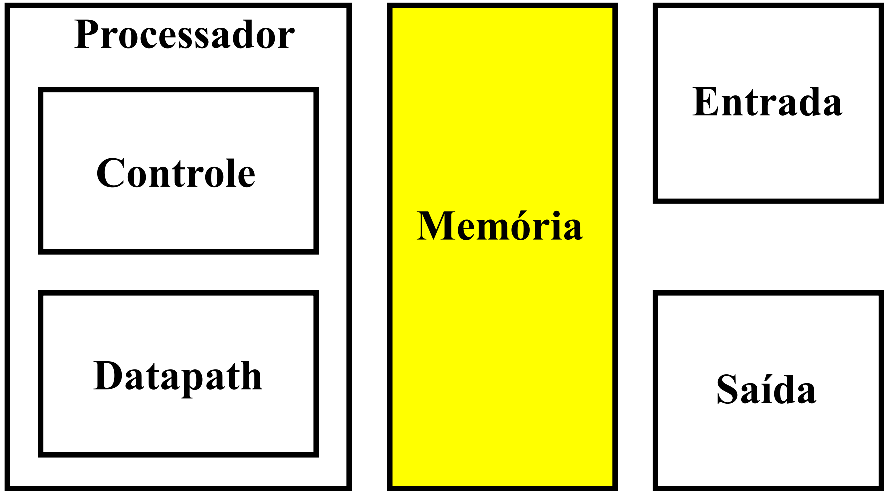
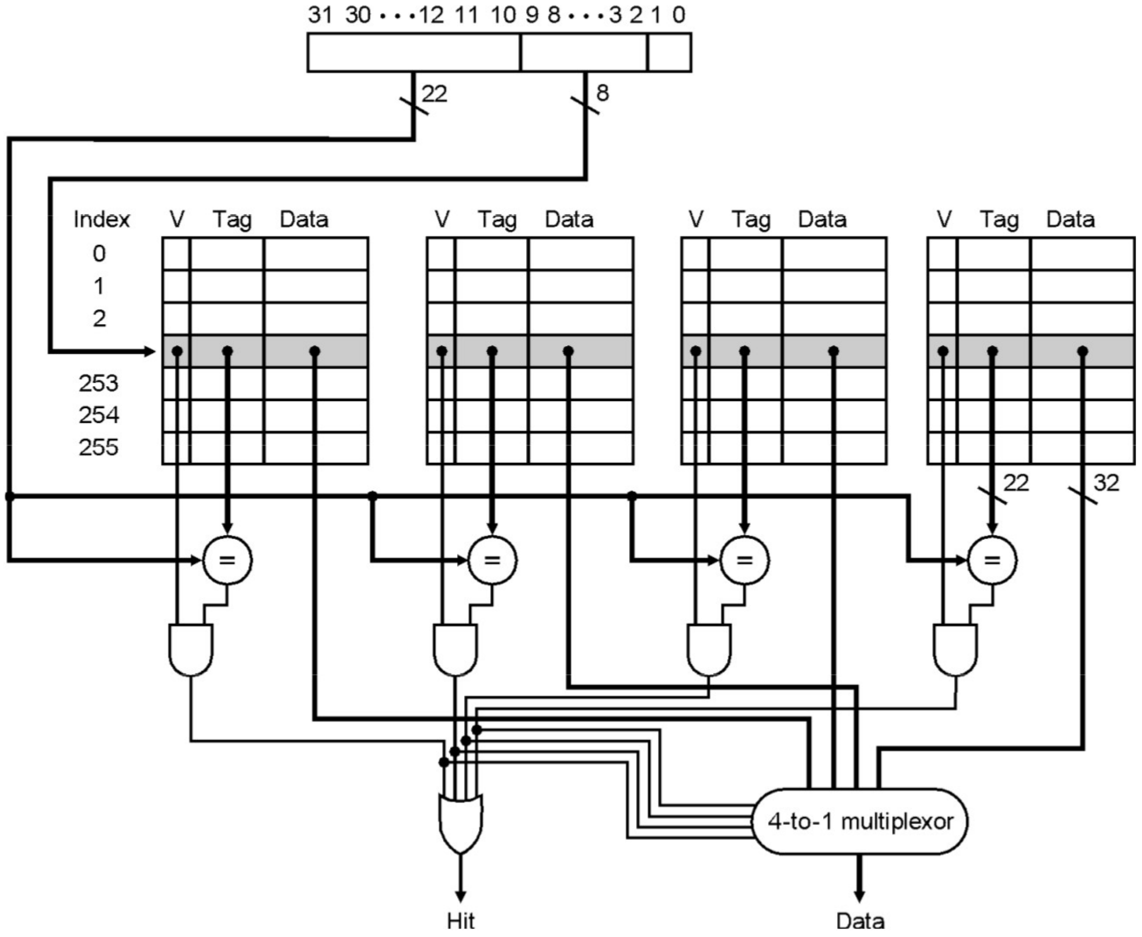
Cache: Associatividade e múltiplos níveis



Organização de uma cache n-way

TAG	ÍNDICE	SEL. NO BLOCO



Para tamanho fixo:

Associatividade ↑

blocos p/ conjunto ↑

num. de conjuntos ↓

Associatividade ↑

tamanho do índice ↓

tamanho do tag ↑

Tag versus associatividade

- **Hipótese:**
  - Cache com 4K blocos
  - 4 palavras/bloco
  - Endereço de 32 bits
- **Quatro cenários de mapeamento:**
  - Direto, 2-way, 4-way e totalmente associativa
- **Tag + índice + offset**
  - 4 palavras/bloco = 2^4 bytes/bloco
  - Restam 32 – 4 = 28 bits (tag + índice)

Tag versus associatividade

- **Número de conjuntos**
  - Direto: 4K conjuntos (índice = 12)
  - 2-way: 2K conjuntos (índice = 11)
  - 4-way: 1K conjuntos (índice = 10)
  - T.A.: 1 conjunto (índice = 0)
- **Número de bits gastos com tag**
  - Direto: (28-12) x 1 x 4K = 64Kbits
  - 2-way: (28-11) x 2 x 2K = 68Kbits
  - 4-way: (28-10) x 4 x 1K = 72Kbits
  - T.A.: (28-0) x 4K x 1 = 112Kbits

Política de atualização da cache

- **Precisa-se armazenar um dado na cache...**
  - **Mas nenhuma posição livre.**
  - **Qual bloco substituir ?**
- **Mapeamento direto:**
  - **Bloco ocupando posição mapeada <u>precisa</u> ser substituído**
- **Cache n-way:**
  - **Critério de escolha mais popular**
    - » **Bloco não usado há mais tempo é substituído**
    - » **LRU (“least recently used”)**

Impacto da cache no desempenho

$$\text{ciclos}_{\text{stall}} = \frac{\text{acessos}}{\text{programa}} \times \text{mr} \times \text{penalidade}$$

- **Sejam:**
  - **I = número de instruções;**
  - **LS = proporção de load + store no "mix";**
  - **acessos/programa = I + LS × I = I (1+ LS);**
  - **CPI_{stall} = $ciclos_{stall}$ / I;**
  - **CPI = CPI_{ideal} + CPI_{stall}**
  - **Supondo CPI_{ideal} = 1:**

$$\text{CPI} = 1 + (1 + \text{LS}) \times \text{mr} \times \text{penalidade}$$

Impacto da cache no desempenho

- **Dado um tamanho fixo de cache...**
  - **Mais palavras por bloco:**
    - » **Maior captura da localidade espacial (↓mr)**
  - **Mais associatividade:**
    - » **Maior captura da localidade temporal (↓mr)**
  - **Nenhum impacto na penalidade !?**

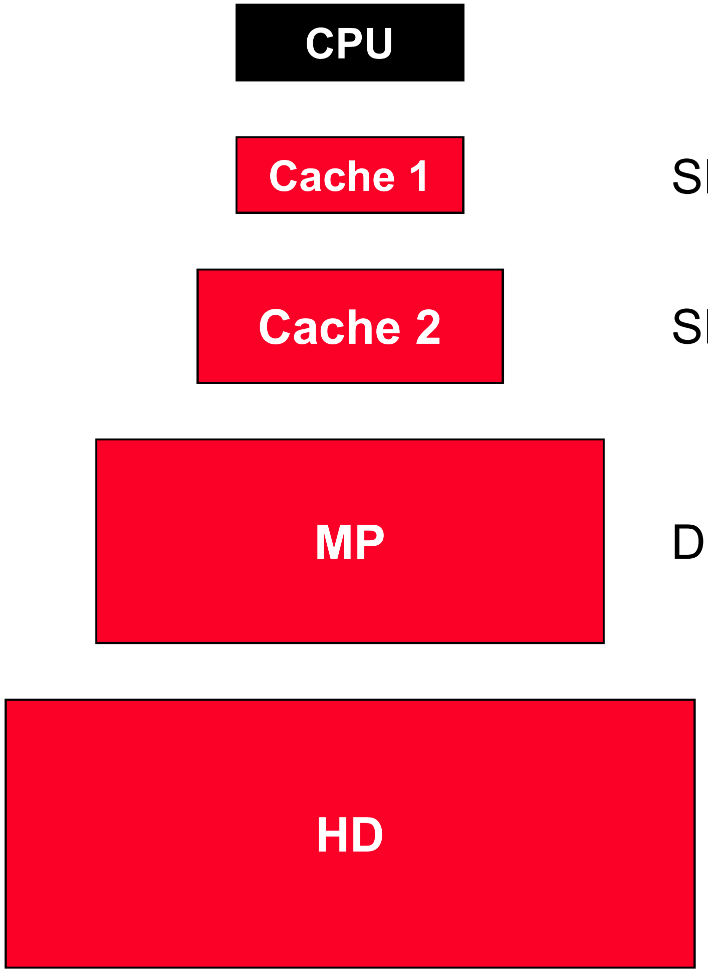
$$\mathrm{CPI} = 1 + (1 + \mathrm{LS}) \times \mathrm{mr} \times \mathrm{penalidade}$$

Impacto da cache no desempenho

- **Tendências tecnológicas até 2005:**
  - **f_r crescia rapidamente, mas f_{acesso} (DRAM) lentamente.**
- **Problema:**
  - **Penalidade de falta tendia a aumentar.**
- **Solução: dois níveis de cache (L1 e L2)**
  - **Quando comparada a cache única de mesma capacidade...**
  - **Objetivo 1: reduzir tempo de acerto e minimizar T**
    - » **L1: precisa ser pequena para ser rápida**
    - » **L1: maior taxa de faltas, mas penalidade compensada por L2**
  - **Objetivo 2: reduzir penalidade e taxa <u>global</u> de faltas**
    - » **L2: pode ser maior, pois não afeta tempo de acerto nem T**
    - » **L2: menor taxa de faltas local, mas oferece menor penalidade**

$$\mathrm{CPI} = 1 + (1 + \mathrm{LS}) \times \mathrm{mr} \times \mathrm{penalidade}$$

Redução da penalidade



SRAM (on-chip)

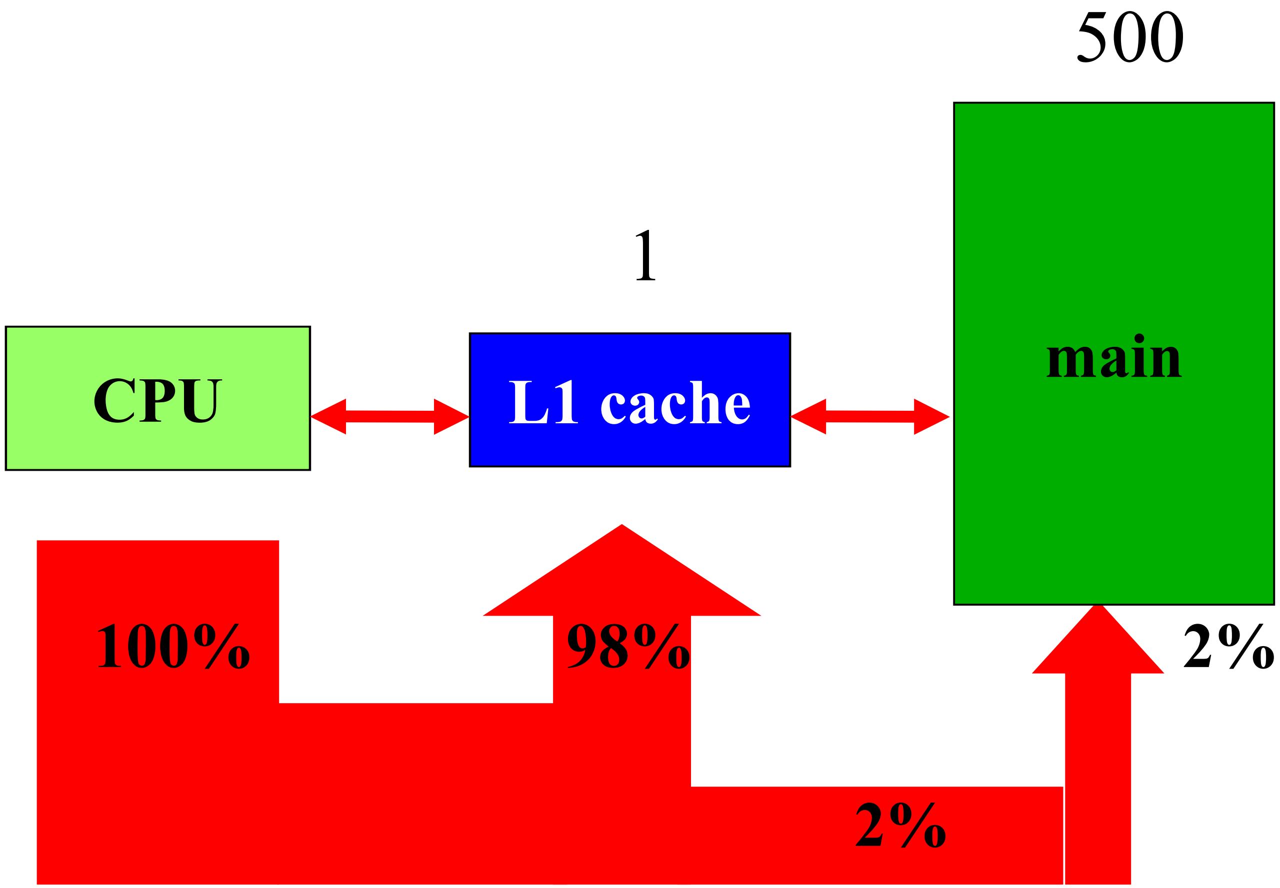
SRAM (on-chip, off-chip)

DRAM (on-board)

Exemplo

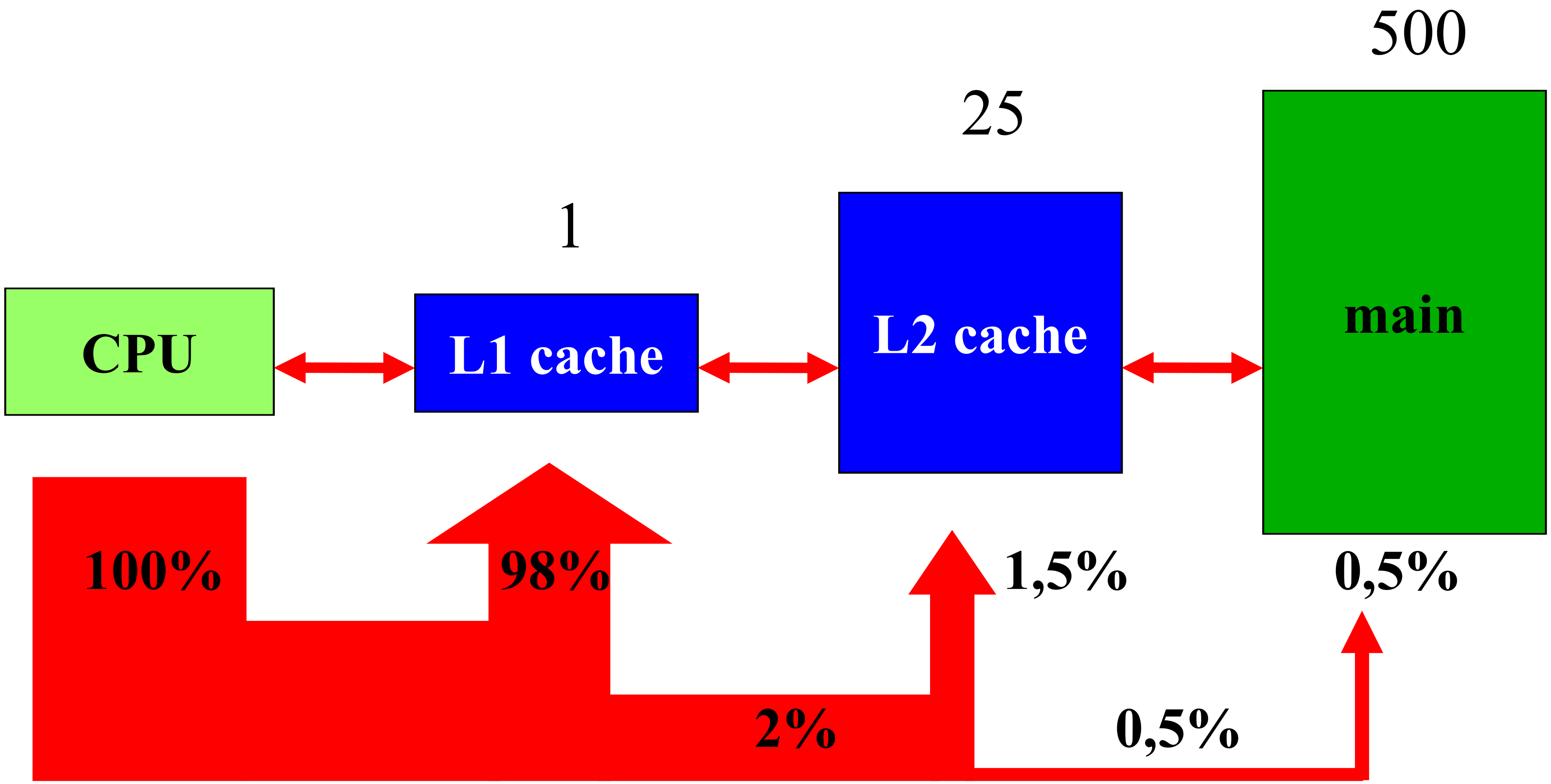
- Impacto da cache secundária
  - f = 5GHz (T = 0,2ns)
  - Primária: 0,2ns, taxa de faltas = 2%
  - Secundária: 5ns, taxa <u>global</u> de faltas = 0,5%
  - Memória principal: 100ns
- Hipóteses:
  - Só acesso a instruções;
  - CPI_{ideal} = 1.
- Comparação:
  - Qual o CPI só com a cache primária ?
  - Qual o CPI para dois níveis de cache ?

Uma única cache



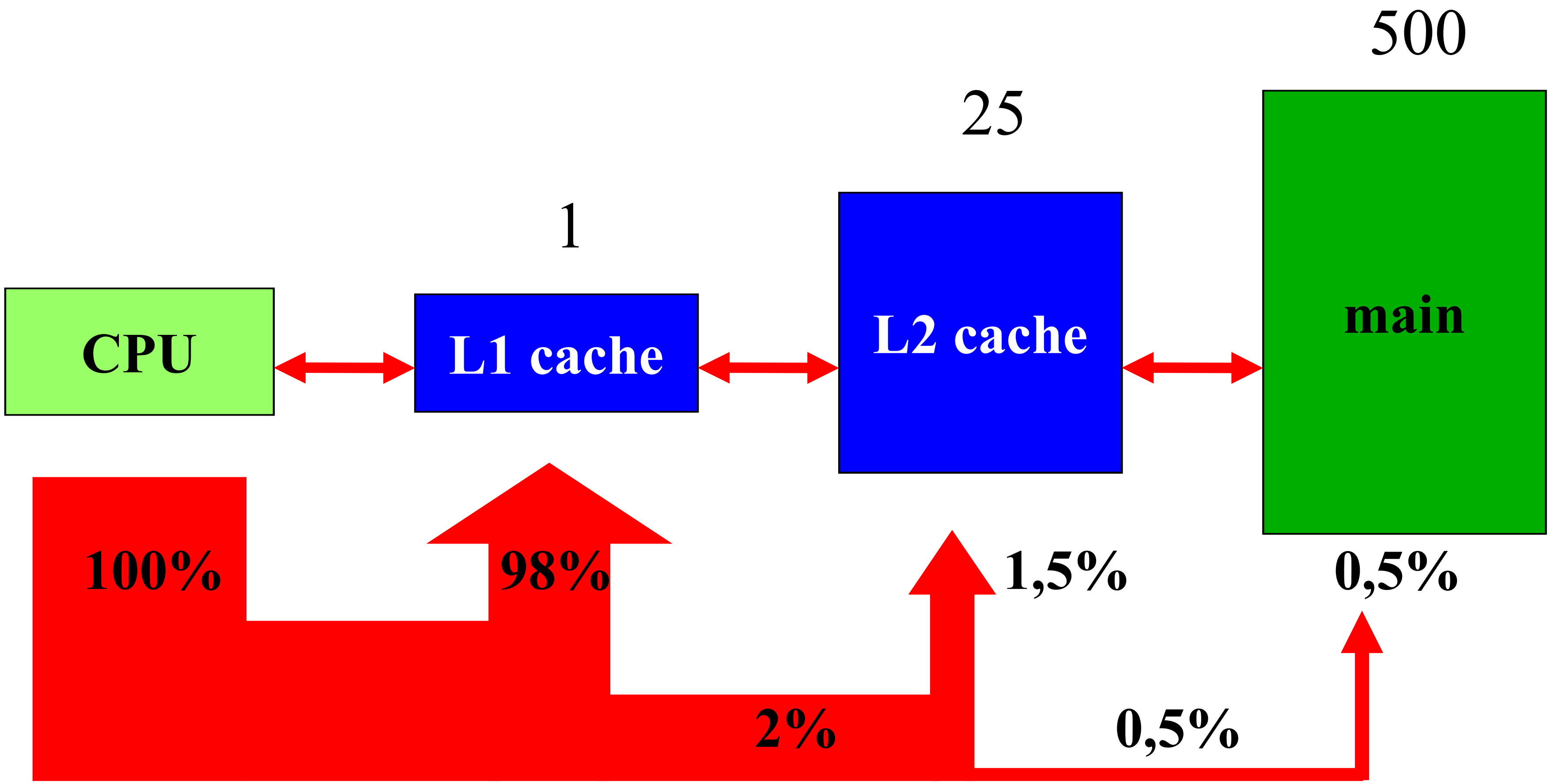
$$CPI = 1 + 0{,}02 \times 500 = 11$$

Dois níveis de cache



$$CPI = 1 + 0{,}02 \times 25 + 0{,}005 \times 500 = 4{,}0$$

Dois níveis de cache



$$CPI = 1 + 0{,}015 \times 25 + 0{,}005 \times 525 = 4{,}0$$

Um panorama contemporâneo

Nehalem:

L1=32KB+32KB

L2=256KB

L3=8MB

Servers

(e.g. Intel Xeon, AMD Opteron)

Barcelona:

L1=64KB+64KB

L2=512KB

L3=2MB

A15:

L1=32KB+32KB

L2= up to 4MB

R7:

L1=64KB+64KB

SoC

(e.g. Qualcomm Snapdragon,

Samsumg Exynos)

M1:

SRAM=1MB+1MB

(sem cache)

MCUs

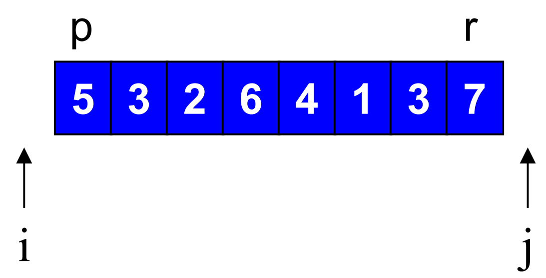
(e.g. ARM M1)

Impacto da cache no desempenho

- **Dois programas de ordenação**
  - **Diferentes números de itens a ordenar (4K a 4M)**
- **Algoritmos com complexidades distintas**
  - **QuickSort: O(n log n) [média]**
  - **RadixSort: O(n) [média]***
- **Complexidade:**
  - **Comportamento assintótico**
    - » **Número de operações como função da entrada (n)**
    - » **Para um n grande**
- **Desempenho**
  - **Número de instruções e número de ciclos**

Revisitando QUICKSORT

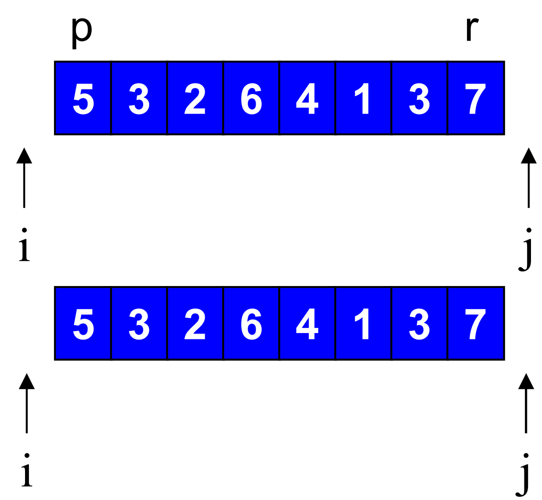
```
QUICKSORT(A,p,r)
if p< r
   then     q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
   do       repeat    j ← j – 1
                      until A[j] ≤ x
            repeat    i ← i + 1
                      until A[i] ≥ x
            if i < j
                      then exchange A[i] ↔A[j]
                      else return j
```



(Fonte: T. Cormen, C. Leiserson, R. Rivest, "Introduction to Algorithms", MIT Press.)

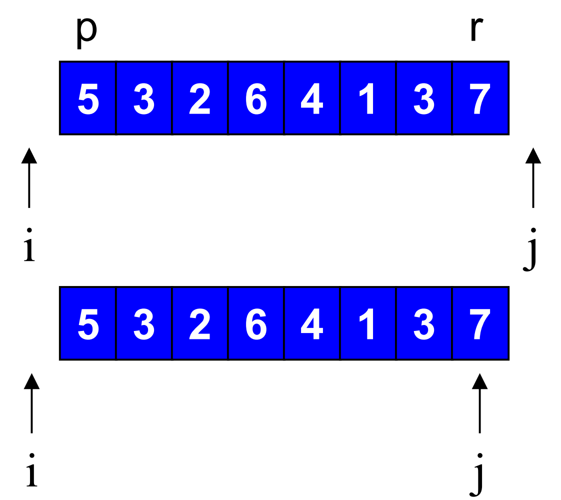
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat      j ← j – 1
                        until A[j] ≤ x
            repeat      i ← i + 1
                        until A[i] ≥ x
            if i < j
                        then exchange A[i] ↔A[j]
                        else return j
```



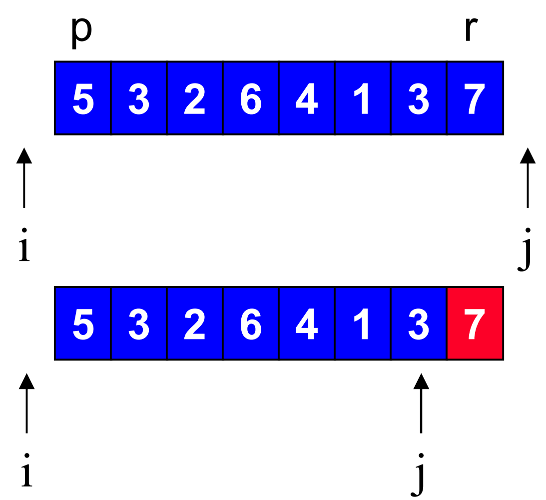
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
   then    q ←PARTITION(A,p,r)
           QUICKSORT(A,p,q)
           QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
   do      repeat     j ← j – 1
                      until A[j] ≤ x
           repeat     i ← i + 1
                      until A[i] ≥ x
           if i < j
                      then exchange A[i] ↔A[j]
                      else return j
```



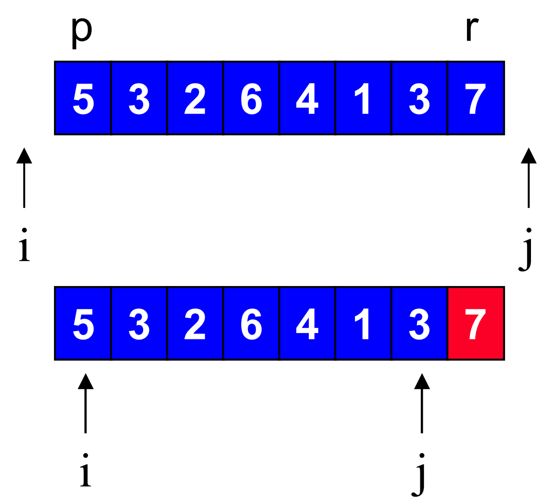
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat      j ← j – 1
                        until A[j] ≤ x
            repeat      i ← i + 1
                        until A[i] ≥ x
            if i < j
                        then exchange A[i] ↔A[j]
                        else return j
```



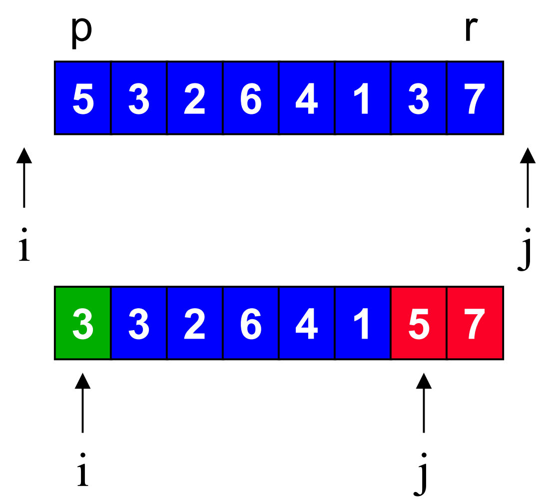
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat      j ← j – 1
                        until A[j] ≤ x
             repeat     i ← i + 1
                        until A[i] ≥ x
            if i < j
                        then exchange A[i] ↔A[j]
                        else return j
```



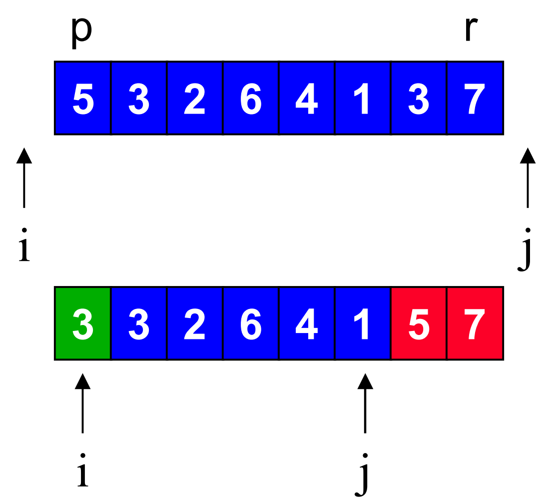
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat      j ← j – 1
                        until A[j] ≤ x
             repeat     i ← i + 1
                        until A[i] ≥ x
            if i < j
                        then exchange A[i] ↔A[j]
                        else return j
```



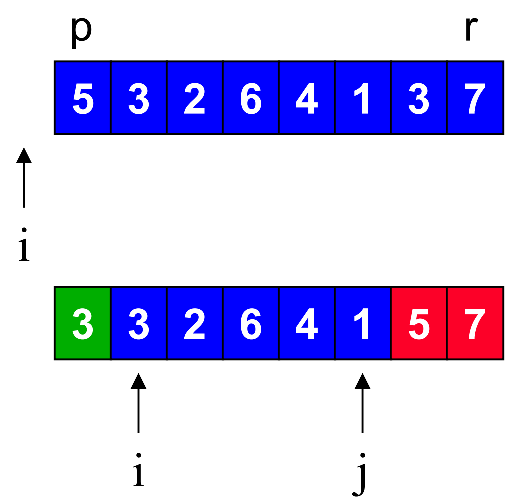
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat    j ← j – 1
                      until A[j] ≤ x
             repeat   i ← i + 1
                      until A[i] ≥ x
            if i < j
                      then exchange A[i] ↔A[j]
                      else return j
```



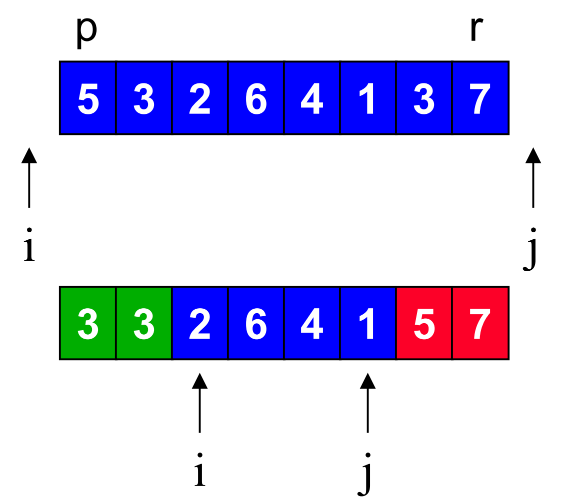
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat      j ← j – 1
                        until A[j] ≤ x
             repeat     i ← i + 1
                        until A[i] ≥ x
            if i < j
                        then exchange A[i] ↔A[j]
                        else return j
```



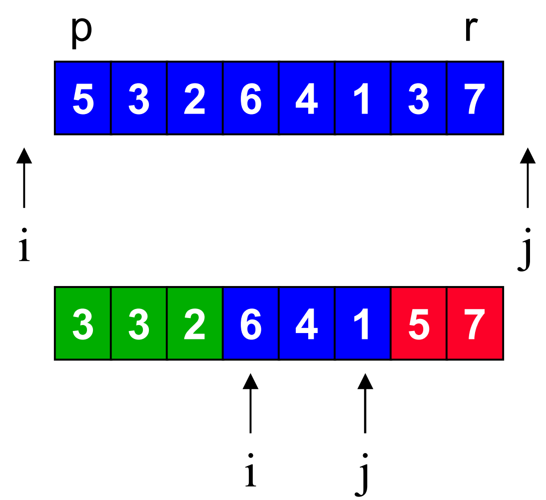
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat    j ← j – 1
                      until A[j] ≤ x
            repeat    i ← i + 1
                      until A[i] ≥ x
            if i < j
                      then exchange A[i] ↔A[j]
                      else return j
```



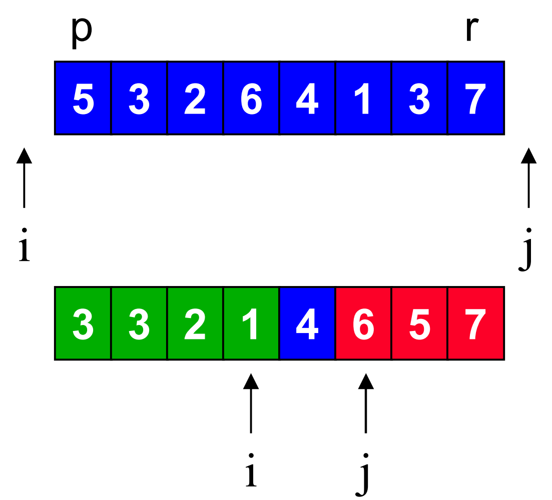
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat     j ← j – 1
                       until A[j] ≤ x
            repeat     i ← i + 1
                       until A[i] ≥ x
            if i < j
                       then exchange A[i] ↔A[j]
                       else return j
```



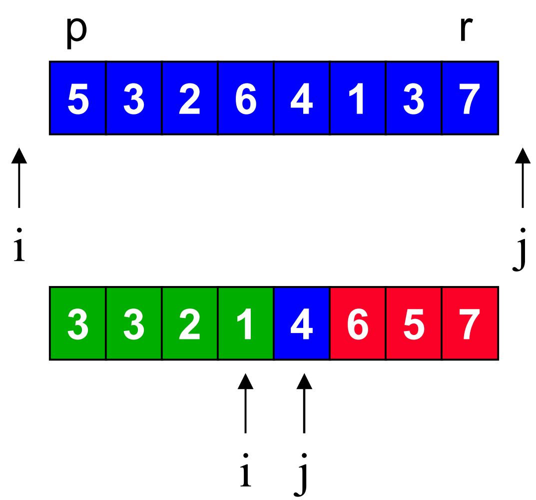
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat    j ← j – 1
                      until A[j] ≤ x
             repeat   i ← i + 1
                      until A[i] ≥ x
            if i < j
                      then exchange A[i] ↔A[j]
                      else return j
```



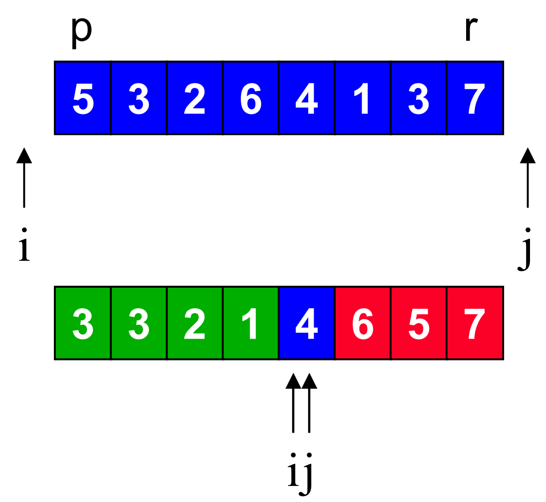
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
   then    q ←PARTITION(A,p,r)
           QUICKSORT(A,p,q)
           QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
   do     repeat    j ← j – 1
                    until A[j] ≤ x
          repeat    i ← i + 1
                    until A[i] ≥ x
          if i < j
                    then exchange A[i] ↔A[j]
                    else return j
```



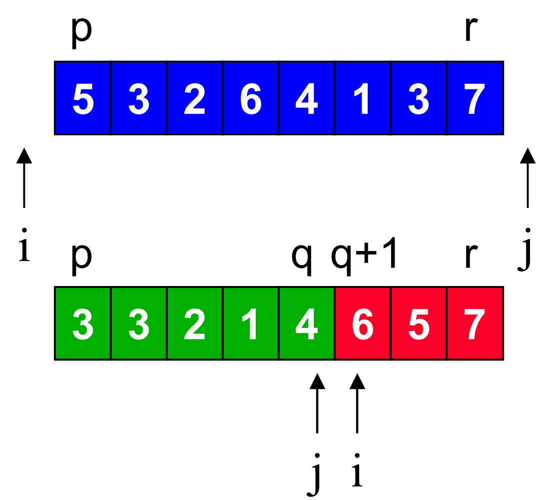
Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
   then    q ←PARTITION(A,p,r)
           QUICKSORT(A,p,q)
           QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
   do      repeat    j ← j – 1
                     until A[j] ≤ x
           repeat    i ← i + 1
                     until A[i] ≥ x
           if i < j
                     then exchange A[i] ↔A[j]
                     else return j
```



Revisitando QUICKSORT

```
QUICKSORT(A,p,r)
if p< r
    then    q ←PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x ← A[p]
i ← p -1
j ← r +1
while TRUE
    do      repeat     j ← j – 1
                       until A[j] ≤ x
            repeat     i ← i + 1
                       until A[i] ≥ x
            if i < j
                       then exchange A[i] ↔A[j]
                       else return j
```



Localidade temporal (instruções)
Grande: recursividade e laços

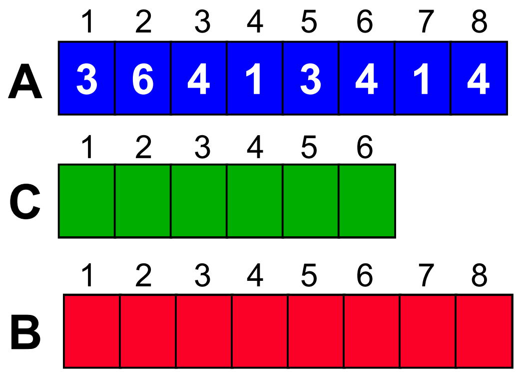
Localidade espacial (instruções)
Pouca: BBs pequenos

Localidade espacial (dados)
Grande: i e j percorrem arranjo sequencialmente

Localidade temporal (dados)
Grande: partições são revisitadas

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



Radix Sort requer método auxiliar de ordenação **estável**. Por exemplo: *Counting Sort*

Hipótese de *Counting Sort:* Elementos a ordenar são **inteiros** no intervalo de **1 a k**

(Fonte: T. Cormen, C. Leiserson, R. Rivest, “Introduction to Algorithms”, MIT Press.)

Estado inicial

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
B								

C inicializado

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	0	0	1	0	0	0

	1	2	3	4	5	6	7	8
B								

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

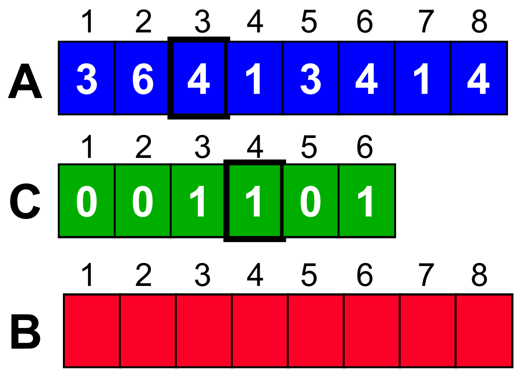
	1	2	3	4	5	6	7	8
A	3	$\boxed{6}$	4	1	3	4	1	4

	1	2	3	4	5	6
C	0	0	1	0	0	$\boxed{1}$

	1	2	3	4	5	6	7	8
B								

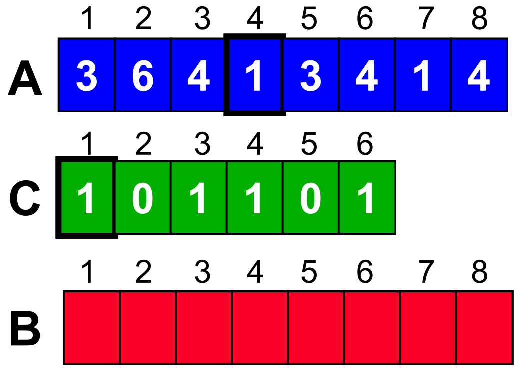
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



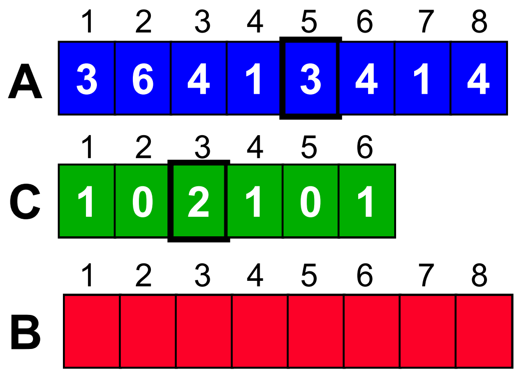
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



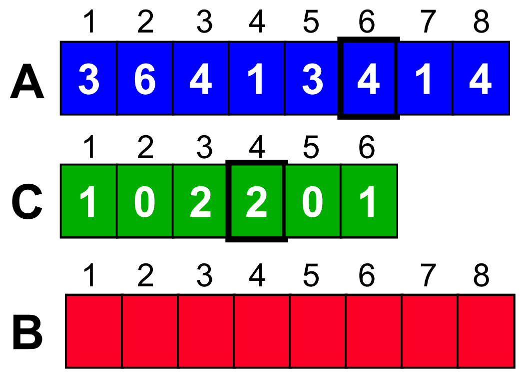
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



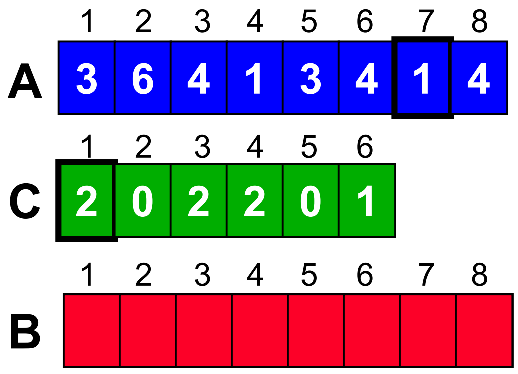
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



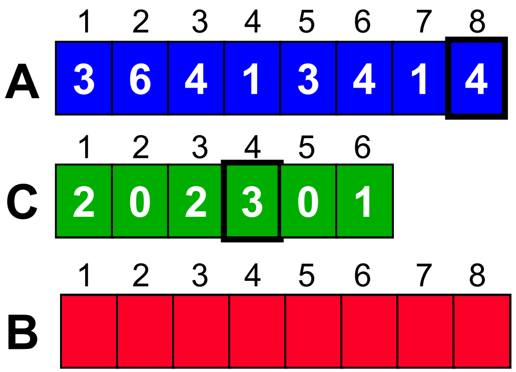
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



Revisitando RADIX-SORT

COUNTING-SORT(A,B,k)
for i ← 1 **to** k
    do C[i] ← 0
for j ← 1 **to** length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 **to** k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] **downto** 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1



Revisitando RADIX-SORT

COUNTING-SORT(A,B,k)
for i ← 1 **to** k
  do C[i] ← 0
for j ← 1 **to** length[A]
  do C[A[j]] ← C[A[j]] + 1
for i ← 2 **to** k
  do C[i] ← C[i] + C[i-1]
for j ←length[A] **downto** 1
  do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

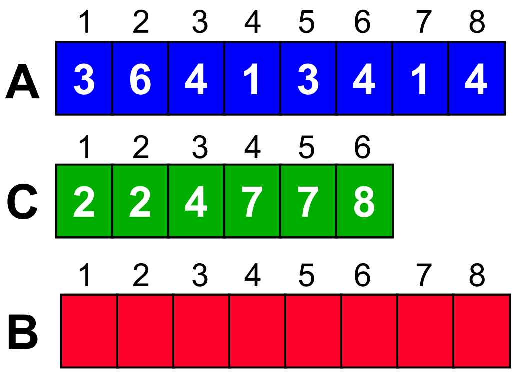
	1	2	3	4	5	6
C	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B								

C[i] contém o número de elementos iguais a i

Revisitando RADIX-SORT

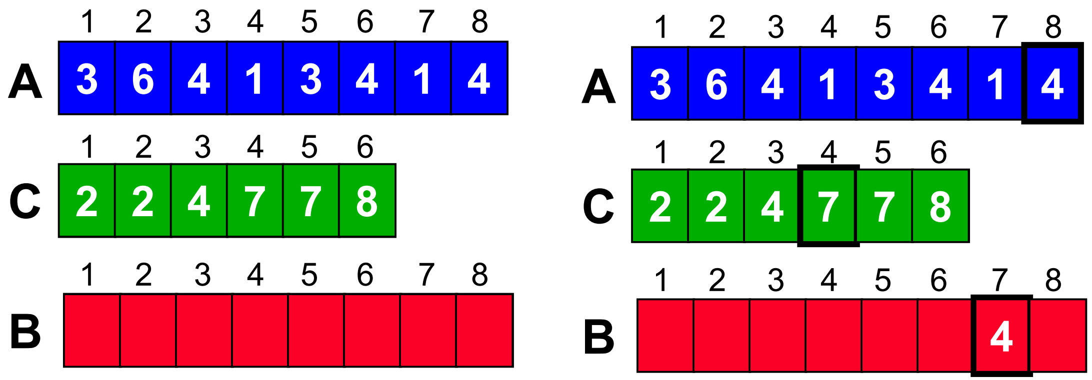
```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



C[i] contém o número de elementos iguais ou menores a i

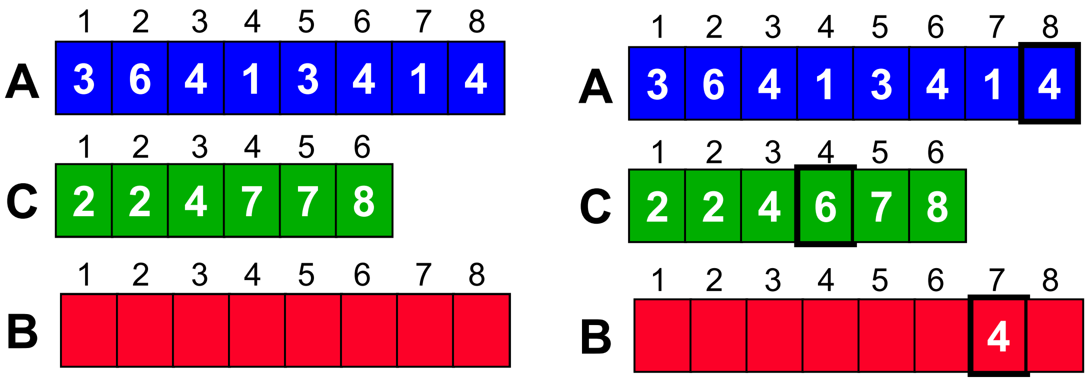
Revisitando RADIX-SORT

COUNTING-SORT(A,B,k)
for i ← 1 **to** k
    do C[i] ← 0
for j ← 1 **to** length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 **to** k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] **downto** 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1



Revisitando RADIX-SORT

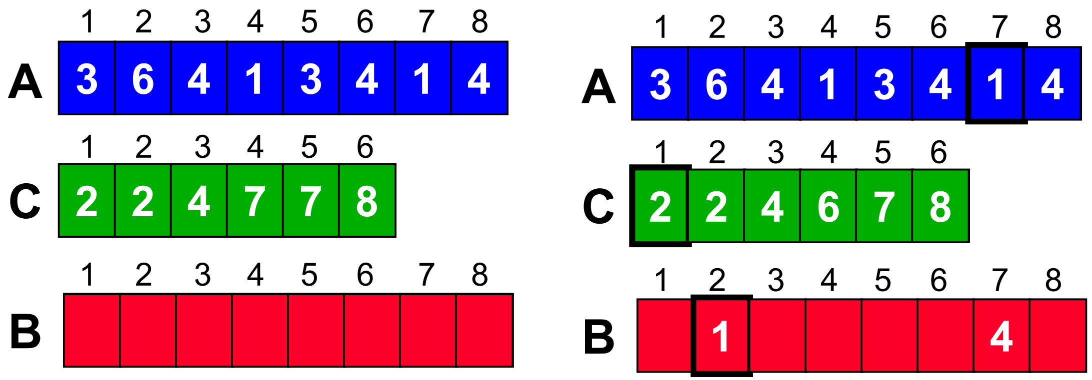
```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



Ao final da primeira iteração do último laço

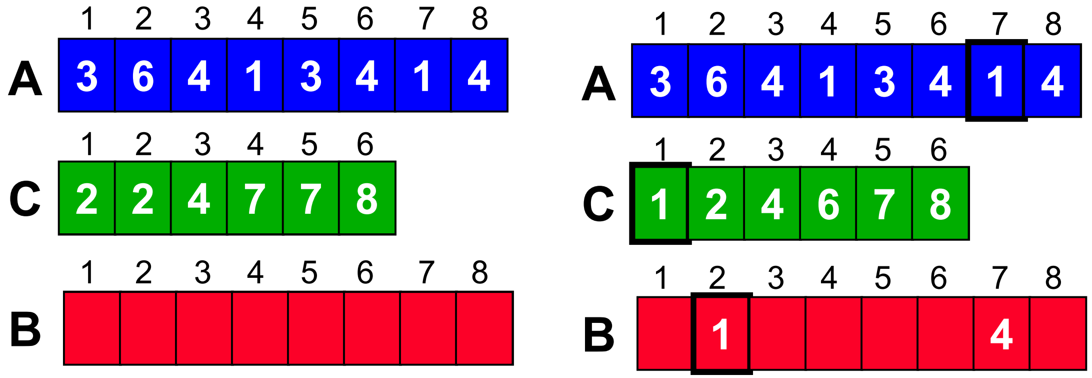
Revisitando RADIX-SORT

COUNTING-SORT(A,B,k)
for i ← 1 **to** k
    do C[i] ← 0
for j ← 1 **to** length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 **to** k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] **downto** 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1



Revisitando RADIX-SORT

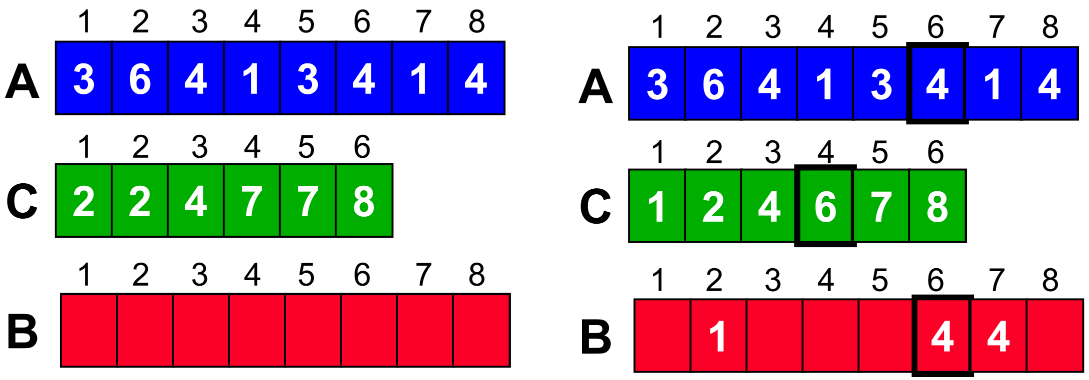
```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



Ao final da segunda iteração do último laço

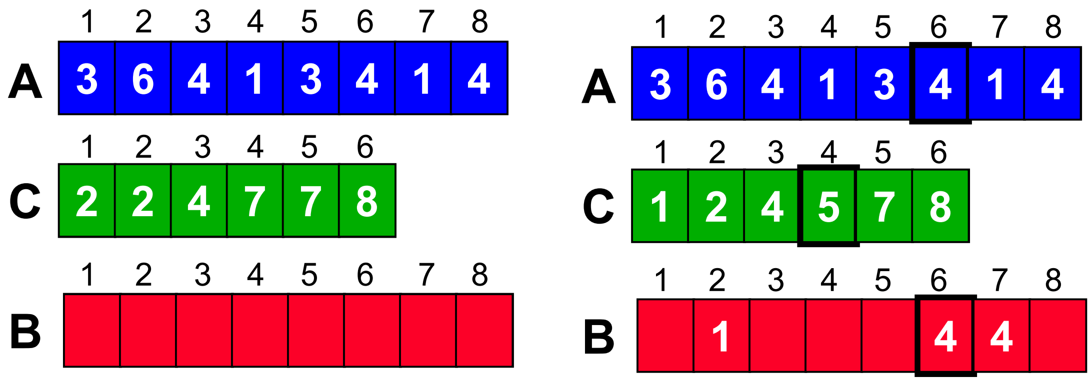
Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



Revisitando RADIX-SORT

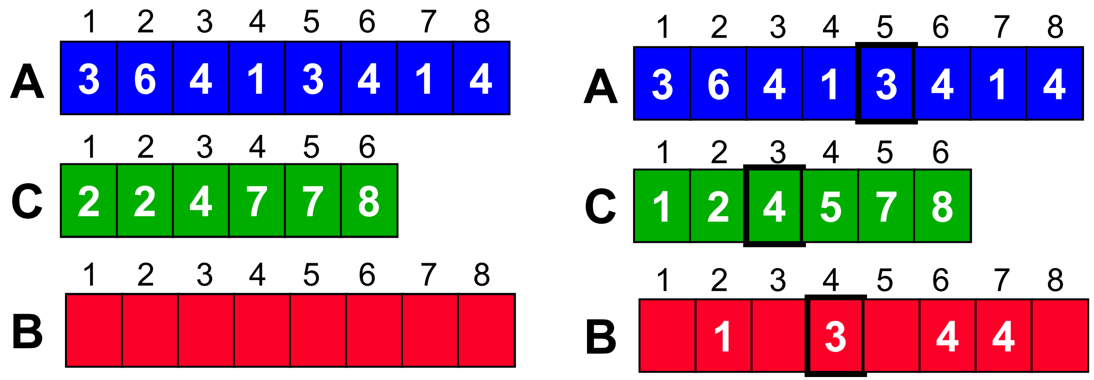
```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



Ao final da terceira iteração do último laço

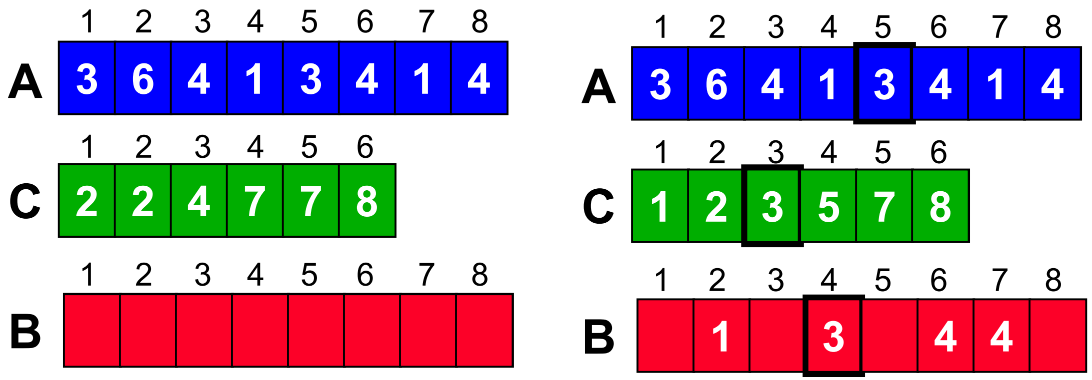
Revisitando RADIX-SORT

COUNTING-SORT(A,B,k)
for i ← 1 **to** k
    do C[i] ← 0
for j ← 1 **to** length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 **to** k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] **downto** 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1



Revisitando RADIX-SORT

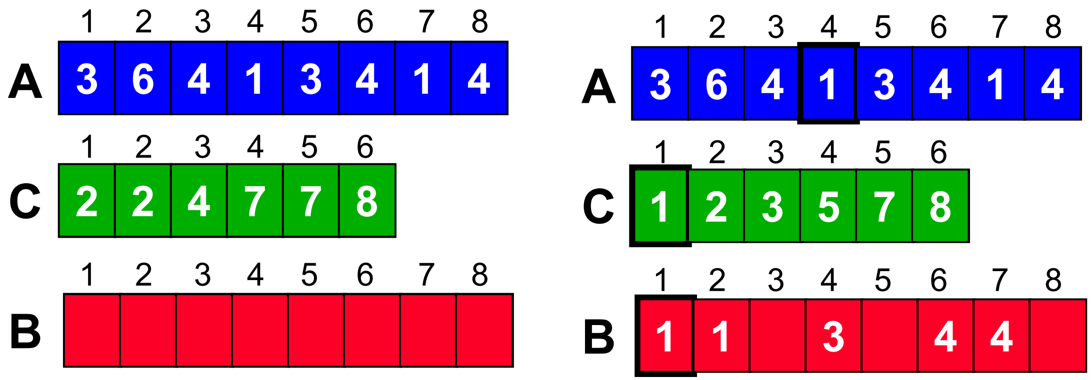
```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



Ao final da quarta iteração do último laço

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	4	$\boxed{1}$	3	4	1	4

	1	2	3	4	5	6
C	$\boxed{0}$	2	3	5	7	8

	1	2	3	4	5	6	7	8
B	$\boxed{1}$	1		3		4	4	

Ao final da quinta iteração do último laço

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	$\boxed{4}$	1	3	4	1	4

	1	2	3	4	5	6
C	0	2	3	$\boxed{5}$	7	8

	1	2	3	4	5	6	7	8
B	1	1		3	$\boxed{4}$	4	4	

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	$\boxed{4}$	1	3	4	1	4

	1	2	3	4	5	6
C	0	2	3	$\boxed{4}$	7	8

	1	2	3	4	5	6	7	8
B	1	1		3	$\boxed{4}$	4	4	

Ao final da sexta iteração do último laço

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	0	2	3	4	7	8

	1	2	3	4	5	6	7	8
B	1	1		3	4	4	4	6

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

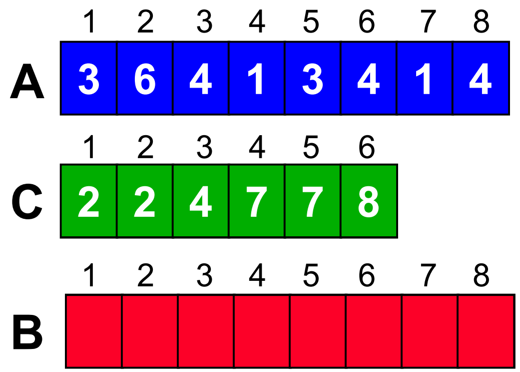
	1	2	3	4	5	6
C	0	2	3	4	7	7

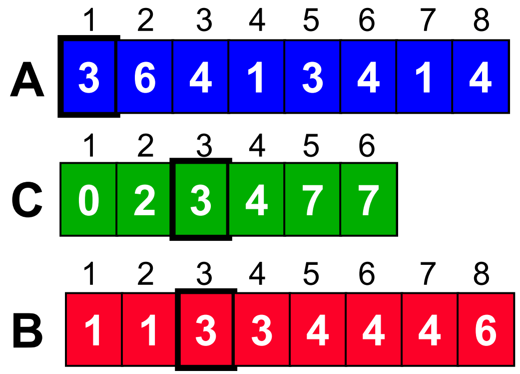
	1	2	3	4	5	6	7	8
B	1	1		3	4	4	4	6

Ao final da sétima iteração do último laço

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
     do C[i] ← 0
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
     do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
     do B[C[A[j]]] ← A[j]
          C[A[j]] ← C[A[j]] - 1
```





Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

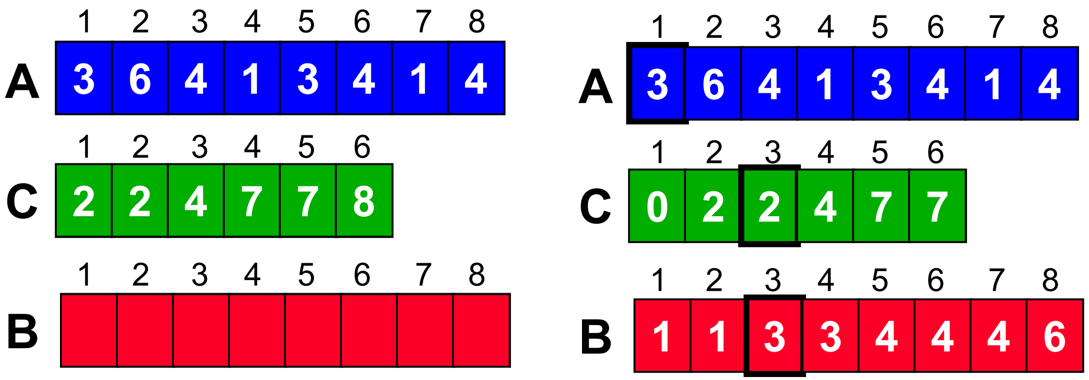
	1	2	3	4	5	6
C	0	2	2	4	7	7

	1	2	3	4	5	6	7	8
B	1	1	3	3	4	4	4	6

Ao final da última iteração do último laço

Revisitando RADIX-SORT

```
COUNTING-SORT(A,B,k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
for j ←length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```




```
RADIX-SORT(A,d)
for i ← 1 to d
    do      use COUNTING-SORT
            to sort A on digit i
```

Requer mais memória por item a ordenar do que QuickSort

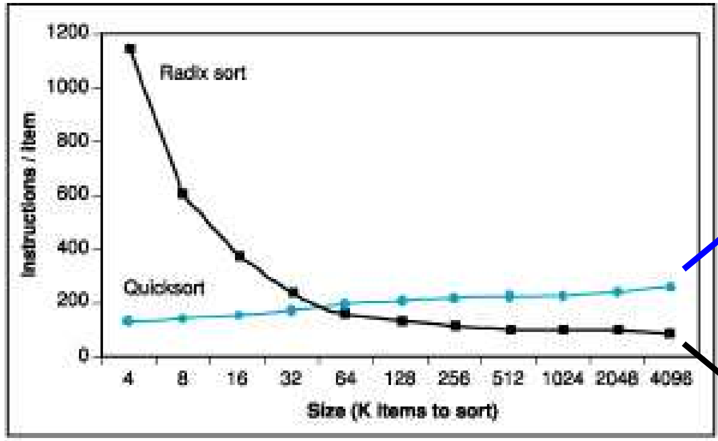
Localidade temporal (instruções)
Grande: laços

Localidade espacial (instruções)
Pouca: BBs pequenos

Localidade espacial (dados)
Pequena: só um dos arranjos sequencialmente*

Localidade temporal (dados)
Média: C e A são revisitados várias vezes

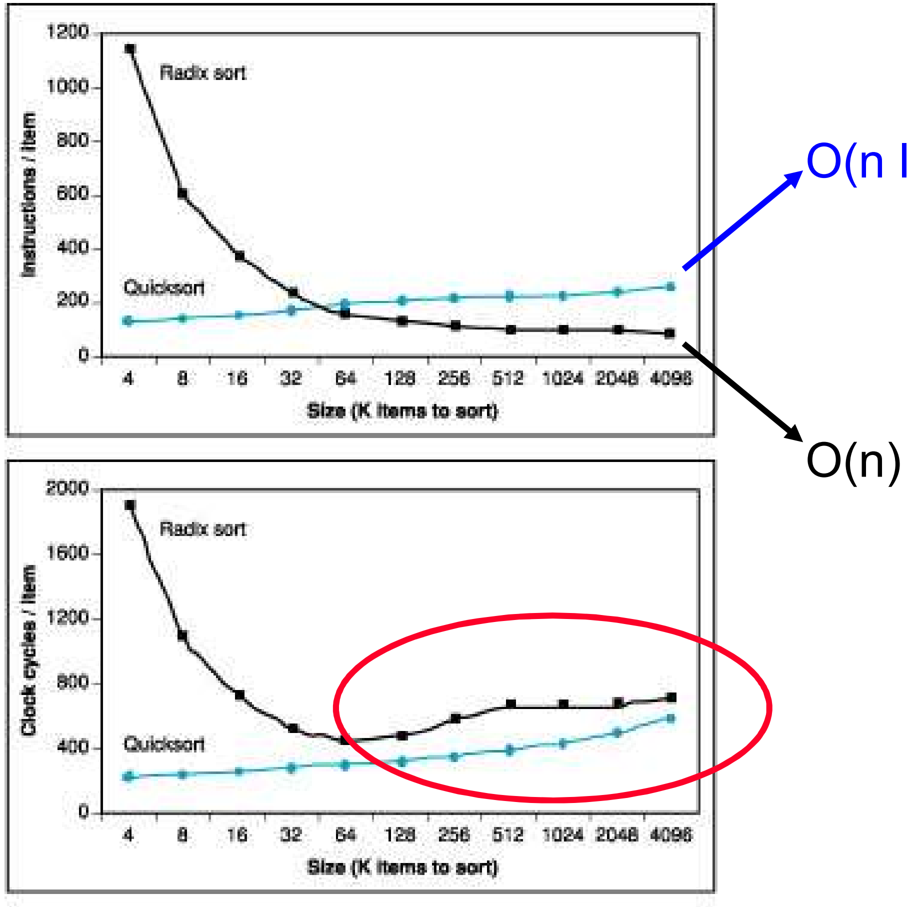
Impacto da cache no desempenho



O(n log n)

O(n)

Instruções / item

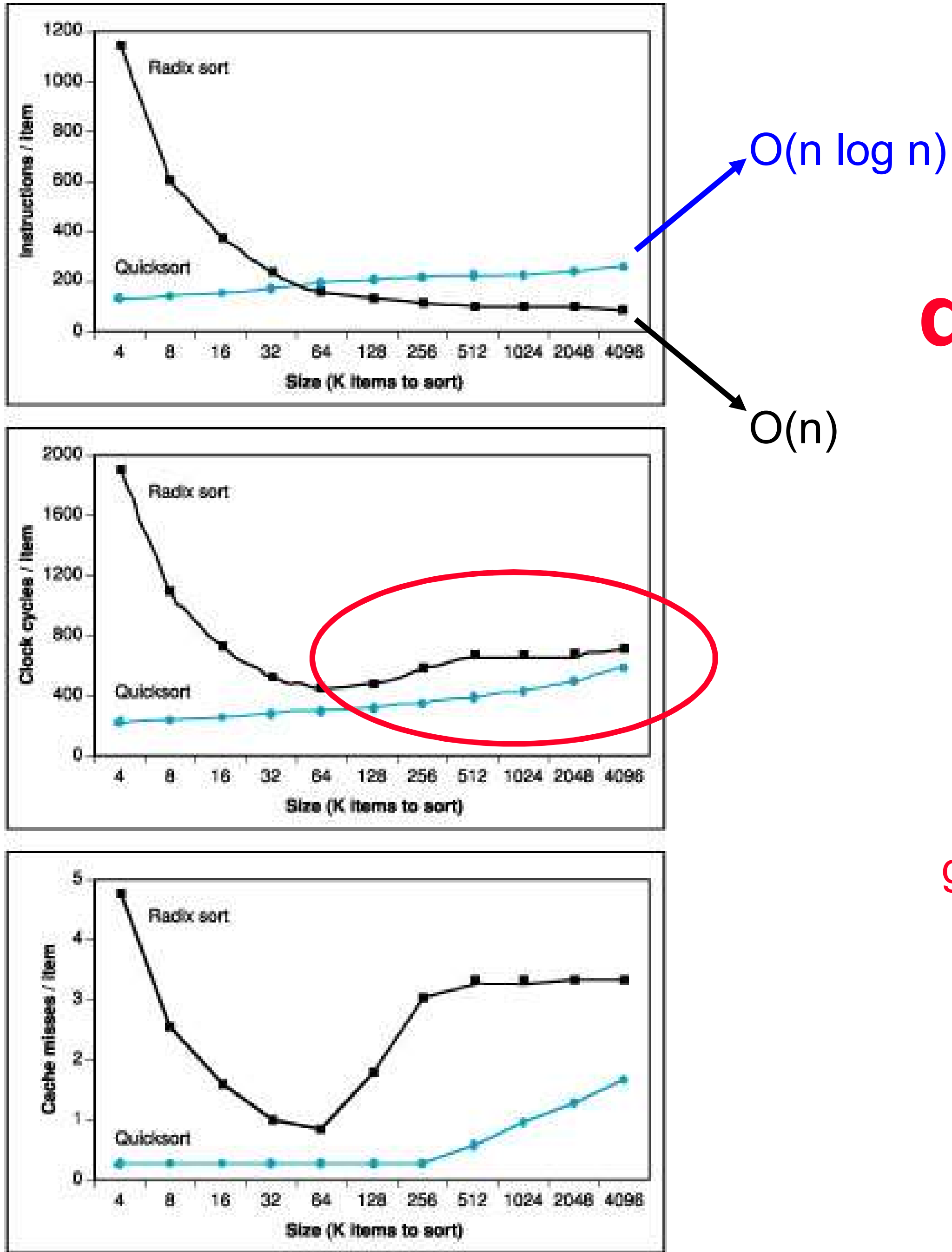


Impacto da cache no desempenho

Instruções / item

Ciclos / item

Impacto da cache no desempenho



Instruções / item

Ciclos / item

Faltas / item

Maior número de ciclos gastos por item deve-se à menor localidade de RadixSort

Otimizar para a cache: algoritmo e compilador

Aumentar desempenho via cache

- **Desenvolvedor de SW**
  - Escolha de algoritmo "cache friendly"
  - Implementação "cache aware"
    - » Ex. Linguagem aloca matrizes por linha, código deve ser ajustado para percorrer linhas (colunas)
  - Otimizações de compilador
- **Projetista do sistema computacional**
  - Seleção de parâmetros da cache para aplicação
    - » Capacidade, associatividade, tamanho de bloco, níveis
  - Algoritmo, implementação, compilador

Conclusão

- **Soluções para aumentar desempenho:**
  - **Aumento da associatividade**
    - » **Impacta mr**
  - **Múltiplos níveis de cache**
    - » **Impacta penalidade**
  - **Algoritmos e compiladores “cache-conscientes”**
    - » **Impactam mr**