Ordenação de dados eficiente

Profª. Rose Yuri Shimizu

Roteiro

Algoritmos de Ordenação Eficientes

- Linearítmicos
  - $O(n \log n)$
  - Melhor custo quando a ordenação é por comparação do valor da chave
  - Vantagem: mais amplo (vários tipos de chaves podem usar o mesmo algoritmo)
- Lineares
  - $O(n)$
  - Melhor custo quando a ordenação é por comparação na estrutura da chave:
    - intervalo de chaves de 0 até R-1
    - inteiros de 32 bits
  - Desvantagem: mais restrito
    - amarrado ao um tipo de chave (estrutura da chave)

Roteiro

Algoritmos de Ordenação Eficientes - Merge Sort

- Método dividir e conquistar
  - Dividir em pequenas partes
  - Ordenar essas partes
  - Combinar essas partes ordenadas
  - Até formar uma única sequência ordenada

Algoritmos de Ordenação Eficientes - Merge Sort

- Abordagem Top-Down: a partir da lista inteira, dividir em sub-listas
- Recursivamente:
  - A cada chamada, divide a entrada em sub-vetores para serem ordenados
    - `merge_sort(int *v, int l, int r)`
  - Quando chegar em um tamanho unitário, está ordenado em 1
  - Volta fazendo o *merge*, a intercalação, do ordenado
    - `merge(int *v, int l, int meio, int r)`
    - Utiliza um vetor auxiliar

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l				m					r
0	1	2	3	4	5	6	7	8	9
[7	2	9	10	4	3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l				m					r
0	1	2	3	4	5	6	7	8	9
[7	2	9	10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l		m		r					
0	1	2	3	4	5	6	7	8	9
[7	2	9	10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l		m		r					
0	1	2	3	4	5	6	7	8	9
[7	2	9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l	m	r							
0	1	2	3	4	5	6	7	8	9
[7	2	9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l	m	r							
0	1	2	3	4	5	6	7	8	9
[7	2]	[9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l=m	r								
0	1	2	3	4	5	6	7	8	9
[7	2]	[9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l=m	r								
0	1	2	3	4	5	6	7	8	9
[7]	[2]	[9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

l=r	l=r								
0	1	2	3	4	5	6	7	8	9
[7]	[2]	[9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l=m	r								
0	1	2	3	4	5	6	7	8	9
[7]	[2]	[9]	[10	4]	[3	1	8	6	5]
i	j								

[	]
k	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

```
l=m   r
 0    1    2    3    4    5    6    7    8    9
[7]  [2]  [9]  [10   4]  [3    1    8    6    5]
 i    j++

[2       ]
 k++
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l=m	r								
0	1	2	3	4	5	6	7	8	9
[7]	[2]	[9]	[10	4]	[3	1	8	6	5]
i		j>r							

[2]

k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

l=m	r								
0	1	2	3	4	5	6	7	8	9
[7]	[2]	[9]	[10	4]	[3	1	8	6	5]
i++									

[2	7]
	k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

```
l=m    r
 0     1     2     3     4     5     6     7     8     9
[7]   [2]   [9]  [10     4]   [3     1     8     6     5]
   i>m


[2     7]
         k
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

l=m	r								
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]

$\Uparrow$

[2 7]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
i		j							

```
[          ]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
i++		j							

```
[2          ]
 k++
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
	i	j							

[2		]
	k	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
	i++	j							

[2	7	]
	k++	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
	i>m	j							

[2	7	]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
		j++							

[2	7	9]
		k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7]	[9]	[10	4]	[3	1	8	6	5]
		j>r							

[2	7	9]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

l	m	r							
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10	4]	[3	1	8	6	5]
	$\Uparrow$								
[2	7	9]							

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

			l=m	r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

			l=m	r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

			l=r	l=r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]
			i	j					

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

			l=r	l=r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]
			i	j++					

[4]
k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

			l=r	l=r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]
			i		j>r				
			[4	]					
				k					

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

```
                        l=r   l=r
0     1     2     3     4     5     6     7     8     9
[2    7     9]   [10]  [4]   [3     1     8     6     5]
                   i++

                 [4     10]
                         k++
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

			l=r	l=r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]
			i>m						

[4 10]

k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

			l=r	l=r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]

$\Uparrow$

[4 10]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
i			j						

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
i++			j						

```
[2                   ]
 k++
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
	i		j						

[2				]
	k			

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
	i		j++						

[2	4			]
	k++			

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
	i			j					

[2	4			]
		k		

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
	i++			j					

[2	4	7		]
		k++		

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
		i		j					

[2	4	7		]
			k	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

```
 l         m         r
 0    1    2    3    4    5    6    7    8    9
[2    7    9]  [4   10]  [3    1    8    6    5]
           i++       j

[2    4    7    9     ]
                k++
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
			i>m	j					

[2	4	7	9	]
				k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[4	10]	[3	1	8	6	5]
				j++					

[2	4	7	9	10]
				k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

```
l          m          r
0    1     2     3    4     5    6    7    8    9
[2   7     9]    [4   10]   [3   1    8    6    5]
                        j>r

[2   4     7     9    10]
                        k
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

l		m		r					
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1	8	6	5]

$\Uparrow$

[2	4	7	9	10]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1	8]	[6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

					l=m	r	l=r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1]	[8]	[6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

					l=r	l=r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i	j			

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i	j++			

[1]

k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i		j>r		

[1	]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i++				

[1	3]
	k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i>m				

[1	3]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]

$\Uparrow$

[1 3]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
					i		j		

```
[          ]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
					i++		j		
					[1		]		
					k++				

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
						i	j		

[1			]
	k		

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
						i++	j		
					[1	3	]		
						k++			

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
						i>m	j		

[1	3	]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
							j++		

[1	3	8]
		k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3]	[8]	[6	5]
								j>r	

[1	3	8]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6	5]

$\Uparrow$

[1 3 8]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6	5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. **Dividir** : `m = l+(r-l)/2 = (l+r)/2`
2. Ordenar e juntar

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]
								i	j++

[5	]
k++	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]
								i	j>r

[5	]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]
								i++	
								[5	6]
									k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]
								i>m	

[5	6]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]

⇑

[5 6]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
					i			j	

[				]
k				

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
					i++			j	
					[1				]
					k++				

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
						i++		j	
					[1	3			]
						k++			

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i	j++	
					[1	3	5		]
							k++		

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i		j++

[1	3	5	6	]
			k++	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i		j>r
					[1	3	5	6	]
									k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i++		

					[1	3	5	6	8]
									k++

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

					l		m		r	
0	1	2	3	4	5	6	7	8	9	
[2	4	7	9	10]	[1	3	8]	[5	6]	
							i>m			
					[1	3	5	6	8]	
										k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8	5	6]
							⇑		
					[1	3	5	6	8]

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l				m					r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	5	6	8]
i					j				

[									]
k									

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Ordenar e juntar** : `v[i]<v[j] ? v[i]:v[j]`

l				m					r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	5	6	8]
			i						j>r

[1	2	3	4	5	6	7	8		]
								k	

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Juntar o restante** : `a[k] = v[i]`

l				m					r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	5	6	8]
					i>m				

[1	2	3	4	5	6	7	8	9	10]	
										k

Algoritmos de Ordenação Eficientes - Merge Sort

1. Dividir : `m = l+(r-l)/2 = (l+r)/2`
2. **Copia de volta**

l				m					r
0	1	2	3	4	5	6	7	8	9
[1	2	3	4	5	6	7	8	9	10]

$\Uparrow$

[1	2	3	4	5	6	7	8	9	10]

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
```

Algoritmos de Ordenação Eficientes

```
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;
```

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);
```

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);
    merge_sort(v, m+1, r);
```

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);
    merge_sort(v, m+1, r);
    merge(v, l, m, r);
```

Algoritmos de Ordenação Eficientes

```
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);
    merge_sort(v, m+1, r);
    merge(v, l, m, r);
}
```

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`
    1. `m = (2+0)/2 = 1`
    2. `merge_sort(v, 0, 1) : esquerda`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`
    1. `m = (2+0)/2 = 1`
    2. `merge_sort(v, 0, 1) : esquerda`
        1. `m = (1+0)/2 = 0`
        2. `merge_sort(v, 0, 0) : esquerda`
        3. `merge_sort(v, 1, 1) : direita`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`
    1. `m = (2+0)/2 = 1`
    2. `merge_sort(v, 0, 1) : esquerda`
        1. `m = (1+0)/2 = 0`
        2. `merge_sort(v, 0, 0) : esquerda`
        3. `merge_sort(v, 1, 1) : direita`
        4. `merge(v, 0, 0, 1)`

           6 5 3 1 2 4 : 5 6

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`
   1. `m = (2+0)/2 = 1`
   2. `merge_sort(v, 0, 1) : esquerda`
      1. `m = (1+0)/2 = 0`
      2. `merge_sort(v, 0, 0) : esquerda`
      3. `merge_sort(v, 1, 1) : direita`
      4. `merge(v, 0, 0, 1)`
         6 5 3 1 2 4 : 5 6
   3. `merge_sort(v, 2, 2) : direita`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, 0, 5)`
2. `meio = (5+0)/2 = 2`
3. `merge_sort(v, 0, meio=2) : esquerda`
   1. `m = (2+0)/2 = 1`
   2. `merge_sort(v, 0, 1) : esquerda`
      1. `m = (1+0)/2 = 0`
      2. `merge_sort(v, 0, 0) : esquerda`
      3. `merge_sort(v, 1, 1) : direita`
      4. `merge(v, 0, 0, 1)`

         6 **5** 3 1 2 4 : 5 6
   3. `merge_sort(v, 2, 2) : direita`
   4. `merge(v, 0, 1, 2)`

      5 6 **3** 1 2 4 : 3

      5 6 3 1 2 4 : 3 5

      5 **6** 3 1 2 4 : 3 5 6

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
    1. `m = (5+3)/2 = 4`
    2. `merge_sort(v, 3, 4) : esquerda`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
   1. `m = (5+3)/2 = 4`
   2. `merge_sort(v, 3, 4) : esquerda`
      1. `m = (4+3)/2 = 3`
      2. `merge_sort(v, 3, 3) : esquerda`
      3. `merge_sort(v, 4, 4) : direita`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
   1. `m = (5+3)/2 = 4`
   2. `merge_sort(v, 3, 4) : esquerda`
      1. `m = (4+3)/2 = 3`
      2. `merge_sort(v, 3, 3) : esquerda`
      3. `merge_sort(v, 4, 4) : direita`
      4. `merge(v, 3, 3, 4)`

         3 5 6 **1** **2** 4 : 1 2

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
   1. `m = (5+3)/2 = 4`
   2. `merge_sort(v, 3, 4) : esquerda`
      1. `m = (4+3)/2 = 3`
      2. `merge_sort(v, 3, 3) : esquerda`
      3. `merge_sort(v, 4, 4) : direita`
      4. `merge(v, 3, 3, 4)`

         3 5 6 **1** **2** 4 : 1 2

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
   1. `m = (5+3)/2 = 4`
   2. `merge_sort(v, 3, 4) : esquerda`
      1. `m = (4+3)/2 = 3`
      2. `merge_sort(v, 3, 3) : esquerda`
      3. `merge_sort(v, 4, 4) : direita`
      4. `merge(v, 3, 3, 4)`

         3 5 6 **1** **2** 4 : 1 2
   3. `merge_sort(v, 5, 5) : direita`

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
   1. `m = (5+3)/2 = 4`
   2. `merge_sort(v, 3, 4) : esquerda`
      1. `m = (4+3)/2 = 3`
      2. `merge_sort(v, 3, 3) : esquerda`
      3. `merge_sort(v, 4, 4) : direita`
      4. `merge(v, 3, 3, 4)`

         3 5 6 **1** **2** 4 : 1 2
   3. `merge_sort(v, 5, 5) : direita`
   4. `merge(v, 3, 4, 5)`

      3 5 6 **1** 2 **4** : 1

      3 5 6 1 **2** **4** : 1 2

      3 5 6 1 2 **4** : 1 2 4

Algoritmos de Ordenação Eficientes

1. `merge_sort(v, meio+1=3, 5) : direita`
    1. `m = (5+3)/2 = 4`
    2. `merge_sort(v, 3, 4) : esquerda`
        1. `m = (4+3)/2 = 3`
        2. `merge_sort(v, 3, 3) : esquerda`
        3. `merge_sort(v, 4, 4) : direita`
        4. `merge(v, 3, 3, 4)`

           3 5 6 **1** **2** 4 : 1 2
    3. `merge_sort(v, 5, 5) : direita`
    4. `merge(v, 3, 4, 5)`

       3 5 6 **1** 2 **4** : 1

       3 5 6 1 **2** **4** : 1 2

       3 5 6 1 2 **4** : 1 2 4
2. `merge(v, 0, 2, 5)`

   3 5 6 **1** 2 4 : 1

   3 5 6 1 **2** 4 : 1 2

   3 5 6 1 2 **4** : 1 2 3

   3 **5** 6 1 2 **4** : 1 2 3 4

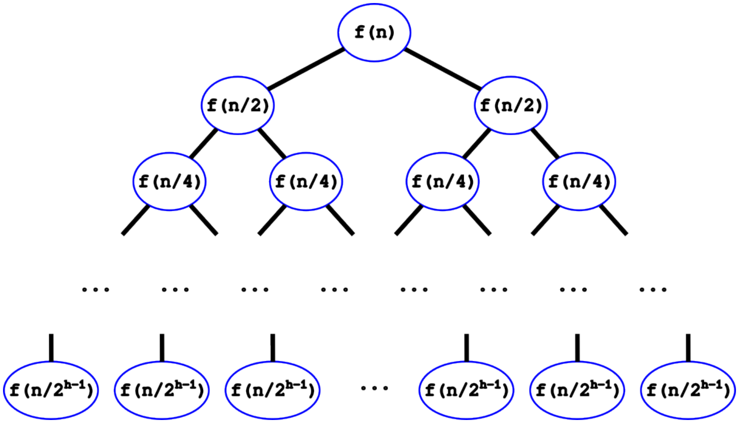
   3 **5** 6 1 2 4 : 1 2 3 4 5

   3 5 **6** 1 2 4 : 1 2 3 4 5

   3 5 6 1 2 4 : 1 2 3 4 5 6

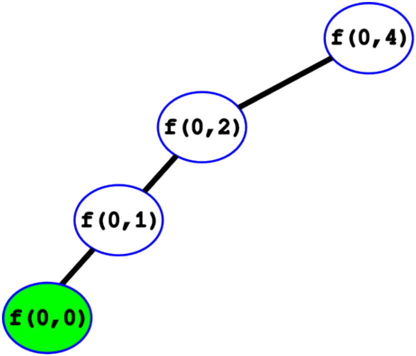
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



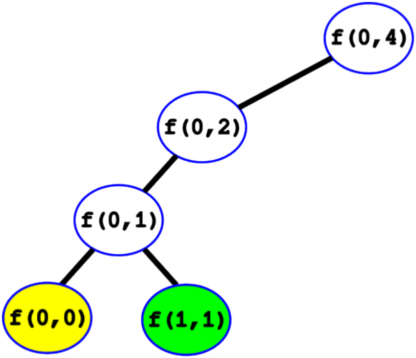
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



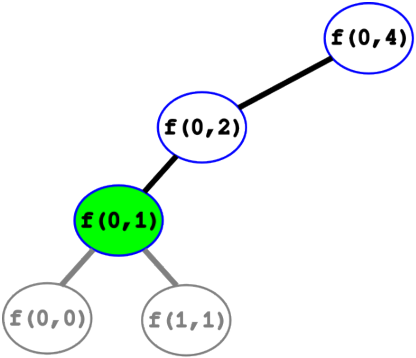
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



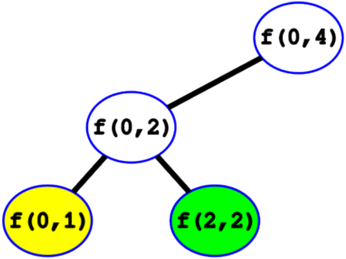
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



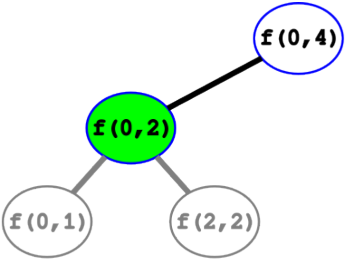
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



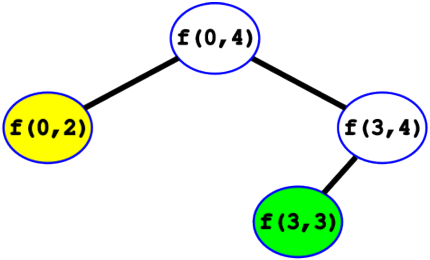
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



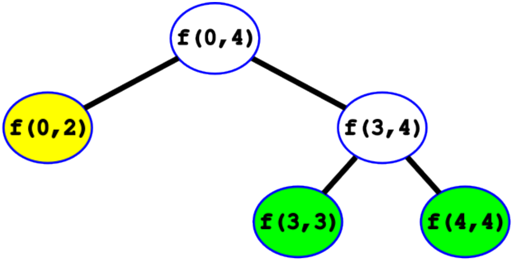
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



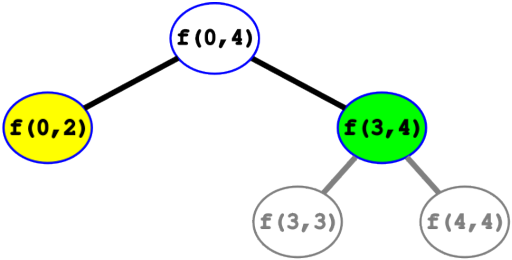
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



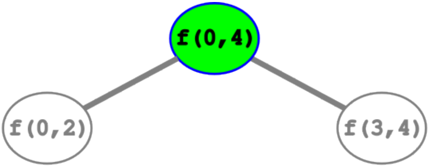
Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



Algoritmos de Ordenação Eficientes

- Árvore recursiva do merge



```c
void merge1(int *v,  int l, int m, int r) { //intercala
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;
  int *aux = malloc(sizeof(int)*tam);//espaço auxiliar
```

```c
void merge1(int *v,  int l, int m, int r) { //intercala
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;
  int *aux = malloc(sizeof(int)*tam);//espaço auxiliar

  int i=l;   //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0;   //inicio do vetor auxiliar

  while(i<=m && j<=r) { //percorrer os sub-vetores
```

```c
void merge1(int *v,  int l, int m, int r) { //intercala
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;
  int *aux = malloc(sizeof(int)*tam);//espaço auxiliar

  int i=l;    //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0;    //inicio do vetor auxiliar

  while(i<=m && j<=r) { //percorrer os sub-vetores
    if(v[i] <= v[j])     //testar sub-vetores
      aux[k++] = v[i++];//ordenar no vetor auxiliar
    else
      aux[k++] = v[j++];//ordenar no vetor auxiliar
  }

  //ainda tem elementos nos sub-vetores?
```

```c
void merge1(int *v,  int l, int m, int r) { //intercala
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;
  int *aux = malloc(sizeof(int)*tam);//espaço auxiliar

  int i=l;    //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0;    //inicio do vetor auxiliar

  while(i<=m && j<=r) { //percorrer os sub-vetores
    if(v[i] <= v[j])     //testar sub-vetores
      aux[k++] = v[i++];//ordenar no vetor auxiliar
    else
      aux[k++] = v[j++];//ordenar no vetor auxiliar
  }

  //ainda tem elementos nos sub-vetores?
  while(i<=m) aux[k++] = v[i++]; //consumir sub-vetor esquerdo
  while(j<=r) aux[k++] = v[j++]; //consumir sub-vetor direito

  //copiar para o vetor original
```

```c
void merge1(int *v,  int l, int m, int r) { //intercala
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;
  int *aux = malloc(sizeof(int)*tam);//espaço auxiliar

  int i=l;   //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0;   //inicio do vetor auxiliar

  while(i<=m && j<=r) { //percorrer os sub-vetores
    if(v[i] <= v[j])    //testar sub-vetores
      aux[k++] = v[i++];//ordenar no vetor auxiliar
    else
      aux[k++] = v[j++];//ordenar no vetor auxiliar
  }

  //ainda tem elementos nos sub-vetores?
  while(i<=m) aux[k++] = v[i++]; //consumir sub-vetor esquerdo
  while(j<=r) aux[k++] = v[j++]; //consumir sub-vetor direito

  //copiar para o vetor original
  for(k=0, i=l; i<=r; i++, k++) //v[i] e aux[k]
    v[i] = aux[k];  //copiar o aux[k] para v[i]

  //liberar memória
  free(aux);
}
```

```
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1;  //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
```

```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1;  //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
```

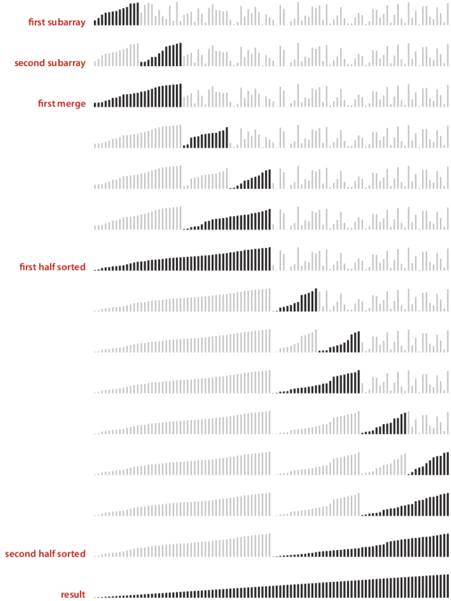
```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1; //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
      aux[k++] = v[j++]; //consome o direito
```

```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1; //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
      aux[k++] = v[j++]; //consome o direito

    else if (j>r) //terminou o sub-vetor direito?
```

```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1; //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
      aux[k++] = v[j++]; //consome o direito

    else if (j>r) //terminou o sub-vetor direito?
      aux[k++] = v[i++]; //consome o esquerdo
```

```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;   //sub-vetor esquerdo
  int j=m+1; //sub-vetor direito
  int k=0;   //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
      aux[k++] = v[j++]; //consome o direito

    else if (j>r) //terminou o sub-vetor direito?
      aux[k++] = v[i++]; //consome o esquerdo

    else if (v[i] <= v[j]) //testar sub-vetores
      aux[k++] = v[i++];

    else //if (v[j] < v[i])
      aux[k++] = v[j++];
  }
```

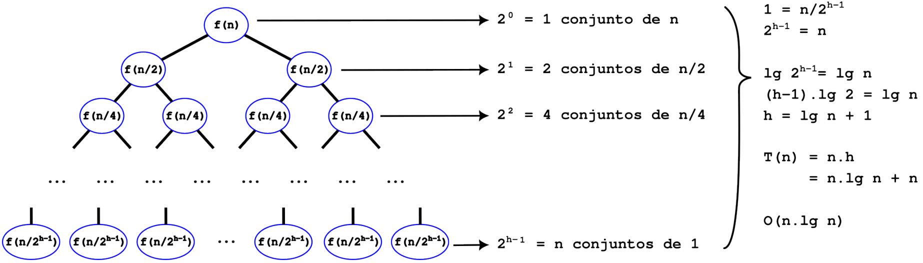
```c
void merge2(int *v, int l, int m, int r) { //intercala
  int tam = r+1-l;
  int *aux = malloc(tam*sizeof(int));

  int i=l;    //sub-vetor esquerdo
  int j=m+1;  //sub-vetor direito
  int k=0;    //vetor auxiliar

  //percorrer o vetor inteiro
  while(k<tam) {
    if(i>m) //terminou o sub-vetor esquerdo?
      aux[k++] = v[j++]; //consome o direito

    else if (j>r) //terminou o sub-vetor direito?
      aux[k++] = v[i++]; //consome o esquerdo

    else if (v[i] <= v[j]) //testar sub-vetores
      aux[k++] = v[i++];

    else //if (v[j] < v[i])
      aux[k++] = v[j++];
  }

  //copiar
  for(k=0, i=l; i<=r; i++)  v[i] = aux[k++];
  free(aux);
}
```



Visual trace of top-down mergesort with cutoff for small subarrays

Algoritmos de Ordenação Eficientes

- Cada merge/intercala, possui tempo linear
- A cada nível da árvore recursiva, a complexidade total (soma das chamadas recursivas) é igual ao tamanho total da entrada
  - $\sum_{i=0}^{h-1} \frac{n}{2^i} = n$, sendo n o tamanho da entrada e i um nível da árvore
- n itens são comparados/movimentados $\log n$ vezes



Algoritmos de Ordenação Eficientes

- Complexidade assintótica
  - Pior, Médio, Melhor caso: $O(n \log n)$

```
//não é necessário comparar todos entre si
//n/2 comparações e n movimentações
void merge(int *v, int l, int m, int r) {
    int tam = r+1-l;
    ...
    while(k<tam) { ... }  //n

}
```

```
void merge_sort(int *v, int l, int r) {
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);  //F(n/2)
    merge_sort(v, m+1, r); //F(n/2)
    merge(v, l, m, r); //n
}
```

Algoritmos de Ordenação Eficientes

$$
\begin{aligned}
f(n) &= 2 * f(\frac{n}{2}) + n \\
&= 2 * (2 * f(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 2^2 * f(\frac{n}{2^2}) + 2 * \frac{n}{2} + n \\
&= 2^2 * f(\frac{n}{2^2}) + 2 * n \\
&= 2^2 * (2 * f(\frac{n}{2^3}) + \frac{n}{2^2}) + 2 * n \\
&= 2^3 * f(\frac{n}{2^3}) + 2^2 * \frac{n}{2^2} + 2 * n \\
&= 2^3 * f(\frac{n}{2^3}) + 3 * n \\
&= 2^i * f(\frac{n}{2^i}) + i * n : 2^i = n : \log_2 2^i = \log_2 n : i = \log_2 n \\
&= n * f(1) + n * \log n
\end{aligned}
$$

Algoritmos de Ordenação Eficientes

- In-place?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge
- Estabilidade?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge
- Estabilidade?
  - Mantém a ordem relativa

Algoritmos de Ordenação Eficientes - Otimizações

- Nos sub-vetores pequenos, alterne para o Insertion Sort
  - Cerca de 15 itens mais ou menos
  - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento (Sedgewick)

Algoritmos de Ordenação Eficientes - Otimizações

- Nos sub-vetores pequenos, alterne para o Insertion Sort
  - Cerca de 15 itens mais ou menos
  - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento (Sedgewick)
- Teste se o vetor já está em ordem
  - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para `merge()`:

Algoritmos de Ordenação Eficientes - Otimizações

- Nos sub-vetores pequenos, alterne para o Insertion Sort
  - Cerca de 15 itens mais ou menos
  - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento (Sedgewick)
- Teste se o vetor já está em ordem
  - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para `merge()`:
    - se $v[meio]$ for menor ou igual a $v[meio + 1]$

Algoritmos de Ordenação Eficientes - Otimizações

- Nos sub-vetores pequenos, alterne para o Insertion Sort
  - Cerca de 15 itens mais ou menos
  - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento (Sedgewick)
- Teste se o vetor já está em ordem
  - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para `merge()`:
    - se $v[meio]$ for menor ou igual a $v[meio+1]$
  - Adapativo, mas não diminui as chamadas recursivas

Algoritmos de Ordenação Eficientes - Otimizações

- Nos sub-vetores pequenos, alterne para o Insertion Sort
  - Cerca de 15 itens mais ou menos
  - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento (Sedgewick)
- Teste se o vetor já está em ordem
  - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para `merge()`:
    - se $v[meio]$ for menor ou igual a $v[meio + 1]$
  - Adapativo, mas não diminui as chamadas recursivas
- Reutilize o array auxiliar tornando o principal
  - É possível eliminar o tempo (mas não o espaço) gasto para copiar para o vetor auxiliar usado no merge
  - Passe os vetores como parâmetros: alterne os vetores que serão ordenados
- Implementem.

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r) {
    /*if (r-l <= 15) {
        insertion(v, l, r);
        return;
    }*/
    if (l >= r) return;

    int m = (r+l)/2;
    merge_sort(v, l, m);   //F(n/2)
    merge_sort(v, m+1, r); //F(n/2)

    if(v[m]<v[m+1]) return;
    merge(v, l, m, r); //caso médio: n/2
}
```

Melhor caso: $F(n) \approx n + \log n \to O(n)$
Caso médio: $F(n) \approx n + \frac{n}{2} * \log n \to O(n \log n)$

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões

```
link merge(link a, link b)
  { struct node head; link c = &head;
    while ((a != NULL) && (b != NULL))
      if (less(a->item, b->item))
        { c->next = a; c = a; a = a->next; }
      else
        { c->next = b; c = b; b = b->next; }
    c->next = (a == NULL) ? b : a;
    return head.next;
  }
```

Merge

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões

```
link merge(link a, link b);
link mergesort(link c)
  { link a, b;
    if (c == NULL || c->next == NULL) return c;
    a = c; b = c->next;
    while ((b != NULL) && (b->next != NULL))
      { c = c->next; b = b->next->next; }
    b = c->next; c->next = NULL;
    return merge(mergesort(a), mergesort(b));
  }
```

Abordagem Top-Down

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões

```
link mergesort(link t)
  { link u;
    for (Qinit(); t != NULL; t = u)
      { u = t->next; t->next = NULL; Qput(t); }
    t = Qget();
    while (!Qempty())
      { Qput(t); t = merge(Qget(), Qget()); }
    return t;
  }
```

Abordagem Bottom-Up

Roteiro

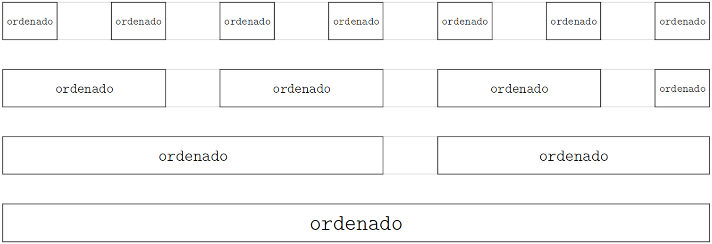
1. Algoritmos de Ordenação Eficientes
   - Merge Sort
   - Quick Sort

Algoritmos de Ordenação Eficientes - Quick Sort

- Um dos mais utilizados
- Simples
- Eficiente
- Muito pesquisado
  - Bem embasado
  - Bem comprovado

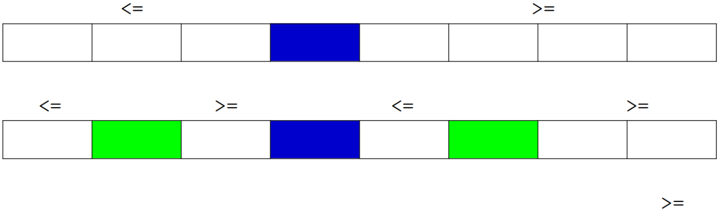
Algoritmos de Ordenação Eficientes - Quick Sort

- Método dividir e conquistar
- Particiona o vetor em sub-vetores
- Ordenando cada sub-vetor independentemente
- Merge x Quick
  - merge:
    - Divide
    - Ordena separadamente
    - Combina reordenando
    - Conquista um vetor mais ordenado



Algoritmos de Ordenação Eficientes - Quick Sort

- Método dividir e conquistar
- Particiona o vetor em sub-vetores
- Ordenando cada sub-vetor independentemente
- Merge x Quick
  - merge:
    - Divide
    - Ordena separadamente
    - Combina reordenando
    - Conquista um vetor mais ordenado
  - quick:
    - Separa os elementos baseados em 1 elemento
    - Conquista um elemento ordenado e dois sub-vetores pseudo-ordenados
    - Divide e repete para os sub-vetores



Algoritmos de Ordenação Eficientes - Quick Sort

- Ideia:
  - Particionar (separar) - processo crucial no quick
  - Escolhar um elemento de referência: pivô
  - Reorganizar os elementos de acordo com o pivô
  - Pivô na posição final
  - Pivô marca a divisão dos sub-vetores
  - Repetir o processo até ordenar todos os elementos
- Condições que devem ser satisfeitas:
  - O elemento a[j] está na sua posição final no vetor, para algum j
  - Nenhum elemento anterior ao a[j] é maior do que o a[j]
  - Nenhum elemento posterior ao a[j] é menor do que o a[j]

<=   >=

[2   1   3   4   5]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar/separar** : pivô (+ direita) e rearranja (maiores e menores)
- dividir

[2 5 3 1 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] $<$ pivô? (enquanto v[i] for menor)

```
[2    5    3    1    4    3]
 i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] $<$ pivô? (enquanto v[i] for menor)

```
[2    5    3    1    4    3]
      i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[2   5   3   1   4   3]
     i
                 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[2   5   3   1   4   3]
     i
             j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j?
- v[i] (maior) está antes de v[j] (menor)?

```
[2    5    3    1    4    3]
      i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ v[j]
- maior p/ direita e menor p/ esquerda

```
[2    1    3    5    4    3]
      i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] $<$ pivô? (enquanto v[i] for menor)

```
[2    1    3    5    4    3]
           i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[2    1    3    5    4    3]
           i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j?
- v[i] (maior) está antes de v[j] (menor)?

```
[2    1    3    5    4    3]
           i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Posicionar o pivô
- swap último maior $\leftrightarrow$ pivô

```
[2    1    3    5    4    3]
           i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\rightarrow$ pivô na sua posição final

```
[2    1    3    5    4    3]
           i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir** : sub-vetores

[2 1 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar/separar** : pivô (+ direita) e rearranja (maiores e menores)
- dividir

[2 1 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] $<$ pivô? (enquanto v[i] for menor)

```
[2   1   3   5   4   3]
 i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[2    1    3    5    4    3]
 i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j?
- v[i] (maior) está antes de v[j] (menor)?

```
[2    1    3    5    4    3]
 i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Posicionar o pivô
- swap último maior ↔ pivô

[1 2 3 5 4 3]
i
j

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\longrightarrow$ pivô na sua posição final

```
[1  2  3  5  4  3]
 i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir** : sub-vetores

[1 2 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- r <= l? sim
- pivô na sua posição final

[1 2 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir** : sub-vetores

[1 2 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar/separar** : pivô (+ direita) e rearranja (maiores e menores)
- dividir

[1 2 3 5 4 3]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] $<$ pivô? (enquanto v[i] for menor)

[1 2 3 5 4 3]

i

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[1   2   3   5   4   3]
             i
                 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô $<$ v[j] e j $>$ l? (enquanto pivô for menor)

```
[1   2   3   5   4   3]
             i
             j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j?
- v[i] (maior) está antes de v[j] (menor)?

```
[1    2    3    5    4    3]
                i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Posicionar o pivô
- swap último maior ↔ pivô

```
[1    2    3    3    4    5]
                i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\rightarrow$ pivô na sua posição final

```
[1    2    3    3    4    5]
                i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir** : sub-vetores

[1 2 3 3 4 5]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar/separar** : pivô (+ direita) e rearranja (maiores e menores)
- dividir

[1 2 3 3 4 5]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô? (enquanto v[i] for menor)

[1 2 3 3 4 5]

i

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô? (enquanto v[i] for menor)

[1 2 3 3 4 5]

i

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor ou igual que o pivô
- pivô < v[j] e j > l? (enquanto pivô for menor)

```
[1    2    3    3    4    5]
                         i
                    j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j?
- v[i] (maior) está antes de v[j] (menor)?

```
[1   2   3   3   4   5]
                     i
                 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Posicionar o pivô
- swap último maior $\leftrightarrow$ pivô

```
[1   2   3   3   4   5]
                     i
                 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\rightarrow$ pivô na sua posição final

```
[1    2    3    3    4    5]
                          i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir** : sub-vetores

[1 2 3 3 4 5]

Algoritmos de Ordenação Eficientes - Quick Sort

- r <= l? sim
- pivô na sua posição final

[1 2 3 3 4 5]

Algoritmos de Ordenação Eficientes - Quick Sort

```
void quick_sort(int *v,  int l, int r)
{
    //condição de parada
    if(r<=l) return;
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
void quick_sort(int *v,  int l, int r)
{
    //condição de parada
    if(r<=l) return;

    //posicionando o pivô
    int p = partition(v, l, r);
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
void quick_sort(int *v,  int l, int r)
{
    //condição de parada
    if(r<=l) return;

    //posicionando o pivô
    int p = partition(v, l, r);

    //ordena os sub-vetores
    quick_sort(v, l, p-1); //menores
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
void quick_sort(int *v,  int l, int r)
{
    //condição de parada
    if(r<=l) return;

    //posicionando o pivô
    int p = partition(v, l, r);

    //ordena os sub-vetores
    quick_sort(v, l, p-1); //menores
    quick_sort(v, p+1, r); //maiores
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
void quick_sort(int *v,  int l, int r)
{
    //condição de parada
    if(r<=l) return;

    //posicionando o pivô
    int p = partition(v, l, r);

    //ordena os sub-vetores
    quick_sort(v, l, p-1); //menores
    quick_sort(v, p+1, r); //maiores
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`
   3. `quick_sort(v, 4, 4)` : **5**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`
   3. `quick_sort(v, 4, 4)` : **5**
   4. 4 5

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`
   3. `quick_sort(v, 4, 4)` : **5**
   4. 4 5
6. **1 2 3 4 5**

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda
2. Laço para procurar elementos:

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda
2. Laço para procurar elementos:
   1. Procurar elementos maiores : esquerda p/ direita

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda
2. Laço para procurar elementos:
   1. Procurar elementos maiores : esquerda p/ direita
   2. Procurar elementos menores : direita p/esquerda

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda
2. Laço para procurar elementos:
   1. Procurar elementos maiores : esquerda p/ direita
   2. Procurar elementos menores : direita p/esquerda
   3. Condição de parada?

Algoritmos de Ordenação Eficientes - Quick Sort

Decisões de projeto e implementação:

1. Qual elemento será o pivô?
   - Elemento mais à direita ou o mais à esquerda
2. Laço para procurar elementos:
   1. Procurar elementos maiores : esquerda p/ direita
   2. Procurar elementos menores : direita p/esquerda
   3. Condição de parada?
      - Varredura à esquerda e direita se cruzarem

```
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  //pivô
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  //pivô
  int pivot = v[r];

  //procurar à esquerda: i = l?
  //procurar à direita:  j = r-1?
  //atualizar e verificar:
  //  garante posição válida no fim da iteração
```

```
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  //pivô
  int pivot = v[r];

  //procurar à esquerda: i = l?
  //procurar à direita:  j = r-1?
  //atualizar e verificar:
  //  garante posição válida no fim da iteração
  //  i e j não podem ultrapassar os limites pois
  //        serão usados nos swaps
```

```
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  //pivô
  int pivot = v[r];

  //procurar à esquerda: i = l?
  //procurar à direita:  j = r-1?
  //atualizar e verificar:
  //  garante posição válida no fim da iteração
  //  i e j não podem ultrapassar os limites pois
  //        serão usados nos swaps
  int i = l-1;
  int j = r;

```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1;
  int j = r;

  //percorrer da esquerda p/ direita
  //percorrer da direita p/esquerda
  //condição de parada?

```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot);
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
```

```
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
    while(v[--j] > pivot && j>l);

    //se o maior está atrás do menor
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
    while(v[--j] > pivot && j>l);

    //se o maior está atrás do menor
    if(i<j) exch(v[i], v[j]); //maior p/ direita e
                              //menor p/ esquerda
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
    while(v[--j] > pivot && j>l);

    //se o maior está atrás do menor
    if(i<j) exch(v[i], v[j]); //maior p/ direita e
                              //menor p/ esquerda
  }
  //posiciona o pivô depois do último menor
  //o primeiro maior vai p/ direita
```

```c
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
    while(v[--j] > pivot && j>l);

    //se o maior está atrás do menor
    if(i<j) exch(v[i], v[j]); //maior p/ direita e
                              //menor p/ esquerda
  }
  //posiciona o pivô depois do último menor
  //o primeiro maior vai p/ direita
  exch(v[i], v[r]);

  //nova posição do pivô
```

```
//Sedgewick: pivô à direita
int partition(int *v, int l, int r) {
  int pivot = v[r];
  int i = l-1, j = r;

  //enquanto percursos não se cruzaram
  while(i<j) {
    //procurar elementos maiores
    while(v[++i] < pivot); //preciso verificar i<r?

    //procurar elementos menores
    while(v[--j] > pivot && j>l);

    //se o maior está atrás do menor
    if(i<j) exch(v[i], v[j]); //maior p/ direita e
                              //menor p/ esquerda
  }
  //posiciona o pivô depois do último menor
  //o primeiro maior vai p/ direita
  exch(v[i], v[r]);

  //nova posição do pivô
  return i;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
//Sedgewick: pivô à esquerda
int partition(int *v,  int l, int r) {
    int pivot = v[l];
    int i = l;
    int j = r+1;
    while (i<j) {
        while (v[++i] < pivot && i < r);
        while (pivot < v[--j] && j > l);
        if (i < j)
            exch(v[i], v[j]);
    }

    //posiciona o pivô antes do primeiro maior
    //o último menor vai p/ esquerda
    exch(v[l], v[j]);

    return j; //nova posição do pivot
}
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- define-se o pivô - elemento mais a direita

[3 1 4 2 4]

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] $\leftrightarrow$ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
 i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] ↔ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3   1   4   2   4]
     i
     j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] $\leftrightarrow$ v[j]: “puxando” o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
               i
               j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] ↔ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] $\leftrightarrow$ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    2    4    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- i == r
- swap v[j] ↔ pivô

```
[3    1    2    4    4]
                     i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- j = pivô na sua posição final

```
[3    1    2    4    4]
                     i
                4
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
//Cormem
int partition(int *v, int l, int r)
{
    int pivot = v[r];
    int j = l;
    int i = l;
    while(i<r) //não ultrapassar o limite superior
    {
        if(less(v[i], pivot)){
            exch(v[i], v[j]);
            j++;
        }
        i++;
    }

    exch(v[r], v[j]);

    return j;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?
  - Tem trocas com saltos? Sim

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?
  - Tem trocas com saltos? Sim
  - Não estável

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                    i<

[2    1    3    4    5]
              <j    i<

[2    1    3    4    5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                     i<

[2    1    3    4    5]
               <j   i<

[2    1    3    4    5]
```

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                i<

[2    1    3    4    5]
          <j   i<

[2    1    3    4    5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.
  - A cada particiona, o pivô, já na sua posição correta, divide o sub-vetor em apenas "menos 1 elemento"

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                    i<

[2    1    3    4    5]
              <j    i<

[2    1    3    4    5]
```

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
               i<

[2    1    3    4    5]
         <j    i<

[2    1    3    4    5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$

```
int partition(int *v, int l, int r) {
  int i=l-1, j=r, pivot = v[r];

  while(i<j) {
    while(v[++i] < pivot);          //l até pivot
    while(pivot < v[--j] && j>l); //r até pivot
    if(i<j) exch(v[i], v[j]);       //até r-l trocas
  }

  exch(v[i], v[r]); //1

  //f(n) ~ 2n + 1
  return i;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$

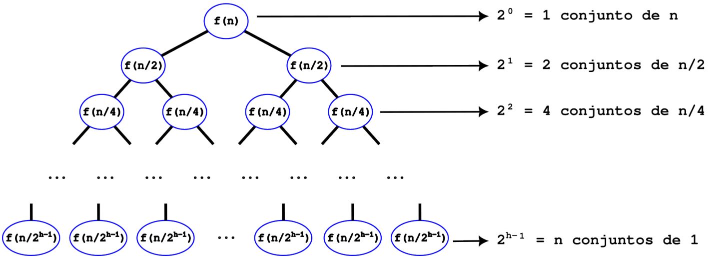
```
void quick_sort(int *v,  int l, int r) {
    if(r<=l) return;

    int p = partition(v, l, r); //O(n)

    quick_sort(v, l, p-1); //~f(n/2): melhor caso
    quick_sort(v, p+1, r); //~f(n/2): melhor caso
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

$$
\begin{aligned}
f(n) &= 2 * f(\frac{n}{2}) + n \\
&= 2 * (2 * f(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 2^2 * f(\frac{n}{2^2}) + 2 * n \\
&= 2^2 * (2 * f(\frac{n}{2^3}) + \frac{n}{2^2}) + 2 * n \\
&= 2^3 * f(\frac{n}{2^3}) + 3 * n \\
&= 2^i * f(\frac{n}{2^i}) + i * n :: 2^i = n : i = \log_2 n \\
&= n * f(1) + \log n * n
\end{aligned}
$$

Algoritmos de Ordenação Eficientes - Quick Sort

- Cada particiona/separa, possui tempo linear
- A cada nível da árvore recursiva, a complexidade total (soma das chamadas recursivas) é igual ao tamanho total da entrada
  - $\sum_{i=0}^{h-1} \frac{n}{2^i} = n$, sendo n o tamanho da entrada e i um nível da árvore
- n itens são comparados/movimentados $\log n$ vezes



$1 = n/2^{h-1}$
$2^{h-1} = n$

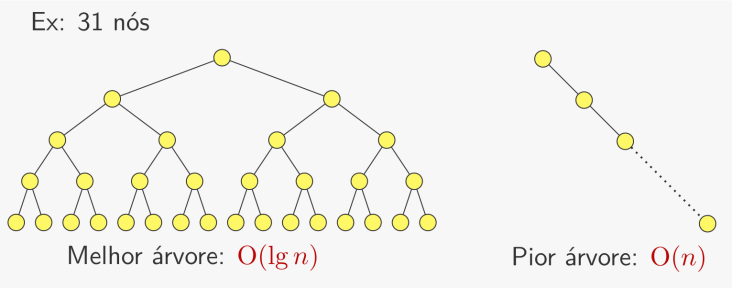
$\lg 2^{h-1} = \lg n$
$(h-1).\lg 2 = \lg n$
$h = \lg n + 1$

$T(n) = n.h$
$= n.\lg n + n$

$O(n.\lg n)$

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$
  - Pior caso: $n^2/2$ comparações
    - Muito itens repetidos, (quase) ordenados, reverso caem nos piores casos
    - Aumenta-se a entrada por recursão/divisão, aumentando o custo linear



Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$
  - Pior caso: $n^2/2$ comparações
    - Muito itens repetidos, (quase) ordenados, reverso caem nos piores casos
    - Aumenta-se a entrada por recursão/divisão, aumentando o custo linear

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Pior caso: $n^2/2$ comparações

```c
void quick_sort(int *v,  int l, int r) {
    if(r<=l) return;

    int p = partition(v, l, r); //n

    quick_sort(v, l, p-1); //~f(n-1)
    quick_sort(v, p+1, r); //~f(0)
}
```

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + n - 1 + f(n-2) \\
&= n + n - 1 + n - 2 + f(n-3) \\
&= n + n - 1 + n - 2 + n - 3 + f(n-4) \\
&= n + n - 1 + n - 2 + n - 3 + \ldots + 0 \\
&= 0 + 1 + 2 + \ldots + n \\
&= \frac{(0+n) * n}{2} = \frac{n^2}{2}
\end{aligned}
$$

Algoritmos de Ordenação Eficientes - Quick Sort

- Ultrapassar quando for igual pivô (impacto quando há muitos repetidos):
  - Buscar maior E buscar menor: muitos iguais para direita ou esquerda, desbalanceamento
  - Buscar maior E buscar menor ou igual: iguais para esquerda, desbalanceamento
  - Buscar maior ou igual E buscar menor: iguais para direita, desbalanceamento
- Não ultrapassar quando for igual ao pivô: partes mais balanceadas
- Desbalanceamento: aumenta-se o custo relativo de cada `partition` de cada recursão

Algoritmos de Ordenação Eficientes - Quick Sort

```
while(i<j) {
  //procurando maior
  //i<r necessário pela perda
  //     do sentinela
  while(v[++i] <= pivot && i<r);

  //procurando maior ou igual
  //while(v[++i] < pivot);

  //procurando menor
  while(pivot <= v[--j] && j>l);

  //procurando menor ou igual
  //while(pivot < v[--j] && j>l);

  if(i<j) exch(v[i], v[j]);
}
```

```
maior     e     menor
2    3    3    3    3    3    3   _9_  _2_   3    3    3'
2    3    3    3    3    3    3   _2_  _9_   3    3    3'
2    3    3    3    3    3    3    2    3'   3    3    9

maior ou igual     e     menor
2   _3_   3    3    3    3    3    9   _2_   3    3    3'
2   _2_   3    3    3    3    3    9   _3_   3    3    3'
2    2   _3_   3    3    3    3    9    3    3    3    3'
2    2    3'   3    3    3    3    9    3    3    3    3

maior     e     menor ou igual
2    3    3    3    3    3    3   _9_   2    3   _3_   3'
2    3    3    3    3    3    3    3    2   _3_   9    3'
2    3    3    3    3    3    3    3    2    3'   9    3

maior ou igual     e     menor ou igual
2   _3_   3    3    3    3    3    9    2    3   _3_   3'
2    3   _3_   3    3    3    3    9    2   _3_   3    3'
2    3    3   _3_   3    3    3    9   _2_   3    3    3'
2    3    3   _2_   3    3    3    9   _3_   3    3    3'
2    3    3    2   _3_   3   _3_   9    3    3    3    3'
2    3    3    2    3   _3_   3    9    3    3    3    3'
2    3    3    2    3    3'   3    9    3    3    3    3
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Otimização: mediana de três
  - Pivô: usar a mediana de uma pequena amostra de itens
  - Melhora o particionamento
  - Três chaves: `v[esquerda]`, `v[meio]` e `v[direita]`
    - Menor vai para esquerda
    - Mediana (pivô) vai para direita

```
if(v[meio] < v[l]) exch(v[meio], v[l]); //menor p/ left
if(v[r] < v[l])    exch(v[l], v[r]);    //menor p/ left
if(v[meio] < v[r]) exch(v[r], v[meio]); //menor dos maiores
                                        //      p/ right
```

```
[6 - - 5 - - - 4] original
[5 - - 6 - - - 4] v[meio]<v[l]?
[4 - - 6 - - - 5] v[r]<v[l]?
[4 - - 6 - - - 5] v[meio]<v[r]?
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Mediana de três (pivô v[r])

```c
int meio = (l+r)/2;
if(v[meio] < v[l]) exch(v[meio], v[l]);
if(v[r] < v[l])     exch(v[l], v[r]);
if(v[meio] < v[r]) exch(v[r], v[meio]);
```

- Utilizando as macros

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

...
compexch(v[l], v[(l+r)/2]); //troca se meio for menor
compexch(v[l], v[r]);       //troca se r for menor
compexch(v[r], v[(l+r)/2]); //troca se meio for menor
```

- Implementem a mediana de três com pivô v[l]!

Algoritmos de Ordenação Eficientes - Quick Sort

- Otimizando a mediana de três (pivô v[r])
  - Menor item já está à esquerda
  - Objetivo: colocar o maior item em r
    - Garantir um item maior que o pivô mais à direita
  - Fazer o particionamento de (l+1, r-1)

```
exch(v[(l+r)/2], v[r-1]); //meio p/ penúltimo
compexch(v[l], v[r-1]);   //menor p/ left
compexch(v[l], v[r]);     //menor p/ left
compexch(v[r-1], v[r]);   //menor dos maiores p/ penúltimo

int p = partition(v, l+1, r-1);
```

[6	-	-	5	-	-	-	4]
[6	-	-	-	-	-	5	4]
[5	-	-	-	-	-	6	4]
[4	-	-	-	-	-	6	5]
[4	-	-	-	-	-	5	6]

Algoritmos de Ordenação Eficientes - Quick Sort

- Melhorias
  - Utilizar o Insertion Sort
    - Insertion Sort é rápido para pequenos vetores
    - É adaptativo
    - Alternar para o Insertion para pequenos vetores
    - Algo entre 5 e 15 chaves

```
void quick_sort(int *v,  int l, int r) {
    //if(r<=l) return;
    if(r-l<=15) return insertion_sort(v, l, r);

    int p = partition(v, l, r);

    quick_sort(v, l, p-1);
    quick_sort(v, p+1, r);
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Insertion
- Mediana de três
- Vamos testar:
  - Dados aleatórios
  - Quase ordenado

```
void quick_sort(int *v,  int l, int r) {
    if(r-l<=15) return insertion_sort(v, l, r);

    exch(v[(l+r)/2], v[r-1]);
    compexch(v[l], v[r-1]);
    compexch(v[l], v[r]);
    compexch(v[r-1], v[r]);

    int p = partition(v, l+1, r-1);
    quick_sort(v, l, p-1);
    quick_sort(v, p+1, r);
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Passando como parâmetro função responsável pela comparação das chaves

```
typedef struct{
    int chave1;
    int chave2;
} Item;

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

void quick_sort(int [],  int, int,
                int (*)(const void*, const void*));
int partition(int [], int, int,
              int (*)(const void*, const void*));

int less(const void* v1, const void* v2){
    return ((Item *)v1)->chave1 < ((Item *)v2)->chave1;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Passando como parâmetro função responsável pela comparação das chaves

```
//invocando a função
quick_sort(v, 20, 100, less);

...

//invocando a função pelo ponteiro
int partition(int *v, int l, int r,
              int (*comp)(const void*, const void*)) {
    ...
    while(i<j) {
        while(comp(&v[++i], &pivot));
        while(comp(&pivot, &v[j]) && j>l) j--;
        if(i<j) exch(v[i], v[j]);
    }
    ...
}
```