**UNIVERSIDADE FEDERAL DE OURO PRETO**

**DEPARTAMENTO DE COMPUTAÇÃO**

**CIÊNCIA DA COMPUTAÇÃO**

**ESTRUTURA DE DADOS II – BCC 203 Data**: 04/02/2016

### PROFESSOR: DR. GUILHERME TAVARES DE ASSIS

**ALUNO:** LINCOLN NONATO DOS SANTOS JÚNIOR

**Atividade**: Este trabalho tem como objetivo um estudo aprofundado da complexidade de desempenho dos métodos de ordenação externa, vistos em sala de aula. Os métodos analisados são: Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição), Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição), Intercalação Polifásica e Quicksort Externo.

**ORDENAÇÃO EXTERNA**

**ANÁLISE DE DESEMPENHO**

OURO PRETO 04 DE FEVEREIRO

2016

ORDENAÇÃO EXTERNA – ANÁLISE DE DESEMPENHO

Trabalho prático apresentado à disciplina Estrutura de Dados II, do curso Ciência da Computação da Universidade Federal de Ouro Preto.

Professor: Dr. Guilherme Tavares de Assis

OURO PRETO 04 DE FEVEREIRO

2016

**Introdução**

Em ciência da computação, um algoritmo de ordenação, de forma geral, é um algoritmo projetado para organizar, ascendente ou descendentente, um dado conjunto de elementos. Para que essa tarefa possa ser realizada com êxito, algoritmos de ordenação fazem uso da memória interna do computador (RAM – Random Access Memory) ou de outros dispositivos de armazenamento, tais como disco rígido e fitas magnéticas. Algoritmos que utilizam a RAM como recurso que auxilia o processo de ordenação, realizam a chamada ordenação em memória interna. Já aqueles que fazem uso de outros dispositivos de armazenamento, como os citados anteriormente, realizam o processo conhecido como ordenação externa. Existem diversos tipos de algoritmos capazes de realizar a tarefa de ordenção e, como cada um deles possui suas características, deve-se utilizar aquele que mais adequa-se à situação.

Como a velocidade de acesso à memória principal é consideravelmente superior à velocidade de acesso a discos rígidos ou fitas magnéticas, algoritmos que realizam ordenação interna são mais rápidos que aqueles que realizam ordenação externa. Porém, infelizmente, nem sempre podemos usufruir de tal velocidade. Caso o conjunto de dados, alvo da ordenação, não possa ser carregado completamente para a memória principal, precisamos de implementar uma técnica capaz de contornar essa limitação. A essas técnicas, damos o nome de algoritmos de ordenação em memória externa, que são os objetos de estudo desse segundo trabalho prático

Em linhas gerais, um algoritmo de ordenação em memória externa, a fim de cumprir sua tarefa, também faz uso da memória principal. A diferença aqui é que ao invés de tentar armazenar todo o conjunto de dados na RAM, o que não seria possível, esses algoritmos trabalham com fragmentos do conjunto. Como esses fragmentos são bem menores que o conjunto de dados original, podem ser armazenados na RAM e ordenação se dá em várias etapas, variando, claro, de acordo com os algoritmos empregados.

Este trabalho apresentará uma análise detalhada da implementação de quatro métodos de ordenação externa, vistos em sala de aula, bem como sua análise de desempenho.

**Descrição do Funcionamento do Programa**

Para atender as especificações deste trabalho prático, um algoritmo foi desenvolvido com um fluxo de execução simples.

A função principal (main) é a responsável por controlar o fluxo do programa e invocar os métodos de ordenação externa implementados.

Assim que a execução se inicia, a primeira tarefa do programa é realizar uma verificação dos argumentos informados via linha de comando, verificando se estes foram informados conforme a especificação do trabalho. Caso a validação não seja aprovada, uma mensagem de erro é retornada ao usuário, informando que os argumentos enviados ao programa são inválidos. Após esta validação, os parâmetros necessários à execução do programa são confiáveis e, com base naquele indica a situação inicial do arquivo (tipo de ordenação: ascendente, descendente ou desordenado aleatoriamente, de acordo com a chave de pesquisa), o programa identifica e prepara o arquivo adequado para o uso. Ou seja, o programan verifica qual dos três arquivos, previamente criados, deve ser alvo da ordenação.

O próximo passo é justamente direcionar o fluxo de execução à função solicitada pelo usuário, através dos parâmetros enviados via linha de comando.

Já que os métodos Intercalação Balanceada de Vários Caminhos (sem uso da técnica de substituição por seleção) e Intercalação Balanceada de Vários Caminhos (com uso da técnica de substituição por seleção) realizam sua tarefa com a ajuda de 10 fitas de entrada e 10 saída, parte das instruções utilizadas para implemetá-los pode ser compartilhada. À função responsável por gerenciar os processos de intercalação (geração dos blocos ordenados e intercalação propriamente dita) é enviado um argumento que indica qual dos métodos de intercalação deve ser realizado. De posse desse argumento, a função decidirá qual das instruções específicas de cada método deve ser chamada.

O método Quicksort Externo foi projetado para realizar uma ordenação *in-place*, que, nesse contexto, significa que a ordenação é realizada no próprio arquivo de entrada, sem uso de fitas magnéticas.

De acordo com a especificação deste trabalho prático, todos os arquivos de saída gerados pelos métodos de ordenação implementados são ordenados de forma descendente. Além disso, as fitas magnéticas foram representadas como arquivos texto.

**Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição)**

O método Interlação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição) faz uso de fitas magnéticas para atingir seu objetivo maior: a ordenação do conjunto de dados.

Inicialmente, os elementos do arquivo de entrada são divididos em blocos. A divisão é regrada pela quantidade de memória interna disponível para o método em questão. Neste trabalho prático, nos limitamos a ter uma área de memória que consiga armazenar 10 elementos, no máximo. Por esse motivo, os blocos gerados possuem, no máximo, 10 elementos.

O processo de geração dos blocos ocorre da seguinte forma. Elementos são lidos, um a um, do arquivo de entrada e inseridos em memória principal. Quando o número de elementos inseridos for igual à capacidade de memória interna disponível para o método, um algortimo de ordenação interna (no caso, o insertion sort) é aplicado nos elementos armazendos na RAM. Uma vez que tais elementos estejam ordenados, um novo bloco passar a existir e é inserido, por completo, em uma fita magnética, identificada por fita de entrada. Nesse momento, a RAM é entendida como vazia e a leitura de elementos do arquivo de entrada continua. Cada novo bloco ordenado gerado é salvo em uma fita magnética diferente. Aqui, há um controle “circular”, com o intuito de evitar que um bloco tente ser inserido em uma fita inválida (a próxima fita após a última é a primeira).

Uma vez que os blocos ordenados tenham sido gerados, tem início o processo responsável pela intercalação desses blocos. Neste trabalho prático, o número de fitas magnéticas disponíveis é compatível com o tamanho da área de memória interna disponível, 20. A ideia desse processo é dividir a quantidade de fitas em duas partes: fitas de entrada e fitas de saída. Ambas as partes possuem a mesma quantidade de fitas.

O processo de intercalação ocorre da seguinte forma. O primeiro elemento de cada uma das fitas de entrada é lido e inserido na área disponível em memória principal. Nesse momento, e mais uma vez, um algoritmos de ordenação interna é aplicado. Com esses dados ordenados, tem início o preenchimento das fitas de saída. O elemento posicionado na primeira posição da área de memória, que, nesse caso, é o elemento com a maior a nota, é retirado (logicamente) da RAM e inserido na fita de saída adequada. O elemento que deixou a RAM deve ser substituído por outro que faça parte daquele mesmo bloco ordenado (mesma fita de entrada). Quando o final de um bloco é alcançado, aquele fita é inativada e, uma vez nesse estado, nenhum outro elemento pode ser lido dela até enquanto existirem outras fitas ativas. Esse processo se repete. No momento em que todas as fitas encontram-se inativadas, todas devem ficar ativas novamente. Esse momento também marca o fim da geração de um novo bloco ordenado. Ou seja, caso as fitas de entrada ainda possuam elementos, estes devem ser inseridas em uma nova fita de saída. .O processo deve se repetir até que não haja mais elementos nas fitas de entrada. Nesse momento, uma etapa da intercalação terá chegado ao fim e novas etapas podem ser necessárias. Caso exista mais de um bloco ordenado nas fitas de saída, todo o processo de intercalação deve se realizado novamente, atentando-se ao seguinte detalhe: fitas de saída passam a ser as novas fitas de entrada. Os dados existentes nas antigas fitas de entrada são apagados.

A alternância entre fitas de entrada e saída é controlada através de um vetor, que representa o conjunto de fitas de entrada (posição 0) e o conjunto de fitas de saída (1), dois ponteiros (um para cada conjunto) e um valor númerico, inteiro, que varia entre 0 e 1. Ao término de cada etapa da intercalação, os endereços apontados pelos ponteiros são trocados, com a ajuda desse valor inteiro.

O processo de ordenação estará concluído quando, após uma etapa da intercalação, apenas um bloco tiver sido gerado.

**Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição)**

Assim como o método anterior, a Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição) faz uso de fitas magnéticas para que o resultado final seja um arquivo ordenado.

A principal diferença entre esse método de ordenação e o anterior está justamente na aplicação da técnica de seleção por substituição. Essa técnica visa a geração de blocos ordenados de tamanhos diferentes.

A técnica de seleção por substituição funciona da seguinte forma. No momento da gerção dos blocos ordenados, os elemetnso são lidos, um a um, do arquivo de entrada, assim como ocorre no método anterior. Porém, quando essa área de memória fica completamente preenchida, seus elementos são ordenados e apenas um deles (no caso, aquele com a nota maior) é removido e inserido numa fita de entrada. Com um espaço vago na área de memória, a programa efetua a leitura de mais um dado do arquivo de entrada e verifica se o dado lido é maior que o último dado escrita na fita de entrada. Caso o dado não seja maior, ele é inserido na RAM normalmente. Caso contrário, apesar de também ser inserido na RAM, vai com o *status* de “marcado”. Itens marcados têm menor prioridade quando comparados a itens “desmarcados”. Então, o processo de leitura de um elemento do arquivo de entrada, verificação se ele é maior que o último escrito na RAM, inserção na RAM (considerando o resultado da verificação) e ordenação dos dados da RAM é realizado até que todos os elementos existentes na memória principal possuam o *status* “marcado”. A partir daí, todos eles recebem o *status* de “desmarcados” e a geração de um novo bloco ordenado, agora pertencente a outra fita de entrada, se inicia.

Já que o controle do comprimento do bloco é feito com base no teste descrito acima, blocos de tamanho variado podem ser gerados. O comprimento dos blocos depende da forma que os elementos estão organizados no arquivo de entrada.

Um recurso que pode ser utilizado para aperfeiçoar a técnica de seleção por substituição é o uso da estrutura de dados chamada *heap*. O *heap* é uma estrutura de dados ideal para atender à necessidade de conhecer o maior (ou menor) elemento de um conjunto. Então, caso a implementação da técnica de seleção por substituição seja feita com uso da estrutura de dados *heap* (que foi o caso desse trabalho prático), ao invés de aplicarmos um método de ordenação, na área de memória interna disponível, deve-se aplicar a regra dos *heaps,* detalhada abaixo*.*

*Heaps* sãoestruturas de dados que trabalham com o conceito das estruturas do tipo árvore binária: nós que possuem dois filhos, um à esquerda e outro à direita. Felizmente, é possível aplicar essa regra não apenas as estruturas do tipo árvore, mas também aos vetores, que é como nossa área de memória é representada no programa.

Em um *heap máximo,* um elemento pai é maior que seus dois filhos. Enquanto que em um *heap mínimo*, um elemento pai é menor que seus dois filhos.

Considerando um vetor cujo índice do primeiro elemento é 0, temos que os filhos à esquerda de um elemento na posição *k* do vetor é o elemento na posição *2k + 1*. Já o filho à direita, deste mesmo elemento *k,* encontra-se na posição *2k + 2* do vetor.

O objetivo é rearranjar os elementos do vetor de tal forma que essa propriedade seja satisfeita para todo elemento que possua filhos. Esse objetivo pode ser alcançado da seguinte maneira:

- Partindo do início do vetor, encontre os filhos de um elemento;

- Compare esses dois elementos e, caso a ideia seja construir um *heap máximo*, que é o caso deste trabalho prático, selecione aquele que for maior;

- Compare o elemento selecionado com seu pai e, caso o pai seja menor, troque-o com esse elemento.

Lembre-se de que a propriedade do *heap* *máximo* não pode ser violada (um pai deve ser maior que seus filhos. Portanto, após trocar pai e filho de posição, poderá ser necessário realizar trocas no sentido oposto (esquerda para direita) a fim de restaurar a propriedade da estrutura de dados. A esse processo, damos o nome de construção do *heap.*

Então, de volta à etapa de geração dos blocos ordenados, caso a técnica de seleção por substituição seja utilizada em conjunto com o *heap,* sempre que um elemento novo for inserido na área de memória, a propriedade do *heap* pode ter sido violada. Por esse motivo, um processo, complementar, denominado reconstrução do *heap,* também deve ser utilizado. Tal processo consiste em um revisita à área de memória principal, com o intuito de realizarcomparações muito semelhantes àquelas realizadas no processo de construção do *heap.* Mas nesse caso, não há necessidade de verificar a necessidade de realizar trocas no sentido oposto.

Note que o processo de reconstrução do *heap* possui complexidade logarítmica. Como a estrutura de dados simula uma árvore, que nesse caso é completa, o número máximo de comparações é igual ao dobro da “altura” do vetor (quantidade de níveis), ou seja, é muito eficiente.

Observação: A técnica de seleção por substituição também por ser utilizada para agilizar o processo de intercalação, mas isso não foi implementado nesse trabalho prático. Durante as etapas de intercalação do método 2, um algoritmo de ordenação interna (insertion sort) foi utilizado.

Observação 2: A implementação desse método faz uso de caracteres separadores para indicar se os elementos estão ou não localizados em um mesmo bloco ordenado. Se entre dos elementos existir um caracter “;”, os elementos são considerados parte de um mesmo blobo. Caso o caracter entre os elementos for o “@”, os elementos pertencem a blocos diferentes. Então, basicamente, esse controle foi usado no lugar de uma variável que armazena a quantidade de itens que forma cada um dos blocos ordenados de uma fita.

**Intercalação Polifásica**

Este método de intercalação faz uso de f + 1 fitas para resolver o problema de ordenação proposto. Assim como no método 2, a Intercalação Polifásica foi implementado com o uso da técnica de seleção por substituição. Este método veio para amenizar uma problema existente no método Intercalação Balanceada de Vários Caminhos, quando esta utiliza f + 1 fitas (método não analisado nesse trabalho prático). Nesse caso, como só há uma fita de saída, após cada etapa do processo de intercalação os blocos inseridos na fita de saída devem ser redistribuídos para as fitas de entrada. Por este motivo, a cada etapa da intercalação dos blocos, uma nova passada pelo arquivo é necessária, gerando operações de leitura/escrita que poderiam ser evitadas.

O método Intercalação Polifásica elimina este problema introduzindo o conceito de que os blocos ordenados gerados são distribuídos, desigualmente, entre as fitas de entrada disponíveis e que a fita de saída nem sempre é a mesma.

Para atingir tal objetivo, pode-se usar, por exemplo, a fita de destino de um bloco ordenado pode ser identificada a partir da geração de números aleatórios (cada número representa uma fita). A intenção é que a quantidade de blocos seja diferente entre as fitas de entrada, pois isso faria com que fitas ficassem vazias em momentos diferentes.

Após cada etapa da da intercalação, é verificado se existe alguma fita vazia. Caso essa fita exista, a próxima etapa da intercalação utilizará essa fita como fita de saída (antiga fita de saída passa a se comportar como fita de entrada).

Observação 1: A implementação desse método faz uso de caracteres separadores para indicar se os elementos estão ou não localizados em um mesmo bloco ordenado. Se entre dos elementos existir um caracter “;”, os elementos são considerados parte de um mesmo blobo. Caso o caracter entre os elementos for o “@”, os elementos pertencem a blocos diferentes. Então, basicamente, esse controle foi usado no lugar de uma variável que armazena a quantidade de itens que forma cada um dos blocos ordenados de uma fita.

Observação 2: Há uma implementação parcial deste método no código fonte. O processo de geração dos blocos ordenados, aparentemente, foi concebido de maneira correta, mas a intercalação não está funcionando corretamente. Por este motivo, não foi possível incluir as estatísticas e o análise do método.

**Quicksort Externo**

Proposto por Monard (1980), é um algoritmo que usa o paradigma da divisão e conquista para realizar a ordenação, *in place*, de um conjunto de dados localizados em um arquivo.

Além de não fazer uso de fitas magnéticas, como os demais métodos de ordenação externa analisados, o tamanho da área de memória utilizada pelo Quicksort Externo varia de acordo com a quantidade de elementos que devem ser ordenados (o tamanho da área de memória é dado pelo logarítmo de N, na base 2 – onde N indica a quantidade de registros).

Diferentemente de sua versão para memória interna, o elemento considerado pivô do conjunto agora não é um único elemento apenas. Os pivôs do Quicksort Externo são justamente os elementos que se encontram na área de memória interna disponível para o método.

Existem também duas variáveis de controle que indicam os limites númericos dos valores localizados no pivô. Tais variáveis são conhecidas como “limite inferior” e “limite superior”.

A ideia básica do algoritmo ainda é a mesma utilizada em sua versão para memória interna. O processo de leitura dos elementos parte das duas “extremidades” do conjunto de dados a ser ordenado e os elementos são lidos de forma alternada. A cada leitura, os elementos são inseridos na área de memória disponível para o método, caso esta possua mais de 1 espaço disponível. Caso contrário, algumas verificações devem ser feitas antes para que o destino do item lido possa ser determinado. Se o elemento lido é maior ou igual ao limite superior e menor ou igual ao limite inferior, ele é inserido na última posição disponível na área de memória (a comparação aqui é, levemente, diferente daquela vista em sala de aula, pois o arquivo deve ser ordenado descendentemente). Se o elemento lido é menor que o limite inferior, ele é escrito diretamente na partição 2 do arquivo de entrada, na posição adequada (mais uma vez, aqui, a partição 2 representa os itens menores que o pivô, já que a ordenação é descendente). Se o elemento lido é maior que o limite superior, ele é escrito na partição 1 do arquivo original. Quando a área de memória encontra-se cheia, é necessário remover um desses elementos. A escolha do elemento que será removido é feita com base do tamanho atual das partições lógicas do arquivo. A intenção aqui é que o processo de partição gere partições balanceadas (quantidade de registros semelhante). Logo, caso a partição que abriga itens menores que o pivô esteja com um tamanho menor que a outra partição, o elemento que será removido da área de memória será o de menor valor.Caso contrário, o item removido será o de maior valor. Dessa forma, espera-se que as partições cresçam de maneira parecida. Esse processo continua até que os ponteiros que realizam a leitura das duas extremidades se cruzem. Ao final, o arquivo original é dividido, logicamente, em duas partições e o método Quicksort Externo é aplicado, recursivamente, a cada uma delas.

**Descrição da Metodologia de Análise**

Após a etapa de implementação dos métodos de ordenação externa, descritos na sessão anterior, serão realizados testes de desempenho para que se possa comparar a eficácia de cada método. Os testes prosseguirão seguindo os seguintes critérios:

1. Cada método de ordenação externa será executado 1 vez (já que a ordenação será realizada em um mesmo arquivo de entrada, o número de operações de leitura/escrita e de comparações será o mesmo para todas as tentativas);
2. Cada método realizará a ordenação dos X primeiros elementos de um arquivo texto, onde X representa os seguintes números: 200, 2000, 20000, 200000, 471705;
3. Cada método realizará a ordenção utilizando como base cada uma das três situações inicias de organização do arquivo de entrada: desordenado (original), ordenado crescentemente e ordenado decrescentemente;
4. Para fins de análise comparativa do desempenho dos métodos, será calculado o tempo médio e o número de comparações médias de cada método em cada cenário de problema, tornando o entendimento dos resultados mais claros e objetivos.

A unidade utilizada para mensurar tempo gasto pelos métodos de ordenção será segundos(s). Após a realização dos testes acima referidos foram registrados os seguintes dados:

**Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição) – Ordenação Crescente**

****

****

****

****



**Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição) – Ordenação Decrescente**











**Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição) – Desordenado aleatoriamente**











**Análise dos resultados – Intercalação Balanceada de Vários Caminhos (sem uso da técnica de seleção por substituição)**

A eficiência desse método está vinculado à quantidade de fitas magnéticas utilizadas por ele. Quanto maior for o número de fitas, menor será a quantidade de intercalações de blocos ordenados.

**Número de transferências (leitura)**: A media que a quantidade de registros a ser ordenada cresce, o número de operações de leitura também aumenta, que é o comportamento esperado. A quantidade de operações de leitura é ditada pela quantidade de etapas de intercalação realizadas pelo método. Quanto maior for o número de intercalações necessárias para que o arquivo seja ordenado com sucesso, maior será o número de operações de leitura.

**Número de transferências (escrita)**: A media que a quantidade de registros a ser ordenada cresce, o número de operações de escrita também aumenta, que é o comportamento esperado. A quantidade de operações de escrita é ditada pela quantidade de etapas de intercalação realizadas pelo método. Quanto maior for o número de intercalações necessárias para que o arquivo seja ordenado com sucesso, maior será o número de operações de escrita.

**Número de comparações de chave:** O número de comparações realizadas varia de acordo com a situação inicial do arquivo de entrada. Nota-se que quando o número de registros a serem ordenados está dentro da faixa 200 – 20000, o número de comparações realizadas quando o arquivo de entrada encontra-se, inicialmente, ordenado crescentemente e decrescentemente é o mesmo. Isso ocorre por que, dentro desse intervalo, todos os elementos do arquivo de entrada ordenado de forma crescente são iguais. Como o método de ordenação interna escolhido é o insertion sort, esse caso se encaixa no melhor caso, onde o número de comparações é linear.

**Tempo de execução**: Como as operações de entra e saída são possuem um custo computacional maior, quando comparadas as operações de comparação de chave, o tempo de execução varia de acordo com quantidade total de transferência (leitura/escrita) realizadas pelo método. Quando maior for o número de transferências, maior será o tempo gasto para que o método termine sua execução.

**Observação**: O número de operações e escrita e leitura são iguais devido à natureza do método. Cada registro lido de uma fita de entrada deve ser escrito em uma fita de saída.

**Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição) – Ordenação Crescente**











**Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição) – Ordenação Decrescente**











**Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição) – Desordenado Aleatoriamente**











**Análise dos resultados – Intercalação Balanceada de Vários Caminhos (com uso da técnica de seleção por substituição)**

Como os blocos ordenados iniciais podem ter comprimento variável, a eficiência desse método, além de relacionada à quantidade de fitas magnéticas utilizadas por ele, está vinculaao à situação inicial do arquivo a ser ordenado.

**Número de transferências (leitura)**: A media que a quantidade de registros a ser ordenada cresce, o número de operações de leitura também aumenta, que é o comportamento esperado. A quantidade de operações de leitura é ditada pela quantidade de etapas de intercalação realizadas pelo método. Quanto maior for o número de intercalações necessárias para que o arquivo seja ordenado com sucesso, maior será o número de operações de leitura.

**Número de transferências (escrita)**: A media que a quantidade de registros a ser ordenada cresce, o número de operações de escrita também aumenta, que é o comportamento esperado. A quantidade de operações de escrita é ditada pela quantidade de etapas de intercalação realizadas pelo método. Quanto maior for o número de intercalações necessárias para que o arquivo seja ordenado com sucesso, maior será o número de operações de escrita.

**Número de comparações de chave:** O número de comparações realizadas varia de acordo com a situação inicial do arquivo de entrada. O algoritmo usado para realizar a ordenação interna, durante as etapas da intercalação, foi o insertion sort.

**Tempo de execução**: Como as operações de entra e saída são possuem um custo computacional maior, quando comparadas as operações de comparação de chave, o tempo de execução varia de acordo com quantidade total de transferência (leitura/escrita) realizadas pelo método. Quando maior for o número de transferências, maior será o tempo gasto para que o método termine sua execução. Nesse ponto, vale observar que o fato de a estrutura de dados heap ter sido usada para obtenção do maior elemento (etapa de geração dos blocos ordenados apenas) reduziu o tempo de execução do método.

**Observação**: Quando comparados ao método anterior, o número de operações e escrita e leitura são bem menore, dentro do intervalo 200 – 20000. Isso ocorre por que, dentro desse intervalo, todos os elementos dos arquivos de entrada, inicialmente ordenados de forma crescente ou decrescente, já estão ordenados (no caso do arquivo ordenado de forma crescente, os elementos são iguais). Por este motivo, o processo de geração dos primeiros blocos ordenados cria blocos muito grandes, reduzindo o número de comparações.

**Quicksort Externo – Ordenação Crescente**











**Quicksort Externo – Ordenação Decrescente**











**Quicksort Externo – Desordenado Aleatoriamente**











**Análise dos resultados – Quicksort Externo**

Por realizar uma ordenação *in place*, este método é indicado quando não há dispositivos auxiliares disponíveis para a realização do processo de ordenação externa.

**Número de transferências (leitura)**: A quantidade de operações de leitura cresce de acordo com o número de partições gerados após cada aplicação do algoritmo.

**Número de transferências (escrita)**: A quantidade de operações de escrita cresce de acordo com o número de partições gerados após cada aplicação do algoritmo.

**Número de comparações de chave:** O número de comparações realizadas varia de acordo com a situação inicial do arquivo de entrada, que indicará dos casos (pior, médio, melhor) foi atingido. Percebe-se que quando a situação inicial do arquivo é desordenado aleatoriamente, o número de comparações de chave é menor. Fato que, até então, me parece estranho, uma vez que isso deveria acontecer quando a situação inicial do arquivo fosse ordenado descendentemente (objetivo do algoritmo). Não sei se é o caso aqui, mas isso pode ter sido causado pelo fato deste algoritmo não ser estável.

**Tempo de execução**: Como as operações de entra e saída são possuem um custo computacional maior, quando comparadas as operações de comparação de chave, o tempo de execução varia de acordo com quantidade total de transferência (leitura/escrita) realizadas pelo método. Quando maior for o número de transferências, maior será o tempo gasto para que o método termine sua execução. De acordo com os resultados obtidos, o método Quicksort Externo foi o pior. Imagino que isso pode ter sido causado pela abordagem utilizada por ele, que é recursiva, diferentemente dos demais métodos. O tempo gasto para a criação dos *frames* na pilha de alocação também pode ter contribuído para esse comportamento. Porém, creio que o motivo principal aqui seja realmente o elevado número de transferências realizadas.

**Observação 1**: A quantidade operações de leitura e escrita são sempre iguais, pois todo elemento lido do arquivo de entrada deve ser escrito em alguma das duas partições.

**Observação 2:** O algoritmo deste método foi implementado com base na teoria apresentada em sala de aula, pelo professor. Optei por não utilizar o código fornecido (slides) para que o entendimento do método fosse ainda maior.

**Observação 3**: Como o arquivo disponibilizado (PROVAO.TXT) está no formato texto, imaginei que todo o processo de implementação deveria realizar manipulação de arquivos no formato texto. Então, para que o algoritmo consiga alcançar seu objetivo, a ordenação do arquivo, faz uso da função fseek(). De acordo com a documentação da linguagem C, tal função, por não ser portável, não é confiável quando usada em conjunto com arquivos texto. Em conversa com o professor, durante aula de dúvidas, foi combinado que o uso da função fseek() poderia ser mantido, já que o algoritmo já estava funcionando. Porém, devido a instabilidade da função, fui orientado a incluir uma observação à respeito da versão do compilador e S.O. utilizados no desenvolvimento e teste deste algoritmo. Poranto, deixo aqui essas informações:

- Compilador: GCC

- Versão: 4.9.2

- Sistema Operacional: Debian 8

**Principais Dificuldades Encontradas**

* Perceber que o arquivo disponibilizado (PROVAO.txt) utiliza quebra de linha no formato Windows, representado por dois caracteres (\r\n) e não apenas um. Como todo o processo de desenvolvimento se deu em ambiente Linux e o arquivo foi manipulado em sua forma original (formato texto), essa realmente foi uma grande dificuldade.
* Trabalhar com diversos ponteiros para arquivos;
* Manipulação correta das cadeias de caracteres, que apesar de um assunto recorrente, deve ser encarado com bastante atenção, pois os erros gerados por uma manipulação errada (invasão de memória) são, não raramente, altamente complexos de serem identificados. Erros deste tipo consomem muito tempo;
* Cria uma estratégia para trabalhar com blocos ordenados de tamanhos diferentes (métodos 2 e 3);
* Implementar o método Quicksort Externo.

**Conclusão**

Após a análise dos resultados obtidos, foi concluído que não há um método ideal para todos as situações. Devido as características dos métodos, cada um deles pode ter sua eficiente atingida se o cenário certo for levado em consideração.

O método 1, Intercalação Balanceado de Vários Caminhos (sem uso da técnica de seleção por substituição) é um método eficiente quando dispositivos de armazenamentos temporário podem ser usados, pois a eficiência deste método depende disso.

O método 2, Intercalação Balanceado de Vários Caminhos (com uso da técnica de seleção por substituição) apresenta um ganho significativo quando ao método 1. Além de também se adequar melhor a ambientes que possuam recursos de armazenamento temporário, a própria natureza do método 2 pode fazer com que o resultado seja alcançado de maneira mais eficiente. O fator que determinará tal condição será a ordem inicial do arquivo a ser ordenada.

O método 3, apesar de ter se mostrado o mais lento dos 3, se destaca por não fazer uso de dispositivos temporários (extras). Portanto, tal método é o ideal quando a necessidade de se ter uma ordenação *in place* é essencial*.*

**Referências:**

SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*. 4 ed. Westford: Addison-Wesley, 2011.

ZIVIANI, Nívio. *Projeto de Algoritmos:* com implementações em PASCAL e C. 3 ed. São Paulo: Cengage Learning, 2012.