

## LISTA DE EXERCÍCIOS 2

As questões, a seguir, tratam do projeto de algoritmos. CLSR é a referência do exercício no livro-texto. As questões estão em inglês por terem sido copiadas da versão original.

**Questão 1 [CLRS, Ex 15.2-4, pg 338]**

Let  $R(i, j)$  be the number of times that table entry  $m[i, j]$  is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is:

$$\sum_{i=1}^n \sum_{j=1}^n R(i, j) = \frac{n^3 - n}{3}.$$

(Hint: You may find equation (A.3) useful.)

**Questão 2 [CLRS, Ex 15.3-1, pg 349]**

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

**Questão 3 [CLRS, Ex 15.3-2, pg 349]**

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization is ineffective in speeding up a good divide-and-conquer algorithm such as MERGE-SORT.

Na Wikipedia (versão em inglês), o termo *memoization* tem a seguinte entrada (<http://en.wikipedia.org/wiki/Memoization>):

The term “memoization” was coined by Donald Michie in 1968 and is derived from the Latin word memorandum (to be remembered), and thus carries the meaning of turning [the results of] a function into something to be remembered. While memoization might be confused with memorization (because of the shared cognate), memoization has a specialized meaning in computing.

A memoized function “remembers” the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it, thus moving the primary cost of a call with given parameters to the first call made to the function with those parameters. The set of remembered associations may be a fixed-size set controlled by a replacement algorithm or a fixed set, depending on the nature of the function and its use. A function can only be memoized if it is referentially transparent; that is, only if calling the function has the exact same effect as replacing that function call with its return value. (Special case exceptions to this restriction exist, however.) While related to lookup tables, since memoization often uses such tables in its implementation, memoization differs from pure table lookup in that the tables which memoization might use are populated transparently on an as-needed basis.

Memoization is a means of lowering a function’s time cost in exchange for space cost; that is, memoized functions become optimized for speed in exchange for a higher use of computer memory space. The time/space “cost” of algorithms has a specific name in computing: computational complexity. All functions have a computational complexity in time (i.e. they take time to execute) and in space.

#### Questão 4 [CLRS, Ex 15.3-5, pg 350]

As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that it is not always necessary to solve all the subproblems in order to find an optimal solution. She suggests that an optimal solution to the matrix-chain multiplication problem can be found by always choosing the matrix  $A_k$  at which to split the subproduct  $A_i A_{i+1} \dots A_j$  (by selecting  $k$  to minimize the quantity  $p_{i-1} p_k p_j$ ) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

#### Questão 5 [CLRS, Prob 15-1, pg 364]

The *euclidean traveling-salesman problem* is the problem of determining the shortest closed tour that connects a given set of  $n$  points in the plane. Figure 15.9(a) shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. Figure 15.9(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an  $O(n^2)$ -time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. (*Hint*: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

#### Questão 6 [CLRS, Prob 15-3, pg 364], Opcional

In order to transform one source string of text  $x[1..m]$  to a target string  $y[1..n]$ , we can perform various transformation operations. Our goal is, given  $x$  and  $y$ , to produce a series of transformations that change  $x$  to  $y$ . We use an array  $z$ —assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially,  $z$  is empty, and at termination, we should have  $z[j] = y[j]$  for  $j = 1, 2, \dots, n$ . We maintain current indices  $i$  into  $x$  and  $j$  into  $z$ , and the operations are allowed to alter  $z$  and these indices. Initially,  $i = j = 1$ . We are required to examine every character in  $x$  during the transformation, which means that at the end of the sequence of transformation operations, we must have  $i = m + 1$ .

There are six transformation operations:

**Copy** a character from  $x$  to  $z$  by setting  $z[j] \leftarrow x[i]$  and then incrementing both  $i$  and  $j$ . This operation examines  $x[i]$ .

**Replace** a character from  $x$  by another character  $c$ , by setting  $z[j] \leftarrow c$ , and then incrementing both  $i$  and  $j$ . This operation examines  $x[i]$ .

**Delete** a character from  $x$  by incrementing  $i$  but leaving  $j$  alone. This operation examines  $x[i]$ .

**Insert** the character  $c$  into  $z$  by setting  $z[j] \leftarrow c$  and then incrementing  $j$ , but leaving  $i$  alone. This operation examines no characters of  $x$ .

**Twiddle** (i.e., exchange) the next two characters by copying them from  $x$  to  $z$  but in the opposite order; we do so by setting  $z[j] \leftarrow x[i + 1]$  and  $z[j + 1] \leftarrow x[i]$  and then setting  $i \leftarrow i + 2$  and  $j \leftarrow j + 2$ . This operation examines  $x[i]$  and  $x[i + 1]$ .

**Kill** the remainder of  $x$  by setting  $i \leftarrow m + 1$ . This operation examines all characters in  $x$  that have not yet been examined. If this operation is performed, it must be the final operation.

As an example, one way to transform the source string algorithm to the target string **altruistic** is to use the following sequence of operations, where the underlined characters are  $x[i]$  and  $z[j]$  after the operation:

Operation	$x$	$z$
<i>initial strings</i>	algorithm	_
copy	a_lgorithm	a_
copy	al_gorithm	al_
replace by t	alt_gorithm	alt_
delete	altr_gorithm	altr_
copy	altru_gorithm	altru_
insert u	altruist_gorithm	altruist_
insert i	altruistic_gorithm	altruistic_
insert s	altruistic_gorithm	altruistic_
twiddle	altruistic_gorithm	altruistic_
insert c	altruistic_gorithm	altruistic_
kill	altruistic_gorithm	altruistic_

Note that there are several other sequences of transformation operations that transform **algorithm** to **altruistic**.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming **algorithm** to **altruistic** is

$$(3 \cdot \text{cost}(\text{copy})) + \text{cost}(\text{replace}) + \text{cost}(\text{delete}) + (4 \cdot \text{cost}(\text{insert})) + \text{cost}(\text{twiddle}) + \text{cost}(\text{kill}).$$

- (a) Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$  and set of transformation-operation costs, the edit distance from  $x$  to  $y$  is the cost of the least expensive operation sequence that transforms  $x$  to  $y$ . Describe a dynamic-programming algorithm that finds the edit distance from  $x[1 \dots m]$  to  $y[1 \dots n]$  and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem is a generalization of the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [272, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences  $x$  and  $y$  consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences  $x'$  and  $y'$  have the same length but do not have a space in the same position (i.e., for no position  $j$  are both  $x'[j]$  and  $y'[j]$  a space.) Then we assign a "score" to each position. Position  $j$  receives a score as follows:

- +1 if  $x'[j] = y'[j]$  and neither is a space,
- -1 if  $x'[j] \neq y'[j]$  and neither is a space,
- -2 if either  $x'[j]$  or  $y'[j]$  is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences  $x = \text{GATCGGCAT}$  and  $y = \text{CAATGTGAATC}$ , one alignment is

```
G ATCG GCAT
CAAT GTGAATC
-*****+---**
```

A + under a position indicates a score of +1 for that position, a - indicates a score of -1, and a \* indicates a score of -2, so that this alignment has a total score of  $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$ .

- (b) Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

### Questão 7 [CLRS, Ex 16.1-1, pg 378]

Give a dynamic-programming algorithm for the activity-selection problem, based on the recurrence (16.3). Have your algorithm compute the sizes  $c[i, j]$  as defined above and also produce the maximum-size subset  $A$  of activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

**Questão 8 [CLRS, Ex 16.1-2, pg 378]**

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

**Questão 9 [CLRS, Ex 16.1-3, pg 379]**

Suppose that we have a set of activities to schedule among a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This is also known as the *interval-graph coloring problem*. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices are given the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

**Questão 10 [CLRS, Ex 16.2-1, pg 384]**

Prove that the fractional knapsack problem has the greedy-choice property.

**Questão 11 [CLRS, Ex 16.2-2, pg 384]**

Give a dynamic-programming solution to the 0–1 knapsack problem that runs in  $O(nW)$  time, where  $n$  is number of items and  $W$  is the maximum weight of items that the thief can put in his knapsack.

**Questão 12 [CLRS, Ex 16.2-4, pg 384]**

Professor Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel  $n$  miles, and his map gives the distances between gas stations on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations he should stop, and prove that your strategy yields an optimal solution.

**Questão 13 [CLRS, Ex 16.2-6, pg 384], Opcional**

Show how to solve the fractional knapsack problem in  $O(n)$  time.

**Questão 14 [CLRS, Ex 16.3-1, pg 392]**

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

**Questão 15 [CLRS, Ex 16.3-2, pg 392]**

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first  $n$  Fibonacci numbers?

**Questão 16 [CLRS, Ex 16.3-4, pg 392]**

Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

---

Para cada um dos problemas abaixo, escreva o algoritmo, faça a implementação, testes e apresente o custo de complexidade identificando a operação considerada relevante. Lembre-se que a linguagem de programação é C. No caso de apresentar uma solução recursiva, discuta também e apresente a complexidade para o crescimento da pilha.

## Exercício de Programação 1: Tangram

O Tangram é um jogo composto de sete peças planas, chamadas “tans”, que são colocadas juntas para formar diversas figuras. O objetivo do jogo é formar uma figura específica, dada apenas sua silhueta, utilizando as sete peças, que não podem se sobrepor, conforme mostrado na figura 1

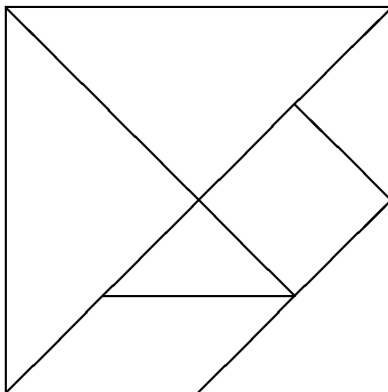


Figura 1: As sete peças do Tangram.

Os tamanhos das peças são dados em relação à um quadrado grande contendo todas elas, com altura, largura e área iguais a 1:

- 5 triângulos:
  - 2 pequenos (hipotenusa de  $\frac{1}{2}$  e lados de  $\frac{1}{2\sqrt{2}}$ );
  - 1 médio (hipotenusa de  $\frac{1}{\sqrt{2}}$  e lados de  $\frac{1}{2}$ );
  - 2 grandes (hipotenusa de 1 e lados de  $\frac{1}{\sqrt{2}}$ );
- 1 quadrado (lado de  $\frac{1}{2\sqrt{2}}$ );
- 1 paralelogramo (lados de  $\frac{1}{2}$  e de  $\frac{1}{\sqrt{2}}$  e ângulos de  $45^\circ$  e  $135^\circ$ ).

Este jogo pode ser resolvido computacionalmente e existem várias heurísticas para resolvê-lo. Dessa forma, os formatos de entrada e saída para o algoritmo são descritos a seguir.

**Entrada:** Serão dadas as coordenadas cartesianas dos pontos extremos da silhueta da figura. Sendo assim, as semi-retas que ligam esses pontos definem a silhueta da figura a ser investigada. Um arquivo texto contendo um par de coordenadas em cada linha é o padrão de entrada do algoritmo.

**Saída:** Devem ser definidas as posições de cada uma das sete peças dentro da silhueta dada, isto é, o arquivo de saída deve conter sete linhas, cada uma delas com as coordenadas cartesianas da peça em questão. Para ordenar as peças de saída, será atribuído um identificador para cada uma delas:

1. triângulo pequeno;
2. triângulo pequeno;
3. triângulo médio;
4. triângulo grande;
5. triângulo grande;
6. quadrado;
7. paralelogramo.

**Exemplo:** Considere a figura 1. O arquivo de entrada conteria:

```
0; 0
1; 0
1; 1
0; 1
```

O arquivo de saída correspondente seria:

```
(0; 0) (0; 5; 0) (0; 25; 0; 25)
```

(0; 75; 0; 25) (0; 75; 0; 75) (0; 5; 0; 5)  
(0; 5; 0) (1; 0) (1; 0; 5)  
(0; 0) (0; 5; 0; 5) (0; 1)  
(0; 1) (0; 5; 0; 5) (1; 1)  
(0; 5; 0) (0; 75; 0; 25) (0; 5; 0; 5) (0; 25; 0; 25)  
(0; 75; 0; 25) (1; 0; 5) (1; 1) (0; 75; 0; 75)

### Avaliação Experimental de Algoritmos

O projeto e análise de algoritmos envolve vários aspectos. Dentre eles temos a avaliação experimental, que já foi discutida em sala de aula, e tem um papel importante em pelo menos dois pontos: estudo de aspectos práticos de implementação de um algoritmo numa dada plataforma computacional e estudo do caso médio de um algoritmo quando não se conhece bem o seu comportamento ou quando esse estudo é difícil.

Fazer uma avaliação experimental exige um projeto cuidadoso dos experimentos. O primeiro ponto é a definição das métricas que serão avaliadas, que dependem fundamentalmente do algoritmo. O segundo ponto é a definição de “estratégias” para resolver o problema, principalmente se não houver uma solução polinomial conhecida. O terceiro ponto diz respeito aos tipos de dados que serão gerados para executar o programa e coletar as métricas de interesse. Por exemplo, serão geradas entradas aleatórias? Ou as entradas serão agrupadas em classes que representam casos distintos? O quarto ponto trata da análise estatística dos resultados obtidos e da análise crítica dos resultados. Fazer uma análise, em Ciência da Computação, significa entender os resultados e explicá-los de forma convincente, i.e., ANALISAR = ENTENDER + EXPLICAR. Possivelmente nessa análise vão existir resultados “interessantes”, inesperados, comprobatórios, etc, que devem ser entendidos pelo projetista dos experimentos e explicados em função das métricas definidas, das estratégias propostas e dos tipos de dados que foram usados.

Os próximos exercícios de programação têm como objetivo a avaliação experimental de algoritmos. Procure explorar os pontos descritos acima, por exemplo, a partir das sugestões apresentadas.

### Exercício de Programação 2: Passeio do Cavalo

O problema do Passeio do Cavalo pede para achar um passeio num tabuleiro de tamanho  $n \times n$ , para  $n \geq 5$ , que comece em cada qualquer desse tabuleiro e visite todas as casas uma única vez, executando o movimento em “L” do cavalo, conforme definido no jogo de xadrez.

O paradigma usual para solucionar este problema é o de tentativa e erro, já que não se conhece um algoritmo polinomial para resolvê-lo.

Sugestões de métricas:

- Movimentos executados com sucesso antes do primeiro retrocesso;
- Quantidade de vezes que movimentos tiveram que ser desfeitos;
- Quantidade total de movimentos desfeitos;
- Quantidade de vezes que uma casa foi visitada;
- Tempo real de execução.

Sugestões de estratégias:

- Escolher os movimentos de forma determinística;
- Escolher os movimentos de forma aleatória;
- Definir tabuleiros de tamanhos de  $5 \times 5$  a  $10 \times 10$  (justificar esses valores).

Tipos de dados:

- Coletar os dados referentes às métricas e estratégias considerando, simultaneamente, todas as casas do tabuleiro;
- Fazer a mesma análise considerando apenas os passeios que começam em casas na borda do tabuleiro, na segunda borda mais interna, etc.

### Exercício de Programação 3: PCV e a Heurística do Vizinho mais Próximo

A literatura do Problema do Caixeiro Viajante (PCV) é extremamente rica e existem várias heurísticas propostas, dentre elas o do vizinho mais próximo (VMP). Este problema trata da avaliação de estratégias do uso do VMP

para resolver o PCV.

Sugestões de estratégias:

- Em cada passo, determinar a cidade mais próxima a partir de um único extremo do caminho;
- Em cada passo, determinar a cidade mais próxima a partir dos dois extremos do caminho;
- Particionar o conjunto de cidades em dois ou mais e aplicar a heurística do VMP em cada conjunto separadamente;
- Propor variantes da heurística do VMP a partir dos dados coletados.

#### **Exercício de Programação 4: Conjectura do Número Palíndromo (Opcional)**

Existe uma conjectura que diz que para qualquer número inteiro se ele for somado ao seu reverso será obtido um número que não se for palíndromo, pode-se repetir esse processo (somar com o reverso) até obter um número palíndromo.

Este problema trata da avaliação dessa conjectura e você deve definir todos os pontos mencionados acima.

Exemplos:

1958	1970
<u>8591</u>	<u>0791</u>
10549	2761
<u>94501</u>	<u>1672</u>
105150	4433
<u>051501</u>	<u>3344</u>
156651	7777