```python
import random
import time
import pandas as pd
import matplotlib.pyplot as plt

# Função para fazer a exponenciação modular
# Retorna (x^y) % p
def potenciacao_modular(x, y, p):
    resultado = 1
    x = x % p  # Atualiza x se for maior que p
    while y > 0:
        if (y & 1):  # Se y for ímpar, multiplica x pelo resultado
            resultado = (resultado * x) % p
        y = y >> 1  # y = y // 2
        x = (x * x) % p
    return resultado

# Função que executa o teste de Miller
# Retorna False se n for composto e True se n for provavelmente primo
def teste_miller(d, n):
    a = 2 + random.randint(1, n - 4)  # Escolhe um 'a' aleatório no intervalo [2, n-2]
    x = potenciacao_modular(a, d, n)
    if x == 1 or x == n - 1:
        return True
    while d != n - 1:
        x = (x * x) % n
        d *= 2
        if x == 1:
            return False
        if x == n - 1:
            return True
    return False

# Função principal que aplica o Teste de Miller-Rabin
# Retorna False se n é composto, True se n é provavelmente primo
def eh_primo(n, k):
    if n <= 1 or n == 4:
        return False
    if n <= 3:
        return True
    d = n - 1
    while d % 2 == 0:
        d //= 2
    for _ in range(k):
        if not teste_miller(d, n):
            return False
    return True

# Função para gerar um número primo de 'bits' bits
def gerar_primo(bits, k=10):
    while True:
        # Gera número ímpar aleatório
        numero = random.getrandbits(bits) | 1 | (1 << (bits - 1))
        if eh_primo(numero, k):
            return numero

# Gerar a tabela de primos e tempos
bits_teste = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
tabela_resultados = []
```

```python

for bits in bits_teste:
    inicio = time.time()
    try:
        primo = gerar_primo(bits)
        fim = time.time()
        tempo = fim - inicio
    except Exception as e:
        primo = None
        tempo = None
    tabela_resultados.append({
        "Algoritmo": "Miller-Rabin",
        "Tamanho do Número (bits)": bits,
        "Número Primo Gerado": primo,
        "Tempo para Gerar (s)": tempo
    })

# Mostrar tabela
df = pd.DataFrame(tabela_resultados)
print(df)

# Plotar gráfico
plt.figure(figsize=(10,6))
plt.plot(df["Tamanho do Número (bits)"], df["Tempo para Gerar (s)"], marker='o')
plt.title("Tempo para Gerar Números Primos usando Miller-Rabin")
plt.xlabel("Tamanho do Número (bits)")
plt.ylabel("Tempo para Gerar (s)")
plt.grid(True)
plt.show()

# -----------------------------------------------------------------------------------------
####### Explicação do Algoritmo de Miller-Rabin: #######

### Função eh_primo(n, k): ###
# 1) Trata os casos base para n < 3.
# 2) Se n é par, retorna False.
# 3) Encontra um número ímpar d tal que n-1 possa ser escrito como d*2^r.
#     Como n é ímpar, n-1 é par e r deve ser maior que 0.
# 4) Executa k iterações:
#     Se teste_miller(d, n) retornar False, retorna False imediatamente.
# 5) Se todas as iterações passarem, retorna True.

#Função teste_miller(d, n):
# 1) Escolhe um número aleatório 'a' no intervalo [2, n-2].
# 2) Calcula x = a^d % n.
# 3) Se x == 1 ou x == n-1, retorna True.
# 4) Caso contrário:
#     Enquanto d não chegar a n-1:
#      a) Atualiza x = (x*x) % n.
#      b) Se x == 1, retorna False.
#      c) Se x == n-1, retorna True.
# 5) Se o laço terminar, retorna False.

"""
Exemplo de Execução (n = 13, k = 2):
- Encontramos d = 3, r = 2, pois 13-1 = 12 = 3*2^2.
- Primeira iteração:
    Escolhemos a = 4, calculamos x = 4^3 % 13 = 12.
    Como x = n-1, retornamos True.
```

```
- Segunda iteração:
    Escolhemos a = 5, calculamos x = 5^3 % 13 = 8.
    Como x não é 1 nem n-1:
    - Calculamos x = (8*8) % 13 = 12.
    - x agora é n-1, então retornamos True.
- Como ambas as iterações retornaram True, concluímos que 13 é provavelmente primo.
"""
# -----------------------------------------------------------------------------------------

### Observações ###:
# - A função eh_primo aplica o teste múltiplas vezes para aumentar a confiança.
# - A geração de números primos maiores (especialmente acima de 1024 bits) pode demorar bastante (provavelmente devido à baixa "densidade" de primos nessa faixa).
# - Para números de 2048 e 4096 bits, a geração levou vários minutos.
# - Nem sempre o primeiro número aleatório gerado é primo, então vários testes foram necessários.

# -----------------------------------------------------------------------------------------

# This code is contributed by mits https://www.geeksforgeeks.org/fermat-method-of-primality-test/

```