UNIVERSIDADE DE SÃO PAULO
EESC / ICMC

Estrutura de Dados II

Análise de complexidade de algoritmos de ordenação

André Vargas Villalba Codorniz - 14558436

Lucas Augusto Moreira Barros - 14590610

PROFESSORA:

Maria Cristina Ferreira de Oliveira

São Carlos, Março de 2024

Conteúdo

1 Introdução

Este trabalho consiste na análise teórica e empírica do algoritmo de ordenação "Bubble Sort", o qual foi implementado em linguagem C. Será visto que o código possui três diferentes versões da função de ordenação 'bubble_sort', as quais diferem entre si apenas por pequenos detalhes de implementação, sendo que tais modificações foram inseridas com a intenção de otimizar o algoritmo em questão. Neste documento, primeiramente foi explicada a lógica das três variações de bubble sort implementadas, na sequência foram feitas análises teóricas e empíricas (também abordando os algoritmos fornecidos no enunciado do trabalho) dos algoritmos de ordenação.Finalmente, foram realizadas conclusões acerca do desenvolvimento do trabalho.

2 Desenvolvimento

2.1 Explicação do algoritmo

Bubble Sort é um dos algoritmos de ordenação mais conhecidos, bem como um dos mais simples. Sua lógica consiste em comparar repetidamente pares de termos adjacentes do vetor e, caso verifique que estão fora de ordem, trocá-los de posição. A seguir está a implementação do algoritmo, em sua versão mais simples, na linguagem C:

```
void bubble_sort_0(int *arr, int size){
    int aux;
    for (int k = 1; k < size; k++) {
        for (int j = 0; j < size - 1; j++) {
            if (arr[j] > arr[j+1]) {
                aux    = arr[j];
                arr[j]   = arr[j+1];
                arr[j+1] = aux;
            }
        }
    }
}
```

Listing 1: Bubble sort sem otimização

Do modo como atua a função, é certo que a cada execução completa do laço interior (segundo **for**), os maiores elementos estarão posicionados corretamente nas últimas "casas"do vetor. É a partir desse fato que surge a primeira ideia de otimização do algoritmo: a cada iteração uma "casa"a menos do vetor é usada para comparação (sempre as últimas "casas"), uma vez que tem-se a certeza de que o final do vetor já está ordenado. Dessa forma, tal mudança busca melhorar o desempenho do código uma vez que reduz-se o número de comparções feitas. O código otimizado é implementado da seguinte maneira (apenas o fator de comparação do segundo "for"é alterado):

```c
void bubble_sort_1(int *arr, int size){
    int aux;
    for (int k = 1; k < size; k++) {
        for (int j = 0; j < size - k; j++) {
            if (arr[j] > arr[j+1]) {
                aux      = arr[j];
                arr[j]   = arr[j+1];
                arr[j+1] = aux;
            }
        }
    }
}
```

Listing 2: Bubble sort com a primeira otimização

A segunda otimização pensada para o código foi a criação de uma flag, a qual indica se houve ou não alguma troca de posição de elementos. Caso o vetor tenha sido percorrido de maneira correta no loop interior e nenhuma troca tenha se efetivado, conclui-se que o vetor já está ordenado. Logo, não são necessárias mais comparações e o algoritmo bubble sort é finalizado. A seguir é mostrada a implementação da variável flag:

```c
void bubble_sort_2(int *arr, int size){
    int aux, flag;
    for (int k = 1; k < size; k++) {
        flag = 0;
        for (int j = 0; j < size - k; j++) {
            if (arr[j] > arr[j+1]) {
                aux      = arr[j];
                arr[j]   = arr[j+1];
                arr[j+1] = aux;
                flag = 1;
            }
        }
        if (flag == 0) {
            break;
        }
    }
}
```

Listing 3: Bubble sort com a segunda otimização

2.2 Análise teórica do algoritmo

Fazendo a anáise teórica do algoritmo Bubble Sort sem nenhuma melhoria (Listing 1) nota-se que sua complexidado no pior caso se dá pela fórmula:

$$C(n) = 6n^2 - 7n + 4 \tag{1}$$

Isso nos dá a curva assintótica do pior caso (big-O) como sendo O(n^2), uma complexidade já conhecida na literatura.

Já no melhor caso, o mesmo código apresenta complexidade dada por:

$$C(n) = 3n^2 - n + 1 \tag{2}$$

Sendo assim, a sua curva assintótica de melhor caso (big-Ω) é de $\Omega(n^2)$. Desta forma pode-se concluir que a complexidade média do algoritimo (big-Θ) é de $\Theta(n^2)$, que não o torna um algoritimo inviável, contudo não é um algoritmo extremamente otimizado

Agora analiza-se a complexidade do algoritimo com a primeira melhoria proposta. Para o big-O desse algoritmo tem-se a seguinte fução:

$$C(n) = 3n^2 + 5n + 4 \tag{3}$$

Já a fórmula de complexidade para o melhor caso é:

$$C(n) = \frac{3}{2}n^2 + \frac{7}{2}n + 1 \tag{4}$$

Desta forma, temos que as curvas assintóticas para o algorítmo com a primeira melhoria são $\Theta(n^2)$, $\Omega(n^2)$ e $\mathrm{O}(n^2)$. É interessante notar que, com essa melhoria, a ordem da complexidade não muda, contudo a sua função, isto é, suas constantes, caem consideravelmente, o que não é irrelevante para a otimização do código.

Agora resta a analise do terceiro código, com as das melhorias propostas. Em seu pior caso, a função que rege sua complexidade é:

$$C(n) = \frac{5}{2}n^2 + \frac{15}{2}n + 8 \tag{5}$$

Já na melhor situção possivel, o big-Ω, a função é:

$$C(n) = 3n + 6 \tag{6}$$

Desse modo, tem-se que a função possui $O(n^2)$ e $\Omega(n)$, logo, o caso médio (big-Θ) terá sua ordem limitada superiormente por n^2 e inferiormente por n. Aqui conclui-se que a segunda otimização trazida ao código é de extrema importância e pode diminuir consideravelmente o seu tempo de execução, visto que big-Θ pode assumir caráter linear ($f(n) = 6n$, por exemplo), ou seja, há a possibilidade de que o algoritmo seja bastante eficiente se comparado aos outros dois mencionados anteriormente.

2.3 Análise empírica do algoritmo

Foi implementado um código em C responsável por ler arquivos com diferentes quantidades de elementos e diferentes maneiras de ordenação. A partir da leitura de tais arquivos foi possível calcular o tempo médio de execução para cada um dos três

algoritmos de ordenação antes mencionados (cada arquivo foi ordenado 20 vezes e foi tomada a média dos tempos de execução). Na sequência, os tempos foram gravados em um arquivo csv (para melhor manuseio deles) e depois lidos em um código na linguagem python responsável por gerar um gráfico dos tempos de execução. A seguir estão dispostas as tabelas contendo as variações dos arquivos e os tempos médios de execução para cada função de ordenação, bem como os gráficos referentes à medição temporal.

Tabela 1: Vetor com dados ordenados crescentemente

Tamanho	100	500	1000	5000	10000
Bubble Sort padrão	$0,080ms$	$1,491ms$	$3,144ms$	$77,359ms$	$323,307ms$
1ª melhoria implementada	$0,044ms$	$0,730ms$	$1,597ms$	$39,071ms$	$169,857ms$
2ª melhoria implementada	$0,005ms$	$0,005ms$	$0,005ms$	$0,018ms$	$0,036ms$

Tabela 2: Vetor com dados ordenados decrescentemente

Tamanho	100	500	1000	5000	10000
Bubble Sort padrão	$0,076ms$	$0,896ms$	$3,533ms$	$80,273ms$	$330,510ms$
1ª melhoria implementada	$0,042ms$	$0,463ms$	$1,899ms$	$42,055ms$	$171,427ms$
2ª melhoria implementada	$0,010ms$	$0,094ms$	$0,458ms$	$6,810ms$	$28,327ms$

Tabela 3: Vetor com dados aleatorizados

Tamanho	100	500	1000	5000	10000
Bubble Sort padrão	$0,077ms$	$0,8119ms$	$3,302ms$	$85,305ms$	$329,405ms$
1ª melhoria implementada	$0,043ms$	$0,424ms$	$1,667ms$	$43,226ms$	$174,482ms$
2ª melhoria implementada	$0,012ms$	$0,071ms$	$0,263ms$	$7,388ms$	$34,347ms$

Tabela 4: Vetor com dados aleatorizados com repetição

Tamanho	100	500	1000	5000	10000
Bubble Sort padrão	$0,082ms$	$0,816ms$	$3,199ms$	$82,855ms$	$338,920ms$
1ª melhoria implementada	$0,044ms$	$0,421ms$	$1,657ms$	$42,569ms$	$179,879ms$
2ª melhoria implementada	$0,012ms$	$0,070ms$	$0,261ms$	$7,709ms$	$34,248ms$

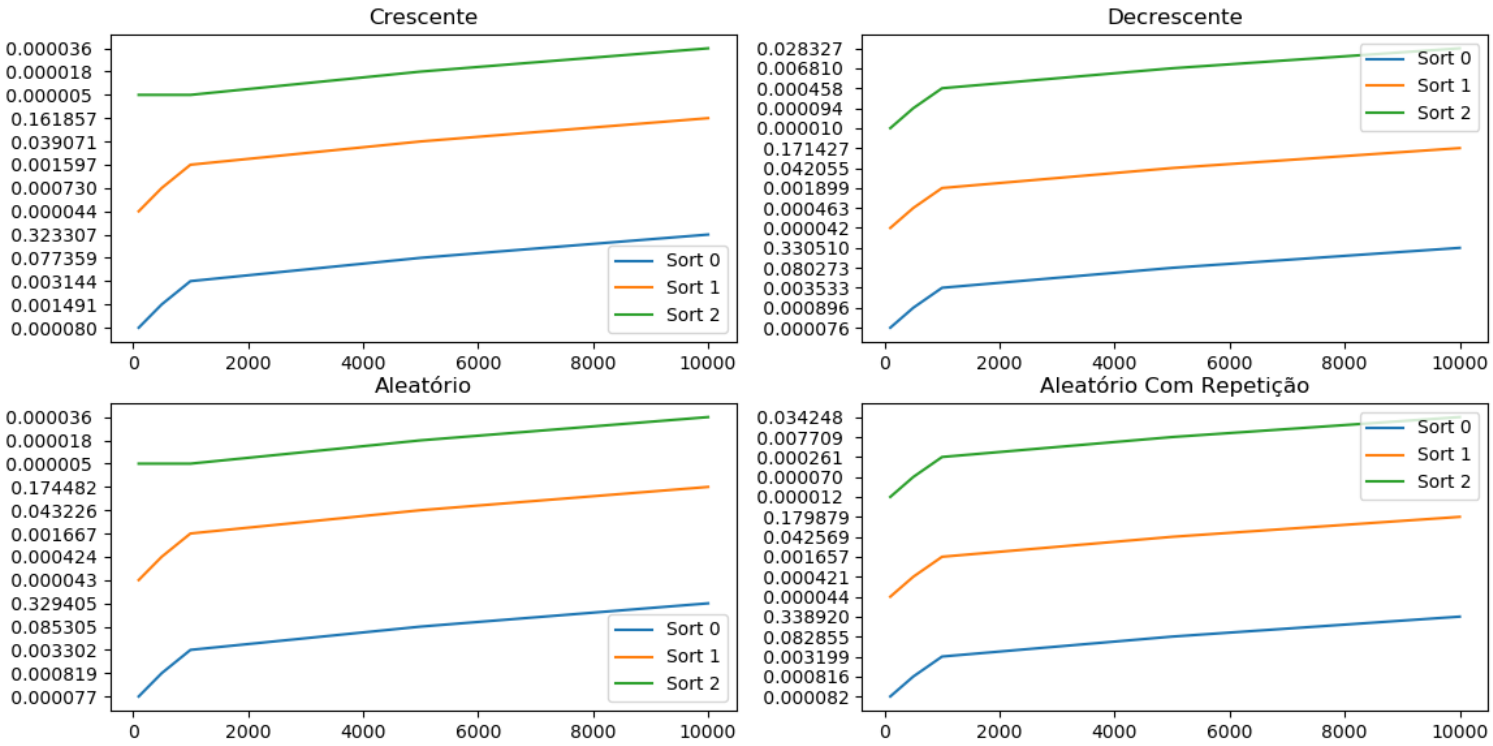

Figura 1: tamanho de entrada X tempo de execução

Analisando-se os gráficos e as tabelas, fica evidente como as melhorias propostas melhoram a otimização do algoritimo, contudo, não mudam a sua complexidade, isto é, os gráficos crescem da mesma forma, mas as melhorias deixam as execuções mais rápidas. Além disso, é possivel notar que os gráficos não evidenciam o crescimento quadrático do tempo em nenhuma situação, isso se deve à pequena quantidade de pontos escritos nele, além do crescimento quase que exponencial das quantidades de dados analisados. Por fim, vale notar que existem variáveis que não podem ser controladas durante a execução dos programas, como memória do computador em que os códigos são executados, entre outros fatores e que as tabelas e os gráficos representam apenas uma execução específica, outras execuções gerariam outros dados, próximos a esses, mas diferentes.

Também foram analisados os tempos de execução dos 4 algoritmos forncedios no enunciado do trabalho. Observando os tempos de execução fornecidos para diferentes tipos de entradas, podemos tirar algumas conclusões:

Inteiros Ordenados Crescentemente:

- Todos os algoritmos têm um aumento significativo no tempo de execução à medida que o tamanho da entrada aumenta.
- O Algoritmo A apresenta um tempo de execução extremamente alto para tamanhos de entrada maiores, indicando uma complexidade quadrática.
- Os Algoritmos B e D têm tempos de execução mais razoáveis, com uma tendência de aumento gradual à medida que o tamanho da entrada aumenta.
- O Algoritmo C tem tempo de execução alto independente do tamanho da entrada, contudo, seu crescimento é linear.

Inteiros Ordenados Decrescentemente:

- Os padrões de desempenho são semelhantes aos do caso de inteiros ordenados crescentemente, com o Algoritmo A apresentando um tempo de execução muito alto.
- Os Algoritmos B e D mantêm tempos de execução razoáveis, com o Algoritmo B sendo especialmente eficiente.
- O algoritmo C segue o mesmo padrão de crescimento linear, mas com tempos de execução altos para pequenas entradas.

Inteiros Aleatorizados e Inteiros Aleatorizados com Repetição:

- Os padrões de desempenho são consistentes com os casos de inteiros ordenados, com o Algoritmo A apresentando tempos de execução desproporcionalmente altos em comparação com os outros algoritmos.
- Os Algoritmos B e D mantêm tempos de execução razoáveis, com o Algoritmo B frequentemente sendo o mais eficiente.
- O Algorimo C segue o padrão já relatado, mostrando que provavelmente tem complexidade de ordem n, mas que as suas constantes na função de complexidade são muito altas.

Conclusão a respeito dos 4 algoritmos fornecidos no enunciado:

- O Algoritmo A parece ter uma complexidade de tempo significativamente pior do que os outros algoritmos em todos os casos de teste, com tempos de execução que aumentam rapidamente com o tamanho da entrada.

- Os Algoritmos B, C e D parecem ter uma performance mais consistente e eficiente em relação ao tamanho da entrada e ao tipo de dados.

- Considerando apenas os tempos de execução, pode-se inferir que o Algoritmo A tem uma complexidade de $O(n^2)$, visto que quando o tamanho da entrada aumenta em 10 vezes, o tempo aumenta por volta de 100 vezes (10^2), enquanto os Algoritmos B, C e D apresentam desempenho mais equilibrado, tendo complexidade de $O(n)$, visto que o tamanho das entradas aumenta proporcinalmente ao tempo. Contudo, é válido notar que, para o estudo específico deste trabalho (entradas variando apenas até 10000), o Algoritmo C chega a ser, para alguns casos, o pior dos três, tendo tempo de execução inclusive pior que o do Algoritimo A para pequenas entradas. Isso se deve, provavelmente, às constantes que regem sua função linear de crescimente (coeficientes angular e linear).

3 Conclusão

A partir das análises teóricas e empíricas realizadas neste trabalho, podemos chegar a algumas conclusões importantes sobre os algoritmos de ordenação estudados.

Primeiramente, ao analisar as diferentes variações do algoritmo Bubble Sort, observamos que a implementação de otimizações pode resultar em melhorias significativas no desempenho do algoritmo. A introdução de uma flag para verificar se houve trocas no vetor e a redução do número de comparações em cada iteração do loop foram estratégias eficazes para diminuir o tempo de execução.

Além disso, ao comparar os quatro algoritmos fornecidos no enunciado, foi possível observar que o Algoritmo A apresentou um crescimento consistentemente pior em todos os casos de teste. Isso sugere que esse algoritmo possui uma complexidade de tempo significativamente maior do que os outros. Por outro lado, os Algoritmos B, C e D demonstraram ter uma performance mais estável e eficiente em relação ao tamanho da entrada, apesar do algotritmo C possuir tempos de execução piores os demais para pequnas entradas. Desta forma, conclui-se que para pequenos valores de entradas, o algoritmo C seria o pior, entretanto, para problemas com mais entradas, o algoritmo A seria consideravelmente pior.

Portanto, constata-se que a escolha do algoritmo de ordenação adequado depende das características do conjunto de dados e dos requisitos de desempenho. Para conjuntos de dados pequenos ou quase ordenados, algoritmos mais simples como o Bubble Sort podem ser suficientes. No entanto, para conjuntos de dados maiores, o fato de tratar-se de um algoritmo $O(n^2)$ pode tornar o processo lento e indesejável.

Essas conclusões destacam a importância de considerar não apenas a complexidade teórica dos algoritmos, mas também sua eficiência na prática em diferentes cenários de uso.

Como um adendo à conclusão, pontuam-se aqui algumas dificuldades encontradas pela equipe ao realizar o trabalho:

- Analisar corretamente os gráficos das três diferentes versões de bubble sort, uma vez que visualmente eles não aparentam ser de caráter quadrático. Entretanto, foi concluido que isso se deve à baixa quantidade de dados fornecidos ao gráfico e que, de fato, analisando-se os pontos do gráfico, as funções possuem $\mathrm{O}(n^2)$.

- Definir precisamente as funções que regem a complexidade dos códigos implementados pela equipe.

- Definir as complexidades no pior caso (big-**O**) para os algoritmos fornecidos no enunciado do trabalho também foi um desafio, visto o pequeno número de dados disponíveis. Embora tenham sido realizadas conclusões acerca das complexidades de tais algoritmos, existe a possibilidade de que erros tenham sido cometidos e, para que se afirme acertivamente seus valores de big-**O**, seria necessário um maior número de amostras, bem como arquivos com elevada quantidade de elementos a serem ordenados (possibilitando uma melhor análise assintótica).