Compacto dos melhores momentos

AULA 21 e 22

BFS versus DFS

- busca em largura usa **fila**, busca em profundidade usa **pilha**
- a busca em largura é descrita em **estilo iterativo**, enquanto a busca em profundidade é descrita, usualmente, em **estilo recursivo**
- busca em largura começa tipicamente num **vértice especificado**, a busca em profundidade, o próprio **algoritmo escolhe o vértice** inicial
- a busca em largura apenas **visita os vértices que podem ser atingidos** a partir do vértice inicial, a busca em profundidade, tipicamente, **visita todos os vértices** do digrafo

`DFS` aplicações

- caminhos x cortes: `DFSpaths`
- ciclos x ordenação topológica (DAGs): `DFStopological`
- componentes conexos: `DFScc`
- grafos bipartidos x ciclos ímpares: `DFScc`
- componentes fortemente conexas: `DFSscc`
- . . .

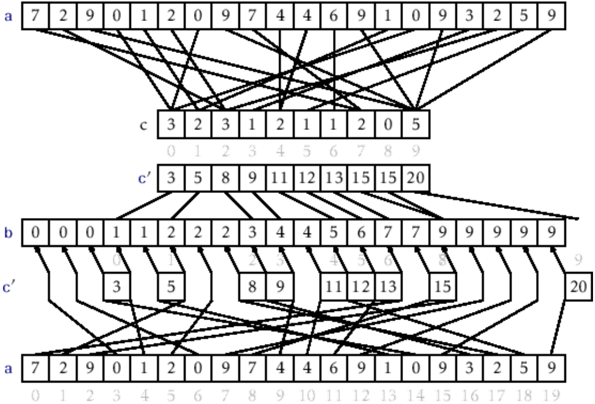
BFS aplicações

- caminhos x cortes: `BFSpaths`
- componentes conexos
- caminhos comprimento mínimo: usa `Queue`
- caminhos de custo/peso mínimo
  - `Dijkstra`: **custo não negativos**, usa `MinPQ`
  - `AcyclicSP`: custos quaisquer, usa `DFStopological`
  - ...
- árvores de custo mínimo/máximo: `PrimMST`

Lembrar o 8 Puzzle de COS226 e sua busca $\mathtt{A}^*$.

AULA 23

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 $\mathtt{a}[0\,.\,.\,\mathtt{n}{-}1]$ e ordena seus elementos.

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

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 $\texttt{a}[0\,.\,.\,\texttt{n}{-}1]$ e ordena seus elementos.

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

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 $\{0,\ldots,5\}$.

	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`										

Passo 1: calcula frequências

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

	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 $\{0, \ldots, 5\}$.

	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 $\{0, \ldots, 5\}$.

	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 $\{0,\ldots,5\}$.

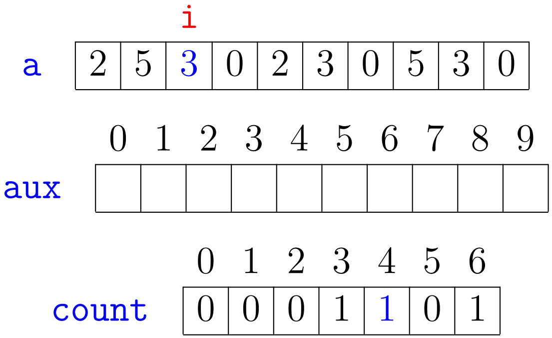
		i								
a	2	5	3	0	2	3	0	5	3	0

	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	1

Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



			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	1	0	1

Passo 1: calcula frequências

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

				`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	1	0	1	1	0	1

Passo 1: calcula frequências

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

					`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	1	0	2	1	0	1

Passo 1: calcula frequências

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

						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	1	0	2	2	0	1

Passo 1: calcula frequências

Cada a[i] está em $\{0,\ldots,5\}$.

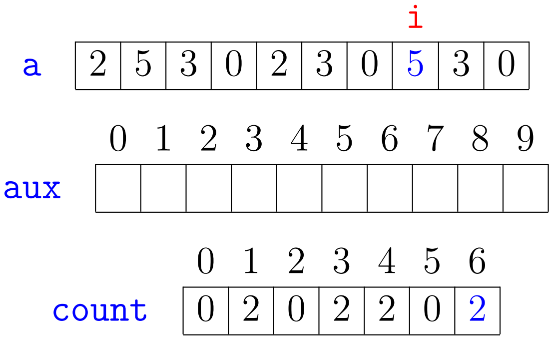
							i			
a	2	5	3	0	2	3	0	5	3	0

	0	1	2	3	4	5	6	7	8	9
aux										

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

Passo 1: calcula frequências

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



								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	2	0	2	2	0	2

Passo 1: calcula frequências

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

									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	2	0	2	3	0	2

Passo 1: calcula frequências

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

										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	0	2	3	0	2

Passo 2: transforma frequências em índices

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

	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 $\{0, \ldots, 5\}$.

	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 $\{0, \ldots, 5\}$.

	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 $\{0, \ldots, 5\}$.

	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 $\{0, \ldots, 5\}$.

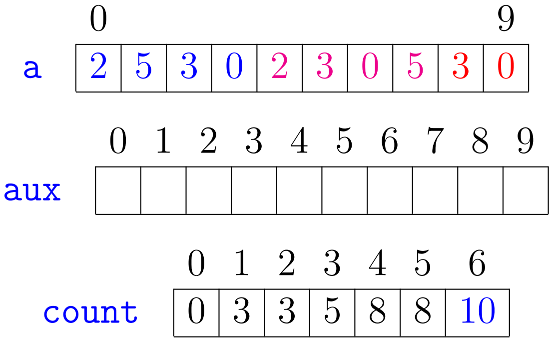
	0									9
`a`	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



	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 $\{0, \ldots, 5\}$.

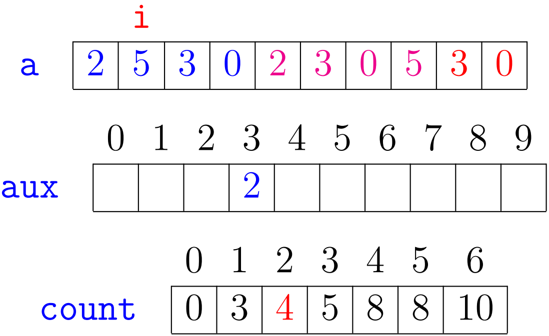
	i									
a	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



		i								
a	2	5	3	0	2	3	0	5	3	0

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

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

Passo 3: distribuição das chaves

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

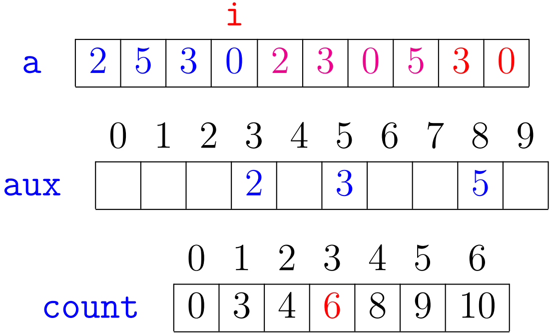
			i							
a	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



i

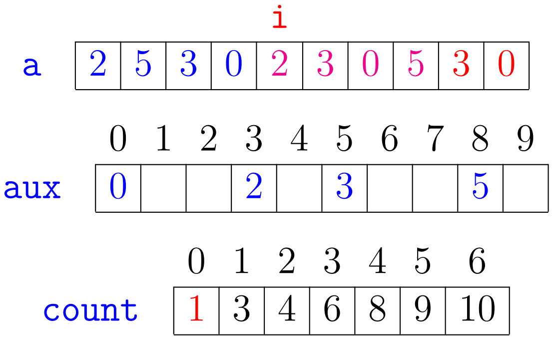
a										
	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



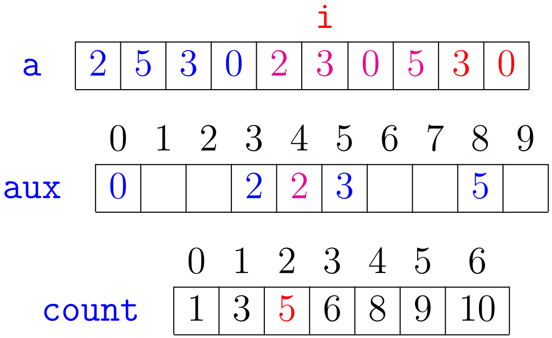
					i					
a	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



						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 $\{0, \ldots, 5\}$.

							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 $\{0, \ldots, 5\}$.

								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 $\{0, \ldots, 5\}$.

									`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 $\{0, \ldots, 5\}$.

										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 $\{0, \ldots, 5\}$.

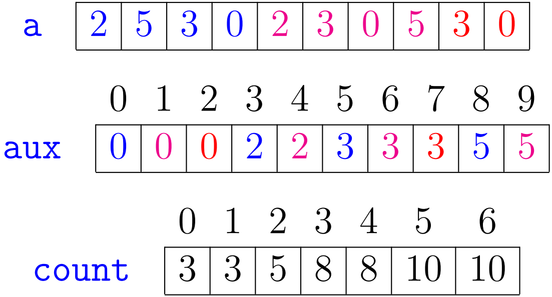
a	2	5	3	0	2	3	0	5	3	0

	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 $\{0, \ldots, 5\}$.



`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`

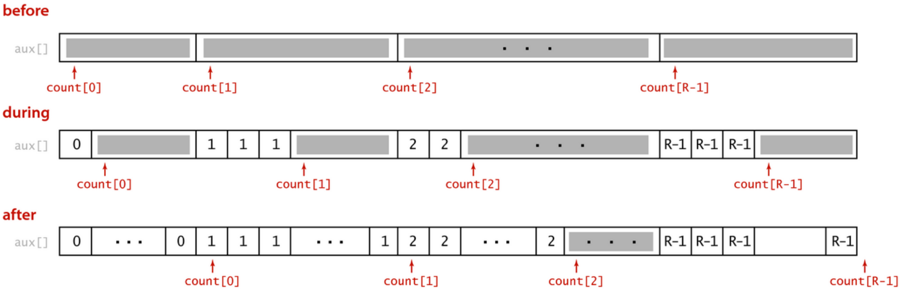
Cada `a`$[$`i`$]$ está em $\{0,\ldots,5\}$.

`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



Key-indexed counting (distribution phase)

Fonte: algs4

Ordenação por contagem

```
int n = a.length;
int[] count = new int[R+1];
for (int i = 0; i < n; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
// fase de distribuição
for (int i = 0; i < n; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < n; i++)
    a[i] = aux[i];
```

Obs: não são feitas comparações entre chaves.

Consumo de tempo

linha	consumo na linha
$1-2$	$\Theta(\mathtt{n})$
$3-4$	$\Theta(\mathtt{R})$
$5-6$	$\Theta(\mathtt{n})$
$7-9$	$\Theta(\mathtt{n})$

Consumo total: $\Theta(\mathtt{n} + \mathtt{R})$

Conclusões

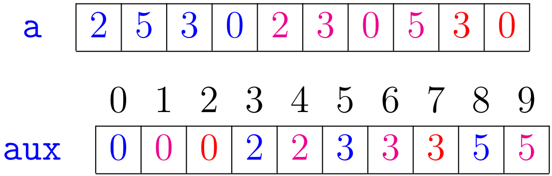
O consumo de tempo da ordenação por contagem é $\Theta(\mathtt{n} + \mathtt{R})$.

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

Estabilidade

A propósito: ordenação por contagem é **estável**:

na saída, chaves com mesmo valor estão na mesma ordem que apareciam na entrada.



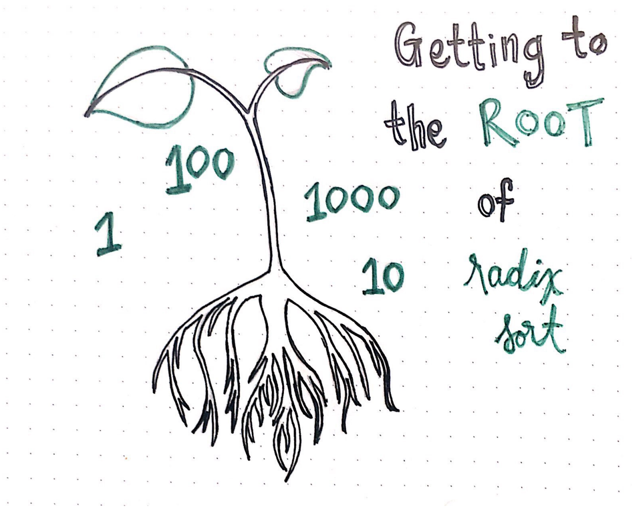
a	2	5	3	0	2	3	0	5	3	0

	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 usando “`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.

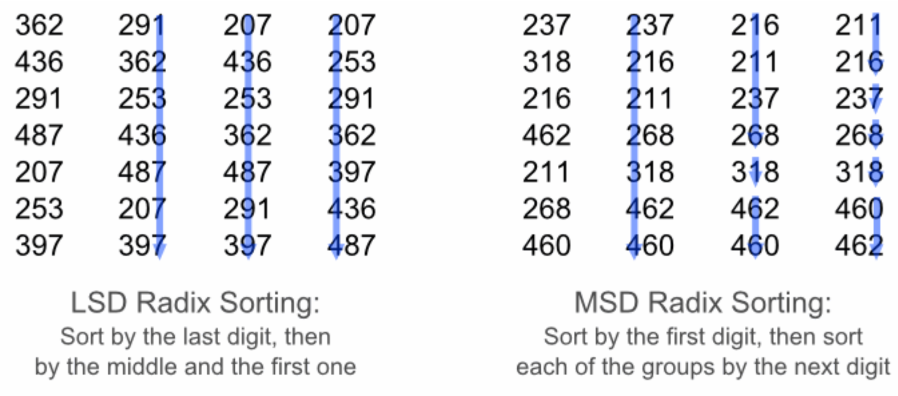
Se as chaves são strings os símbolos podem ser seus caracteres.

LSD e MSD

A ordenação digital aparece frequentemente em dois sabores:

- Least significant digit (LSD): trabalha examinado 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 examinado 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
436	362	436	253
291	253	253	291
487	436	362	362
207	487	487	397
253	207	291	436
397	397	397	487

LSD Radix Sorting:
Sort by the last digit, then
by the middle and the first one

237	237	216	211
318	216	211	216
216	211	237	237
462	268	268	268
211	318	318	318
268	462	462	460
460	460	460	462

MSD Radix Sorting:
Sort by the first digit, then sort
each of the groups by the next digit

Fonte: Radiz 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 $\mathtt{a}[\mathtt{j}]$ têm $\mathtt{d}$ dígitos decimais:

$\mathtt{a}[\mathtt{j}] = \mathtt{a}_\mathtt{d}\,10^{\mathtt{d}-1} + \cdots + \mathtt{a}_2\,10^1 + \mathtt{a}_1\,10^0$

Exemplo com $\mathtt{d} = 3$: $\quad 3 \cdot 10^2 + 2 \cdot 10 + 9$

LSD candidato

input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	1OHV845
1ICK750	1OHV845
1OHV845	1OHV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
1OHV845	3ATW723
1OHV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
3ATW723	4PGC938

keys are all the same length

Typical candidate for LSD string sort

Fonte: `algs4`

LSD

```
public class LSD {
  // extended ASCII
  private static final int R= 256;
  public static void sort(String[] a,
            int W){
     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: $\Theta(\mathtt{Wn})$
- dígitos em $0\,..\,\mathtt{R}{-}1$: $\Theta(\mathtt{W}\,(\mathtt{n}+\mathtt{R}))$.

Exemplo com $\mathtt{d} = 5$ e $\mathtt{R} = 128$:

$$\mathtt{a}[4]128^4 + \mathtt{a}[3]128^3 + \mathtt{a}[2]128^2 + \mathtt{a}[1]128 + \mathtt{a}[0]$$

sendo $0 \leq \mathtt{a}[\mathtt{i}] \leq 127$

Conclusão

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

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

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

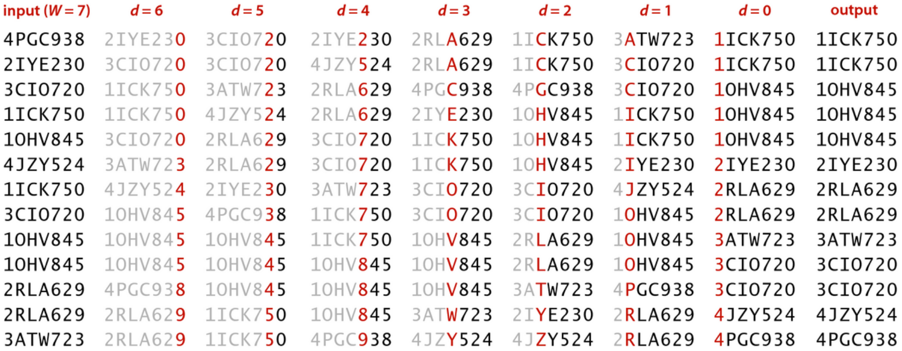
Use ordenação por contagem com $\mathtt{R} = 2^{\mathtt{r}} - 1$.

Cada passada do ordenação por contagem:

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

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

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	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	2RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Fonte: algs4

LSD com baralho

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

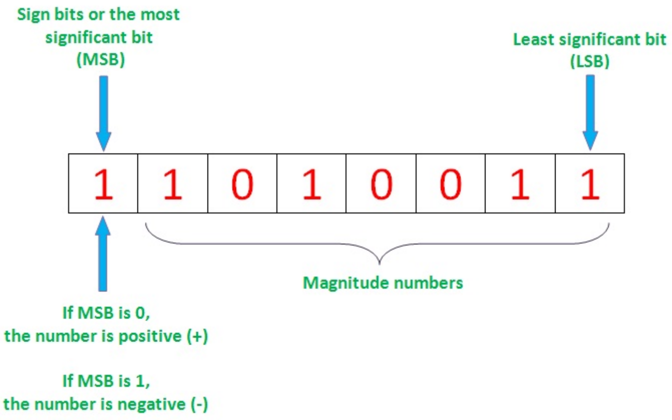
♦A
♥A
♣A
♠A
♠2
♣2
♥2
♦2
♥3
♠3
♣3
♦3
♦4
♣4
♥4
♠4
♠5
♦5
♣5
♥5
♥6
♣6
♠6
♦6
♥7
♣7
♠7
♦7
♦8
♥8
♠8
♣8
♦9
♥9
♠9

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

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 $\mathtt{n} + \mathtt{R}$.

Most-Significant-Digit



Fonte: Complement Number System

MSD ideia


sort on first character value to partition into subarrays

recursively sort subarrays (excluding first character)

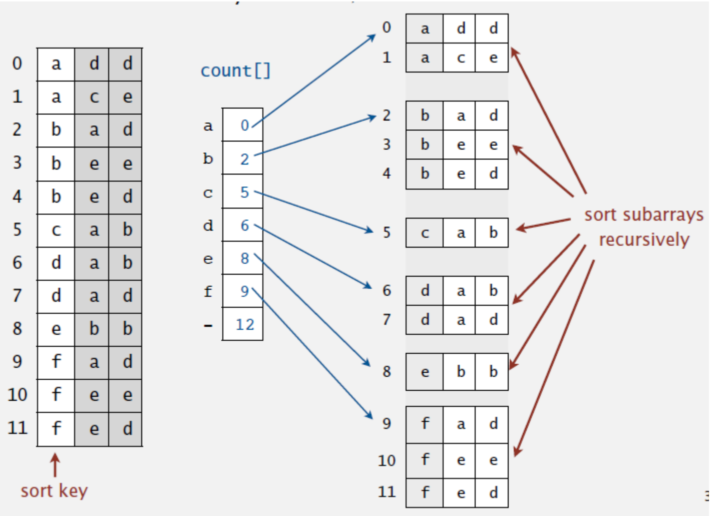
MSD candidato

input	**sorted result**
she	are
sells	by
seashells	seashells
by	seashells
the	seashore
seashore	sells
the	sells
shells	she
she	she
sells	shells
are	surely
surely	the
seashells	the

various key lengths

Fonte: algs4

MSD recursão



0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

count[]

a	0
b	2
c	5
d	6
e	8
f	9
-	12

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

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

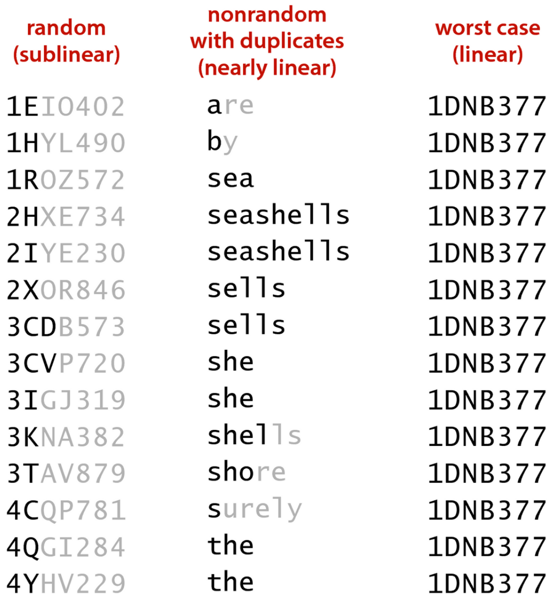
```
private static int charAt(String s,
          int d) {
    if (d == s.length()) return -1;
    return s.charAt(d);
}

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);
} } }
```

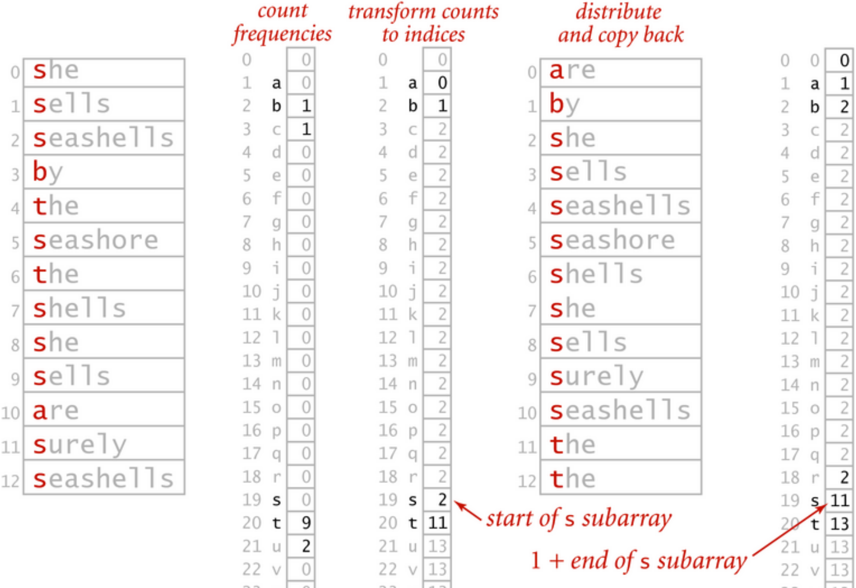
MSD caracteres examinados



random (sublinear)	nonrandom with duplicates (nearly linear)	worst case (linear)
1EIO402	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

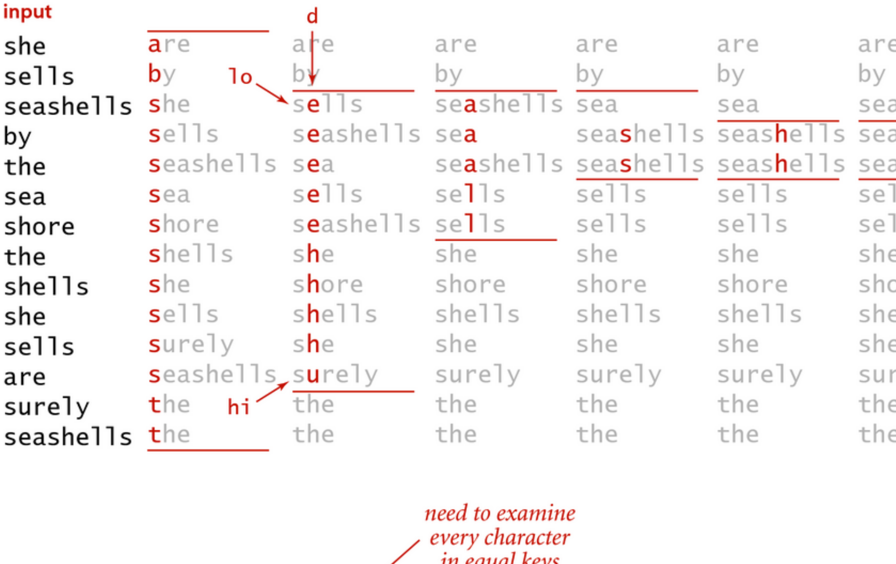
		distribute and copy back
0	she	are
1	sells	by
2	seashells	she
3	by	sells
4	the	seashells
5	seashore	seashore
6	the	shells
7	shells	she
8	she	sells
9	sells	surely
10	are	seashells
11	surely	the
12	seashells	the

		count frequencies	transform counts to indices	
0		0	0	0
1	a	0	0	1
2	b	1	1	2
3	c	1	2	2
4	d	0	2	2
5	e	0	2	2
6	f	0	2	2
7	g	0	2	2
8	h	0	2	2
9	i	0	2	2
10	j	0	2	2
11	k	0	2	2
12	l	0	2	2
13	m	0	2	2
14	n	0	2	2
15	o	0	2	2
16	p	0	2	2
17	q	0	2	2
18	r	0	2	2
19	s	0	2	11
20	t	9	11	13
21	u	2	13	13
22	v	0	13	13

start of s subarray

1 + end of s subarray

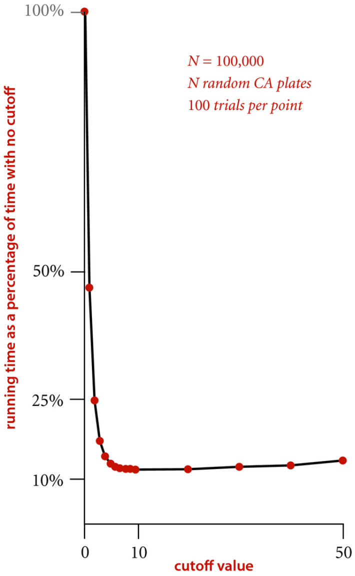
MSD em ação



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	sel
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



Effect of cutoff for small subarrays

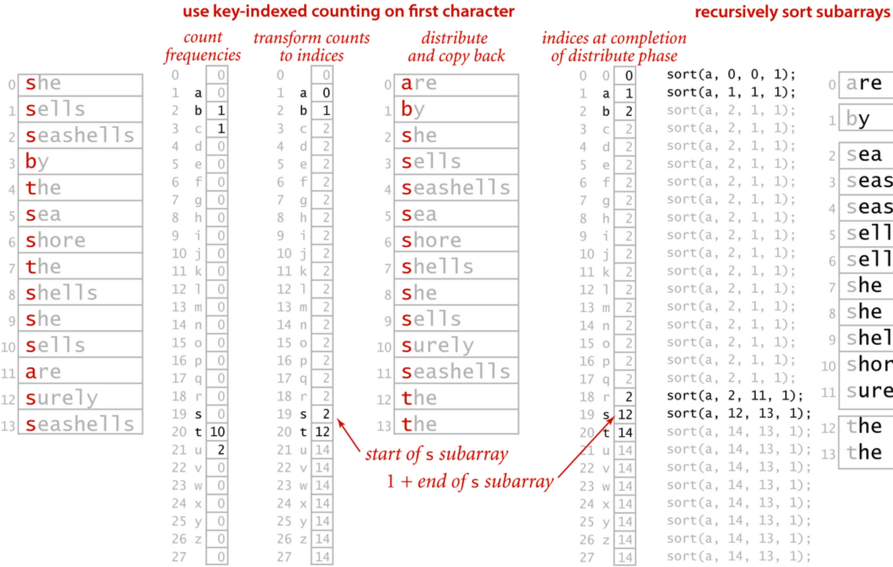
MSD com baralho

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

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

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

MSD



use key-indexed counting on first character

recursively sort subarrays

0	she
1	sells
2	seashells
3	by
4	the
5	sea
6	shore
7	the
8	shells
9	she
10	sells
11	are
12	surely
13	seashells

count frequencies

0		0
1	a	0
2	b	1
3	c	1
4	d	0
5	e	0
6	f	0
7	g	0
8	h	0
9	i	0
10	j	0
11	k	0
12	l	0
13	m	0
14	n	0
15	o	0
16	p	0
17	q	0
18	r	0
19	s	0
20	t	10
21	u	2
22	v	0
23	w	0
24	x	0
25	y	0
26	z	0
27		0

transform counts to indices

0		0
1	a	0
2	b	1
3	c	2
4	d	2
5	e	2
6	f	2
7	g	2
8	h	2
9	i	2
10	j	2
11	k	2
12	l	2
13	m	2
14	n	2
15	o	2
16	p	2
17	q	2
18	r	2
19	s	2
20	t	12
21	u	14
22	v	14
23	w	14
24	x	14
25	y	14
26	z	14
27		14

distribute and copy back

0	are
1	by
2	she
3	sells
4	seashells
5	sea
6	shore
7	shells
8	she
9	sells
10	surely
11	seashells
12	the
13	the

start of s subarray

1 + end of s subarray

indices at completion of distribute phase

0	0	0
1	a	1
2	b	2
3	c	2
4	d	2
5	e	2
6	f	2
7	g	2
8	h	2
9	i	2
10	j	2
11	k	2
12	l	2
13	m	2
14	n	2
15	o	2
16	p	2
17	q	2
18	r	2
19	s	12
20	t	14
21	u	14
22	v	14
23	w	14
24	x	14
25	y	14
26	z	14
27		14

```
sort(a, 0, 0, 1);
sort(a, 1, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 1, 1);
sort(a, 2, 11, 1);
sort(a, 12, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
sort(a, 14, 13, 1);
```

0	are
1	by
2	sea
3	seas
4	seas
5	sell
6	sell
7	she
8	she
9	shel
10	shor
11	sure
12	the
13	the

`MSD` características

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

`MSD` versus quicksort para strings

Desvantagens de usar quicksort para strings:

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

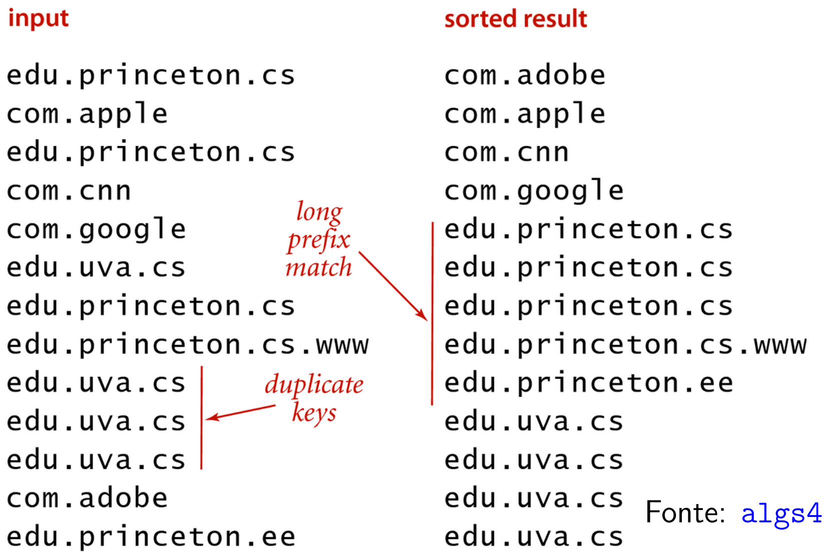
3-way string quicksort: ideia


use first character value to partition into "less," "equal," and "greater" subarrays

recursively sort subarrays (excluding first character for "equal' subarray)

Fonte: algs4

3-way string quicksort: candidato



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

long prefix match

duplicate keys

Fonte: `algs4`

Typical 3-way string quicksort candidate

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);
  }
```

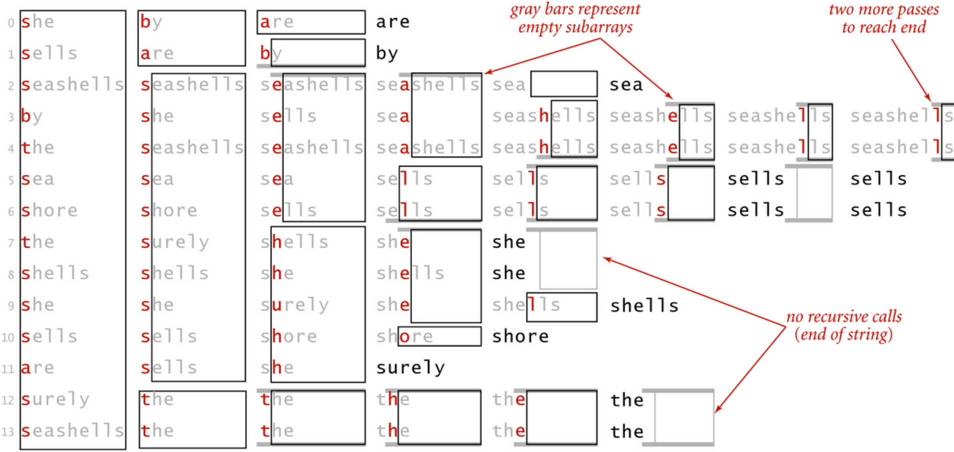
Quick3string

```
// 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



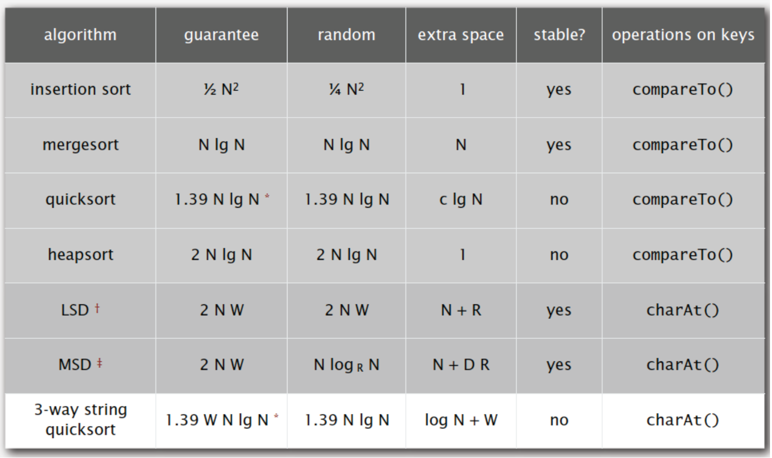
Trace of recursive calls for 3-way string quicksort (no cutoff for small subarrays)

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 $2\mathtt{n} \ln \mathtt{n}$ comparações entre chaves: caro para chaves com prefixos comuns longos.

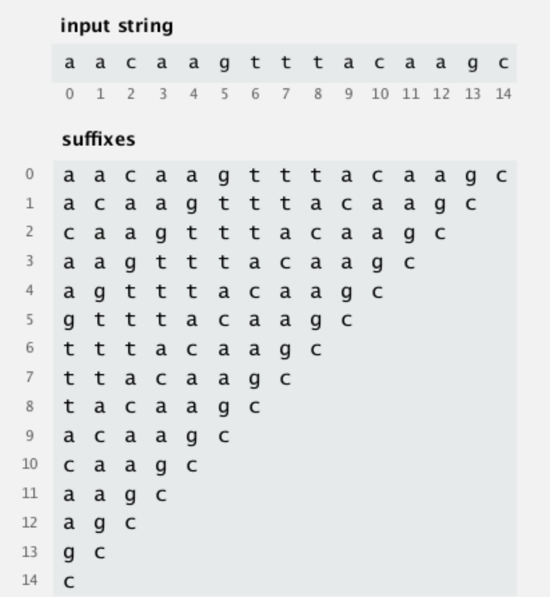
Resumo



algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ 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

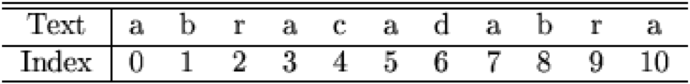
form suffixes

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 toget

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

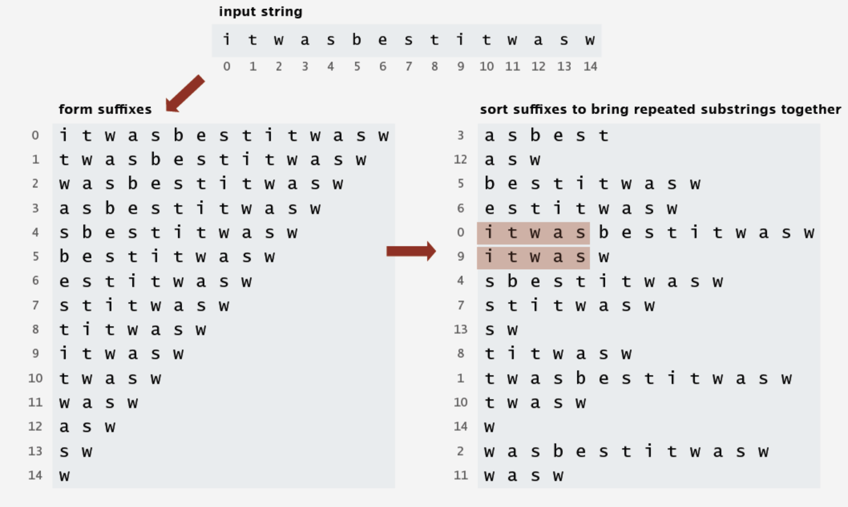
Fonte: `Brief Introduction to Suffix Array`

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

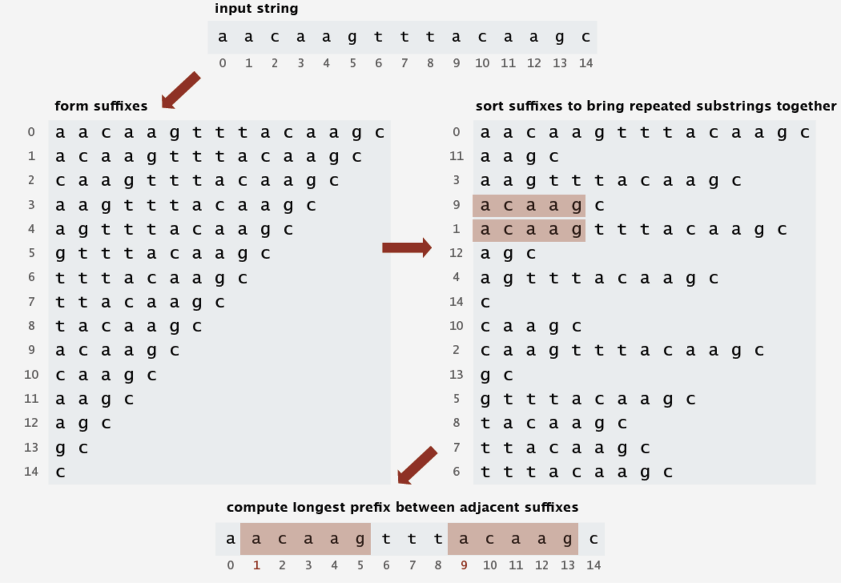
0	itwasbestitwasw
1	twasbestitwasw
2	wasbestitwasw
3	asbestitwasw
4	sbestitwasw
5	bestitwasw
6	estitwasw
7	stitwasw
8	titwasw
9	itwasw
10	twasw
11	wasw
12	asw
13	sw
14	w

sort suffixes to bring repeated substrings together

3	asbest
12	asw
5	bestitwasw
6	estitwasw
0	itwasbestitwasw
9	itwasw
4	sbestitwasw
7	stitwasw
13	sw
8	titwasw
1	twasbestitwasw
10	twasw
14	w
2	wasbestitwasw
11	wasw

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	aacaagtttacaagc
1	acaagtttacaagc
2	caagtttacaagc
3	aagtttacaagc
4	agtttacaagc
5	gtttacaagc
6	tttacaagc
7	ttacaagc
8	tacaagc
9	acaagc
10	caagc
11	aagc
12	agc
13	gc
14	c

sort suffixes to bring repeated substrings together

0	aacaagtttacaagc
11	aagc
3	aagtttacaagc
9	acaagc
1	acaagtttacaagc
12	agc
4	agtttacaagc
14	c
10	caagc
2	caagtttacaagc
13	gc
5	gtttacaagc
8	tacaagc
7	ttacaagc
6	tttacaagc

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

Fonte: algs4

Manber e Meyers

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

Algoritmo é iterativo:

Cada iteração começa com o vetor dos sufixos ordenados de acordo com os $2^{\mathtt{d}}$ 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`.

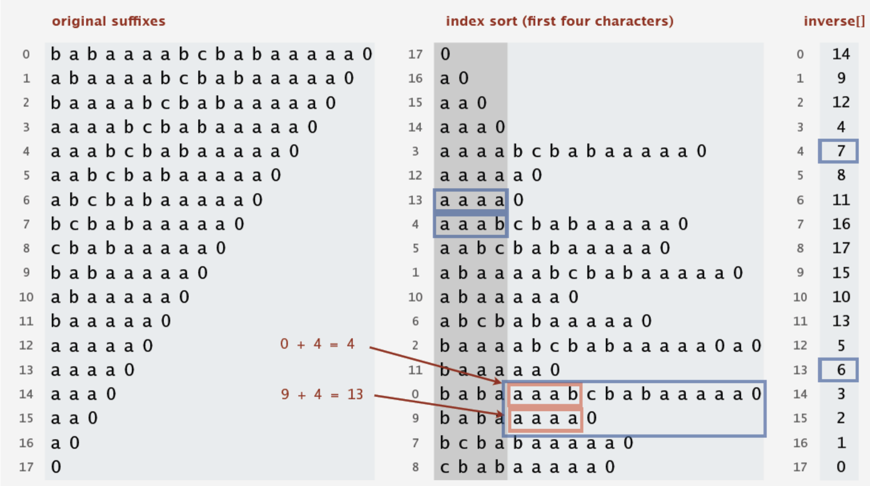
Cada iteração consiste em construir o vetor dos sufixos ordenados de acordo com os $2^{\mathtt{d}+1}$ primeiros caracteres.

Manber e Meyers

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

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

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 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 $\text{suffixes}_8[9] \leq \text{suffixes}_8[0]$