****

算法设计与分析

实验报告(动态规划)

**学生姓名 杨凯楠**

**学 号** 8208201004

**专业班级 信息安全2002班**

**指导教师** 石峰

**学 院** 计算机学院

**完成时间** 2021年12月18日

**一、实验目的**：理解动态规划的基本思想，理解动态规划算法的两个基本要素最优子结构性质和子问题的重叠性质。熟练掌握典型的动态规划问题。掌握动态规划思想分析问题的一般方法，对较简单的问题能正确分析，设计出动态规划算法，并能快速编程实现。

**二、实验内容**：编程实现讲过的例题：最长公共子序列问题、矩阵连乘问题、凸多边形最优三角剖分问题、电路布线问题等。本实验中的问题，设计出算法并编程实现。

1. 石子合并

在一个圆形操场的四周摆放着n堆石子(n<= 100)，现要将石子有次序地合并成一堆。规定每次只能选取相邻的两堆合并成新的一堆,并将新的一堆的石子数,记为该次合并的得分。编一程序，由文件读入堆栈数n及每堆栈的石子数(<=20)。

选择一种合并石子的方案,使得做n－1次合并,得分的总和最小；

输入数据：

第一行为石子堆数n；

第二行为每堆的石子数,每两个数之间用一个空格分隔。

输出数据：

从第一至第n行为得分最小的合并方案。第n+1行是空行.从第n+2行到第2n+1行是得分最大合并方案。每种合并方案用n行表示，其中第i行(1<=i<=n)表示第i次合并前各堆的石子数(依顺时针次序输出，哪一堆先输出均可)。要求将待合并的两堆石子数以相应的负数表示。

**Sample Input**

4

4 5 9 4

**Sample Output**

－4 5 9 －4

－8 －5 9

－13 －9

22 4 －5 －9 4

4 －14 －4

－4 －18

22

1. 最小代价子母树

设有一排数，共ｎ个，例如：22 14 7 13 26 15 11。任意2个相邻的数可以进行归并，归并的代价为该两个数的和,经过不断的归并，最后归为一堆，而全部归并代价的和称为总代价，给出一种归并算法，使总代价为最小。

输入、输出数据格式与“石子合并”相同。

**Sample Input**

4

12 5 16 4

**Sample Output**

－１２ －５ １６ ４

１７ －１６ －４

－１７ －２０

３７

1. 求能获得的最大喜爱度

输入描述：

第一行输入为：能使用的钱的总额 X 和商品种类的总数 N，两个数字通过空格隔开

第二行开始，每行为一个单一的商品，包括的信息是：商品单价 P、该商品个数 amt、该商品喜爱度 fav

示例：

输入：

10 4

2 2 2

5 2 2

4 1 3

9 1 3

输出：

7 2

1 2

3 1

解释：用 10 块钱，最大的喜爱度是7，购买2种商品：购买 第一种商品2 个，得到的喜爱度是 4；再购买第三种商品 1 个，得到的喜爱度是 3。

1. 一个正整数数组 arr, 数组满足 0 <= arr[i] <= 9。挑出任意个数，组成一个最大数，并且这个数能被 3 整除，并以字符串的形式返回
2. \*基因问题

已知两个基因序列如s：AGTAGT；t：ATTAG。现要你给序列中增加一些空格后，首先使得两个序列的长度相等，其次两个串对应符号匹配得到的值最大。基因只有四种分别用A、G、C、T表示，匹配中不允许两个空格相对应，任意两符号的匹配值由下表给出：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | G | C | T | ︺ |
| A | 5 | -2 | -1 | -2 | -4 |
| G | -2 | 5 | -4 | -3 | -2 |
| C | -1 | -4 | 5 | -5 | -1 |
| T | -2 | -3 | -5 | 5 | -2 |
| ︺ | -4 | -2 | -1 | -2 |  |

**三、具体设计**

石子合并是最小代价子母树的升级版，故而先叙述最小代价子母树。

**1.最小代价子母树**

**分析**：设dp[i][k]表示以i为起点，长度为k的直线上各堆石子的最优合并状态

sum[i][k]表示以i为起点，长度为k的直线上各堆石子的总分，对于每一条长度为k的直线有k-1种划分方法，枚举后，就求出最优值

如：4 4 5 9,看成已经合并后的一个大堆，共有3种划分，即

4 459;44 59;445 9;看成是这三种情况下每两个堆的合并

dp[1][4]=dp[1][1]+dp[2][3]+sum[1][4]

=dp[1][2]+dp[3][2]+sum[1][4]

=dp[1][3]+dp[4][1]+sum[1][4]

子问题再类似分解

dp[2][3]=dp[2][1]+dp[3][2]+sum[2][3]

=dp[2][2]+dp[][]+sum[2][3]

dp[2][2]=dp[2][1]+dp[3][1]+sum[2][2]

dp[1][2]=dp[1][1]+dp[2][1]+sum[1][2]

dp[3][2]=dp[3][1]+dp[4][1]+sum[3][2]

dp[1][3]=dp[1][1]+dp[2][2]+sum[1][3]

=dp[1][2]+dp[3][1]+sum[1][3]

而dp[1][1]=4;dp[2][1]=4;dp[3][1]=5;dp[4][1]=9;

从上递归或从下递推都能求得最优值

也就是要把dp[][]和sum[][]这两张表填写完毕

按照假设，这两张表要按列来填

本道题与第一道题类似，反而比第一到题稍微简单一点

**详细设计**：

1. #include <cstring>
2. #include<iostream>
3. #include<algorithm>
4. **using** **namespace** std;
5. **int** sum[20][20] = { 0 };
6. **int** dp[20][20] = { 0 };
7. **int** wealth[20] = { 0 };
8. **int** n = 0;
9. **int** flag[20][20] = { 0 };
10. **int** cnt;
11. **void** setmy() {
12. cin >> n;
13. cnt = n;
14. **for** (**int** i = 1; i <= n; i++) {
15. cin >> wealth[i];
16. }
17. memset(dp, -1, **sizeof**(dp));
18. **for** (**int** i = 0; i <= n; i++) {
19. sum[i][1] = wealth[i];
20. }
21. **for** (**int** i = 2; i <= n; i++) {
22. **for** (**int** j = 1; j <= n - 1 && i + j <= n + 1; j++)
23. sum[j][i] += sum[j + 1][i - 1] + sum[j][1];
24. }
25. }
26. **int** Min(**int** s, **int** l) {
27. **if** (dp[s][l] != -1) {
28. **return** dp[s][l];
29. }
30. **int** temp=0;
31. cnt--;//--------------------
32. dp[s][l] = 10000;
33. **if** (l == 1) {
34. dp[s][1] = 0;
35. **return** 0;
36. }
37. **for** (**int** i = 1; i < l; i++) {
38. temp = Min(s,i) + Min(s + i,l - i) + sum[s][l];
40. **if** (temp < dp[s][l]) {
41. dp[s][l] = temp;
42. }
43. }
44. **return** dp[s][l];
45. }
46. **int** main(**void**) {
47. setmy();
48. cout << Min(1, n);
49. **return** 0;
50. }

**实现**：



**调试分析**：

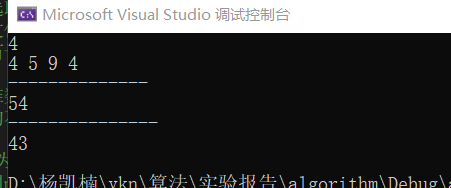
这道题我写了很久的记录路径，但还是没成功，是我的水平太差了。其实这道题是我写的第一道动态规划题，本身对动态规划处于初步认识的阶段，所以我在各种出错以后从网上别人写的博客中学习了一下，但之后的动态规划题就都是自己写的了。

**2.石子归并**

分析：本题是最小代价子母树的升级。最重要的一步就是变环为线，对其取模就好。

Demo：[click\_here](7_石子合并.cpp)(代码就不贴了，因为和第一道题高度相似，放一个超链接在这里)

**运行结果**：



**3. 求能获得的最大喜爱度**

输入描述：

第一行输入为：能使用的钱的总额 X 和商品种类的总数 N，两个数字通过空格隔开

第二行开始，每行为一个单一的商品，包括的信息是：商品单价 P、该商品个数 amt、该商品喜爱度 fav

示例：

输入：

10 4

2 2 2

5 2 2

4 1 3

9 1 3

输出：

7 2

1 2

3 1

解释：用 10 块钱，最大的喜爱度是7，购买2种商品：

购买 第一种商品2 个，得到的喜爱度是 4；再购买第三种商品 1 个，得到的喜爱度是 3。

**分析**：这道题其实是背包问题的升级版，不同的是普通背包问题每种物品只有一种，而这道题每种物品可能不只有一种。本实验有两个限制条件，即在规定的金额内，规定的物品数量内达到最大的喜爱度。

dp[money]表示用money多的钱获得的最大喜爱度。

mem[j]表示在花费为j多钱的情况下新增的商品序号和个数。

**核心过程**：

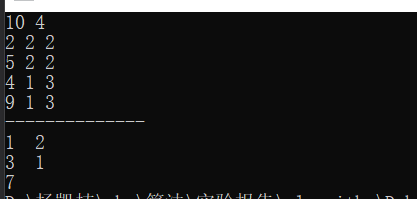
for (int k = 1; k <= dat[i].amt && money - k \* dat[i].p >= 0; k++)

dp[j] = max(dp[j], dp[j - k \* dat[i].p] + k \* dat[i].fav);

**详细设计**：

1. #include<iostream>
2. #include<algorithm>
3. **using** **namespace** std;
4. **int** money;
5. **int** n;
6. **struct** Data {
7. **int** p;
8. **int** amt;
9. **int** fav;
10. };
11. Data dat[20];
12. **int** dp[50] = { 0 };
13. **struct** Mem {
14. **int** num;
15. **int** amt;
16. };
17. Mem mem[20];
18. **void** setmy() {
19. cin >> money >> n;
20. **for** (**int** i = 1; i <= n; i++) {
21. cin >> dat[i].p >> dat[i].amt >> dat[i].fav;
22. }
24. }
26. **int** main(**void**) {
27. setmy();
28. **int** temp = 0;
29. **for** (**int** i = 1; i <= n; i++) {
30. **for** (**int** j = money; j > dat[i].p; j--) {
31. **for** (**int** k = 1; k <= dat[i].amt && money - k \* dat[i].p >= 0; k++) {
32. temp = dp[j];
33. dp[j] = max(dp[j], dp[j - k \* dat[i].p] + k \* dat[i].fav);
34. **if** (dp[j] != temp) {
35. mem[j].amt = k;
36. mem[j].num = i;
37. }
38. }
39. }
40. }
41. temp = 0;
42. printf("--------------\n");
43. **for** (**int** i = 0; i <= money; i++) {
44. **if** (mem[i].num == temp) {
45. **continue**;
46. }
47. printf("%d  %d\n", mem[i].num,mem[i].amt);
48. temp = mem[i].num;
49. }
50. cout << dp[money];
51. **return** 0;
52. }

**实现**：



**测试分析**：本题在输出具体装入问题时采用前后对比法，因为dp[]就包含了前一子问题的结果，所以采用从头开始的前后两两对比法，当发生变化时，说明该物品被装入背包内，且装入背包的个数也是很容易确定的。

**4. 一个正整数数组 arr, 数组满足 0 <= arr[i] <= 9。**

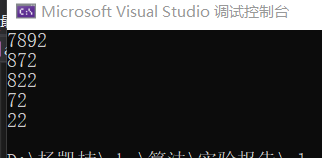
挑出任意个数，组成一个最大数，并且这个数能被 3 整除，并以字符串的形式返回

**分析**：

本题用动态规划很好解决，而我在这道题中另开辟了一种算法，现在介绍这种算法。首先我们知道，一个数能被3整除的充要条件是这个数的各个位加起来也能被三整除，接着引入离散数学中同余和等价关系的概念，不难得出，数组长度，即数的个数只能有三种情况：[0],[1],[2]。而数组中每个数也只有[0],[1],[2]三种情况，不难看出，当去掉n个数中的至多两个数时，该问题必有最优解，经过分析，最坏时间复杂度为O（n2）,但是如果对数组进行预处理，绝大多数情况下可以在很短时间内解决，且位数越多解题速度越快。

**详细设计**：

1. #include<cstdio>
2. #include<algorithm>
3. #include<ctime>
4. #include<vector>
5. **using** **namespace** std;
6. **bool** cmp(**int** a,**int** b){
7. **return** a > b;
8. }
9. **int** main(**void**) {
10. unsigned  **long** seed = time(0);
11. srand(seed);
12. **int** tempsum = 0;
13. **int** flag = 0;
14. unsigned **long** **long** num = **long** **long**(rand());
15. **int** arr[20], len = 0;
16. printf("%lld\n", num);
17. vector<**int**>a,b;
18. **for** (len = 0; num; len++) {
19. arr[len] = num % 10;
20. num /= 10;
21. tempsum += arr[len];
22. a.push\_back(arr[len]);
23. b.push\_back(arr[len]);
24. }
25. sort(a.begin(), a.end(), cmp);
26. sort(b.begin(), b.end(), cmp);
27. **if** (tempsum % 3 == 0) {
28. sort(arr, arr + len, cmp);
29. **for** (**int** i = 0; i < len; i++) {
30. printf("%d", arr[i]);
31. }
32. **return** 0;
33. }
34. **for** (**int** i = 0; i < len; i++) {
35. **if** ((tempsum - arr[i]) % 3 == 0) {
36. flag = 0;
37. a.erase(a.begin() + i);
38. sort(a.begin(), a.end(),cmp);
39. **for** (**int** k = 0; k <a.size(); k++) {
40. printf("%d", a[k]);
41. }printf("\n");
42. a.insert(a.begin() + i, arr[i]);
43. }
44. }
45. **for** (**int** i = 0; i < len-1; i++) {
46. **for** (**int** j = i + 1; j < len; j++) {
47. **if** ((tempsum - arr[i] - arr[j]) % 3 == 0) {
48. b.erase(b.begin() + j);
49. b.erase(b.begin() + i);
50. sort(b.begin(), b.end(),cmp);
51. **for** (**int** k = 0; k < b.size(); k++) {
52. printf("%d", b[k]);
53. }
54. printf("\n");
55. b.insert(b.begin() + i, arr[i]);
56. b.insert(b.begin() + j, arr[j]);
57. }
58. }
59. }
60. **return** 0;
61. }

**运行结果**：

**测试分析**：

如图，生成的是数字7312，可以将所有去掉两位数字以内的三的倍数都求出来，并且输出时就可以将所有从大到小输出，不需要排序，（题目要求是最大的，需要时只用输出第一个就好）。

**5. 基因问题**

已知两个基因序列如s：AGTAGT；t：ATTAG。现要你给序列中增加一些空格后，首先使得两个序列的长度相等，

其次两个串对应符号匹配得到的值最大。基因只有四种分别用A、G、C、T表示，匹配中不允许两个空格相对应，

**分析**：

这道题是最大公共子序列的升级版，同样是从最后一个开始匹配

mv[i],[j]表示 s 串的前 i 个碱基和 t 串的前 j 个碱基的相似度match value

不同的是最后有三种情况

1.碱基与碱基匹配

2.碱基与空格匹配

3.空格与碱基匹配

则状态转移方程为mv[i][j]=max(mv[i][j],mv[i-1][j]+w[s[i]][k],mv[i][j-1]+w[k][t[j]])

其中s[i],t[j]为s,t,串中第i,j个字母，w为匹配值，k为空格

边界条件：递推到一条没了以后那么

mv[i][0]=mv[i-1][0]+w[s[i]][k],或mv[0][j]=mv[0][j-1]+w[k][t[j]]

双方都完了那么f[0][0]=0;

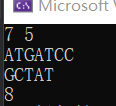
为了简化输入，提前输好每个串的长度

**详细设计**：

1. //#include<iostream>
2. //#include<algorithm>
3. #include<bits/stdc++.h>
4. **using** **namespace** std;
5. **int** lens = 0, lent = 0, k = 5;
6. **int** w[6][6] = {
7. {0,0,0,0,0,0},
8. {0,5,-2,-1,-2,-4},
9. {0,-2,5,-4,-3,-2},
10. {0,-1,-4,5,-5,-1},
11. {0,-2,-3,-5,5,-2},
12. {0,-4,-2,-1,-2,0}
13. };
14. **struct** Mem {
15. **int** s;
16. **int** t;
17. };
18. //Mem mem[30][30];
19. **int** s[200] = { 0 };
20. **int** t[200] = { 0 };
21. **int** mv[200][200] = { 0 };
22. **void** setmy() {
23. cin >> lens>>lent;
24. **char** a;
25. **for** (**int** i = 1; i <= lens; i++) {
26. cin >> a;
27. **switch**(a){
28. **case** 'A':s[i] = 1; **break**;
29. **case** 'G':s[i] = 2; **break**;
30. **case** 'C':s[i] = 3; **break**;
31. **case** 'T':s[i] = 4; **break**;
32. }
33. }
34. **for** (**int** i = 1; i <= lent; i++) {
35. cin >> a;
36. **switch** (a) {
37. **case** 'A':t[i] = 1; **break**;
38. **case** 'G':t[i] = 2; **break**;
39. **case** 'C':t[i] = 3; **break**;
40. **case** 'T':t[i] = 4; **break**;
41. }
42. }
43. }
45. **void** solution() {
46. mv[0][0] = 0;
47. **for** (**int** i = 1; i <= lens; i++) {
48. mv[i][0] = mv[i - 1][0] + w[s[i]][k];
49. }
50. **for** (**int** j = 1; j <= lent; j++) {
51. mv[0][j] = mv[0][j - 1] + w[k][t[j]];
52. }
53. **for** (**int** i = 1; i <= lens; i++)
54. **for** (**int** j = 1; j <= lent; j++) {
55. mv[i][j] = max(max(mv[i-1][j-1]+w[s[i]][t[j]], mv[i - 1][j] + w[s[i]][k]), mv[i][j - 1] + w[k][t[j]]);
56. }

59. printf("%d", mv[lens][lent]);
61. }
62. **int** main(**void**) {
63. setmy();
64. solution();
65. **return** 0;
66. }

**实现**：



**测试分析**：这道题实在不会写具体结果，水平还很低，技术不够，还需要学习。但是根据计算，结果是正确的，（放到平台上可以AC）。

**四、心得体会**

通过本次动态规划实验，我深刻理解到了动态规划算法的两个基本要素最优子结构性质和子问题的重叠性质，体会了动态规划解决问题的巧妙性。



**学生姓名 杨凯楠**

**学 号** 8208201004

**专业班级 信息安全2002班**

**指导教师** 石峰

**学 院** 计算机学院

**完成时间** 2021年11月18日

算法设计与分析

实验报告(贪心)

1. **实验目的**

掌握贪心算法

1. **实验内容**

1. 搬桌子问题

某教学大楼一层有n个教室，从左到右依次编号为1、2、…、n。现在要把一些课桌从某些教室搬到另外一些教室，每张桌子都是从编号较小的教室搬到编号较大的教室，每一趟，都是从左到右走，搬完一张课桌后，可以继续从当前位置或往右走搬另一张桌子。输入数据：先输入n、m，然后紧接着m行输入这m张要搬课桌的起始教室和目标教室。输出数据：最少需要跑几趟。

**Sample Input**

10 5

1 3

3 9

4 6

6 10

7 8

**Sample Output**

3

1. 分发饼干

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s[j] 。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

输入：数组g和数组s

输出：最多能满足的孩子个数

样例：

输入：g = [1,2,3], s = [1,1]

输出：1

输入：g = [1,2], s = [1,2,3]

输出：2

1. 加工生产调度

某工厂收到了 n 个产品的订单，这 n 个产品分别在 A、B 两个车间加工，并且必须先在 A 车间加工后才可以到 B 车间加工。

某个产品 i 在 A，B 两车间加工的时间分别为Ai,Bi。怎样安排这 n 个产品的加工顺序，才能使总的加工时间最短。

这里所说的加工时间是指：从开始加工第一个产品到最后所有的产品都已在 A，B 两车间加工完毕的时间。

输入：

第一行仅—个数据 n ，表示产品的数量；

接下来 n 个数据是表示这 n 个产品在 A 车间加工各自所要的时间；

最后的 n 个数据是表示这 n 个产品在 B 车间加工各自所要的时间。

输出：

第一行一个数据，表示最少的加工时间；

第二行是一种最小加工时间的加工顺序。

输入样例：

5

3 5 8 7 10

6 2 1 4 9

输出样例：

34

1 5 4 2 3

1. 课程表

这里有 n 门不同的在线课程，他们按从 1 到 n 编号。每一门课程有一定的持续上课时间（课程时间）t 以及关闭时间第 d 天。一门课要持续学习 t 天直到第 d 天时要完成，你将会从第 1 天开始。

给出 n 个在线课程用 (t, d) 对表示。你的任务是找出最多可以修几门课（你不能同时修两门课程）。

输入：n和n对数据（t，d）

输出：最多可以修多少门课

样例：

输入：4 [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

输出：3

样例解释：

这里一共有 4 门课程, 但是你最多可以修 3 门:

首先, 修第一门课时, 它要耗费 100 天，你会在第 100 天完成, 在第 101 天准备下门课。

第二, 修第三门课时, 它会耗费 1000 天，所以你将在第 1100 天的时候完成它, 以及在第 1101 天开始准备下门课程。

第三, 修第二门课时, 它会耗时 200 天，所以你将会在第 1300 天时完成它。

第四门课现在不能修，因为你将会在第 3300 天完成它，这已经超出了关闭日期。

1. **具体过程**

**1.搬桌子**

分析：本题采用贪心算法，即从第一个教室开始走，到达目标教室后从目标教室向后搜索，以此类推，直到所有教室需要搬的课桌都被处理。在存储类型中，我将课桌原本在的教室设为数组下表，目标教室为数组中存贮的信息。若该教室不搬桌子，就将其设为0。通过递归函数，走一步就将数组内容变换为下一次处理的数组下表。

**详细设计：**

1. #include<iostream>
2. #include<algorithm>
3. **using** **namespace** std;
4. **int** mov[100] = { 0 };
5. **int** n,m;
6. **void** set() {
7. cin >> n>>m;
8. **int** j = 0, k = 0;
9. **for** (**int** i = 0; i < m; i++) {
10. cin >> j >> k;
11. mov[j] = k;           //mov 上线到n
12. }
13. }
14. **void** solution(**int** i) {
15. **int** j=0;
16. **for** (i; i <= n; i++) {
18. **if** (mov[i] != 0) {
19. j = mov[i];
20. mov[i] = 0;
21. **if** (j <= n)
22. solution(j);
23. **break**;
24. }
25. }
26. }
28. **int** main(**void**) {
29. set();
30. **int** c = 0;
31. **for** (**int** i = 1; i <= n; i++) {
32. **if** (mov[i] != 0) {
33. c++;
34. solution(i);
35. }
36. }
37. cout << c;
38. **return** 0;
39. }

**实现**：



**测试分析**：本类问题需要将数对按出发教室排序，而由于我将出发教室设为了数组下标，故而不需要进行排序，因为数组下标本身就是排好序的。如图有十个教室，五组数对最后需要三次。

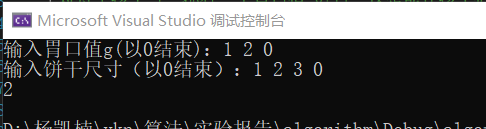
**2. 分发饼干**

分析：本题先对输入的胃口值、饼干尺寸进行非递减排序，然后建立一个二重循环，以人的胃口值为基准，直到找到一个能满足该胃口的饼干为止，当饼干遍历一次后算法结束。在此算法中可以进行相应的优化，比如在对胃口值和饼干大小排好序后，对于每一个胃口值找饼干不必要从头开始，因为下一个人的胃口一定大于等于此人胃口，故而从该处的饼干往后寻找饼干即可。如此其实算法的时间复杂度可能只有O(n),n为饼干个数。

**详细设计**：

1. #include<iostream>
2. #include<stdio.h>
3. #include<algorithm>
4. **using** **namespace** std;
5. **int** g[100] = { 0 };
6. **int** s[100] = { 0 };
7. **int** leng, lens, countnum = 0;
9. **void** set() {
10. **int** i;
11. printf("输入胃口值g(以0结束)：");
12. **for** ( i = 0; i<100; i++) {
13. cin >> g[i];
14. **if** (g[i] == 0) {
15. **break**;
16. }
17. }
18. leng = i;
19. printf("输入饼干尺寸（以0结束）：");
20. **for** ( i = 0; i<100; i++) {
21. cin >> s[i];
22. **if** (s[i] == 0) {
23. **break**;
24. }
25. }
26. lens = i;
27. }
28. **void** solution() {
29. **int** j = 0;
30. **for** (**int** i = 0; i < leng; i++) {
31. **for** ( j ; j < lens; j++) {
32. **if** (g[i] <= s[j] ) {
33. countnum++;
34. **break**;
35. }
36. }
37. }
38. }
39. **int** main(**void**) {
40. set();
41. sort(s, s + lens);
42. sort(g, g + leng);
43. solution();
44. cout << countnum << endl;
45. **return** 0;
46. }

**实现**：



**测试分析**：

如图，胃口值为1,2，饼干尺寸为1,2,3，则可以满足两个人的胃口。

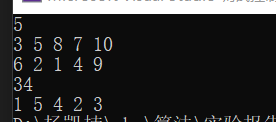
**3.加工生产调度**

**分析**：令a为A中时间，b为B中时间，设 A[]为 a < b 的作业集合，B[] 为 a >= b 的作业集合，将 A[] 的作业按 a 升序排序，B[] 中的作业按照 b 降序排序，则 A[] 作业接 B[] 构成最优顺序。在此问题中，最重要的是考虑两车间的空闲情况，空闲情况分别有1：A车间加工第一个产品时，B车间一定空闲2：在此之后，A车间一直持续工作直到加工完属于自己的任务。3：A车间加工a<b的作业时，B车间除开始的空闲外，一定不会空闲，直到A车间加工完B[]中第一个，此时分两种情况：B车间已经加工完A[]的产品了，此时他可能空闲了一段时间，也可能没有，若B车间还没有加工完A[]的产品，则他一定不会空闲4：以此类推，在A加工最后一件产品后，有两种情况B车间能直接加工最后一件，或B还没有加工完除最后一件以外的产品，如此即可以计算出总得时间。

**详细设计**：

**Demo：**

1. #include<iostream>
2. #include<algorithm>
3. #include<vector>
4. **using** **namespace** std;
5. **int** n;
6. **struct** Data {
7. **int** a;
8. **int** b;
9. **int** num;
10. };
12. Data dat[20];
13. Data A[20], B[20];
15. **bool** cmp1(Data x, Data y) {
16. **return** x.a < y.a;
17. }
18. **bool** cmp2(Data x, Data y) {
19. **return** x.b > y.b;
20. }
21. **void** cal(**int** lena,**int** lenb) {
22. **int** tot = 0;//A中顺序执行，A空闲，主要发生在A已经完了，B还在工作，B中空闲主要发生在A中工作，B手头空闲，以及刚开始。
23. **int** temp1 = 0, temp2 = 0,temp3=0;
24. tot += A[0].a;//
25. temp3 += A[0].a;
26. //temp1 += tot;
27. **for** (**int** i = 0; i < lena + lenb; i++) {
28. **if** (i < lena) {
29. tot += A[i].b;
30. temp1 += A[i].a;
31. temp3 += A[i].b;
32. }
33. **else** **if** (i == lena) {
34. //tot += A[i].b;
35. temp3 += A[i].b;
36. **if** (temp1 + A[lena].a < tot) {//第一车间加工完第二组第一个了，第二车间还没加工完第一组
37. tot = temp1 + A[i].a;
38. }
39. **else** **if** (temp1 + A[lena].a >= tot) {//第一车间还没有加工完第二组第一个，dierchejian 就已经加工完第一组
40. temp2= A[0].a+(temp1+A[i].a-tot);//temp2为第二车间AB两组转换中间等待空闲时间
41. tot = temp1 + A[i].a;
42. }
43. temp3 += temp2;
44. }
45. **else** {
46. tot += A[i].a;
47. temp3 += A[i].b;
48. }
50. }
51. **if** (temp3 < tot) {
52. tot += A[lena + lenb - 1].b;
53. }
54. **else** {
55. tot = temp3 + A[lena + lenb - 1].b;
56. }
57. cout << tot << endl;
58. //cout << "temp2=" << temp2 << endl;
59. **for** (**int** i = 0; i < lena + lenb; i++) {
60. cout << A[i].num << " ";
61. }
62. }
63. **int** main(**void**) {
64. cin >> n;
65. **for** (**int** i = 0; i < n; i++) {
66. cin >> dat[i].a;
67. dat[i].num = i + 1;
68. }
69. **for** (**int** i = 0; i < n; i++) {
70. cin >> dat[i].b;
71. }
73. **int** j = 0, k = 0;
74. **for** (**int** i = 0; i < n; i++) {
75. **if** (dat[i].a < dat[i].b) {
76. A[j++]=dat[i];
77. }
78. **else** {
79. B[k++]=dat[i];
80. }
81. }
82. **int** lena=j, lenb=k;
84. sort(A, A + lena, cmp1);
85. sort(B, B + lenb, cmp2);
86. **for** (**int** i = 0; i < lenb; i++) {
87. A[lena + i] = B[i];
88. }
89. cal(lena,lenb);
90. **return** 0;
91. }



**实现：**

**测试分析：**如图输入5个产品，上面一组为A车间加工时间，下面一组为B车间加工时间，最后需要时间为34，加工顺序为1 5 4 2 3，与题目所给的测试样例一致。

**4.课程表问题**

**分析**：

先按结束时间排序，再依次选择，如果已上课时间加上该次课程时间<该课结束时间则选择上。如果只用贪心的话，如果出现3 6 7，3 8,5 8这样的数据，输出结果为1，但是明显结果应该是2，所以应该有所改进。即在已经选择的课程中将时间最长的与该课程时间比较，如果该课程时间较短则替换。在这一过程中可以选择优先队列或者插入排序，可以达到更快的算法效率，否则需要每做一次就排序一次，很麻烦。

**详细设计**：

1. #include<iostream>
2. #include<algorithm>
3. **using** **namespace** std;
4. **int** n;
5. **struct** Data {
6. **int** t;
7. **int** d;
8. };
9. Data dat[100];
10. **int** flag[100] = { 0 };
11. **bool** cmp(Data x, Data y) {
12. **return** x.d < y.d;
13. }
14. **int** find(**int** n) {
15. **int** temp=0,tempi=0;
16. **for** (**int** i = 0; i < n+1; i++) {
17. **if** (temp < dat[i].t && flag[i]==1) {
18. temp = dat[i].t;
19. tempi = i;
20. }
21. }
22. **return** tempi;
23. }
24. **int** main(**void**) {
25. cin >> n;
26. **for** (**int** i = 0; i < n; i++) {
27. cin >> dat[i].t >> dat[i].d;
28. }
29. //-----------输入完毕
30. sort(dat, dat + n, cmp);
32. **int** tl=0,temp=0,tempi=0,cnt=0;
33. **for** (**int** i = 0; i < n; i++) {
35. tl += dat[i].t;
36. flag[0] = 1;
37. **if** (tl <= dat[i].d) {
38. flag[i]=1;
39. tempi = find(i);
40. cnt++;
41. }
42. **else** {
43. **if** (dat[tempi].t > dat[i].t) {
44. tl -= dat[tempi].t;
45. flag[tempi] = 0;
46. flag[i] = 1;
47. tempi = find(i);
48. }
49. **else** {
50. tl -= dat[i].t;
51. }
52. }
53. }
54. cout << cnt;
55. }

**实现**：

**测试分析**：如图所示，最多可以选择三门，与所给的样例结果相一致。

1. **心得体会**

本次贪心算法实验较为简单，唯一有难度的就是加工生产调度一题时间计算方法比较烧脑，而经过本次实验我深刻的体会到了在编写程序时又有一个清晰的逻辑的重要性。这也是算法实验的最后一个心得体会了，经过这学期的算法学习与实验探究，我受益颇多，脑海中产生了一个坚定的想法并且逐渐深信不疑：算法才是程序的灵魂所在，这也是我深深爱上了算法。