

ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 10 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 1 de 3 sobre ordenação externa: intercalação balanceada.

1 INTRODUÇÃO

Esta série de tutoriais sobre *Algoritmos e Estruturas de Dados III* foi escrita usando o **Microsoft Windows 7 Ultimate, Microsoft Office 2010, Bloodshed Dev-C++** versão 4.9.9.2 (pode ser baixado em <http://www.bloodshed.net>), referências na internet e notas de aula do professor quando estudante. Ela cobre desde os algoritmos de ordenação, passando pela pesquisa em memória primária e culminando com a pesquisa em memória secundária.

Nós entendemos que você já conhece o compilador Dev-C++. No caso de você ainda não o conhecer, dê uma olhada nos tutoriais Dev-C++ 001 a 017, começando pelo [Tutorial Dev-C++ - 001 - Introdução](#).

Se não tem problemas com a linguagem C/C++ e o compilador Dev-C++, então o próximo passo é saber ler, criar e alterar arquivos em disco usando linguagem C/C++. Se ainda não sabe como fazê-lo, dê uma olhada nos tutoriais Dev-C++ 001 e 002, começando pelo [Tutorial Dev-C++ 001 - Criação, Leitura e Alteração de Arquivos](#).

Se sabe todas as coisas anteriores, então a próxima etapa é conhecer os algoritmos mais básicos de ordenação. Em minhas [notas de aula](#) você encontra um material básico, porém detalhado e com algoritmos resolvidos, dos principais métodos de ordenação existentes.

Adotaremos o livro **Projeto de Algoritmos com Implementação em Pascal e C**, Editora Cengage Learning, de Nivio Ziviani, como livro-texto da disciplina. Nele você encontrará os métodos de ordenação que iremos estudar.

Seu próximo passo será estudar os algoritmos de ordenação por [Inserção](#), [Seleção](#), [Shellsort](#), [Heapsort](#) e [Quicksort](#). Você pode usar os links anteriores ou fazer uso do livro-texto.

Se você estiver lendo este tutorial tenha certeza de ter seguido os Tutoriais AED 001 a 009. Agora que você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 INTERCALAÇÃO BALANCEADA

O algoritmo **mergesort** se baseia no conceito de dividir para conquistar. Sua implementação para ordenação externa utiliza a seguinte estratégia geral:

1. Na primeira passagem pelo arquivo, este o decompõe em blocos, do tamanho da memória interna disponível. Cada um destes blocos é ordenado na memória interna.
2. Os blocos ordenados são intercalados, percorrendo-se diversas vezes o arquivo. A cada passagem blocos cada vez maiores são criados, até que todo arquivo esteja ordenado.

Na maioria das implementações este é um algoritmo estável.

2.1 DESCRIÇÃO

Este método de ordenação consiste em um passo de ordenação inicial, seguidos de vários passos de intercalação.

2.2 IMPLEMENTAÇÃO PARA ORDENAÇÃO INTERNA

2.2.1 COMPLEXIDADE

Pior caso: $n \log n$

Caso médio: $n \log n$

Melhor caso: $n \log n$

2.3 IMPLEMENTAÇÃO DE ORDENAÇÃO EXTERNA

Primeira fase: criação dos blocos ordenados

Na primeira fase os elementos são distribuídos entre 1, 2, ..., P dispositivos externos (arquivos em discos rígidos, fitas etc.), em blocos ordenados de M elementos (tamanho máximo de memória interna), exceto eventualmente o bloco final, que pode ter menos elementos, caso N não seja múltiplo de M . Na distribuição, os primeiros M elementos são lidos da entrada, ordenados na memória interna, e escritos no dispositivo 1. Então, os próximos M elementos são lidos e armazenados no dispositivo 2, e assim por diante. Caso após adicionar um elemento no dispositivo P (último) ainda existam mais dados a serem distribuídos ($N > P \times M$), um segundo bloco ordenado é adicionado ao dispositivo P (primeiro), e outro adicionado ao dispositivo $P + 1$, e assim por diante até que toda entrada seja considerada.

Nota prática: Talvez seja interessante colocar um '*', ou outro caractere, que delimite os blocos ordenados em cada dispositivos. Exemplo: Dispositivo 1 -> AAB*MOP*LIJ*.

Segunda fase: intercalação.

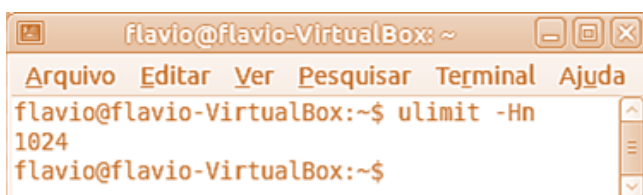
Na segunda etapa, os dispositivos 1 a P são considerados como dispositivos de entrada, enquanto os dispositivos de $P + 1$ a no máximo $2P$ são considerados dispositivos de saída.

1. O primeiro registro de cada dispositivo de entrada existente é lido.
2. Retira-se o registro contendo a menor chave.
3. Armazena-o em um dispositivo de saída.
4. Repete-se o passo 1 até haver somente um dispositivo *
5. Se for a primeira passagem, ao ler o último registro de um dos blocos ordenados, este dispositivo deve ser bloqueado. O dispositivo é desbloqueado quando o último registro do mesmo bloco nos outros dispositivos forem lidos.

Seria ideal poder intercalar todas as partições de uma só vez e obter o arquivo classificado, entretanto sistemas operacionais estabelecem número máximo de arquivos que podem ser abertos simultaneamente. Esse número pode ser bem menor do que o número de partições existentes.

Curiosidade

Ver o número máximo de arquivos que podem ser abertos no Linux:



```
flavio@flavio-VirtualBox: ~  
Arquivo Editar Ver Pesquisar Terminal Ajuda  
flavio@flavio-VirtualBox:~$ ulimit -Hn  
1024  
flavio@flavio-VirtualBox:~$
```

NA IMAGEM PODEMOS VER QUE O NÚMERO MÁXIMO DE ARQUIVOS ABERTOS AO MESMO TEMPO PODE CHEGAR A 1024

A intercalação vai exigir uma série de fases durante cada qual, registros são lidos de um conjunto de arquivos e gravados em outro (partições).

Estratégias de distribuição e intercalação:

- Intercalação balanceada de N caminhos
- Intercalação ótima

2.4 MEDIDA DE EFICIÊNCIA

Uma medida de eficiência do estágio de intercalação é dada pelo número de passos sobre os dados:

$$N^{\circ} \text{ de passos} = \frac{N^{\circ} \text{ total de registro lidos}}{N^{\circ} \text{ total de registros no arquivo classificado}}$$

Número de passos representa o número médio de vezes que um registro é lido (ou gravado) durante o estágio de intercalação.

3 QUICKSORT EXTERNO EFICIENTE

Quicksort Externo é um algoritmo para ordenação externa "in situ". Sua implementação na linguagem C é comparada com softwares eficientes para ordenação externa. As métricas utilizadas na comparação são tempo de execução e quantidade de espaço total utilizado na ordenação.

3.1 INTRODUÇÃO

Atualmente, a necessidade de ordenar grandes volumes de dados tem se tornado muito importante para grandes corporações, instituições governamentais, bancos e outros.

Como não é possível armazenar todos os dados na memória principal de um computador, mecanismos eficientes de ordenação em memória secundária, também conhecida como ordenação externa, devem ser utilizados.

Diferentemente dos algoritmos de ordenação interna, onde a principal operação considerada na análise dos algoritmos é a comparação entre chaves, o principal objetivo em ordenação externa é minimizar o número de operações de entrada e saída (E/S) executadas. Isto porque o tempo para se realizar uma operação de E/S é muito maior do que o tempo de uma operação da unidade central de processamento.

A gerência da quantidade de operações de E/S executadas por um algoritmo de ordenação pode ser feita de duas formas. A primeira delas é quando o algoritmo de ordenação é projetado para utilizar somente a memória interna disponível e com o objetivo de minimizar o número de operações de E/S efetuadas. Em [Leu,

00], é citado que o limite inferior, considerando o número de operações de E/S, para se ordenar n registros é dado por:

$$O\left(\frac{n \log \frac{n}{b}}{b \log \frac{K}{b}}\right) \quad \text{EQUAÇÃO I}$$

sendo K o número de registros que podem ser armazenados na memória interna; b a quantidade de registros que podem ser armazenados no bloco de leitura ou gravação do Sistema Operacional (SO). Caso $K \geq \sqrt{n \times b}$, o limite passa a ser

$$O\left(\frac{n}{b}\right) \quad \text{EQUAÇÃO II}$$

Na outra abordagem de gerência do número de operações de E/S, é utilizado um algoritmo de ordenação interna tradicional, tal como Quicksort [Hoare, 62], [Ziviani, 93] ou Heapsort [Ziviani, 93], porém não é levado em consideração que a memória interna seja limitada. Isto é possível em SOs que implementam um mecanismo de memória virtual, o qual gerencia as operações de E/S. Nessa abordagem, para minimizar o número de operações de E/S, os algoritmos devem possuir características que diminuam a quantidade de faltas de páginas no mecanismo de memória virtual. Sedgewick [Sedgewick, 83] cita que o Quicksort é um bom método para ser utilizado nesse ambiente, pois possui uma pequena localidade de referência. Isto diminui o número de faltas de páginas em um ambiente de memória virtual. Para estudos mais detalhados sobre o Quicksort em memória virtual pode-se consultar [Verkamo, 87].

Outro fator importante diz respeito à quantidade de memória extra utilizada para a ordenação externa. Basicamente, são utilizadas duas abordagens. A primeira é a ordenação “*in place*” [Monard, 80], também conhecida como ordenação “*in situ*”. Nela, a entrada é ordenada de forma que o arquivo original seja sobrescrito. O método utiliza somente $O(\log n)$ unidades de memória interna, onde n é o número de registros a serem ordenados, sendo que não é necessário nenhuma memória externa além da que é utilizada pelo arquivo original. Na outra abordagem, não há restrição quanto ao uso extra de memória interna ou externa.

Na seção 3.2, é apresentado o Quicksort Externo, um algoritmo para ordenação externa proposto por Monard [Monard, 80] que se enquadra na abordagem de ordenação “*in situ*” e utiliza uma

técnica semelhante ao Quicksort tradicional [Hoare, 62], [Ziviani, 93]. Na seção 3.3, o Quicksort Externo é comparado com softwares que realizam eficientemente a ordenação externa. Os softwares escolhidos foram o **Nitrosort** [Nitrosort, 03] e o **Sort**. O primeiro foi escolhido por ser um produto comercial muito eficiente e o segundo por ser muito popular, já que é um aplicativo presente em vários SOs. As principais métricas utilizadas na avaliação foram a quantidade de espaço extra utilizado por cada um deles e a eficiência em termos de tempo de execução. Apesar de não ser um dos objetivos primários, foram realizados testes com o Quicksort tradicional fazendo uso do ambiente de memória virtual, e os resultados foram comparados com os do Quicksort Externo. Enfim, na seção 3.4 o tutorial é concluído e alguns estudos são propostos.

3.2 QUICKSORT EXTERNO

O Quicksort Externo é um algoritmo que utiliza o método de divisão e conquista para ordenar “*in situ*” um arquivo $A = \{R_1, \dots, R_n\}$ de n registros armazenados consecutivamente em memória secundária de acesso randômico.

Considerando que R_i , onde $1 \leq i \leq n$, é o registro que se encontra na i -ésima posição de A , o primeiro passo do algoritmo é particionar A da seguinte forma: $\{R_1, \dots, R_j\} \leq R_{i+1} \leq R_{i+2} \leq \dots \leq R_{j-2} \leq R_{j-1} \leq \{R_j, \dots, R_n\}$, utilizando para isto um “*buffer*” na memória interna de tamanho $K = j - i + 1$, com $K \geq 3$. Logo após, o algoritmo é chamado recursivamente em cada um dos subarquivos $A_1 = \{R_1, \dots, R_j\}$ e $A_2 = \{R_j, \dots, R_n\}$, sendo que primeiramente é ordenado o subarquivo de menor tamanho.

Esta condição é necessária para que, na média, o número de subarquivos com o processamento adiado não ultrapasse $\log n$.

Subarquivos vazios ou com um único registro são ignorados.

Caso o arquivo de entrada A possua no máximo K registros ele é ordenado em um único passo, ou seja, cada registro do arquivo é lido e escrito uma única vez.

Um pseudocódigo, obtido de *Monard* [Monard, 80], para o Quicksort Externo é exibido abaixo:

```

procedure quicksort_externo(esq, dir);
  { esq e dir são, respectivamente, os endereços
    do primeiro e do último registro de A }
  i, j: integer;

  if dir - esq >= 1 then
    particao(esq, dir, i, j);
    { Ordena primeiro o menor subarquivo }
    if i - esq < dir - j then
      quicksort_externo(esq, i);
      quicksort_externo(j, dir);
    else
      quicksort_externo(j, dir);
      quicksort_externo(esq, i);
    end if
  end if
end procedure

```

LISTAGEM 1: PROCEDIMENTO DO QUICKSORT EXTERNO

Uma questão importante é como são determinados os pontos i e j de partição do arquivo. Os valores das chaves dos registros R_i e R_j são denominados, respectivamente, limite inferior L_{inf} e superior L_{sup} . O objetivo do procedimento **particao** chamado na Listagem 1 é copiar todos os registros menores ou iguais a R_i para A_1 e todos os maiores ou iguais a R_j para A_2 . Inicialmente, $L_{inf} = -\infty$ e $L_{sup} = +\infty$.

A leitura de A é controlada por dois apontadores R_L (“read lower”) e R_U (“read upper”), que apontam, no início, para os extremos esquerdo e direito de A , respectivamente. Da mesma forma, a escrita em A é controlada por W_L (“write lower”) e W_U (“write upper”). Os registros do arquivo A são lidos pelo procedimento **particao**, alternadamente, dos extremos de A até que $K - 1$ registros tenham sido lidos e armazenados no “buffer” interno. A cada leitura no extremo esquerdo, R_L é incrementado de um e a cada leitura no extremo direito, R_U é decrementado de um. O mesmo ocorre com W_L e W_U quando escritas são realizadas no extremo que é controlado por cada um destes apontadores. Para garantir que os apontadores de escrita estejam, pelo menos, um passo atrás dos apontadores de leitura, a ordem alternada de leitura é interrompida se $R_L = W_L$ ou $R_U = W_U$. Somente uma dessas condições é verdadeira em um determinado instante, pois $K \geq 3$. Isto faz com que nenhuma informação seja destruída durante a ordenação do arquivo.

Ao ler o K -ésimo registro, sua chave C é comparada com L_{sup} . Caso seja maior, o valor de W_U é atribuído a j e o registro é escrito em A_2 . Caso contrário, sua chave é comparada com L_{inf} , sendo menor, o valor de W_L é atribuído a i e o registro é escrito em A_1 . Se $L_{inf} \leq C \leq L_{sup}$, o registro é inserido no “buffer”. Neste momento, o “buffer” se encontra cheio e uma decisão deve ser tomada para remover um elemento do mesmo. Esta decisão é tomada levando em consideração o tamanho atual de A_1 e A_2 que são, respectivamente, $T_1 = W_L - esq$ e $T_2 = dir - W_U$. Caso $T_1 < T_2$, o registro R de menor chave C_R é extraído do “buffer” e escrito na posição apontada por W_L em A_1 . Neste momento, o valor de L_{inf} é atualizado com o valor de C_R . Caso contrário, o registro R de maior chave C_R é extraído do “buffer” e escrito na posição apontada por W_U em A_2 . Neste momento, o valor de L_{sup} é atualizado com o valor de C_R . O objetivo de se tomar tal decisão é sempre escrever o registro retirado do “buffer” no arquivo de menor tamanho. Desta forma, o arquivo original A é dividido tão uniformemente quanto possível. Assim, a árvore gerada pelas chamadas recursivas de **quicksort_externo** será mais balanceada, o que, como provado em *Monard* [Monard, 80], é um dos critérios para minimizar a quantidade de operações de E/S efetuadas pelo algoritmo. O outro critério seria aumentar o tamanho do “buffer” interno.

O processo de partição continua até que R_L e R_U se cruzem, ou seja, $R_U < R_L$. Neste instante, existirão $K - 1$ registros no “buffer” interno, os quais devem ser copiados já ordenados para A . Para isto, enquanto existir elementos no “buffer”, o menor deles é extraído e escrito na posição apontada por W_L em A .

Um pseudocódigo para o procedimento **particao**, gerado a partir do apresentado por *Monard* [Monard, 80], é exibido na Listagem 2. Nele, o procedimento **readup** lê em *last_read* o registro apontado por R_U , decrementa R_U de um e atribui “false” a **up_reading**. Da mesma forma, **readlow** lê em *last_read* o registro apontado por R_L , incrementa R_L de um e atribui “true” a **up_reading**. Já o procedimento **writeup(r)** escreve o registro r na posição apontada por W_U e decrementa W_U de um. Semelhantemente, **writelow(r)** escreve o registro r na posição apontada por W_L e incrementa W_L de um. Os procedimentos **extractup** e **extractlow** removem e retornam, respectivamente, o maior e o menor

elemento do “buffer” interno. Além disso, ambos decrementam **no_buffer** de um.

```
procedure particao (esq, dir, i, j)
  RU := WU := dir; {Apontadores superiores de E/S}
  RL := WL := esq; {Apontadores inferiores de E/S}
  no_buffer := 0; {Número de registros no “buffer”}
  Linf := -∞; {Limite inferior de partição}
  Lsup = +∞; {Limite superior de partição}
  up_reading; {flag que faz com que a leitura seja alternada}
  last_read; {contém o último registro lido}
  i := esq - 1;
  {inicia com o subarquivo esquerdo vazio}
  j := dir + 1;
  {inicia com o subarquivo direito vazio}
while RU = RL do
  if no_buffer < K - 1 then
    if up_reading then
      readup()
    else
      readlow();
    end if
    {insere last_read no “buffer” e incrementa no_buffer}
    inserir_buffer();
  else
    if RU = WU then
      readup();
    else
      if RL = WL then
        readlow();
      else
        if up_reading then
          readup()
        else
          readlow();
        end if
      end if
    end if
    if last_read.chave > Lsup then
      j := WU; writeup(last_read);
    else
      if last_read.chave < Linf then
        i := WL; writelow(last_read);
      else
        inserir_buffer();
        if WL - esq < dir - WU then
          r := extractlow(); writelow(r);
        else
          r := extractup(); writeup(r);
        end if
      end if
    end if
  end do
  while WL = WU do
    r := extractlow(); writelow(r);
  end do
end procedure
```

LISTAGEM 2: PROCEDIMENTO QUE REALIZA A PARTIÇÃO DO ARQUIVO A SER ORDENADO

Como os controladores de disco leem e escrevem os dados em blocos, Monard [Monard, 80] também propôs a utilização de “buffers” para leitura e escrita de registros, com o objetivo de diminuir o número de operações de E/S. Isto pode ser feito utilizando dois “buffers” de entrada e dois de saída. Desta forma, cada registro pode ser lido do “buffer” de entrada do extremo esquerdo ou direito do arquivo e escrito no “buffer” de saída do extremo esquerdo ou direito. Quando um “buffer” de leitura de um dos extremos estiver vazio e uma leitura for solicitada ao mesmo, o “buffer” é reabastecido pela leitura de um novo bloco de dados daquele extremo no arquivo original. Uma analogia pode ser feita em relação aos “buffers” de saída, pois quando um destes estiver cheio e uma escrita for solicitada, o “buffer” deve ser descarregado no respectivo extremo do arquivo original. Isto deve ser feito antes do novo registro ser armazenado no “buffer”.

O Quicksort Externo, tal como descrito, foi implementado com o objetivo de verificar a sua eficiência de tempo e espaço em relação aos programas Nitrosort [Nitrosort, 03] e Sort. Monard [Monard, 80] não especificou a forma de gerência do “buffer” interno de tamanho K. Apenas sugeriu algumas estruturas, como arranjo ordenado, “heaps” paralelos e outros. Neste tutorial, a estrutura escolhida para o “buffer” foi uma lista duplamente encadeada implementada em vetor e mantida ordenada, de forma que o maior e o menor registros possam ser removidos do “buffer” com custo $O(1)$. Esta abordagem impõe a sobrecarga de manter a lista ordenada. Porém, testes executados indicaram que tal sobrecarga não compromete o desempenho do Quicksort Externo, desde que o “buffer” não seja muito grande.

Após termos detalhado todas as características do Quicksort Externo, exibiremos agora a sua complexidade, a qual foi demonstrada em Monard [Monard, 80]. A complexidade de melhor caso do algoritmo é

$$O\left(\frac{n}{b}\right) \quad \text{EQUAÇÃO III}$$

que ocorre, por exemplo, quando o arquivo de entrada já se encontra ordenado. A complexidade de pior caso é

$$O\left(\frac{n^2}{K}\right) \quad \text{EQUAÇÃO IV}$$

e ocorre quando um dos arquivos retornados pela rotina de partição tem o maior tamanho possível e o outro é vazio, ou seja, a árvore gerada pelas chamadas recursivas é totalmente degenerada. Monard [Monard, 80] provou que, à medida que n cresce, tanto a probabilidade de ocorrência do pior caso quanto a do melhor tendem a zero, sendo que a probabilidade do melhor caso é sempre maior do que a do pior.

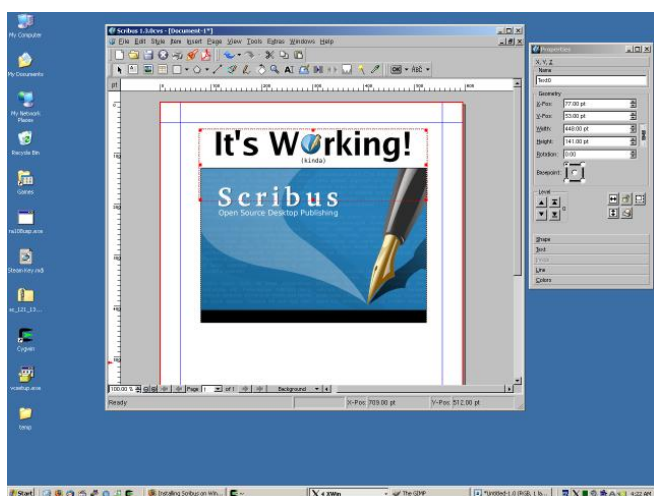
Enfim, em uma demonstração exaustiva, Monard [Monard, 80] provou que a complexidade do caso médio é

$$O\left(\frac{n}{b} \log \frac{n}{K}\right) \quad \text{EQUAÇÃO V}$$

Comparando esta complexidade com a equação (I), apresentada na seção 3.1, percebe-se que na média o Quicksort Externo executa um número pouco maior de operações de E/S do que o exigido no limite inferior do problema de ordenação externa.

3.3. ESTUDOS EMPÍRICOS

Os estudos empíricos foram realizados no ambiente *Cygwin* [Cygwin, 03] em uma máquina equipada com um processador AMD Athlon XP, 2 GHz, 512 MB de memória RAM e com o sistema operacional Windows XP. Entretanto, atualmente podemos rodar Linux sob ambiente VirtualBox, VMWare ou outros virtualizadores. Os arquivos utilizados nos testes continham registros de inteiros de 4 bytes gerados aleatoriamente. Os tamanhos destes arquivos variaram de 1 MB a 1,5 GB, como pode ser visto nas Tabelas 1 e 2.



AMBIENTE CYGWIN RODANDO EM WINDOWS XP

Para possibilitar uma comparação justa, procurou-se por softwares eficientes que realizassem ordenação “*in situ*”, tal como o Quicksort Externo. No entanto, como não se teve acesso a nenhum software com tais

características, foi utilizado nos testes um programa que não ordena arquivos de forma “*in situ*”, mas que é comprovadamente eficiente. O programa escolhido foi o Nitrosort, desenvolvido pela *Cole-Research* [Nitrosort, 03]. O Nitrosort necessita de, no mínimo, o dobro do espaço do arquivo original na memória externa e, portanto, não pode ser classificado como um software de ordenação “*in situ*”. Por ser um software proprietário, não tivemos acesso ao método de ordenação externa utilizado pelo Nitrosort. No entanto, ao entrar em contato com os desenvolvedores do mesmo, estes nos informaram que o método é baseado na ordenação interna e intercalação externa de “*runs*”. Um “*run*” é um segmento do arquivo original que é ordenado na memória interna. Existem vários métodos na literatura baseados neste paradigma, tais como os citados em *Islam* [Islam, 03] e *Leu* [Leu, 00], sendo que o último apresenta um algoritmo ótimo.

Em seguida, tratou-se de delimitar a quantidade de memória interna que seria utilizada pelos algoritmos avaliados. Foram executados testes preliminares, nos quais o Quicksort Externo apresentou um bom desempenho para “*buffers*” capazes de armazenar 100 registros. Como foram utilizados cinco “*buffers*”, a memória total consumida pelo Quicksort Externo foi de 2000 bytes. O Nitrosort permite variar a quantidade de memória interna utilizada, sendo que o valor mínimo suportado é de 1 MB. Para fazer uma comparação justa, procurou-se encontrar a menor quantidade de memória com a qual o Nitrosort apresentasse um bom desempenho. Esta quantidade foi especificada em 3 MB, já que para valores inferiores, o Nitrosort demonstrou ser muito menos eficiente que o Quicksort Externo. O valor ideal, sugerido pelos desenvolvedores do software, é de 21 MB. Neste caso, o Nitrosort chegou a ser duas vezes mais rápido que o Quicksort Externo. Porém, à medida que o arquivo cresce, esta proporção diminui rapidamente e, após um determinado limite, o Quicksort Externo é mais rápido. Esta característica também pode ser observada nos dados obtidos com 3 MB, veja Tabela 1 e Gráfico 1.

Os dados coletados nos testes foram coerentes com a complexidade de caso médio para o Quicksort Externo exibida na seção 3.2. Para averiguar isto, pode-se utilizar a seguinte equação para estimar o tempo de execução a partir da complexidade de caso médio:

$$T_{exe} = \frac{n}{b} \log \frac{n}{K} \times t \quad \text{EQUAÇÃO VI}$$

onde t é uma constante que foi obtida a partir da média de sub-constantes calculadas pela aplicação da equação (VI), utilizando os tempos de execução da coluna **Q. E.** (Quicksort Externo) da Tabela 1. O ideal seria que o valor de t fosse dado pela equação $t = Cte \times t_{1\ op\ E/S}$, onde Cte seria uma constante assintótica e $t_{1\ op\ E/S}$ o tempo de execução de uma operação de E/S na arquitetura onde os testes foram efetuados.

TABELA 1. TEMPOS DE EXECUÇÃO EM SEGUNDOS

Tamanho do Arquivo (MB)	Estimado	Quicksort Externo	Nitrosoft
1	1,07	1,19	1,44
2	2,31	2,23	1,56
4	4,98	4,92	2,33
8	10,68	11,14	3,61
16	22,79	20,05	7,19
32	48,45	49,42	13,59
64	102,64	85,73	26,92
128	216,76	184,61	56,33
256	456,50	412,48	123,30
512	958,94	1029,41	560,45
1024	2009,77	2139,80	2154,42
1536	3095,29	3890,72	4650,08

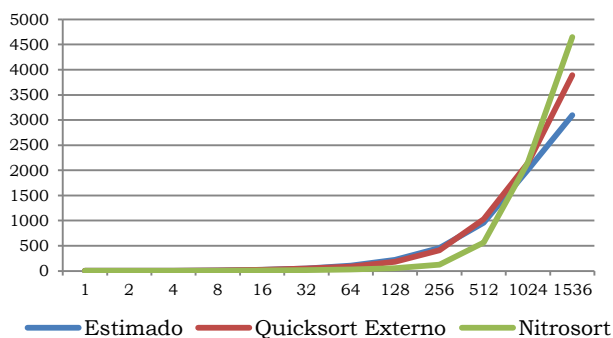


GRÁFICO 1: GRÁFICO DOS DADOS DA TABELA 1

Como mencionado na seção 3.1, o Quicksort Externo foi comparado com o software Sort. Assim como o Nitrosoft, Sort não efetua ordenação “*in situ*”, pois também necessita de memória externa extra para ordenar o arquivo de entrada. Uma peculiaridade do Sort é que ele somente ordena linhas de um arquivo texto. Por isso, os testes com o Sort foram executados separadamente e os dados de entrada, tanto do

Quicksort Externo quanto do Sort, tiveram que ser adaptados de forma que os arquivos possuíssem o mesmo tamanho e o mesmo número de registros. Um registro do arquivo de entrada para o Quicksort Externo corresponde a uma linha do arquivo de entrada para o Sort. Os dados coletados nos testes são exibidos na Tabela 2 e ilustrados no Gráfico 2.

TABELA 2. TEMPOS DE EXECUÇÃO EM SEGUNDOS

Tamanho do Arquivo (MB)	Quicksort Externo	Sort
1	1,17	0,38
2	1,84	1,08
4	4,98	2,19
8	10,00	4,69
16	19,58	9,89
32	41,45	20,50
64	98,83	42,11
128	161,38	84,41
256	425,33	304,38
512	1047,64	980,34
1024	2077,05	2137,23

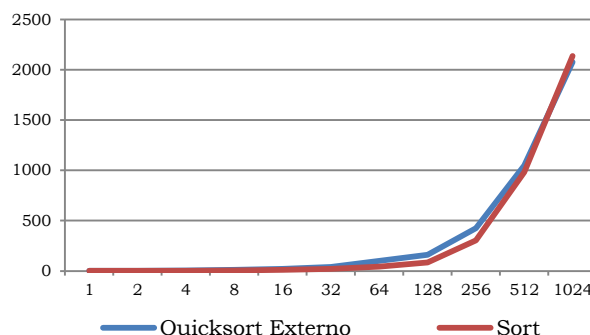


GRÁFICO 2: GRÁFICO DOS DADOS DA TABELA 2

Avaliando os resultados apresentados em termos das métricas propostas na seção 1, percebe-se que a eficiência de tempo do Quicksort Externo está próxima tanto do Nitrosoft quanto do Sort. Sendo que, com o aumento do tamanho da entrada, o Quicksort Externo tende a ser mais eficiente. Em contrapartida, em termos de espaço de memória interna utilizada, o Quicksort Externo é cerca de 1500 vezes mais eficiente que o Nitrosoft. Este valor foi obtido pela razão da memória interna delimitada para o Nitrosoft (3 MB) e a delimitada para o Quicksort Externo (2000 B). Não foi possível fazer a mesma análise para o Sort, pois não foi encontrado o tamanho da memória interna que este utiliza como “buffer” para operações de E/S.

Enfim, foram executados dois testes comparando o Quicksort tradicional em ambiente de memória virtual e o Quicksort Externo. Novamente, o

Quicksort Externo se mostrou mais eficiente com o crescimento do tamanho da entrada, veja Tabela 3.

TABELA 3. TEMPOS DE EXECUÇÃO EM SEGUNDOS

Tamanho do Arquivo (MB)	Quicksort Externo	Quicksort em memória virtual
1024	2139,80	2080,59
1536	3890,72	5191,81

3.4 CONCLUSÕES

Diante do que foi apresentado, pode-se concluir que o Quicksort Externo é um algoritmo muito adequado para ambientes com poucos recursos. Isto porque ele faz ordenação “in situ” de forma eficiente, mesmo com pouca memória interna disponível.

Além disso, seu comportamento, no que diz respeito a tempo de execução, está próximo do comportamento verificado para os softwares utilizados nas comparações.

Tal comportamento era esperado, pois o Quicksort Externo, apesar de possuir uma complexidade de pior caso quadrática, possui uma complexidade de caso médio próxima do número mínimo de operações de E/S necessárias para ordenar externamente um arquivo.

Como proposta para futuros trabalhos, pode-se sugerir, primeiramente, um estudo do impacto no tempo de execução do Quicksort Externo de diferentes estruturas de dados para gerenciamento do “buffer” interno de tamanho K . Além disso, este mesmo estudo poderia determinar o tamanho ideal do “buffer” interno e dos “buffers” de entrada e saída para cada uma das estruturas. Outro estudo poderia ser feito no sentido de tentar melhorar o Quicksort Externo, visando à diminuição do número de operações de E/S realizadas. É importante mencionar que o problema de ordenação externa tem sido muito pesquisado e novas técnicas e algoritmos foram propostos após a publicação do Quicksort Externo. Assim, um estudo poderia buscar a conciliação das vantagens dessas novas técnicas com as apresentadas para o Quicksort Externo em um novo algoritmo.

4 REFERÊNCIAS BIBLIOGRÁFICAS

[Cygwin, 03] Cygwin. **A Linux-like environment for Windows.** Disponível em <<http://www.cygwin.com>>, acessado em abril de 2012.

[Hoare, 62] Hoare, C.A.R. **Quicksort.** The Computer Journal 5, p. 10–15, 1962.

[Islam, 03] Islam, R., Adnan, N., Islam, N., Hossen, S. **A new external sorting algorithm with no additional disk space.** Information Processing Letters 86, p. 229–233, 2003.

[Leu, 00] Leu, F.C., Tsai, Y.T., Yang, C.Y. **An efficient external sorting algorithm.** Information Processing Letters 75, p. 159–163, 2000.

[Monard, 80] Monard, M.C. **Projeto e Análise de Algoritmos de Classificação Externa Baseados na Extratégia de Quicksort.** D. Tese, Pontifícia Univ. Católica, Rio de Janeiro, 1980.

[Nitrosort, 03] Nitrosort. **Software de ordenação externa desenvolvido pela Cole-Research.** Disponível em <<http://www.cole-research.com>>, acessado em abril de 2012.

[Sedgewick, 83] Sedgewick, R. **Algorithms.** Addison-Wesley, Reading, Mass., 1983.

[Verkamo, 87] Verkamo, A.I. **Performance of Quicksort Adapted for Virtual Memory Use.** The Computer Journal, v. 30, n. 4, 1987.

[Ziviani, 93] Ziviani, N. **Projeto de Algoritmos com Implementações em Pascal e C.** ed. Pioneira Thomson Learning, 1993.

5 TERMINAMOS

Terminamos por aqui. Mais em <http://flavioaf.blogspot.com>.

Corra para o próximo tutorial.