The Sequence Alignment Problem - Smith-Waterman Approach

Luís Fonseca nº 89066

***Resumo* – No presente relatório é abordada a aproximação de Smith-Waterman para resolver o problema do Alinhamento (local) de Sequências. Este procura encontrar regiões similares entre 2 sequências biológicas. Para tal são estudados 3 algoritmos diferentes: programação dinâmica, memoization e de recursividade. Por fim é feita uma comparação entre os mesmos.**

***Abstract* – This report abords the Smith-Waterman approach, to solve the (local) Sequence Alignment Problem. Three algorithms are studied: dynamic programming, memoization and recursion. Those are then compared.**

I. CONTEXT

O algoritmo de Smith-Waterman é um método bastante conhecido para a realização do alinhamento de sequências local. Foi motivado pela dificuldade de obtenção de alinhamentos corretos em regiões de baixa similaridade entre sequencia biológicas distantemente relacionadas. Desta forma tem como objetivo encontrar a medida de similaridade máxima envolvendo apenas parte das sequências envolvidas, que por ser local, não considera a sequência em todo o comprimento [2].

O algoritmo funciona, de forma bastante sucinta, da seguinte forma:

Para cada caracter da 1ª sequencia, compara-se com cada caracter da 2ª, procurando encontrar igualdade. A esta similaridade (ou falta dela), são atributos pontos – podendo estes ser negativos, mas caso o *score* de uma "comparação" seja negativo, este é igualado a 0. Desta forma, é construída uma espécie de cadeia de valores crescentes, indicando uma sequência similar.

Por fim, realiza-se a 2ª parte do algoritmo, onde se começa pelo maior valor de pontuação obtido, e se faz *backtrace*, isto é, anda-se para trás a partir desse valor até atingir o início da cadeia (valor 0), construindo-se assim a sequência similar.

II. CONSIDERATIONS

Considera-se ainda relevante expor alguns pontos a ter em conta associados ao estudo feito:

- Foram calculados tempos também para o processo de *Backtrace*, mas sendo estes diminutos (menor que os 0.2s), não foram alvo de estudo específico. Contudo estão incluídos no cálculo do tempo total de execução de cada algoritmo.
- Sabendo que **n** = (tamanho da 1ª sequencia – 1), e **m** = (tamanho da 2ª sequencia – 1), os testes feitos consideram **n**=**m**, uma vez que se considerou que para o *scope* deste projeto seria mais relevante estudar uma matriz quadrada.
- Usou-se um *Gap Penalty* Linear, ou seja, abrir ou estender um *gap* custa o mesmo.
- Nos algoritmos que retornam a sequencia comum, em caso de empate, apenas a 1ª sequencia é retornada.

III. DYNAMIC PROGRAMMING

De acordo com [1], *dynamic programming* "is a technique for solving problems of recursive nature, iteratively and is applicable when the computations of the subproblems overlap. Dynamic programming is typically implemented using tabulation, as in, you solve the problem "bottom up", i.e., by solving all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the "top" / original problem is then computed."

Dito isto, uma aproximação por DP parece bastante intuitiva:

Gera-se uma matriz n×m, notando que há uma adição de "1" a ambos os termos, requerimento do algoritmo. Isto permite que, iterativamente se comece pela base/**bottom** do problema, isto é, os caracteres inicias de ambas as

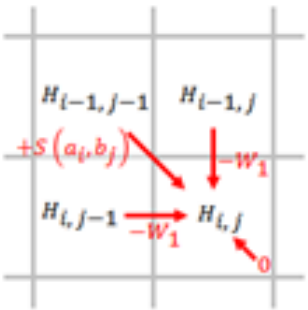


Figura 1 - Exemplificação do Funcionamento do Algoritmo

palavras (correspondendo à posição (1,1) da matriz) e se proceda ao cálculo de cada elemento em função das linhas e colunas anteriores.

Rapidamente se visualiza como funciona a iteração, e se percebe a forma como valores anteriormente calculados são necessários para cálculos "atuais".

Assim, um preenchimento da matriz com resultados já calculados e o seu aproveitamento para cálculos futuros apresenta-se como uma solução promissora.

A. *Time Complexity*

I. Empirical Analysis

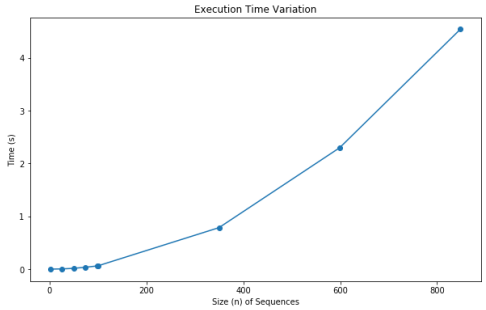


Figura 2 - Tempos de Execução de DP para n menores

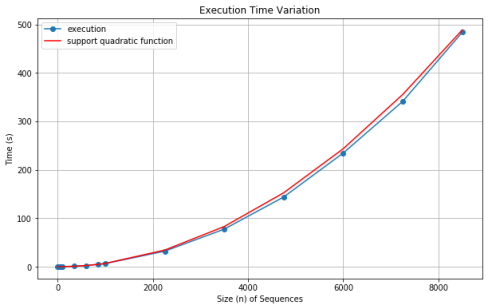


Figura 3 - Tempos de Execução de DP

Os gráficos precedentes, apresentam a evolução do tempo de execução em função do tamanho de **ambas** as sequencias (relembrando que cada uma tem tamanho **n**).

Como se pode observar, atentando principalmente ao 2º gráfico, este segue um crescimento quadrático, facto suportado pela similaridade que apresenta com a função quadrática de suporte, desenhada a vermelho $(0.0026 * n)^2$.

II. Formal Analysis

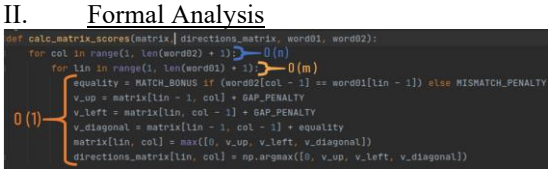


Figura 4 - Trecho de Código de DP

Analisando o trecho de código, é possível observar 2 *loops*. Estes percorrem cada caracter das sequências de *input*, iterando **n** e **m** vezes, respetivamente. Dentro destes blocos encontramos um conjunto de operações que se podem considerar elementares, não sendo sequer influenciadas diretamente pelo *input*.

Devido a este duplo *loop*, é possível perceber novamente, que é um algoritmo de complexidade quadrática.

Assim, para Dynamic Programming, temos a nível de complexidade temporal $O(n \times m) \Rightarrow O(n^2)$, se $n \approx m$.

Desta forma, para valores cada vez maiores de n, iremos ver os valores de tempo de execução a assemelharem-se bastante a $(0.0026 * n)^2$.

B. *Space Complexity*

Uma vez que se tem de criar uma matriz n×m, para este algoritmo a complexidade espacial do mesmo é $O(n \times m) \Rightarrow O(n^2)$, se $n \approx m$.

C. *Basic Operations*

Tendo em conta que o valor para um dado elemento da matriz depende de valores imediatamente anteriores, e como referido acima, a matriz segue um crescimento quadrático, então o conjunto de operações básicas sofre também um crescimento quadrático.

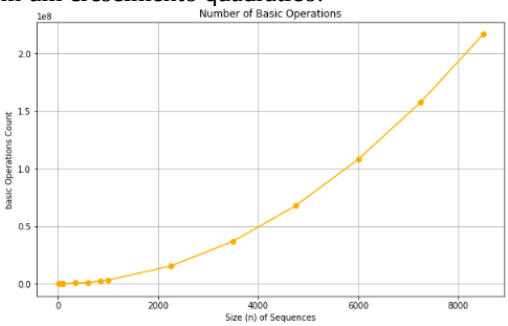


Figura 5 - *BO* de DP

IV. MEMOIZATION

Uma outra aproximação possível é através de *memoization*. Esta apresenta-se como refere [1] "a optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again. If you use memoization to solve the problem you do it by maintaining a map of already solved sub problems. You do it "top down" in the sense that you solve the "top" problem first (which typically recurses down to solve the sub-problems).

If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required".

Dito isto, no que se refere à última frase, salienta-se o facto de se prever que tal não aconteça, ou seja, no contexto deste problema, **todos** os casos devem ser considerados, pelo que talvez esta não seja a melhor solução.

Contrariamente ao método anterior, neste, sendo uma aproximação "top down", não se começa pelo caso base, mas sim pelos últimos caracteres de cada sequencia.

A. *Time Complexity*

I. Empirical Analysis

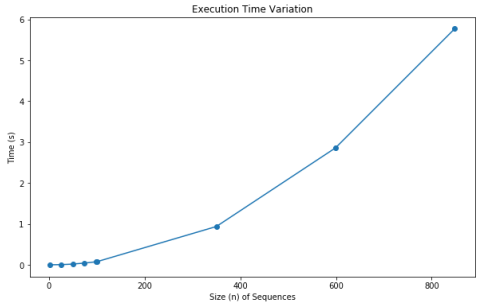


Figura 6 - Tempos de Execução de Memoization para n menores

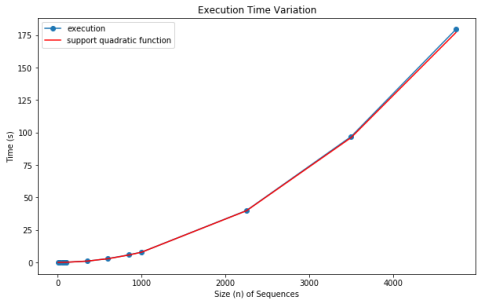


Figura 7 - Tempos de Execução de Memoization

Também neste algoritmo se pode perceber que existe um crescimento de forma quadrática no que diz respeito ao tempo de execução. A função a vermelho $(0.0028 * n)^2$ ajuda a suportar esta afirmação.

II. Formal Analysis

```
def calc_matrix_scores(matrix, directions_matrix, word01, word02, x, y):
    if matrix[x, y] != -1:
        return matrix[x, y]

    eq = (MATCH_BONUS if (word02[y - 1] == word01[x - 1]) else MISMATCH_PENALTY)
    v_diagonal = calc_matrix_scores(matrix, directions_matrix, word01, word02, x - 1, y - 1) + eq
    v_up = calc_matrix_scores(matrix, directions_matrix, word01, word02, x - 1, y) + GAP_PENALTY
    v_left = calc_matrix_scores(matrix, directions_matrix, word01, word02, x, y - 1) + GAP_PENALTY

    directions_matrix[x, y] = np.argmax([0, v_up, v_left, v_diagonal])
    matrix[x, y] = max(0, v_up, v_left, v_diagonal)
    return matrix[x, y]
```

Figura 8 - Trecho de Código de Memoization

Neste caso, é uma situação um pouco mais difícil de analisar, contudo rapidamente se chega a uma conclusão. Embora estejamos perante um algoritmo recursivo, em que para cada chamada a esta função, a mesma é chamada 3 vezes, relembre-se que valores já calculados são guardados. Desta forma, só é efetivamente necessário percorrer cada linha/coluna da matriz uma vez, já que, nas restantes vezes, o valor é imediatamente retornado, numa operação elementar, O(1). Assim, na prática deve-se apenas percorrer cada linha/coluna uma vez, e tendo cada uma tamanho **n**, estamos perante um algoritmo quadrático.

Assim, para Memoization, e à semelhança do anterior, temos a nível de complexidade temporal, $O(n \times m) \Rightarrow O(n^2)$, se $n \approx m$. Da mesma maneira, valores cada vez maiores de n, irão assemelhar-se cada vez mais de $(0.0028 * n)^2$.

B. *Space Complexity*

Pelos mesmos motivos que o anterior, terá complexidade espacial de $O(n \times m) \Rightarrow O(n^2)$, se $n \approx m$.

C. *Basic Operations & Function Calls*

Sendo um algoritmo baseado em recursividade, torna-se relevante fazer um estudo também de *Function Calls*. Assim, e comparativamente ao número de operações básicas temos:

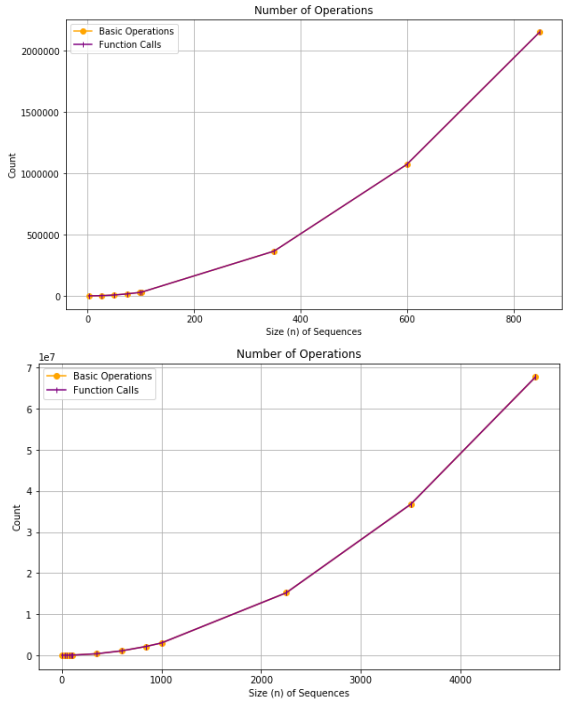


Figura 9 - *BO* e *FC* de Memoization

A sobreposição representada está relacionada com o facto de que *FC* excede o valor de *BO* apenas num valor, para qualquer caso.

Isto faz sentido uma vez que associado a cada *FC* está uma *BO*, pelo que a diferença de 1 entre estes valores se deve à chamada inicial, como se pode perceber pela figura abaixo.

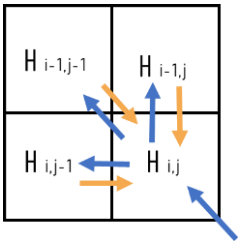


Figura 10 - Ilustração das Operações Feitas. Setas a azul - *FC*, setas a laranja - *BO*

Assim, por motivos idênticos ao análogo anterior, estamos perante um crescimento quadrático. Tanto é que os valores de *BO* são iguais aos do método de DP.

V. RECURSION

Por último analisa-se o caso de recursividade "pura". Pura no sentido em que se faz uso apenas da recursividade, abrindo mão do uso de qualquer *storing* de resultados já computados. Tal leva a que apenas se possa saber o *score* máximo atingido entre as duas sequências de *input*.

Só isso, já representa uma das grandes desvantagens desta aproximação: o facto de não ser possível realizar *traceback* para revelar qual a trecho comum entre sequências, o que afasta um pouco esta abordagem do objetivo do algoritmo de Smith-Waterman.

A. *Time Complexity*

I. Empirical Analysis

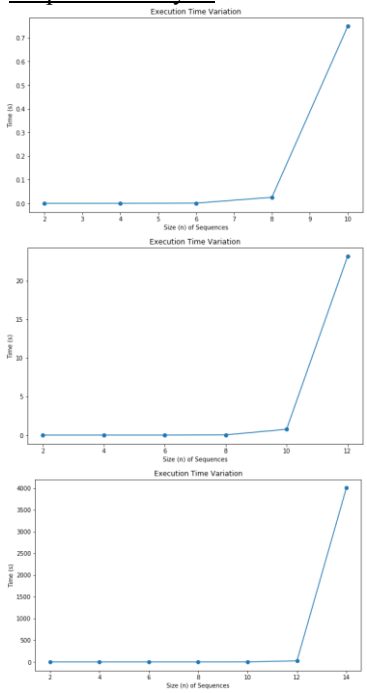


Figura 11 - Tempos de Execução de Recursividade até vários n

Através destes 3 gráficos, que parecem bastante idênticos, deve-se atender à amplitude do eixo das ordenadas e à forma como este aumenta entre gráficos. O uso destas 3 representações tem como objetivos salientar o aumento drástico que se dá entre o último valor em análise em relação ao anterior, servindo assim como indício de qual a eficiência desta aproximação. De n=12 para n=14, o tempo de execução salta de aproximadamente 23s para mais de uma 1h.

Apresentando um comportamento diferente do de uma função quadrática, optou-se por uma outra abordagem para o estudo desta.

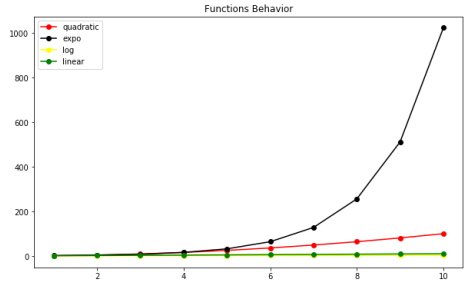


Figura 12 - Exemplo de Visualização de Várias Classes de Funções

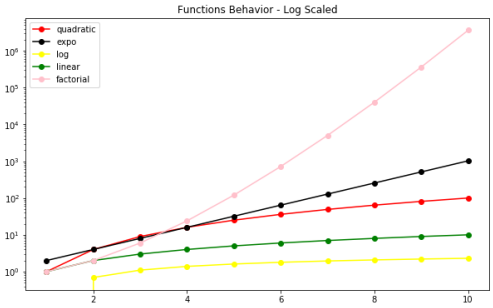


Figura 13 - Exemplo de Visualização de Várias Classes de Funções em Escala Logarítmica

Assim, deve-se agora analisar os dois gráficos anteriores.

O primeiro, demonstra o comportamento de várias "classes" de funções. Daqui, é importante salientar a forma como a exponencial cresce em relação às outras – incluindo a quadrática, e o facto de a função fatorial ter sido omitida, uma vez que tornava a visualização das restantes "linear".

O segundo, já inclui a visualização da função fatorial, porque aqui se usou uma escala logarítmica no eixo das ordenadas. Este segundo gráfico é de particular importância pois permite delinear de forma clara estes tipos de funções. Assim, é importante notar:

- A Função Quadrática, apresenta uma visualização Logarítmica.
- A Função Exponencial, apresenta uma visualização Linear.
- A Função Fatorial, apresenta uma visualização Quadrática.

Tendo isto em conta, e aplicando esta escala ao gráfico de tempos de execução deste algoritmo, obtemos a seguinte visualização:

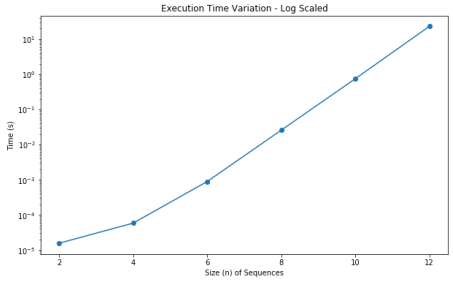


Figura 14 - Tempos de Execução de Recursividade em Escala Logarítmica

Nota-se rapidamente como obtemos praticamente uma reta, dando fundamento ao facto de este ser um algoritmo de crescimento exponencial (a^n).

O valor para n=14 não foi considerado, uma vez que foi testado noutras condições, e o elevado tempo de execução não tornava fazível uma re-execução do mesmo.

Note-se também que se usou este método nos algoritmos anteriores, de forma a verificar com mais confiança se se tratava realmente de funções quadráticas.

II. Formal Analysis

```
def smith_waterman(word01, word02, x, y):
    if x == 0 or y == 0:
        return 0, 0

    eq = (MATCH_BONUS if (word02[y - 1] == word01[x - 1]) else MISMATCH_PENALTY)

    v_up = smith_waterman(word01, word02, x - 1, y)  # + gap_penalty
    v_left = smith_waterman(word01, word02, x, y - 1)  # + gap_penalty
    v_diagonal = smith_waterman(word01, word02, x - 1, y - 1)  # + eq
    local_max_score = max(0, v_up[0] + GAP_PENALTY, v_left[0] + GAP_PENALTY, v_diagonal[0] + eq)
    return local_max_score, max(local_max_score, v_up[1], v_left[1], v_diagonal[1])
```

Atendendo a que, para cada chamada da função, a mesma é chamada outras 3 vezes, até atingir uma "profundidade" n, esta aparenta ser então uma função exponencial de complexidade $O(3^n)$.

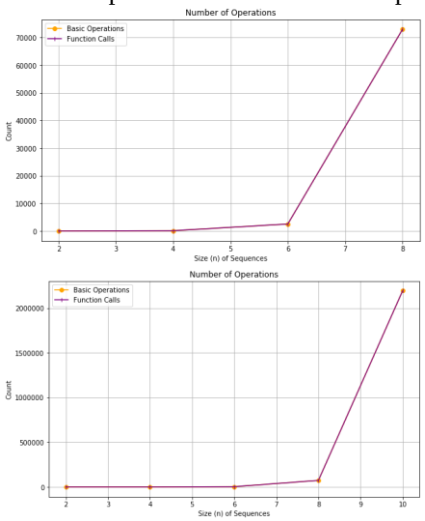
Assim, para valores maiores de n, os valores do tempo de execução apresentarão um comportamento exponencial idêntico a 3^n.

B. *Space Complexity*

Uma vez que não são registados quaisquer valores neste algoritmo, não surge a necessidade de criar qualquer estrutura de dados em particular.

C. *Basic Operations & Function Calls*

Por razões idênticas às agora explicadas, vemos que estamos novamente perante um crescimento exponencial.



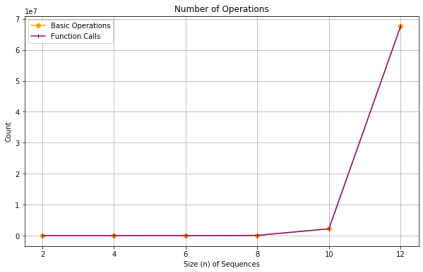


Figura 15 - *BO* e *FC* de Recursividade

A diferença de apenas um valor entre *BO* e *FC* justifica-se pelos mesmos motivos que no subcapítulo análogo de Memoization.

VI. CONCLUSION

A. *DP vs Memoization*

Apresentam-se assim como algoritmos semelhantes, exceto nos tempos de execução, onde aí a diferença se apresenta como relativamente pequena. Por esse motivo, exibesse em forma de tabela parte dos valores em questão – para **n** mais significativos, uma vez que numa representação gráfica não seriam percetíveis:

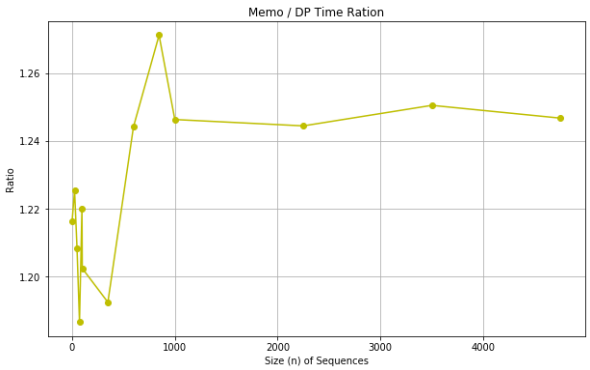
Tabela 1 - Comparação Tempos DP e Memo

n	DP	Memo
1001	6.331	7.890
2251	32.042	39.874
3501	77.471	96.877
4751	143.886	179.387

É possível observar que Memoization apresenta consistentemente tempos maiores em relação a DP, relativamente ao mesmo valor de **n.**

Para responder a tal facto, atente-se a [1]: "If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor". Tal deve-se, entre outros fatores, ao *overhead* associado à recursividade de Memoization.

Assim, pareceu interessante tentar encontrar este fator contante entre os 2 algoritmos. Tal originou o seguinte gráfico:



Aqui é apresentada a razão entre os tempos de execução de Memoization e DP. O facto de não se atingir n=5000,

está relacionado com os limites de recursividade impostos pelo Python3, e que se decidiu não exceder muito.

De todas as formas, é possível perceber que para valores de n maiores, isto é, mais significativos, esta razão parece convergir à volta de 1.24 ou 1.25. Assim, é possível concluir que Memoization apresenta uma performance cerca de 1.24 vezes pior do que DP.

B. Memoization vs Recursion

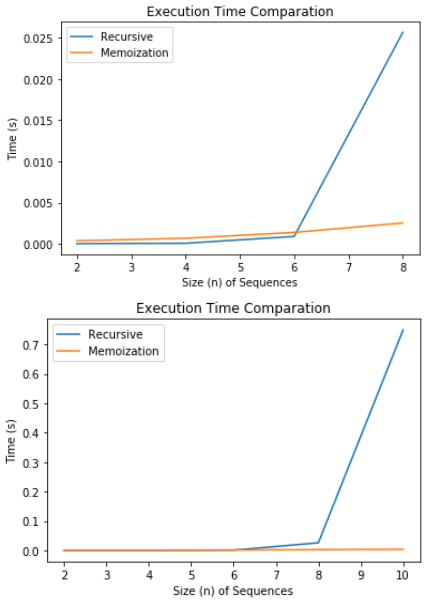


Figura 16 - Comparação dos Tempos de Execução de Recursividade e Memoization

Observando os 2 gráficos anteriores, percebe-se como rapidamente o tempo de execução do algoritmo recursivo ultrapassa o de memoization. Apesar de para valores pequenos (até n=6), recursividade ter uma performance "melhor", não só a diferença temporal é negligenciável, como acontece para comprimentos de sequências sem utilidade prática. Além disso, como referido anteriormente, recursividade tem também a desvantagem de ser incapaz de permitir *backtrace*, ou seja, de apresentar a sequência comum.

Desta forma, torna-se óbvia a clara ineficiência do algoritmo de recursividade em relação ao de memoization.

Assim, a ter de classificar os algoritmos em termos de eficiência, teríamos DP > Memoization > Recursion.

VII. HOW TO RUN

Como última nota, apresenta-se a forma como se podem correr cada um dos 3 algoritmos desenvolvidos:

```
python3 dp_smith_waterman.py AATCG AACG
lines: 6 | cols: 5
('AATCG', 'AA_CG', 6.0)
```

```
python3 memo_smith_waterman.py AATCG AACG
lines: 6 | cols: 5
('AATCG', 'AA_CG', 6.0)
```

```
python3 recur_smith_waterman.py AATCG AACG
 1st element - score of element at len(arg1),len(arg2)
 2nd element - max score of algorithm
(6, 6)
```

Figura 17 - Demonstração de como executar os Algoritmos

Como se pode observar, é apenas necessário indicar o nome do algoritmo que se pretende usar, seguido das 2 sequencias a analisar.

REFERENCES

[1] [ONLINE]. AVAILABLE: HTTPS://STACKOVERFLOW.COM/A/6185005.

[2] [ONLINE]. AVAILABLE: HTTPS://PT.WIKIPEDIA.ORG/WIKI/ALGORITMO_DE_SMITH-WATERMAN.

[3] [ONLINE]. AVAILABLE: HTTPS://STACKOVERFLOW.COM/A/13467808.