Algoritmos de Ordenação Linear

Profª. Rose Yuri Shimizu

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes

- Linearítmicos
  - $O(n \log n)$
  - Melhor custo quando a ordenação é por comparação do valor da chave
  - Vantagem: mais amplo (vários tipos de chaves podem usar o mesmo algoritmo)
- Lineares
  - $O(n)$
  - Melhor custo quando a ordenação é por comparação na estrutura da chave:
    - intervalo de chaves de 0 até R-1
    - inteiros de 32 bits
  - Desvantagem: mais restrito
    - amarrado ao um tipo de chave
    - conhecimento prévio do intervalo das chaves

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Counting Sort
   - Radix Sort

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - ★ 0: 0 vezes

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes
  - Ordenando:
    - zero 0's, três 1's, cinco 2's, seis 3's, seis 4's

  v [

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  `v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]`

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes
  - Ordenando:
    - zero 0's, três 1's, cinco 2's, seis 3's, seis 4's

  `v [1 1 1`

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes
  - Ordenando:
    - zero 0's, três 1's, cinco 2's, seis 3's, seis 4's

  v [1 1 1 2 2 2 2 2

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes
  - Ordenando:
    - zero 0's, três 1's, cinco 2's, seis 3's, seis 4's

  v [1 1 1 2 2 2 2 2 3 3 3 3 3 3

Algoritmos de Ordenação Eficientes - Counting Sort

- Ordenar contando as chaves em um vetor auxiliar:
  - Cada índice é uma chave: limita quantidade de chaves
  - Determina a posição da uma chave, contando quantas chaves menores existem
  - Histograma dos números: distribuição das frequências
  - Cada chave é reposicionada na posição definitiva
- Exemplo:

  v [2 3 3 4 1 3 4 3 1 2 2 1 2 4 3 4 4 2 3 4]

  - Frequências:
    - 0: 0 vezes
    - 1: 3 vezes
    - 2: 5 vezes
    - 3: 6 vezes
    - 4: 6 vezes
  - Ordenando:
    - zero 0's, três 1's, cinco 2's, seis 3's, seis 4's

  v [1 1 1 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4]

Algoritmos de Ordenação Eficientes - Counting Sort

Etapas para implementação:

- Etapa 1: contar as frequências
  - Para o intervalo de chaves: 0 até $R-1$
  - Utiliza-se: `count[R+1]`
  - Cada chave i em 0 até $R-1$ = `count[i]`
  - Cada `count[i]` = frequência da chave $i-1$ (imediatamente < i)
  - Portanto,

```
1 //memset(count, 0, sizeof(int)*(R+1))
2 for(i=0; i<=R; i++) count[i] = 0;
3 for(i=l; i<=r; i++) count[v[i]+1]++;
```

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

count[ ]

0	1	2	**3**	4	5
0	0	0	0	0	0

2

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2
4	0	0	0	1	2	**1**	count[5] += qtde. de chaves < 5 = 1

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2
4	0	0	0	1	2	**1**	count[5] += qtde. de chaves < 5 = 1
1	0	0	**1**	1	2	1	count[2] += qtde. de chaves < 2 = 1

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2
4	0	0	0	1	2	**1**	count[5] += qtde. de chaves < 5 = 1
1	0	0	**1**	1	2	1	count[2] += qtde. de chaves < 2 = 1
3	0	0	1	1	**3**	1	count[4] += qtde. de chaves < 4 = 3

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2
4	0	0	0	1	2	**1**	count[5] += qtde. de chaves < 5 = 1
1	0	0	**1**	1	2	1	count[2] += qtde. de chaves < 2 = 1
3	0	0	1	1	**3**	1	count[4] += qtde. de chaves < 4 = 3
4	0	0	1	1	3	**2**	count[5] += qtde. de chaves < 5 = 1

R = 5, intervalo 0 - 4 , count[6]

v [2 3 3 4 1 3 4 1]

Contando as frequências:

	count[ ]						
	0	1	2	**3**	4	5	
	0	0	0	0	0	0	
2	0	0	0	**1**	0	0	count[3] += qtde. de chaves < 3 = 1
3	0	0	0	1	**1**	0	count[4] += qtde. de chaves < 4 = 1
3	0	0	0	1	**2**	0	count[4] += qtde. de chaves < 4 = 2
4	0	0	0	1	2	**1**	count[5] += qtde. de chaves < 5 = 1
1	0	0	**1**	1	2	1	count[2] += qtde. de chaves < 2 = 1
3	0	0	1	1	**3**	1	count[4] += qtde. de chaves < 4 = 3
4	0	0	1	1	3	**2**	count[5] += qtde. de chaves < 5 = 1
1	0	0	**2**	1	3	2	count[2] += qtde. de chaves < 2 = 2

count [0 0 2 1 3 2]

zero 0's, dois 1's, um 2, três 3's, dois 4's

Algoritmos de Ordenação Eficientes - Counting Sort

- Etapa 2: Calculando as posições/índices através das frequências
  - Se `count[i]` contém a quantidade de chaves imediatamente menores que `i`,
  - Então, se, `count[i] = count[i] + count[i-1] + ... + count[0]`,
  - `count[i]` vai conter a quantidade de todas as chaves menores que `i`,
  - `count[i]` vai conter a quantidade de posições que devem ser "puladas" para a inserção das chaves menores que `i`,
  - Portanto, `count[i]` contém a distância de 0 até a chave `i` (posição de `i`)

```
1 for(i=1; i<=R; i++) count[i] += count[i-1];
```

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

Calculando as posições/índices:

i	count[ ]						
	0	1	2	3	4	5	
1	0	**0**	2	1	3	2	count[1] = count[1] + count[0] = 0 chaves < 1

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

Calculando as posições/índices:

i	**count[ ]**						
	0	1	2	3	4	5	
1	0	**0**	2	1	3	2	count[1] = count[1] + count[0] = 0 chaves < 1
2	0	0	**2**	1	3	2	count[2] = count[2] + count[1] = 2 chaves < 2

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

Calculando as posições/índices:

i	count[ ]						
	0	1	2	3	4	5	
1	0	**0**	2	1	3	2	count[1] = count[1] + count[0] = 0 chaves < 1
2	0	0	**2**	1	3	2	count[2] = count[2] + count[1] = 2 chaves < 2
3	0	0	2	**3**	3	2	count[3] = count[3] + count[2] = 3 chaves < 3

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

Calculando as posições/índices:

i	count[ ]						
	0	1	2	3	4	5	
1	0	**0**	2	1	3	2	count[1] = count[1] + count[0] = 0 chaves < 1
2	0	0	**2**	1	3	2	count[2] = count[2] + count[1] = 2 chaves < 2
3	0	0	2	**3**	3	2	count[3] = count[3] + count[2] = 3 chaves < 3
4	0	0	2	3	**6**	2	count[4] = count[4] + count[3] = 6 chaves < 4

v [2 3 3 4 1 3 4 1]

Contando as frequências: count [0 0 2 1 3 2]

Calculando as posições/índices:

i	count[ ]						
	0	1	2	3	4	5	
1	0	**0**	2	1	3	2	count[1] = count[1] + count[0] = 0 chaves < 1
2	0	0	**2**	1	3	2	count[2] = count[2] + count[1] = 2 chaves < 2
3	0	0	2	**3**	3	2	count[3] = count[3] + count[2] = 3 chaves < 3
4	0	0	2	3	**6**	2	count[4] = count[4] + count[3] = 6 chaves < 4
5	0	0	2	3	6	**8**	count[5] = count[5] + count[4] = 8 chaves < 5

count [0 0 2 3 6 8]

Algoritmos de Ordenação Eficientes - Counting Sort

- Etapa 3: distribuindo as chaves
  - `count[R+1]`: tabela de índices
  - `aux[r-l+1]`: auxiliar para copiar as chaves na ordem,
    - ★ `count[ v[i] ]`: posição ordenada da chave `v[i]`
    - ★ `aux[count[v[i]]] = v[i]`: `v[i]` em sua posição ordenada em `aux[ ]`
    - ★ `count[ v[i] ]++`: próxima `v[i]` em sua posição ordenada em `aux[ ]`

```
1 for (i = l; i <= r; i++) {
2     aux[count[v[i]]] = v[i];
3     count[v[i]]++;
4 }
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ _ _ _ _ _ _
  3
  3
  4
  1
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8
  3
  4
  1
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8
  4
  1
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8
  1
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8     1 _ 2 3 3 _ 4 _
  3
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8     1 _ 2 3 3 _ 4 _
  3     0 1 3 5 7 8
  4
  1
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8     1 _ 2 3 3 _ 4 _
  3     0 1 3 5 7 8     1 _ 2 3 3 3 4 _
  4     0 1 3 6 7 8
  1
```

Contando as frequências: count [0 0 2 1 3 2]
Calculando as posições/índices: count [0 0 2 3 6 2]
Distribuindo as chaves:

v[i]	count[v[i]]						aux[count[v[i]]]							
	0	1	2	3	4	5	0	1	2	3	4	5	6	7
2	0	0	2	3	6	8	_	_	2	_	_	_	_	_
3	0	0	3	3	6	8	_	_	2	3	_	_	_	_
3	0	0	3	4	6	8	_	_	2	3	3	_	_	_
4	0	0	3	5	6	8	_	_	2	3	3	_	4	_
1	0	0	3	5	7	8	1	_	2	3	3	_	4	_
3	0	1	3	5	7	8	1	_	2	3	3	3	4	_
4	0	1	3	6	7	8	1	_	2	3	3	3	4	4
1	0	1	3	6	8	8								

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8     1 _ 2 3 3 _ 4 _
  3     0 1 3 5 7 8     1 _ 2 3 3 3 4 _
  4     0 1 3 6 7 8     1 _ 2 3 3 3 4 4
  1     0 1 3 6 8 8     1 1 2 3 3 3 4 4
```

```
Contando as frequências: count [ 0 0 2 1 3 2 ]
Calculando as posições/índices: count [ 0 0 2 3 6 2 ]
Distribuindo as chaves:
v[i]    count[v[i]]     aux[count[v[i]]]
        0 1 2 3 4 5     0 1 2 3 4 5 6 7
  2     0 0 2 3 6 8     _ _ 2 _ _ _ _ _
  3     0 0 3 3 6 8     _ _ 2 3 _ _ _ _
  3     0 0 3 4 6 8     _ _ 2 3 3 _ _ _
  4     0 0 3 5 6 8     _ _ 2 3 3 _ 4 _
  1     0 0 3 5 7 8     1 _ 2 3 3 _ 4 _
  3     0 1 3 5 7 8     1 _ 2 3 3 3 4 _
  4     0 1 3 6 7 8     1 _ 2 3 3 3 4 4
  1     0 2 3 6 8 8     1 1 2 3 3 3 4 4

aux -> v [1 1 2 3 3 3 4 4]
```

Contando as frequências: count [0 0 2 1 3 2]
Calculando as posições/índices: count [0 0 2 3 6 2]
Distribuindo as chaves:

count[v[i]]						aux[count[v[i]]]							
0	1	**2**	**3**	4	5	0	1	2	3	4	5	6	7
0	2	3	6	8	8	1	1	2	3	3	3	4	4

count[a] = primeira posição do próximo (a+1)
count[a-1] (anterior) = primeira posição do atual (a)
count[a]-1 = última posição do atual (a)

chave 3: count[2] até count[3]-1

```c
#define MAX 10000
#define R 5

int aux[MAX];
void countingsort(int *v, int l, int r) {
  int i, count[R+1];

  //zerando
  for(i = 0; i <= R; i++) count[i] = 0;

  //frequencias
  for(i = l; i <= r; i++) count[v[i] + 1]++;

  //posições
  for(i = 1; i <= R; i++) count[i] += count[i-1];

  //distribuição
  for(i = l; i <= r; i++) {
    aux[count[v[i]]] = v[i];
    count[v[i]]++;
  }

  //cópia : a partir de aux[0]
  for (i = l; i <= r; i++) v[i] = aux[i-l];
}
```

Roteiro

1. Algoritmos de Ordenação Eficientes
   - Counting Sort
   - Radix Sort

Radix Sort

Ordenação: compara-se as chaves/dados

- Comparar a estrutura das chaves
- Decompondo a chave em subestruturas que a compõe:
  - Números: unidades, dezenas, centenas...
  - Palavras: letras
- A cada iteração/recursão,
  - Comparar somente parte da chave
  - Ordenando parcialmente
- Exemplo:
  - ao procurar/posicionar uma palavra em um dicionário
  - cada letra que forma a palavra
  - contribui para localizar a página exata da palavra
  - cada letra restringe as possibilidades de posições

Radix Sort

170 045 075 090 802 024 002 066

Radix Sort

170 045 075 090 802 024 002 066

170 090 802 002 024 045 075 066

Radix Sort

170 090 802 002 024 045 075 066

Radix Sort

170 090 802 002 024 045 075 066

802 002 024 045 066 170 075 090

Radix Sort

802 002 024 045 066 170 075 090

Radix Sort

802 002 024 045 066 170 075 090

002 024 045 066 075 090 170 802

Radix Sort

170 045 075 090 802 024 002 066

$\Downarrow$

002 024 045 066 075 090 170 802

- Ordena pela unidade, dezena, centena, etc.
- Problema?

Radix Sort

- Envolve operações custosas

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2
- Exemplo com inteiros: partes de 1 byte

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2
- Exemplo com inteiros: partes de 1 byte
  - até 255 $\rightarrow$ 0, 1, 2, 3, ..., 255

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2
- Exemplo com inteiros: partes de 1 byte
  - até 255 $\rightarrow$ 0, 1, 2, 3, ..., 255
  - 257 = 00000001 00000001 $\rightarrow$ 256 + 1

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2
- Exemplo com inteiros: partes de 1 byte
  - até 255 $\rightarrow$ 0, 1, 2, 3, ..., 255
  - 257 = 00000001 00000001 $\rightarrow$ 256 + 1
  - 258 = 00000001 00000010 $\rightarrow$ 256 + 2

Radix Sort

- Envolve operações custosas
  - 802: 802%10, (802/10)%10, (802/100)%10
- Decomposição do número:
  - Decimal: por divisões de potência de 10
  - Binário: por divisões de potência de 2
- Exemplo com inteiros: partes de 1 byte
  - até 255 $\rightarrow$ 0, 1, 2, 3, ..., 255
  - 257 = 00000001 00000001 $\rightarrow$ 256 + 1
  - 258 = 00000001 00000010 $\rightarrow$ 256 + 2
  - 65792 = 00000001 00000001 00000000 $\rightarrow$ 65536 + 256 + 0

Radix Sort

- Radix:
  - ordenar pela a raiz(radix) da representação dos dados
  - extraindo o i-ésimo digito da chave
- Obs.: caso a chave seja pequena não compensa a extração dos bits: use o ***counting sort***

Radix Sort

Para extrair o D-ésimo dígito da chave

```
1 #define bitsbyte 8
2 #define bytesword 4
3
4 //00000001 << 8 = 00000001 00000000 = 2^8 = 256
5 #define R (1 << bitsbyte)
6
7 //extraindo o D-ésimo dígito de N
8 #define digit(N,D) (((N) >> ((D)*bitsbyte)) & (R-1))
9
```

- $((N) >> ((D) * bitsbyte))$: remove os primeiros D bytes
- $\&(R-1)$: pegando os últimos bitbytes (máscara)

Radix Sort

Para extrair o D-ésimo dígito da chave

- $((N) >> ((D) * bitsbyte))$: remove os primeiros D bytes
- $\&(R-1)$: pegando os últimos bitbytes (máscara)

```
1     65792 = 00000001 00000001 00000000
2
3     //pegando o digito 1
4     00000001 00000001 00000000 >> 1*8 =
5     00000000 00000001 00000001
6
7     //aplicando a máscara
8       00000000 00000001 00000001
9     & 00000000 00000000 11111111
10    ----------------------------
11      00000000 00000000 00000001
12
```

- E para strings? como acessar o byte-ésimo digito?

Radix Sort

Métodos de classificação - LSD

- A partir dígito menos significativo (least significant digit – LSD): direita para esquerda
  - Ordena estavelmente chaves de comprimento fixo
    - Tamanho da palavra (word) que representa o dado
    - int: 4 * 8 bits = 32 bits = 4 bytes
    - strings de tamanho W
- Complexidade $\approx 7 * W * N + 3 * W * R$ acessos:
  - N chaves
  - chaves de tamanho W
  - cujo alfabeto são de tamanho R
- R, em geral, é muito menor que N, portanto a complexidade é proporcional a $W * N$

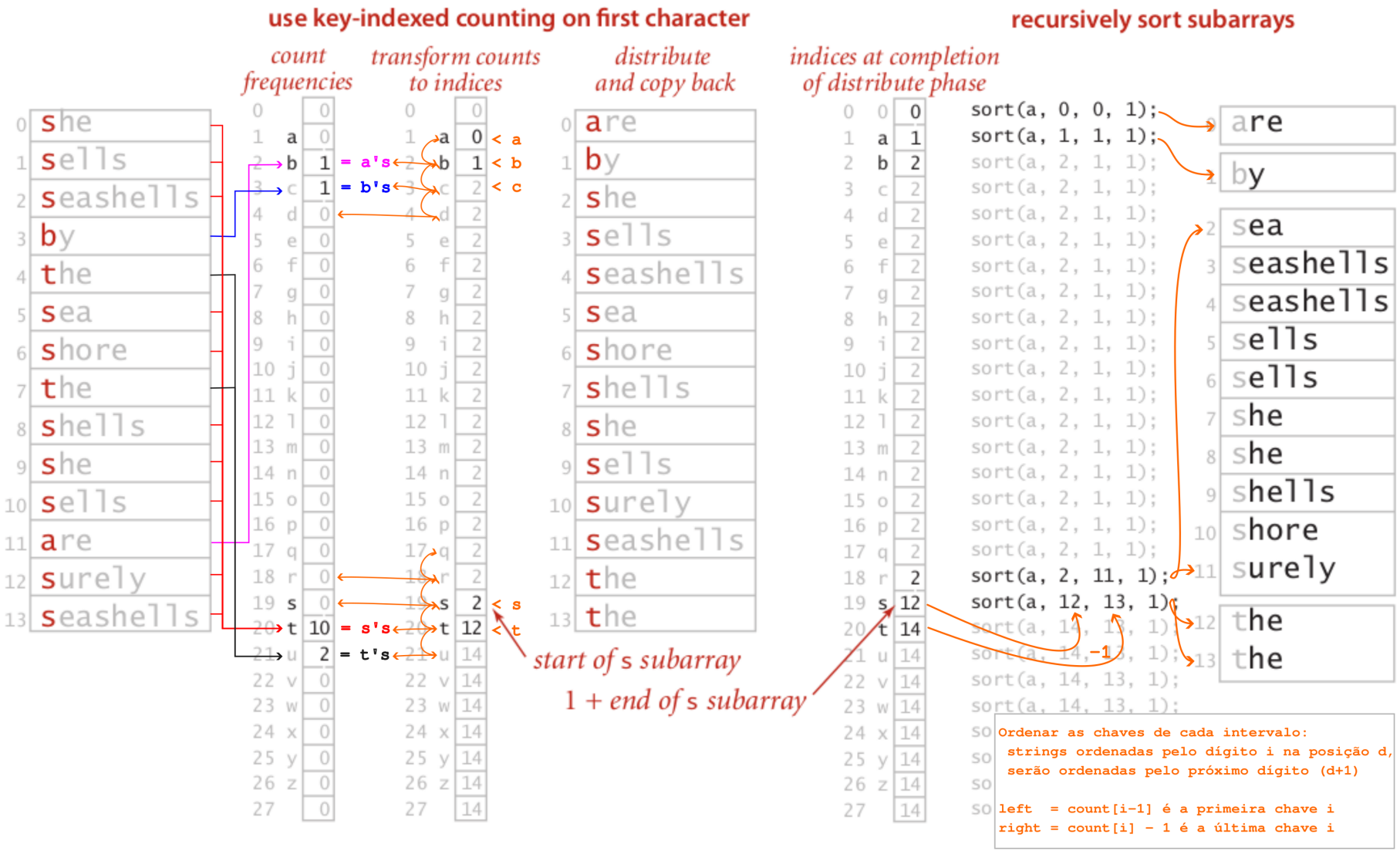
```c
void radix_sortLSD(int *v,  int l, int r) {
  int i, w;
  int aux[r-l+1], count[R+1];

  //w -> deslocamento de bytes
  for(w=0; w<bytesword; w++){
    //for(i=0; i<=R; i++) count[i] = 0;
    memset(count, 0, sizeof(int)*(R+1));

    //frequencias
    for(i=l; i<=r; i++) count[ digit(v[i], w)+1 ]++;

    //posições
    for(i=1; i<=R; i++) count[i] += count[i-1];

    //distruibuição
    for(i=l; i<=r; i++) {
        aux[ count[ digit(v[i], w) ] ] = v[i];
        count[ digit(v[i], w) ]++;
    }

    //copiando : a partir de aux[0]
    for(i=l; i<=r; i++) v[i] = aux[i-l];
  }
}
```

Radix Sort

Métodos de classificação - MSD

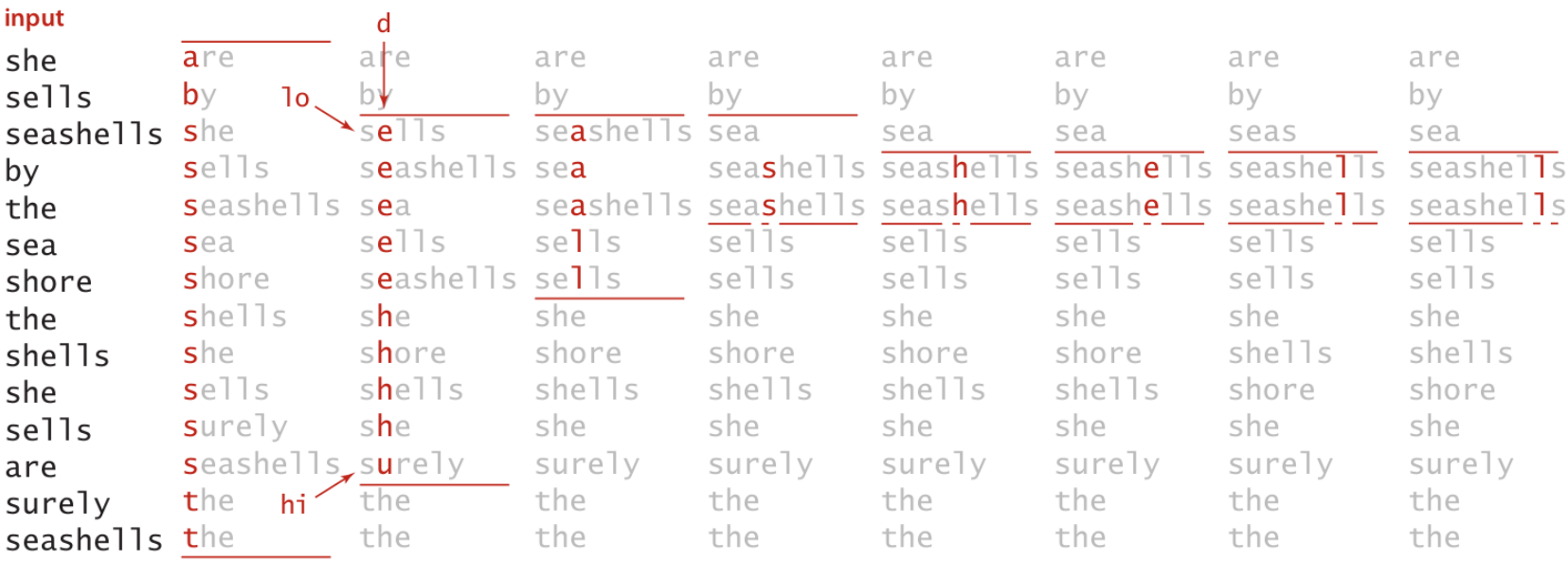
- A partir dígito mais significativo (most significant digit – MSD): esquerda para direita
  - Ordenação de propósito geral: chaves com tamanhos variáveis
    - Conjunto de várias palavras
    - string: N * 8 bits = N * 1 byte, N variável
  - Usa-se primeiro o counting sort (key-indexed counting)
  - Depois, recursivamente, ordena-se os sub-vetores de cada caractere
- Complexidade $\approx 7 * W * N + 3 * W * R$ acessos:

Radix Sort - MSD - strings

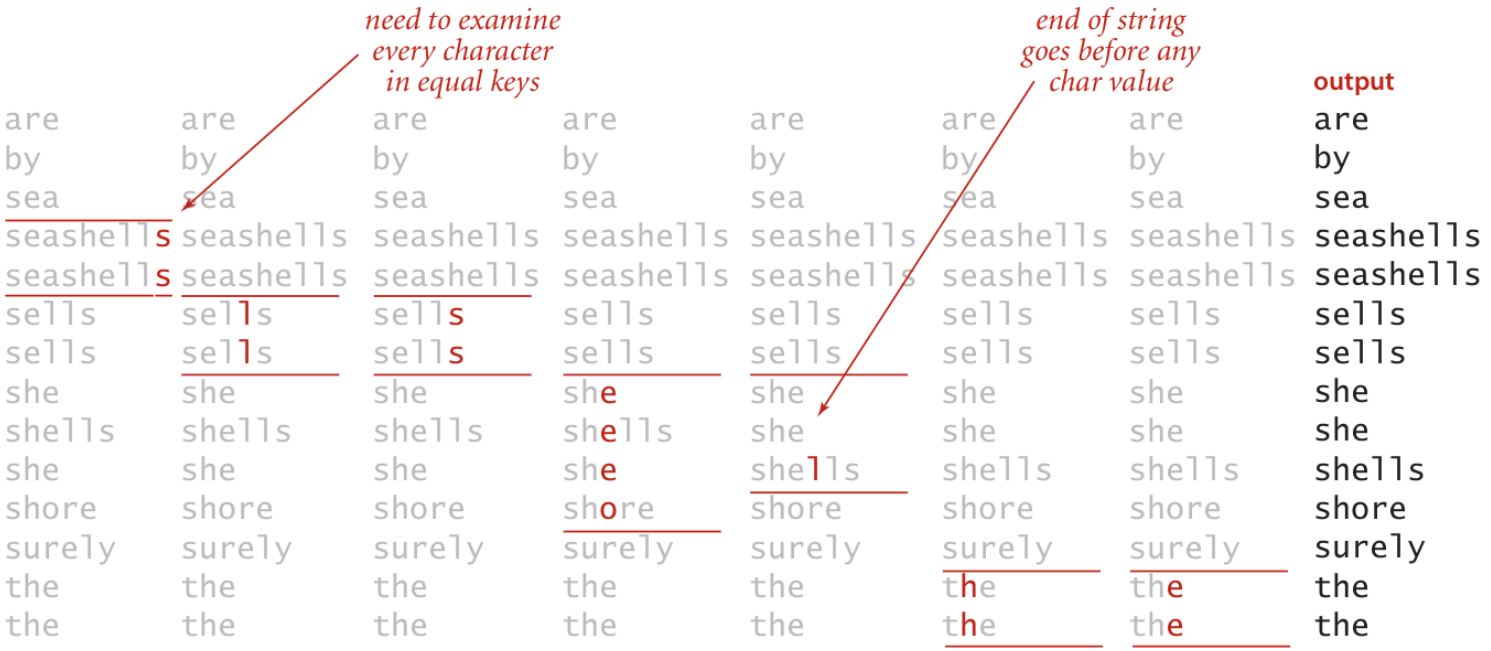


Trace of MSD string sort: top level of `sort(a, 0, 14, 0)`

Radix Sort - MSD - strings



Radix Sort - MSD - strings



Radix Sort - MSD

```
1  typedef char Item;
2  #define maxstring 101
3  #define bitsbyte 8
4  #define R (1 << bitsbyte) //00000001<<8 = 2^8 = 256
5
6  int charAt(char s[], int d) {
7    if (d < strlen(s))
8      return s[d]; //d-ésimo caractere
9    else
10     return -1; //count[-1 + 1] = count[0]
11                //count[0] = qtde. de palavras menores que d
12 }
13
14 //Strings: ordena para o d-ésimo caractere
15 void radixMSD(char a[][maxstring], int l, int r, int d) {
16   if(r<=l) return;
17
18   char aux[r-l+1][maxstring];
19
20   int count[R+1];
21   memset(count, 0, sizeof(int)*(R+1));
22   //for(int i=0; i<=R; i++) count[i] = 0;
23
```

Radix Sort - MSD

```
24    //frequencia dos d-ésimos caracteres
25    for (int i = l; i <= r; i++)
26        count[charAt(a[i], d) + 1]++;
27
28    //tabela de índices: calculando as posições
29    for (int i = 1; i<=R; i++)
30        count[i] += count[i-1];
31
32    //redistribui as chaves: ordena em aux
33    for (int i = l; i <= r; i++)
34        strcpy(aux[count[charAt(a[i], d)]++], a[i]);
35
36    //copia para o original
37    for (int i = l; i <= r; i++)
38        strcpy(a[i], aux[i - l]);
39
40    //ordenar por subconjunto : count[0] = já ordenadas
41    for (int i = 1; i <= R; i++) {
42      //count[i-1] posição da primeira chave com o caractere i
43      //count[i]-1 posição da última chave com o caractere i
44      radixMSD(a, l + count[i-1], l + count[i]-1, d+1);
45    }
46  }
```

Radix Sort - MSD

```
47 //Inteiros: ordena o w-ésimo byte
48 void radix_sortMSD(int *v,  int l, int r, int w) {
49     if(r<=l || w < 0) return;
50     int i, aux[r-l+1], count[R+1];
51     memset(count, 0, sizeof(int)*(R+1));
52 
53     for(i=l; i<=r; i++)
54         count[digit(v[i], w)+1]++;
55 
56     for(i=1; i<=R; i++)
57         count[i] += count[i-1];
58 
59     for(i=l; i<=r; i++)
60         aux[count[digit(v[i], w)]++] = v[i];
61 
62     for(i=l; i<=r; i++) v[i] = aux[i-l];
63 
64     //ordenando os que começam por zero
65     radix_sortMSD(v, l, l + count[0]-1, w-1);
66     for (i = 1; i <= R; i++)
67       radix_sortMSD(v, l + count[i-1], l + count[i]-1, w-1);
68 }
```

Radix Sort

Métodos de ordenação

algorithm	stable?	inplace?	order of growth of typical number calls to charAt() to sort N strings from an R-character alphabet (average length w, max length W)		sweet spot
			running time	extra space	
insertion sort for strings	yes	yes	between N and N^2	1	small arrays, arrays in order
quicksort	no	yes	$N \log^2 N$	$\log N$	general-purpose when space is tight
mergesort	yes	no	$N \log^2 N$	N	general-purpose stable sort
3-way quicksort	no	yes	between N and $N \log N$	$\log N$	large numbers of equal keys
LSD string sort	yes	no	NW	N	short fixed-length strings
MSD string sort	yes	no	between N and Nw	$N + WR$	random strings