ESTRUTURA DE DADOS

Matheus da Silva Serpa

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Ordenação de dados — métodos simples

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Apontar o método *Bubblesort*.
- Identificar o método *Selectionsort*.
- Explicar o método *Insertionsort*.

Introdução

Os algoritmos que implementam as técnicas de pesquisa mais eficientes têm como requisito que os dados estejam ordenados no momento da busca. Algoritmos de compressão de dados, de busca em grafos, de árvore geradora mínima, entre outros, utilizam ordenação de alguma maneira (EDELWEISS; GALANTE, 2009).

Um algoritmo de ordenação tem como entrada um vetor ou uma lista de dados, cujos elementos necessitam ser ordenados. A saída é o vetor, ou a lista, com os seus elementos ordenados de acordo com alguma política. Como exemplos, temos: ordenar em ordem crescente, decrescente (numérica), ascendente e descendente (lexicográfica).

Neste capítulo, você vai estudar sobre os métodos de ordenação simples, que são *Bubblesort*, *Selectionsort* e *Insertionsort*. Você vai analisar as vantagens e desvantagens de cada um desses métodos, tornando possível decidir qual é o melhor para cada problema da vida real.

1 Ordenação por bolha

O *Bubblesort*, ou ordenação por bolha, como é popularmente conhecido, leva esse nome, pois sua estrutura foi pensada como se os elementos maiores do vetor desordenado fossem mais leves e flutuassem como bolhas até as suas posições corretas.

Dado um vetor ou uma lista de elementos que desejamos ordenar, comparamos os elementos dois a dois até que o vetor esteja completamente ordenado. Por exemplo, em um vetor de cinco elementos, vamos comparar o 1º elemento com o 2º, o 2º com o 3º, e assim sucessivamente. Essas operações são repetidas até que nenhuma troca de elementos ocorra, o que indica que o vetor está ordenado.

Por exemplo, dado um vetor com N = 8 elementos, vamos ordená-lo de forma crescente.

26	49	38	14	58	87	35	76

Vamos executar o algoritmo da esquerda para a direita (o inverso também é permitido e correto). Na primeira iteração, o 26 é comparado com o 49; uma vez que o 49 é maior, não faremos nada. Na próxima iteração, o 49 é comparado com o 38, pois este é menor, e vamos trocá-lo de lugar com o 49. Continuamos a execução comparando o 49 com o 14. Uma vez que o 14 é menor, também trocamos os dois de posição. Nas próximas duas comparações, que são 49 com 58 e 58 com 87, os valores são maiores, logo, não devemos fazer nada. Prosseguindo, comparamos o 87 com o 35, e como o 35 é menor, realizamos a troca. Finalmente, comparamos o 87 com o 76 e realizamos a última comparação da primeira rodada. Nessa primeira rodada, o valor 87, que é o maior do vetor, flutuou para a sua posição, como mostra o quadro a seguir.

Original	26	49	38	14	58	87	35	76
Rodada 1	26	38	14	49	58	35	76	87
Rodada 2	26	14	38	49	35	58	76	87
Rodada 3	14	26	38	35	49	58	76	87
Rodada 4	14	26	35	38	49	58	76	87

O algoritmo continua sua execução começando novamente do elemento 26 e indo até o 87. Existe uma otimização após cada execução até o final e, na próxima rodada, podemos não comparar com o elemento que flutuou. Por exemplo, não faz sentido comparar mais algum elemento com o 87, pois já sabemos que ele é o maior elemento. Na próxima rodada, não fará sentido comparar algum elemento com os últimos dois elementos do vetor, pois já sabemos que eles são os dois maiores. Após quatro rodadas, o vetor ficou ordenado, como mostra o quadro acima.

Utilizamos o algoritmo *Bubblesort* para ordenar vetores de inteiros, entretanto, ele e todos os outros algoritmos de ordenação podem ser utilizados para ordenar diferentes tipos de dados, como *strings*, números reais, inteiros e também *structs*.

Link

Acesse o *link* a seguir para ver um exemplo, passo a passo, de ordenação de valores utilizando o algoritmo *Bubblesort*.

https://qrgo.page.link/csN1T

O pseudocódigo e o algoritmo podem ser vistos a seguir. Na linha 1 do pseudocódigo, utilizando um laço faça-enquanto, repetimos a operação de trocar elementos enquanto existem trocas de posição na lista. Na linha 2, definimos no início de toda a iteração externa a variável, e se ocorreu troca, definindo-a como falso, ou seja, zero. Na linha 3, temos um laço de repetição que vai da primeira posição até a penúltima do vetor que desejamos ordenar. Após, comparamos o elemento da posição i com o da posição i + 1 e, caso o elemento anterior seja maior que o próximo, realizamos a troca de posição dos dois elementos. Essa troca é realizada na linha 5 utilizando uma função já definida. Após, na linha 6, quando ocorrem trocas, marcamos a troca como verdadeiro, ou seja, um. Finalmente, na linha 7, é verificado se as trocas ocorreram ou não. Caso sim, a execução continua; caso contrário, a execução termina e a lista está ordenada.

A implementação em Python segue a mesma ideia do pseudocódigo. Na linha 1, definimos a função. Na linha 2, utilizando um laço infinito, repetimos a execução até que não ocorram trocas. A definição dessa variável é feita na linha 3, onde a marcamos como falsa. Após, na linha 4, percorremos toda a lista do início até a penúltima posição. Na linha 5, comparamos os elementos para verificar se uma troca é necessária. Caso sim, nas linhas 6 a 9, realizamos a troca e indicamos que ocorreu uma troca. Finalmente, na linha 10, testamos se alguma troca ocorreu nessa iteração do laço da linha 2. Caso não tenha ocorrido, a execução é interrompida.

Pseudocódigo	Implementação em Python
BUBBLE-SORT[A, N] 1. **faça** 2. ocorreuTroca = 0 3. **para** i = 0 **até** N - 1 4. **se** A[i] > A[i + 1] 5. **então** 6. **troca**(A[i], A[i + 1]) 7. ocorreuTroca = 1 8. **enquanto**(ocorreuTroca == 1)	1. def bubble_sort(lista, N): 2. while True: 3. ocorreu_troca = False 4. for i in range (0, N - 1): 5. if lista[i] > lista[i + 1]: 6. troca = lista[i] 7. lista[i] = lista[i + 1] 8. lista[i + 1] = troca 9. ocorreu_troca = True 10. if ocorreu_troca == False: 11. break

A ordenação por bolha, ou *Bubblesort*, é indicada, principalmente, para aprendizado e primeiro contato com algoritmos de ordenação. Seu desempenho é um dos menores, quando comparado com outros algoritmos. A complexidade de tempo de melhor caso é linear, ou seja, O(N). Isso indica que, no melhor caso, o qual seria tentar ordenar um vetor ordenado, o algoritmo precisa percorrer todos os elementos. No caso da complexidade de pior caso, quando o vetor está ordenado de forma contrária da que buscamos ordenar, a complexidade de tempo é quadrática, ou seja, $O(N^2)$.

Fique atento

Devemos lembrar que nos algoritmos de ordenação, em geral, apenas as chaves devem estar ordenadas, mas não necessariamente os dados. Uma estrutura pode ter nome do aluno, curso e idade. O usuário pode desejar ordenar por idade — neste caso, os outros campos não precisam ser ordenados.

Por exemplo, dada uma lista de nomes e idades com N = 5 elementos, vamos ordená-la de forma ascendente (lexicográfica) de acordo com o nome, ou seja, Airton vem antes de Maria. Confira, a seguir, a aplicação do *Bubblesort* com *strings*.

Sarah, 21	Luana, 27	Isabeli, 29	Bruna, 25	Mariana, 25

Vamos executar o algoritmo da esquerda para a direita. Uma vez que nossa chave de ordenação é o nome, não precisamos levar a idade em consideração. Logo, na primeira iteração, Sarah é comparada com Luana. Uma vez que a letra S vem após a letra L, o elemento é trocado. Após, Sarah é comparada com Isabeli, Bruna e Mariana e é trocada nos três casos. Nesse momento, no final da primeira rodada, Sarah está na posição correta. Após, na segunda rodada, Mariana já está na posição correta, mas o algoritmo não identifica isso e executa da mesma maneira. Ao longo dessa rodada, Luana troca de posição com Isabeli e Bruna. Ao final, na rodada 3, todo vetor está ordenado em ordem ascendente (lexicográfica) por nome. Confira no quadro a seguir o passo a passo da execução do algoritmo *Bubblesort*.

Original	Sarah, 21	Luana, 27	Isabeli, 29	Bruna, 25	Mariana, 25
Rodada 1	Luana, 27	Isabeli, 29	Bruna, 25	Mariana, 25	Sarah, 21
Rodada 2	Isabeli, 29	Bruna, 25	Luana, 27	Mariana, 25	Sarah, 21
Rodada 3	Bruna, 25	Isabeli, 29	Luana, 27	Mariana, 25	Sarah, 21

2 Ordenação por seleção

O *Selectionsort*, ou ordenação por seleção, é um algoritmo que busca o menor elemento (ou o maior depende do tipo de ordenação que está sendo feita) de cada iteração e o coloca na sua posição correta.

Dado um vetor ou uma lista de elementos que desejamos ordenar, esse algoritmo irá buscar na primeira iteração o menor elemento de todos e colocá-lo na primeira posição do vetor. Na segunda iteração, ele buscará pelo segundo menor e o colocará na posição dois, e assim sucessivamente. Essa operação é repetida até que todos os elementos sejam colocamos em suas posições de forma ordenada.

Por exemplo, dado um vetor com N = 8 elementos, vamos ordená-lo de forma crescente, conforme mostra o quadro a seguir, com a aplicação do *Selectionsort*.

28	44	35	12	55	85	31	96

Vamos executar o algoritmo da esquerda para a direita (o inverso também é permitido e correto). Na primeira iteração, vamos comparar elemento a elemento até encontrar o menor de todos. Primeiro, vamos assumir que o 28 é o menor e ir fazendo as comparações. Comparamos com o 44 e o 35, continuando o 28 como menor. Após, comparamos com o 12, que vira o menor. Por fim, comparamos o 12 com o 55, o 85, o 31 e o 96, concluindo que o 12 é o menor elemento de todos. Agora, trocamos o 28, valor que está na primeira posição, pelo 12, e assim terminamos a primeira rodada. O quadro a seguir mostra a execução de rodada a rodada, sendo que, em negrito, encontra-se o valor que foi colocado na posição correta na respectiva rodada. Confira no quadro a seguir o passo a passo da execução do algoritmo *Selectionsort*.

Original	28	44	35	12	55	85	31	96
Rodada 1	**12**	44	35	28	55	85	31	96
Rodada 2	12	**28**	35	44	55	85	31	96
Rodada 3	12	28	**31**	44	55	85	35	96
Rodada 4	12	28	31	**35**	55	85	44	96
Rodada 5	12	28	31	35	**44**	85	55	96
Rodada 6	12	28	31	35	44	**55**	85	96
Rodada 7	12	28	31	35	44	55	**85**	96
Rodada 8	12	28	31	35	44	55	85	**96**

O algoritmo continua sua execução e agora não é necessário mais compará-lo com o elemento na posição 1. Começamos na posição 2 e buscamos o menor elemento do vetor restante. Não faria sentido comparar com a posição 1, pois, após a primeira iteração, já colocamos o valor correto naquela posição. Após oito rodadas, o vetor ficou ordenado, como mostra o quadro acima. É importante salientar que as rodadas 7 e 8 são necessárias, mesmo que não ocorram trocas. O algoritmo não consegue detectar se o vetor já está ordenado sem fazer rodada a rodada até o fim.

Link

Acesse o *link* a seguir para aprender mais sobre a complexidade e o desempenho da ordenação por seleção.

https://qrgo.page.link/oyryL

O pseudocódigo e o algoritmo podem ser vistos a seguir. Na linha 1 do pseudocódigo existe um laço responsável por dizer qual elemento vamos colocar na posição correta agora. Na linha 2, assumimos que o menor elemento dessa iteração encontra-se na posição **i**. Após, na linha 3, começa um novo laço que compara elemento por elemento até o fim do vetor, a fim de descobrir o menor daquela iteração. Na linha 4, realizamos esse teste para verificar se o elemento atual é menor que o nosso menor global. Após, na linha 5, caso o elemento atual seja menor, atualizamos a variável **min** para esta ter o novo menor elemento. Finalmente, nas linhas 6 e 7, realizamos as trocas, caso necessário. Essa troca é feita para colocar o menor elemento daquela iteração na sua posição correta.

A implementação em Python funciona, basicamente, da mesma forma que o pseudocódigo. Na linha 1, definimos a função e os parâmetros, que são a lista ou o vetor e o tamanho dela. Após, na linha 2, criamos um laço para dizer qual posição será ordenada naquela iteração. Após, na linha 3, declaramos a variável **min** com a suposição inicial que o elemento menor é o próprio elemento da posição **i**. Na linha 4, começamos a percorrer o vetor restante em busca do menor elemento existente. Na linha 5, realizamos um teste para verificar se o elemento atual é menor que o elemento da variável **min**. Caso isso seja verdade, atualizamos o valor de **min**. Na sequência, na linha 7, após descobrir qual é o menor valor daquela iteração, caso o menor valor e o valor que ocupa a posição analisada sejam diferentes, trocamos o menor valor pelo valor que estava armazenado naquela posição. A operação de troca é realizada nas linhas 8 a 10.

Pseudocódigo	Implementação em Python
SELECTION-SORT[A, N] 1. **para** i = 0 **até** N - 1 2. min = i 3. **para** j = i + 1 **até** N 4. **se** A[j] < A[min] **então** 5. min = j 6. **se** A[i] != A[min] **então** 7. **troca**(A[i], A[min])	1. def selection_sort(lista, N): 2. for i in range (0, N - 1): 3. min = i 4. for j in range (i + 1, N): 5. if lista[j] < lista[min]: 6. min = j 7. if lista[i] != lista[min]: 8. troca = lista[i] 9. lista[i] = lista[min] 10. lista[min] = troca

A ordenação por seleção, ou *Selectionsort*, tem uma particularidade em relação à sua complexidade de tempo. Ambos os melhores e os piores casos são quadráticos $O(N^2)$, ou seja, tanto para um vetor ordenado quanto para um totalmente desordenado, o desempenho do algoritmo será o mesmo. Comparando-o com o *Bubblesort* visto anteriormente, o desempenho no melhor caso é pior, pois, no caso do *Bubblesort*, basta uma varredura linear O(N) para descobrir que o vetor está ordenado.

Exemplo

Vamos assumir que um novo sistema está sendo construído para concursos federais e precisamos implementar a parte do sistema que ranqueia os candidatos por nota na prova.

Uma vez que os candidatos são ranqueados pela primeira vez, nas próximas iterações não é necessário ordenar de novo. Entretanto, você não tem como garantir que o vetor está ordenado sem verificar.

Confira um exemplo de lista com as notas dos candidatos:

- Aline Cavalcanti Carvalho, 3.5
- Carlos Barros Ribeiro, 9.6
- Vinicius Martins Azevedo, 2.1
- Nicole Sousa Correia, 7.0

Após a ordenação, os dados estarão da seguinte forma:

- Carlos Barros Ribeiro, 9.6
- Nicole Sousa Correia, 7.0
- Aline Cavalcanti Carvalho, 3.5
- Vinicius Martins Azevedo, 2.1

Nas próximas consultas a essa lista, caso novos candidatos não sejam adicionados, o vetor não precisará ser ordenado. Você sabe qual seria o algoritmo mais eficiente para esse caso? O algoritmo de ordenação mais eficiente é o que tem a menor complexidade de tempo de melhor caso, uma vez que já estar ordenado é o melhor caso para os algoritmos de ordenação. Dos algoritmos vistos até agora, o *Bubblesort* é o que tem a menor complexidade de tempo, O(N).

3 Ordenação por inserção

O *Insertionsort*, ou ordenação por inserção, é um algoritmo baseado em um jogo de cartas (ZIVIANI, 2004). Você está com um conjunto de cartas já ordenadas. Quando receber uma nova carta, deve colocá-la na posição correta, de forma que as cartas continuem ordenadas.

Dado um vetor ou uma lista de elementos que desejamos ordenar, comparamos os elementos de um subvetor de tamanho **i** a cada iteração, de forma que esse subvetor estará sempre ordenado. Por exemplo, para **i** igual a 2, sabemos que, dentro do subvetor de tamanho 2, os elementos estão ordenados, e assim por diante, até que **i** seja igual ao número de elementos do vetor, o qual estará completamente ordenado.

Por exemplo, dado um vetor com N = 8 elementos, vamos ordená-lo de forma crescente. Confira na sequência a aplicação do *Insertionsort*:

6	5	3	1	8	7	2	4

Na primeira rodada, o escolhido é o 5. Vamos compará-lo com todos os elementos anteriores; nesse caso apenas o 6. Uma vez que o 5 é menor, vamos trocá-lo de posição. Na segunda rodada, o escolhido é o 3; vamos compará-lo com todos os anteriores, o 5 e o 6. Após as comparações, o 3 fica na primeira posição. Na terceira rodada, o escolhido é o 1, o qual é comparado com o 3, o 5 e o 6, sendo movido para a primeira posição. Na sequência, o escolhido é o 8, que já está na posição correta dentro desse subvetor. Chegamos nessa conclusão comparando-o com o 6 e verificando que não existia elemento maior que ele no subvetor. Continuando as comparações, agora com o 7 como escolhido, a única mudança será a troca do 7 pelo 8. Na rodada 6, o escolhido é o 2, elemento que, comparado com o 3, o 5, o 6, o 7 e o 8, é menor que eles, logo, é movido até chegar na sua posição correta, que é logo após o 1. Finalmente, o escolhido é o 4, este é comparado com todos os elementos anteriores do vetor, que vão de 1 até 8, até que encontra a sua posição entre o 3 e o 5. Na rodada 7, o vetor estará completamente ordenado. Confira no quadro a seguir o vetor original e o passo a passo a cada rodada da execução do algoritmo *Insertionsort*.

Original	6	5	3	1	8	7	2	4
Rodada 1	**5**	**6**	3	1	8	7	2	4
Rodada 2	**3**	**5**	**6**	1	8	7	2	4
Rodada 3	**1**	**3**	**5**	**6**	8	7	2	4
Rodada 4	**1**	**3**	**5**	**6**	**8**	7	2	4
Rodada 5	**1**	**3**	**5**	**6**	**7**	**8**	2	4
Rodada 6	**1**	**2**	**3**	**5**	**6**	**7**	**8**	4
Rodada 7	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

O pseudocódigo e o algoritmo podem ser vistos a seguir. Na linha 1 do pseudocódigo existe um laço que vai da posição 1 até a N da lista; após, na linha 2, o escolhido. Na linha 3, decidimos de onde as comparações vão começar, lembrando que, nesse caso, é de forma reversa, ou seja, de trás para frente. Na linha 4, com um laço, comparamos elemento a elemento do subvetor, de forma a ordená-lo. Na linha 5, os elementos da lista são trocados de posição e, na linha 6, o contador é decrementado. Por fim, na linha 7, o escolhido é armazenado na posição correta.

A implementação em Python funciona basicamente da mesma forma que o pseudocódigo. Na linha 1, definimos a função e os parâmetros, que são: a lista, ou o vetor, e o tamanho dela. Após, na linha 2, o laço de repetição é declarado. Após, na linha 3, decidimos o escolhido, que é a posição **i** da lista. Na linha 4, o contador **j** é declarado tendo o valor **i – 1**. Esse índice é utilizado para comparar os elementos do subvetor atual. Na linha 5, um novo laço é declarado, o qual vai percorrer o subvetor de trás para frente, ou seja, da posição **j** até a 0. Na linha 6, os elementos são trocamos de posição até que o subvetor esteja ordenado. Na linha 7, decrementamos o contador. Por fim, na linha 8, o escolhido é armazenado na sua posição correta.

Pseudocódigo	Implementação em Python
INSERTION-SORT[A, N] 1. **para** i = 1 **até** N 2. escolhido = A[i] 3. j = i - 1 4. **enquanto** j >= 0 **e** A[j] > escolhido 5. A[j + 1] = A[j] 6. j-- 7. A[j + 1] = escolhido	1. def insertion_sort(lista, N): 2. for i in range (1, N): 3. escolhido = lista[i] 4. j = i - 1 5. while j >= 0 and lista[j] > escolhido: 6. lista[j + 1] = lista[j] 7. j = j - 1 8. lista[j + 1] = escolhido

A ordenação por inserção, ou *Insertionsort*, tem diferentes complexidades de tempo para melhor e pior caso. Para melhor caso, aquele no qual o vetor já está ordenado, a complexidade de tempo desse algoritmo é linear, ou seja, O(N). No entanto, para os casos nos quais o vetor está totalmente desordenado, o desempenho assintótico desse algoritmo é quadrático $O(N^2)$.

Saiba mais

Existem inúmeras aplicações dos algoritmos de ordenação. Leia mais sobre o assunto no livro *Matemática discreta e suas aplicações*, de Kenneth H. Rosen.

Para saber mais sobre o ensino de algoritmos de ordenação utilizando jogos, leia o artigo "SORTIA: um jogo para ensino de algoritmo de ordenação: estudo de caso na disciplina de estrutura de dados", de Paulo E. Battistella, Aldo von Wangenheim e Christiane G. von Wangenheim.

Dentre as aplicações, é possível citar o cálculo de anagramas entre duas palavras. Uma palavra é anagrama da outra se a sequência de letras de uma for permutação da sequência de letras da outra. Ao ordenar as duas palavras, podemos verificar se isso é verdade ou não. O cálculo da quantidade de números ou palavras distintas de uma lista também pode ser feito utilizando-se a ordenação. Neste caso, ordenamos o vetor em ordem crescente e incrementamos o contador apenas quando o elemento atual for diferente do anterior. A mediana de uma lista de números também pode ser calculada utilizando-se algoritmos de ordenação. Nesse caso, ordenamos o vetor de forma crescente ou decrescente e, após, em tempo constante, acessamos a posição do meio

do vetor (no caso de o número de elementos ser ímpar) ou as duas posições centrais caso o número de elementos seja par.

Referências

EDELWEISS, N.; GALANTE, R. *Estruturas de dados*. Porto Alegre: Bookman, 2009.

ZIVIANI, N. *Projeto de algoritmos*: com implementações em Pascal e C. 2. ed. São Paulo: Thomson, 2004.

Leituras recomendadas

BATTISTELLA, P. E.; WANGENHEIM, A.; WANGENHEIM, C. G. SORTIA: um jogo para ensino de algoritmo de ordenação: estudo de caso na disciplina de estrutura de dados. *In*: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 23., 2012, Rio de Janeiro. *Anais* [...]. Rio de Janeiro: UFRJ, 2012.

ROSEN, K. H. *Matemática discreta e suas aplicações*. 6. ed. Porto Alegre: AMGH, 2010.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS