

# 斐波拉契数列计算编程实践

23090032047 计算机类 1 班 于景一

## 1. 实践思路

### 1.1. 核心算法

我们被要求使用递归法，首先了解了递归的基本定义，以及与迭代的区别。

递归和迭代都是一种重复的过程，但是它们有一些区别。以下是 Bing Chat 提供的讲解。

递归是指一个函数或者一个问题可以分解为更小的、相似的子问题，然后通过调用自身来解决子问题，最后得到原问题的解。例如，要计算  $n$  的阶乘，可以先计算  $n-1$  的阶乘，然后再乘以  $n$ ，这就是一个递归的过程。递归通常需要一个终止条件，也就是当问题达到最小规模时，可以直接得到答案，而不需要再次调用自身。例如，当  $n$  等于 1 时， $n$  的阶乘就是 1，不需要再计算。

迭代是指一个函数或者一个问题可以通过重复执行一些简单的操作来逐步接近或达到目标。例如，要计算  $n$  的阶乘，可以从 1 开始，每次乘以下一个数，直到乘到  $n$  为止，这就是一个迭代的过程。迭代通常需要一个初始值和一个更新规则，也就是每次重复时，如何改变变量的值。例如，初始值为 1，更新规则为每次乘以下一个数。

递归和迭代各有优缺点：

递归的优点是代码简洁、易于理解、符合数学逻辑；缺点是占用较多的内存空间、效率较低、可能导致栈溢出。

迭代的优点是占用较少的内存空间、效率较高、不会导致栈溢出；缺点是代码复杂、难以理解、不符合数学逻辑。

注：递归的思想是把一个大问题分解成小问题，直到达到基本情况可以直接求解。递归函数需要有一个终止条件，否则会无限循环。

充分了解后，对于求斐波那契数列问题，我们考虑以下几种思路。

① 定义函数，利用公式进行递归计算。我们用定义一个函数  $f$ ，伪代码描述如下：

```
begin
  def f(n) # n ( $\in \mathbb{Z}^+$ ) is the term we expected to calculate.
    if n = 1 or n = 2
      return 1
    else if n = 0
      return 0
    else
      return f(n - 1) + f(n - 2)
end
```

如果使用这种方法，我们只需输入期望的第  $n$  项，而后在主函数（程序主体）中调用此函数  $f(n)$ ，即可完成运算。

使用递归法求解斐波那契数列问题的时间复杂度网络上众说纷纭，然而不论是  
 $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$  还是  $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$  或是  $O(2^n)$ ，都不可避免地很高。

## ② 尝试其他更快速的方法解决。

我本人缺乏优化算法的水平，只考虑到可以还使用通项公式法、迭代法（动态规划法）进行计算。（注：这些方法的名称是询问 Bing Chat 后，我比照、整理出的，如有纰漏请海涵）

### A. 通项公式法

对进行单次运算的情况，任何计算方法（当然，不计算的方法除外，参见 D.面向结果的编程）在时间复杂度上都无法小于通项公式法，因为它只需进行一次的算术运算，时间复杂度为显然的 $O(1)$ 。在  $n$  的值极大时，这种办法具有显然的优越性。当然，凡事都有优劣，这个公式涉及到无理数和开方运算，所以在实际编程中可能会有精度损失或溢出的问题。

然而，本次实验要求我们输出  $F(1)$  到  $F(50)$  的所有结果，意味着依然要进行 50 次运算，在这种情况下，它的复杂度是与所有 $O(n)$ 复杂度的算法差不多的。

基于高中数学基础，我们知道  $F_n = \frac{1}{\sqrt{5}} [(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$ 。实现如下：

```
begin
  def n, result # n ( $\in \mathbb{Z}^+$ ) is the term we expected to calculate.
  return (1 / sqrt(5)) * (((1 + sqrt(5)) / 2) ^ n) - (((1 - sqrt(5)) / 2) ^ n)
end
```

注 I：在 return 环节使用了颜色对应括号，方便辨认。

注 II：在 C 语言中不存在乘方运算，应使用循环或 pow() 函数进行计算。

### B. 迭代法

在上文中我们已进行过学习，了解了迭代法的基本概念。伪代码实现如下：

```
begin
  def n # n ( $\in \mathbb{Z}^+$ ) is the term we expected to calculate.
  def a = 1, b = 1 # a, b is the first and second term we know.
  for i from 0 to n-1 loop
    a, b => b, a + b
    printf a, b
  return a
end
```

这种方法的时间复杂度是线性的，缺点是需要额外的空间来存储前两项的值。它的时间复杂度为 $O(n)$ 。

实际上，当了解了动态规划的原理，我们可以粗略地认为这属于一种动态规划的思路。动态规划法是什么？研究整理如下：（以下内容由网络提供）

动态规划法一般采用自底向上的方式，从最小的子问题开始求解，逐步扩大到原问题的规模。动态规划法通常需要以下几个步骤：

确定状态：即定义一个或多个变量来描述问题和子问题的状态，以及状态之间的关系。

确定边界：即确定状态变量的取值范围和边界条件，以及初始状态和目标状态。

确定转移方程：即根据状态之间的关系，建立数学模型来描述状态之间如何转移，以及

每次转移所需的代价或收益。

确定求解方式:即根据转移方程,选择合适的数据结构和算法来实现状态的转移和更新,以及最终求解目标状态。

动态规划法是一种非常强大和高效的算法思想,可以用来解决许多在贪心算法或分治算法下无法得到最优解的问题。例如,斐波那契数列、矩阵连乘、背包问题、最长公共子序列、最短路径等都是经典的动态规划问题。

实际上,使用动态规划的思想解决斐波那契数列问题,符合我们历来训练有素的顺向思维,是我处理斐波那契数列问题的第一反应。

### C. 面向结果的编程

显然让程序直接输出结果,不进行运算比什么都快,虽然看起来有点像笑话。下面我们给出直接输出结果的办法。

```
#include<stdio.h>
int main()
{
    printf("F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 ");
    printf("F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 ");
    printf("F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610 ");
    printf("F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 ");
    printf("F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 ");
    printf("F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 ");
    printf("F(30)=832040 ");
    printf("F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=5702887 ");
    printf("F(35)=9227465 ");
    printf("F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 ");
    printf("F(40)=102334155 ");
    printf("F(41)=165580141 F(42)=267914296 F(43)=433494437 ");
    printf("F(44)=701408733 F(45)=1134903170 ");
    printf("F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 ");
    printf("F(49)=7778742049 F(50)=12586269025\n");
    return 0;
}
```

### D. 尝试其他实现

那么,在知识、能力有限的情况下,如何学习其他解决方法呢? 我们将询问 Bing Chat with GPT-4 获得指导。

#### ① 矩阵乘方法

这个函数的思路是:如果  $n$  为 0 或 1,那么斐波那契数列的第  $n$  项就是  $n$  本身;如果  $n$  大于 1,那么我们就用一个二阶矩阵  $A$  来表示斐波那契数列的递推关系,我们只需要计算矩阵  $A$  的  $n-1$  次方,然后取左上角的元素,就可以得到斐波那契数列的第  $n$  项。为了提高效率,我们使用快速幂算法来计算矩阵的幂,即将  $n$  分解为二进制位,然后根据每一位是否为 1 来决定是否乘以  $A$ 。这样,我们就可以将时间复杂度降低到  $O(n \cdot \log n)$ 。

以下是由 GPT-4 模型生成的实现矩阵乘方法的伪代码。（我略微改动了不合理部分）

```
define function fib(n) # define the Fibonacci sequence function
    if n == 0 || n == 1
        return 1
    else
        # define the matrix representation of the Fibonacci sequence
        define matrix A = [[1, 1], [1, 0]]
        # calculate the power of A to the n-1
        define matrix B = matrix_power(A, n - 1)
        # return the top left element of B as the answer
        return B[0][0]

define function matrix_power(A, n) # define the matrix power function
    # define the identity matrix
    define matrix I = [[1, 0], [0, 1]]
    # if n is 0, return the identity matrix
    if n == 0
        return I
    # if n is odd, return A times the power of A to the n-1
    elif n % 2 == 1
        return matrix_multiply(A, matrix_power(A, n - 1))
    # if n is even, return the square of the power of A to the n/2
    else
        return matrix_power(matrix_multiply(A, A), n // 2)

define function matrix_multiply(A, B) # define the matrix multiplication
function
    # define an empty matrix
    define matrix C = [[0, 0], [0, 0]]
    # calculate the matrix multiplication
    for i from 0 to 1 loop # loop over the rows of C
        for j from 0 to 1 loop # loop over the columns of C
            for k from 0 to 1 loop # loop over the elements of A and B
                assign C[i][j] += A[i][k] * B[k][j] # update the element
of C by adding the product of A and B
    # return the result matrix
    return C
```

## ② 幂级数法

利用斐波那契数列的生成函数，将其展开为一个幂级数，然后利用幂级数的性质来计算任意一项的值。幂级数法的优点是可以避免浮点数的精度问题，缺点是需要处理大数的运算问题，而且生成函数和幂级数的推导也比较复杂。它的时间复杂度是 $O(n^2)$

```
define function fib(n) # define the Fibonacci sequence function
```

```

# define the generating function of the Fibonacci sequence
define function g(x) = 1 / (1 - x - x^2)
# expand g(x) into a power series using Taylor's theorem
define function h(x) = sum of (g^(n)(0) / n!) * x^n from n = 0 to
infinity
# return the nth coefficient of h(x) as the answer
return h(x) evaluated at x^n

```

总结：我们提供了递归、通项公式、迭代（动态规划）、面向结果的编程、矩阵乘方、幂级数这六种方法，究竟孰优孰劣我们将在 [3.结果比较](#) 中进行分析比较。

## 1.2. 记录时间

我们在执行程序主体之前就记录一次时间，在执行完毕再记录一次，两者之差就是程序主体运行的时长。根据这种思想，对于各个语言，我们有：

- ① 如需在 C 语言中记录时间，我们需要调用 time.h 头文件里的 time()函数进行记录。基本实现方法如下：

```

begin
    start = time(NULL) # Record time when it started
    .....
    end = time(NULL) # Record time when it ended
    timeProgramRun = end - start # Get what's between them
end

```

由于 time()输出的值为整数，若要统计小数级别的时间差，可以使用处理器时钟记录。

```

begin
    clock_t start_t, finish_t;
    start_t = clock();
    .....
    finish_t = clock();
    printf("Run Time: %fs\n", (double)(finish_t - start_t) /
CLOCKS_PER_SEC);
end

```

- ② 如需在 Golang 中记录时间，我们可以使用 time 库的 Since()函数。

```

import (
    .....
    "time"
)
func void()
    start := time.Now() # Record time when it started
    .....
    elapsed := time.Since(start) # Get what's between started and now

```

- ③ 如需在 Python 中记录时间，我们可以使用 time 库的 Since()函数。

```

import time

```

```
begin
    t = time.time() # Record time when it started
    .....
    print(f'coast:{time.time() - t:.4f}s') # Get what's between started
    and now, accurate to 4 decimal places.
end
```

### 1.3. 实现环境

按照要求，我们应准备至少两个系统。结合个人情况，编程环境如下：

#### ① C 语言环境

Windows 环境下，我采用 VS Code 编辑器，使用较新版本 mingw64 的 gcc 编译。

```
PS C:\Users\JStar> gcc --version
gcc.exe (Rev7, Built by MSYS2 project) 13.1.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

图 1.3.1-1 Windows 下的 gcc 环境

Linux 环境，我将直接通过 ssh 连接到我的云服务器，采用基于 Ubuntu 22.04 LTS 的 vim 编辑器，使用内置的 gcc 编译。

```
JStar0Y@JS-S6:~$ cat /proc/version
Linux version 6.2.0-1014-azure (buildd@lcy02-amd64-030) (x86_64-linux-gnu-gcc-11 (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, GNU ld (GNU Binutils for Ubuntu) 2.38) #14~22.04.1-Ubuntu SMP Wed Sep 13 16:15:26 UTC 2023
```

图 1.3.1-2 Linux 下的 gcc 环境

#### ② Go 语言环境

由于我的云服务器已编译安装 caddy（该 Web Server 是基于 Go 实现的），无需重复配置环境。在此处仅谈配置环境的基本指令。

在 Ubuntu 22.04 内置的 apt 源内，golang 的版本并不足够新（似乎为 1.19 以下，而最新的发行版是 1.21）。

为了解决这个问题，参考一个 golang 的 wiki 文档 (<https://github.com/zchee/golang-wiki/blob/master/Ubuntu.md>)，我们采取如下指令添加 golang 的库。

```
$ sudo apt install software-properties-common -y
$ sudo add-apt-repository ppa:longsleep/golang-backports
$ sudo apt update
$ sudo apt install golang-go -y
```

通过这个指令，我们将配置好如下图的环境。

```
JStar0Y@JS-S6:~$ go version
go version go1.21.2 linux/amd64
JStar0Y@JS-S6:~$
```

图 1.3.2-1 配置 Go 语言环境结果

#### ③ Python 语言环境

由于我不清楚是否已在云服务器上配置 Python，简单检查一下是否存在 python 环境。欲进行这项检查，我们可以简单调用尝试，若不存在此命令，则需要配置。

```
$ python
$ python3
```

当然，也可以使用 `which` 命令查询，若输出了对应的路径，证明 `python` 环境正常；若无输出则不存在，需要配置。

```
$ which python
$ which python3
```

我们在使用 `python3` 命令时发现环境是存在的，而 `pip` 是不存在的，可以手动补齐。

```
$ sudo apt install python3-pip -y
```

```
JStar0Y@JS-S6:~$ python
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
JStar0Y@JS-S6:~$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
JStar0Y@JS-S6:~$ pip
Command 'pip' not found, but can be installed with:
sudo apt install python3-pip
```

图 1.3.2-2 Python 环境的检测

## 2. 编程运行

在本环节，我们将对每一种语言使用递归、通项公式、迭代（动态规划）和“面向结果的编程”四种方法，记录下各语言下各算法的运行结果和运行时间，以便下一步建立列表，进行运行效率（时长）分析比较。

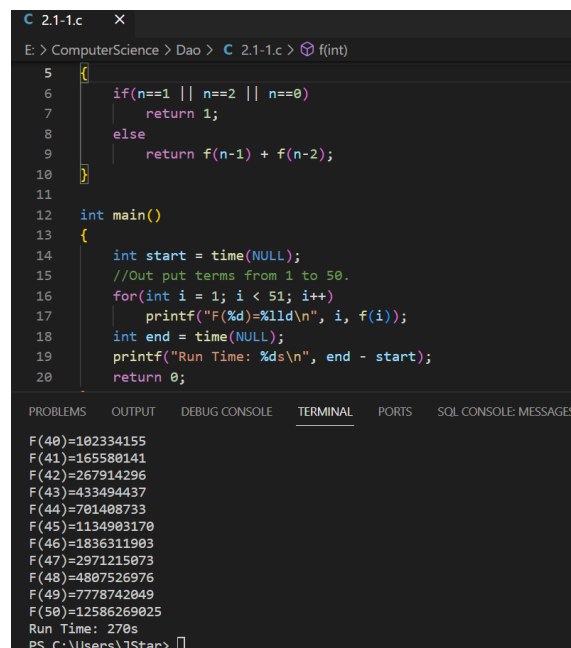
所有编程的源文件已按 `2.*.*` 的格式保存，在每种方法的标题处已做好标注。

将运行时间三次测量取平均值。

### 2.1. 在 Windows 环境使用 C 编程

#### ① 递归法（2.1-1.c）

先编写代码主体部分，检查是否有误；然后加入记录时间的函数。



```
C 2.1-1.c X
E: > ComputerScience > Dao > C 2.1-1.c > f(int)
5 {
6     if(n==1 || n==2 || n==0)
7         return 1;
8     else
9         return f(n-1) + f(n-2);
10 }
11
12 int main()
13 {
14     int start = time(NULL);
15     //Out put terms from 1 to 50.
16     for(int i = 1; i < 51; i++)
17         printf("F(%d)=%lld\n", i, f(i));
18     int end = time(NULL);
19     printf("Run Time: %ds\n", end - start);
20     return 0;
}

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE: MESSAGES
F(40)=102334155
F(41)=165580141
F(42)=267914296
F(43)=433494437
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
F(50)=12586269025
Run Time: 270s
PS C:\Users\JStar>
```

图 2.1-1 使用递归法编写与运行

在本电脑记录运行时间分别为：270s, 256s, 267s（注意：此环境下我在使用电池，在使用 VS Code 编写其他程序文件）

### ② 通项公式法 (2.1-2.c)

```

3
4 int main()
5 {
6     int start = time(NULL);
7     double result = 0;
8     //Out put terms from 1 to 50.
9     for(int i = 1; i <= 51; i++)
10     {
11         result = (1 / sqrt(5)) * ((pow(((1 + sqrt(5)) / 2), i)) - (pow(((1 - sqrt(5)) / 2), i)));
12         printf("F(%d)=%.01f\n", i, result);
13     }
14     int end = time(NULL);
15     printf("Run Time: %ds\n", end - start);
16     return 0;
17 }

```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

SQL CONSOLE

```

F(40)=102334155
F(41)=16558041
F(42)=267914296
F(43)=433404437
F(44)=701488723
F(45)=1134801370
F(46)=1835315903
F(47)=2971215073
F(48)=4807526976
F(49)=7724753840
F(50)=12562662925
Run Time: 0s
PS C:\Users\33star>

```

图 2.1-2 使用通项公式法编写与运行

在本电脑记录运行时间分别为：0s, 0s, 0s（注意：time()的输出为整数值，因此在<1s 的情况下显示为 0s）

### ③ 迭代法 (2.1-3.c)

```
C 21-3.c      X

C> Users > JStar > Desktop > Temps > C 21-3.c > (main)

1  √ #include<stdio.h>
2  #include<time.h>
3
4  √ int main()
5  {
6      int start = time(NULL);
7      long long int a = 1, b = 1, tmp;
8      //Output terms from 1 to 50.
9      printf("F(1)=1\n");
10 √ for(int i = 1; i < 50; i++)
11 {
12     tmp = a;
13     a = b;
14     b = a + tmp;
15     printf("F(%lld)=%lld\n", i + 1, a);
16 }
17 int end = time(NULL);
18 printf("Run Time: %ds\n", end - start);
19 return 0;
20 }
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	<u>TERMINAL</u>	PORTS	SQL CONSOLE
----------	--------	---------------	-----------------	-------	-------------

```
F(40)=102334155
F(41)=165580141
F(42)=267914296
F(43)=433494437
F(44)=701480733
F(45)=1134803170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742849
F(50)=12586269025
Run Time: 0s
PS C:\Users\JStar>
```

图 2.1-3 使用迭代法编写与运行

在本电脑记录运行时间分别为: 0s, 0s, 0s

#### ④ “面向结果的编程” (2.1-4c)

```
C:\21-4c X
C:\Users\JSt> JSt> Desktop > Temps > C 21-4c <> main()

1  int start = time(NULL);
2
3  printf("(F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 )");
4
5  printf("(F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 )");
6
7  printf("(F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610 )");
8
9  printf("(F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 )");
10
11 printf("(F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 )");
12
13 printf("(F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514226 F(30)=832046 )");
14
15 printf("(F(31)=1346353 F(32)=2178309 F(33)=3524556 F(34)=5702881 F(35)=9224565 )");
16
17 printf("(F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=62459084 F(40)=102331545 )");
18
19 printf("(F(41)=16588041 F(42)=267914296 F(43)=433404537 F(44)=701408733 F(45)=1139031704 )");
20
21 printf("(F(46)=1836311593 F(47)=2971215073 F(48)=4807526976 F(49)=777842049 F(50)=12586269825)");
22
23
24 int end = time(NULL);
25
26 printf("Run Time : %d\n", end - start);
27
28 return 0;
29
PROGRAMS CPUPIF EMBASICCONCEPTS TERMINAL PORTS SEQ CONSOLE
PS C:\Users\JSt>
PS C:\Users\JSt> .\code\version10-win\code\cpython3.5-win32\code\debug\debug\main\bin\debug\launcher.exe --stdoutMicroSoftEngine-In-64\js\cvt --stdoutMicroSoftEngine-Out-Ittnewy\lbr --stderrMicroSof
ft.Engine-Error-output\ab --pidMicroSoftEngine-Pid-Singlebr.rmp --dbgExec\c:\msys64\win64\bin\code.exe --int
erpreter=
F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610
F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 F(26)=121393
F(27)=196418 F(28)=317811 F(29)=514226 F(30)=832046 F(31)=1346353 F(32)=2178309 F(33)=3524556 F(34)=5702881 F(35)=9224565 F(36)=14930352
F(37)=24157817 F(38)=39088169 F(39)=62459084 F(40)=102331545 F(41)=16588041 F(42)=267914296 F(43)=433404537 F(44)=701408733 F(45)=1139031704
F(46)=1836311593 F(47)=2971215073 F(48)=4807526976 F(49)=777842049 F(50)=12586269825
Run Time : 0
PS C:\Users\JSt>
```

图 2.1-4 使用“面向结果的编程”编写与运行



在本电脑记录运行时间分别为：0s, 0s, 0s

考虑到后三组实验用时均<1s，我们将在统计时长的时候使用 time()相关的备用算法。

## 2.2. 在 Linux 环境下使用 Golang 编程

### ① 递归法 (2.2-1.go)

```
package main
import (
    "fmt"
    "time"
)

var n int = 50
func fib(n int) int {
    if n <= 2 {
        return 1
    } else {
        return fib(n-1) + fib(n-2)
    }
}

func main() {
    start := time.Now()
    for i := 1; i <= n; i++ {
        fmt.Printf("F(%v)=%v\n", i, fib(i))
    }
    elapsed := time.Since(start)
    fmt.Println("Run time:", elapsed)
}
```

图 2.2-1-1 使用 vim 编写递归法程序代码

我们使用 go run \*.go 执行代码。

```
F(36)=14930352
F(37)=24157817
F(38)=39088169
F(39)=63245986
F(40)=102334155
F(41)=165580141
F(42)=267914296
F(43)=433494437
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
F(50)=12586269025
Run time: 2m28.488229505s
JStar0Y@JS-SG:~/Test4$
```

图 2.2-1-2 运行程序

在我的 Linux 服务器环境下运行时间为： 2m28.49s, 2m28.57s, 2m28.96s.

### ② 通项公式法 (2.2-2.go)

如我们在 1.1.②.A 中所提到的，在 Golang 中使用通项公式求解产生了 $\delta \approx 1$  的误差。由于在 Golang 中使用四舍五入取整特别复杂，在这里我们容忍这个误差。

```
package main
import (
    "fmt"
    "time"
    "math"
)

var n int = 50
var result float64 = 0
func fib(n int) int {
    result = (1 / math.Sqrt(5)) * (math.Pow(((1 + math.Sqrt(5)) / 2), float64(n))) - (math.Pow(((1 - math.Sqrt(5)) / 2), float64(n))))
    return int(result)
}

func main() {
    start := time.Now()
    for i := 1; i <= n; i++ {
        fmt.Printf("F(%v)=%v\n", i, fib(i))
    }
    elapsed := time.Since(start)
    fmt.Println("Run time:", elapsed)
}
```

图 2.2-2-1 使用 vim 编写通项公式法程序代码

```

F(38)=39088169
F(39)=63245986
F(40)=102334154
F(41)=165580140
F(42)=267914295
F(43)=433494437
F(44)=701408732
F(45)=1134903169
F(46)=1836311903
F(47)=2971215072
F(48)=4807526975
F(49)=7778742048
F(50)=12586269024
Run time: 771.778µs
JStar0Y@JS-S6:~/Test4$

```

图 2.2-2-2 运行程序

在我的 Linux 服务器环境下运行时间为： 771.778µs, 733.762µs, 709.272µs.

### ③ 迭代法 (2.2-3.go)

<pre> package main import (     "fmt"     "time" )  func main() {     start := time.Now()     var a, b, tmp int = 1, 1, 0     n := 50     fmt.Printf("F(1)=1\n");     for i := 2; i &lt;= n; i++ {         tmp = a         a = b         b = tmp + b         fmt.Printf("F(%v)=%v\n", i, a)     }     elapsed := time.Since(start)     fmt.Println("Run time:", elapsed) } ~ </pre>	<pre> F(32)=2178309 F(33)=3524578 F(34)=5702887 F(35)=9227465 F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 F(41)=165580141 F(42)=267914296 F(43)=433494437 F(44)=701408733 F(45)=1134903170 F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025 Run time: 703.775µs JStar0Y@JS-S6:~/Test4\$ </pre>
---	---

图 2.2-3-1 使用 vim 编写迭代法程序代码

图 2.2-3-2 运行程序

在我的 Linux 服务器环境下运行时间为： 689.667µs, 696.969µs, 701.77µs.

### ④ “面向结果的编程” (2.2-4.go)

```

package main
import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    fmt.Printf("F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 ")
    fmt.Printf("F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 ")
    fmt.Printf("F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610 ")
    fmt.Printf("F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 ")
    fmt.Printf("F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 ")
    fmt.Printf("F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 F(30)=832040 ")
    fmt.Printf("F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=5702887 F(35)=9227465 ")
    fmt.Printf("F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 ")
    fmt.Printf("F(41)=165580141 F(42)=267914296 F(43)=433494437 F(44)=701408733 F(45)=1134903170 ")
    fmt.Printf("F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025\n")
    elapsed := time.Since(start)
    fmt.Println("Run time:", elapsed)
}
~
~

```

图 2.2-4-1 使用 vim 编写“面向结果的”程序代码

```

JStar0Y@JS-S6:~/Test4$ go run 2.2-4.go
F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 F(11)=89 F
(12)=144 F(13)=233 F(14)=377 F(15)=610 F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 F(21)=10946 F(22)=17711 F(23)=28657 F(24)=
46368 F(25)=75025 F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 F(30)=832040 F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=570
2887 F(35)=9227465 F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 F(41)=165580141 F(42)=267914296 F(43)=43
3494437 F(44)=701408733 F(45)=1134903170 F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025
Run time: 217.819µs
JStar0Y@JS-S6:~/Test4$ go run 2.2-4.go
F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 F(11)=89 F
(12)=144 F(13)=233 F(14)=377 F(15)=610 F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 F(21)=10946 F(22)=17711 F(23)=28657 F(24)=
46368 F(25)=75025 F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 F(30)=832040 F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=570
2887 F(35)=9227465 F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 F(41)=165580141 F(42)=267914296 F(43)=43
3494437 F(44)=701408733 F(45)=1134903170 F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025
Run time: 219.219µs
JStar0Y@JS-S6:~/Test4$

```

图 2.2-4-2 运行程序

在我的 Linux 服务器环境下运行时间为： 219.219 $\mu$ s, 217.819 $\mu$ s, 213.219 $\mu$ s.

## 2.3. 在 Linux 环境下使用 Python 编程

### ① 递归法 (2.3-1.py)

Python 中运行递归法真令人好等! 如何使进程在后台运行 (进程守护), 我选用内置的 screen 实现。基础指令如下:

```
$ screen #新建一个屏幕
$ screen -ls #列出 Detached 的所有屏幕
$ screen -r **(ID) #输入屏幕号以恢复屏幕
```

如此我们便可以放心地断开远程终端, 完成其他任务。

```
import time

def fib( n ):
    if n == 0 or n == 1:
        return n
    else:
        return fib( n - 1 ) + fib( n - 2 )

t = time.time()
for i in range(1, 51):
    print('F(' + str(i) + ')=' + str(fib(i)))
print(f'Time run: {time.time() - t:.4f}s')
~
```

图 2.3-1-1 使用 vim 编写递归法程序代码

我们使用 `python3 ./2.3-1.py` 执行代码。

```
F(32)=2178309
F(33)=3524578
F(34)=5702887
F(35)=9227465
F(36)=14930352
F(37)=24157817
F(38)=39088169
F(39)=63245986
F(40)=102334155
F(41)=165580141
F(42)=267914296
F(43)=433494437
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
```

图 2.3-1-2 运行程序

在我的 Linux 服务器环境下运行时间为: 8h 以上。(在第 9h, 由于运行时间过长, screen 已死掉)

```
JStar0Y@JS-S6:~$ screen -ls
There is a screen on:
  58607.pts-1.JS-S6      (10/13/23 13:19:31)  (Dead ???)
Remove dead screens with 'screen -wipe'.
```

图 2.3-1-3 对应的 screen 已死掉

### ② 通项公式法 (2.3-2.py)

```

import time

def fib( n ):
    return (1 / 5 ** 0.5) * (((1 + 5 ** 0.5) / 2) ** i - ((1 - 5 ** 0.5) / 2) ** i)

t = time.time()
for i in range(1, 51):
    print('F(' + str(i) + ')= ' + str(int(fib(i))))
print('Time run: {time.time() - t:.4f}s')
~
~
~

```

图 2.3-2-1 使用 vim 编写通项公式法程序代码

```

F(30)=832040
F(31)=1346269
F(32)=2178309
F(33)=3524578
F(34)=5702887
F(35)=9227465
F(36)=14930352
F(37)=24157817
F(38)=39088169
F(39)=63245986
F(40)=102334155
F(41)=165580141
F(42)=267914296
F(43)=433494437
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
F(50)=12586269025
Time run: 0.0022s

```

图 2.3-2-2 运行程序

在我的 Linux 服务器环境下运行时间为： 0.0021s, 0.0022s, 0.0021s.

### ③ 迭代法 (2.3-3.py)

```

import time

t = time.time()

a, b, tmp = 1, 1, 0
n = 50
print("F(1)=1")
for i in range(1, 50):
    tmp = a
    a = b
    b = tmp + b
    print("F(" + str(i+1) + ")= " + str(a))

print('Time run: {time.time() - t:.4f}s')
~
~
~

```

图 2.3-3-1 使用 vim 编写迭代法程序代码

```

F(43)=4807526976
F(44)=701408733
F(45)=1134903170
F(46)=1836311903
F(47)=2971215073
F(48)=4807526976
F(49)=7778742049
F(50)=12586269025
Time run: 0.0021s

```

图 2.3-3-2 运行程序

在我的 Linux 服务器环境下运行时间为： 0.0021s, 0.0021s, 0.0021s.

④ “面向结果的编程” (2.3-4.py)

```
import time

t = time.time()

print("F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 ", end="")
print("F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 ", end="")
print("F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610 ", end="")
print("F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 ", end="")
print("F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 ", end="")
print("F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 F(30)=832040 ", end="")
print("F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=5702887 F(35)=9227465 ", end="")
print("F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 ", end="")
print("F(41)=165580141 F(42)=267914296 F(43)=433494437 F(44)=701408733 F(45)=1134903170 ", end="")
print("F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025 ")

print(f"Time run: {time.time() - t:.4f}s")
```

图 2.3-4-1 使用 vim 编写“面向结果的”程序代码

```
JStar0Y@JS-SG:~/Test4$ python3 2.3-4.py
F(1)=1 F(2)=1 F(3)=2 F(4)=3 F(5)=5 F(6)=8 F(7)=13 F(8)=21 F(9)=34 F(10)=55 F(11)=89 F(12)=144 F(13)=233 F(14)=377 F(15)=610 F(16)=987 F(17)=1597 F(18)=2584 F(19)=4181 F(20)=6765 F(21)=10946 F(22)=17711 F(23)=28657 F(24)=46368 F(25)=75025 F(26)=121393 F(27)=196418 F(28)=317811 F(29)=514229 F(30)=832040 F(31)=1346269 F(32)=2178309 F(33)=3524578 F(34)=5702887 F(35)=9227465 F(36)=14930352 F(37)=24157817 F(38)=39088169 F(39)=63245986 F(40)=102334155 F(41)=165580141 F(42)=267914296 F(43)=433494437 F(44)=701408733 F(45)=1134903170 F(46)=1836311903 F(47)=2971215073 F(48)=4807526976 F(49)=7778742049 F(50)=12586269025
Time run: 0.0001s
```

图 2.3-4-2 运行程序

在我的 Linux 服务器环境下运行时间为： 0.0001s, 0.0001s, 0.0001s.

3. 结果比较

为了控制变量，由于 Golang 和 Python 都被要求在我的 Linux 服务器环境下运行，接下来我们将在 Linux 下重新运行并记录 C 语言的各种算法所耗时间。

服务器的具体配置如下：

```
$ uname -a
Linux JS-SG 6.2.0-1014-azure #14~22.04.1-Ubuntu SMP Wed Sep 13 16:15:26
UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
$ cat /proc/cpuinfo | grep model\ name
model name      : Intel(R) Xeon(R) Platinum 8171M CPU @ 2.60GHz
$ cat /proc/meminfo | grep MemTotal
MemTotal:       853972 kB
```

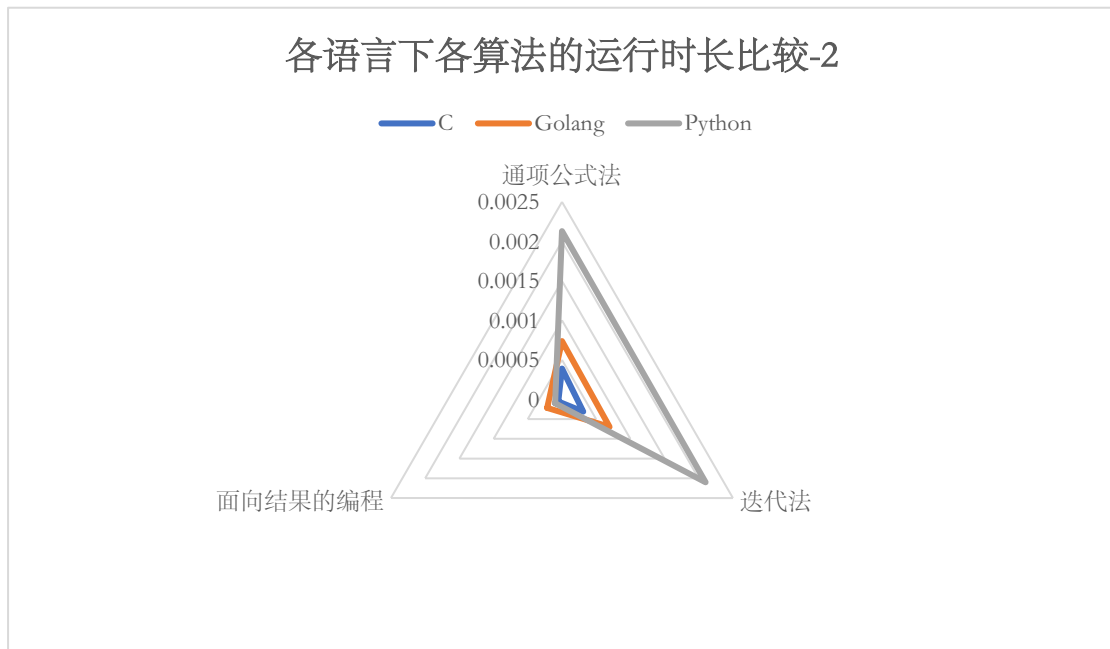
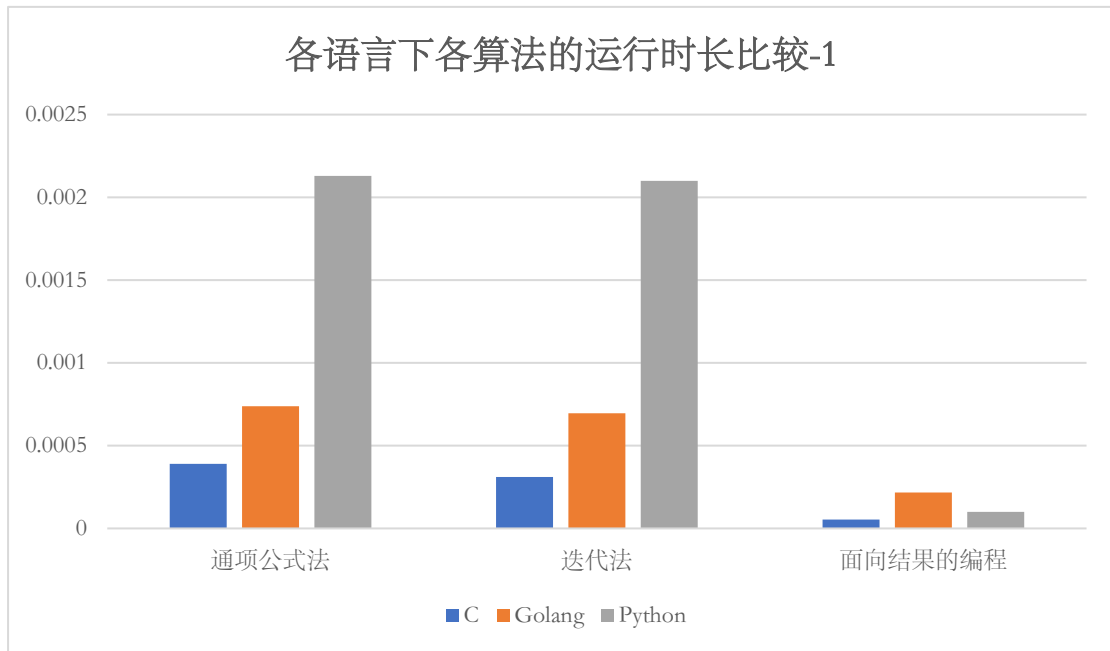
服务器由 Microsoft Azure 提供，采用 B1s 大小，基于 Hyper-V 虚拟化，提供 1vCPU，1GiB Memory, 64GiB P6 Disk. 运行 Azure 提供的 Ubuntu 22.04 LTS 发行版系统。除系统必须的功能外，仅运行了一个轻量 Caddy Webserver。在本环境下：

3.1. 建立列表（以秒为单位）

	C	Golang	Python
递归法	250.778	148.673	N/A(6h+)
通项公式法	0.000390	0.000738	0.00213
迭代法	0.000311	0.00069614	0.00210
面向结果的编程	0.000054	0.00021675	0.0001

3.2. 数据分析

由于递归法与其他算法时间相差量级过大，故不在柱状图、雷达图中展示。



## 4. 问题解决

我的解决方案写在文中了，文中我用黄色粗单下划线标注，在此一一列出，您可以用 Ctrl+左键 跳转到对应位置。

- ① [递归的基本定义，以及与迭代的区别](#)
- ② [递归法求解斐波那契数列问题的时间复杂度](#)
- ③ [通项公式法劣势](#)（参见：[Golang 实践中的误差](#)）
- ④ [动态规划法是什么](#)
- ⑤ [是否有其他解决方法](#)
- ⑥ [如何记录运行时间](#)
  - a) [提高精度：C 中 time\(\) 函数的其他替代——使用 CPU 时钟计时](#)（参见：[C 语言使用 time\(\) 函数无法实现高精度计时的实践](#)）

- b) [Golang, Python 的实现方案](#)
- ⑦ [Ubuntu 22.04 内置的 apt 源内, golang 不够新的解决办法](#)
- ⑧ [如何使用进程守护, 使进程在后台运行](#)

## 5. 原创声明

除文中特别指出的由“Bing Chat with GPT-4”或“网上检索所得”以外, 内容均由我个人原创。在这里特别列出非原创内容:

- ① [递归的基本定义, 以及与迭代的区别](#)
- ② [递归法求解斐波那契数列问题的时间复杂度](#)
- ③ [动态规划法是什么](#)
- ④ [是否有其他解决方法](#)

---

23090032047

计算机类 1 班 于景一