**Trabalho Ordenação**

**Brunno Tatsuo**

Engenharia de Software

Pontifícia Universidade Católica do Paraná (PUCPR) – Curitiba, PR – Brasil

cruz.brunno@pucpr.edu.br

**Abstract.** This report describes the performance evaluation of a sorting algorithm on arrays of integers, with array sizes of 50, 500, 1000, 5000, and 10,000 elements. The main evaluation criteria included execution time, the number of swaps, and the number of iterations. To obtain reliable results, at least 5 rounds of experiments were conducted for each array size.

**Resumo.** Este relatório descreve a avaliação do desempenho de um algoritmo de ordenação em vetores de inteiros, com tamanhos de vetor de 50, 500, 1000, 5000 e 10.000 elementos. Os principais critérios de avaliação incluíram o tempo de execução, o número de trocas e o número de iterações. Para obter resultados confiáveis, foram realizadas pelo menos 5 rodadas de experimentos para cada tamanho de vetor.

1. **Introdução**

Algoritmos de ordenação são uma parte fundamental da ciência da computação e desempenham um papel essencial em inúmeras aplicações, desde sistemas de gerenciamento de bancos de dados até algoritmos de busca e processamento de dados. A ordenação é o processo de organizar um conjunto de elementos em uma sequência específica, seja em ordem crescente, decrescente ou baseada em critérios personalizados. A eficiência e o desempenho desses algoritmos são de grande importância, uma vez que a velocidade com que os dados são ordenados pode impactar significativamente o desempenho de sistemas e aplicativos.

O algoritmo de ordenação Bubble Sort é um dos algoritmos de ordenação mais simples e diretos. Ele recebe esse nome devido à forma como os elementos "burbulham" para suas posições finais no vetor à medida que o algoritmo é executado. O Bubble Sort compara repetidamente pares adjacentes de elementos no vetor e troca seus lugares se estiverem na ordem errada. O processo é repetido até que nenhum par de elementos precise mais ser trocado, o que indica que o vetor está ordenado.

Aqui está uma descrição passo a passo do Bubble Sort:

1. Percorra o vetor a ser ordenado a partir do primeiro elemento (índice 0) até o penúltimo elemento (índice n-1).
2. Para cada elemento, compare-o com o próximo elemento. Se o elemento atual for maior do que o próximo elemento, faça uma troca entre eles.
3. Continue percorrendo o vetor até o final, repetindo as comparações e trocas, de forma que o maior elemento "suba" para a última posição.
4. Após a primeira passagem completa pelo vetor, o maior elemento estará na última posição.
5. Repita os passos 1 a 4 para o subvetor que exclui o último elemento (pois ele já está na posição correta) e continue esse processo até que o vetor inteiro esteja ordenado.
6. O algoritmo termina quando nenhuma troca é feita em uma passagem completa pelo vetor, o que indica que o vetor está ordenado.

O Bubble Sort é fácil de entender e implementar, mas não é eficiente para vetores grandes, pois sua complexidade média e pior caso é O(n^2), onde "n" é o número de elementos no vetor. Isso o torna inadequado para grandes conjuntos de dados.

O algoritmo de ordenação Shell Sort, é uma variação do algoritmo de inserção que melhora significativamente o desempenho desse último, especialmente em conjuntos de dados grandes. O Shell Sort é classificado como um algoritmo de ordenação por inserção com intervalos, onde os elementos são comparados e trocados em incrementos específicos em vez de apenas um por vez.

Aqui está uma descrição passo a passo do algoritmo Shell Sort:

1. Comece definindo um intervalo inicial, geralmente chamado de "h" (gap), que é usado para agrupar elementos em diferentes subconjuntos do vetor.
2. Divida o vetor em subvetores de tamanho "h". Cada subvetor é, na verdade, uma sequência de elementos que estão a uma distância "h" um do outro. Isso significa que os elementos "h" índices à parte são agrupados juntos.
3. Aplique o algoritmo de ordenação por inserção em cada subvetor. Isso significa que você classificará os elementos dentro de cada subvetor usando o algoritmo de inserção padrão.
4. Reduza o valor de "h" (gap) para um novo valor menor. Normalmente, o novo valor de "h" é obtido dividindo o valor anterior de "h" por 3 (ou outro fator) e, em seguida, arredondando para baixo. Esse processo é repetido até que "h" se torne 1.
5. Continue aplicando o passo 3 (ordenar subvetores) com o novo valor de "h" até que o vetor inteiro esteja ordenado.

O Shell Sort é eficaz porque permite que elementos distantes se movam para suas posições corretas mais rapidamente do que o algoritmo de inserção padrão. À medida que o valor de "h" diminui, o algoritmo se aproxima do algoritmo de inserção, mas a maioria dos elementos já está próxima de suas posições corretas, reduzindo o número de trocas necessárias.

O desempenho do Shell Sort depende da escolha dos intervalos (valores de "h"). Diferentes sequências de intervalos podem afetar a eficiência do algoritmo. Embora o Shell Sort seja mais eficiente do que algoritmos quadráticos, como o Bubble Sort e o Insertion Sort, ainda pode ser superado por algoritmos mais avançados, como o Quick Sort e o Merge Sort, em conjuntos de dados muito grandes.

O algoritmo Quick Sort é um dos algoritmos de ordenação mais eficientes e amplamente utilizados. Ele foi desenvolvido por Tony Hoare em 1960 e é conhecido por sua velocidade e desempenho em uma ampla variedade de cenários. O Quick Sort é um algoritmo de divisão e conquista que classifica um vetor dividindo-o em subvetores menores, ordenando esses subvetores e, finalmente, combinando-os em uma única sequência ordenada.

Aqui está uma descrição passo a passo do algoritmo Quick Sort:

1. Escolha um elemento do vetor como o "pivô". O pivô é um elemento de referência que será usado para dividir o vetor em dois subvetores: um contendo elementos menores que o pivô e outro contendo elementos maiores que o pivô.
2. Reorganize o vetor de modo que todos os elementos menores que o pivô fiquem à sua esquerda e todos os elementos maiores fiquem à sua direita. Isso é chamado de particionamento.
3. Agora, o vetor é dividido em duas partes: um subvetor à esquerda do pivô e outro à direita.
4. Repita os passos 1 a 3 para os subvetores esquerdo e direito. Isso é feito recursivamente até que os subvetores sejam pequenos o suficiente para serem considerados ordenados.
5. A combinação dos subvetores menores em um único vetor ordenado é a etapa final do algoritmo. Como os subvetores menores já estão ordenados, a fusão é uma operação simples.

O sucesso do Quick Sort se deve à escolha eficiente do pivô e ao particionamento dos elementos, que permite que muitos elementos sejam movidos para suas posições corretas antes de passar para as etapas recursivas. Em média, o Quick Sort tem uma complexidade de tempo de O(n log n), tornando-o muito rápido na ordenação de grandes conjuntos de dados.

No entanto, é importante observar que o desempenho do Quick Sort pode ser influenciado pela escolha do pivô. Se o pivô for sempre o maior ou o menor elemento do vetor, o desempenho pode degradar para uma complexidade de tempo de O(n^2), embora métodos de escolha de pivô eficazes, como a escolha do pivô mediano, possam ser usados para mitigar esse problema.

1. **Código**

**Classe Main**

public class Main {

public static void main(String[] args) {

int tamanho = 5000;//50,500,1000,5000,10000

int seed =81;//24,33,50,12,81

Lista lista1 = new Lista(tamanho,seed);

lista1.imprimir();

long startB = System.nanoTime();

lista1.bubbleSort();

long endB = System.nanoTime();

System.out.println("Tempo de Execução: "+(endB-startB)+" nanosegundos"+"\nNúmero de Trocas: "+lista1.getTrocas()+"\nNumero de Interações: "+lista1.getPasso());

lista1.imprimir();

System.out.println();

Lista lista2 = new Lista(tamanho,seed);

lista2.imprimir();

long startS = System.nanoTime();

lista2.shellSort();

long endS = System.nanoTime();

System.out.println("Tempo de Execução: "+(endS-startS)+" nanosegundos"+"\nNúmero de Trocas: "+lista2.getTrocas()+"\nNumero de Interações: "+lista2.getPasso());

lista2.imprimir();

System.out.println();

Lista lista3 = new Lista(tamanho,seed);

lista3.imprimir();

long startQ = System.nanoTime();

lista3.quickSort(0,tamanho-1);

long endQ = System.nanoTime();

System.out.println("Tempo de Execução: "+(endQ-startQ)+" nanosegundos"+"\nNúmero de Trocas: "+lista3.getTrocas()+"\nNumero de Interações: "+lista3.getPasso());

lista3.imprimir();

}

}

**Classe Lista**

import java.util.Random;

public class Lista {

private int[] lista;

private int tamanho;

private int passo =0;

private int trocas =0;

public Lista(int tamanho,int seed){

this.lista = new int[tamanho];

this.tamanho = tamanho;

Random rd = new Random(seed);

for(int i=0;i<tamanho;i++){

lista[i]= rd.nextInt(9999);

}

}

public int getPasso() {

return passo;

}

public int getTrocas() {

return trocas;

}

public void imprimir(){

for (int i =0;i<tamanho;i++){

System.out.printf("%7d",lista[i]);

}

System.out.println();

}

public boolean emOrdem(int[] lista){

for(int i=0;i<tamanho-1;i++){

if(lista[i]>lista[i+1]){

return false;

}

}

return true;

}

public void bubbleSort(){

while(emOrdem(lista)==false){

for(int i=0;i<tamanho-1;i++){

if(lista[i]>lista[i+1]){

int temp = lista[i];

lista[i]=lista[i+1];

lista[i+1]=temp;

trocas++;

}

passo++;

}

}

}

public void shellSort(){

for(int incremento = tamanho/2;incremento>0;incremento/=2){

for (int i = incremento;i<tamanho;i++){

passo++;

int temp = lista[i];

int j;

for(j=i; j>=incremento && lista[j-incremento]>temp;j-=incremento){

lista[j] = lista[j-incremento];

trocas++;

}

lista[j] = temp;

}

}

}

public void quickSort(int inferior,int superior){

if (inferior<superior){

passo++;

int indPivo = particao(inferior,superior);

quickSort(inferior,indPivo-1);

quickSort(indPivo+1,superior);

}

}

private int particao(int inferior,int superior){

int pivo = lista[superior];

int i = inferior -1;

for (int j = inferior;j<superior;j++){

if (lista[j]<pivo){

i++;

int temp = lista[i];

lista[i] = lista[j];

lista[j] = temp;

trocas++;

}

passo++;

}

int temp = lista[i+1];

lista[i+1] = lista[superior];

lista[superior] = temp;

trocas++;

return i+1;

}

}

1. **Resultados**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubblesort** | | | **ShellSort** | | | **QuickSort** | | |
| **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** |
| 57600 | 550 | 1960 | 14900 | 164 | 203 | 14800 | 116 | 306 |
| 87800 | 732 | 2352 | 14000 | 166 | 203 | 16200 | 134 | 313 |
| 57600 | 618 | 1911 | 15100 | 198 | 203 | 15900 | 161 | 277 |
| 64000 | 708 | 2205 | 14300 | 158 | 203 | 14700 | 130 | 253 |
| 59200 | 540 | 2058 | 15100 | 135 | 203 | 16900 | 140 | 305 |

**Tabela 01. Tabela das Ordenações com 50 vetores.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubblesort** | | | **ShellSort** | | | **QuickSort** | | |
| **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** |
| 2706900 | 64310 | 243512 | 183600 | 2961 | 3506 | 202900 | 2782 | 5278 |
| 2585500 | 59721 | 238023 | 174200 | 2811 | 3506 | 212900 | 2656 | 5104 |
| 2549800 | 60768 | 228043 | 180800 | 2940 | 3506 | 190600 | 2541 | 5453 |
| 2594800 | 61070 | 245508 | 179900 | 3097 | 3506 | 177300 | 2595 | 5460 |
| 2733600 | 63983 | 245508 | 193700 | 3072 | 3506 | 172300 | 2475 | 5198 |

**Tabela 02. Tabela das Ordenações com 500 vetores.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubblesort** | | | **ShellSort** | | | **QuickSort** | | |
| **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** |
| 4909500 | 253166 | 978021 | 555100 | 7524 | 8006 | 305400 | 5748 | 10766 |
| 4929200 | 259075 | 983016 | 415800 | 7116 | 8006 | 318400 | 5427 | 11153 |
| 4856300 | 246242 | 935064 | 417000 | 7182 | 8006 | 315200 | 7188 | 11483 |
| 4770500 | 250671 | 915084 | 433100 | 7758 | 8006 | 313600 | 6819 | 11866 |
| 4827800 | 247796 | 965034 | 425500 | 8085 | 8006 | 312600 | 5784 | 11174 |

**Tabela 03. Tabela das Ordenações com 1000 vetores.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubblesort** | | | **ShellSort** | | | **QuickSort** | | |
| **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** |
| 43535500 | 6191431 | 24665066 | 2414400 | 61847 | 55005 | 1366000 | 38803 | 74014 |
| 49921400 | 6197710 | 24210157 | 3049000 | 60137 | 55005 | 1034900 | 34936 | 75491 |
| 49174500 | 6198312 | 24460107 | 3086600 | 60027 | 55005 | 1071200 | 35116 | 77779 |
| 44440200 | 6228798 | 24150169 | 2880100 | 59287 | 55005 | 1018700 | 33955 | 77234 |
| 45958600 | 6136147 | 24580083 | 3468900 | 61877 | 55005 | 1213300 | 36474 | 73714 |

**Tabela 04. Tabela das Ordenações com 5000 vetores.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bubblesort** | | | **ShellSort** | | | **QuickSort** | | |
| **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** | **Tempo** | **Trocas** | **Iterações** |
| 182519000 | 24907708 | 97900209 | 3097200 | 141778 | 120005 | 1857500 | 82014 | 161823 |
| 212023500 | 24991851 | 98380161 | 3857200 | 143018 | 120005 | 1971200 | 84233 | 159290 |
| 160681500 | 24672738 | 98350164 | 3759300 | 141245 | 120005 | 1885500 | 82046 | 166108 |
| 171889000 | 24904816 | 98520147 | 3635500 | 156723 | 120005 | 1437700 | 79170 | 157958 |
| 164602100 | 24866039 | 99150084 | 2960100 | 142953 | 120005 | 1564400 | 84564 | 176064 |

**Tabela 05. Tabela das Ordenações com 10000 vetores.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **50** | **500** | **1000** | **5000** | **10000** |
| **Tempo** | 6,52E+04 | 2,63E+06 | 4,86E+06 | 4,66E+07 | 1,78E+08 |
| **Trocas** | 6,30E+02 | 6,20E+04 | 2,51E+05 | 6,19E+06 | 2,49E+07 |
| **Iterações** | 2,10E+03 | 2,40E+05 | 9,55E+05 | 2,44E+07 | 9,85E+07 |

**Tabela 06. Tabela das Médias para BubbleSort.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **50** | **500** | **1000** | **5000** | **10000** |
| **Tempo** | 1,47E+04 | 1,82E+05 | 4,49E+05 | 2,98E+06 | 3,46E+06 |
| **Trocas** | 1,64E+02 | 2,98E+03 | 7,53E+03 | 6,06E+04 | 1,45E+05 |
| **Iterações** | 2,03E+02 | 3,51E+03 | 8,01E+03 | 5,50E+04 | 1,20E+05 |

**Tabela 07. Tabela das Médias para ShellSort.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **50** | **500** | **1000** | **5000** | **10000** |
| **Tempo** | 1,57E+04 | 1,91E+05 | 3,13E+05 | 1,14E+06 | 1,74E+06 |
| **Trocas** | 1,36E+02 | 2,61E+03 | 6,19E+03 | 3,59E+04 | 8,24E+04 |
| **Iterações** | 2,91E+02 | 5,30E+03 | 1,13E+04 | 7,56E+04 | 1,64E+05 |

**Tabela 08. Tabela das Médias para QuickSort.**

1. **Conclusão**

Como já era esperado a Ordenação pelo algoritmo BubbleSort demando mais tempo, realizo mais trocas e mais iterações para conseguir ordenar a lista.

Para listas de 50 e 500 valores, ShellSort foi mais eficiente que a QuickSort, contudo a partir de 1000 valores a QuickSort se tornou mais eficiente. Já que apesar de realizar mais iterações precisa de menos trocas para ordernar a lista. Outro detalhe interessante foi que no ShellSort o número de iterações depende do tamanho da lista, pois mesmo o incremento na hora de ordenar depende do tamanho da lista. Por isso mesmo com listas diferentes o número de iterações sempre era o mesmo.