# 目录

1.[钢管切割（最小值）](#_1.钢管切割)

2.[两条流水线调度](#_2.两条流水线调度)

3.[多条流水线调度](#_3.多条流水线调度)

4.[最长上升子序列](#_4.最长上升子序列)

5.[矩阵链乘](#_5.矩阵链乘)

6.[OBST](#_6.OBST)

7.[免费馅饼](#_免费馅饼)

# 内容

## 1.钢管切割

### 实现解释：

先设数组price[i]存储着i长度钢管切割后的最小值，p[i]存储着i长度钢管不切割的值，price数组既是本问题的dp数组。

经过分析可知状态转移方程为：

price[0] = 0;

price[i] = min(p[1]+price[i-1],p[2]+price[i-2],...p[i-1]+price[1],p[i]);

因为price[i]已经是当前情况下的最小值了，所以只需要遵循转移方程进行代码的完善即可。

若要计算最大值只需在计算前设置最大值为负数（保证最小），然后在判断递归判断小于即可。

### 坑点：

初始化和状态转移方程的书写

### 完整代码：

1. *//钢管切割最小值*
2. #include<iostream>
3. using namespace std;
4. int main()
5. {
6. ios::sync\_with\_stdio(false);
7. int length,MIN;*//分别为长度和最小值*
8. while(cin >> length)
9. {
10. int p[length+1];
11. for(int i = 1;i<=length;i++)
12. cin >> p[i];*//依据题意，i长度的价值*
14. int price[length+1];*//保存每段价值*
15. price[0] = 0;*//会使用到price[0]，防止出错初始化*
17. for(int i = 1;i<=length;i++)
18. {
19. MIN = 2147483647;*//获得最小值时需要设为最大值*
20. for(int j = 1;j<=i;j++)
21. {
22. if(MIN > p[j]+price[i-j])
23. *//不断将i长度切割为前j部分和后i-j部分*
24. *//以找到i长度切割后的最小值*
25. {
26. MIN = p[j] + price[i-j];*//替换最小值*
27. price[i] = MIN;
28. }
29. *//状态转移方程介绍见实现解释*
30. }
31. }
32. cout << price[length] << '\n';
33. *//输出最长部分（i）切割后的最小值*
34. }
35. return 0;
36. }

## 2.两条流水线调度

### 实现解释：

实现方法即得出状态转移方程后完善即可，设a[][i]存储着第一二条线上各家的时间花费，t[][i]存储着i处进行线路切换的花费，f[][i]存储着各线在i处的最小花费。

则对每一个f[][i]应有如下的转移方程：

f[0][1] = a[0][1];

f[1][1] = a[1][1];

f[0][i] = min(f[0][i-1]+a[0][i],f[1][i-1]+t[1][i-1]+a[0][i]);

f[1][i] = min(f[1][i-1]+a[1][i],f[0][i-1]+t[0][i-1]+a[1][i]);

前两个为初始定义，后两个为具体的转换。

min函数中前一个值表示直接在当前这条线的前一个结点前进的花费，后一个表示从另一条线的前一个结点先转移过来后再前进的花费。此时只有两条线因此只有这两种情况，所以比较后便可得出到达该线该位置的最小花费。

多条线的情况可参考：题解：说好的ALS呢？（后续添加链接）

最后比较到达n处（最后一个家）哪条线的时间花费最小即可。

### 坑点：

注意多次输入后数组的初始化，以及状态转移方程的正确转化即可

### 完整代码：

1. *//流水线调度基本问题*
2. #include<iostream>
3. using namespace std;
4. int main()
5. {
6. ios::sync\_with\_stdio(false);
7. int n;
8. int fend;
9. int i;
10. while(cin >> n)
11. {
12. int a[2][n+1],t[2][n];
13. *//两条线上每一家花费的时间*
14. for(i = 1;i<=n;i++)
15. cin >> a[0][i];
16. for(i = 1;i<=n;i++)
17. cin >> a[1][i];
19. *//两条线不同位置转移的时间花费*
20. for(i = 1;i<n;i++)
21. cin >> t[0][i];
22. for(i = 1;i<n;i++)
23. cin >> t[1][i];
25. *//存储两条线不同位置的最小时间*
26. int f[2][n+1];
27. *//初始化防止错误*
28. f[0][1] = a[0][1];
29. f[1][1] = a[1][1];
31. for(i = 2;i<=n;i++)
32. {
33. *//具体状态转移方程介绍见实现解释*
34. if(f[0][i-1]+a[0][i]<f[1][i-1]+t[1][i-1]+a[0][i])
35. f[0][i] = f[0][i-1]+a[0][i];
36. else
37. f[0][i] = f[1][i-1]+t[1][i-1]+a[0][i];
38. if(f[1][i-1]+a[1][i]<f[0][i-1]+t[0][i-1]+a[1][i])
39. f[1][i] = f[1][i-1]+a[1][i];
40. else
41. f[1][i] = f[0][i-1]+t[0][i-1]+a[1][i];
42. }
43. *//计算两条线分别的最后时间花费得最小值*
44. if(f[0][n]<f[1][n])
45. fend = f[0][n];
46. else
47. fend = f[1][n];
48. cout << fend << '\n';
49. }
50. return 0;
51. }

## 3.多条流水线调度

### 实现解释：

相比两条线的其实只是需要多判断几次

设a[i][j]存储着第i条线上第j个位置的花费，t[i][j]存储着第i条线到第j条线的调度费用，f[i][j]存储着i条线在j处的最小花费。

则对每一个f[][i]应有如下的转移方程：

第一个站台：f[i][1] = a[i][1];

其余站台：f[i][j] = min(f[i][j-1]+t[i][i]+a[i][j],f[1][j-1]+t[1][i]+a[i][j],...,f[n][j-1]+t[n][i]+a[i][j]);

注意站台间转移时需要考虑自己调度到自己的情况（视题意而定）

min函数中前一个值表示直接在当前这条线的前一个结点前进的花费，后面则表示从其他线的前一个结点先转移过来后再前进的花费，一次比较后便可得出到达当前线该位置的最小花费。

最后比较到达n处（最后一个站台）哪条线的时间花费最小即可。

### 坑点：

注意按照题意判断是否要考虑自身的调度t[i][i]即可

### 完整代码：

1. #include<iostream>
2. #include<cstring>
3. #include<algorithm>
4. using namespace std;
5. int main()
6. {
7. ios::sync\_with\_stdio(false);
8. int num,n;
9. int fend;
10. int i,j;
11. int tempf;
12. while(cin >> num >> n)*//分别为线路个数和站台个数*
13. {
14. int a[num][n+1],t[num][num];
15. for(i = 0;i<num;i++)
16. for(j = 1;j<=n;j++)
17. cin >> a[i][j];*//i条线j站台的花费*
19. for(i = 0;i<num;i++)
20. for(j = 0;j<num;j++)
21. cin >> t[i][j];*//i条线到j条线的调度费用*
22. int f[num][n+1];*//一维为当前线路，二维为站台位置*
23. for(i = 0;i<num;i++)
24. {
25. f[i][1] = a[i][1];*//每条线第一个站台的最小花费就是第一个*
26. }
27. for(i = 2;i<=n;i++)
28. {
29. for(j = 0;j<num;j++)
30. {
31. tempf = 2147483647;*//为了找到最小值先设定最大的*
32. for(int k = 0;k<num;k++)
33. {*//循环判断所有条道路的情况*
34. if(tempf > f[k][i-1]+t[k][j]+a[j][i])
35. {
36. tempf = f[k][i-1]+t[k][j]+a[j][i];
37. }
38. }
39. f[j][i] = tempf;*//存储j条线在i站台的最小值*
40. }
41. }
42. tempf = f[0][n];
43. for(i = 1;i<num;i++)
44. {
45. if(tempf > f[i][n])
46. {
47. tempf = f[i][n];
48. }
49. }
50. fend = tempf;*//得到最大值*
51. cout << fend << '\n';
52. }
53. return 0;
54. }

## 4.最长上升子序列

### 实现解释：

这里介绍的其实是优化后的方案，即只存储长度，而不是以dp[i][j]的形式存储ij之间的最长长度，不过也是很好理解的，所以就直接分享这一篇吧。

a[i]存储总序列的内容，dp[i]表示以i为结尾的最长子序列长度

那么首先由dp[i]开始一定是1（自己是一个序列）

后面的状态转移方程即：

dp[i] = max(dp[j])+1（j<i&&a[j]<a[i]）

解释：由于是上升序列，而且dp[i]是以i结尾的最长长度，因此长度增加时有两个条件：新的数字在旧数字的后面，新的数字大于旧的数字（新数字：dp[i]，旧数字：dp[j]），也是唯一的转移方程。

### 坑点：

注意最后获取最长序列时，不能直接dp[n]输出，因为可能是在序列中间有最长的上升子序列，也需要循环判断。

### 完整代码：

1. #include<iostream>
2. using namespace std;
3. int main()
4. {
5. int n;
6. cin >> n;
7. int a[n],dp[n];
8. for(int i = 0;i<n;i++)
9. {
10. cin >> a[i];
11. dp[i] = 1;*//自身一定是一个长度的序列*
12. }
13. for(int i = 1;i<n;i++)
14. {
15. for(int j = 0;j<i;j++)
16. {
17. if(a[i] > a[j])*//因为是上升，所以需要只有比前面的值大才可能形成最长上升子序列*
18. {
19. if(dp[i] < dp[j]+1) dp[i] = dp[j] + 1;*//记录i处最长的上升序列长度*
20. *//即前面的序列长度最大长度+1即是i处的最大长度*
21. }
22. }
23. }
24. int max = dp[0];*//不一定最后一个是最长的，因此需要获取最大值*
25. for(int i = 1;i<n;i++)
26. {
27. if(max < dp[i]) max = dp[i];
28. }
29. cout << max << '\n';
30. return 0;
31. }

## 5.矩阵链乘

### 实现解释：

数据介绍：m[0]存储第一个矩阵的行数，m[i]存储第i个矩阵的列数和第i+1个矩阵的行数

num[i][j]记录第i个矩阵和第j个矩阵之间的最小计算次数

cut仅是记录切割位置所用cut[i][j]表示第ij个矩阵之间的最优切割位置

则按照矩阵乘法的知识，两个矩阵相乘的计算次数便可直接由以下的状态转移方程得到：

i=j时num[i][j] = 0

i<j时num[i][j] = min(num[i][k]+num[k+1][j]+m[i-1]\*m[k]\*m[j])（k为[i,j)间的值，即表示以第i个矩阵到j-1个矩阵之间第k个矩阵作为分割处）

解释：i=j自然不用计算，第二个既是当k作为分割点时总次数等于i到k个矩阵的相乘最小次数+第k+1个矩阵到第j个矩阵的相乘最小次数+合并时的产生的新次数（最左行数\*分割点列数\*最右行数）

然后便是正常的翻译了。

其中对于分割方案的输出，只需按照矩阵范围依据cut数组获取切割点输出即可，具体可参考代码。

### 坑点：

循环范围的设定，注意按是否能到达设定范围

### 完整代码：

1. #include<iostream>
2. using namespace std;
3. int length;
4. int cut[2000][2000];
5. int count;
6. void printCut(int i,int j)
7. {
8. if(i == j)
9. cout << 'A' << i;
10. else
11. {
12. cout << '(';
13. printCut(i,cut[i][j]);
14. printCut(cut[i][j]+1,j);
15. count++;
16. cout << ')';
17. }
18. }
19. int main()
20. {
21. ios::sync\_with\_stdio(false);
22. int n;
23. cin >> n;
24. length = n;
25. int m[n+1];
26. for(int i = 0;i<=n;i++)
27. cin >> m[i];
29. int num[n+1][n+1];*//对应矩阵个数即可*
30. for(int i = 1;i<=n;i++)
31. num[i][i] = 0;
33. int temp;
34. count = 0;
35. for(int l = 2;l<=n;l++)
36. *//l为1的情况不必讨论，因为就是矩阵自己，所以从2到n依次进行*
37. {
38. for(int i = 1;i<=n-l+1;i++)
39. *//从第一个矩阵一直到最后一个可到达的矩阵*
40. *//这样i表示的既是连续l个矩阵的第一个矩阵*
41. *//根据输入，第i个矩阵的行数：i-1的值，列数：i的值*
42. {
43. int j = i+l-1;*//当前l个矩阵的最后一个矩阵下标*
44. num[i][j] = 2147483647;*//用于比较*
45. for(int k = i;k<=j-1;k++)
46. *//k是节点，代表第k和k+1个矩阵之间的那个下标*
47. *//所以需要到达j-1，也就是第j个矩阵前的那个位置才算结束*
48. *//由位置可知k的值为第k个矩阵的列数即划分开时需要乘的值*
49. {
50. temp = num[i][k]+num[k+1][j]+m[i-1]\*m[k]\*m[j];
51. *//当前划分状态的计算次数，两个num分别为两侧矩阵链的次数，最后一项为合并后的新增次数*
52. if(num[i][j]>temp)
53. {
54. num[i][j] = temp;
55. cut[i][j] = k;
56. }
57. }
58. }
59. }
60. printCut(1,n);
61. cout << '\n';
62. cout << count << '\n';*//切割次数*
63. cout << num[1][n] << '\n';*//计算次数*
64. return 0;
65. }

## 6.OBST

### 实现解释：

实际等价于矩阵链乘，状态转移方程都是将左侧和右侧的最优相加后再加上合并新增的次数即可。

新增的次数对矩阵链乘来说就是两个最优部分相乘的次数

对OBST来说则是左右子树本来的搜索代价+左右子树深度加一产生的新代价+此时根节点的搜索代价（其中深度加一的新代价和根节点的搜索代价便可合并为此时所有节点的检索频率之和）

解释：深度加一时左右子树节点的搜索代价都增加了一个结点，因此他们的代价分别增加了一次搜索概率（自己被搜索到的概率），而根节点的搜索代价即本身的搜索概率，相加既是范围内所有结点的搜索概率和。

于是数据记录如下：

e[i][j]存储i到j个结点之间的最小搜索代价

w[i][j]存储i到j个结点的总搜索概率之和（用于状态转移快捷增加搜索概率和）

root[i][j]存储i到j个结点中有最小搜索代价时的根节点的脚标

状态转移方程即：

j = i-1时e[i][j] = 0（不存在根节点，因此也没有搜索代价）

j >= i时e[i][j] = min(e[i][r-1]+e[r+1][j]+w[i][j])（r即某次选择作为根节点的脚标）

对建树方案的实现和矩阵链乘的实现同理，只是由于伪关键字的加入需要进行一些特殊处理，即如果某个方向没有子树则需要手动添加伪关键字d形成的结点，其余子树的输出只需做好根节点和左右的判断即可，具体可参考代码，方案不唯一。

### 坑点：

注意dp数组的初始化，初始化既是对没检索到情况的初始化（因此添加的值只是伪关键字的搜索概率）

主要难点即理解搜索子树扩增时平均搜索的变化

### 完整代码：

1. #include<iostream>
2. using namespace std;
3. const int MAX = 2147483647;
4. double \*\*root;
5. int di;
6. int n;
7. void printOBST(int l,int r)
8. {
9. if(l == 1&&r == n)
10. {
11. di = 0;
12. cout << "k" << root[1][n] << "为根" << '\n';
13. }
14. int t,lt,rt;
15. t = root[l][r];
16. if(l == t)
17. cout << "d"<<di++ << "是k"<<t << "的左孩子" << '\n';
18. else
19. if(l < t)
20. {
21. lt = root[l][t-1];
22. cout << "k"<<lt << "是k"<<t << "的左孩子" << '\n';
23. printOBST(l,t-1);
24. }
25. else return ;
27. if(r == t)
28. cout << "d"<<di++ << "是k"<<t << "的右孩子" << '\n';
29. else
30. if(r > t)
31. {
32. rt = root[t+1][r];
33. cout << "k"<<rt << "是k"<<t << "的右孩子" << '\n';
34. printOBST(t+1,r);
35. }
36. else return ;
37. }
38. int main()
39. {
40. int maxi;
41. double temp;
42. while(cin >> n)
43. {
44. double p[n+1],q[n+1];
45. for(int i = 1;i<=n;i++)
46. cin >> p[i];
47. for(int i = 0;i<=n;i++)
48. cin >> q[i];
49. double e[n+2][n+1];*//需要存储q的内容，因此n+2*
50. double w[n+2][n+1];*//同上*
51. root = new double \*[n+1];*//只是在p中选root因此n+1即可*
52. for(int i = 0;i<=n;i++)
53. root[i] = new double[n+1];
55. for(int i = 1;i<=n+1;i++)
56. {
57. e[i][i-1] = q[i-1];
58. w[i][i-1] = q[i-1];
59. }
60. for(int l = 1;l<=n;l++)
61. {
62. for(int i = 1;i<=n-l+1;i++)
63. {
64. maxi = i+l-1;
65. *//i+l-1就是长度l时能到达的最大的数据下标*
66. e[i][maxi] = MAX;
67. w[i][maxi] = w[i][maxi-1]+p[maxi]+q[maxi];
68. *//i--j的总概率和即等于i--j-1的加上新增的maxi处的k和d的概率*
69. for(int j = i;j<=maxi;j++)
70. {
71. temp = e[i][j-1]+e[j+1][maxi]+w[i][maxi];
72. if(temp < e[i][maxi])
73. {
74. e[i][maxi] = temp;
75. root[i][maxi] = j;
76. }
77. }
78. }
79. }
80. cout << "最小搜索期望为：" << e[1][n] << '\n';
81. cout << "树的形状如下：" << '\n';
82. printOBST(1,n);
83. }
84. return 0;
85. }

## 免费馅饼

### Problem Description

都说天上不会掉馅饼，但有一天gameboy正走在回家的小径上，忽然天上掉下大把大把的馅饼。说来gameboy的人品实在是太好了，这馅饼别处都不掉，就掉落在他身旁的10米范围内。馅饼如果掉在了地上当然就不能吃了，所以gameboy马上卸下身上的背包去接。但由于小径两侧都不能站人，所以他只能在小径上接。由于gameboy平时老呆在房间里玩游戏，虽然在游戏中是个身手敏捷的高手，但在现实中运动神经特别迟钝，每秒种只有在移动不超过一米的范围内接住坠落的馅饼。现在给这条小径如图标上坐标：   
  
  
  
 为了使问题简化，假设在接下来的一段时间里，馅饼都掉落在0-10这11个位置。开始时gameboy站在5这个位置，因此在第一秒，他只能接到4,5,6这三个位置中期中一个位置上的馅饼。问gameboy最多可能接到多少个馅饼？（假设他的背包可以容纳无穷多个馅饼）

### Input

输入数据有多组。每组数据的第一行为以正整数n(0 < n < 100000)，表示有n个馅饼掉在这条小径上。在结下来的n行中，每行有两个整数x,T(0 <= T < 100000),表示在第T秒有一个馅饼掉在x点上。同一秒钟在同一点上可能掉下多个馅饼。n=0时输入结束。

### Output

每一组输入数据对应一行输出。输出一个整数m，表示gameboy最多可能接到m个馅饼。  
  
提示：本题的输入数据量比较大，建议用scanf读入，用cin可能会超时。

### Example Input

6

5 1

4 1

6 1

7 2

7 2

8 3

0

### Example Output

4

### 代码：

1. *#include <stdio.h>*
2. *#include <stdlib.h>*
3. int a, b;
4. int s3[100010][13];
5. int s4[100010][13];
6. int main()
7. {
8. int n, i, max, j;
9. while(scanf("%d", &n) != EOF)
10. {
11. if(n == 0)
12. return 0;
13. memset(s3,0,sizeof(s3));
14. memset(s4,0,sizeof(s4));
15. for(i = 1; i <= n; i++)
16. {
17. scanf("%d%d", &a, &b);
18. s3[b][a]++;
19. if(i == 1)
20. {
21. max = b;
22. }
23. else
24. {
25. if(max < b)
26. {
27. max = b;
28. }
29. }
30. }
31. for(i = 0; i <= 10; i++)
32. {
33. s4[max][i] = s3[max][i];
34. }
35. for(i = max - 1; i >= 0; i--)
36. {
37. for(j = 0; j <= 10; j++)
38. {
39. if((j - 1) >= 0 && (j + 1) <= 10)
40. {
41. max = s4[i + 1][j];
42. if(s4[i + 1][j - 1] > max)
43. max= s4[i + 1][j - 1];
44. if(s4[i + 1][j + 1] > max)
45. max = s4[i + 1][j + 1];
46. s4[i][j] = s3[i][j] + max;
47. }
48. else if((j - 1) >= 0 && (j + 1) > 10)
49. {
50. max = s4[i + 1][j];
51. if(s4[i + 1][j - 1] > max)
52. max = s4[i + 1][j - 1];
53. s4[i][j] = s3[i][j] + max;
54. }
55. else if((j - 1) < 0 && (j + 1) <= 10)
56. {
57. max = s4[i + 1][j];
58. if(s4[i + 1][j + 1] > max)
59. max = s4[i + 1][j + 1];
60. s4[i][j] = s3[i][j] + max;
61. }
62. }
63. }
64. printf("%d\n", s4[0][5]);
65. }
66. return 0;
67. }