Chapter 4

分治策略

我们将在这一章中介绍一种问题求解的基本方法，叫作分治策略。分治，顾名思义，就是“分而治之”的意思。分治策略的核心思想是，现将一个比较大的问题拆解成几个小问题，每个小问题都与原问题是相同的。然后再使用每个小问题求解得到的结果去构造得到原来那个大问题的解。

我们之前介绍的归并排序就是利用分治策略设计得到的算法，我们可以这么理解:

1. 分解：归并排序首先将原数组划分成两个子数组。

2. 求解子问题：归并排序将对两个子数组分别进行归并排序。

3. 构造原问题的解：将两个排好序的子数组合并成一个排好序的原数组。

分治策略之所以能够节省算法运行的时间，是因为我们将子问题视作了一个整体，而避免了不必要的重复计算。我们接下来将介绍一些典型的可以应用分治策略来解决的问题。

4.1 再谈二分查找

二分查找背后的思想其实就是分治策略。但是我们前几章介绍的二分查找看上去很难套用到刚才我们介绍的分治策略的框架里，但是不用着急，我们在这一小节中将一步一步的从分治策略的框架开始来推导二分查找算法，以便让你更好的理解二分查找算法是一个分治策略。

我们假设要解决的问题依然是：在一个由小到大排好序的整型数组 $A[1..n]$ 中，寻找是否存在一个元素 *target*。首先我们先搭建一个分治策略的框架:

1. 分解：将原数组从中间分解成为两个子数组。

2. 求解子问题：在两个子数组中分别寻找 *target*。

3. 构造原问题的解：如果两个子数组中均没有 *target*，那么就返回 *False*，否则返回 *True*。

接下来我们将上面的框架写成伪代码，如 BINARY-SEARCH-RECURSIVE 所示。

```
BINARY-SEARCH-RECURSIVE(A, l, r, target)
1: if target < A[l] or target > A[r] then
2:     return False
3: mid = (l + r)/2
4: if A[mid] == target then
5:     return True
6: left = BINARY-SEARCH-RECURSIVE(A, l, mid-1, target)
7: right = BINARY-SEARCH-RECURSIVE(A, mid+1, r, target)
8: return left or right
```

接下来我们证明算法的正确性。首先，算法第 1 行的判断成立时，*target* 一定不存在于数组当中，因为此时 *target* 要么小于数组中的最小的数，要么大于数组中最大的数。否则，我们有 $A[l] \le target \le A[r]$，这时我们考察数组中间的元素 $A[mid]$，如果 $A[mid] == target$，则说明我们找到了 *target*，那么算法将返回 *True*，否则 *target* 可能存在于 $A[l..mid-1]$ 和 $A[mid+1..r]$ 中，我们再递归调用查找算法即可。如果都不存在，说明 *target* 不存在于原数组，否则说明 *target* 存在于原数组中。

接下来我们分析算法的时间复杂度，假设数组元素为 n 时，BINARY-SEARCH-RECURSIVE 的时间复杂度是 $T(n)$，那么我们有递归式：$T(n) = 2T(n/2) + \Theta(1)$。我们尝试着套用一下主定理，此时 $a = b = 2$，$n^{\log_b a} = n$，$f(n) = \Theta(1) = O(n^{\log_b a - \epsilon})$，其中 $\epsilon = 0.5$，于是符合主定理的情况 1，即 $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$。然而 $\Theta(n)$ 是算法时间复杂度的上界，因为如果 $A[mid] == target$ 时，将不会发生递归调用，因此我们需要将 Θ 改为 O，也就是说，算法的时间复杂度是 $O(n)$!

乍一看，这个复杂度不对啊！我们使用二分查找怎么和线性查找的时间复杂度相同了？我们之前分析过二分查找的复杂度是 $O(\lg n)$ 才对啊？是我们之前的分析出错了吗？

其实不是的，我们仔细考察一下 BINARY-SEARCH-RECURSIVE 的第 6 行和第 7 行。当算法执行到第 6 行的时候，说明 $A[mid] \ne target$，也就是说，要么 $A[mid] < target$，要么 $A[mid] > target$。我们接下来就分别讨论这两种情况，首先如果 $A[mid] < target$，由于 $A[mid-1] < A[mid] < target$，那么也就是说此时对于第 6 行的递归调用而言，$A[target] > A[r]$。这时什么意思呢？它的意思是说，原问题在第 6 行递归调用 BINARY-SEARCH-RECURSIVE(A, l, mid-1, target) 的时候，在进入这个函数时，直接会在第 2 行返回 *False*，而根本不发生函数调用。同理，如果 $A[mid] > target$，就说明 $A[mid+1] > A[mid] > target$，那么第 7 行递归调用 BINARY-SEARCH-RECURSIVE(A, mid+1, r, target) 的时候，进入函数后也一定会直接返回 *False* 而根本不发生函数调用。总结下

来就是，递归的子函数调用至多发生一次!

这样一来，我们就需要将递归式修改成为 $T(n) = T(n/2) + \Theta(1)$。这时候 $a = 1, b = 2$, $n^{\log_b a} = 1 = \Theta(1)$，符合主定理的情况 2，于是我们有 $T(n) = \Theta(\lg n)$，又因为当 $target == A[mid]$ 时不发生递归调用，因此 $\lg n$ 应该是个上界，于是就有 $T(n) = O(\lg n)$，这与我们一开始分析的二分查找的时间复杂度终于一致了!

回过头再来看我们之前介绍的二分查找，其实就是将这个递归函数写成了循环的形式罢了，这也就是为什么乍一看之前的二分查找看上去不太像由分治策略设计的算法的原因了，但其实它们是一样的。

接下来我们讨论什么样的问题适用二分查找算法？其实二分查找算法适用的问题比较单一，就是在具备单调性函数中找出符合要求的自变量。例如我们要在一个排好序的数组中找出是否存在 *target*，其实我们就可以将数组视为是一个离散函数 $A(x)$，单调性是指若 $x_1 < x_2$，那么 $A[x_1] \le A[x_2]$，我们要找出满足 $A[x] == target$ 的那个自变量 x。

4.1.1 求开平方

接下来我们要设计一个算法，给定一个数 a，求 $\sqrt{a}$ 是多少？一种解是利用牛顿法，但是这个问题也可以利用二分查找来解决。首先我们知道，当 $x \ge 0$ 时，函数 $f(x) = x^2$ 是一个单调递增函数。那么原问题等价于：寻找一个 $x \in [0, +\infty)$，使得 $x^2 = a$。算法 SQRT-BINARY-SEARCH 描述了这一过程:

```
SQRT-BINARY-SEARCH(a)
1: l = 0, r = a
2: for i = 0 to 100 do
3:     mid = (l + r)/2
4:     if mid × mid ≤ a then
5:         l = mid
6:     else
7:         r = mid
8: return l
```

接下来我们来理解一下该算法的正确性，算法维护的循环不变式是 $l \times l \le a$ 且 $r \times r > a$。初始条件下，$l \times l = 0 \le a$ 成立，且 $r \times r = a^2 > a$ 也成立。算法在进行循环的时候，如果 $mid \times mid \le a$，那么就将 l 设置为 mid，否则将 r 设置为 mid，这就是在维持这个循环不变式。循环的退出条件是循环完 100 次，之所以用这个作为退出条件是因为当不断的对半切分 100 次之后，l 的精度已经非常非常高了，可以用它来近似的表示 $\sqrt{a}$ 了。

然后我们来分析一下算法的时间复杂度，不管输入的 a 是多少，算法都循环 100 次，因此时间复杂度是 $\Theta(1)$。

4.1.2 求中位数

我们接下来考虑这个问题: 给定一个整型数组 $A[1..n]$, 我们由这个数组构造一个新的二维数组 B, 其中 $B[i][j] = |A[i] - A[j]|$, 问 B 数组的中位数是多少?

首先我们先考虑使用蛮力法来解决这个问题, 我们用二重循环求出每一个 $B[i][j]$, 然后对该数组排序, 最后直接取第 $n^2/2$ 个元素即可。该算法的时间复杂度是 $\Theta(n^2 \lg n)$。

接下来我们考虑使用二分查找来解决该问题。首先我们将 A 数组由小到大排序, 那么我们就可以求解下面这个子问题了: 对于给定的一个整数 $C \geq 0$, 有多少个数对 (i, j), 使得 $|A[i] - A[j]| < C$?

我们可以使用如 COUNT 所示的算法来求解这个问题:

COUNT(A, C)

```
1: r = 1, cnt = 0
2: for l = 1 to A.length do
3:     while r ≤ A.length and A[r] − A[l] ≤ C do
4:         r = r + 1
5:     cnt = cnt + (r − l) × 2 − 1
6: return cnt
```

和往常一样, 我们讨论该算法的正确性和时间复杂度。首先, 对于正确性, 我们有对于每一个 l, 找满足 $A[r] - A[l] > C$ 的最小的 r, 那么也就是说从 l 到 $r-1$ 范围内的每一个数减去 l 都会小于等于 C, 那么符合要求的 (l, i) 的数对个数共有 $r - l$ 个。由于我们要计算符合绝对值 $|A[l] - A[r]| \leq C$ 的数对个数, 因此我们要将结果乘以 2, 但是 (l, l) 和交换得到的 (l, l) 是一样的, 所以还要减去 1。算法的时间复杂度是 $\Theta(n)$, 因为数组中的每个元素只会被 l 和 r 各遍历一次。

接下来我们就可以使用二分查找来解决该问题了, 为什么可以使用二分查找来解决该问题呢? 因为该问题可以等价于找一个单调函数符合要求的自变量, 这里单调函数是:

- $f(C)$: 满足$|A[i] - A[j]| \leq C$ 的数对 (i, j) 的个数。

该函数是一个单调增函数, 因为自变量越大, 满足上述绝对值不等式的数对个数越多。于是问题就变成了找一个最小的整数 C, 满足 $f(C) \geq n^2/2$。

于是我们就有如 MEDIAN-BINARY-SEARCH 所示的算法了:

MEDIAN-BINARY-SEARCH(A)

```
1: MERGE-SORT(A)
2: n = A.length
3: l = −1, r = A[n] − A[1]
4: while r − l > 1 do
```

```
5:      mid = (l + r)/2
6:      if COUNT(A, mid) < n^2/2 then
7:          l = mid
8:      else
9:          r = mid
10: return r
```

接下来我们来理解算法的正确性，上面的二分查找算法维持的循环不变式是 $f(l) < n^2/2$，同时 $f(r) \geq n^2/2$。算法的初始状态，$l = -1, r = A[n] - A[1]$，一定满足循环不变式。循环退出的时候，有 $l + 1 == r$，并且 $f(l) < n^2/2$，$f(r) \geq n^2/2$，因此此时 r 是满足不等式的最小值。第 4 行的 while 循环维持了这个循环不变式，因此算法是正确的。

接下来我们计算算法的时间复杂度。首先对 A 数组排序，时间复杂度是 $\Theta(n \lg n)$，第 4 行的循环最多执行 32 次，第 6 行的 COUNT 运行的时间是 $\Theta(n)$，所以总的时间复杂度是 $\Theta(n \lg n) + O(32 \times n) = \Theta(n \lg n)$，这比蛮力法要快。

4.2 快速幂

接下来我们考虑这样一个问题：对于任意一个数 x，和一个指数 $e \in \mathbb{N}$，求 x^e。

4.2.1 蛮力法

我们不难写出一个蛮力算法 POWER:

```
POWER(x, e)
1: result = 1
2: for i = 1 to e do
3:     result = result × x
4: return result
```

接下来我们来简单的理解一下该算法的正确性: 算法维持的循环不变式是当循环进行一轮后，*result* 的值应该为 x^i，初始条件是 $x^0 = 1$，算法终止时得到 x^e，因此这个算法是正确的。算法的时间复杂度是 $\Theta(e)$，因为第 2 行的循环一共循环了 e 次。

4.2.2 分治策略

接下来我们将尝试使用分治策略来计算 x^e，首先我们给出如 POWER-DIVIDE-CONQUER 所示的算法，然后来讨论它的计算复杂性。

```
POWER-DIVIDE-CONQUER(x, e)
```

```
1: if e == 0 then
2:     return 1
3: p = POWER-DIVIDE-CONQUER(x, e/2)
4: if e % 2 == 1 then
5:     return p × p × x
6: else
7:     return p × p
```

首先我们先来证明算法的正确性，这是一个递归算法，所以我们要考虑利用强归纳公理来证明。首先第一步，确定谓词 $P(n)$:

- $P(n)$: 算法 POWER-DIVIDE-CONQUER 能够正确计算 x^n。

第二步, 证明基本情况 $P(0)$ 成立: 算法的第 1 行的判断成立时, $n = 0$, 此时 $x^0 = 1$, 因此算法返回 1 是正确的,得证。第三步,证明一般情况 $P(0) \wedge P(1) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$ 成立,我们需要将指数分奇偶来讨论, 当指数 $n+1$ 是奇数的时候, $2 \times \lfloor (n+1)/2 \rfloor + 1 = n+1$, 由归纳假设我们知道 $P(\lfloor (n+1)/2 \rfloor)$ 是正确的,令 $p = x^{\lfloor (n+1)/2 \rfloor}$,最终的解为 $p \times p \times 1 = x^{2 \times \lfloor (n+1)/2 \rfloor + 1} = x^{n+1}$, 是正确的。类似的, 当指数 $n+1$ 是偶数的时候, $(n+1)/2 \times 2 = n+1$ 由归纳假设, $P((n+1)/2)$ 成立，令 $p = x^{(n+1)/2}$，最终的解为 $p \times p = p^2 = x^{n+1}$，也是正确的，因此整个命题得证。

接下来我们分析算法的时间复杂度，假设算法的时间复杂度是 $T(e)$，那么我们有递归式:

$$T(e) = T(e/2) + \Theta(1)$$

此时 $a = 1, b = 2$，于是 $e^{\log_b a} = 1 = \Theta(1) = f(e)$，于是符合主定理的第二种情况，即 $T(e) = \Theta(e^{\log_b a} \lg e) = \Theta(\lg e)$，这比蛮力法要快! 我们将该算法称作快速幂。

接下来我们来讨论下为什么分治法计算幂运算比蛮力法要快？不失一般性地我们假设指数 e 是偶数，那么我们有 $x^e = x^{e/2} \times x^{e/2}$，假设我们不递归的去计算 $x^{e/2}$，而用蛮力法去计算它，那么两个 $x^{e/2}$ 将分别进行 $e/2$ 次运算，乍一看总的运算次数是 $e+1$ 次，比蛮力法还要多一次，但仔细观察不难发现，虽然总的结果要用到两个 $x^{e/2}$，但是我们实际上只用计算一次 $x^{e/2}$，然后复用该结果就行了，这样一来，计算量就变成了 $e/2+1$ 次，比蛮力算法少了一半!

然而这只是我们往下 “削减” 了一层的开销，而分治策略会递归的再用这种方式去计算 $x^{e/2}$，也就是说，计算 $x^{e/2}$ 也不必使用 $e/2$ 次运算来完成，而是使用比它更小的运算次数来完成，所以综合下来，我们的总的计算开销还要再进一步缩小，这就是快速幂算法要比蛮力算法快的原因了。

4.2.3 计算第 n 个斐波那契数

我们接下来来讨论计算第 n 个斐波那契数的算法。首先我们定义斐波那契数列: 第 0 个数是 0，第 1 个数是 1，往后的数都是前两个数的和。于是我们可以得到一个递推

公式: $f(n) = f(n-1) + f(n-2)$。我们希望设计一个算法，来求得 $f(n)$ 是多少?

通项公式与递归求解

我们不难看出斐波那契数的递推公式 $f(n) = f(n-1) + f(n-2)$ 是一个齐次线性递推关系。我们在离散数学课程中学习过如何求解一个齐次线性递推关系的通项公式，结果是:

$$f(n) = \frac{1}{\sqrt{5}}[\phi^n - (1-\phi)^n]$$

其中 $\phi = \frac{1+\sqrt{5}}{2}$。

虽然我们有斐波那契数的通项公式，但是我们却不能用它来以 $\Theta(1)$ 的时间复杂度来计算第 n 个斐波那契数，原因是计算机的浮点数精度有限，使得我们不能用计算机来准确表示一个无理数，因此强行使用通项公式来计算斐波那契数会得到错误的结果。但是我们可以使用如 FIB 所示的递归函数来计算第 n 个斐波那契数。

FIB(n)

1: if $n \leq 1$ then
2: return n
3: return FIB$(n-1)$ + FIB$(n-2)$

该算法的正确性可以通过强归纳公理来证明。我们重点来分析该算法的时间复杂度，假设该算法的时间复杂度是 $T(n)$，我们有如下的递归式:

$$T(n) = T(n-1) + T(n-2) + 1 \geq T(n-1) + T(n-2) = f(n)$$

也就是说，使用递归函数来计算斐波那契数的时间复杂度是指数阶，我们在第2章中讨论过，指数阶的时间复杂度运行速度非常慢，因此我们要考虑设计一种更加快速的算法。

递推求解

使用递归的方式求解斐波那契数之所以很慢是因为有很多重复计算，比如我们在计算 FIB(n-1) 的过程中计算了 FIB(n-2)，然而在 FIB 函数的第 3 行中的返回语句中又重复计算了 FIB(n-2)，以此类推还重复计算了 FIB(n-3)，FIB(n-4)，$\cdots$。为了我们可以通过备忘录的方式来消除重复计算，我们也可以使用递推的方式来消除重复计算。递推求解斐波那契数的算法如 FIB-ITER 所示:

FIB-ITER(n)

1: if $n \leq 1$ then
2: return n
3: $a = 0, b = 1, c = 1$

```
4: for i = 3 to n do
5:     a = b
6:     b = c
7:     c = a + b
8: return c
```

该算法的正确性可以通过数学归纳法来证明，这里我们不详细展开了。我们重点来讨论该算法的时间复杂度，由于该算法的第 4 行循环了 $\Theta(n)$ 次，所以该算法的时间复杂度是 $\Theta(n)$，这比递归算法要快不少!

基于快速幂算法求解

那么有没有比线性时间复杂度还快的算法呢? 其实是有的，事实上任何一个齐次线性递推关系的第 n 项都能够利用相似的办法得到。我们可以将斐波那契数的递推公式改写为如下的形式:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

如果我们展开等号左边的向量，又能够得到:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix} = \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

以此类推，我们最终可以得到:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

也就是说，我们将斐波那契数的第 n 项求解写成了矩阵的幂的形式! 由于两个 2×2 的矩阵相乘要做 8 次乘法，如果我们使用快速幂算法来求解这里的矩阵的幂，时间复杂度仅有 $\Theta(8 \times \lg n) = \Theta(\lg n)$，这比递推求解还要更快! 该算法如 FIB-POW 所示。

FIB-POW(n)

```
1: if n ≤ 1 then
2:     return n
3: x[1][1] = x[1][2] = x[2][1] = 1
4: x[2][2] = 0
5: p = MATRIX-POW(x, n − 1)
6: return p[1][1]
```

MATRIX-POW(x, n)

```
1: I[1][1] = I[2][2] = 1
2: I[1][2] = I[2][1] = 0
3: if n == 0 then
4:     return I
5: p = MATRIX-POW(x, n/2)
6: p = MATRIX-MULTIPLY(p, p)
7: if n % 2 == 1 then
8:     return MATRIX-MULTIPLY(p, x)
9: else
10:    return p
```

MATRIX-MULTIPLY(x, y)

```
1: for i = 1 to 2 do
2:     for j = 1 to 2 do
3:         for k = 1 to 2 do
4:             z[i][j] = z[i][j] + x[i][k] × y[k][j]
5: return z
```

（此处再举几个齐次线性递推关系的应用快速幂求解例子）

4.3 快速排序

在这一节里我们首先先给出一个被成为快速排序的算法，该算法利用 PARTITION 操作首先选定一个主元 (pivot)，将给定的一个没有排过顺序的数组分成两个部分，其中位于主元左边的元素均小于或等于主元，且位于右边的部分均大于主元。然后快速排序算法再递归的在左右两个部分上运行，最终得到一个由小到大排好序的数组。

快速排序算法和归并排序算法的区别有以下几点：首先快速排序算法的最优情况与归并排序算法相同，都是 $\Theta(n \lg n)$，但是快速排序算法在最差情况下的时间复杂度是 $\Theta(n^2)$，除此之外，快速排序算法是一个原地排序算法，它不像归并排序那样要拷贝数组元素。

4.3.1 基本的快速排序算法

首先我们给出 PARTITION 操作，该操作接受三个参数，分别是数组和要分割的左右边界。该操作将返回一个数组下标 p，并调整数组当中的元素，使得 $\forall i < p, A[i] \leq A[p]$，且 $\forall i > p, A[i] > A[p]$。

PARTITION(A, l, r)

1: $p = l$
2: for $i = l$ to $r - 1$ do
3: if $A[i] \leq A[r]$ then
4: $t = A[i], A[i] = A[p], A[p] = t$
5: $p = p + 1$
6: $t = A[r], A[r] = A[p], A[p] = t$
7: return p

接下来我们先理解 PARTITION 的正确性。该操作在初始状态下，选择数组的最后一个元素作为主元。在循环过程中维持的循环不变式是：$\forall i \in [l, p), A[i] \leq A[r]$。这个循环不变式的初始状态是将 p 的值设置为 l，此时循环不变式是满足的。随着循环的进行，一旦找到一个数字是小于等于主元的，那么就将它与 p 指向的元素交换，然后将 p 指针往后移动一位，从而维持了循环不变式。当循环退出时，p 指向的元素是大于主元的第一个元素，此时 p 左边的元素都小于或等于它，我们将 p 与主元交换，就满足了主元左边的元素都小于等于主元，而主元右边的元素都大于主元。该算法的时间复杂度是 $\Theta(n)$，其中 n 是要分割的数字的数量。

接下来我们就可以利用 PARTITION 操作来构造快速排序算法了，如 QUICK-SORT 所示，算法接受一个待排序的数组和要排序的左右边界，也就是说如果要对整个数组排序，就需要调用 QUICK-SORT$(A, 1, A.length)$。

QUICK-SORT(A, l, r)
1: if $r - l + 1 \leq 1$ then return
2: p = PARTITION(A, l, r)
3: QUICK-SORT$(A, l, p - 1)$
4: QUICK-SORT$(A, p + 1, r)$

接下来我们先证明算法 QUICK-SORT 的正确性：我们将利用强归纳公理来证明它。首先第一步，我们需要确定谓词 $P(n)$:

- $P(n)$: 算法 QUICK-SORT 能够将一个有 n 个元素的数组原地排好顺序。

第二步是证明基本情况，即 $P(0)$，我们也可以多证明一步 $P(1)$，对于数组元素小于等于 1 的情况，算法的第 1 行将直接返回，因为这时数组是排好序的，得证。第三步我们证明基本情况，即 $P(0) \wedge P(1) \wedge P(2) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$，对于任意一个长度为 $n+1$ 的数组，我们的 PARTITION 操作会将数组分割成两个小于等于 n 的子数组，其中一个子数组里的元素均小于等于主元，而另一个子数组里的元素均大于主元。由归纳假设，QUICK-SORT 算法能够将这两个数组由小到大原地排好顺序，因此在对两个子数组排好顺序之后，整个数组就被排好顺序了，原因是主元左边的元素均小于等于主元，且是有序的，同时主元右边的元素均大于主元，且也是有序的。综上所述，命题得证。

接下来我们分析快速排序的时间复杂度。假设算法的时间复杂度是 $T(n)$，同时假设 PARTITION 操作将数组分成 p 和 $n-p-1$ 两个部分，于是我们有如下所示的递归式:

$$\begin{aligned} T(n) &= T(p) + T(n-1-p) + \Theta(n) \\ &\leq T(p) + T(n-p) + \Theta(n) \end{aligned}$$

如果我们画出该递归式展开的递归树，我们就能够发现，如果 $p = n - p = n/2$，那么递归树的深度是最浅的，否则该递归树是不平衡的，深度也一定会更深。所以如果快速排序算法的 PARTITION 操作能够将数组等分成两份，那么我们有:

$$T(n) = 2T(n/2) + \Theta(n)$$

由主定理，这时候 $T(n) = \Theta(n \lg n)$。

接下来我们考虑递归树最深的情况，即如果算法输入的是按倒序排好顺序的数组，那么每次 PARTITION 操作都会将该数组分割成为长度为 0 和 $n-1$ 的子数组，那么此时递归式就是:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-2) + \Theta(n-1) + \Theta(n) \\ &= \cdots \\ &= \sum_{i=1}^{n} \Theta(i) \\ &= \Theta(\sum_{i=1}^{n} i) \\ &= \Theta(n^2) \end{aligned}$$

于是最坏情况下，快速排序的时间复杂度是 $\Theta(n^2)$。

4.3.2 随机化的快速排序算法

我们接下来将讨论随机化的快速排序算法，原因是我们虽然不能控制输入的数据的形式，但是我们可以通过将数组元素位置随机化的方式来降低 PARTITION 操作将数组分割成长度 0 和 $n-1$ 的概率。在随机化的快速排序算法中，我们仅将 PARTITION 操作改为 RANDOMIZED-PARTITION 操作即可，其主要思想是，我们不再将子数组的最后一个元素作为主元，而是随机选定一个元素作为主元，具体的操作如 RANDOMIZED-PARTITION 所示，算法在一开始随机选择一个元素，并与子数组的最后一个元素交换，这样就达到了随机选择主元的目的。

RANDOMIZED-PARTITION(A, l, r)

1: x = RAND(l, r)

2: $t = A[x], A[x] = A[r], A[r] = t$
3: $p = l$
4: for $i = l$ to $r - 1$ do
5: if $A[i] \le A[r]$ then
6: $t = A[i], A[i] = A[p], A[p] = t$
7: $p = p + 1$
8: $t = A[r], A[r] = A[p], A[p] = t$
9: return p

然后我们将 QUICK-SORT 当中的 PARTITION 改成 RANDOMIZED-PARTITION 即可。

接下来我们分析算法的时间复杂度。为了方便我们的分析，不妨假设数组中的每个元素都不相同，此时我们假设 RAND 函数选取 $[l, r]$ 之间每个数的概率都是 $1/(r-l+1)$（即均匀分布），那么算法将数组元素分割成长度为 0 和 $n-1$ 的子数组的概率就是 $2/(r-l+1)$，当数组元素很多的时候，这是一个小概率事件（然而当数组元素很少的时候，就算 $\Theta(n^2)$ 的时间复杂度也不会花太长时间）。

但是反过来想，选定一个主元使得 PARTITION 正好对半分，使得快速排序算法能够达到最优情况的概率仅有 $1/(r-l+1)$，也是一个小概率事件，甚至比划分成长度为 0 和 $n-1$ 的子数组的概率还要小！所以这是不是就说明了我们将快速排序算法随机化之后，依然无法得到更快的运行速度呢?

答案当然不是！虽然 PARTITION 将以小概率事件取到最优的情况，但是哪怕划分不那么优，快速排序的效果也还不错！我们举个例子，假设 PARTITION 总是将数组划分得很不均匀，例如总是划分成 1/10 和 9/10 的两部分，那么我们有如下递归式:

$$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + \Theta(n)$$

如果我们画出递归树，我们可以发现快速排序的时间复杂度仍然是 $\Theta(n \lg n)$！这也就是说，即使划分得很不均匀，最终的结果也还不错。所以虽然随机化的快速排序算法以小概率取到最优情况，但是会以大概率取到非最坏的情况，即使非最坏的情况划分的不均匀，总的时间复杂度也依然是不错的!

那么严格的说，随机化的快速排序算法的期望时间复杂度是多少呢? 为了探究这个问题，我们首先先看快速排序算法，当快速排序算法选定一个元素作为主元后，这个元素将不再参与后续的快速排序算法的过程了，因此总的来说，快速排序算法第 2 行的 RANDOMIZED-PARTITION 会最多选取 n 次主元。接下来每次 RANDOMIZED-PARTITION 会在第 4 行的循环中的每一轮做一次比较，然后进行常数时间的操作。我们记总的比较操作次数为 X，所以快速排序算法需要进行 $O(n + X)$ 次操作。

接下来问题就变成了求 X 和 n 之间的关系了，为了数出一共进行了多少次比较，我们可以记每次递归调用传进来的数组为 $Z = z_1, z_2, z_3, \cdots, z_n$ 的一个排列，其中 z_k 是这个

数组中第 k 小的元素，我们接下来要数出快速排序中任意两个元素的比较次数，首先，我们注意到，任意一个元素最多只会与其它元素比较一次，因为当这个元素被选为主元之后，它在与子数组每个元素比较完了以后，就不再参与后续的快速排序的过程了，所以设 X_{ij} 为 z_i 和 z_j 的比较次数，那么 $X_{ij} \in \{0, 1\}$，于是我们有:

$$E[X] = E[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

其中

$$E[X_{ij}] = 0 \times P(z_i与z_j不发生比较) + 1 \times P(z_i与z_j发生比较) = P(z_i与z_j发生比较)$$

于是我们有:

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(z_i与z_j发生比较)$$

接下来问题就变成了对于任意两个数 z_i 和 z_j，它俩发生比较的概率是多少？对于这个问题，我们要考虑集合 $Z_{ij} = \{z_i, z_{i+1}, z_{i+2}, \cdots, z_j\}$，我们可以知道，如果 z_i 和 z_j 要发生比较，那么要么 z_i 要在集合 Z_{ij} 中先被选作主元，要么 z_j 要在集合 Z_{ij} 中先被选作主元。否则，一旦 Z_{ij} 中的其它元素被选作了主元以后，z_i 和 z_j 就会被分到两个子数组中，未来永远也不可能发生比较了！所以，z_i 和 z_j 发生比较的概率就等于 z_i 或者 z_j 比 Z_{ij} 中其它元素先选为主元的概率!

接下来我们计算这个概率，首先我们计算 Z_{ij} 中先选 z_i 或者 z_j 的排列数，共有 $(j-i)! + (j-i)!$ 个，Z_{ij} 的全排列共有 $(j-i+1)!$ 个，于是 z_i 或者 z_j 在 Z_{ij} 中先被选作主元的概率为 $2 \times (j-i)!/(j-i+1)! = 2/(j-i+1)$。然后我们再把剩下的元素任意的插入进来，无论怎么样都满足 z_i 或者 z_j 在 Z_{ij} 中先被选作主元，于是 z_i 或者 z_j 比 Z_{ij} 中其它元素先选为主元的概率为 $1 \times 2/(j-i+1) = 2/(j-i+1)$。代入刚才的结果中，我们有:

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1}\frac{2}{k} \\ &\le \sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k} \le \sum_{i=1}^{n-1}(1 + \int_{k=1}^{n}\frac{2}{k}dk) = (n-1) + \sum_{i=1}^{n-1} 2\ln n \\ &\le 3(n-1)\ln n = \Theta(n\lg n) \end{aligned}$$

于是有 $E[T(n)] = E[O(n+X)] = O(n + E[X]) = O(n\lg n)$。又因为最好情况下，$T(n) = \Theta(n\lg n)$，于是我们有平均情况下，随机化的快速排序算法的时间复杂度为 $\Theta(n\lg n)$。

接下来我们来建立一种直觉，随机化的快速排序算法的期望时间复杂度为 $\Theta(n\lg n)$，也就是说，虽然 RANDOMIZED-PARTITION 算法有的时候会选择数组的偏左的元素作为主元，有的时候会选择偏右的元素作为主元，但是我们把所有位置平均下来，它选择的是中间的位置作为主元。因此平均情况下，随机化的快速排序算法能够得到最优情况下的时间复杂度。

4.4 选择无序数组中的第 k 小问题

本节讨论的问题是在一个没有排好序的数组中，选出数组中第 k 小的元素。一种蛮力算法是直接将数组排序，然后取出第 k 个元素即可，假设我们使用快速排序算法，那么它的期望时间复杂度是 $\Theta(n \lg n)$，我们接下来将讨论是否存在更快速的算法。

4.4.1 分治法

我们接下来讨论这样一种分治法，首先算法调用 PARTITION 函数将数组分为左右两个部分并返回主元的位置 p，此时如果主元的位置 $p = k$，那么我们就将主元返回。否则如果 $k < p$，说明我们要找的第 k 小的元素在数组的左半边，于是我们递归的求解数组左半边的第 k 小的元素。否则有 $k > p$ 说明我们要找的第 k 小的元素在数组的右半边，那么我们就递归求解数组右半边的第 $k - p$ 小的元素。该算法如 SELECT 所示。

SELECT(A, l, r, k)

1: if $k > r - l + 1$ then return -1
2: p = PARTITION(A, l, r)
3: if $p == k$ then return $A[p]$
4: else if $k < p$ then return SELECT$(A, l, p - 1, k)$
5: else return SELECT$(A, p + 1, r, k - p)$

该算法的正确性可以由强归纳公理来证明得到，我们重点是考虑该算法的时间复杂度。该算法将数组分为长度为 $p - l$ 和 $r - p$ 的两个子数组，为了得到算法运行的上界，我们假设我们要找的元素总是在更长的子数组当中，于是有递归式:

$$T(n) \leq T(\max(p - 1, n - p)) + \Theta(n)$$

我们希望 $\max(p - 1, n - p)$ 越小越好，于是最优的情况是 $\max(p - 1, n - p) = n/2$，即从中间分成两半，此时我们有

$$T(n) \leq T(n/2) + \Theta(n) = \Theta(n)$$

因此，该选择算法在最优情况下的时间复杂度是 $O(n)$。

接下来我们考虑最坏的情况，与快速排序算法相类似的，PARTITION 操作总是将数组划分为 0 和 $n - 1$ 两个部分，同时我们要找的元素总是在较长的那个子数组中，于是我们有:

$$T(n) \leq T(n - 1) + \Theta(n) = \Theta(n^2)$$

所以该算法在最坏的情况下的时间复杂度是 $O(n^2)$，这比先排序再直接取第 k 个元素还慢一些。

4.4.2 随机化的选择算法

与快速排序相类似的,我们也可以将 PARTITION 随机化,也就是使用 RANDOMIZED-PARTITION，来看一看平均情况下的选择算法的时间复杂度是不是有所下降。为了分析随机化的选择算法，我们首先假设算法的运行时间 $T(n)$ 是一个随机变量。此时我们再设一个随机变量 X_i 表示 RANDOMIZED-PARTITION 将数组分割长度为 i 和 $n-i$ 的两个子数组，于是 $X_i \in 01$，且 $P(X_i) = 1/n$，因为当算法选择第 i 小的元素作为主元时，就将它分割为 i 和 $n-i$ 两个部分了，所以 $P(X_i)$ 就等于选择第 i 小的元素的概率。此时我们的递归式为:

$$T(n) \leq \sum_{i=0}^{n} X_i(T(\max(i-1, n-i)) + \Theta(n))$$

两边同时取期望，有:

$$\begin{aligned}
E[T(n)] &\leq E[\sum_{i=0}^{n} X_i(T(\max(i-1, n-i)) + \Theta(n))] \\
&= \sum_{i=0}^{n} E[X_i](E[T(\max(i-1, n-i)] + \Theta(n)) \\
&= \frac{1}{n}\sum_{i=0}^{n} E[T(\max(i-1, n-i)] + \Theta(n) \\
&= \frac{2}{n}\sum_{k=n/2}^{n-1} E[T(k)] + \Theta(n)
\end{aligned}$$

接下来我们将使用代入法来求解 $E[T(n)]$, 为了使用代入法我们首先需要猜测它的解, 我们在快速排序一小节中建立了直觉，就是平均情况下 RANDOMIZED-PARTITION 函数会将数组平均分为两份，也就是最优情况，即 $O(n)$，那么我们就尝试使用代入法来验证 $E[T(n)] = O(n)$。那么就是说 $\exists n_0 > 0, c > 0, \text{s.t. } \forall n > n_0, E[T(n)] \leq cn$，于是有:

$$\begin{aligned}
E[T(n)] &\leq \frac{2}{n}\sum_{k=n/2}^{n-1} ck + an \\
&= \frac{2c}{n}\frac{(n/2)(n/2+n-1)}{2} + an \\
&= \frac{2c}{n}\frac{3n^2/4 - n/2}{2} + an \\
&= \frac{c}{n}(\frac{3n^2}{4} - \frac{n}{2}) + an \\
&= \frac{3}{4}cn - \frac{c}{2} + an \\
&= cn - (\frac{1}{4}cn + \frac{c}{2} - an)
\end{aligned}$$

$E[T(n)] = O(n)$ 当且仅当:

$$0 \leq cn - (\frac{1}{4}cn + \frac{c}{2} - an \geq 0) \leq cn$$

于是我们取 $c = 4a, n_0 = 1$，不等式成立，因此 $E[T(n)] = O(n)$。于是，随机化的选择算法平均的时间复杂度是 $O(n)$。

4.4.3 BFPRT：时间复杂度为线性时间的选择算法

接下来我们介绍一个称作 BFPRT 的算法，它能够在无论什么情况下均为线性的时间复杂度内找到数组的第 k 小的元素。该算法由 Blum、Floyd、Pratt、Rivest、Tarjan 五位计算机科学家共同提出。算法的框架是:

1. 首先将数组每 5 个元素分为 1 组，共有 $\lceil n/5 \rceil$ 组，对于每一组使用插入排序由小到大的排好顺序。

2. 将每一组的中位数放入一个新的数组当中，递归的调用 BFPRT 算法求取它们的中位数。

3. 将第 2 步取得的中位数的中位数作为主元，调用 PARTITION 函数分为两个数组并返回主元位置 p。

4. 如果 $p = k$，则返回主元作为查找到的元素，否则如果 $k < p$，则在左边的子数组上递归的调用 BFPRT 算法找第 k 小的元素，否则有 $k > p$，则在右边的子数组上递归的调用 BFPRT 算法找第 $k - p$ 小的元素。

该算法的正确性可以由强归纳公理得到，我们这里主要关心算法的时间复杂度。首先对于算法的第 1 步将 $n/5$ 个数组作插入排序，每次插入排序的代价是 25，于是整体的复杂度是 $\Theta(n)$。第 2 步在 $\lceil n/5 \rceil$ 个元素上运行 BFPRT 算法，时间复杂度是 $T(n/5)$。第 3 步在 n 个元素的数组上运行 PARTITION 函数，时间复杂度是 $\Theta(n)$。第 4 步在至多 $7n/10$ 个元素上运行 BFPRT 算法，时间复杂度是 $T(7n/10)$。于是我们有递归式:

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + \Theta(n)$$

我们接下来用代入法证明 $T(n) = O(n)$，也就是说找 $c > 0, n_0 > 0, \forall n > n_0, 0 < T(n) \leq cn$。我们需要使得:

$$\begin{aligned} T(n) &\leq \frac{cn}{5} + \frac{7cn}{10} + an \\ &= \frac{9cn}{10} + an \\ &\leq cn \end{aligned}$$

成立，即 $c \geq 10a$ 即可。于是我们有 $T(n) = O(n)$。

4.5 高斯乘法

我们考虑这样一个问题，两个超过计算机最宽数据类型 (long long, 64 位) 的 n 个十进制位的整数相乘，使用 “列竖式” 的方式，时间复杂度是 $\Theta(n^2)$，我们接下来考虑有没有更快速的方法实现。

为了探究更快速的算法，我们先看一看两个复数相乘:

$$(a+bi)(c+di) = ac - bd + (bc + ad)i$$

其中 $i = \sqrt{-1}$。德国数学家高斯很早就发现，虽然两个复数的乘法看上去涉及 4 次实数的乘法运算，但是实际上可以简化为 3 次，因为

$$bc + ad = (a+b)(c+d) - ac - bd$$

因此我们只需要求 ac、bd 和 $(a+b)(c+d)$，就能用三次乘法来解决复数相乘了。从 4 到 3 的改善看起来微不足道，但是如果我们能利用在递归中，就能节省不少时间!

我们将高斯乘法利用到大数乘法当中，得到 GAUSS-MULTIPLY 算法，该算法接受两个 n 个十进制数位的大整数 x 和 y。其中的加号和减号都是重载了大数的加法和大数减法。其原理是:

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \\ &= 10^n x_L y_L + 10^{n/2}((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R \end{aligned}$$

GAUSS-MULTIPLY(x, y)

1: if n == 1 then return $x \times y$
2: x_L =LEFTMOST(x, n - n/2), x_R =RIGHTMOST(x, n/2)
3: y_L =LEFTMOST(y, n - n/2), y_R =RIGHTMOST(y, n/2)
4: P_1 =GAUSS-MULTIPLY(x_L, y_L)
5: P_2 =GAUSS-MULTIPLY(x_R, y_R)
6: P_3 =GAUSS-MULTIPLY($x_L + x_R$, $y_L + y_R$)
7: return LEFTSHIFT(P_1, $(n/2) \times 2$) + LEFTSHIFT($P_3 - P_1 - P_2, n/2$) + P_2

设该算法的时间复杂度为 $T(n)$，那么有递归式:

$$T(n) = 3T(n/2) + \Theta(n)$$

其中 $a = 3, b = 2, n^{\log_b a} = n^{\log_2 3} \approx n^{1.58}$，于是 $f(n) = O(n^{1.58-\epsilon})$，其中 $\epsilon = 0.5$，因此符合主定理的第 1 种情况，即 $T(n) \approx n^{1.58}$，这比竖式乘法 $\Theta(n^2)$ 要快。

用高斯乘法解决大数乘法问题进一步说明了，如果我们能够找到节省运算的方式，并将它利用在分治策略当中，那么即使节省的运算量是微不足道的，通过递归去不断的积累，总的计算开销也会降低不少。

4.6 快速傅里叶变换

4.6.1 多项式乘法与大数乘法

我们假设有两个多项式，分别是 $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ 和 $B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_nx^n$，我们希望求解得到 $C(x) = A(x) \times B(x)$，问要怎样设计一个尽可能高效的算法?

在正式设计算法之前，我们需要先理解一下，多项式乘法的意义是什么？其实我们可以把多项式乘法与大数乘法联系起来。假设我们有一个 10 进制的数，例如 10813352，我们把它展开来写，就表示为:

$$10813352 = 2 + 5 \times 10 + 3 \times 10^2 + 3 \times 10^3 + 1 \times 10^4 + \cdots + 1 \times 10^7$$

它其实就是多项式的一个实例。在这个实例中，$x = 10$，$a_0 = 2, a_1 = 5, a_2 = 3, \cdots, a_7 = 1$。那么我们考虑另一个数字 32974842，也有:

$$32974842 = 2 + 4 \times 10 + 8 \times 10^2 + 4 \times 10^3 + 7 \times 10^4 + \cdots + 3 \times 10^7$$

类似的，在这个实例中，$x = 10$，$b_0 = 2, b_1 = 4, b_2 = 8, \cdots, b_7 = 3$。如果我们要求 10813352×32974842，那么就相当于求 $C(x) = A(x)B(x)$ 的系数。换一句话说，10 进制数的大数乘法问题就是先求两个多项式 $A(x)$ 和 $B(x)$ 的乘积 $C(x)$ 的各项系数，然后再将 10 作为自变量代入 $C(x)$ 即可，即求 $C(10)$。

4.6.2 蛮力法

最简单的做法就是列竖式求解。例如 $A(x) = 6x^3 + 7x^2 - 10x + 9$，$B(x) = -2x^3 + 4x - 5$，那么 $C(x) = A(x)B(x)$ 的求解过程如图4.1所示。

$$\begin{array}{r}
6x^3 + 7x^2 - 10x + 9 \\
\underline{-2x^3 \qquad + 4x - 5} \\
-30x^3 - 35x^2 + 50x - 45 \\
24x^4 + 28x^3 - 40x^2 + 36x \\
\underline{-12x^6 - 14x^5 + 20x^4 - 18x^3 \qquad\qquad} \\
-12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
\end{array}$$

Figure 4.1: 列竖式求多项式乘法

也就是说，系数 c_j 有如下关系:

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

这种蛮力法求解的时间复杂度是 $\Theta(n^2)$。我们接下来的任务是，找一个更快的算法，求得每个 c_j。

4.6.3 系数表示法与点值表示法

我们线性代数中学过，如果一个线性方程组中线性无关的方程个数与未知数的个数相等时，那么可以唯一确定一个解。在中学，我们把这个性质叫作“有多少个未知数，就要有多少个方程组”。我们将使用这一个性质来解决这个问题。已知 $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$，现在我们有四个点：$(0, 9), (1, 12), (2, 65), (3, 204)$。那么我们代入 $A(x)$，有：

$$\begin{cases} A(0) = a_0 + a_1 \times 0 + a_2 \times 0^2 + a_3 \times 0^3 = 9 \\ A(1) = a_0 + a_1 \times 1 + a_2 \times 1^2 + a_3 \times 1^3 = 12 \\ A(2) = a_0 + a_1 \times 2 + a_2 \times 2^2 + a_3 \times 2^3 = 65 \\ A(3) = a_0 + a_1 \times 3 + a_2 \times 3^2 + a_3 \times 3^3 = 204 \end{cases}$$

上面这个线性方程组有唯一解：$a_0 = 9, a_1 = -10, a_2 = 7, a_3 = 6$，也就是说 $A(x) = 6x^3 + 7x^2 - 10x + 9$。

这说明了我们不光可以用系数 $(9, -10, 7, 6)$ 来表示 $A(x)$，我们还可以用四个点 $(0, 9), (1, 12), (2, 65), (3, 204)$ 来表示 $A(x)$。这种表示我们称为多项式 $A(x)$ 的点值表示，与之对应的使用系数来表示多项式的方式，叫作系数表示。

点值表示法有什么好处呢？我们刚才使用了 4 个点来表示了 $A(x)$，实际上这 8 个点也可以唯一表示 $A(x)$：(0, 9)，(1, 12)，(2, 65)，(3, 204)，(4, 465)，(5, 884)，(6, 1497)，(7, 2340)。同理，如果 $B(x) = -2x^3 + 4x - 5$，我们也可以用 8 个点来表示它：(0, -5)，(1, -3)，(2, -47)，(3, -47)，(4, 117)，(5, 235)，(6, 413)，(7, -663)。接下来我们计算 $C(x) = A(x)B(x)$，我们有：$C(0) = A(0)B(0) = 9 \times (-5) = -45$，即第一个点为 (0, -45)。同理，我们有 (1, -36)，(2, -845)，(3, -9588)，(4,-54405)，(5,-207740)，(6,-618261)，(7,-1551420)。而这 8 个点求解得到的多项式正好是 $C(x) = -12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45$。我们通过上面的分析就可以知道，如果我们分别用 $2n$ 个点来表示两个 n 阶多项式，那么我们就可以用 $\Theta(n)$ 的时间复杂度来求得这两个多项式的乘积的点值表示！之所以要用 $2n$ 个点来表示是因为两个 n 阶多项式的乘积是一个 $2n$ 阶多项式，要表示一个 $2n$ 阶的多项式就需要 $2n$ 个点。使用点值表示法来求多项式的乘积的时间复杂度是 $\Theta(n)$，比使用系数表示法求多项式乘积需要的 $\Theta(n^2)$ 要快，因此我们就可以搭建这样一个算法框架来求解多项式 $A(x)$ 和 $B(x)$ 的乘积：

1. 将 $A(x)$ 和 $B(x)$ 由系数表示法转为点值表示法，即分别取 $2n$ 个不同的点：$(x_0, A(x_0))$，$(x_1, A(x_1))$，$\cdots$，$(x_{2n-1}, A(x_{2n-1}))$ 和 $(x_0, B(x_0))$，$(x_1, B(x_1))$，$\cdots$，$(x_{2n-1}, B(x_{2n-1}))$。
2. 计算点值的乘积：$(x_0, A(x_0)B(x_0))$，$(x_1, A(x_1)B(x_1))$，$\cdots$，$(x_{2n-1}, A(x_{2n-1})B(x_{2n-1}))$。
3. 解方程组求系数 $c_0, c_1, \cdots, c_{2n-1}$。

刚才我们已经分析了，第 2 步计算点值的乘积的时间复杂度是 $\Theta(n)$，所以如果第 1 步和第 3 步能够比 $\Theta(n^2)$ 的时间复杂度要快，那么我们就可以得到一个总的时间复杂度比蛮力法更快的算法来计算两个多项式的乘积。

4.6.4 将系数表示转换为点值表示

接下来我们考察如何将多项式的系数表示转换为点值表示。一种简单的办法像我们刚才讨论的那样，分别取 $x = 0, 1, \cdots, 2n-1$，然后分别求出 $A(x)$ 和 $B(x)$ 的值。但是通过这种方法将系数表示转换为点值表示需要的时间复杂度是 $\Theta(n^2)$，那么我们设计的算法框架的总的时间复杂度也是 $\Theta(n^2)$，并不比蛮力法要快，因此我们需要做出一些改进。

我们首先来观察一下，如果我们取 $x = \pm 1, \pm 2, \cdots \pm n$，会是什么样的？首先我们可以对多项式作这样的变换:

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= a_0 + a_2 x^2 + a_4 x^4 + \cdots + x(a_1 + a_3 x^2 + a_5 x^4 + \cdots) \\ &= A_o(x^2) + x A_e(x^2) \end{aligned}$$

其中 $A_o(x) = a_0 + a_2 x + a_4 x^2 + \cdots$，$A_e(x) = a_1 + a_3 x + a_5 x^2 + \cdots$。此时若 $x = c$，有: $A(c) = A_o(c^2) + cA_e(c^2)$，若 $x = -c$，有 $A(-c) = A_o(c^2) - cA_e(c^2)$。所以我们可以先计算 $A_o(c^2)$ 和 $A_e(c^2)$，然后直接利用这两个数去拼凑 $A(c)$ 和 $A(-c)$。总结下来，我们是使用 $1, 2, 3, \cdots, n$ 这 n 个数得到了 $A(\pm 1), A(\pm 2), \cdots A(\pm n)$ 这 $2n$ 个点，也就是说，计算规模少了一半!

问题接下来就变成了求 $A_o(x^2)$ 和 $A_e(x^2)$，它们是两个 n 阶的多项式。同样的，我们取 $\pm x_1^2, \pm x_2^2, \cdots, \pm x_{n/2}^2$，就又把计算规模减小了一半! 这样递归的计算下去，算法的时间复杂度的递归式就是:

$$T(n) = 2T(n/2) + \Theta(n)$$

由主定理，时间复杂度 $T(n) = \Theta(n \lg n)$，这样一来，我们就找到了比 $\Theta(n^2)$ 更快的算法将系数表示转换为了点值表示。

但是这样有个问题，就是当我们选定一组 $x_1, x_2, \cdot x_n$，使得 $x_1^2, x_2^2, \cdots, x_n^2$ 能够表示为 $\pm x_1^2, \pm x_2^2, \cdots, \pm x_{n/2}^2$ 的话，那么就得有 $-x_1^2 = x_{n/2+1}^2$，也就是说一个数的平方得是一个负数! 这说明了 x_i 光取实数是不够的，还得取复数域中的数才行。

4.6.5 离散傅里叶变换与快速傅里叶变换

那么有没有一组数满足平方以后，折半互为相反数呢？答案是有的。首先我们假设两个 $n/2$ 阶多项式的 $n/2$ 是 2 的幂，即 $\exists m, n/2 = 2^m$，如果它不是 2 的幂，我们就在后面补上 0，补齐成 2 的幂就可以了。然后我们就选择 $\omega^0, \omega^1, \omega^2, \cdots, \omega^{n-1}$，其中 $\omega = e^{2\pi i/n}$，$i = \sqrt{-1}$。

为了理解这 n 个数，我们首先要理解一下什么是 e^{ix}？问题的关键在于当 e 的指数是个实数的时候我们好理解，但是现在 e 的指数是个虚数，那么 e^{ix} 是个啥玩意儿?

为了理解什么是 e^{ix}，我们要利用泰勒展开式:

$$f(x) = f(0) + f'(0)x + \frac{f^{(2)}(0)}{2}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \cdots$$

接下来我们分别对 e^x，$\sin x$ 和 $\cos x$ 作泰勒展开:

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3}x^3 + \cdots$$

$$\sin x = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7}x^7 + \cdots$$

$$\cos x = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots$$

于是有:

$$\begin{aligned} e^{ix} &= 1 + ix + \frac{1}{2!}(ix)^2 + \frac{1}{3!}(ix)^3 + \frac{1}{4!}(ix)^4 + \frac{1}{5!}(ix)^5 + \cdots \\ &= 1 + ix - \frac{1}{2!}x^2 - \frac{i}{3!}x^3 + \frac{1}{4!}x^4 + \frac{i}{5!}x^5 + \cdots \\ &= \cos x + i \sin x \end{aligned}$$

所以其实 $e^{ix} = \cos x + i \sin x$。

那么接下来我们就可以来看我们选定的那一组数了，不妨假设 n = 4，那么我们选定的四个数就是: $1, e^{\pi i/2}, e^{\pi i}, e^{3\pi/2}$，首先我们来验证一下这四个数里能不能构成两组相反数呢？我们就用上面的公式来验证,

$$e^{\pi i} = \cos \pi + i \sin \pi = -1$$

而

$$e^{3\pi/2} = \cos \frac{3\pi}{2} + i \sin \frac{3\pi}{2} = -\cos \frac{\pi}{2} - i \sin \frac{\pi}{2} = -e^{\pi i/2}$$

于是我们选定的四个数可以写成 $\pm 1, \pm e^{\pi i/2}$，接下来我们考虑它们的平方，也就是 1 和 $e^{\pi i} = -1$，也能写成一对相反数的情况，因此我们选定的四个数符合我们一开始提出的要求，这样我们就能用 $\Theta(n \lg n)$ 的时间将多项式的系数表示转换到点值表示了!

点值表示 $A(\omega^0), A(\omega^1), \cdots, A(\omega^{n-1})$ 叫作多项式（或者函数）$A(x)$ 的离散傅里叶变换，我们使用分治策略在 $\Theta(n \lg n)$ 的时间内解出这 n 个数的算法叫作快速傅里叶变换。

4.6.6 将离散傅里叶变换为系数表示

我们已经使用了快速傅里叶变换算法求得了两个多项式的离散傅里叶变换，接下来我们就将 $A(x)$ 和 $B(x)$ 的离散傅里叶变换相乘，就能得到 $C(x)$ 的离散傅里叶变换，也就是 $C(x)$ 的一种点值表示。算法的最后一步是将 $C(x)$ 的点值表示转换回系数表示，要怎么做呢?

我们接着用解线性方程组的思想来考虑这个问题，其实要求 $C(x)$ 的系数表示，就是解下面这个线性方程组:

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} C(1) \\ C(\omega) \\ \vdots \\ C(\omega^{n-1}) \end{bmatrix}$$

即 $M_n(\omega)C = Y$，于是我们有唯一的系数解：$C = M_n(\omega)^{-1}Y$。所以关键是解出 $M_n(\omega)$ 矩阵的逆，即 $M_n(\omega)^{-1}$。在这里我们直接给出结果，我们不去证明它，你可以自己验证一下它的正确性:

$$M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega^{-1})$$

这样一来，求解 $C = M_n(\omega)^{-1}Y$ 的过程仍然可以用快速傅里叶变换算法来完成，无非就是我们将点值视作系数，然后取值 $1, \omega^{-1}, \omega^{-2}, \cdots, \omega^{-(n-1)}$，最后再将计算得到的每个系数除以 n 就可以了。

这样一来，我们就可以总结一下一开始的算法框架了，算法的第一步和第三步，在多项式的系数表示和点值表示直接互相转换的时间复杂度都是 $\Theta(n \lg n)$，而第二步，对点值求乘法的时间复杂度是 $\Theta(n)$, 因此总的时间复杂度是 $\Theta(n \lg n)$, 这比蛮力法的 $\Theta(n^2)$ 的时间复杂度要快。

4.7 Strassen 矩阵乘法

最后我们再来看矩阵相乘的问题，假设有两个 $n \times n$ 的矩阵 A 和 B 相乘得到 C, 蛮力法相乘这两个矩阵的时间复杂度是 $\Theta(n^3)$。Strassen 矩阵乘法是将矩阵划分为四个 $n/2 \times n/2$ 的子矩阵，即:

$$\begin{bmatrix} C_{11}C_{12} \\ C_{21}C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}A_{12} \\ A_{21}A_{22} \end{bmatrix} \begin{bmatrix} B_{11}B_{12} \\ B_{21}B_{22} \end{bmatrix}$$

接下来我们计算:

$$\begin{aligned}
S_1 &= B_{11} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}$$

共需要花费 $\Theta(n^2)$ 的时间。然后我们递归地计算 7 次矩阵乘法:

$$
\begin{aligned}
P_1 &= A_{11} \times S_1 \\
P_2 &= S_2 \times B_{22} \\
P_3 &= S_3 \times B_{11} \\
P_4 &= A_{22} \times S_4 \\
P_5 &= S_5 \times S_6 \\
P_6 &= S_7 \times S_8 \\
P_7 &= S_9 \times S_{10}
\end{aligned}
$$

共需要花费 $7T(n/2)$ 的时间。最后我们计算得到答案:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

也花费 $\Theta(n^2)$ 的时间，于是总的递归式为:

$$
T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.81})
$$

这比蛮力法的 $\Theta(n^3)$ 要快。