# BC1424
# Algoritmos e Estruturas de Dados I

## Aula 06:
## Vetores e Algoritmos de busca

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

1Q-2015

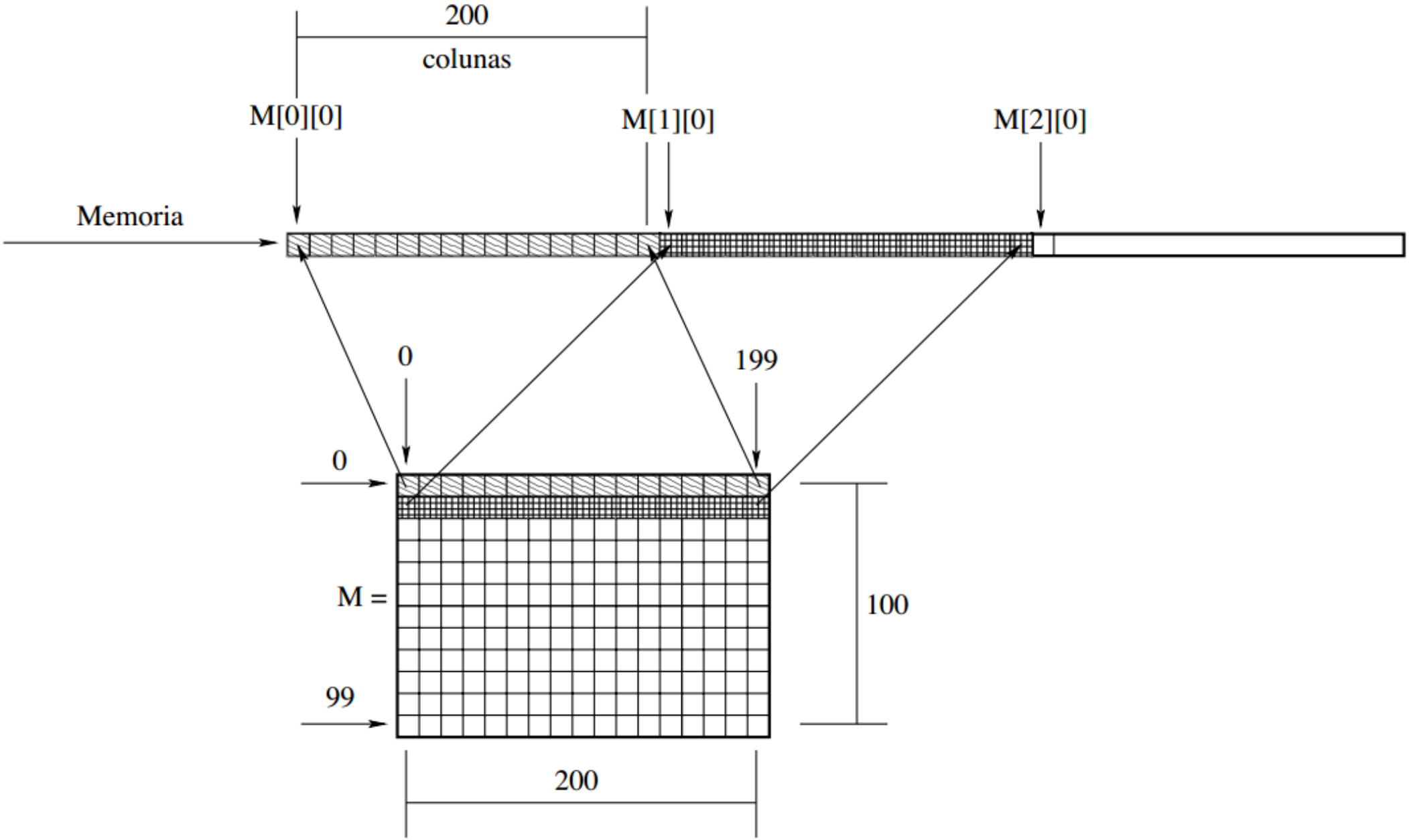
# Alocação estática de memória

```
 5
 6      int V[100];
 7
 8
 9      V[0] = 10;
10      V[1] = 20;
11      V[2] = 30;
```

```
int M[100][200];
```

Declara uma matriz M
de 100 linhas
com 200 colunas
(20mil inteiros)

Estrutura da matriz na memória do computador



# Exercício da aula anterior

Escreva um programa que leia um número inteiro positivo **n** seguido de **n** números inteiros e imprima esses **n** números em ordem invertida.

Por exemplo, ao receber

5 222 333 444 555 6666

o seu programa deve imprimir

6666 555 444 333 222

```
6
5 222 333 444 555 6666
6666 555 444 333 222 5
```

**Seu programa não deve impor limitações sobre o valor de n.**
**Não use colchetes.**

Os ponteiros facilitam a alocação dinâmica de memória

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

int main ( ) {
    int n, i;
    scanf("%d", &n);

    int *p = (int *) malloc(n*sizeof(int));

    for(i=0; i<n; i++)
        scanf("%d", p+i);

    for(i=n-1; i>=0; i--)
        printf("%d ", *(p+i));

    free(p);
}
```

```
6
5 222 333 444 555 6666
6666 555 444 333 222 5
```

# Exercício

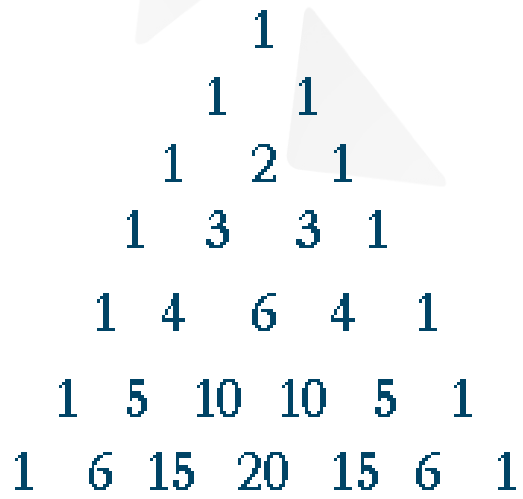
Crie um programa que imprima as **n** primeiras linhas do triângulo de Pascal.

→ Não use colchetes

→ Use a função **malloc**

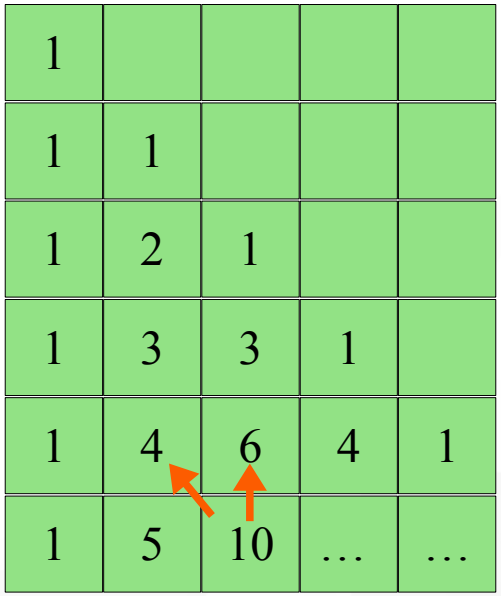
→ Use a função **free**

**Seu programa não deve impor limitações sobre o valor de n.**
**Não use colchetes.**




```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

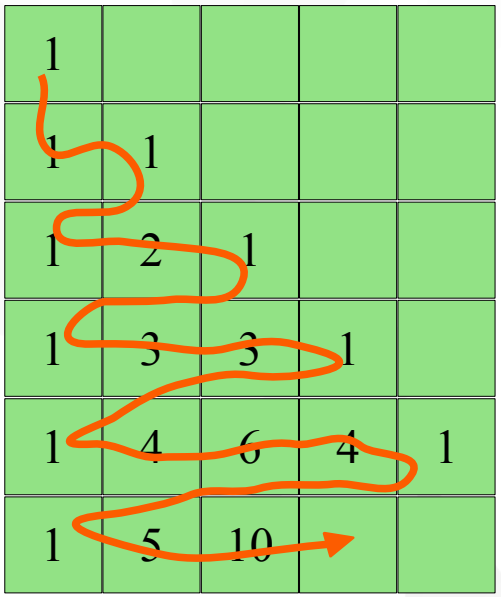
```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

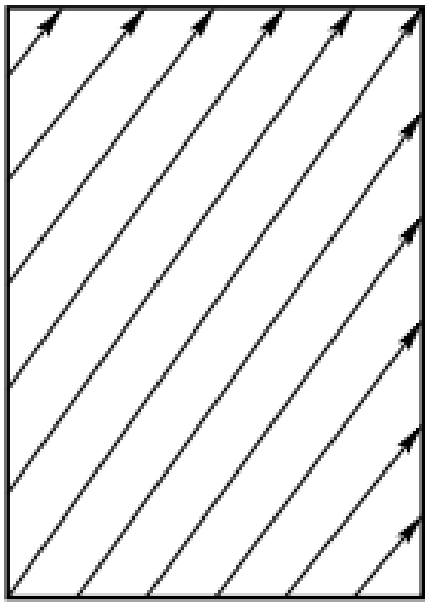


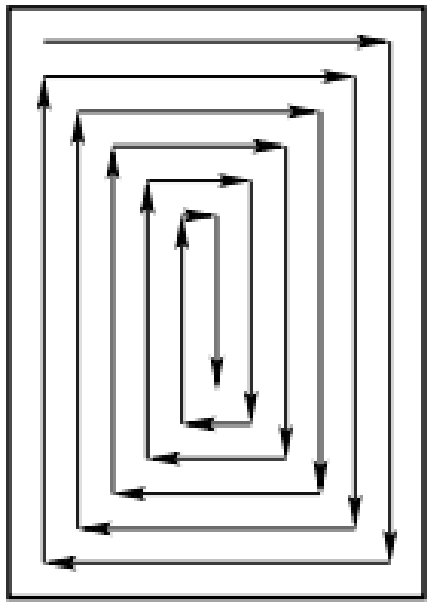
Os ponteiros facilitam a alocação dinâmica de memória

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

int main ( ) {
    int n, i, j;
    scanf("%d", &n);

    int *p = (int *) malloc(n*n*sizeof(int));

    for(i=0; i<n; i++) {
        for(j=0; j<=i; j++) {
            if (i==0 || i==j)
                *(p+i*n+j) = 1;
            else
                *(p+i*n+j) = *(p+(i-1)*n+j-1) + *(p+(i-1)*n+j);

            printf("%d ", *(p+i*n+j));
        }
        printf("\n");
    }

    free(p);
}
```

# Outros tipos de percorrer uma matriz.



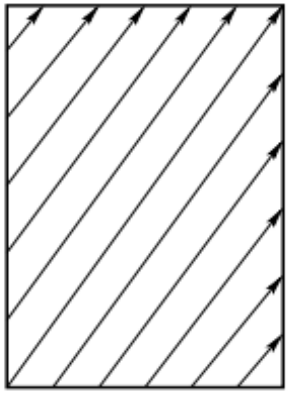



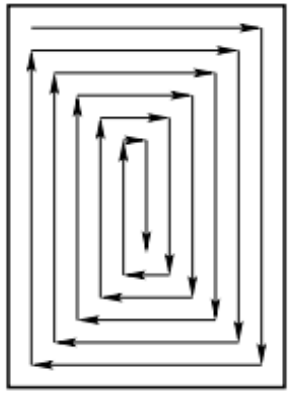


**Desafio 01 – opcional – envio até 11/03 (23h50-Tidia)**

Para quem preferir incrementar sua nota final:

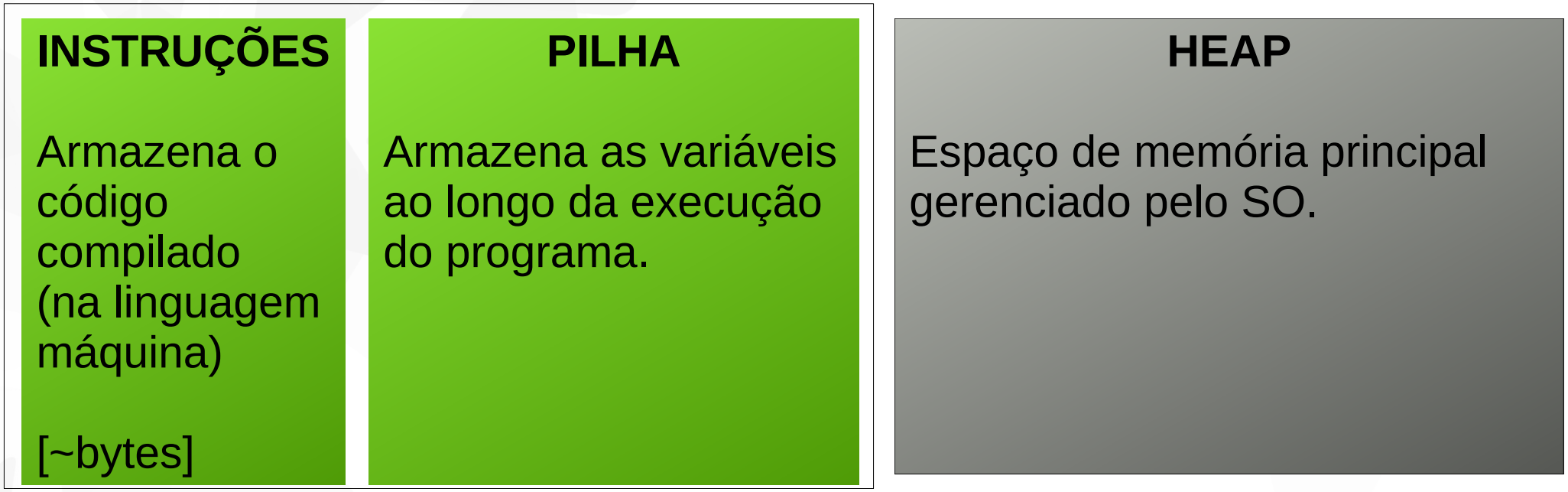
- Crie 2 funções que permitam percorrer uma matriz bidimensional seguindo os seguintes formatos.
- Seu programa não deve impor limitações sobre o número de linhas, nem colunas. Não use colchetes.
- Apresentação livre de exemplos (quanto mais completo melhor).
- Envie apenas arquivo: RA_percorrerMatriz.c





03 desafios = +1,0 na média final da disciplina

Processo na memória




Alocação estática

Alocação dinâmica

```
int x;
double M[10][20];
```

```
double M = malloc(...);
```

Aqui é necessário desalocar a memória free()

# Busca de um elemento

# Exemplo: Busca de um elemento

- Considere o problema de acessar os **registros** de um arquivo (cada registro tem chave única).

- **O problema:**

  Dada uma chave qualquer, localize o registro que contenha esta chave

  → Considere o algoritmo de busca sequencial.

# Exemplo: Busca de um elemento

```c
int main()
{
    int A[10] = {6,7,8,9,0,1,2,3,4,5};
    int chave, n;
    n = sizeof(A)/sizeof(A[0]);

    printf("\nIdentificar posicao da chave: ");
    scanf("%d", &chave);
    printf("\nA chave esta na posicao: %d", buscaChave(chave, A, n));
}
```

# Exemplo: Busca de um elemento

```
int buscaChave(int chave , int A[], int n) {
    int i;

    for(i=0; i<n; i++) {
        if (chave == A[i])
            return i;
    }
    return -1;
}
```

# Exemplo: Busca de um elemento

- Seja ***f*** uma função de complexidade tal que ***f(n)*** é o número de registros consultados.
  - Melhor caso: $f(n) = 1$ — Quando o elemento procurado é o primeiro consultado
  - Pior caso: $f(n) = n$ — Quando o elemento procurado é o último consultado
  - Caso médio: $f(n) = \frac{n+1}{2}$

```c
int buscaChave(int chave , int A[], int n) {
    int i;

    for(i=0; i<n; i++) {
        if (chave == A[i])
            return i;
    }
    return -1;
}
```

```c
int buscaChave(int chave , int A[], int n) {
    int i;

    for(i=0; i<n; i++) {
        if (chave == A[i])
            return i;
    }
    return -1;
}
```

```c
int buscaChave2(int chave , int A[], int n) {
    int i;
    i = n-1;

    while(i>=0 && A[i]!=chave) {
        i-=1;
    }
    return i;
}
```

Versão Elegante!

(*) Fonte: P. Feofiloff. Algoritmos em Linguagem C. 1ª Edição, Editora Campos, 2008.

# Busca de um elemento (Versão recursiva)

```
int buscaChaveRec(int chave , int v[], int n) {
    if (n==0)
        return -1;
    if (chave == v[n-1])
        return n-1;
    return buscaChaveRec(chave, v, n-1);
}
```

```
n = sizeof(A)/sizeof(A[0]);
buscaChaveRec(chave, A, n)
```

Custo computaciaonl
O(n)

# Pesquisa (Busca) Binária

# Algoritmo de Pesquisa Binária

*Binary search algorithm*

*Binary chop*

Parte do presuposto de que o **vetor está ordenado** e realiza sucessivas divisões do espaço de busca:

- O elemento procurado (chave) é comparado com o elemento do meio do vetor:
  - Se são iguais, a busca termina com sucesso.
  - Caso contrário:
    - Se o elemento do meio vier antes da chave, então a busca continua na metade posterior do vetor,
    - Caso contrário, a busca continua na metade anterior do vetor.

# Algoritmo de Pesquisa Binária

```c
int PesquisaBinaria (int *vetor, int chave, int N)
{
     int inf = 0;   //Limite inferior
     int sup = N-1; //Limite superior
     int meio;

     while (inf <= sup)
     {
          meio = inf + (sup-inf)/2;
          if (chave == vetor[meio])
               return meio;
          else if (chave < vetor[meio])
               sup = meio-1;
          else
               inf = meio+1;
     }
     return -1;   // não encontrado
}
```

# Algoritmo de Pesquisa Binária

```
int main()
{
	int A[10] = {6,7,8,9,0,1,2,3,4,5};
	int chave, N;
	N = sizeof(A)/sizeof(A[0]);

	printf("\nProcurar chave: ");
	scanf("%d", &chave);

	if (PesquisaBinaria(A, chave, N)!=-1)
		printf("\nElemento identificado\n");
	else
		printf("\nElemento nao identificado\n");
}
```

*Onde esta o erro?*

# Algoritmo de Pesquisa Binária

```
int main()
{
	int A[10] = {0,1,2,3,4,5,6,7,8,9};
	int chave, N;
	N = sizeof(A)/sizeof(A[0]);

	printf("\nProcurar chave: ");
	scanf("%d", &chave);

	if (PesquisaBinaria(A, chave, N)!=-1)
		printf("\nElemento identificado\n");
	else
		printf("\nElemento nao identificado\n");
}
```

*O algoritmo parte do presuposto de termos o vetor ordenado na forma ascendente.*

***Existe outro erro?***

# Algoritmo de Pesquisa Binária

```c
int PesquisaBinaria (int *vetor, int chave, int N)
{
     int inf = 0;   //Limite inferior
     int sup = N-1; //Limite superior
     int meio;

     while (inf <= sup)
     {
          meio = inf + (sup-inf)/2;
          if (chave == vetor[meio])
               return meio;
          else if (chave < vetor[meio])
               sup = meio-1;
          else
               inf = meio+1;
     }
     return -1;   // não encontrado
}
```

***Melhor caso:***

***Pior caso:***

# Algoritmo de Pesquisa Binária

```
int PesquisaBinaria (int *vetor, int chave, int N)
{
     int inf = 0;   //Limite inferior
     int sup = N-1; //Limite superior
     int meio;

     while (inf <= sup)
     {
          meio = inf + (sup-inf)/2;
          if (chave == vetor[meio])
               return meio;
          else if (chave < vetor[meio])
               sup = meio-1;
          else
               inf = meio+1;
     }
     return -1;   // não encontrado
}
```

***Melhor caso:  1***

***Pior caso: log(n)***

# Algoritmo de Pesquisa Binária (versão 2)

```
int PesquisaBinaria2 (int *vetor, int chave, int inf, int sup)
{
        int meio;
        if (sup>inf)   // não encontrado
                return -1;

        meio = (inf+sup)/2;
        if (chave == vetor[meio])
                return meio;
        if (chave < vetor[meio])
                return PesquisaBinaria2(vetor, chave, inf, meio-1);
        else
                return PesquisaBinaria2(vetor, chave, meio+1, sup);
}
```

*Onde está o erro?*

# Algoritmo de Pesquisa Binária (versão 2)

```
int PesquisaBinaria2 (int *vetor, int chave, int inf, int sup)
{
        int meio;
        if (inf>sup)    // não encontrado
                return -1;

        meio = (inf+sup)/2;
        if (chave == vetor[meio])
                return meio;
        if (chave < vetor[meio])
                return PesquisaBinaria2(vetor, chave, inf, meio-1);
        else
                return PesquisaBinaria2(vetor, chave, meio+1, sup);
}
```

*Erro nos índices (do slide anterior)*

# Algoritmo de Pesquisa Binária (versão 2)

```
int main()
{
	int A[10] = {0,1,2,3,4,5,6,7,8,9};
	int chave, N;
	N = sizeof(A)/sizeof(A[0]);

	printf("\nProcurar chave: ");
	scanf("%d", &chave);

	if (PesquisaBinaria2(A, chave, 0, N-1)!=-1)
		printf("\nElemento identificado\n");
	else
		printf("\nElemento nao identificado\n");
}
```

# Recursividade

**Uma função recursiva é aquela que se chama a si mesma (obrigatoriamente)?**

```
int PesquisaBinaria2 (int *vetor, int chave, int inf, int sup)
{
        int meio;
        if (inf>sup)    // não encontrado
                return -1;

        meio = (inf+sup)/2;
        if (chave == vetor[meio])
                return meio;
        if (chave < vetor[meio])
                return PesquisaBinaria2(vetor, chave, inf, meio-1);
        else
                return PesquisaBinaria2(vetor, chave, meio+1, sup);
}
```

# Recursividade

***Uma função recursiva não necessariamente é aquela que se chama a si mesma***

```
int F1(parametros)
{

   .
   .
   .

  F2(parametros)
   .
   .
}
```

```
int F2(parametros)
{

   .
   .
   .

  F1(parametros)
   .
   .
}
```