Linguagem de Programação I

Aula 5 - Funções e Recursividade: Tipos de recursão, ordem de execução e pilha de recursão

Objetivos da aula

- **Aprofundar-se nos conceitos de recursividade**
  - Segmentos de memória e suas utilidades
  - Entender como a recursividade funciona na memória do computador, cujas instruções são sequenciais
  - Identificar as vantagens e desvantagens do uso de recursividade
  - Conhecer técnicas para, se necessário, remover a recursividade

- Ao final da aula espera-se que o aluno seja capaz de:
  - Empregar adequadamente o uso de recursividade
  - Se necessário, remover a recursividade de seu algoritmo

Memória de trabalho do computador (RAM)

- A memória de trabalho do computador (RAM) é subdividida em vários segmentos lógicos dentro de um programa
  - Segmento de pilha (*stack*): onde sub-rotinas e métodos alocam temporariamente suas variáveis locais
  - Segmento *heap*: onde variáveis dinâmicas são alocadas (tempo de execução)
    - Bastante útil quando não se sabe de antemão quantas variáveis de determinado tipo serão necessárias para o programa
  - Segmento de dados: onde variáveis globais e estáticas são alocadas (tempo de compilação)
  - Segmento de código: onde instruções de máquina do programa são encontradas

Recursividade: exemplo de uso

- Busca binária
  - A essência do algoritmo é recursiva
  - A implementação natural (intuitiva) é, portanto, recursiva

```
1.  int binarySearch( int vector[], int x, int low, int high )
2.  {
3.      if( low > high )                // não há elementos
4.          return -1;
5.
6.      int mid = ( low + high ) / 2;   // índice do meio
7.      int y = vector[mid];            // elemento do índice do meio
8.
9.      if( x == y )
10.         return mid;                 // achou o elemento
11.     else if( x > y )                // busca à direita
12.         return binarySearch( vector, x, mid + 1, high );
13.     else                            // busca à esquerda
14.         return binarySearch( vector, x, low, mid - 1 );
15. }
```

Tipos de recursão

Simples:

$$fat(n) = \begin{cases} 1 & \text{se } n = 0 \\ n.\, fat(n-1) & \text{se } n > 0 \end{cases}$$

Múltipla: $$fib(n) = \begin{cases} n & \text{se } n \in \{0, 1\} \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

Tipos de recursão

Indireta:

$$par(n) = \begin{cases} \text{verdadeiro} & \text{se } n = 0 \\ impar(n-1) & \text{se } n > 0 \end{cases}$$

$$impar(n) = \begin{cases} falso & \text{se } n = 0 \\ par(n-1) & \text{se } n > 0 \end{cases}$$

Aninhada:

$$ack(n,m) = \begin{cases} m+1 & \text{se } n = 0 \\ ack(n-1,m) & \text{se } n > 0; m = 0 \\ ack(n-1, ack(n, m-1)) & \text{se } n > 0; m > 0 \end{cases}$$

Como funciona internamente?

- As chamadas são empilhadas na stack
  - Variáveis locais e seus valores são conservados
  - O estado é retornado ao voltar da chamada

```
1. int fat( int n )
2. {
3.     if( n == 0 )
4.         return 1;
5.     else
6.         return n * fat( n - 1 );
7. }
```

- Em geral, a área alocada para a stack é menor que a heap
- Um grande número de chamadas recursivas pode estourar a stack (stack overflow)

Recursão na stack

Escopo	Var.locais	End.Stack	Valor	Retorno
fat(0)	n	0x100A1018	0	1
fat(1)	n	0x100A1014	1	1
fat(2)	n	0x100A1010	2	2
fat(3)	n	0x100A100C	3	6
fat(4)	n	0x100A1008	4	24
fat(5)	n	0x100A1004	5	120
main()	n	0x100A1000	???	.

```
1. int fat( int n )
2. {
3.     if( n == 0 )
4.         return 1;
5.     else
6.         return n * fat( n - 1 );
7. }
```

Experimento - fatorial e fibonacci

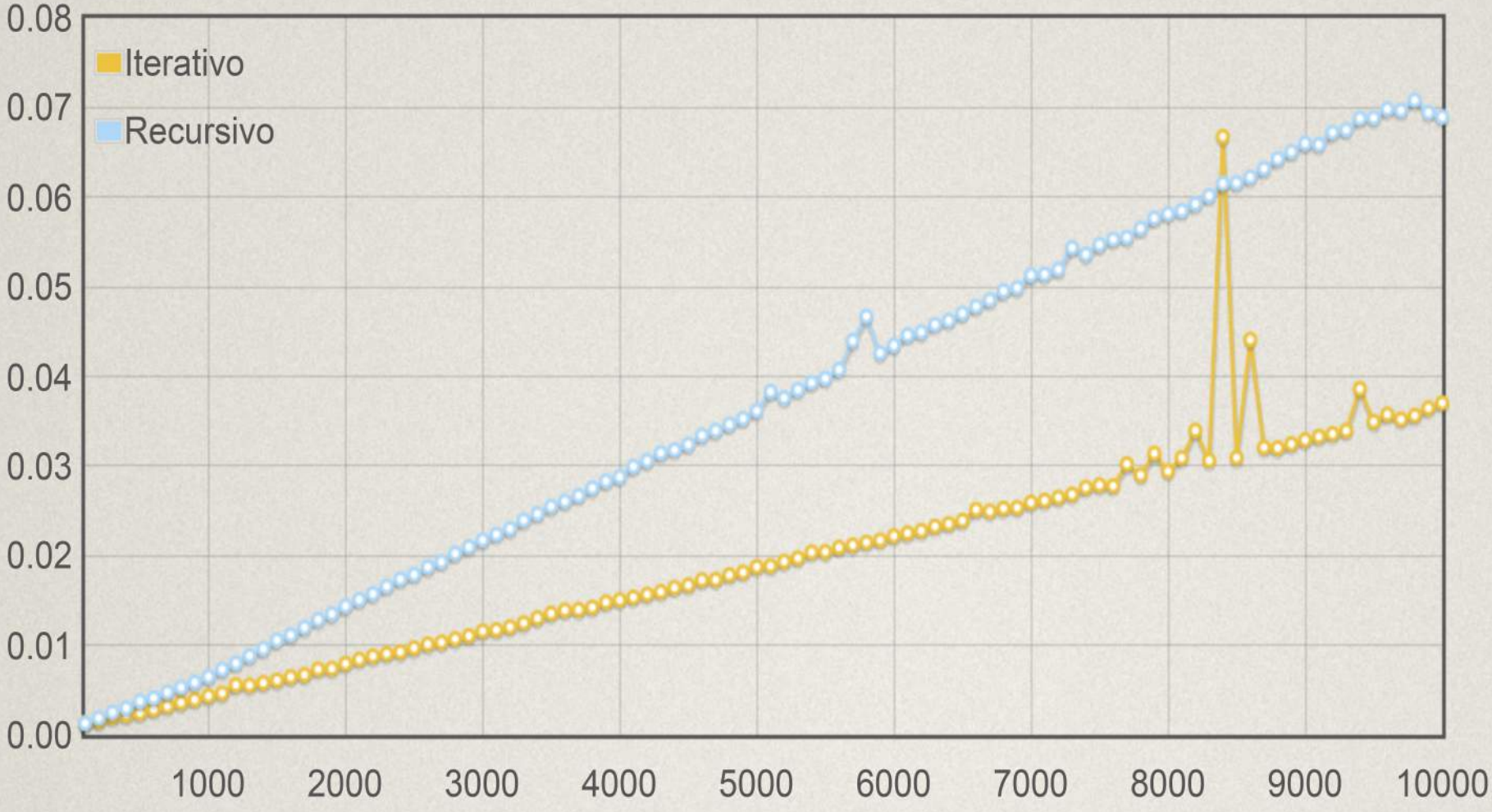
- Análise de desempenho
- Comparação entre as versões **I**terativas e **R**ecursivas

```
1.  int fatI( int n )
2.  {
3.      int fat = 1;
4.      for( int i = 2; i <= n; i++ )
5.          fat *= i;
6.      return fat;
7.  }
8.
9.  int fatR( int n )
10. {
11.     if( n == 0 )
12.         return 1;
13.     return n * fatR( n - 1 );
14. }
```

```
1.  int fibI( int n )
2.  {
3.      int fib = n, fibAA = 0, fibA = 1;
4.      for( int i = 2; i <= n; ++i )
5.      {
6.          fib   = fibA + fibAA;
7.          fibAA = fibA;
8.          fibA  = fib;
9.      }
10.     return fib;
11. }
12.
13. int fibR( int n )
14. {
15.     if( n == 0 || n == 1 )
16.         return n;
17.     else
18.         return fibR( n - 1 ) + fibR( n - 2 );
19. }
```

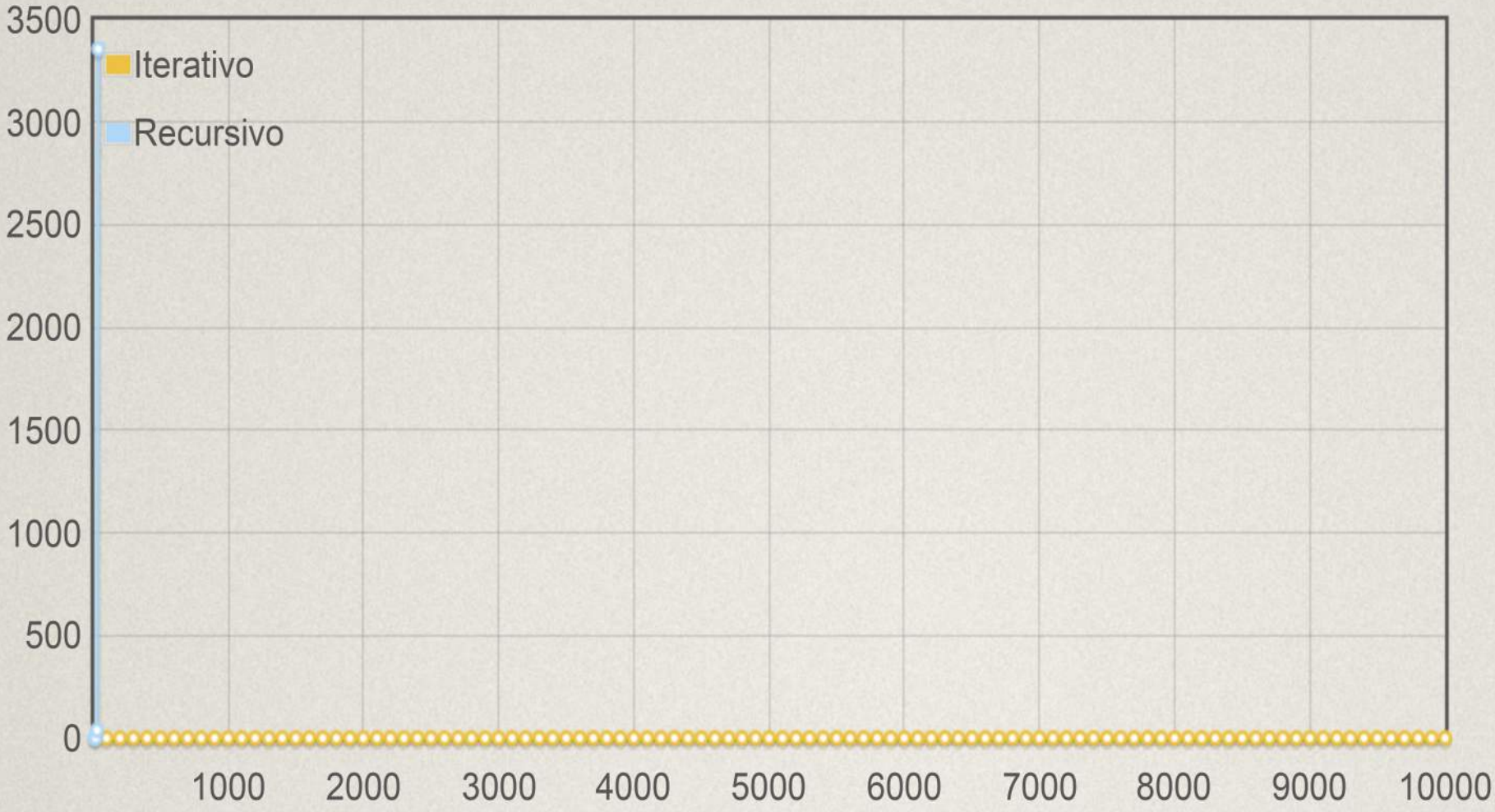
Experimento 1 - fatorial

- Análise de desempenho (tempo em milisegundos)
  - 10 execuções com 100 a 10.000 elementos (passo 100)



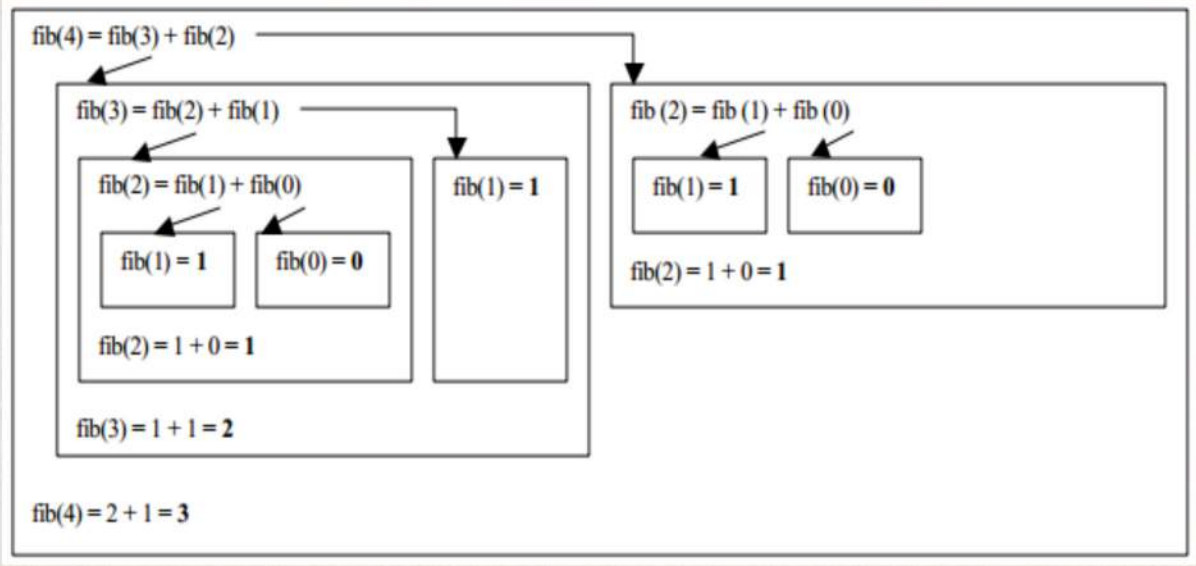
Experimento 2 - fibonacci

- Análise de desempenho (tempo em milisegundos)
  - 10 execuções com 100 a 10.000 elementos (passo 100)



Experimentos - considerações

- No fatorial, o tempo de execução da versão recursiva é cerca de 2x
- No fibonacci, o tempo é "fatorialmente" maior
  - A complexidade da solução é maior devido à redundância dos cálculos



- Se alguns algoritmos são mais "fáceis de se pensar" de forma recursiva... mas com tempo de execução impraticável...

  Como passar um algoritmo recursivo para a forma iterativa???

Recursão e iteração

- Recursão e iteração possuem o mesmo poder de expressividade
  - Algumas linguagens funcionais não possuem instruções para laços (while, for...)
  - Todo laço é realizado de forma recursiva

- Passando a recursão para a forma iterativa...
  - Eliminando a recursão de cauda (aula de hoje)
  - Manipulando com índices
  - Usando uma estrutura auxiliar para "simular" a stack (pilha)

Recursão de cauda

- É quando a última ação de uma função é a chamada recursiva
- Abaixo, mdc() possui recursão de cauda, mas fat() não
  - fat() realiza uma multiplicação após a chamada recursiva

```
1.  int mdc( int x, int y )
2.  {
3.      if( y == 0 )
4.          return x;
5.      return mdc( y, x % y );
6.  }
7.
8.  int fat( int n )
9.  {
10.     if( n == 0 )
11.         return 1;
12.     return n * fatR( n - 1 );
13. }
```

Recursão de cauda

- Na maioria das linguagens, a recursão de cauda é tratada pelo compilador, gerando uma versão iterativa correspondente
  - Como é a última ação, as chamadas não são empilhadas na stack
- Podemos fazer algo similar...

```
1.  int funcao( parametros )
2.  {
3.      if( condicao )
4.          return caso_base( parametros );
5.      operacao1;
6.      operacao2;
7.      operacao3;
8.      ajusta( parametros );
9.      return funcao( parametros );
10. }
```

```
1.  int funcao( parametros )
2.  {
3.      int result = caso_base( parametros);
4.      while( !condicao ) {
5.          operacao1;
6.          operacao2;
7.          operacao3;
8.          ajusta( parametros );
9.          result = caso_base( parametros );
10.     }
11.     return result;
12. }
```

Exemplo 1

- Exemplo de remoção no cálculo do MDC

```
1.  int mdc( int x, int y )
2.  {
3.      if( y == 0 )
4.          return x;
5.      return mdc( y, x % y );
6.  }
```

```
1.  int mdc( int x, int y )
2.  {
3.      int aux;
4.      int result = x;
5.      while( y != 0 )
6.      {
7.          aux = x;
8.          x = y;
9.          y = aux % y;
10.         result = x;
11.     }
12.     return result;
13. }
```

Exemplo 2

- Mesmo algumas sub-rotinas que não possuem recursão de cauda podem se beneficiar da estratégia (com algumas pequenas adaptações)
- Exemplo do fatorial

```
1.  int fat( int n )
2.  {
3.      if( n == 0 )
4.          return 1;
5.      return n * fat( n - 1 );
6.  }
```

```
1.  int fat( int n )
2.  {
3.      int aux;
4.      int result = 1;
5.      while( n != 0 )
6.      {
7.          aux = n;
8.          n = n - 1;
9.          result = aux * result;
10.     }
11.     return result;
12. }
```

Transformar não-cauda em cauda

- Em geral é possível (recursões múltiplas são mais complicadas)
- Usar uma função recursiva auxiliar, passando um "acumulador"
- Exemplo: inverter uma string

```
1.  string reverseString( string word ) { // não possui recursão de cauda
2.      if( word.compare( "" ) == 0 )
3.          return word;
4.      else
5.          return reverseString( word.substr( 1, word.length() - 1 ) ) + word[0];
6.  }
7.
8.  string tailReverse( string word, string res ) { // sub-rotina auxiliar
9.      if( word.compare( "" ) == 0 )                 // possui recursão de cauda
10.         return res;
11.     else
12.         return tailReverse( word.substr( 1, word.length() - 1 ), word[0] + res );
13. }
14.
15. string reverseString( string word ) { // sub-rotina original
16.     return tailReverse( word, "" );
17. }
```

Resumo da aula

- Recursão é útil pela simplicidade
  - Códigos menores
  - Mais fáceis de "ler" e, consequentemente, de entender
  - Porém, são mais custosos
- É possível transformar sub-rotinas recursivas em iterativas
  - Se a recursão é de cauda, em geral, o próprio compilador trata
  - Pode-se tratar também transformando uma sub-rotina que não possui recursão de cauda em uma que possui