Algoritmos de Ordenação: Um Estudo Comparativo

Jackson É. G. Souza[1], João V. G. Ricarte[1], Náthalee C. A. Lima[2]

[1]Universidade Federal Rural do Semi-Árido, Curso de Ciência e Tecnologia.

[2]Universidade Federal Rural do Semi-Árido, Centro Multidisciplinar de Pau dos Ferros.

jacksonegsouza@gmail.com, jv_ricarte@hotmail.com,
nathalee.almeida@ufersa.edu.br

Resumo. *O acúmulo de dados torna-se cada vez mais um problema a ser resolvido. Faz-se então, necessário o uso dos algoritmos de ordenação, que nada mais são do que processos lógicos para se organizar uma determinada estrutura linear, seja ela física ou não. Este estudo objetiva-se avaliar, por meio de experimentação, os dados gerados com a utilização dos algoritmos de ordenação Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Shell Sort, bem como, suas utilidades e eficiências. Para tanto, são executados três testes com três vetores de tamanhos diferentes para cada algoritmo e, os dados coletados foram comparados.*

Abstract. *Data stocking has become a problem to be solved. Thus, we ought to use sorting algorithms, which are none other than logic processes to organize a certain linear structure, being it physical or not. With this in mind, this study aims to evaluate, through experimenting, the data gathered by using the sorting algorithms Bubble Sort, Insertion Sort, Selection Sort, Merge Sort and Shell Sort, as well as their utility and efficiency. To do such, for each algorithm, we have executed three tests on three vectors of different sizes and the data collected compared.*

1. Introdução

Este estudo objetiva-se em avaliar, por meio de experimentação, os dados gerados com a utilização dos algoritmos de ordenação *Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Shell Sort*, bem como, suas respectivas utilidades e eficiências. Para tanto, são executados três testes com três vetores de tamanhos diferentes para cada algoritmo e, os dados coletados são comparados. Apesar de ser possível a implementação dos algoritmos de ordenação com qualquer estrutura linear, optamos por realizar os testes apenas com algarismos numéricos inteiros (sequências de números) a fim de obtermos dados quantitativos que facilitem o entendimento da análise posterior.

Trabalhos similares existem na literatura – (FOLATOR et al., 2014), (HONORATO, 2013), (SILVA, 2010), (TELLES et al.,2015), (YANG et al., 2011) –, mas, estes não relacionam os mesmos seis algoritmos referidos neste artigo.

Ao iniciar com um vetor com capacidade para 100 diferentes valores, são avaliados os tempos de execução, os números de comparações e, os números de movimentações para a organização de: a) uma lista ordenada em ordem crescente; b) uma lista ordenada em ordem decrescente, e; c) uma lista desordenada com números aleatórios. O mesmo

procedimento é realizado com vetores de tamanhos 1.000 e 10.000. Com os resultados espera-se apontar as vantagens e desvantagens ao se fazer uso de um algoritmo em detrimento de outro.

O presente estudo se organiza de forma que, na segunda seção serão apresentados, de forma breve, os algoritmos de ordenação que fazem parte deste estudo e seus respectivos códigos em Linguagem C. Na seção 3 (três) serão apresentados os dados coletados nos testes, bem como uma breve análise e, por fim, na seção 4 (quatro), serão apresentadas as conclusões acerca da análise dos dados.

2. Algoritmos de Ordenação

Algoritmos de ordenação são algoritmos que direcionam para a ordenação, ou reordenação, de valores apresentados em uma dada sequência, para que os dados possam ser acessados posteriormente de forma mais eficiente. Uma das principais finalidades desse tipo de algoritmo é a ordenação de vetores, uma vez que, em uma única variável pode-se ter inúmeras posições, a depender do tamanho do vetor declarado. Por exemplo, a organização de uma lista de presença escolar para que a relação fique organizada em ordem alfabética.

2.1. *Bubble Sort*

Este algoritmo é um do mais simples (SZWARCFITER e MARKENZON, 2015) e seu funcionamento ocorre por meio da comparação entre dois elementos e sua permuta, de modo que o elemento de maior valor fique à direita do outro (PEREIRA, 2010). "Após a primeira passagem completa pelo vetor (...) podemos garantir que o maior item terá sido deslocado para a última posição do vetor" [Pereira 2010, p.95]. Essa movimentação repete-se até que o vetor fique totalmente ordenado.

```
void bubble (int *item, int count)
{   register int a, b;
    register int t;
     for(a=(count-1); a>0; a--) {
       for(b=0; b<a; b++) {
          if(item[b]>item[b+1]) {
             t=item[b];
             item[b]=item[b+1];
             item[b+1]=t;  } } } }
```

Fonte: SCHILDT (1996) (Editado pelo Autor, 2017).

2.2. *Insertion Sort*

O *Insertion Sort* é de fácil implementação, similar ao *Bubble Sort* (SZWARCFITER e MARKENZON, 2015). Seu funcionamento se dá por comparação e inserção direta. A medida que o algoritmo varre a lista de elementos, o mesmo os organiza, um a um, em sua posição mais correta, onde o elemento a ser alocado (*k*) terá, a sua esquerda um valor menor (*k-1*), e, de maneira similar, à sua direita um valor maior (*k+1*).

```
void insert (int *item, int count)
{          register int a, b, t;
           for (a = 1; a < count; ++a)  {
                      t = item[a];
                      for (b = a - 1; b >= 0 && t < item[b]; b--)
                                 item[b+1] = item[b]; item[b+1] = t;} }
```

Fonte: SCHILDT (1996) (Editado pelo Autor, 2017).

2.3. *Selection Sort*

Um dos mais simples e utilizados, o *Selection Sort* tem como principal finalidade passar o menor valor para a primeira posição, o segundo menor para a segunda posição, e assim sucessivamente, para os *n* valores nas *n* posições, onde o valor à esquerda é sempre menor que o valor à direita ($valor_{esquerda} < valor_{direita}$).

```
void select (int  *item, int count)
{  register int a, b, c;
   int exchange,  t;
   for (a = 0; a < count-1; ++a) {
            exchange = 0;
            c = a;
            t = item[a];
            for (b = a+1; b < count; ++b) {
               if (item[b] < t) {
                        c = b;
                        t = item[b];
                        Exchange = 1; } }
            if (exchange) {
              item[c] = item[a];
              item[a] = t; } } }
```

Fonte: SCHILDT (1996) (Editado pelo Autor, 2017).

2.4. *Merge Sort*

Este algoritmo tem como objetivo a reordenação de uma estrutura linear por meio da quebra, intercalação e união dos *n* elementos existentes. Em outras palavras, a estrutura a ser reordenada será, de forma recursiva, subdividida em estruturas menores até que não seja mais possível fazê-lo. Em seguida, os elementos serão organizados de modo que cada subestrutura ficará ordenada. Feito isso, as subestruturas menores (agora ordenadas) serão unidas, sendo seus elementos ordenados por meio de intercalação (SZWARCFITER e MARKENZON, 2015). O mesmo processo repete-se até que todos os elementos estejam unidos em uma única estrutura organizada.

```
void merge(int v[],int n){
            int meio, i, j, k;
            int* auxiliar;
auxiliar =(int*) malloc(n * sizeof(int));
    if (auxiliar==NULL){
    return;}
meio = n/2;
i = 0; k=0;
j = meio;
            while (i < meio && j < n) {
             if (v[i] <= v[j]) {
            auxiliar[k] = v[i++];
             }else {auxiliar[k] = v[j++];
             } ++k;   }
            if (i == meio) {
            while (j < n) {
            auxiliar[k++] = v[j++]; }
            }else {
                        while (i < meio) {
                         auxiliar[k++] = v[i++]; }     }
```

```
for (i = 0; i < n; ++i) {
        v[i] = auxiliar[i]; }
free(auxiliar); }
void mergeSort(int v[], int n) {
        int meio;
        if (n > 1) {
        meio = n / 2;
  mergeSort(v, meio);
        mergeSort(v + meio, n - meio);
        merge(v, n); } }
```

Fonte: GIACON et al. (s.d.) (Editado pelo Autor, 2017).

2.5. *Quick Sort*

O *Quick Sort* usa do mesmo princípio de divisão que o *Merge Sort*, entretanto, o mesmo não utiliza a intercalação, uma vez que não subdivide a dada estrutura em muitas menores. Esse algoritmo simplesmente faz uso de um dos elementos da estrutura linear (determinada pelo programador) como parâmetro inicial, denominado *pivô*. Com o pivô definido, o algoritmo irá dividir a estrutura inicial em duas, a primeira, à esquerda, contendo todos os elementos de valores menores que o pivô, e, à direita, todos os elementos com valores maiores. Em seguida, o mesmo procedimento é realizado com o a primeira lista ($valores_{menores}$<pivô<$valores_{maiores}$). O mesmo processo se repete até que todos os elementos estejam ordenados (SZWARCFITER e MARKENZON, 2015).

```
void quick (int *item, int count)
{    Qs (item, 0, count -1)   }
void Qs (int  *item, int left, int right)
{    register int i, j;
     int x, y;
     i = left; j = right;
     x = item[ ( left + right ) / 2 ];
     do {
        while ( item[ i ] < x && i < right ) i++;
        while ( x < item[ j ] && j > left ) j--;
        if ( i <= j ) {
           y = item[ i ];
           item[ i ] = item[ j ];
           item[ j ] = y;
           i++ ;  j-- ;   }
     }  while ( i <= j );
     if ( left < j )  Qs ( item, left, j );
     if ( i < right )  Qs ( item, i, right ); }
```

Fonte: SCHILDT (1996) (Editado pelo Autor, 2017).

2.6. *Shell Sort*

Baseado, de forma melhorada, no *Insertion Sort*, o *Shell Sort* é um algoritmo que, fazendo uso de um valor de distanciamento, denominado *gap*, rearranja os elementos de uma estrutura por meio de inserção direta (SILVA, 2010). O referido valor de distanciamento, *gap*, nada mais é do que um valor que definirá a distância com que os elementos serão comparados, ou seja, se no *Insertion Sort* cada elemento é comparado com o elemento imediatamente subsequente, e, o de menor valor inserido na posição à esquerda do outro, no *Shell Sort*, o *gap*, determinado pelo programador, fará com que os elementos a serem

comparados, não mais, sejam tão próximos. Essa pequena diferença pode parecer pouca, mas, quando aplicada, faz com o que o processo seja mais rápido, visto que, devido a distância de comparação, seria como se estivéssemos rearranjando não uma, mas duas estruturas por vez.

```
void shellsort(int *vet, int count)
{   int i , j , value;
    int gap = 1;
    while(gap < count) {
        gap = 3*gap+1;  }
    while ( gap > 1) {
        gap /= 3;
        for(i = gap; i < count; i++) {
            value = vet[i];
            j = i - gap;
            while (j >= 0 && value < vet[j]) {
                vet [j + gap] = vet[j];
                j -= gap; }
            vet [j + gap] = value; }  } }
```

Fonte: O Autor (2017).

3. Resultados e Discussões

A Linguagem C foi definida como meio de codificação para a realização das simulações e a IDE *Code Blocks* como ambiente de desenvolvimento. Quanto ao *hardware*, a máquina em que os testes foram executados tem as seguintes configurações:

- Fabricante: 3GREEN Technology;
- Modelo: J1800;
- Processador: Intel(R) Celerom(R) Dual Core;
- Frequência: 2,41 GHz;
- Memória RAM: 8,00 GB (DDR3 1.333MHz);
- Sistema Operacional: Windows 10 PRO (64 bits).

3.1. Teste em um Vetor de 100 Elementos

A Tabela 1 mostra os valores médios do desempenho dos algoritmos ao lidar com uma lista pré-ordenada de forma crescente, decrescente e aleatória.

Tabela 1. Valores médios de desempenho para um vetor [100]

VETOR [100]									
Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas
Bubble Sort	0,0002364	4950	0	0,0002693	4950	4950	0,000236	4950	2377
Insertion Sort	0,0001528	99	0	0,0001953	99	4950	0,000175	99	24941
Selection Sort	0,0001954	4950	0	0,0001929	4950	50	0,0001951	4950	93,5
Merge Sort	0,0000691	699	1200	0,0000672	419	1236	0,000072	659,6	1218,8
Quick Sort	0,000045803	4950	0	0,000118511	4950	4950	0,000031697	641,1	2566,1
Shell Sort	0,0001441	342	0	0,0001479	342	230	0,0001575	342	419,7

Podemos observar que, em uma lista já ordenada, o *Insertion Sort* foi o que apresentou o melhor desempenho por executar o menor número de comparações e nenhuma troca (*swap*). De forma oposta, o *Merge Sort*, além de ter sido o único que apresentou trocas, obteve um alto valor. Esse resultado se deve ao fato do *Insertion Sort* avaliar o vetor, comparando seus elementos, de forma global, o que permite identificar, em primeira instância, que o mesmo já está organizado como deveria, e, portanto, não há necessidade de permuta. O mesmo não pode ser dito a respeito do *Merge Sort* que, recursivamente, subdivide o vetor em estruturas menores para, somente depois, efetuar comparações, e, por fim, reestruturá-lo novamente. Ou seja, independente do vetor já estar ordenado, o algoritmo efetuará permutas até que todas as subdivisões voltem a ser uma única estrutura.

Em uma lista com a ordenação decrescente, é possível identificar que, devido a sua abordagem de comparação e inserção com distanciamento, o *Shell Sort* apresenta o melhor desempenho com o menor número de comparações e trocas. Os *Bubble Sort* e o *Quick Sort*, de modo contrário, ao percorrer todo o vetor, comparando e permutando (quando preciso), demonstram sua ineficiência – para o teste em questão – ao atingir altos valores, mesmo que o *Quick Sort* apresente o segundo menor tempo de execução.

Na ordenação de uma lista com elementos aleatórios, observa-se que, de modo geral, o *Shell Sort* tem a melhor relação comparações-trocas, sendo, assim, considerado o de melhor desempenho. Por outro lado, o *Bubble Sort* apresenta o pior desempenho, resultando em altos valores de comparações e trocas.

3.2. Vetores com 1.000 e 10.000 Elementos.

Abaixo, os dados do vetor [1000] são exibidos nas Tabela 2.

Tabela 2. Valores médios de desempenho para um vetor [1000]

VETOR [1.000]									
Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas
Bubble Sort	0,0043496	499500	0	0,0099207	499500	499500	0,0078604	499500	249816
Insertion Sort	0,1647	999	0	0,0046157	999	499500	0,0024474	999	252804,5
Selection Sort	0,0043271	499500	0	0,0004063	499500	500	0,0044346	499500	993,1
Merge Sort	0,000654	9999	1800	0,0006236	5555	18488	0,000734	9643,5	18247
Quick Sort	0,004434646	499500	0	0,01230268	499500	499500	0,1948871	10592,7	250206,8
Shell Sort	0,0002319	5457	0	0,0002702	5457	3920	0,000385	5457	8301,7

Similar ao vetor com 100 elementos, podemos ver que, em uma lista ordenada de forma crescente, no que se refere as comparações e trocas, o *Insertion Sort* continua sendo o algoritmo com melhor desempenho e, o *Merge Sort* com o pior. Do mesmo modo, em uma lista decrescente, podemos ver que, o *Shell Sort* continua apresentando o melhor desempenho, e, o *Bubble Sort* e o *Quick Sort* o menos satisfatório. Quanto a lista aleatória, o *Shell Sort* mantém a melhor performance, assim como, o *Bubble Sort* permanece com altos valores de comparações e trocas.

Na Tabela 3, assim como nas tabelas anteriores, temos os resultados dos testes realizados com o vetor de dez mil (10.000) posições.

Tabela 3. Valores médios de desempenho para um vetor [10000]

VETOR [10.000]									
Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas
Bubble Sort	0,4269048	49995000	0	0,9847921	49995000	49995000	0,7649256	49995000	25084128,1
Insertion Sort	0,0003026	9999	0	0,4580984	9999	49995000	0,225615	9999	24963151
Selection Sort	0,3637704	49995000	0	0,3827789	49995000	5000	0,360824	49995000	9988
Merge Sort	0,0058387	135423	250848	0,0056613	74911	254944	0,006185	132011,1	252879
Quick Sort	0,4415975	49995000	0	1,192945	49995000	49995000	0,1867259	158055	25098217,7
Shell Sort	0,001431	75243	0	0,0019362	75243	161374	0,0034228	75243	161374

Mais uma vez identificamos que o padrão se repete, em um vetor pré-ordenado, o *Insertion Sort* é o algoritmo que tem a melhor performance. Igualmente, o *Merge Sort* continua a apresentar resultados insatisfatórios. Os resultados obtidos com as listas decrescente e aleatória correspondem aos anteriores, executados com vetores menores.

4. Conclusão

O algoritmo *Bubble Sort*, apesar de ser o de mais fácil implementação, não apresenta resultados satisfatórios, principalmente no número de comparações. A ineficiência desse algoritmo pode ser traduzida como um grande consumo de processamento, o que, para máquinas com poucos (ou limitados) recursos computacionais, resulta em lentidão e longos períodos de espera. Sua aplicação é, na opinião dos autores, indicada somente para fins educacionais, visto que um projeto com o mesmo pode ser considerado ineficiente e/ou fraco (SILVA, 2010).

O *Insertion Sort*, por sua vez, é útil para estruturas lineares pequenas, geralmente entre 8 e 20 elementos, sendo amplamente utilizados em sequências de 10 elementos, tendo ainda, listas ordenadas de forma decrescente como pior caso, listas em ordem crescente como o melhor caso e, as demais ordens como sendo casos medianos. Sua principal vantagem é o pequeno número de comparações, e, o excessivo número de trocas, sua desvantagem. Como exemplo de uso, tem-se a ordenação de cartas de um baralho.

O *Selection Sort* torna-se útil em estruturas lineares similares ao do *Insertion Sort*, porém, com o número de elementos consideravelmente maior, já que, o número de trocas é muito inferior ao número de comparações, consumindo, assim, mais tempo de leitura e menos de escrita. A vantagem de seu uso ocorre quando se trabalha com componentes em que, quanto mais se escreve, ou reescreve, mais se desgasta, e, consequentemente, perdem sua eficiência, como é o caso das memórias EEPROM e FLASH.

Ambos os algoritmos (*Insertion* e *Selection*), apesar de suas diferentes características, são mais comumente utilizados em associação com outros algoritmos de ordenação, como os *Merge Sort*, *Quick Sort* e o *Shell Sort*, que tendem a subdividir as listas a serem organizadas em listas menores, fazendo com que sejam mais eficientemente utilizados.

O *Merge Sort* apresenta-se, em linhas gerais, como um algoritmo de ordenação mediano. Devido a recursividade ser sua principal ferramenta, seu melhor resultado vem ao lidar com estruturas lineares aleatórias. Entretanto, ao lidar com estrutura pequenas e/ou já

pré-ordenada (crescente ou decrescente), a recursividade passa a ser uma desvantagem, consumindo tempo de processamento e realizando trocas desnecessárias. Esse algoritmo é indicado para quando se lida com estruturas lineares em que a divisão em estruturas menores sejam o objetivo, como, por exemplo, em filas para operações bancárias.

O algoritmo *Quick Sort*, ao subdividir o vetor e fazer inserções diretas utilizando um valor de referência (pivô), reduz seu tempo de execução, mas, as quantidades de comparações (leitura) e, principalmente, trocas (escrita) ainda são muito altas. Apesar disso, o *Quick Sort* se apresenta uma boa opção para situações em que o objetivo é a execução em um menor tempo, mesmo que para isso haja um detrimento em recursos computacionais de processamento.

O *Shell Sort*, baseado nos dados deste trabalho, é o que apresenta os resultados mais satisfatórios, principalmente com estruturas maiores e desorganizadas. Por ser considerado uma melhoria do *Selection Sort*, o *Shell Sort*, ao ser utilizado com as mesmas finalidades que seu predecessor – recursos que demandem pouca escrita – irá apresentar um melhor desempenho, e, consequentemente, expandir a vida útil dos recursos.

5. Referências

Folador, J. P., Neto, L. N. P., Jorge, D. C. (2014). “Aplicativo para Análise Comparativa do Comportamento de Algoritmos de Ordenação.” Revista Brasileira de Computação Aplicada (ISSN 2176-6649), Passo Fundo, v. 6, n. 2, p. 76-86, out. 2014.

Giacon, A. P. Folis, D. C., Hernandes, D. N. Spadotim, F. C. "Mergesort". http://www.ft.unicamp.br/liag/siteEd/includes/arquivos/MergeSortResumo_Grupo4_ST364A_2010.pdf, setembro.

Honorato, B. (2013). "Algoritmos de ordenação: análise e comparação." http://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261, abril.

Pereira, S. L. (2010). “Algoritmos e Lógica de Programação em C: uma abordagem didática.” 1ª edição. São Paulo. Érica.

Schildt, H. “C, Completo e Total.” (1996). 3ª edição. São Paulo. Makron Books.

Silva, E. S. (2010). “Estudo Comparativo de Algoritmos de Ordenação.” 33 f. Trabalho de Conclusão de Curso – Universidade Federal do Espírito Santo, São Mateus.

Szwarcfiter, J. L. and Markezon, L. (2015). “Estruturas de Dados e Seus Algoritmos.” 3ª edição. Rio de Janeiro. LTC.

Telles, M. J., Santini, P. H., Rigo, S. J., Santos, J. V. C., Barbosa, J. L.V. (2015). “iSorting: um Estudo Sobre Otimização em Algoritmos de Ordenação”. Revista Brasileira de Computação Aplicada (ISSN 2176-6649), Passo Fundo, v. 7, n. 1, p. 01-15, abr. 2015.

Yang, Y., Yu, P., Gan, Y. (2011). “Experimental Study on the Five Sort Algorithms.” Second International Conference on Mechanic Automation and Control Engineering, 2011. IEEE Xplore, 2011. p. 1314-1317. ISBN 978-1-4244-9439-2.