UNIVERSIDADE DA BEIRA INTERIOR

Algoritmos e Estruturas de Dados **1º Semestre**

Frequência 1 **Resolução** **2025/2026**

A. Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

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

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

T(n) = **1** [k=1] + **2** [k=2] + **3** [k=3] + ... + **n-1** [k=n-1] + 0 [k=n] = (progressão aritmética)

$$= \frac{1+(n-1)}{2} \times (n-1) = \frac{n \times (n-1)}{2} = \frac{n^2 - n}{2}$$

Logo,

T(n) = **O(n^2)**

B. Algoritmos de ordenação e de pesquisa

Considere a seguinte função já implementada, que devolve o **menor índice** de um elemento do array **A** (de tamanho **N**) com valor igual a **E** (se E existe em A) ou **-1** (se E não existe em A):

```
int pesquisaSequencial (int E, int A[], int N);
```

Pretende-se implementar uma **função** em C que dados um array **A** com **N** elementos do tipo inteiro **ordenado** por ordem **crescente** e um inteiro **E**, **determine** e **devolva** a quantidade de elementos do array **A** com valores iguais a **E**.

Desta forma, complete a função seguinte para que consiga o objetivo proposto.

```
int  quantidadeIguais  (int A[], int N, int E)
{
   int  k, cont = 0;
   k = pesquisaSequencial(E, A, N);
   if (k >= 0){
      while(k < N && A[k] == E){
         cont = cont + 1;
         k = k + 1;
      }
   }
   return cont;
}
```

NOTA: A função **pesquisaSequencial** é para ser usada de forma a otimizar o algoritmo (realizar o menor número de operações possível).

C. Listas ligadas

1. 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** com elementos do tipo **DadosLista** (**int**) e um inteiro **X** (parâmetros da função),
- **devolva** a lista **L** após **inserir** um nodo com o elemento **X** como **sucessor** de um nodo com o **menor** elemento da lista **L**; se a lista **L** estiver vazia, o nodo com o **X** será o único elemento da lista.

```
PNodoLista inserirAposMenor (PNodoLista L, DadosLista X) {
    PNodoLista  P, PMenor, Novo;
    Novo = criarNodoLista(X);
    if (Novo == NULL)
        return  L;
    if (L == NULL)
        return  Novo;
    // procurar pelo menor elemento e marcá-lo com PMenor
    PMenor = L;
    P = L->Prox;
    while(P != NULL){
        if(P->Elemento < PMenor->Elemento)
            PMenor = P;
        P = P->Prox;
    }
    // inserir o novo elemento como sucessor do menor
    Novo->Prox = PMenor->Prox;
    PMenor->Prox = Novo;
    return  L;
}
```

---

2. Implemente uma **função recursiva em C** que

- **receba** uma lista **L**, com elementos do tipo **DadosLista** (números inteiros) e um número inteiro **K**
- **determine** e **devolva** a soma dos valores do campo **Elemento** dos elementos de **L** a partir da **K-ésima posição** da lista (o elemento da K-ésima posição não entra na soma).

```
int soma (PNodoLista L, int K) {
    int  s;
    // caso base/terminal
    if (L == NULL)
        return  0;
    // caso geral
    if (K > 0)
        return  soma(L->Prox, K-1);
    else
        return  L->Elemento + soma(L->Prox, K-1);   // K-1 ou K ou 0
}
```

D. Pilhas

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

- **receba** uma **pilha S** com valores inteiros positivos e sem repetições (parâmetro da função),
- **determine** e **devolva** a soma dos elementos que **estão acima do maior elemento da pilha**, mas devolvendo a pilha **S** como no início; se o maior elemento está no topo da pilha **S**, então devolver 0.

Ex: S = [3, 7, 2, 9, 5, 1], com topo(S) = 3 => resultados: 12 (3+7+2) e S = [3, 7, 2, 9, 5, 1]

```
int somaAcimaMaior (PNodoPilha *S) {
   PNodoPilha  SAux;
   int  soma = 0, maior;
   if (pilhaVazia(*S) == 1)  // se a pilha está vazia, a soma é 0
      return  0;
   SAux = criarPilha();
   maior = topo(*S);      // o topo da pilha S é primeiro maior e S é não vazia
   // determinar o maior elemento da pilha e construir uma pilha auxiliar com os elementos de S
   while (pilhaVazia(*S) == 0){
      if (topo(*S) > maior)
         maior = topo(*S);
      SAux = push(topo(*S), SAux);
      *S = pop(*S);
   }
   // reconstruir a pilha S até encontrar o maior elemento
   while (topo(SAux) != maior){
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }
   // transferir o maior elemento para a pilha S
   *S = push(topo(SAux), *S);
   SAux = pop(SAux);
   // somar os restantes elementos da pilha SAux e passá-los para a pilha S
   while (pilhaVazia(SAux) == 0){
      soma = soma + topo(SAux);
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }
   return  soma;
}
```
