



MAC0338 - ANÁLISE DE ALGORITMOS


LISTA 4


CHECKLIST


entregue  fiz  incompleto  não entendi 


questão 1 


questão 10 


questão 2 


questão 11 


questão 3 


questão 12 


questão 4 


questão 13 


questão 5 


questão 14 


questão 6 


questão 15 


questão 7 

questão 16 

questão 8 

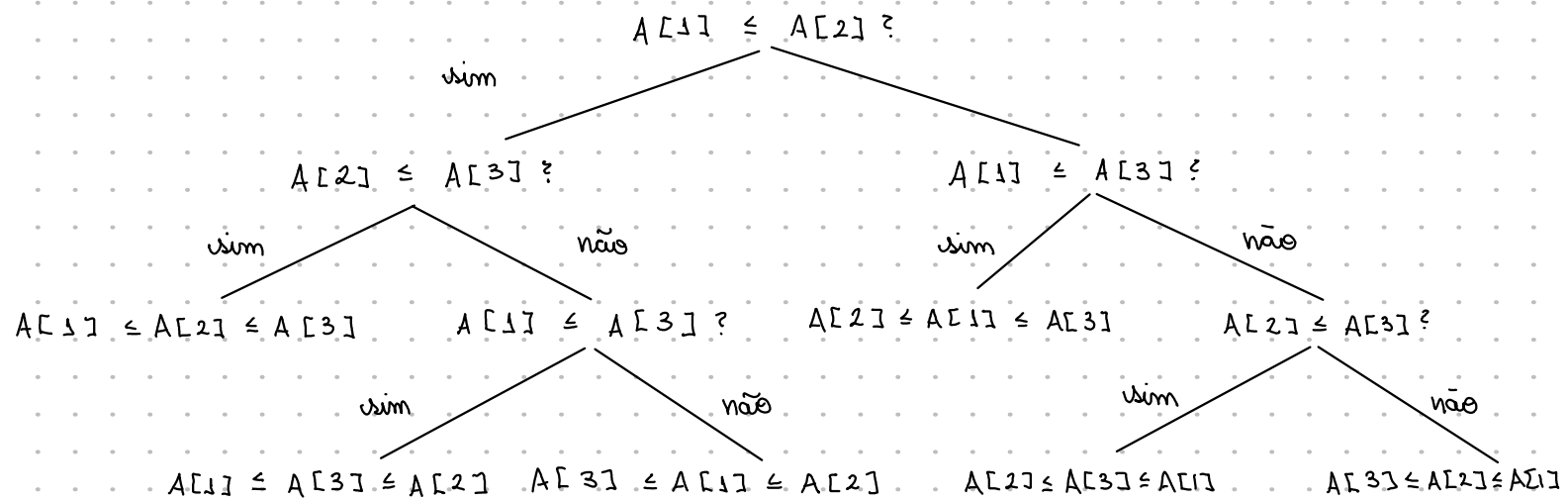
questão 17 

questão 9 

questão 18 

estudar depois da P1

1. Desenhe a árvore de decisão para o SELECTIONSORT aplicado a $A[1..3]$ com todos os elementos distintos.



- a ordenação por seleção melhora o bubble sort
- realiza apenas uma troca a cada passagem pela lista
- procura pelo valor mais alto enquanto faz uma passagem
- depois de completá-la, coloca-o na posição certa
- complexidade $O(n^2)$

2. (CLRS 8.1-1) Qual a menor profundidade (= menor nível) que uma folha pode ter em uma árvore de decisão que descreve um algoritmo de ordenação baseado em comparações?

Considere uma árvore de decisão para $A[1 \dots n]$

A menor profundidade acontece quando a lista a ser ordenada já está ordenada.

Neste caso, uma folha tem profundidade $n-1$, que é exatamente o número de comparações.

- Para se ter certeza que um vetor está ordenado é preciso que todos os inteiros estejam "conectados" por alguma comparação, assim como um grafo conexo que tem $n-1$ arestas a árvore tem que ter altura $n-1$ em seu menor nível.

4. (CLRS 8.1-3) Mostre que não há algoritmo de ordenação baseado em comparações cujo consumo de tempo é linear para pelo menos metade das $n!$ permutações de 1 a n . O que acontece se trocarmos “metade” por uma fração de $1/n$? O que acontece se trocarmos “metade” por uma fração de $1/2^n$?

5. (CLRS 8.2-1) Simule a execução do COUNTINGSORT usando como entrada o vetor

$$A[1..11] = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle.$$

$$A[1..11] =$$

1	2	3	4	5	6	7	8	9	10	11
6	0	2	0	1	3	4	6	1	3	2

• armazena a quantidade de cada inteiro em C

$$C[0..6] =$$

0	1	2	3	4	5	6
2	2	2	2	1	0	2

• calcula a soma acumulada

$$C[0..6] =$$

0	1	2	3	4	5	6
2	4	6	8	9	9	11

• preenche o vetor B.

$$B[1..11] =$$

1	2	3	4	5	6	7	8	9	10	11
					2					

$$C[0..6] =$$

0	1	2	3	4	5	6
	4	5	8	9	9	11

$$B[1..11] =$$

1	2	3	4	5	6	7	8	9	10	11
					2		3			

$$C[0..6] =$$

0	1	2	3	4	5	6
2	4	5	7	9	9	11

$$B[1..11] =$$

1	2	3	4	5	6	7	8	9	10	11
			1		2		3			

$$C[0..6] =$$

0	1	2	3	4	5	6
2	3	5	8	9	9	11

$$B[1..11] =$$

1	2	3	4	5	6	7	8	9	10	11
			1		2		3			6

$$C[0..6] =$$

0	1	2	3	4	5	6
2	3	5	8	9	9	10

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
			1		2		3	4		6

$$C[0 \dots 6] =$$

0	1	2	3	4	5	6
2	3	5	7	8	9	10

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
			1		2	3	3	4		6

$$C[0 \dots 6] =$$

0	1	2	3	4	5	6
2	3	5	6	8	9	10

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
	0	1	1		2	3	3	4		6

$$C[0 \dots 6] =$$

0	1	2	3	4	5	6
1	2	5	6	8	9	10

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
	0	1	1	2	2	3	3	4		6

$$C[0 \dots 6] =$$

0	1	2	3	4	5	6
1	2	4	6	8	9	10

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	2	2	3	3	4	6	6

$$C[0 \dots 6] =$$

0	1	2	3	4	5	6
0	3	4	6	8	9	9

returna

$$B[1 \dots 11] =$$

1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	2	2	3	3	4	6	6

6. (CLRS 8.2-2) Mostre que o COUNTINGSORT é estável.

COUNTINGSORT(A, n)

```
1  para  $i \leftarrow 1$  até  $k$  faça
2     $C[i] \leftarrow 0$ 
3  para  $j \leftarrow 1$  até  $n$  faça
4     $C[A[j]] \leftarrow C[A[j]] + 1$ 
5  para  $i \leftarrow 2$  até  $k$  faça
6     $C[i] \leftarrow C[i] + C[i-1]$ 
7  para  $j \leftarrow n$  decrescendo até 1 faça
8     $B[C[A[j]]] \leftarrow A[j]$ 
9     $C[A[j]] \leftarrow C[A[j]] - 1$ 
10 devolva  $B$ 
```

A linha 7 do counting sort garante que o algoritmo seja estável, pois ordenamos o vetor A de acordo com o vetor C , que nos dá primeiro a última posição que um inteiro ocupa, ou seja, inserimos os inteiros nas posições em B em ordem decrescente, e assim, ao percorrer o vetor A na linha 7 em ordem decrescente, temos que a ordem original dos inteiros iguais será garantida.

7. (CLRS 8.2-3) Suponha que o para da linha 7 do COUNTINGSORT é substituído por

para $j \leftarrow 1$ até n faça

Mostre que o COUNTINGSORT ainda funciona. O algoritmo resultante continua estável?

O counting sort ainda funciona, pois os inteiros serão armazenados em B de acordo com a soma acumulada em C , que indica as posições, porém, como visto no sec 6) para ser estável é preciso que assim como as posições do vetor C que são dadas em ordem decrescente o vetor A também precisa ser percorrido em ordem decrescente para preservar a ordem original dos inteiros iguais.

9. (CLRS 8.3-4) Mostre como ordenar n inteiros no intervalo de 0 até $n^2 - 1$ em tempo $O(n)$.

com Radix Sort

10. (CLRS 8.4-1) Simule a execução do BUCKETSORT com o vetor

$A[1..10] = \langle 0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42 \rangle$.

B[0] :

B[1] : 0.13 0.16

B[2] : 0.20

B[3] : 0.39

B[4] : 0.42

B[5] : 0.53

B[6] : 0.64

B[7] : 0.79 0.71

B[8] : 0.89

B[9]

B[0] :

B[1] : 0.13 0.16

B[2] : 0.20

B[3] : 0.39

B[4] : 0.42

B[5] : 0.53

B[6] : 0.64

B[7] : 0.71 0.79

B[8] : 0.89

B[9]

C: 0.13 0.16 0.20 0.39 0.42 0.53 0.64 0.71 0.79 0.89

11. (CLRS 8.4-2) Qual é o consumo de tempo de pior caso para o BUCKETSORT? Que simples ajuste do algoritmo melhora o seu pior caso para $O(n \lg n)$ e mantém o seu consumo esperado de tempo linear.

O consumo do BucketSort quando os elementos estão bem ordenados é $O(n)$, mas quando cai no pior caso e os elementos caem em um mesmo balde a complexidade fica $O(n^2)$ quando é utilizado o insertion sort, porém utilizando o mergesort o pior caso vai para $O(n \lg n)$.

MAC0338 - ANÁLISE DE ALGORITMOS

LISTA 4

8. (CLRS 8.2-4) Descreva um algoritmo que, dados n inteiros no intervalo de 1 a k , processe sua entrada e então responda em $O(1)$ qualquer consulta sobre quantos dos n inteiros dados caem em um intervalo $[a..b]$. O pré-processamento efetuado pelo seu algoritmo deve consumir tempo $O(n + k)$.

seja A um vetor $A[1..n]$ onde cada elemento é um inteiro entre 1 e K
e assumindo que $1 \leq a \leq b \leq K$

CONSULTA(A, n, K, a, b)

```
1  para  $i \leftarrow 1$  até  $K$  faça           // pré-processa a entrada
2       $C[i] \leftarrow 0$ 
3  para  $j \leftarrow 1$  até  $n$  faça
4       $C[A[j]] \leftarrow C[A[j]] + 1$ 
5  para  $i \leftarrow 2$  até  $K$  faça
6       $C[i] \leftarrow C[i] + C[i-1]$ 
7  se  $a = 1$  faça                       // consulta o n° de inteiros em  $[a..b]$ 
8      retorne  $C[b]$ 
9  senão faça
10     retorne  $C[b] - C[a-1]$ 
```

MAC0338 - ANÁLISE DE ALGORITMOS

LISTA 4

18. A remoção da superfície escondida é um problema em computação gráfica que raramente precisa de introdução: quando o João tá na frente da Maria, você pode ver o João, mas não a Maria; quando a Maria tá na frente do João, ... Você entendeu a idéia.

A beleza desse problema é que você pode resolvê-lo mais rapidamente do que a intuição em geral sugere. Aqui está uma versão simplificada do problema onde já podemos apresentar um algoritmo mais eficiente do que a primeira solução em que se pode pensar. Imagine que são dadas n retas não verticais no plano, denotadas por L_1, \dots, L_n . Digamos que L_i é dada pela equação $y = a_i x + b_i$, para $i = 1, \dots, n$. Suponha que não há três retas entre as retas dadas que se intersectam mutuamente num mesmo ponto. Dizemos que a reta L_i é a *mais alta* numa dada coordenada $x = x_0$ se sua coordenada y em x_0 é maior que a coordenada y em x_0 de todas as outras retas dadas. Ou seja, se $a_i x_0 + b_i > a_j x_0 + b_j$ para todo $j \neq i$. Dizemos que L_i é *visível* se existe uma coordenada x na qual ela é a mais alta. Intuitivamente, isso corresponde a uma parte de L_i ser visível se você olhar para baixo a partir de $y = \infty$.

Escreva um algoritmo $O(n \lg n)$ que recebe uma sequência de n retas, como descrito acima, e devolve a subsequência delas que é visível.

Seja A o vetor de retas

SUBSEQUENCIA (A)

1 $n \leftarrow$ tamanho da lista A

2 se $n > 3$

3 $lista1 \leftarrow$ SUBSEQUENCIA ($A[1 \dots n/2]$)

4 $lista2 \leftarrow$ SUBSEQUENCIA ($A[(n/2)+1 \dots n]$)

5 retorna VISIVEIS ($lista1, lista2$)

6 senão se $n = 3$

7 $interseccao1 \leftarrow |(b_2 - b_1) / (a_1 - a_2)|$

8 $interseccao2 \leftarrow |(b_3 - b_1) / (a_1 - a_3)|$

9 se $interseccao1 > interseccao2$ então

10 remove $A[2]$

11 senão se $n = 2$

12 se a_1 é igual a_2

13 se $b_1 > b_2$

14 remove $A[2]$

15 senão

16 remove $A[1]$

17 retorna A

VISIVEIS (lista 1, lista 2)

```
1  B ← INTERCALA (lista 1, lista 2) // intercala em tempo  $O(n)$ 
2  V[1] ← B[1] // adiciona as duas primeiras retas no vetor de retas visíveis
3  V[2] ← B[2]
4  para cada reta l em B
5      interseccao1 = interseccao da última reta com a penúltima
6      interseccao2 = interseccao da reta l com a penúltima
7      enquanto o tamanho de V[] for  $\geq 2$  e interseccao1 < interseccao2
8          // a reta l intercepta a penúltima anterior antes da anterior
9          remove o último item de V[]
10     adiciona l ao vetor V[]
11 retorna V[] // vetor de retas visíveis
```

- Na função VISIVEIS() o algoritmo ordena a lista em tempo $O(n \lg n)$
- A linha 4 até a linha 10 do algoritmo consome tempo $O(n)$, pois em cada passo $O(n)$ uma linha é adicionada ou removida da lista de retas visíveis, assim, a mesma linha não é adicionada e removida da lista mais de uma vez.
- Em VISIVEIS() o algoritmo consome tempo $O(n)$.
- Em SUBSEQUENCIA() o algoritmo divide o problema em dois subproblemas de tamanho $\frac{n}{2}$.
- Tem-se que:

$$T(n) = 2T(n/2) + O(n) = O(n \lg n)$$

Portanto, o algoritmo consome tempo $O(n \lg n)$.

