Algoritmos e Estruturas de Dados I

Recursividade

Pedro O.S. Vaz de Melo

Problema

- Implemente uma função que classifique os elementos de um vetor em ordem crescente usando o algoritmo quicksort:

1. Seja m o elemento na posição central no vetor;
2. Seja i o índice do primeiro e j, o índice do último elemento do vetor;
3. Enquanto i for menor ou igual a j, faça com que:

   a) O valor de i cresça até encontrar um elemento maior que m;

   b) O valor de j diminua até encontrar um elemento menor que m;

   c) Haja a troca entre os elementos que ocupam as posições i e j.

Problema

- Ao final desses passos, a situação do vetor será a seguinte:

  à esquerda da posição central, existem somente elementos menores do que m;

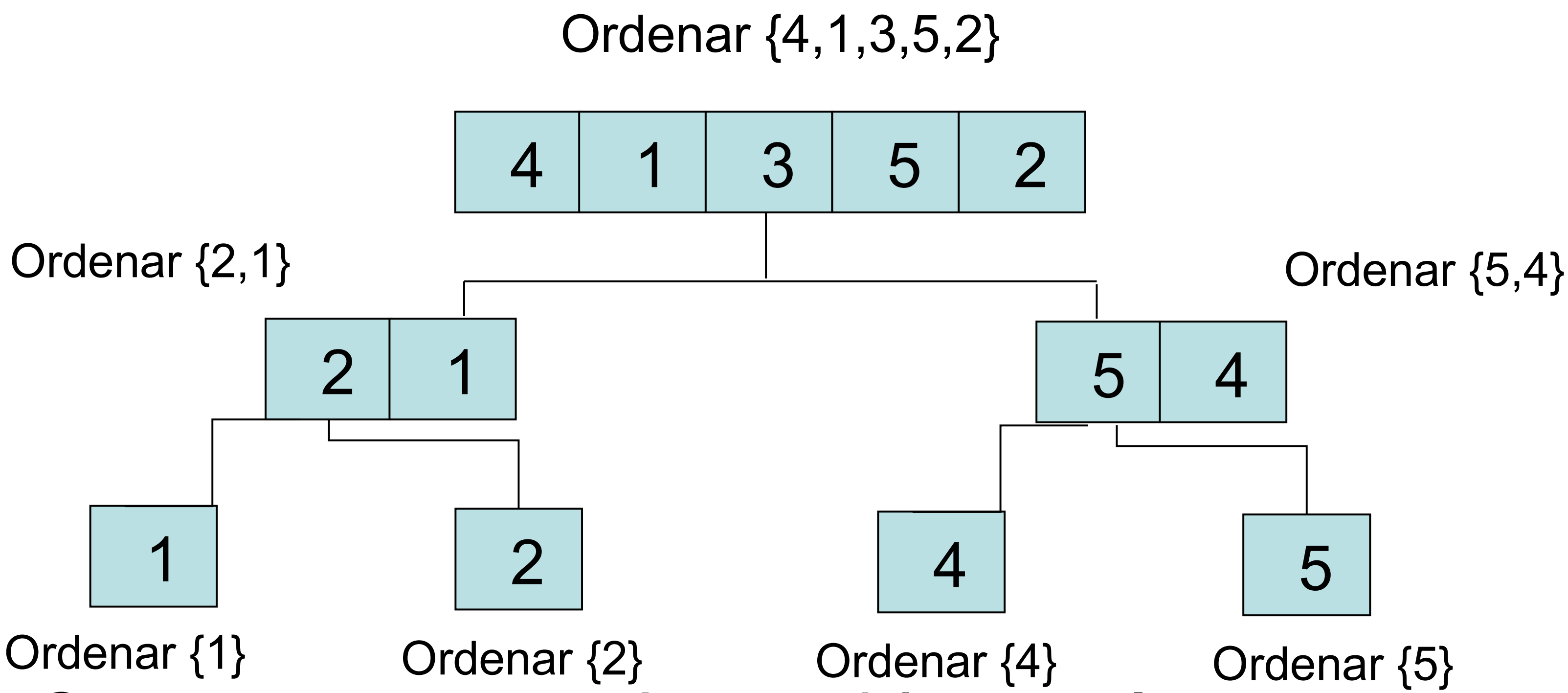
  à direita da posição central, existem somente elementos maiores do que m;

- Assim, o problema de ordenar o vetor se reduz ao problema de ordenar cada uma dessas "metades".

- Os mesmos passos serão aplicadas repetidas vezes à cada nova "metade", até que cada metade contenha um único elemento (caso trivial).

Problema 2

- Considere o exemplo abaixo:



- Como a natureza dos problemas é sempre a mesma ("ordenar um vetor"), o mesmo método pode ser usado para cada subproblema.

Análise do programa

```
void quicksort(int v[], int primeiro, int ultimo)
{
    int i,j;
    int m,aux;

    i = primeiro;
    j = ultimo;
    m = v[(i+j)/2];
    do
    {
      while (v[i] < m) i++;
      while (v[j] > m) j--;
      if (i <= j)
      {
        aux = v[i];
        v[i] = v[j];
        v[j] = aux;
        i++;
        j--;
      }
    }
    while (i <= j);
    if (primeiro < j)
      quicksort(v,primeiro,j);
    if (ultimo > i)
      quicksort(v,i,ultimo);
}
```

Veja que interessante: a função quicksort chama a si mesma!!!

Recursividade

- A função quicksort implementada é um exemplo de função recursiva: função que chama a si mesma.
- A recursividade é uma forma interessante de resolver problemas por meio da divisão dos problemas em problemas menores de mesma natureza.
- Se a natureza dos subproblemas é a mesma do problema, o mesmo método usado para reduzir o problema pode ser usado para reduzir os subproblemas e assim por diante.
- Quando devemos parar? Quando alcançarmos um caso trivial que conhecemos a solução.

Recursividade

- Assim, um processo recursivo para a solução de um problema consiste em duas partes:

1. O caso trivial, cuja solução é conhecida;
2. Um método geral que reduz o problema a um ou mais problemas menores (subproblemas) de mesma natureza.

- Muitas funções podem ser definidas recursivamente. Para isso, é preciso identificar as duas partes acima.

- Exemplo: fatorial de um número e o n-ésimo termo da seqüência de Fibonacci.

Função fatorial

- A função fatorial de um inteiro não negativo pode ser definida como:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

- Esta definição estabelece um processo recursivo para calcular o fatorial de um inteiro n.

- Caso trivial: n=0. Neste caso: n!=1.

- Método geral: n x (n-1)!.

Função fatorial

- Assim, usando-se este processo recursivo, o cálculo de 4!, por exemplo, é feito como a seguir:

$$
\begin{aligned}
4! &= 4 * 3! \\
&= 4 * (3 * 2!) \\
&= 4 * (3 * (2 * 1!)) \\
&= 4 * (3 * (2 * (1 * 0!))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

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

Mas como uma função recursiva como esta é de fato implementada no computador?

Usando-se o mecanismo conhecido como Pilha de Execução!

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(4)`

`fatorial(4)`

`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(3)`

`fatorial(3)` -> `return 3*fatorial(2)`
`fatorial(4)` -> `return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(2)`

`fatorial(2)`	`-> return 2*fatorial(1)`
`fatorial(3)`	`-> return 3*fatorial(2)`
`fatorial(4)`	`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha `fatorial(1)`

`fatorial(1)`	`-> return 1*fatorial(0)`
`fatorial(2)`	`-> return 2*fatorial(1)`
`fatorial(3)`	`-> return 3*fatorial(2)`
`fatorial(4)`	`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Empilha fatorial(0)

Pilha	
fatorial(0)	-> return 1 (caso trivial)
fatorial(1)	-> return 1*fatorial(0)
fatorial(2)	-> return 2*fatorial(1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha fatorial(0)

fatorial(0)	-> return 1 (caso trivial)
fatorial(1)	-> return 1***fatorial(0)**
fatorial(2)	-> return 2*fatorial(1)
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha fatorial(1)

fatorial(1)	-> return 1***1**
fatorial(2)	-> return 2***fatorial(1)**
fatorial(3)	-> return 3*fatorial(2)
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(2)`

fatorial(2)	-> return 2***(1*1)**
fatorial(3)	-> return 3***fatorial(2)**
fatorial(4)	-> return 4*fatorial(3)

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(3)`

fatorial(3)	`-> return 3*(2*1*1)`
fatorial(4)	`-> return 4*fatorial(3)`

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Pilha de Execução

Desempilha `fatorial(4)`

`fatorial(4)` `-> return 4*`**(3*2*1*1)**

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Resultado = 24

Menor elemento de uma lista

- A função menorElemento de uma lista de n inteiros pode ser definida como:

menorElemento(lista, **i**, n) = lista[**i**], se **i** == n
menor(lista[**i**], menorElemento(lista,**i**+1,n)), se **i** != n

E a função **menor(x,y)** retorna o menor número entre **x** e **y**

- Esta definição estabelece um processo recursivo para calcular o menor número de uma lista com n elementos a partir de uma posição inicial **i**.

- Caso trivial: n=**i**. Neste caso: o menor da lista é o seu elemento lista[**i**].

- Método geral: **menor**(lista[**i**], menorElemento(lista,**i**+1,n))

Menor elemento de uma lista

```c
void main() {
    int i;
    int lista[30];
    srand((unsigned)time(NULL));
    printf("lista: ");
    for(i=0; i<30; i++) {
        lista[i] = rand()%100;
        printf("(%d) ", lista[i]);
    }
    printf("\nMenor elemento: %d", menorElemento(lista, 0, 30));

}
```

Menor elemento de uma lista

```
int menorElemento(int lista[], int i, int n) {
    if(i == n-1)
        return lista[i];
    int valor1 = lista[i];
    int valor2 = menorElemento(lista, i+1, n);
    if(valor1 < valor2)
        return valor1;
    return valor2;
}
```

Função Fibonacci

- A função Fibonacci retorna o n-ésimo número da seqüência: 1, 1, 2, 3, 5, 8, 13, ...

- Os dois primeiros termos são iguais a 1 e cada um dos demais números é a soma dos dois números imediatamente anteriores.

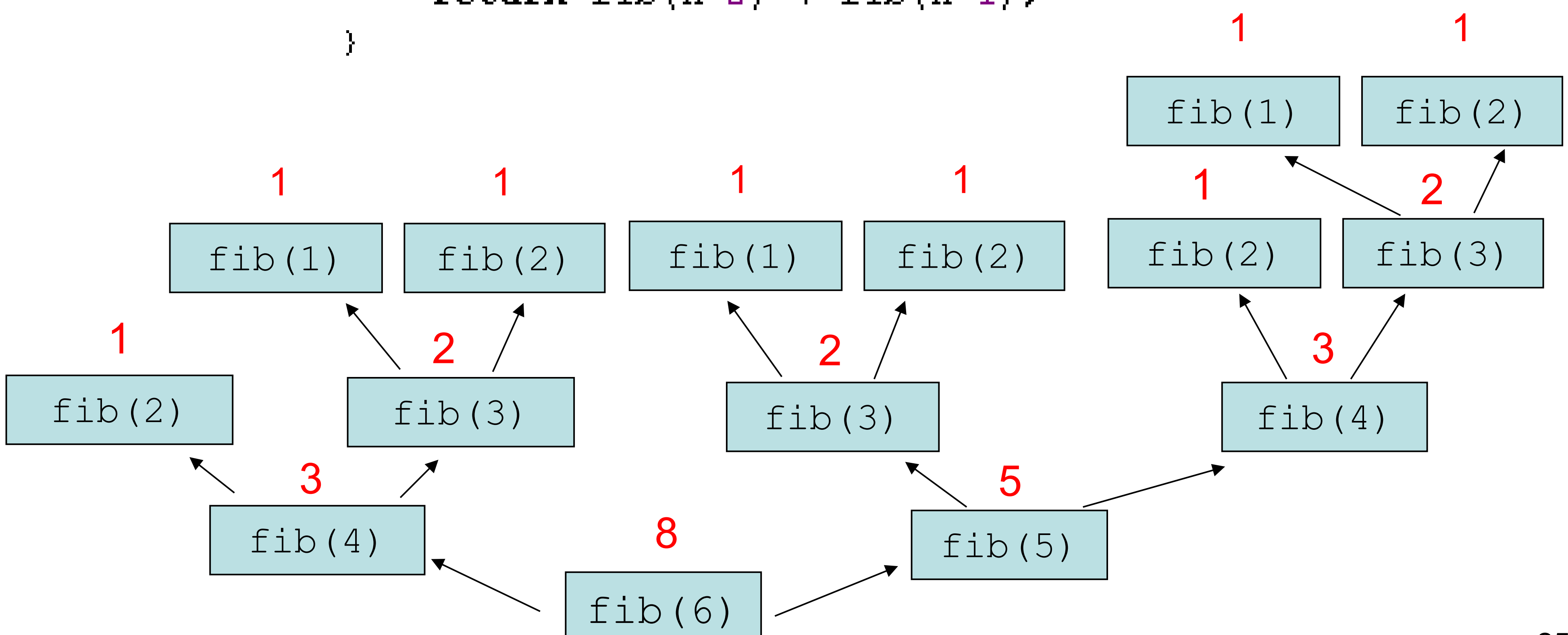
- Sendo assim, o n-ésimo número *fib(n)* é dado por:

$$fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-2) + fib(n-1) & n > 2 \end{cases}$$

Função Fibonacci

- Veja uma implementação recursiva para esta função:

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



Função Fibonacci

- A função recursiva para cálculo do n-ésimo termo da seqüência é extremamente ineficiente, uma vez que recalcula o mesmo valor várias vezes

Observe agora uma versão iterativa da função fib:

```
int fib(int n)
{
  int i,a,b,c;
  if ((n == 1) || (n == 2))
    return 1;
  else
  {
    a = 0;
    b = 1;
    for (i = 3; i <= n; i++)
    {
      c = a + b;
      a = b;
      b = c;
    }
    return c;
  }
}
```

Função Fibonacci

- O livro dos autores Brassard e Bradley (*Fundamentals of Algorithmics*, 1996, pág. 73) apresenta um quadro comparativo de tempos de execução das versões iterativa e recursiva:

n	10	20	30	50	100
recursivo	8 ms	1 s	2 min	21 dias	10^9 anos
iterativo	0.17 ms	0.33 ms	0.50 ms	0.75 ms	1,50 ms

- Portanto: um algoritmo recursivo nem sempre é o melhor caminho para se resolver um problema.
- No entanto, a recursividade muitas vezes torna o algoritmo mais simples.