Estrutura de Dados II (ED2)

Aula 27 – Radix Sort (Parte II)

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides do Professor Eduardo Zambon)

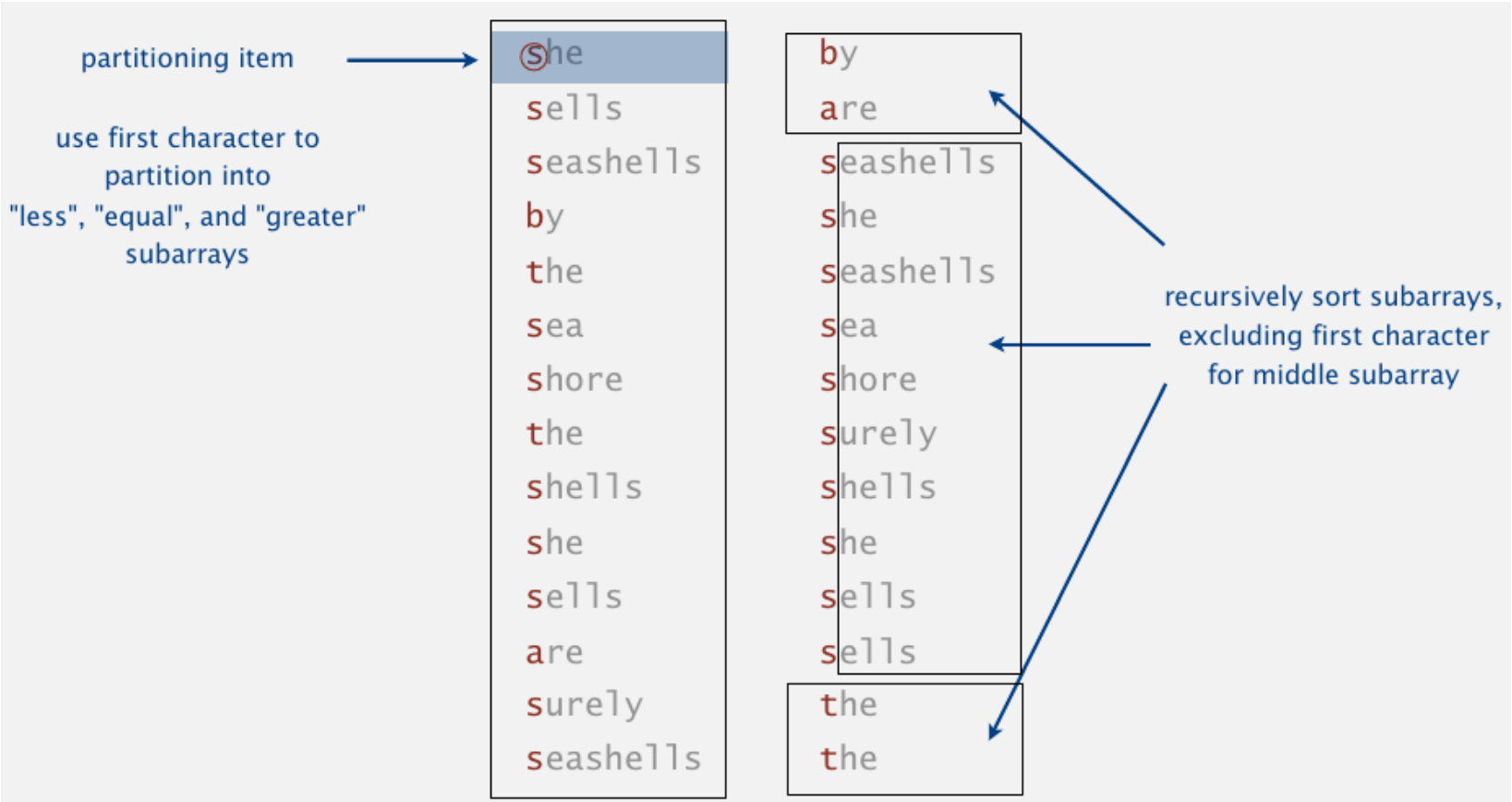
Parte I

3-Way Radix Quick Sort

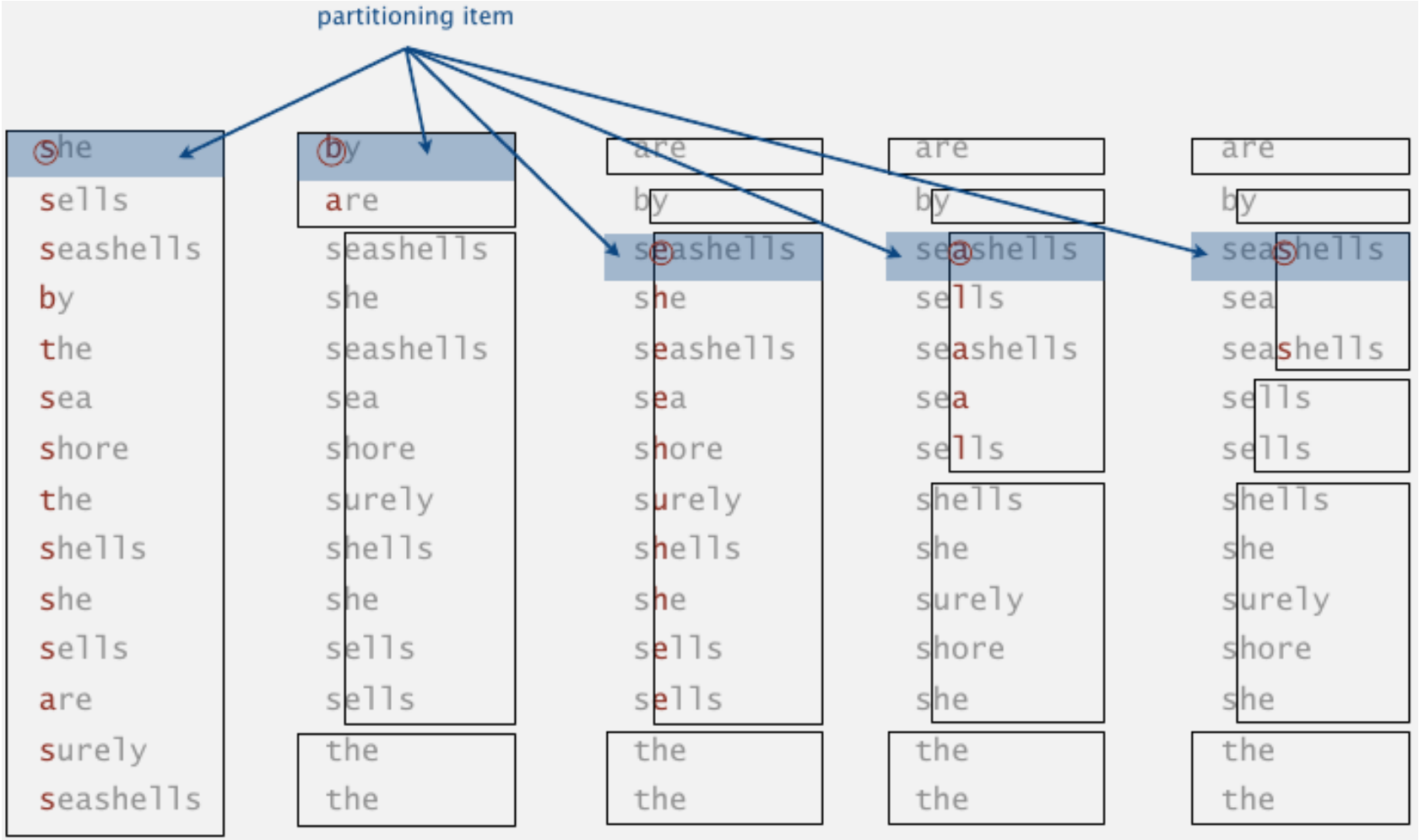
3-way radix quick sort (Bentley & Sedgewick, 1997)

Visão geral: faz particionamento *3-way* sobre o *d*-ésimo `char`.

- Menos *overhead* que particionamento *R-way* do MSD.
- Não re-examina caracteres iguais ao pivô.
  (Mas re-examina os caracteres diferentes do pivô.)



3-way radix quick sort: trace da recursão



3-way radix quick sort: implementação em C

```
void quick_sort(String* *a, int lo, int hi, int d) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;    // 3-way partitioning
    char v = a[lo]->c[d];    // (using dth character).
    int i = lo + 1;
    while (i <= gt) { // Partition.
        char t = a[i]->c[d];
        if      (t < v) { exch(a[lt], a[i]); lt++; i++; }
        else if (t > v) { exch(a[i], a[gt]); gt--; }
        else            { i++; }
    }
    quick_sort(a, lo, lt-1, d); // Sort 3 sub-arrays
    if (v > 0) { quick_sort(a, lt, gt, d+1); }
    quick_sort(a, gt+1, hi, d); // recursively.
}

void sort(String* *a, int N) {
    quick_sort(a, 0, N-1, 0);
}
```

Ordenação de *strings*: comparação de algoritmos

Quick sort padrão:

- Usa $\sim 2N \ln N$ comparações de *strings* na média.
- Custoso para chaves com prefixos comuns longos. (Caso frequente!)

3-way radix quick sort:

- Usa $\sim 2N \ln N$ comparações de caracteres na média.
- Evita comparar prefixos comuns longos mais de uma vez.

MSD radix sort:

- Não usa bem o cache.
- Gasta muita memória com *array* `count[]`.
- Muito *overhead* reinicializando `count[]` e `aux[]`.

3-way radix quick sort:

- Usa bem o cache.
- É *in-place*.
- Tem um *loop* interno curto.

Sumário: desempenho dos algoritmos de ordenação

algorithm	guarantee	random	extra space	stable?
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✔
mergesort	$N \lg N$	$N \lg N$	N	✔
quicksort	$1.39\ N \lg N$ *	$1.39\ N \lg N$	$c \lg N$	
heapsort	$3\ N\ lg\ N$	$3\ N\ lg\ N$	1	
LSD sort †	$W\ (3N + R)$	$W\ (3N + R)$	$N + R$	✔
MSD sort ‡	$W\ (3N + R)$	$N \log_R N$	$N + D\ R$	✔
3-way string quicksort	$1.39\ W\ N \lg R$ *	$1.39\ N \lg N$	$\log N + W$	

* probabilistic
† fixed-length W keys
‡ average-length W keys

Radix sorts para *strings*: análise empírica

```
Algorithm                          dict    book
-------------------------------------------------
System qsort                       0.409   0.559
Standard quick sort                0.430   0.499
Key-counting sort (Only works with strings of len = 1)
LSD                                0.967   0.601
MSD1: standard                     1.327   0.309
MSD2: cut-off ins-sort             0.229   0.202
3-way string quick sort            0.161   0.159
3-way quick sort + cut-off         0.162   0.150
```

Conclusão: *3-way radix quick sort* é o melhor método conhecido para ordenação de *strings*.

Parte II

Radix Sort para Números

Radix sort para inteiros

De *strings* para inteiros:

- Até agora vimos *radix sorts* para *strings*.
- Os mesmos algoritmos pode ser usados para ordenar inteiros.
- Basta identificar os componentes de um número.

Composição:

- *Strings* são formadas por caracteres.
- Inteiros são formados por *bits* e *bytes*.
- $R = 2$: análise de um número *bit* a *bit*.
- $R = 256$: análise de um número *byte* a *byte*.

Radix sort para inteiros: implementação em C

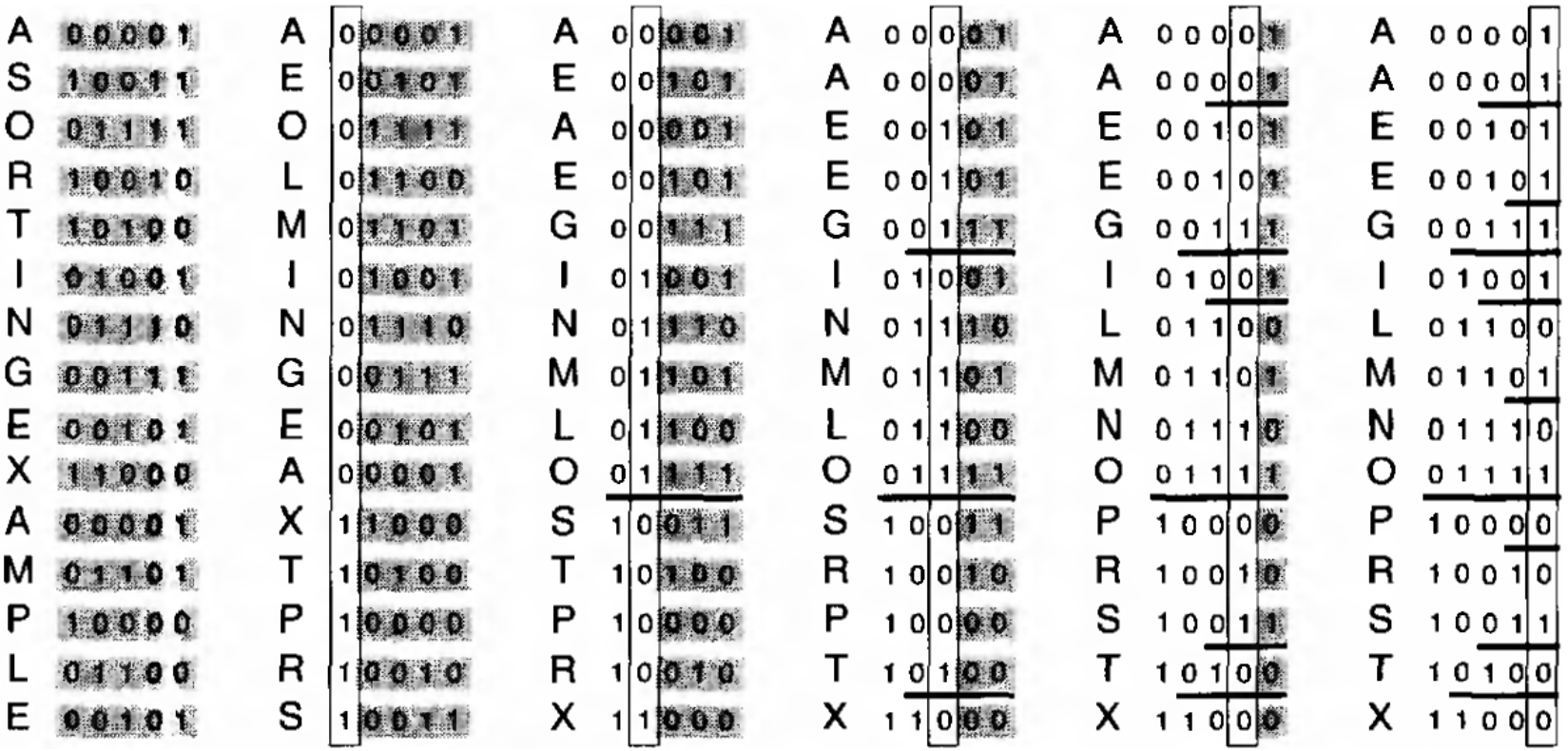
```
typedef int Item;

#define key(A)           (A)
#define less(A, B)       (key(A) < key(B))
#define exch(A, B)       { Item t = A; A = B; B = t; }
#define compexch(A, B)   if (less(B, A)) exch(A, B)

#define BITS_PER_WORD    32
#define BITS_PER_BYTE    8
#define BYTES_PER_WORD   4
#define R                (1 << BITS_PER_BYTE)

#define byte(A, B) \
    (((A) >> (BITS_PER_WORD - ((B)+1) * BITS_PER_BYTE)) & (R-1))

#define bit(A, B) \
    ((A) >> (BITS_PER_WORD - ((B)+1)) & 1U)
```

Alerta: contagem de *bits* e *bytes* é da esquerda para direita!

Bit (byte) 0 é o MSD.

Quick sort binário

- Tomando $R = 2$, podemos implementar uma versão binária do *quick sort*.
- Pivô é o d-ésimo *bit*. Sequência por MSD.

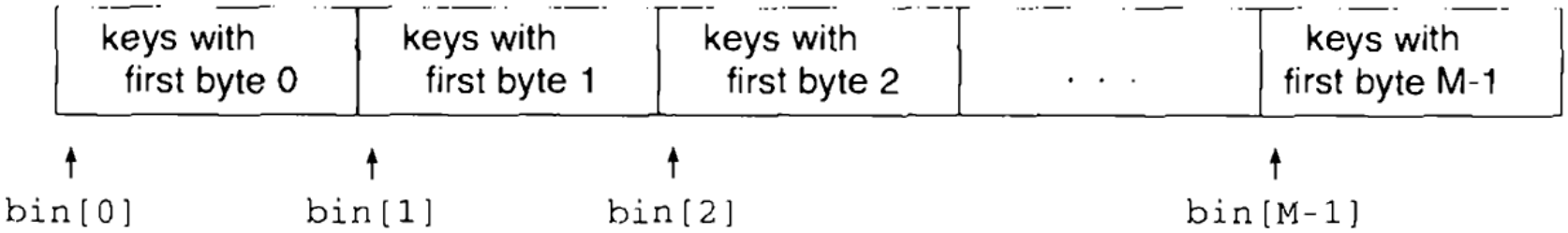


Quick sort binário: implementação em C

```
void bin_quick_sort(Item *a, int lo, int hi, int d) {
    if (hi <= lo || d > BITS_PER_WORD) return;

    int i = lo, j = hi;
    while (i != j) { // Partition.
        while (bit(a[i], d) == 0 && (i < j)) { i++; }
        while (bit(a[j], d) == 1 && (j > i)) { j--; }
        exch(a[i], a[j]);
    }

    if (bit(a[hi], d) == 0) { j++; } // All bits = 0.
    bin_quick_sort(a, lo, j-1, d+1); // Recursive sort
    bin_quick_sort(a, j, hi, d+1);   // on next bit.
}

void sort(Item *a, int N) {
    bin_quick_sort(a, 0, N-1, 0);
}
```

MSD radix sort

- Tomando 1 *bit* no *radix quick sort* $\Rightarrow$ MSD com $R = 2$.
- Fazendo $R = 256$ podemos analisar os números *byte* a byte.
- Com isso o *array* fica particionado em R partes.
- Tradicionalmente, as partes são chamadas de *bins* ou *buckets*.



MSD radix sort: implementação em C

```
void rec_MSD(Item *a, Item *aux, int lo, int hi, int d) {
    if (d > BYTES_PER_WORD) return;
    if (hi <= lo) return;

    int count[R+1];
    count_sort(a, aux, count, lo, hi, d);

    rec_MSD(a, aux, lo, lo + count[0] - 1, d+1);
    for (int r = 0; r < R-1; r++) { // Sort R arrays recursively
      rec_MSD(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
    }
}

void sort(Item *a, int N) {
    Item* aux = malloc(N * sizeof(Item));
    rec_MSD(a, aux, 0, N-1, 0);
    free(aux);
}
```

MSD radix sort: desempenho

Considerações sobre o *MSD radix sort*:

- Fazer $R = 256$ é uma boa ideia porque as arquiteturas atuais favorecem o acesso por *bytes* (ao invés de *bits*).
- O algoritmo MSD consegue deixar um *array* quase ordenado muito rapidamente.
- Mas, como antes, MSD sofre do problema de desempenho para *sub-arrays* pequenos.
- $\Rightarrow$ Usar *insertion sort*.

MSD radix sort: análise empírica

Tempo de execução dos algoritmos em segundos para entradas de tamanho 10^7 (10M).

```
RANDOM NUMBERS
Best: quick sort   2.031
----------------------
Bin quick sort     2.911
MSD               10.824
MSD + cut-off      0.782
----------------------
```

```
SORTED
Best: merge sort   0.089
-----------------------
Bin quick sort     1.274
MSD               10.814
MSD + cut-off      0.725
-----------------------
```

```
REVERSE SORTED
Best: quick sort   0.711
----------------------
Bin quick sort     1.286
MSD               11.006
MSD + cut-off      0.763
----------------------
```

```
NEARLY SORTED
Best: merge sort   0.217
-----------------------
Bin quick sort     1.289
MSD               10.826
MSD + cut-off      0.730
-----------------------
```

3-way radix quick sort para inteiros

```
void quick_sort(Item *a, int lo, int hi, int d) {
    if (hi <= lo + CUTOFF - 1) {
        insert_sort_from(a, lo, hi, d);
        return;
    }
    int lt = lo, gt = hi;       // 3-way partitioning
    Item v = byte(a[lo], d);    // (using dth byte).
    int i = lo + 1;
    while (i <= gt) { // Partition.
        Item t = byte(a[i], d);
        if      (t < v) { exch(a[lt], a[i]); lt++; i++; }
        else if (t > v) { exch(a[i], a[gt]); gt--; }
        else            { i++; }
    }
    quick_sort(a, lo, lt-1, d); // Sort 3 sub-arrays
    quick_sort(a, lt, gt, d+1);
    quick_sort(a, gt+1, hi, d); // recursively.
}

void sort(Item *a, int N) {
    quick_sort(a, 0, N-1, 0);
}
```

3-way radix quick sort: análise empírica

```
RANDOM NUMBERS
Best: quick sort   2.031
----------------------
Bin quick sort     2.911
MSD               10.824
MSD + cut-off      0.782
3-w-qsort + cut    3.124
----------------------
```

```
SORTED
Best: merge sort   0.089
-----------------------
Bin quick sort     1.274
MSD               10.814
MSD + cut-off      0.725
3-w-qsort + cut   14.568
-----------------------
```

```
REVERSE SORTED
Best: quick sort   0.711
----------------------
Bin quick sort     1.286
MSD               11.006
MSD + cut-off      0.763
3-w-qsort + cut   14.845
----------------------
```

```
NEARLY SORTED
Best: merge sort   0.217
-----------------------
Bin quick sort     1.289
MSD               10.826
MSD + cut-off      0.730
3-w-qsort + cut   14.464
-----------------------
```

3-way radix quick sort: discussão

Q: Por que *3-way radix quick sort* é tão bom para *strings* e tão ruim para inteiros?

A: Devido à forma de economia de trabalho do algoritmo.

- Não re-examina símbolos iguais ao pivô.
- Mas re-examina os símbolos diferentes do pivô.
- A partição de símbolos iguais é em média 10x maior para *strings* do que para inteiros.

Por que essa diferença?

- Inteiro pode ser visto como uma *string* de tamanho fixo.
- Em ambos os casos temos $R = 256$.
- Mas para *strings* na prática só comparamos 26 letras.
- Isso aumenta a chance de uma *string* entrar na partição de símbolos iguais ao pivô.

LSD radix sort

Podemos também implementar a versão *LSD radix sort*:

```
void sort(Item *a, int N) {
    Item* aux = malloc(N * sizeof(Item));
    int count[R+1];

    for (int d = BYTES_PER_WORD - 1; d >= 0; d--) {
        count_sort(a, aux, count, 0, N-1, d);
    }

    free(aux);
}
```

Dado que $N >> R$ a complexidade é linear ($\sim 12N$).

LSD radix sort: análise empírica

```
RANDOM NUMBERS
Best: quick sort   2.031
----------------------
Bin quick sort     2.911
MSD               10.824
MSD + cut-off      0.782
3-w-qsort + cut    3.124
LSD                0.554
----------------------
```

```
SORTED
Best: merge sort   0.089
----------------------
Bin quick sort     1.274
MSD               10.814
MSD + cut-off      0.725
3-w-qsort + cut   14.568
LSD                0.552
----------------------
```

```
REVERSE SORTED
Best: quick sort   0.711
----------------------
Bin quick sort     1.286
MSD               11.006
MSD + cut-off      0.763
3-w-qsort + cut   14.845
LSD                0.558
----------------------
```

```
NEARLY SORTED
Best: merge sort   0.217
----------------------
Bin quick sort     1.289
MSD               10.826
MSD + cut-off      0.730
3-w-qsort + cut   14.464
LSD                0.552
----------------------
```

LSD radix sort: discussão

Strings vs. inteiros:

- LSD não é recomendado para *strings* de tamanho variável.
- Inteiros sempre têm o mesmo tamanho (em *bytes*).

Por que *LSD radix sort* é tão eficiente?

- Algoritmo faz `BYTES_PER_WORD` passadas no *array*.
- Desempenho de `count_sort()` é linear em relação a N.
- Forma de acesso aos *bytes* é "*arch-friendly*".

Mas *MSD radix sort* não tem complexidade sub-linear?

- Sim, mas só no melhor caso.
- Na prática, os detalhes de implementação influenciam.
- *Overhead* das chamadas recursivas.
- Pouca localidade de *cache*.

Métodos de ordenação: conclusão

Vários métodos:

- Todos os métodos de ordenação estudados têm uma aplicação em algum cenário específico.
- Algoritmos como *merge sort* e *quick sort* são os mais usados na prática.
- Justificativa para tal é a facilidade de adaptação para diferentes tipos de chave.
- *Radix sorts* podem ter desempenho sub-linear em situações especiais.

Mensagem final:

- O *sort* do sistema foi projetado e testado para ter um bom desempenho na grande maioria dos casos.
- Nas situações especiais em que o desempenho não for suficiente, há métodos específicos melhores.