Algoritmos de Ordenação

Docente: Gil Jader – gil.jader@gmail.com

1. Algoritmo de Ordenação

- em **ciência da computação** é um algoritmo que coloca os elementos de uma dada sequência em uma certa **ordem** -- em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica.

- A forma geral é:

Dada uma lista ordenada de elementos, então:

$$i_1 <= i_2 <= \ldots <= i_n$$

1. Algoritmo de Ordenação

Métodos Simples

- Insertion
- Selection sort
- Bubble sort
- Comb sort

Métodos sofisticados

- Merge sort
- Heapsort
- Shell sort
- Radix sort
- Gnome sort
- Count sort
- Bucket sort
- Cocktail sort
- Timsort
- Quick sort

2. Bubble Sort

- Bubble sort é o algoritmo **mais simples**, mas o menos eficiente.
- Neste algoritmo cada elemento da posição i será comparado com o elemento da posição **i + 1**, ou seja, um elemento da posição 2 será comparado com o elemento da posição 3.
- Caso o elemento da posição 2 for maior que o da posição 3, eles trocam de lugar e assim sucessivamente.
- Por causa dessa forma de execução, o vetor terá que ser percorrido quantas vezes que for necessária, tornando o algoritmo ineficiente para **listas muito grandes**.
- No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vector. No pior caso, são feitas $\mathbf{n^2}$ operações.
- A **complexidade** desse algoritmo é de **Ordem quadrática**. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

2. Bubble Sort

Com a ordenação bolha, o número de comparações é sempre o mesmo, porque os dois laços for repetem o número especificado de vezes, estando a lista inicialmente ordenada ou não. Isso significa que a ordenação bolha sempre executa:

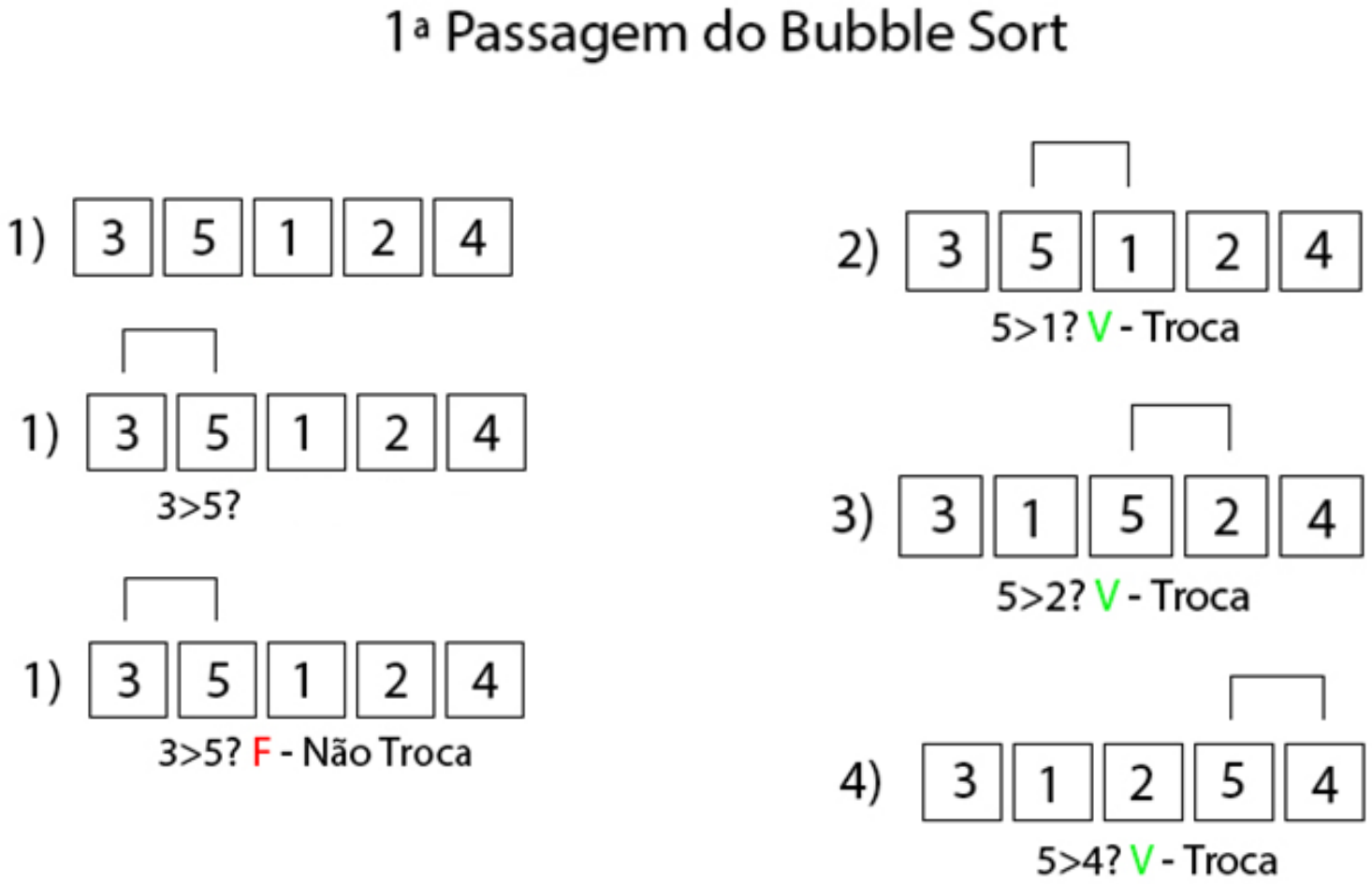
$\frac{1}{2}(n^2-n)$ comparações, onde n é o número de elementos a ser ordenado.

2. Bubble Sort

Algoritmo:

- Percorra o vetor inteiro comparando elementos adjacentes (dois a dois)
- Troque as posições dos elementos se eles estiverem fora de ordem
- Repita os dois passos acima com os primeiros n-1 itens, depois com os primeiros n-2 itens, até que reste apenas um item;

2. Bubble Sort



2. Bubble Sort

Outro exemplo:

Resultado da 1ª interação:

Resultado da 2ª interação:

Resultado da 3ª interação:

Resultado da 4ª interação:

Resultado da 5ª interação:

Resultado da 6ª interação:

Resultado da 7ª interação:

Resultado da 8ª interação:

2. Bubble Sort

Exercício

Ordene um vetor de inteiros de n elementos de forma crescente utilizando-se do método de ordenação Bubble sort.

2. Bubble Sort

Ordenação de String

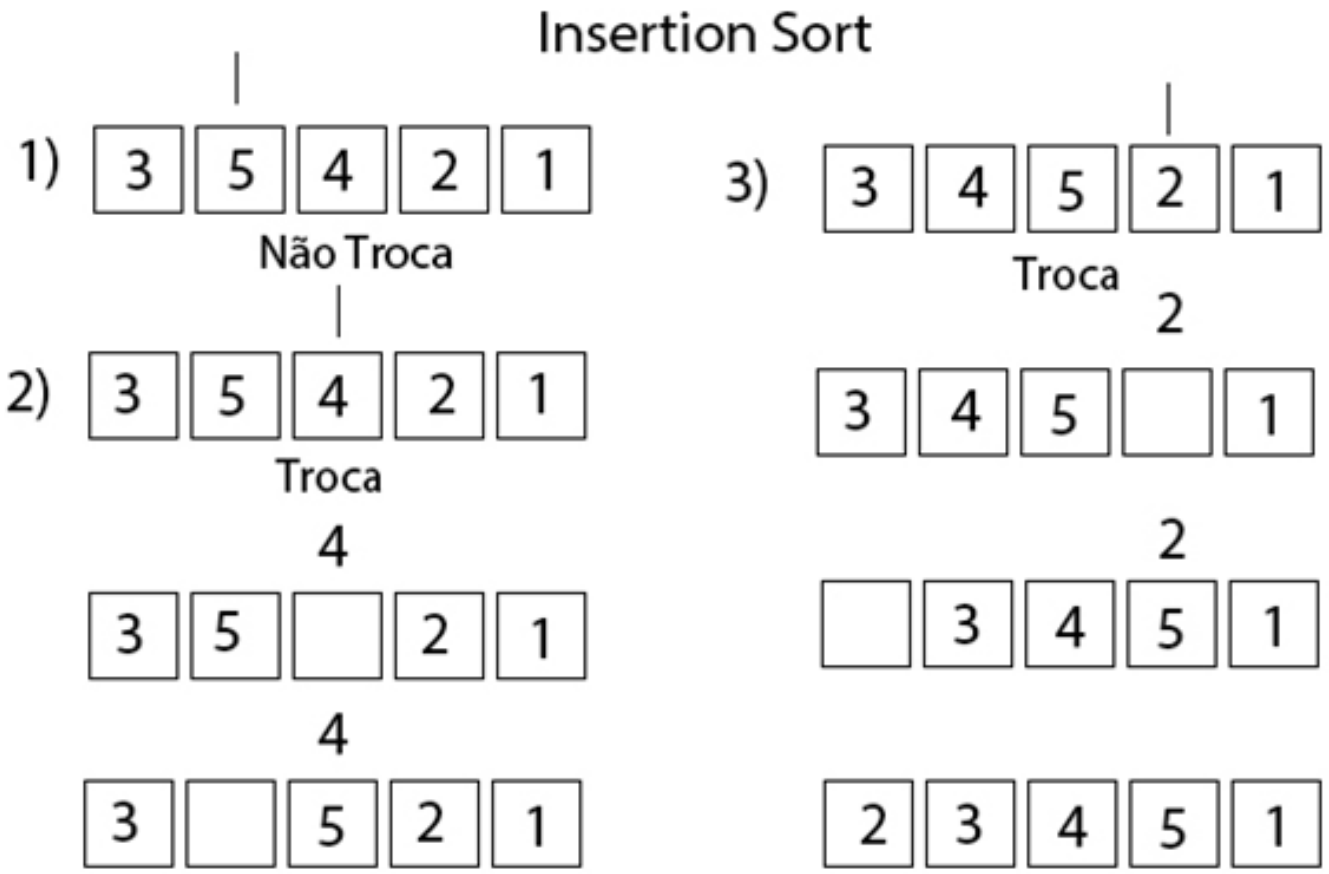
```
for(i = 1; i < soma; i++){
        for (j = 0; j < soma-1; j++){

            if(strcmp(vet[j].nome,vet[j+1].nome) > 0){
              strcpy(temp,vet[j].nome);
               strcpy(vet[j].nome,vet[j+1].nome);
               strcpy(vet[j+1].nome,temp);
            }
      }
    }
```

- **strcmp(s1,s2)** – Retorna 0 se s1 e s2 são iguais; menor que 0 se s1<s2; maior que 0 se s1>s2 (comparação alfabética).

3. Insertion Sort

- O **Insertion sort** é um algoritmo simples e eficiente quando aplicado em pequenas listas. Neste algoritmo a lista é percorrida da esquerda para a direita, à medida que avança vai deixando os elementos mais à esquerda ordenados.
- Este método consiste em realizar a ordenação pela inserção de cada um dos elementos em sua posição correta, levando em consideração os elementos já ordenados.
- Semelhante a organizar cartas no baralho.
- O vertor é dividido em dois segmentos: o primeiro contendo os valores já classificados e o segundo contendo os elementos ainda não classificados
- Inicialmente, o primeiro segmento contém apenas o primeiro elemento do vetor e o segundo contém todos os demais elementos

3. Insertion Sort



3. Insertion Sort

3. Insertion Sort

```
void insertionSortD(int array[], int tamanho) {
	int i, j, tmp;
	for (i = 1; i < tamanho; i++) {
		j = i;
		while (j > 0 && array[j - 1] < array[j]) {
			tmp = array[j];
			array[j] = array[j - 1];
			array[j - 1] = tmp;
			j--;
		}
	}
}
void insertionSortC(int array[], int tamanho) {
	int i, j, tmp;
	for (i = 1; i < tamanho; i++) {
		j = i;
		while (j > 0 && array[j - 1] > array[j]) {
			tmp = array[j];
			array[j] = array[j - 1];
			array[j - 1] = tmp;
			j--;
		}
	}
}
```

4. Selection Sort

Este processo de ordenação consiste em uma seleção sucessiva do **menor** ou do **maior** valor contido no vetor, dependendo se a ordenação dos elementos será em ordem **crescente** ou **decrescente.**

4. Selection Sort

Método:

A cada passo, o elemento de menor (ou maior) valor é **selecionado** e colocado em sua **posição correta** dentro do vetor ordenado.

Esse processo é **repetido** para o segmento do vetor que contém os elementos ainda não selecionados.

4. Selection Sort

Método:

O vetor é dividido em dois segmentos: o **primeiro** contendo os **valores já ordenados** e o **segundo** contendo os **elementos ainda não selecionados**

Inicialmente, o primeiro segmento está vazio e o segundo segmento contém todos os elementos do vetor.

4. Selection Sort

Algoritmo

1. É feita uma varredura no segmento que contém os elementos ainda não selecionados, identificando o elemento de **menor** (ou maior) valor.

2. O elemento identificado no passo 1 é inserido no segmento ordenado na **última posição.**

3. O **tamanho do segmento** que contém os elementos ainda não selecionados é atualizado, ou seja, **diminuído de 1.**

4. O processo é repetido até que este segmento fique com apenas um elemento, que é o **maior(ou menor)** valor do vetor.

4. Selection Sort

A ordenação de um vetor de **n** elementos é feita pela execução de **n-1** passos sucessivos:

Em cada passo, determina-se aquele de **menor valor** dentre os elementos ainda **não selecionados**

4. Selection Sort

- No **primeiro passo**, são feitas **n-1 comparações** para a determinação do **menor valor;**
- No **segundo passo**, **n-2 comparações**, e assim sucessivamente ….
- Até que no **último passo** é efetuada apenas **uma comparação.**

4. Selection Sort

- O número total de comparações é dado por:

  NC = (n-1) + (n-2) + (n-3) + … + 2 + 1

- Essa sequência representa a soma de uma progressão aritmética que pode ser generalizada com a seguinte fórmula:

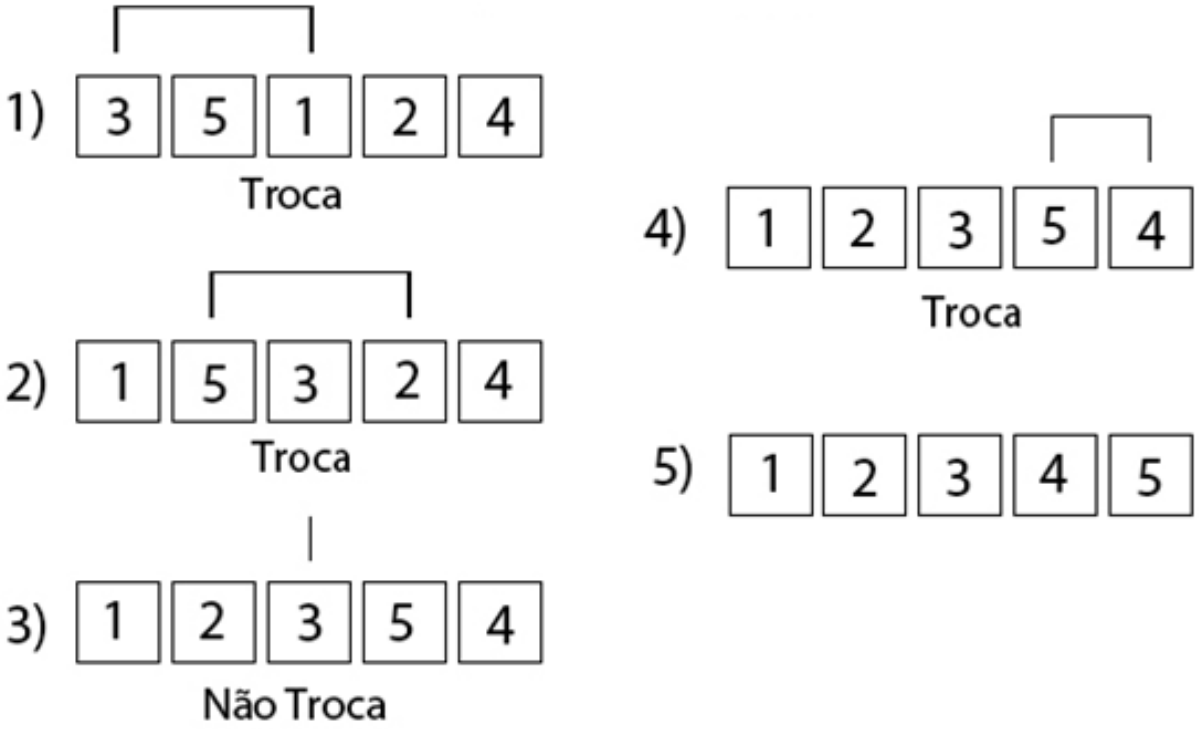
  NC = (((n-1)+1)/2)*(n-1) = (n^2 - n)/2

4. Selection Sort

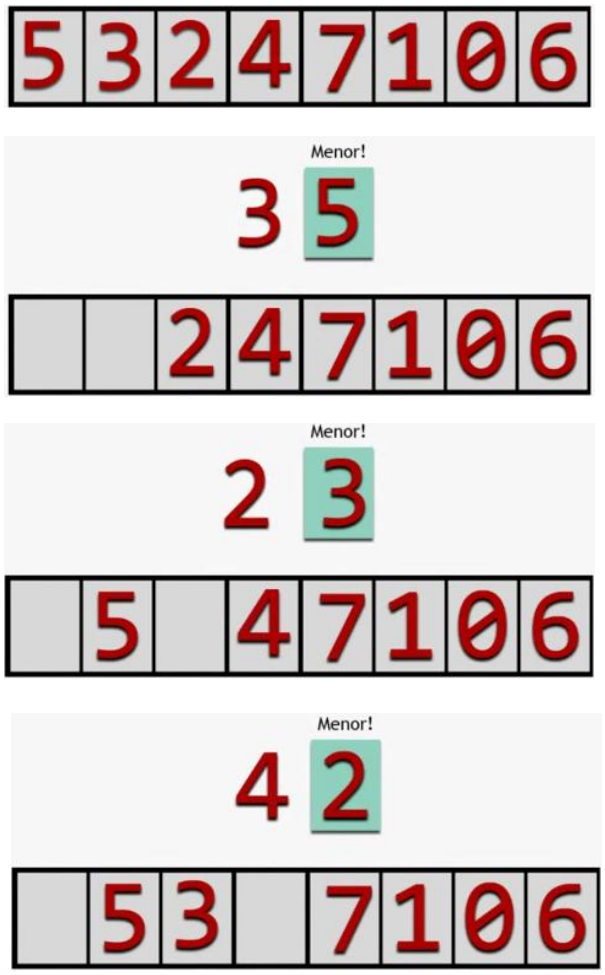
- O desempenho médio do método é da ordem de n^2 **$O(n^2)$**, ou seja, é proporcional ao quadrado do número de elementos do vetor.

- Esse método **não é indicado** para vetores com muito elementos.

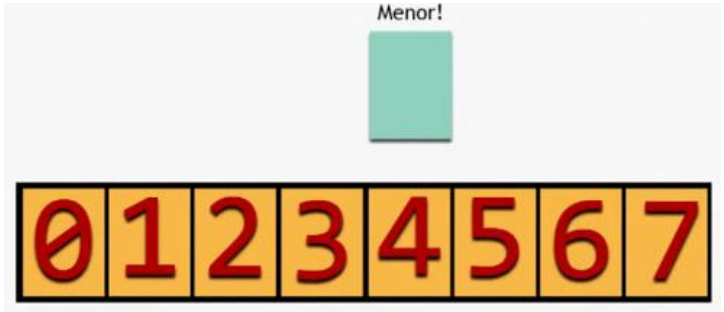
4. Selection Sort



4. Selection Sort



4. Selection Sort



4. Selection Sort

```
#include <stdio.h>
#include <stdlib.h>

void selectionSort(int v[200], int n)
{
    int i, j, aux, min;
    for(i = 0; i < n-1; i++) {
        min = i;
        for(j = i+1; j < n; j++) {
            if(v[j] < v[min]) {
                min = j;
            }
        }
        aux = v[i]; v[i] = v[min]; v[min] = aux; //troca
    }
}
```

5. Quicksort

Histórico:

- É um método de ordenação, inventado por C.A.R. Hoare em 1960;
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações;
- Provavelmente é o mais utilizado.

5. Quicksort

Algoritmo:

- Dividir o problema de ordenar um conjunto com n itens em dois problemas menores
- Os problemas menores são ordenados independentemente
- As partições são combinadas para produzir a solução final

5. Quicksort

Particionamento:

- A parte mais delicada do quicksort é o processo de partição
- O vetor **v** é rearranjado por meio da escolha arbitrária de um *pivô* **p**
- O vetor **v** é particionado em dois:
  - Partição esquerda: chaves ≤ p
  - Partição direita: chaves ≥ p

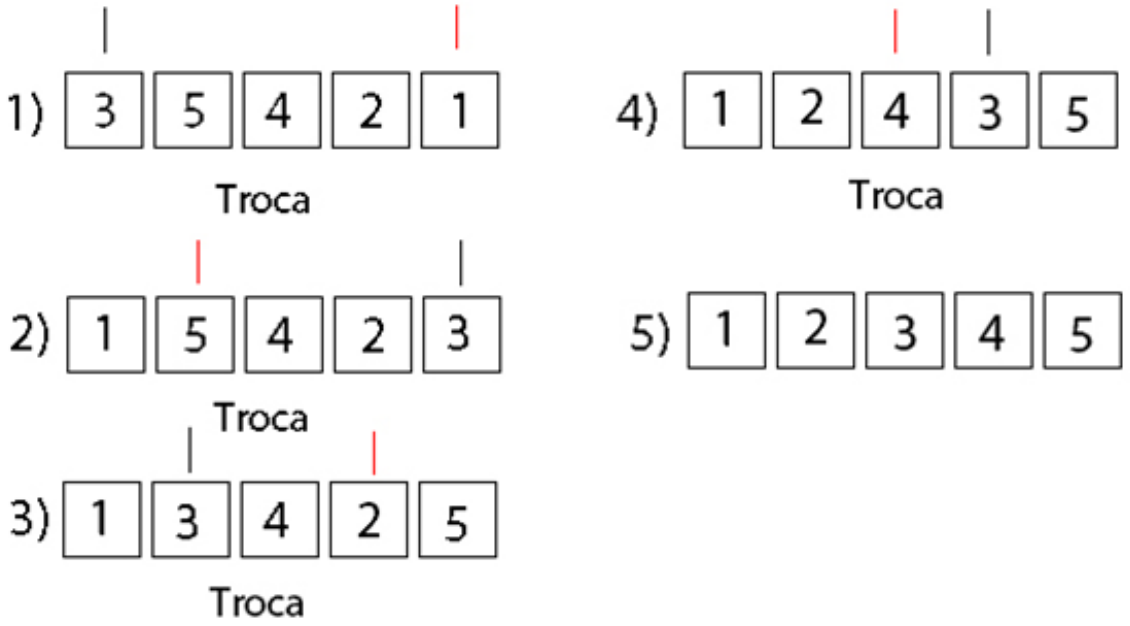
5. Quicksort

Passos:

- Escolha arbitrariamente o *pivô* **p**
- 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*;
- Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

5. Quicksort

Exemplo:

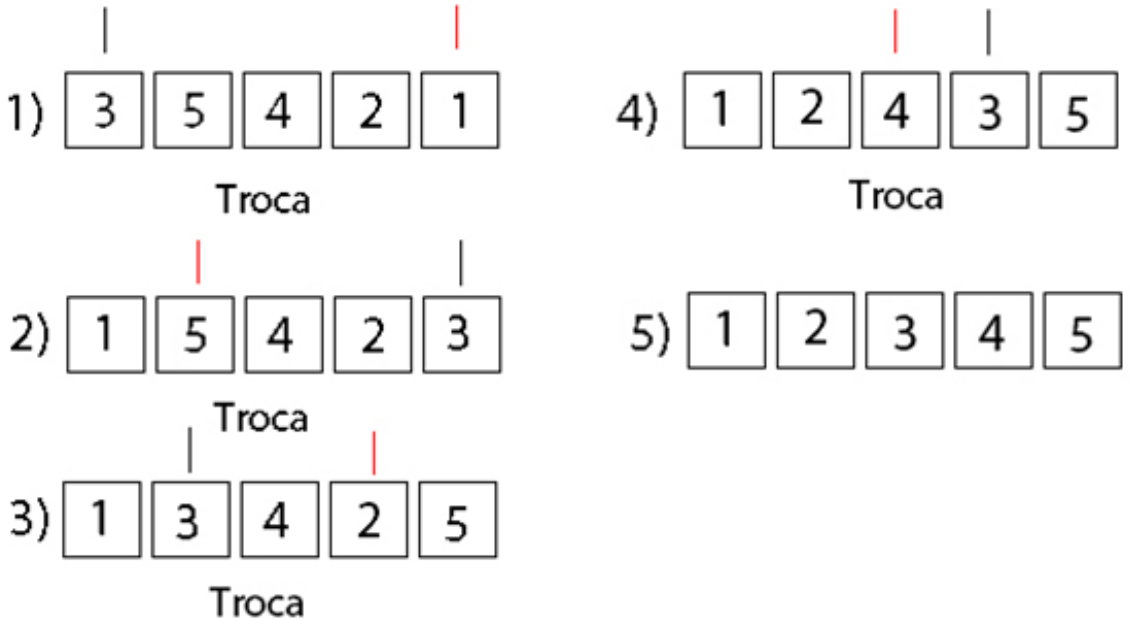


1) O número 3 foi escolhido como pivô, nesse passo é procurado à sua direita um número menor que ele para ser passado para a sua esquerda. O primeiro número menor encontrado foi o 1, então eles trocam de lugar.

2) Depois 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.

5. Quicksort

Exemplo:

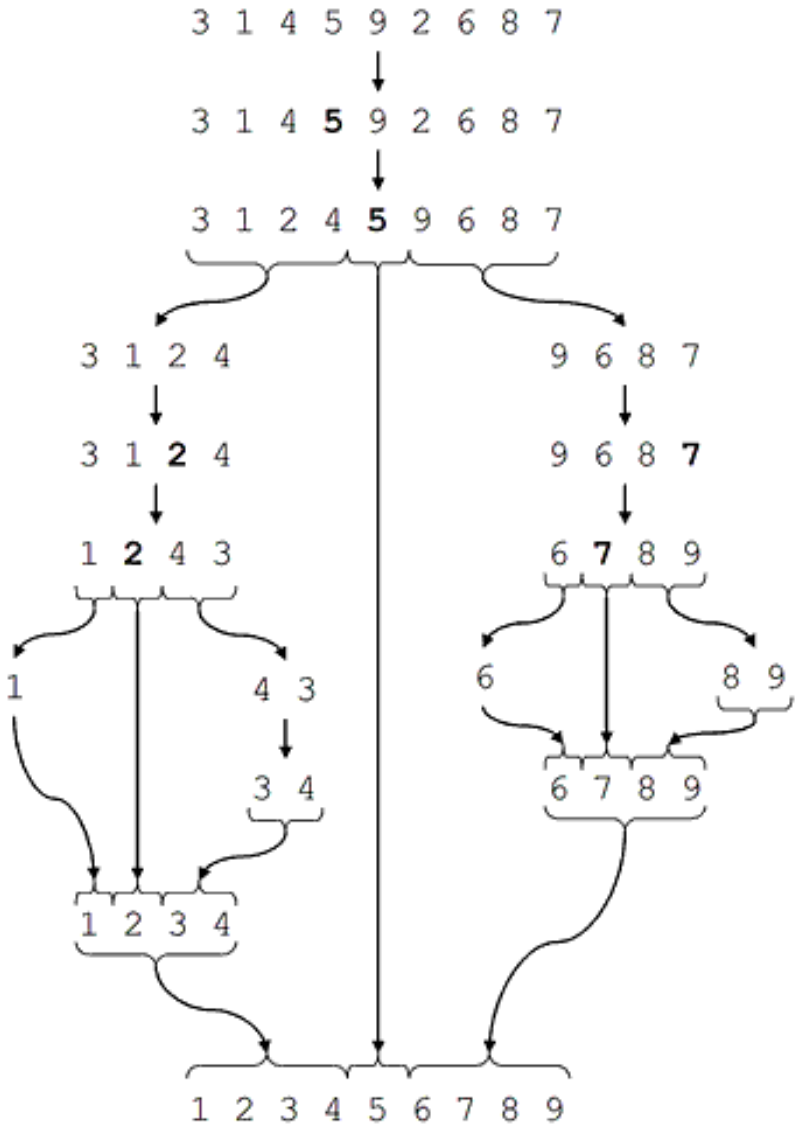


3) O mesmo processo do passo 1 acontece, o número 2 foi o menor número encontrado, eles trocam de lugar.

4) O mesmo processo do passo 2 acontece, o número 4 é o maior número encontrado, eles trocam de lugar.

5. Quicksort

Exemplo 2:



5. Quicksort

Código em C:

```c
#include<stdio.h>
#define TAM 10

void quick(int vet[], int esq, int dir){
    int pivo = esq,i,ch,j;
    for(i=esq+1;i<=dir;i++){
        j = i;
        if(vet[j] < vet[pivo]){
         ch = vet[j];
         while(j > pivo){
            vet[j] = vet[j-1];
            j--;
         }
         vet[j] = ch;
         pivo++;
        }
    }
    if(pivo-1 > esq){
        quick(vet,esq,pivo-1);
    }
    if(pivo+1 < dir){
        quick(vet,dir,pivo+1);
    }
 }
```

```c
int main(){
    int vet[TAM],i;

    for(i=0;i<TAM;i++)
        scanf("%d",&vet[i]);

    quick(vet,0,TAM-1);

    for(i=0;i<TAM;i++)
        printf("%d ",vet[i]);
    printf("\n");
    return 0;
}
```

7. REFERÊNCIAS

- Márcio Alexandre Marques. Algoritmos - Lógica para Desenvolvimento de Programação de Computadores. 1ª Ed. Editora Érica, 2010.

- Sandra Rita. TREINAMENTO EM LOGICA DE PROGRAMAÇAO, Digerati Books, 1 ed. 2009.

- SIMÃO, DANIEL HAIASHIDA; REIS, WELLINGTON JOSÉ DOS. LOGICA DE PROGRAMAÇAO. São Paulo : EDITORA VIENA, 2015. 176p.

- Souza, Marco Antonio Furlan de *et. all*, Algoritimos e Lógica de Programação. 2 ed. São Paulo : Nobel, 2011.