CCM0128 — Computação II

Curso de Ciências Moleculares — Turma 22 — Primeiro Semestre de 2013

Primeiro Exercício-Programa — Data de entrega: até **2 de abril de 2013.**

Reversões de DNA[1]

Este exercício-programa é baseado num problema de Biologia Computacional. Você já deve ter estudado alguma coisa sobre DNA e sabe que uma molécula de DNA pode ser (muito simplificadamente) modelada como uma sequência cujos elementos pertencem ao conjunto $\{A, T, C, G\}$. Assim sendo, consideraremos que um DNA é uma cadeia de caracteres (*string*) na qual podem aparecer somente ocorrências das letras `A`, `T`, `C` e `G`.

O DNA pode sofrer uma série de modificações, algumas das quais podem ser descritas como transformações algorítmicas sobre a cadeia de caracteres. Uma dessas transformações é a reversão (ou inversão). Sua tarefa é escrever um programa C que simule uma sequência de reversões.

Dada uma cadeia de caracteres que representa o DNA inicial (uma *string* formada por ocorrências das letras `A`, `T`, `C` e `G`), uma reversão é uma transformação que corta e extrai um segmento da cadeia de caracteres, inverte a ordem dos caracteres desse segmento e o recoloca de volta na cadeia de caracteres, na mesma posição em que ele estava anteriormente. Para se especificar uma reversão, basta especificar as extremidades (o índice da primeira posição e o da última posição) do trecho a ser invertido.

Seu programa deve ler, da entrada padrão (`stdin`), uma cadeia de caracteres que representa o DNA inicial. Ele deve também ler uma sequência de operações de reversão, aplicar essas operações sobre o DNA, na ordem em que elas foram fornecidas, e imprimir na saída padrão (`stdout`) a cadeia de caracteres resultante.

Formato da entrada

O programa deve ler três linhas da entrada padrão.

- A primeira linha conterá um número m que especifica a quantidade de reversões a serem realizadas.
- A segunda linha conterá a sequência de caracteres correspondente ao DNA inicial. Essa linha pode conter apenas ocorrências das letras `A`, `T`, `C` e `G`. Seja n o número de caracteres nessa linha, excluído o caractere *new line* (`'\n'`) que a termina.
- A terceira linha conterá m pares (a_i, b_i), sendo a_i e b_i números inteiros tais que $1 \leq a_i < bi \leq n$. Cada um desses pares especifica uma reversão. O segmento afetado pela reversão começa no índice a_i (primeira posição do segmento) e vai até o índice b_i (última posição do segmento). Aqui a indexação é a partir de 1, isto é, o índice 1 corresponde ao primeiro caractere e o índice n corresponde ao último caractere da sequência que representa o DNA. (Internamente seu programa pode trabalhar com indexação a partir de 0, mas nos dados de entrada a indexação é a partir de 1.)

Uma entrada que consista de três linhas com o formato descrito acima é uma entrada válida.

[1] Agradeço ao Prof. Arnaldo Mandel, que criou a versão original deste exercício-programa, e assumo toda a responsabilidade pelos erros que posso ter introduzido ao adaptar o enunciado do Arnaldo.

Requisitos

- Seu programa deve aceitar toda entrada válida cujos valores de m e n estejam respectivamente nos intervalos
$$1 \leq m \leq 10.000 \qquad \text{e} \qquad 1 < n \leq 10.000.000.$$
- Para toda entrada válida que satisfaça as condições do item anterior, o programa deve imprimir o resultado correto em menos de 5 segundos.

É o segundo requisito acima que torna este exercício interessante e desafiador.

Exemplo

Esta é uma possível entrada:

```
3
ATATGCGA
3 6 4 8 3 4
```

Note que os três pares de índices na terceira linha não são delimitados por nenhum caractere especial. Os pares não aparecem envolvidos por parênteses, havendo apenas um ou mais espaços entre um índice e outro. (Esse exemplo de entrada é ridiculamente pequeno. Seu programa deve funcionar eficientemente para entradas muito maiores, com até 10.000.000 caracteres na segunda linha e 20.000 números na terceira linha!)

Esta é a saída correspondente à entrada acima:

```
ATACGATG
```

Veja o que aconteceu (em cada passo os segmentos invertidos estão sublinhados):

Sequência inicial:	`ATATGCGA`
Reversão 1:	$(3, 6)$
Sequência 2:	`ATCGTAGA`
Reversão 2:	$(4, 8)$
Sequência 3:	`ATCAGATG`
Reversão 3:	$(3, 4)$
Sequência final:	`ATACGATG`

Observações sobre o requisito de eficiência

O objetivo deste exercício é adquirir alguma prática com o projeto de uma estrutura de dados voltada especificamente para um determinado problema, de modo a reduzir a complexidade computacional temporal da solução e assim chegar a um programa que rode mais rapidamente.

É muito fácil implementar uma solução simples e intuitiva para o problema das reversões de DNA. Essa solução mantém num vetor a sequência de caracteres correspondente ao DNA e efetua as operações de reversão manipulando diretamente os caracteres desse vetor. Tal solução ingênua, entretanto, leva

tempo $O(m \cdot n)$ para efetuar as reversões. (Cada uma das m reversões inverte a ordem dos caracteres de um "subvetor" cujo comprimento é $O(n)$.) Para os valores máximos de m e n que seu programa deve aceitar, a quantidade de atribuições a elementos do vetor efetuadas pela solução ingênua é da ordem de 10^{11}, quantidade essa que deve estourar o limite de 5 segundos.

Em caso de dúvida, faça experimentos com o programa `dna-solucao-vetor`, cujo arquivo-fonte em C está disponível na página da disciplina. Por outro lado, o programa executável `dna-rapido` (para Linux), também disponível na página da disciplina, mostra o que se pode conseguir.

A página da disciplina contém também um ponteiro para uma área com alguns arquivos de entrada e com os arquivos de saída correspondentes. Use esses arquivos para verificar se seu programa está funcionando corretamente e para fazer medições e comparações de tempos de execução. Em dois dos arquivos de entrada os valores de m e n são os máximos que o programa tem de aceitar. (Esses dois arquivos são grandes!) Meça os tempos de execução dos programas `dna-solucao-vetor` e `dna-rapido` para esses arquivos de entrada. Tome como referência o `dna-rapido` e tenha como alvo tempos de execução comparáveis com os desse programa.

Para medir tempos de execução, use (no Linux) o comando `time`:

```
time ./programa < arquivo.in > arquivo.out
```

A verificação do requisito dos 5 segundos será feita com o comando

```
time ./programa < arquivo.in > /dev/null
```

numa máquina específica (a ser definida) da sala pró-aluno do CCM.

Dicas

Como este é um exercício com certo grau de dificuldade, aqui vão algumas dicas:

1. Para se inverter eficientemente um trecho da *string* é preciso representá-la de modo que a pergunta "Qual é o próximo caractere?" seja respondida pela própria representação e não pelo posicionamento dos caracteres. É claro que uma lista encadeada satisfaz essa condição. Só que também é preciso poder se movimentar para trás na *string* de modo eficiente.

2. Uma possibilidade é usar uma lista duplamente encadeada: cada célula da lista contém um ponteiro para a próxima célula e um ponteiro para a célula anterior. A lista duplamente encadeada pode ser percorrida eficientemente nos dois sentidos. Mesmo assim, é difícil arrancar um segmento da lista e recolocá-lo de volta de modo que ele fique com o sentido trocado. Como percorrer a lista modificada? Antes de chegar num "ponto de corte", é preciso ir seguindo o encadeamento dado pelos campos "próxima célula". Ao atingir o primeiro ponto de corte, é preciso começar a seguir o encadeamento dado pelos campos "célula anterior", até que se chegue ao segundo ponto de corte. A coisa fica ainda mais difícil com as reversões sendo aplicadas uma após (e sobre) a outra.

3. Uma possibilidade mais interessante é implementar uma solução verdadeiramente inspirada pelo DNA: junto com a lista (simplesmente encadeada) que avança, manter uma "lista espelhada" que volta, com ligações entre os elementos correspondentes das duas listas.

4. Mesmo usando uma estrutura de dados baseada em lista encadeada, para efetuar uma reversão é preciso conhecer os pontos de corte. Em outras palavras, é preciso obter os endereços das

células que atualmente correspondem aos índices inicial e final da reversão. (É muito fácil obter os pontos de corte percorrendo a lista e contando as células percorridas. Por que essa ideia não é boa?)

5. O problema da obtenção dos pontos de corte pode ser dividido em dois subproblemas:

   (a) Dado o índice atual de uma célula, determinar o índice original dessa célula. Em outras palavras: Depois de uma série de reversões, há uma célula que foi parar na k-ésima posição da cadeia rearranjada. Qual era a posição dessa célula na cadeia inicial?

   (b) Dado o índice original de uma célula, encontrar o endereço da célula. Essa parte é fácil. Basta guardar um mapeamento de índices para células, construído logo no começo.

6. Como o valor máximo de m é bem menor que o de n, uma solução $O(m^2)$ deve ser bem mais rápida que uma solução $O(m \cdot n)$. Na verdade este exercício não admite solução que seja só $O(m^2)$, pois a entrada tem tamanho $O(n + m)$. Qualquer solução levará tempo $O(n + m)$ só para ler a entrada! Mas, se você fizer as reversões em tempo $O(m^2)$, sua solução terá tempo de execução $O(n + m^2)$. Para os valores de m e n que nos interessam, um programa assim deve ser muito mais rápido que o programa ingênuo `dna-solucao-vetor`.

7. Uma solução $O(n+m^2)$ muito provavelmente satisfará o requisito dos 5 segundos para quaisquer valores de m e n nos intervalos especificados. Tal solução deixa de ser boa, entretanto, para valores maiores de m. Se você tiver gás, procure uma solução melhor. Este exercício tem uma solução $O(n + m\sqrt{n})$ baseada numa ideia que é razoavelmente simples, mas de implementação trabalhosa.

   - (Bônus) Descreva uma solução $O(n + m\sqrt{n})$, sem necessariamente implementá-la.
   - (Bônus adicional) Implemente essa solução. Antes que você se decepcione, já vou avisando: para os arquivos de entrada disponíveis na página da disciplina, o programa `dna-rapido` é mais veloz que a minha solução $O(n + m\sqrt{n})$. A vantagem dessa solução só aparece para valores de m maiores que 10.000.

Observações finais

- O programa deve ser compilável com o seguinte comando:

  ```
  gcc -ansi -pedantic -Wall -O2 -U_FORTIFY_SOURCE
  ```

  A compilação deve ocorrer sem erros nem *warnings*.

- O programa não deve usar variáveis globais.

- O uso de funções facilitará muito seu trabalho e fará seu programa ficar bem mais organizado. Recomendo fortemente que, além da função `main`, seu programa tenha pelo menos mais duas funções.

  – É conveniente ter uma função que receba o índice atual de um caractere e devolva o índice original desse caractere. Além de receber o índice atual, essa função precisará receber outras informações. (Quais?) Dependendo de como você implementar o seu programa, pode ser o caso de fazer com que a função devolva o índice original do caractere e mais alguma informação...

– Também é conveniente ter uma função que receba os índices inicial e final de uma reversão e efetue essa reversão. Além de receber os índices inicial e final da reversão, essa função precisará receber outros parâmetros (pois o programa não tem variáveis globais).

Bom trabalho!