

Como solucionar problemas de programação com um método simples em quatro etapas

Um método para solucionar problemas

Esse método foi extraído do livro *How to Solve It*, de George Pólya, publicado originalmente em 1945 e com mais de um milhão de cópias vendidas.

Embora tenha surgido no contexto da matemática, ele é amplamente utilizado no ensino de ciência da computação. É adotado por professores universitários — como David Evans, no curso *Intro to Computer Science* da Udacity — e por educadores modernos de desenvolvimento web, como Colt Steele.

Para tornar o método concreto, vamos aplicá-lo a um problema simples usando JavaScript.

Problema:

Crie uma função que receba dois números, some esses valores e retorne o resultado.

O método de Pólya é composto por quatro etapas:

1. Compreender o problema
2. Elaborar um plano
3. Executar o plano
4. Olhar para trás (analisar e melhorar)

Vamos começar pela primeira.

Etapa 1: Compreender o problema

Quando recebemos um problema de programação — especialmente em uma entrevista — é natural querer sair codando imediatamente. A pressão do tempo torna esse impulso ainda mais forte.

Resista a isso.

Antes de escrever qualquer linha de código, certifique-se de que você entende completamente o problema. Muitos erros surgem não da implementação, mas de uma compreensão incompleta ou equivocada do que foi pedido.

Leia o enunciado com atenção. Em entrevistas, ler em voz alta ajuda a organizar o raciocínio e reduzir a ansiedade.

Se algo não estiver claro, pergunte. Isso demonstra maturidade técnica, não fraqueza. Se estiver estudando sozinho, pesquise e reflita até ter clareza.

Não se resolve corretamente aquilo que não foi bem compreendido.

Quais são as entradas?

Entradas são os dados que a função receberá. Neste problema, sabemos que a função recebe números — mas isso ainda é vago.

Pergunte-se:

- Serão sempre exatamente dois números?
- O que acontece se vierem mais?
- As entradas podem ser inválidas?
- Podem ser strings como "a" e "b"?

Às vezes o próprio enunciado responde isso. Outras vezes, você precisa perguntar ou assumir explicitamente.

Você pode registrar isso como comentário:

```
// entradas: dois números
```

Quais são as saídas?

O que exatamente deve ser retornado?

Neste caso, a saída é um número representando a soma das duas entradas. Confirme se isso é tudo ou se existem condições especiais.

Crie exemplos simples

Exemplos ajudam a solidificar a compreensão e funcionam como testes mentais.

Vamos chamar a função de add:

```
add(2, 3) // -> 5
```

Isso deixa explícito o comportamento esperado.

Crie exemplos mais complexos (casos extremos)

Agora pense além do óbvio:

- E se as entradas forem strings?
- E se não houver entradas?
- E se uma delas for null ou undefined?

Talvez o entrevistador diga para retornar um erro. Talvez diga para assumir que sempre serão números válidos. O importante é **pensar nesses cenários**.

Você pode registrar isso como comentário:

```
// retornar erro se as entradas não forem números
```

// ou assumir que sempre serão números

David Evans chama isso de **código defensivo**: antecipar o que pode dar errado e decidir como o código deve reagir.

Resumo da Etapa 1

- Leia atentamente o problema
- Identifique entradas e saídas
- Crie exemplos simples
- Considere casos extremos

Só avance quando o problema estiver claro.

Etapa 2: Elaborar um plano

Agora que o problema está compreendido, **não escreva código ainda**.

Crie um plano em pseudocódigo — uma descrição em linguagem natural das etapas necessárias para resolver o problema.

Exemplo:

```
// criar uma variável para armazenar a soma  
// somar a primeira entrada com a segunda  
// armazenar o resultado  
// retornar o valor final
```

Esse passo separa **pensamento lógico** de **implementação técnica**.

Em problemas mais complexos, ele evita que você se perca no meio da solução.

David Evans recomenda:

“Pense em como um ser humano resolveria esse problema.”

Ignore linguagens, frameworks ou otimizações. Pense no processo.

Etapa 3: Executar o plano

Agora, finalmente, escreva o código, usando o pseudocódigo como guia:

```
function add(a, b) {  
  const sum = a + b;  
  return sum;  
}
```

A recomendação aqui é clara: **prefira soluções simples e diretas**.

Não tente otimizar antes da hora. Não tente ser “esperto”. Código simples é mais fácil de escrever, entender, testar e corrigir.

E se você travar?

Se uma parte parecer difícil demais, siga o conselho de Colt Steele:

Ignore temporariamente a parte difícil.

Implemente tudo o que você consegue. Muitas vezes, ao avançar, a parte difícil se torna clara.

Parar completamente é pior do que avançar parcialmente.

Etapa 4: Olhar para trás (analisar e melhorar)

Com a solução funcionando, reflita sobre ela.

Pergunte-se:

- Existe outra forma de resolver?
- O código está legível?
- Alguém entenderia isso rapidamente?
- Posso reutilizar essa lógica?
- Dá para melhorar desempenho ou clareza?
- Como outras pessoas resolveram esse problema?

Por exemplo, podemos simplificar:

```
function add(a, b) {  
    return a + b;  
}
```

Essa etapa nunca termina de verdade. Desenvolvedores experientes revisitam códigos antigos e pensam:

“Eu faria isso diferente hoje.”

Isso é sinal de crescimento, não de erro.

Conclusão

Vimos um método simples e poderoso para resolver problemas de programação:

1. Entender o problema
2. Criar um plano
3. Executar o plano
4. Analisar e refatorar

Esse método melhora não apenas entrevistas técnicas, mas a prática diária de programação.

Resolver problemas é uma habilidade **treinável**. Ninguém nasce sabendo. Clareza vem com prática estruturada.

Agora, falando com você — sobre frustração

Frustração não é sinal de incapacidade.
Frustração é sinal de que você se importa.

Todo programador que evoluiu passou por bloqueios, dúvidas e sensação de insuficiência. A diferença não é talento — é **método, repetição e paciência consigo mesmo**.

Quando a mente travar, volte ao básico:

- Entenda o problema
- Escreva em português
- Dê um passo pequeno
- Continue

Você não está atrasado.
Você está aprendendo.

GUIA COMPLETO: Como solucionar problemas de programação com um método simples

PARTE 1: O MÉTODO (Conceito Original)

Introdução

Esse método foi extraído do livro *How to Solve It*, de George Pólya, publicado originalmente em 1945 e com mais de um milhão de cópias vendidas. Embora tenha surgido no contexto da matemática, ele é amplamente utilizado no ensino de ciência da computação. É adotado por professores universitários — como David Evans, no curso *Intro to Computer Science* da Udacity — e por educadores modernos de desenvolvimento web, como Colt Steele.

Para tornar o método concreto, vamos aplicá-lo a um problema simples usando JavaScript.

Problema:

Crie uma função que receba dois números, some esses valores e retorne o resultado.

O método de Pólya é composto por quatro etapas:

1. Compreender o problema
2. Elaborar um plano
3. Executar o plano
4. Olhar para trás (analisar e melhorar)

Vamos começar pela primeira.

Etapa 1: Compreender o problema

Quando recebemos um problema de programação — especialmente em uma entrevista — é natural querer sair codando imediatamente. A pressão do tempo torna esse impulso ainda mais forte. **Resista a isso.**

Antes de escrever qualquer linha de código, certifique-se de que você entende completamente o problema. Muitos erros surgem não da implementação, mas de uma compreensão incompleta ou equivocada do que foi pedido.

Dicas Importantes:

- Leia o enunciado com atenção. Em entrevistas, ler em voz alta ajuda a organizar o raciocínio e reduzir a ansiedade.
- Se algo não estiver claro, pergunte. Isso demonstra maturidade técnica, não fraqueza.
- Se estiver estudando sozinho, pesquise e reflita até ter clareza. Não se resolve corretamente aquilo que não foi bem compreendido.

Perguntas Chave:

1. Quais são as entradas?

Entradas são os dados que a função receberá. Neste problema, sabemos que a função recebe números — mas isso ainda é vago. Pergunte-se:

- Serão sempre exatamente dois números?
- O que acontece se vierem mais?
- As entradas podem ser inválidas?
- Podem ser strings como "a" e "b"?

Às vezes o próprio enunciado responde isso. Outras vezes, você precisa perguntar ou assumir explicitamente. Você pode registrar isso como comentário:

```
// entradas: dois números
```

2. Quais são as saídas?

O que exatamente deve ser retornado? Neste caso, a saída é um número representando a soma das duas entradas. Confirme se isso é tudo ou se existem condições especiais.

3. Crie exemplos simples

Exemplos ajudam a solidificar a compreensão e funcionam como testes mentais. Vamos chamar a função de `add`:

```
add(2, 3) // -> 5
```

Isso deixa explícito o comportamento esperado.

4. Crie exemplos mais complexos (casos extremos)

Agora pense além do óbvio:

- E se as entradas forem strings?
- E se não houver entradas?
- E se uma delas for null ou undefined?

Talvez o entrevistador diga para retornar um erro. Talvez diga para assumir que sempre serão números válidos. O importante é pensar nesses cenários.

Você pode registrar isso como comentário:

```
// retornar erro se as entradas não forem números  
// ou assumir que sempre serão números
```

David Evans chama isso de **código defensivo**: antecipar o que pode dar errado e decidir como o código deve reagir.

Resumo da Etapa 1:

- Leia atentamente o problema.
 - Identifique entradas e saídas.
 - Crie exemplos simples.
 - Considere casos extremos.
 - Só avance quando o problema estiver claro.
-

Etapa 2: Elaborar um plano

Agora que o problema está compreendido, não escreva código ainda. Crie um plano em **pseudocódigo** — uma descrição em linguagem natural das etapas necessárias para resolver o problema.

Exemplo:

Plaintext

```
// criar uma variável para armazenar a soma  
// somar a primeira entrada com a segunda  
// armazenar o resultado  
// retornar o valor final
```

Esse passo separa pensamento lógico de implementação técnica. Em problemas mais complexos, ele evita que você se perca no meio da solução.

David Evans recomenda: "*Pense em como um ser humano resolveria esse problema.*" Ignore linguagens, frameworks ou otimizações. Pense no processo.

Etapa 3: Executar o plano

Agora, finalmente, escreva o código, usando o pseudocódigo como guia:

JavaScript

```
function add(a, b) {  
  const sum = a + b;  
  return sum;
```

}

A recomendação aqui é clara: **prefira soluções simples e diretas**. Não tente otimizar antes da hora. Não tente ser “esperto”. Código simples é mais fácil de escrever, entender, testar e corrigir.

E se você travar?

Se uma parte parecer difícil demais, siga o conselho de Colt Steele: **Ignore temporariamente a parte difícil**. Implemente tudo o que você consegue. Muitas vezes, ao avançar, a parte difícil se torna clara. Parar completamente é pior do que avançar parcialmente.

Etapa 4: Olhar para trás (analisar e melhorar)

Com a solução funcionando, reflita sobre ela.

Pergunte-se:

- Existe outra forma de resolver?
- O código está legível?
- Alguém entenderia isso rapidamente?
- Posso reutilizar essa lógica?
- Dá para melhorar desempenho ou clareza?
- Como outras pessoas resolveram esse problema?

Por exemplo, podemos simplificar:

JavaScript

```
function add(a, b) {  
    return a + b;  
}
```

Essa etapa nunca termina de verdade. Desenvolvedores experientes revisitam códigos antigos e pensam: *"Eu faria isso diferente hoje."* Isso é sinal de crescimento, não de erro.

Conclusão sobre o Método

Vimos um método simples e poderoso para resolver problemas de programação:

1. Entender o problema
2. Criar um plano
3. Executar o plano
4. Analisar e refatorar

Esse método melhora não apenas entrevistas técnicas, mas a prática diária de programação. Resolver problemas é uma habilidade treinável. Ninguém nasce sabendo. Clareza vem com prática estruturada.

Sobre frustração:

Frustração não é sinal de incapacidade. Frustração é sinal de que você se importa. Todo programador que evoluiu passou por bloqueios, dúvidas e sensação de insuficiência. A diferença não é talento — é método, repetição e paciência consigo mesmo.

Quando a mente travar, volte ao básico:

- Entenda o problema
- Escreva em português
- Dê um passo pequeno
- Continue

Você não está atrasado. Você está aprendendo.

PARTE 2: GUIA DE DESBLOQUEIO (Ferramentas Práticas)

Use esta seção como um checklist rápido sempre que sentir que "travou".

1. Checklist Prático para a Etapa 1 (Entendimento)

Copie e cole isso no topo do seu arquivo de código como comentários:

- [] **Objetivo:** Escreva em 1 frase o que o código deve fazer.
- [] **Entradas:** Quais tipos? (Número, Texto, Lista?) Qual o limite? (Vazio, Negativo?)
- [] **Saída:** O que deve ser devolvido? (Um valor, true/false, null?)
- [] **Critério de Sucesso:** Como sei que funcionou? (Ex: `add(2, 3)` tem que dar 5).

Dica de Ouro: Se não conseguir explicar o problema em voz alta para uma criança (ou um pato de borracha), você ainda não entendeu o suficiente para codar.

2. Checklist Prático para a Etapa 2 (Planejamento)

Não sabe por onde começar o pseudocódigo? Use estes modelos mentais:

Modelo A: Transformação Simples (Entrada -> Processo -> Saída)

1. Validar se a entrada é boa.
2. Fazer a transformação (cálculo, formatação).
3. Devolver resultado.

Modelo B: Lista/Loop (Para percorrer dados)

1. Criar uma variável vazia (acumulador/contador).
2. Para cada item da lista:
 - a. Verificar o item.
 - b. Fazer algo com ele.
 - c. Atualizar a variável vazia.

3. Retornar a variável final.

3. Técnicas para Destravar na Etapa 3 (Execução)

- **A Regra dos 5 Minutos:** Comprometa-se a trabalhar apenas 5 minutos em uma pequena parte (ex: apenas criar a função e receber os parâmetros). Geralmente, o medo some depois que você começa.
- **Console.log é seu amigo:** Não tente adivinhar o que está na variável. Imprima.
 - console.log("Entrada A é:", a);
 - console.log("Tipo de A é:", typeof a);
- **Dividir para Conquistar:** Se o problema pede "Ler um texto, inverter as palavras e salvar num banco", faça **apenas** "Ler o texto" primeiro. Teste. Depois faça "Inverter". Teste.

4. O que fazer na Etapa 4 (Refatoração)

Não basta funcionar. Código é lido por humanos.

- **Nomes de Variáveis:** let x é ruim. let totalDeUsuarios é bom.
- **Limpeza:** Apague os console.log usados para teste.
- **Comentários:** Apague comentários óbvios (ex: // soma a com b) e mantenha os explicativos (ex: // usamos coerção aqui porque a API retorna string).

Template de Arquivo para Exercícios

Copie e cole este template no seu editor para começar qualquer exercício com o pé direito:

JavaScript

```
/*
DESAFIO: [Nome do Desafio]
-----
1. ENTENDER:
- Entrada:
- Saída:
- Caso Normal:
- Caso Extremo (vazio/erro):
2. PLANO (Pseudocódigo):
- Passo 1...
- Passo 2...
*/
// 3. EXECUÇÃO:
function minhaSolucao(entrada) {
    // Validação (Defesa)
    if (!entrada) return null;

    // Lógica principal
    // ...

    return resultado;
}

// 4. TESTES MANUAIS (Olhar para trás):
console.log(minhaSolucao("teste 1")); // Esperado: ...
console.log(minhaSolucao(""));
```