Estruturas de Informação

Recursividade

Departamento de Engenharia Informática (DEI/ISEP)
Fátima Rodrigues
mfc@isep.ipp.pt

Definição

- Um método é recursivo se é definido parcialmente em termos de si próprio

- Recursividade é uma técnica poderosa em definições matemáticas. **O poder da recursividade está na possibilidade de se definir elementos com base em versões mais simples deles mesmos**

  Exemplo: potência positiva (n>=0) de um número X

  $x^n = 1$ se n=0

  $x^n = x * x^{n-1}$ se n>0

- A única ferramenta necessária para expressar operações recursivamente é o próprio procedimento ou a função, que tem a capacidade de se invocar a si próprio

Estrutura de um Algoritmo Recursivo

Os problemas que podem ser resolvidos recursivamente têm normalmente as seguintes características:

- um ou mais casos de paragem, em que a solução é não recursiva e conhecida
- casos em que o problema pode ser diminuído recursivamente até se atingirem os casos de paragem

Estrutura de um Algoritmo Recursivo

se caso de paragem atingido então

    resolver o problema

senão

    fazer uma ou mais invocações recursivas

Factorial

Factorial: 0! = 1

n! = n × (n-1)!

```
long factorial_recurs(long num)
{
   if (num == 0)
      return 1 ;
   else
      return num*factorial_recurs(num-1) ;
}
```

A sequência de chamadas `factorial_recurs(5)` é dada por:

```
5*factorial_recurs(4)
   4*factorial_recurs(3)
      3*factorial_recurs(2)
        2*factorial_recurs(1)
         1*factorial_recurs(0)
                        return 1 ⟶ 1
```

5*24 ↑ 4*6 ↑ 3*2 ↑ 2*1 ↑ 1*1 ↑

```
factorial_recursiv(-1) ?  ⟶  if (num < 0)
                                 break;
```

Soma dos Elementos de um vector

Se definirmos soma(k) como a soma dos valores de v com indices de 1 a k:

soma(0) = 0, k = 0

soma(k) = v[k] + soma(k-1), 1 <= k <= n

```
int soma_vect (const int v[], int dim)
{
    if (dim > 0)
        return v[dim-1] + soma_vect(v,dim-1) ;
    else
        return 0 ;
}
```

Soma dos Elementos de um vector

```
void main ()
{
  int vector[5] = {3,6,7,9,5} ;
  cout << "Soma recursiva dos elementos do vector -> "  ;
  cout << soma_vect(vector,5) << endl ;
}
```

A sequência de chamadas `soma_vect(vector,5)` é dada por:

```
soma_vect(vector,5)
  5 + soma_vect(vector,4)                                    5+25
        9 + soma_vect(vector,3)                          9+16
          7 + soma_vect(vector,2)                   7+9
                6 + soma_vect(vector,1)          6+3
                  3 + soma_vect(vector,0)    3+0
                                  return 0
```

Visualização dos Elementos de um vector

Qual a diferença entre as duas funções abaixo ?

```
void imprime1_vect (const int v[], int dim, int i)
{
   if (i < dim)
   {
        cout << v[i]  << " " ;
        imprime1_vect (v,dim,i+1) ;
   }
}
```

```
void imprime2_vect (const int v[], int dim, int i)
{
   if (i < dim)
   {
     imprime2_vect (v,dim,i+1) ;
     cout << v[i] << " " ;
   }
}
```

Visualização dos Elementos de um vector

```
void main ()
{ int vector[5] = {3,6,7,9,5} ;

  cout << "Imprime 1 -> " ; imprime1_vect(vector, 5, 0) ;
  cout << "Imprime 2 -> " ; imprime2_vect(vector, 5, 0) ;
}
```

Sequência de chamadas imprime1_vect:

```
imprime1_vect(vector,5,0)
  3
  imprime1_vect(vector,5,1)
    6
    imprime1_vect(vector,5,2)
      7
     imprime1_vect(vector,5,3)
       9
      imprime1_vect(vector,5,4)
        5
        imprime1_vect(vector,5,5)
```

Sequência de chamadas imprime2_vect:

```
imprime2_vect(vector,5,0)
  imprime2_vect(vector,5,1)
    imprime2_vect(vector,5,2)
      imprime2_vect(vector,5,3)
        imprime2_vect(vector,5,4)
          imprime2_vect(vector,5,5)
                5
              9
            7
          6
        3
```

Recursividade Indirecta

- Dadas duas funções ***a*** e ***b***,

  recursividade indirecta ocorre quando ***a*** invoca ***b e b*** invoca ***a***

```
bool par (int num)  ;

bool impar (int num)
{
   if (num == 0)
        return false ;
   else
        if (num == 1)
          return true ;
        else
          return par(num-1) ;
}
```

```
bool par (int num)
{
   if (num == 0)
        return true ;
   else
        if (num == 1)
          return false ;
        else
          return impar(num-1) ;
}
```

Sequência Fibonacci

Fibonacci Iteractivo:

```
int fib_iter (int n)
{
   if  (n == 0 || n == 1)
     return n;

   segano=0;
   ano=1;

   for (i = 2; i <= n; i++)
   {
      corrente = segano + ano;
      segano = ano;
      ano = corrente;
   }
   return corrente;
}
```

Fibonacci recursivo:

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

- Apesar da **função recursiva ser mais simples** em termos de código, é claramente ineficiente, pois para calcular um dado elemento da sequencia, cálculos serão repetidos

Permutações

Imprimir todas as permutações de um conjunto N de caracteres

Exemplo:

Conjunto {a,b,c} apresenta

{ (a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a) }

existem N! permutações

Permutações do conjunto {a,b,c,d}, são os quatro seguintes grupos de permutações:

- **a** seguido de permutações do conjunto {b,c,d}
- **b** seguido de permutações do conjunto {a,c,d}
- **c** seguido de permutações do conjunto {a,b,d}
- **d** seguido de permutações do conjunto {a,b,c}

É possível resolver eficientemente o problema para N caracteres se tivermos um **algoritmo recursivo** que funcione para N-1 caracteres

Permutações

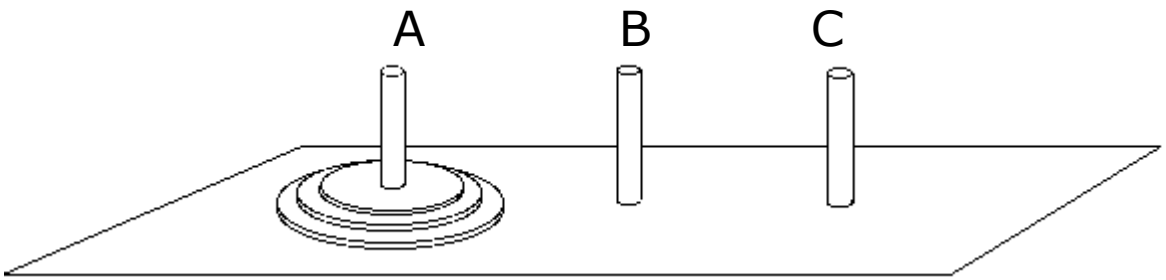
```
void permutacoes (string s, int k)
{
   if (k == s.length()-1)
      cout << s << endl ;
   else
      for (int i = k; i < s.length(); i++)
      {
           char temp;

           temp = s[i] ;
           s[i] = s[k] ;
           s[k] = temp ;
           permutacoes(s,k+1);
      }
}
```

Torres de Hanói

É um problema cuja solução iterativa é muito mais complexa que a correspondente recursiva

É constituído de um conjunto de N discos de tamanhos diferentes todos enfiados na Torre A e três Torres verticais, nos quais os discos podem ser encaixados



O objectivo é mudar os discos para a Torre B, obedecendo às seguintes condições:

- só se pode mudar um disco de cada vez
- só se pode retirar o disco de cima
- nunca se pode colocar um disco sobre outro mais pequeno

Torres de Hanói

- A solução deste problema é trivial caso o número de discos seja 1
- Se tivermos N discos na Torre A a solução consiste em diminuir a complexidade do problema até à situação em que temos apenas 1 disco e para a qual conhecemos a solução

```
Torres-Hanoi (N, TorreA, TorreB, TorreC)
    Se (N = 1)
        Mover disco TorreA → TorreB
    Senão
            Torres-Hanoi (N-1, TorreA, TorreC, TorreB)
            Torres-Hanoi (1, TorreA, TorreB, TorreC)
            Torres-Hanoi (N-1, TorreC, TorreB, TorreA)
```

Consultar http://pt.wikipedia.org/wiki/Torre_de_Hanoi

Backtracking

Em problemas recursivos que envolvam *backtraking* os passos em direção à solução do problema são testados e guardados, mas se estes não conduzirem a uma solução final, **os passos são desfeitos, ou seja, volta-se para trás na recursividade,** e experimentam-se novas possibilidades

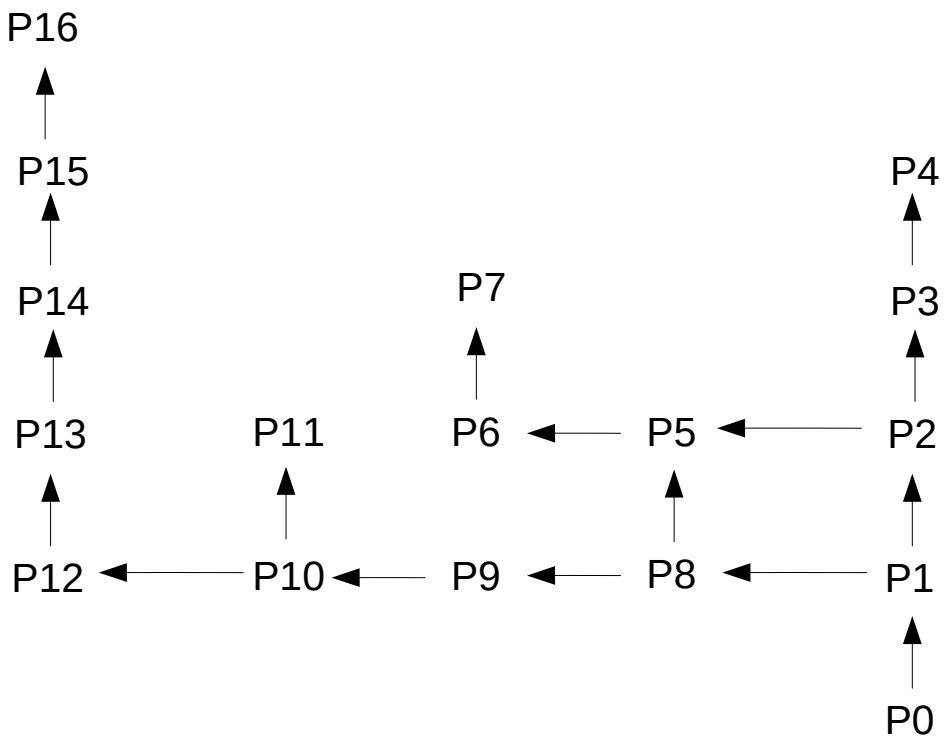
Os passos gerais de qualquer problema recursivo que envolva *backtraking*, assumindo que o número de potenciais candidatos em cada passo é finito, são:

```
Inicializar seleção de candidatos
Repetir
    Selecionar próximo
    Se aceitável
        Guardar
    Se solução incompleta
            Tentar próximo passo
            Se insucesso
                    Cancelar
Até sucesso ou não existir mais candidatos
```

Backtracking

Encontrar um caminho da posição $P_0 \rightarrow P_{16}$

Direcções de movimento: Norte, Oeste



Custo da Recursividade

- A invocação de uma função recursiva produz um **desperdício de tempo e de memória** devido:
  - à criação de uma cópia local dos parâmetros de entrada que são passados por valor
  - à recolha do endereço dos parâmetros passados por referência
  - espaço para guardar variáveis locais
  - bem como a salvaguarda do estado do programa na altura da invocação - memória stack - para que o programa possa retomar a execução na instrução seguinte à invocação da função, quando a execução da função terminar

Esquemas Recursivos não apropriados

- Quando existe uma única chamada do procedimento/função recursivo no fim ou no começo, o procedimento/função é facilmente transformado numa iteração simples

- É boa politica não usar recursividade quando existe um algoritmo iterativo mais simples ou igualmente claro que resolva o problema. É o caso da função Par() vista anteriormente

- Quando o uso de recursividade acarreta num número maior de cálculos, exemplo função Fibonacci

Quando usar Recursividade

- Algoritmos recursivos são apropriados quando o problema a ser resolvido, a função ou os dados, estão definidos em termos recursivos ou por indução

- O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente se comparado com a versão iterativa do mesmo algoritmo

  Exemplos: Permutações, Torres de Hanoi, …