MC202 - Estruturas de Dados

Guilherme P. Telles

IC

3 de maio de 2023

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides usam português anterior à reforma ortográfica de 2009.

Recursão

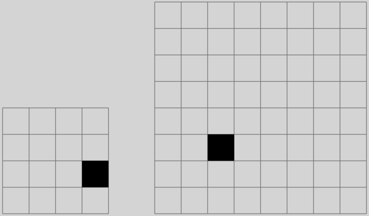
Recursão

- A recursão é uma técnica para resolver problemas, para provar que algoritmos estão corretos e para definir propriedades e estruturas-de-dados.
- É diretamente relacionada com o princípio de indução matemática.

- Como forma de resolver problemas, uma idéia é: ao invés de resolver o problema diretamente, divide-se a instância em algumas sub-instâncias do mesmo problema, resolve-se cada uma recursivamente e depois combinam-se as soluções das sub-instâncias.

Tabuleiro

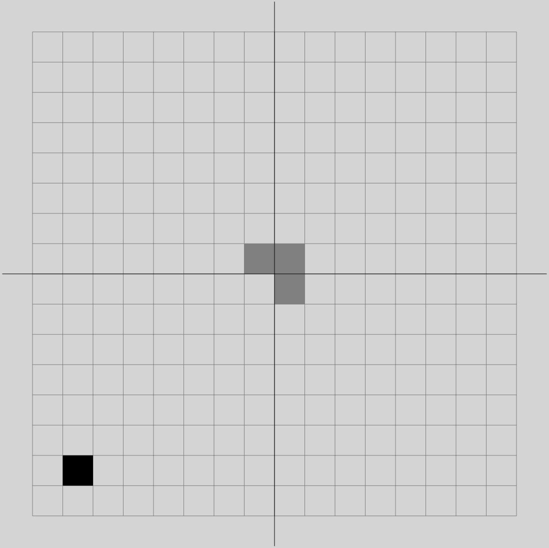
- É dado um tabuleiro $2^n \times 2^n$, $n \geq 2$, com exatamente uma celula preenchida (qualquer uma).



- O problema é preencher o tabuleiro com peças da forma abaixo, que podem ser rotacionadas.

- Se $n = 2$, então é fácil resolver.

- Se $n > 2$, suponha que posicionamos uma peça no centro, de tal forma que cada quadrante fique com uma célula preenchida.



- Cada quadrante é um subproblema: ele tem tamanho 2^{n-1} e exatamente uma célula preenchida.
- Basta resolver cada quadrante recursivamente. Não há trabalho adicional para combinar as soluções.

Ordenação

Queremos reorganizar os elementos do vetor $V[1..n]$ em ordem não-decrescente.

Se $n = 1$, V já está ordenado.

Se $n > 1$, vamos assumir que sabemos ordenar vetores de tamanho no máximo $\lceil n/2 \rceil$.

Para ordenar $V[1..n]$, vamos dividí-lo na posição mediana, em dois sub-vetores L e R de tamanho aproximadamente igual.

L e R têm no máximo $\lceil n/2 \rceil$ elementos e então sabemos ordená-los.

Podemos intercalar os elementos de L ordenado e de R ordenado para obter um vetor ordenado com os mesmos elementos de V.

Merge-Sort

Merge-Sort(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    Merge-Sort(A, p, q)
    Merge-Sort$(A, q + 1, r)$
    Merge(A, p, q, r)

Intercalação

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
for i = 1 to n1
        L[i] = A[p + i − 1]
L[n1 + 1] = +∞
for j = 1 to n2
        R[j] = A[q + j]
R[n2 + 1] = +∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

- MERGE-SORT é um algoritmo de tempo ótimo, $O(n \log n)$.

- Outra idéia é: ao invés de resolver o problema diretamente, remove-se alguns elementos obtendo-se uma instância menor do mesmo problema, resolve-se recursivamente e depois aumenta-se a solução adicionando-se os elementos removidos.

Celebridade

- Uma celebridade é uma pessoa que é conhecida por todas as outras pessoas e que não conhece nenhuma delas.
- O problema é determinar se existe uma celebridade em um conjunto de n pessoas fazendo perguntas da forma "Você conhece aquela pessoa?", supondo que todos respondem e ninguém mente.

- Uma solução é perguntar para cada pessoa sobre todas as demais.
- Essa solução faz até $n(n-1)$ perguntas.

Se há apenas uma pessoa então ela é celebridade.

Se há duas ou mais pessoas, vamos escolher duas pessoas i e j. Com certeza, entre i e j uma delas não é celebridade. Basta uma pergunta para determinar qual delas. Vamos supor que seja i.

Vamos supor que sabemos resolver o problema sem a pessoa i.

Resolvido esse problema de tamanho $n-1$, ou não há celebridade nesse conjunto ou a celebridade é uma pessoa k.

Vamos aumentar a solução incluindo a pessoa i, que não é celebridade:

– se não há celebridade dentre as $n-1$ pessoas, não há celebridade entre as n pessoas;

– se k é celebridade dentre as $n-1$ pessoas mas k conhece i ou i não conhece k então k não é celebridade dentre as n pessoas. Senão é.

Algoritmo

CELEBRIDADE(S)

if $|S| == 1$
    $k = 1$
else
    Sejam i, j quaisquer duas pessoas em S
    if i não conhece j
        $i = j$
    $S' = S \setminus \{i\}$
    $k =$ CELEBRIDADE(S')
    if $k > 0$ and (k conhece i or i não conhece k)
        $k = 0$
return k

Solução recursiva

- Fazemos até $3(n-1)$ perguntas.
- Melhor que $n(n-1)$ perguntas se fizermos todos-contra-todos.

Ordenação

- Queremos ordenar o vetor $V[1\,..\,n]$.
- Se $n = 1$, o vetor já está ordenado.
- Se $n > 1$, vamos remover o último elemento, $V[n]$, e ordenar o vetor $V[1\,..\,n-1]$ recursivamente.
- Para aumentar a solução e incluir $V[n]$, deslocamos os elementos de $V[1\,..\,n-1]$ uma posição para a direita até encontrar a posição k adequada para adicionar $V[n]$. Colocamos $V[n]$ na posição k.
- O vetor $V[1\,..\,n]$ está ordenado.

$\textsc{Sort}(A[1..n])$

if $n == 1$
    return
$\textsc{Sort}(A[1..n-1])$
$key = A[n]$
$i = n - 1$
while $i > 0$ and $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
$A[i+1] = key$

Plataforma

- Nem sempre a recursão avança sobre elementos concretos do problema, mas sobre alguma propriedade do problema.
- P.ex. vamos supor um jogo em que o personagem pode dar saltos para sair de uma plataforma unidimensional.
- O problema é determinar se o personagem consegue sair da plataforma ou não, a partir de dada posição inicial do personagem.
- A plataforma pode ser representada por um vetor da intensidade máxima do salto em cada posição.

```
0 1 2 2 0 6 0 2 2 1 0
posição inicial: 1
```

```
0 1 1 3 4 5 4 3 2 1 0
posição inicial: 2
```

- Uma recursão possível para esse problema não é reduzindo o tamanho da plataforma, mas reduzindo o número de *posições diferentes de zero* na plataforma.
- Essa propriedade não está na descrição do problema, mas faz parte da estrutura do problema: se o personagem volta para uma posição onde já esteve então ele está em loop.

```
int check(int* pltf, int n, int i) {

  if (i < 0 || i >= len(pltf))
    return 1;

  if (pltf[i] == 0)
    return 0;

  t = pltf[i];
  pltf[i] = 0;
  for (k=i-t; k<=i+t; k++)
    if check(pltf, n, k)
      return 1;
  return 0;
}
```

- Há várias estruturas-de-dados que são recursivas: partes da estrutura-de-dados têm as mesmas propriedades da estrutura original.

Listas encadeadas

- Se removermos o primeiro elemento de uma lista encadeada com n nós ficamos com uma lista encadeada com $n - 1$ nós.
- É fácil resolver problemas em uma lista encadeada com apenas um nó ou vazia.

Calcular o tamanho de uma lista

```c
typedef struct node {
  int data;
  struct node* next;
} node;


int length_rec(node* p) {

  if (!p)
    return 0;

  return (1 + length_rec(p->next));
}
```

Copiar uma lista

```c
void copy_rec(node* list, node** copy) {

  if (list == NULL)
    *copy = NULL;
  else {
    copy_rec(list->next, copy);

    node* p = (node*) malloc(sizeof(node));
    p->data = list->data;

    p->next = *copy;
    *copy = p;
  }
}
```

função recursiva

- Uma função recursiva é uma função que tem uma chamada para ela mesma.

- Toda função recursiva tem dois elementos:
    1. Um ou mais casos-base, que são os casos resolvidos diretamente, sem chamadas recursivas.
    2. Um ou mais casos-recursivos, que relacionam o valor da função com valores da mesma função com pelo menos um dos parâmetros com valor diferente.

- Um exemplo tradicional é o problema de calcular o valor da função fatorial de um inteiro positivo n.

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n.$$

- A solução recursiva tem duas partes:

1. se $n = 0$, o valor da fatorial é 1
2. senão, para calcular $n!$ multiplicamos n por $(n-1)!$

```
int fat(int n) {
  if (n == 0)
    return 1;
  else
    return n * fat(n-1);
}
```

- É fácil ver que se $n = 0$ então a função produz a resposta correta.
- E é fácil ver que se a função produz a resposta correta para $n = 0$ então ela produz a resposta correta para $n = 1$.
- E é fácil ver que se a função produz a resposta correta para $n = 1$ então ela produz a resposta correta para $n = 2$.
- E assim por diante.
- De forma geral, se a função produz a resposta correta para $n - 1$ então ela produz a resposta correta para n.

- Outro exemplo tradicional de função recursiva é a de Fibonacci:

```c
int fib(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

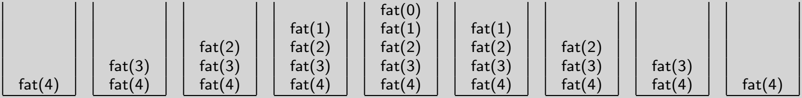
- Quando uma função recursiva é executada, as chamadas para ela vão ficando suspensas até que um caso-base seja executado.
- Depois, os valores vão sendo retornados e as funções vão sendo continuadas, sucessivamente.

```
fatorial(5)
|  fatorial(4)
|  |  fatorial(3)
|  |  |  fatorial(2)
|  |  |  |  fatorial(1)
|  |  |  |  |  fatorial(0)
|  |  |  |  |  return 1
|  |  |  |  return 1*1 = 1
|  |  |  return 2*1 = 2
|  |  return 3*2 = 6
|  return 4*6 = 24
return 5*24 = 120
```

Registros de ativação

- Quando uma função é chamada, um *registro de ativação* é criado.
- Um registro de ativação armazena os parâmetros e as variáveis locais, o endereço da memória onde o retorno da função deve ser escrito e o endereço da instrução que deverá ser executada quando a função terminar.

- Durante a execução de uma função recursiva, cada chamada recursiva acrescenta um registro de ativação na stack.
- Essa memória é parte da memória usada pelo programa.



- Se a profundidade das chamadas recursivas for muito grande, a stack vai ficar cheia e o programa vai ser terminado.

- Cada chamada de função durante a execução de uma função recursiva consome um pouco de tempo e usa um pouco de memória.
- De modo geral isso não é crítico, mas pode ser importante para programas que precisam ser bem otimizados.

Recursão de cauda

- Seja F uma função que chama G (que pode ser igual a F).
  A chamada para G é uma chamada de cauda se F retorna o valor retornado por G sem realizar computação adicional.
- Uma função recursiva F é uma recursão de cauda se todas as chamadas recursivas em F são chamadas de cauda.

```c
int mdc(int a, int b) {
  if (b == 0)
    return a;
  return mdc(b, a%b);
}
```

```
mdc(525,705)
  mdc(705,525)
    mdc(525,180)
      mdc(180,165)
        mdc(165,15)
          mdc(15,0)
          return 15
        return 15
      return 15
    return 15
  return 15
return 15
```

- A recursão de cauda pode ser executada usando um único registro de ativação, que vai sendo atualizado ao longo da seqüência de chamadas.
- Muitos compiladores e interpretadores detectam recursões de cauda e fazem esse tipo de otimização.

Fatorial

- A fatorial não é uma recursão de cauda.

```
int fat(int n) {
  if (n == 0)
    return 1;
  else
    return n * fat(n-1);
}
```

- Uma outra forma de implementar a função fatorial é como uma recursão de cauda:

```c
int fatrc(int n, int res) {
  if (n == 1)
    return res;
  else
    return fatrc(n-1, n * res);
}
```

- `fatrc(n,1)` retorna $n!$.

- `fat(n)` faz $n-1$ chamadas recursivas.
- `fatrc(n,1)` também:

```
fatrc(n-1,n*1),
fatrc(n-2,(n-1)*n*1),
fatrc(n-3,(n-2)*(n-1)*n*1),
...,
fatrc(2,3*...*(n-1)*n*1).
fatrc(1,2*3*...*(n-1)*n*1).
```

- É fácil ver que o valor retornado pela chamada `fatrc(n,1)` será o mesmo que o retornado pela última chamada recursiva, `fatrc(1,2*...*(n-1)*n*1)`.

fat(1)	$\hookleftarrow$ 1	fatrc(1,120)	$\hookleftarrow$ 120
fat(2)	$\hookleftarrow$ 2	fatrc(2,120)	$\hookleftarrow$ 120
fat(3)	$\hookleftarrow$ 6	fatrc(3,60)	$\hookleftarrow$ 120
fat(4)	$\hookleftarrow$ 24	fatrc(4,20)	$\hookleftarrow$ 120
fat(5)	$\hookleftarrow$ 120	fatrc(5,5)	$\hookleftarrow$ 120

Conversão

- Uma forma genérica de transformar uma função recursiva em recursão de cauda é:
  - Todo o trabalho realizado após a chamada da função é antecipado.
  - Se não for possível (por causa de dependências) então os valores são passados para a função através de parâmetros adicionais e o trabalho é realizado ao longo das chamadas.

- A função Fibonacci com recursão de cauda pode ser construída assim:

```
int fibrc(int n, int t1, int t2) {

  if (n == 0)
    return t1;
  else if (n == 1)
    return t2;
  else
    return fibrc(n-1,t2,t1+t2);
}
```

- A chamada `fibrc(n,0,1)` retorna o n-ésimo número da série.

Eliminação de recursão

- A partir de uma função recursiva sempre é possível escrever uma função equivalente sem recursão.
- A idéia geral é usar uma pilha para simular a stack e os registros de ativação.

```
FIB-SEQUENCE(n)
Let S be an empty stack
PUSH(S, 0)
PUSH(S, 1)
n = n − 2
while n > 0
    x = POP(S)
    y = POP(S)
    PUSH(S, y)
    PUSH(S, x)
    PUSH(S, x + y)
    n = n − 1
```

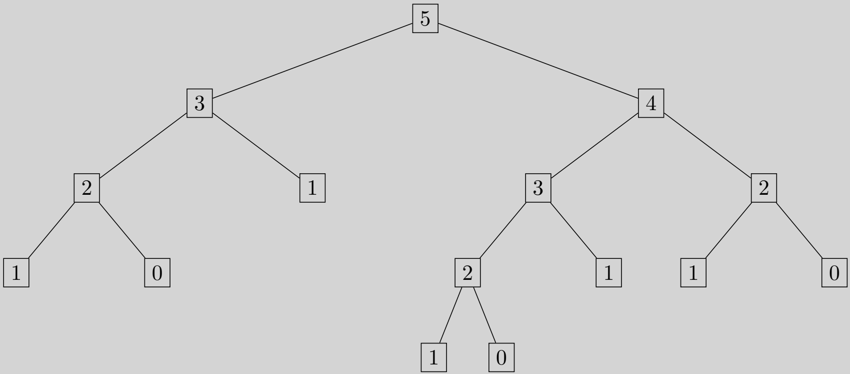
- Um esquema interessante e geral de eliminação de recursão aparece na apostila
  C.L. Lucchesi e T. Kowaltowski. Estruturas de dados e técnicas de programação, 2004.

Sobreposição de subproblemas

- Algumas soluções recursivas reduzem o problema de forma que os subproblemas têm sobreposição.
- Por exemplo, a função de Fibonacci recursiva.

```
int fib(int n) {
  if (n == 0)
    return 1;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Recursão para $n = 5$



- É fácil ver que a recursão resolve o mesmo problema mais de uma vez.
- Essa característica é crítica: o número de chamadas recursivas e o tempo de execução do programa pode aumentar exponencialmente embora o número de sub-problemas seja polinomial.
- Essa situação aparece em vários problemas que são resolvidos recursivamente.

- Algumas técnicas para lidar com esses casos são:
  - Eliminar a recursão totalmente, se for possível.
  - Memorizar subproblemas que já foram resolvidos (técnica chamada de memorização ou programação-dinâmica top-down).
  - Resolver os subproblemas em outra ordem (técnica chamada de programação-dinâmica bottom-up).

Recursão mútua

- Pode haver recursão indireta (ou recursão mútua) quando uma função F chama uma função G, que por sua vez chama F.
- Um exemplo tradicional de recursão mútua são as $f(n) = n \bmod 2$ e $g(n) = (n+1) \bmod 2$:

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ g(n-1) & \text{se } n > 0 \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{se } n = 0 \\ f(n-1) & \text{se } n > 0 \end{cases}$$

- Não necessariamente um algoritmo recursivo vai ser mais eficiente quando for implementado em um programa (mais detalhes depois).
- Esse é o caso p.ex. para fatorial e fibonacci; os programas iterativo são muito simples.
- Mas para muitos problemas a recursão fornece uma solução mais simples e mais eficiente. Os problemas do tabuleiro e da celebridade são exemplos.
- Mesmo se a implementação recursiva não for mais eficiente, diversos problemas têm uma natureza recursiva e a recursão é uma ferramenta importante para pensar sobre eles.

- Não dá para escapar da recursão.