Aula passada:

Todo algoritmo de ordenação
baseado em comparações faz

$$\Omega(n \lg n)$$

comparações no pior caso.

Aula passada:

Todo algoritmo de ordenação
baseado em comparações faz

$$\Omega(n \lg n)$$

comparações no pior caso.

Aula de hoje:

Algoritmos de ordenação lineares!

Ordenação em tempo linear

CLRS cap 8

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.

$\textsc{CountingSort}(A, n)$

para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i]$: nº 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)

para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$ $\qquad \triangleright$ $C[i]$: nº de ocorrências de i em A
para $i \leftarrow 2$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$ $\qquad \triangleright$ $C[i]$: nº 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.

$\textsc{CountingSort}(A, n)$

para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i]$: nº de ocorrências de i em A
para $i \leftarrow 2$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i]$: nº de elementos $\leq i$ em A
para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
devolva B

Counting Sort

$\textsc{CountingSort}(A, n)$

1 **para** $i \leftarrow 1$ **até** k **faça**
2 $\quad C[i] \leftarrow 0$
3 **para** $j \leftarrow 1$ **até** n **faça**
4 $\quad C[A[j]] \leftarrow C[A[j]] + 1$ $\qquad \triangleright$ $C[i]$: nº de ocorrências de i em A
5 **para** $i \leftarrow 2$ **até** k **faça**
6 $\quad C[i] \leftarrow C[i] + C[i-1]$ $\qquad \triangleright$ $C[i]$: nº de elementos $\leq i$ em A
7 **para** $j \leftarrow n$ **decrescendo até** 1 **faça**
8 $\quad B[C[A[j]]] \leftarrow A[j]$
9 $\quad C[A[j]] \leftarrow C[A[j]] - 1$
10 **devolva** B

Por que escrevemos as linhas 7–9 dessa maneira?

Counting Sort

CountingSort(A, n)

1 **para** $i \leftarrow 1$ **até** k **faça**
2 $\quad C[i] \leftarrow 0$
3 **para** $j \leftarrow 1$ **até** n **faça**
4 $\quad C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \triangleright$ $C[i]$: nº de ocorrências de i em A
5 **para** $i \leftarrow 2$ **até** k **faça**
6 $\quad C[i] \leftarrow C[i] + C[i-1]$ $\quad \triangleright$ $C[i]$: nº de elementos $\leq i$ em A
7 **para** $j \leftarrow n$ **decrescendo até** 1 **faça**
8 $\quad B[C[A[j]]] \leftarrow A[j]$
9 $\quad C[A[j]] \leftarrow C[A[j]] - 1$
10 **devolva** B

Por que escrevemos as linhas 7–9 dessa 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 $A[j]$.

1	5	7	1	5	← sort key
e	f	a	b	y	← informaç satélite

1	1	5	5	7
e	b	f	y	a

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.

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?

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

CountingSort(A, n)

para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i]$: nº de ocorrências de i em A
para $i \leftarrow 2$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i]$: nº de elementos $\leq i$ em A
para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
devolva B

Consumo de tempo

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

Consumo de tempo

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

Counting Sort

```
COUNTINGSORT(A, n)
para i ← 1 até k faça
    C[i] ← 0
para j ← 1 até n faça
    C[A[j]] ← C[A[j]] + 1
para i ← 2 até k faça
    C[i] ← C[i] + C[i − 1]
para j ← n decrescendo até 1 faça
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
devolva B
```

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

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

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos
  - datas, contendo os campos, dia, mês e ano
  - nomes completos, contendo os campos “primeiro nome” e “sobrenome”

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos
  - datas, contendo os campos, dia, mês e ano
  - nomes completos, contendo os campos "primeiro nome" e "sobrenome"

dígito 1: menos significativo
dígito d: mais significativo

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito d: mais significativo

$\textsc{RadixSort}(A, n, d)$
para $i \leftarrow 1$ **até** d **faça**
    ORDENE(A, n, i)

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito 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 dígito dos números em A por meio de um algoritmo estável.

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito d: mais significativo

Fig. 8.3, CLRS2

329	720
457	355
657	436
839	457
436	657
720	329
355	839

RadixSort(A, n, d)
para $i \leftarrow 1$ **até** d **faça**
  Ordene(A, n, i)

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito d: mais significativo

Fig. 8.3, CLRS2

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

```
RadixSort(A, n, d)
  para i ← 1 até d faça
      ORDENE(A, n, i)
```

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito d: mais significativo

Fig. 8.3, CLRS2

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

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

Radix Sort

Algoritmo usado para ordenar

- inteiros não-negativos com d dígitos
- cartões perfurados
- registros cuja chave tem vários campos

dígito 1: menos significativo
dígito d: mais significativo

Fig. 8.3, CLRS2

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

RADIXSORT(A, n, d)

para $i \leftarrow 1$ **até** d **faça**
    ORDENE(A, n, i)

induç em i:

BASE: $i=1$ ← pela corretude do ORDENE

PASSO: $i>1$ ← pela HI + estabilid

Consumo de tempo do Radixsort

Depende do algoritmo ORDENE.

Consumo de tempo do Radixsort

Depende do algoritmo ORDENE.

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

Consumo de tempo do Radixsort

Depende do algoritmo ORDENE.

Se cada dígito 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 dígito 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 $d = \mathrm{O}(1)$)
e $k = \mathrm{O}(n)$, então o consumo de tempo é $\Theta(n)$.

Bucket Sort

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

Bucket Sort

Recebe um inteiro n e um vetor $A[1 \mathinner{..} n]$ 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

A =

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

n = # de buckets

$[0, 0.1)$ →

$[0.1, 0.2)$ →

⋮

$B[0]$:	.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

n buckets de igual comprimento dividindo $[0, 1)$

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.

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)
para i ← 0 até n − 1 faça
    B[i] ← NIL
para i ← 1 até n faça
    x ← A[i]
    Insira(B[⌊nx⌋], x)
para i ← 0 até n − 1 faça
    OrdeneLista(B[i])
C ← Concatene(B, n)
devolva C
```

Bucket Sort

$\textsc{BucketSort}(A, n)$

```
para i ← 0 até n − 1 faça
    B[i] ← NIL
para i ← 1 até n faça
    x ← A[i]
    INSIRA(B[⌊nx⌋], x)
para i ← 0 até n − 1 faça
    ORDENELISTA(B[i])
C ← CONCATENE(B, n)
devolva C
```

se $x \in \left[\frac{i}{n}, \frac{i+1}{n}\right)$, então
x deve ir p/ o i-ésimo bucket

$$x \in \left[\frac{i}{n}, \frac{i+1}{n}\right) \Rightarrow nx \in [i, i+1) \Leftrightarrow \lfloor nx \rfloor = i$$

Exemplos:

$B[0]$ guarda $[0, 1/n)$

$B[1]$ guarda $[1/n, 2/n)$

$\vdots$

$B[i]$ guarda $\left[i/n, \frac{i+1}{n}\right)$

$\vdots$

$B[n-1]$ guarda $\left[\frac{n-1}{n}, 1\right)$

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

Bucket Sort

BucketSort(A, n)

para $i \leftarrow 0$ **até** $n-1$ **faça**
    $B[i] \leftarrow$ NIL
para $i \leftarrow 1$ **até** n **faça**
    $x \leftarrow A[i]$
    Insira($B[\lfloor nx \rfloor], x$)
para $i \leftarrow 0$ **até** $n-1$ **faça**
    OrdeneLista($B[i]$)
$C \leftarrow$ Concatene(B, n)
devolva C

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

OrdeneLista(p): ordena a lista apontada por p

Bucket Sort

```
BucketSort(A, n)
    para i ← 0 até n − 1 faça
        B[i] ← NIL
    para i ← 1 até n faça
        x ← A[i]
        Insira(B[⌊nx⌋], x)
    para i ← 0 até n − 1 faça
        OrdeneLista(B[i])
    C ← Concatene(B, n)
    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 $B[0], \ldots, B[n-1]$.

Consumo de tempo: caso médio

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: caso médio

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: caso médio

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

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

Consumo de tempo: caso médio

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

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

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

Consumo de tempo: caso médio

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

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

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 $B[i]$

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

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

Consumo de tempo

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 $Y_i \le \binom{X_i}{2} = \frac{X_i(X_i - 1)}{2} \le \frac{X_i^2}{2} \le {X_i}^2$.

Consumo de tempo

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 $Y_i \leq \binom{X_i}{2} = \frac{X_i(X_i - 1)}{2} \leq \frac{X_i^2}{2} \leq {X_i}^2$.

Logo, $\mathbb{E}[Y_i] \leq \mathbb{E}[{X_i}^2] = \mathbb{E}\left[(\sum_{j=1}^{n} X_{ij})^2\right]$.

Consumo de tempo

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 $Y_i \leq \binom{X_i}{2} = \frac{X_i(X_i-1)}{2} \leq \frac{X_i^2}{2} \leq {X_i}^2$.

Logo, $\mathbb{E}[Y_i] \leq \mathbb{E}[{X_i}^2] = \mathbb{E}\Big[(\sum_{j=1}^n X_{ij})^2\Big]$.

$$\begin{aligned} \mathbb{E}\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] &= \mathbb{E}\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] \\ &= \mathbb{E}\left[\sum_{j=1}^n {X_{ij}}^2 + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n X_{ij} X_{ik}\right] \end{aligned}$$

Consumo de tempo

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 $Y_i \leq {X_i}^2$.

Logo, $$\mathbb{E}[Y_i] \leq \mathbb{E}\left[\sum_{j=1}^{n} {X_{ij}}^2 + \sum_{j=1}^{n}\sum_{\substack{k=1\\k\neq j}}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} \mathbb{E}\left[{X_{ij}}^2\right] + \sum_{j=1}^{n}\sum_{\substack{k=1\\k\neq j}}^{n} \mathbb{E}\left[X_{ij}X_{ik}\right]$$

Consumo de tempo

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 $Y_i \leq {X_i}^2$.

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

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

Consumo de tempo

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 $B[i]$.

Temos $\mathbb{E}[Y_i] \leq \sum_{j=1}^{n} \mathbb{E}\left[X_{ij}{}^2\right] + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \mathbb{E}\left[X_{ij} X_{ik}\right]$.

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

Consumo de tempo

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 $B[i]$.

Temos $\mathbb{E}[Y_i] \leq \sum_{j=1}^{n} \mathbb{E}[{X_{ij}}^2] + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \mathbb{E}[X_{ij} X_{ik}]$.

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

Vamos calcular sua esperança:

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

Consumo de tempo

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

Consumo de tempo

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

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

Consumo de tempo

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

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

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

Consumo de tempo

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

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

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

Logo,
$$
\begin{aligned}
\mathbb{E}[Y_i] &\le \sum_{j=1}^{n} \mathbb{E}\left[{X_{ij}}^2\right] + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \mathbb{E}\left[X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{1}{n^2} \\
&= \frac{n}{n} + n(n-1)\frac{1}{n^2} \\
&= 1 + (n-1)\frac{1}{n} = 2 - \frac{1}{n}.
\end{aligned}
$$

Consumo de tempo

Agora, seja $Y = \sum_{i=1}^{n} Y_i$.

Consumo de tempo

Agora, seja $Y = \sum_{i=1}^{n} Y_i$.

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

Consumo de tempo

Agora, seja $Y = \sum_{i=1}^{n} Y_i$.

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

Assim $\mathbb{E}[Y]$ 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 $Y = \sum_{i=1}^{n} Y_i$.

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

Assim $\mathbb{E}[Y]$ é o número esperado de comparações realizadas pelo algoritmo, e tal número determina o consumo assintótico de tempo do BucketSort.

Mas então $\mathbb{E}[Y] = \sum_{i=1}^{n} \mathbb{E}[Y_i] \leq 2n - 1 = \mathrm{O}(n)$.

Consumo de tempo

Agora, seja $Y = \sum_{i=1}^{n} Y_i$.

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

Assim $\mathbb{E}[Y]$ é o número esperado de comparações realizadas pelo algoritmo, e tal número determina o consumo assintótico de tempo do BucketSort.

Mas então $\mathbb{E}[Y] = \sum_{i=1}^{n} \mathbb{E}[Y_i] \leq 2n - 1 = \mathrm{O}(n)$.

> O consumo de tempo esperado do BucketSort quando os números em $A[1\,.\,.\,n]$ são uniformemente distribuídos no intervalo $[0, 1)$ é $\mathrm{O}(n)$.