acm International Collegiate Programming Contest 2007 IBM event sponsor

Maratona de Programação da SBC 2007

Sub-Regional Brasil do ACM ICPC

22 de Setembro de 2007

(Este caderno contém 10 problemas; as páginas estão numeradas de 1 a 19, não contando esta página de rosto)

Sedes Regionais

Região Centro-Oeste

- Universidade de Brasília, Brasília, DF
- Universidade Católica Dom Bosco, Campo Grande, MS
- Universidade Estadual do Mato Grosso do Sul, Dourados, MS

Região Nordeste

- Faculdades Guararapes, Jaboatão dos Guararapes, PE
- CEFET-PB, João Pessoa, PB
- CEFET-RN, Natal, RN
- Centro Universitário FIB, Salvador, BA
- Universidade Federal do Maranhão, São Luís, MA
- Universidade Federal do Ceará, Sobral, CE

Região Sul

- Unioeste, Cascavel, PR
- FTECBrasil, Caxias do Sul, RS
- Universidade Federal do Paraná, Curitiba, PR
- Universidade Federal de Santa Catarina, Florianópolis, SC
- Universidade Federal do Rio Grande do Sul, Porto Alegre, RG
- FURG, Rio Grande, RS

Região Sudeste

- UniToledo, Araçatuba, SP
- Faculdade Politécnica de Campinas, Campinas, SP
- Centro Universitário Módulo, Caraguatatuba, SP
- Univale, Governador Valadares, MG
- Universidade Salgado de Oliveira, Juiz de Fora, MG
- FINOM, Paracatu, MG
- PUC Poços de Caldas, Poços de Caldas, MG
- UFF-Puro, Rio das Ostras, RJ
- PUC-Rio, Rio de Janeiro, RJ
- Unisantos, Santos, SP
- Mackenzie, São Paulo, SP
- Unitau, Taubaté, SP
- Unitri, Ubertlândia, MG
- Centro Universitário de Vila Velha, Vila Velha, ES

Região Norte

- Instituto de Estudos Superiores da Amazônia, Belém, PA
- Uniron, Porto Velho, RO

Promoção:

Sociedade Brasileira de Computação

Patrocínio:

Fundação Carlos Chagas

Problema A
Vôlei Marciano

Nome do arquivo fonte: `volei.c`, `volei.cpp`, `volei.java` *ou* `volei.pas`

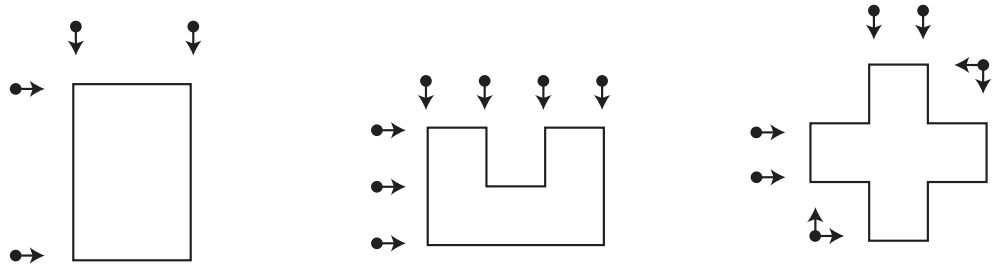
Assim como na Terra, o vôlei é um esporte muito popular em Marte; as regras lá são as mesmas do vôlei terrestre — os times não devem deixar a bola tocar na sua metade da quadra — mas há uma importante diferença: ao contrário do vôlei terrestre, lá as quadras não são necessariamente retangulares; elas podem ser polígonos quaisquer, desde que seus lados sejam paralelos aos eixos coordenados.

Assim como no vôlei terrestre, os lances polêmicos são aqueles em que a bola cai muito próxima à linha da quadra. Para evitar discussões, todos os jogos de vôlei marciano são acompanhados por *juízes de linha*. A função deles é observar a bola quando ela cai próxima a uma das linhas e dizer se ela caiu dentro ou fora da quadra.

Quando um juiz está alinhado com várias linhas da quadra, ele pode observar todas elas ao mesmo tempo (no conjunto de linhas sob responsabilidade de um mesmo juiz pode haver até linhas perpendiculares entre si). No entanto, para evitar acidentes, a Federação Intergaláctica de Vôlei Marciano decretou as seguintes normas de segurança:

- os juízes devem ficar parados durante o jogo;
- os juízes não podem ficar dentro da quadra, nem mesmo sobre a sua linha.

A figura abaixo ilustra três formatos de quadras possíveis, mostrando uma alocação mínima de juízes para cada uma delas; a quadra (a) necessita de quatro juízes, a quadra (b) necessita de sete juízes, e a quadra (c) necessita de seis juízes.



(a) (b) (c)

Você deve escrever um programa que, dado o formato da quadra, determina o número mínimo de juízes de linha necessários para que todas as linhas da quadra sejam acompanhadas por pelo menos um juiz.

Entrada

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém um inteiro par N, que indica o número de lados da quadra de vôlei ($4 \leq N \leq 100$). Cada uma das N linhas seguintes contém dois números inteiros X_i e Y_i, representando as coordenadas de um dos vértices da quadra ($-1.000.000.000 \leq X_i, Y_i \leq 1.000.000.000$). As coordenadas são dadas em ordem, de modo que (X_i, Y_i) forma um lado da quadra com (X_{i+1}, Y_{i+1}), para

$1 \leq i < N$, e (X_N, Y_N) forma um lado com (X_1, Y_1). Lados consecutivos da quadra são sempre perpendiculares, e o polígono descrito na entrada é sempre um polígono simples.

O final da entrada é indicado por $N = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma única linha na saída, contendo um número inteiro, indicando o menor número de juízes de linha necessários.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
4 0 0 9 0 9 18 0 18 8 0 0 0 1 1 1 1 -1 -2 -1 -2 1 -1 1 -1 0 12 1 0 2 0 2 1 3 1 3 2 2 2 2 3 1 3 1 2 0 2 0 1 1 1 0	4 7 6

Problema B
Rouba-Monte

Nome do arquivo fonte: `rouba.c`, `rouba.cpp`, `rouba.java` *ou* `rouba.pas`

Um dos jogos de cartas mais divertidos para crianças pequenas, pela simplicidade, é Rouba-Monte. O jogo utiliza um ou mais baralhos normais e tem regras muito simples. Cartas são distingüidas apenas pelo valor (ás, dois, três, ...), ou seja, os naipes das cartas não são considerados (por exemplo, ás de paus e ás de ouro têm o mesmo valor).

Inicialmente as cartas são embaralhadas e colocadas em uma pilha na mesa de jogo, chamada de pilha de compra, com face voltada para baixo. Durante o jogo, cada jogador mantém um *monte* de cartas, com face voltada para cima; em um dado momento o monte de um jogador pode conter zero ou mais cartas. No início do jogo, todos os montes dos jogadores têm zero cartas. Ao lado da pilha de compras encontra-se uma área denomindada de *área de descarte*, inicialmente vazia, e todas as cartas colocadas na área de descarte são colocadas lado a lado com a face para cima (ou seja, não são empilhadas).

Os jogadores, dispostos em um círculo ao redor da mesa de jogo, jogam em sequência, em sentido horário. As jogadas prosseguem da seguinte forma:

- O jogador que tem a vez de jogar retira a carta de cima da pilha de compras e a mostra aos outros jogadores; vamos chamar essa carta de *carta da vez*.
- Se a carta da vez for igual a alguma carta presente na área de descarte, o jogador retira essa carta da área de descarte colocando-a, juntamente com a carta da vez, no topo de seu monte, com as faces voltada para cima, e continua a jogada (ou seja, retira outra carta da pilha de compras e repete o processo).
- Se a carta da vez for igual à carta de cima de um monte de um outro jogador, o jogador "rouba" esse monte, empilhando-o em seu próprio monte, coloca a carta da vez no topo do seu monte, face para cima, e continua a jogada.
- Se a carta da vez for igual à carta no topo de seu próprio monte, o jogador coloca a carta da vez no topo de seu próprio monte, com a face para cima, e continua a jogada.
- Se a carta da vez for diferente das cartas da área de descarte e das cartas nos topos dos montes, o jogador a coloca na área de descarte, face para cima, e a jogada se encerra (ou seja, o próximo jogador efetua a sua jogada). Note que esse é o único caso em que o jogador não continua a jogada.

O jogo termina quando não há mais cartas na pilha de compras. O jogador que tiver o maior monte (o monte contendo o maior número de cartas) ganha o jogo. Se houver empate, todos os jogadores com o monte contendo o maior número de cartas ganham o jogo.

Entrada

A entrada é composta de vários casos de teste. A primeira linha de um caso de teste contém dois inteiros N e J, representando respectivamente o número de cartas no baralho ($2 \leq N \leq 10.000$)

e o número de jogadores ($2 \leq J \leq 20$ e $J \leq N$). As cartas do baralho são representadas por números inteiros de 1 a 13 e os jogadores são identificados por inteiros de 1 a J. O primeiro jogador a jogar é o de número 1, seguido no jogador de número 2, ..., seguido pelo jogador de número J, seguido pelo jogador de número 1, seguido do jogador de número 2, e assim por diante enquanto houver cartas na pilha de compras. A segunda linha de um caso de teste contém N inteiros entre 1 e 13, separados por um espaço em branco, representando as cartas na pilha de compras. As cartas são retiradas da pilha de compras na ordem em que aparecem na entrada. O final da entrada é indicado por uma linha com $N = J = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste seu programa deve imprimir uma linha, contendo o número de cartas do monte do jogador ou jogadores que ganharam a partida, seguido de um espaço em branco, seguido do(s) identificador(es) dos jogadores que ganharam a partida. Se há mais de um jogador vencedor imprima os identificadores dos jogadores em ordem crescente, separados por um espaço em branco.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**

`4 2`
`10 7 2 5`
`6 3`
`1 2 1 2 1 2`
`8 2`
`3 3 1 1 2 3 4 5`
`0 0`
	`0 1 2`
	`5 1`
	`3 2`

Problema C
Série de Tubos

Nome do arquivo fonte: `tubos.c`, `tubos.cpp`, `tubos.java` *ou* `tubos.pas`

O ano é 2010. O espetacular resultado de um projeto ultra-secreto, iniciado três anos antes por um grupo de pesquisadores da SBC (Soluções Brasileiras em Cabeamento) está prestes a ser divulgado: a SBC conseguiu a proeza de transportar matéria através de cabos de fibra ótica! A pesquisa contraria a famosa e polêmica frase do ex-senador e atual presidente dos EUA, que na época do início da pesquisa, há três anos, afirmara que "a internet não é como um caminhão de carga, em que você despeja o que quiser; a internet na verdade é uma série de tubos". Com isso, a SBC, que atualmente aluga a sua rede de cabos para uma operadora de TV paga, pensa em mudar de negócio e iniciar-se na atividade de transporte de carga — apesar de a tecnologia desenvolvida servir também para o transporte de seres vivos, há dificuldades políticas na homologação desse meio de transporte para seres humanos.

A rede de fibra ótica da SBC cobre todas as capitais do país. A rede é composta por *ramos* de fibra ótica e *concentradores*. Há um concentrador em cada capital, e um ramo de fibra ótica conecta diretamente um par de concentradores. Nem todo concentrador está conectado diretamente por um ramo de fibra a todos os outros concentradores, mas a rede é *conexa*. Ou seja, a partir de um dado concentrador existe uma seqüência de ramos e concentradores que permite que uma informação gerada em qualquer um dos concentradores pode ser enviada a qualquer outro concentrador da rede.

Para comunicação de dados, é normal que um ramo de fibra ótica possa ser utilizado para enviar mensagens nos dois sentidos. A tecnologia desenvolvida, no entanto, tem uma peculiaridade: depois que um ramo de fibra ótica é utilizado para transportar matéria em uma direção, a fibra ótica guarda uma *memória* desse fato, e a partir de então esse ramo somente pode ser utilizado para transportar matéria naquela direção. Concentradores não são afetados por essa memória de direção.

O grupo de pesquisa da SBC é muito bom em física, mas muito fraco em computação. Por isso, você foi contratado para determinar se a rede de fibra ótica existente poderá ser utilizada pela SBC para transportar carga entre qualquer par de capitais, mesmo considerando a restrição de memória de sentido dos ramos de fibra ótica.

Entrada

A primeira linha de cada caso de teste contém dois inteiros N e M separados por um espaço em branco, que representam, respectivamente, a quantidade de capitais ($2 \leq N \leq 1.000$) e a quantidade de ramos de fibra ótica existentes ($1 \leq M \leq 50.000$). As capitais são numeradas de 1 a N. Cada uma das M linhas seguintes de um caso de teste contém dois inteiros A e B ($1 \leq A, B \leq N$, $A \neq B$) separados por um espaço em branco, indicando que existe um ramo de fibra ligando a capital A à capital B. Note que para comunicação de dados o ramo descrito pode ser utilizado para enviar mensagens tanto de A para B quanto de B para A, mas para transferência de matéria ele poderá ser utilizado em apenas uma direção. Há no máximo um único ramo de fibra ligando um par de capitais. O final da entrada é indicado por $N = M = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve imprimir uma única linha, contendo a letra '`S`' caso seja possível utilizar a rede existente conforme especificado, ou a letra '`N`' caso contrário.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Exemplo de saída**
4 3 1 2 2 3 3 4 5 6 1 2 1 3 2 3 2 4 4 5 5 3 6 6 1 2 2 3 3 4 4 5 5 6 1 3 0 0	N S N

Problema D
Mário

Nome do arquivo fonte: `mario.c`, `mario.cpp`, `mario.java` *ou* `mario.pas`

Mário é dono de uma empresa de guarda-volumes, a Armários a Custos Moderados (ACM). Mário conquistou sua clientela graças à rapidez no processo de armazenar os volumes. Para isso, ele tem duas técnicas:

- Todos os armários estão dispostos numa fila e são numerados com inteiros positivos a partir de 1. Isso permite a Mário economizar tempo na hora de procurar um armário;
- Todos os armários têm rodinhas, o que lhe dá grande flexibilidade na hora de rearranjar seus armários (naturalmente, quando Mário troca dois armários de posição, ele também troca suas numerações, para que eles continuem numerados seqüencialmente a partir de 1).

Para alugar armários para um novo cliente, Mário gosta de utilizar armários contíguos, pois no início da locação um novo cliente em geral faz muitas requisições para acessar o conteúdo armazenado, e o fato de os armários estarem contíguos facilita o acesso para o cliente e para Mário.

Desde que Mário tenha armários livres em quantidade suficiente, ele sempre pode conseguir isso. Por exemplo, se a requisição de um novo cliente necessita de quatro armários, mas apenas os armários de número 1, 3, 5, 6, 8 estiverem disponíveis, Mário pode trocar os armários 5 e 2 e os armários 6 e 4 de posição: assim, ele pode alugar o intervalo de armários de 1 até 4.

No entanto, para minimizar o tempo de atendimento a um novo cliente, Mário quer fazer o menor número de trocas possível para armazenar cada volume. No exemplo acima, ele poderia simplesmente trocar os armários 1 e 4 de posição, e alugar o intervalo de 3 até 6.

Mário está muito ocupado com seus clientes e pediu que você fizesse um programa para determinar o número mínimo de trocas necessário para satisfazer o pedido de locação de um novo cliente.

Entrada

A entrada contém vários casos de teste. A primeira linha de cada caso de teste contém dois números inteiros N e L ($1 \leq N \leq L \leq 100.000$), indicando quantos armários são necessários para acomodar o pedido de locação do novo cliente e quantos armários estão disponíveis, respectivamente. A linha seguinte contém L números inteiros positivos separados por espaços em branco, nenhum deles maior do que 1.000.000.000, indicando as posições dos armários disponíveis. Os números dos armários livres são dados em ordem crescente.

O final da entrada é indicado por um caso onde $N = L = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste, imprima uma linha contendo um único número inteiro, indicando o número mínimo de trocas que Mário precisa efetuar para satifazer o pedido do novo cliente (ou seja, ter N armários consecutivos livres).

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
5 6 1 3 4 5 6 8 5 5 1 3 5 6 8 5 6 1 4 5 6 7 8 0 0	1 2 0

Problema E
Histórico de Comandos

Nome do arquivo fonte: `hist.c`, `hist.cpp`, `hist.java` *ou* `hist.pas`

Uma *interface por linha de comando* (ILC) é um dos tipos de interface humano-computador mais antigos que existem. Uma ILC permite a interação com o software através de um *interpretador de comandos*, sendo normalmente acessível em um terminal (ou janela) de texto. A vantagem de um interpretador de comandos é que ele permite que o usuário opere o sistema usando apenas o teclado. Ainda hoje em dia, em que estamos acostumados com interfaces gráficas sofisticadas, muitos aplicativos e sistemas operacionais incluem algum tipo de interface por linha de comando, e muitos usuários ainda preferem usá-la para grande parte das tarefas.

Um dos recursos mais úteis de um interpretador de comandos é o *histórico* de comandos. Quando um comando é digitado e executado, ele é colocado no histórico de comandos do terminal. O comando pode ser exibido novamente no terminal apertando a tecla '↑'; a tecla *Enter* executa o comando novamente quando o comando está sendo exibido no terminal. Todos os comandos executados são guardados no histórico: pressionar a tecla '↑' duas vezes exibe o penúltimo comando executado, pressioná-la três vezes exibe o antepenúltimo comando, e assim sucessivamente.

Por exemplo, se o histórico inicial é (A, B, C, D), para repetir o comando C basta pressionar duas vezes a tecla '↑'. O histórico será então atualizado para (A, B, C, D, C). Nesse ponto, para repetir o comando A será necessário pressionar cinco vezes a tecla '↑'; o histórico será atualizado para (A, B, C, D, C, A). Nesse ponto, para repetir mais uma vez o comando A basta pressionar uma vez a tecla '↑'; o histórico será atualizado para (A, B, C, D, C, A, A).

Leandro é administrador de sistemas e usa freqüentemente o interpretador de comandos para gerenciar remotamente os servidores que administra. Em geral, ele precisa apenas repetir comandos que já havia digitado antes. Enquanto estava trabalhando em um servidor, ele teve uma curiosidade: quantas vezes ele precisa pressionar a tecla '↑' para executar uma determinada seqüência de comandos? Ele sabe quais são as posições no histórico dos comandos que ele necessita executar, mas não sabe resolver esse problema. Por isso, pediu que você fizesse um programa que respondesse à pergunta dele.

Entrada

A entrada é composta de vários casos de teste. A primeira linha de cada caso de teste contém um número inteiro N, indicando o número de comandos que Leandro deseja executar ($1 \leq N \leq 1.000$). A segunda linha de um caso de teste contém N inteiros $P_1, P_2, \ldots, P_N$, que indicam as posições dos comandos no histórico ($1 \leq P_i \leq 1.000.000$) no momento inicial, na ordem em que os comandos devem ser executados. Ou seja, o primeiro comando que deve ser executado está inicialmente na posição P_1 do histórico; depois deve ser executado o comando que está inicialmente na posição P_2 no histórico, e assim por diante, até P_N, que é a posição inicial do último comando que deve ser executado. Note que pode haver $P_i = P_j$.

As posições são dadas em função do número de vezes que a tecla '↑' deve ser pressionada: um comando na posição 5 necessita que a tecla '↑' seja pressionada cinco vezes antes de aparecer no terminal (note que à medida que comandos vão sendo executados, a posição de um dado comando no histórico pode mudar).

O final da entrada é indicado por $N = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste, seu programa deve imprimir apenas uma linha, contendo o número de vezes que Leandro precisa pressionar a tecla '↑' para executar todos os comandos.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
3 2 5 3 4 2 1 4 3 5 1 2 3 4 5 4 1 3 1 3 0	13 16 25 9

Problema F
Desempilhando Caixas

Nome do arquivo fonte: `caixas.c`, `caixas.cpp`, `caixas.java` *ou* `caixas.pas`

Joãozinho e sua família acabaram de se mudar. Antes da mudança, ele colocou todos os seus livros dentro de várias caixas numeradas. Para facilitar a retirada dos livros, ele fez um inventário, indicando em qual caixa cada livro foi colocado, e o guardou na caixa de número 1.

Chegando no seu novo quarto, ele viu que seus pais guardaram as caixas em várias pilhas, arrumadas em fila, com cada pilha encostada na pilha seguinte. Joãozinho é um garoto muito sistemático; por isso, antes de abrir qualquer outra caixa, ele quer recuperar seu inventário.

Joãozinho também é um garoto muito desajeitado; para tirar uma caixa de uma pilha, ele precisa que a caixa esteja no topo da pilha e que ao menos um de seus lados, não importa qual, esteja livre (isto é, não tenham nenhuma caixa adjacente).

Para isso, Joãozinho precisa desempilhar algumas das caixas. Como o quarto dele é bem grande, ele sempre tem espaço para colocar as caixas retiradas em outro lugar, sem mexer nas pilhas que os pais dele montaram.

Para minimizar seu esforço, Joãozinho quer que você escreva um programa que, dadas as posições das caixas nas pilhas, determine quantas caixas Joãozinho precisa desempilhar para recuperar seu inventário.

Entrada

A entrada é composta de vários casos de teste. A primeira linha de cada caso de teste contém dois números inteiros N e P, indicando, respectivamente, o número de caixas e o número de pilhas ($1 \leq P \leq N \leq 1.000$). As caixas são numeradas seqüencialmente de 1 a N.

Cada uma das P linhas seguintes descreve uma pilha. Cada linha contém um inteiro Q_i, indicando quantas caixas há na pilha i, seguido de um espaço em branco, seguido de uma lista de Q_i números, que são os identificadores das caixas. Os elementos da lista são separados por um espaço em branco.

Todas as pilhas contêm pelo menos uma caixa, e todas as caixas aparecem exatamente uma vez na entrada. As caixas em cada pilha são listadas em ordem, da base até o topo da pilha. Todas as caixas têm o mesmo formato.

O final da entrada é indicado por $N = P = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada, seu programa deve imprimir uma única linha, contendo um único inteiro: o número mínimo de caixas, além da caixa 1, que Joãozinho precisa desempilhar para recuperar o seu inventário.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
4 3 1 3 2 1 2 1 4 4 3 1 3 2 2 1 1 4 0 0	2 0

Problema G
Onde Estão as Bolhas?

Nome do arquivo fonte: `bolhas.c`, `bolhas.cpp`, `bolhas.java` *ou* `bolhas.pas`

Uma das operações mais freqüentes em computação é ordenar uma seqüência de objetos. Portanto, não é surpreendente que essa operação seja também uma das mais estudadas.

Um algoritmo bem simples para ordenação é chamado *Bubblesort*. Ele consiste de vários turnos. A cada turno o algoritmo simplesmente itera sobre a seqüência trocando de posição dois elementos consecutivos se eles estiverem fora de ordem. O algoritmo termina quando nenhum elemento trocou de posição em um turno.

O nome *Bubblesort* (ordenação das bolhas) deriva do fato de que elementos menores ("mais leves") movem-se na direção de suas posições finais na seqüência ordenada (movem-se na direção do início da seqüência) durante os turnos, como bolhas na água. A figura abaixo mostra uma implementação do algoritmo em pseudo-código:

Para i **variando de** 1 **a** N **faça**
  Para j **variando de** $N-1$ **a** i **faça**
    Se $seq[j-1] > seq[j]$ **então**
      Intercambie os elementos $seq[j-1]$ e $seq[j]$
    Fim-Se
  Fim-Para
  Se nenhum elemento trocou de lugar **então**
    Final do algoritmo
  Fim-Se
Fim-Para

Por exemplo, ao ordenar a seqüência [5, 4, 3, 2, 1] usando o algoritmo acima, quatro turnos são necessários. No primeiro turno ocorrem quatro intercâmbios: 1×2, 1×3, 1×4 e 1×5; no segundo turno ocorrem três intercâmbios: 2×3, 2×4 e 2×5; no terceiro turno ocorrem dois intercâmbios: 3×4 e 3×5; no quarto turno ocorre um intercâmbio: 4×5; no quinto turno nenhum intercâmbio ocorre e o algoritmo termina.

Embora simples de entender, provar correto e implementar, o algoritmo *bubblesort* é muito ineficiente: o número de comparações entre elementos durante sua execução é, em média, diretamente proporcional a N^2, onde N é o número de elementos na seqüência.

Você foi requisitado para fazer uma "engenharia reversa" no bubblesort, ou seja, dados o comprimento da seqüência, o número de turnos necessários para a ordenação e o número de intercâmbios ocorridos em cada turno, seu programa deve descobrir uma possível seqüência que, quando ordenada, produza exatamente o mesmo número de intercâmbios nos turnos.

Entrada

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém dois inteiros N e M que indicam respectivamente o número de elementos ($1 \leq N \leq 100.000$) na seqüência que está sendo ordenada, e o número de turnos ($0 \leq M \leq 100.000$) necessários para ordenar a seqüência usando *bubblesort*. A segunda linha de um caso de teste contém M inteiros X_i, indicando o número de intercâmbios em cada turno i ($1 \leq X_i \leq N-1$, para $1 \leq i \leq M$). O final da entrada é indicado por $N = M = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma linha na saída, contendo uma permutação dos números $\{1, 2, \ldots, N\}$, que quando ordenada usando *bubblesort* produz o mesmo número de intercâmbios no mesmo número de turnos especificados na entrada. Ao imprimir a permutação, deixe um espaço em branco entre dois elementos consecutivos. Se mais de uma permutação existir, imprima a maior na ordem lexicográfica padrão para seqüências de números (a ordem lexicográfica da permutação $a_1, a_2, \ldots a_N$ é maior do que a da permutação $b_1, b_2, \ldots b_N$ se para algum $1 \leq i \leq N$ temos $a_i > b_i$ e o prefixo $a_1, a_2, \ldots a_{i-1}$ é iqual ao prefixo $b_1, b_2, \ldots b_{i-1}$) .

Em outras palavras, caso exista mais de uma solução, imprima aquela onde o primeiro elemento da permutação é o maior possível. Caso exista mais de uma solução satisfazendo essa restrição, imprima, dentre estas, aquela onde o segundo elemento é o maior possível. Caso exista mais de uma solução satisfazendo as duas restrições anteriores, imprima, dentre estas, a solução onde o terceiro elemento é o maior possível, e assim sucessivamente.

Para toda entrada haverá pelo menos uma permutação solução.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
3 1 1 5 4 4 3 2 1 6 5 2 2 2 2 1 0 0	2 1 3 5 4 3 2 1 6 5 1 2 3 4

Problema H
Jogo de Varetas

Nome do arquivo fonte: `varetas.c`, `varetas.cpp`, `varetas.java` *ou* `varetas.pas`

Há muitos jogos divertidos que usam pequenas varetas coloridas. A variante usada neste problema envolve a construção de retângulos. O jogo consiste em, dado um conjunto de varetas de comprimentos variados, desenhar retângulos no chão, utilizando as varetas como lados dos retângulos, sendo que cada vareta pode ser utilizada em apenas um retângulo, e cada lado de um retângulo é formado por uma única vareta. Nesse jogo, duas crianças recebem dois conjuntos iguais de varetas. Ganha o jogo a criança que desenhar o maior número de retângulos com o conjunto de varetas.

Dado um conjunto de varetas de comprimentos inteiros, você deve escrever um programa para determinar o maior número de retângulos que é possível desenhar.

Entrada

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém um inteiro N que indica o número de diferentes comprimentos de varetas ($1 \leq N \leq 1.000$) no conjunto. Cada uma das N linhas seguintes contém dois números inteiros C_i e V_i, representando respectivamente um comprimento ($1 \leq C_i \leq 10.000$) e o número de varetas com esse comprimento ($1 \leq V_i \leq 1.000$). Cada comprimento de vareta aparece no máximo uma vez em um conjunto de teste (ou seja, os valores C_i são distintos). O final da entrada é indicado por $N = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma única linha na saída, contendo um número inteiro, indicando o número máximo de retângulos que podem ser formados com o conjunto de varetas dado.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
1 10 7 4 50 2 40 2 30 4 60 4 5 15 3 6 3 12 3 70 5 71 1 0	1 3 2

Problema I
Zak Galou

Nome do arquivo fonte: `zak.c`, `zak.cpp`, `zak.java` *ou* `zak.pas`

Zak Galou é um famoso bruxo matador de monstros. Diz a lenda que existe uma caverna escondida nos confins da selva contendo um tesouro milenar. Até hoje nenhum aventureiro conseguiu recuperar o tesouro, pois ele é bem guardado por terríveis monstros. Mas Zak Galou não é um aventureiro qualquer e decidiu preparar-se para recuperar o tão sonhado tesouro.

Zak Galou dispõe de uma certa quantidade de *mana* (uma espécie de energia mágica) e de uma lista de M magias. Cada monstro tem um determinado número de *pontos de vida*. Cada vez que Zak Galou lança uma magia contra um monstro, Zak gasta uma certa quantidade de mana (o *custo* da magia) e inflinge um certo *dano* ao monstro. O dano inflingido provoca a perda de pontos de vida do monstro (o número de pontos perdidos depende da magia). Um monstro está morto se tiver zero ou menos pontos de vida. Zak sempre luta contra um monstro a cada vez. Como é um bruxo poderoso, ele pode usar a mesma magia várias vezes, desde que possua a quantidade necessária de mana.

Em suas pesquisas, Zak Galou conseguiu o mapa do tesouro. A caverna é representada como um conjunto de salões conectados por galerias. Os salões são identificados sequencialmente de 1 a N. Zak sempre inicia no salão 1 e o tesouro está sempre no salão N. Existem K monstros identificados seqüencialmente de 1 a K. Cada monstro vive em um salão, do qual não sai (note que é possível que mais de um monstro viva no mesmo salão). Durante a busca pelo tesouro, Zak Galou pode sair ou recuperar o tesouro de um salão somente se o salão estiver vazio (sem monstro). Em outras palavras, Zak deve sempre, antes de sair ou de recuperar o tesouro de um salão, matar o(s) monstro(s) que lá viver(em).

Dadas as descrições das magias, dos monstros e da caverna, sua tarefa é determinar a quantidade mínima inicial de mana necessária para que Zak Galou consiga recuperar o tesouro.

Entrada

A entrada contém vários casos de teste. A primeira linha de cada caso de teste contém quatro inteiros M, N, G e K, indicando respectivamente o número de magias ($1 \leq M \leq 1.000$), de salões ($1 \leq N \leq 1.000$), de galerias ($0 \leq G \leq 1.000.000$) e de monstros ($0 \leq K \leq 1.000$).

Cada uma das M linhas seguintes descreve uma magia. A descrição de uma magia contém dois números inteiros, a quantidade de mana consumida (entre 1 e 1.000) e o número de pontos de danos provocados (também entre 1 e 1.000).

Em seguida, há G linhas, cada uma descrevendo uma galeria. Uma galeria é descrita por dois números inteiros A e B ($A \neq B$), representando os salões que a galeria conecta. Zak pode utilizar a galeria nos dois sentidos, ou seja, para ir de A para B ou de B para A.

Finalmente, as últimas K linhas de um caso de teste descrevem os monstros. A descrição de um monstro contém dois números inteiros representando respectivamente o salão no qual ele vive (entre 1 e N inclusive) e o seu número inicial de pontos de vida (entre 1 e 1.000 inclusive).

O final da entrada é indicado por $M = N = G = K = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma linha na saída contendo um número inteiro, a quantidade mínima inicial de mana necessária. Caso não seja possível

recuperar o tesouro, você deve imprimir -1.

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
3 4 4 2 7 10 13 20 25 50 1 2 2 4 1 3 3 4 2 125 3 160 3 4 4 1 7 10 13 20 25 50 1 2 2 4 1 3 3 4 2 125 1 3 1 1 1000 1000 1 2 3 1000 0 0 0 0	70 0 -1

Problema J
Olimpíadas

Nome do arquivo fonte: `olimp.c`, `olimp.cpp`, `olimp.java` *ou* `olimp.pas`

Tumbólia é um pequeno país ao leste da América do Sul (ou ao sul da América do Leste) que irá participar dos Jogos Olímpicos pela primeira vez na sua história. Apesar de sua delegação ser muito pequena comparada ao total de atletas que estarão em Pequim (as estimativas oficiais são de mais de dez mil atletas), a participação será fundamental para a imagem e para o turismo de Tumbólia.

Após selecionar os atletas, o Comitê Olímpico Tumboliano (COT) precisa comprar as passagens para eles. A fim de economizar dinheiro, o COT decidiu comprar apenas passagens da Air Rock. No entanto, muitas das passagens da Air Rock já foram vendidas, uma vez que muitos tumbolianos desejam assistir aos Jogos. Sendo assim, o COT deverá comprar passagens de acordo com os assentos vagos em cada vôo.

Todos os vôos da Air Rock partem diariamente antes do meio-dia e chegam após o meio-dia; por isso, um atleta pode tomar apenas um avião por dia. A Air Rock providenciou uma lista contendo todos os vôos operados por ela e o número de assentos vagos em cada um (curiosamente, o número de assentos livres em um mesmo trecho é igual todos os dias).

O COT verificou que realmente é possível ir de Tumbólia para Pequim usando apenas vôos da Air Rock mas, mesmo assim, o COT está tendo dificuldades para planejar a viagem de seus atletas. Por isso, o COT pediu para você escrever um programa que, dada a lista de vôos da Air Rock, determina a menor quantidade de dias necessária para que todos os atletas cheguem em Pequim.

Entrada

A entrada contém vários casos de teste. A primeira linha de cada caso de teste contém três inteiros N, M e A indicando respectivamente a quantidade de aeroportos em que a Air Rock opera ($2 \leq N \leq 50$), a quantidade de vôos em que há assentos vagos ($1 \leq M \leq 2.450$) e quantos atletas a delegação tumboliana tem ($1 \leq A \leq 50$).

Cada uma das M linhas seguintes contém uma descrição de vôo com três inteiros O, D e S que indicam respectivamente o aeroporto de origem ($1 \leq O \leq N$), o aeroporto de destino ($1 \leq D \leq N$ e $O \neq D$) e a quantidade de assentos vagos naquele vôo ($1 \leq S \leq 50$). Os aeroportos são numerados de 1 a N; o Aeroporto Internacional de Tumbólia é o aeroporto 1, e o Aeroporto Internacional de Pequim é o aeroporto N.

A existência de um vôo de A para B **não** implica a existência de um vôo de B para A (mas sempre há no máximo um vôo de um aeroporto para outro em cada direção).

O final da entrada é indicado por $N = M = A = 0$.

A entrada deve ser lida da entrada padrão.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma linha na saída contendo um inteiro, indicando a quantidade mínima de dias necessária para que todos os atletas tumbolianos cheguem em Pequim (alguns atletas podem chegar depois de outros, e eles não precisam chegar na mesma ordem em que partiram).

A saída deve ser escrita na saída padrão.

Exemplo de entrada	**Saída para o exemplo de entrada**
3 3 3 1 2 2 2 3 2 1 3 1 3 3 5 1 2 1 2 3 5 3 1 4 4 4 4 1 4 1 1 2 1 2 3 1 3 4 1 0 0 0	2 6 3