C语言与程序设计

The C Programming Language

5.4 递 归

华中科技大学计算机学院

毛伏兵

本节讲授内容

递归（recursion）是一项非常重要的编程技巧，可以使程序变得简洁和清晰，是许多复杂算法的基础。本章介绍

- 递归、递归函数的概念；
- 递归的执行过程；
- 典型问题的递归函数设计；
- 分治法与快速排序；（了解）
- 回溯法；（不要求）
- 递归在动态规划等算法中的应用。（不要求）

```
#include<stdio.h>
// 输出 n ~ 1
void  prn_int(int n)
{   if (n>0)  {
        printf ("%d ",n);
        prn_int(n-1);
    }
 }
int main(void)
{  prn_int(5);
   return 0;
 }
```

递归函数：在定义中含有递归调用

递归调用：调用自己

5.1 递归概述

- 递归是一种函数在其定义中直接或间接调用自己的编程技巧。递归策略只需少量代码就可描述出解题过程所需要的多次重复计算，十分简单且易于理解。
- **递归调用**：函数直接调用自己或通过另一函数间接调用自己的函数调用方式

  f() { … f(); … }
- **递归函数**：在函数定义中含有递归调用的函数

【例5. 1】 用递归法计算阶乘n！

- 阶乘的计算是一个典型的递归问题。n！定义为：

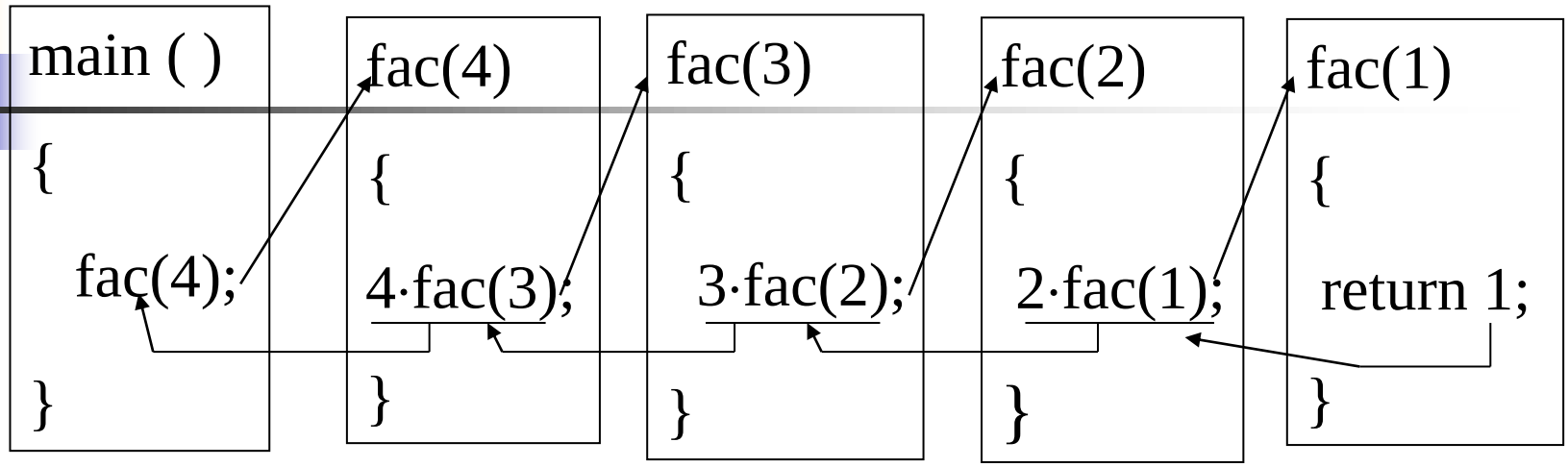
$$n!=\begin{cases}1 & n=0,1\\ n\times(n-1)! & n>1\end{cases}$$

- 这是递归定义式，对于特定的k，k!只与k和(k-1)！有关，上式的第一式是递归结束条件，对于任意给定的n！，将最终求解到1！或0！。

n！的递归实现

```
/*
函数功能：递归法求一个整数的阶乘。
函数参数：参数n为int,表示要求阶乘的数。
函数返回值：n的阶乘值，类型为long。        */
long factorial(int n)
{
 if (n==0||n==1)          /*  递归结束条件 */
        return 1;
 else
        return (n*factorial(n-1));  /* 递归调用 */
}
```

4！的递归执行过程



- 把求解问题转化为规模较小的子问题，通过多次递归一直到可以得出结果的最小解，然后通过最小解逐层向上返回调用，最终得到整个问题的解。
- 将递归概括为一句话：“能进则进，不进则退”
- 简化算法（易于理解)、但不节省存储空间，运行效率也不高(运行时开销大，效率低）

n！ 的迭代实现

```
/* 迭代法计算n！ */
long factorial_iteration( int n )
{
    int result = 1;
    while( n>1 ) {
        result *=n;
        n--;
    }
    return result;
}
```

对比

- **两种方式的比较**

两种实现方式都非常简单易懂，在实际项目中，为了效率，应该优先选择迭代。

- **什么情况下使用递归**

用递归能容易编写和维护代码，且运行效率并不至关重要

递归算法的特点

- 递归算法的运行效率较低，耗费的计算时间较长，占用的存储空间也较多。
- 递归算法结构紧凑、清晰、可读性强、代码简洁。
- 大多数的简单递归函数都能改写为等价的迭代形式。什么情况下使用递归呢？如果用递归能容易编写和维护代码，且运行效率并不至关重要，那么就使用递归。例如，像二叉树这样的数据结构，由于其本身固有的递归特性，特别适合于用递归处理；像回溯法等算法，一般也用递归来实现。

递归的要素

（1）每次调用在规模上有所缩小。

（2）递归结束条件

当子问题的规模足够小时，必须能够直接求出该规模问题的解。

例 编写一个递归函数计算Fibonacci数列的第n项。

```
/*  Recursive fibonacci : Compute the n item  */
long fibonacci (long n )
{
  if ( n = = 1 || n = = 2)  return 1;
  else
    return  fibonacci(n-1)+fibonacci(n-2);
 }
```

这种递归方式有损于运行效率——如何改进？

5.2 递归函数设计

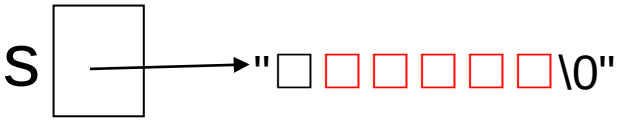
- 递归是一种强大的解决问题的技术，其基本思想是将复杂问题逐步转化为稍微简单一点的类似问题，最终将问题转化为可以直接得到结果的最简单问题。在较高级的程序中，递归是一个很重要的概念。

5.2.1 字符串的递归处理

- 字符串是以空字符（'\0'）结尾的字符序列。因此，可以把字符串看成“一个字符后面再跟一个字符串”，或者仅有一个空字符组成的空串。这个字符串的定义说明字符串是一种递归的数据结构，可以用递归的方法对一些基本的处理字符串函数进行编写。

【例1】 用递归实现标准库函数strlen(s)

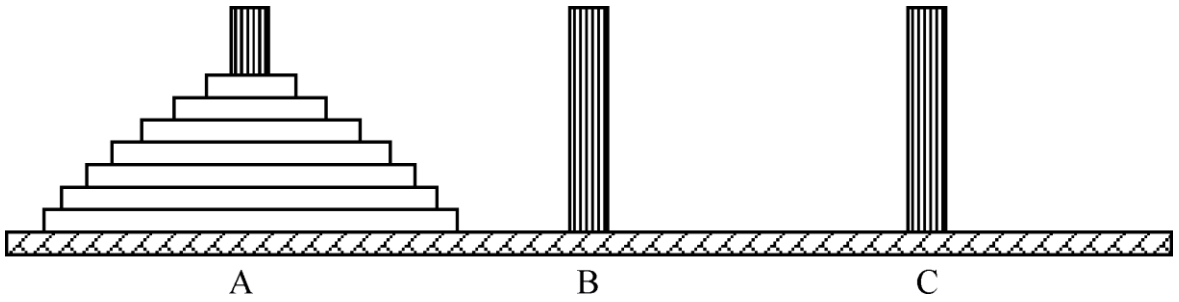
- 字符串看成“一个字符后面再跟一个字符串”




```
int strlen(char s[ ])
{
    if(s[0]=='\0')
        return 0;
    else
        return(1+strlen(s+1));
}
```

5.2.2 汉诺塔问题

- 问题：木桩A上有64个盘子，盘子大小不等，大的在下，小的在上。把木桩A上的64个盘子都移到木桩C上，条件是一次只允许移动一个盘子，且不允许大盘放在小盘的上面，在移动过程中可以借助木桩B。



【例5. 3】 设计一个求解汉诺塔问题的算法。

- 这是一个典型的用递归方法求解的问题。要移动n个盘子，可先考虑如何移动n-1个盘子。分解为以下3个步骤：

  （1）把A上的n-1个盘子借助C移到B。

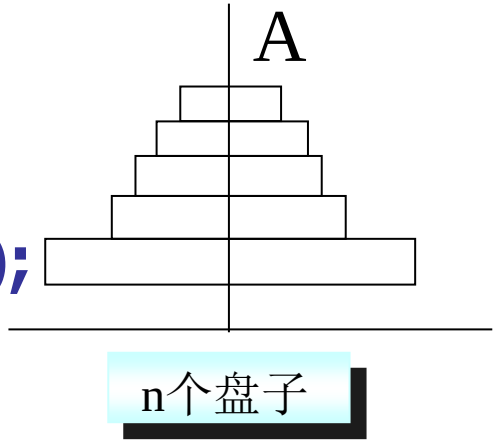
  （2）把A上剩下的盘子（即最大的那个）移到C。

  （3）把B上的n-1个盘子借助A移到C。

- 其中，第（1）步和第（3）步又可用同样的3步继续分解，依次分解下去，盘子数目n每次减少1，直至n为1结束。这显然是一个递归过程，递归结束条件是n为1。

函数move(n, a, b, c)

```
/* 将n个盘从a借助b，移至c */
void move(int n,int a,int b,int c )
 {
   if (n= =1) printf(" %c– –>%c\n ", a, c);
   else {
       move (n–1, a,c, b);
        printf(" %c– –>%c\n ", a, c);
        move (n–1, b, a, c);
    }
 }
```



移动次数：1个盘子：移动1次

2个盘子：移动2*1+1=3次

3个盘子：移动2*3+1=7次

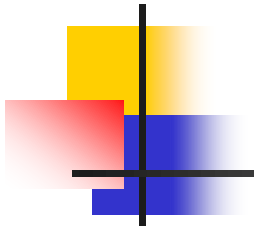
n个盘子：移动（2^n-1）次

64个盘子：移动（2^{64}-1）次

5.3 分治法与快速排序

- **分治法**：将一个大问题划分成若干互相独立的子问题，这些子问题与原问题相同但规模更小。
- **递归求解子问题**：分治与递归像一对孪生兄弟，经常同时应用在算法设计之中。

quick排序算法



- quick排序法是C.A.R.Hoare于1962年发明的。
- 基于分治策略

quick排序算法

（1）分解：将数组分为两部分

给定数组a[left]~a[right]，从中选择一个元素（称为分区元素），并把其余元素划分为两个子集合：

a[left]～a[split-1]、a[split]、a[split+1]~a[right]

左边部分的所有元素都比右边部分的元素小。

（2）递归求解

对两个子集合递归应用同一过程，当某个子集合中的元素个数小于2时，递归结束。

函数QuickSort

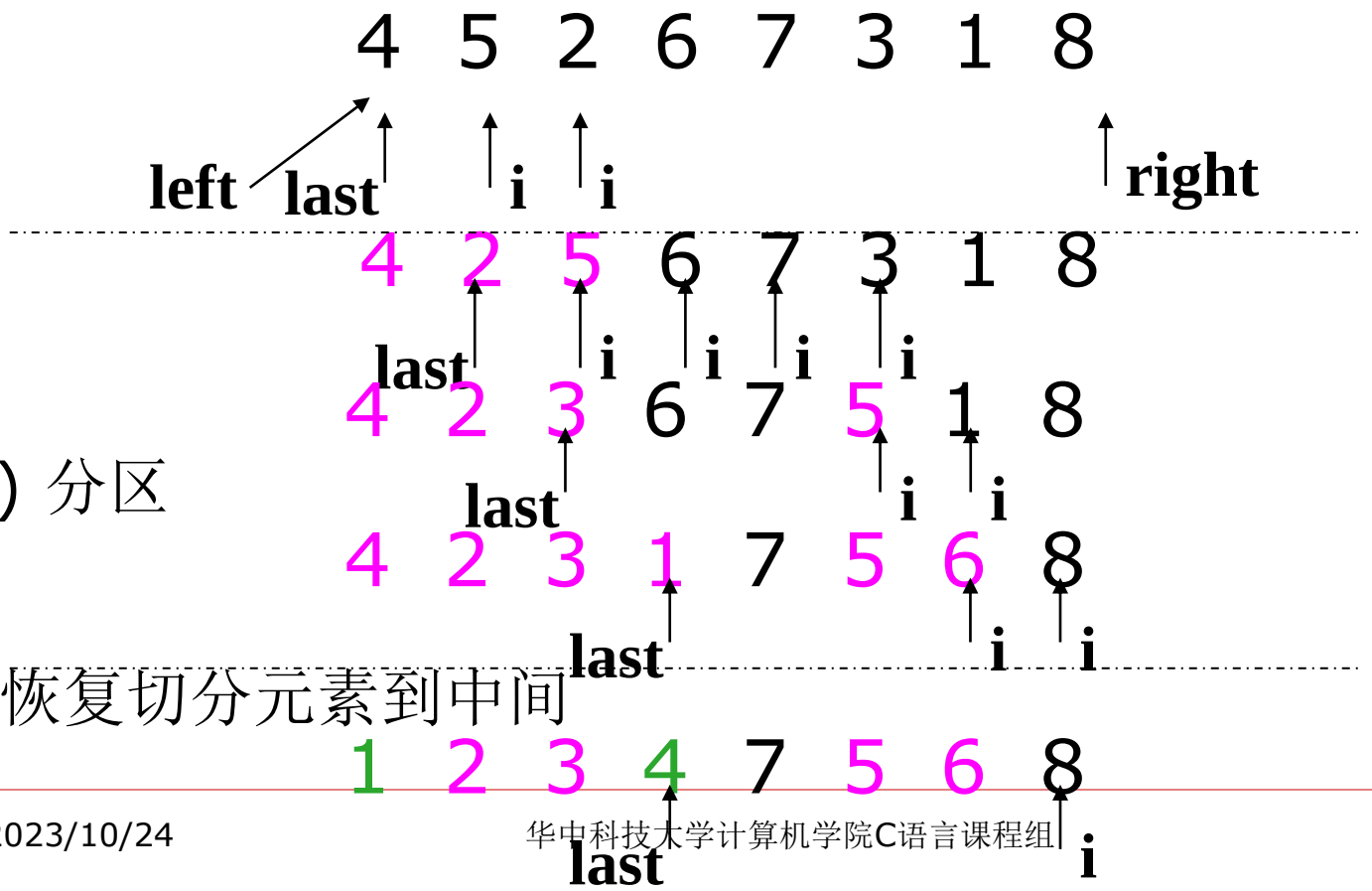
```
/* quick排序法 */
void QuickSort(int a[ ],int left,int right)
{
   int split;     /* 分区位置 */
   if(left<right) /* 待排序数组的元素个数至少为2 */
   {
        split=partition(a,left,right); /* 将数组元素分成两部分*/
        QuickSort(a,left,split-1);     /* 对左边部分递归排序 */
        QuickSort(a,split+1,right);    /* 对右边部分递归排序 */
   }
}
```

分区

- 选择中间元素作为切分元素

v[]={ 6 5 2 4 7 3 1 8}

(1) 移切分元素到最左边位置



(2) 分区

(3) 恢复切分元素到中间

```
/* 将数组a中的元素a[left ]至a[right]分成左右两部分，
   返回切分点的下标。  */
int partition_v1(int a[ ],int left,int right )
{  int i, last;     /*last为左边部分最后一个元素的位置*/
   int  split=(left+right)/2;  /*选择中间元素作为切分元素 */
   swap(a, left, split);  /*  移切分元素到最左边位置  */
   last = left;      /* 初始化last */
   for (i = left + 1; i <= right; i++)   /* 分区：从左至右扫描  */
       if (a[i] < a[left])   /* 小的数移到左边*/
           swap(a, ++last, i);
    swap(a, left, last);   /* 将切分元素移到两部分之间*/
    return last;
}
```

```
/* swap:  交换 v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;
     temp = v[i];
    v[i] = v[j];
    v[j] = temp;
  }
```

- stdlib.h 中有 函数 qsort 能对任意类型的对象排序。

分区

- 选择中间元素作为切分元素

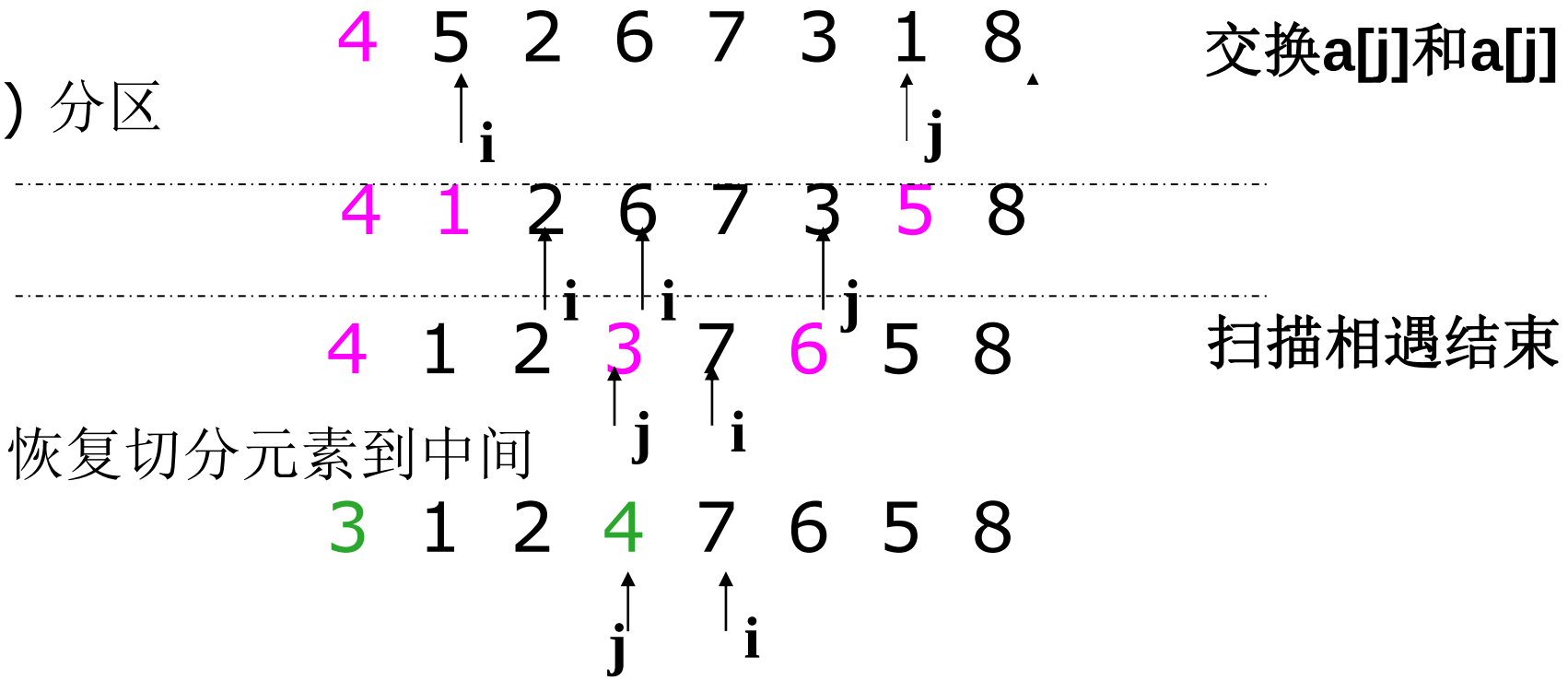
v[]={ 6 5 2 4 7 3 1 8}

i 从左至右扫描，找到比切分元素大的数停止

j从右至左扫描，找到比切分元素小的数停止

(1) 移切分元素到最左边位置

(2) 分区



(3) 恢复切分元素到中间

函数partition

```
int partition(int a[ ],int left,int right )
{
   int  i=left,j=right+1;
   int  split=(left+right)/2;  /*  选择中间元素作为切分元素  */
   swap(a,left,split);   /*  将切分元素移到数组的开头  */
   for ( ; ; )
   {
        while(a[++i]<=a[left] && i <= right);   /*从左至右扫描 */
        while(a[--j]> a[left]);          /*  从右至左扫描 */
        if(i>=j) break;     /*  扫描相遇（或交叉）结束循环  */
              swap(a,i,j);  /*  交换左右两边的元素  */
   } /*  j 是切分元素的位置  */
   swap(a,left,j);    /*  将切分元素重新移到中间  */
   return j;     /*  返回切分元素的下标  */
}
```

切分元素的选择

- 选择切分元素有很多种策略，最简单的方法是选用数组的第一个元素，该法对随机排列的数组很好，如果数据基本有序，则执行效率很差。上述程序中的方法可极大提高对有序或基本有序数组排序的效率。更加完善的策略是选择中间值，或至少是介于最大值和最小值之间的数值。
- 编写测试主函数用于输出排序结果，具体代码如下。
- \源程序\ex12_5.c