



UNIVERSIDADE FEDERAL DE MATO GROSSO
ALGORITMOS E ESTRUTURA DE DADOS II

EDUARDO OLIVEIRA SILVA
MATHEO RODRIGUES BONUCIA

ALGORITMOS DE ORDENAÇÃO

CUIABÁ - MT
2021

EDUARDO OLIVEIRA SILVA
MATHEO RODRIGUES BONUCIA

ALGORITMOS DE ORDENAÇÃO

Trabalho apresentado a disciplina de Algoritmos e Estrutura de Dados II da Universidade Federal de Mato Grosso, para o aprendizado dos discentes e obtenção da nota de conclusão.

Professor (a): Frederico Santos de Oliveira

CUIABÁ - MT

2021

1. INTRODUÇÃO

O projeto avaliativo dois, consiste na criação de um algoritmo que possa desfrutar da ordenação de n elementos de um vetor de oito formas diferentes, sendo cada uma delas recomendadas para certo número de inteiros dentro de um vetor. Ademais, as oito formas diferentes são: *bubblesort*, *insertionsort*, *selectionsort*, *shellsort*, *mergesort*, *heapsort*, *quicksort* e *countingsort*. Assim sendo, respectivamente, as três primeiras formas são indicadas para uma ordenação com baixo número de elementos devido a sua estrutura ser mais simples, entretanto, as restantes, exceto *countingsort*, são responsáveis por terem uma estrutura mais elaborada, possibilitando assim a ordenação de n elementos em quantidades maiores que em relação as três primeiras formas, destacando também, a forma *quicksort* que quando bem estruturada possui uma velocidade de processamento superior as demais. Também, o projeto conta com um algoritmo extra de ordenação, sendo justamente o *countingsort*, que foi escolhido devido a sua possibilidade de processamento extremamente rápido com uma baixa quantidade de números inteiros.

De modo geral, o algoritmo foi estruturado em oito bibliotecas, onde cada uma abriga a função referente a forma de ordenação respectiva e possui o arquivo *main* que é constituído de quatro bibliotecas, “stdio.h”, “stdlib.h”, “time.h” e “math.h”, bem como as oito bibliotecas de ordenação e outra responsável por realizar a contagem de movimentações e comparações nas formas *bubblesort*, *insertionsort* e *selectionsort*, denominada “analysedetempo.h”. Por fim, cada uma das funções e bibliotecas que formam o algoritmo completa serão descritas e explicadas ao desenvolver do projeto.

2. DESENVOLVIMENTO

O projeto parte do princípio em utilizar a criação de “arquivos .h”, ou seja, bibliotecas que nelas possam ser armazenadas as funções que correspondem ao algoritmo da forma de ordenação correspondente, a exemplo o *bubblesort*, como demonstrado logo abaixo (figura 1) em que é possível visualizar o algoritmo da ordenação (*bubblesort*) abrigado em uma biblioteca que será posteriormente chamada na função principal *main*.

```
1  #ifndef BUBBLESORT_H_INCLUDED
2  #define BUBBLESORT_H_INCLUDED
3
4
5
6  void bbsort(int n, int *vPtr){
7      int i, j, a;
8
9      for(i=1; i<n; i++){
10         for(j=0; j<n-i; j++){
11             if(vPtr[j]>vPtr[j+1]){
12                 a = vPtr[j+1];
13                 vPtr[j+1]=vPtr[j];
14                 vPtr[j] = a;
15
16
17
18
19             }
20         }
21     }
22 }
23
24 #endif // BUBBLESORT_H_INCLUDED
25
```

Figura 1

Como o principal propósito do projeto seja apresentar as diferentes formas de ordenação, este método demonstrado acima, possibilita ao programador uma organização melhor do código, assim como a utilização dessas funções em outros códigos de maneira mais rápida e prática. Portanto, todas as oito formas de ordenação e a função responsável pela análise da movimentação e comparação das três primeiras formas mais simples (como citado anteriormente), são inseridas ao projeto da mesma forma, como abaixo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5
6
7  #include "bubblesort.h"
8  #include "selectionsort.h"
9  #include "insertionsort.h"
10 #include "shellsort.h"
11 #include "mergesort.h"
12 #include "heapsort.h"
13 #include "quicksort.h"
14 #include "countingsort.h"
15 #include "analisedetempo.h"
```

Figura 2

Ademais, são criadas as quatro funções responsáveis por gerarem e exibirem o vetor de n elementos, sendo elas: “funcaorandom”, “funcaocrescente”, “funcaodecrescente” e “printvetor”. Todas as funções são do tipo *void* e são, respectivamente, responsáveis pela criação de n elementos de forma aleatória, de forma crescente e decrescente, por exemplo, caso o usuário informe que queira a criação de 5000 elementos na forma crescente, a função irá gerar elementos de 0 a 4999 (0, 1, 2, ..., 4999), bem como a última função irá ser responsável por exibir estes inteiros gerados na tela do usuário.

Partindo ao esqueleto do código, o *main* será responsável por pedir para o usuário informar qual o tipo de ordem que ele deseja para a execução da ordenação (aleatório, crescente ou decrescente), após isso será solicitado qual o tamanho desejado do vetor (quantidade n de elementos) e será reservado um espaço para o ponteiro “vPtr” com o *malloc*. A partir deste ponto, o código será dividido em três partes extensas, sendo elas as ordens 1, 2 e 3, onde a ordem 1 irá significar quando o vetor é gerado por inteiros aleatórios de 0 a 9, a ordem

2 será crescente (0, 1, 2, ..., n) e a ordem 3 decrescente (n-1, n-2, ..., 0), assim sendo, cada parte extensa desta irá abrigar em sua composição uma função *if* que será responsável por indicar qual formato de ordenação deverá ser chamado (aquele escolhido pelo usuário), *bubblesort*, *selectionsort*, *etc.* Quando chamado o formato escolhido, o algoritmo irá calcular o tempo de processamento deste formato a partir da função *clock()*, que se encontra na biblioteca “temp.h” e será printado o resultado após gerar na tela do indivíduo a ordenação dos n elementos, consequentemente, este mesmo processo irá se repetir para os demais formatos de ordenação.

2.1 FUNÇÕES DO VETOR

A primeira função do código de ordenação, é representado pela “funcaorandom” que tem como finalidade gerar números inteiros de 0 a 9, sendo a quantidade n definida pelo usuário. Esta possui em sua composição a função *rand* que pertence a biblioteca “stdlib.h” e é demonstrado abaixo na figura 3.

```
void funcaorandom(int *vPtr, int n){
    int randomIndex;
    int fullrand = 0;
    int i=0;

    while(i<n){
        fullrand = rand();
        randomIndex = fullrand % 10;
        vPtr[i] = randomIndex;
        i++;
    }
}
```

Figura 3

A segunda função, denomina-se “funcaocrescente” e baseia-se praticamente em gerar um vetor de n elementos, onde esses elementos serão gerados já em ordem crescente.

```
void funcaocrescente(int *vPtr, int n){
    int i = 0;

    for(i=0;i<n;i++){
        vPtr[i] = i;
    }
}
```

Figura 4

A terceira função “funcaodecrescente” é o contrário da segunda, onde esta se responsabiliza por gerar n elementos de forma decrescente.

```
void funcaodecrescente(int *vPtr, int n){  
    int i=0;  
    int b = n;  
    for(i=0;i<n;i++){  
        vPtr[i] = b-1;  
        b--;  
    }  
}
```

Figura 5

Por fim, a última função das quatro responsáveis por gerarem o vetor de n elementos, é a “printvetor” onde está irá mostrar na tela do usuário os n elementos gerados a partir da escolha dele.

```
void printvetor(int *vPtr,int n){  
    int i=0;  
    for(i=0; i<n; i++){  
        printf("%d ",vPtr[i]);  
    }  
    printf("\n");  
}
```


Figura 6

2.2 ALGORITMOS DE ORDENAÇÃO

O algoritmo de ordenação será dividido em oito formas diferentes de se ordenar, sendo sete destas apresentadas durante o curso de algoritmos de estrutura de dados 2 e uma restante que foi obtida a partir de estudar realizados na internet. Todas as formas de ordenação têm como propósito ordenar os n números inteiros da melhor forma possível, sendo que cada uma destas formas irá possuir três casos diferentes de resultados, sendo os casos médios, o pior caso e o melhor caso. Ademais, cada algoritmo de ordenação irá apresentar o tempo de processamento levado para ordenar os elementos, bem como o número de movimentações e comparações realizados durante o processo.

2.2.1 BUBBLESORT

Consiste em um algoritmo estável e simples, onde o vetor é analisado várias vezes e é realizado a “flutuação” do elemento maior para o final da sequência. O termo *bubble* está implementado por conta deste processo onde os inteiros vão “borbulhando” para suas posições correspondentes na ordenação. De fato, há uma comparação de dois números inteiros onde o maior é posto a direita e o processo se repete até todos estarem ordenados.



```
void bbsort(int n, int *vPtr){  
    int i, j, a;  
  
    for(i=1; i<n; i++){  
        for(j=0; j<n-i; j++){  
            if(vPtr[j]>vPtr[j+1]){  
                a = vPtr[j+1];  
                vPtr[j+1]=vPtr[j];  
                vPtr[j] = a;  
            }  
        }  
    }  
}
```

Figura 7 (Bubblesort)

2.2.2 SELECTIONSORT

O *selectionsort* consiste em realizar a transferência do menor valor do vetor para o início na primeira iteração, posteriormente realizará este mesmo procedimento para os demais elementos dentro do vetor até que a ordenação esteja concluída. Ademais, o algoritmo não é estável e seu custo para caso os n elementos já esteja ordenado ainda é alto.

```
void slsort(int n, int *vPtr) {  
    int i, j, a, m;  
    for(i=0; i<n-1; i++) {  
        m = i;  
        for(j=i+1; j<n; j++) {  
            if(vPtr[j] < vPtr[m]) {  
                m = j;  
            }  
        }  
        a = vPtr[m];  
        vPtr[m] = vPtr[i];  
        vPtr[i] = a;  
    }  
}
```

Figura 8 (Selectionsort)

2.2.3 INSERTIONSORT

Este algoritmo funciona a base de “inserção” dos valores em seus respectivos lugares após a comparação, levando em consideração que ao ser inserido o valor a sua esquerda deve ser menor que ele e o valor a sua direita maior, gerando assim um vetor ordenado. Um adendo é que este formato é estável e possui um uso muito recorrente quando um vetor possui elementos quase já ordenados.

```
void istsort(int n, int *vPtr){  
    int i, j, a;  
    for(i=1;i<n;i++){  
        a = vPtr[i];  
        j=i-1;  
        while(j>=0 && vPtr[j]>a){  
            vPtr[j+1] = vPtr[j];  
            j=j-1;  
        }  
        vPtr[j+1]=a;  
    }  
}
```

Figura 9 (Insertionsort)

2.2.4 SHELLSORT

O algoritmo de ordenação *shellsort* é basicamente uma extensão do *insertionsort* em que é usado o recurso de distanciamento denominado *gap* ou é representado pela letra “h”, que será responsável por determinar uma distância entre os elementos no vetor que serão comparados. Diferentemente do *insertionsort* onde cada arranjo era realizado com o elemento subsequente, neste formato haverá este distanciamento que proporciona uma melhora na velocidade de processamento do código. Entretanto, o tamanho do distanciamento ou o “h”, possui algumas características que influenciam, já que sempre que “h” tem valor igual 1, irá garantir a ordenação correta, mas caso seja escolhido um distanciamento errado, irá afetar diretamente no processamento da ordenação, uma vez que o valor de “h” quando diferente de 1 irá sendo diminuído até chegar a 1 e realizar a ultima iteração.

```
void sllsort(int n, int *vPtr){
    int h, i, j, a;
    h=1;
    while(h<n){
        h=3*h+1;
    }
    while(h>=1){
        h=h/3;
        for(i=h; i<n; i++){
            a = vPtr[i];
            j=i-h;
            while(j>=0 && vPtr[j]>a){
                vPtr[j+h] = vPtr[j];
                j=j-h;
            }
            vPtr[j+h] = a;
        }
    }
}
```

Figura 10 (Shellsort)

2.2.5 MERGESORT

O algoritmo *mergesort* fará uso de um método chamado Divisão e Conquista, onde este consiste em subdividir o vetor em estruturas menores, perpetuando até que chegue um ponto que as estruturas poderão serem resolvidas de forma direta, possibilitando assim a ordenação dessas subestruturas. Sendo assim, após este processo, haverá a combinação de todas as partes em um vetor que abrigará o resultado da ordenação, por fim, este vetor irá copiar no vetor original a ordenação que foi realocada nele. Por característica, este formato possui um recurso que possibilita o uso de recorrências.

```
void mgsort(int *vPtr, int in, int fi){
    int m;
    if(in<fi){

        m = floor((in+fi)/2);
        mgsort(vPtr,in,m);
        mgsort(vPtr,m+1,fi);
        merg(vPtr,in,m,fi);

    }

}
```

Figura 11 (Parte 1 Mergesort)

```
void merg(int *vPtr, int in, int m, int fi){
    int *v, x, y, tam, i, j, k;
    int fil = 0, fi2 = 0;

    tam = fi-in+1;

    x = in;
    y = m+1;

    v = (int *) malloc(tam*sizeof(int));

    if(v != NULL){
        for(i=0; i<tam; i++){

            if(!fil && !fi2){

                if(vPtr[x] < vPtr[y]){
                    v[i]= vPtr[x++];
                }else{
                    v[i]= vPtr[y++];
                }

                if(x>m)
                    fil=1;

                if(y>fi)
                    fi2=1;

            }

            else{

                if(!fil){
                    v[i]=vPtr[x++];
                } else{
                    v[i]=vPtr[y++];
                }

            }

        }

        for(j=0,k=in;j<tam;j++,k++){
            vPtr[k] = v[j];
        }

        free(v);
    }

}
```

Figura 12 (Parte 2 Mergesort)

2.2.6 HEAPSORT

Este algoritmo utiliza do princípio da seleção que é o que constitui o *selectionsort*, sendo assim, o que difere ambos é que no *heapsort* será utilizado uma fila de prioridades, onde será construído esta fila com n elementos, irá retirar o elemento com a maior prioridade e assim altera a prioridade do item após a iteração. Assim como em uma árvore binária, o item superior irá ser representado por "i" e os demais (raízes) serão representados por $2i+1$ e $2i+2$ a fim de gerar a ordenação dos n números no vetor.

```
void createhp(int *vPtr, int in, int fi) {
    int a = vPtr[in];
    int j = in*2+1;
    while(j <= fi) {
        if(j < fi) {
            if(vPtr[j] < vPtr[j+1]) {
                j = j+1;
            }
        }
        if(a < vPtr[j]) {
            vPtr[in] = vPtr[j];
            in = j;
            j = 2*in+1;
        } else {
            j = fi+1;
        }
    }
    vPtr[in] = a;
}
```

Figura 13 (Parte 1 Heapsort)

```
void hpsort(int *vPtr, int n) {
    int i, a;
    for(i = (n-1)/2; i >= 0; i--) {
        createhp(vPtr, i, n-1);
    }
    for(i = n-1; i >= 1; i--) {
        a = vPtr[0];
        vPtr[0] = vPtr[i];
        vPtr[i] = a;
        createhp(vPtr, 0, i-1);
    }
}
```

Figura 14 (Parte 2 Heapsort)

2.2.7 QUICKSORT

Dentre os métodos de ordenação apresentados em aula durante o curso de Algoritmos e Estrutura de Dados 2, o *quicksort* possui os melhores resultados de processamento para um abrangente tipo de situações. Analogamente ao *mergesort*, este método subdivide o problema em dois subproblemas, tendo como parâmetro diferencial um pivô, sendo que os inteiros a esquerda do pivô irão ser de valores menores e a direita de valores maiores até que ambas iterações do lado esquerdo e direito “se cruzam” e forçam a divisão do vetor que irá ser indicado outro pivô e repetindo o mesmo processo até a ordenação dos mesmos.

```
void qcksort(int *vPtr, int in, int fi){
    int i,j,x,y;
    i = in;
    j = fi;
    x = vPtr[(in+fi)/2];
    while(i <= j){
        while(vPtr[i] < x && i < fi)
            i++;
        while(vPtr[j] > x && j > in)
            j--;
        if(i <= j){
            y=vPtr[i];
            vPtr[i]=vPtr[j];
            vPtr[j]=y;
            i++; j--;
        }
    }
    if(j > in)
        qcksort(vPtr, in, j);
    if(i < fi)
        qcksort(vPtr, i ,fi);
}
```

Figura 15 (Quicksort)

2.2.8 COUNTINGSORT

Por fim, durante o projeto foi solicitado a apresentação de um tipo de ordenação diferente dos demais apresentados e foi escolhido o método *countingsort* que conta com um algoritmo simples e eficiente. Esta forma de ordenação é estável, utiliza-se de um *array* auxiliar que possui o tamanho do maior valor que estará sendo ordenado, bem como este *array* serve para contar quantas vezes cada valor ocorre.

Sua complexidade é baseada no tamanho do *array*, ou também “K”, onde é $O(K+n)$, entretanto, não é recomendável a utilização para valores muito grandes.

```
void countsort(int *vPtr, int n, int ordem){

    int i, j, k;
    int maxi;

    if(ordem == 2 || ordem == 3){
        maxi = n;
    }else if(ordem == 1){
        maxi = 10;
    }
    int contagem[maxi];

    for(i=0; i<maxi; i++){
        contagem[i]=0;
    }for(i=0; i<n; i++){
        contagem[vPtr[i]]++;
    }for(i=0, j=0; j<maxi; j++){
        for(k=contagem[j]; k>0; k--){
            vPtr[i++] = j;
        }
    }
}
```

Figura 16 (Countingsort)

Como pode se observar acima, o código é simplista e possui o limite de 10 quando a ordem vale 1 (aleatórios), já que os valores nesta ordem são gerados de 0 a 9, no entanto, na ordem 2 (crescente) e ordem 3 (decrescente) seu máximo equivale ao n número de elementos.

3. COMPILAÇÃO E EXECUÇÃO

Partindo de um princípio básico e com a utilização de imagens, o código é compilado via GNU GCC Compiler e é executado normalmente via CodeBlocks. Ao executar o código por meio do arquivo “main.c”, este irá solicitar ao usuário que escolha qual a forma de criação do vetor, podendo escolher as seguintes opções: 1 – Aleatório, 2 – Ordem crescente e 3 – Ordem decrescente.

```
Escolha uma forma de criacao:  
1-Aleatorio  
2-Ordem crescente  
3-Ordem decrescente
```

Após a escolha do usuário, irá ser solicitado a quantidade de elementos que deseja serem gerados, da ordem escolhida na primeira pergunta.

```
Digite o tamanho de elementos:
```

Ao inserir a quantidade de elementos e confirmar, irá ser printado na tela todos os n números que foram gerados.

```
Seu vetor gerado e:  
4 9 8 9 5 6 1 0 9 4 2 2 8 2 9 5 1 9 8 9 9 9 3 7  
9 5 1 5 0 1 9 3 6 2 6 8 5 0 4 2 7 0 1 0 4 6 9 6  
9 4 4
```

Demonstrado os números gerados, agora o programa irá solicitar a escolha de um dos oito métodos apresentados no decorrer do projeto.

```
Escolha um metodo de ordenacao, digite o valor exibido:  
1-Bubblesort  
2-Selectionsort  
3-Insertionsort  
4-Shellsort  
5-Mergesort  
6-Heapsort  
7-Quicksort  
8-Countingsort
```

Portanto, após escolher o método de ordenação, será apresentado na tela os n elementos já ordenados e abaixo o número de comparação e movimentação (somente para *bubblesort*, *selectionsort* e *insertionsort*), como também o tempo de processamento do algoritmo (para todos os métodos).


```
Comparacao: 12497500 - Movimentacao: 14997  
O tempo do algoritmo foi de: 38.000000
```

4. TESTES REALIZADOS

Ao finalizar o código, foram realizados testes com todas as oito formas de ordenação. Assim sendo, os modos *bubblesort*, *selectionsort* e *insertionsort* passaram por testes com 100, 1.000 e 10.000 elementos, uma vez que com valores maiores, a exemplo 700.000, o algoritmo não possui capacidade para processamento, já que estes três modos de ordenação foram criados para ordenar uma baixa quantidade de n números.

Entretanto, os modos *shellsort*, *mergesort*, *heapsort*, *quicksort* e *countingsort* já possuem uma estrutura capaz de suportar testes com valores maiores, como de 100.000 até 2.000.000 elementos.

4.1 VETOR [100]

TABELA DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT

Vetor [100]									
	Movimentação			Comparação			Clock (ms)		
	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3
Bubblesort	7425	0	14850	4950	4950	4950	0.035	0.015	0.019
Selectionsort	297	297	297	4950	4950	4950	0.014	0.014	0.012
Insertionsort	5049	198	5148	2525	99	4950	0.008	0.003	0.016

GRÁFICO DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT [100]

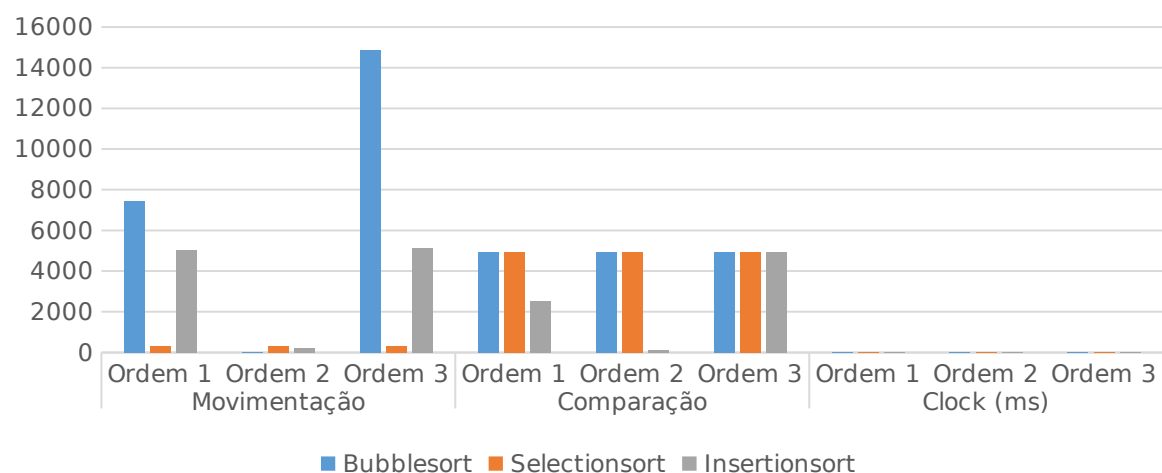
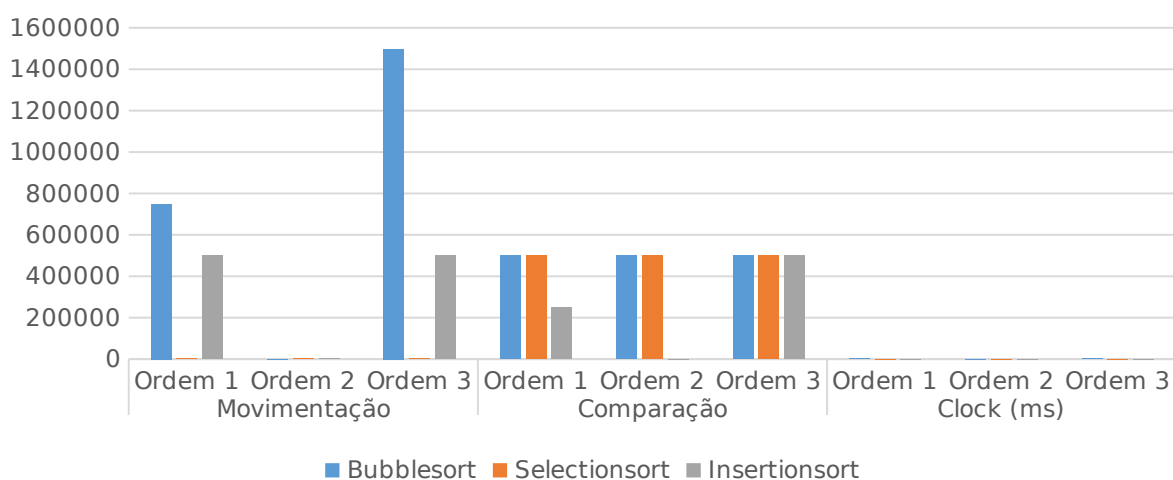




TABELA DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT

Vetor [1.000]									
	Movimentação			Comparação			Clock (ms)		
	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3
Bubblesort	749250	0	1498500	499500	499500	499500	1.718	0.936	1.651
Selectionsort	2997	2997	2997	499500	499500	499500	1.001	0.986	1.005
Insertionsort	500499	1998	501498	250250	999	499500	0.661	0.004	1.074

GRÁFICO DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT



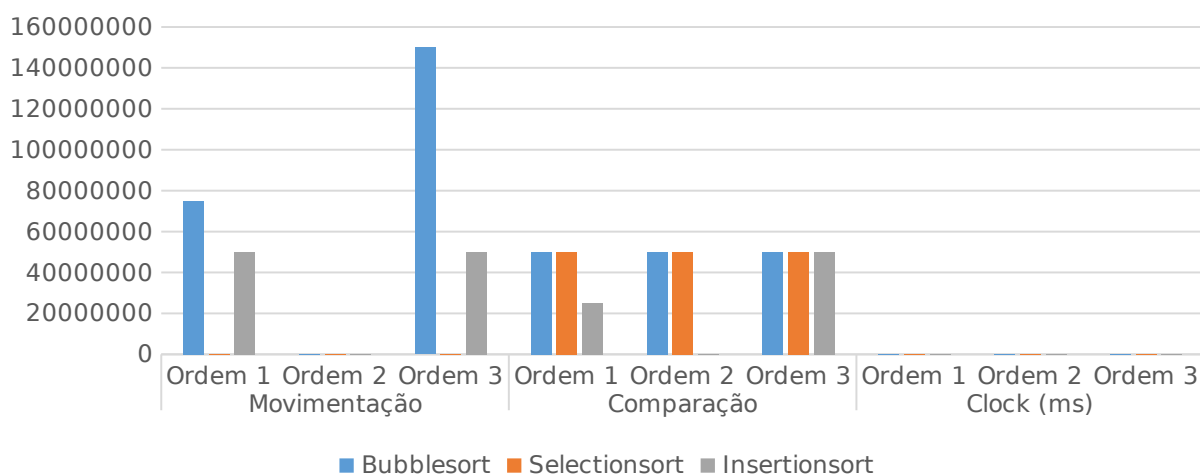


UFMT 4.3 VETOR [10.000]

TABELA DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT

Vetor [10.000]									
	Movimentação			Comparação			Clock (ms)		
	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3	Ordem 1	Ordem 2	Ordem 3
Bubblesort	74992500	0	149985000	49995000	49995000	49995000	208.779	98.244	170.78
Selectionsort	29997	29997	29997	49995000	49995000	49995000	101.197	97.711	97.864
Insertionsort	50004999	19998	50014998	25002500	9999	49995000	53.844	0.032	110.953

GRÁFICO DE COMPARAÇÕES BUBBLESORT, SELECTIONSORT E INSERTIONSORT



5. CONCLUSÃO

O projeto teve como principal finalidade demonstrar os diferentes tipos de ordenação que podem ser realizadas em um algoritmo, tendo como principal foco de comparação o tempo de processamento para cada modo de ordenação.

Durante o desenvolver do código, uma das principais dificuldade apresentadas foi a inserção do contador de movimentações e comparações nas quatro formas de ordenação mais complexas (*shellsort*, *mergesort*, *heapsort* e *quicksort*).



6. REFERÊNCIAS BIBLIOGRÁFICAS

BACKES, A. **Linguagem C – Completa e Descomplicada**, 2ª Edição. São Paulo: GEN LTC, 2018.

Algoritmo	Comparações			Movimentações			Espaço	Estável	In situ
	Melhor	Médio	Pior	Melhor	Médio	Pior			
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$	Sim	Sim
Selection	$O(n^2)$			$O(n)$			$O(1)$	Não*	Sim
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$	Sim	Sim
Merge	$O(n \log n)$			–			$O(n)$	Sim	Não
Quick	$O(n \log n)$		$O(n^2)$	–			$O(n)$	Não*	Sim
Shell	$O(n^{1.25})$ ou $O(n (\ln n)^2)$			–			$O(1)$	Não	Sim

* Existem versões estáveis.

Algoritmo	C_{\min}	$C_{\text{méd}}$	C_{\max}	M_{\min}	$M_{\text{méd}}$	M_{\max}
Inserção Direta	$O(N)$	$O(N^2)$	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
Inserção Binária	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N)$	$O(N^2)$	$O(N^2)$
Seleção Direta	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$	$O(N)$	$O(N)$
Bubblesort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	$O(N^2)$	$O(N^2)$
Shakersort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	$O(N^2)$	$O(N^2)$
Heapsort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
Quicksort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$