UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA

ESCOLA SUPERIOR DE TECNOLOGIA - EST

CURSO DE LICENCIATURA EM COMPUTAÇÃO

Kleyson De Melo Lopes
Lázaro Junior Machado Pontes

MÉTODOS DE ORDENAÇÃO:

BUBBLE SORT

Manaus – AM
2019

Kleyson De Melo Lopes
Lázaro Junior Machado Pontes

MÉTODOS DE ORDENAÇÃO:

BUBBLE SORT

Trabalho apresentado ao Curso de Licenciatura em Computação da Escola Superior de Tecnologia, como requisito parcial para obtenção de nota na disciplina Algoritmos e Estruturas de Dados II, ministrada pelo profº Dr. Sérgio Tamayo.

Manaus - AM
2019

INTRODUÇÃO

Em um mundo, onde a tecnologia está totalmente inclusa no cotidiano do homem moderno, grandes quantidades de dados como fotos, vídeos, documentos, listas telefônicas são armazenados e listados; para acessá-los buscas mais rápidas e eficientes tornam-se um problema a ser resolvido. Para isso é necessário o uso de algoritmos e métodos de ordenação.

Este trabalho objetiva-se a apresentar o método de ordenação **Bubble Sort** (Bolha), suas utilidades, vantagens, desvantagens, formas de ordenação, complexidade e por meio de experimentação (algoritmo com funções e dados gerados aleatoriamente) verificar eficácia e tempos de execução.

BUBBLE SORT

Bubble sort ou Método Bolha (ordenação por flutuação) é o algoritmo mais simples de ordenação por troca, porém é o mais ineficiente entre os métodos de ordenação simples. Neste algoritmo faz-se uma busca no vetor inteiro, comparando elementos adjacentes (dois a dois) da esquerda para a direita, ou seja, cada elemento da posição i será comparado com o elemento da posição i + 1, caso o elemento i + 1 for menor que o elemento da posição i, eles trocam de lugar, e assim sucessivamente até que todo o vetor seja percorrido e não haja mais trocas. A ideia é fazer flutuar para o topo ou última posição do vetor o maior elemento da sequência. Devido a essa forma de flutuação dos elementos, o método é comparado a bolhas em um tanque, daí o nome de método bolha. Este método é muito utilizado para listas, ou arquivos pequenos, ordenados ou semi-ordenados

Vantagens e Desvantagens

O Bubble Sort é um método interno simples, com um algoritmo pequeno, de fácil compreensão e implementação. **Estável**, pois se elementos iguais forem comparados, não se faz a troca, preservando-os em suas posições.

Devido sua forma de comparação entre os elementos, o vetor que o método varrer terá que percorrer quantas vezes que for necessária, tornando o algoritmo ineficiente, extremamente lento e de custo elevado de tempo. Sendo assim não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

Complexidade

No **Pior Caso**, temos um vetor em ordem decrescente, onde as operações de comparações e de troca de posição de elementos são feitas para todos os elementos, o algoritmo realizará n-1 troca para o primeiro passo, depois n-2 trocas para o segundo elemento e assim sucessivamente. Trocas = n-1+n-2+n-3...+2+1 aproximadamente n^2 trocas. Isso também acontece para o **caso médio**, onde teremos um vetor semi-ordenado.

No **Melhor Caso**, temos um vetor ordenado, desta forma, nenhuma troca

será realizada, pois o algoritmo percorrerá apenas uma vez, sendo assim seu tempo de execução é de ordem N comparações.

O tempo gasto na execução do algoritmo varia em ordem quadrática(n^2) em relação ao número de elementos a serem ordenados.

Iterações e Troca: passo a passo

Assumimos o vetor baixo como objeto de estudo:
vetor = {25, 57, 48, 37, 12, 92, 86, 33}

Sabemos que iremos repetir o vetor n -1 vezes. O tamanho do vetor é 8, logo iremos repetir 7 vezes o vetor (8-1).

Vamos chamar cada repetição de passo e as trocas e varreduras de iterações. Também usaremos um algoritmo do método bolha otimizado, onde caso na varredura pelo vetor constatar-se que o mesmo já encontrasse ordenado, o algoritmo para.

Assumiremos como Passo 0 o vetor original

Passo 1:

0	25	57	48	37	12	92	86	33	comparamos 25 e 57: não há troca
1	25	48	57	37	12	92	86	33	comparamos 57 e 48: há troca
2	25	48	37	57	12	92	86	33	comparamos 57 e 37: há troca
3	25	48	37	12	57	92	86	33	comparamos 57 e 12: há troca
4	25	48	37	12	57	92	86	33	comparamos 57 e 92: não há troca
5	25	48	37	12	57	86	92	33	comparamos 92 e 86: há troca
6	25	48	37	12	57	86	33	92	comparamos 92 e 33: há troca

Chegamos ao fim do primeiro passo e, como visto, não foi suficiente para ordenar o vetor.

Teremos que reiniciar, só que agora sabemos que, pelo menos, o último valor (i==7) já está em seu devido lugar

Então iremos marcá-lo e não precisaremos percorrer todo o vetor no segundo passo.

Passo 2:

0	25	48	37	12	57	86	33	92	comparamos 25 e 48: não há troca
1	25	37	48	12	57	86	33	92	comparamos 48 e 37: há troca
2	25	37	12	48	57	86	33	92	comparamos 48 e 12: há troca
3	25	37	12	48	57	86	33	92	comparamos 57 e 12: não há troca
4	25	37	12	48	57	86	33	92	comparamos 57 e 86: não há troca
5	25	37	12	48	57	33	86	92	comparamos 86 e 33: há troca
6	25	37	12	48	57	33	86	92	Não há iteração

Novamente chegamos ao final do segundo passo, o vetor ainda se encontra parcialmente desordenado. Teremos que reiniciar, só que agora sabemos que, pelo menos, o dois último e maiores valores (86 e 92) já estão em seus lugares. Então iremos marcá-los e não precisaremos percorrer todo o vetor no terceiro passo.

Passo 3:

0	25	37	12	48	57	33	86	92	comparamos 25 e 37: não há troca
1	25	12	37	48	57	33	86	92	comparamos 37 e 12: há troca
2	25	12	37	48	57	33	86	92	comparamos 37 e 48: não há troca
3	25	12	37	48	57	33	86	92	comparamos 48 e 57: não há troca
4	25	12	37	48	33	57	86	92	comparamos 57 e 33: há troca
5	25	12	37	48	33	57	86	92	Não há iteração
6	25	12	37	48	33	57	86	92	Não há iteração

No terceiro passo, apesar das trocas terem diminuído, o vetor ainda se encontra parcialmente desordenado. Teremos que retornar ao loop de iterações, só que agora os três últimos e maiores valores (57, 86, 92) já estão em seus lugares. Então os marcamos, assim não precisando percorrer todo o vetor no quarto passo.

Passo 4:

0	12	25	37	48	33	57	86	92	comparamos 12 e 25: há troca
1	12	25	37	48	33	57	86	92	comparamos 25 e 37: não há troca
2	12	25	37	48	33	57	86	92	comparamos 37 e 48: há troca
3	12	25	37	33	48	57	86	92	comparamos 48 e 33: há troca
4	12	25	37	33	48	57	86	92	Não há iteração
5	12	25	37	33	48	57	86	92	Não há iteração
6	12	25	37	33	48	57	86	92	Não há iteração

No quarto passo, o vetor ainda se encontra parcialmente desordenado. Teremos que retornar ao loop de iterações, só que agora os quatro últimos e maiores valores (48, 57, 86, 92) já estão em seus lugares. Então os marcamos, assim não precisando percorrer todo o vetor no quinto passo.

Passo 5:

0	12	25	37	33	48	57	86	92	comparamos 12 e 25: há troca
1	12	25	37	33	48	57	86	92	comparamos 25 e 37: não há troca
2	12	25	33	37	48	57	86	92	comparamos 37 e 33: há troca
3	12	25	33	37	48	57	86	92	Não há iteração
4	12	25	33	37	48	57	86	92	Não há iteração
5	12	25	33	37	48	57	86	92	Não há iteração
6	12	25	33	37	48	57	86	92	Não há iteração

No quinto passo, faz-se a última troca. Porém é necessário mais um passo passo para termos a confirmação que o vetor está totalmente ordenado. Teremos que retornar ao loop de iterações, só que agora os cinco últimos e maiores valores (37, 48, 57, 86, 92) já estão em seus lugares. Então os marcamos, assim não precisando percorrer todo o vetor no sexto passo.

Passo 6:

0	12	25	33	37	48	57	86	92	Vetor completamente ordenado
1	12	25	33	37	48	57	86	92	Iteração não verificada
2	12	25	33	37	48	57	86	92	Não há iteração
3	12	25	33	37	48	57	86	92	Não há iteração
4	12	25	33	37	48	57	86	92	Não há iteração
5	12	25	33	37	48	57	86	92	Não há iteração
6	12	25	33	37	48	57	86	92	Não há iteração

No sexto passo, consta-se na iteração 0 que o vetor encontra-se ordenado, isso devido ao algoritmo otimizado, caso contrário ele ainda faria mais 2 iterações e dois passos.

Implementação

Abaixo segue uma implementação em C++ do método Bubble Sort:

```
void BubbleSort(int n, int *vetor){
	// Vamos ter que percorrer todo o vetor, logo:
	 for (int i = 1; i < n; i++) {
		// Dentro de cada iteração
		// percorremos novamente o vetor
		// em busca dos pares
		for (int j = 0; j < n - i; j++) {
			// Comparamos
			if (vetor[j] > vetor [j + 1]){
				//(trocamos)
				aux = vetor[j];
				vetor[j] = vetor [j + 1];
				vetor [j + 1] = aux;
}}}}
```

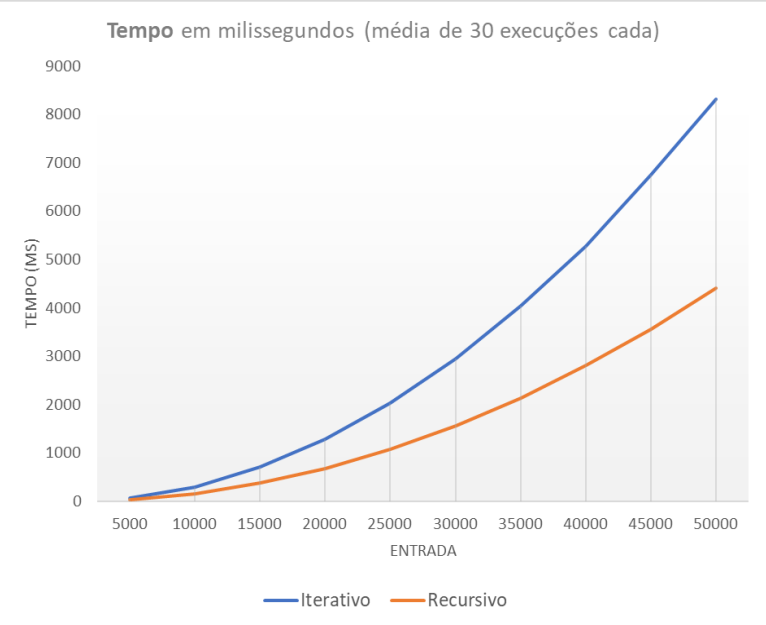
Codificação e avaliação com 100000 números

Tendo um vetor com 100.000 valores como base, e utilizando o método de ordenação bubble sort na forma iterativa, otimizada e recursiva, buscamos através do método identificar a métrica de tempo e memória utilizada pelo sistema

operacional para que o método deixe o vetor totalmente ordenado, levando em conta as várias formas em que o vetor se encontrará ordenado:

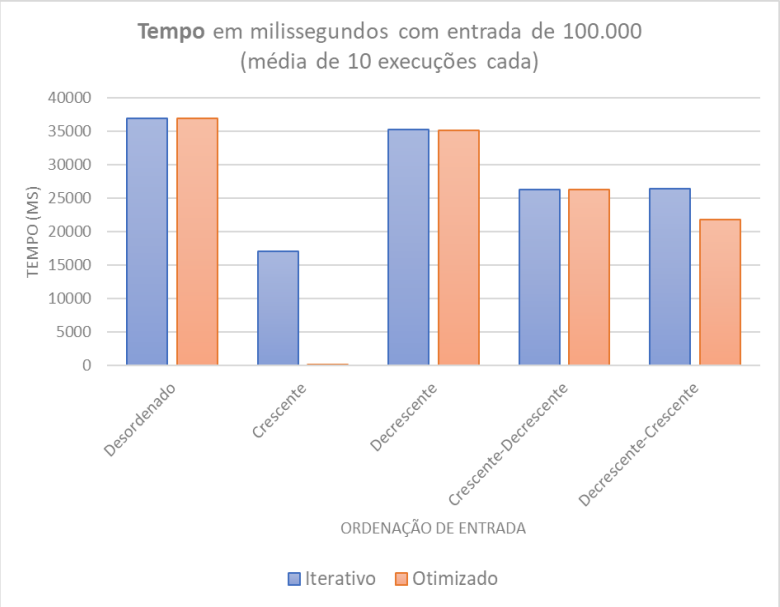
- Crescente;
- Decrescente;
- Decrescente-crescente;
- Crescente-decrescente.

Bubble Sort (Aleatório) - 30 execuções para cada				
	Iterativo		Recursivo	
Entrada	Tempo	Memória	Tempo	Memória
5000	63	408	36	468
10000	299	448	158	576
15000	708	472	372	656
20000	1290	480	680	744
25000	2035	496	1075	828
30000	2940	516	1555	920
35000	4040	536	2135	1008
40000	5280	556	2805	1096
45000	6750	576	3560	1184
50000	8310	596	4410	1272



A tabela e o gráfico acima, revelam que vetores de tamanho que vão de 5000 a 50000 estando com seus valores de forma aleatória, quando varridos pelo algoritmo de ordenação bubble sort utilizam mais memória de sistema e tempo conforme a quantidade de valores no vetor aumentam.

Bubble Sort (Entrada: 100.000) - 10 execuções cada		
	Iterativo	Iterativo otimizado
Entrada	Tempo	Tempo
Desordenado	36892	36886
Crescente	17067	1
Decrescente	35263	35180
Crescente-Decrescente	26252	26303
Decrescente-Crescente	26390	21808



Analisando tanto a tabela quanto o gráfico acima, identificou-se que o algoritmo de ordenação bubble sort na versão otimizada tem uma execução mais eficaz em relação a sua versão iterativa, com um tempo bem baixo quando o vetor encontrasse em ordem crescente, ou seja ordenado. Já quando o vetor está em ordem decrescente ou desordenado, verificamos que o tempo de execução do algoritmos é maior, tanto para a versão otimizada quanto iterativa, isso por que o algoritmo faz uma busca de ordem quadrática (n^2).

REFERÊNCIAS

LAUREANO, Marcos**. Estruturas de Dados com Algoritmos e C.** Rio de Janeiro: Brasport Livros e Multimídia Ltda, 2008.

ZIVIANI, NIVIO. **Projeto de Algoritmos com Implementações em Java e C++**. São Paulo: Thomson Pioneira, 2006.

HONORATO, Bruno de Almeida. **Algoritmos de Ordenação: Análise e Comparação.** Rio de Janeiro: 2013. Disponível em: <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/2826> Acesso em: 17 de agosto de 2019.

WIKIPEDIA. **Bubble Sort**. Disponível em: <https://pt.wikipedia.org/wiki/Bubble_sort> Acesso em: 16 de agosto de 2019.

GATTO, Elaine Cecília. **Algoritmos de Ordenação: Bubble Sort**. São Paulo: 2017. Disponível em: <https://www.embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/> Acesso em: 16 de agosto de 2019.