

Estruturas de Dados Avançadas

Erik Zerbinatti
Guilherme Ventura
Richard Nascimento

Campinas
2015

Índice

[Algoritmos](#)

[Selection Sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Bubble Sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Quick Sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Gnome sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Cocktail Sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Comb Sort](#)

[Classificação](#)

[Complexidade](#)

[Funcionamento](#)

[Algoritmo \(em C\)](#)

[Comparação](#)

[Bibliografia](#)

Algoritmos

Selection Sort

Classificação

Ordenação por Seleção

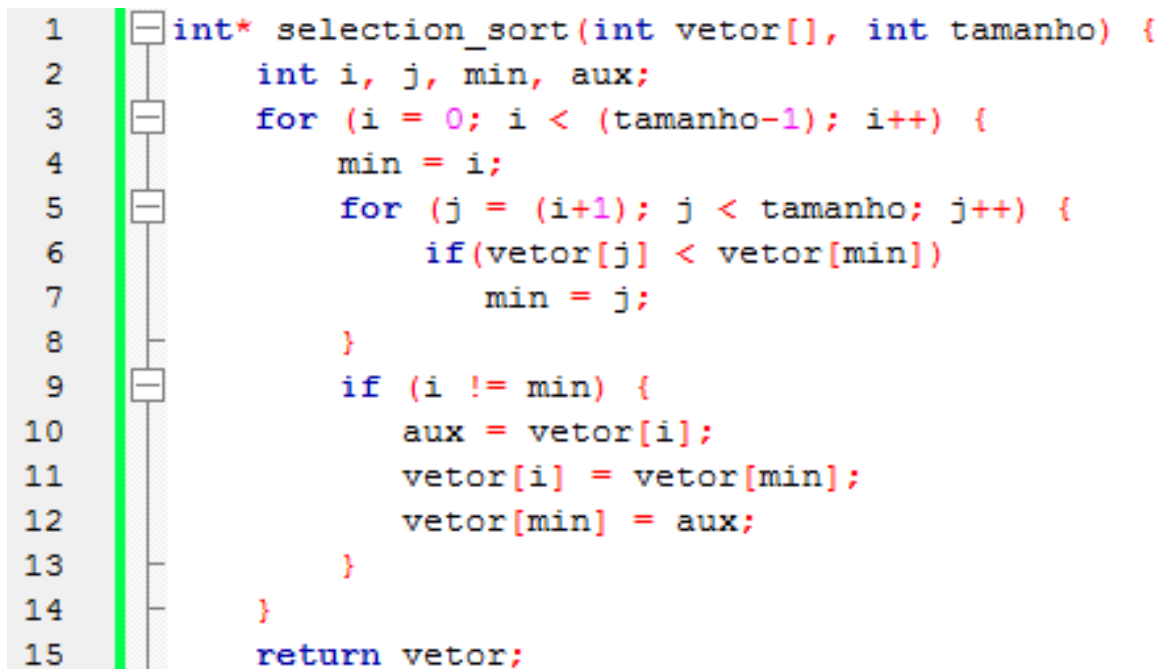
Complexidade

Simples

Funcionamento

Seleciona o elemento da primeira posição do vetor e compara com os próximos, da esquerda para a direita, trocando-os se necessário. Ao término das comparações, seleciona a posição seguinte e compara o elemento com os posteriores. Essa lógica é repetida até que o elemento da ultima posição esteja ordenado.

Algoritmo (em C)



```
1 int* selection_sort(int vetor[], int tamanho) {
2     int i, j, min, aux;
3     for (i = 0; i < (tamanho-1); i++) {
4         min = i;
5         for (j = (i+1); j < tamanho; j++) {
6             if(vetor[j] < vetor[min])
7                 min = j;
8         }
9         if (i != min) {
10            aux = vetor[i];
11            vetor[i] = vetor[min];
12            vetor[min] = aux;
13        }
14    }
15    return vetor;
```

1. Vetor inicial:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

2. O algoritmo seleciona o primeiro elemento (elemento principal) e o compara com o próximo:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

3. Caso o elemento principal seja menor que o elemento comparado, o algoritmo compara com o próximo:

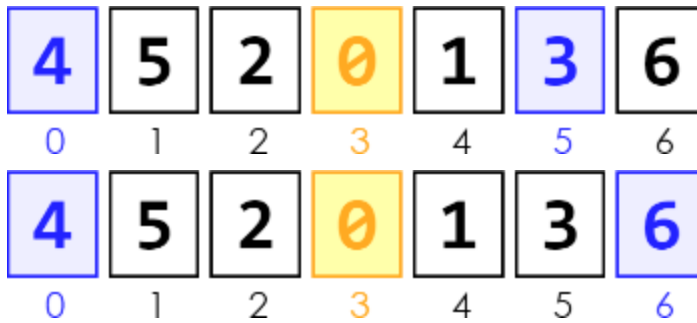
4	5	2	0	1	3	6
0	1	2	3	4	5	6

4. Ao encontrar um elemento menor que o principal, ou seja, na posição incorreta, este é marcado e o algoritmo continua a comparação do elemento com os demais :

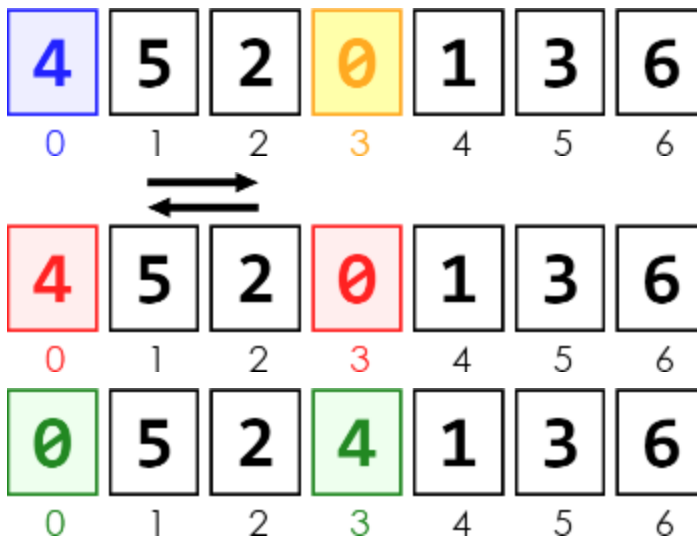
4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	5	2	0	1	3	6
0	1	2	3	4	5	6

5. Ao encontrar um elemento menor que o elemento marcado, este é marcado e o anterior desmarcado e novamente a comparação é feita com os demais, trocando as marcações quando necessário:

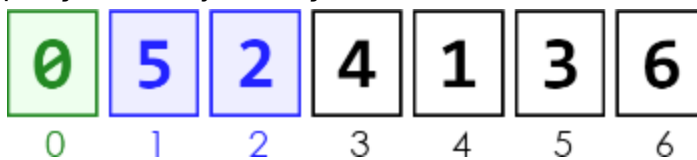
4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	5	2	0	1	3	6
0	1	2	3	4	5	6



6. Ao término do vetor, o elemento marcado troca de lugar com o elemento principal:



7. Após a troca, o elemento principal está ordenado, o valor da próxima posição é selecionado como elemento principal e a lógica se repete até que o elemento na posição N-1 seja alcançado:



Bubble Sort

Classificação

Ordenação por Permutação

Complexidade

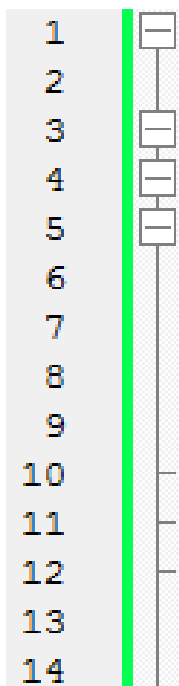
Simples

Funcionamento

Compara o elemento da primeira posição do vetor com o elemento da posição seguinte, trocando-os se necessário, e avançando uma posição, até o fim. Essa lógica é repetida até que não haja troca de posições.

Um dos pontos negativos do algoritmo é a demora para ordenar elementos de menor valor localizados no final do vetor, os chamados *elementos tartaruga*. Pela natureza do algoritmo, esses elementos necessitam de mais iterações para chegar a posição correta. Por outro lado, elementos com valores altos localizados no início do vetor são ordenados rapidamente. Estes são chamados de *elementos coelho*.

Algoritmo (em C)



```
1  int* bubble_sort(int vetor[], int tamanho) {
2      int i, j, aux = 0;
3      for(i = tamanho-1; i > 0; i--) {
4          for(j = 0; j < i; j++) {
5              if(vetor[j] > vetor[j+1]) {
6                  int temp;
7                  temp = vetor[j];
8                  vetor[j] = vetor[j+1];
9                  vetor[j+1] = temp;
10             }
11         }
12     }
13     return vetor;
14 }
```

1. Vetor inicial:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

2. O algoritmo seleciona o elemento da primeira posição (elemento principal) e compara com o próximo:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

3. Caso o elemento principal seja menor que o elemento comparado, o mesmo avança uma posição e o algoritmo compara novamente com o próximo:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

4. Ao encontrar um elemento menor que o principal, ou seja, na posição incorreta, estes são trocados:

4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	2	5	0	1	3	6
0	1	2	3	4	5	6

5. Após a troca, o elemento principal (já na nova posição) é comparado com o próximo novamente. Essa lógica é repetida até a comparação chegar ao fim do vetor:

4	2	5	0	1	3	6
0	1	2	3	4	5	6
4	2	0	5	1	3	6
0	1	2	3	4	5	6

4	2	0	5	1	3	6
0	1	2	3	4	5	6
4	2	0	1	5	3	6
0	1	2	3	4	5	6
4	2	0	1	5	3	6
0	1	2	3	4	5	6
4	2	0	1	3	5	6
0	1	2	3	4	5	6
4	2	0	1	3	5	6
0	1	2	3	4	5	6
4	2	0	1	3	5	6
0	1	2	3	4	5	6

6. Após percorrer o vetor até o fim, o algoritmo garante que a última posição já está ordenada. O elemento principal volta a ser o item na primeira posição e o Bubble Sort é aplicado até que o vetor esteja totalmente ordenado.

4	2	0	1	3	5	6
0	1	2	3	4	5	6

Quick Sort

Classificação

Ordenação por Permutação

Complexidade

Relativamente Complexo

Funcionamento

Utilizando um elemento como guia (denominado pivô), o algoritmo separa os elementos em dois vetores, os menores e os maiores que o pivô. Para os dois vetores criados, a lógica é aplicada repetidamente, até que os itens estejam ordenados.

Geralmente, o elemento central é selecionado como pivô, mas não é uma regra: existem situações onde selecionar o item de um dos cantos é a melhor opção, como quando se sabe que o vetor está parcialmente ordenado.

Algoritmo (em C)

```
1  int* quick_sort(int vetor[],int esq, int dir) {
2      int i, j, pivô, aux;
3      i = esq;
4      j = dir;
5      pivô = vetor[(i+j)/2];
6      do {
7          while(vetor[i] < pivô && i < dir)
8              i++;
9          while(vetor[j] > pivô && j > esq)
10             j--;
11         if(i <= j) {
12             aux = vetor[i];
13             vetor[i] = vetor[j];
14             vetor[j] = aux;
15             i++;
16             j--;
17         }
18     }while (i <= j);
19     if(esq < j) quick_sort(vetor, esq, j);
20     if(dir > i) quick_sort(vetor, i, dir);
21     return vetor;
22 }
```

1. Vetor inicial:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

2. Um elemento do vetor é escolhido como pivô (nesse caso, o elemento central):

4	5	2	0	1	3	6
0	1	2	3	4	5	6

3. O algoritmo compara este pivô com todos os elementos restantes do vetor e os separa em dois vetores. Os maiores que o pivô ficam a direita e os menores a esquerda. Nesse caso, o valor do pivô é 0, então todos os valores ficam no vetor dos maiores:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

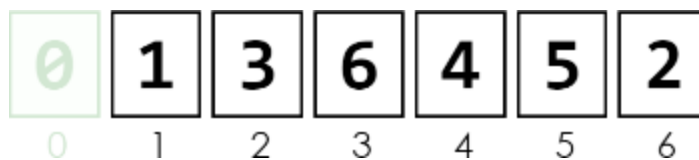
5	2	0	1	3	6	4
0	1	2	3	4	5	6

5	2	0	1	3	6	4
0	1	2	3	4	5	6

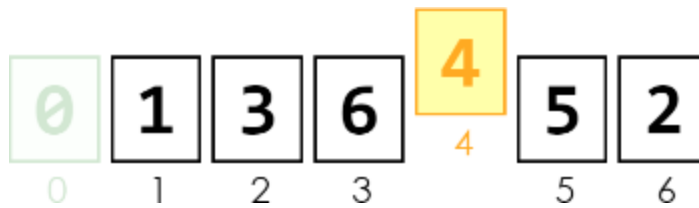
2	0	1	3	6	4	5
0	1	2	3	4	5	6



4. Nesse momento, o algoritmo garante que o elemento pivô já está ordenado. Agora temos o vetor dos maiores:



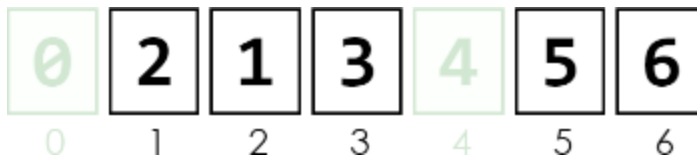
5. Novamente, um elemento do vetor é escolhido como pivô.



6. A comparação do novo pivô é feita com os elementos restantes e novamente os separa em 2 vetores:



7. Novamente, o elemento pivô está garantidamente ordenado. Então, temos dois vetores, os menores e os maiores que o mesmo. A lógica é repetida, em cada vetor, até que o vetor principal esteja completamente ordenado:



Gnome sort

Classificação

Ordenação por Permutação

Complexidade

Relativamente Simples

Funcionamento

O algoritmo denominado Gnome Sort é bem parecido com o Bubble Sort. Ele também percorre o vetor todo da esquerda para a direita, comparando os elementos em pares e trocando os elementos que estão nas posições incorretas. A diferença desse algoritmo é o fato dele não percorrer esse vetor várias vezes. Enquanto não há elementos a serem trocados, o algoritmo continua comparando normalmente. Ao encontrar um item na posição incorreta (o elemento da esquerda é maior que o da direita, por exemplo), ele começa a comparar no modo reverso, da direita para a esquerda, até colocar o elemento capturado em seu respectivo lugar.

Algoritmo (em C)

```
1  int* gnome_sort(int vetor[], int tamanho) {
2      int next = 0, previous = 0;
3      int i = 0;
4      for(i = 0; i < tamanho; i++) {
5          if(vetor[i] > vetor[i+1]) {
6              previous = i;
7              next = i+1;
8              while(vetor[previous] > vetor[next]) {
9                  swap(&vetor[previous], &vetor[next]);
10                 if(previous > 0)
11                     previous--;
12                 if(next > 0)
13                     next--;
14             }
15         }
16     }
17     return vetor;
18 }
19 void swap(int *A, int *B) {
20     int C = *A;
21     *A = *B;
22     *B = C;
23 }
```

1. Vetor inicial

4	5	2	0	1	3	6
0	1	2	3	4	5	6

2. O algoritmo percorre o vetor comparando os elementos de dois em dois:

4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	5	2	0	1	3	6
0	1	2	3	4	5	6

3. Ao encontrar um elemento na posição incorreta, troca a posição dos dois elementos:

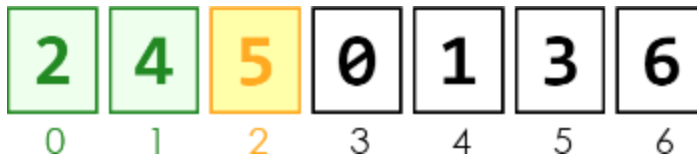
↔						
4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	2	5	0	1	3	6
0	1	2	3	4	5	6

4. Em seguida, volta com este elemento comparando com os anteriores, marcando onde parou:

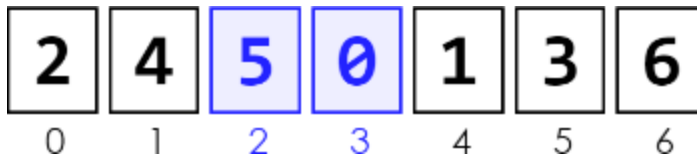
4	2	5	0	1	3	6
0	1	2	3	4	5	6

5. Caso encontre elementos na posição incorreta, a posição dos mesmos é trocada:

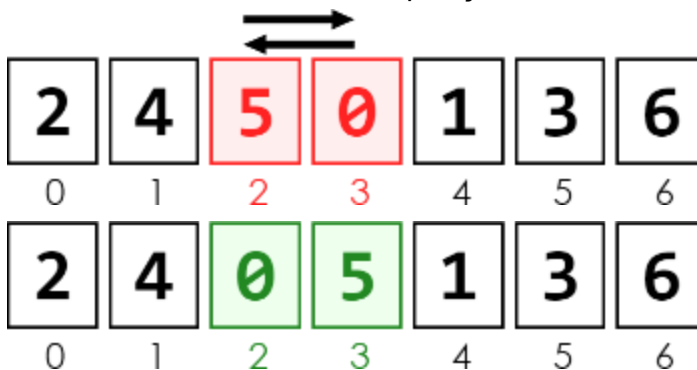
↔						
4	2	5	0	1	3	6
0	1	2	3	4	5	6



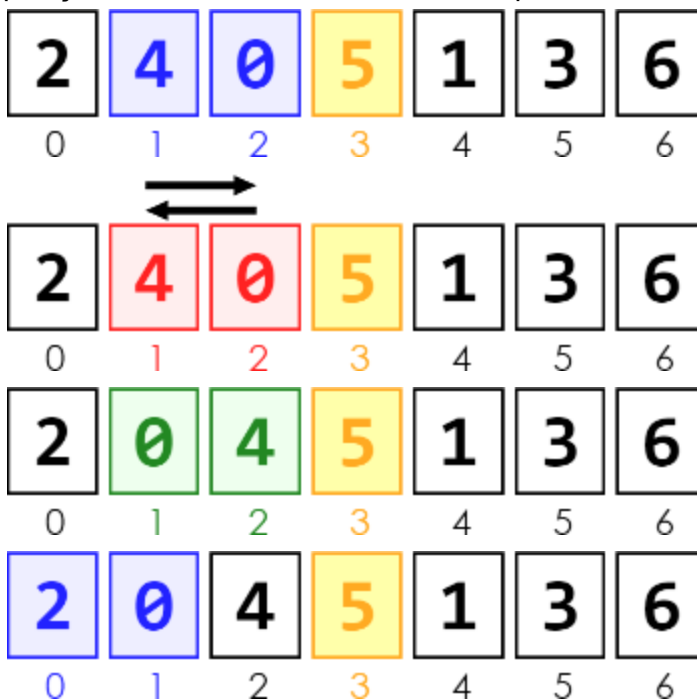
6. Em seguida, a comparação continua de onde parou:

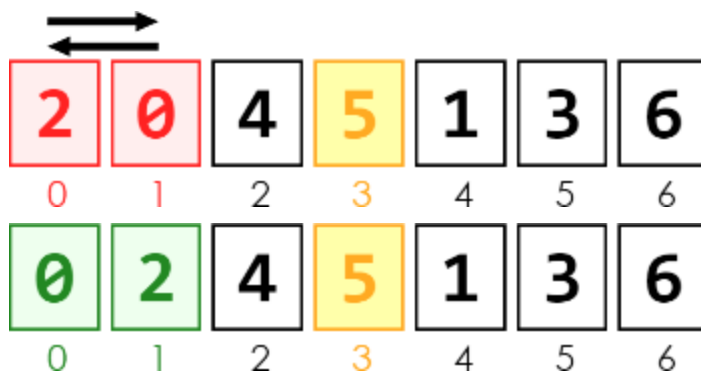


7. Ao encontrar um elemento na posição incorreta, troca a posição dos dois elementos:

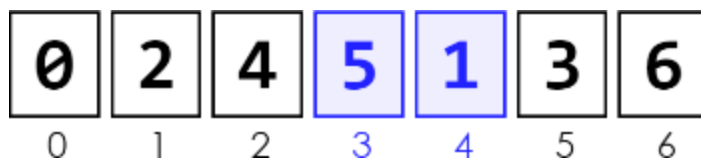


8. Em seguida, volta com este elemento comparando com os anteriores, trocando as posições se necessário e marca onde parou:





9. Ao chegar com o menor elemento até o momento na primeira posição, a comparação volta a ser feita onde estava marcada e essa lógica é repetida até que as comparações em duplas chegue ao fim do vetor:



Cocktail Sort

Classificação

Ordenação por Permutação

Complexidade

Relativamente Complexo

Funcionamento

Também baseado no **Bubble Sort**, a ideia desse algoritmo (também conhecido como **Bubble Sort Bidirecional**) é eliminar os elementos tartaruga mais rapidamente. O Cocktail Sort compara os elementos em pares, da esquerda para a direita, e, ao chegar no final do vetor, volta comparando os elementos da direita para a esquerda.

Algoritmo (em C)

```
1  int* cocktail_sort(int vetor[], int tamanho) {
2      int length, bottom, top, swapped, i, aux;
3      length = tamanho;
4      bottom = 0;
5      top = length-1;
6      swapped = 0;
7      while(swapped == 0 && bottom < top) {
8          swapped = 1;
9          for(i = bottom; i < top; i = i+1) {
10             if(vetor[i] > vetor[i+1]) {
11                 aux = vetor[i];
12                 vetor[i] = vetor[i+1];
13                 vetor[i+1] = aux;
14                 swapped = 0;
15             }
16         }
17         top = top-1;
18         for(i = top; i > bottom; i = i-1) {
19             if(vetor[i] < vetor[i-1]) {
20                 aux = vetor[i];
21                 vetor[i] = vetor[i-1];
22                 vetor[i-1] = aux;
23                 swapped = 0;
24             }
25         }
26         bottom = bottom+1;
27     }
28     return vetor;
29 }
```


1. Vetor inicial:

4	5	2	0	1	3	6
0	1	2	3	4	5	6


2. O algoritmo compara os elementos dois a dois:

4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	5	2	0	1	3	6
0	1	2	3	4	5	6

3. Ao encontrar um elemento desordenado, troca as posições dos elementos:

						
4	5	2	0	1	3	6
0	1	2	3	4	5	6
4	2	5	0	1	3	6
0	1	2	3	4	5	6

4. Em seguida, continua a comparação até o fim do vetor, sempre dois a dois, trocando-os quando necessário:

4	2	5	0	1	3	6
0	1	2	3	4	5	6
						
4	2	5	0	1	3	6
0	1	2	3	4	5	6

4	2	0	5	1	3	6
0	1	2	3	4	5	6

4	2	0	5	1	3	6
0	1	2	3	4	5	6



4	2	0	5	1	3	6
0	1	2	3	4	5	6

4	2	0	1	5	3	6
0	1	2	3	4	5	6

4	2	0	1	5	3	6
0	1	2	3	4	5	6

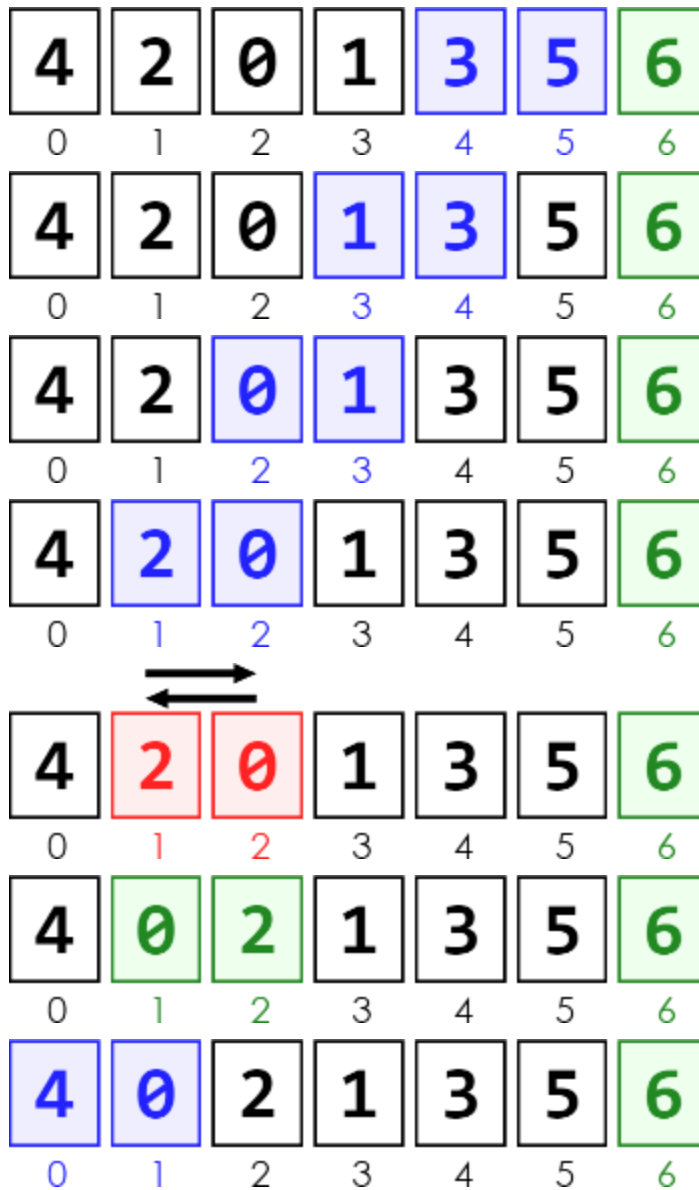


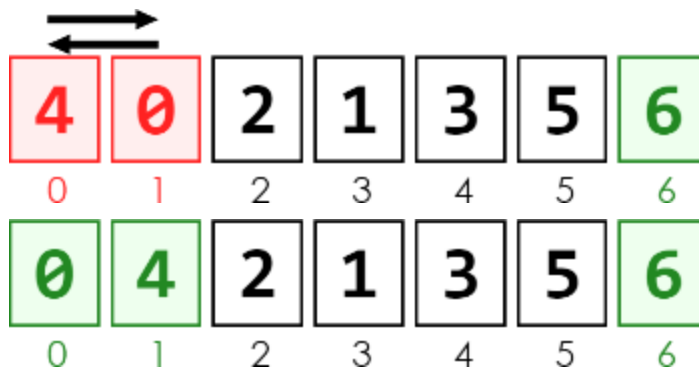
4	2	0	1	5	3	6
0	1	2	3	4	5	6

4	2	0	1	3	5	6
0	1	2	3	4	5	6

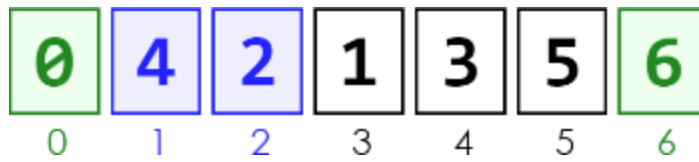
4	2	0	1	3	5	6
0	1	2	3	4	5	6

5. Após comparar os dois últimos elementos do vetor e fazer as trocas necessárias, o algoritmo entende que a última posição está ordenada e volta comparando os elementos da direita para a esquerda, trocando as posições desordenadas até o começo do vetor:





6. Após a chegada no início do vetor, o algoritmo entende que a primeira posição está ordenada e repete a lógica até que o vetor esteja totalmente ordenado:



Comb Sort

Classificação

Ordenação por Permutação

Complexidade

Relativamente Complexo

Funcionamento

O principal objetivo deste algoritmo é a eliminação dos elementos tartaruga, já que em um Bubble Sort estes retardam a classificação tremendamente. Assim como o Bubble Sort, o algoritmo repetidamente reordena diferentes pares de itens. O que os difere é que o intervalo (comumente chamado pela palavra em inglês *gap*) entre um elemento e outro pode ser muito mais do que um, e este é calculado a cada passagem, tornando-o mais eficiente.

Algoritmo (em C)

```
1  int* comb_sort(int vetor[], int tamanho) {
2      float shrink_factor = 1.247330950103979;
3      int gap = tamanho, swapped = 1, swap, i;
4      while ((gap > 1) || swapped) {
5          if (gap > 1)
6              gap = gap/shrink_factor;
7          swapped = 0;
8          i = 0;
9          while ((gap+i) < tamanho) {
10             if (vetor[i]-vetor[i+gap] > 0) {
11                 swap = vetor[i];
12                 vetor[i] = vetor[i+gap];
13                 vetor[i+gap] = swap;
14                 swapped = 1;
15             }
16             ++i;
17         }
18     }
19     return vetor;
20 }
```

1. Vetor Inicial:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

2. O intervalo começa com o comprimento da lista a ser ordenada dividida pelo fator de encolhimento:

4	5	2	0	1	3	6
0	1	2	3	4	5	6

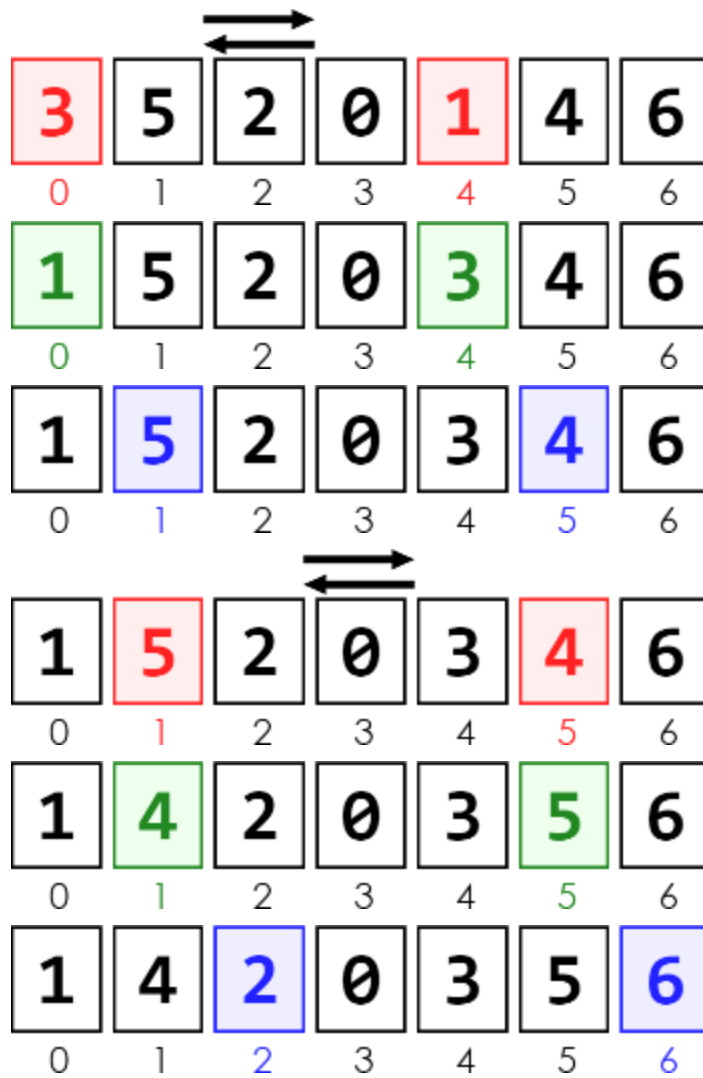
3. Ao encontrar um elemento desordenado, as posições são trocadas e as comparações continuam:

4	5	2	0	1	3	6
0	1	2	3	4	5	6
3	5	2	0	1	4	6
0	1	2	3	4	5	6
3	5	2	0	1	4	6
0	1	2	3	4	5	6

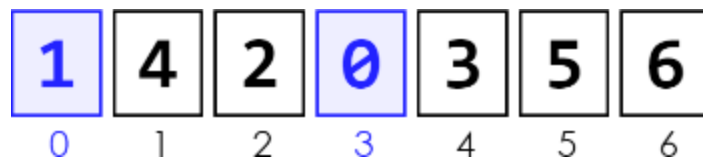
4. Então, a diferença é dividida pelo fator de encolhimento novamente e a lista é ordenada com o valor de 4 para o gap:

3	5	2	0	1	4	6
0	1	2	3	4	5	6

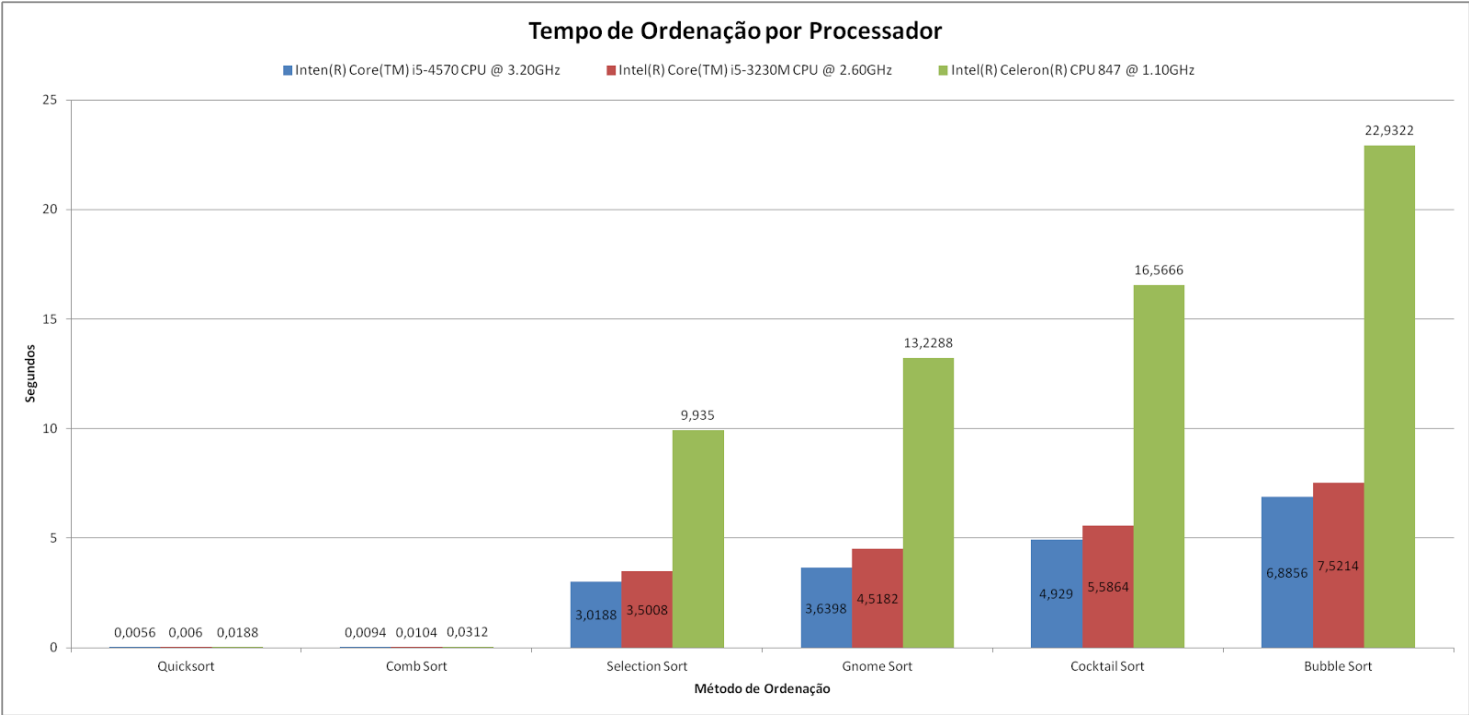
5. Ao encontrar um elemento desordenado, as posições são trocadas e as comparações continuam:



6. Então, a diferença é dividida pelo fator de encolhimento novamente e a lista é ordenada com o valor de 3 para o gap. Essa lógica é repetida até que o vetor esteja totalmente ordenado:



Comparação



Tempo de ordenação de 50.000 itens, por processador

Bibliografia

http://pt.wikipedia.org/wiki/Gnome_sort
http://pt.wikipedia.org/wiki/Cocktail_sort
http://pt.wikipedia.org/wiki/Comb_sort