MÉTODO DE ORDENAÇÃO: MERGE SORT

- Ideia:
  - Vetor de n registros
  - Particionar em n partes de tamanho 1
    - Que são intercalados em pares
      - n/2 partes de tamanho 2
    - Que são intercalados em pares
      - n/4 partes de tamanho 4
    - …
    - Que são intercalados em pares
      - 1 parte de tamanho n
      - Ordenado

- Algoritmo baseado na operação: merge (combinar, juntar, fundir)

- Método **dividir e conquistar**
  - Dividir em pequenas partes
  - Ordenar essas partes
  - Combinar essas partes ordenadas
  - Até formar uma única sequência ordenada
- Gif https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif

Abordagem Top-Down

- Abordagem **recursiva**
  - 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, ou seja, ordenado em **1**

- Volta fazendo o merge ordenado
  - **merge**(**int** *v, **int** l, **int** meio, **int** r)
  - Utiliza um <u>**vetor auxiliar**</u>

merge_sort(v, l, r)

l l+(r-l)/2 r

6	5	3	1	8	7	2	4

m = l+(r-l)/2 = (l+r)/2;

merge_sort(v, l, m)

6	5	3	1

merge_sort(v, m+1, r)

8	7	2	4

merge_sort		merge_sort		merge_sort		merge_sort	
6	5	3	1	8	7	2	4

m_s	m_s	m_s	m_s	m_s	m_s	m_s	m_s
6	5	3	1	8	7	2	4

----

merge(v,0,0,1) | merge(v,2,2,3) | merge(v,4,4,5) |merge(v,6,6,7)

6	-	5	|	3	-	1	|	8	-	7	|	2	-	4
i		j		i		j		i		j		i		j

aux

5	6	1	3	7	8	2	4
k							

original

5	6	1	3	7	8	2	4

```
---------------------------------------------------
   merge(v, 0, 1, 3)           |    merge(v, 4, 5, 7)
```

5	6		1	3		7	8		2	4

aux

1	3	5	6		2	4	7	8

k

original

1	3	5	6		2	4	7	8

```
----------------------------------------------------
                   merge(v, 0, 3, 7)
```

1	3	5	6		2	4	7	8
			i			j		

aux

1	2	3	4	5	6	7	8
k							

original

1	2	3	4	5	6	7	8

merge_sort(v, 0, 7)

1. **meio = (7+0)/2 = 3**
2. merge_sort(v, 0, meio=3) ← **esquerda**
   2.1. **m = (3+0)/2 = 1**
   2.2. merge_sort(v, 0, 1) ← esquerda

a) m = (1+0)/2 = 0

b) merge_sort(v, 0, 0) ← esquerda

c) merge_sort(v, 1, 1) ← direita

d) merge(v, 0, 0, 1)

- vetor original → vetor auxiliar
  - 6 5 3 1 8 7 2 4 → **5**
  - 6 5 3 1 8 7 2 4 → **5 6**
- vetor original ← vetor auxiliar
  - **5 6** 3 1 8 7 2 4

2.3. **merge_sort(v, 2, 3) ← direita**

a) m = (3+2)/2 = 2

b) merge_sort(v, 2, 2) ← esquerda

c) merge_sort(v, 3, 3) ← direita

d) merge(v, 2, 2, 3)

- vetor original → vetor auxiliar
  - 5 6 3 1 8 7 2 4 → **1**
  - 5 6 3 1 8 7 2 4 → **1 3**
- vetor original ← vetor auxiliar
  - 5 6 **1 3** 8 7 2 4

2.4. merge(v, 0, 1, 3)

a) vetor original → vetor auxiliar

- 5 6 1 3 8 7 2 4 → **1**
- 5 6 1 3 8 7 2 4 → **1 3**
- 5 6 1 3 8 7 2 4 → **1 3 5**
- 5 6 1 3 8 7 2 4 → **1 3 5 6**

b) vetor original ← vetor auxiliar

- 1 3 5 6 8 7 2 4

3. merge_sort(v, meio+1, 7) ← direita
   3.1. m = (7+4)/2 = 5
   3.2. merge_sort(v, 4, 5) ← esquerda
      a) m = (5+4)/2 = 4
      b) merge_sort(v, 4, 4) ← esquerda
      c) merge_sort(v, 5, 5) ← esquerda
      d) merge(v, 4, 4, 5)
         - vetor original → vetor auxiliar
            - 1 3 5 6 8 7 2 4 → 7
            - 1 3 5 6 8 7 2 4 → 7 8
         - vetor original ← vetor auxiliar
            - 1 3 5 6 7 8 2 4

   3.3. merge_sort(v, 6, 7) ← direita
      a) m = (7+6)/2 = 6
      b) merge_sort(v, 6, 6) ← esquerda
      c) merge_sort(v, 7, 7) ← esquerda
      d) merge(v, 6, 6, 7)
         - vetor original → vetor auxiliar
            - 1 3 5 6 7 8 2 4 → 2
            - 1 3 5 6 7 8 2 4 → 2 4
         - vetor original ← vetor auxiliar
            - 1 3 5 6 7 8 2 4

   3.4. merge(v, 4, 5, 7)
      a) vetor original → vetor auxiliar

- 1 3 5 6 <u>7</u> 8 <u>2</u> 4 → **2**
- 1 3 5 6 <u>7</u> 8 2 <u>4</u> → **2 4**
- 1 3 5 6 <u>7</u> 8 2 4 → **2 4 7**
- 1 3 5 6 7 <u>8</u> 2 4 → **2 4 7 8**

b) vetor original ← vetor auxiliar

- 1 3 5 6 **2 4 7 8**

4. merge(v, 0, 3, 7)

4.1. vetor original → vetor auxiliar

a) <u>1</u> 3 5 6 <u>2</u> 4 7 8 → **1**

b) 1 <u>3</u> 5 6 <u>2</u> 4 7 8 → **1 2**

c) 1 <u>3</u> 5 6 2 <u>4</u> 7 8 → **1 2 3**

d) 1 3 <u>5</u> 6 2 <u>4</u> 7 8 → **1 2 3 4**

e) 1 3 <u>5</u> 6 2 4 <u>7</u> 8 → **1 2 3 4 5**

f) 1 3 5 <u>6</u> 2 4 <u>7</u> 8 → **1 2 3 4 5 6**

g) 1 3 5 6 2 4 <u>7</u> 8 → **1 2 3 4 5 6 7**

h) 1 3 5 6 2 4 7 <u>8</u> → **1 2 3 4 5 6 7 8**

4.2. vetor original ← vetor auxiliar

a) **1 2 3 4 5 6 7 8**

Vamos implementar.

Complexidade assintótica

- Entre ½ ***N* log *N*** e ***N* log *N*** comparações
- Pior caso: **O(*n*log*n*)**
- Caso médio: **O(*n*log*n*)**
- Melhor caso: **O(*n*log*n*)**

In-place?

- Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - Proporcional a N

Adaptatividade?

- Ordenação
  - diminui as divisões?
  - diminui comparações?
    - diminui as movimentações?

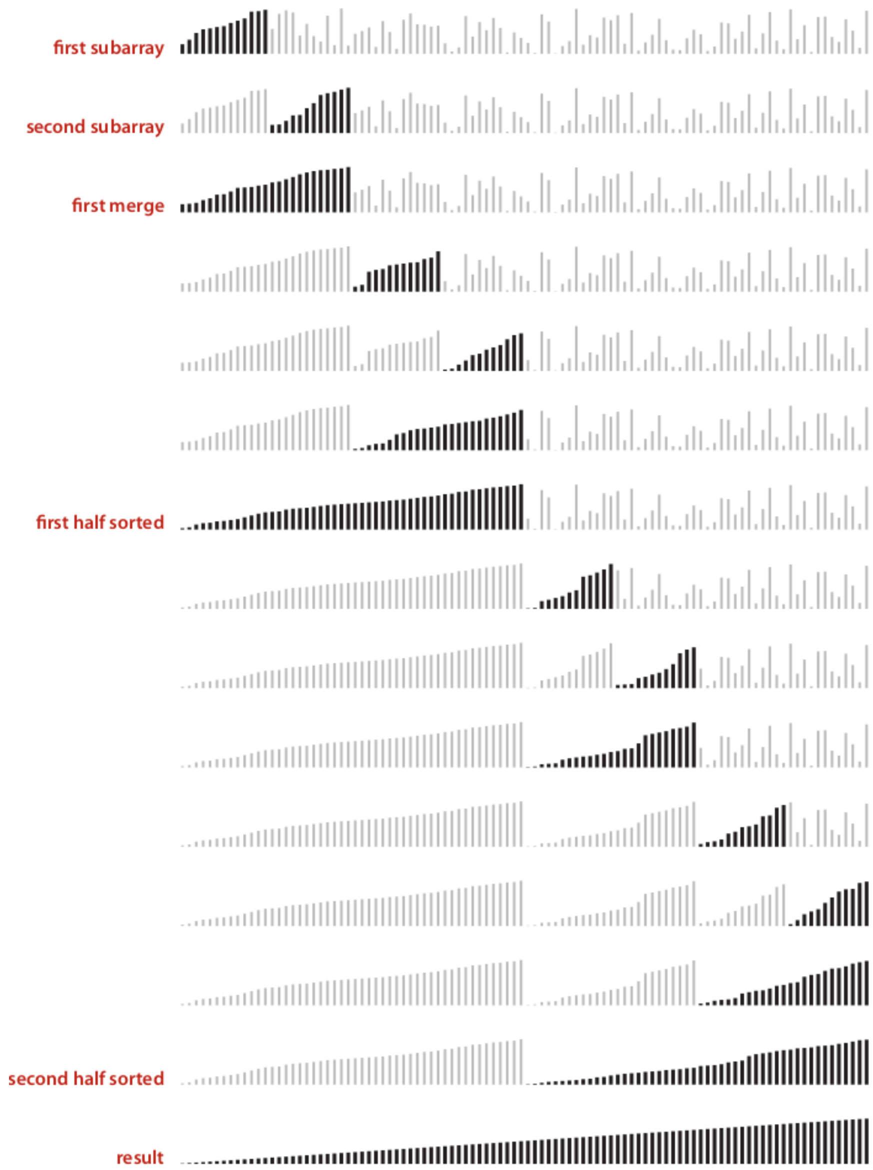
Estabilidade?

- Mantém a ordem relativa?
- Tem trocas com saltos?
  - Depende da implementação do merge
  - Estáveis?

```
Item aux[maxN];
merge(Item a[], int l, int m, int r)
  { int i, j, k;
    for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
    for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
    for (k = l; k <= r; k++)
       if (less(aux[j], aux[i]))
          a[k] = aux[j--]; else a[k] = aux[i++];
  }
```

Recomendações:

- Caso espaço seja um problema: não use o merge
- Nos sub-vetores pequenos
  - **Alterne para o <u>Insertion Sort</u>**
    - Cerca de 15 itens mais ou menos
  - Melhorará o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Fica adaptativo?



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

- **Teste se o vetor já está em ordem**
  - Podemos reduzir o tempo de execução para ser 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]**
  - Não diminui as chamadas recursivas
    - mas o tempo de execução para qualquer subarray ordenado é linear
    - 1 3 5 6

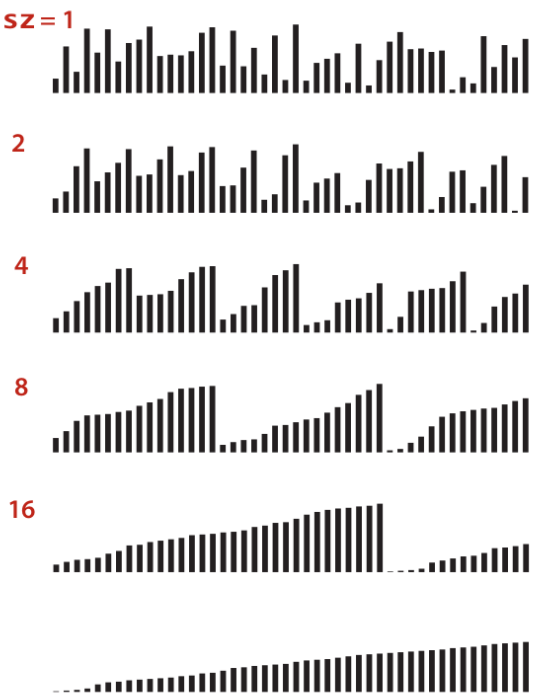
- **Elimine a cópia para o array auxiliar**
    - É possível eliminar o tempo (mas não o espaço) gasto para copiar para o vetor auxiliar usado no merge
    - Faça um vetor auxiliar do tamanho de vetor original
    - Duas chamadas do **merge_sort**:
        - Uma chamada recebe como parâmetro o **vetor original**
            - Coloca a saída ordenada no vetor auxiliar
        - A outra chamada recebe como parâmetro o **vetor auxiliar**
            - Coloca a saída ordenada no vetor original
        - Podemos organizar as chamadas recursivas de forma que a computação alterne os parâmetros do vetor original e do vetor auxiliar em cada nível
- **Implementem as sugestões de otimizações!**

<u>Abordagem Bottom-Up</u>

- No merge:
    - Ocorre a união de dois pequenos sub-vetores
    - Por que não fazer todos esses merges em 1 passo?
        - E o restante continua sendo unidos em pares
- Passos:
    - merge 1 por 1
        - sub-vetores de tamanho 1
        - resultando em um sub-vetor de tamanho 2

- merge 2 por 2:
  - sub-vetores de tamanho 2
  - resultando em um sub-vetor de tamanho 4
- e assim por diante

- O segundo sub-vetor pode ser menor do que o primeiro no último merge de cada passo
  - O que não é problema para função merge
- Consiste em uma sequencia de passos em todo o vetor, fazendo "sz por sz" uniões
  - Começando por 1 por 1 e dobrando em cada passo
  - O sub-vetor final é somente do tamanho sz se o tamanho do vetor é um múltiplo par de sz
    - senão será menor que sz
- Complexidade: mesma da abordagem top-down



Visual trace of bottom-up mergesort

```
sz = 1
merge(a,  0,  0,  1)
merge(a,  2,  2,  3)
merge(a,  4,  4,  5)
merge(a,  6,  6,  7)
merge(a,  8,  8,  9)
merge(a, 10, 10, 11)
merge(a, 12, 12, 13)
merge(a, 14, 14, 15)
sz = 2
merge(a,  0,  1,  3)
merge(a,  4,  5,  7)
merge(a,  8,  9, 11)
merge(a, 12, 13, 15)
sz = 4
merge(a,  0,  3,  7)
merge(a,  8, 11, 15)
sz = 8
merge(a,  0,  7, 15)
```

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(*nlogn*)**para dados aleatórios
- Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos

Vetor auxiliar local do merge?

- Sedegewick
  - Não recomenda a declaração do vetor auxiliar dentro da função merge
  - Diminuir o overhead(sobrecarga) dessa criação a cada merge
  - Recomenda, portanto, declarar o aux no merge_sort e passá-lo como argumento para o merge
  - **Implementem! ;)**

Merge Sort em listas encadeadas?

- Observem os códigos do Sedgewick e tentem implementar suas versões
- Merge

```
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;
  }
```

- Abordagem Top-Down

```
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));
  }
```

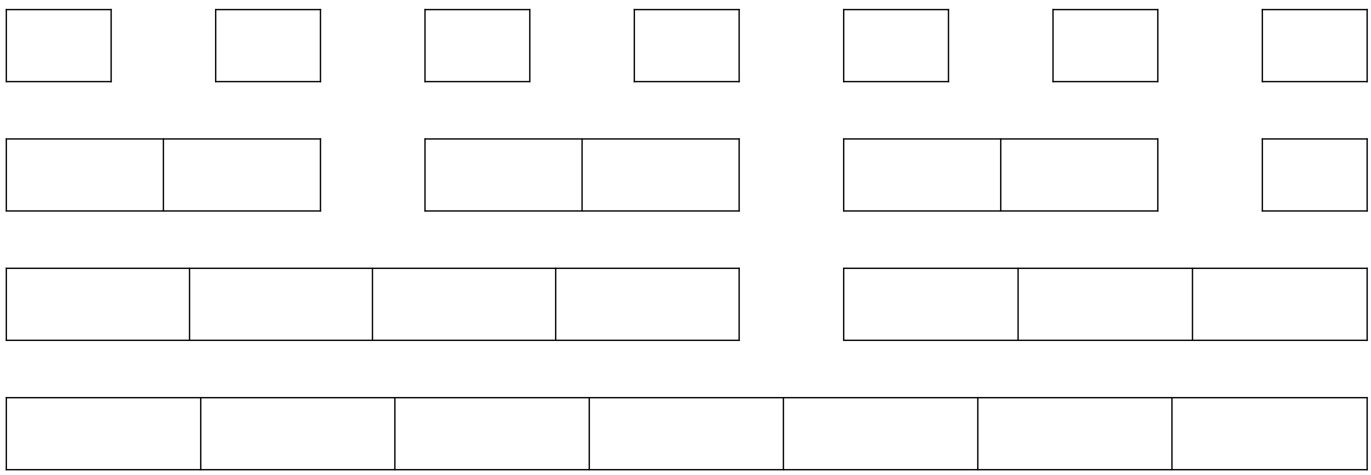
- Abordagem Bottom-Up

```
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;
  }
```

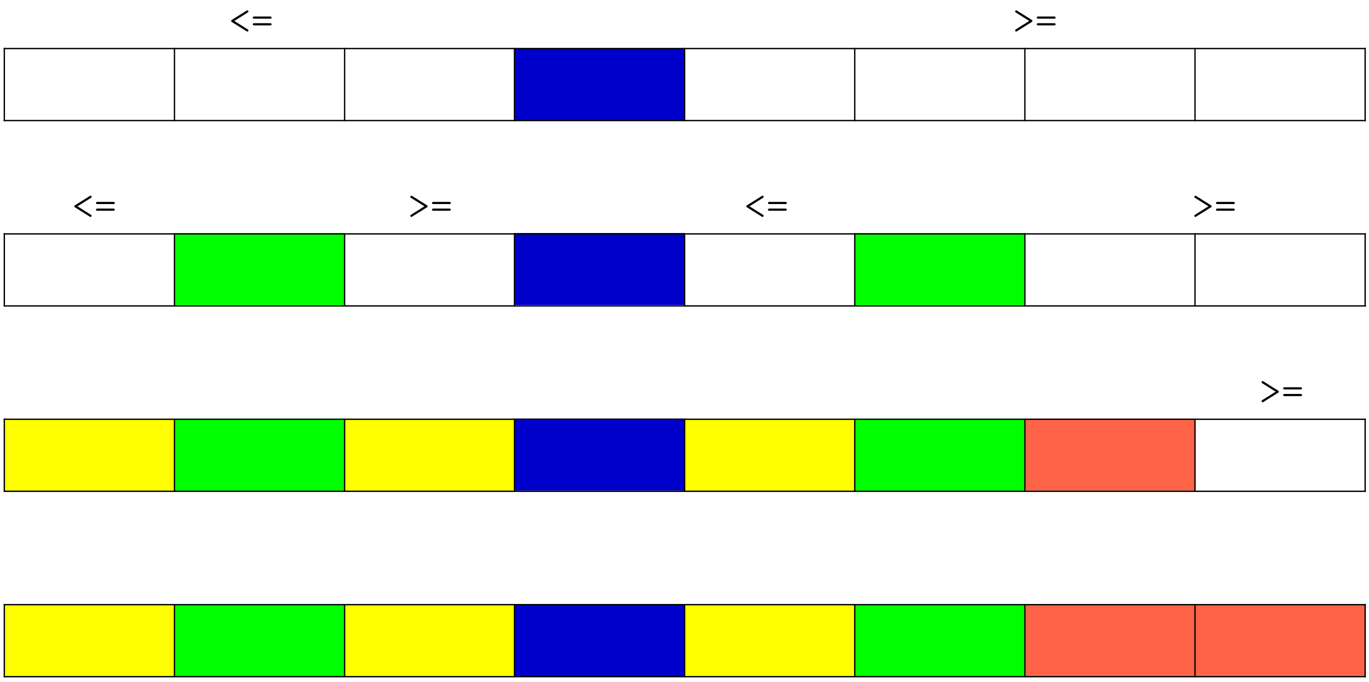
<u>MÉTODO DE ORDENAÇÃO: QUICK SORT</u>

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

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



    - quick:
      - Rearranja
        - Conquista um elemento ordenado e dois sub-vetores pseudo-ordenados
      - Divide
      - Repete



- Ideia:
  - **Particionar (separar)** - processo crucial no quick
    - Escolhar um elemento de referência: pivot
    - Rearranjar
      - Todos elementos posteriores ao pivot
      - Reposicionar o pivot
  - Dividir o vetor em dois
  - Repetir o processo até ordenar todos os elementos

- **Particionamento (separação)**
  - Escolhe um elemento para ser o item de partição (pivô)
    - Inicialmente este será o item mais à direita (ou esquerda)
    - No fim, estará na sua posição final

  - Varremos o vetor, comparando com o pivô
    - Procurando um elemento maior ou igual
    - Procurando um elemento menor ou igual
    - Troca-se os conteúdos desses elementos entre si

  - Troca-se o último maior (ou menor) item com o pivô

    - Reposicionar o pivô para sua posição final no vetor

- **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]

- Retorna a nova posição do pivô

<=			>=	
2	1	**3**	4	5

<u>Vamos simular 1 execução do particionamento/separa - versão não in-place</u>

- **partition(v, 0, 4)**
  - define-se o pivot - elemento mais a direita

3	1	5	2	**4(pivot)**

  - procurando os elementos maiores e menores

3	1	5	2	**4(pivot)**
				i

menores ou igual

3	1	2	**4(pivot)**	
			m	

maiores

5				
	n			

original

3	1	2	**4(pivot)**	5

  - Vantagens? Desvantagens?
  - Implementação?

Vamos simular 1 execução do particionamento/separa

Versão Cormem

- **partition(v, 0, 4)**
  - define-se o pivot - elemento mais a direita

3	1	4	2	**4(pivot)**

  - separando os elementos
    - v[i]<pivot?
      - swap v[i] ↔ v[j]
        - "puxando" o maior elemento para direita
      - j++
    - i++

3	1	4	2	**4(pivot)**
i < j				

3	1	4	2	**4(pivot)**
	i < j			

3	1	4	2	**4(pivot)**
		i < j		

3	1	4	2	**4(pivot)**
		j	i <	

3	1	2	4	**4(pivot)**
			j	i <

  - swap v[j] ↔ pivot

3	1	2	**4(pivot)**	4
			j	

    - pivot na sua posição final
    - retorna nova posição do pivot

Vamos implementar.

Vamos simular 1 execução do particionamento/separa - versão 1 do Sedegewick

- **partition(v, 0, 4)**
  - define-se o pivot - elemento mais a direita

3	1	4	2	**4(pivot)**

  - procurando os elementos maiores

3	1	4	2	**4(pivot)**
i <				

3	1	4	2	**4(pivot)**
	i <			

3	1	4	2	**4(pivot)**
		i <		

  - procurando os elementos menores

3	1	4	2	**4(pivot)**
		i <	j >	

  - i<j? swap v[i] ↔ v[j] *(por que i < j para o swap??!)*

3	1	2	4	**4(pivot)**
		i <	j >	

    - retoma a busca
      - procurando os elementos maiores

3	1	2	4	**4(pivot)**
			i <	
			j >	

- procurando os elementos menores

3	1	2	4	**4(pivot)**

j >   i <

- i<j? Não.

3	1	2	**4(pivot)**	4

j >   i <

- swap com a última maior chave
- pivot na posição final
- swap v[i] $\leftrightarrow$ pivot
- retorna nova posição do pivot

Vamos implementar.

Vamos simular 1 execução do particionamento/separa - versão 2 do Sedegewick

	i	j	a[] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

- **partition(v, 0, 4)**
  - define-se o pivot - elemento mais a esquerda

3(pivot)	1	3	2	4

  - procurando um elemento maior do que o pivot

3(pivot)	1	3	2	4
	i <			

3(pivot)	1	3	2	4
		i <		

  - procurando um elemento menor do que o pivot

3(pivot)	1	3	2	4
		i <		j >

- ○ i<j?

3(pivot)	1	3	2	4
		i <	j >	

  - ■ swap v[i] ↔ v[j]

3(pivot)	1	2	3	4
		i <	j >	

  - ■ retoma a busca
    - procurando um elemento maior do que o pivot

3(pivot)	1	2	3	4
			i < j >	

    - procurando um elemento menor do que o pivot

3(pivot)	1	2	3	4
		j >	i <	

    - i<j? Não.
      - ○ interrompa a busca
      - ○ pivot está na posição final??
      - ○ como garanto os requisitos do quick?
        - ■ itens à esquerda do pivot sejam menores
        - ■ itens à direita do pivot sejam maiores
      - ○ swap o último menor com o pivot

2	1	**3(pivot)**	3	4
		j >	i <	

      - ○ pivot na posição final
      - ○ retorna a nova posição do pivot: j

Vamos implementar.

E para rearranjar todo o vetor?

----

quick_sort(v, l, r)

- pos = partition(v, l, r)
- quick_sort(v, l, pos-1)
  - pos = partition(v, l, r)
  - quick_sort(v, l, pos-1)
  - quick_sort(v, pos+1, r)
- quick_sort(v, pos+1, r)
  - pos = partition(v, l, r)
  - quick_sort(v, l, pos-1)
  - quick_sort(v, pos+1, r)

quick_sort(v, 0, 4)

l				r
3	1	3	2	4

- pos = partition(v, 0, 4)
  - define-se o pivot

3(pivot)	1	3	2	4

  - procurando um elemento maior do que o pivot

3(pivot)	1	3	2	4
		i <		

  - procurando um elemento menor do que o pivot

3(pivot)	1	3	2	4
		i <	j >	

  - i<j? swap v[i] ↔ v[j]

3(pivot)	1	3	2	4
		i <	j >	

- retoma a busca
  - procurando um elemento maior do que o pivot

3(pivot)	1	2	3	4
			i < j >	

  - procurando um elemento menor do que o pivot

3(pivot)	1	2	3	4
		j >	i <	

  - i<j? Não.
    - interrompa a busca
    - swap o menor com o pivot

2	1	**3(pivot)**	3	4
		j >	i <	

    - retorna a nova posição(final) do pivot: j

- **quick_sort(v, l, pos - 1) → quick_sort(v, 0, 2-1)**

l	r			
2	1	3	3	4

  - **partition(v, 0, 1)**
    - Definindo o pivot

2(pivot)	1	3	3	4

    - procurando um elemento maior do que o pivot

2(pivot)	1	3	3	4
	i <			

- procurando um elemento menor do que o pivot

2(pivot)	1	3	3	4

i <

j >

- i<j? Não.

2(pivot)	1	3	3	4

j >

- interrompa a busca
- swap j - pivot

1	**2(pivot)**	3	3	4

j >

- retorne j

- **quick_sort(v, 0, 0-1)**
- **quick_sort(v, 0+1, 1)**

- **quick_sort(v, 2+1, 4)**

l r

1	2	3	3	4

- **partition(v, 3, 4)**
- Definindo o pivot

1	2	3	**3(pivot)**	4

- procurando um elemento maior do que o pivot

1	2	3	**3(pivot)**	4

i <

- procurando um elemento menor do que o pivot

1	2	3	**3(pivot)**	4
				i <
				j >

- i<j? Não.

1	2	3	**3(pivot)**	4
				j >

  - interrompa a busca
  - swap j - pivot

1	2	3	4	**3(pivot)**
				j >

  - retorne j

- `quick_sort(v, 3, 4-1)`
- `quick_sort(v, 4+1, 5)`

Vamos implementar.

Observações sobre as implementações

- **Cuidado com os limites**
  - Os apontadores i e j não podem ultrapassar os limites pois serão usados nos swaps
  - Opções:
    - **"anda" depois verifica**
      - Chega até o fim do vetor
        - while(v[++i] < pivot);
        - while(pivot < v[--j]) if(j==l) break;
      - Se começa "andando", tem que inicializar antes da primeira posição ser verificada
        - i = l-1
        - j = r
    - verifica depois "anda"
      - Tem que garantir que vai até o fim: i<=r
      - Tem que garantir que não vai ultrapassar o fim:
        - Se for o último não incrementa
        - while(i<=r && v[i] < pivot) i++;
        - if(i>r) i=r;
- **Manipular itens iguais ao pivot**
  - É melhor interromper a varredura à esquerda e à direita para itens iguais ao pivot
    - Estudos mostraram que esta estratégia tende a **balancear as partições** quando há várias chaves duplicadas
  - Se considerar o maior ou igual, e o menor ou igual

5(pivot)	4	5	6	5	6
	i			j	

5(pivot)	4	5	6	5	6
	j		i		

4	<u>**5**</u>	5	6	5	6
	j		i		

    - <u>Muitos elementos seriam ultrapassados</u>
    - Próxima partição seria muito desbalanceada

- **In-place?**
  - Utiliza memória extra significativa?
  - Pilha da recursão
    - Proporcional a log$n$
    - Sim.

- **Estabilidade?**
  - Mantém a ordem relativa?
  - Tem trocas com saltos? Sim.
    - Não estável.

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

1(pivot)	2	3	5	4
	i			j

1(pivot)	2	3	5	4
	i	j		

1(pivot)	2	3	5	4
	i j			

<u>1</u>(pivot)	2	3	5	4
j	i			

<u>1</u>	**2(pivot)**	3	5	4

i j

1	**2(pivot)**	3	5	4

i j

1	**2(pivot)**	3	5	4

i j

1	**2(pivot)**	3	5	4

j i

1	2	**3**	5	4

i **j**

   - Ou seja, a cada particiona, cada elemento na sua posição correta, não contribui para a ordenação do restante uma vez que a próxima divisão ocorrerá somente uma posição à frente do particiona atual.
   - Como melhorar??

- **Complexidade assintótica**
   - **2.*n*log*n*** comparações vetor com n chaves distintas
      - Quick Sort funciona bem com entradas aleatórias
      - **O(nlogn)**
   - Pior caso: $\mathbf{n^2/2}$ comparações
      - Muito itens repetidos, (quase) ordenados, reverso caem nos piores casos
      - Soluções??

Melhorias

- **Mediana de três**
  - **Pivot**: usar a **mediana** de uma pequena amostra de itens (3)
  - Melhora o partiocionamento
  - Pivot mais à direita
    - Menor para left
    - Mediana para right

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

5			4				6

4			5				6

4			5				6

4			6				5

- Utilizando as macros (Sedegewick)

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

```
compexch(v[l], v[(l+r)/2]);
compexch(v[l], v[r]);
compexch(v[r], v[(l+r)/2]);
```

  - Pivot mais à esquerda
    - O que mudaria??
    - Implementem!

- Otimizando:
    - Menor item já está à esquerda
    - Colocar o maior item em r
        - Garantir um item maior que o pivot mais à direita
    - Fazer o particionamento de (l+1, r-1)

```
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 = partitionRSEDGEWICK(v, l+1, r-1);
```

6			4				5

6						4	5

4						6	5

4						5	6

- **Utilizar o Insertion Sort**
    - Insertion Sort é mais rápido para pequenos vetores
    - Alternar para o Insertion para pequenos vetores
    - Algo entre 5 e 15 chaves

- **Particionar o vetor em três partes (three-way)**
    - v[l..i]: elementos menores que o pivot
    - v[i+1..j-1]: elementos iguais ao pivot
    - v[j..r]: elementos maiores que o pivot