砝码分盐问题——从数学和计算机的角度分析

作者 ： 余祖波
Blog: http://blog.csdn.net/livelylittlefish, http://abo321.org

Content

0．问题

假设有 280g 盐，有一架天平，有两个砝码，分别是 4g 和 14g。能否在 3 次内将 280g 食盐分为 100g 和 180g 两堆，请详细描述你的解决方法。

这是一些公司(据传是微软、腾讯等)的面试题目，网络上有些答案，但笔者认为这些答案仅仅只是答案而已，没有任何的分析。笔者对该问题进行了一些思考，本文就重点叙述如何从数学和计算机的角度来分析并解决该问题。希望对希望深入思考该类题目的朋友一些启发。

1．一些方法

1.1 去除法

(1) 用 4g 砝码，将 280g 盐分为 142g 和 138g；(得盐：138g，142g)
(2) 用 14g 砝码，从 142g 盐中称出 14g 盐，剩 128g 盐；(得盐：14g，128g，138g)
(3) 用 14g 砝码和 14g 盐，从 128g 盐中称出 28g 盐；(得盐：100g，14g，28g，138g)

或者，
(1) 用 4g 砝码，将 280g 盐分为 142g 和 138g；(得盐：138g，142g)
(2) 用 4g 和 14g 砝码，从 138g 盐中称出 10g 盐，剩 128g 盐；(得盐：10g，128g，142g)
(3) 用 4g、14g 砝码和 10g 盐，从 128g 盐中称出 28g 盐；(得盐：100g，10g，28g，142g)

或者，
(1) 用 14g 砝码称出 14g 盐；(得盐：14g，266g)
(2) 用 14g 砝码和 4g 砝码，将 266g 盐分为 128g 和 138g；(得盐：14g，128g，138g)
(3) 用 14g 砝码和 14g 盐，从 128g 盐中称出 28g 盐；(得盐：100g，14g，28g，138g)

还有其他的去除方法，例如。
(1) 280=140+140
(2) 140-4-14=122 (去 18g 盐)
(3) 122-4-18=100 (18g 盐当作砝码，去 22g 盐)

实质上，这些方法大同小异。

1.2 分解法

(1) 用 4g 砝码，将 280g 盐分为 142g 和 138g；(得盐：138g，142g)
(2) 用 4g 和 14g 砝码，将 142g 盐分为 80g 和 62g；(得盐：62g，80g，138g)
(3) 用 4g 砝码，将 80g 盐分为 42g 和 38g；(得盐：38g，42g，62g，138g)

或者
(1) 将 280 克盐通过天平等分为 140g 和 140g；(得盐：140g，140g)
(2) 将 140 克食盐再等分为 70g 和 70g；(得盐：70g，70g，140g)

(3) 用 4g 和 14g 砝码，将 70g 盐分成 40g 和 30g；(得盐：30g，40g，70g，140g)

或者
(1) 280=138+142
(2) 138=62+76
(3) 62=24+38 (得盐：24g，38g，76g，142g)

本文其他部分将重点讨论从数学和计算机的角度分析分解方法的解及其过程。

1.3 累加法

(1) 用 4g 和 14g 砝码称 18g 盐；(得盐：18g，262g)
(2) 用 4g、14g 砝码和 18g 盐称 36g 盐；(得盐：18g，36g，226g)
(3) 用 14g 砝码和 36g 盐，4g 砝码，称盐 46g；(得盐：18g，36g，46g，180g)

或者
(1) 用 4g 和 14g 砝码称 18g 盐；(得盐：18g，262g)
(2) 用 14g 砝码和 18g 盐称 32g 盐；(得盐：18g，32g，230g)
(3) 用 18g 盐和 32g 盐称 50g 盐；(得盐：18g，32g，50g，180g)

注：该方法为我老婆所想，她的确很聪明。在此赞她一下。

1.4 小结

解题时不能受思维定势干扰。对于天平来说，任何物体都可以作为砝码。这是解题的关键。称量过程中，可以去除，分解，累加。

2．从数学的角度分析

从上面的方法可以看出，对于去除法和累加法，因其去除或者累加情况较多，组合的值也较多，不易确定分析。

但分解应该可以确定，因为分解主要是要利用已有的砝码对盐进行分堆，分堆的个数也容易确定，一定为 4 堆，因为 3 次称量，每次对其中一堆进行再分堆，且每次只能分成 2 堆。且分堆时用到的砝码重量也是确定的(详见后续的分析)，故分解方法的解的个数也一定是确定的。

本节即从数学的角度分析采用***分解方法***的情况下正确的称量过程及正确的解的个数。

2.1 砝码组合状态

砝码的组合有两种情况。

(1) 出现在天平同一侧

14g 砝码	4g 砝码	组合结果
0	0	0
0	1	4g
1	0	14g
1	1	18g

0 表示不出现，1 表示出现。实际上，对于两个砝码均不出现的情况，其动作即为等分盐。

(2) 出现在天平两侧

出现在天平两侧，不论位置，所代表的砝码质量均为 10g。

综上，砝码组合共有 0，4g，10g，14g，18g，共 5 中状态。

(3) 讨论

如果将以上(1)、(2)两种情况综合在一起，也可以，只是需要重新定义。

- 0：砝码不出现
- 1：砝码出现在天平同一侧
- -1：砝码出现在天平两侧(每侧各一个)

14g 砝码	4g 砝码	组合结果
0	0	0
0	1	4g
1	0	14g
1	1	18g
1/-1	-1/1	10g

可参考"砝码称重问题"一文，但要注意其间的区别。

2.2 数学解法

2.2.1 限制规则

(1) 第一次称量

设盐的重量为 z，用重量为 w 的砝码称量，被分为重量为 x 和 y(假设 x<=y)的盐，则 x，y，z，w 满足如下等式。

```
y + x = z
y - x = w
```

(2) 第二次称量

第二次要对重量为 x 或者 y 的盐进行称量分解。共 2 种情况。

若对 x 进行称量，被分为 x1 和 x2(假设 x1<=x2)，则满足如下等式。

```
x2 + x1 = x
x2 - x1 = w
```

若对 y 进行称量，被分为 y1 和 y2(假设 y1<=y2)，则满足如下等式。

```
y1 + y2 = y
y2 - y1 = w
```

(3) 第三次称量

第三次要对重量为 x1 或者 x2 或者 y1 或者 y2 的盐进行称量分解。共 4 中情况。

(3.1) 分解 x1

若对 x1 进行称量，被分为 x11 和 x12(假设 x11<=x12)，则满足如下等式。

```
x12 + x11 = x1
x12 - x11 = w
```

至此，3 次称量后，盐被分为重量为 x11、x12、x2、y 的 4 堆盐。

(3.2) 分解 x2

若对 x2 进行称量，被分为 x21 和 x22(假设 x21<=x22)，则满足如下等式。
x22 + x21 = x2
x22 - x21 = w

至此，3 次称量后，盐被分为重量为 x1、x21、x22、y 的 4 堆盐。

(3.3) 分解 y1

若对 y1 进行称量，被分为 y11 和 y12(假设 y11<=y12)，则满足如下等式。
y12 + y11 = y1
y12 - y11 = w

至此，3 次称量后，盐被分为重量为 x、y11、y12、y2 的 4 堆盐。

(3.4) 分解 y2

若对 y2 进行称量，被分为 y21 和 y22(假设 x21<=x22)，则满足如下等式。
y22 + y21 = y2
y22 - y21 = w

至此，3 次称量后，盐被分为重量为 x、y1、y21、y22 的 4 堆盐。

2.2.2 隐含的限制规则

对于本题，根据 2.2.1 节的讨论，可知

(1) z=280g，砝码 w 可取 0、4g、10g、14g、18g。注意：这些值是砝码组合之后的值。

(2) 所有的变量均为整数。

(3) 因为 w 为偶数，x2-x1=w => x1 和 x2 同为奇数或者同为偶数。
因 x2 + x1 = x，故 x 一定为偶数(不论 x1 和 x2 同为奇数还是同为偶数)。
同样的道理，可知，y、x1、x2、y1、y2 也一定为偶数。

(4) 根据 2.2.1(3)对第三次称量的讨论，例如，(3.1)分解 x1，盐被分为重量为 x11、x12、x2、y 的 4 堆盐，显然，题目要求的 100g 盐和 180g 盐要在这 4 堆盐中组合产生，且已知 x2、y 为偶数，故，x11 和 x12 也一定为偶数。
同样的道理，可知，对于其他对 x2、y1、y2 的分解，x21 和 x22、y11 和 y12、y21 和 y22 也一定为偶数。

2.2.3 规则小结

综上，由 w、z 为偶数，可知，3 次分解出的所有重量
x,y,x1(x11,x12),x2(x21,x22),y1(y11,y12),y2(y21,y22)全都为偶数。——这便是本题隐含的限制规则。

分解可以记为：z(x(x1(x11,x12),x2(x21,x22)),y(y1(y11,y12),y2(y21,y22)))。
其中对 x1,x2,y1,y2 的分解是或关系。因此，该记法明确表示只有 3 次称量。

2.3 称量过程

根据 2.2 节的讨论，称量过程就很好写出来了。以下 T 表示该次分解正确，F 表示错误。10g 砝码表示此次分解用到的是 10g 砝码，其余类同。

(1) 第一次称量

280 = 140 + 140 (0g 砝码, T)

280 = 138 + 142 (4g 砝码, T)
280 = 135 + 145 (10g 砝码，F)
280 = 133 + 147 (14g 砝码，F)
280 = 131 + 149 (18g 砝码，F)

(2) 第二次称量

对 140 分解：
140 = 70 + 70 (0g 砝码, T)
140 = 68 + 72 (4g 砝码, T)
140 = 65 + 75 (10g 砝码, F)
140 = 63 + 77 (14g 砝码, F)
140 = 61 + 79 (18g 砝码, F)

对 138 分解：
138 = 69 + 69 (0g 砝码, F)
138 = 67 + 71 (4g 砝码, F)
138 = 64 + 74 (10g 砝码，T)
138 = 62 + 76 (14g 砝码，T)
138 = 60 + 78 (18g 砝码，T)

对 142 分解：
142 = 71 + 71 (0g 砝码, F)
142 = 69 + 73 (4g 砝码, F)
142 = 66 + 76 (10g 砝码, T)
142 = 64 + 78 (14g 砝码, T)
142 = 62 + 80 (18g 砝码, T)

(3) 第三次称量

过程如下，蓝色表示正确的分解。

第一次称量	第二次称量	第三次称量
280 = 140 + 140 (0g 砝码, T)	140 = 70 + 70 (0g 砝码, T)	分解 70： 70 = 35 + 35 (0g 砝码, F) 70 = 33 + 37 (4g 砝码, F) **70 = 30 + 40 (10g 砝码, T) => 30,40,70,140 => T** 70 = 28 + 42 (14g 砝码, T) => 28,42,70,140 => F 70 = 26 + 44 (18g 砝码, T) => 26,44,70,140 => F
	140 = 68 + 72 (4g 砝码, T)	分解 68： 68 = 34 + 34 (0g 砝码, T) => 34,34,72,140 => F 68 = 32 + 36 (4g 砝码, T) => 32,36,72,140 => F 68 = 29 + 39 (10g 砝码, F) 68 = 27 + 41 (14g 砝码, F) 68 = 25 + 43 (18g 砝码, F) 分解 72： 72 = 36 + 36 (0g 砝码, T) => 68,36,36,140 => F 72 = 34 + 38 (4g 砝码, T) => 68,34,38,140 => F 72 = 31 + 41 (10g 砝码, F) 72 = 29 + 43 (14g 砝码, F) 72 = 27 + 45 (18g 砝码, F)
280 = 138 + 142	138 = 64 + 74	分解 64：

(4g 砝码, T)	(10g 砝码, T)	64 = 32 + 32 (0g 砝码, T) => 32,32,74,142 => F 64 = 30 + 34 (4g 砝码, T) => 30,34,74,142 => F 64 = 27 + 37 (10g 砝码, F) 64 = 25 + 39 (14g 砝码, F) 64 = 23 + 41 (18g 砝码, F) 分解 74: 74 = 37 + 37 (0g 砝码, F) 74 = 35 + 39 (4g 砝码, F) 74 = 32 + 42 (10g 砝码, T) => 64,32,42,142 => F 74 = 30 + 44 (14g 砝码, T) => 64,30,44,142 => F 74 = 28 + 46 (18g 砝码, T) => 64,28,46,142 => F
	138 = 62 + 76 (14g 砝码, T)	分解 62: 62 = 31 + 31 (0g 砝码, F) 62 = 29 + 33 (4g 砝码, F) 62 = 26 + 36 (10g 砝码, T) => 26,36,76,142 => F **62 = 24 + 38 (14g 砝码, T) => 24,38,76,142 => T** 62 = 22 + 40 (18g 砝码, T) => 26,36,76,142 => F 分解 76: **76 = 38 + 38 (0g 砝码, T) => 62,38,38,142 => T** 76 = 36 + 40 (4g 砝码, T) => 62,36,40,142 => F 76 = 33 + 43 (10g 砝码, F) 76 = 31 + 45 (14g 砝码, F) 76 = 29 + 47 (18g 砝码, F)
	138 = 60 + 78 (18g 砝码, T)	分解 60: 60 = 30 + 30 (0g 砝码, T) => 30,30,78,142 => F 60 = 28 + 32 (4g 砝码, T) => 28,32,78,142 => F 60 = 25 + 35 (10g 砝码, F) 60 = 23 + 37 (14g 砝码, F) 60 = 21 + 49 (18g 砝码, F) 分解 78: 78 = 39 + 39 (0g 砝码, F) 78 = 37 + 41 (4g 砝码, F) 78 = 34 + 44 (10g 砝码, T) => 60,34,44,142 => F 78 = 32 + 46 (14g 砝码, T) => 60,32,46,142 => F 78 = 30 + 48 (18g 砝码, T) => 60,30,48,142 => F
	142 = 66 + 76 (10g 砝码, T)	分解 66: 66 = 33 + 33 (0g 砝码, F) 66 = 31 + 35 (4g 砝码, F) 66 = 28 + 38 (10g 砝码, T) => 138,28,38,76 => F 66 = 26 + 40 (14g 砝码, T) => 138,26,40,76 => F **66 = 24 + 42 (18g 砝码, T) => 138,24,42,76 => T** 分解 76: 76 = 38 + 38 (0g 砝码, T) => 138,66,38,38 => F 76 = 36 + 40 (4g 砝码, T) => 138,66,36,40 => F 76 = 33 + 43 (10g 砝码, F) 76 = 31 + 45 (14g 砝码, F) 76 = 29 + 47 (18g 砝码, F)
	142 = 64 + 78	分解 64: 64 = 32 + 32 (0g 砝码, T) => 138,32,32,78 => F

	(14g 砝码, T)	64 = 30 + 34 (4g 砝码, T) => 138,30,34,78 => F 64 = 27 + 37 (10g 砝码, F) 64 = 25 + 39 (14g 砝码, F) 64 = 23 + 41 (18g 砝码, F) 分解 76: 78 = 39 + 39 (0g 砝码, F) 78 = 37 + 41 (4g 砝码, F) 78 = 34 + 44 (10g 砝码, T) => 138,64,34,44 => F 78 = 32 + 46 (14g 砝码, T) => 138,64,32,46 => F 78 = 30 + 48 (18g 砝码, T) => 138,64,30,48 => F
	142 = 62 + 80 (18g 砝码, T)	分解 62: 62 = 31 + 31 (0g 砝码, F) 62 = 29 + 33 (4g 砝码, F) 62 = 26 + 36 (10g 砝码, T) => 138,26,36,80 => F 62 = 24 + 38 (14g 砝码, T) => 138,24,38,80 => F 62 = 22 + 40 (18g 砝码, T) => 138,22,40,80 => F 分解 80: 80 = 40 + 40 (0g 砝码, T) => 138,62,40,40 => F **80 = 38 + 42 (4g 砝码, T) => 138,62,38,42 => T** 80 = 35 + 45 (10g 砝码, F) 80 = 33 + 47 (14g 砝码, F) 80 = 31 + 49 (18g 砝码, F)

2.4 正确的称量过程

综上，正确的称量(分解)过程如下。

No.	第一次称量	第二次称量	第三次称量
1	280 = 140 + 140 (0g 砝码)	140 = 70 + 70 (0g 砝码)	**70 = 30 + 40 (10g 砝码) => 30,40,70,140**
2	280 = 138 + 142 (4g 砝码)	138 = 62 + 76 (14g 砝码)	**62 = 24 + 38 (14g 砝码) => 24,38,76,142**
3	280 = 138 + 142 (4g 砝码)	138 = 62 + 76 (14g 砝码)	**76 = 38 + 38 (0g 砝码) => 62,38,38,142**
4	280 = 138 + 142 (4g 砝码)	142 = 66 + 76 (10g 砝码)	**66 = 24 + 42 (18g 砝码) => 138,24,42,76**
5	280 = 138 + 142 (4g 砝码)	142 = 62 + 80 (18g 砝码)	**80 = 38 + 42 (4g 砝码) => 138,62,38,42**

至此，通过分解的方法，已经得到了全部的解，共 5 个。

2.5 一个疑问

或许细心的读者已经发现一个疑问。

如果第二次称量时分解的是 x(分成 x1 和 x2)，为什么第三次称量不分解 y(分成 y1 和 y2)？
或者如果第二次称量时分解的是 y，为什么第三次称量不分解 x？即称量过程如下。

第一次称量：z = x + y
第二次称量：x = x1 + x2 (或 y = y1 + y2)

第三次称量：y = y1 + y2 (或 x = x1 + x2)
以上假设 x1 <= x2, y1 <= y2。

这是可以证明的，下面给出证明过程。

证明：假设按如上过程称量。

设第二、三次称量使用的砝码重量分别为 w1、w2，根据上述分析，可知，w1,w2∈{0, 4, 10, 14, 18}，因此，
0 ≤ w1+w2 ≤ 36，且-18 ≤ w1-w2 ≤18 或者 -18 ≤w1-w2 ≤18。另有，

x2 -x1 = w1
y2 -y1 = w2

设最后分成的两堆盐为 z1=100、z2=180。

(1) 若
x1 + y1 = z1
x2 + y2 = z2
可得，(x2+x1)-(y2+y1)=w1+w2=80，显然矛盾。

(2) 若
x1 + y2 = z1
x2 + y1 = z2
可得，(x2-x1)+(y1-y2)=w1-w2=80，显然矛盾。

(3) 若
x2 + y1 = z1
x1 + y2 = z2
可得，(x1-x2)+(y2-y1)=w2-w1=80，显然矛盾。

(4) 若
x2 + y2 = z1
x1 + y1 = z2
可得，(x2-x1)+(y2-y1)=w1+w2=-80，显然矛盾。

综上，假设不成立，即不能按如上过程进行称量。

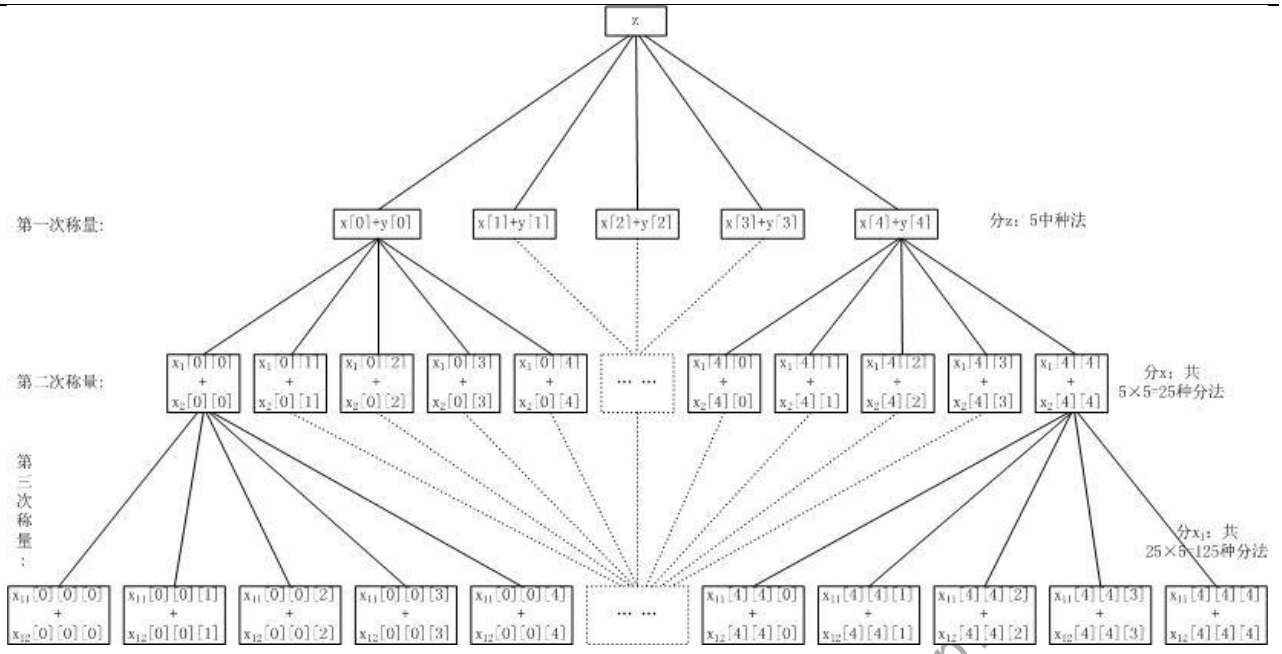
至此，也从侧面证明 2.3 节的称量分解过程是完全的。

3．能否编程计算？

对于第 2 节的分析结果，能否编程计算？该如何写程序？本节从纯数学的角度给出上述分解方法的程序。

3.1 基本思想

该方法的基本思想如下图。



该图只画出了其中的分解 x 并在第三次称量时分解 x1 的分解图，图中每个节点中的数组及程序中用来保存该次称量分解的结果。省略分解 x2、分解 y 并在第三次称量时分解 y1 或者 y2 的节点。

完整的代码请参考附录 1，下面介绍相关数据结构和三次称量过程。

3.2 数据结构描述

图中的一维、二维、三维数组定义如下。

```
00001: / **
00002: * the proble of dividing salt using weight.
00003: * a heap of salt with 280g, divide it into two small heaps of salt with 100g and
180g
00004: * respectively, in 3 steps. the weights are 4g and 14g. there is a balance of
course.
00005: *
00006: * this is a pure mathematical method.
00007: */
00008: #include <stdio.h>
00009:
00010: #define Max_Num 5
00011:
00012: int total = 280;                  //the total weight of the heap of salt
00013: int heap1 = 100, heap2 = 180; //the target weight of the two samll heaps of salt
00014: int ws[]={0, 4, 10, 14, 18};  //all cases of the weights combination
00015:
00016: /* the first division result */
00017: int x[Max_Num] = {0}, y[Max_Num] = {0};
00018:
00019: /* the second division result */
00020: int x1[Max_Num][Max_Num] = {0}, x2[Max_Num][Max_Num] = {0};
00021: int y1[Max_Num][Max_Num] = {0}, y2[Max_Num][Max_Num] = {0};
00022:
00023: /* the third division result */
00024: int x11[Max_Num][Max_Num][Max_Num] = {0}, x12[Max_Num][Max_Num][Max_Num] = {0};
00025: int x21[Max_Num][Max_Num][Max_Num] = {0}, x22[Max_Num][Max_Num][Max_Num] = {0};
00026: int y11[Max_Num][Max_Num][Max_Num] = {0}, y12[Max_Num][Max_Num][Max_Num] = {0};
00027: int y21[Max_Num][Max_Num][Max_Num] = {0}, y22[Max_Num][Max_Num][Max_Num] = {0};
```

注 1：编译该代码时会出现如下警告，是因为在 Linux 平台，有内建的函数名为 y1，命名冲突，故出现该警告，可通过 "man y1"命令查看该函数的说明。如果想消除该警告，请将代码中 y1 数组重新命名。

```
weight3.c:21: warning: built-in function ‘y1’ declared as non-function
```

注 2：gcc 编译器对上述的 x1/x2/x11/x12/x21/x22/y11/y12/y21/y22 数组初始化会出现如下警告。

```
# make
gcc -g -Wall -Wextra -c weight1.c
weight1.c:20: warning: missing braces around initializer
weight1.c:20: warning: (near initialization for ‘x1[0]’)
```

```
weight1.c:20: warning: missing braces around initializer
weight1.c:20: warning: (near initialization for ‘x2[0]’)
weight1.c:21: warning: built-in function ‘y1’ declared as non-function
weight1.c:21: warning: missing braces around initializer
weight1.c:21: warning: (near initialization for ‘y1[0]’)
weight1.c:21: warning: missing braces around initializer
weight1.c:21: warning: (near initialization for ‘y2[0]’)
weight1.c:24: warning: missing braces around initializer
weight1.c:24: warning: (near initialization for ‘x11[0]’)
weight1.c:24: warning: missing braces around initializer
weight1.c:24: warning: (near initialization for ‘x12[0]’)
weight1.c:25: warning: missing braces around initializer
weight1.c:25: warning: (near initialization for ‘x21[0]’)
weight1.c:25: warning: missing braces around initializer
weight1.c:25: warning: (near initialization for ‘x22[0]’)
weight1.c:26: warning: missing braces around initializer
weight1.c:26: warning: (near initialization for ‘y11[0]’)
weight1.c:26: warning: missing braces around initializer
weight1.c:26: warning: (near initialization for ‘y12[0]’)
weight1.c:27: warning: missing braces around initializer
weight1.c:27: warning: (near initialization for ‘y21[0]’)
weight1.c:27: warning: missing braces around initializer
weight1.c:27: warning: (near initialization for ‘y22[0]’)
...
```

若要消除这些警告，可将其中的数组初始化为 0 的元素加上花括号。一维数组加 1 个花括号，二维数组加 2 个花括号，三维数组加 3 个花括号，即初始化一个具体的元素，其他的元素会被自动初始化为 0.如下。

```
00016: /* the first division result */
00017: int x[Max_Num] = {0}, y[Max_Num] = {0};
00018:
00019: /* the second division result */
00020: int x1[Max_Num][Max_Num] = {{0}}, x2[Max_Num][Max_Num] = {{0}};
00021: int y1[Max_Num][Max_Num] = {{0}}, y2[Max_Num][Max_Num] = {{0}};
00022:
00023: /* the third division result */
00024: int x11[Max_Num][Max_Num][Max_Num] = {{{0}}}, x12[Max_Num][Max_Num][Max_Num] =
{{{0}}};
00025: int x21[Max_Num][Max_Num][Max_Num] = {{{0}}}, x22[Max_Num][Max_Num][Max_Num] =
{{{0}}};
00026: int y11[Max_Num][Max_Num][Max_Num] = {{{0}}}, y12[Max_Num][Max_Num][Max_Num] =
{{{0}}};
00027: int y21[Max_Num][Max_Num][Max_Num] = {{{0}}}, y22[Max_Num][Max_Num][Max_Num] =
{{{0}}};
```

3.3 第一次分解过程

第一次分解即将 z 分解为 z=x+y，过程如下，很简单，不解释。

```
00030: /** the first division, according to z=x+y, x <= y */
00031: void weight1()
00032: {
00033:     int z = total;
00034:     int i = 0, j = 0, k = 0, w = 0;
00035:
00036:     for (k = 0; k < Max_Num; k++)
00037:     {
00038:         w = ws[k];
00039:         x[k] = (z - w)/2;
00040:         y[k] = (z + w)/2;
00041:         if (x[k] % 2!= 0) //no need to judge y[k]
00042:             x[k] = y[k] = 0;
00043:     }
00044: }
```

3.4 第二次称量过程

第二次称量过程如下。注意其中的限制规则，简化了后续的搜索过程，也避免了后续的重复搜索。代码中的注释已经很清楚，此处不再解释。

```
00046: /** the second division, according to x=x1+x2, y=y1+y2, x1<=x2, y1<=y2
00047:     for x[i], x[i] = x1[i][k] + x2[i][k]
00048:     for y[i], y[i] = y1[i][k] + y2[i][k]
00049:     do them in the same loop to reduce the occurrence time of for- loop
```

```
*/
void weight2()
{
    int i = 0, j = 0, k = 0, w = 0;

    for (i = 0; i < Max_Num; i++)
    {
        if (x[i] == 0) //no need to judge y[i]
            continue;

        for (k = 0; k < Max_Num; k++)
        {
            w = ws[k];

            x1[i][k] = (x[i] - w)/2;
            x2[i][k] = (x[i] + w)/2;
            if (x1[i][k] % 2!= 0) //no need to judge x2[i][k]
                x1[i][k] = x2[i][k] = 0;

            if (x[i] == y[i]) //to avoid repeatance
                continue;

            y1[i][k] = (y[i] - w)/ 2;
            y2[i][k] = (y[i] + w)/2;
            if (y1[i][k] % 2!= 0) //no need to judge y2[i][k]
                y1[i][k] = y2[i][k] = 0;
        }
    }
}
```

3.5 第三次称量过程

第三次称量过程如下。注意其中的限制规则，可以简化后续的搜索过程。同上，此处不再解释。

```

/** the third division, according to
    x1=x11+x12, y1=y11+y12, x11<=x12, y11<=y12
    x2=x21+x22, y2=y21+y22, x21<=x22, y21<=y22
    for x1[i][j], x1[i][j] = x11[i][j][k] + x12[i][j][k]
    for x2[i][j], x2[i][j] = x21[i][j][k] + x22[i][j][k]
    for y1[i][j], y1[i][j] = y11[i][j][k] + y12[i][j][k]
    for y2[i][j], y2[i][j] = y21[i][j][k] + y22[i][j][k]
    do them in the same loop to reduce the occurrence time of for- loop
*/
void weight3()
{
    int i = 0, j = 0, k = 0, w = 0;

    for (i = 0; i < Max_Num; i++)
    {
        if (x[i] == 0) //no need to judge y[i]
            continue;

        for (j = 0; j < Max_Num; j++)
        {
            for (k = 0; k < Max_Num; k++)
            {
                w = ws[k];

                if (x1[i][j] != 0) //divide x1[i][j]
                {
                    x11[i][j][k] = (x1[i][j] - w)/2;
                    x12[i][j][k] = (x1[i][j] + w)/2;
                    if (x11[i][j][k]%2!= 0) //x11[i][j][k] and x12[i][j][k] must be even
                        x11[i][j][k] = x12[i][j][k] = 0;
                }

                if (x2[i][j] != 0 && x1[i][j] != x2[i][j]) //divide x2[i][j], and to avoid repeatance
                {
                    x21[i][j][k] = (x2[i][j] - w)/2;
                    x22[i][j][k] = (x2[i][j] + w)/2;
                    if (x21[i][j][k]%2!= 0) //x21[i][j][k] and x22[i][j][k] must be even
                        x21[i][j][k] = x22[i][j][k] = 0;
```

```
00118:              }
00119:
00120:              if (y1[i][j] != 0) //divide y1[i][j]
00121:              {
00122:                  y11[i][j][k] = (y1[i][j] - w)/2;
00123:                  y12[i][j][k] = (y1[i][j] + w)/2;
00124:                  if (y11[i][j][k]%2!= 0) //y11[i][j][k] and y12[i][j][k] must be
even
00125:                      y11[i][j][k] = y12[i][j][k] = 0;
00126:              }
00127:
00128:              if (y2[i][j] != 0 && y1[i][j] != y2[i][j]) //divide y2[i][j], and to
avoid repeatance
00129:              {
00130:                  y21[i][j][k] = (y2[i][j] - w)/2;
00131:                  y22[i][j][k] = (y2[i][j] + w)/2;
00132:                  if (y21[i][j][k]%2!= 0) //y21[i][j][k] and y22[i][j][k] must be
even
00133:                      y21[i][j][k] = y22[i][j][k] = 0;
00134:              }
00135:          }
00136:      }
00137:  }
00138: }
```

3.6 如何输出？

为什么要设计这样的数据结构，主要的目的是为了保存称量过程中的分解结果，但保存分解结果的目的也是为了程序输出，能够将整个分解步骤都输出。代码请参考附录 1。

3.7 输出结果

结果如下，程序可以同时输出所有的分解结果和正确的分解结果，此处只列出正确的结果，完整打印结果请参考附录 2。

```
--------------------------------------
the results of all correct divisions:
--------------------------------------
280 = 140 + 140
140 = 70 + 70
70 = 30 + 40

280 = 138 + 142
138 = 62 + 76
62 = 24 + 38

280 = 138 + 142
138 = 62 + 76
76 = 38 + 38

280 = 138 + 142
142 = 66 + 76
66 = 24 + 42

280 = 138 + 142
142 = 62 + 80
80 = 38 + 42
```

可以看出，这个程序的输出结果和第 2 节从数学角度分析得到的结果完全相同。

3.8 讨论

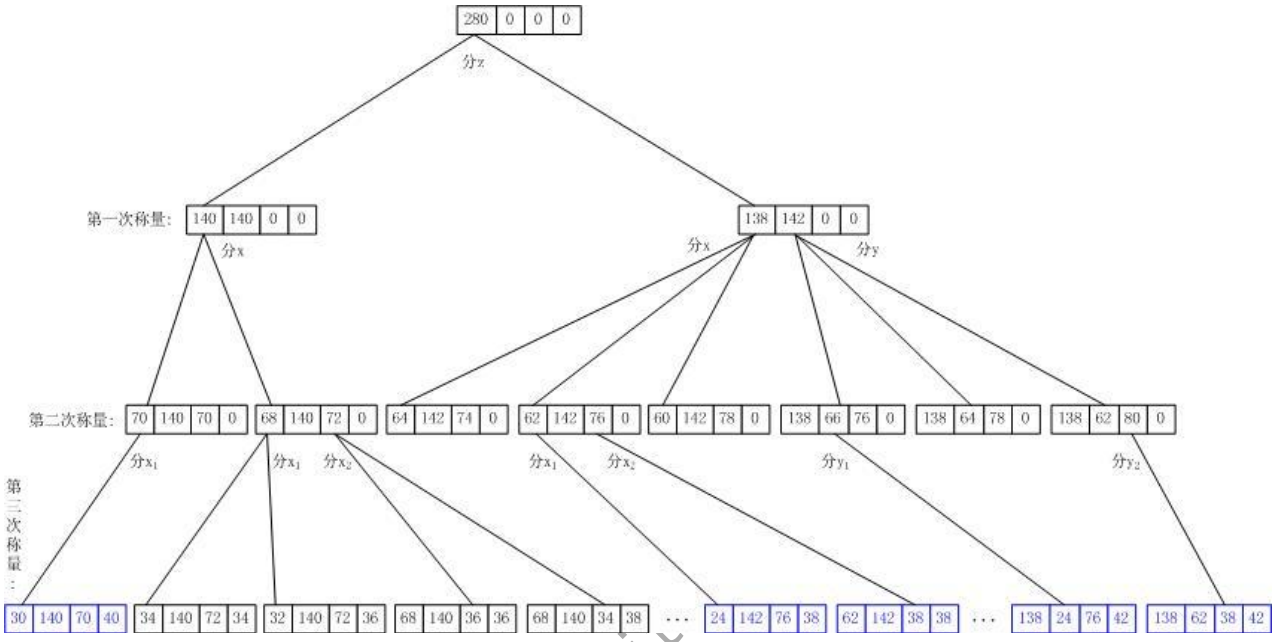
至此，一个纯数学的计算程序介绍已经结束，该方法使用若干个一维、二维、三维数组保存分解结果。该方法数组定义较多，如果加上别的条件，例如再多一次称量，很难扩展。下面介绍另一种数据结构比较简单的方法，从本节的方法改进而来。

4. 一个改进的方法

修改数据结构，不用第 3 节的各种数组，而用树形结构来保存称量过程的分解结果。

4.1 基本思想

该方法的基本思想如下图。



图中每个节点对应分解过程的每一个称量结果，根(第 0 层)表示起始状态，第 1 层表示第 1 次称量结果，依此类推。从根到叶子节点的路径就代表了一次完整的称量过程(该称量过程有分解结果，但结果不一定满足目标)。满足目标的叶子(上图中显示为蓝色)即为正确的称量结果。

比较该图与第 3 节的图，很显然，该方法数据结构统一、简单，且节省大量空间。

4.2 数据结构描述

图中节点结构及根节点定义如下。

```
00001: / **
00002: * the proble of dividing salt using weight.
00003: * a heap of salt with 280g, divide it into two small heaps of salt with 100g and
180g
00004: * respectively, in 3 steps. the weights are 4g and 14g. there is a balance of
course.
00005: *
00006: * this is an improved search method based on the pure mathematical method.
00007: */
00008: #include <stdio.h>
00009: #include <stdlib.h>
00010: #include <string.h>
00011:
00012: #define Max_Num 5
00013: #define To_Be_Devided 2
00014:
00015: #define total 280 //the total weight of the heap of salt
00016: int ws[]={0, 4, 10, 14, 18}; //all cases of the weights combination
00017: int heap1 = 100, heap2 = 180; //the target weight of the two samll heaps of salt
00018:
00019: struct Division_Node
00020: {
00021:     int parent_be_divided_; //array to_be_divided_[] divided from its parent with
this index
00022:     int step_;              //the node step in the division process
00023:     int heap_[Max_Num - 1]; //all divisions of its parent position for all
```

```
weights
00024:     int to_be_divided_[2];  //salt heap index to be divided in next step
00025:     struct Division_Node *next_[2* Max_Num]; //all pointers to all child
divisions
00026:     int child_;             //the numbe of children (not NULL in array next_)
00027: };
00028:
00029: //the root, for step 1, heap_[0] will be divided
00030: static struct Division_Node root = {0, 0, {total}, {0}, {NULL}, 0};
```

当前节点保存其父节点被分解后的结果，heap_[]数组中，下标为 to_be_divided_[0]和 to_be_divided_[1]的元素保存了新生成的两个值，parent_be_divided_表示这两个值是由其父节点的 heap_[]数组中第 parent_be_divided_个元素被分解的结果。

例如，父亲节点 parent，儿子节点为 child，其关系为：
parent.heap_[child.parent_be_divided_]被分解为两个值，分别为 child.heap_[child.to_be_divided_[0]]和 child.heap_[child.to_be_divided_[1]]，且有 parent.next_[k]=&child，表示 child 是 parent 的第 k 个儿子。对于 child 的下一次分解，就依次分解这两个新生成的值。

4.3 分解过程描述

按广度优先的顺序，求得如上图的每一次称量的节点，即可完成所有有结果的称量。当然要避免重复，并使用限制规则来减小搜索空间。

从根节点 root.heap_[4]={280}开始分解，初始时 root.to_be_divided_[2]={0}，即开始时只有 root.heap_[0]=280 需要分解，以分解 x 及 x1 为例说明该分解过程。
(1) 第一次称量：step=1，z=x+y，x 保存在 heap_[0]，y 保存在 heap_[step]，to_be_divided_[2]={0,step}。
(2) 第二次称量：step=2，x=x1+x2，x1 保存在 heap_[0]，x2 保存在 heap_[step]，to_be_divided_[2]={0,step}。
(3) 第三次称量：step=3，x1=x11+x12，x11 保存在 heap_[0]，x12 保存在 heap_[step]，to_be_divided_[2]={0,step}。

该过程从上图也能看到。此处仅给出第三次称量过程的代码，如下，其他省略，具体请参考附录 3。

```
00119: /** the third division, according to
00120:     x1=x11+x12, y1=y11+y12, x11<=x12, y11<=y12
00121:     x2=x21+x22, y2=y21+y22, x21<=x22, y21<=y22
00122:     but, x1 or x2 or y1 or y2 is saved in heap_ of each node2 with index node2-
 >to_be_divided_[0]
00123:     and node2->to_be_divided_[1]. so, unify them to be as x1.
00124: */
00125: void weight3()
00126: {
00127:     int div = 0, i = 0, j = 0, k = 0, w = 0;
00128:     struct Division_Node *cur = &root;
00129:
00130:     int step = 3;
00131:     for (i = 0; i < cur->child_; i++) //for each node1 in step1
00132:     {
00133:         struct Division_Node *node1 = cur->next_[i];
00134:         for (j = 0; j < node1->child_; j++) //for each node2 of node1 in step2
00135:         {
00136:             struct Division_Node *node2 = node1->next_[j];
00137:
00138:             //to avoid repeatance
00139:             int to_be_divided = To_Be_Devided;
00140:             if (node2->heap_[node2->to_be_divided_[0]] == node2->heap_[node2-
>to_be_divided_[1]])
00141:                 to_be_divided--;
00142:
00143:             int child = 0;
                 //for step 3, will divide x1=heap_[0], x2=heap_[2] of node2
00144:             for (div = 0; div < to_be_divided; div++)
00145:             {
00146:                 int curpos = node2->to_be_divided_[div];
```

```
00147:              int x1 = node2->heap_[curpos];//the current heap to be divided,
x1,or x2,or y1,or y2
00148:
00149:              for (k = 0; k < Max_Num; k++)
00150:              {
00151:                  w = ws[k];
00152:                  int x11 = (x1 - w)/2; //divide x or y, use x in order to be in a
uniform
00153:                  int x12 = (x1 + w)/2;
00154:                  if (x11 % 2!= 0) //no need to judge x12
00155:                      continue;
00156:
00157:                  //new a node in step 3
00158:                  struct Division_Node *node3 = (struct
Division_Node*)malloc(sizeof(
                            struct Division_Node));
00159:                  memset(node3, 0, sizeof(struct Division_Node));
00160:                  memcpy(node3->heap_, node2->heap_, (Max_Num-
1)*sizeof(int));//copy from its parent
00161:                  node3->parent_be_divided_ = curpos;
00162:                  node3->step_ = step;
00163:                  node3->heap_[curpos] = x11;
00164:                  node3->heap_[step] = x12;
00165:                  node3->to_be_divided_[0] = curpos; //in fact, this array in step3
needed for dump
00166:                  node3->to_be_divided_[1] = step;
00167:                  node2->next_[child++] = node3; //link current node2 and node3
00168:              }
00169:          }
00170:
00171:          node2->child_ = child;
00172:      }
00173:  }
00174: }
```

4.4 如何输出？

广度遍历该树，即可输出所有结果。如果加入判断，即可输出正确分解结果。此处列出输出正确结果的代码。具体过程不再解释，可参考代码注释进行理解。

```
00204: void dump_correct_results(struct Division_Node* node)
00205: {
00206:     int i = 0, j = 0, k = 0;
00207:     struct Division_Node *cur = node;
00208:
00209:     for (i = 0; i < cur->child_; i++) //for each node1 in step1
00210:     {
00211:         struct Division_Node *node1 = cur->next_[i];
00212:         for (j = 0; j < node1->child_; j++) //for each node2 of node1 in step2
00213:         {
00214:             struct Division_Node *node2 = node1->next_[j];
00215:             for (k = 0; k < node2->child_; k++) //for each node3 of node2 in step 3
00216:             {
00217:                 struct Division_Node *node3 = node2->next_[k];
00218:
00219:                 /** unify them into the following equations
00220:                     z = x + y, x <= y
00221:                     x = x1 + x2, x1 <= x2
00222:                     x1 = x11 + x12, x11 <= x12
00223:                 */
00224:                 int x11 = node3->heap_[node3->to_be_divided_[0]];
00225:                 int x12 = node3->heap_[node3->to_be_divided_[1]];
00226:                 int x2index = 0;
00227:                 if (node3->parent_be_divided_ == node2->to_be_divided_[0])
00228:                     x2index = node2->to_be_divided_[1];
00229:                 else
00230:                     x2index = node2->to_be_divided_[0];
00231:                 int x2 = node2->heap_[x2index];
00232:
00233:                 if (x11 + x2 == heap1 || x12 + x2 == heap1)
00234:                 {
00235:                     printf("%d = %d + %d\n", total, node1->heap_[node1-
>to_be_divided_[0]],
00236:                         node1->heap_[node1->to_be_divided_[1]]);
00237:                     printf("%d = %d + %d\n", node1->heap_[node2->parent_be_divided_],
00238:                         node2->heap_[node2->to_be_divided_[0]],
```

```
00239:                     node2->heap_[node2->to_be_divided_[1]]);
00240:                 printf("%d = %d + %d\n\n", node2->heap_[node3-
>parent_be_divided_], x11, x12);
00241:             }
00242:         }
00243:     }
00244: }
00245: }
```

4.5 输出结果

该方法的输出结果与第 3 节的输出结果完全相同。此处省略，有兴趣的读者可以自行验证。具体的输出结果请参考附录 2。

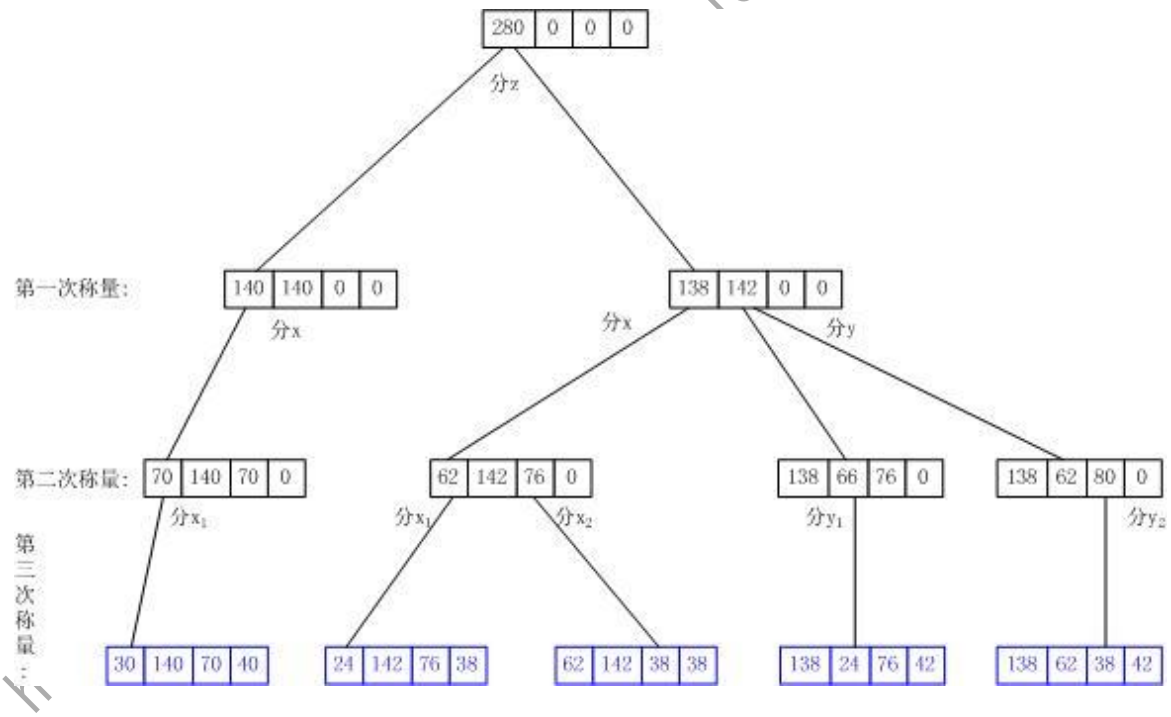
4.6 讨论

比起第 3 节数学的方法，该方法不需要那么多的数组定义，定义数组的方法(第 3 节)如果加上别的条件，例如再多一次称量，很难扩展。很显然，该方法数据结构统一、简单，且节省大量空间。但该方法也有缺点，例如，最终的正确解只有 5 个，但该方法同第 3 节的数学方法一样，计算了所有的有结果的解，共 37 个，也比较浪费。

那么，有没有一种方法，直接在第 2 次称量和第 3 次称量的时候直接判断这个分解是否正确或者直接计算这个正确的解呢？——有。下面介绍这种方法。

5. 再改进的方法

从上图中，删除那些有分解结果但结果并非所求的节点，包括中间节点和叶子节点，如下图所示。



那么，能否通过编程直接计算求得上图所有正确的解呢？
——一定可以。关键就看怎么编程实现目标了。

5.1 基本思想

在第 4 节程序的基础上，继续修改代码，删除 weight3()函数，直接在 weight2()中对第 2 次称量并对第 2 次称量的结果进行再分解(即第 3 次称量)，并对第 3 次分解结果进行判断，只有当该称量过程满足目标时，才建立节点，包括第 2 层和第 3 层的节点。这样，没有分解结果的节点(包括第 2 层和第 3 层的节点)就不

会被建立，如上图，叶子节点全部是正确的分解结果。不仅让问题更直观，且节省了空间，虽然空间并不那么重要。

5.2 第 2 次称量过程

如上所述，第 2 次称量过程中要对第 2 次称量分解的结果进行再分解(第 3 次称量)，并对第 3 次分解结果进行判断，这就是该方法的重点实现过程，如下。第 1 次称量过程的代码及具体的代码请参考附录 4。

```
00095: / ** the second division, according to x=x1+x2, y=y1+y2, x1<=x2, y1<=y2
00096:     but, x and y are saved in node1- >to_be_divided_[0] and node1-
 >to_be_divided_[1].
00097:     so, unify them to be as x.
00098: */
00099: void weight2()
00100: {
00101:     int div = 0, i = 0, k1 = 0, w = 0;
00102:     struct Division_Node *cur = &root;
00103:
00104:     int step = 2;
00105:     for (i = 0; i < cur->child_; i++) //for each node1 in step1
00106:     {
00107:         struct Division_Node *node1 = cur->next_[i]; //step 2 will use all nodes
created in step 1
00108:
00109:         //to avoid repeatance
00110:         int to_be_divided = To_Be_Devided;
00111:         if (node1->heap_[node1->to_be_divided_[0]] == node1- >heap_[node1-
>to_be_divided_[1]])
00112:             to_be_divided-- ;
00113:
00114:         int child1 = 0;
00115:         for (div = 0; div < to_be_divided; div++)//step2, will divide
x=heap_[0],y=heap_[1] of node1
00116:         {
00117:             int curpos1 = node1->to_be_divided_[div];
00118:             int x = node1->heap_[curpos1]; //the current heap to be divided
00119:
00120:             for (k1 = 0; k1 < Max_Num; k1++) //divide x=x1+x2, use x in order to be
in a uniform
00121:             {
00122:                 w = ws[k1];
00123:                 int x1 = (x - w)/ 2;
00124:                 int x2 = (x + w)/ 2;
00125:                 if (x1 % 2 != 0) //no need to judge x2
00126:                     continue;
00127:
00128:                 struct Division_Node *node2 = NULL;
00129:                 int child2 = 0;
00130:
00131:                 /** a correct solution, and only in this case, create node2 and
node3 */
00132:                 child2 = weightx1(curpos1, step, x1, x2, child1, child2, node1,
&node2);//divide x1
00133:                 if (x2 != x1) //to avoid repeatance
00134:                     child2 = weightx2(curpos1, step, x1, x2, child1, child2, node1,
&node2);//divide x2
00135:
00136:                 if (node2)
00137:                 {
00138:                     node2->child_ = child2;
00139:                     child1++;
00140:                 }
00141:             }
00142:         }
00143:         node1->child_ = child1;
00144:     }
00145: }
```

5.3 第 3 次称量过程

该方法中，第 3 次称量过程在 weightx1()和 weightx2()中完成，且是第 2 次过程中嵌套进行，即通过 weight2()调用完成。由于 weightx1()和 weightx2()思想相同，此处以 weightx2()为例叙述，其他的代码可参考附录 4。

```
00173: int weightx2(int curpos1, int step, int x1, int x2, int child1, int child2,
00174:     struct Division_Node* node1, struct Division_Node **node2)
00175: {
00176:     int k2 = 0, w = 0;
00177:     int curpos2 = step;
00178:
00179:     for (k2 = 0; k2 < Max_Num; k2++)
00180:     {
00181:         w = ws[k2];
00182:         int x21 = (x2 - w)/ 2; //divide x or y, use x in order to be in a uniform
00183:         int x22 = (x2 + w)/ 2;
00184:         if (x21 % 2 != 0)       //no need to judge x12
00185:             continue;
00186:
00187:         if (x21 + x1 == heap1 || x22 + x1 == heap1)
00188:         {
00189:             *node2 = create_node(node1, *node2, curpos1, curpos2, step, x1, x2,
x21, x22,
00190:                 child1, child2);
00191:             child2++;
00192:             break;
00193:         }
00194:     }
00195:
00196:     return child2;
00197: }
```

该分解过程如下。

(1) 第 1 次称量：z=x+y，x<=y

(2) 第 2 次称量：x=x1+x2，x1<=x2

(3) 第 3 次称量：x2=x21+x22，x21<=x22

最后的分解结果为(x1，x21，x22，y)，故需要判断 x1+x21 或 x1+x22 是否为 heap1=100，如代码所示。实际上，对第 1 次称量出的 y 的分解在 weight2()第 115 行的 for 循环中完成，即统一为以上过程。

5.4 如何创建节点？

如上所述，只有在满足目标时，才建立节点，包括第 2 层和第 3 层的节点，即代码中的 node2 和 node3。如下。

```
00199: struct Division_Node *create_node(struct Division_Node *node1, struct
Division_Node *node2,
00200:     int curpos1, int curpos2, int step, int x1, int x2, int x11, int x12, int
child1, int child2)
00201: {
00202:     if (node2 == NULL)  //此处放置重复创建 node2(当 node2 有多个儿子节点时)
00203:     {
00204:         //new a node in step 2
00205:         node2 = (struct Division_Node*)malloc(sizeof(struct Division_Node));
00206:         memset(node2, 0, sizeof(struct Division_Node));
00207:         memcpy(node2->heap_, node1->heap_, (Max_Num - 1) * sizeof(int)); //copy
from its parent
00208:         node2->parent_be_divided_ = curpos1;
00209:         node2->step_ = step;
00210:         node2->heap_[curpos1] = x1;
00211:         node2->heap_[step] = x2;
00212:         node2->to_be_divided_[0] = curpos1;
00213:         node2->to_be_divided_[1] = step;
00214:         node1->next_[child1] = node2; //link current node1 and node2
00215:     }
00216:
00217:     //new a node in step 3
00218:     struct Division_Node *node3 = (struct Division_Node*)malloc(sizeof(struct
Division_Node));
00219:     memset(node3, 0, sizeof(struct Division_Node));
00220:     memcpy(node3->heap_, node2- >heap_, (Max_Num - 1) * sizeof(int)); //copy from
its parent
00221:     node3->parent_be_divided_ = curpos2;
00222:     node3->step_ = step + 1;
00223:     node3->heap_[curpos2] = x11;
```

```
00224:     node3->heap_[step + 1] = x12;
00225:     node3->to_be_divided_[0] = curpos2; //in fact, this array in step3 needed for
dump
00226:     node3->to_be_divided_[1] = step + 1;
00227:     node2->next_[child2] = node3; //link current node2 and node3
00228:
00229:     return node2;
00230: } ? end create_node
```

5.5 输出结果

```
-------------------------------------
the results of all correct divisions:
-------------------------------------
280 = 140 + 140
140 = 70 + 70
70 = 30 + 40

280 = 138 + 142
138 = 62 + 76
62 = 24 + 38

280 = 138 + 142
138 = 62 + 76
76 = 38 + 38

280 = 138 + 142
142 = 66 + 76
66 = 24 + 42

280 = 138 + 142
142 = 62 + 80
80 = 38 + 42
```

这就是所有满足目标的正确称量过程，即上图中的所有叶子节点的值。该方法的 3 中写法可参考附录 4，其输出结果可参考附录 5。

5.6 讨论

该方法如其基本思想，对第 3 次分解结果进行判断，只有当该称量过程满足目标时，才建立节点，包括第 2 层和第 3 层的节点，从而叶子节点全部是正确的分解结果，让问题更直观，且节省了大量空间，虽然空间并不那么重要。

6. 能否直接计算求出所有正确解？

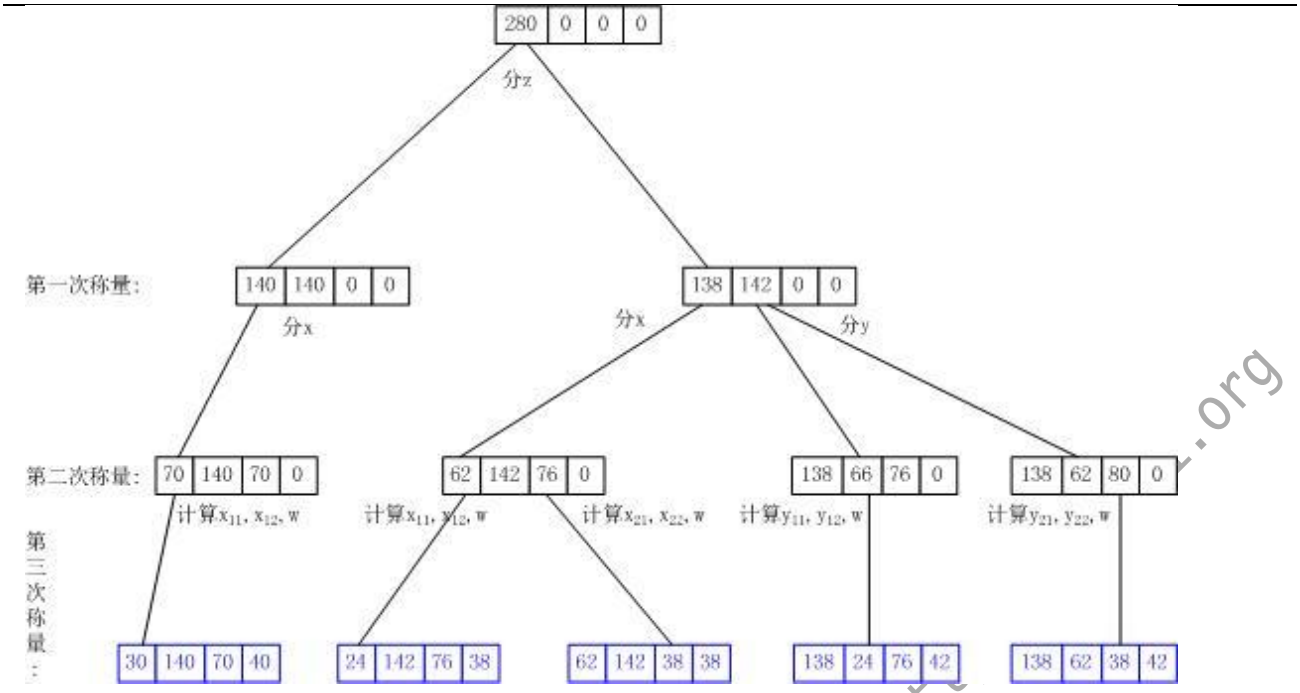
第 3、4 节的讨论，均是从这个角度出发：即找到所有有结果的分解并在所有结果中判断哪些分解过程是满足目标的分解。
而第 5 节的讨论，从对第 3 次称量结果直接进行判断其是否满足目标这个角度出发。

能否再转换一下思路？——能否直接计算求出所有正确的解？

The answer is **yes**!

6.1 基本思想

第 2 次分解(称量)后，直接将满足目标的第 3 次称量解求出，判断这个计算出来的第 3 次称量所用的砝码是否在砝码组合内。如果在，表明该次称量过程为所求的正确解；否则，放弃，继续判断。如下图。



——听起来，这个方法要简单地多。

6.2 第 3 次称量过程

根据该基本思想，改写第 5 节的程序，第 1、2 次称量过程与第 5 节基本相同，不再列出，此处描述第 3 次称量过程。

设 3 次称量过程统一为以下过程。
(1) 第 1 次称量：z=x+y, x<=y
(2) 第 2 次称量：x=x1+x2, x1<=x2
(3) 第 3 次称量：对 x1(x2)直接计算满足目标的 x11 和 x12(x21 和 x22)，下面重点描述。

若对 x1 称量，假设被分解为 x1=x11+x12(这里 x11 和 x12 的大小不定)，则最后的分解结果为(x11,x12,x2,y)，则直接计算出满足目标的 x11=heap1-x2，根据 x11 计算 x12=x1-x11，由此得出 w=x11-x12 或 x12-x11，然后在 ws 数组中查找 w 是否是已知的砝码，若是，则该分解即为所求的正确分解；否则，继续。

代码如下。

```
00136: /** the 3rd weight, in this function, divide x1 and x2
00137:    unify them into the following equations
00138:    z = x + y, x <= y
00139:    x = x1 + x2, x1 <= x2
00140:    x1 = x11 + x12, x11 <= x12
00141:    that is, the salts divided: x11, x12, x2, y, then x11+x2=heap1 or x12+x2=heap
00142:    the basic thoughts: get x11(or x12)=heap1- x2, then, get x12(or x11)=x1-
x11(or x12)
00143:    then, get w=x12- x11, find w in ws, if found, it is a correct division;
otherwise, not a
00144:    correct division, do next.
00145: */
00146: int weight3(int x1, int x2, int curpos1, int child1, int step, struct
Division_Node* node1)
00147: {
00148:    int k = 0, w = 0;
00149:    int curpos2 = curpos1;
00150:    int child2 = 0;
00151:    struct Division_Node *node2 = NULL;
00152:
00153:    //divide x1
00154:    {
00155:       int x11 = heap1 - x2;
00156:       int x12 = x1 - x11;
```

```
00157:        if (x11 <= x12)
00158:           w = x12 - x11;
00159:        else
00160:           w = x11 - x12;
00161:
00162:        for (k = 0; k < Max_Num; k++)
00163:        {
00164:           if (ws[k] == w)
00165:           {
00166:              node2 = create_node(node1, node2, curpos1, curpos2, step,
00167:                 x1, x2, x11, x12, child1, child2);
00168:              child2++;
00169:              break;
00170:           }
00171:        }
00172:     }
00173:
00174:     //divide x2
00175:     if (x2 != x1) //to avoid repeatance
00176:     {
00177:        int x21 = heap1 - x1;
00178:        int x22 = x2 - x21;
00179:        if (x21 <= x22)
00180:           w = x22 - x21;
00181:        else
00182:           w = x21 - x22;
00183:
00184:        curpos2 = step;
00185:        for (k = 0; k < Max_Num; k++)
00186:        {
00187:           if (ws[k] == w)
00188:           {
00189:              node2 = create_node(node1, node2, curpos1, curpos2, step,
00190:                 x1, x2, x21, x22, child1, child2);
00191:              child2++;
00192:              break;
00193:           }
00194:        }
00195:     }
00196:
00197:     if (node2)
00198:     node2->child_ = child2;
00199:
00200:     return child2;
00201: }
```

6.3 如何创建节点？

同 5.4 节。此处省略，具体代码请参考附录 6。

6.4 结果

同 5.5 节，输出结果为所有正确的分解过程。

6.5 讨论

该方法直接计算满足目标的第 3 次分解，好理解，也进一步简化了分解过程。

7. 一个更为简单的方法

由第 2 节从数学的角度分析该问题的分解方法可知，分解方法的解是确定的，其解过程和解个数均是确定的。既然是确定的，那么很容易想到一个问题——***哪种计算机算法能求出这些确定的解？***

通过第 2 节的规则和解过程分析，可以确定，答案是肯定的。本节给出一个更为简单的方法分析***分解方法***的解过程。

7.1 问题分析

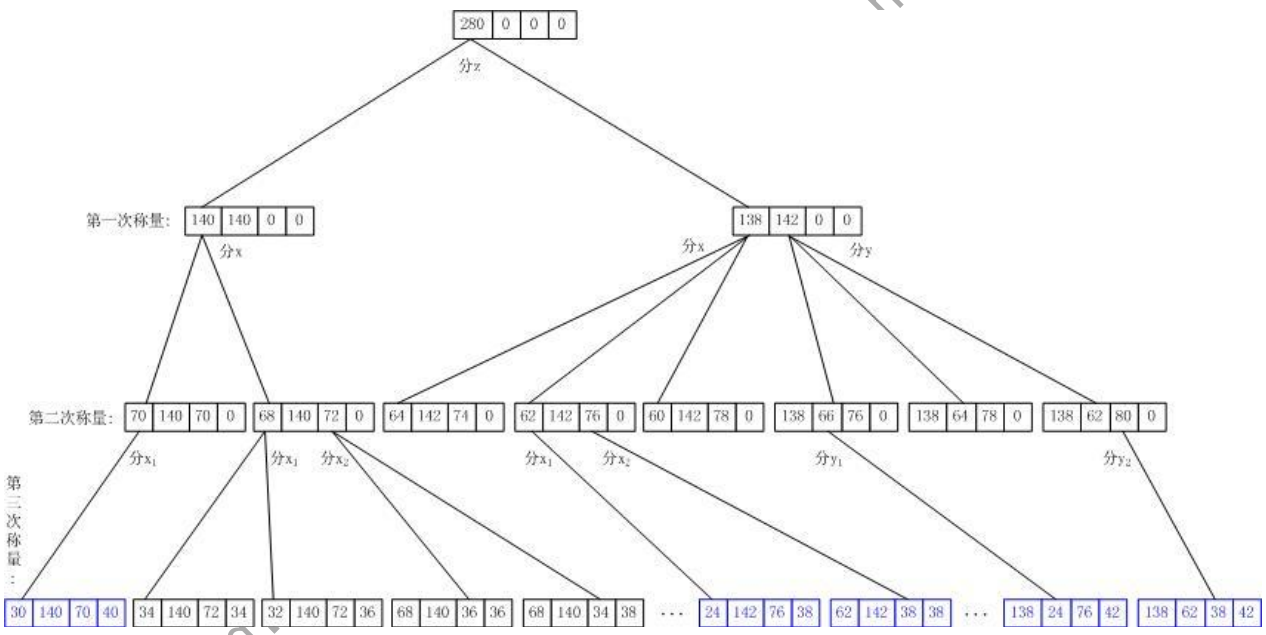
由第 2 节的称量过程可知，在每一次的称量中，对新产生的每一堆(实际上每次新产生 2 堆盐)盐都有 5 中可能的分解方法，如果 3 次称量后没有达到目标，则尝试下一个分解方法，当当前堆的 5 次分解方法全部尝试后仍没有达到目标，则尝试分解另一堆新产生的盐。

——这就是回溯法。

复习一下回溯法的**基本思想**：回溯法是基于对问题实例进行学习，有组织地检查和处理问题实例的解空间，并在此基础上对解空间进行归约和修剪的一种方法。对解空间很大的一类问题，这种方法特别有效。使用回溯法求解的**问题特征**：求解问题要分为若干步，且每一步都有几种可能的选择，而且往往在某个选择不成功时需要回头再试另外一种选择，如果到达求解目标则每一步的选择构成了问题的解，如果回头到第一步且没有新的选择则问题求解失败。

关于解空间，回溯法的详细叙述，可参考有关书籍和资料。关于回溯法的一个例子，可以参考[经典算法（1）--8 皇后问题求解（回溯法）]。

对于第 2 节的讨论，假设保存称量结果，可以画出该问题的状态空间树(搜索树)，第一次/第二次/第三次称量对应的节点分别为 5/50/500 个。因为太大，其中省略了部分向下分解，且对于第一次和第二次称量，只画出满足该次称量限制规则的分解，如下图(同 4.1 节的图)。



为什么第二次称量对应的节点由 5 变成了 50 个？第三次变成了 500 个？

——因为第一次只是对一堆盐进行分解(称量)，即 z=x+y，且该次分解可用 5 种砝码(请参考第 2 节砝码组合分析)，故第一次称量对应 5 个节点。

第二次分解的时候可以对 x 或者 y 都进行分解，而每种分解又可以用 5 种砝码，因此第二次称量对应的节点数为 5×2×5=50。

同理，第三次称量对应的节点数为 50×2×5=500。且这 500 个节点均是叶子节点。

7.2 分解与搜索过程描述

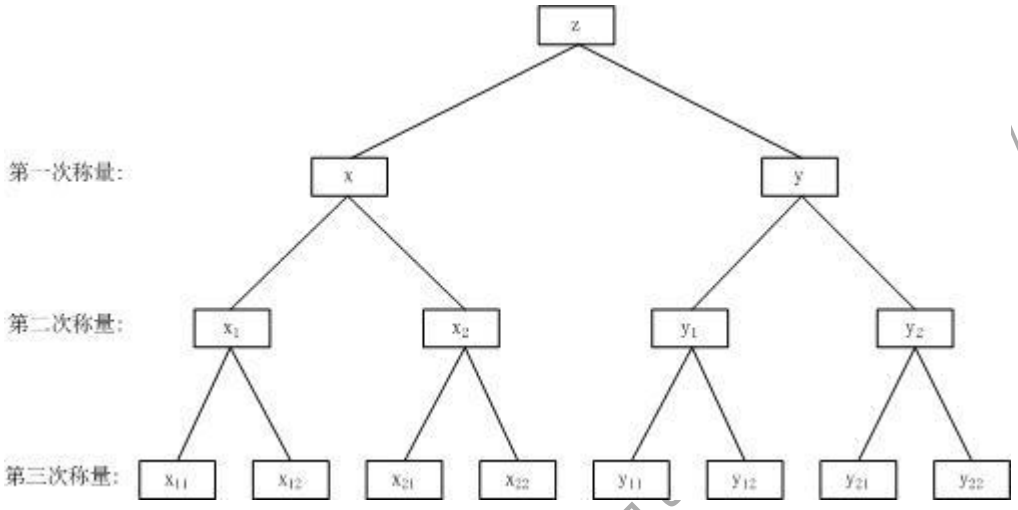
分解与搜索过程如下。

(1) 砝码的组合为：w0，w1，w2，...，wn，满足 w0<w1<w2<...<wn，分解时按从小到大的顺序取砝码进行分解；
(2) 用 wi(0<=i<=n)克的砝码将 z 克的盐分解为 x 克和 y 克的两堆，此处假设 x<=y，有 z=x+y；
(3) 用 wj(0<=j<=n)分解 x，有 x=x1+x2(假设 x1<=x2)；

(4) 用 wk(0<=k<=n)分解 x1，有 x1=x11+x12(假设 x11<=x12)；
(5) 若 x2+x11=heap1 或者 x2+x12=heap1，则该分解为正确的分解，输出该过程；否则，取下一个 wk，继续分解 x1；
(6) 若 wk 已取完，则回溯分解 x2(或者 x2=x1，转(4))；
(7) 若 x2 分解完毕，回溯取下一个 wj，继续分解 x；
(8) 若 wj 已取完，则回溯分解 y(或者 x=y，转(2))；
(9) 若 y 分解完毕，回溯取下一个 wi，继续分解 z；
(10) 直到 wi 也取完，分解过程结束。

7.3 回溯法向普通编程的转化——简单的分解图

如果不保存称量结果，简单的分解图如下。注意，4 对页节点不同时出现。



上述的分解与搜索过程即可转换为普通的编程，3 个层次的嵌套调用即可解决问题，且无需保存每一次称量结果，即可输出全部正确的解。代码如下(文件名 weight5.2.c)，weight5.2.c 由 weight5.1.c 改进而来，weight5.3.c 由 weight5.2.c 改进而来，他们功能完全相同，可参考附录 7。

weight5.2.c

7.4 讨论

相比较前面几节的方法，本节的方法更为简单，只需要 3 层嵌套调用即可求出全部的正确解，且无需保存每一次称量结果即可输出这些正确的解。逻辑简单、直观，实现也简单，应该是首选的方法。

笔者将这个解法放在此处，其主要目的是希望通过循序渐进的思路来找到问题的最佳方法，同时希望能给读者一些启发。

8. 所有代码的自动编译、运行

8.1 如何自动编译？

要自动编译所有的例子代码，很容易想到 makefile 文件。makefile 的编写方法，可参考跟我一起写 makefile、驾驭 makefile、编写 makefile。对于该例子，笔者编写的 makefile 文件如下。

```
CC = gcc
CXXFLAGS += -g -Wall -Wextra

TARGET = weight1 weight2 weight3.1 weight3.2 weight3.3 weight4 weight5.1
weight5.2 weight5.3

CLEANUP = rm -f $(TARGET) *.o

all : $(TARGET)
```

```
clean :
     $(CLEANUP)

weight1.o: weight1.c
     $(CC) $(CXXFLAGS) -c $^
weight2.o: weight2.c
     $(CC) $(CXXFLAGS) -c $^
weight3.1.o: weight3.1.c
     $(CC) $(CXXFLAGS) -c $^
weight3.2.o: weight3.2.c
     $(CC) $(CXXFLAGS) -c $^
weight3.3.o: weight3.3.c
     $(CC) $(CXXFLAGS) -c $^
weight4.o: weight4.c
     $(CC) $(CXXFLAGS) -c $^
weight5.1.o: weight5.1.c
     $(CC) $(CXXFLAGS) -c $^
weight5.2.o: weight5.2.c
     $(CC) $(CXXFLAGS) -c $^
weight5.3.o: weight5.3.c
     $(CC) $(CXXFLAGS) -c $^

all:
weight1: weight1.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight2: weight2.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight3.1: weight3.1.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight3.2: weight3.2.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight3.3: weight3.3.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight4: weight4.o
     $(CC) $(CXXFLAGS) $^ -o $@
weight5.1.o: weight5.1.c
     $(CC) $(CXXFLAGS) -c $^
weight5.2.o: weight5.2.c
     $(CC) $(CXXFLAGS) -c $^
weight5.3.o: weight5.3.c
     $(CC) $(CXXFLAGS) -c $^
     rm -f *.o
```


8.2 如何自动运行并保存结果？

编写自动运行并保存运行结果的脚本 autorun.sh，运行后，其结果被自动保存到同名的.txt 文件中，如下。

```
echo -e "start to run all examples\n"

echo "weight1 running ..."
./weight1 > weight1.txt
echo "    result is in weight1.txt"

echo "weight2 running ..."
./weight2 > weight2.txt
echo "    result is in weight2.txt"

echo "weight3.1 running ..."
./weight3.1 > weight3.1.txt
echo "    result is in weight3.1.txt"

echo "weight3.2 running ..."
./weight3.2 > weight3.2.txt
echo "    result is in weight3.2.txt"

echo "weight3.3 running ..."
./weight3.3 > weight3.3.txt
```

```
echo "    result is in weight3.3.txt"

echo "weight4 running ..."
./weight4 > weight4.txt
echo "    result is in weight4.txt\n"

echo "weight5.1 running ..."
./weight5.1 > weight5.1.txt
echo "    result is in weight5.1.txt"

echo "weight5.2 running ..."
./weight5.2 > weight5.2.txt
echo "    result is in weight5.2.txt"

echo "weight5.3 running ..."
./weight5.3 > weight5.3.txt
echo "    result is in weight5.3.txt"

echo "done. bye."
```

9．问题扩展

面对这样的问题，能否举一反三？能否将其扩展？又如何扩展呢？笔者提供几个思路对其扩展，解决办法，读者可自行编程计算。

(1) 将题目给定的盐的质量及砝码质量**等倍数**增加或减少，达到同样的目标

这个扩展应该是最简单的一种，很好解，只需在程序中将定义的宏，如 total，heap1 和 heap2 进行相应的修改即可。

(2) 将题目给定的盐的质量及砝码质量**不等倍数**增加或减少，问有无解？若有解，有几个解？如何得到这些解？

这个扩展稍微复杂一些，质量不等倍数增加或者减少，首先是砝码组合会发生变化，但还可以参考本文给出的各种方法进行编程计算，修改全局数组 ws[]的内容及相应的称量过程，应该也能解决该问题。

(3) 砝码个数及其质量都发生变化，达到类似的目标，如何解？

砝码组合会发生巨大的变化，求出砝码的组合结果，即本文全局变量 ws[]数组的内容应该是该扩展的重点，可以考虑数学上的组合问题，也可以参考本文**"2.1 砝码组合状态"**的方法，通过一个二进制数表示砝码组合情况，共 n(砝码个数)位，其每一位(0 或者 1)表示砝码出现或者不出现；但该表示方法不能表示砝码出现在天平的哪一端。为了表示砝码出现在天平的哪一端，可以用一个三进制数表示砝码的组合情况，其每一位 0,1,2(此 2 表示-1，同 2.1 节)分别表示不出现，出现在左盘，出现在右盘。该方法也可参考砝码称重问题。

这个扩展的编程有些复杂。关于如何生成数的组合或者排列，可参考笔者的另一文章，或者相关资料。

(4) 更改称量次数，仍然达到类似的目标，又如何解？如果同时砝码个数及其质量也发生了变化，又该如何解？

若再加入 1 次或多次的称量过程，若砝码个数变化不大，还可控，若砝码个数及其质量如(3)也发生变化，则该扩展会变得非常复杂，编程也会很复杂。但解决思路应该还是这些思路，如果前 3 个扩展都能解决，该扩展也能很好的解决，有兴趣的读者可自行编程计算。

(5) 若将每个过程分解出来的盐也当作砝码？又该如何解？

这个扩展就不再本文的范围内了，本文定位于有确定解的数学和计算机分析。实际上，该确定的解就是通过仅仅是已知砝码组合状态分解问题的解，我们在第 2 节和第 7 节均有讨论。

那么这个扩展就类似本文第 1.1 节和第 1.3 节的内容，因此，砝码的组合将出现混合的情况，但本文的思路仍然可以借鉴，有兴趣的读者可自行编程计算。

10. 体会

至此，砝码分盐问题的讨论全部结束。怎么样？是不是觉得本文有点长，甚至有些罗嗦？

此处，笔者要申明的是，笔者并非要炫耀垒码或者写作的能力，主要是想通过问题来锻炼自己思考问题、抽象问题以及解决问题的能力，并锻炼笔者从多角度分析问题的能力；并锻炼笔者讲述问题的思路，形成自己的分析、写作风格。仅此而已。

如果能对读者有一定的启发和帮助，那就是笔者的荣幸了。以下是笔者写作本文的一些体会，与大家分享。

(1) 简单问题，要深入思考，深入挖掘，你会体会其中的乐趣。

(2) 思考，且要深入思考问题的方方面面，包括在计算机中的表示(逻辑表示和物理表示)、存储、解决问题的过程等。

(3) 写作，要详细的写，但不能罗嗦，写重点，要有清晰的思路，包括物理思路和逻辑思路。要对读者负责，不清楚的不能写，以免误导读者。

(4) 数学是百科之母，计算机是工具。如何让计算机为自己服务，为自己工作并解决问题，关键还是对问题的抽象和建模。这里，数学的思维就是你使用计算机的钥匙，算法的好坏直接影响计算机工作的效率和解决问题的能力，同样，数据结构的好坏亦将影响程序的性能，包括时间性能和空间性能。

11. 总结

本文首先给出各种答案，并从数学的角度分析该问题的分解方法(第 2 节)，且在第 3 节给出一个纯数学的计算程序 weight1.c，该方法使用若干个一维、二维、三维数组保存分解结果。该方法数组定义较多，如果加上别的条件，例如再多一次称量，很难扩展。

第 4 节介绍了一种利用树型结构来保存称量过程(分解结果)的方法。比起第 3 节数学的方法，该方法不需要那么多的数组定义，其数据结构统一、简单，且节省大量空间。但该方法也有缺点，例如，最终的正确解只有 5 个，但该方法同第 3 节的数学方法一样，计算了所有的有结果的解，共 37 个，也比较浪费。

第 5 节是第 4 节树型结构保存称量过程方法的改进版本，即对第 3 次分解结果进行判断，只有当该称量过程满足目标时，才建立节点，包括第 2 层和第 3 层的节点。这样，没有分解结果的节点(包括第 2 层和第 3 层的节点)就不会被建立，其叶子节点全部是正确的分解结果。问题更直观，且节省了空间。

第 6 节从直接计算满足目标的第 3 次称量出发，即第 2 次分解(称量)后，直接将满足目标的第 3 次称量解求出，判断这个计算出来的第 3 次称量所用的砝码是否在砝码组合内。如果在，表明该次称量过程为所求的正确解；否则，放弃，继续判断。该方法直接计算满足目标的第 3 次分解，更好理解，也进一步简化了分解过程。

第 7 节从不保存每一称量结果的角度出发，利用回溯法对该问题进行分析，并将回溯法转化为普通编程。相比较前面几节的方法，本节的方法更为简单，只需要 3 层嵌套调用即可求出全部的正确解，且无需保存每一次称量结果即可输出这些正确的解。逻辑简单、直观，实现也简单，**应该是首选的方法**。

第 8 节从程序编译、运行等方面，讨论了本文所有例子程序的自动编译、自动运行并保存运行结果的方法。

第 9 节简单讨论了该问题的扩展。第 10 节简单讨论了笔者写作本文的体会。第 11 节，即本文最后一节总结全文及本文讨论的各种方法。

Reference

<计算机算法设计与分析>
砝码称重问题 (阿波)

编写 makefile （阿波）
驾驭 makefile （李云）
跟我一起写 makefile （陈皓）
经典算法（1）——8 皇后问题求解（回溯法） （阿波）

附录 1：数学分解的代码 weight1.c

weight1.c

附录 2：数学分解程序 weight1 的运行结果

weight1.txt

附录 3：树结构分解的代码 weight2.c

weight2.c

附录 4：再改进的方法的代码 weight3.1.c/3.2.c/3.3.c

weight3.1.c

weight3.2.c

weight3.3.c

附录 5：再改进的方法的代码 weight3.1.c/3.2.c/3.3.c 的输出结果

weight3.txt

附录 6：直接计算正确分解的代码 weight4.c

weight4.c

附录 7：一个更简单的方法的代码 weight5.1.c/5.2.c/5.3.c

weight5.1.c

weight5.3.c