# Laboratório 4: Paradigmas de Programação - Moedas de Troco (Parcial)

Caio Graça Gomes

16 de Junho de 2020

## 1 Introdução

Neste laboratório, foram implementadas três algoritmos distintos para resolução do problema das Moedas de Troco. Esses algoritmos se baseiam em conceitos de busca gulosa (*Greedy* GR), divisão e conquista (DC) e programação dinâmica (PD), sendo eles analisados pelas suas corretudes, alocação de memória e tempo despendido.

## 2 Algoritmos utilizados

### 2.1 *Greedy*

A ideia base para implementação do algoritmo *Greedy* foi: enquanto possível, usar a moeda de maior valor possível para alcançar o valor do troco, quando não for mais possível, usar a moeda de segundo menor valor e assim sucessivamente até que o valor do troco desejado seja alcançado. Isso gerou o seguinte código em $C++$:

```
void TrocoSolverGreedy::solve(const std::vector<unsigned int>
    &denom,unsigned int value, std::vector<unsigned int> &coins) {
   coins.resize(denom.size(),0);
   unsigned int change = 0;
   unsigned int pos = denom.size();
   for(std::vector<unsigned int>::const_reverse_iterator itr =
       denom.rbegin(); itr != denom.rend() && change < value; ++itr){
```

```
        --pos;
        while(change + *itr <= value){
            change += *itr;
            coins[pos]++;
        }
    }
}
```

Assim, obteve-se o seguinte gráfico para seu desempenho temporal quando submetido a testes de valores de troco crescentes:

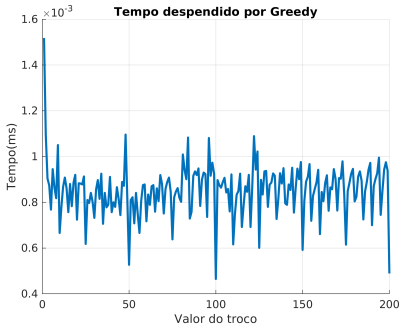


Figure 1: Tempo de execução do algoritmo Greedy

Nota-se que o tempo despendido pelo *Greedy* não apresenta muita correlação com o valor do troco. Ainda, o número de moedas para cada valor de troco foi:

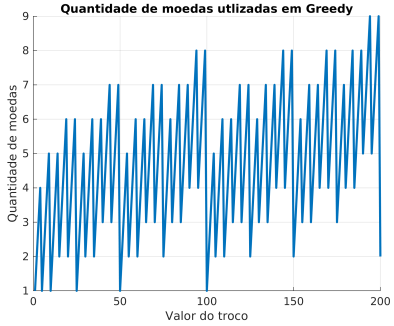


Figure 2: Quantidade de moedas necessárias para fornecer o respectivo troco no algoritmo Greedy

A quantidade de moedas exigidas pelo algoritmo aparenta apresentar um comportamento periódico que é regularmente incrementado por valores contantes.

## 2.2 Divisão e Conquista

A ideia principal do algoritmo de divisão e conquista é: se o valor de troco desejado for nulo, então não precisa adicionar moedas, caso contrário, deve-se utilizar a moeda cuja denominação faz com que seja necessário adicionar a menor quantidade de moedas possíveis anterior a esse valor de troco. Isso é feito numa lógica *top-down* recursiva, analisando a melhor estratégia para alcançar $troco = valordesejado$, depois recursivamente $troco = valordesejado - denominação_1$, $troco = valordesejado - denominação_2$, ..., $troco = valordesejado - denominação_n$ e toma-se a possibilidade que gerou menor quantidade de moedas total ao final.

Com isso, a implementação em $C++$ fica:

```
unsigned int DCMakeChange(const std::vector<unsigned int> &denom,
    unsigned int value, std::vector<unsigned int> &last){
    if(value == 0) return 0;
    unsigned int q = value, pos = 0, aux;
    for(std::vector<unsigned int>::const_iterator itr = denom.begin();
        itr != denom.end(); ++itr, ++pos){
        if(*itr > value) continue;
        aux = DCMakeChange(denom, value - *itr, last);
        if(q > 1 + aux){ q = 1 + aux; last[value] = pos;}
    }
    return q;
}


void TrocoSolverDivConquer::solve(const std::vector<unsigned int>
    &denom,unsigned int value, std::vector<unsigned int> &coins) {

    coins.resize(denom.size(),0);
    std::vector<unsigned int> last(value + 1);
    last[0] = 0;
    DCMakeChange(denom, value, last);
    for(unsigned int aux = value; aux > 0; coins[last[aux]]++, aux -=
        denom[last[aux]]);

}
```

Assim, o tempo despendido pelo algoritmo de divisão e conquista quando submetidos a crescentes valores de troco desejado foi:


Figure 3: Tempo despendido pelo algoritmo de Divisão e Conquista

Torna-se evidente, portanto, sua complexidade exponencial, o que sugere que não é uma solução desejável para o problema. Além disso, a quantidade de moedas utilizadas pelo DC foi:

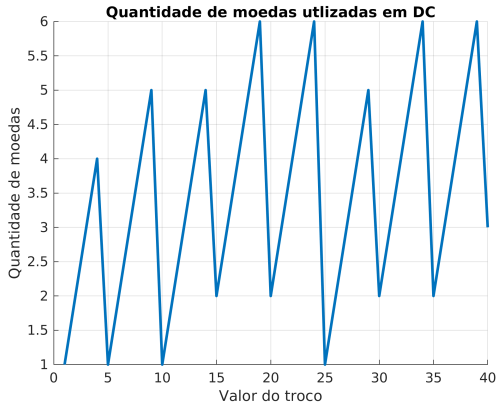


Figure 4: Quantidade de moedas necessárias para fornecer o respectivo troco no algoritmo de Divisão e Conquista

## 2.3 Programação Dinâmica

Veja que na implementação do algoritmo de Divisão e Conquista é possível que ocorra uma chamada para uma função que já foi anteriormente calculada. A ideia do algoritmo de Programação Dinâmica é tomar como a ideia do algoritmo de Divisão e Conquista, mas armazenar os valores de chamadas recursivas já calculados, fazendo uma lógica iterativa *bottom-up*. Assim:

```
void TrocoSolverPD::solve(const std::vector<unsigned int>
    &denom,unsigned int value, std::vector<unsigned int> &coins) {
    coins.resize(denom.size(),0);
    std::vector<unsigned int> quant(value + 1), last(value + 1);
    quant[0] = 0; last[0] = 0;
    unsigned int curquant, curlast, pos;
    for(unsigned int cents = 1; cents <= value; ++cents){
        curquant = cents;
        curlast = 0;
        pos = 0;
        for(std::vector<unsigned int>::const_iterator itr =
            denom.begin(); itr != denom.end(); ++itr, ++pos){
            if(*itr > cents) continue;
            if(quant[cents - *itr] + 1 < curquant){
                curquant = quant[cents - *itr] + 1;
                curlast = pos;
            }
        }
        quant[cents] = curquant;
        last[cents] = curlast;
    }
    for(unsigned int aux = value; aux > 0; coins[last[aux]]++, aux -=
        denom[last[aux]]);
}
```

Para analisar seu desempenho, traçaram-se os gráficos do tempo despendido em função do valor do troco e da quantidade de moedas utilizadas para se obter esse respectivo troco desejado.

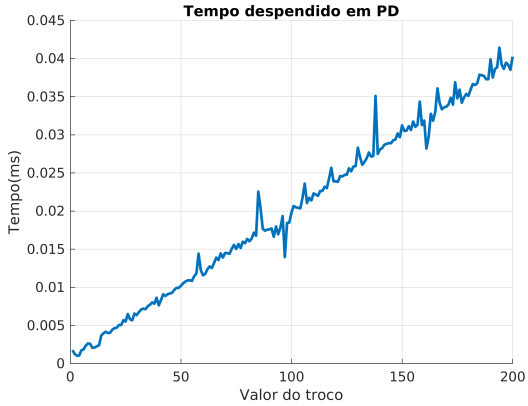


Figure 5: Tempo de execução do algoritmo de Programação Dinâmica

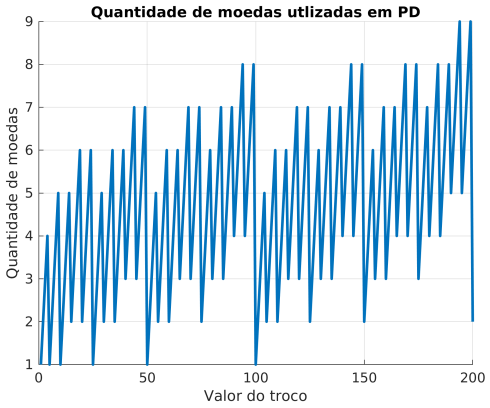


Figure 6: Quantitade de moedas necessárias para obter o respectivo troco no algoritmo de Programação Dinâmica

Nota-se, portanto, que o algoritmo de programação dinâmica apresenta tempo linear, diferentemente do de divisão e conquista cujo tempo é exponencial.

# 3 Análise dos algoritmos

Primeiramente, nota-se que o algoritmo de divisão e conquista apresenta solução ótima pois busca dentre todas as possibilidades anteriores a melhor possível para analisar se incrementa mais uma moeda ou não, a depender das denominações.

Consequentemente, o algoritmo de programação dinâmica deve ser ótimo pelos mesmos motivos, sempre analisa o ótimo até então e vai escalando numa lógica *bottom-up*.

Porém, nota-se que o algoritmo *Greedy* nem sempre obtém solução ótima, pois o número de moedas utilizadas para cada troco no *Greedy* não é sempre igual ao do *Programação Dinâmica*:

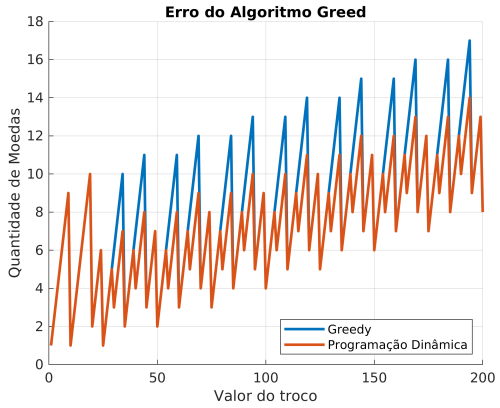


Figure 7: Comparação do número de moedas utilizadas com respeito ao respectivo troco no algoritmo *Greedy* e no algoritmo de Programação Dinâmica. A partir de certo troco, as quantidades começam a divergir

Por fim, o algoritmo de divisão e conquista é significativamente mais lento que os demais. Isso se dá pois, para um valor de troco suficientemente grande, cada chamada do algoritmo de divisão e conquista sempre fará mais $|denom|$ chamadas, isso resulta em uma complexidade de tempo exponencial.
Já o algoritmo de programação dinâmica, em cada iteração verifica $|denom|$ denominações, escalando um troco por um até o troco desejado, resultando em uma complexidade de tempo $\Theta(kn)$, onde $k$ é o número de denominações e $n$ é o valor do troco. Enquanto o algoritmo *Greedy* é apenas $\Theta(k)$, pois no pior caso analisa todas as denominações.
Vale ressaltar ainda que, por exemplo, o *Mergesort* é um algoritmo de divisão e conquista que apresenta complexidade de tempo ótima, e seria ineficiente adotar uma ideia de Programação Dinâmica armazenando sequências de vetores já ordenados em um algoritmo de ordenação, pois não é comum que a ordenação resolva o mesmo problema mais de uma vez. No entanto, para o problema em questão, a resolução de um mesmo problema é recorrente, pois há mais de uma escolha de denominações a serem subtraídas do valor do troco que gera um mesmo valor a ser analisado.
Logo, o pior desempenho temporal é mais por uma limitação do problema em específico do que é inerente ao paradigma em geral.