II Maratona de Programação, UDESC/DCC

Este caderno contém 5 problemas: as páginas são numeradas de 1 a 10
Tempo de duração da prova: 4 horas (240 minutos)

Verifique seu caderno de problemas ANTES do início da competição e solicite a substituição do mesmo caso encontre alguma irregularidade.

Maio, 2006

EM BRANCO

Problema A: Triângulos

Arquivo fonte: `triangulo.c, triangulo.cpp, triangulo.java ou triangulo.pas`

É sempre bom se ter irmãos e irmãs. Você pode pregar peças neles, trancá-los no banheiro ou colocar pimenta no sanduíche deles. Porém, sempre existem os momentos onde eles revidam todas essas maldades.

Imagine que este ano, seus pais lhe informaram que você terá a honra de criar a estrela de prata que será usada na decoração da árvore de Natal na festa da família. O problema é que, quando você pegou o papel prateado com triângulos desenhados (de onde você iria recortar as partes para compor a estrela), percebeu que seus irmãos lhe pregaram uma peça: o papel estava todo perfurado. Sua única chance é montar um algoritmo que seja capaz de analisar o papel e determinar o tamanho do maior triângulo aproveitável.

Dada uma folha de papel em formato triangular com campos brancos (papel) e pretos (buracos), encontrar a maior área de triângulos brancos, conforme apresentado na figura 1.

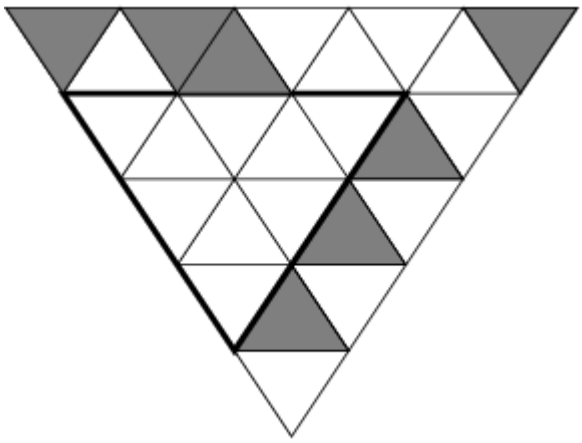

Figura 1: Triângulos

Entrada

O arquivo de entrada contém informação a cerca de diversos pedaços de papel. A primeira linha contém um inteiro n ($1 \leq n \leq 100$), o qual fornece o tamanho do papel (altura do triângulo). As próximas n linhas contém uma sequência dos seguintes caracteres : '#' (buraco), '-' (papel) ou espaços em branco (que servem apenas para fornecer a entrada em formato triangular conforme apresentado no exemplo abaixo que corresponde a figura 1).

Para cada pedaço de papel, o número de '#' e '-', em cada linha, é sempre ímpar e diminui de 2n-1 até 1.

A entrada se encerra por uma linha onde n=0.

Exemplo de Arquivo de Entrada

```
5
#-##----#
 -----#-
  ---#-
   -#-
    -
4
#-#-#--
 #---#
  ##-
   -
0
```

Saída

Para cada folha de papel triangular, primeiro exiba o número do triângulo (ver exemplo abaixo). A seguir, imprima a linha "`The largest triangle area is a.`", onde a é a quantidade de campos dentro do maior triângulo composto apenas por campos brancos. Note que o maior triângulo pode ter seu ápice na parte superior, conforme o segundo exemplo apresentado no arquivo de entrada.

Imprima uma linha em branco entre cada resposta.

Exemplo de Arquivo de Saída

```
Triangle #1
The largest triangle area is 9.

Triangle #2
The largest triangle area is 4.
```

Problema B: Complexidade Automática

Arquivo fonte: `complex.c, complex.cpp, complex.java ou complex.pas`

Analisar a complexidade de execução de algoritmos é uma ferramenta importante para desenvolver programas eficientes que resolvem determinado problema. Um algoritmo que resolva um problema em tempo linear é usualmente muito mais rápido que um em tempo quadrático (para a mesma tarefa), e portanto deve ser preferido.

Geralmente, determina-se o tempo de execução de um programa em relação ao tamanho 'n' da entrada, que pode ser a quantidade de elementos a serem ordenados, o número de pontos em um polígono, etc. Como determinar a formula dependente de n não é trivial, seria ótimo se existisse uma ferramenta automatizada para isso. Infelizmente, isto não é possível fazer isso de forma geral, mas iremos considerar um caso particular para o qual é possível. Nossos programas são construídos de acordo com as seguintes regras (BNF), onde $< number >$ pode ser qualquer valor inteiro não-negativo.

$$
\begin{array}{rcl}
\text{<PROGRAM>} & ::= & \textbf{BEGIN}\text{<STATEMENTLIST>}\textbf{END} \\
\text{<STATEMENTLIST>} & ::= & \text{<STATEMENT>} \mid \text{<STATEMENT><STATEMENTLIST>} \\
\text{<STATEMENT>} & ::= & \text{<LOOP-STATEMENT>} \mid \text{<OP-STATEMENT>} \\
\text{<LOOP-STETEMENT>} & ::= & \text{<LOOP-HEADER><STATEMENTLIST>}\textbf{END} \\
\text{<LOOP-HEADER>} & ::= & \textbf{LOOP} < number > \mid \textbf{LOOP n} \\
\text{<OP-STATEMENT>} & ::= & \textbf{OP} < number >
\end{array}
\tag{1}
$$

O tempo de execução desse programa pode ser computado da seguinte forma: a execução de um "<OP-STATEMENT>" custa $< number >$ unidades de tempo. A lista de comandos definida em "<LOOP-STATEMENT>" é executada tantas vezes quanto for determinado pelo parâmetro associado no cabeçalho (ou uma constante inteira ou n). O tempo de execução de uma lista de comandos é a soma dos tempos de seus componentes. O tempo total geralmente depende de n.

Entrada

O arquivo de entrada inicia com uma linha contendo o número k de programas da entrada. A seguir, são apresentados os k programas (que respeitam a gramática apresentada em 1). Espaços em branco e quebras de linha podem aparecer em qualquer lugar no programa, exceto entre as palavras reservadas **BEGIN, END, LOOP, OP** ou entre digitos de um número inteiro.A profundidade máxima de loops é de 10 iterações.

`Exemplo de Arquivo de Entrada`

```
2
BEGIN
   LOOP n
      OP 4
      LOOP 3
         LOOP n
            OP 1
         END
         OP 2
```

```
      END
      OP 1
   END
   OP 17
END

BEGIN
   OP 1997 LOOP n LOOP n OP 1 END END
END
```

Saída

Para cada programa na entrada, primeiramente imprima o número do programa, conforme apresentado no exemplo de saída abaixo. A seguir, exiba o tempo de execução do programa em função de n; este tempo será uma expressão polinomial de grau ≤ 10. Imprima o polinômio na forma usual, ou seja, agrupando seus termos e apresente-o na forma: "`Runtime = a*n^10+b*n^9+ ... +i*n^2+j*n+k`". Ignore termos cujo coeficiente seja nulo. Fatores iguais a 1 não são escritos. Se o tempo de execução for nulo, apenas imprima "`Runtime = 0`". Imprima uma linha em branco após cada teste.

```
   Exemplo de Arquivo de Saida

Program #1
Runtime = 3*n^2+11*n+17

Program #2
Runtime = n^2+1997
```

Problema C: Caça ao Tesouro

Arquivo fonte: `tesouro.c, tesouro.cpp, tesouro.java ou tesouro.pas`

Encontrar tesouros enterrados é simples: tudo que você precisa é de um mapa! Os piratas do Caribe se tornaram famosos por seus enormes tesouros enterrados e por seus mapas elaborados. Eles se pareciam algo com: "*Comece na base da palmeira solitária. Dê três passos em direção à floresta, então dê 17 passos em direção à caverna ... blá blá blá ... finalmente, seis passos até a pedra gigante. Cave aqui e você achará meu tesouro!*". A maioria dessas direções, recaem sobre as 8 direções principais mostradas em uma bússola (conforme a figura 2(a)).

Obviamente, seguindo os passos fornecidos por estes mapas você seria guiado por um passeio interessante pelo cenário local, mas se você estiver com pressa, existe uma alternativa mais rápida: ande diretamente do ponto de partida para o local onde está enterrado o tesouro. Ao invés de andar três passos para o norte, um passo para o leste, um passo para o norte, cinco passos leste, dois passos sul e dois passos para o oeste (conforme a figura 2(b)), seguindo a rota direta (linha tracejada), resulta em uma trajetória de cerca de 3.6 passos.

Crie um programa que calcule a localização e a distância do tesouro enterrado, dado um mapa "tradicional".

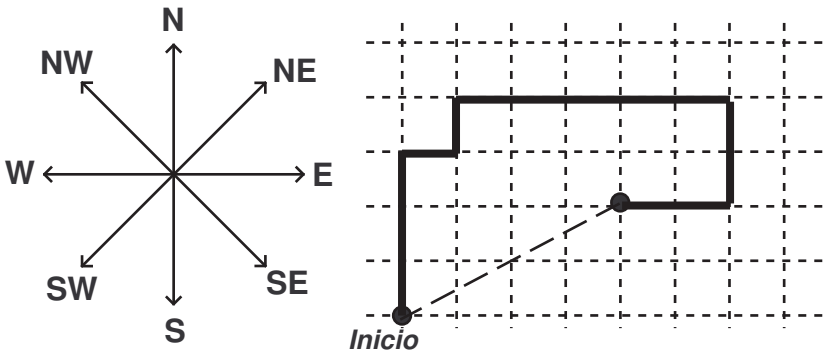

Figura 2: Caça ao Tesouro

Entrada

O arquivo de entrada contém várias strings, cada uma em uma linha separada de (no máximo) 200 caracteres. A última string será a palavra `END`, sinalizando o fim do arquivo de entrada. Todas as outras strings descrevem um mapa do tesouro cada, respeitando o seguinte formato:

A descrição é uma lista de distâncias (que são valores inteiros positivos menores que 1000) e direções (N: norte, S: sul, E: leste, W: oeste, SW: sudoeste, NW: noroeste, SE: sudeste ou NE: nordeste), separadas por vírgula ",". Por exemplo: 3W significa 3 passos na direção oeste e 17NE significa 17 passos na direção nordeste. Um sinal de ponto "." encerra cada descrição a qual não contém espaços em branco.

Exemplo de Arquivo de Entrada

```
3N,1E,1N,3E,2S,1W.
10NW.
END
```

Saída

Para cada descrição do mapa na entrada, primeiramente imprima o número do mapa, conforme mostrado no exemplo de saída abaixo. A seguir, imprima as coordenadas absolutas do tesouro, no seguinte formato: "`The treasure is located at (x,y).`". O sistema de coordenadas é tal que o eixo X aponta para leste e o eixo Y aponta para o norte. O caminho sempre inicia na origem (0,0).

Na próxima linha imprima a distância para esta posição a partir do ponto (0,0), no formato: "`The distance to the treasure is d.`". Valores fracionários para x,y ou d devem ser impressos com três casas decimais obrigatoriamente.

Imprima uma linha em branco entre cada teste.

Exemplo de Arquivo de Saída

```
Map #1
The treasure is located at (3.000,2.000).
The distance to the treasure is 3.606.

Map #2
The treasure is located at (-7.071,7.071).
The distance to the treasure is 10.000.
```

Problema D: Empurra-Caixas

Arquivo fonte: `empurra.c, empurra.cpp, empurra.java ou empurra.pas`

Imagine que você está parado dentro de um labirinto 2D composto por células quadradas as quais podem ou não estar cheias de pedras. Você pode se mover para norte, sul, leste ou oeste, sempre uma célula a cada passo.

Uma das células vazias contém uma caixa que pode ser deslocada (empurrada) para outra célula vazia. A única forma de mover a caixa é estando atrás dela e empurrando-a. Não é possível realizar nenhum outro tipo de movimento com ela, isto significa que se a caixa for empurrada para um canto, você nunca mais conseguirá tirá-la de lá.

No labirinto existe uma célula vazia especial marcada com um "X", esta é a *célula-objetivo*. Seu trabalho é levar a caixa até a célula-objetivo através de um sequência de passos e empurrões. Como a caixa é muito pesada, você quer minimizar seu esforço (número de empurrões). Você pode escrever um programa que determinará a melhor sequência de etapas?

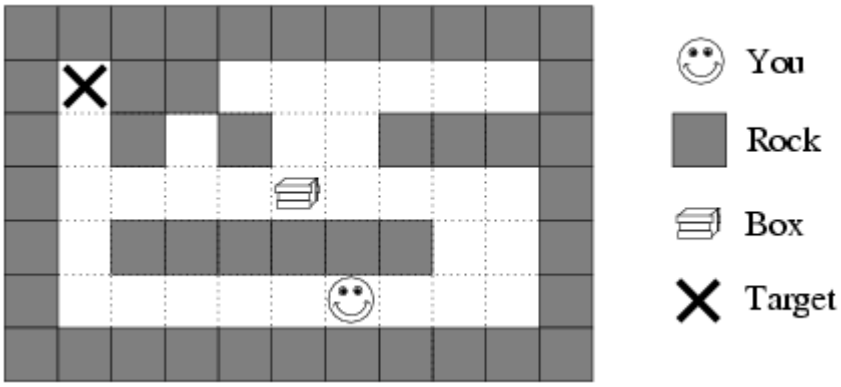

Figura 3: Empurra-Caixas

Entrada

O arquivo de entrada contém a descrição de vários labirintos. Cada labirinto começa com uma linha contendo dois inteiros r e c (ambos ≤ 20) representando o número de linhas e colunas do labirinto.

A seguir, existem r linhas, cada uma com c caracteres. Cada caracter descreve uma célula do labirinto. Uma célula cheia de pedras é indicada por '#' e uma célula vazia é representada por '.'. Sua posição inicial no labirinto é indicada por 'S', a posição inicial da caixa é indicada por 'B' e a célula-objetivo por 'T'.

A entrada termina com dois zeros para r e c.

`Exemplo de Arquivo de Entrada`

```
1 7
SB....T
1 7
SB..#.T
```

```
7 11
###########
#T##......#
#.#.#..####
#....B....#
#.######..#
#.....S...#
###########
8 4
....
.##.
.#..
.#..
.#.B
.##S
....
###T
0 0
```

Saída

Para cada labirinto da entrada, primeiramente imprima o número do labirinto, conforme apresentado no exemplo de saída. Então, se for impossível levar a caixa até a célula-objetivo, imprima "`Impossible.`".

Caso contrário, imprima a sequência de etapas que minimiza o número de empurrões. Se existir mais de uma sequência ótima, escolha aquela que minimiza o número total de movimentos (empurrões e passos). Se ainda sim existir mais de uma solução, qualquer uma é aceitável.

Imprima a sequência de caracteres `N, S, E, W, n, s, e, w` onde as letras maiúsculas significa empurrões e as minúsculas passos nas respectivas direções: N = norte, S = sul, E = leste e W = oeste.

Imprima uma linha em branco após cada teste.

`Exemplo de Arquivo de Saída`

```
Maze #1
EEEEE

Maze #2
Impossible.

Maze #3
eennwwWWWWeeeeeesswwwwwwwnNN

Maze #4
swwwnnnnnneeeesssSSS
```

Problem E: Box of Bricks

Source file: `bricks.c, bricks.cpp, bricks.java or bricks.pas`

Little Bob likes playing with his box of bricks. He puts the bricks one upon another and builds stacks of different height. "*Look, I´ve built a wall!*", he tells his older sister Alice. "*Nah, you should make all stacks the same height. Then you would have a real wall.*", she retorts. After a little consideration, Bob sees that she is right. So he sets out to rearrange the bricks, one by one, such that all stacks are the same height afterwards. But since Bob is lazy, he wants to do this with the minimum number of bricks moved. Can you help?

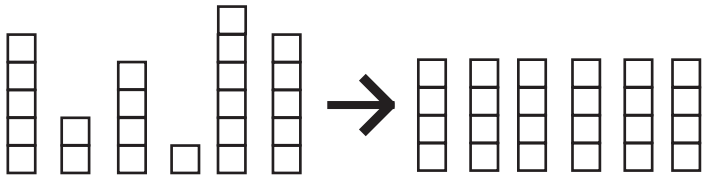

Figura 4: Box of Bricks

Input

The input consists of several data sets. Each set begins with a line containing the number n of stacks Bob has built. The next line contains n numbers, the heights h_i of the n stacks. You may assume $1 \leq n \leq 50$ and $1 \leq h_i \leq 100$.

The total number of bricks will be divisible by the number os stacks. Thus, it is always possible to rearrange the bricks such that all stacks have the same height.

The input is terminated by a set starting with $n = 0$. This set should not be processed.

`Sample Input`

```
6
5 2 4 1 7 5
0
```

Output

For each set, first print the number of set, as shown in the sample output. Then print the line "`The minimum number of moves is k.`", where k is the minimum number of bricks that have to be moved in order to make all the stacks the same height.

Output a blank line after each set.

Sample Output

```
Set #1
The minimum number of moves is 5.
```