**1ª Lista de Exercícios de Introdução à Análise de Algoritmos**

**Prof. Glauber Cintra**

**Equipe:** Francisco Lucas Lima da Silva

Nicolas Holanda de Menezes

José Mateus Azevedo de Sousa

Victor de Sousa Firmino

1. **(0,5 pontos)** Numere as funções abaixo a partir do 1 em ordem **estritamente** crescente de dominação assintótica. Se f e g são tais que f ∈ o(g) então f deve ter número menor do que g. Se f ∈ Θ(g) então f e g devem ter o mesmo número.

( 7 ) n! → nn ( 1 ) log n ( 5 ) 3n ( 1 ) 2log n3 → log(n) ( 5 ) 3n+1 → 3n

( 6 ) fib(n) ( 4 ) n3 ( 2 ) nlog n ( 3 ) 6n2 → n2 ( 3 ) 2n + n2 → n2

n! = n\*(n – 1)\*(n – 2)\*...\*2\*1 → (prop. Aditiva) → nn

2log(n3) = 6log(n) → (prop. Multiplicativa) → log(n)

3n+1 = 3\*3n → (prop. Multiplicativa) → 3n

6n2 → (prop. Multiplicativa) → n2

2n + n2 → (prop. Aditiva) → n2

fib(n) = → (prop. Aditiva e Multiplicativa) →

Ordem crescente de crescimento: Constantes, Logaritmicas, Lineares, Linearitmicas, Polinomiais, Exponenciais

1. **(0,5 pontos)** Explique o significado dos termos *algoritmo*, *algoritmo computacional,* *algoritmo correto,* *algoritmo eficiente* e *tamanho da entrada de um algoritmo*.

**Algoritmo:** sequência de instruções que visam resolver instâncias um problema;

**Algoritmo Computacional:** algoritmo constituído apenas de instruções bem definidas, não ambíguas;

**Algoritmo Correto:** algoritmo que resolve corretamente todas as instancias de um problema para o qual foi desenvolvido;

**Algoritmo Eficiente:** algoritmo que requer que seja executada uma quantidade de instruções elementares limitadas por um polinômio do tamanho da entrada;

**Tamanho da Entrada de um Algoritmo:** quantidade de bits para representar os dados de uma entrada.

1. **(1,5 pontos)** Indique quais são o melhor caso e o pior caso do algoritmo abaixo e sua região crítica. Qual a complexidade temporal desse algoritmo no melhor e no pior caso? Qual a complexidade espacial desse algoritmo? O algoritmo é eficiente? É de cota inferior? Justifique suas respostas. Finalmente, prove que esse algoritmo é correto.

Algoritmo Contido

Entrada: os vetores A e B com m e n posições, respectivamente

Saída: *Sim*, se todos os elementos de A estão contidos em B; *Não*, caso contrário

para i = 0 até m – 1

j = 0

enquanto j < n e A[i] ≠ B[j]

j++

se j >= n

devolva *Não* e pare

devolva *Sim*

- Melhor Caso: é quando o primeiro elemento de A não estiver em B, ou seja, para todo elemento contido no vetor B,

A[0] ≠ B[j], sendo j = 0, 1, ..., n-1.

- Pior Caso: é quando o vetor A estiver contido em B, ou seja, todos os elementos de A ocorrem também em B.

- Complexidade Temporal: no melhor caso, será verificado se o primeiro elemento de A não ocorre em B. Para isso, o algoritmo percorrerá o vetor B para saber se há o elemento no vetor. Por isso, o tempo requerido para este caso é θ(n). Já no pior caso, deve-se verificar se todos os elementos de A ocorrem em B, logo, para cada elemento de A, será percorrido o vetor B para saber se o elemento ocorre no vetor. Logo, o tempo requerido para o pior caso é θ(nm).

- Região Critica: é a condição do Enquanto (enquanto j < n e A[i] ≠ B[j]), porque será verificado se tal elemento de A ocorre em B.

- Complexidade Espacial: o espaço requerido pelo algoritmo é O(1), pois há somente declarações de variáveis para os laços.

- O algoritmo não é eficiente, pois o tempo requerido não é delimitado por um polinômio do tamanho da entrada.

- O algoritmo é de cota inferior, pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(nm) e o algoritmo em questão requer tempo O(nm).

- Prova de Corretude:

Teorema: o algoritmo contido é correto

Prova: Suponha que um elemento *x* do vetor A não ocorra no vetor B. Logo, a 2ª condição do Enquanto (A[i] ≠ B[j]) sempre será satisfeita e, após percorrer o vetor B, j = n, e assim, a condição do Se será satisfeita e o algoritmo devolverá *Não*, o que é correto. Suponha agora que todos os elementos de A ocorram em B. Seja *k* a posição de um elemento *x* em A e *l* a posição desse elemento em B. Nota-se que *0 ≤ k ≤ m – 1* e 0 *≤ l ≤ n – 1*. Quando *i = k* e *j = l*, a 2ª condição do Enquanto não será satisfeita, consequentemente a condição do Se também é falsa. Como todos elementos de A ocorrem em B, isso sempre ocorrerá. No final do laço externo, o algoritmo devolverá *Sim*, o que é correto.

1. **(2 pontos)** Escreva um algoritmo que receba um vetor de números (e, naturalmente, seu tamanho) e devolva *Sim* se existir um número par no vetor; *Não*, caso contrário. Caracterize o pior e o melhor caso do seu algoritmo e determine a complexidade temporal no pior e no melhor caso. Determine também a complexidade espacial do algoritmo. O seu algoritmo é eficiente? É de cota inferior? Prove que seu algoritmo é correto.

**Algoritmo temPar**

Entrada: um vetor V com n números

Saída: *Sim*, se existir um número par no vetor; *Não*, caso contrário.

Para i = 0 até n-1

Se V[i] é par

Devolva *Sim* e pare

Devolva Não

- Melhor Caso: quando o primeiro elemento do vetor for par, logo, a condição será satisfeita.

- Pior Caso: quando não há números pares no vetor, assim o algoritmo percorrerá todo o vetor

- Região Critica: é a condição do Se, pois ali verifica se tal número é par.

- Complexidade Temporal: no melhor caso, logo na primeira iteração do laço, a condição do Se será verdadeira, logo o tempo requerido para este caso é O(1). Já no pior caso, o vetor V será todo percorrido e a condição do Se nunca será satisfeita. Por isso, o tempo requerido no pior caso é θ(n).

- Complexidade Espacial: o espaço requerido é O(1), pois há somente declarações de variáveis para os laços.

- O algoritmo é eficiente, pois o tempo requerido pelo algoritmo é (O(n)) e a entrada tem tamanho n, logo, o tempo é delimitado por um polinômio do tamanho da entrada.

- O algoritmo é de cota inferior, pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(n) e o algoritmo em questão requer tempo O(n).

- Prova de Corretude:

Teorema: o algoritmo temPar é correto,

Prova: Suponha que não haja nenhum elemento par no vetor A. Logo percebemos que a condição do Se nunca será satisfeita e, no final da iteração, o algoritmo devolverá *Não*, o que é correto. Suponha agora que haja um elemento par no vetor A. Seja *k* a posição desse elemento no vetor. Nota-se que *0 ≤ k ≤ n – 1*. Quando *i = k*, a condição do Se será satisfeita, e o algoritmo devolverá *Sim*, o que é correto.

1. **(1 ponto)** Resolva as seguintes fórmulas de recorrência:
   1. T(n) = 2T(⎣n/2⎦) + n, T(1) = 1

Fazendo n = 2k:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Eq. 0 |  |  |
|  | Eq. 1 |  |  |
|  | Eq. 2 |  |  |
|  | ... | ... | ... |
|  | Eq. k-1 |  |  |
|  | Eq. k |  |  |

-------------------------------------------------------

.

Como , então:

. Usando notação assintótica, verifica-se que .

* 1. T(n) = T(n – 1) + n/2, T(1) = ½

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  | ... | ... |
|  |  |  |
|  |  |  |

---------------------------------------------------

.

Usando notação assintótica, verifica-se que .

1. **(1,5 pontos)** Considere o algoritmo *enigma* descrito a seguir:

Algoritmo enigma

Entrada: um vetor V e a posição n

se (n = 0)

devolva V[n]

se não

devolva V[n] \* enigma(V, n - 1)

Seja L = [8, 3, 5, 2, 9]. Simule o cálculo de *enigma*(L, 4). Determine a complexidade temporal e espacial do algoritmo *enigma* (exiba os cálculos realizados para determinar tais complexidades). Para que serve esse algoritmo? Ele é eficiente? Prove que o algoritmo é correto.

1. **(1 ponto)** Escreva um algoritmo **recursivo** que receba um número *a* e um número natural *b* e devolva *ab*. Prove que seu algoritmo é correto e determine a complexidade temporal e espacial do algoritmo (bônus de **0,5 pontos** se o algoritmo for eficiente).

**Algoritmo: potencia**

Entrada:

Saída: ab

aux =

Se b é par

Devolva potencia(a, aux) \* potencia(a, aux)

Senão

Devolva potencia(a, aux) \* potencia(a, aux) \* a

1. **(0,5 pontos)** Cite o nome de um algoritmo de *cota superior* para o *problema de encontrar uma Trilha de Euler* (*Eulerian Trail*). Este algoritmo também é de cota inferior? Justifique.
2. **(1 ponto)** Escreva um algoritmo de *cota inferior* que receba uma matriz de número com *n* linhas e *m* colunas e devolva a soma dos números contidos na matriz. Determine a complexidade temporal e espacial do algoritmo e justifique porque ele é um algoritmo de cota inferior.

**Algoritmo: somaMatriz**

Entrada: uma matriz A com n linhas e n colunas

Saída: a soma dos elementos da matriz

soma = 0

Para i = 0 até n – 1

Para j = 0 até m – 1

soma = soma + M[i, j]

Devolva soma

- Complexidade Temporal: o algoritmo percorre as n linhas e m colunas da matriz, logo a complexidade temporal é O(nm).

- Complexidade Espacial: o espaço requerido é O(1), pois há somente declarações de variáveis para os laços e uma variável usada para guardar a soma dos elementos dos vetores.

- O algoritmo é de cota inferior pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(nm) e o algoritmo em questão requer tempo O(nm).

1. **(2 pontos)** Podemos implementar uma fila *F* usando duas pilhas *P1* e *P2*. A operação *insereNaFila(x, F)* é feita empilhando-se *x* em *P1*. A operação *removeDaFila(F)* é feita como indicado abaixo:

Algoritmo removeDaFila

Entrada: uma fila F implementada através das pilhas P1 e P2

Saída: remove um elemento de F

Se P2 for vazia

Enquanto P1 não for vazia

empilha(desempilha(P1), P2)

devolva desempilha(P2) // pode haver um erro (underflow) se P2 for vazia

Utilizando dois dos métodos de análise amortizada que estudamos, mostre que o custo de uma sequência de *n* operações *insereNaFila* e *removeDaFila* é Θ *(n)* e que o *custo amortizado* da operação *removeDaFila* é *O(1)*, considerando que inicialmente a fila está vazia.