Laboratório 4: Paradigmas de Programação

Caio Graça Gomes

23 de Junho de 2020

1 Introdução

Neste laboratório, foram implementadas cinco algoritmos distintos que remetem a paradigmas da programação para resolução de dois problemas: Moedas de Troco e Soma de Subconjuntos. Esses algoritmos se baseiam em conceitos de busca gulosa (*Greedy* GR), divisão e conquista (DC), programação dinâmica (PD), sendo eles analisados pelas suas corretudes, alocação de memória e tempo despendido.

2 Algoritmos utilizados

2.1 Moedas de Troco

O problema de moedas de troco é: dado um vetor de denominações de moedas e um valor, qual a menor quantidade de moedas a serem utilizadas e quais seriam estas para que se obtenha o valor, OBS: repetições são permitidas. Isto é, dado $D,\ D := \{d_1, d_2, ..., d_n\},\ d_i \in \mathbb{N},\ 1 \leq i \leq n$ e um valor $v, v \in \mathbb{N}$. Uma solução S deve respeitar $S = \{s_1, s_2, ..., s_n\},\ s_j \in \mathbb{N},\ 1 \leq j \leq n$ e $s_1d_1 + s_2d_2 + ... + s_nd_n = v$. Qual o menor valor possível da função $f(S) = s_1 + s_2 + ... + s_n,\ f \in \mathbb{N}^n \to \mathbb{N}$ ao analisar todas as soluções possíveis e em que ponto do domínio $(s_1, s_2, ..., s_n)$ ocorre este mínimo?

2.1.1 *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:

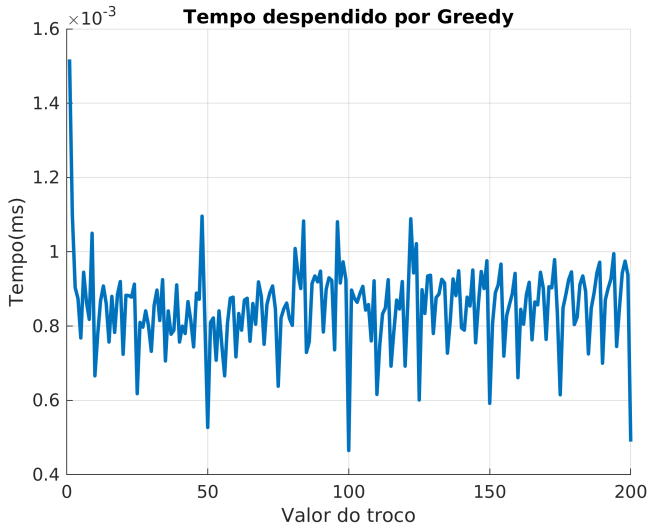

Figure 1: Tempo de execução do algoritmo Greedy

Nota-se que o tempo despendido pelo *Greedy* não apresenta muita correlação com o valor do troco, ainda que não seja $O(1)$. O número de moedas para cada valor de troco foi:

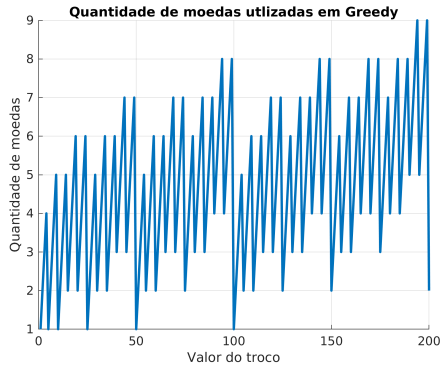

Figure 2: Quantidade de moedas necessárias para fornecer o respectivo troco no algoritmo Greedy

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

2.1.2 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:

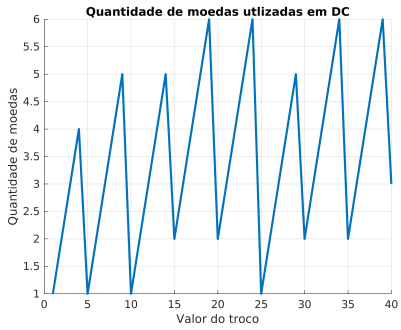

Figure 4: Quantidade de moedas necessárias para fornecer o respectivo troco no algoritmo de Divisão e Conquista

2.1.3 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.

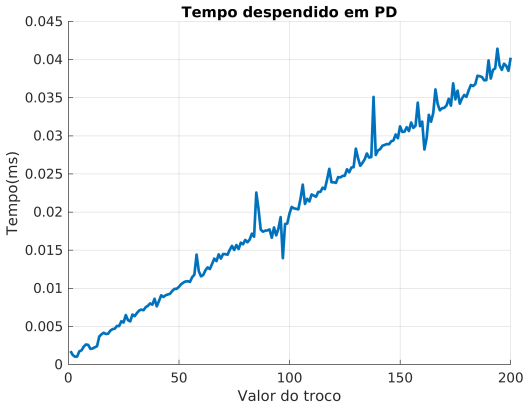

Figure 5: Tempo de execução do algoritmo de Programação Dinâmica

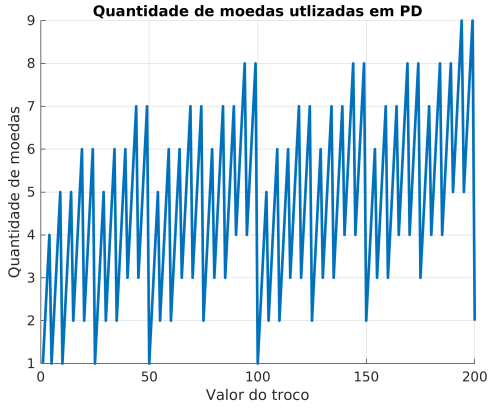

Figure 6: Quantitade de moedas necessárias para obter o respectivo troco no algoritmo de Programação Dinâmica

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

2.2 Soma de Subconjuntos

O problema de soma de subconjuntos a ser resolvido é: dado um vetor de números naturais e um valor, existe alguma combinação destes sem repetição cuja soma dê este valor? Isto é, dado um vetor W, $W := \{w_1, w_2, ..., w_n\}$, $w_i \in \mathbb{N}$, $1 \leq i \leq n$ e um valor v, $v \in \mathbb{N}$, existe $A = \{\alpha_1, \alpha_2, ..., \alpha_m\}$, $\alpha_j \in \mathbb{N}$, $\alpha_j \neq \alpha_k \, \forall j \neq k$, $1 \leq \alpha_j \leq n$, $1 \leq j, k \leq m$ tal que $w_{\alpha_1} + w_{\alpha_2} + ... + w_{\alpha_m} = v$?

Para avaliar o desempenho dos algoritmos a serem implementados para resolução deste problema, serão usados algumas instâncias de testes, serão elas:

1. $P(E)$: w_j uniformemente aleatório em $[1, 10^E]$, $v = n\frac{10^E}{4}$, $n : 4, 8, 12, 16, 20, 24, 28$;
2. $EVEN/ODD$: w_j par e uniformemente aleatório em $[1, 1000]$, $v = 2\lfloor\frac{1000n}{8}\rfloor + 1$, $n = 4, 8, 12, 16, 20, 24, 28$;
3. $AVIS$: $w_j = n(n+1) + j$, $v = \lfloor\frac{n-1}{2}\rfloor n(n+1) + \binom{n}{2}$, $n : 4, 8, 12, 16$;
4. $TODD$: $w_j = 2^{k+n+1} + 2^{k+j} + 1$, com $k = \lfloor\log_2 n\rfloor$, $v = (n+1)2^{k+n} - 2^k + \lfloor\frac{n}{2}\rfloor$, $n : 4, 8, 12, 16$.

2.2.1 Programação Dinâmica

A ideia básica para o algoritmo de programação dinâmica foi: Analisa se é possível obter o valor 0 com nenhum elemento do vetor de pesos, depois com o primeiro elemento, depois com o primeiro e segundo, depois primeiro segundo e terceiro, e assim sucessivamente até com todos os elementos (0 é caso trivial, sempre verdadeiro). Então, verifica-se se é possível obter o valor 1 sem nenhum elemento, só com o primeiro elemento, depois o primeiro e o segundo, e assim sucessivamente até usar todos. Repete-se esse processo até que verifica se é possível obter o valor desejado com todos os elementos do conjunto de pesos, seguindo uma lógica *bottom-up*. Para verificar se com os n primeiros elementos é possível obter a soma v:

1. Se $v = 0$, então é possível;
2. Se $v \neq 0$ e $n = 0$, então é impossível;
3. Se $v \neq 0$ e $n \neq 0$, então é possível se e somente se é possível obter v com $n-1$ ou é possível obter $v - w_n$ com $n-1$.

Para recuperar quais elementos foram utilizados para obter a soma, verifica se foi possível obter v com $n-1$, se sim então não utilizou-se o m-ésimo elemento e analisa se foi possível obter v com $n-2$, caso contrário se utilizou e analisa se foi possível obter $v - w_n$ com $n-2$. Note que as análises se chamam novamente em um processo recursivo ou iterativo, o processo é repetido até analisar se o primeiro elemento foi necessário.

Assim, a implementação em $C++$ foi:

```
bool SSPSolverPD::solve(const std::vector<long> &input, long value,
    std::vector< char> &output){
   output.resize(input.size());
   std::vector<std::vector<bool>> backpacks(input.size()+1);
   for(std::vector<std::vector<bool>>::iterator itr =
       backpacks.begin(); itr != backpacks.end(); ++itr){
      (*itr).resize(value+1, false);
      (*itr)[0] = true;
```

```
    }
    for(long line = 1; line <= input.size(); ++line){
        for(long column = 1; column <= value; ++column){
            if(input[line-1] > column) backpacks[line][column] =
                backpacks[line-1][column];
            else backpacks[line][column] = backpacks[line-1][column] ||
                backpacks[line-1][column - input[line-1]];
        }
    }
    if(backpacks[input.size()][value]){
        long column = value;
        long line = input.size();
        while(column != 0){
            if(backpacks[line-1][column]){
                output[line-1] = false;
                line--;
            }
            else{
                output[line-1] = true;
                column -= input[line-1];
                line--;
            }
        }
    }
    return backpacks[input.size()][value];
}
```

Quanto testado sobre as diferentes instâncias de testes o algoritmo de programação dinâmica apresentou o seguinte comportamento:

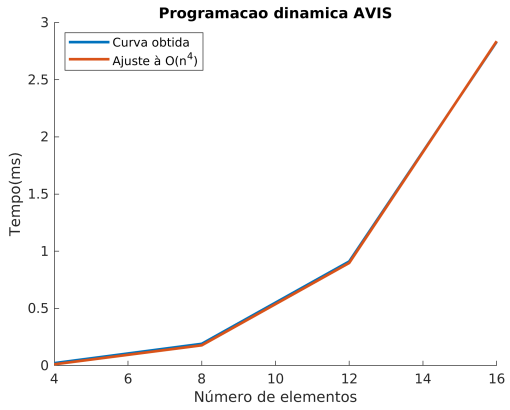

Figure 7: Tempo despendido por algoritmo de programação dinâmica ao elevar o tamanho da entrada na instância de teste AVIS

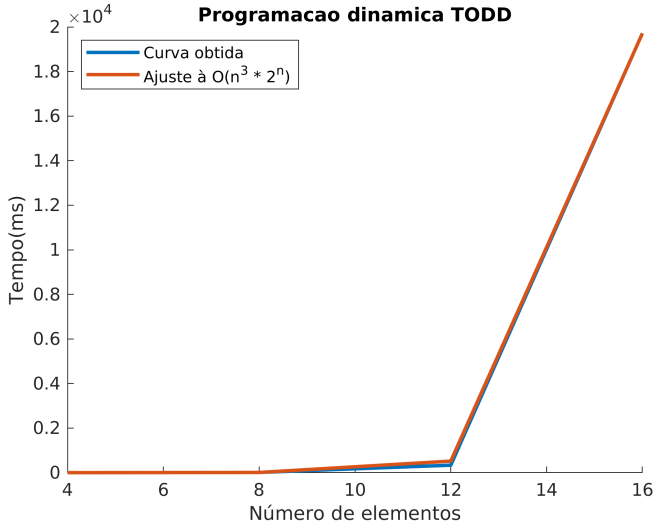

Figure 8: Tempo despendido por algoritmo de programação dinâmica ao elevar o tamanho da entrada na instância de teste TODD

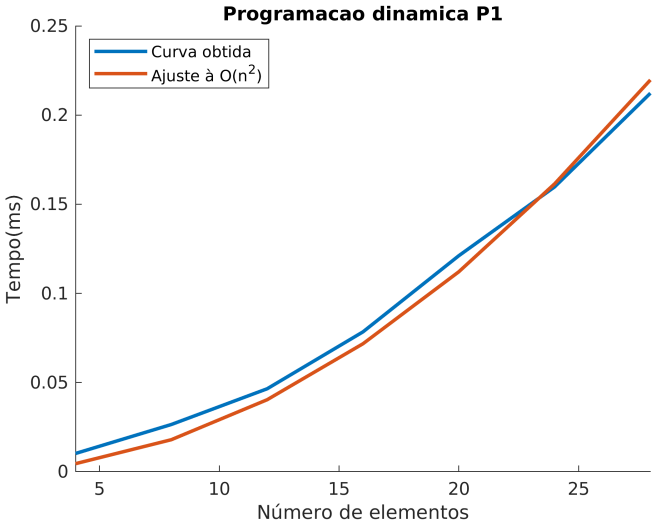

Figure 9: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P1

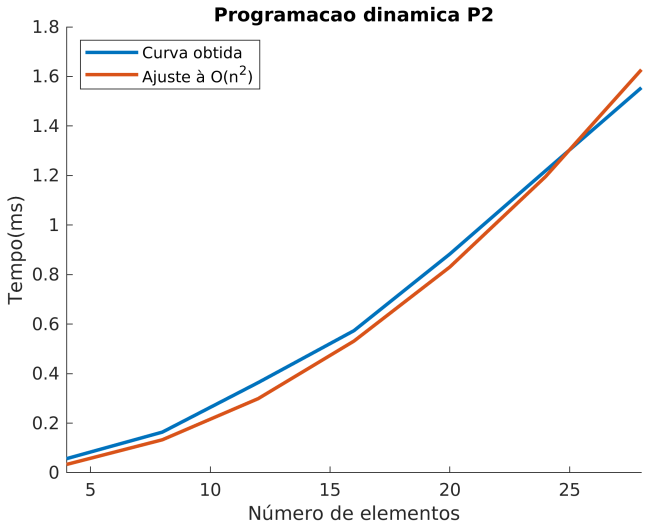

Figure 10: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P2

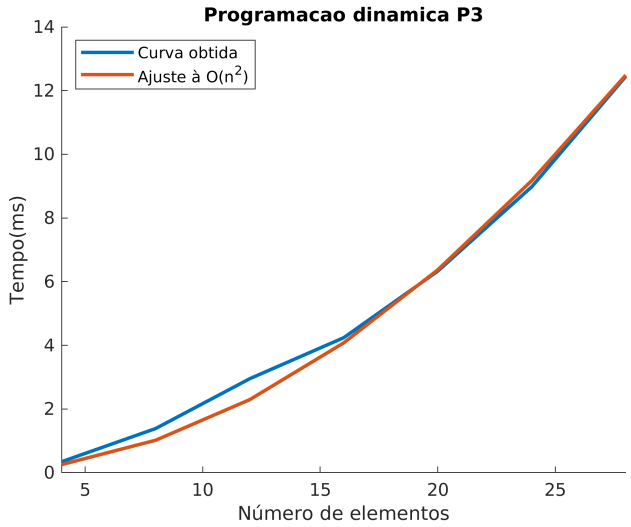

Figure 11: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P3

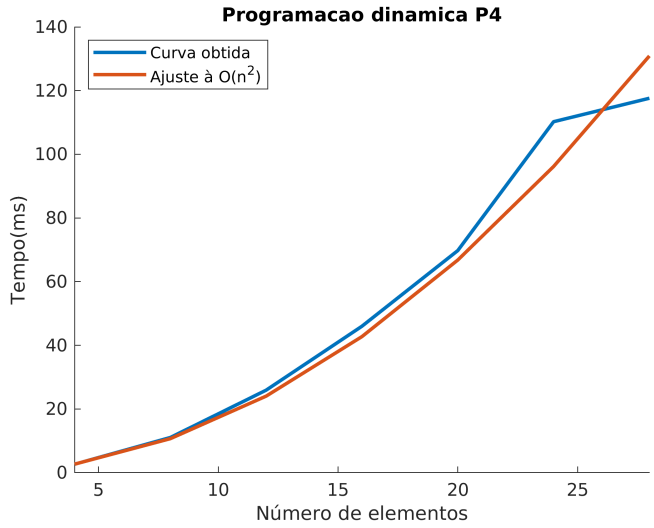

Figure 12: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P4

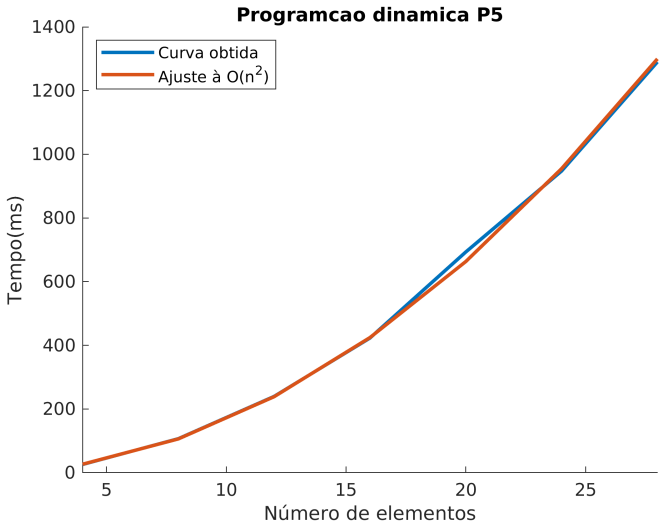

Figure 13: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P5

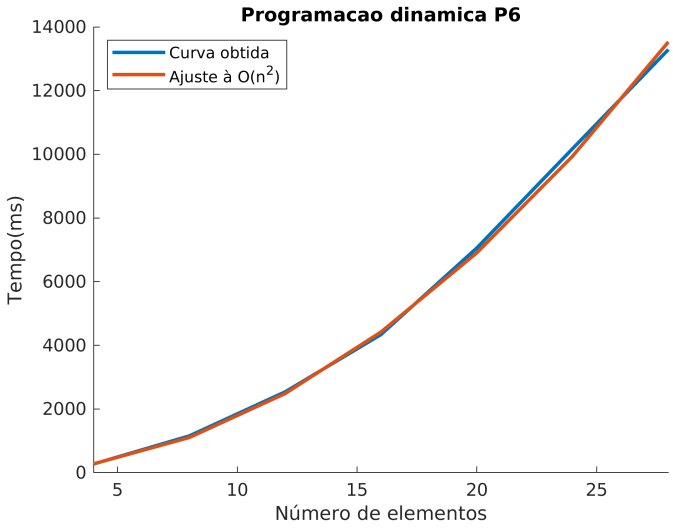

Figure 14: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P6

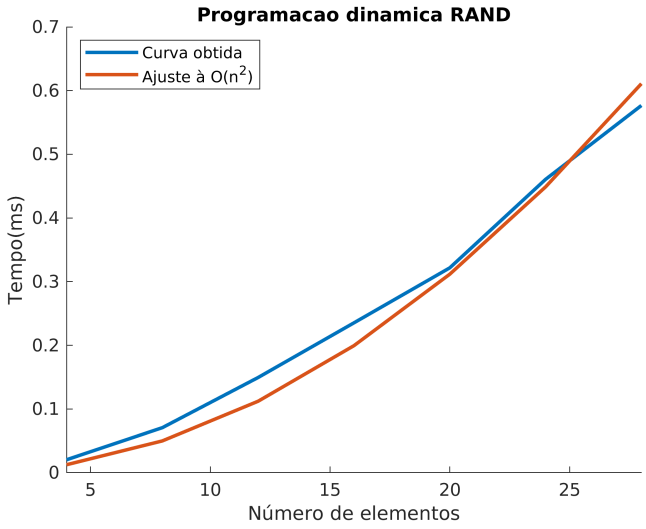

Figure 15: Tempo despendido por algoritmo de programação dinâmica (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste RAND

Considerando uma entrada de tamanho n e um valor v, o algoritmo utilizado preenche uma matriz *booleana* $(n+1)x(v+1)$, sendo cada elemento preenchido

em $O(1)$ tempo. Assim, o algoritmo deve ser $O(nv)$ tempo e espaço. fazendo a análise para cada instância:

1. Para as instâncias de teste PX, tem-se $v = n\frac{10^X}{4}$, portanto o algoritmo despende $O(\frac{10^X}{4}n^2)tempo$. O crescimento quadrático é condizente com os gráficos anteriormente mostrados;
2. Para a instância de teste $EVOD$, tem-se $v = n\frac{10^3}{4}$, sendo uma análise análoga a de PX;
3. Para a instância de teste $AVIS$, tem-se $v = \lfloor\frac{n-1}{2}\rfloor n(n+1) + \binom{n}{2}$, portanto o algoritmo despende $O(n^4)$ tempo;
4. Para a instância de teste $TODD$, tem-se $v = (n+1)2^{k+n} - 2^k + \lfloor\frac{n}{2}\rfloor$, com $k = \lfloor\log_2 n\rfloor$. Portanto o algoritmo é $O(n^3 2^n)$.

2.2.2 *Meet in the Middle*

A ideia básica da implementação *Meet in the Middle* é dividir o vetor de pesos no meio e para cada uma das metades gerar um novo vetor com todas as somas possíveis de elementos desta metade. Após isso, ordenar o vetor de todas as somas possíveis correspondente à primeira metade e, para cada soma no outro vetor de somas, procurar a soma necessária restante para completar o valor desejado no primeiro vetor de somas já ordenado. Se encontar alguma vez é possível, caso contrário é impossível.

Para recuperar os elementos utilizados, cada elemento dos vetores de somas armazena o valor da soma e os números utilizados para se obter aquela soma, por meio de um número inteiro representado em binário. Se o dígito i for 0, então não se utilizou este peso, e se for 1 utilizou. Assim, sabendo das duas somas que somadas dão o valor desejado é possível obter quais elementos foram utilizados.

Assim, a implementação em $C++$ foi:

```
bool func(std::pair<long, long> &a, std::pair<long, long> &b){
    return a.first < b.first;
}

long binary_search(long value, std::vector<std::pair<long, long>> &v,
     long begin, long end){
    if(begin>end) return -1;
    long mid = (begin+end) >> 1;
    if(v[mid].first == value) return v[mid].second;
    else if(v[mid].first < value) return binary_search(value, v, mid +
        1, end);
    else return binary_search(value, v, begin, mid-1);
}

bool SSPSolverBranchBound::solve(const std::vector<long> &input, long
     value, std::vector<char> &output) {
```

```
    output.resize(input.size());
    std::vector<long> v1, v2;
    long l = (input.size()%2 == 0) ? input.size() >> 1 : (input.size()
        >> 1) + 1;
    v1.resize(l); v2.resize(input.size()-l);
    std::copy(input.begin(), input.begin() + l, v1.begin());
    std::copy(input.begin()+l, input.end(), v2.begin());
    std::vector<std::pair<long, long>> s1(1<<v1.size(),
        std::make_pair(0, 0)), s2(1<<v2.size(), std::make_pair(0, 0));
    long aux, aux2;
    for(long i = 0; i < s1.size(); ++i){
        aux = i;
        s1[i].second = i;
        for(long j = 0; j < v1.size() && aux > 0; ++j, aux = aux >> 1){
            s1[i].first += (aux%2) * v1[j];
        }
    }
    for(long i = 0; i < s2.size(); ++i){
        aux = i;
        s2[i].second = i;
        for(long j = 0; j < v2.size() && aux > 0; ++j, aux = aux >> 1){
            s2[i].first += (aux%2) * v2[j];
        }
    }
    std::sort(s1.begin(), s1.end(), func);
    for(long i = 0; i < s2.size(); ++i){
        aux = binary_search(value - s2[i].first, s1, 0, s1.size() - 1);
        if(aux != -1){
            for(long j = 0; j < v1.size() && aux > 0; ++j, aux = aux >>
                1){
                output[j] = (aux%2) ? true : false;
            }
            aux2 = i;
            for(long j = v1.size(); j < input.size() && aux2 > 0; ++j,
                aux2 = aux2 >> 1){
                output[j] = (aux2%2) ? true : false;
            }
            return true;
        }
    }
    return false;
}
```

Quanto testado sobre as diferentes instâncias de testes o algoritmo *Meet in the Middle* apresentou o seguinte comportamento:

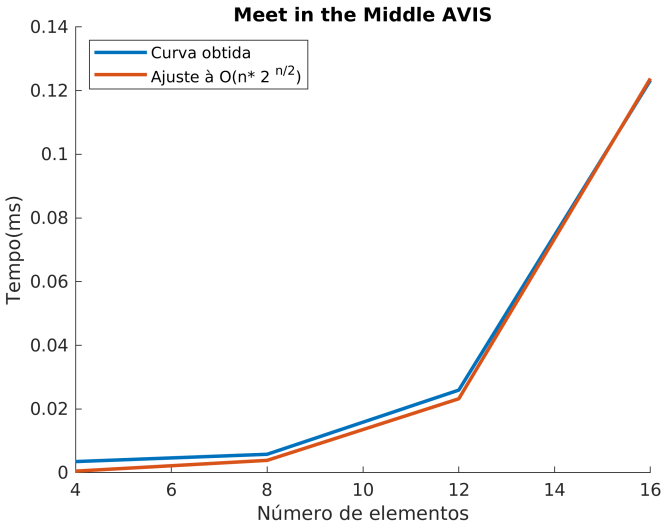

Figure 16: Tempo despendido por algoritmo *Meet in the Middle* ao elevar o tamanho da entrada na instância de teste AVIS

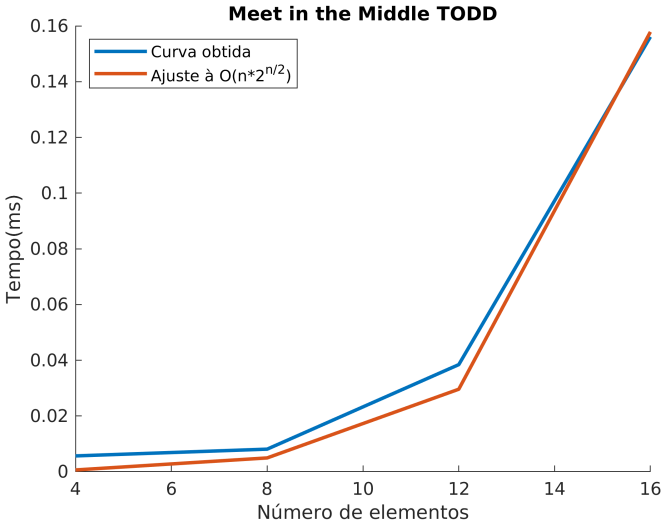

Figure 17: Tempo despendido por algoritmo *Meet in the Middle* ao elevar o tamanho da entrada na instância de teste TODD

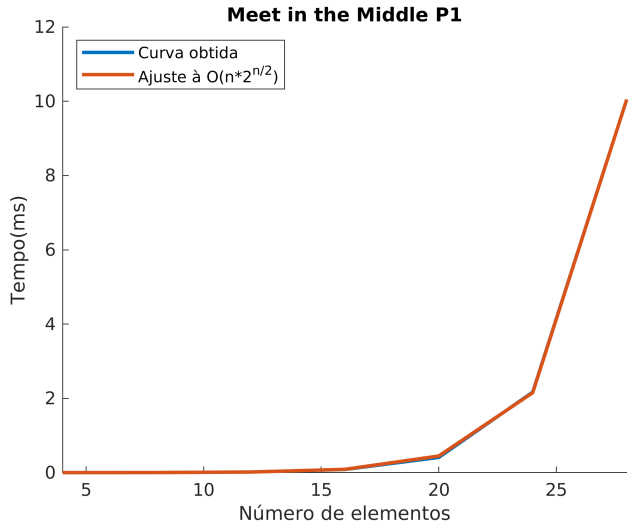

Figure 18: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P1

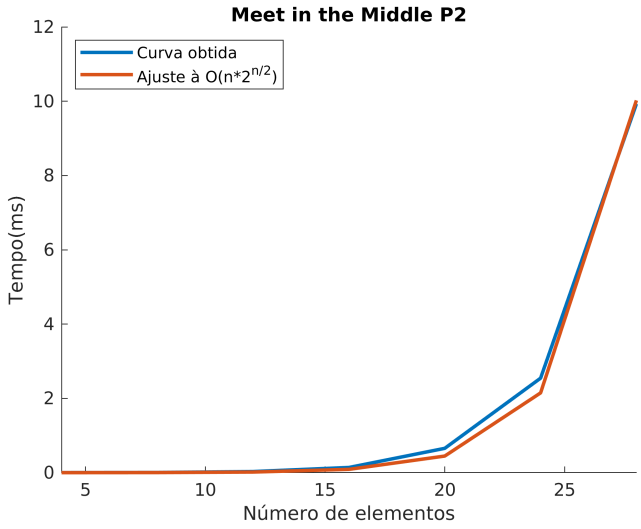

Figure 19: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P2

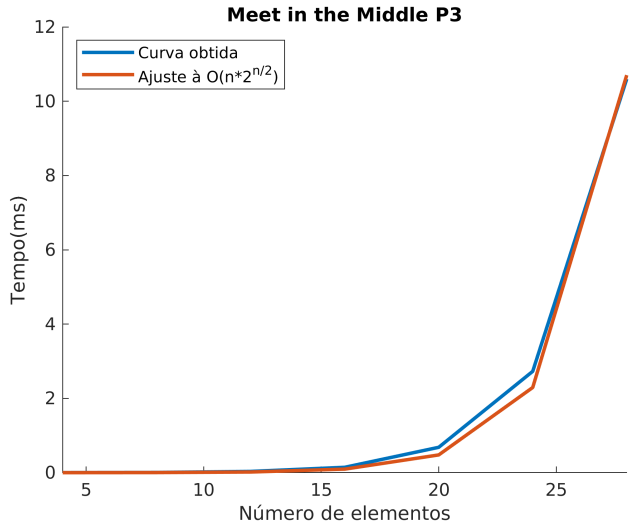

Figure 20: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P3

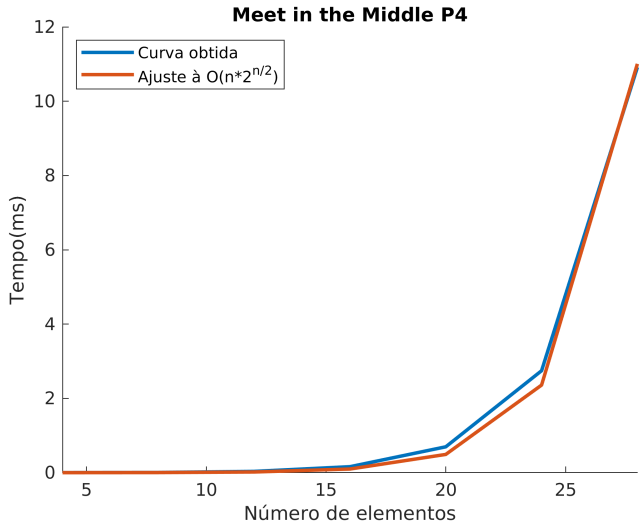

Figure 21: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P4

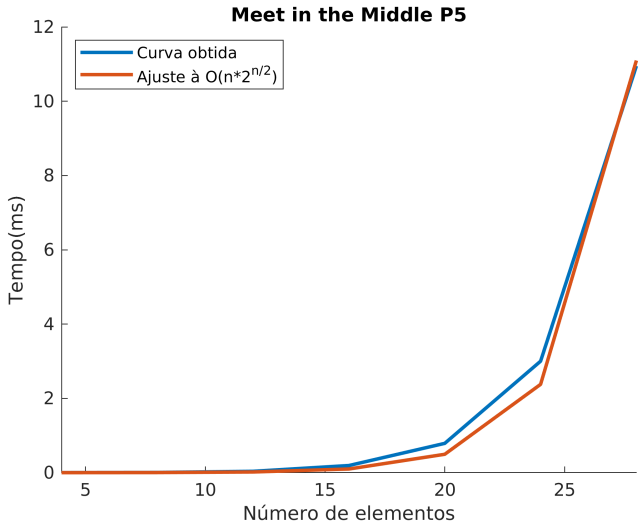

Figure 22: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P5

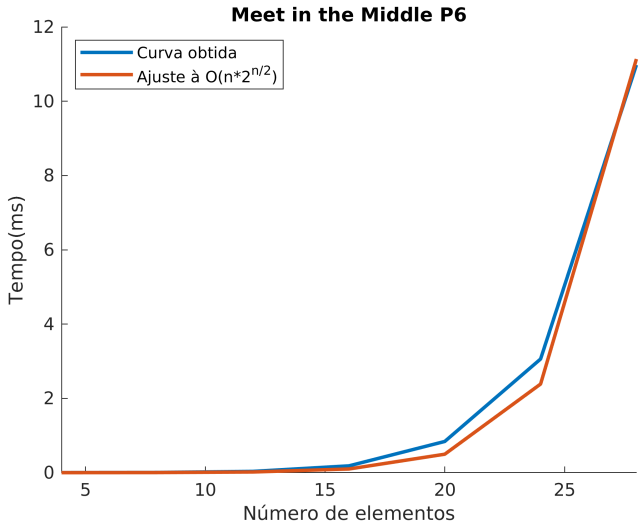

Figure 23: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste P6

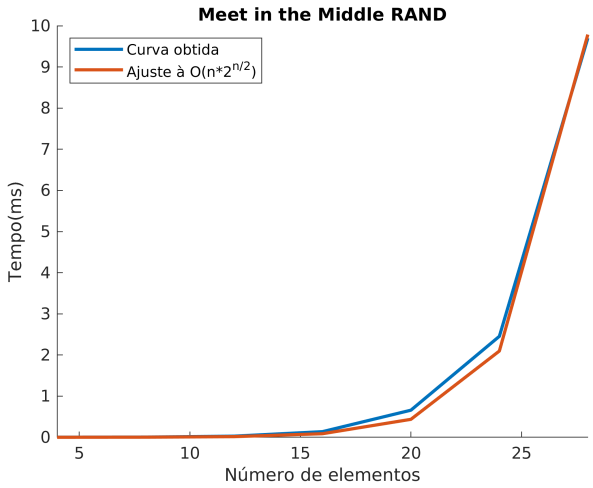

Figure 24: Tempo despendido por algoritmo *Meet in the Middle* (média de 5 repetições) ao elevar o tamanho da entrada na instância de teste RAND

Considerando uma entrada de tamanho n e um valor v, o algoritmo utilizado computa todas as somas possíveis de vetores de tamanho $O(\frac{n}{2})$ em $O(2^{\frac{n}{2}})$ tempo e espaço, ordena um deles em $O(\frac{n}{2}2^{\frac{n}{2}})$ tempo e percorre um deles em $O(2^{\frac{n}{2}})$ tempo. Assim, o algoritmo deve ser $O(n\sqrt{2}^n)$ tempo, independentemente da instância de teste.

3 Comparação dos algoritmos

3.1 Moedas de Troco

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*:

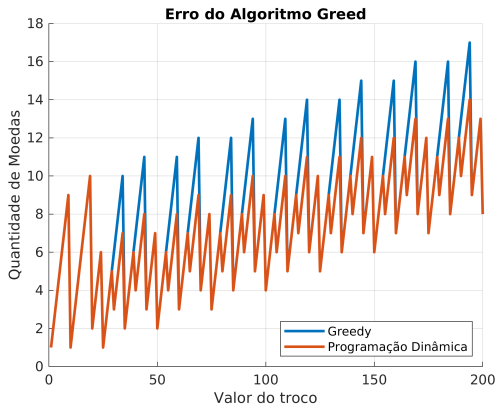

Figure 25: Comparação do número de moedas utilizadas com respeito ao respectivo troco no algoritmo *Greedy* e no algoritmo de Programação Dinâmica. A partir de certo troco, as quantidades começam a divergir

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.

3.2 Soma de Subconjuntos

Note que o desempenho do algoritmo *Meet in the Middle* foi superior ao de Programação Dinâmica nos testes realizados, apesar de possuir uma complexidade de tempo $O(n\sqrt{2}^n)$, contra uma complexidade de tempo $O(nv)$. Isso se dá pois os tamanhos das entradas foram pequenos, para valores n maiores o algoritmo de programação dinâmica teria desempenho superior (com exceção na instância de teste $TODD$, na qual a complexidade de tempo é $O(n^3 2^n)$).

Nesse sentido, vale ressaltar que o algoritmo de programação dinâmica é pseudo-polinomial, isto é, apesar de ser linear, a princípio, em n ($O(nv)$), o valor v pode depender exponencialmente em n, conforme a instância de teste $TODD$.
Por fim, o algoritmo *Meet in the Middle* é mais indicado para se usar para valores de n pequenos, menores que 40 ou 50 e também quando o valor v for muito elevado. Já o algoritmo de programação dinâmica é mais indicado para lidar com tamanhos n maiores que 40 ou 50 e o valor v não for muito elevado.