MC102 – Aula 27
Recursão II

Instituto de Computação – Unicamp

17 de Novembro de 2016

Roteiro

Recursão - Relembrando

- Definições recursivas de funções são baseadas no *princípio matemático da indução* que vimos anteriormente.
- A idéia é que a solução de um problema pode ser expressa da seguinte forma:
  - Definimos a solução para os casos básicos;
  - Definimos como resolver o problema geral utilizando soluções do mesmo problema só que para casos menores.

Cálculo de Potências

Suponha que temos que calcular x^n para n inteiro positivo. Como calcular de forma recursiva?

x^n é:

- 1 se $n = 0$.
- xx^{n-1} caso contrário.

Cálculo de Potências

```
long pot(long x, long n){
  if(n == 0)
    return 1;
  else
    return x*pot(x,n-1);
}
```

Cálculo de Potências

Neste caso a solução iterativa é mais eficiente.

```
long pot(long x, long n){
    long p = 1, i;
    for( i=1; i<=n; i++)
        p = p * x;
    return p;
}
```

- O laço é executado *n* vezes.
- Na solução recursiva são feitas *n* chamadas recursivas, mas tem-se o custo adicional para criação/remoção de variáveis locais na pilha.

Cálculo de Potências

Mas e se definirmos a potência de forma diferente?

x^n é:

- Caso básico:
  - Se $n = 0$ então $x^n = 1$.
- Caso Geral:
  - Se $n > 0$ e é par, então $x^n = (x^{n/2})^2$.
  - Se $n > 0$ e é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note que aqui também definimos a solução do caso maior em termos de casos menores.

Cálculo de Potências

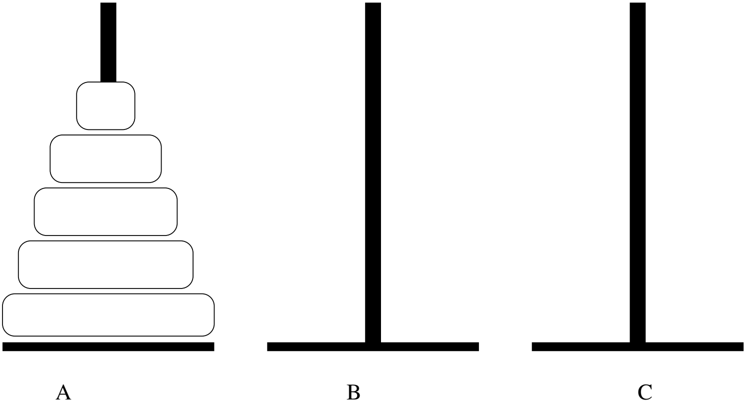
Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
long pot(long x, long n){
    double aux;
    if(n == 0)
      return 1;

    else if(n%2 == 0){ //se n é par
      aux = pot(x, n/2);
      return aux * aux;
    }

    else{ //se n é impar
      aux = pot(x, (n-1)/2);
      return x*aux*aux;
    }
}
```

Cálculo de Potências

- No algoritmo anterior, a cada chamada recursiva o valor de n é dividido por 2. Ou seja, a cada chamada recursiva, o valor de n decai para pelo menos a metade.
- Usando divisões inteiras faremos no máximo $\lceil(\log_2 n)\rceil + 1$ chamadas recursivas.
- Enquanto isso, o algoritmo iterativo executa o laço n vezes.

Torres de Hanoi



Torres de Hanoi

- Inicialmente temos 5 discos de diâmetros diferentes na estaca A.
- O problema das torres de Hanoi consiste em transferir os cinco discos da estaca A para a estaca C (pode-se usar a estaca B como auxiliar).
- Porém deve-se respeitar as seguintes regras:
  - Apenas o disco do topo de uma estaca pode ser movido.
  - Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.

Torres de Hanoi

- Vamos considerar o problema geral onde há n discos.
- Vamos usar indução para obtermos um algoritmo para este problema.

Torres de Hanoi

Teorema

É possível resolver o problema das torres de Hanoi com n discos.

Prova.

- Base da Indução: $n = 1$. Neste caso temos apenas um disco. Basta mover este disco da estaca A para a estaca C.
- Hipótese de Indução: Sabemos como resolver o problema quando há $n - 1$ discos.

□

Torres de Hanoi

Prova.

- Passo de Indução: Devemos resolver o problema para n discos assumindo que sabemos resolver o problema com $n-1$ discos.
  - Por hipótese de indução sabemos mover os $n-1$ primeiros discos da estaca **A** para a estaca **B** usando a estaca **C** como auxiliar.
  - Depois de movermos estes $n-1$ discos, movemos o maior disco (que continua na estaca **A**) para a estaca **C**.
  - Novamente pela hipótese de indução sabemos mover os $n-1$ discos da estaca **B** para a estaca **C** usando a estaca **A** como auxiliar.
- Com isso temos uma solução para o caso onde há n discos.

$\square$

Torres de Hanoi: Passo de Indução

- A indução nos fornece um algoritmo e ainda por cima temos uma demonstração formal de que ele funciona!

Torres de Hanoi: Algoritmo

Problema: Mover n discos de **A** para **C**.

1. Se $n = 1$ então mova o único disco de **A** para **C** e pare.
2. Caso contrário ($n > 1$) desloque de forma recursiva os $n - 1$ primeiros discos de **A** para **B**, usando **C** como auxiliar.
3. Mova o último disco de **A** para **C**.
4. Mova, de forma recursiva, os $n - 1$ discos de **B** para **C**, usando **A** como auxiliar.

Torres de Hanoi: Algoritmo

- A função que computa a solução em C terá o seguinte protótipo:

```
void hanoi(int n, char estacaIni, char estacaFim, char estacaAux);
```

- É passado como parâmetro o número de discos a ser movido (**n**), e um caracter indicando de onde os discos serão movidos (**estacaIni**); para onde devem ser movidos (**estacaFim**); e qual é a estaca auxiliar (**estacaAux**).

Torres de Hanoi: Algoritmo

A função que computa a solução é:

```
void hanoi(int n, char estacaIni, char estacaFim, char estacaAux){
 if(n==1) //Caso base. Move único disco do Ini para Fim
    printf("\nMova disco %d da estaca %c para %c.", n,  estacaIni, estacaFim);
 else{
     //Move n-1 discos de Ini para Aux com Fim como auxiliar
    hanoi(n-1,estacaIni,estacaAux,estacaFim);

    //Move maior disco para Fim
    printf("\nMova disco %d da estaca %c para %c.", n,  estacaIni, estacaFim);

    //Move n-1 discos de Aux para Fim com Ini como auxiliar
    hanoi(n-1,estacaAux,estacaFim,estacaIni);
  }
}
```

Torres de Hanoi: Algoritmo

```c
#include <stdio.h>

void hanoi(int n, char estacaIni, char estacaFim, char estacaAux);

int main(){
  hanoi(4, 'A', 'C' , 'B');
  printf("\n");
}

//Discos são numerados de 1 até n

void hanoi(int n, char estacaIni, char estacaFim, char estacaAux){
 if(n==1)
     printf("\nMova disco %d da estaca %c para %c.", n,  estacaIni, estacaFim);
 else{
     hanoi(n-1,estacaIni,estacaAux,estacaFim);
     printf("\nMova disco %d da estaca %c para %c.", n,  estacaIni, estacaFim);
     hanoi(n-1,estacaAux,estacaFim,estacaIni);
  }
}
```

Recursão e Backtracking

- Muitos problemas podem ser resolvidos enumerando-se de forma sistemática todas as possibilidades de arranjos que formam uma solução para um problema.
- Vimos em aulas anteriores o seguinte exemplo: Determinar todas as soluções inteiras de um sistema linear como:

$$x_1 + x_2 + x_3 = C$$

com $x_1 \geq 0$, $x_2 \geq 0$, $C \geq 0$ e todos inteiros.

```
Para cada possível valor de x1 entre 0 e C
    Para cada possível valor de x2 entre 0 e C−x1
               Faça  x3 = C − (x1 + x2)
               Imprima solução x1 + x2 + x3 = C
```

Recursão e Backtracking

Abaixo temos o código de uma solução para o problema com $n = 3$ variáveis e constante C passada como parâmetro.

```
void solution(int C){
  int x1, x2, x3;

  for(x1=0; x1 <= C; x1++){
    for(x2=0; x2 <= C-x1; x2++){
      x3 = C -x1 -x2;
      printf("%d + %d + %d = %d\n", x1, x2, x3, C);
    }
  }
}
```

Recursão e Backtracking

Como resolver este problema para o caso geral, onde n e C são parâmetros?

$$x_1 + x_2 + \ldots + x_{n-1} + x_n = C$$

- A princípio deveríamos ter $n - 1$ laços encaixados.
- Mas não sabemos o valor de n. Só saberemos durante a execução do programa.

Recursão e Backtracking

- A técnica de recursão pode nos ajudar a lidar com este problema:
  - Construir uma função com um único laço e que recebe uma variável k como parâmetro.
  - A variável k indica que estamos setando os possíveis valores de x_k.
  - Para cada valor de x_k devemos setar o valor de x_{k+1} de forma recursiva!
  - Se $k == n$ basta setar o valor da última variável.

Recursão e Backtracking

```
função solution(n, C, k){
  Se k == n Então
     x_n = C - x_1 - ... - x_{n-1}
     Imprima solução
  Senão
     Para cada valor V entre 0 e (C - x_1 - ... - x_{k-1}) faça
        x_k = V
        solution(n, C, k+1) //Vamos setar os possíveis valores da var. x seguinte
}
```

Recursão e Backtracking

- Em C teremos uma função com o seguinte protótipo:

```c
void solution(int n, int C, int k, int R, int x[])
```

- A variável R terá o valor da constante C menos os valores já setados para variáveis em chamadas recursivas anteriores, i.e, $R = C - x_1 - \ldots - x_{k-1}$.
- O vetor x corresponde aos valores das variáveis.
  - Lembre-se que em C o vetor começa na posição 0, por isso as variáveis serão $x[0], \ldots, x[n-1]$.

Recursão e Backtracking

- Primeiramente temos o caso de parada (quando $k == n - 1$):

```
void solution(int n, int C, int k, int R, int x[]){
  if(k == n-1){
    int i;
    //imprimindo a solução
    for(i=0; i<= n-2; i++){
      printf("%d + ", x[i]);
    }
    printf("%d = %d\n", R, C); //R é o valor de x[n-1]
    return;
  }
  .
  .
  .
}
```

Recursão e Backtracking

- A função completa é:

```
void solution(int n, int C, int k, int R, int x[]){
  if(k == n-1){
    int i;
    for(i=0; i<= n-2; i++){
      printf("%d + ", x[i]);
    }
    printf("%d = %d\n", R, C);
    return;
  }

  for(x[k]=0; x[k]<=R; x[k]++){
    solution(n, C, k+1, R-x[k], x);
  }
}
```

A chamada inicial da função deve ter $k = 0$.

Recursão e Backtracking

```
#include <stdio.h>
#include <stdlib.h>

void solution(int n, int C, int k, int R, int x[]);

int main(int argc, char *argv[]){
  if(argc != 3){
    printf("Execute informando n (num. de vari.) e C (constante int. positiva)\n");
    return 0;
  }
  int n = atoi(argv[1]);
  int C = atoi(argv[2]);
  int *x = malloc(n * sizeof(int));
  solution(n, C, 0, C, x);
  free(x);
}

void solution(int n, int C, int k, int R, int x[]){
  if(k == n-1){
    int i;
    for(i=0; i<= n-2; i++){
      printf("%d + ", x[i]);
    }
    printf("%d = %d\n", R, C);
    return;
  }
  for(x[k]=0; x[k]<=R; x[k]++){
    solution(n, C, k+1, R-x[k], x);
  }
}
```

Exercício

- Defina de forma recursiva a busca binária.
- Escreva um algoritmo recursivo para a busca binária.

Exercício

- Escreva um programa que lê uma string do teclado e então imprime todas as permutações desta palavra. Se por exemplo for digitado "abca" o seu programa deveria imprimir: aabc aacb abac abca acab acba baac baca bcaa caab caba cbaa