ORDENAÇÃO E PESQUISA

São muitas as aplicações dos agregados homogêneos (vetores) em computação.

E em muitas dessas aplicações, é comum que precisemos ordenar o vetor, segundo algum critério (em geral ordem ascendente ou descendente de valores).

Também é bastante comum a tarefa de localização de determinado valor entre os elementos do vetor.

Serão apresentados a seguir, alguns dos métodos dessas importantes tarefas em processamento de dados: **Pesquisa e ordenação de vetores.**

PESQUISA EM VETORES

A pesquisa consiste na verificação da existência de um determinado valor dentro de um vetor, e, em caso afirmativo, da posição da sua ocorrência.

Veremos dois métodos de pesquisa bastante difundidos:

→ **Pesquisa Seqüencial**

→ **Pesquisa Binária**

Pesquisa Sequencial (ou linear)

Método utilizado para encontrar um elemento particular num vetor não ordenado.

A técnica consiste em comparar, sequencialmente, cada valor do vetor com o valor procurado, até que este seja encontrado, ou seja atingido o final do vetor, sem sucesso.

```c
// Retorna a posição onde determinado valor foi
// encontrado ou -1 se a busca for malsucedida

int buscaSeq(int vet[], int n, int val)
{
   int i;
   for(i=0; i< n; i++)
   {
      if(vet[i] == val)
         return i;

   }
   return -1;

}
```

Pesquisa Binária:

Pré-requisito: vetor previamente ordenado.

Baseia-se no princípio de reduzir à metade, sucessivamente, o “universo de busca”. Desse princípio resulta sua eficiência.

Em detalhes:

Determinar o elemento que está no meio do vetor e compará-lo com o valor procurado (`val`). Se o elemento central for igual a `val`, a sua posição é retornada (pesquisa termina com sucesso)

→ Se o elemento médio for menor que `val`, a pesquisa continuará na metade superior (a inferior será descartada);
→ Já se o elemento médio for maior que `val`, continua-se a pesquisa na metade inferior do vetor;
→E assim sucessivamente...

A pesquisa se encerrará em dois casos: ou quando a chave for encontrada, ou quando não houver mais nenhum componente do vetor a ser pesquisado.

Obs: O procedimento acima descrito aplica-se a vetores classificados em ordem crescente. Para vetores em ordem decrescente, aplica-se o raciocínio análogo.

Exemplo – elemento procurado: I

A	**C**	**E**	**G**	**I**	**J**	**N**	**P**	**S**	**T**	**Z**
A	**C**	**E**	**G**	**I**	J	N	P	S	T	Z
A	C	E	**G**	**I**	J	N	P	S	T	Z
A	C	E	G	**I**	J	N	P	S	T	Z

```
int PesqBin (int v[], int val, int n)
{
     int esq = 0;         // limite inferior
     int dir = n-1;       // limite superior
     int meio;

     while (esq <= dir)
     {
          meio = (esq + dir)/2;
          if (val == v[meio])
               return meio;
          if (val < v[meio])
               dir = meio-1;      //abandonar met. superior
          else
               esq = meio+1;      //abandonar met. inferior
     }
     return -1;    // não encontrado
}
```

```c
int PesqBin(int v[], int val, int e, int d)
{
   int meio;
   if (e > d)
      return -1;          // não encontrado
   meio = (e + d)/2;
   if (v[meio] == val)  // elemento encontrado
      return meio;
   if (val < v[meio])   //abandonar met. superior
      return  PesqBin(v, val, e, meio-1);
   return PesqBin(v, val, meio+1, d);
}
```

ORDENAÇÃO DE VETORES

Ordenar (ou classificar) um vetor consiste em organizar seus elementos numa determinada ordem.

Por exemplo:

→ em ordem alfabética um conjunto de dados do tipo cadeia de caracteres

→ de forma crescente (ou decrescente), dados numéricos.

Classificação por Inserção:

Baseia-se na idéia de inserir um a um os elementos em subconjuntos já ordenados do vetor.

Inicialmente, podemos considerar ordenado o subconjunto formado apenas pelo primeiro elemento.

D B E C A

Inserimos, então o segundo elemento, com o que temos um novo subconjunto (de dois elementos) ordenado.

D	**B**	E	C	A
B	**D**	E	C	A

B	**D**	E	C	A

A seguir, o terceiro elemento é inserido…

B **<u>D</u>** **E** C A

B **D** **E** C A (já estava na sua posição relativa)

O processo continua para os demais elementos, até que todos estejam em sua posição correta.

B	**D**	**E**	**C**	A
B	**D**	**C**	**E**	A
B	**C**	**D**	**E**	A

B	**C**	**D**	**E**	A

B	C	D	E	A
B	C	D	A	E
B	C	A	D	E
B	A	C	D	E
A	B	C	D	E

A	B	C	D	E

Exercício: completar o módulo abaixo.

```
void insercao(int V[ ], int n)
{


}
```

Método da Bolha / BubbleSort

Princípio geral: comparar elementos adjacentes e, caso eles estejam fora de ordem trocá-los de posição.

E **C** B F A D

C **E** **B** F A D

C B **E** **F** A D

C B E **F** **A** D

C B E A **F** **D**

C B E A D **F**

Após essa varredura, o maior elemento (em ordenações ascendentes) estará na posição correta.

C B E A D **F**

Segunda varredura:

C **B** E A D **F**

B **C** **E** A D **F**

B C **E** **A** D **F**

B C A **E** **D** **F**

B C A D **E** **F**

Novas varreduras vão sendo feitas até o vetor estar inteiramente ordenado.

B C A D **E** **F**

Obs: as comparações não precisam ser feitas para os elementos que já estão em suas posições.

→ a cada varredura o percurso é menor.

Melhoria possível:

Cada varredura pode ser feita somente até a posição onde se deu a última troca na varredura anterior

(os elementos posteriores já estão em seu lugar correto)

Implementação:

Exercício: completar.

```
void bolha(int V[ ]: int n)
{
   for(i = ...
      for (j = ...


}
```

Método da seleção:

O elemento de valor mais baixo é identificado e permutado com o primeiro elemento.

D B E C **A**

A B E C D

Dos elementos restantes, o de valor mais baixo é identificado e permutado com o segundo elemento do vetor.

A **B** E C D

A **B** E C D (não houve troca: elemento já estava em sua posição)

O processo se repete sucessivamente, até que o vetor esteja ordenado.

A **B** E **C** D

A **B** **C** E **D**

A **B** **C** **D** E

→ mais um passo seria desnecessário...

```
void selecao(int V[], int n)
{
  int i,j, min;
  char aux;
  for(i=0; i<n-1; i++)
  {
    min:= i;
    for(j= i+1; i<n; i++)
       if (V[j] < V[min]
         min = j;
    aux = V[i];
    V[i] = V[min];
    V[min] = aux
  }
}
```

Exercício:

Fazer uma implementação recursiva para o método da seleção.

a) uma função recursiva deverá separadamente identificar o menor elemento do vetor;

b) a função seleção (recorrendo à função "menor") identifica o menor, troca-o com o que está na primeira posição e repete todo o processo para a "sublista ainda não ordenada, isto é, que vai de esq+1 até dir.
OBS: considerar a condição de parada.

```
void menor(int V[], int esq, int dir);
void selecao(int V[], int esq, int dir);
```

ALGORITMOS APERFEIÇOADOS

AVALIAÇÃO DOS ALGORITMOS DE ORDENAÇÃO

Alguns aspectos empregados para avaliação da eficácia de um algoritmo. (para n elementos)

É possível determinar uma expressão matemática que indique o número mínimo, médio e máximo de operações necessárias (as principais são comparações e permutações).
→ assim, tem-se uma indicação do tempo que será gasto.

Por exemplo, para o método da Seleção:

O número mínimo de permutações será 0 (melhor caso: vetor já ordenado)

O número máximo de permutações será n -1 (pior caso: uma troca em cada passo).

Assim, o número médio de permutações será [(n-1) + 0] / 2 = (n-1) / 2

Podemos dizer, assim, que o número de permutações (e, consequentemente, o tempo gasto com elas) é “de ordem n”, ou seja, proporcional a n, o que não é grave em termos de desempenho.

Entretanto, não poderemos dizer o mesmo acerca do número esperado de comparações…

Número de comparações:

A classificação é feita em **n-1** passos, sendo que a cada passo varia o número de elementos a serem comparados para a escolha do menor.

No primeiro passo, há n-1 comparações, no segundo, n- 2,
no terceiro, n-3, … , no último, haverá somente 1 comparação.

Assim, o número total de comparações será a seguinte soma:

$$(n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$

Portanto, o número médio de comparações (média entre o maior e o menor número) será $[(n-1) + 1] / 2 = n / 2$

Então:

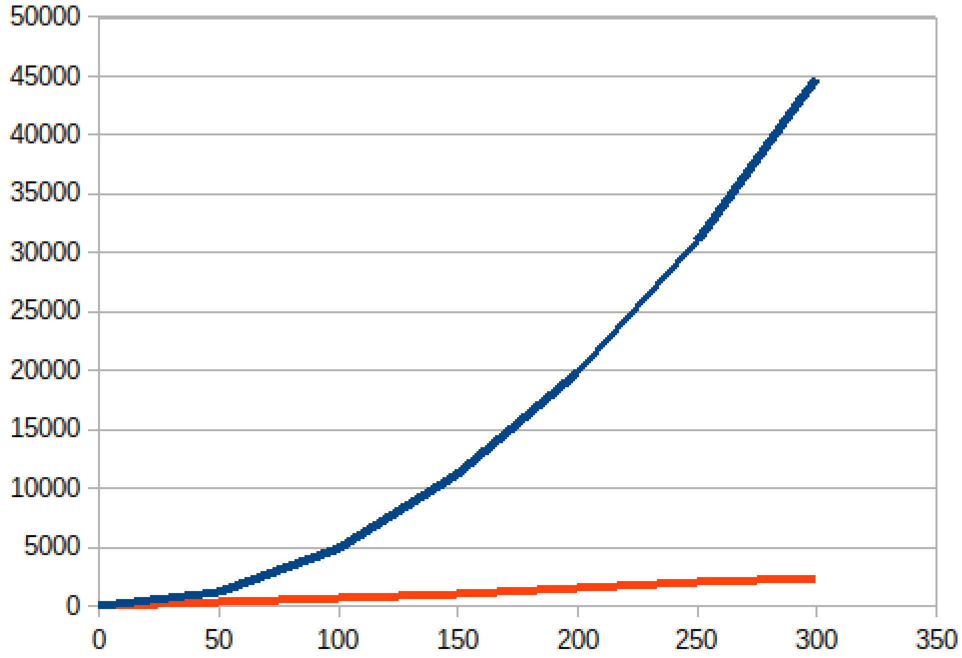
→ `n-1` passos
→ média de comparações: `n/2`

$$\frac{n}{2} * (n-1) = \frac{n(n-1)}{2} = \mathbf{\frac{n^2 - n}{2}}$$

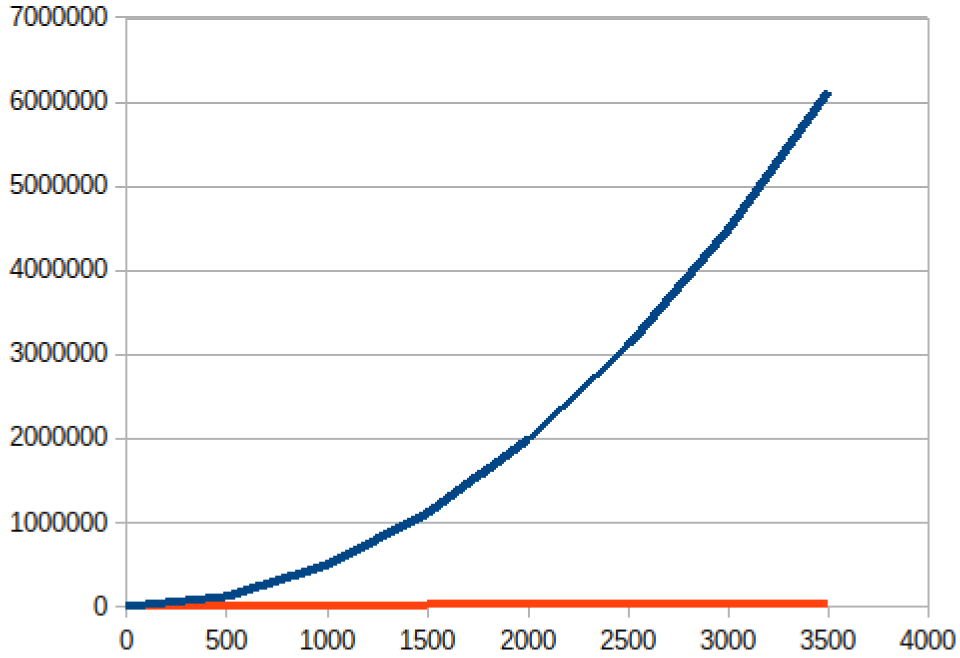
Além do método da Seleção, há vários outros métodos que têm o número de operações (e, portanto, o tempo gasto) dado por funções quadráticas (que envolvem n^2).

Esses métodos têm o seu desempenho bastante comprometido à medida que n (número de elementos da estrutura) aumenta.

O gráfico abaixo ilustra isso, através de duas curvas, uma para $y=(N^2-N)/2$, e outra, de crescimento bem mais suave, para $y = N.\log_2 N$. Esta segunda expressão é a que indica o número de operações a serem feitas em determinados métodos, ditos "aperfeiçoados", que veremos na sequência.



É importante notar ainda que o gráfico acima foi construído para valores relativamente baixos para n (até 300). Caso n cresça de forma substancial, a vantagem da função logarítmica se faz notar de forma muito mais significativa:



De fato, no segundo gráfico (mesmas funções do primeiro, porém para valores maiores de n), os valores correspondentes à função $y=n.\log_{2n}n$ se mostram desprezíveis se comparados aos relativos a $Y=(n^2-n)/2$.

Outros critéros para avaliar os algoritmos de ordenação

Um outro aspecto comumente citado como critério na avaliação de um algoritmo de ordenação é o mesmo ter ou não *comportamento natural*.

Isso ocorre se o seu desempenho melhora sensivelmente quando ele ordena estruturas já próximas do estado final, ou até mesmo já ordenadas.

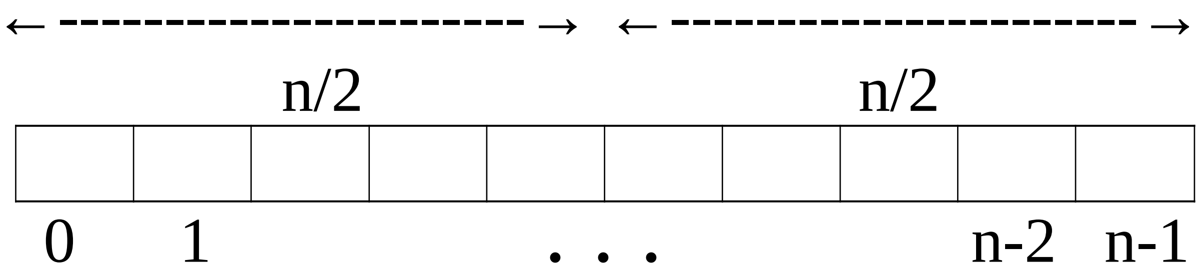
Dos métodos vistos, se compararmos o da **Inserção** e o da **Seleção**, sob esse aspecto, veremos que o primeiro apresenta comportamento natural, enquanto o segundo não: o número de comparações necessárias não muda, mesmo que a estrutura já esteja ordenada.

Podemos dizer, entretanto, que este critério é apenas complementar, uma vez que dificilmente temos situações onde as estruturas já estão (semi-)ordenadas. Devemos, então, privilegiar algoritmos que se comportem bem para os casos médios.

Vejamos então alguns desses algoritmos, cujo tempo de execução não é indicado por funções quadráticas: Quicksort, Heapsort,e ShelSort.

Método de Partição e Troca: Quicksort

Este, que é um dos melhores métodos de ordenação. Baseia-se na interessante idéia de particionar sucessivamente a estrutura, visando ordenar porções menores, separadamente. O ganho de desempenho decorre do fato de que é mais rápido ordenar dois vetores de tamanho N/2 do que um de tamanho N. Vejamos...



Se para uma estrutura de tamanho n, o tempo gasto é proporcinal a n^2,o tempo gasto para as duas metades será proporcionala:

$$(n/2)^2 + (n/2)^2 = 2 \, . \, (n/2)^2 = 2 \, . \, (n^2/4) = n^2/2$$

Ou seja, se dividimos a tarefa em duas partes, o tempo gasto se reduz à metade também. Assim, realizando sucessivas divisões, obteremos melhorias significativas no desempenho, com tempos de processamento proporcionais a $n.\log_2 n$.

O Quicksort exige uma primeira etapa:

→ selecionamos um valor x como referência (o elemento da posição média, por exemplo);
→ dividimos a estrutura em duas partes (não necessariamente de mesmo tamanho): a primeira com os elementos menores ou iguais a x, e a segunda com os maiores que x.

Desta forma, se ordenarmos os dois segmentos separadamente, a estrutura toda estará também ordenada.

O processo descrito será repetido em cada uma das duas partes, e assim sucessivamente (e recursivamente) até que a estrutura esteja ordenada.

Exemplo:

D	C	H	F	**E**	A	I	B	G

O valor **E** será o pivô. Percorremos

→ da esquerda para a direita até encontrarmos elementos maiores ou iguais a x;

→ da direita para a esquerda até encontrar elementos menores ou iguais a x.

Estes elementos serão permutados

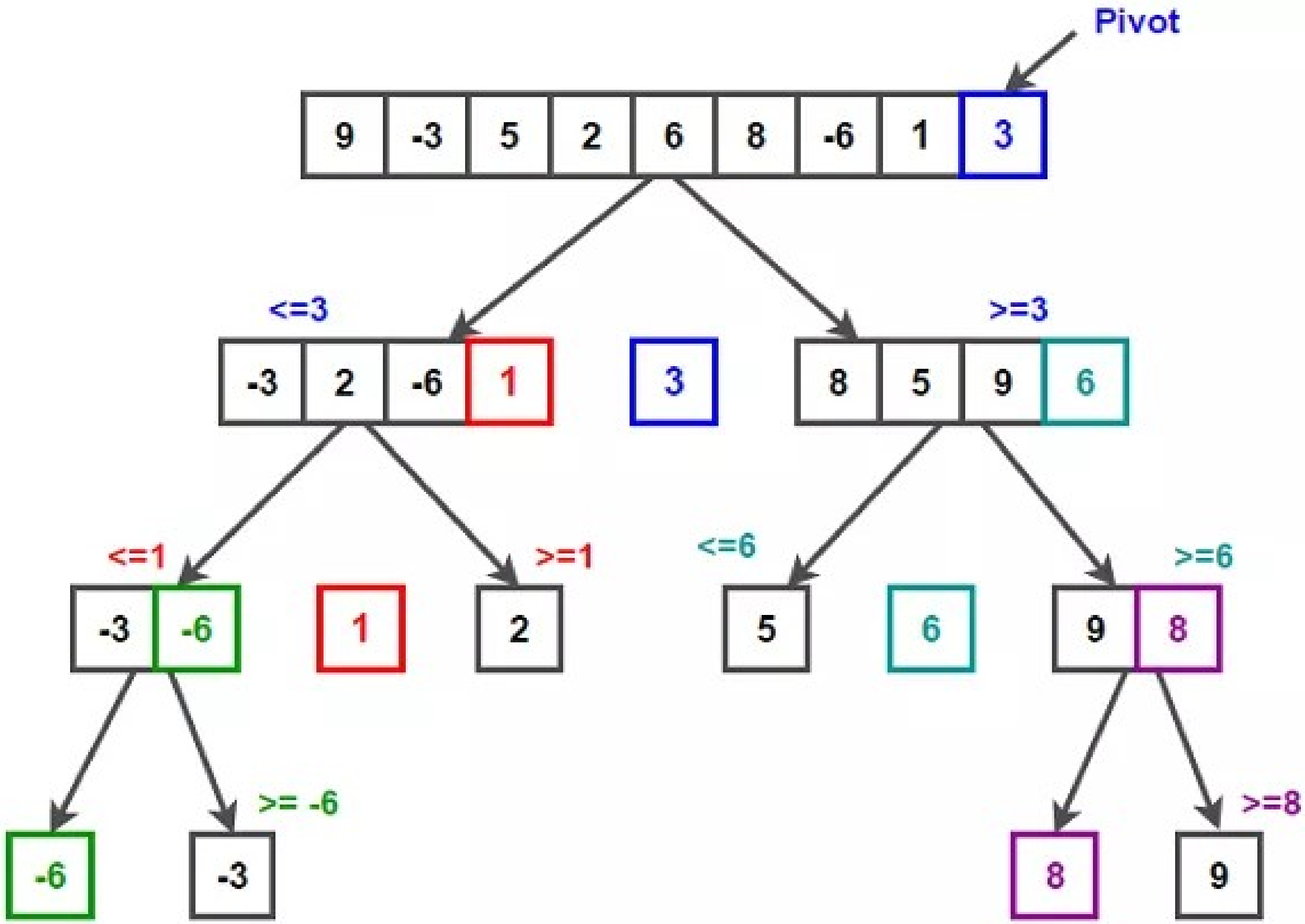
D	C	**H**	F	**E**	A	I	**B**	G
D	C	B	**F**	**E**	**A**	I	H	G
D	C	B	A	**E**	F	I	H	G

D C B A **E** F I H G

---------------------------- ------------------------------

Agora podemos ordenar separadamente cada partição e, assim, a estrutura toda estará ordenada.

A cada partição será aplicado o mesmo procedimento, recursivamente, até que isto seja desnecessário (partição de 1 elemento).


Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
>=1
2
<=6
5
6
>=6
9 8
-6
>= -6
-3
8
>=8
9

Implementação:

```
void quicksort (int v[], int esq, int dir)
{
   if (p < r)
   {
      int j = separa (v, esq, dir);
      quicksort (v, esq, j-1);
      quicksort (v, j+1, dir);
   }
}
```

Exercício: pesquisar uma função que faça a separação