《计算概论A》课程 程序设计部分

习题讲解 与 面向对象提出

李　戈

北京大学 信息科学技术学院 软件研究所

lige@sei.pku.edu.cn

北京大学

关于考试

- 时间： 1月14日下午 2:00-5:00
- 地点： 计算中心 4、5号机房
- 用户名/密码： 20130114
- 登陆域： student6
- 限制： 只能登陆Grids平台
- 座位安排：
  - 4、5 号机房的 座位号：1-68-136
  - 届时门口会张贴 座位号 - 学号 对应表

北京大学

答疑安排

- 时间
  - 1月4、7、9、11号的上课时间
    （虽然已经停课复习，仍按上课时间答疑）
  - 其他时间也可以，提前电话我就好
- 地点
  - 理科一号楼 1542办公室
- 电话
  - 62751794　　18611615085
- 紧急通知
  - Grids平台通知 + 课程网站通知

北京大学

1、习题讲解

2、知识点串讲

3、面向对象

题目 - 习题(15-10) 试剂稀释

显示 编辑 隐藏 删除

描述

一种药剂可以被稀释成不同的浓度供病人使用，且只能稀释不能增加浓度；又已知医院规定同一瓶药剂只能给某个病人以及排在他后面的若干人使用。现为了能最大限度利用每一瓶药剂(不考虑每一瓶容量)，在给出的一个病人用药浓度序列(病人的顺序不能改变)中找出能同时使用一瓶药剂的最多人数。

关于输入

有两行，第一行是一个整数n，为病人的人数，假设不超过100；第二行为一个浮点数（double）序列，为每个病人的用药浓度，浮点数之间用一个空格隔开。

关于输出

输出一行，该行包含一个整数，为所求的最大人数。

例子输入

```
6
0.7 0.9 0.6 0.8 0.8 0.4
```

例子输出

```
4
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n; // 病人的人数
    cin >> n;
    double medicine[100] ={0}; // 每个病人的用药浓度
    int i;
    int f[100] = {0};
    for (i = 0; i < n; i++)
    {
        cin >> medicine[i]; // 输入每个病人的用药浓度
        f[i] = 1; // 假设为1
    }
    int j;
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++) // 检查排在i前的每一个数
        {
            if (medicine[j] >= medicine[i] && f[j] + 1 > f[i])
            {
                f[i] = f[j] + 1;
            }
        }
        int max = 0;
        for (i = 0; i < n; i++) // 找出最大数
        {
            if (f[i] > max)
            {
                max = f[i];
            }
        }
        cout << max << endl; // 输出所求的最大人数
        return 0;
}
```

例子输入

```
6
0.7 0.9 0.6 0.8 0.8 0.4
```

```cpp
#include<iostream>
using namespace std;
int main()
{
    // n代表病人人数，i, j为循环变量，max, max1为统计最大值的变量
    // a[101]代表每个病人对应的试剂已被重复利用的最多次数
    int n, i, j, max, max1, a[101] = {0};
    double b[101] = {0};
    cin >> n;
    for (i = 1; i <= n; i++)
        cin >> b[i];
    for (i = 1; i <= n; i++)  // 判断每个病人用的试剂已被重复利用的最多次数
    {
        max1 = 0;
        for (j = i - 1; j >= 1; j--)
        {
            if (b[j] >= b[i])
            {
                a[i] = a[j] + 1;
                if (a[i] > max1)
                    max1 = a[i];
            }
        }
        a[i] = max1;
    }
    max = 0;
    for (i = 1; i <= n; i++)  // 找出被重复里用的最多次数并输出
        if (max < a[i])
            max = a[i];
    cout << max + 1 << endl;
    return 0;
}
```

最长上升子序列

- 问题描述
  - 一个数的序列 bi，当b1 < b2 < ... < bS 的时候，我们称这个序列是上升的。对于给定的一个序列(a1, a2, ..., aN)，我们可以得到一些上升的子序列(ai1, ai2, ..., aiK)，这里1 <= i1 < i2 < ... <iK <= N。
  - 比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7, 9), (3, 4, 8)等等。
  - 这些子序列中最长的长度是4，比如子序列(1, 3, 5, 8).
- 任务
  - 对于给定的序列，求出最长上升子序列的长度。

北京大学

最长上升子序列

- 问题分析
  - 需将“在全数列a_k（k=1, 2, 3…N）中求最长上升子序列 ”问题进行分解；
  - 子问题：
    - “求以a_k（k=1, 2, 3…N）为终点的最长上升子序列的长度”
    - 终点：一个上升子序列中最右边的那个数。
  - 搞一个数组 MaxLen，MaxLen [k] 存储数列中以a_k为终点的最长上升子序列的长度。

北京大学

最长上升子序列

- 设：
  - MaxLen [k] 存储数列中以a_k为终点的最长上升子序列的长度，则：

MaxLen[1] = 1

MaxLen[k]

= Max { MaxLen[i]： 1< i < k 且 a_i< a_k且 k≠1 } + 1

- 例如：　1　7　3　5　9　4　8

MaxLen[k]：1　2　2　3　4　3　4

最长上升子序列

```
#include <iostream>
using namespace std;
int b[1000], aMaxLen[1000];
int main()
{    int N;
     cin>>N;
     for( int i = 1;i <= N;i ++ )
          cin>>b[i];
     aMaxLen[1] = 1;
     for( i = 2; i <= N; i ++ ) {//每次求以第i 个数为终点的最长上升子序列的长度
          int nTmp = 0;       //记录满足条件的第i 个数左边的上升子序列的最大长度
          for( int j = 1; j < i; j ++ ) { //察看以第j 个数为终点的最长上升子序列
               if( b[i] > b[j] ) {
                    if( nTmp < aMaxLen[j] )
                         nTmp = aMaxLen[j];
               }}
          aMaxLen[i] = nTmp + 1;
     }
     int nMax = -1;
     for( i = 1;i <= N;i ++ )
          if( nMax < aMaxLen[i])
               nMax = aMaxLen[i];
          cout<<nMax;
     return 0;
}
```

再谈动归

- 动态规划
  - 1951年美国数学家R．Bellman等人，根据一类多阶段问题的特点，创立了解决这类过程优化问题的新方法——动态规划。
  - 把多阶段决策问题变换为一系列互相联系的单阶段问题，利用各阶段之间的关系，逐个求解；
  - 1957年出版了名著“Dynamic Programming”，成为该领域的第一本著作。

数字三角形

- 问题描述
  - 有一个数字三角形。从三角形的顶部到底部有很多条不同的路径，路径上的每一步只能从一个数走到下一层上和它最近的左边的数或者右边的数。对于每条路径，把路径上面的数加起来可以得到一个和，和最大的路径称为最佳路径。请找出最佳路径上的数字之和。

```
          7
        3   8
      8   1   0
    2   7   4   4
  4   5   2   6   5
```

数字三角形

- 递归的坏处
  - 每次计算MaxSum(r, j)的时候，都要计算一次MaxSum(r+1, j)；
  - 每次计算MaxSum(r, j+1)的时候，也要计算一次MaxSum(r+1, j)；
  - 重复计算因此产生！
  - 计算次数：

$2^0 + 2^1 + 2^2 \ldots\ldots 2^{N-1} = 2^N$

		1		
	1		1	
1		2		1
	1		3		1
1		4		6		4		1

```
#include <stdio.h>
#include <memory.h>
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int aMaxSum[MAX_NUM + 10][MAX_NUM + 10];
main()
{
    int i, j;
    cin>>N;
    for( i = 1; i <= N; i ++ )
        for( j = 1; j <= i; j ++ )
            cin>>D[i][j];
    for( j = 1; j <= N; j ++ )
            aMaxSum[N][j] = D[N][j];
    for( i = N ; i > 1 ; i -- )
            for( j = 1; j < i ; j ++ ) {
                if( aMaxSum[i][j] > aMaxSum[i][j+1] )
                    aMaxSum[i-1][j] = aMaxSum[i][j] + D[i-1][j];
                else
                    aMaxSum[i-1][j] = aMaxSum[i][j+1] + D[i-1][j];
            }
    cout<< aMaxSum[1][1];
}
```

再谈动归

- 动态规划
  - 是一种解决一类问题的途径；
  - 是对一类方法特点的总结；
  - 不是一个具体的方法或算法；
  - 不同问题，方法不同；
    - 线性动规
    - 区域动规
    - 树形动规
- 除非针对特定问题，无须特意考虑“动归”

聪明的寻宝人

- 问题
  - 一群寻宝人在沙漠中发现一处神秘的宝藏，宝藏中共有n个宝物（n不超过100），每个宝物的重量不同，价值也不同，宝物 i 的重量是 wi，其价值为 pi。每个寻宝人所能拿走的宝物的总重量为m。请帮助寻宝人写一个程序，计算每个人能够获得的最大总价值。
- 输入
  - 第一行有两个整数 n 和 m，分别为物品数和背包能够容纳的重量
  - 接下来 n 行，每行有两个数，分别为背包 i 的重量 wi 和价值 pi
- 输出
  - 输出每个寻宝人所能拿走的最大的总价值。

再谈动态规划

- 贪心与动态规划
  - 贪心法产生一个按贪心策略形成的判定序列；
  - 贪心法只能做到局部最优，不能保证全局最优，因为有些问题不符合最优性原理。
  - 动态规划会产生许多判定序列，再按最优性原理对这些序列加以筛选，去除那些非局部最优的子序列。
  - 动态规划能够获得最优解。

```
#include<iostream>
using namespace std;
int Number,Capacity,maxValue;//Number物品的个数，Capacity最大容量，最大价值
int Value[100],Weight[100],x[100];//物品的价值，物品的重量，x[i]物品的选中情况
void Compute(int i,int cv,int cw)
{                                    //cw当前包内物品重量，cv当前包内物品价值
    if(i==Number)//回溯结束
    {
        if(cv > maxValue)
            maxValue = cv;
    }
    else
        for(int j=0;j<=1;j++)
        {
            x[i]=j;
            if(cw+x[i]*Weight[i]<=Capacity)
            {
                cw+=Weight[i]*x[i];
                cv+=Value[i]*x[i];
                Compute(i+1,cv,cw);
                cw-=Weight[i]*x[i];
                cv-=Value[i]*x[i];
            }
        }
}
int main()
{
    cin>>Number>>Capacity;
    for(int i = 0; i < Number; i++)
        cin>>Weight[i]>>Value[i];
    Compute(0,0,0);
    cout<<"Max Value is: "<<maxValue<<endl;
    system("pause");
    return 0;
}
```

递归回溯解法

```cpp
#include<iostream>
using namespace std;
int Weight[100];
int Value[100];
int Compute(int i, int capacity) // {检查第i件物品} , capacity{剩余可用重量为j})
{    //计算出当有0,1,2...i件物品，且剩余最大可用重量为capacity时，最大可装载价值
     double r1, r2 = 0;
     if (i < 0)
     {
          return 0;
     }
     else if(capacity >= Weight[i])    //剩余可用重量可以放下物品i
     {
          r1 = Compute(i-1,capacity - Weight[i]) + Value[i]; //放入第i件物品所得总价值
          r2 = Compute(i-1,capacity); //不放第i件物品所得总价值
          return (r1>r2)?r1:r2;
     }
     else return 0;
}
int main()
{
     int Number, Capacity;
     cin>>Number>>Capacity;
     for(int i = 0; i < Number; i++)
          cin>>Weight[i]>>Value[i];
          cout<<Compute(Number, Capacity)<<endl;
     system("pause");
     return 0;
}
```

无回溯递归解法

```
#include<iostream>
using namespace std;
int Weight[100], Value[100];
int table[100][1000];                    //假设Capacity不超过1000
//行-物品编号，列-所假设的最大重量值，table值-相应状况下的最大能放价值
int compute(int capacity,int number)
{
     for(int i=1;i<number+1;i++)    //从1号物品开始到number号物品 分别开始“试放”
          for(int j=1;j<capacity+1;j++) //看容量多大时能够把物品放下
          {
               if(Weight[i]<=j){
                    if(Value[i]+table[i-1][j-Weight[i]] > table[i-1][j])
                         //如果放入i号物品 比 不放i号物品 的 最大重量要重
                         table[i][j] = Value[i]+table[i-1][j-Weight[i]];
                    else
                         table[i][j]=table[i-1][j];
               }else
                    table[i][j]=table[i-1][j];
          }
          return(table[number][capacity]);
}
int main()
{
     int Capacity,Number;
     cin>>Number>>Capacity;
     for(int i=1;i<Number+1;i++)
          cin>>Weight[i]>>Value[i];
     cout<<compute(Capacity,Number)<<endl;
     system("pause");
     return 0;
}
```

所谓动归解法

```
#include<iostream>
using namespace std;
int Weight[100], Value[100];
int table[100][1000];                    //假设Capacity不超过1000
//行-物品编号，列-所假设的最大重量值，table值-相应状况下的最大能放价值
int compute(int capacity,int number)
{
     for(int i=1;i<number+1;i++)    //从1号物品开始到number号物品 分别开始“试放”
          for(int j=1;j<capacity+1;j++) //看容量多大时能够把物品放下
          {
               if(Weight[i]<=j){
                    if(Value[i]+table[i-1][j-Weight[i]] > table[i-1][j])
                         //如果放入i号物品 比 不放i号物品 的 最大重量要重
                         table[i][j] = Value[i]+table[i-1][j-Weight[i]];
                    else
                         table[i][j]=table[i-1][j];
               }else
                    table[i][j]=table[i-1][j];
          }
          return(table[number][capacity]);
}
int main()
{
}
```

最大容量M	物品个数N												
				j=0-m									
10	3		C	0	1	2	3	4	5	6	7	8	9	10
物品大小w	物品价值p	编号	0	0	0	0	0	0	0	0	0	0	0	0
3	4	i= 1	1	0	0	0	4	4	4	4	4	4	4	4
4	5	1-n 2	2	0	0	0	4	5	5	5	9	9	9	9
5	6	3w	3	0	0	0	4	5	6	6	9	10	11	11

从结构化到面向对象

——从 C 到 C++

北京大学

程序设计语言的发展

现实世界的问题

现实世界中的解决方案

运行的程序

程序设计语言的发展

现实世界的问题

现实世界的解决方案

自然语言

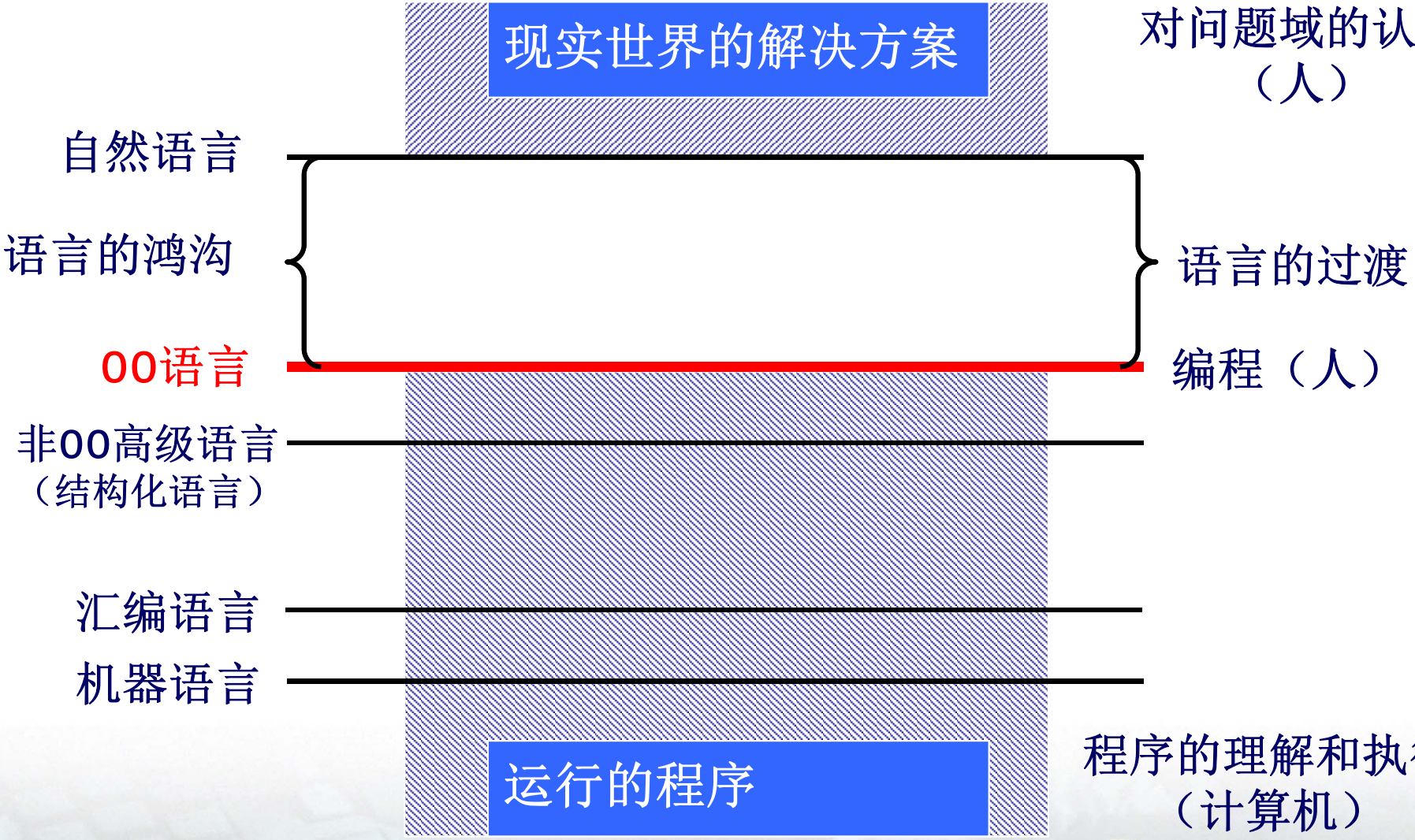
语言的鸿沟

编程语言

运行的程序

北京大学

程序设计语言的发展



北京大学

面向对象程序设计语言

- 基本原则
  - 万物皆为对象；
    - 对象表达为自身信息和可执行方法的组合；
  - 程序是对象的集合；
    - 对象直接可以通过发送消息调用对方；
  - 类似的对象可以抽象为类；
    - 具有相同类型的自身信息和可执行方法的对象可以抽象为类；
  - 类之间可以继承、聚合… …；

```
#include<iostream>
#include<string>
using namespace std;
class Person{
private:
    string name;
public:
    Person(string input_name){
        name = input_name;
    }
    void showname(){
        cout<<name<<endl;
    }
};
```

```
class Student: public Person{
private:
   int id;
public:
   Student(int input_id, string input_name):Person(input_name){
       id = input_id;
   }
   void showid(){
       cout<<id<<endl;
   }
};
int main()
{
   Student mike(20, "Mike");
   mike.showid();
   mike.showname();
   return 0;
}
```


昵图网 nipic.com/
感谢各位同学！

希望能在软件研究所再见到大家！

北京大学