Algoritmo de Karatsuba para Multiplicação de Números Grandes

Karatsuba | Multiplicação de números grandes

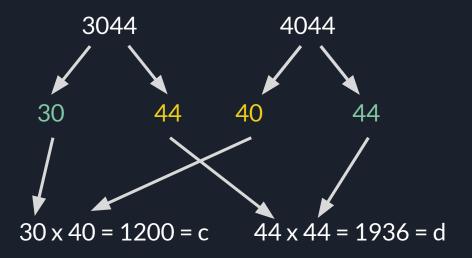
- Dados dois números muito grandes, o algoritmo de Karatsuba simplifica a multiplicação deles, evitando contas extensas
- Pode ser aplicado em números que ultrapassam o limite do inteiro de 4 bytes

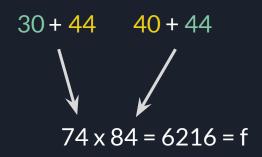
Implementação:

- Java
- Divisão e Conquista
- O protótipo será mostrado em pseudocódigo (parecido com Java) para fins de simplificação

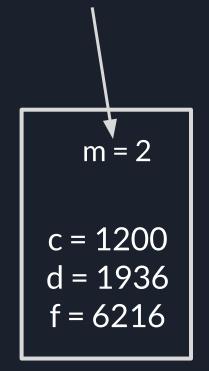
Karatsuba | Como funciona -Divisão

Exemplo: 3044 x 4044





Quantidade de dígitos dos novos números



Karatsuba | Como funciona - Conquista

A fórmula final entrega o resultado da multiplicação. É a soma entre:

- $c \times 10^{2m}$
- $(f c d) \times 10^m$
- d

$$c \times 10^{2m} + (f - c - d) \times 10^{m} + d$$

$$c = 1200$$

$$d = 1936$$

$$f = 6216$$

$$12'000'0000 + 307'6000 + 1'936$$

$$= 12'309'936$$

Karatsuba | Código

```
Number karatsuba(Number a, Number b)
    // Se ambos os números tiverem até 3 dígitos
    // Multiplicar diretamente
    if (a < 1000 | b < 1000) {
        return a * a;
    // Obtém o número com a maior quantidade de dígitos
    int m = Math.max(a.digits(), b.digits());
    // Obtém a metade de m, arredondando pra cima
    m = Math.coil(m / 2);
    // Divisões (de DC)
    // Divisão em al e a2
    long al = a * Math.pow(10, 4);
    long a2 = a % Math.pow(10, 4);
    // Divisão em b1 e b2
    long b1 = b * Math.pow(10, 4);
    long b2 = b % Math.pow(10, 4);
```

Karatsuba | Código

```
// Conquistas
// Multiplicação al x bl
long c = karatsuba(a1, b1);
// Multiplicação a2 x b2
long d = karatsuba(a2, b2);
// Multiplicação ( a1 + a2 ) x ( b1 + b2 )
long f = karatsuba(a1 + a2, b1 + b2);
// Resultado do Karatsuba
// = c * 10^{(m2)} + (f - c - d) * 10^{m} + d
// c = a1 \times b1
// d = a2 \times b2
// f = (a1 \times a2) + (b1 \times b2)
return (((10^m*2) * c) + (10^m) * (f - c - d) + d);
```

Karatsuba | Código

```
Number karatsuba (Number a, Number b)
    // Se ambos os números tiverem até 3 dígitos
   // Multiplicar diretamente
    if (a < 1000 | b < 1000) (
        return a * a;
    // Obtém o número com a maior quantidade de dígitos
    int m = Math.max(a.digits(), b.digits());
    // Obtém a metade de m, arredondando pra cima
    m = Math.coil(m / 2);
    // Divisões (de DC)
    // Divisão em al e a2
   long al = a * Math.pow(10, 4);
    long a2 = a % Math.pow(10, 4);
    // Divisão em b1 e b2
    long b1 = b * Math.pow(10, 4);
    long b2 = b % Math.pow(10, 4);
```

```
// Conquistas

// Multiplicação al x bl
long c = karatsuba(al, bl);

// Multiplicação a2 x b2
long d = karatsuba(a2, b2);

// Multiplicação ( al + a2 ) x ( bl + b2 )
long f = karatsuba(al + a2, bl + b2);

// Resultado do Karatsuba
// = c * 10^(m2) + (f - c - d) * 10^m + d
// c = al x bl
// d = a2 x b2
// f = ( al x a2 ) + ( bl x b2 )
return (((10^m*2) * c) + (10^m) * (f - c - d) + d);
```

Karatsuba | Teorema Mestre

Complexidade:
$$T(n) = 3T(n/2) + n$$

$$T(n) = 3T(rac{n}{2}) + n$$
 $a = 3; b = 2; f(n) = n$ $f(n) < n^{log_b a}
ightarrow n < n^{log_2 3}$

Logo, a solução da recorrência é $T(n) = heta(n^{log_23})$

Karatsuba | Execução

• Karatsuba \$ java karatsuba.java 3524365 55453655

Multiplicação por Karatsuba: (a x b)
a = 3524365
b = 55453655

(a x b) : 1954207042784075

Tempo de execução do algoritmo (em nanossegundos)
Início : 1759002372985724170
Fim : 1759002372985987595

Karatsuba | Execução

• Karatsuba \$ java karatsuba.java 3044 4044
Multiplicação por Karatsuba: (a x b)
a = 3044
b = 4044
(a x b) : 12309936
Tempo de execução do algoritmo (em nanossegundos)
Início : 1759002518341913614
Fim : 1759002518342237482

Número de Formas de Fazer Troco (Coin Change – Count)

Coin Change | Problema das Combinações de Moedas

Problema:

 Dado um conjunto de moedas e um troco a ser dado, de quantas maneiras podemos combinar as moedas para somar exatamente o troco?

Aplicações:

Sistemas de troco, problemas de combinação, algoritmos financeiros.

Vantagens:

- Abordagem da Programação Dinâmica -> quebra o problema em subproblemas menores
- Diferentemente da abordagem da Divisão e Conquista, há memorização de dados -> evita recálculos, e consegue ter eficiência O(n × m)

Coin Change | Contagem Eficiente usando Programação Dinâmica

Princípio Chave:

O número de formas de fazer troco para um valor X é igual à soma das formas de fazer troco para (X - valor_da_moeda) para cada moeda disponível.

Exemplo Prático:

```
Fase 1 - Processando a moeda de R$ 1,00:
```

Valor 1: formas[1] += formas[0] \rightarrow 0 + 1 = 1 maneira

Valor 2: formas[2] += formas[1] \rightarrow 0 + 1 = 1 maneira

Valor 6: formas[6] += formas[5] \rightarrow 0 + 1 = 1 maneira Resultado final: 2 maneiras de fazer R\$ 6,00

Resultado após moeda de 1: [1, 1, 1, 1, 1, 1, 1]

Fase 2 - Processando moeda de R\$ 5,00:

Valor 5: formas[5] += formas[0] \rightarrow 1 + 1 = 2 maneiras

Valor 6: formas[6] += formas[1] \rightarrow 1 + 1 = 2 maneiras

Coin Change | Contagem Eficiente usando Programação Dinâmica | Código

```
def contar_maneiras_de_fazer_troco(moedas: List[int], troco: int) -> int:

formas_por_valor = [0] * (troco + 1)
formas_por_valor[0] = 1 # Caso base: 1 maneira de fazer troco para valor 0

for moeda in moedas:
    for valor in range(moeda, troco + 1):
        formas_por_valor[valor] += formas_por_valor[valor - moeda]

return formas_por_valor[troco]

return formas_por_valor[troco]
```

Coin Change | Listagem de Combinações

Princípio Chave:

Para cada valor, para cada moeda, pegue todas as combinações que faltam apenas essa moeda e adicione-a.

Exemplo visual do processo de construção

Estado Inicial:

Valor 0: [[]] (apenas a combinação vazia)

Processando moeda de R\$ 1,00:

Valor 1: pega combinações do valor 0 e adiciona moeda $1 \rightarrow [[1]]$

Valor 2: pega combinações do valor 1 e adiciona moeda $1 \rightarrow [[1,1]]$

Valor 3: pega combinações do valor 2 e adiciona moeda $1 \rightarrow [[1,1,1]]$

...(até o valor 6)

Processando moeda de R\$ 5,00:

Valor 5: pega combinações do valor 0 e adiciona moeda $5 \rightarrow [[5]]$

Valor 6: pega combinações do valor 1 e adiciona moeda $5 \rightarrow [[1,5]]$

Coin Change | Listagem de Combinações | Código

```
def listar_combinacoes(moedas: List[int], troco: int) -> List[List[int]]:

moedas_ordenadas = sorted(moedas)

# combinacoes_possiveis[valor] armazena todas as combinações possíveis para aquele valor combinacoes_possiveis: List[List[int]]] = [[] for _ in range(troco + 1)]

combinacoes_possiveis[0].append([]) # Combinação vazia para valor 0

for moeda in moedas_ordenadas:

for valor in range(moeda, troco + 1):

# Para cada combinação que soma (valor - moeda), adicionamos a moeda atual

for combinacao in combinacoes_possiveis[valor - moeda]:

nova_combinacao = combinacao + [moeda]

combinacoes_possiveis[valor].append(nova_combinacao)

return combinacoes_possiveis[troco]
```

- "combinacoes_possiveis" é uma lista de listas de listas;
- "combinacoes_possiveis[5]" contém todas as combinações que somam R\$ 5,00;
- Cada combinação é uma lista de moedas: [1,1,1,1,1] ou [5].

Coin Change | Análise de Performance

Análise de Tempo de Execução:

Para troco = R\$ 26,00:

Por Que é Rápido? Operação Tempo

~0.001 ms Apenas $6 \times 26 = 156$ operações Contagem

matemáticas

Listagem ~0.005 ms Memorização de resultados

O Porque da abordagem Bottom-up:

- Constrói progressivamente •
- Reutiliza soluções menores Controle de memória
- Preenchimento ordenado
- Menos overhead
- Cache-friendly

Coin Change | Análise de Performance | Tempos de Execução

```
Moedas disponíveis: [1, 5, 10, 25, 50, 100]
Moedas disponíveis: [1, 5]
                                       troco: 26
troco: 6
                                       Total de combinações: 13
Total de combinações: 2
                                       === Lista de Todas as Combinações ===
=== Lista de Todas as Combinações ===
                                        1. (1+25) = 26
1. (1+5) = 6
                                        2. (1+5+10+10) = 26
2. (1+1+1+1+1+1) = 6
                                        3. (1+5+5+5+10) = 26
                                        4. (1+5+5+5+5+5) = 26
=== Tempo ===
                                        5. (1+1+1+1+1+1+10+10) = 26
Tempo para listagem: 0.010 milissegundos
                                       6. (1+1+1+1+1+1+5+5+10) = 26
Tempo para contagem: 0.007 milissegundos
                                        7. (1+1+1+1+1+1+5+5+5+5) = 26
                                        8. (1+1+1+1+1+1+1+1+1+1+1+5+10) = 26
Tempo total: 0.017 milissegundos
                                        9. (1+1+1+1+1+1+1+1+1+1+5+5+5) = 26
                                       10. (1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1) = 26
                                       11. (1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+5+5) = 26
                                       === Tempo ===
```

Tempo para listagem: 0.074 milissegundos Tempo para contagem: 0.013 milissegundos

Tempo total: 0.086 milissegundos

Repositório

Repositório contendo todos os conteúdos deste trabalho:

https://github.com/joao-lsf/aed2-trabalho1-gp4

Participantes

- João Luiz Schiavini Filho
- Felippe Carballo Leal
- Matheus Gonçalves do Nascimento Bandeira