

Estrutura de Dados II (ED2)

Aula 11 – Merge sort

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

(Material baseado nos slides do Professor Eduardo Zambon)

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

Referências

Chapter 8 – Merging and Mergesort

R. Sedgewick

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

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

Merge sort

Ideia geral.

- Dividir o *array* em duas metades.
- Ordenar **recursivamente** cada metade.
- Unificar (*merge*) as duas metades.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Objetivo.

Dados dois *sub-arrays* ordenados de $a[lo]$ até $a[mid]$ e de $a[mid+1]$ até $a[hi]$, substituí-los pelo *array* ordenado de $a[lo]$ até $a[hi]$.

Ver arquivo 22DemoMerge.mov <https://algs4.cs.princeton.edu/lectures/demo/22DemoMerge.mov>

Merge: implementação em C

```
void merge(Item *a, Item *aux, int lo, int mid, int hi) {  
    for (int k = lo; k <= hi; k++)  
        aux[k] = a[k];           // Copy array  
    int i = lo, j = mid+1;  
    for (int k = lo; k <= hi; k++) { // Merge  
        if (i > mid)                a[k] = aux[j++];  
        else if (j > hi)            a[k] = aux[i++];  
        else if (less(aux[j], aux[i])) a[k] = aux[j++];  
        else                        a[k] = aux[i++];  
    }  
}
```

	lo			i		mid	j			hi
aux[]	A	G	L	O	R	H	I	M	S	T
	k									
a[]	A	G	H	I	L	M				

Merge sort: implementação em C

```
void merge_sort(Item *a, Item *aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2; // Avoid overflow.
    merge_sort(a, aux, lo, mid);
    merge_sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

void sort(Item *a, int lo, int hi) {
    int n = (hi - lo) + 1;
    Item *aux = malloc(n * sizeof(Item));
    merge_sort(a, aux, lo, hi);
    free(aux);
}
```

Para usar: `sort(a, 0, N-1);`

Veja as animações em:

<https://www.toptal.com/developers/sorting-algorithms/merge-sort>

Merge sort: trace

	a[]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

result after recursive call

Merge 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 ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Conclusão: bons algoritmos são melhores que supercomputadores.

Merge sort: número de comparações

Proposição: *Merge sort* usa no máximo $N \lg N$ comparações para ordenar um *array* de tamanho N .

Ideia: O número de comparações $C(N)$ satisfaz a recorrência:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N, \text{ com } C(1) = 0 \quad .$$

Aonde os dois primeiros termos são os tamanhos das metades e o último termo é a operação de *merge*.

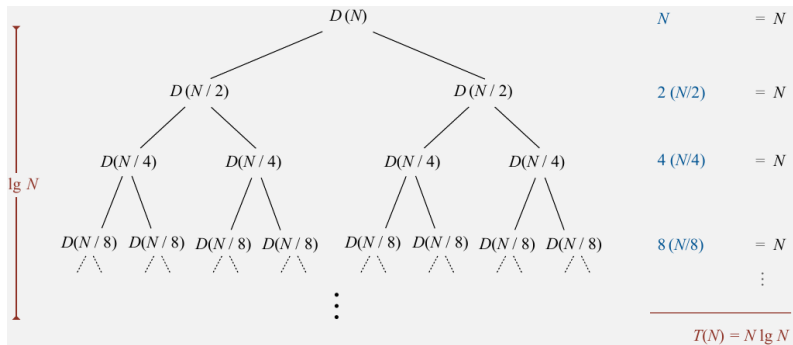
Quando N é uma **potência de 2**, a recorrência fica mais simples:

$$D(N) = 2D(N/2) + N, \text{ com } D(1) = 0 \quad .$$

Recorrência de divisão e conquista: “prova” por figura

Proposição: se $D(N) = 2D(N/2) + N$, com $D(1) = 0$, então $D(N) = N \lg N$.

Ideia:



Resultado **também vale** para quando N **não é** uma potência de 2. Prova por indução (mais complicada).

Merge sort: número de acessos ao *array*

Proposição: *Merge sort* usa no máximo $6N \lg N$ acessos para ordenar um *array* de tamanho N .

Ideia: O número $A(N)$ de acessos ao *array* satisfaz a recorrência:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N, \text{ com } A(1) = 0 \quad .$$

Ponto fundamental. Qualquer algoritmo com a estrutura abaixo leva tempo $N \log N$:

```
void linearithmic(int N) {  
    if (N == 0) return;  
    linearithmic(N/2); // Resolver dois problemas com  
    linearithmic(N/2); // metade do tamanho original.  
    linear(N);         // Quantidade de trabalho linear.  
}
```

Merge sort: memória

Proposição: *Merge sort* usa **espaço extra** proporcional a N .

Justificativa: O array `aux[]` deve ter tamanho N para se realizar a última operação de *merge*.

Definição: Um algoritmo de ordenação é ***in-place*** (ou ***in-situ***) se ele utiliza $\leq c \log N$ de memória extra.

Exemplos: *selection sort*, *insertion sort*, *shell sort*.

Desafio 1 (fácil): Utilizar `aux[]` com tamanho $\sim 1/2N$.

Desafio 2 (muito difícil): *in-place merge* [Kronrod 1969].

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

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

```
void merge_sort(Item *a, Item *aux, int lo, int hi) {  
    if (hi <= lo + CUTOFF - 1) {  
        insert_sort(a, lo, hi);  
        return;  
    }  
    int mid = lo + (hi - lo) / 2;  
    merge_sort(a, aux, lo, mid);  
    merge_sort(a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

Merge sort: melhorias práticas

Parar quando já está ordenado.

- O maior item da primeira metade \leq que o menor item da segunda metade?
- Se sim, não precisa fazer *merge*.
- Ajuda no desempenho para entradas **parcialmente ordenadas**.

```
void merge_sort(Item *a, Item *aux, int lo, int hi) {  
    if (hi <= lo + CUTOFF - 1) {  
        insert_sort(a, lo, hi);  
        return;  
    }  
    int mid = lo + (hi - lo) / 2;  
    merge_sort(a, aux, lo, mid);  
    merge_sort(a, aux, mid+1, hi);  
    if (!less(a[mid+1], a[mid])) return;  
    merge(a, aux, lo, mid, hi);  
}
```


Bottom-up merge sort

Bottom-up merge sort

Ideia geral.

- Percorre *array* com *merge* para *sub-arrays* de tamanho 1.
- Repita para *sub-arrays* de tamanho 2, 4, 8,

				a[i]															
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz = 1				M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)				E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)				E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)				E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)				E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)				E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)				E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 12, 12, 13)				E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 14, 14, 15)				E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																			
merge(a, aux, 0, 1, 3)				E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)				E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)				E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)				E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																			
merge(a, aux, 0, 3, 7)				E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)				E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																			
merge(a, aux, 0, 7, 15)				A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

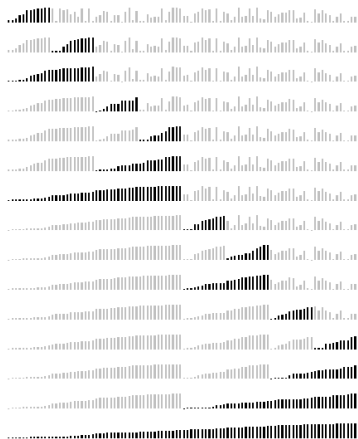
Bottom-up merge sort: implementação em C

```
#define SZ2      (sz+sz)
#define MIN(X,Y) ((X < Y) ? (X) : (Y))

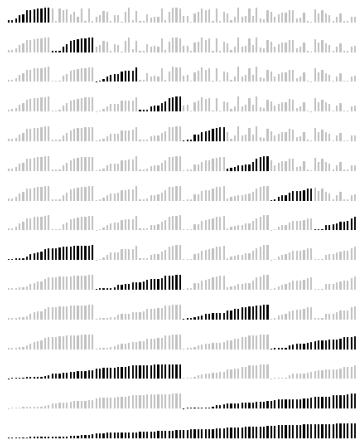
void sort(Item *a, int lo, int hi) {
    int N = (hi - lo) + 1;
    int y = N - 1;
    Item *aux = malloc(N * sizeof(Item));
    for (int sz = 1; sz < N; sz = SZ2) {
        for (int lo = 0; lo < N-sz; lo += SZ2) {
            int x = lo + SZ2 - 1;
            merge(a, aux, lo, lo+sz-1, MIN(x,y));
        }
    }
    free(aux);
}
```

- Versão simples **não-recursiva** do *merge sort*.
- Mas geralmente **mais lenta** que a versão *top-down* recursiva na maioria dos sistemas. (Próximo Lab!)

Merge sort: visualizações



top-down mergesort (cutoff = 12)



bottom-up mergesort (cutoff = 12)

Merge sort natural

Ideia: Explorar ordem pré-existente através da identificação de *runs* que ocorrem naturalmente.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

merge two runs

1	3	4	5	10	16	23	9	13	2	7	8	12	14
---	---	---	---	----	----	----	---	----	---	---	---	----	----

Tradeoff: Menos passadas vs. mais comparações por passada.

- *Merge sort* natural.
- Utiliza *insertion sort* para criar *runs* iniciais (se necessário).
- Mais algumas otimizações espertas.

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than $\lg(N!)$ comparisons needed, and as few as $N-1$), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

...



Tim Peters

Consequência: Tempo linear em vários *arrays* com uma pré-ordem existente.

Amplamente utilizado atualmente: Python, Java 7, Octave,

Complexidade do problema de ordenação

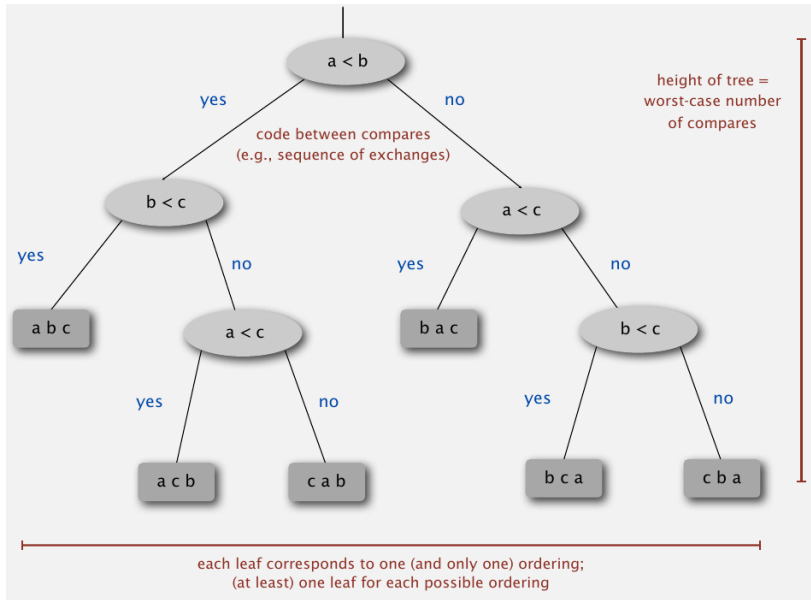
Complexidade computacional: Estudo da eficiência de algoritmos para se resolver um dado problema X .

- **Modelo de computação:** operações permitidas.
- **Modelo de custo:** operações (relevantes) contadas.
- **Upper bound (UB):** Garantia de custo provida por **algum** algoritmo para X .
- **Lower bound (LB):** Limite provado de custo de **todos** os algoritmos para X .
- **Algoritmo ótimo:** algoritmo com a melhor garantia de custo para X ($UB = LB$).

Exemplo: Ordenação.

- **Modelo de computação:** árvore de decisão.
- **Modelo de custo:** número de comparações.
- **UB:** $\sim N \lg N$ (*merge sort*).
- **LB:** ?
- **Algoritmo ótimo:** ?

Árvore de decisão (para 3 chaves distintas a , b e c)



LB para ordenação baseada em comparações

Proposição: Qualquer algoritmo de ordenação baseado em comparações precisa realizar no mínimo $\lg(N!) \sim N \lg N$ comparações no **pior caso**.

Justificativa:

- Assuma que o *array* possui N valores **distintos**.
- Pior caso é dado pela **altura h** da árvore de decisão.
- Árvore binária com altura h tem no máximo 2^h folhas.
- **$N!$** ordenações distintas \Rightarrow no mínimo $N!$ folhas:

$$2^h \geq \# \text{ folhas} \geq N! \quad \Rightarrow \quad h \geq \lg(N!) \quad .$$

- O LB ocorre quando **$h = \lg(N!)$** .
- **Fórmula de Stirling:** $\lg(N!) \sim N \lg N$.

Complexidade do problema de ordenação

Exemplo: Ordenação.

- **Modelo de computação:** árvore de decisão.
- **Modelo de custo:** número de comparações.
- **UB:** $\sim N \lg N$ (*merge sort*).
- **LB:** $\sim N \lg N$.
- **Algoritmo ótimo:** *merge sort*.

Objetivo do projeto de algoritmos: encontrar algoritmos ótimos.

Resultados de complexidade em contexto

Q: Na aula anterior vimos que o *insertion sort* pode executar em tempo N em algumas situações. Por que isso não contradiz o LB encontrado nos slides anteriores?

A: Porque *insertion sort* é N no **melhor caso**. O LB encontrado de $N \lg N$ é para o **pior caso**.

Comparações? *Merge sort* é ótimo quanto ao número de comparações.

Espaço? *Merge sort* não é ótimo quanto ao espaço utilizado.

Lições: Usar a teoria como guia.

Ex.: Projetar um algoritmo que é ótimo para tempo e espaço?

Aplicação típica: Ordene por nome, e **a seguir** por seção.

Sort by Name

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Sort by Section

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Pessoas da seção 3 não estão mais ordenadas por nome.

Um método de ordenação **estável** preserva a ordem relativa dos itens com chaves iguais.

Estabilidade

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

Diagram illustrating the stability of sorting algorithms. The table shows three columns: 'sorted by time', 'sorted by location (not stable)', and 'sorted by location (stable)'. Red arrows indicate the mapping of elements between the columns. The 'sorted by location (not stable)' column shows elements that are no longer sorted by time, while the 'sorted by location (stable)' column shows elements that remain sorted by time.

no longer sorted by time

still sorted by time

Q: Quais *sorts* são estáveis?

A: É preciso verificar o algoritmo (e a implementação).

Estabilidade: *insertion sort*

Insertion sort é **estável**.

```
for (int i = 0; i < N; i++)  
    for (int j = i; j > 0 && less(a[j], a[j-1]); j--)  
        exch(a, j, j-1);  
}
```

i	j	0	1	2	3	4
0	0	B ₁	A ₁	A ₂	A ₃	B ₂
1	0	A ₁	B ₁	A ₂	A ₃	B ₂
2	1	A ₁	A ₂	B ₁	A ₃	B ₂
3	2	A ₁	A ₂	A ₃	B ₁	B ₂
4	4	A ₁	A ₂	A ₃	B ₁	B ₂
		A ₁	A ₂	A ₃	B ₁	B ₂

Item iguais nunca são movidos além da sua posição inicial relativa.

Estabilidade: *selection sort*

Selection sort não é estável.

```
for (int i = 0; i < N; i++)  
{  
    int min = i;  
    for (int j = i+1; j < N; j++)  
        if (less(a[j], a[min]))  
            min = j;  
    exch(a, i, min);  
}
```

i	min	0	1	2
0	2	B ₁	B ₂	A
1	1	A	B ₂	B ₁
2	2	A	B ₂	B ₁
		A	B ₂	B ₁

Trocas entre posições distantes podem embaralhar a posição inicial relativa de chaves iguais.

Shell sort não é estável.

```
while (h >= 1)
{
    for (int i = h; i < N; i++)
    {
        for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
            exch(a, j, j-h);
    }
    h = h/3;
}
```

h	0	1	2	3	4
	B ₁	B ₂	B ₃	B ₄	A ₁
4	A ₁	B ₂	B ₃	B ₄	B ₁
1	A ₁	B ₂	B ₃	B ₄	B ₁
	A ₁	B ₂	B ₃	B ₄	B ₁

Mesma justificativa do *selection sort*.

Estabilidade: *merge sort*

Merge sort é **estável** porque a operação de *merge* é estável.

```
for (int k = lo; k <= hi; k++)  
    aux[k] = a[k];  
  
int i = lo, j = mid+1;  
for (int k = lo; k <= hi; k++)  
{  
    if      (i > mid)           a[k] = aux[j++];  
    else if (j > hi)           a[k] = aux[i++];  
    else if (less(aux[j], aux[i])) a[k] = aux[j++];  
    else                       a[k] = aux[i++];  
}  
}
```

0	1	2	3	4	5	6	7	8	9	10
A ₁	A ₂	A ₃	B	D	A ₄	A ₅	C	E	F	G

Sempre toma elementos da esquerda enquanto as chaves forem iguais.

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
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail