

**UNIVERSIDADE DO ESTADO DO AMAZONAS
ESCOLA SUPERIOR DE TECNOLOGIA**

**Jonatas Travessa Souza de Barros
Lucas Mendes Sonoda
Natanael da Mota Figueira
Paulo André Carneiro Fernandes**

TRABALHO SOBRE O ALGORITMO DE ORDENAÇÃO COMB-SORT

**Manaus
2021**

Jonatas Travessa Souza de Barros
Lucas Mendes Sonoda
Natanael da Mota Figueira
Paulo André Carneiro Fernandes

TRABALHO SOBRE O ALGORITMO DE ORDENAÇÃO COMB-SORT

Trabalho apresentado como requisito parcial para obtenção de aprovação na disciplina Algoritmos e Estrutura de Dados II, no Curso de Bacharelado em Sistemas de Informação, na Universidade do Estado do Amazonas.

Prof. Sergio Cleger Tamayo

Manaus
2021

Introdução

Ordenar é de extrema importância nas mais diversas tarefas, pois facilita a busca por algum elemento, por exemplo, é mais fácil encontrar um nome em uma lista ordenada alfabeticamente do que se os nomes estivessem dispostos de forma aleatória. Em programação não é diferente, visto que o eficiente algoritmo da busca binária precisa se utilizar de um array de itens ordenados para encontrar o item a ser buscado. Neste trabalho será apresentado e analisado o algoritmo comb-sort, um, dentre tantos outros, desenvolvidos para a tarefa de ordenação.

Onde usar e como usar

O Comb-Sort é um algoritmo de ordenação que pode ser aplicado em arrays e listas dinâmicas, tendo uma eficiência maior que o Bubble-Sort com casos onde os valores menores ficam no final de um vetor. Para poder usar ele, primeiramente é necessário definir seu fator de encolhimento, que será usado para diminuir o intervalo entre os pares que serão comparados. No artigo original do algoritmo escrito por Włodzimierz Dobosiewicz e Artur Borowy em 1980, os autores sugeriram 1,3 depois de tentar milhares de listas e encontraram nesse valor, a melhor eficiência. O Comb-sort melhora o Bubble-sort, e rivaliza com algoritmos como o Quicksort. A ideia básica é eliminar as tartarugas ou pequenos valores próximos do final da lista, já que em um Bubble-sort estes retardam a classificação tremendamente.

Forma de ordenação

O Comb-sort repetidamente reordena diferentes pares de dados, separados por um intervalo, que começa com o comprimento do vetor e a cada passagem é dividido pelo fator de encolhimento, sendo considerada a parte inteira do valor. É considerado uma melhoria do Bubble-sort, pois o Comb-sort consegue fazer as ordenações em menos passos, pois ao invés de sempre comparar os elementos de posições adjacentes, ele utiliza um intervalo e com isso, pula várias comparações que o Bubble-sort faz.

Usualmente, o primeiro intervalo pode ser o próprio tamanho do vetor ou esse tamanho dividido pelo fator de encolhimento (parte inteira dessa divisão). A partir disso, em cada passagem pelo vetor haverá as comparações entre elementos separados pelo intervalo (gap). Em cada comparação, quando o primeiro valor é maior que o segundo, eles trocam de posição, caso contrário, apenas vai para a próxima comparação, e o processo se repete até que o segundo elemento da comparação seja o atual último elemento do vetor e então diminui o valor do intervalo dividindo-o por um fator pré estabelecido, esse processo é repetido até que o intervalo seja 1 e o vetor esteja ordenado. Uma melhor visualização do processo de ordenação será mostrado mais adiante com um exemplo passo a passo.

Estabilidade

Comb-sort não é um algoritmo de ordenação estável já que não ordena os elementos na mesma ordem em que aparecem inicialmente. É possível visualizar isso, a partir do exemplo passo a passo, há dois 1 no vetor (um laranja e outro azul), e ao final da ordenação eles não mantêm a mesma ordem de aparição inicial do vetor.

Exemplo Passo a Passo

Vetor:

-6	4	3	56	-10	1	23	8	21	1
----	---	---	----	-----	---	----	---	----	---

gap inicial: $10/1,3 \Rightarrow 7$

-6	4	3	56	-10	1	23	8	21	1
----	---	---	----	-----	---	----	---	----	---

-6	4	1	56	-10	1	23	8	21	3
----	---	---	----	-----	---	----	---	----	---

gap: $7/1,3 \Rightarrow 5$

-6	4	1	56	-10	1	23	8	21	3
----	---	---	----	-----	---	----	---	----	---

-6	4	1	21	-10	1	23	8	56	3
----	---	---	----	-----	---	----	---	----	---

gap: $5/1,3 \Rightarrow 3$

-6	4	1	56	-10	1	23	8	21	3
----	---	---	----	-----	---	----	---	----	---

-6	-10	1	21	4	1	3	8	56	23
----	-----	---	----	---	---	---	---	----	----

gap: $3/1,3 \Rightarrow 2$

-6	-10	1	21	4	1	3	8	56	23
----	-----	---	----	---	---	---	---	----	----

-6	-10	1	1	3	8	4	21	56	23
----	-----	---	---	---	---	---	----	----	----

gap: $2/1,3 \Rightarrow 1$

-6	-10	1	1	3	8	4	21	56	23
----	-----	---	---	---	---	---	----	----	----

-6	-10	1	1	3	4	8	21	23	56
----	-----	---	---	---	---	---	----	----	----

última iteração: vetor ordenado

-6	-10	1	1	3	4	8	21	23	56
----	-----	---	---	---	---	---	----	----	----

Complexidade

Visto que o Comb-sort é baseado no Bubble-sort, a complexidade de pior caso é $O(n^2)$. A complexidade de caso médio possui uma notação não muito usual, no caso o símbolo Ω , sendo dada por $\Omega(N^2/2^p)$, onde n é o número de elementos e p é a quantidade de vezes que o intervalo (gap) precisa ser decrementado. Na prática, isso significa que para o caso médio o Comb-sort não é mais eficiente do que $O(N^2/2^p)$. A complexidade de melhor caso é $O(n \log n)$, no entanto a complexidade de melhor caso só ocorre caso o vetor já esteja quase ordenado.

Quanto à complexidade de espaço, o Comb-sort é $O(1)$, ou seja, o algoritmo não precisa utilizar memória adicional para a ordenação.

Vantagens

- Não precisa de espaço adicional para ordenar os elementos.
- Funciona melhor que Bubble-sort no caso geral.
- A lógica é simples, fácil de entender e programar.
- A complexidade de espaço é $O(1)$.

Desvantagens

- A complexidade do pior caso é a mesma que a do Bubble-sort.
- Não funciona bem com grandes quantidades de dados.
- Não é uma ordenação estável.
- Existem algoritmos mais eficientes.

Código Comentado

Os códigos a seguir foram escritos na linguagem C++ para uma ordenação completa utilizando Comb-sort, há um código para uma ordenação crescente e outro para uma ordenação decrescente.

- Código para gerar o novo intervalo (gap).

```
1  #include <iostream>
2  #include <algorithm> //swap
3
4  using namespace std;
5
6  /* Calcula próximo gap */
7  int next_gap(int gap){
8
9      /* Gap não pode ser menor do que 1 */
10     if (gap <= 1)
11         return 1;
12
13     return gap/1.3;
14 }
```

- Código para ordenar um vetor de forma crescente usando o Comb-sort.

```
16  /* Comb_sort crescente */
17  void comb_sort_cres(int *v, int size){
18
19      /* tamanho inicial do gap (tamnho do vetor) */
20      int gap = size;
21
22      int swapped = 1;
23
24      /* Ordena até gap == 1 e na última passagem não ter tido troca*/
25      while(gap != 1 || swapped == 1){
26          gap = next_gap(gap);
27
28          swapped = 0;
29
30          /* Loop de iteração sob o vetor considerando o gap */
31          for (int i = 0; i < size - gap; i++){
32              if (v[i] > v[i + gap]){
33                  swap(v[i], v[i + gap]);
34                  swapped = 1;
35              }
36          }
37      }
38  }
39 }
```

- Código para ordenar um vetor de forma decrescente usando o Comb-sort.

```
41  /* Comb_sort decrescente */
42  void comb_sort_decres(int *v, int size){
43
44      /* tamanho inicial do gap (tamnho do vetor) */
45      int gap = size;
46
47      int swapped = 1;
48
49      /* Ordena até gap == 1 e na última passagem não ter tido troca*/
50      while(gap != 1 || swapped == 1){
51          gap = next_gap(gap);
52
53          swapped = 0;
54
55          /* Loop de iteração sob o vetor considerando o gap */
56          for (int i = 0; i < size - gap; i++){
57              if (v[i] < v[i + gap]){
58                  swap(v[i], v[i + gap]);
59                  swapped = 1;
60              }
61          }
62      }
63  }
64 }
65 }
```

Codificação e Avaliação com 100000 Números

Para cada uma das configurações solicitadas, executou-se o algoritmo registrando o tempo de execução.

- Configuração 1: vetor com elementos totalmente desordenados;
- Configuração 2: vetor com elementos desordenados crescentemente;
- Configuração 3: vetor com elementos desordenados decrescentemente;
- Configuração 4: vetor com a primeira metade ordenada crescentemente e a segunda metade ordenada decrescentemente;
- Configuração 5: vetor com a primeira metade ordenada decrescentemente e a segunda metade ordenada crescentemente.

A figura seguinte é um exemplo simplificado das entradas e saídas de cada configuração, no exemplo é utilizado um vetor de 6 números positivos e negativos.

Figura 1: Exemplo simplificado ilustrativo das configurações

Configuração	=>	Ordenação
2 6 -7 1 9 -5	=>	-7 -5 1 2 6 9 Configuração 1
-7 -5 1 2 6 9	=>	-7 -5 1 2 6 9 Configuração 2
9 6 2 1 -5 -9	=>	-7 -5 1 2 6 9 Configuração 3
-7 -5 1 9 6 2	=>	-7 -5 1 2 6 9 Configuração 4
1 -5 -7 2 6 9	=>	-7 -5 1 2 6 9 Configuração 5

Um dos resultados obtidos de uma execução dessas configurações utilizando 100000 (cem mil) números pode ser conferido na próxima figura.

Figura 2: tempos de execução em milisegundos

```
Tempos de execuções (ms)
Configuração 1: 33.59400
Configuração 2: 19.58300
Configuração 3: 21.70700
Configuração 4: 20.24000
Configuração 5: 23.89900
```


Discussão dos Resultados

A partir da figura 2 temos um resultado dos tempos de execução para diferentes organizações de números em um vetor, é notável que a configuração 1 (vetor de elementos aleatórios) é a que demanda mais tempo para ser ordenada, requerendo quase o dobro do tempo necessário para ordená-la.

A configuração 2 (vetor já ordenado) requer o menor tempo. Nesse caso, o algoritmo não realiza trocas, sendo este tempo então, relativo apenas às iterações sobre o vetor considerando a diminuição do gap por um fator de 1,3, ou seja, basicamente complexidade de $O(n \log n)$.

Vale a pena ressaltar o relativo bom desempenho na configuração 3 (vetor ordenado de forma inversa), isso decorre do fato de que os primeiros elementos a serem trocados são os elementos menos centralizados, e portanto, já nas primeiras passagens, há uma tendência desses elementos já ocuparem suas posições ordenadas ou estarem bem próximas disso, diminuindo o total de trocas posteriores para ordená-los de forma crescente. Logo, é possível inferir que em uma situação de um vetor quase totalmente ordenado de forma inversa, o comb-sort é um bom algoritmo para tal situação, pois num cenário assim este algoritmo se aproxima da sua complexidade de melhor caso que é $O(n \log n)$.

Nas configurações 4 e 5 (uma metade já está ordenada e a outra está ordenada de forma inversa) há uma tendência de haver menos trocas durante as primeiras passagens (gap de tamanho grande) e mais trocas quando o valor do gap é aproximadamente metade do tamanho do vetor. A configuração 5 se mostrou um pouco mais lenta que a 4, pois a mesma teve mais trocas para conseguir ser ordenada.

Em termos gerais, quanto menos trocas houver, mais rápido é o comb-sort.

Referências

COMB SORT. **GeeksforGeeks**. Disponível em:

<<https://www.geeksforgeeks.org/comb-sort/>>. Acesso em: 7 de maio de 2021.

BHOJASIA, Manish. Comb Sort Multiple Choice Questions and Answers (MCQs).

Sanfoundry. Disponível em:

<<https://www.sanfoundry.com/comb-sort-multiple-choice-questions-answers-mcqs/>>. Acesso em: 10 de maio de 2021.

COMB SORT - ALGORITHMS. **H.urna**. Disponível em:

<<https://hurna.io/academy/algorithms/sort/comb.html>>. Acesso em: 10 de maio de 2021.

COMB SORT. **Mycarrerwise**. Disponível em:

<<https://mycareerwise.com/programming/category/sorting/comb-sort>>. Acesso em: 10 de maio de 2021.

COMB SORT. **Wikipedia**. Disponível em: <https://pt.wikipedia.org/wiki/Comb_sort>.

Acesso em: 13 de maio de 2021.

JONES, Matthew. Comb Sort - The Sorting Algorithm Family Reunion. **exceptionnotfound**. Disponível em:

<<https://exceptionnotfound.net/comb-sort-csharp-the-sorting-algorithm-family-reunion/>>. Acesso em: 10 de maio de 2021.