ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Книга, введение к которой вы сейчас читаете, третья в моем авторском курсе. Две первые – «Современное программирование с нуля» и «Искусство алгоритмизации» – посвящены вхождению в творчество и науку программирования. Книга «Поиск решения» должна познакомить вас с одним из возможных ответов на вопрос «Что делать с нетривиальной задачей, если её решение нельзя прочитать в хорошей книжке?».

Сразу обращаю ваше внимание на одно важное обстоятельство. Программирование можно понимать как технологию сборки решения из более мелких задач. Такой подход создал индустрию программирования. И программирование можно понимать как искусство поиска решения логически сложной задачи, сведение которой к более простым не решает проблемы, потому как она либо просто не разбивается на хорошие задачи, либо эти более простые все равно представляют собой систему, связанную сложной логикой. Технологическому подходу будет посвящена четвертая книга моего курса. А сейчас мы постараемся посмотреть на программирование как на интеллектуальное искусство.

Главная идея книги – показать процесс мыслительной деятельности таким, какой он есть на самом деле, с ошибками, тупиковыми вариантами, рождением красивой идеи. И что, пожалуй, главное – показать возможность такой организации мышления, при которой поиск решения становится системной и плановой работой. Конечно, по прочтении книги у вас не будет рецепта построения решения. Единственный метод борьбы с творческими проблемами – развитый мыслительный аппарат, поэтому все главы книги – это описание процесса поиска решения реальных задач. И поиск этот не бессистемный, путь к решению в каждой задаче лежит не только через интуитивные прорывы и даже не столько через них, сколько через анализ накопленной информации.

Основной метод, действие которого проявляется на каждом шагу, – это метод борьбы с неопределенностями. Решение каждой за-

дачи представляется как последовательность вопросов «Что нам еще не ясно?» и «Как с этим бороться?». Возможно, некоторые из использованных задач искушенному «решателю» покажутся простыми, но цель максимально осложнить чтение не ставилась, поэтому задачи разноуровневые и поэтому книга может оказаться полезной для людей с самой различной степенью подготовки.

Базовый язык книги – Компонентный Паскаль, современная версия хорошо зарекомендовавшего себя языка, но некоторые решения записаны на псевдокоде. В принципе, базовый язык мог бы быть и другим, но все же Паскаль представляется наиболее удачным средством организации простого и в то же время достаточно строгого описания. Мой программистский и главным образом преподавательский, учительский опыт говорит, что императивный стиль письма более соответствует природе программисткого мышления, чем функциональный или другие. Впрочем, даже если читатель не особенно приветствует императивные языки или язык Паскаль, то и такой читатель проблем с прочтением книги не испытает. Я уверен, что языковые особенности для большинства задач нашей предметной области не должны играть существенной роли.

P.S.

Обратите внимание на правила, выведенные в процессе решения, их немного, и они не составляют какой-либо завершенной теории. Создать систему правил можно, но это опасное занятие. Завершенная теория, скорее всего, привела бы не столько к повышению эффективности мыслительного процесса, сколько к ограничению ваших творческих возможностей. Поэтому правила, сформулированные во всех последующих главах, не носят характера предписаний, это скорее небольшие советы, корректирующие направление мышления.

ГЛАВА 1. Как решается сложная задача

Существует один общий подход к поиску решения сложной задачи, независимо от того, из какой она области: математики, физики или программирования. Выражается он в трех простых предложениях:

1. Определим тип задачи.
2. Вспомним, какими методами нам или кому-нибудь другому доводилось решать задачи такого типа.
3. Попробуем применить эти методы к поставленной задаче.

Подход кажется логичным и разумным. Ведь большинство задач, с которыми мы сталкиваемся, кем-то уже решены, кто-то уже знает, как ответить на заинтересовавший вопрос, ответ уже есть в совокупной базе знаний человечества. Но накопленное общечеловеческое знание – это достаточно сложная штука. Оно не так доступно, как хотелось бы в силу своей огромности. Существуют и другие причины, по которым конкретный человек, сталкиваясь с реальной задачей, может не найти готового решения. Поэтому, несмотря на то что человечество в целом знает и умеет довольно много, отдельный человек часто встречается с ситуацией неопределенной и, следовательно, творческой. А в такой ситуации наш алгоритм из трех предложений уже не работает.

Кроме того, если взять действительно интересную задачу, то окажется, что определить, к какому типу она относится, довольно сложно. И часто задача будет относиться не к одному типу, а к нескольким. Например, это может быть задача комбинаторного характера с применением графов, или это может быть задача моделирования физических процессов с использованием графики и методов численной математики и т. д.

Если бы была возможна исчерпывающая классификация задач, с конечным количеством классов, четко отделимых друг от друга, то это бы

означало ограниченность программисткой науки. Программирование, как наука закончилось бы в момент написания последнего алгоритма на последний неразрешенный класс задач.

Вроде бы нет ничего страшного, просто в таких задачах мы имеем дело со сложными типами, и все равно можно действовать по той же схеме, то есть последовательно выполнять действия 1, 2, 3.

К сожалению, не получается. Во-первых, сложных типов можно сконструировать огромное количество, а чем типизация обширнее, тем сложнее выполнять пункт первый. Во-вторых, чем больше типов, тем сложнее в них ориентироваться, а в-третьих, тем сложнее отличить, где заканчивается один и начинается другой. Например, задача графического моделирования физического процесса, – это задача численной математики, физики или это графическая задача? В общем, при попытке составить какую-то классификацию мы столкнемся с таким количеством проблем, что невольно придет мысль поискать другой подход.

Попробуем подойти к сложным задачам с другой стороны. Начнем с небольшого, но очень важного замечания. Что бы мы не изобретали, все упирается в рождение идеи. А идея всегда рождается интуитивно, причем независимо от степени её гениальности и значимости. Происходит это так: вы некоторое время сосредоточенно размышляете на заданную тему, и вдруг приходит решение, причем не совсем понятно, откуда. Это называется интуицией. Существуют различные теории, объясняющие интуицию, откуда она берется, что она такое сама по себе, но для нас эти теории ровным счетом ничего не значат, так как не дают метода мышления.

Все серьезные идеи рождаются интуитивно, что огорчает, так как совершенно не ясно, как интуицией управлять, но, с другой стороны, совершенно не подготовленный человек вряд ли сможет сформулировать красивую идею, что вселяет надежду. Ведь если для рождения эффективной идеи нужна подготовка, то, значит, интуиция где-то в своей основе содержит систему, какой-то метод, а методу можно научиться.

Что такой метод может из себя представлять? Можно ли описать его точно? Конечно, нельзя ожидать, что общий метод решения творческих задач будет иметь алгоритмическую ясность. Невозможно себе представить, что такой метод будет описанием последовательности действий. Хотя, конечно, многие люди, когда речь идет о методе, представляют некую последовательность инструкций, выполнив которые, можно получить верный результат. Но это глубокая, принципиальная ошибка.

Поэтому мы сразу и навсегда откажемся от идеи разработать такой всеобъемлющий алгоритм. Кроме того, мы решительно откажемся от попыток до конца понять тайну творчества. Это слишком невероятно, чтобы интуиция, творческий инсайт подлежал исчерпывающему логическому анализу. Это то, чего мы не будем делать. А попробуем разработать подход, помогающий удержать направление исследования и превращающий хаос творчества в осмысленный процесс.

Итак, мы ищем не методы решения задач, а методы организации мыслительной деятельности. Такова наша цель. И, чтобы её достичь, не будем строить теорию, а получим умения из практики. Решая задачи и отмечая то, что помогло получить решение, как-то обобщать свои наблюдения и, если получится, выводить общие правила.

Общие правила не должны иметь форму запретов, но надо четко понимать, что неограниченное творчество – это, по сути, хаос, из которого не появляется ничего продуктивного. Творчество творчеством, а любому исследователю необходимо получить приемлемое решение за ограниченное время. Поэтому дисциплина мышления должна ставить перед собой задачу, не создавая жестких запретов, тем не менее упорядочивать мыслительный процесс, нацеливая его на результат.

В этой книге большая часть глав посвящена конкретным проблемам, но есть несколько текстов общего значения, как, например, данная глава. А ниже мы рассмотрим подход, который, возможно, отражает самую общую идею поиска программисткого решения и учитывает именно программисткую специфику такой работы.

Пошаговое уточнение неопределенностей

Рассмотрим пример несложной задачи. Пусть требуется разработать алгоритм расчета всех простых чисел, не превосходящих заданное число *N*. Если для решения не требуется особой эффективности, например верхняя граница не слишком велика, то можно воспользоваться самой очевидной идеей: алгоритм решения – это цикл, в теле которого для каждого очередного числа выясняется, простое оно или нет, и если простое, то число печатается как результат.

Для того чтобы принять решение относительно простоты очередного числа, необходимо проверить все числа, могущие быть его делителями, и если хотя бы один делитель есть, то проверяемое очередное не простое, иначе все-таки простое.

Идея вполне понятна, но все же вчитаемся в текст более внимательно, все ли здесь действительно хорошо. Один неясный пункт есть, это фраза, требующая проверить все числа, могущие быть делителями. Я из своего личного преподавательского опыта знаю, что ученики, недостаточно искушенные в задачах математической природы, о такие вещи спотыкаются довольно часто. «Как проверить?» – это и есть уточняющий вопрос, ответив на который, мы сможем записать решение в завершенной форме.

Заметим, что любое исследование начинается с искусства вопроса. Правильно заданный вопрос – это, образно говоря, половина ответа.

Рассмотрим более сложный пример. Есть группа людей со вполне определенными симпатиями и антипатиями друг к другу. Необходимо расставить их по рабочим позициям так, чтобы коллектив оказался психологически наиболее устойчивым. Возможно, психология дает какие-то методы решения подобных задач, но мы психологической наукой не владеем. А из соображений здравого смысла ясно, что необходимо организовать перебор всех возможных вариантов и выбрать из них вариант с наибольшим значением критерия устойчивости.

Это своего рода макроидея, или решение в первом приближении. Здесь сразу видны следующие неопределенности. Во-первых, что значит перебирать варианты? Ясно, что с людьми этого делать не получится, ни в одном языке программирования нет такого термина – «люди». Эту неопределенность мы устраняем легко, пронумеровав группу. Тогда перебор вариантов расстановки людей сводится к перебору всех возможных перестановок их номеров. Кстати, появившийся термин «перестановка» наводит на мысль поискать какой-нибудь уже известный алгоритм построения всех перестановок. Но здесь возникает новая неопределенность. Совершенно не факт, что наше понимание термина перестановка совпадает с математическим пониманием. В этом надо убедиться.

Следующая неопределенность сложнее. Это критерий устойчивости. Ясно, что для каждой полученной перестановки необходимо что-то считать, и это что-то должно характеризовать устойчивость построенного коллектива. Мы должны определиться с математической формой этого критерия.

Как известно, математика – это язык для точной записи знания, поэтому выше не зря появилось упоминание о математической форме записи кри-

терия. Любая работа по устранению неопределенностей в конечном итоге должна привести к математически строгим формулировкам.

Таким образом, данная итерация мыслительного процесса должна нам дать формулу расчета критерия и алгоритм, генерирующий перестановки. В общем-то можно начинать писать код. Но на этапе разработки алгоритма есть еще одна фундаментальная неопределенность – это требование к скорости работы будущей программы или требование к скорости работы алгоритма. Конечно, скорость программы и скорость алгоритма – несколько разные вещи. Хороший алгоритм можно испортить плохой программой, а вот плохой алгоритм хорошей программой не исправишь, поэтому вопросы эффективности – это прежде всего вопросы алгоритмизации.

В отношении поставленной задачи вопрос скорости очень актуален, мы пришли к необходимости использовать математическую конструкцию перестановок, а, как известно, для N элементов можно построить $N!$ перестановок. Это очень большая величина, и если коллектив содержит более 10 человек, чисто переборный вариант решения нас вряд ли устроит, и есть смысл подумать о более быстром алгоритме, использующем какие-то особенности формулировки задачи, или обдумать возможность отказа от идеального решения. То есть следующая итерация борьбы с неопределенностью должна дать ответ на вопрос о желаемой эффективности алгоритма.

Таким образом, процесс поиска решения заключается в записи решения, так как оно уже понято, выявлении неопределенностей и формулировке вопросов, ответы на которые устранят неопределенности. И весь процесс можно разбить на ряд итераций, для каждой из которых выполняется очередная запись решения, выявляются очередные неопределенности, формулируются очередные вопросы.

Заметим, что метод ничего не говорит о том, как будет организован поиск ответов на поставленные вопросы, это уже несколько иная проблема. Метод пошагового уточнения, – это общая форма организации мыслительного процесса, еще не гарантирующая успешности в частном деле поиска ответов на содержательные вопросы.

Важно заметить, что разбиение мыслительного процесса на фиксированные итерации довольно условно. Конкретная траектория поиска решения зависит от имеющейся системы знаний решателя, от степени развитости интуиции, от банальной удачи. Метод предлагает в общем-то простую вещь. Разбить процесс решения на этапы, в конце

каждого решатель должен провести анализ и четко понять, что уже ясно и какие проблемы перед ним стоят.

Таким образом, описываемый процесс уточнения самым прямым образом требует навыков рефлексии (анализа собственного состояния). Вы должны научиться постоянному контролю своего интеллектуального состояния. Внутри вашего мыслительного процесса должен постоянно работать маячок, фиксирующий обнаружение проблемы.

Набор проблем, – кстати, величина не постоянная. Появляющиеся идеи приближают окончательное решение не в арифметическом смысле, с каждым шагом, все теплее и теплее. Полученное промежуточное решение может поставить новую проблему. Например, в разобранном примере вывод о необходимости алгоритма построения перестановок ставит перед решателем сложнейшую проблему комбинаторного взрыва.

Усложнение решения – процесс неизбежный. Необходимо понимать, что решатель, приступая к задаче, скорее всего, не до конца понимает её характер и природу. Процесс решения приводит к более точному пониманию, а следовательно, к проявлению проблем, которые существовали, но не были видны решателю.

Еще одно важное замечание о порядке выбора подзадач. Предположим, что после очередной итерации мы имеем ряд проблем: *A*, *B*, *C*... и т. д. Какую из них выбрать для следующей итерации? Простая логика говорит, что ключевую, то есть такую, которая укажет магистральный путь для решения всей задачи. Наверное, это справедливо. Вопрос только в том, что, не решив задачу, нельзя сказать, какая проблема была главной, хотя опыт говорит, что интуитивно это как раз почти всегда ясно – ключевая проблема выглядит наиболее сложно.

Однако в этом вопросе есть важный психологический нюанс. Для успешной работы в задачу нужно погрузиться. Погружение, особенно для не вполне опытного исследователя, подразумевает хотя бы небольшой, локальный, но успех. Теперь представьте себе ситуацию. Вы приступаете к задаче и четко видите, что нужен ответ на сложный вопрос *A* и относительно простой вопрос *B*. Вы не сомневаетесь, что ответ на вопрос *A* будет ключом к решению, ответ на вопрос *B* имеет чисто технический характер. Простая логика говорит, что необходимо заняться вопросом *A*. Но, быть может, лучше решить технический вопрос *B*?! Вполне возможно, что его решение не очень важно, может быть, даже позже, получив ответ на вопрос *A*, вы поймете, что ответ на

вопрос *B* был не вполне удачным, но все-таки работа над вопросом *B* может дать вам именно то необходимое погружение в задачу, без которого не заработает ваш мыслительный аппарат. Это как разминка перед забегом. Она не приносит победы, но готовит мускулатуру к необходимому рывку.

Сказанное означает, что процесс поиска решения не сводится к чисто интеллектуальному действу, набору каких-то логических умозаключений. Решение ищет человек с определенными психологическими возможностями и особенностями, и это должно быть учтено.

Наконец, последнее, но не в последнюю очередь. Говоря о программировании, традиционно принято рассказывать о структурном программировании и нисходящем проектировании. Во второй книге моего курса, называемой «Искусство алгоритмизации», этим понятиям посвящена целая глава. Здесь же мы говорим о несколько другом. Обсудим различие. Структурное программирование – это парадигма, описывающая технически грамотное построение программы, мы же сейчас обсуждаем не программирование, а процесс поиска решения. Термин «Нисходящее проектирование» несколько ближе. Эта парадигма предписывает разбивать задачу на подзадачи, из решений которых потом будет конструироваться решение исходной задачи. Это ближе, но все же не вполне о нашей проблеме.

Принцип борьбы с неопределенностями предписывает разбивать на части не задачу, а процесс поиска решения, то есть речь, по большому счету, идет не о статической структуре задачи, а о динамичном мыслительном процессе.

Все последующие главы посвящены построению динамичного мыслительного процесса. Описанный выше подход, – это основа, фундамент. Далее мы постараемся выделить некий набор правил, помогающих определить интеллектуальную ситуацию и то, какими действиями её изменять в сторону большей определенности. Но это дальше, а пока глава еще не завершена, обсудим еще некоторые важные вещи.

Формализация задачи

Процесс формализации также можно рассматривать как этап уточнения задачи, но процедура формализации имеет особое значение и особый смысл подготовительного этапа к поиску решения. Пробле-

ма любой реальной задачи – в неопределенности терминологии как минимум, а иногда и в упущении некоторых условий. Строгое определение всех объектов, участвующих в условии задачи, обязательно уберет неточность терминов и, возможно, поможет выявить пропущенные условия.

Что мы имеем в виду, когда говорим о возможных неточностях? Проиллюстрируем проблему примерами. Например, в следующей формулировке «Дано множество объектов, найти такую их комбинацию, что....» скрывается очень грубая неопределенность, делающая условие полностью неясным. Комбинация – это термин, не имеющий однозначного смысла. Это могут быть перестановки, это могут быть размещения, сочетания. Процесс перебора комбинаций самым серьезным образом зависит от типа комбинации, поэтому в условии задачи необходимо прояснить, о каком типе комбинации идет речь, и переписать условие соответствующим образом, например так: «Дано множество объектов, найти такое их размещение, что....». Здесь формулировка уже более строгая, и при этом мы получили значительно больше, нежели просто понимание, мы получили метод решения, сводимый к алгоритму построения размещений.

Таким образом, формализация выполняет функцию уточнения условия, устраняя избыточность терминологии и создавая предпосылки поиска решения.

Далее, решение задачи – это запись некоторым набором терминов. Условие задачи – это также запись некоторым набором терминов. Для того чтобы работать с заданным условием, необходимо терминологическое соответствие между текстом условия и будущим решением. Известно, что любой язык программирования отличается очень небогатым набором выразительных средств в сравнении с языком естественным, что порождает ситуации очень серьезного смыслового разрыва, в результате которого задача, записанная на естественном языке, фактически не решаема, пока мы не запишем её в терминах языка программирования.

В такую ситуацию постоянно попадают участники программистских олимпиад. Приведем несколько простых примеров.

> **Пример 1.** В классе 30 учеников, между которыми есть как симпатии, так и антипатии, учитель поставил перед собой задачу рассадить учащихся так, чтобы создать в классе максимально комфортную психологическую обстановку.

У нас нет понятия «максимально комфортная психологическая обстановка», поэтому сразу договоримся, что речь идет о допустимой или недопустимой рассадке учеников в классе. Далее заметим, что ученические места в классе представляют собой красивый прямоугольник. Это означает, что неопределенный термин «рассадка учеников в классе» можно заменить более формальным «заполнение числовыми значениями прямоугольной матрицы». Термин «множество учеников» можно заменить термином «массив чисел», в котором числа могут играть роль номеров учащихся.

> **Пример 2.** Морской берег имеет сложную конфигурацию, представляющую собой ломаную линию...

Не приводим формулировку полностью, она довольно длинная. Сказанного достаточно для иллюстрации проблемы. Здесь прозвучали два естественных термина: «морской берег», «сложная конфигурация». Если их заменить языковыми терминами, то может получиться, например, такая формулировка:

> «*Дан массив записей, содержащих два целочисленных поля, представляющих собой координаты вершин ломаной линии...*»

В вышеприведенных примерах вопросы формализации решились достаточно легко. Мы просто подобрали термин алгоритмического языка, один в один подходящий для замены термина естественного. Но такой подход работает не всегда.

> **Пример 3.** Дано число вида A^N...

Не так важно, что требуется в этой задаче. Угрозы от прямой замены терминов видны невооруженным глазом. Если N достаточно велико, то целочисленных типов может оказаться недостаточно, а использование действительных типов повлечет за собой неточность в представлении числа, что может оказаться фатальным для решения. В этой задаче, прежде чем решать вопрос о формализации представления данных, необходимо проанализировать, какие должны быть выполнены операции.

Если речь идет о выполнении арифметических целочисленных операций, то может быть разумно представить число в виде массива или связного списка. Если речь идет об исследовании вопросов делимости, то, может быть, можно вообще обойтись без представления конечного числа, а лучше подумать, как связать свойства основания «A» со свойствами операции возведения в степень? В этом случае привычный целый тип окажется достаточным, а задача окажется более математической, нежели программисткой.

Формализация исходных объектов, описанных в условии задачи, самым прямым образом связана с потребностями операций, которые предполагается выполнять над объектами.

Пример 4. Лиса на круглом закрытом поле пытается поймать зайца. Скорость зайца равна скорости лисы. Определить оптимальную стратегию для обоих.

Сходу ясно только то, что и лиса должна постоянно находиться в движении, и заяц может остановиться только в том случае, когда лиса не движется. Если копнуть глубже, то мы сразу упираемся в термин «стратегия». Что это такое и как это описать?

Наверное, стратегия, – это описание действий зайца и лисы, которых, кстати, легко формализовать парой координат на поле. Тогда стратегию поведения зайца можно описать кривой линией, при условии что траектория лисы известна. Но траектория лисы, по соображениям здравого смысла, зависит от траектории зайца. Выходит так, что траекторию зайца нельзя определить без траектории лисы, и наоборот, и получаем тупиковую ситуацию. Здесь есть два выхода из положения.

Выход первый. Описание поведения объектов в некоторой небольшой пространственной или временной окрестности.

Таким способом часто решают свои проблемы физики и математики. Например, зная значение производной от функции в точке, можно интегрированием восстановить внешний вид функции. Зная локальные характеристики газа в небольшом объеме, можно представить его поведение в большом сосуде. Если мы знаем скорость тела на малом участке пути и знаем, что никакие силы на него не действуют, то мы знаем, как оно будет двигаться дальше. Это намек на то, что, зная общее правило для принятия решения лисой и зайцем в конкретный момент времени, мы можем выстроить и траекторию. Вполне возможно, что у неё окажется какой-то очень простой вид. В этом случае решение задачи сводится к поиску такого точечного правила.

Выход второй. Все же возможно решение макрозадачи.

В условии было сказано, что для обоих необходимо построить оптимальную стратегию. Мы уже решили, что термин «стратегия» удачно заменяется термином «траектория». Далее, можно вполне обоснованно выдвинуть гипотезу, что при заданной форме поля существует только одна оптимальная стратегия, а следовательно, и только одна траектория. Справедливость этой гипотезы будет означать, что исходную задачу можно переформулировать так:

> *«Построить для преследуемого и преследователя такие траектории, что отступление от оптимума со стороны преследователя будет означать увеличение расстояния, а отступление со стороны преследуемого будет означать сокращение расстояния».*

В некотором смысле можно сказать, что формализация задачи – это ответ на вопрос, что же на самом деле требуется получить в решении и как это должно выглядеть.

Еще можно сказать, что формализация задачи – это запись условия и процесса решения в виде, не допускающем многозначного толкования, что, в свою очередь, дает еще одно определение формализации, как запись, на формальном языке. В таком коротком параграфе дано сразу несколько пониманий термина «формализация», но, по большому счету, таким образом всего лишь показаны разные стороны одной вещи.

Решение как построение цикла Дейкстры

А сейчас позвольте сформулировать гипотезу относительно формы очень многих программистских решений. Сразу хочу заметить, что это не более чем моя личная гипотеза, поэтому не судите слишком строго, а звучит она так:

> *«Решение алгоритмически конечной задачи сводимо к построению цикла Дейкстры».*

Здесь в соответствии с нашей же технологией рассуждений необходимо убрать две неопределенности. Во-первых, надо пояснить, что такое алгоритмически конечная задача, и, во-вторых, возможно, не все знают, что такое цикл Дейкстры.

Цикл Дейкстры

Э. Дейкстра в теории систематического вывода императивных программ ввел многоветочную форму цикла `WHILE`, представляющую собой описание циклического процесса, управляемого несколькими условными операторами. В языке Компонентный Паскаль цикл Дейкстры можно реализовать безусловным циклом `LOOP`.

Пример:

```
LOOP
   IF условие THEN EXIT
   {ELSIF условие THEN группа операторов}
   ……..
   END;
END;
```

Фигурные скобки, ограничивающие условный оператор, означают возможность многократного повторения этой части. Цикл Дейкстры позволяет исключить необходимость вложенных циклов.

Таблица 1.1. Представление циклом Дейкстры вложенного цикла

Структура с вложенными циклами	Идентичная структура с циклом Дейкстры

`FOR x:=1 TO N DO`
`  FOR y:=1 TO M DO`
`  Группа операторов`
`  оператор y:=y+1;`
`  END;`
`END;`
	`x:=1;y:=1;`
	`LOOP`
	`  IF x>N THEN EXIT`
	`  ELSIF y>M THEN`
	`  x:=x+1; y:=1;`
	`  ELSE`
	`  Группа операторов`
	`  оператор y:=y+1;`
	`  END;`
	`END;`

Указанное свойство цикла Дейкстры позволяет утверждать, что имеет место следующая гипотеза: один процесс (даже сложный) равнозначен одному циклу Дейкстры.

Алгоритмически конечная задача

Назовем таковой задачу, для которой можно точно описать исходный набор данных и конечный результат, также в виде некоторого набора данных. Заметим сразу, что системы управления чем-либо, системы обработки баз данных под данное определение не подходят, там важен процесс, начало которого, по большому счету исходными данными не определяется, а скорее важна временная точка, в которой программа начала работать. Нет и завершения, есть только некоторые промежуточные состояния.

Под наше определение попадают расчетные задачи, наверное, любого характера, а также задачи, цель которых – ответить на вопрос, справедливо некоторое утверждение или нет.

То есть алгоритмически чистая задача реализуется процессом, имеющим начало и завершение, процесс инициируется некоторым набо-

ром данных и завершается некоторым набором. Любой процесс, как известно, состоит из ряда итераций, возможно имеющих сложную структуру, но выше мы выяснили, что сложная структура операций сводится к одному циклу Дейкстры, что, собственно, и требовалось показать.

Конечно, данный текст нельзя воспринимать как строгое доказательство, но такой цели и не ставилось, здесь изложена не более чем гипотеза, надеюсь, что правдоподобная.

Интересный пример

Конечно, очень сложно представить реальную задачу, укладывающуюся в какой-либо точно сформулированный принцип, но тем не менее хотелось бы показать более-менее содержательный пример, демонстрирующий сказанное. Такие примеры, без сомнения, есть, и ниже рассмотрим один из них. Это одна из моих любимых задач. Относительно простая, но с изюминкой, кроме того, представляющая широкий класс совершенно реальной проблемы построения фракталов. Попробуем написать программу, рисующую графический объект, изображенный на картинке ниже:

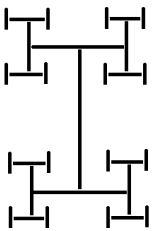

Рис. 1.1. Фрактальная снежинка

Простое наблюдение говорит о том, что здесь имеет место рекурсивный процесс, в ходе которого каждый нарисованный отрезок порождает еще два. Начинается процесс с некоторого исходного отрезка, на нашем рисунке с вертикальной линии определенной длины. Сказанное наводит на мысль о простейшем построении программы:

- рисуем отрезок;
- вызываем рекурсивную процедуру для нового отрезка;
- вызываем рекурсивную процедуру для нового отрезка.

Два совершенно одинаковых вызова – это не описка, а отражение того факта, что каждый отрезок порождает два, а отличий нет по при-

чине отсутствия какой-либо информации, позволяющей нам отделить один вызов от другого. Немного упростим запись:

```
Отрезок(x1,y1,x2,y2);
Рекурсия();
Рекурсия();
```

Два вызова обязаны отличаться списком передаваемых в них параметров, этот список и есть неопределенность, с которой сейчас начнем бороться. Проведя ряд рассуждений, построим алгоритм, решая на каждом шаге очередную небольшую проблему.

Рассуждение первое:

Очередной отрезок строится от конца предыдущего. Поэтому, даже не зная, как он будет строиться, мы должны прийдти к заключению, что информация о координатах конца необходима. И получаем следующее уточнение:

```
Отрезок(x1,y1,x2,y2);
Рекурсия(x1,y1);
Рекурсия(x2,y2);
```

Рассуждение второе:

Очередной отрезок имеет длину меньшую, чем предыдущий. Это наводит на мысль, что необходимо передать длину предыдущего (иначе не вычислить меньший):

```
Отрезок(x1,y1,x2,y2);
Рекурсия(x1,y1,y2-y1);
Рекурсия(x2,y2,y2-y1);
```

Рассуждение третье:

Еще мы знаем, что отрезки на каждом шаге меняют свою ориентацию. Вертикальный отрезок порождает два горизонтальных, горизонтальный порождает два вертикальных. Весьма существенная информация для рекурсивной процедуры, рисующей отрезки, и, видимо, эту информацию необходимо передать. Договоримся, что единица будет означать вертикальную ориентацию, а двойка – горизонтальную, и получим очередное уточнение:

```
Отрезок(x1,y1,x2,y2);
Рекурсия(x1,y1,y2-y1,1);
Рекурсия(x2,y2,y2-y1,1);
```

Рассуждение четвертое:

Похоже, что информация о собственно отрезках исчерпана, но так как мы строим рекурсивную процедуру, то необходимо обдумать свойства рекурсивного процесса, хотя бы самые общие. Хотя бы то, что процесс должен иметь начало и завершение. Это справедливо для любого вычислительного процесса, но рекурсивный устроен достаточно нетривиально, и если его начало очевидно, то о завершении зачастую следует позаботиться.

В общем случае в теле процедуры должен отрабатываться какой-то критерий, позволяющий принимать решение о возможности нового вызова. Мы пока не имеем желания думать о механизме принятия этого решения, минимальная цель анализа – выяснить, какая необходима информация о рекурсивном процессе, для принятия такого решения. Самое простое – это строить критерий на какой-то величине, изменяемой от вызова к вызову. Таких величин две. Это номер вызова и длина отрезка. Таким образом, вызовы можно прекращать, когда номер вызова станет больше некоторого критического значения или длина отрезка станет меньше некоторого критического значения. Если договоримся, что критерий будет использовать номер вызова, то последнее уточнение приобретет следующий вид:

```
Отрезок(x1,y1,x2,y2);
Рекурсия(x1,y1,y2-y1,1,0);
Рекурсия(x2,y2,y2-y1,1,0);
```

Это стартовый код, который будет по-разному реализован в разных языках, без изменения смысла. Все содержательные проблемы здесь решены. Обратите внимание, что законченный фрагмент текста получен без формулировки какой-либо законченной идеи решения. Мы просто на каждом шаге формулировали и устраняли очередную неопределенность. И сейчас так же попробуем построить и процедуру «Рекурсия».

Рассуждение первое:

Очевидно, что её изначальный вид таков:

```
Рекурсия(x,y, длина, тип, номер)
      Начало
      Конец
```

А далее необходимо развернуть информацию, заключенную в передаваемых данных. Начнем с последнего, с номера вызова. Эта величина говорит о том, что некие операции (пока не знаем, какие) будут