

## 实验题目：

**问题 1：**实现 fibonacci 的递归和非递归方法。要求计算 F(100) 的值，比较两种方法的性能。要求 1) 有合适的提示从键盘输入数据；例如“Please input the number to calculate:”  
2) 有输出结果（下同）

**问题 2：**合并排序、快速排序、堆排序、计数排序，至少选两个生成数列比较排序性能。

## 第一个实验(实现 Fibonacci 的递归和非递归的两个版本的算法)：

### 算法问题：

计算 Fibonacci 数列的第 n 项数值 F(n)，比较两种方法的性能。

### 算法原理：

#### 明确问题：

计算 Fibonacci 数列中第 n 项的值，在之后的描述中，使用 F(n) 表示 Fibonacci 中第 n 项的数值，并比较两种方法的性能。

#### 初步分析：

Fibonacci 数列的数值增长近似于指数型增长。因此，F(100) 的值很可能超过 long long 类型的范围。

经计算（python 计算 int 类型长度不受限制）F(100)=354224848179261915075 超过 long long 的范围，因此引入大整数类进行计算。在此例中，因为只需要使用无符号大数的加法，所以设计大数类的代码相对简单。

#### 递归的相关分析：

递归方程为：

$$F(x) = \begin{cases} 1, & x = 1 \text{ 或 } 2 \\ F(x-1) + F(x-2), & x > 2 \end{cases}$$

#### 分析复杂度：

$$T(n) = T(n-1) + T(n-2) + \theta(1) > 2T(n-2) + \theta(1)$$

$$> \dots > 2^{\frac{n}{2}} T(0) + \frac{n}{2} T(1)$$

$$T(n) = \Omega(2^{\frac{n}{2}})$$

上面是最粗略的估计，可以发现，递归的时间复杂度是指数级别的，所以时间复杂度的增加非常恐怖，而  $2^{50}$  是一个非常大的数字，所以 F(100) 很可能计算不出来的，即使有，也没有必要使用这种算法。

实现伪代码：

```
def fibonacci_stack(n):  
    if n <= 2:  
        return 1  
    else:  
        return fibonacci_stack(n-1) + fibonacci_stack(n-2)
```

非递归的相关分析（使用线性算法）：

过程：

其数列的关系仍然遵循递归方法的关系式，观察关系式，发现 $F(n) = F(n-1) + F(n-2)$ ，即 $F(n)$ 与其前两项有关。而且三项的下标是连续的。因此我们需要3个变量（分别为 $x_1, x_2, x_3$ ）存储中间的值（设置三个变量是因为其方程组涉及到3个相关的数列的项）。假设状态 $State(n) = \{x_1 = F(n-2), x_2 = F(n-1), x_3 = F(n)\}$ ，状态 $State(n+1) = \{x_1 = F(n-1), x_2 = F(n), x_3 = F(n+1)\}$ ，在 $State(n) \rightarrow State(n+1)$ 中， $F(n-2)$ 是已经没有用的，所以只要依次执行 $x_1 \leftarrow x_2, x_2 \leftarrow x_3, x_3 \leftarrow x_1 + x_2$ ，便可以转移到下一个状态。

分析复杂度：

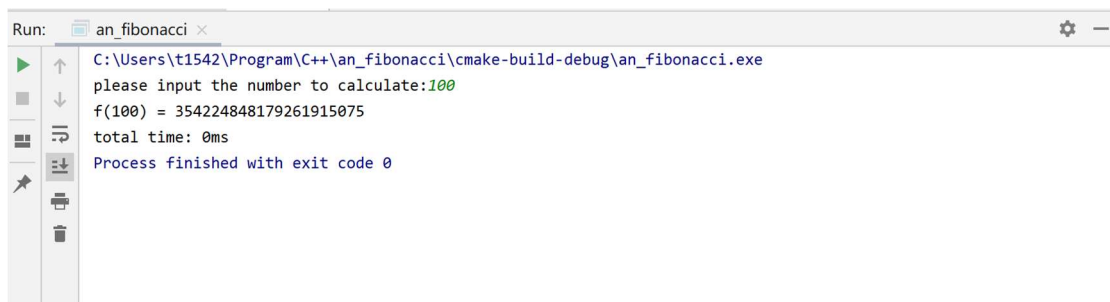
一开始初始化 $State(3)$ ，在依次计算到 $State(n)$ ，每次计算的时间复杂度为 $\theta(1)$ ，需要进行 $n-3$ 次状态变更，所以时间复杂度 $T(n) = n - 3\theta(1) + \theta(1) = \theta(n)$ 。是线性时间复杂的算法。所以效果肯定比递归方法好很多。

伪代码

```
def fibonacci_loop(n):  
    if n <= 2:  
        return 1  
    else:  
        x1 <- 1, x2 <- 1, x3 <- 2  
        for i = 4 -> n  
            x1 <- x2  
            x2 <- x3  
            x3 <- x1 + x2  
        return x3
```

结果：

图 1-1 是非递归版本（循环）版本的运行结果，图 1-2 是递归版本的运行结果。非递归版本很快就出结果了，用时 0ms，而递归版本花了好长时间也没有出结果。



```
Run: an_fibonacci x  
C:\Users\t1542\Program\C++\an_fibonacci\cmake-build-debug\an_fibonacci.exe  
please input the number to calculate:100  
f(100) = 354224848179261915075  
total time: 0ms  
Process finished with exit code 0
```

图 1-1 使用非递归方法计算  $F(100)$

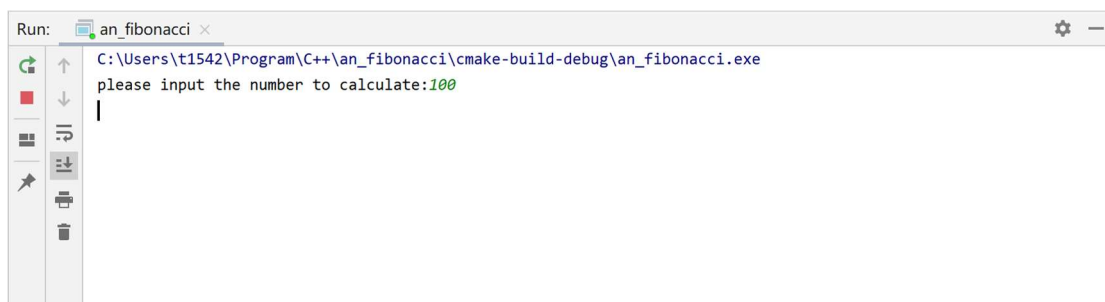


图 1-2 使用递归方法计算  $F(100)$

## 分析

分析两次算法，发现非递归算法比递归算法好很多。经过相关验证得到，递归算法的执行时间确实类似于指数的时间。

递归版本的计算次数比非递归版本次数多很多的原因是递归版本的中间结果只能通过继续递归的方法来实现。而非递归的版本则规避了这一个问题。因为在计算 $F(x)$ 的时候，前面的所有值都已经计算过了。

继续分析递归问题计算 $F(n)$ 过程中计算 $F(n-x)$ 的次数类似于走 $n$ 级楼梯（每次只能走 1 步和 2 步）的走法个数的问题。

值得进步的地方在于一开始需要对程序的执行结果做出一定的估计，例如在此例中，递归方法明显计算 $F(100)$ 是行不通的，如果从理论上能够得出其时间复杂度太高，就没有必要花费额外的时间来设计程序了。

## 第二个实验：比较排序算法的性能

### 算法问题：

实现排序算法，其中归并排序和快速排序选其中一个，计数排序和堆排序选其中一个。要求生成 1 万个数，并比较各个算法的性能，分析算法的性能和算法的时间复杂度。

### 算法原理：

#### 初步分析：

本实现要做的是比较排序的性能问题，除了要对数据进行排序外，还要其他相关的代码支持。其中比较关键的几点是：如何统计代码段的执行时间，如何随机生成指定范围的数，如何确定计数排序相关的数值边界问题。在之后的分析中，全部使用 `int` 型数据进行排序，生成的每一个数据满足 $[0, x)$ ，生成的数据个数为 $size$ 。

如何统计代码段的执行时间？使用`clock()`函数返回结果的差值进行统计。

如何随机生成指定范围的数？使用随机数生成器。

如何确定计数排序相关的数值边界问题？在计数排序中，辅助计数数组的边界为 $size$ ，

在基数排序中，边界为 $\text{ceil}(\log(x - 1))$ 。

快速排序相关分析：

**快速排序**是对冒泡排序的优化，冒泡排序一次只能减少一个逆序数，而快速排序可以一次减少多个逆序数，因此时间复杂度也在 $O(n\log n)$ 的级别。快速排序的核心思想是，确定一个元素的位置，然而按照大小将序列分为左右两部分，左边部分都比中间元素小（或相等），右边部分都比中间元素大。然后分而治之，转化为左右两个小问题，重复上述操作。

实现确定一个元素的位置，并分成左右两个部分的算法：

在这里，我们以例子[5, 2, 6, 9, 1, 4]做例子，分析这个过程。其核心在于如何实现分成左边小右边大的结果呢？我们需要两个工作指针( $pi, pj$ )，将出现在不合适位置的元素放在合适的位置，最快的解决方案，即为 $pi$ 的元素比枢轴值大，然后 $pj$ 的元素比枢轴值小，这样交换一举两得。

首先，我们把 5 当成枢轴值（这个例子使用的第一个数字，当然可以依照实际情况优化随机的元素），并放置在第一个元素的位置。[5, 2, 6, 9, 1, 4]，并初始化 $pi$ 指向第 1 个元素， $pj$ 指向第 $n - 1$ （即最后一个）元素。

然后， $pi$ 从第一个元素开始找起，直到找到第一个比枢轴值大的元素，或者 $pi$ 指针和 $pj$ 重合。此时 $pj$ 从后往前，找到第一个比枢轴值大的元素，或者 $pj < pi$ 停止。在第一步之后。[5, 2, 6, 9, 1, 4]。

如果找到这么一对元素，则两个元素交换，同时 $pi + 1, pj - 1$ 。重复上面的操作。直到 $pj < pi$ 停止操作。[5, 2, 4, 1, 9, 6]。

当退出循环时， $pj$ 将会指向小的部分的最后一个元素，然后和枢轴进行交换，并把这 $pj$ 返回。[1, 2, 4, 5, 9, 6]

总的算法：

先调用一次上述算法（设当前边界为 $[l, r)$ ）（以下称为 $partition$ ），得到中间元素的位置 $m$ ，然后对 $[l, m)$ 和 $[m + 1, r)$ 继续递归调用。

伪代码：

```
def partition(a, l, r):
    if r - l <= 1
        return l
    temp <- a[l]
    pi <- l+1
    pj <- r-1
    while pi < pj
        while pi < pj and a[pi] < temp
            pi++
        while pi < pj and a[pj] > temp
            pj--
        if pi < pj
            swap(a[pi], a[pj])
            pi++
            pj--
    return pj
def quick_sort(a, l, r):
    m <- partition(a, l, r)
```

```
quick_sort(a, l, m)
quick_sort(a, m+1, r)
```

分析复杂度:

*partition*的复杂度为 $\theta(n)$

$$T(n) = T(q) + T(n - q) + \theta(n)$$

最优情况:  $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$ , 使用主方法, 其时间复杂度 $\theta(n \log n)$

一般状况:  $T(n) = T(cn) + T((1 - c)n) + \theta(n), 0 < c < \frac{1}{2}$ , 其时间复杂度 $\theta(n \log n)$

最坏情况:  $T(n) = T(1) + T(n - 1) + \theta(n)$ , 其时间复杂度为 $\theta(n^2)$

计数排序相关分析:

设计思路:

通过统计每个数的位置来进行排序, 目标是建立一个数组 $b[size]$ ,  $b[i]$ 指不大于 $i$ 的元素个数, 这样便能确定每个元素的位置。

首先, 先统计等于 $i$ 的元素个数, 不大于 $i$ 的个数在之前加以变形即可。具体方式为重复将之前的项加到后面的项上。

然后从后向左递归原数列, 设为 $a$ , 将其放在目标数组 $c$ 下标 $b[a[i]] - 1$ 处。

伪代码:

```
def count_sort(a, size, x):
    create array b[x] <- {0}
    create array c[size] no initialize
    for i = 0 -> size - 1:
        b[a[i]] <- b[a[i]] + 1
    for i = 1 -> k - 1:
        b[i] <- b[i] + b[i-1]
    for i = size - 1 -> 0:
        c[b[a[i]]-1] <- a[i]
        b[a[i]] <- b[a[i]] - 1
    copy a <- c
```

时间复杂度分析:

两个 $n$ 次的循环, 一个 $k$ 次的循环。其时间复杂度为 $T(n) = 2O(n) + O(k) = O(\max(n, k))$ 。当 $n \leq k$ 时 $T(n) = O(n)$ , 相对于 $O(n \log n)$ 的比较排序法的下限来说, 计数排序在数字密集的场所更加高效。

## 结果:

图中给出了给范围 $[0, x]$ 的规模为 $size$ 的数组排序(用各种方式)所需要消耗的时间。

```
C:\Users\t1542\Program\C++\an_sort\cmake-build-debug\an_sort.exe
input the range [0,x] and the size:4000 10000
test size:10000
      *random  asc  desc
bubble  834  335  548
select  282  279  273
quick   3   116  323
rx       2    0    2
heap    5    3    3
insert  183    0  311
merge   3    3    3
count   1    0    0
```

图 1-4 排序性能比较-1

```
C:\Users\t1542\Program\C++\an_sort\cmake-build-debug\an_sort.exe
input the range [0,x] and the size:10000000 10000
test size:10000
      *random  asc  desc
bubble  821  393  642
select  300  281  300
quick   2   313  305
rx       2    2    2
heap    5    3    3
insert  144    1  296
merge   3    6    2
count  124  131  129

Process finished with exit code 0
```

图 1-5 排序性能比较-2

## 分析:

从上图发现,快速排序在随机序列中表现很好,而在递增和递减序列中表现不好,这是因为在本程序的快速排序中,枢轴选的是首元素,导致在递增和递减数组排序时每次递归时发生 $T(n) = T(n - 1) + O(n)$ 的情况,导致最终的时间复杂度为 $O(n^2)$ ,为了提高切割的均匀程度,常常使用随机选择和任取三个数取中间数的方法解决。在一般的快速排序中,一般时间复杂度方程为 $T(n) = T(cn) + T((1 - c)n) + O(n)$ ,其中 $c$ 为常数。最后算出其时间复杂度 $O(n \log n)$ ,因此快速排序的时间复杂度和枢轴值选的方法和数据的特征有关。

而计数排序的时间复杂度为 $O(\max(n, k))$ ,如果 $k = O(n)$ ,则其时间复杂度为 $O(n)$ ,图 1-4 和图 1-5 表明,当 $k$ 值过大时,递归的速度也会变慢。