Trabalho Prático 03, AEDS3

Giulia M. S. G. Vieira - 2016006492

1 Introdução

Problemas simples podem se tornar de alguma forma difíceis quando os impomos restrições. Por exemplos, operações em matrizes não costumam ter muito mistério, uma vez que todas as posições são facilmente mapeadas e relacionadas, contudo, se esta matriz for muito grande, temos um limitador chamado tamanho da memória alocada. Se ela for maior, por exemplo, que a capacidade de sua memória RAM disponível, a operação não é possível e podem haver problemas superficiais em sua máquina. Por isso existem métodos como ordenação externa, que particiona o problema em frações de tamanho menor ou igual um espaço fixo determinado na RAM e utiliza memória secundária para guardar os demais dados, visto que esta é maior.

Neste trabalho há interessse em encontrar, sob restrição de memória definida na chamada do programa, a média aritmética e a mediana das linhas de uma matriz NM, além disso, a média aritmética de toda a matriz.

A média aritmética é o somatório dos elementos dividido pelo número de elementos. A mediana é, uma vez que os elementos estão ordenados, o elemento da posição central. Caso o número de elementos seja par, ela é a média entre os dois elementos de posição central.

A chamada do programa deve ser./tp3 <INPUT> <OUTPUT> LIM para LIM = limite da memória a ser usada.

1.1 Entrada

Na primeira linha serão dados dois inteiros, u e c, que determinam o número de linhas e colunas (respectivamente) da matriz. Nas demais u linhas há c inteiros, separados por espaço, a serem processados.

1.2 Saída

Para cada uma das linhas da matriz será printado no arquivo saída um par <MEDIA>,<MEDIANA> e, no fim, a média total da matriz.

2 Modelagem do problema

Para fornecer tal resultado precisamos ordenar externamente cada uma das linhas da matriz, utilizando o espaço de memória definido apenas.

2.1 Separação das linhas

Como a ordenação deve ser feita em cada linha, o primeiro problema é dado em como separar cada uma delas para que seja ordenada corretamente, sem interferir das demais. Aqui utilizamos um arquivo auxiliar chamado temp.txt, que copia a linha atual completa do arquivo original. É neste arquivo que performaremos todas as operações de ordenação etc.

2.2 Ordenação interna

Como as entradas são inteiros, a memória alocada será um vetor de inteiros, sendo assim, não foi necessária a criação de um TAD para guardar os elementos. Por isso, optei por um quicksort interno comum para ordenar o vetor que representa a memória atual.

O quicksort consiste em particionar o vetor no meio e colocar para a direita do pivot (meio) os números maiores que ele e para a esquerda os menores. Então seguimos operando a mesma coisa para todos os subvetores que se formam a direita e esquerda dos pivôs até que não há mais maneira de partir e o vetor está ordenado.

Para um segundo momento, quando precisamos reordenar a memória, mas ela está quase ordenada, visto que usamos o quicksort, decidi usar o insertion-sort, pela simplicidade. Ele funciona inserindo os elementos do vetor na posição que lhe cabe de acordo com os elementos que estão a sua frente.

2.3 Ordenação externa

Visto que há restrição de memória, apenas a ordenação interna não resolve nosso problema, uma vez que lidaremos com vários vetores como o exemplificado anteriormente, que devem ser ordenados entre si. Para tanto, utilizamos aqui o quicksort externo. Este funciona de maneira muito similar ao quicksort interno. Nele podemos imaginar uma imagem igual à anterior, mas ao invés de um vetor temos um arquivo e ao invés de elementos temos vetores. Primeiramente geramos quatro apontadores, um de leitura inferior, um de leitura superior, um de escrita inferior e um de escrita superior; e um vetor de interior do tamanho da memória máxima a ser usada. Em seguida, chamamos a partição e re-chamamos quicksort para a esquerda e para a direita, da mesma forma que fazemos com o interno. Na partição o algoritmo é o seguinte:

1. Ajustamos os ponteiros para a posição correta no arquivo
2. Preenchemos a memória a primeira vez
3. Ordenamos a memória com quickSort interno
4. Enquanto existirem inteiro a serem lidos:
    4.1. Lemos o próximo inteiro (sem sobrescrever outro)
    4.2. Se vem antes do min, escrevemos embaixo

4.3. Se vem depois do max, escrevemos em cima

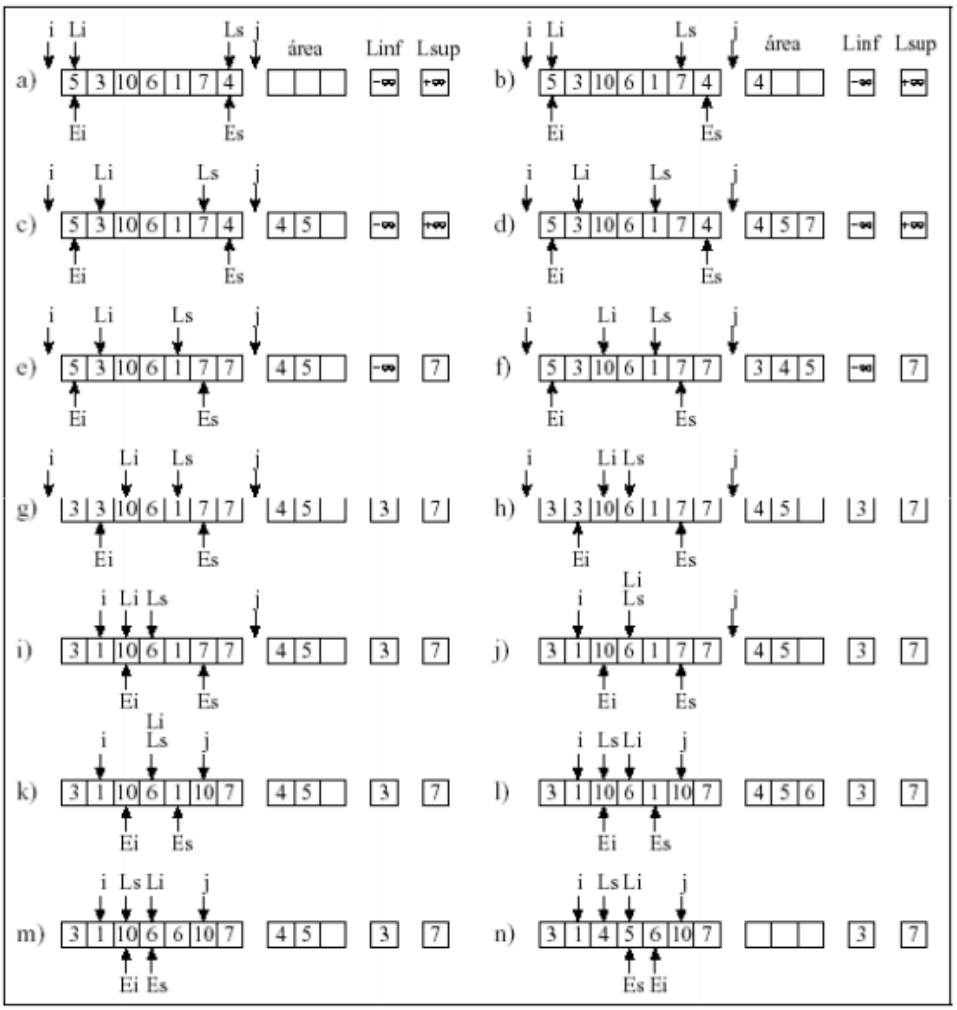
4.4. Se está entre o min e o max substituímos um da memória

4.4.1. Se o apontador de escrita superior andou mais que
o de inferior, substituímos o primeiro da memória

4.4.2. Se não, substituímos o último da memória

4.4.3. Por mim, fazemos insertionSort com a memória,
visto que ela foi alterada, mas está quase toda ordenada.

5. Escrevemos os dados da memória no arquivo. Isso pode ser melhor compreendido com a seguinte figura:



2.4 Média

Para a média da linha lemos todo o arquivo temporário, somando cada um dos números, em seguida, dividimos todos pelo tamanho da linha.Então, retornamos.

Para a média da matriz, toda vez que uma nova média de linha é gerada somamos às anteriores, ao final de tudo, dividimos este somatório pelo número de linhas.

2.5 Mediana

Após a ordenação, verificamos se temos um numero par ou ímpar de linhas. Caso seja ímpar, percorremos o arquivo temporário até a posição (size/2) +1, ou seja, a do meio, e a retornamos. Caso seja par, percorremos até as posições size/2 e (size/2)+1 e fazemos a média de ambas, então retornamos.

3 Análise teórica do custo assintótico

3.1 Tempo

3.1.1 Library

- separateLine: O(c). Para c = número de colunas da matriz. Visto que copiamos cada um dos n elementos para o arquivo novo.
- getAverage: O(c). Para c = número de colunas da matriz. Visto que percorremos toda a linha para somar seus valores.
- getMedian: O(c). Para c = número de colunas da matriz. Visto que percorremos metade da linha para pegar o(s) valor(es) do meio.

3.1.2 InternalSorting

- quicksort: O(K log K).Para K = tamanho da memória interna utilizada. Visto que este é o caso médio e melhor caso do algoritmo e este é usado no vetor memória. Para o pior caso teríamos O(K), que seria resultado da partição ter ocorrido em uma extremidade do vetor, fazendo-nos operá-lo inteiro.
- insertionsort: O(K). Para K = tamanho da memória interna utilizada. Visto que este é o caso médio e pior caso do algoritmo. Contudo, como apenas o usamos quando um único elemento está fora do lugar (no máximo), nos aproximamos muito do melhor caso O(K) na prática.

3.1.3 ExternalSorting

O((c/b)*log(c/K). Para K = número de registros que cabem na memória interna; b = quantidade de registros que podem ser armazenados no bloco de

leitura/gravação do SO; e c = número total de dados a serem ordenados, ou seja, número de colunas da matriz original. Este resultado foi provado por uma demosntração exaustiva de M.C. Monard, publicada em sua tese de doutorado na PUC-RJ chamada "Projeto e análise de algoritmos de classificação externa baseados na estratégia de Quicksort", como caso médio. Contudo, para o pior caso, assim como no quicksort interno, a partição pode ocorrer em uma extremidade, o que nos leva a custo O(c/K).

Neste caso houve um problema no uso da função fseek(), visto que operávamos inteiros separados por espaço. Ela da maneira padrão não conseguia localizar corretamente os inteiros se estes fossem lidos como char, visto que podem ser compostos de mais de um caracter, e dava erro quando eram lidos como int, visto que precisava passar pelos espaços. Para resolver isto, utilizei uma gambiarra de apontar o ponteiro com fseek() sempre para o início do arquivo e percorre-lo com um for, isso adiciona ao tempo uma complexidade O(n), fora do esperado, nestes momentos de ajuste de ponteiro, o que tornou o custo menos satisfatório.

3.2 Espaço

Por se tratar de um algoritmo que trabalha com memória secundária, para a análise assintótica de espaço é necessário analisar somente as funçôes de ordenação. Ambas têm complexidade O(K), para K = tamanho do vetor que aloca a memória primária, visto que apenas mudam elementos de lugar nele mesmo.

4 Avaliação experimental

A implementação da solução proposta foi compilada utilizando gcc 4.8.4. Os experimentos foram executados em um sistema com 8 GB de memória e um processador Intel® Core™ i5-5200U CPU @ 2.20GHz 4. Cada teste foi realizado em torno de 10 vezes e, a partir disso, foi feira uma média dos tempos.

Neste caso usei o teste padrão enviado como toy, que tem um número par de elementos por linha, e um segundo teste, com número ímpar de elementos por linha.

	real	user	sys
Input 1	0m0,002s	0m0,001s	0m0,000s
Input 2	0m0,004s	0m0,003s	0m0,001s

5 Conclusão

Quanto ao tempo e ao espaço a resolução se mostrou satisfatória. Nos testes realizados os resultados deram certo. Giramos sempre em torno de O(n), para n = tamanho da linha, ou O(K), para K = tamanho da memória disponível.

Infelizmente houve o acrescimo do caminhamento no arquivo, mas ainda é bom. Outro problema foi que, como o padrão de entrada dado é em Mb, isso signifca um arquivo muito grande de inteiros, então para testar o arquivo disponibilizado eu setei um tamanho máximo fixo. Contudo, a conta da conversão MB para número de inteiros está lá.