Министерство образования Республики Беларуси

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ МАКСИМА ТАНКА»

На правах рукописи
УДК 004(07)

Бахорина
Александра Павловна

РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ И КОНЕЧНЫЕ АВТОМАТЫ В РЕШЕНИИ ЗАДАЧ ИНФОРМАТИКИ

Диссертация на соискание академической степени
магистра педагогических наук
по специальности 1-08 80 02 теория и методика обучения и воспитания (информатика)

Научный руководитель:
кандидат физико-математических наук, доцент
Пономаренко В.К.

Минск 2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Бурное развитие цифровой техники и потребности математического моделирования динамических систем различной природы вызывают неослабевающий интерес к логико-динамическим моделям с дискретным временем и/или состоянием, методам их исследования и программным средствам автоматизации этих исследований.

Частным классом таких моделей являются автоматные модели с логическими функциями переходов, относимые к числу важных видов управляющих систем. Ими могут описываться технические системы управления, вычислительные системы и устройства, физические среды, в которых реализуются электромагнитные волновые, биологические и другие процессы, а также явления, протекающие в экономике и жизни общества [27].

Автоматное программирование является разновидностью синхронного программирования, которое было создано и нашло применение в Европе для разработки систем управления ответственными объектами, к которым предъявляются высокие требования по качеству программного обеспечения. Автоматная технология является основой технологии алгоритмизации и программирования задач логического управления, обеспечивающей повышение "безопасности" программного обеспечения [32]. Автоматное программирование позволяет решать практически любые сложные циклические задачи с минимальными затратами на отладку [29]. Автоматный подход позволяет также разделять работу и ответственность, легко и корректно вносить изменения в алгоритмы и программы, обеспечивает наиболее компактное и формальное описание поведения программ [27], что весьма актуально в настоящее время.

Switch-технология (автоматное программирование) [34], предназначенная для алгоритмизации и программирования систем со сложным поведением разрабатывается с 1991 года в России как альтернатива громоздким и ненаглядным автоматным языкам SDL (Specification and Description Language) и UML (Unified Modeling Language) [2].

Актуальность разработки автоматной технологии определяется необходимостью создания для различных типов вычислительных устройств и языков программирования единого подхода к формальному и изоморфному построению алгоритмов и программ.

Возрождение интереса к конечным автоматам связано с ростом сложности программных систем. Следует также отметить, что процесс проектирования при автоматном подходе, носит более строгий характер, т.к. опирается на эффективный аппарат теории конечных автоматов. Использование автоматов упрощает формализацию спецификации программы, определяющей ее поведение и играющей "ключевую роль в вопросе сдерживания программных ошибок" [30].

Конечные автоматы применяются и при описании общих свойств языков программирования [21], и при разработке компьютерных программ [34], и при проектировании самих компьютеров [5].

В исследованиях представленной диссертационной работы остановимся на применении автоматного подхода в программировании на примере задачи о ханойских башнях.

В связи с тем, что в большинстве учебных программ средней школы не предусмотрена сдача промежуточного или итогового экзамена по предмету «Информатика», и, учитывая то, что у большинства учащихся имеются в наличии собственные персональные компьютеры, возникает проблема их низкой заинтересованности в указанном предмете. Современных школьников трудно чем-либо удивить и еще более трудно убедить учащихся изучать теоретические основы информатики, в соответствии с требованиями Министерства образования Республики Беларусь.

В данной работе была сделана попытка выявления и ликвидации пробелов существующей программы по информатике. В качестве средства решения данной проблемы проведено исследование, направленное на

определение наиболее «остро» стоящих проблем в школьном курсе обучения данному предмету, разработана программа курса по выбору для учащихся и проведен эксперимент по её внедрению в школьный курс информатики.

В ходе выполнения работы проводилось три этапа исследований:

1. Статистическое изучение ситуации и анализ данных по вопросу преподавания дисциплины «Информатика» в средней школе.

2. Теоретическая разработка новой программы и методического руководства к ней курса по выбору учащихся по дисциплине «Информатика».

3. Материал, входящий в курс по выбору для школьников апробировался на внеклассных занятиях учащихся Экономико-правовой гимназии.

Как показали опросы, проведенные в ряде средних школ г. Минска, общая подготовка учащихся по предмету «Информатика» находится на невысоком уровне. В большинстве случаев это связано с рядом причин:

во-первых, плохой материально-технической базой учебных заведений (устаревшие модели персональных компьютеров), не позволяющей устанавливать последние версии программного обеспечения и быстро решать поставленные задачи;

во-вторых, с малым объёмом выделенных часов по данному предмету (один раз в неделю), приводящему к большим разрывам в обучении и, как следствие, к потере знаний у школьников;

в-третьих, отсутствует игровая составляющая учебного процесса, связанная, в том числе, с возможностью изучения внутреннего устройства самого персонального компьютера;

в-четвёртых, не рассматриваются вопросы, связанные с изучением основ программирования на примере наиболее простых языков, таких как Паскаль, знание которых требуется при участии школьников в городских и республиканских олимпиадах;

в-пятых, отсутствует мотивирующая составляющая предмета, в ходе которой школьник смог бы осознать его важность и неотъемлемость при решении многих актуальных задач в различных областях науки и техники.

Противоречием в преемственности обучения является широкое изучение темы «Рекуррентные соотношения» в различных направлениях вузовского курса (теории вероятностей, вычислительных методах, комбинаторике, теории сложности алгоритмов) и низким уровнем применения в рамках школьного курса (РС применяются при работе с арифметическими и геометрическими прогрессиями, само понятие рекуррентных соотношений в школе не вводится).

Выход из сложившейся ситуации, по мнению автора, возможен за счёт устранения разрыва между широким применением рекуррентных соотношений в вузовской программе и использованием данных закономерностей в школьном обучении. Имеется возможность подготовки учащихся к дальнейшей работе с рекуррентными соотношениями, познакомив их с основными понятиями и методами решения, пользуясь методом «От задачи» (как известно, наиболее эффективным при слабой подготовке и заинтересованности учащихся, что наблюдается повсеместно в настоящее время, тем самым подчеркивая актуальность предложенной разработки).

Конечные автоматы, являющиеся базой программирования (выполнение любой программы является системой из двух взаимодействующих автоматов: компьютера, и самой программы) и проектирования самих компьютеров, совершенно игнорируются в обучении информатике на различных уровнях, при этом уделяя большое количество времени на системы счисления.

Данная работа является попыткой связать на элементарном уровне рекуррентные соотношения и конечные автоматы.

Магистерская диссертация представляет собой исследование, направленное на восполнение пробелов школьной программы по информатике,

и на ликвидацию несоответствия требований действующей программы к школьникам и уровнем подготовки учащихся.

Повышение заинтересованности учащихся к предмету «Информатика» предлагается путем использования метода соперничества (иллюстрации программных средств, созданных другими учащимися и наглядное доказательство того, что данная задача «по силам» учащимся, постановка столь же интересных и реальных задач перед учащимися, акцентуалиация важности решения данных задач и разработка процедуры поощрения).

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

1. Связь работы с тематикой научных исследований кафедры

Тема полностью соответствует утвержденному плану научно-исследовательских работ кафедры.

2. Цель и задачи исследования

Цели диссертационной работы:

- изложение основных положений технологии автоматного программирования применительно к решению задач информатики, и, в частности, рекуррентных соотношений;
- устранение пробелов школьной программы по информатике;
- ликвидация несоответствия требований к школьникам действующей программы и уровнем подготовки учащихся путем разработки программы факультативного курса информатики на углубленном уровне изучения;
- проектирование системы задач школьного курса информатики и их реализация с применением рекуррентных соотношений;
- систематизация и накопление знаний учащихся по предмету Информатика.

Основные задачи исследования, поставленные для решения вышеизложенных целей, состоят в следующем.

1. Дать понятие рекуррентных соотношений и конечных автоматов.

2. Показать связь между рекуррентными соотношениями и конечными автоматами.

3. Провести исследование, направленное на выявление пробелов школьной программы по информатике.

4. Провести обзор школьных задач по информатике, в том числе и олимпиадных.

5. Проанализировать применение рекуррентных соотношений в задачах информатики.

6. Применить автоматную реализацию для вычислительных алгоритмов, которая более наглядна по сравнению с классическими решениями.

7. Сделать обзор существующих решений задачи о Ханойских башнях, создать объемный визуализатор работы данной игры.

8. Повысить заинтересованность учащихся к предмету Информатика.

Объектом исследования являются задачи школьного курса информатики.

3. **Положения, выносимые на защиту**.

В работе получены следующие результаты, которые выносятся на защиту:

1. Для ряда вычислительных алгоритмов использована автоматная реализация, которая более наглядна по сравнению с классическими решениями.

2. Выявлена связь между рекуррентными соотношениями и конечными автоматами.

3. Проведен анализ применения рекуррентных соотношений в задачах информатики.

4. Создан объемный визуализатор игры Ханойские башни.

5. Разработаны и применены на практике методические предложения по программе факультативного курса информатики на углубленном уровне изучения.

4. Апробация результатов диссертации

Результаты проведенной работы обсуждались на заседании кафедры Прикладной математики и информатики (протокол № 16 от 17.05.2007 г.), в результате чего была получена выписка о допуске к защите диссертации.

Материал, входящий в курс по выбору для школьников апробировался на внеклассных занятиях учащихся Экономико-правовой гимназии. До начала занятий и после окончания учебного года предложенный курс по выбору обсуждался на педагогических советах гимназии и получил положительную оценку с предложением продолжить применение на практике выдвинутых методических предложений по проведению факультативного курса.

5. Структура и объем диссертации

Диссертация изложена на страницах и состоит из введения, общей характеристики работы, трех глав, заключения, библиографического списка. Количество использованных библиографических источников содержит 39 наименований. Работа иллюстрирована 11 рисунками. В заключении приводится перечень результатов, выносимых на защиту. Приложения содержат методологическое предложение о проведении курса по выбору учащихся, акт о внедрении результатов исследования, а также созданный визуализатор игры Ханойские башни.

ГЛАВА 1
РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ И ИХ РЕШЕНИЕ

Рекуррентные соотношения возникают в различных разделах математики, в том числе и прикладной: комбинаторике, теории сложности алгоритмов, вычислительных методах и теории вероятностей.

Магистерская диссертация представляет собой исследование, направленное на восполнение пробелов школьной программы по информатике и на ликвидацию несоответствия требований к школьникам действующей программы и уровнем подготовки учащихся.

Методы исследования. В работе использованы методы анализа теоретической информации, дискретной математики, построения и анализа алгоритмов, теории автоматов.

1.1 Рекуррентные соотношения

Соотношения, связывающие значение функции решения исходной задачи от функций подзадач называются рекуррентными соотношениями (рекуррентными уравнениями) [22]. Некоторое количество начальных значений функции, входящей в соотношение, должно быть задано.

Если определяем значение рекуррентной функции для натуральных значений индекса, имеем рекуррентную последовательность.

Простейшими примерами рекуррентных последовательностей являются арифметическая и геометрическая прогрессии, последовательность чисел Фибоначчи.

1. Формула для нахождения общего члена геометрической прогрессии задается функцией

$$b_n = b_{n-1} * q \qquad (1.1)$$

арифметической прогрессии – функцией

$$a_n = a_{n-1} + d \quad (1.2)$$

В этих формулах при вычислении последующего члена последовательности возвращаются к значениям предшествующих членов. Название "рекуррентный" происходит от французского recurrent – возвращающийся (к началу) [22].

Для описания общей ситуации рассмотрим натуральное число k и произвольную функцию $F = F(n, x_1, x_2, ..., x_k)$, определенную при всех неотрицательных целых числах n и для всех вещественных (либо комплексных) значений переменных x_i.

Определение 1. Рекуррентным соотношением для функции одной переменной y(n) называется равенство вида:

$$y(n + k) = F(n, y(n), y(n + 1), ..., y(n + k – 1)), \quad (1.3)$$

где F – некоторая функция от k+1 аргумента.

Натуральное число k называется порядком рекуррентного соотношения [19].

Так, для рекуррентных геометрических последовательностей (1.1), имеем порядок k=1.

Для арифметической прогрессии имеем:

$$a_{n+1} = a_n + d, \quad (1.4)$$

вычитая равенство (1.2) из (1.4), и выполнив преобразования, получим:

$$a_{n+1} = 2*a_n - a_{n-1}.$$

Откуда следует, что арифметическая прогрессия – рекуррентное соотношение 2-го порядка [39].

2. Рекуррентные соотношения, с одной стороны, определяют ту или иную последовательность, а с другой – для конкретной заданной последовательности

контролируют многие ее свойства. Классическим примером последовательности, задаваемой рекуррентным соотношением, являются так называемые *обобщенные числа Фибоначчи* или *p-числа Фибоначчи* [8], которые для данного целого p (p=0, 1, 2, 3, ...) задаются с помощью следующего рекуррентного соотношения:

$$Fp(n) = Fp(n\text{-}1)+Fp(n\text{-}p\text{-}1) \qquad (1.5)$$

Заметим, что при различных начальных условиях:

$$Fp(1) = a1;\ Fp(2) = a2\ ;\ ...;\ Fp(p+1) = ap+1$$

мы получим из (1.5) бесконечное множество рекуррентных числовых последовательностей, которые относятся к классу *рекуррентных p-рядов Фибоначчи.*

В частности, если принять

$$Fp(1) =1;\ Fp(2) = 1;\ ...;\ Fp(p) = 1;\ Fp(p+1) = 1,$$

то при таких начальных условиях рекуррентная формула (1.5) «генерирует» класс p-чисел Фибоначчи, являющихся так называемыми «диагональными суммами» треугольника Паскаля [26].

Заметим, что при различных p основное рекуррентное соотношение (1.5) «генерирует» ряд замечательных числовых последовательностей, широко используемых в математике. Например, при p=0 рекуррентная формула (1.5) вырождается в следующую формулу:

$$F0(n) = 2F0(n\text{-}1),$$

которая при начальном условии

$$F0(1) = 1$$

«генерирует» двоичный ряд чисел: 1, 2, 4, 8, 16, 32, ...

При p=1 основное рекуррентное соотношение (1.5) принимает вид:

$$F1(n) = F1(n\text{-}1)+F1(n\text{-}2). \qquad (1.6)$$

Эта рекуррентная формула при начальных значениях

$$F1(1) = 1;\ F1(2) = 1$$

«генерирует» классический ряд Фибоначчи $F(n)$: 1, 1, 2, 3, 5, 8, 13, 21, ...

Замечательные числа Фибоначчи произошли от древней задачи (1228 год) о популяции кроликов и неожиданным образом оказались пригодными в качестве системы счисления, которая более помехоустойчива, чем двоичная система в компьютерных приложениях [26].

3. Для определения рекуррентного соотношения первого порядка необходимо задать некоторую функцию f (n, x). Тогда вся последовательность определяется однозначно выражением x(n + 1) = f (n, x(n)), n≠0, при заданном значении x(0). Если функция f не зависит явно от дискретной переменной n, то мы имеем дело с итерациями одной функции f = f (x). Именно: x(1) = = f (x(0)), x(2) = f (x(1)) = f (f (x(0))) и т.д. Естественным образом возникает вопрос о стабилизации последовательности x(n) при неограниченном возрастании номера n – порядка итерации. Последнее означает, что последовательность x(n) имеет предел x* при n→∞. Здесь мы имеем дело с уравнением x = f (x), и члены стабилизирующейся последовательности дают приближенные значения решения x* этого уравнения.

Для конкретно выбранной функции f(x), определенной на области своих значений, поведение последовательности, порождаемой соотношением

$x_{n+1}=f(x_n)$, $n\neq0$, в значительной степени зависит от начальной точки итераций x_0. Здесь мы имеем дело с динамической системой при дискретном времени n.

4. Удивительным образом рекуррентные соотношения проявляются при исследовании различных явлений природы. Вот пример из астрономии. Приняв среднее расстояние от Земли до Солнца за 1 (1 а.е.), в соответствии с соотношением Тициуса-Боде [8] расстояния d других планет от Солнца можно найти по правилу d(n + 1) = 2d(n) – 0,4, d(1) = 1. Наблюдается поразительное для космических масштабов совпадение чисел.

1.2 Решение рекуррентных соотношений

Если задано рекуррентное соотношение ***k***-го порядка, то ему удовлетворяет бесконечно много последовательностей. Дело в том, что первые ***k*** элементов последовательности можно задать совершенно произвольно - между ними нет никаких соотношений. Но если первые ***k*** элементов заданы, то все остальные элементы определяются совершенно однозначно – элемент *f(k+1)* выражается в силу рекуррентного соотношения через *f(1),…, f(k)* элемент *f(k+2)* - через *f(2),…, f(k+1)* и т.д.

Некоторая последовательность является решением данного рекуррентного соотношения, если при подстановке этой последовательности соотношение тождественно выполняется.

1.3 Линейные рекуррентные соотношения с постоянными коэффициентами

Для решения рекуррентных соотношений общих правил, вообще говоря, нет. Однако существует весьма часто встречающийся класс соотношений, решаемый единообразным методом. Это - рекуррентные соотношения вида

$$f(n+k) = a_1 f(n+k-1) + a_2 f(n+k-2) + \ldots + a_k f(n), \qquad (1.7)$$

где $a_1, a_2, \ldots, a_k$ - некоторые числа. Такие соотношения называют линейными рекуррентными соотношениями с постоянными коэффициентами [19].

Сначала рассмотрим, как решаются такие соотношения при k=2, то есть изучим соотношение вида:

$$f(n+2) = a_1 f(n+1) + a_1 f(n). \qquad (1.8)$$

Решение этих соотношений основано на следующих двух утверждениях.

Утверждение 1. Если $f_1(n)$ и $f_2(n)$ являются решениями рекуррентного соотношения (1.11), то при любых числах ***A*** и ***B*** последовательность

$$f(n) = Af_1(n) + Bf_2(n)$$

также является решением этого соотношения.

В самом деле, по условию, имеем

$$f_1(n+2) = a_1 f_1(n+1) + a_2 f_1(n)$$
$$f_2(n+2) = a_1 f_2(n+1) + a_2 f_2(n)$$

Умножим эти равенства на ***A*** и ***B*** соответственно и сложим полученные тождества. Получим, что

$$Af_1(n+2) + Bf_2(n+2) = a_1[Af_1(n+1) + Bf_2(n+1)] + a_2[Af_1(n) + Bf_2(n)].$$

А это означает, что $Af_1(n) + Bf_2(n)$ является решением соотношения (1.11).

Утверждение 2. Если r_1 является корнем квадратного уравнения

$$r^2 = a_1 r + a_2,$$

то последовательность

$$1, r_1, r_1^2, \ldots, r_1^{n-1}, \ldots$$

является решением рекуррентного соотношения

$$f(n+2) = a_1 f(n+1) + a_2 f(n).$$

Заметим, что наряду с последовательностью $\{r_1^{n-1}\}$ любая последовательность вида

$$f(n) = r_1^{n+m}, n = 1, 2, \ldots$$

также является решением соотношения (1.8). Для доказательства достаточно использовать утверждение 1, положив в нем $A = r_1^{m+1}, B = 0.$

Из утверждений 1 и 2 вытекает следующее *правило решения линейных рекуррентных соотношений второго порядка с постоянными коэффициентами*.

Пусть дано рекуррентное соотношение

$$f(n+2) = a_1 f(n+1) + a_2 f(n) \qquad (1.9)$$

Составим квадратное уравнение

$$r^2 = a_1 r + a_2, \qquad (1.10)$$

которое называется характеристическим для данного соотношения. Если это уравнение имеет два различных корня r_1, r_2, то общее решение соотношения (1.9) имеет вид

$$f(n) = C_1 r_1^{n-1} + C_2 r_2^{n-1}.$$

Пример на доказанное правило.

Числа Фибоначчи задаются рекуррентным соотношением:

$$f(n) = f(n-1) + f(n-2). \qquad (1.11)$$

Для него характеристическое уравнение имеет вид

$$r^2 = r + 1.$$

Корнями этого квадратного уравнения являются числа

$$r_1 = \frac{1+\sqrt{5}}{2}, r_2 = \frac{1-\sqrt{5}}{2}.$$

Поэтому общее решение соотношения Фибоначчи имеет вид

$$f(n) = C_1(\frac{1+\sqrt{5}}{2})^n + C_2(\frac{1-\sqrt{5}}{2})^n. \qquad (1.12)$$

Мы называли числами Фибоначчи решения соотношения (1.12), удовлетворяющее начальным условиям f(0)=1, f(1)=1. Ясно, что последовательность 1,1,2,3,5,8,13,…. удовлетворяет тому же самому рекуррентному соотношению (1.14) и начальным условиям f(0)=1, f(1)=2. Полагая в формуле (1.13) n=0, n=1 , получаем для C_1 и C_2 систему уравнений

$$C_1 + C_2 = 0$$

$$\frac{\sqrt{5}}{2}(C_1 - C_2) = 1$$

Отсюда находим:

$$f(n) = \frac{1}{5}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]. \qquad (1.13)$$

На первый взгляд кажется удивительным, что это выражение при всех натуральных значениях ***n*** принимает целые значения.

Рекуррентные соотношения возникают также при решении разнообразных задач, в том числе и занимательного характера.

Выводы

1.1 Рекуррентные соотношения исследуются в теории вероятностей, вычислительных методах, комбинаторике, и других разделах вузовского курса, но на уровне школы обнаружен большой пробел.

1.2 Данная работа является первой попыткой восполнить имеющиеся пробелы по указанному направлению в школьном курсе информатики. Выход из сложившейся ситуации, по мнению автора, возможен за счёт устранения разрыва в преемственности обучения путем введения курса по выбору учащихся, направленного на ликвидацию указанных пробелов.

1.3 Имеется возможность подготовки учащихся к дальнейшей работе с рекуррентными соотношениями, познакомив их с основными понятиями и методами решения, пользуясь методом «От задачи» (как известно, наиболее эффективным при слабой подготовке и заинтересованности учащихся, что наблюдается повсеместно в настоящее время, тем самым подчеркивая актуальность предложенной разработки).

ГЛАВА 2

ТЕХНОЛОГИЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

Автоматный подход – быстро развивающееся направление программирования, являющееся на сегодняшний момент одним из наиболее актуальных. Автоматное программирование находится в постоянном движении во всем мире, и в том числе, в России. Появляется все больше публикаций по данной тематике. Однако, к сожалению, несмотря на достаточно сильные научные школы стран СНГ, наблюдается огромный дефицит литературы по этим направлениям.

Автоматное программирование позволяет решать практически любые сложные циклические задачи с минимальными затратами на отладку. Автоматный подход позволяет также разделять работу и ответственность, легко и корректно вносить изменения в алгоритмы и программы, обеспечивает наиболее компактное и формальное описание поведения программ, что весьма актуально в настоящее время.

До последнего времени методология автоматного программирования задач разнообразного назначения в литературе стран СНГ практически не освещалась. Тем временем, на Западе эта методология активно развивалась, в первую очередь Харелом [2].

На постсоветском пространстве данным вопросом занимается ограниченный круг исследователей, среди которых доктор технических наук, профессор Шалыто Анатолий Абрамович и его аспиранты. Статья Технология автоматного программирования, написанная А.А. Шалыто, стала тем толчком, который дал старт моим исследованиям.

2.1 Парадигма автоматного программирования

Switch-технология — технология для поддержки автоматного программирования (технология автоматного программирования), была предложена А.А. Шалыто в 1991 году [34]. При этом под термином

"автоматное программирование" понимается не только построение и реализация конечных автоматов для использования в программах, но и проектирование и реализация программ в целом, поведение которых описывается автоматами.

Основная идея излагаемого подхода состоит в том, что программы предлагается создавать так же, как производится автоматизация технологических (и не только) процессов.

При этом на основе анализа предметной области выделяются источники входных воздействий и автоматизированные объекты, каждый из которых содержит систему управления (систему взаимодействующих конечных автоматов) и объект управления. Этот объект реализует выходные воздействия и формирует значения второй разновидности входных воздействий, которые от него передаются по обратным связям к системе управления.

Объект управления может быть физическим или реализованным программно. В первом случае его логика изменена быть не может, а во втором - она, при необходимости, практически вся может быть вынесена в автоматы. Виртуальные объекты управления, как и система управления, могут быть реализованы с помощью рассматриваемой технологии.

Парадигма автоматного программирования состоит в представлении программ как систем автоматизированных объектов.

2.2 Автоматы и программирование

Конечный автомат — в теории алгоритмов математическая абстракция, позволяющая описывать пути изменения состояния объекта в зависимости от его текущего состояния и входных данных, при условии что общее возможное количество состояний конечно. Конечный автомат является частным случаем абстрактного автомата.

Существуют различные варианты задания конечного автомата. Например, конечный автомат может быть задан с помощью пяти параметров: $\boldsymbol{M = (Q, \Sigma, \delta, q0, F)}$ где:

- Q — конечное *множество состояний* автомата;
- q0 — *начальное состояние* автомата ($q0 \in Q$);
- F — множество *заключительных* (или *допускающих*) состояний, таких что $F \subset Q$. При достижении одного из этих состояний работа автомата прекращается;
- Σ — допустимый *входной алфавит* (конечное множество допустимых входных символов), из которого формируются строки, считываемые автоматом;
- δ — заданное отображение множества $Q \times \Sigma$во множество подмножеств P(Q) $\delta : Q \times \Sigma \to Q$(иногда δ называют *функцией переходов автомата*).

Автомат начинает работу в состоянии q0, считывая по одному символы входной строки. Считанный символ переводит автомат в новое состояние из Q в соответствии с функцией переходов.

Функционирование автомата состоит в порождении двух последовательностей: последовательности очередных состояний автомата $s_1[1]s_2[2]s_3[3]\ldots$ и последовательности выходных символов $y_1[1]y_2[2]y_3[3]\ldots$, которые для последовательности символов $x_1[1]x_2[2]x_3[3]\ldots$ разворачиваются в моменты дискретного времени t = 1, 2, 3, … Моменты дискретного времени получили название тактов.

Функционирование автомата в дискретные моменты времени t может быть описано системой рекуррентных соотношений:

$$s(t+1) = \delta(s(t), x(t));$$

$$y(t) = \lambda(s(t), x(t)).$$

По входным воздействиям автоматы выполняют переходы между состояниями, которые должны быть явно выделены, и формируют в состояниях и/или на переходах выходные воздействия, реализуемые в объектах управления. Такой взгляд на программирование поведения компонент является естественным для решения задач управления различных уровней.

В дискретной математике, разделе информатики, теория автоматов изучает абстрактные машины в виде математических моделей, и проблемы, которые они могут решать. Теория автоматов наиболее тесно связана с теорией алгоритмов. Это объясняется тем, что автомат преобразует дискретную информацию по шагам в дискретные моменты времени и формирует результирующую информацию по шагам заданного алгоритма. Эти преобразования возможны с помощью технических и/или программных средств. Автомат можно представить как некоторое устройство, на которое подаются входные сигналы и снимаются выходные и которое может иметь некоторые внутренние состояния. При анализе автоматов изучают их поведение при различных возмущающих воздействиях и минимизируют число состояний автомата для работы по заданному алгоритму.

Время в автоматах в явном виде не используется. Элементы задержки рассматриваются как объекты управления. При этом задержки запускаются и сбрасываются из автоматов, а информация об истечении времени поступает в автоматы в виде входных воздействий [7].

2.3 Состояния

Состояния делятся на два типа: управляющие (автоматные) и вычислительные (неавтоматные) [30]. При этом устройство управления компьютера с относительно небольшим числом состояний управляет памятью с огромным числом состояний, в которой хранятся результаты вычислений.

Идея разделения состояний на управляющие и вычислительные может быть перенесена в практическое программирование. Так, например, в задаче о ханойских башнях, несмотря на то, что число вычислительных состояний объекта управления (три стержня и n дисков) растет как экспонента от n, автомат, обеспечивающий перекладку дисков, имеет всего лишь два или три управляющих состояния.

Число состояний в автомате определяется числом управляющих состояний в автоматизируемом объекте. Входные воздействия переводят автомат из одного состояния в другое, а выходные воздействия, формируемые в новом состоянии автомата, переводят объект в соответствующее состояние. Если разные состояния объекта управления можно поддержать с помощью одного и того же состояния автомата, то число состояний в нем может быть уменьшено. Поэтому автомат может иметь меньше состояний, чем управляемый им объект. При программной реализации минимизировать состояния в построенном автомате обычно нецелесообразно, так как это может разрушить "образ" графа переходов, сложившийся у разработчика, не приводя ни к каким существенным улучшениям. Явное выделение состояний позволяет обеспечить "устойчивость" программ.

2.4 Схемы связей и графы переходов

В качестве основного документа, определяющего структуру программы, в автоматном программировании используется схема связей [5]. Она может совмещаться со схемой взаимодействия автоматов. Схема связей определяет интерфейс автоматов и позволяет применять в графах переходов и в реализующих их программах символьные обозначения.

Поведение автоматов задается графами переходов [5] (диаграммами состояний), на которых для их компактности входные и выходные воздействия обозначаются символами, а слова используются только для названий пронумерованных состояний. Применение символов позволяет изображать

сложные графы переходов весьма компактно — так, что человек может в большинстве случаев охватить каждый из них одним взглядом. Графы переходов в наглядной для человека форме отражают переходы между состояниями, а также "привязку" выходных воздействий и других автоматов к состояниям и/или переходам.

В программе, построенной по графам переходов, также используются символьные обозначения. Это связано с тем, что текст программы в рамках указанной технологии строится по графу переходов формально и изоморфно, а изменения вносятся не непосредственно в текст программы, а только после корректировки схемы связей и графа переходов, что позволяет обеспечить синхронность изменений программ и их проектной документации, упростить отладку и моделирование автомата.

Возможность постоянного чтения на дисплее десятичного номера состояния каждого графа переходов в каждом программном цикле с помощью всего лишь одной многозначной переменной делает программу полностью наблюдаемой и управляемой, что принципиально отличает предлагаемую технологию от традиционных технологий программирования.

2.5 Основные принципы автоматного программирования

Если в программировании в настоящее время широко используется понятие «событие», то рассматриваемый подход базируется на понятии «состояние». Добавляя к нему понятие «входное воздействие», которое может быть входной переменной или событием, вводится термин «автомат без выхода». Добавляя к последнему понятие «выходное воздействие», вводится термин «автомат» (конечный, детерминированный).

Использование понятия «состояние» в качестве базового позволяет лучше понять и специфицировать поведение программы и ее составных частей.

Автоматное программирование — стиль программирования, основанный на применении конечных автоматов для описания поведения программ, а процесс создания таких программ — «автоматное проектирование программ». В автоматном программировании конечные автоматы используются для описания поведения программ при их спецификации, проектировании, реализации, отладке, документировании и сопровождении.

Технология поддерживает как процедурный, так и объектно-ориентированный стили программирования, однако не исключает применение и иных стилей.

В настоящее время эта технология разрабатывается в нескольких вариантах, различающихся как классом решаемых задач, так и типом вычислительных устройств, на которых осуществляется программирование.

2.6 Преимущества и недостатки автоматной технологии

Использование тетрады (состояние - система взаимосвязанных графов переходов - многозначное кодирование - конструкция switch), а также независимость от глубокой предыстории (в дальнейшем "предыстории" - "будущее зависит от настоящего и не зависит от прошлого") обеспечивает наглядность, структурность, вызываемость, вложенность, иерархичность, управляемость и наблюдаемость программ, а также их изоморфизм (изобразительную эквивалентность) со спецификацией, по которой они формально строятся. Это позволяет решить проблему взаимопонимания, разделения работы и ответственности, легко и корректно вносить изменения в алгоритмы и программы.

Актуальность разработки автоматной технологии определяется, целесообразностью создания для различных типов вычислительных устройств и языков программирования единого подхода к построению алгоритмов и программ.

Продвижение в направлении решения проблемы непригодности для человеческого восприятия (большая часть программ рассчитана на механическое выполнение) для задач логического управления имеет особую важность в связи с большой ответственностью их решения для многих объектов управления (например, для ядерных или химических реакторов), а предпосылки для такого продвижения определяются наличием развитого математического аппарата теории автоматов.

Качество программ, построенных с использованием SWITCH-технологии достигается за счет выразительных средств графов переходов и изоморфного перехода от графов к программам. Эта технология успешно зарекомендовала себя при создании систем логического управления. С применением этой технологии студентами и аспирантами СПбГУ ИТМО было создано более пятидесяти проектов [24] в самых разных областях разработки программного обеспечения. Это позволило рассмотреть технологию с самых разных сторон и проанализировать ее сильные и слабые стороны.

Отметим, что SWITCH-технология обладает рядом недостатков. 1. Монолитность — в отличие от реализации на основе паттерна State Machine невозможно повторно использовать составные части кода класса ThreadFactory. 2. При необходимости добавления входных и (или) выходных воздействий могут возникать ситуации, при которых компилятор не сможет обнаружить некоторые семантические ошибки, такие как, например, несоответствие метода интерфейса класса с вызовом автомата с соответствующим событием.

В связи с использованием языков программирования высокого уровня, применяемых, например, для промышленных (управляющих) компьютеров, излагаемый подход назван SWITCH-технологией, являющейся одним из направлений широко развиваемой в настоящее время CASE - технологии (Computer Aided Software Engineering). Предлагаемая технология может быть также названа STATE-технология (state (англ.) - состояние), AUTOMATON-технология (automaton (англ.) - автомат).

2.7 Применение автоматного программирования

Применение Switch-технологии сдерживается тем, что многие считают, что область применения автоматов в программировании хорошо известна и является достаточно узкой, а их функциональность мала. Традиционно по этому поводу выдвигаются следующие соображения:

- автоматы обладают малой вычислительной мощностью, так как они позволяет описывать только регулярные языки;
- автоматы являются одной из моделей дискретной математики, которая наряду с другими применяется при необходимости.

Все эти утверждения, безусловно, верны. Однако ни одно из них не препятствует использованию автоматного программирования при создании систем со сложным поведением в различных предметных областях. Подход используется в четырех направлениях:

- логическое управление (события отсутствуют, входные и выходные переменные двоичны);
- программирование с явным выделением состояний;
- объектно-ориентированное программирование с явным выделением состояний;
- вычислительные алгоритмы (алгоритмы дискретной математики).

Автоматы применяются:

- при проектировании устройств вычислительной техники (например, счетчиков) и при программировании аппаратных реализаций на больших интегральных схемах;
- для описания в языке UML жизненного цикла объектов при моделировании объектно-ориентированных программ [20];
- при программировании компиляторов и протоколов.

- автоматная реализация интерактивных сценариев образовательной анимации с использованием Macromedia Flash;
- обучающая и тестирующая программа с примером настройки для изучения английского языка;
- совместное использование теории построения компиляторов и Switch-технологии;
- скелетная анимация;
- управление различными технологическими процессами и объектами (упрощенная модель цеха холодной прокатки, упрощенная модель автомобиля, дизель-генератор, турникет, кодовый замок, светофор, кофеварка, телефон, банкомат, лифт, система безопасности банка и т.д);
- игры ("Terrarium", "Robocode", "CodeRally", "Морской бой", "Lines" [32], "Bomber" [34], "Tron", "Однорукий бандит", "Завалинка");
- моделирование игры "Пятнашки" для роботов "LEGO";
- XML-формат для описания внешнего вида видеопроигрывателя;
- примеры клиент-серверных приложений;
- построение пользовательских интерфейсов;
- реализация сетевого протокола SMTP;
- классические "параллельные" задачи ("Синхронизация цепи стрелков", "Обедающие философы");
- управление в задачах логистики.

Для поддержки автоматного программирования создан сайт [24].

Автоматное программирование начинает все шире использоваться в рамках такого научного направления, как "искусственный интеллект" [24], и при программировании мобильных устройств [24].

Указанные выше работы в области Switch-технологии находятся в русле работ по обеспечению высокого качества программного обеспечения, проводимых в Западной Европе при создании синхронного программирования для ответственных систем [3] в *NASA* при создании программного обеспечения для беспилотных космических аппаратов [18].

Автоматный подход может использоваться и при реализации вычислительных алгоритмов. Так, в частности, в работе [28] показано, что произвольный итеративный алгоритм может быть реализован конструкцией, эквивалентной циклу do-while, телом которого является конструкция switch, реализующая автомат.

На основе автоматного программирования на кафедре «Компьютерные технологии» СПбГУ ИТМО предложен новый подход к построению визуализаторов алгоритмов [17], которые используются при обучении программированию и дискретной математике.

Выводы:

2.1 Автоматный подход – быстро развивающееся направление программирования, являющееся на сегодняшний момент одним из наиболее актуальных.

2.2 Автоматное программирование способствует решению циклических задач практически любой сложности с минимальными затратами времени на отладку.

2.3 Автоматный подход может использоваться при реализации вычислительных алгоритмов.

ГЛАВА 3
РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ И КОНЕЧНЫЕ АВТОМАТЫ В РЕШЕНИИ ЗАДАЧ ИНФОМАТИКИ

3.1 Связь рекуррентных соотношений с автоматами

Очевидно, что конструкция ветвления является самой трудной в восприятии программиста, поскольку при множестве альтернатив превращает линейную структуру алгоритма в древовидную. При этом сложность даже последовательных программ (не говоря уже о параллельных) растет стремительно и временами может превосходить дерево вариантов в столь непростой для автоматического анализа модели, как традиционные шахматы [16].

Разбиение программы на процессы и объекты с заменой многоступенчатого ветвления средствами обработки событий заменяет одну проблему на другую: вложенность уменьшается, зато количество взаимодействующих компонентов заметно возрастает. Логика "размывается" и в итоге получаем плохо контролируемую ситуацию, когда из-за хаотичности "ручного" синтеза и невозможности построить исчерпывающий набор тестов нет никакой уверенности в корректности построенной системы. Ключ к решению состоит в применении формальных методов, в создании удобной абстракции, способной "выжать" из алгоритма квинтэссенцию логики его работы и дать возможность проводить весь необходимый анализ [5].

Одними из таких удобных абстракций могут служить конечные автоматы и рекурсивные функции (основывающиеся на рекуррентных соотношениях), оказавшие существенное воздействие на становление информатики, были разработаны почти параллельно в первой половине XX и нашли свое отражение в производственных языках программирования [20].

Парадигма автоматного программирования состоит в представлении программ как систем автоматизированных объектов [34].

Автоматное программирование позволяет повысить уровень абстракции не только в части выполняемых операций, но и при описании поведения программ. Автоматы задаются рекуррентными соотношениями – следующее значение состояния зависит от предыдущего. Это позволяет применять автоматный подход на практике, не ограничиваясь сложностью алгоритма [7].

Функционирование автомата в дискретные моменты времени t может быть описано системой рекуррентных соотношений:

$$\begin{cases} s(t+1)=\delta(s(t),x(t)); \\ y(t)=\lambda(s(t),x(t)) \end{cases}$$

В рамках автоматного подхода объекты управления могут выполнять сколь угодно сложные действия, а система управления может содержать не один автомат, а систему взаимосвязанных автоматов, взаимодействующих между собой [14].

Автоматные модели с логическими функциями переходов, относимые к числу важных видов управляющих систем, являются частным классом логико-динамических моделей с дискретным временем и/или состоянием [23]. Одной из основных задач является задача анализа качественных и метрических характеристик процессов, протекающих внутри этих систем и программных средств автоматизации этих исследований.

Традиционные методы анализа логико-динамических моделей автоматного типа связаны с получением количественных оценок длительности процесса притяжения к предельным циклам и неподвижным точкам, в том

числе с использованием функций Ляпунова [6], с изменением пространственных форм конфигураций [23], а также с качественным анализом довольно ограниченного набора динамических свойств (достижимость, возвратность и т.д.) Таким образом, существующее сегодня методическое обеспечение исследования логико-динамических моделей автоматного типа требует своего дальнейшего развития в направлении расширения класса рассматриваемых динамических свойств.

Довольно общим методом качественного исследования нелинейной динамики является метод сравнения в математической теории систем, предложенный В.М. Матросова [30] и развитый в ряде работ [29, 28], при применении которого используются и символьные, и численные процедуры обработки информации. Он позволяет свести рассматриваемую задачу анализа динамического свойства к аналогичной (но более простой по замыслу) задаче для вспомогательной системы (системы сравнения).

Таким образом, анализ логико-динамических моделей автоматного типа, при применении которого используются и символьные, и численные процедуры обработки информации, *проводится при помощи рекуррентости*.

Автоматное программирование хорошо демонстрирует то, как варьируются практические методы решения логически и/или математически однородных задач.

Создание алгоритма для определенного класса задач является достаточно сложным и требует высокой математической квалификации и изобретательности. Но когда такой алгоритм построен, процесс решения любой задачи данного класса может осуществить исполнитель, способный выполнить только элементарные операции, составляющие вычислительный процесс. Таким исполнителем может быть конечный автомат.

Если в качестве класса задач взять рекуррентные соотношения, то, можно сделать вывод, что *рекуррентные соотношения реализуются при помощи конечных автоматов.*

Конечный автомат был разработан как модель вычислительного устройства [18].

Анализ производительности автоматных алгоритмов позволяет сравнить разные алгоритмы и выбрать наиболее оптимальные из них. Он также помогает оценить поведение автомата при различных условиях. Выделяя только части алгоритма, которые вносят наибольший вклад во время исполнения программы, анализ помогает определить, доработка каких участков кода позволяет внести максимальный вклад в улучшение производительности [18].

Рассмотрение вопроса об эффективности вычислимости осуществляется при помощи рекуррентных соотношений между числом действий, необходимых для выполнения над задачей, чтобы перейти к подзадаче, и количеством времени, которое необходимо затратить на выполнение данного числа элементарных операций.

Каждое продвижение в теории сложности алгоритмов для нужд биоинформатики и других наук находит практическое применение, причем сказываются даже минимальные асимптотические улучшения [11]. *Анализ производительности автоматного алгоритма, производимый при помощи рекуррентных соотношений*, является одним из наиболее важных звеньев по оптимизации программ.

Еще одним практическим применением теории автоматов является математически строгое нахождение разрешимости задач.

Рассмотрение вопроса о существовании набора значений переменных, входящих в булевскую формулу (составленную из логических переменных при помощи дизъюнкции, конъюнкции и отрицания) при котором значение всей формулы будет истинным (задача пропозициональной выполнимости (SAT)), является одним из наиболее важных при проверке схем современных процессоров, что не представляется возможным вручную [23]. Математически

проверка базовой схемы из логических компонентов записывается в виде SAT, когда решения описывающей схему (точнее - описывающей соответствие схемы модельной схеме или спецификации) формулы соответствуют ошибкам.

Сейчас существуют два основных типа алгоритмов для решения SAT: алгоритмы локального поиска, которые начинают с какого-то набора значений, а затем модифицируют его, пытаясь последовательно приблизиться к выполняющему набору, и так называемые DPLL-алгоритмы, которые обходят дерево всевозможных наборов и выполняют поиск в глубину. Анализ сложности алгоритмов локального поиска, как правило, носит вероятностный характер - ведь нужно начать с какого-то набора, который иначе как случайно выбрать трудно, а от него может зависеть очень многое. Анализ же сложности DPLL-подобных алгоритмов более детерминирован, во многом благодаря развитой Оливером Кульманом и Хорстом Люкхардтом теории, связывающей эти оценки с решением рекуррентных уравнений, - их идея оказалась столь плодотворной, что позволила даже создать программы, автоматически доказывающие новые верхние оценки сложности для основанных на этих принципах алгоритмов.

Алгоритмы, основанные на локальном поиске, выигрывают практически, а DPLL-подобные алгоритмы - теоретически, для них удается доказать более сильные верхние оценки [14]

Таким образом, *применение теории автоматов для математически строгого нахождения разрешимости проблем*, также *использует рекуррентные соотношения*.

3.2 Нахождение чисел Фибоначчи

3.2.1 Рекурсивное решение

Рассмотрим использование рекуррентных соотношений при решении задач алгоритмизации и программирования.

Используя рекуррентное соотношение (1.6), можно построить рекурсивный алгоритм вычисления чисел Фибоначчи:

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

Уравнение рекурсивно дважды вызывает функцию Fib, один раз с входным значением N-1, а другой — со значением N-2, что определяет необходимость 2 условий остановки рекурсии: Fib(0)=1 и Fib(1)=1. 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
```

3.2.1.2 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 [28].

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 рекуррентное определение очень похоже на определение чисел Фибоначчи. 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.

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 рекуррентная, но она также используется для оценки времени ее выполнения.

Чтобы 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.

Поскольку время выполнения растет очень быстро, этот алгоритм довольно медленно выполняется для больших входных значений. Фактически, настолько медленно, что на практике почти невозможно вычислить значения функции Fib(N) для N, которые намного больше 30.

3.2.2 Рекурсия с запоминанием

При вычислении значения $F(6)$ будут вызваны процедуры вычисления $F(5)$ и $F(4)$. В свою очередь, для вычисления последних потребуется вычисление двух пар $F(4)$, $F(3)$ и $F(3)$, $F(2)$. Можно нарисовать «дерево рекурсивных вызовов».

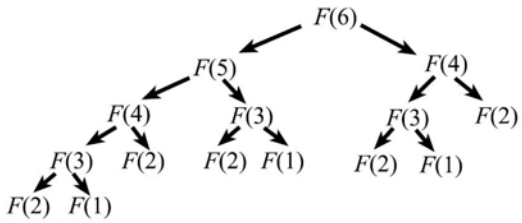
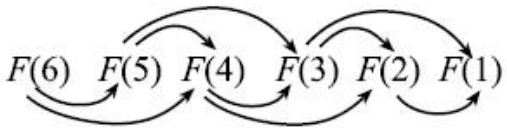


Рисунок.3.1 – Дерево рекурсивных вызовов для F_6.

Можно заметить, что $F(3)$ вычисляется три раза. Если рассмотреть вычисление $F(n)$ при больших n, то повторных вычислений будет очень много. Это и есть основной недостаток рекурсии — повторные вычисления одних и тех же значений.

Если исключить повторный счет, то функция станет заметно эффективней. Для этого приходится завести массив, в котором хранятся значения нашей функции. Однако, срабатывает «золотой закон» программирования - выигрывая в скорости, проигрываем в памяти. Сперва массив заполняется значениями, которые заведомо не могут быть значениями нашей функции. При попытке вычислить какое-то значение, программа смотрит, не вычислялось ли оно ранее, и если да, то берет готовый результат.



Функция принимает следующий вид:

```
        Function F(X : integer) : LongInt;
Begin
  if D[X] = 0 then
      if (X=1) or (X=2)
      then D[X] := 1
      else D[X] := F(x-1) + F(x-2);
  F := D[X]
End;
```

Такая *рекурсия с запоминанием* называется *динамическим программированием сверху.*

Если определить числа Фибоначчи первым способом, то время вычисления $F[40]$ будет более минуты. Если же использовать второе определение, то 209-значное число $F[1000]$ будет вычислено практически мгновенно, хотя, безусловно, и второй способ далеко не самый оптимальный.

3.2.3 Замещающееся запоминание

Есть другой способ решить проблему повторных вычислений – простой «человеческий алгоритм», не использующий рекурсивные вызовы и запоминание всех вычисленных значений. Достаточно помнить два последних числа Фибоначчи, чтобы вычислить следующее. Затем предыдущее можно «забыть» и перейти к вычислению следующего:

1. $a = b = 1$;
2. если $n > 2$, то сделать $n - 2$ раз: $c = a + b$; $a = b$; $b = c$;
3. вернуть ответ b;

3.2.3.2 Анализ времени выполнения программы

Этот алгоритм *линейный* по n, то есть для вычисления n-го числа Фибоначчи требуется n шагов. Но здесь есть важная тонкость: число знаков в

числе Фибоначчи растёт с n, соответственно, время выполнения операции сложения c=a+b тоже увеличивается. А именно, число знаков в числе Фибоначчи растёт примерно линейно (в любой системе счисления). Это следует из явной формулы: (1.16).

Конечно, пока числа не выходят за пределы машинной точности (на компьютерах с 32-битной архитектурой это означает «меньше 2^{32}»), сложение выполняется за фиксированное число тактов. Но, начиная с $F(48)$, уже нельзя использовать элементарные 32-битные целочисленные типы и нужно использовать 64-битные или представлять числа в виде массивов цифр в некоторой системе счисления и писать процедуры сложения таких чисел.

Указанный пошаговый алгоритм вычисления чисел Фибоначчи с учётом затрат на сложения длинных чисел является квадратичным по n (при увеличении n в k раз время вычисления F(n) увеличивается в k^2 раз), а не линейным как многие считают. Числа Фибоначчи в принципе нельзя подсчитать быстрее, чем за линейное время, так как, чтобы вывести цифры числа Фибоначчи F(n), уже требуется линейное по n время.

В общем случае наилучшее применение рекурсии - это решение задач, для которых свойственна одна черта: решение задачи в целом сводится к решению подобной же задачи, но меньшей размерности и, следовательно, легче решаемой.

Поскольку эта фраза звучит довольно загадочно и абстрактно, проиллюстрируем ее примером подобной задачи.

3.3 Задача о ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [9,10,31,37], которая формулируется следующим образом. Имеются три стержня, на первом из которых размещено N дисков. Диск наименьшего диаметра находится сверху, а ниже — диски последовательно увеличивающегося диаметра. Цель игры состоит в определении

последовательности перекладываний по одному диску со стержня на стержень, которые должны выполняться так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра, чтобы все диски оказались на другом стержне.

3.3.1 Классическое рекурсивное решение задачи

Сначала попытаемся построить абстрактную модель процесса переноса части пирамиды (когда мы с ней справимся, станет яснее, как перенести пирамиду целиком). Итак, решаем обобщенную задачу: как перенести пирамиду из n колец со стержня i на стержень j, пользуясь стержнем k как вспомогательным? Эта задача решается следующим образом:

1. Переложить N-1 диск со стержня с номером i на стержень с номером 6-i-j.
2. Переложить диск со стержня с номером i на стержень с номером j.
3. Переложить N-1 диск со стержня с номером 6-i-j на стержень с номером j.

Задача переноса n колец решается через перенос (n-1) кольца. Осталось только добавить тривиальное граничное условие для вырожденного случая переноса пустой пирамиды из 0 колец, чтобы алгоритм завершился:

```
1 function move (n,x,y)
2  if n=1 then
3   передвинуть кольцо с стержня x на стержень y
4  else move (n-1,x,6-x-y)
6  move (1,x,y)
7  move (n-1,6-x-y,y)
8 end if
9 end function
```

Алгоритм, записанный на псевдокоде, реализует рекурсивную идею перемещения колец в игре «Ханойские башни». Функция MOVE(n,x,y) перемещает n колец со стержня с номером x на стержень с номером y.

Указанный процесс можно изобразить с помощью дерева декомпозиции (рисунок. 3.2).

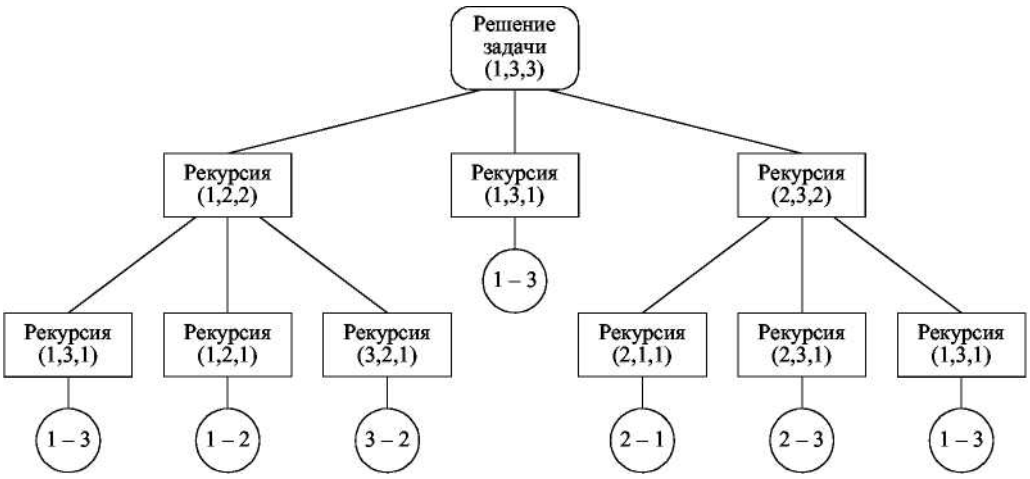


Рисунок. 3.2—Дерево декомпозиции для задачи о ханойских башнях при N = 3

В этом дереве вершины, обозначенные прямоугольниками, соответствуют подзадачам, решаемым при каждом вызове рекурсивной функции. Эти вершины помечаются номерами стержня, с которого перекладывается диск, стержня, на который перекладывается диск, и числом перекладываемых дисков. При этом для вершины с пометкой (i,j,k) левая вершина поддерева будет помечена (i,6-i-j,k-l), средняя — (ij,l), а правая — (6-i-j j,k-l).

Перекладывания дисков выполняются в вершинах, обозначенных кружками. Для каждой из этих вершин сохраняются первые две позиции пометки соответствующего прямоугольника— (i,j).

Приведем программу, написанную на объектном языке программирования Delphy и реализующую рассматриваемый рекурсивный алгоритм (листинг 1). Ее поведение эквивалентно обходу дерева декомпозиции

слева направо, причем в вершинах обозначенных кружками производятся перекладывания.

ЛИСТИНГ 1. Рекурсивная программа

```
unit hanoy;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure move(n, x, y: integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
procedure TForm1.move(n, x, y: integer);
var
  s:  String;
begin
        if n = 1 then
  begin
    s := 'Move ring from ' + inttostr(x) + ' to ' + inttostr(y);
    Memo1.Lines.Add(s);
  end
  else
  begin
        move(n-1,x,6-x-y);
          move(1,x,y);
        move(n-1,6-x-y,y);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  Memo1.Clear();
  if(Edit1.text='')  then
  begin
    Application.MessageBox('Введите число колец', 'Ошибка!!!');
    Exit;
  end;
  move(StrToInt(Edit1.text),1,2);
end;
end.
```

В эту программу введено протоколирование рекурсивных вызовов.

3.3.1.1 Сложность рекурсивного алгоритма

Получили рекурсивный алгоритм: для того, чтобы решить задачу для пирамиды из N колец, достаточно решить её для пирамиды из $N - 1$ кольца. Посчитаем теперь количество действий, необходимое для проведения всей операции. Пусть $f(N)$ — необходимое число действий, для переноса пирамиды из n колец. Для одного кольца ответ равен единице: $f(1) = 1$, для N ответ будет

$$f(N) = f(N - 1) + 1 + f(N - 1) = 2f(N - 1) + 1,$$

Тогда для N -1 получим $f(N) = 2\ (2f(N - 2) + 1) + 1 \ldots = 2^2 f(N - 2) + 2^2 - 1$

Тогда для N -i получим $f(N) = 2^i f(N - i) + 2^i - 1$

Взяв предел от данного соотношения при i→n, решим это рекуррентное соотношение и получим:

$f(N) = 2^N - 1$. А значит f(64)>1 000 000 000 000 000 000 . Таким образом, время, необходимое для перемещения пирамидки из 64 колец, очень велико.

Программу лучше запускать с маленьким числом колец (меньше 5), чтобы видеть весь процесс целиком.

Описанный процесс можно осуществить с применением автомата.

3.3.2 Автоматный подход

Рассмотрим три подхода, применение автоматов в которых позволяет либо формально переходить к итеративным алгоритмам решения этой задачи, либо строить такие алгоритмы непосредственно.

Первый из них обеспечивает формальное построение автоматной программы (программы, построенной с использованием автоматов) на основе раскрытия рекурсии с применением стека. Второй метод также обеспечивает раскрытие рекурсии и состоит в построении автомата, осуществляющего обход дерева действий, выполняемых рекурсивной программой. Третий метод состоит в непосредственном управлении дисками и стержнями, и не использует таких абстракций как деревья и стеки. Отметим, что применение каждого из предлагаемых методов для рассматриваемой задачи порождает автоматы с двумя или тремя состояниями, управляющие "объектом управления" с 2^N состояниями, возникающими в процессе перекладывания дисков.

3.3.2.1 Моделирование рекурсии автоматной программой

Идея метода состоит в раскрытии рекурсии за счет моделирования работы рекурсивной программы. При этом для хранения локальных переменных стек используется явно. Явное выделение стека по сравнению со "скрытым" его применением в рекурсии, позволяет программно задавать его размер, следить за его содержимым и добавлять отладочный код в функции, реализующие операции над стеком.

Перейдем к изложению метода.

1. Каждый рекурсивный вызов в программе выделяется как отдельный оператор. По преобразованной рекурсивной программе строится ее схема (рисунок 3.3), в которой применяются символьные обозначения

условий переходов (x), действий (z) и рекурсивных вызовов (R). Схема строится таким образом, чтобы каждому рекурсивному вызову соответствовала отдельная операторная вершина.

2. Для определения состояний эквивалентного построенной схеме автомата Мили в нее вводятся пометки, по аналогии с тем, как это выполнялось в работе [9] при преобразовании итеративных программ в автоматные. При этом начальная и конечная вершины помечаются номером 0. Точка, следующая за начальной вершиной, помечается номером 1, а точка, предшествующая конечной вершине — номером 2. Остальным состояниям автомата соответствуют точки, следующие за операторными вершинами. В рассматриваемом случае точка с номером 2 совпадает с точками, следующими за операторными вершинами R3 и z0.

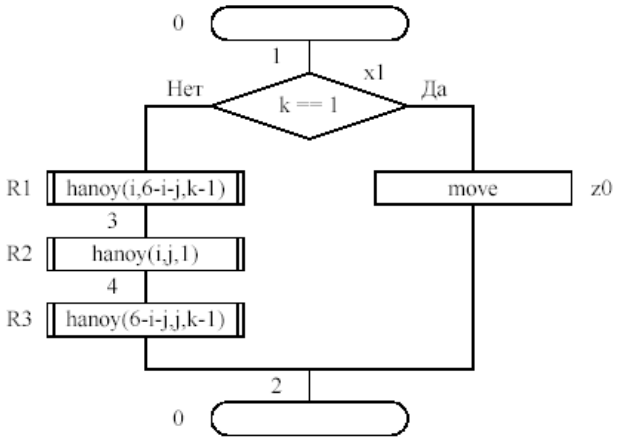


Рисунок 3.3—Схема рекурсивной функции

3. В соответствии с пометками на рис. 2 строится граф переходов автомата Мили (рис. 3).

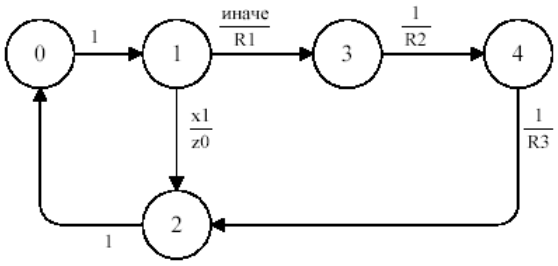


Рисунок 3.4—Граф переходов, построенный по схеме рекурсивной функции

4. Выделяются действия, совершаемые над параметрами рекурсивной функции при ее вызове. Обозначим такие действия для рекурсивных вызовов Rl, R2 и R3 как zl (j = 6-i-j; k——), z2 (k= 1) и z3 (i = 6-i-j; k——) соответственно.

5. Составляется перечень параметров и других локальных переменных рекурсивной функции, определяющий структуру ячейки стека. Если рекурсивная функция содержит более одного оператора рекурсивного вызова, то в стеке также запоминается значение переменной состояния автомата. В данном примере элемент стека содержит значение переменной состояния автомата y и значения локальных переменных i, j и k.

6. Выполняется преобразование графа переходов для моделирования работы рекурсивной функции, состоящее из трех этапов (рис. 4).

6.1.Дуги, содержащие рекурсивные вызовы (R), направляются в вершину с номером 1. В качестве действий на этих дугах указываются: операция запоминания значений локальных переменных push(s), где s — номер вершины графа переходов, в которую была направлена рассматриваемая дуга до преобразования и соответствующее действие, выполняемое над параметрами рекурсивной функции.

6.2.Безусловный переход на дуге, направленной из вершины с номером 2 в вершину с номером 0 ,заменяется условием "стек пуст" с первым приоритетом.

6.3.К вершине с номером 2 добавляются дуги, направленные в вершины, в которые до преобразования графа входили дуги с рекурсией. На каждой из введенных дуг выполняется операция pop, извлекающая из стека верхний элемент. Условия переходов на этих дугах имеют вид: stack[top].y == s, где stack[top] — верхний элемент стека, y— значение переменной состояния автомата, запомненное в верхнем элементе стека, as — номер вершины, в которую направлена рассматриваемая дуга. Таким образом, в рассматриваемом автомате имеет место зависимость от глубокой предыстории— в отличие от классических автоматов, переходы из состояния с номером 2 зависят также и от

ранее запомненного в стеке номера следующего состояния.

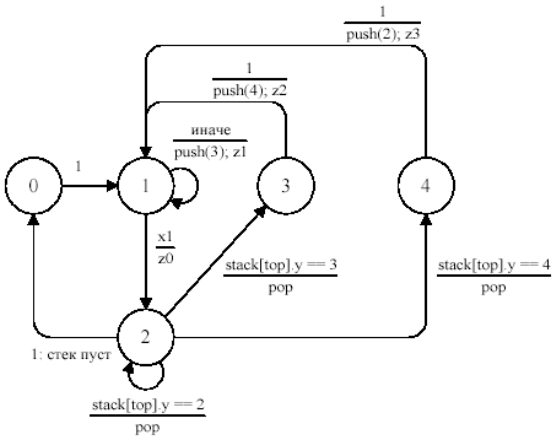


Рисунок 3.5—Граф переходов, моделирующий работу рекурсивной функции

7. Граф переходов на рисунке 3.5 упросим за счет исключения неустойчивых вершин. Такой граф переходов приведен на рисунке 3.6.

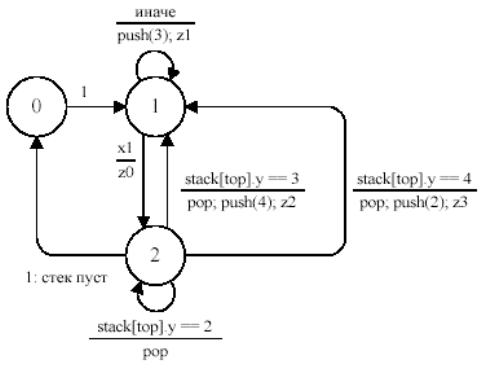


Рисунок 3.6—Упрощенный граф переходов

8. Строится автоматная программа, содержащая функции для работы со стеком и цикл do-while, телом которого является оператор switch, формально и изоморфно построенный по графу переходов [9].

Программа, построенная по графу переходов (рисунок 3.5) и содержащая функции для работы со стеком, приведена на листинге 3. В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2-0), закомментированы и заменены одним оператором pop(&y, &i, &j, &k). Такое упрощение программы, однако, приводит к тому, что по

указанному оператору невозможно определить, в какое состояние будет выполнен переход.

ЛИСТИНГ 2. Автоматная программа, моделирующая рекурсию

```
#include <stdio.h>
// Элемент стека,
typedef struct int y, i, j, k ; } stack_t ;
stack_t  stack[200] int     top = -1 ;
// Стек.
// Индекс верхнего элемента в стеке.
push ( int y, int i, int j, int k ) {
  top++ ;
  stack[top].y = y ;
  stack[top].i = i ; stack[top].j = j ; stack[top].k = k ;
printf( "push {%d,%d,%d,%d}: ", y, i, j, k ) ; show_stack() ; }

pop( int *y, int *i, int *j, int *k ) {
printf( "pop  {%d,%d,%d,%d}: ", stack[top].y, stack[top].i,
stack[top].j, stack[top].k ) ;
if( y ) *y = stack[top].y ;
*i = stack[top].i ; *j = stack[top].j ; *k = stack[top].k ;
  top-— ;
show_stack() ; }

int stack_empty() { return top < 0 ; }

// Вывод содержимого стека для протоколирования.
void show_stack()
{int i ;
  for( i = top ; i >= 0 ; i-— )
    printf( "{%d,%d,%d,%d} ", stack[i].y, stack[i].i, stack[i].j, stack[i].k
printf( "\n" ) ; }

void hanoy( int i, int j, int k ) {
  int y = 0 ;

  do
    switch( y ) {
case 0:
    y = 1 ; break ;
case 1:
            if( k == 1 ) { move( i, j ) ;  y = 2 ; }
            else
            {push( 3, i, j, k ) ; j = 6-i-j ; k— ; } break ;
case 2 :
```

```
            if( stack_empty() )       y = 0 ;
            else
              pop( &y, &i, &j, &k ) ;
            /* Эта операция pop() заменяет закомментированные
```

операторы, но при этом теряется однозначное соответствие программы графу переходов. */ /*

```
            if( stack[top].y == 4 ) {
            pop( NULL, &i, &j, &k ) ;    y = 4 ; }
            else
            if ( stack[top] .y == 3 ) {
            pop ( NULL, &i, &j, &k ) ;    y = 3 ; }
            else
          if(  stack[top].y  ==  2  )  {  pop(
           NULL, &i, &j, &k ) ; } */ break ;
case 3:
            push( 4, i, j, k ) ;
            к = 1 ;       y = 1 ;
            break ;
case 4 :
            push( 2, i, j, k ) ;
          i = 6-i-j ; к— ;       y = 1 ;
          break ; }
          while( y != 0 ) ; }
          void move( i, j ) {
          printf( "%d -> %d\n", i, j ) ;
          }
          int main() {
          int input = 3 ;
          printf ( "\nХаной с %d дисками:\n", input ) ;
          hanoy( 1, 3, input ) ;
          return 0 ; }
```

Возможны и другие варианты реализации функций pop() и push(), например, с передачей в качестве параметра указателя на переменную типа stack_t.

Если функцию hanoy() в этой программе реализовать не по графу переходов (рисунок 3.5), а по графу, полученному после его упрощения (рисунок 3.6), то она сокращается (листинг 3).

ЛИСТИНГ 3. Автоматная программа с минимизированным числом состояний, моделирующая рекурсию

```
void hanoy( int i, int j, int k ) {
  int y = 0 ;

  do
  switch( y ) {
       case 0:
       y = 1 ; break ;

       case 1:
         i f( k == 1 ) { move( i, j ) ;  y = 2 ; }
         else
         {
       push( 3, i, j, k ) ; j = 6 - i - j ; k-— ; }
break ;

       case 2 :
         if( stack_empty() )       y = 0 ;
         else
         if( stack[top].y == 4 )
         {
         pop( NULL, &i, &j, &k ) ;
         push( 2, i, j, k ) ;
         i = 6 - i - j ; k-— ;       y = 1 ; }
         else
         if( stack[top].y == 3 ) {
         pop( NULL, &i, &j, &k ) ;
         push( 4, i, j, k ) ;
         k = 1 ;                    У = 1 ;
         } else
    if( stack[top].y == 2 ) { pop( NULL, &i, &j, &k ) ; }
break ; } while ( y != 0 ) ;
```

Отметим, что при такой реализации, в отличие от программы, приведенной в листинге 3, не удается выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией pop().

3.3.2.2 Обход дерева действий

Дерево декомпозиции (Рисунок 3.2), вершинами которого являются рекурсивные вызовы, может быть преобразовано в дерево действий, выполняемых рассмотренной рекурсивной программой, путем исключения всех вершин кроме тех, в которых выполняется перекладывание (Рисунок 3.7).

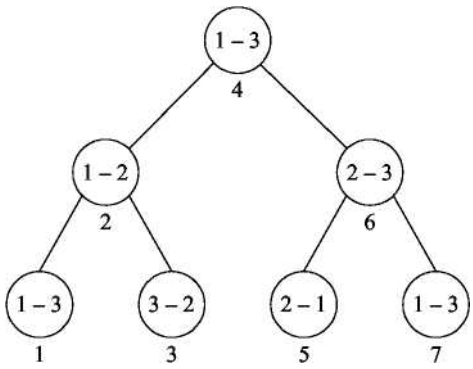


Рисунок 3.7 – Дерево действий при N = 3

Это дерево обладает следующим свойством: для вершины с пометкой (i,j) вершина левого поддерева имеет пометку (i,6-i-j), а вершина правого поддерева — (6-i-j j).

Построение итеративного алгоритма обхода может рассматриваться как способ раскрытия рекурсии. При этом необходимая последовательность перекладываний получается в результате обхода этого дерева слева направо. Такому обходу соответствуют числа, указанные рядом с каждой из вершин.

Построим итеративный алгоритм обхода этого дерева. Не храня его в памяти, будем осуществлять двоичный поиск каждой вершины, начиная с корневой, так как для нее известны номера стержней и способ определения их номеров для корневых вершин левого и правого поддеревьев. Например, для вершины с номером 5, на первом шаге алгоритма осуществляется переход от вершины 4 к ее правому поддереву — вершине 6, так как пять больше четырех. На втором шаге осуществляется переход от вершины 6 к ее левому поддереву — искомой вершине 4. Таким образом, несмотря на то, что обход дерева осуществляется слева направо, алгоритм работает сверху вниз.

Листинг 4 содержит программу, реализующую этот алгоритм.

ЛИСТИНГ 4. Итеративная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int
j, int k ) {
   int max_nodes = (1 << k) - 1 ; // Всего вершин.
   int root = 1 << (k-1) ;       // Номер корневой вершины.
   int node ;  // Номер искомой вершины в дереве.
```

```
// Определить номера стержней для каждой вершины в дереве.
for( node = 1 ; node <= max_nodes ; node++ )
{
  int a = i, b = j ;
  // Начальная позиция поиска соответствует корневой вершине.
  int current = root ;
  // Изменение номера вершины при переходе к следующему поддереву.
int ind = root / 2 ;
// Двоичный поиск нужной вершины.
while( node != current )
{
if( node < current ) {
// Искомая вершина в левом поддереве.
b = 6-a-b ;
current -= ind ; // Переход к левому поддереву. }
else {
// Искомая вершина в правом поддереве.
      a = 6-a-b ;
    current += ind ; // Переход к правому поддереву. }
    // Разница в номерах вершин при переходе к
    // следующему поддереву уменьшается в два раза,
    ind /= 2;// Номера стержней для рассматриваемой вершины определены,
    printf ( "Вершина %d. %d -> %d\n", node, a, b ) ;

  void main()

int input = 3 ;
printf ( "\nХаной с %d дисками:\n", input )
hanoy( 1,  3,  input )   ;
```

Построим схему этой программы (Рисунок 3.8).

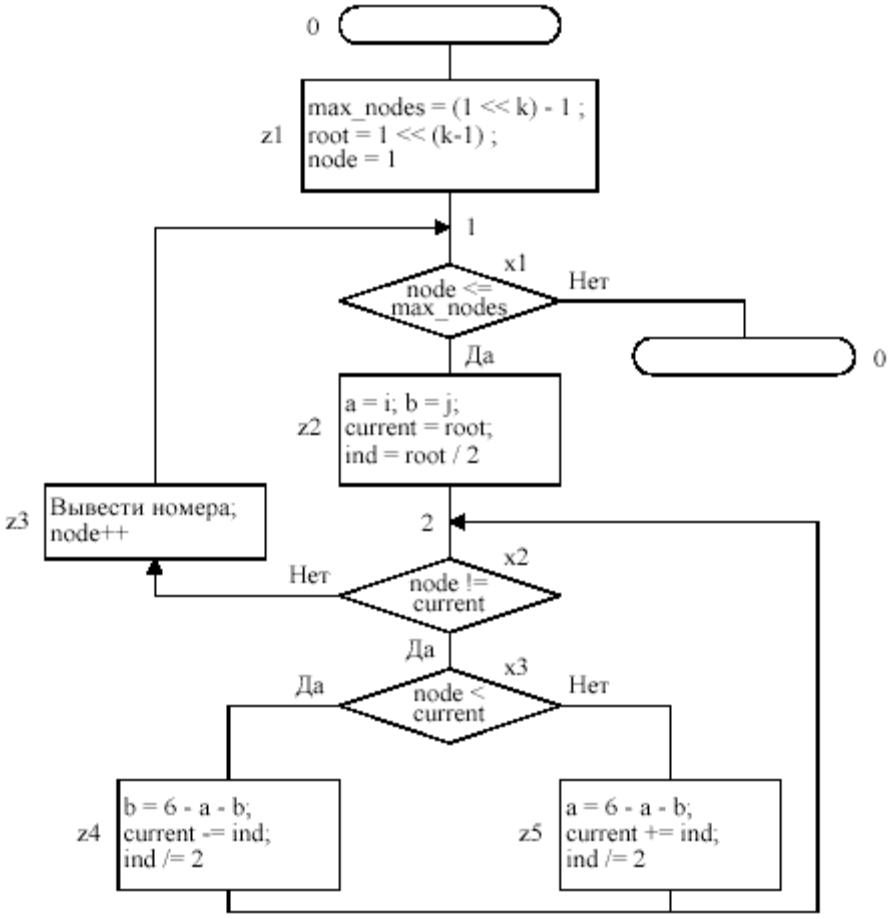


Рисунок 3.8—Схема программы обхода дерева действий

В соответствии с методикой, изложенной в работе [27], построим по этой схеме граф переходов автомата Мили (рисунок 3.9). При этом состояниям автомата Мили на схеме программы соответствуют точки, следующие за операторными вершинами. Нулевому состоянию соответствуют вершины схемы, обозначающие начало и конец программы.

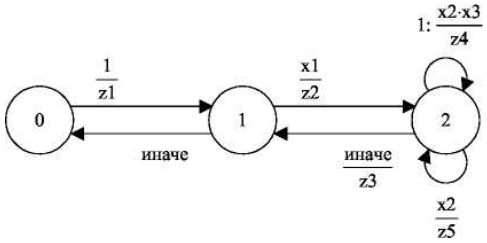


Рисунок 3.9. Граф переходов автомата, реализующего обход дерева действий

Программа, реализующая этот автомат, приведена на листинге 5.

ЛИСТИНГ 5. Автоматная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k ) {
  int y = 0 ;
  int max_nodes, root, node, a, b, current, ind ;

  do
    switch( y ) {
      case 0:

max_nodes = (1 « k) - 1 ; root = 1 « (k-1) ;

node = 1 ;    y = 1 ;
      break ;

      case 1:
        if( node <= max_nodes ) {
          a = i ; b = j ; current = root ;
          ind = root / 2 ;        y = 2 ;}
        else                      y = 0 ;
      break ;

      case 2 :
        if( node != current && node < current ) {
        b = 6-a-b ; current -= ind ; ind /= 2 ; }
        else
        if ( node != current ) {
        a = 6-a-b ; current += ind ; ind /= 2 ; }
        else {
          printf( "Вершина %2d. %d -> %d\n", node, a, b ) ;
          node++ ;                y = 1 ;}
    break ; }
while( y != 0 ) ; }

void main() {
  int input = 4 ;
  printf ( "\nХаной с %d дисками:\n", input ) ;
hanoy( 1, 3, input ) ;
```

Отметим, что при использовании изложенного метода номер перекладываемого диска явно не указывается. Эти номера могут быть определены с помощью подхода, изложенного в работе [9], который состоит в следующем: строится таблица, строки которой содержат двоичное представление номера шага. При этом номер разряда двоичного числа, в

котором размещена "младшая единица" (при условии, что счет разрядов начинается с единицы), является номером перекладываемого на данном шаге диска.

Аналитически номер перекладываемого диска может быть определен как единица плюс количество делений номера шага на два без остатка. При этом для нечетных номеров шагов количество делений на два без остатка равно нулю, и, поэтому, номер диска равен единице. Таким образом, на каждом нечетном шаге всегда перекладывается диск наименьшего диаметра.

Обобщая изложенное выше, построим дерево решения задачи (Рисунок 3.10), которое содержит исчерпывающую информацию для перекладывания дисков. При этом для каждой вершины снизу указан номер шага, а внутри — номера диска и стержней, участвующих в перекладывании.

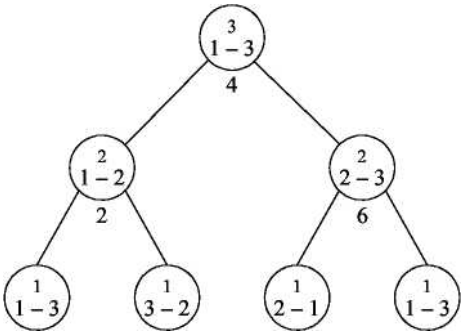


Рисунок 3.10 – Дерево решения задачи при N = 3

Из рассмотрения рис. 9 следует также, что номера перекладываемых дисков во всех вершинах одного уровня равны и совпадают с номером этого уровня.

3.3.2.3 Непосредственное перекладывание дисков

Если алгоритм решения рассматриваемой задачи задавать непосредственно в терминах объекта управления (дисков и стержней), как это предлагается П. Бьюнеманом и Л. Леви [10], то его удается описать в виде графа переходов автомата с двумя состояниями (рис. 10).

Этот автомат по очереди выполняет всего два действия: перекладывает наименьший диск циклически по часовой стрелке (zl) и перекладывает единственно возможный диск, кроме наименьшего (z2). После выполнения действия zl автомат проверяет условие завершения алгоритма (xl).

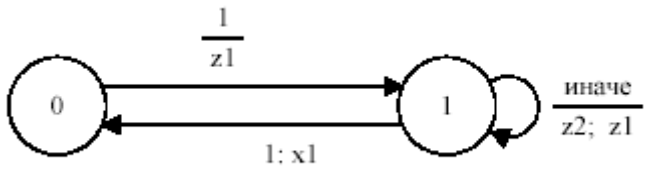


Рисунок 3.11 – Граф переходов при непосредственном перекладывании дисков

Текст программы, реализующей описываемый автомат, приведен в листинге 7. В этой программе функции zl, z2 и xl могут быть реализованы по-разному. Например, определение номеров перекладываемого диска и участвующих в перекладывании стержней может выполняться с помощью перебора (как это имеет место ниже), либо аналитически, например, как это предложено в работе [36].

Особенность рассматриваемой программы состоит в том, что направление перекладывания первого диска (функция zl) зависит от четности числа дисков и номера стержня, на который их требуется переложить. При перекладывании дисков с первого на третий стержень, первый диск следует перекладывать в порядке номеров стержней 1-2-3 при четном количестве дисков, и в порядке 1-3-2 при их нечетном количестве.

ЛИСТИНГ 7. Автоматная программа, реализующая непосредственное перекладывание дисков

```
        #inclu
de  <stdio.h>
#include
<stdlib.h>
#include
<string.h>
        int y = 0 ;
        int N ; // Количество дисков.
```

```
        int dest ; // На какой стержень перекладываем.
        int step = 0 ; // Номер текущего шага.
        int max_steps = 0 ; // Необходимое количество шагов.
        int first_on = 1 ; // На каком стержне находится первый диск.
        // Состояние объекта управления - "содержимое" стержней, char  s[4][100]
= { "", "1", "", "" } ;
        void main()
        int input = 14 ;
        printf ( "\nХаной с %d дисками:\n", input ) ;
        hanoy( 1, 3, input ) ;
        void hanoy( int from, int to, int disk_num ) {
        int i ;
        N = disk_num ;
        max_steps = (1 << N) - 1 ;
        dest = to ;
        // Заполнить первый стержень дисками, for ( i = 2 ; i <= N ; i
        sprintf( s[0],   "-%d",  i strcat ( s[1],  s[0]  )   ;
        do
        switch( y ) {
        case 0:
        z1()   ;         y = 1 ;
        break ;
        case 1:
        if( x1() )         y = 0 ; else
        { z2() ; z1() ; } break ; } while( y != 0 ) ;
        // Вывести результат, printf ( "\nРезультат:\n" ) ; for( i = 1 ; i <= 3 ; i++ )
        printf ( "%d: %s\n", i, s[i] ) ;
        // Проверить, завершено ли перекладывание, int x1() { return step >=
max_steps ; }
        // Переложить первый диск по часовой стрелке.
        void z1()
        {
        int from = first_on ;
        int i ;
        // Определить номер следующего стержня.
        if(    (dest == 2 && N%2 != 0) I | (dest == 3 && N%2 == 0) )
           first_on = (from + 1)%3 ; // Порядок стержней 1-2-3. else
        first_on = (from + 2)%3 ; // Порядок стержней 1-3-2. if( first_on == 0 )
first_on = 3 ; move( 1, from, first_on ) ;
        // Переложить единственный возможный диск, кроме наименьшего.
        void z2()
        {
        int  i, j ;
        int  disk_from, disk_to ;
        // Определить перекладываемый диск, for( i = 1 ; i <= 3 ; i
```

```
disk_from = disk_on(i) ; if( disk_from > 1 )
        // Определить на какой стержень перекладывать.
for( j = 1 ; j <= 3 ; j
disk_to = disk_on(j) ;
if( disk_to == 0 || disk_from < disk_to )
{move( disk_from, i, j ) ;
            return ;// Вернуть номер диска на указанном стержне.
int disk_on( int s_num ) { return atoi( s[s_num] ) ; }
// Переложить заданный диск.
int move( int disk, int from, int to )
{char  *str_pos = strchr( s[from], '-' ) ;
if( str_pos == NULL )
  s[from][0] = 0 ; else
    strcpy( s[from], str_pos+1 ) ;

  if ( s[to] [0] == 0 )
  sprintf( s[to], "%d", disk ) ; else {
    strcpy( s[0], s[to] ) ;
sprintf( s[to], "%d-%s", disk, s[0] ) ;
step++ ;
  printf( "Шаг %d. Диск %d: %d -> %d\n", step, disk, from, to ) ;
    return 0 ; }
```

В целом итеративный алгоритм не сложнее своего рекурсивного аналога. Вместе с тем, любая оптимизация ведет к усложнению анализа условий, обеспечивающих более быстрый выбор нужных стержней для выполнения переносов дисков. Следовательно, сложнее становится и управление программой за счет появления ветвей. Это позволяет интерпретировать алгоритм с позиции теории автоматов, что вряд ли является целесообразным для задачи, базирующейся на описании решения в виде формул.

Каждый из подходов, на мой взгляд, может (и должен) использоваться в соответствующих областях. Теория автоматов нашла эффективное применение при решении задач обработки постоянно меняющихся параметров, взаимосвязанных отношениями со слабо выраженными функциональными зависимостями. В этих случаях на каждом шаге необходимо осуществлять

неочевидный выбор следующего направления среди множества разветвляющихся путей.

В задаче о ханойских башнях, несмотря на то, что количество вычислительных состояний объекта управления (три стержня и n дисков) растет как экспонента от n, автомат, обеспечивающий перекладку дисков, имеет всего лишь два- три управляющих состояния.

Выводы

Предлагаемый подход позволяет:

- использовать теорию конечных детерминированных автоматов при алгоритмизации и программировании задач школьного курса информатики;
- первоначально описывать желаемое поведение управляющего "устройства", а не его структуру, которая является вторичной и поэтому труднее читаемой и понимаемой;
- ввести в алгоритмизацию и программирование в качестве основного понятие "состояние", начиная алгоритмизацию с определения числа состояний;
- ввести понятие "автоматное программирование" и "автоматное проектирование программ";
- применять основные структурные модели теории автоматов;

Основной особенностью программной реализации конечных автоматов является обязательное наличие цикла, объемлющего код собственно реализации автомата. В литературе этот принципиальный момент опускается, делая невозможным понимание практического использования автоматных программ.

Автоматный подход использован при реализации вычислительных алгоритмов.

ЗАКЛЮЧЕНИЕ

Основные научные результаты диссертации

1. Магистерская диссертация представляет собой исследование, направленное на восполнение пробелов школьной программы по информатике и на ликвидацию несоответствия требований к школьникам действующей программы и уровнем подготовки учащихся.
2. В работе рассматриваются вопросы алгоритмизации и программирования задач информатики на основе теории автоматов. Исследована применимость автоматного подхода в программировании к решению задач информатики школьного курса. Описаны решения рекуррентных соотношений.
3. В работе показано, что автоматы являются не просто одной из математических моделей дискретной математики, а могут применяться при реализации любых программ, обладающих сложным поведением.
4. Данная работа связывает рекуррентные соотношения и конечные автоматы, а также восполняет имеющиеся пробелы по указанным направлениям в школьном курсе информатики.
5. В результате выявленного несоответствия школьных программ по информатике, проведенного теоретического исследования теоретически разработана программа обучения школьников информатике на факультативном курсе по выбору, эффективность которой подтверждается экспериментальной проверкой.
6. Повысилась заинтересованность учащихся, входящих в экспериментальную группу, в учебе, в результате чего заметно улучшилась успеваемость не только по предмету информатика, но и по математике, вследствие повышения авторитета учителя среди учащихся.
7. Предложенный факультативный курс «по выбору» обсуждался как до начала занятий, так и после окончания учебного года на педагогических

советах гимназии, в ходе которого получил положительную оценку с предложением продолжить его применение на практике в виде учебно-методических рекомендаций.

В ходе выполнения работы проводилось три этапа исследований:

1. Статистическое изучение ситуации и анализ данных по вопросу преподавания дисциплины «Информатика» в средней школе.

2. Теоретическая разработка новой программы для обучения школьников курсу информатики.

3. Практический эксперимент, основанный на апробации результатов магистерского исследования в виде методического руководства для преподавания курса по выбору дисциплины «Информатика» в средних школах.

Рекомендации по практическому использованию

1. Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами использования методов, предложенных в диссертации, на практике.

2. Материал, входящий в курс по выбору для школьников апробировался на внеклассных занятиях учащихся Экономико-правовой гимназии. В дальнейшем планируется более широкое исследование, направленное на устранение пробелов в программе обучения школьников информатике с целью создания авторской программы всего курса информатики для школы.

3. Результаты, представленные в работе, позволяют повысить эффективность исследований в области применимости рекуррентных задач для школьников. Кроме того, имеется возможность использования разработанного программного средства при обучении и решении практических задач программирования.

4. Практическое значение работы состоит в том, что все полученные результаты могут быть использованы, а некоторые уже используются на практике. Предложенные методы позволяют повысить наглядность программ, упростить их визуализацию, а также упростить внесение изменений в них. При этом, за счет преобразования условной логики в автоматную, упрощается структура программ.

5. Предлагаемый подход позволяет использовать теорию конечных детерминированных автоматов при алгоритмизации и программировании решения задач школьного курса информатики, а также применять основные структурные модели теории автоматов в решении задач практической направленности.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Стивенс, Р. Delphi. Готовые алгоритмы.
2. Kleene, S.C. Representation of events in nerve sets and finite automata //Automata Studies. Princeton University Press. 1956. — P.3 - 41.
3. Shalyto, A.A. Software Automation Design: Algoritmization and Programming of Problems of Logical Control // Journal of Computer and Systems Sciences International. 2000. N6. — P.899-916.
4. Арсак, Ж. Программирование игр и головоломок: Пер. с франц. – М.: Наука. гл. ред. физ.-мат. лит., 1990. – 224 с.
5. Баранов, С.И. Синтез микропрограммных автоматов (граф-схемы и автоматы). Л.: Энергия, 1979.
6. Боглаев, Ю.П. Вычислительная математика и программирование. М.: Высшая школа, 1990. — 544 с.
7. Богомолов, А.М., Твердохлебов В.А. Целенаправленное поведение автоматов. Киев: Наукова думка, 1975. — 124 с.
8. Воробьев, Н.Н. Числа Фибоначчи. М.: Наука, 1983. – 164 с.
9. Гарднер, М. Математические головоломки и развлечения. М.: Мир, 1999. – 447 с.
10. Гарднер, М. Математические новеллы. М.: Мир, 1974. – 458 с.
11. Глушков, В.М. О применении абстрактной теории автоматов для минимизации микропрограмм // Изв. АН СССР. Техническая кибернетика. 1964. № 1. — С. 3 - 8.
12. Горбатов, В.А. Фундаментальные основы дискретной математики. Информационная математика. М.: Наука, 1999. – 544 с.
13. Грин, Д., Кнут, Д. Математические методы анализа алгоритмов. М.: Мир, 1987. – 120 с.
14. Грис, Д. Наука программирования. М.: Мир, 1984. – 416 с.
15. Грэхем, Р., Кнут, Д., Паташник, О. Конкретная математика. Основание информатики. М.: Мир, 1998. – 703 с.
16. Дейкстра Э. Дисциплина программирования. М.: Мир, 1978. – 275 с.
17. Казаков, М. А., Корнеев, Г. А., Шалыто, А. А. Метод построения логики работы визуализатора алгоритмов на основе конечных автоматов // Телекоммуникации и информатизация образования. 2003. №6. — С. 27-58.
18. Карпов, Ю.Г. Теория алгоритмов и автоматов. СПб.: Геликон Плюс, 2000.
19. Котов, В.М., Мощенский, В.А. Рекуррентные соотношения и основные методы их решения: пособие для студентов специальности «Информатика» - Мн.: БГУ, 2007. – 42 с.
20. Лавров С.С. Программирование: Математические основы, средства, теория: Учебное пособие / С.С. Лавров. - СПб.: БХВ-Петербург, 2001. – 320 с.
21. Оллонгрен, А. Определение языков программирования интерпретирующими автоматами. М.: Мир, 1977. – 288 с.

22. Паўлоўскі А.І., Панамарэнка В.К. Рэкуррэнтныя суадносіны ў курсах інфарматыкі педагагічнага універсітэта. // 2004 Весці БДПУ
23. Поспелов, Д.А. Логические методы анализа и синтеза схем. М.: Энергия. 1974. – 368 с.
24. Раздел «Проекты сайта СПбГИТМО» (ТУ) http://is.ifmo.ru/?i0=projects.
25. Седжвик Роберт Фундаментальные алгоритмы на С++. Анализ. Структура данных. Сортировка. Поиск / Пер с англ. – К.: Издательство" "ДиаСофт"",2001. – 688 с.
26. Стахов, А.П., Ткаченко И.С. Гиперболическая тригонометрия Фибоначчи // Докл. НАН Украины. – 1993, вып. 7. — С. 9-14. Стахов А.П., Ткаченко И.С. Гиперболическая тригонометрия Фибоначчи // Докл. НАН Украины. – 1993, вып. 7. — С. 9-14.
27. Трахтенброт, Б.А. Алгоритмы и вычислительные автоматы. М.: Советское радио, 1974. — С. 200.
28. Туккель, Н. И., Шалыто, А. А., Шамгунов, Н. Н. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2002. № 5.
29. Хопкрофт, Д., Мотвани, Р., Ульман, Д. Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
30. Чирков, М.К. Основы общей теории конечных автоматов. Л.: Изд-во ЛГУ. 1975. — 280 с.
31. Шалыто, А., Туккель, Н., Шамгунов, Н. Ханойские башни и автоматы //Программист. 2002. № 8.
32. Шалыто, А.А. Новая инициатива в программировании – "Движение за открытую проектную документацию" //Мир ПК – Диск. 2003. № 8.
33. Шалыто, А.А., Туккель, Н.И. Реализация вычислительных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2001. №6. – С. 35-53.
34. Шалыто, А.А.Технология автоматного программирования. Мир ПК. 2003. №10, с.74-78; Современные технологии. СПбГУ ИТМО. 2003, с.18-26. Статья размещена на сайте http://is.ifmo.ru.
35. Романовский, И.В., Столяр, С.Е. Стек и его использование. http://ips.ifmo.ru.
36. Бобак, И. Алгоритмы: "возврат назад" и "разделяй и влавствуй" // Программист. 2002. №3.
37. Быстрицкий, В.Д. Ханойские башни. http://alglib.chat.ru/paper/hanoy.html
38. Кнут, Д. Искусство программирования для ЭВМ, т.2, — с.482.
39. Маркушевич, А.И. Краткий курс теории аналитических функций. Пятое издание. Мн.: Издательство «Мир», 2006. – 423 с.