```python
import sympy
import random
import time
import matplotlib.pyplot as plt

# Função que encontra o próximo primo válido para o BBS (p ≡ 3 mod 4)
def proximo_primo_valido(x):
    p = sympy.nextprime(x)
    while p % 4 != 3:
        p = sympy.nextprime(p)
    return p

# Função para gerar bits pseudo-aleatórios e retornar detalhes
def gera_bits_pa_bbs(semente, p, q, N):
    M = p * q
    x = semente
    bit_output = ""
    for _ in range(N):
        x = (x * x) % M
        b = x % 2
        bit_output += str(b)
    num_zeros = bit_output.count("0")
    num_uns = bit_output.count("1")
    return bit_output, num_zeros, num_uns, M

# Função principal que testa diversos tamanhos e exibe os dados estendidos
def testa_tamanhos(tamanhos):
    resultados = []
    for tamanho in tamanhos:
        x = random.randint(1, 10**10)
        y = random.randint(1, 10**10)
        p = proximo_primo_valido(x)
        q = proximo_primo_valido(y)
        semente = random.randint(1, 10**10)
        N = tamanho

        tempo_inicio = time.time()
        bits, zeros, uns, M = gera_bits_pa_bbs(semente, p, q, N)
        tempo_fim = time.time()
        tempo_total = tempo_fim - tempo_inicio

        # Adiciona informações completas no resultado
        resultados.append({
            "tamanho": tamanho,
            "time": tempo_total,
            "p": p,
            "q": q,
            "M": M,
            "semente": semente,
            "bits": bits,
            "zeros": zeros,
            "uns": uns
        })

    return resultados

# Tamanhos dos números a serem gerados (em bits)
tamanhos = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

# Executa o teste
```

```
resultados = testa_tamanhos(tamanhos)

# Exibe o tempo de execução
print("Algoritmo: Blum Blum Shub")
print(f"{'Tamanho (bits)':<15}{'Tempo (segundos)'}")
for r in resultados:
    print(f"{r['tamanho']:<15}{r['time']:.6f}")

print("\nDetalhes para cada tamanho testado:")
for r in resultados:
    print(f"\np:  {r['p']}")
    print(f"q:  {r['q']}")
    print(f"M:  {r['M']}")
    print(f"Semente:  {r['semente']}")
    print(f"{r['bits']}")
    print(f"Número de zeros:    {r['zeros']}")
    print(f"Número de uns:     {r['uns']}")

###### Explicação do Código ######

### Funções auxiliares ###

# proximo_primo_valido: Encontra o próximo número primo válido para o algoritmo BBS. Isto é, um número primo congruente a 3 módulo 4.

# gera_bits_pa_bbs: Gera uma sequência de bits pseudo-aleatórios usando o BBS. Para cada bit gerado, o valor de x é elevado ao quadrado e reduzido módulo M (produto de dois primos p e q), e o bit é o valor de x % 2.

# testa_tamanhos: Testa a geração de números de vários tamanhos (em bits), medindo o tempo necessário para gerar cada sequência de bits. O código gera números primos p e q para cada execução, além de uma semente aleatória. A função retorna uma lista com os tempos de geração.

# Tabela de Resultados: checar output do programa no terminal.

###### Possíveis Limitações ######
# Tempo de Geração: Para tamanhos de número muito grandes, como 2048 ou 4096 bits, o tempo de execução pode aumentar significativamente devido à natureza computacionalmente intensa do algoritmo.
# Limitações de Memória: Gerar números com tamanhos muito grandes pode exigir uma quantidade significativa de memória, especialmente para valores próximos a 4096 bits.
# Talvez o Algoritmo Não Funcione: Para tamanhos extremamente grandes, como 4096 bits, a execução do algoritmo pode ser inviável em sistemas com recursos limitados ou em contextos que exigem um tempo de resposta muito rápido. Isso ocorre devido à complexidade do algoritmo, que cresce conforme o tamanho do número aumenta. Em tais casos, pode ser necessário utilizar outras abordagens, como algoritmos de geração de números pseudo-aleatórios mais rápidos ou usar bibliotecas dedicadas a números grandes (que são otimizadas para eficiência).

###### Interpretação técnica ######
# O tempo de geração é proporcional ao número de iterações do laço (for _ in range(N)) em gera_bits_pa_bbs(). Cada iteração envolve uma operação de exponenciação modular x = (x * x) % M, que é computacionalmente "cara".
# O custo dessas operações aumenta conforme o tamanho de M = p * q, que é diretamente influenciado pelos primos escolhidos. Mesmo assim, o uso de primos relativamente pequenos (~10 dígitos) ajuda a manter o desempenho aceitável.
# O resultado demonstra boa escalabilidade para tamanhos de chave utilizados em segurança (ex: 1024, 2048 bits).
```

```
100
101 ###### Referências ######
102 # Adapted from:
103 # https://medium.com/asecuritysite-when-bob-met-alice/cryptography-in-the-family-
    blum-blum-and-blum-6277590f0c94
104 # https://asecuritysite.com/encryption/blum
105 # All credits to the author.
106
107 ###### Geração de Gráfico ######
108 # Visualização da relação entre o tamanho da sequência de bits e o tempo de
    execução.
109
110 # Extraindo dados para o gráfico
111 tamanhos_bits = [r['tamanho'] for r in resultados]
112 tempos_execucao = [r['time'] for r in resultados]
113
114 # Criando o gráfico
115 plt.figure(figsize=(10,6))
116 plt.plot(tamanhos_bits, tempos_execucao, marker='o', linestyle='-', color='b')
117 plt.title('Tempo de Geração de Bits usando Blum Blum Shub')
118 plt.xlabel('Tamanho da sequência (bits)')
119 plt.ylabel('Tempo de execução (segundos)')
120 plt.grid(True)
121 plt.xticks(tamanhos_bits, rotation=45)
122 plt.tight_layout()
123 plt.show()
124
```