Algoritmos de Ordenação

José Carlos Zancanaro[1], Lucas Cardone[2]

Bacharelado em Ciência da Computação – Universidade do Vale do Itajaí (UNIVALI) – 88.302-901 – Itajaí – SC – Brasil

jose.zancanaro@hotmail.com[1], lucascardone@hotmail.com[2]

Abstract. *This subject show us the commonly used sorting algorithms, we use them to order numbers, names. We always deal with them when we access a social network and even when we have our contact list. We will know in this article the most used sorting algorithms and their performance..*

Resumo. *Este trabalho nos mostra os algoritmos de ordenação usados comumente, nós os usamos para ordenar números e nomes. Nos lidamos com eles ao acessar uma rede social e até quando temos uma lista de contatos. Conheceremos através deste artigo os algoritmos mais e suas performances.*

1. Introdução

Os algoritmos de ordenação são ferramentas utilizadas para dar ordem (no sentido de colocar valores sequencialmente, estabelecidos por algum parâmetro, como por exemplo, números em ordem crescente ou decrescente) à estruturas.

É muito comum utilizar estes algoritmos de ordenação em estruturas conhecidas como arrays, que são também conhecidos como matrizes ou vetores. Estas estrutura são capazes de armazenar valores, em diversas dimensões, aderindo um índice para cada elemento dentro desta estrutura. Por exemplo, imaginemos que uma rua é um array, nesta rua, as casas seriam armazenadas sequencialmente e seriam indexadas, assim, quando você precisasse acessar ou redigir à uma casa, não precisaria lembrar como a casa era ou qual o valor dela, porque acessaria através do índice dela.

Este trabalho explicará e mostrará os diversos meios para ordenar um array, isto é, demonstrará diversos algoritmos capazes de estabalecer uma ordem comum entre todos os elementos dos arrays. Estes algoritmos conseguem-se diferenciar-se através do tipo de estrutura que utilizam para repetição, como por exemplo, recursividade ou laços de repetição; e, através da lógica que utilizam para ordenar os arrays.

É importante analisar e ter em mente o desempenho de cada algoritmo de ordenação, pois cada um deles é perito em certas situações.

4. Algoritmos de Ordenação

4.1 Bubble Sort

O Bubble Sort é o algoritmo de ordenação mais simples desenvolvido que, conforme sugerido em seu nome, repetidamente troca os valores adjacentes caso estejam em ordem incorreta, isto é, se não estiverem corretamente posicionados em ordem crescente ou decrescente.

A lógica do Bubble Sort é, em suma, analisar todos os pares disponíveis para cada elemento disponível no array. Isto significa que o algoritmo necessitará de dois laços para repetição, sendo o primeiro deles substituível por recursividade. A intenção do primeiro laço é repetir a análise de pares do segundo laço para todos os valores disponíveis, certificando que nenhum valor será abandonado em ordem incorreta. Abaixo, um exemplo de como o algoritmo pode ser escrito:

```
void bubble_sort(int array[], int tamanhoVetor)
{
    int auxiliar = 0;
    for (int x = 0; x < tamanhoVetor; x++) {
        for (int indice = 0; indice < tamanhoVetor - x - 1; indice++) {
            if (array[indice] > array[indice + 1]) {
                auxiliar = array[indice];
                array[indice] = array[indice + 1];
                array[indice + 1] = auxiliar;
            }
        }
    }

}
```

O segundo laço não necessariamente precisa analisar todos os valores do array em todas repetições do primeiro laço, já que o maior ou menor número será automaticamente jogado para a última posição até pelo menos o final da execução do segundo laço. Por isto, podemos excluir os índices adjacentes.

Exemplo:

1. [7, 6, 4, 10]
   1.1. [**7**, **6**, 4, 10] -> [**6**, **7**, 4, 10]
   1.2. [6, **7**, **4**, 10] -> [6, **4**, **7**, 10]
   1.3. [6, 4, **7**, **10**] -> [6, 4, **7**, **10**]
2. [6, 4, 7, 10]
   2.1. [**6**, **4**, 7, 10] -> [**4**, **6**, 7, 10]

Dependendo dos elementos do array, é possível que o primeiro laço não precise repetir até o final de seu contador, como pudemos perceber no exemplo anterior, onde o array foi corretamente ordenado até pelo menos a metade das repetições disponíveis, portanto, para otimizar o algoritmo poderíamos adicionar uma variável boolean para validar se o segundo laço realizou alguma troca. Caso não tenha realizado mudança nenhuma ao final do segundo laço, poderíamos simplesmente quebrar a repetição do primeiro laço e, desta maneira, economizar tempo, que pode ser exponencialmente significativo em relação à quantidade de elementos que o array dispõe.

O bubblesort é bastante útil para arrays até uma determinada quantidade de elementos, visto que sua curva entre tempo e quantidade de elementos é signficativamente notável, isto é, se a quantidade de elementos for relativamente pequena, o bubblesort fará seu

trabalho muito bem em pouco tempo, contudo, quanto maior esta quantidade, o algoritmo tende a ficar consideravelmente lento, principalmente se o nível de complexidade estiver no seu pior caso, que seriam todos os elementos ordenados na ordem inversa da desejada. Ademais, o algoritmo, por sua simplicidade, é altamente útil em meio didático.

4.2 Insertion Sort

Com uma dificuldade intermediária, o Insertion Sort é capaz de trazer uma aproximação melhor de ordenação, visto que é efetivamente mais rápido que o bubble sort e não tende a demorar tanto quanto em arrays mais numerosos.

É como ordenamos as cartas de baralho, selecionamos uma carta (com início na segunda carta, portanto segundo elemento), e verificamos se a carta anterior é menor que a carta que selecionamos, caso positivo, a carta com maior valor (ordenando em ordem crescente, neste exemplo), é jogada para a posição a frente e ao final, como não há mais cartas, sabemos que a carta que selecionamos tem um valor menor que a carta a sua direita, portanto, posicionamos-a no início do conjunto.

```
void insertion_sort(int array[], int tamanhoVetor)
{
    int posicao, key, posicaoAnterior;
    for (posicao = 1; posicao < tamanhoVetor; posicao++)
    {
        key = array[posicao];
        posicaoAnterior = posicao - 1;
        while (posicaoAnterior >= 0 && array[posicaoAnterior] > key)
        {
            array[posicaoAnterior + 1] = array[posicaoAnterior];
            posicaoAnterior--;
        }
        //se o loop while atingir à extrema esquerda (isto é, decresceu
até -1), é porque todos os valores analisados são maiores que o pivô,
portanto devemos salvar no início do vetor (0) o valor do pivô.
        array[posicaoAnterior + 1] = key;
    }
}
```

Com ênfase no teste condicional do while, é importante assimilar que o mesmo é responsável por determinar o ciclo de análise de um pivô, pois saberemos que se o loop chegar até o primeiro elemento é porque o pivô escolhido é menor que todos anteriores à ele, e se parar antes disso é porque encontrou um valor que é menor que ele e, portanto, salvará o seu valor na posição subsequente.

Exemplo:

1. [15, **14**, 9, 11] (pivô começa sempre no segundo elemento -> 14).
   1.1. [**15**, **14**, 9, 11] -> [**15**, **15**, 9, 11] (elemento maior que pivô é copiado para a direita)
   1.2. [**15**, 14, 9, 11] -> [**14**, 15, 9, 11] (nenhum elemento à esquerda, posicionar pivô)
2. [14, 15, **9**, 11]
   2.1. [14, **15**, **9**, 11] -> [14, **15**, **15**, 11] (copiar o valor para o próximo elemento)
   2.2. [**14**, 15, 15, 11] -> [**14**, **14**, 15, 11] (copiar o valor para o próximo elemento)
   2.3. [**14**, 14, 15, 11] -> [**9**, 14, 15, 11] (nenhum elemento à esquerda, posicionar pivô)
3. [9, 14, 15, **11**]

3.1. [9, 14, **15**, **11**] -> [9, 14, **15**, **15**] (copiar o valor para o próximo elemento)
3.2. [9, **14**, 15, 15] -> [9, **14**, **14**, 15] (copiar o valor para o próximo elemento)
3.3. [**9**, 14, 14, 15] -> [9, **11**, 14, 15] (o elemento 9 é menor que o pivô, portanto, ele posicionará o pivô na última posição antes de encontrar este elemento).

O insertion sort é, todavia, preferencialmente utilizado para pequenos arrays, devido a quantidade de trocas realizadas (cópias para a posição subsequente), contudo, seu desempenho é suficientemente alto em relação à complexidade do algoritmo. Em outras palavras, a complexidade é baixa (um pouco maior que a do bubble sort) e o desempenho é satisfatório. Existem, ainda, métodos que utilizam busca binária para encontrar a melhor posição para o pivô, solucionando portanto o problema das diversas trocas realizadas na execução.

4.3 Quick Sort

Aumentando o padrão de complexidade dos algoritmos, o Quick Sort segue a lógica de utilização dos pivôs. O quick sort, entretanto, é naturalmente uma função recursiva, apesar de ser possível desenvolver o algoritmo utilizando formas iterativas.

```
void troca(int* a, int* b)
{
    int aux = *a;
    *a = *b;
    *b = aux;
}

int fracao(int array[], int low, int high)
{
    int pivot = array[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
    {
        if (array[j] <= pivot)
        {
            i++;
            troca(&array[i], &array[j]);
        }
    }
    troca(&array[i + 1], &array[high]);
    return (i + 1);
}

void quick_sort(int array[], int low, int high)
{
    if (low < high)
    {
        int meio = fracao(array, low, high);
        quick_sort(array, low, meio - 1);
        quick_sort(array, meio + 1, high);
    }
}
```

Através do quick sort, o array será formatado em grupos de menores e maiores e, nestes, outros grupos de menores e maiores, até que o método fracao tenha apenas dois números para analisar e, portanto, retorne o valor 0, impedindo que a recursão prossiga inifinitamente.

Exemplo:

1. [20, 95, 40, 100, 70, 80, **90**] (pivô iniciado sempre na última posição)
   1.1. [**20**, 95, 40, 100, 70, 80, **90**] -> [**20**, 95, 40, 100, 70, 80, **90**]
   i = 0; j = 0
   Como 20 é menor que 90, o elemento 20 é trocado da posição i para j, contudo, como os dois são iguais no início, não teremos mudança.
   1.2. [20, **95**, 40, 100, 70, 80, **90**] -> [20, **95**, 40, 100, 70, 80, **90**]
   i = 0; j = 1
   95 é maior que 90, então não mudaremos nada.
   1.3. [20, 95, **40**, 100, 70, 80, **90**] -> [20, **40**, **95**, 100, 70, 80, **90**]
   i = 1; j = 2
   90 é maior que 40, portanto trocaremos o elemento em i pelo elemento em j.
   1.4. [20, 40, 95, **100**, 70, 80, **90**] -> [20, 40, 95, **100**, 70, 80, **90**]
   i = 1; j = 3
   90 é menor que 100, então não mudaremos nada.
   1.5. [20, 40, 95, 100, **70**, 80, **90**] -> [20, 40, **70**, 100, **95**, 80, **90**]
   i = 2; j = 4
   90 é maior que 70, então trocamos o elemento em i pelo elemento em j.
   1.6. [20, 40, 70, 100, 95, **80**, **90**] -> [20, 40, 70, **80**, 95, **100**, **90**]
   i = 3; j = 5
   90 é maior que 80, portanto trocamos os valores dos elementos i e j.
   1.7. [20, 40, 70, 80, 95, 100, 90] -> [20, 40, 70, 80, **90**, 100, **95**]
   i = 3; j = 5
   Após o final do loop, os valores dos elementos i+1 e do pivô são invertidos automaticamente.
2. [20, 40, 70, **80**] (sortear a fração esquerda, 80 é o pivô)
   2.1. [20, 40, 70, 80] -> [20, 40, 70, 80]
   Como o array já está sorteado em ordem crescente, sabemos que o loop será realizado, mas não mudará nada.
3. [100, **95**] (sortear a fração direita, 90 é o pivô)
   3.1. [**100**, **95**] -> [**100**, **95**]
   i = 4; j = 5
   Como o pivô é menor que 100, nenhuma mudança ocorre.
   3.2. [**100**, **95**] -> [**95**, **100**]
   Ao final do loop, os valores de i+1 e do pivô são invertidos.
4. [20, 40, 70, 80, 90, 95, 100] (array corretamente ordenado).

O quick sort é um método inteligente e rápido de realizar a ordenação do array, com pouca influência da quantidade de elementos em questão de velocidade, apesar de existir. Contudo, apesar de rápido, o quick sort é complexo devido à recursividade e pode não ser muito amigável para ser implementado.

4.4 Merge Sort

O Merge Sort é um algoritmo complexo de ordenação que visa dividir o array ao meio até que sobre apenas um ou dois números para conferência de ordem, em seguida unindo-os no array novamente.

O algoritmo é dividido em dois procedimentos: o primeiro deles, “merge”, responsável por toda a parte de ordenação das partes fornecidas, o procedimento cria dois vetores (para que fosse possível trabalhar com tamanhos dinâmicos) que armazenam as duas partes fornecidas, para futura ordenação; o segundo deles “merge_sort”, que é responsável pela parte de recursividade em si, ele deve dividir o array apropriadamente, solicitar a ordenação das duas metades e, ao final, uní-las novamente ao array inicial, resultando no array ordenado em ordem crescente ou decrescente.

```
void merge(int array[], int inicio, int metade, int fim)
{
    int i, j, k;
    int n1 = metade - inicio + 1;
    int n2 = fim - metade;
    vector<int> L;
    vector<int> R;
    for (i = 0; i < n1; i++)
        L.push_back(array[inicio + i]);
    for (j = 0; j < n2; j++)
        R.push_back(array[metade + 1 + j]);
    i = 0;
    j = 0;
    k = inicio;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            array[k] = L[i];
            i++;
        }
        else
        {
            array[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        array[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        array[k] = R[j];
        j++;
        k++;
    }
}
void merge_sort(int array[], int inicio, int fim)
{
    if (inicio < fim)
    {
        int metade = inicio + (fim - inicio) / 2;
        merge_sort(array, inicio, metade);
        merge_sort(array, metade + 1, fim);
        merge(array, inicio, metade, fim);
    }
}
```

Exemplo:

1. [43, 32, 59, 8, 17, 70, 9] (divisão do array até as menores partes (2 ou 1))
   1.1.[43, 32, 59, 8] e [17, 70, 9]
   1.2.[43, 32] [59, 8] e [17, 70] [9]
   1.3.[43] [32] [59] [8] E [17] [70] [9] (após a divisão completa, começa a união e ordenação)

   Primeiro array

   1.3.1. [32] e [8] -> [8]
   1.3.2. [32] e [59] -> [8, 32]
   1.3.3. [59] e [43] -> [8, 32, 43, 59]

   Segundo array

   1.3.4. [17] e [9] -> [9]
   1.3.5. [17] e [70] -> [9, 17, 70]

   União através da comparação entre os elementos

   1.3.6. [8, 32, 43, 59] e [9, 17, 70]
   1.3.7. [8] e [9] -> [8]
   1.3.8. [9] e [32] -> [8, 9]
   1.3.9. [32] e [17] -> [8, 9, 17]
   1.3.10. [32] e [70] -> [8, 9, 17, 32]
   1.3.11. [70] e [43] -> [8, 9, 17, 32, 43]
   1.3.12. [70] e [59] -> [8, 9, 17, 32, 43, 59, 70]
2. [8, 9, 17, 32, 43, 59, 70] (Array ordenado)

5 Metodologia

Os testes foram realizados em um laptop da marca ASUS. A máquina dispõe de um processador i5 6200U, dual core (com dois núcleos virtuais), com um clock médio de 2,8GHz, e memória RAM de 4GB.

O programa utiliza a biblioteca “ctime” para importar os tipos “clock_t” e medir o tempo de execução através da razão entre a subtração do tempo final de execução com o tempo inicial e os clocks por segundo do processador. Isto significa que o tempo médio de execução é demonstrado em segundos.

Os testes foram realizados com diversos tamanhos de arrays (variando da capacidade de cada método de executar sem sobrecarregar o stack) preenchidos com números aleatórios de 0 a 999.

6 Resultados e discussões

Os resultados dos testes com os diversos tamanhos de arrays, foram:

6.1 Bubble Sort:

Array com 1.000 posições: 0,001 segundos.
Array com 2.500 posições: 0,006 segundos.
Array com 10.000 posições: 0,212 segundos.
Array com 100.000 posições: 20,871 segundos.
Array com 250.000 posições: 126,163 segundos.
Array com 500.000 posições: Stack overflow.

6.2 Bubble Sort Recursivo:

Array com 1.000 posições: 0,001 segundos.
Array com 2.500 posições: 0,029 segundos.
Array com 10.000 posições: 0,214 segundos.
Array com 100.000 posições: 20,065 segundos.
Array com 250.000 posições: 128,304 segundos.
Array com 500.000 posições: Stack overflow.

6.3 Insertion Sort:

Array com 1.000 posições: 0,001 segundos.
Array com 2.500 posições: 0,002 segundos.
Array com 10.000 posições: 0,019 segundos.
Array com 100.000 posições: 2,102 segundos.
Array com 250.000 posições: 10,470 segundos.
Array com 500.000 posições: Stack overflow.

6.4 Insertion Sort Recursivo:

Array com 1.000 posições: 0,001 segundos.
Array com 2.500 posições: 0,001 segundos.
Array com 10.000 posições: 0,021 segundos.
Array com 50.000 posições: 0,699 segundos.
Array com 100.000 posições: Stack overflow.

6.5 Quick Sort:

Array com 1.000 posições: 0 segundos.
Array com 2.500 posições: 0 segundos.
Array com 10.000 posições: 0,001 segundos.
Array com 100.000 posições: 0,016 segundos.
Array com 250.000 posições: 0,060 segundos.
Array com 500.000 posições: Stack overflow.

6.6 Merge Sort:

Array com 1.000 posições: 0,001 segundos.
Array com 2.500 posições: 0,002 segundos.
Array com 10.000 posições: 0,006 segundos.
Array com 100.000 posições: 0,085 segundos.
Array com 250.000 posições: 0,259 segundos.
Array com 500.000 posições: Stack overflow.