Ficha 3

Algoritmos e Complexidade

Análise da complexidade de funções recursivas

1 Relações de Recorrência

1. Utilize uma árvore de recorrência para encontrar limites superiores para o tempo de execução dados pelas seguintes recorrências (assuma que para todas elas $T(0)$ é uma constante):

   (a) $T(n) = n + T(n-1)$

   (b) $T(n) = n + T(n/2)$

   (c) $T(n) = k + 2 * T(n-1)$ com k constante.

   (d) $T(n) = n + 2 * T(n/2)$

   (e) $T(n) = k + 2 * T(n/3)$ com k constante.

2. Considere o seguinte algoritmo para o problema das *Torres de Hanoi*:

```
void Hanoi(int nDiscos, int esquerda, int direita, int meio)
{
  if (nDiscos > 0) {
    Hanoi(nDiscos-1, esquerda, meio, direita);
    printf("mover disco de %d para %d\n", esquerda, direita);
    Hanoi(nDiscos-1, meio, direita, esquerda);
  }
}
```

   Escreva uma relação de recorrência que exprima a complexidade deste algoritmo (por exemplo, em função do número de linhas impressas no ecran). Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade.

3. Considere o seguinte algoritmo para o cálculo de números de Fibonacci:

```
int fib (int n)
{
  if (n==0 || n==1) return 1;
  else return fib(n-1) + fib(n-2);
}
```

Apesar de traduzir exactamente a definição da sequência de números de Fibonacci, este algoritmo é muito ineficiente (de tempo exponencial). Assuma que as operações aritméticas elementares se efectuam em tempo $\mathcal{O}(1)$.

(a) Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.

(b) Efectue uma análise assimptótica do tempo de execução deste algoritmo. Sugestão: Utilize a árvore que desenhou na alínea anterior para fundamentar o seu raciocínio.

(c) Escreva em **C** um algoritmo alternativo mais eficiente. Analise o seu tempo de execução.

2 Algoritmos de Ordenação

1. Considere a seguinte definição em `Haskell` do algoritmo de ordenação por inserção (*insertion sort*) em listas.

```
isort :: (Ord a) => [a] -> [a]
iSort [] = []
isort (h:t) = insert h (isort t)
    where insert y [] = [y]
          insert y (x:xs) | y <= x    = y:x:xs
                          | otherwise = x:(insert y xs)
```

Esta definição pode ser convertida numa função em C que ordena um vector:

```
void isort (int v[], int N) {
int i; int t;
if (N>0) {
    isort (v+1,(N-1));
    i = 0; t = v[0];
    while ((i<N-1) && (v[i] < t)) {
        v[i] = v[i+1]; i++;
    }
    if (i>0) v[i] = t;
   }
}
```

(a) Identifique o melhor e pior casos de execução desta função.

(b) Para esses casos, apresente uma relação de recorrência que traduza o número de comparações entre elementos do vector em função do tamanho do vector.

2. Considere agora a seguinte definição da função que ordena um vector usando o algoritmo de *merge sort*.

```
void msort (inv v[], int N) {
int *aux = (int *) malloc (N*sizeof(int));
```

```
    msortAux (v, aux, 0, N-1);
    free (aux);
}

void msortAux (int v[], int aux [], int a, int b) {
int m;

    if (a<b) {
        m = (a+b)/2;
        msortAux (v, aux, a, m);
        msortAux (v, aux, m+1, b);
        merge (x, aux, a, m, b);
       }
}
```

(a) Defina a função `merge` usada acima e que funde duas porções de um vector (uma com índices $[a \dots m]$ e outra com índices $[m+1 \dots b]$) usando um vector `aux` como auxiliar.

Garanta que o tempo de execução dessa função é $\Theta(N)$ em que $N = b-a+1$, i.e., a soma dos tamanhos dos dois vectores a serem fundidos.

(b) Apresente uma relação de recorrência que traduza o tempo de execução de `msort` (ou equivalentemente de `msortAux`), em função do tamanho do vector argumento. Apresente ainda uma solução dessa recorrência.

3. O algoritmo de ordenação Quick-sort pode ser implementado em C da seguinte forma:

```
void qSort (int v [], int N) {
   qSortAux (v, 0, N-1);
}

void qSortAux (int v[], int a, int b) {
int p;

    if (a<b) {
        p = particao (v,a,b);
        qSortAux (v,a,p-1);
        qSortAux (v,p+1,b);
    }
}
```

A função `particao` reorganiza o vector `v[a..b]` e retorna um índice `p` de tal forma que, após a sua terminação,

$$\forall_{a \leq k < p}(v[k] < v[p]) \wedge \forall_{p < k \leq b}(v[k] \geq v[p])$$

(a) Complete a seguinte definição da função `particao`.

```
int particao (int v[], int a, int b)}
int i, j;
    i = ...; j = ...
    while (...) {
        ...
        j++;
       }
    swap (v,i,b);
    return i;
}
```

Para isso use o seguinte predicado como invariante do ciclo `while`:

$$\forall_{a \leq k < i}(v[k] < v[b]) \wedge \forall_{i \leq k < j}(v[k] \geq v[b])$$

(b) Mostre que o tempo de execução da função `particao` é linear no tamanho do vector ($\Theta(N)$).

(c) Assumindo que os valores do vector são tais que o valor da função `particao (v,a,b)` é sempre de `(a+b)/2`, apresente uma relação de recorrência que traduza o tempo de execução do *quick-sort* em função do tamanho do *array*.

Apresente ainda uma solução dessa recorrência.

(d) Em que casos é que o valor da função `particao (v,a,b)` é sempre igual a `b`? Qual o tempo de execução desta função para esse caso?

(e) Para calcularmos o valor médio do tempo de execução do algoritmo de quick-sort vamos assumir que a probabilidade de cada um dos valores possíveis tem uma distribuição uniforme (i.e., se o vector tiver N elementos, a probabilidade de cada um dos N valores possíveis é $\frac{1}{N}$).

Vamos então tentar calcular o número médio (esperado) do número de comparações efectuadas pelo algoritmo de quick-sort.

Para um vector de tamanho N, a função `particao` tem que comparar o *pivot* com todos os outros elementos, fazendo por isso $N-1$ comparações.

Seguindo a definição recursiva da função `qSortAux` podemos definir com uma recorrência a função $F(N)$ que traduza o número médio de comparações que o algoritmo de quick-sort efectua:

$$F(N) = (N-1) + \sum_{p=1}^{N} \frac{1}{N} \left(F(p-1) + F(N-p)\right) \quad \text{para } N > 0$$

Esta expressão pode ser simplificada, se expandirmos o somatório:

$$\begin{aligned}
&\textstyle\sum_{p=1}^{N} \frac{1}{N} \left(F(p-1) + F(N-p)\right) \\
&\quad = \tfrac{1}{N} \left((F(0) + F(N-1)) + (F(1) + F(N-2)) + \cdots + (F(N-1) + F(0))\right) \\
&\quad = \tfrac{1}{N} \left((F(0) + F(0)) + (F(1) + F(1)) + \cdots + (F(N-1) + F(N-1))\right) \\
&\quad = \tfrac{2}{N} \left(F(0) + F(1) + \cdots + F(N-1)\right) \\
&\quad\ \ \text{uma vez que } F(0) = 0 \\
&\quad = \tfrac{2}{N} \textstyle\sum_{p=1}^{N-1} F(p)
\end{aligned}$$

Temos então que

$$F(N) = (N-1) + \frac{2}{N} \sum_{p=1}^{N-1} F(p)$$

Multiplicando ambos os membros por N,

$$N\,F(N) = N\,(N-1) + 2 \sum_{p=1}^{N-1} F(p)$$

Calculando esta expressão para $N-1$

$$(N-1)\,F(N-1) = (N-1)\,(N-2) + 2 \sum_{p=1}^{N-2} F(p)$$

Subtraindo estas equações (membro a membro), obtemos

$$N\,F(N) - (N-1)\,F(N-1) = N\,(N-1) - (N-1)\,(N-2) + 2\,F(N-1)$$

O que, depois de simplificar, vem

$$F(N) = (2 - \frac{2}{N}) + (1 + \frac{1}{N})\,F(N-1)$$

Que já está na forma canónica de uma equação de recorrência de 1^{a} ordem.

4. Considere agora o problema de, num *array* não ordenado (e sem repetições) determinar o k-ésimo menor elemento. Uma das soluções mais eficientes consiste em aproveitar a função de partição do algoritmo de *quick-sort* apresentada acima.

```
int kesimo (int v[], int N, int k){
int a=0,b=N-1,p=-1;

    while (p!=k) {
        p = particao (b,a,b);
        if p<k b = p-1;
        else   a = p+1;
       }
return v[p];
}
```

Assumindo que os valores do vector são tais que o valor da função `particao (v,a,b)` é sempre de `(a+b)/2`, apresente uma relação de recorrência que traduza o tempo de execução do *kesimo* em função do tamanho do *array*.

Apresente ainda uma solução dessa recorrência.

3 Árvores binárias de procura

1. Considere a seguinte definição de um tipo para representar árvores binárias de procura (BST).

```
typedef struct btree {
        int value;
        struct btree *left, *right;
      } Node, *BTree;
```

   Apresente definições em C para resolver cada um dos problemas abaixo. Para cada caso apresente ainda relações de recorrência que traduzam o comportamento das funções em causa para dois casos extremos: (1) a árvore está perfeitamente desiquilibrada (i.e., o número de nodos da árvore é igual à altura da árvore) e (2) a árvore está equilibrada (i.e., a altura da árvore corresponde ao logaritmo do numero de nodos)

   (a) A função `int size (BTree)` calcula o número de nodos de uma BST.
   (b) A função `int altura (BTree)` calcula a altura de uma árvore.
   (c) A função `BTree add (Btreem int)` adiciona um elemento a uma árvore.
   (d) A função `int search (BTree, int)` determina se um inteiro ocorre numa dada árvore.
   (e) A função `int max (BTree)` determina o maior elemento de uma árvore (não vazia).

2. Uma árvore binária diz-se balanceada se a diferença de pesos entre as suas sub-árvores não for superior a 1. A seguinte função determina se uma árvore binária está ou não balanceada.

```
int balanceada (BTree a) {
    if (a) {
        l = altura (a->left);
        r = altura (a->right);
        return ((abs (l-r) <= 1)     &&
                balanceada (a->left) &&
                balanceada (a->right));
      }
    else return 1;
   }
```

   (a) Identifique o melhor e pior caso do tempo de execução desta função.
   (b) Para cada um dos casos identificados, apresente uma relação de recorrência que traduza o tempo de execução desta função em função do tamanho da árvore. Em ambos os casos apresente uma solução dessa recorrência.
   (c) Uma alternativa para melhorar o comportamento da função acima consiste em usar uma função auxiliar que, além de determinar se uma dada árvore está balanceada, calcula a sua altura.

```
int balanceada (BTree a) {
    int p;
    return (balanceadaAux (a, &p));
   }

int balanceadaAux (BTree a, int *p) {
    int l, r;
    ...
   }
```

Complete a definição acima de forma a garantir que no melhor e pior caso, a função executa em tempo linear ao tamanho da árvore. Justifique a sua solução apresentando recorrências que traduzam o comportamento da função nesses casos extremos.

3. As árvores AVL (propostas por *G.M. Adelson-Velskii e E.M. Landis*) são uma variante das árvores binárias de procura em que em cada nodo se guarda a diferença de pesos entre as sub-árvores.

```
typedef struct avlTree {
    int value;
    int bal;
    struct avlTree *left, *right;
  } AVLNode, *AVLTree;
```

   (a) Defina uma função `void calcBal (AVLTree)` que preenche o campo `bal` de cada nodo da árvore com o valor correspondente. Use a estratégia apresentada acima para determinar se uma árvore está balanceada de forma a que a sua solução execute em tempo linear no tamanho da árvore.

   (b) Defina uma função `int alturaAVL (AVLTree)` que calcule a altura de uma árvore em tempo proporcional à altura da árvore (i.e., em tempo logaritmico relativamente ao tamanho da árvore no caso de uma árvore balanceada).

4. O algoritmo de inserção proposto por *G.M. Adelson-Velskii* e *E.M. Landis* insere um novo elemento numa árvore balanceada mantendo-a balanceada. Para isso usa retorna um valor adicional que indica se essa inserção provocou um aumento do peso da árvore.

```
AVLTree addAVL (AVLTree a, int x) {
int aumentou;
    return (addAVL_a (a, x, &aumentou));
}

AVLTree addAVL_a (AVLTree a, int x, int *aum) {
AVLTree new;
    if (a == NULL) {
        new = (AVLTree) (malloc (sizeof (AVLNode)));
        new->value = x; new->bal = 0;
        new->left = new->right = NULL;
```

```
        *aum = 1;
        return new;
       }
    else if (a->value > x) return (addAVL_left (a,x,aum));
    else addAVL_right (a,x,aum);
}
```

A função `addAVL_left` pode ser definida da seguinte forma

```
AVLTree addAVL_left (AVLTree a, int x, int *aum) {
    a->left = addAVL_a (a->left, x, aum);
    if (aum) if (a->bal == (-1)) // Direita mais pesada
                { a->bal = 0; *aum = 0;}
             else if (a->bal == 0) // balanceada
                     a->bal = 1;
             else // esquerda mais pesada
                  { a = corrige-left (a); *aum = 0;}
    return a;
}
```

(a) Defina a função `corrige-left`. Note as seguintes propriedades da árvore que esta função recebe como argumento:
   - é uma árvore não vazia (de facto tem pelo menos 3 elementos!);
   - trata-se de uma árvore que ficou desbalanceada (para a esquerda) após a inserção de um elemento na sua sub-árvore esquerda;
   - ambas as sub-arvores são árvores balanceadas.

(b) Apresente ainda uma definição da função `addAVL_right` usada acima.

A Relações de recorrência de 1ª ordem

De forma a podermos resolver relações de 1ª ordem (da forma $x_{n+1} = a_{n+1} \; + b_{n+1}\, x_n$) vamos começar por apresentar os casos mais simples:

1. $x_{n+1} = b\, x_n$

   Expandindo os primeiros elementos desta série, podemos induzir facilmente o caso geral:

$$\begin{array}{l} x_1 = b\, x_0 \\ x_2 = b\, x_1 = b^2\, x_0 \\ x_3 = b\, x_2 = b^3\, x_0 \\ \ldots \\ x_n = b^n\, x_0 \end{array}$$

2. $x_{n+1} = b_{n+1}\, x_n$

   Também aqui, expandindo os primeiros elementos desta série, podemos induzir o caso geral:

$$\begin{array}{l} x_1 = b_1\, x_0 \\ x_2 = b_2\, x_1 = b_1\, b_2\, x_0 \\ x_3 = b_3\, x_2 = b_1\, b_2\, b_3\, x_0 \\ \ldots \\ x_n = x_0\, \Pi_{i=1}^{n}\, b_i \end{array}$$

3. $x_{n+1} \; = x_n + c_{n+1}$ Mais uma vez, vamos expandir os primeiros elementos da série:

$$\begin{array}{l} x_1 = x_0 + c_1 \\ x_2 = x_1 + c_2 = x_0 + c_1 + c_2 \\ x_3 = x_2 + c_3 = x_0 + c_1 + c_2 + c_3 \\ \ldots \\ x_n = x_0 + \sum_{i=1}^{n}\, c_i \end{array}$$

4. $x_{n+1} = b_{n+1}\, x_n + c_{n+1}$

   Neste caso, a expansão dos primeiros elementos da série dá apenas uma leve intuição sobre o resultado:

$$\begin{array}{ll} x_1 & = b_1\, x_0 + c_1 \\ x_2 & = b_2\, x_1 + c_2 \\ & = b_2\, (b_1\, x_0 + c_1) + c_2 \\ & = b_1\, b_2\, x_0 + c_1\, b_2 + c_2 \\ x_3 & = b_3\, x_2 + c_3 \\ & = b_3\, (b_1\, b_2\, x_0 + c_1\, b_2 + c_2) + c_3 \\ & = b_1\, b_2\, b_3\, x_0 + c_1\, b_2\, b_3 + c_2\, b_3 + c_3 \\ x_4 & = b_4\, x_3 + c_4 \\ & = b_1\, b_2\, b_3\, b_4\, x_0 + c_1\, b_2\, b_3\, b_4 + c_2\, b_3\, b_4 + c_3\, b_4 + c_4 \\ \ldots & \end{array}$$

   Uma forma de resolver esta recorrência consiste em definir uma nova série $\{y_n\}_{n>0}$ de tal forma que

$$x_n = b_1\, b_2 \ldots b_n\, y_n \qquad (\text{com } y_0 = x_0)$$

Com esta série, podemos reescrever a equação $x_{n+1} = b_{n+1}\, x_n + c_{n+1}$ como

$$b_1\, b_2 \dots b_{n+1}\, y_{n+1} = b_{n+1}\, (b_1\, b_2 \dots b_n)\, y_n + c_{n+1}$$

Dividindo ambos os membros por $b_1\, b_2 \dots b_{n+1}$ obtemos a recorrência

$$y_{n+1} = y_n + \frac{c_{n+1}}{b_1\, b_2 \dots b_{n+1}}$$

Que tem a forma já vista acima. Seja d_i definido por

$$d_i = \frac{c_i}{b_1\, b_2 \dots b_i}$$

Então temos,

$$y_n = x_0 + \sum_{i=1}^{n} d_i = x_0 + \sum_{i=1}^{n} \frac{c_i}{b_1\, b_2 \dots b_i}$$

Substituindo agora na definição de y_n temos a solução para a recorrência:

$$x_n = b_1\, b_2 \dots b_n\, (x_0 + \sum_{i=1}^{n} \frac{c_i}{b_1\, b_2 \dots b_i})$$