CADERNO DE PROBLEMAS

Meia-Maratona de Programação

Escola de Inverno, Chapecó, 3–4 de agosto de 2013

Organizadores:
Marcos Moretto
Leandro Zatesko
Marcelo Cezar Pinto

Apoio:
ACCEPT@UNO
Clube de Programação da UFFS
Coordenação do Curso de Ciência da Computação da UNOCHAPECÓ
Coordenação do Curso de Sistemas de Informação da UNOCHAPECÓ

Patrocínio:
Sociedade Brasileira de Computação
Fundação Carlos Chagas

A vida não é um problema para ser resolvido,
mas uma realidade para ser experienciada.

Søren A. Kierkegaard

Instruções

- Este caderno contém 6 problemas. As páginas estão numeradas de 1 a 21. Verifique se o caderno está completo.
- A competição inicia às 14:30 e termina às 17:30.
- A entrada de seu programa deve ser lida da *entrada padrão*.
- A entrada é composta por vários casos de teste, cada um descrito em um número de linhas que depende do problema.
- Quando uma linha da entrada contém vários valores, estes são separados por um único espaço em branco; a entrada não contém nenhum outro espaço em branco.
- Cada linha, incluindo a última, contém o caractere final-de-linha.
- O final da entrada coincide com o final do arquivo.
- A saída de seu programa deve ser escrita na *saída padrão*.
- Quando uma linha da saída contém vários valores, estes devem ser separados por um único espaço em branco; a saída não deve conter nenhum outro espaço em branco.
- Cada linha, incluindo a última, deve conter o caractere final-de-linha.

Lista de Problemas

Problema A *O Fim Está Próximo!* 2
★ Resolução do Professor 3

Problema B *Pizza Integral* 6
★ Resolução do Professor 7

Problema C *O Jogo do Dicionário* 9
★ Resolução do Professor 10

Problema D *Os Teoremas de Fermat* 13
★ Resolução do Professor 14

Problema E *Palíndromos* 15
★ Resolução do Professor 16

Problema F *Resta-Um Triangular* 17
★ Resolução do Professor 18

Problema A

O Fim Está Próximo!

Arquivo: `fim.[c|cpp|java]`

Problema escrito por Marcelo Cezar Pinto, UFFS.

O conhecido mago Zak Galou fez um curso de final de semana sobre futurologia em uma instituição obscura. A partir dos conhecimentos adquiridos lá, Zak Galou inovou e passou a ministrar o seminário *Como Aproveitar a Vida antes do Fim!* ao redor do mundo. Casualmente, Zak Galou estará em Chapecó e região nesta semana. Como quem já o conhece sabe, Zak Galou é bom em matar monstros e nos trabalhos em vinhedos, mas é péssimo para resolver problemas.

Ele aproveitou sua passagem pela cidade e solicitou um programa que, dados um prazo limite para o final e uma série de atividades a realizar, cada qual com um tempo necessário de execução e um valor positivo de felicidade ao ser cumprida, calcula qual seria o valor máximo de felicidade que se poderia atingir.

Pelas suas qualidades como futurólogo, Zak Galou prevê o tempo limite para o final. Além disso, ele estabelece o tempo de execução e a felicidade gerada para cada atividade (plantar uma árvore, escrever um livro, tomar um sorvete de melancia etc.). Você se prontificou a ajudá-lo nesta tarefa de indicar o máximo de felicidade possível de se atingir ao cumprir algumas (eventualmente nenhuma ou todas) atividades sem estourar o tempo limite para o final.

Entrada

A entrada é composta por vários casos de teste. Para cada um deles, a primeira linha contém os inteiros L e N, separados por um espaço em branco, que indicam o tempo limite para o fim e o número de atividades indicadas por Zak Galou, respectivamente ($1 \leqslant L \leqslant 100$ e $1 \leqslant N \leqslant 100$). A próxima linha contém N inteiros T_i separados por um espaço em branco, que indicam o tempo de execução de cada tarefa ($1 \leqslant T_i \leqslant 100$, $1 \leqslant i \leqslant N$). A linha seguinte contém N inteiros V_i separados por um espaço em branco, que indicam a felicidade gerada ao concluir cada tarefa ($1 \leqslant V_i \leqslant 50$, $1 \leqslant i \leqslant N$). O final da entrada é indicado por $L = N = 0$.

Saída

Para cada caso de teste é escrita uma linha no formato: "`Caso` X `gera felicidade` Y", onde X indica o número do caso de teste (o primeiro lido na entrada será 1, o segundo será 2, e assim por diante) e Y indica a soma das felicidades V_i de modo que os tempos T_i não ultrapassem L.

Exemplo de entrada	**Saída correspondente**
1 2	Caso 1 gera felicidade 12
1 1	Caso 2 gera felicidade 0
10 12	Caso 3 gera felicidade 50
10 5	
12 20 50 13 100	
1 22 25 2 50	
20 4	
5 6 16 4	
23 24 46 3	
0 0	

★ Resolução do Professor

Este é *exatamente* o clássico *Problema da Mochila*: *Dados n objetos $x_1, \ldots, x_n$, tendo cada objeto x_i, $1 \leqslant i \leqslant n$, um peso w_i e um valor v_i, qual é o valor máximo que um ladrão consegue carregar numa mochila de capacidade W sem exceder a capacidade da mochila?* As principais variantes deste problema são:

Variante clássica Para cada item x_i, ou o ladrão leva x_i, ou não leva x_i.

Variante ilimitada Permite que o ladrão leve mais de uma cópia do item x_i.

Variante fracionada Permite que o ladrão leve apenas uma parte de um item x_i.

Variante como problema de decisão É parâmetro do problema também um *bound* V, e se pergunta se é possível que o ladrão leve no mínimo o valor V na mochila sem exceder a capacidade W.

A variante como problema de decisão é um problema $\mathcal{NP}$-completo. Por uma busca binária, uma solução polinomial para essa variante implicaria numa solução polinomial para todas as outras. Porém, a variante fracionada admite solução linear por um algoritmo guloso. As demais variantes podem ser atacadas por Programação Dinâmica e admitem soluções pseudopolinomiais.

... e implicaria $\mathcal{P} = \mathcal{NP}$.

O Problema A, *O Fim Está Próximo!*, se trata de um caso da variante clássica. Os itens são as atividades, o peso de cada item é o tempo que a atividade consome, e o valor de cada item é a felicidade que a atividade agrega. Estabelecendo essa interpretação, vamos deixar de lado as nomenclaturas fornecidas pelo enunciado e vamos tratar do *Problema da Mochila Clássico*. O *estado* da Programação Dinâmica, para um $i \in [0..W]$ e um $j \in [0..n]$, será S_{ij}, que representa o maior valor que podemos carregar numa mochila com capacidade i tendo disponíveis apenas os itens $x_1, \ldots, x_j$. Assim, podemos identificar a *subestrutura ótima* do problema estabelecendo a seguinte recorrência:

O que queremos, no fim das contas, é o valor de S_{Wn}.

$$S_{ij} = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0; \\ S_{i,j-1} & \text{se } i = 0 \text{ e } j = 0 \text{ mas } w_j > i; \\ S_{i-w_j,j-1} & \text{se } i = 0,\ j = 0 \text{ e } w_j \leqslant i. \end{cases}$$

Implementando a recorrência num código iterativo que reaproveita os valores S_{ij} calculados, armazenando-os numa matriz $\mathsf{S}[i][j]$, temos o seguinte algoritmo para o *Problema da Mochila*:

```
Mochila(v_1,...,v_n, w_1,...,w_n, W):
1  para j de 0 até n, faça:
2     S[0][j] ← 0;
3  para i de 1 até W, faça:
4     S[i][0] ← 0;
5     para j de 1 até n, faça:
6        se w_j > i ou S[i][j−1] ⩾ S[i−w_j][j−1] + v_j, então,
7           S[i][j] ← S[i][j−1];
8        senão, S[i][j] ← S[i−w_j][j−1] + v_j;
9  devolva S[W][n].
```

Algoritmo 1

O algoritmo apresentado tem complexidade de tempo $O(nW)$. Parece polinomial, mas não é. Embora o fator n seja linear no tamanho da entrada, W é exponencial, pois W pode ser arbitrariamente grande, e a representação de W na entrada requer só $O(\log W)$ *bits*. No entanto, se o parâmetro numérico W fosse limitado por uma constante L, a complexidade de tempo do algoritmo seria $O(nL) = O(n)$, ou seja, linear. Algoritmos exponenciais que se tornam polinomiais quando seus parâmetros numéricos são limitados por constantes são chamados de *algoritmos pseudopolinomiais*. Problemas $\mathcal{NP}$-completos que admitem algoritmos pseudopolinomiais são chamados de *fracamente* $\mathcal{NP}$-completos.

Como, em nosso caso, ambos W e n são no máximo 100, do jeito que está já temos um código que é aceito, sem necessidade de otimização:

```
#include<stdio.h>

int main(void) {
  int n, W, v[101], w[101], S[101][101], i, j, caso=1;
  scanf("%d %d", &W, &n);
  while (n) {
    for (i = 1; i <= n; i++) scanf("%d", &w[i]);
    for (i = 1; i <= n; i++) scanf("%d", &v[i]);
    for (j = 0; j <= n; j++) S[0][j] = 0;
    for (i = 1; i <= W; i++) {
      S[i][0] = 0;
      for (j = 1; j <= n; j++) {
        if (w[j] > i || S[i][j-1] >= S[i-w[j]][j-1] + v[j])
          S[i][j] = S[i][j-1];
        else S[i][j] = S[i-w[j]][j-1] + v[j];
      }
    }
    printf("Caso %d gera felicidade %d\n", caso++, S[W][n]);
    scanf("%d %d", &W, &n);
  }
  return 0;
}
```

O que podemos dizer a respeito da complexidade de espaço do Algoritmo 2? Evidentemente, a matriz S possui $n \times W$ posições, e temos que a complexidade de espaço também é exponencial: $O(nW)$. Ora, $\mathcal{NP} \subseteq \mathcal{PSPACE}$, e certamente deve haver um algoritmo de espaço polinomial para o *Problema da Mochila*. Tanto do ponto de vista teórico quanto do prático, as restrições de espaço costumam ser bem mais rígidas que as de tempo. Em nosso caso, $n \times W \leqslant 10\,000$, e não tivemos problemas. No entanto, se $n \times W \cong 10^8$, por exemplo, teríamos um código que seria aceito por tempo, mas que não o seria por espaço. A verdade é que não precisamos guardar todas as as $n \times W$ posições da matriz. Se, ao invés de preenchermos a matriz pelas linhas, preenchêssemo-la pelas colunas, notaríamos que a coluna j depende exclusivamente da coluna $j-1$. Assim, podemos fazer todo o serviço num só vetor, desde que tenhamos o seguinte cuidado: a posição $S[i][j]$ depende de $S[i'][j-1]$ com $i' < i$, então, não podemos iterar i crescentemente, mas decrescentemente, senão, não estaríamos olhando para $S[i'][j-1]$, mas para $S[i'][j]$.

Trabalhe com restrições de tempo da ordem de 10^8 e com restrições de espaço da ordem de 10^4.

Eis o algoritmo e o código:

MOCHILA$(v_1, \ldots, v_n, w_1, \ldots, w_n, W)$:
1 **para** i **de** 0 **até** W, **faça**:
2 $S[i] \leftarrow 0$;
3 **para** j **de** 1 **até** n, **faça**:
4 **para** i **de** W **até** 1, **faça**:
5 **se** $w_j \leqslant i$ **e** $S[i] \leqslant S[i - w_j] + v_j$, **então**,
6 $S[i] \leftarrow S[i - w_j] + v_j$;
7 **devolva** $S[W]$.

Algoritmo 2

```
#include<stdio.h>

int main(void) {
  int n, W, v[101], w[101], S[101], i, j, caso=1;
```

```
  scanf("%d %d", &W, &n);
  while (n) {
    for (i = 1; i <= n; i++) scanf("%d", &w[i]);
    for (i = 1; i <= n; i++) scanf("%d", &v[i]);
    for (i = 0; i <= W; i++) S[i] = 0;
    for (j = 1; j <= n; j++)
      for (i = W; i >= 1; i--)
        if (w[j] <= i && S[i] < S[i-w[j]] + v[j])
          S[i] = S[i-w[j]] + v[j];
    printf("Caso %d gera felicidade %d\n", caso++, S[W]);
    scanf("%d %d", &W, &n);
  }
  return 0;
}
```

Problema B

Pizza Integral

Arquivo: `pizza.[c|cpp|java]`

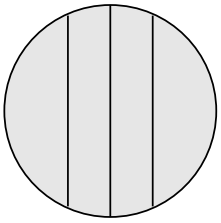
Problema escrito por Leandro Zatesko, UFFS.

Aberta apenas há 3 anos, a *Pizzaria Integral* tem mudado a história de Chapecó, atraindo olhares de renomados gastrônomos do mundo. Quando abriu, a proposta do estabelecimento era servir pizzas inovadoras, feitas com farinha de trigo integral, considerada mais saborosa e mais saudável. Passados alguns meses, o pizzaiolo começou a inovar também no menu, desenvolvendo pizzas cada vez mais exóticas, como *Lembranças duma Infância Mineira* (leva doce-de-leite, queijo minas e amoras flambadas em cachaça), *Romeu, Julieta e Um Terceiro* (leva goiabada, queijo mascarpone e camarões grandes) e *Carne-de-Onça* (leva carne moída de 1ª crua, chalota, alho *chips*, farofa de bala de hortelã, molho de limão e azeite de oliva).

Para fazer farofa de bala de hortelã, basta colocar quantas balas de hortelã quiser num triturador de alimentos.

Mas não é só nas receitas que a desbravadora pizzaria está inovando. Desde o ano passado, foi implantada também uma revolução no modo de se venderem as pizzas. Em pizzarias comuns, geralmente o freguês se limita a escolher o tamanho de sua pizza duma lista de tamanhos pré-definidos, como *pequeno*, *médio*, *grande* e *gigante*. Na *Pizzaria Integral*, contudo, o cliente pode escolher o diâmetro exato de sua pizza, e ainda pode escolher em quantos pedaços quer que ela venha cortada. O pizzaiolo garante não apenas a forma circular perfeita da pizza, atendendo o diâmetro especificado, mas também assegura que todos os pedaços terão exatamente a mesma área. Inovando ainda mais, os cortes são feitos todos paralelos. A Figura ilustra uma pizza cortada em 4 pedaços.

Evidentemente, o preço não é proporcional ao diâmetro, mas ao quadrado dele.



Entrada

A entrada é composta por vários casos de teste, cada qual numa linha. Cada caso de teste consiste de dois inteiros d ($20 \leqslant d \leqslant 100$) e k ($1 \leqslant k \leqslant 10$), separados por um espaço, representando respectivamente o diâmetro, em centímetros, solicitado para a pizza e o número de pedaços em que se deve cortá-la. A entrada é finalizada por $d = k = 0$.

Saída

Note que, quando $k = 1$, apenas a quebra-de-linha é impressa.

Para cada caso de teste, seu programa deverá imprimir em sequência $k - 1$ valores numéricos $v_1, v_2, \ldots, v_{k-1}$ ($0.00 < v_1 < v_2 < \cdots < v_{k-1} < d$), indicando onde devem ser feitos os cortes na pizza, e uma quebra-de-linha. Os valores devem ser separados por um espaço e fornecidos sob precisão de duas casas decimais, sendo ponto, e não vírgula, o separador das casas.

Exemplo de entrada	**Saída correspondente**
20 2	10.00
50 1	
100 5	25.41 42.11 57.89 74.59
80 4	23.84 40.00 56.16
0 0	

★ Resolução do Professor

Como podemos calcular a área $A(x)$ definida por um corte x na pizza, $0 \leqslant x \leqslant r = d/2$? Girando a pizza para alinharmos o corte com o eixo das abscissas, temos, pela Figura 1, que

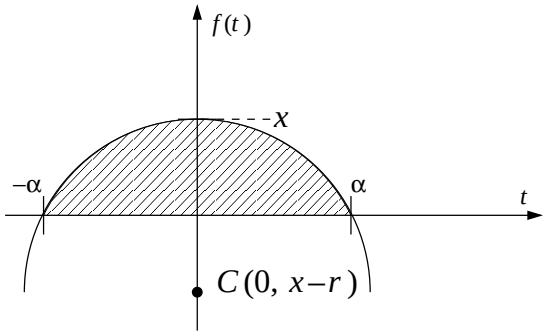


Figura 1

$$A(x) = \int_{-\alpha}^{\alpha} f(t)\mathrm{d}t.$$

Como, da equação da circunferência, $r^2 = t^2 + (f(t) - (x-r))^2$, temos que:

$$f(t) = \sqrt{r^2 - t^2} - (r - x);$$
$$\alpha = \sqrt{2rx - x^2};$$
$$-\alpha = -\sqrt{2rx - x^2}.$$

Assim,

$$A(x) = \int_{-\sqrt{2rx-x^2}}^{\sqrt{2rx-x^2}} \sqrt{r^2 - t^2}\mathrm{d}t - \int_{-\sqrt{2rx-x^2}}^{\sqrt{2rx-x^2}} (r - x)\mathrm{d}t. \tag{1}$$

Ora,

$$\int_{-\sqrt{2rx-x^2}}^{\sqrt{2rx-x^2}} (r - x)\mathrm{d}t = 2(r - x)\sqrt{2rx - x^2}, \tag{2}$$

e como, de qualquer tábua de integrais,

$$\int \sqrt{r^2 - t^2}\mathrm{d}t = \frac{t}{2}\sqrt{r^2 - t^2} + \frac{r^2}{2}\operatorname{arcsen}\frac{t}{r} + c,$$

temos que

$$\int_{-\sqrt{2rx-x^2}}^{\sqrt{2rx-x^2}} \sqrt{r^2 - t^2}\mathrm{d}t = (r - x)\sqrt{2rx - x^2} + r^2 \operatorname{arcsen}\frac{\sqrt{2rx - x^2}}{r}. \tag{3}$$

Substituindo as Equações 2 e 3 na Equação 1, temos que

$$A(x) = r^2 \operatorname{arcsen}\frac{\sqrt{2rx - x^2}}{r} - (r - x)\sqrt{2rx - x^2}. \tag{4}$$

. . . embora todo mundo que passou em Cálculo I saiba fazer esta integral, por substituição trigonométrica, sem precisar consultar uma tábua! No entanto, um bom material para se levar para a Maratona é o Theoretical Computer Science Cheat Sheet, que tem de tudo, inclusive tábua de integrais.

Dá para simplificar $A(x)$ um pouco mais, mas para quê?

Tudo o que queremos encontrar são os valores $v_1, v_2, \ldots, v_{\lfloor (k-1)/2 \rfloor}$ tais que $A(v_j) = j\pi r^2/k$, $\forall j \in [1..\lfloor (k-1)/2 \rfloor]$, já que os demais valores são obtidos espelhando-se esses em relação ao meio da pizza. O problema é que a Equação 4 é uma *equação transcendental*, o que significa que é comprovadamente impossível de se isolar a variável x nela. Mas não precisamos isolar x para calcularmos o valor de x quando $A(x)$ é algum valor fixo. Percebamos que a função $A(x)$ é *monótona* no intervalo $[0, r]$. Assim, tudo o que precisamos fazer é uma busca binária neste intervalo até encontrarmos a área desejada, sob uma precisão de, digamos, 4 casas decimais — só por segurança. Eis o código:

. . . ou monotônica. . .

. . . e só por segurança também trabalharemos com `long double`*!*

```
#include<stdio.h>
#include<math.h>

long double area(long double r, long double x) {
  long double raiz=sqrtl(2*r*x-x*x);
  return r*r*asinl(raiz/r) - (r-x)*raiz;
}

long double busca(long double r, long double A) {
  long double a=0, b=r, m, Am;
  while (b - a > 0.0001) {
    m = (a + b) / 2.0;
    Am = area(r, m);
    if (Am == A) return m;
    if (Am < A) a = m;
    else b = m;
  }
  if (A - area(r, a) < area(r, b) - A) return a;
  return b;
}

int main(void) {
  int d, k, j, tV, primeiro;
  long double r, A, V[5];
  scanf("%d %d", &d, &k);
  while (d) {
    if (k != 1) {
      tV = 0; primeiro = 1;
      r = (long double)d / 2.0;
      A = M_PI*r*r;
      for (j = 1; j <= (k-1)/2; j++) {
        V[tV] = busca(r, j*A/(long double)k);
        if (!primeiro) printf(" ");
        else primeiro = 0;
        printf("%.2Lf", V[tV++]);
      }
      if (k % 2 == 0) {
        if (!primeiro) printf(" ");
        printf("%.2Lf", r);
      }
      for (j = tV-1; j >= 0; j--)
        printf(" %.2Lf", r+r-V[j]);
    }
    printf("\n");
    scanf("%d %d", &d, &k);
  }
  return 0;
}
```

Problema C

O Jogo do Dicionário

Arquivo: `dicionario.[c|cpp|java]`

Problema escrito por Leandro Zatesko, UFFS.

Alice e Bob desenvolveram um jogo muito interessante, o qual batizaram de *Jogo do Dicionário*. No jogo, Alice fornece a Bob uma coleção de palavras do dicionário, selecionadas aleatoriamente. Bob precisa, então, num tempo cronometrado, formar um *ciclo* com o menor número possível das palavras recebidas. Um *ciclo* é uma sequência de no mínimo 3 palavras em que cada palavra difere da sua anterior em apenas uma letra, assumindo-se que a palavra anterior da primeira é a última. Ou seja, cada palavra pode ser obtida a partir da palavra anterior no ciclo através da remoção, adição ou substituição de uma letra. Por exemplo,

cura pura pra pera vera era cera

Veja que, no exemplo, a palavra anterior a 'cura' é 'cera'.

é um ciclo com 7 palavras.

Entrada

A entrada é composta por vários casos de teste, cada um designando uma partida do Jogo do Dicionário. A primeira linha de cada caso de teste é constituída de um só inteiro positivo n ($n \leqslant 10^3$), indicando o número de palavras selecionadas por Alice na partida em questão. Seguem, então, n linhas, cada uma contendo uma das n palavras. As palavras são formadas apenas por algarismos e letras minúsculas do alfabeto latino, e o número de letras numa palavra é no mínimo 1 e no máximo 10. Por fim, o valor $n = 0$ encerra a entrada.

Saída

Para cada caso de teste, seu programa deverá imprimir o número de palavras no menor ciclo que é possível Bob formar com as palavras dadas por Alice. Se não é possível formar ciclo algum, seu programa deverá imprimir a palavra `infinito`. Saídas correspondentes a casos de teste consecutivos devem ser separadas por uma quebra-de-linha.

Exemplo de entrada	Saída correspondente
7	3
cura	infinito
pura	infinito
pra	
pera	
vera	
era	
cera	
3	
cera	
cura	
dura	
1	
oi	
0	

★ Resolução do Professor

Seja G o grafo definido por:

1. os vértices são as palavras que Alice fornece a Bob;
2. existe aresta entre dois vértices se e só se uma pode ser obtida a partir da outra através da remoção, adição ou substituição de uma letra.

Note que o grafo não é dirigido e não possui loops nem arestas múltiplas.

O que se pede é computar $g(G)$, ou seja, a *cintura* de G, ou seja, o comprimento do menor ciclo em G. Um modo eficiente de se fazer isso é através de uma simples busca em largura em G, como exibimos no Algoritmo 3.

```
CINTURA(G):
 1  g ← ∞;
 2  para todo u ∈ V(G), faça:
 3     visitado[u] = F; pai[u] = u; dist[u] = ∞;
 4  enquanto existe s ∈ V(G) com visitado[s] = F, faça:
 5     visitado[s] = V; dist[s] = 0; Q ← {s};
 6     enquanto Q ≠ ∅, faça:
 7        desenfileire u de Q;
 8        para todo w ∈ N_G(u), faça;
 9           se visitado[w] = F, então:
10              visitado[w] = V; pai[w] = u; dist[w] = dist[u] + 1;
11              enfileire w em Q;
12           senão:
13              v ← lca(u, w);
14              C ← dist[u] + dist[w] − 2 dist[v] + 1;
15              se C < g, então, g ← C;
16  devolva g.
```

Algoritmo 3

lca(u,w) significa lowest common ancestor, ou last common ancestor (ancestral comum mais baixo, ou último ancestral comum); é o vértice mais distante da raiz da árvore da busca que é ao mesmo tempo ancestral de u e de w.

Os demais detalhes de implementação falam por si só no código:

```
#include<stdio.h>
#include<string.h>

#define INFTO 10001

char iguais(char D[1000][11], int i, int j) {
  char I[11][11], J[11][11];
  int TI=0, TJ=0, k, l, t;
  for (k = 0; D[i][k] != '\0'; k++) {
    for (l = t = 0; D[i][l] != '\0'; l++)
      if (l != k)
        I[TI][t++] = D[i][l];
    I[TI++][t] = '\0';
  }
  for (k = 0; D[j][k] != '\0'; k++) {
    for (l = t = 0; D[j][l] != '\0'; l++)
      if (l != k)
        J[TJ][t++] = D[j][l];
    J[TJ++][t] = '\0';
  }
  for (k = 0; k < TI; k++)
    if (!strcmp(I[k], D[j]))
      return 1;
  for (k = 0; k < TJ; k++)
```

```
    if (!strcmp(J[k], D[i]))
      return 1;
  for (k = 0; k < TI && k < TJ; k++)
    if (!strcmp(I[k], J[k]))
      return 1;
  return 0;
}

int main(void) {
  char D[1000][11], vis[1000];
  int pai[1000], dist[1000], Q[1000], iQ, fQ;
  int n, i, j, c, G[1000][1000], d[1000];
  int g, C, contvis, s, u, w, pu, pw;
  scanf("%d", &n);
  while (n > 0) {
    scanf("\n");
    for (i = 0; i < n; i++) {
      for (j = 0; (c = getchar()) != '\n'; j++)
        D[i][j] = c;
      D[i][j] = '\0';
    }
    for (i = 0; i < n; i++) d[i] = 0;
    for (i = 1; i < n; i++)
      for (j = 0; j < i; j++)
        if (iguais(D, i, j)) {
          G[i][d[i]++] = j;
          G[j][d[j]++] = i;
        }
    for (i = 0; i < n; i++) {
      vis[i] = 0; pai[i] = i;
    }
    g = INFTO; contvis = 0;
    while (contvis < n && g > 3) {
      for (s = 0; vis[s]; s++);
      vis[s] = 1; dist[s] = 0; contvis++;
      Q[iQ = fQ = 0] = s;
      while (g > 3 && iQ <= fQ) {
        u = Q[iQ++];
        for (i = 0; i < d[u]; i++) {
          w = G[u][i];
          if (!vis[w]) {
            vis[w] = 1;
            contvis++;
            pai[w] = u;
            dist[w] = dist[u] + 1;
            Q[++fQ] = w;
          } else if (w != pai[u]) {
            /* calcula lca(u,w) = lowest common ancestor */
            for (pu = u; dist[pu] > dist[w]; pu = pai[pu]);
            for (pw = w; pu != pw; pu = pai[pu], pw = pai[pw]);
            /* calcula comprimento do ciclo encontrado */
            C = dist[w] + dist[u] - 2*dist[pu] + 1;
            if (C < g) {
              g = C;
            }
```

```
          }
        } /* for */
      } /* while (g > 3 && iQ <= fQ) */
    } /* while (contvis < n) */
    if (g == INFTO) printf("infinito\n");
    else printf("%d\n", g);
    scanf("%d", &n);
  }
  return 0;
}
```

Problema D

Os Teoremas de Fermat

Arquivo: `fermat.[c|cpp|java]`

Problema escrito por Leandro Zatesko, UFFS.

Um dos teoremas mais antigos e mais importantes da humanidade, o Teorema de Pitágoras (séc. VI a.C.), estabelece, como observado pelo matemático grego Diofanto de Alexandria (séc. III d.C.) em sua obra magna *Arithmetica*, que a equação

$$x^2 + y^2 = z^2$$

possui infinitas soluções em $\mathbb{Z}^3$. Nas margens duma cópia da versão em latim de *Arithmetica*, o matemático Pierre de Fermat (séc. XVII) escreveu que a equação

Era comum a partir da Renascença que os livros trouxessem grandes espaços nas margens, destinados a eventuais anotações dos leitores.

$$x^n + y^n = z^n$$

não possui soluções em $\mathbb{Z}^n$ quando $n > 2$. Fermat escreveu: "Eu descobri uma verdadeiramente maravilhosa demonstração para isso, a qual esta margem é pequena demais para conter". No entanto, a referida demonstração nunca foi encontrada, se é que algum dia de fato existiu. O Teorema de Fermat intrigou os matemáticos por séculos, até que em 1995, finalmente, o matemático britânico Andrew J. Willes completou a demonstração.

. . . ou Último Teorema de Fermat, como também é conhecido

Outro teorema de Fermat, também bastante conhecido e de imensa aplicabilidade, é o Pequeno Teorema de Fermat, cuja demonstração consta num trabalho de Fermat de 1640. O teorema diz que, para todo inteiro não-nulo a e todo número primo positivo p, o resto da divisão da a^{p-1} por p é 1. Nem sempre que o resto da divisão de a^{n-1} por n é 1 podemos dizer que n é primo. Se n não é um número primo, mas o resto da divisão de a^{n-1} por n é 1, então, dizemos que n é um *pseudoprimo* para a base a. Graças a resultados embasados no Pequeno Teorema de Fermat, o Teste de Miller-Rabin, um algoritmo probabilístico para o *Problema da Primalidade*, testa se um número é primo muito rapidamente, sujeito a uma probabilidade de erro muito pequena.

Nestes mais de 350 anos em que o Teorema de Fermat carecia de demonstração, muito da Álgebra Moderna foi desenvolvido graças à motivação de se provar o teorema.

Se o resto da divisão de a^{n-1} por n <u>não</u> é 1, temos a certeza de que n não é primo. Por esse motivo, o Teste de Miller-Rabin nunca erra quando responde que um número é composto. Mas é possível que pseudoprimos se passem por primos para o algoritmo. Um algoritmo polinomial determinístico para o Problema da Primalidade foi descoberto em 2002 pelos indianos M. Agrawal, N. Kayal e N. Saxena.

Entrada

Cada linha da entrada, à exceção da última, define um caso de teste a ser avaliado. Cada caso é composto por dois inteiros positivos a e n ($a, n < 2^{31}$), separados por um espaço. Os valores $a = n = 0$ encerram a entrada e não constituem um caso de teste.

Saída

Para cada caso de teste, seu programa deverá imprimir o resto da divisão de a^{n-1} por n. Saídas correspondentes a casos de teste consecutivos devem ser separadas por uma quebra-de-linha.

Exemplo de entrada	Saída correspondente
1 1	0
2 6	2
2 341	1
0 0	

★ Resolução do Professor

Bom, qualquer um que tenha feito um curso de Criptografia sabe destas coisas :P

É óbvio que a e n cabem num `int`, no entanto, a^{n-1} pode ser da ordem de $(2^{31})^{(2^{31})}$, um número muito, muito grande, que requer 7,75 GiB de memória. Mas, por que primeiro calcularmos a^{n-1} para depois tomarmos o resto da divisão de a^{n-1} por n? Afinal, sabemos que

$$\text{se} \quad x_1 \equiv y_1 \,(mod n) \quad \text{e} \quad x_2 \equiv y_2 \,(mod n), \quad \text{então,} \quad x_1 x_2 \equiv y_1 y_2 \,(mod n).$$

Então, basta que todas as multiplicações que fizermos para calcularmos a^{n-1} façamos *módulo* n. Mesmo assim, cada multiplicação deve ser feita num `long long`, pois, antes de tirarmos o resto da divisão por n, uma multiplicação pode precisar de 62 *bits*.

Agora, quantas multiplicações precisamos fazer para termos a^{n-1} *módulo* n? Alguém mais iniciante provavelmente faria $n-1$ multiplicações, e com certeza seu algoritmo não seria aceito, pois n pode ser $2^{31}-1 = 2\,147\,483\,647 > 10^9$. No entanto, no pior caso, de n ser $2^{31}-1$, apenas aproximadamente 31 multiplicações são necessárias, pois

$$a^n = \begin{cases} 1, & \text{se } n = 0; \\ a \cdot a^{n-1}, & \text{se } n > 0 \text{ e } n \text{ é ímpar;} \\ a^{\frac{n}{2}} \cdot a^{\frac{n}{2}}, & \text{se } n > 0 \text{ e } n \text{ é par.} \end{cases}$$

Esta técnica é conhecida como *exponenciação logarítmica*, e calcula a^n com apenas $O(\log n)$ multiplicações.

```
#include<stdio.h>

int expmod(int a, int x, int n) {
  unsigned long long r;
  if (x == 0) return 1 % n;
  if (x % 2) {
    r = (unsigned long long)a * expmod(a, x-1, n) % n;
    return (int)r;
  }
  r = expmod(a, x/2, n);
  r = r * r % n;
  return (int)r;
}

int main(void) {
  int a, n;
  scanf("%d %d", &a, &n);
  while (a) {
    printf("%d\n", expmod(a, n-1, n));
    scanf("%d %d", &a, &n);
  }
  return 0;
}
```

Problema E

Palíndromos

Arquivo: `palindromos.[c|cpp|java]`

A Academia Brasileira de Letras está tentando registrar e catalogar os palíndromos na língua portuguesa. No entanto, o volume de frases que precisa julgar tem crescido a cada dia, e os autores estão realmente se superando, escrevendo palíndromos cada vez maiores. Um palíndromo é uma frase que, lida de trás para a frente, se iguala à frase lida no sentido usual, desconsiderando-se em ambas as leituras diferenciação entre maiúsculas e minúsculas, diacríticos, espaços, pontuações e quaisquer outros símbolos especiais. São exemplos de palíndromos na língua portuguesa as frases:

Problema escrito por Leandro Zatesko, UFFS.

- A grama é amarga. — *Millôr Fernandes*
- Ai, Ravel, arados, serrotes e torres só Dara levaria. — *Marcelo Furtado*
- A diva em Argel alegra-me a vida. — *Rômulo Marinho*
- Até Reagan sibarita tira bisnaga ereta. — *Chico Buarque de Holanda*
- E toma, leva, roda a missa. Reza fará prazer... e lave-me, ótimo é o demo, evite o azar, evite-se esse mês. Acorde pedroca sem esse (e se tive razão, e tive). O medo é o mito e me vale rezar para fazer assim. À Adorável, amo-te. — *Rogério Duarte Filho*

Entrada

A entrada consiste de uma lista de frases. Cada frase é composta por no mínimo um e no máximo 300 caracteres válidos, sendo ao menos um deles um caractere considerável. São caracteres consideráveis algarismos e letras do alfabeto latino maiúsculas e minúsculas sem diacríticos. Além dos caracteres consideráveis, são caracteres válidos o caractere de espaço em branco e os caracteres especiais: `. , ; : ! ? - ( )`

Os diacríticos usados na língua portuguesa são ´, `, ˜, ˆ, ¸ e ¨, mas não se preocupe com eles!

Frases consecutivas são separadas por uma quebra-de-linha, e a entrada é finalizada por fim-de-arquivo (*EOF*, *end-of-file*).

Saída

Seu programa deverá imprimir, de todas as frases lidas, quais são palíndromos. Após cada um dos palíndromos impressos, seu programa deverá imprimir uma quebra-de-linha.

Exemplo de entrada	**Saída correspondente**
OVo	OVo
AB Bab, ABba!!!	AB Bab, ABba!!!
Socorram-me, subi no onibus em Marrocos!	Socorram-me, subi no onibus em Marrocos!
Anotaram a data da Maratona?	Anotaram a data da Maratona?
A droga da gorda	A droga da gorda
Assam a massa	Assam a massa
Assam as massa	Ui! Ac! A torre da derrota caiu!
Ui! Ac! A torre da derrota caiu!	Saudavel leva duas!
Saudavel leva duas!	1234a??4321
olvo	
0100	
1234a??4321	

★ Resolução do Professor

Ou melhor, vou fazer um comentário sim: Se você não consegue resolver nem este problema, então, talvez não seja uma boa ideia você competir na Maratona agora. Vá estudar um pouco mais primeiro. Quem sabe ano que vem!

Sem comentários!

```
#include<stdio.h>

int main(void) {
  char F[301], flag;
  int G[301], tF, tG, c, i, j;
  while ((c = getchar()) != EOF) {
    tF = tG = 0;
    while (c != '\n') {
      if (c >= 'a' && c <= 'z') {
        F[tF++] = c; G[tG++] = c - 'a';
      } else if (c >= 'A' && c <= 'Z') {
        F[tF++] = c; G[tG++] = c - 'A';
      } else if (c >= '0' && c <= '9') {
        F[tF++] = c; G[tG++] = c - '0' + 26;
      } else F[tF++] = c;
      c = getchar();
    }
    F[tF++] = '\0';
    for (flag = 1, i = 0, j = tG-1; flag && i < j; i++, j--)
      if (G[i] != G[j]) flag = 0;
    if (flag) printf("%s\n", F);
  }
  return 0;
}
```

Problema F

Resta-Um Triangular

Arquivo: `resta-um.[c|cpp|java]`

Problema escrito por Leandro Zatesko, UFFS.

Resta-Um é um popular jogo para solitários bastante simples. Numa variante do jogo, conhecida como *Resta-Um Triangular*, o tabuleiro é constituído de 15 casas, dispostas na forma de um triângulo. Inicialmente, todas as casas, à exceção da casa de uma das pontas, estão ocupadas por pinos. O objetivo do jogador é realizar uma sequência de movimentos que leve a restar apenas um pino. Um movimento no *Resta-Um* consiste em pular um pino x sobre uma casa ocupada por um pino y, parando x numa casa anteriormente vazia e removendo-se y do tabuleiro, como na Figura.

. . . como no jogo de Damas.

Entrada

A entrada possui vários casos de teste, sendo cada um uma matriz de caracteres de 5 linhas por 9 colunas, representando uma configuração qualquer do tabuleiro do *Resta-Um Triangular*: `0` representa uma casa vazia, e `1`, uma casa ocupada por um pino. Entre duas casas consecutivas de uma mesma linha há um espaço, e em todas as linhas de um caso de teste, à exceção da última, há espaços em branco extras à esquerda e à direita para completar o número exato de 9 caracteres por linha. Ademais, é certo que em cada caso de teste ao menos uma casa é ocupada por um pino. A primeira linha da entrada é constituída de um só inteiro N ($N \leqslant 2^{15}$) indicando o número de casos de teste.

Cuidado com os espaços em branco no final das linhas!

Saída

Para cada caso de teste seu programa deverá imprimir numa linha um inteiro M, o qual representa o número mínimo de movimentos necessários para se restar apenas um pino no tabuleiro. Se já resta só um pino, então `0` deve ser impresso. Se não é possível restar um só pino a partir da configuração daquele caso de teste, seu programa deverá imprimir a linha

```
caso impossivel
```

Exemplo de entrada	Saída correspondente

2
`    0`
`   1 1`
`  1 1 1`
`1 1 1 1`
`1 1 1 1 1`
`    0`
`   0 0`
`  0 0 0`
`1 0 0 1`
`1 0 0 0 1`
	13
	`caso impossivel`

★ Resolução do Professor

Primeiramente notemos que, se uma determinada configuração de um tabuleiro é solúvel, então, ela seguramente é solúvel em *exatamente* $m-1$ movimentos, sendo m o número de pinos na configuração. Afinal, cada movimento remove um pino, e o objetivo do jogo, como o próprio nome já diz, é fazer restar um só pino. Então, tudo o que queremos saber é, dada uma configuração do tabuleiro, se aquela configuração é solúvel ou não. No caso do Resta-Um Triangular, existem trabalhos que estabelecem condições necessárias e suficientes para uma configuração ser solúvel, mas provavelmente você não saberia desses trabalhos no momento da competição. Sem problemas! Afinal, o número de configurações possível não é assim tão grande: $2^{15} = 32768$.

Como queremos apenas saber se uma configuração é solúvel, podemos simplesmente executar uma busca em profundidade no grafo de todas as configurações partindo da configuração inicial. Com certeza essa solução seria aceita, desde que não sejamos burros e guardemos o conhecimento adquirido nos casos de teste já processados. Aliás, como o enunciado diz que o número de casos de teste é no máximo 2^{15}, até já podemos presumir que todas as configurações serão testadas, embora não saibamos em qual ordem. Então, podemos previamente computar a solubilidade de todas as 2^{15} configurações antes mesmo de ler a entrada. Daí, para cada configuração lida, apenas fazemos uma consulta para sabermos se aquela configuração é solúvel ou não.

É raro um problema exigir essa malandragem de guardar o conhecimento dos casos de teste já processados. Mas este é um caso! :)

Outra forma de se atacar este problema é com *backtracking*, desde que guardemos a informação de solubilidade para todas as configurações e façamos a poda justamente para evitarmos que o *backtracking* explore caminhos que já sabemos não dar em lugar algum. Mesmo na abordagem com *backtracking* se faz necessário reaproveitar informação entre os casos de teste.

Um detalhe de implementação muito útil tanto para a solução com busca em profundidade quanto para a solução com *backtracking* vale comentar. Tratar as configurações dos tabuleiros como matrizes para indexar, por exemplo, informações de solubilidade pode ser um tanto que trabalhoso. Seria muito melhor se cada uma das configurações fosse vista como um inteiro entre 0 e $2^{15}-1$. Mas isso é muito fácil de se fazer, pois cada posição numa configuração é ou 0, ou 1. Podemos rapidamente implementar uma função que transforme uma configuração num inteiro e uma função que destransforme um inteiro numa configuração.

Lembrando que, do ponto de vista teórico, a técnica de backtracking pode ser enxergada como uma busca em profundidade.

Exibimos a seguir a solução com *backtracking*, apenas para o leitor se familiarizar com a técnica. O *vetor-solução* $\boldsymbol{V}[1..15]$ do *backtracking* começará a ser preenchido na posição m, sendo m o número de pinos na configuração do tabuleiro lida, com o inteiro correspondente à configuração lida. O *backtracking* tentará, então, preencher as posições $m-1$, $m-2$, e assim por diante. Se o *backtracking* conseguir preencher a posição 1, então, encontrou uma solução. Teremos também dois vetores: $\boldsymbol{vis}[0..32767]$ e $\boldsymbol{I}[0..32767]$:

- $\boldsymbol{vis}[i] = 1$ se a configuração correspondente ao inteiro i já foi visitada pelo *backtracking*, e $\boldsymbol{vis}[i] = 0$ caso contrário;
- $\boldsymbol{I}[i] = 1$ se já sabemos que a configuração correspondente ao inteiro i é insolúvel, e $\boldsymbol{I}[i] = 0$ caso contrário.

```
#include<stdio.h>

char achou_sol;

int Ttoint(char T[5][5]) {
  int s=0, i, j;
  for (i = 0; i < 5; i++)
    for (j = 0; j <= i; j++)
      s = (s << 1) + T[i][j];
  return s;
}
```

```
void inttoT(int s, char U[5][5]) {
  int i, j;
  for (i = 4; i >= 0; i--)
    for (j = i; j >= 0; j--) {
      U[i][j] = s & 1;
      s >>= 1;
    }
}

void pule(char T[5][5], char U[5][5],
          int iT, int jT, int iM, int jM, int iU, int jU) {
  int i, j;
  for (i = 0; i < 5; i++)
    for (j = 0; j <= i; j++)
      U[i][j] = T[i][j];
  U[iT][jT] = 0;
  U[iM][jM] = 0;
  U[iU][jU] = 1;
}

void construa_os_candidatos(int V[16], int k,
                            char vis[32768], char I[32768],
                            int c[90], int *m) {
  char T[5][5], U[5][5];
  int i, j, u;
  inttoT(V[k+1], T);
  *m = 0;
  for (i = 0; i < 5; i++)
    for (j = 0; j <= i; j++)
      if (T[i][j] == 1) {
        if (i >= 2 && j <= i-2)
          if (T[i-1][j] == 1 && T[i-2][j] == 0) {
            pule(T,U, i,j, i-1,j, i-2,j);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
        if (i >= 2 && j >= 2)
          if (T[i-1][j-1] == 1 && T[i-2][j-2] == 0) {
            pule(T,U, i,j, i-1,j-1, i-2,j-2);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
        if (j >= 2)
          if (T[i][j-1] == 1 && T[i][j-2] == 0) {
            pule(T,U, i,j, i,j-1, i,j-2);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
        if(j <= i-2)
          if(T[i][j+1] == 1 && T[i][j+2] == 0) {
            pule(T,U, i,j, i,j+1, i,j+2);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
        if (i < 3) {
```

```
          if (T[i+1][j] == 1 && T[i+2][j] == 0) {
            pule(T,U, i,j, i+1,j, i+2,j);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
          if (T[i+1][j+1] == 1 && T[i+2][j+2] == 0) {
            pule(T,U, i,j, i+1,j+1, i+2,j+2);
            u = Ttoint(U);
            if (!vis[u] && !I[u]) c[(*m)++] = u;
          }
        }
      }
}

void backtrack(int V[16], int k, char vis[32768], char I[32768]) {
  int c[90], m, j;
  if (k == 1) {
    achou_sol = 1;
    return;
  }
  k--;
  construa_os_candidatos(V, k, vis, I, c, &m);
  for (j = 0; j < m; j++) {
    V[k] = c[j];
    vis[V[k]] = 1;
    backtrack(V, k, vis, I);
    if (achou_sol) return;
    I[V[k]] = 1;
  }
}

int main(void) {
  char T[5][5], I[32768], vis[32768];
  int i, j, c, s, m, V[16], N;
  /* I[i] = 1 <=> i não tem solução */
  /* vis[i] = 1 <=> i já foi visitado pelo backtracking */
  for (i = 0; i < 32768; i++) I[i] = 0;
  scanf("%d\n", &N);
  while(N--) {
    for (i = 0; i < 32768; i++) vis[i] = 0;
    m = 0; achou_sol = 0;
    for (i = 0; i < 5; i++) {
      for (j = 0; j <= i; j++) {
        while ((c = getchar()) == ' ');
        T[i][j] = c - '0';
        if (T[i][j]) m++;
      }
      while ((c = getchar()) != '\n');
    }
    s = Ttoint(T);
    V[m] = s;
    vis[s] = 1;
    backtrack(V, m, vis, I);
    if (achou_sol) printf("%d\n", m-1);
    else printf("caso impossivel\n");
```

```
  } /* while((c = getchar()) != EOF) */
  return 0;
}
```