第3章 蛮力法

本章介绍蛮力法，这是一种最简单直接的设计策略。蛮力法也叫暴力法、枚举法或穷举法。蛮力法解决问题比较简单直接，常常基于问题的描述和所涉及的概念、定义直接求解，逐一列举并处理问题所涉及的所有情形，而后得到问题的答案。因此，把蛮力法概括为一句话就是：直接去做吧。

蛮力法依据的基本技术是扫描技术和枚举方法，关键是依次处理所有的元素或依次处理所有可能的情况。蛮力法的优点是逻辑清晰，往往是解决问题的最直接的算法，但其缺点是效率不高。因此蛮力法是很多算法的基础，对蛮力法加以优化，就可以得到效率更高的算法。

3.1 概述

通常蛮力法使用循环结构来实现，对所处理问题的所有对象或所有可能的情况逐一进行处理。通过循环的上下限控制枚举的范围，循环体中根据问题的定义进行结果的筛查，以便求得所要求的解。

蛮力法框架描述如下。

```
BruteForce()
{
    n = 0;
    for(k = 区间下限; k <= 区间上限; k++)
        if(约束条件)
        {   找到一个解;
            n++;
        }
    return (汇总的解);
}
```

采用扫描或枚举方法的蛮力法求解问题，通常需要实施以下几个步骤。

(1) 确定扫描或枚举变量。

(2) 确定枚举变量的范围，设置相应的循环。

(3) 根据问题要求确定约束条件，以找到合理的解。

在第1章中我们已经采用过一次蛮力法，那就是在求解两个正整数的最大公约数解法

中的循环测试法，在那里可以看到蛮力法的表现。

虽然巧妙高效的算法很少来自蛮力法，但是绝不能忽视蛮力法的作用，具体表现在：

(1) 蛮力法求解问题的范围是最少受限制的，因为蛮力法是以问题的定义为依据的，而问题的定义最能够包含问题的空间。

(2) 在实际应用中，如果待解决的问题规模不大，其运算时间是可以接受的，此时，设计一个更高效的算法在代价上不值得。

(3) 蛮力法可以作为某类问题解决方案时间效率的底限，用于评估同类问题更高效的算法，为教学和科研服务。

因此，在算法规模不大、没有必要使用优化算法或某些优化算法过于复杂(可能因为复杂的算法反而浪费时间)的情况下，蛮力法仍是可选的方法之一。

例 3.1 百鸡百钱问题。

我国古代数学家张丘建在《张丘建算经》一书中提出了“百鸡问题”：今有鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？

【分析】 设鸡翁、鸡母、鸡雏的个数分别为 x、y、z，题意给定共 100 钱要买百鸡，若全买公鸡最多买 20 只，显然 x 的值在 0～20 之间；同理，y 的取值范围在 0～33 之间，可得到下面的不定方程：

$$\begin{cases} 5x + 3y + z/3 = 100 \\ x + y + z = 100 \end{cases}$$

所以此问题可归结为求这个不定方程的整数解的问题。

由程序设计实现不定方程的求解与手工计算不同。在分析确定方程中未知数变化范围的前提下，可通过对未知数可变范围的穷举，验证方程在什么情况下成立，从而得到相应的解。实现算法如下：

```
void HundredFowlsMoney ()
{
    for(x = 0;x <= 20;x++)          //外层循环控制鸡翁数 x 在 0～20 变化
        for(y = 0;y <= 33;y++)      //内层循环控制鸡母数 y 在 0～33 变化
        {
            z = 100 - x - y;        //内外层循环控制下,鸡雏数 z 的值受 x,y 的值的制约
            //验证取 z 值的合理性及得到一组解的合理性
            if(z % 3 == 0&&5 * x + 3 * y + z/3 == 100) 输出 x,y,z 的值;
        }
}
```

运行结果：

1：$x=0$ $y=25$ $z=75$

2：$x=4$ $y=18$ $z=78$

3：$x=8$ $y=11$ $z=81$

4：$x=12$ $y=4$ $z=84$

下面各节将讨论蛮力法在各个实际问题中的应用。

3.2 排序问题

本节考虑用蛮力法求解排序问题。排序问题在第 1 章已经介绍过，为了简化问题，本书所讲的排序问题是指输入是一个可排序的 n 个元素的序列，输出是非降序的序列。

目前人们为解决排序问题已经研究出了几十种排序算法，其中最简单直接的排序方法是选择排序和冒泡排序，这两种算法恰好体现了蛮力法基于问题本身的、清晰的解决思路。

3.2.1 选择排序

选择排序的基本思想是：对有 n 个元素的序列进行 $n-1$ 趟排序。第一趟排序对序列从头到尾进行扫描，找到最小元素与第一个元素交换；第二趟排序对序列从第二个元素起到序列尾扫描，仍找最小元素，然后与第二个元素交换；一般来说，第 i 趟排序从序列的第 i 个元素起到序列尾的 $n-i+1$ 个元素中找最小元素，与第 i 个元素交换。按上述方法完成对 n 个元素序列的排序。

为简单起见，假定序列元素存放在数组中。算法如下：

```
void SelectionSort(A[0..n-1])
{
    for(int i=0; i<=n-2; i++)
    {
        min = i;
        for(int j=i+1; j<=n-1; j++)
            if(A[j]<A[min]) min=j;
        swap  A[i]and A[min]
    }
}
```

作为例子，下面给出了对于序列 80,18,72,95,29,45,12 执行算法的操作步骤，如下所示。

|80 18 72 95 29 45 **12** i=0：min 最后得 6，交换二者。
12 |**18** 72 95 29 45 80 i=1：min 最后得 1。
12 18 |72 95 **29** 45 80 i=2：min 最后得 4，交换二者。
12 18 29 |95 72 **45** 80 i=3：min 最后得 5，交换二者。
12 18 29 45 |**72** 95 80 i=4：min 最后得 4。
12 18 29 45 72 |95 **80** i=5：min 最后得 6，交换二者。
12 18 29 45 72 80 |95 结束。

算法执行过程中，竖线左侧是已经排好序的元素，从竖线右侧的第一个元素开始本轮扫描，直到扫描完最后一个元素，即得本轮要找的最小元素；最小元素的初始位置设置为 i，加粗字体的元素表示本轮扫描得到的最小元素，将最小元素和第 i 个位置的元素交换位置，则完成一次扫描。该序列有 7 个元素，经过 6 趟扫描即可完成排序。

对选择排序算法进行复杂度分析，步骤如下。

(1) 决定算法的输入规模 n 为待排序的元素数量。

(2) 选取算法的基本操作是最内层循环中的元素比较，即当前元素 A[j]和目前的最小元素 A[min]的比较。

(3) 分析算法的基本操作执行次数只与输入规模 n 有关，和具体输入情况无关，所以无须区分最差效率、平均效率和最优效率的情况。

(4) 建立基本操作的执行次数的求和表达式：

$$C(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}$$

(5) 确定时间复杂度 $C(n)=\frac{n(n-1)}{2}\in\theta(n^2)$。

所以，对于任何输入，选择排序的算法效率类型是 $\theta(n^2)$。

对于排序问题，主要操作除了元素的比较外，还有一项是元素的交换操作，算法 SelectionSort 的元素交换操作的复杂度为 $O(n)$，即每进行一趟排序最多只交换一次。

第 1 章中介绍了算法的两个重要特性：稳定性和在位性。对于算法 SelectionSort，除了需要一个额外变量 t 来作为元素交换时的辅助空间外，不再需要额外的辅助空间，因此其具有在位性。若有序列 4,4,2，利用选择排序，算法执行过程如下：

$$\begin{array}{lll} |4_1 & 4_2 & 2 \\ 2 & |4_2 & 4_1 \\ 2 & 4_2 & 4_1 \end{array}$$

可知，排序之后第一个元素 4 被交换到后面，而第二个元素 4 排到了前面。由此可知，选择排序不具有稳定性。

3.2.2 冒泡排序

蛮力法在排序问题上的另一个应用是冒泡排序。冒泡排序的基本思想是：对有 n 个元素的序列也进行 $n-1$ 趟排序。第一趟排序对序列从头到尾进行相邻元素比较，如果逆序(即大的在前，小的在后)，则交换相邻的两个元素，这样一趟排序后，最大的元素“沉”到了序列的第 n 个位置；第二趟排序对序列从头到第 $n-1$ 个元素进行相邻元素比较，如果逆序则交换，这样第二趟排序后，次大元素“沉”到了序列倒数第二的位置；第 i 趟排序从序列的头到第 $n-i+1$ 个元素进行相邻元素比较，做同样的操作，一趟排序后，第 i 大的元素落到了第 $n-i+1$ 的位置上。经过 $n-1$ 趟排序，即可完成排序。算法如下。

```
void BubbleSort(A[0..n-1])
{
    for(int i=0; i<=n-2; i++)
        for(int j=0; j<=n-2-i; j++)
            if(A[j+1]<A[j]) swap A[j] and A[j+1]
}
```

作为例子，对于序列 80,18,72,95,29,45,12 执行算法的操作步骤，如下所示。

80 18 72 95 29 45 12

18 72 80 29 45 12 |**95** i=0：最大值 95 就位。

18 72 29 45 12 |**80** 95 i=1：第二大值 80 就位。

18 29 45 12 |**72** 80 95 i=2：第三大值 72 就位。

18 29 12 |**45** 72 80 95 i=3：第四大值 45 就位。

18 12 |**29** 45 72 80 95 i=4：第五大值 29 就位。

12 |**18** 29 45 72 80 95 i=5：第六大值 18 就位。排序结束。

算法执行过程中，竖线右侧是已经排好序的元素，扫描时从左侧开始，每次比较当前元素及其右侧元素，如果是逆序，则交换，这样的结果是每轮扫描都把当前的极大值“沉”到当前序列的最末尾。这个序列有 7 个元素，经过 6 轮扫描即可完成排序。

对选择排序算法进行复杂度分析，步骤如下。

(1) 算法的输入规模 n 为待排序的元素数量。

(2) 算法的基本操作是最内层循环中的元素比较(比较的是当前元素 A[j]和其右侧元素 A[j+1]，虽然该操作在最内层循环的 if 语句中)。

(3) 算法的基本操作执行次数只与输入规模 n 有关，和具体输入情况无关。虽然基本操作在最内层循环的 if 语句中，但其为条件，所以每次都执行，而且循环没有中途退出的情况。

(4) 基本操作的执行次数

$$C(n)=\sum_{i=0}^{n-2}\sum_{j=0}^{n-2-i}1=\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}$$

(5) $C(n)=\frac{n(n-1)}{2}\in\theta(n^2)$。

所以，对于任何输入，冒泡排序都是复杂度为 $O(n^2)$的算法。

同样的，来看元素的交换操作，算法 BubbleSort 的元素交换操作的最差情况(逆序输入时)会达到 $O(n^2)$，因为每执行一次元素的比较操作，条件都成立，那这样元素的交换操作和元素的比较操作复杂度相同。而最优情况下(即序列已经有序)，算法 BubbleSort 的元素交换操作将会是 0。

下面分析算法 BubbleSort 的稳定性和在位性。首先，算法 BubbleSort 除了需要一个额外变量 t 来做交换时的辅助空间外，不需要更多的额外辅助空间，所以其具有在位性。冒泡排序具有稳定性，这里不再举例，可以简单地看一下，序列每次做比较的都是相邻元素，如果前一个元素和后一个相等，那最内层的条件不会为真，则不会发生元素的交换，这样等值的元素，前面的不会被交换到后面去，因此，冒泡排序具有稳定性。

对于冒泡排序，可以有一个简单的改进，就是设置一个检查值，如果某次扫描的时候没有发生元素的交换，则证明当前序列已经排好序了，无须后续的比较，算法提前结束，增加算法的性能。

3.3 查找问题

本节考虑用蛮力法求解查找问题。查找问题也已经介绍过，这一节介绍元素序列的顺序查找和字符串的顺序查找。如果大家学习过《数据结构》课程，应该已经掌握了若干种查找算法。

3.3.1 顺序查找

第2章介绍算法的时间复杂度分析方法时，曾经两次提到过这个问题的算法，且顺序查找算法是直观的说明蛮力法的一个例子，请参阅第2章的算法 SeqSearch（A[0..n－1]，K），分析这个算法效率的时候会分最差效率、最优效率和平均效率。

最差效率：　　$C_{worst}(n)=n$

最优效率：　　$C_{best}(n)=1$

平均效率：　　$C_{avg}(n)=n*(1-p)+\frac{p*(n-1)}{2}$

从以上的算法及其分析来看，顺序查找有着蛮力法典型的优点和缺点。优点是简单直接，缺点是效率低下。当然，对于蛮力法总是离不开一个话题就是如何提高效率，这里一个简单的办法是将 A[0..n－1]扩展为 A[0..n]，在 A[n]的位置放置查找关键字 K，作为监视哨。这样在循环语句 while（i<n && A[i]< >K）中，可以省去 i<n 的条件判断，由条件 A[i]< >K，在 A[n]的位置一定能结束循环，从而提高了算法的效率。

3.3.2 字符串匹配

字符串匹配问题现在在许多领域都有广泛地应用，在文档编辑器中查找、替换等功能也是基于字符串匹配等算法。这里，待查找的字符串称为文本，与其进行部分匹配的字符串称为模式。蛮力法求字符串匹配问题就是从文本中使用蛮力法找到初次匹配模式的位置。

字符串匹配问题的蛮力算法需要将模式对准文本的开始位置，从左向右逐一检查二者的对应字符，如果相同，则检查下一字符；如果不同，则将模式的起始位置重新设成开始位置（相当于对应的匹配位置又重新开始）。如果成功找到匹配位置，则返回最左端字符在文本中的位置。如果匹配的位置距离文本的最后字符长度小于模式的长度，则宣告匹配失败。如果用 W[0..n－1]表示文本，用 M[0..m－1]表示模式，则在文本中可能匹配成功的位置为0至 $n-m$，因为 $n-m$ 的右侧字符距离文本的最后一个字符 W[n－1]的长度已经不足 m，匹配一定不成功。针对以上叙述有如下算法：

```
//蛮力法求解字符串匹配问题
//输入：文本 W[0..n－1]，模式 M[0..m－1]，二者均为字符数组
//输出：如果查找成功，返回第一个匹配模式的子串在文本中的首字符位置
//如果查找失败，返回－1
int BruteforceIndex (W[0..n－1], M[0..m－1])
{
    for(int i = 0; i <= n－m; i++)
    {
        int j = 0;
        while (j < m)&&(W[i + j] == M[j])
        {
            j++;
            if (j == m) return i;                    //查找成功
        }
    }
```

```
    return (-1);                                    //查找失败
}
```

现在分析这个算法的效率，步骤如下。

(1) 算法输入规模的度量为两个变量：n 和 m，分别代表文本和模式的长度。

(2) 算法的基本操作为内层循环的操作。内层循环为 while 循环，其操作除 W[i+j]==M[j]为字符间的比较外，其他操作为下标操作，即数组的下标变量操作。因此，基本操作选为 W[i+j]==M[j]字符间的比较操作。

(3) 因为最内层循环的操作中存在分支语句，而且分支语句在条件成立时有 return i，因此，循环的执行次数不只与输入规模有关，还与输入的具体情况有关，因此需要分析最优复杂度、最差复杂度以及平均复杂度。

(4)～(5)最优复杂度：如果模式很短(即 m 很小)，则 $C_{best}(n,m)=m$(即在文本的第一个位置成功匹配模式)；如果文本和模式的长度相近(即 $n-m$ 很小)，则 $C_{best}(n,m)=n-m$(即不能成功匹配模式，并且每次匹配均在第一个字符就匹配失败)。因此，$C_{best}(n,m)=\min\{m,n-m\}$。

最差复杂度：$C_{worst}(n,m)=(n-m+1)m\in\theta(nm)$，通过算法可以简单得到这个结论，也就是外层循环执行 $n-m+1$ 次，内层循环每次都执行 m 次。例如，文本为 aaaaaaaaaaaaaaaaaac，而模式为 aaab 时，会出现这种极端的情况，而一般情况下，例如在自然语言文本中查找词的情况下则很少出现，在这种情况下该算法的平均效率要比最差效率好很多，大约是 $\theta(n+m)$的复杂度。

在这一节的最后来看蛮力法求解字符串匹配问题的实例分析，文本为 patch not applied，模式为 app。

```
patch not applied
app                                                 //匹配一次，不成功
 app                                                //匹配两次，不成功
  app                                               //匹配一次，不成功
   app                                              //匹配一次，不成功
    app                                             //匹配一次，不成功
     app                                            //匹配一次，不成功
      app                                           //匹配一次，不成功
       app                                          //匹配一次，不成功
         app                                        //匹配一次，不成功
          app                                       //匹配一次，不成功
           app                                      //匹配三次，成功
```

共计匹配 14 次，最终在 $W[10]$的位置匹配成功。从本例可以看出，实际的匹配在很多位置只需要很少次匹配就可以得到匹配不成功的结果。

3.4 几何问题

本节讨论使用蛮力法求解计算几何中的两个著名问题：最近对问题和凸包问题。这两个问题都是求解平面上有限点集的问题，它们都有相应的实际应用价值。

3.4.1 最近对问题

问题描述：最近对问题是求解平面点集 n 个点中距离最近的两个点间的问题。为简单起见，在二维坐标平面来考虑该问题。如果讨论的点以标准二维坐标形式给出，则有点 $P_i(x_i, y_i)$ 和 $P_j(x_j, y_j)$，二者的两点间距离可以利用公式 $d(P_i, P_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$。通过这个公式，可以计算平面上任意两点之间的距离。

因此，蛮力法的求解思路就是对平面 n 个点两两组对，计算并记录最小距离。这里需要注意两个问题，其一，点对之间的距离是没有方向性的，即 $d(P_i, P_j) = d(P_j, P_i)$，所以在计算点对之间的距离时，任意两点只需要计算一次距离即可，如果点存放在数组中，则只需要考虑下标 $i < j$ 的所有情况；其二，在计算点对之间的距离时，如果仅仅为了比较大小，可以不计算开平方，而是计算距离的平方。简化了公式的计算。算法如下。

```
//蛮力法求解最近对问题.输入为 n 个点的列表,P[i][0]存 i 点的横坐标
//P[i][1]存 i 点的纵坐标.输出距离最近的两个点的下标 ind1 和 ind2
double ClosestPoints(P[n][2],int * ind1, int * ind2)
{
    mind = + ∞;                                          //初值赋成最大值
    double dis = 0;
    ind1 = 0;ind2 = 0;
    for(int i = 0; i < = n - 2; i++)
        for(int j = i + 1; j < = n - 1; j++)
        {
          dis = (P[j][0] - P[i][0]) * (P[j][0] - P[i][0]) + (P[j][1] - P[i][1]) * (P[j][1] - P[i][1]);
          if (dis < mind) {mind = dis; &ind1 = i; &ind2 = j;}
        }
    return(mind);
}
```

对以上算法进行复杂度分析，步骤如下。

(1) 算法的输入规模 n 为点集中点的个数。

(2) 算法的基本操作为最内层循环的坐标乘法。

(3) 算法基本操作的执行次数只与输入规模 n 有关，尽管最内层循环中有分支语句，但不会中途跳出循环，而且基本操作的执行与分支语句无关。因此直接求执行次数 $C(n)$ 即可。

(4) ～ (5) $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 2 = 2\sum_{i=0}^{n-2}(n-1-i) = 2 * \frac{n(n-1)}{2} = n(n-1) \in \theta(n^2)$。

所以，蛮力法求解最近对问题的复杂度是平方级的。

3.4.2 凸包问题

接下来介绍使用蛮力法求解凸包的问题。凸包的定义，直观的来看就是，由平面上 n 个点组成的集合，其凸包就是包含这些点的最小凸多边形，凸多边形的任何一条边所在的直线把凸多边形全部划在了同一个半平面内。

举例来说，如果有平面上的两个点，则包含这两点的凸包是他们构成的线段。如果是平面上的三个点，如果这三点不共线，则包含这三点的凸包是以这三点为顶点构成的三角形；如果这三点共线，则包含这三点的凸包是由距离最远的两点构成的线段（第三点在线上）。令 S 是平面上的一个点集，封闭 S 中所有点的最小凸多边形，称为 S 的凸包，表示为 CH(S)。在图 3.1 中，由线段表示的多边形就是点集 $S=\{p_0,p_1,\ldots,p_{12}\}$ 的凸包。

求取凸包的常用算法有蛮力法、格雷厄姆扫描法（Graham）、分治法和 Jarris 步进法，等等。在这里介绍蛮力法。

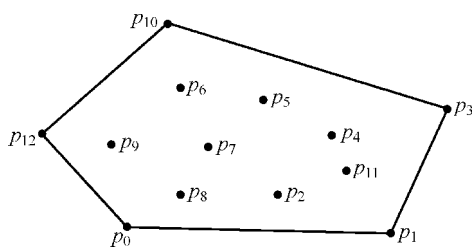

图 3.1　点集 S 的凸包 CH(S)

凸包有一个重要的性质：如果点集中两个点的连线属于凸多边形的边，当且仅当点集中其余的点都在这两个点连线的同一侧。利用这个性质，可以求解凸包问题。若已知由 n 个点构成集合的凸包是以其中某些点为顶点的凸多边形（这个凸多边形一定是最小凸多边形），且这些点具有凸包的性质，则它们之间的某些连线共同构成了凸包的全部边。

为了解决凸包问题，要做的就是从这 n 个点中挑出这些顶点，这些顶点称为“极点”。图 3.1 中的极点是 p_0、p_1、p_3、p_{10}、p_{12}，它们的连线 p_0p_1、p_1p_3、p_3p_{10}、$p_{10}p_{12}$、$p_{12}p_0$ 构成了凸包的边。由此可见，解决凸包问题需要找出所有极点，并且将所有极点按顺时针或逆时针顺序排列。

蛮力法求解凸包问题的基本思想：对于由 n 个点构成的集合 S 中的两个点 p_i 和 p_j，当且仅当该集合中的其他点都位于穿过这两点的直线的同一边时（假定不存在三点同线的情况），它们的连线是该集合凸包边界的一部分。对每一对顶点都检验一遍后，满足条件的线段构成了该凸包的边界。

在平面上，穿过两个点(x_1,y_1)和(x_2,y_2)的直线是由下面的方程定义的：

$$ax + by=c \quad (\text{其中},a=y_2-y_1,b=x_2-x_1,\ c=x_1y_2-y_1x_2)$$

这样一条直线把平面分成了两个半平面，其中一个半平面中的点都满足 $ax+by>c$，另一个半平面中的点都满足 $ax+by<c$，因此，为了检验这些点是否位于这条直线的同一边，可以简单地把每个点代入方程 $ax+by=c$，检验这些表达式的符号是否相同。当然，如果这时有 $ax+by=c$，这说明第三点和前两点共线，这时可以重新审核可能的凸包边界，即这三点中靠两侧的点是可能的极点，而中间的点则不是。算法如下。

```
//凸包问题,在全局变量 point[0..n-1]中存放 n 个点
//point[i]为结构体,point[i].x 和 point[i].y 分别存放点的横、纵坐标值
//point[i].flag 存放是否为极点的标志
void ConvexHull()
{
    for(i=0;i<n;++i)
        for(j=j+1;j<n;++j)
        {
            a = point[j].y - point[i].y;
            b = point[j].x - point[i].x;
```

```
            c = (point[i].x * point[j].y) - (point[i].y * point[j].x);
            sign1 = 0;                              //sign1 和 sign2 分别记录直线两边点的数量
            sign2 = 0;
            for(k = 0;k < n;k++)
            {
                if((k == j) || (k == i)) continue;
                if ((a * point[k].x + b * point[k].y) == c)
                    {++sign1; ++sign2;};          //共线,则算两边都有该点
                if ((a * point[k].x + b * point[k].y)> c)
                    ++sign1;
                if ((a * point[k].x + b * point[k].y)< c)
                    ++sign2;
            }
            if (((sign1 == (n - 2)) || ((sign2 == (n - 2)))
            {
                point[i].flag = 1;
                point[j].flag = 1;                  // my_point[i].flag 为 1 表示是极点
            }
        }
    //此处可以按顺时针或逆时针序输出 flag 为 1 的点
    }
```

这个算法存在一个问题，就是只给出了全部的凸包极点，但是没有将这些极点按照顺时针或逆时针排序，这个问题请大家作为思考题去解决，提示的思路是可以在凸包的内部找一个点，然后以这个点为中心，顺时针或逆时针输出所有的极点。

上述算法由所有点两两组合，共组成了 $n(n-1)/2$ 条边，每条边都要对其他 $n-2$ 个顶点求出在直线方程 $ax+by=c$ 中的符号，所以，其时间复杂度是 $O(n^3)$。

在这节中，讨论了两个几何问题：最近对问题和凸包问题，很显然，蛮力法求解这两个问题简单直接，使用的思路是最简单的定义法，但是效率比较低下，这也印证了在本章之初介绍蛮力法的基本思想时所做的分析。后续的章节中会给出解决这些问题的更好的方法，请大家先思考如何改进这两个问题的解法。

3.5 组合问题

在日常生活中经常遇到排列、组合或者给定集合的子集等问题，并且要求找到一个方案，使得某些期望的特性达到最大值或者最小值，比如路径长度或者分配成本等等。对于这类问题，往往用穷举法，思路简单直接，就是将所有可能的情况一一列举出来，然后计算每种情况的期望特性值，从而找出最大值或者最小值。虽然这种穷举法思路比较简单，也看似容易实现，但是由于穷举往往会产生组合对象，甚至很快达到组合爆炸的情况，所以这种思路只能解决规模非常有限的问题。在这一节里通过两个重要的应用(旅行商问题和背包问题)来阐述蛮力法求解组合问题的方法。

3.5.1 旅行商问题

旅行商问题(Traveling Saleman Problem，TSP)又译为旅行推销员问题、货郎担问题。

该问题是最基本的路线问题，求解旅行者由起点出发，通过所有给定的点之后，最后再回到起点的最小路径成本。该问题的描述虽然简单，但在地点数目增多后求解却极为复杂。以100个地点为例，如果要列举所有路径后再确定最佳行程，那么总路径数量之大，几乎难以计算出来。多年来全球数学家绞尽脑汁，试图找到一个高效的算法，近来在大型计算机的帮助下才取得了一些进展。

旅行商问题最简单的求解方法是枚举法，搜索空间是 n 个点的所有排列的集合，复杂度为 $(n-1)!$。这个问题可以用加权图来建模，即用加权图的顶点表示要到达的各个城市，用边的权重表示城市间的距离，所得的结果是由 $n+1$ 个顶点的序列构成的，其中开始和结束的顶点相同，都是指定的出发城市，其余 $n-1$ 个顶点是剩下的 $n-1$ 个不同城市的排列，所以得到的枚举规模是 $n-1$ 的全排列（当然如果不考虑路径的方向性，则路径条数是 $n-1$ 全排列数的二分之一）。

图 3.2 给出了旅行商问题的一个小规模实例。

图 3.2 的加权图的邻接矩阵为：

$$\begin{bmatrix} \infty & 3 & 3 & 2 & 6 \\ 3 & \infty & 7 & 3 & 2 \\ 3 & 7 & \infty & 2 & 5 \\ 2 & 3 & 2 & \infty & 3 \\ 6 & 2 & 5 & 3 & \infty \end{bmatrix}$$

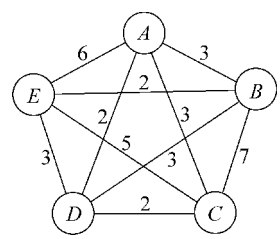

图 3.2　旅行商问题实例

不失一般性，假定出发点和结束点均为 A，表 3.1 给出了采用蛮力法求解旅行商问题的所有线路和相应的代价。

表 3.1　旅行商问题的求解过程

线　路	代　价	线　路	代　价
A-B-C-D-E-A	3+7+2+3+6=21	A-D-B-C-E-A	2+3+7+5+6=23
A-B-C-E-D-A	3+7+5+3+2=20	A-D-B-E-C-A	2+3+2+5+3=15
A-B-D-C-E-A	3+3+2+5+6=19	A-D-C-B-E-A	2+2+7+2+6=19
A-B-D-E-C-A	3+3+3+5+3=17	A-D-C-E-B-A	2+2+5+2+3=14
A-B-E-C-D-A	3+2+5+2+2=14	A-D-E-B-C-A	2+3+2+7+3=17
A-B-E-D-C-A	3+2+3+2+3=13(最优解)	A-D-E-C-B-A	2+3+5+7+3=20
A-C-B-D-E-A	3+7+3+3+6=22	A-E-B-C-D-A	6+2+7+2+2=19
A-C-B-E-D-A	3+7+2+3+2=17	A-E-B-D-C-A	6+2+3+2+3=16
A-C-D-B-E-A	3+2+3+2+6=16	A-E-C-B-D-A	6+5+7+3+2=23
A-C-D-E-B-A	3+2+3+2+3=13(最优解)	A-E-C-D-B-A	6+5+2+3+3=19
A-C-E-B-D-A	3+5+2+3+2=15	A-E-D-B-C-A	6+3+3+7+3=22
A-C-E-D-B-A	3+5+3+3+3=17	A-E-D-C-B-A	6+3+2+7+3=21

从表 3.1 可以看出，一共有 24 种路线，如果路线是无向的，则只有 12 种路线。如果不假定从 A 点出发最终回到 A 点，而是可以从任意点出发回到该点，那么复杂度应该是 $n!$。根据在第 2 章学习的函数增长情况，对于这种复杂度的算法只能计算当 n 很小的时候的情况，因为 $n!$ 的增长速度实在是太快了。

3.5.2 背包问题

背包问题的名称来源于如何选择最合适的物品放置于给定背包中，在 1978 年由 Merkel 和 Hellman 提出。背包问题经常出现在商业、组合数学、计算复杂性理论、密码学和应用数学等领域中，也可以将背包问题描述为决策问题，即在总重量不超过 W 的前提下，总价值是否能达到 V。

0-1 背包问题是最基本的背包问题，描述如下：给定一组物品，每种物品都有自己的重量和价值，在限定的总重量内将物品装入背包，且每个物品只有 0 和 1 两种选择，即要么物品完全放入背包，要么完全不放入背包，最终使背包内物品的总价格最高。

举例如下，这里有 4 件物品，背包容量 $W=10$，4 件物品的重量及价值参见表 3.2，求解将哪些物品装入背包可使这些物品的重量总和不超过背包容量，且价值总和最大。

表 3.2 4 件物品的重量及价值

选项	物品 1	物品 2	物品 3	物品 4
重量 $w[i]$	2	2	6	5
价值 $v[i]$	6	3	5	4

根据蛮力法，可枚举所有物品装入背包的可能性，如表 3.3 所示。

表 3.3 4 件物品装入背包的可能情况

装入物品子集	总重量	总价值	选择情况
φ	0	0	
{1}	2	6	
{2}	2	3	
{3}	6	5	
{4}	5	4	
{1,2}	4	9	
{1,3}	8	11	
{1,4}	7	10	
{2,3}	8	8	
{2,4}	7	7	
{3,4}	11	9	超重
{1,2,3}	10	14	最终结果
{1,2,4}	9	13	
{1,3,4}	13	15	超重
{2,3,4}	13	12	超重
{1,2,3,4}	15	18	超重

在这个问题中，蛮力法枚举给出的是 4 个物品集合的所有子集，为找出最终解，需要计算所有子集的总重量及总价值(如果超重可以不计算总价值)，然后找出符合重量要求的最大价值。本例最终结果是装入物品 1、2 和 3，取得的最大价值是 14。由于 4 个物品的子集数是 $2^4=16$，所以上例一共列举了 16 种可能的情况。如果推广成一般情况，物品数是 n，则

蛮力法需要枚举 2^n 种情况，所以这种算法的最终效率为 $O(2^n)$。

从上面两个问题的分析可以看到，旅行商问题的蛮力法复杂度是 $O(n!)$，背包问题的蛮力法复杂度是 $O(2^n)$，二者的复杂度类型都属于增长非常显著的。实际上，这两个问题就是所谓的 NP 困难问题中最著名的例子。对于 NP 困难问题，目前没有已知的、效率类型可以用多项式来表示的算法，也就是说无法用计算机在有限时间内解决所有这样的问题。多数计算机科学家相信，这样的算法是不存在的，虽然这个猜想并未得到证实（当然，也没被证伪）。对于这种类型的问题，将会在后续的章节讨论其更优的解法。

总结

本章介绍了蛮力法。算法的思想就是根据问题的定义采用直接的方法进行求解，这种方法简单直接，能够最广泛地解决问题，同时也是衡量其他算法的最低标准。一般而言，蛮力法有其自身的缺陷，那就是效率不高，但由此也给大家留下了改进的空间，那就是如何进一步设计算法，以提高效率。在接下来的章节中，可以对照每种问题的蛮力法来探索新的算法。

习题 3

1. 简述蛮力法的基本思想及特点。

2. 请针对实例 2、6、1、4、5、3、2，分别用选择排序和冒泡排序进行排序，并写出详细过程。

3. 选择排序稳定吗？冒泡排序稳定吗？

4. 如果对列表进行一次比较之后，没有元素交换位置，则排序已经完成，根据这条，对冒泡排序进行改进。

5. 对于共线的 n 个点，其最近对问题有没有比蛮力法效率更高的算法？

6. 对三维空间中的 n 个点，设计其最近对问题的算法。推广到 n 维空间是否可行？

7. 求下列集合的凸包，并指出它们的极点（如果存在的话）。

（1）线段。

（2）正方形。

（3）正方形的边界。

（4）直线。

8. 描述一个求解哈密顿回路问题的穷举查找算法。

9. 解释一下如何对排序问题应用穷举查找，并确定这种算法的效率类型。