

Aula 08 – *Radix Sort*

Prof. Eduardo Zambon

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

Estrutura de Dados II (ED2)

Introdução

- Até agora: abstrair os itens a serem ordenados com a definição de uma **chave** de ordenação e fazer **comparação de chaves**.
- Outro método: algoritmos de ordenação por radicais (*radix sorting*) trabalham diretamente com os “**dígitos**” das chaves, comparando-as **em partes**.
- **Aula de hoje:** apresentação das diferentes implementações de *radix* e dos algoritmos de ordenação *radix sort*.
- **Objetivos:** compreender as aplicações e o funcionamento do método de ordenação *radix sort*.

Referências

Chapter 10 – Radix Sorting

R. Sedgewick

Parte I

Strings e Alfabetos

Tipo de dado `char`

Tipo de dado `char` em C:

- Tipicamente é um inteiro de 8 bits.
- Suporta a codificação **ASCII** de 7 bits.
- Pode representar no **máximo 256** caracteres.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

Em Java: suporta Unicode 16 de bits.

Tipo de dado `string` em C:

- Linguagem não possui um tipo *string* **explícito**.
- Representado por um `char*`.
- Como *arrays* em C não carregam o seu próprio tamanho, *strings* são terminadas pelo caractere `'\0'`.
- Maior parte das operações implementadas em `string.h`.

Strings em C são **mutáveis** \Rightarrow Cuidado ao usar como **chaves**!

Manipulação de *strings* em C é **muito propensa a erros**.

Exemplo:

```
int main() {  
    char *s = "abc";  
    s[0] = 'd'; // 'a' to 'd'  
    // Segmentation fault... --  
}
```

Novo tipo de dado *String*

Novo tipo *String* para evitar o inferno astral de ter de lidar com *array* de `char` terminado por `'\0'`.

```
typedef struct {  
    char *c;  
    int len;  
} String;
```

Vantagens:

- Pode ajudar na *imutabilidade* tornando a estrutura *opaca*.
- Obter o tamanho de uma *string* agora é *constante*.
- Elimina erros como do slide anterior já que todas as *strings* são *alocadas no heap*.

Novo tipo de dado *String*: operações

Comparar duas *strings* a partir do caractere na **posição *d***.

```
int compare_from(String *s, String *t, int d) {  
    int min = s->len < t->len ? s->len : t->len;  
    for (int i = d; i < min; i++) {  
        if (s->c[i] < t->c[i]) { return -1; }  
        if (s->c[i] > t->c[i]) { return 1; }  
    }  
    return s->len - t->len;  
}
```

Retorna:

- **-1**: se $s < t$.
- **1**: se $s > t$.
- **0**: se $s = t$.

Atenção:

- Se s é **prefixo** de t , retorna < 0 .
- **Exemplo**: $she < shells$.

Novo tipo de dado *String*: operações

Demais operações são **triviais** de implementar a partir de `compare_from`.

```
int compare(String *s, String *t) {  
    return compare_from(s, t, 0);  
}
```

Interface usual de **Item** também fica simples.

```
typedef String* Item;  
  
#define less(A, B) (compare(A, B) < 0)  
#define less_from(A, B, d) (compare_from(A, B, d) < 0)  
#define exch(A, B) { Item t = A; A = B; B = t; }
```


Comparando duas *strings*

Q: Quantas comparações de caracteres são necessárias para comparar **duas strings** de **comprimento W** ?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x	e	s

Tempo de execução: Proporcional ao comprimento do **maior prefixo em comum**.

- Proporcional a W no **pior caso**.
- Mas geralmente é **sublinear** em relação a W .

Alfabetos

String: Sequência de **símbolos** (caracteres, dígitos) sobre um alfabeto fixo.

Radix: Número ***R*** de **símbolos** de um alfabeto (tamanho).

name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-
ASCII	128	7	<i>ASCII characters</i>
EXTENDED_ASCII	256	8	<i>extended ASCII characters</i>
UNICODE16	65536	16	<i>Unicode characters</i>

Parte II

Contagem Indexada por Chaves *Key-Indexed Counting*

Sumário: desempenho dos algoritmos de ordenação

Quantidade de operações feita por cada algoritmo.

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	

* probabilistic

Limite inferior: pelo menos $\sim N \lg N$ comparações.

- **Q:** É possível fazer melhor? (Apesar do limite inferior.)
- **A:** Sim, evitando **comparação de chaves**.
- **Comparação de chaves:** decisões **binárias**.
- **Comparação por radix R :** decisões têm R resultados.

Key-indexed counting: suposições sobre as chaves

Suposição: chaves são **inteiros** entre **0** e **$R - 1$** .

Implicação: podemos usar as chaves como **índices** de *array*.

Aplicações:

- Ordenar uma *string* pela primeira letra.
- Ordenar empregados por seção.
- Ordenar números de telefone por código de área.
- Sub-rotina em um algoritmo de ordenação (em breve).

input		sorted result	
name	section	(by section)	
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

↑
keys are
small integers

Key-indexed counting: implementação e exemplo

Objetivo: ordenar um *array* $a[]$ de N inteiros entre 0 e $R - 1$.

- Vetor auxiliar 1: $count[]$ de tamanho $R + 1$.
- Vetor auxiliar 2: $aux[]$ de tamanho N .
- No exemplo abaixo: $R = 6$.

```
for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

Key-indexed counting: implementação e exemplo

Objetivo: ordenar um *array* $a[]$ de N inteiros entre 0 e $R - 1$.

■ **P.1:** Contar a frequência de cada valor usando $count[]$.

count frequencies →

```
for (int i = 0; i < N; i++)  
    count[a[i]+1]++;  
  
for (int r = 0; r < R; r++)  
    count[r+1] += count[r];  
  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)  
    a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

↓

r count[r]

a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting: implementação e exemplo

Objetivo: ordenar um *array* $a[]$ de N inteiros entre 0 e $R - 1$.

■ **P.2:** Computar a frequência **acumulada**.

compute
cumulates

```
for (int i = 0; i < N; i++)  
    count[a[i]+1]++;
```

```
for (int r = 0; r < R; r++)  
    count[r+1] += count[r];
```

```
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];
```

```
for (int i = 0; i < N; i++)  
    a[i] = aux[i];
```

i	a[i]	r	count[r]
0	d		
1	a		
2	c		
3	f	a	0
4	f	b	2
5	b	c	5
6	d	d	6
7	b	e	8
8	f	f	9
9	b		12
10	e		
11	a		

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key-indexed counting: implementação e exemplo

Objetivo: ordenar um *array* $a[]$ de N inteiros entre 0 e $R - 1$.

■ **P.3:** Mover os itens para um vetor *auxiliar*.

```
for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

move items →

i	a[i]	r	count[r]	i	aux[i]
0	d			0	a
1	a			1	a
2	c			2	b
3	f	a	2	3	b
4	f	b	5	4	b
5	b	c	6	5	c
6	d	d	8	6	d
7	b	e	9	7	d
8	f	f	12	8	e
9	b	-	12	9	f
10	e			10	f
11	a			11	f

Key-indexed counting: implementação e exemplo

Objetivo: ordenar um *array* $a[]$ de N inteiros entre 0 e $R - 1$.

■ **P.4:** Copiar de volta para o vetor original (se necessário).

```
for (int i = 0; i < N; i++)  
    count[a[i]+1]++;
```

```
for (int r = 0; r < R; r++)  
    count[r+1] += count[r];
```

```
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];
```

copy
back

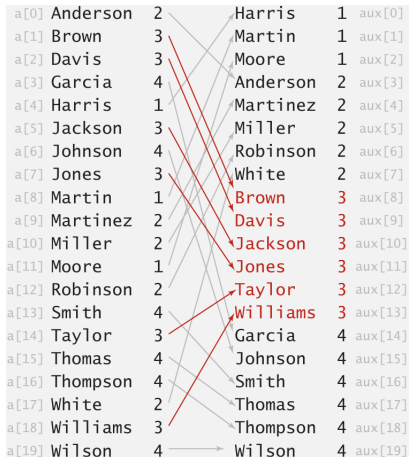
```
for (int i = 0; i < N; i++)  
    a[i] = aux[i];
```

i	a[i]		i	aux[i]
0	a		0	a
1	a		1	a
2	b		2	b
3	b		3	b
4	b	a	4	b
5	c	b	5	c
6	d	c	6	d
7	d	d	7	d
8	e	e	8	e
9	f	f	9	f
10	f	-	10	f
11	f	-	11	f

r	count[r]
2	2
5	5
6	6
8	8
9	9
12	12
12	12

Key-indexed counting: análise

- Tempo de execução é $\sim 3N + R$.
- Espaço extra é $\sim N + R$.
- Método é estável.



Key-indexed counting: implementação em C

Versão final **generalizada** para uso em outros *sorts*:

- Faz a **contagem** sobre o **d-ésimo** caractere.

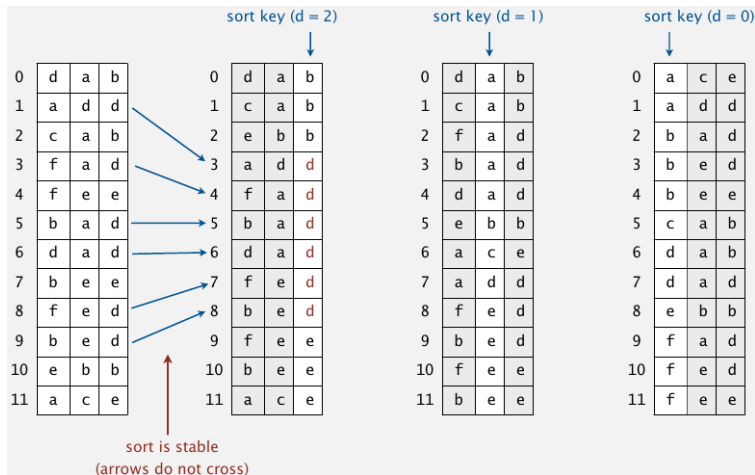
```
void count_sort(String* *a, String* *aux, int *count,
                int lo, int hi, int d, int R) {
    clear_count_array(count, R);
    for (int i = lo; i <= hi; i++) { // Count frequencies.
        count[char_at(a[i], d) + 2]++;
    }
    for (int r = 0; r < R+1; r++) { // Compute cumulates.
        count[r+1] += count[r];
    }
    for (int i = lo; i <= hi; i++) { // Move items.
        int p = count[char_at(a[i], d) + 1]++;
        aux[p] = a[i];
    }
    for (int i = lo; i <= hi; i++) { // Copy back.
        a[i] = aux[i - lo];
    }
}
```

Parte III

LSD Radix Sort

LSD (Least Significant Digit) radix sort

- Considera os caracteres da direita para a esquerda.
- Faz uma ordenação **estável** usando o **d-ésimo** caractere como chave (usa *key-indexed counting*).



LSD radix sort: correção

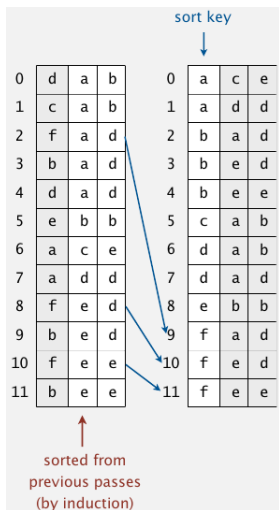
Proposição: LSD ordena *strings* de tamanho fixo de forma ascendente.

Prova (por indução em i):

Depois da passada i , as *strings* estão ordenadas pelos **últimos i** caracteres.

- **Duas *strings* com chave de ordenação diferentes:** *key-indexed sort* as coloca na ordem relativa adequada.
- **Duas *strings* com chave de ordenação iguais:** estabilidade as mantém ordem relativa adequada.

LSD sort é **estável** porque *key-indexed sort* é estável.



LSD radix sort: implementação em C

```
void sort(String* a, int N) {  
    int W = 3; // Change accordingly to input size.  
    int R = 256;  
  
    String* aux = create_str_array(N);  
    int* count = create_count_array(R);  
  
    for (int d = W-1; d >= 0; d--) {  
        count_sort(a, aux, count, 0, N-1, d, R);  
    }  
  
    free(count);  
    free(aux);  
}
```


Sumário: desempenho dos algoritmos de ordenação

Quantidade de operações feita por cada algoritmo.

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 †	$W (3N + R)$	$W (3N + R)$	$N + R$	✓
† fixed-length W keys * probabilistic				

Q: E se as *strings* não são todas do mesmo tamanho?

Como ficar rico ordenando nos anos de 1900?

Cartões perfurados (1900 a 1950):

- Também eram úteis para contabilidade, inventário e outros processos de negócios.
- Principal **mídia** para entrada, armazenamento e processamento de dados.

A empresa de Hollerith depois se fundiu com mais 3 outras para formar a **Computing Tabulating Recording Corporation (CTRC)**. Empresa foi renomeada em 1924.



IBM 80 Series Card Sorter (650 cards per minute)



LSD radix sort: um momento na história (1960s)



card punch



punched cards



card reader



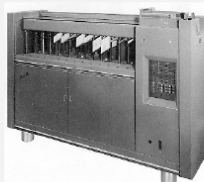
mainframe



line printer

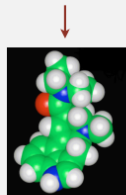
To sort a card deck

- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

not directly related
to sorting



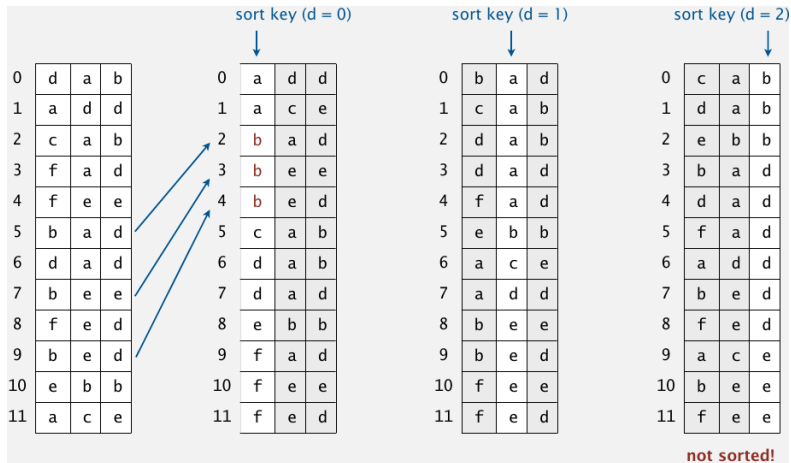
Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

Parte IV

MSD Radix Sort

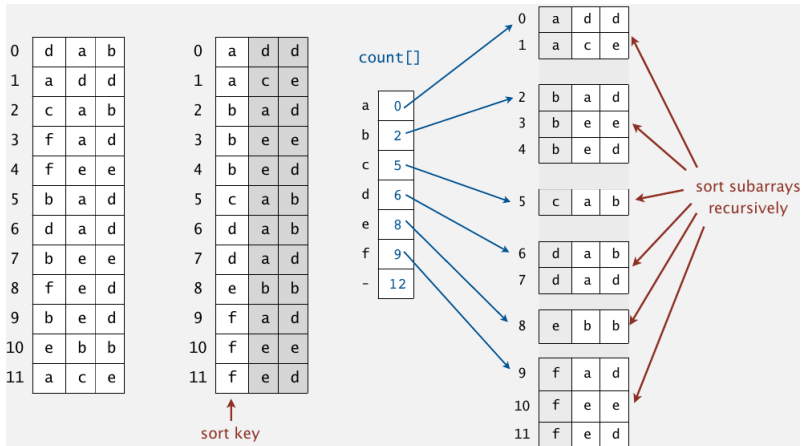
LSD invertido

- Considera os caracteres da esquerda para a direita.
- Faz uma ordenação estável usando o d -ésimo caractere como chave (usa *key-indexed counting*).



MSD (Most Significant Digit) radix sort

- Particiona o *array* em ***R* partes** segundo o primeiro caractere, usando *key-indexed counting*.
- Ordena **recursivamente** cada partição.



MSD radix sort: exemplo

input

she	are	are	are	are	are	are	are
sells	by	by	by	by	by	by	by
seashells	she	seashells	seashells	seashells	seashells	seashells	seashells
by	sells	seashells	sea	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells
sea	sea	sells	sells	sells	sells	sells	sells
shore	shore	seashells	sells	sells	sells	sells	sells
the	shells	she	she	she	she	she	she
shells	she	shore	shore	shore	shore	shore	shore
she	sells	shells	shells	shells	shells	shore	shells
sells	surely	she	she	she	she	she	she
are	seashells	surely	surely	surely	surely	surely	surely
surely	the	the	the	the	the	the	the
seashells	the	the	the	the	the	the	the

Annotations: "10" points to the 's' in 'seashells' of the first row. "hi" points to the 'h' in 'the' of the second row.

need to examine every character in equal keys

end of string goes before any char value

are	are	are	are	are	are	are	output
by	by	by	by	by	by	by	are
sea	sea	sea	sea	sea	sea	sea	by
seashells	seashells	seashells	seashells	seashells	seashells	seashells	sea
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
sells	sells	sells	sells	sells	sells	sells	sells
sells	sells	sells	sells	sells	sells	sells	sells
she	she	she	she	she	she	she	she
shore	shore	shore	shells	she	she	she	she
shells	hells	shells	she	shells	shells	shells	shells
she	she	she	shore	shore	shore	shore	shore
surely	surely	surely	surely	surely	surely	surely	surely
the	the	the	the	the	the	the	the
the	the	the	the	the	the	the	the

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Strings de tamanho variável

Considerar que as *strings* têm um caractere extra no final menor que todos os outros.

0	s	e	a	-1						
1	s	e	a	s	h	e	l	l	s	-1
2	s	e	l	l	s	-1				
3	s	h	e	-1						
4	s	h	e	-1						
5	s	h	e	l	l	s	-1			
6	s	h	o	r	e	-1				
7	s	u	r	e	l	y	-1			

she before shells

Implementação em C é bem simples.

```
#define char_at(S, D) ((D < S->len) ? S->c[D] : -1)
```

MSD radix sort: implementação em C

Caractere “-1” fica na posição 1 de count.

```
for (int i = lo; i <= hi; i++) // Count frequencies.
    count[char_at(a[i], d) + 2]++;
```

Código abaixo reutiliza aux mas não count. Necessário porque chamadas recursivas **modificariam** o vetor count.

```
#define R 256
void rec_MSD(String* *a, String* *aux, int lo, int hi, int d) {
    if (hi <= lo) return;
    int* count = create_count_array(R);
    count_sort(a, aux, count, lo, hi, d, R); // Key-indexed count.
    for (int r = 1; r < R+1; r++) // Sort R arrays recursively.
        rec_MSD(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
    free(count);
}
void sort(String* *a, int N) {
    String* *aux = create_str_array(N);
    rec_MSD(a, aux, 0, N-1, 0);
    free(aux);
}
```

Observação 1: algoritmo **muito lento** para *sub-arrays* **pequenos**.

- Cada chamada de função precisa do seu próprio *array* `count`.
- Tabela ASCII: *array* tem tamanho 256.
- Unicode: *array* tem tamanho 65,536!
- Mesmo para ordenar um *sub-array* de tamanho 2 é necessário alocar o *array* `count` **todo**: não sabemos o conteúdo a ser ordenado.

Observação 2: temos um grande número de *sub-arrays* **pequenos** por causa da recursão.

Cut-off para insertion sort

Solução: *cut-off* para *insertion sort* para *sub-arrays* pequenos.

Insertion sort, mas começando no **d-ésimo** caractere.

```
void insert_sort_from(Item *a, int lo, int hi, int d) {
    for (int i = lo; i <= hi; i++) {
        for (int j = i; j > lo && less_from(a[j], a[j-1], d); j--) {
            exch(a[j], a[j-1]);
        }
    }
}
```

Relembrando: função `less_from()` compara as *strings* a partir do **d-ésimo** caractere.

Modificar a função recursiva como de costume.

```
void rec_MSD(String* *a, String* *aux, int lo, int hi, int d) {
    if (hi <= lo + CUTOFF - 1) {
        insert_sort_from(a, lo, hi, d);
        return;
    }
}
```

MSD radix sort: desempenho

- MSD examina o **mínimo** de caracteres necessários para ordenar as chaves.
- Número de caracteres examinados **depende das chaves**.
- Pode ser **sub-linear** (no **melhor caso**) em relação ao tamanho da entrada!

Random (sublinear)	Non-random 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

Characters examined by MSD string sort

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 ‡	$W (3N + R)$	$N \log_R N$	$N + D R$	✓
<div> <div> * probabilistic † fixed-length W keys ‡ average-length W keys </div> <div> D = function-call stack depth (length of longest prefix match) </div> </div>				

MSD radix sort: análise empírica

Casos de teste:

- `dict`: dicionário com as palavras embaralhadas contendo **370,099** palavras distintas.
- `book`: versão em texto simples do livro **Guerra e Paz**, de *Liev Tolstói*. Contém **562,639** palavras (com repetições).

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

Note como **cut-off** para *insertion sort* faz diferença no MSD!

MSD radix sort vs. quick sort

Desvantagens do *MSD radix sort*:

- Espaço extra para `aux[]` e `count[]`.
- Loop interno tem muitas instruções.
- Acessa a memória “aleatoriamente” (ruim para cache).

Desvantagens do *quick sort*:

- Número de comparações de *strings* não é linear.
- Precisa examinar várias vezes os mesmos caracteres quando há prefixos iguais longos.

Objetivo:

- Combinar as vantagens de MSD e *quick sort*.
- Vantagem do MSD: examina cada caractere uma vez.
- Vantagem do *quicksort*: loop interno simples, faz bom uso do cache.

American flag sort

Otimização 0: *cut-off* para *insertion sort*.

Otimização 1: Substituir recursão por uma **pilha explícita**.

- Empilha os *sub-arrays* a serem ordenados na pilha. (Igual *quick sort* não-recursivo.)
- Feito isso, basta **um único** *array* `count[]`.

Otimização 3: Faça o particionamento *R-way in place*.

- Elimina o *array* `aux[]`.
- Sacrifica estabilidade.



American national flag problem



Dutch national flag problem

Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-acy sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Parte V

3-Way Radix Quick Sort

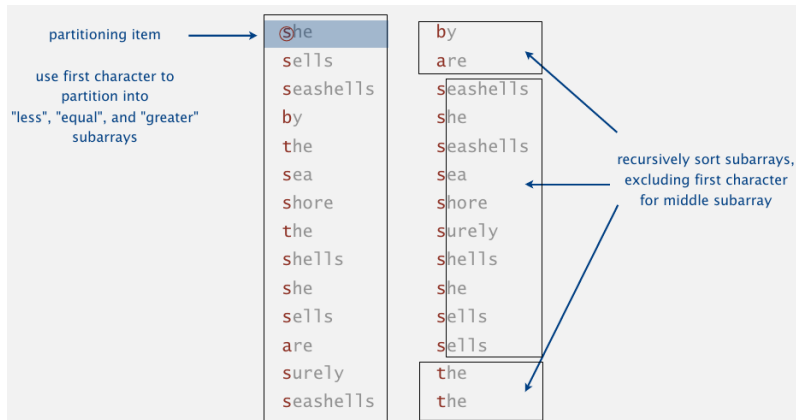
3-way radix quick sort (Bentley & Sedgwick, 1997)

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

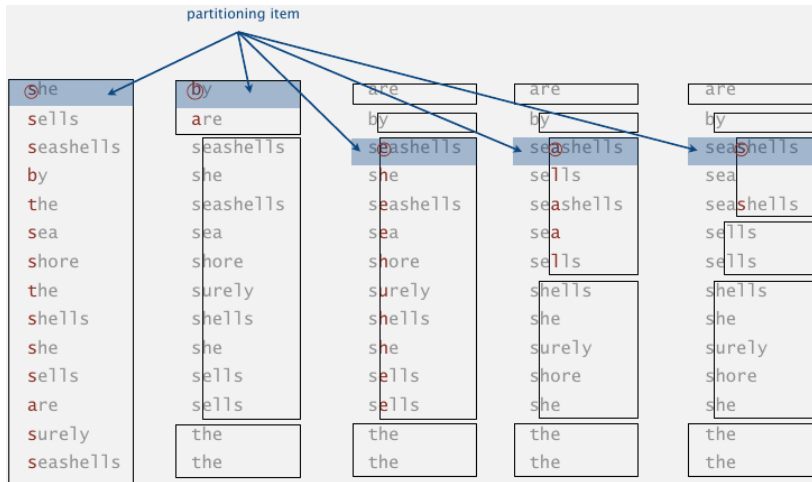
- Menos *overhead* que particionamento *R-way* do MSD.

- **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* reiniciando `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 VI

Radix Sort para Números

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 compech(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.

A	0 0 0 0 1	A	0 0 0 0 1	A	0 0 0 0 1	A	0 0 0 0 1	A	0 0 0 0 1	A	0 0 0 0 1
S	1 0 0 1 1	E	0 0 1 0 1	E	0 0 1 0 1	A	0 0 0 0 1	A	0 0 0 0 1	A	0 0 0 0 1
O	0 1 1 1 1	O	0 1 1 1 1	A	0 0 0 0 1	E	0 0 1 0 1	E	0 0 1 0 1	E	0 0 1 0 1
R	1 0 0 1 0	L	0 1 1 0 0	E	0 0 1 0 1	E	0 0 1 0 1	E	0 0 1 0 1	E	0 0 1 0 1
T	1 0 1 0 0	M	0 1 1 0 1	G	0 0 1 1 1	G	0 0 1 1 1	G	0 0 1 1 1	G	0 0 1 1 1
I	0 1 0 0 1	I	0 1 0 0 1	I	0 1 0 0 1	I	0 1 0 0 1	I	0 1 0 0 1	I	0 1 0 0 1
N	0 1 1 1 0	N	0 1 1 1 0	N	0 1 1 1 0	N	0 1 1 1 0	L	0 1 1 0 0	L	0 1 1 0 0
G	0 0 1 1 1	G	0 0 1 1 1	M	0 1 1 0 1	M	0 1 1 0 1	M	0 1 1 0 1	M	0 1 1 0 1
E	0 0 1 0 1	E	0 0 1 0 1	L	0 1 1 0 0	L	0 1 1 0 0	N	0 1 1 1 0	N	0 1 1 1 0
X	1 1 0 0 0	A	0 0 0 0 1	O	0 1 1 1 1	O	0 1 1 1 1	O	0 1 1 1 1	O	0 1 1 1 1
A	0 0 0 0 1	X	1 1 0 0 0	S	1 0 0 1 1	S	1 0 0 1 1	P	1 0 0 0 0	P	1 0 0 0 0
M	0 1 1 0 1	T	1 0 1 0 0	T	1 0 1 0 0	R	1 0 0 1 0	R	1 0 0 1 0	R	1 0 0 1 0
P	1 0 0 0 0	P	1 0 0 0 0	P	1 0 0 0 0	P	1 0 0 0 0	S	1 0 0 1 1	S	1 0 0 1 1
L	0 1 1 0 0	R	1 0 0 1 0	R	1 0 0 1 0	T	1 0 1 0 0	T	1 0 1 0 0	T	1 0 1 0 0
E	0 0 1 0 1	S	1 0 0 1 1	X	1 1 0 0 0	X	1 1 0 0 0	X	1 1 0 0 0	X	1 1 0 0 0

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;

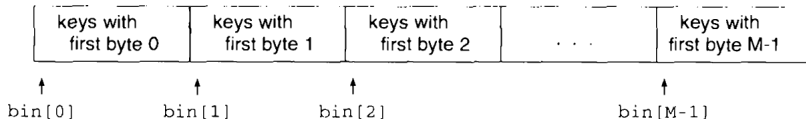
    int i = lo, j = hi;
    while (i != j) { // Partition.
        while (bit(a[i], d) == 0 && (i < j)) { i++; }
        while (bit(a[j], d) == 1 && (j > i)) { j--; }
        exch(a[i], a[j]);
    }

    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);
}
```

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

REVERSE SORTED

Best: quick sort 0.711

Bin quick sort 1.286

MSD 11.006

MSD + cut-off 0.763

SORTED

Best: merge sort 0.089

Bin quick sort 1.274

MSD 10.814

MSD + cut-off 0.725

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

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

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

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.

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 $N \gg R$ 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

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

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

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.

Aula 08 – *Radix Sort*

Prof. Eduardo Zambon

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

Estrutura de Dados II (ED2)