面向21世纪课程教材
普通高等教育“十一五”国家级规划教材
北京市高等教育精品教材
教育部普通高等教育精品教材

算法与数据结构

第一讲：绪论

张乃孝等编著

《算法与数据结构—C语言描述》

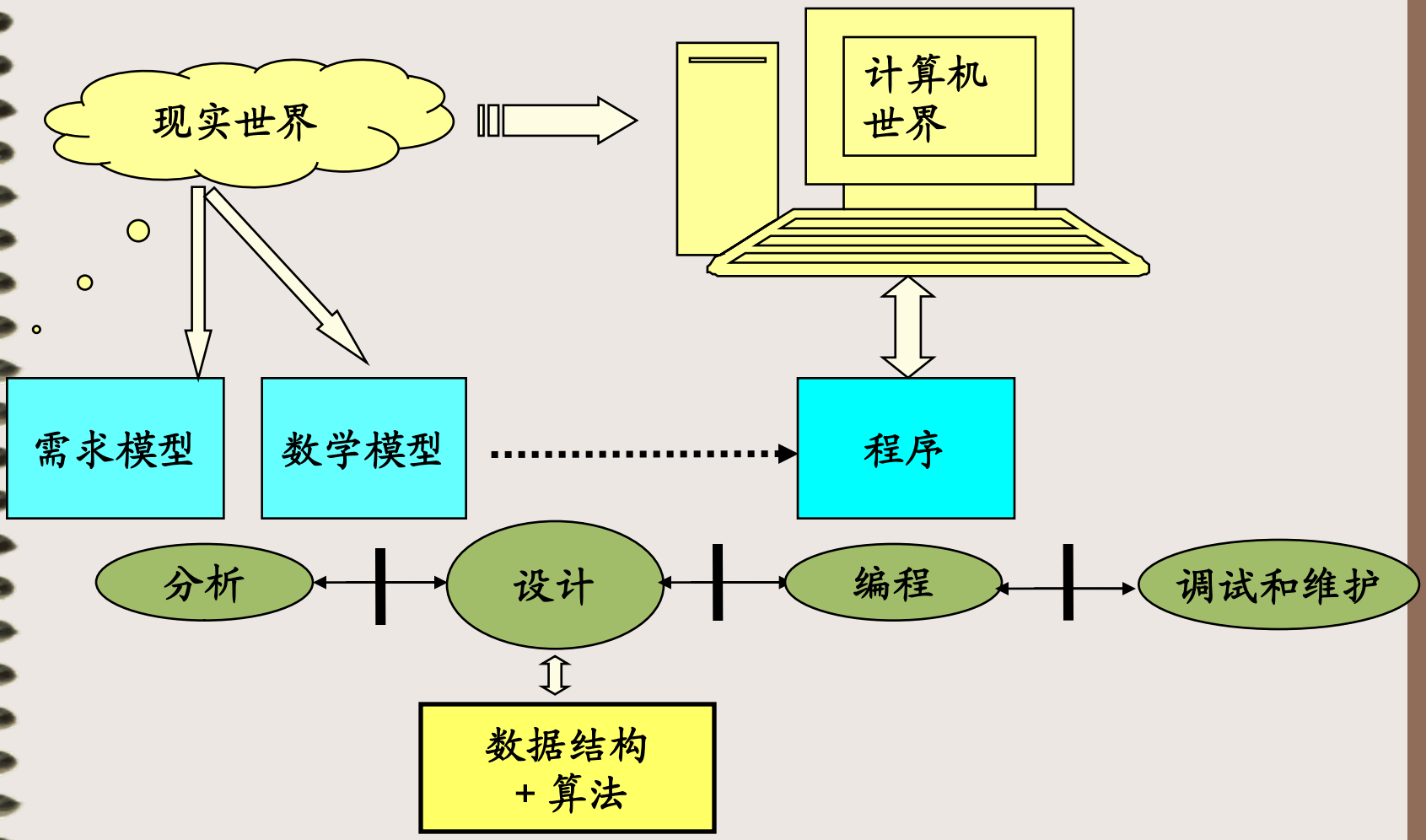
1 绪论

- 1.1 从问题到程序
- 1.2 抽象数据类型
- 1.3 数据结构
- 1.4 算法

1.1 从问题到程序

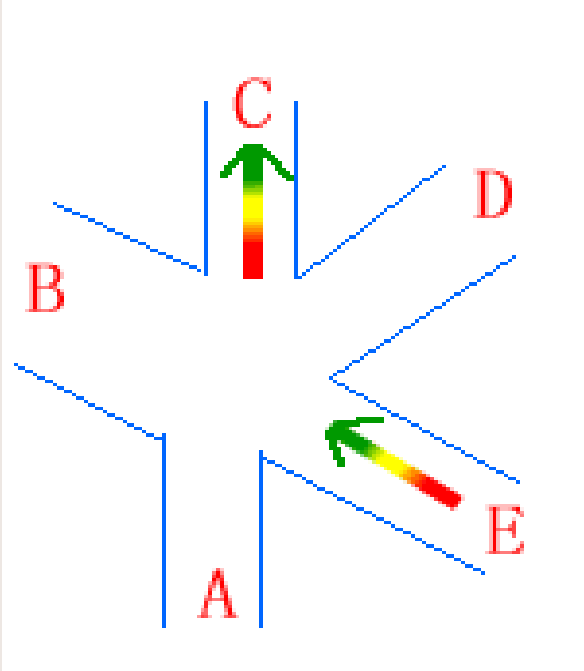
- 用计算机解决实际问题，就是要在计算机中建立一个解决这个问题的模型。
- 程序是使用程序设计语言精确描述的这样的模型。
- 程序中描述的**数据**用来表示问题中涉及的**对象**，程序中描述的**函数(过程)**表示了对于数据的处理**算法**；通过接受实际问题的输入，经过程序的运行，便可以得到实际问题的一个解。

用计算机求解问题



问题求解示例

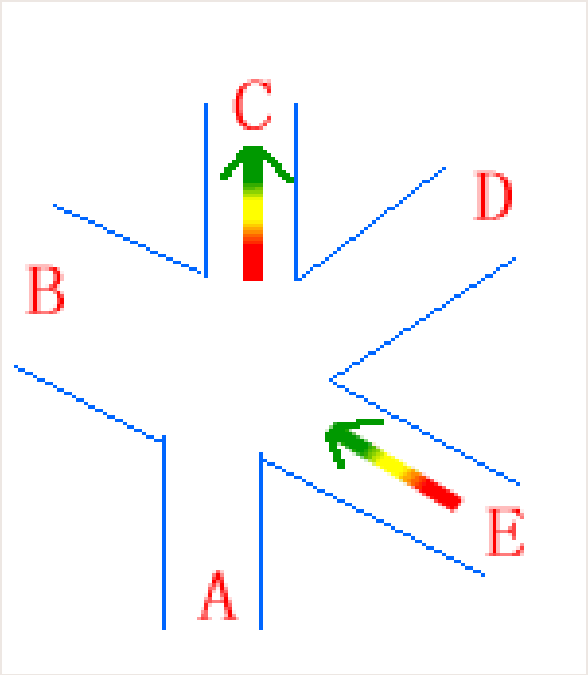
- 为一个多叉路口设计信号灯管理系统
- 对可能行驶路线实行分组:
  - 组内各方向行驶无冲突(可行)
  - 组数尽可能少(有效—最优解)



信号灯问题分析

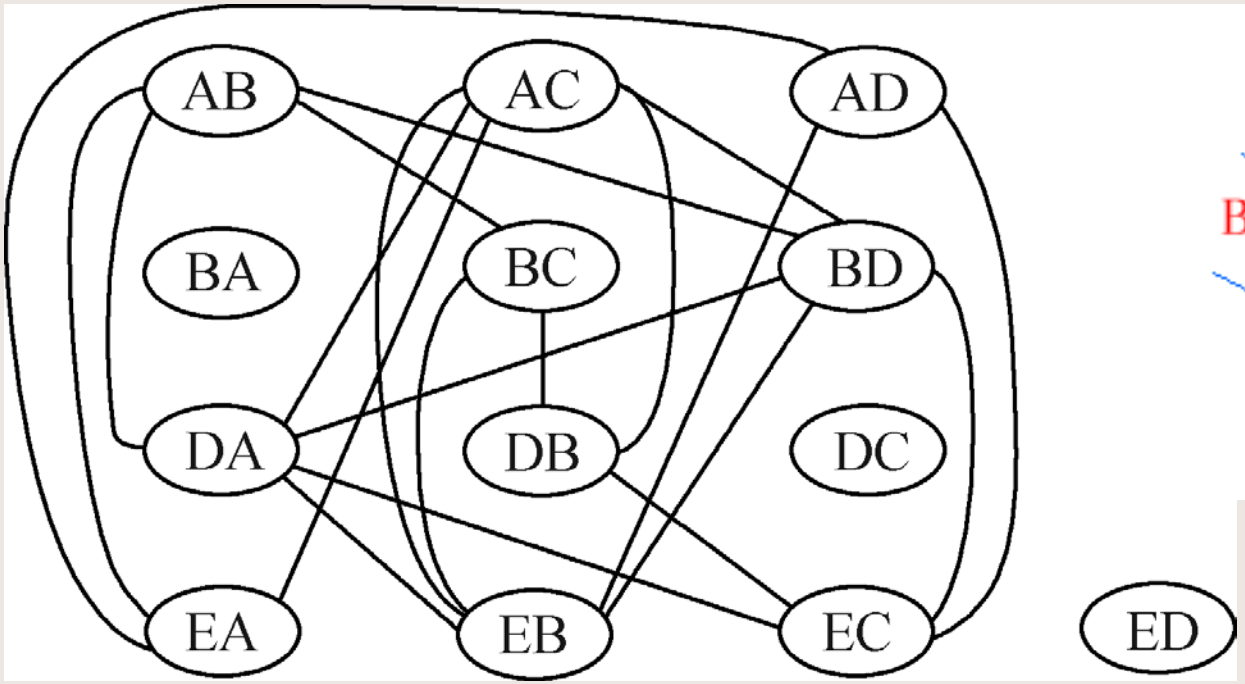
- 可以确定13个可能通行方向:

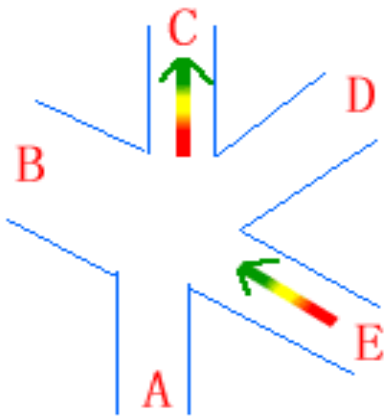
- A→B，A→C，A→D，
- B→A，B→C，B→D，
- D→A，D→B，D→C，
- E→A，E→B，E→C，E→D。



信号灯问题抽象

交叉路口不能同时行驶的用线连接：





着色问题

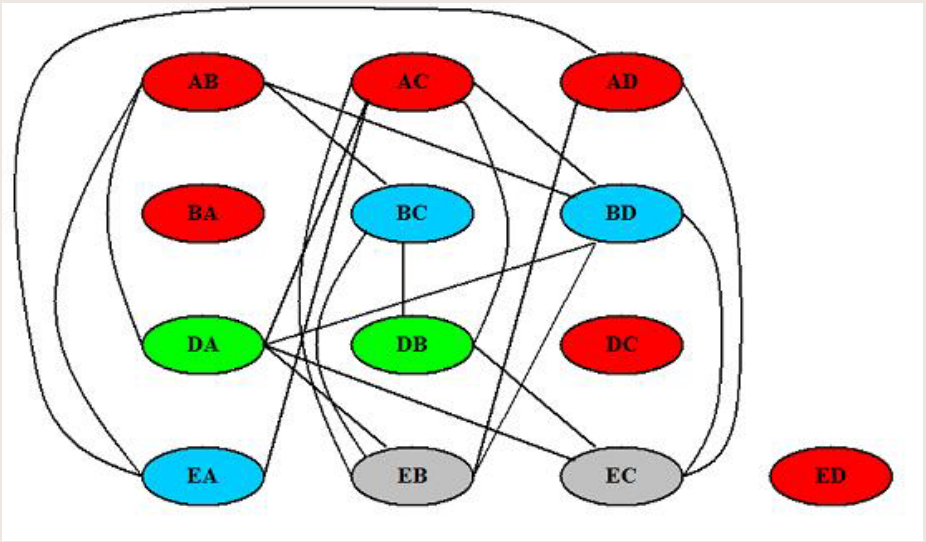
把上图中的一个结点理解为一个国家，结点之间的连线看作两国有共同边界，上述问题就变成著名的“着色问题”：即求出（最少）要几种颜色可将一个地图中所有国家着色，使得任意两个相邻的国家颜色都不相同。

求解的方法——穷举法

- 具体做法：从分为1、2、3…组开始考察，逐个列举出所有可能的着色方案，检查这样的分组方案是否满足要求。首先满足要求的分组，自然是问题的最优解。
- 这类穷举法对结点少的问题(称为“规模小的”问题)还可以用；对规模大的问题，由于求解时间会随着实际问题规模的增长而指数性上升，使计算机无法承受。

求解的方法——贪心法

- 先用一种颜色给尽可能多的结点上色；然后用另一种颜色在未着色结点中给尽可能多的结点上色；如此反复直到所有结点都着色为止。
- 红色：AB AC AD BA DC ED
- 蓝色：BC BD EA
- 绿色：DA DB
- 灰色：EB EC



数据结构的设计

- 使用国名表示国家;国名的集合表示国家的分组。
- 使用一个图表示地图。图中的结点名表示对应的国家；图中的边表示联系的两个国家有公共边界。

- 需要着色的图是G，G中所有结点的集合记为G.V;
- 集合V1存放图中所有未被着色的结点;
- 集合NEW存放可以用某颜色着色的所有结点。

集合和图需要有下面行为：

- 判断元素v是否属于集合V1: v ∈ V1；
- 从集合V1中去掉一个元素v: remove(V1,v)；
- 向集合NEW里增加一个元素v: add(NEW,v)；
- 判断集合V1是否空集合: isEmpty(V1) 。

- 检查结点v与结点集合NEW中各结点之间在图G中是否有边连接: notAdjacentWith (NEW, v, G) 。

解的核心部分

- 从V1中找出可用新颜色着色的结点集的工作可以用下面的伪码描述：

置NEW为空集合;
for 每个v ∈ V1 do
if v与NEW中所有结点间都没有边
从V1中去掉v；将v加入NEW；

- 这个伪码如果能执行，集合NEW中就得到一组可以用新颜色着色的结点。
- 着色程序可以反复调用这段伪码，直到V1为空，每次调用选择一种新颜色，这段伪码执行的次数就是需要的不同颜色个数。

解的抽象描述

```
int colorUp(Graph G) {
    int color = 0;   //记录使用的颜色数
    set V1 = G.V;  //V1初始化为图G的结点集V
    set NEW;
    while(!isEmpty(V1)) {
        NEW = { };
        while (v ∈V1. notAdjacentWith(NEW, v, G)) {
            add(NEW,v);  remove(V1,v);        }
        ++color;
    }
    return color;   //返回使用的颜色数
}
```

抽象数据类型的实现

- 如果集合和图是程序设计语言中预定义的类型，则colorUp中用到的remove(V1,v)和add(NEW,v)等就应该是语言中预定义的内部函数，该程序就几乎可以直接上机运行。否则程序员需要自己用语言所提供的（类型）机制实现这些抽象数据类型（集合、图等）。
- 大多数语言没有这类高级结构，只有基本数据类型、数组、结构和指针等。这时就需要自己实现集合、图及其相应操作。
- 如何有效实现这类高级结构就是数据结构研究的问题。

数据结构的选择和算法的精化

- 在数据结构确定以后，算法的描述可以进一步根据设计的数据结构进行精化。
- 如果在这个过程中又出现比较复杂的问题需要解决（例如：如何检查结点v与结点集合NEW中各结点之间在图G中是否有边连接问题），就可能还需要重复前面的思路构思求解方法，选择必要的抽象数据类型并且用适当的数据结构和算法实现它们。
- 经过这种反复的精化过程，最后将算法中所有部分都细化为能用程序设计语言描述的成分，得到的就是我们希望的程序。

1.2 抽象数据类型

- 类型
  - 一组值(或者对象)的集合
- 数据类型
  - 在计算机(语言)中可以使用的一个类型, 它不但包括这个类型的值的集合,还包括定义在这个类型上的一组操作

抽象数据类型

- 有一定行为（操作）的抽象（数学）类型。

- **抽象**出数据类型的使用要求，而把它的具体表示方式和运算的实现细节都**隐藏**起来。
- 支持数据类型的**实现与使用分离**的原则，是一种十分有效的对问题进行抽象与分解的思维工具。
- **允许独立地考虑数据类型的外部接口和内部的实现**。
- 对于系统的分解,设计,维护和修改十分有利，是**面向对象**技术与方法的主要理论基础。

抽象数据类型举例

ADT Circle **is**
  data
    real r**;** real x, y**;**
  operations
    real area ( )
    real circumference( )
    real getRadius ( )
    ... ...
end ADT Circle**;**

C语言里的ADT技术

- **C**没有为**ADT**提供专门支持（**C**语言设计时还没认识到**ADT**的重要性），但可通过程序技术模拟。
- 在**C**里实现一个**ADT**通常用两个文件，**.h** 文件定义数据表示（定义类型）和操作原型，**.c** 文件实现操作。
- 以**Circle**为例，**circle.h** 文件：

```
typedef struct { double x, y, r; } Circle;
Circle createCircle (double x, double y, double r);
double area (Circle c);
double circumference (Circle c);
... ... /* 其他操作的声明（原型） */
```

circle.c 文件：

```
#include "circle.h"
const double pi = 3.14159265;
circle createCircle (double x, double y, double r) {
  Circle c;
  c.x = x; c.y = y; c.r = r;
  return c;
}
double area (Circle c) { return c.r *c.r *pi; }
double circumference (Circle c) { return 2 * c.r * pi; }
... ... /* 其他操作的实现*/
```

- 使用**Circle**类型的文件应**#include “circle.h”**，做可执行程序时应把文件**circle.c** 作为其中一部分。

1.3 数据结构

- 传统的概念：数据结构是计算机中表示(存储)的、具有一定逻辑关系和行为特征的一组数据。
- 根据面向对象的观点：数据结构是抽象数据类型的物理实现。

主要解决两个问题：

如何具体表示抽象数据类型中的数学模型；

如何给出抽象数据类型中需要操作的实现。

数据结构的三个方面

- 逻辑结构（数学模型）
  - 指数学模型(集合，表，树和图等)中的基本元素（结点）之间的相互关系（在抽象数据类型中这些关系隐含在数学名称中）。
  - **描述方式：B=<K,R>, K是结点的有穷集合, R是K上的一个关系。**
- 存储结构（物理结构）
  - 指数学模型的具体表示方式，包括结点的表示和关系的表示。
  - **数据的逻辑结构在计算机存储器中的映射(或表示)。**
- 操作（行为）
  - 指抽象数据类型关心的各种行为在不同的存储结构上的具体实现(**算法或程序**)。

数据结构的分类（Ⅰ）

- 按逻辑结构分类
  - 给定 B=<K,R>, 若<k1,k2> ∈R, 则称k1为k2的**前驱**, k2为k1的**后继**。没有前驱的结点为**开始结点**, 没有后继的结点为**终端结点**。
  - 根据R的特点可以将数据结构分为以下三类:
    - 线性结构: K中每个结点最多只有一个前驱和一个后继;
    - 树形结构: K中每个结点最多只有一个前驱,但可有多个后继;
    - 复杂结构:K中结点的前驱、后继结点的个数都不作限制;

    特别地
    - 集合结构：当R为空集时，K中结点间没有约束关系。

数据结构的分类(II)

- 按存储结构分类
  - 计算机的内存提供了一个可以连续存储、随即存取和连续编址的存储空间。
  - 存储结构的设计目标，就是使用比较少的空间记录逻辑结构的必要信息，并且有效实现抽象数据类型中要求的操作。
  - 四种最基本的存储方法
    - 顺序表示
    - 链接表示
    - 散列表示
    - 索引表示

顺序表示与链接表示

- 顺序表示
  - 用一个连续的空间, 顺序存放数据结构中的各个结点。
  - 适合表示线性结构：结点之间的逻辑关系可以用存储空间的物理次序来反映。
- 链接表示
  - 结点的存放位置是任意的,结点之间的关系通过与结点关联的指针(或引用)方式显式表达出来。
  - 比较灵活, 适合表示经常进行插入, 删除等动态变化的数据结构。

散列表示与索引表示

- 散列表示:关键码---地址转换法
  - 选择适当的散列(杂凑)函数, 根据关键码的值将结点映射到给定的存储空间(散列表)中。
  - 检索的效率近似随机存取, 适合于要高速检索的结构。
- 索引表示
  - 给出一种从关键码到存储地址的映射方法,即建立辅助的索引结构。
  - 每个索引项包含一个结点的关键码和该结点的存储位置。

数据结构的分类(III)

- 按操作（行为）分类
  - 行为的规范由抽象数据类型决定

    队列——先进先出

    栈——后进先出
  - 行为的实现由算法决定但与存储表示关系密切

    集合元素的排序算法
    - 顺序表示
    - 链接表示

1.4 算法

- **算法**是由有穷规则构成（为解决某一类问题）的运算序列。
  - 算法可以有若干输入(初始值或条件)。
  - 算法通常又有若干个输出(计算结果)。
  - 算法应该具有有穷性。一个算法必须在执行了有穷步之后结束。
  - 算法应该具有确定性。算法的每一步，必须有确切的定义。
  - 算法应该具有可行性。算法中的每个动作，原则上都是能够由机器或人准确完成的。

算法的正确性

- 如果一个算法以一组满足初始条件的输入开始，那么该算法的执行一定终止，并且在终止时得到满足要求的（输出）结果。

算法设计的方法

- 贪心法
- 分治法
- 回溯法
- 动态规划法
- 分枝界限法

贪心法

- 当追求的目标是一个问题的最优解时，设法把对整个问题的求解工作分成若干步骤来完成。在其中的每一个阶段都选择从局部看是最优的方案，以期望通过各阶段的局部最优选择达到整体的最优。

- **示例解着色问题时就是采用的贪心法。**
- 贪心法实际上不能保证都成功地产生一个全局性最优解，但是通常可以得到一个可行的较优解。

分治法

- 把一个规模较大的问题分成两个或多个较小的与原问题相似的子问题。首先对子问题进行求解，然后设法把子问题的解合并起来，得出整个问题的解，即对问题分而治之。如果一个子问题的规模仍然比较大，不能很容易地求得解，就可以对这个子问题重复地应用分治策略。

- 二分法检索就是用分治策略的典型例子。

回溯法

- 有一些问题，需要通过彻底搜索所有可能情况寻找一个满足某些预定条件的最优解。由于彻底搜索的运算量非常大，所时可以采取一步一步向前试探，当前可行就继续向前，一旦发现问题或失败就后退，回到上一步重新选择一种，只要目前可行就继续向前，一意选择一种，只要目前可行就继续向前，一旦发现问题或失败就后退，回到上一步重新选择，借助于回溯技巧和中间增加判断，这样常常可以大大地减少搜索时间。

- 常见的**迷宫问题以及八皇后问题都可以用回溯方法来解决。**

动态规划法

- 与分治法相似都是把一个大问题分解为若干较小的子问题，通过求解子问题而得到原问题的解。
- 不同点是：
  - 分治法每次分解的子问题数目比较少，子问题之间界限清楚，处理的过程通常是自顶向下进行；
  - 动态规划法分解的子问题可能比较多，而且子问题相互包含，为了重用已经计算的结果，要把计算的中间结果全部保存起来，通常是自底向上进行。

在带权图中，**求所有结点之间最短路径的Floyd算法（见第9章）就属于动态规划法。**

分枝界限法

- 与回溯法相似，也是一种在表示问题解空间的树上进行系统搜索的方法。
- 所不同的是，回溯法使用了深度优先策略，而分枝界限法一般采用广度优先策略或者采用最大收益（或最小损耗）策略，并且利用最优解属性的的上下界来控制搜索的分枝。

- 最后一章，在讨论背包问题时，介绍了一个用分枝界限法设计的算法。

算法分析

- 算法不仅要正确，而且要有效。
- 算法分析就是度量算法性质的过程：
  - 分析一个算法主要是看这个算法的执行需要花费多少机器资源。
- 最常用的算法度量特性：
  - **空间代价(空间复杂性)**：被解决问题的规模(以某种单位计)为n时，某求解算法所需存储空间按某种单位为$S(n)$，则称该算法的空间代价为$S(n)$。
  - **时间代价(时间复杂性)**：当问题规模(以某种单位计)为n时，算法所耗时间按某种单位为$T(n)$，则称该算法的时间代价为$T(n)$。

算法分析的三个重要概念

- 问题规模
- 空间单位
- 时间单位

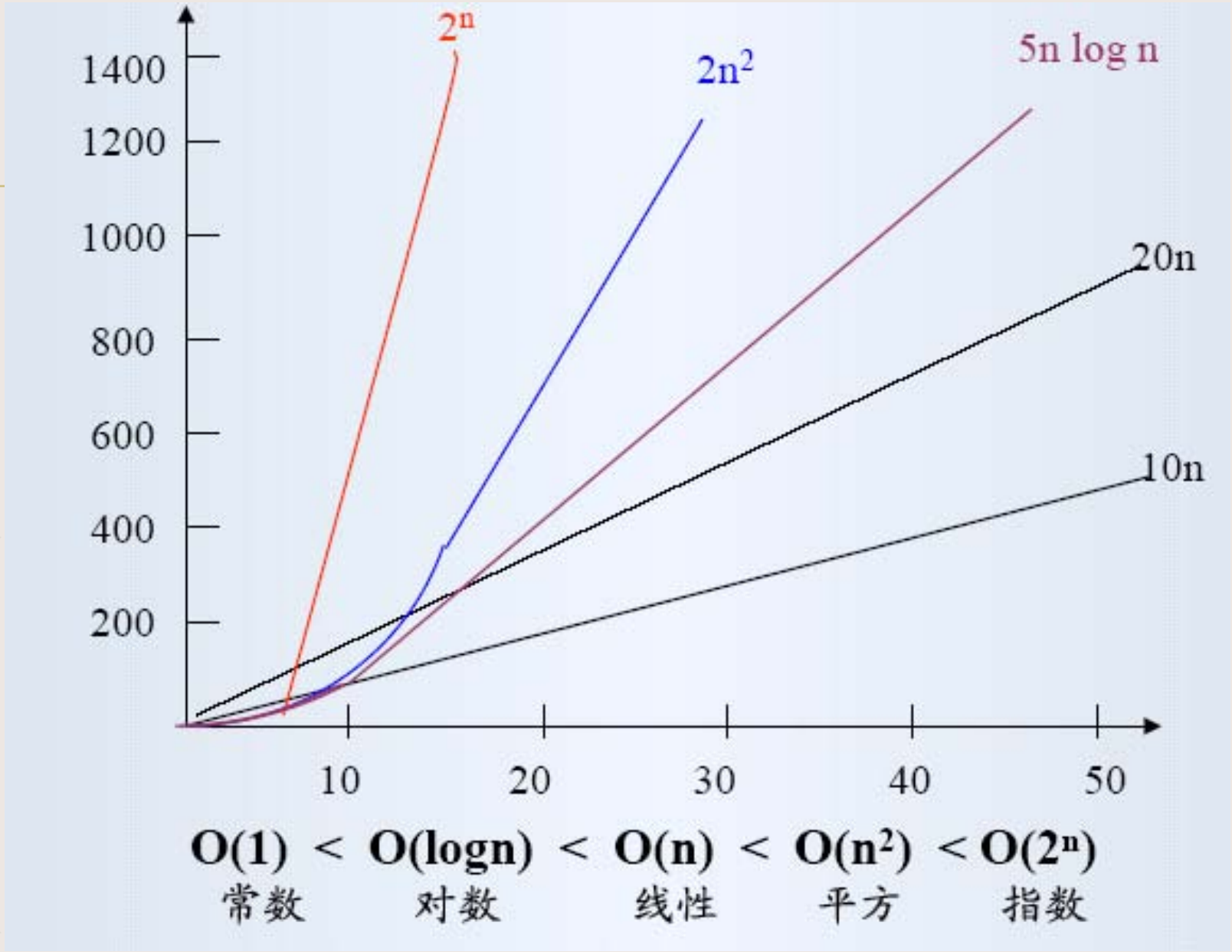
需要根据实际问题的情况确定

全面分析一个算法

- 求解某问题的一个具体算法, 对不同的问题实例, 也可能耗费不同的时间和空间, 全面分析一个算法需要考虑:
  - 最坏情况下时间(空间)复杂性;
  - 平均时间(空间)复杂性;
  - 最好情况下时间(空间)复杂性。

代价的表示和计算

- 大O表示法：
  - 更关注算法复杂性的量级；
  - 若存在正常数c和n_0，当问题的规模$n \geq n_0$后，某算法的时间(或空间)代价$T(n) \leq c \cdot f(n)$，则说该算法的时间(或空间)代价为$O(f(n))$。


1400
1200
1000
800
600
400
200
10
20
30
40
50
2n
2n2
5n log n
20n
10n
O(1) < O(logn) < O(n) < O(n2) < O(2n)
常数 对数 线性 平方 指数

算法时间复杂性的计算规则

- 加法规则(顺序复合)
  - 算法分为两部分时,复杂性是两部分复杂性之和

$$T(n)= T_1(n) + T_2(n)= O(f_1(n))+ O(f_2(n))$$
$$= O(\max(f_1(n), f_2(n)))$$

- 乘法规则(循环)
  - 循环$T_1(n)$ 次,每次$T_2(n)$时间,则

$$T(n)= T_1(n) \times T_2(n)= O(f_1(n)) \times O(f_2(n))$$
$$= O(f_1(n) \times f_2(n))$$

例：矩阵乘法

$C_{n\times n} = A_{n\times n} \times B_{n\times n}$

```
for(i=0;  i<n;   i++)
    for(j=0;  j<n;   j++) {
        c[i][j]=0;
        for(k=0;  k<n;   k++)
            c[i][j] = c[i][j]+a[i][k]*b[k][j];
    }
```

$$T(n) = O(f_1(n) \times f_2(n) \times (O(1)+O(n)))$$
$$= O(f_1(n)) \times O(f_2(n)) \times O(n)$$
$$= O(n) \times O(n) \times O(n) = O(n^3)$$

本讲重点

- 理解从问题到程序的主要过程；
- 体会抽象数据类型、数据结构和算法在问题求解过程中的作用；
- 了解数据结构的主要概念和分类；
- 了解算法的概念和主要设计、分析方法。

ADT，数据结构与算法相互关系

- 一个抽象数据类型决定了一组需要的操作；
- 不同抽象数据类型的实现，可以选择相同的逻辑结构；
- 对于一种逻辑结构,往往可以采用不同的存储结构；
- 一个算法的思想可以独立于具体的数据表示，但是操作的实现，总是依赖于具体的存储结构；
- 具体选择何种存储结构,主要应该考虑操作的要求：
  - 使得主要运算的开销,在时间和空间的权衡中达到最佳效果；
  - 算法的时间代价是选择与评价不同存储结构的关键。

算法+数据结构=程序

- 程序就是在数据的某些特定的结构和表示的基础上对于算法的描述。
- 算法与数据结构是程序设计中相辅相成、不可分割的两个方面。