Entrega 2

Daniel Brito

13) A ideia deste algoritmo consiste em, basicamente, utilizar a implementação do `mergeSort` para ordenar os elementos em tempo $O(n \log n)$, depois, utilizar o `removerDuplicatas`, cujo tempo é $O(n)$, e ir comparando um elemento com o seu adjacente à direita, a fim de armazenar os valores convenientes no array `resultado`.

```
def mergeSort(array):
    if len(array) > 1:
        mid = len(array)//2
        L = array[:mid]
        R = array[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            array[k] = R[j]
            j += 1
            k += 1

def removerDuplicatas(array):
    n = len(array)
    resultado = []

    if n == 1 or n == 0:
        return array

    for i in range(0, n-1):
        if(array[i] != array[i+1]):
            resultado.append(array[i])

    resultado.append(array[n-1])
    return resultado
```

14) Para resolver este problema, serão apresentados dois algoritmos, um iterarito e outro recursivo, ambos bastante intuitivos, pois utilizam a ideia de busca binária, atendendo a complexidade $O(\log n)$. A ideia consiste em realizar uma pequena modificação nos condicionais, ou seja, ao invés de procurar por um determinado valor x, compara-se o valor de `mid`, utilizando um lowerbound (`low`) e um upperbound (`up`) para auxiliar no processo.

I) Iterativo: Recebendo apenas o array como parâmetro.

```
def buscaIterativa(a):
    mid = 0
    low = 0
    up = len(a)-1

    while (low <= up):
        mid = (low + up) // 2

        if mid == a[mid]:
            return True
        if mid < a[mid]:
            up = mid-1
        else:
            low = mid+1

    return False
```

I) Recursivo: Recebendo o array `a`, o lowerbound `low = 0`, e o upperbound `up = len(a)-1`.

```
def buscaRecursiva(a, low, up):
    if up >= low:
        mid = (low+up) // 2

        if mid == a[mid]:
            return True
        if mid > a[mid]:
            return buscaRecursiva(a, (mid+1), up)
        else:
            return buscaRecursiva(a, low, (mid-1))
    return False
```

15) A ideia do algoritmo `busca` consiste em encontrar o sub-array dos elementos válidos, ou seja, antes de ∞, por meio da utilização de índices delimitadores. Uma vez que isto ocorre, tal sub-array é passado para o método de busca binária clássico. Ambos os métodos têm como complexidade $O(\log n)$, atendendo ao requisito do problema.

```
def busca(array, x):
    if array[0] > x:
        return False

    if array[0] == x:
        return True

    left = 1
    right = 1

    while(array[right] != "INF"):
        left = right
        right *= 2

    mid = 0

    while((right-left) > 1):
        mid = left + ((right-left) // 2)

        if(array[mid] == "INF"):
            right = mid
        else:
            left = mid

    return buscaBinaria(array[0:left+1], x)


def buscaBinaria(array, x):
    mid = 0
    low = 0
    up = len(array)-1

    while (low <= up):
        mid = (low + up) // 2

        if array[mid] == x:
            return True
        if x < array[mid]:
            up = mid-1
        else:
            low = mid+1

    return False
```

16) a) A ideia deste algoritmo consiste em combinar pares de arrays por meio do procedimento `merge`, o que resulta em uma complexidade final de $O(n * k * \log(n * k))$.

```
def combinarArrays(arrays):
    k = len(arrays)

    if(k < 2):
        return arrays

    resultado = merge(arrays[0], arrays[1])

    for i in range(2, k):
        resultado = merge(resultado, arrays[i])

    return resultado


def merge(arr1, arr2):
    n1 = len(arr1)
    n2 = len(arr2)
    arr3 = [None] * (n1 + n2)

    i = j = k = 0

    while i < n1 and j < n2:
        if arr1[i] < arr2[j]:
            arr3[k] = arr1[i]
            k = k + 1
            i = i + 1
        else:
            arr3[k] = arr2[j]
            k = k + 1
            j = j + 1

    while i < n1:
        arr3[k] = arr1[i];
        k = k + 1
        i = i + 1

    while j < n2:
        arr3[k] = arr2[j];
        k = k + 1
        j = j + 1

    return arr3
```

16) b) A versão otimizada do algoritmo anterior utiliza uma Min Heap, cuja implementação a ser apresentada foi desenvolvida em C, na disciplina de Estruturas de Dados Avançada. A ideia geral para combinar os arrays é criar uma Min Heap com cada um de seus elementos, e depois ir removendo os mesmos, convenientemente, de maneira a preencher o array `resultado`. Uma vez que temos $n * k$ elementos, e as operações de inserção é remoção da Min Heap possuem custo $O(\log n)$, no final, alcançamos uma complexidade $O(n * k * \log k)$.

```
// Modulo da MinHeap:

heap* criar(int nmax){
        heap* h = (heap*)malloc(sizeof(heap));
        h->n=0;
        h->nmax = nmax;
        h->v = (int*)malloc(sizeof(int)*nmax);
        return h;
}

int heap_vazia (heap *h){
        return h->n == 0;
}

void sobe (heap *h, int i){
        int pai;
        while(i>0){
                pai = pai(i);
                if(h->v[pai] <= h->v[i]) break;
                troca(h, pai, i);
                i = pai;
        }
}

void troca (heap *h, int i, int j){
        int temp = h->v[i];
        h->v[i] = h->v[j];
        h->v[j] = temp;
}

void desce (heap *h, int i){
        int filhoEsquerda = esq(i);
        int filhoDireita = dir(i);
        while(filhoEsquerda < h->n){
                filhoDireita = dir(i);
                if((filhoDireita > h->n) &&
                   (h->v[filhoDireita] < h->v[filhoEsquerda])){
                        filhoEsquerda = filhoDireita;
                }
                if(h->v[filhoEsquerda] > h->v[i]){
                        break;
                }
```

```
                troca(h, i, filhoEsquerda);
                i = filhoEsquerda;
                filhoEsquerda = esq(i);
        }
}

void heap_insere(heap *h, int valor){
        h->v[h->n++] = valor;
        sobe(h, h->n-1);
}

int heap_retira (heap *h){
        int raiz = h->v[0];
        h->v[0] = h->v[--h->n];
        desce(h, 0);
        return raiz;
}

// Programa principal:

int main(void) {
        int i, j, r, maximo_elementos, n = 3;
        int arrays[n][n] = {{1, 3, 5}, {2, 4, 8}, {7, 10, 21}};
        maximo_elementos = n*n;
        r = 0;
        int resultado[maximo_elementos];
        heap *h = criar(maximo_elementos);

        for(i=0; i<n; i++){
                for(j=0; j<n; j++){
                        heap_insere(h, arrays[i][j]);
                }
        }

        while(!heap_vazia(h)){
                resultado[r++] = heap_retira(h);
        }

    return 0;
}
```

17) Neste algoritmo é realizada a verificação de alguns casos base, então, se nenhum deles é atingido, são feitas chamadas recursivas passando-se os segmentos dos arrays para a procura o k-ésimo elemento.

```
def kesimoMenorElemento(arr1, arr2, x1, x2, k):
    if arr1[0] == x1:
        return arr2[k]

    if arr2[0] == x2:
        return arr1[k]

    mid1 = (x1 - arr1[0]) // 2
    mid2 = (x2 - arr2[0]) // 2

    if (mid1 + mid2) < k:
        if arr1[mid1] > arr2[mid2]:
            return kesimoMenorElemento(arr1, arr2[:mid2+1], x1, x2, k-mid2-1)
        else:
            return kesimoMenorElemento(arr1[:mid1+1], arr2, x1, x2, k-mid1-1)
    else:
        if arr1[mid1] > arr2[mid2]:
            return kesimoMenorElemento(arr1, arr2, arr1[mid1], x2, k)
        else:
            return kesimoMenorElemento(arr1, arr2, x1, arr2[mid2], k)
```

18) a) Neste algoritmo, o array `a` é dividido em dois segmentos, então, uma chama recursiva é realizada em ambas a partes para verificar a existência do elemento majoritário.

```
def elementoMajoritario(a):
    n = len(a)

    if n == 1:
        return a[0]
    if n == 0:
        return None

    k = n // 2

    elem1 = elementoMajoritario(a[:k])
    elem2 = elementoMajoritario(a[k+1:])

    if elem1 == elem2:
        return elem1

    count1 = a.count(elem1)
    count2 = a.count(elem2)

    if count1 > k:
        return elem1
    elif count2 > k:
        return elem2
    else:
        return None
```

18) b) Este algoritmo segue a ideia do enunciado, ou seja, dividir o array em $\frac{n}{2}$ pares, depois, realizar o procedimento de "armazenamento" de um dos valores, se ambos forem iguais, ou "descarte", se forem diferentes. Por fim, é verificada a existência do elemento majoritário.

```python
def elementoMajoritarioB(a, e=None):
    n = len(a)

    if n == 0:
        return e

    pares = []

    if n % 2 == 1:
        e = a[-1]

    for i in range(0, n-1, 2):
        if a[i] == a[i+1]:
            pares.append(a[i])

    majoritario = elementoMajoritarioB(pares, e)

    if majoritario is None:
        return None

    nMajoritario = a.count(majoritario)

    if 2 * nMajoritario > n or (2*nMajoritario == n and majoritario == e):
        return majoritario

    return None
```

18) c) Dada a relação de recorrência T(n) = T(n/2) + O(n), com base no Teorema Mestre, temos que T(n) = O(n).