第5章

贪心算法

5.1 找零问题

◇5.1.1 实验目的及要求

(1) 理解贪心算法求解问题的思路。

(2) 练习编程实现贪心算法解决找零问题。

(3) 思考什么问题可以考虑使用贪心算法。

◇5.1.2 实验内容

试编程使用贪心算法求解找零钱问题。

设有 50,20,10,5,1,0.5,0.1 等面额的零钱,顾客购物花费 n 元,该顾客只有整数张 100 元,在支付(n/100+1)×100 元后(恰好是>n 的最小的 100 倍数),收银员应如何找零,才能使找回的钱的张数最少?

输入:n,表示顾客购物花费,最多包含一位小数。

输出:找零钱的各面额所对应张数。

样本输入 1:

```
67.5
```

样本输出:

```
0 1 1 0 2 1 0
```

样本输入 2:

```
243
```

样本输出:

```
1 0 0 1 2 0 0
```

◇5.1.3 实验原理

贪心算法是指在对问题求解时，总是做出在当前看来最好的选择。不从整体最优上加以考虑，所做出的仅仅是在某种意义上的局部最优解，所以贪心算法不是对所有问题都能得到整体最优解。关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

贪心算法分析问题的思路分为以下几步。

(1) 将问题分解为若干个子问题。

(2) 找出适合的贪心策略。

(3) 求解每一个子问题的最优解。

(4) 将局部最优解堆叠成全局最优解。

对于此题，在金额一定的情况下，想要找零钱的张数最少，需要面额尽可能大。所以如果一张一张地找零钱，每次都选择小于剩余钱数的最大面额，就能保证总张数最小。

找零钱的贪心算法例解：如果花费 1 元，总共需要找 99 元。50＜99，所以首先考虑找 50 元的，能找 99/50＝1 张；然后第二大面额 20 元，能找 49/20＝2 张；剩下 9 元，小于 9 的最大面额，10 元不符合，那么就考虑 5 元，以此类推。每次考虑当前看起来最优的选择，也就是找当前所能找的最大面额。

◇5.1.4 实验步骤

结合例解的步骤，试着应用贪心算法的思路编程实现找零问题算法。

(1) 设置可找零的面额数组，并从大到小排序；设置结果数组记录对应的面额需要找几张。

(2) 计算需要找零的总额。

(3) 从最大面额开始，依次判断每种面额需要的数量。

(4) 打印结果数组中的值。

◇5.1.5 参考代码

参考代码如下。

```
1. #include <iostream>
2. using namespace std;
3. //零钱面额的种类数
4. #define N 7
5. int main()
6. {
7. double m[N] = { 50, 20, 10, 5, 1, 0.5, 0.1 };        //定义每种零钱的面额
```

```
8. int result[N] = { 0, 0, 0, 0, 0, 0, 0 };    //保存每种零钱个数的结果,注意应该是整数
9.
10. double n;                                   //输入顾客的花费 n 元,最多一位小数
11. cin >> n;
12. //根据题目,顾客实际支付(n / 100 + 1) * 100 元,注意应该是整数张 100 元
13. int num = (int(n / 100) + 1) * 100;
14. double change = num - n;                    //应找的金额
15.
16. //贪心算法开始
17. for (int i = 0; i < N; i++)
18. {
19. result[i] = change / m[i];                  //从大面额开始,贪心算法取最多的数量
20. change = change - result[i] * m[i];        //计算剩余未找的钱
21. }
22.
23. //打印结果
24. for (int i = 0; i < N; i++)
25. {
26. cout << result[i] << " ";
27. }
28. cout << endl;
29. }
```

◇5.1.6 实验结果

(1) 实验结果验证如图 5-1 所示。

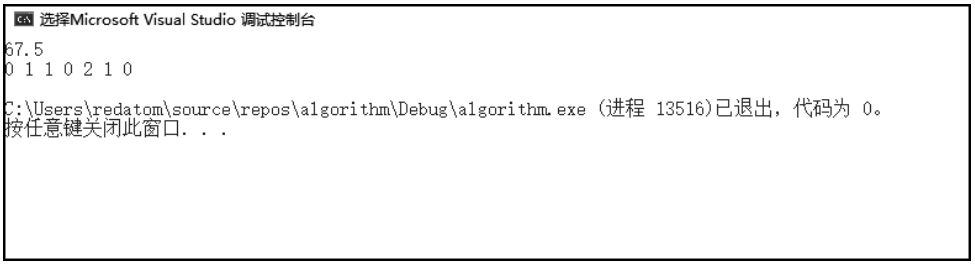

图 5-1 实验结果截图

(2) 算法复杂度分析。

找零数额一定小于 100,且只需要按照面额循环一次就可以得到结果,而面额数是一个常数,所以时间复杂度为 $O(1)$。

◇5.1.7 实验总结

下面对本次实验进行结论陈述。

(1) 总结和学习贪心算法的解题思路,以及贪心算法分析问题的几个步骤。

(2) 思考什么问题可以考虑用贪心算法。

(3) 思考贪心算法为什么不一定能获得最优解。

5.2 活动安排问题

◇5.2.1 实验目的及要求

(1) 结合一些现实生活中的问题,进一步理解贪心算法求解问题的思路。

(2) 练习编程实现贪心算法解决活动安排问题。

◇5.2.2 实验内容

试编程使用贪心算法求解会场活动安排问题。

假设要在某一会场安排一批活动,并希望在不发生冲突的情况下,安排尽可能多的活动,同一时间最多安排一个活动。设计一个有效的算法判断最多能安排的活动数量。如果上一个活动在 t 时间结束,下一个活动最早应该在 $t+1$ 时间开始。

输入:每组测试数据的第一行是一个整数 $n(1<n<10\ 000)$,表示该测试数据共有 n 个活动。随后的 n 行,每行有两个正整数 B_i,$E_i(0\leqslant B_i, E_i<10\ 000)$,分别表示第 i 个活动的起始与结束时间($B_i\leqslant E_i$)。

输出:对于每一组输入,输出最多能够安排的活动数量。

样例输入:

```
输入活动数量:
3
输入每个活动的开始时间和结束时间:
1 10
10 11
11 20
```

样例输出:

```
2
```

◇5.2.3 实验原理

首先分析此问题的目标,不考虑活动的质量,只期望尽可能多地安排活动数量,而需要有更多的可用时间才能安排更多活动。对于期望某种结果尽可能多或少的问题,可以考虑使用贪心算法。分析此问题,试设想如果某活动安排后能剩余尽可能多的时间留给其他活动,则对活动数量最大化更加有利。对于此设想,反过来用贪心算法的思路验证,每次选择活动使得总活动数加 1,同时剩余的时间尽可能多,再做当前状态下的

最优选择。

其次注意题目中有一个限制条件，活动不能冲突，即 $B_i+1>E_i$（下一活动开始时间＞上一活动的结束时间），可以称作相容性，满足相容性的活动才能继续安排，否则应淘汰，检查下一个活动。

根据以上两点，只要将所有活动按照结束时间排序，从第一个活动开始，依次检查相容性，相容的活动保留，最后将获得可以安排的活动最大数量。

◇5.2.4 实验步骤

结合上述思路，应用贪心算法解决活动安排问题的步骤如下。

（1）定义一个结构体 Node 存储所需要计算的活动的相关时间，其中，begin 是开始时间值，end 是结束时间值。

（2）对所有的活动按结束时间排序。

（3）该算法应用贪心策略做选择的意义是，使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。依次取出已排序的数据，第一个数据直接选用，从第二个开始判断相容性，即 $B_i+1>E_i$ 则活动数量 Count++，否则舍弃此活动，继续循环判断下一个活动。

根据此步骤，试使用贪心算法完成对活动安排问题的编码。

◇5.2.5 参考代码

参考代码如下。

```
1. #define _CRT_SECURE_NO_WARNINGS
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5.
6. struct Note                                          //活动
7. {
8. int begin;                                          //开始时间
9. int end;                                            //结束时间
10. };
11.
12. void quicksort(int left, int right, struct Note q[]); //快速排序法
13. int activity_selector(int t, struct Note q[]);      //计算可以安排的活动数量
14.
15. int main()                                         //主函数
16. {
17. int result, i;
```

```
int t;
printf("输入活动数量:\n");
scanf("%d", &t);
struct Note q[100];
printf("输入每个活动的开始时间和结束时间:\n");
for (i = 1; i <= t; i++)
{
scanf("%d%d", &q[i].begin, &q[i].end);
}
quicksort(1, t, q);                    //排序
result = activity_selector(t, q);    //贪心算法求解
printf("%d\n", result);
return 0;
}

//快速排序算法
void quicksort(int left, int right, struct Note q[])
{
int temp = q[left].end;
int temp1 = q[left].begin;
int i, j, t, f;
i = left;
j = right;
if(right < left)
{
return;
}
while (i != j)
{
while (i < j && q[j].end >= temp)
{
j--;
}
while (i < j && q[i].end <= temp)
{
i++;
}
if(i < j)
{
t = q[i].end;
q[i].end = q[j].end;
q[j].end = t;
f = q[i].begin;
q[i].begin = q[j].begin;
```

```
62. q[j].begin = f;
63. }
64. }
65. q[left].end = q[i].end;
66. q[i].end = temp;
67. q[left].begin = q[i].begin;
68. q[i].begin = temp1;
69. quicksort(left, i - 1, q);
70. quicksort(i + 1, right, q);
71. }
72.
73. //计算可以安排的活动数量
74. int activity_selector(int t, struct Note q[])
75. {
76. int count = 1, i = 1;
77. int j;
78. for (j = 2; j <= t; j++)
79. {
80. if(q[j].begin > q[i].end)
81. {
82. i = j;          //更新最后一个已安排的活动
83. count++;
84. }
85. }
86. return count;
87. }
```

◇5.2.6 实验结果

(1) 实验结果验证如图 5-2 所示。

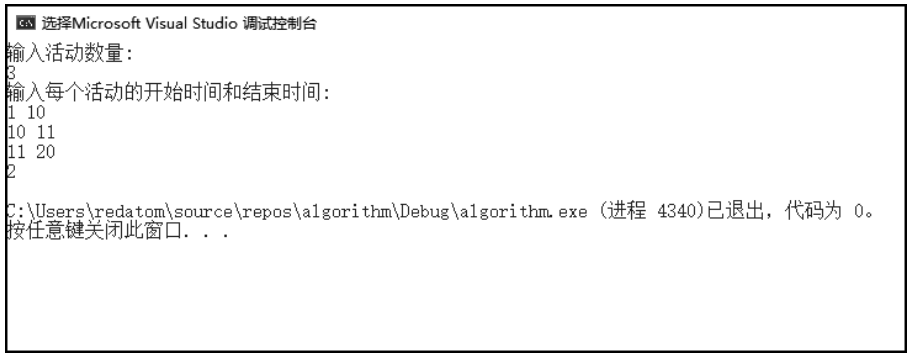

图 5-2 实验结果截图

(2) 算法复杂度分析。

抛开排序算法，只关注贪心算法选择活动的过程，只使用了一层循环，且算法复杂

度与活动规模 n 相关，所以活动选择部分的时间复杂度为 $O(n)$。加上排序使用的快速排序算法，总体时间复杂度为 $O(n\log n)$。

◇5.2.7 实验总结

下面对本次实验进行结论陈述。

(1) 总结和学习贪心算法的解题思路，对期望获得最大或最小值的问题，可尝试用贪心算法的思路做假设分析。

(2) 复习了排序算法，参考代码使用了快速排序。

5.3 普通背包问题

◇5.3.1 实验目的及要求

(1) 使用贪心算法解决具有多个互相关联条件的问题。

(2) 练习编程实现贪心算法解决普通背包问题。

◇5.3.2 实验内容

试编程使用贪心算法求解普通背包问题。

给定 n 种物品和一个背包。物品 i 的重量是 W_i，其价值为 V_i，背包的容量为 W。应如何选择物品装入背包，使得装入背包中的物品的总价值最大？本题是普通背包问题，简化于 0-1 背包问题。区别是普通背包问题允许物品切割，即可以放入某物体的一部分。

有如下样例数据：假设有三种物品，代号分别为 1，2，3，其重量分别为 20kg，30kg，60kg，对应的价值分别是 40kg，90kg，240kg，背包容量为 90kg。

输入：物品个数、各物品的价值和重量。

输出：物品建议取法（因为可以取部分物品，所以应使用浮点数）。

样例输入：

```
请输入物品个数：
3
请输入各物品的价值 ：
40 90 240
请输入各物品的重量 ：
20 30 60
```

样例输出：

```
物品建议取法：
0.000   1.000   1.000
```

◇5.3.3 实验原理

首先分析题目中需要计算的属性参数和限制条件，共有两个属性参数，一个是重量维度，另一个是价值维度。重量维度上有一个限制条件，即背包的总重量，物品的重量和必须小于 W，满足此限制的同时在价值维度上，期望得到最大值。每个物品的价值不同，重量也不同，但两个属性具有关系，可以求得单位重量下的价值，即通常所讲的“性价比”。

物品 1 每千克价值 40/20＝2。

物品 2 每千克价值 90/30＝3。

物品 3 每千克价值 240/60＝4。

对于期望获取最大值的问题，尝试用贪心算法分析：要使装入的物品总价值最大，应将性价比更高的物品先放入，因为普通背包问题可以放入物品的一部分，所以尽可能多地放入物品直到将背包装满为止。

对于样例数据先装物品 3，把 60kg 物品全部装进背包，还剩 30kg 物品，把 30kg 的物品 2 全部装入，此时包的容量正好装满，且背包中的物品的总价值为 240＋90＝330。

可尝试用反证法验证此方法，假设使总价值最大的选取不包括性价比最高的物品，则有操作如下：去掉其中一部分物品，换成性价比更高的物品，就得到了价值更高的选择，与假设的总价值最大产生矛盾，所以假设不成立。得到结论：总价值最大的选择一定包含性价比最高的物品。

从贪心算法的角度再次验证，每次选择都挑选当前最值得装入背包的物品，即性价比最高的物品，做当前状态下最优的选择。所以编程中需要计算每种物品的性价比，并进行排序。

◇5.3.4 实验步骤

结合上述思路，应用贪心算法解决普通背包问题的步骤如下。

(1) 计算每种物品单位重量的价值 V_i/W_i。

(2) 单位重量价值高的优先选择，即按照性价比从大到小排序。

(3) 根据贪心算法的思想，按照排序从最高性价比的物品开始，将尽可能多的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未达到 W，则按照排序选择下一个次高性价比的物品，并尽可能多地装入背包。依此策略一直进行下去，直到达到背包的重量限制 W。编程中应注意第一个物品也可能无法完全装入背包，需要注意重量维度的限制条件。

根据此编程思路，试着使用贪心算法完成对背包问题的编程。

◇5.3.5 参考代码

参考代码如下。

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSIZE 100                                                    //物品最大数
#define M 90                                                           //背包的容量

void getData(float djz[], float dzw[], int * n);                       //输入
void sort(float tempArray[], int sortResult[], int n);                 //排序算法
void greedy(float dzw[], float x[], int sortResult[], int n);          //贪心算法
void output(float x[], int n);                                         //输出

int main()                                                             //主函数
{
float djz[MAXSIZE], dzw[MAXSIZE], x[MAXSIZE];    //物品价值、重量和性价比

int i = 0, n = 0;
int sortResult[MAXSIZE];                                 //待排序后输出的排序结果
getData(djz, dzw, &n);                                   //输入
for (i = 0; i < n; i++)
{
x[i] = djz[i] / dzw[i];
}
sort(x, sortResult, n);                                  //排序算法
greedy(dzw, x, sortResult, n);                           //贪心算法
output(x, n);                                            //输出
return 0;
}

//输入(参数为各物品的价值、重量以及物品个数)
void getData(float djz[], float dzw[], int * n)
{
int i = 0;
printf("请输入物品个数: \n");
scanf("%d", n);
printf("请输入各物品的价值 :\n");
for (i = 0; i < (*n); i++)
{
scanf("%f", &djz[i]);
}
printf("请输入各物品的重量 :\n");
for (i = 0; i < (*n); i++)
{
```