ALGORITMOS DE ORDENAÇÃO

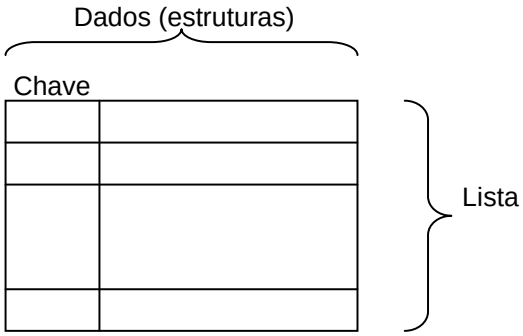
O problema da ordenação é fundamental na computação. Ordenar indica o processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. A atividade de colocar as coisas em ordem está presente na maioria das aplicações em que os objetos armazenados têm de ser pesquisados e recuperados.

O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. Como vimos nos métodos de pesquisa, a eficiência da busca sempre é melhor quando trabalhamos com conjuntos ordenados.

O aspecto predominante na escolha de um algoritmo de ordenação é o tempo gasto para ordenar os itens de uma lista. Existem muitos algoritmos de ordenação e a escolha do mais eficiente vai depender de vários fatores:

- número de itens a ser classificado;
- se os valores já estão agrupados em subconjuntos ordenados (o grau de ordenação já existente),
- se os registros deverão ou não ser removidos ou inseridos periodicamente, etc.

A ordem da ordenação acontece segundo uma chave (normalmente um campo dos dados contidos na lista). A ordenação pode ser crescente ou decrescente.



A ordenação está presente na maioria das aplicações em que os objetos armazenados tem de ser pesquisados e recuperados.

- Ordenação interna: São métodos que não necessitam de uma memória secundária para o processo, a ordenação é feita na memória principal do computador;

- Ordenação externa: Quando o arquivo a ser ordenado não cabe na memória principal e, por isso, tem de ser armazenado em disco.

Alguns métodos de ordenação:

- *Bubble Sort* (Ordenação por bolha)
- *Insert Sort* (Ordenação por Inserção)

- *Select Sort* (ordenação por Seleção)
- *Shell Sort* (Ordenação Shell)
- *Quick Sort* (Ordenação Rápida)
- Heap Sort (Ordenação por Pilha)
- *Merge Sort* (Ordenação por Intercalação)

1. Ordenação por Seleção – *Select Sort*

É considerado um dos algoritmos mais simples de ordenação, cujo princípio de funcionamento é o seguinte: selecione o maior (ou menor) item do conjunto e a seguir coloque-o em sua posição correta dentro da futura lista ordenada. Repita essas duas operações com os n-1 itens restantes, depois com os n-2 itens, até que reste apenas um elemento.

Durante a aplicação do método de seleção, a lista com m registros fica decomposta em duas sub-listas, uma contendo os itens já ordenados e a outra com os restantes ainda não ordenados. No início a sub-lista ordenada é vazia e a outra contém todos os demais itens. No final do processo a sub-lista ordenada apresentará (m-1) itens e a outra apenas 1.

As etapas (ou varreduras) como já descritas acima consistem em buscar o maior elemento da lista não ordenada e colocá-lo na lista ordenada.

Veja nos exemplos abaixo os resultados das etapas da ordenação de vetores (suponha que a 1ª posição do vetor seja 0):

- Seleção (Buscando o maior elemento)

Etapa	v[0]	v[1]	v[2]	v[3]	v[4]
Vetor original	5	9	1	4	3
1	{5	3	1	4}	{9}
2	{4	3	1}	{5	9}
3	{1	3}	{4	5	9}
4	{1}	{3	4	5	9}

- Seleção (Buscando o menor elemento)

Etapa	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
Vetor original	O	R	D	E	N	A
1	{A}	{R	D	E	N	O}
2	{A	D}	{R	E	N	O}
3	{A	D	E}	{R	N	O}
4	{A	D	E	N}	R	O}
5	{A	D	E	N	O}	{R}

Função de Seleção (Buscando o maior elemento):

```
int Selecao(int n, int *v){
    int aux, i, j, maior;
    for (i = 0; i < n-1; i++) {
        maior = 0;
        for(j = 1; j < n-i; j++)
          if (v[j] > v[maior])
             maior = j;
        if (maior != n-i-1){
        aux = v[n-i-1];
        v[n-i-1] = v[maior];
        v[maior] = aux;
    }
}
```

Outra opção para implementação:

```
int SelecaoDireta(int n, int *v) {
     int aux, i, j;
     for(i = 0; i < n-1; i++)
       for (j = i+1; j < n; j++)
         if (v[j] < v[i]) {
             aux = v[i];
             v[i] = v[j];
             v[j] = aux;
         }
}
```

2. Ordenação por Inserção – *Insert Sort*

Este método consiste em, dado um vetor para ordenação, percorrer elemento por elemento deslocando-o e inserindo-o na posição ordenada. A ideia é formar um bloco de valores ordenados e outro de desordenados, e ir passando os valores de um bloco a outro.

Durante o processo de ordenação por inserção a lista fica dividida em duas sub-listas, uma com os elementos já ordenados e a outra com elementos ainda por ordenar.

Uma ordenação por inserção inicia considerando os dois primeiros elementos (suponha que a 1ª posição do vetor seja 0): data[0] e data[1]. Se eles estão fora de ordem, um intercâmbio se realiza. Então, um terceiro elemento, data[2], é considerado e inserido em seu lugar apropriado. Se data[2] é menor do que data[0] e data[1], estes dois são mudados em uma posição; data[0] é colocado na posição 1, data[1] na posição 2 e data[2] na posição 0. Se data[2] é menor do que data[1] e não menor do que data[0], somente data[1] é movido para a posição 2 e seu lugar é tomado por data[2]. Se, finalmente, data[2] não é menor do que seus predecessores, ele permanece em sua posição. Cada elemento data[i] é inserido em seu local apropriado j de tal forma que $0 \leq j \leq i$ e todos os elementos maiores que data[i] são movidos em posição.

Veja nos exemplos abaixo os resultados das varreduras:

Exemplo 1:

Varredura	v[0]	v[1]	v[2]	v[3]	v[4]
Vetor original	9	8	7	6	5
1	{8	9}	{7	6	5}
2	{7	8	9}	{6	5}
3	{6	7	8	9}	{5}
4	{5	6	7	8	9}

Exemplo 2:

Varredura	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
Vetor original	O	R	D	E	N	A
1	{O	R}	{D	E	N	A}
2	{D	O	R}	{E	N	A}
3	{D	E	O	R}	{N	A}
4	{D	E	N	O	R}	{A}
5	{A	D	E	N	O	R}

Nos exemplos, a colocação do item no seu lugar apropriado na sequência de destino é realizada movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia. Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo:

(i) um item com chave menor que o item em consideração é encontrado;
(ii) o final da seqüência destino é atingido à esquerda.

Função de Inserção:

```
int Insercao(int n,int *v){
    int i, j, aux;
    for(i = 1; i < n; i++)     {
        aux = v[i];
        for(j = i-1; (j >= 0) && (aux < v[j]); j--)
            v[j + 1] = v[j];
        v[j + 1] = aux;
     }
}
```

3. Ordenação por Troca – *Bubble Sort*

Um método simples de ordenação por troca é a estratégia conhecida como bolha que consiste, em cada etapa “borbulhar” o maior elemento para o fim da lista. Inicialmente percorre-se a lista dada da esquerda para a direita, comparando pares de elementos consecutivos, trocando de lugar os que estão fora da ordem.

No exemplo abaixo, em cada troca, o maior elemento é deslocado uma posição para a direita.

Varredura	v[0]	v[1]	v[2]	v[3]	Troca
1	10	9	7	6	0 e 1
	9	10	7	6	1 e 2
	9	7	10	6	2 e 3
	9	7	6	10	Fim da Varredura 1

Após a primeira varredura o maior elemento encontra-se alocado em sua posição definitiva na lista ordenada. Podemos deixá-lo de lado e efetuar a segunda varredura na sub lista v[0],v[1],v[2]. Veja a continuação do exemplo:

Varredura	v[0]	v[1]	v[2]	v[3]	Troca
2	9	7	6	10	0 e 1
	7	9	6	10	1 e 2
	7	6	9	10	Fim da Varredura 2

Após a segunda varredura, o maior elemento da sub-lista v[0], v[1], v[2] encontra-se alocado em sua posição definitiva. A próxima sub-lista a ser ordenada é v[0], v[1]. Veja a continuação do exemplo:

Varredura	v[0]	v[1]	v[2]	v[3]	Troca
3	7	6	9	10	0 e 1
	6	7	9	10	Fim da Varredura 3

Função Bubble Sort:

```
int BubbleSort(int n, int *v){
  int i, j, aux, trocado = 1;
  for (i=0; i<n-1 && trocado; i++) {
    trocado = 0;
    for (j=0; j<n-i-1; j++) {
      if (v[j] > v[j+1]) {
        trocado = 1;
        aux = v[j];
        v[j] = v[j+1];
        v[j+1] = aux;
      }
    }
  }
}
```

4. Shell *Sort*

Este algoritmos foi proposto por Donald Shell em 1959. É uma extensão do algoritmo de ordenação por inserção. O algoritmo de ordenação por inserção (*Insert Sort*) só troca itens adjacentes para determinar o ponto de inserção. O método de Shell contorna este problema permitindo trocas de registros distantes.

Desta forma, o algoritmo *Shell Sort* é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção.

Função Shell Sort:

```
void shellSort (int n, int v[]) {
  int i , j , valor;
  int gap = 1;
  while(gap < n) // calcula o gap inicial
    gap = 3*gap+1;
  while ( gap > 1) {
    gap /= 3; // atualiza o valor do gap
    for(i = gap; i < n; i++) {
      valor = v[i];
      j = i - gap;
      // efetua comparações entre elementos com distância gap
      while (j >= 0 && valor < v[j]) {
        v[j+gap] = v[j];
        j -= gap;
      }
      v[j+gap] = valor;
    }
  }
}
```

Vantagens:

- Shellsort é uma ótima opção para arquivos de tamanho moderado.
- Sua implementação é simples e requer uma quantidade de código pequena.

Desvantagens:

- O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
- O método não é estável.

5. Ordenação por Intercalação - *Merge Sort*

O algoritmo *Merge Sort*, ou ordenação por intercalação ou mistura, é um exemplo de algoritmo de ordenação do tipo *dividir-para-conquistar*.

Sua ideia básica consiste em **dividir** (o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade) e **conquistar** (após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas).

Os três passos úteis dos algoritmos dividir-para-conquistar, que se aplicam ao *Merge Sort* são:

1. **Dividir:** Dividir os dados em subsequências pequenas;
2. **Conquistar:** Classificar as metades recursivamente aplicando o *Merge Sort*;

3. **Combinar:** Juntar as metades em um único conjunto já classificado.

Função *Merge Sort*:

```
void mergeSort(int inicio, int fim, int v[]) {
  int i, j, k, meio, vetaux[max];
  if (inicio == fim)
    return;
  // ordenação recursiva das duas metades
  meio = (inicio+fim) / 2;
  mergeSort (inicio,meio,v);
  mergeSort (meio+1,fim,v);
  // intercalação no vetor temporário t
  i = inicio;
  j = meio + 1;
  k = 0;
   while (i<meio+1 || j<fim+1) {
    // i passou do final da primeira metade, pegar v[j]
    if (i == meio + 1 ) {
      vetaux[k] = v[j];
      j++;
      k++;
    }
    else {
      // j passou do final da segunda metade, pegar v[i]
      if (j == fim+1) {
        vetaux[k] = v[i];
        i++;
        k++;
      }
      else {
        if (v[i] < v[j]) {
          vetaux[k] = v[i];
          i++;
          k++;
        }
        else {
          vetaux[k] = v[j];
          j++;
          k++;
        }
      }
    }
  }
  // copia vetor intercalado para o vetor original
  for (i = inicio; i <= fim; i++)
    v[i] = vetaux[i-inicio];
}
```

6. *Quick Sort*

O algoritmo *Quick Sort* foi proposto por Hoare em 1960 e publicado em 1962. É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações e, provavelmente é o mais utilizado.

O algoritmo de ordenação *QuickSort* adota a estratégia de **divisão e conquista**. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o *QuickSort* ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

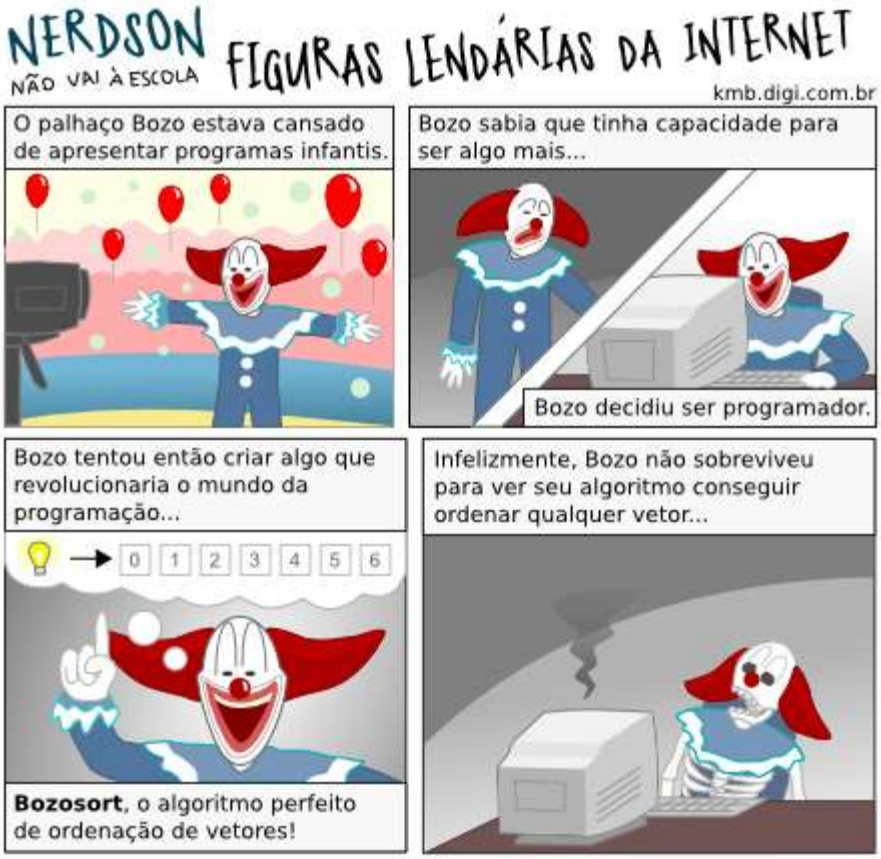
Resumindo, o *Quick Sort* divide o problema de ordenar um conjunto com *n* itens em dois problemas menores. Os problemas menores são ordenados independentemente e, as partições são combinadas para produzir a solução final.

Os passos utilizados pelo algoritmo 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.

Função *Quick Sort*

```
int quickSort (int inicio, int fim, int *v) {
  int i, j, pivo, aux;
  i = inicio;
  j = fim;
  pivo = v[(inicio + fim) / 2];
  while (i < j) {
    while (v[i] < pivo)
      i++;
    while (v[j] > pivo)
      j--;
    if (i <= j) {
      aux = v[i];
      v[i] = v[j];
      v[j]= aux;
      i++;
      j--;
    }
  }
  if (j > inicio)
    quickSort (inicio,j,v);
  if (i < fim)
    quickSort (i,fim,v);
}
```


NERDSON
NÃO VAI À ESCOLA
FIGURAS LENDÁRIAS DA INTERNET
kmb.digi.com.br
O palhaço Bozo estava cansado de apresentar programas infantis.
Bozo sabia que tinha capacidade para ser algo mais...
Bozo decidiu ser programador.
Bozo tentou então criar algo que revolucionaria o mundo da programação...
0 1 2 3 4 5 6
Bozosort, o algoritmo perfeito de ordenação de vetores!
Infelizmente, Bozo não sobreviveu para ver seu algoritmo conseguir ordenar qualquer vetor...