**Рекурсивное вычисление чисел Фибоначчи**

Можно рекурсивно определить числа Фибоначчи (Fibonacci numbers) при помощи уравнений:

Fib(0) = 0

Fib(1) = 1

Fib(N) = Fib(N - 1) + Fib(N - 2)         для N > 1.

Третье уравнение рекурсивно дважды вызывает функцию Fib, один раз с входным значением N-1, а другой — со значением N-2. Это определяет необходимость 2 условий остановки рекурсии: Fib(0)=0 и Fib(1)=1. Если задать только одно из них, рекурсия может оказаться бесконечной. Например, если задать только Fib(0)=0, то значение Fib(2) могло бы вычисляться следующим образом:

Fib(2)  = Fib(1) + Fib(0)

        = [Fib(0) + Fib(-1)] + 0

        = 0 + [Fib(-2) + Fib(-3)]

        = [Fib(-3) + Fib(-4)] + [Fib(-4) + Fib(-5)]

        И т.д.

Это определение чисел Фибоначчи легко преобразовать в рекурсивную функцию:

Public Function Fib(num As Integer) As Integer

    If num <= 1 Then

        Fib = num

    Else

        Fib = Fib(num – 1) + Fib(num - 2)

    End If

End Function

**Анализ времени выполнения программы**

Анализ этого алгоритма достаточно сложен. Во‑первых, определим, сколько раз выполняется одно из условий остановки num <=1. Пусть G(N) — количество раз, которое алгоритм достигает условия остановки для входа N. Если N <= 1, то функция достигает условия остановки один раз и не требует рекурсии.

Если N > 1, то функция рекурсивно вычисляет Fib(N-1) и Fib(N-2), и завершает работу. При первом вызове функции, условие остановки не выполняется — оно достигается только в следующих, рекурсивных вызовах. Полное число выполнения условия остановки для входного значения N, складывается из числа раз, которое оно выполняется для значения N-1 и числа раз, которое оно выполнялось для значения N-2. Все это можно записать так:

G(0) = 1

G(1) = 1

G(N) = G(N - 1) + G(N - 2)               для N > 1.

Это рекурсивное определение очень похоже на определение чисел Фибоначчи. В табл. 5.2 приведены некоторые значения функций G(N) и Fib(N). Легко увидеть, что G(N) = Fib(N+1).

Теперь рассмотрим, сколько раз алгоритм достигает рекурсивного шага. Если N<=1, функция не достигает этого шага. При N>1, функция достигает этого шага 1 раз и затем рекурсивно вычисляет Fib(n-1) и Fib(N-2). Пусть H(N) — число раз, которое алгоритм достигает рекурсивного шага для входа N. Тогда H(N)=1+H(N-1)+H(N-2). Уравнения, определяющие H(N):

H(0) = 0

H(1) = 0

H(N) = 1 + H(N - 1) + H(N - 2)           для N > 1.

Имеем: H(N)=Fib(N+1)-1.

Объединяя результаты для G(N) и H(N), получаем полное время выполнения для алгоритма:

Время выполнения   = G(N) + H(N)

                   = Fib(N + 1) + Fib(N + 1) - 1

                   = 2 \* Fib(N + 1) - 1

Поскольку Fib(N + 1) >= Fib(N) для всех значений N, то:

Время выполнения   >= 2 \* Fib(N) - 1

С точностью до порядка это составит O(Fib(N)). Интересно, что эта функция не только рекурсивная, но она также используется для оценки времени ее выполнения.

Чтобы помочь вам представить скорость роста функции Фибоначчи, можно показать, что Fib(M)>ÆM-2 где Æ — константа, примерно равная 1,6. Это означает, что время выполнения не меньше, чем значение экспоненциальной функции O(ÆM). Как и другие экспоненциальные функции, эта функция растет быстрее, чем полиномиальные функции, но медленнее, чем функция факториала.

Рассмотрим, например, следующую задачу: по заданному номеру *n* найти *n*-ое число в последовательности чисел Фибоначчи. Напомним, что первые два числа в последовательности чисел Фибоначчи равны единице, а каждое из следующих чисел равно сумме двух предыдущих. Другими словами, если обозначить *n*-ое число Фибоначчи через *fn*, то можно выписать следующие формулы:

*f*1 = *f*2 = 1

*fn*+2 = *fn* + *fn*-1

Это простое рекурсивное определение можно положить в основу функции, которая будет решать поставленную задачу, то есть находить число Фибоначчи по его номеру в последовательности:

fib :: Integer -> Integer

fib 1 = 1

fib 2 = 1

fib n = fib (n-1) + fib (n-2)

В этом решении в главном уравнении, определяющем поведение функции, производятся два рекурсивных обращения к той же самой функции. Это приводит к тому, что число обращений к функции резко возрастает при увеличении значения аргумента. Если 30-е число Фибоначчи еще можно получить сравнительно быстро (современный персональный компьютер справляется с задачей за несколько секунд), то уже нахождение 40-го числа требует нескольких часов работы! С увеличением аргумента время, требующееся для решения задачи, растет экспоненциально. Конечно, это совершенно непроизводительные затраты времени. Причина столь плохого поведения программы состоит в том, что одни и те же значения чисел Фибоначчи вычисляются в ней по многу раз. В то же время очевидно, что можно найти решение, которое будет выдавать результат за линейное относительно величины аргумента время.

Очевидно, что для эффективного решения функции просто не хватает информации, и поэтому она вынуждена при каждом вызове независимо находить уже найденные ранее значения чисел Фибоначчи. Гораздо эффективнее будет работать функция, которая имеет дополнительную информацию в виде дополнительных аргументов функции. Например, можно легко определить функцию, которая вычисляет *n*-ое число Фибоначчи, зная, что уже вычислены первые *k* чисел Фибоначчи и имея значения хотя бы двух последних вычисленных чисел. Можно определить такую вспомогательную функцию, которая будет иметь четыре аргумента: n - номер числа Фибоначчи, которое надо вычислить, k - номер последнего вычисленного числа Фибоначчи, fk - значение *k*-го числа Фибоначчи и fk1 - значение предыдущего числа Фибоначчи. Тогда с помощью этой вспомогательной функции легко определить и требуемую нами изначально функцию. Новый вариант программы может выглядеть так:

-- fib - основная функция вычисления n-го числа Фибоначчи

fib :: Integer -> Integer

-- fib' - вспомогательная функция с дополнительными аргументами

fib' :: Integer -> Integer -> Integer -> Integer -> Integer

fib' n k fk fk1 | k == n = fk

| k < n = fib' n (k+1) (fk+fk1) fk -- вычисление очередного числа Фибоначчи и рекурсивный вызов

fib 1 = 1

fib n = fib' n 2 1 1 -- первоначально известны первые два числа Фибоначчи

Описанный вариант программы мгновенно вычисляет не только 40-е, но и 100-е и даже 1000-е число Фибоначчи. Правда, последний результат содержит 209 десятичных цифр.

В качестве примера использования «ленивых» структур данных давайте рассмотрим создание

бесконечной последовательности чисел Фибоначи:

(

defn

fibo

([] (

concat

[1 1] (fibo 1 1)))

([a b]

(

let

[n (+ a b)]

(

lazy-seq

(

cons

n (fibo b n))))))

В данном случае мы определяем функцию, которая при запуске без аргументов создает начальную

последовательность из чисел 1 и 1 и затем вызывает сама себя, передавая эти числа в качестве аргу-

ментов. А функция двух аргументов обернута в вызов

lazy-seq

, который производит вычисле-

ние следующих чисел Фибоначи. При этом мы можем определить переменную, которая, например,

будет содержать первые сто миллионов чисел Фибоначи:

user> (

def

many-fibs (

take

100000000 (fibo)))

#’user/many-fibs

но поскольку мы работаем с «ленивыми» последовательностями, то значение будет создано мгно-

венно, без вычисления всех чисел Фибоначи. А само вычисление чисел будет происходить по мере

надобности. Например, мы можем получить 55-е число Фибоначи с помощью следующего кода:

user> (

nth

many-fibs 55)

225851433717