Lista 5 - Francisco Braz

1-)
a-)
```
algoritmo interseccaoConjuntos(A,n)
{-Entrada: vetor A de m elementos em ordem arbitrária e vetor B de n
elementos em ordem arbitrária
Saída: elementos presentes na intersecção de A com B-}
início
        função Heapsort(A,n):vazio
        início
             MontaMaxHeap(A,n)
             para i = n decrescendo até 2 faça
             início
                  Troca(A[1], A[i])
                  RearranjeMaxHeap(A, i-1);
             fim
        fim

        funcão Encontre(X,esq,dir,z):inteiro
        início
            se (esq = dir) então
                se (X[esq] = z) então retorne esq
                senão retorne 0
            senão
                meio := ⌈(esq+dir)/2⌉
                se (z < X[meio]) então
                    retorne Encontre(X,esq,meio-1,z)
                senão
                    retorne Encontre(X,meio,dir,z)
        fim


        função Intersecção(A, B, m, n):vazio
        início
             para i = 1 até n faça
             início
```

```
            resp := Encontre(A, 0, m, B[ i ])
            se resp <> 0 então imprima B[ i ]
        fim
    fim
fim
```

b-)

A complexidade geral desse algoritmo seria:
complexidade Heapsort() + complexidade Encontre() + complexidade Intersecção()

Começando pela complexidade da função HeapSort:

$$T_{heapsort} = T_{MontaMaxHeap} + \sum_{i=2}^{n} (T_{Troca} + T_{RearranjeMaxHeap})$$

A complexidade do MontaMaxHeap é proporcional à soma das alturas dos nós deste heap, que por sua vez é a altura do nó raíz do heap mais a altura dos demais nós que compõem dois heaps de altura i-1.
Uma árvore binária completa de altura i, por sua vez, tem $n = 2^{i+1} - 1$ nós.
Logo,

$$\begin{aligned} T_{MontaMaxHeap}(m) &= O(H(i)) \\ &= O(m - (\log(m+1))) \\ &= O(m) \end{aligned}$$

A complexidade do Troca, seria constante, então:
$T_{Troca} = k$

A complexidade do RearranjeMaxHeap é proporcional à altura da árvore, esse valor será o número máximo de vezes que o “enquanto”, presente nessa função, será executado. Logo,

$$T_{RearranjeMaxHeap} = \sum_{j=i}^{i} c = c * i = c * \log m$$

Portanto, somando essas complexidades, teremos:

$T_{heapsort} = O(m) + \sum_{i=2}^{m} (k + c * logm)$

$= O(m) + [\ (k + c * logm) * (m-2+1)\ ]$

$= O(m) + [\ (k + c * logm) * (m-1)\ ]$

$= O(m) + [\ m*k - m * c * logm - k - c * logm\ ]$

Ignorando as constantes e considerando o termo de maior grau (m*logm), nossa complexidade seria O(m*logm).

Complexidade da função Encontre:
Usando o método de derivação por substituição visto nos slides
T(m) = T(m/2) + c, T(1) = c1

Supondo n = $2K \rightarrow T(2K) = T(2K-1) + c$, T(1) = c1
$T(m) = T(2^{K-1}) + c2$
$= (T(2^{k-2}) + c2) + c2$
$= ((T(2^{K-3}) + c2) + c2\ ) + c2$
$= T(2^0) + K * c2 = c1 + K * c2$

Como $n = 2^K$, então $K = log_2 n$. Portanto, $T(m) = log_2 m * c2$
Logo, $O(log_2 m)$.

Complexidade da função Intersecção:

$\sum_{i=1}^{n} c + T_{Encontre}$

$= \sum_{i=1}^{n} c + logm$

= (c + logm) * (n-1 + 1)
= n * c + n * logm

Considerando o termo de maior grau, temos O(n * logm).

Logo, juntando as complexidades, temos:
= (m * logm) + (n * logm)
= (m+n) * logm
= O((m+n)*logm)

c-) Implementação no arquivo interseccaoConjuntos.c

2-)
a-)
algoritmo ordenarVetorStrings(A,n)
{-Entrada: vetor A de n elementos em ordem arbitrária
Saída: vetor ordenado-}
início

```
RearranjeMaxHeap(A,n)
início
    pai := 1; filho := 2
    enquanto filho <= n-1 faça
    se A[filho].tamanho < A[filho +1].tamanho então filho := filho + 1
    senão se A[filho].tamanho = A[filho +1].tamanho &&
    A[filho+1].posicao > A[filho].posicao então
        filho := filho + 1

   se A[filho] >= A[pai] então
        se A[filho].tamanho < A[filho +1].tamanho então
             Troca(A[pai], A[filho]); pai := filho; filho := 2 * filho
        senão se A[filho].tamanho = A[filho +1].tamanho &&
        A[filho+1].posicao > A[filho].posicao então
             Troca(A[pai], A[filho]); pai := filho; filho := 2 * filho
   senão filho := n
fim



função MaxHeapify(A, i)
```

```
        início
           esq := 2*i; dir := 2*i+1
           se esq <= n and A[esq] >= A[i] então
                se A[esq] > A[i] então maior := esq
                se A[esq].tamanho = A[i].tamanho &&
                A[esq].posicao >  A[i].posicao então
                    maior := esq
           senão maior:= i

           se dir <= n and A[dir] >= A[maior] então
                se A[dir] > A[i] então maior := dir
                se A[dir].tamanho = A[i].tamanho &&
                A[dir].posicao >  A[i].posicao então
                    maior := dir

           se maior <> i
           então Troca(A[i], A[maior]); MaxHeapify(A, maior)
        fim

fim

função Heapsort(A,n)
início
    MontaMaxHeap(A,n)
    para i = n decrescendo até 2 faça
    início
        Troca(A[1], A[i])
        RearranjeMaxHeap(A, i-1);
   fim
fim
```

b-)

Complexidade da função HeapSort:

$$T_{heapsort} = T_{MontaMaxHeap} + \sum_{i=2}^{n} (T_{Troca} + T_{RearranjeMaxHeap})$$

A complexidade do MontaMaxHeap é proporcional à soma das alturas dos nós deste heap, que por sua vez é a altura do nó raíz do heap mais a altura dos demais nós que compõem dois heaps de altura i-1.
Uma árvore binária completa de altura i, por sua vez, tem $n = 2^{i+1} -1$ nós. Logo,

$T_{MontaMaxHeap} (n) = O(n)$

A complexidade do Troca, seria constante, então:
$T_{Troca} = k$

A complexidade do RearranjeMaxHeap é proporcional à altura da árvore, esse valor será o número máximo de vezes que o "enquanto", presente nessa função, será executado. Logo,

$$T_{RearranjeMaxHeap} = \sum_{j=i}^{i} c = c * i = c * \log n$$

Logo, teremos:

$$T_{heapsort} = O(n) + \sum_{i=2}^{m} (k + c * \log m)$$

$$= O(n) + [(k + c * \log n) * (n-2+1)]$$

$$= O(n) + [(k + c * \log n) * (n-1)]$$

$$= O(n) + [n*k - n * c * \log m - k - c * \log n]$$

Ignorando as constantes e considerando o termo de maior grau (n*logn), nossa complexidade seria O(n*logn).

c-) Implementação no arquivo ordenarVetorString.c

3-)
item a-)
a-)
{-Entrada: vetor A de n elementos que informam limite inferior e superior do intervlado
Saída: vetor ordenado em ordem crescente-}
início

```
    função gerarVetor(A, limInf, limSup, 0)
    função HeapSort(A,n)

    função gerarVetor(A, limInf, limSup, countIndex)
    início
        limite := b-a
        para i = 1 até limite faça
        A[countIndex]  = a
        a = a + 1
        countIndex = countIndex + 1
    fim

    função Heapsort(A,n):vazio
    início
        MontaMaxHeap(A,n)
        para i = n decrescendo até 2 faça
        início
            Troca(A[1], A[i])
            RearranjeMaxHeap(A, i-1);
        fim
    fim
```

fim

b-)
A complexidade da função "gerarVetor" é O(n), já que possui somente um loop simples. Por sua vez, a complexidade do HeapSort, como já foi visto nas questões anteriores é O(n * log). Portanto, temos:

n + n * logn

Considerando somente o termo de maior grau, nossa complexidade seria O(n*logn)

c) Implementação no arquivo vetorIntervalos.c

item b-)
a-) Com o vetor em mãos podemos usar duas funções para encontrar as posições:

```
funcão EncontrePrimeiro(X,esq,dir,z):inteiro
início
    se (esq = dir) então
        se (X[esq] = z) então retorne esq
        senão retorne 0
    senão
        meio := ⌈(esq+dir)/2⌉
        se X[meio] = z então
           se X[meio-1] = z então
                retorne EncontrePrimeiro(X,esq,meio-1,z)
           senão retorne meio

        se (z < X[meio]) então
            retorne Encontre(X,esq,meio-1,z)
        senão
            retorne Encontre(X,meio,dir,z)
fim

funcão EncontreUltimo(X,esq,dir,z):inteiro
início
    se (esq = dir) então
        se (X[esq] = z) então retorne esq
        senão retorne 0
    senão
        meio := ⌈(esq+dir)/2⌉
        se X[meio] = z então
           se X[meio-1] = z então
                retorne EncontrePrimeiro(X,meio+1,dir,z)
```

```
            senão retorne meio

         se (z < X[meio]) então
             retorne Encontre(X,esq,meio-1,z)
         senão
             retorne Encontre(X,meio,dir,z)
    fim
```

b-)

A complexidade da busca binária:

$T(n) = T(n/2) + c2$, $T(1) = c_1$

Supondo $n = 2^K \rightarrow T(2^K) = T(2^{K-1}) + c2$, $T(1) = c_1$

$$
\begin{aligned}
T(n) &= T(2^{K-1}) + c_2 \\
&= (T(2^{K-2}) + c_2) + c_2 \\
&= ((T(2^{K-3}) + c_2) + c_2) + c_2 \\
&= T(2^0) + K \,.\, c_2 \\
&= c_1 + K \,.\, c_2
\end{aligned}
$$

Como $n = 2^K$, então $K = \log_2 n$. Portanto, $T(n) = \log_2 n + c1$

Logo, $O(\log_2 n)$

Como temos duas buscas, seria 2 * logn. Ignorando as constantes, temos O(logn).

c-) Implementação no arquivo vetorIntervalos

4-)

a-)

algoritmo paresDistintos(A,n)

{-Entrada: vetor A de n elementos em ordem arbitrária

Saída: valor da quantidade de pares distintos (i, j), tal que j > i e A[i] = A[j]-}

início

```
função fatorial(v)
início
    fat := 1;
    enquanto (v > 0)
    início
        fat := fat * v
        v= v- 1
    fim
fim

função countingSort(A, B, n, m)
início
    seja C[0..k] um novo vetor
    qtdPares = 0
    para i = 0 até k faça C[i] := 0
    para j = 1 até n faça C[A[j]] := C[A[j]]+1
    para i = 1 até k faça
    início
        combinacao := (fatorial(C[ i ]))/2
        qtdPares = qtdPares + 1

fim
imprima qtdPares
fim
```

b-)

Complexidade função countingSort:

Considerando os simples temos:

$$T(n,k) = \sum_{i=0}^{k} a + \sum_{j=1}^{n} b + \sum_{i=0}^{k} c$$

$$= a * (k-0+1) + b * (n-1+1) + c * (k-0+1) \quad = a + a*k + b*n + c*k + c$$

Se considerarmos os termos de maior grau e ignorarmos as constantes (a, b, c), temos que a complexidade seria O(k + n).

Complexidade função fatorial:

$$\sum_{i=1}^{v} c = c * (v-1+1) = c*v$$

Se ignorarmos a constantes teremos uma complexidade O(n)

Juntando as duas complexidades, ficamos com: k + n + n = 2n + k. Considerando o maior termo, vamos ter uma complexidade de O(n).

c-) Implementação no arquivo paresDistintos.c