Determinação da subsequência crescente mais longa

João Amaral – nº 65772

***Resumo* – Este relatório encontra-se dividido em cinco capítulos. Sendo o primeiro capitulo direcionado sobre o trabalho e os objectivos. O segundo será uma breve explicação sobre o que é a Subsequência crescente mais longa, também conhecida por LIS. O terceiro capitulo consiste em uma breve explicação de cada algoritmo seguindo-se da explicação do trabalho efectuado. No quarto capitulo encontra-se os testes que foram executados com vários *arrays* com diferentes tamanhos e os respectivos *outputs*. No quinto e ultimo capitulo, encontra-se a conclusão que contém as conclusões sobre os resultados do teste assim como uma pequena reflexão sobre o que se aprendeu neste trabalho.**

***Abstract* - This report is divided into five chapters. The first chapter directed about the work is the objectives. The second will be a brief explanation of what is the longest increasing subsequence also known as LIS. The third chapter consists of a brief explanation of each followed by the explanation of the work done. In the fourth chapter are the tests that were executed with several arrays with different sizes and their outputs. In the fifth and last chapter is the conclusion that contains the conclusions about the test results as well as a small reflection on what was learned in this work**

I. INTRODUÇÃO

O trabalho prático realizado e descrito neste relatório pertence ao âmbito da disciplina de Algoritmos Avançados que pertence ao Mestrado de Engenharia de Informática da Universidade de Aveiro.

O trabalho proposto pelo docente da disciplina foi a determinação da subsequência crescente mais longa através do uso de programação dinâminca.

Através de diferentes algoritmos pretende-se avaliar o desempenho computacional de cada algoritmo assim como a sua eficiência e limitações dos mesmos.

Para o desenvolvimento do trabalho proposto foi desenvolvido um algoritmo recursivo, incluindo uma versão do mesmo que tira proveiro da abordagem "*memoization"* e um algoritmo de Programação Dinâmica.

Cada algoritmo terá uma breve descrição assim como uma exposição de como funciona e as soluções encontradas para atingir o objectivo proposto.

Para efectuar a avaliação de cada algoritmo será calculado o tempo de execução de cada algoritmo com diferentes tamanhos de entrada de dados e o numero de operações que cada um deles efectua até chegar a conclusão.

II. *LONGEST INCREASING SUBSEQUENCE* (LIS)

O objectivo do problema entregue com o objectivo de desenvolver os algoritmos é encontrar o tamanho da mais longa subsequência de um array de inteiros de modo que estejam em ordem crescente.

Na figura 1 encontra-se um exemplo do que se pretende.

Sendo "6" o resultado do exemplo apresentado na figura 1, ou seja, o tamanho da maior subsequencia crescente é "6". Sendo a subsquencia o seguinte {15, 27, 38, 55, 65, 85}.[1]

A = {15, 27, 14, 38, 26, 55, 46, 65, 85}

LIS = 6

SubSeq = {15, 27, 38, 55, 65, 85}

Figura 1 - Exemplo de resolução do LIS

As soluções apresentadas neste artigo foram adquiridas e desenvolvidas através do uso de programação dinamica.

III. ALGORITMO DESENVOLVIDO

A. *Dynamic Programing*

a. Analise de complexidade do algoritmo

Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória.[2]

Algoritmo:

```
SSCTF-Max-PD (A, n)
  from  m ← 1  to  n  do
     c[m] ← 1
     from  i ← m-1  to  1  do
        if  A[i] ≤ A[m]  and  c[i]+1 >
    c[m]
           c[m] ← c[i]+1
  return c[1..n]
```

Para resolver o problema proposto através de *dynamic programming* é preciso criar um array com o mesmo

tamanho que o *array* fornecido e preenche-lo com o numero 1 (Figura 2).

A:	3	4	-1	0	6
I:	1	1	1	1	1

Figura 2 - A: *Array* Fornecido | I: *Array* Auxiliar

Um vez com os dois *arrays* criados, é preciso percorrer o *array* fornecido com recurso a dois contadores (i e j) de modo que quando o valor do *array* em array[i] mudar de posição, seja necessário percorrer todas as posições anteriores à que se encontra recorrendo ao segundo contador.

Sempre que se mudar de posição no *array* com o segundo contador é preciso ter atenção a duas condições:

- Verificar se o valor em A[j], for maior que o valor em A[i]

```
A[j] < A [i]
```

- Se o valor no array auxiliar na posição i seja menor que o valor nesse mesmo array na posição j + 1

```
I[i] < I[j] + 1
```

Caso estas condições sejam verdadeiras o valor no *array* auxiliar na posição I[i] será substituído pelo valor de I[j] + 1. Dando o resultado final que se mostra na Figura 3.

I:	1	2	1	2	3

Figura 3 - Array auxiliar no fim do algoritmo

Observando o resultado final no *array* auxiliar, podemos verificar o tamanho maximo da maior subsequência crescente presente no array fornecido.

b. Codigo de resolução usando a linguagem python

```
#Dynamic programming
def lisDP(num):
    #Tamanho do array inserido
    n = len(num)

    #Array auxiliar com para inicialicar o LIS
    s = len(num) * [1]

    for i in range(1,n):
        for j in range(0, i):
            if num[i] > num[j] and s[i] < s[j] + 1:
                s[i] = s[j] + 1

    return (max(s))
```

Figura 4 – Código LIS usando *dynamic programming*

B. Recursive Algorithm

a. Analise de complexidade do algoritmo

Um *Recursive algorithm* é um algoritmo que se chama a si mesmo com valores de parâmentros mais simples que obtém o resultado para a entrada atual aplicando operações simples.

A abordagem recursiva concentra-se no paradigma *divide and conquer* que basicamente divide um grande problema em vários pequenos problemas até chegarem a um ponto que se torna fácil a resolução.[3]

Para poder aplicar um algoritmos recursivo, a estratégia que foi abordada foi a criação de duas funções, uma com o objectivo de "iniciar" a função e devolver o valor de uma variável global que irá conter o numero do tamanho da maior subsequência crescente.

A função recursiva por si recebe como parâmetro o *array* e o numero que consiste do tamanho máximo desse mesmo array.

Antes de se poder proceder para o ciclo aonde irá ocorrer a recursão em si é necessário determinar uma base fixa, digamos 1. Caso a função que seja chamada cujo tamanho do array seja 1 a função irá retornar 1 imediatamente.

O algoritmo desenvolvido, ao verificar que o tamanho do array inserido não é de facto "1", então entra num ciclo for com contador i, que na qual irá ocorrer a recursividade ao chamar a função por cada valor de i de forma recursiva. como esta representado um exemplo na figura5.

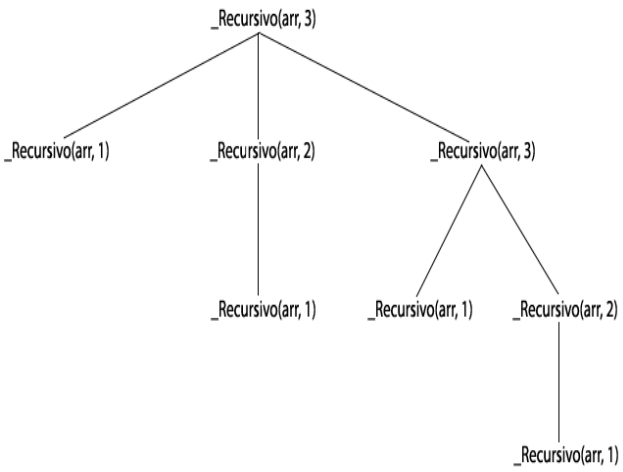

Figura 5 – Logica da recursividade

Depois de cada valor retornado pela função recursiva, é preciso ter atenção a duas condições:

- Se a posição em num[i-1] do array inserido for menor que a posição num[n -1]

```
num[i-1] < num[n-1]
```

- Se a soma do valor recebido pela função recursiva com 1 for superior ao tamanho da

sequencia já defenida. (Atenção, o tamanho da sequencia no começo do programa é 1 por defeito.)

```
valor + 1 > tempSize
```

Caso ambas as condições sejam verdadeiras o valor da variável correspondete ao tamanho da subsequência é atualizado, por fim, esse valor é comparado com o da variável global. Caso seja superior, o valor da variável global é atualizada. Sendo no fim do algoritmo, esse o valor do tamanho da maior subsequência mais longa.

b. Codigo de resolução usando a linguagem python

```python
def _Recursivo(arr, n):
    global valorMaximo

    #Caso base
    if n == 1:
        return 1

    #tamanho temporario calculado na recursao
    tempsize = 1


    # Se num[i-1] for menor que arr[n-1] calcula-se o
    # tamanho temporario
    # Se o valor calculado for maior que o valor na variaval globaa,
    # esta será substituida.
    for i in range(1, n):
        valor = _Recursivo(arr, i)
        if arr[i - 1] < arr[n - 1] and valor + 1 > tempsize:
            tempsize = valor + 1

    #Compara o tempsize com o valor da variavel global valorMaximo,
    #actualizando a variavel global caso necessario!
    valorMaximo = max(valorMaximo, tempsize)

    return tempsize
```

Figura 6 - Código LIS usando um algoritmo recursivo

C. *Recursive com Memoization*

a. Análise de complexidade do algoritmo

Memoization foi desenvolvida para situações peliculares aonde usando a recursividade o algoritmo funciona, mas à medida que num[n] cresce, mais lento se torna a resolução devido a necessidade de recalcular os mesmos cálculos varias vezes.[4]

Para resolver essa situação usa-se a *memoization* que consiste em gravar os resultados de cálculos já efectuados para que esses mesmos cálculos não sejam repetidos.

Para implementar *momoization* no algoritmo anteriormente explicado, foi implementado uma *hashtable* aonde na qual será guardada o resultados dos cálculos.

Visto que, como foi explicado anteriormente, sempre que a função entra no ciclo com um contador i, a função recursiva irá calcular todos os valores até i. Através desta técnica só terá de calcular uma vez visto que o resultado serão guardado sendo o i na altura do calculo como chave. Nesta maneira, o algoritmo antes de efectuar qualquer cálculo irá verificar se a *key* existe. Se existir, o *value* correspondente é utilizado, caso contrario efectua o calculo e guarda o valor com a nova *key*.

O processo encontra-se demonstrado na figura 7.

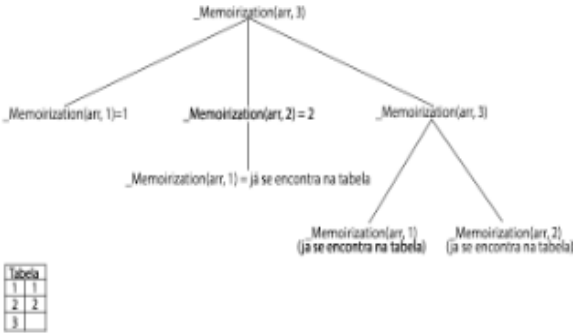

Figura 7 - Logica da *Memoization*

b. Codigo criado

```python
def _Memoization(arr, n, memo=[]):
    global valorMaximo

    if n == 1:
        return 1

    tempsize = 1

    for i in range(1, n):
        # Caso exista o valor de i na tabela devolve o valor correspondente
        if i in memo:
            valor = memo[i]
        else: #Caso o valor i nao exista na tabela a key é inserida mais o valor correspondente
            valor = _Memoization(arr, i)
            memo[i] = valor
        if arr[i - 1] < arr[n - 1] and valor + 1 > tempsize:
            tempsize = valor + 1

    valorMaximo = max(valorMaximo, tempsize)

    return tempsize
```

Figura 8 - Código LIS usando um algoritmo recursivo com *memoization*

IV. TESTES

Teste 1: Array : [2, 6, 4, 5, 1, 3]

```
---------------------------
Usando dynamic Programming:
3
Tempo de processamento: 0.00009 segundos
Numero de operações: 15
---------------------------
Usando algoritmo recursivo:
3
Tempo de processamento: 0.00008 segundos
Numero de operações: 31
---------------------------
Usando recursivo recorrendo a memoirization:
3
Tempo de processamento: 0.00004 segundos
Numero de operações: 5
---------------------------
```

Teste 2: Array : [10, 22, 9, 33, 21, 50, 41, 60]

```
--------------------------
Usando dynamic Programming:
5
Tempo de processamento: 0.00008 segundos
Numero de operações: 28
--------------------------
Usando algoritmo recursivo:
5
Tempo de processamento: 0.00023 segundos
Numero de operações: 127
--------------------------
Usando recursivo recorrendo a memoirization:
5
Tempo de processamento: 0.00005 segundos
Numero de operações: 7
--------------------------
```

Teste 3: Array : [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7]

```
--------------------------
Usando dynamic Programming:
5
Tempo de processamento: 0.00022 segundos
Numero de operações: 105
--------------------------
Usando algoritmo recursivo:
5
Tempo de processamento: 0.03325 segundos
Numero de operações: 16383
--------------------------
Usando recursivo recorrendo a memoirization:
5
Tempo de processamento: 0.00011 segundos
Numero de operações: 14
--------------------------
```

Teste 4: Array: [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 10, 22, 9, 33, 21, 50]

```
--------------------------
Usando dynamic Programming:
8
Tempo de processamento: 0.00020 segundos
Numero de operações: 210
--------------------------
Usando algoritmo recursivo:
8
Tempo de processamento: 1.80663 segundos
Numero de operações: 1048575
--------------------------
Usando recursivo recorrendo a memoirization:
8
Tempo de processamento: 0.00022 segundos
Numero de operações: 20
--------------------------
```

V. CONCLUSÃO

Durante a realização deste trabalho foi possível estudar a fundo diferentes abordagens de resolução de um problema que será uma adição de peso para o desenvolvimento de futuros projectos visto que irão facilitar o desenvolvimento do mesmo.

Durante o desenvolvimento do trabalho, certas dificuldades foram mais notórias durante o desenvolvimento do algoritmo de recursividade devido à extrema preocupação de ocorrer algum ciclo infinito a meio execução e também devido à extensa recursividade. Dependendo do array inserido, o algoritmo poderá atingir algum problema que poderá dificultar a compreensão do mesmo o que irá levar a dificuldade de resolução de problemas.

A secção de testes apresenta dados interessantes, monstrando que o para diferentes tamanhos de arrays, o algoritmo recursivo com *memoization* tem de facto a execução mais rápida com o menor número de operações. Sendo o algoritmo recursivo o mais lento com uma diferença baste considerável tanto em tempo de execução como em numero de operações.

Em nota pessoal, apesar da tentação de escollher a opção de desenvolver qualquer problema em programação dinâmica devido a sua facilidade, tenho de reconhecer que o algoritmo recursivo com *memoization* é uma verdadeira pedra mestra no que se refere em optimização.

REFERENCES

Use the Style "referencia" to the references. Example:

[1] Steven Skiena, 1997, Chapter 8, The Algorithm Design Manual

[2] https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html.

[3] Pdf da aula - Algorithm Design Strategies I

[4] http://www.radford.edu/~nokie/classes/360/dp-memoized.html