Числа Фибоначчи

26-май-2022

Вычисление последовательности чисел Фибоначчи уже стало “классикой” при обучении программированию и уже, возможно, успело поднадоесть. Но при этом здесь есть несколько интересных моментов и с точки зрения алгоритмов, и с точки зрения математики, которые стоит рассмотреть.

Сама последовательность имеет вид: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... (каждое последующее число кроме первых 0 и 1 равно сумме двух предыдущих).

```
In[•]:= Fibonacci /@ Range[0, 25]
Out[•]= {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765, 10 946, 17 711, 28 657, 46 368, 75 025}
```

Далее последовательно рассмотрим несколько способов получения n-го чисел Фибоначчи F_n. Наша программа `fib.py` будет выглядеть так -- по отдельной функции для каждого способа, а далее их вызов с подсчетом времени (и выводом F_n если то получилось не слишком большим).

```
import sys, time

# определения функций
def fib_recursion(n):
   ...

# функция для выполнения заданного метода
def run(func, out=True):
   t0 = time.time()
   f = func(n+1)
   if out:
      return [f, func.__name__, time.time()-t0]
   else:
      return [func.__name__, time.time()-t0]

# первый параметр n: номер числа
# второй параметр out=True/False: выводить ли само число
n = int(sys.argv[1])
if len(sys.argv) == 2 or sys.argv[2].lower() == 'true':
   out = True
else:
   out = False

# запуск методов из списка
fibs = {fib_recursion, ...}
result = []
for f in fibs:
   result.append(run(f, out))

# сортировка результатов по времени выполнения
```

```
if out:
   result = sorted(result, key = lambda x: x[2])
else:
   result = sorted(result, key = lambda x: x[1])

# вывод результата
for r in result:
   print(r)
```

Простая рекурсия

Рекурсивная природа чисел Фибоначчи заложена в их определение, поэтому простейший (наивный) способ получения F_n — это рекурсия:

```
def fib_recursion(n):
   if (n == 0):
      return 0
   elif (n == 1):
      return 1
   else:
      return fib_recursion(n-1) + fib_recursion(n-2)
```

Наивным этот способ является потому, что его скорость экспоненциальная $O(2^n)$. Например, вычисление 40-го числа заняло на моем компьютере 55 сек. Все остальные методы будут гораздо быстрее, поэтому далее рекурсию использовать не будем.

Рекурсия с кэшем

Так как мы постоянно вычисляем предыдущие числа, то для ускорения производительности можно задействовать *кэширование*. Кэширование — это метод оптимизации, который вы можете использовать в своих приложениях, чтобы хранить последние или часто используемые данные в областях памяти, доступ к которым происходит быстрее или с меньшими вычислительными затратами, чем к их источнику.

В качестве реализации кэша обычно используют словарь (помните, что он обеспечивает константное время доступа по ключу). Однако, если ничего не делать, то такой словарь может стать огромным и занять всю память. Существует несколько стратегий кэширования для поддержки заданного размера кэша:

Стратегия	Что удаляем	Когда пр
FIFO (First In, First Out)	Самые старые записи	Если новые записи будут
LIFO (Last In, First Out)	Самые новые записи	Если старые записи будут
LRU (Least Recently Used)	Давно не использованные	Если недавно использованные зап
MRU (Most Recently Used)	Недавно использованные	Если давно использованные запи
LFU (Least Frequently Used)	Редко используемые	Если часто используемые записи

Для нашего случая наиболее подходит LRU кэш, который реализован в модуле `functools`. Код новой функции изменится так:

```
from functools import lru_cache
```

```
@lru_cache()
def fib_cache(n):
   if (n == 0):
      return 0
   elif (n == 1):
      return 1
   else:
      return fib_cache(n-1) + fib_cache(n-2)
```

По умолчанию размер кэша равен 128 (этого вполне достаточно), но вы можете изменить его (например, `@lru_cache(maxsize=16)`). Работает кэш-версия намного лучше:

```
% python3 fib.py 40
[165580141, 'fib_cache', 1.9788742065429688e-05]
% python3 fib.py 240
[1038810421957299147085105183827754016801420367758 41, 'fib_cache',
0.0002319812774658203]
```

Однако, начиная с некоторого момента (n ≈ 500) мы начинаем сталкиваться с другой проблемой — переполнением стека рекурсии (RecursionError: maximum recursion depth exceeded). Для исправления можно увеличить размер стека, однако это не решает проблему принципиально:

```
sys.setrecursionlimit(5000)

@lru_cache()
def fib_cache(n):
   ...
```

Как видно, рекурсия позволяет решать задачи элегантно, но для практического использования не подходит ввиду экспоненциальной сложности и переполнения стека.

Динамическое программирование

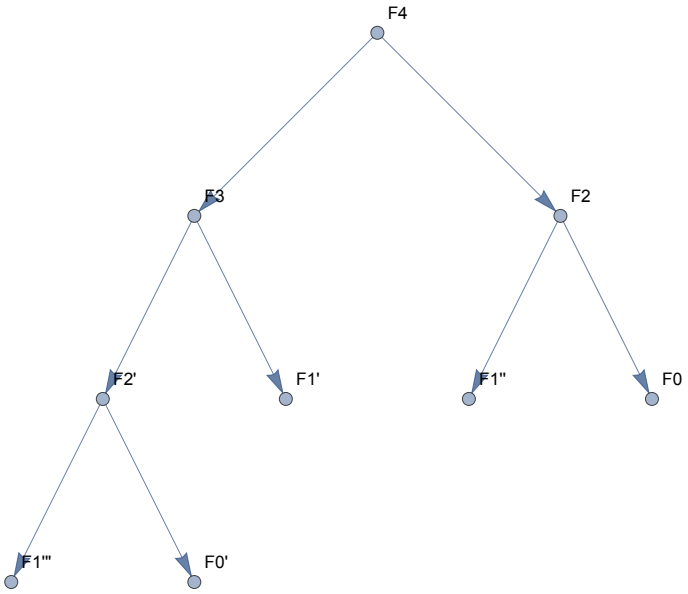
Кэширование является только одним из примеров использования *мемоизации*, т.е. запоминания промежуточных результатов. Мемоизация также используется в *динамическом программировании*. Секрет этого подхода заключается в определении, выдает ли простая рекурсия одинаковые результаты для одинаковых подзадач. Если выдает, то вместо повторения вычислений ответ каждой подзадачи можно сохранять в таблице (словаре) для использования в дальнейшем. Также для применения динамического подхода нужно, чтобы 1) исходная задача имела оптимальную подструктуру и 2) подзадачи перекрывались.

Оптимальная подструктура в динамическом программировании означает, что оптимальное решение подзадач меньшего размера может быть использовано для решения исходной задачи. В нашем случае мы можем эффективно находить F_4 из F_3 и F_2 (путем их сложения).

Подзадачи перекрываются, так как для вычисления F_4 нам нужно вычислить F_3 — 1 раз, F_2 — 2 раза, F_1 — 3 раза, F_0 — 2 раза (см. граф ниже). Если использовать мемоизацию, то каждое значение будет вычислено только 1 раз.

```
GraphPlot[{"F4" ↦ "F3", "F4" ↦ "F2", "F3" ↦ "F2'", "F3" ↦ "F1''", "F2" ↦ "F1'''",
  "F2" ↦ "F0", "F2'" ↦ "F1''''", "F2'" ↦ "F0'"}, VertexLabels → "Name"]
```

Out[ ]=



Приведенный ниже код как раз демонстрирует мемоизацию в действии. Для хранения промежуточных значений используется словарь.

```
def fib_dynamic(n):
   fib = {1: 1, 2: 1}
   if (n > 2):
      for i in range(3, n+1):
         fib[i] = fib[i-1] + fib[i-2]
   return fib[n]
```

Временная сложность алгоритма O(n). Как мы видим, результаты заметно превосходят те, что получены кэшированием.

```
% python3 fib.py 300
[359579325206583560961765665172189099052367214309267232255589801, 'fib_dynamic',
5.793571472167969e-05]
[359579325206583560961765665172189099052367214309267232255589801, 'fib_cache',
0.0002701282501220703]
```

Динамическое программирование обычно придерживается двух подходов к решению задач:
1) Восходящее динамическое программирование (bottom-up): все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи. Пример вы видите выше.
2) Нисходящее динамическое программирование (top-down): задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи.

Упражнение: Реализуйте top-down подход самостоятельно. Сравните его быстродействие с bottom-up.

Итеративный подход

Итеративный подход реализуется тем же циклом (временная сложность O(n)), но с использованием всего двух переменных (пространственная сложность O(1)), поэтому работает он намного быстрее:

```
def fib_iterative(n):
   new, old = 1, 0
   for i in range(n):
      new, old = old, new + old
   return old
```

```
% python3 fib.py 500
[22559151616193633087251269503607207204601132491375819058863886641847462773868688340
5015987052796968498626, 'fib_iterative', 2.8133392333984375e-05]
[22559151616193633087251269503607207204601132491375819058863886641847462773868688340
5015987052796968498626, 'fib_dynamic', 0.00011491775512695312]
[22559151616193633087251269503607207204601132491375819058863886641847462773868688340
5015987052796968498626, 'fib_cache', 0.0004100799560546875]
```

Итеративный метод уже можно использовать для вычисления миллионного по счету числа. На моем компьютере расчет занял около 10 секунд (без вывода самого числа). Для сравнения, динамическое программирование справилось почти за 2 минуты.

```
% python3 fib.py 1000000 False
['fib_iterative', 9.868557929992676]
['fib_dynamic', 117.04595994949341]
```

Можно ли реализовать расчет еще быстрее? Оказывается, математика позволяет довести временную сложность до O(log n).

Матричное возведение в степень

Одна из идей строится на матричном представлении чисел Фибоначчи. Нетрудно заметить, что $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$. Немного труднее заметить, что $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_3 & F_2 \\ F_2 & F_1 \end{pmatrix}$, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} F_4 & F_3 \\ F_3 & F_2 \end{pmatrix}$, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix} = \begin{pmatrix} F_5 & F_4 \\ F_4 & F_3 \end{pmatrix}$, ..., $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$.

In[•]:=
```
MatrixPower[{{1, 1}, {1, 0}}, 2]
MatrixPower[{{1, 1}, {1, 0}}, 3]
MatrixPower[{{1, 1}, {1, 0}}, 4]
```

Out[•]= {{2, 1}, {1, 1}}

Out[•]= {{3, 2}, {2, 1}}

Out[•]= {{5, 3}, {3, 2}}

Последнее соотношение можно доказать по индукции. Для $n = 1$ оно выполняется, допустим его верность для $n = k$ и докажем для $n = k + 1$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{k+1} + F_k & F_{k+1} \\ F_k + F_{k-1} & F_k \end{pmatrix} = \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}.$$

Итак, исходная задача свелась к возведению матрицы $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ в степень n.

Решение “в лоб”

Сначала попробуем решить эту задачу “в лоб”, т.е. сами реализуем возведение матрицы в степень по определению $A^2 = A \cdot A$, $A^3 = A \cdot A \cdot A$, ... Так как наша матрица имеет размер 2x2, то для простоты будем хранить ее значения списком из 4 значений.

```
def fib_matrix(n):
   if n == 0:
      return 0
   elif n == 1 or n == 2:
      return 1
   else:
      q = [1, 1, 1, 0]
      for i in range(n-1):
         q = [q[0] + q[1], q[0], q[2] + q[3], q[2]]
      return q[1]
```

Работает такой наивный метод тоже достаточно быстро, однако все же проигрывает итеративному подходу:

```
% python3 fib.py 1000 False
['fib_iterative', 6.604194641113281e-05]
['fib_matrix', 0.0002219676971435547]
% python3 fib.py 2000 False
['fib_iterative', 0.0001511573791503906 2]
['fib_matrix', 0.0004758834838867187 5]
```

Использование Numpy

Математический пакет Numpy заметно упрощает жизнь математикам, которые используют для вычислений язык Python. Так, они позволяют создавать массивы (матрицы) `ndarray` и эффективно работать с ними. Для вычисления степени матрицы воспользуемся встроенной функцией `matrix_power` из модуля линейной алгебры `linalg` . В качестве ответа мы берем индекс [0, 1] (нулевая строка, первый столбец) результирующей матрицы, где расположено F_n. Обратите внимание, что в качестве типа данных передается значение `object` — без него вы достаточно быстро начнете получать некорректные значения.

```
import numpy as np
...
def fib_numpy(n):
   q = np.array([[1, 1], [1, 0]], dtype='object')
   qn = np.linalg.matrix_power(q, n)
   return qn[0, 1]
```

Numpy показывает сравнимые результаты с итеративным методом при n<1000, однако, затем вырывается заметно вперед. Например, расчет миллионного по счету числа на моем компьютере занял 0.27 сек против 10 сек, т.е. быстрее в 35 раз. За 10 секунд Numpy позволяет

уже вычислить 10-миллинное по счету число Фибоначчи.

```
% python3 fib.py 1000000 False
['fib_numpy', 0.27225708961486816]
['fib_iterative', 9.804173231124878]
```

Конечно, хотелось бы добиться подобного результата алгоритмически, без использования сторонних библиотек.

Быстрое возведение в степень

Возвести что-то в степень n (число, матрицу) можно быстрее чем за n операций. Простейший алгоритм возведения в степень лучше всего демонстрируется схемой вычисления для x^{16}: вычисляем $x\,x = x^2$, вычисляем $x^2\,x^2 = x^4$, вычисляем $x^4\,x^4 = x^8$, вычисляем $x^8\,x^8 = x^{16}$. Такая (бинарная) схема позволяет вычислить значение x^{16} за 4 операции, а не за 15, т.е. свести вреемнную сложность к O(log n).

В общем виде (для любой степени $n > 1$) схема выглядит так:

$$x^n = \begin{cases} \left(x^2\right)^{n/2} & n \text{ четное} \\ x\left(x^2\right)^{(n-1)/2} & n \text{ нечетное} \end{cases}$$

Например, $x^{39} = x\left(x^2\right)^{19} = x\,x^2\left(x^4\right)^9 = x\,x^2\,x^4\left(x^8\right)^4 = x\,x^2\,x^4\left(x^{16}\right)^2 = x\,x^2\,x^4\,x^{32}$.

Эту схему можно реализовать итеративно:

```
def square_power(x, n):
   if n == 0:
      return 1
   y = 1
   while n > 1:
      if n % 2 == 0:
         x = x * x
         n = n / 2
      else:
         y = x * y
         x = x * x
         n = (n - 1) / 2
   return x * y
```

Интереснее всего, что эту же схему можно переписать для матриц, переписав код для операции произведения. В остальном, код полностью совпадает:

```
def fib_matrix_squaring(n):
   def matrix_product(A, B):
      return (a[0]*b[0]+a[1]*b[2], a[0]*b[1]+a[1]*b[3],
           a[2]*b[0]+a[3]*b[2], a[2]*b[1]+a[3]*b[3])

   if n == 0:
      return 0
   elif n == 1 or n == 2:
      return 1
   else:
      x = [1, 1, 1, 0]
      y = [1, 0, 0, 1]
   while n > 1:
```

```
    if n % 2 == 0:
      x = matrix_product(x, x)
      n = n / 2
    else:
      y = matrix_product(x, y)
      x = matrix_product(x, x)
      n = (n - 1) / 2
  return matrix_product(x, y)[1]
```

Производительность этого кода сравнима с использованием Numpy.

```
% python3 fib.py 100000 False
['fib_numpy', 0.006320953369140625]
['fib_matrix_squaring', 0.006509065628051758]
['fib_iterative', 0.09850883483886719]
['fib_matrix', 0.22255778312683105]
```

Таким образом, нам удалось получить достаточно быстрый алгоритм (O(log *n*))для вычисления чисел Фибоначчи. Можно ли произвести расчет еще быстрее, т.е. за константное время O(1)? Для этого нужно свести вычисление F_nк формуле. Оказывается, такая формула есть.

Формула Бине

Собственные значения и векторы

Можно ли еще как-то упростить вычисление $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$? Если бы вместо $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ мы бы имели $\begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}^n$, то это сделало бы возведение в степень тривиальным $\begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}^n = \begin{pmatrix} \lambda^n & 0 \\ 0 & \lambda^n \end{pmatrix}$.

Собственно, матрица $\begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}$ называется матрицей собственных значений и ее можно вычислить.

Собственным вектором (*eigenvector*)матрицы *A* называют такой ненулевой вектор, который при умножении на A дает коллинеарный вектор — тот же вектор, только умноженный на константу, т.е. $A\,v = \lambda\,v$. Эта константа λ называется собственным значением (*eigenvalue*) матрицы *A*. Многие соотношения существенно упрощаются в базисе из собственных векторов, что мы здесь и используем.

Для начала найдем собственные значения и векторы матрицы $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. По определению имееем $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \lambda \cdot \begin{pmatrix} x \\ y \end{pmatrix}$, откуда получаем систему уравнений $x + y = \lambda\,x,\ x = \lambda\,y$, из которой можно исключить $x,\ y$ и получить уравнение $\lambda^2 - \lambda - 1 = 0$ откуда получаем два собственных значения $\lambda_1 = \frac{1}{2}\left(1 + \sqrt{5}\right)$, $\lambda_2 = \frac{1}{2}\left(1 - \sqrt{5}\right)$. Вектора получаются после последовательной подстановки $\lambda_{1,2}$: $v_1 = \left(\frac{1}{2}\left(1 + \sqrt{5}\right), 1\right)$ и $v_2 = \left(\frac{1}{2}\left(1 - \sqrt{5}\right), 1\right)$.

```
In[ ]:= Eigenvalues[{{1, 1}, {1, 0}}]
        Eigenvectors[{{1, 1}, {1, 0}}]
```

Out[ ]= $\left\{\frac{1}{2}\left(1+\sqrt{5}\right), \frac{1}{2}\left(1-\sqrt{5}\right)\right\}$

Out[ ]= $\left\{\left\{\frac{1}{2}\left(1+\sqrt{5}\right), 1\right\}, \left\{\frac{1}{2}\left(1-\sqrt{5}\right), 1\right\}\right\}$

Однако, мы не можем просто так заменить $\begin{pmatrix}1&1\\1&0\end{pmatrix}^n$ на $\begin{pmatrix}\lambda_1&0\\0&\lambda_2\end{pmatrix}^n$, ведь диагональной матрица становится именно в базисе собственных векторов, а не в стандартном базисе. При переходе между двумя базисами матрица A преобразуется по формуле $A' = C\,A\,C^{-1}$, где C — матрица перехода между базисами.

В нашем случае один и тот же вектор w имеет разные координаты в этих двух базисах: $w = b_1\begin{pmatrix}\frac{1}{2}\left(1+\sqrt{5}\right)\\1\end{pmatrix} + b_2\begin{pmatrix}\frac{1}{2}\left(1-\sqrt{5}\right)\\1\end{pmatrix} = a_1\begin{pmatrix}1\\0\end{pmatrix} + a_2\begin{pmatrix}0\\1\end{pmatrix}$. Это можно записать в матричном виде $\begin{pmatrix}\frac{1}{2}\left(1+\sqrt{5}\right)&\frac{1}{2}\left(1-\sqrt{5}\right)\\1&1\end{pmatrix}\cdot\begin{pmatrix}b_1\\b_2\end{pmatrix} = \begin{pmatrix}1&0\\0&1\end{pmatrix}\cdot\begin{pmatrix}a_1\\a_2\end{pmatrix} = \begin{pmatrix}a_1\\a_2\end{pmatrix}$, откуда $C = \begin{pmatrix}\frac{1}{2}\left(1+\sqrt{5}\right)&\frac{1}{2}\left(1-\sqrt{5}\right)\\1&1\end{pmatrix}$ и есть наша матрица перехода.

Тогда $\begin{pmatrix}F_{n+1}&F_n\\F_n&F_{n-1}\end{pmatrix} = \begin{pmatrix}1&1\\1&0\end{pmatrix}^n = C\begin{pmatrix}\lambda_1&0\\0&\lambda_2\end{pmatrix}^n C^{-1} = C\begin{pmatrix}{\lambda_1}^n&0\\0&{\lambda_2}^n\end{pmatrix}C^{-1}$, т.е. мы свели возведение матрицы в степень к возведению чисел в степень.

Не останавливаясь на достигнутом упростим полученное выражение при помощи Mathematica:

```
In[ ]:= {{λ1, λ2}, {1, 1}}.{{λ1^n, 0}, {0, λ2^n}}.Inverse[{{λ1, λ2}, {1, 1}}] // Factor
        % /. {λ1 → 1/2 (1 + Sqrt[5]), λ2 → 1/2 (1 - Sqrt[5]), n → 10} // Simplify
```

Out[ ]= $\left\{\left\{\frac{\lambda 1^{1+n} - \lambda 2^{1+n}}{\lambda 1 - \lambda 2}, -\frac{\lambda 1\,\lambda 2\left(\lambda 1^n - \lambda 2^n\right)}{\lambda 1 - \lambda 2}\right\}, \left\{\frac{\lambda 1^n - \lambda 2^n}{\lambda 1 - \lambda 2}, \frac{-\lambda 1^n\,\lambda 2 + \lambda 1\,\lambda 2^n}{\lambda 1 - \lambda 2}\right\}\right\}$

Out[ ]= {{89, 55}, {55, 34}}

```
In[ ]:= Fibonacci[10]
```

Out[ ]= 55

Удивительно, но возводя в степень и перемножая числа вроде $\frac{1}{2}\left(1+\sqrt{5}\right)$ мы получили на выходе целые числа!

Формула Бине

Еще более удивительно, что число $F_n = \frac{\lambda 1^n - \lambda 2^n}{\lambda 1 - \lambda 2} = \frac{\left(\frac{1}{2}\left(1+\sqrt{5}\right)\right)^n - \left(\frac{1}{2}\left(1-\sqrt{5}\right)\right)^n}{\frac{1}{2}\left(1+\sqrt{5}\right) - \frac{1}{2}\left(1-\sqrt{5}\right)} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$, т.е. мы получили формулу для числа Фибоначчи в замкнутом виде. Это выражение была получено Жаком Филиппом Бине в 1843 году, хотя известно он было жившим гораздо раньше Даниилу

Бернулли и Абрахаму де Муавру.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

In[ ]:=
$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) /. n \to 10 // \text{Expand}$$

Out[ ]= 55

Реализуем формулу Бине на практике. Стандартный Python, в отличие от Mathematica, не оперирует символами, поэтому приходится вычислять вещественное значение, которое в конце округляется до целого.

```
def fib_binet(n):
   r5 = 5**0.5
   f = (((1+r5)/2)**n-((1-r5)/2)**n)/r5
   return round(f)
```

Такая реализация выдает неточный результаты уже для $n = 70$. Кроме того, для $n \geq 1474$ в процессе вычислений превышается максимально возможное вещественное число, о чем Python сообщает “Result too large”.

```
% python3 fib.py 70
[308061521170130, 'fib_binet', 4.0531158447265625e-06]
[308061521170129, 'fib_iterative', 5.245208740234375e-06]
```

Чтобы справиться с этим задействуем модуль `decimal` для создания десятичного объекта с гораздо более высокой точностью (например, 10000 цифр):

```
def fib_binet_decimal(n):
   decimal.getcontext().prec = 10000
   r5 = decimal.Decimal(5).sqrt()
   f = (((1+r5)/2)**n-((1-r5)/2)**n)/r5
   return round(f)
```

Однако, формула Бине не спасает, так как нам 1) нужно возвести числа в степень и 2) эти числа иррациональны. В итоге этот метод работает гораздо медленнее даже итеративного метода.

```
% python3 fib.py 5000 False
['fib_iterative', 0.0005109310150146484]
['fib_binet_decimal', 0.47904109954833984]
```

Конечно, мы можем применить здесь бинарное возведение в степень, но как быть с иррациональностью? На помощь опять приходит математика.

Формула Бине без плавающей точки

Итак, как же быстро вычислять формулу Бине?

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Можно заметить следующее: если у нас числа $a + b\sqrt{5}$ и $c + d\sqrt{5}$, то складывая и умножая их мы также получим числа вида $x + y\sqrt{5}$.

```
In[•]:= Collect[(a + b Sqrt[5]) + (c + d Sqrt[5]), Sqrt[5]]
        Collect[(a + b Sqrt[5]) (c + d Sqrt[5]), Sqrt[5]]
Out[•]= a + c + Sqrt[5] (b + d)
Out[•]= a c + 5 b d + Sqrt[5] (b c + a d)
```

Т.е. множество чисел вида $x = a + b\sqrt{5}$ алгебраически замкнуто относительно операций сложения и умножения. Если задаться целью, то можно показать, что множество таких чисел образует кольцо. Но зачем это нам нужно? Только для того, чтобы избавиться от иррационального $\sqrt{5}$. Действительно, мы можем свести операции сложения, вычитания и умножения в формуле Бине к следующим операциям над парами целых чисел:

$(a, b) + (c, d) = (a + c, b + d)$
$(a, b) - (c, d) = (a - c, b - d)$
$(a, b) * (c, d) = (a\,c + 5\,b\,d, a\,d + b\,c)$

Формулу Бине тогда можно представить в виде $\frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$. Мы знаем, что эта формула выдает целое число, значит числитель должен иметь вид $0 + x\sqrt{5}$, иначе $\sqrt{5}$ в знаменателе не сократится). Получается, для вычисления этой формулы нам нужно:

1) вместо $1 + \sqrt{5}$ возвести в степень n соотвествующую пару (1, 1) из кольца
2) вместо $1 - \sqrt{5}$ возвести в степень n соотвествующую пару (1, –1) из кольца
3) найти разность этих пар и взять второй элемент (т.е. x из $0 + x\sqrt{5}$)
4) разделить его на 2^n

Для быстрого возведения в степень мы будем использовать бинарный подход, описанный ранее. Код функции выглядит следующим образом:

```
def fib_binet_algebraic(n):
   def pair_subtraction(a, b):
      return (a[0]-b[0], a[1]-b[1])

   def pair_product(a, b):
      return (a[0]*b[0] + 5*a[1]*b[1], a[0]*b[1] + a[1]*b[0])

   def pair_power(a, n):
      if n == 0:
         return (1, 0)
      elif n == 1:
         return (1, 1)
      else:
         x = a
         y = (1, 0)
      while n > 1:
         if n % 2 == 0:
            x = pair_product(x, x)
            n = n / 2
         else:
            y = pair_product(x, y)
            x = pair_product(x, x)
```

```
            n = (n - 1) / 2
        return pair_product(x, y)

    a = pair_power((1, 1), n)
    b = pair_power((1, -1), n)

    return pair_subtraction(a, b)[1] // (1<<n)
```

Обратите внимание, что в конце мы делим разность на 1<<n. Этот трюк позволяет вычислить степень двойки намного быстрее чем 2**n. Оператор << представляет собой сдвиг битов на указанное количество разрядов влево. Сдвигая 1 влево мы получаем 1<<1 → 2 (10 в двоичной записи), 1<<2 → 4 (100 в двоичной записи) и т.д.

```
% python3 fib.py 100000 False
['fib_numpy', 0.006067991256713867]
['fib_matrix_squaring', 0.0066699981689453125]
['fib_binet_algebraic', 0.013692140579223633]
['fib_iterative', 0.097168924243164]
```

Скорость такого “алгебраического” метода в два раза медленнее чем у матричного. Связано это с тем, что мы возводим в степень два числа. Однако, можно заметить, что второе число $\left(\frac{1-\sqrt{5}}{2}\right)^n$ очень быстро становится близким к нулю. Таким образом, для достаточно больших n мы можем им пренебречь и свести формулу Бине к виду $F^*_n = \frac{\left(1+\sqrt{5}\right)^n}{2^n\ \sqrt{5}}$.

Упражнение: Избавьтесь от второго слагаемого в формуле Бине, убедитесь, что формула по-прежнему верна и сравните скорость с другими методами.

Таким образом, матричный и алгебраический метод сводят вычисление чисел Фибоначчи к бинарному возведению в степень со сложностью O(log n). Формулы для вычисления за константное время O(1) нет — только если вы не создадите огромный словарь и не поместите туда все известные числа.

Но можно ли как-то ускорить существующие алгоритмы?

Самый быстрый алгоритм

Наиболее быстрый (из найденных мною) использует следующие соотношения между числа Фибоначчи:

$F(2k) = F(k)\,(2F(k+1) - F(k))$
$F(2k+1) = F(k+1)^2 + F(k)^2$

Код программы основан на алгоритме возведения матрицы в степень, но понять его заметно сложнее:

```
def fib_fast_doubling(n):
    def doubling(n):
        if n == 0:
            return (0, 1)
        else:
            a, b = doubling(n // 2)
            c = a * (b * 2 - a)
            d = a * a + b * b
```

```
        if n % 2 == 0:
           return (c, d)
        else:
           return (d, c + d)

  return doubling(n)[0]
```

Асимптотика у этого алгоритма та же O(log *n*), но работает он быстрее остальных.

```
% python3 fib.py 1000000 False
['fib_fast_doubling', 0.062354087829589844]
['fib_matrix_squaring', 0.2655608654022217]
['fib_numpy', 0.2684822082519531]
['fib_binet_algebraic', 0.6422977447509766]
```

Любой алгоритм с такой (трюковой) природой имеет ограниченное применение. Матричный и алгебраический методы гораздо понятнее и используют “полезную” математику, а значит расширяют вас кругозор и арсенал инструментов.

Использованные источники

https://habr.com/ru/company/skillfactory/blog/555914/ -- Как я посчитал миллионное число Фибоначчи
https://habr.com/ru/post/559520/ -- N-е число обобщённых Фибоначчи за O(log N)
https://habr.com/ru/post/148336/ -- N-е число Фибоначчи за O(log N)
https://habr.com/ru/post/449616/ -- Фибоначчи на собеседовании
https://habr.com/ru/post/261159/ -- 5 способов вычисления чисел Фибоначчи: реализация и сравнение
https://habr.com/ru/post/645829/ -- Формула Бине без плавающей точки
http://www.oranlooney.com/post/fibonacci/ -- A Fairly Fast Fibonacci Function
https://fabiandablander.com/r/Fibonacci.html -- The Fibonacci sequence and linear algebra
https://www.nayuki.io/page/fast-fibonacci-algorithms -- Fast Fibonacci algorithms