UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ – UNIOESTE

CIÊNCIA DA COMPUTAÇÃO

Guilherme Rodrigues Sganderla

Renan Giuseppe Bertolazo

TRABALHO DE PROJETO E ANALISE DE ALGORITMOS

RELATORIO

Foz do Iguaçu, PR

2019

Analise de complexidade de tempo e espaço de cada algoritmo

Propriedade de Somatórios usado na análise das complexidades:

$$\sum_{i=m}^{n}(1) = n + 1 - m$$

$$\sum_{i=m}^{n} xi + yi = \sum_{i=m}^{n} xi + \sum_{i=m}^{n} yi$$

Algoritmo de multiplicação de matrizes de $O(n^3)$

```c
void mult_matriz(int **A, int **B, int **C, int n){

    for(int i = 0; i < n; i++){

        C[i] = (int*)malloc(sizeof(int)*(unsigned long long)n);
        for(int j = 0; j < n; j++){

            C[i][j] = 0;

            for(int k = 0; k < n; k++){

                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Figura 1. Função de multiplicação de matrizes.[1]

Para esse algoritmo, a função da complexidade fica:

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

$$(II)\ T(n) = \sum_{i=0}^{n}(1 + \sum_{j=0}^{n}(1 + n + 1) \to \sum_{i=0}^{n}(1 + \sum_{j=0}^{n}(n + 2))$$

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

$$(IV)\ T(n) = \sum_{i=0}^{n}(1 + (2n + 2 + n^2 + n)) \to \sum_{i=0}^{n}(3 + (3n + n^2))$$

$$(V)\ T(n) = \sum_{i=0}^{n} 3 + 3 * \sum_{i=0}^{n} n + \sum_{i=0}^{n} n^2 = 3 * (n + 1) + 3 * n * (n + 1) + n * (n^2 + 1)$$

$$(VI)\ T(n) = 3n + 3 + 3n^2 + 3n + n^3 + n \to n^3 + 3n^2 + 6n$$

$$\boldsymbol{(VI)\ T(n) = O(n^3)}$$

Esse algoritmo tem uma complexidade de tempo de O(n³) e para espaço de memória se tem 3*n² para o tamanho do tipo de dados, já que são 3 matrizes de ordem n por n.

Algoritmo Strassen para multiplicação de matrizes

Como o algoritmo de Strassen usa duas outras funções auxiliares, sendo elas, soma e sub, que são respectivamente uma soma de matrizes (C = A+B) e uma subtração de matrizes (C = A-B) precisamos descobrir a complexidade dessas funções, então requer uma análise de cada uma.

Analise da função de soma de matrizes

```
void soma(int **A, int **B, int **C, int n){

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

Figura 2. Função de soma de matrizes.[1]

Para esse algoritmo, se tem uma instrução, que resulta em 1 na complexidade, um for de 0 a n executando essa instrução e outro for executando esse for de 0 a n, partindo disso a analise fica:

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

$$(II)\ T(n) = \sum_{i=0}^{n} 1 + \sum_{i=0}^{n} n \rightarrow (n+1) + \big(n * (n+1)\big)$$

$$(III)\ T(n) = n + 1 + n^2 + n \rightarrow n^2 + 2n + 1$$

$$\boldsymbol{(IV)\ T(n) = O(n^2)}$$

Analise da função de subtração de matrizes

```
void sub(int **A, int **B, int **C, int n){

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}
```

Figura 3. Função de subtração de matrizes.[1]

Para o algoritmo de subtração de matrizes a complexidade é O(n²) também, por semelhanças com o algoritmo de soma.

Analise da função de multiplicação de matrizes do algoritmo Strassen:

```c
void mult_strassen(int **A, int **B, int **C, int n){

    if(n <= LIMITE){

        C[0][0] = A[0][0] * B[0][0];

    }
    else{

        int novo_n = n/2, i, j;

        int **a11 = alocaMat(novo_n), **a12 = alocaMat(novo_n);
        int **a21 = alocaMat(novo_n), **a22 = alocaMat(novo_n);
        int **b11 = alocaMat(novo_n), **b12 = alocaMat(novo_n);
        int **b21 = alocaMat(novo_n), **b22 = alocaMat(novo_n);
        int **c11 = alocaMat(novo_n), **c12 = alocaMat(novo_n);
        int **c21 = alocaMat(novo_n), **c22 = alocaMat(novo_n);
        int **m1 = alocaMat(novo_n), **m2 = alocaMat(novo_n);
        int **m3 = alocaMat(novo_n), **m4 = alocaMat(novo_n);
        int **m5 = alocaMat(novo_n), **m6 = alocaMat(novo_n);
        int **m7 = alocaMat(novo_n), **A_res = alocaMat(novo_n);
        int **B_res = alocaMat(novo_n);

        for(i = 0; i < novo_n; i++){

            for (j = 0; j < novo_n; j++){

                a11[i][j] = A[i][j];
                a12[i][j] = A[i][j + novo_n];
                a21[i][j] = A[i + novo_n][j];
                a22[i][j] = A[i + novo_n][j + novo_n];

                b11[i][j] = B[i][j];
                b12[i][j] = B[i][j + novo_n];
                b21[i][j] = B[i + novo_n][j];
                b22[i][j] = B[i + novo_n][j + novo_n];
            }
        }

        soma(a11, a22, A_res, novo_n);
        soma(b11, b22, B_res, novo_n);
        mult_strassen(A_res, B_res, m1, novo_n);

        soma(a21, a22, A_res, novo_n);
        mult_strassen(A_res, b11, m2, novo_n);

        sub(b12, b22, B_res, novo_n);
        mult_strassen(a11, B_res, m3, novo_n);

        sub(b21, b11, B_res, novo_n);
        mult_strassen(a22, B_res, m4, novo_n);

        soma(a11, a12, A_res, novo_n);
        mult_strassen(A_res, b22, m5, novo_n);
```

Figura 4. Parte da função do algoritmo de Strassen.[1]

```
        sub(a21, a11, A_res, novo_n);
        soma(b11, b12, B_res, novo_n);
        mult_strassen(A_res, B_res, m6, novo_n);

        sub(a12, a22, A_res, novo_n);
        soma(b21, b22, B_res, novo_n);
        mult_strassen(A_res, B_res, m7, novo_n);

        soma(m3, m5, c12, novo_n);
        soma(m2, m4, c21, novo_n);

        soma(m1, m4, A_res, novo_n);
        soma(A_res, m7, B_res, novo_n);;
        sub(B_res, m5, c11, novo_n);

        soma(m1, m3, A_res, novo_n);
        soma(A_res, m6, B_res, novo_n);
        sub(B_res, m2, c22, novo_n);

        for(i = 0; i < novo_n; i++){

            for(j = 0; j < novo_n; j++){

                C[i][j] = c11[i][j];
                C[i][j + novo_n] = c12[i][j];
                C[i + novo_n][j] = c21[i][j];
                C[i + novo_n][j + novo_n] = c22[i][j];
            }
        }

        a11 = desalocaMat(a11,novo_n);
        a12 = desalocaMat(a12,novo_n);
        a21 = desalocaMat(a21,novo_n);
        a22 = desalocaMat(a22,novo_n);
        b11 = desalocaMat(b11,novo_n);
        b12 = desalocaMat(b12,novo_n);
        b21 = desalocaMat(b21,novo_n);
        b22 = desalocaMat(b22,novo_n);
        c11 = desalocaMat(c11,novo_n);
        c12 = desalocaMat(c12,novo_n);
        c21 = desalocaMat(c21,novo_n);
        c22 = desalocaMat(c22,novo_n);
        m1 = desalocaMat(m1,novo_n);
        m2 = desalocaMat(m2,novo_n);
        m3 = desalocaMat(m3,novo_n);
        m4 = desalocaMat(m4,novo_n);
        m5 = desalocaMat(m5,novo_n);
        m6 = desalocaMat(m6,novo_n);
        m7 = desalocaMat(m7,novo_n);
        A_res = desalocaMat(A_res,novo_n);
        B_res = desalocaMat(B_res,novo_n);
    }
}
```

Figura 5. Parte final da função do algoritmo de Strassen.[1]

Nesse algoritmo, temos 2 funções cujo complexidade são $O(n^2)$, que são os dois pares de for para atribuir o valor das matrizes, temos 7 chamadas recursivas para a função de multiplicação, 12 chamadas para a função de soma de matrizes, 6 chamadas para a função de subtração de matrizes, total das atribuições e 21 chamadas de função para alocação e desalocação, estas que são O(n^2) também, o conjunto disso da:

$$T(n) = \begin{cases} 1, & x = 1 \\ 7 * T\left(\frac{n}{2}\right) + (12 + 6 + 42) * O(n^2), & x \geq 2 \end{cases}$$

Se ignorarmos as 60 chamadas de O(n^2) e colocarmos como n^2, o cálculo da complexidade será:

$$T(n) = \begin{cases} 1, & x = 1 \\ 7 * T\left(\frac{n}{2}\right) + n^2, & x \geq 2 \end{cases}$$

Usando o método mestre, temos:

$$a = 7, b = 2, f(n) = n^2$$

$$\log_2 7 = 2{,}807$$

Pela regra do método mestre:

$$f(n) \in O\left(n^{\log_2 7 - \varepsilon}\right) \; onde \; \varepsilon = 0{,}807$$

Portanto $\boldsymbol{T(n)} \in \boldsymbol{\theta(n^{2,807})}$

O algoritmo de Strassen tem complexidade de tempo de O($n^{2,807}$) e uma complexidade de espaço de memória de: 3 matrizes NxN, 3 matrizes $2^n x 2^n$, onde 2^n é a próxima potencia de 2 próxima ao N das 3 matrizes anteriores e para cada chamada recursiva da função são alocadas 21 matrizes de tamanho n/2.

Analise da função do problema da mochila usando a técnica de Programacao Dinâmica.

```c
int prog_dinamica(Objeto *ob, int W, int n){

    int i, j;

    for(i = 1; i < n+1; i++){

        for(j = 0; j <= W; j++){

            if(ob[i-1].w <= j && ((s[i-1][j - ob[i-1].w] + ob[i-1].c) > s[i-1][j])){

                s[i][j] = s[i-1][j - ob[i-1].w] + ob[i-1].c;

                aux_x[i-1][j] = 1;
            }
            else{

                s[i][j] = s[i-1][j];

                aux_x[i-1][j] = 0;
            }
        }
    }

    j = W;

    for(i = n-1; i >= 0; i--){

        if(aux_x[i][j] == 1){

            x[i] = 1;

            peso_final += ob[i].w;

            j -= ob[i].w;
        }
        else
            x[i] = 0;
    }

    return s[n][W];
}
```

Figura 6. Função do problema da mochila usando Programação Dinâmica.[1]

Para esse algoritmo se tem pelo menos 11 instruções de complexidade constante 1, um for de 0 a W com execução de pelo menos 2 instruções, um for de 1 a n+1 que executa o for anterior, e um for de n-1 a 0, que executa 3 ou 1 instruções. Com base nisso:

$$(I)\ T(n) = \sum_{i=1}^{n+1}\left(\sum_{j=0}^{W}(2)\right) + 1 \sum_{i=n-1}^{0}(4) + 1 = \sum_{i=1}^{n+1}(2 * (W+1)) + (0+1+n+1) + \ 2$$

$$(II)\ T(n) = \sum_{i=1}^{n+1}(2W+2) + (2+n) + 2 \rightarrow \sum_{i=1}^{n+1} 2W + \sum_{i=1}^{n+1} 2 + 4 + n$$

$$(III)\ T(n) = 2W * \sum_{i=1}^{n+1} 1 + 2 * (n+1) + 4 + n \ \rightarrow 2W * (n+1) + 2n + 6 + n$$

$$(IV)\ T(n) = 2Wn + 2W + 3n + 6$$

Portanto a complexidade de tempo desse algoritmo é **O(n*W)**, para a complexidade de espaço temos: 1 matriz de ordem n+1 por W+1, 1 matriz de ordem n por W, um vetor de n tamanho e 2 variáveis para guardar o peso e valor total. A complexidade de espaço seria O((n+1)*(W+1)), sendo n+1 e W+1 a complexidade maior.

Analise da função do problema da mochila usando a técnica de Backtracking.

Para o algoritmo com uso de Backtracking, foi necessário usar duas outras funções, sendo essas a função pesos, que calcula o peso total da possível solução atual, a função atualiza, que faz com que o vetor solução receba a possível solução atual, para calcular a complexidade total do algoritmo do problema da mochila, precisa calcular a complexidade dessas duas funções. Tendo isso:

Analise da função pesos

```
int pesos(int n, int *temp, Objeto* ob, int W){

    int resultado = 0;

    for(int i = 0; i < n; i++){

        if(resultado < W)
            resultado += temp[i] * ob[i].w;
        else
            return 0;
    }

    return 1;
}
```

Figura 7. Função auxiliar da função do problema da mochila usando Backtracking.[1]

Para esse algoritmo, temos 4 instruções de constante 1 e um for de 0 a n, o cálculo da complexidade fica:

$$(I)\ T(n) = \sum_{i=0}^{n} (1) + 2 \ \rightarrow \ (n + 1 - 0) + 2$$

$$(II)\ \boldsymbol{T(n) = n + 3}$$

Portanto, a função pesos tem uma complexidade de tempo de **O(n)** e uma complexidade de espaço de 1 inteiro, que é a variavel *resultado*.

Analise da função atualiza

```c
void atualiza(Objeto *ob, int n, int *temp){

    int peso_total = 0;
    int valor_total = 0;

    for(int i = 0; i < n; i++){

        if(temp[i] == 1){

            peso_total += ob[i].w;
            valor_total += ob[i].c;
        }
    }

    if(valor_total > valor_final){

        for(int i = 0; i < n; i++){

            x[i] = temp[i];
        }

        valor_final = valor_total;
        peso_final = peso_total;
    }
}
```

Figura 8. Função auxiliar atualiza.[1]

Nessa função temos, 7 instruções de complexidade 1, 1 for de 0 a n e outro for de 0 a n, com isso em mente temos:

$$(I)\ T(n) = 2 + \sum_{i=0}^{n}(2) + \sum_{i=0}^{n} 1 + 2 \ \rightarrow\ \big(2 * (n+1)\big) + (n+1) + 4$$

$$(II)\ T(n) = 2n + 2 + n + 1 + 4 \ = \ 3n + 7$$

Portanto, essa função tem complexidade de **O(n)** e complexidade de espaço de 2 variáveis**.**

Analise da função com Backtracking

```
void backtracking(Objeto *ob, int W, int i, int n, int *tempX){

    tempX[i] = -1;

    while(tempX[i] < 1){
        tempX[i] += 1;

        if((pesos(n, tempX, ob, W)) && i == n-1){

            atualiza(ob, n, tempX);
        }

        else if(i < n-1){

            backtracking(ob, W, i+1, n, tempX);
        }
    }

}
```

Figura 9. Função principal do problema da mochila com Backtracking.[1]

Para essa função, temos 4 instruções de complexidade 1, um while de -1 a 1 e uma chamada recursiva da função, com isso:

$$(I)\ T(n) = \textstyle\sum_{-1}^{1}(1 + \mathrm{O(n)} + \mathrm{O(n)} + \mathrm{T(i+1)}) \rightarrow \textstyle\sum_{-1}^{1}(1 + 2n + T(i+1))$$

$$(II)\ T(n) = \sum_{-1}^{1}\big(1 + n + T(n+1)\big) \rightarrow \sum_{-1}^{1} 1 + \sum_{-1}^{1} 2n + \sum_{-1}^{1} T(i+1)$$

$$(III)\ T(n) = (1 * (1+1) + \big(n * (1+1)\big) + (1+1) * T(i+1)$$

$$(IV)\ T(n) = 2 + 2n + 2 * T(i+1)$$

$$(V)\ T(n) = \begin{cases} 1, & i = n-1 \\ 2 * T(i+1) + 2n + 2, & i < n-1 \end{cases}$$

Pelo método da iteração, temos:

$$T(n) = 2T(i+1) + 2n + 2$$

$$T(n) = 2T(i+1+1) + 2 * (2n+2) + 2n + 2$$

$$T(n) = 2T(i+3) + 2 * \big(2 * (2n+2)\big) + 2 * (2n+2) + 2n + 2$$

$$T(n) = 2T(i+3) + 2 * 2 * (2n+2) + 2 * (2n+2) + 2n + 2$$

$$T(n) = 2T(i+4) + 2 * 2 * 2 * (2n+2) + 2 * 2 * (2n+2) + 2 * (2n+2) + 2n + 2$$

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

Fazendo k = n-1 e i = 0, temos:

$$T(n) = 2 * T(0) + (2n + 2) * \left(\frac{2 * (2^{k-1} - 2^{-1})}{2 - 1}\right)$$

$$T(n) = 2 + (2n + 2) * \left(2 * (2^{k} - 2^{0})\right)$$

$$T(n) = 2 + (2n + 2) * (2^{k+1} + 2)$$

$$T(n) = O(2^{n} * n)$$

A análise da complexidade do algoritmo com Backtracking deu O(n*2^n), mas pelos tempos de execução desse algoritmo em valores de n de 10 a 20, a complexidade de tempo do algoritmo deu exponencial(2^n), pela a analise ser teórica e a execução do algoritmo ser pratica, a complexidade de tempo do algoritmo com técnica de Backtracking é **O(2^n)** e a complexidade de espaço é de O(n) pelo fato que se usa dois vetores de tamanho n.

Tempo de execução das instancia de cada algoritmo de cada problema

Para cada algoritmo de cada problema, foi criado 3 instancias e executado 3 vezes para se ter um intervalo de tempo de execução.

O problema de multiplicação de matriz teve 3 instâncias para o algoritmo O(n^3) de tamanhos 1000, 2500 e 3584, todos os elementos das matrizes A e B foram geradas aleatoriamente em um valor no intervalor de 0 a 100.000. A Tabela 1 apresenta os resultados das 3 execuções das 3 instâncias.

Tabela 1. Nome dos arquivos para as instancias e 3 tempos de execução de cada no algoritmo O(n^3)

Nome do Arquivo / Instancia	Tempo 1	Tempo 2	Tempo 3
n_1000nxn.txt / 1	5.844s	7.640s	10.375s
n_2500nxn.txt / 2	188.462s	189.676s	190.048s
n_3584nxn.txt / 3	489.945s	514.374s	585.663s

No algoritmo de O($n^{2.807}$) ou Strassen, foi usado instancias de tamanhos 512, 1024 e 2048, com valores também gerados aleatoriamente. A Tabela 2 apresenta os resultados

Tabela 2. Nome dos arquivos para as instancias e 3 tempos de execução de cada no algoritmo O($n^{2,807}$)

Nome do Arquivo / Instancia	Tempo 1	Tempo 2	Tempo 3
n_512nxn.txt / 1	23.477s	25.067s	26.330s
n_1024nxn.txt / 2	168.316s	171.657s	172.201s
n_2048nxn.txt / 3	1170.623s	1175.276[a]	1233.013s

O problema da mochila teve o mesmo conjunto que o problema anterior, 3 instancias com 3 tempos de execução, para o algoritmo com a técnica da programacao dinâmica, foi usado tamanhos de 2000000 para n e 100 para W, 50000 para n e 50000 para W, 11 para n e 200000000 para W. Para cada um dos itens, o peso e o valor foram gerados aleatoriamente com máximo sendo o peso total da mochila. A Tabela 3 apresenta os resultados dos testes.

Tabela 3. Nome dos arquivos para as instancias e 3 tempos de execução de cada no algoritmo O(n*W)

Nome do Arquivo / Instancia	Tempo 1	Tempo 2	Tempo 3
pdin-i1.txt / 1	0.828s	1.328s	4.876s
pdin-i2.txt / 2	44.161s	51.089s	85.964
pdin-i3.txt / 3	45.509s	52.134s	57.205s

Para o algoritmo com a técnica de Backtracking, foi usado nas 3 instancias os tamanhos n = 21 e W = 98539560, n = 32 e W = 656, e n = 34 e W = 12500, e o intervalo de valores de wi e vi de cada instancia foi: 0 a 100000000 para a instancia 1, 0 a 100000 para a instancia 2 e 0 a 1000 para a instancia 3. A Tabela 4 mostra o resultado dos 3 tempos de execução das 3 instancias.

Tabela 4. Nome dos arquivos para as instancias e 3 tempos de execução de cada no algoritmo O(2^n)

Nome do Arquivo / Instancia	Tempo 1	Tempo 2	Tempo 3
back-i1.txt / 1	0.070s	0.071s	0.075s
back-i2.txt / 2	208.376s	208.693s	209.780s
back-i3.txt / 3	764.096s	765.774s	772.561s

Analise dos resultados obtidos e comparação de algoritmos do mesmo problema

Os resultados obtidos para o problema da multiplicação de matrizes indica a que a complexidade do algoritmo calculada as vezes pode não ser a verdadeira, no caso do algoritmo de multiplicação de matrizes comum, este com complexidade de $O(n^3)$, ele apresentou resultados perto do esperado, tendo uma variação de 300s para 1084 linhas e colunas a mais, mas para o algoritmo de Strassen o tempo obtido não foi esperado, o algoritmo rodou rápido casos como uma multiplicação de matrizes 512x512, mas para a próxima potência de 2, no caso a 1024, ele rodou em uma velocidade abaixo da multiplicação normal, mas para o caso do 2048, ele rodou a 1000s a mais do que o 1024, devido a grande chamada de funções de $O(n^2)$, 42 mais ou menos, fez com que o programa tenha um tempo maior que o outro.

No problema da mochila, os resultados do algoritmo com Backtracking comprovaram a complexidade, o tempo de execução de uma instancia de 21 itens executou a menos de 1 segundo, já uma instancia com 31 itens executou a mais de 200 segundos, 10 itens e 200 segundos de diferença, para o algoritmo com programacao dinâmica, os resultados foram bem pertos por causa da tentativa de tentar alcançar um tempo mais alto através do aumento do W e do n, mas devido a limitação da memória, o programa não pode rodar uma instancia de tamanho maior.

A tabela 5 mostra o comparativo entre as 3 instancias do primeiro algoritmo executado por ambos os dois, e outra vez é usado 3 instancias, mas do segundo algoritmo executado por também ambos os algoritmos, sendo os dois do mesmo problema, que no caso é a multiplicação de matrizes.

Tabela 5. Comparativo entre os tempos das 3 instancias de cada algoritmo para o problema de multiplicação de matrizes.

Instancias	Tempo para o Algoritmo Mult. Matriz (s)	Tempo para o Algoritmo Strassen (s)_
Instancia 1 do Mult. Matriz	10.375	156.579
Instancia 2 do Mult. Matriz	190.048	> 60000
Instancia 3 do Mult. Matriz	585.663	> 60000
Instancia 1 do Strassen	0.805	26.330
Instancia 2 do Strassen	7.845	172.201
Instancia 3 do Strassen	84.389	1233.013

Analisando essa tabela, mostra a diferença entre os algoritmos, onde o $O(n^3)$ que tem uma complexidade maior, mas devido ao fato que não se usa nenhuma outra função ele se torna mais rápido que o algoritmo de Strassen que utiliza varias chamadas de função de $O(n^2)$. As instancias 2 e 3 da multiplicação de matriz teve tempo de execução maior de que 1 hora.

Na tabela 6 é feita a mesma coisa, executado as 6 instancias nos dois algoritmos, mas do problema da mochila.

Tabela 6. Comparativo entre os tempos das 3 instancias de cada algoritmo para o problema da mochila.

Instancias	Tempo para o Algoritmo Prog. Dinâmica (s)	Tempo para o Algoritmo Backtracking (s)
Instancia 1 do Prog. Dinam.	4.876	> 60000
Instancia 2 do Prog. Dinam.	85.964	> 60000
Instancia 3 do Prog. Dinam.	57.205	0
Instancia 1 do Backtracking	46.408	0.070
Instancia 2 do Backtracking	0	208.376
Instancia 3 do Backtracking	0.002	772.561

Ao comparar os dois algoritmos, temos a diferença entre a complexidade deles explicita, o Algoritmo de Programação Dinâmica tem foco no produto de n por W, assim, quando ambos forem baixo, no caso na instancia 2 e 3 do Backtracking, o tempo de execução dele foi de 0 ou perto de 0 segundos, mas quando há um numero alto de W e n, caso da instancia 1 do Backtracking, o tempo de execução foi maior em contraste com o tempo do Backtracking. Como o Backtracking tem foco no n, a instancia 1 e 2 da Programacao Dinâmica, que tem 2000000 e 50000 como valor de n, o tempo de execução em 2^n levaria mais de 1 hora, podendo levar ate dias para ser executado, mas em caso em que o W é um valor alto e n um baixo, o tempo de execução foi 0.

REFERENCIAS

[1]. Screenshots feita pelo autor.

Rômulo. Resolução de Recorrências. Disponível em <https://drive.google.com/file/d/0B27fNgyt7Xg_Q1pvTVN2ZjBzOUk/view>. Acessado em 25 jun. de 2019.