UNIVERSIDADE DA BEIRA INTERIOR

Algoritmos e Estruturas de Dados — **2º Semestre**

Frequência 1 (8 val) — **Resolução** — **2024/2025**

A. [1.25 val] Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

```
...
for (k = 1; k <= n; k++) {
   for (j = k; j < k+2; j++)
      printf("OLA");
}
...
```

1. Simule o bloco de código dado, considerando que **n** é um número inteiro **muito grande**. Apresente os dados da simulação numa tabela cujo cabeçalho é o seguinte:

k	k <= n (S/N)	j	j < k+2 (S/N)	printf
(k=1) 1	1 <= n (S)	(j=k) 1	1 < 1+2 (S)	OLA
		(j++) 2	2 < 1+2 (S)	OLA
		(j++) 3	3 < 1+2 (N)	
(k++) 2	2 <= n (S)	(j=k) 2	2 < 2+2 (S)	OLA
		(j++) 3	3 < 2+2 (S)	OLA
		(j++) 4	4 < 2+2 (N)	
...	...	...	...	...
(k++) n	n <= n (S)	(j=k) n	n < n+2 (S)	OLA
		(j++) n+1	n+1 < n+2 (S)	OLA
		(j++) n+2	n+2 < n+2 (N)	
(k++) n+1	n+1 <= n (N)			

2. A partir dos resultados da simulação obtidos em 1, determine a **ordem de complexidade** do algoritmo associado ao bloco de código dado. Justifique, apresentando os cálculos efetuados.

Para determinar a ordem de complexidade do algoritmo, basta contabilizar o número de vezes que a operação dominante é executada. Analisando a tabela de simulação, conclui-se que a operação dominante é a comparação "j < k + 2".

T(n) = **3** [k=1] + **3** [k=2] + ... + **3** [k=n] = **3.n** $\Rightarrow$ **T(n)** = **O(n)**

Podia-se também contabilizar todas as operações, somando o número de vezes que cada uma das operações é executada, da seguinte forma:

T(n) = **1** [k=1] + **n** [k++] + **(n+1)** [k <= n] + **n** [j = k] + **2.n** [k++] + **3.n** [j < k+2] + **2.n** [printf]

T(n) = **10.n + 2** $\Rightarrow$ **T(n)** = **O(n)**

B. [1.50 val] Pilhas

Considere as seguintes declarações associadas a EAD Pilha:

```
typedef  int  INFOPilha;
struct NodoPilha { ... };
typedef struct NodoPilha *PNodoPilha;
```

assim como os seguintes protótipos de funções já implementadas (operações básicas sobre a EAD Pilha):

```
PNodoPilha criarPilha ();
int pilhaVazia (PNodoPilha S);
PNodoPilha push (INFOPilha X, PNodoPilha S);
PNodoPilha pop (PNodoPilha S);
INFOPilha topo (PNodoPilha S);
```

Usando as operações básicas sobre uma EAD Pilha, implemente uma **função em C** que

- **receba** uma **pilha S** já preenchida com valores inteiros (parâmetro da função),
- **determine** e **devolva** o elemento (número inteiro) com valor **negativo** que se encontra **mais afastado** do topo da pilha **S** (ou seja, **mais próximo do fundo** da pilha), mas deixando a pilha S inalterada; se não há elementos negativos na pilha S, então devolver **0**.

Ex: S = [-3, -12, 4, -5, -4, 5], com topo(S) = -3 => resultados: -4 e S = [-3, -12, 4, -5, -4, 5]

```
{ INFOPilha elementoNegativo (PNodoPilha *S)  }
int elementoNegativo (PNodoPilha *S)   // int = INFOPilha
{
   int  res;   // INFOPilha res;

   res = 0;
   SAux = criarPilha();
   while (pilhaVazia(*S) == 0) {
      if (topo(*S) < 0)
         res = topo(*S);
      SAux = push(topo(*S), SAux);
      *S = pop(*S);
   }

   while (pilhaVazia(SAux) == 0) {
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }

   return res;
}
```

C. [1.75 val] Árvores Binárias (AB)

Considere as seguintes declarações associadas a EAD Árvore Binária:

```
typedef  int  INFOAB;
struct NodoAB {
   INFOAB  Elemento;
   struct  NodoAB  *Esquerda;
   struct  NodoAB  *Direita;
};
typedef  struct  NodoAB  *PNodoAB;
```

Implemente uma função em C que

- **receba** uma AB **T** e um número inteiro **N** (parâmetros da função),
- **determine** e **devolva** a quantidade de elementos de **T** com valor no campo **Elemento** maior que **N**.

{ int quantidadeMaiorN (PNodoAB T, INFOAB N) }

```
int quantidadeMaiorN (PNodoAB T, int N)   //  int = INFOAB
{
   int  qEsq, qDir;
   // caso base/terminal
   if (T == NULL)
      return  0;
   // caso geral
   qEsq = quantidadeMaiorN(T->Esquerda, N);
   qDir = quantidadeMaiorN(T->Direita, N);
   if (T->Elemento > N)
      return  1 + qEsq + qDir;
   else
      return  qEsq + qDir;
}
```

D. [3.50 val] Listas ligadas

Considere as seguintes declarações associadas a EAD Lista:

```
typedef  int  INFOLista;
struct NodoLista {
    INFOLista  Elemento;
    struct NodoLista *Prox;
};
typedef struct NodoLista *PNodoLista;
```

assim como os seguintes protótipos de funções já implementadas (operações básicas sobre a EAD Lista):

```
int listaVazia (PNodoLista L); // devolve 1 se lista L é vazia e 0 caso contrário
PNodoLista libertarNodoLista (PNodoLista P); // liberta a memória apontada por P e devolve NULL
```

Usando as definições e as funções dadas (protótipos), resolva as seguintes questões:

1. [1.5 val] Implemente uma **função recursiva em C** que

- **receba** uma lista **L**, com elementos do tipo INFOLista (números **inteiros**) e um número inteiro **X**,
- **determine** e **devolva** a soma dos valores do campo **Elemento** dos elementos de **L** com valores **menores** que **X** (< X) daquele campo.

```
{ INFOLista somaMenoresX (PNodoLista L, INFOLista X) }
int somaMenoresX (PNodoLista L, int X)   // int = INFOLista
{
   int  soma;   //  INFOLista soma;

   // caso base/terminal
   if (L == NULL)
      return  0;

   // caso geral
   soma = somaMenoresX(L->Prox, X);
   if (L->Elemento < X)
      return  soma + L->Elemento;
   else
      return  soma;
}
```

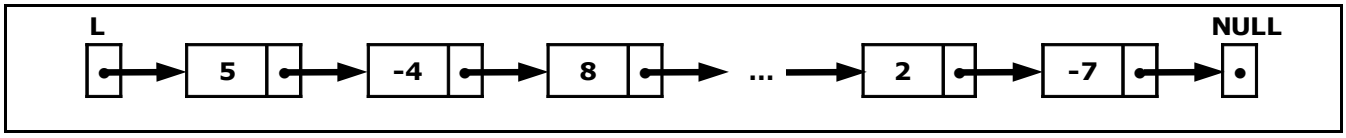
---

2. [2.0 val] Usando ou **não** as funções fornecidas em cima, mas **não podendo** usar outras funções, **implemente uma função iterativa (não recursiva) em C** que

- **receba** uma lista **L** de elementos do tipo **inteiro** (parâmetro da função),
- **devolva** a lista L após **remover** o elemento com valor positivo no campo Elemento que se encontra **mais afastado** da cabeça da lista **L** (**mais próximo da cauda**), caso exista; se não existir, manter a lista inalterada.

Nota: Otimizar o algoritmo, de forma a que a lista L seja percorrida apenas uma vez.

Exemplo: Se **L** é a lista em baixo, então o elemento a remover é o com o número **2**, pois é o elemento mais afastado da cabeça da lista (que é o nodo com 5) com valor positivo.



```
PNodoLista removerPositivo (PNodoLista L)
{
   PNodoLista PPos, P, PAnt;

   if (L == NULL)
      return L;

   PPos = NULL;    // ponteiro para o antecessor do último nodo positivo analisado (nodo a remover)
   P = L;          // ponteiro para percorrer todos os nodos da lista
   PAnt = NULL;    // ponteiro para o nodo anterior ao nodo apontado por P
   while (P != NULL) {
      if (P->Elemento >= 0)
         PPos = PAnt;
      PAnt = P;
      P = P->Prox;
   }

   if ( PPos == NULL) {  // não existe nodo a remover ou o nodo a remover é a cabeça da lista
      if (L->Elemento >= 0) {  // o nodo a remover é a cabeça da lista
         P = L;
         L = L->Prox;
         P = libertarNodoLista(P);
      }
   }
   else {  // o nodo a remover não é a cabeça da lista
      P = PPos->Prox;
      PPos->Prox = P->Prox;
      P = libertarNodoLista(P);
   }
   return L;
}
```