OBI2011

Caderno de Tarefas

Modalidade **Programação** • Nível **2**, Fase **2**

30 de abril de 2011

A PROVA TEM DURAÇÃO DE **5** HORAS

Promoção:

Sociedade Brasileira de Computação

Patrocínio:

Fundação Carlos Chagas

Instruções

LEIA ATENTAMENTE ESTAS INSTRUÇÕES ANTES DE INICIAR A PROVA

- Este caderno de tarefas é composto por 12 páginas (não contando a folha de rosto), numeradas de 1 a 12. Verifique se o caderno está completo.
- A prova deve ser feita individualmente.
- É proibido consultar a Internet, livros, anotações ou qualquer outro material durante a prova. É permitida a consulta ao *help* do ambiente de programação se este estiver disponível.
- As tarefas têm o mesmo valor na correção.
- A correção é automatizada, portanto siga atentamente as exigências da tarefa quanto ao formato da entrada e saída de seu programa.
- Não implemente nenhum recurso gráfico nas suas soluções (janelas, menus, etc.), nem utilize qualquer rotina para limpar a tela ou posicionar o cursor.
- As tarefas **não** estão ordenadas, neste caderno, por ordem de dificuldade; procure resolver primeiro as questões mais fáceis.
- Preste muita atenção no nome dos arquivos fonte indicados nas tarefas. Soluções na linguagem C devem ser arquivos com sufixo *.c*; soluções na linguagem C++ devem ser arquivos com sufixo *.cc* ou *.cpp*; soluções na linguagem Pascal devem ser arquivos com sufixo *.pas*; soluções na linguagem Java devem ser arquivos com sufixo *.java* e a classe principal deve ter o mesmo nome do arquivo fonte; e soluções na linguagem Python devem ser arquivos com sufixo *.py*. Para problemas diferentes você pode escolher trabalhar com linguagens diferentes, mas apenas uma solução, em uma única linguagem, deve ser submetida para cada problema.
- Ao final da prova, para cada solução que você queira submeter para correção, copie o arquivo fonte para o seu diretório de trabalho ou disquete, conforme especificado pelo seu professor.
- Não utilize arquivos para entrada ou saída. Todos os dados devem ser lidos da entrada padrão (normalmente é o teclado) e escritos na saída padrão (normalmente é a tela). Utilize as funções padrão para entrada e saída de dados:
  - em Pascal: *readln, read, writeln, write*;
  - em C: *scanf, getchar, printf, putchar*;
  - em C++: as mesmas de C ou os objetos *cout* e *cin*.
  - em Java: qualquer classe ou função padrão, como por exemplo *Scanner, BufferedReader, BufferedWriter* e *System.out.println*
  - em Python: *read,readline,readlines,print,write*
- Procure resolver o problema de maneira eficiente. Na correção, eficiência também será levada em conta. As soluções serão testadas com outras entradas além das apresentadas como exemplo nas tarefas.

Quadrado Mágico

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

Arnaldo e Bernardo são dois garotos que compartilham um peculiar gosto por curiosidades matemáticas. Nos últimos tempos, sua principal diversão tem sido investigar propriedades matemágicas de tabuleiros quadrados preenchidos com inteiros. Recentemente, durante uma aula de matemática, os dois desafiaram os outros alunos da classe a criar *quadrados mágicos*, que são quadrados preenchidos com números de 1 a N^2, de tal forma que a soma dos N números em uma linha, coluna ou diagonal principal do quadrado tenham sempre o mesmo valor. A *ordem* de um quadrado mágico é o seu número de linhas, e o *valor* do quadrado mágico é o resultado da soma de uma linha. Um exemplo de quadrado mágico de ordem 3 e valor 15 é mostrado na figura abaixo:

2	7	6
9	5	1
4	3	8

Para surpresa de Arnaldo e Bernardo, os outros alunos criaram um grande número de quadrados, alguns enormes, e alegaram que todos eram quadrados mágicos. Arnaldo e Bernardo agora precisam de sua ajuda, para verificar se os quadrados criados são realmente mágicos.

Você deve escrever um programa que, dado um quadrado, verifique se ele é realmente mágico.

Entrada

A primeira linha da entrada contém um único número inteiro N, indicando a ordem do quadrado (seu número de linhas). As N linhas seguintes descrevem o quadrado. Cada uma dessas linhas contém N números inteiros separados por um espaço em branco.

Saída

Seu programa deve imprimir uma única linha. Caso o quadrado seja mágico, a linha deve conter o valor do quadrado (ou seja, a soma de uma de suas linhas). Caso contrário, a linha deve conter o número `0`.

Restrições

- $3 \leq N \leq 1000$.
- $1 \leq$ valor de cada célula $\leq 10^9$.

Informações sobre a pontuação

- Em um conjunto de casos de teste que totaliza 30 pontos, $N \leq 3$.
- Em um conjunto de casos de teste que totaliza 70 pontos, $N \leq 100$.

Exemplos

Entrada	**Saída**
3 1 1 1 1 1 1 1 1 1	0

Entrada	**Saída**
4 16 3 2 13 5 10 11 8 9 6 7 12 4 15 14 1	34

Entrada	**Saída**
3 4 8 9 11 7 3 6 5 10	0

Expressões

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

Pedrinho e Zezinho estão precisando estudar resolução de expressões matemáticas para uma prova que irão fazer. Para isso, eles querem resolver muitos exercícios antes da prova. Como sabem programar, então decidiram fazer um gerador de expressões matemáticas.

O gerador de expressões que eles criaram funciona em duas fases. Na primeira fase é gerada uma cadeia de caracteres que contém apenas os caracteres '{', '[', '(', '}', ']' e ')'. Na segunda fase, o gerador adiciona os números e operadores na estrutura criada na primeira fase. Uma cadeia de caracteres é dita *bem definida* (ou válida) se atende as seguintes propriedades:

1. Ela é uma cadeia de caracteres vazia (não contém nenhum caractere).
2. Ela é formada por uma cadeia *bem definida* envolvida por parênteses, colchetes ou chaves. Portanto, se a cadeia S é *bem definida*, então as cadeias (S), $[S]$ e $\{S\}$ também são *bem definidas*.
3. Ela é formada pela concatenação de duas cadeias *bem definidas*. Logo, se as cadeias X e Y são *bem definidas*, a cadeia XY é *bem definida*.

Depois que Pedrinho e Zezinho geraram algumas expressões matemáticas, eles perceberam que havia algum erro na primeira fase do gerador. Algumas cadeias não eram *bem definidas*. Eles querem começar a resolver as expressões o mais rápido possível, e sabendo que você é um ótimo programador (e participa da OBI) resolveram pedir que escreva um programa que dadas várias cadeias geradas na primeira fase, determine quais delas são *bem definidas* e quais não são.

Entrada

A entrada é composta por diversas instâncias. A primeira linha da entrada contém um inteiro T indicando o número de instâncias. Em seguida temos T linhas, cada uma com uma cadeia A.

Saída

Para cada instância imprima uma linha contendo a letra `S` se a cadeia é *bem definida*, ou a letra `N` caso contrário.

Restrições

- $1 \leq T \leq 20$.
- a cadeia de caracteres A tem entre 1 e 100000 caracteres.
- a cadeia de caracteres A contém apenas caracteres '{', '[', '(', '}', ']' e ')'.

Exemplos

Entrada	**Saída**
12 () [] {} (] }{ ([{}]) {}()[] ()] {[] ((([{}{}()[]])(){}) ((((((((((){[])}]))))))))	S S S N N S S N N N S N

Escalonamento ótimo

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

O SBC (*System for Batch Computing*) é um sistema operacional voltado para a execução sequencial de tarefas. O operador do sistema cria tarefas e o sistema operacional é responsável por agendar a execução destas tarefas.

Cada tarefa pode depender da conclusão de algumas tarefas para poder começar. Se uma tarefa A depende de uma tarefa B, a tarefa B deve terminar antes que a tarefa A inicie sua execução.

Além disto, cada tarefa possui uma prioridade. É sempre mais vantajoso para o sistema começar executando uma tarefa de mais alta prioridade, depois continuar executando uma tarefa de mais alta prioridade dentre as que sobraram e assim por diante.

Neste problema, será dado um inteiro N, que irá representar o número de tarefas no sistema. As tarefas serão numeradas de 0 até $N-1$. Tarefas com índice menor possuem prioridade maior, de forma que a tarefa 0 é a tarefa de mais alta prioridade, a tarefa 1 é a tarefa com a segunda maior prioridade e assim por diante, até a tarefa $N-1$, que é a tarefa com a menor prioridade. Além disso, serão dadas M relações de dependência entre as tarefas.

Seu objetivo será decidir se é possível executar as tarefas em alguma ordem. Caso seja possível, você deverá produzir uma ordem de execução ótima para as tarefas, isto é, desempate as ordens possíveis pela prioridade da primeira tarefa. Se o empate ainda persistir, desempate pela prioridade da segunda tarefa, e assim por diante.

Entrada

A primeira linha da entrada contém inteiros N e M. As próximas M linhas descrevem, cada uma, uma dependência entre as tarefas da entrada. Cada uma dessas linhas irá conter dois inteiros A e B que indicam que a tarefa B depende da tarefa A, isto é, que a tarefa A deve terminar antes que a tarefa B inicie.

Saída

Se não for possível ordenar as tarefas de forma que as dependências sejam satisfeitas, imprima uma única linha contendo o caracter "`*`". Caso contrário, imprima N linhas contendo cada uma um número inteiro. O inteiro na i-ésima linha deve ser o índice da i-ésima tarefa a ser executada na ordem ótima de execução das tarefas.

Restrições

- $0 \leq N \leq 50000$.
- $0 \leq M \leq 200000$.
- $0 \leq A, B < N$.

Informações sobre a pontuação

- Em um conjunto de casos de teste totalizando 60 pontos, $N \leq 1000$.

Exemplos

Entrada	**Saída**
3 1 2 0	1 2 0

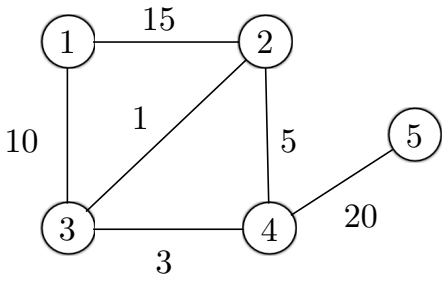
Entrada	**Saída**
2 2 0 1 1 0	*

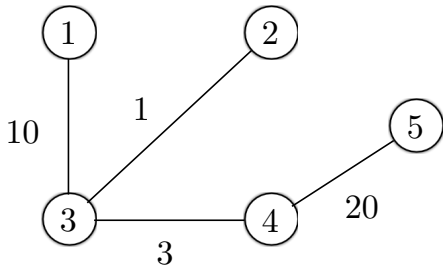
Reduzindo detalhes em um mapa

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

Leonardo Nascimento é um garoto de 13 anos apaixonado por cartografia. Durante as férias de janeiro de 2011, ele alternava seu tempo entre navegar na internet (pesquisando sobre mapas) e arrumar sua coleção de mapas. Navegando na internet, Leonardo descobriu um site especializado em mapas, o Google Maps. Depois de alguns dias usando o site, Leonardo percebeu que quando diminuía o zoom algumas ruas não eram mais exibidas no mapa, isto é, o zoom determinava também o nível de detalhe do mapa. A figura abaixo ilustra um dos testes feito por Leonardo.

Ele sabe que você participa da OBI e que você adora resolver os problemas que envolvem mapas. Então resolveu formular o seguinte problema: dado um mapa de cidades e rodovias que as ligam, selecione um subconjunto das rodovias tal que entre qualquer par de cidades exista uma rota ligando-as e a soma dos comprimentos das rodovias é mínimo. Na figura abaixo e à esquerda temos um exemplo com cinco cidades e seis rodovias ligando-as. A figura abaixo e à direita ilustra uma solução cuja soma dos comprimentos é 34.





Para facilitar um pouco sua vida, Leonardo, determinou que você só precisa dizer a soma dos comprimentos das rodovias do subconjunto selecionado para um dado mapa.

Entrada

A primeira linha da entrada contém dois números N e M que representam o número de cidades e o número de rodovias respectivamente. Cada uma das próximas M linhas é composta por três inteiros U, V e C que indiciam que existe uma rodovia de comprimento C que liga as cidades U e V.

Saída

A saída consiste em apenas uma linha contendo a soma do comprimento das rodovias selecionadas.

Restrições

- $1 \leq N \leq 500$.
- $1 \leq M \leq 124750$.
- $1 \leq U, V \leq N$ e $U \neq V$.
- $1 \leq C \leq 500$.

Exemplos

Entrada	**Saída**
5 6 1 2 15 1 3 10 2 3 1 3 4 3 2 4 5 4 5 20	34

Entrada	**Saída**
4 6 1 2 1 1 3 10 1 4 1 2 3 1 2 4 10 3 4 1	3

Vira!

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

Vira! é um jogo individual que se inicia com N peças igualmente espaçadas em uma linha. Cada peça do Vira! possui dois lados, sendo um branco e um preto; assim, ao virar uma peça, alterna-se a cor que está sendo mostrada entre branco e preto. A figura abaixo ilustra um possível arranjo com 5 peças, duas mostrando o lado branco e duas mostrando o lado preto.

Um movimento consiste em retirar uma peça preta — criando um espaço — e inverter as peças vizinhas à retirada. Sendo assim, dependendo do número de peças vizinhas à retirada, um movimento pode inverter duas, uma, ou mesmo nenhuma peça (se não houver peças vizinhas à que está sendo retirada). Você vence o jogo quando consegue remover todas as peças. A figura abaixo exemplifica uma sequência de movimentos que resolvem uma instância do problema com 5 peças, em que as peças são retiradas na ordem 5-2-1-3-4.

1	2	3	4	5	Descrição do movimento
○	●	○	●	●	Configuração inicial
○	●	○	○		Removemos a peça da posição 5
●		●	○		Removemos a peça da posição 2
		●	○		Removemos a peça da posição 1
			●		Removemos a peça da posição 3
					Removemos a peça da posição 4
					Fim do jogo.

Para uma determinada disposição inicial das peças, podem existir várias soluções diferentes. Por exemplo, poderíamos retirar as peças na ordem 5-2-3-4-1 e ainda assim conseguir retirar todas as peças.

Sua tarefa, neste problema, consiste em contar o número de soluções diferentes para uma dada disposição inicial das peças. Como o número de soluções pode ser muito grande, você deve imprimir apenas o resto do número quando dividido por 10007.

Entrada

A primeira linha da entrada contém o inteiro N. A linha seguinte contém N letras separadas por espaço representando o arranjo inicial das peças. Uma peça branca é indicada pela letra `B` na entrada, e uma peça preta é indicada pela letra `P`.

Saída

Seu programa deve imprimir uma linha contendo o número de soluções distintas que resolvem o jogo.

Restrições

- $1 \leq N \leq 1000$.

Informações sobre a pontuação

- Em um conjunto de casos de teste que totaliza 30 pontos, $N \leq 7$.
- Em um cojunto de casos de teste que totaliza 60 pontos, $N \leq 100$.

Exemplos

Entrada	**Saída**
5 B P B P P	15

Entrada	**Saída**
3 B P B	2