Micael Andrade Dos Santos

1. Questão

   S : $T(n) = 2T(n/2) + n, T(1) = 0$

   Podemos usar aqui o teorema mestre, pois a recorrência está no formato $T(n) = aT(n/b) + f(n)$

   Logo, $a = 2, b = 2, f(n) = n$. Podemos usar o `caso 2` do teorema o qual nos diz: `se f(n) ∈ Θ(n ^ logb(a)) então T(n) ∈ Θ(n ^ logb(a) * log(n))`.

   Temos que `f(n) ∈ Θ(n ^ logb(a))` pois: `C1n^logb(a) ≤ n ≤ C2n^logb(a)`, ou melhor, C1n ≤ n ≤ C2n para C1 e C2 = 1.

   Sendo assim, T(n) ∈ Θ(n ^ logb(a) * log n) = $\Theta(nlog(n))$ ou simplesmente $O(nlog(n))$

   t : $T(n) = 2T(n/4) + n^{1/2}, T(1) = 0$ *(para n > 2 e para n ≤ 2, T(n) =c=O(1)*

   Temos que a = 2, b =4, f(n)=n^1/2

   Aqui podemos usar o teorema mestre, também o caso dois pois `f(n) ∈ Θ(n^logb(a)` visto que `C2√n ≤ √n ≤ C√n` , para `C1 e C2 = 1`. Logo, `T(n) ∈ Θ(n^logb(a) log(n))` ou $O(\sqrt{n} * log(n))$.

   Sendo assim, a solução **t** seria a melhor escolha.

---

2. Questão

   a. `IDEIA`

      Temos que o vetor está ordenado e sem elementos repetidos. Sendo assim a quantidade de rotações seria a **posição do menor elemento.** Pois, a cada rotação o elemento menor será deslocado para a próxima posição. Como temos a certeza de que o vetor original está ordenado *(Temos uma sequência cíclica)*, podemos utilizar um busca-binária para encontrar o index do menor elemento. Temos que se $R[meio] < R[alto]$ o índice do menor elemento não está entre $meio < i \leq alto$. Por outro lado, se$R[meio] > R[alto]$então o índice está $meio < i \leq alto$. Por fim, basta retornar o valor da variável $meio$.

   b. `CODE`

```
algoritmo encontraRot(R, n)
{-Entrada: Vetor R com n elementos sendo que R sofreu K rotações.
 Saida: Inteiro K -}
inicio
  baixo := 1
  alto := n

  enquanto(baixo <= alto) faça
    meio := ⌊ (baixo + alto) / 2⌋
    se (R[meio] < R[alto]) então
      alto := meio
    senão
      baixo :=meio+1
  retorne meio
fim
```

c. `COMPLEXIDADE`

Podemos assumir que há $n = 2^k$ elementos na lista em que k é um número inteiro não negativo. Note que $k = log(n)$. (Caso o número de elementos da lista não seja uma potência de 2, a lista será parte de uma lista maior com 2^(k+1) elementos em que 2^k < n < 2^(k+1).

Em cada etapa do algoritmo, `baixo` e `alto`, as localizações do primeiro e último termo da lista são comparadas para ver se a lista não chegou ao fim.

Temos que no laço `enquanto` *1(atribuição)* + *1(comparação)* + *1(operação)*

A primeira e tapa da busca é restrita à lista com 2^(k-1) termos. Até aqui três operações foram realizadas. Este procedimento continua, usando três comparações em cada etapa par restringir a busca a uma lista com metade dos termos. Ou seja, três operações são usadas na primeira etapa do algoritmo, quando a lista tem 2^k elementos, mais três quando a busca foi reduzida à lista com 2^(k-2) elementos, e assim por diante até que três comparações sejam usadas quando a busca for reduzida à lista com 2^1 = 2 elementos.

Logo, pelo menos $3K + 2 = 3logn + 2$ comparações são realizadas para realizar a busca do menor elemento quando a lista tiver 2^k elementos. Sendo assim, temos no máximo $\Theta(logn)$ comparações, ou simplesmente $O(logn)$.

---

3. Questão

a. `IDEIA`

Podemos estruturar a solução em duas etapas. Primeiro seria utilizar uma variação da busca binárias para encontrar o primeiro valor inteiro o qual $f(x) >= 0$. Sendo que $a <= x <= b$, para valores positivos a e b. Ao encontrar o valor $meio = \lceil (a + b)/2 \rceil$ temos que: se $f(meio)$ for negativo, podemos eliminar o range $[a, meio]$, caso $f(meio)$ seja positivo podemos restringir para$[a, meio - 1]$, caso contrário encontramos o x. Também precisamos do intervalo a, b onde x está. Para isso podemos criar uma função que irá `estimar` o intervalo.

`CODE`

```
algoritmo encontraX(a, b)
{-Entrada: Intervalo [a, b] sendo a e b > 0.
 Saida: Inteiro c contido em a,b tal que para uma função monoticamente
crescente f(x), f(c) >= 0 -}
inicio
  baixo := a
  alto := b

  enquanto(baixo <= alto) faça
    inicio
      meio := ⌈(baixo+alto)/2⌉
      se (fdeX(meio) < 0 então
        baixo := meio
      senão
        se(fDX(meio) > 0 então)
          alto := meio - 1
        senão
          retorne meio
    fim
  retorne meio
fim
```

```
algoritmo encontraAB(b)
{-Entrada: b começa valendo 1
 Saida: Dois inteiros A,B sendo o intervalo onde para uma função
f(x) ∃C | A<=C<=B  onde f(C) >= 0-}
inicio
    se(fdeX(b) >= 0) então
        retorne encontraX(b//2, b)
    senao:
        retorne encontraAB(b*2)
fim
```

```
algoritmo fdeX(x)
{-Entrada: Valor inteiro
  Saída um inteiro y
-}
```

```
inicio
   retorne 6*x-9000
fim
```

`COMPLEXIDADE`:

As operações críticas da função `encontraX` está na estrutura de repetição `enquanto`. Sua complexidade foi calculada na questão 3, sendo O(log n). Temos que `encontraAB` tem complexidade O(log n) como veremos logo abaixo. Logo, O(log n) + O(log n) = 2O(log n) ou apenas O(log n).

Temos que $T(n*2) + 1 = T(n/4^{-1/2}) + 1$ e $T(p(x) \geq 0) = 1$

Utilizando o teorema mestre tem que $a = 1, b = 4^{-1/2} = 1/2$, f(n) = 1

Temos que $f(n) \in \Theta(n)$pois `C1*n^(logb(a)) ≤ 1 ≤ C2*n^(logb(a))`. Logo pelo teorema mestre `T(n) ∈ Θ(n^logb(a)* log n)` ou `O(log n)`

---

4. Questão

   a. `1º IDEIA`

   A ideia consiste em verificar se o índice é `par` ou não, e comparar o elemento do índice atual com seu predecessor. Se o estivermos em um índice `par` e o elemento desse índice for menor que seu predecessor então realizamos uma troca. Caso o índice seja ímpar e o elemento desse índice, seja maior que o seu predecessor também precisamos realizar uma troca.

   b. `1º CODE`

```
algoritmo inLocoPar(V, n)
{-Entrada: Vetor V com n elementos.
 Saida: Vetor V com n elementos.}
inicio
  para i = 2 até n faça
    inicio
      se(i mode 2 = 0) então
        se (V[i] < V[i-1])
          troque(V[i], V[i-1])
      senão
        se(V[i]>V[i-1]) então
          troque(V[i], V[i-1])
    fim
fim
```

   c. `1ºCOMPLEXIDADE` *1(comparação) + 1(comparação) 1(uma operação básica de troca) = C*. A função `troque` manipula os elementos através de ponteiro, então é considerada constante. Essas três operações no laço, são realizadas $C\sum[2 <= i <= n] = (n-1)$, ou seja, $C\sum n.\ O(n)$.

---

a. `2º IDEIA`

Podemos criar uma cópia do vetor original e utilizar duas variáveis, `esq` e `dir`, sendo que `esq` irá percorrer os valores do menor para o maior, e `dir` irá percorrer os valores do maior para o menor. Quando encontrarmos um índice par, podemos pegar o último valor do vetor ordenado e inserir nessa posição, caso contrário pegamos o menor.

b. `2 ºCODE`

```
algoritmo trocaPar(V, n)
{-Entrada: Vetor V com n elementos.
 Saida: Vetor V com n elementos.}
inicio
  W := copiar(V)
  selectionSort(W)
  esq := 1
  dir:= n
    para i = 1 até n faça
      inicio
        se i mode 2 = 0 então
          V[i]:=w[dir]
          dir := dir - 1
        senão
          V[i] := W[esq]
          es:= esq + 1
      fim
  retorne V
fim
```

```
algoritmo selectionSort(V, n)
{-Entrada: Vetor V desordenado
 Saida: Vetor V em ordem crescente.}
inicio
    para i = 1 até n-1 faça
      inicio
        menor := i
        para j = i + 1 até n faça
          inicio
            se (V[j] < V[menor]) então
              menor := j
          fim
        troque(V[i], V[menor])
      fim
  retorne V
fim
```

c. `2º COMPLEXIDADE`

Temos que a complexidade de `trocaPar` é linear, pois, $C \sum[1 <= i <= n] = (n - 1 + 1)$ logo $O(n)$.
Já a complexidade do `selectionSort` é

$(\sum[1 <= i <= n-1] * \sum[i+1 <= j <= n])C =$

$C \sum[1 <= i <= n-1](n-(i+1)+1)$=

$C \sum[1 <= i <= n-1](n-i) =$

$C[1+(n-3)+(n-2)+(n-1)]$=

$[n(n+1)/2] - n$ = $(n^2+n-2n)/2 = (n^2-n)/2$ =

$O(n^2)$

Logo, temos que $O(n^2) + O(n)$ = $O(max(n^2, n))$ = O(n^2) para segunda solução. Temos que a primeira solução seria a melhor escolha, pois consome $O(n)$

---

5. Questão

a. `1º IDEIA`

Podemos usar o `mergeSort` para ordenar o array e depois multiplicar os dois últimos termos e os dois primeiros termo e retornar o maior produto.

b. `1º CODE`

```
algoritmo merge(Y, l, m, r)
{- Entrada: vetor Y com indicadores l, m e r
saída: vetor Y de dados ordenados entre l e r -}
início
  para i = l até m faça -- C.n/2
    A[i] := Y[i]
  para j = m+1 até r faça -- C.n/2
    B[j-m] := Y[j]
  tamA := m-l+1;
  tamB := r-m;
  i := 1;
  j := 1;
  k := l;
  enquanto (i <= tamA) && (j <= tamB) faça -- n
    início
      se A[i] <= B[j]
      então
        Y[k] := A[i]
        i := i+1
      senão
        Y[k] := B[j];
        j:=j+1
```

```
    fim
  k := k+1;
  enquanto (i <= tamA) faça --n
    Y[k] := A[i]
     i := i+1
     k := k+1
  enquanto (j <= tamB) faça --n
   Y[k] := B[j]
   j := j+1
   k:= k+1
fim
```

```
algoritmo mergeSort(V, l, r)
{-Entrada: vetor V de dados arbitrários com limites l e r
saída: Vetor V de dados ordenados.}
inicio
    se (l < r) então
      m := ⌊(esq+dir)/2⌋
    senão
      mergeSort(V, l, r);
      mergeSort(V, m+1, r);
      merge(V, l, m, r);
    retorne V
fim
```

```
algoritmo maxProduct(V, n)
{-Entrada: Vetor V ordenado com n elementos.
 Saida: Inteiro k }
inicio
    se (n = 1) então
      retorne V[1]
    senão
      primeirosMax := V[1] * V[2]
      ultimosMax := V[n] * V[n-1]
      se(primeirosMax > ultimosMax) então
        retorne primeirosMax
      senão
        retorne ultimosMax
    retorne V
fim
```

---

a. `2º IDEIA`

   Podemos definir `4 variáveis` que irá armazenar os `dois maiores valores positivos` `e os dois menores valores`. Ao varrer o vetor iremos atualizando essas variáveis ao final da iteração basta retornar o maior produto desses pares de variáveis.

b. `2º CODE`

```
algoritmo maxProductLinear(V, n)
{-Entrada: Vetor V desordenado
 Saida: Inteiro k.}
inicio
    maiorPositivo_1 := 0
    maiorPositivo_2 := 0

    maiorNegativo_1 := 0
    maiorNegativo_2 := 0

    para i = 1 até n faça
      inicio
        se (V[i] > maiorPositivo_1) então
          maiorPositivo_2 := maiorPositivo_1
          maiorPositivo_1 := V[i]
        se não
          se(V[i] > maiorPositivo_2) então
            maiorPositivo_2 = V[i]

        se (V[i] < 0 && abs(V[i]) > abs(maiorNegativo_1)) então
                maiorNegativo_2 := maiorNegativo_1
                maiorNegativo_1 = V[i]
        se não
          se(V[i] < 0 && abs(V[i]) > abs(maiorNegativo_2)) então
            maiorNegativo_2 = V[i]
      fim

      proc_positivos := maiorPositivo_1*maiorPositivo_2
      proc_negativos := maiorNegativo_1*maiorNegativo_2

      se(proc_positivos > proc_negativos) então
        retorne proc_positivos
      senão:
        retorne proc_negativos
fim
```

c. `COMPLEXIDADE`

Essa abordagem utiliza apenas um laço para encontrar os maiores valores. Dessa forma tempos para um vetor com n elementos: $C\sum[1 <= i <= n] = C(n - 1 + 1) = O(n)$ uma abordagem bem mais eficiente do que a estratégia da ordenação.

---

6. Questão

a. `1º IDEIA`

Podemos criar uma variável que irá apontar para a próxima posição onde um elemento zero será inserido, inicialmente começamos no fim do array. Para cada elemento zero encontrado ao varrer o array, devemos encontrar sua posição adequada. Para isso utilizamos um `while` que irá encontrar essa posição.

b. `1º CODE`

```
algoritmo deslocaZeros(V, n)
{-Entrada: Vetor V com elementos zeros em qualquer posição
 Saida: Vetor V com elementos zeros do lado direito.}
inicio
    posiZero := n
    para i = 1 até n faça
      inicio
            se (V[i] = 0 && i < posiZero) então
              enquanto(V[posiZero] = 0 && posiZero != 0) faça
                inicio
                  posiZero:=posiZero - 1
                fim
              se(i<=posiZero) então
                troque(V[i], V[posiZero])
      fim
    retorne V
fim
```

c. `1º COMPLEXIDADE`

Temos que o ponto crítico do algoritmo será no corpo do `while`, o qual só irá executar se o elemento atual $a[i]$ for zero. Além disso, o `while` será executado varrendo o vetor da direita para esquerda *somente uma vez* tentando encontrar uma posição adequada para inserir o elemento 0.
Temos que no pior caso, quando temos um vetor do tipo [0, 0, 0, 0, ... , 0, 0] a complexidade seria:
$(\sum[1 <= i <= n]C2 + \sum[1 <= j <= n]C1) =$
$C(\sum[1 <= i <= n] + \sum[1 <= j <= n]) =$
$C(\sum[1 <= i <= n](n - 1 + 1) + \sum[1 <= j <= n](n - 1 + 1)) =$
$C(n + n) = C(2n) = O(n)$

---

a. `2º IDEIA`

Podemos mover todos os elementos não-zeros para a esquerda da seguinte forma:
Basta criar uma variável `contador` que irá armazenar a posição onde o próximo elemento não-zero será inserido, assim, cada vez que encontrarmos um elemento não-zero ele será inserido na posição `contador`. Começamos com o `contador` := 1. E cada vez que encontramos um elemento não-zero incrementamos o `contador`.

Finalmente precisamos preencher o intervalo `[contador, n]` com os zeros.

`2º CODE`

```
algoritmo deslocaZeros(V, n)
{-Entrada: Vetor V com elementos zeros em qualquer posição
 Saida: Vetor V com elementos zeros do lado direito.}
inicio
  contador := 1
  para i = 1 até n faça
    inicio
      se(V[i] <> 0) então
        V[contador] := V[i]
        contador := contador + 1
    fim

  para j = contador até n faça
    inicio
      V[j] = 0
    fim
  retorne V
fim
```

`2º COMPLEXIDADE`

Temos que no pior caso a expressão `V[contador] := V[i]` do primeiro laço será executada `n` vezes. E `V[j] = 0` será também executada n vezes, logo:

(∑ [1<= i <=n]C2 + ∑ [1<= j <=n]C1) =

C(n+n) = C(2n) = $O(n)$