[00:00] Olá, boas-vindas ao curso de **algoritmos com JavaScript**.

[00:07] Nesse curso, nós vamos conversar sobre o que são algoritmos e por que o estudo de algoritmos é tão importante em programação. Vamos começar analisando alguns problemas comuns de programação e de que forma podemos resolvê-los com algoritmos, algoritmos diversos.

[00:23] Esse curso está voltado para quem está começando do começo em programação e escolheu o JavaScript como linguagem, mas que já teve contato com os fundamentos do JavaScript: o que são dados primitivos, número, string, booleano, arrays, objetos e funções.

[00:43] Onde nós começamos e até onde vamos? Vamos começar traduzindo problemas, às vezes problemas do dia a dia mesmo, que resolvemos de cabeça, como falamos.

[00:54] E vamos traduzir a resolução desses problemas para o passo a passo de um algoritmo. Vamos escrever esses algoritmos usando funções e métodos do JavaScript. Vamos transcrever esses passos, esses fluxos para código.

[01:11] Juntar algoritmos. Às vezes, nós precisamos, para resolver problemas, trabalhar com mais de um algoritmo de uma vez. Vamos ver como usamos isso, como isso funciona no código.

[01:21] E testes de mesa. Às vezes, nós sabemos que um bloco de código está funcionando, mas não entendemos totalmente de que forma isso está ocorrendo. Por exemplo, quando usamos um algoritmo pronto – vamos ver como isso funciona durante o curso – temos lá um código que resolver um problema, mas o que está acontecendo dentro desse código?

[01:41] O que está guardado nas variáveis, o que faz cada linha? Então, os testes de mesa nos ajudam a seguir o fluxo de um algoritmo, o fluxo de um bloco de código linha por linha, variável por variável, e nós conseguimos entender exatamente todos os passos. Então, vamos fazer alguns testes de mesa.

[02:00] Vamos comparar algoritmos também. Como sabemos se um algoritmo funciona bem para o nosso caso e o que é funcionar bem, quando falamos de algoritmos.

[02:10] Análise de complexidade porque, pensando que algoritmos executam instruções, instruções de código, como conseguimos saber quão complexo um algoritmo pode se tornar baseado na quantidade de dados que temos para trabalhar com ele?

[02:26] Então, tenho muitos dados, tenho uma lista com muitos livros, tenho vários candidatos de um vestibular, como eu uso algoritmos de busca para isso? Como uso um algoritmo de ordenação?

[02:39] E o quão complexo que o código desse algoritmo, que as instruções podem se tornar em termos de processamento, de acordo com a quantidade de dados que trabalhamos? Vamos ver como isso funciona também.

[02:51] Nesse curso, vamos trabalhar firme com algumas partes fundamentais do JavaScript. Tipos de dados, variáveis, funções, arrays e objetos. Mas não vamos parar para explicar em detalhes o funcionamento, o conceito por trás de cada uma dessas partes fundamentais.

[03:07] Você vai aproveitar melhor o curso se já tiver feito os cursos anteriores sobre fundamentos do JavaScript ou se você já tiver estudado sobre esses assuntos.

[03:17] Lembrando que, nesse curso, nós não vamos falar sobre assuntos específicos de front-end, mas tudo que vamos ver aqui você pode abstrair e aproveitar os conceitos da mesma forma.

[03:29] Depois desse curso, você pode estudar mais algoritmos por conta própria, entender como diversos algoritmos funcionam para diversos problemas.

[03:37] Quais as complexidades de cada um e como você consegue, por exemplo, pegar o conceito de um algoritmo escrito em pseudocódigo e passá-lo para JavaScript para você conseguir usar em um projeto ou testar e entender melhor como ele funciona.

[03:49] Eu sou a Ju Amoasei e vamos começar.

JavaScript é conhecida como a “linguagem da web”. Embora tenha se desenvolvido no front-end, atualmente cada vez mais a utilizamos no chamado back-end, ou server side (lado do servidor) das aplicações web. Este curso foca no uso do JavaScript para back-end.

Assim, precisaremos instalar duas ferramentas: o **NodeJS**, necessário para executar código JavaScript sem precisar do navegador, e o **Visual Studio Code** ou VSC, editor que usamos para escrever código.

## NodeJS

Caso já tenha feito algum curso anterior de JavaScript para back-end, o NodeJS pode já estar instalado. Se não tiver certeza, é possível conferir com os seguintes passos:

1) Abra uma janela do terminal em seu computador. Isso pode ser feito da seguinte forma:

**Windows**: Você pode acessar a busca no menu inicial e procurar por Prompt de Comando, ou acessar via Menu Iniciar > Sistema do Windows > Prompt de Comando.

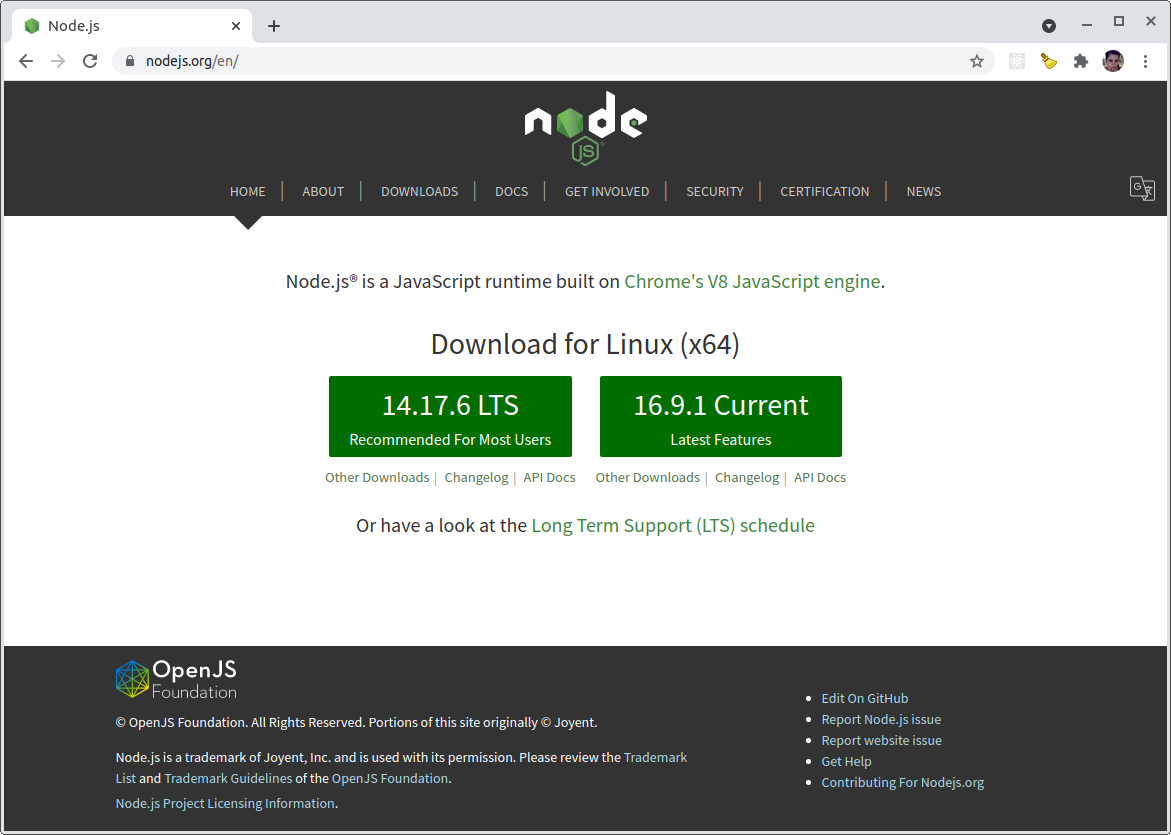
**MacOs**: O Terminal está disponível no menu de Aplicações, dentro da pasta de Utilitários.

**Linux (Ubuntu)**: O Terminal está disponível no menu de Programas/Aplicações. Caso não localize, pode estar dentro da pasta Utilitários.

2) No terminal, digite node --version ou node -v - caso retorne um número de versão, como por exemplo, v14.16.0, o NodeJS já está instalado.

Caso precise instalar, siga as instruções para cada sistema operacional que estão na [página inicial do NodeJS](https://nodejs.org/pt-br/). Este curso foi desenvolvido usando a versão 14.10.1.

O site do NodeJS oferece duas opções para download, a LTS e a "current" (atual). Você pode escolher a versão LTS (Long Term Support, ou Suporte a Longo Prazo) e clicar no botão correspondente para baixar e instalar normalmente como qualquer outro programa.



## Visual Studio Code

O Visual Studio Code é o editor que escolhemos utilizar para escrever nossos códigos durante este curso. Os links para baixar e instalar de acordo com o seu sistema operacional estão na [página inicial](https://code.visualstudio.com/) do VSC.

Existem vários outros editores, como o Atom, o Sublime e o Notepad++. Caso já tenha algum destes instalado ou preferência por outro, fique à vontade! Note apenas que alguns recursos e ferramentas variam de editor para editor, e podem não estar disponíveis ou funcionar de forma diferente.

Agora podemos começar!

[00:00] Boas-vindas ao curso de algoritmos com JavaScript. Vamos começar vendo qual é o nosso problema, o que nós vamos trabalhar nesse curso.

[00:10] Vamos supor que nós ganhamos um vale-presente em uma livraria de X, digamos, cinquenta reais, cem reais, e queremos ganhar esse vale-presente em livros de programação.

[00:20] Nós entramos no site da livraria, procuramos por livro de programação, e recebemos algumas opções. Só que o site da livraria trouxe para nós essas opções – um livro de JavaScript (25,00), um livro de PHP (15,00), um livro de Java (30,00), Elixir (50,00), Go (45,00) e Python (20,00) – que não estão em nenhuma ordem particular que consigamos ver.

[00:41] Não está em ordem alfabética e não está em ordem de preço, os preços estão misturados, os mais caros com mais baratos etc. Então, qual é o problema que nós temos? Quando temos um vale-presente, normalmente nós queremos fazer o melhor uso possível dele.

[00:54] Então, o que vale mais a pena? Tenho um vale de cinquenta reais, comprar cinco livros de dez reais ou comprar dois livros de vinte e cinco reais? Para conseguirmos fazer isso, temos que saber quais são os livros mais baratos, como eu consigo somar etc.

[01:08] Então, temos aqui a nossa busca de livros que trouxe uma lista desordenada para nós, não está em nenhuma ordem que nós conseguimos identificar. Essa lista de livros é pequena, então nós conseguimos ir guardando na memória, mais ou menos, a ordenação.

[01:26] Vamos ver qual é o mais barato? Eu olho aqui, o de JavaScript é o primeiro da lista, 25,00 reais, é o único livro que eu tenho. Depois tem o de PHP, que é 15,00. O livro de PHP é mais barato que o de JavaScript, então guardei isso na memória.

[01:39] Tem o de Java de 30,00 reais, o de Elixir de 50,00. Tenho na cabeça que o de PHP ainda é o mais barato, 15,00 reais. O de Go por 45,00 e de Python por 20,00. Então, dá para guardar na memória que os mais baratos são de PHP, de JavaScript e o de Python.

[01:58] O de PHP do 15,00, o de Python por 20,00 e o de JavaScript por 25,00. Temos 50 reais, mas dá para compensar o resto com o cartão. Então, vamos pensar no que fizemos.

[02:11] Na nossa cabeça, nós guardarmos alguns valores enquanto íamos percorrendo uma lista. Então, vamos dar uma olhada no que conseguimos fazer, de novo, anotando mais ou menos os passos que nós percorremos.

[02:30] Vou voltar minha lista ao que ela era. Vamos percorrer. Vamos dizer que essa lista de livros é um *array* de livros, podemos tratar dessa forma. Como que na nossa cabeça nós criamos mais ou menos um passo a passo de como resolver essa ordenação de livros?

[02:54] Começamos no que seria a primeira posição da nossa lista. Temos um valor só, não temos o que fazer com ele. Mas agora, passando para o segundo valor, eu tenho dois valores, 25,00 e 15,00, já consigo comparar.

[03:09] Então, eu consigo já passar, por exemplo, para o primeiro valor da lista o meu livro de 15,00 reais, então o livro de 15,00 está guardado como o mais barato. E, em seguida, ir passando para os próximos da lista.

[03:23] Então, agora eu sei que o livro de 15,00 é mais barato. Tenho em seguida um livro de 25,00, um livro de 30,00 – ok, por enquanto está ordenado –, um livro de 50,00 – por enquanto está ordenado –, depois do livro de 50,00, eu tenho o livro de 45,00.

[03:38] Ok, o livro de 45,00 é mais barato que o de 50,00, então já sei que o de 50,00 não está entre os livros mais baratos, posso trocá-los de lugar. E meu último livro da lista é um livro de 20,00, que eu já sei que é mais barato que é de 45,00, que o de 50,00, que o de 30,00 e que o de 25,00.

[03:59] Então, posso ficar trocando todos os meus livros de lugar até eu, na minha cabeça, fazer algo parecido com uma lista ordenada. Agora, se eu quiser, por exemplo, só o livro mais barato, é o livro de 15,00 reais; se eu quiser os três primeiros livros eu já sei que são os livros de PHP, Python e JavaScript, 15,00, 20,00 e 25,00 reais, e por aí vai.

[04:23] Então, vamos um pouco além de guardar as coisas na cabeça. Como vamos trabalhar com JavaScript, nós podemos supor que a nossa lista de livros é um *array*, como já tínhamos conversado. Os *arrays* no JavaScript começam com 0, então nós temos uma lista que vai do índice 0 até o índice 5.

[04:43] Como é que nós pensamos, na nossa cabeça, para guardar os valores? Nós meio que fomos guardando na nossa cabeça sempre qual era o valor mais barato.

[04:53] Então, na nossa cabeça, nós percorríamos a lista. Vi primeiro o livro de 25,00, então o livro de 25,00, quando nós começamos a percorrer essa lista, é o menor. Só tenho um valor, 25,00, é o único valor que eu tenho e também é o valor menor.

[05:11] Então, meu valor atual é o índice 0 e meu valor menor também é o índice 0. Passando para o próximo item da lista, o índice 1 do nosso *array*, agora eu tenho um livro de 15,00.

[05:24] 15,00 é menor do que 25,00, então o meu atual, o livro que estou verificando agora, é o livro que está no índice 1, e o meu menor também é o índice 1, porque 15,00 é menor do que 25,00.

[05:43] Eu sei que agora, passando para frente, estou no índice 2. O índice 2 do meu *array* é um livro de Java, que é de 30,00 reais, então o meu atual agora passa a ser 2, o terceiro elemento da minha lista.

[05:59] Ele não é menor do que 15,00, 30,00 não é menor do que 15,00, então o meu menor continua sendo o 15,00 reais. Eu mantenho isso na cabeça e mantenho isso na minha “variável mental” que estou chamando de menor.

[06:14] Passamos para o próximo, Elixir, que está no índice 3 do *array*, sendo o quarto item, o quarto elemento. 50,00 ainda não é menor do que 15,00, então meu item menor continua sendo o PHP, que está no índice 1.

[06:30] Passamos para a frente, agora vou para o próximo da lista, que é o livro de Go, no índice 4. Então, atualmente, estou analisando o 4. 45,00 também não é menor do que 15,00, então, na minha variável menor, continuo guardando o livro de PHP.

[06:47] E, por último, eu passo para o último elemento do *array*, que é o índice 5, é o sexto elemento, que é um livro de Python que é 20,00 reais. 20,00 não é menor do que 15,00 também, então continuo a minha variável menor, minha variável mental, continua guardando o livro que estava no segundo elemento, o índice de número 1 do *array*.

[07:12] Então, no final, o que nós fazemos na nossa cabeça sem perceber? Nós percorremos um *array*, percorremos uma lista de elementos e guardamos na cabeça, na nossa variável mental que eu chamei de menor, o índice – qual era o produto, na verdade, na cabeça nós não guardamos por índice –, qual era o produto que tinha o menor preço.

[07:35] Que é o de PHP, que no *array* de JavaScript está no índice 1, e por aí vai. Então, o que nós fizemos foi basicamente não compilar, porque o JavaScript não é uma linguagem compilada, mas nós interpretamos, fizemos um pequeno programa que rodou sem nem percebermos – nós fazemos isso muitas vezes por dia –, e o que nós criamos na nossa cabeça foi um algoritmo.

[08:03] Então, o que é um algoritmo? O que é essa sequência que nós criamos na cabeça e criamos com programas de computador? Segundo a definição da Wikipédia, podemos colocar aqui, o algoritmo é uma sequência finita de ações executáveis que visam obter uma solução para um determinado tipo de problema.

[08:23] Dá para traduzirmos essa explicação, traduzir algoritmo como basicamente tudo que nós fazemos, também no dia a dia, e as instruções que damos para o computador que envolvam sequências de passos definidas.

[08:38] Então, por exemplo, o que mais nós fazemos no dia a dia que podemos mudar para um algoritmo? Quando seguimos uma receita. Então, vamos ver, por exemplo, uma receita de bolo é uma sequência finita de ações executáveis, nós executamos todas as partes da receita, que visa obter uma solução para um determinado tipo de problema.

[09:00] No caso, falta de bolo, um aniversário, vontade de comer bolo, por exemplo. Na vida real, o que nós fazemos? Nós dividimos um algoritmo de uma receita de bolo em vários passos, nós não percebemos.

[09:14] Por exemplo, o primeiro passo seria conferir se temos todos os ingredientes. E tem condicionais nesses passos. Por exemplo, podemos dizer que, se não tiver ingredientes, o que nós fazemos? Nós saímos para comprar o que falta ou desencanamos?

[09:29] Se desencanamos de fazer o bolo, o restante do programa, que é fazer o bolo em si, nem é executado. E tem uma outra coisa que fazemos também, nós acabamos dividindo o algoritmo de fazer um bolo em várias partes pequenas sem perceber.

[09:43] Não é só as duas partes principais de ver sem tem todos os ingredientes, tem várias coisas que aprendemos no dia a dia de fazer bolo. Por exemplo, conferir os ingredientes é uma, para não esquecermos no final de botar o fermento que não tem fermento, e também algumas outras coisas.

[10:01] Se você faz bolo com ovos, por exemplo, você vai querer quebrar o ovo em um pote ao invés de quebrar o ovo na massa, porque o ovo pode estar podre. Essa é uma instrução que, normalmente, nós aprendemos, recebemos essa instrução pronta, alguém nos fala, alguém nos dá essa dica.

[10:16] Você pegar o ovo, tirar o ovo do fluxo do bolo, quebrar em um pote e voltar, é uma pequena parte que nós separamos. Então, na vida real, nós escrevemos algoritmos na cabeça, nós programamos esses algoritmos, terminamos as nossas sequências finitas de instruções – por exemplo, fazer um bolo, botar no forno etc. –, só que não percebemos que fazemos isso na cabeça.

[10:46] E não só nós seguimos essas sequências de passos, como também fazemos essas pequenas reparações, o que seriam algoritmos menores, algoritmos de ver se tem todos os produtos, se tem tudo que nós queremos, se não tem, compra, ou se não tem, desencana, não cumpre o resto.

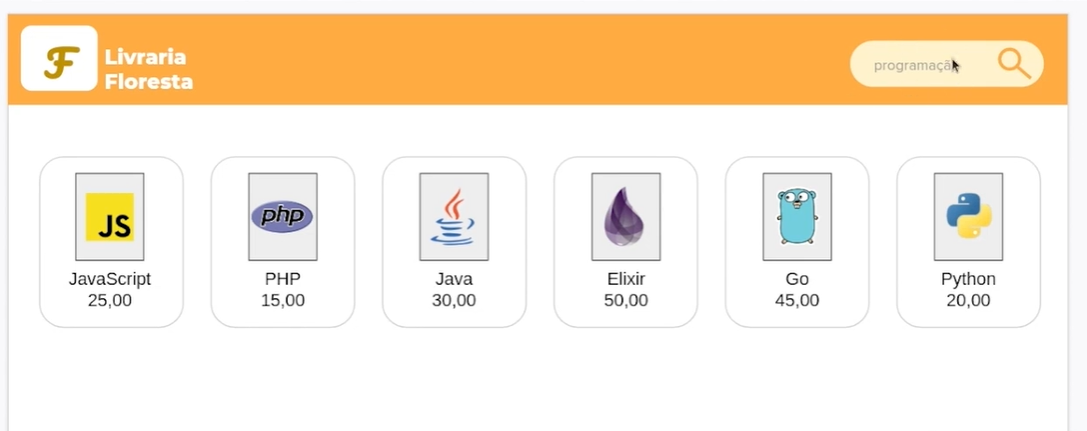
[11:03] Nós podemos traduzir todo esse rolê do bolo que nós fizemos para o nosso problema de encontrar o livro mais barato em uma lista desordenada. Essa história de quebrar o ovo no pote, de aprendermos a fazer bolo com o tempo, recebendo diversas dicas, é que existem vários algoritmos já estabelecidos, já prontos para vários problemas comuns de programação.

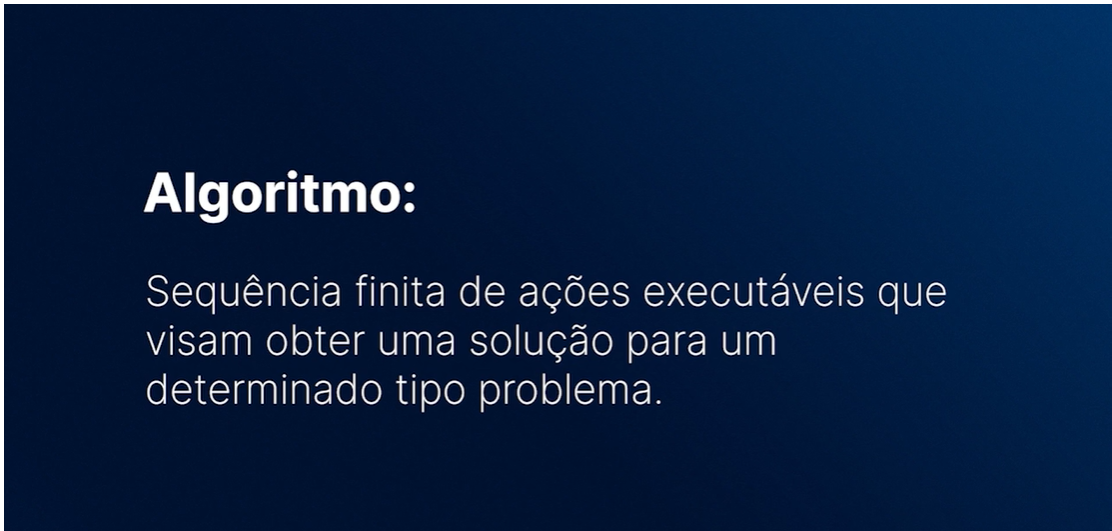
[11:28] Isso é mais ou menos o que vamos ver nesse curso: como pensamos em algoritmos, como testamos algoritmos, vamos ver alguns que já existem e se eles funcionam para nós, se eles são o melhor caso etc.

[11:41] Porque um mesmo problema, por exemplo, ordenar uma lista, fazer um bolo, pode ter mais de uma forma de ser resolvido, e a escolha errada também pode causar problemas.

[11:51] Por exemplo, eu tenho um algoritmo de me trocar e eu posso escolher colocar o tênis antes de colocar a calça. Funciona? Funciona, mas obviamente, colocar o tênis antes de colocar a calça vai ser muito mais trabalhoso do que colocar a calça antes de colocar o tênis.

[12:04] E por aí vai. Então, vamos dar uma olhada como transformamos esse algoritmo mental nosso em JavaScript, mas isso vamos ver no próximo vídeo.





[00:00] Agora que já temos o nosso "algoritmo mental" do fluxo que fizemos em JavaScript, vamos escrever esse código. Então, como fazemos para passar aquele fluxo que pensamos que criamos direto na página da nossa livraria? Vamos passar isso para o JavaScript, então.

[00:22] Primeira coisa que vamos precisar é criar um *array*, então vou criar const precosLivros. Por enquanto, criamos somente um *array* com o preço de cada livro, para começarmos em pequenos passos.

[00:37] Os preços são [25, 15, 30, 50, 45 20]. Então, um *array* em JavaScript. Nós precisamos guardar em algum lugar aquilo que eu estava chamando de variáveis mentais – agora não vai ser, agora vamos fazer no JavaScript.

[01:08] Uma que guarde o valor atual que estamos vendo, qual o produto atual que estamos analisando, vou chamar de let atual. Vou iniciá-la com 0, vai ser o primeiro índice do *array* que estamos trabalhando.

[01:24] E um outro let que vou chamar de maisBarato, que também vou iniciar com 0. É aquele onde vamos guardar - que estávamos guardando na cabeça - qual é o produto, qual é o índice do *array* que tem o menor valor. No caso do nosso *array*, teria que guardar o 15 mais do que qualquer outro.

[01:47] O que temos que fazer agora? Temos que percorrer toda essa lista de livros, confirmar qual é o mais barato de todos, porque sabemos na cabeça, mas agora o JavaScript tem que nos dizer.

[01:59] Podemos usar uma das várias formas que o JavaScript tem para trabalhar com *loops*, com laços de repetição, o que temos que fazer é percorrer listas, percorrer um *array*, que é um for. Então, vou criar um for que vai começar com let.

[02:16] Vou chamar a variável inicial desse *for* de atual, então for (let atual = 0;). Já que estamos iniciando, dentro do nosso *for*, uma let atual, nós não precisamos iniciá-la do lado de fora, nós podemos trabalhar só com a variável atual que está dentro do *for*, já vai fazer o mesmo trabalho.

[02:37] Vamos percorrer esse *array* enquanto atual for menor do que o tamanho do nosso *array* de livros, então (let atual = 0; atual < precosLivros.length;), com “gth” no final – sem errar, porque eu sempre erro e troco as últimas letras.

[02:58] Vamos atualizar a nossa variável atual, vamos incrementá-la de um em um, atual++, para ir pulando de índice em índice do *array*. E aqui dentro vamos criar o nosso //código.

[03:14] Se lembrarmos a forma como nós criamos esse algoritmo na nossa cabeça – vou voltar na nossa página –, o que tem que ser feito dentro desse *for* nós já falamos. Se o produto atual for menor, ele se torna o mais barato.

[03:33] Então, se o que nós estivermos vendo no momento na lista, por exemplo, “1” e “PHP”, ele é o que nós estamos vendo na lista? Ele é o de preço menor do que tínhamos antes? É. Então, ele se torna o menor, ele se torna o mais barato. E por aí vai.

[03:53] Nós meio que falamos, na cabeça, o que temos que transformar em código. Então, voltando para o nosso *for*, o que temos que fazer é uma condição, if.

[04:07] Se o produto atual, for o mais barato da lista, nós o guardamos dentro da nossa variável “preço menor”. No caso, eu chamei de maisBarato, então ele vai ser o mais barato e nós vamos guardar o número de índice dele dentro da nossa variável let maisBarato.

[04:27] A nossa condição é if (precosLivros), que é o nome do nosso *array*, na posição [atual], for menor do que nós tínhamos anteriormente – o que tínhamos anteriormente?

[04:44] Nós começamos com “0” para comparar, então o primeiro livro que nós comparamos, ainda não tem muito o que ser comparado. Se só temos um livro de 25, ele é o atual e é também o mais barato, então estamos começando tudo com 0.

[04:58] Então, vamos dizer que precosLivros na posição maisBarato. Lembrando que as variáveis guardam valores inteiros que traduzimos como índices do *array*. Então, precosLivros na variável atual é precosLivros na posição 0, ou seja, na posição 25.

[05:22] Vou até marcar com o comentário aqui, a posição 0. E precosLivros na posição maisBarato, o maisBarato também é uma variável que está guardando 0, e 0 é 25, então é onde estamos começando.

[05:37] Se a atual for menor do que já está guardado como maisBarato, vamos fazer alguma coisa, ou seja, temos que fazer com que o livro na posição atual se torne o mais barato – esquece o que tinha antes.

[05:53] Então, nós começamos na posição 0, mas, se a posição 1 for mais maisBarato, esquece o 0 que estava guardado dentro da variável e vamos colocar o 1 aqui.

[06:03] Então, o que tem que acontecer? maisBarato = atual. Nós declaramos uma let na nossa linha 4, ou seja, podemos substituir o valor dela de 0 para 1, de 1 para 2, de 2 para 3, de acordo com o índice onde está o menor valor do *array*.

[05:24] No caso, o 15, que já sabemos, olhando, que é o índice 1, mas o JavaScript ainda não sabe, ele vai ter que processar isso para nós.

[06:34] A última coisa a fazer, depois que sai desse *for*, é um console.log e olivro mais barato custa e podemos colocar, com *template strings* aqui, abrir cifrão e chaves, colocar ${precosLivros}, que é o nome do nosso *array*, em qual posição? Na posição maisBarato.

[07:04] Porque, lembrando, a partir do momento que o JavaScript entra no *for*, ele vai percorrer o *for* inteiro antes de continuar. Então, o console.log que nós colocamos por último, na linha 12, só vai ser executado depois que passar por todo o *array*.

[07:20] Então, nesse momento, nossa variável maisBarato teoricamente já está atualizada, passou pelo *array* inteiro, e já está guardando o que é realmente o menor valor da lista.

[07:30] Então, é precosLivros na posição que não sabemos – sabemos que é 1, mas o JavaScript ainda não. maisBarato que vai ter aqui um número entre 0 e 5 guardado dentro dessa variável.

[07:44] Vou salvar, venho no terminal e mando executar com node index.js, que é o nome que eu criei para esse arquivo. Então, o livro mais barato custa 15, ou seja, nós já sabíamos e o JavaScript agora também já sabe.

[08:01] Porque nós fizemos com que o JavaScript percorresse, com o código, exatamente o mesmo caminho que nós fizemos mentalmente usando o nosso site, usando as nossas variáveis.

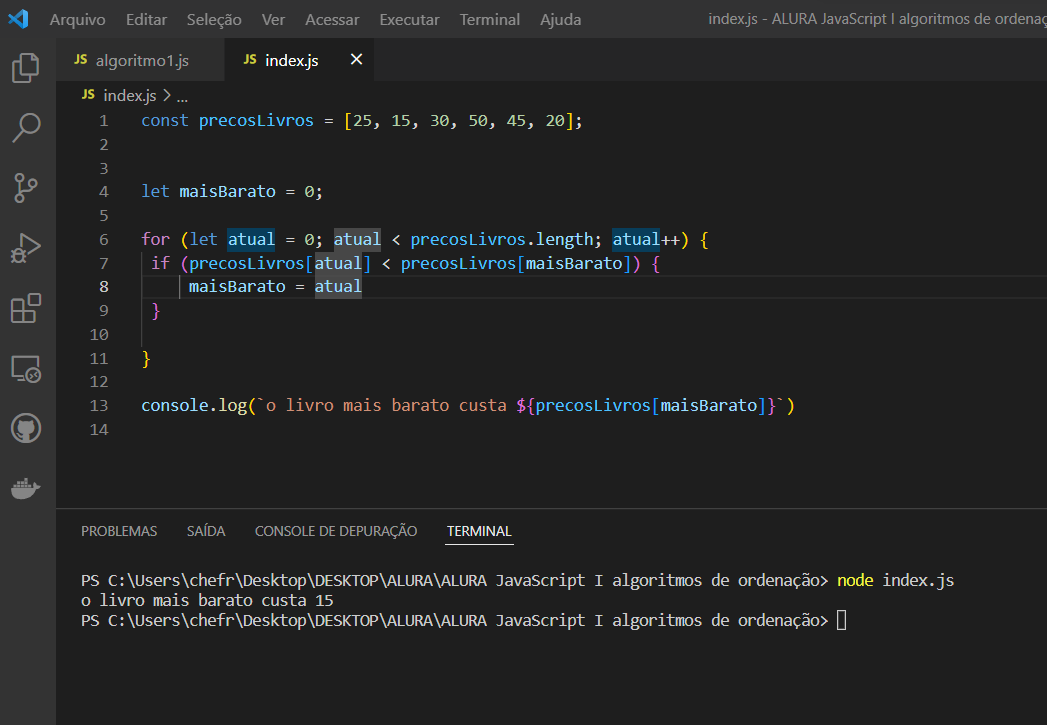
[08:20] Nós fizemos o que chamamos de “teste de mesa”, vou deixar um material extra para vocês sobre o que é um teste de mesa, mas, basicamente, nós fizemos uma interpretação do código na nossa cabeça e fomos marcando em cada momento do *loop*, em cada momento do processamento, o que tinha guardado em determinadas variáveis.

[08:37] Aqui eu chamei a variável de “menor”, no código eu acabei chamando de maisBarato, mas é a mesma variável. E a variável “atual”, que é a que está fazendo o controle desse *array*, de passar índice por índice.

[08:49] Então, o que nós fizemos foi criar o nosso primeiro algoritmo para resolver o nosso primeiro problema que é encontrar o livro mais barato em uma lista desordenada.

[08:58] Fizemos isso a partir da nossa cabeça e sem perceber, é uma coisa que fazemos todos os dias. Com o código, podemos passar os mesmos passos que fazemos na cabeça para o código, só fazendo de uma forma que o JavaScript entende.

[09:13] Mas, como falei antes, existe mais de uma forma de resolver um problema. Sempre ou quase sempre existe. Vamos dar uma olhada, continuaremos a estudar os nossos algoritmos.



[00:00] Antes de continuarmos, vamos refatorar esse *array* de livros que nós criamos para termos informações mais completas para trabalharmos. Porque trabalhar com *Array* de preços vai cumprir a missão, mas fica um pouco mais trabalhoso, porque vamos receber os valores sem saber de que livros estamos falando, e se tiver dois livros com o mesmo preço, como faz?

[00:22] Então, vamos fazer o seguinte, vou extrair esse *array* que chamei de precosLivros, vou criar outro arquivo, vou chamá-lo de “listaLivros.js”, na mesma pasta, e vou extrair essa *array* para esse arquivo.

[00:41] Então, vou continuar o chamando de precosLivros, tirá-lo do nosso “index.js” e passá-lo para a “listaLivros.js”, então “Ctrl + X” no nosso *array*, e “Ctrl + V” no nosso listaLivros.

[01:06] Então, “listaLivros.js” eu vou deixar com a variável precosLivros mesmo, e aqui, onde temos um *array* só de números, nós podemos ter um *array* de objetos, que cada objeto terá seu nome.

[01:25] Então, tenho a propriedade título: “JavaScript” para o primeiro livro, e a propriedade preco: 25 que, para JavaScript, é 25. A mesma coisa nós fazemos para todos os outros.

[01:45] O segundo elemento é PHP e a propriedade é 15: titulo: “PHP”, preco: 15.

[02:00] O terceiro eu já não lembro, nós olhamos, que é o de Java, 30,00. Então, titulo: “Java”, preco: 30. E, quase finalizando, o de Elixir é titulo: “Elixir”, preco: 50.

[02:40] Tem também o de Go e o de Python. titulo: “Go” e preco: 45; finalmente o último, mas não menos importante, que é titulo: “Python” e preco: 20.

[03:15] Agora temos um *array* de objetos e conseguimos identificar o que é o que, qual é o livro que custa 25, qual é o livro que custo 30 etc. Vamos dar uma última refatorada nesse código.

[03:30] O nosso arquivo “index.js”, agora preciso trazer o nosso *array* para cá. Vou exportar a nossa const precosLivros. Na última linha do arquivo "listaLivros.js", vou chamar module.exports = precosLivros.

[03:53] Podemos agora voltar para “index.js” e importar, na primeira linha do arquivo, antes de qualquer coisa, o const livros = require(‘./listaLivros’). Então, agora podemos chamar a const Livros que ela vai trazer o *array* para nós.

[04:25] E podemos refatorar essas partes do código do nosso *for* onde nós trabalhamos com os valores dos livros. Então, temos a nossa let atual, fica igual. Agora, precosLivros.length não é mais precosLivros porque estamos importando esse *array* de outro lugar, de livros.

[04:46] Então, precosLivros, em todos os lugares onde nós declaramos, agora é apenas livros. Lembrando que, agora, não adianta só trazermos livros[maisBarato], porque estamos trabalhando com objetos. Então, o que queremos comparar?

[05:05] Temos duas propriedades: a propriedade do titulo, que é do tipo *string*, e a propriedade preco, que é do tipo número. Então, temos que dizer no nosso *for* agora que o que nós vamos comparar é livros na posição [atual].preco e livros na posição [maisBarato].preco também.

[05:27] Senão o JavaScript não vai saber qual propriedade do objeto queremos usar para fazer essa comparação. No console.log(‘ o livro mais barato custa ${precosLivros} na posição [maisBarato].preco e podemos até adicionar um nome, e o título é, coloca outro *place holder* com cifrão e chaves, ${}.

[05:55] ${livros[maisBarato].titulo}, e aproveito e corrijo o meu console.log que não é mais (precosLivros[maisBarato].preco), agora é só (livros[maisBarato].preco) e (livros[maisBarato].titulo).

[06:18] Então, testamos novamente para ver se está tudo funcionando, limpar o terminal. E retornou: “o livro mais barato custa 15 e o título é PHP”. Tudo funcionando direito.

[06:33] O que nós fizemos aqui, na verdade, nós só demos uma melhorada no código para termos informações mais recheadas do que estamos trabalhando, e também começamos a extrair um pouco as partes do código para não ficar tudo nesse “index.js”.

[06:48] Vamos fazendo outros melhoramentos durante o curso, tem mais coisas para mexer, vamos extrair um pouco mais de código, separar um pouco mais de código, mas vamos fazendo isso aos poucos, vamos continuar vendo os algoritmos.

# Para saber mais: Representando algoritmos

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98358/next)

Você já viu ou já ouviu falar de fluxogramas?

Um fluxograma é uma forma de representarmos os passos de um algoritmo, através de símbolos bem definidos para cada tipo de processo (entrada de dados, processamento, condicionais, etc) para facilitar a visualização e o mapeamento de todos os passos necessários e/ou possíveis em um determinado processo. São muito utilizados para descrever algoritmos, mas podem ser utilizados em qualquer situação que envolva um fluxo finito de ações, com entrada e saída.

Há outras formas de se representar um algoritmo além do fluxograma, como a descrição textual/verbal dos passos (que estamos utilizando nesse curso), o pseudo-código ou mesmo a própria leitura do código em JavaScript ou outra linguagem de programação. Você pode conferir exemplos destas três formas [neste artigo da Alura](https://www.alura.com.br/artigos/algoritmos-e-logica-de-programacao).

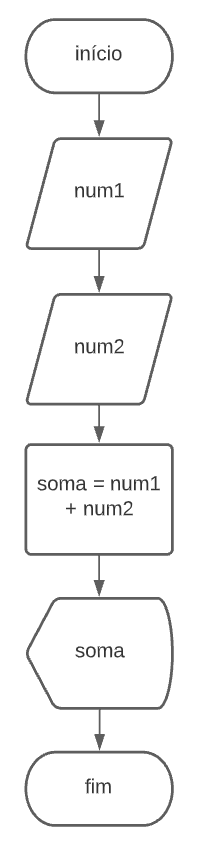
No caso do fluxograma, os diferentes formatos das caixas de texto ajudam a definir os passos lógicos de um processo. Veja alguns exemplos:



As figuras padrão para os tipos de operação em um fluxograma são:

1. Início/fim: Marcam o início ou término do processamento que está sendo descrito, normalmente “início” ou “fim”, mas também podem ser utilizados as saídas de processamentos anteriores ou entradas de processamentos posteriores, como “envio do formulário”. Representado por uma figura oval ou retângulo com bordas arredondadas;
2. Entrada manual: input (ou entrada) manual de dados, por exemplo dados informados pelo usuário do sistema. Representado por um retângulo com o lado direito mais alto;
3. Processamento: bloco ou conjunto de operações que serão efetuadas com os dados recebidos. Representado por um retângulo;
4. Decisão: quando o programa pode seguir em mais de um caminho, que envolve uma **decisão** cujo resultado é “sim” ou “não” (os equivalentes a true e false). Representado por um losango;
5. Entrada/saída: Processo de recebimento dos dados pelo programa (input ) ou retorno do resultado de um processamento (output). Representado por um paralelogramo.

Vamos ilustrar melhor com um exemplo de fluxograma para uma soma entre dois números:



O fluxograma acima descreve:

1. o ponto de início do processamento;
2. a entrada de dois dados (as variáveis num1 e num2), cada um por vez;
3. o processamento (operação matemática entre os dois números);
4. a saída do resultado através da variável soma;
5. o fim do processamento.

Podemos representar o algoritmo acima com código JavaScript:

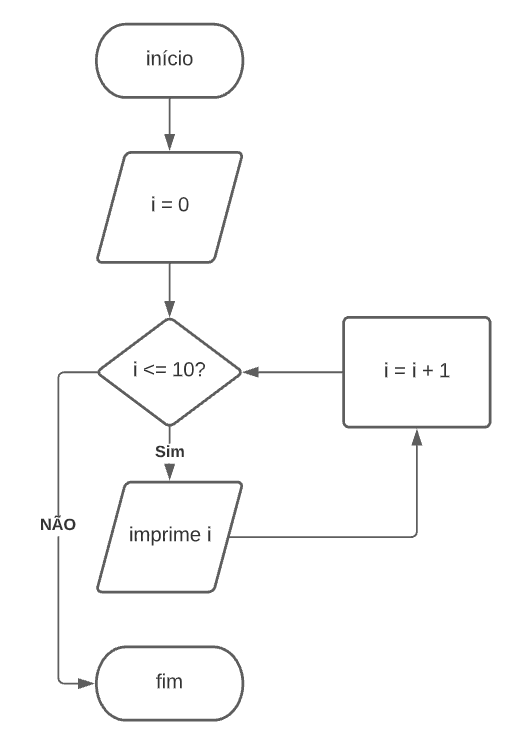
const num1 = 2;

const num2 = 2;

const soma = num1 + num2;

console.log(soma) //4COPIAR CÓDIGO

Vamos ver mais um exemplo, para um loop que exibe no console números de 0 a 10:



O fluxograma acima pode ser traduzido, em código JavaScript, em um for:

for (let i = 0; i <= 10; i++) {

console.log(i);

}COPIAR CÓDIGO

Existem outros símbolos que usamos nos fluxogramas para ilustrar outros tipos de operações possíveis, como impressão em papel ou envio de dados para um banco.

Você pode praticar com fluxogramas para criar algoritmos tanto para códigos que você já escreveu quanto para “colocar no papel” e ajudar a organizar a estrutura para um algoritmo que esteja desenvolvendo; ou mesmo treinar com outras situações, como fazer um bolo ou pagar uma conta de luz.

# Faça como eu fiz: Encontrar o livro mais caro

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98360/next)

Durante a aula, criamos um algoritmo para encontrar o menor valor em uma lista (array). Mas a mesma lógica pode ser invertida para funcionar com o maior valor.

Dentro do for, basta invertermos a condição do if, para ao invés de verificar se o produto atual tem valor **menor** <, agora verifique se tem valor **maior** >:

let maisCaro = 0;

for (let atual = 0; atual < livros.length; atual++) {

if (livros[atual].preco > livros[maisCaro].preco) {

maisCaro = atual;

}

}COPIAR CÓDIGO

Futuramente este loop poderá ser extraído para uma função que faz ambas as operações e consegue, com o mesmo código, encontrar tanto o valor menor quanto o maior.

VER OPINIÃO DO INSTRUTOR

### Opinião do

# 9 O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98361/next)

## Nessa aula, você aprendeu

* Que algoritmos são sequências finitas de passos lógicos para a resolução de um problema e que podem representar tanto problemas comuns em programação quanto qualquer situação que envolva passos lógicos;
* Que pensar de forma algorítmica não se resume a código e nem precisa necessariamente começar pelo código em si;
* Como traduzir os passos de um algoritmo para a linguagem de programação que escolhemos trabalhar (no caso, JavaScript), através de estruturas fundamentais da linguagem como variáveis, laços de repetição e condicionais.
* [00:00] Já sabemos, então, como obtemos o nome e o preço do livro mais barato de uma lista de livros. Nós criamos um algoritmo, um processo, passos finitos para resolver um problema, que percorre uma lista de livros e encontra o mais barato.
* [00:20] Acontece, normalmente, quando estamos nessa situação de eu tenho um vale-presente e quero comprar alguns livros, eu tenho um valor X para gastar em livros/mercado, nós recebemos uma lista de livros e queremos saber quais são os mais baratos.
* [00:38] Tenho um vale-presente de cem reais, vamos ver quantos livros eu consigo comprar com cem reais, não necessariamente um com o preço cheio, pode ser vários, dez livros de 10 reais etc.
* [00:50] Então, muitas vezes, nós não queremos só a opção mais barata, como fizemos anteriormente, para achar só PHP de 15,00. Nós queremos ter opções. Por exemplo, dois ou três livros mais baratos.
* [01:02] Então, novamente, o que conseguimos fazer? Podemos criar variáveis mentais, como criamos na nossa cabeça, para separar uns dois ou três livros mais baratos.
* [01:11] Mas, se a lista for muito grande, acabamos tendo que pegar papel e caneta e ir escrevendo. Se fossemos colocar essas variáveis mentais por extenso para ver, seria algo mais ou menos assim.
* [01:24] Temos o livro “atual” que estamos vendo, temos o “maisBarato”, e depois teríamos que começar a criar um monte de outras variáveis. Uma variável para o segundo mais barato, outra variável para o terceiro mais barato.
* [01:43] Se eu quisesse os cinco mais baratos de uma lista de cinquenta, teria que criar os cinco mais baratos, os seis mais baratos etc. Imagina se fossemos colocar isso no JavaScript, não ficaria muito prático.
* [01:57] Aqui, ao invés de let maisBarato, teria que ter let maisBarato, segundo maisBarato, terceiro maisBarato, e, obviamente, acaba ficando impraticável.
* [02:07] Acho que já deu para ter uma ideia que esse é um problema muito comum em programação, é um problema de ordenação, afinal de contas, é muito comum esse situação em que nós queremos ir em um *e-commerce* e achar os mais baratos, queremos achar as mensagens mais recentes, queremos achar os vídeos que têm mais *likes*, etc.
* [02:30] No caso da nossa lista de livros, os valores estão misturados, aí dificulta um pouco, mas, se modificarmos a nossa lista, como já fizemos anteriormente de ir visitando índice por índice, por exemplo, o primeiro índice é JavaScript e em seguida tenho o mais barato que é de 15,00, troco eles de lugar.
* [02:49] Agora tenho 15,00, 25,00, 30,00, 50,00, 45,00 – 45,00 é menor que 50,00, vamos botá-lo de volta – e agora tenho 45,00, 50,00 e 20,00, então sei que tenho que empurrar o 20,00 como segundo elemento, para a posição de número 1 no nosso *array*.
* [03:08] Então, se modificamos um pouco a lista e a ordenamos, fica muito mais fácil de pegarmos, por exemplo, os três livros mais baratos. Agora eu sei que os três livros mais baratos são o de PHP 15,00, Python 20,00 e JavaScript 25,00.
* [03:23] Se for uma lista de cinquenta livros e eu quero os cinco primeiros livros mais baratos, também já está ordenado, é só pegar os cinco primeiros elementos da lista.
* [03:32] Então, agora sabemos qual é o mais caro, qual é o livro mais caro que tem nessa livraria? Também está fácil porque ele está lá no final de uma lista ordenada do menor para o maior, ou também poderíamos inverter isso, seria a primeira posição de uma lista ordenada de maior para menor.
* [03:50] Fica muito mais fácil agora buscarmos elementos. Vale para produtos mais baratos ou mais caros etc. Porém, uma coisa importante, nós formos fazendo aqui, arrastando os nossos livros, meio que com a ajuda da cabeça, mas a forma como vamos ordenar esses elementos quando trabalhamos com código é super importante.
* [04:12] Porque, normalmente, nós trabalhamos com volumes de dados grandes. A nossa livraria é pequena, tem só seis livros de programação, mas se fosse uma livraria especializada, ela teria centenas, milhares de títulos relacionados a programação.
* [04:27] E o que vamos ver a partir de agora é algumas formas de fazer isso, ou seja, alguns algoritmos de ordenação. Vamos ver como trabalhar com eles a partir do próximo vídeo.
* [00:00] Já vimos que, às vezes, vale a pena ordenarmos uma lista para conseguir pegar os dois primeiros, o primeiro, os três primeiros etc. Então, o que vamos fazer agora é construir no código mais ou menos os mesmos passos que fizemos anteriormente usando as nossas "variáveis mentais", digamos assim.
* [00:22] Novamente, o que temos que fazer é traduzir para o JavaScript, para uma linguagem que o computador consegue compreender. No código, a primeira coisa que eu vou fazer é criar um novo arquivo. Vou chamar esse novo arquivo de “selectionSort.js”.
* [00:41] *Sort* é ordenar, em inglês, é um algoritmo de *sorting*, um algoritmo de ordenação. Já vamos ver por que ele tem esse nome legal. Criei meu arquivo “selectionSort.js” e vou importar const livros = require('./listaLivros') e importei o nosso *array* de livros.
* [01:09] Já que estamos trabalhando com a ajuda do nosso site da livraria, vamos lembrar que nós, mentalmente, criamos, para fazer a ordenação, um *loop*, um laço de repetição que percorreu toda a lista e foi ordenando, trocando de lugar os livros mais caros com os livros mais baratos.
* [01:31] Então, o que temos que fazer no nosso arquivo “selectionSort.js” é também criar um for. Vamos continuar trabalhando com esse *for* mais clássico que tem no JavaScript e tem também em várias linguagens de programação. Se você quiser depois pensar em outra linguagem, fica mais fácil de fazer essa transferência.
* [01:47] De novo, vamos trabalhar com for (let atual = 0), começar do começo da nossa lista. Nosso for vai funcionar enquanto atual < livros.length, o tamanho da nossa lista de livros. E vamos incrementar atual++ para percorrer a lista toda.
* [02:18] O que nós fizemos, mentalmente? Nós comparamos valores de livros e vimos qual tinha o menos preço. Nós já temos um código pronto, que foi o que nós fizemos anteriormente e está em “index.js”, para saber qual é o menor preço.
* [02:37] Então, nós podemos pegar o que foi feito anteriormente em “index.js” e extrair isso para uma função para reaproveitarmos, ao invés de simplesmente copiar o *for* que fizemos antes e colar no “selectionSort.js”, não é muito prático.
* [02:52] Vou trocar o nome desse arquivo “index.js” para “menorValor.js”, fica mais fácil de identificarmos. Dentro de “menorValor.js”, vamos criar a nossa função. Vou criar uma function, vou chama-la de menorValor.
* [03:12] Essa função tem que receber um *array*, vou chamar esse parâmetro de arrProdutos, deixar mais genérico qual tipo de produto vai receber, pode receber livros, colchões, chocolates etc.
* [03:27] Essa função tem que rodar a partir de uma posição inicial do *array* para percorrer, então vou dizer que essa função também vai receber uma posição inicial, posicaoInicial, para fazer a comparação, porque lembra que o nosso *for* vai andar do primeiro índice do *array* em direção ao fim.
* [03:52] A posição onde vai ser feita a verificação sempre vai estar modificando, então nós passamos isso através de parâmetro para dentro da função. O código que nós criamos anteriormente, o let maisBarato e o for, nós passamos para dentro da função.
* [04:06] E agora, ao invés de um console.log, a nossa função menorValor vai retornar o valor da variável return maisBarato;, que vai ser um número de 0 a X, que vai ser o número referente à posição no *array* onde vamos encontrar o item mais barato, começando a partir de certo ponto.
* [04:29] Como assim, começando a partir de certo ponto? Porque nós começamos a nossa verificação a partir do índice 0, mas depois que o índice 0 já está ordenado, nós não precisamos mais nos preocupar com ele, podemos começar a nossa verificação na próxima vez a partir do índice 1, depois a partir do índice 2, etc.
* [04:47] Por isso, nós vamos passar o nosso índice inicial por parâmetro e, para cada *loop*, a nossa função vai retornar qual é o índice referente ao produto mais barato.
* [05:06] Vou deletar o console.log que fizemos anteriormente, não precisamos mais dele, e vou exportar, antes que eu esqueça, para o module.exports = menorValor. Agora podemos usar essa função em outras partes do código.
* [05:22] Voltando ao arquivo “selectionSort.js”, no topo do arquivo, junto com a nossa const livros, eu vou criar uma const menorValor = require(‘./menorValor’).
* [05:42] Agora já podemos usar a função nesse arquivo também. Então, temos um *for*, por enquanto ele não está fazendo nada dentro, que começa em uma posição atual que nós colocamos como = 0.
* [05:57] Vamos voltar só no menorValor porque eu esqueci de uma coisa muito importante. Eu criei a função, passei parâmetros para ela, mas, dentro do for, eu não fiz nenhuma modificação para incluir esses parâmetros em nenhum lugar, então já estava esquecendo disso.
* [06:15] Dentro do nosso for não é mais livros porque vamos estar trabalhando com um *array* genérico, então arrProdutos.length, e a mesma coisa dentro do if, arrProdutos na posição [atual] for menor do que arrProdutos[maisBarato]preco.
* [06:46] E agora maisBarato = atual, só que uma coisa muito importante, nós também temos que passar uma posicaoInicial. A posicaoInicial agora vai ser igual a maisBarato.
* [07:01] Por que maisBarato vai ser igual à posicaoInicial? Porque, quando nós começamos – vamos voltar para o nosso site – a verificar, nós só temos o primeiro produto, e o primeiro produto nós não sabemos se é mais barato ou mais caro, ele é onde estamos começando.
* [07:19] Então, se eu tenho só JavaScript como o primeiro elemento do *array*, eu não tenho com o que compará-lo, eu só vou ter como compará-lo a partir do momento que eu começo a percorrer.
* [07:31] E a mesma coisa quando nós passamos a trabalhar com o segundo elemento. O segundo elemento é o PHP, o primeiro nós já sabemos que está ordenado – vou trocar aqui. Então, a mesma coisa, o primeiro que tem no *array*, seja qualquer produto, nós não sabemos se ele é o mais barato ou o mais caro até o momento em que começamos a fazer as comparações.
* [07:58] Então, nós fazemos essa modificação, o maisBarato deixa de ser 0, o maisBarato vai ser a posicaoInicial onde nós começamos a fazer a comparação. Nós começamos no 0, a partir do momento em que já arrumamos o 0, não precisa mais se podemos passar para o próximo.
* [08:16] Agora, a posição inicial é o 1, então vai percorrer o *array* a partir do 1. Depois que trocamos e verificamos que não tem, na posição, nenhum mais barato, nós passamos para o 2, então a posição inicial agora é o índice 2, e vamos do índice 2 para frente, e por aí vai.
* [08:35] Por isso que, agora que nós extraímos o nosso código de menorValor para uma função, nós temos que passar a posição onde ele começa a trabalhar e a posicaoInicial é onde nós começamos, então o maisBarato sempre começa da posicaoInicial, seja 0, seja 1 ou seja 2.
* [08:54] E dentro do for, a mesma coisa. O nosso let atual, ao invés de iniciar a variável com o valor de 0, começa também a partir de posicaoInicial, porque, lembrando como fizemos lá atrás, nós começamos os dois com 0.
* [09:14] Só que agora vamos andar com esse código do 0 para o 1, para o 2, para o 3. Não faz sentido a nossa let atual começar sempre do 0, ela vai começar da posição atual que nós indicarmos.
* [09:27] Agora podemos voltar para o nosso selectionSort e agora estamos fazendo o *loop* que o “selectionSort.js” vai fazer, que o nosso algoritmo de ordenação vai fazer.
* [09:44] Estamos fazendo partes pequenas. Lembra que eu falei do ovo que nós fazíamos um pequeno desvio no algoritmo para fazer uma sequência de passos para quebrar o ovo em um pote para voltar para o bolo? A mesma coisa que estamos fazendo aqui.
* [09:59] Nós extraímos em menorValor e agora vamos usá-lo dentro de outro algoritmo para conseguir ir vendo o menor e ordenando.
* [00:00] Primeira coisa que temos que fazer é verificar, a partir do 0, que é onde estamos começando. Vou desordenar novamente a nossa lista de livros. Estamos começando no 0 e o *for* do nosso algoritmo de ordenação também está começando do 0.
* [00:23] Primeira coisa a fazer é verificar qual é o menor valor do nosso *array* começando com 0. Nós já criamos o código para isso, eu vou executar a função menorValor, criar uma let menor para salvar o retorno dessa função.
* [00:44] Então, let menor, dentro do for mesmo nós vamos criar essa let menor = menorValor, chamamos a função, passamos para ela, como primeiro parâmetro, o *array* que ela está esperando receber, o arrProdutos e aqui é livros, na hora que executarmos, vamos passar livros.
* [01:04] Qual é a posição inicial que queremos começar? Estamos começando o nosso *for* nesse momento, então vamos começar com atual. atual, nesse momento, é 0.
* [01:20] Então, quando a função menorValor for executada nesse momento que a função vai rodar, nós vamos ter a nossa variável livros e o valor de atual vai ser 0. Então, essa função vai ser executada e vai retornar para nós um inteiro de 0 a 5, e esse inteiro vai ser guardado nessa variável.
* [01:53] A variável menor, o tipo de dado que ela vai guardar é um inteiro com o número do índice de 0 a 5 onde vai estar o produto mais barato, em um intervalo que vai, nesse momento, de 0 a 5, porque estamos começando o nosso *loop* nesse momento.
* [02:15] Executamos, isso vai ser salvo. Há dois próximos passos que é criar mais duas variáveis. Então, vou criar uma variável que vou chamar de let livroAtual. Não estou mais guardando o índice, estou guardando a própria informação, que é livros na posição [atual].
* [02:41] Porque lembra que o que nós queremos fazer é trocar as informações de lugar, então tenho que pegar realmente o dado, o objeto que tem lá – chave, valor, nome e preço do livro – e trocar essas informações de lugar.
* [02:59] Já que eu tenho que trocar as informações de lugar, eu preciso guardá-las em uma variável para poder fazer essa troca. Então, a minha variável livroAtual está guardando uma informação, que é livros na posição [atual], livros na posição 0, que, no caso, vai ser um objeto que é o que está na posição 0 do meu *array*.
* [03:23] E eu vou criar mais uma variável que vou chamar de let livroMenorPreco. Aqui vou guardar a informação de qual é o livro que está na posição livros[menor].
* [03:40] Por que posição menor? Porque o retorno da função menorValor vai me voltar um número, uma posição. Ela vai procurar dentro do *array* de livros, vai achar qual é a posição de 0 a 5 que tem o livro mais barato.
* [04:01] E o que eu estou pedindo para o JavaScript fazer é acessar o *array* de livros no índice que está guardado, 0, 1, ou 2 – no caso, sabemos que é o 1 porque estamos vendo o *array* e o JavaScript não vê, ele não sabe.
* [04:20] Então, sabemos que aqui é o 15,00. O que estou pedindo para ser guardado na variável livroMenorPreco é o dado que está guardado no *array* de livros, na posição 1, que é PHP, é um objeto que tem chave e valor, titulo: PHP e preco: 15.
* [04:37] Então, o que nós fizemos nessas duas variáveis não é guardar uma posição, é guardar uma informação, um dado, porque vamos precisar trocar esses dados de lugar no *array*.
* [04:48] A partir do momento que fazemos isso, nós trocamos, fazemos livros[atual] = livroMenorPreco, então estou acessando a posição atual e trocando a informação que tem dentro dela, trocando o dado que tem dentro dela, o objeto.
* [05:10] E a mesma coisa para livros[menor] = livroAtual. Então, estou acessando posições no *array* e trocando as informações que tem dentro dela, trocando os objetos, objeto com chave e valor JavaScript de lugar com o objeto de chave e valor PHP.
* [05:40] Vou trocar de lugar o PHP com JavaScript, e agora, sim, depois de trocado de lugar, nós conseguimos até exibir, no nosso console, o resultado. Então, trocou de lugar, passou por todos os itens do *array*, nós exibimos console.log, posso exibir o *array* inteiro, console.log(livros).
* [06:12] Mas, lembrando que o *for* vai fazer isso, todo esse processo, passar a atual de 0 para 1, a atual passa a ser 1. Então, como o atual passa a ser 1 e o valor dessa variável de início começa do 0 e depois vai trocando.
* [06:36] Como a atual passa a ser 1, o menor valor – lembrando que a função vai ser executada novamente, porque ela está dentro do *for*, só que agora passando com o parâmetro 1 ao invés de 0.
* [06:54] Na prática, o que acontece? O 0 já está ordenado, nós não precisamos mais dele, então a atual passa a ser 1. A posição inicial, onde vai começar a ver o menor valor agora é 1, então do 1 para frente. Ele checa com o 1, com o 2, com o 3, com o 4 e com o 5.
* [07:14] Vamos repetir o mesmo processo aqui, de ver qual é o livroAtual, livros na posição 1, e ver qual é o livroMenorPreco, que é o resultado da nossa função. Ele vai retornar um número, nós estamos guardando as informações, estamos trocando as informações de lugar.
* [07:35] O que nós podemos fazer aqui, para ficar mais legal, é colocarmos mais um console.log no meio para ele dizer para nós qual é o livro atual e qual é o livro menor preço, livros[menor], que é o retorno da função.
* [08:08] Vamos fazer o seguinte, estou fazendo um monte de código aqui e não rodei nada ainda. Vamos rodar isso para ver se está tudo certo, vai que tem algum erro e não vimos até agora.
* [08:20] Vamos lá, node selectionSort.js, vamos ver se está tudo certo, ele tem que exibir alguns console.log para nós. Vamos voltar aqui, ele já exibiu no final a nossa *array* ordenada, 15, 20, 25, 30, 45 e 50.
* [08:41] E ele exibiu também os passos do nosso algoritmo. Ele começa com livro atual { titulo: JavaScript, preco: 25} e o menor preço {titulo: PHP, preco 15}, porque estamos trabalhando com o começo do *array*, que é o índice 0.
* [09:02] Vou pedir mais um console.log para vermos também, que é o atual, console.log(‘posição atual’, atual). Vamos executar novamente, subi aqui tudo. posição atual 0, livro atual { titulo: JavaScript, preco: 25}, o livro com menor preço {titulo: PHP, preco 15}.
* [09:35] Ele vai trocar tudo de lugar, posição atual 0, que é de acordo com o que estávamos vendo aqui. Trocá-los de lugar, “PHP 15,00”, achou, trocou. Agora, a posição atual é 1 porque já estamos para frente do *loop*.
* [09:51] Então, o livro atual é JavaScript, preco:25 porque, no *loop* anterior, nós trocamos essas duas informações de lugar. Então, sim, atual é JavaScript, e ele vai achar o menor preço que é Python, porque é o retorno do nosso algoritmo de menorPreco.
* [10:13] Então, JavaScript achou, Python, trocou os dois de lugar. O JavaScript agora foi lá para frente, e agora posição 2. Agora a nossa posição atual de trabalho é 2. Qual é o 2? É Java, está certo.
* [10:33] O menor preço titulo: JavaScript, preco: 25} que ele achou lá atrás. Então, ele vai trocar o Java de lugar com o JavaScript. Estamos trocando, se voltarmos para o nosso código, as informações de lugar.
* [10:58] Agora já vimos o Java e estamos na posição 3. A posição “3” é “Elixir 50,00”. Vamos ver qual é o menor entre Elixir e Java. Java, preco: 30, está certo, está lá no final do *array*, trocou de lugar Java com Elixir.
* [11:19] Vamos passar para o próximo, agora o nosso *loop* está na posição 4. Vamos ver qual é o livro atual do “4”, é o Go, está certo, e o mais barato é o Go, ele não faz nada.
* [11:33] Passou para o “5”, posição atual 5, o livro atual é o Elixir, preco: 50, e menor preço {titulo: Elixir, só tem um, ele retornou ele mesmo aqui, o Elixir. E, no final, o que temos é o nosso *array* de livros ordenado.
* [11:55] Porque nós passamos as informações, fomos brincando – se voltarmos para o código – de dança das cadeiras e fomos trocando as informações de lugar. Então, trocou a informação que estava em um índice do *array* com outra, e nesses *loops* de troca de lugar, nosso *array* ficou ordenado.
* [12:18] Como o último índice é ele mesmo, ele já está ordenado, ou seja, depois que já fizemos todas as trocas de lugar e o livro mais caro já ficou na última posição, nós, inclusive, poderíamos começar o nosso *for* a partir de 0 e indo até livros.length -1, porque nem precisamos conferir a última posição do *array*, ele para antes, já está ordenado, já está trocado de posição.
* [12:51] Agora, o que nós temos é uma lista de livros ordenada e, se quisermos ver qual é o primeiro mais barato, os dois primeiros mais baratos ou os três primeiros mais baratos, é só pegar a primeira posição do *array*, as três primeiras posições do *array* etc.
* [13:08] E nós também reaproveitamos o código que tínhamos feito anteriormente para criar uma função que, dado determinado *array*, determinada posição inicial, retorna o índice onde está o menor valor.
* [13:21] Esse algoritmo que acabamos de implementar aqui se chama *selection sort*, que foi como eu nomeei o arquivo, ou seja, ordenação por seleção. A ideia central desse algoritmo é sempre encontrar o menor valor e colocar na primeira posição.
* [13:38] Primeira posição que já conversamos a partir de onde estamos trabalhando. E por aí vai. Então, ele começa colocando o menor valor na primeira posição, depois percorre tudo de novo, coloca o segundo menor valor na segunda posição, e por aí vai.
* [13:54] Mas nós conseguimos pensar em outras formas de pegarmos uma lista desordenada ou pegarmos um conjunto de elementos e ordená-los começando a fazer esse processo de uma forma diferente.
* [14:08] Então, é isso que vamos ver nos próximos vídeos.

# 05Para saber mais: O sort() do JavaScript

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98363/next)

Durante este curso estamos criando nossos próprios **algoritmos de busca e ordenação**. Porém, se formos investigar a documentação do JavaScript, descobrimos que o pacote da linguagem já traz, prontos para uso, métodos/funções que buscam ocorrências e ordenam listas, da mesma forma que estamos fazendo. Por exemplo, para varrer arrays em busca de dados específicos, podemos utilizar [includes()](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/includes), [find()](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/find" \t "_blank) ou [filter()](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/filter" \t "_blank), entre outros, e para ordenar arrays podemos usar o método [sort()](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/sort" \t "_blank).

Mais do que pensar “por que não usamos o método pronto do JavaScript ao invés de escrever outro do zero?” vamos olhar por outro ângulo: se o JavaScript tem um método/função de ordenação “pronto para usar”, esse método com certeza utiliza algum **algoritmo de ordenação** para fazer esse trabalho. E qual seria ele? É o mesmo que utilizamos?

A verdade é que o sort() utiliza mais de um algoritmo de ordenação, dependendo de principalmente duas coisas: 1) as características do array a ser ordenado e 2) a implementação do JavaScript que está sendo utilizada.

Como assim, implementação?

O JavaScript se desenvolveu como uma linguagem interpretada pelos navegadores, e a partir da especificação do que cada versão da linguagem deve seguir (definida pelo [ECMA](https://www.ecma-international.org/publications-and-standards/standards/ecma-262/)) cada navegador implementa os métodos e funções de acordo com suas próprias engines, ou seja, cada navegador tem seu próprio “motor” de interpretação do JavaScript, e a forma como o código é interpretado “por baixo dos panos” em cada navegador pode ser diferente.

No caso do método sort(), a engine de interpretação do JavaScript implementada pela Mozilla no navegador Firefox utiliza um algoritmo chamado Merge Sort, enquanto o Google utiliza, na engine do Chrome (chamada de V8) alterna entre outros dois algoritmos de ordenação, o Quick Sort e o Insertion Sort, dependendo do caso. Existem vários algoritmos de ordenação!

No caso do NodeJS, uma vez que utiliza como base a engine V8 do Google, os algoritmos utilizados no sort() também serão os mesmos.

E que casos são estes? O método sort(), executado sem nenhum parâmetro, interpreta todos os elementos do array como strings e ordena em ordem alfabética crescente, a partir da tabela [Unicode](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Grammar_and_types). Para outros tipos de ordenação, por exemplo numérica ou decrescente, é preciso passar parâmetros de comparação para o sort().

A [documentação do MDN sobre o sort()](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) tem vários exemplos sobre os diversos casos de ordenação e como utilizar este método.

# Faça como eu fiz: Usando loops do JavaScript

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98365/next)

Durante o curso vamos usar o for para escrever os laços de repetição. Esta forma de escrever laços é muito comum em linguagens de programação e pode ser encontrada praticamente da mesma forma tanto no JavaScript quanto em outras linguagens muito utilizadas, como o Java e a família C (C, C++, C#, entre outras). Assim como o for, a maior parte dessas linguagens, além de várias outras, também utiliza os laços while e do… while.

O JavaScript tem outras formas de escrever laços de repetição além desse for “clássico” e podemos reescrever o nosso código para testar com um método mais moderno, o forEach().

forEach() é um [método de array do JavaScript](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) que percorre o array e executa um bloco de código para cada índice.

Este método pede como parâmetro uma função que chamamos de **função callback**. Esta função callback controla o laço para que percorra cada um dos elementos do array, através do parâmetro que estamos chamando de livro (afinal de contas, trata-se de um array de livros):

livros.forEach((livro) => {

// código aqui

})COPIAR CÓDIGO

Também vamos precisar do número do índice de cada elemento, para fazer a troca de lugares entre os valores. Para isso vamos precisar do segundo parâmetro da função callback, que é opcional, e que leva justamente o número do índice atual que está sendo percorrido. Ou seja, é o equivalente à nossa variável atual:

livros.forEach((livro, indice) => {

// código aqui

})COPIAR CÓDIGO

Agora podemos passar para dentro do bloco a lógica do algoritmo, substituindo atual por indice:

livros.forEach((livro, indice) => {

let menor = menorValor(livros, indice)

let livroAtual = livros[indice];

let livroMenorPreco = livros[menor];

livros[indice] = livroMenorPreco

livros[menor] = livroAtual

})COPIAR CÓDIGO

Se examinarmos o laço, vemos que o primeiro parâmetro da função callback, livro, não está sendo utilizado; nesse caso, podemos usar a notação do JavaScript para informar que o primeiro parâmetro não será utilizado no código, através do \_:

livros.forEach((\_, indice) => {

let menor = menorValor(livros, indice)

let livroAtual = livros[indice];

let livroMenorPreco = livros[menor];

livros[indice] = livroMenorPreco

livros[menor] = livroAtual

})

console.log(livros)COPIAR CÓDIGO

Agora é só testar com node selectionSort.js.

# O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98366/next)

## Nessa aula, você aprendeu

* Que, da mesma forma que criamos “algoritmos mentais” para encontrar o menor valor de uma lista, fazemos um processo parecido para ordenar uma lista;
* Como podemos abstrair os passos que fazemos “de cabeça” para código, observando a sequência dos passos utilizados;
* Que, para resolver um problema, muitas vezes utilizamos mais de um algoritmo; para resolver isso, criamos a função menorValor() para reaproveitar o código junto com o Selection Sort.
* [00:00] Continuando a pensar em formas que nós utilizamos a ordenação no nosso dia a dia mesmo sem perceber, nós fazemos várias ordenações. Vimos o selectionSort, que é um algoritmo que transcreve, que traduz um tipo de ordenação que nós também fazemos sem perceber na cabeça quando precisamos ordenar coisas.
* [00:22] Vamos, então, abstrair um pouco a nossa livraria e pensar nela como uma livraria física. Temos algumas estantes para preencher, ou a estante de casa mesmo, tem uma pilha de livros para colocar em uma estante, nós costumamos usar alguma ordenação.
* [00:36] As livrarias, normalmente, usam a ordem alfabética pelo sobrenome da autora ou autor. Vamos fazer um pouco diferente, vamos supor que, na estante da nossa livraria, nós queremos ordenar esses livros por ordem alfabética dos títulos.
* [00:50] Temos uma pilha de livros, vamos pegar o primeiro livro meio que a esmo. Peguei o primeiro livro, o primeiro livro da pilha é o livro de PHP. Eu o coloco na estante, só tem ele, não tenho o que ordenar.
* Aí eu pego o segundo livro, o segundo livro na pilha é um livro de Go, então, se eu ordenar em ordem alfabética, eu já tenho de PHP na estante, no canto da estante, vou dar uma empurrada no livro de PHP, afinal de contas, "P" vem depois de "G", e vou inserir o livro de Go antes do livro de PHP.
* [01:21] Podemos pegar o próximo livro da pilha, o livro de Java. Dou uma olhada, tenho PHP, tenho Go, então, entre esses dois livros na estante, eu vou dar uma afastada no PHP e colocar entre os dois que já tinham o livro de Java – "G", "J", "P".
* [01:39] E a mesma coisa vamos fazendo para cada livro que pegamos na nossa pilha. O próximo livro é um livro de JavaScript. Go, Java, PHP, afasto o PHP de novo, abro um espaço, coloco o de JavaScript, e o próximo na pilha é Python.
* [01:54] Então, Go, Java, JavaScript, PHP, Python com "Py", vai para o final da lista, não preciso empurrar nada, só coloco o Python no final. E agora o último livro que sobrou da pilha, que é o Elixir.
* [02:11] Tenho Go, Java, JavaScript, PHP, Python. O livro que começa com "E" entra antes de todos, então vou empurrar todo mundo um pouco, vou abrir um vão, abrir um espaço, empurrar todos para a direita e colocar o livro de Elixir no começo da minha estante.
* [02:31] Agora tenho uma estante, uma lista de livros ordenados alfabeticamente, só que ao invés de nós, dessa vez, termos uma lista e ir trocando de lugar, como fizemos anteriormente, nós fomos inserindo cada livro no seu lugar, afastando os posteriores – por exemplo, se vou inserir Java, eu afasto os posteriores, JavaScript para frente.
* [02:57] E checando se os anteriores realmente estão no lugar. Se eu coloco o Java antes de Go, tenho que confirmar se o de Go está certo, ele tem que ser antes mesmo, e se o de Elixir está certo e tem que ser antes mesmo. Agora está tudo certo.
* [03:13] Agora, o que nós fizemos foi, na nossa cabeça novamente, um outro tipo de ordenação que podemos transformar também em um algoritmo, seja por ordem alfabética, seja por ordem numérica, o que for, nós temos um quesito que vamos usar para ordenar.
* [03:31] Eu usei aqui como alfabética só para termos uma ideia melhor, uma coisa que nós geralmente fazemos em estantes de livros ou procuramos em livrarias dessa forma, mas podemos continuar trabalhando com ordem numérica para termos livros mais baratos.
* [03:45] Como o computador só entende condições, *for*, *if*, *while* etc., o que temos que fazer agora é passar esse algoritmo para a linguagem que o computador entende, a partir do que fizemos aqui. Então, vamos ver como fazer isso no próximo vídeo.
* [00:00] Agora que já criamos mentalmente um outro algoritmo, uma outra forma de ordenar, ordenamos a nossa estante de livros por ordem alfabética, vamos dar uma olhada em como ficaria esse tipo de ordenação passado para o JavaScript, porque ele é um pouco diferente do que fizemos anteriormente.
* [00:17] No selectionSort, para cada índice, para cada item que nós analisamos, tem que percorrer toda a lista até o final e olhar se troca algum item ou não. Então, se a lista tiver mil itens, vai até o final dos mil itens.
* [00:34] E o que nós fizemos agora, quando ordenamos a nossa estante de livros, foi um pouco diferente. Nós inserimos o elemento no espaço onde ele deveria ficar analisando o que vinha antes, se estava na posição correta, e o que vinha depois.
* [00:54] Então, o que vamos fazer agora é examinar, por exemplo, o Java. Se voltarmos a ver a nossa estante, que agora está ordenada alfabeticamente, mas se voltarmos a ver em relação aos valores, se pegarmos o livro de Java, por exemplo, e analisarmos de acordo com o item anterior, que é Go, Java custa 30,00, Go custa 45,00.
* [01:16] Se fossemos ordenar do menor para o maior, Java estaria antes do livro de Go. Um custa 45,00 e o outro custa 30,00. Analisando novamente Java com relação a Elixir que custa 50,00, também Java tem que estar antes de Elixir. Então, nós continuamos o trocando de lugar.
* [01:38] O que estamos fazendo agora é percorrendo a nossa lista de uma forma um pouco diferente. Vamos dar uma olhada como isso ficaria no código. Vou voltar na pasta do projeto, criar um novo arquivo, vou chamar de “insertionSort.js”.
* [02:00] insertionSort porque, acho que você deve ter suspeitado, isso que nós fizemos enquanto ordenávamos a nossa estante de livros é um outro algoritmo de ordenação e ele tem um nome, ele já existe, já foi pesquisado.
* [02:15] Nós chamamos esse algoritmo de insertionSort porque vamos inserindo os elementos na posição correta com relação ao que vem antes dele. Então, 1, 2, 3, insere o 3 no lugar, e por aí vai.
* [02:29] A primeira coisa que temos que fazer no nosso arquivo é criar uma const, vou chamar de const livros e trazer para dentro dessa const o *array* que estamos trabalhando, require(‘./listaLivros’). Já temos o *array* para trabalhar.
* [02:46] Para ficar um pouco mais organizado, vamos criar uma função para colocar o algoritmo que vamos escrever dentro, porque fica mais fácil de reaproveitar código, fica tudo mais organizado.
* [02:57] Vou criar uma function, vou chamá-la de insertionSort, e ela vai receber como parâmetro, por enquanto, vamos passar só para ela arr de *array*, ou podemos passar para ela uma lista para ficar mais fácil de falarmos.
* [03:14] Ela vai receber uma lista que pode ser qualquer lista que queiramos ordenar, vou deixar mais genérico. Dentro dessa função, vamos já colocar um console.log só para exibir no console a nossa lista no final, quando tivermos completado o nosso código.
* [03:35] Voltando para a nossa estante de livros, o que vamos fazer é percorrer uma lista. Então, se vamos percorrer uma lista, podemos continuar usando o for, que é o que estávamos usando anteriormente.
* [03:48] Vou criar um for, começar declarando a let atual = 0, porque, novamente, vamos analisar item por item, sempre vai ter que ter o item atual, o item da vez, e vamos começar com 0 para começar no começo do *array*.
* [04:04] O nosso *loop* vai funcionar enquanto a atual for menor do que lista.length, o comprimento da lista. E terminamos a condição com atual++ para incrementar o valor da variável atual e ir percorrendo a lista até o fim.
* [04:25] Para ficar um pouco mais fácil de visualizarmos o que estamos fazendo, vamos criar uma variável para olharmos exatamente qual é o item que está sendo analisado por vez.
* [04:39] Eu vou chamar de let analise = atual porque nós vamos mover a nossa análise de item por item a partir do índice 0 do nosso *array*. Então, a partir desse item, temos que examinar o que vem antes dele.
* [05:01] Vamos voltar novamente à nossa lista e pensar como nós fizemos esse algoritmo de ordenação na nossa cabeça. Podemos dizer que, dado o nosso item atual, o item que estamos analisando, queremos que, enquanto o valor do item que estamos analisando for menor – estamos ordenando do menor para o maior – do que o que vem antes, nós trocamos de lugar.
* [05:32] Então, Go que custa 45,00 estava à direita do Elixir que custa 50,00, nós os trocamos de lugar, e por aí vai. Enquanto uma condição é verdadeira, nós podemos usar o outro tipo de *loop* que é o *while*, que significa justamente "enquanto".
* [05:53] Enquanto dada condição for verdadeira, vamos executar o que estiver dentro do bloco de código. E qual é a condição que queremos que aconteça? Então, enquanto, while (lista[analise].preco), não pode esquecer do .preco, porque tem que entrar em cada um dos objetos e pegar o preço, que é o número que estamos usando para comparar.
* [06:16] Enquanto o preco for menor do que qual? Voltando para a nossa estante, enquanto o valor do item atual for menor do que o item que vem antes. Então, se o item que estamos analisando está na posição X do *array*, tem que ser na posição X menos 1.
* [06:42] Então, o nosso while (lista[anlise].preco <lista[analise -1].preco). Essa vai ser a nossa condição.
* [06:57] Essa condição sendo verdadeira, vamos abrir o bloco e dizer o que queremos que aconteça. O que queremos que aconteça quando vermos que o valor do item que estamos analisando é menor do que o que vem antes? Queremos que eles troquem de lugar.
* [07:11] Para trocar de lugar, nós já criamos o código anteriormente quando fizemos o selectionSort, então podemos revisitar o selectionSort e pegar as linhas referentes a essa troca que já criamos.
* [07:27] Já estamos fazendo uma repetição de código aqui, então acho que, no futuro, podemos extrair essa lógica de troca de lugar para uma função. Por enquanto, vamos fazer por extenso, depois podemos criar uma função para isso.
* [07:43] Agora que já copiamos as quatro linhas do selectionSort, só precisamos revisar e dar uma arrumada, porque estávamos, no código anterior, nos referindo a livros, mas agora, como estamos usando uma função um pouco mais genérica, vamos dar uma melhorada nesses nomes de variáveis.
* [08:01] Então, vou chamar a primeira variável, que é a que vai guardar o valor, o objeto que está sendo analisado, de itemAnalise = lista[analise].
* [08:15] Lembrando, o que estamos fazendo aqui é acessando a lista na posição que estamos analisando no momento e guardando em uma variável os dados. Os dados aqui é o objeto que tem o nome do livro e o preço dele.
* [08:30] A mesma coisa vamos fazer para o anterior. Vou chamar a variável de let itemAnterior = lista[analise – 1], e agora podemos trocar novamente de lugar. Vou trocar lista[analise] pelo valor de itemAnterior, e vou trocar lista[analise -1] por itemAnalise.
* [09:11] Uma coisa importante sobre a diferença, por exemplo, entre o *for* e o *while*, é que, ao contrário do *for* onde a condição de parada já está dada entre as outras condições, no caso do *while*, temos que informar a condição de parada do *loop* no bloco de código, porque senão entramos em um *loop* infinito.
* [09:31] Então, quando nós queremos que esse código pare de funcionar? Nós estamos voltando, então passamos analise-- para dentro desse *while* que ele vai analisando e vai voltando.
* [09:53] Isso aqui vai voltar até o momento em que essa condição do while não for mais verdadeira. Quando essa condição deixar de ser verdadeira, quer dizer que já chegamos no lugar da lista, do *array*, onde o elemento tem que ficar. Então, aí ele vai sair do *while*.
* [10:13] Nosso console.log já está no lugar, vamos executar, vamos rodar esse código porque já escrevemos bastante sem testar. Vou chamar, no fim do arquivo, a nossa função insertionSort, passando para dentro dela livros, que é o *array* que nós importamos na primeira linha do nosso código.
* [10:33] Vou abrir o Terminal, node insertionSort.js – vou só puxar o Terminal, minha janela está muito pequena –, rodei node insertinSort.js e tem um erro no meu código, deixa eu ver o que aconteceu.
* [10:54] Aconteceu um erro no *while*, ele reclamou da nossa comparação de lista[analise] com lista[analise - 1]. Sim, está certo, tem um erro no código que eu deixei passar.
* [11:08] Vamos voltar para a nossa estante. Se vamos rodar esse código pela primeira vez, o primeiro item a ser analisado é o item do índice 0, então a análise vai ser igual a 0. A análise é igual ao atual, o atual está começando no 0.
* [11:24] Se tirarmos -1 de 0, o JavaScript vai tentar achar a posição -1 no *array*, e a posição -1 não existe, o *array* começa no 0. Então, dentro do *while*, vamos passar uma condição a mais.
* [11:37] Vamos passar while (analise > 0 &&, passando esse duplo "E comercial" significa que só vai entrar no *while* se ambas as condições forem verdadeiras – se a analise for maior do que 0 e a outra condição, que é a condição de comparação dos valores.
* [12:03] Aí, sim, ele vai entrar no *while*. Isso é para, na primeira vez, ele passar reto pelo *while*, não entrar. Quando atual for atualizado para 1, aí, sim, atual = analise, analise vai ser igual a 1, 1 maior que 0, aí ele vai entrar no *while*.
* [12:21] Vamos rodar novamente e ver se o nosso erro foi embora. Puxar meu Terminal aqui para cima, node insertionSort.js, e agora ele retornou um *array* totalmente arrumado por valor.
* [12:37] PHP, preco: 15, Python, preco: 20, JavaScript, preco: 25, Java, preco: 30, Go, preco: 45, Elixir, preco: 50. Agora tudo funciona certo. O que nós fizemos aqui?
* [12:50] Pegamos os passos que nós pensamos no algoritmo quando fizemos a ordenação por ordem alfabética e vimos que é diferente da ordenação anterior, e só passamos para o JavaScript, por uma forma que conseguimos fazer com que o código execute para nós.
* [13:07] Caso fique alguma dúvida sobre como funcionou esse *while* dentro do *for*, trocamos coisas de lugares, vamos fazer mais um teste de mesa para ver isso funcionando passo a passo, variável por variável. Vamos ver no próximo vídeo.
* [00:00] O algoritmo em si já está funcionando, nós conferimos, mas vale a pena nós conferirmos os passos de uma forma mais passo a passo, mais pontual. O que vamos fazer, usando a tela, é o que chamamos de teste de mesa, de *desk checking*.
* [00:19] Quando costumamos fazer isso no papel, que, no caso do JavaScript, é interpretar o código passo a passo, anotando o que tem dentro de cada variável, em cada momento de cada linha de cada passo do *loop*. Então, vamos lá que vamos ver esse algoritmo em ação.
* [00:38] Nós começamos com o nosso *for*, nosso *for* começa com let atual = 0, vai rodar enquanto lista.length for maior do que atual, então temos uma lista que vai de 0 a 5.
* [00:51] E vai ter o incremento. A primeira coisa que nós fizemos no nosso código foi criar uma variável que chamamos de analise e passamos para ela o mesmo valor de atual.
* [01:02] Como nós criamos essa variável e atribuímos valor a ela dentro do *for*, cada vez que o *for* rodar, a atual vai ser de 0 para 1, de 1 para 2, de 2 para 3. A analise também vai ser recriada com esses valores.
* [01:19] Então, analise começa com 0, depois, no próximo, o *loop* vai ser 1, vai ser 2, vai ser 3, e por aí vai. Nesse momento, analise = 0 e o nosso *loop* entra enquanto a analise for maior do que 0.
* [01:33] Analise, nesse momento, é 0, então já não entraria no *while*, porque com o operador de &&, só entra no *while* se ambas as condições forem verdadeiras.
* [01:46] Já não entraria, mas, se quisermos dar uma olhada na outra condição, a condição é se lista na posição analise, ou seja, lista na posição 0 – que é, por enquanto, o livro de JavaScript – for menor do que lista na posição [analise -1].
* [02:04] [analise -1], se fossemos seguir esse código, seria -1, não entraria no *array* porque o *array* começa no 0. O que significa isso na prática é que o primeiro elemento não tem com o que comparar porque não tem nada à esquerda dele, então passa reto.
* [02:22] Não entrou no *while*, vamos voltar para o começo do *for*. agora a nossa let atual vai ser atualiza de 0 para 1, isso significa que analise também vai ser atualizada de 0 para 1.
* [02:38] Próxima linha *while*. Enquanto analise for maior do que 0. analise é maior do que 0 agora? É, porque o JavaScript já atualizou. E && lista [analise].preco. Qual é o preço do elemento que está em lista na posição analise?
* [03:02] Agora é lista na posição 1. O valor dele é 15 enquanto ele for do que o elemento que está em lista na posição [analise – 1]. [analise – 1] é 0, já estava 0, .preco.
* [03:20] Qual é o preço do elemento que está na posição 0? É 25,00, então ambas as condições são verdadeiras, porque 15,00 é menor do que 25,00. O que está na posição 1 é menor do que o que está na posição 0, então vai entrar no *while*.
* [03:36] O que está acontecendo no *while*? Está salvando o objeto dado do elemento que está na lista analise, ou seja, que está no índice 1, o PHP, salvando dentro da variável.
* [03:55] Então, dentro da variável itemAnalise agora vai ter PHP 15, aquele objeto. A mesma coisa embaixo, nós criamos uma let, chamamos de itemAnterior que está salvando o que tem na lista posição 0.
* [04:16] Então, aqui, dentro da let itemAnterior, vamos ter um objeto que é nome: JavaScript, valor 25. Fizemos essa parte e agora as outras duas linhas são para trocar de lugar.
* [04:33] Agora, o que tem em lista na posição analise, ou seja, lista na posição 1, é o que nós salvamos em itemAnterior. Agora, o elemento que está na posição analise, ou seja, na posição 1, é o JavaScript, e a mesma coisa na linha de baixo.
* [04:51] lista na posição analise -1, que é 0, vai guardar o objeto do item que estava sendo analisado, então vai guardar PHP. Nós terminamos o *while* fazendo a analise-- se ganhar o valor de 0, estamos dando analise--, estamos decrementando.
* [05:14] Então, analise volta a ser 0 e ele vai sair do *while*, não vai mais entrar, porque só entraria no *while* se analise fosse maior do que 0. Saiu do *while*, agora nós passamos o nosso *for*, agora o nosso atual, que estava valendo 1 anteriormente, vale 2.
* [05:34] analise = atual, agora analise tem um valor de 2. analise – 1 já vai ter valor de 1 porque 2-1 = 1. Vamos entrar no *while*. analise é maior do que 0? É, porque analise é 2, e agora nós analisamos o elemento que está na lista.
* [05:57] Então, lista na posição [analise], ou seja, lista na posição 2, que é Java, o valor dele é menor do que o elemento que está em lista[analise -1], que é 1? Não, porque Java, na posição 2, é 30,00, e JavaScript, que está na posição 1, é 25,00.
* [06:19] Então, já não deu a condição, é falso, não vai entrar no *while*. Voltando para o *for*, agora a nossa let atual vale 3. A let analise vai ser reiniciada de novo com o valor de 3, que é igual à atual, e agora nossa variável analise -1 é 2.
* [06:43] Passamos a fazer a mesma coisa, analise é maior do que 0? É, então entra no *while*, vamos analisar os valores. lista na posição analise, ou seja, lista na posição 3, Elixir, o preço dela é menor do que o elemento que está em lista na posição 2, que é a variável analise -1?
* [07:03] Não, porque Elixir é 50,00 e Java é 30,00, então já deu falso, não vai entrar no *while*, vamos atualizar o nosso *for* para 4. Agora let atual é 4, ou seja, a nossa let analise, de novo, nós iniciamos a variável com o valor igual à atual, ou seja, 4.
* [07:25] analise -1 agora passa a ser 3, vamos ver o *while* de novo. analise é maior do que 0, essa condição está ok, e lista na posição analise, ou seja, lista na posição 4, o preço dela é menor do que em lista na posição 3, que é analise – 1?
* [07:44] Sim, porque Go é 45,00 e Elixir é 50,00, então nós os trocamos de lugar. Trocamos de lugar, a analise agora vale 3 e analise -1 vale 2.
* [08:04] Vamos voltar para o *while*. Lembrando que a condição aqui provavelmente continua verdadeira, vamos ver. analise é maior do que 0, continua sendo *true*, e de novo, lista na posição analise, ou seja, lista na posição 3, 45,00, o preço é menor do que lista na posição 2, que é analise – 1, que é Java?
* [08:27] Não, porque Go é 45,00 e Java é 30,00. Já deu falso aqui, não entramos mais no *while*. Aumentamos, incrementamos a nossa let atual, agora a análise passa a valer 5 porque nós reiniciamos essa variável com um novo valor, e analise -1 passa a valer 4.
* [08:51] Vamos agora, de novo, para o *while*. A primeira condição está ok porque analise é maior do que 0; a segunda condição lista na posição analise, ou seja, lista na posição 5, o valor é menor do que lista na posição 4, que é analise – 1?
* [09:09] Sim, é verdadeiro, porque Python custa 20,00 e é menor do que Elixir que custa 50,00. Então, vai entrar no *while*, vai trocar esses dois de lugar. Trocou de lugar, decrementou o valor da variável analise, então a variável da analise vale 4 e a variável analise -1 vale 3.
* [09:37] Vamos ver se vai entrar no *while* de novo. analise é maior do que 0, sim; lista na posição analise – qual é a posição analise agora? é 4 – é menor do que lista na posição 3?
* [09:49] Sim, continua sendo menor porque Python 20,00 custa menos do que Go que custa 45,00. Vai entrar no *while* de novo porque ambas as condições são verdadeiras, então vai trocar de lugar e vai, por último, decrementar novamente o valor da variável analise.
* [10:05] Que agora é 3, e analise – 1 agora é 2. Voltando para o nosso *while*, vai entrar, a primeira condição é verdade, analise é maior do que 0, vamos analisar de novo.
* [10:18] lista na posição 3, que é a nossa variável analise atual, o preço é menor do que na posição lista 2, que é a analise – 1? Sim, Python 20,00 é menor do que Java 30,00. Vai trocar de lugar, deixa eu pegar o Java – ele não quer colaborar, não quer sair do lugar, mas vai sair.
* [10:50] Saiu, troquei de lugar o Java com o Python. Então, fiz as quatro linhas do código que trocam de lugar, analise--, vou decrementar de novo o valor da variável analise, 2, e analise -1 agora vale 1.
* [11:05] Voltamos para as condições do *while*. analise continua sendo maior do que 0, lista na posição analise, ou seja, lista na posição 2, o preço é menor do que lista na posição 1, que é analise -1?
* [11:19] Sim, porque 20,00 Python é menor do que JavaScript 25,00. Vamos trocando isso de lugar. Trocamos de lugar, decrementamos a variável analise, que agora vale 1, e a analise -1 vale 0.
* [11:32] Voltamos a analisar as condições do *while*. Vai entrar porque analise 1 continua sendo maior que 0, e a nossa lista na posição 1, o valor dela agora é menor do que lista na posição 0?
* [11:49] Agora não, porque Python que custa 20,00 não é menor do que PHP que custa 15,00. Então, não vai entrar no *while*, não vai acontecer mais nada. Voltamos para o *for*.
* [12:03] Agora a nossa let atual vai virar 6 e atual é maior do que o tamanho da lista? A lista tem seis elementos e atual é igual a 6, então não é maior, é igual. Essa condição barrou o *for* porque essa condição é falsa.
* [12:26] Então, saiu do *for*, parou de executar ambos os *loops*, o *for* e o *while*, e temos aqui no final uma lista ordenada. Inclusive, vemos aqui que dá para começar o nosso código – vou voltar para o código.
* [12:45] Daria para começar com let atual = 1 e não precisaríamos da primeira condição de analise > 0 &&, porque começaríamos já da primeira posição do *array* comparando com a posição anterior. Se quiser, também dá para fazer dessa forma.
* [13:06] Então, o que nós fizemos aqui? Esses testes de mesa, os *desk checks*, são super úteis, vimos aqui quando temos que realmente ter certeza de qual é o valor de uma variável em cada pequeno passo do nosso algoritmo.
* [13:24] Então, é como executar um problema em um papel. Você pode, inclusive, ir anotando o valor das variáveis no papel e seguindo esse mesmo fluxo.
* [13:32] Às vezes, quando temos esses algoritmos que trocam alguma coisa de lugar ou criam variáveis temporárias, vale a pena nós fazermos esse tipo de teste para irmos debugando erros que podem acontecer ou mesmo para entendermos com calma o que está acontecendo. Então, o que tem na variável nesse momento, não estou entendo, coloca um monte de console.log na tela.
* [13:54] Às vezes vale a pena nós passarmos passo a passo em um papel mesmo para entendermos o que está acontecendo no algoritmo que, de repente, vimos pronto e não conseguimos pegar de cara todas as trocas de lugar, as trocas de valores, trocas de índice, trocas de posição no *array*, um monte de coisa acontecendo.
* [14:14] Então, o que temos agora? Temos dois algoritmos que fazem essencialmente a mesma coisa, ambos pegam uma lista desordenada e ordenam essa lista com base em algum dado. No caso, estamos ordenando com base em um valor numérico, do menor para o maior.
* [14:30] Nós selecionamos quem tem que ficar em cada índice da lista, encontramos o índice certo e vamos inserindo os elementos dentro dele ou trocando tudo de lugar.
* [14:40] Qual é o melhor? Eu escolho o selectionSort, escolho o insertionSort? E como escolhemos qual é o melhor algoritmo e o que é, o que caracteriza ser o melhor algoritmo? Isso é o que vamos ver em seguida.

# Para saber mais: Testes de mesa

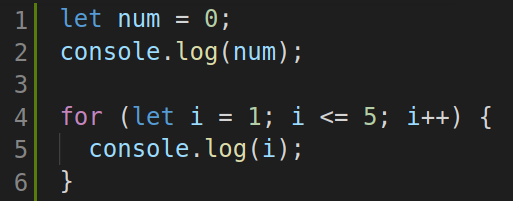
[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98368/next)

Comentamos durante a aula sobre **testes de mesa**, ou desk checking, em inglês.

Um teste de mesa é justamente o processo manual de verificar todas as linhas de um código e executá-lo passo a passo com ajuda de, por exemplo, papel e lápis para anotar qual os valores das variáveis em cada passo de cada linha. É como compilar/interpretar um bloco de código usando o cérebro como compilador/interpretador.

Apesar de hoje as [IDEs](https://www.alura.com.br/artigos/o-que-e-uma-ide" \t "_blank) e as linguagens terem ferramentas práticas, modernas e eficientes para avisar, “debugar” e encontrar erros de código, o teste de mesa é uma forma muito boa de se estudar lógica de programação e para entender o que acontece em cada passo de execução do código, especialmente no caso de algoritmos que envolvem laços de repetição, substituição de valores e variáveis temporárias, como vimos durante a aula.

Uma forma muito comum de fazer o registro de valores de variáveis durante um teste de mesa é utilizando tabelas de rastreio, ou trace tables. Por exemplo, um teste de mesa para um for que imprime números de 0 a 5 no console poderia ser representado pelo seguinte teste de mesa/tabela de rastreio:



| **linha** | **num** | **i** | **saída** |
| --- | --- | --- | --- |
| 1 | 0 |  |  |
| 2 |  |  | 0 |
| 4 |  | 1 |  |
| 5 |  |  | 1 |
| 4 |  | 2 |  |
| 5 |  |  | 2 |
| 4 |  | 3 |  |
| 5 |  |  | 3 |
| 4 |  | 4 |  |
| 5 |  |  | 4 |
| 4 |  | 5 |  |
| 5 |  |  | 5 |
| 4 |  | 6 |  |

Nos seus exercícios e testes, a tabela pode tranquilamente ser feita com papel e lápis, se preferir.

O importante no teste de mesa (com ou sem uso da tabela) é percorrer cada linha e executar realmente o código, anotando as criações/reatribuições de variáveis, resultados de operações matemáticas (se houver), alterações em arrays e objetos, etc. Dessa forma, resultados não esperados na execução do código já vão aparecer na hora.

Pratique com os códigos que já desenvolveu!

# 07 Faça como eu fiz: Extraindo funções

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98370/next)

Durante a aula, utilizamos duas vezes a seguinte estrutura de código:

let itemAnalise = lista[analise];

let itemAnterior = lista[analise - 1];

lista[analise] = itemAnterior

lista[analise - 1] = itemAnaliseCOPIAR CÓDIGO

Repetição de código quase sempre é sinal de que o trecho repetido poderia ser extraído para uma função, assim pode ser reaproveitado. Vamos ver como podemos fazer isso:

Vamos começar criando um novo arquivo, que vamos chamar de troca.js, e dentro dele a função:

function troca(lista, analise) {

// código aqui

}COPIAR CÓDIGO

A função troca() vai fazer toda a lógica de substituição de valores dentro de insertionSort(), então vamos trazer de lá o código correspondente e ver o que precisamos refatorar:

function troca(lista, analise) {

let itemAnalise = lista[analise];

let itemAnterior = lista[analise - 1];

lista[analise] = itemAnterior

lista[analise - 1] = itemAnalise

}COPIAR CÓDIGO

Veja que a função troca() está fazendo alterações direto na lista que está sendo passada por parâmetro; assim, essa função não precisa retornar nenhum valor.

Sem esquecer de exportar a função no final do arquivo, para que possamos importá-la em insertionSort.js:

function troca(lista, analise) {

let itemAnalise = lista[analise];

let itemAnterior = lista[analise - 1];

lista[analise] = itemAnterior;

lista[analise - 1] = itemAnalise;

}

module.exports = troca;COPIAR CÓDIGO

Agora, no arquivo insertionSort.js, começamos importando a função troca() no topo do arquivo:

const troca = require('./troca');COPIAR CÓDIGO

E todo o trecho que foi passado para a função troca() pode ser substituído pela chamada da função:

function insertionSort(lista) {

for (let atual = 0; atual < lista.length; atual++) {

let analise = atual;

while (analise > 0 && lista[analise].preco < lista[analise - 1].preco) {

troca(lista, analise);

analise--

}

}

console.log(lista);

}COPIAR CÓDIGO

Veja que estamos passando como parâmetro da função troca() o array que vamos trabalhar (lista, o mesmo parâmetro que passamos como parâmetro da função externa, insertionSort()) e a variável analise. Dessa forma, a cada laço de repetição feito com o for, de 0 ao fim do array, o valor da variável analise também será atualizado, e este novo valor é passado como parâmetro da função troca() também a cada laço.

Agora você pode fazer o mesmo processo para o código do algoritmo selectionSort que criamos anteriormente, e substituir o código pela chamada da função troca().

# 08 O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98371/next)

## Nessa aula, você aprendeu

* Que um problema pode ser resolvido com algoritmos diferentes, por exemplo, um problema de ordenação resolvido com Selection Sort ou Insertion Sort;
* Que, apesar do resultado final ser o mesmo, o processo (ou passos lógicos) de cada um dos algoritmos é diferente;
* A criar funções para melhor reaproveitamento de código, como para fazer o Insertion Sort receber uma lista de produtos diversos;
* A praticar mais com testes de mesa e ver o código funcionando passo a passo, e como isso pode nos ajudar a entender melhor como o código de um algoritmo funciona.
* [00:00] Por que temos que comparar algoritmos? Quando nós analisamos problemas de programação, sempre temos que pensar em performance, ou seja, quão rápido é um algoritmo, quanto de processamento de computador ele vai consumir.
* [00:17] E como fazemos essas comparações com os algoritmos que trabalhamos? Nós trabalhamos com dois algoritmos que já existem, são bem famosos, *SelectionSort* e *Insertion Sort*, mas existem muitos outros, só falando de algoritmos de ordenação, porque tem vários tipos de algoritmos para resolver vários tipos de problema.
* [00:37] Alguns são mais rápidos, outros consomem mais processamento, outros menos processamento, o importante é nós analisarmos, porque a quantidade de operações que um algoritmo faz para chegar no resultado faz muita diferença.
* [00:51] Quando estamos trabalhando com uma lista de seis livros, por exemplo, parece que não, mas, no dia a dia, nós vamos trabalhar, normalmente, com listas de milhares de dados.
* [01:01] Então, telefones ou dados de uma população, de uma empresa grande, quantos funcionários; e se precisamos ordenar um vestibular grande, por exemplo, como a Fuvest que tem dezenas de milhares de candidatos e candidatas. Como fazemos?
* [01:16] Então, o que vamos fazer a partir de agora é justamente analisar a complexidade dos algoritmos que nós trabalhamos para que vejamos qual é o impacto que esses algoritmos podem ter em um programa quando trabalhamos com uma grande quantidade de dados e porque performance é tão importante.
* [01:34] Então, vamos lá.

## Transcrição

[00:00] Vamos começar analisando os algoritmos que nós criamos para fazer uma busca pelo menor número, nós chamamos essa função de menorValor. Então, vamos ver como nós analisamos, em termos de processamento, esse algoritmo de busca que acabamos criando.

[00:15] Depois, comparamos com outros que fizemos e com o selection for, que é onde nós usamos o menorValor dentro dele. Então, lembrando com o menorValor funciona.

[00:25] Ele é um for, é um laço de repetição que vai percorrer todos os elementos de uma lista, não importa o tamanho dessa lista. Então, se for uma lista de mil elementos, vai fazer esse for mil vezes.

[00:37] Vai mudar o inicial porque ele vai percorrer mudando o inicial do índice 0, para o índice 1, para o índice 2 e para o índice 3 em diante, mas, mesmo assim, esse for ainda vai rodar mil vezes para um array de mil, dez mil para um array de dez mil, seis vezes para um array de seis elementos, como foi o que fizemos.

[00:56] Dentro do bloco for, tem um outro bloco que é uma condicional, é um if, é um “se”. O código dentro do if só é executado se a condição do if for verdadeira, ou seja, podemos dizer que, para cada loop, para cada laço desse for, pode ou não entrar dentro do if de acordo com o elemento que estamos analisando.

[01:23] Se formos passar isso que falamos agora para uma forma que consigamos calcular, podemos dizer que, em uma lista de seis elementos, vamos ter entre seis operações – porque entrar no for já é uma operação – e doze operações, porque não sabemos se, para cada elemento, vai entrar ou não no if.

[01:43] Então, se formos dizer que esse algoritmo, obviamente, tem que funcionar tanto para uma lista de seis quanto para uma lista de mil elementos, para uma lista de seis elementos tem entre seis operações e doze operações, se entrar no if ou não, mas no for sempre vai entrar.

[02:00] E para uma lista de mil, entre mil e duas mil operações. Então, podemos dizer que esse é um algoritmo que faz entre N operações, sendo N o tamanho da nossa lista, o tamanho do nosso array. Ele faz entre N e dois N, entre uma operação ou duas operações, mil operações ou duas mil operações etc.

[02:27] Vamos pedir ajuda para o Excel para fazermos esses cálculos e vermos o resultado de uma forma um pouco mais fácil. No Excel, vou criar uma tabela. A primeira coluna, vou chamar de “elementos” e nós colocamos aqui quantos elementos vão ter nas nossas listas que queremos calcular.

[02:45] A primeira opção vou passar “n”, que é sendo “n” igual à quantidade de itens que tem na minha lista, porque, se não entrar no if, se entrar só no for, é uma operação para cada item. Então, mil operações para uma lista de mil itens.

[03:03] E também vai ter “2n”, porque nós temos que calcular também caso todos os itens do array entrem no if. Não vai ser o caso no nosso algoritmo, mas temos que levar isso em consideração.

[03:19] Então, se eu começo com uma lista de “1” elemento, “n” vai ser =+a2, só isso. Vai ser “1” porque tem um elemento no meu array. “2n” vai ser “n”, que está na minha célula “b2”, então =+b2\*2, que é “2n”.

[03:55] Para uma lista de um elemento, eu tenho aqui um cálculo ou dois cálculos, mas nós não trabalhamos com uma lista de um elemento, trabalhamos com listas grandes normalmente.

[04:07] Vamos dobrando esses valores para vermos esses cálculos ficarem com números um pouco mais recheados. Na nossa coluna “elementos”, eu vou pegar =+a2\*2 e vou puxar esse cálculo mais alguns itens para baixo.

[04:29] Agora, pego “2”, “4”, “8”, “16”, “32”, fomos dobrando o tamanho do nosso array e vamos ter que também dobrar a quantidade de cálculos.

| **elementos** | **n** | **2n** |
| --- | --- | --- |
| 1 | 1 | 2 |
| 2 | 2 | 4 |
| 4 | 4 | 8 |
| 8 | 8 | 16 |
| 16 | 16 | 32 |
| 32 | 32 | 64 |
| 64 | 64 | 128 |
| 128 | 128 | 256 |
| 256 | 256 | 512 |
| 512 | 512 | 1024 |
| 1024 | 1024 | 2048 |
| 2048 | 2048 | 4096 |
| 4096 | 4096 | 8192 |

[04:40] Então, agora, sim, para uma lista de 128 itens, eu posso ter 128 operações ou 256 operações.

[04:49] Vamos fazer uma coisa mais legal, eu vou pegar essa planilha, esses cálculos que fizemos e vou pedir para o Excel gerar um gráfico para nós. “View > Chart”, um gráfico do que está acontecendo.

[05:07] Ele gerou um gráfico para nós de n e 2n. Pelo gráfico, pelas linhas do gráfico, uma um pouco menor, um pouco mais baixa é a linha do “2n”, a azul, é de “n” e a vermelha que está um pouco mais para cima é a de 2n.

[05:25] À medida que o número vai aumentando, o número de operações possíveis também vai aumentando, mas, segundo o gráfico, ele aumenta de forma linear. O gráfico forma uma linha, está formando duas linhas ascendentes.

[05:40] Então, tanto no caso de “n” quanto no caso de “2n”, o crescimento da quantidade de operações com relação à quantidade de elementos que temos em uma lista – então, se eu tenho quatro mil livros em uma lista – sempre cresce de forma linear, tanto no “n” quanto no “2n”.

[06:01] Por enquanto, podemos deixar esse cálculo esperando. Vamos agora fazer uma comparação para termos mais dados para trabalharmos e ver os algoritmos fazendo seu trabalho.

[06:12] Para isso, para termos o que comparar, vamos dar uma olhada novamente no código do “selectionSort.js”, que foi onde usamos a nossa função menorValor dentro dele.

[06:23] Já sabemos que o algoritmo menorValor de busca tem uma complexidade que vai de “n”, que é “n” ou “2n”, pode se tanto um quanto o outro. Só que estamos usando esse algoritmo, essa função menorValor dentro do for que construímos para fazer o selectionSort.

[06:47] Então, temos o for do selectionSort que também temos que percorrer array, e para cada índice do array que percorremos, nós chamamos a função menorValor.

[06:58] Então, o for que nós criamos para o selectionSort também vai percorrer todos os itens da lista, seja qual for o tamanho da lista – uma lista de mil itens, passam mil itens. E dentro dele temos dele temos outro for que é o for que criamos para o menorValor.

[07:17] Nós colocamos um loop dentro do outro. Estava extraído para uma função, mas ainda é um loop dentro de outro loop. A complexidade do menorValor nós já calculamos, então dá para concluirmos, vendo que temos um for dentro de um for, que o selectionSort não vai ter a mesma complexidade do menorValor.

[07:41] Porque ele tem, dentro dele, “n” ou “2n” e, fora, tem outro “n”. Então, tem um for que faz o cálculo n vezes e dentro dele outro for que pode calcular tanto “n” vezes quanto “2n” vezes.

[08:04] O que temos aqui é n ao quadrado, que é muito diferente de duas vezes n, porque executamos o código do for que está do lado de fora, o código do for do nosso algoritmo de busca, e o código do for que está dentro da função menorValor.

[08:25] Agora, começamos a ver que, quando estamos escrevendo códigos, nós colocamos em uma função e parece que está tudo bem porque estamos trabalhando com uma lista pequena, mas agora vamos começar a ver que, realmente, as coisas são um pouco mais complexas.

[08:43] Já estamos vendo que no código não está aparecendo, mas na execução eu tenho um loop dentro de outro loop. Vamos, então, passar esses dados, agora que já sabemos, para dentro da planilha, e ver isso com números.

[00:00] Já estamos desenhando de cabeça qual é o processo de descobrir a complexidade de um algoritmo, mas é sempre mais legal trabalharmos com números para termos uma ideia melhor.

[00:11] Como fizemos anteriormente com o nosso algoritmo buscaMenor, vamos deixar a planilha de Excel calcular para nós a quantidade de operações dada uma quantidade n de elementos em uma lista.

[00:25] Passando para a planilha, já coloquei uma coluna com a quantidade de elementos, comecei com uma lista de “1” elemento e fui dobrando até “4096” elementos.

[00:37] O que temos que calcular baseado no que tínhamos visto sobre a complexidade do selectionSort? Temos que ter uma opção que tem n2, que vou representar com “n^2”.

[00:52] E temos que ter outra opção também, que é “2\*N^2”.

[00:59] Porque temos que levar em consideração se vai entrar no *if* do buscaMenor ou não, então pode ser mil operações dentro do buscaMenor, pode ser dois mil, temos que ter os dois cenários.

| **elementos** | **n^2** | **2\*n^2** |
| --- | --- | --- |
| 1 |  |  |
| 2 |  |  |
| 4 |  |  |
| 8 |  |  |
| 16 |  |  |
| 32 |  |  |
| 64 |  |  |
| 128 |  |  |
| 256 |  |  |
| 512 |  |  |
| 1024 |  |  |
| 2048 |  |  |
| 4096 |  |  |

[01:16] O cenário da lista com “1” elemento é tranquilo porque vamos fazer, simplesmente, quando temos um elemento, =+A2\*A2. Vai dar “1”, mas, se puxarmos para baixo esse cálculo para calcular as outras quantidades de elementos, já temos uma quantidade muito maior de operações a serem feitas.

[01:47] Passo de uma operação para “1”, claro, mas, em uma lista de “128” elementos, agora passamos passa “16384” operações. E agora vamos para o outro cenário que é “2\*n^2”.

[02:00] Fazemos =+b2\*B2 que vai dar “2”, mas se puxarmos a conta para baixo, agora uma lista de “128” itens vai ter “16384” mil operações ou “32768” mil operações.

| **elementos** | **n^2** | **2\*n^2** |
| --- | --- | --- |
| 1 | 1 | 2 |
| 2 | 4 | 8 |
| 4 | 16 | 32 |
| 8 | 64 | 128 |
| 16 | 256 | 512 |
| 32 | 1024 | 2048 |
| 64 | 4096 | 8192 |
| 128 | 16384 | 32768 |
| 256 | 65536 | 131072 |
| 512 | 262144 | 524288 |
| 1024 | 1048576 | 2097152 |
| 2048 | 4194304 | 8388608 |
| 4096 | 16777216 | 33554432 |

[02:26] Comparado com o nosso buscaMenor, em uma lista de “128” elementos, tínhamos ou “128” operações ou “256”. Então, já cresceu bem a quantidade de operações que estão sendo feitas quando colocamos um *for* dentro de outro *for*.

[02:44] Em uma lista de “4096” elementos, que não é uma lista grande – uma escola que tem quatro mil estudantes – o pior cenário, que é o cenário de "2\*\*\*n^2" já dá “33.554.432” operações.

[03:11] Vamos colocar isso no gráfico? Vou selecionar toda a minha planilha e pedir para o Excel, em “Inserir”, gerar um gráfico de linha para calcular o crescimento do nosso algoritmo.

[03:28] Dá para ver que esse algoritmo já não cresce de forma linear, ele vai crescendo. Enquanto tínhamos um cenário que crescia de 4 mil elementos até 8 mil operações, agora em um cenário de 4 mil elementos nós passamos para 33 milhões de operações, então ele já subiu bastante.

[03:59] Já vimos que a probabilidade de trabalharmos com uma lista de 4 mil itens em um cenário de dia a dia não é pequena. Só o ranqueamento de vestibular de uma faculdade não muito grande já daria isso, ou votos para vereadores de uma cidade pequena também já daria isso ou mais.

[04:18] Vamos juntar tudo que fizemos, então. Eu vou pegar os valores do buscaMenor e vou juntar na planilha do selectionSort. Vou apagar o gráfico, vou empurrar os cálculos do selectionSort que nós fizemos um pouco para o lado para encaixarmos os cálculos anteriores que fizemos para o buscaMenor.

[04:44] Agora podemos selecionar tudo isso e gerar um novo gráfico e comparar. O gráfico que foi gerado agora tem os quatro cenários: o melhor e pior cenário do buscaMenor, que é um algoritmo de crescimento linear; e esse que fizemos agora, que em vez de “2n”, estamos trabalhando com “n^2” ou “2n^2”.

[05:11] Nessa representação de gráfico, as linhas referentes ao crescimento linear nem estão aparecendo, elas nem chegaram a triscar a quantidade numérica que foi necessária para o gráfico conseguir ser desenhado.

[05:28] O que eu vou fazer? Vou deletar esse gráfico e deletar algumas linhas para trabalharmos com uma quantidade um pouco menor de elementos.

| **elementos** | **n** | **2n** | **n^2** | **2\*n^2** |
| --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 1 | 2 |
| 2 | 2 | 4 | 4 | 8 |
| 4 | 4 | 8 | 16 | 32 |
| 8 | 8 | 16 | 64 | 128 |
| 16 | 16 | 32 | 256 | 512 |
| 32 | 32 | 64 | 1024 | 2048 |
| 64 | 64 | 128 | 4096 | 8192 |

[05:38] Vou inserir o gráfico de novo, vamos ver se melhor a visualização.

[05:44] Então, novo gráfico de linhas. Com uma lista de 64 elementos e, mesmo assim, quase não dá para ver o resultado do algoritmo de crescimento linear, porque esse aqui, o de “n^2”, é um algoritmo que não está simplesmente dobrando, mas ele vai ao quadrado.

[06:11] Nós chamamos isso de um algoritmo de crescimento quadrático e não linear, como é o caso do menorValor. Se pensarmos que a diferença já é bem grande em uma lista só de 64 elementos, o que vai acontecer quando trabalharmos uma lista de milhares de elementos, de dez mil elementos, que não é um caso muito difícil de acontecer no dia a dia.

[06:35] Uma coisa interessante para analisarmos tanto em um caso quanto no outro é que a diferença entre “n” e “2n” nem é tão grande, ela não afeta tanto, tanto no caso do buscaMenor quanto se colocarmos no selectionSort quando fazemos “n^2” e “2n^2”.

[06:53] A multiplicação, nesses casos, não está afetando muito a quantidade de operações. O que está afetando a linha de crescimento é a potência, que agora estamos crescendo ao quadrado.

[07:06] Como, afinal, o computador vai manipular todo esse crescimento, essa quantidade de processamento, essa quantidade de operações?

[07:14] Agora que temos valores de comparação, nós conseguimos começar a pensar “se eu tenho uma quantidade X de algoritmos, eu tenho mais de algoritmos de busca, eu tenho mais de algoritmos de ordenação para escolher”.

[07:26] Então agora começamos ver por que às vezes temos que entender qual é o melhor algoritmo para ser utilizado dependendo da quantidade de processamento que vai ser feita.

[07:37] Agora já vimos um algoritmo que tem crescimento linear e um algoritmo que tem crescimento quadrático, mas existem outros tipos e vamos agora, em seguida, fazer uma outra simulação para vermos como o computador vai encarar esses números e como isso impacta no processamento.

[00:00] Você já prestou atenção nas informações técnicas de um computador? Quando vamos comprar um computador, o que significam alguns daqueles números de processador, de velocidade etc., e o que isso tem a ver com os cálculos que estamos fazendo aqui, que estamos até utilizando a planilha, já saímos do código.

[00:18] Não vou entrar em detalhe sobre como funcionam os processadores em quantidade de núcleo, aceleração de *clock* etc. Temos alguns cursos, alguns conteúdos na Alura sobre esse assunto, vou deixar como material extra.

[00:32] O que vamos fazer é ver de uma forma um pouco mais simplificada como que a quantidade de operações que estamos calculando na nossa planilha impacta no processamento do computador, em como o computador processa essas informações.

[00:46] Ainda mais que estamos trabalhando com JavaScript, que é uma linguagem que, no caso do *front-end*, todo processamento é feito no computador do lado do cliente, no lado do usuário, então isso passa a ser ainda mais importante.

[01:00] O que é legal sabermos agora, já que não vamos entrar em detalhes? A velocidade de um processor determina quantas operações um computador consegue em determinado tempo.

[01:10] Por exemplo, um processador de “3,2GHz” executa 3.2 bilhões de ciclos por segundo. Não precisa se preocupar muito com o que é ciclo. Para entendermos um pouco melhor, vamos dar uma simplificada, vamos dar uma diminuída nessa quantidade, nesses números – porque trabalhar com 3 bilhões vai ficar um pouco complicado de vermos os resultados na tela.

[01:45] Vamos supor que o nosso computador tem a capacidade de fazer 8 mil operações por segundo. Então, colocar aqui que é um “comp 8000”, 8 mil operações por segundo.

[02:04] Na vida real, os computadores mais modernos têm capacidade de fazer muito mais do que isso, uma capacidade de processamento muito maior do que essa, mas vamos diminuir para os números ficarem mais visíveis, mais fáceis de lidarmos.

[02:18] Vamos dizer que um computador que faz 8 mil operações por segundo, vamos ver como ele maneja a quantidade de operações do nosso algoritmo quando tem que lidar com uma lista de “n” em um crescimento linear, que é, no caso, a nossa função menorValor.

[02:39] O que eu vou fazer? Vou pegar os valores de “n” que estão na coluna “B”, então vou pegar =+B2/8000. Vamos ver quantas operações ele vai fazer por segundo.

[02:54] No caso de uma lista de “1” elemento, vai ser uma quantidade muito rápida, “0.000125” segundos para ele fazer cada operação; se temos uma lista de “1024” elementos, também vai levar 1 décimo de segundo para fazer essa operação.

[03:12] “0.128” segundos para percorrer a lista e executar um algoritmo que tem crescimento “n”. O crescimento “2n” vai ser o dobro disso. No caso, para uma lista de “1024” elementos, poderíamos pegar o nosso resultado direto que está na célula “F12”.

[03:40] Então, conseguimos fazer =+f12\*2, vai levar “0,256” segundos, menos de 1 segundo ainda. Agora, o que nós fazemos? Vamos fazer uma simulação de como o nosso computador de 8 mil operações por segundo lida com algoritmo de crescimento quadrático, um algoritmo de “n^2”.

[04:14] Agora temos que pegar os valores da coluna “D”, estou trabalhando com a célula D2, então =+d2\*d2, que é “1”, e agora vamos dividir isso por 8000, então =+d2\*d2/8000, que é a quantidade de operações do nosso computador – coloquei para fazer por segundo para ficar mais fácil de vermos os valores.

[04:44] Também vou puxar esses cálculos para baixo para vermos, e agora o que nós temos é, em uma lista de “1024” elementos, um tempo para fazer essa operação em um computador que faz 8 mil operações por segundo que, se dividirmos isso por 60, para vermos esse valor em minutos – por enquanto, ele é “137.438.953” segundos.

[05:21] Vou dividir esse número por 60, =+G12/60para ter uma quantidade de minutos de 2 milhões de minutos. Esse número está muito grande, então uma lista de “1024” elementos está muito difícil de trabalhar, vamos fazer o seguinte.

[05:42] Vou diminuir para “32” elementos. Um computador que faz 8 mil operações por segundo vai levar ainda 131 segundos para conseguir lidar com uma lista de “32” elementos.

[06:00] “131” segundos dá 2 minutos e alguma coisa. Se formos comparar com o algoritmo linear, que leva muito menos de 1 segundo para processar uma lista com a mesma quantidade de elementos, “32” elementos, é muita diferença.

[06:22] Já vimos que o fator de multiplicação você vai fazer uma vez, é um “n”, “2n”, não influi tanto. Você pode falar: “você mesma falou que os computadores atuais são muito mais rápidos do que 8 mil operações simples por segundo”.

[06:42] Na verdade, um computador hoje em dia não vai levar 2 minutos para fazer uma operação em uma lista de 32 elementos. Não importa muito aqui quão moderno é o computador, se os computadores são mais rápidos, os atuais são mais modernos etc.

[07:03] O problema que temos continua o mesmo. Nós temos uma diferença de quantidade de operações muito grande entre um tipo de algoritmo, o linear, e o outro, que é o quadrático.

[07:17] Isso que eu queria mostrar para vocês. O que estamos fazendo é analisar a eficiência de um algoritmo baseado na curva de crescimento dele, na curva de crescimento da quantidade de operações.

[07:33] Isso independente dos recursos que temos – se o computador é mais rápido, se o computador é menos rápido –, porque a diferença de performance, a diferença da quantidade de operações ainda vai existir.

[07:48] Nós chamamos isso de análise assintótica. Independentemente dos recursos, analisarmos a eficiência de um algoritmo baseado na curva de crescimento, na curva de crescimento da quantidade de operações que temos que fazer dentro dele.

[08:06] É aí que começamos a abordar o problema de outras formas. Se for para achar só o menor, precisa mesmo ordenar? Podemos ficar só com o menor valor?

[08:18] Começamos a ver o problema de algumas formas e uma coisa que parecia que ia nos ajudar a fazer a ordenação antes de achar o menor começa a aparecer menos atrativa, mas, como gostamos de falar em programação, tudo depende, então depende muito do caso.

[08:34] O que vimos agora foi um exemplo reduzido de como a complexidade de um algoritmo impacta no processamento. Afinal de contas, os computadores não têm memória infinita, mesmo que seja para fazer operações simples, quando trabalhamos com quantidades grandes de dados.

[08:54] Mas tem um outro algoritmo para fazermos ainda a análise, que não fizemos, que é o insertionSort. Então, vamos ver como o insertionSort entra na nossa tabela.

Durante a aula usamos um vídeo que compara a performance de diversos algoritmos de ordenação. Você pode conferir o vídeo [nesse link](https://youtu.be/BeoCbJPuvSE)! Veja como existem vários outros algoritmos apenas para resolver o problema de ordenar listas, como Quick Sort, Radix Sort e etc. Todos esses algoritmos já foram desenvolvidos anteriormente, assim como o Selection Sort e o Insertion Sort que utilizamos durante o curso.

Se quiser checar outros modos de visualizar os dados, pode buscar por sorting algorithm comparison no YouTube e comparar os resultados.

Você pode praticar a implementação em javaScript de alguns desses outros algoritmos!

# 07 Para saber mais: Como funciona um processador

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98374/next)

Nesta aula falamos muito brevemente sobre como funciona um processador, e simplificamos um pouco o tema para trabalhar com nossos exemplos.

É interessante que você, ao começar seus estudos em programação, busque também relacionar o código e a forma como ele é interpretado pelo computador. Esse conhecimento nos ajuda a entender o porquê de alguns comportamentos das linguagens de programação, porque alguns bugs acontecem e como corrigi-los ou evitá-los. E, como vimos durante a aula, essa questão pode ser essencial para uma boa performance dos nossos programas.

Para entender melhor como o computador processa as informações e o que significam alguns dos números e siglas que acompanham os modelos dos componentes de computadores, você pode conferir o nosso curso de [Arquitetura de computadores](https://cursos.alura.com.br/course/arquitetura-computadores-funcionamento-programa).

# 09 Faça como eu fiz: Tabelas de comparação

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98376/next)

Utilizamos o Google Sheets (equivalente ao Excel) para montar as tabelas de comparação de algoritmos.

Caso queira replicar as tabelas e fazer mais testes, pode aproveitar as fórmulas que usamos na aula:

| **elementos** | **n** | **2n** | **n^2** | **2\*n^2** | **1** | **n^3** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | =+A2 | =2\*A2 | =A2\*A2 | =2\*D2 | 1 | =A2A2A2 |
| =+A2\*2 | =+A3 | =2\*A3 | =A3\*A3 | =2\*D3 | 1 | =A3A3A3 |
| =+A3\*2 | =+A4 | =2\*A4 | =A4\*A4 | =2\*D4 | 1 | =A4A4A4 |
| =+A4\*2 | =+A5 | =2\*A5 | =A5\*A5 | =2\*D5 | 1 | =A5A5A5 |
| =+A5\*2 | =+A6 | =2\*A6 | =A6\*A6 | =2\*D6 | 1 | =A6A6A6 |

A tabela acima está preparada para uma lista de até 16 elementos, mas você pode expandir a partir daí e ver os resultados!

# 10 O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98377/next)

# Nessa aula, você aprendeu

* Que entender a complexidade de um algoritmo é essencial para que nossos programas funcionem da melhor forma possível;
* Que analisamos esta complexidade observando as operações feitas pelo código e transformando essas operações em números;
* Que performance e tempo impactam em custos e também na experiência do usuário do programa, pois um programa pode ficar muito lento dependendo da quantidade de dados que precisam ser processados;
* A analisar os dados de performance e comparar os números para verificar a performance de um algoritmo com relação a quantidade de operações efetuadas e tempo;
* Como o volume de operações efetuadas por um algoritmo se traduz em tempo de processamento, simulando a capacidade de um processador de computador.
* [00:00] Voltando à planilha e aos gráficos que estávamos gerando, nós comparamos o crescimento de um algoritmo linear com um algoritmo quadrático anteriormente, mas será que existe algum algoritmo que seja mais rápido do que o linear?
* [00:13] Como vimos desde o começo, a lógica para resolver um problema pode envolver mais do que um algoritmo, então, por exemplo, quando nós fizemos o selectionSort, usamos dentro do selectionSort um outro algoritmo que chamamos de menorValor, a função que criamos que encontrava o menor valor em uma lista.
* [00:32] Dessa forma, já vimos que conseguimos combinar vários algoritmos para resolver um problema. Se voltarmos para a nossa planilha e vermos o gráfico entre o algoritmo linear e o algoritmo quadrático – vamos recolocar o gráfico só para relembrarmos.
* [00:54] Dá para ver aquilo que já sabíamos, tem uma grande diferença entre os dois. O linear cresce de forma linear, vai ter a quantidade de operações igual à quantidade de elementos ou o dobro.
* [01:10] Como que nós conseguimos fazer um algoritmo ficar mais simples do que isso? Vamos pensar, pensando em usar vários algoritmos para resolver um problema. Vamos pensar em um algoritmo que pegue, simplesmente, o índice do meio de uma lista.
* [01:26] Em uma lista de 64 elementos, eu quero pegar o elemento do meio, ou quero pegar o primeiro elemento, ou quero pegar o último elemento. Para fazermos qualquer uma dessas coisas, só precisamos de uma operação, seja pegar o meio, seja pegar o começo, seja pegar o final.
* [01:46] Nós dizemos que esse tipo de algoritmo é um algoritmo de crescimento constante. Para uma lista de 64 itens, é uma operação só; para pegar o elemento do meio em uma lista de mil itens, é só uma operação etc.
* [02:02] Então, não importa o tamanho da nossa lista, da quantidade de elementos, é só uma operação. De que forma isso fica na planilha? Na planilha, é sempre “1”, é sempre uma operação, não importa a quantidade de elementos.
* [02:18] Se colocarmos isso no nosso gráfico, vai ficar – deixa eu colocar um gráfico de linha – a mesma coisa, não importa a quantidade de elementos. O linear vai crescendo de uma forma bem menor que a quadrática, que está meio que tomando o gráfico inteiro, e quase não dá para perceber que tem uma linha do constante, que é sempre “1”.
* [02:49] Esse é um exemplo de um algoritmo que consegue ser mais rápido ainda que o linear, encontrar o meio de uma lista, encontrar o fim de uma lista, o primeiro elemento etc.
* [02:58] Parece um exemplo meio bobo, para que vou usar isso? Mas guarde essa informação porque esse tipo de algoritmo vai ser necessário, vai ser útil, especialmente, como vimos no caso do selectionSort, se você pensar que combinamos algoritmos.
* [03:14] Então, em combinação com outros, esse tipo de algoritmo constante de encontrar algo no meio de uma lista, dividir uma lista em duas etc., pode ser bem útil.
* [03:25] No caso, para pegar os três primeiros itens de um alista, também é algoritmo de crescimento constante – se a lista estiver ordenada. Para ordenar a lista, já vimos que é outra história, mas cai na mesma coisa: combinar algoritmos.
* [03:38] Já gastamos processamento para ordenar a nossa lista da melhor forma possível, se quisermos pegar os três primeiros, vai ser provavelmente um algoritmo constante, porque é sempre uma operação para pegar as três primeiras posições do *array*, não importa o tamanho do *array*.
* [03:53] Da mesma forma que existe o crescimento constante e o linear, existem alguns algoritmos que estão no meio termo, não são uma coisa nem outra. Por exemplo, se temos uma lista telefônica – não sei se você já viu uma lista telefônica de papel, mas uma lista telefônica de papel é um catálogo com telefones de uma cidade que estão ordenados por ordem alfabética.
* [04:17] Se quiséssemos, por exemplo, procurar o “Edson”, sabemos que não precisamos começar do “A”, podemos começar do “Ed” e vai estar em algum ponto do “Ed” letras para frente.
* [04:30] Ele está meio que no meio termo de uma busca quadrática e uma busca linear, então existem várias formas de combinar algoritmos e de ter meio termos entre a complexidade.
* [04:43] Se quiséssemos pensar em um algoritmo que seja mais complexo do que o quadrático, por exemplo. Vamos sair dos casos menores, de menor crescimento, e vamos partir para os maiores.
* [04:58] Se fossemos seguir essa lógica que usamos para fazer tanto o selectionSort quanto o insertionSort e acabássemos criando mais um *loop* dentro de outro?
* [05:10] Tanto no caso de um, do insertionSort, como no selectionSort, já vimos que temos um *for* dentro de um *for* e um *while* dentro de um *for* – os dois são *loops*.
* [05:20] Então, e colocássemos um outro *for* ou outro *while* dentro para funcionar um dentro do outro, teríamos um algoritmo de crescimento cúbico. Vamos ver como fica isso na planilha.
* [05:34] Aqui, ao invés de “n^2”, vamos ter “n^3”, “n3”. Vou, então, criar uma nova coluna =A2\*A2\*A2. Começa com “1”, no caso, mas, se expandirmos essa conta para uma lista de “64” posições, que é o que estamos trabalhando para conseguirmos ver bem o gráfico, que dá uma classe em uma universidade, já temos “262” mil operações.
* [06:23] Se crescermos para “128”, por exemplo, e fomos comparar a mesma coisa, já vai para “2.097.152” operações para uma lista de 128 elementos. Aí não dá, começa a ficar um pouco complexo, mesmo pensando que os computadores hoje em dia têm uma boa capacidade de processamento.
* [06:49] Mas tem cenários que podem ser piores ainda. Nós fizemos “n^2”, mas se fizéssemos, por exemplo, 2 potência de 2, 2 elevado à quantidade de elementos. Por exemplo, colocar aqui, “2^n” na nossa coluna e a fórmula ficaria no Excel, então =pow(2, A2), para começar no começo da planilha.
* [07:30] Começamos com “2” e, se expandirmos para uma lista de “128”, o Excel não mostra o número real. Só em uma lista de “32” elementos, temos aqui “4” trilhões de operações.
* [07:49] E a partir de “64” elementos, o Excel desencanou e está me passando um número em anotação científica. Ele tem “1.84467+19” casas além disso, e no outro “+38” casas além disso.
* [08:06] Se atualizarmos esse gráfico, vamos dar uma olhada no que acontece. Vou inserir um novo gráfico de linha para darmos uma olhada, vamos ver o que aconteceu.
* [08:16] Aconteceu que bugou o meu gráfico, o Excel se recusou a calcular isso, bugou a visualização dos números. Toda essa brincadeira com o gráfico é bem legal.
* [08:36] A intenção é mostrar que, como vimos no caso do algoritmo de crescimento quadrático, é bem fácil explodirmos a quantidade de processamento quando trabalhamos com um número muito grande de elementos.
* [08:51] Então, lembrando sempre, normalmente, no dia a dia, nós não trabalhamos com uma lista de 128 elementos, trabalhamos com quantidades muito maiores de dados.
* [08:59] Por isso, sempre temos que saber: como analisamos algoritmos, que eles precisam ser analisados com relação à complexidade deles; tem que levar o índice de crescimento em conta – se é quadrático, se é linear, se é logarítmica etc.
* [09:15] Que existem vários tipos de crescimento e vários algoritmos prontos - selectionSort e insertionSort são só dois deles para ordenação, porque existem algoritmos para todos os tipos de problema; e cada um tem as suas vantagens e desvantagens.
* [09:32] Vamos fechar por aqui, com esses exemplos, só estamos fazendo uma introdução ao tema do estudo de algoritmos. O estudo de algoritmos é muito importante, ele é parte super importante do estudo de programação, e dá para ter centenas de horas de conteúdo só para isso, só analisando cada um dos que já existem, a curva de crescimento, os cenários etc.
* [09:56] Mas, agora, com esse começo, agora que já temos uma ideia do que são os algoritmos, de como analisamos, de porque é importante analisarmos, você já pode tentar ver quais são os outros, por exemplo, os outros algoritmos famosos de busca.
* [10:12] Eu deixei material extra há uns vídeos trás, umas aulas atrás com alguns *links*, e você pode tentar ler o código pode trás deles, muitas vezes eles podem não estar em JavaScript, mas você pode tentar fazer o que fizemos aqui.
* [10:27] Às vezes, eles podem estar em outra linguagem ou escrito em pseudocódigo. Aí começa a fazer o exercício e começa a analisar e ver como descobrimos qual é o algoritmo melhor para cada situação que podemos ter no dia a dia de trabalho.

# 03 Para saber mais: Big O Notation

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98379/next)

Ao começarmos o estudo de algoritmos, é comum nos depararmos com a expressão **Big O Notation** (ou Notação “O Grande”).

Essa notação é utilizada para expressar a complexidade de um algoritmo, com relação ao tempo e volume de recursos computacionais necessários para executar esse algoritmo, de acordo com o volume de dados de entrada. Em geral, utiliza-se o tempo de uso de CPU (unidade central de processamento) do computador.

Ou seja, usamos O para classificar funções (implementações de algoritmos) de acordo com a forma como crescem em complexidade. Algoritmos diferentes podem crescer em complexidade da mesma forma, então podem ter a mesma classificação O, como vimos durante o curso com o Selection Sort e o Insertion Sort.

Podemos classificar os exemplos que vimos durante a aula da seguinte forma:

| **notação** | **nome** |
| --- | --- |
| O(n) | linear |
| O(n²) | quadrático |

A função menorValor() é um exemplo de algoritmo de crescimento linear: o “custo” (referente à quantidade total de operações necessárias) depende da quantidade de elementos na lista; uma lista de 10 elementos resulta em 10 operações de busca e uma lista de 1000 elementos resulta em 1000 operações. Ou seja, **menorValor() tem complexidade de O(n), onde n é o tamanho (quantidade de elementos) da lista**.

Já nas funções selectionSort() e insertionSort(), vimos que um laço de repetição dentro do outro faz com que a taxa de crescimento não aumenta mais de forma linear com relação à quantidade de elementos na lista, porém de forma quadrática, pois um laço dentro de outro significa que a quantidade de operações será de n\*n (sendo n a quantidade de elementos). Ou seja, **selectionSort() e insertionSort() têm complexidade de O(n²)**.

Como os exemplos que vimos na planilha e nos gráficos da aula, existem outras formas de avaliar o crescimento de um algoritmo, e que também podem ser representada segundo a notação O grande:

| **notação** | **nome** |
| --- | --- |
| O(n) | linear |
| O(n²) | quadrático |
| O(1) | constante |
| O(log(n)) | logarítmica |
| O(nc) | polinomial |
| O(cn) | exponencial |

A notação O grande não é usada apenas em computação, mas também em várias áreas da matemática e existem outras notações na lista além dos exemplos que utilizamos.

Se quiser saber mais e ter outros exemplos de notações relacionadas a algoritmos comuns em programação, o site [Big O cheatsheet](https://www.bigocheatsheet.com/) (em inglês) fornece ótimo material para estudo.

# 05 Faça como eu fiz: Mais cálculos de complexidade

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/algoritmos-javascript-i-algoritmos-ordenacao/task/98382/next)

Podemos aproveitar a planilha e os gráficos que utilizamos na aula para testar a complexidade (pela notação O grande) de outros algoritmos, como a logarítmica e a linear-logarítmica, além das que vimos na aula:

| **elementos** | **n** | **n^2** | **1** | **log** | **n\*log(n)** | **n^3** | **2^n** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 2 |
| 2 | 2 | 4 | 1 | 1 | 2 | 8 | 4 |
| 4 | 4 | 16 | 1 | 2 | 8 | 64 | 16 |
| 8 | 8 | 64 | 1 | 3 | 24 | 512 | 256 |
| 16 | 16 | 256 | 1 | 4 | 64 | 4096 | 65536 |
| 32 | 32 | 1024 | 1 | 5 | 160 | 32768 | 4294967296 |
| 64 | 64 | 4096 | 1 | 6 | 384 | 262144 | 1.84467E+19 |
| 128 | 128 | 16384 | 1 | 7 | 896 | 2097152 | 3.40282E+38 |

Se quiser fazer mais testes, atualize sua planilha com as fórmulas que utilizamos e expanda para maiores quantidades de elementos:

| **elem** | **n** | **2n** | **n^2** | **1** | **log(n)** | **n \* log(n)** | **2^n** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | =A2 | =2\*A2 | =A2\*A2 | 1 | =log(A2,2) | =A2\*log(A2,2) | =POW(2,A2) |

Atualize os gráficos e veja os resultados!

[00:00] Obrigada por ter me acompanhado até o final desse curso de algoritmos com JavaScript.

[00:05] Nós pegamos, a partir de um exemplo que conseguimos relacionar para o dia a dia, por exemplo, ordenação de uma lista de livros, ordenação de produtos, coisas que trabalhamos e usamos bastantes como usuário em *e-commerces*, vendo e-mails etc.

[00:21] Então, partimos de um problema como esse, no caso, uma lista de livros em uma livraria, e abstraímos os problemas, por exemplo, de encontrar o menor preço, ou de ordenar livros por preço.

[00:32] A partir desses problemas que conseguimos fazer um fluxo de como resolver, nós passamos para o código, e descobrimos como utilizar dois algoritmos de ordenação que já são conhecidos, o *Insertion Sort* e o *Selection Sort*.

[00:50] Como podemos escrever esses algoritmos com o JavaScript e como eles resolvem os problemas, quais são os passos. Fizemos testes de mesa para entender o que tem em cada variável, o que acontece em cada linha, cada parte do código, em cada momento.

[01:08] Isso nos ajuda a compreender de que forma esses algoritmos funcionam, de que forma eles resolviam os nossos problemas de ordenação. Também descobrimos que algoritmos têm complexidade, como lidamos com a complexidade desses algoritmos, e por que isso é importante.

[01:26] Quantas operações um algoritmo faz para resolver nosso problema, é muito, é pouco? O que acontece toda vez que fazemos um *loop* em uma lista, um *array*, e como isso se traduz em processamento e como isso pode impactar na performance de um algoritmo?

[01:44] E performance de algoritmo é custo, é experiência de usuário, afinal de contas, ninguém quer esperar muito para a lista de produtos ser ordenada, e por aí vai.

[01:55] Essa foi a nossa introdução ao estudo de algoritmos com JavaScript, vimos dois algoritmos de ordenação clássicos, *Selection Sort* e *Insertion Sort*.

[02:06] Então, a partir de agora, você consegue continuar a fazer os seus estudos de algoritmo, agora que entendemos como eles funcionam, como eles tentam resolver os problemas da melhor forma possível, e que existem vários algoritmos tentando cada vez mais resolver da forma mais performática os problemas de programação que temos no dia a dia.

[02:26] Nos vemos por aí nos cursos da Alura. Não esqueça de, se você ainda não deu uma olhada no material extra, nos exercícios, nos "Para saber mais", tem bastante conteúdo que nós preparamos a mais para enriquecer o estudo de algoritmo e tudo que fazemos.

[02:44] Então, se você não fez ainda os exercícios, não conferiu os "Para saber mais", nem as atividades extras, volta lá que agora é uma boa hora. Até o próximo curso.