

Rock Project - Números Perfeitos

Bem-vindo ao **Rock Project**, uma aplicação web moderna desenvolvida para explorar e verificar **Números Perfeitos**. Este projeto demonstra o uso de algoritmos matemáticos eficientes, processamento em background com Web Workers e uma interface reativa construída com Next.js.

Deploy e Demonstração

A aplicação está hospedada e disponível publicamente através da Vercel. Você pode acessá-la e testá-la agora mesmo:

 Acesse: <https://rock-project-eosin.vercel.app>

O que é um Número Perfeito?

Na matemática, um **número perfeito** é um número inteiro positivo que é igual à soma de seus divisores positivos próprios (excluindo ele mesmo).

Exemplo: **28** Divisores de 28: 1, 2, 4, 7, 14. Soma: $1 + 2 + 4 + 7 + 14 = 28$.

Este projeto utiliza a relação entre números perfeitos e **Primos de Mersenne**. Todo número perfeito pode ser gerado pela fórmula: $2^{p-1} \times (2^p - 1)$ Onde p é um primo de Mersenne.

Funcionalidades

1. Verificar Número (`VerifyNumber`)

Permite que o usuário insira um número (de qualquer tamanho) e verifique instantaneamente se ele é um número perfeito.

- **Validação Exata:** Para números "pequenos" (até ~150.000 dígitos), o sistema realiza uma comparação exata utilizando `BigInt`.
- **Validação Híbrida:** Para números astronômicos, utilizamos uma heurística matemática que compara a quantidade de dígitos, o prefixo e o sufixo do número, garantindo precisão sem estourar a memória.

2. Encontrar Números (`FindNumber`)

Busca todos os números perfeitos dentro de um intervalo definido pelo usuário.

- **Web Workers:** A busca é executada em uma thread separada (Web Worker) para garantir que a interface do usuário nunca trave, mesmo durante cálculos pesados.
- **Suporte a BigInt:** Capaz de buscar e comparar números muito maiores que o limite padrão de inteiros do JavaScript ($2^{53} - 1$).
- **Download de PDF:** Para os números encontrados, é possível gerar e baixar um arquivo PDF com o número completo (para números menores) ou visualizá-los de forma abreviada (para números gigantes).

Guia de Uso

Como Testar a Aplicação

1. Verificando um Número (Aba "Verificar Número")

Use esta funcionalidade se você já possui um número e quer saber se ele é perfeito.

1. Acesse a aba "**Verificar Número**".
2. Digite ou cole o número no campo de texto.
 - *Dica:* Você pode testar com números pequenos (ex: 6 , 28) ou números gigantes.
3. Clique em "**Verificar**".
4. O sistema informará instantaneamente se o número é Perfeito ou Não.

2. Encontrando Números (Aba "Encontrar Números")

Use esta funcionalidade para descobrir números perfeitos dentro de um intervalo.

1. Acesse a aba "**Encontrar Números**".
2. Defina o intervalo de busca:
 - **De (Início):** O número onde a busca deve começar (ex: 1).
 - **Até (Fim):** O limite superior da busca.
3. Clique em "**Buscar Números**".
4. A lista de números encontrados aparecerá abaixo.
 - Para números onde a visualização é viável, você pode **baixar um PDF** com o número completo.
 - Para números muito grandes, o sistema exibe um aviso de que o download não está disponível devido ao tamanho do arquivo.

3. Internacionalização (i18n)

Utilizamos `next-intl` para criar uma experiência globalizada.

- **Middleware Inteligente:** Configurado para roteamento limpo (sem prefixos `/pt` ou `/en` na URL) e detecção baseada em cookies.
- **Server Components:** Traduções carregadas no servidor para máxima performance.

4. Persistência de Estado (State Management)

Implementamos uma camada de persistência local para melhorar a UX.

- **Session Storage:** Dados de inputs e resultados de cálculos são salvos em tempo real.
- **Recuperação Automática:** Ao recarregar a página, o estado anterior é restaurado, evitando frustração do usuário.



Performance e Resultados de Testes

Durante os testes internos de estresse e performance, a aplicação demonstrou alta capacidade de processamento.



Recorde de 51 Números Perfeitos

Em nossos testes intensivos, conseguimos encontrar e validar os primeiros **51 Números Perfeitos** conhecidos.

- O 51º número perfeito é um número colossal.
- **Limitação de Hardware:** Não foi possível avançar para o 52º número devido a limitações físicas de hardware (memória RAM e capacidade de processamento da CPU) da máquina utilizada nos testes, e não por limitações lógicas do algoritmo.

É importante notar que quanto maior o número ou o intervalo de busca, **mais pesado é o processamento**.

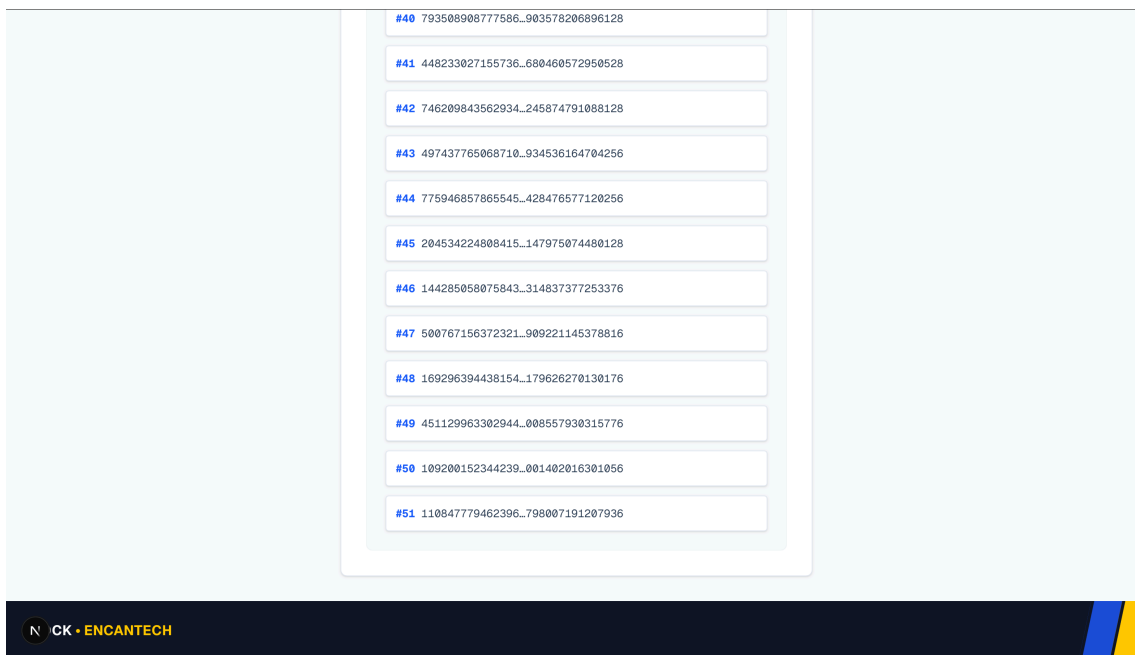
- Buscas por números perfeitos de alta magnitude exigem cálculos matemáticos intensivos.
- Em dispositivos com menos recursos, tentar processar números astronômicos pode causar lentidão ou até mesmo travar momentaneamente a aba do navegador. Recomenda-se cautela ao buscar em intervalos muito grandes.

Evidências dos Testes

Abaixo estão as capturas de tela dos testes internos onde alcançamos a marca de 51 números encontrados:



Teste de busca atingindo números de alta magnitude.



Lista de resultados confirmando a descoberta de múltiplos números perfeitos.

Arquitetura e Estratégias de Resolução

Este projeto adota princípios de **Clean Architecture** e separa responsabilidades de forma clara, tanto no Frontend quanto no Backend. Um dos pontos altos do projeto é a utilização de **duas abordagens distintas** para resolver problemas de alta complexidade computacional.

1. Duas Abordagens para Cálculos Pesados

Para lidar com a verificação e busca de Números Perfeitos (que podem ser astronomicamente grandes), utilizamos estratégias diferentes dependendo do caso de uso:

A. Verificação Unitária: Server-Side Offloading (Next.js API)

No componente `VerifyNumber`, quando o usuário insere um número, o sistema decide onde processá-lo:

1. **Verificação Local:** Se o número for "pequeno" (gerado por `$p \le 107$`), o cálculo é feito instantaneamente no navegador usando `BigInt`.
2. **Verificação Remota:** Se o número for gigantesco, a requisição é enviada para nossa **API interna do Next.js**.
 - o **Por que?** Isso mantém o bundle do cliente leve e centraliza a lógica complexa de verificação híbrida (matemática avançada) no backend.

B. Busca em Intervalo: Client-Side Parallelism (Web Workers)

No componente `FindNumber`, o usuário pode buscar números em um intervalo. Como isso exige testar milhões de possibilidades:

1. **Web Workers:** Utilizamos a API de Workers para rodar o algoritmo de busca em uma **thread separada**.
2. **Resultado:** A interface (UI) permanece 100% fluida e responsiva, mesmo enquanto o processador está fritando nos cálculos em segundo plano.

- **Por que?** Enviar um intervalo inteiro para o backend poderia causar timeout ou sobrecarga no servidor. Distribuir esse trabalho para a máquina do cliente (via Worker) é uma estratégia mais escalável para este tipo de tarefa.

2. Estrutura Arquitetural (Frontend & Backend)

O projeto segue uma adaptação da Clean Architecture para o ecossistema React/Next.js:

🏠 Frontend (Camadas)

1. **Presentation (UI):** Componentes (`VerifyNumber` , `FindNumber`) que apenas exibem dados e capturam eventos.
2. **Application (Hooks):** Custom Hooks (`useVerifyNumber`) agem como "Controllers", gerenciando estado local e chamando serviços.
3. **Domain (Core):**
 - **Use Cases:** `VerifyNumberUseCase` (Frontend) contém a regra de negócio que decide se a verificação deve ser Local ou Remota.
 - **Interfaces:** `IVerifyRepository` define o contrato para as fontes de dados.
4. **Infrastructure (Data):**
 - `LocalVerifyRepository` : Implementação que calcula no browser.
 - `RemoteVerifyRepository` : Implementação que faz `fetch` para `/api/verify` .

🏠 Backend (Next.js Internal)

1. **API Route:** `app/api/verify/route.ts` recebe a requisição HTTP.
2. **Service:** `VerifyService` orquestra a execução.
3. **Use Case:** `VerifyNumberUseCase` (Backend) executa a lógica matemática pesada (validação exata ou híbrida/heurística).

Esta separação permite que testemos cada parte isoladamente (como feito nos testes unitários) e facilita a manutenção futura.

🔧 Tecnologias Utilizadas

- **Core:** [Next.js 15](#) (App Router), [React 19](#)
- **Linguagem:** [TypeScript](#)
- **Estilização:** [Tailwind CSS v4](#)
- **Testes:** [Jest](#), [React Testing Library](#)
- **Performance:** Web Workers API para processamento paralelo

📁 Estrutura do Projeto

```
src/
├── app/                # Camada de Entrada (Next.js App Router)
│   ├── api/           # Rotas de API (Backend Entrypoint)
│   └── page.tsx       # Página Principal
├── components/        # Camada de Apresentação (UI)
│   ├── home/         # VerifyNumber e FindNumber
│   └── forms/         # Componentes base (Button, TextArea)
├── hooks/             # Camada de Aplicação (React Hooks)
└── useVerifyNumber    # Controller do Frontend
```

— domain/	# Camada de Domínio (Core Business Rules)
— usecases/	# Regras de negócio puras (Frontend)
— repositories/	# Interfaces (Contratos)
— infrastructure/	# Camada de Infraestrutura (Implementações)
— repositories/	# LocalVerifyRepository e RemoteVerifyRepository
— backend/	# Backend Logic (Server-Side)
— services/	# Orquestração de serviços
— usecases/	# Regras de negócio pesadas (Matemática)
— workers/	# Processamento Paralelo (Client-Side)
— utils/	# Helpers Matemáticos Compartilhados

Como Rodar o Projeto

Pré-requisitos

- Node.js (v18 ou superior recomendado)
- npm ou yarn

Instalação

1. Clone o repositório:

```
git clone https://github.com/cardosorenanalves/Rock_Project
cd rock-project
```

2. Instale as dependências:

```
npm install
```

3. Inicie o servidor de desenvolvimento:

```
npm run dev
```

Acesse <http://localhost:3000> no seu navegador.

Testes

O projeto possui uma suíte de testes unitários robusta cobrindo utilitários matemáticos, hooks e componentes.

Para executar os testes:

```
npm test
```

Cobertura de Testes

- **Utils:** Garante que cálculos de dígitos, exponenciação modular e formatação estejam corretos.
 - **Hooks:** Testa a lógica de estado, carregamento e tratamento de erros dos formulários.
 - **Componentes:** Verifica a renderização, interação do usuário e feedback visual (loading/erros).
-

Detalhes Matemáticos (Under the Hood)

Algoritmo de Verificação

O verificador não testa divisores um por um (o que seria impossível para números grandes). Em vez disso:

1. Verifica se o número candidato tem a mesma quantidade de dígitos de um número perfeito gerado por um primo de Mersenne conhecido.
2. Se a contagem de dígitos bater, ele gera o número perfeito esperado.
3. Compara o número gerado com a entrada do usuário.

Otimizações

- **Logaritmos:** Usados para calcular o número de dígitos instantaneamente: $\lfloor (2p - 1) \times \log_{10}(2) \rfloor + 1$.
- **Aritmética Modular:** Usada para calcular sufixos de números gigantes sem gerar o número inteiro.

Desenvolvido por Renan Alves Cardoso