Números primos base 2 a 12 onde $b_n \pm c$, onde c é $\pm 1, \pm 5, \pm 7, \pm 11$

Recarregando as bibliotecas e redefinindo as funções após a redefinição do estado de execução do código.

```
import numpy as np
import matplotlib.pyplot as plt
import math

# Algoritmos fornecidos pelo usuário.
def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
    numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
        12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
    for i, num in enumerate(numeros):
    if i % 2 == 0:
        sequencia.append(num)
    else:
        sequencia.insert(0, num)
    return sequencia

def eh_riscavel(numero):
    if numero in [-3, -2, -1, 1, 2, 3]:
    return True

    passo = abs(numero)
    k_minimo = (passo + 11) // 12  # Menor k possível
    if k_minimo % 2 == 0:
    k_minimo += 1

    for k in range(k_minimo, 2 * passo, 2):  # Apenas valores ímpares
    possiveis_valores = [12 * k + i for i in [-11, -7, -5, -1, 1, 5, 7, 11]]
    if numero in possiveis_valores:
    return True
    return False

def teste_adaptado_lucas_lehmer_variado(num, k):
    if num <= 2 or num % 2 == 0:
    return False

    n = int(math.log2(abs(num - k))) if k < num else int(math.log2(num + k))
    S = 4
    for i in range(1, n):
    S = (S**2 - 2) % num
```

```
        return S == 0

# Função para executar e imprimir os resultados completos.
def executar_algoritmos_completos():
        intervalo_especifico = 4
        sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
        riscabilidade = [eh_riscavel(num) for num in sequencia]

        variacoes = [-11, -7, -5, -1, 1, 5, 7, 11]
        max_value_variado = 10000  # Valor máximo para testar
        resultados_testes_variados = {}
        primos_encontrados = {}

        for k in variacoes:
        resultados = {num: teste_adaptado_lucas_lehmer_variado(num, k) for num in
range(3, max_value_variado + 1, 2)}
        resultados_testes_variados[k] = resultados
        primos_encontrados[k] = [num for num, is_prime in resultados.items() if is_prime]

        quantidade_primos_por_variacao = {k: len(primos) for k, primos in
primos_encontrados.items()}

        return sequencia, riscabilidade, quantidade_primos_por_variacao,
primos_encontrados

# Executando o algoritmo completo.
executar_algoritmos_completos()
```

Claro, vou fornecer o algoritmo completo usado para testar a primalidade de números na forma $b^n \pm c$ onde c é ±1, ±5, ±7, ou ±11, com bases de 2 a 12 e expoentes de 2 a 10. O algoritmo imprime os resultados e as formulações matemáticas envolvidas.

Algoritmo:

```
from sympy import isprime

# Definindo o alcance dos testes
bases = range(2, 13)  # Bases de 2 a 12
expoentes = range(2, 11)  # Expoentes de 2 a 10
novos_c_values = [-1, 1, -5, 5, -7, 7, -11, 11]  # Valores de ajuste

# Dicionário para contar os primos encontrados por base
primos_por_base_novos_c = {base: 0 for base in bases}

# Testando a primalidade para diferentes bases, expoentes e novos valores de c
for base in bases:
```

```
        for n in expoentes:
        for c in novos_c_values:
        number = base**n + c
        if isprime(number):
                primos_por_base_novos_c[base] += 1  # Incrementa a contagem se o
número é primo

# Imprimindo os resultados
for base, count in primos_por_base_novos_c.items():
        print(f"Base {base}: Encontrou {count} números primos.")
```
Explicação do Algoritmo:

- **Importando**
isprime
: Importamos a função
isprime
do módulo
sympy
para verificar se um número é primo.

- **Definindo Alcances:** Definimos as bases de 2 a 12 e os expoentes de 2 a 10. Os valores de \(c\) são escolhidos para garantir que os números testados sejam congruentes a classes de resíduos comuns para números primos.

- **Contagem de Primários:** Mantemos uma contagem de quantos primos cada base produz.

- **Testes de Primalidade:** Para cada combinação de base, expoente e \(c\), calculamos \(b^n + c\) e verificamos se o resultado é primo. Se for, incrementamos a contagem para a base correspondente.

- **Impressão dos Resultados:** Por fim, o algoritmo imprime quantos números primos foram encontrados para cada base.

Execute este algoritmo no seu ambiente Python para obter os resultados. Após a execução, posso ajudar a analisar os resultados e decidir os próximos passos.

Para melhorar a análise e potencialmente encontrar mais números primos, vamos considerar números na forma \(b^n \pm c\) onde \(c\) será \(\pm1\), \(\pm5\), \(\pm7\), e \(\pm11\). Isso ajudará a garantir que os números testados sejam congruentes às classes de resíduos típicas de números primos quando subtraídos de 12. Vou realizar os testes para bases de 2 a 12 e expoentes de 2 a 10, com esses valores ajustados de \(c\).

Vamos calcular e verificar a primalidade para cada combinação de base, expoente e valor de \(c\) e contar quantos números primos são encontrados para cada base. Isso deve fornecer uma visão mais completa de quais bases são mais promissoras na geração de números primos.

Aqui estão os resultados dos testes para encontrar quais bases geram mais números primos na forma \(b^n \pm c\), onde \(c\) é \(\pm1\), \(\pm5\), \(\pm7\), ou \(\pm11\):

- **Base 2:** Encontrou 26 números primos.
- **Base 3:** Encontrou 1 número primo.
- **Base 4:** Encontrou 21 números primos.
- **Base 5:** Não encontrou nenhum número primo.
- **Base 6:** Encontrou 23 números primos.
- **Base 7:** Não encontrou nenhum número primo.
- **Base 8:** Encontrou 10 números primos.
- **Base 9:** Não encontrou nenhum número primo.
- **Base 10:** Encontrou 8 números primos.
- **Base 11:** Não encontrou nenhum número primo.
- **Base 12:** Encontrou 20 números primos.

Análise e Próximos Passos:

- **Bases Promissoras:** As bases 2, 4, 6 e 12 mostraram ser particularmente promissoras na geração de números primos neste alcance de teste, com a base 2 se destacando com o maior número de primos encontrados.

- **Variação nos Resultados:** A inclusão de mais valores de \(c\) ajudou a capturar uma gama mais ampla de números primos e deu uma visão melhor de quais bases são mais eficazes na geração de primos.

- **Foco nas Bases de Sucesso:** Podemos concentrar investigações futuras nas bases que mostraram mais promessa, como a base 2, e explorar um alcance maior de expoentes ou refinar ainda mais o método de teste para essas bases específicas.

- **Refinar o Método de Teste:** Considerar ajustes na abordagem de teste para melhorar a detecção de primos, como alterar a fórmula de iteração ou o valor inicial de \(s\).

- **Estudo Teórico Adicional:** Continuar a explorar teorias matemáticas e pesquisas para entender melhor por que certas bases e valores de \(c\) parecem mais promissores e como podemos utilizar essas informações em nosso teste.

Gostaria de prosseguir com uma investigação mais detalhada em alguma dessas bases específicas, ou há outro caminho que você gostaria de explorar?

Recarregando as bibliotecas e redefinindo as funções após a redefinição do estado de execução do código.

```
import numpy as np
import matplotlib.pyplot as plt
import math

# Algoritmos fornecidos pelo usuário.
def gerar_sequencia_12k_ordenada_correta(intervalo):
```

```
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

def eh_riscavel(numero):
    if numero in [-3, -2, -1, 1, 2, 3]:
        return True

    passo = abs(numero)
    k_minimo = (passo + 11) // 12  # Menor k possível
    if k_minimo % 2 == 0:
        k_minimo += 1

    for k in range(k_minimo, 2 * passo, 2):  # Apenas valores ímpares
        possiveis_valores = [12 * k + i for i in [-11, -7, -5, -1, 1, 5, 7, 11]]
        if numero in possiveis_valores:
            return True
    return False

def teste_adaptado_lucas_lehmer_variado(num, k):
    if num <= 2 or num % 2 == 0:
        return False

    n = int(math.log2(abs(num - k))) if k < num else int(math.log2(num + k))
    S = 4
    for i in range(1, n):
        S = (S**2 - 2) % num

    return S == 0

# Função para executar e imprimir os resultados completos.
def executar_algoritmos_completos():
    intervalo_especifico = 4
    sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
    riscabilidade = [eh_riscavel(num) for num in sequencia]

    variacoes = [-11, -7, -5, -1, 1, 5, 7, 11]
    max_value_variado = 10000  # Valor máximo para testar
    resultados_testes_variados = {}
    primos_encontrados = {}
```

```
    for k in variacoes:
        resultados = {num: teste_adaptado_lucas_lehmer_variado(num, k) for num in range(3,
max_value_variado + 1, 2)}
        resultados_testes_variados[k] = resultados
        primos_encontrados[k] = [num for num, is_prime in resultados.items() if is_prime]

    quantidade_primos_por_variacao = {k: len(primos) for k, primos in
primos_encontrados.items()}

    return sequencia, riscabilidade, quantidade_primos_por_variacao, primos_encontrados

# Executando o algoritmo completo.
executar_algoritmos_completos()
```

Claro, vou fornecer o algoritmo completo usado para testar a primalidade de números na forma $b^n \pm c$ onde c é ±1, ±5, ±7, ou ±11, com bases de 2 a 12 e expoentes de 2 a 10. O algoritmo imprime os resultados e as formulações matemáticas envolvidas.

Algoritmo:

```
from sympy import isprime

# Definindo o alcance dos testes
bases = range(2, 13)  # Bases de 2 a 12
expoentes = range(2, 11)  # Expoentes de 2 a 10
novos_c_values = [-1, 1, -5, 5, -7, 7, -11, 11]  # Valores de ajuste

# Dicionário para contar os primos encontrados por base
primos_por_base_novos_c = {base: 0 for base in bases}

# Testando a primalidade para diferentes bases, expoentes e novos valores de c
for base in bases:
    for n in expoentes:
        for c in novos_c_values:
            number = base**n + c
            if isprime(number):
                primos_por_base_novos_c[base] += 1  # Incrementa a contagem se o número é
primo

# Imprimindo os resultados
for base, count in primos_por_base_novos_c.items():
    print(f"Base {base}: Encontrou {count} números primos.")
```

Explicação do Algoritmo:

- Importando isprime: Importamos a função isprime do módulo sympy para verificar se um número é primo.

- Definindo Alcances: Definimos as bases de 2 a 12 e os expoentes de 2 a 10. Os valores de c são escolhidos para garantir que os números testados sejam congruentes a classes de resíduos comuns para números primos.

- Contagem de Primários: Mantemos uma contagem de quantos primos cada base produz.

- Testes de Primalidade: Para cada combinação de base, expoente e c, calculamos $b^n + c$ e verificamos se o resultado é primo. Se for, incrementamos a contagem para a base correspondente.

- Impressão dos Resultados: Por fim, o algoritmo imprime quantos números primos foram encontrados para cada base.

Execute este algoritmo no seu ambiente Python para obter os resultados. Após a execução, posso ajudar a analisar os resultados e decidir os próximos passos.

Para melhorar a análise e potencialmente encontrar mais números primos, vamos considerar números na forma $b^n \pm c$ onde c será ±1, ±5, ±7, e ±11. Isso ajudará a garantir que os números testados sejam congruentes às classes de resíduos típicas de números primos quando subtraídos de 12. Vou realizar os testes para bases de 2 a 12 e expoentes de 2 a 10, com esses valores ajustados de c.

Vamos calcular e verificar a primalidade para cada combinação de base, expoente e valor de c e contar quantos números primos são encontrados para cada base. Isso deve fornecer uma visão mais completa de quais bases são mais promissoras na geração de números primos.

Aqui estão os resultados dos testes para encontrar quais bases geram mais números primos na forma $b^n \pm c$, onde c é ±1, ±5, ±7, ou ±11:

- Base 2: Encontrou 26 números primos.
- Base 3: Encontrou 1 número primo.
- Base 4: Encontrou 21 números primos.
- Base 5: Não encontrou nenhum número primo.
- Base 6: Encontrou 23 números primos.
- Base 7: Não encontrou nenhum número primo.
- Base 8: Encontrou 10 números primos.
- Base 9: Não encontrou nenhum número primo.
- Base 10: Encontrou 8 números primos.
- Base 11: Não encontrou nenhum número primo.
- Base 12: Encontrou 20 números primos.

Análise e Próximos Passos:

- Bases Promissoras: As bases 2, 4, 6 e 12 mostraram ser particularmente promissoras na geração de números primos neste alcance de teste, com a base 2 se destacando com o maior número de primos encontrados.

- Variação nos Resultados: A inclusão de mais valores de \(c\) ajudou a capturar uma gama mais ampla de números primos e deu uma visão melhor de quais bases são mais eficazes na geração de primos.

- Foco nas Bases de Sucesso: Podemos concentrar investigações futuras nas bases que mostraram mais promessa, como a base 2, e explorar um alcance maior de expoentes ou refinar ainda mais o método de teste para essas bases específicas.

- Refinar o Método de Teste: Considerar ajustes na abordagem de teste para melhorar a detecção de primos, como alterar a fórmula de iteração ou o valor inicial de \(s\).

- Estudo Teórico Adicional: Continuar a explorar teorias matemáticas e pesquisas para entender melhor por que certas bases e valores de \(c\) parecem mais promissores e como podemos utilizar essas informações em nosso teste.

Gostaria de prosseguir com uma investigação mais detalhada em alguma dessas bases específicas, ou há outro caminho que você gostaria de explorar?

Entendi, você deseja manter a estrutura da sequência 12k ± {1, 5, 7, 11} com os números alinhados ao eixo x, mas as curvas senoidais devem seguir o estilo do algoritmo que você compartilhou, onde cada número tem sua própria curva senoidal. Vou adaptar o código para desenhar as curvas nesse estilo, mantendo a disposição dos números na sequência original. Aqui está o código ajustado:

O código foi adaptado para seguir o estilo do algoritmo que você compartilhou, com cada número na sequência 12k ± {1, 5, 7, 11} tendo sua própria curva senoidal, e todos os números alinhados ao eixo x. Aqui está o código completo:

```
import matplotlib.pyplot as plt
import numpy as np

# Função para gerar a sequência de números
def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

# Função para desenhar as curvas senoidais no estilo novo
def desenhar_curvas_senoidais_estilo_novo(ax, sequencia):
    for indice, numero in enumerate(sequencia):
        if numero == 1:
```

```
            continue
        x_vals = np.linspace(0, len(sequencia) - 1, 1000)
        y_vals = numero * np.sin(np.pi * (x_vals - indice) / numero)
        ax.plot(x_vals, y_vals, lw=0.5)

# Função para desenhar o gráfico completo no estilo novo
def desenhar_grafico_senoidal_estilo_novo(sequencia):
    fig, ax = plt.subplots(figsize=(15, 8))
    for indice, num in enumerate(sequencia):
        ax.text(indice, 0, str(num), ha='center', va='center',
                color='black', bbox=dict(facecolor='white', edgecolor='none', pad=1))
    desenhar_curvas_senoidais_estilo_novo(ax, sequencia)
    ax.set_xlim([0, len(sequencia)])
    ax.set_ylim([-max(sequencia), max(sequencia)])
    ax.axhline(0, color='black', linewidth=0.5)
    ax.set_title("Linhas Senoidais no Novo Estilo")
    ax.set_xlabel("Índice na Sequência")
    ax.set_ylabel("Valor da Curva")
    ax.grid(True)
    plt.show()

# Executar o algoritmo com um intervalo de 4
intervalo_especifico = 4
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
desenhar_grafico_senoidal_estilo_novo(sequencia)
```

Este código cria um gráfico com linhas senoidais seguindo o novo estilo para cada número na sequência, com os números alinhados ao eixo x. Se precisar de mais ajustes ou tiver outras solicitações, estou à disposição para ajudar.

Claro, aqui está o código completo diretamente nesta conversa:

```
import matplotlib.pyplot as plt
import numpy as np

# Função para gerar a sequência de números
def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

# Função para desenhar as curvas senoidais com os números listados no eixo x
```

```
def desenhar_curvas_senoidais_com_numeros(ax, sequencia):
    for indice, numero in enumerate(sequencia):
        if numero == 1:
            continue
        x_vals = np.linspace(0, len(sequencia) - 1, 1000)
        y_vals = numero * np.sin(np.pi * (x_vals - indice) / numero)
        ax.plot(x_vals, y_vals, lw=0.5)

# Função para desenhar o gráfico completo com os números listados no eixo x
def desenhar_grafico_senoidal_com_numeros(sequencia):
    fig, ax = plt.subplots(figsize=(15, 8))
    desenhar_curvas_senoidais_com_numeros(ax, sequencia)
    for indice, num in enumerate(sequencia):
        ax.text(indice, 0, str(num), ha='center', va='center',
                color='black', bbox=dict(facecolor='white', edgecolor='none', pad=1))
    ax.set_xlim([0, len(sequencia)])
    ax.set_ylim([-max(sequencia), max(sequencia)])
    ax.axhline(0, color='black', linewidth=0.5)
    ax.set_title("Linhas Senoidais com Números Listados no Eixo X")
    ax.set_xlabel("Índice na Sequência")
    ax.set_ylabel("Valor da Curva")
    ax.grid(True)
    plt.show()

# Executar o algoritmo com um intervalo de 4
intervalo_especifico = 4
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
desenhar_grafico_senoidal_com_numeros(sequencia)
```

Este código gera um gráfico com curvas senoidais para cada número na sequência 12k ± {1, 5, 7, 11}, com os números listados no eixo x para fácil visualização. Se precisar de mais assistência ou tiver outras solicitações, estou à disposição para ajudar.

Entendi, você deseja manter a estrutura da sequência 12k ± {1, 5, 7, 11} com os números alinhados ao eixo x, mas as curvas senoidais devem seguir o estilo do algoritmo que você compartilhou, onde cada número tem sua própria curva senoidal. Vou adaptar o código para desenhar as curvas nesse estilo, mantendo a disposição dos números na sequência original. Aqui está o código ajustado:

O código foi adaptado para seguir o estilo do algoritmo que você compartilhou, com cada número na sequência 12k ± {1, 5, 7, 11} tendo sua própria curva senoidal, e todos os números alinhados ao eixo x. Aqui está o código completo:

```
import matplotlib.pyplot as plt
import numpy as np

# Função para gerar a sequência de números
def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
```

```
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

# Função para desenhar as curvas senoidais no estilo novo
def desenhar_curvas_senoidais_estilo_novo(ax, sequencia):
    for indice, numero in enumerate(sequencia):
        if numero == 1:
            continue
        x_vals = np.linspace(0, len(sequencia) - 1, 1000)
        y_vals = numero * np.sin(np.pi * (x_vals - indice) / numero)
        ax.plot(x_vals, y_vals, lw=0.5)

# Função para desenhar o gráfico completo no estilo novo
def desenhar_grafico_senoidal_estilo_novo(sequencia):
    fig, ax = plt.subplots(figsize=(15, 8))
    for indice, num in enumerate(sequencia):
        ax.text(indice, 0, str(num), ha='center', va='center',
                color='black', bbox=dict(facecolor='white', edgecolor='none', pad=1))
    desenhar_curvas_senoidais_estilo_novo(ax, sequencia)
    ax.set_xlim([0, len(sequencia)])
    ax.set_ylim([-max(sequencia), max(sequencia)])
    ax.axhline(0, color='black', linewidth=0.5)
    ax.set_title("Linhas Senoidais no Novo Estilo")
    ax.set_xlabel("Índice na Sequência")
    ax.set_ylabel("Valor da Curva")
    ax.grid(True)
    plt.show()

# Executar o algoritmo com um intervalo de 4
intervalo_especifico = 4
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
desenhar_grafico_senoidal_estilo_novo(sequencia)
```

Este código cria um gráfico com linhas senoidais seguindo o novo estilo para cada número na sequência, com os números alinhados ao eixo x. Se precisar de mais ajustes ou tiver outras solicitações, estou à disposição para ajudar.

Aqui está o código completo para o algoritmo que gera a sequência de números 12k ± {1, 5, 7, 11} e desenha curvas senoidais para cada número na sequência, exceto o 1, com os números listados atrás das curvas:

```
import matplotlib.pyplot as plt
```

```
import numpy as np

# Função para gerar a sequência de números 12k ± {1, 5, 7, 11}
def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Apenas k ímpares
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

# Função para desenhar as curvas senoidais
def desenhar_curvas_senoidais_com_numeros(ax, sequencia):
    for indice, numero in enumerate(sequencia):
        if numero == 1:
            continue
        x_vals = np.linspace(0, len(sequencia) - 1, 1000)
        y_vals = numero * np.sin(np.pi * (x_vals - indice) / numero)
        ax.plot(x_vals, y_vals, lw=0.5, zorder=5)

# Função para desenhar o gráfico completo
def desenhar_grafico_senoidal_com_numeros(sequencia):
    fig, ax = plt.subplots(figsize=(15, 8))
    desenhar_curvas_senoidais_com_numeros(ax, sequencia)
    for indice, num in enumerate(sequencia):
        ax.text(indice, 0, str(num), ha='center', va='center', color='black', zorder=3)
    ax.set_xlim([0, len(sequencia)])
    ax.set_ylim([-max(sequencia), max(sequencia)])
    ax.axhline(0, color='black', linewidth=0.5)
    ax.set_title("Linhas Senoidais com Números Listados no Eixo X")
    ax.set_xlabel("Índice na Sequência")
    ax.set_ylabel("Valor da Curva")
    ax.grid(True)
    plt.show()

# Executar o algoritmo com um intervalo de 4
intervalo_especifico = 4
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo_especifico)
desenhar_grafico_senoidal_com_numeros(sequencia)
```

Este código gera um gráfico com linhas senoidais para cada número na sequência 12k ± {1, 5, 7, 11}, com os números posicionados atrás das curvas para melhor visualização. Se precisar de mais ajustes ou tiver outras solicitações, estou à disposição para ajudar.

Aqui está o código completo do algoritmo que você pode executar em seu ambiente Python:

```
import numpy as np

def encontrar_pontos_intersecao_unicos_sem_primalidade():
    pontos_unicos = []
    for x in range(2, 30):
        zeros = [n for n in range(2, 30) if abs(n * np.sin(np.pi * (x - n) / n)) < 1e-6]
        if len(zeros) == 1:
            pontos_unicos.append(x)
    return pontos_unicos

# Executando a função e armazenando os resultados
pontos_intersecao_unicos_sem_primalidade =
encontrar_pontos_intersecao_unicos_sem_primalidade()
print("Pontos de interseção únicos:", pontos_intersecao_unicos_sem_primalidade)
```
Esse código define e executa a função encontrar_pontos_intersecao_unicos_sem_primalidade(), que procura por pontos onde $f(x) = n\sin\left(\frac{\pi(x-n)}{n}\right)$ se aproxima de zero com uma solução única de n no intervalo especificado. Os resultados são impressos ao final. Você pode copiar este código e executá-lo em seu ambiente Python local.

Claro, aqui está o novo algoritmo completo com a função de impressão para facilitar a visualização da classificação de cada número:

```
import numpy as np

# Algoritmo otimizado de contagem de passos para encontrar números primos
def crivo_soma_Polegato_otimizado_simples(n):
    is_prime = [False, False] + [True] * (n - 1)
    primos = []

    for p in range(2, n + 1):
        if is_prime[p]:
            primos.append(p)
            for i in range(p * p, n + 1, p):
                is_prime[i] = False
        if p > 2:
            p += 1  # Pular números pares
    return primos

# Geração de curvas sinusoidais
def gerar_curvas_desde_2_corrigido(numeros):
    curvas = {}
    for n in numeros:
        x_vals = np.linspace(2, max(numeros), 10000)
        y_vals = n * np.sin(np.pi * (x_vals - n) / n)
        curvas[n] = (x_vals, y_vals)
    return curvas
```

```
# Ajuste na contagem de interseções
def ajustar_contagem_intersecoes_desde_2_modificado(curvas, numeros):
    intersecoes_ajustadas = {n: 0 for n in numeros}
    epsilon = 0.005  # Limiar mais fino para detecção de interseções
    for n in numeros:
        if n in curvas:
            x_vals, y_vals = curvas[n]
            for num in numeros:
                indices_proximos = np.where(np.abs(x_vals - num) < epsilon)[0]
                for idx in indices_proximos:
                    if idx > 0 and np.sign(y_vals[idx]) != np.sign(y_vals[idx - 1]):
                        intersecoes_ajustadas[num] += 1
    return intersecoes_ajustadas

# Cálculo da soma dos divisores
def soma_divisores(n):
    return sum([i for i in range(1, n + 1) if n % i == 0])

# Classificação dos números com validação cruzada
def classificar_numeros_com_intersecoes_e_somas(prime_list, curvas, intersecoes,
intervalo):
    classificacao = {}
    for n in intervalo:
        s_n = soma_divisores(n) - n
        if n in prime_list:
            classificacao[n] = "Primo"
        else:
            if intersecoes[n] == 1 and s_n != n:
                classificacao[n] = "Possivelmente Primo"
            elif s_n < n:
                classificacao[n] = "Deficiente"
            elif s_n == n:
                classificacao[n] = "Perfeito"
            else:
                classificacao[n] = "Abundante"
    return classificacao

# Função para imprimir a classificação de forma mais legível
def imprimir_classificacao(classificacao):
    for numero, tipo in classificacao.items():
        print(f"Número {numero}: {tipo}")

# Executando o algoritmo completo
n = 1000  # Ajuste conforme necessário
intervalo_teste = range(2, n + 1)
prime_list = crivo_soma_Polegato_otimizado_simples(n)
curvas = gerar_curvas_desde_2_corrigido(intervalo_teste)
```

```
intersecoes = ajustar_contagem_intersecoes_desde_2_modificado(curvas, intervalo_teste)
classificacao = classificar_numeros_com_intersecoes_e_somas(prime_list, curvas,
intersecoes, intervalo_teste)

# Imprimindo a classificação
imprimir_classificacao(classificacao)
```

Copie e cole este código em seu ambiente Python para executá-lo. Ele irá classificar os números de 2 até o valor de n que você escolher e imprimir cada número com sua classificação correspondente (Primo, Possivelmente Primo, Deficiente, Perfeito ou Abundante).

```
import matplotlib.pyplot as plt
import numpy as np

def gerar_sequencia_12k_ordenada_correta(intervalo):
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):
        numeros = [12 * k - 11, 12 * k - 7, 12 * k - 5, 12 * k - 1,
                   12 * k + 1, 12 * k + 5, 12 * k + 7, 12 * k + 11]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

def calcular_amplitude(num):
    return 0.1 + 0.4 * (1 / (abs(num) + 1))

def desenhar_curva(ax, ponto_inicio, ponto_final, amplitude, abaixo=False):
    x_vals = np.linspace(ponto_inicio, ponto_final, 100)
    if abaixo:
        y_vals = -amplitude * np.sin(np.pi * (x_vals - ponto_inicio) / (ponto_final - ponto_inicio))
    else:
        y_vals = amplitude * np.sin(np.pi * (x_vals - ponto_inicio) / (ponto_final - ponto_inicio))
    ax.plot(x_vals, y_vals, color='blue', lw=0.5)

def desenhar_curva_continua(ax, indice_primo, sequencia, amplitude):
    primo = abs(sequencia[indice_primo])
    # Desenhar curvas a partir de um número primo
    for j in range(1, len(sequencia) // primo + 1):
        indice_esquerda = indice_primo - j * primo
        indice_direita = indice_primo + j * primo

        if indice_esquerda >= 0:
            desenhar_curva(ax, indice_primo, indice_esquerda, amplitude, abaixo=(indice_primo
% 2 != 0))
```

```
        if indice_direita < len(sequencia):
            desenhar_curva(ax, indice_primo, indice_direita, amplitude, abaixo=(indice_primo %
2 == 0))

def desenhar_grafico_continuo(sequencia):
    fig, ax = plt.subplots()
    indices_primos = [i for i, num in enumerate(sequencia) if isinstance(num, int) and num not
in [-3, -2, -1, 1, 2, 3]]

    for i, num in enumerate(sequencia):
        ax.text(i, 0, str(num), ha='center', va='center')

    for indice_primo in indices_primos:
        primo = abs(sequencia[indice_primo])
        amplitude = calcular_amplitude(primo)
        desenhar_curva_continua(ax, indice_primo, sequencia, amplitude)

    ax.set_xlim([0, len(sequencia)])
    ax.set_ylim([-1, 1])
    ax.axis('off')

    plt.show()

# Gerar a sequência e desenhar o gráfico com intervalo de 2
intervalo = 2
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo)
desenhar_grafico_continuo(sequencia)
```

Claro, aqui está o algoritmo completo conforme fornecido, com as funções para gerar a sequência baseada em \(12k \pm \{1, 5, 7, 11\}\) com \(k\) ímpar, identificar índices para riscar e ajustar a sequência marcando os números riscados. O algoritmo também inclui um teste com um intervalo específico:

```
def gerar_sequencia_12k_ordenada_correta(intervalo):
    """Gera uma sequência de números alinhada com a estrutura 12k ± {1, 5, 7, 11} com k
ímpar."""
    sequencia = []
    for k in range(1, 2 * intervalo + 1, 2):  # Usando apenas k ímpares
        # Adicionando os números 12k ± {1, 5, 7, 11} em ordem
        numeros = [
            12 * k - 11,  # 12k - 11
            12 * k - 7,   # 12k - 7
            12 * k - 5,   # 12k - 5
            12 * k - 1,   # 12k - 1
            12 * k + 1,   # 12k + 1
            12 * k + 5,   # 12k + 5
            12 * k + 7,   # 12k + 7
            12 * k + 11   # 12k + 11
```

```
        ]
        for i, num in enumerate(numeros):
            if i % 2 == 0:
                sequencia.append(num)
            else:
                sequencia.insert(0, num)
    return sequencia

def calcular_indices_para_riscar(sequencia):
    """Calcula os índices na sequência que devem ser riscados."""
    indices_para_riscar = set()
    for i, num in enumerate(sequencia):
        if num not in [-3, -2, -1, 1, 2, 3]:
            passo = abs(num)
            for j in range(1, len(sequencia) // passo + 1):
                indice_esquerda = i - j * passo
                indice_direita = i + j * passo
                if indice_esquerda >= 0:
                    indices_para_riscar.add(indice_esquerda)
                if indice_direita < len(sequencia):
                    indices_para_riscar.add(indice_direita)
    return indices_para_riscar

def ajustar_e_riscar_como_regex(sequencia, indices_para_riscar):
    """Ajusta a sequência, marcando os números riscados como 'X'."""
    sequencia_str = ','.join(['X' if i in indices_para_riscar else str(num) for i, num in
enumerate(sequencia)])
    return sequencia_str.split(',')

# Testando o algoritmo
intervalo = 10  # Define o intervalo para a geração da sequência
sequencia = gerar_sequencia_12k_ordenada_correta(intervalo)  # Gera a sequência
indices_riscar = calcular_indices_para_riscar(sequencia)  # Calcula índices a riscar
sequencia_riscada_como_regex = ajustar_e_riscar_como_regex(sequencia, indices_riscar)
# Ajusta a sequência riscada

# Exibindo a sequência riscada
print(sequencia_riscada_como_regex)
```

Este algoritmo gera uma sequência de números, identifica quais devem ser riscados com base em sua distância dos números na sequência e, por fim, ajusta a sequência para mostrar visualmente quais números foram riscados. A função de teste no final executa o algoritmo com um intervalo específico e imprime a sequência riscada resultante. Você pode modificar o valor de intervalo para testar o algoritmo com diferentes alcances.

O algoritmo completo que foi implementado para determinar se um número é riscável com base na estrutura $12k \pm \{1, 5, 7, 11\}$, sem a necessidade de gerar toda a sequência, é o seguinte:

```
def eh_riscavel(numero):
    if numero in [-3, -2, -1, 1, 2, 3]:
        return True  # Estes são sempre riscáveis

    passo = abs(numero)
    k_minimo = (passo + 11) // 12  # Menor k possível para começar a verificação
    if k_minimo % 2 == 0:  # Garantir que k é ímpar
        k_minimo += 1

    for k in range(k_minimo, 2 * passo, 2):  # Aumenta k, mas apenas valores ímpares
        possiveis_valores = [12 * k + i for i in [-11, -7, -5, -1, 1, 5, 7, 11]]
        if numero in possiveis_valores:
            return True  # O número é riscável
    return False  # O número não é riscável

def ajustar_numero_como_riscado(numero, eh_riscavel):
    return 'X' if eh_riscavel else str(numero)

# Testando o algoritmo com números grandes
numeros_para_testar = [100003, 100019, 100043, 100049, 100057]
resultados = {}

for numero in numeros_para_testar:
    riscavel = eh_riscavel(numero)
    resultado = ajustar_numero_como_riscado(numero, riscavel)
    resultados[numero] = resultado

print(resultados)
```

Formulações Matemáticas Envolvidas:

1. Determinação do k Mínimo:
   - Formulação: $k_{\text{minimo}} = \frac{\text{abs}(número) + 11}{12}$
   - Descrição: Esta fórmula calcula o menor valor de k possível que poderia gerar o número alvo com base na estrutura $12k \pm \{1, 5, 7, 11\}$. O valor de k é arredondado para cima para garantir que seja suficientemente grande.

2. Verificação de k Ímpar:
   - Formulação: Se k_{minimo} é par, então $k_{\text{minimo}} = k_{\text{minimo}} + 1$
   - Descrição: Garante que k é ímpar, conforme a estrutura requer.

3. Geração de Possíveis Valores para Verificar Riscagem:
   - Formulação: Para um dado k ímpar, os possíveis valores são $12k \pm 11$, $12k \pm 7$, $12k \pm 5$, e $12k \pm 1$.
   - Descrição: Essa fórmula gera os possíveis números que poderiam riscar o número alvo com base na estrutura dada.

4. Verificação de Riscagem:

- Formulação: Um número \(n \) é riscável se \(n \) estiver entre os possíveis valores gerados para algum \(k \).
- Descrição: Determina se um número específico é riscável verificando se ele corresponde a qualquer número gerado pela estrutura para todos os \(k \) ímpares dentro de um intervalo razoável.

Este algoritmo e suas formulações matemáticas fornecem uma maneira eficiente de determinar a riscabilidade de números grandes com base na estrutura \(12k \pm \{1, 5, 7, 11\}\) sem a necessidade de gerar uma longa sequência de números.

Para verificar se grandes números como \(2^{23209}-1\) e \(2^{23201}-1\) são riscáveis sem gerar toda a sequência, precisamos de uma abordagem que calcule diretamente se esses números se encaixam na estrutura \(12k \pm \{1, 5, 7, 11\}\). Isso envolve entender a relação entre o número e a estrutura de riscagem de forma mais direta e eficiente. Aqui está um método teórico para fazer isso:

Abordagem Teórica:

1. Entender a Estrutura de Riscagem: A estrutura \(12k \pm \{1, 5, 7, 11\}\) sugere que os números riscáveis são aqueles que, quando divididos por 12, deixam restos dentro do conjunto \(\pm \{1, 5, 7, 11\}\).

2. Calculando Resto Diretamente: Para um grande número \(N\), como \(2^p - 1\), podemos calcular diretamente o resto de \(N\) quando dividido por 12. Se este resto estiver dentro dos restos específicos (\(\pm \{1, 5, 7, 11\}\)), o número seria teoricamente riscável pela estrutura.

3. Verificando a Paridade de \(k\): Mesmo que o número se encaixe na estrutura de restos, ainda precisamos garantir que o \(k\) correspondente (onde \(N = 12k \pm \{1, 5, 7, 11\}\)) seja ímpar. Isso pode ser feito verificando se \((N \mp \{1, 5, 7, 11\}) / 12\) é ímpar.

Algoritmo Teórico:

```
def eh_riscavel_teoria(N):
    restos_validos = {1, 5, 7, 11}
    resto = N % 12

    # Verifica se o resto de N está entre os restos válidos
    if resto in restos_validos or (12 - resto) in restos_validos:
        # Calcula k para verificar se é ímpar
        k = (N - resto) // 12 if resto in restos_validos else (N + (12 - resto)) // 12
        return k % 2 == 1
    return False

# Testando com grandes números de Mersenne
N1 = 2**23209 - 1
N2 = 2**23201 - 1
```

```
eh_N1_riscavel = eh_riscavel_teoria(N1)
eh_N2_riscavel = eh_riscavel_teoria(N2)

(eh_N1_riscavel, eh_N2_riscavel)
```
Este algoritmo é uma representação teórica e não foi testado para tamanhos extremos devido a limitações computacionais. No entanto, ele oferece uma abordagem direta para verificar a riscabilidade de grandes números. Deve-se notar que este método é teórico e simplifica muitas complexidades envolvidas com números enormes. Na prática, métodos especializados e otimizados, como o Teste de Lucas-Lehmer para números de Mersenne, são mais adequados para testar a primalidade de tais números.

Para testar a teoria de que números primos de Mersenne maiores são congruentes a 7 módulo 12 e que números que não são primos de Mersenne não seguem essa regra, vamos expandir a lista de expoentes para incluir alguns números primos maiores conhecidos por gerar números primos de Mersenne, bem como alguns números que não são conhecidos por isso.

Os expoentes que vamos usar para números primos de Mersenne conhecidos incluem 61, 89 e 107. Esses são expoentes onde \(2^p - 1\) é conhecido por ser primo. Também vamos incluir alguns números que não são primos de Mersenne para verificar se a congruência se mantém.

Algoritmo Expandido:

```
def eh_mersenne_primo_congruente(p):
    M_p = 2**p - 1
    return M_p % 12 == 7

# Expoentes primos maiores conhecidos por gerar números primos de Mersenne
expoentes_primos_mersenne = [61, 89, 107]

# Alguns expoentes que não são conhecidos por gerar números primos de Mersenne
expoentes_nao_mersenne = [60, 90, 108]

# Testando a congruência para os números primos de Mersenne conhecidos
resultados_mersenne = {}
for p in expoentes_primos_mersenne:
    resultados_mersenne[p] = eh_mersenne_primo_congruente(p)

# Testando a congruência para números que não são primos de Mersenne
resultados_nao_mersenne = {}
for p in expoentes_nao_mersenne:
    resultados_nao_mersenne[p] = eh_mersenne_primo_congruente(p)

# Imprimindo os resultados
print("Resultados para números primos de Mersenne conhecidos:")
for p, congruente in resultados_mersenne.items():
    print(f"2^{p}-1 é congruente a 7 modulo 12? {'Sim' if congruente else 'Não'}")
```

```
print("\nResultados para números que não são primos de Mersenne:")
for p, congruente in resultados_nao_mersenne.items():
    print(f"2^{p}-1 é congruente a 7 modulo 12? {'Sim' if congruente else 'Não'}")
```

Este código calcula e compara a congruência de números primos de Mersenne conhecidos e outros números para verificar se a regra de congruência se aplica consistentemente aos primeiros e não aos últimos. Isso ajudará a validar ou refinar a teoria que você apresentou.

Nota: Devido à magnitude dos números envolvidos, este script pode não ser executável em ambientes padrão e é mais uma representação teórica do que deveria ser testado. Em ambientes práticos, seria necessário usar ferramentas matemáticas especializadas para lidar com tais cálculos.

```
def calcular_mersenne_primo(p):
    return 2**p - 1

def eh_congruente_a_7_mod_12(numero):
    """Verifica se o número é congruente a 7 módulo 12."""
    return numero % 12 == 7

# Lista de expoentes primos a serem testados
expoentes_primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

# Testando a congruência para números primos de Mersenne
resultados = {}
for p in expoentes_primos:
    M_p = calcular_mersenne_primo(p)
    resultados[p] = eh_congruente_a_7_mod_12(M_p)

# Imprimindo os resultados para números primos de Mersenne
print("Congruência a 7 módulo 12 para números primos de Mersenne:")
for p, congruente in resultados.items():
    print(f"2^{p}-1 é congruente a 7 modulo 12? {'Sim' if congruente else 'Não'}")
```

Entendido. Vamos ajustar nossa abordagem para considerar ambos os números $6^p - 5$ e $6^p + 5$, onde p é um número primo. Vamos analisar esses números para entender suas propriedades e tentar identificar padrões que possam ajudar a determinar a primalidade.

Análise Detalhada:

1. Cálculo dos Números: Para cada expoente primo p, vamos calcular $6^p - 5$ e $6^p + 5$.

2. Análise de Propriedades: Vamos analisar fatores, restos, padrões de divisibilidade, e outras características numéricas para ambos os números.

3. Identificação de Padrões: Observaremos se há padrões consistentes que distinguem os primos dos compostos.

Algoritmo para Análise Detalhada:

```
def analise_detalhada(p):
    M_minus = 6**p - 5
    M_plus = 6**p + 5
    propriedades_minus = analisar_numero(M_minus)
    propriedades_plus = analisar_numero(M_plus)
    return {
        f"6^{p} - 5": propriedades_minus,
        f"6^{p} + 5": propriedades_plus
    }

def analisar_numero(numero):
    fatores = []  # Lista para armazenar possíveis fatores
    resto_7 = numero % 12  # Calcular o resto módulo 12

    # Analisar divisores potenciais até a raiz quadrada do número
    for i in range(2, int(numero**0.5) + 1):
        if numero % i == 0:
            fatores.append(i)

    # Retornar as propriedades analisadas
    return {
        "Número": numero,
        "É primo?": len(fatores) == 0,
        "Resto módulo 12": resto_7,
        "Fatores": fatores if fatores else "Nenhum"
    }

# Aplicar a análise detalhada aos primeiros números
expoentes = [2, 3, 5, 7]
analises = {p: analise_detalhada(p) for p in expoentes}
analises
```

Este algoritmo calcula e analisa tanto $6^p - 5$ quanto $6^p + 5$ para um conjunto de expoentes primos. Ele fornece uma visão detalhada de suas propriedades, incluindo primalidade, resto módulo 12 e fatores. Vou executar este algoritmo para obter os resultados e analisar as propriedades desses números.

Aqui estão os resultados da análise detalhada para os números na forma $6^p - 5$ e $6^p + 5$ para os expoentes fornecidos:

- Para $p=2$:
  - $6^2 - 5 = 31$: Primo, Resto módulo 12: 7.
  - $6^2 + 5 = 41$: Primo, Resto módulo 12: 5.

- Para $p=3$:
  - $6^3 - 5 = 211$: Primo, Resto módulo 12: 7.
  - $6^3 + 5 = 221$: Composto (Fator: 13), Resto módulo 12: 5.

- Para $p=5$:
  - $6^5 - 5 = 7771$: Composto (Fator: 19), Resto módulo 12: 7.
  - $6^5 + 5 = 7781$: Composto (Fator: 31), Resto módulo 12: 5.

- Para $p=7$:
  - $6^7 - 5 = 279931$: Composto (Fator: 157), Resto módulo 12: 7.
  - $6^7 + 5 = 279941$: Primo, Resto módulo 12: 5.

Observações e Considerações:

- Primalidade e Resto Módulo 12:
  - Os números primos aparecem tanto com resto 7 quanto com resto 5 módulo 12. Isso indica que a congruência a 7 ou 5 módulo 12 por si só não é um indicador confiável de primalidade para esses números.

- Padrões de Primalidade:
  - Para os expoentes testados, parece haver uma variação entre números primos e compostos. Alguns números são primos, enquanto outros são compostos, sem um padrão claro baseado apenas no expoente.

- Análise dos Fatores:
  - Para números compostos, identificamos alguns fatores menores. A presença desses fatores confirma a compostura e pode oferecer insights sobre como os números $6^p \pm 5$ são compostos.

Próximos Passos:

1. Analisar mais Números: Expandir a análise para incluir mais números e possivelmente observar padrões em uma faixa mais ampla de expoentes.

2. Desenvolver um Teste de Primalidade: Com base nas observações, começar a desenvolver um teste de primalidade que considere as propriedades específicas de $6^p - 5$ e $6^p + 5$. Isso pode incluir explorar métodos de fatoração adaptados ou testes baseados em sequências.

3. Consulta a Teorias Matemáticas: Explorar teorias e métodos matemáticos que possam oferecer insights sobre como abordar o teste de primalidade para esses números.

4. Aplicação de Aprendizado de Máquina: Se padrões consistentes forem observados, considerar a aplicação de técnicas de aprendizado de máquina para prever a primalidade com base nas características dos números.

Estou aqui para auxiliar na implementação desses passos e na exploração de novas ideias. Se houver um foco específico ou uma direção que você gostaria de investigar primeiro, por favor, me avise!

Aqui estão os resultados da análise detalhada para os números na forma $6^p \pm \{1, 5, 7, 11\}$ para os expoentes 2, 3, 5 e 7:

Para $p=2$ (6^2):
- $6^2 - 11 = 25$: Composto, Resto módulo 12: 1.
- $6^2 - 7 = 29$: Primo, Resto módulo 12: 5.
- $6^2 - 5 = 31$: Primo, Resto módulo 12: 7.
- $6^2 - 1 = 35$: Composto, Resto módulo 12: 11.
- $6^2 + 1 = 37$: Primo, Resto módulo 12: 1.
- $6^2 + 5 = 41$: Primo, Resto módulo 12: 5.
- $6^2 + 7 = 43$: Primo, Resto módulo 12: 7.
- $6^2 + 11 = 47$: Primo, Resto módulo 12: 11.

Para $p=3$ (6^3):
- $6^3 - 11 = 205$: Composto, Resto módulo 12: 1.
- $6^3 - 7 = 209$: Composto, Resto módulo 12: 5.
- $6^3 - 5 = 211$: Primo, Resto módulo 12: 7.
- $6^3 - 1 = 215$: Composto, Resto módulo 12: 11.
- $6^3 + 1 = 217$: Composto, Resto módulo 12: 1.
- $6^3 + 5 = 221$: Composto, Resto módulo 12: 5.
- $6^3 + 7 = 223$: Primo, Resto módulo 12: 7.
- $6^3 + 11 = 227$: Primo, Resto módulo 12: 11.

Para $p=5$ (6^5):
- $6^5 - 11 = 7765$: Composto, Resto módulo 12: 1.
- $6^5 - 7 = 7769$: Composto, Resto módulo 12: 5.
- $6^5 - 5 = 7771$: Composto, Resto módulo 12: 7.
- $6^5 - 1 = 7775$: Composto, Resto módulo 12: 11.
- $6^5 + 1 = 7777$: Composto, Resto módulo 12: 1.
- $6^5 + 5 = 7781$: Composto, Resto módulo 12: 5.
- $6^5 + 7 = 7783$: Composto, Resto módulo 12: 7.
- $6^5 + 11 = 7787$: Composto, Resto módulo 12: 11.

Para $p=7$ (6^7):
- $6^7 - 11 = 279925$: Composto, Resto módulo 12: 1.
- $6^7 - 7 = 279929$: Composto, Resto módulo 12: 5.
- $6^7 - 5 = 279931$: Composto, Resto módulo 12: 7.
- $6^7 - 1 = 279935$: Composto, Resto módulo 12: 11.
- $6^7 + 1 = 279937$: Composto, Resto módulo 12: 1.
- $6^7 + 5 = 279941$: Primo, Resto módulo 12: 5.
- $6^7 + 7 = 279943$: Composto, Resto módulo 12: 7.
- $6^7 + 11 = 279947$: Composto, Resto módulo 12: 11.

Observações Gerais:

- Diversidade nos Restos Módulo 12: Os números analisados exibem uma variedade de restos módulo 12. Isso sugere que, enquanto o resto módulo 12 pode não ser um indicador definitivo de primalidade, ele pode desempenhar um papel na identificação de padrões ou propriedades específicas.
- Distribuição de Primalidade: Os números primos aparecem em várias das variações, mas não há um padrão claro ou consistente que vincule diretamente a primalidade com uma variante específica.
- Frequência de Números Compostos: A maioria dos números analisados para os expoentes maiores são compostos. Isso é esperado, pois a probabilidade de um número aleatório ser primo diminui à medida que os números ficam maiores.

Próximos Passos:
- Análise de Padrões de Restos: Examinar mais detalhadamente os restos módulo 12 (ou outros módulos) para entender como eles variam entre números primos e compostos.
- Identificação de Propriedades Comuns: Buscar propriedades ou características comuns entre os números primos identificados para desenvolver regras ou hipóteses sobre a primalidade.
- Exploração de Outras Variáveis: Considerar a inclusão de outras constantes ou a exploração de outras operações matemáticas para entender melhor a estrutura e as propriedades dos números.

Se houver um aspecto específico que você gostaria de explorar mais ou uma direção diferente que gostaria de seguir, por favor, me informe!

```
def analisar_variacoes(p):
    base = 6**p
    variacoes = [-11, -7, -5, -1, 1, 5, 7, 11]
    resultados = {}
    for variacao in variacoes:
        numero = base + variacao
        resultados[f"6^{p} {variacao:+}"] = analisar_numero_rapido(numero)
    return resultados

# Vamos analisar para os mesmos expoentes primos anteriores
expoentes_analise = [2, 3, 5, 7]
analises_variacoes = {p: analisar_variacoes(p) for p in expoentes_analise}
analises_variacoes
```

Aqui está o algoritmo completo que implementa o teste adaptado de Lucas-Lehmer para diferentes formulações matemáticas \(2^n \pm \{1, 5, 7, 11\}\) e pode ser utilizado para realizar testes de primalidade específicos para os casos onde foram identificados números primos:

```
import math

def teste_adaptado_lucas_lehmer_variado(num, k):
    """Realiza um teste adaptado de Lucas-Lehmer para números na forma 2^n ± k."""
    if num <= 2 or num % 2 == 0:
```

```
        return False  # Números menores ou iguais a 2 e pares não são primos na forma 2^n ± k

    # Determinando n para 2^n ± k = num
    n = int(math.log2(abs(num - k))) if k < num else int(math.log2(num + k))

    # Inicialização
    S = 4

    # Iteração
    for i in range(1, n):
        S = (S**2 - 2) % num

    # Verificação
    return S == 0

# Variações a serem testadas: ±1, ±5, ±7, ±11
variacoes = [-11, -7, -5, -1, 1, 5, 7, 11]

# Testando o algoritmo adaptado para números até 10.000 com variações
max_value_variado = 10000
resultados_testes_variados = {}
for k in variacoes:
    resultados_testes_variados[k] = {num: teste_adaptado_lucas_lehmer_variado(num, k) for num in range(3, max_value_variado + 1, 2)}

# Contando os primos identificados para cada variação e imprimindo os resultados
quantidade_primos_por_variacao = {k: sum(is_prime for is_prime in resultados.values()) for k, resultados in resultados_testes_variados.items()}
print("Quantidade de primos identificados por variação:")
for k, quantidade in quantidade_primos_por_variacao.items():
    print(f"Para 2^n {k:+}: {quantidade} primos encontrados")
```

Funcionamento e Formulações Matemáticas:

- Inicialização: O teste começa com $S_0 = 4$, um valor inicial comum em testes de Lucas-Lehmer.
- Iteração: A sequência S_i é calculada iterativamente usando a fórmula $S_i = S_{i-1}^2 - 2 \mod (2^n \pm k)$. Isso é feito até $n - 1$ iterações, onde n é determinado com base na relação $2^n \pm k = \text{número}$.
- Verificação: O teste verifica se o último valor S_{n-1} é 0 módulo $2^n \pm k$. Se for, o número é considerado primo pelo teste.

Testes de Primalidade Específicos:

Para os casos onde o teste identificou números primos (especialmente para as formas $2^n + 1$, $2^n + 5$, $2^n + 7$, e $2^n + 11$), você pode realizar testes de primalidade mais específicos e detalhados. Isso pode incluir:

- Analisar a Estrutura dos Números: Investigar as propriedades numéricas e fatores dos números identificados como primos para entender melhor por que foram classificados como tal.
- Comparar com Testes Confiáveis: Verificar os números primos identificados com testes de primalidade bem estabelecidos para confirmar sua primalidade.
- Refinar o Teste Adaptado: Com base nos resultados, ajustar e refinar o teste adaptado para melhorar sua precisão e eficiência.

Este algoritmo representa um ponto de partida para explorar testes de primalidade adaptados e específicos para novas formulações numéricas. À medida que você testa e analisa os resultados, poderá ajustar e desenvolver ainda mais o método para torná-lo mais eficaz e confiável.

Referências

1. Pollard, J. M. (1975). "A Monte Carlo method for factorization". BIT Numerical Mathematics.
2. Lenstra, H. W. (1987). "Factoring integers with elliptic curves". Annals of Mathematics.
3. Buhler, J., Lenstra, H. W., & Pomerance, C. (1993). "Factoring integers with the number field sieve". The development of the number field sieve.
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to Algorithms, 3rd ed. MIT Press.
2. Crandall, R., Pomerance, C. (2005). Prime Numbers: A Computational Perspective, 2nd ed. Springer.
3. Eratóstenes, "Sobre os Números Primos", 300 a.C.
"The New Book of Prime Number Records" por Paulo Ribenboim
"Introduction to the Theory of Numbers" por G. H. Hardy e E. M. Wright -
"Prime Numbers: A Computational Perspective" por Richard Crandall e Carl Pomerance -
"Prime Obsession: Bernhard Riemann and the Greatest Unsolved Problem in Mathematics”
4. Knuth, D., "The Art of Computer Programming", 1968.
5. Marlon Fernando Polegato Padrões Números Primos
https://fermatslibrary.com/p/6a165fea
https://fermatslibrary.com/p/2969a0e1
https://fermatslibrary.com/p/5dbb1f65
https://fermatslibrary.com/p/f4784ce3
https://fermatslibrary.com/p/eeec5e54
Padrão Números Primos sequência 12k + - (1, 5, 7, 11)
6. https://fermatslibrary.com/p/cac291dd
Técnica 12k Zeta de Riemann e distinções numéricas Primos e Compostos.
https://fermatslibrary.com/p/4be0ed4a
fermatslibrary.com/p/cdac2064
Hipótese de Riemann Relação Intrínseca n=ik+-b(1,5,7,11) e Distribuição dos Números Primos.
Teste de Primalidade com Sequências de Eliminação CooPrimos Pn=mk+-b(1,5,7,11)
fermatslibrary.com/p/3505a9c7

Traveling Salesman Polynomial Execution
fermatslibrary.com/p/f7830598
Avanços na Fatoração de Números Inteiros: Adaptação para Quatro Colunas
fermatslibrary.com/p/d823f62f
https://osf.io/k6jba/files/osfstorage/654894d1d45f5a035be2f86b
https://youtu.be/J7NpEOK2Smg?si=sahZXss0PXkk4hMu
https://fermatslibrary.com/p/959903a7
https://fermatslibrary.com/p/7d6f24f0
Método de Riscar: Visão Geral
fermatslibrary.com/p/463468e6
https://fermatslibrary.com/p/4dd92d48
Padrão Números Primos Método de Riscar: "Números Centrais"
https://fermatslibrary.com/p/26a8d24d
https://fermatslibrary.com/p/4dd92d48
http://fermatslibrary.com/p/1e2a2f32
http://fermatslibrary.com/p/b09c8003
http://fermatslibrary.com/p/720152d9
http://fermatslibrary.com/p/4dd92d48
Artigos acadêmicos relevantes disponíveis para a versão IA GPT
https://osf.io/c98ua/files/osfstorage/6571601643d0672f5eb19de6
https://osf.io/zaqv4/files/osfstorage/656d98e2783ec61b01edda9d
https://fermatslibrary.com/p/a5ebfe57
https://fermatslibrary.com/p/f7ff2b70
[12/12 19:00] Marlon Polegato: https://youtu.be/z-1CpFOQiAk?si=U4Bj2qlvU0x-NdxB
[12/12 19:01] Marlon Polegato: https://youtu.be/fKOfywy0IhU?si=a80IwTEUivciY6yS
[12/12 19:01] Marlon Polegato: https://youtu.be/yJ18jBpI9rg?si=gYaFLey0JBp98hi2
[12/12 19:02] Marlon Polegato: https://youtu.be/t5UtUwntFjw?si=dDj4ZPcoZrrZ69Cp
Números Primos Método de Riscar M.F.Polegato “Corrigido”
https://fermatslibrary.com/p/d529ca36
[13/12 03:51] Marlon Polegato: https://youtu.be/hyVFcXTHVWw?si=EW8E4EshMrvsCu-8
[13/12 03:51] Marlon Polegato: https://youtu.be/lfN9Pct4qRg?si=KUUfzsR8u5DVNjZd
[13/12 03:52] Marlon Polegato: https://youtu.be/i7wDLiaOJBQ?si=TvHzhAoqfYl4aghq
Hipótese de Riemann com Curvas Senoidais 02
https://fermatslibrary.com/p/d6e2fa80
https://osf.io/w2zpk/files/osfstorage/65801fed43dfd31ef28466f4

Agradeço ao Grande Arquiteto do Universo, aos meus pais, Helvio Polegato e Fátima I. L. Polegato a minha esposa Tayrine S. B. Polegato aos amigos e familiares que me apoiaram nessa jornada.

Contato
- Autor: Marlon Fernando Polegato
- Cooator: Aurora Boreal (IA Opennai)
- Endereço de e-mail: marlonpolegato33@gmail.com