Micael Andrade Dos Santos

1. Questão

   a. `IDEIA`

      Podemos utilizar um `heap-min` para encontrar o k-ésimo menor valor, para isso basta extrair do heap k vezes, o próximo será o k-ésimo menor.

   b. `ALGORITMO`

```
algoritmo MontaMinHeap(V, n)
{-Entrada: Vetor V com elementos em ordem arbitrária.
 -SAÍDA: Vetor V cujo seus elementos obdecem a propriedade heap-min.
.}
inicio
  para i = ⌊2/n⌋ decrescendo até 1 faça
    MinHeapfy(V, i)
fim
```

```
algoritmo MinHeapify(V, i)
{- Entrada vetor V e o nível do heap-min
  Saída Vetor V obedecendo a propriedade heap-min
-}
inicio
  esq:=2*1
  dir:=2*i+1
  se esq <= n && V[esq] < V[i] então
    menor:=esq
  senão
    menor:=i

  se dir <= n && V[dir] < V[menor] então
    menor := dir
  se menor <> i então
    Troca(V[i], V[menor]);
    MinHeapify(V, menor)
fim
```

O tempo de execução de MinHeapfy é proporcional a altura do heap. Sabemos que a altura de uma árvore é `h = log(n)` , sendo assim $T(MinHeapfy) = O(h) = O(logn)$

```
algoritmo heap_extract_min(V, n):
{-
  Entrada: Um heap mínimo V
  Saída: A raiz do heap.
-}
inicio
        se n < 1 então
            return 'Heap Vazio!'

        menor := V[1] -- Pegando a raiz.
        V[1] := V[n] --Move o último valor do Heap para raiz.
        n = n - 1 --Diminuindo o tamanho do heap
        MinHeapify(1) --Mantendo a propriedade heap

        retorne menor
fim
```

Temos que o tempo de remover o menor elemento é constante, porém precisamos manter a propriedade heap após sua remoção. Por isso chamamos `MinHeapify(1)` .

Logo `T(heap_extract_min) = T(MinHeapify) = O(logn)`

```
algoritmo k_esimo_menor(V,n,k)
{--Entrada: Vetor V
  Saida: k-ésikmo menor valor de V
--}
inicio
  montaMinHeap(V,n)
  para i = 1 até k-1 faça
    inicio
      heap_extract_min(V,n)
    fim
  retorneV[1] -- k-ésimo menor
fim
```

c. `COMPLEXIDADE`

Através do `slide 11` sabemos que a complexidade para `MontaMinHeap` é $O(n)$ pois só muda pelo fato que queremos a raiz com o menor elemento. Temos que o `para` executa $\Sigma[1 \leq i \leq k-1] = ((k-1) - 1 + 1) = (k-1) * T(heap_extract_min)$ = `(k-1) * log(n)`, ficamos com `T(montaMinHeap) + O((k-1) * log(n))`

T(k_esimo_menor) = `O((k-1) * log(n))`

Tomando $f(n) = (k-1) * logn$ e $g(n) = n * logk$
Limite n→∞ = $f(n)/g(n)$ = 0
Logo temos $O(g(n)) = O(n * logk)$

---

2. Questão

   a. `IDEIA`

      Podemos usar um `heap-min` para minimizar os custos das conexões entre os fios da seguinte forma:

      Defirnir uma variável `custo_total` a qual irá armazenar o custo total das conexões iniciando com 0.

      Definir uma outra variável `conectados` que irá armazenar o valor da conexão entre dois fios naquele momento, também iniciando com 0. Ou seja, `conectados` armazena `fio1` + `fio2` sendo que esses dois elementos são os dois menores de nosso `heap-min`, em seguida precisamos incrementar essa soma em `custo_total` e colocar esse novo fio conectado novamente em nosso heap-min.

      Resumindo, vamos tirando pares de menores fios em nosso heap-min, `fio1, fio2` somando-os e incrementando no custo total, depois precisamos inserir o custo de `fio1 + fio2` dentro do `heap-min`. Precisamos repetir esse procedimento enquanto o tamanho do `heap-min` seja > 1. Pois caso contrário, não temos como tirar um par de fios.

   b. `ALGORITMO`

```
algoritmo conecta_fios(F, n)
{-Entrada: Vetor F contando n fios.
 Saida: Cuso da conexão entre os fios.}
inicio
  montaMinHeap(F,n)
  custo_total := 0
  conectados := 0
  enquanto (n > 1) faça
    inicio
      conectados := heap_extract_min(F,n) + heap_extract_min(F,n)
      custo_total = custo_total + conectados
      InsereMinHeap(F,n,conectados)
    fim
  retorne custo_total
fim
```

c. `COMPLEXIDADE`

Temos que acomplexidade de `montaMinHeap` $O(n)$ *(slide 11)*, as linhas 2 e 3 são $O(1)$.
Nossa complexidade está dentro do `enquanto`, na primeria linha do `enquanto` temos que quando removemos um elemento do nosso `heap` precisamos rearranjar isso custa `O(logn)` *(custo do heap_extract_min)*. Logo, `heap_extract_min(F,N) + heap_extract_min(F,n)` = $2 * O(logn)$, outra parte importante é na hora de inserir o fio conectado em nosso heap, isso custa também $O(logn)$, pois também precisamos chamar `minHeapify`. Logo dentro do `enquanto` temos $2 * O(logn) + O(logn) = O(logn)$. A cada passada no corpo do `enquanto` removemos 2 elentos e inserimos 1 em nosso heap-min, na prática estamos removendo apenas um, além disso temos que ao final do `enquanto` existirá um elemento no heap, sendo assim o `enquanto` será executado `n-1` onde `n` é o tamanho do heap-min. Logo, $\Sigma[1 \leq i \leq n-1]O(logn) =$
$(n-1-1+1) * [O(logn)] =$
$(n-1) * O(logn) = O(nlogn)$

---

3. Questão

a. `IDEIA`

Para resolver esse problema vamos usar duas funções: `extraiPalavra` que irá receber um texto e irá retornar uma lista de tuplas da seguinte forma `(OCORRENCIA, palavra)`. Ou seja, o número que aquela palavra apareceu no texto e a respectiva palavra. Por exemplo: `"HEAP HEAP ALGORITMO ALGORITMO MAX"` nossa função `extraiPalavra` retornaria o seguinte: `[(2, "HEAP"), (2, "ALGORITMO"), (1,"MAX")]`. Para contabilizar as ocorrencias de cada palavra, podemos usar uma `tabelaHash`. Em seguida, basta retornar uma lista de tuplas contendo o par `valor, chave` da tabela `tabelaHash`. Essa lista de tuplas será ordenada pelo `heapSort` levando em consideração a quantidade de ocorrências. Sendo assim, precisamos de uma função que recebe uma tupla e retorna o primeiro elemento da tupla *(ocorrência)*.

```
algoritmo ExtraiOcorr(TUPLA)
{-Entrada: Uma tupla do tipo (VALOR, STRING).
 Saida: primeiro elemento da tupla: VALOR
inicio
```

```
  retorne TUPLA.primeiroElemento
fim
```

Para isso precisamos fazer apenas uma pequena alteração em `MaxHeapify` no slide 11 (pg 24).

```
se esq <= n and ExtraiOcorr([esq]) > ExtraiOcorr(A[i]) então
  maior:= esq
senão maior:= i
se dir <= n and ExtraiOcorr(A[dir]) > ExtraiOcorr(A[maior]) então
  maior := dir
```

Isso não muda a complexidade de `MaxHeapify` pois `ExtraiOcorr` é `O(1)`. Apenas estamos mudando o fator comparativo.

b. `ALGORITMO`

```
algoritmo extraiPalavras(TEXTO, n)
{-Entrada: Uma string contendo várias palavras.
 Saida: lista de tuplas do tipo (ocorrencia, palavra).}
inicio
  ocorrencia_palavras := Criar_TabelaHash()
  listaTuplas := []
  para p = palavra até n
    inicio
      se(ocorrencia_palavra[p] <> Null) então
        ocorrencia_palavra[p] := ocorrencia_palavra[p]+1
      senao
        ocorrencia_palavra[p] = 1
    fim
  para i=1 até ocorrencia_palavra.tamanho:
    inicio
      listaTupla.inserir(ocorrencia_palavra[valor], ocorrencia_palavra[chave])
    fim
  retorne listaTuplas
fim
```

```
algoritmo maisFrequentes(TEXTO, n)
{--Entrada: Uma string de tamanho n
   Saida: As duas palavras que mais aparece em TEXTO--}
inicio
  palavras_extraidas := extraiPalavras(texto, n)
  HeapMax.heapSort(palavras_extraidas)
  retorne (palavras_extraidas[n-1], palavras_extraidas[n])
fim
```

c. `COMPLEXIDADE`

a. Temos que afunção `extraiPalavras` na linha 1 e linha 2 é `O(1)` + `O(1)`.

No primeiro `for` temos que seu corpo irá executar proporcional ao tamanho do texto. Logo, $\Sigma[1 \leq i \leq n]C$ =
$C\Sigma[1 \leq i \leq n](n - 1 + 1) =$
$C(n) = O(n)$

No segundo `for` temos que também seu corpo irá executar no pior caso n vezes para cada chave da tabela hash. Logo temos:
$\Sigma[1 \leq i \leq n]C$ =
$C\Sigma[1 \leq i \leq n](n - 1 + 1) =$
$C(n) = O(n)$

$T(extraiPalavras)$= O(1) + O(1) + O(n) + O(n) = $O(n)$

Precisamos usar uma vez a função `extraiPalavras` dentro de `maisFrequntes`.
Sendo assim $T(maisFrequentes)$ = $T(extraiPalavras)$ + $T(heapSort)$ + $O(1)$.
No *slide 11*, vimos que a complexidade do heapSort é $O(nlogn)$ sendo assim temos:
$T(maisFrequentes)$ = $O(n) + O(nlogn) + O(1)$. Como a função $nlogn$ assintoticamente domina n, temos que$T(maisFrequentes) = O(nlogn)$

---

4. Questão

a. `IDEIA`

Podemos usar o `heapSort` e `tabelaHash` para resolver esse problema. Primeiro precisamos de uma função que irá contabilizar cada palavra e outra para impressão.
Para contabilizar as palavras precisamos de uma `tabelahash` , sendo que essa tabela irá conter a palavra e uma lista de todas as páginas onde a mesma aparece, exemplo: `"sorting":[70, 90, 190, 200]`.

b. `ALGORITMO`

```
algoritmo contabilizaPalavras(LISTA_T, n)
{-Entrada: Lista de tuplas do tipo (palavra,pagina).
 Saida: Um ponteiro para uma tabelaHash do tipo:
b => [1,2,3...]
```

```
a => [100,200]
c => [100,200]
...
}
inicio
  dicionario := Criar_TabelaHash()
  heapSort(LISTA_T) --Numero da página como critério de ordenação
  para i=1 até n:
    inicio
      se(dicionario[i].palavra != Null) então
        dicionario[i].paginas.adicione(pagina)
      senão
        dicionario[i].palavra = palavra
        dicionario[i].paginas.adicione(pagina)
    fim
  retorne dicionario
fim
```

Complexidade de `contabilizaPalavras`: A operação de criar a Tabela é O(1) além disso temos que o heapSort tem complexidade O(nlogn) *(slide 11),* No `para` temos que as operações realizada em seu corpo são elementares, ou seja, C.

Logo, $\Sigma[1 \leq i \leq n]C$= $C\Sigma[1 \leq i \leq n](n-1+1) = O(n)$.Temos que T(contabilizaPalavras) = $O(1) + O(nlogn) + O(n)$, como afunção $nlogn$ domina assintoticamente todas as outras, segue que $T(contabilizaPalavras) = O(nlogn)$.

```
algoritmo imprimeOrdAlf(TABELA_H, n)
{-ENTRADA: Uma tabela Hash do tipo:
b => [1,2,3...]
a => [100,200]
c => [100,200]
SAÍDA: Representação das palavras em ordem alfabética
b => [1,2,3...]
a => [100,200]
c => [100,200]
}
inicio
  palavra_paginas := [] --vetor de tuplas do tipo [(a, [100,200])]
  para i=1 até n faça
    inicio
      palavra_paginas.adicione((TABELA_H[i].palavra,TABELA_H[i].paginas))
  fim
  heapSort(palavra_paginas) ----Palavra como critério de ordenação
  para i=1 até n:
    imprima palavra_paginas[1], palavra_pagina[2]
fim
```

Complexidade de `imprimeOrdAlf`:A primeira linha O(1).
Precisamos adicionar os pares (palavra, paginas) dentro do vetor `palavra_paginas` .Para isso usamos um laço `para` com operações elementares em seu corpo: $C\Sigma 1 \leq i \leq n = O(n)$.

Logo, em seguinda precisamos ordenar `palavra_paginas` em ordem alfabetica para isso usamos o heapSort com custo `O(nlogn)` *(slide 11).*

Por fim, precisamos mostrar o contéudo de `palavra_paginas` ordenado, para isso utilizamos um laço simples. $C\Sigma 1 \leq i \leq n = O(n)$.

$T(imprimeOrdAlf) = O(n) + O(nlogn) + O(n)$. Novamente como $nlogn$ assintoticamente domina as outras funções temos que $T(imprimeOrdAlf) = O(nlogn)$.

Como precisamos da composição dessas duas funções para resolver o problema a complexidade total seria $O(nlogn) + O(nlogn) = 2O(nlogn) = O(nlogn)$.

---

5. Questão

`IDEIA`

Primeiro precisamos saber como estruturar esse cadastro. Adotei a seguinte maneira, uma matriz que irá armazenar o cadastro de cada município. Sendo que o cadastro de cada cidadão em um determinado município será uma coleção de tuplas no seguinte formato:

```
[
   [
    (CPF, NOME, DATA_NASCIMENTO, MUNICIPIO, ESTADO, VACINA, DATA_DOSE1, DATA_DOSE2),
    (CPF, NOME, DATA_NASCIMENTO, MUNICIPIO, ESTADO, VACINA, DATA_DOSE1, DATA_DOSE2)...
    ],

    [
    (CPF, NOME, DATA_NASCIMENTO, MUNICIPIO, ESTADO, VACINA, DATA_DOSE1, DATA_DOSE2),
    (CPF, NOME, DATA_NASCIMENTO, MUNICIPIO, ESTADO, VACINA, DATA_DOSE1, DATA_DOSE2)...
    ]
]
```

Podemos usar o `k-Way-Merge` para gerar um cadastro único dos cidadãos vacinados pois sabemos que os cidadãos estão ordenados por CPF. Assim, como fizemos na questão 3, podemos mudar o fator comparativo de nosso `minHeap` para ser o primeiro elemento da tupla. Ou seja, o `CPF`. Isso seria `O(1)` pois é uma função que recebe uma tupla e retorna sua primeira posição.

`ALGORITMO`

```
algoritmo k-Way-Merge-SE()
{-SAÍDA: Vetor V ordenado por CPF contendo um cadastro unificado
         dos cidadãos lidos apartir do disco rígido.
}
inicio
  MATRIZ[n] := lerDoDisco(ArquivoCadastro)

  minHeap := MontaMinHeap()
  arrUnificadoOrd := []

  para i=1 até n faça
    insereMinHeap((MATRIZ[i].pop_primeiroElemento(), i))

  enquanto minHeap.tamanho > 0 faça
    inicio
      tupla := minHeap.heap_extract_min() -- Uma tupla (elemento, index)
      elemento := tupla[1] -- Elemento
      index := tupla[2] -- Indice do array do qual o elemento saiu.
      arrUnificadoOrd.insira(elemento)

      se (MATRIZ[index] <> Null) -- Caso haja elementos ainda
        insereMinHeap((MATRIZ[index].pop_primeiroElemento(), index))
    fim
  retorne arrUnificadoOrd
fim
```

`COMPLEXIDADE`

`T(MontaMinHeap) = O(n)` *(slide 11)*, o primeiro para tem complexidade Σ1≤i≤n * long(n)

`n*log(n)`, pois para cada município estamos pegando apenas seu primeiro elemento.

O `enquanto` será executado enquanto existir elementos em nosso heap, ou seja `n` vezes *(número de cidadãos)*, sendo que dentro do `enquanto` temos `T(minHeap.heap_extract_min) = log m`, onde `m` é o número de municípios. T(k-Way-Merge-SE) = `n*logm`

6. Questão

`IDEIA`

Podemos usar a ideia do algoritmo de ordenação por contagem, porém não precisamos ordenar os dados. Apenas contabilizar a frequencia de quantos cidadãos estão vacinados até aquela idade.
Como a ordenação por contagem utiliza uma array de frequencias, temos que cada índice do array será umadeterminada idade e seu valor será quantos cidadãos foram vacinados `até aquela idade`. Logo basta pegar a frequencia da idade máxima e subtrair da idade mínima. Assim temos, o total de vacinados naquela idade. *(Aqui adotei o critério de se o cidadão tomou as duas doses, então está vacinado.)*

`ALGORITMO`

```
algoritmo contaBilizaVacinados(dadosUnificados, n)
{-
Entrada Vetor dadosUnificados com todos os cidadãos do estado de sergipe.
SAÍDA: Vetor V de inteiros com todas as idades de cidadãos que foram vacinados
-}
inicio
  V := [] -- Vetor
  parar i = 1 até n faça
    inicio
      se(estaVacinado(dadosUnificados[i]) então -- cidadão está vacinado?
        vacinados.insira(dadosUnificados[i].idade)
    fim
  retorne V
fim
```

Temos que no pior caso a instrução `insira` será executada n vezes. Ao utilizar uma fila ou pilha, temos que `insira` será `O(1)`.
Logo, temos `T(contabilizaVacinados)` = $C\Sigma 1 \leq i \leq n = O(n)$

```
algoritmo contSortIdades()
{-
SAÍDA: Vetor C com as frequencias das idades vacinadas
       onde os índeces são as idades e os valores são quantos foram vacinados
        até aquela idade.
-}
inicio
  vacinados := contaBilizaVacinados(dadosUnificados, n)
  C:=[200] -- 200 será a idade máxima

  para i=1 até 200 faça -- inicializa as frequencias
    inicio
      C[i] = 0
    fim
  para j = 1 até vacinados.tamanho faça --Contabilizando as frequencias
```

```
    inicio
      C[vacinados[j]] := C[vacinados[j]] + 1
    fim

  para i=2 até 200 faça --Contando os valores
    inicio
      C[i] = C[i] + C[i-1]
    fim

  retorne C
fim
```

No `para` de inicilizar as frequencia temos: $C1 * C2\Sigma 1 \leq i \leq 200 = C$,
`para` de contabilizar as frequencias: $C\Sigma 1 \leq i \leq n = O(n)$
`para` de contar os valores: $C3 * C4\Sigma 2 \leq i \leq 200 = C5$. Logo temos,
`T(contSortIdades) =`

`T(contaBilizaVacinados) + C + O(n) + C5 =`

`O(n) + C + O(n) + C5 =` $O(n)$

```
algoritmo totalPorFaixa(idadeMin, idadeMax)
{-
Entrada Recebe dois inteiros representando a faixa de idade.
Sapida: Inteiro t informando quantos estão vacinados nessa faixa.
-}
inicio
  C := contSortIdades()
  retorne C[idadeMax] - C[idadeMin]
fim
```

Temos que na primeira execução de `totalPorFaixa` temos:

`T(totalPorFaixa)` = `T(contSortIdades) + C` = `O(n)`