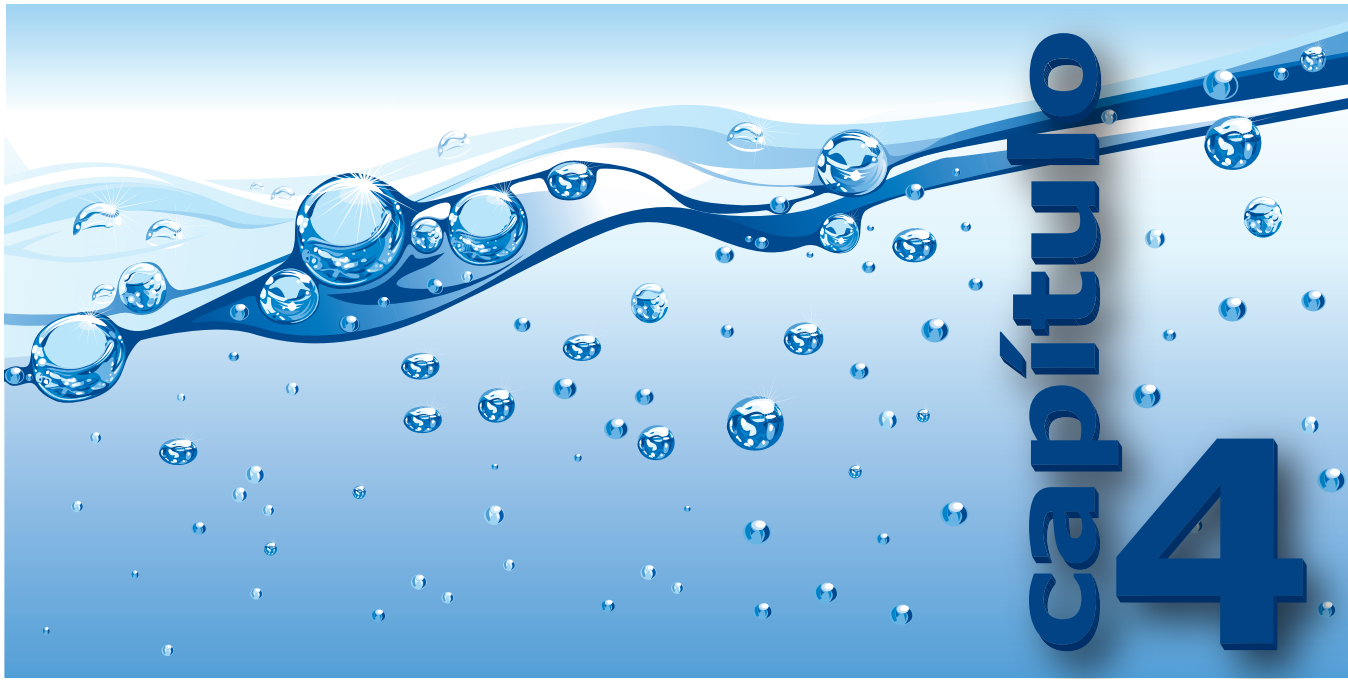

Recursão e Retrocesso

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Recursão
  - ❒ Caso base
  - ❒ Caso recursivo
  - ❒ Retrocesso
  - ❒ Diagrama de recursão
  - ❒ Pilha de execução
  - ❒ Registro de ativação
  - ❒ Esgotamento de pilha
  - ❒ Fase de decréscimo
  - ❒ Fase de acréscimo
  - ❒ Satisfação de restrição
  - ❒ Função acionadora
- Para uma dada função recursiva, executar as seguintes tarefas:
  - ❒ Determinar o caso base
  - ❒ Determinar o caso recursivo
  - ❒ Determinar se a função termina
  - ❒ Determinar o que a função faz
  - ❒ Desenhar a pilha de execução durante a fase de acréscimo e decréscimo de uma função recursiva
- Comparar recursão com iteração
- Mostrar como o espaço reservado em memória para execução de um programa é dividido
- Explicar por que uma função recursiva é geralmente menos eficiente do que uma função equivalente iterativa
- Identificar quando uma função recursiva apresenta recursão de cauda e transformá-la em função iterativa
- Decidir se uma solução recursiva é adequada para um determinado problema
- Descrever o problema das torres de Hanói e sua solução
- Saber como e quando usar recursão para resolver um determinado problema

objetivos

RECURSÃO É UMA PODEROSA FERRAMENTA em matemática e, especialmente, em programação. Quando utilizada adequadamente, a recursão tem a capacidade de reduzir o tempo gasto na implementação de algoritmos que, de outro modo, demandariam um tempo considerável. O uso de recursão pode ainda tornar programas mais simples de entender. Por outro lado, recursão apresenta um inerente efeito negativo em programação que é o fato de sempre demandar mais uso de recursos computacionais do que soluções funcionalmente equivalentes que não usam recursão.

A técnica de retrocesso, por sua vez, tem como objetivo resolver uma certa categoria de problemas computacionais utilizando o mínimo de recursos possível. Tal técnica é implementada por meio de recursão.

Este capítulo tem como enfoque essas duas importantes ferramentas de programação.

4.1 Funções Recursivas

A maioria das linguagens de programação modernas permite a escrita de funções que chamam, direta ou indiretamente, a si mesmas. Tais funções são denominadas **recursivas**. Uma função recursiva deve conter pelo menos duas partes (casos) a saber:

- **Caso base** (**não recursivo**), que estabelece uma **condição de parada** (ou **condição terminal**) da recursão e sem o qual a recursão será infinita. Esta parte da definição da função não deve fazer referência à própria função.
- **Caso recursivo**, no qual a função chama a si mesma. O programador deve garantir que uma das chamadas recursivas atinja certamente a condição de parada.

Uma função recursiva pode ter mais de um caso recursivo e mais de uma condição de parada.

A função `SomaAteN()` apresentada a seguir ilustra o processo de recursão em C:

```
int SomaAteN(int n)
{
   if (n <= 1)
      return n;           /* Condição de parada */
   else
      return (n + SomaAteN(n - 1)); /* Caso recursivo */
}
```

A função `SomaAteN()` retorna o valor da soma dos inteiros compreendidos entre `1` e `n`, sendo `n` o parâmetro de entrada da função. Por exemplo, a chamada `SomaAteN(5)` deve resultar em *15* (pois, $1 + 2 + 3 + 4 + 5 = 15$).

Um **diagrama de recursão** é uma representação gráfica utilizada como artifício para facilitar o acompanhamento de chamadas recursivas de uma função. O diagrama de recursão na **Figura 4–1 (a)** ilustra a sequência de chamadas recursivas que ocorre após a chamada inicial `SomaAteN(5)`. Nesse diagrama, quando é feita a chamada `SomaAteN(1)`, a condição de parada é atingida e a função retorna `1`. Com esse valor retornado, é possível voltar sucessivamente ao passo anterior na representação esquemática acima até que a chamada original seja atingida. Isto resulta no diagrama de recursão na **Figura 4–1 (b)**.

Note que, a cada chamada recursiva da função `SomaAteN()`, o valor do parâmetro `n` é cada vez menor, de modo que a condição de parada seja certamente atingida. Entretanto, a função `SomaAteN()` produz resultados indesejáveis se o número introduzido for menor do que `1` (verifique isso). Uma forma de corrigir a função `SomaAteN()` é modificando-a de modo que ela seja encerrada quando `n < 1`. Isto é feito na versão da função `SomaAteN()` a seguir:

```
int SomaAteN2(int n)
{
   if (n <= 0)
      return 0; /* Erro de domínio */

   if (n <= 1)
      return n; /* Condição de parada */

   return n + SomaAteN2(n - 1); /* Caso recursivo */
}
```

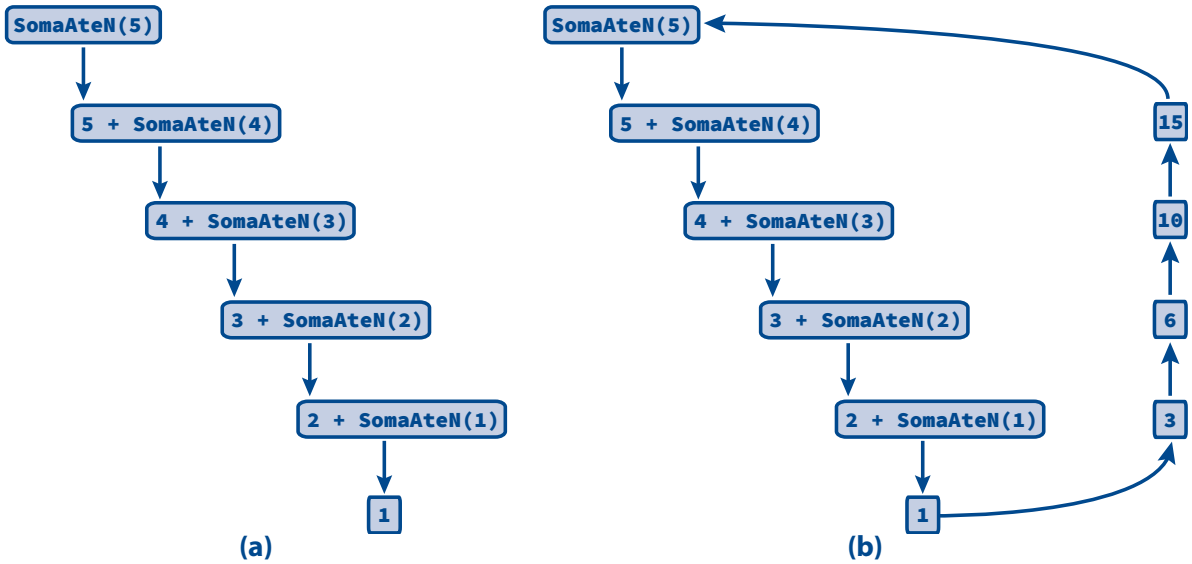

Figura 4–1: Acompanhamento de uma Função Recursiva

A função `SomaAteN2()` serve como exemplo introdutório do uso de recursão em C, mas essa evidentemente não é a forma mais elegante de se resolver o problema da soma dos números inteiros compreendidos entre `1` e `n`. Isto é, esse problema é muito mais fácil de ser resolvido utilizando um laço iterativo ao invés de recursão. Além de ser mais legível, uma versão iterativa da função `SomaAteN2()` irá provavelmente ser executada com um melhor desempenho, pois a versão recursiva envolve o uso da pilha de execução para guardar parâmetros e variáveis locais a cada chamada recursiva (v. **Seção 4.3**).

Numa chamada recursiva, um ou mais parâmetros devem ser alterados de modo a reduzir o tamanho do problema e fazer com que uma condição de parada seja atingida. Entretanto, é importante salientar que o fato de se ter certeza que uma condição de parada seja teoricamente atingida não significa que, na prática, uma função recursiva irá terminar. Quer dizer, pode ser que o número de chamadas recursivas seja tão grande que esgote a capacidade da pilha de execução e o programa seja abortado.

Um erro comum na escrita de funções recursivas é o uso de laços de repetição, em vez de desvios condicionais. Laços de repetição raramente aparecem numa função recursiva e esse tipo de erro é provavelmente decorrente do fato de o programador inexperiente ser induzido a pensar que como funções recursivas envolvem repetição, essa repetição deve ser implementada por meio de laços de repetição, como ocorre com funções iterativas.

Os diagramas de recursão apresentados nas figuras desta seção são úteis para o acompanhamento de chamadas recursivas, mas nem sempre esses diagramas são suficientes para um completo entendimento dessas chamadas. Outros diagramas mais sofisticados e que têm a mesma finalidade são árvores de recursão (v. **Seção 4.8.2**) e representações esquemáticas de pilhas de execução (v. **Seção 4.3**).

4.2 Cadeias Recursivas

Uma função pode ser recursiva sem que chame a si mesma diretamente. Isto é, uma função pode ser considerada recursiva se ela faz parte de uma **cadeia recursiva de funções**. Por exemplo, se uma função `f()` chama uma outra função `g()` que, por sua vez, chama `f()`, ambas as funções `f()` e `g()` são consideradas recursivas e formam uma cadeia recursiva.

O perigo de se ter recursão infinita é maior em cadeias recursivas do que com funções que são diretamente recursivas. Também, em termos de estilo, cadeias recursivas não são fáceis de ser identificadas como tais, pois examinando-se apenas uma das funções envolvidas não dá para perceber que a mesma chama indiretamente a si mesma.

4.3 Pilha de Execução e Registros de Ativação

O espaço reservado para execução de um programa é dividido em partes que possuem finalidades próprias, como mostra de modo simplificado a **Figura 4–2**.

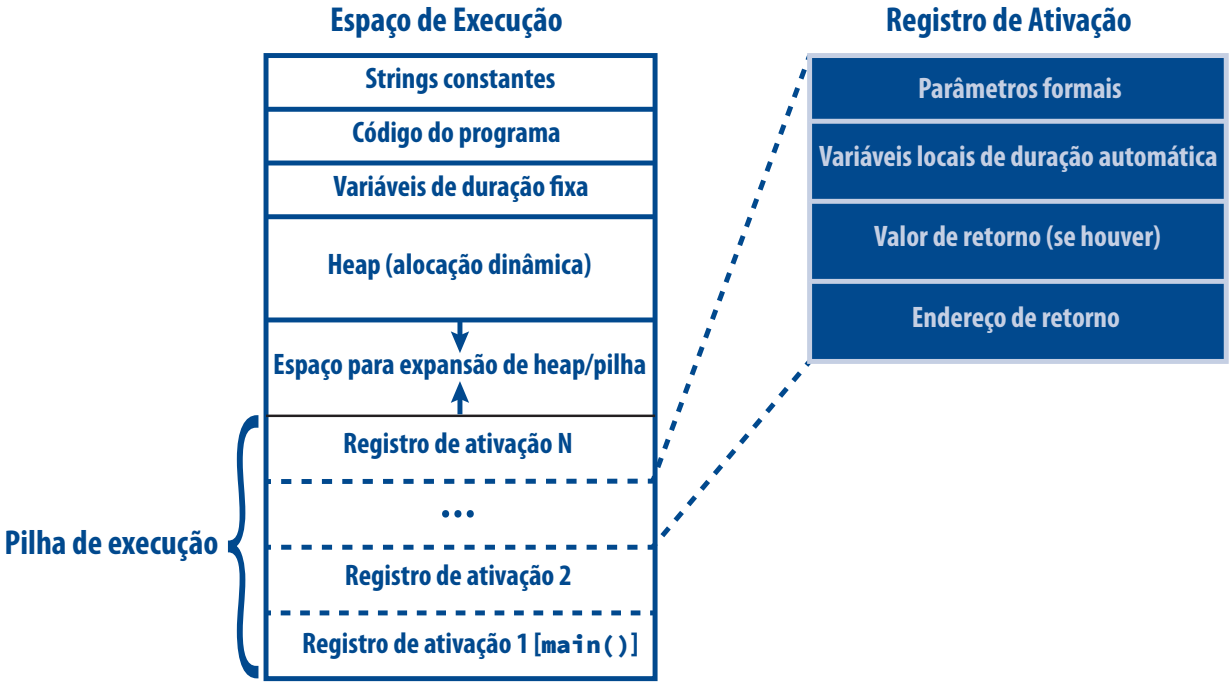

Figura 4–2: Espaço de Execução de Programa

As partições de memória reservadas para a execução de um programa podem ser descritas de modo simplificado como:

- **Strings constantes** e **código de programa**. Essas duas partições na porção superior da **Figura 4–2** são reservadas para conter as instruções do programa em linguagem de máquina bem como os dados do programa que não devem ser alterados. Isto é, esse espaço é considerado apenas para leitura e muitos sistemas operacionais encerram um programa se ele tentar alterar o conteúdo dessa área. O espaço alocado para essas partições tem tamanho fixo durante toda a execução do programa.
- **Variáveis de duração fixa**. A terceira partição de baixo para cima na **Figura 4–2** abriga variáveis de duração fixa e o espaço alocado é fixo durante toda a execução do programa.

Na **Figura 4–2**, há duas partes que aumentam ou diminuem de tamanho durante a execução do programa:

- **Heap**[1]. Essa é a partição de memória é reservada para alocação dinâmica de memória (v. **Capítulo 9**). Essa porção de memória aumenta de tamanho à medida que espaço é alocado dinamicamente em memória e diminui de tamanho quando esse espaço é liberado. O programador é diretamente responsável pela variação de tamanho do heap. Essa partição será discutida em profundidade mais adiante no **Capítulo 9**. A alocação de memória no heap é mais lenta do que na pilha porque envolve gerenciamento de memória mais complexo.
- **Pilha de execução**. Essa partição no topo da figura é denominada *pilha* porque seu funcionamento se assemelha ao de uma pilha de objetos. Isto é, os blocos armazenados nesse espaço são liberados na ordem inversa de alocação (como ocorre, por exemplo, com uma pilha de pratos). Alocação na pilha ocorre quando uma função é chamada e liberação ocorre quando uma função retorna. Ou seja, quando uma função é chamada, nesse espaço são alocados os parâmetros da função, suas variáveis de duração automática e o endereço da instrução que será executada quando a função retornar. Mais precisamente, a pilha de execução de um programa é dividida em blocos contíguos em memória denominados registros de ativação. A cada chamada de função, é criado um registro de ativação para essa chamada contendo: o endereço da instrução que fez a chamada, cópias dos parâmetros reais utilizados na chamada e as variáveis locais de duração automática da função. Quando a função retorna, o espaço alocado em memória para o registro de ativação da chamada é liberado. Em qualquer instante, a pilha de execução contém todos os registros de ativação associados a funções correntemente em execução (i.e., que ainda não retornaram). Nessa porção de memória são armazenadas informações sobre cada chamada de função efetuada no programa. A pilha de ativação é subdividida em partes denominadas **registros de ativação**, sendo que cada um desses registros está associado a uma chamada de função. Em C, o primeiro registro de ativação armazenado na pilha de execução está associado à função **main()** porque essa é a primeira função de um programa a ser chamada. Apenas o registro de ativação da função **main()** permanece ativo durante toda a execução de um programa escrito em C. A razão pela qual essa porção de memória é denominada pilha será discutida no **Capítulo 9**.

Sempre que uma função é chamada, um novo registro de ativação é criado e armazenado (i.e., empilhado) na pilha de execução. Enquanto uma função não encerra sua execução (i.e., não retorna) seu registro de ativação permanece armazenado na pilha. O conteúdo de um registro de ativação, ilustrado na porção direita da **Figura 4–2**, é o seguinte:

- **Parâmetros formais**. Seção de um registro de ativação reservada para alocação dos parâmetros formais da função (se houver algum).
- **Variáveis locais de duração automática**. Se uma função possuir variáveis locais de duração automática, elas serão alocadas nessa seção do registro de ativação da função. Variáveis de duração fixa (locais ou não) não são alocadas na pilha (v. lado esquerdo da **Figura 4–2**).
- **Valor de retorno**. Se o tipo de retorno de uma função não for **void**, o registro de ativação da função terá um espaço reservado para armazenamento do valor a ser retornado pela função.
- **Endereço de retorno**. O endereço de retorno armazenado no registro de ativação de uma função é o endereço da instrução para a qual o fluxo de execução do programa será desviado quando a função retornar.

Durante a criação de um registro de ativação ocorre o seguinte:

1. Parâmetros formais são alocados e iniciados (i.e., recebem valores dos parâmetros reais)

[1] Existe uma estrutura de dados, que será estudada no **Volume 2**, cuja denominação também é *heap*. Essa estrutura de dados não possui nenhuma relação com o conceito de heap discutido aqui.

2. Variáveis locais são armazenadas na pilha
3. Endereço de retorno é armazenado na pilha
4. O indicador de topo da pilha (v. **Seção 8.1**) é incrementado
5. É feito o desvio para a primeira instrução da função

Durante a remoção de um registro de ativação ocorrem os seguintes fatos:

1. O indicador de topo da pilha é decrementado da quantidade de memória usada pelas variáveis locais (i.e., as variáveis locais são desempilhadas — v. **Seção 8.1**)
2. Endereço de retorno é removido da pilha
3. O indicador de topo da pilha é decrementado da quantidade de memória usada pelos parâmetros formais
4. É feito o desvio para o local da chamada da função

Como exemplo de funcionamento de uma pilha de execução, considere o seguinte programa:

```
#include <stdio.h> /* printf() */

int SomaAteN3(int n)
{
   int s;

   if (n <= 0)
      return 0; /* Erro de domínio */

   if (n <= 1)
      return n; /* Condição de parada */

      /* Caso recursivo */
   s = n + SomaAteN3(n - 1); /* endereço e2 */

   return s;
}

int main(void)
{
   int soma;

   soma = SomaAteN3(3); /* endereço e1 */

   printf("\n>> Soma de 1 ate' 3: %d\n", soma);

   return 0;
}
```

No programa acima, utiliza-se uma versão da função `SomaAteN2()` apresentada na **Seção 4.1**. Essa nova versão recebeu o acréscimo da variável `s`, que, conforme foi visto na **Seção 4.1**, é absolutamente desnecessária. Aqui, o propósito dessa variável é meramente didático conforme será visto adiante. Aliás, a variável `soma` definida na função **main()** também é desnecessária e serve o mesmo propósito didático.

A **Figura 4–3 (a)** mostra a pilha de execução do programa em questão antes da chamada da função `SomaAteN3()` enquanto a **Figura 4–3 (b)** apresenta a mesma pilha logo após a execução dessa chamada. O endereço `e1` que aparece nessa última figura é o endereço de retorno da função `SomaAteN3()`; ou seja, `e1` é o endereço da instrução:

```
soma = SomaAteN3(3); /* endereço e1 */
```

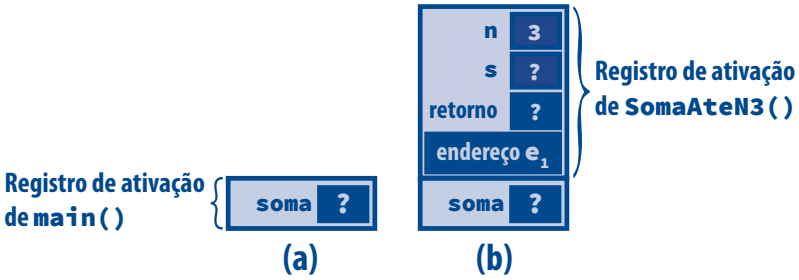

Figura 4–3: Pilha de Execução e Registros de Ativação

A **Figura 4–4** mostra a pilha de execução do programa em tela após as duas chamadas recursivas da função `SomaAteN3()`. O endereço `e2` que aparece nessa figura é o endereço de retorno de cada chamada recursiva; quer dizer, `e2` é o endereço da instrução:

```
s = n + SomaAteN3(n - 1); /* endereço e2 */
```

Quando uma chamada recursiva de função acrescenta um registro de ativação na pilha de execução, diz-se que ela se encontra em sua **fase de acréscimo**. Por outro lado, quando a base da recursão de uma função recursiva é atingida, e os registros de ativação passam a ser devidamente removidos da pilha de execução, diz-se que a função está em **fase de decréscimo**. Uma função recursiva que apresenta apenas fase de acréscimo é uma função com **recursão infinita** e o programa que a executa será encerrado por falta de espaço na pilha para alocação de mais registros de ativação. Esse tipo de erro é conhecido como **esgotamento de pilha** (ou *stack overflow*, em inglês). A **Figura 4–4** mostra a função `SomaAteN3()` em sua fase de acréscimo enquanto a **Figura 4–5** apresenta a fase de decréscimo dessa função.

Figura 4–4: Fase de Acréscimo de uma Função Recursiva

A discussão apresentada nesta seção leva à conclusão que manter informações sobre cada chamada de função em registros de ativação pode consumir muito espaço em memória, especialmente quando se lida com programas contendo muitas chamadas recursivas. Além disso, empilhar e desempilhar registros de ativação na pilha de execução são atividades que podem consumir tempo considerável em programas dessa natureza.

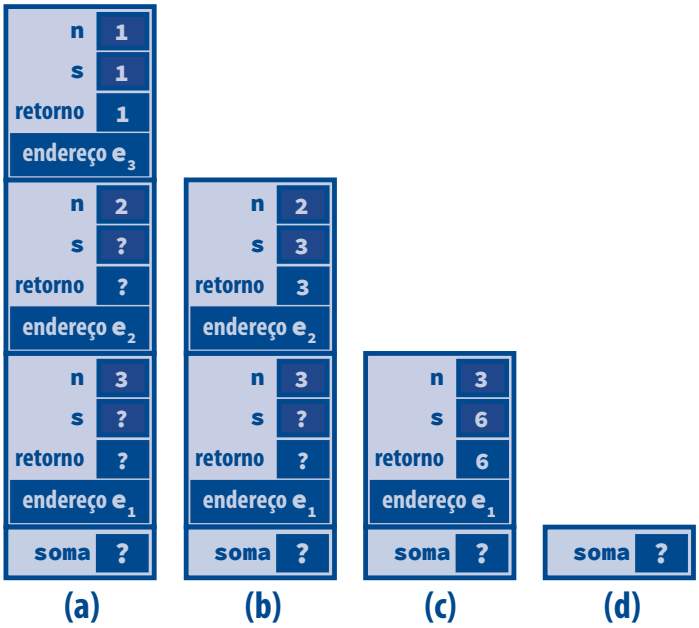

Figura 4–5: Fase de Decréscimo de uma Função Recursiva

Cada chamada de função, inclusive chamada de função recursiva, acarreta a criação de um registro de ativação. Referências a parâmetros e variáveis locais de duração automática utilizam os valores nos respectivos registros de ativação. Por exemplo, na chamada da função `SomaAteN3()`:

```
SomaAteN3(5)
```

o valor armazenado no espaço reservado para o parâmetro `n` é `5`. Então, a função `SomaAteN3()` é chamada recursivamente como:

```
SomaAteN3(4)
```

Agora, o valor armazenado no espaço reservado para o parâmetro `n` é `4`. Talvez, você esteja se perguntando: e o que ocorre com o valor anterior de `n` que era `5`? A resposta a esse aparente dilema é simples: os dois valores coexistem. Lembre-se que o que caracteriza uma variável ou parâmetro são três atributos: **(1)** nome, **(2)** endereço e **(3)** conteúdo (ou valor). Aqui, os dois valores do parâmetro `n` residem em registros de ativação diferentes e, portanto, possuem endereços diferentes. Portanto não importa que eles apresentem o mesmo nome. Neste contexto, pode-se pensar como se parâmetros e variáveis locais de duração automáticas em registros de ativação diferentes tivessem escopos diferentes.

Para gerenciamento da pilha de execução, compilador age da seguinte maneira:

- Para cada chamada de função, o compilador cria um **prólogo**, que é o código responsável pela criação do registro de ativação (i.e., alocação de variáveis e parâmetros, casamento de parâmetros etc.)
- Para cada retorno de função, o compilador cria um **epílogo**, que é o código responsável pela liberação do espaço ocupado pelo registro de ativação e o devido retorno ao local no qual a função foi chamada

4.4 Recursão de Cauda

Conforme foi mostrado acima, recursão é uma técnica de programação que requer custos relativamente elevados em termos de espaço em memória e tempo de processamento em relação a soluções iterativas equivalentes. Uma situação na qual recursão pode ser facilmente transformada em iteração é quando a única chamada recursiva de

uma função é a última instrução da função *a ser executada*, sem levar em consideração os casos não recursivos. Esse tipo de recursão é denominado recursão é denominada **recursão de cauda** (ou *tail recursion*, em inglês).

Quando se diz no parágrafo acima que recursão de cauda ocorre quando a única chamada recursiva de uma função é a última instrução *executada* da função, essa afirmação deve ser interpretada literalmente. Quer dizer, quando uma chamada recursiva *faz parte* da última instrução de uma função, não se pode dizer que essa função possui recursão de cauda. Por exemplo, a função `Fatorial()` a seguir não possui recursão de cauda:

```
int Fatorial(int n)
{
   if (n < 0)
      return -1; /* Erro de domínio */

   if (!n)
      return 1; /* Caso terminal */
   return n*Fatorial(n - 1); /* Caso recursivo */
}
```

A função `Fatorial()` não possui recursão de cauda porque sua última instrução a ser executada é:

```
return n*Fatorial(n - 1);
```

que não é uma chamada recursiva, apesar de uma chamada recursiva fazer parte dessa instrução.

É importante salientar que recursão de cauda ocorre apenas quando a última instrução executada numa função é uma chamada recursiva e essa chamada encerra a função. É importante notar ainda que a *última instrução a ser executada* não precisa necessariamente aparecer na última linha de instrução. Por exemplo, a função `EmArrayRec()` apresentada abaixo e discutida em detalhes na **Seção 4.6** possui recursão de cauda:

```
int EmArrayRec(const int ar[], int inf, int sup, int num)
{
      /* Verifica os casos base */
   if (ar[inf] == num)  /* Caso base 1 */
         /* O valor procurado é o primeiro do array */
      return inf; /* Encontrado um elemento igual a 'num' */
   else if (inf >= sup)  /* Caso base 2 */
         /* Quando o índice inferior do array é maior */
         /* do que ou igual ao seu índice superior,   */
         /* todo o array já foi examinado             */
      return -1; /* Elemento não foi encontrado */

      /* Caso recursivo: procura no restante do array */
   return EmArrayRec(ar, inf + 1, sup, num);
}
```

A função `InverteArrayRec()`, que aparece no programa apresentado a seguir, inverte a ordem dos elementos e também apresenta recursão de cauda.

```
#include <stdio.h> /* printf() */

/****
 * InverteArrayRec(): Inverte um array recursivamente
 *
 * Parâmetros:
 *      ar (entrada e saída) - array que será invertido
 *      inicio (entrada) - primeiro índice do array
 *      fim (entrada) - índice final do array
 *
 * Retorno: Nada
 ****/
```

```
void InverteArrayRec(int ar[], int inicio, int fim)
{
   int aux;

   if (inicio < fim) {
         /* Troca elementos que se encontram nos índices 'inicio' e 'fim' */
      aux = ar[inicio];
      ar[inicio] = ar[fim];
      ar[fim] = aux;

      InverteArrayRec(ar, inicio + 1, fim - 1); /* Endereço e2 */
   }
}

int main(void)
{
   int array[] = {2, 5, 1, 7, 3},
       i, final;

      /* Calcula o último índice do array */
   final = sizeof(array)/sizeof(array[0]) - 1;

   InverteArrayRec(array, 0, final); /* Endereço e1 */

   printf("\n\n\t>>> Array Invertido <<<\n\n\t> ");
   for (i=0; i <= final; i++)
      printf("%d   ", array[i]);

   return 0;
}
```

A **Figura 4–6** mostra a pilha de execução do programa acima com o registro de ativação da chamada da função `InverteArrayRec()` logo após essa função ser chamada pela função **main()**. Por sua vez, a **Figura 4–7** apresenta as pilhas de execução do programa em questão após a penúltima e a última chamadas recursivas da função `InverteArrayRec()`.

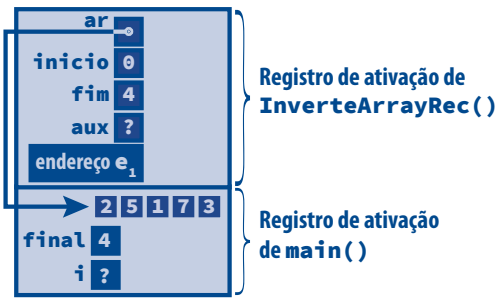

FIGURA 4–6: REGISTROS DE ATIVAÇÃO DE UMA FUNÇÃO COM RECURSÃO DE CAUDA 1

A função `InverteArrayRec()`, apresentada acima pode ser facilmente reescrita como uma função iterativa conforme mostrado a seguir:

```
void InverteArray(int ar[], int inicio, int fim)
{
   int aux;

   while (inicio < fim) {
         /* Troca elementos que se com índices 'inicio' e 'fim' */
      aux = ar[inicio];
      ar[inicio] = ar[fim];
      ar[fim] = aux;
```

```
        ++inicio;
        --fim;
    }
}
```

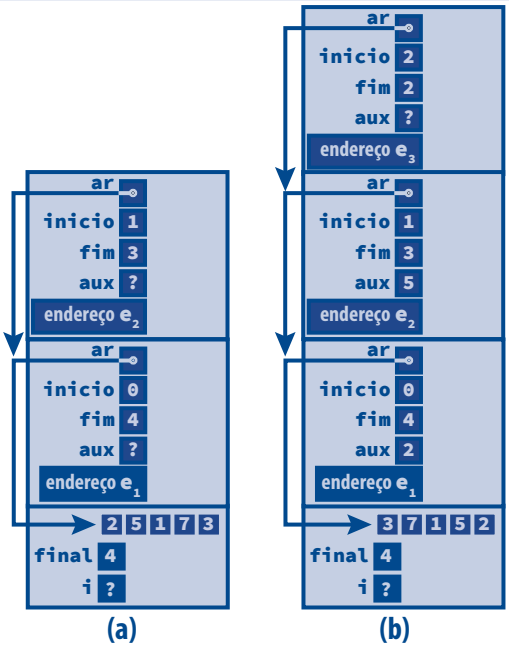

Figura 4–7: Registros de Ativação de uma Função com Recursão de Cauda 2

Para o programador, a importância de saber identificar recursão de cauda numa função recursiva é que ela frequentemente pode ser transformada numa iteração mais eficiente. Isso acontece porque, quando uma chamada recursiva é a última instrução a ser executada de uma função, sua execução não precisa mais dos dados armazenados no registro de ativação criado na última chamada dessa função, de modo que ele pode ser desempilhado da pilha de execução.

Transformar uma função que não apresenta recursão de cauda em função iterativa é mais complicado porque requer o uso explícito de pilha (v. **Capítulo 9**). Além disso, a função iterativa resultante pode não ser fácil de entender.

Bons compiladores são capazes de reconhecer recursão de cauda e otimizar o código gerado. Quer dizer, quando identificam uma recursão de cauda esses compiladores não criam um novo registro de ativação associado a essa chamada. Em vez disso, eles sobrescrevem o registro de ativação atual, uma vez que ele não terá mais utilidade.

4.5 Retrocesso (*Backtracking*)

Retrocesso (ou **backtracking** em inglês) é uma técnica de programação usualmente implementada por meio de recursão e especialmente apropriada para uma categoria de problemas denominados *problemas de satisfação de restrições* (v. **Seção 4.5.2**).

A técnica de retrocesso pode ser aplicada apenas para problemas que admitem candidatos a soluções parciais e possuem testes para verificar se, num dado instante, os candidatos podem constituir uma solução completa. Quer dizer, a técnica de retrocesso enumera um conjunto de candidatos parciais que, em princípio, podem ser completados de várias maneiras para resultar em todas as possíveis soluções para um dado problema. Retrocesso não serve, por exemplo, para problemas como busca e ordenação de dados.

O processo de retrocesso funciona da seguinte maneira:

- Se um possível candidato a solução não for promissor, outros candidatos que o incluam não são explorados. Por exemplo, no problema das oito rainhas, que será explorado adiante, se uma rainha não pode ser alocada numa determinada casa, é inútil tentar encontrar a solução do problema posicionando as demais rainhas se aquela rainha continuar ocupando uma casa inadequada.
- Em cada passo, busca-se o próximo candidato à solução e, se for descoberto que não se pode mais progredir até uma solução, retrocede-se um nível atrás e recomeça-se com um novo candidato. No problema das oito rainhas, se todas as casas de uma linha já foram tentadas na alocação de uma dada rainha, tenta-se reposicionar a rainha na linha anterior. Se isso não for possível retrocede-se à linha anterior à linha anterior e assim por diante.

À primeira vista, não parece fácil entender o que o procedimento acima pretende realmente dizer. Por isso, esse procedimento será ilustrado a seguir com o clássico problema das oito rainhas. Após estudar a seção a seguir, recomenda-se que você retorne à presente seção. Você verá que o procedimento descrito acima faz sentido.

4.5.1 O Problema das N Rainhas

Se você não conhece jogo de xadrez, precisa ser informado que a peça denominada *rainha* é aquela mais poderosa, pois lhe é permitido atacar qualquer peça que esteja na mesma coluna, linha ou diagonal na qual ela se encontra, como mostra a **Figura 4–8**.

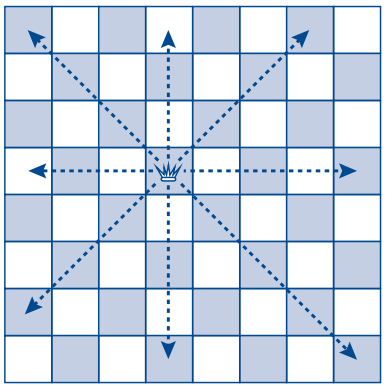

Figura 4–8: Movimentos de uma Rainha num Jogo de Xadrez

Para entender o problema clássico das oito rainhas, você não precisa saber jogar xadrez, damas ou mesmo porrinha. O problema consiste em colocar oito rainhas num tabuleiro de xadrez (que, para quem não sabe, contém exatamente oito linhas e oito colunas), de tal modo que nenhuma rainha seja capaz de atacar outra. Ou seja, uma solução para o problema requer que nenhum par de rainhas ocupe a mesma linha, coluna ou diagonal.

Esse é um problema típico de satisfação de restrições (v. acima) e aqui as restrições consistem no fato de nenhuma rainha poder atacar outra.

Um exemplo de configuração de tabuleiro de xadrez que é solução para o problema em discussão é apresentado na **Figura 4–9**. Note que a disposição das rainhas no tabuleiro dessa figura satisfaz as restrições do problema (i.e., nenhuma rainha é capaz de atacar outra).

O problema original das oito rainhas possui 92 soluções distintas. Mas, se soluções simétricas ou obtidas por meio de rotações do tabuleiro forem unificadas, existem apenas 12 soluções.

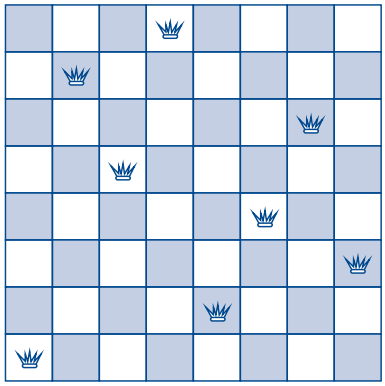

Figura 4–9: Uma Solução para o Problema das Oito Rainhas

Apesar de o problema das oito rainhas ter sido originalmente proposto considerando o tabuleiro de xadrez real (i.e., *8 x 8*), ele tem sido estendido para levar em consideração tabuleiros fictícios com *n* linhas e *n* colunas para posicionamento de *n* rainhas. De fato, essas extensões têm pouca relevância no entendimento do problema e sua solução. Mas, por razões didáticas, vale a pena considerar um (pseudo) tabuleiro *4 x 4*.

Entendendo o Problema

A **Figura 4–10** e a **Figura 4–11** mostram passo a passo como se obtém a primeira solução para o problema reduzido das quatro rainhas. Nessas figuras, uma rainha cortada por um *X* significa uma tentativa frustrada de colocá-la na posição em que se encontra no tabuleiro.

A **Figura 4–10 (a)** mostra o primeiro passo da tentativa de resolução do problema das quatro rainhas. Esse passo é trivial porque, neste instante, não há nenhuma restrição a ser satisfeita. Isto é, ainda não há nenhuma outra rainha a ameaçar a primeira rainha a ser disposta no tabuleiro. Portanto ela pode ser colocada em qualquer posição. Sendo assim, por que não colocá-la na **Coluna 0**, que é a mais óbvia posição?

Na **Figura 4–10 (b)**, ocorrem duas tentativas frustradas de posicionamento da segunda rainha por que esse posicionamento nas duas primeiras colunas da **Linha 1** não satisfaz a restrição do problema, que é o fato de nenhuma rainha poder atacar outra. Na terceira tentativa, consegue-se posicionar a segunda rainha na **Coluna 2** da **Linha 1**.

A **Figura 4–10 (c)** mostra quatro tentativas malsucedidas de posicionamento da terceira rainha na **Linha 2**. Como, nesse caso, esgotam-se as possibilidades de posicionamento dessa rainha, ocorre o primeiro retrocesso na tentativa de resolução do problema. No corrente contexto, retroceder significa desfazer o último posicionamento bem-sucedido e tentar fazer um novo posicionamento. Assim retrocede-se até a **Linha 1** e tenta-se reposicionar a segunda rainha, conforme é mostrado na **Figura 4–10 (d)**.

Na **Figura 4–10 (e)**, ocorrem novas tentativas de relocação da terceira rainha na **Linha 2**, que, enfim, consegue ser posicionada na **Coluna 1** dessa linha.

A **Figura 4–10 (f)** apresenta quatro tentativas frustradas de posicionamento da quarta rainha na **Linha 3**. Essas tentativas fracassadas conduzem a outro retrocesso.

Esse segundo retrocesso é mostrado na **Figura 4–11 (a)**. Nesse caso, tenta-se reposicionar a terceira rainha na **Linha 2**, mas isso não é possível, como mostra essa figura. Então, o retrocesso passa para a **Linha 1**, na qual se encontra a segunda rainha. Ocorre, porém, que essa última tentativa também é frustrada, visto que essa rainha se encontra na última coluna da **Linha 1**. Logo mais um retrocesso se faz necessário.

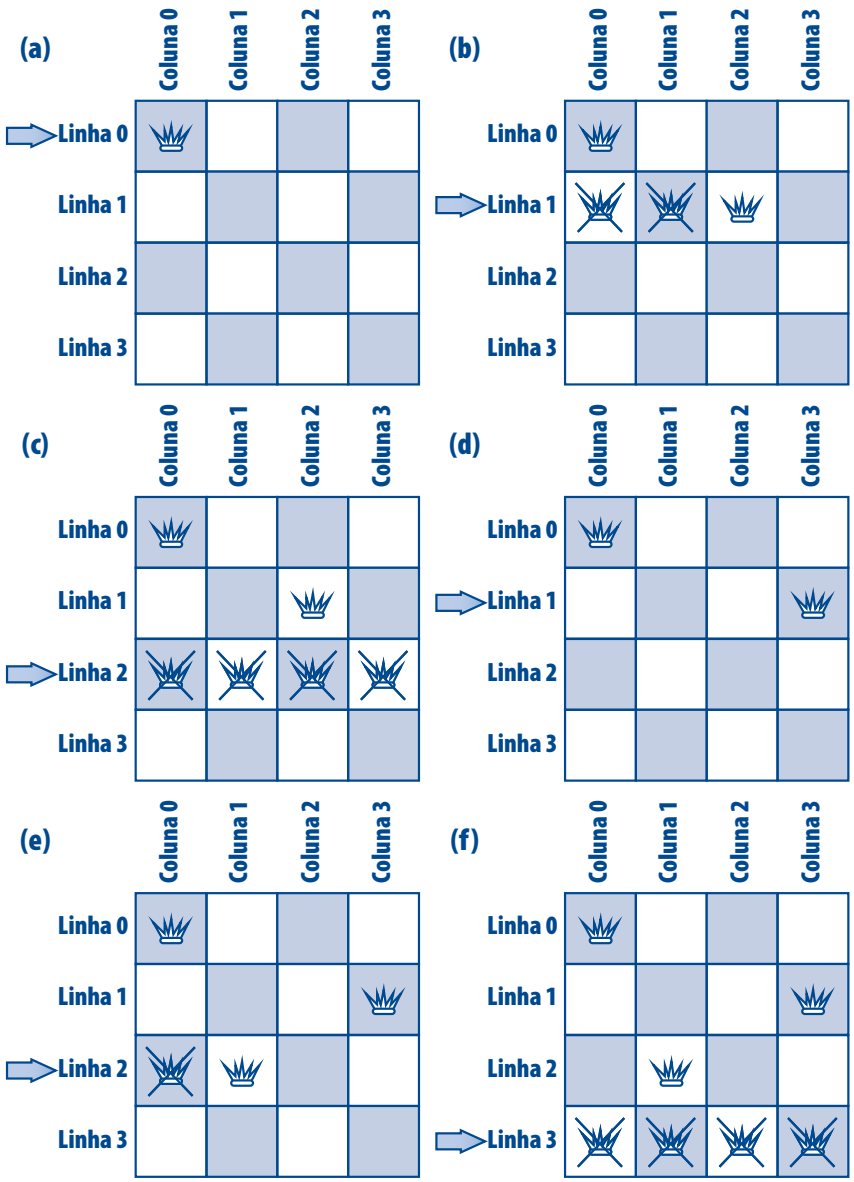

Figura 4–10: Problema das Quatro Rainhas 1

Na **Figura 4–11 (b)**, a primeira rainha é reposicionada na **Coluna 1** da **Linha 0**. É importante observar que essa é a última chance de retrocesso. Quer dizer, se não fosse possível refazer o posicionamento da primeira rainha, não haveria mais nada que pudesse ser feito. Em geral, quando se retrocede até a primeira configuração da solução de um problema de satisfação de restrições e não é possível refazer essa configuração, só há duas possibilidades **(1)** o problema não possui solução ou **(2)** todas as soluções já foram encontradas. Nenhuma dessas possibilidades ocorre no problema sob análise, pois é possível refazer o posicionamento da primeira rainha, como mostra a **Figura 4–11 (b)**.

A **Figura 4–11 (c)** ilustra o posicionamento da segunda rainha na **Coluna 3** da **Linha 1** após o reposicionamento da primeira rainha. Esse posicionamento ocorre após três tentativas frustradas nas colunas **0**, **1** e **2** dessa linha.

A **Figura 4–11 (d)** mostra o posicionamento da terceira rainha na **Linha 2**. Finalmente, a **Figura 4–11 (e)** ilustra o novo posicionamento da quarta rainha na **Coluna 2** da **Linha 3** após duas tentativas malsucedidas.

A **Figura 4–11 (e)** representa uma solução para o problema de posicionamento de quatro rainhas num (pseudo) tabuleiro *4 x 4*. Mas essa não é a única solução para esse problema. De fato, existe outra solução que pode ser obtida retrocedendo-se até o posicionamento da primeira rainha.

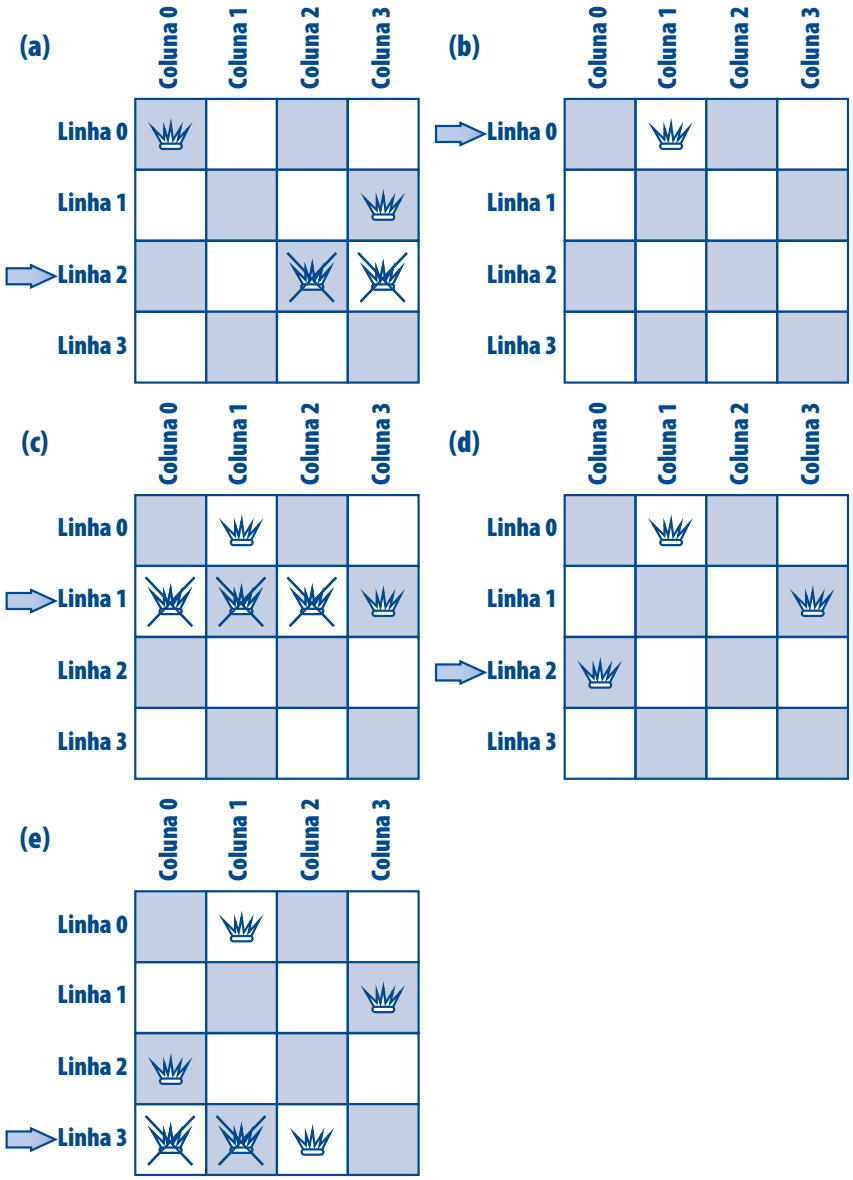

Figura 4–11: Problema das Quatro Rainhas 2

Exercício: Desenhe diagramas semelhantes às figuras apresentadas acima que mostrem como a segunda solução para o problema de posicionamento de quatro rainhas num tabuleiro *4 x 4* pode ser obtida.

É importante entender o funcionamento do mecanismo básico da técnica de retrocesso: o último candidato a solução considerado é o primeiro a ser desconsiderado quando é necessário retroceder. No problema das rainhas examinado acima, se uma rainha for colocada na linha n e não for possível posicionar a próxima rainha na linha $n + 1$, então tenta-se reposicionar a rainha que se encontra posicionada na linha n. Se isso não for possível, tenta-se reposicionar a rainha que se encontra na linha $n - 1$ e assim por diante. Isto é, retrocesso permite retornar a um ponto que oferece outras possibilidades para resolver o problema.

Quando aplicável, retrocesso é bem mais rápido do que qualquer técnica de força bruta na qual todas as possíveis soluções são testadas.

Num problema de posicionamento de *n* rainhas num tabuleiro *n x n*, há n^2 posições no tabuleiro passíveis de ocupação, dentre as quais *n* devem ser escolhidas para acomodar as *n* rainhas. Portanto, quando *n = 4*, há *256* candidatos a solução e, conforme pode ser comprovado examinando-se as figuras acima, para encontrar a primeira solução para o problema das quatro rainhas usando-se retrocesso, foram levados em consideração apenas *27* movimentos.

Implementação da Solução

Aqui, se mostrará como o uso de recursão simplifica a codificação da técnica de retrocesso na resolução do problema das quatro rainhas. Em primeiro lugar, será necessário elaborar um meio para armazenar as diagonais esquerdas e direitas do tabuleiro de modo que se possa indicar quais são as diagonais sob influência de uma dada rainha. Diagonais esquerdas são aquelas que cruzam o tabuleiro da direita para a esquerda e diagonais direitas são aquelas que cruzam o tabuleiro da esquerda para a direita, sendo que, nos dois casos, o cruzamento é efetuado de cima para baixo. Examine a **Figura 4–12** e note que as casas do tabuleiro cruzadas por cada diagonal esquerda apresentam um valor constante para a soma da linha e da coluna que identificam a casa. Por exemplo, a diagonal esquerda superior cruza apenas uma casa para a qual a linha e a coluna valem zero e a soma desses valores também é zero. A próxima diagonal cruza as casas identificadas por linha e coluna como *(0, 1)* e *(1, 0)*. Nesses dois casos, a soma da linha com a coluna é 1. Seguindo esse raciocínio, as diagonais esquerdas podem ser representadas por um array indexado de 0 a 6.

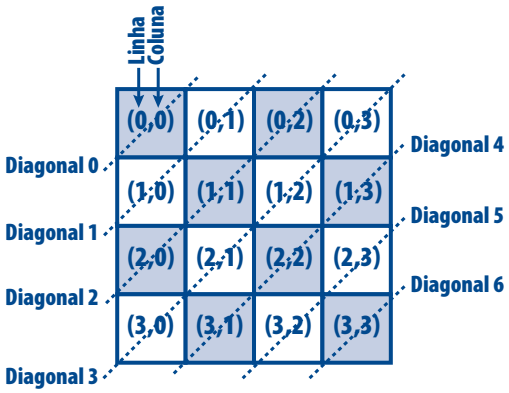

Figura 4–12: Diagonais Esquerdas no Problema das Quatro Rainhas

Agora, examine a **Figura 4–13 (a)** que apresenta as diagonais direitas do tabuleiro. Observe que, nesse caso, a diferença entre os valores que representam colunas e linhas é a mesma para todas as casas cruzadas por uma dada diagonal direita. Por exemplo, para todas as casas cruzadas pela diagonal direita central essa diferença é zero. Portanto essas diagonais podem ser indexadas pelo intervalo *[-3, 3]*. Nesse caso, parece que não será possível representar diagonais direitas por meio de um array como foi realizado para as diagonais esquerdas, visto que não se pode indexar um array com valores negativos. Entretanto, se for somado *3* a cada valor no intervalo *[-3, 3]* obtém-se o intervalo *[0, 6]* que, evidentemente, pode ser utilizado para indexar um array [v. **Figura 4–13 (b)**].

O valor *3* acrescentado ao intervalo de diagonais da **Figura 4–13 (a)** para obter a indexação da **Figura 4–13 (b)** corresponde a *N – 1*, em que *N* representa o número de linhas ou colunas do tabuleiro.

Agora, um array que representa as colunas do tabuleiro também se faz necessário para identificar as colunas sob influência de alguma rainha. Esse array é indexado de *0* a *N – 1*.

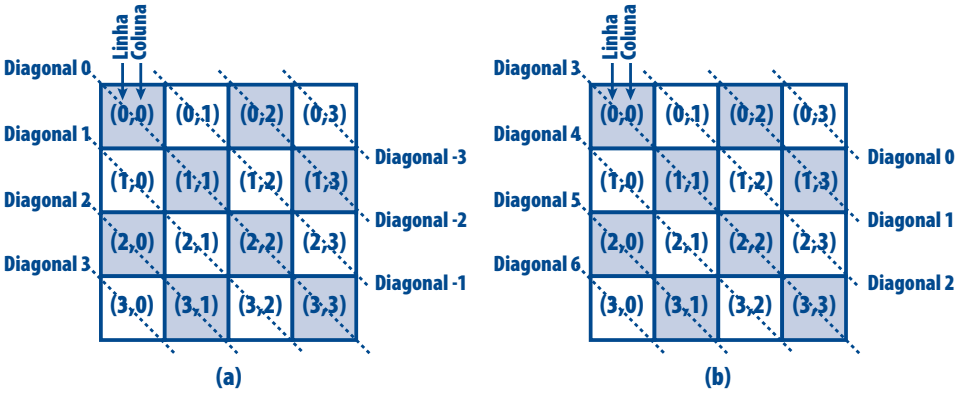

Figura 4–13: Diagonais Direitas no Problema das Quatro Rainhas

Os arrays discutidos acima são definidos na função **main**() do programa como:

```
tDisponibilidade colunas[N] = {DISPONIVEL},
                 diagEsq[2*N - 1] = {DISPONIVEL},
                 diagDir[2*N - 1] = {DISPONIVEL};
```

A constante simbólica `N` e o tipo `tDisponibilidade` usados nessa definição são definidos no início do programa como:

```
#define N 4 /* Número de linhas, colunas ou rainhas */

typedef enum {DISPONIVEL, INDISPONIVEL} tDisponibilidade;
```

As constantes do tipo enumeração `tDisponibilidade` definido acima indicam se uma posição num dos arrays discutidos acima está disponível (i.e., pode acomodar uma rainha) ou não. Lembre-se que, como foi discutido na **Seção 1.13**, a constante `DISPONIVEL` vale `0`, ao passo que a constante `INDISPONIVEL` vale `1`.

A função **main**(), que será apresentada a seguir, apenas inicia o número de soluções com zero e os arrays cujos papéis desempenhados na resolução do problema foram discutidos acima. Quem de fato realiza a tarefa de resolver o problema é a função `Posiciona()`, que será abordada mais adiante.

```
int main(void)
{
   tDisponibilidade colunas[N] = {DISPONIVEL},
                    diagEsq[2*N - 1] = {DISPONIVEL},
                    diagDir[2*N - 1] = {DISPONIVEL};
   int              pLinha[N], /* Posição de rainha numa linha */
                    ns = 0;  /* Número de soluções encontradas */

      /* Passa a bola para a função Posiciona() */
   Posiciona( 0, colunas, diagEsq, diagDir, pLinha, &ns );

   return 0;
}
```

O array `pLinha[]` armazena a posição da rainha em cada linha para cada solução do problema e não precisa ser iniciado na função **main**() (v. adiante). Por sua vez, a variável `ns` representa o número de soluções encontradas e é iniciada com `0`, visto que inicialmente ainda não foi encontrada nenhuma solução. A função **main**() chama a função `Posiciona()`, que é responsável por implementar a solução do problema discutido acima. Os parâmetros dessa última função, que será apresentada adiante são:

- `linha` (entrada) — linha na qual será efetuado o posicionamento
- `colunas[]` (entrada e saída) — array que armazena a disponibilidade em cada coluna

- `diagonalE[]` (entrada e saída) — array que armazena a disponibilidade em cada diagonal esquerda
- `diagonalD[]` (entrada e saída) — array que armazena a disponibilidade em cada diagonal direita
- `pos[]` (saída) — array que armazena a posição da rainha em cada linha
- `*solucoes` (entrada e saída) — número de soluções encontradas

```
void Posiciona( int linha, tDisponibilidade colunas[], tDisponibilidade diagonalE[],
                tDisponibilidade diagonalD[], int pos[], int *solucoes )
{
   int coluna;

      /* Procura uma coluna para a rainha na linha recebida como parâmetro */
   for (coluna = 0; coluna < N; coluna++) {
         /* Verifica se a coluna corrente está disponível */
      if ( colunas[coluna] == DISPONIVEL &&
           diagonalE [linha + coluna] == DISPONIVEL &&
           diagonalD[linha - coluna + N - 1] == DISPONIVEL) {
            /* Encontrada uma coluna disponível e */
            /* a rainha é colocada nessa coluna   */
         pos[linha] = coluna;

            /* Essa coluna passa a estar indisponível e o mesmo ocorre com */
            /* as diagonais que cruzam a mesma coluna e a mesma linha      */
         colunas[coluna] = INDISPONIVEL;
         diagonalE[linha + coluna] = INDISPONIVEL;
         diagonalD[linha - coluna + N - 1] = INDISPONIVEL;

            /* Se a linha corrente não for a última,       */
            /* posiciona uma rainha na próxima linha. Caso */
            /* contrário, apresenta a solução corrente.    */
         if (linha < N - 1) {
               /* Posiciona a rainha na próxima linha */
            Posiciona( linha + 1, colunas, diagonalE, diagonalD, pos, solucoes );
         } else { /* Essa era a última linha. Portanto */
                  /* foi encontrada mais uma solução.  */
               /* Incrementa o número de soluções encontradas até */
               /* aqui e apresenta a solução corrente             */
            (*solucoes)++;
            ApresentaSolucao(pos, *solucoes);
         }

            /* Torna a presente coluna e as diagonais que a cruzam */
            /* disponíveis novamente para que ocorra retrocesso    */
         colunas[coluna] = DISPONIVEL;
         diagonalE[linha + coluna] = DISPONIVEL;
         diagonalD[linha - coluna + N - 1] = DISPONIVEL;
      } /* if */
   } /*  for */
}
```

A função `Posiciona()` detém o papel principal na resolução do problema das quatro rainhas. Ela é crucial na compreensão da técnica de retrocesso, mas, ao mesmo tempo, não é tão facilmente entendida devido ao fato de alguns detalhes importantes serem ocultos por chamadas recursivas. Essa função será discutida mais adiante, de modo que, por enquanto, observe outros detalhes menos importantes do programa.

A função `ApresentaSolucao()` complementa ao programa e é apenas uma função de apresentação na tela de um tabuleiro que representa uma solução para o problema das `N` rainhas. O parâmetro `posEmLinha[]` dessa

função é um array contendo a posição das rainhas em cada linha, enquanto `nSolucao` é o número da de ordem solução.

```
void ApresentaSolucao(const int posEmLinha[], int nSolucao)
{
   int i, j;

      /* Apresenta o número da solução */
   printf("\nSolucao %d:\n\n", nSolucao);

      /***********************/
      /* Desenha o tabuleiro */
      /***********************/

   for(i = 0; i < N; ++i) {  /* Desenha cada linha */
         /* Apresenta cada coluna vazia antes da rainha */
      for(j = 0; j < posEmLinha[i]; ++j)
         printf("\t-");

      printf("\tR"); /* Apresenta a rainha da linha */

         /* Apresenta cada coluna vazia depois da rainha */
      for(j = posEmLinha[i] + 1; j < N; ++j)
         printf("\t-");

      printf("\n"); /* Passa para a próxima linha */
   }
}
```

Resultado de execução do programa:

```
Solucao 1:

        -       R       -       -
        -       -       -       R
        R       -       -       -
        -       -       R       -

Solucao 2:

        -       -       R       -
        R       -       -       -
        -       -       -       R
        -       R       -       -
```

A seguir, a função `Posiciona()` será escrutinada para que você possa, de fato, entender o seu funcionamento bem como aquilo que diz respeito à técnica de retrocesso em si. Para completo entendimento do processo, é interessante que se leve em consideração os diversos estados da pilha de execução do programa.

A **Figura 4–14** **(a)** mostra o status da pilha de execução do programa no instante em que é efetuada a primeira chamada da função `Posiciona()` no corpo da função **main()**. Note que, nessa figura e nas demais figuras que serão apresentadas, os endereços de retorno das funções são omitidos porque eles não interessam à discussão em questão. Observe ainda que o array `pLinha[]`, que armazenará a solução do problema, não é iniciado na função **main()** e essa é a razão pela qual aparecem pontos de interrogação como valores de seus elementos. De fato, essa iniciação não tem importância, visto que esse array será utilizado pela função `Posiciona()` como parâmetro de saída apenas. Por outro lado, todos os elementos dos arrays `colunas[]`, `diagEsq[]` e `diagDir[]` são iniciados com zero, que é o valor da constante de enumeração `DISPONIVEL` (v. acima).

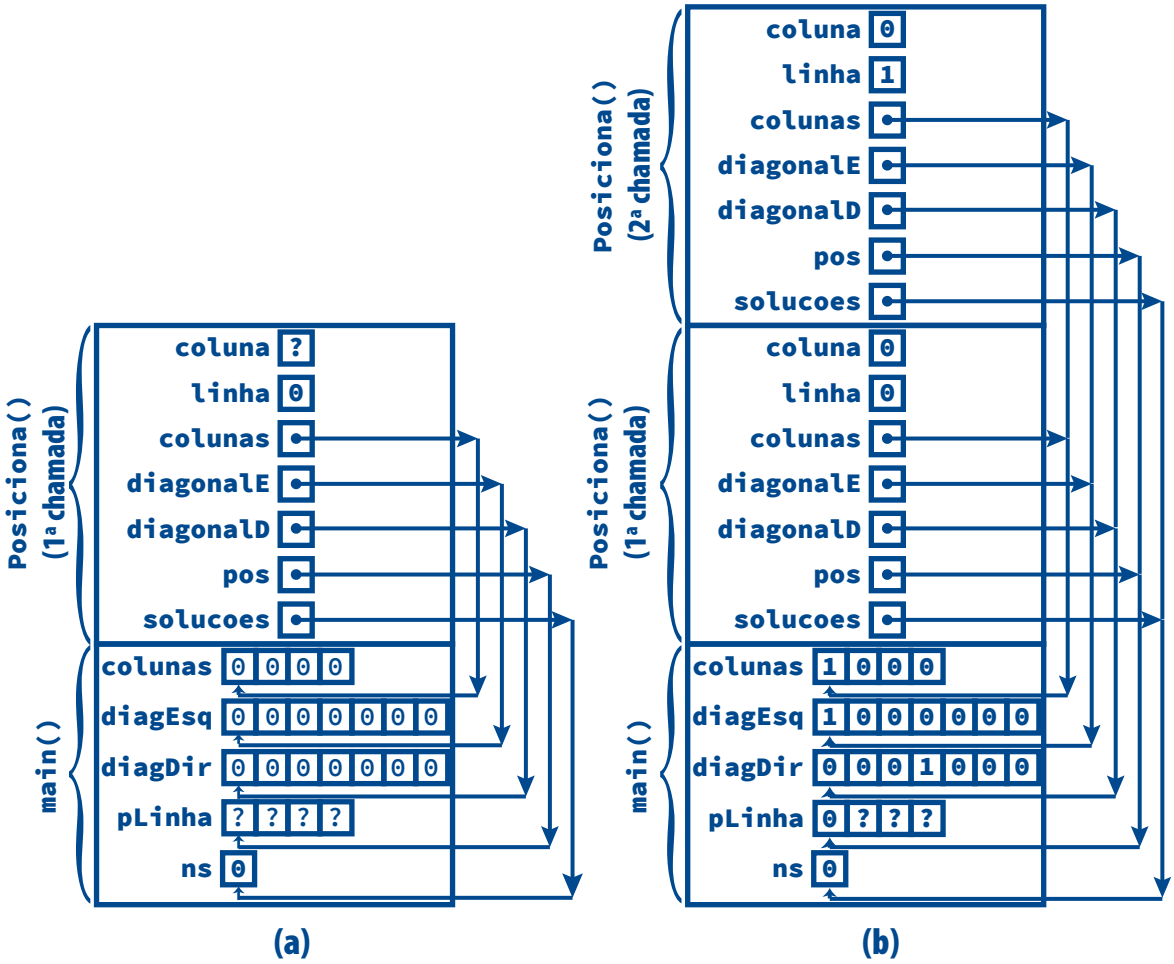

Figura 4–14: Problema das Quatro Rainhas: Pilha de Execução 1

Na **Figura 4–14 (b)**, ocorre a primeira chamada recursiva da função `Posiciona()` (que, obviamente, é a segunda chamada dessa função). Essa chamada acontece após o posicionamento da rainha na primeira coluna da linha. Observe as alterações efetuadas nos arrays que se encontram armazenados no registro de ativação da função **main**().

A **Figura 4–15 (a)** mostra a configuração da pilha de execução logo após a terceira chamada da função `Posiciona()` (segunda chamada recursiva). Antes dessa chamada, a segunda rainha foi posicionada, conforme se pode constatar examinado as alterações nos arrays no registro de ativação de **main**().

Na **Figura 4–15 (b)**, após quatro tentativas malsucedidas de posicionamento da terceira rainha, a terceira chamada da função `Posiciona()` é encerrada com o consequente desempilhamento de seu registro de ativação. Em seguida, ocorre o primeiro retrocesso.

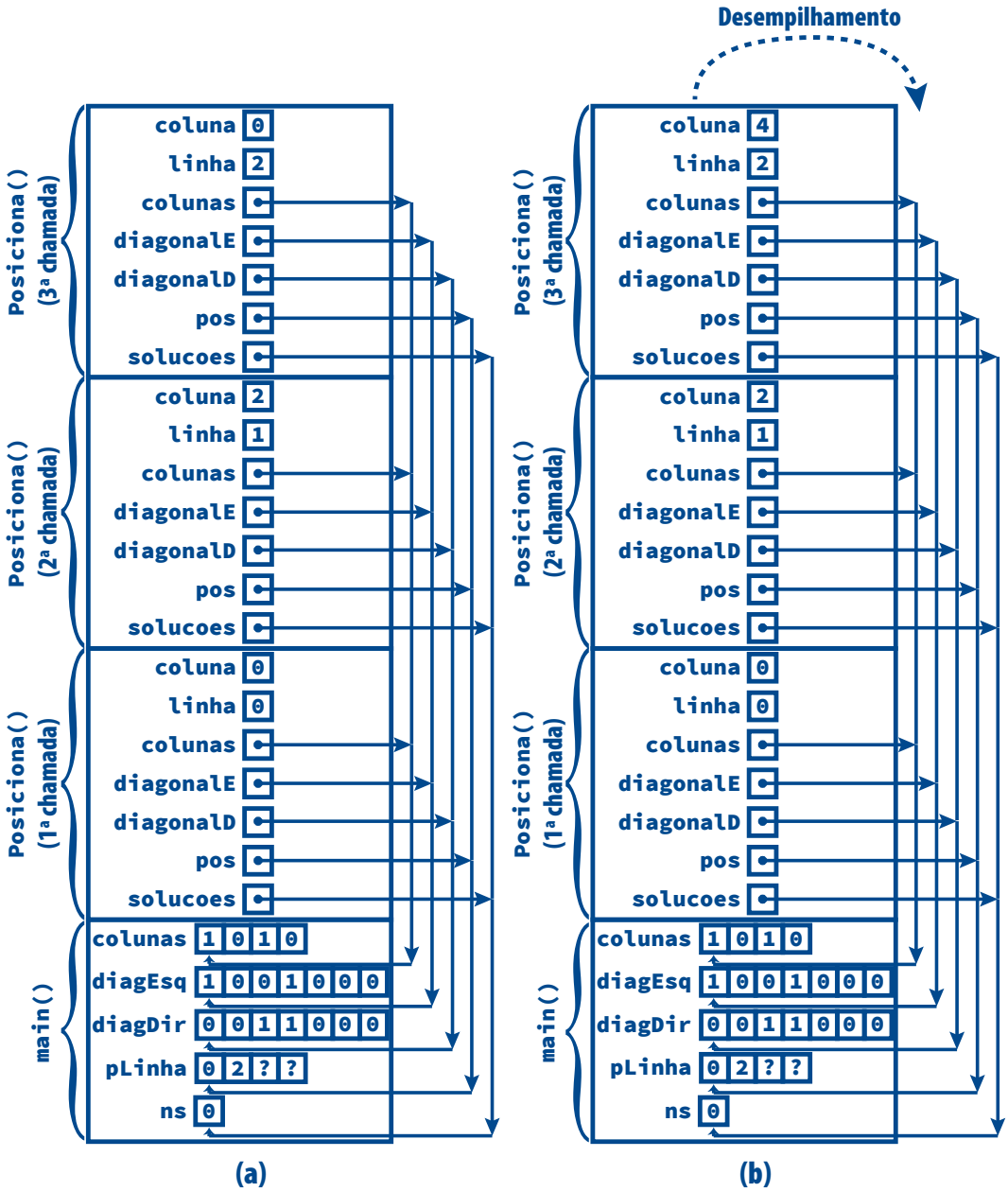

Figura 4–15: Problema das Quatro Rainhas: Pilha de Execução 2

A **Figura 4–16 (a)** apresenta a execução do retrocesso promovido pela segunda chamada da função `Posiciona()`. O primeiro passo desse retrocesso consiste em desfazer as últimas alterações efetuadas nos arrays `colunas[]`, `diagEsq[]` e `diagDir[]`. Essas novas alterações fazem com que a rainha que se encontrava na coluna 2 da linha 1 seja removida dessa posição. As três últimas instruções da função são responsáveis pelo referido desfazimento. Em seguida, essa rainha é reposicionada na coluna 3 dessa mesma linha. Novamente, os referidos arrays mais o array `pLinha[]` são alterados para refletir esse novo posicionamento.

Na **Figura 4–16 (b)**, a função `Posiciona()` é novamente chamada (quarta chamada). Essa chamada é responsável pelo posicionamento da terceira rainha na coluna 1 da linha 2, como se pode constatar nas alterações dos

arrays `colunas[]`, `diagEsq[]` e `diagDir[]` mostradas na **Figura 4–17 (a)**. Essa última figura mostra ainda que mais uma chamada de `Posiciona()` (quinta chamada) é levada a efeito. Na **Figura 4–17 (b)**, ocorre o retorno dessa última chamada com subsequente desempilhamento do seu registro de ativação. Esse retorno é devido às tentativas frustradas de alocação da quarta rainha na linha 3.

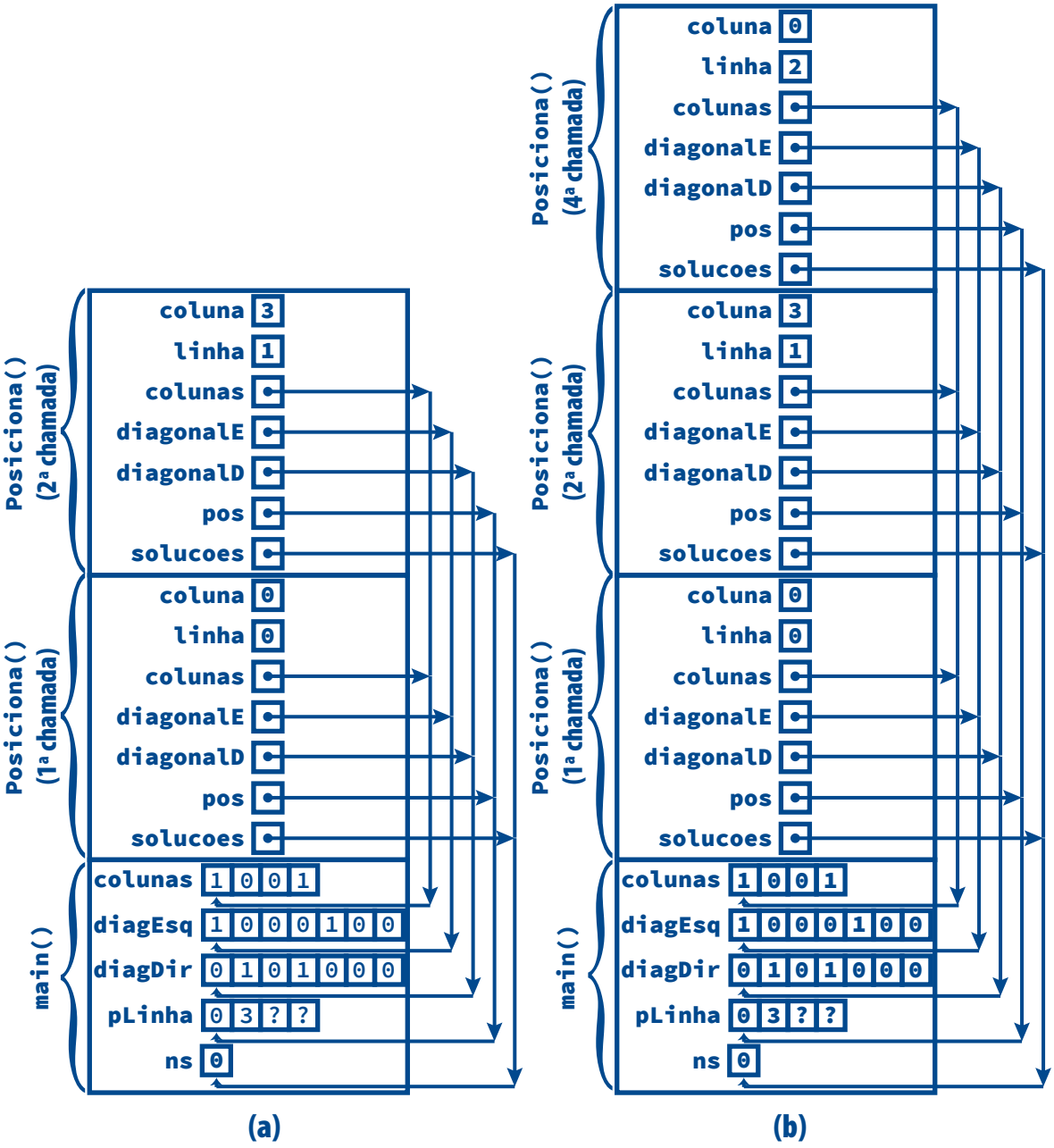

Figura 4–16: Problema das Quatro Rainhas: Pilha de Execução 3

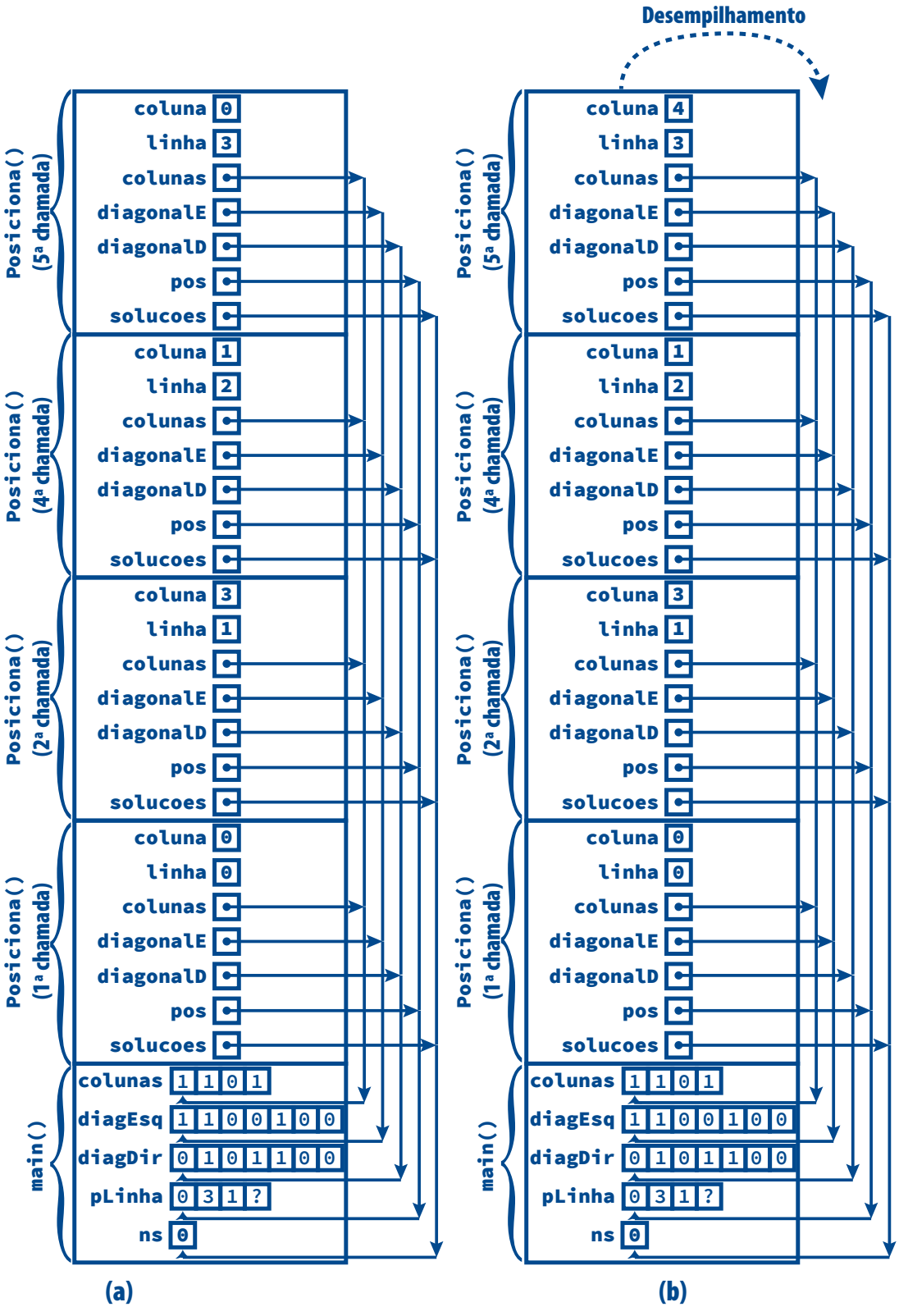

FIGURA 4–17: PROBLEMA DAS QUATRO RAINHAS: PILHA DE EXECUÇÃO 4

Na **Figura 4–18 (a)**, ocorre o segundo retrocesso levado a efeito pela quarta chamada de `Posiciona()`. Nesse retrocesso, são desfeitas as alterações efetuadas por essa chamada nos arrays `colunas[]`, `diagEsq[]` e `diagDir[]`. Ocorre, porém, que essa chamada não consegue realocar a terceira rainha na linha 2. Assim ocorre o retorno dessa chamada com o respectivo desempilhamento do registro de ativação, como mostra a **Figura 4–18 (a)**.

A **Figura 4–18 (b)** mostra o terceiro retrocesso, desta vez promovido pela segunda chamada de `Posiciona()`. Nesse retrocesso, essa chamada de função desfaz as alterações que ela havia efetuado nos arrays `colunas[]`, `diagEsq[]` e `diagDir[]`. Entretanto, esse retrocesso não é capaz de reposicionar a rainha que se encontra na coluna 3 da linha 1. Desse modo, essa chamada retorna e ocorre o desempilhamento do seu registro de ativação.

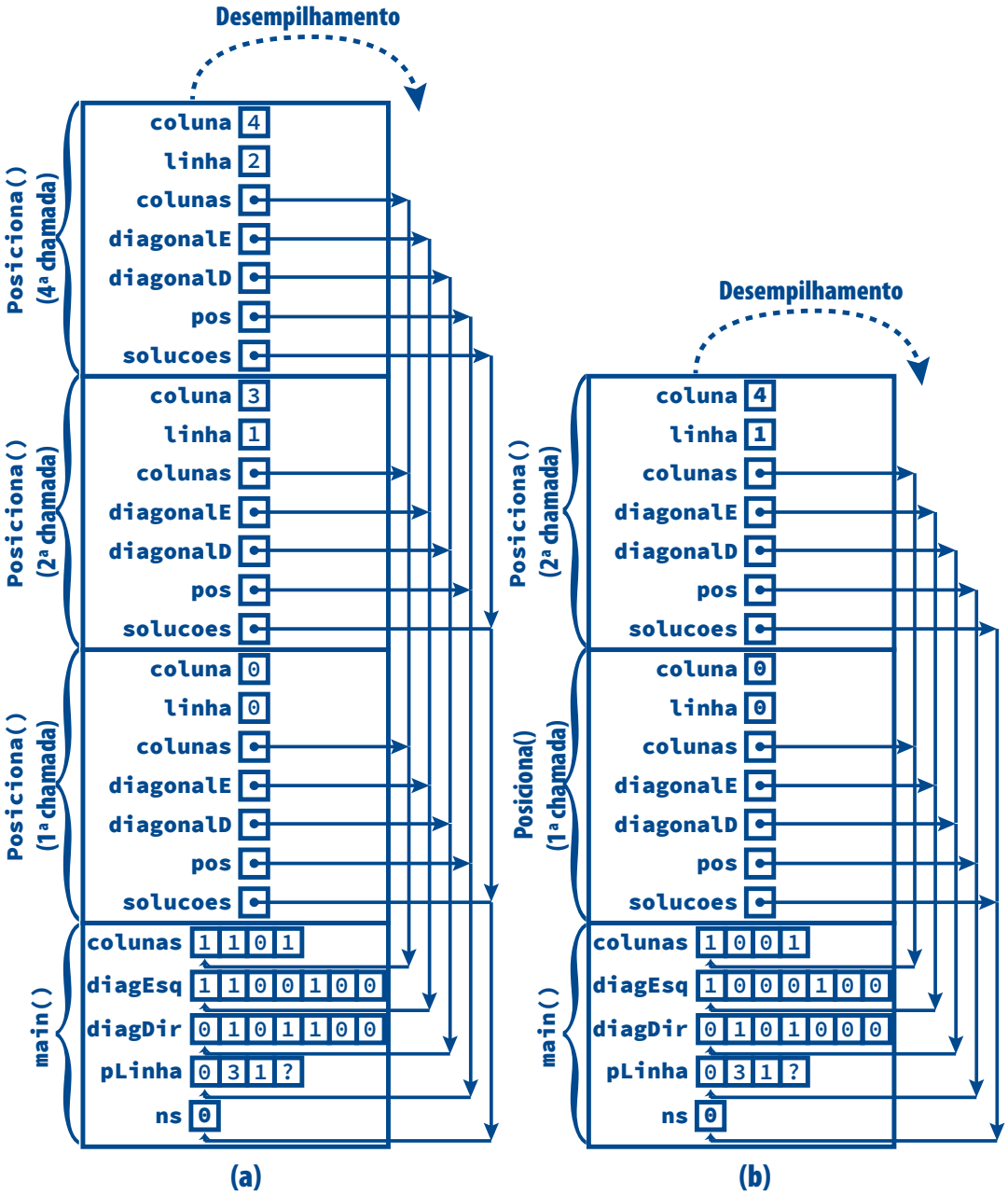

Figura 4–18: Problema das Quatro Rainhas: Pilha de Execução 5

A **Figura 4–19 (a)** ilustra mais um retrocesso. Dessa vez, a primeira chamada de é responsável por sua execução. Primeiro, as três últimas instruções dessa chamada de função desfazem o último posicionamento da primeira rainha. Em seguida, essa chamada é bem-sucedida na tentativa de realocar essa rainha, de modo que ela passa a ocupar a coluna 1 da linha 0. Finalmente, ocorre a sexta chamada da função `Posiciona()` [v. **Figura 4–19 (b)**].

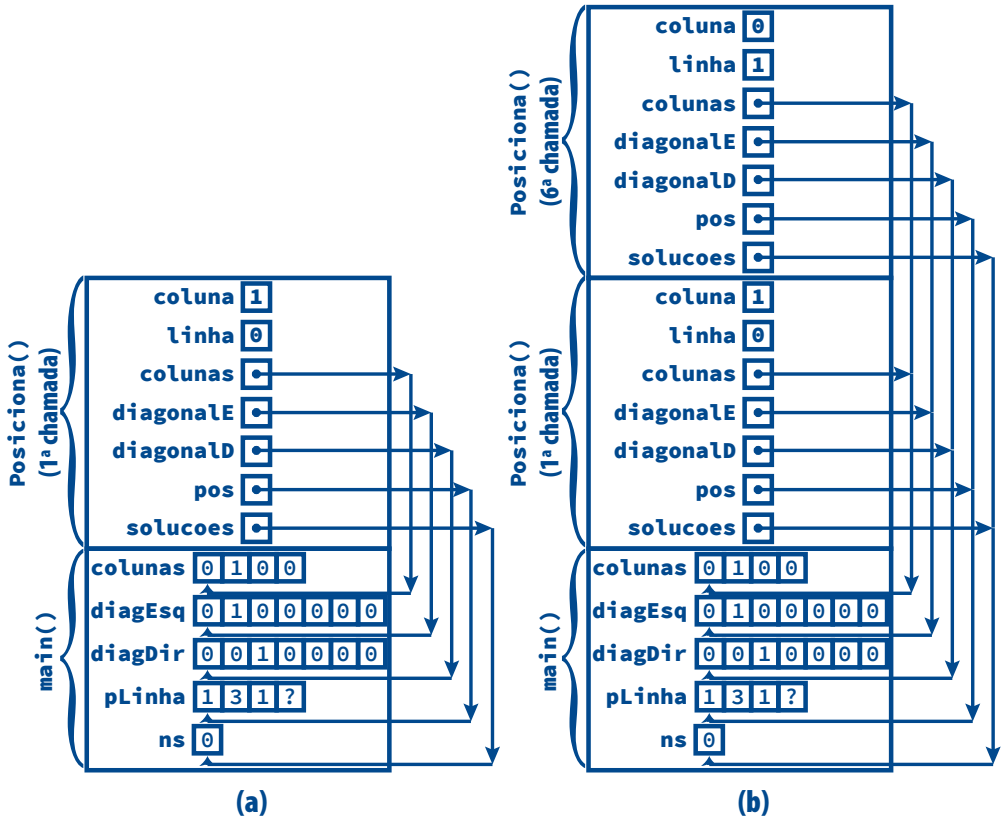

FIGURA 4–19: PROBLEMA DAS QUATRO RAINHAS: PILHA DE EXECUÇÃO 6

A **Figura 4–20 (a)** mostra o instante em que a sexta chamada da função `Posiciona()` efetua a sétima chamada dessa função, após ter posicionado a segunda rainha na coluna 3 da linha 1. Observe as alterações nos arrays armazenados no registro de ativação da função **main()**. Na **Figura 4–20 (b)**, após posicionar a terceira rainha na coluna 0 da linha 2, a sétima chamada de `Posiciona()` realiza a oitava chamada dessa função.

A **Figura 4–21 (a)** mostra a situação logo após a oitava chamada de `Posiciona()` posicionar a quarta rainha na coluna 2 da linha 3. Por outro lado, a **Figura 4–21 (b)** mostra essa chamada prestes a retornar. Observe que, diferentemente do que ocorreu nas chamadas anteriores nas quais a parte **if** da segunda instrução **if-else** foi executada, agora a parte **else** dessas instrução é executada, de modo que é incrementado o conteúdo apontado pelo parâmetro `solucoes`, que corresponde ao conteúdo da variável `ns` definida na função **main()**. Esse fato indica que foi encontrada a primeira solução para o problema, o que realmente aconteceu. Além disso, a função `ApresentaSolucao()` é invocada para exibir essa solução na tela.

A **Figura 4–21 (b)** mostra ainda que após o retorno da oitava chamada de `Posiciona()`, a sétima e a sexta chamadas dessa função também irão retornar (nessa ordem). Mas, antes que ocorra o retorno, ocorre retrocesso em cada uma delas. Cada função tenta reposicionar a rainha na linha sob responsabilidade da respectiva função. É interessante lembrar ainda que, no retrocesso, cada função desfaz a alteração que efetuou nos arrays `colunas[]`, `diagEsq[]` e `diagDir[]`.

Após os desempilhamentos mostrados na **Figura 4–21 (b)**, a configuração passa a ser aquela ilustrada na **Figura 4–22**. A partir daí até que seja encontrada a segunda e última solução para o problema, a história apresentada aqui se repetirá.

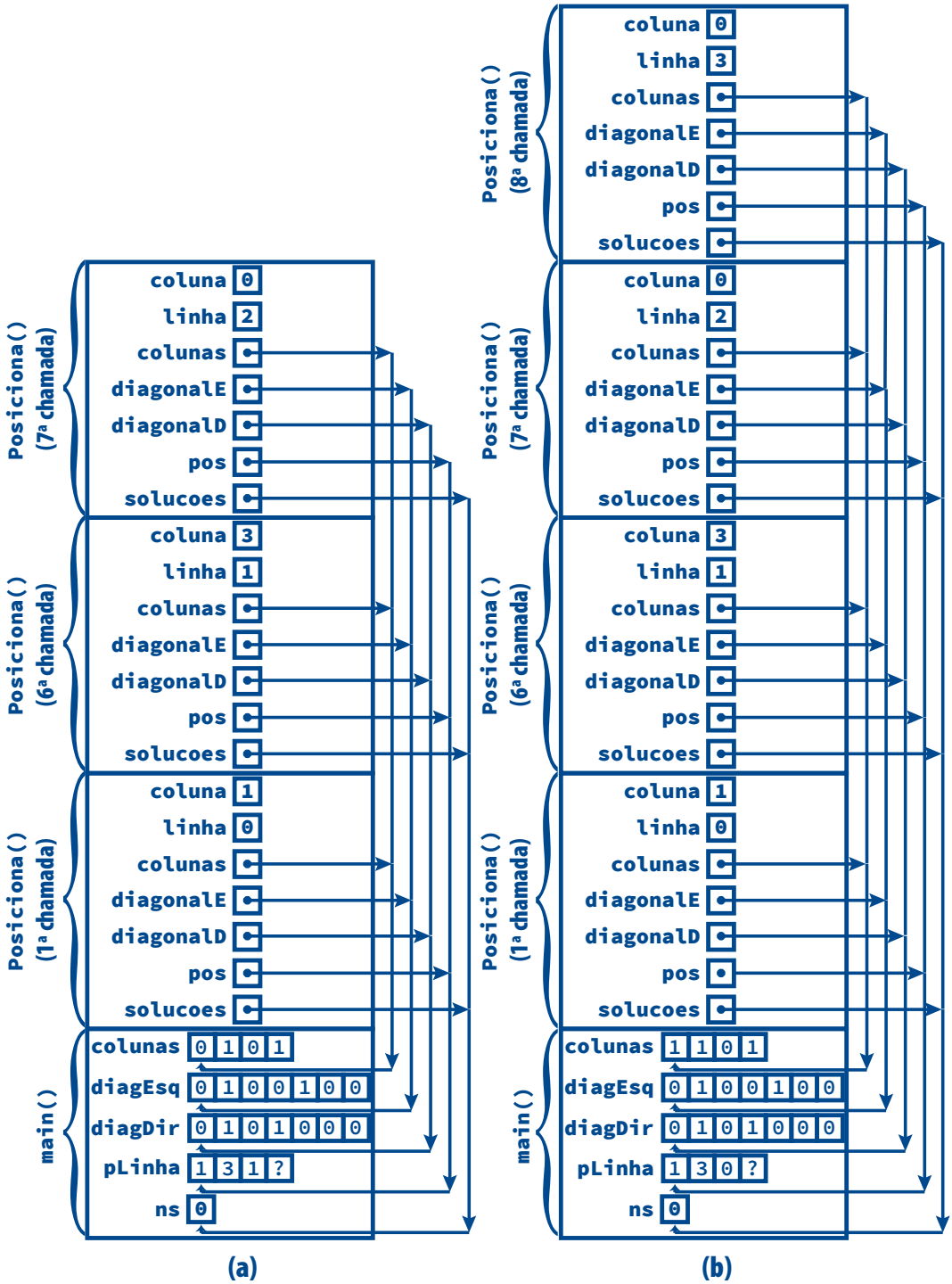

Figura 4–20: Problema das Quatro Rainhas: Pilha de Execução 7

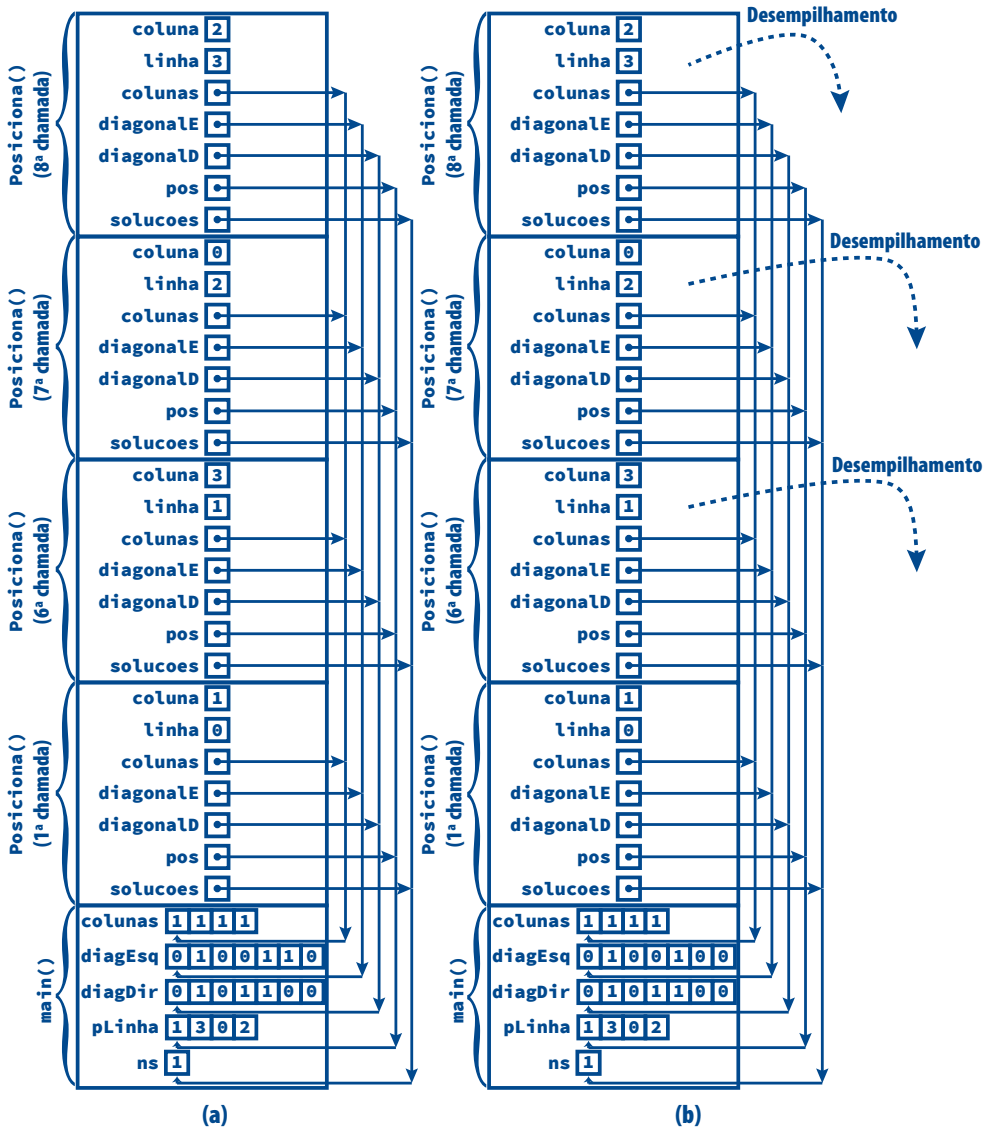

FIGURA 4–21: PROBLEMA DAS QUATRO RAINHAS: PILHA DE EXECUÇÃO 8

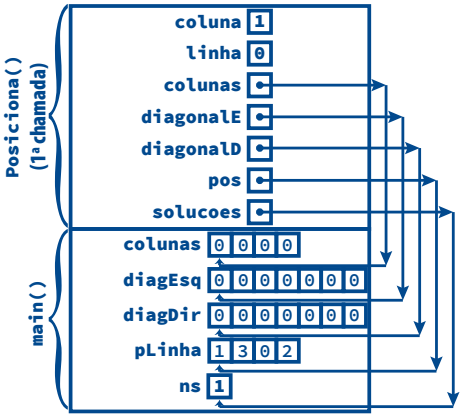

FIGURA 4–22: PROBLEMA DAS QUATRO RAINHAS: PILHA DE EXECUÇÃO 9

4.5.2 Outros Problemas Propícios ao Uso de Retrocesso

Conforme foi antecipado, retrocesso é especialmente adequado para resolução de **problemas de satisfação de restrições**. Um problema dessa natureza possui os seguintes componentes:

- Um **conjunto de variáveis**. O problema das oito rainhas, discutido na **Seção 4.5.1**, é um exemplo de problema de satisfação de restrições. Nesse exemplo, as variáveis são exatamente as oito rainhas.
- Um **domínio** para cada variável. No problema das oito rainhas, o domínio é o mesmo para cada variável e consiste no conjunto de pares *(i, j)*, em que *i* e *j* são inteiros e $1 \leq i, j \leq 8$. Esse domínio corresponde a todas as casas de um tabuleiro de xadrez, com linhas e colunas indexadas de *1* a *8*.
- Um **conjunto de restrições**. No problema das oito rainhas, a única restrição é que nenhuma rainha (i.e., variável) pode ser posicionada numa casa de modo que qualquer outra rainha seja ameaçada. Ou mais precisamente, uma variável não pode assumir um valor *(i, j)* tal que qualquer outra variável assuma um valor (i, k), (l, j), $(i - m, j + m)$, $(i + m, j - m)$, $(i - m, j - m)$ ou $(i + m, j + m)$, em que: $1 \leq k = 8$, $1 \leq l \leq 8$, $1 - i \leq m \leq 8 - i$ e $1 - j \leq m \leq 8 - j$.

Se você não entendeu o formalismo apresentado acima, não se preocupe, pois ele não é essencial para entender a técnica de retrocesso, que foi explorada em detalhes na **Seção 4.5.1**.

Alguns exemplos de problemas de satisfação de restrições são apresentados abaixo:

- Problema das oito (ou *N*) rainhas (v. **Seção 4.5.1**)
- Problemas de Sudoku (v. **Seção 4.8.8**)
- Criptoaritmética
- Problemas envolvendo grafos
- Problemas que aparecem com frequência na área de Inteligência Artificial

Retrocesso é também um mecanismo de resolução de problemas incorporado em linguagens de programação lógica, como, por exemplo, Prolog.

4.6 Como Pensar Recursivamente

Em cursos introdutórios de programação, você certamente estudou a abordagem básica de resolução de problemas denominada dividir e conquistar. Utilizando essa abordagem, divide-se sucessivamente um problema em subproblemas menores até que cada um deles possa ser resolvido por meio de um subprograma (i.e., função em C). As soluções para os subproblemas são então combinadas de modo a resultar na solução para o problema original. Essa técnica de resolução de problemas também é conhecida como **método de refinamentos sucessivos**.

O raciocínio básico que deve ser utilizado na criação de funções recursivas é o mesmo que norteia a abordagem de refinamentos sucessivos. A diferença diz respeito a como essa abordagem é usada em iteração e em recursão. Na resolução de problemas usando iteração são criadas funções para resolver os subproblemas resultantes da proposta da abordagem. No caso de recursão uma mesma função chamada recursivamente lida com os respectivos subproblemas.

Para adaptar a abordagem de refinamentos sucessivos para recursão siga os seguintes passos:

1. Obtenha uma descrição precisa do problema que a função irá resolver, como sugerido na **Seção 5.3.3**. Em especial, determine o tamanho do problema, que deverá ser representado por um ou mais parâmetros da função.

2. Divida o problema em questão em um ou mais subproblemas e escreva chamadas recursivas da função que resolvam esses subproblemas. Ou seja, resolva o caso geral do problema, identificado no **Passo 1**, em termos de casos menores (i.e., valores menores do tamanho do problema).
3. Descubra quais são os casos base; i.e., aqueles subproblemas que podem ser resolvidos sem uso de recursão. Certifique-se que esses casos base serão infalivelmente atingidos.

É importante salientar que, às vezes, ao criar uma função recursiva, é necessário acrescentar ou um ou mais parâmetros que não existiriam se a função fosse iterativa em vez de recursiva. É possível ainda que uma função iterativa tenha um ou mais parâmetros substituídos em sua versão recursiva.

Por exemplo, a função `EmArray()`, apresentada a seguir, verifica se um valor do tipo **int** faz parte de um array de elementos desse tipo. Essa função retorna o índice do referido valor se ele for encontrado no array ou `-1` em caso contrário e seus parâmetros são:

- `ar[]` (entrada) — o array que será pesquisado
- `tam` (entrada) — número de elementos do array
- `num` (entrada) — o número que será procurado

```
int EmArray(const int ar[], int tam, int num)
{
   int i;

      /* Compara cada elemento do array com o parâmetro 'num'. Se for */
      /* encontrado um elemento igual a 'num', retorna seu índice.    */
   for (i = 0; i < tam; ++i)
      if (ar[i] == num)
         return i; /* Encontrado um elemento igual */

   return -1; /* Valor recebido não foi encontrado no array */
}
```

A função recursiva `EmArrayRec()`, que será apresentada adiante, tem a mesma especificação de retorno da função `EmArray()` e usa os seguintes parâmetros:

- `ar[]` (entrada) — o array que será pesquisado
- `tam` (entrada) — número de elementos do array
- `inf` (entrada) — índice inferior do array
- `sup` (entrada) — índice superior do array

```
int EmArrayRec(const int ar[], int inf, int sup, int num)
{
      /* Verifica os casos base */
   if (ar[inf] == num)  /* Caso base 1 */
         /* O valor procurado é o primeiro do array */
      return inf; /* Encontrado um elemento igual a 'num' */
   else if (inf >= sup)  /* Caso base 2 */
         /* Quando o índice inferior do array é maior do que ou igual */
         /* ao seu índice superior, todo o array já foi examinado     */
      return -1; /* Elemento não foi encontrado */

      /* Caso recursivo: procura no restante do array */
   return EmArrayRec(ar, inf + 1, sup, num);
}
```

As funções `EmArray()` e `EmArrayRec()` são funcionalmente equivalentes, já que ambas efetuam a mesma tarefa. Essas funções também são equivalentes em termos de custo temporal, embora esse não seja o caso em termos de custo espacial (v. **Capítulo 6**). Quer dizer, espera-se que ambas as funções sejam executadas, aproximadamente,

no mesmo intervalo de tempo quando o tamanho do array se tornar muito grande. Mas, à medida que o tamanho do array cresce, o espaço consumido pela versão recursiva se torna bem maior do que aquele utilizado pela versão interativa por causa do correspondente aumento no número de registros de ativação necessários para executar a função recursiva (v. **Seção 4.3**).

Comparando-se os protótipos das duas funções, nota-se que o parâmetro `tam`, que representa o tamanho do array na versão iterativa, foi trocado pelos parâmetros `inf` e `sup`, que representam os índices inicial e final do array na versão recursiva. Esses últimos parâmetros são realmente necessários para implementar uma função recursiva com as mesmas especificações da função `EmArray()`.

Agora, o protótipo da função `EmArrayRec()` deve parecer muito bizarro para qualquer programador de C que exiba um mínimo de experiência. Afinal, por que utilizar como parâmetros os índices inicial e final de um array quando o mais natural seria usar apenas o tamanho do array? Para ocultar essas esquisitices apresentadas por algumas funções recursivas, como `EmArrayRec()`, costumam-se usar **funções acionadoras**. Uma função acionadora funciona como intermediária e, tipicamente, contém apenas uma instrução, que é uma chamada da função *esquisita*, como mostra a função `EmArray2()` a seguir.

```
int EmArray2(const int ar[], int tam, int num)
{
   return EmArrayRec(ar, 0, tam - 1, num);
}
```

4.7 Quando Usar (e Não Usar) Recursão

Quando usada adequadamente, recursão pode simplificar a solução de um problema, resultando em programas mais curtos e mais fáceis de entender.

Não custa repetir que muitos problemas (talvez a maioria deles) apresentados como exemplos ou solicitados como exercícios de programação neste e em outros livros de programação são melhor resolvidos sem o uso de recursão. Até o presente ponto deste livro, apenas o problema das *n* rainhas é propício ao uso de recursão.

Então, a pergunta mais óbvia a ser levantada neste instante seria: *se a maioria desses exemplos não deve ser seguida na prática, por que eles aparecem com tanta frequência em textos de programação?* Pode não parecer, mas a resposta a essa questão também é óbvia: *esses textos são dedicados ao ensino de programação e a principal motivação deles é (ou deveria ser) didática*. Mais precisamente, esses exemplos são escolhidos pela facilidade de entendimento que eles apresentam.

A função `Fatorial()` apresentada abaixo é um dos exemplos favoritos em muitos textos de programação:

```
int Fatorial(int n)
{
   if (!n)
      return 1; /* Caso terminal */
   return n*Fatorial(n - 1); /* Caso recursivo */
}
```

Essa função é frequentemente utilizada como primeiro exemplo de recursão em muitos textos porque ela naturalmente implementa a definição matemática de fatorial que muitas vezes é apresentada de forma recursiva:

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

Comparando-se a implementação da função com a definição de fatorial acima, nota-se que a função realmente reflete a definição matemática.

Acontece que o fatorial de um número também pode ser definido matematicamente sem o uso de recursão como:

$$n! = \begin{cases} 1, & se\ n = 0 \\ 1.2.3. \ldots .(n-1).n & se\ n > 0 \end{cases}$$

Agora, com um pouco de conhecimento de programação nota-se que essa última definição de fatorial é mais propícia a ser implementada como uma função iterativa, como a função `Fatorial2()` abaixo:

```
int Fatorial2(int n)
{
   int i, fat = 1;

   if (n < 0)
      return -1; /* Erro de domínio */

   for (i = 1; i <= n; ++i)
      fat *= i;

   return fat;
}
```

A pergunta agora é: *qual das duas funções que calculam fatorial é a melhor?* Para responder essa pergunta as duas funções precisam ser analisadas de acordo com três critérios:

- **Eficiência**. Claramente, a função iterativa é mais eficiente do a versão recursiva, conforme já foi discutido acima.
- **Clareza** (ou **legibilidade**). No presente caso, esse critério é um tanto subjetivo, pois ambas as implementações aparentam ser bastante fáceis de entender.
- **Funcionalidade**. Aparentemente, a função iterativa leva vantagem sobre a função recursiva porque existe a possibilidade de ocorrência de aborto de programa devido a esgotamento de pilha (*stack overflow*) se o parâmetro real recebido pela função recursiva for excessivamente grande. Mas, de fato, essa aparente desvantagem da função recursiva é uma vantagem, pois se ocorrer aborto de programa, pelo menos o programador notará que há algo de errado. O problema é que as duas funções padecem de um mal maior: *overflow* de inteiro. Quer dizer, o fatorial de `n` resulta em valores muito grandes mesmo para valores relativamente pequenos de `n`. Por exemplo, se o tipo **long long int**, que é o maior tipo inteiro primitivo de C, for utilizado em substituição ao tipo **int** em qualquer das duas funções que calculam fatorial, o maior número para o qual o fatorial pode ser calculado sem ocorrência de *overflow* de inteiro é `20`.

A principal vantagem advinda do uso de recursão é que essa técnica pode reduzir consideravelmente o custo de implementação de um programa. E, como o custo de recursos computacionais (i.e., tempo de processamento e memória) tem cada vez mais diminuído, enquanto o custo de tempo de programação tem aumentado, vale a pena considerar implementações recursivas.

A **Tabela 4–1** a seguir compara iteração e recursão e pode servir como guia para decidir se o uso de recursão é aceitável numa dada situação.

Iteração	Recursão
Usa apenas laços de repetição	Usa espaço adicional na pilha de execução
Execução mais rápida	Execução mais lenta
Entendimento mais difícil	Entendimento mais fácil

Tabela 4–1: Iteração versus Recursão

A seguir, serão apresentadas algumas recomendações práticas para uso de recursão:

- O uso de recursão é recomendado quando ela não é muito profunda; i.e., quando ela não requer a criação de um número demasiadamente grande de registros de ativação (v. **Seção 4.3**) para a obtenção de resultados relativamente simples. Por exemplo, um algoritmo que satisfaz bem esse requisito é o algoritmo de busca binária (v. **Seção 6.11.4** e **Seção 7.2.3**). Por outro lado, o cálculo recursivo de números de Fibonacci resulta numa recursão muito profunda na qual muitos valores são calculados repetidamente (v. **Seção 4.8.2**).
- O uso de recursão é recomendado quando a versão recursiva de um algoritmo apresenta aproximadamente o mesmo custo temporal (v. **Capítulo 6**) de uma versão funcionalmente equivalente desse algoritmo. Por exemplo, funções que calculam fatorial iterativa e recursivamente apresentam ambas custo temporal linear (v. **Capítulo 6**). Por outro lado, uma função que calcula números de Fibonacci iterativamente apresenta custo temporal linear, ao passo que uma função que efetua esse cálculo recursivamente apresenta custo temporal que cresce exponencialmente.
- Um conselho mais simples e evidente é evitar o uso de recursão em situações nas quais existem algoritmos iterativos relativamente curtos, simples e fáceis de implementar. Em especial, evite imitar os exemplos didáticos apresentados aqui (e em outros textos sobre programação), tais como `Fatorial()` (v. acima), `Fib()` (v. **Seção 4.8.2**), `SomaAteN2()` (v. **Seção 4.1**), etc. Isso não quer dizer, entretanto, que todos os exemplos de recursão apresentados neste livro (e em outros texto de programação devem ser evitados). O uso de recursão é especialmente indicado para processamento de estruturas de dados definidas recursivamente (p.ex., listas generalizadas no **Capítulo 11** e árvores, vistas no **Capítulo 12**). Outra situação de uso de recursão é na implementação de algoritmos de divisão e conquista, como o problema das torres de Hanói (v. **Seção 4.8.1**).
- Evite o uso de recursão para funções que contêm grandes arrays de duração automática, pois um grande número de chamadas recursivas pode rapidamente levar a sobrecarga de pilha (*stack overflow*).
- Enfim, use recursão apenas quando essa técnica realmente simplifica o código de um programa.

Recursão deve ser definitivamente removida de um programa se ela implica em processamento redundante, como será visto na **Seção 4.8.2**. Às vezes, esse processamento redundante só é percebido com o uso de representações gráficas de chamadas recursivas.

Alguns fabricantes de software adotam uma política de tolerância zero com respeito a recursão. Esses fabricantes usam programas de análise estática de código para identificar chamadas recursivas e então elas são removidas das versões finais dos programas.

4.8 Exemplos de Programação

4.8.1 O (Cansativo) Problema das Torres de Hanói

Preâmbulo: Na maioria das vezes, os problemas encontrados pelo programador não precisam ser resolvidos de maneira recursiva. Isto é, a maioria dos problemas pode ser resolvida de maneira iterativa e o programador não tem que se preocupar em procurar soluções recursivas. Entretanto, existem problemas que possuem soluções naturalmente recursivas mais fáceis de serem encontradas. Um tal problema, conhecido como o **problema das torres de Hanói**, será descrito a seguir.

Inicialmente, no problema das torres de Hanói, existem três hastes e um determinado número de discos de diâmetros diferentes empilhados uns sobre os outros numa das hastes. O que o problema requer é que os discos sejam movidos de uma haste para outra obedecendo duas restrições:

1. Nenhum disco pode ser colocado sobre um outro disco de diâmetro menor.
2. Apenas o disco do topo de uma haste pode ser movido num dado instante. Ou, em outras palavras, para mover-se um dado disco, deve-se primeiro mover os discos que estão sobre ele.

A **Figura 4–23** ilustra a condição inicial do problema das torres de Hanói para três discos em que os discos foram numerados para facilitar referências a eles.

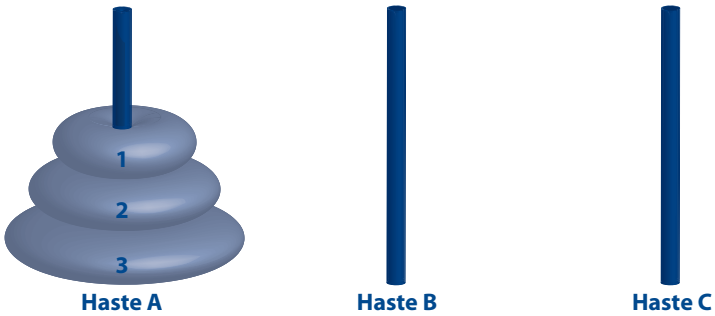

FIGURA 4–23: PROBLEMA DAS TORRES DE HANÓI

Problema: Escreva um programa que resolva o problema das torres de Hanói.

Solução: Considerando o diagrama da **Figura 4–23**, o problema consiste em deslocar os três discos na *Haste A* (denominada de **haste de origem**) para a *Haste C* (denominada de **haste de destino**), utilizando a *Haste B* como haste auxiliar. Por enquanto, não se preocupe com entrada e saída de dados do programa a ser desenvolvido e concentre-se na solução do problema para o caso geral no qual existem n discos.

Para começar, suponha que a solução do problema para $n - 1$ discos seja conhecida. Então, se for possível descrever a solução para n discos em termos da solução para $n - 1$ discos, o problema será facilmente resolvido. De fato, isto é verdade porque, movendo-se $n - 1$ discos para a *Haste B* (auxiliar) deixa-se apenas um disco para ser removido e, no caso trivial de um único disco, a solução é imediata: apenas mova esse disco da *Haste A* para a *Haste C*. Para um melhor entendimento, considere novamente o caso particular onde $n = 3$ e acompanhe passo a passo a solução do problema mostrada na **Figura 4–24**.

Seguindo o raciocínio utilizado para resolver o problema das torres de Hanói com três discos, pode-se generalizar a solução para mover n discos da haste A para a haste C, utilizando a haste B como auxiliar, por meio do algoritmo da **Figura 4–25**.

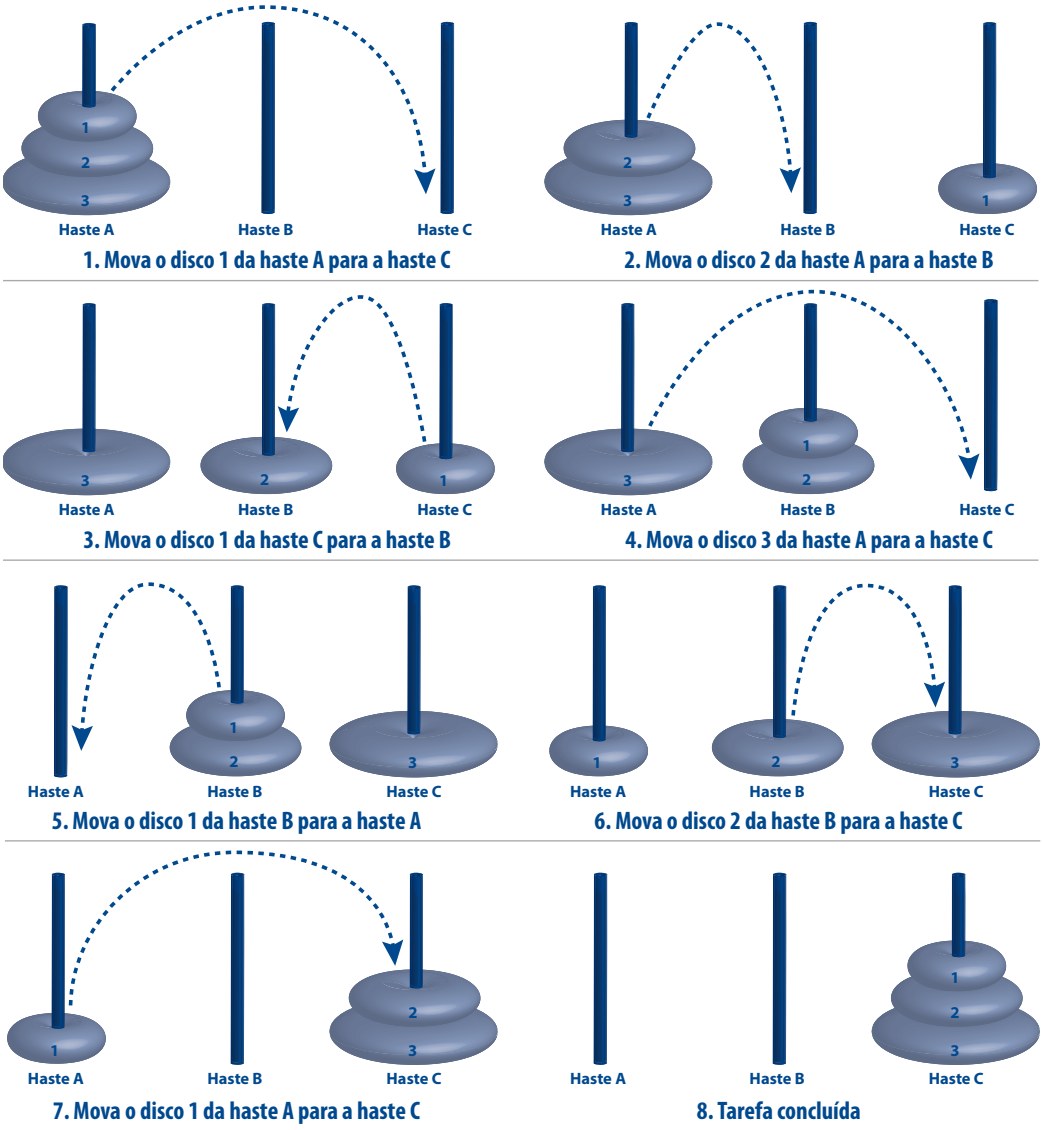

Figura 4–24: Solução do Problema das Torres de Hanói com Três Discos

Algoritmo TorresDeHanói

Entrada: n discos numa haste (A)

Saída: n discos noutra haste (C)

1. Se $n = 1$, mova o único disco da haste A para a haste C e pare
2. Mova os $n - 1$ discos do topo da haste A para a haste B utilizando a haste C como auxiliar
3. Mova o disco remanescente na haste A para a haste C
4. Mova os $n - 1$ discos da haste B para a haste C utilizando a haste A como auxiliar

Figura 4–25: Algoritmo Torres de Hanói

Não é difícil verificar que esse algoritmo realmente produz a solução correta para o problema das torres de Hanói para qualquer valor de $n \geq 1$. Observe que, para $n = 1$, o **Passo 1** resultará na solução esperada. Se $n = 2$, a solução para $n - 1$ já é conhecida, de modo que os **Passos 2** e **4** podem ser executados sem problemas. Quando

$n = 3$, a solução para $n - 1$ já é conhecida e os **Passos 2** e **4** podem ser novamente executados. Prosseguindo com esse raciocínio, pode-se mostrar que a solução fornecida pelo algoritmo acima funciona para $n = 1, 2, \ldots, k$, em que k é um valor inteiro positivo arbitrário.

A solução algorítmica para o problema das torres de Hanói ainda não está completa. É necessário ainda que sejam definidas a entrada e a saída para o problema. A entrada é fácil de ser identificada, pois ela corresponde ao número `n` de discos. A saída do algoritmo refere-se às representações de discos e hastes e como o movimento de discos de uma haste para outra deve ser apresentado. A escolha mais natural de saída parece ser de natureza gráfica com os discos sendo movidos utilizando uma interface gráfica com animação. Esse tipo de saída não será utilizado aqui, pois o interesse aqui é entender as ideias fundamentais de recursão, e não detalhes de interface gráfica. Portanto a saída será apresentada de uma forma textual mais simples contendo frases do tipo:

Mova o disco D da haste H1 para a haste H2

Também, serão adotadas as seguintes convenções:

[1] A haste de origem é denominada `A`, a haste de destino é denominada `C` e a haste auxiliar é denominada `B`.

[2] Os discos são numerados de `1` a `n`, a partir do disco do topo na haste de origem na situação inicial (em outras palavras, os discos são numerados de `1` a `n` do menor para o maior).

A função `TorresDeHanoi()` apresentada a seguir implementa o que foi exposto.

```
void TorresDeHanoi(int nDiscos, int hasteOrigem, int hasteDestino, int hasteAuxiliar)
{
   if (nDiscos == 1){
         /* Passo 1 do algoritmo: escreve e encerra */
      printf( "Mova o disco 1 da haste %c para a haste %c\n",
              hasteOrigem, hasteDestino );
      return;
   }

      /* Passo 2 do algoritmo: move os n-1 discos */
      /* de A para B usando C como auxiliar       */
   TorresDeHanoi(nDiscos - 1, hasteOrigem, hasteAuxiliar, hasteDestino);

      /* Passo 3 do algoritmo: move último disco de A para C */
   printf( "Mova o disco %d da haste %c para a haste %c\n",
           nDiscos, hasteOrigem, hasteDestino );

      /* Passo 4 do algoritmo: move os n-1 discos */
      /* de B para C usando A como auxiliar       */
   TorresDeHanoi(nDiscos - 1, hasteAuxiliar, hasteDestino,hasteOrigem);
}
```

A função **main**() a seguir lê o número de discos introduzidos pelo usuário e invoca a função para resolver o problema.

```
int main(void)
{
   int nDiscos;

      /* Solicita dado ao usuário */
   printf("Introduza o numero de discos: ");
   nDiscos = LeNaturalPositivo();

   TorresDeHanoi(nDiscos, 'A', 'C', 'B');

   return 0;
}
```

Exemplo de execução do programa: Executando o programa acima com um número de discos igual a 3 obtém-se:

```
Introduza o numero de discos: 3

Mova o disco 1 da haste A para a haste C
Mova o disco 2 da haste A para a haste B
Mova o disco 1 da haste C para a haste B
Mova o disco 3 da haste A para a haste C
Mova o disco 1 da haste B para a haste A
Mova o disco 2 da haste B para a haste C
Mova o disco 1 da haste A para a haste C
```

Uma questão que pode surgir na definição da função **`TorresDeHanoi()`** acima é: *como os parâmetros desta função são escolhidos?* Parece trivial entender por que o número de discos deve ser um parâmetro que é reduzido a cada chamada recursiva até que a condição de parada seja satisfeita. O uso das três hastes (**`hasteOrigem`**, **`hasteDestino`** e **`hasteAuxiliar`**) como parâmetros também não é difícil de entender. Basta perceber que soluções intermediárias do problema envolvem movimentos com as hastes **`A`**, **`B`** e **`C`** sendo ora origem, ora destino, ora auxiliar do movimento. Deve-se observar ainda que o programa foi facilitado pelo fato de a numeração dos discos ter sido feita do menor para o maior (verifique isto).

Observe que, da mesma forma que na elaboração da função **`SomaAteN2()`**, a solução obtida aqui para o problema das torres de Hanói foi desenvolvida identificando-se um caso trivial (i.e., quando o número de discos é igual a 1) e uma solução para o caso geral (i.e., para **`n`** discos) em termos de um caso mais simples (i.e., para **`n - 1`** discos). Em termos de estratégia de resolução de problemas, entretanto, existe uma diferença fundamental entre os dois problemas aqui descritos. O problema de encontrar a soma dos números entre **`1`** e **`n`** pode facilmente ser resolvido sem o uso de recursão e a solução iterativa não apenas é mais clara como também é mais eficiente em termos de recursos computacionais. O problema das torres de Hanói, por outro lado, não possui solução não recursiva trivial e, portanto, representa uma situação prática na qual o uso de recursão é recomendável[2].

4.8.2 Fibonacci + Recursão = Ineficiência 1

Preâmbulo: Uma **sequência de Fibonacci** é uma sequência de números naturais, cujo primeiro termo é igual a **`0`**, o segundo termo é igual a **`1`** e cada número (exceto os dois primeiros) na sequência é igual à soma de seus dois antecedentes mais próximos. Isto é, a sequência de Fibonacci é constituída da seguinte forma:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Problema: É relativamente fácil escrever uma função iterativa que gera números de Fibonacci (i.e., números que fazem parte de uma sequência de Fibonacci). Mas é tentador escrever tal função recursivamente, pois, afinal, um número de Fibonacci de ordem *n* pode ser definido recursivamente como:

$$Fib(n) = \begin{cases} n & se\ n < 2 \\ Fib(n-1) + Fib(n-2) & se\ n \geq 2 \end{cases}$$

(a) Implemente uma função em C que reflita a definição recursiva apresentada acima. (b) Mostre que essa implementação é deveras ineficiente.

Solução de (a):

[2] O quebra-cabeça das torres de Hanói foi inventado pelo matemático francês Edouard Lucas em 1883 e possui propriedades matemáticas bastante curiosas. Por exemplo, utilizando o quebra-cabeça com 64 discos e movendo-se um disco por segundo, levar-se-iam 580 bilhões de anos para completar a tarefa (v. **Seção 6.11.6**).

```
int Fib(int n)
{
   printf("Fib(%d)\t", n);

   if (n < 2)
      return n; /* Dois primeiros termos */
   return Fib(n - 2) + Fib(n - 1); /* Caso recursivo */
}
```

Solução de (b): A **Figura 4–26** mostra chamadas sucessivas da função `Fib()` quando ela é invocada num programa recebendo `6` como parâmetro real. Observe que, para calcular o número de Fibonacci de ordem 6 são necessárias 25 chamadas da função `Fib()`. E o pior é que várias dessas chamadas são para efetuar o mesmo cálculo. Por exemplo, há duas chamadas `Fib(4)`, três chamadas `Fib(3)` e cinco chamada `Fib(0)`. Para $n = 20$, o número de chamadas cresce para cerca de *22.000*.

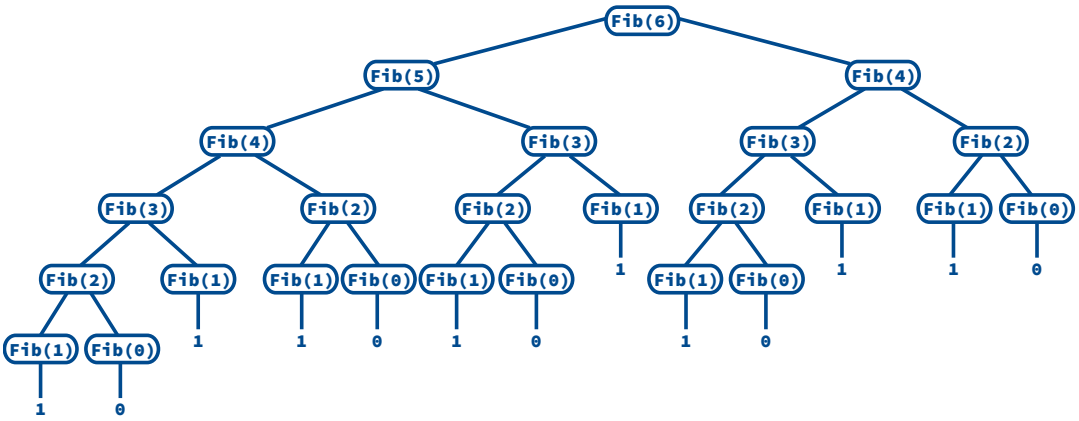

FIGURA 4–26: INEFICIÊNCIA DA FUNÇÃO RECURSIVA QUE CALCULA NÚMEROS DE FIBONACCI

É interessante notar que um diagrama de recursão, como aqueles apresentados na **Seção 4.1**, representando chamadas da função `Fib()` não é capaz de mostrar o processamento redundante que acontece com nessa função. Por outro lado, uma representação gráfica de chamadas, como aquela ilustrada na **Figura 4–26**, pode mostrar a ineficiência que seria difícil de enxergar de outro modo. Essa última representação gráfica é denominada **árvore de recursão**.

Pelo exposto acima, definitivamente, o uso de recursão não é recomendado para cálculo de números de Fibonacci. Existem duas maneiras bem mais eficientes para calcular esses números. A primeira delas é usando uma função iterativa, o que é relativamente trivial, como é mostrado a seguir:

```
int Fib2(int n)
{
   int antecedente1, antecedente2, atual;

   if (n < 2)
      return n;

   antecedente1 = 0;
   antecedente2 = 1;

   for (int i = 2; i <= n; i++) {
      atual = antecedente1 + antecedente2;

      antecedente1 = antecedente2; /* Atualiza os termos antecedentes */
      antecedente2 = atual;
   }
```

```
   return atual;
}
```

A segunda maneira eficiente de calcular números de Fibonacci é usando raciocínio matemático. Quer dizer, resolvendo a relação de recorrência que representa a sequência (v. **Seção 6.9**).

O exemplo chocante discutido nesta seção mostra que recursão deve ser cuidadosamente analisada antes que se decida adotá-la. A função `Fib()` é elegante, mas extremamente ineficiente; por outro lado, a função iterativa `Fib2()`, que é funcionalmente equivalente a `Fib()`, pode não ser tão elegante, mas é eficiente e relativamente fácil de implementar.

4.8.3 Calculando o Comprimento de um String Recursivamente

Problema: Escreva uma função recursiva funcionalmente equivalente à função **strlen()** da biblioteca padrão de C que calcula o comprimento de um string.

Solução: A função `ComprimentoStrRec()` definida abaixo calcula o comprimento de um string recursivamente e retorna o resultado obtido.

```
int ComprimentoStrRec(const char *str)
{
   if (!*str)  /* O string é vazio */
      return 0; /* Caso terminal */

      /* Caso recursivo */
   return 1 + ComprimentoStrRec(str + 1);
}
```

A função apresentada acima é relativamente fácil de entender. O caso terminal ocorre quando o string é vazio (i.e., quando ele contém apenas o caractere terminal `'\0'`). O caso recursivo deve ser interpretado assim: o comprimento de um string é um mais o comprimento do string sem seu primeiro caractere. Em termos de custo temporal, tanto a função apresentada aqui quanto a função **strlen()** (que não é implementada recursivamente) tem custo diretamente proporcional ao tamanho do string. Entretanto, essas duas funções diferem em termos de custo de espaço utilizado: a função **strlen()** não usa espaço adicional (i.e., espaço além daquele necessário para armazenar o próprio string); por outro lado, a função apresentada acima usa espaço adicional correspondente aos registros de ativação alocados na pilha de execução durante as chamadas recursivas.

Observação: Este exemplo é *meramente didático* e não deve ser imitado na prática. Ou seja, a melhor maneira de calcular o comprimento de strings é por meio de iteração.

4.8.4 Removendo Vogais de um String Recursivamente

Problema: Escreva uma função recursiva que remove todas as vogais de um string.

Solução:

```
void RemoveVogais(char *str)
{
   static const char *const vogais = "aeiouAEIOU";

   if (!*str) /* O string é vazio */
      return; /* Caso terminal */

      /* Casos recursivos */
   if (strchr(vogais, *str)) /* Encontrada uma vogal */
      RemoveVogais(strcpy(str, str + 1));
   else
      RemoveVogais(str + 1);
}
```

A função `RemoveVogais()` é responsável pela remoção das vogais de um string e funciona do seguinte modo:

- Novamente, caso terminal ocorre quando o string é vazio (v. **Seção 4.8.3**), pois, evidentemente, um string vazio não possui nenhuma vogal a ser removida.
- A função sob análise apresenta dois casos recursivos. O primeiro deles ocorre quando o primeiro caractere do string é uma vogal. Nesse caso, a porção do string que sucede essa vogal é copiada para a porção inicial do string, assim, sobrescrevendo a vogal. Então, a função `RemoveVogais()` é chamada recursivamente para remover as vogais desse novo string assim construído.
- A segunda chamada recursiva refere-se à situação na qual nenhuma vogal foi encontrada no início do string. Nesse caso, a função é chamada recursivamente para remover as vogais no string que resta quando não se leva em consideração o primeiro caractere.

Observação: Este exemplo é *meramente didático* e não deve ser imitado na prática. Ou seja, a melhor maneira de remover vogais de strings é por meio de iteração.

4.8.5 Exponenciação por Quadratura 1

Preâmbulo: A exponenciação de um número real x elevado a um número inteiro $n > 0$ pode ser definida recursivamente como:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ \left(\frac{1}{x}\right)^{-n} & \text{se } n < 0 \\ x \cdot \left(x^{\frac{n-1}{2}}\right)^2 & \text{se } n \text{ for ímpar} \\ \left(x^{\frac{n}{2}}\right)^2 & \text{se } n \text{ for par} \end{cases}$$

Problema: Implemente uma função em C que reflita essa definição.

Solução: Por simplicidade, a função `ExponenciacaoQuad()` apresentada a seguir não lida com casos excepcionais (i.e., ela não funciona adequadamente quando o expoente é negativo ou nulo e a base é `0`).

```
double ExponenciacaoQuad(double x,int n)
{
   if (n < 0)
      return ExponenciacaoQuad(1/x, -n);

   if (n == 0)
      return 1;

   if (n == 1)
      return x;

   if (n%2 == 0)
      return ExponenciacaoQuad(x*x, n/2);

   return x * ExponenciacaoQuad(x*x, (n-1)/2);
}
```

O algoritmo seguido na implementação dessa última função é chamado **exponenciação por quadratura** porque a exponenciação é calculada por meio de contínua elevação ao quadrado de resultados parciais. Na **Seção 6.11.8**, será mostrado que a função `ExponenciacaoQuad()` é bastante eficiente.

Observação: Na ausência de algoritmo melhor, a função `ExponenciacaoQuad()` apresentada neste exemplo pode usada na prática. Entretanto, existem implementações mais eficientes do que ambas as funções, mas que não são discutidas neste livro.

4.8.6 Invertendo Entradas 1

Problema: Escreva um programa que inverte qualquer sequência de caracteres que o usuário digitar no meio de entrada padrão.

Solução: O programa solicitado usa a função recursiva `Inverte()`, apresentada abaixo para inverter os caracteres introduzidos pelo usuário no meio de entrada padrão

```
void Inverte(void)
{
   int c;

   if ((c = getchar()) != '\n')
      Inverte(); /* Caso recursivo */

   putchar(c); /* Caso terminal */
}
```

A função **main()** a seguir completa o programa:

```
int main (void)
{
   int    c;

   printf( "\n>>> Digite uma sequencia de caracteres "
           "\n>>> para o programa inverter: " );

   Inverte();

   return 0;
}
```

Exemplo de execução do programa:

```
>>> Digite uma sequencia de caracteres
>>> para o programa inverter: Roma

amoR
```

A função `Inverte()` possui um caso terminal e um caso recursivo, a saber:

- O caso terminal ocorre quando o caractere lido é o caractere `'\n'` que, normalmente, encerra a entrada de dados via teclado. Nesse caso, a função simplesmente escreve esse caractere.
- O caso recursivo é acionado quando o caractere lido *não* é o caractere `'\n'`. Nesse caso, a função é chamada recursivamente para ler outro caractere.

Observação: Esta solução representa um belo exemplo de uma função recursiva sem nenhum parâmetro e que *não* é meramente didática. Quer dizer, encontrar uma solução para esse problema mais simples de implementar não é trivial. Portanto, a não ser que você tenha uma razão muito boa para não adotar essa solução recursiva, adote-a. A **Seção 8.5.1** apresentará outra solução para esse problema. A solução apresentada naquela seção não é recursiva, mas mesmo assim não é a melhor para o caso em questão, o que mostra que nem toda solução iterativa é melhor uma solução recursiva equivalente.

4.8.7 Exibindo-se em Frente e Verso

Problema: Escreva um programa que exibe seu código-fonte na tela de duas maneiras: **(1)** do início para o final e **(2)** do final para o início. Implemente essas duas operações usando funções recursivas.

Solução: A função `ExibeArquivoNaTelaRec()`, apresentada a seguir, escreve um arquivo de texto na tela recursivamente. Seu único parâmetro é o stream associado ao arquivo que se pressupõe estar aberto em modo que permite leitura e com o apontador de posição apontando para o primeiro byte do arquivo.

```
void ExibeArquivoNaTelaRec(FILE *stream)
{
   if ( !feof(stream) && !ferror(stream) ) {
      putchar(fgetc(stream));
      ExibeArquivoNaTelaRec(stream);
   }
}
```

A função `ExibeArquivoNaTelaRec()` não aparenta possuir nenhum caso terminal, pois o corpo dessa função contém apenas:

```
   if ( !feof(stream) && !ferror(stream) ) {
      putchar(fgetc(stream));
      ExibeArquivoNaTelaRec(stream);
   }
```

Acontece, porém, que o caso terminal dessa função ocorre exatamente quando a expressão condicional que acompanha a instrução **if** não é satisfeita, pois, nesse caso, a função retorna imediatamente.

A função `ExibeArquivoInvNaTelaRec()`, apresentada a seguir, escreve recursivamente um arquivo de texto invertido na tela. Seu único parâmetro tem a mesma interpretação do parâmetro da função `ExibeArquivoNaTelaRec()`.

```
void ExibeArquivoInvNaTelaRec(FILE *stream)
{
   int c;

   c = fgetc(stream);

   if ( !feof(stream) && !ferror(stream) )
      ExibeArquivoInvNaTelaRec(stream);

   putchar(c);
}
```

A função **main()** que completa o programa é a seguinte:

```
int main(int argc, char** argv)
{
   char *nome;
   FILE *stream;

      /* Tenta determinar o nome do arquivo fonte */
   if (strchr(*argv, '.')) { /* Arquivo parece que tem extensão */
      *(strchr(*argv, '.') + 1) = 'c';
      *(strchr(*argv, '.') + 2) = '\0';
   } else {
         /* O arquivo não tem extensão. Não se pode usar o mesmo truque da parte   */
         /* if, pois causará corrupção de memória. Alocação dinâmica é necessária. */
         /* O número mágico 3 representa os três caracteres: '.' + 'c' + '\0'.     */
      nome = malloc(strlen(*argv) + 3);
      strcat(strcpy(nome, *argv), ".c");
   }

      /* Esgotou a capacidade de adivinhação do programa */
      /* Se não conseguir abrir o arquivo, desiste.      */
   if (!(stream = fopen(*argv, "r"))) {
      printf( "\nNao foi possivel abrir o aquivo: %s\n", *argv );
      return 1;
   }

      /* Exibe-se do início para o final */
   ExibeArquivoNaTelaRec(stream);
```

```
    /* Faz o apontador de posição voltar para o primeiro byte */
  rewind(stream);

    /* Exibe-se do final para o início */
  ExibeArquivoInvNaTelaRec(stream);

  return 0;
}
```

Um fato que merece destaque na função **main()** é o uso de **malloc()** para determinar o nome do arquivo fonte. Se você desconhece alocação dinâmica de memória, que será estudada no **Capítulo 9**, talvez tenha dificuldade em entender qual é o papel de **malloc()** nesse programa. Mas isso não deve ser empecilho para o entendimento do tema central deste exemplo que é recursão.

4.8.8 Resolvendo Sudoku

Preâmbulo: **Sudoku** é um quebra-cabeça que tem como objetivo o preenchimento com dígitos de 1 a 9 de quadrados dispostos numa **grade** 9 x 9. Essa grade é subdividida em nove **subgrades** e alguns dos quadrados são previamente preenchidos, como mostra a **Figura 4–27**. O preenchimento dos quadrados com dígitos é sujeito às seguintes restrições: **(1)** nenhum dígito pode ser repetido em qualquer linha ou coluna (essa restrição não se aplica a diagonais) e **(2)** nenhum dígito pode ser repetido em qualquer subgrade.

Problema: Escreva um programa para encontrar soluções para quebra-cabeças Sudoku.

8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

FIGURA 4–27: QUEBRA-CABEÇA SUDOKU

Solução: Esse é mais um exemplo clássico de problema de satisfação de restrições cuja técnica de resolução indicada é retrocesso. O programa discutido a seguir utiliza essa técnica para resolver quebra-cabeças Sudoku.

A função `PreencheSudoku()` apresentada abaixo preenche recursivamente cada quadrado de uma grade que representa o quebra-cabeça Sudoku e retorna 1, se a grade que representa o quebra-cabeça foi preenchida, ou 0, em caso contrário. Essa função utiliza os seguintes parâmetros:

- `grade[][]` (entrada e saída) é um array bidimensional que representa o estado atual do quebra-cabeça
- `linha` e `coluna` (entrada) representam a linha e a coluna que identificam um quadrado da grade a ser preenchido

```
int PreencheSudoku(int grade[][9], int linha, int coluna)
{
  int numero;

    /* Verifica se a linha e a coluna recebidas como parâmetros fazem parte  */
    /* da grade. Se não for o caso, retorna-se 1, indicando que uma linha ou */
    /* coluna está completa.                                                 */
```

```
  if (linha < 9 && coluna < 9) {
        /* Verifica se o quadrado representado por 'linha' e 'coluna' está     */
        /* preenchido. Se for o caso, tenta-se preencher o quadrado na próxima */
        /* coluna da mesma linha ou na próxima linha da mesma coluna. Se esse  */
        /* quadrado não existir, retorna-se 1, informando que o quebra-cabeça  */
        /* foi resolvido.                                                      */
     if(grade[linha][coluna]) {
           /* Se existir um quadrado na próxima coluna */
           /* desta linha, tenta-se preenchê-lo        */
        if (coluna + 1 < 9)
           return PreencheSudoku(grade, linha, coluna + 1);
           /* Se existir um quadrado na próxima linha */
           /* desta coluna, tenta-se preenchê-lo      */
        else if(linha + 1 < 9)
           return PreencheSudoku(grade, linha + 1, 0);
           /* Não existe tal quadrado. Logo retorna-se 1,  */
           /* indicando que o quebra-cabeça foi resolvido. */
        else return 1;
     } else {
           /* O quadrado representado por 'linha' e 'coluna' não está preenchido. */
        for(numero = 1; numero <= 9; ++numero) {
              /* Tenta preencher o quadrado com o número corrente */
           if(EstaDisponivel(grade, linha, coluna, numero)) {
              grade[linha][coluna] = numero;

                 /* Se existir um quadrado na próxima coluna */
                 /* desta linha, tenta-se preenchê-lo        */
              if(coluna + 1 < 9) {
                    /* Tenta preencher o quadrado. Se for possível, indica-se */
                    /* o sucesso; caso contrário, torna-o vazio novamente para */
                    /* provocar o retrocesso.                                  */
                 if(PreencheSudoku(grade, linha, coluna + 1))
                    return 1;
                 else
                    grade[linha][coluna] = 0;
                 /* Se existir um quadrado na próxima linha */
                 /* desta coluna, tenta-se preenchê-lo      */
              } else if(linha + 1 < 9) {
                    /* Tenta preencher o quadrado. Se for possível, indica-se  */
                    /* o sucesso; caso contrário, torna-o vazio novamente para */
                    /* provocar o retrocesso.                                  */
                 if(PreencheSudoku(grade, linha + 1, 0))
                    return 1;
                 else
                    grade[linha][coluna] = 0;
                 /* O quadrado não existe tal. Retorna-se 1, */
                 /* indicando que o problema foi resolvido   */
              } else
                   return 1;
           }
        }
     }
     return 0;
  } else
     return 1; /* Problema resolvido */
}
```

A função `PreencheSudoku()` chama `EstaDisponivel()` para verificar se um número pode ser usado no quadrado especificado por uma linha e uma coluna de grade do quebra-cabeça Sudoku. Quando o referido número estiver disponível para uso, essa última função retorna 1; caso contrário, ela retorna zero. Os parâmetros dessa função são:

- `grade[][]` (entrada) que é o array bidimensional que representa o estado atual do quebra-cabeça
- `linha` e `coluna` (entrada) representam a linha e a coluna que definem o referido quadrado
- `numero` (entrada) é o número a ser verificado

```
int EstaDisponivel( int grade[][9], int linha, int coluna, int numero )
{
        /* Linha inicial de uma subgrade (0, 3 ou 6) */
   int linhaInicial = (linha/3) * 3,
        /* Coluna inicial de uma subgrade (0, 3 ou 6) */
       colunaInicial = (coluna/3) * 3;

   for(int i = 0; i < 9; ++i) {
        /* Verifica se o número já ocorre na linha em que o quadrado se encontra */
      if (grade[linha][i] == numero)
         return 0; /* O número já aparece na linha */

         /* Verifica se o número já ocorre na coluna em que o quadrado se encontra */
      if (grade[i][coluna] == numero)
         return 0; /* O número já aparece na coluna */

        /* Verifica se o número já ocorre na subgrade em que o quadrado se encontra */
      if (grade[linhaInicial + (i%3)][colunaInicial + (i/3)] == numero)
         return 0; /* O número já aparece na subgrade */
   }

      /* O número está disponível */
   return 1;
}
```

A função **main()** apresentada a seguir define um array bidimensional que representa a grade de um quebra-cabeça Sudoku e chama a função `PreencheSudoku()` para resolver o problema.

```
int main(void)
{
      /* Array que que representa a grade do quebra-cabeça. */
      /* 0 significa um quadrado não preenchido.            */
   int grade[9][9]={ {8, 0, 0, 4, 0, 6, 0, 0, 7},
                     {0, 0, 0, 0, 0, 0, 4, 0, 0},
                     {0, 1, 0, 0, 0, 0, 6, 5, 0},
                     {5, 0, 9, 0, 3, 0, 7, 8, 0},
                     {0, 0, 0, 0, 7, 0, 0, 0, 0},
                     {0, 4, 8, 0, 2, 0, 1, 0, 3},
                     {0, 5, 2, 0, 0, 0, 0, 9, 0},
                     {0, 0, 1, 0, 0, 0, 0, 0, 0},
                     {3, 0, 0, 9, 0, 2, 0, 0, 5} };

      /* Apresenta o desafio a ser enfrentado */
   printf("\n>>> Quebra-cabeca a ser resolvido <<<\n");
   ApresentaGrade(grade);

      /* Verifica se o quebra-cabeça foi resolvido */
   if(PreencheSudoku(grade, 0, 0)) { /* Foi resolvido */
         /* Apresenta o resultado */
      printf("\n>>> Quebra-cabeca resolvido <<<\n");
```

```
      ApresentaGrade(grade);
   } else { /* O quebra-cabeça não foi resolvido     */
            /* provavelmente porque foi mal formulado */
      printf("\n\nEste Sudoku nao tem solucao\n\n");
      return 1;
   }

   return 0;
}
```

A função **main**() acima chama `ApresentaGrade()`, definida abaixo, para apresentar na tela uma grade que representa o quebra-cabeça Sudoku.

```
void ApresentaGrade(int grade[][9])
{
      int i, j;

      printf("\n+-----+-----+-----+\n");

      for(i = 1; i < 10; ++i) {
         for(j = 1; j < 10; ++j)
            printf("|%d", grade[i - 1][j - 1]);

         printf("|\n");

         if (i%3 == 0)
            printf("+-----+-----+-----+\n");
      }
}
```

O ponto central do programa apresentado acima é a função `PreencheSudoku()` que, mesmo recheada de comentários, não é fácil de entender por meio de uma simples leitura. Mas, assim como o funcionamento da função `Posiciona()`, discutida na **Seção 4.5.1**, foi esclarecido por meio do uso de digramas que mostravam diversos estados da pilha de execução, esse será o caso se você fizer o mesmo para a função `PreencheSudoku()`.

4.9 Exercícios de Revisão

Funções Recursivas (Seção 4.1)

1. (a) O que é iteração? (b) O que é recursão?
2. É verdade que todo algoritmo não recursivo é iterativo? Explique.
3. (a) O que uma função iterativa? (b) O que uma função recursiva?
4. (a) O que é caso base? (b) O que é caso recursivo?
5. (a) O que é um diagrama de recursão? (b) Para que servem diagramas de recursão?
6. O que realiza a seguinte função:

```
void F(void)
{
   int c = getchar();

   if (c != '\n')
      F();

   putchar(c);
}
```

7. Apresente o resultado escrito na tela por cada um dos seguintes programas:

(a)

```
#include <stdio.h>

int Funcao(int  x)
{
   int  y = 0;
   y += x;
   return  y;
}

int main()
{
   int a, contador;
   for (contador = 1; contador <= 5; ++contador)  {
      a = Funcao(contador);
      printf("%d  ", a);
   }
   return 0;
}
```

(b)

```
#include <stdio.h>

int Funcao1(int  a)
{
   int  b = 1;
   b += 1;
   return  b + a;
}

int Funcao2(int x)
{
   int  b;
   b = Funcao1(x);
   return  b;
}

int main()
{
   int  a = 0, b = 1, contador;
   for (contador = 1; contador <= 5; ++contador)  {
      b += Funcao1(a) + Funcao2(a);
      printf("%d  ", b);
   }

   return 0;
}
```

8. Considere a seguinte função

```
int F(int n, int m)
{
   if (n > m)
      return -1;
   else if (n == m)
      return 1;
   else
     return n*F(n+1, m);
}
```

(a) Qual é o caso base dessa função?

(b) Qual é o caso recursivo dessa função?

(c) Qual é o resultado da chamada `F(10, 5)` dessa função?

(d) Qual é o resultado da chamada `F(0, 0)` dessa função?

9. (a) Determine o que a seguinte função computa e (b) escreva uma função iterativa com a mesma finalidade.

```
int F(int n)
{
   if (n == 0)
      return 0;
   else
      return n + F(n - 1);
}
```

10. Se a função `F()` for definida como a seguir, qual será o valor que ela retorna quando for efetuada a chamada `F(5)`?

```
int F(int x)
{
   if (x == 1)
      return 0;
   else
      return x*F(x - 1) + x*x;
}
```

11. Considere a seguinte função:

```
int F(int n)
{
   if (!n)
      return 0;
   else
      return n + F(n + 1);
}
```

(a) O que há de errado com essa função?

(b) Qual é o resultado da chamada `F(1)`?

(c) O que ocorre com o programa que efetua essa última chamada?

12. (a) Qual é o valor retornado pela função `F()` a seguir quando é efetuada a chamada `F(2)`? (b) Para quais valores de `n` o programa que usa essa função é abortado?

```
int F(int n)
{
   if (n == 4)
      return 2;
   else
      return 2*F(n + 1);
}
```

13. Qual é o valor retornado pela função `F()` a seguir quando é efetuada a chamada `F(4)`?

```
int F(int n)
{
   if (n < 3)
      return n;
   else
      return F(n - 1)* F(n - 2) + F(n - 3);
}
```

14. Se $n > 0$, quantas vezes a função `F()` a seguir será chamada para calcular o valor resultante da chamada `F(n)`?

```
int F(int n)
{
   if (n == 1)
      return 2;
   else
      return 2*F(n - 1);
}
```

15. Qual é o valor retornado pela função `F()` a seguir quando é efetuada a chamada `F(3, 2, 6)`?

```
int F(int n, int a, int d)
{
   if (n == 1)
      return a;
   else
      return d + F(n - 1, a, d);
}
```

16. Qual é o valor retornado pela função `F()` a seguir quando é efetuada a chamada `F(6, 8)`?

```
int F(int i, int j)
{
   if (j == i)
      return i;
   else if (j > i)
      return F(i, j - i);
   else
      return F(i - j, j);
}
```

17. Considere a função `Escreve()` apresentada a seguir. (a) O que será escrito na tela quando for efetuada a chamada `Escreve(3)`? (b) Quantas chamadas recursivas serão efetuadas nesse caso?

```
void Escreve(int n)
{
   if (n > 0) {
      Escreve(n - 1);
      printf("%d", n);
      Escreve(n - 1);
   }
}
```

Cadeias Recursivas (Seção 4.2)

18. O que é uma cadeia recursiva?

19. Que perigo pode representar uma cadeia recursiva?

20. Considerando as funções `F1()` e `F2()` a seguir, qual será o valor retornado quando for efetuada a chamada `F1(5, 3)`?

```
extern int F2(int, int);

int F1(int n, int m)
{
   if (n == m)
      return m;
   else
      return n + F2(n - 1, m);
}
```

CONTINUA

```
int F2(int n, int m)
{
   if (n < m)
      return n + m;
   else
      return n + F1(n - 2, m);
}
```

Pilha de Execução e Registros de Ativação (Seção 4.3)

21. Como o espaço reservado em memória para execução de um programa é dividido?

22. O que é pilha de execução?

23. (a) O que são registros de ativação? (b) Qual é o conteúdo de um registro de ativação?

24. Qual é o primeiro registro de ativação armazenado na pilha de execução de um programa escrito em C?

25. Por que uma função recursiva é geralmente menos eficiente do uma função equivalente iterativa?

26. O que ocorre quando uma função recursiva é chamada e a base da recursão nunca é atingida?

27. (a) O que significa esgotamento de pilha (*stack overflow*)? (b) Qual é a relação entre *stack overflow* e recursão? (c) Uma função não recursiva pode causar *stack overflow*?

28. (a) O que é fase de acréscimo de uma função recursiva? (b) O que ocorre quando uma função recursiva só apresenta fase de acréscimo?

29. (a) O que é fase de decréscimo de uma função recursiva? (b) Uma função recursiva pode apresentar apenas fase de decréscimo?

Recursão de Cauda (Seção 4.4)

30. O que é recursão de cauda?

31. Comente a seguinte observação: *um bom compilador sabe otimizar recursão de cauda*.

32. A seguinte função `F()` apresenta recursão de cauda? Explique seu raciocínio.

```
int F(int n)
{
   if (n < 1)
      return 0;
   else
      return F(n - 1);
}
```

33. A seguinte função `G()` apresenta recursão de cauda? Explique seu raciocínio.

```
int G(int n)
{
   if (n < 1)
      return 0;
   else
      return n + G(n - 1);
}
```

Retrocesso (*Backtracking*) (Seção 4.5)

34. O que é um problema de satisfação de restrições?

35. Descreva em linhas gerais a técnica de retrocesso.

36. Qual é a relação entre retrocesso e recursão?

37. Cite três categorias de problemas que são propícios ao uso de retrocesso.

Como Pensar Recursivamente (Seção 4.6)

38. Qual é a relação entre recursão e a abordagem de refinamentos sucessivos de construção de algoritmos?

39. Como a abordagem de refinamentos sucessivos de construção de algoritmos deve ser adaptada para acomodar recursão?

40. Por que o protótipo de uma função recursiva parece às vezes bizarro? Apresente um exemplo.

41. O que é uma função acionadora de uma função recursiva?

Quando Usar (e Não Usar) Recursão (Seção 4.7)

42. Em que situações práticas devem ser utilizadas funções recursivas ao invés de iterativas (i.e., não recursivas)?

43. Qual é a melhor maneira de implementar uma função que calcula fatorial: iterativa ou recursiva? Explique sua resposta.

44. Se a forma recursiva de implementação de fatorial não é recomendável, por que implementações recursivas de fatorial aparecem tanto em livros de programação (inclusive este)?

Exemplos de Programação (Seção 4.8)

45. Se a função recursiva `Fib()` apresentada na **Seção 4.8.2** for chamada recebendo como parâmetro real um valor bem grande (digamos, maior do que `50`), é provável que ocorrerá esgotamento de pilha? Explique seu raciocínio.

46. Determine o número de adições na chamada `Fib(10)` da função `Fib()` apresentada na **Seção 4.8.2**.

47. (a) A função `ExibeArquivoNaTelaRec()` apresentada na **Seção 4.8.7** é justificável? (b) A função `ExibeArquivoInvNaTelaRec()` apresentada na **Seção 4.8.7** é justificável?

48. Descreva o algoritmo de resolução do problema das torres de Hanói.

49. Descreva o método de cálculo de exponenciação por quadratura.

50. A seguinte função foi escrita como alternativa para a função `ExponenciacaoQuad()` apresentada na **Seção 4.8.5**. Explique por que esta função é menos eficiente do que a função `ExponenciacaoQuad()`.

```
double ExponenciacaoAlt(double x, int y)
{
   if (!y)
      return 1;

   if (y%2) /* Base é ímpar */
      return x*ExponenciacaoAlt(x, y/2)*ExponenciacaoAlt(x, y/2);
   else /* Base é par */
      return ExponenciacaoAlt(x, y/2)*ExponenciacaoAlt(x, y/2);
}
```

51. Dentre os exemplos de funções recursivas apresentados neste capítulo, quais deles são justificáveis na prática?

4.10 Exercícios de Programação

EP4.1 Escreva uma função recursiva em C, denominada `Multiplica()`, que avalie o produto de dois números inteiros não negativos usando apenas adição.

EP4.2 (a) Escreva uma função recursiva em C que calcula um coeficiente binomial de acordo com a definição:

$$\binom{n}{k} = \begin{cases} 1 & \text{se } k = 0 \text{ ou } k = n \\ \binom{n-1}{k-1} + \binom{n}{k-1} & \text{em caso contrário} \end{cases}$$

(b) Desenhe uma árvore de recursão semelhante àquela apresentada na **Seção 4.8.2** e discuta a ineficiência da função solicitada.

EP4.3 Escreva uma função recursiva em C, denominada `Soma()`, que avalie a soma de dois números inteiros não negativos usando a função `Sucessor()` definida como:

```
int Sucessor(int x)
{
   return  ++x;
}
```

EP4.4 (a) Escreva uma função recursiva que calcule o máximo divisor comum de dois números inteiros positivos. (b) Escreva uma versão iterativa da função solicitada no item (a).

EP4.5 (a) Determine o que a seguinte função recursiva realiza:

```
int   MinhaFuncao(int  x)
{
   if  (!n)
      return  0;
   return  (n + MinhaFuncao(n - 1));
}
```

(b) Escreva uma função iterativa que tenha o mesmo efeito da função anterior.

EP4.6 Escreva uma função recursiva que apresenta na tela um número inteiro não negativo em base binária.

EP4.7 Escreva uma função recursiva que converte strings numéricos (i.e., strings contendo apenas dígitos) em números inteiros. Por exemplo, essa função transformaria o string `"345"` em `345`.

EP4.8 Repita o exercício **EP2.2** usando desta vez uma função recursiva para determinar se um número é perfeito ou não.

EP4.9 Escreva uma função recursiva para calcular o **número harmônico** de ordem n definido como:

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

EP4.10 Modifique o programa para o problema das torres de Hanói apresentado na **Seção 4.8.1** acima considerando que os discos são numerados do maior para o menor.

EP4.11 Escreva uma função recursiva para determinar o maior valor de um array de elementos do tipo **int**.

EP4.12 Escreva uma função recursiva para determinar a média dos valores de um array de elementos do tipo **int**.

EP4.13 (a) Escreva uma função recursiva que remove de um string todas as ocorrências de um dado caractere. O protótipo dessa função deve ser:

```
char *RemoveCaractere(char *str, int remover)
```

Nesse protótipo, `str` é o string que será eventualmente modificado, `remover` é o caractere a ser removido e o retorno da função deve ser o endereço inicial do string. (b) Escreva um programa que lê um string e um caractere via teclado e remove todas as ocorrências do caractere no string. O string deve ser apresentado na tela antes e depois das eventuais substituições.

EP4.14 Implemente uma função recursiva, denominada `ComparaStrings()`, funcionalmente equivalente à função **strcmp()**.

EP4.15 Um algoritmo, denominado **método de busca binária**, para calcular a raiz cúbica de um número x é o seguinte:

ALGORITMO RAIZCÚBICAPORBUSCABINÁRIA

ENTRADA: Um número real *x*

SAÍDA: A raiz de *x*

1. Comece com um limite inferior e outro superior
2. Se o número for maior do que *1*, use *1* como limite inferior e o próprio número *x* como limite superior
3. Se o número for menor do que *1*, use *x* como limite inferior e *1* como limite superior
4. Se a diferença entre os limites inferior e superior é menor do que um certo valor de precisão (por exemplo, *0.000001*), então o resultado será a média aritmética desses limites e o problema estará resolvido
5. Se o problema ainda não estiver resolvido, verifique se a média dos limites inferior e superior é maior ou menor do que a raiz cúbica de x elevando esta média ao cubo
6. Se a média for menor do que a raiz cúbica de x, repita o processo a partir do **passo 2** considerando agora a média como limite inferior e mantendo inalterado o limite superior
7. Se a média for maior do que a raiz cúbica de x, repita o processo a partir do **passo 2** mantendo o mesmo limite inferior e considerando a média como limite superior

(a) Implemente o algoritmo acima como uma função recursiva em C que recebe como parâmetros um número de ponto-flutuante `x` e os limites inferior e superior descritos no algoritmo e retorna a raiz cúbica de `x`.

(b) Escreva um programa que solicita um valor numérico do usuário, calcula a raiz cúbica deste valor utilizando a função descrita em (a) e imprime o resultado.

Sugestões:

(1) Defina a precisão como uma constante simbólica. Por exemplo:

```
#define PRECISAO 0.000001
```

(2) Utilize o seguinte protótipo para a função que calcula a raiz cúbica:

```
double RaizCubica(double x, double inferior, double superior)
```

EP4.16 Escreva uma função recursiva que retorna `1` quando um string possui apenas letras e dígitos ou `0`, em caso contrário. [**Sugestão:** Use a função **isalnum()** declarada em `<ctype.h>`.]

EP4.17 Escreva uma função recursiva que substitui cada caractere de tabulação de um string por um espaço em branco.

EP4.18 Escreva uma função recursiva, denominada `OcorrenciasCar()`, que conta o número de ocorrências de um caractere num string.

EP4.19 Escreva uma função recursiva que recebe um array bidimensional *N x N* como parâmetro e calcula a soma dos elementos do array.

EP4.20 Apesar de não apresentar recursão de cauda, a função `ExponenciacaoQuad()` apresentada na **Seção 4.8.5** pode ser facilmente implementada sem o uso de recursão. Reimplemente a função `ExponenciacaoQuad()` de modo iterativo.