COM111 — Recursão

Hokama

UNIFEI

24 de Setembro de 2021

Disclaimer

Esses slides foram adaptados dos slides elaborados pelo Prof. Eduardo C. Xavier para a disciplina MC102 da UNICAMP.

Roteiro

Recursão – Indução

- Devemos criar uma algoritmo para resolver um determinado problema.
- Usando o método de recursão/indução, a solução de um problema pode ser expressa da seguinte forma:
  - Primeiramente, definimos a solução para casos básicos;
  - Em seguida, definimos como resolver o problema para um caso geral, utilizando-se de soluções para instâncias menores do problema.

Indução

- **Indução:** Técnica de demonstração matemática onde algum parâmetro da proposição a ser demonstrada envolve números naturais.
- Seja T uma proposição que desejamos provar como verdadeira para todos valores naturais n.
- Ao invés de provar diretamente que T é válida para todos os valores de n, basta provar as duas condições 1 e 3 a seguir:
  1. **Passo base:** PROVAR que T é válido para $n = 1$.
  2. **Hipótese de Indução:** Assumimos que T é válido para $n - 1$.
  3. **Passo de Indução:** Sabendo que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- Por que a indução funciona? Por que as duas condições são suficientes?
  - Mostramos que T é valida para um caso base, como $n = 1$.
  - Com o passo da indução, automaticamente mostramos que T é válida para $n = 2$.
  - Como T é válida para $n = 2$, pelo passo de indução, T também é válida para $n = 3$, e assim por diante.

Exemplo

Teorema

A soma $S(n)$ dos primeiros n números naturais é $n(n+1)/2$

Prova.
Base: Para $n = 1$ devemos mostrar que $n(n+1)/2 = 1$. Isto é verdade:

$$1(1+1)/2 = 1.$$

Hip. de Indução: Vamos assumir que é válido para $(n-1)$, ou seja, $S(n-1) = (n-1)((n-1)+1)/2$.
Passo: Devemos mostrar que é válido para n, ou seja, devemos mostrar que $S(n) = n(n+1)/2$. Por definição, $S(n) = S(n-1) + n$ e por hipótese $S(n-1) = (n-1)((n-1)+1)/2$, logo

$$\begin{aligned} S(n) &= S(n-1) + n \\ &= (n-1)((n-1)+1)/2 + n \\ &= n(n-1)/2 + 2n/2 \\ &= n(n+1)/2 \end{aligned}$$

Recursão

- Definições recursivas de funções funcionam como o *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 casos básicos;
  - Definimos como resolver o problema geral utilizando soluções do mesmo problema só que para casos menores.

Fatorial

Problema: Calcular o fatorial de um número ($n!$).
Qual o caso base e o passo da indução?

- Se n é igual a 1, então o fatorial é 1.

Qual seria o passo indutivo?

- Temos que expressar a solução para $n > 1$, supondo que já sabemos a solução para algum caso mais simples.
- $n! = n * (n-1)!$.

Este caso é trivial pois a própria definição do fatorial é recursiva.

Fatorial

Portanto, a solução do problema **pode ser expressa de forma recursiva** como:

- Se $n = 1$ então $n! = 1$.
- Se $n > 1$ então $n! = n * (n-1)!$.

Note como aplicamos o princípio da indução:

- Sabemos a solução para um caso base: $n = 1$.
- Definimos a solução do problema geral $n!$ em termos do mesmo problema só que para um caso menor $(n-1)!$.

Fatorial em C

```
long fatr(long n){
    long x,r;
    if(n == 1) //Passo Básico
      return 1;
    else{
      x = n-1;
      r = fatr(x); //Sabendo o fatorial de (n-1)
      return (n* r);  //calculamos o fatorial de n
    }
}
```


Fatorial

- Para solucionar o problema, é feita uma chamada para a própria função, por isso, esta função é chamada *recursiva*.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

O que acontece na memória

- Precisamos entender como é feito o controle sobre as variáveis locais em chamadas recursivas.
- A memória de um sistema computacional é dividida em alguns segmentos:
  - **Espaço Estático**: Contém as variáveis globais e código do programa.
  - **Heap:** Para alocação dinâmica de memória.
  - **Pilha:** Para execução de funções.

O que acontece na memória

O que acontece na pilha:

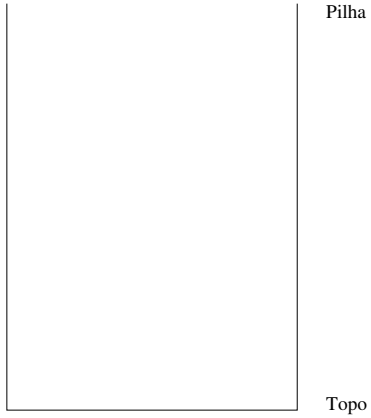
- Toda vez que uma função é invocada, suas variáveis locais são armazenadas no topo da pilha.
- Quando uma função termina a sua execução, suas variáveis locais são removidas da pilha.

Considere o exemplo:

```
int f1(int a, int b){
   int c=5;
   return (c+a+b);
}

int f2(int a, int b){
   int c;
   c = f1(b, a);
   return c;
}

int main(){
  f2(2, 3);
}
```

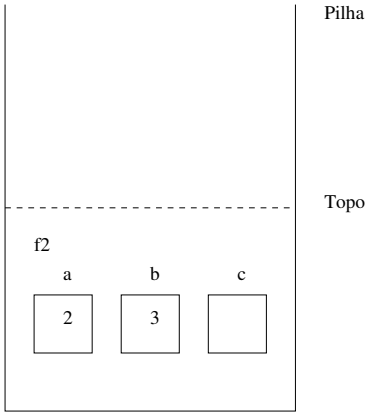

O que acontece na memória

Inicialmente a pilha está vazia.



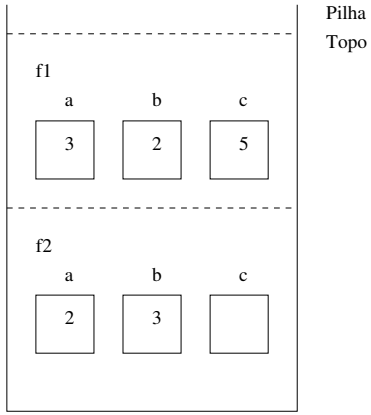
O que acontece na memória

Quando **f2(2,3)** é invocada, suas variáveis locais são alocadas no topo da pilha.



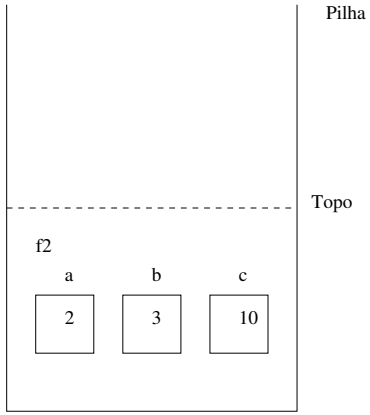
O que acontece na memória

A função **f2** invoca a função **f1(b,a)** e as variáveis locais desta são alocadas no topo da pilha sobre as de **f2**.



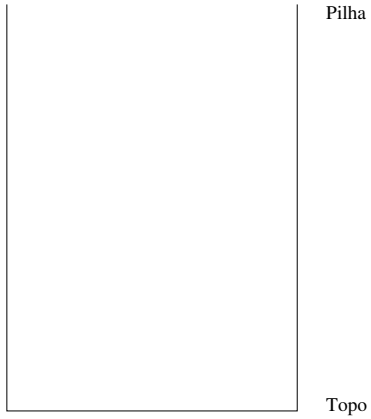
O que acontece na memória

A função **f1** termina, devolvendo 10. As variáveis locais de **f1** são removidas da pilha.



O que acontece na memória

Finalmente **f2** termina a sua execução devolvendo 10. Suas variáveis locais são removidas da pilha.



O que acontece na memória

No caso de chamadas recursivas para uma mesma função, é como se cada chamada correspondesse a uma função distinta.

- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.
- O último conjunto de variáveis alocadas na pilha, que está no topo, corresponde às variáveis da última chamada da função.
- Quando termina a execução de uma chamada da função, as variáveis locais desta são removidas da pilha.

Usando recursão em programação

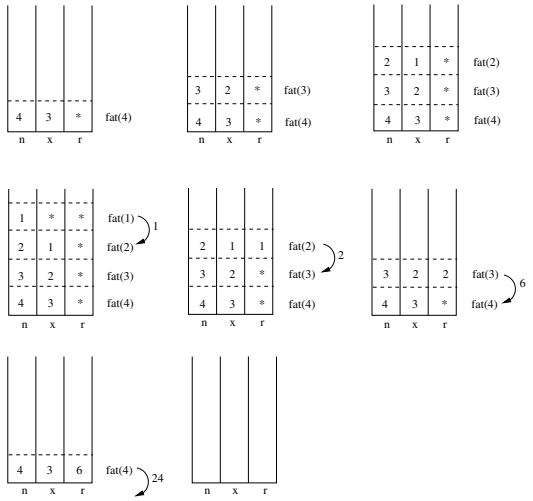
Considere novamente a solução recursiva para se calcular o fatorial e assuma que seja feito a chamada **fatr(4)**.

```
long fatr(long n){
    long x,r;
    if(n == 1) //Passo Básico
      return 1;
    else{
      x = n-1;
      r = fatr(x); //Sabendo o fatorial de (n-1)
      return (n* r);  //calculamos o fatorial de n
    }
}
```


O que acontece na memória

- Cada chamada da função *fatr* cria novas variáveis locais de mesmo nome (n, x, r) .
- Portanto, várias variáveis n, x e r podem existir em um dado momento.
- Em um dado instante, o nome n (ou x ou r) refere-se à variável local ao corpo da função que está sendo executada naquele instante.

fatr(4)



O que acontece na memória

- É claro que as variáveis x e r são desnecessárias.
- Você também deveria testar se n não é negativo!

```
long fatr (long n){
    if(n <= 1) //Passo Básico
      return 1;
    else  //Sabendo o fatorial de (n-1)
          //calculamos o fatorial de n
      return (n* fatr(n-1));
}
```

Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas.
- Soluções iterativas em geral têm a memória limitada enquanto as recursivas, não.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.

Recursão × Iteração

Neste caso, uma solução iterativa é mais eficiente. Por quê?

```
long fat(long n)
{
   long r = 1;

   for(int i = 1; i <= n; i++)
      r = r * i;

   return r;
}
```


Exemplo: Soma de elementos de um vetor

- Dado um vetor v de inteiros de tamanho *tam*, devemos calcular a soma dos seus elementos da posição 0 até $tam - 1$.
- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por $S(n)$ a soma dos elementos das posições 0 até n do vetor, e portanto devemos achar $S(tam - 1)$.
- O valor de $S(n)$ pode ser calculado com a seguinte definição recursiva:
  - Se $n = 0$ então a soma $S(0)$ é igual a $v[0]$.
  - Se $n > 0$ então a soma $S(n)$ é igual a $v[n] + S(n - 1)$.

Algoritmo em C

```
int soma(int v[], int n){
  if(n == 0)
     return v[0];
  else
     return v[n] + soma(v,n-1);
}
```

Algoritmo em C

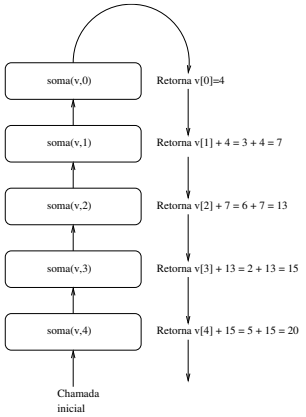
Exemplo de uso:

```c
#include <stdio.h>

int soma(int v[], int n);

int main(){
  int vet[5] = {4,3,6,2,5};
  printf("%d\n", soma(vet, 4));
}

int soma(int v[], int n){
  if(n == 0)
     return v[0];
  else
     return v[n] + soma(v,n-1);
}
```

- Note que na chamada da função o segundo parâmetro é exatamente o índice da última posição do vetor ($tam - 1$).

Exemplo de execução

$V = (4, 3, 6, 2, 5)$



Soma do vetor recursivo

- O método recursivo sempre termina:
  - Existência de um caso base.
  - A cada chamada recursiva do método temos um valor menor de n.

Algoritmo em C

Neste caso, a solução iterativa também seria melhor (não há criação de variáveis das chamadas recursivas):

```c
int calcula_soma(int[] v, int n){
    int soma=0, i;
    for(i=0;i<=n;i++)
       soma = soma + v[i];
    return soma;
}
```

Relembrando

- Recursão é uma técnica para se criar algoritmos onde:
  1. Devemos descrever soluções para casos básicos.
  2. Assumindo a existência de soluções para casos menores, mostramos como obter a solução para o caso maior.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Implementador deve avaliar clareza de código $\times$ eficiência do algoritmo.

Exercício

Mostre a execução da função recursiva **imprime** abaixo:
O que será impresso?

```
#include <stdio.h>

void imprime(int v[], int i, int n);

int main(){
  int vet[] = {1,2,3,4,5,6,7,8,9,10};

  imprime(vet, 0, 9);
  printf("\n");
}


void imprime(int v[], int i, int n){
  if(i==n){
    printf("%d, ", v[i]);
  }
  else{
    imprime(v,i+1,n);
    printf("%d, ", v[i]);
  }
}
```


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

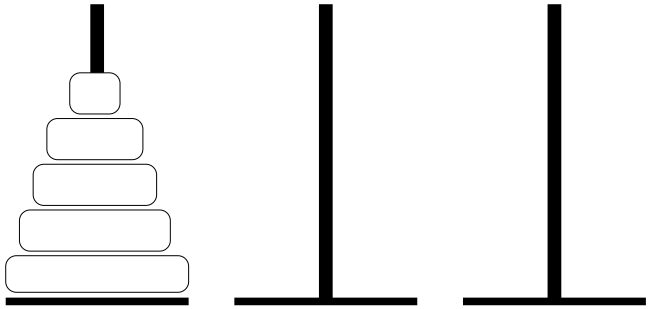
Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
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.

□

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

```
#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 Enumeração

- 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:

$$x_0 + x_1 + x_2 = C$$

com $x_0 \geq 0$, $x_1 \geq 0$, $x_2 \geq 0$, $C \geq 0$ e todos inteiros.

```
Para cada possível valor de x0 entre 0 e C
    Para cada possível valor de x1 entre 0 e C-x0
                Faça  x2 = C - (x0 + x1)
                Imprima solução x0 + x1 + x2 = C
```


Recursão e Enumeração

Abaixo temos o código de uma solução para o problema com $n = 2$ e constante C passada como parâmetro.

```
void solution(int C){
  int x0, x1, x2;

  for(x0=0; x0 <= C; x0++){
    for(x1=0; x1 <= C-x0; x1++){
      x2 = C -x0 -x1;
      printf("%d + %d + %d = %d\n", x0, x1, x2, C);
    }
  }
}
```

Recursão e Enumeração

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

$$x_0 + x_1 + \ldots + x_{n-1} + x_n = C$$

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

Recursão e Enumeração

- A técnica de recursão/indução nos ajuda a lidar com este problema.
- Suponha o problema geral de determinar soluções do problema

$$x_0 + x_1 + \ldots + x_{n-1} + x_n = C$$

  - Se $n = 0$ a única variável deve assumir o valor C ($x_0 = C$).
  - Se $n > 0$ para cada valor possível de x_n, resolvemos recursivamente o problema menor

$$x_0 + x_1 + \ldots + x_{n-1} = C - x_n$$

Recursão e Enumeração

A técnica de recursão pode nos ajudar a lidar com este problema:

- Construir uma função que recebe como parâmetros n, C e as variáveis x como um vetor.
- Se $n = 0$ basta setar o valor da variável $x[0] = C$ e imprimir o vetor solução x.
- Se $n > 0$
  - Dentro de um laço setamos cada valor possível de x_n, e para cada um dos valores setados resolvemos o problema menor recursivamente com parâmetros ($n - 1$, $C - x[n]$, e vetor x).

Recursão e Enumeração

```
função solution(n, C, x){
  Se n == 0 Então
     x[0] = C
     Imprima vetor solução x
  Senão
     Para cada valor V entre 0 e C faça
        x[n] = V
        solution(n, C - x[n] , x) //Resolvemos o prob. menor recursivamente
}
```

Recursão e Enumeração

- Em C teremos uma função com o seguinte protótipo:

```
void solution(int n, int C, int x[], int nOri, int Cori);
```

- Além dos parâmetros (n, C e x[]) já explicados, vamos passar os valores originais de n e C (nOri e COri), que não serão alterados durante a recursão e serão utilizados para imprimir a solução.

Recursão e Enumeração

- Primeiramente temos o caso base (quando $n == 0$):

```
void solution(int n, int C, int x[], int nOri, int COri){
  if(n==0){ //Caso base
    x[0] = C;
    printSol(x, nOri, COri);
  }else{
    .
    .
    .
}
```

- Ao chegar no caso base imprimimos o vetor solução x e para isso precisamos dos valores originais de n e C.

Recursão e Enumeração

- A função completa é:

```
void solution(int n, int C, int x[], int nOri, int COri){
  if(n==0){ //Caso base
    x[0] = C;
    printSol(x, nOri, COri);
  }else{
    int i;
    for(i=0; i<=C; i++){
      x[n] = i; //para cada valor de x[n]
                //resolva o problema menor
      solution(n-1, C - x[n], x, nOri, COri);
    }
  }
}
```



```
#include <stdio.h>
#include <stdlib.h>
void solution(int n, int C, int x[], int nOri, int Cori);
void printSol(int x[], int n, int C);

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+1) * sizeof(int));
  solution(n, C, x, n, C);
  free(x);
}
void printSol(int x[], int n, int C){
  int i, aux=0;
  for(i=0; i<n; i++)
    printf("%d + ", x[i]);
  printf("%d = %d\n", x[n], C);
}
void solution(int n, int C, int x[], int nOri, int COri){
  if(n==0){ //Caso base
    x[0] = C;
    printSol(x, nOri, COri);
  }else{
    int i;
    for(i=0; i<=C; i++){
      x[n] = i; //para cada valor setado de x[n]
                //resolva o problema menor
      solution(n-1, C - x[n], x, nOri, COri);
    }
  }
}
```

Exercício

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