MC-202
Revisão de recursão

Lehilton Pedrosa
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

- Primeiro, definimos as soluções para casos básicos
- Em seguida, tentamos reduzir o problema para instâncias menores do problema
- Finalmente, combinamos o resultado das instâncias menores para obter um resultado do problema original

Genericamente

Caso base:

- resolve instâncias pequenas diretamente

Caso geral:

- reduz o problema para instâncias menores do mesmo problema
- chama a função recursivamente

```c
int fat(int n) {
  if (n == 0) /* caso base */
    return 1;
  else /* caso geral */
    return n * fat(n - 1); /* instância menor */
}
```

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão
  - multiplicação, divisão, exponenciação, etc...

Isso nos permite projetar algoritmos para lidar com essas operações/objetos

Ex: Exponenciação

Seja a é um número real e b é um número inteiro não-negativo

- Se $b = 0$, então $a^b = 1$
- Se $b > 0$, então $a^b = a \cdot a^{b-1}$

```
double potencia(double a, int b) {
  if (b == 0)
    return 1;
  else
    return a * potencia(a, b - 1);
}
```

Palíndromos

Uma palavra é um palíndromo se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
  - α é uma letra
  - p é um palíndromo

```
int eh_palindromo(char *palavra, int ini, int fim) {
  if (ini >= fim)
    return 1;
  return palavra[ini] == palavra[fim] &&
         eh_palindromo(palavra, ini + 1, fim - 1);
}

eh_palindromo(palavra, 0, strlen(palavra) - 1);
```

Busca Binária

Para buscar `x` no vetor ordenado `dados` entre as posições `l` e `r`

Casos base:

- Se o intervalo for vazio (`l > r`), `x` não está no vetor
- Se `dados[m] == x`, onde `m = (l + r)/2`
  - Devolvemos `m`

Caso geral:

- Se `dados[m] < x`, então `x` só pode estar entre `m + 1` e `r`
  - Devolvemos o resultado da chamada recursiva
- Se `dados[m] > x`, então `x` só pode estar entre `l` e `m - 1`
  - Devolvemos o resultado da chamada recursiva

```
int busca_binaria(int *dados, int l, int r, int x) {
  int m = (l + r)/2;
  if (l > r)
    return -1;
  if (dados[m] == x)
    return m;
  else if (dados[m] < x)
    return busca_binaria(dados, m + 1, r, x);
  else
    return busca_binaria(dados, l, m - 1, x);
}
```

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Isso sempre é possível? Quando for possível, sempre melhora a eficiência do algoritmo?

- Veremos mais sobre isso no curso...

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: $1, 1, 2, 3, 5, 8, 13, \ldots$

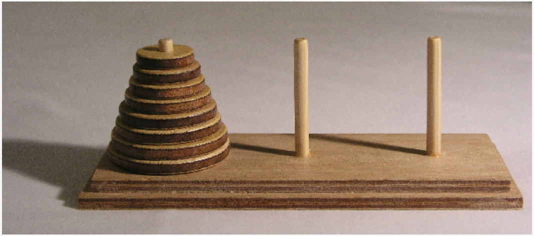
```c
int fib_rec(int n) {
  if (n == 1)
    return 1;
  else if (n == 2)
    return 1;
  else
    return fib_rec(n-2) +
           fib_rec(n-1);
}
```

```c
int fib_iterativo(int n) {
  int ant, atual, prox, i;
  ant = atual = 1;
  for (i = 3; i <= n; i++) {
    prox = ant + atual;
    ant = atual;
    atual = prox;
  }
  return atual;
}
```

Número de operações:

- iterativo: $\approx n$
- recursivo: $\approx \texttt{fib}(n)$ (aproximadamente 1.6^n)

Torres de Hanói



A torre de Hanói é um brinquedo com três estacas A, B e C e discos de tamanhos diferentes

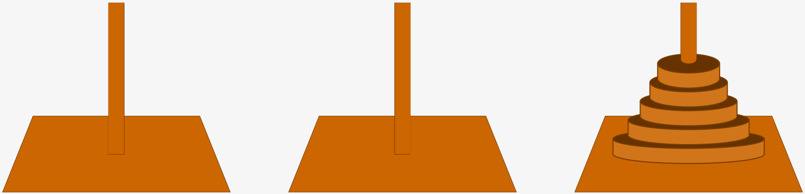
Objetivo:

- mover todos os discos da estaca A para a estaca C

Regras:

- Apenas um disco pode ser movido de cada vez
- Um disco maior não pode ser colocado sobre um menor

Torres de Hanói recursivo




```c
void hanoi(int n, char orig, char dest, char aux) {
  /* caso base: n == 0 - não faz  nada */
  if (n > 0) {  /* caso geral */
    hanoi(n - 1, orig, aux, dest);
    printf("move de %c para %c\n", orig, dest);
    hanoi(n - 1, aux, dest, orig);
  }
}
```

Chamada da função: `hanoi(n, 'a', 'c', 'b');`

Exercício — Calculando o Máximo

Escreva uma função recursiva que calcule o máximo de um vetor dado com n elementos

```
int maximo(int *v, int n)
```

Solução

```c
int maximo(int *v, int n) {
    if (n == 1)
        return v[0];
    int m = maximo(v, n - 1);
    return v[n - 1] > m ? v[n - 1] : m;
}
```

Exercício — Coeficientes Binomiais

Escreva uma função recursiva que calcule, para $n \geq 0$ e $k \geq 0$

$$\binom{n}{k}$$

Relação de Stifel:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$$

Solução

```c
int binomio(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    return binomio(n - 1, k) + binomio(n - 1, k - 1);
}
```

Dúvidas?