Cálculo da n-ésima potência

Trabalho 1 - Análise de Algoritmos

Vitor Hugo Honorato Tiago

Definição do Problema

- Calcular n-ésima potência de um número
- Para todos os testes de execução foi usada a base 2
- $2^x = 2 . 2 . 2 ... 2 . 2$
- Foram usadas três abordagens para solução do problema
    - Força bruta
    - Algoritmo Divide and Conquer ingênuo
    - Algoritmo Divide and Conquer otimizado

Algoritmo convencional

```python
# Function to calculate power without Divide and Conquer

def power(x, n):
    pow = 1

    for i  in range(n):
        pow = pow * x

    return pow
```

Custo	Quantidade de execuções
c_4	1
c_6	$(n + 1)$
c_7	n
c_9	1

$f(n) = c_4 + c_6(n + 1) + c_7(n) + c_9$

$f(n) = c_4 + nc_6 + c_6 + nc_7 + c_9$

$f(n) = (c_6 + c_7)n + (c_4 + c_6 + c_9)$

Todas as linhas só fazem comparações e atribuições, logo seu custo é constante. Supondo $a = c_6 + c_7$ e $b = c_4 + c_6 + c_9$, temos uma função afim, com seu crescimento linear.

f(n) tem crescimento linear, $f(n)$ é $\Theta(n)$

Algoritmo Divide and Conquer ingênuo

```
1 # Function to calculate power with Divide and Conquer
2
3 def recursivePower(x, n):
4     if n == 0:
5       return 1;
6
7     if n%2 == 0:
8       return recursivePower(x, n/2) * recursivePower(x, n/2)
9
10    return x * recursivePower(x, (n-1)/2) * recursivePower(x, (n-1)/2)
```

$$T(n) = \begin{cases} \theta(1), & \text{se } n = 0 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{se } n > 0 \end{cases}$$

$$D(n) = \theta(1) \quad C(n) = \theta(1)$$

$$T(n) = \begin{cases} \theta(1), & \text{se } n = 0 \\ 2T\left(\frac{n}{2}\right) + \theta(1) + \theta(1), & \text{se } n > 0 \end{cases}$$

Divisão do problema de tamanho n em dois subproblemas de tamanho $\frac{n}{2}$.

Para a divisão do problema são realizadas apenas a operação aritmética $\frac{n}{2}$, logo o tempo para divisão do problema é constante.

Para combinação dos subproblemas são realizadas a linha 8 ou 10. Independente de qual linha ser executada sempre serão realizadas 2 ou 3 multiplicações, logo podemos considerar como constante.

O caso base na linha 5, somente retorna 1, logo também tem tempo constante.

Cálculo Complexidade Recursiva

Resolvendo pelo método da expansão:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + 1\right] + 1$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 2 + 1$$

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + 1\right] + 2 + 1$$

$$T(n) = 8T\left(\frac{n}{8}\right) + 4 + 2 + 1$$

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} 2^j$$

$$\left(\frac{n}{2^i}\right) = 1 \qquad i = \log_2 n$$

$$T(n) = 2^{\log_2 n} T(1) + \sum_{j=0}^{\log_2 n - 1} 2^j$$

$$T(n) = 3n + n - 1$$

$$T(n) = 4n - 1$$

$T(n)$ é $\theta(n)$

$$T(1) = 2T(0) + 1$$

$$T(0) = 1$$

Algoritmo Divide and Conquer otimizado

```
1  # Function to calculate power with Divide and Conquer optimized
2
3  def recursivePowerOptimized(x, n):
4      if n == 0:
5        return 1;
6      temp = recursivePowerOptimized(x, int(n/2))
7      if n%2 == 0:
8        return temp * temp
9
10     return x * temp * temp
```

$$T(n) = \begin{cases} \theta(1), & \text{se } n = 0 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{se } n > 0 \end{cases}$$

$$D(n) = \theta(1) \quad C(n) = \theta(1)$$

$$T(n) = \begin{cases} \theta(1), & \text{se } n = 0 \\ T\left(\frac{n}{2}\right) + \theta(1) + \theta(1), & \text{se } n > 0 \end{cases}$$

Divisão do problema de tamanho n em um único subproblema de tamanho $\frac{n}{2}$.

Para a divisão do problema são realizadas apenas a operação aritmética $\frac{n}{2}$, logo o tempo para divisão do problema é constante.

Para combinação dos subproblemas são realizadas a linha 8 ou 10. Independente de qual linha ser executada sempre serão realizadas 2 ou 3 multiplicações, logo podemos considerar como constante.

O caso base na linha 5, somente retorna 1, logo também tem tempo constante.

Cálculo abordagem recursiva 2

Resolvendo pelo método da expansão:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$$

$$T(1) = T(0) + 1$$

$$T(0) = 1$$

$$T(n) = T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i} 1$$

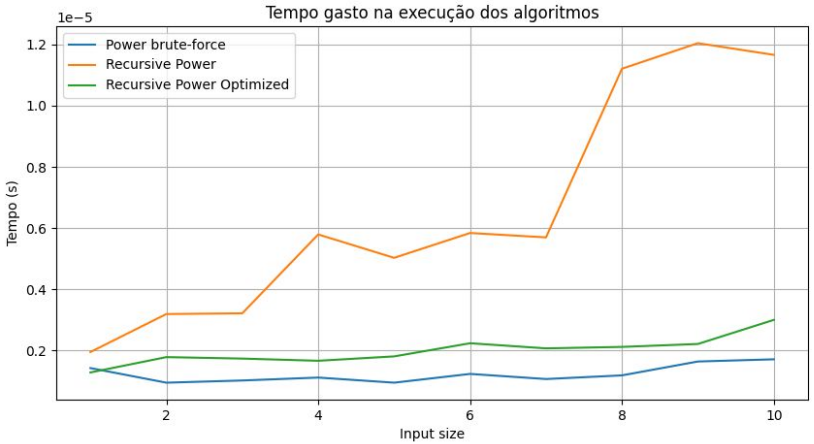
$$\left(\frac{n}{2^i}\right) = 1 \qquad i = \log_2 n$$

$$T(n) = T(1) + \sum_{j=0}^{\log_2 n} 1$$

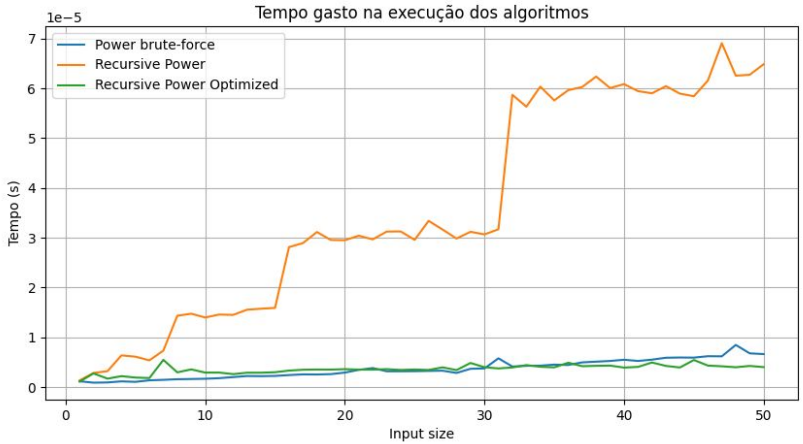
$$T(n) = 2 + \log_2 n + 1$$

$$T(n) \text{ é } \theta(\log\ n)$$

Execuções



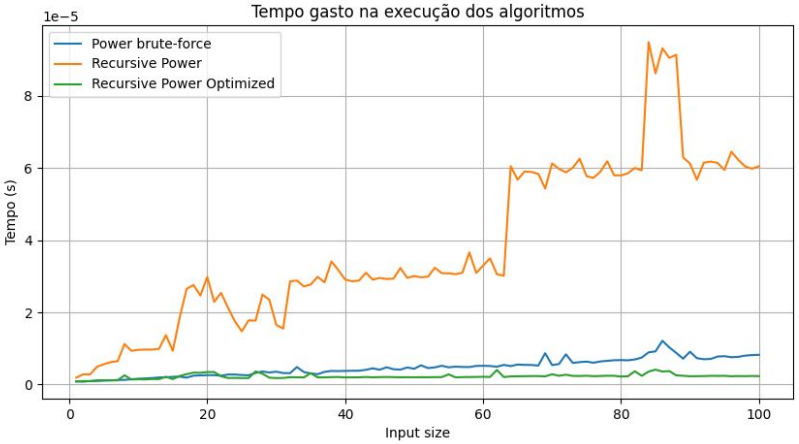
10 elementos



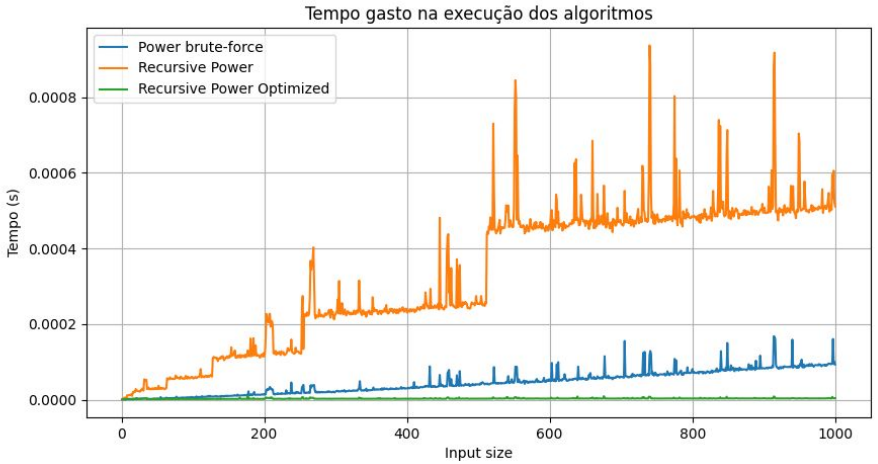
50 elementos

Entradas pequenas até 10.000 foram feitas 10 execuções e considerada a média do tempo gasto.

Execuções

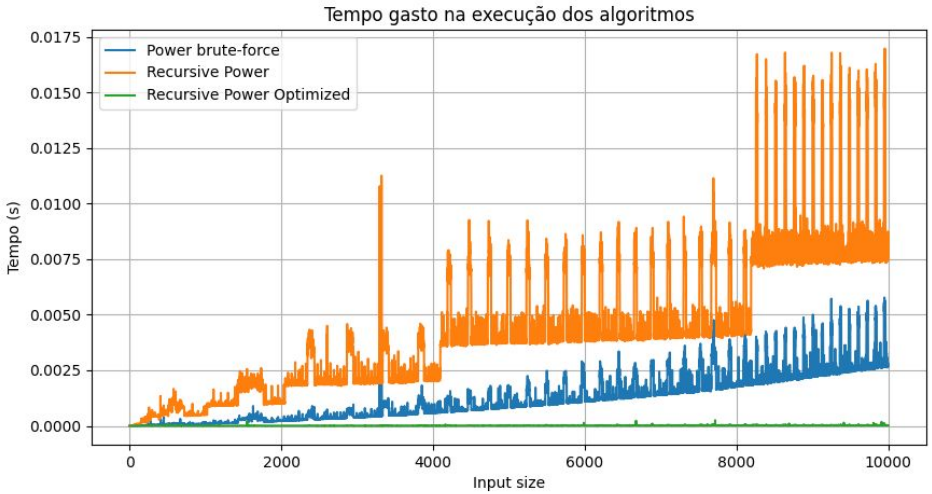


100 elementos

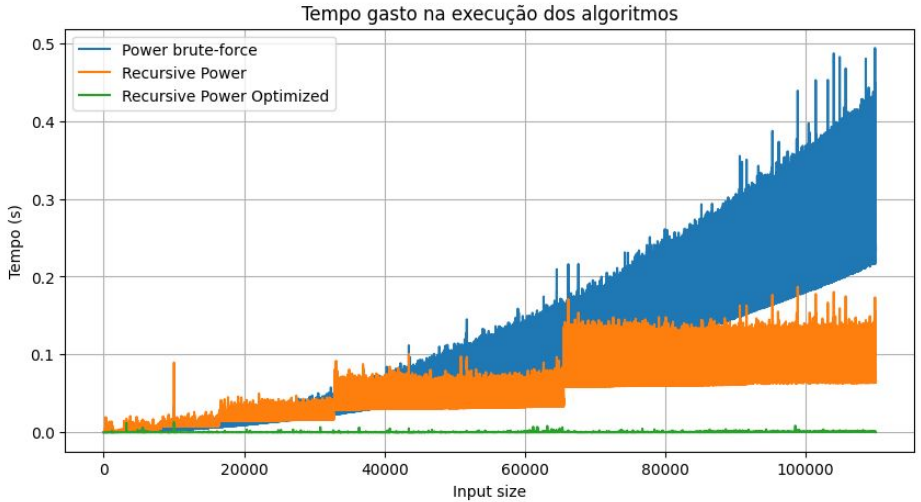


1000 elementos

Execuções

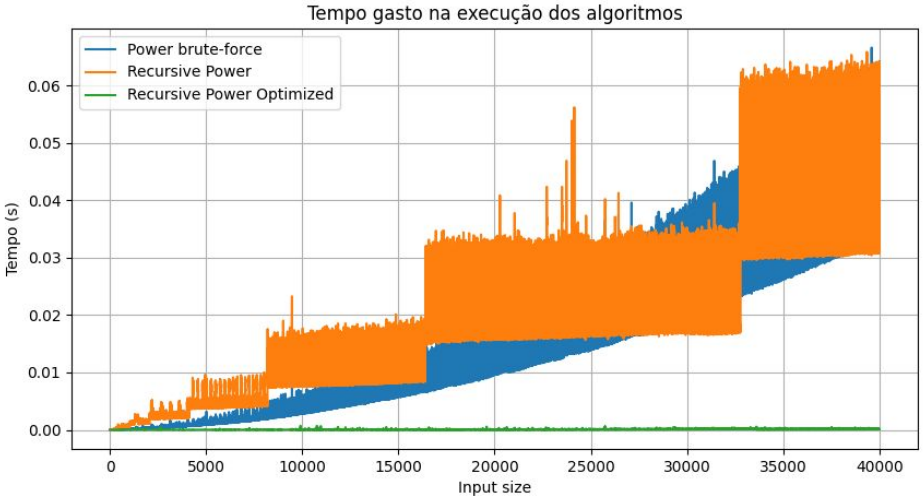


10000 elementos

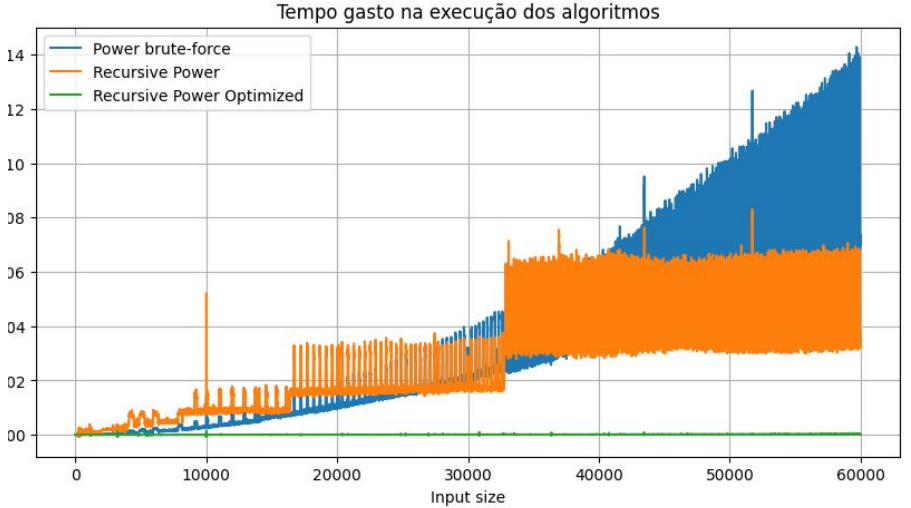


110000 elementos

Execuções

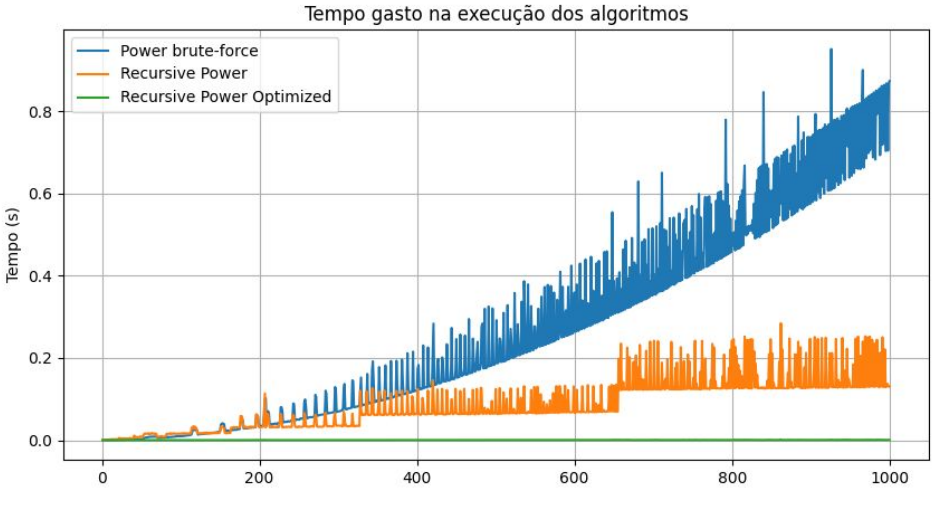


40000 elementos

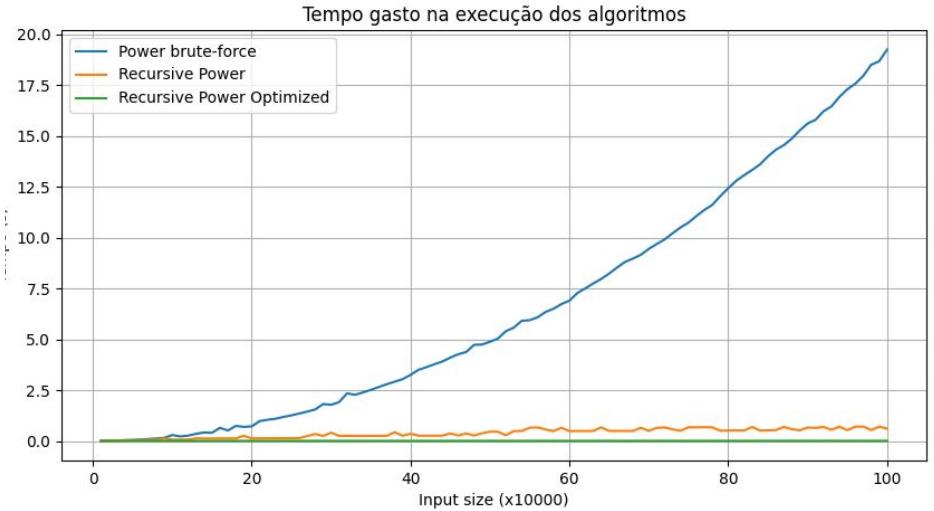


60000 elementos

Execuções



1000 elementos de 200 em 200



100 elementos de 10000 em 10000

Execuções

Entrada: 10, 100, 1.000, 10.000, 100.000, 1.000.000, 2.500.000, 5.000.000, 10.000.000

Execução de cada entrada 10 vezes e retirado a média de tempo de execução

Conclusões

- O algoritmo na força bruta tem complexidade linear.
- A técnica Divide and Conquer pode produzir tanto um algoritmo linear quanto log n.
- Para entradas pequenas, o algoritmo Divide and Conquer perde no tempo.
- No gráfico com 50 dados já é possível notar que o algoritmo de complexidade log n tem tempo menor que o de complexidade linear tradicional.
- Embora ambos tenham complexidade n ,o algoritmo Divide and Conquer ingênuo demora até uma entrada de 20.000 e começa a superar o tempo do algoritmo tradicional.
- O algoritmo com complexidade log n rapidamente se torna o melhor, enquanto o técnica Divide and Conquer com complexidade n demora um pouco, mas se torna melhor do que a abordagem tradicional, que também tem complexidade n.