Algoritmos e Estruturas de Dados BCC 2018/2

Exercício Programa 3: Análise de sequências

Rodrigo de Souza

18 de dezembro de 2018

1 Enunciado

Sequências de caracteres ou números inteiros aparecem em muitas aplicações importantes da Computação como Biologia Computacional e Criptografia. Tipicamente, interessa conhecer certas propriedades de uma sequência, a fim de podermos responder questões como existência de regularidades ou se a sequência parece ter sido gerada aleatoriamente. Para isso, estatísticas como frequências e comprimentos de subsequências especiais são de grande utilidade.

Neste EP, nosso objetivo é ler uma sequência de inteiros de tamanho desconhecido (você pode imaginar que é uma longa cadeia de DNA, ou um fluxo de medidas recebidas de uma fonte cujo término não podemos prever) que está armazenada em um arquivo, e ser capaz exibir *em tempo real* algumas informações sobre a mesma. São elas:

I-1 Impressão do conjunto de inteiros que apareceram até o momento, com a quantidade absoluta de ocorrências de cada um, e sua densidade (= relação entre quantidade de ocorrências e tamanho da sequência lida);

I-2 Impressão da maior pirâmide (explicação mais abaixo) da sequência lida até o momento;

I-3 Dado um inteiro p, impressão da lista de valores, *em ordem crescente*, que ocorrem exatamente p vezes na sequência lida até o momento;

Dizer que seu sistema funciona em tempo real (*online*) significa que o sistema atualiza a cada inteiro lido suas estruturas de dados de forma a permitir que essas operações sejam realizadas eficientemente a qualquer momento. Claro que a escolha da estrutura de dados tem impacto direto na eficiência de sua solução, e o principal objetivo deste exercício é precisamente o projeto dessas estruturas. Ou seja, não adianta simplesmente armazenar os inteiros já lidos em uma lista e percorrer essa lista sistematicamente para realizar as operações.[1]

[1] Não vamos na verdade testar esse aspecto *online* do seu programa, porque, como você verá na sequência, só vou cobrar que ele imprima os dados ao término da leitura. Mas você deve implementar as estruturas de dados solicitadas e atualizá-las a cada inteiro lido. Ou seja, seu programa deve ser efetivamente *online*.

Ao término da leitura da sequência, seu programa deverá gravar em um novo arquivo os dados I-1, I-2 e I-3 definitivos, e uma codificação comprimida da sequência conhecida como *Run-Length Encoding* ou RLE. A ideia da RLE é simples e baseia-se no conceito de *run*, que vamos chamar aqui de *carreira*. Uma carreira é simplesmente uma subsequência maximal (não pode ser estendida nem para esquerda nem para direita) de valores repetidos. Na codificação, uma carreira é representada por um par de inteiros, contendo o valor repetido, e o número de repetições. A RLE de nossa sequência é uma nova sequência, formada por esses pares de inteiros, representando as carreiras da sequência, na ordem em que aparecem.[2]

Uma *pirâmide* é uma subsequência estritamente crescente de inteiros, que começa ou no início da sequência de entrada ou após uma descida (= inteiros consecutivos a_i, a_{i+1} com $a_i > a_{i+1}$), e é seguida pela mesma subsequência, mas espelhada (ou seja, o reverso dessa subsequência crescente, que é uma subsequência decrescente). Sua sequência de entrada pode ter várias pirâmides, e o item I-2 pede para imprimir o comprimento da maior delas (imprimir a maior pirâmide complicaria um pouco; vamos pedir só o comprimento). Para isso, você *deve usar uma estrutura de pilha* que empilha sempre que uma subsequência crescente é lida (começo da sequência de entrada, ou logo após uma descida), e desempilha para cada elemento correspondente na subsequência espelhada.

Sobre o item I-3, você deve imprimir somente os números p tais que existam inteiros que ocorrem exatamente p vezes (por exemplo, se nenhum inteiro ocorre exatamente 3 vezes na sequência, você não vai imprimir o 3). Você deve imprimir no arquivo de saída esses inteiros p em ordem crescente.

2 Exemplo

Considere a seguinte sequência:

$$30, 20, 50, 50, 20, 10, 40, 30, 20, 20, 60, 60, 60, 50, 50,$$
$$40, 30, 30, 40, 30, 50, 70, 70, 50, 30, 30, 30, 30, 60, 40, 10$$

Conforme dissemos, o arquivo de saída deve ter a impressão dos itens I-1, I-2 e I-3, além da codificação RLE.

Para o item I-1, a impressão para este exemplo é a seguinte tabela de três colunas, apresentando, respectivamente, os inteiros que aparecem na sequência, as quantidades absolutas de ocorrência e as densidades (aqui expressas como uma fração entre a quantidade de ocorrências e o tamanho da sequência, 31; mas você vai fazer a conta, e o resultado é um valor do tipo float).

10	2	2/31
20	4	4/31
30	9	9/31
40	4	4/31
50	6	6/31
60	4	4/31
70	2	2/31

[2]Note que nem sempre a representação RLE garante compressão – isso só acontece se a sequência tiver carreiras longas. Para entender melhor, veja mais detalhes e alguns exemplos em `https://www.fileformat.info/mirror/egff/ch09_03.htm`.

Passando ao item I-2, notamos que essa sequência tem duas pirâmides: $20, 50, 50, 20$ e $30, 50, 70, 70, 50, 30$. A primeira tem comprimento 3, a segunda comprimento 6, então seu programa simplesmente imprime 6.

Sobre o item I-3, vemos que os inteiros que representam números exatos de ocorrência são $2, 4, 6, 9$ – basta olhar na segunda coluna da tabela do primeiro item.[3] Queremos imprimir esses valores em ordem crescente e, para cada número p de ocorrências, a lista de inteiros (em ordem crescente) que ocorre exatamente p vezes. Para nosso exemplo, seu programa vai imprimir o seguinte no arquivo de saída:

$$\begin{array}{ll} 2 & 10, 70 \\ 4 & 20, 40, 60 \\ 6 & 50 \\ 9 & 30 \end{array}$$

Finalmente, a codificação RLE desse exemplo é a seguinte:

$$1.30, 1.20, 2.50, 1.20, 1.10, 1.40, 1.30, 2.20, 3.60, 2.50, 1.40,$$
$$2.30, 1.40, 1.30, 1.50, 2.70, 1.50, 4.30, 1.60, 1.40, 1.10$$

Nessa notação, escrevemos, para cada carreira, o tamanho da carreira (quantidade de ocorrências repetidas), em seguida um ponto, em seguida o inteiro que se repete na carreira, uma vírgula, e aí começa uma nova carreira. Isso é uma longa cadeia, sem espaços.

3 Estruturas

Como já dissemos, você poderia resolver esse problema de forma trivial, guardando a sequência em um vetor ou uma lista e percorrendo exaustivamente essa lista a cada solicitação. Mas isso é muito ineficiente em tempo e espaço.

Então, seu programa deverá ter um mínimo de sofisticação, e deverá obrigatoriamente implementar a solução descrita abaixo.

A entrada é lida de um arquivo de texto puro, `seq.in` (por favor, use esse nome em sua função de leitura), contendo inteiros separados por espaço. Você não sabe o tamanho desse arquivo, não sabe a quantidade de inteiros que tem nele. Seu programa deve necessariamente manter as seguintes estruturas, que são atualizadas a cada inteiro lido:

ED-1 Uma fila de carreiras, para produzir no final a codificação RLE;

ED-2 Uma árvore binária de busca para armazenar os números p de ocorrências, e que armazena, em cada nó, além da quantidade p, uma lista ligada com os inteiros que ocorrem exatamente p vezes (item I-2);

ED-3 Uma tabela de espalhamento para armazenar os inteiros lidos e seus números de ocorrências (item I-1);

ED-4 Uma pilha de inteiros para detectar pirâmides (item I-3).

[3] Todavia, para resolver o item I-3, você não pode simplesmente copiar os valores da segunda coluna da tabela. Você deverá, como estamos orientando aqui, armazenar esses números de ocorrência em uma árvore binária, e fazer consultas a essa árvore.

Implemente uma pequena biblioteca contendo as implementações dessas estruturas. Deixe essas implementações em arquivos separados de seu programa principal; seu programa principal simplesmente faz a leitura, e atualiza as estruturas de dados, chamando para isso as funções pertinentes implementadas em sua biblioteca. Ou seja, *seu programa principal é curto* e *não implementa nenhuma estrutura de dados ou função para manipulação de estruturas.* A organização modular do seu programa será levada em consideração na avaliação.

A seguir, detalhamos um pouco o uso que seu programa deverá fazer dessas estruturas.

- Sua tabela de espalhamento (hashing) de inteiros deve usar encadeamento para tratar colisões. Você pode escolher o tamanho da tabela e a função de espalhamento; para mais detalhes consulte

  `https://www.ime.usp.br/~pf/algoritmos/aulas/hash.html`

  Ou seja, nas listas ligadas usadas para tratar colisões, cada nó da lista tem, além do inteiro lido na sequência, um outro inteiro, para o número de ocorrências. Para resolver o item I-1, seu programa simplesmente percorre a tabela e imprime os valores dos nós (neste item, não precisa imprimir os inteiros em ordem crescente).

- Para tratar as solicitações de lista de inteiros com exatamente p ocorrências, você deve implementar uma *árvore binária de busca.* Veja os detalhes em

  `https://www.ime.usp.br/~pf/algoritmos/aulas/binst.html`

  Cada nó de sua árvore guarda, além do número p de ocorrências, uma lista ligada com os inteiros que ocorrem exatamente p vezes. A cada inteiro lido, além da atualização na tabela de espalhamento, você deve procurar o antigo número de ocorrências desse inteiro na árvore, retirar o inteiro da lista desse nó, e colocar o inteiro na lista do nó com a quantidade atualizada de ocorrências. Cuidado com a situação em que um nó fica com uma lista vazia de inteiros (*neste caso, ele deve ser removido da árvore*). Para imprimir as quantidades de ocorrência em ordem crescente, seu programa deve percorrer a árvore na ordem e-r-d.

  O melhor mesmo seria usar uma árvore com balanceamento (AVL), mas não vou exigir isso neste exercício.

4 Instruções gerais

- Seu programa deve ser feito em C.
- Documente cada função dizendo o quê ela faz. Se o seu código não está legível, fica muito difícil adivinhar o que a função faz e isso prejudica a correção. Esse comentário deve especificar o que a função recebe (os parâmetros) e o que devolve.

- Você deve separar o código principal da implementação das estruturas de dados. Ou seja, implemente as estruturas em arquivos .c e .h separados; depois, entregue tudo em um .zip. Comente claramente no seu código que estruturas usou, bem como o funcionamento e complexidade das operações pertinentes.
- Capriche. Cuidado com a indentação. Deixei seu programa suficientemente modularizado para que cada tarefa específica esteja implementada em uma função (documentada explicando o que recebe e o que faz). Faça isso em particular para as estruturas de dados, implementando funções que realizam as operações pertinentes. Modularização e capricho serão considerados na nota.
- Escreva no início do código um cabeçalho com comentários, indicando nome, número do EP, data, nome da disciplina.
- A entrega será eletrônica no ambiente Moodle da disciplina (não receberei exercícios impressos ou via email).