PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Graduação em Engenharia de Computação

Gabriel Henrique Silva Pinto

ALGORITMOS DE ORDENAÇÃO
análise de complexidade, medição e desempenho

Belo Horizonte
2020

SUMÁRIO

1 ALGORITMOS DE ORDENAÇÃO

1.1 Função de Custo

Para se chegar à função de custo, normalmente se conta quantas instruções são executadas pelo algoritmo para resolver um problema. A função de custo é expressa por um polinômio, em relação ao tamanho da entrada.

```
for(int i = 0; i < n; i++) {
    print(i);
}
```

Como exemplo, no algoritmo acima, poderíamos dizer que o tempo gasto é:

T(n)=

- n * (Tempo gasto para comparar i com n)
- n * (Tempo para incrementar i)
- n * (Tempo para imprimir i)

A complexidade utilizando a notação assintótica, geralmente mais usada, tem como fator determinante o descarte de constantes e valores pequenos para n, e concentração exclusivamente em valores enormes para n. Dessa forma, as funções n^2, $10n^2$, $100n^2$, $1000n^2$, $10000n^2$, etc., sendo todas de mesma ordem, crescem igualmente em velocidade e são, portanto, equivalentes. Este comportamento, considerando valores enormes para n, é chamado de comportamento assintótico.

Nesse sentido, faz-se necessário focar na rapidez que uma função cresce de acordo com o tamanho da entrada, a taxa de crescimento do tempo de execução. Considerando que um algoritmo, sendo executado com uma entrada de tamanho n, leve $5n^2 + 50n + 500$ instruções de máquina, observa-se que o termo n^2 cresce tão rapidamente que se torna desprezível a presença da constante que o multiplica e dos demais termos, sobrando, então, apenas n^2.

2 BUBBLESORT

2.1 Implementação

```
int i, j;
for (i = (n - 1); i > 0; i--) {
  for (j = 0; j < i; j++) {
     if (array[j] > array[j + 1]) {
        swap(&array[j], &array[j + 1]);
     }
  }
}
```

2.2 Função de Custo

No BubbleSort, podemos dizer que o tempo gasto é:

T(n)=

- Tempo gasto para subtrair 1 em n
- (n – 1) * (Tempo gasto para comparar de i com 0)
- (n – 1) * (Tempo gasto para decrementar i)
- (n – 1) * (Tempo gasto para comparar j com i)
- (n – 1) * (Tempo gasto para incrementar j)
- (n – 1) * (Tempo para somar 1 em j)
- (n – 1) * (Tempo para comparar o array na posição j com o array na posição j + 1)
- (n – 1) * (Tempo para trocar o valor das posições j e j + 1 do array caso a comparação acima seja satisfeita)

O BubbleSort tem como principal vantagem a simplicidade do algoritmo, porém, por ser de ordem quadrática, não é recomendado para programas que precisem de rapidez e/ou trabalhem com uma grande massa de dados.

2.3 Complexidade

- Melhor caso: O(n)
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$
- Espacial: O(1)

2.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

2.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	8680.0	0.5
2ª Execução	8717.0	0.4
3ª Execução	8753.0	0.5
4ª Execução	8548.0	0.5
5ª Execução	8703.0	0.5
Resultado *(média 2ª, 3ª e 4ª)*	8551.0	0.466

2.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	5284.0	1.7133
2ª Execução	6163.0	1.7127
3ª Execução	5305.0	1.7133
4ª Execução	5477.0	1.7133
5ª Execução	5418.0	1.7133
Resultado *(média 2ª, 3ª e 4ª)*	5480.0	1.7131

3 COUNTINGSORT

3.1 Implementação

```
int maior = array[0];
for (int i = 0; i < n; i++) {
    if(maior < array[i]){
        maior = array[i];
    }
}
return maior;
```

3.2 Função de Custo

No CountingSort, podemos dizer que o tempo gasto é:

T(n)=

- n * (Tempo gasto para comparar de i com n)
- n * (Tempo gasto para incrementar i)
- n * (Tempo gasto para comparar o maior com o array na posição i)

O CountingSort tem como principal vantagem a estabilidade do algoritmo, sendo de ordem linear. Em sua estrutura original, este algoritmo não é capaz de ordenar numeros negativos, uma vez que não existe índice de matriz negativo.

3.3 Complexidade

- Melhor caso: O(n + k)
- Caso médio: O(n + k)
- Pior caso: O(n + k)
- Espacial: O(n + k)

3.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

3.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	2.0	0.9
2ª Execução	2.0	0.9
3ª Execução	4.0	0.8
4ª Execução	5.0	0.9
5ª Execução	2.0	0.9
Resultado *(média 2ª, 3ª e 4ª)*	3.66	0.86

3.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	5.0	2.0
2ª Execução	5.0	2.0
3ª Execução	4.0	2.0
4ª Execução	3.0	2.0
5ª Execução	5.0	2.0
Resultado *(média 2ª, 3ª e 4ª)*	4.0	2.0

4 HEAPSORT

4.1 Implementação

```
void construir(int *array, int tamHeap){
    for(int i = tamHeap; i > 1 && array[i] > array[i/2]; i /= 2){
        swap(array + i, array + i/2);
    }
}
//=============================================================================
int getMaiorFilho(int *array, int i, int tamHeap){
    int filho;
    if (2*i == tamHeap || array[2*i] > array[2*i+1]){
        filho = 2*i;
    } else {
        filho = 2*i + 1;
    }
    return filho;
}
//=============================================================================
void reconstruir(int *array, int tamHeap){
    int i = 1;
    while(i <= (tamHeap/2)){
        int filho = getMaiorFilho(array, i, tamHeap);
        if(array[i] < array[filho]){
            swap(array + i, array + filho);
            i = filho;
        }else{
            i = tamHeap;
        }
    }
}
//=============================================================================
void heapsort(int *array, int n) {
    //Alterar o vetor ignorando a posicao zero
    int arrayTmp[n+1];
    for(int i = 0; i < n; i++){
        arrayTmp[i+1] = array[i];
    }

    //Contrucao do heap
    for(int tamHeap = 2; tamHeap <= n; tamHeap++){
        construir(arrayTmp, tamHeap);
    }

    //Ordenacao propriamente dita
    int tamHeap = n;
    while(tamHeap > 1){
        swap(arrayTmp + 1, arrayTmp + tamHeap--);
        reconstruir(arrayTmp, tamHeap);
    }

    //Alterar o vetor para voltar a posicao zero
    for(int i = 0; i < n; i++){
        array[i] = arrayTmp[i+1];
    }
}
```

4.2 Função de Custo

No HeapSort, podemos dizer que o tempo gasto é:

T(n)=

- n * (Tempo gasto para comparar i com n)
- n * (Tempo gasto para incrementar i)
- ...

O HeapSort tem como principal vantagem o desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio.

4.3 Complexidade

- Melhor caso: Θ(n log n)
- Caso médio: Θ(n log n)
- Pior caso: Θ(n log n)
- Espacial: Θ(n) / Θ(1)

4.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

4.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	16.0	0.7
2ª Execução	31.0	0.7
3ª Execução	18.0	0.7
4ª Execução	18.0	0.7
5ª Execução	15.0	0.7
Resultado *(média 2ª, 3ª e 4ª)*	21.33	22.33

4.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	25.0	2.0
2ª Execução	18.0	2.0
3ª Execução	22.0	2.0
4ª Execução	25.0	2.0
5ª Execução	21.0	2.0
Resultado *(média 2ª, 3ª e 4ª)*	18.3	2.0

5 INSERTIONSORT

5.1 Implementação

```
for (int i = 1; i < n; i++) {
  int tmp = array[i];
  int j = i - 1;

  while ((j >= 0) && (array[j] > tmp)) {
     array[j + 1] = array[j];
     j--;
  }
  array[j+1] = tmp;
}
```

5.2 Função de Custo

No InsertionSort, podemos dizer que o tempo gasto é:

T(n)=

- (n – 1) * (Tempo gasto para comparar de i com n)
- (n – 1) * (Tempo gasto para incrementar i)
- (n – 1) * (Tempo gasto para subtrair 1 de i)
- ...

O InsertionSort tem como principal vantagem a eficiência para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

5.3 Complexidade

- Melhor caso: O(n)
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$
- Espacial: O(n) / O(1)

5.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

5.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	1909.0	0.6
2ª Execução	1872.0	0.5
3ª Execução	2020.0	0.5
4ª Execução	2051.0	0.5
5ª Execução	2109.0	0.5
Resultado *(média 2ª, 3ª e 4ª)*	1981.0	0.5

5.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	253.0	1.7133
2ª Execução	269.0	1.7133
3ª Execução	269.0	1.7133
4ª Execução	266.0	1.7133
5ª Execução	269.0	1.7133
Resultado *(média 2ª, 3ª e 4ª)*	268.0	1.7133

6 MERGESORT

6.1 Implementação

```
void mergesort(int *array, int n) {
   mergesortRec(array, 0, n-1);
}
//=============================================================================
void mergesortRec(int *array, int esq, int dir){
   if (esq < dir){
      int meio = (esq + dir) / 2;
      mergesortRec(array, esq, meio);
      mergesortRec(array, meio + 1, dir);
      intercalar(array, esq, meio, dir);
   }
}
//=============================================================================
void intercalar(int* array, int esq, int meio, int dir){
   int n1, n2, i, j, k;

   //Definir tamanho dos dois subarrays
   n1 = meio-esq+1;
   n2 = dir - meio;

   int* a1 = (int*) malloc((n1+1) * sizeof(int));
   int* a2 = (int*) malloc((n2+1) * sizeof(int));

   //Inicializar primeiro subarray
   for(i = 0; i < n1; i++){
      a1[i] = array[esq+i];
   }

   //Inicializar segundo subarray
   for(j = 0; j < n2; j++){
      a2[j] = array[meio+j+1];
   }

   //Sentinela no final dos dois arrays
   a1[i] = a2[j] = 0x7FFFFFFF;


   //Intercalacao propriamente dita
   for(i = j = 0, k = esq; k <= dir; k++){
      array[k] = (a1[i] <= a2[j]) ? a1[i++] : a2[j++];
   }
}
```

6.2 Função de Custo

No MergeSort, podemos dizer que o tempo gasto é:

T(n)=

- ...

Como o MergeSort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

6.3 Complexidade

- Melhor caso: Θ(n log n) / Θ(n)
- Caso médio: Θ(n log n)
- Pior caso: Θ(n log n)
- Espacial: Θ(n log n)

6.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

6.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	15.0	5.4
2ª Execução	16.0	5.3
3ª Execução	31.0	5.4
4ª Execução	16.0	5.3
5ª Execução	22.0	5.3
Resultado *(média 2ª, 3ª e 4ª)*	21.0	5.33

6.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	19.0	6.63
2ª Execução	18.0	6.63
3ª Execução	18.0	6.63
4ª Execução	22.0	6.63
5ª Execução	22.0	6.63
Resultado *(média 2ª, 3ª e 4ª)*	19.33	6.63

7 QUICKSORT

7.1 Implementação

```
void quicksortRec(int *array, int esq, int dir) {
    int i = esq, j = dir;
    int pivo = array[(dir+esq)/2];
    while (i <= j) {
        while (array[i] < pivo) i++;
        while (array[j] > pivo) j--;
        if (i <= j) {
            swap(array + i, array + j);
            i++;
            j--;
        }
    }
    if (esq < j)  quicksortRec(array, esq, j);
    if (i < dir)  quicksortRec(array, i, dir);
}
//=============================================================================
void quicksort(int *array, int n) {
    quicksortRec(array, 0, n-1);
}
```

7.2 Função de Custo

No QuickSort, podemos dizer que o tempo gasto é:

T(n)=

- ...

O QuickSort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

7.3 Complexidade

- Melhor caso: O(n log n) / O(n)
- Caso médio: O(n log n)
- Pior caso: $O(n^2)$
- Espacial: O(n)

7.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

7.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	15.0	0.5
2ª Execução	15.0	0.5
3ª Execução	9.0	0.5
4ª Execução	7.0	0.5
5ª Execução	11.0	0.5
Resultado *(média 2ª, 3ª e 4ª)*	10.33	0.5

7.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	33.0	1.713
2ª Execução	15.0	1.713
3ª Execução	17.0	1.713
4ª Execução	18.0	1.713
5ª Execução	32.0	1.713
Resultado *(média 2ª, 3ª e 4ª)*	16.66	1.713

8 SELECTIONSORT

8.1 Implementação

```
for (int i = 0; i < (n - 1); i++) {
  int menor = i;
  for (int j = (i + 1); j < n; j++){
     if (array[menor] > array[j]){
        menor = j;
     }
  }
  swap(&array[menor], &array[i]);
}
```

8.2 Função de Custo

No SelectionSort, podemos dizer que o tempo gasto é:

T(n)=

- (n - 1) * (Tempo gasto para subtrair 1 de n)
- (n - 1) * (Tempo gasto para comparar i com n - 1)
- (n - 1) * (Tempo gasto para incrementar i)
- (n - 1) * (n - i - 1) * (Tempo gasto para adicionar 1 em i)
- (n - 1) * (n - i - 1) * (Tempo gasto para comparar j com n)
- (n - 1) * (n - i - 1) * (Tempo gasto para incrementar j)
- (n - 1) * (n - i - 1) * (Tempo gasto para comparar o array na posição "menor"com o array na posição "j")
- (n - 1) * (Tempo gasto para trocar a posição do array na posição "menor"com o array na posição "j")

O SelectionSort tem como principais vantagens a fácil implementação do algoritmo, além de ocupar pouca memória se comparado a algoritmos como quick e merge sort

8.3 Complexidade

- Melhor caso: $O(n^2)$
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$
- Espacial: O(n) / O(1)

8.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

8.4.1 Linguagem C

	Tempo (ms)	**Memória (mb)**
1ª Execução	3037.0	0.6
2ª Execução	2833.0	0.5
3ª Execução	2802.0	0.5
4ª Execução	2831.0	0.5
5ª Execução	2823.0	0.5
Resultado *(média 2ª, 3ª e 4ª)*	2822.0	0.5

8.4.2 Linguagem Java

	Tempo (ms)	**Memória (mb)**
1ª Execução	1260.0	1.713
2ª Execução	1283.0	1.713
3ª Execução	1249.0	1.713
4ª Execução	1260.0	1.713
5ª Execução	1254.0	1.713
Resultado *(média 2ª, 3ª e 4ª)*	1264.0	1.713

9 SHELLSORT

9.1 Implementação

```
void insercaoPorCor(int *array, int n, int cor, int h){
    for (int i = (h + cor); i < n; i+=h) {
        int tmp = array[i];
        int j = i - h;
        while ((j >= 0) && (array[j] > tmp)) {
            array[j + h] = array[j];
            j-=h;
        }
        array[j + h] = tmp;
    }
}
//=============================================================================
void shellsort(int *array, int n) {
    int h = 1;

    do { h = (h * 3) + 1; } while (h < n);

    do {
        h /= 3;
        for(int cor = 0; cor < h; cor++){
            insercaoPorCor(array, n, cor, h);
        }
    } while (h != 1);
}
```

9.2 Função de Custo

No ShellSort, podemos dizer que o tempo gasto é:

T(n)=

- ...

O ShellSort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. Sua principal vantagem é a simples implementação e a pequena requisição de código.

9.3 Complexidade

- Melhor caso: O(n log n)
- Caso médio: Depende
- Pior caso: O(n log n)
- Espacial: O(n)

9.4 Experimento

Considerando um vetor com 50.000 elementos gerados aleatoriamente, temos o seguinte resultado:

9.4.1 Linguagem C

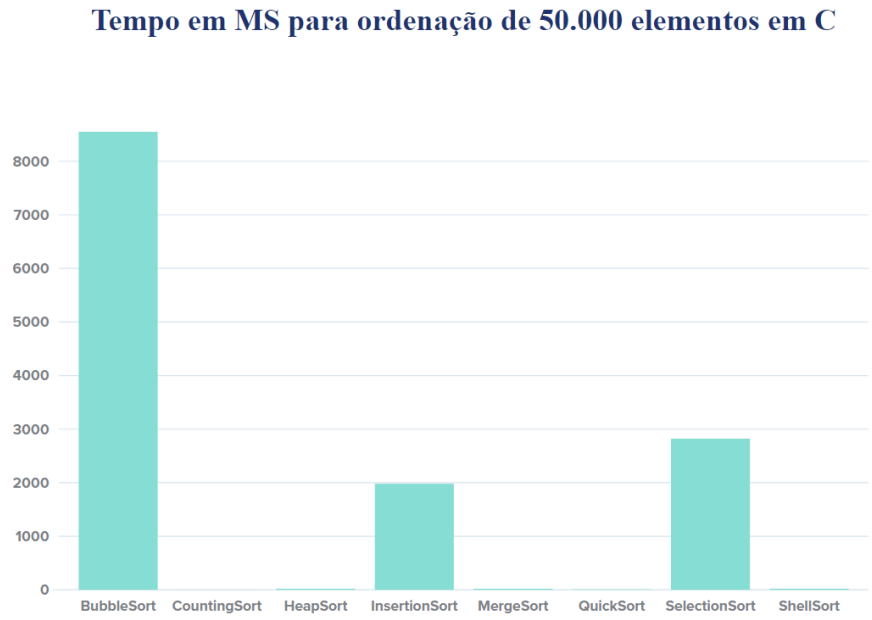
	Tempo (ms)	**Memória (mb)**
1ª Execução	15.0	0.5
2ª Execução	16.0	0.5
3ª Execução	31.0	0.5
4ª Execução	19.0	0.5
5ª Execução	14.0	0.5
Resultado *(média 2ª, 3ª e 4ª)*	22.0	0.5

9.4.2 Linguagem Java

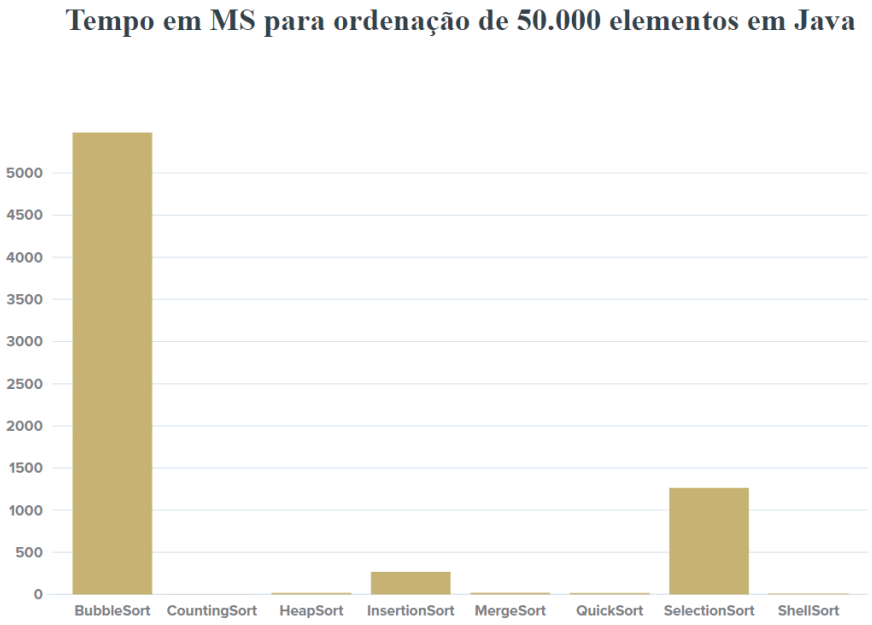
	Tempo (ms)	**Memória (mb)**
1ª Execução	11.0	1.713
2ª Execução	11.0	1.713
3ª Execução	9.0	1.713
4ª Execução	10.0	1.713
5ª Execução	14.0	1.713
Resultado *(média 2ª, 3ª e 4ª)*	10.0	1.713

10 TEMPO DE EXECUÇÃO - REPRESENTAÇÃO EM GRÁFICO

10.1 Tempo em C

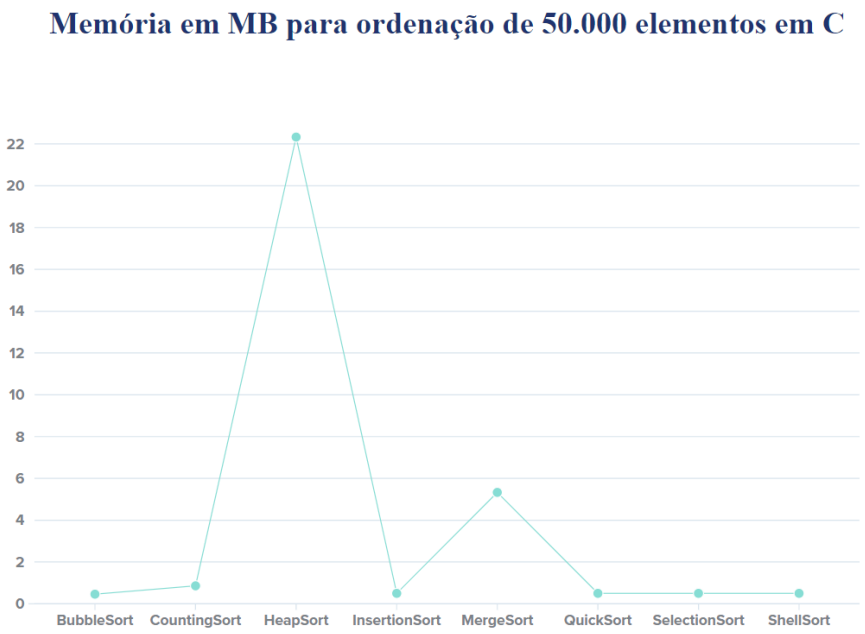


10.2 Tempo em Java

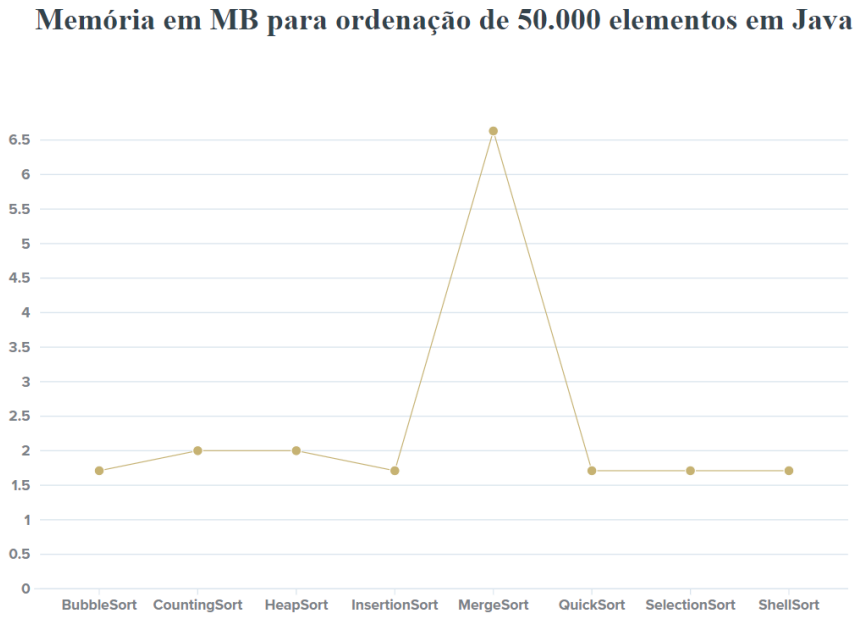


11 MEMÓRIA CONSUMIDA - REPRESENTAÇÃO EM GRÁFICO

11.1 Memória em C



11.2 Memória em Java



12 CONCLUSÃO

12.1 BubbleSort

Simples e de fácil entendimento, está entre os métodos de ordenação mais difundidos existentes. Em contra-partida, não é um algoritmo eficiente (sua eficiência diminui drasticamente a medida que o número de elementos no array aumenta, então é estudado apenas para fins de desenvolvimento de raciocínio).

12.2 CountingSort

Estável (não altera a ordem dos dados iguais), processamento simples. Porém, não recomendado para grandes conjuntos de dados (K muito grande) e não ordena valores fracionários e/ou negativos.

12.3 HeapSort

Comportamento sempre O(n log n), qualquer que seja a entrada. Porém, não é estável e o anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.

12.4 InsertionSort

Fácil implementação, estável, pode ordenar elementos em tempo real e, na prática, é mais eficiente que a maioria dos algoritmos de ordem quadrática (como o selection sort e o bubble sort), um dos mais rápidos algoritmos de ordenação para conjuntos pequenos de dados (superando inclusive o quick sort).

12.5 MergeSort

Estável (não altera a ordem dos dados iguais). Entretanto, possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação, cria uma cópia do array para cada chamada recursiva.

12.6 QuickSort

Apesar de seu pior caso ser quadrático, costuma ser a melhor opção prática para ordenação de grandes conjuntos de dados. Porém, não é um algoritmo estável.

12.7 SelectionSort

Estável (não altera a ordem dos dados iguais). Entretanto, sua eficiência diminui drasticamente a medida que o número de elementos no array aumenta.

12.8 ShellSort

Uma ótima opção para arquivos de tamanho moderado, implementação é simples e requer uma quantidade de código pequena. Porém, não é estável e o tempo de execução do algoritmo é sensível à ordem inicial do arquivo.

13 BIBLIOGRAFIA

https://pt.wikipedia.org/wiki/Bubble$_s ort$
$https://pt.wikipedia.org/wiki/Counting_s ort$
$https://pt.wikipedia.org/wiki/Heapsort$
$https://pt.wikipedia.org/wiki/Insertion_s ort$
$https://pt.wikipedia.org/wiki/Merge_s ort$
$https://pt.wikipedia.org/wiki/Quicksort$
$https://pt.wikipedia.org/wiki/Selection_s ort$
$https://pt.wikipedia.org/wiki/Shell_s ort$
$http://www.facom.ufu.br/\ backes/gsi011/Aula06 - Ordenacao.pdf$
$https://homepages.dcc.ufmg.br/\ cunha/teaching/20121/aeds2/shellsort.pdf$
$http://www.decom.ufop.br/anascimento/site_m edia/uploads/bcc202/aula_16_f ila_d e_p rioridade_e h eapsort.$