



Lista 2: Recorrências e corretude de algoritmos recursivos

INSTRUÇÕES IMPORTANTES

(1) Em qualquer exercício que peça para você fornecer um algoritmo/solução para um problema, a menos que explicitamente dito o contrário, você deve:

- descrever em palavras qual é a ideia do seu algoritmo (**obrigatório**);
- escrever um pseudocódigo (opcional, se a descrição do item anterior ficou clara o suficiente);
- provar ou fornecer um argumento intuitivo para sua corretude (**obrigatório**). É claro que, se o exercício explicitamente pede uma prova, então um argumento intuitivo não será o suficiente;
- analisar o tempo do seu algoritmo (**obrigatório**).

O não cumprimento dos itens acima implica que o exercício está incorreto e o mesmo será desconsiderado.

(2) Você pode utilizar qualquer algoritmo visto em aula sem reescrevê-lo ou provar sua corretude novamente, mesmo que o algoritmo necessite de alguma pequena alteração.

- Descrever clara e sucintamente o que o algoritmo recebe, o que ele devolve e qual o seu consumo de tempo.
- Se houver alterações, descreva-as, indicando, por exemplo, quais linhas estão sendo alteradas/acrescentadas.

1. Seja F_n o n -ésimo número da sequência de Fibonacci. Temos $F_1 = 1$, $F_2 = 1$ e $F_n = F_{n-1} + F_{n-2}$ para $n \geq 3$. Use indução e prove que $F_n \geq 2^{0.5n}$ para todo $n \geq 6$. Qual o significado desse resultado?
2. Considere os seguintes algoritmos recursivos que resolvem um mesmo problema em uma entrada de tamanho n :

Algoritmo A: Divide o problema em 2 partes de tamanho $n/4$ cada, resolve-as e gasta tempo adicional n .

Algoritmo B: Divide o problema em 9 partes de tamanho $n/3$ cada, resolve-as e gasta tempo adicional n .

Algoritmo C: Divide o problema em 6 partes de tamanho $n/5$ cada, resolve-as e gasta tempo adicional \sqrt{n} .

No que segue, assuma $T(1) = 1$. Para cada um dos três algoritmos, monte uma recursão para seu tempo de execução $T(n)$ e a resolva com o melhor limitante possível (notação O ou Θ) usando:

- (a) **Algoritmo A:** Método de substituição.
- (b) **Algoritmo B:** Método de iteração.
- (c) **Algoritmo C:** Método mestre.

Qual algoritmo é mais eficiente?

3. Suponha que $T(1) = 1$ e $T(2) = 1$. Resolva as seguintes recorrências com notação O com o resultado mais justo possível (quando não especificado, use o método que preferir):

(a) $T(n) = T(\frac{n}{3}) + n^2$ (método de iteração)

(b) $T(n) = 2T(n-1) + n$ (método de iteração)

(c) $T(n) = 4T(\frac{n}{2}) + n$ (método de substituição, suponha $T(n) = \Theta(n^2)$)

(d) $T(n) = T(n-1) + T(n-2) + 3$ (método de substituição, suponha $T(n) = \Theta(2^n)$)

(e) $T(n) = 4T(\frac{n}{2}) + \sqrt{n}$ (árvore de recursão e método mestre)

(f) $T(n) = 7T(\frac{n}{3}) + n^2$ (árvore de recursão e método de substituição)

(g) $T(n) = 2T(\frac{n}{4}) + 1$ (método mestre)

(h) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$ (método mestre)

(i) $T(n) = 2T(\frac{n}{4}) + n$ (método mestre)

(j) $T(n) = 2T(\frac{n}{4}) + n^2$ (método mestre)

(k) $T(n) = T(\sqrt{n}) + 1$ (faça uma mudança de variáveis, usando $m = \log n$, escreva uma recorrência equivalente e resolva-a)

(l) $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n$

(m) $T(n) = 2T(\frac{3n}{4}) + n$

(n) $T(n) = T(\frac{2n}{3}) + 2T(\frac{n}{3}) + 1$ (método de substituição)

(o) $T(n) = 64T(\frac{n}{8}) + 7n^3$

(p) $T(n) = 4T(\frac{n}{8}) + \sqrt{n}$

(q) $T(n) = 2T(\frac{n}{2}) + n \log n$ (chute: $T(n) = \Theta(n \log^2 n)$)

$$(r) T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + T\left(\frac{n}{9}\right) + n$$

4. É fácil provar que $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ é $\Theta(n \log n)$ sempre que n é potência de 2, por exemplo usando árvore de recursão. Suponha agora que $n \geq 3$ não é potência de 2. Prove que, ainda assim, $T(n) = \Theta(n \log n)$. *Dica:* se n não é potência de 2, então existe um inteiro $k \geq 2$ tal que $2^{k-1} < n < 2^k$, e podemos assumir $T(2^{k-1}) \leq T(n) \leq T(2^k)$.

5. Um algoritmo A tem o seu consumo de tempo descrito através da recorrência

$$T(n) = 7T(n/2) + n^2.$$

Um outro algoritmo B tem o seu consumo de tempo descrito através da recorrência

$$T'(n) = aT'(n/4) + n^2,$$

onde a é uma constante. Qual é o maior valor inteiro possível para a de tal forma que o algoritmo B seja assintoticamente mais eficiente que o algoritmo A ? Justifique a sua resposta.

6. O *Quicksort* é outro algoritmo de divisão e conquista para o problema da ordenação. Ele é muito utilizado na prática, pois seu tempo esperado de execução é $\Theta(n \log n)$ e as constantes escondidas pela notação assintóticas são bem pequenas.

Seja $A[1..n]$ um vetor com n elementos. Dizemos que A está *particionado* com relação a um elemento, chamado *pivô*, se os elementos que são menores do que o pivô estão à esquerda dele e os outros elementos (maiores ou iguais) estão à direita dele. Note que o pivô está em sua posição correta final com relação ao vetor ordenado. A ideia do *Quicksort* é particionar o vetor e recursivamente ordenar as partes à direita e à esquerda do pivô, desconsiderando-o. Seu pseudocódigo é dado no Algoritmo 1.

Algorithm 1 Algoritmo *Quicksort* para ordenação.

```

1: Função QUICKSORT( $A$ ,  $inicio$ ,  $fim$ )
2:   Se  $inicio < fim$  então
3:      $p =$  ESCOLHEPIVO( $A$ ,  $inicio$ ,  $fim$ )
4:     troque  $A[p]$  com  $A[fim]$ 
5:      $x =$  PARTICIONA( $A$ ,  $inicio$ ,  $fim$ )
6:     QUICKSORT( $A$ ,  $inicio$ ,  $x - 1$ )
7:     QUICKSORT( $A$ ,  $x + 1$ ,  $fim$ )

```

Considere que a função ESCOLHEPIVO sempre devolve a posição fim , executando, portanto, em tempo $\Theta(1)$. A função PARTICIONA recebe o vetor A e as posições $inicio$ e fim e devolve a posição final do pivô após a partição. Ela executa em tempo $\Theta(n)$. Responda:

- (a) Por que o tempo de execução de QUICKSORT no pior caso é $\Theta(n^2)$?
 (b) Assumindo que PARTICIONA está correto, prove que QUICKSORT está correto.

7. No ensino fundamental, aprendemos um algoritmo para multiplicar dois números “grandes”. Esse algoritmo tem tempo de execução $O(n^2)$ quando os números têm n dígitos¹.

¹É um bom exercício entender por que isso vale.

O algoritmo de Karatsuba é um algoritmo de divisão e conquista que promete multiplicar dois números inteiros x e y que possuem n dígitos de uma forma mais eficiente. Sua ideia principal baseia-se em reescrever um número em partes menores com aproximadamente metade do número de dígitos cada. Por exemplo, 3147869 pode ser escrito como $10^4 \cdot 314 + 7869$.

O algoritmo é apresentado a seguir. Ele usa a função $\text{IGUALATAM}(x, y)$, que deixa os números x e y com o mesmo número de dígitos (igual ao número de dígitos do maior deles) colocando zeros à esquerda se necessário. Ela devolve o número de dígitos (agora igual) desses números.

```

1: Função KARATSUBA( $x, y$ )
2:    $n = \text{IGUALATAM}(x, y)$ 
3:   Se  $n \leq 2$  então
4:     Devolve  $xy$ 
5:   seja  $x = 10^{\lceil n/2 \rceil}a + b$  e  $y = 10^{\lceil n/2 \rceil}c + d$ , onde  $a$  e  $c$  têm  $\lfloor \frac{n}{2} \rfloor$  dígitos cada e  $b$  e  $d$ 
   têm  $\lceil \frac{n}{2} \rceil$  dígitos cada
6:    $p_1 = \text{KARATSUBA}(a, c)$ 
7:    $p_2 = \text{KARATSUBA}(b, d)$ 
8:    $p_3 = \text{KARATSUBA}(a + b, c + d)$ 
9:   Devolve  $10^{2\lceil n/2 \rceil}p_1 + 10^{\lceil n/2 \rceil}(p_3 - p_1 - p_2) + p_2$ 

```

Prove a corretude e analise seu tempo de execução do algoritmo de Karatsuba. *Dica:* primeiro, imagine como os números estão sendo representados (afinal, se fossem duas variáveis inteiras bastaria fazer $x \times y$ para resolver o problema).

8. A ordenação por inserção pode ser expressa sob a forma de um procedimento recursivo como a seguir. Para ordenar $A[1..n]$, ordenamos recursivamente $A[1..n - 1]$ e depois inserimos $A[n]$ no arranjo ordenado $A[1..n - 1]$. Escreva o pseudocódigo desse algoritmo, uma recorrência para seu tempo de execução e resolva a recorrência pelo método de substituição.
9. Suponha que um vetor (não necessariamente ordenado) $A[1..n]$ contém todos os números em $\{1, 2, \dots, n - 1\}$, ou seja, um dos números está aparecendo duas vezes. Dê um algoritmo de divisão e conquista que determina qual é o número que ocorre duas vezes em A .

1 Questões retiradas do Enade e Poscomp

QUESTÃO DISCURSIVA 03

Listas lineares armazenam uma coleção de elementos. A seguir, é apresentada a declaração de uma lista simplesmente encadeada.

```
struct ListaEncadeada {
    int dado;
    struct ListaEncadeada *proximo;
};
```

Para imprimir os seus elementos da cauda para a cabeça (do final para o início) de forma eficiente, um algoritmo pode ser escrito da seguinte forma:

```
void mostrar(struct ListaEncadeada *lista) {
    if (lista != NULL) {
        mostrar(lista->proximo);
        printf("%d ", lista->dado);
    }
}
```

Com base no algoritmo apresentado, faça o que se pede nos itens a seguir.

- a) Apresente a classe de complexidade do algoritmo, usando a notação Θ . (valor: 3,0 pontos)
- b) Considerando que já existe implementada uma estrutura de dados do tipo pilha de inteiros — com as operações de empilhar, desempilhar e verificar pilha vazia — reescreva o algoritmo de forma não recursiva, mantendo a mesma complexidade. Seu algoritmo pode ser escrito em português estruturado ou em alguma linguagem de programação, como C, Java ou Pascal. (valor: 7,0 pontos)

QUESTÃO 25

A sequência de Fibonacci é uma sequência de números inteiros que começa em 1, a que se segue 1, e na qual cada elemento subsequente é a soma dos dois elementos anteriores. A função `fib` a seguir calcula o n -ésimo elemento da sequência de Fibonacci:

```
unsigned int fib (unsigned int n)
{
    if (n < 2)
        return 1;
    return fib(n - 2) + fib (n - 1);
}
```

Considerando a implementação acima, avalie as afirmações a seguir.

- I. A complexidade de tempo da função `fib` é exponencial no valor de n .
- II. A complexidade de espaço da função `fib` é exponencial no valor de n .
- III. É possível implementar uma versão iterativa da função `fib` com complexidade de tempo linear no valor de n e complexidade de espaço constante.

É correto o que se afirma em

- A** I, apenas.
- B** II, apenas.
- C** I e III, apenas.
- D** II e III, apenas.
- E** I, II e III.

QUESTÃO 33

Considere a função recursiva F a seguir, que em sua execução chama a função G :

```
1 void F(int n) {
2     if(n > 0) {
3         for(int i = 0; i < n; i++) {
4             G(i);
5         }
6         F(n/2);
7     }
8 }
```

Com base nos conceitos de teoria da complexidade, avalie as afirmações a seguir.

- I. A equação de recorrência que define a complexidade da função F é a mesma do algoritmo clássico de ordenação *mergesort*.
- II. O número de chamadas recursivas da função F é $\Theta(\log n)$.
- III. O número de vezes que a função G da linha 4 é chamada é $O(n \log n)$.

É correto o que se afirma em

- A I, apenas.
- B II, apenas.
- C I e III, apenas.
- D II e III, apenas.
- E I, II e III.

QUESTÃO 21 – Suponha que, ao invés de dividir em duas partes, foi criada uma versão do mergesort que divida a entrada em quatro partes, ordene cada quarta-parte, e, finalmente, combine essas quatro partes usando um procedimento $O(n)$. A equação de recorrência que descreve o tempo de execução desse algoritmo é:

- A) $T(n) = 4 \cdot T(n/4) + O(n)$
- B) $T(n) = 4 \cdot T(n/2) + 2 \cdot O(n)$
- C) $T(n) = T(n/4) + 4 \cdot O(n)$
- D) $T(n) = 4 \cdot T(n/4) + 4 \cdot O(n)$
- E) $T(n) = T(n/4) + O(n)$

QUESTÃO 22 – A complexidade de tempo da questão 21 é:

- A) $O(n^2)$
- B) $O(n^4)$
- C) $O(4 \cdot n)$
- D) $O(n \log n)$
- E) $O(n)$

QUESTÃO 21 – Dadas as seguintes relações de recorrência:

- I. $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- II. $T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$
- III. $T(n) = T\left(\frac{n}{2}\right) + O(1)$

As relações de recorrência I, II, e III pertencem, nessa ordem, às classes de complexidade:

- A) $\theta(n^2)$, $\theta(n^3)$, e $\theta(n)$
- B) $\theta(n)$, $\theta(n^2)$, e $\theta(n^3)$
- C) $\theta(n \log n)$, $\theta(n^3)$, e $\theta(\log n)$
- D) $\theta(\log n)$, $\theta(n \log n)$, e $\theta(n^3)$
- E) $\theta(n^2)$, $\theta(n^2)$, e $\theta(n^2)$

QUESTÃO 21 – Considere os seguintes algoritmos recursivos que resolvem o mesmo problema em uma entrada de tamanho n :

Algoritmo 1: Divide o problema em 3 partes de tamanho $n/4$ cada e gasta um tempo adicional $O(1)$ por chamada.

Algoritmo 2: Divide o problema em 3 partes de tamanho $n/2$ cada e gasta um tempo adicional $O(n^2)$ por chamada.

Algoritmo 3: Divide o problema em 3 partes de tamanho $n/3$ cada e gasta um tempo adicional de $O(n)$ por chamada.

A complexidade dos algoritmos 1, 2 e 3 é, respectivamente:

- A) $\theta(n^{\log_4 3})$, $\theta(n^2)$, $\theta(n \log n)$
- B) $\theta\left(\frac{n}{4}\right)$, $\theta\left(\frac{n}{2}\right)$, $\theta\left(\frac{n}{3}\right)$
- C) $\theta(1)$, $\theta(n^2)$, $\theta(n)$
- D) $\theta(n^4)$, $\theta(n^2)$, $\theta(n^3)$
- E) $\theta(n^{\log_4 3})$, $\theta(n^{\log_2 3})$, $\theta(n^{\log_3 3})$