Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação
SCC0224 - Estruturas de Dados II

Relatório Projeto 1

Alunos: Bruno Ideriha Sugahara, Raul Ribeiro Teles, Vinicius de Moraes
Professor: Robson L. F. Cordeiro

Maio
2023

Conteúdo

1 Introdução

Foram implementados 5 algoritmos de ordenação (Bubblesort, Bubblesort otimizado, Quicksort, Radix Sort e Heapsort) de forma fiel à apresentada nos pseudocódigos do projeto.

A implementação foi realizada com alocação dinâmica de memória para medição do tempo, além de ser estruturada em um TAD com arquivos .c e .h e structs para os elementos das listas.

O código está organizado em um arquivo makefile, no qual pode ser usado o comando "make"para compilar o programa e "make run"para executá-lo.

2 Análises de complexidade

A complexidade de cada um foi analisada através de uma análise assintótica teórica de cada caso (melhor, pior e médio) e através de uma análise empírica na qual foi medido o tempo de execução em segundos para três tipos de sequência (aleatória, crescente e decrescente) com tamanhos N 1000, 10000 e 100000. Sequências maiores não foram utilizadas por questões práticas de limitação de hardware.

2.1 Bubblesort

2.1.1 Análise teórica

```
74 void bubbleSort(lista *lista) {
75         for (int i = 0; i < lista->tamanho - 1; i++) {
76                 for (int j = 0; j < lista->tamanho - 2 - i; j++) {
77                         if (lista->elementos[j] > lista->elementos[j+1]) {
78                                 troca(&(lista->elementos[j]), &(lista->elementos[j+1]));
79                         }
80                 }
81         }
82 }
```

O bubblesort consiste em dois laços de repetição "for"aninhados: O laço externo repete n - 1 vezes, já o laço interior repete n - 1 - i vezes.

Dentro dessas repetições há um teste condicional que testa se um elemento na posição j é maior que o elemento à sua direita no vetor. Se for verdadeiro, as variaveis são trocadas por um custo constante, caso contrário, não há nenhum custo adicional ao da comparação.

Esse bloco de custo constante vai ser repetido n - 1 - i vezes pelo for interno e esse for interno vai repetir n - 1 vezes, resultando em um custo de:

$$f(n) = ((n-1-0)*k + (n-1-1)*k + ... + 2*k + 1)\cancel{*k}$$

$$= ((n-1) + (n-2) + ... + 2 + 1)] = n + n + ... + n$$

$$= n*(n-1)/2$$

$$= (n^2 - n)/2$$

O que nos leva a conclusão de que o bubblesort é ordem de $O(n^2)$. O algoritmo sempre repete o mesmo número de comparações independente do vetor, possuindo a mesma complexidade assintótica no pior caso, no melhor caso e no caso médio e independente da ordem do vetor.

2.1.2 Análise empírica

Os tempos de execução para listas de ordem 3 a 5 eram excessivamente longos no computador utilizado, fazendo com que fossem substituídas por listas de ordem 2 a 4.

N	Aleatória	Crescente	Decrescente
10^2	0.0001 ± 0.0000	0.0000 ± 0.0000	0.0001 ± 0.0000
10^3	0.0032 ± 0.0004	0.0012 ± 0.0001	0.0044 ± 0.0005
10^4	0.3525 ± 0.0038	0.1253 ± 0.0011	0.4953 ± 0.0062

Os resultados demonstram um grande crescimento do tempo de execução: Podemos muitas vezes ver o tempo aumentar em duas ordens de magnitude, o que faz sentido com o comportamento quadrático do algorítimo.

Não há grandes diferenças entre as sequências, mas ainda é possível ver que as crescentes são ordenadas mais rapido e as decrescentes mais devagar.

Uma hipótese para explicar essa diferença seria que, enquanto o número de comparações é o mesmo entre os vetores, o número de swaps é significativamente mais em vetores próximos de decrescentes.

2.2 Bubblesort Otimizado

2.2.1 Análise teórica

```c
void bubbleSortOtimizado(lista *lista) {
    for (int i = 0; i < lista->tamanho - 1; i++) {
        bool ordenado = true;
        for (int j = 0; j < lista->tamanho - 1 - i; j++) {
            if (lista->elementos[j] > lista->elementos[j+1]) {
                ordenado = false;
                troca(&(lista->elementos[j]), &(lista->elementos[j+1]));
            }
        }
        if (ordenado) {
            break;
        }
    }
}
```

O bubblesort pode ser melhorado ao adicionar uma variável booleana que é inicializada como verdadeira no laço de repetição externo e caso no laço interno seja feita uma troca, ela é marcada como falsa.

Temos agora os custos adicionais de settar a variável como verdadeira e testar se ela foi modificada após cada iteração. Ambos esses custos são $O(n)$ (pois são custos constantes que repetem n vezes), ou seja, não aumentam o custo assintótico do programa. Há também o custo de settar a variável como falsa quando após uma troca, mas pode ser desprezável.

O custo do bubblesort vai se manter como $O(n^2)$ no pior caso (vetor em ordem decrescente) e no caso medio (vetor aleatório). No entanto, temos um

custo linear no melhor caso (vetor já ordenado), ao sair do laço na primeira iteração.

2.2.2 Análise empírica

N	Aleatória	Crescente	Decrescente
10^2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000
10^3	0.0029 ± 0.0001	0.0000 ± 0.0000	0.0043 ± 0.0004
10^4	0.3587 ± 0.0024	0.0003 ± 0.0000	0.4041 ± 0.0219

Novamente, os tempos de execução para listas de ordem 3 a 5 eram excessivamente longos no computador utilizado, fazendo com que fossem substituídas por listas de ordem 2 a 4.

Os casos de vetor aleatório e vetor decrescente não apresentaram muita mudança. No entanto esse algoritmo "ordena"vetores ordenados em um tempo muito menor, pois ele é linear nesse caso. Outro resultado interessante é o desvio padrão de 0, pois o bubblesort otimizado realizou a mesma quantidade de comparações(n-1) e trocas(0) para todos os vetores ordenados.

2.3 Quicksort

2.3.1 Análise teórica

```
218 elemento particiona(lista *lista, long ini, long fim) {
219         /* Escolha do pivo */
220         elemento pivo = lista->elementos[fim];
221
222         long i = ini;
223         for (long j = ini; j < fim; j++) {
224                 if (lista->elementos[j] <= pivo) {
225                         troca(&(lista->elementos[i]), &(lista->elementos[j]));
226                         i++;
227                 }
228         }
229         /* coloca o pivo na posicao correta e retorna a posicao do pivo */
230         troca(&(lista->elementos[i]), &(lista->elementos[fim]));
231         return i;
232 }
233 elemento particaoAleatoria(lista *lista, long ini, long fim) {
234         srand(clock());
235         long k = ini + rand() % (fim - ini); /* pivo escolhido de posicao aleatoria */
236         troca(&(lista->elementos[k]), &(lista->elementos[fim])); /* Coloca o pivo escolhido no fim */
237         return particiona(lista, ini, fim); /*particiona o vetor de acordo com o pivo */
238 }
```

O quick sort consiste em achar um pivô e colocá-lo na posição correta, resultando em um vetor à esquerda e outro à direta do pivo, sendo possível ordená-los recursivamente.

A função particaoAleatoria escolhe um pivô com custo constante e chama a função particionar, que coloca o pivô na posição correta com o loop da linha 223. Esse loop repete fim - ini vezes e dentro dele temos um custo constante. É possível arredondar o custo da etapa para $O(n)$.

Em seguida repetimos o processo com os subvetores esquerdo e direito a partir do pivô. É nessa parte que a complexidade varia de acordo com o vetor, ou mais especificamente, com a escolha dos pivôs.

O melhor caso é quando são escolhidos pivôs próximos à mediana, assim o vetor é dividido no meio, o que vai resultar em uma complexidade final do programa de $T(n) = n + T(n/2)$. Esse custo pode ser calculado para $O(n \log n)$. O caso médio não se distancia muito do melhor caso e possui a mesma complexidade. Já o pior caso, é quando o pivô escolhido é sempre o maior ou o menor elemento do vetor. Isso faz com que cada particionamento divida o vetor em um vetor vazio e outro vetor de tamanho n - 1. O custo fica $T(n) = n + T(n-1)$, resultando em uma complexidade $O(n^2)$ para o algoritmo no pior caso.

2.3.2 Análise empírica

N	Aleatória	Crescente	Decrescente
10^3	0.0016 ± 0.0003	0.0012 ± 0.0001	0.0017 ± 0.0005
10^4	0.0145 ± 0.0005	0.0120 ± 0.0002	0.0131 ± 0.0002
10^5	0.1440 ± 0.0021	0.1224 ± 0.0015	0.1328 ± 0.0031

É possível ver claramente o quão mais rápido é o quicksort em contra partida ao bubblesort. O crescimento foi muito mais controlado em comparação ao algoritmo anterior.

Por mais que o pior caso do quicksort seja quadrático, ele aparentemente é muito raro: em todos os casos os tempos foram pequenos, assim como o desvio padrão. Houve também pouca variação quando comparados vetores aleatórios, crescentes e decrescentes.

2.4 Radix Sort

2.4.1 Análise teórica

O radix sort é um algoritmo de ordenação de números inteiros que não precisa fazer comparações e que classifica os dados com chaves inteiras, agrupando as chaves pelos valores individuais que compartilham a mesma posição e valor.

```
void radix_sort(lista *lista, long tamanho)
{
    elemento maior = pega_maior(lista,tamanho);
    int posicao = 1;

    while(maior/posicao > 0)
    {
        counting_sort(lista,tamanho,posicao);
        posicao *= 10;
    }
}
```

Inicialmente é procurado o maior valor no vetor: O algorítmo percorre o vetor todo $O(n)$. Com o maior valor encontrado, se entra em um laço que repete k vezes, sendo k o numero de dígitos do maior valor.

```
void counting_sort(lista *lista, long tamanho, int posicao)
{
    int b[10] = { 0 };
    int c[tamanho];

    for(int i = 0; i < tamanho; i++)
    {
        b[(lista->elementos[i]/posicao)%10]++;
    }

    for(int i = 1; i < 10; i++)
    {
        b[i] += b[i-1];
    }

    for(int i = tamanho - 1; i >= 0; i--)
    {
        c[b[(lista->elementos[i]/posicao)%10]-1] = lista->elementos[i];
        b[(lista->elementos[i]/posicao)%10]--;
    }

    for(int i = 0; i < tamanho; i++)
    {
        lista->elementos[i] = c[i];
    }
}
```

Dentro desse laço há o counting sort,que realiza três laços $O(n)$ e um laço que se repete 10 vezes, resultando em uma complexidade $O(n)$.

Temos então um algoritmo de complexidae k*n, mas podemos desconsiderar o k pois o numero de dígitos raramente vai acompanhar o crescimento de n. Logo a complexidade final do algoritmo é $O(n)$. O algoritmo não varia com a ordem do vetor, mantendo sempre a mesma ordem de complexidade.

2.4.2 Análise empírica

N	Aleatória	Crescente	Decrescente
10^3	0.0002 ± 0.0000	0.0003 ± 0.0000	0.0002 ± 0.0000
10^4	0.0025 ± 0.0002	0.0024 ± 0.0003	0.0024 ± 0.0003
10^5	0.0240 ± 0.0004	0.0254 ± 0.0025	0.0246 ± 0.0010

É possível ver que o Radix Sort cresce de forma mais lenta ainda, mesmo sendo comparado ao quicksort. A diferença de tempo entre os casos crescente, decrescente e aleatório é praticamente imperceptível. Essa consistência está de acordo com a lógica do algoritmo.

2.5 Heapsort

2.5.1 Análise teórica

```
void heapSort(lista *A, long n) {
    for(long i = n / 2 - 1; i >= 0; i--) {
        heapify(A, n , i);
    }
    for(long i = n - 1; i >= 1; i--) {
        troca(&(A->elementos[i]), &(A->elementos[0]));
        heapify(A, i, 0);
    }
}
```

O heapsort utiliza uma árvore binária, de modo que o procedimento de inserir e remover valores através do heapify() pode percorrer toda a altura da árvore (log n), possuindo um custo $O(logn)$.

```
void heapify(lista *A, long n, long i) {
    long esquerda, direita, maior;

    maior = i;
    esquerda = 2 * i + 1;
    direita = 2 * i + 2;

    if(esquerda < n && A->elementos[esquerda] > A->elementos[maior])
        maior = esquerda;
    if(direita < n && A->elementos[direita] > A->elementos[maior])
        maior = direita;
    if(maior != i) {
        troca(&(A->elementos[i]), &(A->elementos[maior]));

        heapify(A, n, maior);
    }
}
```

Na construção do heap, o procedimento heapify() é chamado $n/2$ vezes. Em seguida, chamado novamente aproximadamente $n-1$ vezes para colocar o maior elemento no fim da lista. Logo, o custo total é $O(n * logn)$.

O melhor caso para o heapsort possui custo $O(n)$ (chaves repetidas), enquanto o caso médio e o pior caso possuem custo $O(n * logn)$.

2.5.2 Análise empírica

N	Aleatória	Crescente	Decrescente
10^3	0.0002 ± 0.0000	0.0002 ± 0.0000	0.0001 ± 0.0000
10^4	0.0031 ± 0.0004	0.0022 ± 0.0002	0.0021 ± 0.0002
10^5	0.0316 ± 0.0004	0.0258 ± 0.0010	0.0252 ± 0.0010

O heapsort teve resultado surpreendente. Como um algoritmo nlogn, era esperado que ele se comporta-se de forma semelhante ao quicksort, mas ele apresentou execuções consideravelmente mais rápidas, semelhante ao radixsort. Todos os algoritmos estão funcionando corretamente e foram realizadas várias medidas, sendo que os resultados se repetiram em todas as vezes.

O desvio padrão também foi pequeno, o que era esperado, pois o heapsort é um algoritmo bem estável.

3 Conclusão

Após consideração dos resultados é possível observar que o algoritmo Bubblesort, mesmo em sua versão optimizada, é extremamente ineficiente em situações de alto volume, com listas de tamanho 10^5 em diante sendo inviáveis num cenário realista.

Os algoritmos Radix sort e Heapsort demonstraram o melhor desempenho para as sequências utilizadas, com um pequeno destaque para o Radix sort nas maiores listas, condizente com sua complexidade linear. Os dois aparentam ser a melhor escolha em relação ao tempo, além de possuirem vantagens como a estabilidade.

O Heapsort apresentou desempenho acima do esperado, principalmente quando comparado com o Quicksort, cuja expectativa era de ser o algoritmo mais rápido do projeto. A implementação de ambos está funcional e muito similar às versões apresentadas no pseudocódigo do projeto, sendo que a diferença se manteve ao decorrer de diversos experimentos, podendo ser possivelmente atribuída à um fator externo como o sistema operacional do computador.