Gabarito da Avaliação a Distância 1 – Estrutura de Dados – Primeiro período de 2019

1. **(1,0) Descreva dois algoritmos, um recursivo e outro iterativo, para o cálculo da função de Fibonacci, dada pela seguinte fórmula recorrente:**

   F(1) = 0; F(2) = 1; F(n) = F(n-2) + F(n-1), para n ≥ 3.

   Responda: qual é a complexidade dos dois algoritmos em relação ao número de operações de adição efetuadas? Utilize a notação O na sua resposta, fornecendo apenas o termo dominante da expressão de complexidade. Justifique.

   R: Os algoritmos, interativo e recursivo, para o cálculo da sequência de Fibonacci são apresentados abaixo. A versão interativa possui complexidade linear de O(N), uma vez que o algoritmo cria uma laço que percorre os índices de 1 até o N. Por outro lado, a versão recursiva apresenta complexidade exponencial de $O(2^n)$. Isso acontece pois para cada operação de adição o algoritmo irá calcular a sequência de Fibonacci duas vezes, uma para cada um de seus operadores.

```
função f_interativo(n):
    n1 := 0
    n2 := 1

    para i:= 1 até N faça
        soma := n1 + n2
        n1 := n2
        n2 := soma

    retorne n2

função f_recursivo(n):
   se n <= 1 então
       retorne n
   senão
       retorne(f_recursivo(n - 2) + f_recursivo(n - 1))


n := 10
imprimir(f_interativo(n))
imprimir(f_recursivo(n))
```

2. **(1,0) Escreva um algoritmo que elimine de uma pilha P todos os elementos iguais a um certo valor X, mantendo os valores restantes na pilha na mesma ordem relativa em que se encontram. Observação: Só são permitidas operações de desempilhamento e empilhamento. Dica: Utilize uma pilha auxiliar Q para resolver este problema.**

   R: O algoritmo abaixo apresenta uma solução para este problema. Considerando que as operações permitidas são apenas empilhar e desempilhar, o algoritmo percorre toda a pilha P desempilhando elemento a elemento. Nesta operação, se o valor do elemento desempilhado for diferente de X, o algoritmo o empilha na pilha auxiliar Q. Por fim, toda a pilha Q é desempilhada e os elementos são empilhados novamente em P, agora sem os elementos que continham o valor X.

   As funções auxiliares *empilha*(), *desempilha*() e *tamanho*(), respectivamente, são responsáveis por empilhar um valor em uma pilha, desempilhar o último elemento de uma pilha e, por fim, retornar o tamanho de uma pilha.

```
P := [4, 5, 7, 4, 8, 4, 12] %Pilha
Q := []                     %Pilha auxliar
topo := 7                   %Topo da pilha P

X := 4

enquanto topo > 1 faça:
    valor := desempilha(P)
    se valor != X, então
        empilha(Q, valor)

    topo -=1

topo := tamanho(Q)
enquanto topo > 1 faça
    valor := desempilha(Q)
    empilha(P, valor)

    topo -=1
```

3. **(1,0) Suponha um vetor V de tamanho n, contendo apenas valores 0 e 1. Elabore um algoritmo de tempo linear que ordene o vetor, através de trocas entre elementos.**
Exemplo: Se V = [0, 1, 1, 0, 0, 1, 0, 1], a resposta será [0, 0, 0, 0, 1, 1, 1, 1].
R: O algoritmo apresenta uma solução em tempo linear – O(n) – para este problema. O programa percorre a lista comparando o primeiro elemento com os demais itens. Assim que for encontrado outro elemento igual ao primeiro, o algoritmo realiza uma troca, trazendo o elemento em questão para a posição seguinte ao elemento de comparação.

```
lista := [ 0, 1, 1, 0, 0, 1, 0, 1]
indice_comparacao := 0

para i := 2 até N faça
    se lista[indice_comparacao] == lista[i] então
        temp := lista[i]
        lista[i] := lista[indice_comparacao + 1]
        lista[indice_comparacao + 1] := temp
        indice_comparacao += 1
```

4. **(1,5) Elabore em algoritmo que realize a seguinte tarefa: Dado um vetor V desordenado com n elementos e um certo número k tal que 1 ≤ k ≤ n, determinar o elemento que se encontra na posição k após o vetor estar ordenado. Exemplo: se V = [3, 1, 9, 2, 8, 5, 4] e k = 6, a resposta deve ser 8.**

   R: Dado o vetor desordenado e um número K, o algoritmo abaixo encontra a posição K no vetor V, após a sua ordenação.

```
v := [3, 1, 9, 2, 8, 5, 4]    %Vetor Inicial
N := 7                        %Tamanho do vetor
k := 6                        %Indice de busca

para i := 1 até N faça:
    menor := i

    para j := i + 1 até N faça):
        se v[j] < v[menor] então
            menor: = j

    aux := v[i]
    v[i] := v[menor]
    v[menor] := aux

valor = v[k]

imprimir ("O elemento presente na posição K do vetor, após a ordenação, é ", valor)
```

5. **(1,5) Estude a Aula 8 (que não faz parte do cronograma de 2019-1) sobre o tema "Busca Binária", e depois responda a seguinte pergunta: Determine uma lista ordenada L com 20 elementos e um elemento X a ser procurado em L de modo que o algoritmo de busca binária realize o pior caso possível em termos do número de comparações. Quantas comparações são feitas nesse caso?**

   R: O algoritmo de Busca Binária irá realizar, no máximo, $1 + \lfloor log_2 N \rfloor$ comparações, portanto, apresentando assintóticamente em seu pior caso, complexidade de $O(log_2 N)$. Isso significa que, em uma lista com 20 elementos, o algoritmo irá realizar no máximo 5 comparações, uma vez que $1 + \lfloor log_2 20 \rfloor = 5$.

   Neste sentido, considerando uma lista com os elementos enumerados sequencialmente de 1 à 20, o elemento X pode assumir, por exemplo, o valor 20, levando o algoritmo a atingir o piro caso.

6. **(1,5) Faça uma análise comparativa entre os métodos de ordenação por seleção e ordenação por bolha, em termos de pior caso e melhor caso, com relação a dois critérios: número de comparações e número de trocas de elementos realizadas. Ilustre sua análise com exemplos, utilizando vetores com 5 elementos.**

R: Para o Método Bolha, uma sequência que leva o algoritmo ao seu pior caso, tanto em número de trocas quanto em número de comparações, é aquela cujo elementos estão ordenados na ordem inversa a desejada. Isto é, caso pretenda-se ordenar de forma crescente, uma sequência cujo elementos estão ordenados de forma decrescente, figura o pior caso. Portanto, L = [5, 4, 3, 2, 1] resulta, neste caso, em 10 trocas e 10 comparações. Veja, na Tabela 1, que para cada troca houve uma comparação.

Tabela 1. Pior Caso - Método Bolha					
5	4	3	2	1	Estado Inicial
4 *	5	3	2	1	Trocou 4 e 5
4	3 *	5	2	1	Trocou 3 e 5
3 *	4	5	2	1	Trocou 3 e 4
3	4	2 *	5	1	Trocou 2 e 5
3	2 *	4	5	1	Trocou 2 e 4
2 *	3	4	5	1	Trocou 2 e 3
2	3	4	1 *	5	Trocou 1 e 5
2	3	1 *	4	5	Trocou 1 e 4
2	1 *	3	4	5	Trocou 1 e 3
1 *	2	3	4	5	Trocou 1 e 2

O melhor caso do Método Bolha é aquele cuja sequência numérica já está ordenada, o que não causará nenhuma troca, apenas comparações. Portando, L= [1, 2, 3, 4, 5] resulta em 0 trocas e apenas 4 comparações. Neste caso as comparações são:

a) 2 e 1
b) 3 e 2
c) 4 e 3
d) 5 e 4

Já o Método de Ordenação por Seleção, diferente do bolha, não possui diferenças entre o seu pior e melhor caso. Isso acontece, pois, o algoritmo irá comparar cada elemento do vetor com os demais, mesmo se o vetor já se encontrar ordenado. Contudo, o número máximo de trocas realizadas sempre será igual N, independente do estado inicial da sequência. Veja a Tabela 2.

Tabela 2. Método de Ordenação por Seleção											
Sequência Invertida						Sequência ordenada					
5	4	3	2	1	Estado Inicial	1	2	3	4	5	Estado Inicial
1	4	3	2	5	Trocou 1 e 5	1	2	3	4	5	Trocou 1 consigo mesmo
1	2	3	4	5	Trocou 2 e 4	1	2	3	4	5	Trocou 2 consigo mesmo
1	2	3	4	5	Trocou 3 consigo mesmo	1	2	3	4	5	Trocou 3 consigo mesmo
1	2	3	4	5	Trocou 4 consigo mesmo	1	2	3	4	5	Trocou 4 consigo mesmo
1	2	3	4	5	Trocou 5 consigo mesmo	1	2	3	4	5	Trocou 5 consigo mesmo

Neste exemplo, o algoritmo efetuou 5 trocas e 10 comparações para ambas as configurações de ordem da sequência numérica.

7. **(1,0) Seja L uma lista ordenada, simplesmente encadeada, com nó cabeça. Elabore um algoritmo que retire de L os elementos repetidos. Calcule sua complexidade.**

   R: O algoritmo a seguir apresenta uma solução em O(N) capaz de retirar os elementos repetidos em uma dada lista L, ordenada e simplesmente encadeada.

```
no_corrente := cabeca

enquanto no_corrente.proximo != Nulo:
    proximo := no_corrente.proximo

    se no_corrente.valor == proximo.valor:
        no_corrente.proximo := proximo.proximo

    senão:
        no_corrente := no_corrente.proximo
```

8. **(1,5) Dada uma lista L, simplesmente encadeada, deseja-se construir uma lista R, também simplesmente encadeada, que seja o reverso de L. Isto é, a lista R contém os mesmos elementos que L, porém em ordem inversa. Explicar, por meio de palavras, a estratégia empregada para a construção de R e, em seguida, descrever o processo em uma linguagem algorítmica.**

   R: A solução apresentada abaixo faz uso de recursividade para acessar os elementos da lista L e armazená-los na ordem inversa. O método *inverter*() percorre recursivamente toda a lista L até o ultimo nó. Ao chegar no fim, o valor armazenado neste nó é encadeado em R através do método *inserirProximo*(). Desse modo, ao passo que a recursividade vai se desempilhando o algoritmo vai armazenando na ordem inversa os nós encontrados em L.

```
%Cada nó da lista tem um campo de informação (valor) e um campo que aponta para o próximo elemento.
No(val):
    valor := val
    proximo := Nulo


função inverter(no):
    se (no == Nulo) então
        retorne

    inverter(no.proximo)
    inserirProximo(R, no.valor)


função inserirProximo(cabecaLista2, valor):
    no_corrente := cabecaLista2

    enquanto no_corrente.proximo != Nulo faça
        no_corrente := no_corrente.proximo

    no_corrente.proximo := No(valor)


R := No(Nulo)         %Cabeça Lista 2 (Ordem Inversa)
inverter(L)           %Começa inverter pelo nó cabeca da Lista L
```