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`
- ...

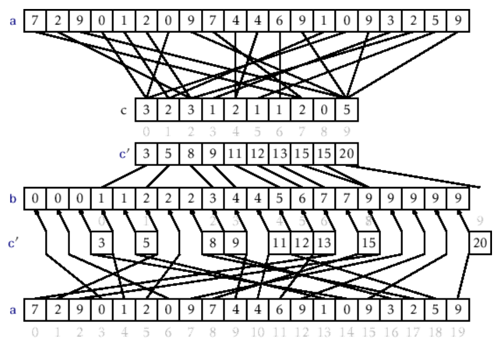
BFS aplicações

- 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 `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 $a[0\,.\,.\,n-1]$ e ordena seus elementos.

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

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 $\{0, \ldots, R-1\}$.

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

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

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

			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	1	0	1

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	1	0	1	1	0	1

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	1	0	2	1	0	1

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	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 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	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

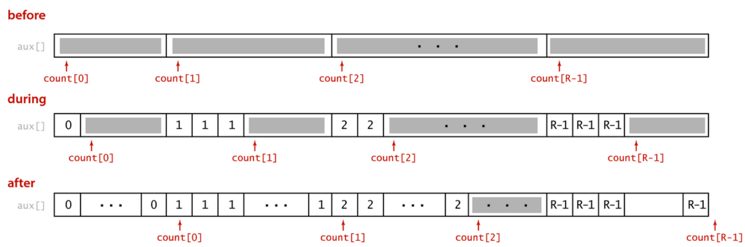
Cada a[i] está em $\{0,\ldots,5\}$.

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];
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**:

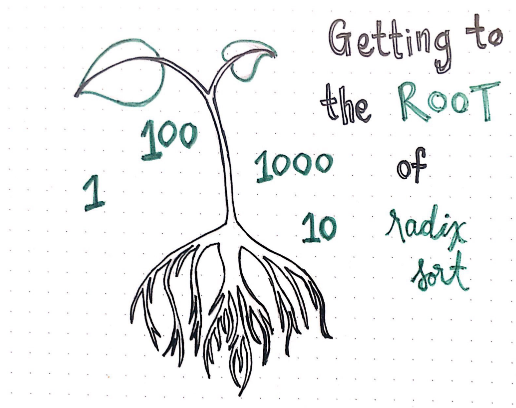
na saída, chaves com mesmo valor estão na mesma ordem que apareciam na entrada.

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 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.

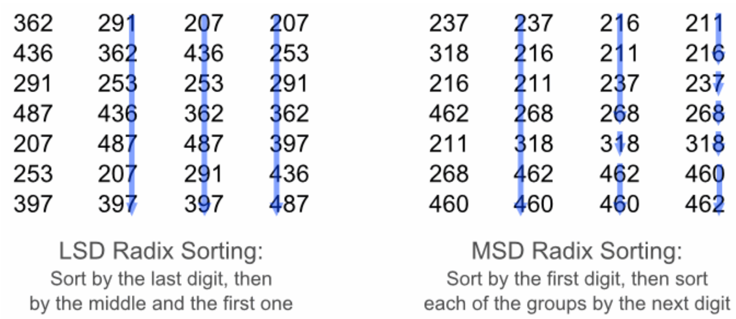
Se as chaves são strings os símbolos podem ser seus caracteres.

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



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 $a[j]$ têm d dígitos decimais:

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

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

LSD candidato

input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	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 $0 \le \mathtt{a}[\mathtt{i}] \le 127$

Conclusão

Dados $\mathtt{n}$ números com $\mathtt{b}$ bits e um inteiro $\mathtt{r} \le \mathtt{b}$, 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

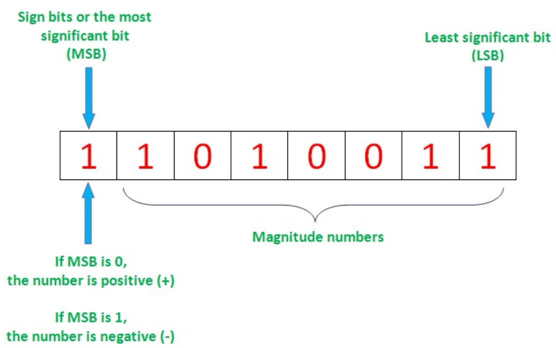
LSD com baralho

♣J	♦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

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 $\mathtt{d}$ das strings deve ser estável.
- Faz cerca de $\mathtt{Wn}$ chamadas de `charAt()`.
- Utiliza espaço extra proporcional a $\mathtt{n}+\mathtt{R}$.

Most-Significant-Digit



Fonte: Complement Number System

MSD ideia

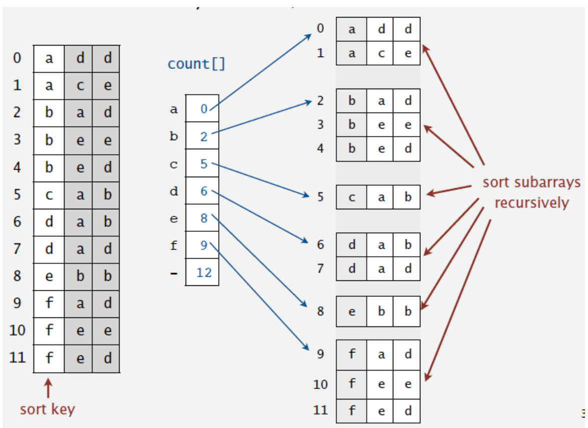
MSD candidato

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



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

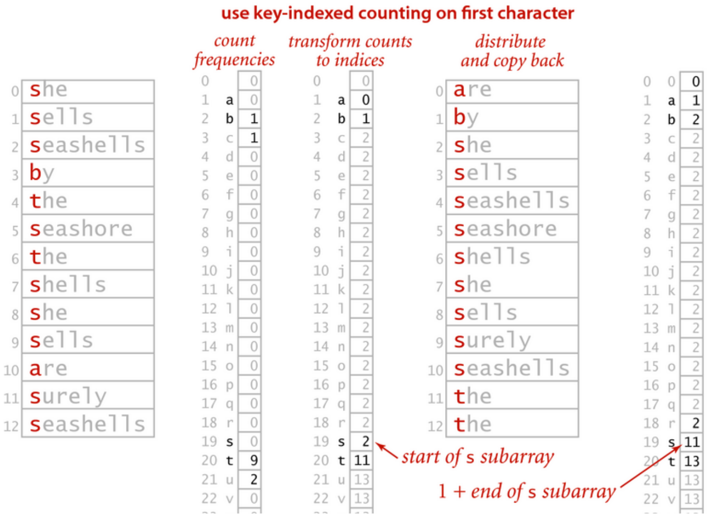
```
for (int i = lo; i<= hi; i++) {
    int c = charAt(a[i], d);
    count[c+2]++; }
for (int r = 0; r < R+1; r++)
    count[r+1] += count[r];
for (int i = lo; i <= hi; i++) {
    int c = charAt(a[i], d);
    aux[count[c+1]++] = a[i]; }
for (int i = lo; i <= hi; i++)
    a[i] = aux[i - lo];
for (int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

MSD caracteres examinados

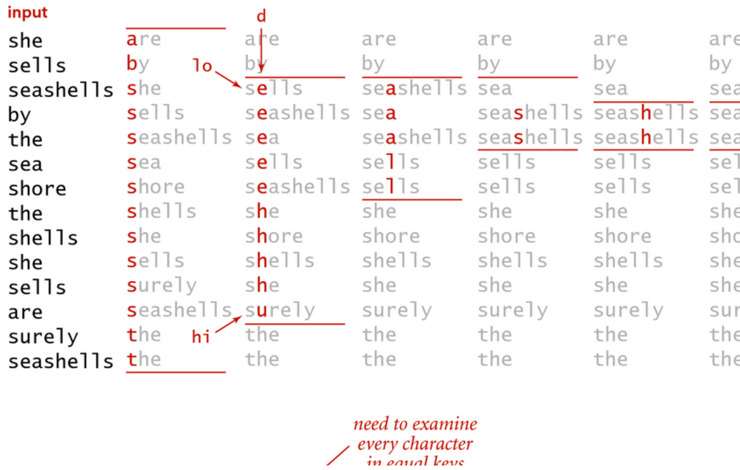
random (sublinear)	nonrandom with duplicates (nearly linear)	worst case (linear)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Fonte: algs4

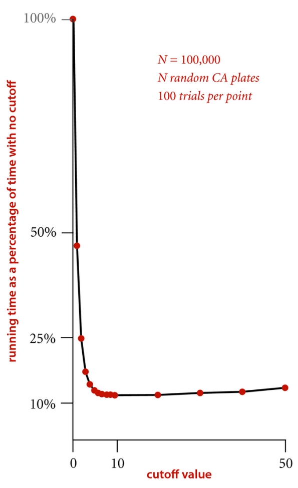
MSD



MSD em ação



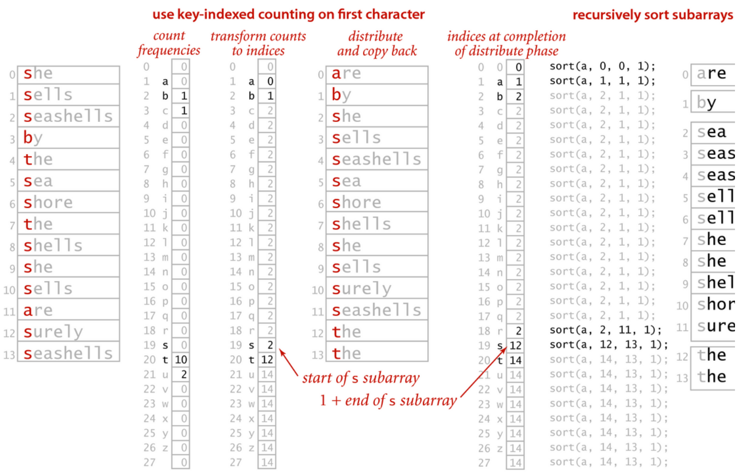
MSD



Effect of cutoff for small subarrays

MSD com baralho

MSD



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:

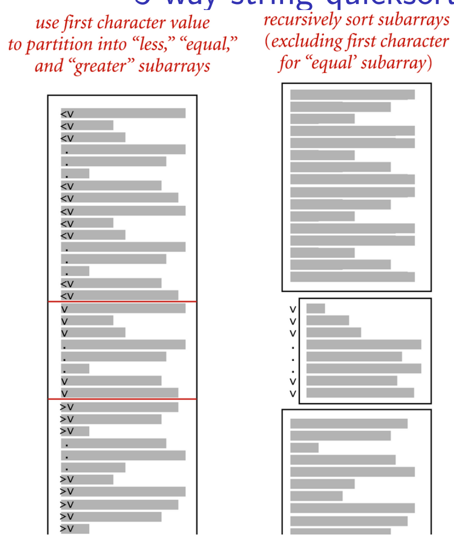
- Espaço extra para `aux[]` devido a ordenação por contagem.
- Espaço extra para `count[]` devido a ordenação por contagem.
- Laço interno com muitas instruções devido a ordenação por contagem.
- acesso aleatório da memória faz com que seja *cache ineficient*.

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



Fonte: algs4

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
```

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);
  }
```

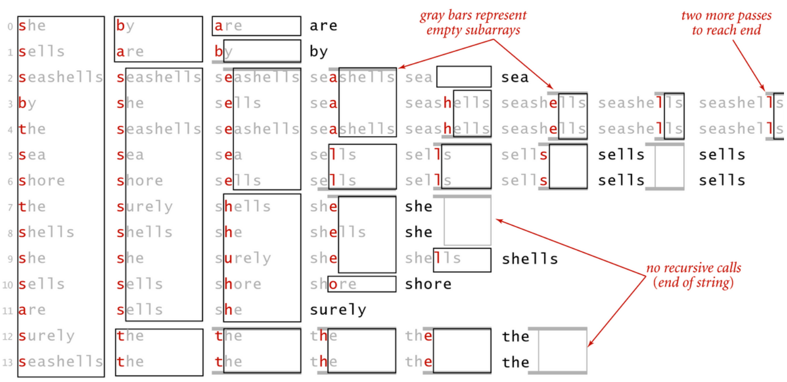
Quick3string

```
// 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

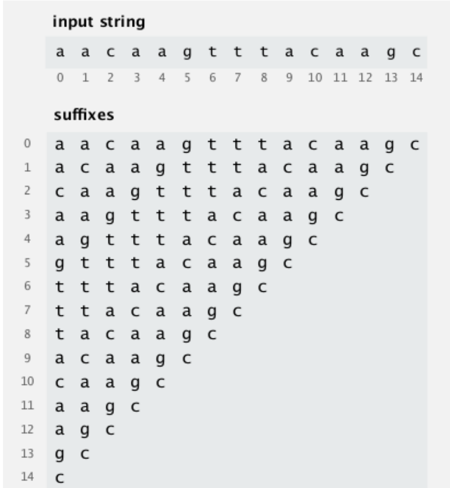
- Faz *3-way partition* segundo o d-ésimo caractere.
- Menos pesada que a R-way partition do MSD.
- Não reexamina os caracteres iguais ao caractere pivô; mas reexamina os caracteres diferentes do pivô.
- quicksort padrão faz na média aproximadamente $2n \ln n$ comparações entre chaves: caro para chaves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

Resumo

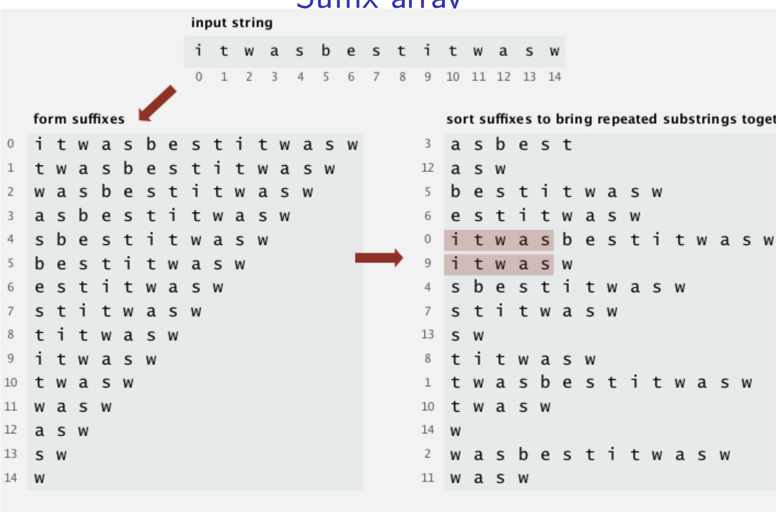
algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	N lg N	N lg N	N	yes	compareTo()
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	compareTo()
heapsort	2 N lg N	2 N lg N	1	no	compareTo()
LSD †	2 N W	2 N W	N + 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



Suffix array



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

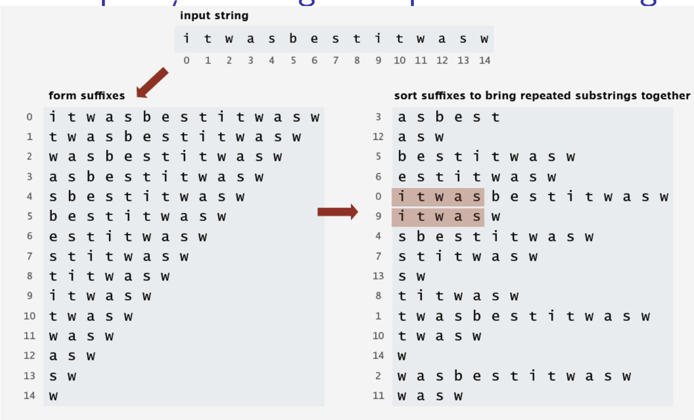
Fonte: Brief Introduction to Suffix Array

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

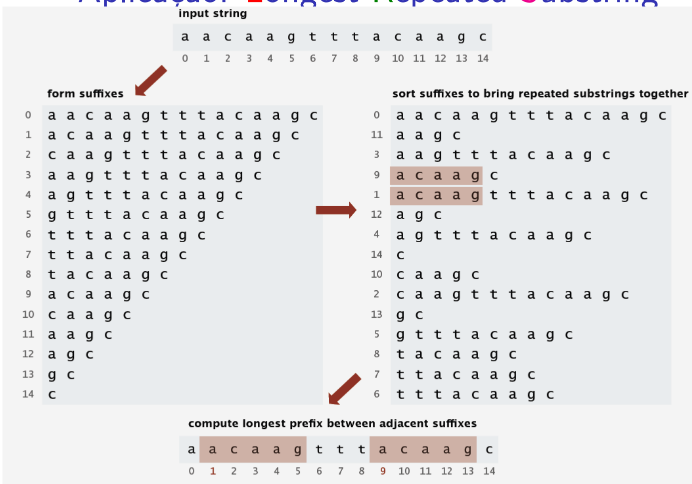
Fonte: Brief Introduction to Suffix Array

Aplicação: Longest Repeated Substring



Fonte: algs4

Aplicação: Longest Repeated Substring



Fonte: algs4

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