

Nome: Pedro Lemos Zatke

**Atenção:** As questões que solicitarem o projeto de um algoritmo deverão conter em sua resposta (1) o pseudocódigo do algoritmo, ou (2) uma descrição textual sem ambiguidade com os passos necessários para implementá-lo.

### Questão 1 (2 pontos)

Determine uma função  $f(n)$  para o código a seguir tal que  $T(n) = \theta(f(n))$ . Explique como a solução foi encontrada.

Semelhante ao mergesort!

```

HashTable t;

void f(int v[], int startInx, int endInx) {
    if (startInx < endInx - 1) { → se Tamanho > 1
        int midInx = (startInx + endInx) / 2; → metade
        f(v, startInx, midInx); → recurrivamente →  $\frac{m}{2}$ 
        f(v, midInx, endInx); → frequências

        for (int i=startInx; i < endInx; i++) { } item em m → O(m)
            t[v[i]]++; → faz contagem de
        } → frequências

        int r[endInx - startInx]; → tamanho m, vetor auxiliar
        int aInx = startInx;
        int bInx = midInx; → indireto para andar na esquerda e na direita
        int rInx = 0;

        while (aInx < midInx && bInx < endInx) { → selecionar o menor entre as b
            r[rInx] = v[aInx] <= v[bInx] ? v[aInx++] : v[bInx++]; → a vermelha
            r[rInx++] += t[r[rInx]]; → marron em r
        } → marron em r
        (a → primeiro metade)
        (b → segundo metade)

        while (aInx < midInx) { } → terminou b, fiz 2
            r[rInx] = v[aInx++]; → terminou a, fiz b
            r[rInx++] += t[r[rInx]]; → (não fiz m das duas)

        while (bInx < endInx) { } → terminou b, fiz 6
            r[rInx] = v[bInx++]; → terminou a, fiz 6
            r[rInx++] += t[r[rInx]];
    }
}

```

itens em m → O(m)

```

for (aInx = startInx; aInx < endInx; ++aInx) { iter pelo bloco de memória.
    v[aInx] = r[aInx - startInx];
}
} if
void
    
```

→ insere o que está em  $r$  no array principal

## Questão 2 (1 ponto)

A seguir são apresentados cenários que exigem a escolha de um algoritmo de ordenação. Dentro os algoritmos apresentados em aula, proponha um adequado para cada cenário, buscando a maior eficiência de tempo e espaço. Explique a sua escolha.

- Uma estação meteorológica precisa manter continuamente ordenados os  $n$  maiores valores de temperatura em um sistema com memória limitada. Os dados chegam em tempo real, incluindo temperaturas com muitas casas decimais em uma ampla faixa de valores, e o conjunto é atualizado frequentemente. É essencial que o algoritmo apresente complexidade  $O(n \log n)$  no pior cenário.
- Uma empresa de logística precisa ordenar 10 milhões de códigos de rastreio de encomendas. Cada código consiste em 12 dígitos numéricos (por exemplo, 202407260134). A empresa precisa ordenar esses códigos frequentemente para otimizar as rotas de entrega e o processo de triagem nos centros de distribuição. O sistema tem memória suficiente para processar todos os códigos de uma vez.

## Questão 3 (2 pontos)

Uma empresa de análise de redes sociais, que possui uma base de dados com milhões de usuários, está desenvolvendo um algoritmo para identificar influenciadores emergentes. Além de informações cadastrais, em cada usuário é armazenado o seu número de seguidores e uma pontuação representando o seu índice de engajamento, calculado com base em uma fórmula envolvendo likes, comentários e compartilhamentos. Um influenciador emergente é um usuário que está no top 10% em termos de engajamento, porém não está no top 10% em número de seguidores. Projete um algoritmo eficiente que, dada uma sequência  $A$  com  $n$  tuplas  $<id\_usuário, seguidores, engajamento>$ , identifique todos os influenciadores emergentes. O algoritmo deverá apresentar uma complexidade  $O(n)$ .

Caso seja necessário, é permitido nessa questão fazer referência a algoritmos apresentados em aula para compor a solução.

## Questão 4 (5 pontos)

Uma sequência  $A$  de tamanho  $n$  contém inteiros positivos e negativos.

- Projete um algoritmo  $O(n^3)$  capaz de determinar os índices  $i$  e  $j$ , sendo  $i < j$  tal que a soma de  $A[i] + \dots + A[j]$  é máxima.
- Otimize a solução do item anterior produzindo um algoritmo  $O(n^2)$ .

- c) Avalie se é possível produzir um algoritmo  $O(n)$  para esse problema. Caso seja possível, apresente o algoritmo, caso contrário prove que é impossível.

Nessa questão não é permitido apenas fazer referência a algoritmos apresentados em aula para compor a solução. Tais algoritmos podem ser utilizados, no entanto devem ser descritos explicitamente.

- ① Podemos observar logo de cara que o algoritmo faz uso de uma hash table. Como foi estudado que, no caso médio, hash tables realizam operações de inserção e busca em tempo constante  $O(1)$ , esse tempo será considerado ímpar.
- caso base  $O(1)$
- Algoritmo: Vamos de tudo verificar se a entrada tem tamanho 1, se tiver muito é feito. Para entradas maiores, o que é feito é:
- Partir os vetos em dois ma métode (sempre ma métode) e fazer uma chamada recursiva tanto na esquerda quanto na direita (2 chamadas com  $\frac{n}{2}$  em cada)
  - Iterar pelo vetor (após as chamadas recursivas) e fazer uma contagem de frequências dos elementos usando uma hash table. Como a hash table aparente modificações em  $O(1)$ , essa etapa tem complexidade  $O(n)$  sempre (faz  $O(1)$  n vezes)
  - criar um vetor auxiliar de tamanho m, chamar de auxiliar de complexidade desejado e na metade direita do array, e inserir elementos presentes no vetor auxiliar. São 3 whiles, sendo que o primeiro termina quando o algoritmo termina de analisar essa metade do array, e algum desses outros while termina de andar no outro. Como as apenações em hash table são constantes, aqui temos m iterações fazendo operações  $O(1)$ , então essa etapa tem complexidade  $O(m)$

semelhante a etapa de merge  
do mergesort

- Por último o algoritmo copia todos os elementos dos vetores auxiliares para o vetor de entrada, com custo  $\Theta(n)$ , já que faz operações contínuas m vezes.

No geral, esse algoritmo é bem semelhante ao Merge Sort, e sua recorrência, baseado nas análises dadas, é

$$T(m) = \begin{cases} \Theta(1) & \text{se } m = 1 \\ 2T\left(\frac{m}{2}\right) + \Theta(m) & \text{se } m \geq 1 \end{cases}$$

$(f(m) = m^{\log_2 2})$

Como  $m^{\log_2 2} = m$ , essa recorrência cumpre o segundo passo do teorema acima e temos  $T(m) = \Theta(m \log m)$

✓ 2.0

**(2)\*a)** Como a estrutura tem memória limitada, podemos derivar outros algoritmos que tenham complexidade de espaço  $\Theta(n)$ , como o merge sort. Também podemos derivar o quicksort, já que sua complexidade em pior caso pode ser  $\Theta(n^2)$ .

Logo, o algoritmo mais adequado é o Heap Sort. Ele é realizado in-place em arrays, mas necessita de ponteiros para construir a heap.

Mais especificamente, ele que pode ser feito:

- Construir inicialmente uma max-heap no vetor de temperaturas (max-heapify na primeira metade)
- Executar o heap sort nos m primeiros elementos e inverte-los dessa forma.

Um dos vantagens do heap sort, além daquele pior caso  $\Theta(n \log n)$  e as operações implícias, é que ele permite ordenar apenas os m primeiros elementos, o que é vantajoso caso a estrutura tenha m de m temperaturas registradas.

PODERIA  
ELABORAR  
MAIS

✓ 0.4

(b) Como o número tem memória influente, não é possível prever em um algoritmo que faga uso de isto. Como o número de dígitos em um resíduo parcial é fixo (12) e a quantidade de dígitos também (do), é possível usar radix para fazer a ordenação. Como sabemos, a complexidade desse algoritmo é  $\underline{\underline{O(w(n+k))}}$ , sendo  $w$  o nº de dígitos e  $k$  o domínio do alfabeto deles.

Como 10 milhões é maior que 12 e 10 em duas ordens de magnitude, o algoritmo terá no pior caso complexidade  $O(n)$ , o que representa um vantagem para implementar, que podem aumentar a base de cálculo de radix para ter um aumento não-linear nos tempos de ordenação.

✓ 0.5

Passo: algoritmo principal, nomei o algoritmo mediano das medianas, porém modificado.

Como cada elemento da lista é um triple, é preciso que as comparações na mediana das medianas sejam feitas em algum das índices do triple (ou seja, no lugar de fazer

③  $A[i] < A[j]$ , fazer  $A[i][index] < A[j][index]$ ). O index será passado para o algoritmo,

que irá passar para todas as suas instruções que usam comparações.

$MOM(v, start, end, Search, index)$  ← representação da versão modificada

Além disso, nomei o algoritmo de particionamento visto em aula, mas com

uma modificação: ele irá procurar por um elemento, que será usado para partitionar.

PARTICIONA PORELEMENTO(v, start, end, elemento, index)

- para  $i = start$  até  $end$ :

- se  $v[i] == elemento$ :

- swap(v, i, end)

- break;

- PARTICIONA(v, start, end, index) -  $O(n)$

O algoritmo particionou é visto em aula, que partiu pelo último elemento.

Ele foi modificado assim como o MOM para fazer comparações pelo triple.

Por último, nomei hash tables, e o conjunto de chaves será o conjunto de

índices de medianas (ou seja, uma chave representa um índice)

0.8 ✓

ENCONTRA EMERGENTES ( $A, n$ )  
 define hashtable  $H$   
 define emergentes [ $n/10$ ]  
 define topengajamento = MOM( $A, 0, n, \frac{9n}{10}, 3$ )  
 define topseguidores = MOM( $A, 0, n, \frac{9n}{10}, 2$ )  
 PARTICIONA PORELEMENTO( $A, 0, n, topseguidores, 2$ )  
 - para  $i = \frac{9n}{10}$  até  $n$ :  
      $H[A[i][0]] = true$ . } item pelas TOP 10% seguidores  
     { salvo eles na hash

PARTICIONA POR ELEMENTO( $A, 0, n, topengajamento, 3$ )  
 define aux = 0

para  $i = \frac{9n}{10}$  até  $n$ :  
     se  $H[A[i][1]] != true$ :  
         emergentes[aux] =  $A[i][1]$   
         aux++

return emergentes, aux

Como tanto partitionar quanto usar MOM não operações lineares, temos que a complexidade dessas partes será  $\underline{O(n)}$ . Iterar pelas top 10% (tanto seguidores quanto engajamento) não operações em  $O(n/10)$ , já que dentro do loop apenas operações em tempo constante são realizadas.

Logo, a complexidade do algoritmo será  $T(n) = O(n) + O(n) + O(n) + O(n) + O(n/10) + O(n/10)$   
 $= \underline{O(n)}$

✓ 2.0

## ENCONTRA MAIOR SUBLISTA (A, n)

define melhor par = (-1, -1)

define melhor soma = -∞

- para  $i=0$  até  $n-1$

$O(n)$

- para  $j=i+1$  até  $n$

$O(n)$

soma atual = 0

- para  $k=i$  até  $j$

$O(n)$

soma atual +=  $A[k]$

se soma atual > melhor soma:

melhor soma = soma atual

melhor par = (i, j)

retorna melhor par, melhor soma

três  $O(n)$  dentro de um

dentro de outro

$$= m \cdot m \cdot m = m^3$$

$O(n^3)$

✓ 0.5

O que o algoritmo faz é bem simples: para todos pares  $(i, j)$  com  $i < j$ , tenta-se ver se  $i$  e  $j$  é melhor que a última melhor encontrada. É simples, mas a complexidade desse algoritmo é  $O(n^3)$  porque  $m^3$  não realizadas um dentro de outro,

resultando em complexidade  $O(n^3)$

b) É possível otimizar salvando a soma anterior e apenas multiplicando-a.

## ENCONTRA MAIOR SUBLISTA (A, n)

define melhor par = (-1, -1)

define melhor soma = -∞

define soma atual = 0

- para  $i=0$  até  $n-1$

soma atual =  $A[i]$

- para  $j=i+1$  até  $n$

soma atual +=  $A[j]$

se soma atual > melhor soma:

melhor soma = soma atual

melhor par = (i, j)

retorna melhor soma, melhor par

Caso as operações no segundo loop  
não em tempo constante, entao  
vamos para  $O(n^2)$

✓ 1.5

4) Lim, é perineal.

ENCONTRA MAIOR SUBLISTA (A, n)

define melhor par = (-1, -1)  
 define melhor soma = -∞  
 define soma atual = A[0]  
 define j = 0  
 para j=1 até n:  
 soma atual += A[j]  
 se soma atual - A[i] > soma atual:  
     j++  
     soma atual = soma atual - A[i]  
 se soma atual > melhor soma:  
     melhor soma = soma atual  
     melhor par = (i, j)  
 para i=i até melhor par, melhor soma:  
     soma atual -= A[i]  
 se soma atual > melhor soma:  
     melhor soma = soma atual  
     melhor par = (i, j)

No lugar de círculos sólidos ou combinações de i e j prevêem-se, o algoritmo aranjo j até o final da lista e ~~até cada passo~~ decide se deve varanjar i ou manter-lo onde está. Ao final, ele varanjo i até o final da array.

Indem farmo, se encontro cartas combinatórias de i e j que já não servem mais nenhuma, para não se arranjar é de novo é necessária.

X 10