Universidade Federal de Alagoas
Campus A. C. Simões
Centro de Tecnologia
Programa de Pós-graduação em Engenharia Civil

Weverton Marques da Silva

Lista 1

Maceió/AL
agosto de 2018

Weverton Marques da Silva

Lista 1

Trabalho apresentado ao professor Willian Wagner Matos Lira do Programa de Pós-graduação em Engenharia Civil, do Centro de Tecnologia, da Universidade Federal de Alagoas – UFAL, como requisito parcial para definição de conceito na disciplina de Técnicas Computacionais Avançadas.

Maceió/AL

agosto de 2018

SUMÁRIO

1 Introdução

Implemente vários algoritmos de ordenação de n números, incluindo um algoritmo $O(n^2)$ e um algoritmo O($n$log$n$). Verifique que eles estão corretos, isto é, que funcionam corretamente para qualquer instância do problema (em particular, quando há dados repetidos). Compare os algoritmos implementados usando vários valores para n, por exemplo 100, 1.000, 10.000, 100.000, 1.000.000. Veja como os algoritmos se comportam para dados aleatórios, para conjuntos já ordenados, e para conjuntos ordenados na ordem inversa. Compare os tempos de cada algoritmo e também o esforço necessário para sua implementação. Use gráficos para ilustração dos resultados obtidos. Elaborar e entregar relatório.

2 Implementação

Os algoritmos foram implementados em linguagem C, em que cada função corresponde a um algoritmo de ordenação. A seguir, detalha-se a implementação de cada um dos algoritmos escolhidos. Em todos os algoritmos e implementações, x se refere ao vetor com o conjunto de números a serem ordenados, e n se refere a quantidade de elementos nesse conjunto.

2.1 Selection

A implementação do algoritmo de ordenação por seleção foi implementação bastante direta. No Quadro 2.1.1 tem-se o algoritmo e a implementação em C++.

Quadro 2.1.1 - Método de ordenação *selection sort.*

Algoritmo	Implementação em C++

<pre>para i = 1..n-1
    m = i
    para j = i+1..n
        se x_j < x_m
            então m = j
        troque x_i com x_m</pre>
	<pre>void selectionSort(double *x, int n)
	{
	    for (int i = 0; i < n - 1; i++)
	    {
	        int m = i;
	        for (int j = i + 1; j < n; j++)
	        {
	            if (x[j] < x[m])
	            {
	                m = j;
	            }
	        }
	        double aux = x[i];
	        x[i] = x[m];
	        x[m] = aux;
	    }
	}</pre>

2.2 Merge

Devido à conhecida possibilidade de elevado consumo de memória do algoritmo quando se uso em um conjunto grande de dados, fez-se optou-se por trabalha com endereços de memória (ponteiros), o que acrescentou um grau de dificuldade no processo. Outro ponto de dificuldade foi a implementação dos elementos sentinelas, cuja ideia é fazer com que, no momento da recombinação, quando um dos subconjuntos de elementos já ordenados não tiverem mais elementos a serem reposicionados no novo conjunto ordenado, sejam usados os elementos do outro subconjunto. No Quadro 2.2.1 tem-se o algoritmo e a implementação em C++.

Quadro 2.2.1 - Método de ordenação *merge sort*.

Algoritmo	Implementação em C++

<pre>mergesort(x, n)
    se n < 2 então
        retorne
    m = n/2
    l = x[1..m]
    r = x[m+1..n]
    mergesort(l, m)
    mergesort(r, n-m)
    i = 1
    j = 1
    para k = 1..n
        se li < r_j então
            x_k = l_i;
            i = i+1;
        senão
            x_k = r_j;
            j = j+1;</pre>
	<pre>void mergeSort(double* x, int n)
	{
	    if (n < 2)
	    {
	        return;
	    }
	    int m = n / 2;
	    mergeSort(x, m);
	    mergeSort(x + m, n - m);
	    double *l = new double[m];
	    memcpy(l, x, m*sizeof(double));
	    double *r = new double[n - m];
	    memcpy(r,&x[m],(n-m)*sizeof(double));
	    int i = 0;
	    int j = 0;
	    for (int k = 0; k < n; ++k)
	    {
	        if (l[i] < r[j])
	        {
	            x[k] = l[i];
	            i++;
	            if (i == m)
	            {
	                i--;
	                l[i] = r[n - m - 1];
	            }
	        }
	        else
	        {
	            x[k] = r[j];
	            j++;
	            if (j == n - m)
	            {
	                j--;
	                r[j] = l[m - 1];
	            }
	        }
	    }
	}</pre>

2.3 Quick

Assim como no algorítimo *merge sort*, aqui foi evitado a realocação de memória, optando por fazer modificações "*in place*" no próprio vetor de números a serem ordenados. Para isso, estratégia foi usar o pivô como sendo o primeiro elemento do vetor de números a serem ordenados, . A implementação do algoritmo de ordenação *quick* foi bastante direta. No Quadro 2.3.1 tem-se o algoritmo e a implementação em C++.

Quadro 2.3.1 - Método de ordenação *quick sort*.

<table>
<tr><th>Algoritmo</th><th>Implementação em C++</th></tr>
<tr><td>

```
quicksort(r, s)
    se s ≤ r então retorne
    v = x_r
    i = r
    j = s+1
    repita
        repita i = i+1 até x_i ≥ v
        repita j = j-1 até x_j ≤ v
        troque x_i com x_j
    até j ≤ i
    troque x_i com x_j
    troque x_i com x_s
    quicksort(r, i-1)
    quicksort(i+1, s)
```

</td><td>

```
void quickSort(double *x, int r, int s)
{
   if (s <= r)
   {
      return;
   }
   double pivot = x[r];
   int pivotID = r;
   for (int i = r + 1; i < s; i++)
   {
      if (x[i] < pivot)
      {
         double aux = x[i];
         x[i] = x[pivotID + 1];
         x[pivotID + 1] = pivot;
         x[pivotID] = aux;
         pivotID++;
      }
   }
   quickSort(x, r, pivotID);
   quickSort(x, pivotID + 1, s);
}
```

</td></tr>
</table>

2.4 Insertion sort

A implementação desse algoritmo de ordenação também ocorreu bastante direta. No Quadro 2.4.1 tem-se o algoritmo e a implementação em C++.

Quadro 2.4.1 – Método de ordenação *insertion sort*.

Algoritmo	Implementação em C++

para i = 2..n-1
    v = x_i
    j = i
    enquanto x_j-1 > v
        x_j = x_j - 1
        j = j - 1
    x_j = v
	void insertionSort(double* x, int n)
	{
	    for (int i = 1; i < n; i++)
	    {
	        double v = x[i];
	        int j = i;
	        while (x[j - 1] > v && j > 0)
	        {
	            x[j] = x[j - 1];
	            j--;
	        }
	        x[j] = v;
	    }
	}

2.5 Shell sort

Criado por Donald Shell em 1959, publicado pela Universidade de Cincinnati, o método s*hell sort* é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles[1]. O desempenho deste método depende da forma como subdividi o conjunto de dados a serem ordenados (determinados no algoritmo a seguir pela variável `gap`), por isso está foi a implementação mais complexa. No Quadro 2.5.1 tem-se o algoritmo e a implementação em C++. A complexidade desse algoritmo no pior caso é $O(n^2)$, mas pode ser de $O(n \log n)$ no melhor caso.

[1] https://pt.wikipedia.org/wiki/Shell_sort

Quadro 2.5.1 – Método de ordenação *shell sort*.

Algoritmo

```
gap = 1
Enquanto gap < n
   gap = 3 * gap + 1
Enquanto gap > 1 faça gap = gap/3
   i = gap
   Para i = gap..n
      v = x_i
      j = i
      Enquanto j >= gap e v < x_j-gap
         x_j = x_j-gap
         j = j - gap
      x_j = v
```

Implementação em C++

```cpp
void shellSort(double *x, int n)
{
   int i, j, v;
   int gap = 1;
   while (gap < n)
   {
      gap = 3 * gap + 1;
   }
   while (gap > 1)
   {
      gap /= 3;
      for (i = gap; i < n; i++)
      {
         v = x[i];
         j = i;
         while (j >= gap && v < x[j-gap])
         {
            x[j] = x[j - gap];
            j = j - gap;
         }
         x[j] = value;
      }
   }
}
```

3 Avaliação de performance

Cada um dos métodos implementados, foi realizado um teste de performance, onde se mediu o tempo em segundos para ordenação de um conjunto x com N elementos em três situações:

(1) x com números aleatórios de 1 a N;

(2) x com números de 0 a N-1 em ordem crescente;

(3) x com números de 0 a N-1 em ordem decrescente.

Para os valores de N, foram usando potências de 10, a começar por 10^1 até 10^5. Para medir o tempo gasto na ordenação, foram realizadas em 4 execuções, e tomou-se como referência de comparação a média dos 4 valores. Nos gráficos a seguir, usa-se escala logarítmica no eixo horizontal para melhor visualização dos dados.

3.1 Selection sort

A Figura 3.1, apresenta o tempo gasto pelo algoritmo *selection sort* para ordenação dos diferentes conjuntos de dados.

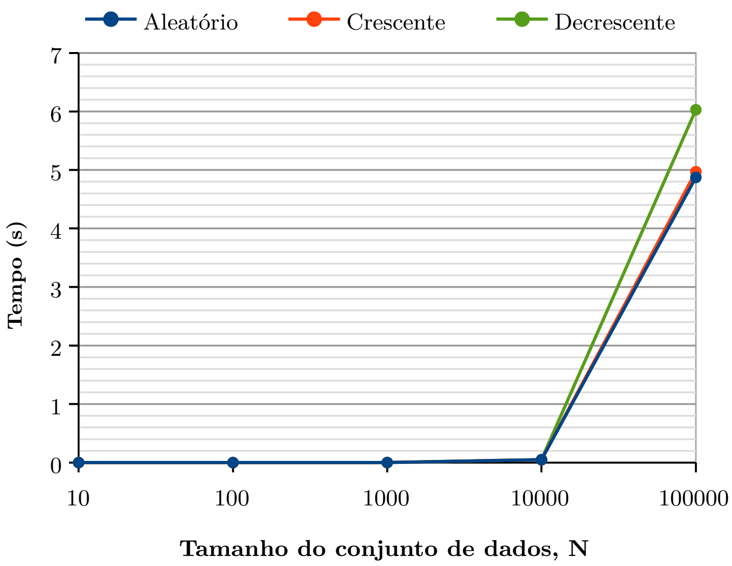

Figura 3.1 – Custo computacional do algoritmo *selection sort*.

É possível observar que, em relação aos casos com dados aleatórios, o desempenho do método não melhora nos casos em que os dados estão previamente ordenados, mas piora no caso os números estejam em ordem inversa. De toda forma, o crescimento do em função do tamanho do conjunto de dados cresce de forma bastante acentuada, o que é esperado, uma vez que a complexidade inerente desse algoritmo é $O(n^2)$.

3.2 Merge sort

A Figura 3.2, apresenta o tempo gasto pelo algoritmo *merge sort* para ordenação dos diferentes conjuntos de dados. É possível ver que o crescimento do tempo é menos acentuado que no caso do *selection sort*, insinuando que esse *merge sort* uma complexidade inferior ao primeiro – o que já é sabido, sendo sua complexidade n*log*n. Vê-se ainda que o pior desempenho é no caso de dados aleatório, mas no não há grandes diferenças no caso de dados ordenados, seja de forma crescente ou decrescente. A complexidade desse algoritmo é $O(n \log n)$..

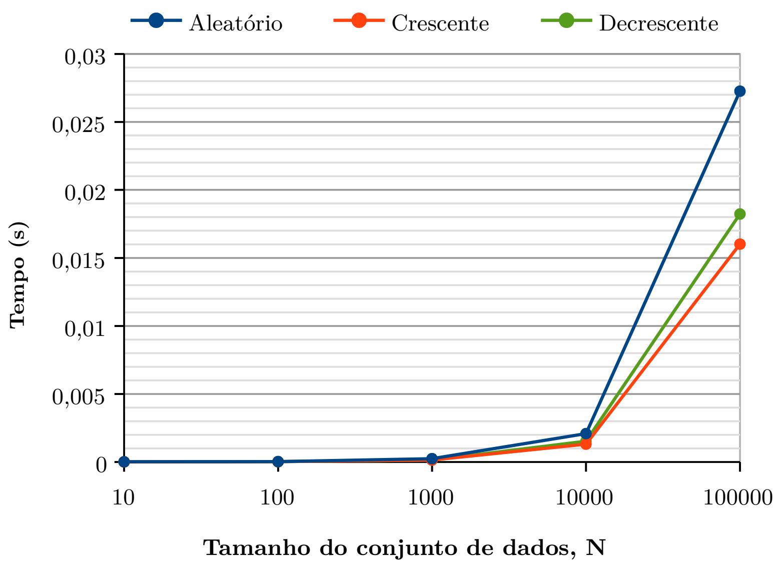

Figura 3.2 – Custo computacional do algoritmo *merge sort*.

3.3 Quick sort

Uma análise do algoritmo permite demonstrar que a complexidade média do algoritmo é $O(n \log n)$ (sob a suposição de que os elementos a serem ordenados são independentemente escolhidos segundo alguma distribuição de probabilidade). No entanto, a complexidade deste algoritmo no pior caso é $O(n^2)$.

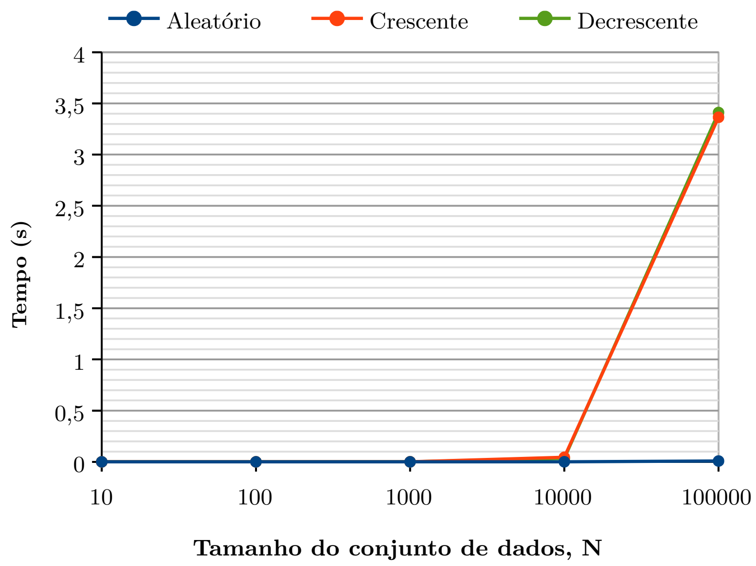

Figura 3.3 – Custo computacional do algoritmo *quick sort*.

Apesar do comportamento quadrático apresentado nos casos em que os dados estão previamente ordenados (crescente ou decrescente), o melhor desempenho do algorítimo ocorreu em conjuntos de elementos aleatórios (vide Figura 3.3).

3.4 Insertion sort

Com um comportamento oposto ao do algoritmo anterior, o *insertion sort* foi muito pior em conjuntos de números aleatórios em relação a conjuntos previamente ordenados (crescente ou decrescente), conforme mostrado na Figura 3.4. A complexidade desse algoritmo é $O(n^2)$.

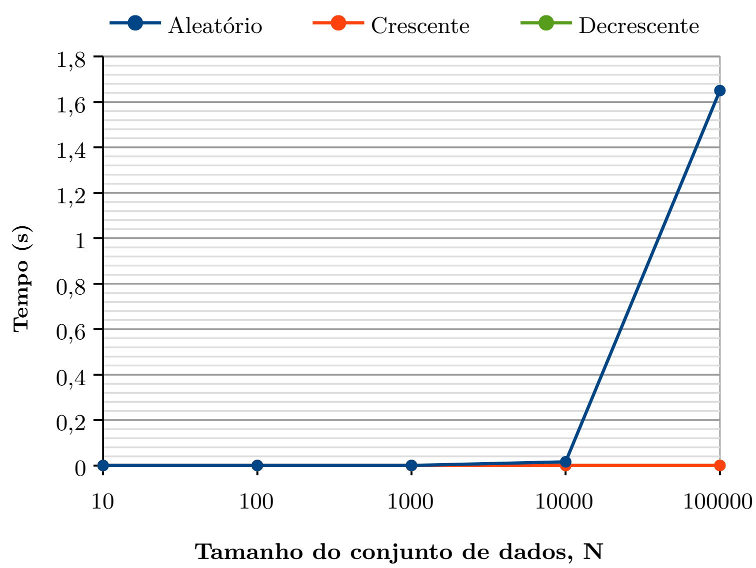

Figura 3.4 – Custo computacional do algoritmo *insertion sort*.

3.5 Shell sort

Com um comportamento semelhante ao algoritmo anterior, o *shell sort* foi muito pior em conjuntos de números aleatórios em relação a conjuntos previamente ordenados (crescente ou decrescente), conforme mostrado na Figura 3.5.

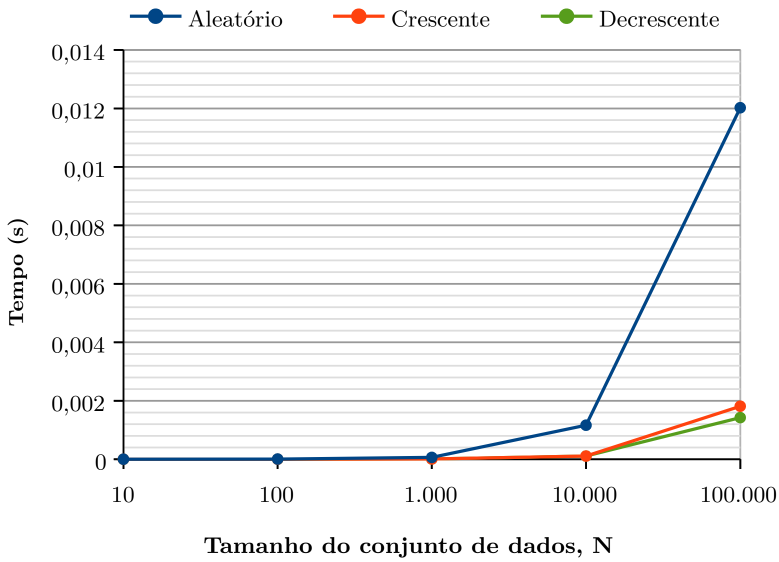

Figura 3.5 – Custo computacional do algoritmo *insertion sort*.

3.6 Comparação entre os métodos

Comparando o desempenho dos métodos uns com os outros para cada caso de ordenação do conjunto de dados – aleatório, crescente e decrescente, tem-se os gráficos apresentados nas figuras a seguir. Nestes gráficos, foram empregados eixos verticais em escala logarítmica para facilitar a visualização dos valores.

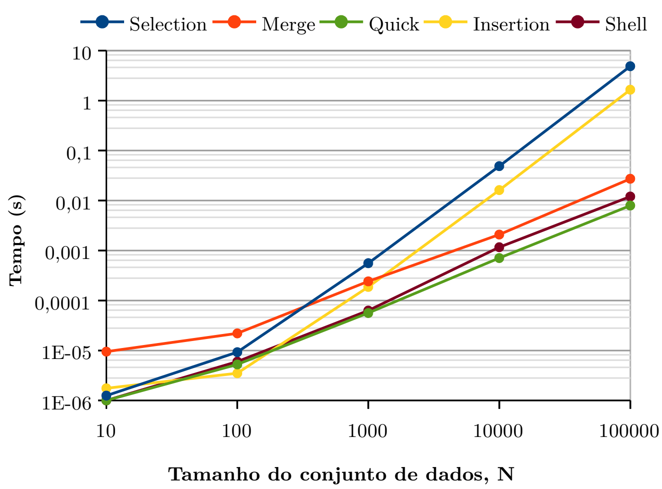

Figura 3.6 – Custo computacional dos algoritmos para conjuntos de números

aleatórios.

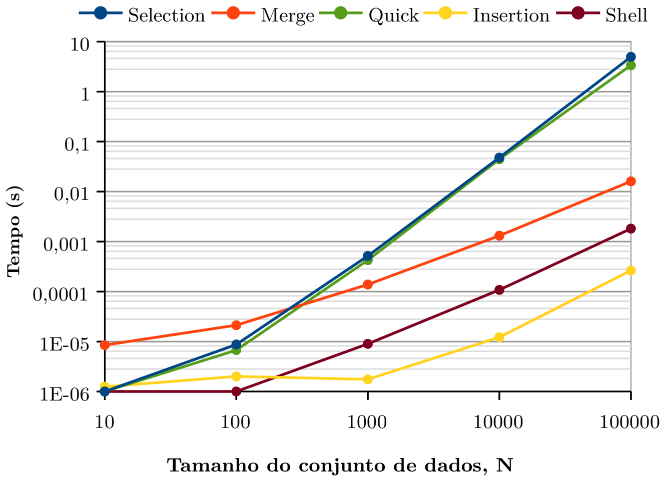

Figura 3.7 – Custo computacional dos algoritmos para conjuntos de números em ordem crescente.

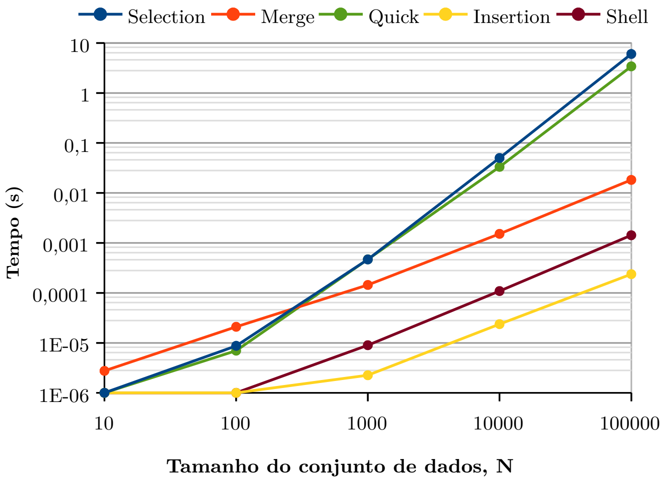

Figura 3.8 – Custo computacional dos algoritmos para conjuntos de números em ordem decrescente.

Como pode ser observado na Figura 3.6, o melhor algoritmo para dados aleatórios foi *quick*, seguido de perto pelo *shell* e *merge sort*. Já para os casos em que os números estavam ordenados de forma crescente ou decrescente (Figura 3.7 e 3.8), os algoritmos *selection* e *quick* tiverem desempenho muito pior que os demais, sendo o insertion o melhor deles, seguido pelo *shell* e *merge*, respectivamente.

4 Considerações finais

Com este trabalho foi possível observar as vantagens e desvantagens de alguns algorítimos populares de ordenação, assim como as características de suas implementações. Além disso, viu-se que os algoritmos apresentam melhor ou pior desempenho a depende da disposição dos elementos a serem ordenados. Uma vez que os algoritmos *merge* e *shell* apresentaram relativamente bons desempenhos independe da disposição dos dados, eles podem ser considerados estáveis e bons para qualquer uso.