Trabalho Prático 2 AEDS II

Switch gravitacional

https://github.com/danilo-p/aeds-ii-tp-2

Danilo Pimentel de Carvalho Costa

E-mail: danilo.pimentel@dcc.ufmg.br

Matrícula: 2016058077

Introdução

Fernanda construiu uma caixa com a funcionalidade de alterar a gravidade. Dentro desta caixa ela colocou cubos empilhados. O cenário do problema consiste na implementação de um programa que determina a quantidade de cubos em cada coluna da caixa construída por Fernanda após a mudança de gravidade.

Mudança da gravidade dentro da caixa

Após a mudança de gravidade, percebe-se que as colunas com menos cubos sempre ficarão à direita da caixa, enquanto as colunas

com mais cubos ficarão no sentido oposto. Considerando esta observação, a solução proposta neste trabalho foi a ordenação crescente das colunas de acordo com a quantidade de cubos.

Desenvolvimento

A solução proposta foi a ordenação das colunas de cubos contidas na caixa, ou seja, os números recebidos no arquivo de entrada. No caso deste programa, o arquivo *io/input.txt* deve conter a entrada. Este arquivo é lido pelo programa, e o resultado da ordenação é escrito no arquivo *io/output.txt*.

Depois da implementação de outros algoritmos de ordenação, o algoritmo escolhido, com base na análise de complexidade e tempo de execução, foi o *Quicksort*. A implementação deste e dos outros algoritmos está no arquivo *src/lib/sort.c*. A fim de não estender esta documentação ao longo de todos os algoritmos implementados, só será detalhado o algoritmo utilizado. Abaixo se encontra a implementação do *Quicksort*:

Implementação do *Quicksort*

```
void quickSort (int *array, int length) {
    if(length > 1) {
        int i, j, tmp,
            first = array[0],
            last = array[length-1],
            middle = array[length / 2],
            pivot = (first > last) ?
                        (middle > first ? first : middle):
                        (middle > last ? last : middle);

        for (i = 0, j = length - 1;; i++, j--) {
            while (array[i] < pivot) i++;
            while (array[j] > pivot) j--;

            if (i >= j) break;

            tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
        }
        quickSort(array, i);
        quickSort(array + i, length - i);
    }
}
```

O algoritmo é recursivo, sendo o critério de término da recursão o tamanho do *array* recebido como parâmetro. O *pivot* é escolhido com base na mediana do primeiro número, último e o número que está no meio do array.

A escolha desta estratégia foi baseada na consideração da pior combinação de entrada. Com a mediana de três, o pior caso se dá pela escolha do maior ou menor elemento do *array* como *pivot*. Considerando que as entradas são aleatoriamente escolhidas, a organização dos elementos desta forma fica mais difícil.

Análise

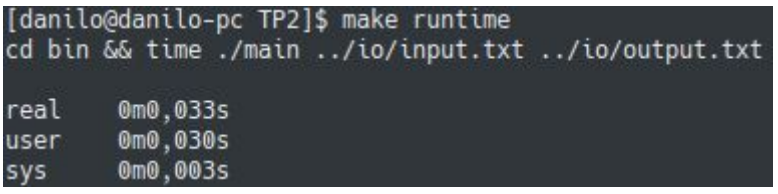
Como o algoritmo escolhido para a ordenação do array foi o Quicksort, a complexidade do programa é de $O(n * log(n))$, sendo n a largura do *array* lido do arquivo de entrada.

A complexidade de todos os algoritmos implementados se encontra em comentários no arquivo *src/lib/sort.h*, com exceção da função *main*. No caso da função *main*, a complexidade está documentada no arquivo *src/main.c*.

Além da comparação entre a complexidade dos algoritmos, foram feitos experimentos para determinar-se o algoritmo mais conveniente. Utilizando a mesma entrada para todos os algoritmos implementados (*Quicksort, Insertion Sort, Selection Sort* e *Bubble Sort*), foram também comparados os tempos de execução.

Execute o comando *make runtime* para obter informações sobre o tempo de execução do programa. Abaixo encontram-se os resultados obtidos:

Quicksort



Insertion Sort

```
[danilo@danilo-pc TP2]$ make runtime
cd bin && time ./main ../io/input.txt ../io/output.txt

real    0m6,883s
user    0m6,879s
sys     0m0,003s
```

Selection Sort

```
[danilo@danilo-pc TP2]$ make runtime
cd bin && time ./main ../io/input.txt ../io/output.txt

real    0m13,396s
user    0m13,366s
sys     0m0,027s
```

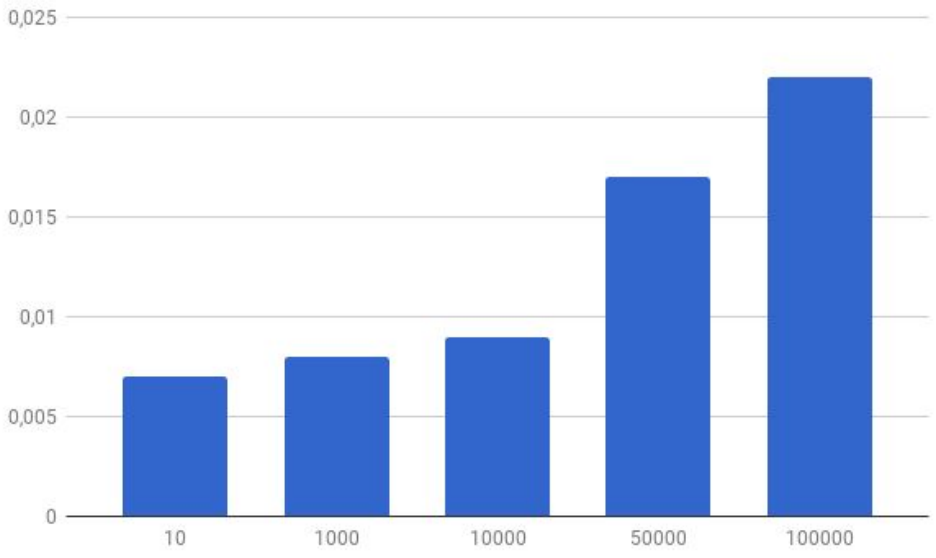
Bubble Sort

```
[danilo@danilo-pc TP2]$ make runtime
cd bin && time ./main ../io/input.txt ../io/output.txt

real    0m29,989s
user    0m29,982s
sys     0m0,007s
```

Uma vez decidido o algoritmo mais conveniente para a solução do problema, foi feita a análise do tempo de execução do programa com base no tamanho da entrada. Os dados obtidos são apresentados abaixo, em forma de gráfico de barras:

Tempo de execução (segundos) x Tamanho da entrada



Conclusão

Após a comparação do tempo de execução do programa, chega-se à conclusão de que, dos algoritmos implementados, o *Quicksort* é o algoritmo mais conveniente. Além disso, analisando os valores obtidos na execução do programa utilizando o algoritmo de ordenação Quicksort, podemos ver que a curva formada pelos pontos aproxima-se da curva da função $n * log(n)$, confirmando a análise de complexidade total do programa.