UFMG/ICEx/DCC — Projeto e Análise de Algoritmos

# Lista de Exercícios 2

Pós-Graduação em Ciência da Computação — 1º Semestre de 2011

**Observações**:

1. Comece a fazer esta lista imediatamente. Você nunca terá tanto tempo para resolvê-lo quanto agora!
2. **Data de Entrega**: até 2 de maio, **às 9:30 horas**, ou antes. Após essa data e hora haverá uma penalização por atraso: $2^d$, onde $d$ é o número de dias de atraso.
3. Envie qualquer material referente a esta lista de exercícios para o endereço eletrônico `esub.para.loureiro@gmail.com` tendo como assunto `[PAA 2011/1 LE2: "seu nome completo"]` e como anexo um arquivo zip, descrito abaixo, com o nome `LE2_"SeuNomeCompleto".zip` onde o string `"SeuNomeCompleto"` é o seu nome completo sem espaços em branco.

   Exemplo para o aluno Zoroastro Felizardo e Sortudo:

   - Assunto: `[PAA 2011/1 LE2: Zoroastro Felizardo e Sortudo]`
   - Arquivo zip: `LE2_ZoroastroFelizardoESortudo.zip`
4. O seu programa deve ser executado em alguma máquina do ambiente computacional do Departamento de Ciência da Computação da UFMG, onde os monitores irão avaliá-lo. No arquivo `leiame.txt`, a ser incluído no arquivo zip, você deve dizer qual é o ambiente computacional para executar o seu TP bem como todas as instruções necessárias.
5. Linguagem do exercício de programação: C, C++ ou Java.
6. As questões, a seguir, tratam do projeto de algoritmos. CLSR é a referência do exercício no livro-texto.
7. Das primeiras 16 questões, escolha sete para resolver, faça a questão 17 e o exercício de programação.

---

## 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]**

In order to transform one source string of text $x[1 \dots m]$ to a target string $y[1 \dots n]$, 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 | $x$ | $z$ |
|---|---|---|
| *initial strings* | a`lgorithm` | _ |
| copy | a`lgorithm` | a_ |
| copy | al`gorithm` | al_ |
| replace by `t` | alg`orithm` | alt_ |
| delete | alg`orithm` | alt_ |
| copy | algo`rithm` | altr_ |
| insert `u` | algo`rithm` | altru_ |
| insert `i` | algor`ithm` | altrui_ |
| insert `s` | algor`ithm` | altruis_ |
| twiddle | algori`thm` | altruisti_ |
| insert `c` | algori`thm` | altruistic_ |
| kill | algorithm_ | 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

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

(a) Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$ 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 $x[1 \ldots m]$ to $y[1 \ldots n]$ 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 $x =$ `GATCGGCAT` and $y =$ `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]**

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.

**Questão 17**

Para cada um dos seguintes paradigmas de projeto de algoritmos, formule cinco questões que tenham como resultado "verdadeiro" ou "falso" (similar às questões da primeira prova) e apresente as soluções justificadas. Procure incluir questões que também tenham um trecho de código, preferencialmente no formato apresentado no livro "Introduction to Algorithms".

(a) Indução
(b) Recursividade
(c) Tentativa e erro
(d) Divisão e conquista
(e) Balanceamento
(f) Programação dinâmica
(g) Algoritmos gulosos
(h) Algoritmos aproximados

---

Para o problema abaixo, escreva o algoritmo, faça a implementação, testes e apresente o custo de complexidade identificando a operação considerada relevante. No caso de apresentar uma solução recursiva, discuta também e apresente a complexidade para o crescimento da pilha.

**Exercício de Programação: 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

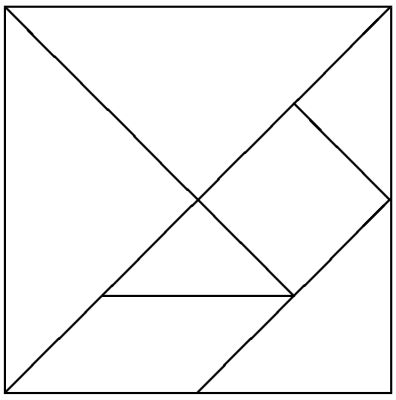

Figura 1: As sete peças do Tangram.

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

- 5 triângulos:
  - 2 pequenos (cada um com hipotenusa de $\frac{1}{2}$ e lados de $\frac{1}{2\sqrt{2}}$ e área de $\frac{1}{16}$);
  - 1 médio (hipotenusa de $\frac{1}{\sqrt{2}}$, lados de $\frac{1}{2}$ e área de $\frac{1}{8}$);
  - 2 grandes (hipotenusa de 1, lados de $\frac{1}{\sqrt{2}}$ e área de $\frac{1}{4}$);
- 1 quadrado (lado de $\frac{1}{2\sqrt{2}}$ e área de $\frac{1}{8}$);
- 1 paralelogramo (lados de $\frac{1}{2}$ e de $\frac{1}{\sqrt{2}}$, ângulos de 45° e 135° e área de $\frac{1}{8}$).

Este jogo pode ser resolvido computacionalmente e existem várias heurísticas para resolvê-lo. Ou seja, **existem diferentes estratégias de projeto de algoritmos** para resolver esse problema.

Neste exercício de programação, você deve identificar uma técnica de projeto de algoritmos usada para resolver o tangram e discutir essa solução. Preferencialmente, essa discussão deve fazer referências a outras técnicas de projeto para resolver este exercício de programação.

Os formatos de entrada e saída a serem usados neste exercício de programação estão descritos a seguir.

**Entrada:** A silhueta a ser identificada deve ter a seguinte entrada:

- Primeira linha: unidade da escala usada para fazer referência aos pontos da figura. As posições de todos os pontos de entrada e de saída serão dados nessa escala. Esse valor deve ser maior ou igual a 1. A sugestão é usar um valor que gere o menor erro possível na identificação da silhueta e da saída. Essa escala pode ser um número inteiro ou real.
- Próximas linhas: A silhueta da figura. Serão dadas as coordenadas cartesianas $(x, y)$ dos pontos extremos da silhueta da figura na escala previamente definida. Sendo assim, os segmentos de reta que ligam esses pontos consecutivamente e o último ponto ao primeiro definem a silhueta da figura a ser investigada.

**Saída:** Composta de duas partes: arquivo texto e um arquivo que exibe a entrada (apenas a silhueta) e a saída (posição de cada peça) em algum formato como postscript ou pdf.

O arquivo texto de saída deve possuir as posições $(x, y)$ de cada ponto 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 de acordo com a seguinte ordem:

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

**Exemplo:** Considere a figura 1. Nos arquivos abaixo, os comentários são usados para explicar cada linha e não fazem parte do arquivo.

Um exemplo de entrada é:

```
100                                                    % Escala
  0    0                                               % 1o. ponto
100    0                                               % 2o. ponto
100  100                                               % 3o. ponto
  0  100                                               % 4o. ponto
```

Note que as coordenadas estão apresentadas no sentido anti-horário.

Um arquivo exemplo de saída é:

```
 25   25   75   25   50   50                    % Triângulo menor
100   50  100  100   75   75                    % Triângulo menor
 50    0  100    0  100   50                    % Triângulo médio
  0    0   50   50    0  100                    % Triângulo grande
 50   50  100  100    0  100                    % Triângulo grande
 75   25  100   50   75   75   50   50         % Quadrado
  0    0   50    0   75   25   25   25         % Paralelogramo
```

Para cada peça, as coordenadas estão apresentadas no sentido anti-horário.

**Arquivos:** Os nomes dos arquivos devem seguir a seguinte regra:

1. Arquivo de entrada: `tangram[n].in`;
2. Arquivos de saída:
   - `tangram[n].out`;
   - `tangram[n].{ps|pdf}`;

onde `[n]` é um número inteiro positivo que identifica a configuração do tangram e `{ps|pdf}` identifica a saída gerada no formato postscript ou pdf, respectivamente.

Tomando novamente o exemplo da figura 1, teríamos os seguintes arquivos:

`tangram1.in`

```
100
  0    0
100    0
100  100
  0  100
```

`tangram1.out`

```
 25   25   75   25   50   50
100   50  100  100   75   75
 50    0  100    0  100   50
  0    0   50   50    0  100
 50   50  100  100    0  100
 75   25  100   50   75   75   50   50
  0    0   50    0   75   25   25   25
```

`tangram1.pdf`