Recursão

- A maioria dos programas possui frações de código que se repetem várias vezes. Isso é necessário principalmente para cálculos matemáticos.
  - Exemplo:

    $2 \times 4 =$

    $2 \times (2 \times 3) =$

    $2 \times 2 \times (2 \times 2) = 8$
- A linguagem de programação C fornece três estruturas de repetição para realizar cálculos iterativos: *for*, *while* e *do-while*. No entanto, para casos mais complexos que o exemplo anterior, a leitura desses códigos nem sempre é muito simples para a identificação de erros.
- *Recursão* é um conceito fundamental em matemática e ciência da computação. Uma função recursiva é toda aquela que chama a si mesma dentro do corpo de sua função
- Exemplo: Tente enxergar a recursão transformando uma potência de expoente alto em outra de expoente menor usando a própria teoria de potenciação:

  $3^4 =$

  $3 \times 3^3 =$

  $3 \times 3 \times 3^2 =$

  $3 \times 3 \times 3 \times 3^1 = 81$

  Pode-se dizer que, para todo inteiro positivo x e y: $x^y = x \times x^{y-1}$. Logo, para uma função recursiva:

```
int potencia(int base, int expoente)
```

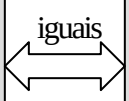
que calcula o resultado de se elevar a base ao expoente, é correto afirmar que:

```
resultado = base * potencia( base, (expoente – 1) );
```

Pelo código anterior, a cada nova chamada recursiva da função `potencia()`, o expoente será menor em uma unidade. Esse processo continua até que o expoente seja igual a 1, quando já se tem a resposta, visto que todo número elevado a 1 é igual a ele mesmo. A função recursiva fica como a seguir:

usando `recursão`

```
1: int pot(int base, int exp)
2: {
3:    if (exp == 1)
4:       return base;
5:    else
6:       return base * pot(base, exp – 1);
7: }
```



usando `iteração for`

```
1: int pot (int base, int exp)
2: {
3:    int i, resultado = 1;
4:
5:    for (i = 0; i < exp; i++)
6:       resultado *= base;
7:    return resultado;
8: }
```

Note que esta é uma função válida apenas para potências com expoente maior que zero (como todo número elevado a zero é igual a um, a função também pode receber expoente zero se a condição da linha 3 for alterada para checar se o expoente é zero e retornar o valor 1 ao invés da base).

- As abordagens da solução de problemas com recursão têm diversos elementos em comum. Genericamente, um módulo recursivo segue o algoritmo abaixo:

```
if (<condição de parada>) {
   // Resolva
} else {
   // Divida o problema num caso mais simples utilizando recursão
}
```

Note que, sem a condição de parada, o problema entraria em "loop", dividindo o problema infinitas vezes sem nunca parar. Seguindo a idéia do algoritmo proposto, a função recursiva só sabe como resolver o(s) caso(s) mais simples (caso *base* identificado na condição de parada). Na função que calcula potência, o caso base é elevar um número qualquer ao expoente 1 e retornar o próprio número (linhas 3 e 4). Se a função é chamada com um problema mais complexo, a função divide o problema em dois pedaços conceituais: um pedaço que a função sabe como resolver (caso base) e um pedaço que a função não sabe como resolver (linhas 5 e 6). Como esse novo problema se parece com o problema original, a função faz uma chamada a si mesma para resolver o problema menor (*etapa de recursão*). A etapa de recursão é executada enquanto a chamada original para a função ainda está ativa, podendo resultar ainda em muitas outras chamadas recursivas, sempre dividindo cada sub-problema novo em dois pedaços conceituais. Para que a recursão termine em algum momento, a seqüência de problemas cada vez menores deve convergir para o caso base. Nesse ponto, a função reconhece o caso base, devolve um resultado para a cópia da função anterior e uma seqüência de devoluções se segue até a chamada original da função devolver o resultado final.

- Outro problema comumente usado para exemplificar recursão é a ***série de Fibonacci***:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

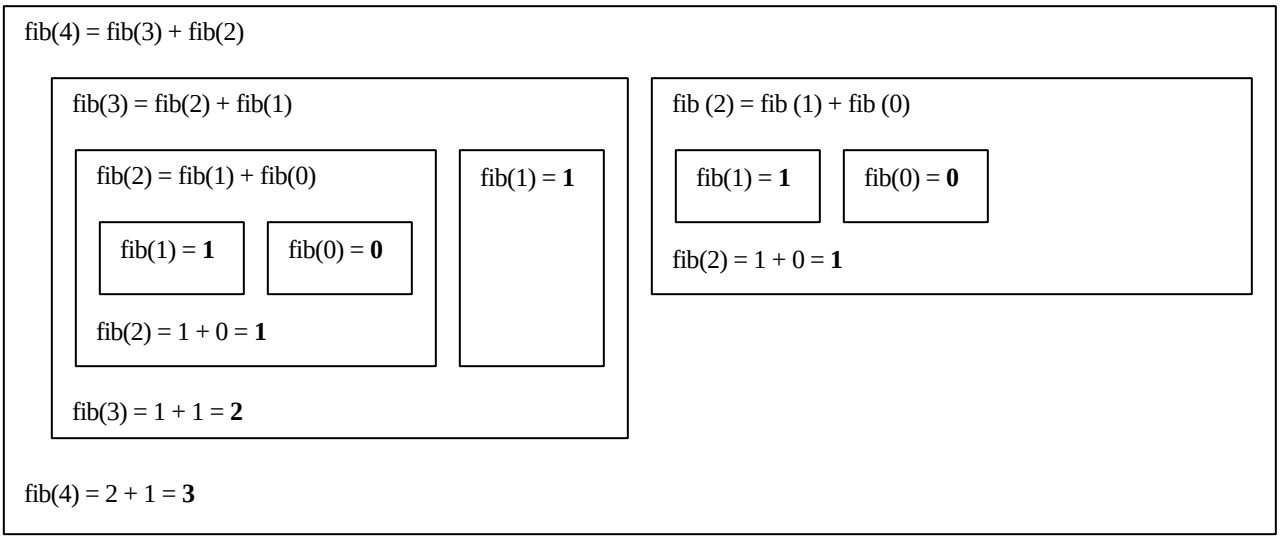
Ela inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subseqüente é a soma dos dois números que o precedem. A série é definida recursivamente como:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

Note que neste caso temos dois casos base: fibonacci(0) e fibonacci(1). A função recursiva que retorna o *i*-ésimo elemento da série de Fibonacci é dada a seguir:

```
1: int fibonacci(int num)
2: {
3:    if (num == 0)
4:       return 0;
5:    else if (num == 1)
6:       return 1;
7:    else
8:       return fibonacci(num - 1) + fibonacci(num - 2);
9: }
```

Esquematização de fibonacci(4):



Como se pode perceber pelo esquema anterior, cada invocação à função fibonacci que não corresponde a um dos casos base resulta em mais duas chamadas recursivas. Esse número de chamadas cresce rapidamente mesmo para números bem pequenos da série. Por exemplo, o fibonacci de 31 exige 4.356.617 chamadas à função enquanto que o fibonacci de 32 exige 7.049.155, ou seja, 2.692.538 chamadas adicionais à função. Logo, é interessante evitar programas recursivos no estilo de Fibonacci que resultam em uma "explosão" exponencial de chamadas. Isso porque, a cada chamada recursiva da função, ela é adicionada a uma pilha; da mesma forma, a cada término da função, ela é desempilhada. Esse empilhamento e desempilhamento repetitivo consome tempo, fazendo com que o mesmo programa implementado iterativamente venha a ser mais rápido que o recursivo (embora, por outro lado, provavelmente não seja de compreensão e manutenção tão simples). Além disso, a pilha de funções apresenta um tamanho máximo, limitando assim o uso de recursões que mantenham muitas funções na pilha (como o fibonacci de um número grande).