UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
MESTRADO EM INFORMÁTICA
PROJETO E ANÁLISE DE ALGORITMOS

UFAM

TRABALHO PRÁTICO 1
ORDENAÇÃO EXTERNA POR INTERCALAÇÃO MULTIVIAS

Manaus, 2016

UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
MESTRADO EM INFORMÁTICA
PROJETO E ANÁLISE DE ALGORITMOS

UFAM

TRABALHO PRÁTICO 1
ORDENAÇÃO EXTERNA POR INTERCALAÇÃO MULTIVIAS

Aluna: Bruna Afonso Schramm

Prof.: Dr. Altigran Soares

Disciplina: Projeto e Análise de Algoritmos

Manaus, 2016

Questão 1. Descreva em alto nível um algoritmo de ordenação externa por intercalação (Mergesort) multivias para ordenar um arquivo de dados fornecido como entrada.

Algoritmo: Mergesort multivias
  Entrada:
    Arquivo de entrada: arquivo binário desordenado de tamanho N (entrada)
    Arquivo de saída: arquivo contendo os N valores da entrada ordenados (saida)
    Memória disponível: máximo de memória em bytes que o programa pode carregar (M)
    Número de vias: quantidade de vias utilizadas no merge (K)

  Saída:
    Arquivo: arquivo "saida" contendo os N valores da entrada ordenados

msort():

```
  maximoValores = M / 4      # quantidade valores que pode ser carregada
  arquivosParciais = []

  #Etapa 1 - Criação dos subarquivos ordenados - total de teto ( n / m )
  contadorParcial = 0

  Enquanto entrada != vazio:
    valores = []

    # carrega o máximo de valores suportados na memória
    valores = carregarValores (entrada, maximoValores)

    # Ordena os valores carregados em memória
    ordenarValores(valores)

    # salva o novo arquivo parcial
    salvarArquivoParcial(valores, contadorParcial, arquivosParciais)

    # incrementa o identificador do arquivo parcial
    contadorParcial++

  # Etapa 2 - Merge do arquivos parcias
  mergeParciais( arquivosParciais, k, saida, etapa )
```

```
mergeParciais ( arquivosParciais, k, saida, etapa ):

  Se arquivosParciais.tamanho == 1:
    #Finalizou programa!
    salvarSaida(saida, arquivosParciais)

  Senão:
    novosArquivos = []

    indice = 0

    Enquanto arquivosParciais != vazio:

      novoArquivo = mergeKarquivos( arquivosParciais, k, indice, etapa )

      novosArquivos.adicionar(novoArquivo)

    etapa += 1

    mergeParciais ( novosArquivos, k, saida, etapa )
```

mergeKarquivos(arquivos, k, indice, etapa):

```
  totalArquivos = arquivos.tamanho

  heap = []

  # Quantidade de arquivos mergeados, será 1 < arquivosCarregados <= k
  arquivosCarregados = 0

  Para cada arq em arquivos a partir do indice:
    Se arquivosCarregados < k:
      # Carrega o primeiro elemento do arquivo para o heap
      carregaElemento( heap, arq )
    Senão:
      arquivosCarregados += 1

  # Atualiza o indice do ultimo arquivo carregado
  indice += arquivosCarregados

  # Constroi heap de minima
  heap.controiHeap()
```

```
  # Cria novo arquivo
  novoArquivo = novo Arquivo(etapa, indice)

  Enquanto heap != vazio:

    menor = heap.extrairMin()

    salvaValorSaida(novoArquivo, menor.valor)

    # Verifica se ainda tem algum elemento do arquivo para ler
    Se menor.arquivo != vazio:
      carregaElemento( heap, menor.arquivo )

  retorna novoArquivo
```

Questão 2. Apresente a complexidade de tempo do seu algoritmo considerando o ambiente de execução em memória secundária.

Dado que M = memoriaDisponível / 4, representa a quantidade de elementos lidas na memória.

N = total de elementos do arquivo de entrada

Complexidade da etapa 1 - Geração dos arquivos parciais ordenados se dá por:

- Custo de ler todos os elementos: N * C1
- Custo de ordenar os blocos de valores de tamanho M: (N/M log M) * C2
- Custo de salvar os blocos em memória secundária: N * C3

Custo da primeira etapa = N(C1 + C3) + (N/M log M) * C2

Complexidade da etapa 2 - Merge dos arquivos parciais é dada por:

Questão 3. Baseado em seu algoritmo, implemente um programa para ordenar arquivos cujos registros são inteiros de 32 bits. O programa resultante (msort) deverá receber os seguintes parâmetros: arquivo de entrada – arquivo que contém dos dados a serem ordenados; arquivo de saída – arquivo que contém os dados ordenados; memória – a quantidade total de memória disponível

para a ordenação. O programa não deve alocar mais memória do que o especificado neste parâmetro; k: número de vias usadas pelo Mergesort.

Implementação do programa msort detalhada na questão 4.

Questão 4. Apresente um relatório de sua implementação, considerando as "Recomendações para Elaboração e Documento de Trabalhos de Implementação" disponíveis no blog da Disciplina.

Descrição

O problema de ordenação de elementos é um dos problemas mais fundamentais da computação, porque é utilizada como subrotina para solução de diversos outros problemas. Exitem diversos algoritmos para fazer isso, no caso em particular tratado por esse trabalho será demonstrado um algoritmo para ordenação de grandes volumes elementos, que não cabem completamento na memória principal do computador.

Se trata do algoritmo de ordenação externa por intercalação multivias, esse algoritmo quebra o arquivo original, composto por valores (a1, a2, … an) desordenados, em partes menores que cabem na memória, ordena esses pedaços e os salva em arquivos auxiliares.

Em seguida, aos poucos esses arquivos são mergeados, formando arquivos cada vez maiores, até que se tenha um único arquivo final. Esse arquivo será uma permutação do arquivo entrada, de forma que: a'1 < a'2 < … < a'n.

Um ponto a se observar nesse algoritmo é que o custo de acessar os dados em memória secundária é significativamente maior que o custo de comparação dos itens em memória principal, então deve a leitura/escrita dos elementos em memória secundária deve ser evitada ao máximo.

Módulos e estruturas de dados utilizados

O programa de ordenação é composto por 2 classes ExternalSorter e HeapOfPair.

1. ExternalSorter

A primeira a ser descrita é a classe principal do programa, chamada de ExternalSorter. Esta é responsável pela execução dos passos do algoritmo de ordenação externa. Contém os métodos:

- **validaParamentros**: Este método é responsável por algumas validações sobre os parâmetros passados na execução do programa:
    - Verifica se o arquivo de entrada existe realmente, senão existe encerra o programa.
    - Verifica se com a memória informada é possível executar o programa usando a quantidade de vias especificada, senão for possível encerra o programa.

- **gerarArquivosBase**: Este método é responsável por dividir o arquivo de entrada em pequenos arquivos ordenados. São carregados M valores por vez na memória, ordenados e salvos em um novo arquivo parcial, isso se repete até que todos os valores do arquivo de entrada tenha sido lidos.

    Cada arquivo parcial tem no máximo tamanho M, representa a quantidade máxima de valores suportados na memória, é definido pelo cálculo (memória disponível / 4).

    Ao final desse método serão gerados P = **teto**(N / M) arquivos parciais, onde N é a quantidade de valores total da entrada.

- **mergeArquivosOrdenados**: Este método é responsável por juntar (fazer o merge) os arquivos parciais gerados pelo método anterior e formar o arquivo final ordenado contendo todos os N valores da entrada, ordenados de forma crescente.

    Esse procedimento é feito por etapas, a cada etapa são mergeados de k em k arquivos parciais resultando em novos arquivos parciais, a partir desses novos arquivos é iniciada uma nova etapa merge, isso é repetido até que se tenha somente 1 arquivo ordenado com todos os valores, esse será o arquivo final do programa. O merge desses arquivos é feito utilizando a classe HeapOfPair que será descrita a seguir.

    Esse processo é ilustrado na imagem 1, supondo que k = 2 e o número de arquivos parciais = 8, então seriam necessárias 3 etapas de merge para se obter o arquivo final ordenado com N valores de entrada.

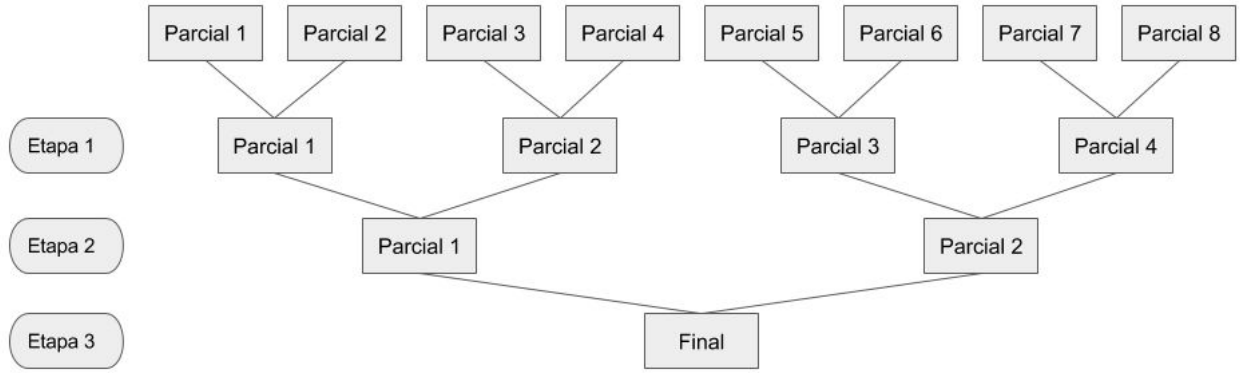

Imagem 1 - Exemplo da etapa de merge dos arquivos parciais

2. HeapOfPair

A segunda classe é a classe HeapOfPair, ela consiste da adaptação implementação de um heap de mínima baseado no algoritmo descrito no livro Algoritmos. Teoria e Prática do Thomas H. Cormen.

Nessa implementação cada nó do heap, é composto por uma estrutura *pair* do c++ onde o primeiro elemento é o valor a ser ordenado e o segundo é o ponteiro do arquivo parcial ao qual esse elemento pertence.

Essa classe é utilizada na segunda etapa do algoritmo do merge sort externo, dentro do método mergeArquivosOrdenados da classe ExternalSorter. Ela é utilizada para ordenar e manter ordenados os valores que são lidos dos arquivos parciais enquanto estes são mergeados.

Uma adaptação feita nessa estrutura foi a criação dos métodos:

- **consultaMin**: método retorna o elemento do topo do heap.
- **adicionaValorInicio**: método substitui o elemento do topo do heap pelo valor passado por parâmetro e então heapifica a partir do topo.

Esses métodos foram criados com objetivo de otimizar o processo de merge dos arquivos, na implementação original sempre que o elemento do topo é removido, o ultimo elemento do heap é colocado como no topo e então é feita a heapificação a partir dele. Em seguida quando um novo elemento é inserido no fim do heap, é necessário fazer outra heapificação.

Como essas operações de extrairMinimo e adicionarValor se repetem diversas vezes e sempre seguem essa ordem foram implementados os 2 métodos acima. O processo ficou assim, o primeiro elemento é consultado, sem ser extraido do heap, se houver mais elemento a ser inserido no heap será inserido no início do heap e então a

heapificação é feita somente 1 vez. Se não houver mais elemento a ser inserido no heap, então é chamada a traducional função extrair minimo.

Formato de Entrada dos Dados

O programa de ordenação de externa irá receber um arquivo binário composto por números inteiros de 4 bytes em ordem aleatória. Esses são os valores que devem ser ordenados pelo programa do menor para o maior.

Mais abaixo, será detalhada a maneira de gerar esse arquivo entrada utilizando o programa inputGenerator.cpp.

Formato de Saída dos Dados

O programa de ordenação externa ao final da execução gerará um arquivo binário contendo números inteiros de 4 bytes, estes representam os valores obtidos do arquivo de entrada que foram ordenados de forma crescente pelo programa.

Mais abaixo será detalhada maneira de validar se o arquivo de saida está ordenado corretamente, utilizando o programa validador.cpp.

Execução do Programa

O programa encontra-se no arquivo compactado de nome tpa1-brunaschramm.tar.gz, para descompactá-lo usar o comando: tar -xvf tp1-brunaschramm.tar.gz.

Após descompactá-lo, dentro da pasta **external-sorter** estarão os códigos a serem utilizados.

Inicialmente, antes de executar o programa de ordenação externa é preciso preparar o arquivo de entrada usado por ele, para isso basta seguir os seguintes passos:

1. Compilar o código do programa que gera entradas aleatórias, através do comando: g++ inputGenerator.cpp -o inputgen
2. Em seguida, executá-lo usando o comando: ./inputgen <tamanho-em-gb>
    a. **inputgen**: é o nome do binário gerado na compilação (passo 1)
    b. **Tamanho**: representa o tamanho que o arquivo deve ter, esse tamanho é dado em gigabytes.

Ao final da execução desse programa será gerado um arquivo de nome **input.bin**, formado por inteiros de 4 bytes em ordem aleatória.

Finalmente, para executar o programa de ordenação externa por intercalação são necessários os seguintes passos:

3. Compilar o programa através do comando: g++ msort.cpp -o msort
4. Depois executá-lo usando o comando: ./msort input.bin <arquivo-saida> <memoria-em-bytes> <numero-vias>
    a. **msort**: é o nome do binário gerado na compilação do programa (passo 3).
    b. **input.bin**: é o nome do arquivo de entrada a ser ordenado pelo programa. É o arquivo binário, formado por inteiros gerados em ordem aleatória de 4 bytes gerado no passo 2.
    c. **Arquivo de saída**: nome do arquivo onde os dados ordenados devem ser salvos ao final da execução do programa. Será um arquivo binário formado por inteiros de 4 bytes.
    d. **Memória**: representa a quantidade de memória disponível pelo programa, para realizar a ordenação dos elementos.
    e. **Vias**: representa o número de vias que o programa deverá usar para fazer a intercalação dos arquivos parciais gerados para a construção do arquivo final ordenado.

Ao final da execução desse programa será gerado o arquivo com a permutação dos valores do arquivo de entrada, de forma que a'1 <= a'2 <= … <= a'n. Para validar se os valores gerados no arquivo de saída estão realmente em ordem crescente, basta seguir os seguintes passos:

5. Compilar o código do programa que valida se um arquivo binário composto por inteiros de 4 bytes está em ordem crescente, através do comando: g++ validador.cpp -o validador
6. Executá-lo da seguinte forma: ./validador <arquivo-binario>
    a. Arquivo: representa o arquivo gerado pelo programa de ordenação externa no passo 4.

Se o arquivo esteja ordenado corretamente, será exibida a mensagem “Arquivo validado com sucesso!”. Caso contrário, será exibida a mensagem: “Arquivo invalido :(”.

Questão 5. Usando o programa implementado, execute e apresente gráficos dos seguintes experimentos de desempenho (tempo de execução). Para os experimentos, devem ser gerados arquivos não ordenados cujos registros são inteiros de 32 bits. Devem ser usados arquivos tamanhos representativos, ou seja, de pelo menos 4 GBytes.

a) Fixando os parâmetros memória e k, verifique o tempo de execução para diferentes tamanhos de arquivo.

Para essa questão foram fixa dos os valores:

- Memória disponível: 1 Gb
- Quantidade de vias: 8

A tabela 1 mostra a variação dos valores:

Tamanho Arquivo	Tempo de Execução (ms)
4	482753
6	1055467
8	1358398
16	3589711
32	6812580

Tabela 01. Tempo em milissegundos para diferentes tamanhos de arquivo.

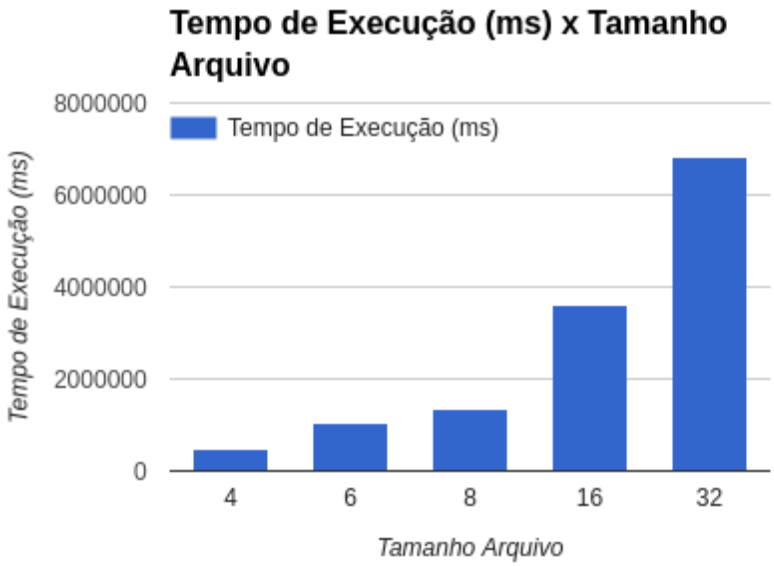

Imagem 02. Gráfico da variação do tempo de execução em função do tamanho da entrada

b) Fixando o parâmetro k e usando um mesmo arquivo, apresente a performance (em milissegundos) do programa implementado para diversos valores de capacidade de memória.

Para essa questão foram fixa dos os valores:

- Tamanho do arquivo: 4 Gb

- Quantidade de vias: 8

A tabela 2 mostra a variação dos valores:

Memória	Tempo de Execução (ms)
128 mb	627678
512 mb	522044
1 gb	483753
2 gb	482924
4gb	360097

Tabela 02. Tempo em milissegundos para diferentes quantidades de memória disponíveis.

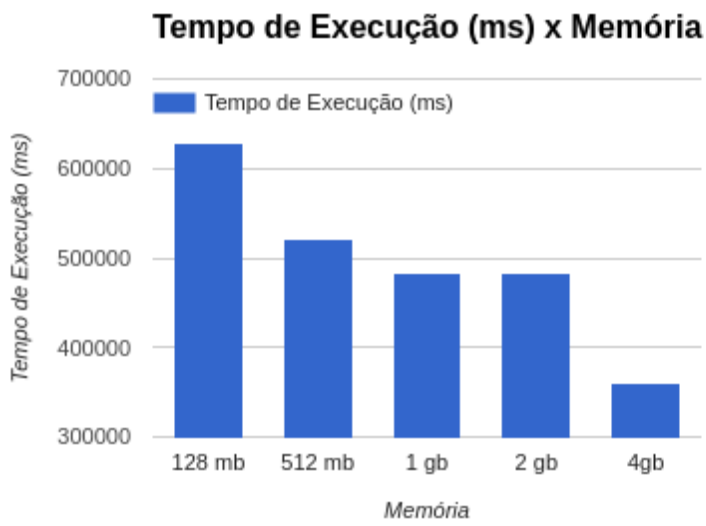

Imagem 03. Gráfico da variação do tempo de execução em função da memória utilizada

c) Fixando o parâmetro memória e usando um mesmo arquivo, apresente a performance (em milissegundos) do programa implementado para diversos valores de k.

Para essa questão foram fixa dos os valores:

- Tamanho do arquivo: 4 Gb
- Memória disponível: 1 Gb

A tabela 3 mostra a variação dos valores:

Vias	Tempo de Execução (ms)
2	699687
4	631291
8	522044
16	518466
32	506481

Tabela 03. Tempo em milissegundos para diferentes quantidades de vias utilizadas.

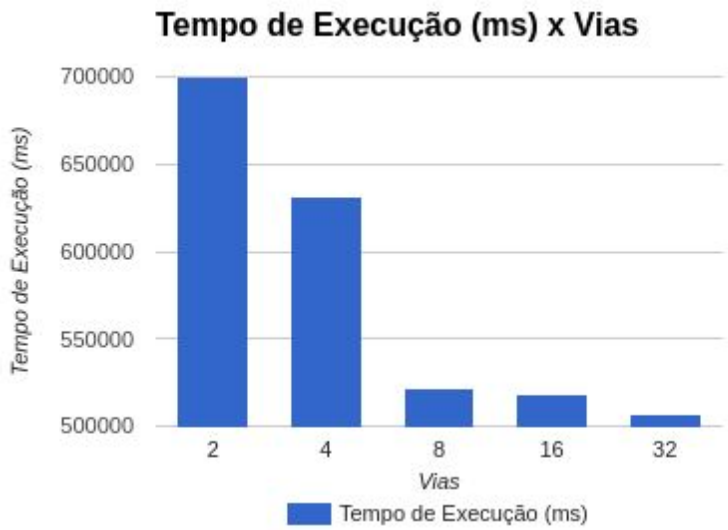

Imagem 04. Gráfico da variação do tempo de execução em quantidade de vias utilizadas