TP0: Anagramas

Reuben Nascimento Morais

21 de setembro de 2015

1 Introdução

Esse trabalho prático pode ser dividido em duas partes: identificar anagramas, e contar o número de cópias de um elemento em uma lista. Com essas duas peças, é simples implementar o objetivo final do trabalho: agrupar e contar anagramas em listas de palavras.

A solução apresentada é baseada na observação de que ao ordenar os caracteres de várias strings, anagramas correspondem a strings idênticas, e na clássica combinação de ferramentas Unix: `sort arquivo | uniq -c | sort -n`. Em prosa: ordene a entrada, conte o tamanho dos grupos, ordene os grupos por tamanho em ordem decrescente. Ao ordenar uma sequência, itens idênticos se agrupam, e a tarefa de contar os tamanhos dos grupos se torna trivial.

O problema soa bem artificial, e quanto à motivação, o caso menos surreal que vem à mente é o de apresentar a alunos os requerimentos dos trabalhos de uma disciplina, assim como servir como aquecimento para o semestre e para garantir que o conteúdo dos semestres passados ainda não foi esquecido. Distante da nossa realidade, mas ainda assim um caso válido!

2 Modelagem do problema

A lógica principal do programa é aproximada pelo seguinte pseudo-código:

Código 1: Lógica principal

```
1 para cada linha da entrada:
2     para cada palavra em linha:
3         ordene os caracteres de palavra
4     ordene linha palavra a palavra
5     conte os grupos de palavras iguais em linha
6     ordene os grupos por tamanho em ordem decrescente
```

A ideia é primeiro fazer a ordenação interna de cada palavra na linha, e depois a ordenação da linha inteira, palavra a palavra. Após esses dois passos, a entrada terá duas propriedades úteis: todos os anagramas estarão reduzidos a palavras idênticas; e palavras idênticas estarão agrupadas na lista. Basta então contar os grupos e imprimir o resultado.

Abusando da liberdade oferecida por C, e utilizando a máxima "todo problema em ciência da computação pode ser resolvido com mais camadas de indireção", é possível fazer tudo isso sem copiar a entrada nenhuma vez. A entrada é percorrida caractere a caractere, e quando o final de uma palavra é identificado, a palavra é então ordenada, e seu endereço é adicionado à lista de palavras. Esse processo se repete até o fim da entrada. As figuras 1, 2, 3 e 4 ilustram esse processo.

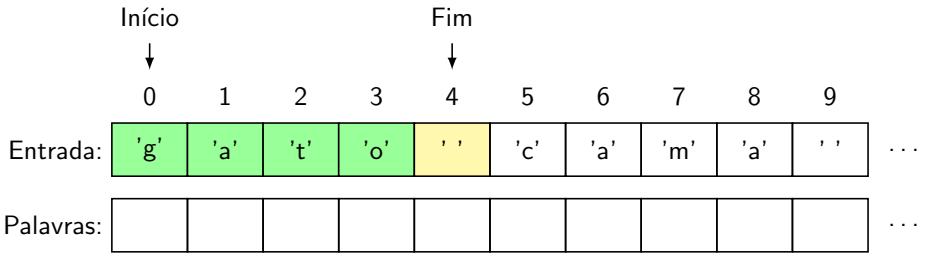

Figura 1: O final de uma palavra é identificado

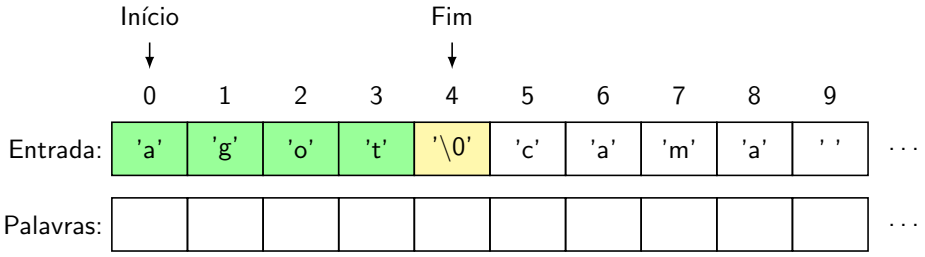

Figura 2: A palavra é ordenada

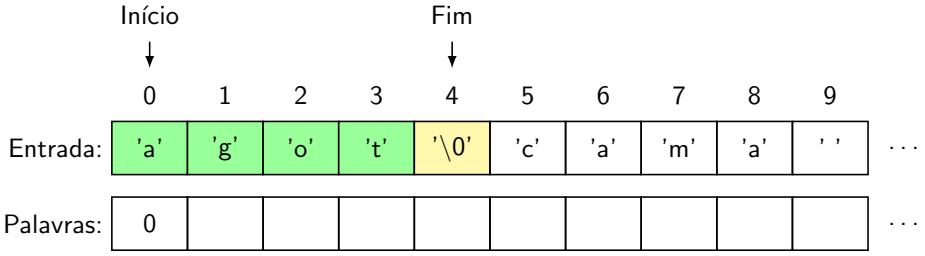

Figura 3: A palavra atual é inserida na lista de palavras

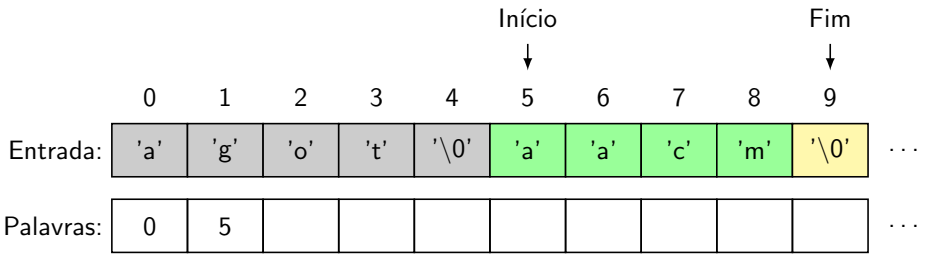

Figura 4: O algoritmo continua até o final da entrada

Ao final da execução desse algoritmo, todas as palavras estão ordenadas internamente. A lista de palavras é então ordenada, e por fim os grupos de palavras idênticas são contados, como descrito no pseudo-código 2.

Código 2: Contagem de grupos

```
total_atual = 1
grupo_atual = 0
atual = palavras[0]
para i de 1 ate tamanho(palavras):
    se palavras[i] != atual:
        totais[grupo_atual] = total_atual
        total_atual = 1
        grupo_atual = grupo_atual + 1
        atual = palavras[i]
    senao:
        total_atual = total_atual + 1
```

Finalmente, a lista `totais` é ordenada em ordem decrescente e impressa na saída principal.

3 Análise teórica do custo assintótico

Para as análises a seguir, $\boldsymbol{P}$ é o número de palavras na entrada, e $\boldsymbol{p}$ é o tamanho médio das palavras, e finalmente $\boldsymbol{n = P * p}$ é o número total de caracteres na entrada.

3.1 Análise teórica do custo assintótico de tempo

A análise do custo assintótico de tempo pode ser divida em dois algoritmos: ordenação e contagem de grupos. Todas as ordenações são feitas utilizando a função padrão `qsort`, que, apesar de não especificar qual algoritmo deve ser usado por implementações, na prática é implementada usando o algoritmo Quicksort[1][2][3], de custo assintótico $O(n \log n)$. A contagem do tamanho dos grupos é feita em um único loop sobre os elementos da lista ordenada de P palavras, executando $O(p)$ operações em cada iteração. O custo assintótico total da contagem é então $O(P * p) = O(n)$.

Analisando o programa por partes:

- Todas as P palavras são ordenadas internamente: $O((P * p) \log(P * p)) = O(n \log n)$.
- A lista de palavras é ordenada: $O(P \log P)$.
- Os tamanhos dos grupos de palavras idênticas são contados: $O(n)$.
- A lista de tamanhos dos grupos é ordenada: $O(P)$.

Como $P < n$, temos que $O(P) = O(n)$ e $O(P \log P) = O(n \log n)$, e portanto o custo total do programa é

$$O(n \log n + P \log P + n + P) = O(n \log n + n \log n + n + n) = O(n \log n)$$

3.2 Análise teórica do custo assintótico de espaço

Duas alocações são feitas para cada linha da entrada: o buffer necessário para ler a própria linha, e uma lista de ponteiros para `char` que representa a lista de palavras. Temos então espaço $O(n)$ para ler a entrada, e $O(P)$ para armazenar a lista de palavras. Como $P < n$, o custo total é

$$O(n + P) = O(n + n) = O(n)$$

4 Análise de experimentos

Devido à simplicidade do problema e da solução (menos de 100 linhas de código, sem contar a estrutura auxiliar de lista), não realizei nenhum experimento para comparar com a análise teórica de complexidade. Além disso, confundir a data de entrega do trabalho com 28 de Setembro certamente não ajudou.

5 Conclusão

Nesse trabalho, foi resolvido o problema de agrupar e contar o número de repetições de anagramas em listas de palavras. O problema foi resolvido utilizando ordenação e contagem linear de grupos. A análise de complexidade teórica de tempo não foi comprovada experimentalmente.

Referências

[1] Apple Computer Inc., NeXT Computer Inc., and The Regents of the University of California. XNU. `http://www.opensource.apple.com/source/xnu/xnu-2782.40.9/bsd/kern/qsort.c`, 2015.

[2] Douglas C. Schmidt. glibc. `https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/qsort.c`, 2015.

[3] The Regents of the University of California. FreeBSD. `https://github.com/freebsd/freebsd/blob/master/lib/libc/stdlib/qsort.c`, 2015.