

Recursão e Busca Binária

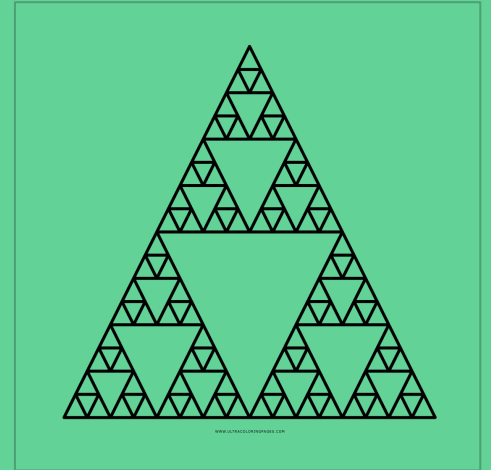
Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Gregghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva



Roteiro

- Recursão
 - Busca Binária
-

Recursão



Recursão

- O conceito de recursão é fundamental na matemática e na ciência da computação.
 - Também conhecido como recursividade.
- **Definição:** Em programação, um subprograma recursivo é um subprograma que chama a si mesmo.
- **Definição:** Em matemática, uma função recursiva é uma função definida em termos de si mesma por meio de uma relação de recorrência.

Relação de recorrência

Uma relação de recorrência é uma equação que define uma função recursivamente, usualmente por meio de um ou mais casos bases e um ou mais casos gerais. Por exemplo, a relação de recorrência que define o fatorial de um número inteiro positivo **N** é dada a seguir:

$$N! = \begin{cases} 1, & \text{se } N = 0 \\ N \times (N-1)!, & \text{se } N > 0 \end{cases}$$

Recursão e Funções

Uma das maneiras de traduzirmos a relação de recorrência:

$$N! = \begin{cases} 1, & \text{se } N = 0 \\ N \times (N-1)!, & \text{se } N > 0 \end{cases}$$

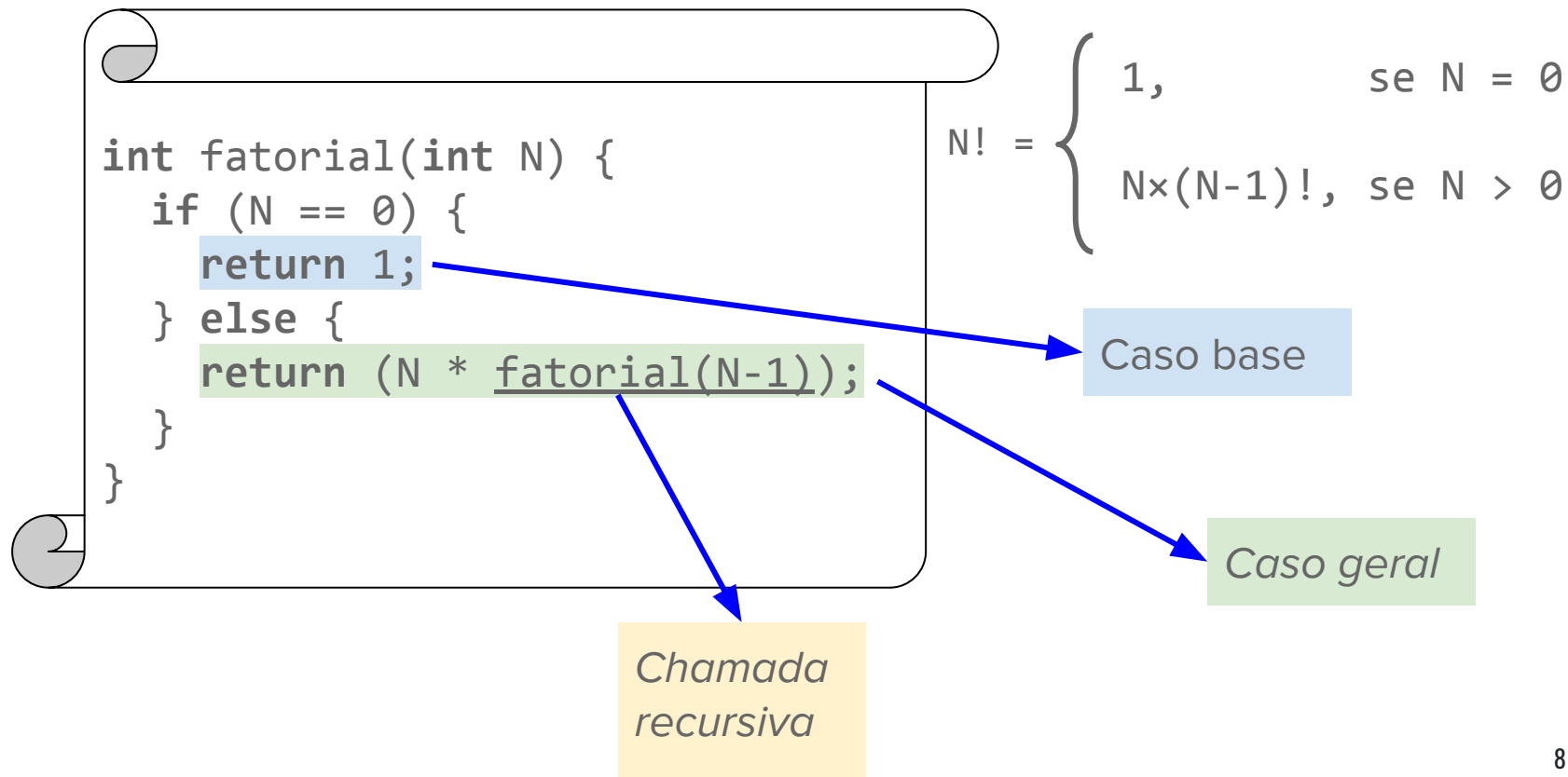
em um programa seria por meio da implementação de uma função que chama a si mesma, como no exemplo a seguir.

Exemplo de função recursiva

```
int fatorial(int N) {  
    if (N == 0) {  
        return 1;  
    } else {  
        return (N * fatorial(N-1));  
    }  
}
```



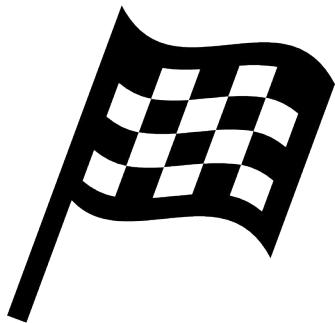
Exemplo de função recursiva



Condição de parada

Uma função recursiva (em matemática) não pode ser definida em termos de si mesma sempre, ou a definição seria circular.

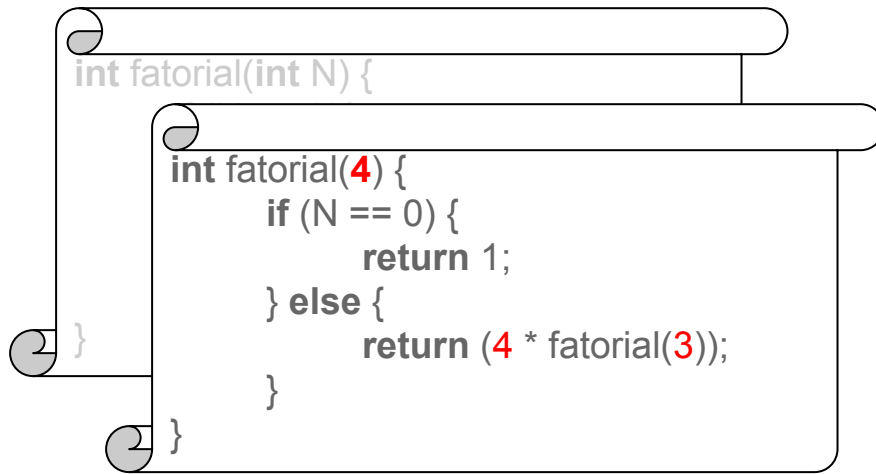
Logo, em programação, sempre deverá existir uma condição de término (caso base) em que o subprograma não chame a si mesmo.



Como exemplo, vamos analisar a pilha de recursão do nosso subprograma fatorial, considerando inicialmente N igual a 4.

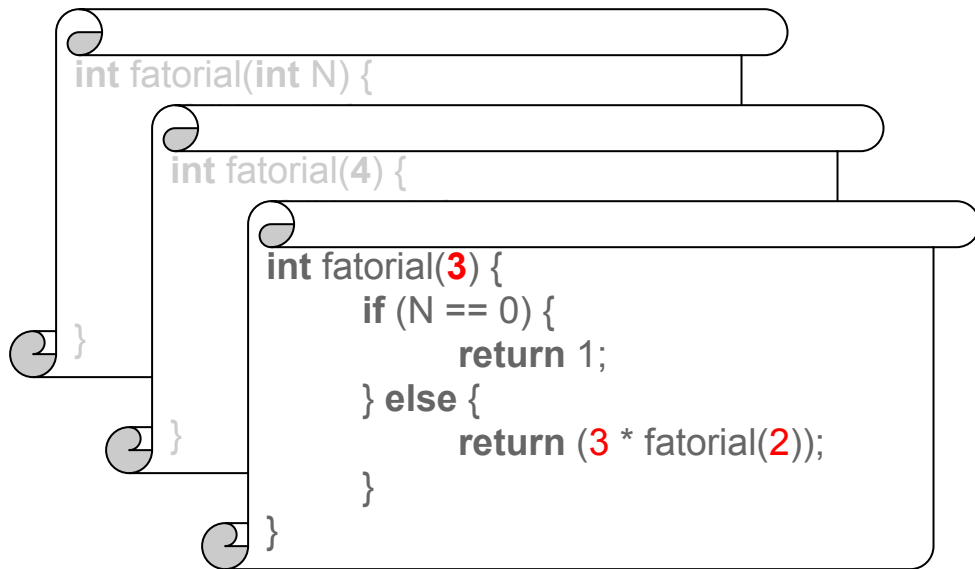
```
int fatorial(int N) {  
    if (N == 0) {  
        return 1;  
    } else {  
        return (N * fatorial(N-1));  
    }  
}
```

Considere que queremos
calcular **4!**



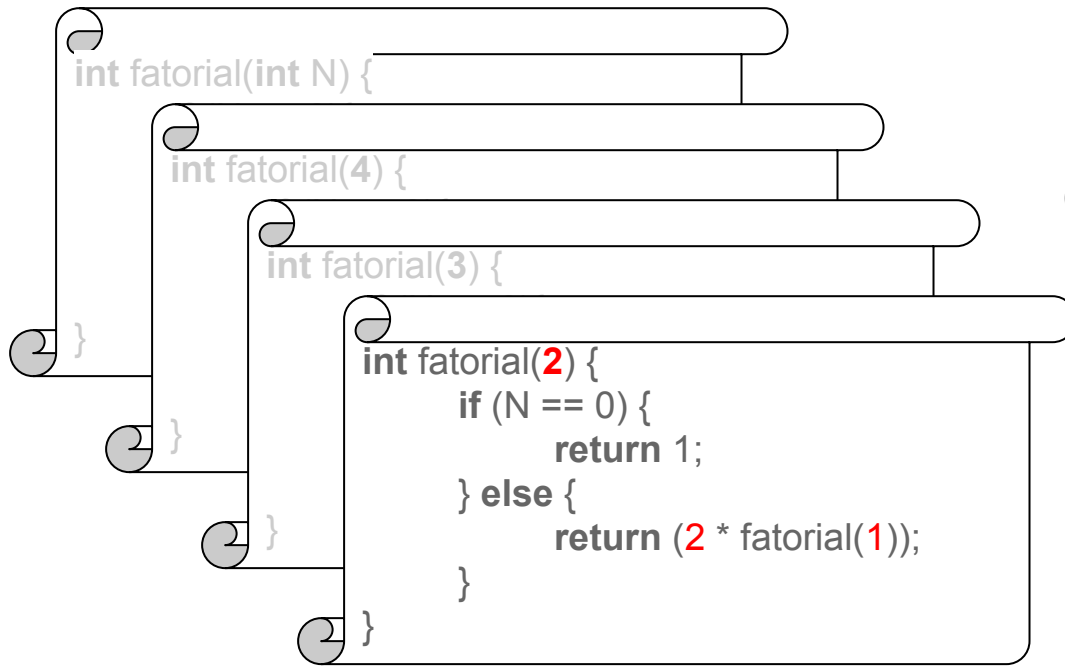
N = 4

Chamada recursiva: fatorial(3)



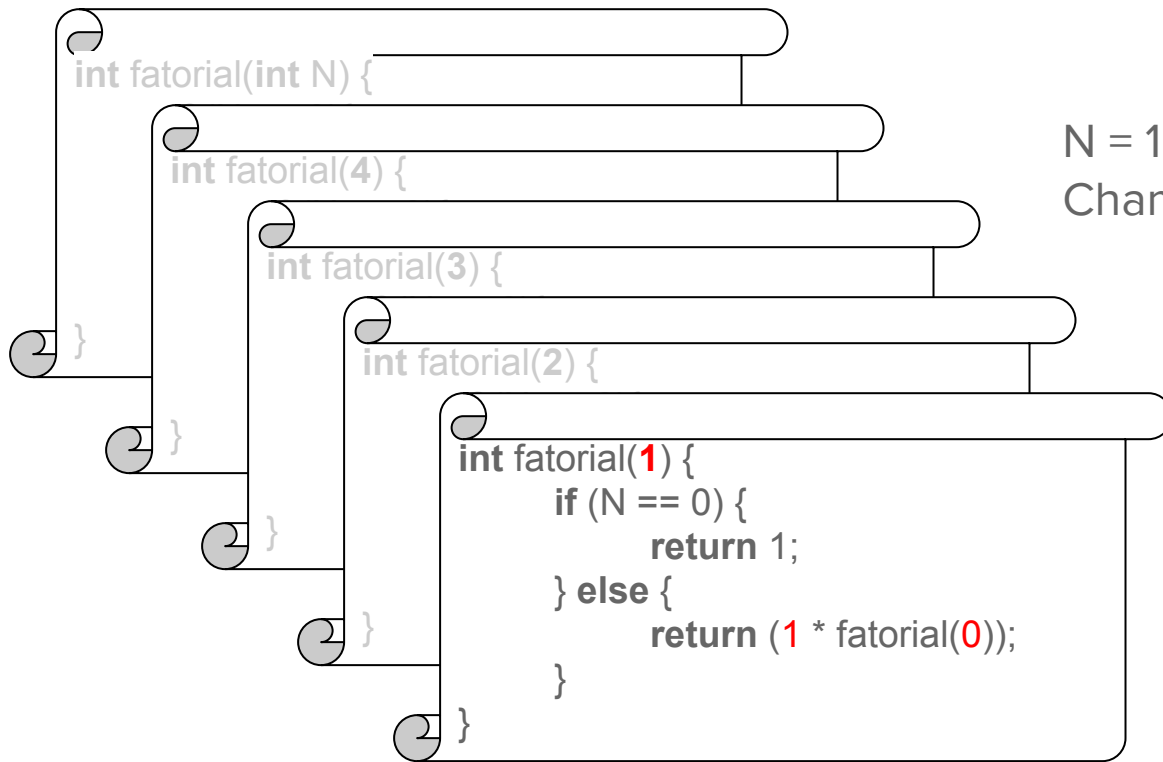
$N = 3$

Chamada recursiva: `fatorial(2)`



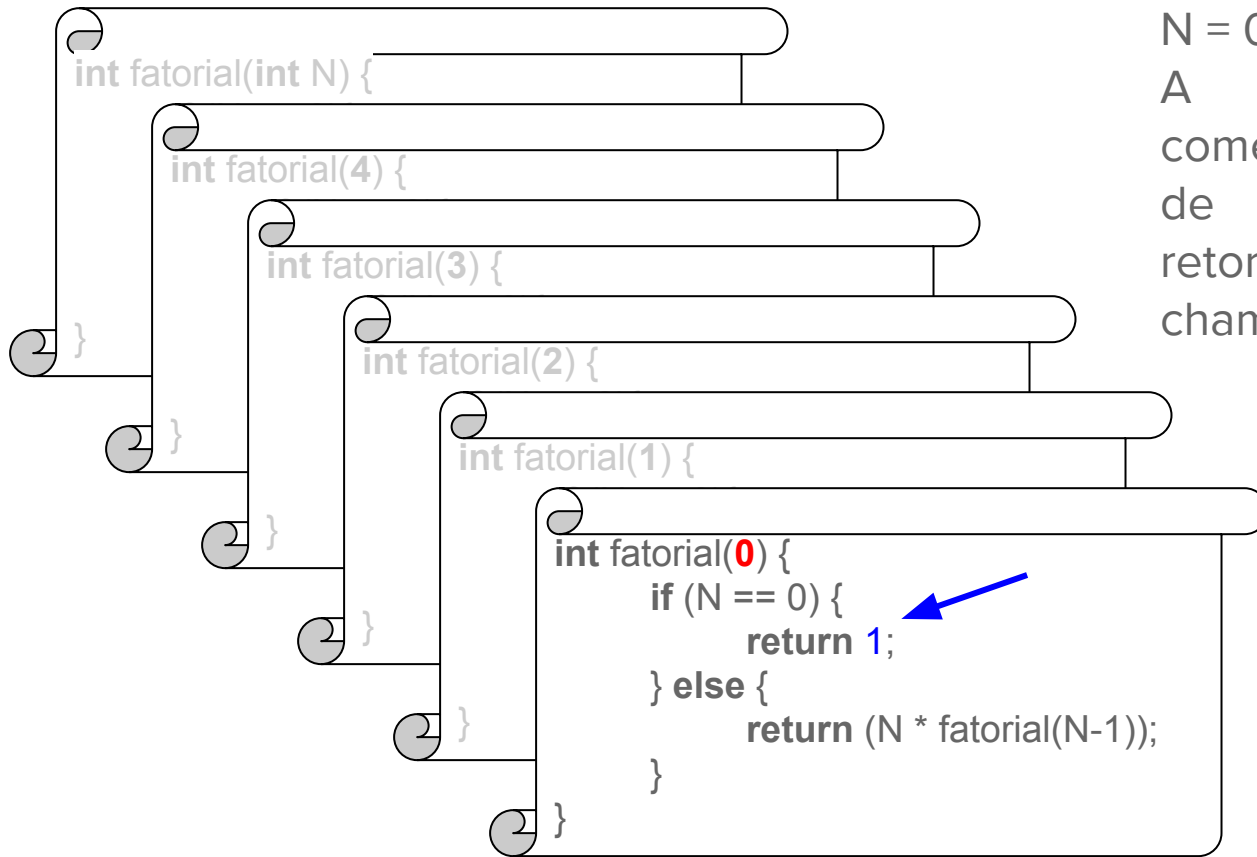
`N = 2`

Chamada recursiva: `fatorial(1)`



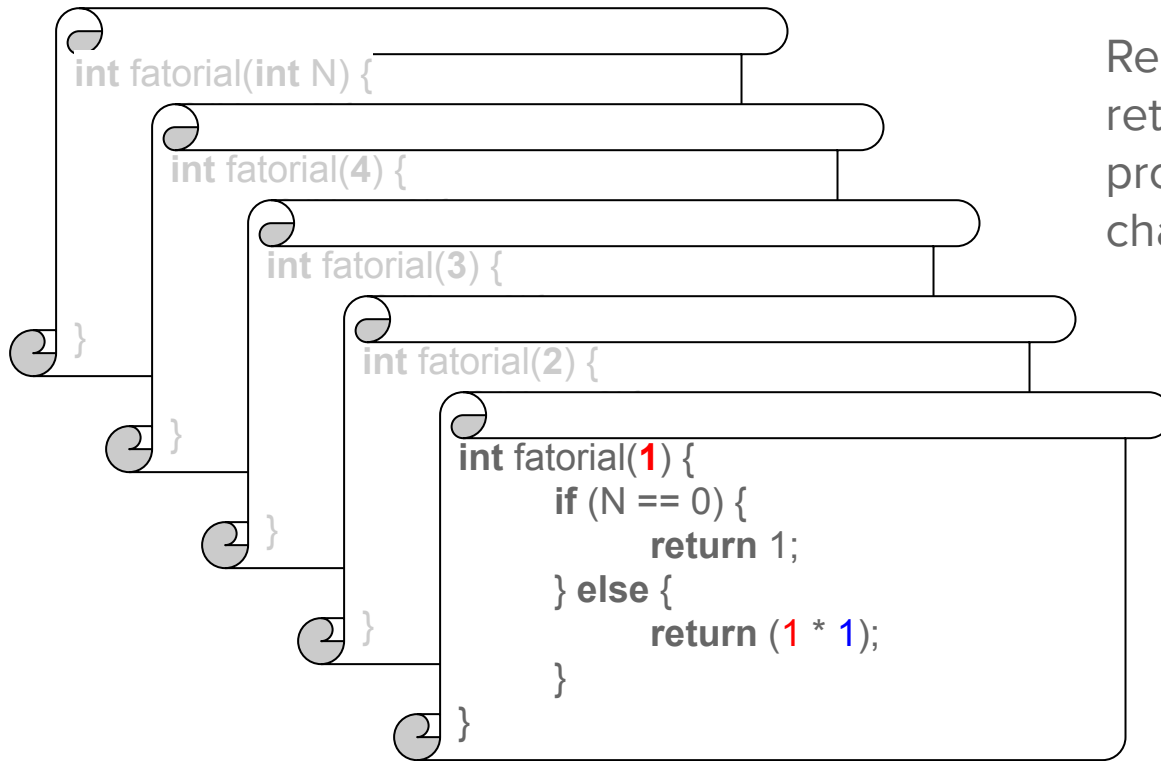
`N = 1`

Chamada recursiva: `fatorial(0)`

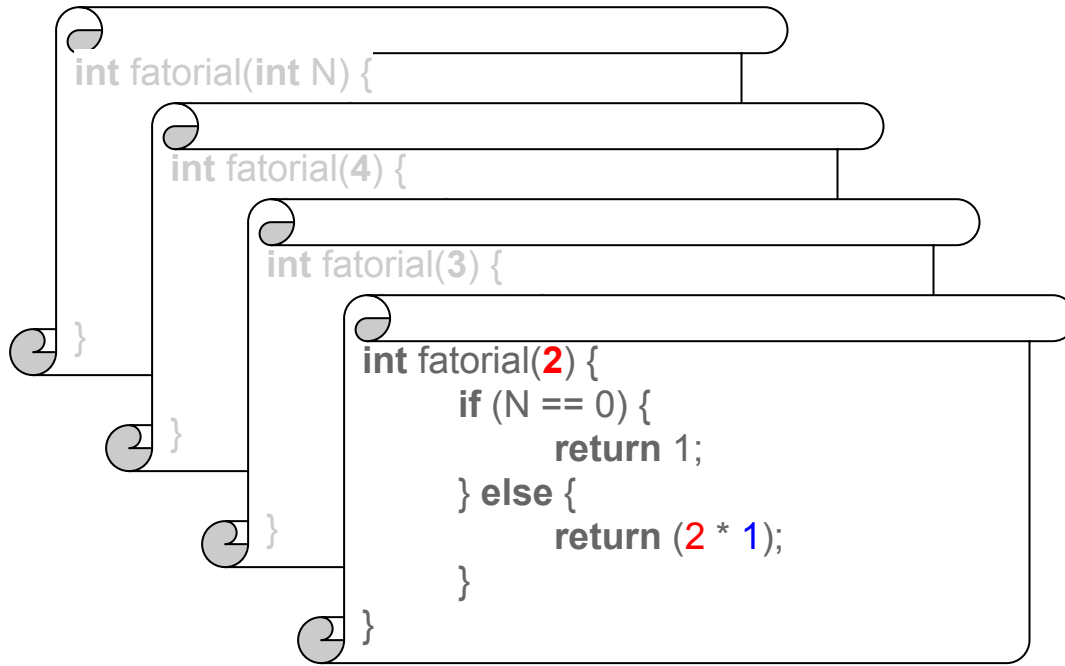


N = 0 (caso base)

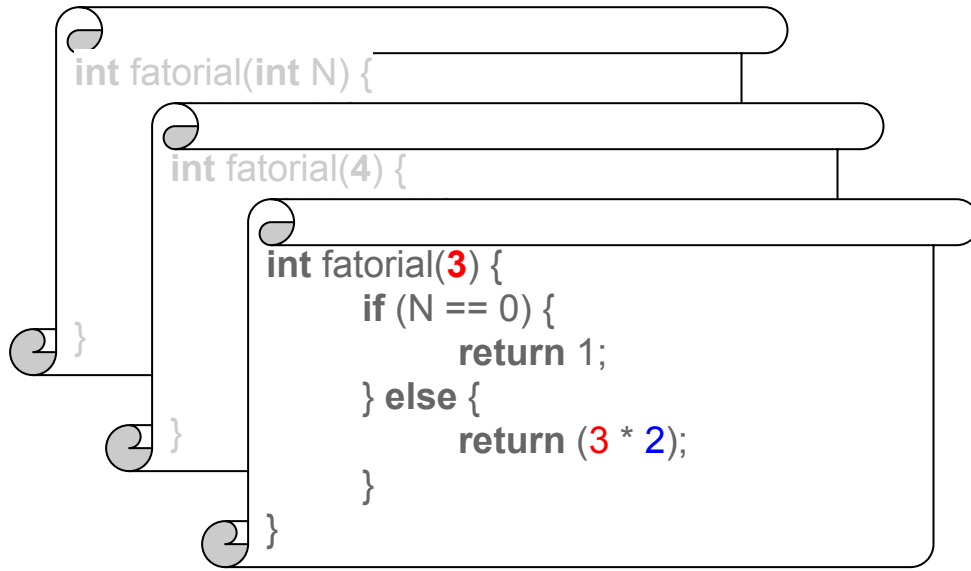
A partir deste ponto, começamos a resolver a pilha de recursão, inicialmente retornando o valor 1 para a chamada anterior.



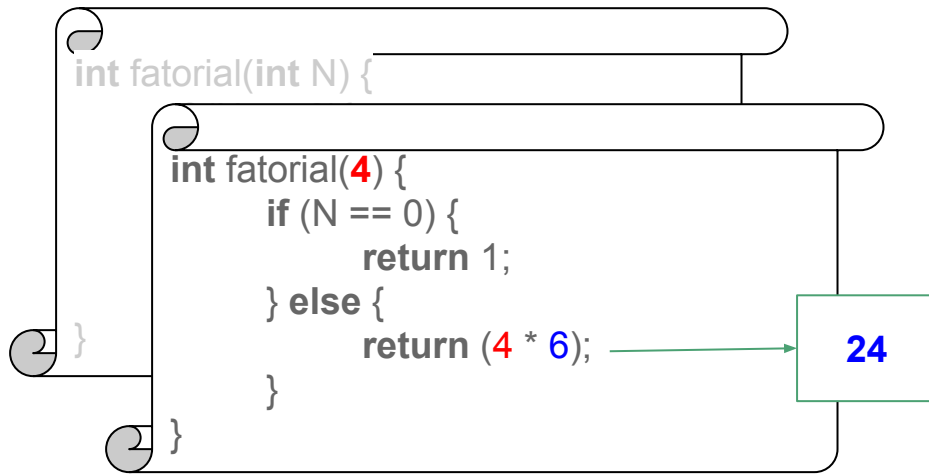
Resolve o `fatorial(1)`,
retornando o resultado do
produto $1 * 1$ (ou seja, 1) para a
chamada anterior



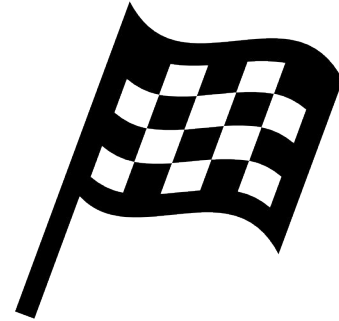
Resolve o `fatorial(2)`, retornando o resultado do produto $2 * 1$ (ou seja, 2) para a chamada anterior



Resolve o `fatorial(3)`, retornando o resultado do produto $3 * 2$ (ou seja, 6) para a chamada anterior



Resolve o `fatorial(4)`, retornando o resultado do produto $4 * 6$ (ou seja, 24) para o ponto que inicialmente solicitou o cálculo de `fatorial(4)`



Outro exemplo - apresentação resumida

```
fatorial(6) =  
  = 6 * fatorial(5)  
  = 6 * (5 * fatorial(4))  
  = 6 * (5 * (4 * fatorial(3)))  
  = 6 * (5 * (4 * (3 * fatorial(2))))  
  = 6 * (5 * (4 * (3 * (2 * fatorial(1)))))  
  = 6 * (5 * (4 * (3 * (2 * (1 * fatorial(0))))))) CASO BASE  
  = 6 * (5 * (4 * (3 * (2 * (1 * 1))))) =  
  = 6 * (5 * (4 * (3 * (2 * 1))))  
  = 6 * (5 * (4 * (3 * 2)))  
  = 6 * (5 * (4 * 6))  
  = 6 * (5 * 24)  
  = 6 * 120  
  = 720
```

Pilha de recursão

Note que o programa precisa esperar o fim das chamadas recursivas subsequentes para que possa alcançar um resultado.

Deve-se ter cuidado ao definir os casos bases de um subprograma recursivo, caso contrário, o mesmo nunca irá parar (loop infinito).



Programas recursivos e não-recursivos

Geralmente é possível transformar um programa recursivo em um programa não-recursivo, e vice-versa.

Contudo, programas recursivos tendem a ser mais lentos e utilizar mais memória do que suas contrapartidas não-recursivas.

Como vantagem, a recursão nos permite expressar algoritmos complexos de forma mais elegante, compacta e com melhor entendimento.



Recursão mútua

É possível definir funções recursivas, em que a recursão de uma função é dada por uma outra, e vice-versa. Tal recurso permite a implementação de programas mais claros e legíveis.

Deve-se tomar cuidado com os casos bases de cada função.

Exemplo de problema: de modo geral, pode-se dizer que um número inteiro positivo é par se o seu antecessor imediato for um número ímpar. Em contrapartida, um número é ímpar se o seu antecessor imediato for um número par.

Recursão mútua

- Exemplo com recursão mútua: números pares e ímpares

```
bool par(int N) {  
    if (N == 0)  
        return true;  
    else  
        return impar(N-1);  
}
```

```
bool impar(int N) {  
    if (N == 0)  
        return false;  
    else  
        return par(N-1);  
}
```

Embora as declarações destes subprogramas isoladamente estejam corretas, se montarmos um programa completo com elas, um erro de compilação irá ocorrer.

Cuidado: Exemplo com erro

```
#include <iostream>
using namespace std;

bool par(int N) {
    if (N == 0) return true;
    else return impar(N-1);
}

bool impar(int N) {
    if (N == 0) return false;
    else return par(N-1);
}

int main(){
    int X;
    cin >> X;
    if (par(X)) cout << X << " é par." << endl;
    else cout << X << " é ímpar." << endl;
    return 0;
}
```

Cuidado: Exemplo com erro

```
#include <iostream>
using namespace std;

bool par(int N) {
    if (N == 0) return true;
    else return impar(N-1);
}
bool impar(int N) {
    if (N == 0) return false;
    else return par(N-1);
}
int main(){
    int X;
    cin >> X;
    if (par(X)) cout << X << " é par." << endl;
    else cout << X << " é ímpar." << endl;
    return 0;
}
```

Erro de sintaxe

In function 'bool par(int):
error: 'impar' was not declared in this scope



Declaração / implementação

- Em C e C++, qualquer função precisa ser declarada antes de ser utilizada, por isso, o código anterior não compila.
 - No exemplo, a função **par()** utiliza uma função não declarada chamada **impar()**.
- Contudo, é importante destacar, que a declaração de um subprograma em C e C++ não precisa vir imediatamente acompanhada de sua implementação.
 - Podemos declarar qualquer subprograma apenas indicando: (1) tipo de retorno do subprograma; (2) identificador do subprograma; e (3) tipos de dados dos parâmetros formais do subprograma.

Exemplo corrigido

```
#include <iostream>
using namespace std;

bool impar(int);
bool par(int N) {
    if (N == 0) return true;
    else return impar(N-1);
}
bool impar(int N) {
    if (N == 0) return false;
    else return par(N-1);
}
int main(){
    int X;
    cin >> X;
    if (par(X)) cout << X << " é par." << endl;
    else cout << X << " é ímpar." << endl;
    return 0;
}
```

Declaração da função ímpar - note que o identificador do parâmetro não é necessário!

Exemplo corrigido

```
#include <iostream>
using namespace std;

bool impar(int N);
bool par(int N) {
    if (N == 0) return true;
    else return impar(N-1);
}
bool impar(int N) {
    if (N == 0) return false;
    else return par(N-1);
}
int main(){
    int X;
    cin >> X;
    if (par(X)) cout << X << " é par." << endl;
    else cout << X << " é ímpar." << endl;
    return 0;
}
```

Declaração da função ímpar - apesar do identificador do parâmetro não ser necessário, o código fica mais legível!



Algoritmos de busca em vetores



Algoritmos de busca em vetores

- Contextualização

- Muitas vezes, é preciso verificar se um dado elemento está presente em um vetor e em qual posição.
- Em outras palavras, dado um vetor **V** de **N** posições, queremos determinar se um elemento **K** está ou não no vetor.



Algoritmos de busca em vetores

- Contextualização

- Por exemplo: dado $\mathbf{V} = \{2, -3, 101, 104, 0, 1, 2, 3, -3, 0\}$ com $\mathbf{N} = 10$ posições, queremos saber se $\mathbf{K} = 104$ está presente ou não no vetor. Em caso afirmativo, queremos saber também qual posição ele ocupa.



$\mathbf{K} \leftarrow 104$

Neste exemplo, $\mathbf{K} \in \mathbf{V}$ e ocupa a posição (índice) **3** do vetor.

Algoritmos de busca em vetores

- Contextualização
 - O exemplo anterior foi bem simples, agora imagine verificar se um dado elemento está presente em um vetor com milhares ou milhões de elementos. **Como resolver este problema?**
 - Existem na literatura, diversos algoritmos que resolvem o problema da busca em vetores. Nesta disciplina, veremos duas estratégias:
 - **Busca linear ou sequencial.**
 - **Busca binária.**

Algoritmos de busca em vetores

- **Busca linear ou sequencial**

- Consiste em verificar sequencialmente as posições em um vetor **V** de **N** elementos, uma a uma, até encontrar o elemento **K** ou chegar ao final do vetor.
 - No **melhor** caso, verificamos apenas **1** posição do vetor (o primeiro elemento investigado do vetor é o próprio **K**).
 - No **pior** caso, verificamos **N** posições (**K** é o último elemento investigado do vetor ou ele não está presente no mesmo).
 - No caso **médio**, verificamos **N/2** posições (**K** está em alguma posição mediana do vetor).

Algoritmos de busca em vetores



- **Busca linear ou sequencial**

- A seguir, no próximo slide, será apresentado um exemplo de busca sequencial em um vetor de números inteiros.
- **Atenção:** o exemplo de implementação do próximo slide é apenas isso, um exemplo. É possível implementar o conceito da busca sequencial de diferentes formas e utilizando vetores de diferentes tipos de dados.
 - **Recomendação:** não tente decorar a implementação do exemplo. Tente entender como a busca sequencial funciona!

Exemplo: Busca sequencial

```
#include <iostream>
using namespace std;
int main() {
    int K, N = 10;
    int V[N];
    int i, posicao = -1;
    for (i = 0; i < N; i++) cin >> V[i];
    cin >> K;
    i = 0;
    while (i < N) {
        if (V[i] == K){
            posicao = i;
        }
        i++;
    }
    cout << posicao << endl;
    return 0;
}
```

Variáveis

- **K**: elemento a ser buscado no vetor.
- **N**: quantidade de elementos do vetor.
- **V**: vetor de elementos.
- **i**: variável auxiliar para caminhar nas posições do vetor.
- **posicao**: variável que armazena a posição do elemento **K** no vetor **V**, caso ele seja encontrado.



Exemplo: Busca sequencial

```
#include <iostream>
using namespace std;
int main(){
    int K, N = 10;
    int V[N];
    int i, posicao = -1;
    for (i = 0; i < N; i++) cin >> V[i];
    cin >> K;
    i = 0;
    while (i < N){
        if (V[i] == K){
            posicao = i;
        }
        i++;
    }
    cout << posicao << endl;
    return 0;
}
```

Inicialização da variável **posicao**. Nesta implementação, estamos assumindo o valor **-1** para indicar que o elemento não foi encontrado

Leitura dos elementos do vetor **V**

Leitura do elemento de busca **K**

Repetição da busca linear. Note que, caso o elemento **K** seja encontrado no vetor **V**, alteramos o valor da variável **posicao** para demarcar o índice no qual ele foi encontrado.

Algoritmos de busca em vetores

- **Busca linear ou sequencial**

- É possível melhorar a eficiência da busca no algoritmo do exemplo de implementação apresentado.
- Note que, na implementação apresentada, mesmo que encontremos o elemento **K** no vetor **V**, ainda precisamos percorrer todos os índices presentes no vetor até que cheguemos no valor de **N** (pois construímos a repetição *while (i < N)*).
 - Sendo assim, podemos melhorar nosso algoritmo, interrompendo a busca tão logo identifiquemos que **K** está presente em **V**.

Exemplo: Busca sequencial **melhorada**

```
#include <iostream>
using namespace std;
int main(){
    int K, N = 10;
    int V[N];
    int i, posicao = -1;
    for (i = 0; i < N; i++) cin >> V[i];
    cin >> K;
    i = 0;
    while ((i < N) and (V[i] != K))
        i++;
    if (i != N)
        posicao = i;
    cout << posicao << endl;
    return 0;
}
```

Repetição melhorada da busca linear. Note que neste caso avançamos o valor do índice **i** enquanto não encontrarmos o elemento **K** ou chegarmos no final do vetor.

Se ao sair do comando **while**, o valor de **i** não é igual a **N** (quantidade de elementos), isto quer dizer que saímos do **while** porque encontramos o elemento **K** dentro do vetor e, portanto, podemos registrar o último índice **i** assumido dentro da repetição como sendo a posição do elemento **K** no vetor **V**.

Algoritmos de busca em vetores



*E se os dados do vetor
estiverem ordenados?
Dá para parar antes???*

Exemplo: Busca sequencial **ordenada**

```
#include <iostream>
using namespace std;
int main(){
    int K, N = 10;
    int V[N];
    int i, posicao = -1;
    for (i = 0; i < N; i++) cin >> V[i];
    cin >> K;
    i = 0;
    while ((i < N) and (V[i] < K))
        i++;
    if ((i != N) and (V[i] == K))
        posicao = i;
    cout << posicao << endl;
    return 0;
}
```

Neste **while**, interrompemos a repetição da busca linear assim que encontramos um elemento maior do que o valor procurado ou quando chegamos ao final do vetor.

Se ao sair do comando **while**, o valor de **i** não é igual a **N** (quantidade de elementos), isto quer dizer que saímos do **while** porque encontramos o elemento **K** dentro do vetor ou que encontramos um elemento cujo valor é maior do que **K**. Sendo assim, registramos o índice **i** apenas se **K** for igual a **V[i]**.

Algoritmos de busca em vetores

- **Custo da busca em vetor ordenado**
 - Se o vetor estiver ordenado, a busca é mais eficiente em casos médios. Contudo, o pior caso continua sendo a verificação de **N** posições do vetor.



- Futuramente, iremos aprender a **como** ordenar um vetor de elementos. Assim, não precisaremos assumir que o usuário do programa irá fornecer um vetor já ordenado.

Algoritmos de busca em vetores

- **Busca binária**

- É possível melhorar a eficiência da busca em vetores ordenados se nós modificarmos a nossa estratégia do processo de busca.
- A busca binária é um método de busca em **vetores ordenados** e que reduz o tamanho do problema (quantidade de elementos a serem investigados) a cada iteração.
 - A ideia básica é ir “quebrando” o vetor ao meio, em cada iteração, e verificar em qual metade pode estar o valor. Repete-se o processo até que o vetor de busca tenha tamanho 1, sendo encontrado ou não o elemento de interesse.

Algoritmos de busca em vetores

- **Busca binária**

- Seja **V** um vetor de **N** posições e queremos determinar se um elemento **K** está ou não presente no vetor, o algoritmo geral em pseudocódigo da busca binária pode ser expresso como:

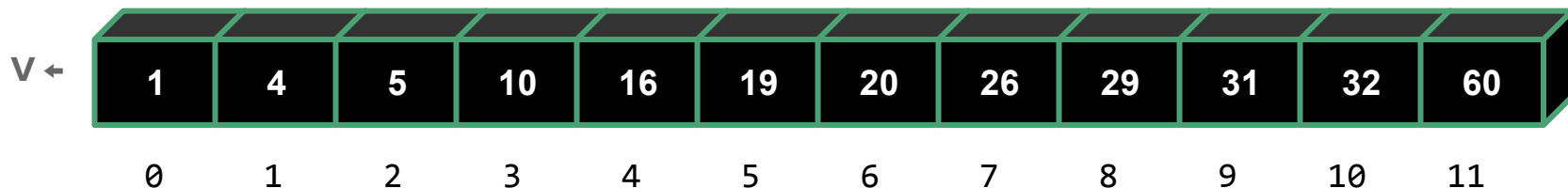
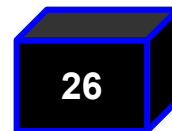
Pseudocódigo da busca binária em vetores de ordem crescente

1. Comparamos **K** com o elemento do meio do vetor ordenado em ordem crescente.
2. Se o elemento do meio for igual a **K**, terminamos a busca, pois encontramos o elemento.
3. Se o elemento do meio for maior do que **K**, a metade da direita pode ser descartada da busca.
4. Se o elemento do meio for menor do que **K**, a metade da esquerda pode ser descartada da busca.
5. Repetimos então este mesmo processo com a metade restante do vetor. Se esta metade for vazia, conclui-se que o elemento não está presente no vetor.

Algoritmos de busca em vetores

- Busca binária

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$

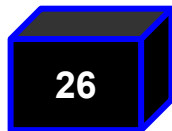


Nesse caso, calculando o meio,
Meio = $(11-0)/2 = 5$ // *divisão inteira!*
tem-se a posição 5, de valor 19.

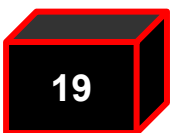
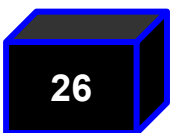
Algoritmos de busca em vetores

- Busca binária

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$



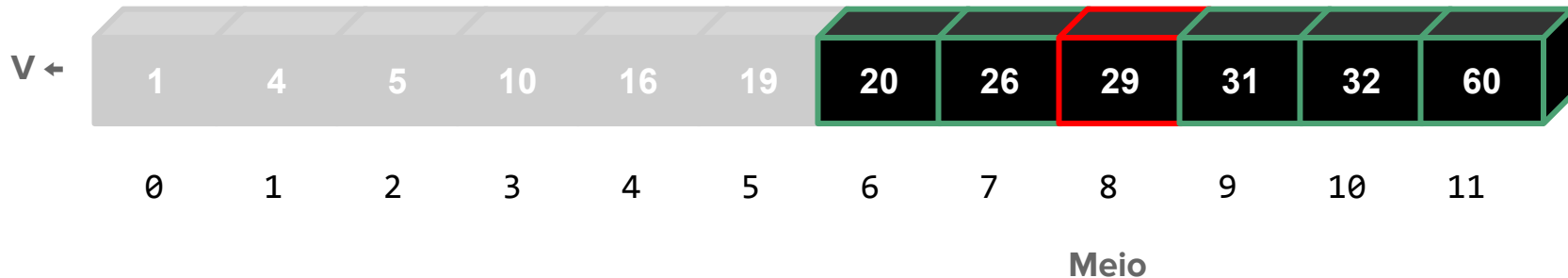
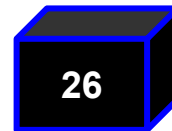
Meio

Como  $<$  podemos descartar a metade esquerda do vetor

Algoritmos de busca em vetores

- **Busca binária**

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$



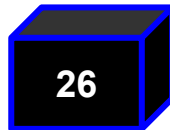
Calculando a posição central novamente (usando posições iniciais e finais do trecho), tem-se:

$$\text{Meio} = (11+6)/2 = 8$$

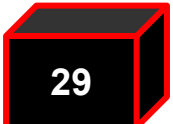
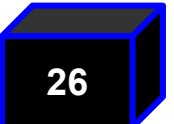
Algoritmos de busca em vetores

- **Busca binária**

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$



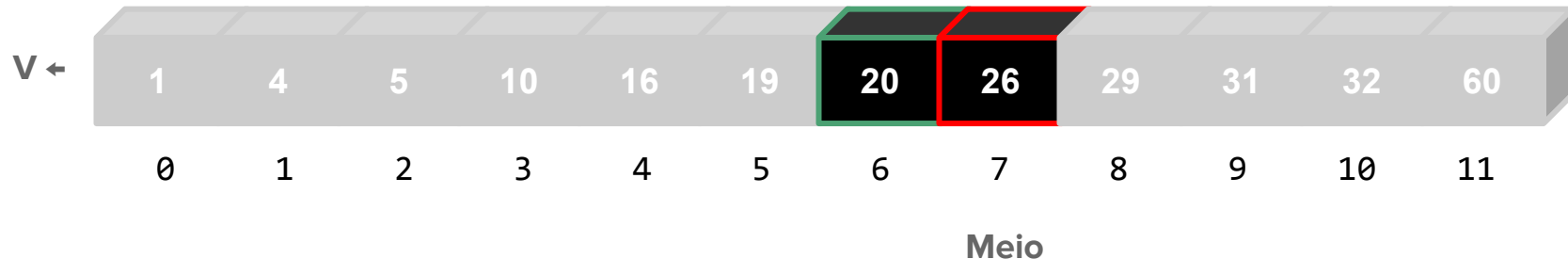
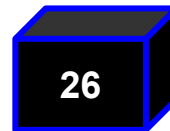
Meio

Como  $>$  podemos descartar a metade direita do vetor

Algoritmos de busca em vetores

- **Busca binária**

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$



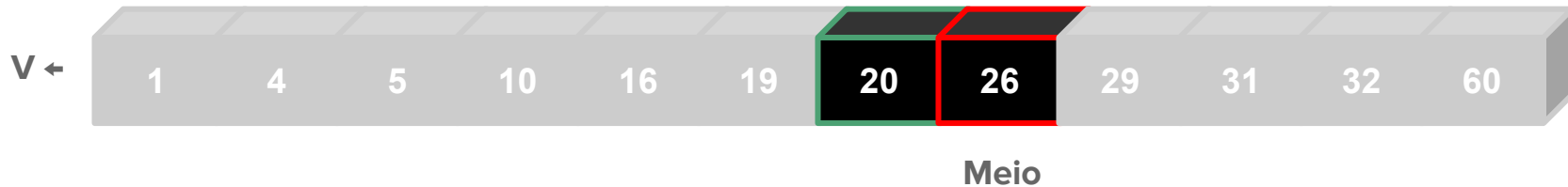
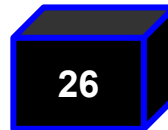
Calculando a posição central novamente (usando posições iniciais e finais do trecho), tem-se:

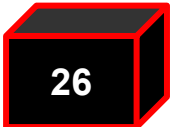
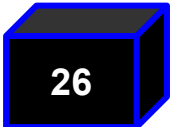
$$\text{Meio} = (8+6)/2 = 7$$

Algoritmos de busca em vetores

- **Busca binária**

- Exemplo: suponha que desejamos buscar $K \leftarrow 26$



Como  =  podemos encerrar a busca, pois encontramos o elemento



Ok, no desenho
tá legal...

Vamos ao
código??

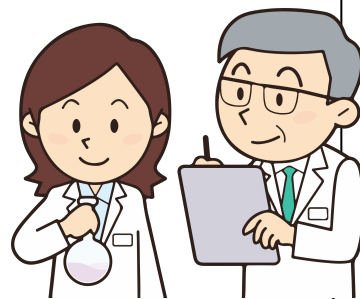
Exemplo: Busca binária 1/2

```
#include <iostream>
using namespace std;
int main(){
    int K, N = 10;
    int V[N];
    int i, posicao = -1;
    int pos_inicial = 0;
    int pos_final = N - 1;
    int meio;

    // os elementos do vetor devem ser
    // fornecidos em ordem crescente
    for (i = 0; i < N; i++) cin >> V[i];
    cin >> K;
```

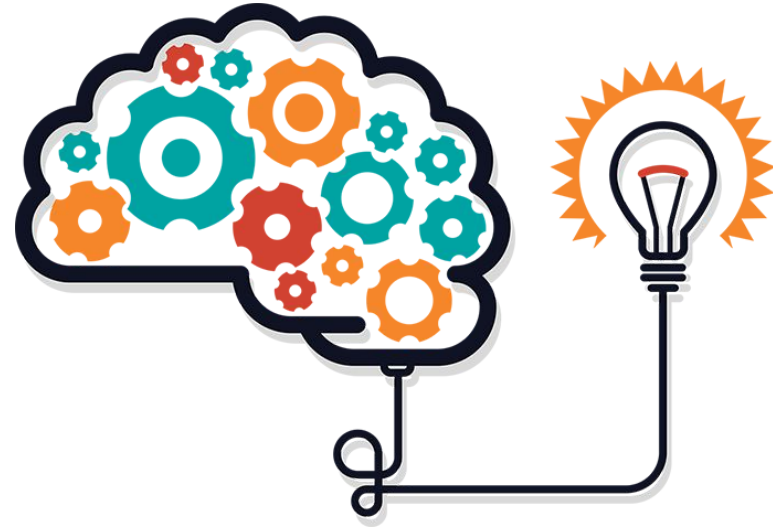
Exemplo: Busca binária 2/2

```
while (pos_inicial <= pos_final) {
    meio = (pos_inicial + pos_final)/2;
    if (K == V[meio]){
        posicao = meio;
        pos_inicial = pos_final + 1; // parar while
    } else{
        if (K > V[meio]) pos_inicial = meio + 1;
        else pos_final = meio - 1;
    }
}
cout << posicao << endl;
return 0;
}
```



Variáveis

- **K**: elemento a ser buscado no vetor.
- **N**: quantidade de elementos do vetor.
- **V**: vetor de elementos.
- **i**: variável auxiliar para caminhar nas posições do vetor.
- **posicao**: variável que armazena a posição do elemento **K** no vetor **V**, caso ele seja encontrado.
- **pos_inicial**: variável que armazena o valor do índice mais à esquerda do trecho do vetor considerado na busca. Como no primeiro momento, todo o vetor é considerado, a variável é inicializada com o índice do primeiro elemento.
- **pos_final**: variável que armazena o valor do índice mais à direita do trecho do vetor considerado na busca. Como no primeiro momento, todo o vetor é considerado, a variável é inicializada com o índice do último elemento.
- **meio**: variável que armazena o valor do índice central do trecho do vetor considerado na busca.



Interessante,
mas e se eu
quiser a busca
binária em forma
de função
recursiva?



```
#include <iostream>
using namespace std;

int BuscaBinaria(int V[], int pos_inicial, int pos_final, int K) {
    int meio = (pos_inicial + pos_final)/2;

    if (K == V[meio]) { //caso base: elemento encontrado
        return meio;
    }
    else if (pos_inicial < pos_final) { //caso geral: processo de busca
        if (V[meio] < K) return BuscaBinaria(V, meio+1, pos_final, K);
        else return BuscaBinaria(V, pos_inicial, meio-1, K);
    } else { //caso base: elemento não encontrado
        return -1;
    }
}
```

Exemplo: Busca binária **recursiva** 1/2

```
#include <iostream>
using namespace std;

int BuscaBinaria(int vetor[], int inicio, int fim, int procurado) {
    if (inicio <= fim){
        int meio = (inicio+fim)/2;
        if (procurado > vetor[meio])
            return binariaRecursiva(vetor,meio+1,fim,procurado);
        else if (procurado < vetor[meio])
            return binariaRecursiva(vetor,inicio,meio-1,procurado);
        else
            return meio;
    }
    return -1;
}
```




```
int main() {  
    int K, N = 10;  
    int V[N];  
  
    // os elementos do vetor devem ser  
    // fornecidos em ordem crescente  
    for (int i = 0; i < N; i++) cin >> V[i];  
    cin >> K;  
  
    cout << BuscaBinaria(V, 0, N-1, K) << endl;  
    return 0;  
}
```



Algoritmos de busca em vetores

- Comentários finais
 - Quando o vetor não se encontra ordenado, utiliza-se a busca sequencial, com custo proporcional ao tamanho do vetor.
 - Quando o vetor encontra-se ordenado, utiliza-se a busca binária, que possui complexidade $\lg(N)$, em que **N** é o tamanho do vetor. Para vetores muito grandes (milhares ou milhões de posições), esta diferença nos custos de execução se mostra determinante.
 - Devido às características de custo nestes dois casos, métodos de ordenação de vetores são fundamentais e serão apresentados futuramente na disciplina.

Sobre o Material



Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).