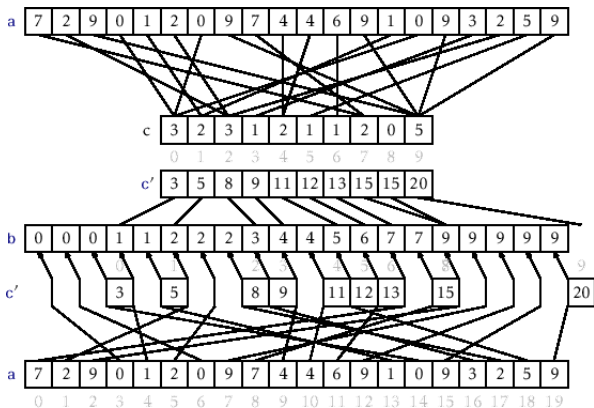


AULA 19

Ordenação de strings



Fonte: [Counting Sort and Radix Sort](#)

Referências: [String sorts \(SW\)](#); [slides \(SW\)](#); [LSD, video \(SW\)](#); [MSD, video \(SW\)](#);

Ordenação em tempo linear

Key-indexed counting

Referência: String sorts (SW);

Ordenação por contagem

Recebe um vetor $a[0..n-1]$ e ordena seus elementos.

Cada $a[i]$ está em $\{0, \dots, R-1\}$.

Entra:

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

Ordenação por contagem

Recebe um vetor $a[0..n-1]$ e ordena seus elementos.

Cada $a[i]$ está em $\{0, \dots, R-1\}$.

Entra:

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

Sai:

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

Passo 1: calcula frequências

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0	1	2	3	4	5	6	7	8	9
a	2	5	3	0	2	3	0	5	3	0
aux										

Passo 1: calcula frequências

Cada $a[i]$ está em $\{0, \dots, 5\}$.

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

	0	1	2	3	4	5	6	7	8	9
aux										

	0	1	2	3	4	5	6
count							

Passo 1: calcula frequências

Cada $a[i]$ está em $\{0, \dots, 5\}$.

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

	0	1	2	3	4	5	6	7	8	9
aux										

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

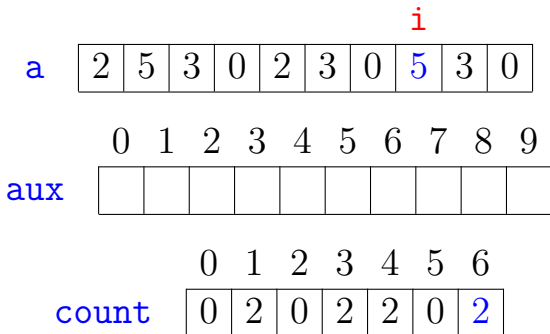
Passo 1: calcula frequências

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	i									
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
$count$	0	0	0	1	0	0	0			

Passo 1: calcula frequências

Cada $a[i]$ está em $\{0, \dots, 5\}$.



Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	0	2	3	0	2			

Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	3	2	3	0	2			

Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	3	5	3	0	2			

Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	3	5	8	0	2			

Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	3	5	8	8	2			

Passo 2: transforma frequências em índices

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	0								9	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux										
	0	1	2	3	4	5	6			
count	0	3	3	5	8	8	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

	i										
a		2	5	3	0	2	3	0	5	3	0
		0	1	2	3	4	5	6	7	8	9
aux											
		0	1	2	3	4	5	6			
$count$		0	3	3	5	8	8	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

				i						
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux				2						
count	0	3	4	5	8	8	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

				i						
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux				2					5	
	0	1	2	3	4	5	6			
count	0	3	4	5	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

				i						
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux				2		3			5	
	0	1	2	3	4	5	6			
count	0	3	4	6	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

				i						
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0			2		3			5	
	0	1	2	3	4	5	6			
count	1	3	4	6	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

					i					
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0			2	2	3			5	
	0	1	2	3	4	5	6			
count	1	3	5	6	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

						i				
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0			2	2	3	3		5	
	0	1	2	3	4	5	6			
count	1	3	5	7	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

							i			
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0		2	2	3	3		5	
	0	1	2	3	4	5	6			
count	2	3	5	7	8	9	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

								i		
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0		2	2	3	3		5	5
	0	1	2	3	4	5	6			
count	2	3	5	7	8	10	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

									i	
a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0		2	2	3	3	3	5	5
	0	1	2	3	4	5	6			
count	2	3	5	8	8	10	10			

Passo 3: distribuição das chaves

Cada $a[i]$ está em $\{0, \dots, 5\}$.

a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0	0	2	2	3	3	3	5	5
	0	1	2	3	4	5	6			
count	3	3	5	8	8	10	10			

Passo 4: copia chaves ordenadas para a

Cada $a[i]$ está em $\{0, \dots, 5\}$.

a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0	0	2	2	3	3	3	5	5
	0	1	2	3	4	5	6			
count	3	3	5	8	8	10	10			

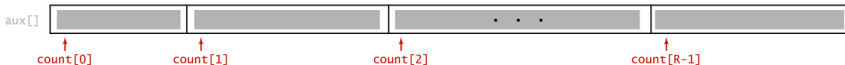
Passo 4: copia chaves ordenadas para a

Cada $a[i]$ está em $\{0, \dots, 5\}$.

a	0	0	0	2	2	3	3	3	5	5
	0	1	2	3	4	5	6	7	8	9
aux	0	0	0	2	2	3	3	3	5	5
	0	1	2	3	4	5	6			
count	3	3	5	8	8	10	10			

Ilustração da fase de distribuição

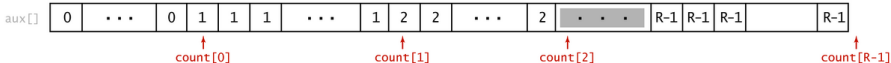
before



during



after



Key-indexed counting (distribution phase)

Fonte: [algs4](#)

Ordenação por contagem

```
int n = a.length;
int[] count = new int[R+1];
1 for (int i = 0; i < n; i++)
2     count[a[i]+1]++;
3 for (int r = 0; r < R; r++)
4     count[r+1] += count[r];
// fase de distribuição
5 for (int i = 0; i < n; i++)
6     aux[count[a[i]]++] = a[i];
7 for (int i = 0; i < n; i++)
8     a[i] = aux[i];
```

Obs: não são feitas **comparações** entre **chaves**.

Consumo de tempo

linha	consumo na linha
1–2	$\Theta(n)$
3–4	$\Theta(R)$
5–6	$\Theta(n)$
7–9	$\Theta(n)$

Consumo total: $\Theta(n + R)$

Conclusões

O consumo de tempo da ordenação por contagem é $\Theta(n + R)$.

- ▶ se $R \leq n$ então consumo é $\Theta(n)$
- ▶ se $R \leq 10n$ então consumo é $\Theta(n)$
- ▶ se $R = O(n)$ então consumo é $\Theta(n)$
- ▶ se $R \geq n^2$ então consumo é $\Theta(R)$
- ▶ se $R = \Omega(n)$ então consumo é $\Theta(R)$

Estabilidade

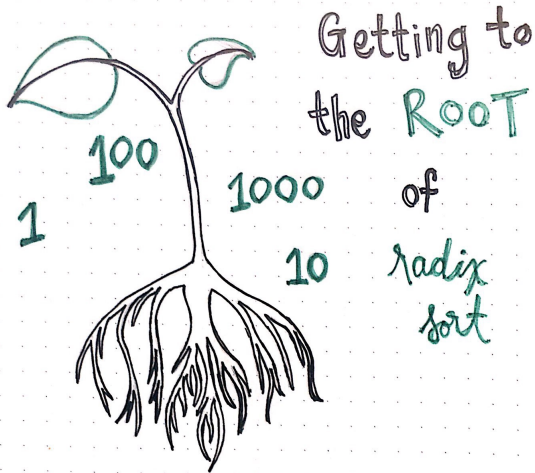
A propósito: **ordenação por contagem** é **estável**:
na saída, chaves com mesmo valor estão na mesma ordem que apareciam na entrada.

a	2	5	3	0	2	3	0	5	3	0
	0	1	2	3	4	5	6	7	8	9
aux	0	0	0	2	2	3	3	3	5	5

Características

- ▶ **Supõe** que as chaves ($=key$) são inteiros entre 0 e $R-1$.
- ▶ **Usado** como subrotina em algoritmos de ordenação.
- ▶ **Conta** frequência usado “ key ” como índice.
- ▶ **Transforma** as frequências em destino dos valores.
- ▶ **Supera** o **limite inferior de ordenação** pois evita comparações entre chaves (não há `compareTo()`).

Radix



Fonte: [Getting To The Root Of Sorting With Radix Sort](#)

Raiz (*radix*)

Raiz (= *radix*) é um outro termo para *base*.

A raiz nos diz o número R de **dígitos** ou **símbolos** ou **caracteres** ou **bits** ou ... que usamos para representar número ou string.

R é também dito o tamanho do **alfabeto**.

Ordenação digital (*radix sorting*)

Ordenação digital (= *radix sorting*) ordena chaves (sobre um alfabeto) agrupando-as conforme os símbolos (do alfabeto) em determinadas posições, frequentemente usando ordenação por contagem como subrotina para implementar a ordenação.

Se as chaves são inteiros os símbolos podem ser seus bytes.

Se as chaves são strings os símbolos podem ser seus caracteres.

LSD e MSD

A **ordenação digital** aparece frequentemente em dois sabores:

- ▶ **Least significant digit (LSD)**: trabalha examinando as chaves, representadas por inteiros, começando do **dígito menos** significativo e prosseguindo até o **dígito mais** significativo. A implementação é **usualmente iterativa** e usa ordenação por contagem.
- ▶ **Most significant digit (MSD)**: trabalha examinando as chaves, representadas por inteiros, começando do **dígito mais** significativo e prosseguindo até o **dígito menos** significativo. A implementação é **usualmente recursiva** e usa ordenação por contagem.

LSD e MSD

362	291	207	207	237	237	216	211
436	362	436	253	318	216	211	216
291	253	253	291	216	211	237	237
487	436	362	362	462	268	268	268
207	487	487	397	211	318	318	318
253	207	291	436	268	462	462	460
397	397	397	487	460	460	460	462

LSD Radix Sorting:

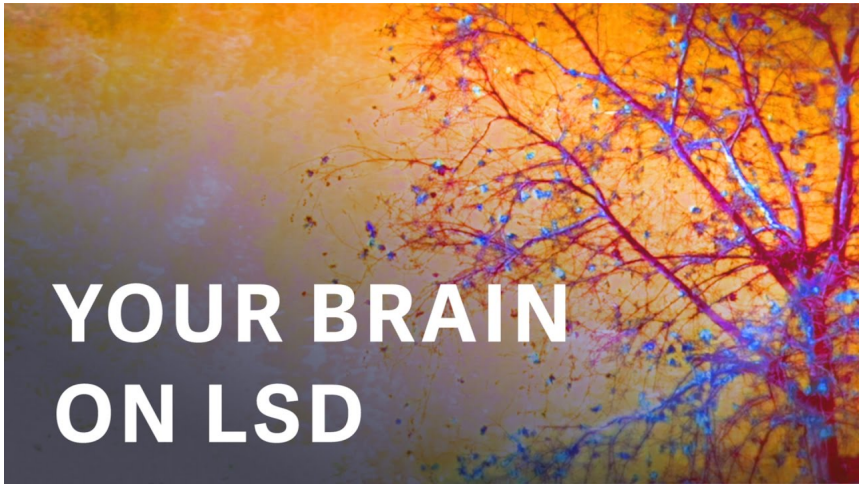
Sort by the last digit, then
by the middle and the first one

MSD Radix Sorting:

Sort by the first digit, then sort
each of the groups by the next digit

Fonte: [Radix sort in C](#)

Least-Significant-Digit



Fonte: [The first modern images of a human brain on LSD](#)

LSD ideia

Exemplo:

329

457

657

839

436

720

355

LSD ideia

Exemplo:

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

LSD ideia

Exemplo:

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

LSD ideia

Exemplo:

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

LSD ideia

Exemplo:

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

Cada $a[j]$ têm d dígitos decimais:

$$a[j] = a_d 10^{d-1} + \dots + a_2 10^1 + a_1 10^0$$

Exemplo com $d = 3$: $3 \cdot 10^2 + 2 \cdot 10 + 9$

LSD candidato

input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	10HV845
1ICK750	10HV845
10HV845	10HV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
10HV845	3ATW723
10HV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
<u>3ATW723</u>	4PGC938

↑
*keys are all
the same length*

Fonte: [algs4](#)

Typical candidate for
LSD string sort

LSD

```
public class LSD {  
    public static void sort(String[] a,  
        int W){  
        int R = 256; // extended ASCII  
        int n = a.length;  
        String[] aux = new String[n];
```


LSD

```
for(int d = W-1; d >= 0; d--){
    int[] count = new int[R+1];
    for (int i = 0; i < n; i++)
        count[a[i].charAt(d)+1]++;
    for (int r = 0; r < R; r++)
        count[r+1] += count[r];
    for (int i = 0; i < n; i++)
        aux[count[a[i].charAt(d)]++] = a[i];
    for (int i = 0; i < n; i++)
        a[i] = aux[i];
}
}
}
```

Exemplos

- ▶ dígitos decimais: $\Theta(Wn)$
- ▶ dígitos em $0 \dots R-1$: $\Theta(W(n + R))$.

Exemplo com $d = 5$ e $R = 128$:

$$a[4]128^4 + a[3]128^3 + a[2]128^2 + a[1]128 + a[0]$$

sendo $0 \leq a[i] \leq 127$

Conclusão

Dados n números com b bits e um inteiro $r \leq b$, LSD ordena esses números em tempo

$$\Theta\left(\frac{b}{r}(n + 2^r)\right).$$

Prova: Considere cada chave com $d = \lceil b/r \rceil$ dígitos com r bits cada.

Use ordenação por contagem com $R = 2^r - 1$.

Cada passada do ordenação por contagem:

$$\Theta(n + R) = \Theta(n + 2^r).$$

$$\text{Tempo total: } \Theta(d(n + 2^r)) = \Theta\left(\frac{b}{r}(n + 2^r)\right).$$

LSD simulação

input ($W=7$)	$d=6$	$d=5$	$d=4$	$d=3$	$d=2$	$d=1$	$d=0$	output
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	10HV845	10HV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	10HV845	1ICK750	10HV845	10HV845
10HV845	3CIO720	2RLA629	3CIO720	1ICK750	10HV845	1ICK750	10HV845	10HV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	10HV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	10HV845	4PGC938	1ICK750	3CIO720	3CIO720	10HV845	2RLA629	2RLA629
10HV845	10HV845	10HV845	1ICK750	10HV845	2RLA629	10HV845	3ATW723	3ATW723
10HV845	10HV845	10HV845	10HV845	10HV845	2RLA629	10HV845	3CIO720	3CIO720
2RLA629	4PGC938	10HV845	10HV845	10HV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	10HV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Fonte: [algs4](#)

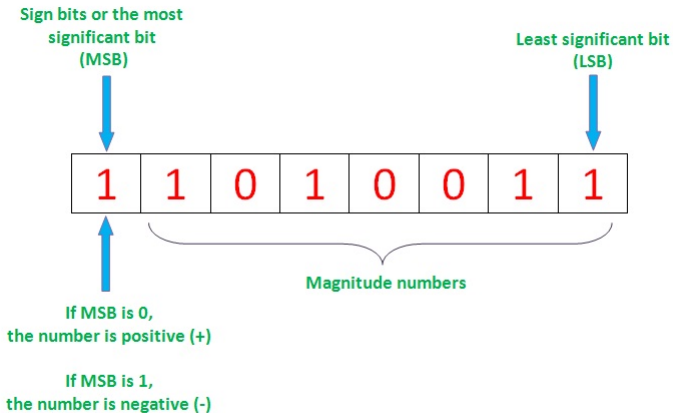
LSD com baralho

♣ J	♦ A	♠ A
♥ 6	♥ A	♠ 2
♦ A	♣ A	♠ 3
♥ A	♠ A	♠ 4
♠ K	♠ 2	♠ 5
♥ J	♣ 2	♠ 6
♦ Q	♥ 2	♠ 7
♣ 6	♦ 2	♠ 8
♠ J	♥ 3	♠ 9
♣ A	♠ 3	♠ 10
♦ 9	♣ 3	♠ J
♥ 9	♦ 3	♠ Q
♦ 8	♦ 4	♠ K
♠ 9	♣ 4	♥ A
♣ K	♥ 4	♥ 2
♦ 4	♠ 4	♥ 3
♠ 5	♠ 5	♥ 4
♣ Q	♦ 5	♥ 5
♥ 3	♣ 5	♥ 6
♠ 2	♥ 5	♥ 7
♣ 10	♥ 6	♥ 8
♣ 9	♣ 6	♥ 9
♥ 7	♠ 6	♥ 10
♣ 4	♦ 6	♥ J
♥ 4	♥ 7	♥ Q
♦ 10	♣ 7	♥ K
♠ A	♠ 7	♦ A
♦ 5	♦ 7	♦ 2
♠ 3	♦ 8	♦ 3
♥ 8	♥ 8	♦ 4
♣ 2	♠ 8	♦ 5
♦ K	♣ 8	♦ 6
♠ 4	♦ 9	♦ 7
♣ 7	♥ 9	♦ 8
♥ Q	♠ 9	♦ 9
♦ 3	♠ 10	♦ 10

LSD características

- ▶ **Exige** strings de comprimento fixo; isso pode ser contornado com uma espécie de **padding**.
- ▶ **Considera** caracteres da **direita** para a **esquerda**.
- ▶ **Algoritmo utilizado** para ordenar o caractere **d** das strings **deve ser estável**.
- ▶ **Faz** cerca de Wn chamadas de `charAt()`.
- ▶ **Utiliza** espaço extra proporcional a $n + R$.

Most-Significant-Digit

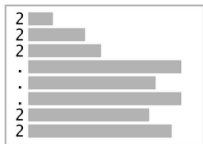
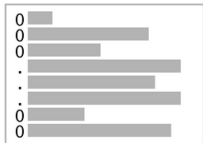


Fonte: Complement Number System

MSD ideia

*sort on first character value
to partition into subarrays*

*recursively sort subarrays
(excluding first character)*



...

MSD candidato

input

she
sells
seashells
by
the
seashore
the
shells
she
sells
are
surely
seashells

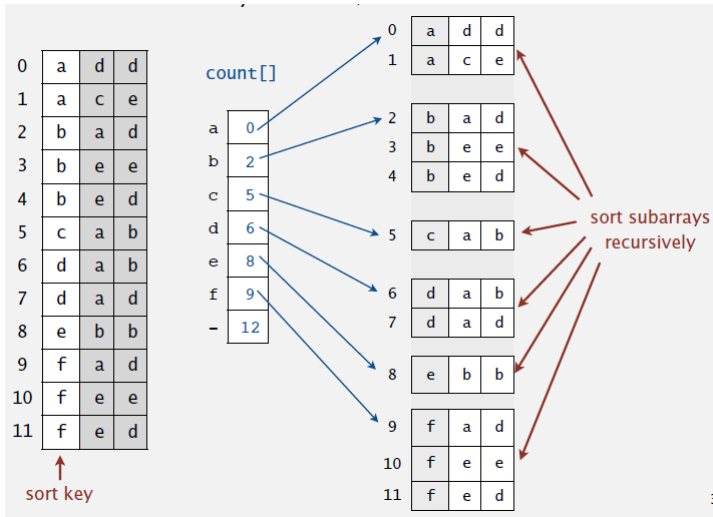
sorted result

are
by
seashells
seashells
seashore
sells
sells
she
she
shells
surely
the
the

*various
key
lengths*



MSD recursão



Fonte: [algs4](#)

MSD

```
public class MSD {  
    private static final int R= 256;  
    // corte para usar inserção  
    private static final int M = 15;  
    public static void sort(String[] a) {  
        int n = a.length;  
        String[] aux= new String[n];  
        sort(a, 0, n-1, 0, aux);  
    }  
}
```

MSD

```
private static int charAt(String s,  
    int d) {  
    if (d == s.length()) return -1;  
    return s.charAt(d);  
}  
  
private static void sort(String[] a,  
    int lo int hi, int d,  
    String[] aux) {  
    if (hi <= lo+ M) {  
        insertion(a, lo, hi, d);  
        return;  
    }  
    int[] count = new int[R+2];
```

MSD

```
for (int i = lo; i <= hi; i++) {
    int c = charAt(a[i], d);
    count[c+2]++; }
for (int r = 0; r < R+1; r++)
    count[r+1] += count[r];
for (int i = lo; i <= hi; i++) {
    int c = charAt(a[i], d);
    aux[count[c+1]++] = a[i]; }
for (int i = lo; i <= hi; i++)
    a[i] = aux[i - lo];
for (int r = 0; r < R; r++)
    sort(a, lo+count[r],
        lo+count[r+1]-1, d+1, aux);
} } }
```

MSD caracteres examinados

random (sublinear)	nonrandom with duplicates (nearly linear)	worst case (linear)
1EI0402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Fonte: [algs4](#)

MSD

use key-indexed counting on first character

		<i>count frequencies</i>	<i>transform counts to indices</i>	<i>distribute and copy back</i>	
0	she	0	0	0	are
1	sells	1 a	1 a	1	by
2	seashells	2 b	2 b	2	she
3	by	3 c	3 c	3	sells
4	the	4 d	4 d	4	seashells
5	seashore	5 e	5 e	5	seashore
6	the	6 f	6 f	6	shells
7	shells	7 g	7 g	7	she
8	she	8 h	8 h	8	sells
9	sells	9 i	9 i	9	surely
10	are	10 j	10 j	10	seashells
11	surely	11 k	11 k	11	the
12	seashells	12 l	12 l	12	the
		13 m	13 m		
		14 n	14 n		
		15 o	15 o		
		16 p	16 p		
		17 q	17 q		
		18 r	18 r		
		19 s	19 s		
		20 t	20 t		
		21 u	21 u		
		22 v	22 v		
		23	23		

start of s subarray (arrow pointing to index 19 in the 'transform counts to indices' column)

1 + end of s subarray (arrow pointing to index 20 in the 'transform counts to indices' column)

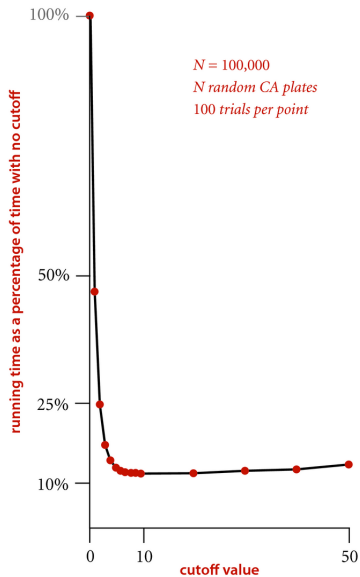
MSD em ação

input

she	are	are	are	are	are	are
sells	by	by	by	by	by	by
seashells	she	sells	seashells	sea	sea	sea
by	sells	seashells	sea	seashells	seashells	sea
the	seashells	sea	seashells	seashells	seashells	sea
sea	sea	sells	sells	sells	sells	se
shore	shore	seashells	sells	sells	sells	sel
the	shells	she	she	she	she	she
shells	she	shore	shore	shore	shore	sho
she	sells	shells	shells	shells	shells	she
sells	surely	she	she	she	she	she
are	seashells	surely	surely	surely	surely	sur
surely	the	the	the	the	the	the
seashells	the	the	the	the	the	the

*need to examine
every character
in equal keys*

MSD



MSD com baralho

♣ J	♠ K	♠ A
♥ 6	♠ J	♠ 2
♦ A	♠ 9	♠ 3
♥ A	♠ 5	♠ 4
♠ K	♠ 2	♠ 5
♥ J	♠ A	♠ 6
♦ Q	♠ 3	♠ 7
♣ 6	♠ 4	♠ 8
♠ J	♠ 6	♠ 9
♣ A	♠ 7	♠ 10
♦ 9	♠ 8	♠ J
♥ 9	♠ 10	♠ Q
♦ 8	♠ Q	♠ K
♠ 9	♥ 6	♥ A
♣ K	♥ A	♥ 2
♦ 4	♥ J	♥ 3
♠ 5	♥ 9	♥ 4
♣ Q	♥ 3	♥ 5
♥ 3	♥ 7	♥ 6
♠ 2	♥ 4	♥ 7
♣ 10	♥ 8	♥ 8
♣ 9	♥ Q	♥ 9
♥ 7	♥ 10	♥ 10
♣ 4	♥ 2	♥ J
♥ 4	♥ K	♥ Q
♦ 10	♥ 5	♥ K
♠ A	♦ A	♦ A
♦ 5	♦ Q	♦ 2
♠ 3	♦ 9	♦ 3
♥ 8	♦ 8	♦ 4
♣ 2	♦ 4	♦ 5
♦ K	♦ 10	♦ 6
♠ 4	♦ 5	♦ 7
♣ 7	♦ K	♦ 8
♥ Q	♦ J	♦ 9
♦ 3	♦ 3	♦ 10

MSD características

- ▶ **Particiona** o vetor em R segundo o caractere sendo examinado.
- ▶ **Recursivamente** ordena todas as strings agrupadas segundo os d caracteres examinados.
- ▶ **Strings de tamanho variado**: trata as strings como se tivessem ao final um caractere **menor que todos** do alfabeto.
- ▶ **no pior caso** usa espaço $n + R \times W$ (W = maior comprimento de uma string).
- ▶ **na média** examina cerca de $n \log_R n$ caracteres.

MSD características

Problemas de desempenho:

- ▶ Lento para subvetores pequenos; cada chamada tem o seu vetor `count[]`;
- ▶ número grande de subvetores por causa da recursão.

Solução:

- ▶ usar ordenação por inserção para subvetores pequenos;
- ▶ ordenação por inserção começa após `d` caracteres;
- ▶ ordenação por inserção compara a partir do caractere `d`.

MSD versus quicksort para strings

Desvantagens do MSD:

- ▶ Espaço extra para `aux []` devido a ordenação por contagem.
- ▶ Espaço extra para `count []` devido a ordenação por contagem.
- ▶ Laço interno com muitas instruções devido a ordenação por contagem.
- ▶ acesso aleatório da memória faz com que seja *cache inefficient*.

MSD versus quicksort para strings

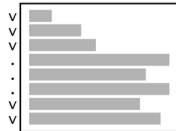
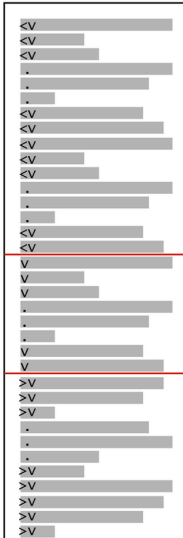
Desvantagens de usar quicksort para strings:

- ▶ **número de comparações** entre strings é $O(n \log n)$ e não linear.
- ▶ **deve examinar** várias vezes os mesmos caracteres de chaves com longos prefixos iguais.

3-way string quicksort: ideia

*use first character value
to partition into “less,” “equal,”
and “greater” subarrays*

*recursively sort subarrays
(excluding first character
for “equal” subarray)*



3-way string quicksort: candidato

input

edu.princeton.cs
com.apple
edu.princeton.cs
com.cnn
com.google
edu.uva.cs
edu.princeton.cs
edu.princeton.cs.www
edu.uva.cs
edu.uva.cs
edu.uva.cs
com.adobe
edu.princeton.ee

sorted result

com.adobe
com.apple
com.cnn
com.google
edu.princeton.cs
edu.princeton.cs
edu.princeton.cs
edu.princeton.cs.www
edu.princeton.ee
edu.uva.cs
edu.uva.cs
edu.uva.cs
edu.uva.cs
edu.uva.cs

*long
prefix
match*



*duplicate
keys*



Fonte: [algs4](#)

Quick3string

```
public class Quick3string{
    private static final int M = 15;

    public static void sort(String[] a) {
        sort(a, 0, a.length-1, 0);
    }

    private static int charAt(String s,
        int d) {
        if (d == s.length()) return -1;
        return s.charAt(d);
    }
}
```

Quick3string

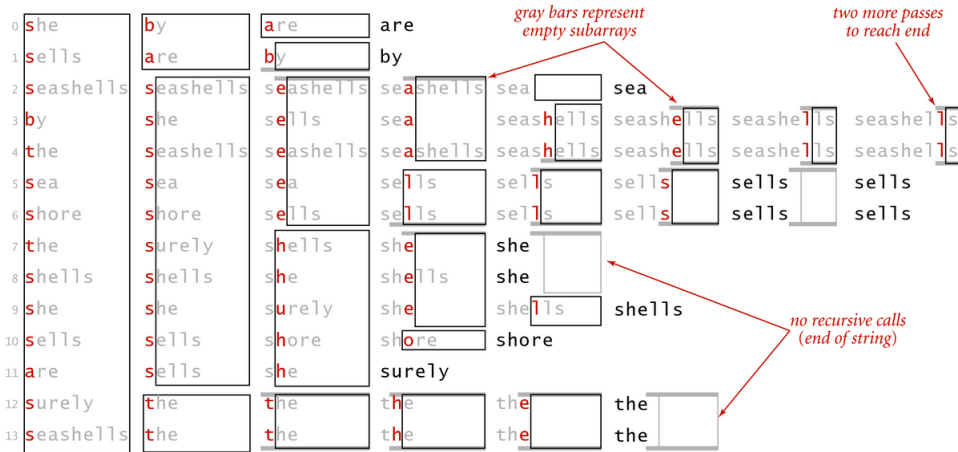
```
// 3-way string quicksort a[lo..hi]
// começando no caractere d
private static void sort(String[] a,
    int lo, int hi, int d) {
    if (hi <= lo + M) {
        insertion(a, lo, hi, d);
        return;
    }
}
```

Quick3string

```
int lt = lo, gt = hi;
int v = charAt(a[lo], d);
int i = lo + 1;
while (i <= gt) {
    int t = charAt(a[i], d);
    if (t < v)  exch(a, lt++, i++);
    else if (t > v) exch(a, i, gt--);
    else i++;
}
sort(a, lo, lt-1, d);
if (v >= 0) sort(a, lt, gt, d+1);
sort(a, gt+1, hi, d);
```

```
}
```

3-way string quicksort simulação



Trace of recursive calls for 3-way string quicksort (no cutoff for small subarrays)

Fonte: [algs4](#)

3-way string quicksort características

- ▶ Faz *3-way partition* segundo o d -ésimo caractere.
- ▶ Menos pesada que a R -way partition do MSD.
- ▶ Não reexamina os caracteres iguais ao caractere pivô; mas reexamina os caracteres diferentes do pivô.
- ▶ quicksort padrão faz na média aproximadamente $2n \ln n$ comparações entre chaves: caro para chaves com prefixos comuns longos.

Resumo

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	yes	compareTo()
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$	no	compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	compareTo()
LSD †	$2 N W$	$2 N W$	$N + R$	yes	charAt()
MSD ‡	$2 N W$	$N \log_R N$	$N + D R$	yes	charAt()
3-way string quicksort	$1.39 W N \lg N^*$	$1.39 N \lg N$	$\log N + W$	no	charAt()

Fonte: [algs4](#)

Suffix array

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

Suffix array

input string

```
i t w a s b e s t i t w a s w  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

form suffixes



```
0 i t w a s b e s t i t w a s w  
1 t w a s b e s t i t w a s w  
2 w a s b e s t i t w a s w  
3 a s b e s t i t w a s w  
4 s b e s t i t w a s w  
5 b e s t i t w a s w  
6 e s t i t w a s w  
7 s t i t w a s w  
8 t i t w a s w  
9 i t w a s w  
10 t w a s w  
11 w a s w  
12 a s w  
13 s w  
14 w
```

sort suffixes to bring repeated substrings together



```
3 a s b e s t  
12 a s w  
5 b e s t i t w a s w  
6 e s t i t w a s w  
0 i t w a s b e s t i t w a s w  
9 i t w a s w  
4 s b e s t i t w a s w  
7 s t i t w a s w  
13 s w  
8 t i t w a s w  
1 t w a s b e s t i t w a s w  
10 t w a s w  
14 w  
2 w a s b e s t i t w a s w  
11 w a s w
```

Suffix array

Text	a	b	r	a	c	a	d	a	b	r	a
Index	0	1	2	3	4	5	6	7	8	9	10

Fonte: [Brief Introduction to Suffix Array](#)

Suffix array

Suffix										Index	
a	b	r	a	c	a	d	a	b	r	a	0
	b	r	a	c	a	d	a	b	r	a	1
		r	a	c	a	d	a	b	r	a	2
			a	c	a	d	a	b	r	a	3
				c	a	d	a	b	r	a	4
					a	d	a	b	r	a	5
						d	a	b	r	a	6
							a	b	r	a	7
								b	r	a	8
									r	a	9
										a	10

Fonte: [Brief Introduction to Suffix Array](#)

Suffix array

Sorted Suffix	Index
a	10
a b r a	7
a b r a c a d a b r a	0
a c a d a b r a	3
a d a b r a	5
b r a	8
b r a c a d a b r a	1
c a d a b r a	4
d a b r a	6
r a	9
r a c a d a b r a	2

Fonte: [Brief Introduction to Suffix Array](#)

Aplicação: Longest Repeated Substring

input string

```
i t w a s b e s t i t w a s w  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

form suffixes

```
0 i t w a s b e s t i t w a s w  
1 t w a s b e s t i t w a s w  
2 w a s b e s t i t w a s w  
3 a s b e s t i t w a s w  
4 s b e s t i t w a s w  
5 b e s t i t w a s w  
6 e s t i t w a s w  
7 s t i t w a s w  
8 t i t w a s w  
9 i t w a s w  
10 t w a s w  
11 w a s w  
12 a s w  
13 s w  
14 w
```

sort suffixes to bring repeated substrings together

```
3 a s b e s t  
12 a s w  
5 b e s t i t w a s w  
6 e s t i t w a s w  
0 i t w a s b e s t i t w a s w  
9 i t w a s w  
4 s b e s t i t w a s w  
7 s t i t w a s w  
13 s w  
8 t i t w a s w  
1 t w a s b e s t i t w a s w  
10 t w a s w  
14 w  
2 w a s b e s t i t w a s w  
11 w a s w
```

Fonte: [algs4](#)

Aplicação: Longest Repeated Substring

input string

```
a a c a a g t t t a c a a g c
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

form suffixes

```
0 a a c a a g t t t a c a a g c
1 a c a a g t t t a c a a g c
2 c a a g t t t a c a a g c
3 a a g t t t a c a a g c
4 a g t t t a c a a g c
5 g t t t a c a a g c
6 t t t a c a a g c
7 t t a c a a g c
8 t a c a a g c
9 a c a a g c
10 c a a g c
11 a a g c
12 a g c
13 g c
14 c
```

sort suffixes to bring repeated substrings together

```
0 a a c a a g t t t a c a a g c
11 a a g c
3 a a g t t t a c a a g c
9 a c a a g c
1 a c a a g t t t a c a a g c
12 a g c
4 a g t t t a c a a g c
14 c
10 c a a g c
2 c a a g t t t a c a a g c
13 g c
5 g t t t a c a a g c
8 t a c a a g c
7 t t a c a a g c
6 t t t a c a a g c
```

compute longest prefix between adjacent suffixes

```
a a c a a g t t t a c a a g c
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Manber e Meyers

Ordenação dos sufixos de uma string em tempo $O(n \log n)$.

Algoritmo é iterativo:

Cada iteração começa com o vetor dos sufixos ordenados de acordo com os 2^d primeiros caracteres.

No início da primeira iteração temos o vetor dos sufixos ordenados de acordo com o primeiro caractere. Esse vetor é obtida através de ordenação por contagem como o MSD.

Cada iteração consiste em construir o vetor dos sufixos ordenados de acordo com os 2^{d+1} primeiros caracteres.

Manber e Meyers

Manber e Meyers mostraram como cada iteração pode ser realizada em **tempo linear**.

Como o número de iterações é $\lg n$ o consumo de tempo do algoritmo é proporcional a $n \lg n$.

Ideia de Manber e Meyers

original suffixes		index sort (first four characters)		inverse[]	
0	b a b a a a a b c b a b a a a a a 0	17	0	0	14
1	a b a a a a b c b a b a a a a a 0	16	a 0	1	9
2	b a a a a b c b a b a a a a a 0	15	a a 0	2	12
3	a a a a b c b a b a a a a a 0	14	a a a 0	3	4
4	a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0	4	7
5	a a b c b a b a a a a a 0	12	a a a a a 0	5	8
6	a b c b a b a a a a a 0	13	a a a a a 0	6	11
7	b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0	7	16
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0	8	17
9	b a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0	9	15
10	a b a a a a a 0	10	a b a a a a a 0	10	10
11	b a a a a a 0	6	a b c b a b a a a a a 0	11	13
12	a a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0	12	5
13	a a a a 0	11	b a a a a a 0	13	6
14	a a a 0	0	b a b a a a a b c b a b a a a a a 0	14	3
15	a a 0	9	b a b a a a a a 0	15	2
16	a 0	7	b c b a b a a a a a 0	16	1
17	0	8	c b a b a a a a a 0	17	0

$0 + 4 = 4$

$9 + 4 = 13$

$\text{suffixes}_4[13] \leq \text{suffixes}_4[4]$ (because $\text{inverse}[13] < \text{inverse}[4]$)
 so $\text{suffixes}_8[9] \leq \text{suffixes}_8[0]$

