

AOJ 0005: GCD and LCM
POJ 2429: GCD & LCM Inverse
POJ 1930: Dead Fraction

■ 素数

AOJ 0009: Prime Number
POJ 3126: Prime Path
POJ 3421: X-factor Chains
POJ 3292: Semi-prime H-numbers

■ 快速幂运算

POJ 3641: Pseudoprime numbers
POJ 1995: Raising Modulo Numbers

# 第 3 章
# 出类拔萃——中级篇

# 3.1 不光是查找值！“二分搜索”

➠二分搜索法，是通过不断缩小解可能存在的范围，从而求得问题最优解的方法。在程序设计竞赛中，经常可以见到二分搜索法和其他算法结合的题目。接下来，给大家介绍几种经典的二分搜索法的问题。

## 3.1.1 从有序数组中查找某个值

**lower_bound**

给定长度为 $n$ 的单调不下降数列 $a_0,\cdots a_{n-1}$ 和一个数 $k$，求满足 $a_i \geqslant k$ 条件的最小的 $i$。不存在的情况下输出 $n$。

**限制条件**

- $1 \leqslant n \leqslant 10^6$
- $0 \leqslant a_0 \leqslant a_1 \leqslant \cdots \leqslant a_{n-1} < 10^9$
- $0 \leqslant k \leqslant 10^9$

**样例**

**输入**

```
n = 5
a = {2, 3, 3, 5, 6}
k = 3
```

**输出**

```
1 (a0<3, a1>=3)
```

当然，如果朴素地按照顺序逐个查找的话，也可以求得答案。但是如果利用数列的有序性这一条件，则可以得到更高效的算法。首先我们来看一下第$n/2$个值。如果$a[n/2] \geqslant k$的话，就可以知道解不大于$n/2$。反之，如果$a[n/2]<k$，就可以知道解大于$n/2$。通过这样的一次比较，可以把解的范围缩小一半。反复与区间的中点进行比较，就可以不断把解的范围缩小到原来的一半，最终在$O(\log n)$次的比较之内求得最终的解。

解的范围的变化

```
// 输入
int n, k;
int a[MAX_N];

void solve() {
  // 初始化解的存在范围
  int lb = -1, ub = n;

  // 重复循环，直到解的存在范围不大于1
  while (ub - lb > 1) {
    int mid = (lb + ub) / 2;
    if (a[mid] >= k) {
      // 如果mid满足条件，则解的存在范围变为(lb, mid]
      ub = mid;
    } else {
      // 如果mid不满足条件，则解的存在范围变为(mid, ub]
      lb = mid;
    }
  }

  // 这时，lb + 1 = ub
  printf("%d\n", ub);
}
```

这种算法被称为二分搜索。此外，STL以lower_bound函数的形式实现了二分搜索。这个算法除了在上例中提到的在有序数列查找值之外，在求最优解的问题上也非常的有用。让我们考虑一下“求满足某个条件$C(x)$的最小的$x$”这一问题。对于任意满足$C(x)$的$x$，如果所有的$x' \geq x$也满足$C(x')$的话，我们就可以用二分搜索来求得最小的$x$。首先我们将区间的左端点初始化为不满足$C(x)$的值，右端点初始化为满足$C(x)$的值。然后每次取中点mid=$(lb+ub)/2$，判断$C$(mid)是否满足并缩小范围，直到$(lb, ub]$足够小了为止。最后$ub$就是要求的最小值。最大化的问题也可以用同样的方法求解。

### 3.1.2　假定一个解并判断是否可行

**Cable master （POJ No. 1064）**

有 $N$ 条绳子，它们的长度分别为 $L_i$。如果从它们中切割出 $K$ 条长度相同的绳子的话，这 $K$ 条绳子每条最长能有多长？答案保留到小数点后 2 位。

**限制条件**

- $1 \leqslant N \leqslant 10000$
- $1 \leqslant K \leqslant 10000$
- $1 \leqslant L_i \leqslant 100000$

**样例**

**输入**

```
N = 4
K = 11
L = {8.02, 7.43, 4.57, 5.39}
```

**输出**

```
2.00（每条绳子分别可以得到4条、3条、2条、2条，共计11条绳子）
```

这个问题用二分搜索可以非常容易地求得答案。让我们套用二分搜索的模型试着解决这个问题。令：

条件$C(x)$:=可以得到$K$条长度为$x$的绳子

则问题变成了求满足$C(x)$条件的最大的$x$。在区间初始化时，只需使用充分大的数INF(>$\mathrm{MAX_L}$)作为上界即可：

$lb$=0
$ub$=INF

现在的问题是是否可以高效地判断$C(x)$。由于长度为$L_i$的绳子最多可以切出floor($L_i$ / $x$)段长度为$x$绳子，因此

$C(x)$=(floor($L_i/x$)的总和是否大于或等于$K$)

它可以在$O(N)$的时间内被判断出来。

只能切出6段，所以解比3.00小

```
// 输入
int N, K;
double L[MAX_N];

// 判断是否满足条件
bool C(double x) {
  int num = 0;
  for (int i = 0; i < N; i++) {
    num += (int)(L[i] / x);
  }
  return num >= K;
}

void solve() {
  // 初始化解的范围
  double lb = 0, ub = INF;

  // 重复循环，直到解的范围足够小
  for (int i = 0; i < 100; i++) {
    double mid = (lb + ub) / 2;
    if (C(mid)) lb = mid;
    else ub = mid;
  }

  printf("%.2f\n", floor(ub * 100) / 100);
}
```

像这样，如果在求解最大化或最小化问题中，能够比较简单地判断条件是否满足，那么使用二分搜索法就可以很好地解决问题。

**专栏　二分搜索法的结束判定**

在输出小数的问题中，一般都会指定允许的误差范围或者是指定输出中小数点后面的位数。因此在使用二分搜索法时，有必要设置合理的结束条件来满足精度的要求。在上面的程序中，我们指定了循环次数作为终止条件。1 次循环可以把区间的范围缩小一半，100 次的循环则可以达到 $10^{-30}$ 的精度范围，基本上是没有问题的。除此之外，也可以把终止条件设为像 $(ub-lb)>\mathrm{EPS}$ 这样，指定一个区间的大小。在这种情况下，如果 EPS 取得太小了，就有可能会因为浮点小数精度的原因导致陷入死循环，请千万小心。

### 3.1.3 最大化最小值

**Aggressive cows（POJ No.2456）**

农夫约翰搭了一间有 $N$ 间牛舍的小屋。牛舍排在一条线上，第 $i$ 号牛舍在 $x_i$ 的位置。但是他的 $M$ 头牛对小屋很不满意，因此经常互相攻击。约翰为了防止牛之间互相伤害，因此决定把每头牛都放在离其他牛尽可能远的牛舍。也就是要最大化最近的两头牛之间的距离。

**限制条件**

- $2 \leqslant N \leqslant 100000$
- $2 \leqslant M \leqslant N$
- $0 \leqslant x_i \leqslant 10^9$

**样例**

**输入**

```
N = 5
M = 3
x = {1, 2, 8, 4, 9}
```

**输出**

```
3（在位置1，4，9的牛舍中放入三头牛）
```

类似的最大化最小值或者最小化最大值的问题，通常用二分搜索法就可以很好地解决。我们定义

$C(d)$:=可以安排牛的位置使得最近的两头牛的距离不小于$d$

那么问题就变成了求满足$C(d)$的最大的$d$。另外，最近的间距不小于$d$也可以说成是所有牛的间距都不小于$d$，因此就有

$C(d)$=可以安排牛的位置使得任意的牛的间距都不小于$d$

这个问题的判断使用贪心法便可非常容易地求解。

- 对牛舍的位置$x$进行排序
- 把第一头牛放入$x_0$的牛舍
- 如果第$i$头牛放入了$x_j$的话，第$i+1$头牛就要放入满足$x_j+d \leqslant x_k$的最小的$x_k$中

对$x$的排序只需在最开始时进行一次就可以了，每一次判断对每头牛最多进行一次处理，因此复杂度是$O(N)$。

```
// 输入
int N, M;
int x[MAX_N];

// 判断是否满足条件
bool C(int d) {
  int last = 0;
  for (int i = 1; i < M; i++) {
    int crt = last + 1;
    while (crt < N && x[crt] - x[last] < d) {
      crt++;
    }
    if (crt == N) return false;
    last = crt;
  }
  return true;
}

void solve() {
  // 最开始时对x数组排序
  sort(x, x + N);

  // 初始化解的存在范围
  int lb = 0, ub = INF;

  while (ub - lb > 1) {
    int mid = (lb + ub) / 2;
    if (C(mid)) lb = mid;
    else ub = mid;
  }

  printf("%d\n", lb);
}
```

### 3.1.4 最大化平均值

**最大化平均值**

有 $n$ 个物品的重量和价值分别是 $w_i$ 和 $v_i$。从中选出 $k$ 个物品使得单位重量的价值最大。

**限制条件**

- $1 \leqslant k \leqslant n \leqslant 10^4$
- $1 \leqslant w_i, v_i \leqslant 10^6$

**样例**

**输入**

```
n = 3
```

```
k = 2
(w, v) = {(2, 2), (5, 3), (2, 1)}
```

**输出**

```
0.75（如果选0号和2号物品，平均价值是(2+1)/(2+2)=0.75）
```

一般最先想到的方法可能是把物品按照单位价值进行排序，从大到小贪心地进行选取。但是这个方法对于样例输入得到的结果是5/7=0.714。所以这个方法是不可行的。那么应该如何求解呢？

实际上，对于这个问题使用二分搜索法可以很好地解决。我们定义

条件$C(x)$:=可以选择使得单位重量的价值不小于$x$

因此，原问题就变成了求满足$C(x)$的最大的$x$。那么怎么判断$C(x)$是否可行呢？假设我们选了某个物品的集合$S$，那么它们的单位重量的价值是

$$\sum_{i\in S} v_i / \sum_{i\in S} w_i$$

因此就变成了判断是否存在$S$满足下面的条件

$$\sum_{i\in S} v_i / \sum_{i\in S} w_i \geqslant x$$

把这个不等式进行变形就得到

$$\sum_{i\in S} (v_i - x\times w_i) \geqslant 0$$

因此，可以对$(v_i - x\times w_i)$的值进行排序贪心地进行选取。因此就变成了

$C(x)$=($(v_i - x\times w_i)$从大到小排列中的前$k$个的和不小于0)

每次判断的复杂度是$O(n\log n)$。

```
// 输入
int n, k;
int w[MAX_N], v[MAX_N];

double y[MAX_N]; // v - x * w

// 判断是否满足条件
```

```
bool C(double x) {
  for (int i = 0; i < n; i++) {
    y[i] = v[i] - x * w[i];
  }
  sort(y, y + n);

  // 计算y数组中从大到小前k个数的和
  double sum = 0;
  for (int i = 0; i < k; i++) {
    sum += y[n - i - 1];
  }

  return sum >= 0;
}

void solve() {
  double lb = 0, ub = INF;
  for (int i = 0; i < 100; i++) {
    double mid = (lb + ub) / 2;
    if (C(mid)) lb = mid;
    else ub = mid;
  }

  printf("%.2f\n", ub);
}
```

# 3.2 常用技巧精选（一）

在此我们介绍一些程序设计竞赛中的常用技巧。[①]

## 3.2.1 尺取法[②]

**Subsequence （POJ No.3061）**

给定长度为 $n$ 的数列整数 $a_0,a_1,\cdots,a_{n-1}$ 以及整数 $S$。求出总和不小于 $S$ 的连续子序列的长度的最小值。如果解不存在，则输出 0。

**限制条件**

- $10 < n < 10^5$
- $0 < a_i \leqslant 10^4$
- $S < 10^8$

**样例 1**

输入

```
n = 10
S = 15
a = {5, 1, 3, 5, 10, 7, 4, 9, 2, 8}
```

输出

```
2 (5+10)
```

**样例 2**

输入

```
n = 5
```

① 本节所涉及的多种技巧，其应用范围都比这里所介绍的模型要广得多，注意蕴含在技巧内的算法思想。许多看似非常复杂困难的问题，其问题关键都可以运用这里看似简单的技巧处理。——译者注

② 这里直接使用了日文原文的汉字，尺取法通常是指对数组保存一对下标（起点、终点），然后根据实际情况交替推进两个端点直到得出答案的方法，这种操作很像是尺蠖（日文中称为尺取虫）爬行的方式故得名。——译者注

```
S = 11
a = {1, 2, 3, 4, 5}
```

**输出**

```
3 (3+4+5)
```

由于所有的元素都大于零，如果子序列$[s, t)$满足$a_s+\cdots+a_{t-1}\geqslant S$，那么对于任何的$t<t'$一定有$a_s+\cdots+a_{t'-1}\geqslant S$。此外对于区间$[s,t)$上的总和来说如果令

$$\mathrm{sum}(i)=a_0+a_1+\cdots+a_{i-1}$$

那么

$$a_s+a_{s+1}+\cdots+a_{t-1}=\mathrm{sum}(t)-\mathrm{sum}(s)$$

因此预先以$O(n)$的时间计算好sum的话，就可以以$O(1)$的时间计算区间上的总和。这样一来，子序列的起点$s$确定以后，便可以用二分搜索快速地确定使序列和不小于$S$的结尾$t$的最小值。

```
// 输入
int n, S;
int a[MAX_N];

int sum[MAX_N + 1];

void solve() {
  // 计算sum
  for (int i = 0; i < n; i++) {
    sum[i + 1] = sum[i] + a[i];
  }

  if (sum[n] < S) {
    // 解不存在
    printf("0\n");
    return;
  }

  int res = n;
  for (int s = 0; sum[s] + S <= sum[n]; s++) {
    // 利用二分搜索求出t
    int t = lower_bound(sum + s, sum + n, sum[s] + S) - sum;
    res = min(res, t - s);
  }

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

这个算法的复杂度是$O(n\log n)$，虽然足以解决这个问题，但我们还可以更加高效地求解。我们设以$a_s$开始总和最初大于$S$时的连续子序列为$a_s+\cdots+a_{t-1}$，这时

$a_{s+1}+\cdots+a_{t-2}<a_s+\cdots+a_{t-2}<S$

所以从$a_{s+1}$开始总和最初超过$S$的连续子序列如果是$a_{s+1}+\cdots+a_{t'-1}$的话，则必然有$t\leqslant t'$。利用这一性质便可以设计出如下算法：

(1) 以 $s = t = \text{sum} = 0$ 初始化。
(2) 只要依然有 sum<$S$，就不断将 sum 增加 $a_t$，并将 $t$ 增加 1。
(3) 如果(2)中无法满足 sum⩾$S$ 则终止。否则的话，更新 res = min(res, $t-s$)。
(4) 将 sum 减去 $a_s$，$s$ 增加 1 然后回到(2)。

对于这个算法，因为$t$最多变化$n$次，因此只需$O(n)$的复杂度就可以求解这个问题了。

```
void solve() {
  int res = n + 1;
  int s = 0, t = 0, sum = 0;
  for (;;) {
    while (t < n && sum < S) {
      sum += a[t++];
    }
    if (sum < S) break;
    res = min(res, t - s);
    sum -= a[s++];
  }

  if (res > n) {
    // 解不存在
    res = 0;
  }
  printf("%d\n", res);
}
```

样例数据1对应的区间的变化

像这样反复地推进区间的开头和末尾，来求取满足条件的最小区间的方法被称为尺取法。

**Jessica's Reading Problem （POJ No.3320）**

为了准备考试，Jessica 开始读一本很厚的课本。要想通过考试，必须把课本中所有的知识点都掌握。这本书总共有 $P$ 页，第 $i$ 页恰好有一个知识点 $a_i$（每个知识点都有一个整数编号）。全书中同一个知识点可能会被多次提到，所以她希望通过阅读其中连续的一些页把所有的知识点都覆盖到。给定每页写到的知识点，请求出要阅读的最少页数。

**限制条件**

- $1 \leqslant P \leqslant 10^6$

**样例**

**输入**

```
P = 5
a = {1, 8, 8, 8, 1}
```

**输出**

```
2 (只要阅读第1页和第2页即可)
```

我们假设从某一页$s$开始阅读，为了覆盖所有的知识点需要阅读到$t$。这样的话可以知道如果从$s$+1开始阅读的话，那么必须阅读到$t' \geqslant t$页为止。由此这题也可以使用尺取法。

在某个区间$[s,t]$已经覆盖了所有的知识点的情况下，下一个区间$[s+1,t'](t' \geqslant t)$要如何求出呢？

所有的知识点都被覆盖⇔每个知识点出现的次数都不小于1

由以上的等价关系，我们可以用二叉树等数据结构来存储$[s,t]$区间上每个知识点的出现次数，这样把最开头的页$s$去除后便可以判断$[s+1,t]$是否满足条件。

从区间的最开头把$s$取出之后，页$s$上书写的知识点的出现次数就要减一，如果此时这个知识点的出现次数为0了，在同一个知识点再次出现前，不停将区间末尾$t$向后推进即可。每次在区间末尾追加页$t$时将页$t$上的知识点的出现次数加1，这样就完成了下一个区间上各个知识点出现次数的更新，通过重复这一操作可以以$O(P\log P)$的复杂度求出最小的区间。

```
// 输入
int P;
int a[MAX_P];

void solve() {
  // 计算全部知识点的总数
```

```
    set<int> all;
    for (int i = 0; i < P; i++) {
      all.insert(a[i]);
    }
    int n = all.size();

    // 利用尺取法来求解
    int s = 0, t = 0, num = 0;
    map<int, int> count;  // 知识点→出现次数的映射
    int res = P;
    for (;;) {
      while (t < P && num < n) {
        if (count[a[t++]]++ == 0) {
          // 出现新的知识点
          num++;
        }
      }
      if (num < n) break;
      res = min(res, t - s);
      if (--count[a[s++]] == 0) {
        // 某个知识点的出现次数为0了
        num--;
      }
    }

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

## 3.2.2　反转（开关问题）

**Face The Right Way （POJ No. 3276）**

$N$ 头牛排成了一列。每头牛或者向前或者向后。为了让所有的牛都面向前方，农夫约翰买了一台自动转向的机器。这个机器在购买时就必须设定一个数值 $K$，机器每操作一次恰好使 $K$ 头连续的牛转向。请求出为了让所有的牛都能面向前方需要的最少的操作次数 $M$ 和对应的最小的 $K$。

**限制条件**

- $1 \leqslant N \leqslant 5000$

**样例**

**输入**

```
N = 7
BBFBFBB (F: 面向前方, B: 面向后方)
```

**输出**

```
K = 3
M = 3
（先反转1~3号的三头牛，然后再反转3~5号，最后反转5~7号）
```

K=3时的解法

首先我们来看看对于一个特定的$K$如何求出让所有的牛面朝前方的最小操作次数。如果把牛的方向作为状态进行搜索的话，由于状态数有$2^N$个，是无法在时限内找出答案的。那么不搜索的话要怎么办呢？

首先，交换区间反转的顺序对结果是没有影响的。此外，可以知道对同一个区间进行两次以上的反转是多余的。由此，问题就转化成了求需要被反转的区间的集合。于是我们先考虑一下最左端的牛。包含这头牛的区间只有一个，因此如果这头牛面朝前方，我们就能知道这个区间不需要反转。

反之，如果这头牛面朝后方，对应的区间就必须进行反转了。而且在此之后这个最左的区间就再也不需要考虑了。这样一来，通过首先考虑最左端的牛，问题的规模就缩小了1。不断地重复下去，就可以无需搜索求出最少所需的反转次数了。

此外，通过上面的分析可以知道，忽略掉对同一个区间重复反转这类多余操作之后，只要存在让所有的牛都朝前的方法，那么操作就和顺序无关可以唯一确定了。

这个算法的复杂度又如何呢？首先我们需要对所有的$K$都求解一次，对于每个$K$我们都要从最左端开始来考虑$N$头牛的情况。此时最坏情况下需要进行$N-K+1$次的反转操作，而每次操作又要反转$K$头牛，于是总的复杂度就是$O(N^3)$。这样的话还不足以在时限内解决问题。但是区间反转的部分还是很容易进行优化的。

$f[i]$ := 区间$[i, i+K-1]$进行了反转的话则为1，否则为0

这样，在考虑第$i$头牛时，如果$\sum_{j=i-K+1}^{i-1} f[i]$为奇数的话，则这头牛的方向与起始方向是相反的，否则方向不变。由于

$$\sum_{i=(i+1)-K+1}^{i} f[j] = \sum_{i=i-K+1}^{i-1} f[j] + f[i] - f[i-K+1]$$

所以这个和每一次都可以用常数时间计算出来，复杂度就降为了$O(N^2)$，能在时限内解决了。

```
// 输入
int N;
int dir[MAX_N];  // 牛的方向(0:F, 1:B)

int f[MAX_N];    // 区间[i,i+K-1]是否进行反转

// 固定K，求对应的最小操作回数
// 无解的话则返回-1
int calc(int K) {
  memset(f, 0, sizeof(f));
  int res = 0;
  int sum = 0;   // f的和
  for (int i = 0; i + K <= N; i++) {
    // 计算区间[i,i+K-1]
    if ((dir[i] + sum) % 2 != 0) {
      // 前端的牛面朝后方
      res++;
      f[i] = 1;
    }
    sum += f[i];
    if (i - K + 1 >= 0) {
      sum -= f[i - K + 1];
    }
  }

  // 检查剩下的牛是否有面朝后方的情况
  for (int i = N - K + 1; i < N; i++) {
    if ((dir[i] + sum) % 2 != 0) {
      // 无解
      return -1;
    }
    if (i - K + 1 >= 0) {
      sum -= f[i - K + 1];
    }
  }

  return res;
}

void solve() {
  int K = 1, M = N;
  for (int k = 1; k <= N; k++) {
    int m = calc(k);
```

```
    if (m >= 0 && M > m) {
      M = m;
      K = k;
    }
  }
  printf("%d %d\n", K, M);
}
```

## Fliptile （POJ No.3279）[1]

农夫约翰知道聪明的牛产奶多。于是为了提高牛的智商他准备了如下游戏。有一个 $M \times N$ 的格子，每个格子可以翻转正反面，它们一面是黑色，另一面是白色。黑色的格子翻转后就是白色，白色的格子翻转过来则是黑色。游戏要做的就是把所有的格子都翻转成白色。不过因为牛蹄很大，所以每次翻转一个格子时，与它上下左右相邻接的格子也会被翻转。因为翻格子太麻烦了，所以牛都想通过尽可能少的次数把所有格子都翻成白色。现在给定了每个格子的颜色，请求出用最小步数完成时每个格子翻转的次数。最小步数的解有多个时，输出字典序最小的一组。解不存在的话，则输出 IMPOSSIBLE。

**限制条件**

- $1 \leqslant M, N \leqslant 15$

**样例**

**输入**

```
M = 4
N = 4
每个格子的颜色如下(0表示白色，1表示黑色)

1 0 0 1
0 1 1 0
0 1 1 0
1 0 0 1
```

**输出**

```
0 0 0 0
1 0 0 1
1 0 0 1
0 0 0 0
```

① 这道题的模型常被称为开关问题或关灯问题，较早见于Extended Lights Out（Greater New York 2002）这道题。高斯消元法也可以用于求解一组可行解，并且通过这些分析可以知道，自由变元的个数不会超过$N$个，所以也可以用于求解最优解。——译者注

解法示意

首先，同一个格子翻转两次的话就会恢复原状，所以多次翻转是多余的。此外，翻转的格子的集合相同的话，其次序是无关紧要的。因此，总共有$2^{NM}$种翻转的方法。不过这个解空间太大了，我们需要想出更有效的办法。

让我们再回顾一下前面的问题。在那道题中，让最左端的牛反转的方法只有1种，于是用直接判断的方法确定就可以了。同样的方法在这里还行得通吗？

不妨先看看最左上角的格子。在这里，除了翻转(1,1)之外，翻转(1,2)和(2,1)也可以把这个格子翻转，所以像之前那样直接确定的办法行不通。

于是不妨先指定好最上面一行的翻转方法。此时能够翻转(1,1)的只剩下(2,1)了，所以可以直接判断(2,1)是否需要翻转。类似地(2,1)~(2,$N$)都能这样判断，如此反复下去就可以确定所有格子的翻转方法。最后($M$,1)~($M$,$N$)如果并非全为白色，就意味着不存在可行的操作方法。

像这样，先确定第一行的翻转方式，然后可以很容易判断这样是否存在解以及解的最小步数是多少，这样将第一行的所有翻转方式都尝试一次就能求出整个问题的最小步数。这个算法中最上面一行的翻转方式共有$2^N$种，复杂度为$O(MN2^N)$。

```
// 邻接的格子的坐标
const int dx[5] = {-1, 0, 0, 0, 1};
const int dy[5] = {0, -1, 0, 1, 0};

// 输入
int M, N;
int tile[MAX_M][MAX_N];

int opt[MAX_M][MAX_N];    // 保存最优解
int flip[MAX_M][MAX_N];   // 保存中间结果

// 查询(x,y)的颜色
int get(int x, int y) {
  int c = tile[x][y];
  for (int d = 0; d < 5; d++) {
    int x2 = x + dx[d], y2 = y + dy[d];
    if (0 <= x2 && x2 < M && 0 <= y2 && y2 < N) {
      c += flip[x2][y2];
    }
  }
  return c % 2;
```

```
}

// 求出第1行确定情况下的最小操作次数
// 不存在解的话返回-1
int calc() {
  // 求出从第2行开始的翻转方法
  for (int i = 1; i < M; i++) {
    for (int j = 0; j < N; j++) {
      if (get(i - 1, j) != 0) {
        // (i-1,j)是黑色的话，则必须翻转这个格子
        flip[i][j] = 1;
      }
    }
  }

  // 判断最后一行是否全白
  for (int j = 0; j < N; j++) {
    if (get(M - 1, j) != 0) {
      // 无解
      return -1;
    }
  }

  // 统计翻转的次数
  int res = 0;
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      res += flip[i][j];
    }
  }
  return res;
}

void solve() {
  int res = -1;

  // 按照字典序尝试第一行的所有可能性
  for (int i = 0; i < 1 << N; i++) {
    memset(flip, 0, sizeof(flip));
    for (int j = 0; j < N; j++) {
      flip[0][N - j - 1] = i >> j & 1;
    }
    int num = calc();
    if (num >= 0 && (res < 0 || res > num)) {
      res = num;
      memcpy(opt, flip, sizeof(flip));
    }
  }

  if (res < 0) {
    // 无解
    printf("IMPOSSIBLE\n");
  } else {
    for (int i = 0; i < M; i++) {
```

```
      for (int j = 0; j < N; j++) {
        printf("%d%c", opt[i][j], j + 1 == N ? '\n' : ' ');
      }
    }
  }
}
```

**专栏　集合的整数表示**

前面的代码里，为了尝试第一行的所有可能性，使用了集合的整数表现。在程序中表示集合的方法有很多种，当元素数比较少时，像这样用二进制码来表示比较方便。集合$\{0,1,\cdots,n-1\}$的子集 $S$ 可以用如下方式编码成整数。

$$f(S)=\sum_{i\in S}2^i$$

像这样表示之后，一些集合运算可以对应地写成如下方式。

- 空集$\phi$：……0
- 只含有第$i$个元素的集合$\{i\}$：……1<<i
- 含有全部$n$个元素的集合$\{0,1,\cdots,n-1\}$：……(1<<n)-1
- 判断第$i$个元素是否属于集合$S$：……if (S>>i&1)
- 向集合中加入第$i$个元素$S\cup\{i\}$：……S|1<<i
- 从集合中去除第$i$个元素$S\backslash\{i\}$：……S&~(1<<i)
- 集合$S$和$T$的并集$S\cup T$：……S|T
- 集合$S$和$T$的交集$S\cap T$：……S&T

此外，想要将集合$\{0,1,\cdots,n-1\}$的所有子集枚举出来的话，可以像下面这样书写

```
for (int S = 0; S < 1 << n; S++) {
  // 对子集的处理
}
```

按照这个顺序进行循环的话，$S$ 便会从空集开始，然后按照$\{0\}$、$\{1\}$、$\{0,1\}$、$\cdots$、$\{0,1,\cdots,n-1\}$的升序顺序枚举出来。

接下来介绍一下如何枚举某个集合 sup 的子集。这里 sup 是一个二进制码，其本身也是某个集合的子集。例如给定了 01101101 这样的集合，要将 01100000 或者 00101101 等子集枚举出来。前面是从 0 开始不断加 1 来枚举出了全部的子集。此时，sub+1 并不一定是 sup 的子集。而(sub+1)&sup 虽然是 sup 的子集，可是很有可能依旧是 sub，没有任何改变。

所以我们要反过来，从 sup 开始每次减 1 直到 0 为止。由于 sub-1 并不一定是 sup 的子集，所以我们把它与 sup 进行按位与&。这样的话就可以将 sup 所有的子集按照降序列举出来。(sub-1)&sup 会忽略 sup 中的 0 而从 sub 中减去 1。

```
int sub = sup;
do {
  // 对子集的处理
  sub = (sub - 1) & sup;
} while(sub != sup); // 处理完0之后，会有-1&sup=sup
```

最后我们介绍一下枚举$\{0,1,\cdots,n-1\}$所包含的所有大小为 $k$ 的子集的方法。通过使用位运算我们可以像如下代码所示简单地按照字典序升序地枚举出所有满足条件的二进制码。

```
int comb = (1 << k) - 1;
while (comb < 1 << n) {
  // 这里进行针对组合的处理
  int x = comb & -comb, y = comb + x;
  comb = ((comb & ~y) / x >> 1) | y;
}
```

按照字典序的话，最小的子集是(1<<k)-1，所以用它作为初始值。现在我们求出 comb 其后的二进制码。例如 0101110 之后的是 0110011，0111110 之后的是 1001111。下面是求出 comb 下一个二进制码的方法。

(1) 求出最低位的 1 开始的连续的 1 的区间（0101110→0001110）
(2) 将这一区间全部变为 0，并将区间左侧的那个 0 变为 1（0101110→0110000）
(3) 将第(1)步里取出的区间右移，直到剩下的 1 的个数减少了 1 个（0001110→0000011）
(4) 将第(2)步和第(3)步的结果按位取或（0110000|0000011=0110011）

对于非零的整数，x&(-x)的值就是将其最低位的 1 独立出来后的值。这是由于计算机中负数采用二进制补码表示，-x 实际上对应于(~x)+1（将 x 按位取反然后加上 1）。

| $x$ | $x$的二进制 | $-x$的二进制 | $x$ &-$x$ |
|---|---|---|---|
| 1 | 0001 | 1111 | 0001 |
| 2 | 0010 | 1110 | 0010 |
| 3 | 0011 | 1101 | 0001 |
| 4 | 0100 | 1100 | 0100 |
| 5 | 0101 | 1011 | 0001 |
| 6 | 0110 | 1010 | 0010 |
| 7 | 0111 | 1001 | 0001 |

将最低位的 1 取出后，设它为 x。那么通过计算 y=comb+x，就将 comb 从最低位的 1 开始的连续的 1 都置 0 了。我们来比较一下~y 和 comb。在 comb 中加上 x 后没有变化的位，在~y 中全都取相反的值。而最低位 1 开始的连续区间在~y 中依然是 1，区间左侧的那个 0 在~y 中也依然是 0。于是通过计算 z=comb&~y 就得到了最低位 1 开始的连续区间。比如如果 comb=0101110，则 $x$=0000010，$y$=0110000，$z$=0001110。

同时，y 也恰好是第(2)步要求的值。那么首先将 z 不断右移，直到最低位为 1，这通过计算 z/x 即可完成。这样再将 z/x 右移 1 位就得到了第(3)步要求的值。这样我们就求得了 comb

之后的下一个二进制列。因为是从 $n$ 个元素的集合中进行选择，所以 comb 的值不能大于等于 1<<n。如此一来，就完成了大小为 $k$ 的所有子集的枚举。

除上述例子之外，还可以利用位运算完成满足其他条件的集合的枚举，例如不包含相邻元素的集合等。

### 3.2.3　弹性碰撞

**Physics Experiment （POJ No.3684）**

用 $N$ 个半径为 $R$ 厘米的球进行如下实验。

在 $H$ 米高的位置设置了一个圆筒，将球垂直放入（从下向上数第 $i$ 个球的底端距离地面高度为 $H+2R$）。实验开始时最下面的球开始掉落，此后每一秒又有一个球开始掉落。不计空气阻力，并假设球与球或地面间的碰撞是弹性碰撞。

请求出实验开始后 $T$ 秒种时每个球底端的高度。假设重力加速度为 $g=10\text{m/s}^2$。

**限制条件**

- $1 \leqslant N \leqslant 100$
- $1 \leqslant H \leqslant 10000$
- $1 \leqslant R \leqslant 100$
- $1 \leqslant T \leqslant 10000$

**样例 1**

输入

```
N = 1
H = 10
R = 10
T = 100
```

输出

```
4.95
```

**样例 2**

输入

```
N = 2
```

```
H = 10
R = 10
T = 100
```

**输出**

```
4.95 10.20
```

初始状态

首先考虑一下只有一个球的情形。这时只是单纯的物理问题。从高为$H$的位置下落的话需要花费的时间是

$$t=\sqrt{\frac{2H}{g}}$$

这样的话，在时刻$T$时，令$k$为满足$kt\leqslant T$的最大整数，那么

$$y=\begin{cases} H-\frac{1}{2}g(T-kt)^2 & (\text{是 } k \text{ 偶数时}) \\ H-\frac{1}{2}g(kt+t-T)^2 & (\text{是 } k \text{ 奇数时}) \end{cases}$$

接下来再考虑多个球的情形。乍一看，因为多个球之间会有碰撞，必须对物理运动进行模拟，事实上并没有这个必要。我们来回忆一下此前热身题目“Ants”。在那道题目中两只蚂蚁相遇后并不是各自折返，而是当作擦身而过继续走下去，于是就将问题简化了。

这里的问题可以用同样方法思考。首先先来考虑一下$R$=0的情况。如果认为所有的球都是一样的，就可以无视它们的碰撞，视为直接互相穿过继续运动。由于在有碰撞时球的顺序不会发生改变，所以忽略碰撞，将计算得到的坐标进行排序后，就能知道每个球的最终位置。那么，$R$>0时要怎