Algoritmo e Estrutura de Dados II

Trabalho Pratico 2

Isabela Meneguci de Souza Petrim

Junho de 2017

1 Indrudução

O trabalho prático foi o 'Switch gravitacional' , no qual uma aluna montou uma caixa que tinha a capacidade de alterar a gravidade. Esse brinquedo inicialmente estava com a gravidade direcionada para o centro da terra (ou "para baixo") e após realizar a alteração o sentido da gravidade seria alterado para esquerda.

O problema proposto consistia em fazer um programa que mostrasse qual seria a quantidade de cubos de cada uma das n colunas depois que a aluna alterou a gravidade. Para realização do problema foi observado que apos o switch gravitacional o vetor de entrada era ordenada. Deste modo foi utilizado um algoritmo de ordenação para realização desde problema .

2 Desenvolvimento

Para realizar o trabalho prático era necessário que o programa gerado fosse capaz de ler um arquivo de entrada e gerar como resultado um arquivo de saída. O arquivo de entrada de nome "input.txt "consiste em duas linhas, onde a primeira representa o numero de colunas que a caixa possui. E a segunda cada número (separados por espaço) a quantidade de blocos em cada coluna. Além disso, o programa deveria gerar um arquivo de saída "output.txt"contendo apenas uma linha com a saída dos cubos após o Switch gravitacional.

Além dos de texto o programa possui outros seis arquivos. Desses, tem-se um arquivo de nome "biblioteca.h"que faz a ligação entre 3 arquivos ".c". No arquivo de nome "main.c"ele abre o arquivo de texto de entrada e grava na variável x o retorno da chamada da função "openFile.c":

Algoritmo 1 main.c

```
function INT MAIN(int argc, char *argv[])
    FILE *arq;
    arq = fopen("input.txt","r");
    int x=openFile(arq);
    ...
```

Esse arquivo por sua fez tem por objetivo validar a abertura correta do arquivo de entrada "input.txt"e retornar 0 caso tenha algum erro com o arquivo e 1 caso ocorra tudo certo:

Algoritmo 2 openfile.c

```
function INT OPENFILE(FILE *arq)
    int open;
    if if(arq == NULL) then
        printf("ao abrir o arquivo!");
        return open=0
    return open=1;
```

Após chamar a função se a variável 'x' conter 1, o programa lê a primeira linha do arquivo que contém o número de colunas e grava esse na variável 'colunas' e aloca-se memoria dinamicamente para o vetor 'v' de acordo com a quantidade de colunas:

Algoritmo 3 main.c

```
function INT MAIN(int argc, char *argv[])
    ...
    if if(x == 0) then
        fscanf(arq,"d",colunas);
        v = (int*) malloc(colunas*sizeof(int));
```

Logo em seguida em uma estrutura de 'loop' é gravado em cada posição do vetor v a segunda linha do arquivo, onde cada número contém a quantidade de blocos em cada coluna:

A posteriore, o arquivo main chama a função "heapsort"que foi implementada para ordenar o vetor 'v'. Este algoritmo foi escolhido, visto que nas especificações do trabalho foi informado que a quantidade de colunas (representado por n) poderia variar de: 1 n 100000 , ou seja, a quantidade de colunas poderia ser de no máximo 100.000, quantidade essa considerada moderada.

Algoritmo 4 main.c

```
function INT MAIN(int argc, char *argv[])
    ...
    for (j=0;j<colunas;j++) do
        fscanf(arq,"d",v[j]);
```

Após a função 'heapsort' retornar o vetor v ordenado, o main chama a função "writeFile"passando para ela o vetor v – que já está ordenado pelo heapsort- e o numero de colunas. Essa função por sua vez abre ou se não existir cria o arquivo "output.txt", valida se essa abertura ou criação foi feita de forma correta, se sim grava no arquivo de saída o vetor v ordenado.

Algoritmo 5 writefile.c

```
function VOID WRITEFILE(int *v, int colunas)
    ...
    for (i=0;i<colunas;i++) do
        fprintf(arqout,"d ",v[i]);
```

Além disso o programa possui um arquivo "biblioteca.h"que faz o intercâmbio entre as funções '.c', um arquivo "complile.sh"que contem em bash (linguagem de programação nativa do linux) a compilação de todos os arquivos '.c' e execução do programa.

2.1 Como executar o programa?

Para executar o programa entre na pasta que o arquivo está através do terminal e se estiver utilizando o sistema operacional Linux basta digitar ". complile.sh"e abrir o arquivo "ouput.txt"que irá conter a saída final do programa. Caso esteja usando o sistema operacional windows siga os passos a baixo:

```
gcc -c heapsort.c
gcc -c openFile.c
gcc -c writeFile.c
gcc main.c heapsort.o openFile.o writeFile.o -o program
program
```

3 Análise de Complexidade

Para fazer o estudo da análise de complexidade do programa começaremos pela análise do arquivo HeapSort. A complexidade do arquivo heapsort é (n log n). A analise da função openFile é o(1) já que essa só realiza comparações e não possui nenhum loop:

Algoritmo 6 openfile.c

```
function INT OPENFILE(FILE *arq)
    if (arq == NULL) then
        ...
=0 ...
```

A complexidade do arquivo Writefile é O(n) devido ao loop, com n=colunas:

Algoritmo 7 writefile.c

```
function VOID WRITEFILE(int *v, int colunas)
    ...
    for (i=0;i<colunas;i++) do
     ...
```

A complexidade do arquivo main é a complexidade de todas as funções que ele chama mais a complexidade dos loops e comparações nela. Dentro do próprio main a complexidade dos loops e comparações é O(n), já que ela é O(1) para a comparação do if + O(n) para o loop

Algoritmo 8 main.c

```
function INT MAIN(int argc, char *argv[])
    if (x == 1) then
        ...
        for (j=0;j<colunas;j++) do
            ..
```

Ficando assim a complexidade total igual a : O(1)+O(n)+O(n(log(n)))+O(n)=O(n(log(n)))

4 Testes Experimentais

O tempo de execução do programa praticamente não varia. Para constatar este fato foi realizado testes usando entradas de tamanhos diferentes e obitido o tempo de execução em milissegundos, aproximadamente:

Entrada: Ordem aleatoria	tempo
ate 1.000	1
5.000	4
10.000	7
50.000	29
100.000	56

Tabela 1: Tabela para ordem aleatoria

Entrada: Ordem crescente	tempo
ate 1.000	2
5.000	3
10.000	5
50.000	18
100.000	34

Tabela 2: Tabela para ordem crescente

Entrada: Ordem decrescente	tempo
ate 1.000	2
5.000	6
10.000	8
50.000	16
100.000	33

Tabela 3: Tabela para ordem decrescente

5 Conclusão

A partir da analise de complexidade feita é possível concluir que o algoritmo usado possui um bom desempenho comparado aos outros algoritmos de ordenação que poderiam ser usados.