
UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DE PROJETO E ANÁLISE DE ALGORITMOS

# ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

YURI DE SOUSA NASCIMENTO

PALMAS (TO)
2023

**SUMÁRIO**

# 1 INTRODUÇÃO

Conforme Cormen (CORMEN et al., 2022), um algoritmo é qualquer procedimento computacional bem definido que recebe algum valor, ou um conjunto de valores, como sua entrada e produz algum valor, ou um conjunto de valores, como sua saída, em uma quantidade finita de tempo.

Uma instância de um problema consiste de uma entrada – tal que ela satisfaça as restrições impostas pelo problema – necessária para computar a solução do problema.

Diferentes algoritmos criados para solucionar o mesmo problema geralmente diferem drasticamente quanto às suas eficiências. Nesse sentido, o problema analisado, no presente trabalho, refere ao problema de ordenação. Onde, dada uma entrada contendo uma sequência de $n$ números $\{a_1, a_2, ..., a_n\}$, temos, como saída, a sequência $\{a_1 \leq a_2 \leq a_3 ... \leq a_n\}$. Desse modo, para determinar se um algoritmo é mais eficiente que outro, pode-se usar de duas abordagem:

- Análise empírica: comparação entre os programas.
- Análise matemática: estudo das propriedades do algoritmo.

No presente relatório será abordado a análise empírica, sendo que ela avalia o custo/complexidade de um algoritmo a partir da avaliação da execução dele, quando implementado. Por meio dessa análise, pode-se:

- avaliar o desempenho em uma determinada configuração de computador/linguagem.
- comparar computadores
- comparar linguagens

Apesar de algumas vantagens, a análise empírica tem suas desvantagens, como, por exemplo, se o programa for rodado várias vezes com a mesma instância, o tempo de execução necessário para processar tal instância pode variar. Além disso, para algumas instâncias, limitações de hardware podem limitar a análise.

# 2 MÉTODOS

## 2.1 Análise

Para realizar as análises empíricas dos algoritmos de ordenação, os seguintes algoritmos foram implementados na linguagem C. Foram eles Bubble, Gnome, Heap, Insertion, Merge, Quick, Selection, Shell, e Tim sort. Tais algoritmos foram retirados das seguintes fontes: (CORMEN et al., 2022), (BACKES, 2022) e (GEEKS, 2023). Para analisar tais algoritmos, foi-se passado para cada um deles uma instância contendo elementos em ordem aleatória, crescente e decrescente tal que essas instância possuíam $1 \times 10^3, 1 \times 10^4, \{1, 2, ..., 8, 9\} \times 10^5$ e $1 \times 10^6$ elementos. Para medir e comparar a eficiência do processamento dos algoritmos para as diferentes instâncias passadas, foi-se contabilizado a quantidade de comparações e trocas que cada algoritmo realizou, bem como o tempo de execução necessário para ordenar a instância.

É importante deixar claro que, como é uma análise empírica, o tempo de execução dos algoritmos testados aqui pode diferir de máquina para máquina, já que esse fator é dependente de hardware, visto que diferentes CPUs executam diferentes quantidades de instruções por segundo. Ainda sobre tempo de execução, se muitos processos estiverem rodando no mesmo momento que o algoritmo estiver executando, o tempo para processar a instância será mais demorado, uma vez que os núcleos de processamento terão de ser compartilhados entre os demais processos. Além disso, a quantidade de comparações e trocas que cada algoritmo realizou para determinadas instâncias podem diferir a depender da implementação para um mesmo algoritmo. Quero dizer, uma de duas implementações diferentes de um Bubble sort pode realizar menos comparações e trocas de elementos que uma outra implementação.

## 2.2 Hardware usado

Para a realização da análise empírica, os algoritmos foram executados em uma máquina contendo as seguintes especificações:

- CPU: Ryzen 3 2200G de 3.6 Ghz, quad core
- GPU: AMD Radeon RX 570, 4G
- RAM: HyperX 8GB 2666mhz, DDR4

# 3 IMPLEMENTAÇÕES DOS ALGORITMOS

## 3.1 Bubblesort

O algoritmo *Bubblesort* trabalha de forma a movimentar, uma posição por vez, o maior valor existente na porção não ordenada de um array para a sua respectiva posição no array ordenado. Isso é repetido até que todos os elementos estejam nas suas posições correspondentes.

```
1  void bubblesort(long int* vet, size_t n){
2
3      for(long int i = 0; i < n; ++i){
4          for(long int j = n-1; j > i; --j){
5              ++comp;
6              if(vet[j-1] > vet[j]){
7                  exchange(&vet[j], &vet[j-1]);
8                  ++troca;
9              }
10         }
11     }
12 }
13 }
```

## 3.2 Gnomesort

O algoritmo *Gnomesort* é uma variação do insertionsort que não usa loops anhinhados. A ideia desse algoritmo é verificar os elementos $i$ e $i+1$ e, se eles estiverem na ordem correta, verificar os elementos seguintes; caso o os elementos não estejam na ordem correta, trocar $i+1$ com elementos anteriores até que ele esteje na posição correta.

```
1  void gnomesort(long int *vet, size_t n){
2      long int index = 0;
3
4      while(index < n){
5          ++comp;
6          if(index == 0)
7              index++;
8          if(vet[index -1] <= vet[index])
9              index++;
10         else{
11             exchange(&vet[index], &vet[index -1]);
```

```
12              index--;
13              ++troca;
14          }
15      }
16  }
```

### 3.3 Heapsort

O algoritmo *Heapsort* simula uma árvore binária completa (exceção do último nível) a partir de um array. Cada posição i do array passa a ser considerada o pai de duas outras posições, chamadas filhos: (2i + 1) e (2i + 2). Em seguida, o algoritmo reorganiza o array para que o pai seja sempre maior que os dois filhos. Ao fim desse processo, o elemento que é pai de todos é também o maior elemento do array. Este elemento poderá ser removido do heap e colocado na última posição do array e esse processo continua para o restante do heap/array.

```
1  void build_heap(long int* vet, size_t pai, size_t fim){
2      long int aux = vet[pai];
3      long int filho = 2 * pai + 1;
4
5      while(filho <= fim){
6          if(filho < fim){
7              if(vet[filho] < vet[filho +1]){
8                  ++filho;
9              }
10         }
11
12         ++comp;
13         if(aux < vet[filho]){
14             vet[pai] = vet[filho];
15             pai = filho;
16             filho = 2 * pai +1;
17             ++troca;
18         }else{
19             filho = fim +1;
20         }
21     }
22     vet[pai] = aux;
23 }
24
25 void heapsort(long int* vet, size_t size){
26     for(long int i = (size-1)/2; i >= 0; --i){
27         build_heap(vet, i, size-1);
28     }
```

```
29      for(long int i = size-1; i >= 1; --i){
30          exchange(&vet[0], &vet[i]);
31          build_heap(vet, 0, i-1);
32      }
33  }
```

### 3.4 Insertionsort

O algoritmo *Insertionsort* percorre um array e, para cada posição X, verifica se o seu valor está na posição correta. Isso é feito andando para o começo do array a partir da posição X e movimentando uma posição para frente os valores que são maiores do que o valor da posição X. Desse modo, teremos uma posição livre para inserir o valor da posição X em seu devido lugar.

```
1  void insertionsort(long int* vet, long int left, long int right){
2      for (size_t i = left+1; i < right; i++){
3          long int key = vet[i];
4          long int j = i -1;
5          ++comp;
6          while(j >= left && key < vet[j]){
7              vet[j+1] = vet[j];
8              --j;
9              ++troca;
10         }
11         vet[j+1] = key;
12     }
13 }
```

### 3.5 Mergesort

O algoritmo *Mergesort* divide, recursivamente, o array em duas partes até que cada posição dele seja considerada como um array de um único elemento. Em seguida, o algoritmo combina dois arrays de forma a obter um array maior e ordenado. Essa combinação dos arrays é feita intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). Esse processo se repete até que exista apenas um array.

```
1  void merge(long int* vet, long int l, long int m, long int r){
2      long int len1 = m -l +1, len2 = r -m;
3      long int left[len1], right[len2];
4
5      for (size_t i = 0; i < len1; i++)
6          left[i] = vet[l + i]; // Filling left array
```

```
7     for (size_t i = 0; i < len2; i++)
8         right[i] = vet[m +1 +i]; // Filling right array
9
10    size_t i = 0;
11    size_t j = 0;
12    size_t k = l;
13    while (i < len1 && j < len2){ // Iterate into both arrays left and
          right{
14          ++comp;
15        if (left[i] <= right[j]){ // IF element in left is less then
              increment i by pushing into larger array{
16            vet[k] = left[i];
17            i++;
18        } else {
19            vet[k] = right[j]; // Element in right array is greater
20            j++;
21            ++troca;
22        }
23        k++;
24    }
25    while (i < len1){ // This loop copies remaining element in left array
          {
26        vet[k] = left[i];
27        k++;
28        i++;
29    }
30    while (j < len2){ // This loop copies remaining element in right
          array{
31        vet[k] = right[j];
32        k++;
33        j++;
34    }
35 }
36
37 void mergesort(long int* vet, size_t inicio, size_t fim){
38
39          if(inicio < fim){
40                  size_t meio = (inicio+fim)/2;
41                  mergesort(vet, inicio, meio);
42                  mergesort(vet, meio+1, fim);
43                  merge(vet, inicio, meio, fim);
44          }
45 }
```

### 3.6 Quicksort

O algoritmo *Quicksort*, primeiramente, rearranja os valores de array de modo que os valores menores que o valor do pivô fiquem na parte esquerda do array, enquanto os valores maiores que o pivô ficam na parte direita. Supondo um array de tamanho N e que o pivô esteja na posição X, esse processo cria duas partições: [0, ..., X – 1] e [X + 1, ..., N – 1]. Em seguida, o algoritmo é aplicado recursivamente a cada partição. Esse processo se repete até que cada partição contenha um único elemento.

```
1  long int partition(long int* vet, long int inicio, long int final){
2      long int key = vet[final], iterador = inicio -1;
3
4      for(long int j = inicio; j < final; ++j){
5          ++comp;
6          if(vet[j] < key){
7              ++iterador;
8              exchange(&vet[iterador], &vet[j]);
9              ++troca;
10         }
11     }
12     exchange(&vet[iterador+1], &vet[final]);
13     ++troca;
14     return iterador +1;
15 }
16
17
18 void quicksort(long int* vet, long int inicio, long int fim){
19     if(inicio < fim){
20         long int pivo = partition(vet, inicio, fim);
21         quicksort(vet, inicio, pivo -1);
22         quicksort(vet, pivo +1, fim);
23     }
24 }
```

### 3.7 Selectionsort

O algoritmo *Selectionsort* divide o array em duas partes: a parte ordenada, à esquerda do elemento analisado, e a parte que ainda não foi ordenada, à direita do elemento. Para cada elemento do array, começando do primeiro, o algoritmo procura na parte não ordenada (direita) o menor valor (ordenação crescente) e troca os dois valores de lugar. Em seguida, o algoritmo avança para a próxima posição do array e repete esse processo. Isso é feito até que todo o array esteja ordenado.

```
1  void selectionsort(long int* vet, size_t n){
2
3      for(size_t i = 0; i < n-1; ++i){
4          size_t menor = i;
5
6          for(size_t j = i+1; j < n; ++j){
7              ++comp;
8              if(vet[j] < vet[menor]){
9                  menor = j;
10             }
11         }
12
13         if(i != menor){
14             exchange(&vet[i], &vet[menor]);
15             ++troca;
16         }
17     }
18 }
```

### 3.8 Shellsort

O *Shellsort* é uma otimização do insertionsort que permite a troca de elementos que estão distantes. A ideia é ordenar a lista de elementos para que, iniciando em qualquer lugar, tomando uma quantidade h de elemetentos, ela seja ordenada; tal que essa quantidade h de elementos é chamada de h-ardenada.

```
1  void shellsort(long int* vet, size_t n){
2      for (long int gap = n/2; gap > 0; gap /= 2)
3      {
4          for (size_t i = gap; i < n; i += 1)
5          {
6              long int temp = vet[i];
7              ++comp;
8              long int j;
9              for (j = i; j >= gap && vet[j -gap] > temp; j -= gap){
10                 vet[j] = vet[j -gap];
11                 ++troca;
12             }
13             vet[j] = temp;
14         }
15     }
16 }
```

### 3.9 Timsort

O *Timsort* é um algoritmo de ordenação baseado nos algoritmos Insertion e Merge sort. A ideia é dividir o vetor em blocos chamados de RUN, ordenar esses blocos, um por um, usando o Insertionsort e, então, unir esses blocos usando a função Merge, do Mergesort. Se o tamanho do vetor for menor que o tamanho de um bloco RUN, então ele é ordenado usando o Insertionsort.

```
1  const int RUN = 32;
2  void timsort(long int* vet, size_t n){
3      for (size_t i=0; i < n; i+=RUN)
4          insertionsort(vet, i, min((i+31), n)); //Call insertionSort()
5
6      for (size_t s = RUN; s < n; s = 2*s){
7          for (size_t left = 0; left < n; left += 2*s){
8              long int mid = left + s - 1;
9              long int right = min((left + 2*s - 1), (n-1));
10             if(mid < right)
11                 merge(vet, left, mid, right); // merge sub array
12         }
13     }
14 }
```

# 4 RESULTADOS

A seguir são apresentadas tabelas que mostram o comportamento dos algoritmos para instâncias aleatória, crescente e decrescente, com quantidade de elementos de $1 \times 10^3, 1 \times 10^4, \{1, 2, ..., 8, 9\} \times 10^5$ e $1 \times 10^6$.

## 4.1 Tabelas

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.005514 | 499500 | 249185 |
| 10000 | 0.300751 | 49995000 | 24975864 |
| 100000 | 32.723894 | 4999950000 | 2503447952 |
| 200000 | 171.666300 | 19999900000 | 9972990404 |
| 300000 | 388.640345 | 44999850000 | 22450657498 |
| 400000 | 686.498350 | 79999800000 | 40027774620 |
| 500000 | 1059.212025 | 124999750000 | 62394784027 |
| 600000 | 1522.274723 | 179999700000 | 90007121038 |
| 700000 | 2107.295848 | 244999650000 | 122787869370 |
| 800000 | 2764.840854 | 319999600000 | 159986677534 |
| 900000 | 3542.099250 | 404999550000 | 202396643276 |
| 1000000 | 4391.192904 | 499999500000 | 250053898107 |

**Tabela 1 – Bubblesort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.002989 | 499500 | 0 |
| 10000 | 0.092450 | 49995000 | 0 |
| 100000 | 8.651060 | 4999950000 | 0 |
| 200000 | 33.401417 | 19999900000 | 0 |
| 300000 | 79.523717 | 44999850000 | 0 |
| 400000 | 139.712857 | 79999800000 | 0 |
| 500000 | 231.353135 | 124999750000 | 0 |
| 600000 | 450.526261 | 179999700000 | 0 |
| 700000 | 612.281525 | 244999650000 | 0 |
| 800000 | 802.528626 | 319999600000 | 0 |
| 900000 | 1012.964044 | 404999550000 | 0 |
| 1000000 | 1252.151656 | 499999500000 | 0 |

**Tabela 2 – Bubblesort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.007090 | 499500 | 499500 |
| 10000 | 0.306663 | 49995000 | 49995000 |
| 100000 | 47.558374 | 4999950000 | 4999950000 |
| 200000 | 193.932538 | 19999900000 | 19999900000 |
| 300000 | 309.478123 | 44999850000 | 44999850000 |
| 400000 | 494.990543 | 79999800000 | 79999800000 |
| 500000 | 1144.230130 | 124999750000 | 124999750000 |
| 600000 | 1749.233020 | 179999700000 | 179999700000 |
| 700000 | 1987.995923 | 244999650000 | 244999650000 |
| 800000 | 3003.158171 | 319999600000 | 319999600000 |
| 900000 | 3809.207139 | 404999550000 | 404999550000 |
| 1000000 | 4583.418876 | 499999500000 | 499999500000 |

**Tabela 3 – Bubblesort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.002500 | 499361 | 249185 |
| 10000 | 0.207900 | 49961718 | 24975864 |
| 100000 | 30.618306 | 5006995894 | 2503447952 |
| 200000 | 105.114902 | 19946180791 | 9972990404 |
| 300000 | 242.197568 | 44901614983 | 22450657498 |
| 400000 | 437.111799 | 80055949229 | 40027774620 |
| 500000 | 619.277228 | 124790068042 | 62394784027 |
| 600000 | 955.060298 | 180014842066 | 90007121038 |
| 700000 | 1313.222865 | 245576438724 | 122787869370 |
| 800000 | 1791.881784 | 319974155056 | 159986677534 |
| 900000 | 2278.671625 | 404794186537 | 202396643276 |
| 1000000 | 2900.027902 | 500108796201 | 250053898107 |

**Tabela 4 – Gnomesort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000009 | 999 | 0 |
| 10000 | 0.000069 | 9999 | 0 |
| 100000 | 0.000276 | 99999 | 0 |
| 200000 | 0.000568 | 199999 | 0 |
| 300000 | 0.000820 | 299999 | 0 |
| 400000 | 0.001090 | 399999 | 0 |
| 500000 | 0.001377 | 499999 | 0 |
| 600000 | 0.001629 | 599999 | 0 |
| 700000 | 0.001924 | 699999 | 0 |
| 800000 | 0.002201 | 799999 | 0 |
| 900000 | 0.002496 | 899999 | 0 |
| 1000000 | 0.002726 | 999999 | 0 |

**Tabela 5 – Gnomesort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.010468 | 999000 | 499500 |
| 10000 | 0.446132 | 99990000 | 49995000 |
| 100000 | 42.932450 | 9999900000 | 4999950000 |
| 200000 | 215.838401 | 39999800000 | 19999900000 |
| 300000 | 551.661895 | 89999700000 | 44999850000 |
| 400000 | 744.832737 | 159999600000 | 79999800000 |
| 500000 | 1450.105700 | 249999500000 | 124999750000 |
| 600000 | 1967.578099 | 359999400000 | 179999700000 |
| 700000 | 2792.319604 | 489999300000 | 244999650000 |
| 800000 | 3584.016330 | 639999200000 | 319999600000 |
| 900000 | 4499.910007 | 809999100000 | 404999550000 |
| 1000000 | 5588.802260 | 999999000000 | 499999500000 |

**Tabela 6 – Gnomesort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000200 | 8436 | 8102 |
| 10000 | 0.002630 | 117623 | 114127 |
| 100000 | 0.033956 | 1509883 | 1474971 |
| 200000 | 0.042776 | 3219627 | 3149996 |
| 300000 | 0.069413 | 5000406 | 4896067 |
| 400000 | 0.100111 | 6839212 | 6699222 |
| 500000 | 0.130451 | 8698333 | 8523703 |
| 600000 | 0.167227 | 10601458 | 10392393 |
| 700000 | 0.203725 | 12532355 | 12287829 |
| 800000 | 0.249984 | 14478935 | 14199590 |
| 900000 | 0.277332 | 16434101 | 16120846 |
| 1000000 | 0.318731 | 18397055 | 18048206 |

**Tabela 7 – Heapsort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000236 | 8813 | 8709 |
| 10000 | 0.002944 | 122288 | 121957 |
| 100000 | 0.024257 | 1556441 | 1550855 |
| 200000 | 0.029632 | 3313748 | 3299161 |
| 300000 | 0.045399 | 5140727 | 5111161 |
| 400000 | 0.061669 | 7031767 | 6983469 |
| 500000 | 0.079243 | 8922789 | 8855701 |
| 600000 | 0.096501 | 10884155 | 10828705 |
| 700000 | 0.116497 | 12878903 | 12818735 |
| 800000 | 0.133143 | 14873643 | 14808701 |
| 900000 | 0.151755 | 16869595 | 16798193 |
| 1000000 | 0.172521 | 18864660 | 18787793 |

**Tabela 8 – Heapsort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000223 | 7991 | 7317 |
| 10000 | 0.002899 | 113360 | 106697 |
| 100000 | 0.030072 | 1463377 | 1397435 |
| 200000 | 0.029441 | 3128277 | 2995729 |
| 300000 | 0.045526 | 4870499 | 4676247 |
| 400000 | 0.062690 | 6659505 | 6410615 |
| 500000 | 0.079769 | 8489282 | 8168451 |
| 600000 | 0.115379 | 10354936 | 9956841 |
| 700000 | 0.115745 | 12234832 | 11759605 |
| 800000 | 0.135163 | 14132618 | 13584465 |
| 900000 | 0.150760 | 16069055 | 15458795 |
| 1000000 | 0.168636 | 18001491 | 17333409 |

**Tabela 9 – Heapsort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000510 | 999 | 249185 |
| 10000 | 0.050798 | 9999 | 24975864 |
| 100000 | 6.067503 | 99999 | 2503447952 |
| 200000 | 24.411696 | 199999 | 9972990404 |
| 300000 | 55.154221 | 299999 | 22450657498 |
| 400000 | 98.553641 | 399999 | 40027774620 |
| 500000 | 154.346585 | 499999 | 62394784027 |
| 600000 | 257.821471 | 599999 | 90007121038 |
| 700000 | 306.128545 | 699999 | 122787869370 |
| 800000 | 396.245174 | 799999 | 159986677534 |
| 900000 | 552.124342 | 899999 | 202396643276 |
| 1000000 | 600.033389 | 999999 | 250053898107 |

**Tabela 10 – Insertionsort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000016 | 999 | 0 |
| 10000 | 0.000094 | 9999 | 0 |
| 100000 | 0.000235 | 99999 | 0 |
| 200000 | 0.000492 | 199999 | 0 |
| 300000 | 0.000677 | 299999 | 0 |
| 400000 | 0.000958 | 399999 | 0 |
| 500000 | 0.001218 | 499999 | 0 |
| 600000 | 0.001588 | 599999 | 0 |
| 700000 | 0.001750 | 699999 | 0 |
| 800000 | 0.001960 | 799999 | 0 |
| 900000 | 0.002236 | 899999 | 0 |
| 1000000 | 0.002507 | 999999 | 0 |

**Tabela 11 – Insertionsort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.003530 | 999 | 499500 |
| 10000 | 0.096134 | 9999 | 49995000 |
| 100000 | 9.011488 | 99999 | 4999950000 |
| 200000 | 53.639241 | 199999 | 19999900000 |
| 300000 | 141.065438 | 299999 | 44999850000 |
| 400000 | 166.225205 | 399999 | 79999800000 |
| 500000 | 258.234492 | 499999 | 124999750000 |
| 600000 | 366.967374 | 599999 | 179999700000 |
| 700000 | 906.812994 | 699999 | 244999650000 |
| 800000 | 938.283282 | 799999 | 319999600000 |
| 900000 | 991.555619 | 899999 | 404999550000 |
| 1000000 | 1942.532041 | 999999 | 499999500000 |

**Tabela 12 – Insertionsort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000080 | 8711 | 4318 |
| 10000 | 0.001012 | 120537 | 59244 |
| 100000 | 0.011782 | 1536468 | 759889 |
| 200000 | 0.025045 | 3272881 | 1620005 |
| 300000 | 0.040071 | 5085014 | 2511101 |
| 400000 | 0.052426 | 6945514 | 3439798 |
| 500000 | 0.066677 | 8837110 | 4390212 |
| 600000 | 0.080510 | 10769183 | 5320898 |
| 700000 | 0.095935 | 12724655 | 6275876 |
| 800000 | 0.109991 | 14691344 | 7279485 |
| 900000 | 0.125090 | 16678608 | 8256186 |
| 1000000 | 0.143499 | 18675006 | 9278929 |

**Tabela 13 – Mergensort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000059 | 5044 | 0 |
| 10000 | 0.000575 | 69008 | 0 |
| 100000 | 0.006636 | 853904 | 0 |
| 200000 | 0.014012 | 1807808 | 0 |
| 300000 | 0.021336 | 2797264 | 0 |
| 400000 | 0.028990 | 3815616 | 0 |
| 500000 | 0.037257 | 4783216 | 0 |
| 600000 | 0.044394 | 5894528 | 0 |
| 700000 | 0.053678 | 7000336 | 0 |
| 800000 | 0.061870 | 8031232 | 0 |
| 900000 | 0.070271 | 9084240 | 0 |
| 1000000 | 0.077618 | 10066432 | 0 |

**Tabela 14 – Mergesort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000048 | 4932 | 4932 |
| 10000 | 0.000589 | 64608 | 64608 |
| 100000 | 0.006806 | 815024 | 815024 |
| 200000 | 0.033115 | 1730048 | 1730048 |
| 300000 | 0.021871 | 2678448 | 2678448 |
| 400000 | 0.029894 | 3660096 | 3660096 |
| 500000 | 0.038579 | 4692496 | 4692496 |
| 600000 | 0.047065 | 5656896 | 5656896 |
| 700000 | 0.055464 | 6651088 | 6651088 |
| 800000 | 0.063807 | 7720192 | 7720192 |
| 900000 | 0.072123 | 8767184 | 8767184 |
| 1000000 | 0.079721 | 9884992 | 9884992 |

**Tabela 15 – Mergesort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000232 | 10677 | 7336 |
| 10000 | 0.002041 | 149364 | 83257 |
| 100000 | 0.015794 | 2079900 | 1217010 |
| 200000 | 0.040811 | 4254051 | 2359739 |
| 300000 | 0.059895 | 7067680 | 3844969 |
| 400000 | 0.075544 | 9101331 | 4980831 |
| 500000 | 0.106553 | 11776390 | 6243400 |
| 600000 | 0.145120 | 15101888 | 8285236 |
| 700000 | 0.153213 | 16565377 | 8886213 |
| 800000 | 0.192934 | 19728710 | 10399255 |
| 900000 | 0.209007 | 21928147 | 11270308 |
| 1000000 | 0.240464 | 23986050 | 13039846 |

**Tabela 16 – Quicksort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.005046 | 499500 | 500499 |
| 10000 | 0.493096 | 49995000 | 50004999 |
| 100000 | 51.733237 | 4999950000 | 5000049999 |

**Tabela 17 – Quicksort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.005001 | 499500 | 250499 |
| 10000 | 0.331269 | 49995000 | 25004999 |
| 100000 | 31.296026 | 4999950000 | 2500049999 |
| 20000 | 46.070497 | 7366469190 | 3683360147 |

**Tabela 18 – Quicksort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.002785 | 499500 | 991 |
| 10000 | 0.114248 | 49995000 | 9992 |
| 100000 | 11.261241 | 4999950000 | 99990 |
| 200000 | 45.037807 | 19999900000 | 199984 |
| 300000 | 101.247408 | 44999850000 | 299988 |
| 400000 | 179.941800 | 79999800000 | 399980 |
| 500000 | 281.443916 | 124999750000 | 499987 |
| 600000 | 406.063803 | 179999700000 | 599995 |
| 700000 | 554.224734 | 244999650000 | 699981 |
| 800000 | 725.953637 | 319999600000 | 799993 |
| 900000 | 922.778084 | 404999550000 | 899980 |
| 1000000 | 1142.254526 | 499999500000 | 999981 |

**Tabela 19 – Selectionsort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.002846 | 499500 | 0 |
| 10000 | 0.113849 | 49995000 | 0 |
| 100000 | 11.256547 | 4999950000 | 0 |
| 200000 | 45.009932 | 19999900000 | 0 |
| 300000 | 101.462775 | 44999850000 | 0 |
| 400000 | 180.911438 | 79999800000 | 0 |
| 500000 | 282.873749 | 124999750000 | 0 |
| 600000 | 409.004904 | 179999700000 | 0 |
| 700000 | 557.167017 | 244999650000 | 0 |
| 800000 | 727.749909 | 319999600000 | 0 |
| 900000 | 918.501047 | 404999550000 | 0 |
| 1000000 | 1135.334070 | 499999500000 | 0 |

**Tabela 20 – Selectionsort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.001516 | 499500 | 500 |
| 10000 | 0.135242 | 49995000 | 5000 |
| 100000 | 11.587708 | 4999950000 | 50000 |
| 200000 | 46.394593 | 19999900000 | 100000 |
| 300000 | 104.351848 | 44999850000 | 150000 |
| 400000 | 186.114453 | 79999800000 | 200000 |
| 500000 | 293.186715 | 124999750000 | 250000 |
| 600000 | 419.122492 | 179999700000 | 300000 |
| 700000 | 573.336565 | 244999650000 | 350000 |
| 800000 | 751.208892 | 319999600000 | 400000 |
| 900000 | 951.649183 | 404999550000 | 450000 |
| 1000000 | 1171.327412 | 499999500000 | 500000 |

**Tabela 21 – Selectionsort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000281 | 8006 | 6958 |
| 10000 | 0.004266 | 120005 | 155236 |
| 100000 | 0.042395 | 1500006 | 2792208 |
| 200000 | 0.075192 | 3200006 | 6797940 |
| 300000 | 0.103793 | 5100008 | 10661386 |
| 400000 | 0.164657 | 6800006 | 17121153 |
| 500000 | 0.207145 | 8500007 | 19950705 |
| 600000 | 0.259706 | 10800008 | 24841577 |
| 700000 | 0.265710 | 12600009 | 31646869 |
| 800000 | 0.373621 | 14400006 | 38853061 |
| 900000 | 0.393763 | 16200011 | 40634531 |
| 1000000 | 0.422650 | 18000007 | 46859906 |

**Tabela 22 – Shellsort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000077 | 8006 | 0 |
| 10000 | 0.001130 | 120005 | 0 |
| 100000 | 0.014297 | 1500006 | 0 |
| 200000 | 0.030285 | 3200006 | 0 |
| 300000 | 0.030838 | 5100008 | 0 |
| 400000 | 0.046829 | 6800006 | 0 |
| 500000 | 0.059074 | 8500007 | 0 |
| 600000 | 0.074330 | 10800008 | 0 |
| 700000 | 0.088945 | 12600009 | 0 |
| 800000 | 0.101213 | 14400006 | 0 |
| 900000 | 0.113399 | 16200011 | 0 |
| 1000000 | 0.123279 | 18000007 | 0 |

**Tabela 23 – Shellsort, crescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000084 | 8006 | 4700 |
| 10000 | 0.001138 | 120005 | 62560 |
| 100000 | 0.014434 | 1500006 | 844560 |
| 200000 | 0.026136 | 3200006 | 1789120 |
| 300000 | 0.043615 | 5100008 | 2500880 |
| 400000 | 0.052759 | 6800006 | 3778240 |
| 500000 | 0.070801 | 8500007 | 4428752 |
| 600000 | 0.090154 | 10800008 | 5301760 |
| 700000 | 0.106377 | 12600009 | 6413904 |
| 800000 | 0.124701 | 14400006 | 7956480 |
| 900000 | 0.139076 | 16200011 | 8207632 |
| 1000000 | 0.154852 | 18000007 | 9357504 |

**Tabela 24 – Shellsort, decrescente**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000210 | 5831 | 9541 |
| 10000 | 0.002941 | 94677 | 111653 |
| 100000 | 0.026297 | 1276955 | 1287282 |
| 200000 | 0.035982 | 2753670 | 2673149 |
| 300000 | 0.054198 | 4363683 | 4093668 |
| 400000 | 0.073122 | 5907449 | 5543342 |
| 500000 | 0.102139 | 7409614 | 7051012 |
| 600000 | 0.124170 | 9327568 | 8494572 |
| 700000 | 0.127660 | 10850869 | 9950926 |
| 800000 | 0.163255 | 12614629 | 11494006 |
| 900000 | 0.183983 | 14125065 | 13007155 |
| 1000000 | 0.187233 | 15819188 | 14600667 |

**Tabela 25 – Timsort, aleatório**

| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000090 | 3497 | 0 |
| 10000 | 0.001367 | 56095 | 0 |
| 100000 | 0.012788 | 721718 | 0 |
| 200000 | 0.025227 | 1543436 | 0 |
| 300000 | 0.030268 | 2464898 | 0 |
| 400000 | 0.035690 | 3286872 | 0 |
| 500000 | 0.045810 | 4016526 | 0 |
| 600000 | 0.057688 | 5229796 | 0 |
| 700000 | 0.069349 | 6019034 | 0 |
| 800000 | 0.080581 | 6973744 | 0 |
| 900000 | 0.089822 | 7696742 | 0 |
| 1000000 | 0.106465 | 8533052 | 0 |

**Tabela 26 – Timsort, crescente**

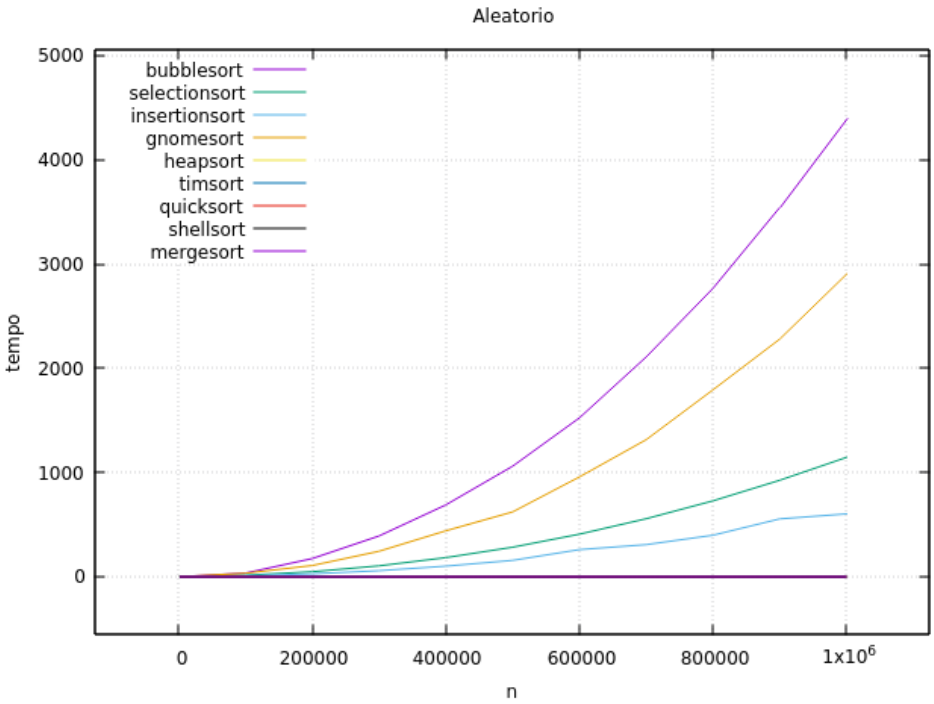
| n | Tempo(seg) | Comparações | Trocas |
|---|---|---|---|
| 1000 | 0.000194 | 3377 | 16883 |
| 10000 | 0.002454 | 48991 | 184816 |
| 100000 | 0.029697 | 658774 | 2018149 |
| 200001 | 0.024875 | 1417554 | 4136304 |
| 300000 | 0.046530 | 2209698 | 6287823 |
| 400000 | 0.057458 | 3035096 | 8472596 |
| 500000 | 0.065329 | 3911246 | 10708121 |
| 600000 | 0.081208 | 4719396 | 12875646 |
| 700000 | 0.112376 | 5557338 | 15072963 |
| 800000 | 0.138323 | 6470192 | 17345192 |
| 900000 | 0.135368 | 7360934 | 19595309 |
| 1000000 | 0.167905 | 8322492 | 21916242 |

**Tabela 27 – Timsort, decrescente**

## 4.2 Gráficos

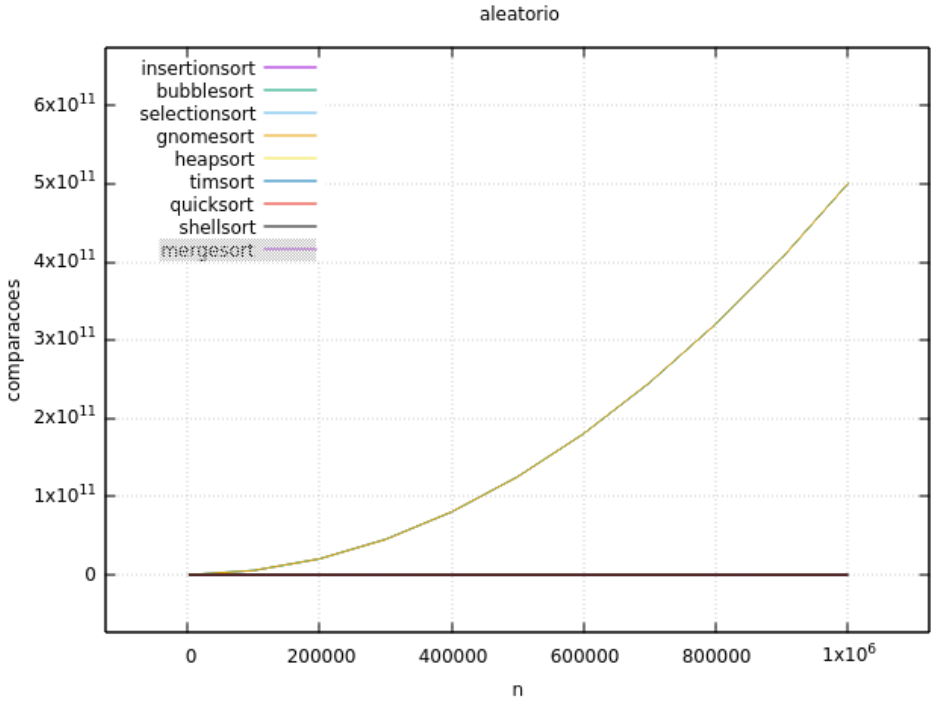
A partir das tabelas anteriores, foram gerados os gráficos a seguir com o objetivo de comparar o comportamento dos algoritmos de forma visual. Desse modo, em cada gráfico, há o comportamento de todos os algoritmo para as devidas instâncias, que estão especificadas em cada gráfico. Deve-se deixar claro que, por conta das grandes diferenças de execução de algoritmos eficientes e não eficiente, alguns gráficos dão a impressão que certos algoritmos tiveram um comportamento costante, ficando apenas em 0 (zero).

**Figura 1 – Aleatório, N x Tempo**



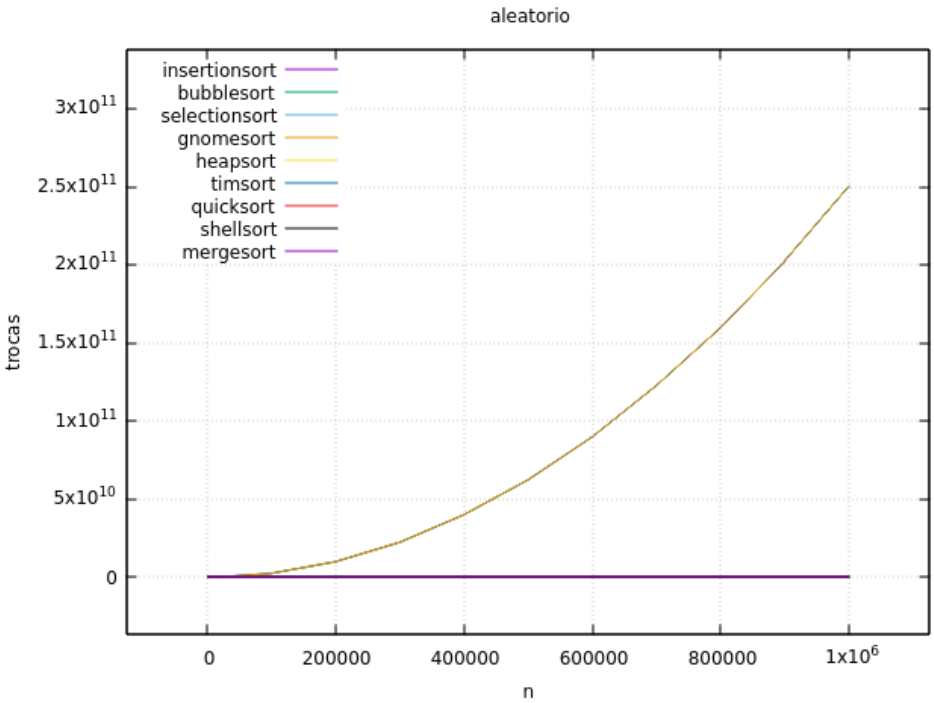

Na Figura 1, os algoritmos que mais custaram para terminar de executar as grandes instâncias foram o bubble, insertion, selection e gnome. Os demais não levaram um pouco menos de 1 segundo de tempo, como se pode ver no gráfico.

**Figura 2 – Aleatório, N x Comparações**



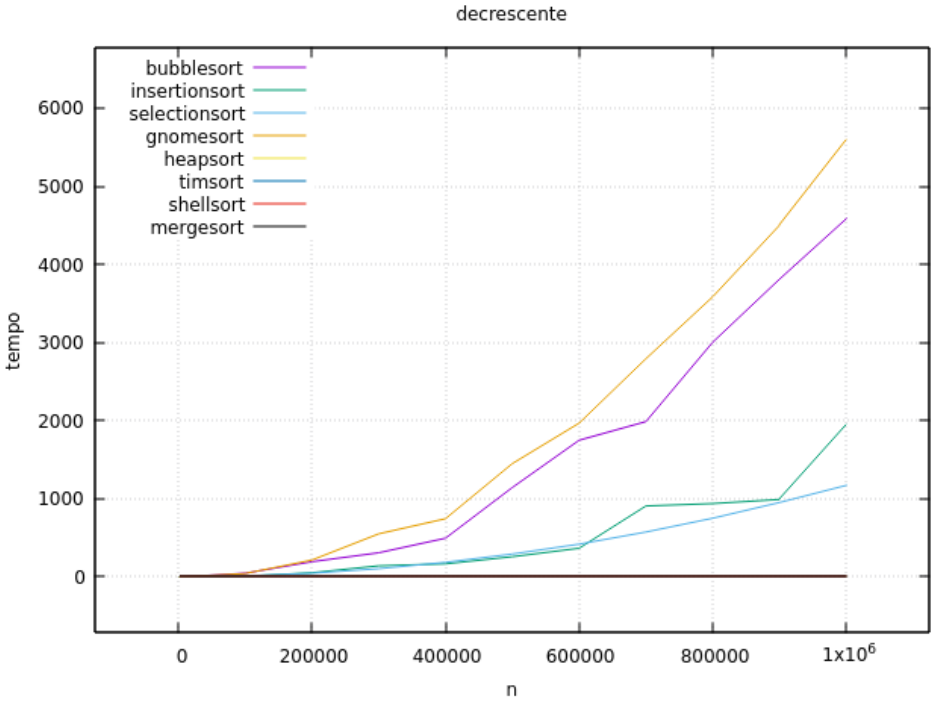

Na figura 2, os algoritmos que mais realizaram comparações para processar um vetor disposto de forma aleatório foram o bubble, selection e gnome sort. Quanto aos demais algoritmos, eles tiveram uma quantidade significativa de comparações, mas não tanto como os três algoritmos citados.

**Figura 3 – Aleatório, N x Trocas**




Como mostra a Figura 3, os algoritmos que mais realizaram trocas para ordenar um vertor com os elementos desordenados aleatoriamente foram o insertion e bubble sortl. Porém, diferente do que o gráfico mostra, os demais algoritmos tiveram uma grande quantidade de trocas, no processamento, mas não tanta como os dois algoritmos citados.

**Figura 4 – Decrescente, N x Tempo**




Na Figura 4, os algorimos que levaram mais tempo para executar entradas maiores foram o bubblesort, insertionsort, selectionsort e gnomesort. Os demais algoritmos não levaram nem sequer 1 segundo para executar instâncias de $1 \times 10^6$.

**Figura 5 – Decrescente, N x Comparações**

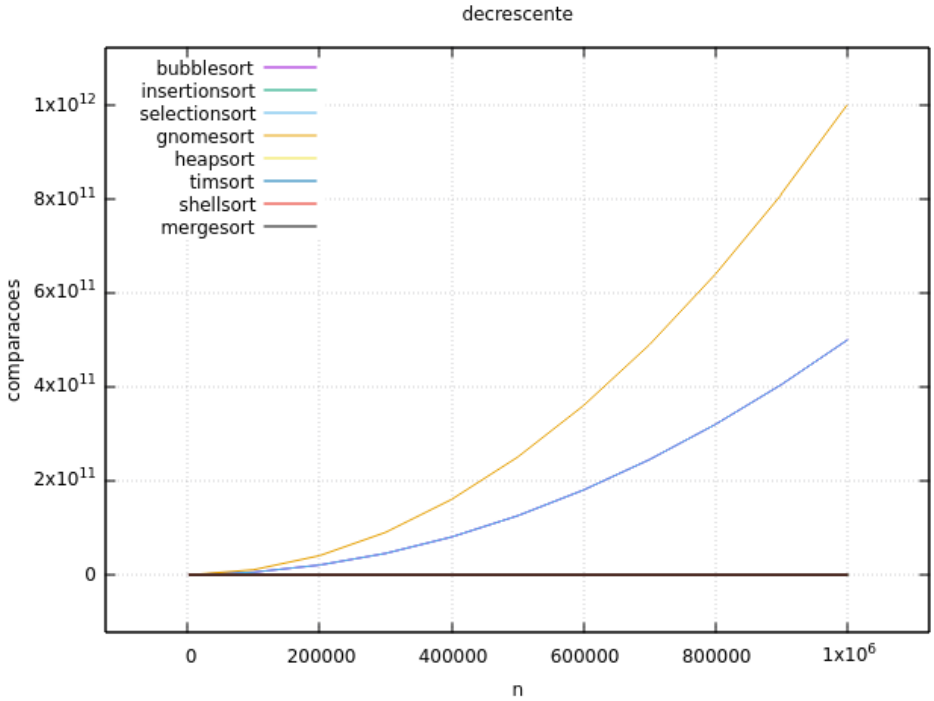
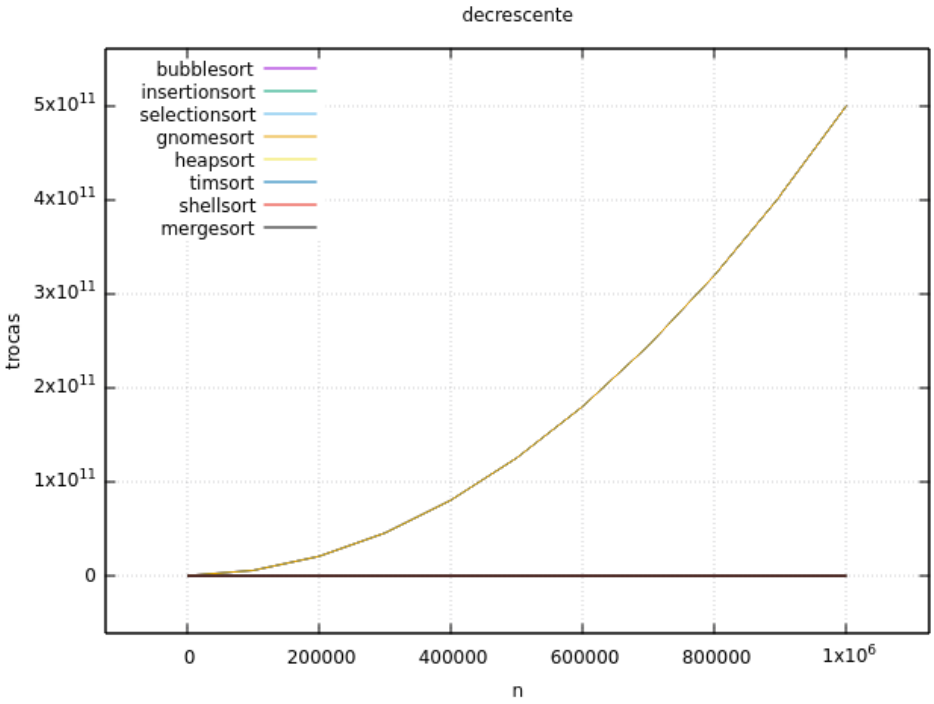
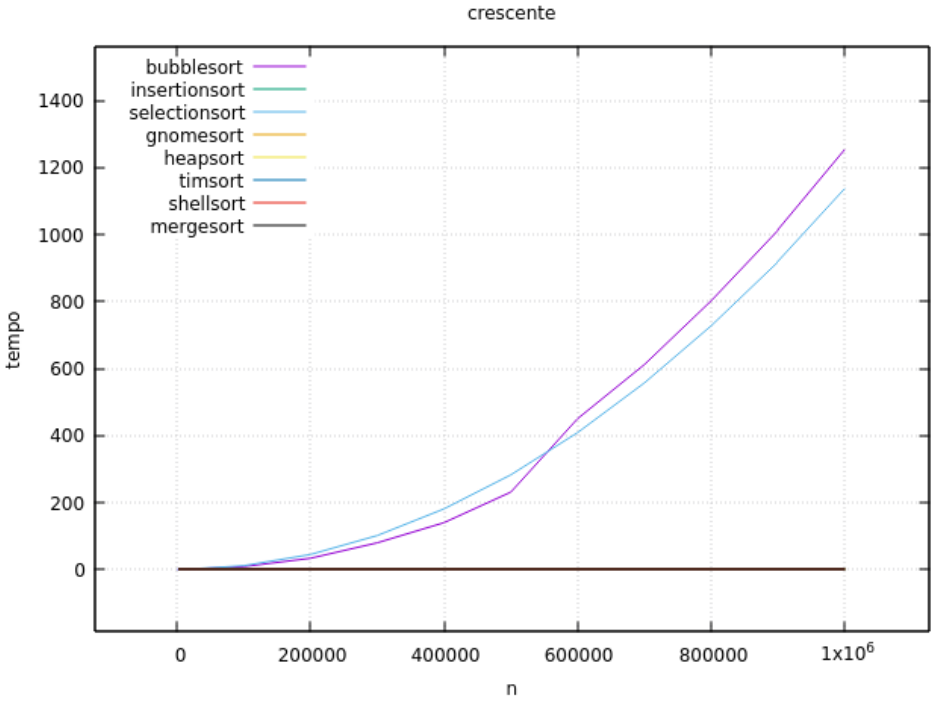


**Figura 6 – Decrescente, N x Trocas**



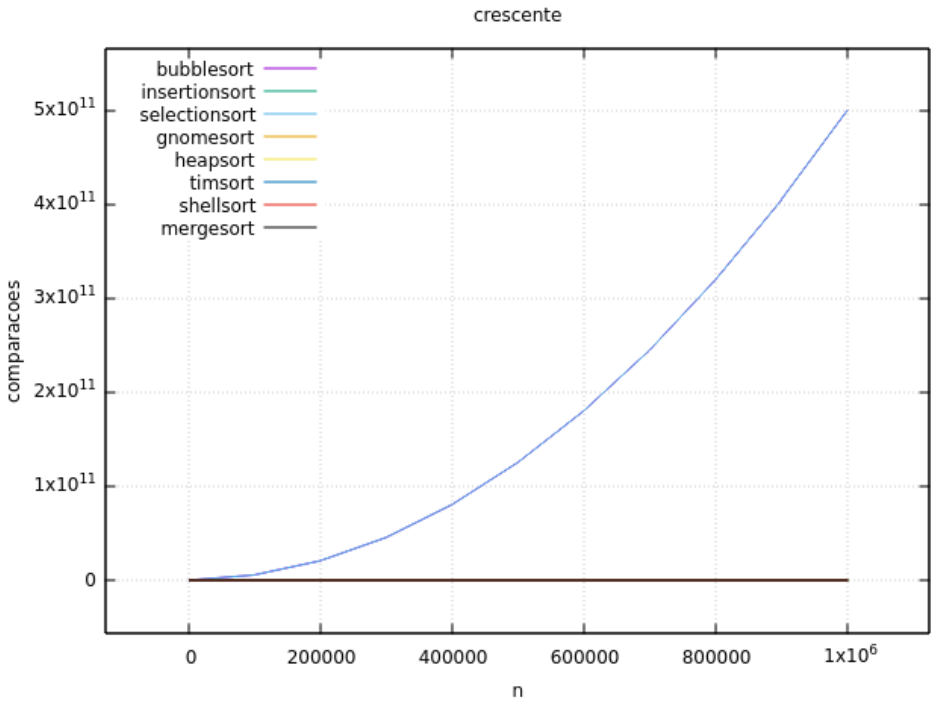

Como é possível ver na Figura 6, os algoritmos que realizaram mais trocas foram o bubble, insertion e gnome. Os demais tiveram um quantidade de trocas significativas, mas, comparado aos anteriores, eles realizaram bem poucas, quase linear à quantide de elementos.

**Figura 7 – Crescente, N x Tempo**




Na Figura 7, apesar do vetor já está ordenado, alguns algoritmos levaram um tempo considerável só para verificar o estado do vetor, como foram os casos do bubblesort e do selectionsort. Os demais algorimos não levaram nem 1 segundo para processar vetores ordenados de $1 \times 10^6$. Isso se dá ao fato que tanto o bubble quanto o selection sort verificam todos os elementos do vetor para cada elemento, na busca de colocá-lo na lugar correto.

**Figura 8 – Crescente, N x Comparações**




Já na Figura 8, os algorimtos que mais realizaram comparações em um vetor ordenado foram o bubble e selection sort. Novamente, como foi dito, esses algorimtos verificam todos os elementos do vetor para cada elemento, na busca de colocá-lo na lugar correto.

### 4.3 Considerações

A partir da análise dos gráficos, pode-se perceber que em todos eles alguns algoritmos mantiveram seus comportamentos parecidos, independente da forma que os elementos estão dispostos na instância passada, isto é, aleatório, crescente ou decrescente. Como exemplo, nos três gráficos que envolvem o tempo, algoritmos como o Bubble, Insertion, Selection e Gnome sort foram os algoritmos que mais custaram para processar as instâncias, conforme a quantidade de elementos delas aumentava. Por outro lado, os demais levaram pouco menos de 1 segundo. Como consequência disso, algoritmos como o Tim, Heap e Merge são mais eficientes quanto ao tempo de processamento. O que já era de se esperar, visto que as complexidades de execução deles são melhores em comparação aos algoritmos antes citados.

Quanto à quantidade de comparações que os algoritmos realizaram para processar as devidas instâncias, pode ser visto nos três gráficos que envolvem *comparacoes* que, de modo geral, os piores algoritmos foram o Selection, Gnome e Heap sort, sendo que os demais algoritmos realizaram tantas comparações quanto a quantidade de elementos da instância. Algoritmos que realizam poucas comparações são mais desejáveis para aplicações, visto que realizar essa tarefa exige trabalho da unidade de processamento da máquina. Desse modo, algoritmos como o Merge, Tim e Shell sort são mais almejados para aplicações que é voltada para manipular grandes quantidades de registros.

Por último, não menos importante, tem-se as considerações sobre a quantidade de trocas necessárias para ordenar as instâncias passadas para os algoritmos. Como pode-se notar nos gráficos de *trocas*, os algoritmos Gnome e Heap sort realizaram uma quantidade absurda de trocas para instâncias de $1 \times 10^6$ elementos. Assim como em quantidade de comparações, realizar poucas trocas é necessário para aplicações que envolvem muitos registros, visto que consome um certa quantidade de trabalho da unidade de processamento. Por esse motivo, para aplicações que buscam poupar trabalho desnecessário de CPU, algoritmos mais estratégicos são necessários, como o Shell, Merge e Tim sort, por exemplo.

# REFERÊNCIAS

BACKES, A. **Algoritmos e Estruturas de Dados em Linguagem C**. 1st. ed. [S.l.]: LTC, 2022. ISBN 8521638302.

CORMEN, T. H. et al. **Introduction to Algorithms**. 4th. ed. [S.l.]: The MIT Press, 2022. ISBN 9780262046305.

GEEKS, G. F. **Sorting Algorithms**. 2023. Disponível em: <https://www.geeksforgeeks.org/sorting-algorithms>.