Maratona Mineira de Programação 2012

Caderno de Problemas – Aquecimento

26 de Maio de 2012

Realização

Apoio

Instruções:

- Este caderno contém 2 problemas. As páginas estão numeradas de 1 a 7, não contando esta página de rosto. Verifique se o caderno está completo.
- Em todos os problemas, a entrada de seu programa deve ser lida da *entrada padrão*. A saída deve ser escrita na *saída padrão*.

Informações importantes

- Em todos os problemas, a entrada deve ser lida da *entrada padrão* e a saída deve ser escrita na *saída padrão*.
- Há múltiplos casos de teste por problema, e seu programa deve ser capaz de lidar com todos eles *na mesma execução*.
- O enunciado de cada problema especifica como os casos de teste são divididos na entrada.
- Quando uma linha da entrada possui múltiplos valores, esses valores são separados por um (e um único) espaço em branco, exceto se o problema especificar outro separador.
- Todas as linhas da entrada terminam com o indicador de fim de linha (`\n`), inclusive a última.
- Alguns problemas especificam que o fim da entrada é sinalizado por uma linha que contém certos valores especiais que dependem do problema. Essa linha **não** deve ser processada como caso de teste. Nos problemas onde não há um valor especial que sinaliza o fim da entrada, os casos de teste se encerram com o fim do arquivo.
- Quando programando em C ou C++, lembre-se de que a compilação pelos juízes é feita usando-se as opções `-O2 -lm`.

Em seguida, discutimos como a entrada e saída podem ser feitas em cada uma das linguagens de programação permitidas na competição.

C

Em C, as principais funções para a leitura e escrita da entrada e saída padrão são `scanf` e `printf`. Para usar estas funções, você precisa incluir o cabeçalho `<stdio.h>`

Para ler da entrada padrão, você pode usar a `scanf`, como demonstrado no exemplo.

```
int    i;
float  f;
char   s[100];
char   c;

scanf("%i", &i);
scanf("%f", &f, s, c);
scanf("%s", s);
scanf(" %c",&c);
...
```

A string de formato, primeiro parâmetro na chamada da função `scanf` especifica como a entrada será interpretada e armazenada nos demais parâmetros da função. No exemplo, os seguintes modificadores foram usados na string de formato:

- `d` – a entrada é lida até que algo diferente de espaços seja lido e então até o próximo caracter não dígito númerico. Os dados lidos são então interpretados como um número em base 10 e armazenados em `i`.
- `f` – similar a `d`, mas o número é pode ter casas decimais e armazenados em `f`.
- `s` – a entrada é lida até que um espaço ou `'\n'` e armazenada em `s`.
- `c` – um caractere é lido e armazenados em `c`.

`scanf`, contudo, não consegue ler strings *com espaços* (em um única operação). Para fazê-lo, você pode usar a função `gets`.

```
char   s[100];

gets(s);
```

Esta função lê a entrada até o primeiro `'\n'`, inclusive, e coloca-o na variável `s`. A função `scanf`, contudo, não remove o `'\n'` no fim da entrada e, por isso, se você usar `gets` após `scanf`, precisa remover o `'\n'` você mesmo, ou o `gets` lerá uma string vazia. Para fazê-lo, você pode usar a função `getchar` ou, usar abordagem mais usada: `fflush(stdin)`. Esta chamada de `fflush` *joga fora* tudo o que já está disponível na entrada padrão mas que ainda não foi lido pelo seu programa.

Para escrever na saída padrão, você pode usar a função `printf`, como no exemplo.

```
int    i = 10;
float  f = 4.4;
char   s[] = "esta é uma string";
char   c = 'A';

printf("%4i, %6.2f, %s, %c\n", i, f, s, c);
...
```

A string de formato, primeiro parâmetro na chamada da função `printf` especifica como os demais parâmetros serão impressos na saída padrão. No exemplo, os seguintes modificadores foram usados:

- `4d` – `i` é impresso como inteiro na base 10, usando no mínimo quatro espaços.
- `6.2f` – `d` é impresso como ponto fixo base 10, usando no mínimo 6 espaços, com 2 duas casas decimais.
- `s` – `s` é impresso como uma string.
- `c` – `c` é impresso como caractere.

Uma última observação é uso do `'\n'`; ele é necessário para gerar a quebra de linha.

C++

Em C++, é possível usar as mesmas funções utilizadas em C para tratamento de entrada e saída. Há, porém, uma alternativa, que pode ser mais interessante em alguns casos: a biblioteca de *streams* da linguagem.

Essa biblioteca define o stream `std::cin` que representa a entrada padrão e o stream `std::cout` que representa a saída padrão. Para ler, por exemplo, um inteiro, basta direcionar a entrada padrão para a variável:

```
int n;
std::cin >> n; // Lê um inteiro da entrada padrão e o armazena em n
```

Para esta biblioteca, é necessário incluir o cabeçalho `<iostream>`.

Assim como inteiros, pode-se ler qualquer tipo simples (`float`, `long long`, `std::string`, etc.). Para ler múltiplos valores, é possível encadear as chamadas:

```
int a;
long long b;
std::string c;

std::cin >> a >> b >> c; // Lê três valores: um inteiro, um long long e uma string
```

Esses valores não necessariamente precisam estar na mesma linha. O `cin` ignora espaços em branco, tabulações e quebras de linha. Caso você leia uma string, ela será lida apenas até o primeiro espaço em branco (ignorando os iniciais). Por exemplo, se o conteúdo da entrada padrão é:

```
Isso tem espacos.
```

e você lê a entrada usando o código

```
std::string s;
std::cin >> s;
```

então o conteúdo de `s` após a execução será `"Isso"`. Para ler todo o conteúdo da linha atual para uma string, incluindo os espaços, deve-se usar a função getline:

```
std::string s;
std::getline(s, std::cin);
```

A leitura com getline, porém, possui uma peculiaridade: o getline lê do caractere atual até a próxima quebra de linha, qualquer que seja ela. Suponha, por exemplo, que o conteúdo da entrada padrão seja:

```
14
Essa eh a linha que eu quero ler.
```

Suponha que você use o seguinte código para ler essa entrada:

```
int n;
std::string line;
std::cin >> n;
getline(line, std::cin);
```

Ao fim da execução deste código, `n` contém o valor 14, mas `line` contém uma string vazia, e não a segunda linha da entrada. Isso ocorre porque a leitura de `n` é interrompida assim que a primeira quebra de linha, e a posição de leitura fica sendo essa quebra de linha. Como o getline lê apenas até encontrar uma quebra de linha, o primeiro caractere que ele lê é exatamente um `\n` e ele para por ali, retornando uma string vazia. Uma forma de contornar esse problema é forçar a leitura da quebra de linha, para que o getline comece a partir do primeiro caractere da segunda linha. Isso pode ser feito usando a função `std::cin.get()`:

```
int n;
std::string line;
std::cin >> n;
std::cin.get(); // Lê a quebra de linha e avança a posição atual
getline(line, std::cin); // Lê corretamente a segunda linha
```

Já para imprimir a saída, é possível usar o stream `std::cout`. O funcionamento é simples e análogo ao do `cin`:

```
int n = 42;
std::cout << n << '\n'; // Imprime uma linha contendo o valor 42
```

Note que as quebras de linha devem ser incluídas explicitamente. Há duas maneiras de se fazer isso: imprimir o caractere `\n` diretamente (como no exemplo acima) ou usar `std::endl`:

```
int n = 42;
std::cout << n << std::endl; // Imprime uma linha contendo o valor 42
```

A única diferença entre essas duas maneiras é que ao usar `std::endl`, além de imprimir o caractere de quebra de linha na saída, o sistema faz *flush* no buffer de saída, o que pode ser uma operação cara. Em nenhum dos problemas dessa competição esse *flush* é relevante (pois a saída só será lida e julgada após o fim da execução do programa). Assim, o uso de `std::endl` **não** é recomendado: a diferença de performance causada pelo flush pode, em alguns casos, ser grande o suficiente para que uma solução receba o veredito de Tempo Limite Excedido quando a mesma solução usando `'\n'` é aceita.

Assim como `int`, quase qualquer tipo simples pode ser escrito diretamente com `std::cout`. É preciso tomar cuidado, no entanto, com a escrita de números de ponto flutuante. Normalmente, quando um número de ponto flutuante deve ser escrito na saída, o enunciado pede que ele seja escrito com uma certa quantidade de casas decimais. Se você simplesmente imprime o número com `cout` sem tomar nenhuma precaução especial, é extremamente improvável que ele seja escrito com o número correto de casas decimais.

Para forçar que um número de ponto flutuante seja impresso com um certo número de casas decimais, é preciso usar os manipuladores `fixed` e `setprecision`, que estão definidos no cabeçalho `<iomanip>`:

```
double pie = 3.14159265358979;
std::setprecision(4);
std::cout << std::fixed << pie << '\n'; // Imprime "3.1416", com 4 casas decimais
```

Por padrão, a biblioteca `<iostream>` é sincronizada com a biblioteca de C (`<stdio.h>`). Isso faz com que seja possível usar, num mesmo programa, `cin` e `scanf` ou `cout` e `printf`. Porém, essa sincronização não é barata, e faz com que a performance da biblioteca de streams seja ruim. Em alguns problemas, onde entrada e/ou saída são grandes, a diferença de performance pode ser tal que chega a aumentar o tempo de execução em mais de uma ordem de grandeza. Há duas opções para evitar esse problema:

1. Não usar a biblioteca de streams, e usar as funções de C para entrada e saída;
2. Desabilitar a sincronização entre as bibliotecas. Isso pode ser feito adicionando uma linha de código no início da função `main`:

   ```
   std::ios::sync_with_stdio(false);
   ```

   Dado que em vários casos a biblioteca de streams torna o tratamento de entrada e saída muito mais simples, essa é a opção mais recomendada. Note porém que ao usá-la o comportamento das funções da biblioteca `stdio.h` de C passa a ser indefinido e você não deve mais usá-las.

Java

A escrita na saída padrão usando Java é muito similar à do C e C++, com streams correspondentes à entrada (`System.in`) e saída padrão (`System.out`).

A classe `Scanner` provê um *wrapper* para `System.in` com várias funções úteis na leitura da entrada dos problemas. Usando `Scanner`, para efetuar a leitura de `int`, `double` utilize as funções `nextInt` e `nextDouble`. Para ler uma `string`, utilize `next`. Outras funções estão disponíveis para ler `long`, `short`, `BigInteger` e assim por diante. O exemplo a seguir demonstra o uso de algumas destas funções.

```
import java.util.Scanner;
...

Scanner in = new Scanner(System.in);
int    i = in.nextInt();    // lê int
double d = in.nextDouble(); // lê double
String s = in.next();       // lê string até o primeiro espaço ou '\n'
...
```

Observe que não há uma função `getChar` no `Scanner`. Se precisar ler caractere por caractere, então você deverá ler uma `string` e acessar o caractere na `string` usando `charAt`, da seguinte forma:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
String s = in.next();       // lê string até o primeiro espaço ou '\n'
char c = s.charAt(2);       // lê caractere na posição 2 da string.
...
```

Para ler até o fim de uma linha, `Scanner` provê a função `nextLine`, exemplificado a seguir:

```
import java.util.Scanner;
...
      Scanner in = new Scanner(System.in);
      String line = in.nextLine();    // Lê a linha até o '\n'
...
```

Ao final do `nextLine` o `\n` é ignorado e o `Scanner` posicionado no início da próxima linha.

Para escrever na saída padrão, você pode usar as funções `print` e `println` do `System.out`, que lhe permitem imprimir strings com e sem `'\n'` no final. Estas strings podem ser geradas pela concatenação de várias strings e outros objetos, como no exemplo a seguir, ou não. Para imprimir tipos básicos, como é mais comum em maratonas, uma forma mais eficiente é o uso de `printf`, que tem funcionamento semelhante ao do `printf` do C. Veja o exemplo.

```
int    i = 10;
double d = 4.4;
String s = "esta é uma string";
char c = 'A';

System.out.printf("%4d, %6.2f, %s, %c\n", i, d, s, c);
...
```

A string de formato, primeiro parâmetro na chamada da função `printf` especifica como os demais parâmetros serão impressos na saída padrão. No exemplo, os seguintes modificadores foram usados:

- `4d` – `i` é impresso como inteiro na base 10, usando no mínimo quatro espaços.
- `6.2f` – `d` é impresso como ponto fixo base 10, usando no mínimo 6 espaços, com 2 duas casas decimais.
- `s` – `s` é impresso como uma string.
- `c` – `c` é impresso como caractere.

Uma última observação é uso do `'\n'`; ele é necessário para gerar a quebra de linha.

Problema A. Ajudando Og

Nome do arquivo fonte: `og.c`, `og.cpp`, ou `og.java`

Og, o ogro, possui vários filhos. E seus filhos, por sua vez, possuem vários filhos. Og quer saber quantos netos ele tem. Mas ogros, como você sabe, são péssimos em matemática. Portanto, Og quer sua ajuda: dado o número de filhos que cada filho de Og tem, determine o número total de netos de Og.

Entrada

A entrada começa com uma linha contendo um inteiro T ($1 \leq T \leq 20$), que representa o número de casos de teste. Cada caso de teste é descrito em duas linhas. A primeira linha contém um inteiro N ($1 \leq N \leq 1000$), o número de filhos de Og. A segunda linha de cada caso de teste possui N inteiros $f_1, f_2, \ldots, f_N$. O número f_i ($0 \leq f_i \leq 1000$, para todo i entre 1 e N inclusive) representa o número de filhos que o i-ésimo filho de Og possui.

Saída

Para cada caso de teste, imprima uma linha contendo um único inteiro: o número de netos de Og.

Exemplos

Entrada	Saída
2 3 7 5 9 2 0 98	21 98

Problema B. Bernardo e o Código Da Vinci

Nome do arquivo fonte: `davinci.c`, `davinci.cpp`, ou `davinci.java`

Leonardo da Vinci escreveu a maior parte de suas anotações pessoais "de trás para frente", de forma tal que para que alguém pudesse lê-las, era preciso usar um espelho. Assim, era bem difícil que alguém conseguisse decifrá-las.

Bernardo é um estudante de ciência da computação obcecado pelo trabalho de da Vinci, e decidiu criar um programa de computador para decifrar as anotações pessoais dele. Usando imagens escaneadas dos documentos e um sistema de reconhecimento ótico de caracteres, Bernardo conseguiu uma representação em texto dos documentos. Porém, apesar de todas as letras terem sido decifradas, o texto ainda está de trás para frente! Por exemplo, se a frase "Batata frita." está presente no texto, ela estará escrita como ".atirF atataB".

Bernardo quer sua ajuda, e pediu que você crie um programa de computador que lê o texto de trás para frente e gera uma versão "normal" dele.

Entrada

A primeira linha da entrada possui um inteiro N ($1 \leq N \leq 100$), o número de linhas do texto. As próximas N linhas contém o texto, e são compostas apenas de letras maiúsculas e minúsculas, sinais de pontuação, dígitos e espaços em branco. Cada linha possui entre zero e 100 caracteres (inclusive). Todas as linhas, inclusive a última, terminam com o marcador de fim de linha (`\n`).

Saída

Para cada linha de texto da entrada, imprima a versão em ordem normal da linha na saída.

Exemplos

Entrada
6 ,oiraid odireuQ .emof moc iedroca ue ejoH .satatab mare aicnediser ahnim me levitsemoc ed aivah euq asioc acinu A .ahnizoc an oelo aivah euq ietoN :etnahlirb aiedi amu evit ,iaD !atirF atataB
Saída
Querido diario, Hoje eu acordei com fome. A unica coisa que havia de comestivel em minha residencia eram batatas. Notei que havia oleo na cozinha. Dai, tive uma ideia brilhante: Batata Frita!

Note que apesar de cada linha estar em reverso, a ordem das linhas não se altera.