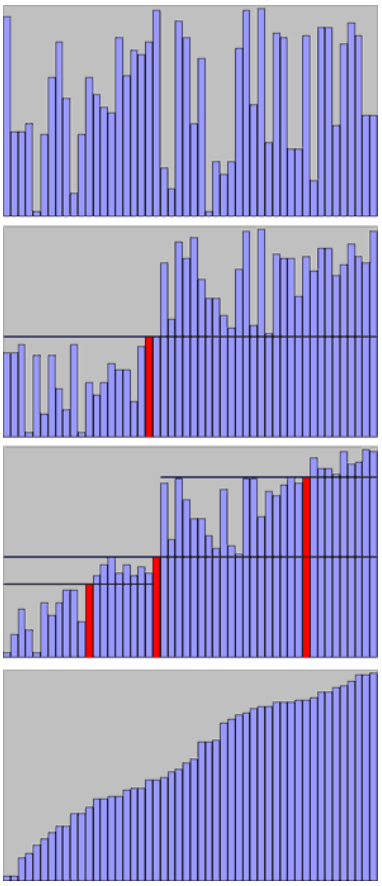
Quicksort

O algoritmo **Quicksort** é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. Ele criou o *'Quicksort* ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais facil e rapidamente. Foi publicado em 1962 após uma série de refinamentos.

O Quicksort é um algoritmo de ordenação não-estável.

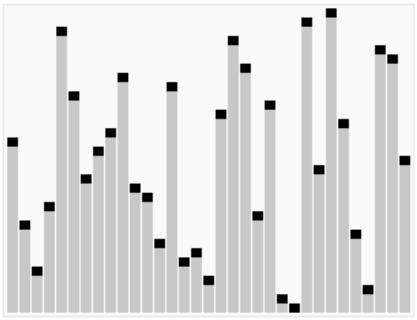


Algumas etapas do algoritmo Quicksort.

O algoritmo

O Quicksort adota a estratégia de divisão e conquista. Os passos são:

1. Escolha um elemento da lista, denominado *pivô*;
2. Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada *partição*;
3. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;



Animação do algoritmo rearranjando um conjunto de valores consecutivos embaralhados.

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

Complexidade

- Complexidade de tempo: $\theta(n \lg_2 n)$ no melhor caso e no caso médio e $\theta(n^2)$ no pior caso;
- Complexidade de espaço: $\theta(\lg_2 n)$ no melhor caso e no caso médio e $\theta(\lg_2 n)$ no pior caso. R. Sedgewick desenvolveu uma versão do Quicksort com partição recursão de cauda que tem complexidade $\theta(\lg_2 n)$ no pior caso.

Implementações

Pseudocódigo

```
procedimento QuickSort(X[], IniVet, FimVet)
var
   i, j, pivo, aux
início
   i <- IniVet
   j <- FimVet
   pivo <- X[(IniVet + FimVet) div 2]
   repita
      enquanto (X[i] < pivo) faça
      início
         i <- i + 1
      fim
      enquanto (X[j] > pivo) faça
      início
         j <- j - 1
      fim
      se (i <= j) então
      início
         aux  <- X[i]
         X[i] <- X[j]
         X[j] <- aux
         i <- i + 1
         j <- j - 1
      fim
   até_que (i <= j)
   se (j > IniVet) então
   início
      QuickSort(X, IniVet, j)
   fim
   se (i < FimVet) então
   início
      QuickSort(X, i, FimVet)
   fim
fim
```

PHP 5 - OO

```
class QuickSortUtil {

   private static function partition(&$array, $f, $l, $property) {
     $pivot = $array[$f]->$property;
     while ($f < $l) {
        while ($array[$f]->$property < $pivot) $l++;
        while ($array[$l]->$property > $pivot) $f--;
        $temp = $array[$f];
        $array[$f] = $array[$l];
        $array[$l] = $temp;
     }
     return $f;
   }

   public static function sort(&$array, $property, $f=null, $l=null) {
      if(is_null($f)) $f = 0;
      if(is_null($l)) $l = count($array)-1;
      if ($f >= $l) return;
      $pivot_index = self::partition($array, $f, $l, $property);
      self::sort($array, $property, $f, $pivot_index);
      self::sort($array, $property, $pivot_index+1, $l);
   }

}
```

Delphi (Método Recursivo)

```
procedure QuickSort(var A: array of Integer);

 procedure QuickSort(var A: array of Integer; iLo, iHi: Integer);
 var
   Lo, Hi, Mid, T: Integer;
 begin
   Lo := iLo;
   Hi := iHi;
   Mid := A[(Lo + Hi) div 2];
   repeat
     while A[Lo] < Mid do Inc(Lo);
     while A[Hi] > Mid do Dec(Hi);
     if Lo <= Hi then
     begin
       T := A[Lo];
       A[Lo] := A[Hi];
       A[Hi] := T;
       Inc(Lo);
       Dec(Hi);
     end;
```

```
   until Lo > Hi;
   if Hi > iLo then QuickSort(A, iLo, Hi);
   if Lo < iHi then QuickSort(A, Lo, iHi);
 end;

begin
 QuickSort(A, Low(A), High(A));
end;

{Chamando em um evento de onClick}
procedure TForm1.Button1Click(Sender: TObject);
var
 arr: array[0..100] of integer;
 I: Integer;
begin
 for I:=Low(arr) to High(arr) do
   arr[I]:=Random(High(Integer));

 Quick_Sort(arr);
end;
```

Visual Basic

```
Sub QuickSort(ByRef vetor() As Integer, ByVal inicio As Integer, ByVal final As Integer, ByRef iteracoes As Long)
    Dim pivo As Integer
    Dim i As Integer
    Dim j As Integer
    iteracoes = iteracoes + 1
    If final > inicio Then
        i = inicio
        j = final
        pivo = vetor(Fix((inicio + final) / 2))
        While i <= j
            While vetor(i) < pivo
                i = i + 1
            Wend
            While pivo < vetor(j)
                j = j - 1
            Wend
            If i <= j Then
                Call Troca(vetor(i), vetor(j))
                i = i + 1
                j = j - 1
            End If
        Wend
        Call QuickSort(vetor, inicio, j, iteracoes)
        Call QuickSort(vetor, i, final, iteracoes)
    End If
```

```
End Sub


Sub Troca(ByRef val1 As Integer, ByRef val2 As Integer)
    Dim aux As Integer
    aux = val1
    val1 = val2
    val2 = aux
End Sub
```

Python

```
def pivot(v, left, right):
    i = left
    for j in range(left + 1, right + 1):
        if v[j] < v[left]: # Se um elemento j for menor que o pivo
            i += 1 # .. incrementa-se i
            v[i], v[j] = v[j], v[i] # .. e troca o elemento j de
posicao o elemento i
    v[i], v[left] = v[left], v[i] # O pivo e' colocado em sua posicao
final
    return i

def qsort(v, left, right):
    if right > left:
        r = pivot(v, left, right) # Ordena um elemento
        qsort(v, left, r - 1) # .. entao ordena-se os dois sub-vetores
        qsort(v, r + 1, right)


# Exemplo
a = [4,2,4,6,3,2,5,1,3]
qsort(a, 0, 8)
print a
```

Assembly x86-gas-Linux

```
/*void quicksort_as (int *x, int n);*/
.globl quicksort_as
quicksort_as:
      pushl %ebp
      movl %esp, %ebp
      /* 8(%ebp)= ponteiro do arranjo */
      /* 12(%ebp)= num de elementos */
      movl 12(%ebp), %eax
      dec %eax
      movl $4, %ebx
      mul %ebx
      pushl %eax
```

```
      pushl $0
      pushl 8(%ebp)
      call quicksort_as_
      leave
      ret
quicksort_as_:
      pushl %ebp
      movl %esp, %ebp
      /* 8(%ebp)= ponteiro do arranjo */
      /* 12(%ebp)= esq */
      /* 16(%ebp)= dir */
      movl 12(%ebp), %ecx
      movl %ecx, %edx
      movl 8(%ebp), %eax
      addl %ecx, %eax
      movl 16(%ebp), %ecx
      movl 8(%ebp), %ebx
      addl %ecx, %ebx
      /* agora %eax aponta p/ x[esq] e %ebx x[dir]*/
      addl %edx, %ecx         /*%ecx eh esq + dir*/
      pushl %eax
      movl %ecx, %eax
      movl $2, %ecx
      cltd
      idivl %ecx                        /* div %eax por 2=%ecx*/
      movl $4, %ecx
      cltd
      idivl %ecx
      mul  %ecx
      movl 8(%ebp), %ecx
      addl %eax, %ecx
      movl (%ecx), %ecx
      popl %eax
      /*%ecx = compare(cmp)*/
quicksort_imj:
      cmp %eax, %ebx
      jle quicksort_fim
quicksort_inci:
      cmp (%eax), %ecx
      jle quicksort_incj
      addl $4, %eax
      jmp quicksort_inci
quicksort_incj:
      cmp (%ebx), %ecx
      jge quicksort_troca
      subl $4, %ebx
      jmp quicksort_incj
```

```
quicksort_troca:
      cmp %eax, %ebx
      jl quicksort_fim
      movl (%ebx), %edx
      pushl (%eax)
      movl %edx, (%eax)
      popl (%ebx)
      addl $4, %eax
      subl $4, %ebx
      jmp quicksort_imj
quicksort_fim:
      /*salvando registradores na pilha p/ fazer chamada de função*/
      pushl %eax
      pushl %ebx
      /*passando parametros p/ chamada recursiva*/
      subl 8(%ebp), %eax
      cmp %eax, 16(%ebp)
      jle quicksort_2a_rec
      pushl 16(%ebp)
      pushl %eax
      pushl 8(%ebp)
      call quicksort_as_
      addl $12, %esp
quicksort_2a_rec:
      /*recuperando registradores apos chamada de funcao*/
      popl %ebx
      popl %eax
      subl 8(%ebp), %ebx
      cmp %ebx, 12(%ebp)
      jge quicksort_final
      /*passando parametros p/ chamada recursiva*/
      pushl %ebx
      pushl 12(%ebp)
      pushl 8(%ebp)
      call quicksort_as_
quicksort_final:
      leave
      ret
```

Haskell

```
  sort :: (Ord a)   => [a] -> [a]
  sort []           = []
  sort (pivot:rest) = (sort [y | y <- rest, y < pivot])
                       ++ [pivot] ++
                      (sort [y | y <- rest, y >=pivot])
```

Lisp

```
(defun partition (fun array)
  (list (remove-if-not fun array) (remove-if fun array)))

(defun sort (array)
  (if (null array) nil
    (let ((part (partition (lambda (x) (< x (car array))) (cdr array))))
      (append (sort (car part)) (cons (car array) (sort (cadr part)))))))
```

Perl

Versões anteriores ao Perl 5.6 utilizavam o algoritmo quicksort para implementar a função sort [1][2] , então o código em Perl pode resumir-se a:

```
my @ordenada = sort @lista;
```

Como a implementação do quicksort pode assumir tempos quadráticos para algumas entradas, o algoritmo foi substituído na versão 5.8 pelo mais estável mergesort, cujo pior caso é $\theta(n \lg_2 n)$. Ainda assim, é possível forçar o uso do quicksort através do pragma 'sort' [3]:

```
use sort '_quicksort';
my @ordenada = sort @lista;
```

Uma implementação em Perl puro (mais lenta que o 'sort' embutido) pode ser:

```
sub quicksort {
    my @lista = @_;
    my (@menores, @iguais, @maiores);

    return @lista if @lista < 2;
    foreach (@lista) {
        if ($_ < $lista[0]) {
            push @menores, $_;
        }
        elsif ($_ = $lista[0]) {
            push @iguais, $_;
        }
        else {
            push @maiores, $_;
        }
    }
    return quicksort(@menores), @iguais, quicksort(@maiores);
}
```

Já uma versão menor (e certamente menos legível) poderia ser:

```
sub quicksort {
    return @_ if @_ < 2;
    my (@iguais, @maiores, @menores);
    push @{ (\@iguais, \@menores, \@maiores)[ $_[0] <=> $_ ]}, $_ for @_;
    quicksort(@menores), @iguais, quicksort(@maiores);
```

```
}
```

Ruby

```ruby
def sort( array )
   return array if array.size <= 1
   pivot = array[0]
   return sort( array.select { |y| y < pivot } )
          + array.select { |y| y == pivot } +
          sort( array.select { |y| y > pivot } )
end
```

Groovy

```
def sort(list) {
    if (list.isEmpty()) return list
    anItem = list[0]
    def smallerItems = list.findAll{it < anItem}
    def equalItems = list.findAll{it = anItem}
    def largerItems = list.findAll{it > anItem}
    sort(smallerItems) + equalItems + sort(largerItems)
}
```

ML

```
fun filter nil elem cmp = nil
  | filter (x::xl) elem cmp  =
      if (cmp(x, elem))
        then x :: filter xl elem cmp
        else filter xl elem cmp;

fun quicksort nil = nil
  | quicksort (pivot::xl) =
      let
        val small = filter xl pivot (op <);
        val medium = pivot :: filter xl pivot (op =);
        val large = filter xl pivot (op >);
      in
        (quicksort small) @ medium @ (quicksort large)
      end;
```

PHP

```php
<?php
/**********************************************************************
*Obs.: a implementação usa parâmetros por referência, isto é, o vetor
*passado será modificado.
**********************************************************************/

function troca(&$v1, &$v2){
```

```
    $vaux = $v1;
    $v1 = $v2;
    $v2 = $vaux;
}

//divide o array em dois
function divide(&$vet, $ini, $fim){
    $i = $ini;
    $j = $fim;
    $dir = 1;
    while ($i > $j){
        if ($vet[$i] < $vet[$j]){
            troca($vet[$i], $vet[$j]);
            $dir = - $dir;
        }
        if ($dir = 1) {
            $j--;
        }else{
            $i++;
        }
    }
    return $i;
}
//ordena
function quicksort(&$vet, $ini, $fim){
    if ($ini < $fim){
        $k = divide($vet, $ini, $fim);
        quicksort($vet, $ini, $k-1);
        quicksort($vet, $k+1, $fim);
    }
}
quicksort($vet,0,count($vet));
?>
```

C++

```
#include <algorithm>
#include <iterator>
#include <functional>
using namespace std;

template <typename T>
void sort(T begin, T end) {
    if (begin != end) {
        T middle = partition (begin, end, bind2nd(less<iterator_traits<T>::value_type>(),
*begin));
        sort (begin, middle);
        sort (max(begin + 1, middle), end);
```

```
    }
}
```

C

```c
void swap(int* a, int* b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

int partition(int vec[], int left, int right) {
  int i, j;

  i = left;
  for (j = left + 1; j <= right; ++j) {
    if (vec[j] < vec[left]) {
      ++i;
      swap(&vec[i], &vec[j]);
    }
  }
  swap(&vec[left], &vec[i]);

  return i;
}

void quickSort(int vec[], int left, int right) {
  int r;

  if (right > left) {
    r = partition(vec, left, right);
    quickSort(vec, left, r - 1);
    quickSort(vec, r + 1, right);
  }
}
```

Implementaçao usando 'fat pivot':

```c
void sort(int array[], int begin, int end) {
   int pivot = array[begin];
   int i = begin + 1, j = end, k = end;
   int t;

   while (i < j) {
      if (array[i] < pivot) i++;
      else if (array[i] > pivot) {
         j--; k--;
         t = array[i];
```

```
            array[i] = array[j];
            array[j] = array[k];
            array[k] = t; }
         else {
            j--;
            swap(array[i], array[j]);
      }  }
      i--;
      swap(array[begin], array[i]);
      if (i - begin > 1)
         sort(array, begin, i);
      if (end - k   > 1)
         sort(array, k, end);
}
```

Lembrando que quando você for chamar a função recursiva terá que chamar a mesma da seguinte forma ordenar_quicksort_nome(0,n-1). O 0(zero) serve para o início receber a posição zero do vetor e o fim será o tamanho do vetor -1.

```
void ordenar_quicksort_nome(int ini, int fim)
{
 int i = ini, f = fim;
 char pivo[50];
 strcpy(pivo,vetor[(ini+fim)/2]);
 if (i<=f)
  {
   while (strcmpi(vetor[i],pivo)<0) i++;
   while (strcmpi(vetor[f],pivo)>0) f--;
   if (i<=f)
    {
     strcpy (aux_char,vetor[i]);
     strcpy (vetor[i],vetor[f]);
     strcpy (vetor[f],aux_char);
     i++; f--;
    }
  }
  if (f>ini) ordenar_quicksort_nome(ini,f);
  if (i<fim) ordenar_quicksort_nome(i,fim);
}
```

Java

Exemplo em J2SE 5.0 com método para teste.

```
import java.util.Arrays;
import java.util.Random;

public class QuickSort {
    public static final Random RND = new Random();
```

```
    private static void swap(Object[] array, int i, int j) {
        Object tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }

    private static <E extends Comparable<? super E>> int partition(E[] array, int begin,
        int index = begin + RND.nextInt(end - begin + 1);
        E pivot = array[index];
        swap(array, index, end);
        for (int i = index = begin; i < end; ++i) {
            if (array[i].compareTo(pivot) <= 0) {
                swap(array, index++, i);
            }
        }
        swap(array, index, end);
        return (index);
    }

    private static <E extends Comparable<? super E>> void qsort(E[] array, int begin, int
        if (end > begin) {
            int index = partition(array, begin, end);
            qsort(array, begin, index - 1);
            qsort(array, index + 1, end);
        }
    }

    public static <E extends Comparable<? super E>> void sort(E[] array) {
        qsort(array, 0, array.length - 1);
    }

    // Exemplo de uso
    public static void main(String[] args) {

        // Ordenando Inteiros
        Integer[] l1 = { 5, 1024, 1, 88, 0, 1024 };
        System.out.println("l1  start:" + Arrays.toString(l1));
        QuickSort.sort(l1);
        System.out.println("l1 sorted:" + Arrays.toString(l1));

        // Ordenando Strings
        String[] l2 = { "gamma", "beta", "alpha", "zoolander" };
        System.out.println("l2  start:" + Arrays.toString(l2));
        QuickSort.sort(l2);
        System.out.println("l2 sorted:" + Arrays.toString(l2));
    }
}
```

Fonte: Wikibooks QuickSort [4]

C#

```
static class QuickSort
{
   public static void Ordenar(int[] vetor) {
      Ordenar(vetor, 0, vetor.Length - 1);
   }

   private static void Ordenar(int[] vetor, int inicio, int fim) {
      if (inicio < fim) {
         int posicaoPivo = Separar(vetor, inicio, fim);
         Ordenar(vetor, inicio, posicaoPivo - 1);
         Ordenar(vetor, posicaoPivo + 1, fim);
      }
   }

   private static int Separar(int[] vetor, int inicio, int fim) {
      int pivo = vetor[inicio];
      int i = inicio + 1, f = fim;
      while (i <= f) {
         if (vetor[i] <= pivo)
            i++;
         else if (pivo < vetor[f])
            f--;
         else {
            int troca = vetor[i];
            vetor[i] = vetor[f];
            vetor[f] = troca;
            i++;
            f--;
         }
      }
      vetor[inicio] = vetor[f];
      vetor[f] = pivo;
      return f;
   }
}
```

Assembly x86

```
  qsort:  @ Takes three parameters:
        @   a:     Pointer to base of array a to be sorted (arrives in r0)
        @   left:  First of the range of indexes to sort (arrives in r1)
        @   right: One past last of range of indexes to sort (arrives in r2)
        @ This function destroys: r1, r2, r3, r4, r5, r7
        stmfd   sp!, {r4, r6, lr}     @ Save r4 and r6 for caller
        mov     r6, r2                @ r6 <- right
```

```
  qsort_tailcall_entry:
        sub     r7, r6, r1            @ If right - left <= 1 (already sorted),
        cmp     r7, #1
        ldmlefd sp!, {r1, r6, pc}     @ Return, moving r4->r1, restoring r6
        ldr     r7, [r0, r1, asl #2]  @ r7 <- a[left], gets pivot element
        add     r2, r1, #1            @ l <- left + 1
        mov     r4, r6                @ r <- right
  partition_loop:
        ldr     r3, [r0, r2, asl #2]  @ r3 <- a[l]
        cmp     r3, r7                @ If a[l] <= pivot_element,
        addle   r2, r2, #1            @ ... increment l, and
        ble     partition_test        @ ... continue to next iteration.
        sub     r4, r4, #1            @ Otherwise, decrement r,
        ldr     r5, [r0, r4, asl #2]  @ ... and swap a[l] and a[r].
        str     r5, [r0, r2, asl #2]
        str     r3, [r0, r4, asl #2]
  partition_test:
        cmp     r2, r4                @ If l < r,
        blt     partition_loop        @ ... continue iterating.
  partition_finish:
        sub     r2, r2, #1            @ Decrement l
        ldr     r3, [r0, r2, asl #2]  @ Swap a[l] and pivot
        str     r3, [r0, r1, asl #2]
        str     r7, [r0, r2, asl #2]
        bl      qsort                 @ Call self recursively on left part,
                                      @  with args a (r0), left (r1), r (r2),
                                      @  also preserves r6 and
                                      @  moves r4 (l) to 2nd arg register (r1)
        b       qsort_tailcall_entry  @ Tail-call self on right part,
                                      @  with args a (r0), l (r1), right (r6)
```

Prolog

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])     --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Lua

```lua
function quickSort(v, ini, fim)
	i=ini
	j=fim
	pivo= v[ math.floor((ini + fim)/2) ] --Math.floor trunca o número

	while(i<=j)
	do
		while (v[i] < pivo)
		do
			i=i+1
		end

		while (v[j] > pivo)
		do
			j=j-1
		end

		if(i<=j)
		then
			v[i],v[j]=v[j],v[i] --Troca de valores
			i=i+1
			j=j-1
		end
	end

	if(j>ini)
	then
		quickSort(v, ini, j)
	end

	if(i<fim)
	then
		quickSort(v, i, fim)
	end
end
```

Veja também

- Ordenação de vetor
- Merge sort
- Heapsort
- Selection sort
- Bubble sort
- Busca linear

[1] http://perldoc.perl.org/functions/sort.html
[2] *Documentação oficial da função 'sort' em Perl* (http://perldoc.perl.org/functions/sort.html). perl.org. Página visitada em 2008-10-20.
[3] http://perldoc.perl.org/sort.html
[4] http://en.wikibooks.org/wiki/Algorithm_implementation/Sorting/Quicksort

Ligações externas

- Rápida aula de Quicksort (http://www.ime.usp.br/~pf/algoritmos/aulas/quick.html)
- Animação do processo de ordenação pelo Quicksort (http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html)
- Explanação video de Quicksort usando cartões e de código em C++ (http://www.datastructures.info/what-is-quicksort-and-how-does-it-work-quick-sort-algorithm/)
- QuickSort Code (http://www.algorithm-code.com/wiki/Quick_Sort)

Fontes e Editores da Página

Quicksort *Fonte*: http://pt.wikipedia.org/w/index.php?oldid=20493959 *Contribuidores*: Adailton, Al3xeng, Bertoche, Bisbis, BrunoSupremo, Camponez, Carnevalli, Ccuembej, ChristianH, Diego UFCG, Dirceu Júnior, Dobau, E2m, EduM, Eparente, Eric Duff, Fabiano Tatsch, Fernando Mussio, Firmo, Garoto burns, Gbiten, Ghisi.cintia, JoaoMiranda, Josehneto, Josepojr, Kaze Senshi, LeonardoG, LeonardoRob0t, Malafaya, Manuel Anastácio, Marcelo Reis, Mauro schneider, Nuno Tavares, OS2Warp, Osias, PatríciaR, Pedro7x, Rafaelcosta1984, Rod, Ruy Pugliesi, Salgueiro, Sturm, Thiagoharry, Vini 175, 126 edições anónimas

Fontes, licenças e editores da imagem

Ficheiro:Quicksort example small.png *Fonte*: http://pt.wikipedia.org/w/index.php?title=Ficheiro:Quicksort_example_small.png *Licença*: Public Domain *Contribuidores*: Maksim, 1 edições anónimas

Ficheiro:Sorting quicksort anim.gif *Fonte*: http://pt.wikipedia.org/w/index.php?title=Ficheiro:Sorting_quicksort_anim.gif *Licença*: Creative Commons Attribution-Sharealike 2.5 *Contribuidores*: Berrucomons, Cecil, Chamie, Davepape, Diego pmc, Editor at Large, German, Gorgo, Howcheng, Jago84, Jutta234, Lokal Profil, MaBoehm, Minisarm, Miya, Mywood, NH, PatríciaR, Qyd, Soroush83, Stefeck, Str4nd, 11 edições anónimas

Licença