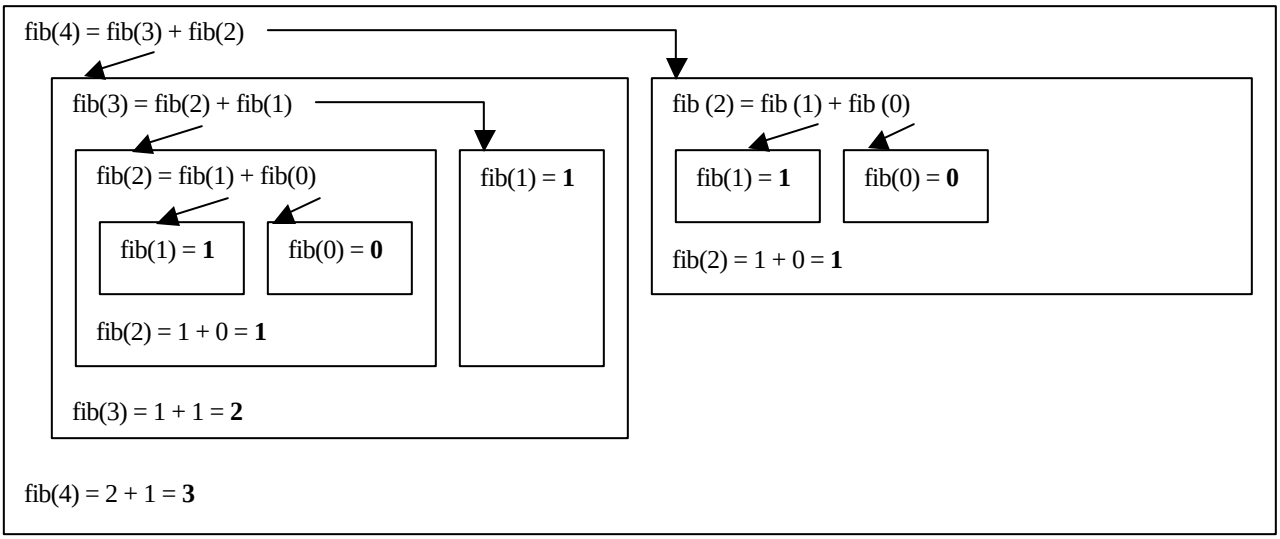
Esquematização de fibonacci(4):



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).

Recursão de Cauda

- A recursão de cauda é uma técnica de recursão que faz menos uso de memória durante o processo de empilhamento, o que a torna mais rápida que a recursão comum.
- Em uma recursão comum, a cada chamada recursiva realizada, é necessário guardar a posição do código onde foi feita a chamada para que continue a partir dali assim que receber o resultado. Logo, como dito anteriormente, se fibonacci(32) faz 7.049.155 chamadas recursivas, também serão necessárias 7.049.155 variáveis para armazenar a posição onde foi feita a chamada.
- Em uma recursão de cauda, não é necessário guardar a posição onde foi feita a chamada, visto que a chamada é a última operação realizada pela função.
- Exemplo: o ***fatorial*** de um inteiro *n* não-negativo, escrito n!, é o produto $n \times (n-1) \times (n-2) \times \ldots \times 1$.

  O cálculo recursivo do fatorial de *n* pode ser feito como a seguir:

```
1: int fatorial(int num)
2: {
3:    if (num == 1)
4:       return num;
5:    else
6:       return num * fatorial(num - 1);
7: }
```

Note que após receber o resultado da chamada recursiva (linha 6) ele deve ser multiplicado por *n*. Daí a necessidade de uma variável para guardar a posição onde foi feita a chamada recursiva. Uma recursão de cauda é aquela onde a chamada recursiva é a última operação a ser realizada pela função recursiva. A função `fatorial_cauda()` a seguir implementa a recursão de cauda para o fatorial:

```
1:  int fatorial_aux(int num, int parcial)
2:  {
3:     if (num == 1)
4:        return parcial;
5:     else
6:        return fatorial_aux((num - 1), (parcial * num));
7:  }
8:
9:  int fatorial_cauda(int num)
10: {
11:    return fatorial_aux(num, 1);
12: }
```

Nesse caso, o conceito da avaliação do caso base é diferente. Embora exista uma checagem deste caso na função `fatorial_aux()` (linha 3), o problema inicial não é reduzido até este ponto para que então a função retorne a resposta deste caso (ou seja, `fatorial(1) = 1`). O que acontece é uma transcrição direta de um cálculo iterativo: passa-se inicialmente o valor parcial 1 para a função `fatorial_aux()` (linha 11) e ela "atualiza" este valor a cada chamada recursiva, multiplicando o cálculo parcial do fatorial por n, $n - 1$, $n - 2$, …, até o "caso base". Neste ponto, o valor do fatorial já foi calculado e é apenas retornado (linha 4).

Esta recursão de cauda poderia gastar menos memória ainda se utilizasse passagem por referência dos parâmetros. Desta forma, cada função recursiva empilhada não criaria um espaço de memória para os parâmetros *n* e *parcial*, mas apenas atualizaria estas duas variáveis sem ficar fazendo cópias delas.

Recursão versus Iteração

- Tem-se a seguir uma comparação entre as duas abordagens de implementação para que o programador possa escolher qual a ideal de acordo com uma situação em particular:

Recursão	Iteração
Estruturas de seleção *if*, *if-else* ou *switch*.	Estruturas de repetição *for*, *while* ou *do-while*.
Repetição por chamadas de funções repetidas.	Repetição explícita.
Termina quando um caso base é reconhecido.	Termina quando teste do laço falha.
Pode ocorrer infinitamente.	Pode ocorrer infinitamente.
Lento.	Rápido.
Soluções simples e de fácil manutenção.	Soluções complicadas e de difícil manutenção.

- Mesmo diminuindo-se os gastos de memória da recursão através de uma implementação que usa recursão de cauda e passagem de parâmetros por referência, o cálculo iterativo será comumente mais rápido por não fazer repetidas chamadas de funções. No entanto, a recursão de cauda é importante principalmente em linguagens de programação funcionais (por exemplo, a linguagem *Scheme*), onde *não existem estruturas de repetição*. Neste caso, a recursão é a única abordagem disponível e o desempenho pode ser fator crítico, exigindo implementações de recursão de cauda.

Exercícios: Faça funções que usem recursão comum para resolver os seguintes problemas. Se for possível, construa também soluções que usem recursão de cauda.

a) *Multiplicação* – Recebe dois números positivos não nulos *a* e *b* e retorna o resultado de $a \times b$.

```
int multiplicacao(int a, int b) {
   if (b == 1)
      return a;
   else
      return a + multiplicacao( a, (b – 1) );
}
```

```
int multiplicacao_aux(int a, int b, int parcial) {
   if (b == 1)
      return parcial;
   else
      return multiplicacao_aux( a, (b - 1), (parcial + a) );
}
int multiplicacao_cauda(int a, int b) {
   return multiplicacao_aux(a, b, a);
}
```

b) *Maior* – Recebe um ponteiro para vetor de inteiros e seu tamanho e retorna o maior da lista.

c) *Pertence* - Recebe um número, um ponteiro para vetor de inteiros e seu tamanho e retorna 1 se o número pertence ao vetor e 0 caso contrário.

d) *Ocorrências* – Recebe um número, um ponteiro para vetor de inteiros e seu tamanho e retorna o número de ocorrências desse número no vetor.

e) *Primo* – Recebe um número positivo não nulo e retorna 1 se ele for primo e 0 caso contrário.

f) *Torres de Hanói* – Em um templo no Extremo Oriente, os sacerdotes estavam tentando mover uma pilha de discos de um pino para outro. A pilha inicial tinha 64 discos sobrepostos em um pino e ordenados de baixo para cima por tamanho decrescente. Os sacerdotes tentavam mover a pilha desse pino *A* para um terceiro pino *C* usando um segundo pino *B* para conter os discos temporariamente. Apenas um disco podia ser movido por vez e em nenhum momento um disco maior podia ser colocado sobre um disco menor.

Dica: Pense no mais simples possível: mover 1 disco apenas. Basta tirá-lo do pino *A* e colocá-lo no *C*. Agora pense em mover 2 discos. Primeiro move-se o disco 2 de *A* para *B* e caímos no caso anterior (pois o pino *A* agora só tem o disco 1). Então move-se o disco 1 de *A* para *C* e o disco 2 de *B* para *C*. Veja que mover *n* discos pode ser visualizado em termos de mover *n* – 1 discos como a seguir:
– Mover *n* – 1 discos do pino *A* para o pino *B*, utilizando o pino *C* como armazenamento temporário;
– Mover o último disco (o maior) do pino *A* para o pino *C*;
– Mover os *n* – 1 discos do pino *B* para o pino *C*, utilizando o pino *A* como armazenamento temporário;
Implemente a função com a seguinte declaração: `void hanoi (char a, char b, char c, int discos)`