MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

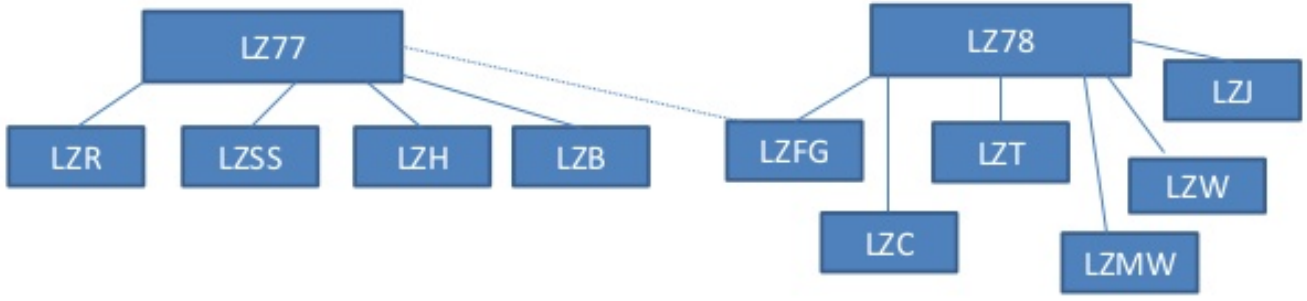
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 26

Algoritmo LZW

Limpel Ziv Algorithm Family



APPLICATIONS:
- ZIP
- GZIP
- STACKER

APPLICATIONS:
- GIF
- V.42
- COMPRESS

Fig 1: Limpel Ziv Algorithm Family

Referências: Data Compression (SW), slides (SW), LZW compression, video (SW)

Esqueleto da biblioteca LZW

Rotinas em LZW.c

```
/* tamanho do alfabeto */
static int R = 256;

/* número de bits dos códigos */
static int W = 12;

/* number códigos 2^W */
static int L = 4096;

void compress() {...}
void expand() {...}
```

LZW compress()

input = string de entrada

```
crie dicionário com símbolos do alfabeto
repita
   encontre o maior prefixo s de input
             que está no dicionário
   transmita para saída o índice de s
   ponha s+c no dicionário onde c é
         o símbolo que segue s em input
         (unmatched symbol).
   apague do input o prefixo s
até que input é vazio
```

LZW exemplo

Decisões de projeto:

- Alfabeto $= \{a, b\} \Rightarrow R = 2$
- $W = 3$ bits $\Rightarrow$ índices entre 0 e 7
- tamanho L do dicionário é $2^W = 8$
- índice R codificará EOF (end of file)

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação											

dicionário

string	código
a	0
b	1
EOF	2

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0										

dicionário

string	código
a	0
b	1
EOF	2
ab	3

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1									

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3								

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5						

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4			

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4		1	

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

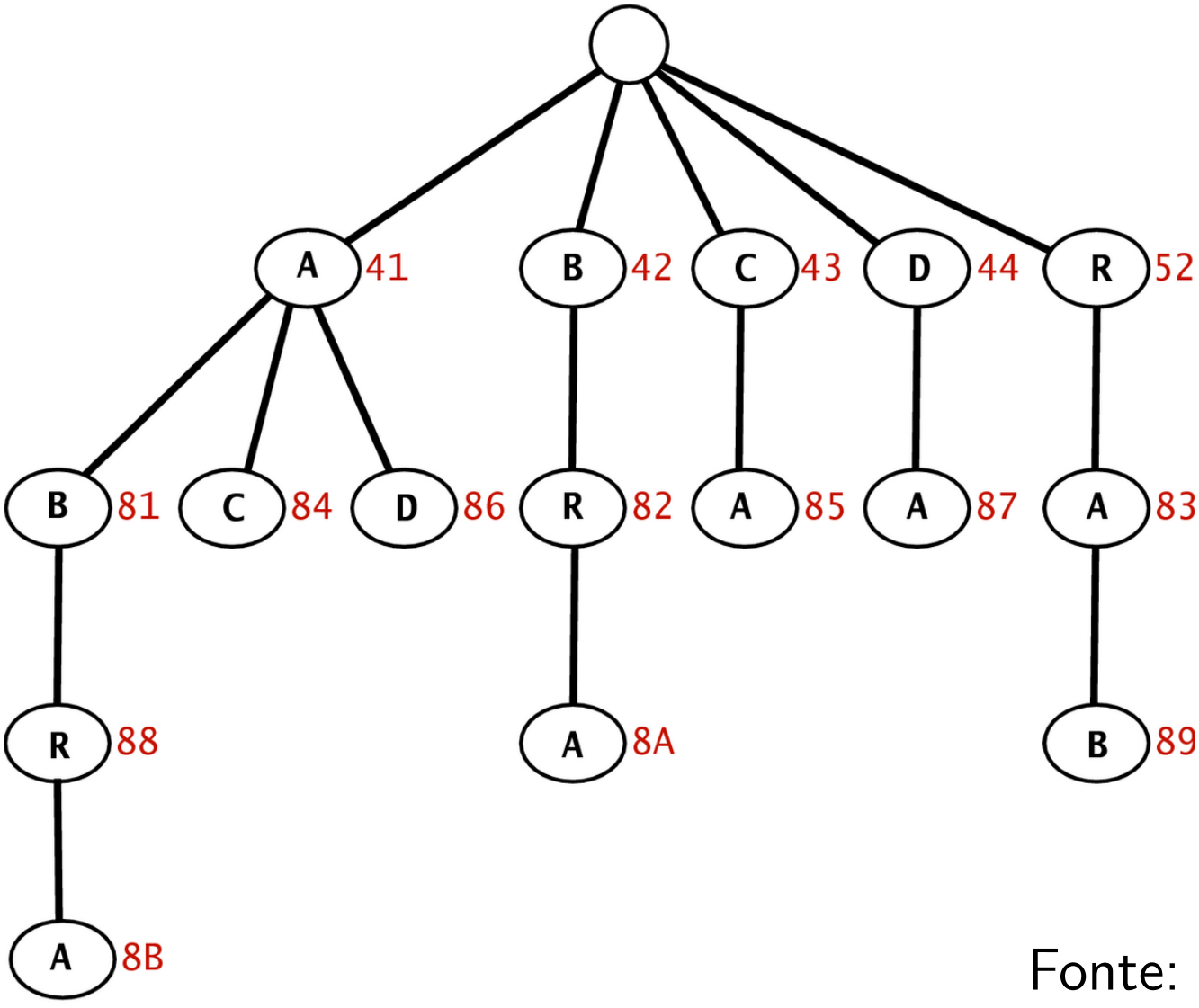
LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4		1	2

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

Dicionário (Trie) de LZW



Fonte: algs4

Trie representation of LZW code table

compress()

Estatística dos dados não precisa ser passada por `compress()` para `expand()`.

string(`i`): devolve string de comprimento 1 com o caracter de código `i`

```
void compress() {
   char *input = readString();
   TST st = TSTInit();     /* trie ternária */
   for (int i = 0; i < R; i++)
      put(st, string(i), i);

   /* R é o código para EOF */
   int code = R+1;
```

compress()

```
  while (strlen(input) > 0) {
     char *s = longestPrefixOf(st, input);
     write(get(st, s), W);
     int t = strlen(s), n = strlen(input);
     if (t < n && code < L) {
        s = substring(input, 0, t+1);
        put(st, s, code++);
     }
     input = substring(input, t, n);
  }
  write(R, W);                          /* EOF */
  close();
}
```

AULA 27

LZW `expand()`

`expand()`

- simula `compress()` para reconstruir o dicionário.
- anda *um pouquinho atrás* de `compress()`, mas olhando para frente (*lookahead*).

LZW expand(): exemplo

msg codificada

0	1	3	5	4	1	2

output

dicionário

código	string

LZW expand(): exemplo

msg codificada

0	1	3	5	4	1	2

output

dicionário

código	string
0	a
1	b
2	EOF

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a						

dicionário

código	string
0	a
1	b
2	EOF
3	a?

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a	b					

dicionário

código	string
0	a
1	b
2	EOF
3	ab

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a	b					

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	b?

LZW expand(): exemplo

msg codificada	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba

LZW expand(): exemplo

msg código	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?

LZW expand(): exemplo

msg código	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?
	Xiii!

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	?			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?
	Xiii!

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	a			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
	:-)

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	a			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	aba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	aba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	ba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a	b	

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	ba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a	b	

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	bab

LZW expand()

```
crie dicionário com símbolos do alfabeto
decodifique o primero índice code para a
            string val
coloque val+? no dicionário
repita
   decodifique o primeiro símbolo c
            do próximo índice code
   complete com c a última entrada da tabela
   termine de decodificar code e obtenha
            a string s
   transmita s para a saída
   ponha s+? no dicionário
até que input é vazio
```

expand()

```
void expand() {
   char *st = mallocSafe(L*sizeof(char));
   int i; /* próximo código disponível */
   /* inicialize a ST */
   for (i = 0; i < R; i++)
      st[i] = string(i);
   st[i++] = string(EOF);
   int codeword = readInt(W);
   /* mensagem é vazia?  */
   if (codeword == R) return;
   char *val = st[codeword];
```

expand()

```
    while(true) {
        write(val);
        codeword = readInt(W);
        if (codeword == R) break;
        char *s = st[codeword];
        if (i == codeword)
            /* caso especial */
            s = concat(val, val[0]);
        if (i < L)
            st[i++] = concat(val, s[0]);
        val = s;
    }
    close();
}
```

LZW exemplos

virus (50000 bits)

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```

```
12536 bits

% java LZW - < genomeVirus.txt | java PictureDump 512 36
```

```
18232 bits
```
← *not as good as 2-bit code because repetitive data is rare*

bitmap (6144 bits)

```
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 bits

% java LZW - < q64x96.bin | java BinaryDump 0
2824 bits
```
← *not as good as run-length code because file size is too small*

entire text of *Tale of Two Cities* (5812552 bits)

```
% java BinaryDump 0 < tale.txt
5812552 bits

% java Huffman - < tale.txt | java BinaryDump 0
3043928 bits

% java LZW - < tale.txt | java BinaryDump 0
2667952 bits
```
← *compression ratio 2667952/5812552 = 46% (best yet)*

Compressing and expanding various files with LZW 12-bit encoding

Fonte: algs4

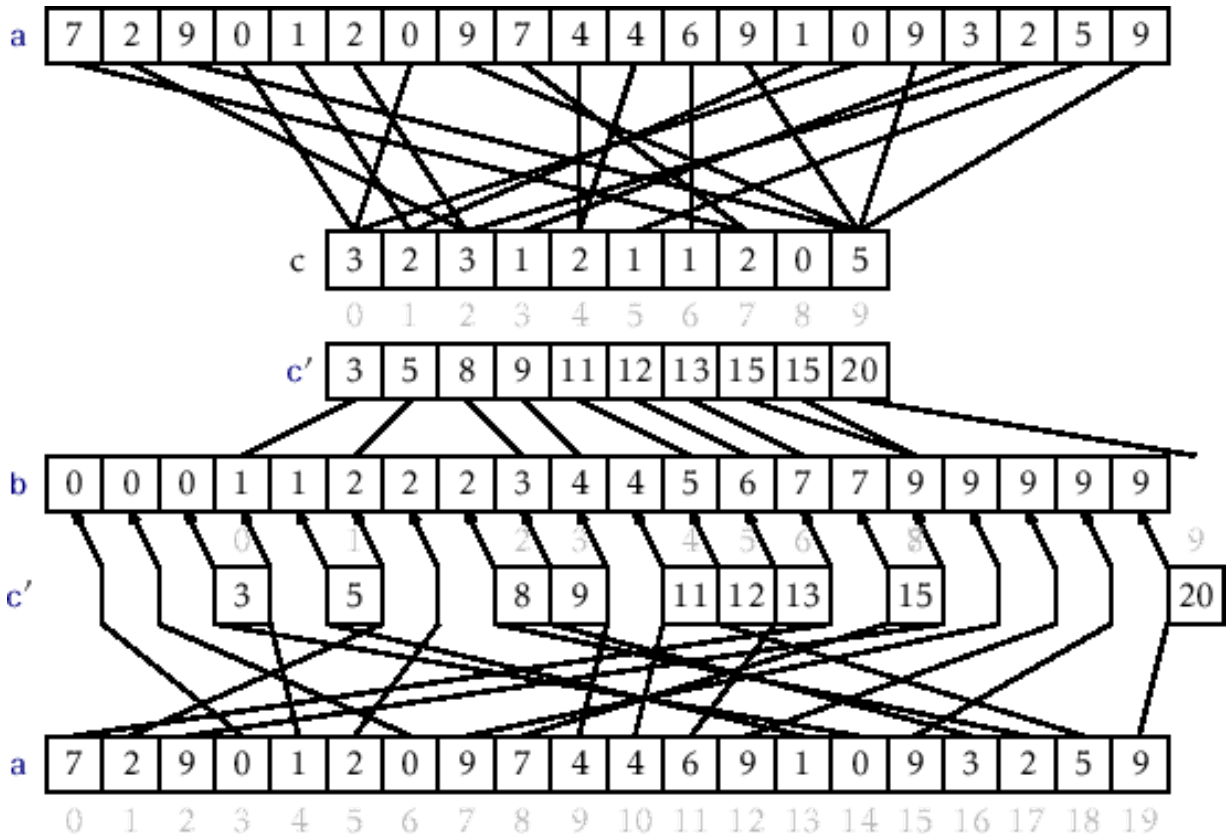
Huffman versus LZW

```
% java BinaryDump 0 < les-miserables.txt
```
26581192 bits

```
% java Huffman - < les-miserables.txt |
java BinaryDump 0
```
15211904 bits

```
% java LZW - < les-miserables.txt | java
BinaryDump 0
```
15316048 bits **demorou muito para responder!**

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

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

Entra:

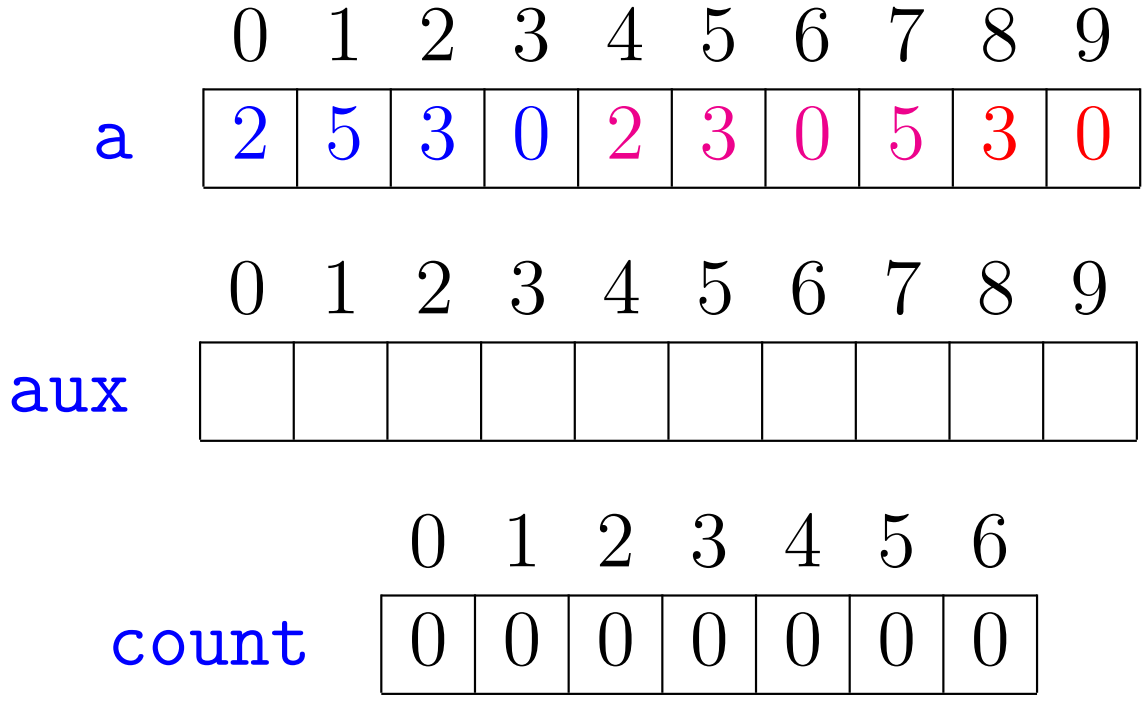
	0	1	2	3	4	5	6	7	8	9
a	2	5	3	0	2	3	0	5	3	0

Sai:

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

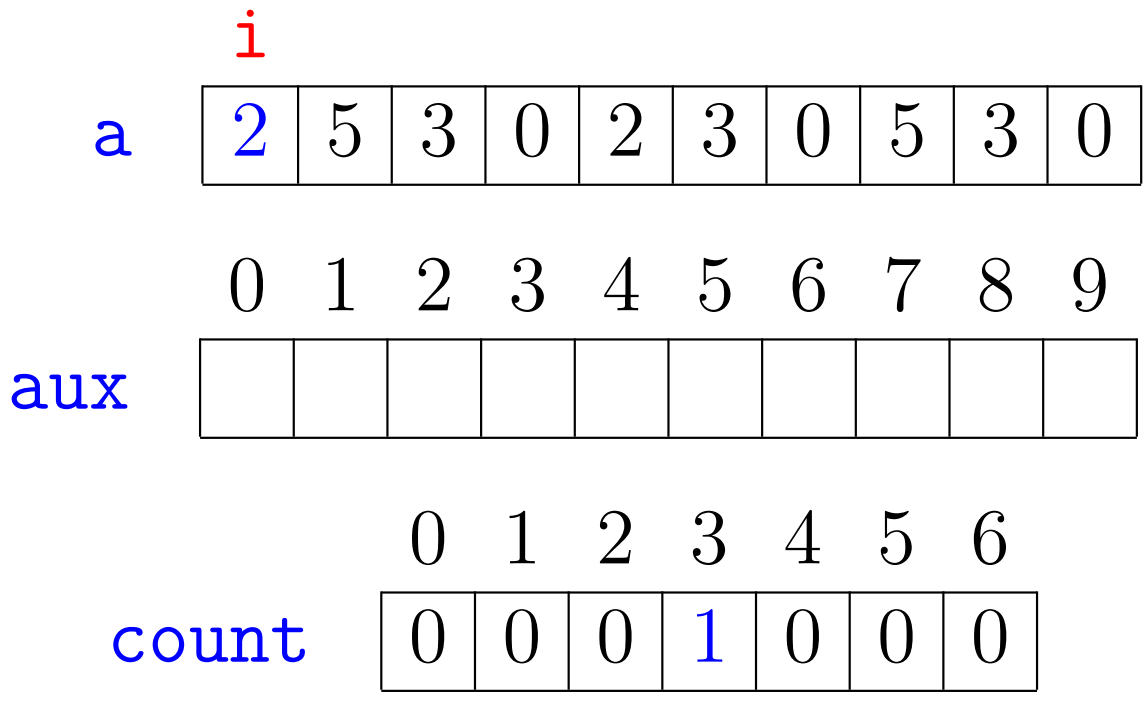
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



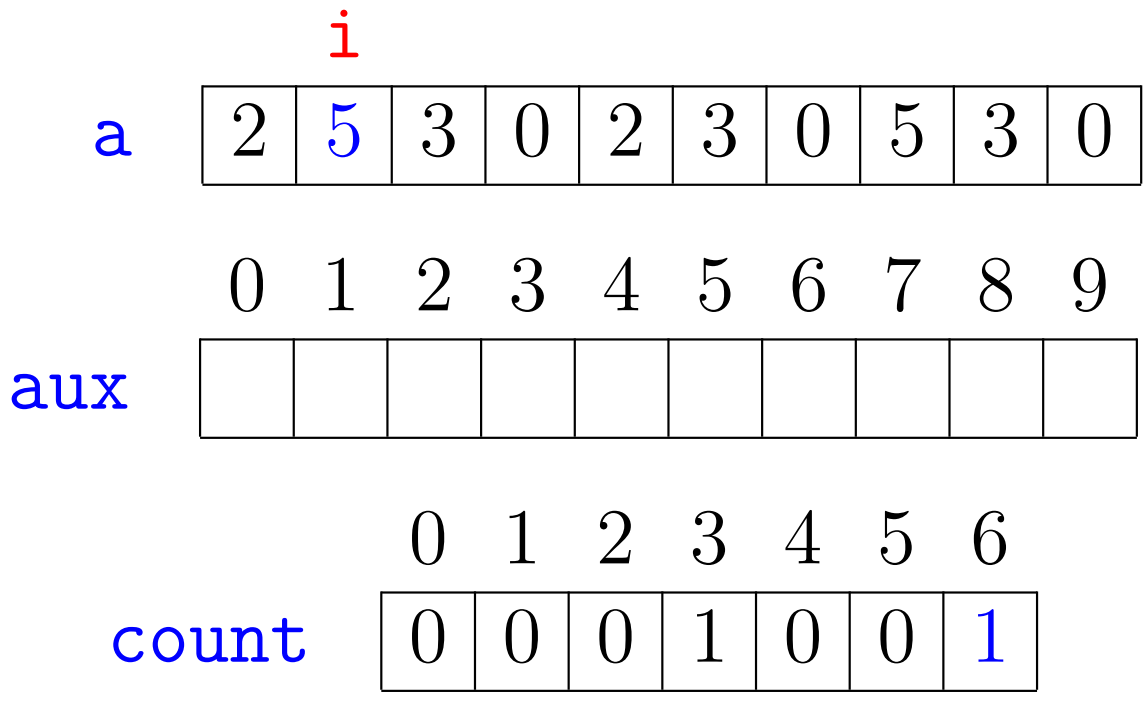
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



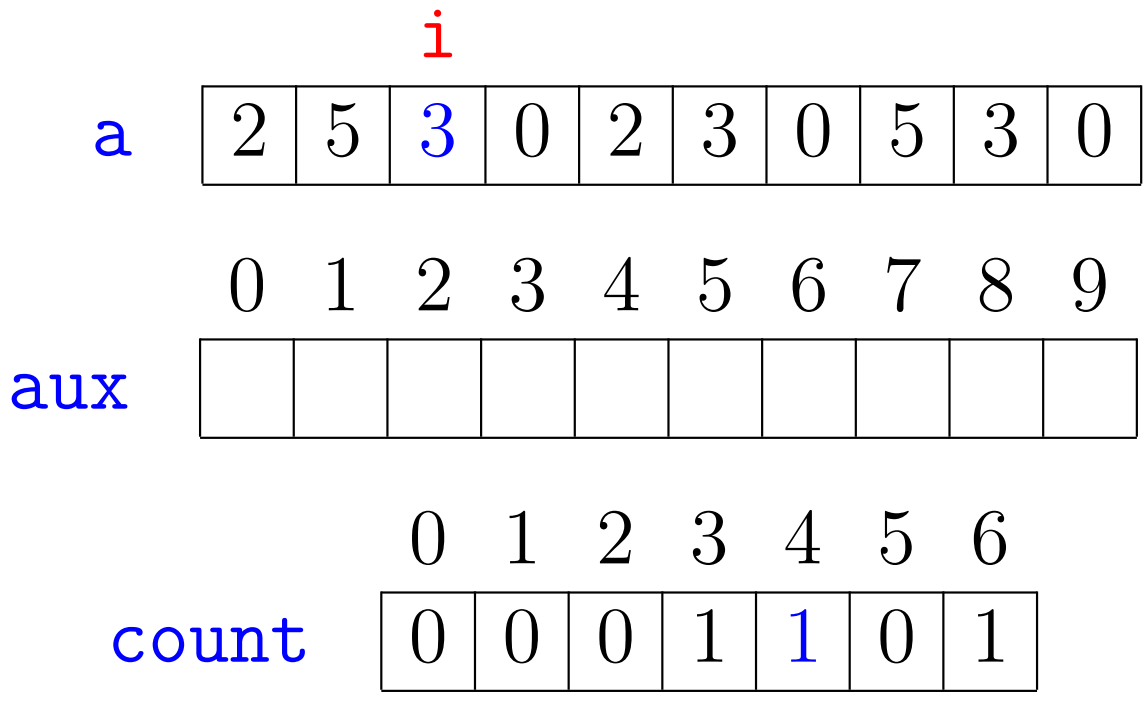
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



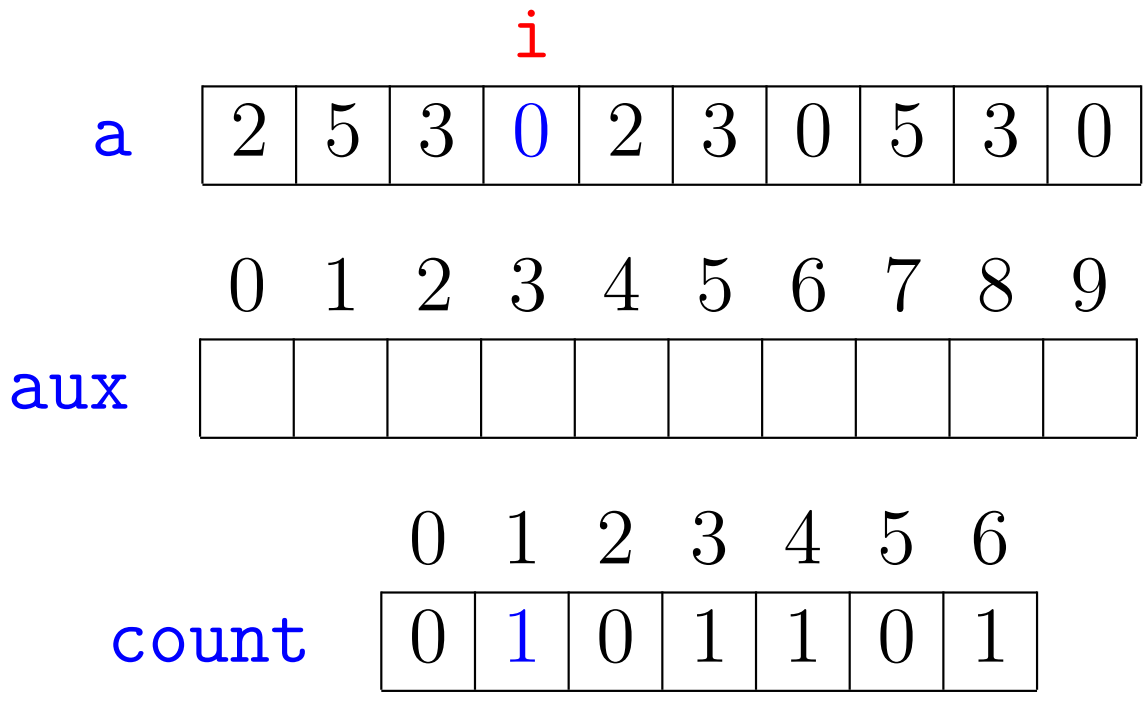
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



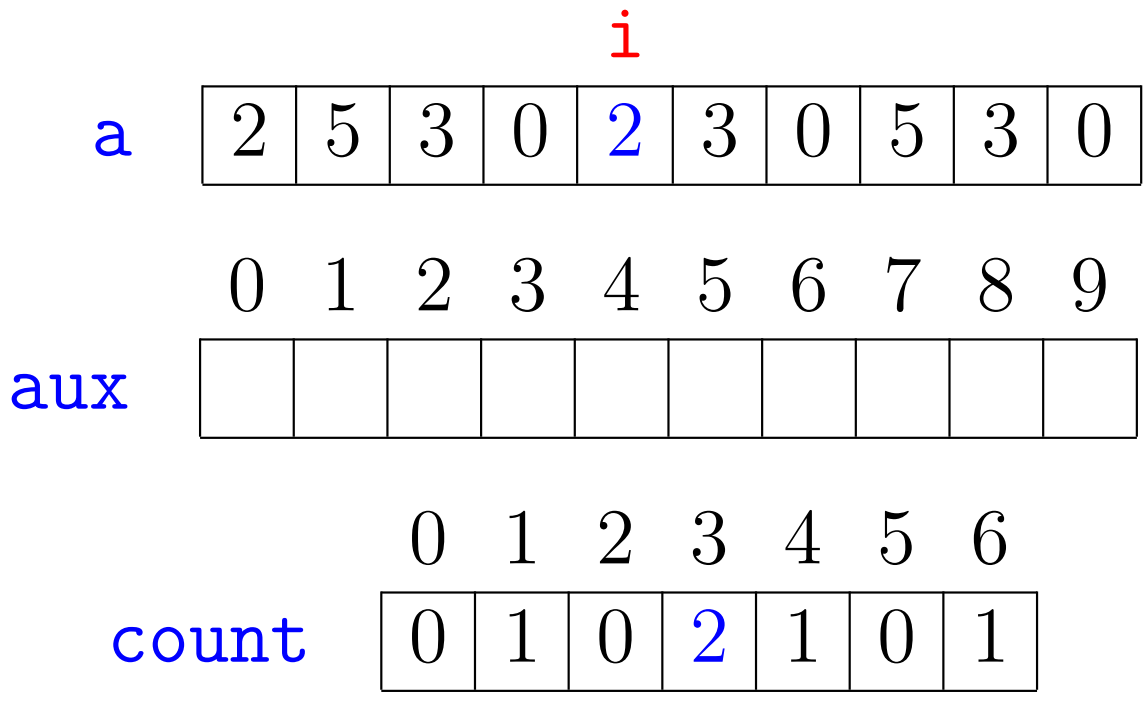
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



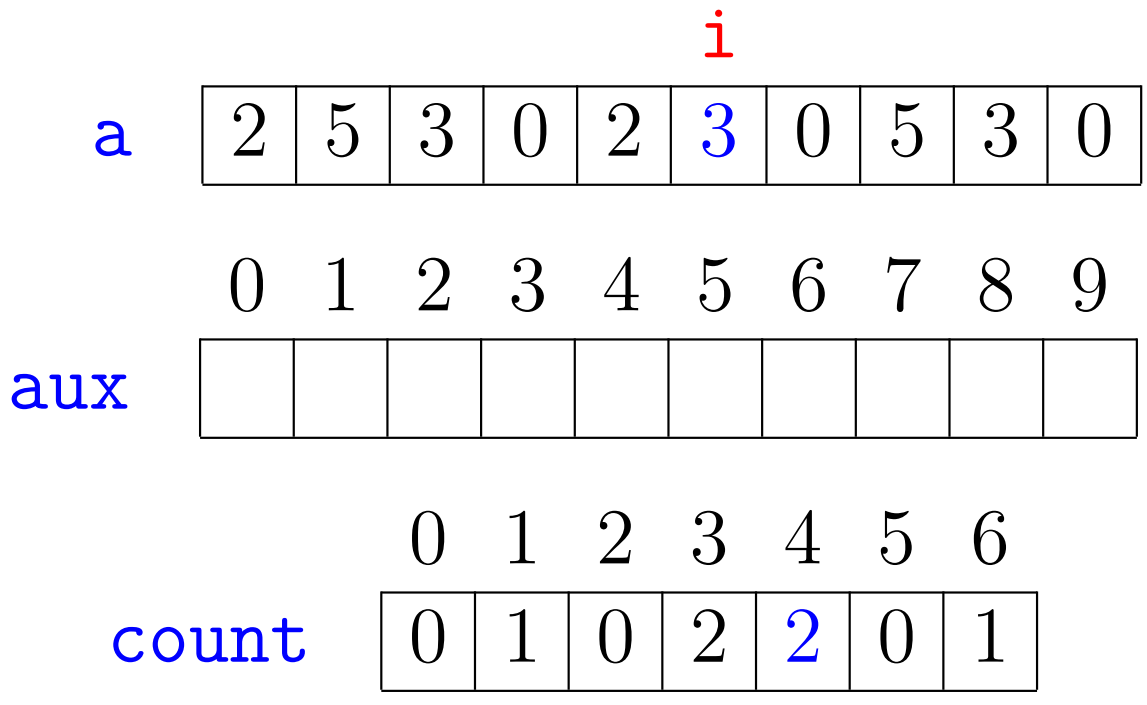
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



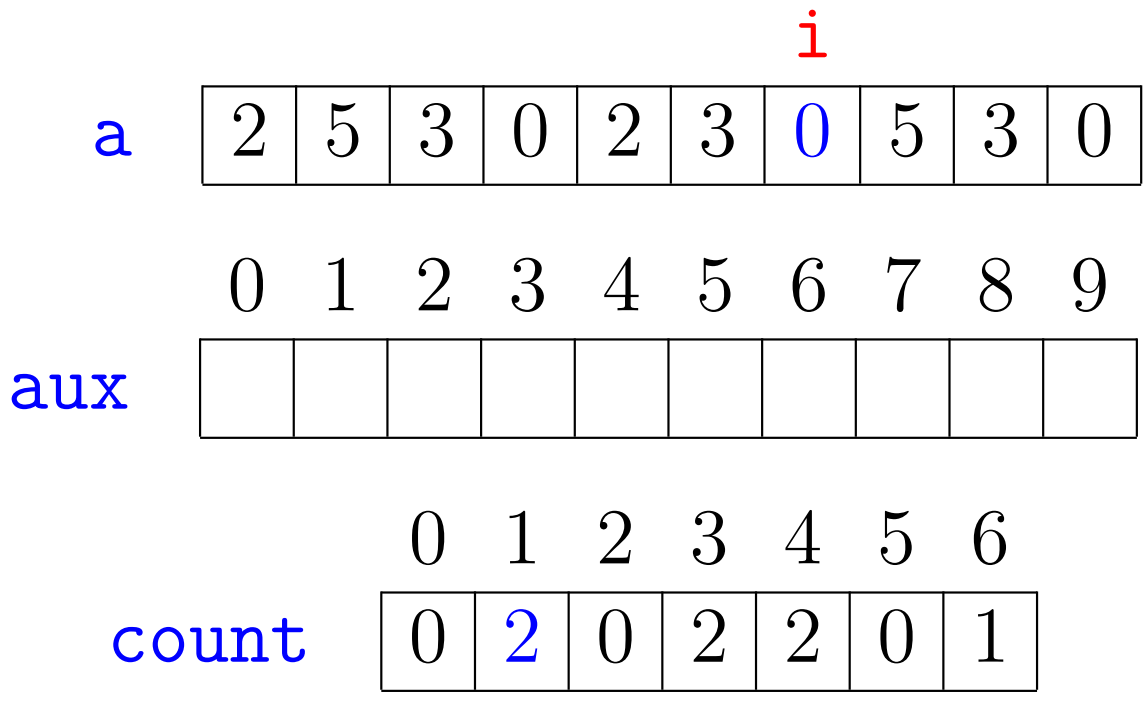
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



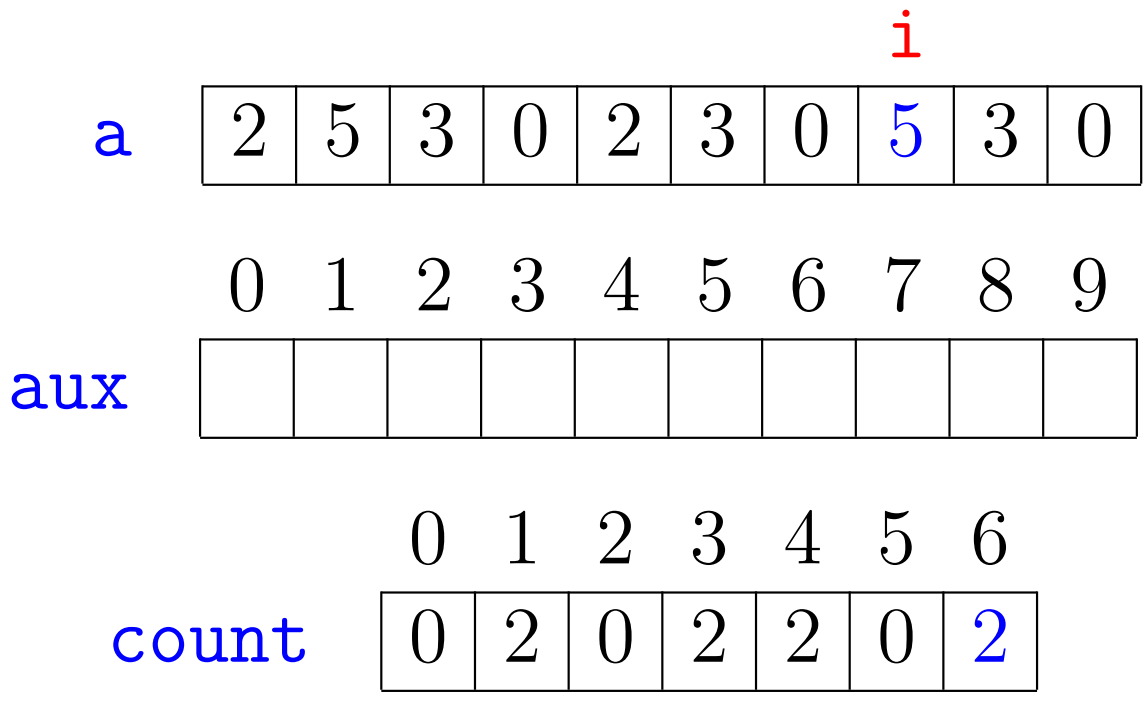
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



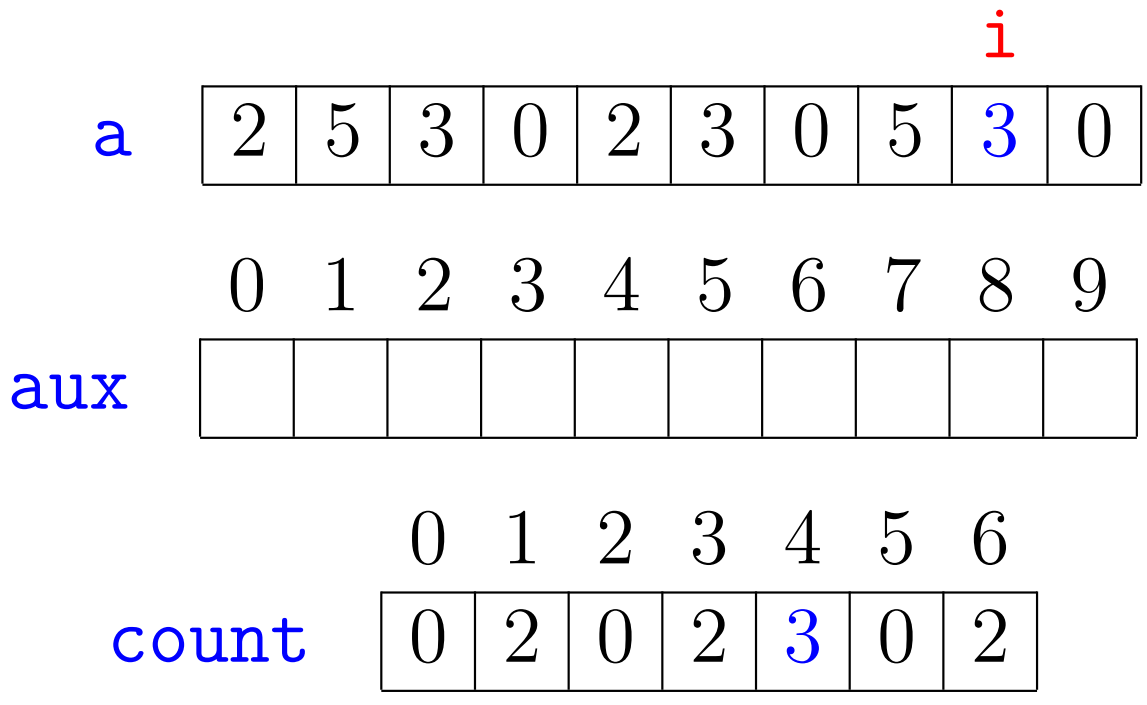
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



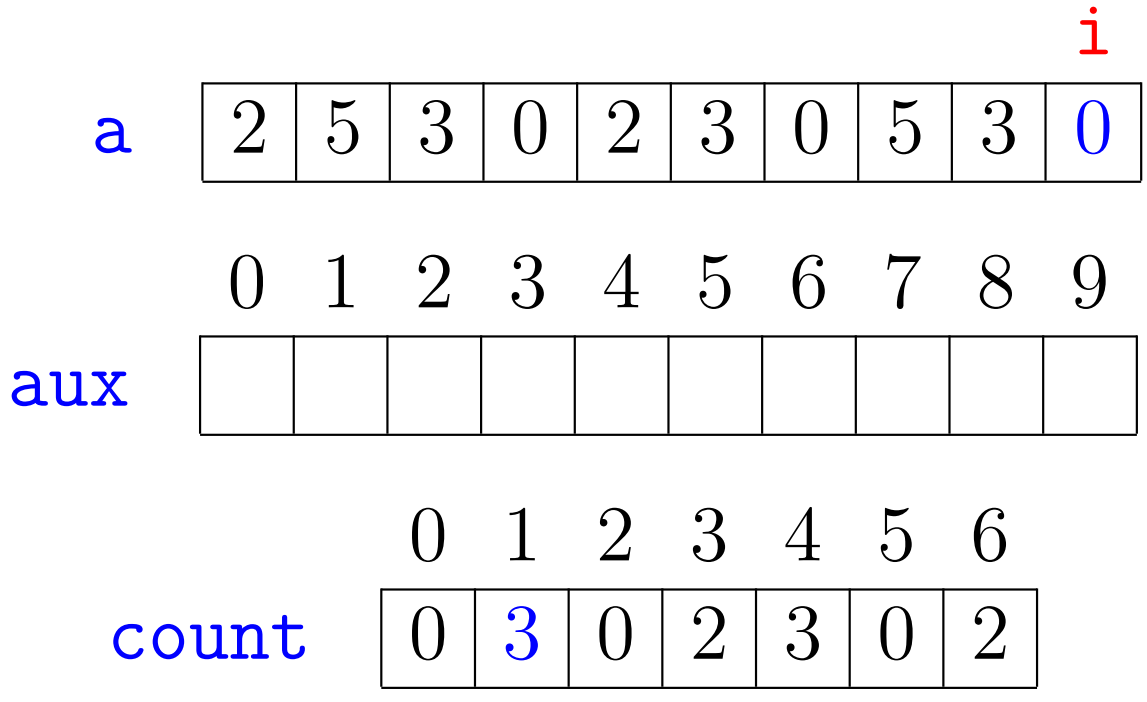
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



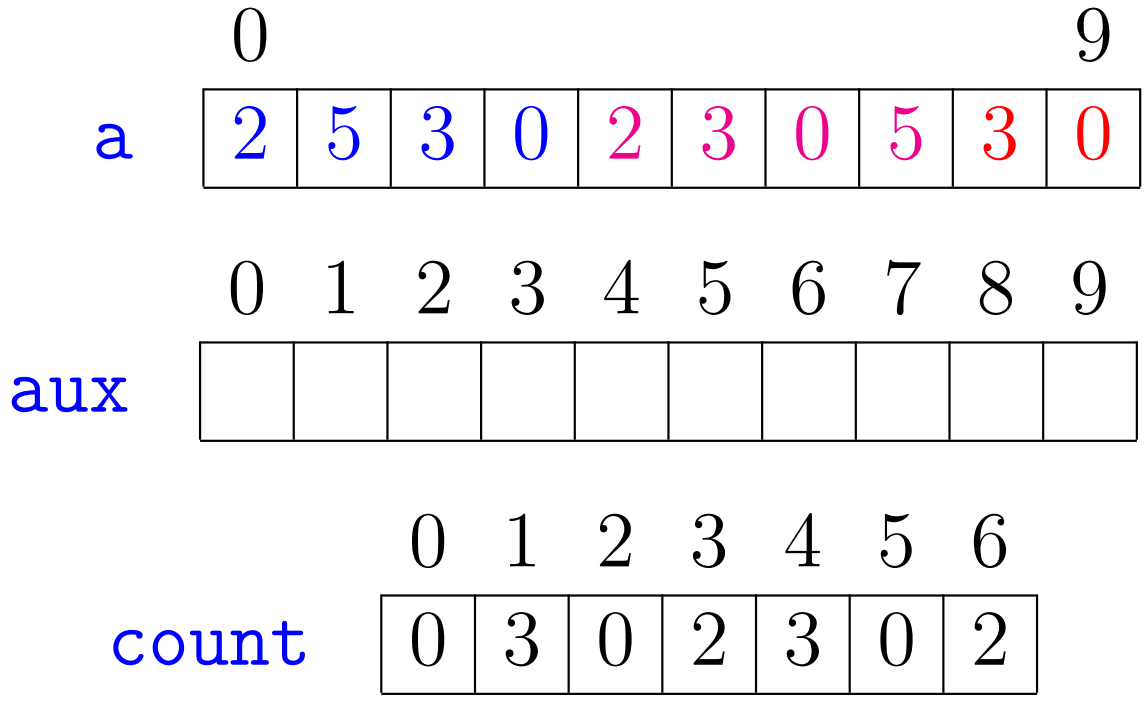
Passo 1: calcula frequências

Cada a[i] está em $\{0, \ldots, 5\}$.



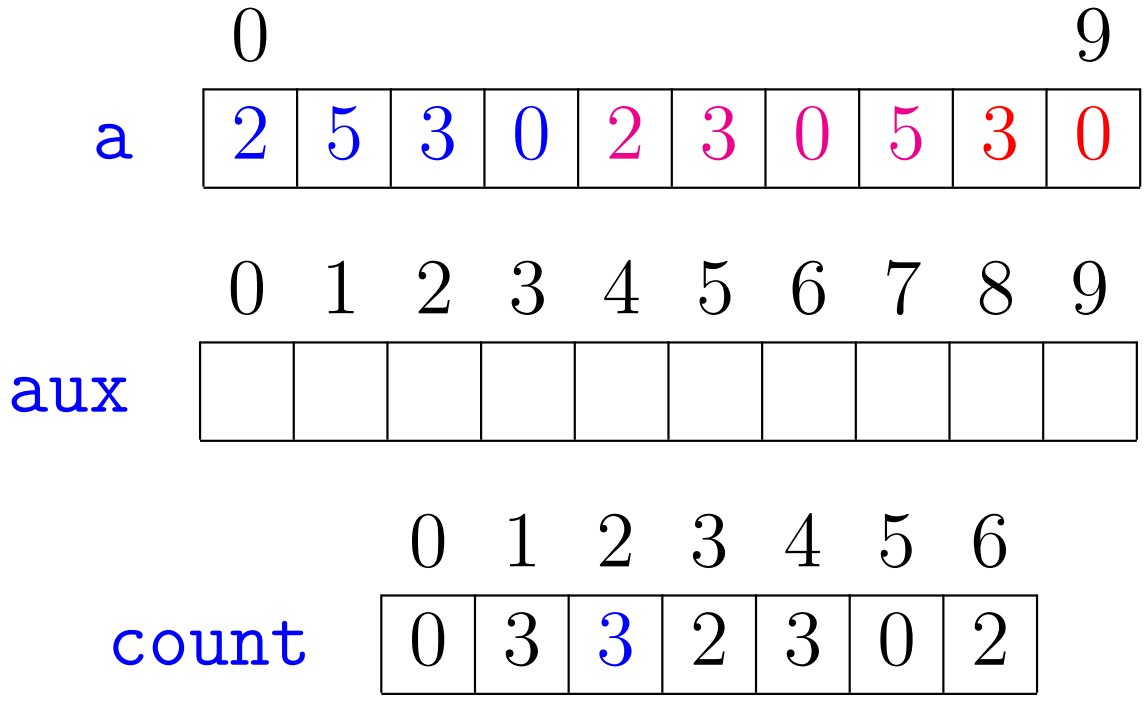
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



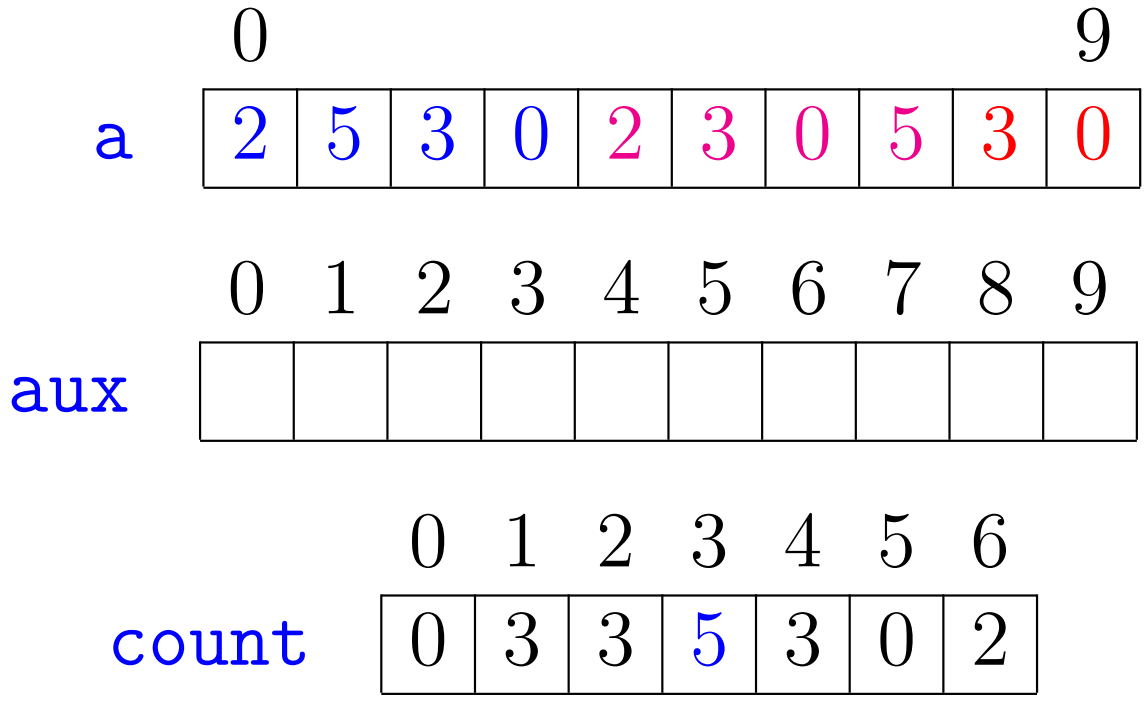
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



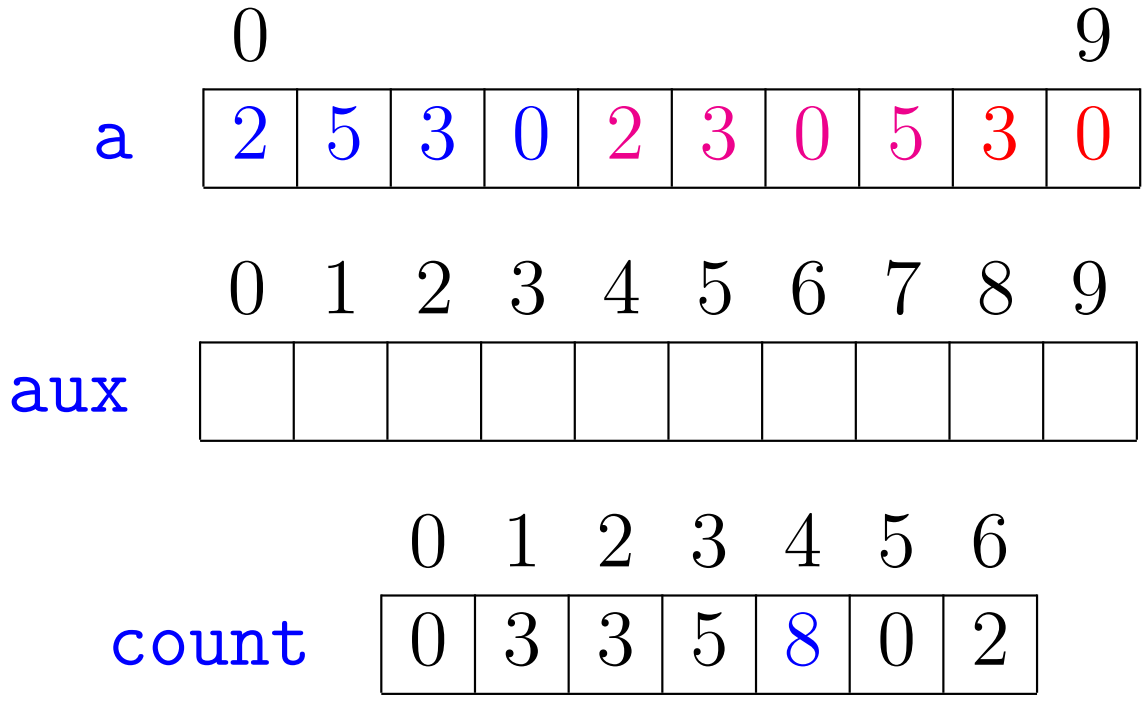
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



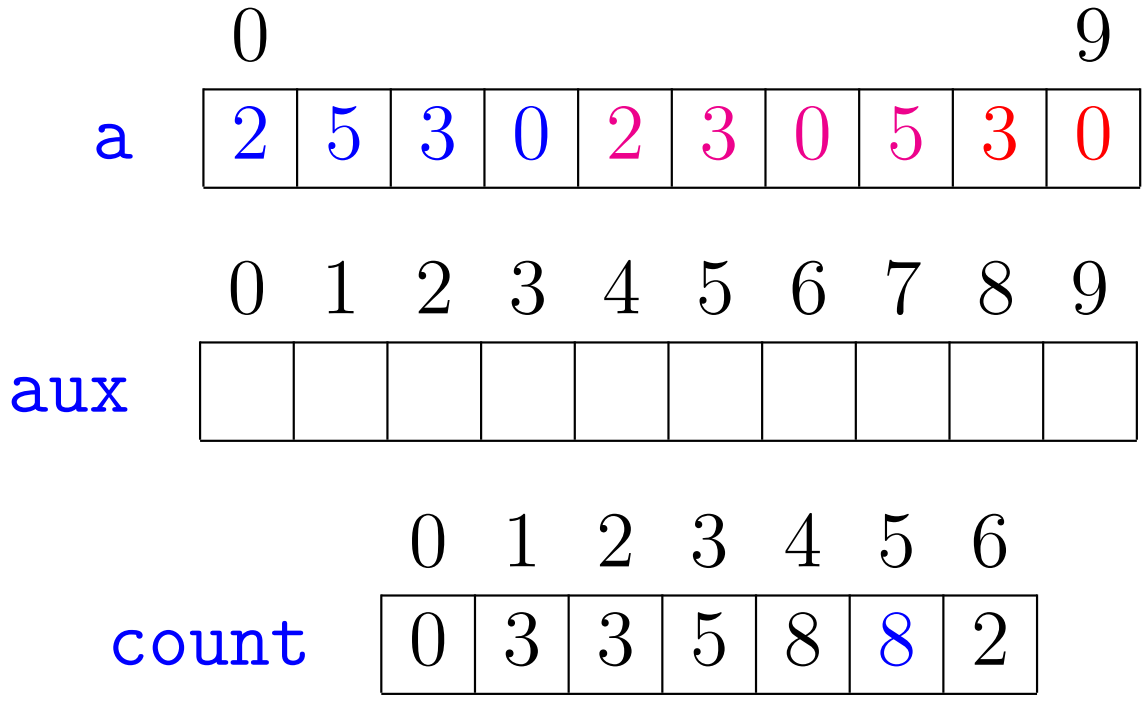
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



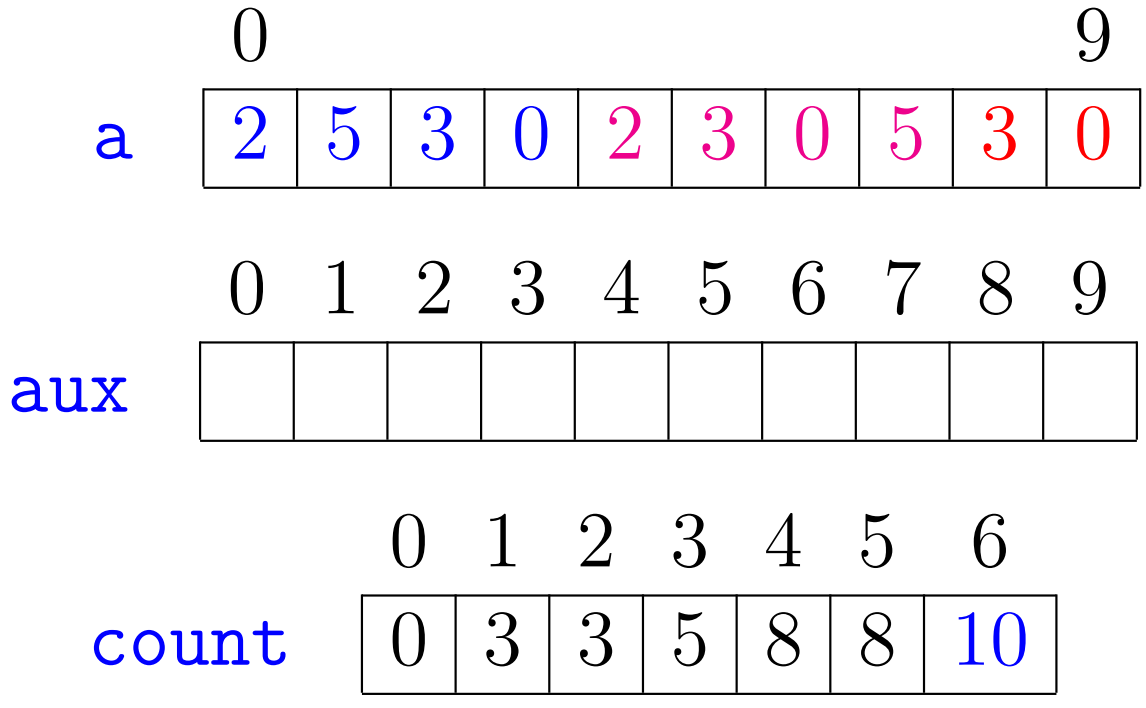
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



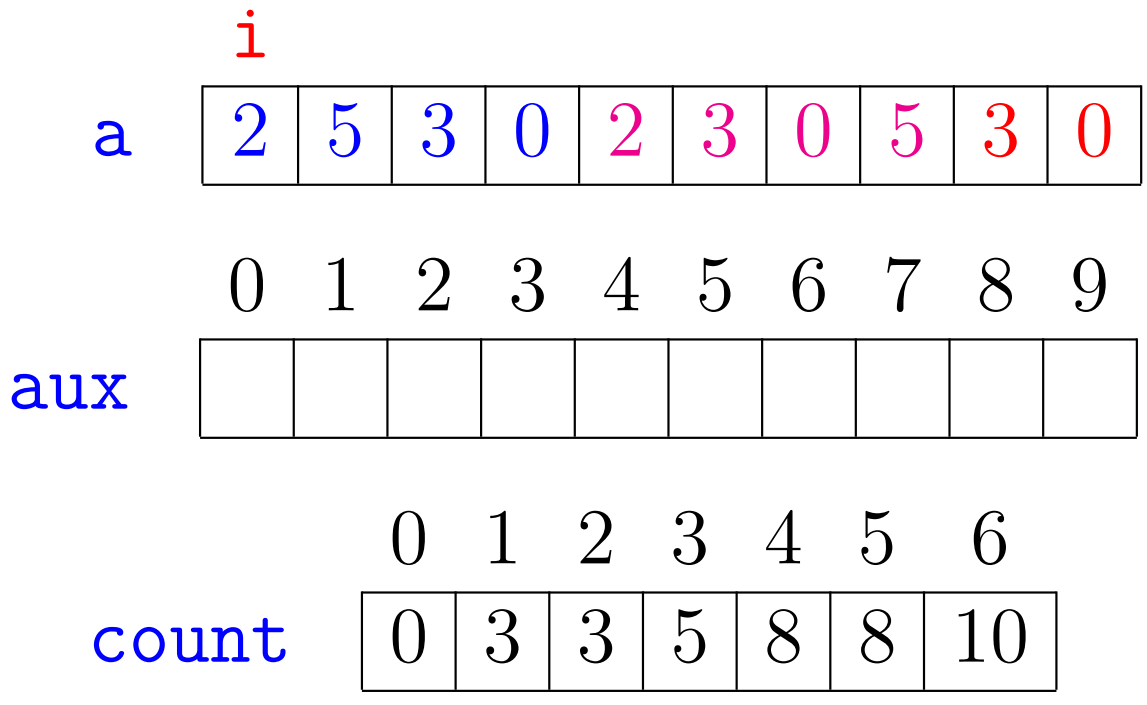
Passo 2: transforma frequências em índices

Cada a[i] está em $\{0, \ldots, 5\}$.



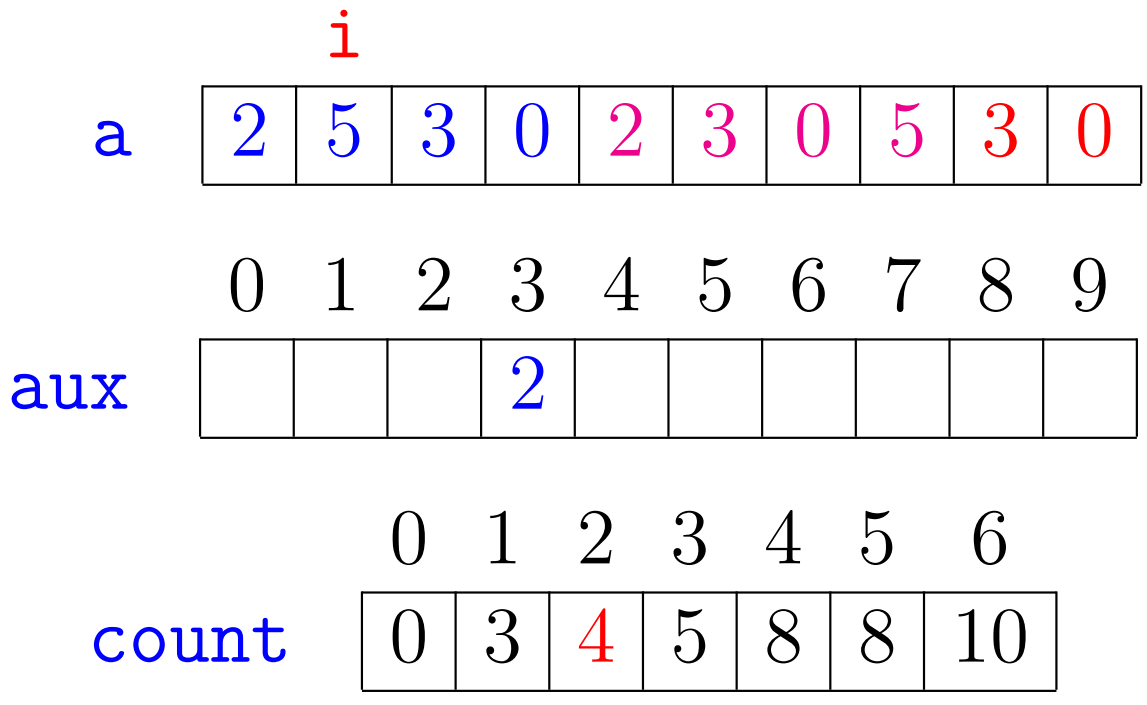
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



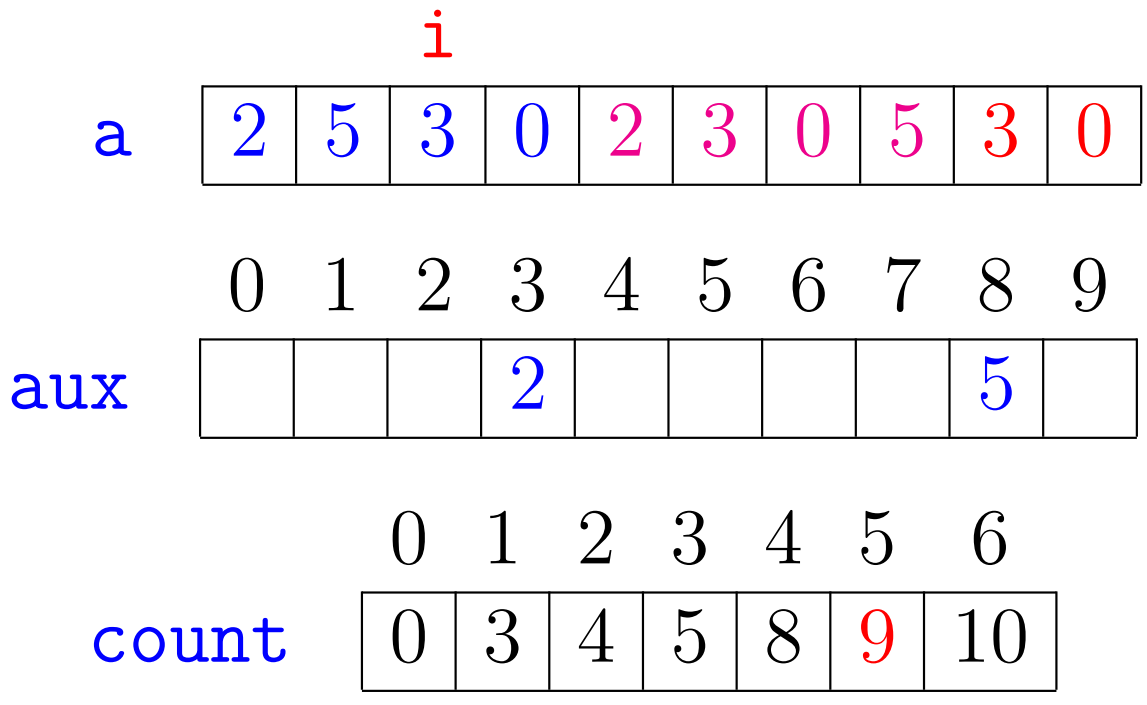
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



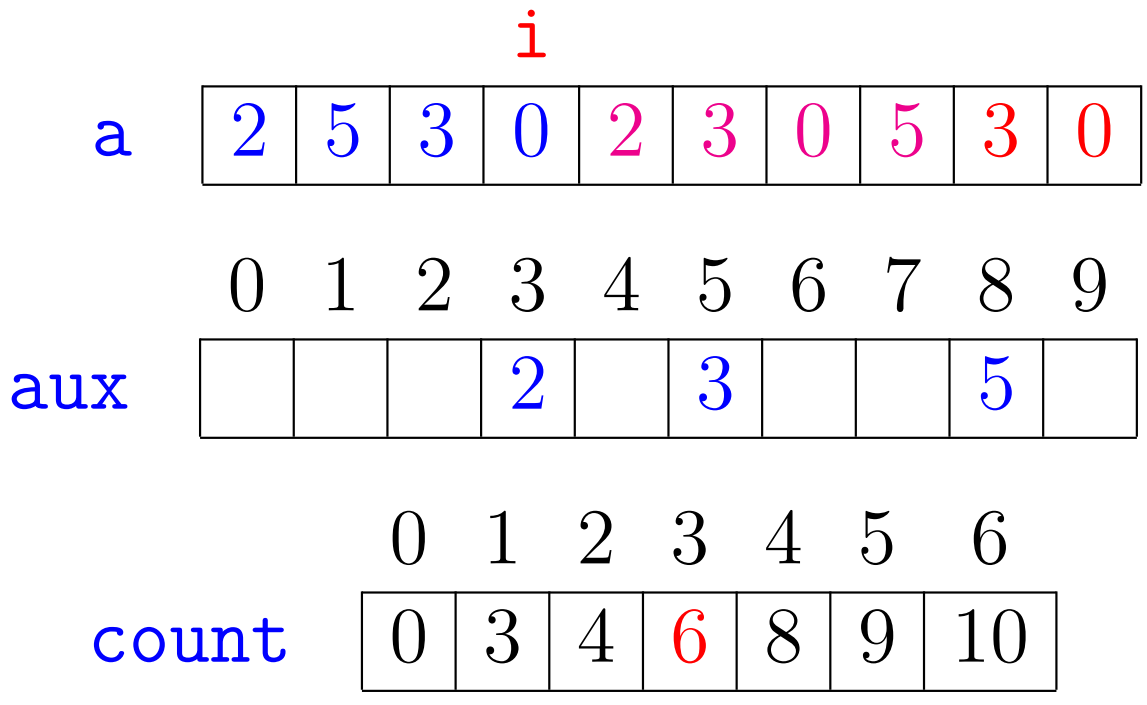
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



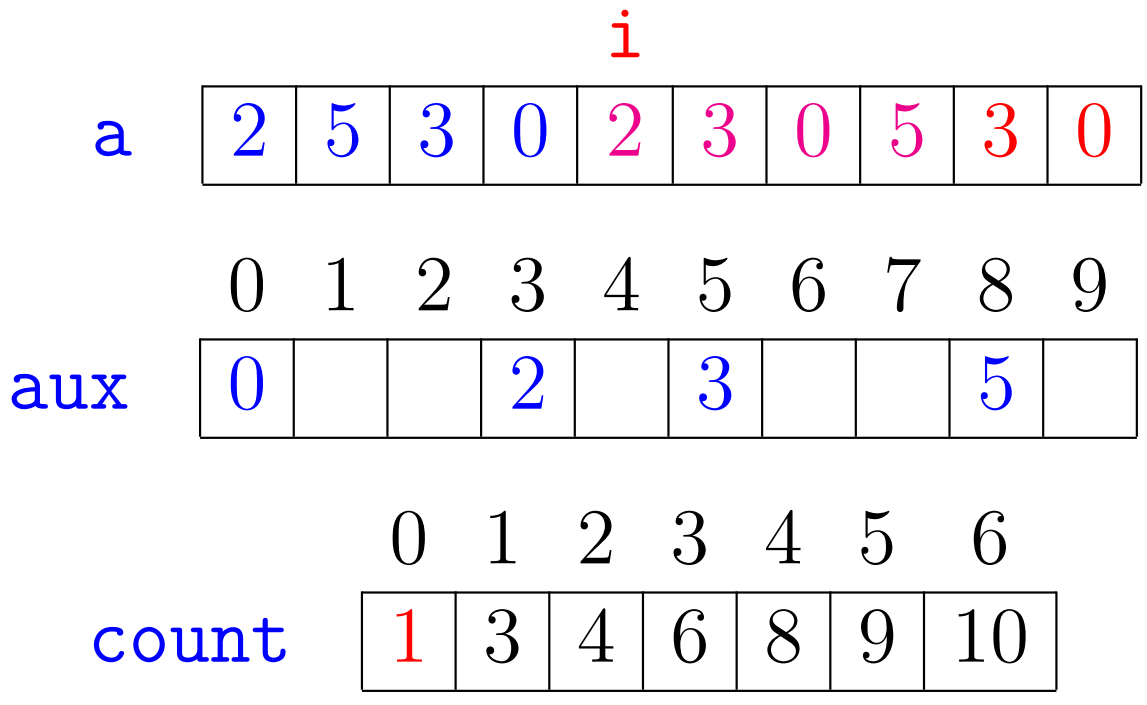
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



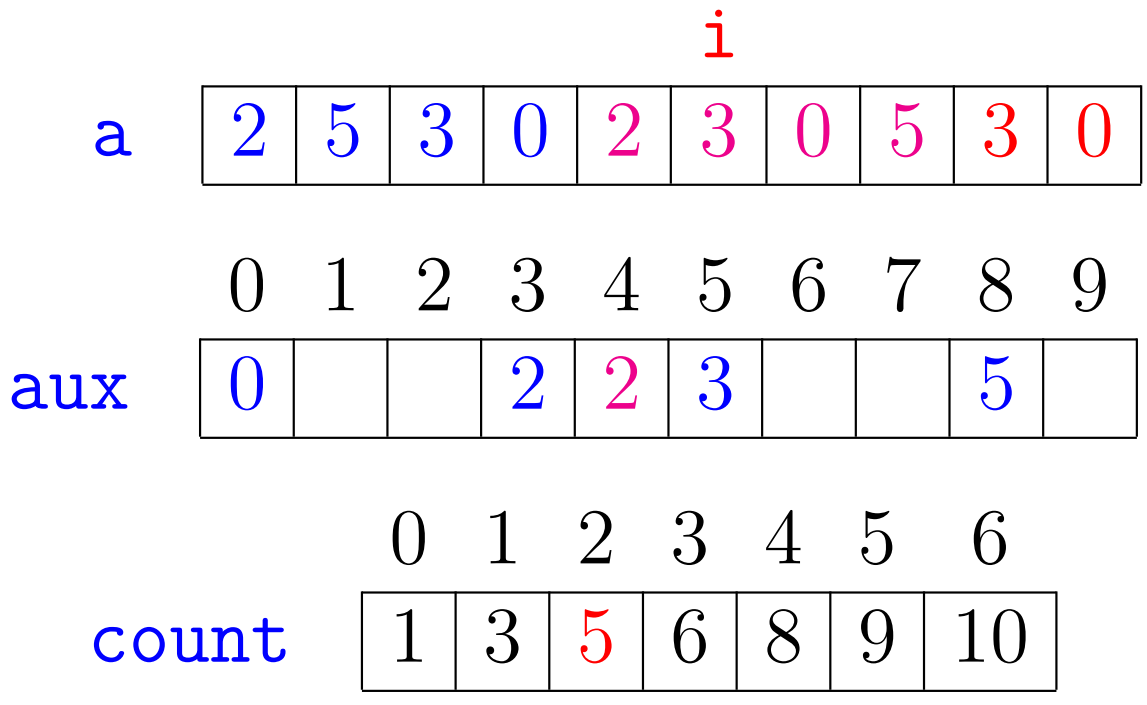
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



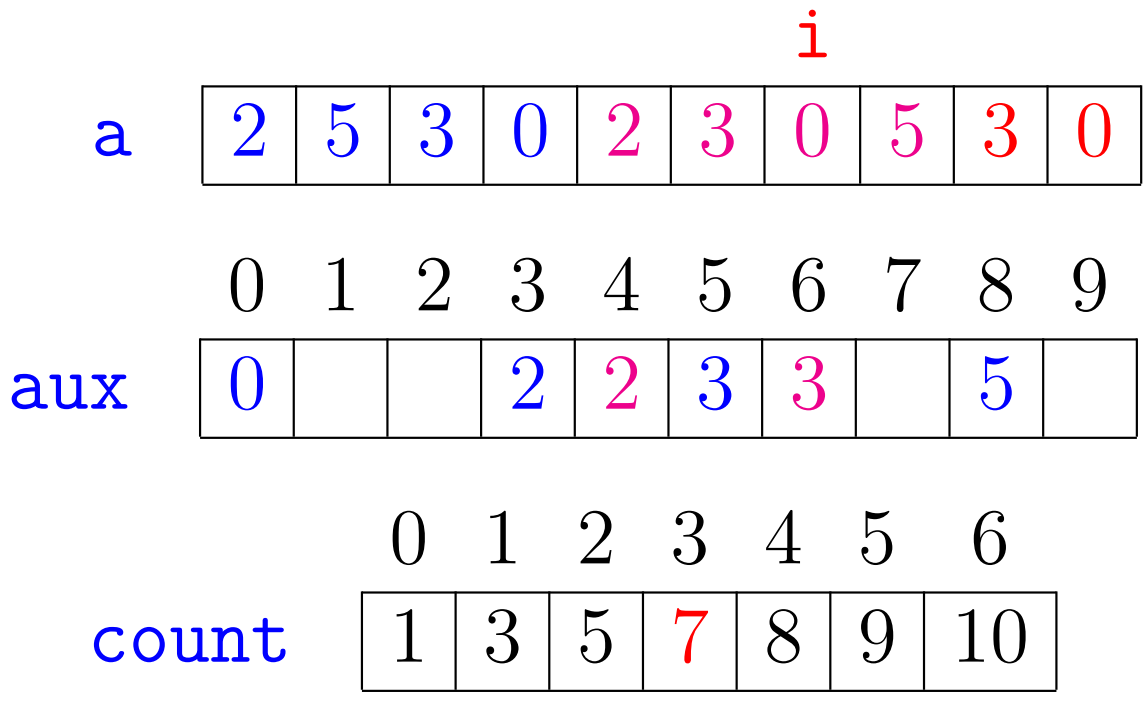
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



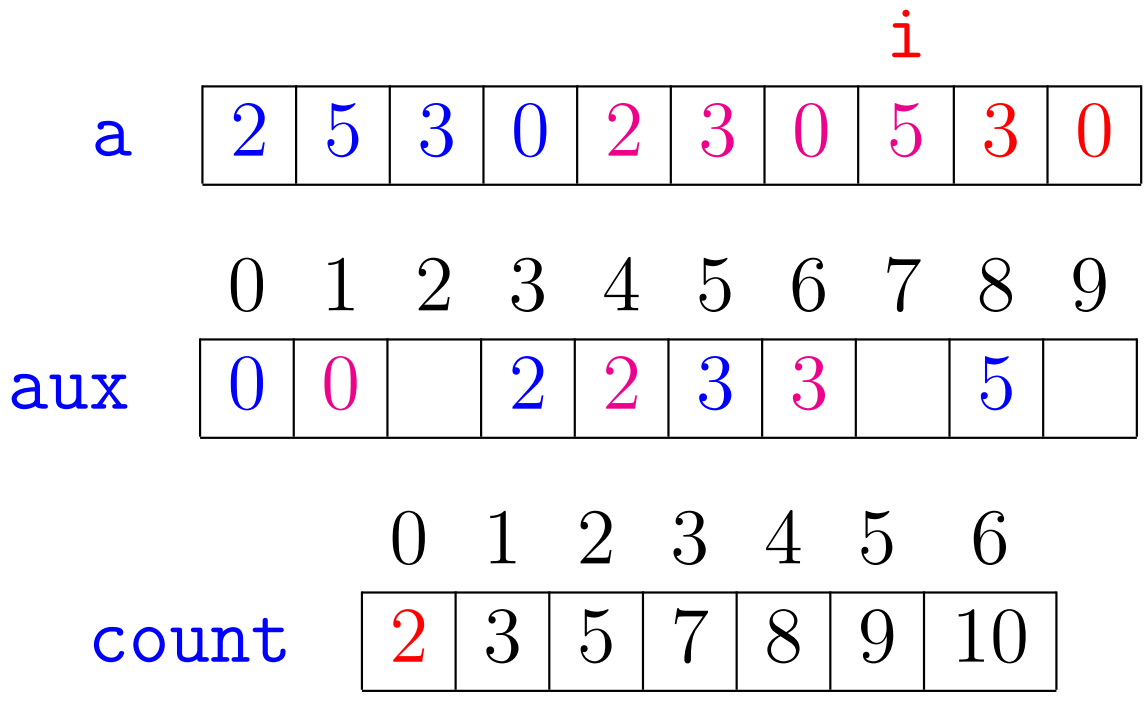
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



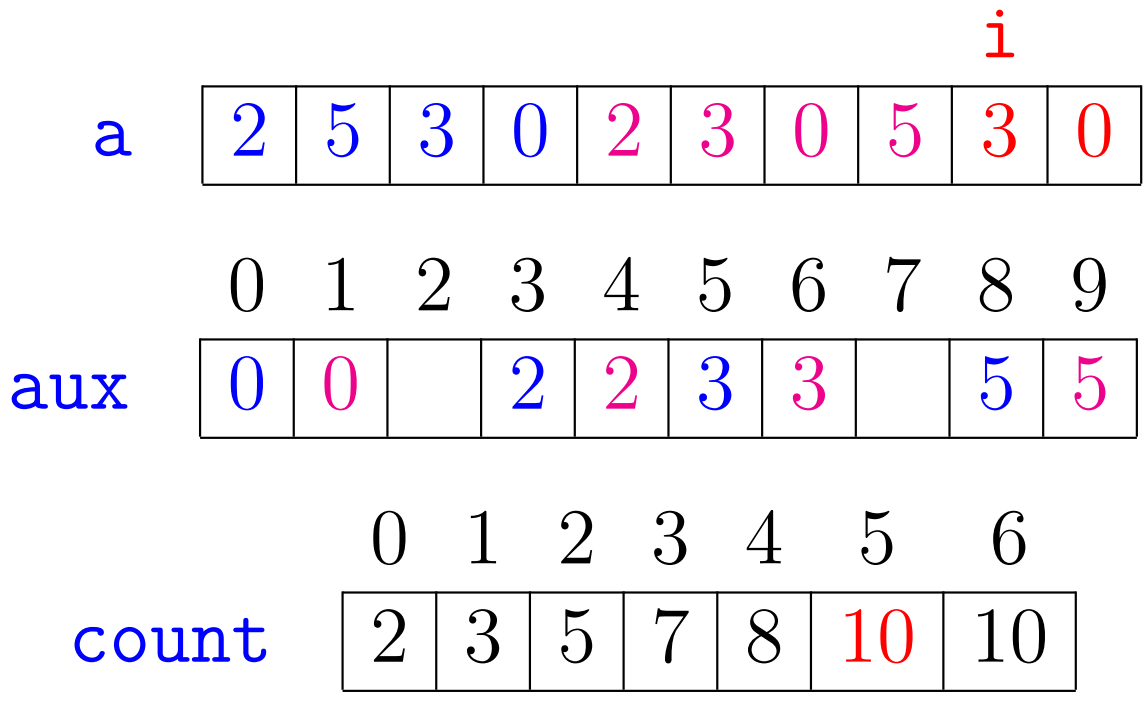
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



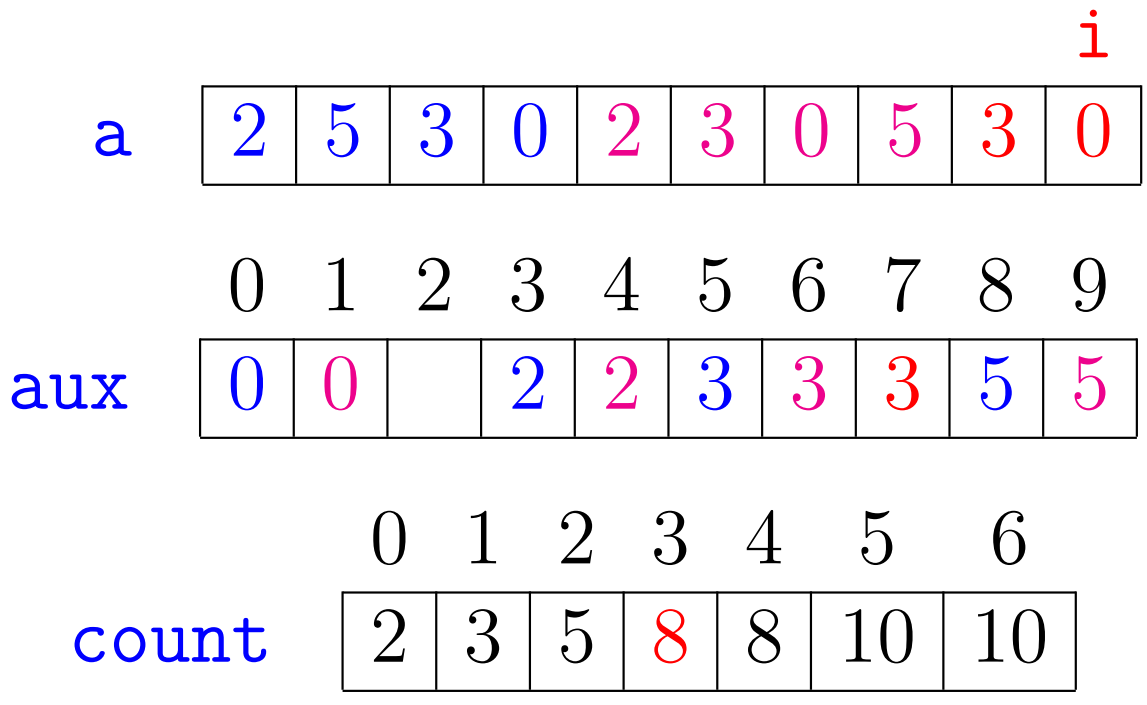
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



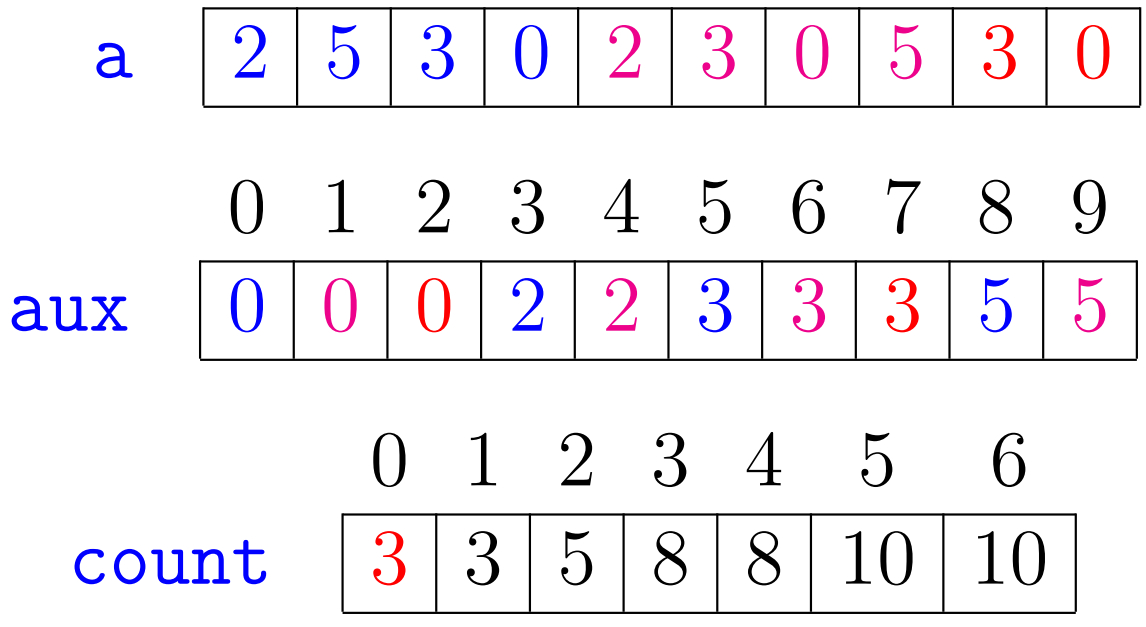
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



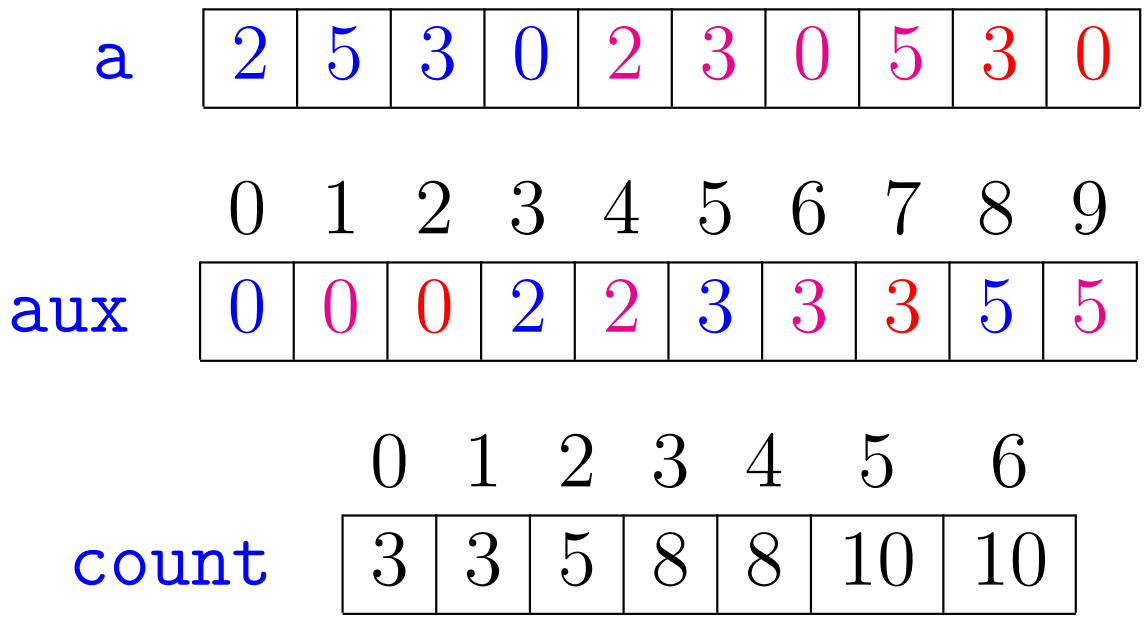
Passo 3: distribuição das chaves

Cada a[i] está em $\{0, \ldots, 5\}$.



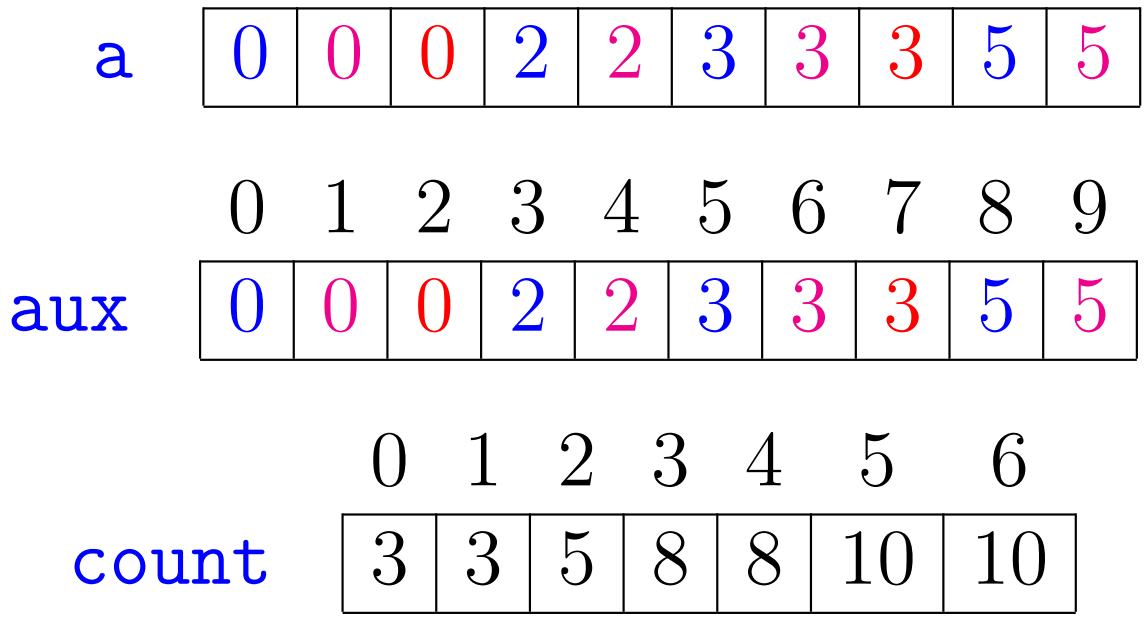
Passo 4: copia chaves ordenadas para a

Cada a[i] está em $\{0, \ldots, 5\}$.



Passo 4: copia chaves ordenadas para a

Cada a[i] está em $\{0, \ldots, 5\}$.



Ordenação por contagem

```
    int n = strlen(a), count[R+1];
 1  for (int r = 0; r <= R; r++)
 2     count[r] = 0;
 3  for (int i = 0; i < n; i++)
 4     count[a[i]+1]++;
 5  for (int r = 0; r <= R; r++)
 6     count[r+1] += count[r];
    /* fase de distribuição */
 7  for (int i = 0; i < n; i++)
 8     aux[count[a[i]]++] = a[i];
 9  for (int i = 0; i < n; i++)
10     a[i] = aux[i];
```

Obs: não são feitas comparações entre chaves.

Consumo de tempo

linha	consumo na linha
$1{-}2$	$\Theta(\mathtt{R})$
$3{-}4$	$\Theta(\mathtt{n})$
$5{-}6$	$\Theta(\mathtt{R})$
$7{-}8$	$\Theta(\mathtt{n})$
$9{-}10$	$\Theta(\mathtt{n})$

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

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.

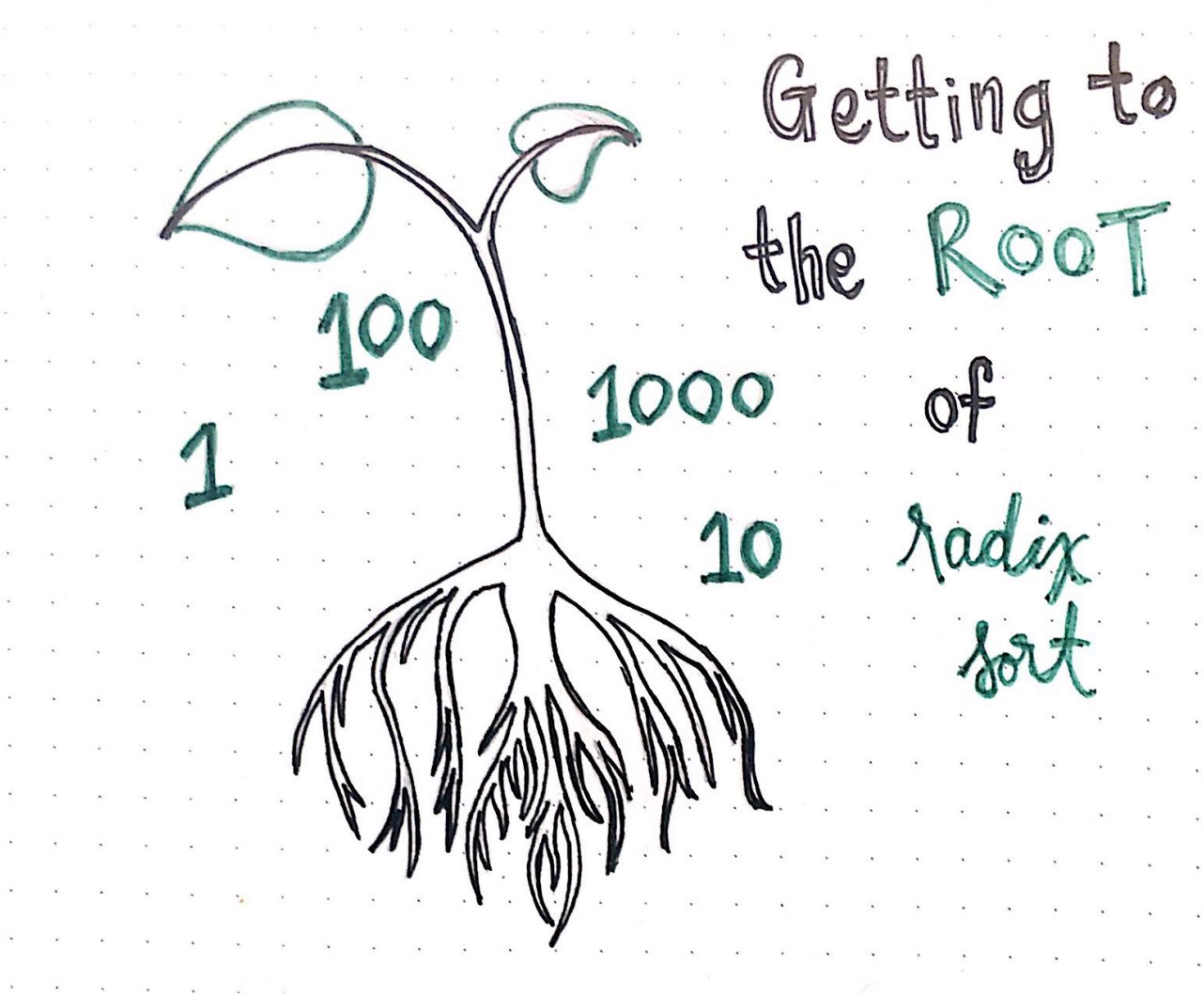
a	2	5	3	0	2	3	0	5	3	0

	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.

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.

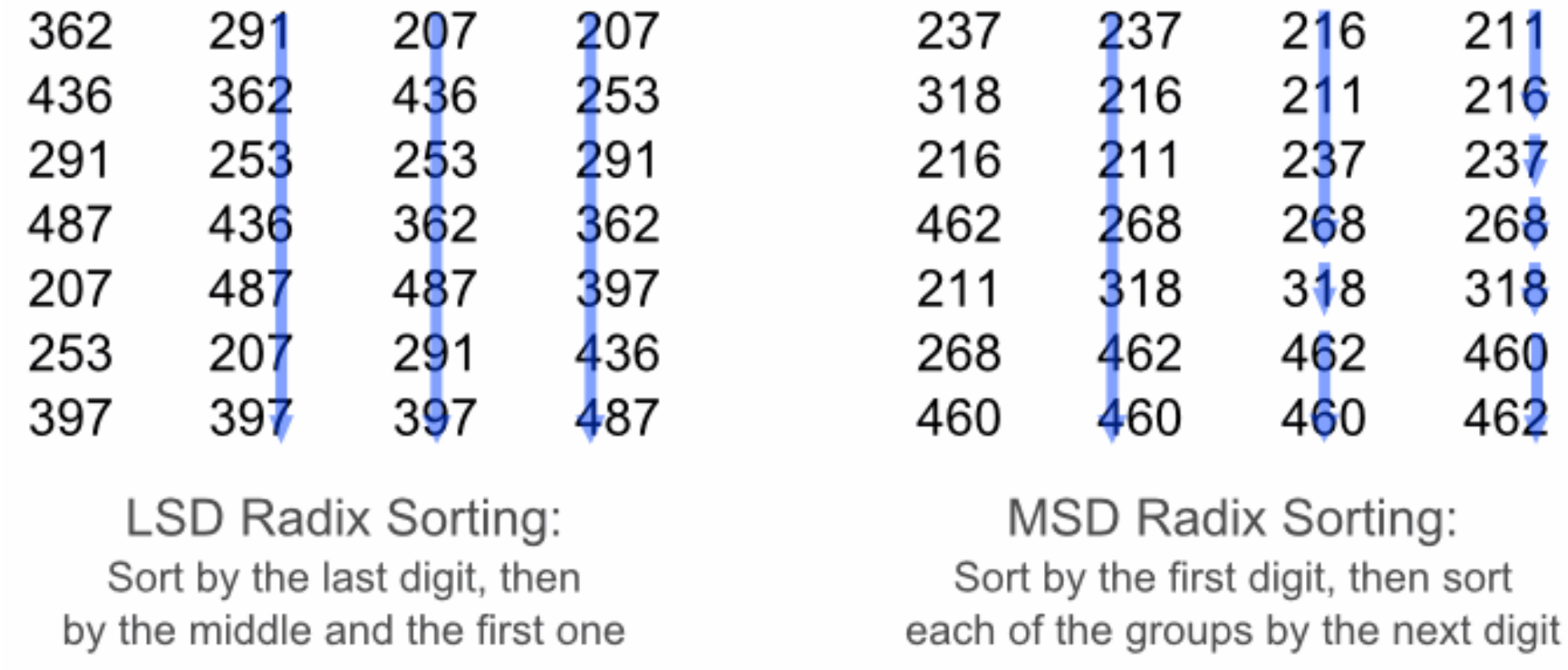
Se as chaves são strings,
os símbolos podem ser seus caracteres.

LSD e MSD

Ordenação digital aparece em geral 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

LSD ideia

Exemplo:

$$
\begin{array}{r}
329 \\
457 \\
657 \\
839 \\
436 \\
720 \\
355
\end{array}
$$

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

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

```
/* extended ASCII */
static int R = 256;

void sort(char **a, int n, int W) {

   char **aux;
   int *count;

   aux   = mallocSafe(n*sizeof(char*));
   count = mallocSafe((R+1)*sizeof(int));
```

LSD

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

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 da ordenação por contagem:
$\Theta(n + R) = \Theta(n + 2^r)$.
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	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
♦J	♣9	♦10

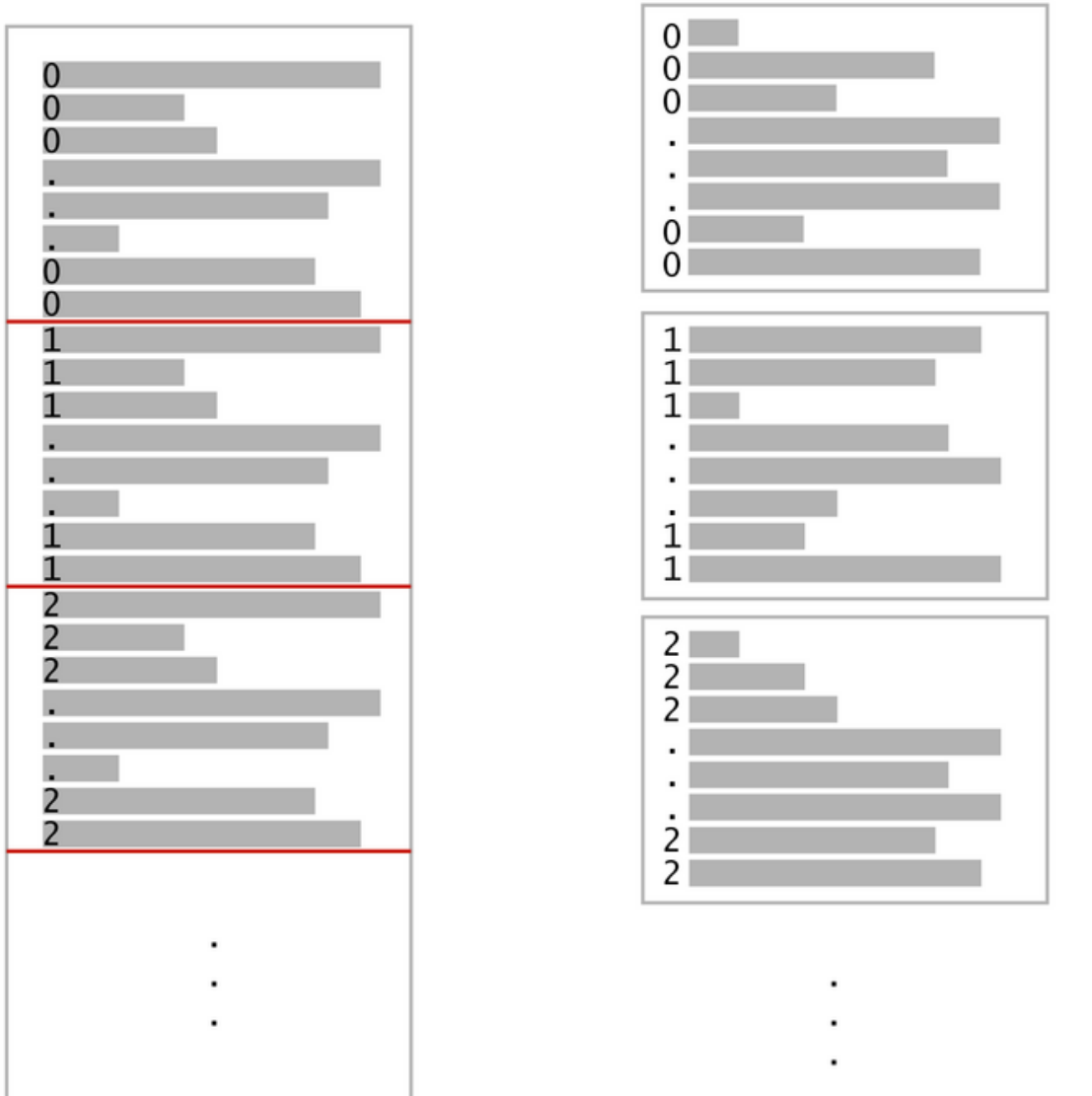
LSD características

- 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` acessos a posições das strings.
- Utiliza espaço extra proporcional a $n + R$.

Most-Significant-Digit

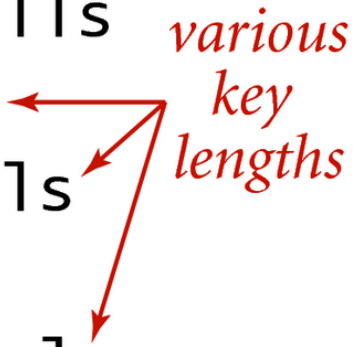
sort on first character value to partition into subarrays

recursively sort subarrays (excluding first character)



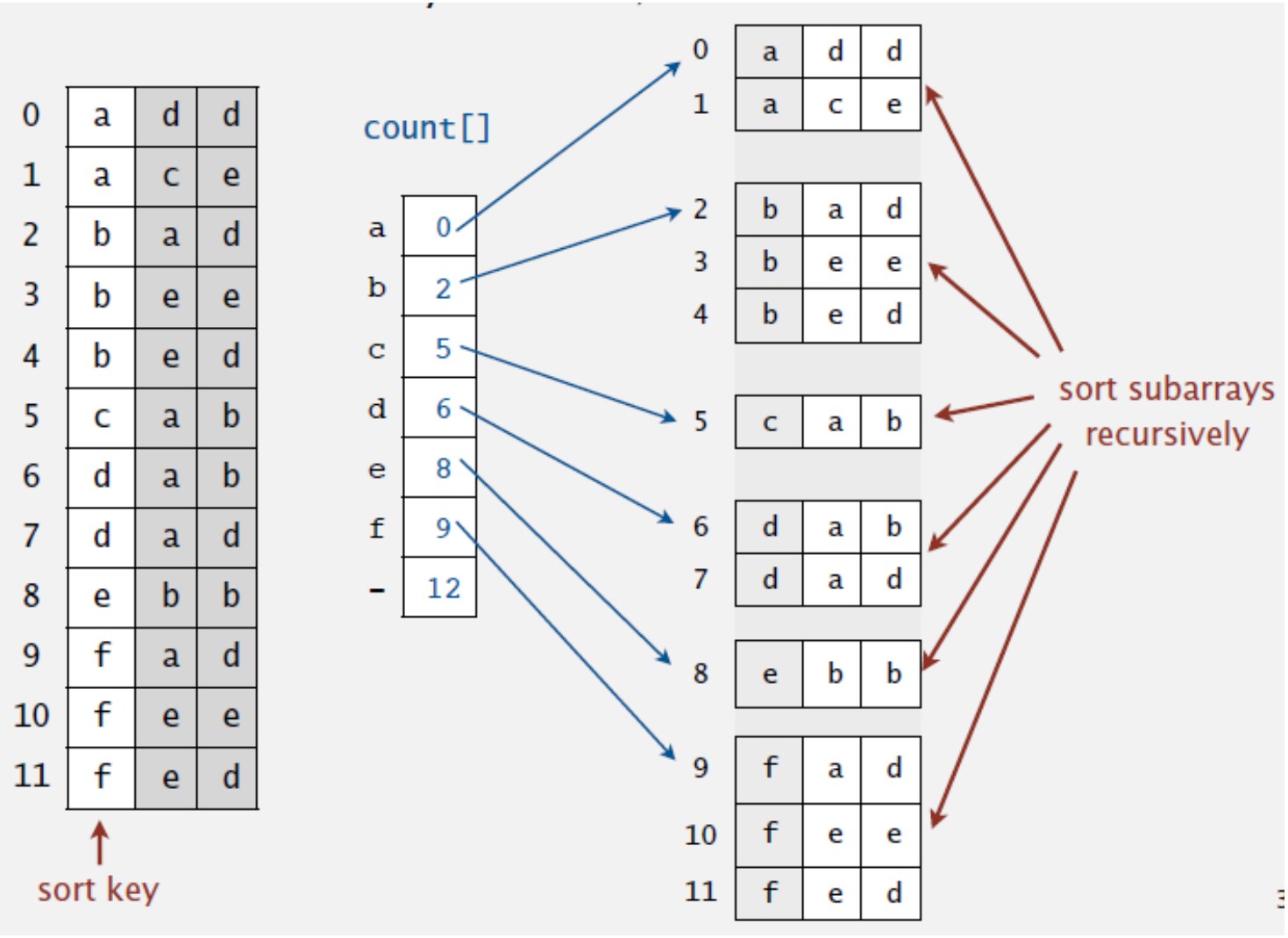
MSD candidato

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



Fonte: algs4

MSD recursão



Fonte: algs4

MSD

```
static int R = 256;

/* corte para usar inserção */
static int M = 15;

void sort(char **a, int n) {
   char **aux;
   aux = mallocSafe(n*sizeof(char*));
   sortR(a, 0, n-1, 0, aux);
}
```

MSD

```
static int R = 256;

/* corte para usar inserção */
static int M = 15;

void sort(char **a, int n) {
   char **aux;
   aux = mallocSafe(n*sizeof(char*));
   sortR(a, 0, n-1, 0, aux);
}
static int charAt(char *s, int d) {
   if (d >= strlen(s)) return -1;
   return s[d];
}
```

MSD

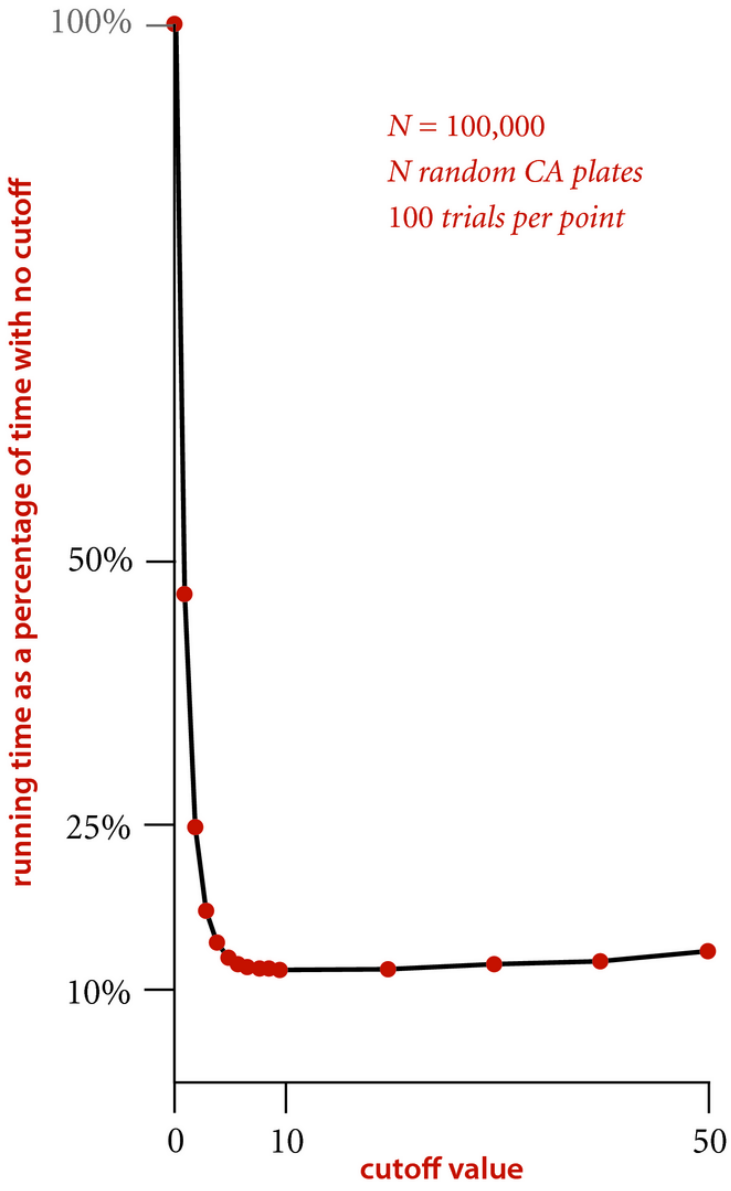
```
static void sortR(char **a, int lo, int hi,
                  int d, char **aux) {
   int *count, r, i, c;

   if (hi <= lo + M) {
      insertion(a, lo, hi, d);
      return;
   }
   count = mallocSafe((R+2)*sizeof(int));
   for (r = 0; r <= R+1; r++)
      count[r] = 0;
```

MSD

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

MSD



Effect of cutoff for small subarrays

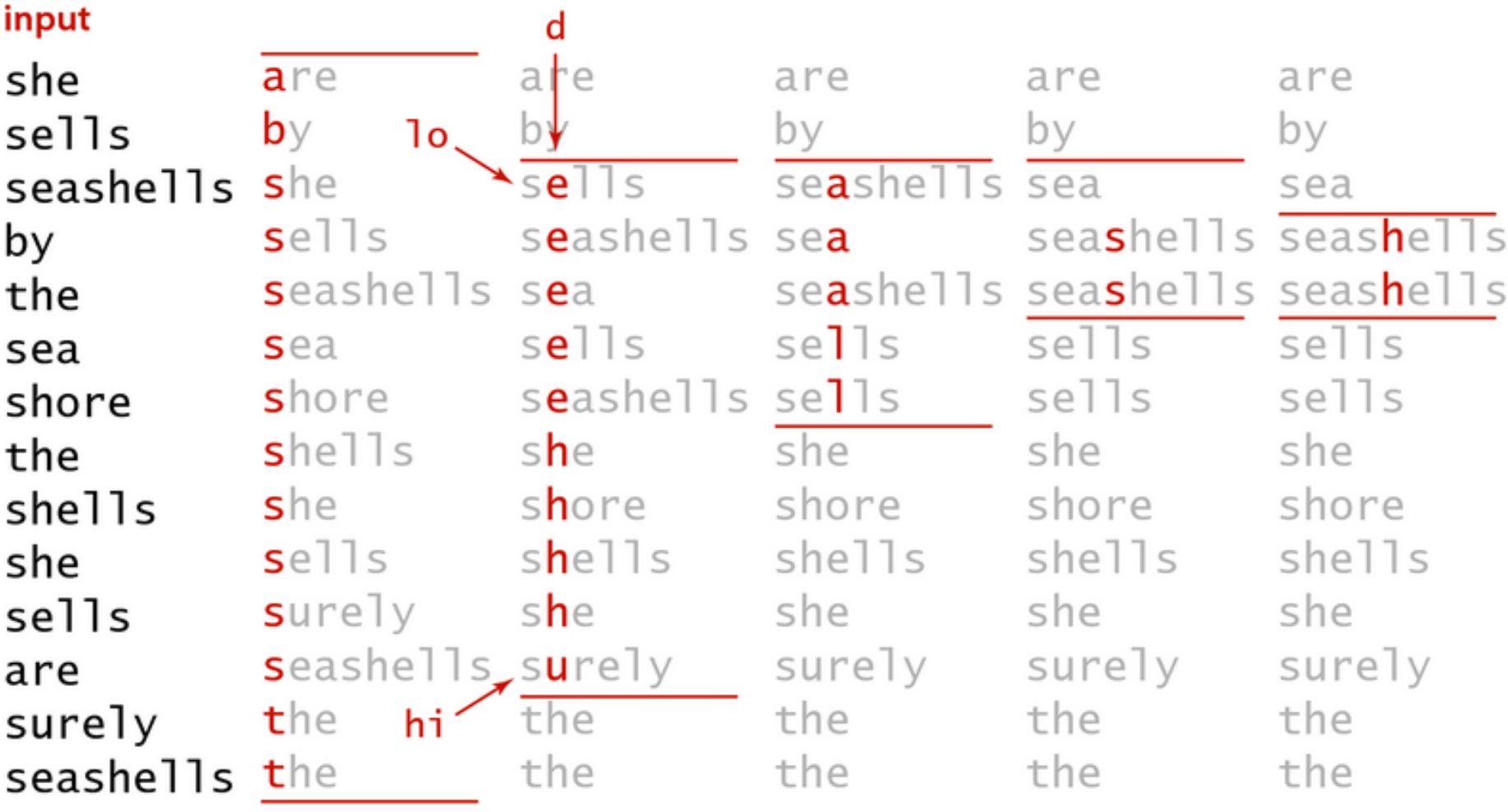
MSD caracteres examinados

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 em ação

Fonte: algs4



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
♦J	♦3	♦10

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 $n + R \times W$ (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 à ordenação por contagem.
- Espaço extra para `count[]`
  devido à ordenação por contagem.
- Laço interno com muitas instruções
  devido à 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 $\mathrm{O}(\mathtt{n} \log \mathtt{n})$ e não linear.
- deve examinar várias vezes os mesmos caracteres de chaves com longos prefixos iguais.

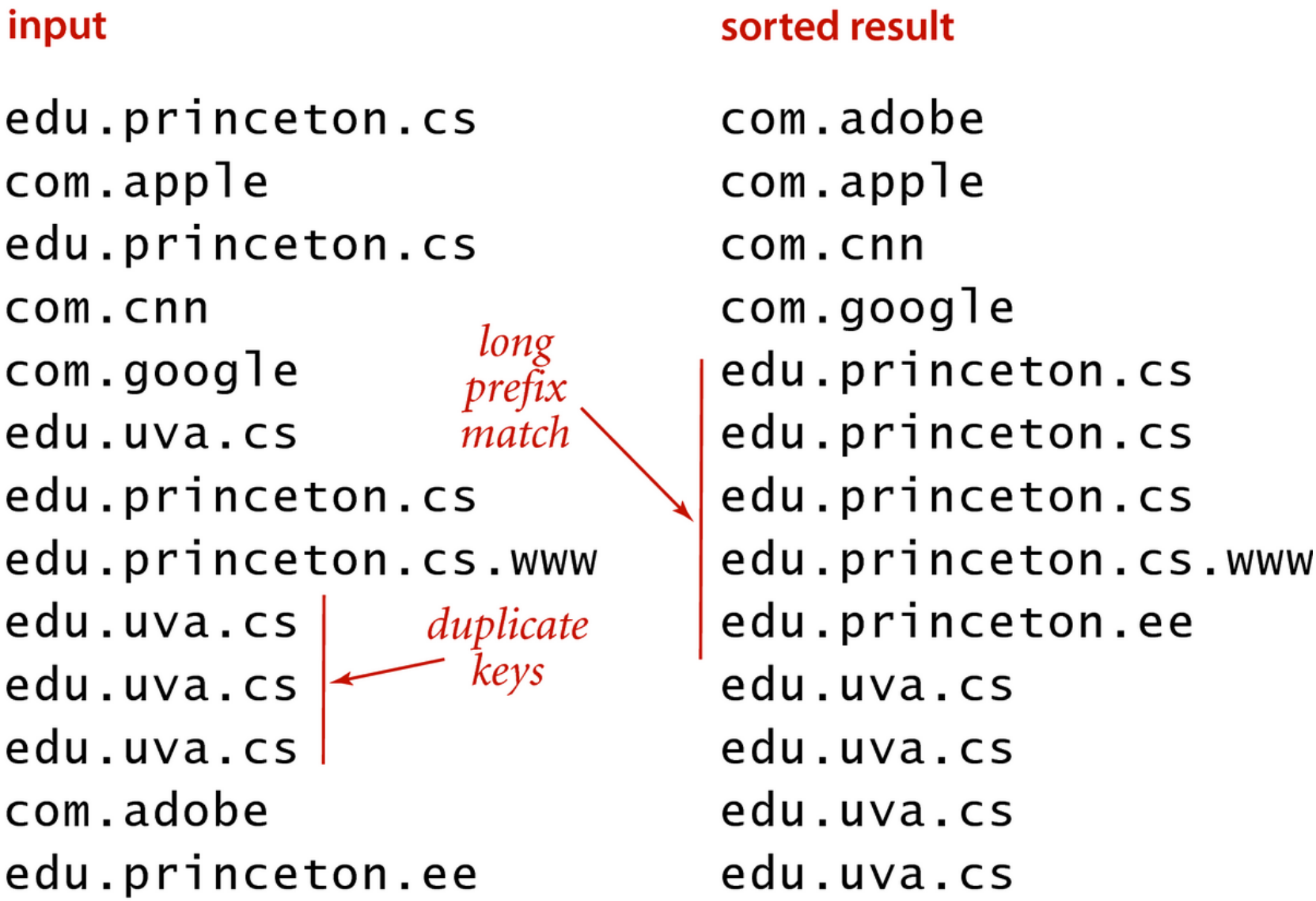
3-way string quicksort: ideia

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



Fonte: algs4

Quick3string

```
static int M = 15;

void sort(char **a, int n) {
   sortR(a, 0, n-1, 0);
}

static int charAt(char *s, int d) {
   if (d <= strlen(s)) return -1;
   return s[d];
}
```

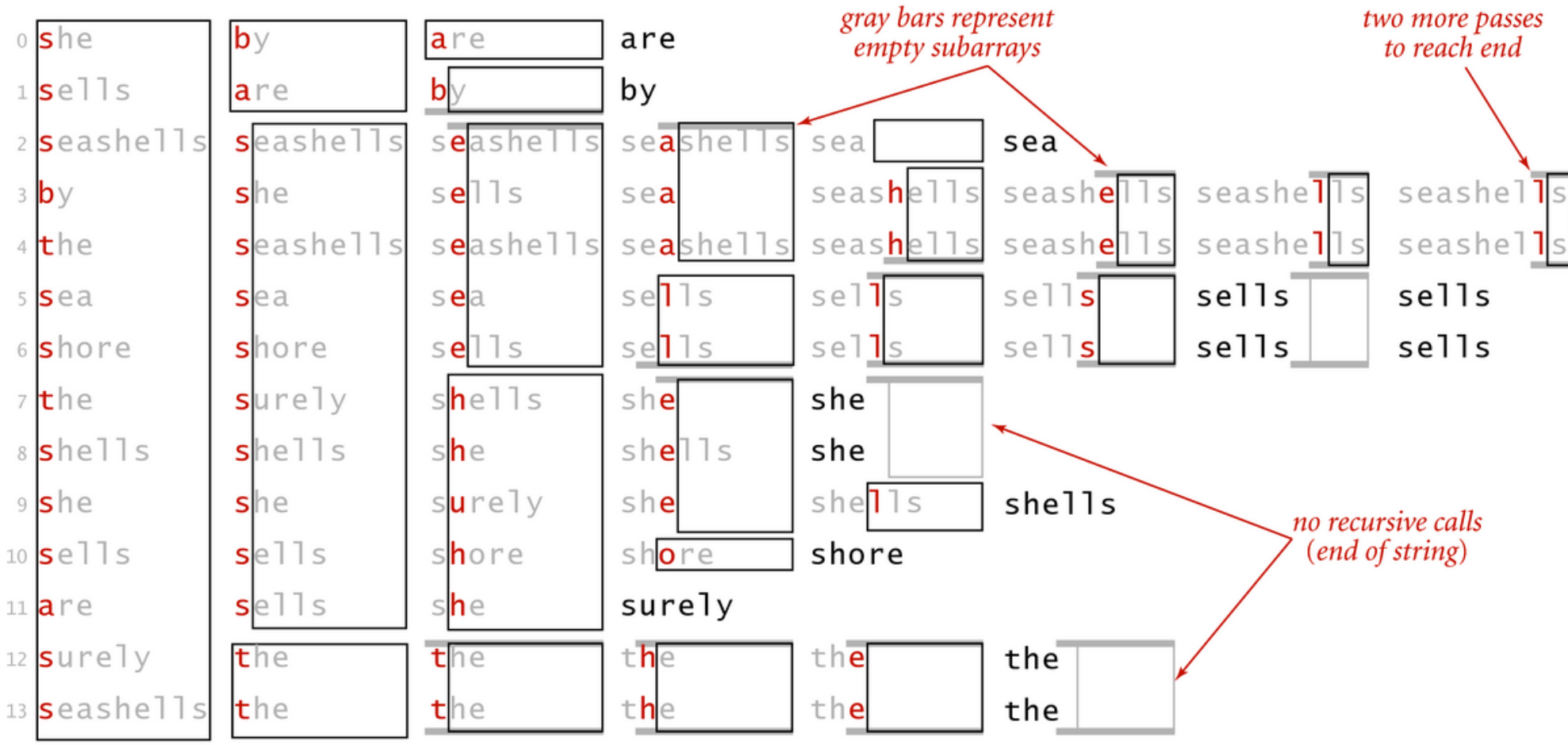
Quick3string

```
/* 3-way string quicksort a[lo..hi] */
/* começando no caractere d */
static void sortR(char **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++;
    }
    sortR(a, lo, lt-1, d);
    if (v >= 0) sort(a, lt, gt, d+1);
    sortR(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