2ª Lista de Exercícios de Introdução à Análise de Algoritmos
Prof. Glauber Cintra – Entrega: 26/mar/2012

Esta lista deve ser feita por grupos de no **mínimo** 3 e no **máximo** 4 alunos.

Nomes: ____

____

____

____

1) **(2 pontos)** Resolva as seguintes fórmulas de recorrência:

a) T(n) = 2T(n – 1) + n, T(1) = 1

T(n) = 2T(n – 1) + n
2T(n – 1) = 4T(n – 2) + 2(n – 1)
4T(n – 2) = 8T(n – 3) + 4(n – 2)
$\ldots$
$2^{(n-2)}T(2) = 2^{(n-1)}T(1) + 2^{(n-2)}\cdot 2$ $(2^{(n-2)}T(n-(n-2)))$
$2^{(n-1)}T(1) = 2^{(n-1)}$

$T(n) = n + 2(n-1) + 4(n-2) + \ldots + 2^{(n-2)}\cdot 2 + 2^{(n-1)}\cdot 1$

$T(n) = \sum_{x=0}^{n-1} 2^x (n-x)$

$T(n) = 2^{(n+1)} - n - 2$

Logo $T(n) \in O(2^n)$.

b) T(n) = T(n/2) + n, T(1) = 1

Como $n = \Omega(n^{\log_2 1 + \varepsilon})$, para $\varepsilon > 1$, e $\frac{1}{2}\cdot n \leq c\cdot n$, para $c \geq \frac{1}{2}$, então pelo Teorema Mestre temos que $T(n) \in \Theta(n)$.

2) Considere o algoritmo *buscabinária* descrito a seguir:

```
Algoritmo buscabinária
     Entrada: um número x, um vetor v em ordem crescente e duas posições inicio e fim
     Saída: verdadeiro, se x ocorre entre as posições início e fim de v;
            falso, caso contrário
     se (inicio > fim)  devolva falso        /* devolva finaliza a execução do algoritmo */
     meio = (inicio + fim) / 2                /* divisão inteira */
     se (x = v[meio])  devolva verdadeiro
     se (x < v[meio])  devolva buscabinária(x, v, inicio, meio – 1)
     devolva buscabinária(x, v, meio + 1, fim)
```

a) **(0,2 pontos)** Seja L = {2, 5, 5, 7, 8, 9, 10, 12, 15, 16, 18, 20, 21}. Simule o cálculo de *buscabinária*(18, L, 0, 12), exibindo os parâmetros de entrada e o valor devolvido por cada chamada ao algoritmo *buscabinária*.

buscabinária(18, L, 0, 12) →
buscabinária(18, L, 7, 12) →
buscabinária(18, L, 10, 12) →
buscabinária(18, L, 10, 10) →
verdadeiro.

b) **(0,6 pontos)** Determine a complexidade de tempo e de espaço do algoritmo *buscabinária* (mostre os cálculos realizados para determinar tais complexidades). O algoritmo *buscabinária* é eficiente?

As principais variáveis de controle da recursão são os parâmetros início e fim. Se tomarmos $n = inicio - fim + 1$, temos a quantidade de elementos sobre os quais o algoritmo *buscabinária* executará.

Claramente vemos que uma chamada a *buscabinária* com n elementos leva um tempo igual a uma chamada a *buscabinária* com $\lfloor n/2 \rfloor$ elementos mais um tempo constante, caso $n \neq 0$ e o elemento pesquisado não esteja em L[$\lfloor n/2 \rfloor$] , casos esse em que as chamadas recursivas param.

Sendo assim temos a recursão (I), onde T(0) é uma das condições de parada.

$$\begin{cases} T(n) = T(\lfloor n/2 \rfloor) + c \\ T(0) = c \end{cases} \quad \text{(I)}$$

Como $c \in \Theta(n^{\log \frac{1}{2}})$ temos pelo Teorema Mestre que $T(n) \in \Theta(\log n)$.

O algoritmo é eficiente, pois relacionando log n com o tamanho da entrada temos que o algoritmo leva tempo linear.

c) **(1,2 pontos)** Prove que o algoritmo é correto.

Primeiramente notemos que uma chamada recursiva executa sobre um sub-vetor que tem aproximadamente metade do tamanho do vetor da chamada original, isso faz com que no pior caso (o número x não está em v) o algoritmo (e a indução abaixo) caminhe para o fim das chamadas recursivas devolvendo falso, esse sub-vetor tem os limites determinados pelas variáveis início e fim.

Suponha que tenhamos o número n (n = fim – início + 1) de elementos do vetor sobre o qual o algoritmo executará.

Agora façamos indução em n com base 0. Temos que

$$fim - início + 1 = 0$$
$$início = fim + 1$$
$$início > fim \quad ,$$

com isso o algoritmo retorna falso, o que é correto, pois um número x não pode existir em um vetor vazio.

Suponha que uma chamada a buscabinária(x, v, início, meio -1) retorna verdadeiro se x estiver entre o intervalo início e meio-1 do vetor v, retorna falso caso contrário, e uma chamada a buscabinária(x, v, meio+1, fim) retorna verdadeiro se x estiver entre o intervalo meio+1 e fim do vetor, retorna falso caso contrário. (H.I)

Em uma chamada a buscabinária(x, v, início, fim), com $n \neq 0$, se o número x estiver na posição v[meio] o algoritmo retorna verdadeiro, o que é correto. Caso contrário temos duas possibilidades: como temos um vetor ordenado se x for menor do que o elemento em v[meio] o número x só pode estar, se existir, no intervalo início e meio-1 logo é feita uma chamada a buscabinária(x, v, início, meio -1) , o que é correto, senão o número x só pode estar, se existir, no intervalo meio+1 e fim logo é feita uma chamada a buscabinária(x, v, meio+1, fim) , o que é correto.

3) **(2 pontos)** Escreva um algoritmo **recursivo** que receba um número a e um número natural b e devolva a^b. Prove que seu algoritmo é correto e determine a complexidade de tempo e de espaço do algoritmo. Você ganhará um bônus de **1 ponto** se o seu algoritmo for eficiente.

Algoritmo Potência
Entrada: Um número a e um numero natural b
Saída: a^b

Início:
    se b == 0
        devolva 1;
    senão
        aux = Potência(a, $\lfloor b/2 \rfloor$);
        se a % 2 == 0
            devolva aux*aux;
        senão
            devolva aux*aux*a;
Fim.

Teorema: O algoritmo Potência é correto.

Prova: Façamos indução em b com base 0. Trivial. Suponha agora de uma chamada a Potência(a, $\lfloor b/2 \rfloor$) devolve $a^{\lfloor b/2 \rfloor}$. Sendo assim temos que $aux = a^{\lfloor b/2 \rfloor}$.

Sendo b um número par ($\lfloor b/2 \rfloor = b/2$) temo que o algoritmo devolve $aux \cdot aux = a^{\frac{b}{2}} \cdot a^{\frac{b}{2}} = a^b$, o que é correto. Sendo b um número ímpar ($\lfloor b/2 \rfloor = (b-1)/2$) temos que o algoritmo devolve $aux \cdot aux \cdot a = a^{\frac{b-1}{2}} \cdot a^{\frac{b-1}{2}} \cdot a = a^{b-1} \cdot a = a^b$, o que é correto.

Complexidade Temporal e Espacial: Tanto a complexidade temporal e espacial do algoritmo potência respondem a mesma recorrência (II), podemos simplificar a recorrência a variável b, pois é ela que controla o fim da recursão.

$$\begin{cases} T(b) = T(\lfloor b/2 \rfloor) + c \\ T(0) = c \end{cases} \qquad \text{(II)}$$

Como $c = \Theta(b^{\log_2 1})$, então pelo Teorema Mestre temos que $T(b) \in \Theta(\log b)$.

4) **(1 ponto)** O *problema da mediana* consiste em, dada uma lista de números, determinar um número *x* tal que pelo menos metade dos números da lista seja menor ou igual a x e pelo menos metade dos números da lista seja maior ou igual a *x*. Uma forma de calcular a mediana é colocar a lista em ordem. Se a lista tiver uma quantidade ímpar de elementos, a mediana é o elemento central. Se a lista tiver uma quantidade par de elementos, a mediana é a média aritmética dos dois elementos centrais. Por exemplo, a mediana da lista (2, 4, 4, 5, 7) é 4 e a mediana da lista (2, 4, 4, 6, 7, 8) é 5. Pesquise e informe a cota inferior do problema da mediana.

O limite inferior é ligeiramente maior do que 2n, onde n é o número de elementos na lista.

5) **(1 ponto)** Pesquise e informe um algoritmo de *cota superior* para o problema de fluxos em redes.

O algoritmo "Highest Label Preflow-Push" possuí uma complexidade temporal de $O(V^2 \cdot \sqrt{E})$, sendo uma variação do algoritmo "Push-Relabel" que tem complexidade temporal $O(V^2 \cdot E)$. Temos uma implementação em Python o algoritmo "*relabel-to-front*" que é outra variação do "Push-Relabel" mas com complexidade temporal de $O(V^3)$.

```
def relabel_to_front(C, source, sink):
     n = len(C) # C is the capacity matrix
     F = [[0] * n for _ in xrange(n)]
     # residual capacity from u to v is C[u][v] - F[u][v]

     height = [0] * n # height of node
     excess = [0] * n # flow into node minus flow from node
     seen   = [0] * n # neighbours seen since last relabel
     # node "queue"
     list   = [i for i in xrange(n) if i != source and i != sink]
```

```
    def push(u, v):
        send = min(excess[u], C[u][v] - F[u][v])
        F[u][v] += send
        F[v][u] -= send
        excess[u] -= send
        excess[v] += send

    def relabel(u):
        # find smallest new height making a push possible,
        # if such a push is possible at all
        min_height = ∞
        for v in xrange(n):
            if C[u][v] - F[u][v] > 0:
                min_height = min(min_height, height[v])
                height[u] = min_height + 1

    def discharge(u):
        while excess[u] > 0:
            if seen[u] < n: # check next neighbour
                v = seen[u]
                if C[u][v] - F[u][v] > 0 and height[u] > height[v]:
                    push(u, v)
                else:
                    seen[u] += 1
            else: # we have checked all neighbours. must relabel
                relabel(u)
                seen[u] = 0

    height[source] = n   # longest path from source to sink is less than
n long
    excess[source] = Inf # send as much flow as possible to neighbours of
source
    for v in xrange(n):
        push(source, v)

    p = 0
    while p < len(list):
        u = list[p]
        old_height = height[u]
        discharge(u)
        if height[u] > old_height:
            list.insert(0, list.pop(p)) # move to front of list
            p = 0 # start from front of list
        else:
            p += 1

    return sum(F[source])
```

6) **(2 pontos)** Escreva um algoritmo de cota superior para o problema da soma de matrizes. Prove que seu algoritmo é correto e que é um algoritmo de cota inferior.

Algoritmo SomaMatrizes
Entrada: Matrizes A[m,n] e B[m,n]
Saída: Matriz C[m,n] contendo a soma de A e B
início:
    para i=0 até m-1
        para j=0 até n-1
            C[i, j] = A[i, j] + B[i, j];
    devolva C;
Fim.

Teorema: O algoritmo SomaMatrizes é correto.

Prova: Trivial. Claramente vemos que as matrizes A e B serão varridas e os elementos de mesmo índice (i , j) somados e postos no índice (i, j) da matriz C.

Teorema: O algoritmo SomaMatrizes é de cota inferior.

Prova: Para realizar a soma de duas matrizes de tamanho [m,n] é necessário varrer as duas matrizes para gerar uma terceira matriz com tamanho [m,n]. Claramente vemos que a complexidade intrínseca do problema é de ordem $m \cdot n$ (n^2 para matrizes quadradas).

O algotimo SomaMatrizes possui complexidade temporal da ordem $m \cdot n$ (n^2 para matrizes quadradas). Logo o algoritmo SomaMatrizes é de cota inferior.