# Algoritmo e Estrutura de Dados III

Documentação do Trabalho Prático 1

Ramon Gonçalves Gonze

15 de junho de 2017

## 1. INTRODUÇÃO

Hetelberto Topperson é uma pessoa que conversa frequentemente através de um aplicativo de troca de mensagens. O problema de Hetelberto é lembrar das pessoas com quem ele teve uma determinada conversa. Seu desejo é que o aplicativo possua um buscador que retorne um trecho de uma conversa quando uma palavra for pesquisada. Supondo que o buscador em questão já exista, para o funcionamento deste, necessitamos de um índice invertido das palavras das conversas. Esse índice consiste em uma lista ordenada de forma lexicográfica de todas as palavras em conversas no aplicativo.

Consideremos cada item dessa lista um registro, que contém uma palavra **w** - que tem até 20 caracteres do alfabeto minúsculo (a-z) -, o documento (conversa) **d** que esta palavra se encontra, a frequência **f** da palavra na conversa e a posição **p** em bytes da palavra na conversa. O valores de **d**, **f** e **p** serão armazenados como números inteiros (**int**) de **4** bytes, e a palavra como um vetor de caracteres (**char**) de **20** bytes. Logo, o tamanho máximo de um registro é de **32** bytes.

Há uma restrição para o algoritmo que construa esse índice invertido: a memória disponível é limitada. Para isso, o algoritmo deve utilizar ordenação externa para ordenar lexicograficamente o índice invertido. Ziviani (2013, p.139) descreve “Nas memórias externas, tais como fitas, discos e tambores magnéticos, os dados são armazenados como um arquivo sequencial, em que apenas um registro pode ser acessado em dado momento.”. Utilizaremos arquivos de texto como fitas, nas quais os registros serão colocados e posteriormente intercalados.

## 2. SOLUÇÃO DO PROBLEMA

Este trabalho prático consiste na implementação de um algoritmo de ordenação por intercalação balanceada de vários caminhos. Consideremos **M** o tamanho máximo de nossa memória interna em bytes. Dado um arquivo a ser ordenado, que neste caso será o índice invertido desordenado, o algoritmo executa três etapas:

1) Construir fitas (no máximo **M**) com blocos de no máximo **M** registros cada um;
2) Intercalar os blocos das fitas, colocando o resultado desta intercalação em uma fita vazia. Nesta etapa, ao fim de cada n-ésimo bloco intercalado de todas as fitas, uma nova fita vazia será utilizada para intercalar os (n + 1)-ésimo blocos (se estes existirem). Poderão ser utilizadas no máximo **M** novas fitas vazias;
3) Repetir a segunda etapa até que reste somente uma fita, que conterá todo o nosso índice ordenado.

O **Algoritmo 1** explica o funcionamento da intercalação balanceada:

**Algoritmo 1**

```
function intercalacaoBalanceada(quantidade_max_de_fitas)
1       while houver mais de uma fita do
2               while todas as fitas não esvaziarem do
3                       coloca o primeiro registro do bloco de cada fita na memória interna
4                       while o bloco que está sendo intercalado não acabar do
5                               escreve o menor elemento da memória interna na nova_fita
6                               substitui o elemento retirado pelo próximo da fita que ele veio
7                       escreve na nova fita os elementos que restaram na memória interna
8                       nova_fita ← nova_fita + 1
9       return index_ordenado
```

O **Algoritmo 2** explica o funcionamento o programa:

**Algoritmo 2**

```
function constroiIndiceInvertido(M)
1       index ← constroiIndiceDesordenado();
2       quantidade_max_de_fitas ← (M/ 32) // 32 é o tamanho máximo de cada registro
3       controiFitas(index, quantidade_max_de_fitas);
4       index ← intercalacaoBalanceada(index, quantidade_max_de_fitas);
5       index ← contaFrequenciaDeCadaRegistro(index);
6       return index
```

A estratégia do algoritmo é realizar uma primeira passada sobre o index desordenado, quebrando-o em blocos de tamanho **M**. Cada bloco é ordenado internamente na memória interna através do algoritmo *quicksort*, e então salvo em uma fita. O blocos ordenados são intercalados através do **Algoritmo 1**. Estipulando a *quantidade_max_de_fitas* = **M**/32,

garantimos que no momento das intercalações, só seja movido para a memória interna **M**/32 registros, respeitando assim a restrição do problema. No final da intercalação, restará somente uma fita que estará ordenada por **w**, **d** e **p**. A função *contaFrequenciaDeCadaRegistro()* é então chamada, para escrever a frequência de cada registro. Ela conta a quantidade de uma determinada palavra **w** em um determinado documento **d**, e então atualiza **f**, formando finalmente o índice invertido.

## 3. ANÁLISE DE COMPLEXIDADE

A análise de complexidade das funções será feita em relação aos acessos à memória secundária, que são os arquivos de texto. Como o acesso à memória secundária tem um custo muito mais alto que operações na memória primária, devemos considerar que as operações em memória primária serão *O(1)*. Sendo *m* o limite de memória interna, *n* a quantidade total de registros e *f* a quantidade de fitas (que é também a quantidade máxima de registros que se pode ter na memória interna → $f = m/32$), seguem as análises:

- ➔ **buildUnsortedIndex(D, E, S):** A função faz a leitura de todas as palavras dos documentos, constrói o registro de cada uma (**w**, **d**, **1**, **p**) e o escreve no índice desordenado. Esse procedimento de leitura e escrita é feito palavra por palavra, portanto tem-se um custo fixo de memória, o que faz a complexidade espacial ser *O(1)*. A soma das palavras de todos os documentos é *n*, logo a quantidade total de acesso aos arquivos é *n*. Sua complexidade temporal é *O(n)*.
- ➔ **buildTapes(S, tapes_amount):** A função possui um loop principal que percorre o índice desordenado, ou seja, é executado *n* vezes. Inserido nesse loop, há a chamada do *Quicksort* e outros dois loops: o primeiro loop escreve *f* registros na memória interna. Logo após este, é chamado o *Quicksort*. Posteriormente, o segundo loop escreve os registros do vetor retornado pelo *Quicksort* em uma fita, logo ele é executado no máximo *f* vezes. A complexidade temporal final da função é $O(n * (O(f) + O(1) + O(f))) = O(nf)$ (ou $O(nm/32) = O(nm)$). Em relação à complexidade espacial, a leitura (no índice desordenado) e a escrita (nas fitas) dos registros são feitas utilizando um vetor de tamanho *f*. Portanto sua complexidade espacial é *O(f)* (ou $O(m/32) = O(m)$).

➔ **intercalate(int tapes_amount, char S[]):** A complexidade temporal desta função se deve diretamente aos três valores, de *n*, *m* e *f*. A quantidade de acessos à memória secundária é $log_f \frac{n}{m}$ para uma intercalação-de-*f*-caminhos (Ziviani, 2013). Logo, sua complexidade tempora é $O(log_f \frac{n}{m})$. Para a capacidade espacial, temos variáveis auxiliares (valor ocupado na memória sempre é o mesmo), um vetor de tamanho *f* que contém os registros que estão sendo intercalados e um vetor de caracteres *V_status* (**char**) também de tamanho *f* que indica a situação atual de cada fita . Portanto, sua complexidade espacial é $O(f) + O(f) = O(f)$ (ou $O(m/32) + O(m/32) = O(m)$).

O programa principal é a implementação do **Algoritmo 2**, que chama 3 funções. As complexidade das três primeiras já foram apresentadas. Para a contagem das frequências dos registros, a função *contaFrequenciaDeCadaRegistro()* abre o arquivo duas vezes (com dois ponteiros). Cada um destes ponteiros percorre o *index* - que possui *n* registros - uma vez, acessando cada registro somente uma vez. E durante esses acessos, é aberto um outro arquivo, o novo *index*, onde os *n* registros, já com a frequência contada, serão escritos. Como três arquivos são acessados *n* vezes, a complexidade temporal da função é $O(n^3)$. A função utiliza-se de duas variáveis que comparam os registros (para contar suas frequências). Logo, no máximo dois registros ficam armazenados na memória interna, o que torna a complexidade espacial $O(1)$.

## 4. AVALIAÇÃO EXPERIMENTAL

Os testes foram feitos em um ambiente Linux utilizando a distribuição Ubuntu 16.04 LTS, em um computador com processador core i3 de 2.1 GHz e 6GB de RAM. Os testes exaustivos foram realizados com o intuito de verificar qual o comportamento do algoritmo frente à variação da quantidade total de de palavras.

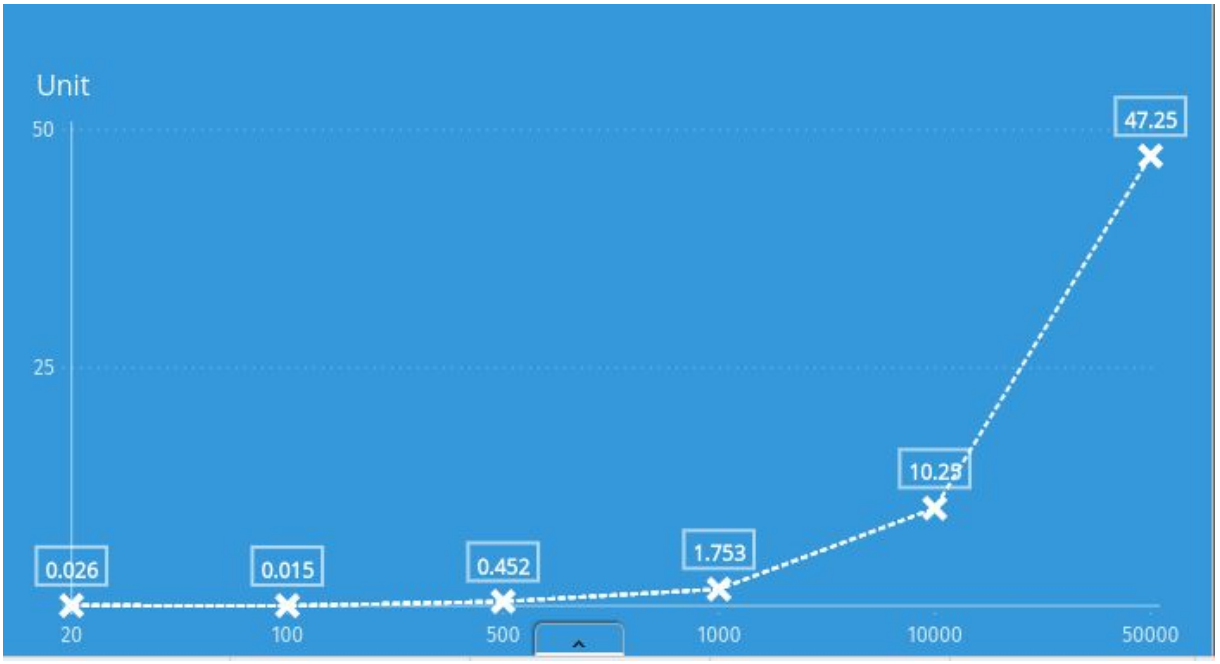


*Gráfico 1: Tempo de execução do algoritmo com relação à quantidade de palavras, utilizando uma memória de 160 bytes*

Pode-se notar com a análise que o tempo de execução do algoritmo cresce de forma linear em realação á quantidade de palavras.

## 5. REFERÊNCIAS

ZIVIANI, N. Projeto de Algoritmos. Terceira edição. São Paulo. 2013. 639p.