Técnicas de Busca e Ordenação (TBO)

Quick Sort

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

(Material baseado nos slides dos Professores Eduardo Zambon e Giovanni Comarela)

Introdução

- Métodos de ordenação são essenciais nas mais diferentes aplicações.
- **Aula de hoje**: apresentação do algoritmo clássico de ordenação *quick sort* e suas principais características.
- **Objetivos**: compreender o funcionamento do método de ordenação *quick sort*, e analisar o seu desempenho.

Referências

Chapter 7 – Quicksort
R. Sedgewick

Exemplo

1, 8, 9, 4, 5, 9, 12, 15

Exemplo

15, 12, 9, 9, 8, 5, 4, 1

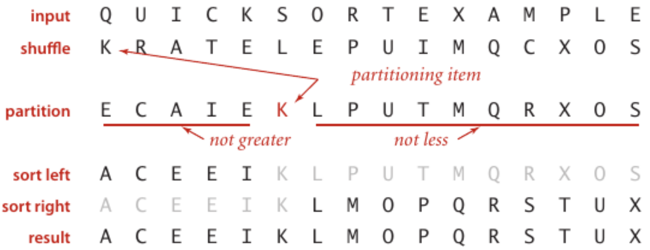
Quick sort

Passo 1: embaralhe o *array*.

Passo 2: Particione o *array* de forma que, para algum `j`

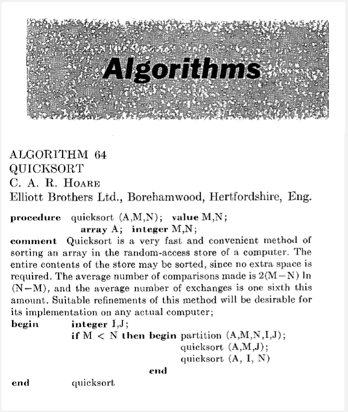
- Item `a[j]` está na sua posição.
- Nenhum item maior está à esquerda de `j`.
- Nenhum item menor está à direita de `j`.

Passo 3: Ordene cada *sub-array* recursivamente.



Tony Hoare

- Inventou o *quick sort* para traduzir Russo para Inglês.
- Mas não conseguiu explicar, nem implementar!
- Voltou para Inglaterra: aprendeu Algol 60 (e recursão).
- Implementou o *quick sort*.

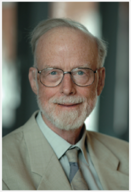


Algorithms

ALGORITHM 64
QUICKSORT
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure   quicksort (A,M,N);   value M,N;
               array A;   integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin        integer I,J;
             if M < N then begin partition (A,M,N,I,J);
                                 quicksort (A,M,J);
                                 quicksort (A, I, N)
                           end
end          quicksort
```

Communications of the ACM (July 1961)



Tony Hoare
1980 Turing Award

Tony Hoare

> *" There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. "*

> *" I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. "*

Tony Hoare
1980 Turing Award

- Refinou e popularizou o *quick sort*.
- Analisou muitas versões do *quick sort*.

Programming Techniques — S. L. Graham, R. L. Rivest, Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 19, 1976

Summary. The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

Bob Sedgewick

Quick sort partitioning demo

Repita até que os índices `i` e `j` se cruzem.

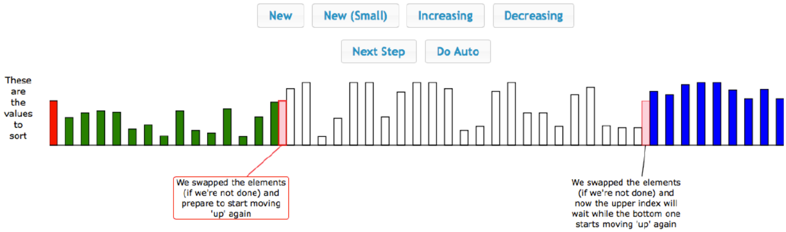
- Percorra `i` da esquerda para a direita enquanto (`a[i] < a[lo]`).
- Percorra `j` da direita para a esquerda enquanto (`a[j] > a[lo]`).
- Troque `a[i]` e `a[j]`.

Quando `i` e `j` se cruzarem

- Troque `a[lo]` e `a[j]`.

Ver arquivo `23DemoPartitioning.pdf`, primeiro exemplo.

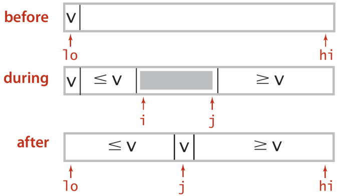
A “música” do particionamento do *quick sort*



https://learnforeverlearn.com/pivot_music/

Quick sort: implementação em C do particionamento

```
int partition(Item *a, int lo, int hi) {
    int i = lo, j = hi+1;
    Item v = a[lo];
    while(1) {
        while (less(a[++i], v))  // Find item on left to swap.
            if (i == hi) break;
        while (less(v, a[--j]))  // Find item on right to swap.
            if (j == lo) break;
        if (i >= j) break; // Check if pointers cross.
        exch(a[i], a[j]);
    }
    exch(a[lo], a[j]); // Swap with partitioning item.
    return j; // Return index of item known to be in place.
}
```



Quick sort: implementação em C

```c
void shuffle(Item *a, int N) {
    struct timeval tv; gettimeofday(&tv, NULL);
    srand48(tv.tv_usec);
    for (int i = N-1; i > 0; i--) {
        int j = (unsigned int) (drand48()*(i+1));
        exch(a[i], a[j]);
    }
}

void quick_sort(Item *a, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
}

void sort(Item *a, int lo, int hi) {
    shuffle(a, hi-lo+1); // Needed for performance guarantee.
    quick_sort(a, lo, hi);
}
```

Quick sort trace

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

Quick sort: animações

Veja as animações em:
`https://www.toptal.com/developers/sorting-algorithms/quick-sort`

Quick sort: detalhes de implementação

Particionamento *in-place*: Usar um *array* extra facilita o particionamento e o deixa estável. Custo alto de cópia.

Terminando o *loop*: Testar se os índices se cruzam é mais difícil do que parece quando há chaves iguais. (Detalhes adiante.)

Chaves iguais: contra-intuitivamente é melhor parar os *scans* nas chaves iguais à chave particionadora. (Detalhes adiante.)

Preservando aleatoriedade: *shuffling* é necessário para garantia de desempenho.

Alternativa equivalente: escolher a chave particionadora (AKA pivô) aleatoriamente em cada *sub-array*.

Quick sort: análise empírica (1961)

Tempo de execução estimado:

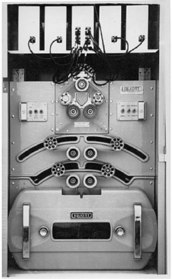
- Implementação em Algol 60.
- Execução no computador National Elliott 405.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting n 6-word items with 1-word keys



Elliott 405 magnetic disc (16K words)

Quick sort: análise empírica

Tempo de execução estimado:

- Computador pessoal executa 10^8 comparações/segundo.
- Supercomputador executa 10^{12} comparações/segundo.

	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Melhor caso: # de comparações é $\sim N \lg N$ (partição no meio).

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quick sort: análise de pior caso

Pior caso: # comparações é $\sim \frac{1}{2}N^2$ (partição no extremo).

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quick sort: análise de caso médio

Proposição: O número médio de comparações C_N para ordenar um *array* de N chaves distintas é $< 2N \lg N$.

Prova: Temos $C_0 = C_1 = 0$ e para C_N com $N \geq 2$, vale a recorrência:

$$C_N = (N+1) + \left(\frac{C_0 + C_{N-1}}{N}\right) + \cdots + \left(\frac{C_{N-1} + C_0}{N}\right) \quad .$$

Aonde o primeiro termo é a quantidade de comparações do particionamento atual e os demais termos são as diferentes partições que podem surgir, ponderadas pelas suas probabilidades.

Quick sort: análise de caso médio

Com alguma manipulação matemática, a relação de recorrência pode ser resolvida, dando como resultado:

$$C_N = 2(N+1)\Big(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{N+1}\Big) \quad .$$

A soma acima pode ser aproximada por uma integral:

$$C_N \sim 2(N+1)\int_3^{N+1} \frac{1}{x}dx \quad .$$

Levando enfim ao resultado final:

$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N \quad .$$

Quick sort: sumário de desempenho

Quick sort é um algoritmo randomizado.

- Com corretude garantida.
- Mas com tempo de execução dependente do *shuffle*.

Caso médio: # esperado de comparações é $\sim 1.39 N \lg N$.

- 39% mais comparações que o *merge sort*.
- Mas mais rápido na prática devido a menos movimentação dos dados.

Melhor caso: número de comparações é $\sim N \lg N$.

Pior caso: número de comparações é $\sim {}^{1}\!/_{2} N^2$. (Mas é mais provável que caia um raio no computador durante a execução!)

Lição: análise empírica é fundamental!

Quick sort: propriedades

Quick sort é um algoritmo de ordenação *in-place*.

- Particionamento: espaço extra constante.
- Profundidade da recursão: espaço extra logarítmico (com alta probabilidade).
- É possível garantir profundidade logarítmica mas requer o uso explícito de uma pilha. (Versão não-recursiva adiante.)

Quick sort não é estável.

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quick sort: melhorias práticas

Use *insertion sort* para *sub-arrays* pequenos.

- *Quick sort* também tem um *overhead* muito grande para *sub-arrays* pequenos.
- *Cutoff* para *insertion sort* quanto o *array* tiver ≈ 10 itens.

```
void quick_sort(Item *a, int lo, int hi) {
     if (hi <= lo + CUTOFF - 1) {
        insert_sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
}
```

Quick sort: melhorias práticas

Mediana de uma amostra.

- Melhor escolha do pivô de particionamento: mediana do *array*.
- Estimar a mediana real tomando a mediana de uma amostra aleatória.
- Mediana de 3 chaves aleatórias.
- 14% menos comparações. 3% mais trocas.

```
void quick_sort(Item *a, int lo, int hi) {
     if (hi <= lo + CUTOFF - 1) {
        insert_sort(a, lo, hi);
        return;
    }
    int median = median_of_3(a, lo, hi);
    exch(a[lo], a[median]);
    int j = partition(a, lo, hi);
    quick_sort(a, lo, j-1);
    quick_sort(a, j+1, hi);
}
```

Quick sort não-recursivo

Versão não-recursiva do *quick sort* usa uma pilha auxiliar.

```
#define push2(A, B) push(B); push(A)

void quick_sort(Item *a, int lo, int hi) {
    stack_init();
    push2(lo, hi);
    while(!stack_empty()) {
        lo = pop(); hi = pop();
        if (hi <= lo) continue; // Could add cutoff here.
        int i = partition(a, lo, hi);
        if (i-lo > hi-i) {  // Test the size of sub-arrays.
            push2(lo, i-1); // Push the larger one.
            push2(i+1, hi); // Sort the smaller one first.
        } else {
            push2(i+1, hi);
            push2(lo, i-1);
        }
    }
}
```

Ordenar o menor *sub-array* primeiro garante que a pilha nunca fica com mais de lg N elementos durante a execução.

Chaves duplicadas

Em geral, o propósito da ordenação é agrupar chaves iguais.
Exemplos:

- Ordenar população por idade.
- Remover duplicatas de uma lista de contatos.
- Ordenar alunos por nota alcançada.

Características típicas dessas aplicações:

- Array muito grande.
- Número pequeno de chaves.
- $\Rightarrow$ Muitas chaves duplicadas!

História de guerra: *system sort* em C

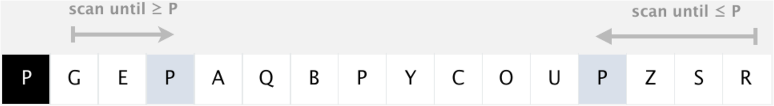
Bug report [Allan Wilks & Rick Becker, 1991]:

```
We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

main (int argc, char**argv) {
   int n = atoi(argv[1]), i, x[100000];
   for (i = 0; i < n; i++)
      x[i] = i;
   for ( ; i < 2*n; i++)
      x[i] = 2*n-i-1;
   qsort(x, 2*n, sizeof(int), intcmp);
}

Here are the timings on our machine:
$ time a.out 2000
real    5.85s
$ time a.out 4000
real   21.64s
$time a.out 8000
real   85.11s
```

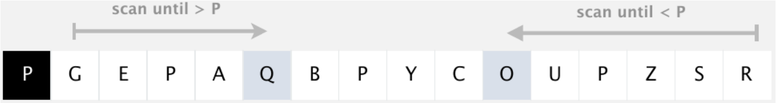
Na época, implementações de `qsort` no Unix era quadráticas para *arrays* com muitas duplicatas.

Chaves duplicadas: parar em chaves iguais

Nossa função de particionamento encerra ambos os *scans* em chaves iguais.

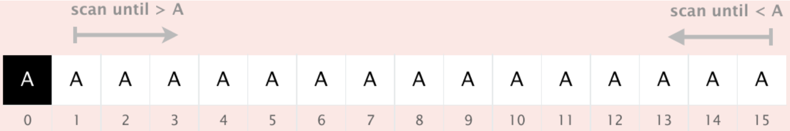


Q: Por que não continuar o *scan* sobre chaves iguais?



Chaves duplicadas: parar em chaves iguais

Q: Qual é o resultado do particionamento do *array* abaixo quando continuamos o *scan* sobre chaves iguais?



Q: Qual é o resultado do particionamento do *array* abaixo quando paramos o *scan* em chaves iguais?

Particionando um *array* com todas chaves iguais

		a[ ]															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Chaves duplicadas: estratégias de particionamento

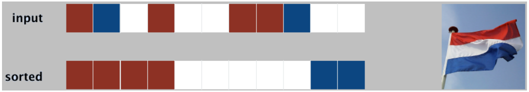
Ruim: continuar o *scan* sobre chaves iguais.
$\sim 1/2N^2$ comparações quando todas as chaves são iguais.

Bom: parar o *scan* em chaves iguais.
$\sim N \lg N$ comparações quando todas as chaves são iguais.

Melhor: Colocar todas as chaves iguais em posição. Como?
$\sim N$ comparações quando todas as chaves são iguais.

Dutch National Flag Problem

Problema [Edsger Dijkstra, 1976]: Dado um *array* de N posições, cada qual contendo um elemento vermelho, branco ou azul, ordená-los por cor.



Operações permitidas:

- `swap(i,j)`: troca elementos nas posições i e j.
- `color(i)`: cor na posição i.

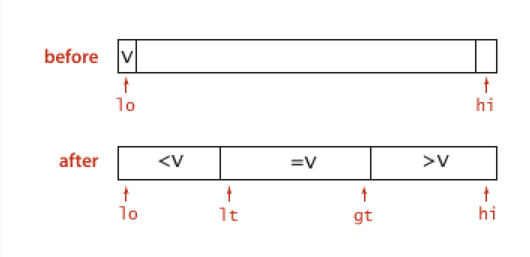
Requisitos da solução:

- Exatamente N chamadas de `color()`.
- No máximo N chamadas de `swap()`.
- Espaço extra constante.

Particionamento *3-way*

Objetivo: Particionar o *array* em três partes aonde:

- Chaves entre `lt` e `gt` são iguais o pivô de particionamento.
- Não há chaves maiores que o pivô à esquerda de `lt`.
- Não há chave menores que o pivô à direita de `gt`.

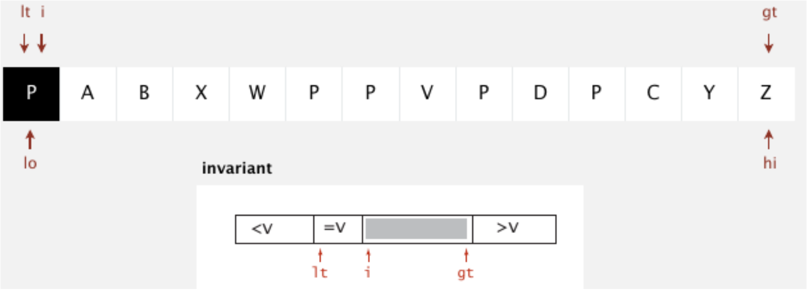


Dutch National Flag Problem [Edsger Dijkstra, 1976]:

- Consenso até ∼ 1995: não vale a pena.
- Agora incorporado em praticamente todos os códigos.

Particionamento *3-way* de Dijkstra

- Seja `v` o pivô em `a[lo]`.
- Percorra `i` da esquerda para a direita.
  - `a[i] < v`: troque `a[lt]` e `a[i]`, incrementando ambos os índices.
  - `a[i] > v`: troque `a[gt]` e `a[i]`, decrementando `gt`.
  - `a[i] == v`: incremente `i`.

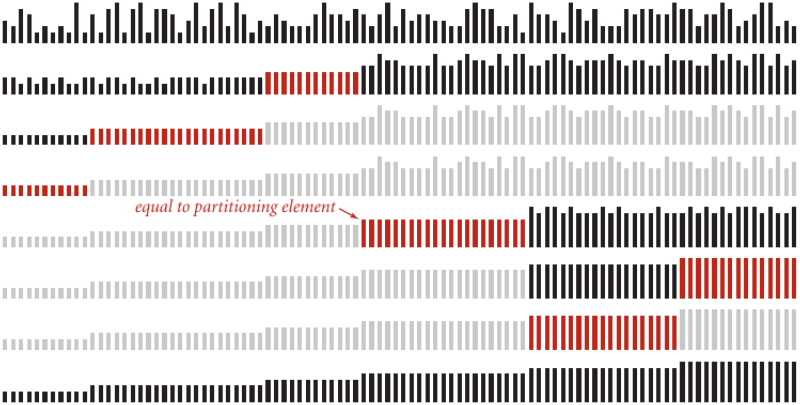


Ver arquivo `23DemoPartitioning.pdf`, segundo exemplo.

3-way quicksort: implementação em C

```
void quick_sort(Item *a, int lo, int hi) {
    if (hi <= lo) return;
    Item v = a[lo];
    int lt = lo, gt = hi, i = lo;
    while (i <= gt) {
        if (a[i] < v) {
            exch(a[lt], a[i]);
            lt++; i++;
        } else if (a[i] > v) {
            exch(a[i], a[gt]);
            gt--;
        } else {
            i++;
        }
    }
    quick_sort(a, lo, lt-1);
    quick_sort(a, gt+1, hi);
}
```

3-way quicksort: trace



Chaves duplicadas: *lower bound*

Lower bound (LU) da ordenação: Se existem N chaves distintas e a i-ésima chave ocorre x_i vezes, então qualquer método de ordenação baseado em comparações precisa fazer pelo menos

$$\lg\left(\frac{N!}{x_1!x_2!\cdots x_n!}\right)$$

comparações no pior caso.

- Todas as chaves distintas: $N \lg N$.
- Número constante de chaves distintas: N.

Proposição: O número esperado de comparações do *quicksort 3-way* é proporcional ao LU da ordenação.

Prova: além do escopo deste curso.

Conclusão: *Quick sort* com particionamento *3-way* reduz o tempo de execução de $N \lg N$ para N em uma série de casos.

Aplicações de ordenação

Algoritmos de ordenação são essenciais em várias aplicações.

Aplicações óbvias:

- Ordenar uma lista de nomes.
- Organizar uma biblioteca de músicas.
- Mostrar o resultado de uma busca.

Problemas que se tornam simples após ordenação:

- Encontrar a mediana.
- Identificar *outliers* estatísticos.
- Busca binária em uma base de dados.
- Encontrar duplicatas em uma lista.

Aplicações não óbvias:

- Compressão de dados.
- Computação gráfica.
- Biologia computacional.
- Balanceamento de carga em um computador paralelo.
- ...

A engenharia de um *system sort* (em 1993)

Bentley-McIlroy *quick sort*.

- *Cut-off* para *insertion sort* para *sub-arrays* pequenos.
- Pivô de particionamento: mediana de 3.
- Esquema de particionamento: Bentley-McIlroy *3-way*. (Similar o esquema de Dijkstra mas com menos trocas quando não há muitas chaves iguais.)

Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

SUMMARY

We recount the history of a new `qsort` function for a C library. Our function is clearer, faster and more robust than existing sorts. It chooses partitioning elements by a new sampling scheme; it partitions by a novel solution to Dijkstra's Dutch National Flag problem; and it swaps efficiently. Its behavior was assessed with timing and debugging testbeds, and with a program to certify performance. The design techniques apply in domains beyond sorting.

Amplamente utilizado: C, C++, Java 6, ...

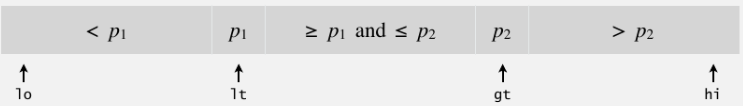
Dual-pivot quicksort (Yaroslavskiy, 2009)

Dual-pivot quicksort: proposto por Vladimir Yaroslavskiy para substituir a implementação do *quick sort* no Java 6.

Usa dois pivôs p_1 e p_2 para particionar em três *sub-arrays*:

- Chaves menores que p_1.
- Chaves entre p_1 e p_2.
- Chaves maiores que p_2.

Ordena recursivamente os três *sub-arrays*.



Pula o *sub-array* do meio se $p_1 = p_2$: degenera para o particionamento de Dijkstra.

Amplamente utilizado: Java 7, Python, Android, ...

Multi-pivot quicksort

Multi-Pivot Quicksort: Theory and Experiments

Shrinu Kushagra
skushagr@uwaterloo.ca
University of Waterloo

Alejandro López-Ortiz
alopez-o@uwaterloo.ca
University of Waterloo

J. Ian Munro
imunro@uwaterloo.ca
University of Waterloo

Aurick Qiao
a2qiao@uwaterloo.ca
University of Waterloo

Uso de três pivôs pode ser mais eficiente em poucos casos.
Na prática não se justifica usar mais que dois.

Q: Por que usar 2 ou 3 pivôs é melhor que 1?
A: Menos *cache misses*, mas é preciso testar em cada sistema!

System sort no Java 7

Arrays.sort():

- Um método para objetos do tipo Comparable.
- Um método sobrecarregado para cada tipo primitivo.

Algoritmos:

- *Dual-pivot quick sort* para tipos primitivos.
- *Tim sort* para objetos.

Q: Por que usar diferentes algoritmos para tipos primitivos e objetos?

- *Tim sort* é estável, mantém a ordem relativa dos objetos a serem ordenados.
- Custo adicional de espaço N não é tão relevante: provavelmente os objetos ocupam muito mais espaço.
- Tipos primitivos não têm identidade: não há problema em usar um algoritmo não-estável.
- Variantes de *merge sort* dobrariam o uso de memória.

Resumo dos métodos de ordenação vistos até agora

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail