

4.17 课堂笔记 7

本周作业：

作业要求：在代码最前面用注释表示出来：

- 1) 明确 $dp[i]$ 表示什么含义
- 2) 状态转移方程
- 3) 边界条件
- 4) 输出结果

1、课上最后一题路径求出来

在一个地图上有 n 个地窖 ($n \leq 200$)，每个地窖中埋有一定数量的土豆。同时，给出地窖之间的连接路径，并规定路径都是单向的，且保证都是小序号地窖指向大序号地窖，也不存在可以从一个地窖出发经过若干地窖后又回到原来地窖的路径。某人可以从任一处开始挖土豆，然后沿着指出的连接往下挖（仅能选择一条路径），当无连接时挖土豆工作结束。设计一个挖土豆的方案，使他能挖到最多的土豆。

如下图所示：圆圈内的 1 2 3 4 5 6，代表 6 个地窖的编号，地窖编号旁边的数字代表这个地窖土豆的数量！

输入

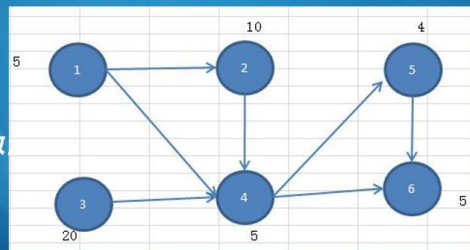
第一行：地窖的个数；
第二行为依次每个地窖土豆的个数
下面若干行：

xi yi

表示从 x_i 可到 y_i ， $x_i < y_i$ 。

输出

第一行输出挖土豆的地窖编号的顺序： $k_1-k_2-...-k_v$
第二行输出一个整数，代表最多能挖到的土豆的数量



样例输入

```
6
5 10 20 5 4 5
1 2
1 4
2 4
3 4
4 5
4 6
5 6
```

样例输出

```
3-4-5-6
34
```

2、P2008

3、P1233

动态规划

1、动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。

2、使用动态规划的前提：

- 能划分阶段
- 符合最优化原理
- 具备无后效性

3、使用动态规划求解四步骤：

- 1) 明确 $dp[i]$ 表示什么含义
- 2) 状态转移方程
- 3) 边界条件
- 4) 输出结果

4、线性动态规划

线性动态规划：问题模型是线性的，数据结构表现为线性表的形式

练习：

1、

题目描述 prefixMax.cpp

求一个数列的所有前缀最大值之和。

即：给出长度为 n 的数列 $a[i]$, 求出对于所有 $1 \leq i \leq n$, $\max(a[1], a[2], \dots, a[i])$ 的和。

对于每个位置的前缀最大值解释如下：对于第1个数666，只有一个数，一定最大；对于第2个数，求出前两个数的最大数，还是666；对于第3个数，求出前3个数的最大数是692.....其余位置依次类推，最后求前缀最大值得和。

$n \leq 100000$

样例输入

5

666,304,692,188,596

样例输出

3408

```
1  #include <iostream>
2  using namespace std;
3  int a[100], dl[100], n;
4  long long sum;
5  /*
6   状态表示: dp[i]: 以i为结尾的前缀最大值
7   状态转移: dl[i] = max(a[i], dl[i - 1]);
8   边界: dl[1] = a[1];
9   结果: dp[1]到dp[n]的和
10  */
11 int main(){
12     cin >> n;
13     for(int i = 1; i <= n; i++)
14         cin >> a[i];
15     dl[1] = a[1];
16     for(int i = 2; i <= n; i++){
17         if( a[i] > dl[i - 1])
18             dl[i] = a[i];
19         else
20             dl[i] = dl[i - 1];
21         // dl[i] = max(a[i], dl[i - 1]);
22     }
23     for(int i = 1; i <= n; i++)
24         sum += dl[i];
25     cout << sum;
26     return 0;
27 }
```

2、

lis.cpp最长不下降子序列

有一个长为 n 的数列 a_0, a_1, \dots, a_{n-1} 。请求出这个序列中最长不下降子序列的长度。不下降序列指的是对于任意的 $i < j$ 都满足 $a_i \leq a_j$ 的子序列，该问题被称为最长不下降子序列（LIS, Longest Increasing Subsequence）

举个栗子：给你一个序列为（1, 5, 2, 6, 9, 10, 3, 15, 1），那么它的最长不下降子序列为：（1, 2, 6, 9, 10, 15）

样例输入：

9

1 5 2 6 9 10 3 15 1

样例输出：6

- 1) `_dp` 数组存储当前项作为子序列结尾的最大不下降长度
- 2) 状态转移方程： $dp[i] = \max(dp[j] + 1) \quad j < i \text{ \& \& } dp[j] \leq dp[i]$
- 3) 问题的边界： $i=1$ 时： $dp[1] = 1$
- 4) 输出结果： $\max(dp[i])$

```
1  #include<iostream>
2  using namespace std;
3  int a[10000], dp[10000];
4  int n, _max;
5  int main(){
6      cin >> n;
7      for(int i = 1; i <= n; i++)
8          cin >> a[i];
9      for(int i = 1; i <= n; i++){ //状态转移过程
10         dp[i] = 1; //边界
11         for(int j = 1; j <= i - 1; j++){
12             if(a[j] <= a[i]){
13                 if( dp[j] + 1 > dp[i])
14                     dp[i] = dp[j] + 1;
15                 else dp[i] = dp[i];
16             }
17         }
18     }
19     _max = dp[1];
20     for(int i = 1; i <= n; i++){
21         _max = max(dp[i], _max);
22     }
23     cout << _max;
24 }
```

3、P1115

题目描述

[展开](#)

给出一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

输入格式

第一行是一个整数，表示序列的长度 n 。

第二行有 n 个整数，第 i 个整数表示序列的第 i 个数字 a_i 。

输出格式

输出一行一个整数表示答案。

输入输出样例

输入 #1

复制

输出 #1

复制

```
7
2 -4 3 -1 2 -4 3
```

```
4
```

- 1) $dp[i]$ 的含义：第 i 项为结尾的连续子段的最大和
- 2) 状态转移方程：阶段 $i-1$ 的子问题 \rightarrow 阶段 i 的子问题

$$dp[i] = \max(a[i] + dp[i - 1], a[i]);$$

- 3) 边界： $dp[1] = a[1]$
- 4) 输出结果： $\max(dp[i])$

```
1  #include<iostream>
2  using namespace std;
3  int a[200010], dp[200010];
4  int n, _max;
5  int main(){
6      cin >> n;
7      for(int i = 1; i <= n; i++)
8          cin >> a[i];
9      dp[1] = a[1]; // 确定边界
10     for(int i = 2; i <= n; i++){ // 状态转移过程
11         if(dp[i - 1] < 0)
12             dp[i] = a[i];
13         else
14             dp[i] = a[i] + dp[i - 1];
15         // 方法二：将上述if else替换为 dp[i] = max(a[i] + dp[i - 1], a[i]);
16     }
17     _max = dp[1];
18     for(int i = 1; i <= n; i++){
19         _max = max(dp[i], _max);
20         // cout << dp[i] << " "; // 输出状态转移方程
21     }
22     cout << _max;
23 }
```


4、

取数

设有 N 个正整数 ($1 \leq N \leq 50$)，其中每一个均是大于等于1、小于等于300的数。

从这 N 个数中任取出若干个数 (不能取相邻的数)，要求得到一种取法，使得到的和为最大。

例如：当 $N=5$ 时，有5个数分别为：13, 18, 28, 45, 21

此时，有许多种取法，如：13, 28, 21 和为62;

13, 45 和为58; 18, 45 和为63

和为63应该是满足要求的一种取法

输入

一行，有 N 个符合条件的整数。

输出

一个整数，即最大和

样例输入

13 18 28 45 21

样例输出

63

```
1  #include <iostream>
2  using namespace std;
3  int a[100], dp[100], n, sum;
4  /*
5   状态表示: dp[i]: 以i为结尾的最大和
6   状态转移: dp[i] = max(dp[i - 2] + a[i], dp[i - 1]);
7   边界:   dp[1] = a[1]; dp[2] = max(a[1], a[2]);
8   结果:   dp[n]
9   */
10 int main(){
11     cin >> n;
12     for(int i = 1; i <= n; i++)
13         cin >> a[i];
14     dp[1] = a[1];
15     dp[2] = max(a[1], a[2]);
16     for(int i = 2; i <= n; i++){
17         if( dp[i - 1] > dp[i - 2] + a[i])
18             dp[i] = dp[i - 1];
19         else
20             dp[i] = dp[i - 2] + a[i];
21     }
22     cout << dp[n];
23     return 0;
24 }
```

5、

最长公共子序列

longest common sequence

对任意两个字符串，求其最长公共子序列的长度

例如：给定序列 $s_1=\{1,3,4,5,6,7,7,8\}$, $s_2=\{3,5,7,4,8,6,7,8,2\}$,
 s_1 和 s_2 的相同子序列，且该子序列的长度最长，即是LCS。
 s_1 和 s_2 的其中一个最长公共子序列是 $\{3,4,6,7,8\}$

例2： $a=\{A,D,A,B,E,C\}$ $b=\{D,B,D,C,A\}$,最长公共子序列为 $\{D,B,C\}$, 长度为3

- 1) 状态变量： $f[i][j]$ 表示 $s_1[1...i]$ 与 $s_2[1...j]$ 的最长公共子序列长度
- 2) 状态转移：

$$f[i][j]=f[i-1][j-1] + 1; \quad \text{当 } s_1[i] == s_2[j]$$

$$f[i][j]=\max(f[i-1][j], f[i][j-1]); \quad \text{当 } s_1[i] != s_2[j]$$
- 3) 边界： $i=0$ 或 $j=0$, $f[i][j]=0$
- 4) 输出结果： $f[m][n]$

```

14  for(int i = 1; i <= _s1.size(); i++) {
15      for(int j = 1; j <= _s2.size(); j++) {
16          if(_s1[i - 1] == _s2[j - 1]) { //字符串，下标-1
17              _dp[i][j] = _dp[i - 1][j - 1] + 1;
18          }
19          else {
20              _dp[i][j] = max(_dp[i - 1][j], _dp[i][j - 1]);
21          }
22      }
23  }
24
25  cout << _dp[_s1.size()][_s2.size()];

```

求最长公共子序列本身：

		j	0	1	2	3	4	5
i				D	B	D	C	A
			0	0	0	0	0	0
0			0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↖ 1
2	D	0	↖ 1	← 1	↖ 1	← 1	↑ 1	
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	
4	B	0	↑ 1	↖ 2	← 2	← 2	↑ 2	
5	E	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	
6	C	0	↑ 1	↑ 2	↑ 2	↖ 3	← 3	

标记数组 p[i][j]:

```
for(i=1;i<=m;i++){ //m行
    for(j=1;j<=n;j++){ //n列
        if(a[i-1]==b[j-1])//注意下标
        {
            f[i][j]=f[i-1][j-1]+1;
            p[i][j]=1; //左上方↖
        }
        else if(f[i][j-1]>f[i-1][j])
        {
            f[i][j]=f[i][j-1];
            p[i][j]=2; //左边←
        }
        else
        {
            f[i][j]=f[i-1][j];
            p[i][j]=3; //上边↑
        }
    }
}
```

逆推求最长公共子序列本身:

```
int i,j,k; char s[200];
i=m; j=n; k=f[m][n];
while(i>0&&j>0){
    if(p[i][j]==1)//左上方↖
    {
        s[k--]=a[i-1];
        i--; j--;
    }
    else if(p[i][j]==2)//左边←
        j--;
    else//上边↑
        i--;
}
```

最后输出 s 数组即可。

6、

在一个地图上有 n 个地窖 ($n \leq 200$) ,每个地窖中埋有一定数量的土豆。同时,给出地窖之间的连接路径, 并规定路径都是单向的,且保证都是小序号地窖指向大序号地窖, 也不存在可以从一个地窖出发经过若干地窖后又回到原来地窖的路径。某人可以从任一处开始挖土豆, 然后沿着指出的连接往下挖 (仅能选择一条路径), 当无连接时挖土豆工作结束。设计一个挖土豆的方案, 使他能挖到最多的土豆。

如下图所示: 圆圈内的1 2 3 4 5 6, 代表6个地窖的编号, 地窖编号旁边的数字代表这个地窖土豆的数量!

输入

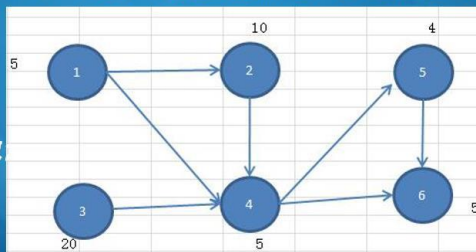
第一行: 地窖的个数;
第二行为依次每个地窖土豆的个数

下面若干行:

xi yi
表示从xi可到yi, $xi < yi$ 。

输出

第一行输出挖土豆的地窖编号的顺序: k1-k2-...-kv
第二行输出一个整数, 代表最多能挖到的土豆的数量



样例输入

```
6
5 10 20 5 4 5
1 2
1 4
2 4
3 4
4 5
4 6
5 6
样例输出
3-4-5-6
34
```

解题思路:

(1) 邻接矩阵 $mapp[i][j]$: 记录了由 i 到 j 是否有路径

i \ j	1	2	3	4	5	6
1		True		True		
2				True		
3				True		
4					True	True
5						True
6						

(2) $a[i]$ 记录每个地窖中的土豆数量, 如下表所示

a 数组	5	10	20	5	4	5
下标	1	2	3	4	5	6

(3) 分析动态四要素, 如下:

- ① 状态表示: $dp[i]$: 以 i 为起点到终点的最大和
- ② 状态转移: $dp[i] = \max(dp[i], dp[j] + a[i])$ 其中 $j > i$ & $mapp[i][j] == \text{true}$;
- ③ 边界: $dp[n] = a[n]$
- ④ 结果: $\max(dp[1-n])$


```

21 int main(){
22     int maxn = 0;
23     cin >> n;
24     int curI, curJ;
25     for(int i = 1; i <= n; i++)
26         cin >> a[i];
27     while( cin >> curI >> curJ ){
28         mapp[curI][curJ] = true;
29     }
30
31     dp[n] = a[n]; // 边界
32     for(int i = n - 1; i >= 1; i--){ // 倒序计算dp数组
33         for(int j = i + 1; j <= n; j++){
34             if(mapp[i][j]){ // 如果i可以到达j的话, 寻找最大的dp
35                 dp[i] = max(dp[i], dp[j] + a[i]); // 状态转移过程
36             }
37         }
38     }
39
40     // 下面是找dp最大的值maxn
41     for(int i = 1; i <= n; i++){
42         maxn = max(maxn, dp[i]);
43     }
44     cout << maxn;
45     return 0;
46 }

```