Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estruturas de Dados
1º Semestre de 2019
Profa. Raquel Prates
Monitor: Matheus Nunes

28/03/2019

Exercícios de Revisão

1. Sejam f(n), g(n) duas funções assintóticamente positivas. Prove que as afirmativas abaixo são verdadeiras ou falsas, usando para isso as definições das notações assintóticas ou contraexemplos.

a) $2^{n+1} = O(2^n)$

Verdadeiro:
Existe c tal que:

$2^{n+1} <= c2^n$

$2.2^n <= c2^n$

C >= 2

b) $2^{2n} = O(2^n)$

Falso:
Não existe c tal que:

$2^{2n} <= c2^n$

$2^n.2^n <= c2^n$

$C >= 2^n$ -> Não existe constante

c) f(n) + g(n) = O (Max(f(n), g(n))

Verdadeiro:
Considere h(n) = Max(f(n), g(n)) ou seja h(n) = f(n) se f(n) >= g(n) ou h(n) = f(n) se f(n) < g(n)
Queremos provar que: f(n) + g(n) <= c . h(n)
Pela definição de h temos que h(n) >= f(n) e h(n) >= g(n) é sempre verdadeiro.
Logo somando os dois termos temos:
h(n) >= f(n)
h(n) >= g(n)

2h(n) >= f(n) + g(n). Logo c = 2 satisfaz a equação.

d) A notação θ é simétrica, ou seja, f(n) = θ (g(n)) se e somente se g(n) = θ (f(n))

Se f(n) = θ (g(n)) então existem constantes c1 e c2 tais que:
c1g(n) <= f(n) <= c2g(n)
c1g(n) <= f(n) -> g(n) <= (1/c1)f(n)

f(n) <= c2g(n) -> (1/c2)f(n) <= g(n).
Logo:
(1/c2)f(n) <= g(n) <= (1/c1)f(n)
g(n) = θ (f(n))
Mesmo raciocínio para provar o outro lado do *se e somente se*

2. Resolva a seguinte questão sobre recursividade:
   a. Escreva uma **função recursiva** `int Palindromo(int esq, int dir, char palavra[])` que testa se uma determinada palavra é um palíndromo e retorna 1 em caso positivo e 0 em caso negativo. Um palíndromo é uma palavra que é lida da mesma forma da esquerda para direita ou da direita para esquerda (ex. ovo, arara). A palavra é passada para o função através de um vetor de caracteres limitada pelos os índices `esq` e `dir`, por exemplo: `Palindromo(0,4,"arara")`

```
int Palindromo(int esq, int dir, char palavra[])
{
if (dir <= esq)
return 1;
else if(palavra[esq] != palavra[dir])
return 0;
else
return Palindromo(esq+1,dir-1,palavra);
}
```

   b. Calcule qual é a **função de complexidade** para o número de comparações de caracteres da sua função no melhor caso e no pior caso. Para isso, **determine e resolva** a equação de recorrência dessa função recursiva. Qual é a **ordem de complexidade** de sua função?

No melhor caso, a primeira comparação é falsa e a o função retorna. Logo T(n) = 1;

No pior caso, quando a palavra é um palíndromo, são feitas n/2 comparações (considere n par apara simplificar). Esse resultado é obtido resolvendo a seguinte equação de recorrência:

T(n) = 1 + T(n-2); se n>=2

T(n) = 0; se n<2

Fazendo a expansão de termos

T(n) = 1 + T(n-2)

T(n-2) = 1 + T(n-4)

...

T(2) = 1 + T(0)

T(0) = 0

Logo T(n) = 1 + 1 + 1 + ... + 1 (n/2 vezes) = 1 . n/2

c. Qual seria a complexidade de uma implementação não recursiva dessa mesma função? Qual das duas implementações vocês escolheria? Justifique.

A função não recursiva teria a mesma complexidade de tempo (n/2 comparações no pior caso). O custo de memória da implementação recursiva seria maior devido aos registros empilhados na pilha de ativação a cada chamada recursiva. Dessa forma, é melhor escolher a versão não recursiva.

3. Vários algoritmos em computação usam a técnica de "Dividir para Conquistar": basicamente eles fazem alguma operação sobre todos os dados, e depois dividem o problema em sub-problemas menores, repetindo a operação. Uma equação de recorrência típica para esse tipo de algoritmo é mostrada abaixo. Resolva essa equação de recorrência usando o Teorema Mestre.

```
T(n) = 2T(n/2) + n;

T(1) = 1;
```

Pelo teorema Mestre

a = 2

b = 2

f(n) = n

$n^{\log_b a} = n$

logo f(n) = θ ($n^{\log_b a}$) -> caso 2 do teorema mestre. Nesse caso, T(n) = θ ($n^{\log_b a}$logn) = θ (nlogn)