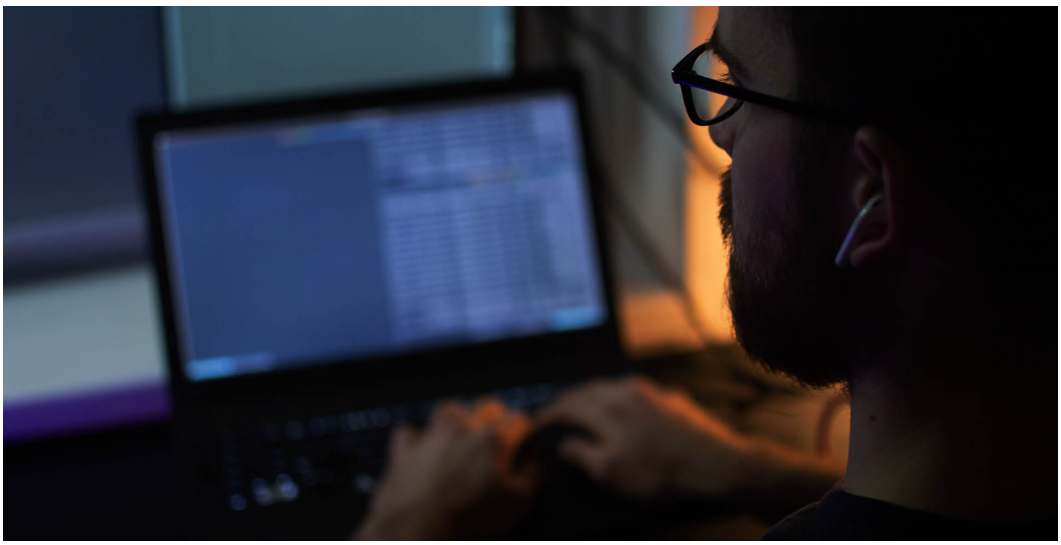


Algoritmos de ordenação avançados em Python

Prof. Edgar Augusto Gonçalves Gurgel do Amaral

Prof. Kleber de Aguiar

Descrição

Apresentação dos algoritmos de ordenação avançados (*merge sort, quick sort* e *shell sort*) e discussão sobre suas complexidades.

Propósito

Entender o funcionamento dos algoritmos de ordenação básicos e avançados é essencial para que os profissionais de tecnologia da informação (TI) escolham corretamente o algoritmo mais adequado à solução do problema a ser tratado.

Objetivos

Módulo 1

Algoritmos de ordenação elementar

Reconhecer os algoritmos de ordenação elementar.

Módulo 2

Algoritmo de ordenação por intercalação

Descrever o funcionamento do algoritmo de ordenação por intercalação (*merge sort*).

Módulo 3

Algoritmo de ordenação rápida

Descrever o funcionamento do algoritmo de ordenação rápida (*quick sort*).

Módulo 4

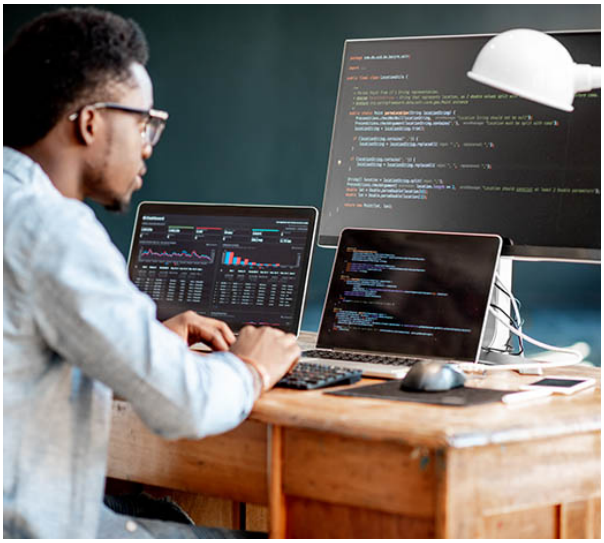
Algoritmo de ordenação shell sort

Descrever o funcionamento do algoritmo de ordenação *shell sort*.

Introdução

No mundo da computação, operações essenciais - e muito analisadas - são a ordenação e a pesquisa por determinado elemento em um conjunto de valores. Essas tarefas são usadas em muitos programas, além de compiladores, interpretadores, banco de dados e sistemas operacionais.

Neste conteúdo, apresentaremos os fundamentos dos principais métodos de ordenação e analisaremos sua complexidade.



1 - Algoritmos de ordenação elementar

Ao final deste módulo, você será capaz de reconhecer os algoritmos de ordenação elementar.

Ordenação

Processo de ordenação

Ordenação corresponde ao método de organizar um conjunto de objetos em uma ordem (ascendente ou descendente). Atividade relevante e fundamental na área de computação, ela tem como objetivo facilitar a recuperação dos itens do conjunto, como, por exemplo, a recuperação de nomes em uma lista telefônica.

> **O ato de colocar um conjunto de dados em determinada ordem predefinida, estando os elementos ordenados conforme o critério definido, permitirá que o acesso aos dados seja feito de forma mais eficiente.**

A ordenação de um conjunto de dados é feita utilizando como base uma chave chamada de **chave de ordenação**, que é o campo do item utilizado para comparação. É por meio dela que sabemos se determinado elemento está à frente ou não de outros no conjunto de dados.

Para realizar a ordenação, podemos usar qualquer tipo de chave, desde que exista uma regra de ordenação bem definida. Existem vários tipos possíveis de ordenação.

Os mais comuns são:

Ordenação numérica

1, 2, 3, 4, 5.

Ordenação lexicográfica (ordem alfabética)

Ana, Bianca, Michele, Ricardo.

Tipos de algoritmos de ordenação

Um **algoritmo de ordenação** é aquele que coloca os elementos de dada sequência em certa ordem predefinida. Há vários algoritmos para realizar a ordenação dos dados.

Eles podem ser classificados como:

Algoritmos de ordenação interna (IN-PLACE)

O conjunto de dados é pequeno e cabe todo na memória principal. Todo o processo de ordenação é realizado internamente na memória principal. Os dados a serem ordenados estão em um vetor ou em uma tabela (etapa na qual a ordenação é feita por um campo denominado chave, que identifica cada elemento do vetor que forma a tabela).

Algoritmos de ordenação externa

O conjunto de dados não cabe completamente em memória principal. Os dados a serem ordenados estão em algum disco de armazenamento ou fita, em arquivos no formato de registros, que são acessados sequencialmente ou em blocos.

Mais especificamente, há diversos tipos de algoritmos de ordenação. Entre eles, podemos destacar:

Ordenação por inserção

Inserção direta (*insertion sort*);
Incrementos decrescentes.

Ordenação por troca

Bolha (*bubble sort*);
Troca e partição (*quick sort*).

Ordenação por seleção

Seleção direta (*selection sort*);
Seleção em árvore (*heap sort*).

Vamos abordar a seguir alguns dos principais algoritmos básicos de ordenação de dados armazenados em *arrays* (ordenação interna).

Bubble sort

Também conhecido como ordenação por "bolhas", o *bubble sort* é um dos algoritmos de ordenação mais conhecidos e um dos mais simples. Ele funciona da seguinte forma:

Etapa 1

Em cada etapa, cada elemento será comparado com o próximo, havendo uma troca se ele não estiver na ordem correta.

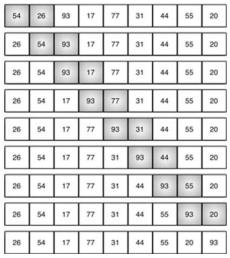
Etapa 2

A comparação é novamente realizada até que as trocas não sejam mais necessárias.

Para entendermos melhor como esse algoritmo é utilizado, apresentaremos um caso adiante.

Primeiramente, vamos colocar os elementos representados em um vetor, com Item[1] mais à esquerda e Item[n] mais à direita.

Em cada passo, o maior elemento é deslocado para a direita até encontrar um elemento maior ainda, como mostra a imagem a seguir:



Funcionamento *bubble sort* – primeira passagem.

Aqui é importante observar que o maior valor na lista está em alguma comparação. Vejamos alguns pontos:

- Esse valor é continuamente empurrado até o fim da passagem;
- Inicialmente, o valor 54 é empurrado;
- Depois, o valor 93 é empurrado até o final, e os valores menores vão subindo como bolhas;
- Os itens sombreados são aqueles comparados para se verificar se estão fora de ordem.

Após a primeira iteração, isto é, comparar dois a dois todos os elementos adjacentes do vetor, trocando-os de posição se for o caso. Deve-se repetir a iteração, considerando os elementos (Item[1], item[3],... Item[n-1]).

Comentário

Se imaginarmos que o vetor esteja na posição vertical, com o item[1] na parte inferior e o item[n] na superior, a cada comparação o maior elemento "subirá" de forma semelhante ao que ocorreria com uma bolha em um tubo com líquido.

O código em Python do algoritmo é o seguinte:

Exercício 1 TUTORIAL COPIAR

Python3

```
def bolha(vetor):
    for n in range(len(vetor)-1, 0, -1):
        trocou = False # para indicar se houve trocas

        for i in range(n):
```

null

null

O esforço computacional despendido pela ordenação de um vetor pode ser determinado pelo número de comparações, que também serve para estimar o número máximo de trocas possíveis de se realizar.

Na primeira passada, fazemos **n-1** comparações; na segunda, **n-2**; na terceira, **n-3**; e assim por diante. Logo, o tempo total gasto pelo algoritmo é proporcional a:

$$(n-1)+(n-2)+\ldots+2+1$$

Rotacione a tela.

A soma desses termos é proporcional ao quadrado de n. Portanto, o desempenho computacional desse algoritmo varia de forma quadrática em relação ao tamanho do problema.

> **Denotamos por *n* o tamanho da instância do problema, assim, a função matemática utilizada para expressar a complexidade computacional do algoritmo é escrita em função de n, isto é, em função do tamanho da instância do problema.**

Em geral, usamos a notação **Big-O** para expressar como a complexidade de um algoritmo varia com o tamanho do problema.

Assim, nesse caso, em que o tempo computacional varia de forma quadrática com o tamanho do problema, trata-se de um algoritmo de **ordem quadrática**. Expressamos isso escrevendo **O(n²)**.

Atenção!

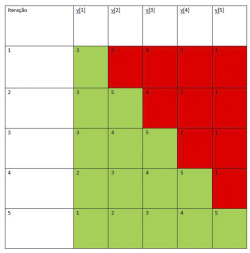
A análise de complexidade é feita com base no pior caso sempre! Em alguns algoritmos, os melhores casos, podem apresentar desempenho muito elevado. Este é o caso do método da bolha que, quando recebe o vetor já ordenado, fornece o resultado em uma única iteração, isto é: n-1 comparações. Entretanto, isto não é levado em consideração para fins de análise de complexidade computacional.

Insertion sort (inserção)

Também conhecido como **ordenação por inserção**, o *insertion sort* é outro algoritmo de ordenação bastante simples. Ele tem esse nome por se assemelhar ao processo de ordenação de um conjunto de cartas de baralho com as mãos: pega-se uma carta de cada vez e a insere em seu devido lugar, sempre deixando as cartas da mão em ordem.

A ideia do algoritmo é bastante simples. Dividimos o vetor a ser ordenado em duas partições: a primeira contendo os elementos já ordenados e a segunda com os elementos ainda por ordenar. Assim, na primeira iteração a partição já ordenada é composta pelo primeiro elemento do vetor e a segunda pelos restantes.

Note que um vetor unitário sempre está ordenado. A figura abaixo mostra como funciona o algoritmo, em verde os elementos já ordenados e vermelho por ordenar:



Na iteração 1, vamos inserir o elemento V[2] = 5, na porção ordenada do vetor, 5 > 3, assim, 5 já está na posição dele e podemos estender a porção ordenada até a posição 2. Em seguida, no passo 2, analisamos V[3] = 4, observamos que 4 é menor que 5, ultimo elemento da porção ordenada. Assim inserimos 4 na posição correta e temos agora um vetor ordenado com três posições. Analogamente, repetimos para v[4] e v[5]. Obtendo o vetor ordenado.

O algoritmo *insertion sort* funciona da seguinte forma:

Etapa 1

Os elementos são divididos em uma sequência de destino (a1, ..., ai-1) e em uma sequência fonte (ai, ..., an).

Etapa 2

Em cada passo, a partir de i = 2, o i-ésimo item da sequência fonte é retirado e transferido para a sequência destino, fase em que ele é é inserido na posição adequada.

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

O código que implementa o algoritmo de inserção é o seguinte:

Exercício 1 TUTORIAL COPIAR

Python3

```
def insercao(vetor):
    for i in range(1, len(vetor)):
        valor_atual = vetor[i]# guarda o valor da posição'i'
        j = i - 1 # guarda a posição anterior à posição 'i'
```

null

null

Assim como no método da bolha, o número de comparações que ocorre durante a ordenação por inserção depende de como a lista está inicialmente ordenada. Se a lista estiver em ordem, o número de comparações será n-1. Se estiver fora de ordem, ele será 1/2(n² + n).

O número de troca para cada caso é o seguinte:

Melhor

2 (n – 1)

Médio

1/4(n² – n)

Pior

$1/2(n^2 + n)$

Portanto, para o pior caso, a ordenação por inserção é tão ruim quanto a ordenação bolha e a ordenação por seleção (que veremos a seguir), ou seja, sua complexidade é $O(n^2)$. Para o caso médio, ela é somente um pouco melhor.

No entanto, a ordenação por inserção tem duas vantagens:

Primeira

Ela se comporta naturalmente, isto é, trabalha menos, quando a matriz já está ordenada, e o máximo, quando ela está ordenada no sentido inverso. Isso torna a ordenação excelente para listas quase em ordem.

Segunda

Ela não rearranja elementos de mesma chave. Isso significa que uma lista ordenada por duas chaves permanece ordenada para ambas após uma ordenação por inserção.

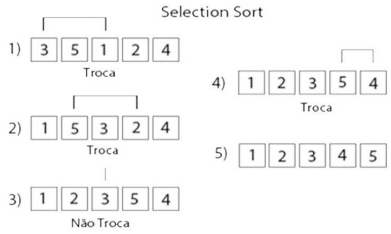
Selection sort (seleção)

Também conhecido como ordenação por seleção, o *selection sort* é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois, a cada passo, seleciona o melhor elemento (maior ou menor, dependendo do tipo de ordenação) para ocupar a primeira posição da porção não ordenada do vetor.

Comentário

Na prática, esse algoritmo possui um desempenho quase sempre superior na comparação com o *bubble sort*.

Veja a seguir a imagem que ilustra seu funcionamento:



Funcionamento do *selection sort*.

O algoritmo *selection sort* funciona da seguinte forma:

Etapa 1

Selecione o elemento com a chave de **menor** valor.

Etapa 2

Troque-o com o primeiro elemento da sequência.

Etapa 3

Repita essas operações com os n-1 elementos restantes; depois, com os n-2 elementos; e assim sucessivamente, até restar um só elemento (o maior deles), conforme mostra a imagem anterior.

O algoritmo divide o *array* em duas partes:

A parte ordenada

À esquerda do elemento analisado.

A parte ainda não ordenada

À direita do elemento analisado.

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*. Esse processo é feito até que todo o *array* esteja ordenado.

O código que implementa o algoritmo de seleção é o seguinte:

Exercício 1 TUTORIAL COPIAR

Python3

```python
def selecao(vetor):
    for j in range(len(vetor)):
        ind_min = j # guarda o índice 'j'

        for i in range(j + 1, len(vetor)):
```

null

null

O tempo de execução do selection sort é O(n²) para o melhor e o pior caso. Independentemente do vetor de entrada, o algoritmo se comportará da mesma maneira.

Lembre-se de que a notação indica que o tempo de execução do algoritmo é limitado superior e inferiormente pela função n². Ele é ineficiente para grandes conjuntos de dados.

A complexidade dos algoritmos de ordenação elementar

Assista neste vídeo à apresentação dos três algoritmos de ordenação elementar, demonstrando a complexidade de cada um deles.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Algoritmos como *bubble sort* e *selection sort* têm como finalidade:

A a verificação da integridade de vetores.

B o armazenamento de vetores.

C a ordenação de vetores.

D a recuperação de vetores.

E	a exclusão de vetores.

Parabéns! A alternativa C está correta.

Os dois algoritmos (*bubble sort* e *selection sort*) são voltados para a ordenação de estruturas de vetores de acordo com o critério que foi definido. As duas formas mais comuns são a ordenação lexicográfica e a numérica.

Questão 2

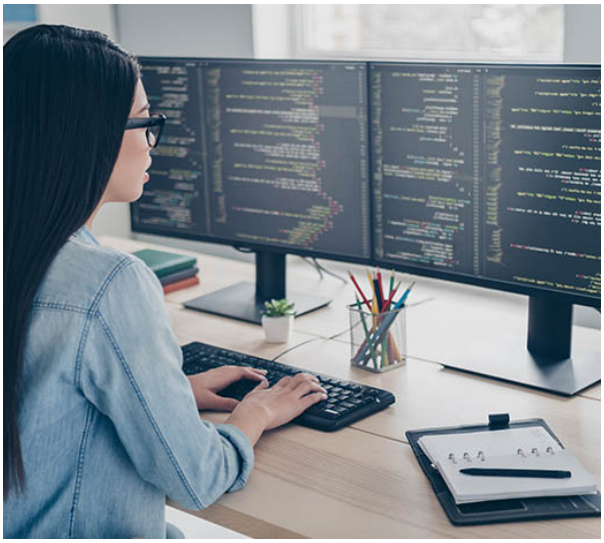
O algoritmo a seguir, descrito em pseudocódigo, pode ser utilizado para ordenar um vetor V[1..n] em ordem crescente:

Esse algoritmo é conhecido como:

A	*Insertion sort*
B	*Selection sort*
C	*Merge sort*
D	*Quick sort*
E	*Bubble sort*

Parabéns! A alternativa E está correta.

O *bubble sort* realiza múltiplas passagens por uma lista. Ele compara itens adjacentes e troca aqueles fora de ordem. Cada passagem pela lista coloca o próximo maior valor em sua posição correta. Em essência, cada item se desloca como uma “bolha” para a posição à qual pertence. Se existem n itens na lista, então há −1 pares de itens que precisam ser comparados na primeira passagem.



2 - Algoritmo de ordenação por intercalação

Ao final deste módulo, você será capaz de descrever o funcionamento do algoritmo de ordenação por intercalação (*merge sort*).

Algoritmo de ordenação por intercalação (*merge sort*)

Definição

Também conhecido como ordenação por intercalação, o *merge sort* é um algoritmo recursivo que usa a ideia de dividir para conquistar a fim de ordenar os dados de um *array*.

Esse algoritmo defende que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos dados. Ele os divide em conjuntos cada vez menores para, em seguida, ordená-los e combiná-los por meio de intercalação (*merge*).

Características

Nessa forma de ordenação, o vetor é dividido em vetores com a metade do tamanho do original por meio de um procedimento recursivo. A divisão ocorre até que o vetor fique com apenas um elemento e que tais elementos estejam ordenados e intercalados.

A técnica recursiva de dividir para conquistar, aplicada ao algoritmo de *merge sort*, é formada por três etapas:

Dividir

O problema em certo número de subproblemas: dividir a sequência de n elementos a serem ordenados em duas subsequências de n/2 elementos.

Conquistar

Os subproblemas solucionados recursivamente (se os tamanhos dos subproblemas são suficientemente pequenos, então solucionam-se os subproblemas de forma simples): ordenar duas subsequências recursivamente, utilizando a ordenação por intercalação.

Combinar

As soluções dos subproblemas na solução de problema original: intercalar as duas subsequências ordenadas por intercalação.

Esse algoritmo apresenta algumas vantagens, como:

- Requerer menos acesso à memória;
- Ser ótimo candidato para o emprego da programação paralela – caso exista mais de um processador, certamente ele terá um bom desempenho.

Execução

A execução do algoritmo *merge sort* pode ser representada com o uso de uma árvore binária da seguinte forma:

Nó

Representa uma chamada recursiva do *merge sort*.

Nó raiz

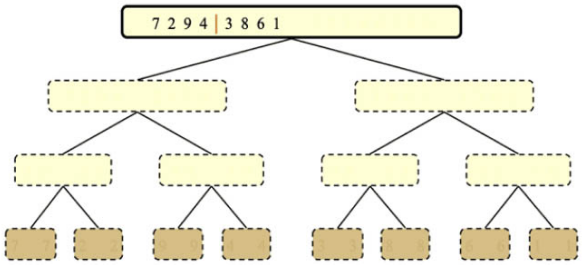
É a chamada inicial.

Nós folhas

São vetores de 1 ou 2 números, que são os casos base.

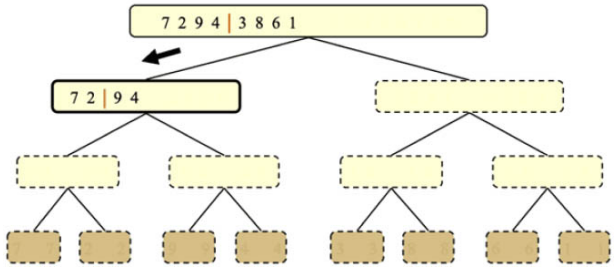
Agora vamos mostrar um exemplo de execução do *merge sort* para ordenar o vetor: 7 2 9 4 3 8 6 1.

Vejamos cada passo da ordenação a seguir:



Partição do vetor inicial.

O primeiro passo é a partição do problema (sempre no meio do vetor). Dessa forma, o vetor inicial é particionado em duas partes (7 2 9 4 e 3 8 6 1).

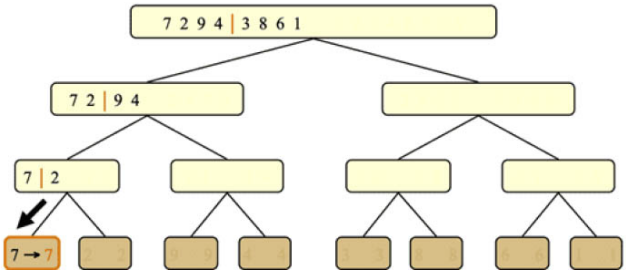


Partição do vetor inicial.

O segundo passo é uma chamada recursiva para a primeira partição do vetor 7 2 9 4, que será particionada em duas partes (7 2 e 9 4).

Partição do vetor inicial.

O terceito passo é uma nova chamada recursiva ser feita para essa primeira partição 7 2, de tal forma que vamos particioná-la em duas partes (7 e 2) por meio da chamada recursiva.



Partição do vetor inicial.

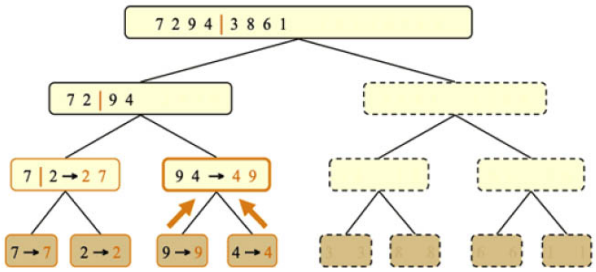
O quarto passo é fazer duas chamadas recursivas para o caso base encontrado (7 e 2). A primeira chamada recursiva é representada na imagem acima.

Partição do vetor inicial.

O quinto passo é fazer a segunda chamada recursiva para o caso base (7 e 2).

Partição do vetor inicial.

O sexto passo é realizar uma operação de intercalação para ordenar o caso base (7 e 2).



Partição do vetor inicial.

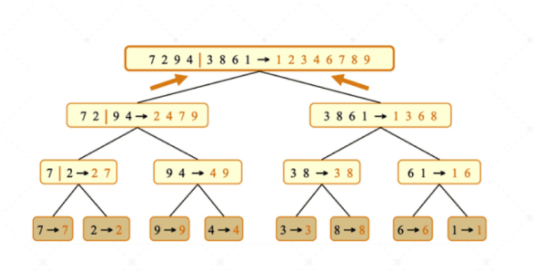
O sétimo passo é particionar em duas partes (9 e 4) as chamadas recursivas para a partição 9 4; depois, as duas chamadas recursivas para o caso base; e, por fim, a intercalação para o caso base 9 e 4.

Partição do vetor inicial.

O próximo passo é realizar uma intercalação (*merge*) para ordenar a primeira partição do vetor.

Partição do vetor inicial.

O passo seguinte é executar os mesmos procedimentos para ordenar a segunda partição do vetor, que ficará da seguinte forma mostrada na imagem acima.



Partição do vetor inicial.

O passo final é servir o último *merge* para ordenar as duas partições do vetor. Dessa forma, teremos as duas partições (2 4 7 9 e 1 3 6 8) ordenadas.

Implementação

O algoritmo *merge sort* será implementado pelas funções *merge* e *mergeSort* e pela função principal. Primeiramente, vamos apresentar a função principal (*main*), que é composta apenas por uma chamada à função *mergeSort* no box a seguir:

PYTHON

```python
def main(vetor):
    mergeSort(vetor, 0, len(vetor) - 1)
    print ("Vetor ordenado:", vetor)
```

Agora vamos apresentar suas funções.

Dado um vetor de inteiros **vetor** e dois inteiros *left* >= 0 e *right* >= 0, a função *mergeSort* ordena as duas partes do vetor e as mescla em seguida por intermédio de uma chamada à função *merge*:

PYTHON

```python
def mergeSort(vetor, ini, fin):
    if ini >= fin:
        return

    meio = (ini + fin) // 2
    mergeSort(vetor, ini, meio)
    mergeSort(vetor, meio + 1, fin)
    merge(vetor, ini, meio, fin)
```

Dado um **vetor** de inteiros vetor e três inteiros **ini, meio** e **fin**, colocam-se os elementos ordenados em ordem crescente em **vetor[ini...fin]**:

PYTHON

```python
def merge(vetor, ini, meio, fin):
    # separa 'vetor' nos 2 subvetores que serão mesclados
esq_vetor = vetor[ini:meio + 1] #subvetor esquerdo
    dir_vetor = vetor[meio + 1: fin + 1] #subvetor direito
    # variáveis para controlar a iteração entre os subvetores
    i = 0 # índice do subvetor esquerdo
    j = 0 #  índice do subvetor direito
    k = ini # índice para a parte ordenada do vetor
    # percorrer ambos os subvetores até que um deles fique sem elementos
    while i < len(esq_vetor) and j < len(dir_vetor):
        # se o subvetor esquerdo tiver o menor elemento ele será colocado
        # na parte ordenada e o índice do subvetor esquerdo será incrementado
        if esq_vetor[i] <= dir_vetor[j]:
            vetor[k] = esq_vetor[i]
```

Veja a seguir o programa completo com a implementação do algoritmo *merge sort* em Python. Aproveite e teste-o mais vezes, mudando os elementos do vetor **vet** (parâmetro da função *main*)."

Exercício 1 TUTORIAL COPIAR

Python3

```python
def merge(vetor, ini, meio, fin):
    # separa 'vetor' nos 2 subvetores que serão mesclados
    esq_vetor = vetor[ini:meio + 1] # subvetor esquerdo
    dir_vetor = vetor[meio + 1: fin + 1] # subvetor direito
    # variáveis para controlar a iteração entre os subvetores
```

null

null

Análise de complexidade

O trecho desse pseudocódigo relevante do algoritmo *merge sort* é o seguinte:

PSEUDOCÓDIGO

```
1  função merge (x, inicio, fim)
2  início
3   var meio: numérico;
4   Se (inicio < fim) então
5    inicio
6      meio ← parteinteira (incio + fim) / 2;
7      merge (x, inicio, meio);
8      merge (x, meio+1, fim);
9      intercala (x, inicio, fim, meio);
10    fim;
11   fim-se;
12  fim-função-merge.
```

Para o cálculo do tempo de execução do *merge sort*, é preciso calcular inicialmente o tempo de execução da função intercala. O algoritmo da função intercala, que realiza a intercalação de dois vetores, cujos tamanhos suponhamos que sejam m_1 **e** m_2, respectivamente, faz a varredura de todas as posições dos dois vetores, gastando, com isso, um tempo $\mathrm{n} = \mathrm{m}_1 + \mathrm{m}_2$.

Analisando o trecho de código anterior, verificamos que a função *merge* possui três chamadas de função:

As duas primeiras

Chamadas de recursivas, elas recebem a metade dos elementos do vetor passado.

A última

É a chamada para função que realiza a intercalação das duas metades.

A linha de comparação Se e a linha da atribuição gastam um tempo constante O(1). Para calcularmos o tempo de execução de um programa recursivo, inicialmente teremos de obter a expressão de recorrência da função *merge sort*, que é dada por:

$$T(n) = 2T(n/2) + n$$

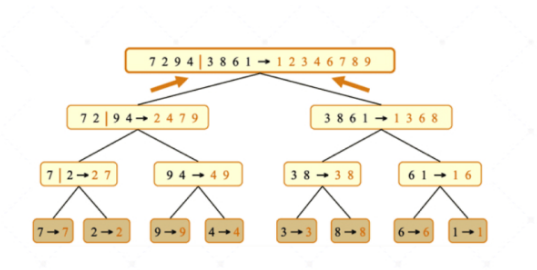
Em que:

- **2T (n/2)** = duas chamadas recursivas;
- **n** = tempo gasto com a intercalação das duas metades do vetor.

O tempo gasto nas demais linhas da função **(O(1))**, ou seja, o tempo constante, é menor que o gasto pela função intercala. Por isso, somamos apenas **n** à expressão de recorrência.

A solução da equação recorrente $T(n) = 2T(n/2) + n$ fornece, sem dúvida, a complexidade computacional do algoritmo. Porém existe uma forma mais intuitiva de chegar ao resultado.

Conforme vimos, as divisões sucessivas do vetor produzem um estrutura de árvore binária. A figura abaixo, ilustra as divisões.



Ao chegar o último nível, temos os problemas infantis, isto é, aqueles de solução trivial, vetores unitários que sempre estão ordenados.

Observe que para obter o penúltimo nível temos que intercalar propriamente todos os elementos do vetor e, como vimos, esta operação é feita em ordem de *n*. Ou seja, a complexidade do algoritmo é *kn*, onde *k* é a quantidade de níveis da árvore.

Como determinar *k*? A cada divisão temos que o pedaço analisado é metade do anterior. Assim, no nível 1 temos $\frac{n}{2^0} = n$, no nível 2 temos $\frac{n}{2^1}$ elementos, no último nível, que é o nível *k*, temos $\frac{2}{2^{k-1}} = 1$, uma vez que no último nível os vetores são unitários. Assim basta resolve a equação $2^{k-1} = n$, para descobrir o número de níveis da árvore. Aplicando log na base 2 em ambos os lados da equação temos $\log_2 2^{k-1} = \log_2 n$, ou seja, $k - 1 = \log_1 n$, logo $k = \log_2 n + 1$.

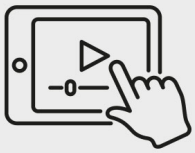
Como a complexidade do algoritmo é *kn*, então $f(n) = (\log_2 n + 1)n$, que é $O(n \log n)$.

No *merge sort*, independentemente do vetor de entrada, o algoritmo trabalhará da mesma maneira, dividindo o vetor ao meio, ordenando cada metade recursivamente e intercalando as duas metades ordenadas.

Algoritmo de ordenação *merge sort*

Veja neste vídeo a apresentação, com exemplos práticos, do algoritmo de ordenação *merge sort*, demonstrando a análise de complexidade.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

As estratégias de divisão e conquista são utilizadas pelo seguinte algoritmo de ordenação:

A *Selection sort*

B *Insertion sort*

C *Bubble sort*

D *Shell sort*

E *Merge sort*

Parabéns! A alternativa E está correta.

A ideia básica do *merge sort* consiste em dividir o problema em vários subproblemas e resolvê-los por meio da recursividade, bem como conquistar, ou seja, unir as resoluções dos subproblemas após todos terem sido resolvidos.

Questão 2

Uma fábrica de *software* foi contratada para desenvolver um produto de análise de riscos. Em determinada funcionalidade desse *software*, é necessário realizar a ordenação de um conjunto formado por muitos números inteiros. Que algoritmo de ordenação oferece a melhor complexidade de tempo (notação O) no pior caso?

A *Merge sort*

B *Insertion sort*

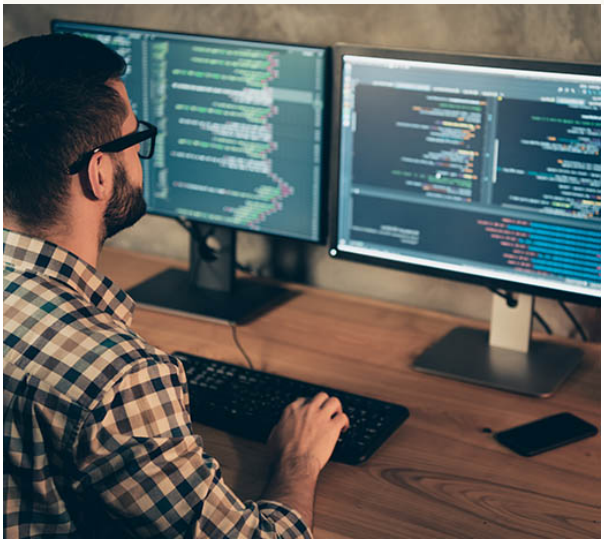
C *Bubble sort*

D *Half sort*

E *Selection sort*

Parabéns! A alternativa A está correta.

Para o *merge sort*, não importa a maneira como os elementos estão organizados no vetor. Sua complexidade será sempre a mesma. Esse algoritmo é ideal para aplicações que precisam de ordenação eficiente, que não toleram desempenho ruim no pior caso e que possuem espaço de memória extradisponível.



3 - Algoritmo de ordenação rápida

Ao final deste módulo, você será capaz de descrever o funcionamento do algoritmo de ordenação rápida (*quick sort*).

Algoritmo de ordenação rápida (*quick sort*)

Definição

O algoritmo de ordenação *quick sort* foi proposto em 1960 pelo cientista da computação britânico Charles Antony Richard Hoare, embora só tenha sido publicado dois anos depois.

Ele se caracteriza por dividir o vetor em duas partes utilizando um procedimento recursivo. Essa divisão se repetirá até que o vetor tenha apenas um elemento, enquanto os demais ficam ordenados ao passo que ocorre o particionamento.

Trata-se de um algoritmo que também se baseia na técnica de divisão e conquista. Tal técnica é aplicada da seguinte forma:

Dividir

O vetor X[p..r] é particionado (rearranjado) em dois subvetores não vazios X[p..q] e X[q+1..r], tais que cada elemento de X[p..q] é menor ou igual a cada elemento de X[q+1..r].O índice q é calculado como parte do processo de particionamento. Para encontrar esse índice, o elemento que está na metade do vetor original – chamado de pivô – é escolhido, e os elementos do vetor são reorganizados. Os que ficarem à esquerda de q são menores ou iguais ao pivô, e os que ficarem à direta de q são maiores ou iguais ao pivô.

Conquistar

Chamadas recursivas ao *quick sort* são usadas para ordenar os dois subvetores: X[p..q] e X[q+1..r].

Combinar

Durante o procedimento recursivo, os elementos são ordenados no próprio vetor. Nesta fase, nenhum processamento é necessário.

De forma geral, para o particionamento, o algoritmo pode ser descrito dessa maneira:

Etapa 1

Escolha arbitrariamente um pivô **x**.

Etapa 2

Percorra o vetor a partir da esquerda até que v[i] >= **x**.

Etapa 3

Percorra o vetor a partir da direita até que v[j] <= **x**.

Etapa 4

Percorra o vetor a partir da direita até que v[j] <= **x**. Troque **v[i]** com **v[j]**.

Etapa 5

Continue esse processo até os apontadores **i** e **j** se cruzarem.

Ao final do procedimento, o vetor X[p..r] estará dividido em dois segmentos:

Os itens em X[p..q], X[p+1], ..., X[i]

São menores ou iguais a **x**.

Os itens em X[q+1..r], X[q+1], ..., X[j]

São maiores ou iguais a **x**.

Método para particionamento

Considere o pivô como o elemento da posição final que queremos encontrar.

Dica

Utilizar o último elemento do vetor ou o primeiro elemento é só um mecanismo para facilitar a implementação.

Inicialize dois ponteiros, denominados alto e baixo, como o limite inferior e o superior do vetor a ser analisado. Em qualquer momento da execução, os elementos **acima de alto** serão **maiores que x**, enquanto os elementos **abaixo de baixo** serão **menores que x**.

Vamos mover os ponteiros alto e baixo um em direção ao outro de acordo com estes passos:

Etapa 1

Baixo deve ser incrementado em uma posição até que a[baixo] >= pivô.

Etapa 2

Alto tem de ser decrementado em uma posição até que a[alto] < pivô.

Etapa 3

Se alto > baixo, é preciso trocar as posições de a[baixo] por a[alto].

Esse procedimento tem de ser repetido até que a condição colocada no passo 3 falhe, ou seja, quando alto <= baixo. Assim, a[alto] é trocado pelo pivô, que é o objetivo da procura.

Execução

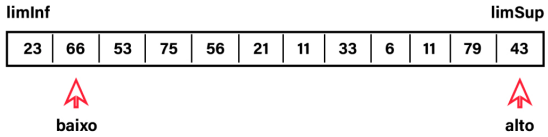
Para entendermos melhor o funcionamento do algoritmo *quick sort*, vamos simular a execução do vetor a seguir, considerando o limite superior e o inferior (pivô, alto e baixo). O objetivo é ordenar esse vetor.

O limInf será escolhido como o pivô. Inicialmente, o vetor será dividido em duas partes: a parte da esquerda terá os elementos menores que o pivô; a da direita, os maiores que o pivô.

Observe:

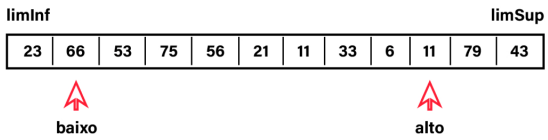
Simulação de execução do vetor.

Vamos incrementar baixo até que seja encontrado um valor maior que o pivô escolhido 23. Com isso, obtemos o valor 66, como mostra a figura a seguir:



Simulação de execução do vetor.

Agora vamos decrementar alto até que seja encontrado um valor menor que o pivô escolhido 23. Desse modo, encontramos o valor 11:

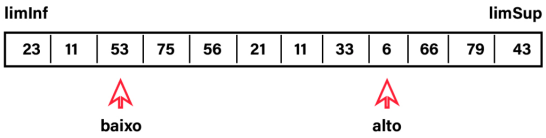


Simulação de execução do vetor.

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 66 e 11. Após a troca desses valores, vamos incrementar baixo até que um valor maior que o pivô 23 seja encontrado. Dessa forma, obtemos o valor 53:

Simulação de execução do vetor.

Agora vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Dessa maneira, encontramos o valor 6:



Simulação de execução do vetor.

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 53 e 6. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Com isso, obtemos o valor 75:

Simulação de execução do vetor.

Agora vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Desse modo, encontramos o valor 11:

Simulação de execução do vetor.

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 75 e 11. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Encontramos então o valor 56:

Simulação de execução do vetor.

Agora vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Com isso, obtemos o valor 21:

Simulação de execução do vetor.

Em seguida, vamos trocar os valores de a[alto] e a[baixo], ou seja, 56 e 21. Após a troca desses valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 23 seja encontrado. Desse modo, encontramos o valor 56 novamente:

Simulação de execução do vetor.

Agora vamos decrementar alto até que um valor menor que o pivô 23 seja encontrado. Com isso, encontramos o valor 21 novamente. Entretanto, veja que atingimos a condição de parada do algoritmo, pois alto 21 é menor que baixo 56:

Simulação de execução do vetor.

Em seguida, vamos trocar o pivô a[limInf]= 23 com a[alto] = 21. Dessa forma, a divisão do vetor está concluída.

Agora devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da seguinte forma (em que limSup = alto – 1 = 4):

Simulação de execução do vetor.

Em seguida, vamos incrementar baixo até que um valor maior ou igual ao pivô 21 seja encontrado. Dessa forma, encontramos o valor alto 11. Também decrementaremos alto até que um valor menor que o pivô 21 seja encontrado. Com isso, obtemos o valor baixo 11:

Simulação de execução do vetor.

Depois, vamos trocar o pivô a[limInf]= 21 com a[alto] = 11. Assim, a divisão do vetor está concluída.

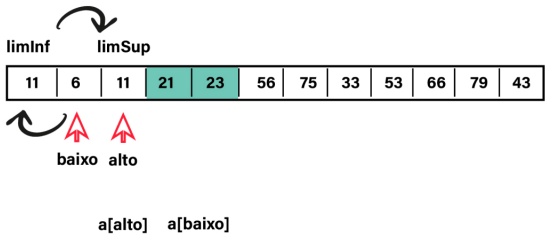
Agora devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da seguinte forma (em que limSup = alto – 1 = 3):

Simulação de execução do vetor.

Em seguida, vamos incrementar baixo até que um valor maior ou igual ao pivô 11 seja encontrado. Então encontramos o valor baixo 11. Também vamos decrementar alto até que um valor menor que o pivô 11 seja encontrado. Com isso, obtemos o valor alto 6:

Simulação de execução do vetor.

Depois, vamos trocar os valores de a[alto] e a[baixo], ou seja, 11 e 6:



Simulação de execução do vetor.

Após a troca dos valores, vamos incrementar baixo até que um valor maior ou igual ao pivô 11 seja encontrado. Desse modo, encontramos o valor 11:

Simulação de execução do vetor.

Agora vamos decrementar alto até que um valor menor que o pivô 11 seja encontrado. Então encontramos o valor 6. No entanto, veja que atingimos a condição de parada do algoritmo, pois alto 6 é menor que baixo 11:

Simulação de execução do vetor.

Em seguida, vamos trocar o pivô a[limInf]= 11 com a[alto] = 6. Assim, a divisão do vetor está concluída.

Agora devemos particionar o subvetor esquerdo e reorganizar os elementos para esse subvetor da forma mostrada a seguir (em que limSup = alto – 1 = 1). Entretanto, como LimSup > LimInf é falso, devemos parar, como aponta esta imagem:

Simulação de execução do vetor.

O mesmo procedimento deve ser executado para ordenar a segunda metade do vetor.

Implementação

Veja a seguir o algoritmo que implementa o *quick sort* com o desenvolvimento do procedimento partição:

Exercício 1 TUTORIAL COPIAR

Python3

```
def divide(vetor, inf, sup):
    # seleção do pivô
    pivo = vetor[inf]
    # particionamento
    esq = inf
    ...
```

null

null

Análise de complexidade

A ideia principal do algoritmo de ordenação quick sort é fazer o processo de particionamento do vetor em duas partes, que é realizado pelo procedimento partição. Nesse procedimento, o vetor é particionado na posição **j** de forma que todos os elementos do lado esquerdo de **j** são menores ou iguais ao elemento denominado pivô, enquanto todos do lado direito são maiores que ele.

Além disso, nesse procedimento, o tempo de execução é limitado pelo tamanho do vetor – no caso, **n**. Isso acontece porque, para realizar esse particionamento, o algoritmo compara os elementos da posição **i**, cujo valor se inicia na primeira posição e vai aumentando, e os da posição **j**, cujo valor tem início com a última posição e decresce, com o valor pivô.

Comentário

Em outros termos, o algoritmo comparará todos os elementos do vetor com o pivô enquanto os índices atenderem à condição `i < j`. Desse modo, o procedimento partição realizará `O(n)` comparações.

O pior caso ocorre quando:

- O particionamento produz uma região com n-1 elementos e outra com apenas um elemento;
- O tempo de execução, no pior caso, é Θ(n2) para n>= 1.

O melhor caso ocorre quando:

- o particionamento produz duas regiões de tamanho n/2;
- O tempo de execução, no melhor caso, é Θ(n.log n).

Algoritmo de ordenação *quick sort*

Veja uma apresentação, com exemplos práticos, do algoritmo de ordenação *quick sort*, demonstrando a análise de complexidade.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

As estratégias de divisão e conquista são utilizadas pelo seguinte algoritmo de ordenação:

A *Selection sort*

B *Insertion sort*

C *Bubble sort*

D *Half sort*

E	*Quick sort*

Parabéns! A alternativa E está correta.

O *quick sort* é um algoritmo que utiliza a estratégia de divisão e conquista para ordenar um vetor, ou seja, ele o particiona e vai realizando a ordenação de cada partição. Quando cada partição estiver ordenada, o vetor estará ordenado.

Questão 2

(Cesgranrio - Petrobras - analista de sistemas júnior - engenharia de *software* - 2011):

A ordenação é um problema muito importante para os desenvolvedores de *software*. Para implementá-la, existem vários algoritmos que já foram amplamente estudados, como o *bubble sort*, o *quick sort* e o *merge sort*. Uma das características estudadas desses algoritmos é o tempo de execução, que, usualmente, é medido pela notação O (Big-Oh).

I. O tempo de pior caso do algoritmo *quick sort* é de ordem menor que o tempo médio do algoritmo *bubble sort*.

II. O tempo médio do *quick sort* é O(nlog2 n), pois ele usa como estrutura básica uma árvore de prioridades.

III. O tempo médio do *quick sort* é de ordem igual ao tempo médio do *merge sort*.

Está correto apenas o que se afirma em:

A	I
B	II
C	III
D	I e III
E	II e III

Parabéns! A alternativa C está correta.

A complexidade de pior caso do *quick sort* é O(n2), e o tempo médio do *bubble sort* é O(n2). O tempo médio do *quick sort* é O(nlog2 n), mas sua estrutura básica é uma lista ou um vetor. O tempo médio do *quick sort* é O(n log2 n), assim como o tempo médio do *merge sort*.

4 - Algoritmo de ordenação *shell sort*

Ao final deste módulo, você será capaz de descrever o funcionamento do algoritmo de ordenação *shell sort*.

Algoritmo de ordenação *shell sort*

Definição

A ordenação *shell* é assim chamada devido a seu inventor: o cientista da computação norte-americano Donald Shell. Mas provavelmente seu nome vingou, porque seu método de operação é descrito, com frequência, como conchas do mar empilhadas umas sobre as outras.

Características

Derivado da ordenação por inserção, seu método geral é baseado na diminuição dos incrementos.

Veja esta imagem:

Ordenação *shell*.

Observe a seguir a descrição de cada passo:

- Primeiramente, todos os elementos afastados em três posições uns dos outros são ordenados;
- Em seguida, todos os elementos afastados em duas posições são ordenados;
- Finalmente, todos os elementos adjacentes são ordenados;
- Assim, temos como resultado: a b c d e f.

Não é fácil perceber que esse método conduz a bons resultados ou mesmo que ordene o vetor, mas ele executa ambas as funções. Desse modo, a ordenação *shell* tem as seguintes características:

Poucos elementos

Cada passo da ordenação envolve relativamente poucos elementos ou aqueles que já estão razoavelmente em ordem.

Ordenação dos dados

A ordenação shell é eficiente, e cada passo aumenta a ordenação dos dados.

A sequência exata para os incrementos pode mudar. A única regra é que o último incremento deve ser 1. Por exemplo, a sequência 9 5 3 2 1 funciona bem e é usada na ordenação *shell* mostrada.

Atenção!

Evite sequências que são potências de 2, pois, por razões matemáticas complexas, elas reduzem a eficiência do algoritmo de ordenação, embora a ordenação ainda funcione.

Implementação

O algoritmo que implementa a ordenação shell sort é o seguinte:

Exercício 1 TUTORIAL COPIAR

Python3

```
def shellSort(vetor):
    # iniciando com incremento 'incr' razoavelmente grande
    # o 'incr' será decrementado dentro do 'while'
    incr = len(vetor) // 2
    # inserindo elementos de acordo com o incremento
```

null

null

Observe os seguintes pontos:

- O laço **while** mais interno tem duas condições de teste;
- A comparação **vetor[j – incr] > temp** é obviamente necessária para o processo de ordenação;
- O teste **j >= incr**, sendo que o valor de **incr** nunca será menor do que zero, pois a condição de parada do laço **while** mais externo é **incr > 0**, evitando que os limites do **vetor** sejam ultrapassados.

Essas verificações extras diminuirão até certo ponto o desempenho da ordenação *shell*. Versões um pouco diferentes desse tipo de ordenação empregam elementos especiais de matriz. Chamados de **sentinelas**, eles não fazem parte realmente da matriz ordenada.

Sentinelas guardam valores especiais de terminação que indicam o menor e o maior elemento possível. Dessa forma, as verificações dos limites são desnecessárias. No entanto, usar sentinelas requer um conhecimento específico dos dados, o que limita a generalização da função de ordenação.

Análise de complexidade

A análise de ordenação do *shell* apresenta alguns problemas matemáticos. O tempo de execução é proporcional a $n^{1.2}$ para se ordenar n elementos.

Trata-se de uma redução significativa em relação às ordenações n-quadrado.

Para entender o quanto essa ordenação é melhor, veja os gráficos das ordenações n² e $n^{1.2}$:

Gráfico: Curvas n² e $n^{1.2}$.

Comparação entre algoritmos

Agora vamos analisar, de forma comparativa, a complexidade dos algoritmos apresentados.

> **A eficiência do algoritmo *bubble sort* diminui drasticamente à medida que o número de elementos no *array* aumenta. Esse algoritmo não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade.**

Considerando um *array* com N elementos, o tempo de execução do *bubble sort* é:

O(n)

Melhor caso – os elementos estão ordenados.

O(n)

Pior caso – os elementos estão ordenados na ordem inversa.

O(n²)

Caso médio.

Assim como o *bubble sort*, o algoritmo *selection sort* não é eficiente. Sua eficiência diminui drasticamente à medida que o número de elementos no *array* aumenta. Por isso, esse algoritmo também não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade.

Considerando um *array* com N elementos, o tempo de execução do *selection sort* é sempre da ordem de $O(n^2)$. Como podemos notar, sua eficiência não depende da ordem inicial dos elementos.

Diante de um *array* com N elementos, o tempo de execução do *insertion sort* é:

O(n)

Melhor caso – os elementos já estão ordenados.

$O(n^2)$

Pior caso – os elementos estão ordenados na ordem inversa.

$O(n^2)$

Caso médio.

Agora, levando em conta um *array* de N elementos, o tempo de execução do *merge sort* é sempre de ordem O(N log N). Como podemos observar, a eficiência do *merge sort* não depende da ordem inicial dos elementos.

Embora a eficiência do *merge sort* seja a mesma independentemente da ordem dos elementos, esse algoritmo possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação. Isso ocorre porque ele cria uma cópia do *array* para cada chamada recursiva.

Em outra abordagem, é possível utilizar um único *array* auxiliar ao longo de toda a execução do *merge sort*. Observe a comparação a seguir:

Pior caso

O *merge sort* realiza cerca de 39% menos comparações que o *quick sort* em seu caso médio.

×

Melhor caso

O *merge sort* realiza cerca de metade do número de iterações de seu pior caso.

O tempo de execução do *quick sort* depende de o particionamento ser ou não balanceado, o que, por sua vez, depende de quais elementos são utilizados para o particionamento.

Se o particionamento é balanceado, o algoritmo executa tão rapidamente quanto o *merge sort*. Porém, se ele não o é, o algoritmo *quick sort* é tão lento quanto o *insertion sort*.

Comentário

O *shell sort* possui a vantagem de ser uma ótima opção para arquivos de tamanho moderado. Sua implementação é simples e requer uma quantidade de código pequena. Entretanto, como desvantagem, o tempo de execução do algoritmo é sensível à ordem inicial do arquivo; além disso, o método não é estável.

A tabela a seguir apresenta resumidamente uma comparação entre os algoritmos de ordenação:

Algoritmo	Comparações			Movimentações	
	Melhor	Médio	Pior	Melhor	Médio
Bubble	$O(n^2)$			$O(n^2)$	
Selection	$O(n^2)$			$O(n)$	
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$
Merge	$O(n \log n)$			-	
Quick	$O(n \log n)$		$O(n^2)$	-	
Shell	$O(n^{1.25})$ ou $O(n (\ln n)^2)$			-	

Tabela: Resumo comparativo dos algoritmos de ordenação.
Edgar Augusto Gonçalves Gurgel do Amaral.

Algoritmo de ordenação *shell sort*

Observe neste vídeo uma apresentação, com exemplos práticos, do algoritmo de ordenação *shell sort*, demonstrando a análise de complexidade.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Entre os métodos de ordenação a seguir, qual deles é derivado da ordenação por inserção e baseado na diminuição dos incrementos?

A *Quick sort*

B *Bubble sort*

C *Selection sort*

D *Merge sort*

E *Shell sort*

Parabéns! A alternativa E está correta.

Adaptação do algoritmo *insertion sort*, o *shell sort* é baseado na diminuição do número de incrementos para a comparação dos elementos do vetor.

Questão 2

Uma ótima opção para arquivos de tamanho moderado, cuja implementação é simples, é o algoritmo de ordenação:

A *Shell sort*

B *Bubble sort*

C *Selection sort*

D *Merge sort*

E *Quick sort*

Parabéns! A alternativa A está correta.

O *shell sort* é bom para ordenar um número moderado de elementos, pois requer uma quantidade de código pequena.

Considerações finais

Neste conteúdo, percorremos os diversos algoritmos de ordenação que podem ser empregados para a resolução de problemas. A correta escolha do algoritmo, afinal, impactará significativamente no desempenho do programa a ser desenvolvido.

Inicialmente, abordamos os algoritmos de ordenação elementares (inserção, seleção e bolha) e realizamos a análise de complexidade de cada um desses métodos. Por fim, discutimos sobre os algoritmos de ordenação avançados (*merge sort, quick sort* e *shell sort*) e apresentamos as complexidades computacionais de cada um deles.

Podcast

Ouça esse podcast e veja uma visão geral do conteúdo de algoritmos de ordenação avançados.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Referências

CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Elsevier, 2002.

PERKOVIC, L. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PONTI JÚNIOR, M. **Ordenação em memória interna (parte 1)**. São Paulo: USP, 2010.

SAMPAIO NETO, N. C. **Análise de algoritmos** – algoritmos de ordenação. Macapá: UNIFAP, 2016.

SAUNDERS, D. **Algorithms**: recurrence relations – master theorem and muster theorem. Newark: University of Delaware, 2011.

Explore +

Pesquise na internet sobre outros algoritmos de ordenação, como o *heap sort* (seleção em árvore), e analise a complexidade computacional de cada um.