Corrida ao Banco

nome do arquivo/classe: banco.c , banco.cpp ou banco.java - tempo: 3 segundos

Você está no centro da cidade e acaba de lembrar que em sua mochila há uma conta que vence hoje e os bancos irão fechar em poucos minutos.

Sua tarefa é desenvolver um código que informe qual é a distância ao banco mais próximo da sua posição, dado um mapa bidimensional onde constam: (i) a sua posição; (ii) um ou mais bancos; (iii) alguns obstáculos. Lembre-se: faltam poucos minutos para os bancos fecharem, então seu programa precisa ser eficiente! Você, obviamente, pode andar para a esquerda, direita, frente, trás e nas quatro diagonais.

Entrada

A entrada conterá de 1 a 50 casos de teste. Cada caso de teste será iniciado por uma linha contendo dois inteiros correspondendo ao tamanho (***L*** e ***C***) do mapa bidimensional do centro da sua cidade (tanto L como C terão seus valores podendo variar de 1 até 100). Em seguida existirão L linhas com C caracteres em cada correspondendo a cada uma das linhas do mapa. Há 4 caracteres possíveis utilizados para representar cada célula do mapa:

. (ponto): corresponde a um espaço vazio do mapa (por onde você pode andar)
B: corresponde a um banco
X: corresponde à posição inicial (onde você está)
#: corresponde a um obstáculo (pelo qual você não poderá andar)

Todo mapa conterá ao menos um banco e exatamente uma posição inicial.

A entrada será encerrada com dois valores 0 (zero) para L e C.

Saída

A saída consistirá de uma linha para cada caso de teste no seguinte formato:

- Se existir um caminho até um banco: "*Passos para o banco:* ***Y***" onde Y corresponde à menor quantidade de passos necessária para se chegar a um banco;

- Caso não exista nenhum caminho: "*Nenhum caminho encontrado!*" (não esqueça de imprimir a exclamação).

Exemplo de Entrada

```
20 20
....................
.B................B.#
....................
....................
....................
....................
....................
....................
.........#..........
.........X..........
....................
....................
.......###..........
........B...........
....................
....................
....................
....................
....................
....................
3 3
B#.
#X#
.#.
3 3
B#.
##.
..X
4 3
B..
B.X
B..
B..
4 3
B..
BX.
B..
B..
0 0
```

Exemplo de Saída

```
Passos para o banco: 5
Passos para o banco: 1
Nenhum caminho encontrado!
Passos para o banco: 2
Passos para o banco: 1
```

Duelo entre Equipes

nome do arquivo/classe: duelo.c , duelo.cpp ou duelo.java – tempo: 3 segundos

Desde os primórdios da civilização a rivalidade entre a equipe verde e a equipe amarela só vem aumento. A situação está chegando a um ponto insuportável.

Ao longo dos últimos anos os membros de cada equipe têm treinado bastante para ficarem mais fortes e se preparem para o grande embate. Esse embate é baseado em duelos entre os membros mais fortes de cada equipe. Em cada duelo há no máximo um vencedor e o derrotado será eliminado. Esses duelos ocorrerão até que só restem integrantes de uma única equipe.

Sua função é implementar um algoritmo que irá identificar qual equipe será a vencedora e quantos integrantes estarão em pé ao final da competição, de acordo com as seguintes regras:

- ocorrerá um duelo de cada vez, envolvendo sempre os integrantes mais fortes de cada equipe;

- durante um duelo há duas possibilidades: os dois integrantes que se enfrentaram tinham a mesma força e, então, nenhum deles sobreviverá ao duelo; ou um integrante é mais forte que o outro e, neste caso, este integrante sobreviverá enquanto o outro será eliminado. Porém, o integrante que sobreviveu terá sua força reduzida por causa do duelo da seguinte maneira: *força_reduzida = força_atual – metade_da_força_do_oponente* (ou seja, a força do vitorioso será igual a sua força antes do duelo subtraída por metade da força do oponente, se a força do oponente for um número ímpar, o valor a ser subtraído deve ser arredondado para baixo);

- após cada duelo, o vencedor (se houver) deve voltar para sua equipe (agora mais fraco) e, enquanto houver integrantes nas duas equipes, um próximo duelo será realizado entre os integrantes mais fortes de cada equipe.

Entrada

A entrada conterá um ou mais casos de teste. Cada caso de teste é composto por três linhas:

1. a primeira linha contém dois inteiros V e A (ambos variando de 1 a 1000) que correspondem a quantidade de integrantes das equipes verde e amarela, respectivamente;

2. a segunda linha conterá V números inteiros (cada um variando de 1 a 999999) correspondendo a força de cada um dos integrantes da equipe verde;

3. a terceira linha conterá A números inteiros (cada um variando de 1 a 999999) correspondendo a força de cada um dos integrantes da equipe amarela;

A entrada será encerrada por dois zeros.

Saída

Para cada caso de teste você deverá imprimir uma linha da seguinte forma:

- "*Verde ganhou: X*" caso a equipe verde tenha ganho o embate, onde X corresponde ao número de integrantes de equipe que 'sobreviveram';
- "*Amarelo ganhou: Y*" caso a equipe amarela tenha ganho o embate, onde Y corresponde ao número de integrantes de equipe que 'sobreviveram';
- "*Empate.*" caso tenha ocorrido um empate, isto é, ninguém sobreviveu (para este caso, não esqueça de imprimir o ponto final).

Exemplo de Entrada

```
34 45
100 97 94 91 88 85 82 79 76 73 70 67 64 61 58 55 52 49 46 43 40 37 34 31 28 25
22 19 16 13 10 7 4 1
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55
57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89
33 45
97 94 91 88 85 82 79 76 73 70 67 64 61 58 55 52 49 46 43 40 37 34 31 28 25 22
19 16 13 10 7 4 1
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55
57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89
10 5
8 8 8 8 8 8 8 8 8 8
16 16 16 16 16
2 2
8 4
4 6
1 1
10 10
2 2
8 4
4 7
2 2
8 1
6 7
2 2
8 4
6 6
0 0
```

Exemplo de Saída

```
Verde ganhou: 22
Amarelo ganhou: 29
Amarelo ganhou: 5
Verde ganhou: 2
Empate.
Verde ganhou: 2
Amarelo ganhou: 1
Empate.
```

Números Estranhos – Parte 1

nome do arquivo/classe: estranhos1.c , estranhos1.cpp ou estranhos1.java – tempo: 3 segundos

Diversos números possuem algumas propriedades específicas. Algumas curiosas, outras úteis e, por fim, algumas simplesmente estranhas. Seu professor de matemática tem por hobby elevar números ao quadrado e, por estar entediado, decidiu pedir para que você encontre alguns números que ele chamou de estranhos. Estes números são aqueles cujo quadrado possui exatamente a mesma quantidade de cada um dos valores (dígitos) do número original e, obviamente, podem possuir outros dígitos diferentes. Por exemplo, o número 11 se enquadra nesta definição pois o número 11 possui dois números 1 e seu quadrado (121) possui exatamente dois números 1 (além, é claro, de um número 2). Outro número "estranho" é o 305 que possuí um número 3, um número 0 e um número 5, seu quadrado é o 93025 que também possui exatamente um número 3, um número 0 e um número 5. Um terceiro exemplo de número estranho é o 27, cujo quadrado é 729. No exemplo de saída são apresentados outros números estranhos.

Seu trabalho é encontrar todos os números estranhos em um dado intervalo (incluindo os extremos do intervalo).

Entrada

A entrada será composta de um ou mais casos de teste e será encerrada com um fim-de-arquivo (*end-of-file*). Cada caso de teste conterá dois números inteiros (cujos valores podem variar de 1 a 100000) correspondendo ao limite do intervalo no qual os números estranhos serão procurados.

Saída

A saída consistirá de uma linha para cada caso de teste contendo a lista de todos os números estranhos (em ordem crescente) encontrados no intervalo solicitado. Cada número dever ser separado do outro por exatamente um espaço em branco e não deverá haver nenhum espaço em branco no fim da linha (após o último elemento).

Caso não haja nenhum número estranho no intervalo, uma linha em branco deverá ser impressa (sem nenhum espaço em branco nela).

Exemplo de Entrada

```
100 10
10 100
100 1000
2 3
99995 99996
1 20
```

Exemplo de Saída

```
11 25 27 63 64 74 95 96
11 25 27 63 64 74 95 96
142 255 277 305 364 376 405 463 497 504 507 508 509 524 593 602 625 644 645 676 705 724
755 766 805 825 867 871 905 969 974 976 995 996

99995 99996
1 5 6 11
```

Números Estranhos – Parte 2

nome do arquivo/classe: estranhos2.c , estranhos2.cpp ou estranhos2.java – tempo: 3 segundos

Diversos números possuem algumas propriedades específicas. Algumas curiosas, outras úteis e, por fim, algumas simplesmente estranhas. Seu professor de matemática (aquele mesmo que tem por hobby elevar números ao quadrado) ficou tão satisfeito com o programa que você fez para ele que resolveu te passar uma nova tarefa: identificar números ***mais estranhos***. Estes números são aqueles cujo quadrado contém ao menos a mesma quantidade dos dígitos do número original e na mesma ordem!!! Ou seja, dado o quadrado desse número "mais estranho" é possível apagar zero ou mais de seus dígitos e obter assim o número original ("Que propriedade mais fantástica!!!", pensou seu professor). Entre os números que satisfazem essa propriedade estão o número 11, cujo quadrado é 121 (e basta apagar o número 2 de 121 que ele volta a ser o número 11); o número 25, cujo quadrado é 625 (basta apagar o número 6 para voltar ao 25); e o número 99996, cujo quadrado é 9999200016.

Seu trabalho é encontrar todos os números *mais estranhos* em um dado intervalo (incluindo os extremos do intervalo).

Entrada

A entrada será composta de um ou mais casos de teste e será encerrada com um fim-de-arquivo (*end-of-file*). Cada caso de teste conterá dois números inteiros (cujos valores podem vaiar de 1 a 100000) correspondendo ao limite do intervalo no qual os números estranhos serão procurados.

Saída

A saída consistirá de uma linha para cada caso de teste contendo a lista de todos os números mais estranhos (em ordem crescente) encontrados no intervalo solicitado. Cada número dever ser separado do outro por exatamente um espaço em branco e não deverá haver nenhum espaço em branco no fim da linha (após o último elemento).

Caso não haja nenhum número mais estranho no intervalo, uma linha contendo a palavra "*nenhum*" deverá ser impressa.

Exemplo de Entrada

```
100 10
10 100
100 1000
2 3
99995 99996
1 20
```

Exemplo de Saída

```
10 11 25 50 60 76 95 96 100
10 11 25 50 60 76 95 96 100
100 101 105 110 125 205 250 305 371 376 405 441 500 501 505 506 525 600 601 605 625 676
705 756 760 805 825 826 905 946 950 960 976 995 996 1000
nenhum
99995 99996
1 5 6 10 11
```

Última da Fila

nome do arquivo/classe: fila.c , fila.cpp ou fila.java – tempo: 3 segundos

Uma das coisas mais chatas desta vida é estar numa longa fila. Porém, mais chato do que isso é ser o último a ser atendido na fila.

Cansado de ouvir reclamações dos últimos indivíduos de uma fila, o dono de uma loja teve uma "brilhante" ideia e resolveu inovar o atendimento das pessoas na fila: ao invés de atender o primeiro indivíduo da fila, depois o segundo e assim por diante ele criou uma estratégia própria: dada uma fila de n pessoas, ele dá uma senha de 1 a n para cada pessoa de acordo com a ordem de chegada (a primeira pessoa recebe a senha número 1, a segunda a senha número 2 e assim por diante). Em seguida ele sorteia um número k (de 1 a n) que indicará qual será a primeira pessoa a ser atendida. Esta pessoa é atendida (e consequentemente sai da fila) e em seguida o próximo a ser atendido será aquele que está x posições atrás dessa pessoa na fila (onde x corresponde ao valor da senha da última pessoa que foi atendida). Caso a fila chegue ao seu final, a contagem continua do início da fila ("fila circular") e assim por diante até que todos sejam atendidos.

Por exemplo, dada uma fila com 4 pessoas e se o k (valor sorteado) for igual a 1: a primeira pessoa a ser atendida será a pessoa com a senha igual a 1; após ser atendida a pessoa que esteja uma posição ("uma posição" porque a senha da pessoa que foi atendida era igual a 1) atrás da pessoa atendida será a próxima a ser atendida, ou seja, a pessoa com senha igual a 2. Após atender a esta pessoa, a próxima a ser atendida será a pessoa que estará duas posições atrás dela (pois sua senha era igual a 2), ou seja, a pessoa com senha igual a 4. Por fim, só resta a pessoa com senha igual a 3 para ser atendida.

Sua tarefa é identificar qual será a última pessoa a ser atendida, dados o número de pessoas na fila (***n***) e o valor sorteado para o primeiro atendimento (***k***).

Entrada e Saída

A entrada será composta de um ou mais casos de teste e será encerrada com um fim-de-arquivo (*end-of-file*). Cada caso de teste conterá dois números inteiros n e k (onde $1<n<2000$ e $1 \leq k \leq n$).

A saída contém uma linha para cada caso de teste com o número da senha da última pessoa a ser atendida.

Exemplo de Entrada

```
3 1
4 1
5 1
5 2
1000 1
```

Exemplo de Saída

```
3
3
5
3
195
```

Multiplicação do QuickSort

nome do arquivo/classe: multiplicacao.c , multiplicacao.cpp ou multiplicacao.java – tempo: 3 segundos

Diversas operações sobre conjuntos de números organizados em um arranjo precisam encontrar um número de interesse (geralmente chamado de pivô) e daí fazer algum processamento (muitas vezes recursivo) sobre os números que estão a esquerda e a direita do pivô). Um exemplo de programa que encontra um pivô em um arranjo e faz chamadas recursivas sobre dois subarranjos é o QuickSort (mas observem a seguir que a divisão feita aqui não é a mesma do QuickSort).

Seu desafio é implementar a seguinte função, chamada aqui de *fx*, que processa um arranjo de números inteiros:

*fx = s_esquerda_pivô * s_direita_pivô + pivô + fx(esquerda_pivô) + fx(direita_pivô)*

Onde: *pivô* é o elemento com maior valor entre aqueles que estão sendo examinados;
s_esquerda_pivô corresponde a soma dos valores dos elementos a esquerda do pivô;
s_direita_pivô corresponde a soma dos valores dos elementos a direita do pivô;
esquerda_pivô corresponde ao arranjo dos elementos que estão a esquerda do pivô;
direita_pivô corresponde ao arranjo dos elementos que estão a direita do pivô;

Notem que essa fórmula para o conjunto de um único número vale esse número, isto é, fx({y})=y; e definimos fx({})=0 (a fórmula para nenhum valor é igual a zero.

Dado o seguinte arranjo de seis números: {1, -2, 5, 2, -3}, o pivô (elemento de maior valor) é o 5, então a fórmula ficará:

fx({1, -2, 5, 2, -3}) = -1*-1 + 5 + fx({1,-2}) + fx({2,-3}) (equação 1)

fx({1,-2}) tem como pivô o 1, então tempos:

fx({1,-2}) = 0*-2 + 1 + fx({}) + fx({-2})
= 0 + 1 + 0 -2
= -1 (resultado 1)

fx({2,-3}) tem como pivô o 2, então tempos:

fx({2,-3}) = 0*-3 + 2 + fx({}) + fx({-3})
= 0 + 2 + 0 -3
= -1 (resultado 2)

Juntando os resultados (1) e (2) na equação (1) temos:

fx({1, -2, 5, 2, -3}) = -1*-1 + 5 -1 -1

fx({1, -2, 5, 2, -3}) = 4

Entrada

A entrada será composta por uma série de casos de teste. Cada caso de teste será composto por duas linhas. Na primeira linha de cada caso haverá um número N ($1 \leq N \leq 1000$) que indica a quantidade de números do arranjo. A segunda linha de cada caso de teste conterá os N números inteiros do arranjo a ser processado (separados por um espaço em branco), o valor de cada um desses números pode variar de -255 a +255. A entrada será encerrada por um número 0 (zero) para o valor de N.

Saída

Para cada caso de teste, seu programa deverá imprimir uma linha contendo o resultado da função fx para o respectivo arranjo de entrada.

Exemplo de Entrada

```
6
1 2 3 4 5 6
6
-3 -2 -1 1 2 3
6
3 2 1 -1 -2 -3
5
1 -2 5 2 -3
0
```

Exemplo de Saída

```
21
0
0
4
```

Percurso de Menor Custo

nome do arquivo/classe: percurso.c , percurso.cpp ou percurso.java – tempo: 3 segundos

Muitos problemas em computação envolvem percorrer todos os elementos de um grafo ou uma matriz e, eventualmente, passar por todos estes elementos de maneira a maximizar algum benefício ou minimizar os custos.

O objetivo deste problema é, dada uma matriz bidimensional e uma posição inicial, identificar qual será o menor custo para se passar exatamente uma vez por todos os elementos da matriz. Mas para este problema, a função de custo será calculada de uma maneira pouco convencional: o custo de sair da célula *a* para ir para célula *b* será equivalente a multiplicação dos valores destas duas células.

Dada a seguinte matriz 2x3 e a posição inicial 0,0 (onde 0,0 corresponde a coordenada do elemento cujo valor é 5):

5	-1	-2
3	4	1

Existem 3 maneiras de se percorrer todos os elementos dessa matriz a partir do elemento 5 (a direita de cada percurso está sendo apresentado seu custo):

5 => 3 => 4 => -1 => -2 => 1 custo: 23 = (5*3)+(3*4)+(4*-1)+(-1*-2)+(-2*1)
5 => -1 => -2 => 1 => 4 => 3 custo: 11 = (5*-1)+(-1*-2)+(-2*1)+(1*4)+(4*3)
5 => 3 => 4 => 1 => -2 => -1 custo: 31 = (5*3)+(3*4)+(4*1)+(1*-2)+(-2*-1)

Notem que o custo pode ser um número negativo.

Entrada

A entrada será composta de um ou mais casos de teste. Cada caso de teste conterá 3 partes. Na primeira parte haverá uma linha com dois inteiros ***L*** e ***C*** correspondendo ao número de linhas e de colunas da matriz (cada um deles valendo menos do que 7). Em seguida, na segunda parte de cada caso de teste, existirão *L* linhas com *C* números em cada (estes números serão separados por um espaço em branco). Cada número terá seu valor entre -255 e +255. Por fim, haverá uma linha contendo dois inteiros correspondendo a posição inicial da matriz (usando o sistema de coordenadas iniciado por 0,0).

A entrada será encerrada por dois valores 0 (zero) para L e C.

Saída

Para cada caso de teste, imprima o menor custo de percurso (que, é claro, envolva todas as células da matriz). Se não existir percurso possível, imprima 0 (zero).

Exemplo de Entrada

```
1 1
5
0 0
2 3
5 -1 -2
3 4 1
0 0
1 3
1 1 1
0 1
3 3
1 2 3
-3 -2 -1
1 -2 3
1 1
0 0
```

Exemplo de Saída

```
0
11
0
-22
```