Алгоритмы.
Ряд Фибоначчи. Реализация на Python и Java.

Ряд Фибоначчи

Ряд (последовательность) Фибоначчи — последовательность целых чисел. Задается линейным рекуррентным соотношением.

$$F_0=0, F_1=1, F_n=F_{n-1}+F_{n-2}$$

$$n \geqslant 2, n \in \mathbb{Z}$$

Иногда последовательность Фибоначчи продолжают и в область отрицательных чисел для этого используется «обратная формула».

$$F_n=F_{n+1}+F_{n+2}$$

Сведения о ряде Фибоначчи

$$F_1+F_2+F_3+F_4+\ldots+F_n=F_{n+2}-1$$

$$F_1+F_3+F_5+F_7+\ldots+F_{2n-1}=F_{2n}$$

$$F_2+F_4+F_6+F_8+\ldots+F_{2n}=F_{2n+1}-1$$

Формула Бине выражает в явном виде зависимость:

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

где $\varphi=\frac{1+\sqrt{5}}{2}$ золотое сечение

Применение и способ вычисления ряда Фибоначчи

Ряды Фибоначчи используются как в математических исследованиях, в моделировании биологических процессов и в некоторых алгоритмах (например в алгоритме поиска Фибоначчи). Поэтому рассмотрения способов генерации этой последовательности выглядит довольно интересным. Наиболее часто используются следующие подходы:

1) Циклический
2) Рекурсивный
3) Рекурсивный плюс мемоизация

Рассмотрим реализацию этих подходов на Python и Java.

Реализация алгоритма на Python

Генерация с помощью цикла

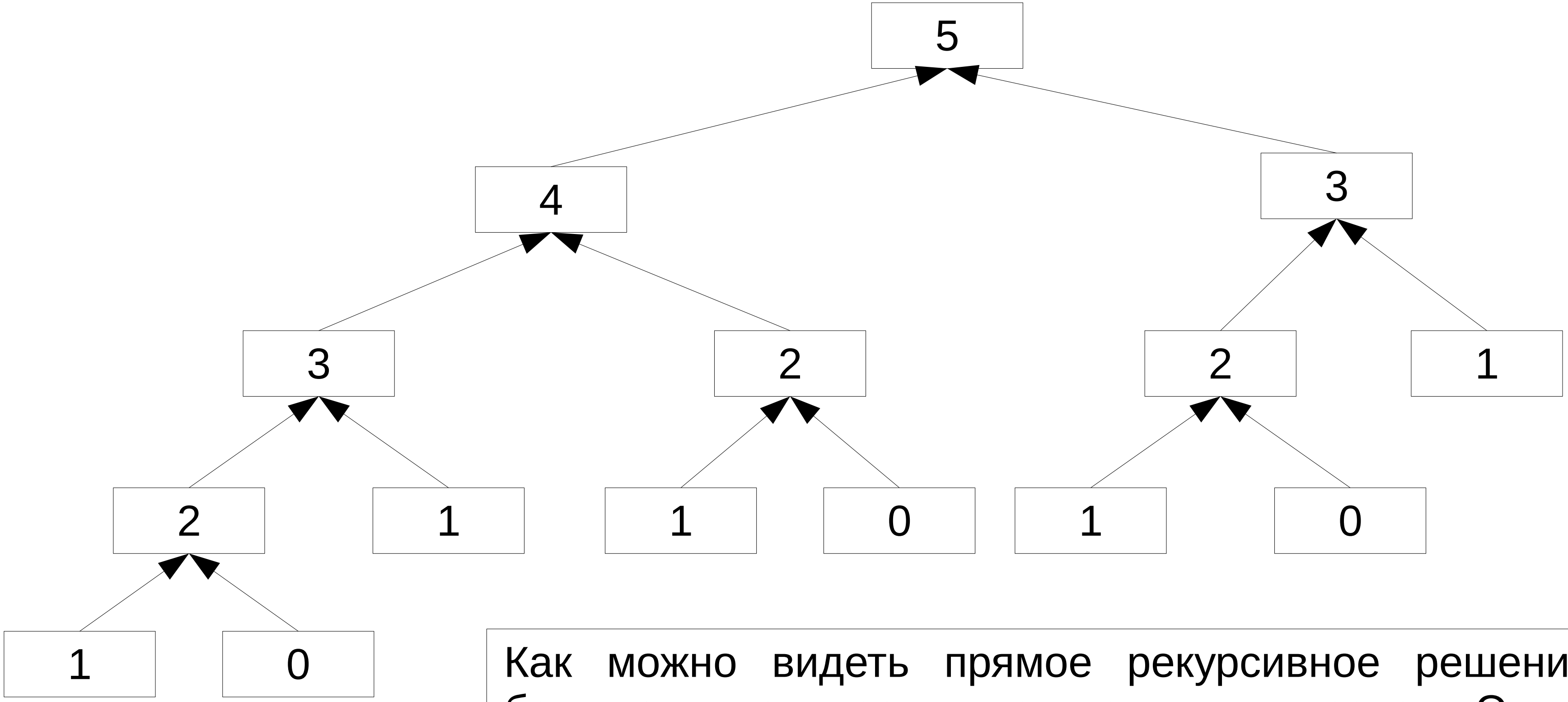
```
def fibonacci_sequince(n):
    result = 0
    next_element = 1
    index = 0
    while index < n:
        next_element, result = next_element+result, next_element
        index += 1
    return result
```

Генерация с помощью рекурсии

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

Такое решение хоть и простое, но обладает существенными недостатками. При таком решение получается параллельная (множественная) рекурсия. Поэтому довольно быстро количество рекурсивных вызовов превышает предельно допустимое и вычисляется очень долго. Например уже вычисление 50 члена ряда Фибоначчи становится очень долгим.

Причины большой ресурсоемкости данного решения



Как можно видеть прямое рекурсивное решение приведет к большому количеству рекурсивных вызовов. Эти вызовы часто идут с одним и тем же параметром. Значит применение мемоизации сможет значительно ускорить данное решение.

Python

Генерация с помощью рекурсии с мемоизацией

```
mem = dict()
```

Хранилище для использованных параметров функции

```
def fibonacci_sequince(n):
    if n in mem:
        return mem[n]
```

Если параметр уже был использован то просто возвращаем результат

```
    if n == 0:
        mem[n] = 0
    elif n == 1:
        mem[n] = 1
    else:
        mem[n] = fibonacci_sequince(n-1)+fibonacci_sequince(n-2)
    return mem[n]
```

Если нет то вычисляем значение и помещаем в хранилище

Применение мемоизации снимает проблему множественных вызовов. Теперь вычисление 50 члена ряда Фибоначчи не представляет труда.

Реализация алгоритма на Java

Реализация с помощью цикла

```java
public static BigInteger fibonacciSequince(int n) {
    BigInteger result = BigInteger.ZERO;
    BigInteger next = BigInteger.ONE;
    int currentIndex = 0;
    for (; currentIndex < n;) {
        next = result.add(next);
        result = next.subtract(result);
        currentIndex++;
    }
    return result;
}
```

Java

Реализация с помощью рекурсии

```
public static BigInteger fibonacciSequince(int n) {
    if (n == 0) {
        return BigInteger.ZERO;
    }
    if (n == 1) {
        return BigInteger.ONE;
    }
    return fibonacciSequince(n - 1).add(fibonacciSequince(n - 2));
}
```

Терминальная часть

Рекурсивная

Реализация с помощью рекурсии и мемоизации

```
static Map<Integer, BigInteger> mem = new HashMap<>();

public static BigInteger fibonacciSequince(int n) {
    BigInteger result = mem.get(n);
    if (result != null) {
        return result;
    } else if (n == 0) {
        mem.put(0, BigInteger.ZERO);
    } else if (n == 1) {
        mem.put(1, BigInteger.ONE);
    } else {
        mem.put(n, fibonacciSequince(n - 1).add(fibonacciSequince(n - 2)));
    }
    return mem.get(n);
}
```

Хранилище для хранения

Если параметр уже был использован то просто возвращаем результат

Если нет то вычисляем значение и помещаем в хранилище

Теорема Цекендорфа

Теорема Цекендорфа гласит, что всякое натуральное число можно единственным образом представить в виде суммы одного или нескольких различных чисел Фибоначчи так, чтобы в этом представлении не оказалось двух соседних чисел из последовательности Фибоначчи.

Для любого натурального числа N существуют натуральные числа

$c_i \geqslant 2, c_{i+1} > c_i + 1$

$$N = \sum_{i=0}^{k} F(c_i)$$

Где c_i член ряда Фибоначчи

Например, представление Цекендорфа для 100 = 89 + 8 + 3.

В то же время разложения вида:
100 = 89 + 8 + 2 + 1
100 = 55 + 34 + 8 + 3
не являются представлением Цекендорфа поскольку 1 и 2 или 34 и 55 являются последовательными числами Фибоначчи.

Цель исследования

Целью исследования является реализация алгоритма для нахождения представления Цекендорфа для произвольного натурального числа. Также можно исследовать количество членов в подобном разложении.

В основу реализации этого алгоритма положим вычисление n члена ряда Фибоначчи рекурсивным методом с использованием мемоизации. Так, как при решении нам явно потребуются все промежуточные результаты, то будем использовать подход с хранением всех промежуточных вычислений.

Описание алгоритма

1) Находим такое целое число k что F(k)≤N где N это число представление Цекендрофа которого мы ищем.
2) Если F(k)≤N заносим F(k) в результирующее разбиение и указываем новое значение N как N-F(k). Уменьшаем значение k на единицу. Переходим к пункту **3**.
3) Если N=0 **заканчиваем алгоритм** в противном случае возвращаемся к пункту **2**.

Вычисление k члена ряда

```
static Map<Integer, BigInteger> mem = new HashMap<>();

public static BigInteger fibonacciSequince(int n) {
    BigInteger result = mem.get(n);
    if (result != null) {
        return result;
    } else if (n == 0) {
        mem.put(0, BigInteger.ZERO);
    } else if (n == 1) {
        mem.put(1, BigInteger.ONE);
    } else {
        mem.put(n, fibonacciSequince(n - 1).add(fibonacciSequince(n - 2)));
    }
    return mem.get(n);
}
```

Получение представления Цекендорфа

```java
public static List<BigInteger> representationOfZeckendorf(BigInteger number) {
    List<BigInteger> roz = new ArrayList<>();
    int k = 0;
    for (; fibonacciSequince(k).compareTo(number) <= 0;) {
        k = k + 1;
    }
    for (; number.compareTo(BigInteger.ZERO) > 0;) {
        if (fibonacciSequince(k).compareTo(number) <= 0) {
            BigInteger n = fibonacciSequince(k);
            roz.add(n);
            number = number.subtract(n);
        }
        k = k - 1;
    }
    return roz;
}
```

Пример использования

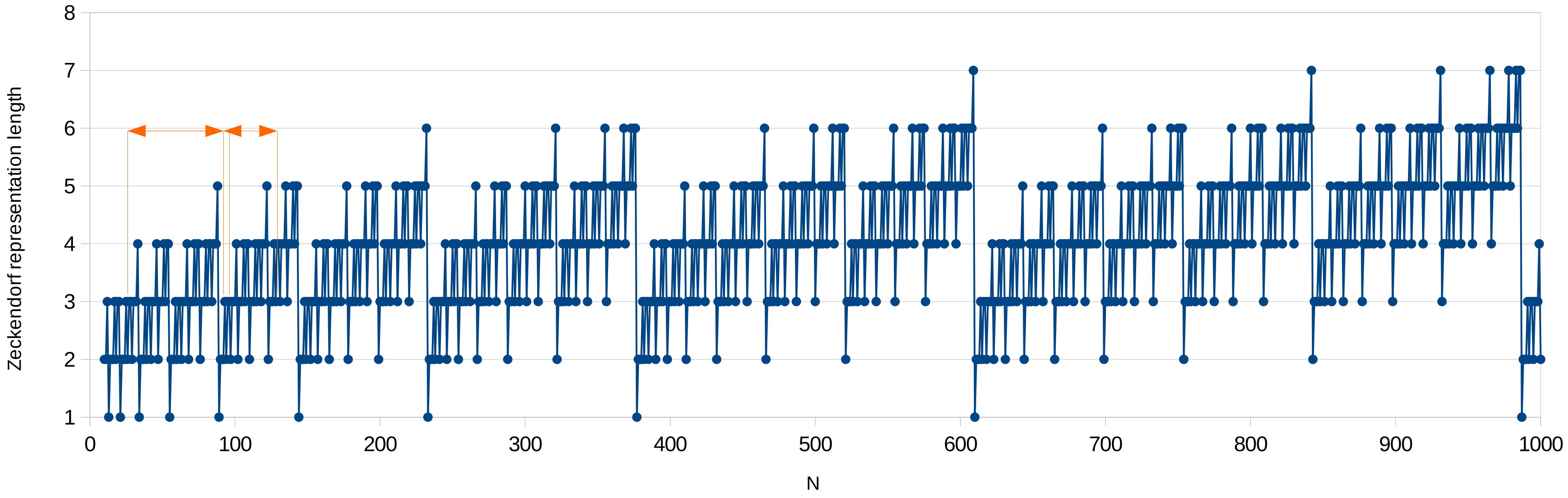
```
public static void main(String[] args) {

    System.out.println(representationOfZeckendorf(BigInteger.valueOf(100)));

}
```

В результате работы получим [89, 8, 3] что является правильным представлением.

Результаты исследования



На графике вы видите зависимость длинны представления в зависимости от числа. Эта зависимость носит любопытную фрактальную природу. Один элемент постоянно используется в последующих элементах(выделено на графике), и дальше структуры создаются на основе подобных конструкций.

Список литературы

1) Дональд Кнут. Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol. 1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. — С. 720. — ISBN 0-201-89683-4.
2) Теорема Цекендорфа