

Ordenação em tempo linear

AULA 8

CLRS cap 8

Counting Sort

CLRS sec 8.2

Counting Sort

Recebe inteiros n e k , e um vetor $A[1..n]$
onde cada elemento é um inteiro entre 1 e k .

Counting Sort

Recebe inteiros n e k , e um vetor $A[1..n]$
onde cada elemento é um inteiro entre 1 e k .

Devolve um vetor $B[1..n]$ com
os elementos de $A[1..n]$ em ordem crescente.

Counting Sort

Recebe inteiros n e k , e um vetor $A[1..n]$ onde cada elemento é um inteiro entre 1 e k .

Devolve um vetor $B[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

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 $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright C[i]$: número de ocorrências de i em A

Counting Sort

Recebe inteiros n e k , e um vetor $A[1..n]$ onde cada elemento é um inteiro entre 1 e k .

Devolve um vetor $B[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

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 $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i]$: número de ocorrências de i em A
- 5 para $i \leftarrow 2$ até k faça
- 6 $C[i] \leftarrow C[i] + C[i - 1]$ ▷ $C[i]$: número de elementos $\leq i$ em A

Counting Sort

Recebe inteiros n e k , e um vetor $A[1..n]$ onde cada elemento é um inteiro entre 1 e k .

Devolve um vetor $B[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

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       $C[A[j]] \leftarrow C[A[j]] + 1$     ▷  $C[i]$ : número de ocorrências de  $i$  em  $A$ 
5  para  $i \leftarrow 2$  até  $k$  faça
6       $C[i] \leftarrow C[i] + C[i - 1]$     ▷  $C[i]$ : número de elementos  $\leq i$  em  $A$ 
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  $B$ 
```

Counting Sort

```
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  $B$ 
```

Por que escrevemos as linhas 7-9 desta maneira?

Counting Sort

```
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  $B$ 
```

Por que escrevemos as linhas 7-9 desta maneira?

Porque isso garante que o COUNTINGSORT seja estável.

Estabilidade

Um algoritmo de ordenação é **estável** se sempre que, inicialmente, $A[i] = A[j]$ para $i < j$, a cópia $A[i]$ termina em uma posição menor do vetor que a cópia $A[j]$.

Estabilidade

Um algoritmo de ordenação é **estável** se sempre que, inicialmente, $A[i] = A[j]$ para $i < j$, a cópia $A[i]$ termina em uma posição menor do vetor que a cópia $A[j]$.

Isso só é relevante quando temos **informação satélite**.

Consumo de tempo

Quanto tempo consome o COUNTINGSORT em função do n e do k ?

```
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  $B$ 
```

Consumo de tempo

linha	consumo na linha
1	$\Theta(k)$
2	$O(k)$
3	$\Theta(n)$
4	$O(n)$
5	$\Theta(k)$
6	$O(k)$
7	$\Theta(n)$
8	$O(n)$
9	$O(n)$
10	$\Theta(1)$
total	????

Consumo de tempo

linha	consumo na linha
1	$\Theta(k)$
2	$O(k)$
3	$\Theta(n)$
4	$O(n)$
5	$\Theta(k)$
6	$O(k)$
7	$\Theta(n)$
8	$O(n)$
9	$O(n)$
10	$\Theta(1)$
total	$\Theta(k + n)$

Counting Sort

```
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  $B$ 
```

Consumo de tempo: $\Theta(k + n)$

Se $k = O(n)$, o consumo de tempo é $\Theta(n)$.

Radixsort

CLRS sec 8.3

Radix Sort

Algoritmo usado para ordenar

- ▶ inteiros não-negativos com d dígitos
- ▶ cartões perfurados (Hollerith!)
- ▶ registros cuja chave tem vários campos

Radix Sort

Algoritmo usado para ordenar

- ▶ inteiros não-negativos com d dígitos
- ▶ cartões perfurados (Hollerith!)
- ▶ registros cuja chave tem vários campos

campo 1: menos significativo

campo d : mais significativo

Radix Sort

Algoritmo usado para ordenar

- ▶ inteiros não-negativos com d dígitos
- ▶ cartões perfurados (Hollerith!)
- ▶ registros cuja chave tem vários campos

campo 1: menos significativo

campo d : mais significativo

```
RADIXSORT( $A, n, d$ )  
1  para  $i \leftarrow 1$  até  $d$  faça  
2      ORDENE ( $A, n, i$ )
```

Radix Sort

Algoritmo usado para ordenar

- ▶ inteiros não-negativos com d dígitos
- ▶ cartões perfurados (Hollerith!)
- ▶ registros cuja chave tem vários campos

campo 1: menos significativo

campo d : mais significativo

```
RADIXSORT( $A, n, d$ )  
1  para  $i \leftarrow 1$  até  $d$  faça  
2      ORDENE ( $A, n, i$ )
```

ORDENE (A, n, i): ordena $A[1..n]$ pelo i -ésimo campo dos registros em A por meio de um algoritmo **estável**.

Estabilidade

Um algoritmo de ordenação é **estável** se sempre que, inicialmente, $A[i] = A[j]$ para $i < j$, a cópia $A[i]$ termina em uma posição menor do vetor que a cópia $A[j]$.

Isso só é relevante quando temos **informação satélite**.

Quais dos algoritmos que vimos são estáveis?

- ▶ inserção direta? seleção direta? bubblesort?
- ▶ mergesort?
- ▶ quicksort?
- ▶ heapsort?
- ▶ countingsort?

Consumo de tempo do Radixsort

Depende do algoritmo **ORDENE**.

Consumo de tempo do Radixsort

Depende do algoritmo **ORDENE**.

Se cada campo é um inteiro de 1 a k ,
então podemos usar o **COUNTINGSORT**.

Consumo de tempo do Radixsort

Depende do algoritmo **ORDENE**.

Se cada campo é um inteiro de 1 a k ,
então podemos usar o **COUNTINGSORT**.

Neste caso, o consumo de tempo é $\Theta(d(k + n))$.

Consumo de tempo do Radixsort

Depende do algoritmo **ORDENE**.

Se cada campo é um inteiro de 1 a k ,
então podemos usar o **COUNTINGSORT**.

Neste caso, o consumo de tempo é $\Theta(d(k + n))$.

Se d é limitado por uma constante (ou seja, se $d = O(1)$)
e $k = O(n)$, então o consumo de tempo é $\Theta(n)$.

Bucketsort

CLRS sec 8.4

Bucket Sort

Recebe um inteiro n e um vetor $A[1..n]$ onde cada elemento é um número no intervalo $[0,1)$.

Bucket Sort

Recebe um inteiro n e um vetor $A[1..n]$ onde cada elemento é um número no intervalo $[0,1)$.

A

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Bucket Sort

Recebe um inteiro n e um vetor $A[1..n]$ onde cada elemento é um número no intervalo $[0,1)$.

A	.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Devolve um vetor $C[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

Bucket Sort

Recebe um inteiro n e um vetor $A[1..n]$ onde cada elemento é um número no intervalo $[0,1)$.

A

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Devolve um vetor $C[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

C

.03	.12	.38	.42	.47	.62	.77	.82	.91	.93
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Exemplo

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Exemplo

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$B[0]$:	.03
$B[1]$:	.12
$B[2]$:	
$B[3]$:	.38
$B[4]$:	.47 .42
$B[5]$:	
$B[6]$:	.62
$B[7]$:	.77
$B[8]$:	.82
$B[9]$:	.93 .91

Exemplo

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$B[0]$:	.03
$B[1]$:	.12
$B[2]$:	
$B[3]$:	.38
$B[4]$:	.42 .47
$B[5]$:	
$B[6]$:	.62
$B[7]$:	.77
$B[8]$:	.82
$B[9]$:	.91 .93

Exemplo

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$B[0]$:	.03
$B[1]$:	.12
$B[2]$:	
$B[3]$:	.38
$B[4]$:	.42 .47
$B[5]$:	
$B[6]$:	.62
$B[7]$:	.77
$B[8]$:	.82
$B[9]$:	.91 .93

.03	.12	.38	.42	.47	.62	.77	.82	.91	.93
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Bucket Sort

Recebe um inteiro n e um vetor $A[1..n]$ onde cada elemento é um número no intervalo $[0, 1)$.

Devolve um vetor $C[1..n]$ com os elementos de $A[1..n]$ em ordem crescente.

BUCKETSORT(A, n)

- 1 **para** $i \leftarrow 0$ **até** $n - 1$ **faça**
- 2 $B[i] \leftarrow \text{NIL}$
- 3 **para** $i \leftarrow 1$ **até** n **faça**
- 4 **INSIRA**($B[\lfloor nA[i] \rfloor], A[i]$)
- 5 **para** $i \leftarrow 0$ **até** $n - 1$ **faça**
- 6 **ORDENELISTA**($B[i]$)
- 7 $C \leftarrow \text{CONCATENE}(B, n)$
- 8 **devolva** C

Bucket Sort

BUCKETSORT(A, n)

- 1 **para** $i \leftarrow 0$ até $n - 1$ **faça**
- 2 $B[i] \leftarrow \text{NIL}$
- 3 **para** $i \leftarrow 1$ até n **faça**
- 4 INSIRA($B[\lfloor nA[i] \rfloor], A[i]$)
- 5 **para** $i \leftarrow 0$ até $n - 1$ **faça**
- 6 ORDENELISTA($B[i]$)
- 7 $C \leftarrow \text{CONCATENE}(B, n)$
- 8 **devolva** C

INSIRA(p, x): insere x na lista apontada por p

ORDENELISTA(p): ordena a lista apontada por p

CONCATENE(B, n): devolve a lista obtida da concatenação das listas apontadas por $B[0], \dots, B[n - 1]$.

Consumo de tempo

Suponha que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0, 1)$.

Suponha que o **ORDENELISTA** seja o INSERTIONSORT.

Consumo de tempo

Suponha que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0, 1)$.

Suponha que o **ORDENELISTA** seja o INSERTIONSORT.

Seja X_i o número de elementos na lista $B[i]$.

Consumo de tempo

Suponha que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Suponha que o **ORDENELISTA** seja o INSERTIONSORT.

Seja X_i o número de elementos na lista $B[i]$.

Seja

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Consumo de tempo

Suponha que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Suponha que o **ORDENELISTA** seja o INSERTIONSORT.

Seja X_i o número de elementos na lista $B[i]$.

Seja

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Observe que $X_i = \sum_j X_{ij}$.

Consumo de tempo

Suponha que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Suponha que o **ORDENELISTA** seja o INSERTIONSORT.

Seja X_i o número de elementos na lista $B[i]$.

Seja

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Observe que $X_i = \sum_j X_{ij}$.

Y_i : número de comparações para ordenar a lista $B[i]$.

Consumo de tempo

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Consumo de tempo

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$.

Logo $E[Y_i] \leq E[X_i^2] = E[(\sum_j X_{ij})^2]$.

Consumo de tempo

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$.

Logo $E[Y_i] \leq E[X_i^2] = E[(\sum_j X_{ij})^2]$.

$$\begin{aligned} E[(\sum_j X_{ij})^2] &= E[\sum_j \sum_k X_{ij} X_{ik}] \\ &= E[\sum_j X_{ij}^2 + \sum_j \sum_{k \neq j} X_{ij} X_{ik}] \end{aligned}$$

Consumo de tempo

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$.

Logo $E[Y_i] \leq E[X_i^2] = E[(\sum_j X_{ij})^2]$.

$$\begin{aligned} E[(\sum_j X_{ij})^2] &= E[\sum_j \sum_k X_{ij} X_{ik}] \\ &= E[\sum_j X_{ij}^2] + E[\sum_j \sum_{k \neq j} X_{ij} X_{ik}] \end{aligned}$$

Consumo de tempo

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$.

Logo $E[Y_i] \leq E[X_i^2] = E[(\sum_j X_{ij})^2]$.

$$\begin{aligned} E[(\sum_j X_{ij})^2] &= E[\sum_j \sum_k X_{ij} X_{ik}] \\ &= \sum_j E[X_{ij}^2] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}] \end{aligned}$$

Consumo de tempo

Primeiro vamos calcular $E[X_{ij}^2]$.

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Consumo de tempo

Primeiro vamos calcular $E[X_{ij}^2]$.

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Lembre-se que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Consumo de tempo

Primeiro vamos calcular $E[X_{ij}^2]$.

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Lembre-se que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Observe que X_{ij}^2 é uma variável aleatória binária.

Consumo de tempo

Primeiro vamos calcular $E[X_{ij}^2]$.

$$X_{ij} = \begin{cases} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{cases}$$

Lembre-se que os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0,1)$.

Observe que X_{ij}^2 é uma variável aleatória binária. Então

$$E[X_{ij}^2] = \Pr[X_{ij}^2 = 1] = \Pr[X_{ij} = 1] = \frac{1}{n}.$$

Consumo de tempo

Voltando...

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \left\{ \begin{array}{ll} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{array} \right\}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$. Ademais,

$$E[Y_i] \leq \sum_j E[X_{ij}^2] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}]$$

Consumo de tempo

Voltando...

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \left\{ \begin{array}{ll} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{array} \right\}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$. Ademais,

$$\begin{aligned} E[Y_i] &\leq \sum_j E[X_{ij}^2] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}] \\ &= \sum_j \frac{1}{n} + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}] \end{aligned}$$

Consumo de tempo

Voltando...

X_i : número de elementos na lista $B[i]$

$$X_{ij} = \left\{ \begin{array}{ll} 1 & \text{se o } j\text{-ésimo elemento foi para a lista } B[i] \\ 0 & \text{se o } j\text{-ésimo elemento não foi para a lista } B[i]. \end{array} \right\}$$

Y_i : número de comparações para ordenar a lista $B[i]$.

Observe que $Y_i \leq X_i^2$. Ademais,

$$\begin{aligned} E[Y_i] &\leq \sum_j E[X_{ij}^2] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}] \\ &= \sum_j \frac{1}{n} + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}] \\ &= 1 + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}]. \end{aligned}$$

Consumo de tempo

Para calcular $E[X_{ij}X_{ik}]$ para $j \neq k$, primeiro note que X_{ij} e X_{ik} são variáveis aleatórias independentes.

Portanto, $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]$.

Ademais, $E[X_{ij}] = \Pr[X_{ij} = 1] = \frac{1}{n}$.

Consumo de tempo

Para calcular $E[X_{ij}X_{ik}]$ para $j \neq k$, primeiro note que X_{ij} e X_{ik} são variáveis aleatórias independentes.

Portanto, $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]$.

Ademais, $E[X_{ij}] = \Pr[X_{ij} = 1] = \frac{1}{n}$.

Logo,

$$\begin{aligned} E[Y_i] &\leq 1 + \sum_j \sum_{k \neq j} \frac{1}{n^2} \\ &= 1 + n(n-1) \frac{1}{n^2} \\ &= 1 + (n-1) \frac{1}{n} \\ &= 2 - \frac{1}{n}. \end{aligned}$$

Consumo de tempo

Agora, seja $Y = \sum_i Y_i$.

Note que Y é o número de comparações realizadas pelo **BUCKETSORT** no total.

Assim $E[Y]$ é o número esperado de comparações realizadas pelo algoritmo, e tal número determina o consumo assintótico de tempo do **BUCKETSORT**.

Consumo de tempo

Agora, seja $Y = \sum_i Y_i$.

Note que Y é o número de comparações realizadas pelo **BUCKETSORT** no total.

Assim $E[Y]$ é o número esperado de comparações realizadas pelo algoritmo, e tal número determina o consumo assintótico de tempo do **BUCKETSORT**.

$$E[Y] = \sum_i E[Y_i] \leq 2n - 1 = O(n).$$

Consumo de tempo

Agora, seja $Y = \sum_i Y_i$.

Note que Y é o número de comparações realizadas pelo **BUCKETSORT** no total.

Assim $E[Y]$ é o número esperado de comparações realizadas pelo algoritmo, e tal número determina o consumo assintótico de tempo do **BUCKETSORT**.

$$E[Y] = \sum_i E[Y_i] \leq 2n - 1 = O(n).$$

O consumo de tempo esperado do **BUCKETSORT** quando os números em $A[1..n]$ são uniformemente distribuídos no intervalo $[0, 1)$ é $O(n)$.

Dicas de implementação

Mergesort

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Quicksort

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Quicksort

- ▶ Elementos repetidos: partição ternária

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Quicksort

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Quicksort

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda

Dicas de implementação

Mergesort

- ▶ Onde declarar o vetor auxiliar?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado
- ▶ Inserção direta para vetores pequenos

Quicksort

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda
- ▶ Inserção direta para vetores pequenos

Dicas de implementação: Mergesort

Rearranja $A[p..r]$ em ordem crescente.

```
MERGESORT ( $A, p, r$ )  
1  se  $p < r$   
2    então  $q \leftarrow \lfloor (p + r) / 2 \rfloor$   
3        MERGESORT ( $A, p, q$ )  
4        MERGESORT ( $A, q + 1, r$ )  
5        INTERCALA ( $A, p, q, r$ )
```

- ▶ Onde declarar o vetor auxiliar usado no **INTERCALA**?

Dicas de implementação: Mergesort

MERGESORT (A, n) ← aqui

1 MERGESORTREC ($A, 1, n$)

MERGESORTREC (A, p, r)

1 se $p < r$

2 então $q \leftarrow \lfloor (p + r) / 2 \rfloor$

3 MERGESORTREC (A, p, q)

4 MERGESORTREC ($A, q + 1, r$)

5 INTERCALA (A, p, q, r)

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?

Dicas de implementação: Mergesort

MERGESORT (A, n)

1 MERGESORTREC ($A, 1, n$)

MERGESORTREC (A, p, r)

1 **se** $p < r$

2 **então** $q \leftarrow \lfloor (p + r) / 2 \rfloor$

3 MERGESORTREC (A, p, q)

4 MERGESORTREC ($A, q + 1, r$)

5 INTERCALA (A, p, q, r)

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?
- ▶ Alternância entre dois vetores

Dicas de implementação: Mergesort

MERGESORT (A, n)

- 1 $B \leftarrow A$
- 2 MERGESORTREC ($B, A, 1, n$)

MERGESORTREC (A, B, p, r)

- 1 **se** $p < r$
- 2 **então** $q \leftarrow \lfloor (p + r) / 2 \rfloor$
- 3 MERGESORTREC (B, A, p, q)
- 4 MERGESORTREC ($B, A, q + 1, r$)
- 5 INTERCALA (A, B, p, q, r)

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?
- ▶ Alternância entre dois vetores

Dicas de implementação: Mergesort

INTERCALA (A, B, p, q, r)

1 $i \leftarrow p$ $j \leftarrow q + 1$ $k \leftarrow p$

2 enquanto $i \leq q$ e $j \leq r$ faça

3 se $A[i] \leq A[j]$

4 então $B[k] \leftarrow A[i]$ $k \leftarrow k + 1$ $i \leftarrow i + 1$

5 senão $B[k] \leftarrow A[j]$ $k \leftarrow k + 1$ $j \leftarrow j + 1$

6 enquanto $i \leq q$ faça

7 $B[k] \leftarrow A[i]$ $k \leftarrow k + 1$ $i \leftarrow i + 1$

8 enquanto $j \leq r$ faça

9 $B[k] \leftarrow A[j]$ $k \leftarrow k + 1$ $j \leftarrow j + 1$

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?
- ▶ Alternância entre dois vetores

Dicas de implementação: Mergesort

INTERCALA (A, p, q, r)

```
1 para  $i \leftarrow p$  até  $q$  faça  $B[i] \leftarrow A[i]$ 
2 para  $j \leftarrow q + 1$  até  $r$  faça  $B[r + q + 1 - j] \leftarrow A[j]$ 
3  $i \leftarrow p$   $j \leftarrow r$ 
4 para  $k \leftarrow p$  até  $r$  faça
5     se  $B[i] \leq B[j]$ 
6         então  $A[k] \leftarrow B[i]$ 
7              $i \leftarrow i + 1$ 
8     senão  $A[k] \leftarrow B[j]$ 
9          $j \leftarrow j - 1$ 
```

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado

Dicas de implementação: Mergesort

```
INTERCALA (A, p, q, r)
0  se  $A[q] > A[q+1]$  então ← aqui
1    para  $i \leftarrow p$  até  $q$  faça  $B[i] \leftarrow A[i]$ 
2    para  $j \leftarrow q+1$  até  $r$  faça  $B[r+q+1-j] \leftarrow A[j]$ 
3     $i \leftarrow p$     $j \leftarrow r$ 
4    para  $k \leftarrow p$  até  $r$  faça
5      se  $B[i] \leq B[j]$ 
6        então  $A[k] \leftarrow B[i]$ 
7           $i \leftarrow i+1$ 
8        senão  $A[k] \leftarrow B[j]$ 
9           $j \leftarrow j-1$ 
```

- ▶ Onde declarar o vetor auxiliar usado no INTERCALA?
- ▶ Alternância entre dois vetores
- ▶ **Teste se já ordenado**

Dicas de implementação: Quicksort

```
QUICKSORT ( $A, p, r$ )  
1  se  $p < r$   
2    então  $q \leftarrow$  PARTICIONE( $A, p, r$ )  
3         QUICKSORT( $A, p, q - 1$ )  
4         QUICKSORT( $A, q + 1, r$ )
```

Dicas de implementação: Quicksort

```
QUICKSORT ( $A, p, r$ )  
1  se  $p < r$   
2    então  $(q, t) \leftarrow \text{TRIPARTICIONE}(A, p, r)$   
3         QUICKSORT ( $A, p, q - 1$ )  
4         QUICKSORT ( $A, t + 1, r$ )
```

- ▶ Elementos repetidos: partição ternária

Dicas de implementação: Quicksort

PARTICIONE-ALEA(A, p, r)

1 $i \leftarrow \text{RANDOM}(p, r)$

2 $A[i] \leftrightarrow A[r]$

3 **devolva** PARTICIONE(A, p, r)

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três

Dicas de implementação: Quicksort

PARTICIONE-ALEA(A, p, r)

1 $i \leftarrow \text{RANDOM}(p, r)$

2 $A[i] \leftrightarrow A[r]$

3 **devolva** PARTICIONE(A, p, r)

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três

Em vez de escolher um pivô aleatoriamente, escolhe três elementos do vetor aleatoriamente, e usa como pivô a mediana dos três.

Dicas de implementação: Quicksort

PARTICIONE-ALEA(A, p, r)

1 $i \leftarrow \text{RANDOM}(p, r)$

2 $A[i] \leftrightarrow A[r]$

3 **devolva** PARTICIONE(A, p, r)

- ▶ Elementos repetidos: partição ternária
- ▶ **Mediana de três**

Em vez de escolher um pivô aleatoriamente, escolhe três elementos do vetor aleatoriamente, e usa como pivô a mediana dos três.

Aproveite para colocar o menor dos três no extremo esquerdo, e o maior dos três no extremo direito, e use-os de sentinela na partição!

Particione

Rearranja $A[p..r]$ de modo que $p \leq q \leq r$ e
 $A[p..q-1] \leq A[q] < A[q+1..r]$

Supõe que $A[p] \leq A[r] \leq A[r-1]$ e use $A[r]$ como pivô.

PARTICIONE (A, p, r)

- 1 $x \leftarrow A[r]$ $\triangleright x$ é o “pivô”
- 2 $i \leftarrow p + 1$ $j \leftarrow r - 2$
- 3 **enquanto** $i \leq j$ **faça**
- 4 **enquanto** $A[i] \leq x$ **faça** $i \leftarrow i + 1$
- 5 **enquanto** $A[j] \geq x$ **faça** $j \leftarrow j - 1$
- 6 **se** $i < j$ **então** $A[i] \leftrightarrow A[j]$
- 7 $A[i] \leftrightarrow A[r]$
- 8 **devolva** i

Dicas de implementação: Quicksort

QUICKSORT (A, p, r)

1 se $p < r$

2 então $q \leftarrow$ PARTICIONE(A, p, r)

3 QUICKSORT($A, p, q - 1$)

4 QUICKSORT($A, q + 1, r$)

← aqui

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda

Dicas de implementação: Quicksort

QUICKSORT (A, p, r)

1 enquanto $p < r$ faça ← **virou enquanto aqui!**

2 $q \leftarrow$ PARTICIONE(A, p, r)

3 QUICKSORT($A, p, q - 1$)

4 $p \leftarrow q + 1$ ← **ajusta os parâmetros!**

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ **Recursão de cauda**

Substitua essa chamada recursiva por um enquanto.

Dicas de implementação: Quicksort

QUICKSORT (A, p, r)

1 enquanto $p < r$ faça ← virou enquanto aqui!

2 $q \leftarrow$ PARTICIONE(A, p, r)

3 QUICKSORT($A, p, q - 1$)

4 $p \leftarrow q + 1$ ← ajusta os parâmetros!

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda

Correto: substitua a chamada da maior metade!

Dicas de implementação: Quicksort

QUICKSORT (A, p, r)

```
1 enquanto  $p < r$  ← virou enquanto aqui!  
2      $q \leftarrow$  PARTICIONE( $A, p, r$ )  
3     se  $q - p < r - q$   
4         então QUICKSORT( $A, p, q - 1$ )  
5              $p \leftarrow q + 1$   
6         senão QUICKSORT( $A, q + 1, r$ )  
7              $r \leftarrow q - 1$ 
```

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda
Correto: substitua a chamada da maior metade!

Dicas de implementação: Quicksort

QUICKSORT (A, p, r)

```
1  enquanto  $p < r$            ← virou enquanto aqui!  
2       $q \leftarrow$  PARTICIONE( $A, p, r$ )  
3      se  $q - p < r - q$   
4          então QUICKSORT( $A, p, q - 1$ )  
5               $p \leftarrow q + 1$   
6      senão QUICKSORT( $A, q + 1, r$ )  
7           $r \leftarrow q - 1$ 
```

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda

Correto: substitua a chamada da maior metade!

Profundidade da pilha da recursão: $O(\lg n)$.

Dicas de implementação

Inserção direta para vetores pequenos.

Mergesort

- ▶ Onde declarar o vetor auxiliar usado no **INTERCALA**?
- ▶ Alternância entre dois vetores
- ▶ Teste se já ordenado

Quicksort

- ▶ Elementos repetidos: partição ternária
- ▶ Mediana de três
- ▶ Recursão de cauda

Heapsort

- ▶ Desce tudo sem comparar, depois sobe o que precisa.

Comparação entre os algoritmos

Mergesort:

Entre $\frac{1}{2} n \lg n$ e $n \lg n$ comparações.

Usa espaço extra linear.

Comparação entre os algoritmos

Mergesort:

Entre $\frac{1}{2} n \lg n$ e $n \lg n$ comparações.

Usa espaço extra linear.

Quicksort:

Em média, $2n \lg n$ comparações (e $\frac{1}{3} n \lg n$ trocas).

Espaço extra logarítmico.

Comparação entre os algoritmos

Mergesort:

Entre $\frac{1}{2} n \lg n$ e $n \lg n$ comparações.

Usa espaço extra linear.

Quicksort:

Em média, $2n \lg n$ comparações (e $\frac{1}{3} n \lg n$ trocas).

Espaço extra logarítmico.

Heapsort:

Menos de $2n \lg n + 2n$ comparações.

(Em média, cai para metade disso com a melhora.)

Espaço extra constante.

Performance de cache ruim.

Primeira prova

Data: 29 de abril (sexta), às 10h.

Duração: 2 horas (reserve mais um pouco).

Matéria da prova: tudo que vimos até a aula passada.

Primeira prova

Data: 29 de abril (sexta), às 10h.

Duração: 2 horas (reserve mais um pouco).

Matéria da prova: tudo que vimos até a aula passada.

Dúvidas? Perguntas?