X Maratona de Programação

Universidade de São Paulo

Caderno de Problemas

Departamento de Ciência da Computação
IME-USP

Domingo, 20 de agosto de 2006

Problema A: Brainfuck

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

Linguagens de programação, times de futebol e religião não se discutem. Cada um tem seus favoritos e não admite que o do outro seja melhor (que me perdoem os corinthianos, palmeirenses e são-paulinos). Um grupo de pesquisadores (que não tinha o que fazer) resolveu escrever uma linguagem de programação "ideal" (ideal prá quem, cara pálida?): o Brainfuck.

Brainfuck é uma linguagem de programação cujo funcionamento é muito parecido com uma máquina de Turing. Essa máquina possui como componentes um vetor de 30000 bytes, indexado de 0 a 29999, e um ponteiro, que guarda uma posição desse vetor.

Em cada passo, a máquina realiza uma instrução de acordo com o byte armazenado na posição do vetor indicada pelo ponteiro. Quando esse byte é igual a zero, a execução é terminada.

O conjunto de instruções válidas da linguagem é o seguinte:

Instrução	Descrição
>	Incrementa o ponteiro.
<	Decrementa o ponteiro.
+	Incrementa o byte na posição indicada pelo ponteiro.
−	Decrementa o byte na posição indicada pelo ponteiro.
.	Imprime o valor do byte na posição indicada pelo ponteiro.
,	Lê um byte e armazena na posição indicada pelo ponteiro. Se não houver nada que possa ser lido (entrada acabou), armazenar zero.
[	Início do loop: Executa o código delimitado até que o byte na posição indicada pelo ponteiro seja igual a zero.
]	Fim do loop.
#	Imprime os valores das 10 primeiras posições do vetor.

O ponteiro sempre começa com valor 0, assim como todas as posições do vetor. Na descrição de programas na linguagem brainfuck, caracteres diferentes dos descritos acima são ignorados.

Entrada

A entrada é composta de diversas instâncias. O número de instâncias é dado na primeira linha da entrada. Cada instância começa com uma linha em branco. A próxima linha contém uma cadeia de caracteres não-brancos (ou seja, diferentes de espaço em branco e tabulação), que vai conter a entrada para o programa. Ou

seja, os comandos de leitura são realizados nessa cadeia. Toda a entrada para o programa está contida em uma única linha.

Por fim, a terceira linha contém a descrição do programa. Assim como a segunda linha, esta também não contém caracteres brancos e está inteiramente contida em uma única linha (a separação feita no segundo exemplo de entrada foi feita para evitar o estouro de linha).

Tanto a segunda como a terceira linha têm entre 1 e 100000 caracteres.

Saída

Para cada instância, você deverá imprimir um identificador `Instancia k`, onde h é o número da instância atual. Na linha seguinte você deve imprimir a saída do código fornecido na entrada.

Após cada instância, seu programa deve imprimir uma linha em branco.

Exemplo de entrada

```
2

marrocos
+[>,]<-[+.<-]

nada
++++++++++[>+++++++>++++++++++> +++>+<<<<-]>++.>+.+++++++..+++.>++.<<
+++++++++++++++.>.+++.------.--------.>+.>.
```

Exemplo de saída

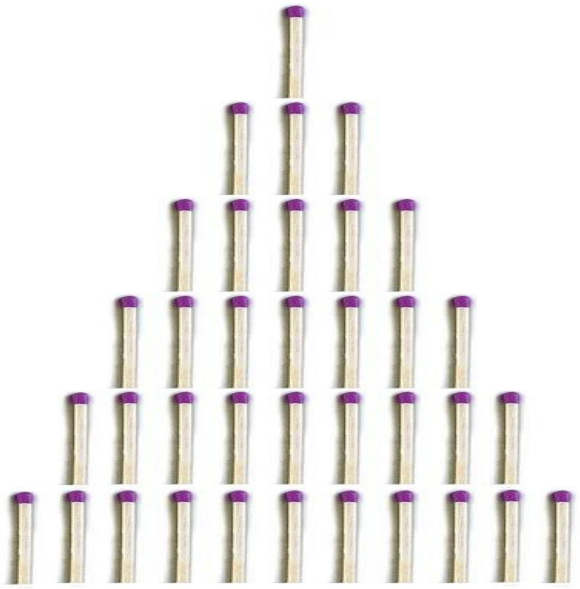
```
Instancia 1
socorram

Instancia 2
Hello World!
```

Problema B: Last Year at Marienbad

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

Durante a Guerra Fria a cidade de Marienbad na República Tcheca ficou imortalizada pelos espiões que usaram seus hotéis luxuosos para troca de informações e até mesmo como um recanto de férias e descanso. Na cidade ficou famoso o jogo de "Streichholzpiramidentfernungspiel", como era conhecido e apreciado pelos espiões das duas Alemanhas.

O jogo começa com 6 fileiras de palitos. A primeira fileira contém 1 palito, a segunda contém 3, a terceira 5, a quarta 7, a quinta 9 e a sexta 11. Segue abaixo um desenho com o esquema do jogo inicial.



Participam do jogo duas pessoas, que alternam seus movimentos. Em cada jogada, uma pessoa deve tirar uma quantidade diferente de zero de palitos do tabuleiro. Todos os palitos retirados em uma jogada devem pertencer à mesma fileira. Assim, se uma fileira contém k palitos e um jogador decide retirar palitos dessa fileira em sua jogada atual, ele tem k opções distintas de jogadas (poderá remover entre 1 e k palitos).

Se após uma jogada o tabuleiro ficar completamente vazio (i.e., sem palitos em qualquer uma das 6 fileiras), o jogador que realizou a última jogada (o jogador que removeu os últimos palitos) perde o jogo.

Dada a descrição de uma configuração do tabuleiro após algumas jogadas, determinar se o jogador que fará a próxima jogada pode vencer o jogo, assumindo

que o adversário é inteligente e portanto sempre escolhe a melhor jogada possível.

Entrada

A entrada começa com um número inteiro n na primeira linha, indicando o número de instâncias do problema que seu programa deve resolver. As próximas n linhas contêm a descrição das instâncias. Cada uma dessas linhas contém uma seqüência de 6 números inteiros. O i-ésimo número da seqüência indica quantos palitos ainda restam na i-ésima fileira de palitos do jogo. Todos os números da seqüência são válidos (ou seja, o i-ésimo inteiro contém um valor entre 0 e o número de palitos com o qual a i-ésima fileira começa o jogo).

Saída

Para cada instância, você deverá imprimir um identificador `Instancia k`, onde k é o número da instância atual. Na linha seguinte, seu programa deve imprimir `sim` se o jogador pode vencer a partida, e `nao` caso contrário. Imprima uma linha em branco após cada instância.

Exemplo de entrada

```
2
1 1 0 0 0 0
1 0 0 0 0 0
```

Exemplo de saída

```
Instancia 1
sim

Instancia 2
nao
```

Problema C: ET phone home

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

Desde o início de 2006 o Seti@home (programa de busca de vida alienígena) tem registrado padrões estranhos em transmissões de rádio recebidas do espaço. Inicialmente imaginou-se tratar apenas de estática. Porém, com o tempo e a repetição das transmissões os pesquisadores foram se convencendo que algo mais havia. Convidados a participar do projeto, lingüistas da Universidade de Baylor identificaram uma linguagem na transmissão. Era uma linguagem bastante simples.

A língua tem várias regras de composição de palavras. As regras de composição serão descritas nesse problema pelos seguintes elementos: um conjunto de símbolos não-terminais V; um conjunto de símbolos terminais T; um símbolo não-terminal especial chamado de raiz; um conjunto de regras de composição de palavras.

Todas as regras de composição que consideramos aqui serão ou da forma `A -> BC` ou da forma `A -> a`, onde A, B, C são elementos de V e a é um elemento de T. A notação acima indica que podemos substituir o não-terminal `A` à esquerda da seta pelo terminal `a` (no primeiro caso) ou pela concatenação dos não-terminais `A` e `B` (no segundo caso) que aparecem à direita da seta.

Aplicando repetidamente as regras de composição sobre o símbolo raiz, podemos montar palavras válidas na língua.

Por exemplo, suponha que o seguinte conjunto de regras de composição é válido:

```
S -> AB
A -> a
B -> b
```

A palavra ab pode ser obtida a partir desse conjunto de regras de composição da seguinte maneira:

`S -> AB`

`AB -> aB`, pois `A -> a`

`aB -> ab`, pois `B -> b`

Já a palavra `b` não pode ser produzida a partir de S a partir desse mesmo conjunto de regras de composição.

Dado um conjunto de regras de composição e uma lista de palavras, sua tarefa é determinar, para cada uma das palavras, se ela pode ou não ser produzida a partir das regras descritas na instância atual.

Entrada

A entrada é composta por vários casos de teste. Cada teste segue as regras descritas acima.

Na primeira linha de cada teste aparece o símbolo raiz, que sempre será uma letra maiúscula. Na segunda linha, o conjunto V será fornecido como uma palavra composta apenas por letras maiúsculas. Cada letra dessa palavra será identificada como um membro de V.

O conjunto T será dado como uma palavra de caracteres imprimíveis (com exceção de # e caracteres em branco) na terceira linha. Cada caractere dessa palavra será identificada como um membro de T.

A seguir, serão fornecidas várias linhas, que descreverão as regras de composição para a instância atual. Uma regra de composição na forma `# -> #` indica o fim da lista de regras de composição.

Por fim, são fornecidas várias linhas, cada uma contendo uma palavra que desejamos saber se pode ou não ser produzida a partir da raiz por meio das regras de composição. Essas palavras não vão conter qualquer caractere em V e são compostas por no máximo 50 caracteres. A lista de palavras termina com uma linha contendo # na primeira coluna.

Saída

No início de cada instância imprima a linha `Instancia k`, onde k é o número da instância atual. Em seguida, para cada palavra x da lista, imprima uma linha na saída dizendo `x e uma palavra valida` se ela pode ser obtida a partir da raiz por meio das regras de composição, e `x nao e uma palavra valida` caso contrário. Imprima uma linha em branco após cada instância.

Exemplo de entrada

```
S
SAB
ab
S -> AB
A -> a
B -> b
# -> #
ab
a
#
S
SAB
ab
S -> AB
A -> a
```

```
B -> b
S -> a
# -> #
ab
a
#
```

Exemplo de saída

```
Instancia 1
ab e uma palavra valida
a nao e uma palavra valida

Instancia 2
ab e uma palavra valida
a e uma palavra valida
```

Problema D: O Cubo

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

Num futuro não muito distante as pessoas buscarão jogos cada vez mais perigosos para se divertir. Depois de ultra-leve e bungee-jump as pessoas precisarão de jogos em que suas atividades mentais sejam também colocadas a prova. É o caso deste jogo, chamado "cube", inventado na Nova Zelândia. Em alguns lugares o jogo também é conhecido pelo seu nome em japonês: sokoban.

Considere um labirinto bi-dimensional composto por células quadradas, onde cada uma delas ou está livre ou está sendo ocupada por uma pedra. A cada passo, você pode sair de uma célula e mover-se para outra célula vizinha (i.e., cima, baixo, direita e esquerda) livre. Você está ocupando uma das células livres desse labirinto.

Uma célula do labirinto contém uma pilha de caixas. A pilha pode ser movida de uma célula i para uma célula k (por exemplo, $k = i + 1$), vizinha de i, na direção ik se você estiver numa célula j (no caso, $j = i - 1$), vizinha de i, e a direção ik é igual à direção ji. Uma caixa não pode ser movida de qualquer outra maneira (ou seja, você não pode puxá-la para trás). Logo, se ela for parar em algum canto do labirinto, você não será capaz de movê-la novamente. Por fim, note que em cada empurrão você dá um passo, e que o contrário não é necessariamente verdade.

Uma das células vazias é marcada como a célula final. Sua tarefa é trazer a caixa para essa célula final através de uma seqüência de passos e empurrões na caixa. Como a caixa é pesada, você quer realizar o menor número possível de empurrões na caixa.

Observe que no jogo de vida real há a possibilidade de você se prender ou mesmo ser esmagado pela caixa, tornando tudo muito mais divertido.

Entrada

O arquivo de entrada é composto por várias instâncias. Cada instância começa com uma linha contendo dois inteiro r e c (ambos menores ou iguais a 20) representando o número de linhas e colunas do labirinto.

Em seguida, são fornecidas r linhas, cada uma contendo c caracteres. Cada caractere descreve uma célula do labirinto. Uma célula ocupada por uma pedra é indicada por `#` e uma célula vazia é representada por um "`.`" (sem as aspas). Sua posição inicial é indicada por `S`, a posição inicial da caixa é indicada por `B` e a posição final da caixa é indicada por `T`.

A entrada termina quando $r = c = 0$.

Saída

Para cada labirinto, inicialmente imprima o número da instância, conforme mostra o exemplo de saída abaixo. Se for impossível levar a caixa até sua posição final, imprima "Impossivel"

Caso contrário, você deve imprimir dois inteiros x e y, onde x indica o número de movimentos (passos + empurrões) e y o número de empurrões de uma seqüência que faz com que você leve a caixa até a posição final. O número de empurrões dever ser minimizado. Caso exista mais de uma seqüência possível que utiliza um número mínimo de empurrões, o número total de movimentos deve ser minimizado. Imprima uma linha em branco após cada instância.

Exemplo de entrada

```
1 7
SB....T
1 7
SB..#.T
7 11
###########
#T##......#
#.#.#..####
#....B....#
#.######..#
#.....S...#
###########
0 0
```

Exemplo de saída

```
Instancia 1
5 5

Instancia 2
Impossivel

Instancia 3
28 6
```

Problema E: Back to the Future

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

Um grupo de amigos resolveu ir a Alemanha para apoiar a seleção brasileira em sua jornada gloriosa rumo ao hexa. Como as passagens aéreas e as estadias eram caras, cada um trouxe uma quantidade de dinheiro que julgou suficiente para passar o mês com conforto e voltar para casa sem problemas.

Porém, após a bela campanha do Brasil na copa do mundo, o grupo de amigos se viu obrigado a gastar o dinheiro que tinha guardado para as etapas finais da copa com a famosa cerveja alemã. As conseqüências de tais atos foram terríveis. Após uma grande bebedeira, todos foram pegos pela polícia local dormindo na rua, e receberam multas pesadíssimas. Além disso, todos perderam suas passagens de volta. Devido a esses contratempos, a viagem de volta ficou ameaçada. De repente, eles descobriram que precisavam voltar para casa gastando a menor quantidade possível de dinheiro.

Analisando as rotas aéreas disponíveis, os amigos notaram que em todas as rotas o número de assentos disponíveis nos aviões era sempre o mesmo. Porém, os preços das viagens entre uma cidade e outra eventualmente variavam bastante. Assustados com a possibilidade de não encontrar lugares suficiente nos aviões para que todos pudessem voltar e preocupados em gastar a menor quantidade possível de dinheiro, o grupo de amigos resolveu pedir sua ajuda.

Entrada

O problema é composto por várias instâncias. Cada instância começa com uma linha com dois inteiros positivos n ($2 \leq n \leq 100$) e m ($1 \leq m \leq 5000$), onde n é o número de cidades que pertencem às m rotas de vôo consideradas. Os amigos querem ir da cidade 1 até a cidade n.

Nas próximas m linhas são fornecidos triplas de inteiros `A B C` descrevendo a rota do avião (A e B) e o preço da passagem aérea por pessoa (`C`). Os valores de A e B estão entre 1 e n. As rotas são bidirecionais (ou seja, há um vôo de `A` até `B` e um vôo de `B` até `A` com preço `C`) e haverá no máximo uma rota entre duas cidades. Na próxima linha são dados dois inteiros, D e K, onde D é o número de amigos e K é o número de assentos livres em cada vôo. Cada rota só pode ser utilizada uma vez.

Saída

Para cada instância, imprima a linha `Instancia k`, onde k é o número da instância atual. Além disso, imprima a menor quantidade possível de dinheiro que os amigos

vão gastar para voltar ao Brasil (que está limitada por 10^{15}). Caso não seja possível escolher um conjunto de vôos que levem todos para casa, imprima `impossivel`. Imprima uma linha em branco após cada instância.

Exemplo de Entrada

```
4 5
1 4 1
1 3 3
3 4 4
1 2 2
2 4 5
20 10
4 4
1 3 3
3 4 4
1 2 2
2 4 5
20 100
4 4
1 3 3
3 4 4
1 2 2
2 4 5
20 1
```

Exemplo de Saída

```
Instancia 1
80

Instancia 2
140

Instancia 3
impossivel
```

Problema F: Final do ICPC

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

Como todos sabemos, ainda não foi decidido o local da próxima final do concurso do ICPC. Desta vez o diretor da competição, Prof. Poucher, tentou escolher uma sede que, de alguma forma, ficasse o mais central possível para os vários participantes.

Para resolver isso, em um grande mapa ele marcou a posição dos participantes prováveis da final. De posse desses dados o Prof. Poucher deseja escolher a sede mais central possível, computando o centro e o raio da menor circunferência que cobre todas as cidades marcadas no mapa (uma cidade está coberta se estiver no interior ou borda desta circunferência).

Entrada

Esse problema é composto por várias instâncias. A primeira linha é composta por um inteiro n, $3 \leq n \leq 100$, e indica o número de cidades. As próximas n linhas contêm a descrição do posicionamento das cidades a partir de suas coordenadas x e y no plano. As coordenadas são números reais. Seu programa deve encerrar a execução quando 0 for o valor de n dado na entrada.

Saída

Para cada instância, imprima uma linha dizendo `Instancia k`, onde k é o número da instância atual. Na segunda linha, imprima a coordenada x e a coordenada y do centro e o raio da circunferência. Após cada instância, seu programa deve imprimir uma linha em branco.

Exemplo de entrada

```
2
0.0 0.0
3 0
5
0 0
0 1
1 0
1 1
2 2
0
```

Exemplo de saída

```
Instancia 1
1.50 0.00 1.50

Instancia 2
1.00 1.00 1.41
```

Problema G: Sudoku

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

O jogo de Sudoku espalhou-se rapidamente por todo o mundo, tornando-se hoje o passatempo mais popular em todo o planeta. Muitas pessoas, entretanto, preenchem a matriz de forma incorreta, desrespeitando as restrições do jogo. Sua tarefa neste problema é escrever um programa que verifica se uma matriz preenchida é ou não uma solução para o problema.

A matriz do jogo é uma matriz de inteiros 9×9. Para ser uma solução do problema, cada linha e coluna deve conter todos os números de 1 a 9. Além disso, se dividirmos a matriz em 9 regiões 3×3, cada uma destas regiões também deve conter os números de 1 a 9. O exemplo abaixo mostra uma matriz que é uma solução do problema.

$$\left(\begin{array}{ccc|ccc|ccc} 1 & 3 & 2 & 5 & 7 & 9 & 4 & 6 & 8 \\ 4 & 9 & 8 & 2 & 6 & 1 & 3 & 7 & 5 \\ 7 & 5 & 6 & 3 & 8 & 4 & 2 & 1 & 9 \\ \hline 6 & 4 & 3 & 1 & 5 & 8 & 7 & 9 & 2 \\ 5 & 2 & 1 & 7 & 9 & 3 & 8 & 4 & 6 \\ 9 & 8 & 7 & 4 & 2 & 6 & 5 & 3 & 1 \\ \hline 2 & 1 & 4 & 9 & 3 & 5 & 6 & 8 & 7 \\ 3 & 6 & 5 & 8 & 1 & 7 & 9 & 2 & 4 \\ 8 & 7 & 9 & 6 & 4 & 2 & 1 & 5 & 3 \end{array}\right)$$

Entrada

São dadas várias instâncias. O primeiro dado é o número $n > 0$ de matrizes na entrada. Nas linhas seguintes são dadas as n matrizes. Cada matriz é dada em 9 linhas, em que cada linha contém 9 números inteiros.

Saída

Para cada instância, imprima uma linha dizendo `Instancia k`, onde k é o número da instância atual. Na segunda linha, seu programa deverá imprimir `SIM` se a matriz for a solução de um problema de Sudoku, e `NAO` caso contrário. Imprima uma linha em branco após cada instância.

Exemplo de entrada

```
2
1 3 2 5 7 9 4 6 8
```

```
4 9 8 2 6 1 3 7 5
7 5 6 3 8 4 2 1 9
6 4 3 1 5 8 7 9 2
5 2 1 7 9 3 8 4 6
9 8 7 4 2 6 5 3 1
2 1 4 9 3 5 6 8 7
3 6 5 8 1 7 9 2 4
8 7 9 6 4 2 1 5 3
1 3 2 5 7 9 4 6 8
4 9 8 2 6 1 3 7 5
7 5 6 3 8 4 2 1 9
6 4 3 1 5 8 7 9 2
5 2 1 7 9 3 8 4 6
9 8 7 4 2 6 5 3 1
2 1 4 9 3 5 6 8 7
3 6 5 8 1 7 9 2 4
8 7 9 6 4 2 1 3 5
```

Exemplo de saída

```
Instancia 1
SIM

Instancia 2
NAO
```

H: Engenharia de Software

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

Wander Vega é um experiente gerente de projetos numa grande empresa de desenvolvimento de sistemas. Ele recentemente leu na renomada revista científica Boas Práticas os resultados de uma pesquisa que indicam que alguns aspectos de metodologias de desenvolvimento ágil podem ser aplicadas em grandes equipes aumentando a produtividade. Ele ficou surpreso ao descobrir que um desses aspectos é a programação pareada (*pair programming*), onde dois desenvolvedores trabalham juntos, usando o mesmo computador. Ávido por impor mudanças que sejam notadas pela diretoria, Wander resolveu adotar programação pareada no próximo grande projeto que irá gerenciar. Só que como todo bom engenheiro de software, Wander quer otimizar esse processo. Ele resolveu que irá usar pares fixos de desenvolvedores. Além disso ele vai alocar os pares de programadores previamente.

Porém, Wander não está disposto a correr riscos desnecessários, e só permitirá a composição de duplas de desenvolvedores que tenham níveis aceitáveis de produtividade, comunicação e capacidade de interação em trabalhos conjuntos. Caso isso não seja possível, Wander colocará todos os desenvolvedores de seu próximo projeto numa sala quente, com várias esfihas, refrigerantes e um computador, e aplicará as técnicas de programação extrema (*extreme programming*) para viabilizar o desenvolvimento do sistema.

Avaliando suas possibilidades ele percebeu que seu plano seria mais reutilizável em outros projetos se ele tivesse um programa que verificasse a viabilidade do pair programming em sua empresa.

Nesse momento ele pensou em você, o mais novo estagiário da empresa, para escrever um programa que resolva esse problema. Wander fez uma profunda análise de requisitos e chegou na seguinte especificação que seu programa deve seguir.

Entrada

A primeira linha da entrada contém um número k, que indica o número de instâncias. Cada instância é composta por uma linha contendo um número inteiro $2 \leq n \leq 100$, a quantidade de profissionais de desenvolvimento da empresa, seguida por n linhas. A i-ésima linha começa com um número p, indicando o número de pessoas com a qual o i-ésimo programador tem produtividade aceitável, e vem seguida por p inteiros, cada um entre 1 e n, indicando tais parceiros. Quando n for 0 seu programa deve parar.

Saída

O programa deve imprimir a cada instância uma linha com `Instancia i`, onde i é o número de i-ésima instância. A linha seguinte deve conter a expressão `pair programming` se a proposta de Wander for viável. Caso contrário, imprima a expressão `extreme programming`. Após cada instância, seu programa deve imprimir uma linha em branco.

Exemplo de Entrada

```
2
6
0
3 3 4 6
1 2
1 2
0
1 2
4
2 3 4
1 4
2 1 4
3 1 2 3
```

Exemplo de Saída

```
Instancia 1
extreme programming

Instancia 2
pair programming
```

I: Lisp é melhor que Java, C e C++

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

Acredite ou não, esse foi o resultado de um estudo conduzido por Ron Garret (Erann Gat) [1] no início do século. A motivação de Garret foi um outro estudo, feito por Lutz Prechelt e publicado na Communications of the ACM, que comparava a performance de tempo de execução e uso de memória de programas escritos em C, C++ e Java. Porém, diferentemente de benchmarcks tradicionais, Prechelt comparou diferentes implementações de uma mesma tarefa feita por 38 desenvolvedores diferentes (em experiência e conhecimento). O estudo de Prechelt mostrou que Java é de 3 a 4 vezes mais lento que C ou C++, porém a variação maior ocorreu entre os programadores, não entre as linguagens, sugerindo que é melhor gastar mais tempo treinando os desenvolvedores do que discutindo que linguagem deve ser escolhida.

Anos depois Garret estendeu esse estudo adicionando Lisp como uma das implementações possíveis para o problema, e dessa vez, além de considerar todos os fatores de comparação de Prechelt, acrescentou o tempo de desenvolvimento como métrica. Os resultados de Garret foram surpreendentes: Lisp ganhou disparado em todos os quesitos, necessitando de menos tempo e linhas de código, consumindo menos memória e executando mais rápido que os programas feitos em C, C++ ou Java. Ficou provado que os programadores de Lisp são muito melhores que os outros programadores. Essa é a sua chance de mostrar que o estudo de Garret está errado. Como? Resolvendo o mesmo problema proposto, em menos tempo e com implementações mais rápidas.

O problema que foi a base de ambos os estudos é o seguinte: Considere o seguinte mapeamento entre letras e dígitos:

```
E | J N Q | R W X | D S Y | F T | A M | C I V | B K U | L O P | G H Z
e | j n q | r w x | d s y | f t | a m | c i v | b k u | l o p | g h z
0 |   1   |   2   |   3   |  4  |  5  |   6   |   7   |   8   |   9
```

Queremos usar esse mapeamento para codificar números de telefone em palavras de forma que seja fácil decorá-los. Sua tarefa é escrever um programa que ache, dado um número de telefone, todas as possíveis codificações do mesmo em palavras. Um número de telefone é uma string arbitrária contendo apenas hífen (-), barras (/), e dígitos. As barras e hífen não devem ser codificados. As palavras são tiradas de um dicionário informado em ordem alfabética. Você deve imprimir

[1] De fato, procurar refências de artigos escritos por Erann/Ron Gat/Garret é algo difícil, pois por algum motivo que foge ao nosso conhecimento, ele mudou de nome e sobrenome, e agora tem publicações com ambos.

apenas as palavras que codifiquem completamente o número de telefone. As palavras no dicionário podem ter letras maiúsculas e minúsculas, hífen (-) e aspas ("), porém você deve usar apenas as letras para codificar um número. A palavra deve ser impressa como foi dada no dicionário. A codificação de um número de telefone pode consistir de uma ou mais palavras, separadas por espaço. A codificação é construída palavra por palavra, da esquerda para a direita. Se, em um dado ponto da codificação nenhuma palavra do dicionário pode ser inserida, então um único digito do número de telefone pode ser usado para a codificação, porém dois números consecutivos não são permitidos numa codificação válida. Em outras palavras: em uma codificação parcial que cobre k dígitos, o dígito $k+1$ é codificado por ele mesmo se e somente se, primeiro, o dígito k não foi codificado por um dígito e, segundo, não existe palavra no dicionário que pode ser usada na codificação começando no dígito $k+1$.

Entrada

Cada instância é composta por uma linha contendo um número inteiro $0 < n \leq 75000$, o número de palavras no dicionário. As próximas n linhas contêm palavras com no máximo 50 caracteres. Depois do dicionário segue um inteiro $1 < t < 100000$, e nas t linhas seguintes os números de telefone a serem codificados. Quando n for 0 seu programa deve parar.

Saída

Para cada instância seu programa deve imprimir uma linha contendo `Instancia k`, onde k é o numero da k-ésima instância. Para cada número de telefone processado seu programa deve imprimir todas as codificações possíveis em ordem lexicográfica (a ordem da tabela ASCII) crescente. Cada codificação deve ser impressa no seguinte formato: o número do telefone seguido de dois pontos (:), um espaço e a codificação.

Exemplo de Entrada

```
23
an
blau
bo"s
Boot
Bo"
da
fern
```

```
fort
Fee
Fest
je
jemand
mir
Mix
Mixer
neu
Name
o"d
Ort
so
Tor
Torf
Wasser
6
5624-82
4824
10/783--5
1078-913-5
381482
04824
0
```

Exemplo de Saída

```
Instancia 1
5624-82: mir Tor
5624-82: Mix Tor
4824: fort
4824: Torf
4824: Tor 4
10/783--5: je bo"s 5
10/783--5: je Bo" da
10/783--5: neu o"d 5
381482: so 1 Tor
04824: 0 fort
04824: 0 Torf
04824: 0 Tor 4
```

Referências

- Lutz Prechelt: *Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences*, Communications of the ACM (Outubro 1999)

- Erann Gat (Ron Garret): Lisp as an alternative to Java. Intelligence 11(4): 21-24, 2000