Aula 10

Imagine que você está na wikipedia, lendo algum artigo muito interessante e, de repente aparece uma tabela mostrando dados ainda mais interessantes. Cada coluna da tabela tem uma seta para cima e outra para baixo. Quando você clica em uma delas todas as linhas são ordenadas de forma crescente ou decrescente. Se quiser testar um exemplo, basta ver a lista de países por PIB nominal.

Imagine agora que você viajou para o Japão. No hotel em que você está hospedado tem uns computadores e você decide logar na sua conta do Facebook. Seu *feed* é então carregado em, no máximo, alguns segundos. O carregamento rápido do seu *feed* soa natural, correto? De repente levou um pouco menos de 10 segundos, mas mesmo assim foi rápido o suficiente. Então você percebe que está em uma rede ultrarrápida do hotel, no Japão. Se tiver demorado mesmo quase uns 10 segundos, existe algum motivo para essa demora?

Daí você lembra que os dados de sua conta estão armazenados em algum(ns) servidor(es) no mundo e, provavelmente, esse (s) servidor(es) não está(ão) no Japão. Ou seja, no site do Facebook você forneceu seu *login* e senha e o site procurou e achou seus dados, onde quer que eles estivessem, e te entregou. Então uma pergunta surge: como que os seus dados foram encontrados e entregues a você de forma tão rápida em um computador aleatório no Japão?

Parte da resposta está nos algoritmos de pesquisa e ordenação. Esses são algoritmos específicos para ordenar e buscar dados que estão armazenados em alguma estrutura de dados. Os dois tipos mais simples de pesquisa são:

- **Pesquisa sequencial**: onde a busca ocorre sequencialmente, a partir do primeiro elemento, até que se encontre o elemento procurado.
- **Pesquisa em intervalos**: tem como exemplo a pesquisa binária, a qual vimos rapidamente na aula anterior. Entretanto, neste caso a estrutura de dados precisa estar ordenada para que a pesquisa tenha um desempenho satisfatório. Lembrando rapidamente, basta verificar a chave do elemento procurado (por exemplo, 5) e compará-la à chave do elemento central (por exemplo, 25). Se a chave procurada for menor (no exemplo é, pois 5 < 25), então a busca ficará concentrada na primeira

metade dos elementos, e podemos ignorar seguramente o restante. A comparação é feita novamente, e a busca é novamente direcionada para uma das metades. Isso vai acontecendo até que o elemento seja encontrado.

Existem vários outros algoritmos de busca, inclusive é uma área de estudo dentro da Inteligência Artificial. Porém, o foco da aula será sobre os algoritmos de ordenação. Esses são algoritmos que irão ordenar os elementos contidos em uma estrutura de dados, de forma a deixá-los dispostos em uma ordem crescente ou decrescente, de acordo com algum critério. Por exemplo, ordenar uma lista telefônica de forma crescente e como critério a ordem alfabética dos contatos.

Algoritmos de Ordenação

Veremos os algoritmos de ordenação mais comuns: `Selection Sort`, `Insertion Sort`, `BubbleSort`, `QuickSort` e `MergeSort`.

Alguns algoritmos de ordenação, como o `QuickSort` e o `MergeSort` são capazes, ou possuem versões para ordenação de estruturas de dados que não estão por completo na memória principal. A diferença da quantidade de dados a ser ordenado faz com que existam duas categorias de ordenação

- **Ordenação Interna**: quando todos os dados cabem/estão na memória principal.
- **Ordenação Externa**: quando os dados não cabem por completo na memória principal e, portanto, parte deles precisa ficar armazenado em alguma mídia externa.

Os algoritmos de ordenação mais comuns/simples são aplicados em `arrays` / `vetores`. Por isso, a partir de agora, utilizaremos apenas `arrays` de números inteiros.

Por fim, temos duas métricas comuns para comparar os algoritmos de ordenação: (1) quantidade de comparações e (2) quantidade de trocas.

Selection Sort

Ou `Ordenação por Seleção`, é um algoritmo que busca o menor valor para a menor posição "disponível" (numa ordenação crescente).

Descrição

- Elementos possíveis do algoritmo
  - Vetor: *v*
  - Tamanho do vetor (quantidade de índices)
  - Índice do último elemento ordenado: *u*
  - Índice do menor elemento em questão: *min*
  - Índice do elemento atual: *i*

O algoritmo começa com ***u = min = 0*** e ***i = 1***. O vetor começa a ser percorrido e toda vez que v[i] < v[min], o valor de ***min = i***. Quando o vetor termina de ser percorrido sabemos que ***min*** tem o índice do menor elemento. Então trocamos o valor de **v[u]** pelo de **v[min]**. O valor de ***u*** é acrescido em 1, ou seja ***u += 1***, e percorremos o vetor novamente, em busca do menor elemento. Todo o processo é repetido até que todo o vetor esteja devidamente ordenado.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Vamos ver o exemplo do gif acima. Temos ***v = [8 5 2 6 9 3 1 4 0 7]***.

- Primeira vez percorrendo o vetor (até o seu fim)
  - Começamos então com
    - u = 0
    - min = 0
    - i = 1
    - v[u] = 8

        - v[min] = 8
        - v[i] = 5
    - Como ***v[i] < v[min]***, então
        - u = 0
        - min = i = 1
        - i += 1 = 2
        - v[u] = 8
        - v[min] = 5
        - v[i] = 2
    - Como novamente ***v[i] < v[min]***, então
        - u = 0
        - min = i = 2
        - i += 1 = 3
        - v[u] = 8
        - v[min] = 2
        - v[i] = 6
    - Dessa vez ***v[i] > v[min]***, portanto atualizamos somente o valor de i
        - u = 0
        - min = 2
        - i += 1 = 4
        - v[u] = 8
        - v[min] = 2
        - v[i] = 9
    - Novamente ***v[i] > v[min]***, portanto continuamos atualizando somente o valor de ***i***
        - u = 0
        - min = 2
        - i += 1 = 5
        - v[u] = 8
        - v[min] = 2
        - v[i] = 3
    - Mais uma vez, apenas o valor de ***i*** é atualizado
        - u = 0
        - min = 2
        - i += 1 = 6
        - v[u] = 8

        - v[min] = 2
        - v[i] = 1
    - Encontramos então um elemento que é menor do que aquele em ***v[min]***. Portanto, atualizamos assim
        - u = 0
        - min = i = 6
        - i += 1 = 7
        - v[u] = 8
        - v[min] = 1
        - v[i] = 4
    - Como ***v[i] > v[min]***, então atualizamos somente o valor de ***i***
        - u = 0
        - min = 6
        - i += 1 = 8
        - v[u] = 8
        - v[min] = 1
        - v[i] = 0
    - Encontramos novamente um elemento menor. Então fazemos as atualizações necessárias
        - u = 0
        - min = i = 8
        - i += 1 = 9
        - v[u] = 8
        - v[min] = 0
        - v[i] = 7
    - O valor de ***i*** chegou ao limite, ou seja, ao valor do último índice de ***v***. Então trocamos o valor de ***v[u]*** pelo valor de ***v[min]***:
        - v[u] = v[min] = 0
        - v[min] = v[u] = 8
        - v = [0 5 2 6 9 3 1 4 8 7]
    - O valor de ***u*** é atualizado em 1, ou seja
        - u += 1 = 1
- Segunda vez percorrendo o vetor (até o seu fim)
    - Temos
        - u = 1
        - min = 1

    - i = 2
    - v[u] = 5
    - v[min] = 5
    - v[i] = 2

A partir de então todo o vetor é percorrido novamente, da mesma forma que foi mostrado na primeira iteração. O segundo menor valor é encontrado e colocado na segunda posição, e assim por diante.

Vídeos explicativos

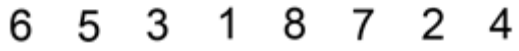
- Dança
- Animação (short)
- Animação

Insertion Sort

Ou `Ordenação por Inserção`, verifica um elemento de um vetor e o insere na posição correta.

Descrição

O algoritmo inicia no segundo índice do `array`. Esse elemento é então comparado ao primeiro. Caso seja menor, é trocado. O algoritmo continua verificando os índices seguintes e, para cada um, há uma comparação com os elementos nos índices anteriores, até encontrar o índice onde deve ser inserido. Ao ser encontrado o índice, os demais elementos devem ser deslocados.



Vídeos explicativos

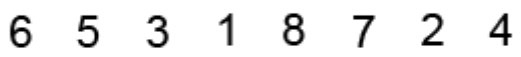
- Dança

- Animação (short)
- Animação

BubbleSort

Ou `Método da Bolha`, percorre um vetor de 2 em 2 elementos. Se o primeiro é maior que o segundo, então ocorre uma troca. O algoritmo é repetido até que não haja mais trocas.

Uma vez que os maiores elementos acabam sendo colocados nas últimas posições a cada vez que o vetor é percorrido, o `bubblesort` pode ser implementado de forma a ignorar os últimos elementos a cada vez que o vetor é percorrido, de forma que toda nova iteração menos elementos são comparados e trocados.

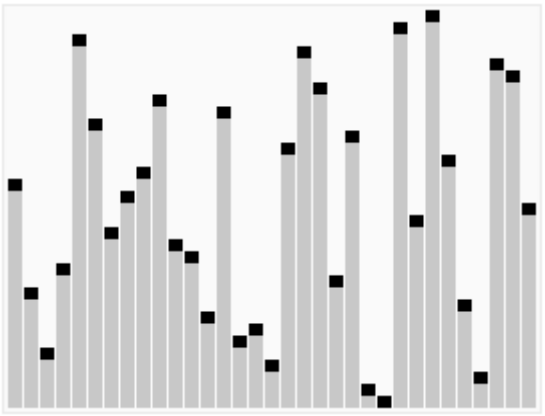


Vídeos explicativos

- Dança
- Animação (short)
- Animação

QuickSort

Esse algoritmo adota a estratégia `dividir para conquistar`. Na prática um elemento é escolhido como `pivô` dentro de uma faixa de índices possíveis. Essa escolha pode ser aleatória ou não. Um ponto de divisão é escolhido de forma que todos os valores que são menores do que o pivô são deixados antes dele e os maiores são deixados depois. Dessa forma é possível garantir que o elemento pivô está no seu índice correto. Entretanto, os elementos que estão antes e depois dele foram dois subvetores não ordenados. De forma recursiva cada subvetor é escolhido como uma faixa de pindices para ser processado pelo `quicksort`.



Vídeos explicativos

- Dança
- Animação

MergeSort

Também adota as estratégias `dividir para conquistar` e recursividade. O algoritmo pode ser descrito basicamente através de dois passos:

1. Divida a lista (ou *array*) em *n* sublistas, cada uma contendo apenas um elemento.
2. Junte as sublistas repetidamente, de forma a produzir novas sublistas ordenadas até que reste somente uma única lista ordenada.

6 5 3 1 8 7 2 4

Vídeos explicativos

- Dança
- Animação (short)
- Animação