Micael Andrade Dos Santos

Matrícula: 201900051051

1. Questão
   a. Estruturação por indução

      `CASO BASE`:

      Caso a o vetor A tenha apenas um elemento. Temos que se A[1] ≥ P então retorna 1 caso contrário -1.

      `HIPO`

      Suponha que saibamos ressolver o problema com a metade dos dados, ou seja, para ≤ teto(n/2) sei determinar o índice da maior paralavra menor ou igual a P.

      `CASO GERAL`

      Resolução do problema para n dados.

      Temos a seguinte estratégia: meio = teto((L+R)/2). Podemos definir dois ponteiros L e R sendo que quando eles se cruzarem verificamos se R ultrapassou o tamanho do vetor, logo retornamos -1. Caso contrária retornamos meio.

      Se vetor[meio] > P vamos para primeira metade dos dados, pois o vetor A está ordenado. (Aplico minha HIP)

      Se vetor[meio] ≤ Zvamos para segunda metade dos dados. (Aplico novamente minha HIP)

   b. Algoritmo

```
algoritmo BB_adaptada(A, n, p)
{- Entrada um vetor A de palavras ordenado detamanho n. E uma palavra p
Saída: Indice i talque A[i] é a maior palavra menor ou igual a p no vetor A
-}
Inicio
  retorne ache(A, 1, n, p)
fim
função ache(A, L R, p)
inicio
  meio := teto((L+R)/2)
  se ( R < L) então
    se (meio > n) então -- Estourou o tamanho de A
      retorne -1
  se (compareString(A[meio], p) = 0) ẽntão -- achou
    retorne meio
  senão
    se(compareString(A[meio], p) = 1) então
      retorne ache(A, L, meio -1, p)
    senão
      retorne ache(A, meio, R, p)
fim
```

   c. Fórmula de recorrência

      Temos que para cada chamada recursiva da função `ache` estamos passando metade dos valores. Ou seja, n/2, além disso podemos considerar as outras operações de comparação de string como tempo constante C. Logo ficariamos com a seguinte fórmula: T(1) = C e T(n/2) logo $T(n) = T(n/2) + C$

   d. Assumindo que n é potência de 2, ou seja, $n = 2^k$ e aplicando a função log de ambos os lados temos: $k = logn$. Resolvendoa a recorrência ficamos.

T(2^k) = T(2(k-1)) + C

= (T(2^(k-2))) + C) + C

= ((T(2(k-3)+ C) + C) + C) ← Opa! Temos um padrão aqui, a quantidade de C é a mesma do último termo do expoente.

....

T(2^(k-k)) + k

T(2^0) + K ← Sei calcular o caso base

C + K

Agora precisamos voltar para nossa variável de indução `n`, como k = logn, temos que C + logn = $O(logn)$

---

2. Questão

    a. Podemos representar as cores da seguinte forma: AMARELO = 1, AZUL = -1, VERDE = 3. Vamos utilizar um algoritmo de partição utilizado no quicksort para realizar essa tarefa.

    b. `Algoritmo`

```
algoritmo partition(V, l, r)
{- Entrada um vetor V de cores com n cores (Amarelo, Verde e Azul), um ponteiro do primeiro elemento e para o útimo.
Saída: Vetor A tal que todas cores azuis estão à esquerda e todas cores verdes estão à direita.
-}
inicio
  pivo := escolhaPivo(V) -- Escolhe o elemento 1 como pivô (amarelo)
  L := l
  R := n
  enquanto L < R faça
    enquanto (V[L] <= pivo) e (L <= r) faça L := L+
    enquanto (V[R] > pivo) e (R >= l) faça R := R-1
  se (L < R) então Troca(V[L], V[R])
  pos := R
  Troca(V[l], X[pos])
  retorne V
fim
```

    c. `Complexidade`

    Temos que escolher um elemento Amarelo como pivô, para fazer isso uma simples função linear `O(n)` pode ser usada para retorna essa posição. Além disso o algoritmo de partição percorre os elementos apenas uma vez. Ou seja também será `O(n)`. Logo `T(partition) = O(n)`

3. Questão

    a. Podemos usar o heapSort para ordenar o vetor P de palavras e para cada palavra do vetor CH verificar se a mesma está em P. Dessa forma, podemos ir pegando as palavras de CH e verificando se estão em P.

    b. `Algoritmo`

```
algoritmo relevante(P, n, CH, m, k)
{- Entrada um vetor P de palavras pre-processadas de tamanho n e um vetor CH de tamanho m e um inteiro k.
-}
inicio
    totalEncontrados := 0
    Heapsort(A,n) -- Podemos usar uma função O(1) para mudar o critério de comparação.
    ponteiro = 1
    flag = 1
    para i=1 ate m faça
        enquanto (poteiro <= n && flag <> 1)
```

```
          se (compareString(CH[i], A[ponteiro]) = 0) então
            totalEncontrados := totalEncontrados + 1
            ponteiro := poteiro + 1
            se (compareString(CH[i], A[ponteiro]) = 1) então
              flag = 0 -- para o laço
          se totalEncontrados = k então
           imprima 'Texto Relevante'
            flag = 0
        imprima 'Texto Inrelevante' -- Chegou ao fim e não encontrou um total maior que k
fim
```

d. `Complexidade`

Temos que o Heapsort custa `O(nlogn)` para ordenação. Alem disso temos Σ(i=1)(m) * T(enquanto) para cada palavra em CH. Σ(i=1)(m) = m - 1 + 1 = m * T(enquanto). Dentro do enquanto nós temos um ponteiro e uma flag que irá controlar seu fluxo. Sendo assim, quando estivermos em uma palavra que é menor a qual estamos procurando não adianta continuar percorrendo, pois A está ordenado. T(relevante) = O(nlogn) + m*n = `O(nlogn)`.

4.

`Algoritmo`

```
algoritmo KMP_algoritmo(T, N, P, M)
{-Entrada: Texto T de tamanho N e um padrão P de tamanho M.
 -SAÍDA: Ultima ocorrência de P em T caso exista, -1 caso contrário.
.}
inicio
    i := 1
    j := 1
    ultimaOcorr := -1
    ARR_PRE := ComputaNext(P,M) -- pré-processa next(não precisa alterar)
    enquanto i <= N: --Até o tamanho do texto.
        se(T[i] = P[j]) então
            i := i + 1
            j := j + 1
        senão:
            se(j <> 0) então
                j := ARR_PRE[j]
            senão:
                i := i + 1
        se(j = M) então -- Deu matched
            ultimaOcorr = i-j
            j = ARR_PRE[j]

    retorne ultimaOcorr
fim
```

`não deu tempo` 😢