实验题目:

问题 1：大数乘法问题。分别尝试计算 9*9， 9999*9999, 9999999999*8888888888 的结果。分析算法性能。

问题 2：线性时间选择问题：

1） 在 4 59 7 23 61 55 46 中找出最大值，第二大值，和第四大的值（要求不允许采用排序算法），并与第 1 题实现的快速排序算法进行比较。

2） 随机生成 10000 个数，要求找出其中第 4999 小的数，并与第 1 题实现的快速排序算法进行比较。

第一个实验（大数乘法问题）

算法问题：

大数乘法问题。分别尝试计算 9*9， 9999*9999, 9999999999*8888888888 的结果。分析算法性能。

算法原理

注意：

在本题中，只介绍相同位数的大整数分治乘法算法，在附录的代码也仅仅粘贴与此相关的代码。假设在本实验实验时，已经提供了写好的加减法和其他相关的代码。

明确问题：

计算大整数的乘法，使用分治法进行解决，要求理论上能够计算任意的位数。

初步分析：

大整数的存储主要使用一个符号位(bool)和一个存储每一位数字的向量(vector)来进行保存，其中在向量的储存中为了便于计算，低位的数字保存在索引较小的地方。

公式和时间复杂度推导：

为了便于说明，两个大整数的位数为 l+r，并分别拆分成长度为 l 的高位数字和长度为 r 的低位数字。即(定义<<为以 10 为基底的左移运算，从而把乘法的运算区分开来)：

$$X = [a(l), b(r)] = a(l) \ll r + b(r)$$
$$Y = [c(l), d(r)] = c(l) \ll r + d(r)$$

则有：

$$X * Y = \big(a(l) \ll r + b(r)\big) * \big(c(l) \ll r + d(r)\big) = \big(a(l) * c(l)\big) \ll 2r + \big(a(l) * d(r) + b(r) * c(l)\big) \ll r + b(r) * d(r)$$

为了便于复杂度的计算，我们约定$l = r = \frac{1}{2}(l + r)$

传统的分治法的复杂度：

传统的分治法需要计算四项乘法$a(l) * c(l), a(l) * d(r), b(r) * c(l), b(r) * d(r)$，时间复杂度传递公式为$T(n) = 4T\left(\frac{n}{2}\right) + \theta(n)$，时间复杂度为$\theta\left(n^{\log 4}\right) = \theta(n^2)$，而一般的使用位乘

法的时间复杂度也是$\theta(n^2)$，因此，要让递归法发挥真正的作用，需要对四项乘法进行特殊的优化。从而降低时间复杂度。

改进的分治法的复杂度：

在改进的式子中，有

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

我们用上式代替，就可以减少一次乘法，从而时间复杂度就优化成了$\theta(n^{\log 3})$。

伪代码：

```
    def multiply_stack(x, y):
        standardlize(x, y)  # 因为传进来的数字可能是不规范的，这个函数将两个大数标准化，从而使其长度一致。
        length := len(x)
        r_length := length / 2
        a, b = cut(x)
        c, d = cut(y)
        ac = multiply_stack(a, c)
        bd = multiply_stack(b, d)
        a_b = a - b
        d_c = d - c
        t = multiply_stack(valof(a_b), valof(d_c))
        f = a_c.s != d_c.s  # 表示(a-c)*(d-c)是否为负数
        m = ac + bd
        if f:
            m = m - t
        else:
            m = m + t
        # 计算到这里时，ac、bd、m 为三个系数。
        ac = ac << 2 * r_length
        m = m << r_length  # 放大一定的倍数
        return ac + bd + m
```

结果：

图 2-1 大数乘法（1）

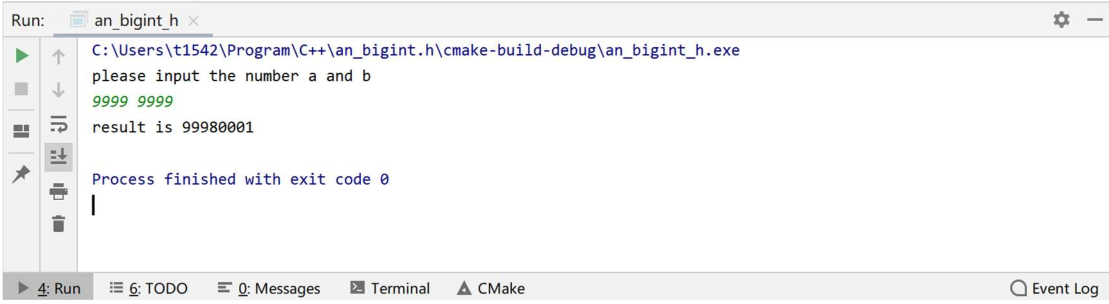

图 2-3 大数乘法（1）

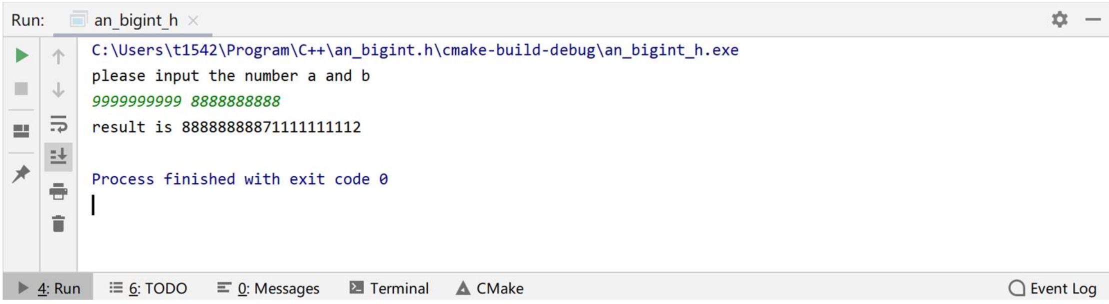

图 2-3 大数乘法（1）

图 2-1 到 2-3 为三个大整数相乘得到的结果，结果正确。

分析：

在实际的编写代码过程中，还是会在很多地方出现或大或小的问题。
第一个问题是使用分治法进行计算时必须对传入的参数进行标准化。假设两个传入的“数字”是 000 和 003，则需要优化成 0 和 3，这样可以大幅度减少不必要的计算。
第二个问题是+0 和-0 的问题，在标准的表示中，0 的符号只能是正的，这样可以减少在程序运行期间可能出现的 bug，而且程序执行期间因为 0 的原因造成了不少的 bug。
第三个问题是进位问题，每一位的数字的范围要远远大于 0~9 的数字表示范围，所以在进行加法计算时，为了提高效率，常常使用先存储，最后进位的策略。乘法的话因为中间步骤进位超多，在进行超大数的乘法很可能某一位累加溢出，因此必须引入溢出判断，将要溢出时必须立即进位，方式溢出导致的数值错误。

第二个实验（线性时间选择）

算法问题：

线性时间选择问题，题目的描述即为在长度为$size$的序列x寻找第i大的元素。

算法原理：

初步分析：

线性选择的关键问题在于某一个元素的定位，回忆之前快速排序中的$partition$的函数，在调用这个函数时，可以确定某一个元素的位置。因为这次的描述时寻找第i大的元素，所以我们在进行分割时，左边元素大，右边元素小。

原理概述：

设原序列为$O = [\ldots]$，代查找的序号为q，进行一轮排序后，将集合分割成三个部分：

$$A = [\ldots](\forall x \epsilon A, x > r), r(i) 和 B = [\ldots](\forall x \epsilon A, x \le r)$$

此时，有以下结论：

$$x\begin{cases} = r, i = q \\ \epsilon A, q < i \\ \epsilon B, q > i \end{cases}$$

若$i = q$，则找到了目标元素，退出递归，否则，在 A 或 B 集合中继续递归调用$partition$，直到找到对应索引的元素。

伪代码：

因为我们在第 1 次实验时已经使用过$partition$函数，在本次实验中，假设该函数是已经写好的。

```
# 在a[l..r)中查找第i 大的元素
def line_search(a, l, r, q):
    i = partition(a, l, r)
    if i = q:
        return a[i]
    else if q < i:  # 表示待查元素在左边
        return line_search(a, l, i, q)
    else:
        return line_search(a, i+1, r, q-i)
```

时间复杂度分析：

在一般情况下，有

$$T(n) = T(cn) + \theta(1)(0 < c < 1)$$

使用主方法，可以算出时间复杂度为$\theta(n)$。

与排序算法（指快速排序算法）进行比较，发现线性选择的算法的复杂度比快速排序算法要低。因为排序算法经过 partition 切割后的左右两个元素都需要进行递归调用，而线性选择可以提前退出或者选择一个集合。

结果：

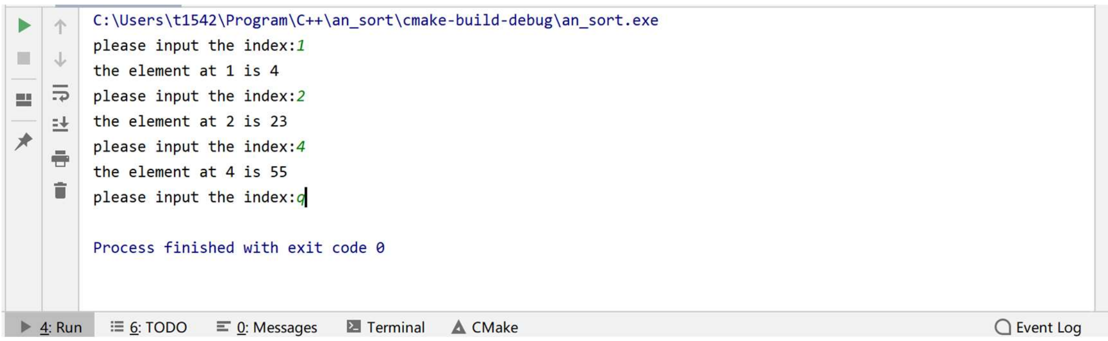

图 2-4 线性时间选择（1）

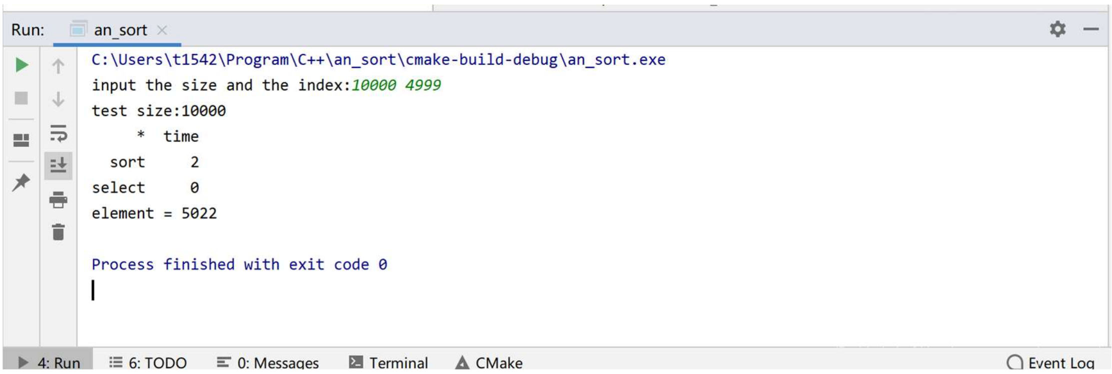

图 2-5 线性时间选择（2）

分析：

从图 2-5 可以看出，排序的时间消耗要比线性时间选择时间长，可以验证两者时间复杂度的关系。

线性时间选择的时间复杂度虽然是线性的，但是在同一个序列中多次选择的时间复杂度可能超过快速排序。

假设序列的长度为n，需要进行k次线性选择。则其时间复杂度为

$$T(n) = k * \theta(n) = \theta(kn)$$

若$k = \Omega(\log n)$时，则此时可能还是直接来一次排序然后找比较好。