UNIVERSIDADE DA BEIRA INTERIOR

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

Frequência 1 **Resolução** **2023/2024**

A. Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

```
cont = 0;
for (k = 0; k < n; k++) {
   P = k + 1;
   for (j = 1; j <= P; j++)
      cont = cont + 1;
}
```

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 (V/F)	P	j	j <= P (V/F)	cont
(k=0) 0	0 < n V.	(P=k+1) 1	(j=1) 1 (j++) 2	1 <= 1 V. 2 <= 1 F.	1 (0+1)
(k++) 1	1 < n V.	(P=k+1) 2	(j=1) 1 (j++) 2 (j++) 3	1 <= 2 V. 2 <= 2 V. 3 <= 2 F.	2 (1+1) 3 (2+1) [3=1+2]
(k++) 2	2 < n V.	(P=k+1) 3	(j=1) 1 (j++) 2 (j++) 3 (j++) 4	1 <= 3 V. 2 <= 3 V. 3 <= 3 V. 4 <= 3 F.	4 (3+1) 5 (4+1) 6 (5+1) [6=1+2+3]
...	...	...	...	...	…
(k++) n-1	n-1 < n V.	(P=k+1) n	(j=1) 1 (j++) 2 ... (j++) n (j++) n+1	1 <= n V. 2 <= n V. ... n <= n V. n+1 <= n F.	s + 1 s + 2 ... s + n
(k++) n	n < n F.				

NOTA: s = 1 + 2 + 3 + ... + (n-1) $\Rightarrow$ cont = s + n = 1 + 2 + 3 + ... (n-1) + n

2. A partir dos resultados da simulação obtidos em 1, determine a ordem de complexidade do algoritmo associado ao bloco de código dado em cima. Incidir os cálculos sobre a operação que mais vezes é executada ("**j <= P**"). Justifique, apresentando os cálculos efetuados.

T(n) = número de vezes que a operação dominante ("**j <= P**") é executada

T(n) = 2 [k=0] + 3 [k=1] + 4 [k=2] + ... + (n+1) [k=n-1] + 0 [k=n] =

= 2 + 3 + 4 + ... + (n+1) [progressão aritmética com n termos] =

= ((2 + (n+1)) / 2) x n = ((n + 3) / 2) x n = (1/2) x (n^2 + 3.n)

Logo, a ordem de complexidade é: **O(n^2)** (ou: **T(n)** = **O(n^2)**)

B. EAD Pilha

Considere as seguintes declarações de EAD Pilha de elementos do tipo inteiro:

struct NodoPilha { ... };

typedef struct NodoPilha *PNodoPilha;

e os seguintes protótipos de funções (operações básicas):

PNodoPilha criarPilha ();

int pilhaVazia (PNodoPilha S);

PNodoPilha push (int X, PNodoPilha S);

PNodoPilha pop (PNodoPilha S);

int 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),
- **devolva** o elemento com valor positivo que se encontra **mais próximo** do topo da pilha **S**, mas deixando a pilha S inalterada.

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

```
int primeiroPositivo (PNodoPilha *S)
{
   int  res;
   PNodoPilha  SAux;
   SAux = criarPilha();
   // remover todos os elementos negativos da pilha até ficar vazia ou aparecer um elemento positivo
   while (pilhaVazia(*S) == 0 && topo(*S) < 0)
   {
      SAux = push(topo(*S), SAux);
      *S = pop(*S);
   }
   // se a pilha de entrada não ficou vazia, então o seu topo é um número positivo
   if (pilhaVazia(*S) == 0)  // pilha não vazia
      res = topo(*S);
   else   // pilha vazia
      res = -1;   // valor absurdo que significa que todos os elementos da pilha são negativos
   // reconstruir a pilha de entrada
   while (pilhaVazia(SAux) == 0)
   {
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }
   return  res;
}
```

C. EAD Árvore Binária (AB)

Considere a seguinte declaração de EAD Árvore Binária:

```
struct NodoAB {
    int    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 **num**,
- **devolva** o elemento (**int**) da AB **T** com **maior valor positivo** (≥ 0), se existir; caso não existam elementos com valores positivos, a função deve **devolver** um valor **negativo** qualquer.

```
int maiorElementoPositivo (PNodoAB T){
    int  maxEsq, maxDir, max;
    // caso terminal/base
    if (T == NULL)
        return -1;
    // caso geral: T não vazia
    maxEsq =  maiorElementoPositivo(T->Esquerda);
    maxDir =  maiorElementoPositivo(T->Direita);
    // determinar o maior entre maxEsq e maxDir
    if (maxEsq > maxDir)
        max = maxEsq;
    else
        max = maxDir;
    // determinar o maior elemento entre raiz e max
    if (T->Elemento < max)
        return max;
    else
        return T->Elemento;
}
```

D. Listas ligadas

Considere as seguintes declarações:

```
struct Nodo {
    int   Elemento;
    struct  Nodo  *Prox;
};
typedef  struct  Nodo *PNodo;
```

Considere também os seguintes protótipos de funções já implementadas:

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

Usando as definições e os protótipos das funções dadas, resolva as seguintes questões:

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

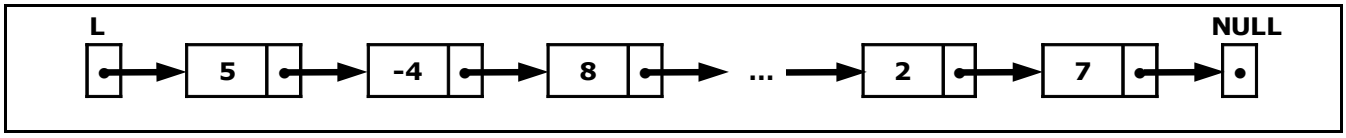
- **receba** uma lista não vazia **L**, com elementos do tipo **inteiro**,
- **devolva** o elemento de **L** com **maior** valor.

```
int  maiorElemento  (PNodo L){
    int  maior;
    // caso terminal/base: L só com um elemento
    if (L->prox == NULL)
        return  L->Elemento;
    // caso geral: L com pelo menos 2 elementos
    maior = maiorElemento(L->Prox);
    if (L->Elemento > maior)
        return  L->Elemento
    else
        return  maior;
}
```

---

2. 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 não vazia **L** de elementos do tipo **inteiro**,
- **devolva** a lista L após **remover** o **segundo** elemento com valor positivo que se encontra **mais próximo** da cabeça da lista **L**, caso exista; se não existir, a lista não sofre alteração.

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 aquele com o número **8**, pois é o segundo elemento mais próximo da cabeça da lista (nodo com 5) com valor positivo



```
PNodo removerSegundoPositivo (PNodo L) {
   PNodo P, PAnt;
   int k;
   if (L->Elemento >= 0)
      k = 1;
   else
      k = 0;
   P = L;
   while (P->Prox != NULL && k < 2) {
      if (P->Prox->Elemento >= 0)
         k = k + 1;
      PAnt = P;
      P = P->Prox;
   }
   if (k == 2) {  // remover nodo apontado por P (2º positivo); PAnt aponta para o antecessor de P
      PAnt->Prox = P->Prox;
      P = libertarNodo(P);
   }
   return L;
}
```

ou também:

```
PNodo removerSegundoPositivo (PNodo L) {
   PNodo P, PAnt;
   int k = 0;
   P = L;
   while (P != NULL) {
      if (P->Elemento >= 0) {
         k = k + 1;
         if (k == 2) {
            PAnt->Prox = P->Prox;
            P = libertarNodo(P);
            return L;
         }
      }
      PAnt = P;
      P = P->Prox;
   }
   return L;
}
```
