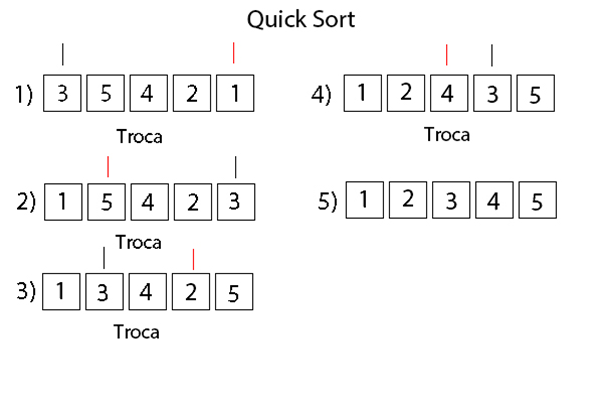
**Quick sort**

O Quicksort é o algoritmo mais eficiente na ordenação por comparação. Nele se escolhe um elemento chamado de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada.



* O número 3 foi escolhido como pivô, nesse passo é procurado à sua direita um número menor que ele para ser passado para a sua esquerda. O primeiro número menor encontrado foi o 1, então eles trocam de lugar.
* Agora é procurado um número à sua esquerda que seja maior que ele, o primeiro número maior encontrado foi o 5, portanto eles trocam de lugar.
* O mesmo processo do passo 1 acontece, o número 2 foi o menor número encontrado, eles trocam de lugar.
* O mesmo processo do passo 2 acontece, o número 4 é o maior número encontrado, eles trocam de lugar.
* O vetor desse exemplo é um vetor pequeno, portanto ele já foi ordenado, mas se fosse um vetor grande, ele seria dividido e recursivamente aconteceria o mesmo processo de escolha de um pivô e comparações.

A principal desvantagem deste método é que ele possui uma implementação difícil e delicada, um pequeno engano pode gerar efeitos inesperados para determinadas entradas de dados.

**Código do método**

#include *<iostream>*

void quicksort(int values[], int began, int end)

{

int i, j, pivo, aux;

i = began;

j = end-1;

pivo = values[(began + end) / 2];

**while**(i <= j)

{

**while**(values[i] < pivo && i < end)

{

i++;

}

**while**(values[j] > pivo && j > began)

{

j--;

}

**if**(i <= j)

{

aux = values[i];

values[i] = values[j];

values[j] = aux;

i++;

j--;

}

}

**if**(j > began)

quicksort(values, began, j+1);

**if**(i < end)

quicksort(values, i, end);

}

int main(int argc, char \*argv[])

{

int array[10] = {5, 8, 1, 2, 7, 3, 6, 9, 4, 10};

**for**(int i = 0; i < 10; i++)

{

std::cout << array[i] << ' ';

}

std::cout << std::endl;

quicksort(array, 0, 10);

**for**(int i = 0; i < 10; i++)

{

std::cout << array[i] << ' ';

}

**return** 0;

}

**Merge Sort**

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

O diagrama a seguir da [Wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg" \t "_blank) mostra o processo completo de classificação por mesclagem para um exemplo de matriz {38, 27, 43, 3, 9, 82, 10}. Se olharmos mais de perto o diagrama, podemos ver que a matriz é dividida recursivamente em duas metades até que o tamanho se torne 1. Uma vez que o tamanho torna-se 1, os processos de mesclagem entram em ação e começam a mesclar as matrizes de volta até que a matriz completa seja fundido.



### Desvantagens

* Utiliza funções recursivas;
* Gasto extra de memória. O algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a {\displaystyle O(n\log n)}.

**Código do Método**

void merge(int \*saida, int \*auxiliar, int inicio, int meio, int fim){

int i, j, k;

i = inicio;

j = meio + 1;

k = inicio;

**while**(i <= meio && j <= fim){

**if**(auxiliar[i] < auxiliar[j]){

saida[k] = auxiliar[i];

i++;

}

**else**{

saida[k] = auxiliar[j];

j++;

}

k++;

}

**while**(i <= meio){

saida[k] = auxiliar[i];

i++;

k++;

}

**while**(j <= fim){

saida[k] = auxiliar[j];

j++;

k++;

}

*//Copia os elementos que foram ordenados para o auxiliar*

**for**(int p = inicio; p <= fim; p++)

auxiliar[p] = saida [p];

}

void mergeSort(int \*saida, int \*auxiliar, int inicio, int fim){

**if**(inicio < fim){

int meio = (inicio + fim) / 2;

mergeSort(saida, auxiliar, inicio, meio);

mergeSort(saida, auxiliar, meio + 1, fim);

merge(saida, auxiliar, inicio, meio, fim);

}

}