Trabalho Prático 0

Wilson Moreira Tavares

September 21, 2015

1 Introdução

Anagrama é uma palavra ou frase que é constituida da alteração da ordem dos caracteres de uma outra frase ou palavra. Uma característica essencial dos anagramas é que duas palavras que são anagramas possuem exatamente os mesmos caracteres dispostos de diferentes formas, como por exemplo "roma" e "ramo". O objetivo deste trabalho prático é implementar um algoritmo capaz de detectar e agrupar anagramas entre determinadas palavras que foram dadas, imprimindo em ordem decrescente o tamanho dos grupos de anagramas que foram formados.

Na entrada, o programa recebe um inteiro N, esse inteiro é o número de casos que serão inseridos nessa execução. Executamos N vezes a operação de ler uma linha inteira de até 10^6 palavras que podem possuir até 50 caracteres. Durante a leitura dessa linha, agrupamos essas palavras em lotes de anagramas, imprimindo o tamanho de cada lote.

Para a elaboração da solução, foi observada a característica de que duas palavras que são anagramas possuem exatamente os mesmos caracteres, e consequentemente são exatamente a mesma palavra quando ordenadas.

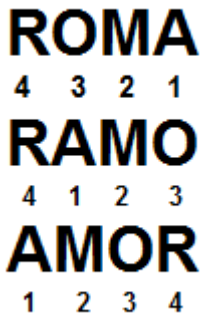

Figure 1: Por exemplo: "roma" e "ramo" em ordem alfabética são "amor".

Com essa propriedade definida, o algoritmo deve simplesmente agrupar palavras ordenadas iguais em lotes, imprimindo no fim o tamanho de cada lote em ordem decrescente.

2 Solução do problema

Para solucionar o problema, foi implementada uma lista de palavras, com uma estrutura que abriga a palavra já ordenada, o número de vezes que essa palavra se repete e um apontador para a próxima palavra da lista.

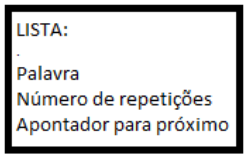

Figure 2: Estrutura utilizada para a implemetação do algoritmo

A entrada do programa consiste em uma linha com um inteiro N($1<N\leq10$), seguida de N linhas de até 10^6 palavras contendo até 50 caracteres, alocar todo esse espaço ($5x10^7$) somente para a entrada seria muito custoso. Por esse motivo, o algoritmo, na função *estruturaLista*, lê caractere por caractere, inserindo em um vetor de 51 posições. Quando uma palavra chega ao fim (espaço ou \n) o algoritmo executa todas as manipulações e comparações possíveis para a construção da lista, que serão descritas nos próximos parágrafos.

Após a função *estruturaLista* reconhecer e armazenar uma palavra completa no vetor, a função chamada é a *contabiliza*, que ordena a palavra através da função *qsort*, presente a biblioteca *stdlib.h*. Após a ordenação, a função contabiliza a palavra na lista, sendo como uma nova palavra ou como uma repetição de uma palavra já existente na lista, é importante ressalvar que anagramas ordenados são a mesma palavra.

Para contabilizar a palavra na lista existem dois métodos, inserir a palavra na lista ou acrescentar um no contador de número de execuções da palavra. A palavra é inserida em dois casos, quando a lista está vazia ou quando a palavra ainda não existe na lista. O contador é acrescido quando a palavra já existe, a existência é verificada através da comparação entre as palavras presentes na lista e a palavra a ser contabilizada. A função utilizada é *strcmp*, presente na biblioteca *string.h*.

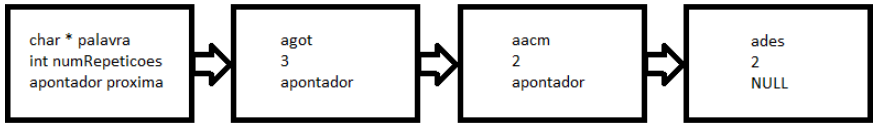

Figure 3: Exemplo de uma lista alocada, caso retirado do *toy example*.

Durante o processo de montagem da lista, há uma variável que armazena o tamanho da lista da atual execução. Armazenar esse tamanho é possível pois a função *contabiliza* retorna 1 quando uma nova palavra é inserida e 0 caso contrário, e esses valores são acrescidos nessa variável que armazena o tamanho da lista. A lista da $Figure3$ teria tamanho 3, por exemplo.

Sabendo quantos componentes existem na lista, a função $fazVetor$ organiza o número de repetições de cada palavra em um vetor de inteiros, chamando posteriormente a função $imprimeVetor$, que ordena o vetor que foi passado como parâmetro e imprime.

Após imprimir na tela o vetor ordenado, a lista é liberada através da função *liberaLista*. A primeira posição da lista é imediatamente alocada caso haja uma ou mais execuções pendentes.

3 Análise teórica de custo assintótico

3.1 Espaço

Para definir o custo assintótico do ponto de vista de espaço, vamos considerar uma entrada com N casos de teste (N=1), contendo P palavras com tamanho médio de C caracteres e um número G de grupos formados entre essas palavras.

- A função *estruturaLista* aloca um vetor de caracteres de 51 posições, chamando posteriormente a função *contabiliza*. $\theta(51)$.
- A função *contabiliza* faz uso da função nativa *qsort* que possui complexidade de espaço $\theta(\log C)$. Podendo inserir ou não uma nova palavra, quando uma nova palavra é inserida, é chamada a função $insereNovaPalavra$.
- A função $insereNovaPalavra$ cria uma nova palavra na lista, então a mesma aloca uma posição nova para a lista e aloca um vetor do tamanho da palavra recebida. O custo é $\theta(1)$ para a criação do novo campo da lista e $\theta(C)$ para a palavra.
- A função $fazVetor$ cria um vetor de tamanho G, ocupando $\theta(G)$ na memória, chamando a função $imprimeVetor$.
- A função $imprimeVetor$ faz uso da função *qsort*, que por sua vez, apresenta complexidade de espaço $\theta(\log G)$.

  Dados os itens acima, a complexidade de espaço do algoritmo completo é:

$$\theta(51 + P \log C + G + GC + G + \log G)$$

  Ficando:

$$\theta(GC + 2G + P \log C + \log G + 51)^1$$

[1] A notação acima se aplica a N execuções, o valor não se altera, já que os números abordados são uma média entre os casos e que a cada execução a memória correspondente é liberada.

3.2 Tempo

Para definir o custo assintótico do ponto de vista de tempo seguiremos as mesmas diretrizes do item 3.1, vamos considerar uma entrada com apenas um caso de teste, contendo P palavras com tamanho médio de C caracteres e um número G de grupos formados entre essas palavras.

- A função *estruturaLista* possui um laço de tamanho C que é executado P vezes. $O(C \times P)$.
- A função *contabiliza* faz uso da função *qsort*, que por sua vez possui complexidade $O(C \log C)$[2] no caso médio. Além disso há mais P*[3] comparações.
- A função *insereNovaPalavra* é $O(C)$, pois copia um vetor de caracteres de tamanho médio C.
- A função $fazVetor$ é da ordem $O(G)$, pois aloca e popula um vetor de tamanho médio G.
- A função *imprimeVetor* faz uso da função *qsort*, que possui caso médio $O(G \log G)$[2] e imprime um vetor de tamanho G, $O(G)$.
- A função *liberaLista* possui complexidade $O(P)$, já que libera todas as posições da lista de tamanho P.

Dadas as análises acima, chegamos a um custo linear, o que pode aumentar esse custo é a função *qsort* no seu pior caso, veremos se haveŕa influência nos testes executados.

4 Execução e análise de experimentos

4.1 Diretrizes

Para medir o tempo de execução de um programa foi usado o comando *time./program*[4], as diretrizes das entradas são as seguintes:

- Em todas as entradas $N = 10$, ou seja, 10 casos de teste em cada entrada.
- O tamanho médio das palavras é de 40 caracteres.
- O número de palavras para cada teste foi o seguinte:
  1. 10^3 palavras em cada linha.
  2. 10^4 palavras em cada linha.
  3. 10^5 palavras em cada linha.
  4. 10^6 palavras em cada linha.

[2] O pior caso possui complexidade quadrática, o que não se aplica muito na prática.
[3] P* é o tamanho da lista no momento exato da iteração.
[4] Foram utilizados como teste os arquivos disponibilizados no moodle.

4.2 Conclusão

- A medida do tempo de execução dos testes apresentou os seguintes tempos de execução:

    1. 0,086s
    2. 0,725s
    3. 7,177s
    4. 74,613s

Foi possível perceber que o tempo de execução cresce em ritmo aproximado de 10, assim como a entrada. Com esses testes, é possível comprovar que a complexidade do tempo é linear.

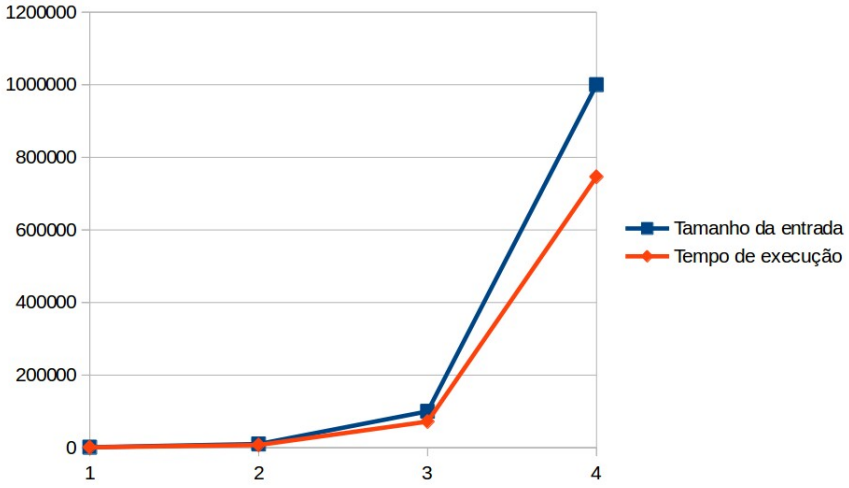

Figure 4: Comportamento assintótico da função, tempo em $s \times 10^{-4}$.